Analyse réaliste d'algorithmes standards

Nicolas Auger

Université Paris-Est de Marne-La-Vallée

THÈSE

Pour obtenir le grade de

Docteur de l'Université Paris-Est de Marne-La-Vallée

Spécialité : **Informatique**

préparée au **Laboratoire d'Informatique Gaspard Monge**

dans le cadre de l'École Doctorale **MSTIC**

Présentée et soutenue publiquement par

Nicolas Auger

le jeudi 20 décembre 2018

Analyse réaliste d'algorithmes standard

Directeur de thèse : **Cyril Nicaud**

Directrice de thèse : **Carine Pivoteau**

Jury

M. Loïck Lhote	Rapporteur
M. Vlady Ravelomanana	Rapporteur
Mme Frédérique Bassino	Examinatrice
M. Antoine Génitrini	Examinateur
M. Cyril Nicaud	Directeur de thèse
Mme Carine Pivoteau	Directrice de thèse

À l'origine de cette thèse, nous nous sommes intéressés à l'algorithme de tri TimSort qui est apparu en 2002, alors que la littérature sur le problème du tri était déjà bien dense. Bien qu'il soit utilisé dans de nombreux langages de programmation, les performances de cet algorithme n'avaient jamais été formellement analysées avant nos travaux. L'étude fine de TimSort nous a conduit à enrichir nos modèles théoriques, en y incorporant des caractéristiques modernes de l'architecture des ordinateurs.

Nous avons, en particulier, étudié le mécanisme de prédiction de branchement. Grâce à cette analyse théorique, nous avons pu proposer des modifications de certains algorithmes élémentaires (comme l'exponentiation rapide ou la dichotomie) qui utilisent ce principe à leur avantage, améliorant significativement leurs performances lorsqu'ils sont exécutés sur des machines récentes.

Enfin, même s'il est courant dans le cadre de l'analyse en moyenne de considérer que les entrées sont uniformément distribuées, cela ne semble pas toujours refléter les distributions auxquelles nous sommes confrontés dans la réalité. Ainsi, une des raisons du choix d'implanter TimSort dans des bibliothèques standard de Java et Python est probablement sa capacité à s'adapter à des entrées partiellement triées. Nous proposons, pour conclure cette thèse, un modèle mathématique de distribution non-uniforme sur les permutations qui favorise l'apparition d'entrées partiellement triées, et nous en donnons une analyse probabiliste détaillée.

At first, we were interested in TimSort, a sorting algorithm which was designed in 2002, at a time where it was hard to imagine new results on sorting. Although it is used in many programming languages, the efficiency of this algorithm has not been studied formally before our work. The fine-grain study of TimSort lead us to take into account, in our theoretical models, some modern features of computer architecture.

In particular, we propose a study of the mechanisms of branch prediction. This theoretical analysis allows us to design variants of some elementary algorithms (like binary search or exponentiation by squaring) that rely on this feature to achieve better performance on recent computers.

Even if uniform distributions are usually considered for the average case analysis of algorithms, it may not be the best framework for studying sorting algorithms. The choice of using TimSort in many programming languages as Java and Python is probably driven by its efficiency on almost-sorted input. To conclude this dissertation, we propose a mathematical model of non-uniform distribution on permutations, for which permutations that are almost sorted are more likely, and provide a detailed probabilistic analysis.

Table des matières

Chapitre 1

Introduction

Sommaire

À la genèse de cette thèse, nous avions pour but d'étudier l'algorithme TimSort. Il s'agit d'un algorithme de tri qui est apparu au début des années 2000 à une époque où le sujet du problème du tri avait déjà été grandement étudié depuis plus de 50 ans avec notamment la création du tri fusion par Von Neumann en 1945 ou bien encore du tri rapide par Hoare en 1961. De nos jours, il existe toujours des résultats sur le problème du tri, mais ce sujet a surtout connu un essor dans le monde de la recherche durant la double décénie de 1960 à 1980. Tim Peters qui a conçu TimSort travaillait à l'époque sur le langage de programmation Python et cherchait un algorithme de tri plus performant dans le but de remplacer celui qui était jusqu'alors utilisé par défaut dans la librairie standard, à savoir SampleSort qui est une des variantes du tri rapide. Avec une littérature si riche sur le sujet, il est alors surprenant de voir que la proposition de modifier SampleSort par TimSort a été acceptée par la communauté des développeurs du langage Python. Dans le but de comprendre ce choix, nous devons alors regarder comment TimSort fonctionne pour ensuite pouvoir analyser sa complexité. Il était annoncé quasiment partout, dont wikipedia, que sa complexité dans le pire cas était en $\mathcal{O}(n \log n)$ comparaisons, où n est le nombre d'éléments de la séquence à trier. Nous avons cependant découvert au commencement de notre étude, qu'il n'existait aucune preuve de cette complexité. Nous pensons que cela est dû au fait que la complexité a été annoncée dans une note d'explication de l'algorithme par son concepteur [37] et que la preuve semblait triviale. Pour mieux comprendre le problème, nous devons entrer dans le détail du fonctionnement de l'algorithme. TimSort, qui est un algorithme de type tri fusion, utilise une pile où sont stockés des segments triés de la séquence d'entrée avant de les fusionner. Dans les explications de l'algorithme par Tim Peters, ce dernier donne un ensemble de règles locales sur la tête de pile, pour effectuer les fusions des éléments de cette pile. Nous pensons que c'est probablement à cause de la simplicité de ces règles et du fait que nous pouvons facilement prouver que la taille de cette pile est petite, en $\mathcal{O}(\log n)$, que la preuve de cette complexité a été considérée comme triviale. Cependant, ces règles entraînent une dynamique qui rend difficile l'analyse. Nous sommes parvenus à démontrer cette complexité en exploitant diverses propriétés sur la pile et en utilisant un raisonnement d'analyse amortie sur les opérations de la pile. Cette démonstration remet donc en question la trivialité de la preuve.

Bien que nous connaissons la complexité de TimSort dans le pire cas, cette dernière n'est pas suffisante pour justifier les raisons pour lesquelles ce dernier a été implanté dans les librairies standards de plusieurs langages. En effet, le tri fusion est a priori aussi bien que TimSort et il nous faut en savoir plus à son sujet pour comprendre ce qui a poussé les développeurs à faire ce choix.

Oracle qui est propriétaire du langage Java, a également pris la décision d'implanter TimSort comme tri par défaut pour une collection de types primitifs. Pour avoir pris cette décision, nous pensons qu'ils ont fait tourner en interne une série de benchmarks dans le but de mesurer les performances de TimSort sur des cas réels. Nous pouvons donc penser que TimSort se comporte très bien lorsqu'il est utilisé sur des applications réelles provenant du domaine industriel par exemple. Nous pensons que cela est dû au fait que les séquences à trier provenant de ces applications sont déjà partiellement triées. Or, il se trouve que TimSort appartient à la catégorie des tris adaptatifs, qui ont la particularité d'être plus efficaces lorsque la séquence d'entrée est presque triée. Si nous désirons avoir une approche réaliste pour analyser un algorithme adaptatif, et sous

l'hypothèse que TimSort a été implanté dans la librairie standard Java après l'utilisation de benchmarks, nous pouvons penser qu'il est raisonnable de considérer que, pour la plupart des utilisations réelles de TimSort, la distribution sur l'ordre des séquences d'entrées n'est pas uniforme et favorise les séquences partiellement triées. Dans le cadre de l'analyse dans le cas moyen, nous considérons souvent par défaut que les entrées sont sous une distribution uniforme. Dans le cas de l'analyse du tri rapide randomisée, la distribution sur l'entrée n'est pas un problème, puisque le choix d'un pivot aléatoire qui est équivalent à faire un mélange uniforme de la séquence. Cependant, lorsque nous ne faisons aucun mélange uniforme, et si nous voulons être proche des utilisations pratiques d'un algorithme, il peut être intéressant de considérer des distributions non-uniformes. C'est pour cette raison que nous avons étudié le modèle d'Ewens qui est un cas de distribution non-uniforme sur les permutations. Cette distribution influence l'apparition de permutations en fonction du nombre de leurs cycles. Remarquons que nous pouvons utiliser les permutations pour représenter l'ordre d'une séquence d'entrée à trier. Si par exemple nous notons une séquence $S = \langle s_1, \ldots, s_n \rangle$ de taille $n \in \mathbb{N}$, il existe alors une permutation $\sigma \in \mathfrak{S}_n$ de taille n telle que la séquence $S' = \left\langle s_{\sigma^{-1}(1)}, \ldots, s_{\sigma^{-1}(n)} \right\rangle$ est la séquence obtenue une fois S triée. Nous nous intéressons à la notion d'ordre partielle dans la séquence, et la statistique du nombre cycles nous semble peu pertinente pour en faire la mesure. De plus, il existe de nombreuses statistiques qui peuvent servir à mesurer le désordre dans une permutation. Pour ces raisons, nous nous sommes alors intéressés à une généralisation de la distribution d'Ewens pour d'autres statistiques sur les permutations, dont en particulier le cas du nombre de records qui nous semble être un choix de statistique plus naturel que le nombre de cycles.

Une autre raison qui aurait poussé les développeurs à choisir d'implanter TimSort serait qu'il est relativement bien optimisé pour fonctionner sur nos machines actuelles. L'algorithme utilise certaines heuristiques qui fonctionnent particulièrement bien sur les architectures modernes. Nous avons évoqué précédemment l'utilisation de règles locales de fusions des segments stockés dans la pile. Dans ses explications, Tim Peters affirme avoir choisi ces règles dans le but d'exploiter efficacement la mémoire cache. Cette mémoire est plus proche, du point de vue électronique, du processeur et permet d'accéder plus rapidement aux données, l'exploiter correctement permet donc de gagner en performance de temps. Lorsque nous faisons l'analyse d'algorithmes, de manière générale nous considérons un modèle sous-jacent de machine. Ce modèle, appelé RAM pour Random Access Machine, est en général un assez bon modèle pour comparer des algorithmes et deviner lequel sera le plus efficace en pratique. Ce modèle ne prend cependant pas en compte certaines caractéristiques d'architecture de nos machines actuelles comme c'est le cas par exemple des caches dans la mémoire ou bien encore de la prédiction de branchement. Si nous analysons TimSort sans prendre en compte ces caractéristiques, nous aurons du mal à comprendre les raisons pour lesquelles il est si efficace en pratique. Il nous faut donc proposer une extension du modèle RAM qui permettrait de prendre en compte ces caractéristiques. Il existe déjà une littérature assez riche concernant l'extension de ce modèle aux caches dans la mémoire, mais relativement peu pour le cas de la prédiction de branchement. Nous avons donc décidé de nous intéresser plus particulièrement à cette dernière caractéristique. Lorsque nous exécutons un programme sur un processeur récent, nous avons une suite d'instructions à exécuter dans un certain ordre. Il se trouve que nos processeurs actuels font une

parallélisation en décomposant ces instructions en micro-instructions élémentaires. Il est alors possible d'exécuter plusieurs micro-instructions de différentes instructions consécutives en même temps. Il y a cependant des obstacles à ce modèle, et l'un d'entre eux concerne les instructions de branchement (comme par exemple un *if*). Lorsqu'une instruction fait un branchement, la prochaine instruction dépend du résultat de la première instruction. Il est alors impossible de savoir l'instruction que nous pouvons commencer à réaliser. Pour palier à ce problème, les processeurs utilisent des prédicteurs de branchement qui vont tenter d'anticiper le résultat de l'instruction de branchement et commencer à exécuter les instructions suivantes. Bien entendu, si le processeur se trompe, il doit alors tout annuler et recommencer à partir de la bonne instruction à exécuter. Quand le prédicteur se trompe, on dit que nous avons eu une erreur de prédiction. Une conséquence immédiate est alors qu'un programme qui fait peu d'erreurs de prédictions a des chances d'être rapide. La littérature au sujet de la prédiction de branchement contient principalement des résultats expérimentaux obtenus à l'aide de benchmarks. Il existe également des études plus théoriques sur le sujet, dont notamment une analyse qui démontre que la prédiction de branchement n'influence quasiment pas le tri rapide. Ce résultat, bien qu'étant sur un autre algorithme de tri, ainsi que la difficulté à analyser la complexité de TimSort à cause de la dynamique engendrée par les stratégies locales de fusions ne nous ont pas encouragé à tenter de faire une analyse de l'influence de la prédiction de branchement sur ses performances. Cependant, à l'instar de TimSort qui utilise ces règles locales pour essayer d'exploiter au mieux la mémoire cache, nous pouvons essayer de concevoir des algorithmes qui exploiteraient la prédiction de branchement. Il existe déjà une première étude en ce sens, pour laquelle l'influence de la prédiction de branchement a été analysée pour des algorithmes s'appuyant sur des arbres binaires de recherches déséquilibrés. Les bons résultats de cette étude nous ont donné une motivation supplémentaire pour capitaliser nos connaissances sur la prédiction de branchement dans la conception d'algorithmes. Nous proposons ainsi des variantes légères d'algorithmes élémentaires connus qui réduisent le nombre moyen d'erreurs de prédictions et qui par conséquence sont plus performantes une fois implantées et exécutées sur nos machines actuelles.

Cette thèse est ainsi constituée de trois grands axes et nous allons détailler chacun d'entre eux avec les diverses références bibliographiques dans le domaine.

1.1 Plan détaillé

1.1.1 Tri adaptatif et TimSort

La première partie est orientée sur l'algorithme TimSort de Tim Peters. Dans un premier temps nous y définirons ce qu'est le problème du tri et ce qu'est un algorithme de tri. Nous verrons qu'il existe une borne inférieure connue pour le nombre de comparaisons pour les algorithmes qui n'exploitent aucune connaissance sur la structure de la séquence d'entrée. Nous verrons cependant qu'il est possible, tout en respectant cette borne, d'avoir des algorithmes de tri qui sont particulièrement efficaces pour trier une séquence qui est déjà partiellement triée. Ces derniers sont appelés des algorithmes de tris adaptatifs, car ils s'adaptent au désordre initial dans la séquence à trier. Dans le contexte de

l'étude des algorithmes de tri adaptatifs, nous verrons qu'il existe des notions d'optimalité dont le formalisme provient en grande partie des travaux de Mannila [33]. Nous verrons des exemples de tris adaptatifs, comme par exemple le tri fusion naturel de Knuth [30]. L'algorithme de tri fusion naturel fonctionne en deux étapes ; premièrement, il détermine les runs de la séquence d'entrée qui sont les sous-séquences contiguës croissantes et de tailles maximales ; deuxièmement, il fusionne ces runs deux à deux pour créer de nouveaux runs plus grands, cette étape est alors répétée jusqu'à ce qu'il ne reste plus qu'un run correspondant à la séquence d'entrée triée. Si la séquence d'entrée à trier est de taille n et qu'elle est composée de r runs, il est facile de montrer que le nombre de comparaisons effectuées pour trier cette séquence est $\mathcal{O}(n \log r)$, qui est optimal au sens de Mannila. Le fonctionnement de TimSort est proche du fonctionnement du tri fusion naturel de par sa décomposition de la séquence d'entrée en runs. La gestion de ces runs se fait cependant de manière différente. TimSort ne considère pas uniquement les runs croissants mais également les runs décroissants. Les étapes de fusion et d'identification des runs se font simultanément. Lorsqu'un run est identifié ce dernier est ajouté à une pile de runs à fusionner. Dans cette pile règne différentes règles locales qui régissent le comportement du haut de la pile. Si une règle n'est pas respectée lorsqu'un nouveau run est ajouté dans la pile, il est alors possible qu'une série de fusions soient effectuées pour rétablir le respect de ces règles. La dynamique entrainée par ces règles rend l'analyse de TimSort difficile. Il n'est alors pas simple de démontrer que sa complexité dans le pire cas est en $\mathcal{O}(n \log n)$ comme l'a annoncé son concepteur. Dans ce chapitre, nous reprenons les résultats que nous avons pré-publiés dans cet article [9]. Ce chapitre va donc donner une explication des principes de fonctionnement de TimSort, ainsi qu'une preuve de la complexité annoncée dans le pire cas.

1.1.2 Prédiction de branchement

La deuxième partie traite de l'exploitation de la prédiction de branchement. Dans cette partie, nous remettons en question le modèle RAM qui ne prend pas en compte de caractéristiques des architectures actuelles. Nous tentons ainsi de l'affiner en y ajoutant une analyse dans le cas moyen du nombre d'erreurs de prédictions. Un algorithme qui fait peu d'erreurs de prédictions peut avoir des chances d'être plus performant une fois implanté sur des machines suffisamment récentes. Il existe à ce jour de nombreux travaux expérimentaux dans ce domaine. Ces travaux utilisent des compteurs spécialement installés sur les processeurs pour compter divers évènements dont en particulier le cas où une erreur de prédiction se produit. La librairie PAPI utilise ces compteurs pour ainsi comptabiliser un ensemble d'évènements d'intérêts [2]. Biggar et ses coauteurs ont, en utilisant cette librairie, mis en avant l'influence de la prédiction de branchement pour divers algorithmes de tri [11].

Des résultats théoriques ont également été publiés, comme par exemple l'article de Brodal, Fagerberg et Moruz qui ont cherché à étudier des compromis entre le nombre de comparaisons et le nombre d'erreurs de prédictions dans le cadre de l'analyse d'algorithmes de tri [15]. Ces mêmes auteurs ont également publié, à notre connaissance, la première analyse théorique en considérant un prédicteur statique, qui prédit toujours la même issue, pour analyser l'influence du nombre d'inversions sur le nombre d'erreurs de prédictions à l'exécution du tri rapide [14]. L'analyse sur des prédicteurs statiques reste cependant un

cas simple qui ne prend pas en compte les dépendances entre différentes exécutions d'une même instruction de branchement. Une analyse considérant des prédicteurs dynamiques a également été proposée par Kaligosi et Sanders pour analyser l'algorithme du tri rapide en utilisant des chaînes de Markov [28]. Il s'agit d'une technique que nous utilisons également par la suite et qui est donc détaillée dans ce manuscrit. De la même manière, Martinez, Nebel et Wild ont montré que l'influence de la prédiction de branchement n'est pas suffisante pour expliquer les bonnes performances en pratique du tri rapide à deux pivots [34].

D'autres études concernent la conception d'algorithmes où les auteurs proposent des solutions pour contourner la prédiction de branchement. C'est par exemple le cas de Sanders et Winkel qui propose une variante de l'algorithme de tri SampleSort qui utilise des instructions particulières du processeur semblables à des *CMOV* [40]. Dans le même style, nous avons les travaux de Elmasry, Katajainen et Stenmark qui proposent une variante du tri fusion à l'aide d'instructions spéciales du même genre [22].

Il existe cependant des travaux qui tentent au contraire de se servir de la prédiction de branchement pour améliorer les performances à l'exécution des algorithmes. C'est par exemple le cas des travaux de Brodal et Moruz qui ont fait une étude expérimentale sur des arbres binaires de recherches déséquilibrés, et mis en avant que les algorithmes de parcours sur ces arbres avaient un bon comportement en pratique. C'est sur cette idée d'exploiter la prédiction que repose les travaux que nous exposons dans ce manuscrit.

Ce que nous proposons a été publié dans cet article [10]. Nous cherchons à modifier directement des algorithmes élémentaires pour leurs permettre d'exploiter la prédiction de branchement. Notre espoir est de rendre ces algorithmes plus performants avec ces modifications, lorsque ces derniers sont exécutés sur des machines suffisamment récentes. Nous montrons une première analyse expérimentale et théorique sur deux algorithmes connus de recherche dans une séquence de taille n du minimum et du maximum. Le premier algorithme naïf a besoin de $2n$ comparaisons pour trouver la solution tandis que le deuxième a besoin de $1.5n$ comparaisons. On peut montrer par un argument d'adversaire que le deuxième est optimal en nombre de comparaisons. Du point de vue de la prédiction de branchement, c'est cependant l'algorithme naïf qui est le meilleur avec de l'ordre de $\Theta(\log n)$ erreurs de prédictions contre $\Theta(n)$ erreurs de prédiction pour le second. Il se trouve que lorsque nous implantons ces deux algorithmes sur une machine récente, nous nous apercevons que c'est alors l'algorithme naïf qui est le plus rapide lorsque la séquence d'entrée est générée selon une distribution uniforme. Cet exemple illustre qu'il est alors possible d'avoir des compromis entre le coût des opérations de bases (ici les comparaisons) et le nombre d'erreurs de prédictions. Ce résultat est donc encourageant pour que nous proposons d'autres variantes à d'autres algorithmes. C'est ce que nous avons fait avec les algorithmes de l'exponentiation rapide et de la recherche dichotomique qui font légèrement plus de comparaisons après modification, mais qui sont bien plus prévisibles pour le prédicteur. Enfin nous verrons des pistes qui proposent de généraliser les modifications que nous avons effectuées pour ces deux algorithmes.

1.1.3 Distribution généralisée d'Ewens

La troisième partie concerne l'étude de permutations non-uniformes et nous nous attarderons sur une distribution en particulier. Lorsque nous faisons de l'analyse dans le cas moyen, nous considérons souvent par défaut que l'entrée est distribuée uniformément. Comme nous l'avons dit précédemment, pour le cas des algorithmes de tri, mais sûrement dans beaucoup d'autres, il est peu probable que la distribution de l'entrée soit uniforme lorsque ces algorithmes sont utilisés pour résoudre des problèmes issus de données obtenues dans la réalité. En particulier, nous pensons que ces données sont bien souvent partiellement triées, ce qui pourrait expliquer les performances de TimSort en pratique, et l'une des raisons pour lesquelles ce dernier est implanté dans les librairies standards de divers langages de programmations réputés. La motivation de cette partie était donc d'offrir un cadre pour étudier des distributions qui favorisent ce genre de séquences d'entrées. Comme nous nous intéressons à l'analyse des algorithmes de tri, nous avons donc étudié des distributions non-uniformes sur les permutations. Ces dernières peuvent en effet être utilisées pour représenter l'ordre dans la séquence. La permutation identité est ainsi le cas particulier où la séquence d'entrée est triée, notre but est donc d'avoir des distributions qui permettraient d'avoir avec grande probabilité des permutations qui sont proches de cette dernière. La distribution d'Ewens est un exemple de distribution non-uniforme sur les permutations qui connait une littérature riche de par les contributions apportées par la communauté des probabilistes. À l'origine Ewens travaillait sur une généralisation de cette distribution dans le but d'étudier des problèmes de statistiques [23]. Le cas particulier qui a principalement été étudié par les probabilistes est celui où la permutation a un poids dépendant d'un paramètre réel $\theta > 0$ de façon à avoir pour $\sigma \in \mathfrak{S}_n$, $w_{\theta,n}(\sigma) = \theta^{\mathbf{cyc}(\sigma)}$. Lorsque $\theta > 1$, la distribution d'Ewens va ainsi favoriser l'apparition de permutations qui ont un grand nombre de cycles. Dans cette partie, nous reprenons les travaux que nous avons publiés pour la conférence AofA de 2016 [7]. Dans ces travaux nous proposons une généralisation naturelle de la distribution d'Ewens pour toute statistique sur les permutations. Comme nous le démontrerons par la suite, le nombre de records dans une permutation peut être utilisé pour définir une mesure d'ordre partiel qui permet de mesurer le désordre dans une permutation. Pour cette raison, et comme cette statistique est proche de celle des cycles grâce à la bijection fondamentale, nous étudierons en particulier la distribution d'Ewens généralisée au nombre de records. Nous verrons en particulier son influence sur d'autres statistiques et leurs comportements asymptotiques lorsque nous faisons varier le paramètre réel θ. Sous cette distribution, nous ferons également quelques analyses d'algorithmes dans le cas moyen, comme par exemple le tri par insertion.

Cette thèse propose donc un cadre pour l'analyse d'algorithmes dans un contexte qui se veut plus proche des applications provenant de la réalité. Nous proposons donc pour résumer d'étudier des distributions non-uniformes sur les permutations ainsi que l'exploitation de la caractéristique de prédiction de branchement de nos processeurs actuels afin de permettre à des algorithmes standards de devenir plus efficaces sur nos machines. L'algorithme TimSort joue le rôle de pont entre ces deux sujets puisque ce dernier est à la fois conçu pour être particulièrement efficace sur des distributions non-uniforme sur l'ordre des séquences d'entrées, mais aussi parce qu'il est également optimisé pour l'archi-

tecture de nos machines modernes.

Cette thèse propose une première piste d'exploration dans plusieurs directions possibles pour faire de l'analyse d'algorithmes plus réaliste, que ce soit du point de vue des architectures utilisées par les machines modernes ou par les jeux de données utilisés dans les applications pratiques de ces derniers. Les travaux fournis se sont déjà avérés riches en surprises et en résultats, et ouvrent la porte à de nombreux développements.

Chapitre 2

Notions de mathématiques

Sommaire

Dans ce chapitre, nous introduisons les notions mathématiques qui nous seront nécessaires au cours de cette thèse. La première partie introduit le groupe de symétrie sur un ensemble de taille n dont les éléments sont appelés des permutations. Nous verrons en particulier des définitions de statistiques sur ces permutations ainsi que de bijections qui préservent ces dernières. La deuxième partie concerne la théorie des probabilités. Nous nous intéresserons en particulier aux chaînes de Markov dont nous aurons besoin dans le cadre de l'analyse dans le cas moyen dans le contexte de la prédiction de branchements.

2.1 Les permutations

Soit un ensemble fini X à n éléments, nous pouvons attribuer un numéro à chaque élément de cet ensemble à l'aide d'une bijection $e : X \to [n]$. Nous nous intéressons ici au groupe de symétrie des ensembles finis, c'est à dire l'ensemble des bijections de ces ensembles sur eux mêmes. Comme il existe une bijection e pour ces ensembles finis, il n'est pas nécessaire de travailler sur l'ensemble X, nous pouvons donc nous intéresser uniquement au groupe de symétrie sur l'ensemble $[n]$. Nous notons $\mathfrak{S}_n$ le groupe de symétrie de l'ensemble $[n]$. Les éléments qui appartiennent à $\mathfrak{S}_n$ sont appelés des permutations de taille n.

2.1.1 Représentation des permutations

Il existe de nombreuses façons de représenter une permutation, et nous allons en montrer deux.

La première est l'utilisation typique issue de l'algèbre sur les permutations. Comme les permutations appartiennent à un groupe fini, il est possible de représenter ces dernières comme un ensemble de cycles. Un *cycle* dans la permutation $\sigma \in \mathfrak{S}_n$ de représentant $i \in [n]$ est défini par la séquence $(i, \sigma(i), \sigma(\sigma(i)), \ldots, \sigma^{-1}(i))$. Nous pouvons donc ainsi représenter une permutation comme un ensemble de tous ses cycles. Par exemple si $\sigma = \{(1,2),(3)\}$, alors nous avons la permutation $\sigma(1) = 2$, $\sigma(2) = 1$ et $\sigma(3) = 3$. Pour ce cas, nous avons un cycle de taille 1 ce qui nous donne un *point fixe*.

La deuxième représentation provient d'un contexte que nous verrons plus souvent dans ce manuscrit. Il s'agit de représenter la permutation sous la forme d'un mot. Nous nous en servons particulièrement pour le cas du problème du tri, où cette représentation est utilisée pour encoder l'ordre dans la séquence d'entrée. Supposons que nous ayons une séquence ordonnée que nous voulons trier $S = (s_1, \ldots, s_n)$. Il existe alors une permutation $\sigma \in \mathfrak{S}_n$ telle que $S' = (s_{\sigma^{-1}(1)}, \ldots, s_{\sigma^{-1}(n)})$ est la séquence triée des éléments de S. Pour tout i le rang de s_i est ainsi donné par $r_i = \sigma(i)$. La séquence $(r_1, \ldots, r_n)$ est alors une autre façon de définir σ. Nous verrons par la suite qu'il existe des algorithmes de tri qui n'utilisent que la comparaison entre les éléments pour trier une telle séquence S. Pour ces algorithmes, il n'y a alors aucune différence, en terme de coût, à trier S ou à trier la séquence $(r_1, \ldots, r_n)$. Nous écrivons de cette façon $\sigma = (r_1, \ldots, r_n)$ sa représentation sous forme de mot et qui définie un ordre sur la séquence S. Si nous reprenons, notre exemple précédent, nous avons que la permutation $\sigma = (2,1,3) = \{(1,2)(3)\}$. Par la suite, nous omettrons souvent les parenthèses et les virgules, lorsqu'il n'y a aucune ambiguïté, nous aurons ainsi sur notre exemple $\sigma = 213 = (21)(3)$.

2.1.2 Statistiques sur les permutations

Dans ce qui suit, nous allons introduire des statistiques et des notations sur les permutations qui auront un intérêt particulier au chapitre 6 sur la génération de permutations non-uniformes.

Cycles : Nous l'avons vu précédemment, nous pouvons écrire une permutation sous la forme d'une décomposition en cycles. Il se trouve que le nombre de cycles est une statistique d'intérêt et est même à la base de la distribution d'Ewens qui sera étudiée au chapitre 6. Nous dénotons par $\mathbf{cyc}(\sigma)$ le nombre de cycles d'une permutation $\sigma \in \mathfrak{S}_n$. Nous dénotons de plus par $\mathbf{cyc}_{\geq k}(\sigma)$ le nombre de cycles dont la taille est supérieure ou égale à k. Finalement nous utilisons $\mathbf{cyc}_k(\sigma)$ pour le nombre de cycles dont la taille est égale à k. En particulier $\mathbf{cyc}_1(\sigma)$ compte le nombre de points fixes de σ.

Records : Les records sont de deux types possibles. Un min-record est un élément i d'une permutation σ tel que dans sa représentation en mot $\sigma(i)$ est plus petit que tout élément qui le précède, soit $\forall j \in [i-1]$ on a $\sigma(i) < \sigma(j)$. Respectivement, un max-record est défini de la même façon pour la relation $>$. Sur tout $\mathfrak{S}_n$, ces statistiques avec le nombre de cycles ont les mêmes cardinalités et nous verrons par la suite des bijections pour le prouver. Dans la permutation $\sigma = 46527381$ nous comptons 4 max-records et 3 min-records.

Descentes et montées : Une permutation σ admet une descente en position i si l'on a la relation $\sigma(i) > \sigma(i+1)$. Respectivement, une montée en position i signifie que l'on a la relation $\sigma(i) < \sigma(i+1)$. Par exemple, dans la permutation $\sigma = 46527381$ nous comptons 3 montées et 4 descentes. Le nombre de montées et le nombre de descentes est une statistique bien connue en combinatoire sur les permutations.

2.1.3 Relations entre les statistiques

Nous l'avons évoqué précédemment, il existe des relations entre certaines de ces statistiques, et nous allons le démontrer à l'aide de bijections.

La bijection fondamentale : La première bijection que nous allons voir est la bijection fondamentale (voir [13, p. 109-110] pour plus de détails et références). Cette bijection permet de montrer qu'à partir d'une permutation de taille n et de k cycles, il est possible de construire une autre permutation de taille n et de k min-records. La bijection fondamentale procède en écrivant la permutation dans sa décomposition en cycles. Pour chacun de ces cycles, nous faisons en sorte de prendre la représentation avec le plus grand élément en première position. Enfin, nous trions les cycles dans l'ordre croissant par les valeurs prisent par leurs premiers éléments, puis nous retirons les parenthèses pour obtenir la forme en mot de la permutation image.

Exemple. *Prenons la permutation* $\sigma = 926857341$. *Sa représentation en cycle est* $\sigma = (367)(19)(5)(48)(2)$. *Nous pouvons la réarranger en faisant en sorte que le plus grand élément de chaque cycle apparaît en premier ce qui donne* $\sigma = (736)(91)(5)(84)(2)$. *On tri ensuite dans l'ordre croissant de ces premiers*

éléments et on obtient $\sigma = (2)(5)(736)(84)(91)$. *En retirant les parenthèses nous obtenons l'image de* σ *par la bijection fondamentale soit* $F(\sigma) = 257368491$. *On voit bien ici, que chaque élément de début de cycle de* σ *est alors un record dans* $F(\sigma)$ *et qu'il y a donc conservation entre ces deux statistiques. De plus, il est facile de voir à l'aide de cette remarque comment revenir en arrière puisqu'il suffit de chercher les min-records de l'image pour retrouver la décomposition en cycles de la permutation initiale et donc de voir que cette application est bien une bijection.*

Remarque. Il est facile d'obtenir une bijection fondamentale pour les max-records en mettant le plus grand élément de chaque cycle en première position et en les triant dans l'ordre décroissant.

Bijections entre montées et descentes : Il existe deux involutions qui permettent à partir d'une permutation de transformer toutes les montées en descentes et réciproquement. La première est de représenter une permutation σ sous forme de mot et de prendre le mot miroir $\widetilde{\sigma}$ qui pour chaque $i \in [n]$ associe $\widetilde{\sigma}(i) = \sigma(n+1-i)$. Dans ce cas, une montée en position $i \in [n]$ dans $\sigma \in \mathfrak{S}_n$ devient une descente en position $n-i$ dans $\widetilde{\sigma}$. Une autre involution, notons la η, qui possède la même propriété d'inversion des descentes et des montées procède en remplaçant chaque élément $\sigma(i)$ de $\sigma \in \mathfrak{S}_n$, avec $i \in [n]$, par $n+1-\sigma(i)$. Cette application laisse invariant la position des montées devenues des descentes et réciproquement.

Exemple. *Prenons à nouveau la permutation* $\sigma = 926857341$. *Nous avons alors* $\widetilde{\sigma} = 143758629$ *et* $\eta(\sigma) = 184253769$.

2.1.4 Résultats provenant de la combinatoire

Nombres de Stirling de la première espèce non-signés : Les nombres de Stirling apparaissent dans de nombreux problèmes de combinatoire. En particulier, les nombres de Stirling de la première espèce non-signés comptent les permutations de $\mathfrak{S}_n$ composées de k cycles. Ces nombres sont notés $c(n,k)$. Il n'est alors pas étonnant de les voir apparaîtrent dans les expressions des probabilités faisant intervenir des permutations distribuées en fonction de leur nombre de cycles, comme c'est le cas de la distribution d'Ewens dont nous discuterons au cours du chapitre 6.

Remarque. Comme nous avons pu le voir précédemment, les records sont en bijection avec les cycles, et ainsi $c(n,k)$ comptent également les permutations de taille n avec k records.

Par convention, nous considérons que $c(0,0) = 1$. Par définition, comme il est impossible d'avoir des permutations de tailles non-nulles et composées de moins de 0 cycle, nous avons pour tout $n > 0$ et tout $k \leq 0$ que $c(n,k) = 0$. De la même façon, il est impossible d'avoir des permutations composées d'un plus grand nombre de cycles que son nombre d'éléments, autrement dit pour tout $n \geq 0$ et pour tout $k > n$, nous avons $c(n,k) = 0$.

Nous allons maintenant présenter des propriétés remarquables de ces nombres.

Lemme 1. *Pour tout* $n > 0$, *pour tout* $k > 0$, *nous avons la relation de récurrence*

$$c(n+1,k) = nc(n,k) + c(n,k-1)$$

qui est vérifiée.

Démonstration. La preuve se fait par construction. Pour obtenir une permutation de taille $n+1$ qui est composée de k cycles, nous pouvons partir d'une permutation de taille n et insérer l'élément $n+1$ afin d'obtenir une nouvelle permutation de taille $n+1$. Pour avoir une permutation de taille $n+1$ composée de k cycle après l'insertion de l'élément $n+1$, il y a deux cas possibles :

— Soit la permutation initiale possède déjà k cycles, dans ce cas, il suffit de rajouter l'élément $n+1$ dans un des cycles déjà existants. Il y a n positions possibles dans ce cas. Comme nous avons $c(n,k)$ permutations qui ont exactement k cycles, il y a en tout $nc(n,k)$ possibilités d'obtenir une permutation de taille $n+1$ à k cycles de cette façon.

— Soit la permutation initiale possède déjà $k-1$ cycles, dans ce cas il est nécessaire de créer un nouveau cycle constitué uniquement de $n+1$. Comme il y a en tout $c(n,k-1)$ permutations qui ont exactement $k-1$ cycles, il y a en tout $c(n,k-1)$ possibilités d'obtenir une permutation de taille $n+1$ à k cycles de cette façon.

En combinant les contributions de ces deux cas, nous obtenons le résultat annoncé. □

Lemme 2. *Pour tout $x \in \mathbb{C}$, l'identité*

$$\sum_{k=0}^{n} c(n,k)x^k = \prod_{k=0}^{n-1} x+k$$

est vérifiée.

Démonstration. Il est possible de démontrer cette relation à l'aide de la méthode symbolique et de l'analyse combinatoire . La preuve peut, par exemple, être trouvée dans le livre de Philippe Flajolet [25]. Nous ne prouverons ici ce résultat qu'avec une preuve par récurrence.

Remarquons que pour $n=0$ nous avons

$$\prod_{k=0}^{n-1} x+k = 1,$$

car il s'agit d'un produit vide. La convention $c(0,0)=1$ a été choisie pour rendre cette relation vraie pour $n=0$. Supposons que la relation est vraie pour un certain rang n. Développons alors l'expression pour $n+1$,

$$\sum_{k=0}^{n+1} c(n+1,k)x^k.$$

En appliquant le Lemme 1, nous avons que cette somme est égale à

$$\sum_{k=0}^{n+1} nc(n,k)x^k + \sum_{k=0}^{n+1} c(n,k-1)x^k.$$

Après changement de la variable de la somme de droite par $k' = k-1$ et

comme $c(n, n+1) = c(n, -1) = 0$, nous avons que cette quantité est égale à

$$\sum_{k=0}^{n} nc(n,k)x^k + \sum_{k'=0}^{n} c(n,k')x^{k'+1}.$$

Nous pouvons faire une factorisation et ainsi obtenir que cette quantité est égale à

$$(n+x)\sum_{k=0}^{n} c(n,k)x^k.$$

En appliquant l'hypothèse de récurrence sur l'expression $\sum_{k=0}^{n} c(n,k)x^k$, nous obtenons que

$$\sum_{k=0}^{n+1} c(n+1,k)x^k = (n+x)\prod_{k=0}^{n-1} x+k,$$

L'hypothèse de récurrence pour n implique que la relation est vérifiée pour $n+1$ et est donc vraie pour tout $n \in \mathbb{N}$. □

2.2 Théorie des probabilités

Dans le cadre de cette thèse, nous utilisons des éléments de la théorie des probabilités. Nous allons donc dans cette section faire quelques rappels des axiomes de cette théorie ainsi que de propriétés simples qui en découlent.

2.2.1 Espace de probabilité

Nous devons, pour commencer, définir ce qu'est un espace probabilisé, ou espace de probabilité. Soit Ω un *univers* qui est l'ensemble des états qui peuvent être observés au cours d'une expérience aléatoire. Par exemple, dans le cas d'un lancer de dé à six faces, nous pouvons avoir $\Omega = \{1, 2, 3, 4, 5, 6\}$. Pour définir ce qu'est un espace probabilisé, nous devons premièrement introduire la notion de *tribu*.

Définition 1. Soit $\mathcal{A}$ un ensemble de sous-ensembles d'un univers Ω. L'ensemble $\mathcal{A}$ est une tribu si elle vérifie les trois propriétés suivantes :

1. L'ensemble Ω appartient à $\mathcal{A}$.
2. L'ensemble $\mathcal{A}$ est stable par complémentaire. Si $E \in \mathcal{A}$ alors $E^{\complement} = \Omega \backslash E \in \mathcal{A}$.
3. L'ensemble est stable à l'intersection et à l'union dénombrable. Soit une famille $(A_i)_{i\in\mathbb{N}}$ telle que pour tout $i \in \mathbb{N}$ $A_i \in \mathcal{A}$. Nous avons alors

$$\bigcap_{i\in\mathbb{N}} A_i \in \mathcal{A},$$

et

$$\bigcup_{i\in\mathbb{N}} A_i \in \mathcal{A}.$$

Intuitivement, une mesure de probabilité peut être vue comme la fréquence d'apparition d'un événement. Si nous réalisons un grand nombre N de fois une expérience aléatoire, nous nous attendons à ce que la probabilité d'un événement soit proche du nombre de fois que ce dit événement s'est produit divisé par N. Une conséquence de cette intuition, est qu'une probabilité prend une valeur comprise dans l'intervalle $[0, 1]$.

Définition 2. Une *mesure de probabilité* est une application $\mathbb{P} : \mathcal{A} \to [0, 1]$. De plus, cette application doit vérifier les deux propriétés suivantes :

1. $\mathbb{P}(\Omega) = 1$.
2. Soit une famille $(A_i)_{i\in\mathbb{N}}$ telle que pour tout $i \in \mathbb{N}$ $A_i \in \mathcal{A}$ disjoints deux-à-deux, c'est-à-dire $A_i \cup A_j = \emptyset$ si $i \neq j$. Nous avons

$$\mathbb{P}\left(\bigcup_{i\in\mathbb{N}} A_i\right) = \sum_{i\in\mathbb{N}} \mathbb{P}(A_i). \tag{2.1}$$

Nous définissons un *espace probabilisé* par un tel triplet $(\Omega, \mathcal{A}, \mathbb{P})$. L'équation (2.1) est appelée la propriété de σ-additivité.

Définition 3. Un espace de probabilité est défini par un triplet $(\Omega, \mathcal{A}, \mathbb{P})$, où Ω est un univers, $\mathcal{A}$ une tribu dans cet univers, et $\mathbb{P}$ une mesure de probabilité qui associe à chaque événement de $\mathcal{A}$ une valeur dans l'intervalle $[0, 1]$.

Remarque. Il est naturel de choisir $\mathcal{A} = 2^\Omega$, cependant il existe des Ω et des $\mathbb{P}$, pour lesquels $\mathbb{P}$ ne peut pas être définie sur tous les ensembles de 2^Ω sans créer de contradictions. Dans le cas où Ω est un ensemble fini ou dénombrable, il est possible de créer un espace probabilisé $(\Omega, 2^\Omega, \mathbb{P})$ à la condition que, pour tout $\omega \in \Omega$, il existe $p(\omega) \in [0, 1]$ tel que $\sum_{\omega\in\Omega} p(\omega) = 1$ et en définissant pour tout $A \in 2^\Omega$, $\mathbb{P}(A) = \sum_{\omega\in A} p(\omega)$. Dans de nombreux cas, nous aurons à faire à des espaces probabilisés de ce type.

Des exemples d'espaces probabilisés

Distribution uniforme sur un ensemble fini : En utilisant ce qui a été dit lors de la remarque précédente, nous pouvons en déduire un espace trivial pour le cas où Ω est un ensemble fini. L'ensemble $\mathcal{A} = 2^\Omega$ et nous choisissons d'attribuer le même poids à chaque élément de Ω. Nous définissons ainsi, pour tout $A \in \mathcal{A}$ la mesure de probabilité uniforme $\mathbb{P}(A) = \frac{|A|}{|\Omega|}$.

Distribution uniforme sur l'intervalle $[0, 1]$: Un espace que nous verrons souvent par la suite, est la distribution uniforme pour le cas où $\Omega = [0, 1]$. Nous n'avons pas $\mathcal{A} = 2^\Omega$. $\mathcal{A}$ est construit en générant la tribu minimale à partir des intervalles ouverts $]a, b[$, avec $0 \leq a \leq b \leq 1$. Il s'agit donc de la tribu la plus petite possible qui contient tous ces intervalles. La mesure de probabilité est la *mesure de Lebesgue* υ qui pour un intervalle $]a, b[$ associe $\upsilon(]a, b[) = b - a$. En particulier, si $a = 0$, $\upsilon(]a, b[) = b$.

Nous verrons d'autres distributions par la suite, notamment au cours du Chapitre 6.

2.2.2 Variable aléatoire

Il est courant de transformer l'espace probabilisé à l'aide de variables aléatoires. Concrètement, une variable aléatoire est une application $X : \Omega \to E$ dans un espace E. Bien souvent, une variable représente une grandeur et nous choisissons $E = \mathbb{R}$. Nous pouvons transformer l'espace probabilisé pour la variable aléatoire en posant une nouvelle mesure de probabilité que nous définissons par

$$\mathbb{P}_X(x) = \mathbb{P}(X^{-1}(x)),$$

où pour tout $x \in E$, $X^{-1}(x) = \{\omega \in \Omega \mid X(\omega) = x\}$. Cette nouvelle mesure est appelée la loi de la variable X. Une autre notation courante est d'écrire $\mathbb{P}_X(x) = \mathbb{P}(X = x)$. Nous pouvons ainsi définir un nouvel espace probabilisé pour l'univers E et la mesure $\mathbb{P}_X$.

Un exemple de variable aléatoire peut être la somme de deux dés à six faces. Choisissons la distribution uniforme en prenant l'espace probabilisé où $\Omega = \{1, \dots, 6\}^2$, $\mathcal{A} = 2^\Omega$ et pour $\omega = (d_1, d_2) \in \Omega$, nous avons $\mathbb{P}(\omega) = \frac{1}{36}$. Nous posons la variable aléatoire $X(d_1, d_2) = d_1 + d_2$. Nous avons alors, par exemple, $\mathbb{P}_X(2) = \frac{1}{36}$ puisque $(1, 1) \in \Omega$ est le seul lancer qui permet d'obtenir une somme égale à 2, et nous avons aussi $\mathbb{P}_X(3) = \frac{2}{36}$ puisque $(1, 2)$ et $(2, 1)$ sont les seuls lancers possibles pour obtenir une somme égale à 3.

2.2.3 Probabilité conditionnelle et dépendance

Nous allons maintenant définir des notions de dépendances entre les événements. Soit deux événements $A, B \in \mathcal{A}$, nous voulons connaître les chances que l'événement A se réalise si l'on sait que l'événement B s'est réalisé.

Définition 4. Soit $A, B \in \mathcal{A}$ deux événements dans un espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$. La probabilité conditionnelle que A se réalise sachant que B s'est réalisé est définie par la mesure

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

Cette relation définie une nouvelle mesure de probabilité dans cette espace.

Pour mieux comprendre cette définition, nous allons réutiliser l'intuition fréquentiste, qui considère qu'une probabilité est une fréquence limite obtenue après la répétition d'un grand nombre N d'expériences aléatoires. Nous voulons déterminer le nombre de fois que l'événement B a été obtenu sur l'ensemble de ces expériences, si nous notons N_B ce nombre, nous avons alors

$$N_B \sim N\mathbb{P}(B).$$

De la même façon, nous pouvons définir le nombre de fois que l'événement $A \cap B$ s'est réalisé et nous avons

$$N_{A\cap B} \sim N\mathbb{P}(A \cap B).$$

Nous voulons connaître la fréquence d'apparition de l'événement A pour toutes les fois où l'événement B a eu lieu, cette fréquence est alors la fraction

du nombre de fois où nous avons eu à la fois A et B divisé par le nombre de fois où nous avons eu B, ce qui donne

$$\frac{N_{A\cap B}}{N_B} \sim \frac{N\mathbb{P}(A\cap B)}{N\mathbb{P}(B)} \sim \frac{\mathbb{P}(A\cap B)}{\mathbb{P}(B)},$$

ce qui semble correspondre avec la définition de probabilité conditionnelle. Ce raisonnement n'est cependant pas assez rigoureux et il faut bien définir ce que nous voulons dire par convergence d'une probabilité en une fréquence, il s'agit donc plus d'une intuition pour mieux comprendre cette définition.

Il est possible que la réalisation de B ne donne aucune information quant à la réalisation de A. Dans ce cas nous déclarons ces événements comme indépendants.

Définition 5. Soit $A, B \in \mathcal{A}$, A et B sont indépendants si et seulement si nous avons

$$\mathbb{P}(A\cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

Notons que si A et B sont indépendants, nous avons alors $P(A|B) = P(A)$.

2.2.4 Espérance et Variance

En statistique, nous voulons souvent déterminer la valeur moyenne d'un échantillon de données et savoir à quel point ces données sont proches de cette valeur moyenne en calculant la variance. En probabilité, la notion d'espérance et de variance est relativement similaire. Si nous avons une variable aléatoire X, nous aimerions déterminer la valeur moyenne après un échantillon de N observations de X. La définition dans le cas où l'ensemble Ω est dénombrable rappelle l'expression de la moyenne en fonction des fréquences.

Définition 6. Soit un espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$ avec Ω dénombrable. Soit $X : \Omega \to \mathbb{R}$, une variable aléatoire. L'espérance de X est définie par la relation

$$\mathbb{E}[X] = \sum_{\omega\in\Omega} \mathbb{P}(\omega)X(\omega) \tag{2.2}$$

La variance représente l'écart entre les observations de X et de $E[X]$, elle est définie comme suit.

Définition 7. Soit un espace probabilisé $(\Omega, \mathcal{A}, \mathbb{P})$ avec Ω dénombrable. Soit $X : \Omega \to \mathbb{R}$, une variable aléatoire. La variance de X est définie par la relation

$$Var[X] = \mathbb{E}[(X - \mathbb{E}[X])^2].$$

Pour des espaces probabilisés plus complexes, il est possible de définir rigoureusement l'espérance à l'aide d'éléments de la théorie de la mesure en topologie. Ces définitions dépassent, cependant, l'utilisation que nous aurons de la théorie des probabilités au cours de cette thèse.

2.2.5 Formules usuelles

Nous allons dans cette section donner quelques formules usuelles en probabilité que nous utiliserons à de nombreuses occasions.

Identité sur les mesures de probabilités

Monotonie de la mesure de probabilité : Soit $A \subset B$ deux événements de $\mathcal{A}$. Nous avons $B = A \cup (B \setminus A)$. Comme $A \cap (B \setminus A) = \emptyset$, l'union est disjointe et par additivité, nous avons $\mathbb{P}(B) = \mathbb{P}(A) + \mathbb{P}(B \setminus A)$. Comme $\mathbb{P}(B \setminus A) \geq 0$, nous obtenons l'inégalité :

$$\mathbb{P}(A) \leq \mathbb{P}(B). \tag{2.3}$$

Principe d'inclusion-exclusion : Soit deux événements A et B dans $\mathcal{A}$. D'après la propriété de σ-additivité de l'équation (2.1), nous avons

$$\begin{aligned}\mathbb{P}(A \cup B) &= \mathbb{P}(A \setminus B) + \mathbb{P}(B \setminus A) + \mathbb{P}(A \cap B) \\ &= \mathbb{P}(A) - \mathbb{P}(A \cap B) + \mathbb{P}(B) - \mathbb{P}(A \cap B) + \mathbb{P}(A \cap B) \\ &= \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B).\end{aligned}$$

Le raisonnement peut être généralisé à un ensemble fini d'événements $A_1, \ldots, A_n$ de $\mathcal{A}$ avec $n \in \mathbb{N}$. Nous obtenons alors dans ce cas l'équation :

$$\mathbb{P}(\bigcup_{i=1}^{n} A_i) = \sum_{k=1}^{n} (-1)^{k-1} \sum_{1 \leq i_1 < i_2 < \cdots < i_k \leq n} \mathbb{P}(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}). \tag{2.4}$$

Sous-additivité : Soit une famille $(A_i)_{i \in \mathbb{N}}$ d'éléments de $\mathcal{A}$. Il découle, de la propriété de σ-additivité et de la monotonie, l'identité

$$\mathbb{P}\left(\bigcup_{i \in \mathbb{N}} A_i\right) \leq \sum_{i \in \mathbb{N}} \mathbb{P}(A_i). \tag{2.5}$$

Probabilités totales : Soit $(E_i)_{i \in \mathbb{N}}$ une partition dénombrable de Ω, avec pour tout $i \in \mathbb{N}$ on a $E_i \in \mathcal{A}$. Pour tout $A \in \mathcal{A}$, nous avons, en utilisant la définition de probabilité conditionnelle ainsi que la σ-additivité :

$$\mathbb{P}(A) = \sum_{i \in \mathbb{N}} \mathbb{P}(A|E_i)\mathbb{P}(E_i). \tag{2.6}$$

Identité sur les variables aléatoires

Autres écritures de l'espérance : Il est possible de ré-écrire l'équation (2.2) en agglomérant les $\omega \in \Omega$ tels que $X(\omega) = x$ avec $x \in \mathbb{R}$. Nous avons alors

$$\mathbb{E}[X] = \sum_{x \in X^{-1}(\Omega)} x\mathbb{P}(X = x). \tag{2.7}$$

Dans le cas où $X^{-1}(\Omega) = \mathbb{N}$, il est également possible de ré-écrire cette dernière équation par

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{P}(X \geq i). \tag{2.8}$$

Pour se convaincre de cette formule, il suffit de voir que les $\omega \in \Omega$ tels que $X(\omega) = i$ appartiennent à l'ensemble $X \geq j$ pour tout les $j \in \{1, \ldots i\}$. De cette façon, chacun de ces ω sont comptés exactement i fois.

Autre écriture d'une loi associée à une variable X : Soit une variable aléatoire $X : \Omega \to \mathbb{N}$, nous avons la relation pour $k \in \mathbb{N}$

$$\mathbb{P}(X = k) = \mathbb{P}(X \geq k) - \mathbb{P}(X \geq k + 1). \tag{2.9}$$

Inégalité de Markov : Soit $X : \Omega \to \mathbb{R}^+$ une variable aléatoire réelle positive, nous avons pour une valeur réelle $\alpha > 0$,

$$\mathbb{P}(X \geq \alpha) \leq \frac{\mathbb{E}[X]}{\alpha}. \tag{2.10}$$

Dans le cadre de cette thèse, nous pourrons être amenés à utiliser toutes ces formules. Nous allons maintenant nous intéresser aux processus stochastiques des chaînes de Markov dont nous allons avoir besoin pour des problèmes de prédictions de branchement, que nous étudierons en détail au cours du Chapitre 3 Section 3.5 et du Chapitre 5.

2.2.6 Chaîne de Markov discrète

Nous allons donc nous intéresser aux processus stochastiques des chaînes de Markov. Cette section est un survol de la théorie sur les chaînes de Markov discrètes dont les détails peuvent être retrouvés dans de nombreux livres sur la théorie des probabilités et de la stochastique. Nous avons pris une partie des résultats donnés dans le chapitre 8 du livre de Rosenthal [38], les preuves sont omises mais peuvent être retrouvées dans ce livre. Nous allons dès à présent noter par S un ensemble fini ou dénombrable d'états. C'est sur cet ensemble d'états que nous faisons une marche aléatoire. Nous notons pour tout $t \in \mathbb{N}$, la variable aléatoire $X(t)$ qui prend une valeur sur l'ensemble S. Cette variable X va décrire un mouvement sur cette ensemble S dans le temps. Pour tout couple d'états $(x, y) \in S \times S$, nous connaissons la probabilité de transition p_{xy}. Cette probabilité de transition correspond à la probabilité lors de la marche d'aller de l'état x vers l'état y. Plus concrètement, si à tout instant $t \in \mathbb{N}$, nous avons $X(t) = x$, alors nous avons

$$\mathbb{P}(X(t + 1) = y | X(t) = x) = p_{xy}.$$

Bien évidemment, nous devons avoir pour tout $x \in S$,

$$\sum_{y \in S} p_{xy} = 1.$$

Nous devons enfin considérer une distribution initiale qui, pour tout $x \in S$, associe la probabilité de commencer par cet état $\nu_x = \mathbb{P}(X(0) = x)$.

Une chaîne de Markov est la séquence des valeurs prises par $(x_t)_{t \in \mathbb{N}}$, telles que $X(t) = x_t$ pour tout $t \in \mathbb{N}$. Pour les N premières étapes de la marche, nous avons alors la probabilité

$$\mathbb{P}(X(0) = x_0, \ldots, X(N) = x_N) = \nu_{x_0} p_{x_0 x_1} \ldots p_{x_{N-1} x_N} \tag{2.11}$$

qui est vérifiée.

Nous pouvons écrire tous les paramètres de la chaîne de Markov sous forme matricielle. Nous posons ainsi le vecteur ν, dont les composantes sont les dis-

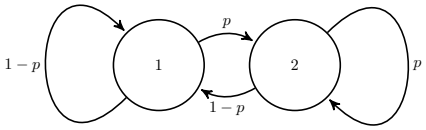


FIGURE 2.1 – Représentation sous forme d'automate d'une chaîne de Markov avec $S = \{1, 2\}$.

tributions initiales ν_x pour tout $x \in S$. Nous notons également la matrice de transition $\mathcal{M} = [p_{xy}]_{(x,y)\in S\times S}$. À chaque instant $t \in \mathbb{N}$, nous avons la distribution d'être à chaque état qui est donnée par les composantes du vecteur $\pi(t)$. Nous avons bien évidemment $\pi(0) = \nu$, et pour tout $n \in \mathbb{N}$, nous avons

$$\pi(n) = (\mathcal{M}^T)^n \nu.$$

Nous allons terminer cette section avec un exemple que nous verrons plus en détail au cours du chapitre 3. Nous allons voir qu'il est également possible de représenter cette chaîne comme un automate (voir Figure 2.1).

Dans cet exemple, l'ensemble des états est $\{1, 2\}$, et nous avons pour un réel $0 < p < 1$ les probabilités de transitions suivantes :

$$p_{11} = 1 - p \qquad p_{12} = p$$

$$p_{21} = 1 - p \qquad p_{22} = p$$

Nous pouvons compresser ces probabilités en une matrice de transition :

$$\begin{bmatrix} 1-p & p \\ 1-p & p \end{bmatrix}.$$

Nous continuerons à utiliser cet exemple pour illustrer les propos qui vont suivre.

Propriétés essentielles des chaînes de Markov

Nous allons maintenant définir des propriétés intéressantes que certaines chaînes de Markov peuvent posséder. Nous verrons à la section suivante que ces propriétés sont essentielles pour démontrer la convergence vers une distribution stationnaire unique.

À partir de maintenant, pour tout $x \in S$ et pour tout événement A, nous notons $\mathbb{P}_x(A) = \mathbb{P}(A|X(0) = x)$. Nous définissons les deux probabilités suivantes :

Pour tout états $i, j \in S$,

$$f_{ij}^{(n)} = \mathbb{P}_i(X(n) = j, \forall m \in [n-1] X(m) \neq j)$$

et

$$f_{ij} = \sum_{n=0}^{\infty} f_{ij}^{(n)},$$

qui sont respectivement la probabilité qu'en commençant par l'état i, nous atteignons pour la première fois j au temps n et la probabilité que l'état j soit atteignable en commençant par l'état i.

Définition 8. Un état $x \in S$ est *récurrent* si et seulement si $f_{xx} = 1$, autrement dit, nous sommes certains de revenir à l'état x en commençant la chaîne par ce dernier. Si un état n'est pas récurrent, on dit alors qu'il est *transient*.

Définition 9. Une chaîne de Markov est *irréductible*, si et seulement si, pour tout $i, j \in S$, nous avons $f_{ij} > 0$. S'il existe $i, j \in S$ tels que $f_{ij} = 0$, alors la chaîne de Markov est *réductible*.

Une chaîne de Markov qui est irréductible est donc une chaîne où il est possible d'atteindre n'importe quel état en commençant par n'importe quel autre état. Il suffit donc de montrer qu'il existe un chemin entre ces deux états qui a une probabilité non-nulle. Si nous notons pour $x, y \in S$ la probabilité $p_{xy}^{(n)} = \mathbb{P}_x(X(n) = y)$ nous avons le théorème suivant qui est vérifié.

Théorème 1. *Soit une chaîne de Markov qui est définie sur un ensemble d'états S et par les probabilités de transitions $\{p_{ij}\}_{i,j \in S}$. Si cette chaîne est irréductible, alors les propriétés suivantes sont équivalentes :*

1. *Il existe un état dans S qui est récurrent.*
2. *Pour tout $i, j \in S$, nous avons $f_{ij} = 1$ et ainsi en particulier, tous les états sont récurrents.*
3. *Il existe $k, l \in S$ tels que $\sum_{n=1}^{\infty} p_{kl}^{(n)} = \infty$.*
4. *Pour tout $i, j \in S$, nous avons $\sum_{n=1}^{\infty} p_{ij}^{(n)} = \infty$.*

Une conséquence de ce théorème est que, sur une chaîne de Markov irréductible, les états sont, soit tous récurrents, soit tous transients.

Définition 10. Pour tout état $x \in S$, la *période* de cet état est le PGCD de l'ensemble des n tels que $p_{xx}^{(n)} > 0$. Si tous les états des S ont une période de 1, alors on dit que la chaîne de Markov est *apériodique*

Cette définition est sensiblement similaire à la définition de périodicité sur les automates. Nous avons de plus le théorème suivant :

Théorème 2. *Si une chaîne de Markov est irréductible, alors tous ses états ont la même période.*

Autrement dit, si nous parvenons à démontrer qu'il existe un état de période 1 dans une chaîne de Markov irréductible, nous montrons aussi qu'elle est apériodique. En particulier, cela est vraie si nous avons un état $x \in S$ tel que $p_{xx} > 0$. Ce qui est le cas de notre exemple où nous avions $S = \{1, 2\}$.

Distribution stationnaire

Soit une distribution sur les états qui est définie par un vecteur $\pi^* = (\pi^*_x)_{x \in S}$.

Définition 11. La distribution π^* est *stationnaire* si pour toutes ses composantes, nous avons

$$\sum_{x \in S} \pi^*_x p_{xy} = \pi^*_y.$$

Nous pouvons écrire cette relation sous forme matricielle, et nous avons alors

$$\mathcal{M}^T \pi^* = \pi^*. \tag{2.12}$$

Le terme stationnaire vient du fait que si à un temps $t \in \mathbb{N}$ nous sommes sous cette distribution, alors pour tout $k \in \mathbb{N}$, nous restons dans cette distribution au temps $t + k$. Si une chaîne de Markov présente les bonnes propriétés, il est alors possible qu'elle ait une unique distribution stationnaire vers laquelle elle converge quelle que soit la distribution initiale. Autrement dit, nous avons un rang T tel que pour tout $t > T$, nous avons $\pi(t) \approx \pi^*$. Nous avons un théorème que nous pouvons utiliser pour démontrer qu'une chaîne de Markov converge vers une telle distribution.

Théorème 3. *Pour toutes chaînes de Markov sur un ensemble fini d'états. Si la chaîne est irréductible et apériodique, alors il existe une unique distribution stationnaire π^* vers laquelle elle converge indépendamment de la distribution initiale. Nous avons donc pour tout $x \in S$, $\lim_{n \to \infty} \mathbb{P}(X(n) = x) = \pi^*_x$.*

Démonstration. Il s'agit de la combinaison du corollaire 8.3.11 et de la proposition 8.4.10 que l'on peut trouver dans [38][p.93 et p.98]. □

Nous allons maintenant montrer une méthode pour déterminer la distribution stationnaire dans le cas où le Théorème 3 est vérifié. Remarquons dans ce cas que par définition, l'Equation (2.12) est vérifiée. Il se trouve que cette équation, signifie que la distribution stationnaire est un vecteur propre de M^T associé à la valeur propre de 1. Nous savons également, comme il s'agit d'une distribution que la somme de ses composantes est égale à 1. Il nous suffit donc de déterminer ce vecteur propre, en déterminant le noyau de la matrice $\mathcal{M}^T - Id$ où Id est la matrice identité de même format que $\mathcal{M}^T$. Il nous suffit donc de résoudre le système linéaire

$$(\mathcal{M}^T - Id) \times \pi^* = 0$$

Appliquons ces nouvelles connaissances sur notre exemple. Comme pour tout x dans $S = \{1, 2\}$, nous avons $p_{xx} = p^{(1)}_{xx} > 0$, il est évident d'après tous les théorèmes que nous venons de voir, que la chaîne est irréductible et apériodique, et comme S est fini, nous convergeons vers une unique distribution stationnaire. La matrice de transitions et sa transposée sont données par

$$\mathcal{M} = \begin{bmatrix} 1-p & p \\ 1-p & p \end{bmatrix}$$

$$\mathcal{M}^T = \begin{bmatrix} 1-p & 1-p \\ p & p \end{bmatrix}$$

Si nous essayons de résoudre le système linéaire

$$(\mathcal{M}^T - Id)x = \begin{bmatrix} -p & 1-p \\ p & p-1 \end{bmatrix} \times x = 0$$

Nous trouvons alors que le noyau est l'ensemble des vecteurs de la forme suivante

$$\mathcal{N}(\mathcal{M}^T - Id)) = \left\{ \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \middle| x_1 = \frac{1-p}{p} x_2 \right\}.$$

Si nous cherchons dans ce noyau le seul vecteur qui a la somme de ses composantes à 1, nous avons alors le vecteur de la distribution stationnaire

$$\pi^* = \begin{bmatrix} 1-p \\ p \end{bmatrix}.$$

Le théorème ergodique

Dans le cadre de l'analyse d'algorithmes dans le cas moyen, nous sommes bien souvent intéressés par l'asymptotique de l'espérance de certaines variables aléatoires. Le théorème ergodique, nous permet de cette façon d'obtenir de tels équivalents asymptotiques. Pour nos besoins, nous allons utiliser un corollaire du théorème ergodique original. Nous donnons directement ce corollaire, que nous appellerons par la suite le théorème ergodique bien qu'il s'agit d'un léger abus.

Théorème 4. *Soit $(\mathcal{M}, \nu)$ une chaîne de Markov irréductible et apériodique qui est définie sur un ensemble S. Notons π^* sa distribution stationnaire. Soit E un ensemble de couples $(x, y) \in S^2$ de probabilités de transitions non-nulles $\mathcal{M}(x, y) > 0$. Pour tout entier strictement positif n, soit L_n une variable aléatoire strictement positive telle que $\lim_{n\to\infty} \mathbb{E}[L_n] = +\infty$. Soit X_n une variable aléatoire qui compte le nombre de fois qu'une transition dans E est prise sur une marche aléatoire de longueur L_n dans $\mathcal{M}$. Indépendamment de la distribution initiale ν, nous avons l'équivalence asymptotique*

$$\mathbb{E}[X_n] \sim \mathbb{E}[L_n] \sum_{(x,y)\in E} \pi(x)\mathcal{M}(x, y).$$

qui est vérifiée.

Démonstration. D'après le théorème ergodique original [32, p.58], si F est un sous-ensemble de S, et si Y_n est une variable aléatoire qui compte le nombre de fois qu'un élément de F est atteint au cours d'une marche aléatoire de taille n sur $\mathcal{M}$, alors, quand n est grand, nous avons l'équivalence asymptotique

$$\frac{1}{n}\mathbb{E}[Y_n] \sim \sum_{x\in F} \pi(x).$$

Soit B l'ensemble de tous les couples $(x, y) \in S^2$ tels que $\mathcal{M}(x, y) > 0$. Nous considérons la chaîne de Markov de deuxième ordre $\mathcal{M}_2$ dont l'ensemble des états est B, et ses transitions sont uniquement définies pour tous les couples $(x, y), (y, z) \in B$ avec $\mathcal{M}_2((x, y), (y, z)) = \mathcal{M}(y, z)$.

$\mathcal{M}_2$ est une chaîne de Markov irréductible. Pour tous les couples $(x, y), (w, z) \in B$, nous avons alors $f_{(x,y),(w,z)} \geq \mathcal{M}(x, y) f_{yw} \mathcal{M}(y, w)$, et comme $\mathcal{M}$ est irréductible et par définition $\mathcal{M}(x, y) > 0$ et $\mathcal{M}(w, z)$, nous avons ainsi $f_{(x,y),(w,z)} > 0$.

$\mathcal{M}_2$ est apériodique. Pour tout état $(x, y) \in B$, comme $\mathcal{M}$ est irréductible, il existe une marche de longueur $n - 1$ avec $n > 1$ commençant par l'état y et se terminant par l'état x de probabilité non nulle, ainsi

$$\mathcal{M}^{(n-1)}(y, x) > 0.$$

De plus nous avons que la probabilité d'un cycle de longueur n dans $\mathcal{M}_2$ commençant par l'état (x, y) vérifie l'expression

$$\mathcal{M}_2^{(n)}((x, y), (x, y)) = \mathcal{M}^{(n-1)}(y, x)\mathcal{M}(x, y),$$

ce qui est non-nulle pour le même n. Comme $\mathcal{M}$ est irréductible, nous avons $f_{xx} > 0$ et donc il existe un ensemble $\Delta = \{\delta_1, \delta_2 \ldots\}$ tel que pour tout $\delta \in \Delta$, nous avons

$$\mathcal{M}^{(\delta)}(x, x) > 0.$$

Il existe donc des cycles de probabilités non-nulles dans $\mathcal{M}_2$ commençant par l'état (x, y) de tailles $n + \delta$ pour tout $\delta \in \Delta \cup \{0\}$. Si nous notons par $\ell_1, \ell_2, \ldots$ les longueurs d'autres cycles à probabilités non-nulles qui ne s'expriment pas comme une somme $n + \delta$, alors nous avons que la période de l'état (x, y) dans $\mathcal{M}_2$ est définie par l'expression

$$pgcd(n, n + \delta_1, n + \delta_2, \ldots, \ell_1, \ell_2, \ldots).$$

Si nous prouvons que $pgcd(n, n + \delta_1, n + \delta_2, \ldots)$ alors nous démontrons que la période de (x, y) vaut 1 et par irréductibilité de $\mathcal{M}_2$, tous les états sont de périodicité 1 et donc $\mathcal{M}_2$ est irréductible. Une première étape consiste à ré-écrire ce PGCD sous la forme suivante :

$$pgcd(n, n + \delta_1, n + \delta_2, \ldots) = pgcd(pgcd(n, n + \delta_1), pgcd(n, n + \delta_2), \ldots).$$

De cette façon nous pouvons utiliser la relation provenant de l'algorithme d'Euclide, qui dit que $pgcd(a, a + b) = pgcd(a, b)$. Nous avons ainsi,

$$pgcd(pgcd(n, n + \delta_1), pgcd(n, n + \delta_2), \ldots) = pgcd(pgcd(n, \delta_1), pgcd(n, \delta_2), \ldots).$$

Nous pouvons ré-écrire cette dernière égalité par

$$pgcd(pgcd(n, \delta_1), pgcd(n, \delta_2), \ldots) = pgcd(n, \delta_1), \delta_2), \ldots).$$

Comme $\mathcal{M}$ est apériodique, le PGCD des éléments de Δ définissent la période de l'état x dans $\mathcal{M}$ qui vaut donc 1. Nous avons alors

$$\begin{aligned} pgcd(n, n + \delta_1, n + \delta_2, \ldots) &= pgcd(n, pgcd(\delta_1, \delta_2, \ldots)) \\ &= pgcd(n, 1) \\ &= 1, \end{aligned}$$

ce qui nous permet de conclure sur que $\mathcal{M}_2$ est apériodique.

Par définition, nous avons directement que $\mathcal{M}_2$ converge vers l'unique dis-

tribution stationnaire π_2^* définie pour tout couple $(x, y) \in B$ par

$$\pi_2^*(x, y) = \pi^*(x)\mathcal{M}(x, y).$$

Nous pouvons maintenant appliquer le théorème ergodique original sur la variable aléatoire Z_ℓ qui compte le nombre de fois qu'un couple dans E est parcouru durant une marche aléatoire de longueur ℓ dans $\mathcal{M}_2$. Nous avons ainsi

$$\mathbb{E}[Z_\ell] = \ell \left(\sum_{(x,y)\in E} \pi^*(x)\mathcal{M}(x, y) \right) (1 + \epsilon_\ell),$$

avec $\epsilon_\ell \to 0$ quand ℓ tend vers l'infini. Intéressons nous à présent au cas où la longueur de la marche aléatoire est la variable aléatoire L_n. Nous avons alors

$$\begin{aligned}
\mathbb{E}[X_n] &= \sum_{\ell\geq 0} \mathbb{P}(L_n = \ell)\mathbb{E}[X_n|L_n = \ell] \\
&= \sum_{\ell\geq 0} \mathbb{P}(L_n = \ell)\mathbb{E}[Z_\ell] \\
&= \sum_{\ell\geq 0} \mathbb{P}(L_n = \ell)\ell \left(\sum_{(x,y)\in E} \pi^*(x)\mathcal{M}(x, y) \right) (1 + \epsilon_\ell) \\
&= (\mathbb{E}[L_n] + \mathbb{E}[L_n\epsilon_{L_n}]) \left(\sum_{(x,y)\in E} \pi^*(x)\mathcal{M}(x, y) \right).
\end{aligned}$$

Pour terminer la preuve, il nous reste à démontrer que $\mathbb{E}[L_n\epsilon_{L_n}]$ est en $o(\mathbb{E}[L_n])$. Par définition, comme ϵ_ℓ tend vers 0 quand ℓ est grand, pour tout $\alpha > 0$, il existe un rang ℓ_0 tel que pour tout $\ell > \ell_0$, nous avons

$$\epsilon_\ell \leq \tfrac{\alpha}{2}.$$

Posons $\eta_0 = \max\{\epsilon_\ell : \ell \in \{0, \ldots, \ell_0\}\}$, nous avons alors

$$\begin{aligned}
E[L_n\epsilon_{L_n}] &= \sum_{\ell\geq 0} \mathbb{P}(L_n = \ell)\ell\epsilon_\ell \\
&= \sum_{\ell=0}^{\ell_0} \mathbb{P}(L_n = \ell)\ell\epsilon_\ell + \sum_{\ell>\ell_0} \mathbb{P}(L_n = \ell)\ell\epsilon_\ell \\
&< \sum_{\ell=0}^{\ell_0} \mathbb{P}(L_n = \ell)\ell\epsilon_\ell + \frac{\alpha}{2}\mathbb{E}[L_n] \\
&< \eta_0\ell_0 + \frac{\alpha}{2}\mathbb{E}[L_n].
\end{aligned}$$

Comme $\mathbb{E}[L_n]$ tend vers l'infini quand n est suffisamment grand, il existe un rang n_0 tel que pour tout $n > n_0$ nous avons

$$\frac{\eta_0\ell_0}{\mathbb{E}[L_n]} \leq \frac{\alpha}{2}.$$

Ainsi pour tout $\alpha > 0$, il existe un rang n_0 tel que pour tout $n > n_0$, nous avons $\mathbb{E}[L_n \epsilon_{L_n}] \leq \alpha \mathbb{E}[L_n]$, ce qui par définition signifie que $\mathbb{E}[L_n \epsilon_{L_n}] = o(\mathbb{E}[L_n])$ ce qui nous permet de conclure cette preuve. $\square$

Chapitre 3

Notions d'informatique

Sommaire

3.1 Algorithmes et théorie de la complexité

Dans cette section, nous allons présenter la discipline qu'est l'algorithmique. Pour commencer, nous donnerons une définition de ce qu'est un algorithme. Nous aborderons également la théorie de la complexité, qui est le domaine dans lequel s'inscrit ce manuscrit. Puis, nous discuterons des paradigmes les plus utilisés dans l'élaboration d'algorithmes.

3.1.1 Qu'est-ce qu'un algorithme ?

Il n'est pas aisé de donner une définition de ce qu'est un algorithme. L'origine du mot viendrait d'une déformation du nom du mathématicien perse Al-Khwārizmāri. Une analogie qui est souvent faite est celle de la recette de cuisine qui consiste en une suite d'instruction pour transformer les ingrédients en un plat. Concrètement un algorithme fait la même chose, il donne une suite d'instruction à effectuer pour transformer une entrée en une sortie, chaque instruction forme une étape de l'algorithme. Un algorithme connu de beaucoup, est le célèbre algorithme d'Euclide que nous apprenons tôt à l'école pour déterminer le plus grand commun diviseur PGCD entre deux nombres entiers. Donald Knuth, qui est l'un des pionniers dans le domaine de l'algorithmique, définit 5 caractéristiques qu'un algorithme doit avoir [29][pp. 5-6] Pour illustrer ses propos, Knuth utilise l'exemple de l'algorithme d'Euclide, nous allons procéder de la même façon en donnant l'algorithme d'Euclide sous la forme suivante :

Données : Deux entiers m et n
Résultat : Le PGCD de m et n, qui est le plus grand entier positif qui divise à la fois n et m.

1 **tant que** $n \neq 0$ **faire**
2 | Faire la division euclidienne de m par n, soit r le reste obtenu de cette dernière.
3 | $m \leftarrow n$
4 | $n \leftarrow r$
5 **retourner** m

Algorithme 1 : Algorithme d'Euclide

Dans son livre, Knuth avait décomposé l'algorithme en 3 étapes. La première, *E1* est l'étape de calcul du reste r de la division euclidienne de m par n qui correspond à la ligne 2. La deuxième, *E2* est l'étape du test de r qui se fait par la combinaison de la ligne 4 suivie par la ligne 1, cette étape vérifie si le reste est 0 et dans ce cas l'algorithme s'arrête. La troisième *E3* est l'étape de réduction, dans cette étape nous réduisons le problème de calculer le PGCD entre m et n par le problème de calculer le PGCD entre n et r. Cela revient à modifier la variable m par la valeur de n, puis de modifier la variable n par le reste r, ce que nous faisons aux lignes 3 et 4.

Nous pouvons dès à présent énumérer les 5 caractéristiques qu'un algorithme possède selon Knuth.

— *Finitude.* Un algorithme doit toujours s'arrêter après un nombre fini d'étapes. L'intérêt d'utiliser un algorithme est évidemment de trouver une solution à un problème, et il est donc nécessaire que ce dernier se termine en un temps fini pour espérer pouvoir obtenir la solution. Dans notre exemple, l'algorithme d'Euclide se termine toujours, si n et m sont

eux même finis. En effet, après l'étape de réduction $n = r$ et $0 \geq r < m$, si on note par $n_0 = n, n_1 = r, \ldots$ les valeurs prises par n après chaque modification, cette suite est alors strictement décroissante et positive et sera donc bien amenée à se terminer par 0 au bout d'un certain nombre fini d'étapes.

— *Définitude.* Chaque étape doit être définie de manière précise. Chaque cas doit ainsi être traité de manière rigoureuse et sans ambiguïté. De manière générale, les étapes sont constituées d'actions élémentaires autorisées. Ce qui est le cas par exemple avec les jeux d'instructions dans un langage informatique, une étape ne peut être définie qu'avec des mots autorisés par ce langage. Dans notre exemple, à l'étape de recherche du reste, le lecteur est supposé comprendre le sens de l'opération de division euclidienne entre deux entiers m et n, cela veut aussi dire qu'à tout instant, au cours de cette étape, il faut que m et n soit des entiers auquel cas la division euclidienne n'est pas bien définie. Ici c'est toujours le cas, puisque c'est vrai à l'initialisation et que l'étape de réduction ne change pas la nature de n et m.

— *Entrée.* Un algorithme peut avoir un certain nombre d'entrées. Une entrée est une quantité qui est donnée à l'initialisation avant même que l'algorithme ne commence ou bien, de manière dynamique au cours de l'exécution. Ces entrées sont choisies dans des ensembles spécifiques. Dans notre exemple, les deux entrées sont n et m qui sont deux éléments de l'ensemble des entiers naturels $\mathbb{N}$.

— *Sortie.* Un algorithme a au moins une sortie. Une sortie est une quantité qui a une relation particulière avec les entrées. Dans le cas de l'algorithme d'Euclide, la sortie est le PGCD des entrées. Pour prouver que c'est bien cette sortie que nous obtenons, il suffit de voir que le PGCD de m et n et le PGCD de n et r sont les mêmes.

— *Efficacité.* Les opérations effectuées par l'algorithme doivent être efficaces, dans le sens où elles doivent être suffisamment basiques pour pouvoir être effectuées en un temps fini par quelqu'un utilisant juste un crayon et une feuille de papier. Les opérations utilisées dans l'algorithme d'Euclide ne consistent qu'à faire la division entre deux entiers, tester si une variable est nulle, et modifier des variables par des nouvelles valeurs entières. De ce fait ces opérations sont efficaces dans ce sens, puisque les nombres entiers peuvent être écrits sur un papier sous une forme finie, et nous connaissons une méthode elle même finie pour diviser deux nombres entre eux. Si nous remplacions cette division d'entier par une division sur des nombres réels arbitraires qui peuvent avoir un nombre de décimal infini, l'opération deviendrait alors inefficace puisqu'il est impossible de représenter ces nombres de manière finie sur un papier.

Pour revenir sur notre analogie de recette de cuisine, cette dernière possède les propriétés de finitude, d'entrée et de sortie. Elle peut cependant manquer de définition rigoureuse, par exemple, si l'instruction consiste à rajouter une poignée de sel, la définition d'une poignée de sel est ambigüe. Un algorithme doit pouvoir être exécuté par une machine et il faut donc faire preuve d'une certaine rigueur lorsque nous écrivons un algorithme.

3.1.2 Théorie de la complexité

Un des éléments importants dans plusieurs de ces définitions est qu'un algorithme doit pouvoir résoudre un problème avec efficacité. Nous devons donc définir des méthodes pour pouvoir mesurer l'efficacité d'un algorithme, et c'est ce que la théorie de la *complexité* nous permet de faire. Le but de la théorie de la complexité est ainsi de permettre de quantifier les performances d'un algorithme. Concrètement, nous nous intéressons au nombre d'opérations élémentaires effectuées par l'algorithme au cours de son exécution. Ce nombre d'opérations doit de plus dépendre de la taille de l'entrée, par exemple, si l'entrée est une séquence, nous pouvons considérer son nombre d'éléments comme la taille de l'entrée. Bien souvent, nous nous intéresserons au comportement asymptotique de ce nombre d'opérations exprimé sous forme d'une fonction de coût $f(n)$ quand la taille de l'entrée n est grande. En pratique, les fonctions de coûts qui vont nous intéresser permettent d'obtenir des estimations de la **durée d'exécution** ou de la **mémoire utilisée** d'une implantation de l'algorithme sur une machine RAM dont nous discuterons par la suite dans la section 3.2.

Il y a trois grands types de comportement qui sont principalement étudiés en théorie de la complexité :

- L'analyse dans **le meilleur des cas**, qui comme son nom le laisse à penser, consiste à regarder le comportement de l'algorithme dans le cas où il reçoit l'entrée qui lui est la plus favorable.
- L'analyse dans **le pire cas**, qui à l'opposée du premier type consiste à regarder le comportement de l'algorithme lorsque celui-ci reçoit l'entrée qui lui est la moins favorable.
- L'analyse dans le **cas moyen**, qui revient à faire la moyenne de tous les coûts sur l'ensemble des entrées possibles. Bien souvent, nous considérons que l'entrée est générée selon une distribution qui nous donne ainsi sa probabilité d'apparition. Nous définissons alors la complexité dans le cas moyen en regardant le comportement asymptotique de l'espérance du coût. Par défaut, nous choisissons souvent la distribution uniforme ce qui revient à la définition d'origine. Il peut cependant être intéressant de considérer d'autres distributions, et c'est ce que nous faisons par la suite dans le Chapitre 6.

Remarque. Il est possible de faire une analogie avec les statistiques. Le pire cas est ainsi un maximum de la fonction de coût, sur toutes les entrées nous savons que nous ne dépasserons pas le coût du pire cas. De manière similaire, le meilleur cas est un minimum, pour toute entrée nous avons un coût au moins aussi grand que le meilleur cas. Évidemment, le cas moyen est une moyenne par définition. Bien qu'en pratique il est possible de rencontrer peu fréquemment le pire cas, l'analyse de ce dernier peut rester important pour des applications critiques qui jouent le rôle d'une garantie sur le temps d'exécution de l'algorithme une fois implanté sur une ressource critique.

La notation de Landau

Nous allons faire un rappel sur la notation de Landau qui est grandement utiliser en théorie de la complexité. Nous allons, par la suite, nous intéresser à des fonctions définies sur les entiers positifs à valeurs réelles et croissantes.

Majoration, notation $\mathcal{O}$ Nous l'avons utilisée précédemment pour discuter de la complexité du tri rapide. La notation $\mathcal{O}$ permet de définir une majoration d'une fonction par une autre à une constante près. Plus formellement, on dit qu'une fonction $f(n) = \mathcal{O}(g(n))$, s'il existe un rang n_0 et une constante $c \in \mathbb{R}$, tels que pour tout $n > n_0$, on a l'inégalité $f(n) < c \times g(n)$ qui est vérifiée.

Minoration, notation Ω À l'opposé de la majoration, il est possible de définir une minoration d'une fonction par une autre à une constante près. Dans ce cas, on dit que $f(n) = \Omega(g(n))$ si et seulement si $g(n) = \mathcal{O}(f(n))$.

Borne, notation Θ On dit qu'une fonction est bornée par une autre fonction à une constante près si cette première est à la fois majorée et minorée par cette deuxième. Autrement dit, nous avons $f(n) = \Theta(g(n))$ si et seulement si $f(n) = \Omega(g(n))$ et $f(n) = \mathcal{O}(g(n))$.

Négligeable, notation o On dit que la fonction f est négligeable devant la fonction g si pour toute constante réelle $c \in \mathbb{R}$, il existe un rang $n_0 \in \mathbb{N}$ tel que pour tout $n > n_0$ l'inégalité $f(n) < c \times g(n)$ est vérifiée. Nous notons alors que $f(n) = o(g(n))$. Nous pouvons également remarquer que, si $f(n) = o(g(n))$, alors $f(n) = \mathcal{O}(g(n))$.

Domine, notation ω On dit que f domine g si l'on a $g(n) = o(f(n))$. On note alors que $f(n) = \omega(g(n))$. Nous pouvons ainsi faire une remarque similaire à celle qui a été faite pour la notation o, à savoir que si $f(n) = \omega(g(n))$, alors $f(n) = \Omega(g(n))$.

Équivalent, notation $\sim$ On dit que f est équivalent à g, si et seulement si, $f(n) = g(n) + o(g(n))$. On note alors $f(n) \sim g(n)$. Une fois de plus, nous pouvons faire une remarque similaire aux précédentes, si $f(n) \sim g(n)$ alors $f(n) = \Theta(g(n))$.

D'après les trois remarques précédentes, on peut dire que ces trois premières définitions sont des cas particuliers plus faibles de ces trois dernières définitions. Lorsque l'on cherche à faire l'analyse sur la complexité d'un algorithme, nous procéderons souvent ainsi, premièrement, nous choisirons le type de comportement qui nous intéresse comme par exemple le pire cas, ensuite nous choisissons une fonction de coût comme, par exemple, le nombre de comparaisons total pendant l'exécution de l'algorithme, et enfin, nous utilisons la notation de Landau pour comparer cette fonction avec des fonctions plus simples.

3.1.3 Méthodes d'analyse

Toujours dans le cadre de la théorie de la complexité, nous allons voir différentes méthodes pour déterminer des bornes. Avec l'argument d'adversaire, nous pouvons obtenir des bornes inférieures sur l'ensemble des algorithmes d'un problème spécifique. La méthode d'analyse amortie consiste à considérer les coûts moyens d'une opération dans une structure. Nous utilisons une méthode d'analyse amortie pour faire notre analyse de l'algorithme TimSort dans le chapitre 4.

Argument d'adversaire

Dans cette section, nous aborderons la méthode d'argument d'adversaire qui permet de déterminer une borne inférieure à un problème. Cette méthode nous permet donc de démontrer rigoureusement, que tout algorithme qui trouve une solution à ce problème doit avoir une complexité qui est supérieure à cette borne.

Illustrons le principe de fonctionnement avec l'exemple de la bataille navale. Pour rappel, une partie de bataille navale se compose en une première étape où nous plaçons des bâteaux dans un espace quadrillé. La deuxième étape est la partie en elle-même où chaque joueur essaye de deviner la position de la flotte de son adversaire en tirant un missile à une position à tour de rôle. Après un tir, l'adversaire doit alors dire si le joueur a fait un tir à blanc, touché un bâteau, ou couler un bâteau si ce dernier a subi un tir sur toutes ses positions. Un adversaire malhonnête pourrait faire durer la partie en trichant. Au lieu de positionner ses bateaux dès le début de la partie, il va le faire à la volée au cours de la partie en fonction des tirs du joueur opposé. Tant que ses réponses restent cohérentes, ce joueur ne peut pas se rendre compte que son adversaire est un tricheur.

C'est exactement le fonctionnement d'un *argument d'adversaire*. L'idée étant qu'un adversaire va mentir à l'algorithme en construisant l'entrée à la volée, dans le but de faire trainer le jeu et faire poser le plus de questions à ce dernier. L'algorithme n'a donc pas accès directement à l'entrée et ne peut donc pas faire d'observation directement. L'adversaire est l'intermédiaire entre l'algorithme et l'entrée. Bien entendu, si l'algorithme recevait l'entrée que l'adversaire génère à la volée, il prendrait le même temps. De cette façon, si nous montrons qu'un adversaire peut faire durer ce jeu de questions pour un certain temps indépendamment de l'algorithme contre lequel il joue, nous obtenons alors une borne inférieure, puisqu'il existe pour tous ces algorithmes au moins une entrée qui aura le coût correspondant à cette borne.

Essayons de formaliser un peu tout ce que nous venons de dire jusqu'à présent. Dans un argument d'adversaire, un jeu est joué entre un algorithme et un adversaire. Dans ce jeu, l'algorithme doit générer la ou les sorties en faisant certaines observations sur l'entrée. Il n'a cependant pas directement accès à cette dernière et doit alors poser des *questions* à l'adversaire. Le but de l'algorithme est alors de poser le moins de questions possibles afin de déterminer la sortie le plus rapidement possible. Le but de l'adversaire est, au contraire, de faire traîner le jeu le plus longtemps possible en faisant en sorte que l'algorithme pose un maximum de questions. L'adversaire peut mentir aux questions posées par l'algorithme mais il doit rester cohérent dans ses réponses. Par exemple, dans le cas où on a des algorithmes de tri pour une séquence $X = \langle x_1, \ldots, x_n \rangle$, si l'algorithme demande si $x_1 < x_2$ et $x_2 < x_3$ et que l'adversaire répond "oui" à ces deux questions, il n'a pas le droit de répondre "non" si on lui demande en troisième question si $x_1 < x_3$, car par transitivité ça doit être vraie. Bien entendu, il faut faire attention aux questions qu'un algorithme peut poser. Nous considérons que les questions appartiennent à un ensemble de questions élémentaires autorisées. De cette façon, toujours dans notre exemple d'algorithme de tri, il est interdit de poser la question "est-ce que la séquence est triée ?". Les questions doivent refléter la borne que nous cherchons à établir. Dans le cas d'un algorithme de tri par comparaisons, nous n'autoriserons, par exemple, que les questions de comparaisons entre deux éléments de la séquence d'entrée. Si l'adversaire a la garantie de pouvoir faire durer le jeu pendant m questions,

alors m est une borne inférieure au problème, et ainsi tout algorithme qui essaye de résoudre ce problème doit nécessairement poser au moins autant de questions pour résoudre le problème. Si par exemple, pour un problème donné, l'ensemble des questions est défini par toutes les comparaisons possibles entre deux éléments dans une séquence d'entrée, alors un algorithme qui n'utilise que ces comparaisons pour résoudre ce problème a une complexité qui est au moins de m.

Une démonstration par argument d'adversaire consiste donc à bien définir le jeu qui est joué, ainsi que les questions qui sont autorisées. Notre but est alors de décrire la stratégie utilisée par un adversaire et de montrer combien de temps il est capable de faire traîner le jeu indépendamment de l'algorithme contre lequel il joue. Nous devons ensuite montrer que, pour cette stratégie les réponses données sont toujours cohérentes. Voyons donc dès à présent des exemples d'arguments d'adversaires.

Problème de recherche du minimum : Nous avons en entrée une séquence de taille n, $X = \langle X_1, \ldots, X_n \rangle$ et nous avons une relation d'ordre sur ces éléments que l'on peut donc comparer. Nous désirons obtenir en sortie l'élément minimal de la séquence. Dans ce jeu, les seules questions autorisées sont celles où l'on compare deux éléments de la séquence. La stratégie utilisée par l'adversaire consiste à construire la séquence d'entrée $X = \langle X_1, \ldots, X_n \rangle$ au fur et à mesure qu'il donne des réponses aux questions de l'algorithme. Au commencement, la séquence est initialisée avec uniquement la valeur 0 pour tous ses éléments. Lorsque le joueur pose une question de type "Est-ce que $X_i < X_j$?" pour $i, j \in [n]$, l'adversaire a alors deux cas possibles :

1. Soit $X_i = X_j = 0$, dans ce cas il répondra systématiquement par "oui" à la question du joueur. Il faut ensuite mettre à jour la séquence X. Pour ce faire, nous ajoutons 1 à toutes les valeurs non-nulles de X. Nous fixons également $X_j = 1$.
2. Soit $X_i \neq X_j$, il suffit alors de répondre correctement à la question en fonction des valeurs de X_i et de X_j. Pour ce cas, il n'y a aucune mise à jour à effectuer dans la séquence X.

Nous pouvons alors remarquer que cette stratégie permet à l'adversaire d'être cohérent dans ses réponses. Ces deux cas conservent l'ordre entre les éléments. Dans le deuxième cas, nous ne faisons aucune mise à jour et donc il est évident que l'ordre entre les éléments est inchangé. Remarquons en particulier qu'un nombre qui est de valeur nulle est alors plus petit que tous les autres éléments qui ont déjà été modifiés par la mise à jour du premier cas. Soit X_i et X_j les éléments qui ont été comparés dans le premier cas, nous savons donc que $X_i = X_j$. Remarquons que pour tout $k \in [n]$ si $X_k > X_j = 0$ alors après la mise à jour, nous avons X_k qui devient $X_k + 1$ et donc $X_k + 1 > 1$, X_k reste donc plus grand que X_j après la mise à jour. De plus, comme nous ajoutons la même quantité à toutes les valeurs non nulles, l'ordre reste identique entre ces valeurs après mise à jour.

Maintenant que nous avons vu que cette stratégie d'adversaire fonctionne, nous pouvons obtenir une borne inférieure au problème de la recherche du minimum.

Proposition 1. *Le problème de la recherche d'un élément minimum dans une séquence d'entrée S de taille n admet une borne inférieure en $n-1$ comparaisons.*

Démonstration. L'algorithme peut trouver une solution lorsqu'il ne reste plus qu'un seul 0 dans le tableau de l'adversaire. En effet, s'il ne reste qu'un seul 0, si l'algorithme demande à l'adversaire si cet élément est plus petit qu'un autre, ce dernier répondra systématiquement "oui" d'après le deuxième cas de notre stratégie. Cet élément est donc plus petit que tous les autres dans la séquence et est ainsi le minimum que l'algorithme peut donner en solution. Si il reste au moins deux 0 dans le tableau, il est impossible de conclure pour l'algorithme, puisqu'il existe au moins deux éléments dans le tableau dans la séquence dont l'algorithme ne peut déterminer lequel est le plus petit. La condition du 0 unique dans le tableau est donc nécessaire et suffisante. Remarquons que l'unique façon pour l'algorithme de retirer des 0 dans le tableau de l'adversaire est de forcer le premier cas, lorsque l'on compare $X_i = X_j = 0$. De plus, il ne peut retirer qu'un seul 0 par question de cette façon. Il est alors nécessaire pour l'algorithme de forcer ce cas au moins $n-1$ fois afin que le tableau de l'adversaire ne contienne plus qu'une unique valeur à 0, ce qui demande alors nécessairement de poser au moins autant de questions. Nous pouvons donc conclure la preuve, quel que soit l'algorithme qui joue contre cet adversaire, il est nécessaire de faire au moins $n-1$ comparaisons. □

Remarque. Si nous cherchons à trouver le maximum au lieu du minimum, nous pouvons utiliser le même argument. En effet, chercher le maximum dans une séquence $X = \langle X_1, \ldots, X_n \rangle$ revient à chercher le minimum dans $X' = \langle -X_1, \ldots, -X_n \rangle$. Il suffit donc que l'adversaire réponde aux questions de l'algorithme en générant X' à la place de X.

Problème de recherche du minimum et du maximum : Nous avons, une fois de plus, une séquence d'éléments que nous pouvons ordonner. Cette fois-ci, nous voulons avoir en sortie l'élément minimum et maximum de cette séquence. Ce problème est proche du précédent, mais nous voulons également obtenir dans le même temps ce maximum. Comme la recherche est simultanée, la borne inférieure n'est pas nécessairement une somme des deux bornes, c'est ce que nous allons montrer avec cet exemple.

Les questions autorisées pour ce jeu sont les mêmes que pour le jeu précédent. Une fois de plus, nous allons définir une stratégie d'adversaire pour déterminer une borne inférieure à ce problème. Pour maintenir la cohérence, l'adversaire va, premièrement, construire une séquence de la taille de l'entrée n. Ce tableau sera rempli premièrement par des symboles dont nous allons voir la signification. Le premier symbole est $+$ qui signifie que l'élément est potentiellement le maximum. Le deuxième symbole est $-$ qui signifie que l'élément est potentiellement le minimum. Enfin, le dernier symbole est $\pm$ qui signifie que l'élément peut à la fois être le minimum ou le maximum. Au début, comme l'algorithme ne sait absolument rien sur l'entrée, chaque élément peut alors potentiellement être soit le minimum soit le maximum. Nous initialisons donc la séquence par tous ses éléments égaux à $\pm$. Nous allons également retenir un compteur c initialisé par la valeur 1. Au cours du jeu, si il devient certain qu'un élément n'est ni le maximum ni le minimum de la séquence, une valeur dépendant du compteur c lui sera alors attribuée. Regardons maintenant dans le détail les différents cas de questions que l'algorithme pourrait poser. Nous donnons pour tous ces cas la réponse à donner et les mises à jours nécessaires à effectuer dans la séquence

$X_i \setminus X_j$	$-$	$+$	$\pm$	cst.
$-$	non	oui	oui	oui
$+$	non	oui	non	non
$\pm$	non	oui	oui	oui
cst.	non	oui	oui	$X_i < X_j$

(a) Réponses à $X_i < X_j$

$X_i \setminus X_j$	$-$	$+$	$\pm$	cst.
$-$	$X_i \leftarrow -c$ $c \leftarrow c+1$	r.à.m.	$X_j \leftarrow +$	r.à.m.
$+$	r.à.m	$X_i \leftarrow c$ $c \leftarrow c+1$	$x_j \leftarrow -$	r.à.m.
$\pm$	$X_i \leftarrow +$	$X_i \leftarrow -$	$X_i \leftarrow -$ $X_j \leftarrow +$	$X_i \leftarrow -$
cst.	r.à.m.	r.à.m.	$X_j \leftarrow +$	r.à.m.

(b) Modifications après réponse

FIGURE 3.1 – La stratégie à employer par l'adversaire dans un jeu de recherche simultanée du minimum et du maximum dans une séquence $X = \langle X_1, \ldots, X_n \rangle$. Pour certains $i, j \in [n]$ nous donnons la stratégie à suivre lorsque l'algorithme pose la question $X_i < X_j$. Le tableau (a) donne les réponses à donner pour tous les cas tandis que le tableau (b) donne les modifications à faire après avoir donner la réponse. Le terme r.à.m. dans le tableau (b) signifie qu'il n'y a rien à modifier.

X ainsi que du compteur c. La stratégie et les mises à jours à effectuer sont résumées dans les deux tableaux de la figure 3.1.

Nous allons regarder dans le détail la troisième ligne de ces tableaux, où nous faisons la comparaison $X_i < X_j$ pour certains $i, j \in [n]$ avec $X_i = \pm$. Le but de l'adversaire est de faire en sorte de laisser un maximum de "doutes" à l'algorithme et c'est ce que nous arrivons à faire avec cette stratégie. Ainsi, quand $X_j = -$, l'adversaire doit répondre par "non" ce qui signifie que $X_j \leq X_i$, donc X_j reste potentiellement un minimum mais X_i ne peut être qu'un maximum puisqu'il est forcément plus grand que X_i, d'où la mise à jour $X_i \leftarrow +$. Si l'adversaire avait choisi de répondre par "oui" à la place, alors nous aurions $X_i < X_j$, et donc X_j ne peut plus être un minimum, mais en plus, X_i ne peut plus être un maximum. Nous aurions donc ainsi eu à faire deux modifications et de la même façon, nous pouvons considérer que l'adversaire donne deux fois plus d'informations à l'algorithme, ce qui n'est pas une bonne idée quand le but est de faire durer la partie. Le raisonnement est similaire pour la colonne suivante où $X_j = +$. Pour la colonne suivante, $X_j = \pm$, quelle que soit la réponse, nous avons une symétrie évidente et nous donnons donc la même quantité d'informations à l'algorithme. Nous avons donc choisi de répondre par "oui" et de faire les modifications nécessaires (i.e. $X_i \leftarrow -$ et $X_j \leftarrow +$). La dernière colonne est le cas où X_j est une constante. Pour ce cas, peu importe la réponse, nous perdons l'indétermination qui dit que X_i est potentiellement le minimum ou le maximum. Une fois encore, nous répondons par défaut "oui", et par conséquent X_i ne peut plus être le maximum d'où la modification $X_i \leftarrow -$.

Nous allons maintenant donner l'intuition pour montrer que l'adversaire reste cohérent dans ses réponses. Nous avons déjà vu dans l'exemple précédent, que les modifications étaient cohérentes avec les réponses. Remarquons qu'il y a deux cas pour lesquels nous assignons une constante à un élément X_i. Dans les deux cas nous comparons X_i avec un X_j qui a le même symbole simple $+$ ou $-$. Pour le cas où $X_i = -$, nous fixons alors $X_i = -c$, tandis que pour le cas où $X_i = +$, nous le fixons à $X_i = c$. Tout élément qui est donc potentiellement minimum par l'attribution du symbole $-$ peut alors être considéré comme un nombre strictement négatif tandis que tout élément qui est potentiellement un maximum par l'attribution du symbole $+$ peut alors être considéré comme un nombre strictement positif. Cette considération devient vraie au moment où le symbole a été attribué et restera vraie jusqu'à la fin du jeu. De ce fait si $X_i = -$ et si $X_j = +$ la réponse "oui" reste cohérente jusqu'à la fin du jeu. Comme les valeurs attribuées ne changent plus jusqu'à la fin du jeu, la comparaison entre deux constantes est elle aussi inchangée. Enfin, comme nous incrémentons le compteur c pour les deux cas $X_i = X_j = -$ et $X_i = X_j = +$, l'ordre est préservé entre les éléments jusqu'à la fin du jeu. Par exemple si $X_i = X_j = -$, $X_i = -c$ après modification. Si plus tard, nous faisons une comparaison entre $X_j < X_k$ pour un certain $k \in [n]$, avec $X_j = X_k = -$, alors le compteur c aura déjà été incrémenté au moins une fois et donc nous aurons à ce moment $X_i > -c$ donc après modification de X_j par $-c$, nous aurons toujours et ce jusqu'à la fin du jeu que $X_j < X_i$, la réponse reste donc cohérente.

Nous allons maintenant utiliser cette stratégie d'adversaire pour démontrer la proposition suivante :

Proposition 2. *Le problème de la recherche simultanée du minimum et du maximum dans une séquence d'entrée S de taille n admet une borne inférieure en $\frac{3n}{2} + \mathcal{O}(1)$ comparaisons.*

Démonstration. Le terme en $\mathcal{O}(1)$ ne sert qu'à capter la parité de n, ainsi, pour plus de simplicité, nous allons donc considérer ici uniquement le cas où $n > 0$ est pair. L'algorithme peut découvrir le minimum et le maximum de la séquence, uniquement quand il ne reste plus qu'un unique élément dont le symbole est $-$ et un unique élément dont le symbole est $+$ et plus aucun élément $\pm$. En effet, c'est une condition suffisante, puisque dans ce cas il est évident que l'élément avec le signe $-$ est le minimum de la séquence et l'élément avec le symbole $+$ est le maximum. C'est également une condition nécessaire, si il reste au moins deux éléments avec soit le symbole $-$ soit le symbole $+$, alors si on prend deux de ces éléments de même symbole nous ne savons pas les comparer et il est donc nécessaire de poser la question à l'adversaire. De même, si il reste au moins un élément avec un symbole $\pm$, alors il est impossible de savoir si cet élément est potentiellement un minimum ou un maximum. Il faut donc faire une comparaison de cet élément avec un autre élément quelconque dans la séquence.

Comme nous pouvons le remarquer sur la Figure 3.1, quand nous comparons un élément dont le symbole est $\pm$, ce dernier ne sera pas directement remplacé par une constante mais soit par $-$ soit par $+$. L'algorithme doit donc faire en sorte d'éliminer tous ces symboles, il ne peut parvenir qu'à en éliminer au plus 2 en faisant des comparaisons entre deux éléments X_i et X_j, pour certains $i, j \in [n]$, tels que $X_i = X_j = \pm$. De cette façon comme n est pair, il est nécessaire de faire au moins $\frac{n}{2}$ comparaisons pour parvenir à éliminer tous ces symboles $\pm$. Notons par n_+ le nombre d'éléments qui se verront attribuer le

symbole + au cours du jeu. Nous notons respectivement par n_- le nombre d'éléments qui se verront attribuer le symbole $-$ au cours du jeu. Nous avons bien évidemment $n = n_- + n_+$. Pour éliminer un symbole $-$, l'unique façon d'y parvenir est de comparer $X_i = X_j = -$ pour certains $i, j \in [n]$. Ce type de comparaisons n'élimine qu'un seul des deux symboles et il est donc nécessaire de faire au moins $n_- - 1$ de ces comparaisons. De la même façon, pour éliminer les symboles $+$, il faut forcer les comparaisons entre $X_i = X_j = +$ pour certains $i, j \in [n]$. Il faut donc faire au moins $n_+ - 1$ de ces comparaisons. Pour parvenir à trouver une solution, l'algorithme va donc avoir à faire au moins

$$\frac{n}{2} + (n_- - 1) + (n_+ - 1) = \frac{3n}{2} - 2$$

comparaisons, ce qui nous permet de conclure. □

Cette borne est la meilleure possible, nous verrons dans le chapitre 5 qu'il existe un algorithme qui dans le pire cas atteint cette borne.

Problème de la fusion de listes triées Nous avons en entrée deux listes d'éléments comparables triées dans le même ordre. En sortie, nous souhaitons obtenir une seule liste contenant tous les éléments de ces deux listes et cette liste doit être triée dans le même ordre que les deux listes à l'entrée. Ce problème a de multiples applications, mais la plus connue est, sans doute, l'utilisation en tant que sous-routine dans les algorithmes de type *tri fusion* où l'on appelle récursivement l'algorithme pour trier plusieurs sous-listes qui sont ensuite fusionnées par cette routine. Une autre application connue, mais qui n'apparaît pas directement quand on lit ce problème, est qu'il est possible de faire une recherche d'un élément dans une liste triée L. En effet, si l'on transforme l'élément en une liste de taille 1 alors elle est forcément triée, nous pouvons alors faire en sorte de fusionner cette liste de taille 1 avec la liste L, la position où se trouve alors cet élément nous donne un encadrement et si l'élément est dans L alors, soit l'élément précédent, soit le suivant seront de même valeur. Par la suite, nous considérerons que tous les éléments des deux listes sont distincts. Cette considération est sans conséquence, nous pourrions imposer un système de priorité entre les deux listes et l'ordre d'apparition des éléments dans les deux listes pour aider à la décision et avoir un ordre pour lequel on peut considérer que tous les éléments sont distincts. Nous allons maintenant définir une stratégie d'adversaire pour ce problème et déduire une première borne inférieure. Par la suite, nous posons $X = \langle x_1, \ldots, x_k \rangle$ et $Y = \langle y_1, \ldots, y_p \rangle$ les listes à l'entrée respectivement de tailles k et p. La stratégie d'adversaire consiste à maintenir un graphe orienté $G = (V, E)$ dont chaque sommet représente un élément d'une des deux listes d'entrées et dont chaque arc $(x, y) \in E$ signifie que l'on sait que $x < y$. À l'initialisation, le graphe est constitué de deux composantes connexes correspondant à deux arbres unaires. Chaque arbre représente alors une de ces deux listes. Ces deux composantes apparaissent car nous savons que les deux listes sont triées. La Figure 3.2 représente la configuration initiale pour $k = 4$ et $p = 3$.

Remarque. À tout moment ce graphe est acyclique. En effet, si il existait un cycle nous aurions une contradiction avec la propriété de transitivité de la relation d'ordre.

Dans ce jeu, les questions autorisées ne concernent que la comparaison entre deux éléments, le premier élément doit appartenir à la première liste, le deuxième élément doit appartenir à la seconde liste. Nous ignorons donc les questions entre deux éléments d'une même liste pour lesquels la réponse est triviale de par l'hypothèse que ces deux listes sont triées. Nous noterons par la suite pour tout $x \in V$ et $y \in V$ l'ensemble de tous les chemins allant de x jusqu'à y par $\Gamma(x, y)$. Nous noterons par la suite la fonction de longueur $\lambda : \cup_{x,y \in V} \Gamma(x, y) \mapsto \mathbb{N}$, qui à un chemin entre deux sommets x et y de V associe son nombre de sommets. En particulier, l'algorithme peut trouver une solution, si et seulement si, il existe un chemin C entre deux sommets x et y tel que $\lambda(C) = p+k$. Cette condition revient à dire qu'il existe un chemin dans le graphe qui passe par tous les sommets, nous prouvons par la suite que cette condition est nécessaire et suffisante pour que l'algorithme soit capable de trouver une solution.

Lemme 3. *L'algorithme peut trouver une solution, si et seulement si, il existe un chemin passant par tous les sommets du graphe.*

Démonstration. Remarquons que, si nous avons dans ce graphe un chemin de taille ℓ, $C = \langle c_1, \ldots, c_\ell \rangle$, nous savons par définition que C est trié. Ainsi, si il existe un chemin passant par tous les sommets du graphe, ce chemin définit la liste triée que nous voulons obtenir en sortie. Considérons maintenant que nous avons trouvé une solution, mais qu'il n'existe aucun chemin passant par tous les sommets du graphe. Il existe alors au moins deux sommets $x \in V$ et $y \in V$ pour lesquels il n'existe aucun chemin qui passent par ces deux sommets. Or, comme nous avons une solution, nous connaissons l'ordre entre x et y. Par construction du graphe, il doit donc exister un chemin reliant x à y, ce qui est une contradiction. □

Nous pouvons nous servir de ce lemme pour faire en sorte de retenir des informations sur le graphe. Pour chaque sommet $x \in V$, nous notons par $\pi(x) = \max\{\lambda(C) \mid \forall y \in V, \forall C \in \Gamma(y, x)\}$ la longueur du plus long chemin dans le graphe qui s'arrête à x. De la même façon, nous notons par $\tau(x) = \max\{\lambda(C) \mid \forall y \in V, \forall C \in \Gamma(x, y)\}$ la longueur du plus long chemin dans le graphe qui commence par x. Nous avons mis ces valeurs en étiquette de chaque sommet de la configuration initiale de la Figure 3.2 sous la forme $(\pi(x), \tau(y))$ pour tout sommet $x \in V$. Le lemme précédent dit alors que l'algorithme peut obtenir une solution uniquement lorsqu'il existe un sommet $x \in V$ tel que $\tau(x) = k + p$.

Remarque. Par définition de π et de τ, nous avons que pour tout sommet $x \in V$, $\pi(x) + \tau(x) = q_x + 1$, où $q_x = \max\{\lambda(C) \mid \forall y \in V, \forall z \in V, \forall C = \langle \ldots, x, \ldots \rangle \in \Gamma(y, z)\}$ est la longueur du plus long chemin passant par x. La preuve se fait par l'absurde, si l'on suppose que cette relation est fausse alors, on aurait une contradiction sur le fait qu'au moins $\pi(x)$ ou $\tau(x)$ ne serait pas un maximum.

Lorsque l'algorithme pose une question $x < y$, pour deux éléments x, y dans nos deux listes, voici comment l'adversaire doit répondre. Il répond "oui" si $\pi(x) + \tau(y) \leq \pi(y) + \tau(x)$ et ajoute l'arc (x, y) dans le graphe. Dans le cas contraire il répond "non" et ajoute l'arc (y, x) dans le graphe. Il est alors nécessaire de mettre à jour les valeurs retenues pour les fonctions π et τ pour chaque sommet. L'opération à appliquer pour faire ces mises à jour est la suivante, soit deux sommets u et v de V tels que $(u, v) \in E$.

— Si $\tau(u) < \tau(v) + 1$ alors on modifie $\tau(u)$ par $\tau(v) + 1$.

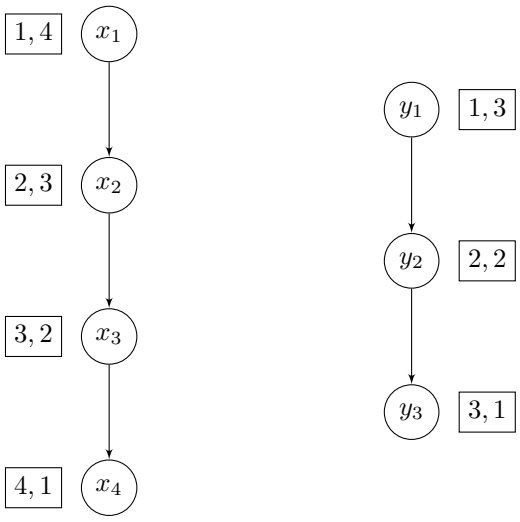


FIGURE 3.2 – Cette figure donne la représentation, stockée par l'adversaire, de l'entrée pour le problème de la fusion de listes triées au début du jeu. À côté de chaque sommet x sont affichées la valeur $\pi(x)$ à gauche et la valeur $\tau(x)$ à droite. Par exemple $\pi(x_2) = 2$ et $\tau(x_2) = 3$.

— Si $\pi(v) < \pi(u) + 1$ alors on modifie $\pi(v)$ par $\pi(u) + 1$.

Cette opération est réitérée jusqu'à stabilité, c'est-à-dire lorsque plus aucune des contraintes ci-dessus est vraie. Si l'algorithme pose la question $x \geq y$, il suffit de répondre le contraire de la question $x < y$ et d'exécuter les mêmes opérations décrites, ci-dessus, en fonction de la réponse à $x < y$. Ces opérations mettent correctement à jour les valeurs de π et de τ. Regardons le cas de la modification de τ, le cas de π est similaire. Nous regardons pour chaque arc entre (u, v) quel est le plus grand chemin qui commence par u. Après l'ajout d'un nouvel arc, le meilleur chemin est alors soit rester le même, soit il s'agit du chemin qui passe par l'arc (u, v) et qui ensuite passe par le meilleur chemin commençant par v.

Nous avons illustré un exemple de partie entre l'algorithme classique où les deux plus petits éléments des deux listes sont comparés avec notre stratégie d'adversaire. Cet exemple est montré en Figure 3.1.

Proposition 3. *Après une comparaison entre deux éléments, la longueur du plus long chemin dans le graphe est au plus augmentée de* 1.

Définition 12. Nous appellerons par la suite une comparaison qui, lorsqu'elle est effectuée par l'algorithme, fait augmenter la taille du plus long chemin, une comparaison utile.

Démonstration. Pour qu'une comparaison soit utile, il est nécessaire de comparer deux éléments x et y appartenant à deux chemins distincts. Posons q_1 et q_2 la longueur des plus longs chemins passant chacun respectivement par x et y. Nous pouvons supposer, sans nuire à la généralité de la preuve, que $q_1 \leq q_2$. Soit q la longueur des plus longs chemins dans le graphe. Par définition, nous avons alors $q_1 \leq q_2 \leq q$. Il existe alors $\alpha \in [0, q_1 - 1]$ tel que $\pi(x) = 1 + \alpha$

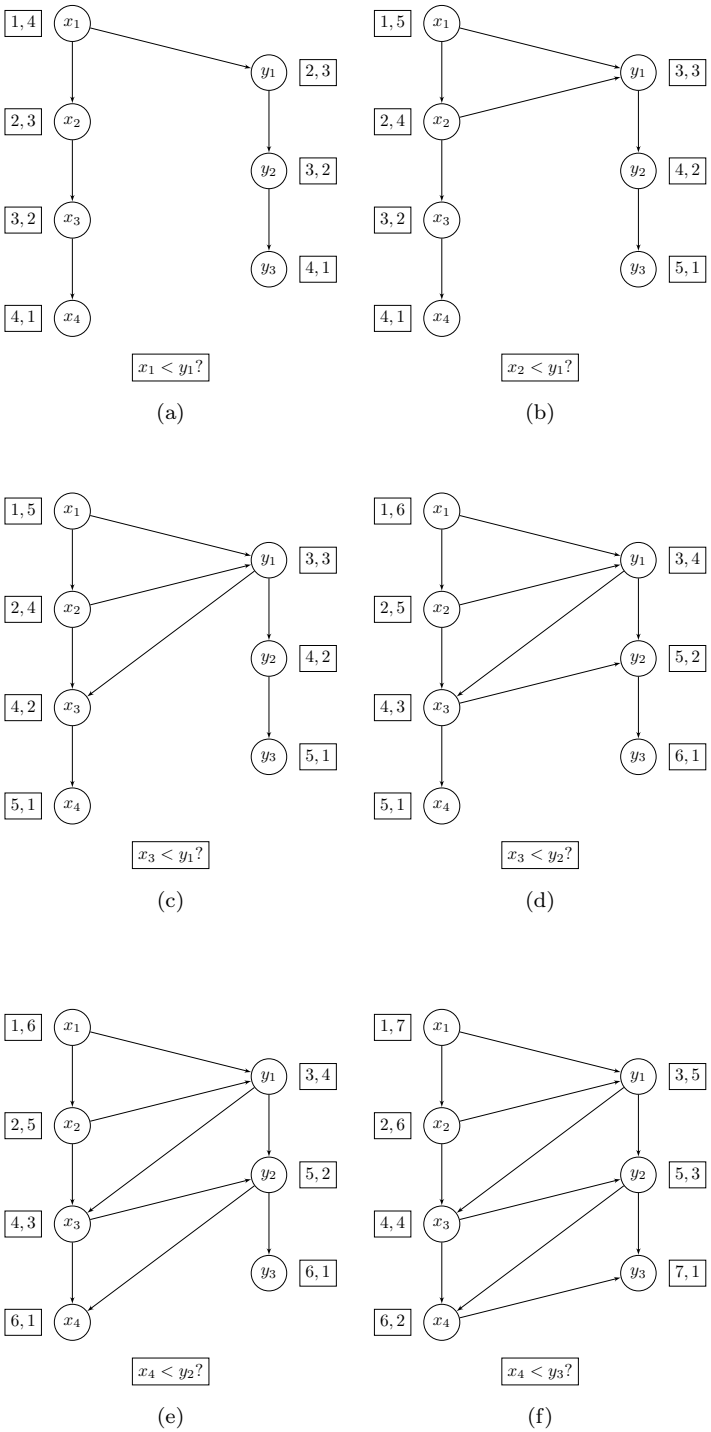


FIGURE 3.1 – Un exemple de jeu entre l'algorithme classique qui compare les deux éléments les plus petits des deux listes contre l'adversaire.

et $\tau(x) = q_1 - \alpha$. De même, il existe $\beta \in [0, q_2 - 1]$ tel que $\pi(y) = 1 + \beta$ et $\tau(y) = q_2 - \beta$. Après qu'une réponse à une question ait été donnée, la nouvelle longueur du chemin le plus long est donnée par

$$p' = \min(q_1 + 1 + \delta; q_2 + 1 - \delta) > p,$$

avec $\delta = \beta - \alpha$. Si $\delta < 0$, alors $p' = q_2 + 1 + \delta < q + 1$ ce qui est une contradiction à l'hypothèse que la comparaison est utile. Si $\delta > q_2 - q_1$, alors $p' = q_1 + 1 - \delta < q + 1$ ce qui est une autre contradiction à cette hypothèse. Le dernier cas est $\delta \in [0, q_2 - q_1]$, dans ce cas nous avons

$$q_1 + 1 \leq min(q_1 + 1 + \delta; q_2 + 1 - \delta) \leq q_2 + 1 \leq q + 1. \tag{3.1}$$

La comparaison est alors utile si $q_2 = q$ et l'on a alors une augmentation de la longueur du chemin le plus long de 1 ce qui est le résultat annoncé. □

Remarque. Si l'on regarde encore plus dans le détail le dernier cas, il est possible de voir qu'il est nécessaire d'avoir $q_1 = q_2 = q$ et donc ce cas se résume à avoir $\delta = 0$. Nous obtenons ainsi le lemme qui suit.

Lemme 4. *Si une comparaison entre x et y est utile, alors il existe deux chemins distincts de longueurs maximales dans le graphe, dont l'un passe par x et l'autre passe par y. De plus $\pi(x) = \pi(y)$ et $\tau(x) = \tau(y)$.*

Démonstration. On a vu précédemment qu'une comparaison est utile dans la preuve de la Proposition 3 uniquement pour $\delta \in [0, q_2 - q_1]$ et que dans ce cas l'Équation (3.1) est vérifiée. Comme $q_2 + 1 \leq q + 1$, il est alors nécessaire d'avoir $q_2 = q$. Pour avoir $\min(q_1 + 1 + \delta; q_2 + 1 - \delta) = q_2 + 1$, il y a potentiellement deux valeurs de δ qui sont $\delta = 0$ et $\delta = q_2 - q_1$. Supposons que $q_1 < q_2$, nous avons alors pour $\delta = q_2 - q_1$ que $q_1 + 1 + \delta = q_2 + 1$ et $q_2 + 1 - \delta = q_1 + 1$ ce qui donne $\min(q_1 + 1 + \delta; q_2 + 1 - \delta) = q_1 + 1$. Pour le cas $\delta = 0$, nous montrons de la même façon que $\min(q_1 + 1 + \delta; q_2 + 1 - \delta) = q_1 + 1$. Il y a donc contradiction sur le fait que la comparaison est utile et donc $q_1 \geq q_2$. Comme nous avons $q_1 \leq q_2$, nous obtenons alors que $q_1 = q_2$. Comme $q_1 = q_2$, nous avons alors que $\delta \in [0, 0]$ et ainsi $\delta = 0$. □

La Proposition 3 nous permet de donner une première borne inférieure.

Théorème 5. *Le problème de la fusion de deux listes triées de tailles p et k admet une borne inférieure en nombre de comparaisons définie par la relation* $\min(p, k)$.

Démonstration. À l'initialisation, le plus long chemin est de longueur $\max(p, k)$. L'algorithme peut trouver une solution lorsque le plus long chemin est de longueur $p + k = \max(p, k) + \min(p, k)$. Comme d'après la Proposition 3, une comparaison ne peut faire augmenter la longueur du plus long chemin d'au plus 1, il est nécessaire de faire au moins $\min(p, k)$ comparaisons utiles pour pouvoir atteindre cette longueur. □

Il doit être possible d'affiner ce raisonnement afin d'obtenir des bornes plus fines, mais, nous nous contenterons de cette borne, notre but étant ici de montrer un dernier exemple d'argument d'adversaire.

Analyse amortie

Dans le cadre de l'analyse amortie, nous nous intéressons à une suite de n opérations que l'on peut effectuer sur une structure de données particulière. Le terme amortie vient de l'idée que dans une suite de ces opérations, les plus coûteuses seront amorties par les moins coûteuses. C'est une notion que l'on retrouve en comptabilité lorsqu'une entreprise fait un investissement coûteux dans l'immédiat mais qui, s'il est comptabilisé sur une longue durée, comme si nous payons une partie de la totalité de sa somme tous les mois, sera amorti.

Définition 13. Soit une structure de données pour laquelle nous appliquons n opérations. Nous notons pour $i \in [n]$, son *coût réel* par c_i. Une analyse amortie consiste à associer à chacune de ces opérations un *coût amorti* $\widehat{c_i}$ de façon à avoir la propriété suivante qui est vérifiée

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i}. \tag{3.2}$$

Remarque. Cette relation n'impose pas qu'il existe une façon unique d'associée des coûts amortis à une opération. En effet, nous pourrions, par exemple, ajouter une quantité positive à chacun des coûts amortis et nous obtiendrions de nouveaux coûts pour lesquels la relation serait toujours vérifiée. De plus, cette relation montre directement que le coût amorti total donne un majorant du coût réel total de ces opérations. Si nous parvenons à démontrer que le choix des coûts amortis permet de vérifier l'équation, nous pouvons donc nous servir de l'analyse amortie pour en déduire des bornes supérieures de la totalité du coût réel de l'ensemble de ces opérations.

Par la suite, nous allons montrer deux méthodes qui permettent d'obtenir des coûts amortis valides au sens de l'Équation (3.2). La première méthode est appelée la *méthode de l'agrégat* et la deuxième est appelée la *méthode comptable*.

La méthode de l'agrégat : Cette méthode est probablement la plus simple et la plus directe pour faire une analyse amortie. Cette méthode ne nous permet pas de déterminer une borne supérieure au coût réel de l'ensemble des opérations effectuées sur la structure. Elle permet cependant d'avoir une première approche concrète d'analyse amortie. L'idée est d'associer à chacune des opérations le même coût amorti, à savoir le coût moyen du coût réel des n opérations. Plus formellement, nous associons pour tout $i \in [n]$,

$$\widehat{c_i} = \frac{\sum_{i=1}^{n} c_i}{n}.$$

Il est évident que pour ce cas l'Équation (3.2) est vérifiée puisque

$$\sum_{i=1}^{n} \widehat{c_i} = n\frac{\sum_{i=1}^{n} c_i}{n} = \sum_{i=1}^{n} c_i.$$

Voyons maintenant comment calculer le coût amorti par cette méthode sur un exemple. Nous allons nous intéresser à une structure de pile pour laquelle nous pouvons faire trois types d'opérations :

— L'empilement consiste à ajouter en haut de la pile un nouvel élément donné en paramètre.
— Le dépilement simple consiste à retirer l'élément en haut de la pile.
— Le dépilement multiple consiste à itérer d'un paramètre k l'opération de dépilement simple. Si k est plus grand que la taille de la pile, alors tous les éléments sont dépilés.

Il est facile d'obtenir des implantations de piles permettant de faire les deux premières opérations avec un coût en temps constant $\mathcal{O}(1)$. Comme l'opération de dépilement multiple consiste à itérer $\min(k, s)$ fois l'opération de dépilement simple, avec s la taille de la pile au moment où l'opération est exécutée. Nous avons que le coût en temps de cette opération est en $\mathcal{O}(\min(k, s)) = \mathcal{O}(s)$. Dans le pire cas, l'opération de dépilement multiple peut donc s'avérer onéreuse en temps de calcul. Si nous faisons n opérations à la suite en partant d'une pile vide initialement, nous avons qu'à tout moment $s \leq n$. Ainsi une première borne supérieure que l'on obtient consiste à considérer que l'on fait $\Theta(n)$ opérations de dépilement multiple. Dans ce cas, nous obtenons alors que le total des coûts en temps des n opérations est en $\mathcal{O}(n^2)$. Cette borne n'est cependant pas très fine, on peut sentir que l'opération de dépilement multiple ne peut pas être aussi coûteuse si elle est exécutée trop fréquemment, puisque la taille de la pile se retrouve en conséquence réduite. Remarquons qu'à chaque élément inséré dans la pile, cet élément ne peut être impliqué que pour au plus deux opérations, celle qui a empilé cet élément, et le dépilement simple provenant d'une des deux opérations de dépilement. De ce fait, comme nous avons vu que $s \leq n$, nous avons que le coût total en temps des n opérations est $\mathcal{O}(n)$. Ainsi le coût amorti de chacune de ces opérations est

$$\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1).$$

Il est intéressant de remarquer que dans ce cas le coût amorti de toutes ces opérations ne dépend pas de n et est constant. Du point de vue d'un utilisateur de la structure, nous pouvons alors considérer que quel que soit le nombre d'opérations que nous faisons sur cette pile, chacune de ces opérations s'effectuent en temps constant. Le coût de l'opération de dépilement multiple étant grossièrement amorti par son coût moyen.

La méthode comptable : Cette méthode consiste à maintenir un invariant pour faire en sorte que pour tout $k \in [n]$, nous avons

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} \widehat{c_i}.$$

En particulier, nous avons quand $k = n$, l'Équation (3.2) qui est vérifiée. À la différence de la méthode de l'agrégat, il est donc ici tout à fait possible d'avoir des coûts amortis différents par types d'opérations. Une façon de visualiser la méthode comptable, est de considérer que nous associons un compte en banque virtuel à notre structure de données. Lorsqu'une opération est effectuée, elle crédite ce compte d'un montant correspondant à son coût amorti. Simultanément, le compte est débité du coût réel de l'opération. L'invariant que l'on maintient revient à dire que ce compte en banque n'est jamais à découvert après chaque

opération. Lorsqu'une opération a un coût amorti plus petit que son coût réel, on dit que cette opération est *sous-facturée*. Une opération dont le coût amorti est plus grand que son coût réel est dit *sur-facturée*. En un sens, nous pouvons dire que les opérations sur-facturées payent pour les opérations sous-facturées.

Reprenons l'exemple précédent avec notre structure de pile. Nous avions fait la remarque qu'un élément dans la pile pouvait entraîner des coûts de deux façons différentes, à savoir quand il est ajouté et quand il est supprimé de la pile. Cette remarque peut nous guider pour le choix des coûts amortis de chaque opération. L'idée est de considérer que l'opération d'empilement va être sur-facturée, elle va payer son propre coût réel et payer en avance le coût réel du dépilement de l'élément qu'elle vient d'ajouter. Comme l'opération d'empilement coûte 1 et que le dépilement de cet élément coûte 1 également, le coût amorti de l'opération d'empilement est alors de 2. Comme la suppression d'un élément dans la pile a déjà été payée en avance par l'opération d'empilement, les opérations de dépilement simple et multiple n'ont alors plus rien à payer. Ces deux opérations ont donc un coût amorti de 0. Nous voyons bien ici, que pour cette attribution de coût amorti, il est impossible pour la structure d'être à découvert après chaque opération effectuée sur la structure. Nous avons donc à la fin l'Équation (3.2) qui est vérifiée. Comme le coût amorti maximum est de 2 pour toutes les opérations effectuées sur la pile, nous avons alors

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i} \leq \sum_{i=1}^{n} 2 \leq 2n.$$

Et donc, nous avons montré que le coût réel total est en $\mathcal{O}(n)$. Ce que nous venons de réaliser est très intéressant pour obtenir des bornes supérieures de la complexité d'un algorithme. Ici, nous avons démontré que le compte en banque de la structure n'est jamais à découvert. De ce fait, nous pouvons majorer le coût réel total par le coût amorti total d'après l'Équation 3.2. Il se trouve que le coût total amorti est simple à majorer ce qui nous permet d'obtenir une borne supérieure du coût réel. Ce schéma que nous venons d'utiliser peut être utilisé pour déterminer des bornes supérieures en lissant les coûts réels par les coûts amortis. Nous utiliserons, par ailleurs, la méthode comptable pour déterminer la complexité dans le pire cas de TimSort au chapitre 4.

3.2 La machine RAM

Lorsque nous souhaitons faire l'analyse de la complexité d'un algorithme, nous sommes confrontés à un problème. En effet, nous avons vu précédemment, que l'analyse d'algorithmes consistait à obtenir l'asymptotique de fonctions de coût qui permettent de, par exemple, donner une estimation du temps ou de la mémoire requise au cours de l'exécution d'une implantation de l'algorithme sur une machine réelle. On peut alors se demander en quoi la fonction choisie est pertinente, qu'est-ce qui nous permet, par exemple, dans le cas d'algorithmes de tri de considérer que le nombre de comparaisons ou que le nombre de déplacements mémoire donne une bonne estimation du temps d'exécution sur une machine réelle. La vérité est que, bien souvent lorsque l'on fait l'analyse d'algorithmes, nous considérons que son implantation est exécutée sur une machine abstraite à savoir le modèle RAM, qui est un acronyme pour Random Access

Machine.

Voyons concrètement comment fonctionne ce modèle. Une machine RAM est constituée d'une unité de logiques et d'arithmétiques ALU, qui est capable de faire toutes les opérations arithmétiques standards en une même durée de temps. Concernant sa mémoire, la machine est constituée d'un nombre illimité de registres qui sont eux-mêmes de taille infinie. Un registre peut être vu comme une case dans la mémoire permettant d'accueillir une valeur. Ce registre est également nommé avec, par exemple, un nombre entier, que l'on appelle son adresse. Un registre est donc ainsi caractérisé par un couple (**adresse**, **valeur**).

Remarque. Cette dernière considération peut être étendue à une machine où chaque registre a une taille fixée. Pour stocker un registre de plus grande taille, il suffit alors d'utiliser une collection de plusieurs registres, comme par exemple un tableau.

Cook et Reckhow [19] qui, à notre connaissance, ont posé les bases de la théorie de la complexité dans le cadre du modèle RAM, ont proposé que le temps d'accès d'un registre en lecture et en écriture soit logarithmique en la valeur du registre. L'utilisation du modèle RAM reste une bonne approximation de ce que font les ordinateurs modernes et reste donc un modèle pertinent pour analyser la complexité d'un algorithme. Il peut toutefois être intéressant de considérer des extensions de ce modèle pour affiner les résultats de ces analyses afin d'être plus proche de ces derniers.

En plus de faire des accès directs dans les registres, un programme fonctionnant sur une machine RAM peut également faire des accès indirects, c'est à dire accéder à un registre dont l'adresse est pointée par un autre registre. Le modèle RAM est fixé à un seul programme, mais il est également possible d'avoir une architecture de Von Neumann, où le programme est situé également dans la mémoire, nous parlons alors de modèle RASP. Nous avons donc avec le modèle RASP, une architecture abstraite qui s'approche du modèle des ordinateurs que nous utilisons dans la pratique, avec une mémoire à accès aléatoire.

Dans la pratique, les machines modernes ont des similitudes avec le modèle RAM mais ces dernières possèdent quelques caractéristiques supplémentaires. De ce fait, le choix des fonctions utiles de coût pour l'analyse en temps et en mémoire reste assez souvent pertinent pour comparer en complexité les algorithmes, même sur des machines réelles récentes. Nous ajoutons cependant un petit bémol à tout cela. Dans le cadre de l'analyse de la complexité, nous avons introduit la notation de Landau. Avec cette notation, nous avons également introduit une notion de variables cachées qui sont les constantes multiplicatives que nous utilisons pour borner la fonction de coûts par une autre fonction. Lorsque deux algorithmes sont de complexités similaires sous cette notation, il devient alors intéressant de regarder de plus près les valeurs de ces constantes pour comparer ces derniers. Ces constantes sont cependant dépendantes du modèle RAM. Il peut alors arriver qu'en pratique pour deux algorithmes dont on connaît les constantes cachées, qu'il y ait des surprises et que l'algorithme que nous pensions être le plus rapide, car sa constante cachée est la plus petite, soit au final plus lent que le deuxième. C'est par exemple ce qui se produit pour les algorithmes de recherche simultanée du minimum et du maximum dans une séquence et dont nous discuterons au chapitre 5. Les raisons de ces résultats surprenants proviennent alors généralement de l'incomplétude du modèle RAM concernant des caractéristiques modernes d'architectures qu'utilisent nos ordi-

nateurs actuels. Le choix de nos fonctions de coût ne serait alors plus suffisant pour comparer les algorithmes et il faudrait s'intéresser à des coûts associés à ces caractéristiques que nous ignorons dans le modèle RAM. Pour être rigoureux sur la comparaison de deux algorithmes dont les fonctions de coût d'intérêts dans le modèle RAM sont relativement du même ordre de grandeur, il nous faudrait en vérité faire l'analyse des implantations de ces derniers sur la machine utilisée en considérant que nous connaissons tout de ses caractéristiques. Bien qu'il soit difficile de procéder ainsi, nous allons dans les prochaines sections discuter de deux caractéristiques à savoir la *hiérarchie mémoire* et la *prédiction de branchement*. Nous verrons en quoi ces caractéristiques diffèrent des hypothèses considérées dans le modèle RAM et nous verrons comment affiner nos analyses en considérant de nouvelles fonctions de coût.

3.3 Les hiérarchies de mémoires

Nous allons donc commencer par discuter de la hiérarchie mémoire qui est, comme nous l'avons vu précédemment, une des caractéristiques d'architecture de nos machines actuelles. Si dans le modèle RAM, deux algorithmes font un nombre d'opérations du même ordre de grandeur, s'intéresser aux constantes cachées dans ce modèle peut s'avérer insuffisant pour les comparer. En effet, il est possible que l'algorithme qui est considéré comme le plus lent dans ce modèle soit le plus rapide en pratique sur une véritable machine. Une des raisons possibles est alors que cet algorithme gère mieux les mémoires et qu'il gagne considérablement du temps de cette façon. Pour ce genre de cas, le modèle RAM n'est donc plus suffisant pour comparer de tels algorithmes une fois implantés sur un ordinateur. Il est alors nécessaire de le compléter.

3.3.1 Concept et définition

Dans le modèle RAM, nous considérons qu'il existe une unique zone de mémoire dans laquelle est stockée un ensemble de registres. Si nous considérons que ces registres sont de tailles fixes, alors les temps d'accès dans ces derniers sont tous égaux. Cette zone constitue alors un unique niveau de mémoire. Nous avons donc dans le cas du modèle RAM, une unique zone pour de multiples registres. Dans nos machines actuelles, le fonctionnement est différent, nous avons en réalité plusieurs niveaux de mémoires pour plusieurs registres. Chaque niveau est paramétré par *une taille mémoire*, autrement dit sa capacité en quantité d'informations en bits qu'il peut stocker, et le *temps d'accès* à la lecture et à l'écriture. Pour chaque niveau, ces paramètres sont différents de tous les autres niveaux. En général, il existe une relation entre la taille mémoire et le temps d'accès. La taille mémoire d'un niveau est quasiment proportionnelle à son temps d'accès. Cela s'explique pour des raisons de coûts des matériaux utilisés qui sont plus chers pour les mémoires à accès rapide. Il y a donc moins de mémoire dont le temps d'accès est court sur une machine pour éviter que cette dernière ne soit trop chère à produire et à vendre. Le paramètre du temps d'accès implique alors qu'il existe un ordre hiérarchique entre les différents niveaux. Intuitivement, les niveaux qui ont un temps d'accès rapide, sont donc les plus petits en taille également et sont donc des ressources plus critiques. Il est alors important de savoir exploiter cette mémoire sur des données dont nous faisons très fréquemment des

accès.

Nous allons particulièrement nous intéresser ici aux niveaux de mémoires qui sont les plus proches du cœur d'un processeur puisque c'est principalement ces mémoires que nous utilisons dans la pratique lorsque nous implantons nos algorithmes. Nous allons donc dès à présent discuter de la mémoire cache et des niveaux les plus couramment existant dans la plupart de nos machines. Le processeur possède lui-même un ensemble de registres très restreint. Cet ensemble de registres forme le niveau de mémoire le plus bas possible avec le temps d'accès qui correspond à un cycle du processeur. Nous avons ensuite un premier niveau de cache, noté $L1$, à capacité faible mais rapide qui est partagé en deux zones, une zone pour mémoriser les instructions à exécuter et une zone pour les données. Les niveaux supérieurs sont ensuite en général partagés par les instructions et les données. Nous avons ainsi, pour la plupart des machines récentes, jusqu'à 3 niveaux supplémentaires $L2$, $L3$ et $L4$. Une fois atteint ce dernier niveau de cache, nous avons une mémoire bien plus grande qui est souvent la mémoire vive.

Il serait bien évidemment plus pratique d'avoir une mémoire comme pour le modèle RAM avec le meilleur temps d'accès mais c'est évidemment compliqué pour des raisons de prix. De la même façon, l'intérêt d'utiliser ce système de hiérarchie mémoire est de pouvoir gagner du temps, comparé à un modèle RAM qui n'utiliserait qu'un seul type de mémoire.

3.3.2 Gestion des accès

Nous avons dit précédemment qu'il était nécessaire d'exploiter au maximum la mémoire de plus bas niveau pour gagner du temps, voyons donc dès à présent comment le processeur gère ces différentes mémoires pour optimiser les temps d'accès.

Lorsqu'il y a un accès, par exemple, la lecture d'une variable dans un programme en cours d'exécution, voici ce qui va se passer. Dans un premier temps, le processeur va essayer de chercher cette donnée dans un de ses registres, si il ne s'y trouve pas, il va alors chercher dans la mémoire de cache $L1$. De la même façon, si cette dernière n'est pas dans le cache $L1$, il va alors chercher dans le cache $L2$. Le processeur va ainsi parcourir la mémoire en cherchant dans les niveaux supérieurs jusqu'à parvenir à le trouver. Lorsque le processeur ne parvient pas à trouver la donnée dans un niveau, nous avons alors ce que nous appelons une *erreur de cache*. Lorsqu'il y a une erreur de cache, le processeur va déplacer la mémoire vers des niveaux de cache les plus bas exploitant ainsi un principe de *temporalité*. Ce principe fait une hypothèse sur la façon dont les données sont parcourues. Cette hypothèse est qu'une donnée qui a été fraîchement utilisée a de forte chance d'être utilisée à nouveau dans un avenir proche. Un exemple de données qui sont fréquemment utilisées sont les itérateurs qui vont être très fréquemment lus et mis à jour lors de l'exécution d'une boucle, il est alors naturel de vouloir accéder rapidement à ce genre de variables en les laissant dans le cache le temps de l'exécution de cette boucle. Après une erreur, le processeur rapatrie également les données appartenant au même bloc mémoire dans les niveaux les plus bas du cache. Il faut voir ici un bloc comme un ensemble de cases voisines dans la mémoire. Procéder de cette manière permet d'exploiter alors un deuxième principe qui est celui de *localité*. Ce principe fait lui aussi une hypothèse sur la façon dont les données sont parcourues. Cette hypothèse est

que les données au voisinage d'une donnée, qui a été fraîchement utilisée, ont de fortes chances d'être utilisées dans un futur proche. Un exemple où ce genre de situations se produisent est le cas d'un parcours linéaire d'un tableau, il est alors fréquent de parcourir les éléments du premier élément jusqu'au dernier. De ce fait, par principe de localité, le premier élément ainsi que ses voisins dans un même bloc vont être chargés dans le cache puis vient le bloc suivant et ainsi de suite. Quand nous parcourons un tableau de cette façon, nous produisons le moins d'erreurs de cache possibles. Il est possible d'observer ce phénomène sur nos machines en calculant, par exemple, la somme des éléments d'un tableau en procédant en un parcours du premier au dernier élément, et en choisissant une permutation au hasard pour décrire le parcours. Nous observons, en général, que le premier est souvent plus rapide que le dernier.

Nous discutions dans la section précédente de la pertinence de nos fonctions de coût. Il existe une littérature riche d'analyse d'algorithmes qui en plus de regarder les fonctions de coûts classiques utilisées avec le modèle RAM vont choisir d'étudier le coût en nombre d'erreurs de cache. Dans le modèle du cache, il existe divers paramètres à définir comme le nombre de niveaux de caches, la façon dont on crée des blocs et bien d'autres. Il existe de nombreux modèles de caches qui étendent le modèle RAM. Ces modèles simplifient le cas réel mais permettent, d'un autre côté, de faire de l'analyse de ce genre de complexités. Le lecteur intéressé par le domaine de l'analyse d'algorithmes dans le cadre d'un modèle de cache peut, par exemple, consulter l'article de LaMarca et Ladner [31] qui s'intéressent à l'influence du cache sur les performances de divers algorithmes de tri.

3.4 Le pipelining

Au début, les processeurs fonctionnaient de manière séquentielle. Un programme consiste en une suite d'instructions, et toutes ces instructions sont exécutées dans l'ordre par le processeur. Il est intéressant de noter que l'étymologie du mot ordinateur vient en particulier de cette notion d'ordonnancement des instructions dans un programme. Pour ces processeurs, il est donc nécessaire d'attendre que l'exécution d'une instruction soit terminée avant de pouvoir commencer à exécuter la suivante. Il était donc nécessaire d'attendre un certain nombre de cycles d'horloge du processeur entre l'exécution de deux instructions, ce temps correspondant au temps d'exécution de la première instruction. L'apparition des microprocesseurs à jeu d'instructions réduits RISC a permis d'introduire l'utilisation de *pipelines* qui fait en sorte d'exécuter une nouvelle instruction à chaque cycle d'horloge. Le jeu d'instructions étant réduit en un minimum de formats, il est possible de décomposer ces dernières en plusieurs étapes. De cette façon, il n'est plus nécessaire pour le processeur d'attendre la fin de l'exécution d'une instruction avant de pouvoir commencer à exécuter la suivante. La majorité des ordinateurs construits à l'heure où sont écrites ces lignes reprennent ce modèle de pipelining dont nous allons voir le détail de fonctionnement dès à présent.

Le pipeline reprend le concept de chaîne de montages et permet de paralléliser l'exécution des instructions. Concrètement, les instructions décomposées en micro-instructions sont exécutées par une unité du processeur correspondant à un **étage** du pipeline. Chacun de ces étages fonctionne indépendamment des

autres, et il est donc possible d'exécuter plusieurs micro-instructions de plusieurs instructions consécutives dans le programme pendant un même cycle d'horloge. Nous pouvons observer sur la Figure 3.2, qu'il y a un chevauchement dans l'exécution des instructions du programme.

3.4.1 Le pipeline RISC classique

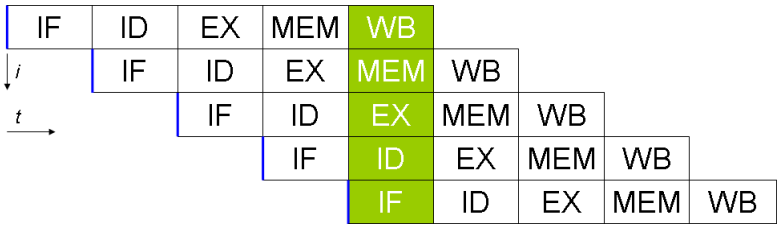


FIGURE 3.2 – Un exemple d'exécution de programme pour le cas d'un pipeline à 5 étages. L'axe des i représente l'ordre d'arrivée des instructions, tandis que l'axe t représente le temps. Par exemple au temps $t = 3$, l'instruction 1 est à l'étape d'exécution, l'instruction 2 est à l'étape de décodage de l'instruction, et l'instruction 3 est à l'étape de chargement de l'instruction. Les autres instructions sont toujours en attente. Cette image est issue de la page Wikipedia sur le pipelining.

Un exemple de processeur utilisant un pipeline de 5 étages est détaillé dans les cours de Patterson, le concepteur du processeur RISC, comme par exemple [27] et [36]. La Figure 3.2 représente une exécution d'une suite d'instructions pour ce modèle de pipeline. Voyons maintenant le détail des différents étages de ce dernier :

- **Instruction Fetch** Charge l'instruction pointée par le registre PC (Program Count) depuis la mémoire et met à jour ce pointeur. Ce registre PC doit toujours pointer sur l'adresse de la prochaine instruction à exécuter.
- **Instruction Decode** Pendant cette étape l'instruction est décodée et les opérandes sont lus dans les registres correspondants. C'est également durant cette étape que certaines conditions de branchements sont testées.
- **Execution** C'est pendant cette étape que l'opération est effectuée par l'ALU afin d'obtenir un résultat.
- **Memory Access** Dans cette étape, des opérations de lectures ou d'écritures dans la mémoire sont effectuées en fonction du type de l'instruction en cours d'exécution.
- **Write Back** Pendant cette étape, le résultat est écrit dans les registres donnés en paramètre à l'instruction.

Ce pipeline est un cas d'école et en pratique les processeurs contiennent bien plus d'étages que celui-ci. Le maximum dont nous ayons connaissance provient de l'Intel Pentium 4 Prescott avec un total de 31 étages. Les derniers processeurs d'Intel de la famille des i3, i5 et i7 semblent comptabiliser une vingtaine d'étages au moment où sont écrites ces lignes.

3.4.2 Les problèmes du pipelining

Des problèmes peuvent apparaître lors de l'utilisation d'un pipeline dans un programme. Les problèmes surgissent, par exemple, lorsqu'il y a des dépendances entre deux instructions consécutives. Il est possible d'avoir des problèmes de dépendances de données, où, par exemple, deux instructions consécutives vont écrire ou lire dans une même adresse mémoire. Un autre type de dépendances, qui nous intéressera par la suite, provient des instructions de branchement. Une instruction de branchement peut avoir à choisir la prochaine instruction après le résultat d'un test. Dans ce cas il est impossible de savoir l'instruction suivante avant l'issue de ce test.

Pour pallier ces problèmes, il existe différentes optimisations logicielles et matérielles qui peuvent être réalisées. Il est, par exemple, possible de placer une instruction au milieu de deux instructions qui sont dépendantes entre elles. L'instruction se trouvant au milieu pourra ainsi commencer à être exécutée sans avoir à attendre le résultat obtenu à la fin de l'exécution de la première instruction et va donc éviter l'apparition de *bulles*, où des étages n'ont rien à exécuter pendant une durée d'au moins un cycle du processeur. Il existe aussi des court-circuits entre les étages qui permettent d'éviter d'attendre les ré-écritures en mémoire de certaines données. Une autre optimisation est de dérouler les boucles afin d'obtenir plus d'instructions qui ne sont pas dépendantes entre-elles et d'utiliser la première méthode de réarrangement dans cette nouvelle boucle.

Dans le cas où nous avons une instruction de branchement, il y deux solutions dont nous avons connaissance. La première solution revient simplement à attendre que le test soit terminé avant de continuer. Cependant, cette solution peut s'avérer désastreuse vis-à-vis de la rentabilité du pipeline car il peut introduire un nombre important de bulles. La deuxième solution, dont nous allons discuter dans la section suivante, est d'anticiper le résultat que nous pensons être le plus probable et d'exécuter les instructions qui suivent ce résultat anticipé. Dans le cas où le résultat qui a été anticipé n'était pas le bon, le processeur doit alors annuler toutes les instructions exécutées en avance et rétablir le contexte juste avant l'instruction de branchement qui a causée cette erreur d'anticipation. Lorsqu'une erreur de ce type se produit, le temps perdu semble être à peu près équivalent au temps perdu de la première solution. L'unité dans le processeur qui permet d'anticiper le résultat d'une instruction de branchement est appelée un prédicteur de branchement et nous allons voir dans la section suivante les fonctionnements d'un ensemble de modèles de prédicteurs dont les variantes sont encore utilisées à ce jour sur nos processeurs actuels.

3.5 Prédiction de branchement

Comme vu précédemment, une des possibilités qu'a le processeur pour éviter de perdre du temps à attendre le résultat après une instruction de branchement, est d'essayer de prédire la branche qui sera exécutée. En cas d'erreur pour la prédiction du test, le processeur doit alors annuler ce qui a été exécuté en avance et recommencer à partir de l'autre branche. Quand une erreur de ce genre arrive,

le processus perd donc un temps relativement important [1] et on dit alors que le *prédicteur* utilisé par le processeur a fait une *erreur de prédiction*.

Un prédicteur est une machine à états, c'est en fonction de ces états que ce dernier prend sa décision. Par la suite, nous dirons lorsque le prédicteur anticipe que le résultat est vrai, qu'il est dans un état TAKEN, et lorsqu'il prédit que le résultat est faux, qu'il est dans un état NOT TAKEN. Dans cette section, nous ferons un parcours non-exhaustif des prédicteurs les plus connus et utilisés en pratique.

Données : Le premier élément de la suite u_0 et le nombre d'itération n
Résultat : u_n

```
1 pour i ← 1 à n faire
2     si u_{i-1} pair alors
3         u_i ← u_{i-1} / 2
4     sinon
5         u_i ← 3u_{i-1} + 1
6     retourner u_n
```

Algorithme 2 : Algorithme calculant le n-ème terme de la suite de Syracuse commençant par la valeur u_0.

Pour comprendre le fonctionnement de ces prédicteurs, rien ne vaut un exemple, et nous allons considérer leurs fonctionnements sur quelques exécutions de l'Algorithme 2 qui consiste en un calcul du n-ème terme de la suite de Syracuse paramétrée par une valeur initiale choisie u_0. La condition à la ligne 2 va ainsi produire une suite de résultats qui seront soit vrais, représenté par la lettre V, soit faux, représenté par la lettre F. Il est alors possible d'écrire cette séquence sous la forme d'un mot $w(n, u_0)$, par exemple $w(5, 42) = VFVVV$, $w(7, 1024) = VVVVVVV$ et $w(17, 52692) = VVFVVVVVFVFVVVFVF$. Pour chaque prédicteur p, nous allons regarder son nombre d'erreurs de prédiction produites à la ligne 2 que nous notons $\Upsilon_p(n, u_0)$.

3.5.1 Le prédicteur statique

Un premier prédicteur assez primitif mais qui a l'avantage de n'utiliser aucune mémoire auxiliaire est le prédicteur statique. Celui-ci choisit toujours la même branche, son état est ainsi fixé.

Exemple. *Pour le prédicteur statique en état TAKEN, nous avons* $\Upsilon_{staticV}(5, 42) = 1$, $\Upsilon_{staticV}(7, 1024) = 0$ *et* $\Upsilon_{staticV}(17, 52692) = 5$. *Pour le prédicteur statique en état NOT TAKEN, nous avons* $\Upsilon_{staticF}(5, 42) = 4$, $\Upsilon_{staticF}(7, 1024) = 7$ *et* $\Upsilon_{staticF}(17, 52692) = 12$

3.5.2 Le prédicteur 1-bit

Le prédicteur 1-bit est un prédicteur qui, comme les autres prédicteurs que nous verrons par la suite, entre dans la catégorie des prédicteurs dynamiques. Ces derniers, contrairement aux prédicteurs statiques, utilisent une mémoire auxiliaire afin de retenir des informations sur les derniers résultats d'un test

1. Ce temps peut varier en fonction de l'architecture de la machine, et de sa façon de gérer l'erreur. Pour un processeur Intel Core i7, par exemple, une erreur de prediction peut coûter environ 15 cycles d'horloge (voir [27]).

dans le but de déterminer des régularités qui pourraient permettre, une fois exploitées, d'améliorer la prédiction. Ainsi, le principe du prédicteur 1-bit est d'utiliser un compteur qui prend pour mémoire 1-bit. Ce bit peut ainsi être utilisé pour retenir le résultat du précédent test. Le prédicteur prédira alors que le prochain test produira la même issue. Ce prédicteur peut également être représenté sous la forme d'un automate comme on peut le voir pour la Figure 3.3. Les prédicteurs dynamiques sont plus flexibles que les prédicteurs statiques, ce qui leur permet de plus facilement gérer des cas où le test va produire un certain résultat avec une proportion supérieure à 50% de l'ensemble des tests effectués.

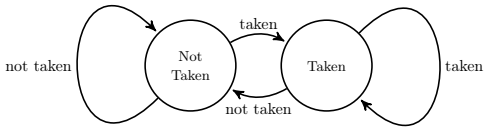


FIGURE 3.3 – prédicteur 1-bits sous forme d'automate.

Exemple. *Supposons que le prédicteur* 1*-bit soit initialement dans l'état TAKEN. La condition va produire une erreur de prédiction à partir du premier test qui sera faux, puis à chaque alternance entre un résultat vrai et un résultat faux. Nous allons représenter, en gras, les lettres correspondant aux résultats qui ont été mal prédits pour différentes valeurs de* $w(n, u_0)$*. Nous avons* $w(5, 42) = V\boldsymbol{F}\boldsymbol{V}VV$*,* $w(7, 1024) = VVVVVVV$ *et* $w(17, 52692) = VV\boldsymbol{F}\boldsymbol{V}VVVV\boldsymbol{F}\boldsymbol{V}\boldsymbol{F}\boldsymbol{V}VV\boldsymbol{F}\boldsymbol{V}\boldsymbol{F}$*. Ainsi, nous avons* $\Upsilon_{1-bit}(5, 42) = 2$*,* $\Upsilon_{1-bit}(7, 1024) = 0$ *et* $\Upsilon_{1-bit}(17, 52692) = 9$*.*

3.5.3 Les prédicteurs 2-bits

Une simple amélioration du prédicteur 1-bit consiste à doubler l'espace mémoire utilisé et ainsi d'obtenir un prédicteur utilisant 2-bits de mémoire pour stocker des informations sur les résultats précédents. Avec cette mémoire supplémentaire, le prédicteur sera en général plus robuste et permet d'éviter de changer d'état lorsque des variations apparaissent avec faible fréquence comme, par exemple, les résultats faux que l'on voit dans $w(17, 52692)$. Nous allons voir ici deux prédicteurs 2-bits que l'on retrouve souvent dans la littérature : le compteur saturé et le compteur qui fait un saut après deux erreurs de prédiction consécutives.

Compteur saturé

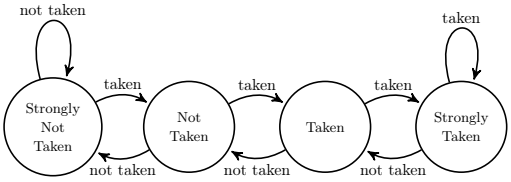


FIGURE 3.4 – prédicteur 2-bits saturé sous forme d'automate.

Le premier modèle de prédicteur 2-bits consiste en un compteur saturé, qui va associer aux valeurs 0 et 1 l'état NOT TAKEN, et pour les valeurs 2 et 3 l'état TAKEN. Soit x la valeur du compteur avant un test. Si le test est vrai, le compteur prend la valeur $min(x, x+1)$. Sinon, le compteur prend la valeur $max(x, x-1)$. Le compteur va donc s'incrémenter ou se décrémenter jusqu'à atteindre les valeurs extrêmes 0 ou 3, d'où son nom de compteur saturé. Ces valeurs extrêmes 0 et 3 sont respectivement appelé STRONGLY NOT TAKEN et STRONGLY TAKEN car on sait que l'issue prédite par ces états s'est produite au moins deux fois consécutivement. Le compteur 2-bits saturé peut également, comme pour le prédicteur 1-bit, être représenté sous forme d'automate (voir Figure 3.4).

Exemple. *En considérant que l'état initial du prédicteur est STRONGLY TAKEN. Nous reprenons les mêmes exemples utilisés pour le prédicteur* 1*-bit et la même convention qu'une lettre en gras est une des lettres qui provoque une erreur de prédiction. Nous avons alors* $w(5,42) = V\boldsymbol{F}VVV$, $w(7,1024) = VVVVVVV$ *et* $w(17,52692) = VV\boldsymbol{F}VVVV\boldsymbol{F}V\boldsymbol{F}VVV\boldsymbol{F}V\boldsymbol{F}$. *Ainsi nous avons,* $\Upsilon_{2-bit-sat}(5,42) = 1$, $\Upsilon_{2-bit-sat}(7,1024) = 0$ *et* $\Upsilon_{2-bit-sat}(17,52692) = 5$. *Nous remarquons que ces résultats sont identiques à ceux du prédicteur statique pour l'état TAKEN. Ce prédicteur reste cependant bien plus flexible, si on remplaçait tous les* V *par des* F *et tous les* F *par des* V *dans cet exemple, alors il ferait moins d'erreurs de prédictions que le statique pour l'état TAKEN.*

Compteur avec saut

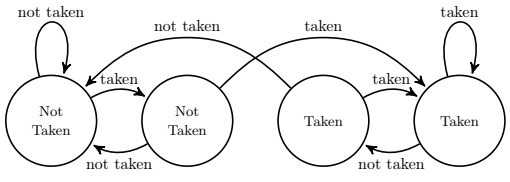


FIGURE 3.5 – prédicteur 2-bits avec saut sous forme d'automate.

Le deuxième modèle de prédicteur 2-bits est un compteur qui utilise les mêmes états que le premier, mais ne fait pas les transitions entre ces états de la même façon. Les transitions sont les mêmes sauf pour les états TAKEN et NOT

TAKEN. Il est encore une fois possible de représenter ce modèle à l'aide d'un automate et c'est ce que nous faisons dans la Figure 3.5. Les transitions qui ont été remplacées peuvent ainsi être vues comme des sauts dans la représentation en automate. L'idée de ces sauts est que le prédicteur a fait deux fois la même erreur de prédire un mauvais résultat, il y a donc de fortes chances que le prochain résultat soit encore le même, d'où l'intérêt de se mettre dans l'état le plus fort qui prédit cette dernière issue.

Exemple. *Nous allons utiliser le même exemple que le compteur saturé mais cette fois-ci en considérant que l'état initial est STRONGLY NOT TAKEN. Nous avons alors* $w(5,42) = \boldsymbol{VF}\,\boldsymbol{VV}V$, $w(7,1024) = \boldsymbol{VV}VVVVV$ *et* $w(17,52692) = \boldsymbol{VVF}VVVVV\,\boldsymbol{F}V\,\boldsymbol{F}VVV\,\boldsymbol{F}V\,\boldsymbol{F}$. *Ainsi* $\Upsilon_{2-bit-saut}(5,42) = 3$, $\Upsilon_{2-bit-saut}(7,1024) = 2$ *et* $\Upsilon_{2-bit-saut}(17,52692) = 7$.

3.5.4 Prédicteur 3-bit

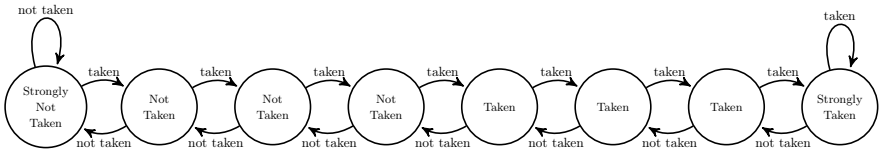


FIGURE 3.6 – prédicteur 3-bits saturé sous forme d'automate.

Il est encore une fois possible d'obtenir des prédicteurs différents en utilisant de la mémoire supplémentaire. Il existe bien évidemment un compteur 3-bits saturés fonctionnant sur le même principe que le 2-bits saturés mais qui possède deux fois plus d'états, celui-ci peut être représenté sous forme d'automate (voir Figure 3.6).

Exemple. *Reprenons notre exemple en considérant que, l'état initial est NOT TAKEN. Nous avons alors* $w(5,42) = \boldsymbol{VF}\boldsymbol{V}VV$, $w(7,1024) = \boldsymbol{V}VVVVVV$ *et* $w(17,52692) = \boldsymbol{VV}\boldsymbol{F}VVVVV\,\boldsymbol{F}V\,\boldsymbol{F}VVV\,\boldsymbol{F}V\,\boldsymbol{F}$. *Ainsi* $\Upsilon_{3-bit-sat}(5,42) = 3$, $\Upsilon_{3-bit-sat}(7,1024) = 1$ *et* $\Upsilon_{3-bit-sat}(17,52692) = 6$.

3.5.5 Prédicteur global

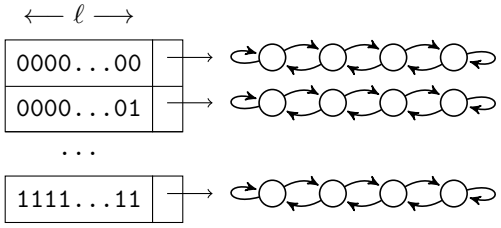


FIGURE 3.7 – Un prédicteur global avec une table d'histoires de table de taille 2^ℓ et des prédicteurs 2-bits saturés associé pour chaque histoire.

Les prédicteurs qui ont été présentés jusqu'à présent sont tous des prédicteurs locaux, c'est à dire qu'un prédicteur va être associé uniquement à une seule instruction de branchement. Cependant, ces prédicteurs ont le désavantage de ne pas tenir compte du comportement répétitif qui peut être associé à un branchement comme par exemple le fameux cycle triviale $4-2-1$ dans l'Algorithme 2 qui va produire le même motif VVF. Un autre désavantage est que ces prédicteurs ne peuvent pas prendre en compte les éventuelles dépendances qu'il peut y avoir entre les différentes instructions de branchements à l'exécution d'un algorithme.

Les prédicteurs globaux permettent de remédier à ce problème en ajoutant un niveau supplémentaire. Nous allons ici présenter un type de prédicteur global qui utilise une *table d'histoires* comme niveau supplémentaire. Une représentation de ce prédicteur est donnée dans la Figure 3.7. En premier lieu, ce prédicteur va garder en mémoire les ℓ derniers résultats des précédents tests, nous définissons ℓ comme *la taille de l'histoire* du prédicteur. De plus, nous écrivons h la trace de ces résultats, que nous appelons l'*histoire* du prédicteur. L'histoire h est alors un mot binaire où 0 signifie que l'on a eu un résultat pour lequel le branchement n'a pas été pris et 1 dans le cas contraire. Par exemple, si $\ell = 4$, $h = 1011$, alors cela signifie qu'au cours des 4 dernières instructions de branchements les résultats ont été dans l'ordre chronologique TAKEN, NOT TAKEN, TAKEN puis TAKEN. Pour chaque histoire possible, le prédicteur va alors prendre une décision, cette dernière dépendra ainsi des précédentes fois qu'elle a été rencontrée au cours de l'exécution. Concrètement la table d'histoires T possède une case pour chaque histoire possible h, à chaque case $T[h]$ est associée un autre prédicteur. Sur notre figure pour tout h, nous avons $T[h]$ qui est un prédicteur 2-bits saturé. Une conséquence de cette construction est que T est constituée de 2^ℓ cases. Sur notre figure, le prédicteur a alors besoin de $2^{\ell+2} + \ell$ bits en mémoire pour fonctionner.

Soit $h = h_1 h_2 \dots h_\ell$ l'histoire au moment où une nouvelle instruction de branchement est exécutée, nous résumons le fonctionnement de ce prédicteur en trois étapes :

1. Faire la prédiction du prédicteur $T[h]$.
2. Soit r le résultat du branchement, mettre à jour $T[h]$ en fonction de r.
3. Mettre à jour l'histoire par $h' = h_2 h_3 \dots h_\ell r$.

Exemple. *Nous reprenons encore une fois notre exemple, l'état initial est* $h = 00$ *et que pour tout* h *possible* $T[h]$ *est à l'état NOT TAKEN. Ici* $\ell = 2$ *et chaque case de la table contient un prédicteur 2-bits saturé. Nous avons alors* $w(5, 42) =$ ***VF VVV****,* $w(7, 1024) =$ ***VVV****VVVV et* $w(17, 52692) =$ ***VVF VVV VV FV F*** *VVV* ***FV F****. Ainsi* $\Upsilon_{global}(5, 42) = 4$*,* $\Upsilon_{global}(7, 1024) = 3$ *et* $\Upsilon_{global}(17, 52692) = 9$*. Dans cette exemple ce prédicteur a jusque-là fait moins bien que ceux que nous avons vus précédemment, cela peut s'expliquer car nos exemples sont trop courts. Si l'hypothèse de Syracuse est vraie, alors pour tout* $n \in \mathbb{N}$ *et pour un* m *très grand, il y a de fortes chances que ce prédicteur soit alors le meilleur parmi ceux que nous avons vus jusqu'à présent pour prédire le mot* $w(m, n)$*. En effet, après le temps d'envol de la suite, le motif se répète par le motif* VVF *qui est très bien anticipé par ce prédicteur global, contrairement aux autres que nous avons vus jusqu'à présent.*

3.5.6 Autres types de prédicteurs

Dans la pratique, il existe bien d'autres prédicteurs qui peuvent être des variantes de ceux qui ont été présentés ici, mais aussi des méthodes de prédictions différentes. Il existe, par exemple, des cas d'hybridations entre les prédicteurs globaux et locaux qui rajoutent un troisième niveau à la prédiction. Ce prédicteur va utiliser, par exemple, le prédicteur global avec table d'histoires et le prédicteur 3-bits saturés simultanément pour une même instruction de branchement. Lorsque cette instruction de branchement est exécutée, la prédiction se fait en deux étapes : une sélection du meilleur prédicteur dont le score est mesuré sur l'ensemble des fois où l'instruction a été exécutée ; la prédiction en elle-même obtenue à l'aide du prédicteur sélectionné. Il est aussi possible de faire des prédicteurs en utilisant les dernières connaissances en apprentissage artificiel. Ces prédicteurs sont peu applicables dans la pratique car beaucoup trop gourmands en ressources mais ils ont tout de même été étudiés.

De nouveaux prédicteurs sont encore créés de nos jours , comme par exemple dans le cadre d'un concours qui est réalisé et organisé par le journal JILP, tous les deux ans. Ce concours réuni des membres de compagnies réputées dans l'architecture et la fabrication de microprocesseurs. Le but de ce concours est de proposer sa propre implantation de prédicteurs de branchements. Ce concours présente plusieurs catégories, comme par exemple celle du prédicteur qui est le plus rapide sous des contraintes de mémoires, ceux qui utilisent l'apprentissage artificiel et bien d'autres. Ce concours est une vraie mine d'informations quant aux prédicteurs qui pourraient être utilisés dans l'avenir, le lecteur intéressé par ce dernier est invité à regarder cette référence [4]

3.5.7 Association d'une chaîne de Markov à un prédicteur :

Nous allons maintenant voir, comment nous pouvons associer une chaîne de Markov aux prédicteurs locaux que nous avons introduits dans cette section. L'approche d'utiliser des chaînes de Markov pour analyser des algorithmes sur leur nombre d'erreurs de prédiction a déjà été réalisée dans le passé, cela a, par exemple, été fait dans les articles [28] et [34]. Les prédicteurs de branchements sont des machines à états avec transition entre ces derniers, si nous rajoutons une probabilité de prendre les transitions entre ces états, il est alors assez évident que nous avons défini une chaîne de Markov.

Si nous nous plaçons dans un cas idéal, nous pouvons supposer qu'à chaque fois qu'une instruction de branchements est exécutée, la probabilité que le résultat du test soit vrai est toujours égale à une certaine quantité $p \in [0, 1]$. De cette façon, la probabilité de prendre une transition vers un état de type TAKEN est toujours p, tandis que la probabilité de prendre une transition vers un état de type NOT TAKEN est toujours $1 - p$. La chaîne de Markov $\mathcal{M}$, associée à un prédicteur dynamique va donc consister à prendre le même ensemble d'états que le prédicteur. Soit deux états x et y dans cet ensemble, si il existe une transition de x vers y dans le prédicteur, quand le résultat du test à l'instruction de branchement associée au prédicteur est vrai, alors $\mathcal{M}(x, y) = p$. S'il existe une transition de x vers y dans le prédicteur, quand le test est faux, alors $\mathcal{M}(x, y) = 1 - p$. Nous avons déjà vu dans l'introduction sur les chaînes de Markov dans la section 2.2.6 du chapitre 2, la représentation sous forme

d'automate de la chaîne de Markov du prédicteur 1-bit qui est donnée dans la Figure 2.1. Nous donnons, par ailleurs, dans la Figure 3.8, la représentation sous forme d'automate de la chaîne de Markov du prédicteur 2-bits saturé.

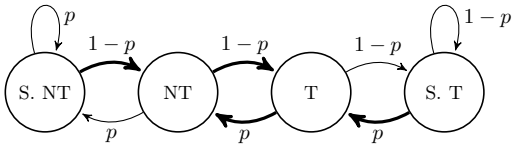


FIGURE 3.8 – La chaîne de Markov associée au prédicteur 2-bits saturé. Les transitions en gras correspondent à des erreurs de prédiction lorsqu'elles sont prises.

Pour tous les prédicteurs locaux que nous avons vus jusqu'à présent, il est évident que leurs chaînes de Markov sont irréductibles et apériodiques, de ce fait ils convergent vers une distribution stationnaire unique de par l'application du Théorème 3 et ce, indépendamment de la distribution initiale. Pour déterminer ces distributions, il suffit de déterminer le noyau de $\mathcal{M}^T - Id$ qui dépend de p. Nous avons déjà fait ce calcul pour le cas du prédicteur 1-bit et nous ne pensons pas qu'il est nécessaire de détailler les calculs pour les autres prédicteurs. Sous la distribution stationnaire, il est cependant intéressant de connaitre la probabilité de faire une erreur de prédiction. Ces probabilités sont directement données ci-après avec μ_1 qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 1-bit, μ_2 qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 2-bits saturé, μ_2' qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 2-bits avec sauts, et enfin μ_3 qui donne la probabilité de faire une erreur de prédiction pour le prédicteur 3-bits saturé. Nous avons ainsi

$$\mu_1(p) = 2p(1-p), \tag{3.3}$$

$$\mu_2(p) = \frac{p(1-p)}{1-2p(1-p)}, \tag{3.4}$$

$$\mu_2'(p) = \frac{2p^2(1-p)^2 + p(1-p)}{1-p(1-p)}, \tag{3.5}$$

et enfin

$$\mu_3(p) = \frac{p(1-p)(1-3p(1-p))}{1-2p(1-p)(2-p(1-p))}. \tag{3.6}$$

Pour étendre le modèle RAM qui considère que le coût d'une instruction de branchement est toujours le même quel que soit le contexte dans lequel cette dernière est exécutée, il peut être intéressant d'étudier une nouvelle fonction de coût qui est le nombre total d'erreurs de prédiction qu'un algorithme peut effectuer au cours de son exécution. Bien entendu, ce nombre est dépendant du modèle de prédiction utilisé et il est donc important de fixer ce dernier pour pouvoir faire une étude. Les équations que nous venons de donner permettent de faire l'analyse de ce coût pour les modèles de prédicteurs locaux. Nous verrons au cours du chapitre 5 que nous pouvons aussi, dans certain cas, y parvenir

pour le prédicteur global avec table d'histoires. Bien que les prédicteurs utilisés en pratique sont différents des modèles que nous avons présentés, il s'agit bien souvent d'améliorations de ces derniers. Nous pouvons supposer que le choix d'un prédicteur est justifié par des résultats après une série de benchmarks pour vérifier leurs bons fonctionnements. Il n'est pas garanti que ces derniers fonctionnent toujours mieux que les modèles que nous avons présentés au cours de ce chapitre, mais nous pouvons espérer que ce sera souvent le cas ou bien que la différence de performance ne devrait pas être trop grande en nombre d'erreurs de prédiction. De ce fait, analyser le nombre d'erreurs de prédiction sur les modèles simples que nous avons vus dans ce chapitre devrait nous donner une bonne idée sur la performance réelle d'un algorithme une fois implanté sur un véritable ordinateur.

3.5.8 La simulation

Comme nous venons de le dire précédemment, les performances des prédicteurs utilisés en pratique ont tendance à être meilleures que les modèles que nous avons présentés dans cette section. Il peut cependant être intéressant de pouvoir vérifier nos résultats théoriques en faisant de la simulation de ces prédicteurs. De cette façon, lorsque nous annonçons un résultat pour un prédicteur en particulier, il nous sera possible de vérifier ce dernier en simulant ce-dit prédicteur.

C'est pour cette raison que nous avons développé une petite librairie Python pour pouvoir simuler ces prédicteurs et savoir exactement le nombre d'erreurs de prédiction qu'ils peuvent effectuer en exécutant des implantations en python de nos algorithmes. La librairie est donnée en Annexe A et repose sur la classe Predictor qui peut être étendue pour pouvoir créer ses propres prédicteurs. Les prédicteurs implantés sont le statique qui prédit toujours faux, le 1-bit, le 2-bits saturé, le 2-bits avec sauts, le 3-bits saturés et le prédicteur global avec une table des historiques qui associe un des prédicteurs locaux précédemment cités.

La simulation présente un autre avantage par rapport à l'utilisation de benchmarks pour tester nos implantations. Il n'est pas toujours nécessaire de conserver la structure de donnée sur laquelle travaille un algorithme. La simulation est alors moins gourmande en mémoire, ce qui est un avantage si nous voulons tester notre algorithme pour de très grandes entrées. Par exemple, si nous voulons tester un algorithme naïf, mais optimal d'après notre argument d'adversaire, de recherche d'un minimum dans une séquence. L'algorithme en question va simplement parcourir chaque élément de la séquence et retenir au fur et à mesure le minimum qu'il a rencontré dans une variable min. Pour cet algorithme, il est facile de simuler la probabilité qu'il ait à changer la valeur de sa variable au prochain élément rencontré. Dans le cadre de l'analyse de la prédiction de branchement, simuler cette probabilité nous est suffisant. Ainsi, si nous voulons simuler cet algorithme pour la mesure du nombre d'erreurs de prédiction, il nous suffit de stocker dans une variable min le minimum rencontré jusqu'à présent. À chaque étape, nous générons un nouvel élément x au hasard qui pourrait être dans la vraie séquence, et nous comparons x avec min. Les comparaisons effectuées sont ainsi les mêmes avec même probabilité sauf que dans le cas où nous simulerions entièrement l'algorithme il nous faudrait une mémoire de taille n, alors qu'ici nous n'avons besoin que d'une mémoire de taille constante. Avec cette méthode, il est alors possible d'aller au delà des tailles limites de nos

mémoires, ce qui nous permet d'obtenir les constantes cachées de nos complexités avec une plus grande précision. De plus, comme nous réduisons l'utilisation mémoire, nous pouvons également gagner en temps d'exécution de la simulation, dans notre exemple les variables utilisées tiennent dans des registres du processeur.

Chapitre 4

TimSort

Sommaire

Dans ce chapitre, nous allons discuter du problème du tri, qui consiste à trouver des méthodes pour trier efficacement une séquence. Il s'agit d'un des problèmes les plus connus dans le domaine de l'algorithmique qui trouve de nombreuses applications. Les algorithmes de tris sont souvent utilisés comme des pré-traitements à effectuer sur un ensemble de données avant d'appliquer d'autres opérations. De ce fait le problème du tri a été étudié très tôt et ce dès l'apparition des premiers ordinateurs, voire même depuis le concept des machines de Turing. Parmi les algorithmes connus, nous pouvons, par exemple, citer le tri fusion inventé par Von Neumann en 1945 ou bien encore le tri rapide inventé par Hoare en 1961. Le problème du tri a donc été longuement étudié et la littérature à son sujet est ainsi très riche. Nous démontrerons que si la séquence à trier a en tout n éléments, alors un algorithme qui n'utilise que des comparaisons entre les éléments pour obtenir la séquence triée doit faire dans le pire cas au moins $\Omega(n \log n)$ comparaisons pour y parvenir. Il existe de nombreux algorithmes qui atteignent cette borne, dont notamment le tri fusion qui est ainsi optimal pour le nombre de comparaisons.

Il est alors étonnant de voir émerger en 2001 un nouvel algorithme de tri, qui de plus a fini par être utilisé dans de nombreux langages connus comme Java et Python. Cet algorithme se nomme TimSort, et a été développé à la base pour remplacer la méthode de tri qui était utilisée jusqu'alors dans la librairie standard du langage Python. Ce chapitre se concentre particulièrement sur ce nouvel algorithme qui est pourtant de plus en plus populaire. Nous présentons donc cet algorithme dans son fonctionnement et nous essayons de comprendre les raisons qui ont poussé les développeurs à choisir ce tri plutôt qu'un autre alors qu'il en existe pourtant une myriade.

Ce chapitre est constitué de trois parties. Dans un premier temps, nous introduisons le problème du tri. Nous montrons des premiers algorithmes connus qui offrent une solution à ce problème, puis nous démontrons la borne inférieure annoncée précédemment. Dans un deuxième temps, nous nous intéressons à la famille des algorithmes adaptatifs dont TimSort fait partie. Il s'agit d'algorithmes qui sont particulièrement efficaces pour trier des séquences qui sont déjà presque triées. Dans un troisième et dernier temps, nous discutons de TimSort. Nous allons voir les principes de fonctionnement de ce dernier et nous y donnons une démonstration de sa complexité dans le pire cas.

Remarque. La rédaction de ce chapitre précède des nouveaux travaux que nous avons effectués sur TimSort. Nous y ferons des références à travers différentes remarques ainsi que dans la conclusion.

4.1 Problème du tri

4.1.1 Définition

Avant de pouvoir expliquer ce qu'est l'algorithme TimSort, il est important de donner une définition de ce qu'est le problème du tri. En entrée, nous avons une séquence $X = \langle x_1, x_2, ..., x_n \rangle$ provenant d'un ensemble $(E, \leq)$ muni de l'ordre total $\leq$. Le but est d'obtenir en sortie une permutation $\sigma \in \mathfrak{S}_n$ telle que $\forall i \in [n-1], x_{\sigma(i)} \leq x_{\sigma(i+1)}$. Un algorithme qui donne une solution à ce problème est alors appelé un algorithme de tri. Bien souvent en pratique, il est plus intéressant d'obtenir la séquence triée $Y = \langle x_{\sigma(1)}, \ldots, x_{\sigma(n)} \rangle$. La plupart (si ce

n'est tous) des algorithmes de tri qui seront étudiés par la suite procéderont de cette façon. Il n'y a aucune difficulté à transformer ces algorithmes pour obtenir la permutation σ.

4.1.2 Quelques exemples d'algorithmes de tri

Il existe deux grandes catégories d'algorithmes de tri, nous allons voir quelques exemples pour chacune d'elles.

Les tris par comparaisons

La première catégorie d'algorithmes de tri, consiste à utiliser exclusivement la comparaison entre les éléments pour déterminer leur ordre dans la séquence triée. Cette catégorie est la plus utilisée dans la conception de librairies standards puisqu'elle permet de trier des données sans avoir de connaissances a priori dessus. Une implantation d'un algorithme de tri par comparaisons pourra ainsi être écrite sous la forme d'un *template*, laissant à l'utilisateur le choix de la fonction de comparaison qui permettra de trier ses données.

Les algorithmes naïfs qui entrent dans cette catégorie sont le *tri par bulle*, le *tri par insertion*, le *tri par sélection* et bien d'autres. Ces algorithmes, bien qu'ils soient simples et intuitifs, souffrent d'une complexité trop grande, dans le pire cas comme dans le cas moyen, en $\Theta(n^2)$. L'algorithme 3 décrit l'algorithme du tri par insertion que nous donnons à titre d'exemple.

Données : Une séquence X de taille n.
Résultat : X triée.
1 **pour** $i \leftarrow 1$ **à** n **faire**
2 $position \leftarrow i$
3 **tant que** $position > 1$ ***et*** $X[position] < X[position - 1]$ **faire**
4 *Échanger* $X[position]$ *avec* $X[position - 1]$
5 $position \leftarrow position - 1$

Algorithme 3 : Tri par insertion

Il existe néanmoins des algorithmes de tri par comparaisons qui sont plus efficaces comme par exemple le **tri fusion** et le **tri par tas** ayant une complexité dans le pire cas comme dans le cas moyen en $\mathcal{O}(n \log n)$, mais aussi le **tri rapide** qui a dans le pire cas une complexité en $\mathcal{O}(n^2)$ et dans le cas moyen une complexité en $\mathcal{O}(n \log n)$. En utilisant l'algorithme pour calculer la médiane en temps linéaire, dont on se servira comme pivot, il est possible d'avoir une version du tri rapide qui est $\mathcal{O}(n \log n)$ dans le pire cas. Cependant cette solution n'est pas utilisée en pratique à cause des coûts trop importants qui apparaissent dans les constantes de la notation $\mathcal{O}$. Voir [20, 10.3].

Les tris exploitant des informations sur l'entrée

En plus des algorithmes qui utilisent exclusivement des comparaisons pour obtenir une séquence triée, il existe des algorithmes qui s'affranchissent de cette règle en utilisant des connaissances *a priori* sur la séquence en entrée.

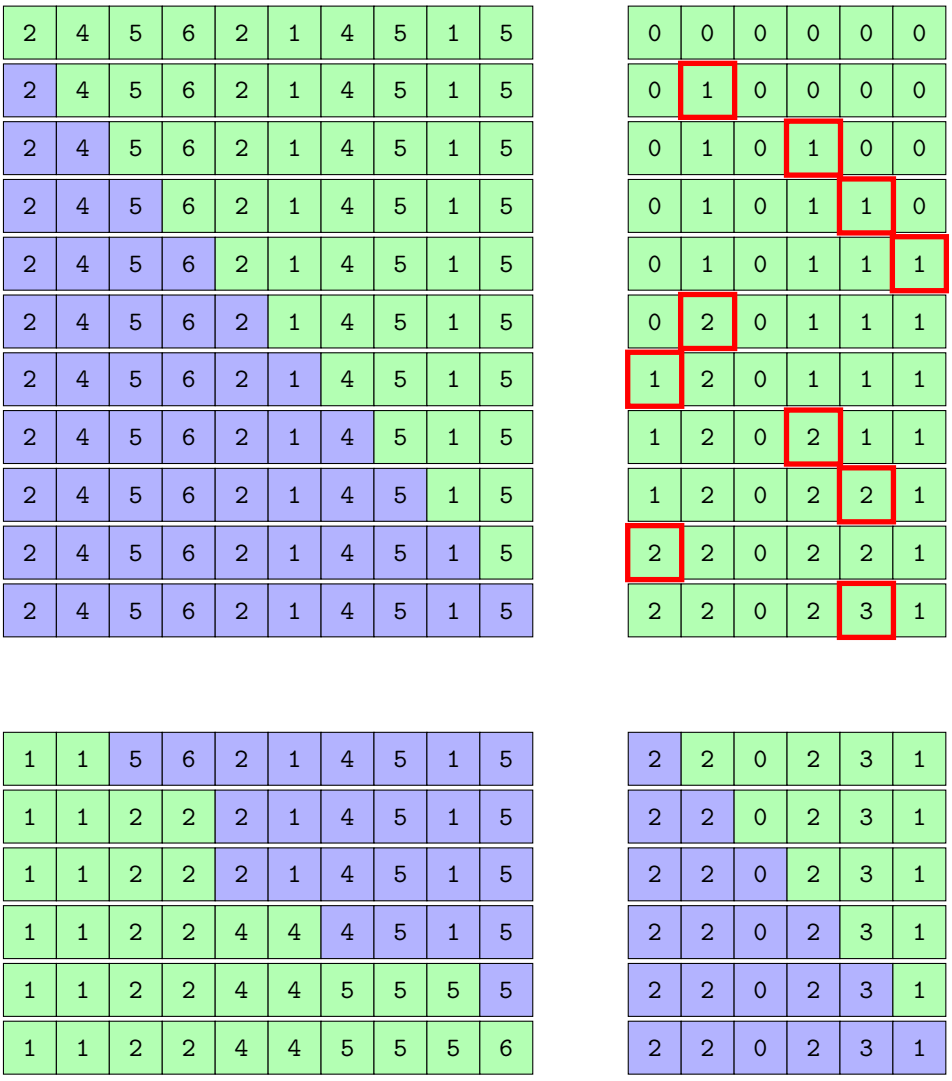


FIGURE 4.1 – Une trace d'exécution de la première étape de comptage du tri par dénombrement.

Par exemple le *tri par dénombrement* part de l'hypothèse que le nombre d'éléments distincts dans la séquence d'entrée est inférieur à une certaine valeur $m \in \mathbb{N}$, et qu'il existe une application injective (qui les numérote) simple à calculer de ces éléments vers l'ensemble $[m]$. Il suffit ensuite de compter dans un tableau de taille m combien de fois l'image d'un élément apparaît dans la séquence d'entrée et ainsi d'obtenir la sortie depuis ce décompte. La complexité de cet algorithme est alors en $\mathcal{O}(n + m)$ puisqu'il suffit de faire une première étape qui consiste à parcourir la séquence d'entrée, puis une seconde étape qui consiste cette fois-ci à parcourir la séquence de compteurs. La Figure 4.1, donne un exemple d'exécution de l'algorithme pour la séquence d'en-

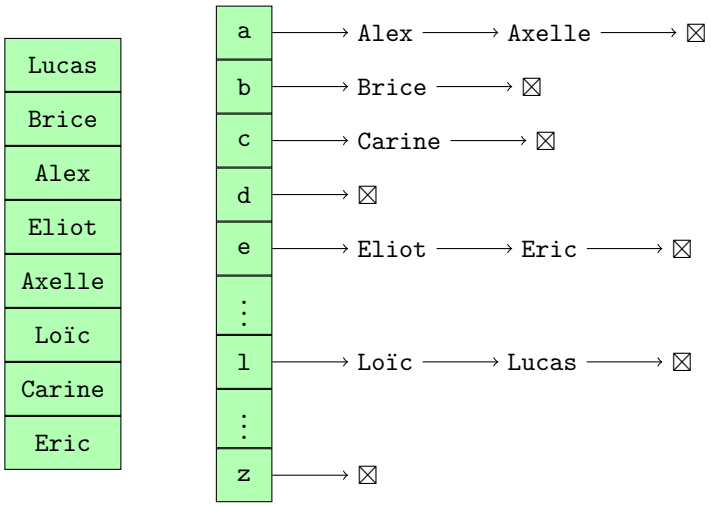


FIGURE 4.2 – Cette figure illustre l'exemple d'un tri par paquets, où le but est de trier des individus par leurs prénoms. Chaque paquet correspond à la première lettre du prénom de l'individu, il existe donc un ordre naturel entre tous les paquets. Il ne reste plus qu'à concaténer chaque paquet en partant de la lettre 'a' pour obtenir la séquence d'individus triée dans l'ordre lexicographique.

trée $\langle 2, 4, 5, 6, 2, 1, 4, 5, 1, 5\rangle$. Comme on ne trie que des entiers dans cet exemple, l'application injective utilisée n'est autre que l'identité et $m = 6$.

Le *tri par paquets* utilise des sous-séquences, que l'on nommera des paquets, et dont le nombre $m \in \mathbb{N}^+$ est choisi arbitrairement. Soit $X = \langle x_1, \ldots, x_p\rangle$ et $Y = \langle y_1, \ldots, y_q\rangle$ deux paquets, X est plus petit que Y si pour tout couple $(x, y) \in X \times Y$, nous avons $x < y$. À l'aide de cette relation, l'algorithme impose qu'il existe un ordre entre ces paquets. Chaque élément de la séquence sera alors ajouté dans un paquet de façon à conserver l'ordre pré-établi entre les différents paquets. Attribuons un numéro à chacun de ces paquets, et choisissons d'avoir les paquets par ordre croissant de leurs numéros, c'est-à-dire que le paquet numéro i est plus petit que le paquet numéro j si et seulement si $i < j$. Il suffit ensuite de trier chacun de ces paquets à l'aide d'un algorithme de tri auxiliaire comme par exemple le tri par insertion. Il ne reste plus qu'à concaténer les paquets dans l'ordre croissant de leurs numéros. En effet comme nous avons imposé un ordre sur ces paquets, la concaténation est suffisante pour obtenir la séquence triée. Si l'on doit trier par exemple des individus par leurs noms, nous pouvons alors choisir $m = 26$, et avoir la sous-séquence numéro 1 qui correspond aux individus dont le nom commence par la lettre 'a', la sous-séquence numéro 2 qui correspond aux individus dont le nom commence par la lettre b et ainsi de suite... Une trace d'exécution de cet exemple a été illustrée dans la Figure 4.2. L'exemple d'implantation le plus courant du tri par paquets concerne le tri de nombres réels dans l'intervalle $[0, 1]$. Dans ce cas, nous disposons de m paquets numérotés de 0 jusqu'à $m-1$, et pour un élément $x \in [0, 1]$ nous l'ajoutons alors à la sous-séquence numéro $\lfloor xm\rfloor$. Lorsque tous les éléments ont été ajoutés dans

un paquet, nous procédons à un tri de tous les paquets à l'aide d'un algorithme naïf en $\mathcal{O}(n^2)$. Pour ce dernier cas si l'on considère que chaque élément de la séquence d'entrée est distribué uniformément et indépendamment des autres, pour $m = n$, il est possible de prouver que la complexité de l'algorithme est en $\mathcal{O}(n)$ comparaisons. Le principe de la preuve consiste à montrer qu'il y a avec forte probabilité peu de valeurs à insérer dans chaque paquet. La preuve repose principalement sur un calcul assez direct de l'espérance du nombre de comparaisons. Voir [20, 9.4] pour les détails de la preuve.

4.1.3 Quelques propriétés sur les algorithmes de tri

Il existe de nombreux algorithmes qui répondent au problème du tri, certaines de leurs implantations possèdent des propriétés qu'il peut être intéressant de considérer lorsque l'on doit choisir un algorithme de tri en fonction de nos contraintes.

Sur place : Un algorithme trie *sur place* si la mémoire auxiliaire utilisée est en $\mathcal{O}(1)$. L'intérêt est évidemment de gagner de la place mémoire, ce qui peut être important dans des systèmes où la mémoire est restreinte comme par exemple dans le domaine du système embarqué. Parmi les algorithmes qui trient sur place, on peut trouver des implantations du tri par sélection, du tri par insertion, du tri par bulle et du tri par tas. À notre connaissance, il n'existe pas d'implantation de TimSort qui a cette propriété.

Stable : Un algorithme de tri est dit *stable* si l'ordre initial est conservé en cas d'égalité de deux éléments dans la séquence d'entrée. Les algorithmes possédant cette propriété sont très utiles pour trier une séquence d'objets sur plusieurs critères différents. Par exemple, nous pouvons imaginer le cas d'un répertoire d'individus qui sont triés dans l'ordre alphabétique de leurs noms puis ensuite de leurs prénoms. Il suffit alors dans cet exemple d'utiliser un tri stable pour trier les individus premièrement par leurs prénoms, puis ensuite par leurs noms. Il existe des variantes stables de la plupart des algorithmes de tri classiques de la littérature, nous verrons par la suite que l'algorithme TimSort est stable.

Hybride : Un algorithme de tri est dit *hybride* s'il fait appel à d'autres algorithmes de tri pour résoudre des sous-problèmes de tri. En général, l'hybridation d'un algorithme de tri par un autre permet de le rendre plus rapide. Il existe de nombreuses variantes du tri rapide et du tri fusion qui utilisent le tri par insertion afin de trier des petites portions de la séquence d'entrée. Nous verrons également par la suite que l'algorithme TimSort, qui est le sujet de ce chapitre, est également hybride.

4.1.4 Borne inférieure pour les tris par comparaisons

Le problème du tri est l'un des problèmes pour lequel il n'est pas trivial de déterminer une borne inférieure du nombre de comparaisons nécessaires à effectuer dans le pire cas, bien que la solution soit désormais classique. Il est tout d'abord nécessaire d'introduire une formalisation qui va nous permettre de définir tous les algorithmes de tri par comparaisons. Pour ce faire, il faut

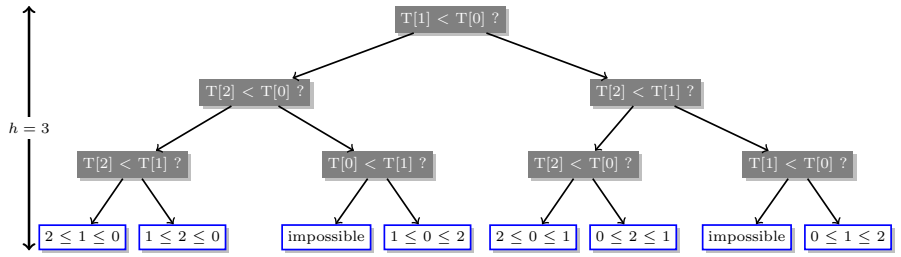


FIGURE 4.3 – L'arbre de décision du tri par bulle pour le tri de séquences de taille 3. (Merci à Cyril Nicaud pour la figure)

introduire un modèle d'arbres de décision. Ce niveau d'abstraction va en outre nous permettre d'obtenir une borne inférieure concernant ces algorithmes.

Arbres de décision

Il est possible de décrire tous les algorithmes de tri par comparaisons à l'aide du modèle d'arbres de décision. Un arbre de décision est un arbre binaire dont chaque nœud interne est étiqueté par une question qui possède un nombre borné de réponses possibles. Dans notre contexte, ces questions seront toujours des comparaisons entre deux éléments de la séquence d'entrée. Chaque feuille est étiquetée par l'unique permutation vérifiant toutes les inéquations du chemin allant de la racine à la feuille. Un exemple d'arbre de décision pour le tri par bulle est montré Figure 4.3. Un algorithme de tri défini par un arbre de décision va au cours de son exécution suivre un des chemins de l'arbre de la racine vers une feuille. La hauteur de cette feuille correspond alors au nombre de comparaisons que l'algorithme effectuera pour trier la séquence. La complexité dans le pire des cas de l'algorithme est donc la hauteur de l'arbre.

Théorème 6. *Tout algorithme de tri par comparaisons doit dans le pire cas effectuer au moins* $\Omega(n \log n)$ *comparaisons.*

Démonstration. Prenons l'arbre de décision correspondant à un algorithme de tri par comparaisons quelconque. Le pire cas de cet algorithme correspond à la hauteur de la feuille enfouie le plus profondément possible dans l'arbre. Soit h la hauteur de cette feuille. Comme il y a $n!$ permutations possibles en sortie, il y a également au moins $n!$ feuilles dans l'arbre de décision. Soit x le nombre de feuilles de l'arbre, nous avons donc $n! \leq x$. Il ne peut pas y avoir également plus de 2^h feuilles dans l'arbre de décision, soit $x \leq 2^h$. En combinant toutes ces propriétés, nous obtenons $2^h \geq x \geq n!$ ce qui implique que $h = \Omega(n \log n)$. □

Tout algorithme de tri par comparaisons qui parvient à atteindre cette borne dans le pire cas est alors asymptotiquement *optimal*. Parmi ces algorithmes, on peut trouver le tri fusion et le tri par tas.

Remarque. L'algorithme du tri rapide classique (où le pivot est choisi uniformément au hasard) n'est pas optimal dans le pire cas, il est possible de trouver une exécution où l'algorithme a besoin de $\Omega(n^2)$ comparaisons. Cependant il reste très utilisé en pratique et s'est souvent montré plus rapide que le tri fusion

sur des entrées similaires pour des machines récentes. Rappelons également que le tri fusion se comporte relativement bien en moyenne avec une complexité en $\mathcal{O}(n \log n)$.

4.2 Les algorithmes de tri adaptatif

Il est, a priori, impossible d'avoir un algorithme de tri qui fasse mieux dans le pire cas que $\mathcal{O}(n \log n)$ comparaisons. Cependant, il est possible en pratique de faire moins de comparaisons pour les meilleurs cas. Par exemple, pour le tri par insertion, il est bon de remarquer que le nombre de comparaisons nécessaires à l'insertion du prochain élément correspond au nombre d'éléments plus grands qui ont été placés avant cet élément dans la séquence d'entrée. Le nombre de comparaisons nécessaires pour trier la séquence d'entrée avec le tri par insertion va donc correspondre au nombre de couples d'éléments qui seront placés dans le mauvais ordre par rapport à la séquence triée. Ce nombre de couples peut être considéré comme du désordre dans la séquence d'entrée. Par conséquent, il faut remarquer que, moins il y a de désordre, plus rapidement sera triée la séquence par le tri par insertion. Il peut être intéressant d'analyser les algorithmes de tri par comparaisons en regardant également divers paramètres, qui mesurent le désordre, sur la séquence d'entrée et non uniquement sa taille.

4.2.1 Définitions

Un algorithme de tri est dit *adaptatif* si il est asymptotiquement optimal dans le pire cas, c'est à dire qu'il a une complexité en $\mathcal{O}(n \log n)$, et qu'il est capable de trier la séquence d'entrée plus rapidement si celle-ci est déjà *partiellement triée*.

Pour pouvoir évaluer les performances d'un algorithme de tri adaptatif, il est donc nécessaire de savoir comment déterminer à quel degré la séquence d'entrée est partiellement triée. Nous allons voir qu'il y a plusieurs façons pertinentes de mesurer le taux de désordre dans une séquence.

Exemples de mesures de désordre

Nous nous intéresserons par la suite à l'idée de pouvoir donner une quantification de l'ordre et du désordre d'une séquence. Il nous faut donc des fonctions de *mesure*, et la première intuition est que plus la valeur de la mesure d'une séquence est grande et plus celle-ci est chaotique. Nous allons voir dès à présent des exemples de fonctions de mesures naturelles.

Nombre d'inversions : Une inversion dans une séquence $X = \langle x_1, \ldots, x_k \rangle$ est un couple $(i, j) \in [k]$ tel que $i < j$ et $x_i > x_j$. Le nombre d'inversions $inv(.)$ correspond à une mesure de désordre. Il a été vu précédemment que le nombre d'inversions correspond au nombre de comparaisons effectuées par le tri par insertion.

Nombre d'éléments échangeables : On dit que l'on fait un échange des éléments x_i et x_j de la séquence $X = \langle x_1, \ldots, x_i, \ldots, x_j, \ldots, x_k \rangle$ lorsque l'on transpose ces deux éléments pour obtenir une nouvelle séquence $Y = \langle x_1, \ldots, x_j, \ldots$

$, x_i, \ldots, x_k \rangle$. Le nombre d'éléments échangeables $exc(.)$ d'une séquence X est le nombre minimum d'éléments qu'il est nécessaire d'échanger dans la séquence pour que celle-ci soit triée. Knuth [30] a démontré qu'il existe un lien entre $exc(X)$ et le nombre de cycles de la permutation $\sigma(X)$ par la relation suivante $exc(X) = |X| - \mathbf{cyc}(\sigma(X))$.

Nombre d'éléments supprimables : Le nombre d'éléments supprimables $rem(.)$ d'une séquence X est le nombre minimum d'éléments qu'il est nécessaire de retirer de la séquence pour que celle-ci soit triée.

Nombre de runs ascendants : Un run ascendant d'une séquence X est une sous-séquence contiguë maximale et qui est triée dans l'ordre croissant. Le nombre de runs $run(.)$ correspond à une mesure de désordre. Nous verrons par la suite que cette mesure est assez importante puisqu'elle est partiellement liée à la mesure qui est exploitée par TimSort.

Exemple. *Posons* $X = \langle 5, 6, 1, 7, 3, 2, 9, 8, 4 \rangle$. *Nous avons plusieurs inversions dans* X, *comme par exemple les couples* $(5, 1)$, $(5, 3)$, $(5, 2)$, $(5, 4)$ *et bien d'autres. Au total nous pouvons montrer qu'il y a* $inv(X) = 15$ *inversions. La décomposition en cycles de* X, *ici* $X \in \mathfrak{S}_9$, *est donnée par la séquence* $\big\langle (5, 3, 1), (2, 6), (7, 9, 4), (8) \big\rangle$. *Nous avons donc par la relation de Knuth que* $exc(X) = 9 - 4 = 5$ *échanges à faire pour obtenir une séquence triée. De plus, il est possible à l'aide de la décomposition en cycle de trouver une suite de* 5 *échanges, le but étant de casser les cycles de taille supérieure ou égale à* 2 *en échangeant un élément dans un cycle avec l'un de ses voisins. On peut donc par exemple échanger* 1 *avec* 5, *puis* 5 *avec* 3, *puis* 2 *avec* 6, *puis* 4 *avec* 7, *puis enfin* 7 *avec* 9. *À l'aide d'un programme, il est possible de déterminer qu'une plus longue sous-séquence ascendante de* X *est* $\langle 5, 6, 7, 8 \rangle$, *et donc* $rem(X) = 5$. *L'ensemble des runs de* X *est* $\{\langle 5, 6 \rangle, \langle 1, 7 \rangle, \langle 3 \rangle, \langle 2, 9 \rangle, \langle 8 \rangle, \langle 4 \rangle\}$, *ainsi* $run(X) = 6$.

4.2.2 Propriétés générales des mesures de désordre

Mannila [33] a posé les fondements de l'analyse des algorithmes de tri adaptatif en définissant les axiomes des *mesures de désordre*. C'est lui qui a introduit cette notion en proposant un ensemble de cinq propriétés que ces fonctions doivent respecter. Ces propriétés sont vérifiées par chacune des mesures qui ont été vues dans la section précédente, à condition de faire quelques translations pour respecter la première règle, comme c'est le cas pour la mesure *run*.

Formellement, une mesure de désordre est une fonction $m : E^+ \to \mathbb{N}$ qui vérifie les propriétés suivantes, pour $X = \langle x_1, \ldots, x_k \rangle, Y = \langle y_1, \ldots, y_p \rangle \in E^+$: [1]

- m(X) = 0 si X est triée. Autrement dit le désordre d'une séquence triée est nulle.
- Si $k = p$ et $x_i < x_j \Leftrightarrow y_i < y_j$, alors $m(X) = m(Y)$. La mesure d'une séquence ne dépend donc que de l'ordre qu'il existe entre ses éléments. Cette propriété permet de restreindre aux permutations l'étude des mesures d'ordre partiel.
- Si X est une sous-séquence de Y, alors $m(X) \leq m(Y)$. Si l'on considère les actions utilisées pour trier la séquence Y, alors la restriction de ces actions aux éléments de X permettent de trier X.

1. On rappelle que l'opérateur $\cdot$ est la concaténation entre deux séquences.

— Si tout élément de X est plus petit que tout élément de Y, alors $m(X \cdot Y) \leq m(X) + m(Y)$. Pour assurer que l'on ne fait pas pire que trier X et Y séparément.
— Pour tout $e \in E$, $m(\langle e \rangle \cdot X) \leq |X| + m(X)$. De la même façon que pour la propriété précédente, pour trier $\langle e \rangle \cdot X$, il suffit de trier en premier X puis d'insérer e dans cette séquence triée.

Ces mesures de tri partiel vont par la suite nous permettre d'analyser la complexité des algorithmes de tri adaptatif. L'idée étant de s'en servir comme de paramètres qui seront utilisés dans la fonction de coût en nombre de comparaisons.

Un exemple de mesure de désordre sur le nombre de records

Il est possible de définir une mesure basée sur la statistique du nombre de records d'une séquence X par $m_{rec}(X) = |X| - rec(X)$. On rappelle qu'un record est un élément dans la séquence qui est plus grand que tous les éléments qui sont situés à sa gauche.

Proposition 4. *Soit X une séquence ordonnable, la fonction $m_{rec}(X) = |X| - rec(X)$ est une mesure de tri partiel.*

Démonstration. Pour prouver cette proposition, il suffit de vérifier tous les points qui font qu'une fonction est une mesure de désordre selon Mannila.

— soit X une séquence triée, alors $rec(X) = |X| \iff m_{rec}(X) = 0$.
— Soit $X = \langle x_1, \ldots, x_n \rangle$ et $Y = \langle y_1, \ldots, y_n \rangle$ telles que pour tout $i < j \in [n]$, $x_i < x_j \iff y_i < y_j$. Dans ce cas il est facile de montrer que x_i est un record dans X si et seulement si y_i est un record dans Y pour $i \in [n]$. Ainsi $rec(X) = rec(Y) \iff m_{rec}(X) = m_{rec}(Y)$.
— Soit $Y = \langle y_1, \ldots, y_n \rangle$, et soit $Y_i = \langle y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n \rangle$ pour $i \in [n]$. Il est facile de voir que les éléments précédant le i-ème dans Y restent des records ou des non-records dans Y_i. Pour les élément restant il y trois possibilités, soit $j \in [i+1, n]$
  — Soit y_j est un record dans Y, dans ce cas il reste un record dans Y_i.
  — Soit y_i est l'unique élément dans Y qui précède et est plus grand que y_j, alors y_j est un record dans Y_i.
  — Soit y_i n'est pas l'unique élément dans Y qui précède et est plus grand que y_j, alors y_j n'est pas un record dans Y_i.

  En combinant ces 3 cas et la possibilité que y_i peut être un record dans Y, l'inégalité suivante est obtenue $rec(Y_i) \geq rec(Y) - 1$, qui est équivalente à $m_{rec}(Y) \geq m_{rec}(Y_i)$. Il est ainsi possible d'obtenir n'importe quelle sous-séquence X de Y en procédant en plusieurs extractions d'un élément à chaque itération. Par transitivité des relations d'inégalités on obtiendra le résultat attendu que $m_{rec}(Y) \geq m_{rec}(X)$.
— Soit X et Y deux séquences ordonnables telles que $X < Y$, alors les éléments de X et de Y restent des records ou des non-records dans la concaténation $X \cdot Y$, et ainsi $m_{rec}(X \cdot Y) = m_{rec}(X) + m_{rec}(Y)$.
— Par définition de m_{rec}, les inégalités $m_{rec}(\langle e \rangle \cdot X) \leq |X|$ et $m_{rec}(X) \geq 0$ sont vérifiées. Ainsi en combinant ces deux inégalités le résultat $m_{rec}(\langle e \rangle \cdot X) \leq |X| + m_{rec}(X)$ est obtenu.

□

4.2.3 Critère d'optimalité

Mannila a défini un critère d'optimalité pour un algorithme de tri adaptatif en fonction d'une mesure de désordre arbitraire. Ce critère d'optimalité se base sur une notion de borne inférieure similaire à celle qui a été vue précédemment pour les algorithmes de tri par comparaisons.

Définition 14. Soit une mesure de désordre arbitraire m et une séquence X, l'ensemble $below(X, m)$ est défini par

$$below(X, m) = \{Y \in \mathfrak{S}_{|X|} \mid m(Y) \leq m(X)\}.$$

Cet ensemble nous permet d'obtenir un raisonnement similaire à celui de la borne inférieure utilisant la notion d'arbres de décision. L'idée est de restreindre les arbres de décisions d'un tri aux séquences dont la mesure m est plus petite que la mesure $m(X)$. Ainsi nous avons dans la preuve que le nombre de feuilles $x \geq |below(X, m)|$ qui remplace ici le $n!$ de la preuve originale.

Nous allons maintenant utiliser un paramètre $z \in \mathbb{N}$, et poser l'ensemble $below'(z, n, m) = \{Y \in \mathfrak{S}_n \mid m(Y) \leq z\}$

Théorème 7 (Mannila)**.** *Soit une mesure de désordre arbitraire m et des entiers n et z, tout algorithme de tri par comparaisons admet une borne inférieure en $\Omega(\max(n, \log |below'(z, n, m)|)$ pour trier toute séquence X de taille n ayant pour mesure $m(X) \leq z$.*

Démonstration. Nous nous intéressons ici à l'arbre de décision auquel nous avons retiré toutes les feuilles dont la mesure est supérieure à z. Ces feuilles, ne peuvent pas être la séquence à trier qui nous intéresse ici, d'où l'intérêt de les retirer. Le nombre de feuilles de cet arbre est alors d'au moins $|below'(z, n, m)|$. Le reste du raisonnement de la démonstration du Théorème 6 est alors similaire après avoir fait un remplacement de la borne en $n!$ du nombre de feuilles par $|below'(z, n, m)|$. L'ensemble $below'(z, n, m)$ peut cependant être petit ce qui conduirait à des résultats sous-linéaires.

Or, trier une séquence revient à découvrir l'ordre entre tous ces éléments. De ce fait, le problème du tri d'une séquence est au moins aussi difficile que le problème qui consiste à trouver le minimum dans cette séquence, puisque nous cherchons dans ce dernier problème à découvrir l'ordre entre un élément et tous les autres. Nous avons montré au cours du Chapitre 3, à l'aide d'un argument d'adversaire, que ce problème admettait une borne inférieure de $n - 1$ comparaisons indépendamment de l'ordre initial dans la séquence d'entrée. D'après cet argument de réduction, il faut donc au moins $(n - 1)$ comparaisons pour trier la séquence X. Nous avons ainsi deux bornes inférieures, et sans perte de généralité, le maximum de ces deux bornes est aussi une borne inférieure. C'est pour cette raison que nous utilisons un max dans l'expression de cette borne inférieure. □

Définition 15. Un algorithme de tri adaptatif est optimal pour la mesure m si et seulement si il peut trier une séquence $X \in E^*$ en au plus $\mathcal{O}(\max(|X|, \log |below(X, m)|)$ comparaisons.

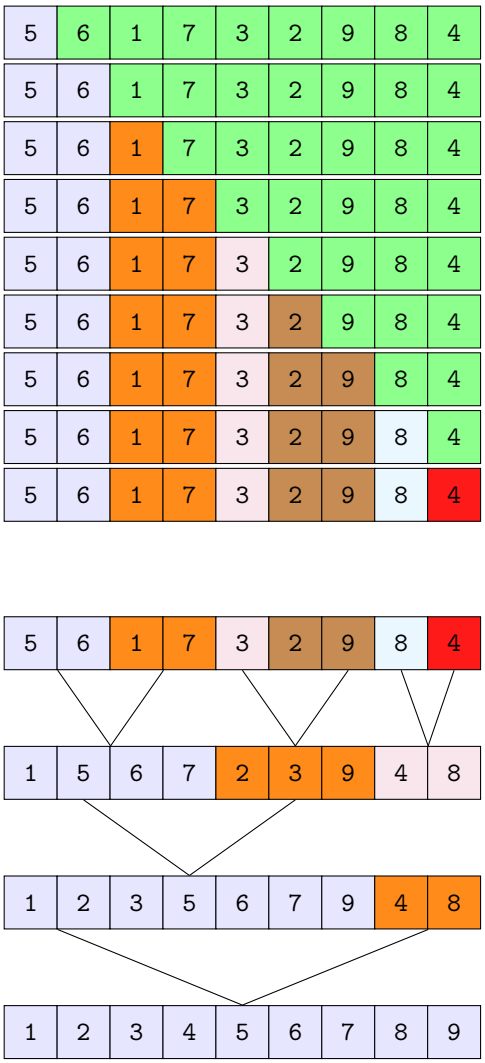


FIGURE 4.4 – Cette figure représente une trace d'exécution de l'algorithme du tri fusion naturel pour la séquence d'entrée $\langle 5, 6, 1, 7, 3, 2, 9, 8, 4\rangle$. La première partie procède en parcourant les éléments de la séquence de la gauche vers la droite. Dans cette partie chaque run ascendant est identifié et colorié. La deuxième partie procède en faisant 3 étapes de fusions, les runs fusionnés dessine alors un arbre de fusion dont on peut voir l'esquisse.

4.2.4 Le tri fusion naturel

Un exemple d'algorithme de tri qui est optimal pour la mesure du nombre de runs ascendants est le tri fusion naturel de Knuth (voir [30, 5.2.4]).

Proposition 5. *Tout algorithme de tri par comparaisons admet une borne inférieure en* $\Omega(n(1 + \log r))$ *pour trier une séquence de taille* n *et de* r *runs.*

L'algorithme du tri fusion naturel consiste à déterminer les runs ascendants de la séquence d'entrée à l'aide d'un parcours linéaire. Une étape de fusions consiste alors à parcourir chaque run deux par deux et à fusionner chaque couple de runs ainsi parcourus ensemble. On procède alors à plusieurs étapes de fusions jusqu'à n'obtenir plus qu'un seul et unique run qui sera bien évidemment la séquence d'entrée triée. Une trace d'exécution de l'algorithme pour la séquence d'entrée $\langle 5, 6, 1, 7, 3, 2, 9, 8, 4\rangle$ est donnée dans la Figure 4.4.

Proposition 6. *Pour trier une séquence de taille n et de r runs, le tri fusion naturel fait au plus $\mathcal{O}(n(1 + \log r))$ comparaisons.*

Démonstration. Le parcours linéaire pour déterminer les runs se fait en exactement $n - 1$ comparaisons quelle que soit la séquence à trier. Comme les runs sont fusionnés deux-à-deux jusqu'à ce qu'il n'en reste plus qu'un seul, il y a en tout $\lceil \log_2(r) \rceil$ étapes de fusions. Pour chaque fusion de deux runs de taille r_1 et r_2, il faut faire au plus $r_1 + r_2 - 1$ comparaisons. Durant une étape de fusion, tous les runs peuvent être fusionnés avec leur voisin dans le pire cas, ainsi il y a au plus $n - \frac{r}{2}$ comparaisons. En combinant le coût de chaque étape de fusions et du parcours linéaire, la borne supérieure est ainsi obtenue. □

4.3 TimSort

En 2002, Tim Peters cherchait à améliorer l'algorithme de tri SampleSort jusqu'alors utilisé par défaut dans le langage Python. Il mit au point un nouvel algorithme qu'il nomma TimSort en référence à son propre nom. C'est un algorithme de tri par comparaisons hybride du tri par insertion et d'un tri fusion adaptatif semblable au tri fusion naturel de Knuth [30, 5.2.4], mais s'inspirant également d'idées basées sur les travaux de McIlroy [35]. Lors de la conception de son algorithme, Tim Peters rédigea par la suite une note [37] dans laquelle il expliqua les différents tests qu'il a conduit pour vérifier l'efficacité de son algorithme. Au vu des résultats obtenus, l'algorithme fût par la suite implanté dans la nouvelle version 2.3 de Python et reste, au moment où ce manuscrit est rédigé, toujours utiliser dans les versions les plus récentes de Python 2 et Python 3. Joshua Bloch proposa également de faire un portage de TimSort pour la librairie standard Java du package `java.util.collections`. Ce portage fût adopté par les développeurs et fût intégré dans la version 7 de Java pour trier une collection d'objets, un tri rapide à double pivots est utilisé pour les types primitifs. Au moment où sont rédigées ces lignes, la version 9 de Java vient d'être publiée et TimSort reste utilisé pour trier les données de type non primitif. Il y a cependant peu de résultats qui sont réellement connus d'un point de vue théorique sur TimSort. En particulier sa complexité en comparaisons dans le pire cas, bien qu'étant annoncée en $\mathcal{O}(n \log n)$ par son créateur, ne semble à ce jour pas avoir été prouvée. [2] Pour le reste de ce chapitre, nous utiliserons la notation $R_1 \oplus R_2$, pour deux runs R_1 et R_2, pour représenter le run obtenu à la fusion de ces deux runs.

2. On pourrait croire que cette complexité découle de la structure de pile utilisée par TimSort qui comme nous le verrons par la suite a une hauteur en $\mathcal{O}(\log(n))$. Cependant, nous verrons par la suite un contre-exemple qui montre que cette condition est loin d'être suffisante pour garantir la complexité en nombre de comparaisons de TimSort. Une preuve de sa complexité sera donnée dans la partie 4.3.5.

4.3.1 TimSort dans les grandes lignes

TimSort possède un fonctionnement similaire à l'algorithme du tri fusion naturel dans son principe de découpage en runs et de fusions de ces runs. Cependant, comme vu précédemment, des idées provenant des travaux de McIlroy ont également été utilisées, l'une d'elles est notamment qu'un algorithme de tri devrait être capable de trier une séquence et son renversé avec seulement un coût supplémentaire linéaire. Pour ce faire TimSort n'identifie pas uniquement les runs ascendants, il identifie également les runs descendants. Contrairement au tri fusion naturel qui fait les fusions une fois que tous les runs sont connus dans la séquence à trier, TimSort procède différemment en faisant les fusions à la volée à l'aide de règles de décision. Nous allons maintenant voir dans le détail, comment sont identifiés et fusionnés les runs.

Création de la pile des runs

Une pile de runs est utilisée pour stocker chaque nouveau run détecté lors du parcours de la séquence. L'identification d'un nouveau run se fait à l'aide de la procédure suivante :

1. Les deux premiers éléments sont comparés afin de détecter si le run est ascendant ou descendant.
2. Chaque élément est comparé à celui qui le précède afin de vérifier si il est dans la continuité du run.
3. Une fois le dernier élément du run trouvé, si le run est descendant, il est renversé.
4. La taille du run est ensuite comparée à une valeur seuil arbitraire. Si elle est plus petite que ce seuil, alors le run est agrandi jusqu'à cette taille seuil en utilisant une variante du tri par insertion, le tri binaire. C'est donc ici que l'hybridation avec le tri par insertion se fait dans l'algorithme. Le tri binaire diffère du tri par insertion classique en effectuant une recherche dichotomique pour trouver la position du prochain élément à insérer.
5. Le run est finalement ajouté à la pile des runs.

Remarque. Le point 4 est utilisé par TimSort pour améliorer ses performances en pratique, on pourrait l'enlever sans changer la complexité de l'algorithme. Nous avons choisi de le laisser pour coller au plus près des implantations de l'algorithme.

Un exemple d'exécution de la création de la pile est donné dans la Figure 4.5.

Règle de fusions

Lorsqu'un run est ajouté à la pile lors de l'étape précédente, une autre procédure va être exécutée pour vérifier si il est nécessaire de faire des fusions. La stratégie de fusion employée par TimSort consiste à appliquer la première règle dont la condition est vérifiée parmi les trois suivantes :

Appelons W, X, Y et Z les runs de tailles respectives w, x, y et z, qui se trouvent en haut de la pile, dans cet ordre, Z étant au sommet de la pile.

1. Si $x \leq z$ alors fusionner X et Y.
2. Si $x \leq y + z$ ou $w \leq x + y$ alors fusionner Y et Z.

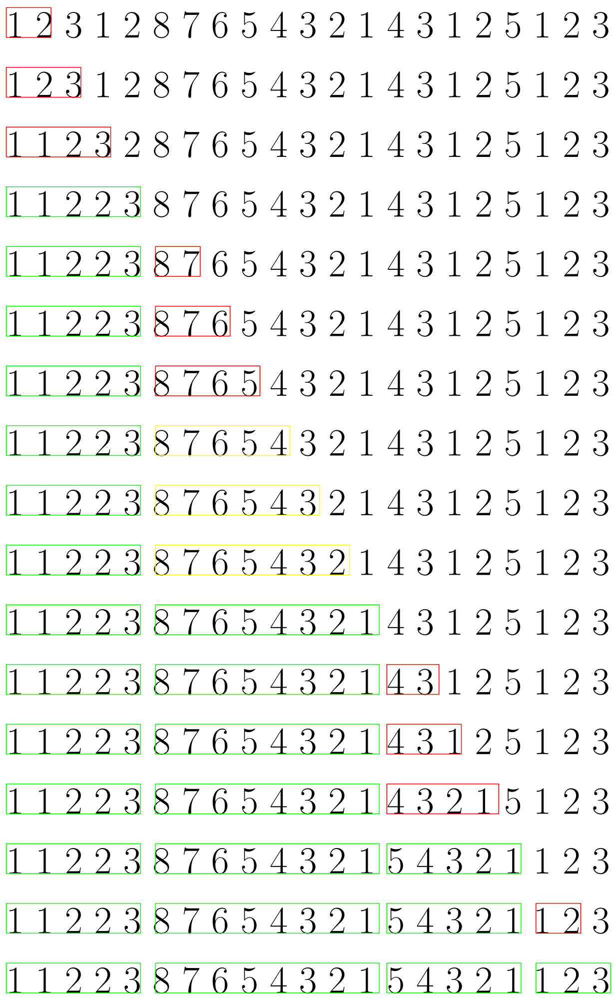


FIGURE 4.5 – Cette figure représente la détection des runs de la séquence d'entrée ⟨123128765432143125123⟩. Pour cet exemple nous avons fixé la taille minimale d'un run à 5. Lorsqu'un run est encadré en vert, cela signifie qu'il est complet et est alors envoyé à la pile des runs. Lorsqu'un run est encadré en rouge, cela signifie que le run n'est pas complètement identifié et qu'il est trop petit. Dans le cas où un run est encadré en rouge et que celui-ci est déjà terminé, nous voyons alors que TimSort va le prolonger en insérant les éléments suivants dans le run jusqu'à ce que la taille minimale du run soit atteinte. Un run encadré en jaune est un run dont la taille est supérieure à la taille minimale mais qui n'a pas encore été complètement identifié.

3. Si $y \leq z$ alors fusionner Y et Z.

Si il y a moins de quatre runs dans la pile de fusion, nous ignorons les règles dont la condition ne peut pas être évaluée. Une fusion entraînant la modification du haut de la pile, il est alors nécessaire de recommencer à vérifier si une des conditions est encore satisfaite (en repartant de la règle 1). Il faut alors prendre soin de renommer les runs W, X, Y et Z par les nouveaux runs qui forment le haut de la pile comme nous pouvons le voir dans l'exemple de la Figure 4.6. Dès qu'aucune des trois conditions n'est vérifiée, le processus de fusions se termine ; s'il reste des éléments à traiter, nous retournons alors à l'étape précédente pour créer un nouveau run. Lorsqu'il n'y a plus d'éléments à traiter, les runs restants dans la pile sont fusionnés deux à deux en partant du haut de la pile, jusqu'à ce qu'il ne reste plus qu'un seul run correspondant à la séquence triée.

Remarque. La version d'origine de TimSort ne comportait pas le test $w \leq x+y$. Il s'agit d'une correction qui a été apportée à la suite de la découverte d'un bug, dont nous parlerons plus en détail dans la sous-section 4.3.4. Cette version de TimSort est celle actuellement utilisée par Python.

La procédure de fusion

La procédure utilisée pour faire les fusions contient elle-même quelques optimisations qui essayent d'exploiter la structure de la séquence d'entrée.

La première optimisation concerne la mémoire auxiliaire utilisée pour copier des éléments des runs qui vont être fusionnés. TimSort copie le run de plus petite taille dans un tableau auxiliaire. Si le run de gauche est copié, les éléments sont placés dans l'ordre croissant depuis la position de départ du run de gauche, en allant de la gauche vers la droite dans la séquence d'entrée. Réciproquement, si le run de droite est copié, les éléments sont placés dans l'ordre décroissant depuis la position de fin du run de droite et en allant de la droite vers la gauche dans la séquence d'entrée.

Le galop : Lors de la fusion les premiers éléments des deux runs à fusionner sont comparés, comme dans le cas d'un tri fusion classique. Cependant afin d'exploiter une potentielle structure dans la séquence, un compteur est utilisé. Celui-ci permet de compter le nombre consécutif de fois qu'un élément est pris dans un même run. Lors de cette étape, le run dont l'élément vient d'être ajouté au résultat de la fusion est alors dit *gagnant*. Le compteur est donc incrémenté à chaque fois que le même run gagne, et est réinitialisé lorsque celui-ci perd. Lorsque le compteur atteint une valeur seuil arbitraire, la fusion passe en *mode galop*.

Le galop consiste à trouver de manière efficace le premier élément du run gagnant qui est plus grand que le prochain élément du run perdant. Une fois la position de cet élément trouvée, tous les éléments le précédant sont insérés dans la zone de fusion suivis par le premier élément du run perdant. Le run perdant est ensuite considéré comme le run gagnant et le galop se poursuit jusqu'à ce qu'il soit arrêté ou qu'il ne reste plus aucun élément à traiter. Il est possible que le galop soit interrompu si après deux étapes de galop, le nombre d'éléments insérés est plus petit que la valeur seuil utilisée pour déclencher le galop.

Pour faire la recherche en mode galop, on commence par chercher un intervalle de la forme $[2^k, 2^{k+1}[$ auquel la position appartient. Une fois l'intervalle

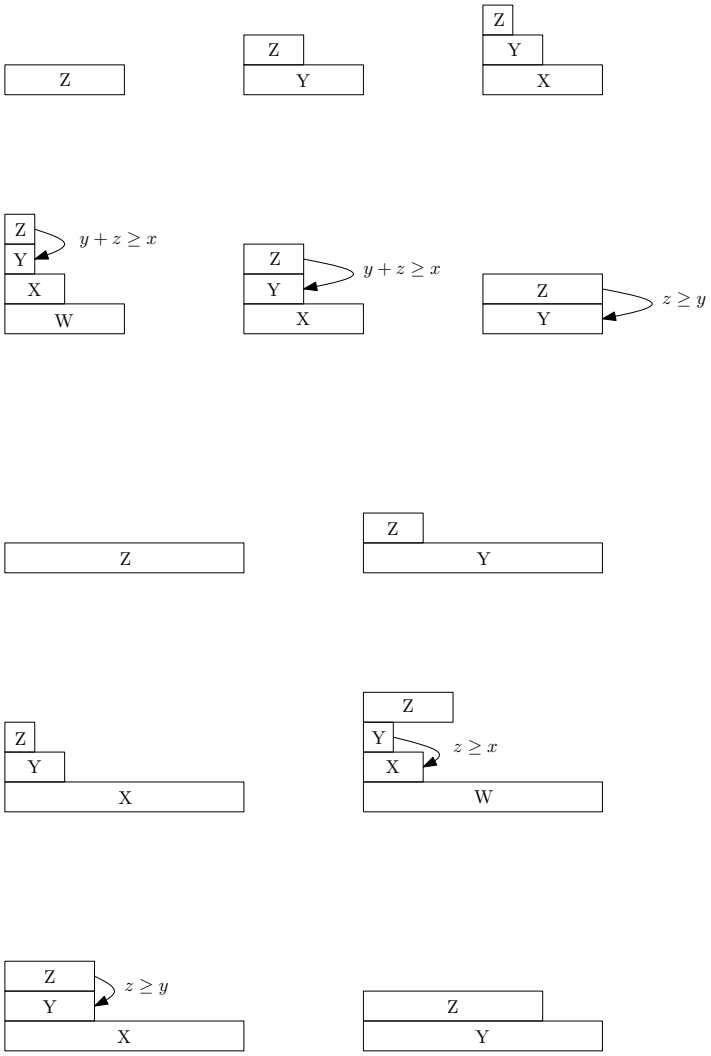


FIGURE 4.6 – Ce schéma représente un exemple des différentes fusions qui peuvent avoir lieu lorsqu'une des règles de fusion n'est pas respectée. Ici chaque rectangle représente un run et ceux-ci sont insérés dans une pile. C'est en regardant le haut de la pile que l'on vérifie si chacune des règles sont vérifiées. La taille des runs correspond à la largeur des rectangles qui les représentent.

trouvé, il suffit de faire une recherche dichotomique sur cet intervalle pour déterminer la position exacte de l'élément à rechercher.

La valeur seuil est premièrement fixée à une valeur arbitraire, mais elle peut être incrémentée ou décrémentée dynamiquement en fonction du succès du galop, à savoir si il a dû être interrompu ou non. Cela permet d'avoir un deuxième niveau d'adaptation dans le cas où la séquence d'entrée présente une très forte structure.

Remarque. Le galop est un procédé efficace en pratique qui permet expérimentalement de gagner quelques comparaisons. Cependant, il n'améliore pas la complexité dans le pire cas de l'algorithme.

4.3.2 Généralisation d'algorithmes de tri "à la TimSort"

Le procédé de fusion de TimSort peut facilement être généralisé à d'autres règles de fusions. Dans cette partie, nous donnons un cadre formel pour ces différentes stratégies et proposons une version épurée de TimSort. Cet algorithme simple est un bon support pour comprendre les mécanismes de TimSort et analyser leur complexité.

Stratégie de fusions

Soit la pile de runs, de hauteur h, $S = \langle S_1, \dots, S_h \rangle$, où S_1 est le bas de la pile. Nous écrirons également la séquence $\langle s_1, \dots, s_h \rangle$ des tailles des runs de S. Une *stratégie de fusions* χ consiste en une séquence de couples de la forme (ρ, μ) où ρ est une contrainte qui doit être vérifiée sur la pile, et μ est une suite d'opérations de fusions à effectuer dans le cas où la contrainte n'est pas respectée. Nous appellerons un couple de cette forme une *règle* de la stratégie de fusion. Lorsque l'action μ doit être exécutée, on dit que la règle (ρ, μ) est *activée*. La procédure de fusions d'un tri "à la TimSort" va pour chaque itération activer la première règle dont la contrainte n'est pas respectée dans χ. L'ordre des règles de la stratégie χ est donc un système à priorités. L'algorithme 4 permet de donner un patron des algorithmes de tri "à la TimSort". Par la suite, nous appellerons les suites d'opérations effectuées de la ligne 1 à la ligne 4 la *création de la pile*, de la ligne 5 à la ligne 7 une *étape de fusions*, et la ligne 8 à la ligne 11 la *fusion finale*.

Exemples de stratégies possibles

α-StackSort : Les algorithmes de la famille α-StackSort ont une stratégie simple qui est constituée d'une unique règle dépendant d'un paramètre réel $\alpha > 1$. La stratégie des algorithmes α-StackSort est défini comme suit :

1. $\rho := s_{h-1} > \alpha s_h$, $\mu :=$ fusionner S_{h-1} et S_h.

Exemple. *Nous allons regarder le début de la trace d'exécution du 2-StackSort pour un cas concret. Considérons que les tailles des runs entrant dans la pile sont données par la séquence* $s = \langle 15, 7, 4, 5, 3, \dots \rangle$. *Au moment où les deux premiers runs sont ajoutés à la pile, il ne se passe rien :* $15 > 2 \times 7$. *Lorsque le troisième run entre dans la pile, une première fusion a lieu, car* $7 < 2 \times 4$. *La pile contient alors deux runs de tailles* 15 *et* 11, *allant du bas vers le haut. Comme* $15 < 2 \times 11$, *une deuxième fusion se produit laissant un unique run de*

Données : X une séquence d'objets comparables
Résultat : X triée
Initialiser une nouvelle pile de run S vide
tant que *Il reste des éléments de X qui ne sont dans aucun run de S* **faire**
    Soit R le prochain run obtenu à partir des éléments restants de X
    $S \leftarrow S \cdot R$
    tant que *Une règle de χ est activée* **faire**
        Soit (ρ, μ), la première règle de χ qui est activée
        $S \leftarrow \mu(S)$
$\ell \leftarrow |S|$
tant que $\ell > 1$ **faire**
    $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
    $\ell \leftarrow \ell - 1$

Algorithme 4 : Le patron d'un algorithme à la TimSort pour une stratégie de fusion χ. Nous considérons ici que l'action μ d'une règle (ρ, μ) de χ est une fonction qui prend en entrée la pile des runs et donne en sortie une nouvelle pile.

taille 26 *dans la pile. Lorsque le quatrième run entre dans la pile, il ne se passe rien puisque* $26 > 2 \times 5$. *Lorsque le cinquième run entre dans la pile, comme* $5 < 2 \times 3$, *il y a une fusion de ces deux runs laissant alors dans la pile deux runs de tailles* 26 *et* 8, *allant du bas vers le haut. Comme* $26 < 2 \times 8$ *plus aucune fusion ne se produit après l'ajout du cinquième run.*

TimSort Évidemment TimSort est également un algorithme "à la TimSort", et il est défini par la stratégie suivante :

1. $\rho_1 := s_{h-2} > s_h$, $\mu_1 :=$ fusionner S_{h-2} et S_{h-1}.
2. $\rho_2 := s_{h-2} > s_{h-1} + s_h$, $\mu_2 :=$ fusionner S_{h-1} et S_h.
3. $\rho_3 := s_{h-3} > s_{h-2} + s_{h-1}$, $\mu_3 :=$ fusionner S_{h-1} et S_h.
4. $\rho_4 := s_{h-1} > s_h$, $\mu_4 :=$ fusionner S_{h-1} et S_h.

4.3.3 L'invariant de la pile

En fonction de la stratégie employée pour la fusion des runs, des propriétés invariantes peuvent être observées. C'est notamment le cas des deux précédents exemples qui admettent chacun un invariant décrit dans la proposition 7 pour les algorithmes de la famille α-StackSort et la proposition 8 pour TimSort.

Proposition 7. *Soit la pile de run $S = \langle S_1, \ldots, S_h \rangle$ de tailles $\langle s_1, \ldots, s_h \rangle$ obtenue après une étape de fusions de l'Algorithme 4 pour la stratégie de fusion de la famille α-StackSort. Nous avons, pour tout $i \in [h-1]$, $s_i > \alpha s_{i+1}$.*

Démonstration. Initialement la pile est vide et la propriété invariante est forcément vérifiée. Nous allons donc poser à partir de maintenant l'hypothèse qu'avant l'arrivée du nouveau run S_{h+1} et de taille s_{h+1}, la pile $S = \langle S_1, \ldots, S_h \rangle$ de tailles $\langle s_1, \ldots, s_h \rangle$ vérifie la propriété invariante. Nous devons remarquer

qu'en entrant dans la pile, comme il n'y a qu'une seule règle de fusion, le run peut créer une chaîne de fusions au sommet de la pile tant que la règle est activée. Soit $k < h$ le premier indice tel que pour la nouvelle pile $S' = \langle S_1, \ldots, S_k \oplus S_{k+1} \cdots \oplus S_{h+1} \rangle$, nous avons $s_{k-1} > \alpha(s_k + s_{k+1} + \cdots + s_{h+1})$. Alors la pile S' d'après notre remarque précédente est la nouvelle pile obtenue à la fin des fusions. De plus, d'après l'invariant la propriété est respectée jusqu'à l'avant dernier run mais également jusqu'au dernier run de par la dernière inégalité. La règle est ainsi respectée et les fusions intermédiaires s'arrêtent. Nous venons de plus de voir que lors de l'ajout d'un nouveau run dans la pile, l'invariant était toujours vrai une fois les fusions intermédiaires effectuées, prouvant ainsi la véracité de la proposition. □

Proposition 8. *Soit la pile de run $S = \langle S_1, \ldots, S_h \rangle$ de tailles $\langle s_1, \ldots, s_h \rangle$ après une étape de fusions de l'Algorithme 4 pour la stratégie de fusion de TimSort. Nous avons, pour tout $i \in [h-2]$, $s_i > s_{i+1} + s_{i+2}$ et $s_{h-1} > s_h$.*

Démonstration. La preuve formelle de cet invariant a été effectuée à l'aide de l'outil KeY qui permet de faire de la vérification formelle à partir d'un code source Java par l'équipe de de Gouw, Rot, de Boer, Bubel et Hähnle [21] et contient un nombre considérable d'étapes. Il est cependant assez simple de démontrer que l'invariant est bel et bien conservé à l'aide de cette stratégie de fusion. Soit la pile $S = \langle S_1, \ldots S_h \rangle$ de tailles $\langle s_1, \ldots, s_h \rangle$ avant les fusions engendrées en cas d'activation des règles de la stratégie de fusion. À la fin de ces fusions, la pile sera dans un nouvel état $S' = \langle S'_1, \ldots, S'_{h'} \rangle$ de tailles $\langle s'_1, \ldots, s'_{h'} \rangle$, avec h' la hauteur de S'. De plus, comme la stratégie de fusion de TimSort ne procède qu'à des fusions impliquant toujours l'avant dernier run de la pile, nous avons $S'_i = S_i$ et $s'_i = s_i$ pour $i \in [h'-2]$. Les deux derniers runs peuvent, quant à eux, être des fusions des runs qui se trouvaient en haut de S. Par l'invariant, de l'étape précédent, nous avons déjà $s'_i > s'_{i+1} + s'_{i+2}$ pour $i \in [h'-4]$. Comme nous sommes à la fin de l'étape de fusions, nous avons, de par les contraintes des règles de la stratégie de fusions, les relations suivantes : $s'_{h'-3} > s'_{h'-2} + s'_{h'-1}$, $s'_{h'-2} > s'_{h'-1} + s'_{h'}$, et $s'_{h'-1} > s'_{h'}$. En combinant l'ensemble de ces inégalités, nous montrons que l'invariant est vérifié à la fin de l'étape de fusion. □

À l'aide de la Propositions 7, nous démontrons la proposition 9.

Proposition 9. *Pour l'Algorithme 4 avec la stratégie de la famille α-StackSort, la taille de la pile des runs après une étape de fusions admet un majorant de la forme $K \log n$ pour $n > 2$ la taille de la séquence d'entrée, et pour une certaine constante $K \in \mathbb{R}$.*

Démonstration. Pour démontrer cette proposition, il suffit d'exhiber la plus longue suite possible en partant de la fin et qui respecte les invariants des propositions 7 et 8. Soit $s = \langle s_1, \ldots, s_h \rangle$ les tailles des runs de la pile de hauteur h, telle que s_1 est la taille du run le plus au bas de la pile. Comme les tailles sont entières et positives, le but est donc que chaque run soit le plus petit possible. Ainsi donc pour les deux cas, la taille du dernier run devra être la valeur $s_h = MinRun \geq 1$. D'après l'invariant, pour $i \in [h-1]$, nous avons $s_i > \alpha s_{i+1}$. Prouvons par récurrence que pour tout $i \in [0, h-1]$, nous avons $s_{h-i} \geq \alpha^i$. Nous savons déjà que cette propriété est vraie pour $i = 0$ puisque $s_h \geq 1$. Supposons donc que cette propriété est vraie pour tout $i' < i$. $s_{h-(i+1)} = s_{h-i-1} > \alpha s_{h-i}$ d'après l'invariant. Par hypothèse de récurrence, nous avons alors $s_{h-(i+1)} \geq \alpha \times \alpha^i = \alpha^{i+1}$.

La propriété est donc vraie pour $i+1$ également, et par récurrence nous en concluons que la propriété est vraie pour tout i. Comme n est la somme de la taille de tous les runs de la pile et de ceux qui n'y ont pas encore été insérés, nous avons alors la relation

$$n \geq \sum_{i=1}^{h} s_i \geq \sum_{i=0}^{h-1} \alpha^i. \tag{4.1}$$

Nous reconnaissons ici une somme de termes d'une suite géométrique et obtenons donc

$$n \geq \frac{1-\alpha^h}{1-\alpha}. \tag{4.2}$$

Après un dernier réarrangement nous obtenons l'inéquation

$$\alpha^h \leq (\alpha-1)n+1 \tag{4.3}$$

En passant au logarithme, nous obtenons alors

$$h \leq \log_\alpha\left((\alpha-1)n+1\right) \tag{4.4}$$

Nous allons maintenant faire apparaître la constante K de cette dernière inéquation.

$$\begin{aligned}
h &\leq \log_\alpha\left(\left(\alpha-1+\frac{1}{n}\right)n\right) \\
h &\leq \log_\alpha\left((\alpha-1+1)n\right) \\
h &\leq \log_\alpha\left((\alpha n\right) \\
h &\leq \log_\alpha(\alpha)+\log_\alpha(n) \\
h &\leq 1+\log_\alpha n \\
h &\leq \left(\frac{1}{\log_\alpha n}+1\right)\log_\alpha n \\
h &\leq \left(\frac{\log\alpha}{\log n}+1\right)\log_\alpha n \\
h &\leq (\log\alpha+1)\log_\alpha n && (\text{car } n \geq 3 > e) \\
h &\leq \left(1+\frac{1}{\log\alpha}\right)\log n
\end{aligned}$$

En posant ainsi, $K = \left(1+\frac{1}{\log\alpha}\right)$, nous obtenons le résultat annoncé. □

À l'aide de la Proposition 8, nous pouvons faire de même pour TimSort et démontrer la Proposition 10.

Proposition 10. *Pour l'Algorithme 4 avec la stratégie de TimSort, la taille de la pile des runs après une étape de fusions admet un majorant de la forme $K\log n$ pour $n > 2$ la taille de la séquence d'entrée, et pour une certaine constante $K \in \mathbb{R}$.*

Démonstration. Nous savons d'après l'invariant que pour tout $i \in [h-2]$ nous avons $s_i > s_{i+1}+s_{i+2}$. Cette relation peut faire penser à la formule de récurrence de la suite de Fibonacci, et nous allons voir dès à présent qu'il est possible d'exprimer la taille des runs de la pile à l'aide du nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$ comme pour cette première. Rappelons de plus que par définition $\phi^2 = \phi + 1$. Nous avons de plus la relation triviale $s_h > 1$ et comme $s_{h-1} > s_h$ par transitivité $s_{h-1} > 1$. Nous allons démontrer la proposition $s_{h-i} \geq \phi^{i-2}$ pour tout $i \in [1, h-1]$. Nous venons déjà de vérifier que la proposition était vraie pour $i = 0$ et $i = 1$. Supposons donc dès à présent que la proposition est vraie pour tout $i' \in [0, i]$. Nous avons vu par l'invariant que $s_{h-(i+1)} > s_{h-(i+1)+1} + s_{h-(i+1)+2}$. Par hypothèse de récurrence, nous avons donc $s_{h-(i+1)} > \phi^{i-2}+\phi^{i-3} = \phi^{i-3}(\phi+1) = \phi^{i-3}\phi^2$. Nous avons alors que $s_{h-(i+1)} \geq \phi^{(i+1)-2}$ et donc notre proposition est vérifiée pour $i+1$, ce qui conclut la preuve par récurrence. Nous avons, comme pour le cas du α-StackSort, la relation suivante :

$$n \geq \sum_{i=1}^{h} s_i \geq \sum_{i=0}^{h-1} \phi^{i-2} \tag{4.5}$$

Nous reconnaissons une fois de plus une somme de termes d'une suite géométrique de raison ϕ, et nous obtenons alors l'inégalité

$$n \geq \phi^{-2} \frac{1-\phi^h}{1-\phi}. \tag{4.6}$$

Un dernier réarrangement nous donne alors

$$\phi^h \leq \frac{\phi^2 n}{\phi - 1} + 1. \tag{4.7}$$

En passant par le logarithme nous obtenons ainsi

$$h \leq \log_\phi \left(\frac{\phi^2 n}{\phi - 1} + 1 \right) \tag{4.8}$$

Nous allons à partir de cette dernière équation obtenir la constante K.

$$
\begin{aligned}
h &\leq \log_\phi \left(\left(\frac{\phi^2}{\phi - 1} + \frac{1}{n} \right) n \right) \\
h &\leq \log_\phi \left(\left(\frac{\phi^2}{\phi - 1} + 1 \right) n \right) \\
h &\leq \log_\phi \left(\frac{2\phi}{\phi - 1} n \right) \\
h &\leq \log_\phi \left(\frac{2\phi}{\phi - 1} \right) + \log_\phi n \\
h &\leq \left(\frac{\log_\phi \left(\frac{2\phi}{\phi - 1} \right)}{\log_\phi n} + 1 \right) \log_\phi n \\
h &\leq \left(\frac{\log \left(\frac{2\phi}{\phi - 1} \right)}{\log n} + 1 \right) \log_\phi n \\
h &\leq \left(\log \left(\frac{2\phi}{\phi - 1} \right) + 1 \right) \log_\phi n \\
h &\leq \left(\log_\phi \left(\frac{2\phi}{\phi - 1} \right) + \frac{1}{\log \phi} \right) \log n
\end{aligned}
$$

En posant $K = \left(\log_\phi \left(\frac{2\phi}{\phi - 1} \right) + \frac{1}{\log \phi} \right)$ nous obtenons alors le résultat annoncé.

□

Remarque. Dans le code source du portage de TimSort pour Java, dont nous donnons un extrait dans le Listing 1, la taille de la pile est fixée en comparant la taille n de la séquence à trier avec des valeurs "magiques".

```
int stackLen = (len <    120  ?  5 :
                len <   1542  ? 10 :
                len < 119151  ? 19 : 40);
runBase = new int[stackLen];
runLen = new int[stackLen];
```

Listing 1 – Un extrait de l'implantation Java de TimSort : l'allocation de la pile des runs.

Ces valeurs sont faciles à retrouver et proviennent de la relation de récurrence que l'on trouve dans l'invariant. En effet, pour déterminer la taille du i-ème run de la pile il suffit de poser $s_i = s_{i+1} + s_{i+2} + 1$ qui est le plus petit run possible qui respecte l'invariant. L'initialisation est faite avec $s_h = MinRun$ et $s_{h+1} = MinRun + 1$. Le tableau qui suit permet de mettre en lumière ces valeurs, nous avons dans ce cas $MinRun = 16$.

h	1	2	3	4	5	6	7	8	9	10	11
s_h	16	17	34	52	87	140	228	369	598	968	1567
$\sum_{i=1}^{h} s_i$	16	33	67	119	206	346	574	943	1541	2509	4076

h	12	13	14	15	16	17	18	19
s_h	2536	4104	6641	10746	17388	28135	45524	73660
$\sum_{i=1}^{h} s_i$	6612	10716	17357	28103	45491	73626	119150	192810

Nous retrouvons alors les valeurs de n correspondant à la troisième ligne à laquelle on ajoute 1 dans le code de Java du Listing 1 car les inégalités sont strictes. Le premier test se fait pour $n < 120$ ce qui correspond à la colonne $h = 4$, le deuxième test se fait pour $n < 1542$ ce qui correspond à la colonne $h = 9$, enfin le dernier test se fait pour $n < 119151$ ce qui correspond à la colonne $h = 18$.

4.3.4 TimSort et ses bugs

Nous allons dans cette sous-section nous intéresser à des bugs qui ont été ou qui sont encore présents dans TimSort. Nous avons évoqué un premier bug lors de la présentation des règles de fusions de ce dernier.

Une correction a été apportée à la stratégie de fusion de TimSort par les auteurs de Gouw, Rot, de Boer, Bubel et Hähnle [21], qui ont obtenu la preuve formelle de l'invariant de la Proposition 8 à l'aide de leur outil KeY. Auparavant, la règle concernant le run W était omise et cela pouvait provoquer des cas pour lesquels cet invariant était faux.

Voyons cette faille avec un exemple concret pour lequel une erreur sur l'invariant peut être obtenue si nous omettons la règle sur le run W : considérons que les tailles des runs entrants sont données par la suite $\langle 176, 128, 40, 32, 48\rangle$. Pour ce cas, une seule fusion va se produire, quand le dernier run va entrer dans la pile. En effet, $48 > 40$ ce qui viole la première règle, entrainant une fusion des runs de taille 32 et 40. À la suite de cette fusion, les tailles des runs de la pile sont alors $\langle 176, 128, 72, 48\rangle$. Si la règle impliquant W est omise, aucune fusion ne va se produire : $128 > 72 + 48$ et $72 > 48$. Cependant l'invariant n'est plus vrai puisque $176 < 128 + 72$. Bien entendu, cette dernière inéquation serait captée par la règle impliquant W qui permettrait de rétablir l'invariant dans la version de TimSort corrigée.

En exploitant cette faille sur l'invariant, ils ont ensuite réussi à trouver un exemple provoquant une exception dans le cas où TimSort était exécuté sur une machine virtuelle Java. Ce problème ne concernait que l'implantation Java de TimSort qui allouait la taille de sa pile de runs statiquement, comme nous pouvons le voir dans le Listing 1, qui est un extrait de l'implantation de TimSort pour Java. La version Python de TimSort procède à une augmentation dynamique de la taille de la pile dans le cas où celle-ci s'avère trop petite.

Dans leur article [21], l'équipe propose de modifier TimSort de deux façons différentes. La première modification est de rajouter le test de $w \leq x + y$ que nous avons présenté précédemment. Cette modification a été adoptée par les développeurs de Python. La deuxième modification ne s'applique que pour Java et consiste à augmenter la taille de la pile des runs dans le Listing 1. C'est cette dernière modification que les développeurs de Java ont choisie. Les auteurs de cet article ont également fait une erreur et cette nouvelle version contient également un bug pour lequel ces modifications de tailles ne sont pas suffisantes. Dans leur article, il y a un lemme qui affirme qu'il est impossible d'avoir l'erreur sur l'invariant apparaitre deux fois de manière consécutive. Autrement dit à tout instant dans la pile, il est impossible d'avoir quatre runs consécutifs de tailles

respectives $\alpha, \beta, \gamma, \omega$ tels que $\alpha \leq \beta + \gamma$ et $\beta < \gamma + \omega$. Cette proposition est en réalité fausse, il est possible de créer des séquences qui permettent de propager l'erreur au moins plus d'une fois. Il est alors une fois de plus possible de faire planter la machine virtuelle de Java en exploitant ce résultat. Le lecteur intéressé par ce dernier résultat peut regarder notre article [8].

4.3.5 Complexité de TimSort

Dans cette section, nous donnons une preuve de la complexité, dans le pire cas, en nombre de comparaisons qui a été annoncée en $\mathcal{O}(n \log n)$. Nous procédons à un découpage de l'analyse en plusieurs parties où des comparaisons sont effectuées dans l'algorithme. Premièrement, nous regardons la création de la pile où les éléments consécutifs de la séquence sont comparés pour obtenir de nouveaux runs. C'est aussi pendant cette étape qu'un algorithme de tri par insertion est appelé afin d'agrandir les runs trop petits. Le coût en comparaisons de ces appels est inclus dans cette partie de l'analyse. Deuxièmement, nous étudions la fusion finale. Nous rappelons que cette étape a lieu après insertion du dernier run dans la pile, et s'il reste plus de deux runs à la fin de la dernière *étape de fusions*. Enfin, nous analysons le coût en comparaisons de l'ensemble des *étapes de fusions* qui se produisent à chaque fois qu'un nouveau run est inséré dans la pile. Cette dernière partie est la plus technique à démontrer, car nous utilisons la *méthode comptable* d'analyse amortie pour déterminer une borne supérieure du coût. Une explication détaillée de l'analyse amortie est disponible dans le chapitre 3

La création de la pile des runs

Proposition 11. *Pour la séquence d'entrée X de taille n, l'Algorithme 4, à la création de la pile, de la ligne 1 à la ligne 4, a besoin de faire au plus $\mathcal{O}(n)$ comparaisons.*

Démonstration. Lors de la création d'un nouveau run, deux étapes se produisent : le run est identifié en comparant des éléments consécutifs dans la séquence, si le run est trop petit, il est prolongé. Nous allons appeler respectivement ces deux étapes, *l'identification* et *la prolongation*.

Nous nous intéressons maintenant à l'étape d'identification. L'algorithme continue d'effectuer des comparaisons entre éléments consécutifs jusqu'à trouver un élément qui perturbe la monotonie du run en cours d'identification ou jusqu'à atteindre la fin de la séquence. Soit r la taille du run à la fin de l'identification. Pour les deux cas d'arrêt de l'étape, l'algorithme doit parcourir deux-à-deux tous les éléments consécutifs du run, ce qui requiert un coût de $r - 1$ comparaisons. Pour le cas où un élément perturbe la monotomie du run, il est nécessaire d'effectuer une dernière comparaison entre cet élément perturbateur et le dernier élément du run. Ainsi, pour tout run de taille r qui n'est pas le dernier de la séquence, l'étape d'identification s'effectue en faisant r comparaisons. Pour le dernier run de la séquence, il n'est nécessaire de faire que $r - 1$ comparaisons. Posons $\langle t_1, \ldots, t_p \rangle$ les tailles des runs qui ont été obtenues lors de cette étape. Nous avons la relation

$$\sum_{i=1}^{p} t_i \leq n \tag{4.9}$$

qui est vérifiée. En combinant toutes ces remarques, nous obtenons que le coût des comparaisons entre éléments consécutifs est donné par la relation

$$\sum_{i=1}^{p-1} t_i + (t_p - 1). \tag{4.10}$$

Ces deux équations une fois combinées montre que le coût de ces comparaisons est au plus $n - 1$.

Regardons maintenant l'étape de prolongation. Soit q le nombre de runs prolongés par le tri par insertion. Nous définissons l'ensemble $\{\tau_1, \ldots, \tau_q\}$ des sous-séquences constituées d'un run obtenu par l'étape d'identification et des éléments qui suivent dans la séquence d'entrée X de façon à avoir en tout $MinRun$ éléments par sous-séquence. Autrement dit, lorsque les sous-séquences de cet ensemble sont triées, nous obtenons les runs prolongés. Nous rappelons qu'il existe deux constantes entières α et β telles que

$$\alpha < MinRun < \beta. \tag{4.11}$$

Comme nous ne pouvons pas avoir plus de $\frac{n}{MinRun}$ runs à prolonger, la relation

$$q < \frac{n}{\alpha} \tag{4.12}$$

est triviale.

Soit $C_{ins}(Y)$, le nombre de comparaisons qu'il faut au tri par insertion appelé par TimSort pour trier une séquence Y. Nous définissons pour tout $k \in \mathbb{N}\backslash\{0, 1\}$, $C_{ins}(k) = \max\{C_{ins}(\sigma) \mid \sigma \in \mathfrak{S}_k\}$. Pour toute séquence Y de taille k, nous avons alors $C_{ins}(Y) \leq C_{ins}(k)$. Le coût total en comparaisons des étapes de prolongation est donné par la relation

$$\sum_{i=1}^{q} C_{ins}(\tau_i) < \sum_{i=1}^{q} C_{ins}(MinRun). \tag{4.13}$$

En combinant ces trois dernières équations, nous obtenons

$$\sum_{i=1}^{q} C_{ins}(\tau_i) < \frac{C_{ins}(\beta)}{\alpha} n \tag{4.14}$$

Les deux étapes sont donc toutes les deux en $\mathcal{O}(n)$ comparaisons, et l'étape de création de la pile ne dépassant pas la somme des contributions de ces deux étapes est donc également en $\mathcal{O}(n)$. □

La fusion finale

À la fin de la dernière étape de fusion, il est possible qu'il reste plus d'un run dans la pile. Il est alors nécessaire de fusionner ces runs afin d'obtenir à la fin un unique run restant dans la pile et qui correspond à la séquence triée. Il s'agit de la fusion finale de la ligne 8 à la ligne 11 de l'Algorithme 4. Lors de cette étape, les deux premiers runs situés en haut de la pile sont systématiquement

fusionnés. [3] Le coût total en nombre de comparaisons de ces fusions dans le pire cas est donné par la relation de l'équation (4.15),

$$\sum_{i=1}^{h-1}\sum_{j=i}^{h} s_j - (h-1) = \sum_{i=1}^{h} i \times s_i - s_h - (m-1), \tag{4.15}$$

avec la pile $S = \langle S_1, \ldots, S_h \rangle$ de tailles $\langle s_1, \ldots, s_h \rangle$ qui représentent l'état de la pile au début de l'étape.

Pour déterminer un majorant à la fonction de coût de l'équation (4.15), nous utilisons le lemme 5.

Lemme 5. *Pendant la fusion finale, pour une pile de la forme $S = \langle S_1, \ldots, S_m \rangle$ de tailles $\langle s_1, \ldots, s_m \rangle$, pour tout $i \in [3, h]$ la relation*

$$s_{i-2} > \sum_{j=i}^{h} s_j$$

est vérifiée.

Démonstration. Nous allons prouver ce résultat par récurrence. Rappelons qu'au moment de la fusion finale, l'invariant de TimSort sur la pile de la Proposition 8 doit être vérifié. L'invariant donne le résultat suivant $s_{h-2} > s_{h-1} + s_h > s_h$ et donc valide la proposition du lemme pour $i = h$. De même, l'invariant donne $s_{h-3} > s_{h-2} + s_{h-1} > s_h + s_{h-1}$ par transitivité et donc valide encore une fois la proposition du lemme pour $i = h - 1$. Considérant maintenant que la proposition est vérifiée pour $i \in]k, h]$ avec $k \in]3, h[$, alors

$$\begin{aligned} & s_{k-3} > s_{k-2} + s_{k-1} \text{ (invariant)} \\ \implies & s_{k-3} > \sum_{j=k}^{h} s_j + s_{k-1} \text{ (hypothèse de récurrence)} \\ \iff & s_{k-3} > \sum_{j=k-1}^{h} s_j \end{aligned}$$

La proposition est également vérifiée pour $i \in [k-1, h]$. Par récurrence, on conclut que la proposition est vraie pour tout $i \in [3, m]$. □

En appliquant le Lemme 5, il est possible de démontrer la proposition 12.

Proposition 12. *Pour l'Algorithme 4 et pour toute stratégie de fusions, le coût total de la fusion finale, dans le pire cas, admet une complexité en $\mathcal{O}(n)$ comparaisons.*

Démonstration. En utilisant la première formule de coût donnée à l'équation (4.15),

3. Dans le code d'origine de TimSort, le deuxième run en haut de la pile est fusionné avec son plus petit voisin. Il peut cependant être montré que le premier run est toujours le plus petit si l'invariant est vrai, ce qui est le cas lors de l'appel de la procédure de ces fusions forcées.

il est facile de remarquer que le coût est majoré par la fonction

$$\sum_{i=1}^{h-1}\sum_{j=i}^{h} s_j = \sum_{j=1}^{h} s_j + \sum_{j=2}^{h} s_j + \sum_{i=3}^{h-1}\sum_{j=i}^{h} s_j$$

En appliquant le lemme 5, nous avons alors

$$\sum_{i=1}^{h-1}\sum_{j=i}^{h} s_j < 2n + \sum_{i=3}^{h-1} s_{i-2} < 3n$$

□

Les étapes de fusions

Nous allons enfin étudier le nombre de comparaisons de l'ensemble des étapes de fusions décrites de la ligne 5 à ligne 7 de l'Algorithme 4. Une première remarque à faire est que les propositions 9 et 10 ne suffisent pas à démontrer la complexité de ces étapes de fusion en $\mathcal{O}(n \log n)$. Autrement dit, le fait que la pile ait une taille en $\mathcal{O}(\log(n))$ ne permet pas de faire de conclusion sur la complexité de ces étapes de fusions. Un simple contre-exemple permet de nous convaincre. Considérant un cas particulier de la famille α-StackSort avec $\alpha = \infty$. Dans ce cas, chaque nouveau run est directement fusionné avec son prédécesseur puisque tous les runs sont de tailles finies. Il est trivial de dire que la taille de la pile n'excède pas 2 à tout moment dans l'exécution. Considérons maintenant le cas où chaque run entrant est de taille 1. La première fusion coûte 1 comparaison, la deuxième coûte 2 comparaisons et de manière générale la i-ème coûte i comparaisons. Lorsque l'on fait la somme les coûts des $n-1$ fusions, nous obtenons alors une complexité en $\Theta(n^2)$. Dans ce contre-exemple nous avons donc une pile en taille $\mathcal{O}(1) \subset \mathcal{O}(\log(n))$ et pourtant nous avons une complexité qui est plus importante que $\mathcal{O}(n \log(n))$ comparaisons.

Raisonnement pour 2-StackSort : Pour permettre une meilleure compréhension du raisonnement qui va suivre, nous allons dans un premier temps nous intéresser à l'étude des étapes de fusions pour le cas où la stratégie de fusion est celle du 2-StackSort. 2-StackSort a l'avantage, comparé à TimSort, d'avoir une stratégie de pile simple avec seulement une seule règle. Dans la démonstration, nous utiliserons une méthode d'analyse amortie de la complexité, en attribuant à chaque run, lorsqu'il est inséré dans la pile, un crédit qui devrait être suffisamment grand afin de majorer le coût total des étapes de fusions dans lesquels le run est impliqué. Pour ce faire, le crédit attribué à un run sera ajouté à un compteur C dans l'algorithme 5 qui est équivalent à l'algorithme du 2-StackSort mais qui s'occupera en plus de gérer cette variable C. Celle-ci nous sera utile pour démontrer la complexité des fusions intermédiaires du 2-StackSort.

La propriété invariante décrite dans la proposition 13 peut alors être démontrée.

Proposition 13. *Pour l'algorithme 5 après exécution des lignes 8 et 12, pour $S = \langle S_1, S_2, \ldots, S_\ell \rangle$ de tailles $\langle s_1, s_2, \ldots, s_\ell \rangle$, l'inéquation $C \geq \sum_{i=1}^{\ell} 3i s_i$ est vérifiée.*

Données : X une séquence d'objets comparables
Résultat : X triée

```
1  Initialiser une nouvelle pile de run S vide
2  C ← 0
3  tant que Un run dans X n'a pas été inséré dans S faire
4      Soit R le prochain run, de taille r, dans X
5      S ← ⟨S_1, S_2, ... S_ℓ, R⟩
6      s ← ⟨s_1, s_2, ..., s_ℓ, r⟩
7      ℓ ← ℓ + 1
8      C ← C + 3 × ℓ × r
9      tant que s_{ℓ-1} ≤ 2s_ℓ faire
10         S ← ⟨S_1, S_2, ..., S_{ℓ-1} ⊕ S_ℓ⟩
11         s ← ⟨s_1, s_2, ..., s_{ℓ-1} + s_ℓ⟩
12         C ← C − (s_{ℓ-1} + s_ℓ − 1)
13         ℓ ← ℓ − 1
14 tant que ℓ > 1 faire
15     S ← ⟨S_1, ..., S_{ℓ-2}, S_{ℓ-1} ⊕ S_ℓ⟩
16     ℓ ← ℓ − 1
```

Algorithme 5 : Algorithme du 2-StackSort, nous notons ici par $\langle s_1, s_2, \ldots, s_\ell \rangle$ pour $|S| = \ell$, les tailles des runs stockés à tout instant dans la pile S.

Démonstration. À l'initialisation, la variable C vaut 0 et la pile est vide rendant vraie la proposition. On suppose alors la proposition comme vraie à partir de la ligne 4. Par hypothèse de récurrence, nous avons juste après cette ligne l'inéquation $C \geq \sum_{i=1}^{\ell} 3is_i$ qui est vraie. Après insertion du run R de taille r à la ligne 5, la pile devient $S' = \langle S_1, \ldots, S_\ell, R \rangle$ de tailles $\langle s'_1, \ldots, s'_\ell, s'_{\ell+1} = r \rangle$. Notons $\ell' = \ell + 1$, la nouvelle taille de la pile. Après la ligne 8, la valeur de C est remplacée par

$$C + 3(\ell+1)r \geq \left(\sum_{i=1}^{\ell} 3is_i + 3(\ell+1)r = \sum_{i=1}^{\ell'} 3is'_i \right),$$

vérifiant ainsi la propriété invariante. Si la condition de la boucle à la ligne 9 est vérifiée alors les lignes 10 et 12 seront exécutées. Cependant, par hypothèse de récurrence la propriété invariante est vérifiée avant l'exécution de la ligne 10, une fois cette ligne exécutée, la pile est de la forme $S' = \langle S_1, S_2, \ldots, S_{\ell-1} \oplus S_\ell \rangle$ et de taille $\ell' = \ell - 1$. Nous noterons une fois de plus par s'_i la taille du run à la position i dans S'. Après exécution de la ligne 12, la valeur de C est remplacée par

$$C' = C - (s_{\ell-1} + s_\ell - 1) \geq \sum_{i=1}^{\ell} 3is_i - (s_{\ell-1} + s_\ell - 1).$$

On sait de plus d'après la ligne 9 que $s_{\ell-1} \leq 2s_\ell$ et ainsi $(s_{\ell-1} + s_\ell - 1) \leq 3s_\ell$.

Nous avons alors que

$$C' \geq \left(\sum_{i=1}^{\ell} 3is_i - (s_{\ell-1} + s_\ell - 1) = \sum_{i=1}^{\ell'} 3is'_i + 3s_\ell - (s_{\ell-1} + s_\ell - 1) \right) \geq \sum_{i=1}^{\ell'} 3is'_i,$$

préservant ainsi la propriété invariante après la ligne 12. □

La Proposition 13 permet de démontrer le théorème 8.

Théorème 8. *Pour l'Algorithme 5, les étapes de fusions de la ligne 9 à la ligne 13 demandent au plus $\mathcal{O}(n \log n)$ comparaisons.*

Démonstration. Soit $C = C_+ - C_-$ la valeur de la variable C à la fin de l'exécution de l'algorithme 5, où C_+ est la somme des incrémentations effectuées à la ligne 8 et C_- est la somme des valeurs absolues des décrémentations effectuées à la ligne 12. Comme chaque décrémentation de la ligne 12 majore le coût réel de la fusion effectuée à la ligne 10, alors C_- donne une borne supérieure du coût total des étapes de fusions. De plus, d'après l'invariant de la proposition 13, comme tous les termes de la somme $\sum_{i=1}^{\ell} s_i$ sont positifs, nous avons $C \geq 0$ ce qui implique que

$$C_- \leq C_+.$$

Il suffit donc de prouver que C_+ est en $\mathcal{O}(n \log n)$ pour prouver que C_- l'est également. Soit $R = \langle R_1, \ldots, R_m \rangle$ la séquence des runs de la séquence d'entrée X de tailles $\langle r_1, \ldots, r_m \rangle$. Soit la séquence $\langle h_1 = 1, h2, \ldots, h_m \rangle$, où h_i est la taille de la pile S juste après insertion du run R_i à la ligne 5. Par construction, nous avons

$$C_+ = \sum_{i=1}^{m} 3r_i h_i.$$

Rappelons de plus que la Proposition 9 nous dit que pour tout $i \in [m]$, $h_i < K \log n$ avec K une constante réelle. En combinant cette proposition et la relation précédente, nous obtenons la relation

$$C_+ < \sum_{i=1}^{m} 3Kr_i \log n = 3K \log n \sum_{i=1}^{m} r_i = 3Kn \log n.$$

Nous en concluons alors que le coût total réel en comparaisons des étapes de fusions est en au plus $\mathcal{O}(n \log n)$. □

Raisonnement pour TimSort : Nous pouvons utiliser un raisonnement similaire au précédent pour le cas de TimSort. Nous utiliserons encore une fois une variable auxiliaire C pour nous aider qui sera initialisée à 0 au début et qui sera incrémentée lors de l'insertion d'un run dans la pile, et qui sera décrémentée lorsqu'il y aura des fusions dans la pile. L'algorithme 6 décrit le comportement de TimSort, les lignes colorées en bleu correspondent aux opérations effectuées pour l'analyse sur la variable C. Avec cette version de l'algorithme, nous ne pouvons directement obtenir une conclusion similaire à la preuve de la proposition 13. Notre idée consiste à aller plus loin dans l'exécution d'une étape de fusions. Pour cela, nous procédons à un déroulement de la boucle de l'étape de fusions. Il est possible d'obtenir un tel déroulement car il existe des dépendances

entre certaines des règles de fusions de TimSort. Ces dépendances sont décrites dans les lemmes 6 et 7.

Lemme 6. *Si la règle (ρ_2, μ_2) est activée, alors la contrainte de la règle (ρ_4, μ_4) est vérifiée à l'itération suivante.*

Démonstration. Soit $S = \langle S_1, \ldots, S_\ell \rangle$ de tailles $s = \langle s_1, s_2, \ldots, s_\ell \rangle$ la pile avant que la règle (ρ_2, μ_2) ne soit activée. Comme cette règle est activée, on sait que $s_{\ell-2} \leq s_{\ell-1} + s_\ell$ et $S_{\ell-1}$ est fusionné avec S_ℓ. Après l'activation de (ρ_2, μ_2) la pile est alors de la forme $S' = \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$. Notons $\ell' = \ell - 1$ la nouvelle taille de la pile, ainsi que $s' = \langle s'_1, s'_2, \ldots, s'_{\ell'} \rangle$ les tailles des runs de la pile. Ainsi $s_{\ell-2} \leq s_{\ell-1} + s_\ell \iff s'_{\ell'-1} \leq s'_{\ell'}$ ce qui correspond à la contrainte de la règle (ρ_4, μ_4) à l'itération suivante. □

Lemme 7. *Si la règle (ρ_3, μ_3) est activée, alors la contrainte de la règle (ρ_2, μ_2) est vérifiée à l'itération suivante.*

Démonstration. Soit $S = \langle S_1, \ldots, S_\ell \rangle$ la pile de tailles $\langle s_1, \ldots, s_\ell \rangle$ avant que la règle (ρ_3, μ_3) ne soit activée. Comme cette règle est activée, on sait que $s_{\ell-3} \leq s_{\ell-2} + s_{\ell-1}$ et $S_{\ell-1}$ est fusionné avec S_ℓ. Après l'activation de ρ_3 la pile est alors de la forme $S' = \langle S_1, \ldots, S_{\ell-3}, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$ et sa nouvelle taille est $\ell' = \ell - 1$. Notons $s' = \langle s'_1, \ldots, s'_{\ell'} \rangle$ la taille de ses runs. Nous avons ainsi $s_{\ell-3} \leq s_{\ell-2} + s_{\ell-1} \implies s_{\ell-3} \leq s_{\ell-2} + s_{\ell-1} + s_\ell \iff s'_{\ell'-2} \leq s'_{\ell'-1} + s'_{\ell'}$ ce qui correspond à la contrainte de la règle (ρ_2, μ_2) qui est donc vérifiée pour l'itération qui suit. □

En nous servant du Lemme 6, il est maintenant possible de dérouler la boucle pour le test de la contrainte de la règle (ρ_2, μ_2) à la ligne 17 de l'Algorithme 6. Pour cela, il est nécessaire de vérifier en premier si la règle (ρ_1, μ_1) n'est pas activée à l'itération suivante. Ensuite si (ρ_1, μ_1) n'est pas activée, nous fusionnons les deux runs du haut de la pile qui est l'action effectuée par l'une des règles (ρ_2, μ_2), (ρ_3, μ_3) ou (ρ_4, μ_4). On sait par le lemme, que la dernière est forcément activée si toutes les autres ne le sont pas. En nous servant du Lemme 7, nous procédons également à un déroulement de la boucle du test de la contrainte de la règle (ρ_3, μ_3) à la ligne 22. Pour ce cas, nous devons vérifier que la règle (ρ_1, μ_1) n'est pas activée à l'itération suivante. Si ce n'est pas le cas, nous savons de par le lemme que (ρ_2, μ_2) est alors activée. Dans ce cas, comme (ρ_2, μ_2) est activée, nous pouvons une fois de plus utiliser le Lemme 6 pour appliquer un déroulement supplémentaire de la boucle en procédant de manière identique à ce qui a été vus précédemment. En appliquant tout ces déroulements de boucle, nous obtenons l'Algorithme 7 et qui a donc un comportement similaire à l'Algorithme 6. Nous pouvons maintenant à partir de cet algorithme utiliser un raisonnement similaire à celui que nous avons utilisé pour obtenir la complexité des étapes de fusions du 2-StackSort. Nous démontrons ainsi la proposition 14.

Proposition 14. *Soit $S = \langle S_1, \ldots, S_\ell \rangle$ l'état de la pile utilisée dans l'Algorithme 7 au moment où la boucle à la ligne 11 est exécutée. Soit $s = \langle s_1, \ldots, s_\ell \rangle$ les tailles des runs dans S. La boucle admet l'invariant $C \geq \sum_{i=1}^{\ell} 3is_i \geq 0$.*

Démonstration. Nous devons montrer que l'invariant est préservé pour tous les cas où la variable C est modifiée. À l'initialisation, la pile ne contient aucun run, $\ell = 0$ et $C = 0 = \sum_{i=1}^{\ell} 3is_i$. Pour chacun de ces cas nous noterons par S' de

taille ℓ' l'état de la pile après exécution de ces lignes. Nous noterons également par $s' = \langle s'_1, \ldots, s'_{\ell'} \rangle$ les tailles des runs de la pile S'.

Lors de l'insertion d'un nouveau run dans la pile à la ligne 7, la pile devient $S' = \langle S'_1 = S_1, \ldots, S'_{\ell'-1} = S_\ell, S'_\ell = R \rangle$ de tailles $s' = \langle s'_1 = s_1, \ldots, s'_{\ell'-1} = s_\ell, s'_\ell = r \rangle$. La taille de la pile devient $\ell' = \ell + 1$ et la valeur de C devient

$$C' = C + 3\ell' r \geq \left(\sum_{i=1}^{\ell} 3is_i + 3\ell' r = \sum_{i=1}^{\ell'} 3is_i \right).$$

Nous devons maintenant regarder chaque cas où des runs sont fusionnés. Le premier cas est celui des lignes 12 à 14. La nouvelle pile est alors $S' = \langle S_1, \ldots, S_{\ell-3}, S_{\ell-2} \oplus S_{\ell-1}, S_\ell \rangle$ et la nouvelle valeur du compteur est

$$C' = C - (s_{\ell-2} + s_{\ell-1} - 1).$$

On sait par l'invariant que

$$C \geq \sum_{i=1}^{\ell} 3is_i = \sum_{i=1}^{\ell'} 3is'_i + 3s_{\ell-1} + 3s_\ell.$$

Comme $s_{\ell-2} \leq s_\ell$, on a $3s_{\ell-1} + 3s_\ell - (s_{\ell-2} + s_{\ell-1} - 1) \geq 0$. Ainsi

$$C' \geq \sum_{i=1}^{\ell'} 3is'_i.$$

Le deuxième cas est celui des lignes 18 à 20. La nouvelle pile est alors $S' = \langle S_1, \ldots, S_{\ell-4}, S_{\ell-3} \oplus S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$ et la nouvelle valeur du compteur est

$$C' = C + 2 - \sum_{i=0}^{3} s_{\ell-i}.$$

Par l'invariant,

$$C \geq \sum_{i=1}^{\ell'} 3is'_i + 3s_{\ell-2} + 3s_{\ell-1} + 6s_\ell \Longrightarrow C' \geq \sum_{i=1}^{\ell'} 3is'_i + \left(3s_{\ell-2} + 3s_{\ell-1} + 6s_\ell + 2 - \sum_{i=0}^{3} s_{\ell-i} \right).$$

Comme $s_{\ell-3} \leq s_{\ell-1} + s_\ell$, la somme située entre les parenthèses est positive et ainsi

$$C' \geq \sum_{i=1}^{\ell'} 3is'_i.$$

Le troisième cas est celui des lignes 23 à 25. La nouvelle pile est alors $S' = \langle S_1, \ldots, S_{\ell-3}, S_{\ell-2} \oplus (S_{\ell-1} \oplus S_\ell) \rangle$ et la nouvelle valeur du compteur est

$$C' = C - (s_{\ell-2} + 2s_{\ell-1} + 2s_\ell - 2).$$

Par l'invariant,

$$C \geq \sum_{i=1}^{\ell'} 3is'_i + 3s_{\ell-1} + 6s_\ell \Longrightarrow C' \geq \sum_{i=1}^{\ell'} 3is'_i + (3s_{\ell-1} + 6s_\ell - (s_{\ell-2} + 2s_{\ell-1} + 2s_\ell - 2)).$$

Comme $s_{\ell-2} \leq s_{\ell-1} + s_\ell$, la somme située entre parenthèses est positive et ainsi

$$C' \geq \sum_{i=1}^{\ell'} 3is'_i.$$

Le quatrième cas est celui des lignes 29 à 31 et est résolu de manière similaire au deuxième cas.

Le cinquième cas est celui des lignes 35 à 37. La nouvelle pile est alors $S' = \langle S_1, \ldots, S_{\ell-5}, S_{\ell-4} \oplus S_{\ell-3}, S_{\ell-2} \oplus (S_{\ell-1} \oplus S_\ell) \rangle$ et la nouvelle valeur du compteur est

$$C' = C + 3 - s_{\ell-1} - s_\ell - \sum_{i=0}^{4} s_{\ell-i}.$$

Par l'invariant,

$$C \geq \sum_{i=1}^{\ell'} 3is'_i + 3s_{\ell-3} + 3s_{\ell-2} + 6s_{\ell-1} + 9s_\ell.$$

Comme $s_{\ell-4} \leq \sum_{i=0}^{2} s_{\ell-i}$, la somme $3s_{\ell-3} + 3s_{\ell-2} + 6s_{\ell-1} + 9s_\ell + 3 - s_{\ell-1} - s_\ell - \sum_{i=0}^{4} s_{\ell-i}$ est positive et ainsi

$$C' \geq \sum_{i=1}^{\ell'} 3is'_i.$$

Le sixième cas est celui des lignes 40 à 42. La nouvelle pile est alors $S' = \langle S_1, \ldots, S_{\ell-4}, S_{\ell-3} \oplus (S_{\ell-2} \oplus (S_{\ell-1} \oplus S_\ell)) \rangle$ et la nouvelle valeur du compteur est

$$C' = C + 3 + s_\ell - \sum_{i=0}^{3} (4-i) s_{\ell-i}.$$

Par l'invariant,

$$C \geq \sum_{i=1}^{\ell'} 3is'_i + 3s_{\ell-2} + 6s_{\ell-1} + 9s_\ell.$$

Comme $s_{\ell-3} \leq s_{\ell-2} + s_{\ell-1}$, la somme $3s_{\ell-2} + 6s_{\ell-1} + 9s_\ell + 3 + s_\ell - \sum_{i=0}^{3} (4-i) s_{\ell-i}$ est positive et ainsi

$$C' \geq \sum_{i=1}^{\ell'} 3is'_i.$$

Le septième cas est celui des lignes 45 à 47. La nouvelle pile est alors $S' = \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$ et la nouvelle valeur du compteur est

$$C' = C - (s_{\ell-1} + s_\ell - 1).$$

Par l'invariant,

$$C \geq \sum_{i=1}^{\ell'} 3is'_i + 3s_\ell.$$

Comme $s_{\ell-1} \leq s_\ell$, la somme $3s_\ell - (s_{\ell-1} + s_\ell - 1)$ est positive et ainsi

$$C' \geq \sum_{i=1}^{\ell'} 3is'_i.$$

□

Finalement, nous utilisons cette dernière proposition pour démontrer le Théorème 9.

Théorème 9. *La complexité en nombre de comparaisons de l'étape de fusions, dans le pire cas, de TimSort décrit par l'algorithme 6 est* $\mathcal{O}(n \log n)$.

Démonstration. Pour commencer cette preuve, il faut remarquer dans un premier temps que comme l'algorithme 7 est obtenu en procédant en un déroulement de boucles de l'algorithme 6, les deux sont équivalents et ont donc la même complexité en nombre de comparaisons dans le pire cas. À partir de maintenant l'analyse se fera donc sur l'algorithme 7. Le raisonnement est similaire à la preuve qui a été utilisée pour démontrer le Théorème 8. En effet, il est possible de décomposer une fois encore la variable C en fin d'exécution de l'algorithme comme une somme $C = C_+ - C_-$ avec C_+ la somme des incrémentations et C_- la somme des valeurs absolues des décrémentations. D'après la proposition 14, on sait que $C \geq 0$ à la fin de l'exécution de l'algorithme, ainsi $C_- \leq C_+$. De plus chaque décrémentation de C correspond au coût réel des fusions qui ont eu lieu dans la boucle à la ligne 11. Une borne supérieure de C_+ est donc également une borne supérieure du nombres de comparaisons, dans le pire cas, pour les fusions qui ont lieu dans la boucle. Les incrémentations sont équivalentes à celles du 2-StackSort et en utilisant la Proposition 10, on aboutit à la même conclusion que pour la démonstration du théorème 8. □

Comme la contribution des trois étapes où TimSort effectue des comparaisons sont toutes en $\mathcal{O}(n \log n)$, nous avons alors démontré que le nombre de comparaisons effectuées au total par TimSort admet une borne supérieure en $\mathcal{O}(n \log n)$.

4.4 Conclusion

Dans ce chapitre, nous avons donné une présentation du fonctionnement de l'algorithme de tri TimSort. Nous avons vu qu'il s'agit d'un algorithme qui se montre efficace en pratique, et que c'est très probablement pour ces raisons qu'il a fini par être implanté dans les librairies standards de langages de programmation populaires comme Java et Python.

Du point de vue théorique, l'état de l'art concernant TimSort est pauvre et la plupart des articles que nous pouvons trouver à son sujet proviennent du domaine de l'ingénierie.

D'une certaine manière, TimSort illustre le retard qu'il peut exister en informatique entre la recherche et l'ingénierie. En ingénierie, une solution peut être choisie si elle s'est montrée efficace lors de tests. Nous pensons que TimSort, qui s'est montré efficace bien que nous avons peu de résultats théoriques à son sujet, est un exemple de ce genre de pratiques.

Ce choix n'est pas sans risques cependant, TimSort comportait des bugs comme nous avons pu le voir dans la section 4.3.4. Il n'y a, a priori, plus de bugs concernant la version Python de l'algorithme. La version Java a cependant, à l'heure où sont écrites ces lignes, toujours des bugs qui peuvent déclencher des exceptions et faire s'arrêter la machine virtuelle.

Nous avons apporté, à notre connaissance, une des premières contributions théoriques concernant cet algorithme.

Le premier résultat que nous avons obtenu concerne la complexité dans le pire cas de l'algorithme en nombre de comparaisons. Cette dernière a été annoncée par Tim Peters en $\mathcal{O}(n \log n)$. Nous n'avons cependant jamais trouvé de preuves de cette complexité dans la littérature, et il était donc important de vérifier la véracité de cette dernière. Pour obtenir cette démonstration, nous nous sommes appuyés sur un raisonnement d'analyse amortie que nous avons introduit au cours du chapitre 3. Le raisonnement que nous proposons a depuis été grandement simplifié. Le lecteur intéressé pourra trouver l'intégralité du raisonnement sur nos travaux publiés pour la conférence ESA [8]. L'argument garde une approche comptable, mais cette fois nous attribuons deux types de crédits aux éléments d'un run. En fonction de la règle qui est violée un type de crédit est utilisé en particulier. Cette méthode permet de manière élégante d'éviter de faire les déroulements de boucles que nous avons effectués dans la démonstration présentée dans la section 4.3.5.

Dans un deuxième temps, nous avons défini une classe d'algorithmes qui permet de généraliser TimSort. Dans cette famille d'algorithmes, nous gardons la notion de pile de runs dans laquelle nous insérons les runs croissants ou décroissants de la séquence d'entrée. Les runs dans cette pile sont également fusionnés de la même manière. La différence entre les algorithmes de cette famille provient de leurs stratégies de fusions. Naturellement, TimSort a sa propre stratégie de fusions qui s'inspire de la suite de Fibonacci, et nous avons également présenter les algorithmes de type α-StackSort qui ne contenaient qu'une seule règle dépendante d'un paramètre réelle $\alpha > 1$. Cette stratégie s'inspire, quant à elle, d'une suite géométrique de raisons α.

Il reste des problèmes ouverts concernant TimSort. Nous avons vu que ce dernier appartenait à la catégorie des tris adaptatifs. Ces algorithmes parviennent à trier plus efficacement des séquences lorsque ces dernières sont déjà partiellement triées. En particulier dans le cas de TimSort, moins il y a de runs et plus il sera efficace. Nous avons vu dans les travaux de Manilla qu'il existe une notion d'optimalité en fonction d'une mesure de désordre. Par exemple, le tri fusion naturel de Knuth est optimal, car il trie une séquence de taille n à r runs en $\mathcal{O}(n + n \log r)$ comparaisons. Une question que nous nous posions alors était de savoir si TimSort parvenait également à trier une séquence en $\mathcal{O}(n + n \log r)$ avec r le nombre de runs entrant dans la pile de fusions. La réponse a cette question a depuis été trouvée, et elle est affirmative [8].

Un autre problème ouvert concerne le cas de l'exploitation du cache par l'algorithme. Dans sa note, Tim Peters dit avoir choisi les règles de sa stratégie de fusions de façon à exploiter au mieux le cache des ordinateurs modernes tout

en ne faisant pas exploser le nombre de comparaisons. Il faut alors faire des fusions rapidement pour exploiter les données encore en cache, mais sans trop de précipitations. Nous n'avons pour le moment pas regarder si TimSort est aussi efficace que son auteur le désirerait de ce point de vue, mais il s'agit d'un problème intéressant à regarder. Nous pouvons par exemple choisir différents modèles de cache et essayer de compter le nombre d'erreurs de cache que l'algorithme produit à l'exécution. Il est également possible d'obtenir des résultats expérimentaux à l'aide de librairies comme par exemple PAPI.

Pour finir, une analyse de l'algorithme dans le cas moyen avec le choix d'une distribution uniforme risque de ne pas donner de résultats intéressants comme pour la plupart des algorithmes de tris adaptatifs. Il pourrait cependant être intéressant de faire l'analyse dans le cas moyen avec des distributions bien choisies qui devraient avoir la propriété de s'approcher de ce qu'on pourrait avoir en entrée en pratique mais qui devraient rester simple à manipuler d'un point de vue mathématique.

Données : X une séquence d'objets comparables
Résultat : X triée

Initialiser une nouvelle pile de run S vide
Soit s une séquence vide
$\ell \leftarrow 0$
$C \leftarrow 0$
tant que *Un run dans X n'a pas été inséré dans S* **faire**
    Soit R le prochain run, de taille r, dans X
    $S \leftarrow \langle S_1, \ldots, S_\ell, R \rangle$
    $s \leftarrow \langle s_1, \ldots, s_\ell, r \rangle$
    $\ell \leftarrow \ell + 1$
    $C \leftarrow C + 3\ell r$
    tant que *Une règle de la stratégie de fusion est activée* **faire**
        si $s_{\ell-2} \leq s_\ell$ **alors**
            $S \leftarrow \langle S_1, \ldots, S_{\ell-3}, S_{\ell-2} \oplus S_{\ell-1}, S_\ell \rangle$
            $s \leftarrow \langle s_1, \ldots, s_{\ell-3}, s_{\ell-2} + s_{\ell-1}, s_\ell \rangle$
            $C \leftarrow C - (s_{\ell-2} + s_{\ell-1} - 1)$
            $\ell \leftarrow \ell - 1$
        sinon si $s_{\ell-2} \leq s_{\ell-1} + s_\ell$ **alors**
            $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
            $s \leftarrow \langle s_1, \ldots, s_{\ell-2}, s_{\ell-1} + s_\ell \rangle$
            $C \leftarrow C - (s_{\ell-1} + s_\ell - 1)$
            $\ell \leftarrow \ell - 1$
        sinon si $s_{\ell-3} \leq s_{\ell-2} + s_{\ell-1}$ **alors**
            $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
            $s \leftarrow \langle s_1, \ldots, s_{\ell-2}, s_{\ell-1} + s_\ell \rangle$
            $C \leftarrow C - (s_{\ell-1} + s_\ell - 1)$
            $\ell \leftarrow \ell - 1$
        sinon si $s_{\ell-1} \leq s_\ell$ **alors**
            $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
            $s \leftarrow \langle s_1, \ldots, s_{\ell-2}, s_{\ell-1} + s_\ell \rangle$
            $C \leftarrow C - (s_{\ell-1} + s_\ell - 1)$
            $\ell \leftarrow \ell - 1$
tant que $\ell > 1$ **faire**
    $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
    $\ell \leftarrow \ell - 1$

Algorithme 6 : Algorithme du TimSort

Données : X une séquence d'objets comparables
Résultat : X triée

Initialiser une nouvelle pile de run S vide
Soit s une séquence vide
$\ell \leftarrow 0$
$C \leftarrow 0$
tant que *Un run dans X n'a pas été inséré dans S* **faire**
    Soit R le prochain run, de taille r, dans X
    $S \leftarrow \langle S_1, \ldots, S_\ell, R \rangle$
    $s \leftarrow \langle s_1, \ldots, s_\ell, r \rangle$
    $\ell \leftarrow \ell + 1$
    $C \leftarrow C + 3\ell r$
    tant que *Une règle de la stratégie de fusion est activée* **faire**
        si $s_{\ell-2} \leq s_\ell$ **alors**
            $C \leftarrow C - (s_{\ell-2} + s_{\ell-1} - 1)$
            $S \leftarrow \langle S_1, \ldots, S_{\ell-3}, S_{\ell-2} \oplus S_{\ell-1}, S_\ell \rangle$
            $s \leftarrow \langle s_1, \ldots, s_{\ell-3}, s_{\ell-2} + s_{\ell-1}, s_\ell \rangle$
            $\ell \leftarrow \ell - 1$
        sinon si $s_{\ell-2} \leq s_{\ell-1} + s_\ell$ **alors**
            si $s_{\ell-3} \leq s_{\ell-1} + s_\ell$ **alors**
                $C \leftarrow C + 2 - \sum_{i=0}^{3} s_{\ell-i}$
                $S \leftarrow \langle S_1, \ldots, S_{\ell-4}, S_{\ell-3} \oplus S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
                $s \leftarrow \langle s_1, \ldots, s_{\ell-4}, s_{\ell-3} + s_{\ell-2}, s_{\ell-1} + s_\ell \rangle$
                $\ell \leftarrow \ell - 2$
            sinon
                $C \leftarrow C - (s_{\ell-2} + 2s_{\ell-1} + 2s_\ell - 2)$
                $S \leftarrow \langle S_1, \ldots, S_{\ell-3}, S_{\ell-2} \oplus (S_{\ell-1} \oplus S_\ell) \rangle$
                $s \leftarrow \langle s_1, \ldots, s_{\ell-3}, s_{\ell-2} + s_{\ell-1} + s_\ell \rangle$
                $\ell \leftarrow \ell - 2$
        sinon si $s_{\ell-3} \leq s_{\ell-2} + s_{\ell-1}$ **alors**
            si $s_{\ell-3} \leq s_{\ell-1} + s_\ell$ **alors**
                $C \leftarrow C + 2 - \sum_{i=0}^{3} s_{\ell-i}$
                $S \leftarrow \langle S_1, \ldots, S_{\ell-4}, S_{\ell-3} \oplus S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
                $s \leftarrow \langle s_1, \ldots, s_{\ell-4}, s_{\ell-3} + s_{\ell-2}, s_{\ell-1} + s_\ell \rangle$
                $\ell \leftarrow \ell - 2$
            sinon
                si $s_{\ell-4} \leq \sum_{i=0}^{2} s_{\ell-i}$ **alors**
                    $C \leftarrow C + 3 - s_{\ell-1} - s_\ell - \sum_{i=0}^{4} s_{\ell-i}$
                    $S \leftarrow \langle S_1, \ldots, S_{\ell-5}, S_{\ell-4} \oplus S_{\ell-3}, S_{\ell-2} \oplus (S_{\ell-1} \oplus S_\ell) \rangle$
                    $s \leftarrow \langle s_1, \ldots, s_{\ell-5}, s_{\ell-4} + s_{\ell-3}, s_{\ell-2} + S_{\ell-1} + S_\ell \rangle$
                    $\ell \leftarrow \ell - 3$
                sinon
                    $C \leftarrow C + 3 + s_\ell - \sum_{i=0}^{3} (4-i) s_{\ell-i}$
                    $S \leftarrow \langle S_1, \ldots, S_{\ell-4}, S_{\ell-3} \oplus (S_{\ell-2} \oplus (S_{\ell-1} \oplus S_\ell)) \rangle$
                    $s \leftarrow \langle s_1, \ldots, s_{\ell-4}, s_{\ell-3} + s_{\ell-2} + s_{\ell-1} + s_\ell \rangle$
                    $\ell \leftarrow \ell - 3$
        sinon si $s_{\ell-1} \leq s_\ell$ **alors**
            $C \leftarrow C - (s_{\ell-1} + s_\ell - 1)$
            $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
            $s \leftarrow \langle s_1, \ldots, s_{\ell-2}, s_{\ell-1} + s_\ell \rangle$
            $\ell \leftarrow \ell - 1$
tant que $\ell > 1$ **faire**
    $S \leftarrow \langle S_1, \ldots, S_{\ell-2}, S_{\ell-1} \oplus S_\ell \rangle$
    $\ell \leftarrow \ell - 1$

Algorithme 7 : Algorithme du TimSort avec déroulage de boucle

Chapitre 5

Exploitation de la prédiction de branchement

Sommaire

Dans ce chapitre, nous allons nous intéresser à l'influence de la prédiction de branchement sur les performances de divers algorithmes. Rappelons que le sujet de la prédiction de branchements relève du domaine de l'architecture qui a été abordé au cours du chapitre 3. Il s'agit également d'une première occasion pour nous d'affiner le modèle RAM qui fait des simplifications concernant le coût des instructions de branchement. Nous allons, en plus des coûts d'opérations basiques comme les comparaisons, nous intéresser à un nouveau type de coût, à savoir le nombre d'erreurs de prédiction. Nous distinguons donc de cette façon les coûts des opérations classiques du modèle RAM et les coûts issus de la prédiction de branchement. Le sujet de la prédiction de branchement a déjà été brièvement étudié par le passé mais principalement d'un point de vue pratique. Un premier travail expérimental est celui de Biggar et ses co-auteurs [12] qui ont mis en avant l'influence de la prédiction de branchements sur divers algorithmes de tri. Pour obtenir leurs résultats, les auteurs ont utilisé la librairie PAPI [2] que nous utilisons également pour nos expérimentations. Des analyses théoriques d'algorithmes de tri sur le nombre d'erreurs de prédiction ont également été effectuées par Brodal, Fagerberg et Moruz. Ces derniers ont cherché à étudier les compromis entre le nombre de comparaisons et le nombre d'erreurs de prédiction pour le problème du tri [16]. À l'aide d'utilisations d'arbres de décision comme ceux que nous avons vu au cours du chapitre précédent, ils montrent par exemple que si un algorithme trie une séquence de taille n en $\mathcal{O}(dn \log n)$ comparaisons, alors le nombre d'erreurs de prédiction est $\Omega(n \log_d n)$. Ce théorème est vrai quel que soit le modèle de prédicteur utilisé. Ces mêmes auteurs [14] ont également étudié l'influence du nombre d'inversions sur le nombre d'erreurs de prédiction générées à l'exécution de QuickSort. Ces travaux sont à notre connaissance les premiers pour lesquels une analyse théorique a été effectuée dans le cadre de l'utilisation de prédicteurs statiques. Sanders et Winkel [40] ont quant à eux proposé une variante de l'algorithme de tri SampleSort qui dissocie les comparaisons de la prédiction par l'utilisation d'instructions spécifiques au microprocesseur. Ces instructions peuvent s'apparenter à des instructions de mouvement conditionnelle *CMOV*. Similairement, Elmasry, Katajainen et Stenmark ont proposé une variante de MergeSort[22] qui essaye de contourner au maximum l'utilisation des instructions de branchements.

Kaligosi et Sanders [28] ont publié des travaux sur une analyse poussée du nombre d'erreurs de prédiction produites en moyenne par des prédicteurs dynamiques locaux. Cette étude a été faite dans le cadre de l'étude de QuickSort. Dans ce papier, les auteurs exploitent le fait que ces prédicteurs vont se comporter comme une chaîne de Markov avec de bonnes propriétés qui convergent rapidement vers une unique distribution stationnaire. Par la suite, dans ce chapitre, nous utilisons des raisonnements similaires. Au moment où sont écrites ces lignes, la dernière version de Java utilise en plus de l'algorithme TimSort, une version de QuickSort a deux pivots qui, contre toutes attentes, a de bonnes performances en pratique. Les travaux de Martinez, Nebel et Wild [34] ont cependant montré que l'influence de la prédiction de branchements n'est pas suffisante pour expliquer ces bonnes performances.

Brodal et Moruz ont conduit une étude expérimentale [17] sur des arbres binaires de recherches déséquilibrés, mettant en évidence qu'il était possible pour des structures non équilibrées d'être plus efficaces en pratique par rapport au cas où ces structures sont bien équilibrées. Il s'agit ici d'un des premiers travaux à notre connaissance qui essaye de guider les prédicteurs de branchement. Notre

contribution s'approche de ces travaux mais se concentre sur les algorithmes plutôt que sur la structure des données. Nos travaux ont été publiés dans la conférence STACS 2016 [10]. Ce chapitre présente une version détaillée des travaux qui ont été publiés. Les algorithmes étudiés sont principalement composés de quelques instructions de branchement qui sont exécutées à plusieurs reprises par l'action d'une boucle. C'est par exemple le cas de l'exponentiation rapide qui permet de calculer pour un réel x et un entier naturel n la valeur x^n. En plus de fournir une analyse de ces algorithmes sur le nombre d'erreurs de prédiction, nous avons proposé des modifications peu coûteuses de ces derniers afin de guider la prédiction de branchement. Une contribution que nous avons apportée avec ces travaux concerne la prédiction globale. Nous avons proposé une analyse du nombre d'erreurs de prédiction pour la recherche dichotomique dans le cas où la prédiction est effectuée par un prédicteur global. Nous y sommes arrivés en exploitant des propriétés de l'algorithme, mais il s'agit à notre connaissance du premier cas où une analyse a été faite sur ce type de prédicteur.

Dans nos travaux, nous confirmons nos résultats théoriques par des résultats expérimentaux. Bien qu'il est possible de démontrer que pour un certain type de prédicteur, nous gagnons en nombre d'erreurs de prédiction, il est difficile de conclure que l'implantation sur une machine réelle sera efficace. Les spécificités des architectures dépendent des choix des constructeurs de processeurs qui ne communiquent, pour des raisons de secret industriel, que très peu sur l'utilisation d'un prédicteur particulier. Nous ne pouvons que faire des suppositions en effectuant des benchmarks (comme ce qui a été fait par exemple ici [3]) . Comme nous ne savons pas exactement combien une erreur de prédiction coûte réellement par rapport à une comparaison, il n'est pas non plus évident qu'un algorithme A qui fait plus de comparaisons et moins d'erreurs de prédiction qu'un algorithme B sera plus ou moins efficace en pratique. Il y a bien des compromis pour ces cas, mais il n'est possible d'être sûr que l'algorithme A est plus efficace en temps d'exécution que l'algorithme B sur une machine arbitraire qu'après avoir vérifié expérimentalement que c'était bien le cas. Cependant, nous pensons que l'analyse théorique pour les prédicteurs étudiés nous permet de mettre en évidence qu'il peut y avoir un compromis à trouver en pratique. Les modèles de prédicteurs étudiés sont parmi les plus basiques et les derniers processeurs utilisent des prédicteurs qui parviennent à être encore plus performants que ces derniers. On a donc espoir qu'un algorithme qui se comporte bien en théorie pour ces prédicteurs a des chances d'être encore plus performant en pratique sur un processeur suffisamment récent.

Ce chapitre détaille l'ensemble des travaux cités plus haut. Nous proposons également des pistes pour généraliser les méthodes que nous avons employées pour obtenir les variantes des différents algorithmes que nous étudions dans ce chapitre. Le but est de proposer une abstraction de leurs propriétés intrinsèques. L'exploitation de ces propriétés nous permet aisément de modifier nos algorithmes. Ces modifications peuvent être intéressantes pour guider le prédicteur de branchements.

5.1 Recherche simultanée du minimum et du maximum

Nous avons vu dans le chapitre 3 le problème de la recherche du minimum et du maximum dans une séquence. Pour rappel, nous avons en entrée une séquence d'éléments comparables stockée dans un tableau T. Le problème consiste à obtenir les éléments minimum et maximum de ce tableau. Par exemple, si $T = \langle 7, 5, 2, 4, 14, 9 \rangle$, la sortie sera donnée par le couple $(2, 14)$ car 2 et 14 sont respectivement le minimum et le maximum de T.

5.1.1 Un premier algorithme naïf

Données : Une séquence T de taille n.
Résultat : $\min(T)$, $\max(T)$

```
1 min ← max ← T[1]
2 pour i ← 2 à n faire
3 |   a ← T[i]
4 |   si a < min alors
5 |   |   min ← a
6 |   si a > max alors
7 |   |   max ← a
```

Algorithme 8 : Algorithme de recherche du minimum et du maximum par la méthode naïve. Par la suite, nous nommons cet algorithme NaiveMinmax.

Une première méthode que l'on nomme NaiveMinmax est décrite par l'algorithme 8. L'algorithme procède en faisant un parcours de tous les éléments de la séquence. Le minimum et le maximum des éléments parcourus sont gardés en mémoire. Lorsque tous les éléments sont parcourus, nous obtenons ainsi le minimum et le maximum qui correspondent aux valeurs gardées en mémoire. Nous initialisons à la ligne 1 le minimum et le maximum par la valeur $T[1]$ qui est forcément le minimum et le maximum des éléments parcourus. Le parcours des éléments se fait par la boucle ligne 2 en partant du deuxième et en allant jusqu'au dernier. L'élément que l'on est en train de parcourir est alors comparé aux deux éléments gardés en mémoire. Si l'élément est le plus petit rencontré jusqu'à présent, il faut alors mettre à jour le minimum que l'on a gardé en mémoire. De la même façon, il faut mettre à jour le maximum si l'élément est le plus grand. Comme l'algorithme consiste en un balayage linéaire des $n-1$ éléments restants après l'initialisation, et qu'il fait deux comparaisons par itération, le coût total de l'algorithme est $2(n-1)$. La Figure 5.1 montre un exemple d'exécution de l'algorithme pour une entrée choisie arbitrairement.

5.1.2 Un algorithme optimisé pour le nombre de comparaisons

Nous avons vu lors du chapitre 3, un argument d'adversaire qui nous donne une borne inférieure au problème de la recherche du minimum et du maximum qui est de $\frac{3n}{2}$. Nous allons à présent voir un algorithme qui atteint cette borne

FIGURE 5.1 – Cette figure représente une exécution de l'algorithme de recherche naïve du minimum et du maximum pour la séquence d'entrée $\langle 5, 6, 1, 7, 3, 2, 9, 8, 4\rangle$. À l'initialisation à la première ligne, le premier élément, dont la case est remplie en jaune, est à la fois le minimum et le maximum temporaires. Les lignes suivantes représentent les itérations correspondant à la boucle de parcours des éléments. Pour chacune de ces lignes, l'élément dont la case est remplie en rouge est le maximum temporaire, l'élément dont la case est remplie en vert est le minimum temporaire, les cases en bleu ne sont, ni le maximum ni le minimum et ont déjà été parcourues aux itérations précédentes.

et qui, par conséquence, est optimal. Cet algorithme est une variante de l'algorithme naïf vu précédemment. Nous l'appelons $\frac{3}{2}$-MINMAX et il est décrit par l'Algorithme 9. L'idée consiste toujours à garder en mémoire le minimum et le maximum des éléments déjà explorés, mais la façon dont nous parcourons ces éléments est différente. Dans l'algorithme naïf précédent, nous parcourions chaque élément un par un, cette fois nous allons parcourir deux éléments x et y simultanément par itération de la boucle principale. Soit m et M respectivement les éléments minimum et maximum que l'on connaît de la séquence à une itération. Nous faisons une première comparaison entre x et y. Si $x < y$, nous savons alors que x ne peut pas être le maximum de la séquence et il n'est donc pas nécessaire de comparer x à M. Respectivement, y ne peut pas être le minimum de la séquence et il n'est donc pas nécessaire de comparer y à m. Il nous reste alors deux comparaisons à faire, la première entre x et m, la deuxième entre y et M. Si $x < m$ alors nous affectons x à m et respectivement si $y > M$ nous affectons y à M. Dans le cas où $x > y$, nous faisons les mêmes remarques et les mêmes comparaisons supplémentaires en remplaçant x par y et y par x. Pour résumer, pour chaque itération, nous faisons une première comparaison entre x et y, une comparaisons entre $\min(x, y)$ et m, puis une comparaisons entre $\max(x, y)$ et M. Pour chaque itération nous faisons donc au total 3 comparaisons. À l'initialisa-

Données : Une séquence T de taille n.
Résultat : $\min(T)$, $\max(T)$

```
1  min ← max ← T[n]
2  pour i ← 1 à ⌊n/2⌋ faire
3      a ← T[2i − 1]
4      b ← T[2i]
5      si a < b alors
6          si a < min alors
7              min ← a
8          si b > max alors
9              max ← b
10     sinon
11         si b < min alors
12             min ← b
13         si a > max alors
14             max ← a
```

Algorithme 9 : Algorithme de recherche du minimum et du maximum par une méthode optimale. Par la suite nous nommons cet algorithme $\frac{3}{2}$-MINMAX.

tion, nous choisissons d'affecter $m = M = T[n]$, et nous parcourons les éléments de la séquence dans l'ordre avec un compteur i initialisé à 1 que nous incrémentons de 2 après chaque itération. Cette affectation nous permet de faire le même parcours sans avoir à considérer la parité de n. En contrepartie, si n est pair, nous parcourons deux fois l'élément $T[n]$. Dans tous les cas, le nombre d'itérations effectuées par le parcours est de $\lfloor \frac{n}{2} \rfloor$. Le nombre total de comparaisons effectuées par l'algorithme $\frac{3}{2}$-MINMAX est ainsi $3\lfloor \frac{n}{2} \rfloor = \frac{3n}{2} + \mathcal{O}(1)$. En comparaison avec NAIVEMINMAX, nous faisons de l'ordre de 25% de comparaisons en moins avec $\frac{3}{2}$-MINMAX. Nous donnons un exemple de trace d'exécution pour la même séquence d'entrée que pour l'exemple précédent dans la Figure 5.2.

5.1.3 Un résultat inattendu

Nous avons implanté ces deux algorithmes afin de pouvoir mesurer leurs performances en temps sur une machine. Les listings 8 et 9, disponibles en Annexe B, donnent des extraits du code source de nos implantations en langage C.

Nous avons ensuite compilé ce fichier avec le logiciel *gcc* sur un environnement *gnu/linux*. Nous avons utilisé l'option *-O0* qui permet de compiler le code source sans optimisation. Ce choix nous permet d'obtenir une version de notre code en langage machine qui devrait être assez proche de notre code source initial. La machine utilisée pour faire ces tests fonctionne avec un microprocesseur *Intel(R) Celeron(R) CPU 1037U @ 1.80GHz*. Le listing 10 de l'annexe B donne le code assembleur que l'on obtient sur cette machine après compilation du fichier avec les options *-O0* et *-S*. Ce dernier nous permet de vérifier la proximité qu'a le fichier exécutable avec le code source en C. Les résultats obtenus sur cette machine ne sont pas isolés. Des résultats similaires ont été obtenus sur

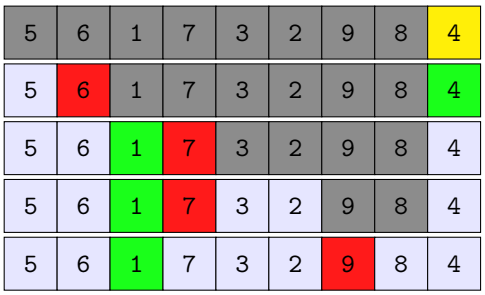


FIGURE 5.2 – Cette figure représente une exécution de l'algorithme de recherche optimisée du minimum et du maximum pour la séquence d'entrée $\langle 5, 6, 1, 7, 3, 2, 9, 8, 4\rangle$. À l'initialisation à la première ligne, le dernier élément, dont la case est remplie en jaune, est à la fois le minimum et le maximum temporaire. Les lignes suivantes représentent les itérations de la boucle principale. Pour chacune de ces lignes, l'élément dont la case est remplie en rouge est le maximum temporaire, l'élément dont la case est remplie en vert est le minimum temporaire, les cases en bleu ne sont, ni le maximum ni le minimum et ont déjà été parcourues aux itérations précédentes.

d'autres machines suffisamment récentes pour avoir des caractéristiques basées sur les dernières avancées en architecture.

Pour générer le tableau en entrée, nous le remplissons initialement de valeurs entières avec la fonction standard *random()* qui a un comportement proche d'une loi uniforme sur l'intervalle $[0, RAND\_MAX]$, avec $RAND\_MAX$ qui est une valeur particulière dépendant de l'environnement. [1] À chaque nouveau test, le tableau est ensuite mélangé avec un algorithme de type Knuth Shuffle afin d'obtenir un mélange uniforme. Le tableau obtenu est alors utilisé en entrée pour les deux algorithmes. Lorsque nous exécutons ce programme, nous obtenons sur la sortie standard les temps pris par les implantations de ces deux algorithmes pour des tableaux de tailles allant de 1000 à 1000×2^{18}, la taille étant doublée à chaque itération. Pour chaque taille, nous avons exécuté ces deux implantations 10 fois afin d'obtenir une mesure moyenne des temps. La Figure 5.3 montre les résultats ainsi obtenus sous forme de courbes de performances.

Lorsque nous observons ces deux courbes, nous pouvons être surpris du résultat. En effet, nous observons que la courbe de l'algorithme optimal en nombre de comparaisons est située en tous points au-dessus de la courbe de l'algorithme naïf. En d'autres termes, l'algorithme naïf s'avère être en pratique plus efficace que l'algorithme optimisé. Si nous considérons que notre machine est basée sur le modèle RAM, ce résultat est surprenant car les deux algorithmes font les mêmes opérations, exceptées pour les comparaisons. Or, nous avons vu que $\frac{3}{2}$-MINMAX fait moins de comparaisons que NAIVEMINMAX. Pour ces deux opérations l'algorithme optimisé fait moins de travail que l'algorithme naïf et devrait par conséquent être le plus rapide des deux. Il nous est donc nécessaire de remettre en question l'hypothèse du modèle RAM. Parmi les principales caractéristiques

1. Sur la machine où ont été réalisé les tests, la valeur de $RAND\_MAX$ est de $2^{31} - 1 \approx 2 \times 10^9$.

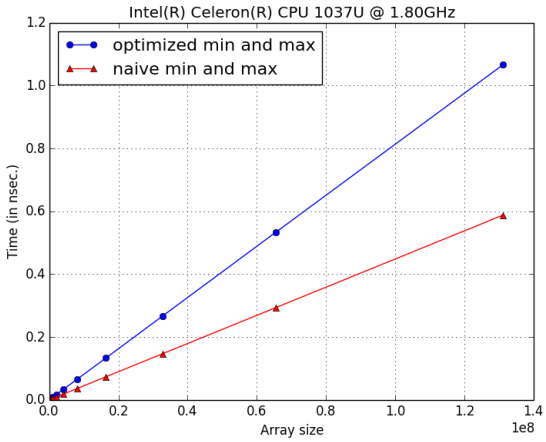


FIGURE 5.3 – Courbe de performances en temps des algorithmes de recherche du minimum et du maximum. L'abscisse correspond à la taille du tableau en entrée, l'ordonnée correspond à la durée moyenne d'exécution de l'algorithme sur 10 essais.

des machines récentes qui diffèrent de ce modèle, nous avons abordé lors du chapitre 3 la *hiérarchie mémoire* et la *prédiction de branchement*. Dans notre implantation, nous avons fait en sorte d'éliminer l'influence de la hiérarchie mémoire sur les performances. Nous faisons exactement un seul accès par élément du tableau qui est ensuite stocké dans des variables temporaires pour faire les comparaisons et les affectations qui suivent. De plus, ces accès se font exactement dans le même ordre. Il est cependant assez évident que la prédiction de branchement diffère grandement entre ces deux algorithmes. Notre intuition à ce sujet provient du fait que le premier test effectué par l'algorithme optimisé est imprédictible dans le cas où l'entrée est générée par une distribution uniforme. En effet, le test qui consiste à comparer deux éléments consécutifs dans le tableau a une probabilité de $\frac{1}{2}$ d'être vrai. Nous avons vérifié en utilisant la librairie PAPI le nombre d'erreurs de cache et d'erreurs de prédiction générées en moyenne par ces deux implantations. La méthode est relativement similaire à celle employée précédemment pour obtenir les performances de temps mais en utilisant des compteurs apportés par la librairie. Les compteurs que nous utilisons sont les mêmes que ceux utilisés par la commande *perf* qui fonctionne de manière similaire à la commande time. Nous n'utilisons pas ici cette commande car, comme pour la commande time, nous ne pouvons faire des mesures sur un bloc de code isolé dans un programme. Les implantations et la génération des entrées restent donc inchangées. Le code source modifié est disponible en Annexe B. Les optimisations au niveau hardware sur le cache font que l'influence du cache est négligeable quelle que soit la taille de l'entrée. Nous observons que le nombre d'erreurs de cache (au niveau L1) est de l'ordre de $10^{-3}\%$ de la taille du tableau en entrée. Nous donnons dans la Table 5.1, le nombre d'erreurs de prédiction générées à l'exécution de ces deux implantations en fonction de la taille de l'entrée. Nous observons une différence très importantes du nombre

Taille de l'entrée	#err. prédictions trois demi	#err. prédictions naïf
1000	263	11
2000	515	12
4000	1003	13
8000	2029	15
16000	3967	14
32000	8023	15
64000	16055	17
128000	32014	15
256000	63950	18
512000	128102	18
1024000	255842	20
2048000	512006	23
4096000	1024469	22
8192000	2047809	25
16384000	4095771	23
32768000	8191399	25
65536000	16382484	26
131072000	32766697	27

TABLE 5.1 – Nombre d'erreurs de prédiction mesuré pour les implantations des deux algorithmes de recherche du minimum et du maximum dans un tableau. Les résultats sont donnés en fonction de la taille de l'entrée.

d'erreurs de prédiction entre ces deux implantations. Sur les tailles que nous avons choisies, nous observons que l'influence de la prédiction de branchement est négligeable pour NAIVEMINMAX avec un nombre d'erreurs de prédiction qui est de l'ordre de $10^{-5}\%$ de la taille de l'entrée. On ne peut pas affirmer la même chose du côté de $\frac{3}{2}$-MINMAX qui fait en moyenne 25% de la taille de l'entrée en nombre d'erreurs de prédiction. Nous pouvons donc pour le moment conjecturer que le nombre d'erreurs de prédiction générées lors de l'exécution de $\frac{3}{2}$-MINMAX est $\frac{n}{4}$ avec n la taille de l'entrée pour le prédicteur utilisé par la machine. Nous allons voir par la suite que c'est bien le cas et que c'est vrai quel que soit le prédicteur utilisé. Nous verrons également des expressions du nombre d'erreurs de prédiction pour le cas de l'algorithme naïf. Nous avons plusieurs expressions car ces dernières dépendent du prédicteur utilisé.

5.1.4 Analyse du nombre d'erreurs de prédiction pour ces deux algorithmes

Nous allons désormais proposer une analyse de ces deux algorithmes en nombre d'erreurs de prédiction. Pour réaliser ce genre d'analyse, il est nécessaire de fixer le prédicteur que nous utilisons. Chaque prédicteur a un comportement différent et donc les erreurs qu'ils font le sont également. Il est donc fort probable que les prédicteurs utilisés dans la pratique se comportent différemment. Les prédicteurs que nous avons choisis pour cette section sont des prédicteurs locaux 1-bit, 2-bits et 3 bits saturés. Pour chaque instruction de branchement dans nos algorithmes, nous associons un prédicteur distinct qui s'occupe de pré-

dire uniquement la prochaine exécution de cette instruction. Afin de rester fidèle à l'expérimentation que nous avons décrite lors de la section précédente, nous considérons que l'entrée est une permutation sous la forme d'un mot de taille n. Cela correspond bien aux entrées de notre expérimentation en remplaçant la valeur de chaque élément par leurs rangs ce qui ne changent pas les résultats et l'ordre des comparaisons effectuées par nos deux algorithmes. La Proposition 15 donne un premier résultat sur le nombre d'erreurs de prédiction effectuées par NaiveMinmax. La preuve repose sur le fait que les comparaisons effectuées aux Lignes 4 et 6 sont en relation avec des statistiques connues et évoquées lors du Chapitre 2 les *min-records* et les *max-records*.

Proposition 15. *Le nombre moyen d'erreurs de prédiction effectuées par* NaiveMinmax*, pour une distribution uniforme sur les tableaux de taille n, est asymptotiquement équivalent à $4 \log n$ pour le prédicteur 1-bit et à $2 \log n$ pour les prédicteurs 2-bits et 3-bits saturés.*

Démonstration. Soit $\mathcal{F}$ la bijection fondamentale qui permet de faire une correspondance entre la représentation en cycle des permutations $\sigma \in \mathfrak{S}_n$ et les max-records de $\mathcal{F}(\sigma)$ vu au cours du Chapitre 2. Comme $\mathcal{F}$ est une bijection, alors pour une variable aléatoire ξ à valeurs réelles, l'identité

$$\mathbb{E}_n[\xi] = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \xi(\sigma) = \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \xi(\mathcal{F}(\sigma))$$

est vérifiée.

Pour commencer, nous allons nous intéresser au comportement du prédicteur 1-bit. Soit σ une permutation de $\mathfrak{S}_n$ dont les cycles, triés dans l'ordre croissant de leurs plus grands éléments, sont $C_1, \ldots, C_m$. La bijection fondamentale appliquée à σ est de la forme $\mathcal{F}(\sigma) = f(C_1) \cdot f(C_2) \ldots f(C_m)$, avec f la fonction qui concatène les éléments d'un cycle en commençant par le plus grand élément. Nous allons maintenant chercher la relation entre les cycles de σ et le nombre d'erreurs de prédiction de $\mathcal{F}(\sigma)$ au cours de l'exécution de la Ligne 6. Nous allons maintenant regarder en détail un cas particulier en fixant l'état initial du prédicteur à NT au début du parcours de $f(C_i)$. Comme le premier élément de $f(C_i)$ est un max-record par construction de $\mathcal{F}(\sigma)$, il provoque une erreur de prédiction et le prédicteur passe à l'état T. Si C_i contient au moins deux éléments, alors son deuxième élément dans $f(C_i)$ est plus petit que le premier, et va à son tour provoquer une erreur de prédiction et le prédicteur retourne alors à l'état NT. Si le cycle contient plus de deux éléments, les éléments qui restent ne provoquent plus d'erreur et laisse l'état inchangé en NT, puisqu'ils sont tous plus petits que le premier élément de $f(C_i)$. Pour résumer, nous avons vu qu'il existait deux cas importants : soit C_i est de taille 1 et le parcours de $f(C_i)$ provoque une erreur de prédiction et fait passer l'état du prédicteur à T, soit C_i est de taille supérieure ou égale à 2 et le parcours de $f(C_i)$ provoque deux erreurs de prédiction et l'état du prédicteur passe à NT. Le même raisonnement peut être fait en supposant que l'état initial du prédicteur est T. Nous résumons l'ensemble de ces cas dans le tableau qui suit en dessous.

état initial	cycle de taille 1		cycle de taille ≥ 2	
	err. pred.	état final	err. pred.	état final
NT	1	T	2	NT
T	0	T	1	NT

Pour déterminer le nombre d'erreurs de prédiction causées par l'entrée $\mathcal{F}(\sigma)$, remarquons qu'en lisant la première ligne, nous observons qu'il y a toujours au moins une erreur lorsque l'état initial est NT. En faisant une lecture des colonnes, nous remarquons que le prédicteur se trouve à l'état NT uniquement lorsque le cycle précédent est de taille au moins 2. Nous avons donc $\mathbf{Cyc}_{\geq 2}(\sigma) + \mathcal{O}(1) = \mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma) + \mathcal{O}(1)$ erreurs de prédiction qui proviennent du cas où le prédicteur commence par cet état. Le terme $\mathcal{O}(1)$ permet de capter l'état initial lors du parcours de $f(C_1)$ qui peut dépendre d'exécutions d'autres programmes dans le passé et qui a donc un caractère aléatoire. Nous remarquons en lisant la dernière colonne qu'il y a toujours au moins une erreur de prédiction lorsque le cycle est de taille supérieure ou égale à 2 pour tout état initial. Nous comptons donc encore une fois $\mathbf{Cyc}_{\geq 2}(\sigma) = \mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma)$ erreurs de prédiction. Si nous faisons la somme de la contribution des cas où le prédicteur est à l'état NT et des cas où les cycles sont de taille supérieure à 2, nous obtenons ainsi la totalité des erreurs de prédiction. Il n'y a pas de terme d'exclusions à soustraire pour le cas où l'état initial est NT et est de taille supérieure à 2 puisque dans ce cas il y a deux erreurs. La quantité d'erreurs de prédiction pour le prédicteur 1-bit associé à la Ligne 6 est donc donnée par la relation

$$2\mathbf{Cyc}(\sigma) - 2\mathbf{Cyc}_1(\sigma) + \mathcal{O}(1).$$

Le même raisonnement peut s'appliquer avec la Ligne 4, il faut dans ce cas utiliser la bijection fondamentale pour les min-records. Nous pouvons résumer les erreurs de prédiction avec le même tableau que pour le cas précédent. Ainsi, le nombre d'erreurs est alors donné par la même relation et la somme de ces deux quantités permet de donner le nombre d'erreurs de prédiction effectuées au total lors de l'exécution de NaiveMinmax avec pour entrée σ qui est donné par la formule

$$4\mathbf{Cyc}(\sigma) - 4\mathbf{Cyc}_1(\sigma) + \mathcal{O}(1).$$

Comme $\mathbb{E}_n[\mathbf{Cyc}] \sim \log n$ et $\mathbb{E}_n[\mathbf{Cyc}_1] \to 1$ quand n tend vers l'infini, cette quantité est donc asymptotiquement équivalent à $4\log(n)$ ce qui nous permet de conclure.

Pour le prédicteur 2-bits saturé, la même méthode peut être utilisée. Comme le prédicteur 2-bits a plus d'état que le 1-bit, il faut faire attention à prendre en compte des tailles de cycles plus grands qu'au raisonnement précédent. Regardons une fois de plus en détail un cas particulier. Supposons que l'état initial est NT au début du parcours de $f(C_i)$. Une erreur de prédiction se produit alors au parcours du premier élément et le prédicteur passe à l'état T. Si C_i a une longueur d'au moins 2, alors le prochain élément provoque également une erreur de prédiction et le prédicteur retourne à l'état NT. Si la longueur est d'au moins 3, alors le prédicteur passe à l'état SNT et ne provoque pas plus d'erreurs de prédiction. Nous avons résumé une fois de plus l'ensemble de ces cas dans le tableau ci-dessous.

état initial	cycle de taille 1		cycle de taille 2		cycle de taille 3		cycle de taille ≥ 4	
	err. pred.	état final	err. pred.	état final	err. pred.	état final	err. pred.	état final
SNT	1	NT	1	SNT	1	SNT	1	SNT
NT	1	T	2	NT	2	SNT	2	SNT
T	0	ST	1	T	2	NT	2	SNT
ST	0	ST	1	T	2	NT	2	SNT

Soit $\chi(\mathcal{F}(\sigma))$ le nombre d'erreurs de prédiction provoquées lors du test de la Ligne 6. Nous allons chercher un encadrement de cette quantité. La minoration se fait en lisant la dernière colonne du tableau tandis que la majoration se fait en lisant les lignes. Nous remarquons qu'il y a au moins une erreur de prédiction lorsque le cycle est de taille au moins 4, et donc nous pouvons directement en déduire que $\mathbf{Cyc}_{\geq 4}(\sigma) \leq \chi(\mathcal{F}(\sigma))$. Pour la majoration, nous avons deux cas : l'état initial est SNT et le prédicteur fait au plus 1 erreur de prédiction, l'état initial n'est pas SNT et le prédicteur fait au plus 2 erreurs de prédiction. L'état initial est SNT si le cycle parcouru précédemment est de taille au moins 2. Le nombre d'erreurs de prédiction pour ce cas ne dépasse donc pas $\mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma) + \mathcal{O}(1)$ avec le terme $\mathcal{O}(1)$ qui permet de capter les effets de bords associés à l'état initial au début de l'exécution de l'algorithme et de la taille du dernier cycle. L'état initial n'est pas SNT si le cycle qui le précède est de taille inférieure à 3. Le nombre d'erreurs pour le deuxième cas ne dépasse donc pas $2\mathbf{Cyc}_{\leq 3}(\sigma)$. En faisant la somme des contributions de ces deux cas nous obtenons un majorant du nombre d'erreurs de prédiction qui est donné par la relation

$$\chi(\mathcal{F}(\sigma)) \leq \mathbf{Cyc}(\sigma) - \mathbf{Cyc}_1(\sigma) + 2\mathbf{Cyc}_{\leq 3}(\sigma) + \mathcal{O}(1) \leq \mathbf{Cyc}_{\geq 4}(\sigma) + 3\mathbf{Cyc}_{\leq 3}(\sigma) + \mathcal{O}(1).$$

Rappelons que nous avons $\mathbb{E}_n[\mathbf{Cyc}_{\geq 4}] \sim \log n$ et $\mathbb{E}_n[\mathbf{Cyc}_{\leq 3}] = \mathcal{O}(1)$ quand n tend vers l'infini. En appliquant le même raisonnement pour la Ligne 4, nous pouvons conclure que pour le prédicteur 2-bits saturé nous avons asymptotiquement $2 \log n$ erreurs de prédiction.

Nous procédons de même avec le prédicteur 3-bits. L'ensemble des cas possibles est détaillé dans le tableau ci-dessous.

état initial	cycle de taille 1		cycle de taille 2		cycle de taille 3		cycle de taille 4	
	err. pred.	état final	err. pred.	état final	err. pred.	état final	err. pred.	état final
$SSSNT$	1	$SSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
$SSNT$	1	SNT	1	$SSNT$	1	$SSSNT$	1	$SSSNT$
SNT	1	NT	1	SNT	1	$SSNT$	1	$SSSNT$
NT	1	T	2	NT	2	SNT	2	$SSNT$
T	0	ST	1	T	2	NT	2	SNT
ST	0	SST	1	ST	2	T	3	NT
SST	0	$SSST$	1	SST	2	ST	3	T
$SSST$	0	$SSST$	1	SST	2	ST	3	T

état initial	cycle de taille 5		cycle de taille 6		cycle de taille 7		cycle de taille ≥ 8	
	err. pred.	état final	err. pred.	état final	err. pred.	état final	err. pred.	état final
$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
$SSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
SNT	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$	1	$SSSNT$
NT	2	$SSSNT$	2	$SSSNT$	2	$SSSNT$	2	$SSSNT$
T	2	$SSNT$	2	$SSSNT$	2	$SSSNT$	2	$SSSNT$
ST	3	SNT	3	$SSNT$	3	$SSSNT$	3	$SSSNT$
SST	4	NT	4	SNT	4	$SSNT$	4	$SSSNT$
$SSST$	4	NT	4	SNT	4	$SSNT$	4	$SSSNT$

Nous allons chercher une fois de plus un nouvel encadrement de la quantité du nombre d'erreurs de prédiction effectuées par NaiveMinmax avec pour entrée σ. La lecture de la dernière colonne nous donne, une fois de plus, un minorant simple qui est le nombre de cycles de taille supérieur à 8. Pour le majorant nous allons utiliser la même astuce qui a été utilisée pour le prédicteur 2-bits. Nous pouvons remarquer en lisant la première ligne que nous faisons au plus 1

erreur de prédiction quand le prédicteur est à l'état initial SSSNT. Le prédicteur est dans cet état au plus $\mathbf{Cyc}(\sigma) + \mathcal{O}(1)$ fois avec le terme $\mathcal{O}(1)$ qui capte les effets de bords. Si l'état initial du prédicteur n'est pas SSSNT, alors le nombre d'erreurs est au plus 4. Le prédicteur n'est pas dans cet état, au début de chaque cycle, au plus $\mathbf{Cyc}_{\leq 7}$ fois. Quand on combine ces deux cas, nous obtenons une borne supérieure du nombre de prédictions qui est donné par la relation

$$\chi(\mathcal{F}(\sigma)) \leq \mathbf{Cyc}(\sigma) + 4\mathbf{Cyc}_{\leq 7}(\sigma) + \mathcal{O}(1) \leq \mathbf{Cyc}_{\geq 8}(\sigma) + 5\mathbf{Cyc}_{\leq 7}(\sigma) + \mathcal{O}(1).$$

Une fois de plus en ajoutant la contribution de la Ligne 4, nous avons asymptotiquement un total de $2 \log n$ erreurs de prédiction. □

Intéressons nous maintenant au cas du nombre d'erreurs de prédiction produites par l'algorithme $\frac{3}{2}$-MINMAX.

Le test à la Ligne 3 du $\frac{3}{2}$-MINMAX est vraie si il y a une montée à la position $i + 1$. Il faut alors analyser les alternances de montées et de descentes pour obtenir la proposition suivante.

Proposition 16. *Le nombre d'erreurs de prédiction effectuées par le* $\frac{3}{2}$-MINMAX*, pour une distribution uniforme sur un tableau de taille* n*, est asymptotiquement équivalent à* $\frac{n}{4}$ *pour tous les prédicteurs* 1*-bits,* 2*-bits,* 3*-bits saturés.*

Démonstration. Il est facile de montrer que le nombre d'erreurs de prédiction engendrées par les instructions aux lignes 6,8,11 et 13 sont en $\mathcal{O}(\log n)$ pour ces prédicteurs. En effet, ces lignes ne s'activent uniquement que quand les éléments comparés sont des records. Elles ne font donc pas plus d'erreurs de prédiction que les instructions de branchements de NAIVEMINMAX dont on a vu le comportement asymptotique dans la Proposition 15. Concernant l'instruction de la Ligne 5, sous le modèle uniforme comme il y a autant de chance de trouver les deux éléments comparés dans n'importe quel ordre, une erreur de prédiction aura lieu avec probabilité $\frac{1}{2}$ quel que soit le prédicteur utilisé. En effet, comme ces tests sont indépendants à chaque itération et qu'il y a une chance sur deux d'avoir le résultat vrai ou faux pour ce test, celui-ci est imprédictible. Ainsi en moyenne, ce test provoque $\frac{n}{4}$ erreurs de prédiction. En faisant la somme de toutes ces contributions, nous parvenons à la conclusion annoncée. □

La Table 5.2 donne similairement à la Table 5.1 obtenu avec PAPI le nombre d'erreurs de prédiction pour ces trois prédicteurs de ces deux algorithmes. Ces résultats ont été obtenus en simulant ces prédicteurs à l'aide de la librairie predictors. Une description de la librairie ainsi que le code source de cette simulation sont disponibles en Annexe A dans le Listing 6.

Une conséquence de la Proposition 16 est que le nombre d'erreurs de prédiction est négligeable devant le nombre de comparaisons pour le cas de NAIVEMINMAX. Les observations de la Table 5.1 et de la Table 5.2 semblent confirmer ce résultat.

Nous pensons que cette différence du nombre d'erreurs de prédiction entre les deux algorithmes est la raison qui explique cette différence de performance, étant donné que les erreurs de prédiction coûtent beaucoup de cycles de CPU et que les comparaisons de deux réels ne sont pas des opérations trop coûteuses. Nous avons également testé la robustesse de nos programmes aux optimisations proposées par le compilateur gcc. Dans le cas de l'optimisation -O3, tous les

branchements conditionnels excepté celui du test de la Ligne 5 du $\frac{3}{2}$-MINMAX sont remplacés par des mouvements conditionnels qui sont insensibles aux effets de la prédiction. Ainsi donc le $\frac{3}{2}$-MINMAX continue de causer environ $\frac{n}{4}$ erreurs de prédiction en moyenne.

Prédicteurs	1-bit		2-bit		3-bit	
Taille de l'entrée	#err. trois demi	#err. naïf	#err. trois demi	#err. naïf	#err. trois demi	#err. naïf
1000	274	25	274	16	264	16
2000	533	31	520	18	525	17
4000	1028	29	1019	21	1015	18
8000	2022	33	2014	22	2012	20
16000	4008	33	4008	22	3993	21
32000	8039	36	8060	22	8019	22
64000	16030	41	15975	26	16042	24
128000	32042	44	32020	26	32060	24
256000	64078	49	64149	26	64008	29
512000	128113	47	127957	30	128071	29
1024000	256092	53	256151	30	256068	29
2048000	511992	57	512108	30	511862	30
4096000	1023857	53	1023987	35	1024652	32
8192000	2047648	62	2047923	32	2047111	33
16384000	4095846	62	4095597	37	4095816	36
32768000	8192121	63	8192497	37	8192314	40
65536000	16384437	66	16384640	36	16383884	35
131072000	32767836	75	32769580	41	32768201	40

TABLE 5.2 – Tableau du nombre d'erreurs de prédiction engendrées par les algorithmes NAIVEMINMAX et $\frac{3}{2}$-MINMAX sur des prédicteurs 1-bit, 2-bits saturé et 3-bits saturé.

5.2 Exponentiation rapide

Dans la section précédente, nous avons vu qu'une instruction de branchement dont les deux issues sont équiprobables rend la prédiction de branchement impossible. Ces instructions sont alors très coûteuses lorsqu'elles sont exécutées par un processeur RISC et il serait préférable de les éviter en pratique. Nous avons deux possibilités qui nous viennent à l'esprit pour y parvenir. La première est de contourner la prédiction de branchements en utilisant des instructions spéciales du microprocesseur comme par exemple les mouvements conditionnels *CMOV*. La deuxième méthode, que nous avons choisie, consiste à modifier les algorithmes pour faire en sorte que les instructions de branchements ne soient pas imprédictibles. Cette démarche va à l'encontre des paradigmes qui ont permis de créer la grande majorité des algorithmes classiques populaires. Dans ces derniers, les branchements ont bien souvent intérêt à avoir une équiprobabilité des issues. Cette équiprobabilité permet d'obtenir dans nombreux cas des complexités optimales. C'est par exemple le cas de $\frac{3}{2}$-MINMAX que nous avons vu dans la section précédente. Cet algorithme est optimal en nombre de comparaisons mais a besoin de faire une comparaison qui a autant de chance d'être vrai ou fausse dans le cas où l'entrée suit une loi uniforme. Nous avons vu que NAIVEMINMAX est plus rapide en pratique par rapport à $\frac{3}{2}$-MINMAX bien que ce premier fait plus de comparaisons que ce dernier. Nous allons donc essayer de modifier des algorithmes classiques dont les instructions de branchements posent problème à la prédiction. Cette modification ne se fera pas toujours sans un coût supplémentaire, et il sera donc nécessaire de trouver un compromis pour que ce coût ne soit pas trop important. Dans cette section, nous allons nous intéresser en particulier à l'algorithme d'exponentiation rapide qui prend en entrée un élément x appartenant à un monoïde $(M, .)$ et un entier naturel n et donne en sortie la valeur x^n.

5.2.1 Présentation de l'exponentiation rapide et identification du problème

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1 r ← 1
2 y ← x
3 k ← n
4 tant que k ≠ 0 faire
5 |   si k est impair alors
6 |   |   r ← r · y
7 |   k ← ⌊k/2⌋
8 |   y ← y²
9 retourner r
```

Algorithme 10 : Algorithme de l'exponentiation rapide classique

Nous avons donné dans l'Algorithme 10, une version en pseudo-code de l'exponentiation rapide dont nous allons décrire dès à présent le fonctionnement. À l'initialisation, nous créons trois variables r, y et k qui sont respectivement affectées par les valeurs $1, x$ et n avec 1 qui est l'élément neutre du monoïde M. L'algorithme consiste principalement en une boucle qui s'arrête lorsque la valeur de k est 0. À chaque itération, nous élevons y au carré et nous divisons k par deux. Il est donc ici évident que la complexité en nombre d'itérations est asymptotiquement équivalente à $\log_2 n$. Pour chaque itération nous avons deux cas, soit k est impair et nous multiplions r par y, soit k est pair et nous ne faisons rien de plus. Lorsque la boucle s'arrête nous savons que $r = x^n$ et nous retournons donc le résultat. Pour démontrer que cet algorithme fonctionne correctement, nous pouvons faire une preuve en utilisant l'invariant du Lemme 8.

Lemme 8. *Au début de la ligne 4, l'équation suivante est toujours vérifiée :*

$$x^n = r \cdot y^k$$

Démonstration. À l'initialisation, comme $r = 1$, $y = x$ et $k = n$, il est évident que l'équation est vérifiée. Supposons maintenant que l'invariant est vrai au début d'une certaine itération. Nous avons alors $x^n = r \cdot y^k$ qui est vraie. Nous avons deux cas possibles. Soit k est pair, dans ce cas nous pouvons ré-écrire l'expression de droite sous la forme :

$$r \cdot y^k = r \cdot y^{2(\lfloor \frac{k}{2} \rfloor)} = r \cdot (y^2)^{\lfloor \frac{k}{2} \rfloor}.$$

Dans ce cas l'invariant est bien rétabli en affectant $y \cdot y$ à y et $\lfloor \frac{k}{2} \rfloor$ à k.

Soit k est impair, dans ce cas nous pouvons ré-écrire l'expression de droite sous la forme :

$$r \cdot y^k = r \cdot y^{2(\lfloor \frac{k}{2} \rfloor)+1} = (r \cdot y) \cdot (y^2)^{\lfloor \frac{k}{2} \rfloor}.$$

Dans ce cas l'invariant est également rétabli en affectant en plus des affectations du premier cas la valeur $r \cdot y$ à r. □

Lorsque la boucle s'arrête, l'invariant est alors vrai et $k = 0$. Nous savons

donc que $r \cdot y^k = x^n$. Or comme $k = 0$, $y^k = 1$ et donc $r = x^n$.

Nous avons donc avec l'Algorithme 10, une méthode rapide pour calculer x^n. Cependant en ce qui concerne la prédiction de branchement nous avons un problème. Supposons que le paramètre n est distribué aléatoirement et uniformément dans l'intervalle $[0, 2^K[$ pour un entier $K \in \mathbb{N}$. Nous avons alors une chance sur deux que chaque bit de n une fois écrit en binaire soit 0 ou 1. Or le test de parité revient à tester pour la i-ème itération si le i-ème bit de n est 0 ou 1. Autrement dit pour cette distribution, le test de parité de la ligne 5 est imprédictible. La Proposition 17 donne la complexité du nombre d'erreurs de prédiction pour plusieurs prédicteurs locaux décrit au Chapitre 3.

Proposition 17. *Pour l'algorithme 10, si l'entrée n est choisie uniformément dans l'intervalle $[0, 2^K[$ pour un entier $K \in \mathbb{N}$ arbitraire, alors quel que soit le prédicteur, le nombre d'erreurs de prédiction est asymptotiquement équivalente à $0.5 \log_2 n$.*

Démonstration. Comme l'instruction de branchement de la ligne 5 est imprédictible et qu'il y a en tout $\log_2 n$ itérations, ce résultat est trivial. □

5.2.2 Les modifications apportées à l'algorithme classique

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1  r ← 1
2  y ← x
3  k ← n
4  tant que k ≠ 0 faire
5  |  si k est impair alors
6  |  |  r ← r · y
7  |  y ← y²
8  |  si ⌊k/2⌋ est impair alors
9  |  |  r ← r · y
10 |  y ← y²
11 |  k ← ⌊k/4⌋
12 retourner r
```

Algorithme 11 : Algorithme de l'exponentiation rapide classique avec déroulement de la boucle principale

Nous allons maintenant voir comment nous avons modifié l'Algorithme 10, que nous nommons dorénavant ClassicalPow, pour qu'il soit capable de guider le prédicteur de branchements. Il semble difficile de pouvoir rajouter directement un biais au branchement de la ligne 5. Nous avons donc procédé en plusieurs étapes en obtenant diverses variantes de l'algorithme. Une méthode simple dans notre cas consiste à faire un déroulement de la boucle de la ligne 4. En procédant de cette façon, nous obtenons l'Algorithme 11, que nous nommons UnrolledPow, qui se comporte exactement de la même façon que l'Algorithme 10 mais qui fait deux fois moins d'itérations. En pratique, une implantation de l'Algorithme 11 sera plus rapide que l'algorithme 10 car elle

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1  r ← 1
2  y ← x
3  k ← n
4  tant que k ≠ 0 faire
5  |  z ← y²
6  |  si k est impair ou ⌊k/2⌋ est impair alors
7  |  |  si k est impair alors
8  |  |  |  r ← r · y
9  |  |  si ⌊k/2⌋ est impair alors
10 |  |  |  r ← r · z
11 |  y ← z²
12 |  k ← ⌊k/4⌋
13 retourner r
```

Algorithme 12 : Algorithme de l'exponentiation rapide permettant de guider le prédicteur de branchement

fait moins d'opérations de contrôle qui sont liées à la boucle. Cependant, ce nouvel algorithme n'a rien changé à la complexité du nombre d'erreurs de prédiction excepté que maintenant nous avons deux instructions de branchement indépendantes aux lignes 5 et 8.

L'idée que nous avons eue pour modifier ce nouvel algorithme afin qu'il soit capable de guider le prédicteur a été de créer artificiellement de la dépendance entre ces deux instructions de branchement. Nous y parvenons en ajoutant un troisième test qui consiste simplement à vérifier qu'au moins un des deux autres tests est vrai. Si le résultat de ce test est vrai, alors nous faisons les deux autres tests comme précédemment. L'algorithme 12, que nous nommons GuidedPow, est ainsi obtenu en procédant de cette manière. Cet algorithme fait plus de comparaisons que les deux précédents, mais il est capable en contrepartie de guider le prédicteur. La Table 5.3 résume la différence entre ces trois algorithmes pour divers types d'opérations qu'elles effectuent. Le coût en multiplication est identique pour ces trois algorithmes, nous y parvenons grâce à l'ajout d'une variable supplémentaire.

Regardons maintenant ce qu'il se passe du point de vue probabiliste pour chacune de ces instructions de branchements. Gardons à l'esprit que nous sommes dans le cas uniforme où les instructions de branchements, si elles sont faites de manière indépendante comme dans l'Algorithme 11, ont une chance sur deux de produire chaque issue. Les tests des lignes 6,7 et 9 utilisent deux clauses : la première est de savoir si k est impair, la deuxième est de savoir si $\lfloor \frac{k}{2} \rfloor$ est impair. Nous avons donc quatre possibilités au total sur les valeurs de ces deux clauses qui ont chacune une chance sur quatre de se produire. Le résultat de la première instruction de branchement de la ligne 6 est faux uniquement si ces deux clauses sont fausses. Ainsi la probabilité que cette instruction soit prise est égale à $\frac{3}{4}$. Le résultat de l'instruction de branchement de la ligne 7 est vrai si la clause "k est impaire" est vraie, peu importe la valeur de l'autre clause, ce qui fait donc deux possibilités. Comme nous faisons ce test uniquement si

le résultat du test précédent est vrai, cela signifie qu'au moins une des deux clauses est vraies et il ne reste donc au total plus que trois possibilités. Nous avons donc une probabilité de $\frac{2}{3}$ que le résultat de l'instruction de la ligne 7 soit vrai. Le même raisonnement permet de montrer que le résultat de la ligne 9 est vrai avec probabilité $\frac{2}{3}$.

Remarque. Nous avons réussi, en rajoutant une connaissance sur les deux clauses apportées par l'instruction de la ligne 6, à créer plus facilement un déséquilibre sur toutes les instructions de branchements qui deviennent alors prédictibles. Remarquons cependant que cette instruction supplémentaire augmente la complexité en nombre de comparaisons comme nous pouvons le voir sur la table 5.3. Remarquons également que l'instruction que nous avons ajoutée est également parfaitement inutile au fonctionnement de l'Algorithme 12 qui se réduit au cas de l'Algorithme 11 une fois cette instruction retirée.

Version	Itérations	Mult./itér.	Branch./itér
Classique	$\log_2 n$	1.5	1
Avec déroulement	$\frac{\log_2 n}{2}$	3	2
Avec guidage de la prédiction	$\frac{\log_2 n}{2}$	3	2.5

TABLE 5.3 – Comparatifs des Algorithmes 10,11 et 12, sur le nombres d'itérations, de multiplication par itération, d'instructions de branchement en moyenne.

5.2.3 Résultats expérimentaux

Dans le but de comparer nos trois algorithmes d'exponentiation rapides, nous effectuons des expérimentations sur des implantations de ces algorithmes en C. Les résultats qui sont présentés dans cette section ont été obtenus sur la même machine que nous avons utilisée pour les mesures sur la recherche du minimum et du maximum. Pour rappel, le microprocesseur sur cette machine est un *Intel(R) Celeron(R) CPU 1037U @ 1.80GHz*. Nous avons effectué ces expérimentations sur d'autres machines avec d'autres processeurs et avons obtenu des différences de performances similaires. Les implantations de nos trois algorithmes sont données dans le Listing 11, disponible en Annexe B. Nous avons mesuré ces trois algorithmes sur plusieurs critères à savoir : le temps d'exécution, le nombre d'itérations, le nombre d'instructions de branchements, et le nombre d'erreurs de prédiction. Les résultats que nous présentons donnent une somme des mesures pour tous ces critères sur un ensemble de $5 \cdot 10^7$ exécutions de nos implantations. Une exécution correspond à choisir au hasard et uniformément un entier n dans l'intervalle $[1, 4^{13}[$, ensuite nous calculons pour toutes les implantations la valeur 2^n. Le choix de l'intervalle $[1, 4^{13}[$ permet d'avoir une chance sur deux qu'un bit de n soit 0 ou 1 sur les 26 premiers bits. Nous rendons donc ainsi les tests faits sur un seul bit sans connaissances supplémentaires imprédictibles. Les résultats obtenus sont donnés dans la table 5.4. Nous pouvons vérifier que le nombre d'itérations des implantations de l'exponentiation rapide est deux fois plus grand que celui des deux autres. Le nombre de multiplications est légèrement plus petit pour l'exponentiation classique, cette différence est due à un effet de bord qui provient du fait que n n'est pas for-

cément une puissance de 4. Venons en maintenant à ce qui nous intéresse le plus, à savoir les instructions de branchements et la prédiction de ces dernières. Nous remarquons que l'exponentiation classique fait le moins de comparaisons suivie de près par l'exponentiation avec juste le déroulement de la boucle principale. Cette différence est négligeable et s'explique encore à cause d'un effet de bord. Cependant, la différence entre l'exponentiation classique et l'exponentiation qui guide le prédicteur est bien plus importante. L'exponentiation qui guide le prédicteur compte environ 27.5% d'instructions de branchements en plus que l'exponentiation rapide. Cette différence était prévisible, nous pouvions estimer cette dernière à 25% en utilisant les données de la Table 5.3. Le résultat obtenu est donc très proche de ce que nous avons estimé. Le nombre d'erreurs de prédiction de branchement est sensiblement identique pour l'exponentiation classique et la version avec déroulement de la boucle principale. Cependant, comme nous l'avons espéré, l'implantation de l'algorithme qui guide la prédiction fait environ 16.7% d'erreurs en moins que les deux autres implantations. Ce gain en prédictions se montre efficace puisque nous observons un gain en temps d'exécution d'environ 30% contre l'exponentiation rapide classique et d'environ 14.7% contre l'exponentiation rapide avec déroulement de la boucle principale. Ce résultat laisse penser qu'il existe des compromis à chercher entre le nombre de comparaisons et le nombre d'erreurs de prédiction de branchement dans le cas où nous calculons des puissances de nombres réelles.

exponentiation	temps (en sec.)	itér. $\times 10^9$	mult. $\times 10^9$	ins. branch. $\times 10^9$	err. pred. $\times 10^9$
classique	13.859	1.250	1.900	1.300	0.674
avec déroulement	12.229	0.633	1.917	1.317	0.683
avec guide	10.659	0.633	1.917	1.658	0.577

TABLE 5.4 – Mesure de performances, réalisée à l'aide de PAPI, des implantations en C des trois versions de l'exponentiation rapide sur un total de 5.10^7 exécutions. Les mesures prises sont le temps d'exécution total sur ces exécutions, le nombre d'itérations, le nombre de multiplications, le nombre d'instructions de branchements, et le nombre d'erreurs de prédiction.

5.2.4 Analyse dans le cas moyen du nombre d'erreurs de prédiction de GUIDEDPOW

Nous allons maintenant nous intéresser à l'analyse du nombre d'erreurs de prédiction générées pour une même entrée par chacun de ces algorithmes. Les nombres d'erreurs produites par l'algorithme classique et l'algorithme avec déroulement de boucle sont des cas simples de par l'impossibilité de prédire l'issue des instructions de prédictions quel que soit le prédicteur choisit. Nous allons donc principalement nous intéresser au cas de GUIDEDPOW. Pour l'analyse de ces algorithmes nous allons considérer que l'exposant n est choisi au hasard dans un intervalle $\{0, \dots N\}$ pour un N entier strictement positif. Pour la suite, nous présenterons l'analyse du nombre d'erreurs de prédiction si le prédicteur utilisé fait partie des prédicteurs locaux qui ont été présentés dans la section 3.5 du chapitre 3.

Remarque. Les instructions de branchements permettant de tester la divisibilité par 2 et par 4 des algorithmes 10,11, et 12 reviennent à tester la valeur des deux derniers bits de la variable k, ce qui correspond à ce que nous avons fait dans

nos implantations en C. Si l'on décompose sous sa forme binaire $n = \sum_{i \geq 0} n_i 2^i$ alors les algorithmes vont tester dans l'ordre chacun de ces bits en commençant par celui de poids faible. Les algorithmes 11 et 12 testent ces bits deux par deux ce qui peut les amener à tester un bit supplémentaire en plus du bit de poids fort. En particulier pour ces algorithmes, nous regardons à l'itération j les bits $n_{2(j-1)}$ et n_{2j-1}.

Le cas idéal

Pour commencer l'analyse, nous considérons que n est choisi uniformément sur $\{0 \dots N\}$, avec $N = 4^k - 1$ pour un $k \geq 1$. Ce modèle est similaire à celui qui consiste à choisir chacun des $2k$ bits de la représentation binaire de n uniformément et indépendamment par une loi de Bernoulli de paramètre $\frac{1}{2}$. Cette indépendance des bits de n rend le cas simple à analyser et nous le considérons donc comme un cas idéal. Soit $L_k(n)$ le nombre d'itérations de la boucle principale de GuidedPow. Il s'agit d'une variable aléatoire qui est simple à analyser puisqu'elle est égale au plus petit entier l tel que 4^l est plus grand que n. En particulier, son espérance est donnée par :

$$\mathbb{E}[L_k] = k - \frac{1}{3} + o(1) \sim k.$$

Cette espérance peut facilement être obtenue en sommant la probabilité $\mathbb{P}(L_k > l) = 1 - 4^{l-k}$ car $L_k(n) > l$ implique $n \in \{4^l + 1 \dots 4^k\}$.

En appliquant les outils mathématiques vu dans la Section 3.5 du Chapitre 3, nous obtenons le résultat donné dans le Théorème 10.

Théorème 10. *Supposons que n est choisi aléatoirement selon une distribution uniforme dans* $\{0, N-1\}$. *L'espérance du nombre d'instructions de branchements effectuées par* ClassicalPow *et* UnrolledPow *est asymptotiquement équivalent à* $\log_2 N$, *tandis qu'il est asymptotiquement équivalent à* $\frac{5}{4}\log_2 N$ *pour* GuidedPow. *L'espérance du nombre d'erreurs de prédiction est asymptotiquement équivalent à* $\frac{1}{2}\log_2 N$ *pour* ClassicalPow *et* UnrolledPow, *pour tous les types de prédicteurs. Pour* GuidedPow, *il est asymptotiquement équivalent à* $\alpha \log_2 N$, *avec* $\alpha = \frac{1}{2}\mu(\frac{3}{4}) + \frac{3}{4}\mu(\frac{2}{3})$, *où* μ *est la probabilité de faire une erreur de prédiction sous la distribution stationnaire de la chaîne de Markov associée au prédicteur local (pour plus de détails , voir la section 3.5.7 du chapitre 3).*

Démonstration. La preuve est faite pour $N = 4^k$. Nous verrons plus loin comment gérer le cas général. Il est simple de montrer, comme nous l'avons fait précédemment avec UnrolledPow, que le nombre d'itérations en moyenne de ClassicalPow est asymptotiquement équivalent à $\log_2 N + \mathcal{O}(1)$. Comme pour chaque itération, un test est effectué avec une probabilité $\frac{1}{2}$ de causer une erreur de prédiction, le résultat annoncé est ainsi obtenu pour cet algorithme. Pour UnrolledPow et GuidedPow, nous savons déjà que l'espérance du nombre d'itérations est asymptotiquement équivalente à $\log_4 N = \frac{1}{2}\log_2 N$. UnrolledPow fait pour chaque itération exactement deux tests qui ont chacun une probabilité $\frac{1}{2}$ de causer une erreur de prédiction, nous amenant au résultat. GuidedPow va quant à lui pour chaque itération exécuter le premier test, si il est vrai avec probabilité $\frac{3}{4}$, alors deux tests supplémentaires sont effectués. Ceci nous amène donc au premier résultat annoncé qui est que le nombre de tests effectués en moyenne est asymptotiquement équivalent à $\frac{5}{4}\log_2 N$. Pour

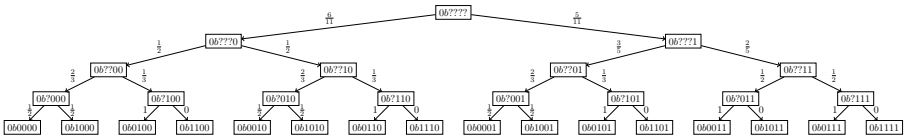


FIGURE 5.4 – Arbre de décomposition du choix d'un entier n dans l'intervalle $\{0 \dots 10\}$.

le deuxième résultat, nous utilisons le Théorème 4 ergodique et le fait que $\mathbb{E}[L_k] \sim k \sim \log_4 N$. Le premier test cause un nombre d'erreurs de prédiction en moyenne qui est asymptotiquement équivalent à $\mu(\frac{3}{4}) \log_4 N$ et chacun des deux tests internes causent un nombre d'erreurs de prédiction en moyenne qui est asymptotiquement équivalent à $\frac{3}{4}\mu(\frac{2}{3}) \log_4 N$. En combinant ces résultats, le nombre d'erreurs de prédiction en moyenne est asymptotiquement équivalent à $(\mu(\frac{3}{4}) + \frac{3}{2}\mu(\frac{2}{3})) \log_4 N$, concluant ainsi la preuve. □

En utilisant le Théorème 10 ainsi que les Equations (3.3),(3.4),(3.5) et (3.6), nous obtenons $\alpha = \frac{25}{48} \approx 0.52$, $\frac{9}{20} \approx 0.45$, $\frac{2045}{4368} \approx 0.47$ et $\frac{1095}{2788} \approx 0.39$ pour respectivement le 1-bit, le 2-bits saturé, le 2-bits avec saut et le 3-bits saturé. Ces valeurs doivent être comparées avec la constante multiplicative $\frac{1}{2}$ des deux autres algorithmes. En particulier pour le prédicteur 1-bit, l'espérance du nombre d'erreurs de prédiction est plus importante pour GUIDEDPOW que pour les deux autres. Ce prédicteur n'est donc pas suffisamment efficace pour compenser les erreurs de prédiction causées par la condition supplémentaire. Pour le compteur 3-bits saturé, GuidedPow fait approximativement $0.25 \log_2 n$ comparaisons de plus que UNROLLEDPOW, mais en contrepartie fait approximativement $0.11 \log_2 n$ erreurs de prédiction de moins. Nous avons pu vérifier ces résultats à l'aide de la librairie de simulation en python que nous présentons à l'Annexe A en utilisant le script du Listing 7 qui met en évidence la constante α pour chacun de ces prédicteurs.

Cas général

Le Théorème 10 fonctionne pour toutes valeurs de N y compris celles qui ne sont pas des puissances de 4. Pour ces cas, l'analyse est plus compliquée puisque les bits ne sont pas fixés à 0 ou à 1 avec une probabilité de $\frac{1}{2}$ et ne sont pas indépendants les uns des autres. Nous allons maintenant nous intéresser au cas général où N n'est pas nécessairement de la forme $4^k - 1$. Nous allons démontrer que le cas idéal est une approximation du terme de premier ordre du cas général. Notre idée est de montrer que le choix de n dans l'intervalle $\{0 \dots N\}$ est quasiment similaire sur les bits de poids faibles à choisir un nombre dans l'intervalle $\{0 \dots 4^{\widehat{k}} - 1\}$ avec $\widehat{k}$ la plus petite valeur telle que $4^{\widehat{k}} - 1 \geq N$. Nous pouvons représenter la génération d'un nombre choisi pour ces deux intervalles uniformément sous la forme d'un arbre de probabilité. Pour chaque nœud de hauteur i nous associons l'expérience aléatoire qui consiste à choisir la valeur du i-ème bit de ce nombre sachant que les bits de poids faibles de ce nombre sont déjà fixés par le chemin parcouru pour atteindre ce nœud. Nous notons pour un tel nœud x, par $p_x(N)$ la probabilité que le prochain bit soit 1 après la réalisation de l'expérience aléatoire associée à ce nœud pour le choix d'un entier

dans l'intervalle $\{0 \dots N\}$. Similairement, nous définissons $\widehat{p}_x(N)$ la probabilité pour l'arbre du cas idéal. La figure 5.4 donne une représentation des arbres de probabilités associés au choix aléatoire uniforme d'un entier dans les intervalles $\{0 \dots 10\}$. Nous n'avons pas donné d'exemple d'arbre pour le cas idéal car il s'agit d'un cas trivial où l'on remplace toutes les probabilités de chaque arrête par $\frac{1}{2}$. Les feuilles de chaque arbre correspond au choix d'un entier n dans l'intervalle considéré. Pour chaque nœud, nous avons au plus deux successeurs, le successeur de gauche correspond au cas où le prochain bit vaut 0 et le successeur de droite correspond au cas où le prochain bit vaut 1. Nous avons de plus étiqueté chaque arrête par la probabilité de réalisation. Par la suite, nous noterons par $d(x)$ le successeur droit de x et par $g(x)$ son successeur gauche.

Remarque. Comme la loi $\widehat{p}_i(N)$ est un cas idéal provenant d'un intervalle entre 0 et une puissance de 2 exclue. Chaque nœud correspond à une expérience de Bernoulli de paramètre $\frac{1}{2}$. La loi de $p_x(N)$ n'est évidemment pas un cas aussi simple et dépends à la fois de x et de N. Nous allons voir avec le Lemme 9 une expression bien déterminée de sa mesure de probabilité.

Par la suite, nous notons n_x le nombre obtenu en lisant les bits du chemin partant de la racine jusqu'à un nœud x d'un de ces arbres de probabilités.

Lemme 9. *La probabilité $p_x(N)$ que le résultat de l'épreuve au nœud x, de hauteur i dans l'arbre, soit 1 sachant que le nombre obtenu doit être choisi uniformément dans l'intervalle $\{0 \dots N\}$ est donné par la relation*

$$\frac{\lfloor \frac{N-n_x-2^i}{2^{i+1}} \rfloor + 1}{\lfloor \frac{N-n_x}{2^i} \rfloor + 1}.$$

Démonstration. Par définition, nous savons que

$$n_x = \sum_{j=0}^{i-1} b_j 2^j,$$

pour des $b_j \in \mathbb{B}$. À la fin, lorsque le nombre n est choisi, comme nous connaissons déjà les i premiers bits de n, nous pouvons écrire

$$\begin{aligned} n &= \sum_{j \geq 0} b_j 2^j && (5.1)\\ &= \sum_{j=0}^{i-1} b_j 2^j + \sum_{j \geq i} b_j 2^j && (5.2)\\ &= n_x + 2^i \sum_{j \geq 0} b'_j 2^j && (5.3)\\ &= n_x + 2^i n_q, && (5.4) \end{aligned}$$

pour un certain n_q qui est le quotient de la division euclidienne entre n et 2^i. Pour un nœud y, nous notons par N_y le nombre de feuilles associées au sous-arbre dont la racine est y. Comme le choix de n est uniforme, la probabilité que

l'issue de l'expérience du nœud x soit 1 est donnée par l'identité

$$p_x(N) = \frac{N_{d(x)}}{N_x} \tag{5.5}$$

D'après la relation de l'Equation (5.4), nous pouvons facilement exprimer N_x comme le cardinal de l'ensemble des quotients n_q tels que $2^i n_q + n_x \leq N$. Cet ensemble est équivalent à chercher tous les n_q tels que $n_q \leq \frac{N-n_x}{2^i}$. Or comme n_q est entier, cet ensemble est simplement l'intervalle $\{0 \dots \lfloor \frac{N-n_x}{2^i} \rfloor\}$ dont le cardinal est

$$N_x = \left\lfloor \frac{N - n_x}{2^i} \right\rfloor + 1 \tag{5.6}$$

De façon similaire, nous obtenons $N_{d(x)}$ en remarquant que $n_{d(x)} = n_x + 2^i$. Ainsi, nous avons

$$N_{d(x)} = \left\lfloor \frac{N - n_x - 2^i}{2^{i+1}} \right\rfloor + 1 \tag{5.7}$$

En combinant les Equations (5.5),(5.6) et (5.7) nous obtenons le résultat. □

Nous allons procéder à un couplage entre ces deux lois. Pour ce faire, nous allons générer des valeurs de n et $\widehat{n}$ pour respectivement le cas général et le cas idéal à partir d'une suite à valeurs réelles $u = u_0 u_1 \dots u_{\widehat{k}} \in [0,1]^{\widehat{k}}$ dont chaque élément est choisi uniformément et indépendamment des autres. Nous associons à ce mot les chemins $\mathrm{C}(u)$ et $\widehat{\mathrm{C}}(u)$ respectivement aux arbres du cas général et du cas idéal. Ces chemins sont construits en lisant les éléments de u dans l'ordre. Nous allons maintenant décrire la construction pour l'arbre du cas général. Au début nous commençons par la racine r de l'arbre et nous lisons l'élément u_0, si $u_0 < p_x(N)$ alors le prochain nœud à parcourir est $d(x)$ sinon c'est $g(x)$. Notons x_ℓ le nœud atteint après la lecture de l'élément u_ℓ. Nous continuons à parcourir l'arbre en comparant $u_{\ell+1}$ à $p_{x_\ell}(N)$ pour déterminer $x_{\ell+1}$. Pour le cas idéal, il suffit de remplacer les probabilités de la forme $p_y(N)$ par $\widehat{p}_y(N)$. La feuille obtenue, comme elle est issue de nos arbres, est équivalente à un choix uniforme dans nos intervalles. Maintenant que nous savons construire nos chemins, ce que nous aimerions connaître est à quel point pour une même suite u les chemins $\mathrm{C}(u)$ et $\widehat{\mathrm{C}}(u)$ diffèrent. Avant de répondre à cette question, il est important de connaître la probabilité qu'il y ait une bifurcation à la lecture de la lettre u_ℓ sachant que les chemins étaient identiques jusqu'alors. Cette probabilité est donnée dans le Lemme 10

Lemme 10. *La probabilité qu'il y ait une bifurcation à l'étape ℓ entre les chemins $\mathrm{C}(u)$ et $\widehat{\mathrm{C}}(u)$ à la lecture de $u = u_0 u_1 \dots u_{\widehat{k}} \in [0,1]^{\widehat{k}}$ sachant que ces chemins étaient identiques pour les ℓ premières étapes admet une borne supérieure de la forme*

$$p_{bif}(\ell) \leq \frac{\alpha 2^\ell}{N - 2^\ell},$$

avec $\alpha > 0$ constant.

Démonstration. Pour quantifier cette probabilité, commençons premièrement par voir intuitivement ce que ça signifie de bifurquer à la lecture de la lettre u_ℓ. Pour cela nous avons tracé un segment dans la figure 5.5. Cette figure montre que, pour qu'il y ait une bifurcation il faut que u_ℓ soit compris dans l'intervalle

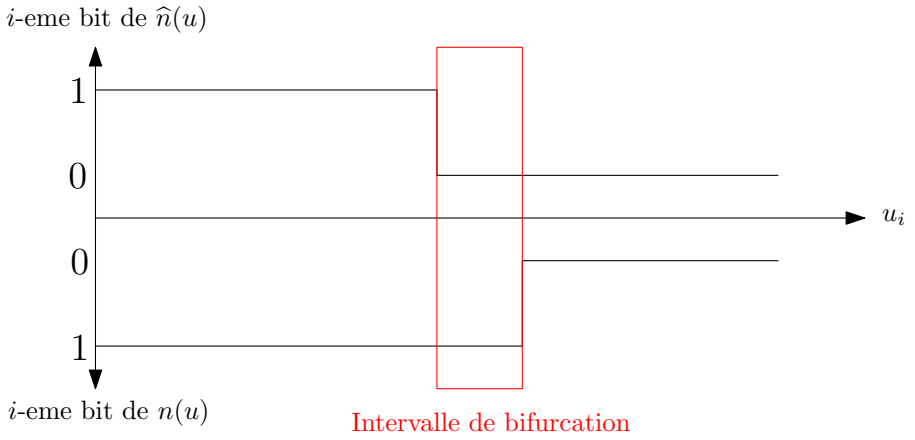


FIGURE 5.5 – Cette figure est une représentation du choix de la valeur du i-ème bit en fonction de la valeur de l'élément u_i de la suite à valeurs réelles u. la courbe du dessus représente le choix du bit du nombre $\widehat{n}(u)$ du cas idéal tandis que la courbe du dessous représente le choix du bit du nombre $n(u)$ du cas général. Les deux bits sont au début à 1 puis à partir de la valeur $\widehat{p}_x(N)$ la valeur du bit de $\widehat{n}(u)$ passe à 0 avec x qui est le noeud de l'arbre atteint à la lecture des i premiers éléments de u. Similairement, la valeur du bit de $n(u)$ passe à 0 à partir de $u_i > p_x(N)$. Entre ces deux valeurs se trouve un intervalle de bifurcation où le bit de $n(u)$ et de $\widehat{n}(u)$ sont différents.

$[\min(\widehat{p}_{x_\ell}(N), p_{x_\ell}(N)), \max(\widehat{p}_{x_\ell}(N), p_{x_\ell}(N))[$. Nous le nommerons par la suite *l'intervalle de bifurcation.*

Comme l'élément u_ℓ est choisie uniformément dans $[0,1]$, nous avons alors une identité simple pour quantifier cette probabilité qui est donnée par

$$p_{bif}(\ell) = |p_{x_\ell}(N) - \widehat{p}_{x_\ell}(N)| = |p_{x_\ell}(N) - \tfrac{1}{2}|. \tag{5.8}$$

Il ne nous reste plus qu'à déterminer un encadrement de la quantité $p_{x_\ell}(N)$ pour conclure. Remarquons premièrement par construction que $0 \leq n_{x_\ell} 2^\ell$. Rappelons que pour tout x nous avons également $x - 1 < \lfloor x \rfloor \leq x$. Avec ces deux encadrements, nous pouvons déterminer des bornes à $p_{x_\ell}(N)$. Commençons par trouver une borne inférieure,

$$\begin{aligned} p_{x_\ell}(N) &= \frac{\left\lfloor \frac{N - n_{x_\ell} - 2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N - n_{x_\ell}}{2^\ell} \right\rfloor + 1} \\ &\geq \frac{\left\lfloor \frac{N - 2^\ell - 2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N}{2^\ell} \right\rfloor + 1} \\ &\geq \frac{\frac{N}{2^{\ell+1}} - 1}{\frac{N}{2^\ell} + 1} \end{aligned}$$

Une dernière simplification de cette expression nous donne

$$p_{x_\ell}(N) \geq \frac{1}{2} - \frac{\frac{3}{2}}{\frac{N}{2^\ell}+1}. \tag{5.9}$$

Procédons de même pour trouver une borne supérieure,

$$\begin{aligned} p_{x_\ell}(N) &= \frac{\left\lfloor \frac{N-n_{x_\ell}-2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N-n_{x_\ell}}{2^\ell} \right\rfloor + 1} \\ &\leq \frac{\left\lfloor \frac{N-2^\ell}{2^{\ell+1}} \right\rfloor + 1}{\left\lfloor \frac{N-2^\ell}{2^\ell} \right\rfloor + 1} \\ &\leq \frac{\frac{N-2^\ell}{2^{\ell+1}} + 1}{\frac{N-2^\ell}{2^\ell} - 1 + 1} \end{aligned}$$

Après simplification, nous obtenons le majorant

$$p_{x_\ell}(N) \leq \frac{1}{2} + \frac{1}{\frac{N}{2^\ell}-1}. \tag{5.10}$$

En combinant les Equations (5.9) et (5.10), nous obtenons l'encadrement

$$-\frac{\frac{3}{2}}{\frac{N}{2^\ell}+1} \leq p_{x_\ell}(N) - \frac{1}{2} \leq \frac{1}{\frac{N}{2^\ell}-1}. \tag{5.11}$$

De ce dernier nous obtenons

$$|p_{x_\ell}(N) - \tfrac{1}{2}| \leq \frac{\frac{3}{2}}{\frac{N}{2^\ell}+1} + \frac{1}{\frac{N}{2^\ell}-1}, \tag{5.12}$$

nous permettant de conclure après simplification du terme de droite. □

Intéressons nous maintenant à la probabilité que les chemins $C(u)$ et $\widehat{C}(u)$ soient identiques après la lecture des m premières valeurs de la suite u, avec $m \in \mathbb{N}$. Nous notons cette probabilité par $P_{same}(m)$, par la définition des probabilités conditionnelles, nous avons la relation de récurrence suivante qui est vérifiée :

$$P_{same}(m) = (1 - p_{bif}(m))P_{same}(m-1),$$

pour tout $m > 0$. Nous pouvons ainsi directement montrer par récurrence que

$$P_{same}(m) = \prod_{i=0}^{m} 1 - p_{bif}(i). \tag{5.13}$$

Nous sommes maintenant prêt à démontrer que le terme dominant du cas idéal est une approximation du cas général. Nous commençons ainsi avec la Proposition 18.

Proposition 18. *Soit $u = (u_1, u_2, \ldots, u_{\widehat{k}})$ une suite de réels choisis uniformément et indépendamment dans $[0, 1]$. Soit $C(u)$ et $\widehat{C}(u)$ les chemins respectifs sur l'arbre du cas général et du cas idéal. La probabilité que les deux chemins soient identiques avant les $\log_2(N) - \sqrt{\log N}$ premières itérations est d'au moins $1 - o\left(\frac{1}{\log N}\right)$.*

Démonstration. D'après l'Equation (5.13), nous savons que la probabilité que les deux chemins soient identiques avant les $\delta_N = \log_2(N) - \lambda_N$ étapes est donnée par

$$P_{same}(\lfloor \delta_N \rfloor) = \prod_{i=0}^{\lfloor \delta_N \rfloor} 1 - p_{bif}(i),$$

pour un certain λ_N.

Nous savons de plus par le Lemme 10 que

$$p_{bif}(i) \leq \frac{\alpha 2^i}{N - 2^i},$$

avec $\alpha > 0$ constant.

En combinant ces deux lemmes, nous obtenons

$$P_{same}(\lfloor \delta_N \rfloor) \geq \prod_{i=0}^{\lfloor \delta_N \rfloor} 1 - \frac{\alpha 2^i}{N - 2^i}.$$

De plus pour tous les termes de ce produit nous avons l'encadrement trivial

$$1 \leq 2^i < 2^{\delta_N}.$$

En combinant tout ça, nous obtenons

$$P_{same}(\lfloor \delta_N \rfloor) > \left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)^{\delta_N + 1} \tag{5.14}$$

Nous allons utiliser plusieurs développements limités pour parvenir à nos fins. Commençons par ré-écrire le terme de droite de l'Equation (5.14),

$$\left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)^{\delta_N + 1} = \exp\left((\delta_N + 1)\left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)\right).$$

Quand N est grand, si

$$\lim_{N \to \infty} \frac{2^{\delta_N}}{N - 2^{\delta_N}} = 0,$$

alors nous avons

$$\exp\left((\delta_N + 1)\left(1 - \frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}\right)\right) = \exp\left(-(\delta_N + 1)\frac{\alpha 2^{\delta_N}}{N - 2^{\delta_N}}(1 + o(1))\right).$$

Si

$$\lim_{N \to \infty} \frac{\delta_N 2^{\delta_N}}{N - 2^{\delta_N}} = 0,$$

alors nous avons

$$\exp\left(-(\delta_N+1)\frac{\alpha 2^{\delta_N}}{N-2^{\delta_N}}(1+o(1))\right) = 1-(\delta_N+1)\frac{\alpha 2^{\delta_N}}{N-2^{\delta_N}}+o\left((\delta_N+1)\frac{\alpha 2^{\delta_N}}{N-2^{\delta_N}}\right).$$

Si nous choisissons λ_n de façon à avoir $\lim_{N\to\infty}\delta_N = \infty$ et $\frac{\delta_N 2^{\delta_N}}{N-2^{\delta_N}} = o\left(\frac{1}{\log N}\right)$, alors cette dernière égalité est équivalente à $1 - o\left(\frac{1}{\log N}\right)$.

Le choix de $\lambda_N = \sqrt{\log N}$ vérifie ces hypothèses et nous permet donc de conclure. □

Remarque. $\lambda_N = \sqrt{\log N}$ est aussi en $o(\log N)$ ce qui va nous être utile pour le théorème qui va suivre.

Théorème 11. *Le nombre d'erreurs de prédiction engendrées par l'Algorithme 12 lorsque l'exposant est choisi uniformément dans l'intervalle $\{0\dots N\}$ est asymptotiquement équivalent à $\alpha\log_2 N$, avec $\alpha = \frac{1}{2}\mu(\frac{3}{4}) + \frac{3}{4}\mu(\frac{2}{3})$, où μ est la probabilité de faire une erreur de prédiction sous la distribution stationnaire de la chaîne de Markov associée au prédicteur local (pour plus de détails , voir la section 3.5.7 du chapitre 3).*

Démonstration. Soit l'ensemble $\mathcal{G}_N$ des suites réelles $u \in [0,1]^{\widehat{k}}$ dont les chemins $C(u)$ et $\widehat{C}(u)$ sont identiques pour les $\log_2 N - \sqrt{\log N}$ premières étapes. Si une suite appartient à $\mathcal{G}_N$, alors le cas général diffère du cas idéal en nombre de prédictions sur les instructions de branchement qui lisent les $\sqrt{\log N}$ derniers bits de $n(u)$ et de $\widehat{n}(u)$. Notons $\mathcal{M}_N(u)$ et $\mathcal{I}_N(u)$ respectivement le nombre d'erreurs de prédiction de l'Algorithmes 12 lorsque l'exposant est $n(u)$ et lorsque l'exposant est $\widehat{n}(u)$. Il existe donc pour $u \in \mathcal{G}_N$ une constante $\beta > 0$ telle que

$$|\mathcal{M}_N(u) - \mathcal{I}_N(u)| < \beta\sqrt{\log N}.$$

Si $u \notin \mathcal{G}_N$, alors il existe une constante γ telle que

$$|\mathcal{M}_N(u) - \mathcal{I}_N(u)| < \gamma \log N.$$

Nous pouvons maintenant procéder de la façon suivante :

$$\begin{aligned}
|\mathbb{E}[\mathcal{M}_N(u)] - \mathbb{E}[\mathcal{I}_N(u)]| &\leq \mathbb{E}[|\mathcal{M}_N(u) - \mathcal{I}_N(u)|] \\
&< \beta\sqrt{\log N}\mathbb{P}(u \in \mathcal{G}_N) + \gamma \log N \mathbb{P}(u \notin \mathcal{G}_N) \\
&< \beta\sqrt{\log N}\cdot\left(1 - o\left(\frac{1}{\log N}\right)\right) + \gamma\log N \cdot o\left(\frac{1}{\log N}\right) \\
&< o(\log N)
\end{aligned}$$

Le cas idéal approxime le terme dominant du cas général et nous pouvons donc conclure qu'asymptotiquement le nombre d'erreurs de prédiction est le même que celui qui est donné dans le Théorème 10.

□

Nous venons de discuter du cas de l'exponentiation rapide et nous avons vu comment modifier l'algorithme classique pour l'aider à guider la prédiction de branchement.

Au cours de l'analyse du nombre d'erreurs de prédiction, à l'exécution de ces algorithmes, nous avons distinguer deux cas. Le premier, le cas idéal considère que chaque instruction de branchement d'intérêt a une probabilité fixée d'obtenir chacune de ses issues possibles. Nous avons vu que ce cas se produit lorsque l'exposant était une puissance de 4. Le deuxième, le cas général ne fait aucune considération sur la valeur de l'exposant. Ainsi, pour le cas général la probabilité n'est pas fixée comme précédemment, mais, elle n'est jamais vraiment très loin du cas idéal. Nous avons démontré cette proximité entre les deux cas avec ce dernier théorème.

De plus, nous avons obtenu divers résultats expérimentaux à l'aide de la librairie PAPI et de nos simulations. Ces résultats n'entrent pas en conflit avec notre analyse théorique.

Nous allons maintenant nous intéresser à un nouvel algorithme et nous verrons que les méthodes utilisées dans cette section pourront être en partie réutilisées.

5.3 Recherche dichotomique et variantes

Données : Une séquence T de taille n. Une valeur x dans T.
Résultat : la position de x dans T.

```
1 debut ← 1
2 fin ← n
3 tant que debut < fin faire
4 |   m ← (debut+fin)/2
5 |   si x > T[m] alors
6 |   |   debut ← m + 1
7 |   sinon
8 |   |   fin ← m
9 retourner fin
```

Algorithme 13 : Algorithme de recherche dichotomique classique

Le dernier algorithme que nous étudions dans ce chapitre est l'algorithme de recherche dichotomique. Cet algorithme résout le problème suivant : en entrée nous avons une séquence triée X et une valeur x, le but est d'obtenir en sortie la position de x dans X. Nous donnons une version en pseudo-code de la recherche dichotomique dans l'Algorithme 13. Il est possible que x ne soit pas dans X, dans ce cas nous retournons quand même un indice et il est donc nécessaire de faire un test supplémentaire pour vérifier que l'élément à cet indice est identique à x. L'algorithme est optimal en nombre de comparaisons et fonctionne selon un principe simple. Au début x peut se trouver à n'importe quelle position dans la séquence X. Il est cependant possible de réduire ce nombre de possibilités de moitié. Pour cela, il suffit de comparer x avec la valeur médiane de la séquence qui se situe au milieu de X car cette dernière est triée, notons cette valeur médiane x_m. Nous avons deux possibilités ; soit $x \leq x_m$ et par transitivité, x est située avant x_m dans la séquence X, soit $x > x_m$ et par transitivité, x est située après x_m dans la séquence X. Dans tous les cas, comme m est au milieu de X, nous éliminons la moitié des possibilités. Nous réitérons ainsi de suite sur la

sous-séquence de possibilités restantes jusqu'à ce qu'il ne reste plus qu'une seule possibilité. Comme à chaque itération nous divisons le nombre de possibilités par 2, si n est la taille de la séquence X, alors nous réduisons le nombre de possibilités à 1 en $\lceil \log_2 n \rceil$ étapes. Comme nous faisons une seule comparaison par itération, ce nombre d'étapes est également le nombre de comparaisons effectuées au total.

5.3.1 Identification des problèmes pour la prédiction de branchement et solutions possibles

Données : Une séquence T de taille n. Une valeur x dans T.
Résultat : la position de x dans T.

```
1 debut ← 1
2 fin ← n
3 tant que debut < fin faire
4 |  q1 ← (3debut+fin)/4
5 |  si x > T[q1] alors
6 |  |  debut ← q1 + 1
7 |  sinon
8 |  |  fin ← q1
9 retourner fin
```

Algorithme 14 : Algorithme du BiasedBinarySearch

Données : Une séquence T de taille n. Une valeur x dans T.
Résultat : la position de x dans T.

```
1  debut ← 0
2  fin ← n
3  tant que debut < fin faire
4  |  q1 ← (3debut+fin)/4
5  |  si x > T[q1] alors
6  |  |  m ← (debut+fin)/2
7  |  |  si x > T[m] alors
8  |  |  |  debut ← m + 1
9  |  |  sinon
10 |  |  |  debut ← q1 + 1
11 |  |  |  fin ← m
12 |  sinon
13 |  |  fin ← q1
14 retourner fin
```

Algorithme 15 : Algorithme du SkewSearch

Avec l'algorithme de recherche dichotomique, nous sommes une fois de plus confrontés à un problème de prédiction de branchement. Supposons que nous choisissons uniformément un élément x dans X comme paramètre de la recherche dichotomique. La comparaison effectuée à la ligne 5 de l'Algorithme 13

branche sur l'une des issues possibles de manière équiprobable. Une fois de plus cette instruction permet d'obtenir l'optimalité du nombre de comparaisons mais pose problème au prédicteur car elle est imprédictible. Nous allons donc, comme pour l'exponentiation rapide, chercher un moyen de guider le prédicteur. En conséquence, nous ne ferons plus un nombre de comparaisons optimal mais nous espérons trouver un compromis qui nous permettra de gagner en performance de temps d'exécution en pratique. Cette fois, nous ne pouvons pas faire un déroulement de boucle comme pour l'exponentiation rapide. Dans le cas de l'exponentiation rapide, nous pouvions effectuer les instructions de branchements de manière complètement indépendantes et nous avons créé une dépendance. Dans le cas de la recherche dichotomique, les instructions de branchements après déroulement de la boucle présentent déjà une forte dépendance entre elles. Il est cependant plus facile de transformer l'instruction de la ligne 5 pour rendre cette dernière prédictible. Notre première idée a donc été de remplacer la comparaison avec la médiane par le premier quartile. De cette manière, la probabilité que x soit plus petit que ce premier quartile est proche de $\frac{1}{4}$ [2], ce qui rend le branchement prédictible. Nous donnons dans l'Algorithme 14 une version pseudo-code de cet algorithme que nous nommerons par la suite BiasedBinarySearch. Cette méthode crée cependant un gros déséquilibre ce qui peut augmenter drastiquement le nombre de comparaisons. Nous avons pensé à une autre variante qui crée moins de déséquilibre tout en guidant la prédiction de branchement. Cette solution est un peu entre la solution de déroulement de boucle vue dans la section précédente pour l'exponentiation rapide, et la solution provenant de BiasedBinarySearch. Nous avons introduit un découpage dans BiasedBinarySearch qui est $(\frac{1}{4}, \frac{3}{4})$, c'est-à-dire que si le test de la ligne 5 de l'Algorithme 14 est vrai alors x est positionné dans le premier quart des éléments possibles à l'étape en cours, si le test est faux alors x est positionné dans les trois derniers quarts d'éléments possibles. Ce dernier découpage en environ $\frac{3}{4}$ peut engendrer un plus grand nombre de comparaisons. Notre dernière modification va consister à trouver un découpage plus intéressant. Le découpage que nous proposons est de la forme $\frac{1}{4}, \frac{1}{4}, \frac{1}{2}$. Pour obtenir ce découpage, nous pouvons modifier BiasedBinarySearch en rajoutant une comparaison supplémentaire avec la médiane de X dans le cas où x est plus grand que le premier quartile. En procédant de cette manière nous obtenons alors l'Algorithme 15 que nous nommons SkewSearch.

Remarque. Ce dernier algorithme est un "semi-déroulement" de boucle malin de la recherche dichotomique classique. En effet, il s'agit ici de procéder en un déroulement de la boucle uniquement pour le cas où x est plus petit que la médiane. Enfin, remarquons que le test avec la médiane n'est plus imprédictible dans ce cas, car nous savons alors que x est plus grand que le premier quartile, et intuitivement nous avons donc plus de chance d'être plus grand que la médiane. Le premier test a une probabilité d'environ $\frac{1}{4}$ d'être vrai, tandis que le deuxième test a une probabilité d'environ $\frac{1}{3}$ de l'être. Il pourrait être tentant d'essayer de faire des découpages de types $(\frac{1}{3}, \frac{2}{3})$ ou encore $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, nous avons cependant éviter de le faire pour des raisons de coût de la division qui sont plus important pour des diviseurs qui ne sont pas des puissances de 2 ce qui est le cas de 3.

2. En vérité, nous avons, en premier, choisi de regarder le premier tercile. Le problème de ce choix est qu'il nous faut faire une division par 3 ce qui se révèle être trop couteux. Pour cette raison, nous avons choisi de remplacer ce choix par le quartile qui demande des division par 4, qui est une puissance de 2 et qui se fait rapidement par décalage dans les registres.

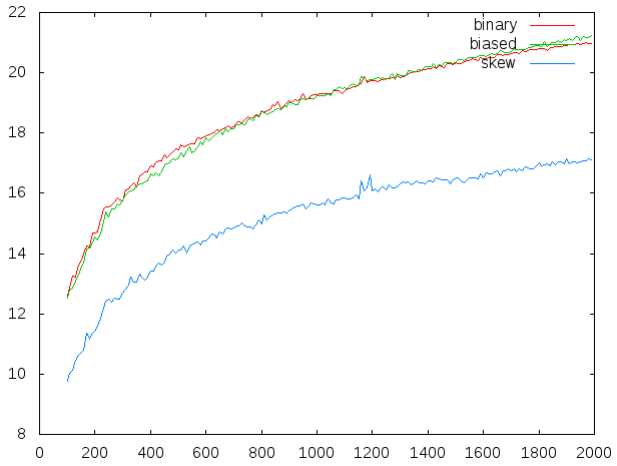


FIGURE 5.6 – Courbes de performances de nos trois versions de la recherche dichotomique. La taille des séquences d'entrées est petite et rentre entièrement dans le cache.

5.3.2 Expérimentations

Nous avons une fois de plus souhaité vérifier les performances de nos algorithmes modifiés. Nous présentons dans cette section deux expériences que nous avons réalisées. La première expérience a été faite sur des implantations en C de nos trois algorithmes de recherche dichotomique. Cette première expérience met en évidence que nos modifications peuvent présenter un intérêt pour certains microprocesseurs. Nous montrons une fois de plus les résultats pour le même microprocesseur à savoir un *Intel(R) Celeron(R) CPU 1037U @ 1.80GHz*, mais nous avons pu vérifier des résultats similaires sur d'autres machines. Nous donnons dans le Listing 12, disponible en Annexe B, les implantations de nos trois algorithmes qui ont été utilisées pour réaliser nos expériences. Dans le détail, nous avons testé chaque implantation pour les mêmes séquences sur différentes tailles. Chacune de ces séquences sont des tableaux triés de flottants simples choisis uniformément dans l'intervalle $[0, 1]$. Pour chaque séquence et pour chaque algorithme, nous avons effectué la recherche, dans le même ordre, de plusieurs éléments appartenant à ce même intervalle.

Remarque. Contrairement aux algorithmes étudiés précédemment, l'influence du cache est importante dans le cadre de la recherche dichotomique. Il nous a donc été nécessaire de prendre quelques précautions à ce sujet.

Par la suite les résultats que nous montrons proviennent du cas où ces implantations ont été compilées sans optimisation. Nous avons regroupé nos résultats sur plusieurs graphiques. Le premier graphique est donné dans la Figure 5.6 et concerne les cas où la séquence d'entrée est petite et peut tenir entièrement dans le cache. C'est un cas intéressant qui nous permet d'observer l'influence de la prédiction de branchement sur les performances de ces algorithmes sans que l'influence du cache ne vienne bruiter ces résultats. Pour ce cas nous observons que SKEWSEARCH a les meilleures performances tandis que les deux autres ont des performances relativement identiques. En particulier, nous observons un gain

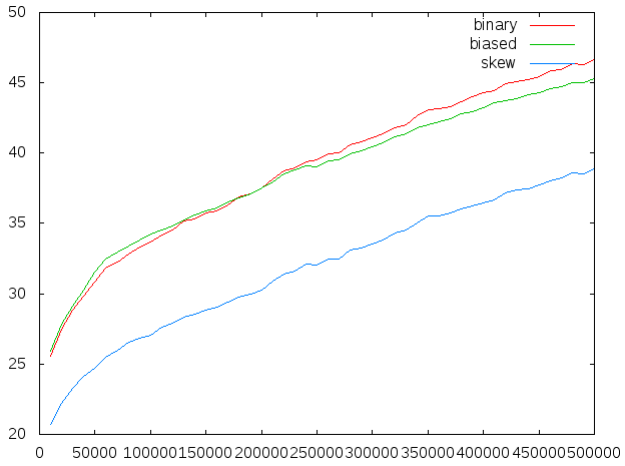


FIGURE 5.7 – Courbes de performances de nos trois versions de la recherche dichotomique. La taille des séquences d'entrées est de taille moyenne et rentre entièrement dans les caches de plus bas niveaux.

pour SKEWSEARCH en tout point supérieur de 22% à la recherche dichotomique classique sur ce graphe.

La Figure 5.7 donne le résultat obtenu pour des séquences de tailles moyennes. Les séquences de tailles moyennes sont influencées par le cache mais uniquement pour les niveaux les plus proches du microprocesseur. Nous observons des résultats similaires au cas où les tailles sont petites. Le gain est cependant moins important entre SKEWSEARCH et la recherche dichotomique classique qui est cette fois en tout point supérieur à 19%.

La Figure 5.8 donne le résultat obtenu pour des séquences de grandes tailles. Les séquences de grandes tailles sont lourdement influencées par les effets de cache puisque ces dernières n'entrent pas dans le cache L3. Nous observons alors qu'il semble exister une taille critique pour laquelle l'algorithme le plus performant dans ce cas est le BIASEDBINARYSEARCH avec des gains allant environ jusqu'à 30% par rapport à la recherche dichotomique classique. Pour ce dernier cas, BIASEDBINARYSEARCH reste un cas intéressant et une analyse de l'influence du cache pour comprendre la différence de performance pour ce cas avec SKEWSEARCH semble nécessaire.

La recherche dichotomique est un algorithme qui est implanté dans la librairie standard du langage C. Nous pourrions avoir envie de vérifier si nos modifications apportées à la fonction *bsearch* permettent d'obtenir des gains en temps d'exécutions. Le problème de cette fonction et qu'elle utilise le pattern *Template* en laissant le choix à l'utilisateur de la fonction de comparaison dont il donne l'adresse. La fonction ainsi donnée en paramètre ne peut pas être transformée en fonction *inline* et va ainsi s'exécuter lentement. De ce fait, le coût de la comparaison dans la fonction *bsearch* est très coûteuse et il est préférable de minimiser le nombre de comparaisons en préservant l'équilibre apporté par l'implantation standard. Cependant, il existe un ensemble de surcharges d'implantations de la recherche dichotomique en Java qui utilisent les fonctions de comparaisons standards pour les types primitifs qui sont peu coûteuses. Notre

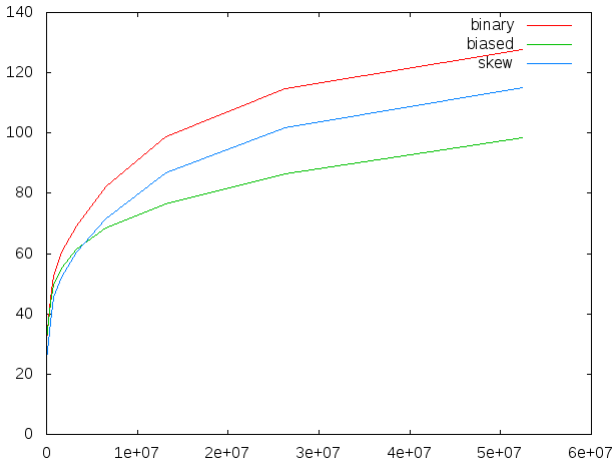


FIGURE 5.8 – Courbes de performances de nos trois versions de la recherche dichotomique. La taille des séquences d'entrées est de grande taille et ne rentre pas entièrement dans le cache de plus bas niveau.

deuxième expérimentation consiste donc à comparer une de ces fonctions de la librairie standard Java avec nos versions modifiées et qui guident la prédiction de branchement. Nous avons choisi de réaliser cette expérience lorsque la séquence est un tableau de *double* et l'élément à rechercher est également un *double*. Pour réaliser ces comparaisons de performances, nous avons utilisé la librarie *jmh* [1] qui permet de créer des benchmarks. Les codes sources utilisés pour réaliser ces benchmarks sont disponibles en téléchargement [6].

L'expérimentation a été effectuée sur un microprocesseur *Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz* dans un environnement *GNU/Linux*. Nous avons fait tourner ces benchmarks pour comparer les implantations sur des tableaux de petites et moyennes tailles. L'implantation de la recherche dichotomique classique est ici directement celle utilisée dans la librairie standard Java qui porte le nom *binarySearch()*. Nous retrouvons des résultats similaires à ceux que nous avons obtenus avec nos implantations en C. Ces résultats sont donnés sous forme de courbes de performances dans la Figure 5.9. Nous avons donc potentiellement, sur des machines avec des processeurs récents, une implantation d'une variante de la recherche dichotomique qui se montre plus efficace sur des tableaux de tailles moyennes que les fonctions de la librairie standard de Java. Nous avons testé ces fonctions uniquement sur le cas particulier que nous avons décrit ci-dessus. Il est donc trop tôt pour conclure et il faudrait très certainement faire tout une batterie de benchmarks, du même niveau de ceux qu'a probablement utilisés l'entreprise Oracle avant de choisir d'implanter TimSort dans leur librairie standard.

5.3.3 Analyse pour des prédicteurs locaux

Comme il a été vu dans la Section 5.2, nous cherchons à utiliser le théorème ergodique afin d'obtenir une bonne estimation asymptotique du nombre d'erreurs de prédiction. Nous devons donc commencer par calculer le nombre de fois

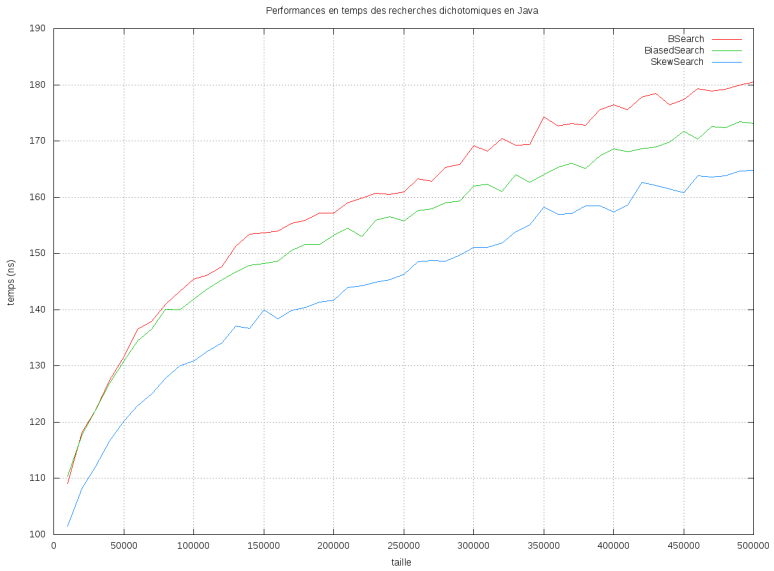


FIGURE 5.9 – Courbes de performances de temps mesurées sur les implantations Java des variantes de la recherche dichotomique.

que chaque condition est exécutée en moyenne, dans nos différents algorithmes. Nous considérons le modèle où toutes les sorties possibles sont équiprobables. Autrement dit, l'indice de l'élément à rechercher dans le tableau est choisi selon une loi uniforme dans $[n]$. Une estimation du terme dominant du nombre d'exécutions d'une instruction de branchement peut être obtenue en utilisant la version suivante du Théorème maître de Roura [39], qui a été simplifié pour nos besoins :

Théorème 12. *Soit une suite $(F_n)_{n\in\mathbb{N}}$ définie pour tout $n > 0$ par l'équation de récurrence*

$$F_n = Bn + \sum_{d=1}^{D} F_{S_{d,n}}, \tag{5.15}$$

avec $S_{d,n} = z_d \cdot n + \mathcal{O}(1)$ et D et B deux constantes positives. Si nous avons $z_d > 0$ et $\sum_{d=1}^{D} z_d = 1$, alors nous avons

$$F_n \sim -\frac{Bn \log n}{\sum_{d=1}^{D} z_d \log z_d}. \tag{5.16}$$

Démonstration. Nous sommes dans un cas particulier du théorème 2.3 de l'article de Roura [39]. Nous avons $w_d = 1$ et $r_{d,n} = 0$ pour tout $1 \geq d \geq D$ et vérifions donc la condition que $\sum_{d=1}^{D} |r_{d,n}| = \mathcal{O}(n^{-1})$. Nous avons $z_d > 0$ et $\sum_{d=1}^{D} z_d = 1$ et $s_{d,n} = \mathcal{O}(1)$ ce qui implique que $\frac{\sum_{d=1}^{D} |s_{d,n}|}{n} = \mathcal{O}(n^{-1})$. La fonction t_n est simplement égale à $n = 1 \times n^1 \times \log^0 n \times 1$. Nous sommes alors dans le cas (2.1) ce qui nous donne directement le résultat annoncé. □

Nombre d'itération en moyenne

Nous pouvons appliquer le théorème de Roura afin de déterminer le nombre de fois où chaque instruction de branchement est exécutée.

Proposition 19. *L'espérance du nombre d'exécution $I_c(n)$ de la ligne 5 de l'Algorithme 13 est asymptotiquement équivalent à $I_c(n) \sim \log_2 n$.*

Démonstration. Cette espérance $I_c(n)$ satisfait la relation

$$I_c(0) = 0;\ I_c(n) = 1 + \frac{\left\lfloor \frac{n}{2} \right\rfloor}{n} I_c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \frac{\left\lceil \frac{n}{2} \right\rceil}{n} I_c\left(\left\lceil \frac{n}{2} \right\rceil\right).$$

Le Théorème 12 est applicable en prenant $J_c(n) = nI_c(n)$ et nous donne directement le résultat. □

Proposition 20. *L'espérance du nombre d'exécution $I_b(n)$ de la ligne 5 de l'Algorithme 14 est asymptotiquement équivalent à $I_b(n) \sim \lambda \log n$, avec $\lambda = \frac{4}{4\log 4 - 3\log 3} \approx 1.78$.*

Démonstration. Cette espérance $I_b(n)$ satisfait la relation

$$I_b(0) = 0;\ I_b(n) = 1 + \frac{\left\lfloor \frac{n}{4} \right\rfloor}{n} I_b\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \frac{\left\lceil \frac{3n}{4} \right\rceil}{n} I_b\left(\left\lceil \frac{3n}{4} \right\rceil\right).$$

Le Théorème 12 est applicable pour $J_b(n) = nI_b(n)$ et nous donne directement le résultat. □

Proposition 21. *L'espérance du nombre d'exécutions $I_s(n)$ de la ligne 5 de l'Algorithme 15 est asymptotiquement équivalent à $I_s(n) \sim \frac{2}{3}\log_2 n$. L'espérance du nombre d'exécution $I'_s(n)$ de la ligne 7 est asymptotiquement équivalent à $I_s(n) \sim \frac{1}{2}\log_2 n$. La somme de ces deux termes nous donne un équivalent asymptotique de la totalité des comparaisons effectuées en moyenne par cet algorithme qui est approximativement égal à $\frac{7}{6}\log_2 n$.*

Démonstration. L'espérance du nombre d'itérations $I_s(n)$ de SkewSearch satisfait les relations

$$\begin{aligned}
&I_s(0) = 0;\ I_s(n) = 1 + \frac{\alpha_n}{n} I_s(\alpha_n) + \frac{\beta_n}{n} I_s(\beta_n) + \frac{\delta_n}{n} I_s(\delta_n)\\
&\alpha_n = \left\lfloor \frac{n}{4} \right\rfloor\\
&\beta_n = \left\lfloor \frac{n}{2} \right\rfloor - \left\lfloor \frac{n}{4} \right\rfloor\\
&\delta_n = \left\lceil \frac{n}{2} \right\rceil.
\end{aligned}$$

Pour ces mêmes α_n,β_n et δ_n, nous avons

$$I'_s(0) = 0;\ I'_s(n) = \frac{3}{4} + \frac{\alpha_n}{n} I'_s(\alpha_n) + \frac{\beta_n}{n} I'_s(\beta_n) + \frac{\delta_n}{n} I'_s(\delta_n)$$

Le Théorème 12 est applicable pour ces deux systèmes en posant $J_s(n) = nI_s(n)$ et $J'_s(n) = nI'_s(n)$ et nous donne directement le résultat. □

Remarque. Comme nous l'avons anticipé lors de la présentation des algorithmes BiasedBinarySearch et SkewSearch, ces derniers font chacun plus de comparaisons en moyenne que la recherche dichotomique classique. On remarque également que SkewSearch est bien celui qui génère un plus faible déséquilibre, ce dernier fait environ 5% de comparaisons en moins que BiasedBinarySearch.

Nous allons maintenant effectuer un raisonnement similaire à ce qui a été fait au cours de la Section 5.2. Dans un premier temps nous allons faire l'analyse de nos algorithmes en considérant un cas idéal, où la probabilité qu'une instruction de branchement soit vraie est donnée par une loi de Bernoulli de paramètre p pour chaque itération. Dans la pratique, il y a de légère fluctuation autour de cette valeur et nous montrerons qu'elles sont négligeables et que l'analyse dans le cas idéal est suffisant pour obtenir le comportement asymptotique de nos algorithmes. C'est ce que nous montrerons dans un deuxième temps.

Cas idéal

Le cas idéal correspond au cas où l'on considère qu'il n'y a pas de fluctuation de la probabilité que les différentes instructions de branchements soient vraies. Plus concrètement, nous allons considérer qu'à chaque itération la probabilité que l'instruction de branchement soit vraie à la ligne 5 de l'Algorithme 13 est $\frac{1}{2}$, celle de l'instruction de la ligne 5 de l'Algorithme 14 est $\frac{1}{4}$, celle de la ligne 5 de l'Algorithme 15 est $\frac{1}{4}$ et celle de la ligne 7 de ce même algorithme est $\frac{1}{3}$. Pour ce cas, la marche aléatoire du prédicteur est une chaîne de Markov qui converge rapidement vers sa seule distribution stationnaire et nous pouvons appliquer le Théorème 4 Ergodique du Chapitre 3.

Théorème 13. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}_{\mathcal{I}}[M_c(n)]$ du nombre d'erreurs de prédiction produites à l'exécution de l'Algorithme 13 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\mu\left(\frac{1}{2}\right)\log_2(n).$$

Démonstration. La Proposition 19 nous donne l'équivalent asymptotique de l'espérance du nombre $I_c(n)$ de fois que l'instruction de la ligne 5 de l'Algorithme 13 est exécutée. Comme nous considérons dans ce cas idéal que cette instruction est vraie avec probabilité $\frac{1}{2}$, à chaque itération à l'état stationnaire il y a une probabilité

$$\mu\left(\frac{1}{2}\right)$$

d'obtenir une erreur de prédiction. En appliquant le Théorème 4 ergodique nous obtenons

$$\mu\left(\frac{1}{2}\right)I_c(n),$$

ce qui correspond au résultat annoncé. □

Avec les mêmes raisonnements, nous pouvons trouver des théorèmes similaires pour BiasedBinarySearch et SkewSearch.

Théorème 14. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}_{\mathcal{I}}[M_b(n)]$ du nombre d'erreurs de prédiction produites à l'exécution de l'Algorithme 14 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\frac{4\mu\left(\frac{1}{4}\right)}{4\log 4 - 3\log 3}\log n$$

Démonstration. La preuve est similaire à celle du théorème précédent qui ne change uniquement que sur les constantes choisies. □

Théorème 15. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}_{\mathcal{I}}[M_s(n)]$ du nombre d'erreurs de prédiction produites à l'exécution de l'Algorithme 15 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\left(\frac{2}{3}\mu\left(\frac{1}{4}\right) + \frac{1}{2}\mu\left(\frac{1}{3}\right)\right)\log_2 n$$

Démonstration. La preuve est similaire aux preuves précédentes mais il faut faire attention car nous avons deux instructions de branchement. La Proposition 21 nous donne le nombre de fois que les instructions des lignes 5 et 7 sont chacune exécutées. Pour l'instruction de la ligne 5, nous avons quand n est grand de l'ordre de $\frac{2}{3}\log_2 n$ itérations effectuées. Pour cette ligne dans le cas idéal, nous avons dans l'état stationnaire une probabilité $\mu\left(\frac{1}{4}\right)$ d'avoir une erreur de prédiction. En appliquant le Théorème 4 nous avons que cette ligne génère en moyenne de l'ordre de

$$\frac{2}{3}\mu\left(\frac{1}{4}\right)\log_2 n$$

erreurs de prédiction. En procédant de la même façon pour la ligne 7, nous obtenons en moyenne de l'ordre de

$$\frac{1}{2}\mu\left(\frac{1}{3}\right)\log_2 n$$

erreurs de prédiction. C'est en faisant la somme de ces deux contributions que nous obtenons le résultat annoncé. □

Cas général

Dans le cas général, nous avons des fluctuations de la probabilité à chaque itération. Cependant pour les premières itérations nous sommes très proches du cas idéal. Nous pouvons une fois de plus appliquer une méthode de couplage similaire à ce que nous avons fait lors de la section précédente avec l'algorithme de l'exponentiation rapide.

Soit un *arbre de décomposition* τ associé à chacun des algorithmes de recherche, qui est définit comme suit. Si l'entrée est de taille n, sa racine est étiquetée par la paire $(0, n)$, et chaque nœud correspond aux valeurs possibles d et f (respectivement le début et la fin de l'intervalle) lors d'une itération de l'algorithme. Les feuilles sont les paires (i, i), pour $i \in [1, n]$ et correspondent

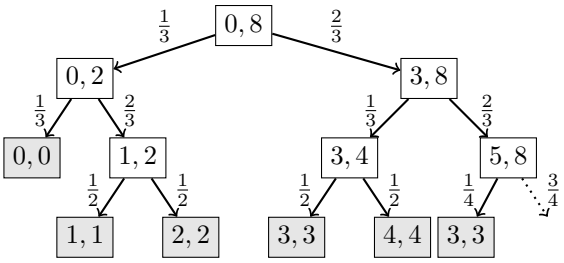


FIGURE 5.10 – Arbre de décomposition de BIASEDBINARYSEARCH pour $n = 8$.

aux différentes sorties possibles de l'algorithme. Les nœuds (d, f) et (d', f') sont adjacents si d et f peuvent être modifiés en d' et f' au cours de l'itération de la boucle. Une arrête entre ces deux nœuds est étiquetée par la probabilité $\frac{f'-d'+1}{f-d+1}$, qui est la probabilité que la modification ait lieu dans le modèle que nous considérons. Les enfants d'un nœud interne sont alors ordonnés par leur valeur d de la gauche vers la droite dans l'arbre. Un exemple d'arbre de décomposition pour BIASEDBINARYSEARCH peut être vu sur la Figure 5.10. Soit $u = (u_0, u_1, \dots)$, une suite infinie de valeurs de $[0, 1]$ choisies uniformément et indépendamment. Le chemin de la racine à une feuille dans τ associé à u est $\mathbf{Path}_n(\tau, u)$, où pour l'étape i, nous allons à gauche si u_i est plus petit que la probabilité de l'arrête allant vers le fils gauche et à droite dans le cas contraire. Soit $L_n(\tau, u)$ la longueur de $\mathbf{Path}_n(\tau, u)$. Soit également, $\mathbf{Path}_n(\mathcal{I}, u)$ le chemin suivi par la valeur u dans l'arbre idéal (et infini) $\mathcal{I}$ où l'on va à gauche avec probabilité $\frac{1}{4}$ et à droite avec probabilité $\frac{3}{4}$. Le résultat suivant est vérifié.

Lemme 11. *La probabilité que les chemins* $\mathbf{Path}_n(\tau, u)$ *et* $\mathbf{Path}_n(\mathcal{I}, u)$ *de la recherche dichotomique classique diffèrent sur l'une des* $L_n(\tau, u) - \sqrt{\log n}$ *premières étapes est* $o\left(\frac{1}{\log n}\right)$.

Démonstration. Soit (d, f) un nœud de τ qui n'est pas une feuille, et soit $t = f - d + 1$. La probabilité d'aller à gauche à la prochaine étape est donnée par $p(d, f) = \frac{m-d+1}{t}$, où $m = \left\lfloor \frac{d+f}{2} \right\rfloor$. Rappelons que pour tout $x \in \mathbb{R}$, nous avons $x-1 < \lfloor x \rfloor \leq x$. Avec cette inégalité, il est facile de montrer que $|p(d, f) - \frac{1}{2}| \leq \frac{\alpha}{t}$ pour une certaine constante $\alpha > 0$ et ainsi, la probabilité que les deux chemins diffèrent à cette étape est au plus $\frac{\alpha}{t}$. Soit $t_g = m - d + 1$ et $t_d = t - t_g = f - m$. Comme (d, f) n'est pas une feuille on a forcément $t \geq 2$. Il est alors facile de montrer que $\frac{1}{4} \leq \frac{t_g}{t}, \frac{t_d}{t} \leq \frac{3}{4}$. Ainsi pour t' la taille de l'intervalle du prochain nœud atteint à l'étape suivante, on a $t' = c \times t$ avec $c \in \left[\frac{1}{4}, \frac{3}{4}\right]$. Soit (d', f') le nœud atteint après $L_n(u) - \lambda_n$ étapes. Comme le dernier nœud atteint correspond à un intervalle de taille 1 et qu'il reste λ_n étapes. Nous avons en utilisant l'observation précédente,

$$t' = f' - d' + 1 \geq \left(\frac{4}{3}\right)^{\lambda_n}.$$

Par conséquent, tout nœud sur le chemin induit par u se trouvant à une distance $L_n(u) - \lambda_n$ sera donc plus grand que $\gamma_n = \left(\frac{4}{3}\right)^{\lambda_n}$. De plus pour n suffisam-

ment grand, on a $L_n(u) \leq \log_{4/3} n$. Ainsi la probabilité que les deux chemins $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ soient identiques pour les $L_n(u) - \lambda_n$ premières étapes est au moins $\left(1 - \frac{\alpha}{\gamma_n}\right)^{\log_{4/3} n - \lambda_n}$. Nous avons alors,

$$\begin{aligned}\left(1 - \frac{\alpha}{\gamma_n}\right)^{\log_{4/3} n - \lambda_n} &= \exp\left((\log_{4/3} n - \lambda_n) \log\left(1 - \frac{\alpha}{\gamma_n}\right)\right) \\ &= \exp\left(-\frac{\alpha}{\gamma_n}(\log_{4/3} n - \lambda_n)(1 + o(1))\right) \\ &= 1 - \frac{\alpha}{\gamma_n}(\log_{4/3} n - \lambda_n) + o\left(\frac{\alpha}{\gamma_n}(\log_{4/3} n - \lambda_n)\right).\end{aligned}$$

Si on choisit maintenant $\lambda_n = o(\log n)$ et $\lambda_n \to \infty$ quand n tend vers l'infini, on a

$$\begin{aligned}\left(1 - \frac{\alpha}{\gamma_n}\right)^{\log_{4/3} n - \lambda_n} &= 1 - \frac{\alpha}{\gamma_n} \log_{4/3} n + o\left(\frac{\log n}{\gamma_n}\right) \\ &= 1 - o\left(\frac{1}{\log n}\right),\end{aligned}$$

ce qui conclut la preuve puisque $\lambda_n = \sqrt{\log n}$ est un choix de fonction qui vérifie les conditions précédentes. □

Lemme 12. *La probabilité que les chemins* $\mathbf{Path}_n(\tau, u)$ *et* $\mathbf{Path}_n(\mathcal{I}, u)$ *de* BiasedBinarySearch *diffèrent sur l'une des* $L_n(\tau, u) - \sqrt{\log n}$ *premières étapes est* $o\left(\frac{1}{\log n}\right)$.

Démonstration. Soit (d, f) un nœud de τ qui n'est pas une feuille, et soit $t = f - d + 1$. La probabilité d'aller à gauche à la prochaine étape est donnée par $p(d, f) = \frac{m-d+1}{t}$, où $m = \left\lfloor \frac{3d+f}{4} \right\rfloor$. Rappelons que pour tout $x \in \mathbb{R}$, nous avons $x - 1 < \lfloor x \rfloor \leq x$. Avec cette inégalité, il est facile de montrer que $|p(d, f) - \frac{1}{4}| \leq \frac{\alpha}{t}$ pour une certaine constante $\alpha > 0$ et ainsi, la probabilité que les deux chemins diffèrent à cette étape est au plus $\frac{\alpha}{t}$. Soit $t_g = m - d + 1$ et $t_d = t - t_g = f - m$. Comme (d, f) n'est pas une feuille on a forcément $t \geq 2$. Il est alors facile de montrer que $\frac{1}{8} \leq \frac{t_g}{t}, \frac{t_d}{t} \leq \frac{7}{8}$. Ainsi pour t' la taille de l'intervalle du prochain nœud atteint à l'étape suivante, on a $t' = c \times t$ avec $c \in \left[\frac{1}{8}, \frac{7}{8}\right]$. Soit (d', f') le nœud atteint après $L_n(u) - \lambda_n$ étapes. Comme le dernier nœud atteint correspond à un intervalle de taille 1 et qu'il reste λ_n étapes. Nous avons en utilisant l'observation précédente,

$$t' = f' - d' + 1 \geq \left(\frac{8}{7}\right)^{\lambda_n}.$$

Par conséquent, tout nœud sur le chemin induit par u se trouvant à une distance $L_n(u) - \lambda_n$ sera donc plus grand que $\gamma_n = \left(\frac{8}{7}\right)^{\lambda_n}$. De plus pour n suffisamment grand, on a $L_n(u) \leq \log_{8/7} n$. Ainsi la probabilité que les deux chemins $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ soient identiques pour les $L_n(u) - \lambda_n$ premières

étapes est au moins $\left(1-\frac{\alpha}{\gamma_n}\right)^{\log_{8/7} n-\lambda_n}$. Nous avons alors,

$$\begin{aligned}\left(1-\frac{\alpha}{\gamma_n}\right)^{\log_{8/7} n-\lambda_n} &= \exp\left((\log_{8/7} n-\lambda_n)\log\left(1-\frac{\alpha}{\gamma_n}\right)\right)\\ &= \exp\left(-\frac{\alpha}{\gamma_n}(\log_{8/7} n-\lambda_n)(1+o(1))\right)\\ &= 1-\frac{\alpha}{\gamma_n}(\log_{8/7} n-\lambda_n)+o\left(\frac{\alpha}{\gamma_n}(\log_{8/7} n-\lambda_n)\right).\end{aligned}$$

Si on choisit maintenant $\lambda_n = o(\log n)$ et $\lambda_n \to \infty$ quand n tend vers l'infini, on a

$$\begin{aligned}\left(1-\frac{\alpha}{\gamma_n}\right)^{\log_{8/7} n-\lambda_n} &= 1-\frac{\alpha}{\gamma_n}\log_{8/7} n+o\left(\frac{\log n}{\gamma_n}\right)\\ &= 1-o\left(\frac{1}{\log n}\right),\end{aligned}$$

ce qui conclut la preuve puisque $\lambda_n = \sqrt{\log n}$ est un choix de fonction qui vérifie les conditions précédentes. □

Nous pouvons maintenant démontrer que l'algorithme de la recherche dichotomique classique et BiasedBinarySearch produisent un nombre d'erreurs de prédiction similaire au cas idéal.

Théorème 16. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}[M_b(n)]$ d'erreurs de prédiction produites à l'exécution de l'Algorithme 13 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\mu\left(\frac{1}{2}\right)\log_2(n).$$

Démonstration. Soit $M_n(u)$ le nombre d'erreurs de prédiction produites par l'algorithme en suivant le chemin induit par u, et soit $I_n(u)$ le nombre d'erreurs de prédiction produites par l'algorithme idéalisé en suivant les $L_n(u)$ premières étapes dans l'arbre idéal $\mathcal{I}$. Soit $\mathcal{G}_n$ l'ensemble de $u \in [0,1]^+$ tel que $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ sont égaux pour les premières $L_n(u) - \lambda_n$ étapes. Remarquons que si $u \in \mathcal{G}_n$, alors nous avons $|M_n(u) - I_n(u)| \leq \lambda_n$ comme ils ne peuvent différer que sur les λ_n dernières étapes. Si $u \notin \mathcal{G}_n$, alors nous avons forcément $|M_n(u) - I_n(u)| \leq L_n(u) \leq \beta \log n$ pour une constante β bien choisie. En utilisant le Lemme 11 et en choisissant $\lambda_n = \sqrt{\log n}$ nous avons

$$\begin{aligned}|\mathbb{E}[M_n] - \mathbb{E}[I_n]| &\leq \lambda_n \mathbb{P}(u \in \mathcal{G}_n) + \beta \log n \mathbb{P}(u \notin \mathcal{G}_n)\\ &= \lambda_n \cdot \left(1 - o\left(\frac{1}{\log n}\right)\right) + \beta \log n \cdot o\left(\frac{1}{\log n}\right)\\ &= o(\log n)\end{aligned}$$

Ainsi $\mathbb{E}[M_n] \sim \mathbb{E}[I_n]$, et le premier ordre de l'équivalent asymptotique du modèle idéalisé est identique pour le cas réel qui est donné dans le théorème 13. □

Le même raisonnement peut s'appliquer pour BiasedBinarySearch.

Théorème 17. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}[M_b(n)]$ d'erreurs de prédiction produites à l'exécution de l'Algorithme 14 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\frac{4\mu\left(\frac{1}{4}\right)}{4\log 4 - 3\log 3}\log n$$

Pour l'analyse de SkewSearch, nous devons faire attention à l'arité de l'arbre. Pour chaque itération nous avons 3 choix possibles et l'arbre est donc ternaire. Pour construire un chemin dans cet arbre à l'étape i, nous considérons les probabilités $p_g(i), p_m(i)$ et $p_d(i)$ d'aller respectivement à gauche, au milieu ou à droite. Pour une suite u, nous allons ensuite à gauche si $u_i < p_g(i)$, au milieu si $p_g(i) \leq u_i < p_g(i) + p_m(i)$ et à droite sinon.

Lemme 13. *La probabilité que les chemins $\mathbf{Path}_n(\tau, u)$ et $\mathbf{Path}_n(\mathcal{I}, u)$ de* BiasedBinarySearch *diffèrent sur l'une des $L_n(\tau, u) - \sqrt{\log n}$ premières étapes est $o\left(\frac{1}{\log n}\right)$.*

Démonstration. La preuve est similaire à celle des Lemmes 11 et 12. Nous devons regarder la probabilité qu'à l'étape i nous ayons une bifurcation. C'est le cas si u_i se trouve dans l'union d'intervalles où les choix du cas idéal et du cas général sont différents. Il s'agit donc de faire la somme d'inégalités similaires à celles déjà vu auparavant et il existe donc une constante $\alpha > 0$ telle que la probabilité que les chemins soient différents à l'étape i est plus petite que $\frac{\alpha}{t}$ avec t le nombre d'éléments restants à traiter de l'étape. De plus comme le nombre d'éléments restants est d'au moins deux, nous pouvons encore une fois trouver une constante $c \in]0, 1[$ telle que pour un nœud de hauteur plus grande que λ_n, nous avons que le nombre d'éléments restants à traiter pour ce nœud est au moins

$$\gamma_n = c^{-n}.$$

En procédant à une suite de développements limités similaires aux preuves des Lemmes 11 et 12, nous arrivons à la même conclusion en choisissant $\lambda_n = \sqrt{\log n}$. □

Nous pouvons directement, comme les fois précédentes, déduire le prochain théorème de ce dernier lemme.

Théorème 18. *Soit μ la probabilité d'avoir une erreur de prédiction à l'état stationnaire associée à un prédicteur local vu au cours du Chapitre 3. L'espérance $\mathbb{E}[M_s(n)]$ d'erreurs de prédiction produites à l'exécution de l'Algorithme 15 pour une entrée de taille n dans le cas idéal est asymptotiquement équivalent à*

$$\left(\frac{2}{3}\mu\left(\frac{1}{4}\right) + \frac{1}{2}\mu\left(\frac{1}{3}\right)\right)\log_2 n$$

5.3.4 Analyse pour le prédicteur global de SkewSearch

Cette section a pour but de donner des pistes concernant le comportement du prédicteur global comme évoqué dans le Chapitre 3 à la Section 3.5. Il s'agit donc

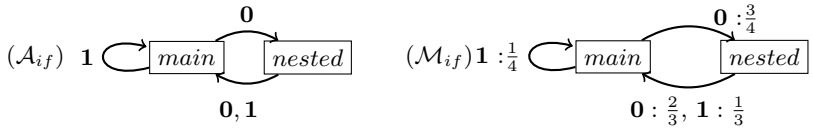


FIGURE 5.11 – À gauche, l'automate $\mathcal{A}_{if}$. À droite, son automate de Markov associée $\mathcal{M}_{if}$ avec les probabilités de transition $\mathbb{P}(1 \mid main) = \frac{1}{4}$, $\mathbb{P}(0 \mid main) = \frac{3}{4}$, $\mathbb{P}(0 \mid nested) = \frac{2}{3}$ and $\mathbb{P}(1 \mid nested) = \frac{1}{3}$.

d'un prédicteur avec une table d'histoires dont nous considérons qu'à chaque histoire est associée un prédicteur local 2-bits. Nous allons par la suite faire l'analyse du nombre d'erreurs de prédiction lors de l'exécution de SKEWSEARCH. Pour simplifier le problème, nous allons considérer que nous sommes dans le cas idéal que nous avons étudié à la section précédente. Comme en pratique les processeurs disposent également de prédicteurs de boucles, nous ne considéreront pas les branchements provenant de la boucle principale de l'Algorithme 15. Nous encodons le résultat d'une condition vraie par un 1 et dans le cas contraire par un 0. La trace de l'exécution de l'algorithme est donc donnée par un mot binaire $w = w_1 \dots w_N$. Par la suite, nous nommons par **main** l'instruction de branchement de la ligne 5 et par **nested** l'instruction de branchement de la ligne 7. Il est possible de garder une trace de l'instruction de branchement en cours d'exécution en utilisant un simple automate déterministe $\mathcal{A}_{if}$ constitué de deux états et qui est représenté dans la Figure 5.11 à gauche. Dans notre modèle, **main** est vraie avec probabilité $\frac{1}{4}$ et **nested** l'est avec probabilité $\frac{1}{3}$. Il est alors possible d'obtenir une chaîne de Markov associée à $\mathcal{A}_{if}$ en utilisant une méthode similaire à l'association d'une chaîne à un prédicteur (voir Figure 5.11 à droite). Un calcul direct permet de démontrer que le vecteur stationnaire π_{if} satisfait $\pi_{if}(\mathbf{main}) = \frac{4}{7}$ et $\pi_{if}(\mathbf{nested}) = \frac{3}{7}$. Nous considérons donc à partir de maintenant que la variable histoire h n'est modifiée que par les résultats des conditions **main** et **nested**. Nous supposerons pour la suite que la taille ℓ de cette variable est paire. Le mot 0^ℓ représente l'histoire consistant uniquement en des 0. Lorsqu'un test est effectué au temps t, le prédicteur utilise alors l'entrée dans sa table en position h_t pour faire une prédiction, avec h_t l'histoire au temps t.

Pour suivre l'évolution de l'algorithme au temps $t + 1$, nous avons donc seulement besoin de garder des traces de la table d'histoires T_t, l'histoire h_t et l'instruction de branchement exécutée IF_t. Cette dernière connaissance est nécessaire et suffisante pour calculer la probabilité d'avoir la sortie 0 ou 1 à la fin de l'instruction de branchement à l'étape t. À tout moment t, nous avons donc un ensemble d'états possibles pour chaque valeurs possibles de h_t, T_t et IF_t. Après l'exécution de la prochaine instruction de branchement, il y a eu une transition vers un nouvel état de ce même ensemble qui correspond au valeur prise par h_{t+1}, T_{t+1} et IF_{t+1}. Tout cela définit une chaîne de Markov $\mathcal{M}_{up}$ des modifications de la table d'histoires. À partir de $\mathcal{M}_{up}$, il est possible d'estimer théoriquement le nombre moyen d'erreurs de prédiction en utilisant le théorème ergodique, comme pour le cas des prédicteurs locaux. Le principal problème avec cette approche est que le calcul de π_{up} prend typiquement $\mathcal{O}(m^3)$ opérations, où m est le nombre d'états dans $\mathcal{M}_{up}$. Comme le nombre d'états est exponentiel en ℓ, le calcul est alors impraticable pour une longueur de l'histoire raisonnable

(comme par exemple $\ell > 6$) même en retirant les états non-atteignables. Par la suite, nous exploiterons donc la structure particulière de $\mathcal{M}_{up}$ pour calculer directement le nombre d'erreurs de prédiction.

Remarque. Dans la Figure 5.11, l'automate admet un mot synchronisant trivial qui est "1". Après la lecture de ce mot nous remarquons en effet que l'instruction de branchement qui suit est toujours **main**.

Définition 16. Nous définissons par $\mathcal{H}_{\mathbf{main}} \subset \mathbb{B}^\ell$ l'ensemble des histoires dont nous savons avec certitude que si $h_t = h$ alors $IF_t = \mathbf{main}$.

De façon similaire, nous définissons par $\mathcal{H}_{\mathbf{nested}} \subset \mathbb{B}^\ell$ l'ensemble des histoires dont nous savons avec certitude que si $h_t = h$ alors $IF_t = \mathbf{nested}$.

Exemple. *Pour $\ell = 8$, nous avons "00101011" qui appartient à l'ensemble $\mathcal{H}_{\mathbf{main}}$, tandis que, "10101010" appartient à l'ensemble $\mathcal{H}_{\mathbf{nested}}$.*

Remarque. D'après la remarque précédente, il est évident que le seul mot qui n'appartient à aucun de ces deux ensembles est le mot 0^ℓ. De plus, par construction, ces deux ensembles sont disjoints.

En conséquence de cette dernière remarque, chaque entrée $h \neq 0^\ell$ dans la table T se comporte comme un prédicteur local 2-bits saturé avec probabilité fixée à $\frac{1}{4}$ (respectivement $\frac{1}{3}$) pour une histoire associée à *main* (respectivement *nested*). C'est donc l'histoire $h = 0^\ell$ qui concentre les différences entre le prédicteur global et les prédicteurs locaux dans ce cas. Cet automate peut être transformé en une chaine de Markov, et le théorème Ergodique permet d'obtenir une estimation précise du nombre moyen d'erreurs de prédiction. C'est un cas particulier qui provient de l'existence de ce mot synchronisant. En suivant cette idée, nous obtenons le résultat du Théorème 19.

Le prédicteur global suit une marche aléatoire qui peut être représentée sous la forme d'une grande chaîne de Markov comme nous l'avons évoqué lors de l'introduction. Il est possible de représenter cette marche comme un automate dont chaque nœud consiste en un état qui retient l'histoire du prédicteur, les états de chaque prédicteur 2-bits local associé à chaque histoire, et l'instruction de branchement en cours d'exécution. Nous nommons cet automate $\mathcal{M}_{up}$. Nous avons donné une représentation partielle de cet automate dans la figure 5.12. Nous ne donnons pas une représentation exhaustive car cet automate contient beaucoup d'états (à savoir $2^{(\ell+1)+2^{\ell+1}}$).

Commençons par faire l'analyse du cas particulier quand $h = 0^\ell$.

Lemme 14. *Au cours de l'exécution de l'Algorithme 15, dans le cadre d'un prédicteur global avec table de prédicteurs 2-bits saturés, soit $h_t = 0^\ell$ avec ℓ un entier positif pair et un instant $t > 1$ suffisamment grand, la probabilité d'avoir une erreur de prédiction est donnée par*

$$\mu_0 = \frac{41}{119}.$$

Démonstration. Soit le processus dont chaque état consiste en une paire (s, i) avec s l'état de la table pour $h = 0^\ell$ et i l'instruction de branchement en cours d'exécution. Ce processus est une chaîne de Markov qui peut être représentée sous la forme d'un automate que nous nommons $\mathcal{M}_0$ que nous avons dessiné dans la Figure 5.13.

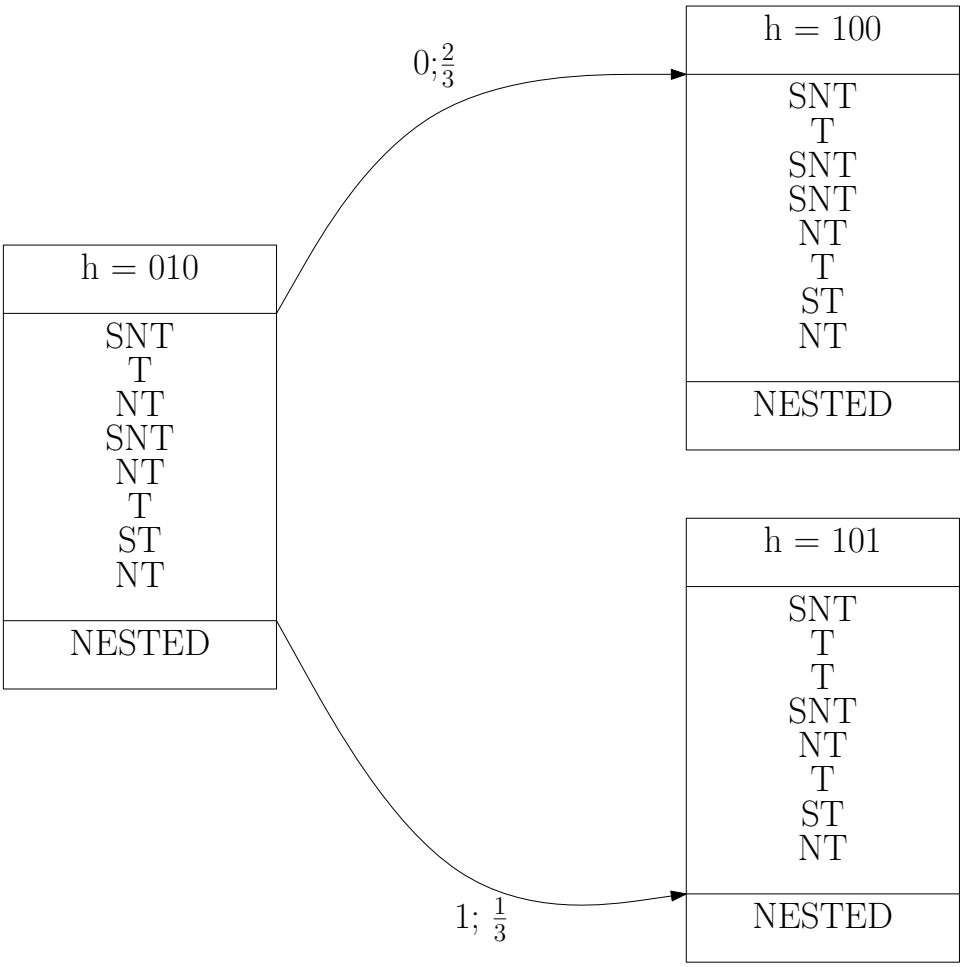


FIGURE 5.12 – Représentation partielle de la chaîne de Markov avec l'ensemble des états possibles à l'exécution de SKEWSEARCH

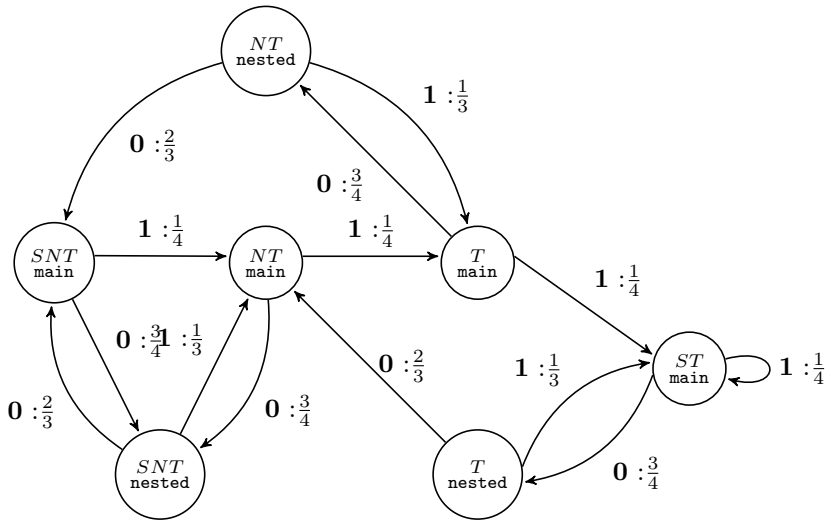


FIGURE 5.13 – La chaîne de Markov associée à la table 0^ℓ.

Supposons que nous sommes dans le cas où l'histoire au moment où nous exécutons la prochaine instruction de branchement est $h = 0^\ell$, nous avons alors deux cas possibles. Le premier cas est que le résultat de l'instruction de branchement est faux. Pour ce cas, la prochaine valeur de h est donnée une fois encore par $h' = 0^\ell$. D'après la Figure 5.11, si le prochain résultat d'une instruction est faux, alors la prochaine instruction qui sera exécutée n'est pas la même. Pour ce cas, si (s, i) représente l'état de l'automate $\mathcal{M}_0$ pour l'histoire h, alors l'état suivant est (s', i') avec $s' = \max(s - 1, 0)$ et $i' \neq i$. Le deuxième cas est que le résultat de l'instruction de branchement est vraie. Pour ce cas, la prochaine valeur de h est donnée par $h' = 0^{\ell-1}1$. Comme $h' \neq 0^\ell$, nous ne modifions pas lors de l'exécution de la prochaine instruction la table pour 0^ℓ. Cependant, nous pouvons obtenir après un certain nombre d'étapes l'histoire $h = 0^\ell$ à nouveau. Comme nous avons un 1 dans l'histoire à partir de h', il est alors nécessaire qu'à l'étape qui précède cette nouvelle occurrence, nous ayons une histoire de la forme $10^{\ell-1}$ et que le résultat de l'instruction à cette étape soit faux. Il se trouve que comme ℓ est pair, nous savons que $10^{\ell-1} \in \mathcal{H}_{nested}$, et donc comme le résultat est faux, à la prochain étape nous aurons à nouveau l'histoire de la forme 0^ℓ et l'instruction de branchement en cours est **main**. Ainsi pour ce cas, si l'état précédent est de la forme (s, i), le prochain état est de la forme $(s', \textbf{main})$ avec $s' = \min(3, s + 1)$. La probabilité des transitions correspond à la probabilité des transitions de l'automate à droite de la Figure 5.11. Il est possible de calculer la distribution stationnaire de cette chaîne de Markov. La conclusion est une conséquence directe de l'application du théorème ergodique pour cette distribution. □

Théorème 19. *Pour le prédicteur global avec table d'historiques et qui pour chaque histoire de longueur ℓ associe un prédicteur 2-bits saturé, le nombre moyen d'erreurs de prédiction obtenues lors de l'exécution de l'Algorithme 15*

pour une entrée de taille n est asymptotiquement équivalente à

$$\left(\frac{12}{35}+\frac{1}{595\cdot\sqrt{2}^{\ell}}\right)\frac{7}{6}\log_2 n.$$

Démonstration. Soit $w = w_0 w_1 \dots w_{N-1} \in \mathbb{B}^N$, le mot représentant les résultats, dans l'ordre de leurs exécutions, des instructions de branchements des lignes 5 et 7, où 1 signifie que le résultat est vrai et 0 qu'il est faux. Soit $h \neq 0^\ell$ une histoire. Soit les temps $1 \leq \tau_1 < \tau_2 < \cdots < \tau_m \leq N-1$ telles que pour tout t dans cet ensemble $h_t = h$. Il y a une occurrence de h qui se termine à la position t dans le mot w. Bien évidemment, les τ_i et m dépendent de N et de h, et sont des variables aléatoires pour des entrées aléatoires de l'algorithme. D'après le théorème ergodique, il existe une constante $\alpha_h > 0$ tel que $\mathbb{E}[m(h)] \sim \alpha_h N$, quand n tend vers l'infini. En effet, soit π_{up} le vecteur stationnaire de la partie fortement connexe de $\mathcal{M}_{up}$ définie plus haut, alors α_h est la somme de tous les $\pi_{up}(x)$ avec x qui fait partie de l'ensemble des états dont l'histoire est h. Observons également que le prédicteur $T_t(h)$ ne peut être modifié qu'aux temps τ_i. Autrement dit, $T_t(h)$ est constant pour $\tau_i + 1 \leq \tau < \tau_{i+1}$, en prenant pour convention $\tau_{m+1} = N$. Soit $\mathcal{H}_{main}$ et $\mathcal{H}_{nested}$ comme définis lors de la Définition 16. Pour tout $h \in \mathcal{H}_{main}$, les déplacements dans $T_t(h)$ pris aux temps $\tau_1 + 1, \tau_2 + 1, \dots, \tau_m + 1$ décrivent une marche aléatoire sur la chaîne de Markov associée au prédicteur 2-bits saturé de paramètre $\frac{1}{4}$. Ainsi le nombre moyen d'erreurs de prédiction provoquées par le prédicteur local $T[h]$ est asymptotiquement équivalent à $\mu\left(\frac{1}{4}\right)\alpha_h N$, d'après le Théorème 4. Respectivement pour tout $h \in \mathcal{H}_{nested}$, l'espérance du nombre d'erreurs de prédiction provoquées par $T[h]$ est asymptotiquement équivalent à $\mu\left(\frac{1}{3}\right)\alpha_h N$.

Il reste maintenant à analyser le comportement pour $h = 0^\ell$. C'est ce que nous avons déjà fait avec le résultat du Lemme 5.3.4 qui nous donne la probabilité d'erreur de prédiction espérée de $\mathcal{M}_0$ qui est $\mu_0 = \frac{41}{119}$. La probabilité d'avoir à un temps t l'histoire $h_t = 0^\ell$ est donnée par

$$\begin{aligned}
p_0 &= \mathbb{P}(h_t = 0^\ell) \\
&= \mathbb{P}(h_t = 0^\ell \textit{ et } IF_{t-\ell+1} = \textbf{main}) + \mathbb{P}(h_t = 0^\ell \textit{ et } IF_{t-\ell+1} = \textbf{nested}) \\
&= \mathbb{P}(IF_{t-\ell+1} = \textbf{main})\mathbb{P}(w_{t-\ell+1} = 0 \textit{ et } w_{t-\ell+2} = 0 \dots w_t = 0 \mid IF_{t-\ell+1} = \textbf{main}) \\
&\qquad + \mathbb{P}(IF_{t-\ell+1} = \textbf{nested})\mathbb{P}(w_{t-\ell+1} = 0 \textit{ et } w_{t-\ell+2} = 0 \dots w_t = 0 \mid IF_{t-\ell+1} = \textbf{nested}) \\
&= \mathbb{P}(IF_{t-\ell+1} = \textbf{main})\left(\frac{3}{4}\right)^{\frac{\ell}{2}}\left(\frac{2}{3}\right)^{\frac{\ell}{2}} + \mathbb{P}(IF_{t-\ell+1} = \textbf{nested})\left(\frac{3}{4}\right)^{\frac{\ell}{2}}\left(\frac{2}{3}\right)^{\frac{\ell}{2}} \\
&= \sqrt{2}^{-\ell}.
\end{aligned}$$

Nous posons par la suite C_n le nombre de comparaisons effectuées au total par l'Algorithme 15. D'après le théorème Ergodique, le nombre G_n d'erreurs de prédiction effectuées par le prédicteur global admet une espérance qui est asymptotiquement équivalent à

$$\mathbb{E}[G_n] \sim \left(\sum_{h\in\mathcal{H}_{main}} \alpha_h \mu_2\left(\frac{1}{4}\right) + \sum_{h\in\mathcal{H}_{nested}} \alpha_h \mu_2\left(\frac{1}{3}\right) + \frac{\mu_0}{2^\ell}\right)\mathbb{E}[C_n].$$

Cependant, comme la probabilité stationnaire d'être dans l'état *main* est donnée par $\pi_{up}(\mathbf{main}) = \frac{4}{7}$, nous avons

$$\sum_{h \in \mathcal{H}_{main}} \alpha_h = \pi_{up}(\mathbf{main})(1 - p_0) = \frac{4}{7}(1 - \sqrt{2}^{-\ell}).$$

De même comme $\pi_{up}(\mathbf{nested}) = \frac{3}{7}$,

$$\sum_{h \in \mathcal{H}_{nested}} \alpha_h = \pi_{up}(\mathbf{nested})(1 - p_0) = \frac{3}{7}(1 - \sqrt{2}^{-\ell}).$$

Comme $\mu_2\left(\frac{1}{4}\right) = \frac{3}{10}$ et $\mu_2\left(\frac{1}{3}\right) = \frac{2}{5}$, nous avons

$$\mathbb{E}[G_n] \sim \left(\frac{12}{35}(1 - \sqrt{2}^{-\ell}) + \frac{41}{119}\sqrt{2}^{-\ell}\right)\mathbb{E}[C_n] = \left(\frac{12}{35} + \frac{1}{595 \cdot \sqrt{2}^{\ell}}\right)\mathbb{E}[C_n].$$

Il nous suffit de remplacer la valeur de $\mathbb{E}[C_n]$ par $\frac{7}{6}\log_2 n$ que nous avons obtenu dans la Proposition 21 pour obtenir le résultat annoncé. □

D'après le Théorème 13, si on utilise un prédicteur local 2-bit pour chaque condition, le nombre d'erreurs de prédiction moyen est asymptotiquement équivalent à $\frac{12}{35}\frac{7}{6}\log_2 n$. La différence avec le prédicteur global est donc extrêmement faible, ce qui n'est pas une surprise puisque la seule différence entre les deux modèles pour le cas de l'algorithme 15 provient principalement de l'histoire $h = 0^{\ell}$ qui entremêle les deux instructions de branchement.

Dans cette section, nous avons proposé une modification de l'algorithme de la recherche dichotomique classique. Contrairement à l'exponentiation rapide, nous avons proposé un type différent de modifications à apporter pour que ce dernier soit capable de guider le prédicteur. Nous verrons, dans la section suivante, qu'il y a, malgré tout, quelques points communs entre les modifications apportées à ces deux algorithmes.

Pour finir, nous avons proposé une analyse des erreurs de prédiction produites par l'algorithme SkewSearch. Bien que, nous utilisons des propriétés de l'algorithme, à savoir l'existence d'un mot synchronisant, nous pensons qu'il s'agit du premier résultat théorique concernant ce type de prédicteur dans la littérature.

5.4 Généralisation à d'autres algorithmes

Dans la section qui suit, nous donnons des pistes dans le but de généraliser les modifications que nous avons apportées à nos algorithmes. Nous allons voir en effet qu'il existe des propriétés dans nos algorithmes que nous pouvons utiliser pour obtenir facilement des variantes de ces derniers. Ces variantes ont des chances de donner de bons résultats pour la prédiction comme nous avons pu le voir pour les cas que nous avons étudiés précédemment qui sont des cas particuliers de ces dernières

5.4.1 Algorithmes sous forme d'automates

Pour déterminer des propriétés intéressantes, nous allons définir la représentation d'un algorithme sous forme d'automate. Ce type de représentation est connue, elle apparaît dans la littérature, par exemple, sous la nomination de flot de contrôles.

Nous commençons par définir les ensembles ε et Σ qui représentent respectivement l'ensemble des entrées et des sorties de l'algorithme. Par exemple dans le cas de l'algorithme de l'exponentiation rapide, nous pouvons avoir $\varepsilon = \mathbb{R} \times \mathbb{N}$ et $\Sigma = \mathbb{R}$, puisque l'algorithme cherche à obtenir des valeurs de la forme x^n pour tout couple $(x, n) \in \mathbb{R} \times \mathbb{N}$. Soit l'ensemble $\mathcal{H}$, qui représente *l'environnement* dans lequel l'algorithme agit. Nous pouvons imaginer cet environnement comme étant l'ensemble des variables internes utilisées par l'algorithme pour fonctionner. Si nous reprenons notre exemple précédent, l'environnement sur lequel travaille l'exponentiation rapide est $\mathbb{R}^2 \times \mathbb{N}$ car à chaque instant nous retenons un triplet (r, y, k) tel que pour une entrée $(x, n) \in \varepsilon$, nous avons $r \times y^k = x^n$.

Définition 17. Une fonction *d'initialisation* $\mathcal{I} : \varepsilon \to \mathcal{H}$ transforme une entrée en environnement.

Dans le cas de l'exponentiation rapide, cette fonction est définie pour tout $(x, n) \in \varepsilon$ par $\mathcal{I}(x, n) = (1, x, n)$.

Définition 18. Une fonction de *sortie*, $\sigma : \mathcal{H} \to \Sigma$ transforme un environnement en une sortie.

Dans le cas de l'exponentiation rapide, cette fonction est définie pour tout $(r, y, k) \in \mathcal{H}$ par $\sigma(r, y, k) = r \times y^k$.

Définition 19. Une *question sur l'environnement* $q : \mathcal{H} \to \mathbb{B}$, est une fonction qui associe à tout état de l'environnement $h \in \mathcal{H}$ une réponse *oui* ou *non*. L'ensemble des questions sur l'environnement est noté par $\mathbb{B}^{\mathcal{H}}$.

Une question sur l'environnement qui est utilisée dans le cas de la boucle principale de l'exponentiation rapide est la fonction qui à tout $h = (r, y, k)$ associe $q(h) = oui$ si $k = 0$ et $q(h) = non$ sinon.

Définition 20. Une **action sur l'environnement** $a : \mathcal{H} \to \mathcal{H}$ est une fonction qui associe à un environnement $h \in \mathcal{H}$ un autre environnement $h' \in \mathcal{H}$ par l'action de a. L'ensemble des actions sur l'environnement est noté par $\mathcal{H}^{\mathcal{H}}$.

Une action qui est utilisée dans le cas de l'exponentiation rapide est l'action qui à tout $h = (r, y, k)$ associe $a(h) = (r \times y, y^2, \lfloor \frac{k}{2} \rfloor)$. Par la suite nous nommons une *fonction d'environnement*, une fonction qui est soit une question sur l'environnement soit une action sur l'environnement. L'ensemble des fonctions d'environnement est donc $\mathcal{H}^{\mathcal{H}} \cup \mathbb{B}^{\mathcal{H}}$.

Définition 21. Un automate $\mathcal{A}$ *à action sur* $\mathcal{H}$ est un septuple $\langle S, \mathbb{B} \cup \{\epsilon\}, \delta, i_0, F, h_0, \alpha \rangle$ avec :

- S l'ensemble des états de l'automate.
- $\delta : S \times (\mathbb{B} \cup \{\epsilon\}) \to S$ la fonction de transition.
- i_0 l'état initial.
- F l'ensemble des états finaux.

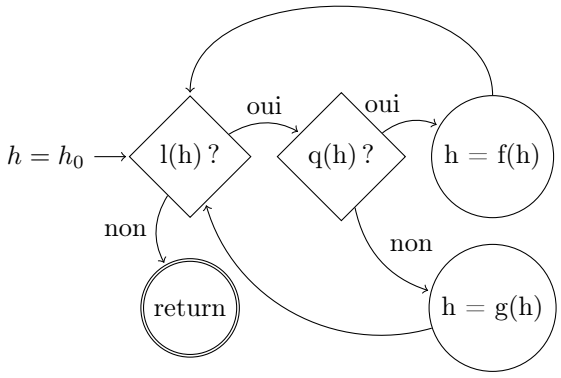


FIGURE 5.14 – Automate à action composé d'une boucle qui s'arrête quand $l(h)$ donne la réponse non, qui exécute $f(h)$ si $q(h)$ est oui et $f(h)$ sinon.

— $\alpha : S \to \mathcal{H}^{\mathcal{H}} \cup \mathbb{B}^{\mathcal{H}}$, la fonction d'environnement effectuée lorsque l'on atteint un des états de l'automate.

Remarque. La fonction α définit deux types possibles des états. Soit il s'agit d'une question qui appartient à $\mathbb{B}^{\mathcal{H}}$ et donc la seule chose que fait cet état c'est de poser la question en laissant l'environnement inchangé. Nous représentons dans nos figures ces états par une forme en losange. Soit il s'agit d'une action qui, à la fin de son application, change l'environnement. Nous représentons ces derniers états par une forme circulaire.

Définition 22. Nous appelons les états dont la fonction d'environnement est une question dans $\mathbb{B}^{\mathcal{H}}$ des *états de branchement*. Un état de branchement qui est au commencement d'un cycle est nommé un *état bouclant*.

Nous imposons des contraintes supplémentaires à ces automates pour la suite :

— Il est impossible d'avoir un cycle commençant par un état qui n'est pas de branchement.
— Si un état est de branchement, alors il n'appartient pas à F.
— Tous les états dans l'ensemble F sont des puits, c'est-à-dire qu'ils n'admettent pas de transition vers un autre état.

Une marche sur un automate d'action $\mathcal{A}$ initialisée par l'environnement $h_0 \in \mathcal{H}$ consiste en un chemin commençant par l'état initial i_0. Le chemin consiste à suivre les transitions définies par δ, ainsi lorsque nous atteignons un état $x \in S$ et que l'environnement est $h \in \mathcal{H}$, dont la fonction d'environnement $\alpha(x)$ est une question, l'état suivant est alors $\delta\left((\alpha(x))(h)\right)$. Soit $(x_0 = i_0, \ldots, x_m \in F)$ le chemin obtenu après cette marche. Nous pouvons extraire la sous-séquence de ce chemin dont les fonctions d'environnements sont des actions sur l'environnement sous la forme $(y_1, \ldots y_k)$, les valeurs prises par l'ensemble forment également une suite de la forme $(h_0, \ldots h_k)$ qui est construite par la récurrence valable à partir de $i > 0$, $h_i = (\alpha(x_i))(h_{i-1})$.

Théorème 20. *Pour une fonction d'initialisation $\mathcal{I} : \varepsilon \to \mathcal{H}$, un automate à action sur $\mathcal{H}$, et une fonction de sortie $\sigma : \mathcal{H} \to \Sigma$ il existe un algorithme A*

```
   Données : e ∈ ε
   Résultat : s ∈ Σ
1  h ← I(e)
2  tant que l(h) est oui faire
3  |  si q(h) est oui alors
4  |  |  h ← f(h)
5  |  sinon
6  |  |  h ← g(h)
7  retourner σ(h)
8  A(i_0, h)
```

Algorithme 16 : Algorithme équivalent à l'automate de la Figure 5.14.

qui pour toute entrée $e \in \varepsilon$ produit la même sortie $s = \sigma(h^*(e))$, avec $h^*(e)$ l'environnement final après une marche sur l'automate $\mathcal{A}$ initialisée par $h_0 = \mathcal{I}(e)$.

Démonstration. Notons $A(x, h)$, avec $x \in S$ et $h \in \mathcal{H}$, l'algorithme équivalent au sous-automate de $\mathcal{A}$ et dont l'état initial est x. L'algorithme A que nous cherchons à obtenir consiste alors à créer une variable h et à lui affecter la valeur $\mathcal{I}(e)$ et à appeler l'algorithme $A(i_0, h)$ qui va simuler la marche sur $\mathcal{A}$ avec l'environnement initialisé correctement. L'algorithme A est donc de la forme suivante :

```
   Données : e ∈ ε
   Résultat : s ∈ Σ
1  h ← I(e)
2  A(i_0, h)
```

Nous allons maintenant voir que $A(i_0, h)$ existe et que nous pouvons le construire. La construction se fait de manière récursive à l'aide de différentes règles. Essayons de construire $A(x, h)$ pour un état $x \in S$ et un environnement $h \in \mathcal{H}$ quelconque. Nous avons quatre cas possibles que nous allons maintenant détailler.

— Soit x n'est pas un état de branchement et $x \notin F$. Dans ce cas, la marche sur l'automate consiste à exécuter l'action $h = (\alpha(x))(h)$ et de continuer la marche à partir de $\delta(x, \epsilon)$. Cela revient donc à faire une affectation de h par $(\alpha(x))(h)$ et d'appeler l'algorithme $A(\delta(x, \epsilon), (\alpha(x))(h))$ ce qui revient à écrire l'algorithme suivant :

```
1  h ← (α(x))(h)
2  A(δ(x, ε), h)
```

— Soit x n'est pas un état de branchement et $x \in F$. Dans ce cas nous arrivons à la fin de la marche où nous exécutons l'action de x et nous obtenons alors que la sortie est donnée par $\sigma((\alpha(x))(h))$. Il est ainsi facile d'obtenir la même sortie en écrivant $A(x, h)$ sous la forme de l'algorithme suivant :

```
1  h ← (α(x))(h)
2  retourner σ(h)
```

— Soit x est un état de branchement non-bouclant. Si les deux branchements ne se rejoignent en aucun état, nous avons alors deux marches possibles, soit $(\alpha(x))(h) = oui$ et dans ce cas nous continuons la marche à partir de $\delta(x, oui)$, sinon nous continuons à partir de $\delta(x, non)$. Cela revient donc à tester $(\alpha(x))(h)$ et d'exécuter $A(\delta(x, oui), h)$ ou $A(\delta(x, non), h)$ selon le résultat. Nous obtenons alors l'algorithme $A(x, h)$ sous la forme suivante :

```
1 si (α(x))(h) est oui alors
2 |  A(δ(x, oui), h)
3 sinon
4 |  A(δ(x, non), h)
```

Notons $A_y(z, h)$ l'algorithme qui consiste à exécuter la marche sur l'automate $\mathcal{A}$ à partir de l'état z en s'arrêtant avant l'état y. Plus concrètement, $A_y(z, h)$ respecte les mêmes règles que $A(z, h)$ en ajoutant une règle supplémentaire qui consiste à omettre les lignes qui appellent $A(y, h')$ pour tout $h' \in \mathcal{H}$. Si les deux branchements se rejoignent en un état y, alors la marche consiste à exécuter les actions de la branche qui est prise. C'est ce qui est fait en exécutant $A_y(\delta(x, oui, h)$ ou $A_y(\delta(x, non, h)$ en fonction de la valeur de $(\alpha(x))(h)$. Nous obtenons alors dans ce cas l'algorithme $A(x, h)$ sous la forme suivante :

```
1 si (α(x))(h) est oui alors
2 |  A_y(δ(x, oui), h)
3 sinon
4 |  A_y(δ(x, non), h)
5 A(y, h)
```

— Soit x est un état bouclant. Nous allons avoir une boucle qui va répéter les actions à partir de l'état $\delta(x, oui)$ tant que $(\alpha(x))(h)$ a la valeur oui. L'algorithme $A(x, h)$ est de la forme suivante :

```
1 tant que (α(x))(h) est oui faire
2 |  A(δ(x, oui), h)
3 A(δ(x, non), h)
```

En appliquant toutes ces règles nous obtenons alors une écriture d'un algorithme équivalent à l'automate $\mathcal{A}$. □

Un algorithme qui agit sur $\mathcal{H}$ initialisé par l'état $h_0 \in \mathcal{H}$ est alors équivalent à un automate $\mathcal{A}$ à action sur $\mathcal{H}$ pour ce même h_0. Un exemple d'un tel automate est donné dans la Figure 5.14 qui est équivalent à l'Algorithme 16. En choisissant les fonctions $f, g \in \mathcal{H}^{\mathcal{H}}$ et des fonctions $l, q \in \mathbb{B}^{\mathcal{H}}$, il est possible d'obtenir l'algorithme d'exponentiation rapide ou de la recherche dichotomique. Ces deux algorithmes partagent donc un schéma commun. Pour le cas de l'exponentiation rapide, les fonctions à choisir sont définies de la façon suivante : Soit $h = (r, y, k)$,

— $l(h) :=$ "$k = 0$?"
— $q(h) :=$ "k *pair* ?"
— $f(h) = (r, y^2, \lfloor \frac{k}{2} \rfloor)$
— $g(h) = (r \times y, y^2, \lfloor \frac{k}{2} \rfloor)$

5.4.2 Automates à actions équivalentes

Nous allons désormais travailler principalement sur les automates. Notre but est d'apporter des transformations à nos automates qui permettent d'obtenir des résultats équivalents.

Définition 23. Deux automates initialisés respectivement par $\mathcal{I}_1 : \varepsilon \to \mathcal{H}_1$ et $\mathcal{I}_2 : \varepsilon \to \mathcal{H}_2$ sont *équivalents en actions* sur l'environnement si pour tout entrée $e \in \varepsilon$ on a $\sigma_1(h_1^*(e)) = \sigma_2(h_2^*(e))$, avec $\sigma_1 : \mathcal{H}_1 \to \Sigma$, $\sigma_2 : \mathcal{H}_2 \to \Sigma$ deux fonctions de sorties et $h_1^* : \varepsilon \to \mathcal{H}_1$, $h_2^* : \varepsilon \to \mathcal{H}_2$ les environnements atteints à l'état final pour toutes les entrées possibles. Nous avons donc deux automates qui travaillent sur des environnements pas nécessairement identiques mais qui sont initialisés par une même entrée et qui donne une même sortie à la fin de leurs marches.

Remarque. Comme nous allons étudier des transformations sur les automates, nous ne changeons pas l'environnement sur lequel travaille sa transformée, nous aurons donc $\mathcal{H}_1 = \mathcal{H}_2 = \mathcal{H}$. Nous utiliserons également les mêmes fonctions d'initialisation et de sortie. Nous verrons cependant des cas où $h_1^* \neq h_2^*$.

Changement des fonctions qui agissent sur l'environnement

Une première transformation qui peut être effectuée pour obtenir un automate équivalent en actions consiste à simplement changer les fonctions de α. La forme de l'automate reste donc inchangée, mais le comportement est différent. L'exemple où nous procédons ainsi est la recherche dichotomique où nous remplaçons par exemple la question q "est-ce que la valeur recherchée est avant la médiane ?" par la question q' "est-ce que la valeur recherchée est avant le premier quartile ?". Bien sûr avec cette nouvelle question il faut également changer en conséquence les actions des autres états de l'automate pour faire en sorte que les automates soient équivalents en actions.

Remarque. Cette méthode fonctionne bien si nous considérons des fonctions d'environnements dépendant d'un paramètre particulier. C'est un peu le cas avec la recherche dichotomique où le paramètre est la sélection du pivot. Si l'on note φ le nombre d'éléments plus petits que le pivot et ρ le nombre d'éléments plus grands que le pivot. Nous avons alors que la recherche dichotomique classique est paramétrée par $\frac{\varphi}{\varphi+\rho}$ que l'on souhaite le plus proche possible de $\frac{1}{2}$. Dans le cas de BiasedBinarySearch nous voulons que ce paramètre soit le plus proche possible de $\frac{1}{4}$.

Déroulement de boucle

La deuxième opération à appliquer concerne le déroulement de boucle. Il s'agit d'une opération qui peut être effectuée sur des automates comme celui de la Figure 5.14 et qui donne une fois transformé l'automate de la Figure 5.15.

Sous certaines conditions sur les fonctions associées aux actions de l'automate, il est possible d'obtenir un automate équivalent qui est celui de la Figure 5.16.

Lemme 15. *Soit pour toute entrée $e \in \varepsilon$ l'environnement $h^*(e)$ atteint à l'état final après une marche sur l'automate de la Figure 5.14 pour des fonctions l, q, f, g arbitraires. Si $q(h^*(e)) = oui$ et $l(f(h^*(e))) = non$ et $\sigma(f(h^*(e))) =$*

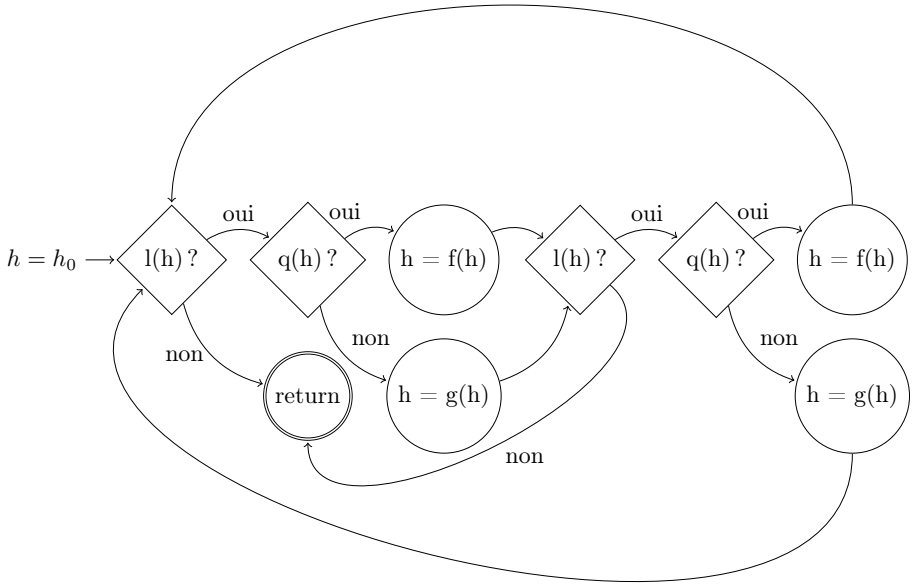


FIGURE 5.15 – déroulement de boucle simple

$\sigma(h^*(e))$ ou si $q(h^*(e)) = non$ et $l(g(h^*(e))) = non$ et $\sigma(g(h^*(e))) = \sigma(h^*(e))$, alors cet automate et l'automate de la Figure 5.16 sont équivalents en actions en gardant ces mêmes choix de fonctions.

Démonstration. Les modifications apportées à l'environnement par les marches sur ces deux automates sont identiques excepté à l'approche de la dernière étape pour une question de parité. Les différents environnements obtenus par l'action de l'automate de la Figure 5.14 forment une suite $(h_0 = \mathcal{I}(e), h_1, \ldots, h_m = h^*(e))$. Soit $m = 2k$ pair, et dans ce cas les environnements obtenus par l'action de l'automate de la Figure 5.16 forment la suite $(h_0, h_2, h_4, h_6, \ldots h_{2k} = h_m)$. Soit $m = 2k + 1$ impair, et dans ce cas les environnements obtenus par l'action de l'automate de la Figure 5.16 forment la suite $(h_0, h_2, \ldots, h_{2k}, h_{2k+2})$ avec $h_{2k+2} = f(h^*(e))$ si $q(h^*(e)) = oui$ et $l(f(h^*(e))) = non$ ou $h_{2k+2} = g(h^*(e))$ si $q(h^*(e)) = non$ et $l(g(h^*(e))) = non$. Si $q(h^*(e)) = oui$ il est donc suffisant que $\sigma(f(h^*(e))) = \sigma(h^*(e))$ pour que les deux automates donnent la même sortie. De même, si $q(h^*(e)) = non$ il est donc suffisant que $\sigma(g(h^*(e))) = \sigma(h^*(e))$. □

La condition nécessaire pour appliquer ce lemme est par exemple vérifiée dans le cas de l'exponentiation rapide. Rappelons que nos choix de fonctions sont les suivants :

— $l(h) :=$ "$k = 0$?"
— $q(h) :=$ "k *pair* ?"

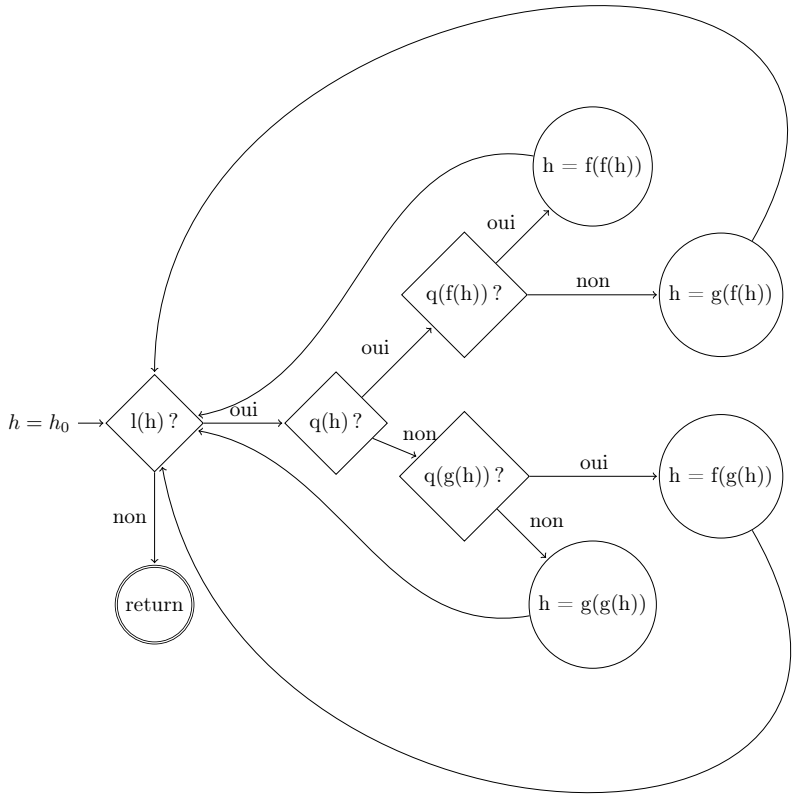


FIGURE 5.16 – déroulement de boucle complexe

— $f(h) = (r, y^2, \lfloor \frac{k}{2} \rfloor)$
— $g(h) = (r \times y, y^2, \lfloor \frac{k}{2} \rfloor)$

Pour l'automate de la Figure 5.14, pour une entrée $e = (x, n)$, l'environnement $h^*(e)$ est de la forme $(x^n, x^{2^i}, 0)$ avec $i \in \mathbb{N}$ le nombre d'itérations effectuées en tout. Si on applique $\sigma(h^*(e))$ nous obtenons bien x^n. Remarquons que nous avons $q(h^*(e)) = oui$ et que de plus $f(h) = (x^n, x^{2^{i+1}}, 0)$. Nous avons ainsi que $\sigma(f(h)) = x^n$ et $l(f(h)) = non$, la condition est donc satisfaite ce qui nous permet d'appliquer le lemme.

5.4.3 Ajout d'une redondance

Nous allons maintenant généraliser la transformation que nous avons appliquée pour obtenir l'exponentiation rapide qui guide la prédiction de branchement. L'idée est de rajouter une redondance qui est inutile au fonctionnement

de l'algorithme mais qui peut avoir pour effet d'améliorer la prédiction. Nous donnons le lemme suivant :

Lemme 16. *Soit pour toute entrée $e \in \varepsilon$ l'environnement $h^*(e)$ atteint à l'état final après une marche sur l'automate de la Figure 5.16 pour des fonctions l, q, f, g arbitraires. Si pour tout $h \in \mathcal{H}$, $q(f(h)) = q(g(h))$ alors il y a équivalence en action entre cet automate et l'automate de la Figure 5.17 en gardant ces mêmes choix de fonctions.*

Démonstration. La seule différence entre ces deux automates est l'ajout d'un état qui fait le test de savoir si $q(h)$ est oui ou si $q(f(h))$ est oui. Comme par hypothèse, $q(f(h)) = q(g(h))$ faire le test de $q(f(h))$ revient à faire le test de $q(g(h))$. Nous pouvons vérifier pour chaque cas que les deux automates vont changer l'environnement de la même façon à chaque itération. Si $q(h)$ est oui, alors $q(h)|q(f(h))$ est oui également. Dans ce cas les deux automates testent alors la valeur de $q(f(h))$ et font les mêmes modifications de la valeur de h. Si $q(h)$ est non, alors l'automate de la Figure 5.16 fait le test de la valeur de $q(g(h))$, si $q(g(h)) = oui$ alors h est modifiée par $f(g(h))$. Dans ce cas, $q(f(h)) = q(g(h)) = oui$ et donc l'automate de la Figure 5.17 fait la même action. Si $q(h) = non$ et $q(g(h)) = non$ alors h est modifiée par $h = g(g(h))$. Dans ce cas le test $q(h)|q(f(h))$ est non également et l'on voit que la même modification est apportée. □

En appliquant ce dernier lemme et le Théorème 20 au cas de l'exponentiation rapide, nous obtenons l'Algorithme 12. Il est facile de vérifier ici que $q(f(h)) = f(g(h))$ pour tout $h \in \mathcal{H}$. En effet si $h = (r, y, k)$, le test $q(h)$ se fait uniquement sur la variable k, et dans les deux cas après avoir appliqué $f(h)$ ou $g(h)$ nous avons que k est divisé par 2.

Dans la Figure 5.17, nous remarquons qu'il y a un état qui est inaccessible. En retirant cet état, nous obtenons l'automate de la Figure 5.18 qui est de ce fait équivalent en action au premier. Nous ne nous étions pas aperçus de ce cas lorsque nous avons fait l'étude de cet algorithme dans nos travaux. En appliquant le Théorème 20, nous obtenons ainsi l'Algorithme 17 qui réduit encore légèrement le nombre d'erreurs de prédiction par rapport à l'Algorithme 12 ainsi que le nombre de comparaisons et est donc plus efficace encore en pratique.

Dans cette section, nous donnons des pistes permettant de généraliser les modifications que nous avons faites pour modifier l'algorithme de l'exponentiation rapide et de la recherche dichotomique. Nous avons vu que ces algorithmes avaient des propriétés particulières que nous avons exploitées pour parvenir à nos fins.

Les travaux présentés ici ne sont encore qu'à un stade préliminaire, mais ils peuvent donner des méthodes pour permettre de modifier d'autres algorithmes, afin que ces derniers soient capables de guider la prédiction.

Dans l'avenir, il pourrait être intéressant de continuer à développer ces travaux afin d'exploiter d'autres propriétés des algorithmes. Nous pourrions, aussi, essayer de mettre en évidence les probabilités des branchements, afin de démontrer que ces modifications nous conduisent à des gains en ce qui concerne la prédiction de branchement.

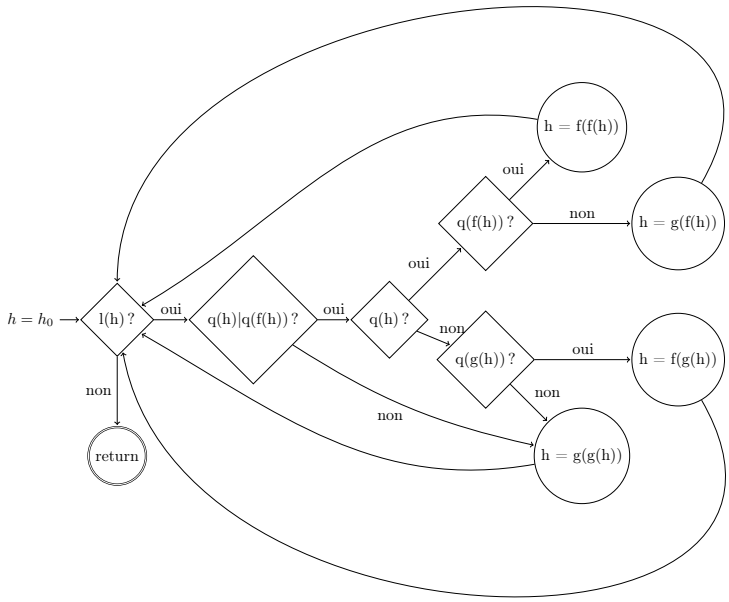


FIGURE 5.17 – Ajout d'une redondance dans l'automate Figure 5.16.

5.5 Conclusion

Dans ce chapitre, notre but était de modifier des algorithmes connus dans le but de guider la prédiction. Nous avions pour espoir que ces variantes, une fois implantées sur nos machines actuelles, aient de bonnes performances en temps d'exécution. Il nous faut souvent pour faire moins d'erreurs de prédiction faire plus d'opérations standards. Il y a donc des compromis que nous avons dû faire. Notre espoir est cependant justifié, nous avons, par exemple, montré que l'algorithme naïf de recherche simultanée du minimum et du maximum dans une séquence est plus efficace, en pratique, que l'algorithme optimisé qui fait pourtant 25% de comparaisons en moins. Ce premier résultat mettait en évidence que ces compromis étaient possibles.

Nous avons ensuite proposé nos propres modifications à deux algorithmes classiques, l'exponentiation rapide et la recherche dichotomique. Pour ces deux cas, il s'agit de deux algorithmes avec une structure similaire. Ces deux algorithmes consistent en une boucle au sein de laquelle nous avons une instruction de branchement. Comme cette instruction de branchement est exécutée plusieurs fois, il peut être intéressant de guider la prédiction. Pour y parvenir, dans les deux cas, nous avons fait apparaître de nouvelles instructions de branchement dans la boucle tout en faisant en sorte que le comportement de ces variantes reste similaire aux algorithmes d'origines. Nous avons, par exemple, effectué un déroulement de boucle pour l'exponentiation rapide. Nous avons ensuite fait en

Données : Un élément $x \in (M, \cdot)$, l'exposant $n \in (N)$.
Résultat : x^n

```
1  r ← 1
2  y ← x
3  k ← n
4  tant que k ≠ 0 faire
5      z ← y²
6      si k est impaire ou ⌊k/2⌋ est impaire alors
7          si k est impaire alors
8              r ← r · y
9              si ⌊k/2⌋ est impaire alors
10                 r ← r · z
11         sinon
12             r ← r · z
13     y ← z²
14     k ← ⌊k/4⌋
15 retourner r
```

Algorithme 17 : Algorithme de l'exponentiation rapide permettant de guider le prédicteur de branchement

sorte que ces instructions de branchement soient dépendantes entre-elles pour déséquilibrer les probabilités de l'issue choisie par ces dernières. C'est ce déséquilibre qui nous permet de guider la prédiction. En contrepartie, nous avions plus d'opérations à effectuer, dans le cas de l'exponentiation rapide nous avions environ 25% d'instructions de branchement supplémentaires à faire. Cependant, nous avons vu que ce surcout s'est avéré être compensé par la prédiction de branchement au point d'observer de meilleures performances en temps d'exécution de nos variantes par rapport aux algorithmes originaux.

Comme les structures de nos algorithmes sont similaires, nous avons donné des pistes pour essayer de déterminer des règles pour modifier des algorithmes avec la même structure. Nous sommes ainsi parvenus à extraire des propriétés abstraites que nous avons exploitées pour obtenir nos variantes. Nous pensons qu'il reste à ce jour de nombreux algorithmes qui peuvent être modifiés pour guider suffisamment bien la prédiction de branchement et obtenir de meilleures performances en pratique. Les pistes que nous avons données à la fin de ce chapitre pourraient par exemple permettre de détecter ces algorithmes et d'appliquer le même genre de modifications. Par la suite, il pourrait donc être intéressant de développer un peu plus cette généralisation et de les appliquer à des algorithmes encore plus complexes.

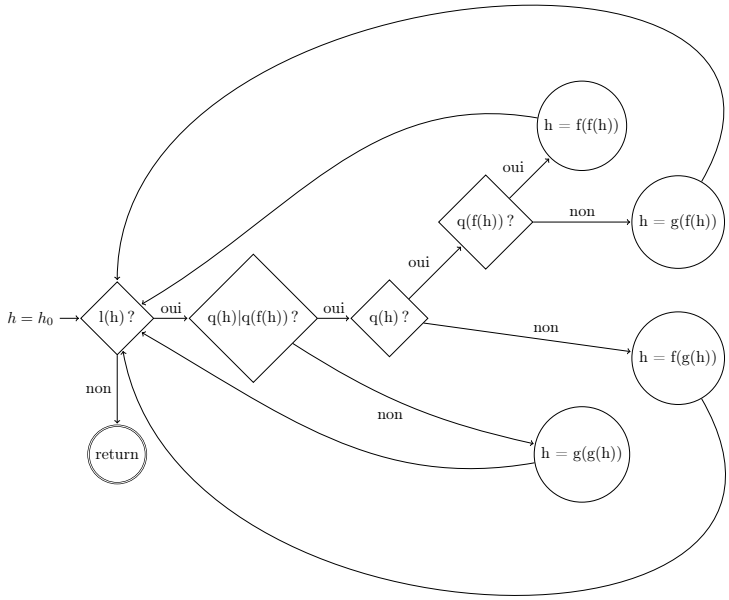


FIGURE 5.18 – Automate obtenu à partir de la Figure 5.17 en retirant l'état inaccessible.

Chapitre 6

Modèle d'Ewens

Sommaire

Souvent, lorsque nous faisons de l'analyse d'algorithmes dans le cas moyen nous considérons, par défaut, que l'entrée est distribuée selon une loi uniforme. En pratique, il n'est pas rare d'avoir une incertitude sur l'entrée et de considérer que cette dernière est bien aléatoire. Le choix de la distribution uniforme n'est cependant pas nécessairement un choix justifié si nous essayons de faire une analyse qui se veut proche de la réalité. Nous avons vu au cours du chapitre 4, qu'il existe des algorithmes de tri qui s'adaptent au désordre de la séquence d'entrée à trier. Ces algorithmes, comme c'est le cas de TimSort par exemple, se montrent efficaces en pratique. Cependant, si nous essayons de faire l'analyse en moyenne pour le cas uniforme de ces algorithmes, les résultats risquent de nous en apprendre peu sur ce qui se passe en pratique. Il est préférable de mettre plus de poids aux cas où le désordre est petit qui sont des cas favorables pour ces algorithmes. Pour le problème du tri, il y a des utilisations où ce genre de considérations ne sont pas déraisonnables. Par exemple, il est possible qu'un tri soit utilisé régulièrement pour trier certaines données qui ont subit de légères modifications par des ajouts de nouvelles clefs ou des altérations. Pour cet exemple, les données sont donc triées à la fin de chaque appel à ce tri et présentent donc peu de désordres. Nous avons vu qu'il existe des résultats qui mettent en évidence cette utilisation du désordre dans le pire cas, nous souhaiterions qu'il existe un équivalent pour le cas moyen.

L'analyse lisse qui consiste à prendre le pire cas et de le perturber par un bruit (souvent gaussien), est un exemple d'analyse en moyenne qui n'utilise pas une distribution uniforme. Cette méthode a, par exemple, été utilisée pour l'algorithme du simplex qui permet de résoudre des programmes linéaires, et à montrer que lorsque l'entrée est bruitée par un bruit gaussien, l'algorithme se comportait en temps polynomiale (voir [41]). Cette méthode donne une bonne idée de la raison pour laquelle bien que le pire cas du simplex soit exponentielle, ce dernier se comporte souvent bien mieux en pratique.

L'analyse lisse s'applique bien pour les cas où l'entrée appartient à un domaine continu. Or, nous nous intéressons pour notre part à l'ensemble des permutations de tailles n et dont le domaine est discret. Pour cette raison, nous n'allons pas dans ce chapitre appliquer une analyse lisse, mais nous allons nous intéresser à un type de distribution non-uniforme sur l'ensemble des permutations $\mathfrak{S}_n$. Le but étant de favoriser, si nous avons une mesure de désordre $m(X)$ comme définit lors du chapitre 4, les cas où $m(\pi)$ est petit pour $\pi \in \mathfrak{S}_n$. Plus concrètement, ce chapitre se base sur une distribution connue des probabilistes, à savoir la distribution d'Ewens. Cette distribution permet de favoriser l'apparition de permutations de $\mathfrak{S}_n$ en fonction du nombre de cycles de ces derniers. Ce chapitre donne en détail les résultats de nos travaux publiés pour la conférence AofA de 2016 [7]. Dans ces travaux nous avons généralisé la distribution d'Ewens à de nouvelles statistiques. En particulier, nous avons étudié le cas du modèle d'Ewens qui dépend du nombre de max-records qui est comme nous l'avons vu au cours du chapitre 4 associé à une mesure de désordre.

Données : X une séquence d'objets comparables
Résultat : X triée
1 Initialiser une séquence Y vide
2 **pour** $i \leftarrow 1$ **à** n **faire**
3 **si** x_i *est un record* **alors**
4 $Y \leftarrow Y \cdot \langle x_i \rangle$
5 $X \leftarrow X \ominus Y$
6 trier X à l'aide d'une fonction auxiliaire
7 $X \leftarrow X \oplus Y$

Algorithme 18 : Algorithme adaptatif pour m_{rec}.

6.1 Algorithme de tri optimal pour la mesure m_{rec}

Pour motiver le choix de ces distributions, nous allons nous intéresser à un algorithme de tri qui s'adapte au nombre de records. Pour rappel, si une séquence X a $rec(X)$ max-records, alors sa mesure est donnée par $m_{rec}(X) = |X| - rec(X)$. Nous pouvons également démontrer qu'il existe une borne inférieure pour les algorithmes de tri par comparaisons. Cette borne est donnée dans la Proposition 22.

Proposition 22. *Le problème du tri d'une séquence X, de taille n, telle que $m_{rec}(X) = k$ admet une borne inférieure de $\Omega(n + k \log k)$ comparaisons.*

Démonstration. Il suffit d'appliquer le théorème 7 qui, pour rappel, nous dit dans le cas de m_{rec} que le problème du tri admet une borne inférieure de la forme $\Omega(\max(n, log|below(X, m_{rec}))|)$ avec $below(X, m_{rec})$ qui sont les permutations σ de taille n telles que $m_{rec}(\sigma) \leq k$. Pour obtenir le résultat, il ne nous reste donc qu'à déterminer ce qu'est le cardinal de cet ensemble. En particulier, nous allons démontrer l'inégalité

$$|below(X, m_{rec})| \geq k!$$

Construisons l'ensemble $\Lambda = \{\sigma \in \mathfrak{S}_n \,|\sigma(1) = 1, \sigma(2) = 2, \ldots, \sigma(n-k) = n-k\}$ qui est l'ensemble des permutations pour lesquelles les $n - k$ premiers éléments sont des points fixes. Observons que ces points fixes sont également des max-records dans ce cas et donc $m_{rec}(\sigma) \leq k$ pour $\sigma \in \Lambda$. Ainsi, nous avons la relation $\Lambda \subset below(X, m_{rec})$. Comme les k derniers éléments peuvent être fixés indépendamment pour toute permutation de Λ, nous avons que son cardinal est égal à $k!$. Cette égalité nous permet de démontrer l'inéquation. En utilisant cette inéquation nous prouvons le résultat annoncé. □

Un algorithme adaptatif qui atteint cette borne est alors considéré comme optimal pour cette mesure d'après les travaux de Mannila. L'algorithme 18 tri une séquence en s'adaptant à cette mesure de désordre. Cette algorithme fonctionne en plusieurs étapes. La première étape est l'extraction des max-records qui sont stockés dans l'ordre dans une sous-séquence temporaire que nous nommons ici Y. Comme ils sont dans l'ordre et par définition de ce qu'est un max-records, la séquence Y est donc triée. La deuxième étape est l'utilisation d'une fonction de

tri auxiliaire pour trier les éléments restants dans X qui ne sont pas des records. La troisième étape est la fusion des deux séquences triées X et Y qui permet d'obtenir la sortie. La proposition qui suit montre que l'algorithme est optimal pour la mesure m_{rec}.

Proposition 23. *L'algorithme 18 peut trier une séquence X de taille n, tel que $m_{rec}(X) = k$ en $\mathcal{O}(n + k \log k)$ comparaisons.*

Démonstration. Pour la première étape, il est suffisant de faire un balayage de la séquence en retenant le dernier max-record qui a été rencontré. Cet étape est donc équivalent à la recherche du maximum de X et nécessite $n - 1$ comparaisons. Pour la deuxième étape, le nombre d'éléments à trier restant dans X est exactement $m_{rec}(X) = k$. Si l'algorithme de tri auxiliaire utilisé est en $\mathcal{O}(k \log k)$, comme par exemple le tri fusion, alors cet étape nécessite $\mathcal{O}(k \log k)$ comparaisons. La troisième étape est une fusion de X et Y ce qui demande $n - 1$ comparaisons dans le pire des cas. En combinant la contribution de ces trois étapes, nous obtenons le résultat annoncé. □

L'étude d'une distribution qui favorise l'apparition de records dans une permutation est en partie motivée par le souhait de pouvoir analyser en moyenne l'Algorithme 18 qui joue le rôle d'exemple jouet.

6.2 Distribution d'Ewens

Nous allons dès à présent donner une définition d'une distribution connue dans la littérature par les probabilistes, à savoir la *distribution d'Ewens*. Il s'agit d'une distribution non-uniforme qui a l'avantage de favoriser les permutations qui ont un petit ou un grand nombre de cycles. Concrètement, cette distribution est paramétrée par une valeur réelle positive que nous noterons par la suite θ. Ce paramètre va favoriser l'apparition de cycles si $\theta > 1$ et va, au contraire, favoriser les permutations qui présentent peu de cycles si $\theta < 1$. Le cas où $\theta = 1$ donne la distribution uniforme.

Voyons maintenant comment est définie la probabilité d'avoir une permutation $\sigma \in \mathfrak{S}_n$ sous cette distribution.

Définition 24. Nous définissons la fonction $w_{\theta,n} : \mathfrak{S}_n \to \mathbb{R}^+$ qui associe pour tout paramètre $\theta > 0$ un poids à une permutation de taille n. La fonction est définie par l'équation suivante pour tout $\sigma \in \mathfrak{S}_n$,

$$w_{\theta,n}(\sigma) = \theta^{\mathbf{cyc}(\sigma)}. \tag{6.1}$$

Nous notons par la suite le poids total $W_{\theta,n}$ qui est la somme de tous des poids de toutes les permutations de taille n, soit

$$W_{\theta,n} = \sum_{\sigma \in \mathfrak{S}_n} w_{\theta,n}\sigma. \tag{6.2}$$

Définition 25. Pour toutes permutations $\sigma \in \mathfrak{S}_n$, la fonction définie par la relation

$$\mathbb{P}_{\theta,n}(\sigma) = \frac{w_{\theta,n}(\sigma)}{W_{\theta,n}}$$

est une mesure de probabilité. La distribution engendrée par cette mesure de probabilité est nommée la distribution d'Ewens.

Remarque. Comme cette distribution fait intervenir le nombre de cycles, il est naturel de voir apparaitre les nombres de Stirling de la première espèce non-signés pour certaines identités. Ces nombres de Stirling comptent le nombre de permutations de taille n qui sont composées d'exactement k cycles. Nous notons ces nombres $s_{n,k}$. Une identité évidente que nous avons est donnée par l'équation suivante :

$$W_{\theta,n} = \sum_{k=0}^{n} s_{n,k}\theta^k. \tag{6.3}$$

De cette remarque nous pouvons facilement démontrer le Lemme 17.

Lemme 17. *Soit $x^{(n)} = \prod_{k=0}^{n-1} x + k$ La fonction de Pochhammer, le poids total admet l'expression*

$$W_{\theta,n} = \theta^{(n)}. \tag{6.4}$$

Démonstration. Comme $W_{\theta,n} = \sum_{k=0}^{n} s_{n,k}\theta^k$, il suffit d'appliquer directement le Lemme 2 vu au cours du chapitre 3. □

6.3 Généralisation à d'autres statistiques

Nous venons de voir le modèle classique de la distribution d'Ewens. Il est naturel de généraliser cette distribution pour toute statistique $\chi_n : \mathfrak{S}_n \to \mathbb{R}^+$. L'idée est de remplacer la définition du poids en remplaçant les apparitions de $\mathbf{cyc}(\sigma)$ par la statistique $\chi(\sigma)$. Nous modifions donc nos définitions en conséquence.

Définition 26. Nous définissons la fonction $w_{\theta,n,\chi} : \mathfrak{S}_n \to \mathbb{R}^+$ qui associe pour tout paramètre $\theta > 0$ un poids à une permutation de taille n. La fonction est définie par l'équation suivante pour tout $\sigma \in \mathfrak{S}_n$,

$$w_{\theta,n,\chi}(\sigma) = \theta^{\chi(\sigma)}. \tag{6.5}$$

Le poids total $W_{\theta,n,\chi}$ est la somme de tous des poids de toutes les permutations

$$W_{\theta,n,\chi} = \sum_{\sigma\in\mathfrak{S}_n} w_{\theta,n,\chi}\sigma. \tag{6.6}$$

Définition 27. Pour toutes permutations $\sigma \in \mathfrak{S}_n$, la fonction définie par la relation

$$\mathbb{P}_{\theta,n,\chi}(\sigma) = \frac{w_{\theta,n,\chi}(\sigma)}{W_{\theta,n,\chi}}$$

est une mesure de probabilité. La distribution engendrée par cette mesure de probabilité est nommée la distribution d'Ewens généralisée à la statistique χ.

Remarque. Si $\theta = 1$, la distribution d'Ewens est encore une fois équivalente à la distribution uniforme. Chaque permutation ayant le même poids égal à 1.

Il est possible dans la littérature de trouver une distribution d'Ewens généralisée aux inversions connue sous le nom de distribution de Mallow. Cette distribution a été introduite par le statisticien Mallow, plus de détails sur cette distributions peuvent être trouvés dans l'article de Gladkich et Peled [26].

6.4 Génération aléatoire

Nous allons maintenant nous intéresser aux méthodes qui nous permettent de générer des permutations sous la distribution d'Ewens classique avec une fonction $random()$ qui à chaque appel nous donne un nombre dans $[0, 1]$ dont la mesure de probabilité est celle de Lebesgue, ce qui correspond au cas uniforme. Nous considérons de plus que chaque appel à cette fonction est indépendant des appels précédents.

La génération aléatoire nous permet de mieux comprendre les permutations que nous manipulons sous cette distribution. Savoir générer des permutations sous nos distributions peut nous permettre de vérifier les résultats que nous annoncerons par la suite. Pour ce qui suit, nous notons par $\theta > 0$ la valeur du paramètre de la distribution d'Ewens.

6.4.1 L'arbre de génération pour la distribution d'Ewens classique

Nous allons dans un premier temps nous intéresser à l'arbre de génération de la distribution d'Ewens classique. Il s'agit d'un arbre infini dont le premier nœud est l'ensemble vide. Un nœud de hauteur n va avoir plusieurs fils dont la permutation de taille $n + 1$ aura une relation avec la permutation de taille n du nœud père. Concrètement la relation consiste à insérer l'élément $n + 1$ dans la nouvelle permutation. Nous choisirons d'insérer l'élément dans un cycle existant dans la permutation du nœud père ou bien de créer un nouveau cycle avec cet élément. Comme nous sommes dans le cas du modèle d'Ewens, nous associons un poids de θ à l'action de création d'un nouveau cycle avec l'élément $n+1$, tandis que les actions d'insertion dans un cycle déjà existant n'aura qu'un poids de 1. Nous étiquetons chaque arête par un poids, soit 1 si aucun cycle n'est ajouté, soit θ dans le cas contraire. La Figure 6.1 représente cet arbre de génération pour les nœuds de hauteur allant jusqu'à 3. Nous avons la garantie sur cette arbre qu'à chaque nœud de hauteur n, nous n'avons qu'une unique permutation associée et qu'elles sont toutes présentes. Pour montrer l'unicité, remarquons dans un premier temps que par construction, si une permutation est présente plus d'une fois dans l'arbre, alors ils n'ont pas le même père. Nous pouvons alors montrer facilement par récurrence l'unicité. Nous savons déjà que l'unique permutation de $\mathfrak{S}_1$ est unique dans l'arbre.

Supposons donc que pour tout $i \leq n$ la propriété d'unicité est vraie. Raisonnons par l'absurde en supposant qu'il existe une permutation $\sigma' \in \mathfrak{S}_{n+1}$ qui est présente en plusieurs endroits. Le père de σ' est obtenu en retirant l'élément $n + 1$ de son cycle, de ce fait il n'existe qu'une unique permutation $\sigma \in \mathfrak{S}_n$. Comme les différents nœuds de σ' ne peuvent pas avoir le même père il faut donc que ces nœuds aient chacun un père différent dont la permutation correspondante est σ ce qui est une contradiction avec l'hypothèse de récurrence. Nous avons donc validé la propriété pour $n + 1$ et nous pouvons donc conclure par récurrence.

Comme nous avons déjà prouvé l'unicité, pour démontrer que toutes les permutations sont présentes, il nous suffit de compter. Nous pouvons une fois de plus le faire par récurrence. Nous savons que toutes les permutations de taille 1 sont présentes dans l'arbre. Supposons donc que pour tout $i \leq n$, nous avons l'ensemble des permutations de $\mathfrak{S}_i$ qui sont présentent. Si nous posons par $\mathcal{F}(\sigma)$

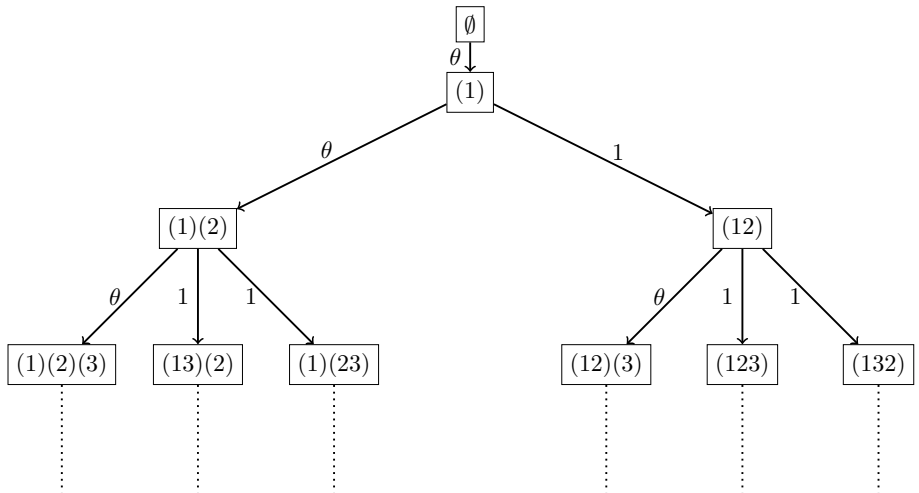


FIGURE 6.1 – Arbre de génération de la distribution d'Ewens de paramètre $\theta \in \mathbb{R}^+$.

l'ensemble des fils du nœud associé à $\sigma \in \mathfrak{S}_n$. Nous avons alors que l'ensemble des permutations de taille $n+1$ qui apparaissent dans l'arbre sont tous les fils des nœuds de hauteur n, ainsi

$$\sum_{\sigma \in \mathfrak{S}_n} |\mathcal{F}(\sigma)|.$$

Pour créer un fils au nœud qui associe σ, nous avons deux cas possibles, soit nous créons un nouveau cycle en faisant de $n+1$ un point fixe, soit nous l'insérons dans un cycle déjà existant. Pour le deuxième cas, il suffit d'insérer $n+1$ dans un cycle après un des éléments de σ, il y a donc n possibilités. Nous avons un total de $n+1$ possibilités et avons donc ainsi,

$$\begin{aligned}\sum_{\sigma \in \mathfrak{S}_n} |\mathcal{F}(\sigma)| &= \sum_{\sigma \in \mathfrak{S}_n} n+1 \\ &= n!(n+1) \\ &= (n+1)!\end{aligned}$$

Nous avons le compte correct de permutations de taille $n+1$ ce qui nous permet de conclure.

Maintenant que nous avons défini notre arbre, voyons comment nous pouvons nous en servir pour générer une permutation sous la distribution d'Ewens. Si nous voulons choisir une permutation de taille n, il suffit de choisir un des chemins de hauteur n et de prendre la permutation associée au nœud atteint au bout de ce chemin. Soit $\mathcal{C} = \langle \sigma_0 = \emptyset, \sigma_1, \sigma_2, \ldots, \sigma_n \rangle$ le chemin parcouru pour atteindre la permutation σ_n. Pour tout $i \in [n]$, nous avons $\sigma_i \in \mathfrak{S}_i$. Pour tout $i \in \{0 \ldots n-1\}$, nous savons qu'il y a une relation entre σ_i et σ_{i+1}. Soit $\mathcal{S}(\sigma)$, l'ensemble des permutations associées aux fils du nœud associé à la permutation $\sigma \in \mathfrak{S}_k$. Soit $p(\sigma, \gamma)$ qui associe à une permutation $\sigma \in \mathfrak{S}_k$ et à $\gamma \in \mathcal{S}(\sigma)$ le poids

de l'arête entre leurs nœuds respectifs. La probabilité de prendre le chemin σ_{i+1} sachant que son père est σ_i est donnée par

$$\mathbb{P}(\sigma_{i+1}|\sigma_i) = \frac{p(\sigma_i, \sigma_{i+1})}{\sum_{\gamma \in \mathcal{S}(\sigma_i)} p(\sigma, \gamma)}.$$

Nous pouvons remarquer qu'il n'y a qu'une seule permutation dans $\mathcal{S}(\sigma_i)$ qui a un cycle supplémentaire et qui a donc un poids de θ. Si nous l'insérons dans un cycle déjà existant, cela revient à regarder, dans la représentation en cycle de σ_i, après quel élément nous le plaçons. Pour tout $\sigma_i \in \mathfrak{S}_i$, nous avons toujours i possibilités. Nous obtenons ainsi,

$$\sum_{\gamma \in \mathcal{S}(\sigma_i)} p(\sigma, \gamma) = \theta + i$$

En utilisant la formule de Bayes, nous avons alors que la probabilité de choisir le chemin $\mathcal{C}$ dans l'arbre est donnée par la relation

$$\begin{aligned}
\mathbb{P}(\mathcal{C}) &= \prod_{i=0}^{n-1} \mathbb{P}(\sigma_{i+1}|\sigma_i) \\
&= \prod_{i=0}^{n-1} \frac{p(\sigma_i, \sigma_{i+1})}{\theta + i} \\
&= \frac{\prod_{i=0}^{n-1} p(\sigma_i, \sigma_{i+1})}{\theta^{(n)}}.
\end{aligned}$$

Dans le produit $\prod_{i=0}^{n-1} p(\sigma_i, \sigma_{i+1})$, nous avons des facteurs θ qui apparaissent à chaque fois qu'un cycle est créé entre deux nœuds. Comme les cycles précédents entre ces nœuds sont conservés, nous avons donc multiplié par θ autant de fois qu'il y a de cycles dans la dernière permutation du chemin, à savoir σ_n. Nous avons donc,

$$\prod_{i=0}^{n-1} p(\sigma_i, \sigma_{i+1}) = \theta^{\mathbf{cyc}(\sigma_n)},$$

et ainsi,

$$\mathbb{P}(\mathcal{C}) = \frac{\theta^{\mathbf{cyc}(\sigma_n)}}{\theta^{(n)}}.$$

La probabilité de choisir $\mathcal{C}$ et donc de choisir σ_n est égale à celle de la distribution d'Ewens comme nous l'avons souhaité.

6.4.2 Générateur naïf

Nous allons donc à partir de maintenant choisir un chemin dans l'arbre de génération. Le premier algorithme que nous proposons est une approche top-down, nous commençons par le nœud interne correspondant à l'unique permutation de $\mathfrak{S}_1$. Nous parcourons alors le chemin en descendant d'un niveau à chaque étape. Nous retenons directement l'ensemble des cycles de la permutation associé au nœud en cours. Une fois que nous avons ajouté l'élément n dans l'ensemble des cycles, il ne nous reste plus qu'à convertir cet ensemble en une permutation sous

forme d'un mot. L'algorithme 19 donne une description de ce générateur, nous y utilisons une procédure $random(a, b)$ qui donne un entier dans l'intervalle $\{a, \ldots, b\}$.

Données : Le paramètre $\theta \in \mathbb{R}^+$.
Résultat : $\pi \in \mathfrak{S}_n$ une permutation sous la distribution d'Ewens
$Cycles \leftarrow \{(1)\}$
pour $i \leftarrow 2$ **à** n **faire**
  si $random() < \frac{\theta}{\theta+i-1}$ **alors**
    $Cycles \leftarrow Cycles \cup \{(i)\}$
  sinon
    $k \leftarrow random(1, i-1)$
    Soit $c = (c(1), \ldots, k, \ldots, c(m)) \in Cycles$
    $c' \leftarrow (c(1), \ldots, k, i, \ldots, c(m))$
    $Cycles \leftarrow (Cycles \setminus \{c\}) \cup \{c'\}$
pour chaque $c = (c(1), \ldots, c(m_c)) \in Cycles$ **faire**
  $\pi(c(m_c)) \leftarrow c(1)$
  pour $i \leftarrow 1$ **à** $m_c - 1$ **faire**
    $\pi(c(i)) \leftarrow c(i+1)$
retourner π

Algorithme 19 : Générateur de permutation sous la distribution d'Ewens naïf.

En ne retenant que les cycles, les insertions dans un cycle déjà existant peuvent se révéler couteuses. Si les cycles sont stockés dans des tableaux, le pire cas est celui où nous obtenons une permutation à un seul cycle et où chaque insertion se fait après le premier élément. Pour ce choix de structure de données, nous aurions alors de l'ordre de $\Theta(n^2)$ déplacements. Il est possible de faire mieux et nous allons voir dans la section suivante comment nous pouvons modifier cet algorithme pour obtenir une complexité linéaire.

6.4.3 Générateur linéaire pour la distribution d'Ewens classique

Nous allons maintenant voir comment nous pouvons générer une permutation sous la distribution d'Ewens avec une complexité linéaire. Il est en réalité possible de générer une permutation sous sa forme de mot stocké dans un tableau. À chaque itération, la permutation associée au nœud interne de hauteur h est stockée dans les $h+1$ premières valeurs du tableau (la valeur à la position 0 n'étant pas utilisée). L'implémentation en Python de cet algorithme est donnée dans le Listing 2.

À l'initialisation, il est important d'avoir $\sigma(1) = 1$ pour commencer par le nœud associé à l'unique permutation de $\mathfrak{S}_1$. Nous initialisons en particulier σ par l'identité avec $\sigma(0) = 0$. De cette façon, nous avons déjà chargé l'espace nécessaire en mémoire pour stocker la permutation à retourner. Lorsque nous voulons insérer l'élément i dans la permutation, nous avons deux cas. Le premier cas est le cas où nous créons un nouveau cycle avec i, pour ce faire il suffit de faire de i un point fixe, et comme nous avons initialisé la permutation par

```python
from random import random, randint

def ewens_sampler(theta, n):
    sigma = [i for i in range(n+1)]
    for i in range(2,n+1):
        if random() < (i-1)/(theta+i-1):
            k = randint(1,i-1)
            sigma[i], sigma[k] = sigma[k], i
    return sigma
```

Listing 2 – Implémentation en python d'un générateur linéaire de permutation sous la distribution d'Ewens classique.

l'identité il n'y a donc rien à faire. Dans le deuxième cas, nous choisissons une valeur $k \in [i-1]$ de façon à avoir i dans le même cycle que k et situé juste après dans ce cycle. C'est ce que nous faisions dans les lignes 5 à 9 de l'Algorithme 19. Pour insérer i de cette façon, il y a une méthode simple. Le cycle avant l'insertion de i est de la forme $C = (k, \sigma(k), \sigma(\sigma(k)), \ldots, \sigma^{-1}(k))$. Après l'insertion, nous obtenons une nouvelle permutation σ' et le cycle C devient alors $C' = (k, i, \sigma(k), \sigma(\sigma(k)), \ldots, \sigma^{-1}(k))$. Il n'y a donc que peu de changements entre σ et σ', à savoir $\sigma'(i) = \sigma(k)$ et $\sigma'(k) = i$. C'est ce que nous appliquons dans les lignes 7 à 8 du Listing 2. Comme nous obtenons directement σ sous sa forme de mot nous n'avons pas à faire en plus la conversion provenant de la dernière boucle de l'Algorithme 19.

Dans le pire cas, le test de la ligne 6 est toujours vrai, et nous faisons alors deux déplacements par itération. Nous avons donc dans le pire cas de l'ordre de $\Theta(n)$ déplacements.

6.4.4 Générateur linéaire pour la distribution d'Ewens généralisée aux records

Nous allons par la suite nous intéresser particulièrement à la distribution d'Ewens généralisée aux records. Nous allons donc également créer un générateur pour cette distribution. Comme nous savons déjà générer une permutation sous la distribution d'Ewens classique, et que nous avons la bijection fondamentale pour convertir une permutation à k cycles en une permutation de même taille à k records, nous pouvons facilement générer une permutation sous la distribution d'Ewens généralisée aux records. Il suffit de générer une permutation $\pi \in \mathfrak{S}_n$ sous la distribution d'Ewens classique et d'appliquer la bijection fondamentale dessus pour obtenir $\sigma = F(\pi)$.

Pour que le générateur soit linéaire, il est donc nécessaire de pouvoir appliquer la bijection fondamentale en un temps au plus linéaire également. L'astuce est simple, il suffit de parcourir les cycles en commençant par ceux qui ont leur élément maximal le plus grand. Il est évident que le premier cycle à parcourir est celui qui contient n. Ce cycle correspond alors à la suite $(\pi(n), \pi(\pi(n)), \ldots, n)$. Il nous suffit donc de parcourir dans cet ordre ces éléments et de remplir en partant de la fin la permutation σ. Ainsi, pour ce premier cycle, nous aurons $\sigma(n) = \pi(n)$, $\sigma(n-1) = \pi(\pi(n))$ et ainsi de suite jusqu'à avoir un $j \in [n]$ tel que $\sigma[j] = n$. Une fois ce cycle parcouru, il faut chercher le cycle qui contient

le plus grand élément suivant, qui est la plus grande valeur qui n'appartient pas au cycle précédent. Pour savoir comment déterminer cet élément, nous pouvons simplement marquer les éléments déjà parcourus dans π et décrémenté un compteur jusqu'à trouver l'élément qui n'a pas été marqué. Si k est l'élément en question, nous remplissons alors $\sigma(j-1) = \pi(k)$, $\sigma(j-2) = \pi(\pi(k))$ et ainsi de suite. Nous parcourons tous les cycles de cette façon jusqu'à ce que nous ayons marqué tous les éléments de π. Nous donnons une implémentation de cette fonction, ainsi que du générateur linéaire pour cette distribution dans le Listing 3.

```
1  from linearsampler import ewens_sampler
2
3  def fundamental_bijection(pi):
4      p = i = len(pi) - 1
5      sigma = [0 for _ in range(len(pi))]
6      while i != 0:
7          while pi[p] != 0:
8              sigma[i] = pi[p]
9              pi[p] = 0
10             p = sigma[i]
11             i -= 1
12         p -= 1
13     return sigma
14
15 def ewens_record_sampler(theta, n):
16     return fundamental_bijection(ewens_sampler(theta,n))
```

Listing 3 – Implémentation en python d'un générateur linéaire de permutation sous la distribution d'Ewens généralisée aux records.

Nous avons en particulier ici décidé de marquer les éléments parcourus dans π par 0 à la ligne 9. Cette méthode est destructive mais nous n'avons plus besoin de la permutation d'origine, c'est pourquoi nous nous permettons de le faire. Procédons maintenant à une analyse de la complexité de cette implémentation. Les lignes 6 et 12 dépendent du nombre de fois où l'on peut décrémenter la valeur de p initialisé à n. Comme nous ne pouvons pas avoir $p < 0$, nous ne pouvons donc exécuter ces lignes qu'au plus n fois. Les lignes 7 à 11 sont exécutées exactement une fois par élément dans π. En effet, nous entrons dans la boucle de la ligne 7 uniquement si $\pi[p]$ n'est pas marqué. Nous avons donc exactement n exécutions de ces lignes. Toutes ces contributions se somment donc à une complexité en $\Theta(n)$ opérations. Comme l'appel des deux fonctions *fundamental_ bijection()* et *ewens_ sampler()* se font toutes deux en temps linéaire, notre générateur pour la distribution d'Ewens généralisée aux records est également linéaire.

6.5 Étude de la distribution d'Ewens généralisée aux records

Pour la suite, nous nous intéresserons à une distribution d'Ewens généralisée aux max-records. Comme il n'y a pas d'ambiguïté, nous noterons par la suite que $w_{\theta,n} = w_{\theta,n,\mathbf{rec}}$, $W_{\theta,n} = W_{\theta,n,\mathbf{rec}}$ et $\mathbb{P}_{\theta,n} = \mathbb{P}_{\theta,n,\mathbf{rec}}$.

Ce modèle présente l'avantage qu'il est simple à étudier. En effet, comme il existe la bijection fondamentale qui relie les permutations à k cycles avec les permutations à k records. Nous avons donc

$$W_{\theta,n,\mathbf{rec}} = W_{\theta,n,\mathbf{cyc}} = \theta^{(n)}. \tag{6.7}$$

De plus, ce modèle permet de favoriser l'apparition de séquences d'entrée proches du meilleur cas pour l'Algorithme 18.

Notre distribution favorise l'apparition de permutations qui sont presque triées lorsque le paramètre θ est grand. De ce fait, d'autres statistiques dépendantes de l'ordre dans la permutation peuvent se retrouver impactées. Nos premiers résultats théoriques concernent l'influence de cette distribution sur d'autres statistiques sur les permutations. Nous allons donner dans cette sous-section ces résultats.

Définition 28. Nous allons étendre la fonction de poids à d'autres séquences dont les éléments peuvent être ordonnés. Soit S une telle séquence d'éléments pris dans un ensemble E, alors le poids $w_\theta : E^+ \to \mathbb{R}^+$ est la fonction définie par la relation

$$w_\theta(S) = \theta^{\mathbf{rec}(S)}.$$

Exemple. *Nous pouvons par exemple avoir*
- $w_\theta(\langle 0.5, 6, 1.5, 9, 1.1, \pi, 10, 3\rangle) = \theta^4$
- $w_\theta(\langle 6, 4, 1, 7, 2\rangle) = \theta^2$
- $w_\theta(\text{"}abracadabra\text{"}) = \theta^3$ *où l'ordre utilisé est l'ordre alphabétique.*

Par la suite, nous utilisons cette extension de la fonction en particulier pour avoir le poids de sous-permutations d'une permutation $\sigma \in \mathfrak{S}_n$.

Définition 29. Pour une séquence S ordonnée, la fonction $rang(S, i)$ associe à un élément de position i dans S sa position dans la séquence $sort(S)$ obtenue après avoir trié S. Si deux éléments ont la même valeur alors nous choisirons de préserver l'ordre initial par principe de stabilité. La normalisation d'une séquence S par leurs rangs est la permutation de taille $n = |S|$, qui pour tout $i \in [n]$ associe $\mathrm{norm}(S)(i) = rang(S, i)$.

Exemple. *Si nous reprenons notre exemple précédent, nous avons*
- $\mathrm{norm}(\langle 0.5, 6, 1.5, 9, 1.1, \pi, 10, 3\rangle) = \langle 1, 6, 3, 7, 2, 5, 8, 4\rangle$
- $\mathrm{norm}(\langle 6, 4, 1, 7, 2\rangle) = \langle 4, 3, 1, 5, 2\rangle$
- $\mathrm{norm}(\text{"}abracadabra\text{"}) = \langle 1, 6, 10, 2, 8, 3, 9, 4, 7, 11, 5\rangle$ *où l'ordre utilisé est l'ordre alphabétique.*

Remarque. Comme dans une séquence S, la fonction de rang conserve l'ordre entre les éléments. Nous avons la relation suivante qui est vérifiée

$$w_\theta(S) = w_\theta(\mathrm{norm}(S)),$$

Définition 30. Nous allons maintenant étendre la fonction de poids à un ensemble de séquences dont les éléments peuvent être ordonnés. Soit X un ensemble de séquences de E^+, avec E un ensemble ordonné E. On définit $w_\theta(X) = \sum_{S \in X} w_\theta(S)$.

Remarque. Nous pouvons également faire une extension de l'opérateur norm à ces ensembles. De manière formelle, soit X un ensemble de séquences dans E^+. Nous définissons $\text{norm}(X) = \{norm(S) | S \in X\}$ qui consiste à ne garder que les séquences de X sous leurs formes normalisées par leurs rangs. D'après la remarque faite précédemment, nous avons alors que la relation

$$w_\theta(X) = w_\theta(\text{norm}(X))$$

est vérifiée.

Nous avons également la relation

$$W_{\theta,n} = w_\theta(\mathfrak{S}_n).$$

6.5.1 Lemmes techniques

Nous allons maintenant démontrer plusieurs lemmes qui vont nous servir à démontrer un grand nombre de nos résultats par la suite.

Nous utilisons la représentation d'une permutation $\sigma \in \mathfrak{S}_n$ sous la forme d'un mot. Dans cette représentation, il est possible de décomposer une permutation en une concaténation d'un préfixe π et d'un suffixe τ tel que $\sigma = \pi \cdot \tau$.

Définition 31. Pour tout entier positif n et pour tout $k \in \{0, \ldots, n\}$, nous définissons les arrangements par l'ensemble de toutes les sous-séquences de k éléments distincts pris dans $[n]$

$$\mathfrak{S}_{k \text{ in } n} = \left\{ s \in [n]^k \mid \forall i, j \in [k], i \neq j \Rightarrow s_i \neq s_j \right\}.$$

En particulier, si τ est de taille k, alors nous avons $\tau \in \mathfrak{S}_{k \text{ in } n}$ et $\pi \in \mathfrak{S}_{n-k \text{ in } n}$. Nous allons maintenant démontrer le lemme suivant :

Lemme 18. *Soit $n \in \mathbb{N}^*$, pour toute permutation $\sigma \in \mathfrak{S}_n$ et pour tout $k \in \{0, \ldots, n\}$ tel que $\sigma = \pi \cdot \tau$ avec $\tau \in \mathfrak{S}_{k \text{ in } n}$. La relation*

$$w_\theta(\sigma) = \frac{w_\theta(\pi) \times w_\theta(\langle \max(\pi) \rangle \cdot \tau)}{\theta} \tag{6.8}$$

est vérifiée, avec $\max(\pi)$ *qui est l'élément maximal dans la sous-séquence π.*

Démonstration. La preuve se fait en décomposant le comptage du nombre de records sur la permutation σ. Soit p le nombre de records de σ dans la sous-séquence π et soit t le nombre de records de σ dans la sous-séquence τ. Il est alors évident que nous avons la relation

$$\begin{aligned} w_\theta(\sigma) &= \theta^{p+t} \\ &= \theta^p \times \theta^t \end{aligned}$$

qui est vérifiée. Il est évident que nous avons la relation

$$w_\theta(\pi) = \theta^p$$

qui est vérifiée, par définition de p. Pour qu'un élément dans τ soit un record dans σ, il est, par définition des max-records, plus grand que tous les éléments qui le précèdent. Autrement dit, il s'agit d'un record dans τ qui est plus grand que l'élément maximal de π. Les records, dans la séquence où nous insérons au début de la séquence τ, sont donc tous, à l'exception du premier élément, des records de τ présent dans σ. Nous avons alors la relation

$$w_\theta(\langle \max(\pi) \rangle \cdot \tau) = \theta^{t+1}$$

qui est vérifiée. Il suffit de remplacer pour obtenir ainsi

$$\begin{aligned} w_\theta(\sigma) &= \theta^p \times \theta^t \\ &= w_\theta(\pi) \times \frac{w_\theta(\langle \max(\pi) \rangle \cdot \tau)}{\theta}, \end{aligned}$$

concluant ainsi la preuve. □

Définition 32. Nous définissons la fonction $w'_{\theta,n} : \cup_{k=0}^{n} \mathfrak{S}_{k \text{ in } n} \to \mathbb{R}^+$, qui associe pour chaque τ

$$w'_{\theta,n}(\tau) = \frac{w_\theta(\langle \max([n] \setminus \tau) \rangle \cdot \tau)}{\theta},$$

où $[n] \setminus \tau$ est l'ensemble des éléments de $[n]$ privé de tous les éléments dans τ.

En particulier, en appliquant le Lemme 18, nous avons, pour tout $\sigma = \pi \cdot \tau \in \mathfrak{S}_n$, que la relation

$$w_\theta(\sigma) = w_\theta(\pi) \times w'_{\theta,n}(\tau)$$

est satisfaite.

Exemple. *Nous avons par exemple* $w'_{\theta,8}(5378) = \frac{w_\theta(65378)}{\theta} = \theta^2$ *et* $w'_{\theta,9}(5378) = \frac{w_\theta(95378)}{\theta} = 1$.

Définition 33. Soit un ensemble $X \subset \mathfrak{S}_n$. Soit, pour tout suffixe $\tau \in \mathfrak{S}_{k \text{ in } n}$, l'ensemble des préfixes $X/\tau = \{\pi \in \mathfrak{S}_{n-k \text{ in } n} \,|\exists \sigma \in X,\ \sigma = \pi \cdot \tau\}$. L'ensemble X est **quotient-stable** de degré k, si il existe une constante que nous notons $w_k^q(X)$ telle que pour tout $\tau \in \mathfrak{S}_{k \text{ in } n}$ l'équation

$$w_\theta(X/\tau) = w_k^q(X)$$

est vérifiée.

Exemple. *L'ensemble* $\{4321, 3421, 4132, 3142, 4123, 2143, 3124, 1324\}$ *est quotient-stable de degré* 1. *On a en effet*

$$\begin{aligned} w(X/1) &= w(\{432, 342\}) \\ = w(X/2) &= w(\{413, 314\}) \\ = w(X/3) &= w(\{412, 214\}) \\ = w(X/4) &= w(\{312, 132\}) \\ = w_1^q(X) &= \theta + \theta^2. \end{aligned}$$

Proposition 24. *Si pour tout $\tau \in \mathfrak{S}_{k\,in\,n}$, il existe un ensemble $q(X,k)$ tel que* $\mathrm{norm}(X/\tau) = q(X,k)$*, alors X est quotient-stable.*

Démonstration. Rappelons que pour tout ensemble de séquence $Y \in E^+$, avec E ordonnée, nous avons $w_\theta(Y) = w_\theta(\mathrm{norm}(Y))$. C'est donc naturellement vrai si on choisit $Y = X/\tau$.

Ainsi, nous avons pour tout $\tau \in \mathfrak{S}_{k\,\mathrm{in}\,n}$ l'équation suivante qui est vraie :

$$w_\theta(X/\tau) = w_\theta(\mathrm{norm}(X/\tau)) = w_\theta(q(X,k)).$$

Et donc comme $q(X,k)$ ne dépend pas de τ il suffit de poser

$$w_k^q(X) = w(q(X,k))$$

ce qui nous permet de conclure. □

Remarque. Il faut toutefois faire attention, cette proposition n'est pas une équivalence mais bien une implication. En effet il existe des ensembles X qui n'admettent pas un tel $q(X,k)$ pour un degré k bien choisi mais qui sont quand même quotient-stables. C'est le cas, par exemple, de $X = \{4321, 4132, 4123\}$ qui est quotient-stable de degré 1 avec $w_k^q(X) = 1$ mais nous avons

$$(\mathrm{norm}(X/1) = \{321\}) \neq (\mathrm{norm}(X/2) = \{312\}).$$

La proposition que nous venons de voir implique directement le lemme suivant.

Lemme 19. *L'ensemble $\mathfrak{S}_n$ est quotient-stable pour tous les degrés $k \in [n]$ avec*

$$w_k^q(X) = \theta^{(n-k)}$$

Démonstration. Il suffit de voir que pour tout $\tau \in \mathfrak{S}_{k\,\mathrm{in}\,n}$, nous avons

$$\mathrm{norm}(\mathfrak{S}_n\,/\tau) = q(\mathfrak{S}_n, k) = \mathfrak{S}_{n-k}\,.$$

En appliquant la proposition de la remarque précédente, nous avons donc que $\mathfrak{S}_n$ est quotient-stable de degrés k avec

$$w_k^q(X) = w_\theta(\mathfrak{S}_{n-k}).$$

□

Lemme 20. *Soit $X \subset \mathfrak{S}_n$ quotient-stable de degrés $k \in [n]$, alors la relation*

$$w_\theta(X) = \frac{\theta^{(n)}}{\theta^{(n-k)}} w_k^q(X)$$

est vérifiée.

Démonstration. Par définition nous avons les équations suivantes qui sont véri-

fiées

$$\begin{aligned} w_\theta(X) &= \sum_{\sigma \in X} w_\theta(\sigma) \\ &= \sum_{\tau \in \mathfrak{S}_{k \text{ in } n}} \sum_{\pi \in X/\tau} w_\theta(\pi \cdot \tau) \end{aligned}$$

D'après le Lemme 18, nous pouvons décomposer $w_\theta(\pi \cdot \tau)$ par le produit $w_\theta(\pi) w'_{\theta,n}(\tau)$. Nous avons alors

$$\begin{aligned} w_\theta(X) &= \sum_{\tau \in \mathfrak{S}_{k \text{ in } n}} \sum_{\pi \in X/\tau} w_\theta(\pi) w'_{\theta,n}(\tau) \\ &= \sum_{\tau \in \mathfrak{S}_{k \text{ in } n}} w'_{\theta,n}(\tau) \sum_{\pi \in X/\tau} w_\theta(\pi) \end{aligned}$$

Comme par définition, $\sum_{\pi \in X/\tau} w_\theta(\pi) = w_\theta(X/\tau)$ et comme X est quotient stable, nous avons également $w_\theta(X/\tau) = w_k^q(X)$. Nous obtenons de cette façon l'équation suivante :

$$w_\theta(X) = w_k^q(X) \sum_{\tau \in \mathfrak{S}_{k \text{ in } n}} w'_{\theta,n}(\tau). \tag{6.9}$$

D'après le Lemme 19, nous savons que $\mathfrak{S}_n$ est quotient-stable de degrés k et que $w_k^q(\mathfrak{S}_n) = \theta^{(n-k)}$. L'Equation (6.9) est donc applicable et nous obtenons alors

$$w_\theta(\mathfrak{S}_n) = w_k^q(\mathfrak{S}_n) \sum_{\tau \in \mathfrak{S}_{k \text{ in } n}} w'_{\theta,n}(\tau).$$

De cette dernière égalité, nous obtenons alors l'équation suivante :

$$\sum_{\tau \in \mathfrak{S}_{k \text{ in } n}} w'_{\theta,n}(\tau) = \frac{\theta^{(n)}}{\theta^{(n-k)}}. \tag{6.10}$$

En combinant les Equations (6.9) et (6.10), nous obtenons le résultat annoncé. □

6.5.2 Influence aux autres statistiques

Le Lemme 20 nous permet d'obtenir nos premiers résultats sur l'influence de la distribution d'Ewens généralisée au record pour les autres statistiques. Cette section est consacrée à ces résultats.

Record à une position fixée

Le premier résultat que nous démontrerons concerne la probabilité de voir apparaitre un record à une position choisie $i \in [n]$. La distribution d'Ewens favorisant l'apparition de ces records dans la permutation en entier, il semble évident que cette probabilité varie grandement en fonction de la valeur de θ.

Lemme 21. *Soit l'ensemble* $X_{i,n} = \{\sigma \in \mathfrak{S}_n \,|\sigma(i) \text{ est un record}\}$. *L'ensemble* $X_{i,n}$ *est quotient-stable de degrés* $n-i$ *avec*

$$w^q_{n-i}(X_{i,n}) = \theta\theta^{(i-1)}.$$

Démonstration. Soit une permutation $\sigma \in X_{i,n}$, nous savons alors par définition que $\sigma(i)$ est un record. Par définition du max-record, nous savons que si nous décomposons $\sigma = \pi \cdot \tau$, avec $\pi \in \mathfrak{S}_{i\,\mathrm{in}\,n}$ et $\tau \in \mathfrak{S}_{n-i\,\mathrm{in}\,n}$, alors $\max(\pi) = \pi(i)$. Après normalisation, nous avons donc que $(\mathrm{norm}(\pi))(i) = i$, nous pouvons donc écrire que $\mathrm{norm}(\pi) = \pi' \cdot \langle i \rangle$ avec $\pi' \in \mathfrak{S}_{i-1\,\mathrm{in}\,n}$. Comme les éléments de π' peuvent apparaitre dans n'importe quel ordre nous avons de plus que $\pi' \in \mathfrak{S}_{i-1}$. Comme ce raisonnement ne dépend pas des valeurs de τ et de τ lui même, nous venons donc de montrer que pour tout $\tau \in \mathfrak{S}_{n-i\,\mathrm{in}\,n}$, la relation

$$\mathrm{norm}(X_{i,n}/\tau) = \{\sigma' \in \mathfrak{S}_i \,|\forall \pi' \in \mathfrak{S}_{i-1},\ \sigma' = \pi' \cdot \langle i \rangle\}$$

est vérifiée. Nous avons donc un même ensemble après normalisation $q(X_{i,n}, n-i) = \{\sigma' \in \mathfrak{S}_i \,|\forall \pi' \in \mathfrak{S}_{i-1},\ \sigma' = \pi' \cdot \langle i \rangle\}$, ce qui nous permet de conclure sur la quotient-stabilité de degrés $n-i$ de $X_{i,n}$. De plus, nous avons

$$\begin{aligned} w^q_{n-i}(X_{i,n}) &= w_\theta(q(X_{i,n}, n-i)) \\ &= \sum_{\sigma' \in q(X_{i,n}, n-i)} w_\theta(\sigma') \\ &= \sum_{\pi' \in \mathfrak{S}_{i-1}} w_\theta(\pi' \cdot \langle i \rangle). \end{aligned}$$

Comme i est un record dans $\pi' \cdot \langle i \rangle$, nous avons que

$$w_\theta(\pi' \cdot \langle i \rangle) = w_\theta(\pi')\theta.$$

Ainsi,

$$w^q_{n-i}(X_{i,n}) = \sum_{\pi' \in \mathfrak{S}_{i-1}} w_\theta(\pi')\theta \qquad\qquad = \theta\theta^{(i-1)},$$

ce qui conclut cette preuve. □

À l'aide de ce dernier lemme, nous pouvons démontrer le théorème qui suit.

Théorème 21. *Soit* $\sigma \in \mathfrak{S}_n$ *une permutation obtenue à partir de la distribution d'Ewens pour les records de paramètre* $\theta \in \mathbb{R}^{+*}$. *La probabilité que* $\sigma(i)$ *soit un record pour* $i \in [n]$ *est donnée par la relation*

$$\mathbb{P}(\sigma(i) \text{ est un record}) = \frac{\theta}{\theta + i - 1}.$$

Démonstration. Si $\sigma \in \mathfrak{S}_n$ a un record à la position $i \in [n]$, alors σ appartient à l'ensemble $X_{i,n}$. De ce fait, la probabilité que σ ait un record à la position i revient à vérifier que la permutation obtenue avec la distribution d'Ewens est dans l'ensemble $X_{i,n}$. Autrement dit, par additivité, nous avons

$$\begin{aligned}\mathbb{P}(\sigma(i) \text{ est un record}) &= \mathbb{P}(\sigma \in X_{i,n}) \\ &= \sum_{\sigma \in X_{i,n}} \mathbb{P}(\sigma) \\ &= \sum_{\sigma \in X_{i,n}} \frac{w_\theta(\sigma)}{w_\theta(\mathfrak{S}_n)} \\ &= \frac{\sum_{\sigma \in X_{i,n}} w_\theta(\sigma)}{w_\theta(\mathfrak{S}_n)} \\ &= \frac{w_\theta(X_{i,n})}{w_\theta(\mathfrak{S}_n)}.\end{aligned}$$

D'après le Lemme 21, nous savons que $X_{i,n}$ est quotient-stable de degrés $n-i$ avec

$$w^q_{n-i}(X_{i,n}) = \theta\theta^{(i-1)}.$$

En appliquant le Lemme 20, nous obtenons alors que

$$w_\theta(X_{i,n}) = \theta\theta^{(i-1)}\frac{\theta^{(n)}}{\theta^{(i)}}.$$

Ainsi nous obtenons,

$$\begin{aligned}\mathbb{P}(\sigma(i) \text{ est un record}) &= \frac{\theta\theta^{(i-1)}\frac{\theta^{(n)}}{\theta^{(i)}}}{\theta^{(n)}} \\ &= \frac{\theta\theta^{(i-1)}}{\theta^{(i)}}\end{aligned}$$

qui après une dernière simplification de $\frac{\theta^{(i-1)}}{\theta^{(i)}}$ par $\frac{1}{\theta+i-1}$ nous donne le résultat annoncé. □

Remarque. Nous avons un résultat qui correspond à nos attentes sur les cas critiques.

Quand $\theta = 1$, nous obtenons que

$$\mathbb{P}(\sigma(i) \text{ est un record}) = \frac{1}{i},$$

ce qui correspond bien à la probabilité que i soit un record pour le cas uniforme. Quand θ tend vers l'infini, la seule permutation que nous obtenons est l'identité et donc tous les éléments sont des records. Nous obtenons ici, comme prévu, que

$$\lim_{\theta \to \infty} \frac{\theta}{\theta + i - 1} = 1.$$

Dans le cas où θ est au voisinage de 0, nous obtenons des permutations qui présentent très peu de records. En particulier pour $i > 1$, nous avons

$$\lim_{\theta \to 0^+} \frac{\theta}{\theta + i - 1} = 0.$$

Intuitivement pour ce cas, nous obtenons donc avec forte probabilité, des permutations qui n'ont aucun record à l'exception du premier élément.

Il est possible d'obtenir une généralisation de ce théorème que nous allons utiliser par la suite. Nous allons nous intéresser au cas où il y a une suite de records et de non-records dans les permutations obtenues par la distribution d'Ewens pour les records.

Nous définissons deux ensembles pour la suite. Le premier ensemble considère les permutations qui ont une suite de non-records suivis par une suite de records. Pour tout $n \in \mathbb{N}^*$ et pour tout $a, b, c \in [n]$ avec $a \leq b \leq c$, nous posons l'ensemble

$$\mathcal{R}_n^{\circ\bullet}(a,b,c) = \{\sigma \in \mathfrak{S}_n \mid \forall i \in [a,b], \sigma(i) \textit{ est un non}-\textit{record et } \forall j \in [b{+}1,c], \sigma(j) \textit{ est un record}\}.$$

Le deuxième ensemble inter-change la zone des non-records avec la zone des records. Pour tout $n \in \mathbb{N}^*$ et pour tout $a, b, c \in [n]$ avec $a \leq b \leq c$, nous posons l'ensemble

$$\mathcal{R}_n^{\bullet\circ}(a,b,c) = \{\sigma \in \mathfrak{S}_n \mid \forall i \in [a,b], \sigma(i) \textit{ est un record et } \forall j \in [b{+}1,c], \sigma(j) \textit{ est un non}-\textit{record}\}.$$

Lemme 22. *L'ensemble $\mathcal{R}_n^{\circ\bullet}(a,b,c)$ est quotient-stable de degrés $n-c$ avec*

$$w_{n-c}^q(\mathcal{R}_n^{\circ\bullet}(a,b,c)) = \theta^{c-b}\theta^{(a-1)}\frac{(b-1)!}{(a-2)!}$$

Démonstration. Pour tout $\tau \in \mathfrak{S}_{n-c \text{ in } n}$, nous nous intéressons à l'ensemble $\mathcal{R}_n^{\circ\bullet}(a,b,c)/\tau$. Soit $\sigma = \pi \cdot \tau \in \mathfrak{S}_n$, par définition pour tout $i \in [c]$, le fait que $\sigma(i) = \pi(i)$ est un record dépend uniquement des valeurs des $\sigma(j)$ pour $j \in [i-1]$. L'apparition de records dans π ne dépendent pas des éléments de τ et donc $\mathcal{R}_n^{\circ\bullet}(a,b,c)/\tau$ est non-vide. Soit $\pi \in \mathcal{R}_n^{\circ\bullet}(a,b,c)$, étudions en détails les propriétés de $\rho = \text{norm}(\pi) \in \mathfrak{S}_c$.

- Nous pouvons montrer que pour tout $i \in [b+1, c]$, nous avons $\rho(i) = i$. Pour ce faire, nous pouvons faire un raisonnement par récurrence et par l'absurde. Supposons que $\rho(c) \neq c$, dans ce cas comme $\rho \in \mathfrak{S}_c$, alors $\rho(c) < c$ et il existe une position $j \in [c-1]$ telle que $\rho(j) = c$. Dans ce cas le couple (c, j) est une inversion et donc $\rho(c)$ n'est pas un record ce qui est une contradiction avec la définition de $\mathcal{R}_n^{\circ\bullet}(a,b,c)$. Nous avons donc montré par l'absurde que $\rho(c) = c$. Supposons maintenant que pour un certain rang $b' > b$, pour tout $i \in [b'+1, c]$ nous avons $\rho(i) = i$. Nous venons de montrer que c'était vrai pour $b' = c-1$. Supposons que $\rho(b') \neq b'$, dans ce cas comme $\rho \in \mathfrak{S}_c$ et comme l'hypothèse de récurrence implique que pour tout $k \in [b'+1, c]$ il existe un $i \in [c] \setminus \{b'\}$ tel que $\rho(i) = k$, nous avons alors $\rho(b') < b'$. De plus, l'hypothèse de récurrence implique que pour tout $i \in [b'+1, c]$ nous avons $\rho(i) \neq b'$, et donc pour que $\rho \in \mathfrak{S}_n$, il est nécessaire qu'il existe une position $j \in [b'-1]$ telle que $\rho(j) = b'$. Nous avons donc que le couple (j, b') est une inversion et donc $\rho(b')$ ne peut pas être un record ce qui est une contradiction avec la définition de $\mathcal{R}_n^{\circ\bullet}(a,b,c)$. Nous venons donc de démontrer que l'hypothèse de récurrence au rang b' implique notre hypothèse de récurrence au rang $b'-1$. Comme nous ne pouvons itérer ce raisonnement jusqu'au rang $b' = b+1$, nous démontrons finalement par récurrence que l'hypothèse est vraie pour $b' = b$ ce qui correspond à la propriété que nous voulions

démontrer.

* Il existe une position $i \in [a-1]$ telle que $\rho(i) = b$. Supposons le contraire, du fait que la première propriété est vraie et fixe les valeurs prises aux positions de l'intervalle $[b+1, c]$, il n'est possible de placer la valeur b que dans une des positions de l'intervalle $[a, b]$. Autrement dit, il existe dans ce cas une position $j \in [a, b]$ telle que $\rho(j) = b$. Or comme les valeurs plus grandes que b sont placées après la position $b+1$, nous savons que b est l'élément maximal parmi les éléments aux positions de l'intervalle $[b]$ et ainsi $\rho(j)$ est un record. Cette dernière affirmation est en contradiction avec la définition de $\mathcal{R}_n^{\circ\bullet}(a, b, c)$ et nous venons donc de démontrer par l'absurde la propriété.
* Les valeurs aux positions de $[b] \setminus \{p\}$ sont libres d'accueillir les valeurs restantes dans $[b-1]$. En effet, la première propriété implique que les valeurs aux positions $[b+1, c]$ sont déjà fixées. Comme les valeurs aux positions $[a, b]$ sont choisies dans $[b-1]$ et comme la deuxième propriété est vérifiée, nous avons que pour tout $q \in [a, b]$, le couple (p, q) est une inversion et donc $\rho(q)$ n'est pas un record. Il n'y a donc aucune contradiction à cette propriété.

Ces trois propriétés sont nécessaires et suffisantes pour définir l'ensemble $\mathrm{norm}(\mathcal{R}_n^{\circ\bullet}(a, b, c)/\tau)$. Nous obtenons ainsi,

$$\begin{aligned} X_{a,b,c}^{\circ\bullet} &= \mathrm{norm}(\mathcal{R}_n^{\circ\bullet}(a, b, c)/\tau) \\ &= \{\sigma' \in \mathfrak{S}_c \,|\exists p \in [a-1]\sigma'(p) = b \; et \; \forall i \in [b+1, c]\sigma'(i) = i\}. \end{aligned}$$

Nous pouvons partitionner cette ensemble en considérant un ensemble $A \subset [b-1]$ des valeurs aux positions $[a, b]$. Ainsi nous avons

$$\begin{aligned} X_{a,b,c}^{\circ\bullet} &= \bigcup_{\substack{A \subset [b-1] \\ |A| = b-a+1}} \{\sigma' \in \mathfrak{S}_c \,|\forall j \in [a, b]\sigma'(j) \in A \; et \; \forall i \in [b+1, c]\sigma'(i) = i\} \\ &= \bigcup_{\substack{A \subset [b-1] \\ |A| = b-a+1}} Y_{a,b,c}(A). \end{aligned}$$

Comme cet ensemble ne dépend pas de τ, nous avons déjà démontré que l'ensemble $\mathcal{R}_n^{\circ\bullet}(a, b, c)$ est quotient-stable de degrés $n - c$. Nous savons de plus que

$$w_{n-c}^q(\mathcal{R}_n^{\circ\bullet}(a, b, c)) = w_\theta(X_{a,b,c}^{\circ\bullet}).$$

Pour tout ensemble A de ce type, déterminons la valeur de $w_\theta(Y_{a,b,c}(A))$. Toute permutation $\sigma' \in Y_{a,b,c}(A)$ peut être vue comme la concaténation de trois mots. Nous avons en particulier pour $\alpha \in \mathfrak{S}_{a-1\,\mathrm{in}\,c}$, $\beta \in \mathfrak{S}_{b-a+1\,\mathrm{in}\,c}$ et $\gamma \in \mathfrak{S}_{c-b\,\mathrm{in}\,c}$ la décomposition

$$\sigma = \alpha \cdot \beta \cdot \gamma.$$

De cette décomposition, nous obtenons d'après le Lemme 18, l'identité suivante :

$$w_\theta(\sigma) = \frac{w_\theta(\alpha) \times w_\theta(\langle \max(\alpha) \rangle \cdot \beta) \times w_\theta(\langle \max(\alpha \cdot \beta) \rangle \cdot \gamma)}{\theta^2}.$$

Par définition de $Y_{a,b,c}(A)$, $\max(\alpha) = b$ et tous les éléments de β sont plus petit que b, ainsi nous avons

$$w_\theta(\langle \max(\alpha) \rangle \cdot \beta) = \theta.$$

De la même façon, nous savons que $\max(\alpha \cdot \beta) = b$ et que tous les éléments de γ sont plus grand que b. De plus, comme pour tout $i \in [b+1, c]$, $\sigma'(i) = i$, chaque élément de γ est un record. Nous avons ainsi

$$w_\theta(\langle \max(\alpha \cdot \beta) \rangle \cdot \gamma) = \theta^{c-b+1}.$$

En combinant ces égalités, nous avons alors que

$$\begin{aligned} w_\theta(\sigma) &= \frac{w_\theta(\alpha) \times \theta \times \theta^{c-b+1}}{\theta^2} \\ &= w_\theta(\alpha)\theta^{c-b}. \end{aligned}$$

Nous avons alors les égalités suivantes qui sont vérifiées

$$\begin{aligned} w_\theta(Y_{a,b,c}(A)) &= \sum_{\sigma \in Y_{a,b,c}(A)} w_\theta(\sigma) \\ &= \sum_{\substack{\alpha \in \mathfrak{S}_{a-1 \text{ in } c} \\ \exists \tau' \in \mathfrak{S}_{c-a+1 \text{ in } c}\, \alpha \cdot \tau' \in Y_{a,b,c}(A)}} \left(\sum_{\substack{\beta \in \mathfrak{S}_{b-a+1 \text{ in } c} \\ \exists \gamma \in \mathfrak{S}_{c-b \text{ in } c}\, \alpha \cdot \beta \cdot \gamma \in Y_{a,b,c}(A)}} w_\theta(\alpha)\theta^{c-b} \right) \\ &= \theta^{c-b} \sum_{\substack{\alpha \in \mathfrak{S}_{a-1 \text{ in } c} \\ \exists \tau' \in \mathfrak{S}_{c-a+1 \text{ in } c}\, \alpha \cdot \tau' \in Y_{a,b,c}(A)}} \left(w_\theta(\alpha) \sum_{\substack{\beta \in \mathfrak{S}_{b-a+1 \text{ in } c} \\ \exists \gamma \in \mathfrak{S}_{c-b \text{ in } c}\, \alpha \cdot \beta \cdot \gamma \in Y_{a,b,c}(A)}} 1 \right) \end{aligned}$$

Comme nous pouvons choisir d'attribuer à toute position de l'intervalle $[a, b]$ une valeur de A, nous avons $(b-a+1)!$ possibilités. Ainsi, nous avons

$$\sum_{\substack{\beta \in \mathfrak{S}_{b-a+1 \text{ in } c} \\ \exists \gamma \in \mathfrak{S}_{c-b \text{ in } c}\, \alpha \cdot \beta \cdot \gamma \in Y_{a,b,c}(A)}} 1 = (b-a+1)!,$$

et donc,

$$w_\theta(Y_{a,b,c}(A)) = \theta^{c-b}(b-a+1)! \sum_{\substack{\alpha \in \mathfrak{S}_{a-1 \text{ in } c} \\ \exists \tau' \in \mathfrak{S}_{c-a+1 \text{ in } c}\, \alpha \cdot \tau' \in Y_{a,b,c}(A)}} w_\theta(\alpha).$$

Construire un préfixe α comme décrit plus haut revient à placer aux positions de l'intervalle $[a-1]$ les valeurs de l'ensemble $A^{\complement}$. En normalisant, α, nous

obtenons alors l'ensemble des permutations de tailles $a-1$. Ainsi, nous avons

$$\begin{aligned}\sum_{\substack{\alpha\in\mathfrak{S}_{a-1\,\mathrm{in}\,c}\\ \exists\tau'\in\mathfrak{S}_{c-a+1\,\mathrm{in}\,c}\,\alpha\cdot\tau'\in Y_{a,b,c}(A)}} w_\theta(\alpha) &= \sum_{\substack{\alpha\in\mathfrak{S}_{a-1\,\mathrm{in}\,c}\\ \exists\tau'\in\mathfrak{S}_{c-a+1\,\mathrm{in}\,c}\,\alpha\cdot\tau'\in Y_{a,b,c}(A)}} w_\theta(\mathrm{norm}(\alpha))\\ &= w_\theta(\mathfrak{S}_{a-1})\\ &= \theta^{(a-1)}.\end{aligned}$$

De cette dernière égalité, nous obtenons l'identité suivante :

$$w_\theta(Y_{a,b,c}(A)) = \theta^{c-b}(b-a+1)!\theta^{(a-1)}.$$

Nous pouvons maintenant obtenir les égalités suivantes à partir du partitionnement de $X^{\circ\bullet}_{a,b,c}$:

$$\begin{aligned}w_\theta(X^{\circ\bullet}_{a,b,c}) &= \sum_{\substack{A\subset[b-1]\\|A|=b-a+1}} w_\theta(Y_{a,b,c}(A))\\ &= \sum_{\substack{A\subset[b-1]\\|A|=b-a+1}} \theta^{c-b}(b-a+1)!\theta^{(a-1)}\\ &= \theta^{c-b}(b-a+1)!\theta^{(a-1)} \sum_{\substack{A\subset[b-1]\\|A|=b-a+1}} 1\\ &= \theta^{c-b}(b-a+1)!\theta^{(a-1)}\binom{b-1}{b-a+1}\\ &= \theta^{c-b}\theta^{(a-1)}\frac{(b-1)!}{(a-2)!}.\end{aligned}$$

Cette dernière égalité nous permet de conclure la preuve. □

Nous pouvons utiliser ce dernier lemme pour démontrer une proposition qui nous donne un outil supplémentaire pour manipuler la fonction de Pochhammer.

Proposition 25. *Pour tout $\theta\in\mathbb{R}^+$, Nous avons l'égalité suivante qui est vérifiée*

$$\sum_{i=0}^{n-2}\frac{\theta^{(i)}}{i!} = \frac{\theta^{(n-1)}}{\theta(n-2)!}. \tag{6.11}$$

Démonstration. Cette preuve est basée sur une autre décomposition de l'ensemble $X^{\circ\bullet}_{a,b,c}$ que nous avons définie au cours de la preuve du Lemme 22. Nous avions déjà montré que cet ensemble était définit par la relation

$$X^{\circ\bullet}_{a,b,c} = \{\sigma'\in\mathfrak{S}_c\,|\exists p\in[a-1]\sigma'(p)=b\ et\ \forall i\in[b+1,c]\sigma'(i)=i\}.$$

Il est facile de partitionner cet ensemble en une union pour tous les p possibles

dans $[a-1]$, ce qui nous donne simplement

$$\begin{aligned} X^{\circ\bullet}_{a,b,c} &= \bigcup_{p=1}^{a-1} \{\sigma' \in \mathfrak{S}_c \,|\sigma'(p) = b \; et \; \forall i \in [b+1, c]\sigma'(i) = i\} \\ &= \bigcup_{p=1}^{a-1} Z_{b,c,p}. \end{aligned}$$

Nous allons maintenant utiliser un partitionnement similaire à la preuve précédente, le partitionnement se fera cependant sur l'ensemble $Z_{b,c,p}$. L'idée est de choisir les éléments qui seront placés aux positions de l'intervalle $[p+1, b]$. Pour ce faire, nous posons un ensemble $A \subset [b-1]$ et de cardinal $|A| = b - p$. Nous définissons maintenant l'ensemble $T_{b,c,p}(A)$ par la relation

$$T_{b,c,p}(A) = \{\sigma' \in \mathfrak{S}_c \,|\sigma'(p) = b \, et \, \forall i \in [b{+}1, c]\sigma'(i) = i \, et \, \forall j \in [p{+}1, b]\sigma'(j) \in A\}.$$

Il est évident que nous pouvons partitionner pour tous les A possibles, nous avons alors

$$Z_{b,c,p} = \bigcup_{\substack{A \subset [b-1] \\ |A| = b-p}} T_{b,c,p}(A).$$

Nous avons enfin

$$X^{\circ\bullet}_{a,b,c} = \bigcup_{p=1}^{a-1} \bigcup_{\substack{A \subset [b-1] \\ |A| = b-p}} T_{b,c,p}(A),$$

et comme cette union est disjointe, nous avons directement

$$w_\theta(X^{\circ\bullet}_{a,b,c}) = \sum_{p=1}^{a-1} \sum_{\substack{A \subset [b-1] \\ |A| = b-p}} w_\theta(T_{b,c,p}(A)).$$

Il nous reste alors à déterminer ce que vaut $w_\theta(T_{b,c,p}(A))$. Soit $\sigma' \in T_{b,c,p}(A)$, nous pouvons le décomposer de la façon suivante

$$\sigma' = c(\pi', \tau') = \pi' \cdot \langle b \rangle \cdot \tau' \cdot \langle b+1, \cdots, c \rangle \,,$$

avec $\pi' \in \mathfrak{S}_{p-1 \,\mathrm{in}\, c}$ et $\tau' \in \mathfrak{S}_{b-p \,\mathrm{in}\, c}$. En utilisant le Lemme 18, nous pouvons montrer que

$$w_\theta(\sigma') = w_\theta(\pi') \times \theta \times 1 \times \theta^{c-b},$$

et nous avons alors

$$\begin{aligned}
w_\theta(T_{b,c,p}(A)) &= \sum_{\pi' \in \mathfrak{S}_{p-1 \text{ in } c}} \sum_{\substack{\tau' \in \mathfrak{S}_{b-p \text{ in } c} \\ c(\pi',\tau') \in T_{b,c,p}(A)}} w_\theta(\pi') \theta^{c-b+1} \\
&= \theta^{c-b+1} \sum_{\pi' \in \mathfrak{S}_{p-1 \text{ in } c}} w_\theta(\pi') \sum_{\substack{\tau' \in \mathfrak{S}_{b-p \text{ in } c} \\ c(\pi',\tau') \in T_{b,c,p}(A)}} 1 \\
&= \theta^{c-b+1} \sum_{\substack{\pi' \in \mathfrak{S}_{p-1 \text{ in } c} \\ \exists \tau' \in \mathfrak{S}_{b-p \text{ in } c}\, c(\pi',\tau') \in T_{b,c,p}(A)}} w_\theta(\pi')(b-p)! \\
&= \theta^{c-b+1} \theta^{(p-1)} (b-p)!
\end{aligned}$$

Au final nous avons

$$\begin{aligned}
w_\theta(X^{\circ\bullet}_{a,b,c}) &= \sum_{p=1}^{a-1} \sum_{\substack{A \subset [b-1] \\ |A| = b-p}} \theta^{c-b+1} \theta^{(p-1)} (b-p)! \\
&= \theta^{c-b+1} \sum_{p=1}^{a-1} \theta^{(p-1)} (b-p)! \sum_{\substack{A \subset [b-1] \\ |A| = b-p}} 1 \\
&= \theta^{c-b+1} \sum_{p=1}^{a-1} \theta^{(p-1)} (b-p)! \binom{b-1}{b-p} \\
&= \theta^{c-b+1} (b-1)! \sum_{p=1}^{a-1} \frac{\theta^{(p-1)}}{(p-1)!}.
\end{aligned}$$

Nous savons également par le Lemme 22 que

$$w_\theta(X^{\circ\bullet}_{a,b,c}) = \theta^{c-b} \theta^{(a-1)} \frac{(b-1)!}{(a-2)!},$$

et ainsi nous avons l'égalité

$$\theta^{c-b+1} (b-1)! \sum_{p=1}^{a-1} \frac{\theta^{(p-1)}}{(p-1)!} = \theta^{c-b} \theta^{(a-1)} \frac{(b-1)!}{(a-2)!}.$$

En procédant à quelques réarrangements et changements de variables de cette dernière égalité nous obtenons le résultat annoncé. □

Lemme 23. *L'ensemble $\mathcal{R}_n^{\bullet\circ}(a,b,c)$ est quotient-stable de degrés $n-c$ avec*

$$w^q_{n-c}(\mathcal{R}_n^{\bullet\circ}(a,b,c)) = \theta^{b-a+1} \theta^{(a-1)} \frac{(c-1)!}{(b-1)!}$$

Démonstration. Pour les mêmes raisons que ce que nous avons vu durant la preuve du Lemme 22, pour tout $\tau \in \mathfrak{S}_{n-c \text{ in } n}$ l'ensemble $\mathcal{R}_n^{\bullet\circ}(a,b,c)/\tau$ est non-vide. Pour tout $\pi \in \mathcal{R}_n^{\bullet\circ}(a,b,c)/\tau$, la permutation $\rho = \text{norm}(\pi) \in \mathfrak{S}_c$ vérifie les propriétés suivantes :

* $\rho(b) = c$. Premièrement, remarquons que pour tout $j \in [b+1, c]$, nous avons $\rho(j) \neq c$. En effet, si c'était le cas, comme c est l'élément maximal de ρ alors $\rho(j) = c$ serait un record, ce qui entre en contradiction avec la définition de $\mathcal{R}_n^{\bullet\circ}(a, b, c)$. Si l'on suppose à présent que $\rho(b) \neq c$, il doit alors exister une position $p \in [b-1]$ telle que $\rho(p) = c$ et $\rho(b) < c$ et ainsi $\rho(b)$ ne peut pas être un record ce qui est une contradiction avec la définition de $\mathcal{R}_n^{\bullet\circ}(a, b, c)$.
* Soit l'ensemble $A_{\sigma,b} = \{\sigma(i) | \forall i \in [b-1]\}$ des éléments de σ aux positions dans l'intervalle $[b-1]$. L'ensemble $A_{\sigma,b}$ peut être n'importe quel ensemble de cardinal $b-1$ qui est inclus dans $[c-1]$. En effet, choisir les éléments de $A_{\sigma,b}$ revient à éliminer les éléments de $[c-1]$ qui seront placés aux positions de l'intervalle $[b+1, c]$. Comme $\rho(b) = c$ alors toutes ces valeurs sont plus petites que $\rho(b)$ ce qui implique que quelles que soient les valeurs choisies dans A, les éléments aux positions de l'intervalle $[b+1, c]$ ne sont pas des records, conformément à la définition de $\mathcal{R}_n^{\bullet\circ}(a, b, c)$.
* Soit $A_{\sigma,b} = \{\alpha_1, \ldots, \alpha_{b-1}\}$ tel que $\alpha_1 < \cdots < \alpha_{b-1}$. Pour tout $i \in [a, b-1]$, nous avons $\rho(i) = \alpha_i$. La preuve est similaire à la première propriété vue au cours du Lemme 22 qui utilisait un raisonnement par récurrence et par l'absurde.

Ces trois propriétés sont nécessaires et suffisantes pour définir $\mathrm{norm}(\mathcal{R}_n^{\bullet\circ}(a, b, c))$. Si nous posons

$$X_{a,b,c}(A) = \{\rho \in \mathfrak{S}_c \,|\forall p \in [a-1]\ \rho(p) \in A \,et\, \forall q \in [a, b-1]\ \rho(q) = \alpha_q \,et\, \rho(b) = c\},$$

nous avons alors

$$\mathrm{norm}(\mathcal{R}_n^{\bullet\circ}(a, b, c)/\tau) = \bigcup_{A=\{\alpha_1,\ldots,\alpha_{b-1}\}} X_{a,b,c}(A).$$

Comme cet ensemble ne dépend pas de τ, l'ensemble $\mathcal{R}_n^{\bullet\circ}(a, b, c)$ est quotient-stable de degrés $n - c$ et nous avons

$$w_{n-c}^q(\mathcal{R}_n^{\bullet\circ}(a, b, c)) = w_\theta(\mathrm{norm}(\mathcal{R}_n^{\bullet\circ}(a, b, c)/\tau)).$$

Pour tout $A = \{\alpha_1, \ldots, \alpha_{b-1}\} \subset [c-1]$ avec $\alpha_1 < \cdots < \alpha_{b-1}$. Déterminons $w_\theta(X_{a,b,c}(A))$. Soit $\rho \in X_{a,b,c}(A)$, alors il existe $\beta \in \mathfrak{S}_{a-1\,\mathrm{in}\,c}$, $\gamma \in \mathfrak{S}_{b-a+1\,\mathrm{in}\,c}$ et $\delta \in \mathfrak{S}_{c-b\,\mathrm{in}\,c}$ tels que $\rho = \beta \cdot \gamma \cdot \delta$. En utilisant le Lemme 18, nous avons l'égalité suivante :

$$w_\theta(\rho) = \frac{w_\theta(\beta) w_\theta(\langle \max(\beta) \rangle \cdot \gamma) w_\theta(\langle \max(\beta \cdot \gamma) \rangle \cdot \delta)}{\theta^2}.$$

Par définition, tous les éléments de γ sont des records de ρ, nous avons donc

$$w_\theta(\langle \max(\beta) \rangle \cdot \gamma) = \theta^{b-a+2}.$$

De même, tout élément de δ n'est pas un record de ρ, nous avons donc

$$w_\theta(\langle \max(\beta \cdot \gamma) \rangle \cdot \delta) = \theta.$$

Ainsi, nous avons

$$w_\theta(\rho) = w_\theta(\beta)\theta^{b-a+1}.$$

Nous avons les égalités suivantes qui sont alors vérifiées

$$\begin{aligned} w_\theta(X_{a,b,c}(A)) &= \sum_{\rho \in X_{a,b,c}(A)} w_\theta(\rho) \\ &= \sum_{\beta \in \mathfrak{S}_{a-1 \text{ in } c}} \sum_{\gamma \in \mathfrak{S}_{b-a+1 \text{ in } c}} \sum_{\substack{\delta \in \mathfrak{S}_{c-b \text{ in } c} \\ \beta\cdot\gamma\cdot\delta \in X_{a,b,c}(A)}} w_\theta(\beta)\theta^{b-a+1} \end{aligned}$$

Par définition, il n'existe qu'un unique γ^* tel que $\rho = \beta \cdot \gamma^* \cdot \delta$. Nous avons alors

$$w_\theta(X_{a,b,c}(A)) = \theta^{b-a+1} \sum_{\beta \in \mathfrak{S}_{a-1 \text{ in } c}} w_\theta(\beta) \sum_{\substack{\delta \in \mathfrak{S}_{c-b \text{ in } c} \\ \beta\cdot\gamma^*\cdot\delta \in X_{a,b,c}(A)}} 1$$

δ est une permutation des éléments de l'ensemble $A^{\complement}$ ce qui laisse $(c-b)!$ possibilités. De plus β est également une permutation des éléments de l'ensemble $\{\alpha_i | i \in [a-1]\}$, et donc la somme des poids de toutes les permutations possibles est $w_\theta(\mathfrak{S}_{a-1}) = \theta^{(a-1)}$. Nous obtenons finalement l'égalité

$$w_\theta(X_{a,b,c}(A)) = \theta^{b-a+1}(c-b)!\theta^{(a-1)}.$$

Enfin, nous avons les égalités suivantes :

$$\begin{aligned} w^q_{n-c}(\mathcal{R}_n^{\bullet\circ}(a,b,c)) &= \sum_{\substack{A \subset [c-1] \\ |A| = b-1}} w_\theta(X_{a,b,c}(A)) \\ &= \sum_{\substack{A \subset [c-1] \\ |A| = b-1}} \theta^{b-a+1}(c-b)!\theta^{(a-1)} \\ &= \theta^{b-a+1}(c-b)!\theta^{(a-1)}\binom{c-1}{b-1} \\ &= \theta^{b-a+1}\theta^{(a-1)}\frac{(c-1)!}{(b-1)!}. \end{aligned}$$

□

Théorème 22. *Sous le modèle d'Ewens pour les records dans $\mathfrak{S}_n$ de paramètre $\theta \in \mathbb{R}^{+*}$, nous posons $\mathcal{R}_n^{\bullet\circ}(a,b,c) = \{\sigma \in \mathfrak{S}_n \mid \forall i \in [a,b], \sigma(i)$ est un record et $\forall j \in [b+1,c], \sigma(j)$ est un $non-record\}$ (respectivement $\mathcal{R}_n^{\circ\bullet}(a,b,c) = \{\sigma \in \mathfrak{S}_n \mid \forall i \in [a,b], \sigma(i)$ est un $non-record$ et $\forall j \in [b+1,c], \sigma(j)$ est un record$\}$), l'ensemble des permutations telles que les éléments des positions de a à b inclus sont des records et les éléments des positions de $b+1$ à c inclus sont des non-records. Les identités suivantes sont vérifiées*

$$w(\mathcal{R}_n^{\bullet\circ}(a,b,c)) = \frac{\theta^{(n)}\theta^{(a-1)}\theta^{b-a+1}}{\theta^{(c)}}\frac{(c-1)!}{(b-1)!},$$

$$w(\mathcal{R}_n^{\circ\bullet}(a,b,c)) = \frac{\theta^{(n)}\theta^{(a-1)}\theta^{c-b}}{\theta^{(c)}}\frac{(b-1)!}{(a-2)!}.$$

Démonstration. Il suffit d'appliquer directement les lemmes 22 et 23 avec le Lemme 20. □

Remarque. Le Théorème 21 est un cas particulier du Théorème 22. En effet,

$$\mathbb{P}_n(\textit{record en position } i) = \frac{w(\mathcal{R}_n^{\bullet\circ}(i,i,i))}{w(\mathfrak{S}_n)} = \frac{w(\mathcal{R}_n^{\circ\bullet}(i,i-1,i))}{w(\mathfrak{S}_n)}.$$

Montées et descentes à une position fixée

Deux autres statistiques d'intérêt sont les montées et les descentes. Si une permutation est obtenue en ajoutant une légère perturbation à la permutation identité, l'intuition que nous avons est qu'il y aura un grand nombre de montées. Rappelons également que le nombre de descentes est également en relation avec le nombre de runs dans une séquence. Nous allons donc par la suite étudier deux ensembles. Le premier ensemble est celui des permutations de taille n qui ont une montée à la position $i \in [n]$, cet ensemble est défini de la façon suivante :

$$X_{i,n}^{\nearrow} = \{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i)\}.$$

Le deuxième ensemble que nous allons étudier est celui des permutations de taille n qui ont une descente à la position $i \in [n]$, soit

$$X_{i,n}^{\searrow} = \{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) > \sigma(i)\}.$$

Nous allons maintenant étudier le cas de l'ensemble $X_{i,n}^{\nearrow}$. Remarquons qu'il est possible de voir cet ensemble comme une union disjointe, nous avons

$$\begin{aligned} X_{i,n}^{\nearrow} = &\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \textit{ et } \sigma(i) \text{ est un record}\} \\ &\cup \{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \textit{ et } \sigma(i) \text{ n'est pas un record}\}. \end{aligned}$$

Pour que $\sigma(i)$ soit un record, il est nécessaire d'avoir $\sigma(i-1) < \sigma(i)$, il y a donc redondance sur le premier ensemble de cette union et nous avons donc

$$\begin{aligned} X_{i,n}^{\nearrow} = &\{\sigma \in \mathfrak{S}_n \,|\sigma(i) \text{ est un record}\} \\ &\cup \{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \textit{ et } \sigma(i) \text{ n'est pas un record}\}. \end{aligned}$$

Le premier ensemble se ramène donc à un cas que nous avons déjà étudié, puisqu'il s'agit de savoir si nous avons un record à la position i. Il nous reste donc plus qu'à étudier le deuxième ensemble de notre union à savoir

$$\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \textit{ et } \sigma(i) \text{ n'est pas un record}\}.$$

Lemme 24. *Pour tout entier $n \in \mathbb{N}^*$ et pour tout $i \in [n]$, l'ensemble $\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i)$ et $\sigma(i)$ n'est pas un record$\}$ est quotient-stable de degré $n-i$ et nous avons*

$$w_{n-i}^q(\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \textit{ et } \sigma(i) \textit{ n'est pas un record}\}) = \frac{(i-1)(i-2)\theta^{(i-2)}}{2}.$$

Démonstration. Soit pour tout $\tau \in \mathfrak{S}_{n-i \text{ in } n}$, une permutation $\rho \in \text{norm}(\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \text{ } et \text{ } \sigma(i) \text{ n'est pas un record}\}/\tau)$. Remarquons que comme $\sigma(i)$ n'est pas un record et que celui-ci est plus grand que $\sigma(i-1)$ alors $\sigma(i-1)$ ne peut pas être lui même un record. Cela implique que la valeur i doit être dans une position qui se trouve dans l'intervalle $[i-2]$ et donc $\rho(i-1)$ et $\rho(i)$ ont tous les deux une valeur prise dans l'intervalle $[i-1]$. Nous avons alors la relation suivante :

$$\begin{aligned}&\text{norm}(\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \text{ } et \text{ } \sigma(i) \text{ n'est pas un record}\}/\tau)\\ &= \bigcup_{\substack{\alpha,\beta\in[i-1]\\ \alpha<\beta}} \{\rho \in \mathfrak{S}_i \,|\rho(i-1) = \alpha \text{ } et \text{ } \rho(i) = \beta\}.\end{aligned}$$

Cet ensemble est donc indépendant de τ, et donc $\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \text{ } et \text{ } \sigma(i) \text{ n'est pas un record}\}$ est quotient-stable de degrés $n-i$ et nous avons

$$\begin{aligned}&w^q_{n-i}(\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \text{ } et \text{ } \sigma(i) \text{ n'est pas un record}\})\\ &= w_\theta\left(\bigcup_{\substack{\alpha,\beta\in[i-1]\\ \alpha<\beta}} \{\rho \in \mathfrak{S}_i \,|\rho(i-1) = \alpha \text{ } et \text{ } \rho(i) = \beta\}\right)\\ &= \sum_{\substack{\alpha,\beta\in[i-1]\\ \alpha<\beta}} w_\theta(\{\rho \in \mathfrak{S}_i \,|\rho(i-1) = \alpha \text{ } et \text{ } \rho(i) = \beta\}),\end{aligned}$$

car il s'agit d'une union disjointe. Il nous reste alors à déterminer, pour chaque $\alpha, \beta \in [i-1]$ avec $\alpha < \beta$, la valeur de $w_\theta(\{\rho \in \mathfrak{S}_i \,|\rho(i-1) = \alpha \text{ } et \text{ } \rho(i) = \beta\})$. Nous savons que $\rho(i-1)$ et $\rho(i)$ ne sont pas des records. Les éléments aux positions de l'intervalle $[i-2]$ peuvent être présents dans n'importe quel ordre et se comportent donc comme une permutation de taille $i-2$. En appliquant directement le Lemme 18, nous obtenons alors

$$w_\theta(\{\rho \in \mathfrak{S}_i \,|\rho(i-1) = \alpha \text{ } et \text{ } \rho(i) = \beta\}) = \theta^{(i-2)}.$$

Comme il y a $\binom{i-1}{2}$ façons de choisir α et β, nous pouvons conclure la preuve. □

Théorème 23. *Soit une permutation $\sigma \in \mathfrak{S}_n$ obtenue par la distribution d'Ewens généralisée au record pour un paramètre $\theta > 0$. Pour tout $i \in [2, n]$, la probabilité que $\sigma(i-1) < \sigma(i)$ est donnée par l'égalité*

$$\mathbb{P}_{\theta,n}(X^{\nearrow}_{i,n}) = \frac{2\theta^2 + (i-1+2\theta)(i-2)}{2(\theta+i-1)(\theta+i-2)}.$$

Démonstration. Il nous suffit de calculer la somme

$$w_\theta(\{\sigma \in \mathfrak{S}_n \,|\sigma(i) \text{ est un record}\}) + w_\theta(\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \text{ } et \text{ } \sigma(i) \text{ n'est pas un record}\}).$$

Nous savons d'après le Théorème 21 que

$$w_\theta(\{\sigma \in \mathfrak{S}_n \,|\sigma(i) \text{ est un record}\}) = \frac{\theta^{(n)}\theta}{\theta+i-1}.$$

En utilisant les Lemmes 24 et 20, nous obtenons

$$w_\theta(\{\sigma \in \mathfrak{S}_n \,|\sigma(i-1) < \sigma(i) \textit{ et } \sigma(i) \text{ n'est pas un record}\}) = \frac{\theta^{(n)}(i-1)(i-2)}{2(\theta+i-1)(\theta+i-2)}.$$

En normalisant la somme des poids de ces deux ensembles par $\theta^{(n)}$ nous pouvons conclure cette preuve. □

Maintenant que nous avons la probabilité qu'une permutation soit dans $X^{\nearrow}_{i,n}$, le cas de $X^{\searrow}_{i,n}$ est immédiat. En effet si il n'y a pas de montée à la position i, il y a forcément une descente à cette position. Ainsi les ensembles $X^{\nearrow}_{i,n}$ et $X^{\searrow}_{i,n}$ sont complémentaires.

Théorème 24. *Soit une permutation $\sigma \in \mathfrak{S}_n$ obtenue par la distribution d'Ewens généralisée au record pour un paramètre $\theta > 0$. Pour tout $i \in [2,n]$, la probabilité que $\sigma(i-1) > \sigma(i)$ est donnée par l'égalité*

$$\mathbb{P}_{\theta,n}(X^{\searrow}_{i,n}) = \frac{(i-1)(2\theta+i-2)}{2(\theta+i-1)(\theta+i-2)}.$$

Inversions

Une autre statistique classique qui a une influence directe sur l'algorithme de tri par insertion et du tri par bulle est le nombre d'inversions. Pour tout $n \in \mathbb{N}^*$ et tout $j \in [n]$, nous notons $inv_j(S)$ qui associe pour une séquence $S \in E^n$ pour un ensemble E ordonné, le nombre d'inversions de la forme (i,j) avec $i < j$. Plus formellement, nous avons

$$inv_j(S) = |\{i \in [j-1] | (i,j) \textit{ est une inversion dans } S\}|.$$

Nous posons pour tout $k \in [j-1]$, l'ensemble

$$\mathcal{I}_{n,j,k} = \{\sigma \in \mathfrak{S}_n \,|inv_j(\sigma) = k\}.$$

Lemme 25. *Pour tout entier $n \in \mathbb{N}^*$, pour tout $j \in [n]$, et pour tout $k \in [j-1]$, l'ensemble $\mathcal{I}_{n,j,k}$ est quotient-stable de degrés $n-j$ et nous avons*

$$w^q_{n-j}(\mathcal{I}_{n,j,k}) = \begin{cases} \theta\theta^{(j-1)} & (k=0) \\ \theta^{(j-1)} & (sinon) \end{cases}.$$

Démonstration. Pour tout $\sigma \in \mathcal{I}_{n,j,0}$, nous avons que $\sigma(j) > \sigma(i)$ pour tout $i \in [j-1]$. Autrement dit, $\sigma(j)$ est un record. Le cas où $k=0$ se ramène donc à regarder si nous avons un record à la position j qui est déjà donné dans le Lemme 21. Regardons donc uniquement le cas où $k > 0$. Pour tout $\tau \in \mathfrak{S}_{n-k \,\text{in}\, n}$, intéressons nous aux propriétés d'une permutation $\rho \in \text{norm}(\mathcal{I}_{n,j,k}/\tau)$. Remarquons que nous avons par conservation de l'ordre par l'opérateur norm que $inv_j(\rho) = k$. Il existe donc k positions dans l'intervalle $[j-1]$ dont les éléments

sont plus grands que $\rho(j)$. Cela n'est possible uniquement si et seulement si $\rho(j) = j - k$. Cette dernière affirmation nous permet d'obtenir l'égalité suivante :

$$Y_{n,j,k} = \mathrm{norm}(\mathcal{I}_{n,j,k}/\tau) = \{\rho \in \mathfrak{S}_j \,|\rho(j) = j - k\}.$$

Cette définition ne dépendant pas de τ, nous savons que $\mathcal{I}_{n,j,k}$ est quotient-stable de degrés $n - j$. Nous avons de plus,

$$w^q_{n-j}(\mathcal{I}_{n,j,k}) = w_\theta(Y_{n,j,k}).$$

En appliquant le Lemme 18, nous avons

$$w_\theta(\rho) = w_\theta(\rho(1)\dots\rho(j-1))w'_{\theta,j}(\rho(j)).$$

Or, comme $\rho(j) = j - k < j$ ne peut pas être un record, nous avons

$$w_\theta(\rho) = w_\theta(\rho(1)\dots\rho(j-1)).$$

Comme les éléments aux positions dans l'intervalle $[j-1]$ peuvent apparaitre dans n'importe quel ordre, en faisant la somme pour tous les $\rho \in Y_{n,j,k}$, nous avons alors

$$w_\theta(Y_{n,j,k}) = w_\theta(\mathfrak{S}_{j-1}) = \theta^{(j-1)},$$

ce qui nous permet de conclure. □

Théorème 25. *Pour tout $n \in \mathbb{N}^*$, soit une permutation $\sigma \in \mathfrak{S}_n$ obtenue par une distribution d'Ewens généralisée pour les records de paramètre $\theta \in \mathbb{R}^{+*}$. Pour tout $j \in [n]$ et tout $k \in [j-1]$, la probabilité d'avoir $inv_j(\sigma) = k$ est donnée par la relation*

$$\mathbb{P}_{\theta,n}(\mathcal{I}_{n,j,k}) = \begin{cases} \frac{\theta}{\theta+j-1} & (k=0) \\ \frac{1}{\theta+j-1} & (sinon) \end{cases}.$$

Démonstration. La preuve se fait en appliquant directement le Lemme 20 avec le Lemme 25. □

Valeur du premier élément

Nous allons maintenant nous intéresser à une autre statistique qui n'a pas vraiment d'intérêt pour les algorithmes de tri, mais qui nécessite un raisonnement pour lequel nous ne pouvons pas utiliser le Lemme 20. Il s'agit de l'étude de la valeur du premier élément. Pour la suite, pour tout $n \in \mathbb{N}^*$ nous posons l'ensemble

$$\mathcal{F}_{n,k} = \{\sigma \in \mathfrak{S}_n \mid \sigma(1) \geq k\}$$

Nous ne pouvons pas utiliser le Lemme 20 car cet ensemble n'est pas quotient-stable pour tous les degrés possibles. Pour s'en convaincre, choisissons un suffixe $\tau \in \mathfrak{S}_{i\,\mathrm{in}\,n}$ pour un certain $i \in [n]$. Il existe une permutation $\sigma = \pi \cdot \tau \in \mathcal{F}_{n,k}$ tel que $\mathbf{rec}(\sigma) = \max(\tau) - k + 1$. Cette permutation suffit à montrer que le poids de $w_\theta(\mathrm{norm}(\mathcal{F}_{n,k}/\tau))$ dépend de τ et donc $\mathcal{F}_{n,k}$ n'est pas quotient-stable de degrés i.

Notre but est de déterminer $w_\theta(\mathcal{F}_{n,k})$. Pour résoudre ce problème, nous allons dans un premier temps poser la fonction $r_{n,k} : \mathfrak{S}_n \to \mathfrak{S}_{n-k+1\,\mathrm{in}\,n}$. Pour toute

permutation $\sigma \in \mathfrak{S}_n$, nous associons la sous-séquence $r_{n,k}(\sigma)$ qui retire tous les éléments plus petits strictement à k. Pour tout $\sigma \in \mathcal{F}_{n,k}$ comme $\sigma(1) \geq k$, nous avons de toute évidence $\mathbf{rec}(\sigma) = \mathbf{rec}(r_{n,k}(\sigma))$ et donc

$$w_\theta(\sigma) = w_\theta(r_{n,k}(\sigma)). \tag{6.12}$$

Théorème 26. *Pour tout $n \in \mathbb{N}^*$, soit $\sigma \in \mathfrak{S}_n$ une permutation obtenue par la distribution d'Ewens généralisée pour les records de paramètre $\theta \in \mathbb{R}^{+*}$. La probabilité d'avoir une permutation $\sigma \in \mathfrak{S}_n$ tel que $\sigma(1) \geq k$ pour $k \in [0, n-1]$ est donnée par la relation*

$$\mathbb{P}_{\theta,n}(\mathcal{F}_{n,k}) = \frac{(n-1)!\theta^{(n-k+1)}}{(n-k)!\theta^{(n)}}.$$

Démonstration. Nous allons déterminer la quantité $w_\theta(\mathcal{F}_{n,k})$. D'après les définitions, nous avons l'égalité suivante :

$$w_\theta(\mathcal{F}_{n,k}) = \sum_{\sigma \in \mathcal{F}_{n,k}} w_\theta(\mathrm{norm}(r_{n,k}(\sigma))).$$

Pour tout $\rho \in \mathfrak{S}_{n-k+1}$, nous posons

$$Q_\rho = \{\sigma \in \mathfrak{S}_n \,|\, \mathrm{norm}(r_{n,k}(\sigma))) = \rho\}.$$

Nous avons alors la relation

$$w_\theta(\mathcal{F}_{n,k}) = \sum_{\rho \in \mathfrak{S}_{n-k+1}} |Q_\rho| w_\theta(\rho)$$

qui est satisfaite. Pour construire une permutation $\sigma \in Q_\rho$, il suffit de prendre une permutation $\phi \in \mathfrak{S}_{k-1}$ et de faire un mélange entre ϕ et $\rho(2)\cdots\rho(n-k+1)$. Le premier élément de σ est par définition $\rho(1)$. D'un point de vue ensembliste, nous avons alors

$$Q_\rho = \rho(1) \cdot \left(\bigcup_{\phi \in \mathfrak{S}_{k-1}} \phi \sqcup\!\sqcup \rho(2)\cdots\rho(n-k+1) \right).$$

Ainsi nous avons

$$\begin{aligned} |Q_\rho| &= \sum_{\phi \in \mathfrak{S}_{k-1}} |\phi \sqcup\!\sqcup \rho(2)\cdots\rho(n-k+1)| \\ &= \sum_{\phi \in \mathfrak{S}_{k-1}} \binom{n-1}{k-1} \\ &= \frac{(n-1)!}{(n-k)!}. \end{aligned}$$

Finalement, nous avons les égalités suivantes qui sont vérifiées :

$$\begin{aligned} w_\theta(\mathcal{F}_{n,k}) &= \sum_{\rho \in \mathfrak{S}_{n-k+1}} |Q_{rho}| w_\theta(\rho) \\ &= \sum_{\rho \in \mathfrak{S}_{n-k+1}} \frac{(n-1)!}{(n-k)!} w_\theta(\rho) \\ &= \frac{(n-1)!}{(n-k)!} w_\theta(\mathfrak{S}_{n-k+1}) \\ &= \frac{(n-1)!}{(n-k)!} \theta^{(n-k+1)}. \end{aligned}$$

En divisant par $\theta^{(n)}$, nous obtenons le résultat annoncé. □

Posons maintenant l'ensemble

$$\mathcal{G}_{n,k} = \{\sigma \in \mathfrak{S}_n \mid \sigma(1) = k\}$$

Nous avons alors un corollaire à ce théorème.

Corollaire 1. *Pour tout $n \in \mathbb{N}^*$, soit $\sigma \in \mathfrak{S}_n$ une permutation obtenue par la distribution d'Ewens généralisée pour les records de paramètre $\theta \in \mathbb{R}^{+*}$. La probabilité d'avoir une permutation $\sigma \in \mathfrak{S}_n$ tel que $\sigma(1) = k$ pour $k \in [0, n-1]$ est donnée par la relation*

$$\mathbb{P}_{\theta,n}(\mathcal{G}_{n,k}) = \frac{(n-1)!\theta^{(n-k)}\theta}{(n-k)!\theta^{(n)}}.$$

Démonstration. Remarquons dans un premier temps que nous avons la relation suivante :

$$\mathcal{F}_{n,k} = \mathcal{G}_{n,k} \cup \mathcal{F}_{n,k+1}.$$

Cette relation implique que nous avons l'égalité

$$\mathbb{P}_{\theta,n}(\mathcal{F}_{n,k}) = \mathbb{P}_{\theta,n}(\mathcal{G}_{n,k}) + \mathbb{P}_{\theta,n}(\mathcal{F}_{n,k+1}).$$

En remplaçant $\mathbb{P}_{\theta,n}(\mathcal{F}_{n,k})$ par l'égalité donnée par le Théorème 26, nous obtenons le résultat annoncé. □

6.5.3 Espérances et asymptotiques des différentes statistiques

Pour finir cette étude sur l'influence de la distribution d'Ewens pour les records sur d'autres statistiques, nous allons nous intéresser à l'espérance de ces statistiques et leurs comportements asymptotiques pour plusieurs cas. Les théorèmes que nous venons de démontrer précédemment nous permettent de déterminer facilement ces espérances.

Dans les expressions des probabilités de ces statistiques, nous voyons très fréquemment apparaitre des fractions de la forme $\frac{1}{x+i}$ pour un certain paramètre réel x et un entier i. Nous allons faire la somme sur tous les i possibles de termes de cette forme. Il se trouve que la somme de ces termes est une fonction connue

que l'on nomme la fonction digamma. Cette fonction est définie par la relation

$$\Psi(x) = \frac{\Gamma'(x)}{\Gamma(x)},$$

où Γ est la fonction gamma définie par la relation

$$\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} \mathrm{d}t.$$

La fonction Ψ vérifie également la propriété suivante que nous utilisons beaucoup par la suite,

$$\sum_{i=0}^{n-1} \frac{1}{x+i} = \Psi(x+n) - \Psi(x).$$

Nous notons, par ailleurs, la fonction

$$\Delta(x, y) = \Psi(x+y) - \Psi(x),$$

et nous avons donc

$$\sum_{i=0}^{n-1} \frac{1}{x+i} = \Delta(x, n). \tag{6.13}$$

Le comportement asymptotique de la fonction Ψ, quand x est grand, est également connu et nous avons pour les trois premiers termes,

$$\Psi(x) = \log(x) - \frac{1}{2x} - \frac{1}{12x^2} + o\left(\frac{1}{x^3}\right). \tag{6.14}$$

Nous allons étudier le comportement des autres statistiques pour trois régimes particuliers, un régime sous linéaire en regardant les cas où θ est une constante ou bien une expression de la forme $\theta = n^\epsilon$ pour une certaine valeur réelle $\epsilon \in]0, 1[$, un régime linéaire où θ est de la forme λn pour une certaine valeur réelle $\lambda > 0$, et enfin un régime sur-linéaire où θ est de la forme $\theta = n^\delta$ pour une certaine valeur réelle $\delta > 1$.

Nombre moyen de records : Nous commençons une fois de plus à regarder en moyenne combien de records nous avons pour cette distribution.

Corollaire 2. *Sous la distribution d'Ewens généralisée aux records sur $\mathfrak{S}_n$, avec $n \in \mathbb{N}^*$ et de paramètre $\theta \in \mathbb{R}^{+*}$, l'espérance du nombre de records des permutations est donnée par la relation suivante :*

$$\mathbb{E}_{\theta,n}[\mathbf{rec}] = \theta \Delta(\theta, n).$$

Démonstration. Par la définition de l'espérance, nous avons

$$\mathbb{E}_{\theta,n}[\mathbf{rec}] = \sum_{i=1}^{n} \mathbb{P}_{\theta,n}(\sigma(i) \text{ est un record}).$$

Or nous savons d'après le Théorème 21 que

$$\mathbb{P}_{\theta,n}(\sigma(i) \text{ est un record}) = \frac{\theta}{\theta + i - 1}$$

En utilisant l'Equation (6.13) nous obtenons alors le résultat. □

Nous allons maintenant à partir de cette expression de l'espérance du nombre de records déterminer son comportement asymptotique, pour n grand, sur différents cas.

Le premier cas que nous allons regarder est le cas où θ est une constante. Nous avons alors d'après le Corollaire 2 et par l'Equation (6.14), nous avons

$$\begin{aligned}
\mathbb{E}_{\theta,n}[\mathbf{rec}] &= \theta\left(\Psi(\theta+n) - \Psi(\theta)\right) \\
&= \theta\left(\log(\theta+n) + o(\log(\theta+n)) - \Psi(\theta)\right) \\
&= \theta\left(\log\left(n\left(1+\frac{\theta}{n}\right)\right) + o(\log(n))\right) \\
&= \theta\left(\log(n) + \log\left(1+\frac{\theta}{n}\right) + o(\log(n))\right) \\
&= \theta\log(n) + o(\log(n)).
\end{aligned}$$

Remarque. Nous observons en particulier que pour $\theta = 1$, le cas uniforme, nous avons le résultat connu qui est que le nombre de records est asymptotiquement équivalent à $\log(n)$. Le paramètre θ agit donc ici simplement comme une constante multiplicative du cas uniforme.

Nous allons maintenant regarder des cas où nous faisons dépendre θ de n. Le premier cas est un cas où nous faisons légèrement dépendre θ de n. Nous avons choisi le cas où pour un réel $\epsilon \in]0,1[$, $\theta(n) = n^\epsilon$. Dans ce cas nous avons alors

$$\begin{aligned}
\mathbb{E}_{\theta,n}[\mathbf{rec}] &= \theta\left(\Psi(\theta+n) - \Psi(\theta)\right) \\
&= n^\epsilon\left(\Psi(n^\epsilon+n) - \Psi(n^\epsilon)\right) \\
&= n^\epsilon\left(\log(n^\epsilon+n) + o(1)\right) - \log(n^\epsilon) - o(1)) \\
&= n^\epsilon\left(\log\left(\frac{n^\epsilon+n}{n^\epsilon}\right) + o(1)\right) \\
&= n^\epsilon\left(\log(n^{1-\epsilon}(n^{\epsilon-1}+1)) + o(1)\right) \\
&= n^\epsilon\left(\log(n^{1-\epsilon})\log(n^{\epsilon-1}+1) + o(1)\right) \\
&= n^\epsilon\left(\log(n^{1-\epsilon}) + o(1)\right) \\
&= (1-\epsilon)n^\epsilon\log(n) + o(n^\epsilon\log(n)).
\end{aligned}$$

Nous avons pour ce cas que cette quantité est supérieure en ordre de grandeur à $\log n$.

Nous allons maintenant prendre une dépendance linéaire, nous posons pour un $\lambda > 0$, $\theta(n) = \lambda n$. Nous avons alors en développant jusqu'au deuxième terme,

$$\begin{aligned}
\mathbb{E}_{\theta,n}[\mathbf{rec}] &= \theta\left(\Psi(\theta+n)-\Psi(\theta)\right)\\
&= \lambda n\Bigg(\log(\lambda n+n)-\frac{1}{2(\lambda n+n)}+o\left(\frac{1}{\lambda n+n}\right)\\
&\qquad -\log(\lambda n)+\frac{1}{2(\lambda n)}+o\left(\frac{1}{\lambda n}\right)\Bigg)\\
&= \lambda n\left(\log\left(\frac{\lambda n+n}{\lambda n}\right)+\frac{\lambda n+n-\lambda n}{2(\lambda n+n)(\lambda n)}+o\left(\frac{1}{n}\right)\right)\\
&= \lambda n\left(\log\left(\frac{\lambda+1}{\lambda}\right)+o(1)\right)\\
&= \lambda\log\left(\frac{\lambda+1}{\lambda}\right)n+o(n).
\end{aligned}$$

Avec une dépendance linéaire à la taille de la permutation, nous avons alors un nombre de record qui est lui même linéaire.

Le dernier cas que nous allons regarder est une dépendance très forte, où pour tout $\delta > 1$, nous avons $\theta(n) = n^\delta$. Avec un développement jusqu'au deuxième terme, nous avons

$$\begin{aligned}
\mathbb{E}_{\theta,n}[\mathbf{rec}] &= \theta\left(\Psi(\theta+n)-\Psi(\theta)\right)\\
&= n^\delta\Bigg(\log(n^\delta+n)-\frac{1}{2(n^\delta+n)}+o\left(\frac{1}{n^\delta+n}\right)\\
&\qquad -\log(n^\delta)+\frac{1}{2(n^\delta)}+o\left(\frac{1}{n^\delta}\right)\Bigg)\\
&= n^\delta\left(\log\left(\frac{n^\delta+n}{n^\delta}\right)+o\left(\frac{1}{n^{\delta-1}}\right)\right)\\
&= n^\delta\left(\log\left(1+\frac{n}{n^\delta}\right)+o\left(\frac{1}{n^{\delta-1}}\right)\right)\\
&= n^\delta\left(\log\left(1+\frac{1}{n^{\delta-1}}\right)+o\left(\frac{1}{n^{\delta-1}}\right)\right).
\end{aligned}$$

Comme $\frac{1}{n^{\delta-1}}$ est au voisinage de 0 quand n est grand, nous pouvons développer le logarithme jusqu'au premier terme et nous obtenons

$$\begin{aligned}
\mathbb{E}_{\theta,n}[\mathbf{rec}] &= n^\delta\left(\frac{1}{n^{\delta-1}}-o\left(\frac{1}{n^{\delta-1}}\right)\right)\\
&= n-o(n).
\end{aligned}$$

Quelle que soit la valeur choisie pour δ, nous poussons la distribution à avoir un nombre de records qui est très proche de la taille de la permutation. Ce cas va donc générer des permutations proches de la permutation identité avec un bruit qui est de plus en plus négligeable lorsque δ croît.

Nombre moyen de descentes :

Corollaire 3. *Sous la distribution d'Ewens généralisée aux records sur $\mathfrak{S}_n$, avec $n \in \mathbb{N}^*$ et de paramètre $\theta \in \mathbb{R}^{+*}$, l'espérance du nombre de descentes des permutations est donnée par la relation suivante :*

$$\mathbb{E}_{\theta,n}[desc] = \frac{n(n-1)}{2(\theta+n-1)}.$$

Démonstration. Rappelons que nous avons d'après le Théorème 24 la relation

$$\mathbb{P}_n(\sigma(i-1) > \sigma(i)) = \frac{(i-1)(2\theta+i-2)}{2(\theta+i-1)(\theta+i-2)}.$$

Nous allons décomposer en fractions simples le double de cette probabilité. Nous avons alors

$$2\mathbb{P}_n(\sigma(i-1) > \sigma(i)) = 1 + \frac{\theta(\theta-1)}{\theta+i-1} - \frac{\theta(\theta-1)}{\theta+i-2}.$$

En faisant la somme de ces probabilités pour i allant de 2 jusqu'à n, nous obtenons alors le double de l'espérance, ce qui nous donne donc

$$\begin{aligned}
2\mathbb{E}_{\theta,n}[desc] &= \sum_{i=2}^{n} \left(1 + \frac{\theta(\theta-1)}{\theta+i-1} - \frac{\theta(\theta-1)}{\theta+i-2}\right) \\
&= n - 1 + \theta(\theta-1) \sum_{i=2}^{n} \left(\frac{1}{\theta+i-1} - \frac{1}{\theta+i-2}\right) \\
&= n - 1 + \theta(\theta-1)\Bigg(\frac{1}{\theta+1} - \frac{1}{\theta} + \frac{1}{\theta+2} - \frac{1}{\theta+1} + \dots \\
&\quad + \frac{1}{\theta+n-2} - \frac{1}{\theta+n-3} + \frac{1}{\theta+n-1} - \frac{1}{\theta+n-2}\Bigg).
\end{aligned}$$

Les termes vont donc mutuellement s'annuler exceptés les termes $-\frac{1}{\theta}$ et $\frac{1}{\theta+n-1}$. Nous avons alors après simplification

$$2\mathbb{E}_{\theta,n}[desc] = n - 1 + \theta(\theta-1)\left(\frac{1}{\theta+n-1} - \frac{1}{\theta}\right).$$

Il suffit alors de mettre au même dénominateur et de diviser cette quantité par 2 pour obtenir le résultat annoncé. □

L'analyse de cette quantité quand n est grand pour les différents cas étudiés précédemment est simple ici puisqu'il s'agit d'une fonction rationnelle en n. Il suffit donc de ne regarder que la division entre les termes dominants du numérateur et du dénominateur.

Le premier cas que nous avons étudié est le cas où θ est une constante. Dans ce cas nous avons directement

$$\mathbb{E}_{\theta,n}[desc] = \frac{n(n-1)}{2(\theta+n-1)} \sim \frac{n}{2}.$$

Nous obtenons la même approximation pour le cas où $\theta(n) = n^\epsilon$ pour $\epsilon \in]0,1[$. Dans ce cas $2n$ reste le terme dominant du dénominateur. Pour tous ces

cas, nous n'arrivons donc pas à forcer la distribution à faire mieux que le cas uniforme.

Le cas où $\theta(n) = \lambda n$ est le premier cas qui introduit un biais sur le nombre de descentes. Nous avons dans ce cas

$$\mathbb{E}_{\theta,n}[desc] = \frac{n(n-1)}{2(\lambda n + n - 1)} \sim \frac{n}{2(\lambda + 1)}.$$

Le nombre de descentes reste cependant une fonction linéaire de la taille de la permutation.

Sans grande surprise, le cas où $\theta(n) = n^\delta$ avec $\delta > 1$ peut introduire un biais important. Nous avons dans ce cas

$$\mathbb{E}_{\theta,n}[desc] = \frac{n(n-1)}{2(n^\delta + n - 1)} \sim \frac{n^{2-\delta}}{2}.$$

Nous avons donc non seulement une fonction qui est sous linéaire, mais qui en particulier peut donner un nombre de descentes négligeable quand $\delta > 2$.

Valeur moyenne du premier élément :

Corollaire 4. *Sous la distribution d'Ewens généralisée aux records sur $\mathfrak{S}_n$, avec $n \in \mathbb{N}^*$ et de paramètre $\theta \in \mathbb{R}^{+*}$, l'espérance du premier élément est donnée par*

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{\theta + n}{\theta + 1}.$$

Démonstration. Par définition, cette espérance nous est donnée par l'égalité

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \sum_{k=1}^{n} \mathbb{P}_{\theta,n}(\sigma(1) \geq k).$$

D'après le Théorème 26, nous avons

$$\mathbb{P}_{\theta,n}(\sigma(1) \geq k) = \frac{(n-1)!\theta^{(n-k+1)}}{(n-k)!\theta^{(n)}},$$

et ainsi

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{(n-1)!}{\theta^{(n)}} \sum_{k=1}^{n} \frac{\theta^{(n-k+1)}}{(n-k)!}.$$

La somme est proche de l'équation (6.11) de la Proposition 25. En faisant un premier changement de variable par $i = n - k$, nous avons

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{(n-1)!}{\theta^{(n)}} \sum_{i=0}^{n-1} \frac{\theta^{(i+1)}}{i!}.$$

Comme $\theta^{(i+1)} = \theta(\theta+1)^{(i)}$, nous obtenons alors

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{(n-1)!}{\theta^{(n)}} \left(\sum_{i=0}^{n-2} \frac{\theta(\theta+1)^{(i)}}{i!} + \frac{\theta^{(n)}}{(n-1)!} \right).$$

Nous utilisons maintenant l'équation (6.11) et nous obtenons l'égalité suivante :

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{(n-1)!}{\theta^{(n)}} \frac{\theta(\theta+1)^{(n-1)}}{(\theta+1)(n-2)!} + 1.$$

Après simplification sur le terme de gauche nous avons

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{(n-1)}{\theta+1} + 1,$$

qui une fois mis au même dénominateur nous donne le résultat attendu. □

Nous avons, comme pour le cas précédent, une fonction rationnelle. Regardons ce qui se passe pour n grand pour nos différents cas. Si θ est fixe et ne dépend pas de n, nous avons

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{\theta+n}{\theta+1} \sim \frac{n}{\theta+1}.$$

En particulier, si $\theta = 1$ nous avons $\frac{n}{2}$ qui est ce que nous nous attendons à obtenir pour le cas uniforme.

Si $\theta(n) = n^\epsilon$ pour $\epsilon \in]0,1[$ alors

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{n^\epsilon+n}{n^\epsilon+1} \sim n^{1-\epsilon}.$$

Si $\theta(n) = \lambda n$ pour $\lambda > 0$ alors

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{\lambda n+n}{\lambda n+1} \sim \frac{\lambda+1}{\lambda}.$$

Nous avons donc pour ce cas réussi à fixer la valeur du premier élément à une constante.

Si $\theta(n) = n^\delta$ pour $\delta > 1$, alors

$$\mathbb{E}_{\theta,n}[\sigma(1)] = \frac{n^\delta+n}{n^\delta+1} \sim 1.$$

Encore une fois ce cas force la permutation obtenue sous cette distribution à être proche de l'identité et nous fixons donc $\sigma(1) = 1$.

Nombre moyen d'inversions :

Corollaire 5. *Sous la distribution d'Ewens généralisée aux records sur $\mathfrak{S}_n$, avec $n \in \mathbb{N}^*$ et de paramètre $\theta \in \mathbb{R}^{+*}$, l'espérance du nombre d'inversions est donnée par*

$$\mathbb{E}_{\theta,n}[Inv] = \frac{n(n+1-2\theta)}{4} + \frac{\theta(\theta-1)}{2}\Delta(\theta,n).$$

Démonstration. Rappelons que le Théorème 25 nous donne

$$\mathbb{P}_{\theta,n}(inv_j = k) = \begin{cases} \frac{\theta}{\theta+j-1} & (k=0) \\ \frac{1}{\theta+j-1} & (sinon) \end{cases}.$$

L'espérance du nombre d'inversions est donnée par la somme

$$\mathbb{E}_{\theta,n}[Inv] = \sum_{j=2}^{n} \sum_{k=0}^{j-1} k \mathbb{P}_{\theta,n}(inv_j = k).$$

Comme pour $k = 0$ le terme $k\mathbb{P}_{\theta,n}(inv_j = k) = 0$, nous avons

$$\begin{aligned}
\mathbb{E}_{\theta,n}[Inv] &= \sum_{j=2}^{n} \sum_{k=1}^{j-1} \frac{k}{\theta + j - 1} \\
&= \sum_{j=2}^{n} \frac{1}{\theta + j - 1} \sum_{k=1}^{j-1} k \\
&= \sum_{j=2}^{n} \frac{(j-1)j}{2(\theta + j - 1)}
\end{aligned}$$

Pour continuer, il nous faut décomposer $\frac{(j-1)j}{2(\theta+j-1)}$. Nous avons

$$\begin{aligned}
\frac{(j-1)j}{2(\theta + j - 1)} &= \frac{(\theta + j - 1)j - \theta j}{2(\theta + j - 1)} \\
&= \frac{j}{2} - \frac{\theta j}{2(\theta + j - 1)} \\
&= \frac{j}{2} - \frac{\theta(\theta - 1 + j) - \theta(\theta - 1)}{2(\theta + j - 1)} \\
&= \frac{j}{2} - \frac{\theta}{2} + \frac{\theta(\theta - 1)}{2(\theta + j - 1)}.
\end{aligned}$$

Remarquons que ce terme s'annule pour $j = 1$, nous avons alors en remplaçant,

$$\mathbb{E}_{\theta,n}[Inv] = \sum_{j=1}^{n} \frac{j}{2} - \frac{\theta}{2} + \frac{\theta(\theta - 1)}{2(\theta + j - 1)}.$$

Avec le changement de variable $i = j - 1$, nous avons

$$\mathbb{E}_{\theta,n}[Inv] = \frac{n(n+1)}{4} - \frac{\theta n}{2} + \sum_{i=0}^{n-1} \frac{\theta(\theta - 1)}{2(\theta + i)}$$

Nous utilisons l'équation (6.13) pour obtenir le résultat annoncé. □

Pour l'analyse asymptotique, remarquons que pour les cas où θ est fixe, ou bien $\theta = n^{\epsilon}$ pour $\epsilon \in]0, 1[$, nous avons

$$\lim_{n \to +\infty} \frac{\frac{\theta(\theta-1)}{2} \Delta(\theta, n)}{\frac{n(n+1-2\theta)}{4}} = 0.$$

Nous avons alors pour ces cas

$$\mathbb{E}_{\theta,n}[Inv] = \frac{n(n+1-2\theta)}{4} + \frac{\theta(\theta-1)}{2} \Delta(\theta, n-1) \sim \frac{n(n+1-2\theta)}{4} \sim \frac{n^2}{4},$$

qui est le résultat attendu pour le cas uniforme. Pour $\theta(n) = \lambda(n)$, nous avons $\theta(\theta - 1)$ qui est du même ordre de grandeur que le premier terme de l'expression de cette espérance et il nous faut donc développer le $\Delta(\theta, n)$ à l'aide de l'équation (6.14). Développons donc le deuxième terme pour commencer.

Nous avons,

$$\begin{aligned}
\frac{\theta(\theta-1)}{2}\Delta(\theta,n) &= \frac{\lambda n(\lambda n-1)}{2}\left(\log(\lambda n+n) - \frac{1}{2(\lambda n+n)} + o(n^{-1}) - \log(\lambda n) + \frac{1}{2\lambda n} - o(n^{-1})\right) \\
&= \frac{\lambda n(\lambda n-1)}{2}\left(\log\left(\frac{(\lambda+1)n}{\lambda n}\right) + \frac{2\lambda n + 2n - 2\lambda n}{4(\lambda n+n)(\lambda n)} + o(n^{-1})\right) \\
&\sim \frac{\lambda^2 \log\left(\frac{\lambda+1}{\lambda}\right)}{2} n^2.
\end{aligned}$$

Finalement, nous avons

$$\begin{aligned}
\mathbb{E}_{\theta,n}[Inv] &= \frac{n(n+1-2\lambda n)}{4} + \frac{\lambda n(\lambda n)}{2}\Delta(\lambda n, n) \\
&\sim \frac{1-2\lambda}{4} n^2 + \frac{\lambda^2 \log\left(\frac{\lambda+1}{\lambda}\right)}{2} n^2 \\
&\sim c_\lambda \frac{n^2}{4},
\end{aligned}$$

avec

$$c_\lambda = 1 - 2\lambda + 2\lambda^2 \log\left(\frac{\lambda+1}{\lambda}\right).$$

Intéressons nous maintenant au cas où $\theta(n) = n^\delta$ avec $\delta > 1$. Nous avons plusieurs développements à effectuer pour ce cas. Commençons par développer la fonction Δ. Nous avons alors

$$\begin{aligned}
\frac{\theta(\theta-1)}{2}\Delta(\theta,n) &= \frac{n^\delta(n^\delta-1)}{2}\Bigg(\log(n^\delta+n) - \frac{1}{2(n^\delta+n)} - \frac{1}{12(n^\delta+n)^2} + o(n^{-3\delta}) \\
&\qquad - \log(n^\delta) + \frac{1}{2n^\delta} + \frac{1}{12n^{2\delta}} - o(n^{-3\delta})\Bigg) \\
&\sim \frac{n^{2\delta}}{2}\left(\log\left(\frac{n^\delta+n}{n^\delta}\right) - \frac{6n^\delta+6n+1}{12(n^\delta+n)^2} + \frac{6n^\delta+1}{12n^{2\delta}} + o(n^{-3\delta})\right) \\
&= \frac{n^{2\delta}}{2}\left(\log\left(\frac{n^\delta+n}{n^\delta}\right) - \frac{n^{1-2\delta}}{2} - \frac{1}{12(n^\delta+n)^2} + \frac{1}{12n^{2\delta}} + o(n^{-3\delta})\right) \\
&= \frac{n^{2\delta}}{2}\left(\log\left(\frac{n^\delta+n}{n^\delta}\right) - \frac{n^{1-2\delta}}{2} + \frac{n^{2\delta}+2n^{\delta+1}+n^2-n^{2\delta}}{12n^{2\delta}(n^\delta+n)^2} + o(n^{-3\delta})\right) \\
&= \frac{n^{2\delta}}{2}\left(\log(1+n^{1-\delta}) - \frac{n^{1-2\delta}}{2} + o(n^{3(1-\delta)})\right).
\end{aligned}$$

Nous développons maintenant le logarithme jusqu'au troisième terme et nous obtenons

$$\frac{\theta(\theta-1)}{2}\Delta(\theta,n) \sim \frac{n^{2\delta}}{2}\left(n^{1-\delta} - \frac{n^{2(1-\delta)}}{2} + \frac{n^{3(1-\delta)}}{3} - \frac{n^{1-2\delta}}{2} + o(n^{3(1-\delta)})\right).$$

Nous avons finalement,

$$\begin{aligned}\mathbb{E}_{\theta,n}[Inv] &= \frac{n(n+1-2\theta)}{4} + \frac{\theta(\theta-1)}{2}\Delta(\theta,n) \sim \frac{n^2}{4} + \frac{n}{4} - \frac{n^{1+\delta}}{2} \\ &\quad + \frac{n^{2\delta}}{2}\left(n^{1-\delta} - \frac{n^{2(1-\delta)}}{2} + \frac{n^{3(1-\delta)}}{3} - \frac{n^{1-2\delta}}{2} + o(n^{3(1-\delta)})\right) \\ &= \frac{n^2}{4} + \frac{n}{4} - \frac{n^{1+\delta}}{2} + \frac{n^{1+\delta}}{2} - \frac{n^2}{4} + \frac{n^{3-\delta}}{6} - \frac{n}{4} + o(n^{3-\delta}) \\ &= \frac{n^{3-\delta}}{6} + o(n^{3-\delta}).\end{aligned}$$

Nous arrivons à obtenir un résultat sous-quadratique pour ce choix de θ. Remarquons cependant que cette mesure nous montre qu'il y a un bruit qui reste relativement important sur les permutations que nous avons considérées proches de l'identité jusqu'à présent en regardant les autres statistiques. Dans la permutation identité, nous nous attendons à n'avoir aucune inversion. Or, il se trouve que tant que $\delta \leq 3$, nous avons $\mathbb{E}_{\theta,n}[Inv]$ qui ne tend pas vers 0. Cette mesure nous montre donc qu'il reste un bruit relativement important pour $\delta \in]1, 3]$.

Remarque. Nous avons vu lors du Chapitre 4 que le tri par insertion, que nous avons donné dans l'Algorithme 3, effectue un nombre de comparaisons qui est égal au nombre d'inversion de la séquence à trier. De ce fait, les comportements asymptotiques que nous avons analysés nous donne la complexité dans le cas moyen du nombre de comparaisons de cet algorithme. Nous constatons alors qu'en particulier l'algorithme a un comportement qui reste quadratique excepté pour le dernier cas où $\theta(n) = n^\delta$ avec $\delta > 1$. Il nous faut donc faire dépendre très fortement θ de n pour avoir l'espoir d'avoir des performances intéressantes du tri par insertion en moyenne. Nous pouvons résumer cette remarque en un nouveau corollaire.

Corollaire 6. *Sous la distribution d'Ewens pour les records dans $\mathfrak{S}_n$ de paramètre $\theta \in \mathbb{R}^{+*}$, si $\theta = \mathcal{O}(n)$, la complexité en espérance du tri par insertion en comparaisons est en $\Theta(n^2)$. Si $\theta = n^\delta$ avec $\delta \in 1 < \delta < 2$, alors elle est en $\Theta(n^{3-\delta})$. Enfin pour $\theta = \Omega(n^2)$, elle est en $\Theta(n)$.*

Démonstration. Pour trier $\sigma \in \mathfrak{S}_n$, l'Algorithme 3 doit effectuer $\max(n, inv(\sigma)$ comparaisons. L'espérance du nombre de comparaisons est alors donnée par la somme

$$\sum_{\sigma \in \mathfrak{S}_n} \mathbb{P}_{\theta,n}(\sigma) \max(n, inv(\sigma)).$$

Comme

$$n \leq \max(n, inv(\sigma)) < n + inv(\sigma),$$

nous avons

$$n \leq \sum_{\sigma \in \mathfrak{S}_n} \mathbb{P}_{\theta,n}(\sigma) \max(n, inv(\sigma)) < n + \mathbb{E}_{\theta,n}[Inv]$$

Nous connaissons le comportement asymptotique de $\mathbb{E}_{\theta,n}[Inv]$ à partir de la formule obtenue par le Corollaire 5 ce qui nous permet de conclure directement. □

Nous avons résumé les comportements asymptotiques pour les statistiques étudiées sur ces différents cas dans la Table 6.1.

	$\theta = 1$ (uniforme)	$theta > 0$ θ constant	$\theta := n^\epsilon$, $0 < \epsilon < 1$	$\theta := \lambda n$, $\lambda > 0$	$\theta := n^\delta$ $\delta > 1$
$\mathbb{E}_n[rec]$	$\log n$	$\theta \cdot \log n$	$(1-\epsilon) \cdot n^\epsilon \log n$	$\lambda \log(1 + 1/\lambda) \cdot n$	n
$\mathbb{E}_n[desc]$	$n/2$	$n/2$	$n/2$	$n/2(\lambda + 1)$	$n^{2-\delta}/2$
$\mathbb{E}_n[\sigma(1)]$	$n/2$	$n/(\theta + 1)$	$n^{1-\epsilon}$	$(\lambda + 1)/\lambda$	1
$\mathbb{E}_n[inv]$	$n^2/4$	$n^2/4$	$n^2/4$	$n^2/4 \cdot f(\lambda)$	$n^{3-\delta}/6$

TABLE 6.1 – Comportement asymptotique pour certaines statistiques sous le modèle d'Ewens dans $\mathfrak{S}_n$ pour les records. La fonction f est définie par $f(\lambda) = 1 - 2\lambda + 2\lambda^2 \log(1 + 1/\lambda)$.

Remarque. Certains des résultats sur les espérances de ces statistiques étaient déjà connus dans le cadre du modèle d'Ewens classique. Nous devons utiliser la bijection fondamentale pour retrouver ces résultats. Les statistiques étudiées ne sont donc pas les mêmes mais ces dernières sont en correspondance directe par cette bijection. La statistique la plus évidente est probablement celle du nombre de cycles dans la distribution d'Ewens classique qui correspond au nombre de records étudié dans le Corollaire 2. Ce résultat était connu également dans la distribution d'Ewens, nous pouvons par exemple le retrouver dans le livre écrit par Arratia, Barbour et Tavaré [5, The Ewens Sampling Formula]. De la même façon, le nombre moyen d'anti-excédences dans la distribution classique correspond au nombre moyen de descentes sous la distribution généralisée au record et a été étudié par Féray [24]. Une anti-excédences dans une permutation $\sigma \in \mathfrak{S}_n$ est une position $i \in [n]$ telle que $\sigma(i) < i$. La valeur moyenne du premier élément obtenue dans le Corollaire 4 est équivalente sous la distribution d'Ewens classique à la valeur moyenne du plus petit élément maximal sur tous les cycles. À notre connaissance, cette statistique n'a pas été étudiée auparavant dans la littérature. Enfin, nous ne savons pas si il existe une interprétation naturelle des inversions de σ dans sa pré-image $F^{-1}(\sigma)$ et à notre connaissance il n'y a pas d'équivalent du Corollaire 5.

6.6 Conclusion

Dans ce chapitre, nous avons cherché à étudier des distributions non-uniformes sur les permutations. Nous avons vu la distribution d'Ewens qui associe à une permutation $\sigma \in \mathfrak{S}_n$ une probabilité $\mathbb{P}_{\theta,n}(\sigma)$ qui dépend du nombre de cycles de σ. Nous avons vu qu'il est naturel de généraliser cette distribution à d'autres statistiques sur les permutations. Nous avons en particulier, étudié le cas de la distribution d'Ewens généralisée au nombre de records. Pour ce dernier, nous avons vu que le choix du paramètre θ a une influence sur le comportement d'autres statistiques comme le nombre de descentes, les inversions et la valeur du premier élément. Nous avons pu voir par exemple qu'il faut faire dépendre θ très fortement de la taille de la permutation pour espérer obtenir une complexité intéressante dans le cas moyen pour le tri par insertion. Dans notre article [7], nous avions également fait une étude des algorithmes 8 et 9 dans le contexte de la prédiction de branchements. Il est possible à l'aide du Lemme 20

de déterminer le nombre d'erreurs de prédictions de ces deux algorithmes pour un prédicteur 1-bit. Dans ce cas, les erreurs de prédictions se produisent à la position i si il y a une alternance de montées et de descentes à cette position. On peut montrer que ces ensembles sont quotient-stables et parvenir à obtenir ainsi ce résultat. Nous avons omis le détail de ces derniers car ils ne concernent que le prédicteur 1-bit et les formules sont peu élégantes. Ce que nous pouvons en retenir cependant, c'est que lorsque la séquence d'entrée présente peu de désordre, alors l'Algorithme 9 est meilleur que l'Algorithme 8 en prédiction et donc en temps d'exécution puisqu'il fait moins de comparaisons. Cela peut se comprendre intuitivement puisque si la séquence d'entrée est presque triée, le test des deux éléments consécutifs qui pénalisait l'Algorithme 9 dans le cas uniforme devient hautement prévisible.

Notre motivation était de donner un cadre pour analyser les algorithmes dont l'entrée est une séquence ordonnée dans le cas moyen, comme par exemple des algorithmes de Tri. Nous avons au début de ce chapitre l'Algorithme 18 qui tri efficacement les séquences qui ont un grand nombre de records. Nous savons que si $\theta = n^\delta$ avec $\delta > 1$, alors $\mathbb{E}_{\theta,n}[rec] \sim n$. Rappelons que la mesure $mrec$ d'une séquence compte son nombre de non-records, en utilisant l'inégalité de Tchebytchev nous avons alors

$$\mathbb{P}(mrec \geq 1) \leq n - \mathbb{E}_{\theta,n}[rec] \to 0,$$

et donc $\mathbb{P}(mrec = 0) \to 1$. Ainsi, sous la distribution d'Ewens généralisée pour les records en choisissant ce type de θ, nous montrons que la complexité dans le cas moyen est linéaire. En effet, nous avons pour $\mathcal{C}_{rs}$ le nombre de comparaisons effectuées par l'algorithme 18 :

$$\mathbb{E}[\mathcal{C}_{rs}] = \sum_{k=0}^{n-1} \mathbb{P}(mrec = k)(n + \mathcal{O}(k \log k)) \sim n + \mathcal{O}(1).$$

Pour la suite, il pourrait être intéressant de faire l'analyse d'autres algorithmes sous la même distribution. Cette généralisation peut facilement s'appliquer à d'autres statistiques, si bien que nous pourrions également étudier la distribution pour d'autres fonctions sur les permutations, comme par exemple le nombre de descentes qui est fortement corrélé au nombre de runs. Pour cette distribution, nous ne pouvons plus utiliser un raisonnement similaire au Lemme 20 qui consistait à faire un découpage simple de la permutation sous forme de mot en un préfixe et un suffixe. Cependant, de par la relation entre le nombre de descentes et le nombre de runs d'une séquence, cette distribution présente cependant un intérêt dans le cadre de l'analyse en moyenne de NaturalMergeSort, mais également de TimSort qui s'adaptent à cette mesure.

Chapitre 7

Conclusion et perspectives

7.1 Conclusion

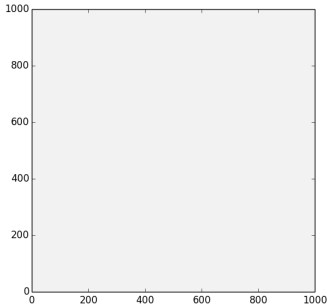
À l'origine, notre but était d'étudier TimSort, un algorithme de tri, qui est apparu au début du 21ème siècle dans un monde où il existait déjà une connaissance considérable dans le domaine. Ce qui est intéressant avec TimSort c'est qu'il a été adopté comme algorithme de tri par défaut de langages de programmation célèbres aujourd'hui alors qu'il y existait de nombreux algorithmes de tri qui ont déjà été conçus et étudiés pendant plus de 70 ans. Il était donc intéressant d'essayer de comprendre le fonctionnement de TimSort afin de déterminer ce qui fait son efficacité dans les applications que nous avons de cet algorithme dans la réalité.

Dans le chapitre 4, nous avons présenté le problème de tri dans le but d'introduire les algorithmes de tri adaptatif et en particulier TimSort. Il était dit dans la note d'explication laissée par Tim Peters lors de la création de TimSort que l'algorithme avait une complexité dans le pire cas en $\mathcal{O}(n \log n)$ comparaisons. Nous n'avons trouvé aucune trace de la preuve de cette complexité, c'est pourquoi nous proposons une démonstration de cette dernière. Un bug avait été trouvé dans TimSort, et il a été corrigé depuis. C'est sur cette version corrigée que nous avons principalement travaillé, la preuve de la complexité considère également ces corrections.

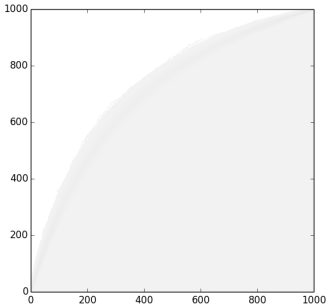
Nous avons vu que l'algorithme du tri fusion naturel de Knuth, pour trier une séquence X de taille n à r runs, avait une complexité en $\mathcal{O}(n \log r)$ et est optimal au sens de Mannila. Nous savons que TimSort trie la séquence en fonction des runs croissants et décroissants dans la séquence d'entrée. Il est évident que TimSort s'adapte donc à la mesure qui consiste à compter le nombre de runs mais nous pouvons alors nous demander de ce qu'il en est de l'optimalité de TimSort vis-à-vis de cette mesure. Au moment où nous rédigeons ces lignes, nous avons publié un article [8] qui démontre que si r est le nombre de runs croissants et décroissants dans la séquence d'entrée de taille n, alors TimSort a besoin de $\mathcal{O}(n \log r)$ comparaisons pour trier cette séquence. Nous avons vu également que dans ses notes, Tim Peters explique que les règles choisies ont été conçues pour essayer d'exploiter le cache mémoire, à l'heure actuelle nous n'avons proposé aucune analyse sur le nombre d'erreurs de cache à l'exécution de TimSort, mais il pourrait être intéressant de faire cette analyse et de la comparer avec d'autres algorithmes classiques de tri.

Dans le chapitre 5, nous nous sommes intéressés à la caractéristique d'architecture de nos processeurs actuels de la prédiction de branchements. Nous nous sommes particulièrement intéressés à la conception d'algorithmes qui essayent de guider les prédicteurs pour minimiser le nombre d'erreurs de prédictions. Nous avons, en particulier, modifié les algorithmes de l'exponentiation rapide et de la recherche dichotomique pour lesquels nous avons obtenu de meilleurs résultats sur leurs performances comparés à leurs versions d'origines. Nous pensons que la prédiction de branchement explique ces différences de performances. À l'avenir, il pourrait être intéressant de modifier d'autres algorithmes élémentaires afin d'améliorer leurs temps d'exécutions. Nous avons proposé pour cela des pistes de généralisation des méthodes que nous avons appliquées pour obtenir nos versions modifiées et qui guident la prédiction des deux algorithmes mentionnés précédemment. Cette méthode pourrait, si elle est développée, servir de recette pour obtenir de meilleurs algorithmes, dans le sens où ils guident la prédiction, et ce dès leurs conceptions.

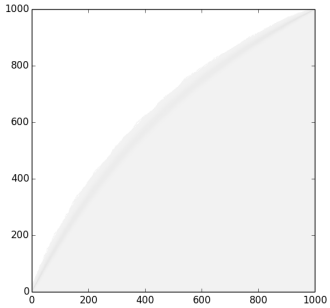
Dans le chapitre 6, nous nous sommes intéressés à l'étude de permutations non-uniformes. Nous avons découvert l'existence de la distribution d'Ewens, très connue des probabilistes, qui attribue un poids à chaque permutation en fonction de son nombre de cycles et d'un paramètre réel $\theta > 0$. Nous avons proposé une généralisation naturelle de cette distribution à d'autres statistiques quelconques sur les permutations. Nous avons ensuite étudié un cas particulier avec la distribution d'Ewens généralisée au nombre de records, qui est associé à une mesure de désordre au sens de Mannila. Nous avons alors regardé l'influence que pouvait avoir cette distribution sur les autres statistiques associées à d'autres mesures de désordre. Nous avons, par la suite, regardé différents cas critiques lorsque le paramètre θ dépend plus ou moins fortement de la taille de la permutation. Par ailleurs, nous avons déduit l'influence du nombre de records sur les inversions que l'algorithme de tri par insertion restait en moyenne quadratique sous cette distribution tant que θ ne dépend pas très fortement de la taille de la permutation n avec par exemple $\theta = n^{\delta}$ pour $\delta > 0$. Pour l'étude de ces cas critiques, nous nous sommes intéressés à observer, en faisant des simulations, la probabilité des valeurs de chaque élément. Nous avons ensuite donné une représentation graphique de ces probabilités. La figure 7.1 donne ces représentations sous forme de graphes. Pour obtenir ces graphes, nous avons généré plusieurs permutations sous la distribution d'Ewens généralisée au nombre de records. Soit n, la taille de nos permutations, nous avons alors regardé pour $i \in [n]$ et pour une valeur $j \in [n]$ le nombre de fois que $\sigma(i) = j$, nous notons par N_{ij} cette quantité. Sur le graphe nous avons, au voisinage de la position (i, j), dessiné un carré de couleur gris. Plus la quantité N_{ij} est grande et plus le carré sera sombre, dans le cas contraire il sera plus clair. Pour chacun des cas que nous avons étudiés, nous pouvons nous apercevoir qu'il existe des courbes caractéristiques. Pour déterminer ces courbes, il faut nous intéresser à la définition de permuton. L'étude de ces courbes devrait être donnée dans une version longue de notre article.



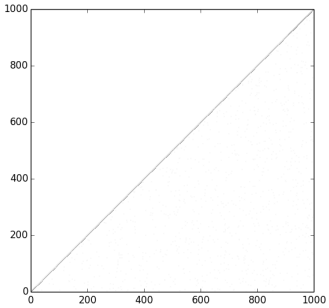
(a) θ constant, ici nous avons $\theta = 1$ le cas uniforme.



(b) $\theta = n^{\epsilon}$ pour $0 < \epsilon < 1$, ici $\delta \approx 0.8673$.



(c) $\theta = \lambda n$ pour $\lambda \in \mathbb{R}^{+}$, ici $\lambda = 1$.



(d) $\theta = n^{\delta}$ pour $\delta > 1$, ici $\delta \approx 2.2329$.

FIGURE 7.1 – Une représentation graphique de la distribution de nos permutations sous la distribution d'Ewens généralisée au nombre de records. Les permutations sont ici de taille 1000 et l'échantillon généré en contient 10000.

7.2 Perspectives

Les travaux présentés dans ce manuscrit donnent des premiers résultats intéressants concernant l'algorithme TimSort, l'exploitation de la prédiction de branchement et l'utilisation des distributions non-uniformes pour l'étude d'algorithmes de tris adaptatifs. Il reste cependant beaucoup de pistes qui peuvent être explorées. Nous allons émettre plusieurs directions de recherche pour chacun de ces trois chapitres.

7.2.1 TimSort

Nous avons proposé une preuve de la complexité de TimSort dans le pire cas et sous le modèle RAM en nous intéressant au nombre de comparaisons. Nous pensons qu'une des raisons qui ont poussé les développeurs de Java et de Python à utiliser cet algorithme est sa capacité à exploiter les caractéristiques des architectures modernes, et en particulier le cache mémoire comme le revendique Tim Peters dans sa note [37]. Il pourrait donc être intéressant dans un

premier temps de vérifier ces propos en faisant une analyse du nombre d'erreurs de cache sous divers modèles de cache proposés dans la littérature, comme par exemple celui utilisé par Lamarca pour son analyse des tris [31]. Nous pourrions, de plus, regarder comment se comporte TimSort du point de vue de la prédiction de branchement, avec des analyses similaires à celles que nous avions effectuées dans le chapitre 5.

Nous ne proposons qu'une analyse dans le pire cas de TimSort. Bien que cela semble difficile, nous pourrions essayer de déterminer sa complexité dans le cas moyen. Dans un premier temps, nous pouvons considérer le cas où l'entrée est une permutation issue de la distribution uniforme. Pour mettre en évidence la propriété adaptative de TimSort, nous pourrions considérer dans un deuxième temps, des distributions non-uniformes comme celles que nous avons vues au cours du chapitre 6.

TimSort est un algorithme qui a su se montrer efficace, cependant le choix de son implantation a été probablement fait après avoir testé ce dernier en conditions réelles. Néanmoins, certaines fonctionnalités de TimSort ne sont peut-être pas aussi utiles que ce que nous pourrions croire. Nous avons déjà vu comment nous pouvions créer de nouveaux algorithmes à la TimSort en choisissant une stratégie sur la pile des runs. Nous avions utilisé cette méthode pour faire une première analyse d'une version simplifiée de TimSort, mais il ne nous semble pas impossible qu'il existe une autre stratégie qui soit plus efficace que celle utilisée actuellement par TimSort. Nous n'avons que très peu étudié la fonctionnalité du galop, qui n'est pas censée modifier le comportement de TimSort dans le pire cas. Il pourrait être intéressant de regarder plus en détail son fonctionnement pour vérifier si cette fonctionnalité est réellement efficace.

7.2.2 Prédiction de branchement

Nous avons exploré une faible quantité d'algorithmes standards. Il serait donc particulièrement intéressant de trouver d'autres algorithmes pour lesquels nous pourrions obtenir des améliorations similaires. Parmi nos recherches infructueuses, nous avons essayé d'obtenir des arbres de recherches binaires déséquilibrés, car nous savons d'après l'article de Brodal et Moruz [18] que si nous parvenons à obtenir un tel déséquilibre, nous y gagnons en temps d'exécutions sur les algorithmes qui parcourent ces derniers. Nous pensons qu'il existe de nombreux algorithmes dans le domaine de la géométrie algorithmique où il est possible d'obtenir des gains de performance intéressants. Dans ce domaine, il est souvent nécessaire de faire des tests d'orientations, pour, par exemple, savoir si un point est à l'intérieur ou à l'extérieur d'une surface. Si le résultat de ce test est équiprobable pour les deux issues possibles, alors il serait intéressant d'appliquer ce que nous avons vu au cours du chapitre 5. Les algorithmes de recherches de motifs dans un texte, comme celui de Knuth-Moris-Pratt, sont utilisés dans de nombreux programmes comme une étape de pré-traitement. Il y a de nombreux tests d'égalité pour vérifier si une lettre du texte est identique à une lettre du motif à rechercher. Ce test est donc, une fois encore, susceptible d'être modifié afin de guider la prédiction de branchement.

Contrairement à ce que nous avons fait jusqu'ici, il peut également être intéressant de faire tout ce qui est possible pour ne pas guider le prédicteur. Un domaine où ça pourrait être le cas est celui de la cryptographie. Supposons qu'un espion surveille le temps nécessaire pour chiffrer ou déchiffrer un message d'une

taille qui lui est connue. Supposons également que cet espion soit au courant de la méthode utilisée pour obtenir le résultat, et que dans cette dernière nous avons des branchements. Si ces branchements guident le prédicteur, alors il est possible que la durée d'exécution de cette méthode donne des indices sur les clés utilisées pour le chiffrement, ce qui n'est pas un effet désirable. Pour résoudre ce problème, nous pouvons utiliser des instructions de type CMOV, mais dans le cas où il est impossible de remplacer l'instruction de branchement par un CMOV, il serait alors intéressant de faire en sorte que le prédicteur ne soit pas guidé en rajoutant volontairement des perturbations.

Nous avions proposé, à la section 5.4 du chapitre 5, des pistes de généralisations qui mettaient en évidence l'exploitation de propriétés particulières des algorithmes que nous avons modifiés. Il pourrait être intéressant de développer ces pistes pour proposer une véritable théorie qui pourrait être utilisée pour automatiser l'application de modifications qui guident le prédicteur de branchement. Nous y avions montré comment obtenir des équivalences d'algorithmes avec des opérations de déroulements de boucles et de réarrangements des tests, si certaines conditions étaient vérifiées dans l'algorithme. Nous n'avons cependant pas montré si ces modifications étaient toujours efficaces, bien que nous savons que c'est le cas pour la recherche dichotomique et l'exponentiation rapide. Il serait donc dans un premier temps intéressant de rajouter des espaces de probabilités dans cette théorie et de déterminer les conditions pour lesquelles ces équivalents permettent d'obtenir un gain en performance.

Il pourrait être intéressant de pouvoir proposer des optimisations pour guider le prédicteur au moment de la compilation. Nous pourrions par exemple effectuer une première version sans optimisations. Il faudrait ensuite mesurer les performances de cette première version pour des entrées intéressantes pour l'utilisateur, ce qui pourrait permettre d'avoir une idée de la distribution de l'entrée en pratique. À l'aide de ces mesures, le compilateur propose alors une deuxième version qui contient des optimisations pour guider le prédicteur si nécessaire. Ces optimisations pourraient être obtenues en appliquant les équivalences de la théorie ci-dessus. Comme les performances peuvent varier en fonction du modèle de prédicteur utilisé, faire ces modifications à l'étape de compilation pourrait ainsi permettre d'obtenir des implantations de nos algorithmes qui seraient plus adaptées à chaque ordinateur.

7.2.3 Modèle d'Ewens

Nous avons proposé l'étude des distributions non-uniformes sur les permutations dans le but de proposer un cadre pour analyser les algorithmes de tris adaptatifs. Nous n'avons cependant proposer que très peu d'analyses dans le cas moyen sous ces distributions. Il pourrait donc être intéressant de faire l'analyse de divers algorithmes de tris sous ces distributions, nous pourrions en particulier regarder le cas du tri fusion naturel et de TimSort.

Nous avons, par soucis de simplicité, étudié le cas de la distribution d'Ewens généralisée au nombre de records. Nous pourrions nous demander ce que nous pouvons obtenir avec le choix d'autres statistiques sur les permutations. Dans le cas du tri fusion naturel et de TimSort, les nombres de descentes et de montées devraient être des choix pertinents puisque ces algorithmes s'adaptent à ces statistiques. Nous avons essayé d'étudier ces distributions, mais nous avions peu de résultats intéressants, ce qui nous a conduit à ne pas aller plus loin

dans le cadre de cette thèse. La difficulté vient du fait qu'il n'est plus possible d'utiliser le Lemme 18 de décomposition qui nous a été fort utile pour démontrer de nombreux résultats sous la distribution d'Ewens généralisée au nombre de records.

Nous pourrions, à l'instar des développeurs de Java, faire une série de benchmarks pour obtenir des statistiques sur les séquences d'entrée couramment utilisées lors de l'appel des fonctions de tris dans l'industrie. Pour y parvenir, il nous faudrait potentiellement établir des partenariats avec des entreprises dans le but de pouvoir utiliser des espions qui feraient des analyses sur ces entrées. À l'aide de ces statistiques, nous pourrions comparer nos distributions non-uniformes avec un échantillon des véritables distributions. Cela nous permettrait alors d'affiner ces distributions pour s'approcher encore plus de la réalité.

Annexe A

Librairie PredSim

Afin de vérifier nos résultats, nous avons développé une librairie qui nous permet de simuler des prédicteurs de branchement. Il s'agit d'une librairie python3 que nous avons nommé *predictors*. Cette librairie a été créée dans le but d'être extensible et permet à ses utilisateurs de créer leurs propres prédicteurs. Une partie des prédicteurs qui ont été détaillés au cours du chapitre 3 ont déjà leurs implémentations et peuvent donc être immédiatement utilisés. Cette annexe est composée de deux parties. La première explique comment installer la librairie et comment l'utiliser. La deuxième partie contient les scripts que nous avons utilisés pour vérifier nos résultats théoriques.

A.1 Principe de fonctionnement de la librairie

A.1.1 Premiers pas avec la librairie

Nous allons dans un premier temps voir comment nous pouvons installer la librairie et son utilisation de base avec un exemple complet.

Installation

En utilisant la commande pip : La librairie est disponible sur le dépôt pypi et est donc accessible avec les différents logiciels qui permettent d'installer les packages qui y sont déposés comme *pip* ou *easy_install*. Pour installer simplement la librairie vous pouvez par exemple utiliser la commande suivante :

```
python3 -m pip install predictors
```

En téléchargeant le code source : Il est également possible de créer votre propre installation de la librairie en téléchargeant directement son code source. Ce dernier est disponible sur le dépôt github se trouvant à l'adresse `https://github.com/ognico/predictors`.

Utilisation de la librairie

La librairie est basée sur la classe *Predictor* qui se trouve dans le fichier *predictor.py*. Lorsque l'on veut utiliser une sous-classe de *Predictor*, il y a quatre méthodes qui vont nous intéresser et deux attributs. Pour les deux attributs, il s'agit de deux compteurs. Le premier compteur *mispredictions* compte le nombre d'erreurs de prédiction que le prédicteur a effectué. Le deuxième compteur *instructions* compte le nombre d'instructions de branchement qui ont été traitées par le prédicteur. Ces compteurs comptent à partir de l'instanciation du prédicteur ou après ré-initialisation. Nous allons maintenant voir en détail les différentes méthodes qui nous seront utiles pour développer nos simulations :

- *__init__()*, il s'agit du constructeur de l'objet, par défaut la classe *Predicteur* ne prend pas d'arguments, mais il est possible que les sous-classes en proposent. C'est par exemple le cas du prédicteur 1-bit qui prend un argument pour déterminer son état initial, si ce dernier n'est pas précisé alors l'état initial est choisi au hasard.
- *eval()*, il s'agit de la méthode principale qui est utilisée pour évaluer une instruction de branchement. Cette méthode prend en argument une condition et retourne la valeur booléenne de cette dernière. L'intérêt de

retourner ce résultat est qu'il nous est alors possible d'écrire cette dernière directement dans une instruction python *if*. Par exemple le code

```
if x > 0:
        #do something
```

peut facilement être transformé pour utiliser un prédicteur *pred* de la façon suivante :

```
if pred.eval(x > 0):
        #do something
```

— *reset_counters()*, cette méthode permet de réinitialiser les deux compteurs à 0.

— *get_and_reset_counters()*, cette méthode retourne les valeurs des compteurs sous la forme d'un couple (mispredictions, instructions) et les réinitialisent ensuite à la valeur 0.

```python
#importe les prédicteurs statiques always False et always True
from predictors.static import FStaticPredictor, TStaticPredictor
#importe le prédicteur 1 bit
from predictors.one_bit import OneBitPredictor
#importe les prédicteurs 2 bits saturé et avec saut
from predictors.two_bits import TwoBitsPredictorSat, TwoBitsPredictorFlipFlap
#importe le prédicteur 3 bits saturé
from predictors.three_bits import ThreeBitsPredictorSat
#importe un prédicteur global avec table d'historiques
from predictors.global_ import GlobalPredictor

from random import randint

def collatz(val, n, pred):
  """Implémentation de la suite de collatz.

  Prends 3 arguments :
    - val est la valeur initiale de la suite
    - n est le nombre d'itérations à effectuer dans la suite
    - pred est un prédicteur de branchement qui hérite de la classe Predictor

  La fonction retourne la valeur du n-ème terme de la suite en commençant par val.
  """
  for _ in range(n):
    if pred.eval(val%2 == 0): #val est pair
      val = val/2
    else:
      val = 3 * val + 1
  return val

Predictors = (
  FStaticPredictor,
  TStaticPredictor,
  OneBitPredictor,
  TwoBitsPredictorSat,
  TwoBitsPredictorFlipFlap,
  ThreeBitsPredictorSat,
  GlobalPredictor,
)

N = 2**20
x = randint(5,1024)

for Predcls in Predictors:
  print('==================', Predcls.__name__, '==================')
  pred = Predcls()
  ans = collatz(x, N, pred)
  print("ans = ", ans, "#mp = ", pred.mispredictions)
```

Listing 4 – Exemple d'utilisation de la librairie *predictors* sur la suite de Collatz.

Nous donnons dans le Listing 4 un exemple d'utilisation de la librairie. Cette exemple simule les différents prédicteurs qui sont fournis par défaut dans la

librairie sur une implémentation de la suite de Collatz comme nous l'avons vu au cours du chapitre 3.

Dans ce script nous choisissons une valeur initiale choisie au hasard entre 5 et 1024 et nous itérons 2^{20} fois. Nous exécutons alors cette suite pour chacun des prédicteurs et nous affichons le nombre d'erreurs de prédictions qui se sont produites en tout. Concentrons nous sur les lignes 25, 46 et 48. La ligne 25 est sans doute la plus intéressante, dans cette dernière nous faisons le test de la parité de la valeur actuelle dans la suite de Collatz pour savoir comment déterminer la valeur suivante. Pour ce faire, nous utilisons la méthode *eval()* qui prend en paramètre le test de parité. Nous avons directement utilisé cette méthode dans une instruction if ce qui rend la lecture quasiment similaire à l'usage classique sans l'appel de cette dernière. Dans la ligne 46, nous appelons le constructeur de chacun des prédicteurs dont la classe se trouve dans le tuple *Predictors* que nous avons créé à la ligne 31. À la ligne 48, nous faisons un accès au compteur nous donnant le nombre d'erreurs de prédictions qui se sont produites après l'appel de la fonction *collatz*. Ce compteur est alors affiché par l'intermédiaire de la fonction *print()*.

Création de nouveaux prédicteurs

Nous allons maintenant nous intéresser à la création d'une nouvelle classe de prédicteur dans le cadre de cette librairie. Nous allons voir en détail comment nous avons implémenté le prédicteur 1-bit qui se trouve dans le module *predictors.one_bit*.

Hérité de la classe Predictor : Comme nous l'avons vu précédemment, toutes nos classes de prédicteurs héritent de la classe de base Predictor dans le module *predictors.predictor*. Notre fichier commence donc naturellement par importé cette classe et d'en hériter.

```
from predictors.predictor import Predictor

class OneBitPredictor(Predictor):
        #contenu de la classe
```

Extension du constructeur : Si votre prédicteur doit retenir des données concernant son état, il vous sera très probablement nécessaire d'étendre le constructeur en faisant une ré-écriture de la méthode *__init__()*. Dans notre cas, nous initialisons le prédicteur 1-bit en initialisant sa prédiction initial. Nous avons laissé la possibilité à l'utilisateur de préciser cet état. Dans le cas où ce dernier argument n'est pas précisé, nous choisissons une valeur au hasard.

```
def __init__(self, state = None):
  super().__init__()
  self.state_ = state
  if state is None:
    self.state_ = not getrandbits(1) #fast way to choose random boolean
```

Implémenter la méthode *_prediction()* : Il est obligatoire d'implémenter cette méthode. Si ce n'est pas fait, alors cette dernière lève une exception. Cette méthode est probablement la plus importante lors de la création d'un prédicteur, puisque c'est cette dernière qui va définir son comportement. Le rôle de cette méthode est de donner la valeur de la prédiction de la prochaine instruction de branchement à évaluer. Dans le cas d'un prédicteur 1-bit, la méthode renvoie le résultat de la dernière instruction de branchement qui a été évaluée. Cette valeur est simplement stockée dans l'attribut *state_* .

```
def _prediction(self):
  return self.state_
```

Implémenter la méthode *_update()* : Si nécessaire, il est possible d'implémenter cette méthode. Cette dernière a été mise à disposition pour mettre à jour l'état du prédicteur après l'évaluation d'une instruction de branchement. Cette méthode prend deux arguments, le premier est la prédiction qui a été donné par la méthode *_prediction()* pour l'instruction qui vient d'être évalué et le deuxième est son vrai résultat. Dans le cas du prédicteur 1-bit, seul le résultat nous intéresse et est directement affecté à l'attribut *state_* .

```
def _update(self, pred, result):
  self.state_ = result
```

Nous venons de voir toutes les étapes à réaliser pour la création d'une nouvelle classe de prédicteur dans la librairie *predictors*. Le Listing 5 donne le code complet de la classe *OneBitPredictor*. D'autres exemples de classes peuvent être consultés dans le dépôt github de la librairie `https://github.com/ognico/predictors`.

A.2 Sources des simulations effectuées

Dans cette section, nous donnons les codes sources des simulations que nous avons effectuées dans le but de vérifier nos résultats.

```python
"""This module exports the class OneBitPredictor.
This class extends the class Predictor of the module predictor.
It implements a simple one bit predictor.
"""

from random import getrandbits
from predictors.predictor import Predictor

class OneBitPredictor(Predictor):
  """An implementation of the simple one bit predictor.

  This class extends the class Predictor of the module predictor.
  This is an implementation of the one bit predictor.
  This predictor always predict the result of the previous branch instruction.
  A description of these predictors can be consulted in the section 2 of the following article:
  http://drops.dagstuhl.de/opus/volltexte/2016/5713/pdf/13.pdf

  This class behavior is mostly inherited by the class Predictor.
  The following methods have been extended:
    - __init__()
  The following methods have been overriden:
    - _prediction()
    - _update()
  """

  def __init__(self, state = None):
    """Initialize the counters and initial prediction.

    Initialize counters prediction and instruction to zero.
    Initialize the initial state that correspond to the first prediction.
    This function take one argument:
        - state a boolean corresponding to the initial state.
          If left to None then the initial state is choosed randomly.
    """
    super().__init__()
    self.state_ = state
    if state is None:
        self.state_ = not getrandbits(1) #fast way to choose random boolean

  def _prediction(self):
    """Return the prediction for the next branch."""
    return self.state_

  def _update(self, pred, result):
    """Update the state to the result of the branch instruction."""
    self.state_ = result
```

Listing 5 – La classe OneBitPredictor qui implémente un prédicteur 1-bit dans la librairie *predictors*.

A.2.1 Recherche simultanée du minimum et du maximum

```
from predictors.one_bit import OneBitPredictor
from predictors.two_bits import TwoBitsPredictorSat
from predictors.three_bits import ThreeBitsPredictorSat

from random import random, shuffle

SAMPLE_SIZE = 10
MIN_SIZE = 1000
NB_ITER = 18
NUM_PREDS_OPTI = 5
NUM_PREDS_NAIVE = 2

def naive_minmax(next_value, preds):
  min_ = naive_minmax.minimum
  max_ = naive_minmax.maximum
  if preds[0].eval(next_value < min_):
    min_ = next_value
  if preds[1].eval(next_value > max_):
    max_ = next_value
  naive_minmax.minimum = min_
  naive_minmax.maximum = max_
  return min_, max_

def optimized_minmax(next_values, preds):
  min_ = optimized_minmax.minimum
  max_ = optimized_minmax.maximum
  if preds[0].eval(next_values[0] < next_values[1]):
    if preds[1].eval(next_values[0] < min_):
      min_ = next_values[0]
    if preds[2].eval(next_values[1] > max_):
      max_ = next_values[1]
  else:
    if preds[3].eval(next_values[0] > max_):
      max_ = next_values[0]
    if preds[4].eval(next_values[1] < min_):
      min_ = next_values[1]
  optimized_minmax.minimum = min_
  optimized_minmax.maximum = max_
  return min_, max_

def get_result(preds):
  res = 0
  for pred in preds:
    res += pred.get_and_reset_counters()[0]
  return res

onebit_preds_opti = [OneBitPredictor() for _ in range(NUM_PREDS_OPTI)]
twobits_preds_opti = [TwoBitsPredictorSat() for _ in range(NUM_PREDS_OPTI)]
```

```
threebits_preds_opti = [ThreeBitsPredictorSat() for _ in range(NUM_PREDS_OPTI)]

onebit_preds_naive = [OneBitPredictor() for _ in range(NUM_PREDS_NAIVE)]
twobits_preds_naive = [TwoBitsPredictorSat() for _ in range(NUM_PREDS_NAIVE)]
threebits_preds_naive = [ThreeBitsPredictorSat() for _ in range(NUM_PREDS_NAIVE)]

predictors_list_opti = [onebit_preds_opti, twobits_preds_opti, threebits_preds_opti]
predictors_list_naive = [onebit_preds_naive, twobits_preds_naive, threebits_preds_naive]

def reset_all():
    naive_minmax.prev_minimum = float("inf")
    naive_minmax.prev_maximum = float("-inf")
    optimized_minmax.prev_minimum = float("inf")
    optimized_minmax.prev_maximum = float("-inf")

asize = MIN_SIZE*(2**17)
for _ in range(17,NB_ITER):
  print(asize, end='\t')
  for _ in range(SAMPLE_SIZE):
    reset_all()
    for _ in range(asize//2):
        current_vals = (random(), random())
        for i in range(len(predictors_list_opti)):
            optimized_minmax.minimum = optimized_minmax.prev_minimum
            optimized_minmax.maximum = optimized_minmax.prev_maximum
            naive_minmax.minimum = naive_minmax.prev_minimum
            naive_minmax.maximum = naive_minmax.prev_maximum
            optimized_minmax(current_vals, predictors_list_opti[i])
            naive_minmax(current_vals[0], predictors_list_naive[i])
            naive_minmax(current_vals[1], predictors_list_naive[i])
        optimized_minmax.prev_minimum = optimized_minmax.minimum
        optimized_minmax.prev_maximum = optimized_minmax.maximum
        naive_minmax.prev_minimum = naive_minmax.minimum
        naive_minmax.prev_maximum = naive_minmax.maximum
  for i in range(len(predictors_list_opti)):
    print(get_result(predictors_list_opti[i])/SAMPLE_SIZE, get_result(predictors_list_naiv
  print()
  asize *= 2
```

Listing 6 – Simulation pour les algorithmes de recherche du minimum et du maximum.

A.2.2 Exponentiation rapide

```
from predictors.one_bit import OneBitPredictor
from predictors.two_bits import TwoBitsPredictorSat, TwoBitsPredictorFlipFlap
from predictors.three_bits import ThreeBitsPredictorSat

from random import randint

SAMPLE_SIZE = 100
MAX_EXP = 2**10
N = 4**MAX_EXP
X = 0.99
NUM_PREDS = 3

def guidedPow(x, n, preds):
  ans = 1
  while n > 0:
    t = x*x
    if preds[0].eval(n % 4 > 0):
      if preds[1].eval(n % 4 >= 2):
        ans *= t
      if preds[2].eval(n % 2 == 1):
        ans *= x
    x = t*t
    n //= 4
  return ans

def get_result(preds):
  res = 0
  for pred in preds:
    res += pred.get_and_reset_counters()[0]
  return res

onebit_preds = [OneBitPredictor() for _ in range(NUM_PREDS)]
twobitssat_preds = [TwoBitsPredictorSat() for _ in range(NUM_PREDS)]
twobitsfp_preds = [TwoBitsPredictorFlipFlap() for _ in range(NUM_PREDS)]
threebits_preds = [ThreeBitsPredictorSat() for _ in range(NUM_PREDS)]

predictors_list = [onebit_preds, twobitssat_preds, twobitsfp_preds, threebits_preds]
results = [0 for _ in range(len(predictors_list))]

for _ in range(SAMPLE_SIZE):
  n = randint(0, N-1)
  for i in range(len(predictors_list)):
      guidedPow(X,n,predictors_list[i])
      results[i] += get_result(predictors_list[i])

for result in results:
  print(result/(2*MAX_EXP*SAMPLE_SIZE))
```

Listing 7 – Simulation l'algorithme d'exponentiation rapide guidé. Ce script met en évidence les constantes α calculé à partir du Théorème 10.

Annexe B

Codes source pour les expérimentations

Nous donnons dans cette annexe, des extraits des codes sources, que nous avons utilisés au cours de nos expérimentations.
Le code source complet est disponible en téléchargement [6].

B.1 Prédiction de branchements

B.1.1 Recherche simultanée du minimum et du maximum

```
1       for(i=0;i<nbr;i++){
2         int a1=ALEA[i];
3         if (a1 < min)
4           min = a1;
5         if (a1 > max)
6           max = a1;
7       }
```

Listing 8 – Extrait de l'implantation en C de l'algorithme naïf

```
1       for(i=0;i<nbr;i+=2){
2         a1=ALEA[i], a2=ALEA[i+1];
3         if (a1 < a2) {
4           if (a1 < min) min = a1;
5           if (a2 > max) max = a2;
6         }
7         else {
8           if (a2 < min) min = a2;
9           if (a1 > max) max = a1;
10        }
11      }
```

Listing 9 – Extrait de l'implantation en C de $\frac{3}{2}$-Minmax

```
/*
  Correspondance entre adresse dans la pile et nom des variables du code source en .c
  i := -4(%rbp)
  min := -12(%rbp)
  max := -16(%rbp)
  a1 := -60(%rbp)
  a2 := -64(%rbp)
*/
.L19:
        movq        ALEA(%rip), %rax
        movl        -4(%rbp), %edx
        movslq        %edx, %rdx
        salq        $2, %rdx
        addq        %rdx, %rax
        movl        (%rax), %eax
        movl        %eax, -60(%rbp)
        movq        ALEA(%rip), %rax
        movl        -4(%rbp), %edx
        movslq        %edx, %rdx
        addq        $1, %rdx
        salq        $2, %rdx
        addq        %rdx, %rax
        movl        (%rax), %eax
        movl        %eax, -64(%rbp)
        movl        -60(%rbp), %eax
        cmpl        -64(%rbp), %eax /*compare a1 et a2*/
        jge         .L14              /*si a1 >= a2 on va en .L14*/
        movl        -60(%rbp), %eax
        cmpl        -12(%rbp), %eax /*compare min avec a1*/
        jge         .L15              /*si a1 >= min on va en L15*/
        movl        -60(%rbp), %eax /*sinon on affecte a1 à min*/
        movl        %eax, -12(%rbp)
.L15:                   /*meme operation en comparant max et a2*/
        movl        -64(%rbp), %eax
        cmpl        -16(%rbp), %eax
        jle         .L17
        movl        -64(%rbp), %eax
        movl        %eax, -16(%rbp)
        jmp         .L17
.L14:                   /*compare min avec a2*/
        movl        -64(%rbp), %eax
        cmpl        -12(%rbp), %eax
        jge         .L18
        movl        -64(%rbp), %eax
        movl        %eax, -12(%rbp)
.L18:                   /*compare max avec a1*/
        movl        -60(%rbp), %eax
        cmpl        -16(%rbp), %eax
        jle         .L17
        movl        -60(%rbp), %eax
```

```
51          movl        %eax, -16(%rbp)
52  .L17:
53          addl        $2, -4(%rbp)    /*i += 2*/
54  .L13:
55          movl        -4(%rbp), %eax
56          cmpl        -100(%rbp), %eax
57          jl        .L19              /*on reboucle le for si i n'est pas assez grand*/
```

Listing 10 – Code assembleur de l'algorithme $\frac{3}{2}$-MINMAX après compilation avec gcc -O0.

B.1.2 Exponentiation rapide

```
1  /* exponentiation rapide classique */
2  double binPow(double x,int n){
3      double r;
4      r = 1;
5      while (n > 0) {
6          if (n & 1) {
7              r = mult(r,x);
8          }
9          n >>= 1;
10         x = mult(x,x);
11     }
12     return r;
13 }
14
15 /* exponentiation rapide classique avec deroulement de la boucle principale. */
16 double skewPow(double x,int n){
17     double r,t;
18     r = 1;
19     while (n > 0) {
20         t = mult(x,x);
21         if (n & 1) {
22             r = mult(r,x);
23         }
24         if (n & 2) {
25             r = mult(r,t);
26         }
27         n >>= 2;
28         x = mult(t,t);
29     }
30     return r;
31 }
32
33 /* exponentiation rapide qui guide le predicteur */
34 double superSkewPow(double x,int n){
35     double r,t;
36     r = 1;
37     while (n > 0) {
38         t = mult(x,x);
39         if (n & 3) {
40             if (n & 1) {
41                 r = mult(r,x);
42             }
43             if (n & 2) {
44                 r = mult(r,t);
45             }
46         }
47
48         n >>= 2;
49         x = mult(t,t);
50     }
51     return r;
52 }
```

Listing 11 – Une implantation en C de nos trois algorithmes d'exponentiations rapides

B.1.3 Recherche dichotomique

```
1  inline int binary(double *T,double xb2,int n){
2    unsigned int d,f,m;
3    d = 0;
4    f = n;
5
6    while (d < f){
7      m = (d+f) / 2;
8      if (T[m] < xb2)
9        d = m+1;
10     else
11       f = m;
12   }
13   return f;
14 }
15
16 inline int binary_biased(double *T,double xb3,int n){
17   unsigned int d,f,m;
18   d = 0;
19   f = n;
20
21   while (d < f){
22     m = (3*d+f)/4;
23     if (T[m] < xb3)
24       d = m+1;
25     else
26       f = m;
27   }
28   return f;
29 }
30
31 inline int skew(double *T,double xs2,int n){
32   unsigned int d,f,m1,m2;
33   d = 0;
34   f = n;
35
36   while (d < f){
37     m1 = (3*d+f)/4;
38     if (T[m1] > xs2)
39       f = m1;
40     else {
41       m2 = (d+f)/2;
42       if (T[m2] > xs2){
43          d = m1+1;
44          f = m2;
45       }
46       else
47          d = m2+1;
48     }
49   }
50   return f;
51 }
```

Listing 12 – Implantation en C de nos trois versions de la recherche dichotomique.

Bibliographie

[1] Librairie de benchmarking java jmh. `http://openjdk.java.net/projects/code-tools/jmh/`.

[2] Librairie papi performance application programming interface. `http://icl.cs.utk.edu/papi/overview/index.html`.

[3] The microarchitecture of intel, amd and via cpus. `http://www.agner.org/optimize/microarchitecture.pdf`.

[4] 5th jilp workshop on computer architecture competitions (jwac-5) : Championship branch prediction (cbp-5). https ://www.jilp.org/cbp2016/, 2016.

[5] Richard Arratia, Andrew D Barbour, and Simon Tavaré. *Logarithmic combinatorial structures : a probabilistic approach*, volume 1. European Mathematical Society, 2003.

[6] Nicolas Auger. Page web personnelle. `http://www-igm.univ-mlv.fr/~auger/`.

[7] Nicolas Auger, Mathilde Bouvel, Cyril Nicaud, and Carine Pivoteau. Analysis of algorithms for permutations biased by their number of records. *CoRR*, abs/1605.02905, 2016.

[8] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the Worst-Case Complexity of TimSort. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4 :1–4 :13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[9] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge Strategies : from Merge Sort to TimSort. working paper or preprint, December 2015.

[10] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Good Predictions Are Worth a Few Comparisons. In Nicolas Ollinger and Heribert Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)*, volume 47 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12 :1–12 :14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg. An experimental study of sorting and branch prediction. *Journal of Experimental Algorithmics*, 12 :1, June 2008.

[12] Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg. An experimental study of sorting and branch prediction. *J. Exp. Algorithmics*, 12 :1.8 :1–1.8 :39, June 2008.

[13] M. Bóna. *Combinatorics of permutations.* Chapman-Hall and CRC Press, 2d edition edition, 2012.

[14] Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. On the adaptiveness of quicksort. *J. Exp. Algorithmics*, 12 :3.2 :1–3.2 :20, August 2008.

[15] Gerth Stølting Brodal and Gabriel Moruz. Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms. In *Algorithms and Data Structures*, volume 3608, pages 385–395. Springer Berlin Heidelberg, 2005.

[16] Gerth Stølting Brodal and Gabriel Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *WADS*, pages 385–395. Springer, 2005.

[17] Gerth Stølting Brodal and Gabriel Moruz. Skewed binary search trees. In *ESA*, pages 708–719. Springer, 2006.

[18] Gerth Stølting Brodal and Gabriel Moruz. Skewed Binary Search Trees. In *Algorithms – ESA 2006*, volume 4168, pages 708–719. Springer Berlin Heidelberg, 2006.

[19] Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4) :354–375, 1973.

[20] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. Introduction à l'algorithmique. 1994.

[21] Stijn De Gouw, Jurriaan Rot, Frank S de Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java. utils. collection. sort () is broken : The good, the bad and the worst case. In *International Conference on Computer Aided Verification*, pages 273–289. Springer, 2015.

[22] Amr Elmasry, Jyrki Katajainen, and Max Stenmark. Branch Mispredictions Don't Affect Mergesort. In *Experimental Algorithms*, volume 7276, pages 160–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[23] Warren J Ewens. The sampling theory of selectively neutral alleles. *Theoretical population biology*, 3(1) :87–112, 1972.

[24] Valentin Féray et al. Asymptotic behavior of some statistics in ewens random permutations. *Electronic Journal of Probability*, 18, 2013.

[25] Philippe Flajolet and Robert Sedgewick. *Analytic combinatorics.* cambridge University press, 2009.

[26] Alexey Gladkich, Ron Peled, et al. On the cycle structure of mallows permutations. *The Annals of Probability*, 46(2) :1114–1169, 2018.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition : A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[28] Kanela Kaligosi and Peter Sanders. How Branch Mispredictions Affect Quicksort. In *Algorithms – ESA 2006*, volume 4168, pages 780–791. Springer Berlin Heidelberg, 2006.

[29] Donald E. Knuth. *The Art of Computer Programming, Vol. 1 : Fundamental Algorithms, 3rd Edition.* Addison-Wesley Professional, 1997.

[30] Donald E Knuth. Sorting and searching, 2nd edn. the art of computer programming, vol. 3, 1998.

[31] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1) :66–104, 1999.

[32] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2008.

[33] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE transactions on computers*, 100(4) :318–325, 1985.

[34] Conrado Martínez, Markus E. Nebel, and Sebastian Wild. Analysis of branch misses in quicksort. In *Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2015, San Diego, CA, USA, January 4, 2015*, pages 114–128, 2015.

[35] Peter McIlroy. Optimistic sorting and information theoretic complexity. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 467–474, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[36] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition : The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[37] Tim Peters. Original notes on timsort. `http://bugs.python.org/file4451/timsort.txt`, 2001.

[38] Jeffrey S Rosenthal. *A first look at rigorous probability theory*. World Scientific Publishing Company, 2006.

[39] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *J. ACM*, 48(2) :170–205, March 2001.

[40] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *ESA*, volume 3221, pages 784–796. Springer, 2004.

[41] Daniel A Spielman and Shang-Hua Teng. Smoothed analysis of algorithms : Why the simplex algorithm usually takes polynomial time. *Journal of the ACM (JACM)*, 51(3) :385–463, 2004.