Université Gustave Eiffel
Laboratoire d'Informatique Gaspard-Monge, UMR 8049

Analysis of Algorithms: Towards a More Realistic Model

Analyse d'algorithmes : vers un modèle plus réaliste

Mémoire d'Habilitation à diriger des Recherches
présentée et soutenue publiquement par

Carine Pivoteau

le 26/01/2026

devant le jury composé de

Contents

Résumé

La combinatoire et l'analyse moyenne d'algorithmes, qui constituent les principaux champs de ma recherche, sont rattachées en France à l'informatique théorique, mais sont souvent considérées à l'étranger comme relevant des mathématiques (éventuellement appliquées). D'ailleurs, parmi les cursus classiques qui conduisent à ces thématiques, plusieurs relèvent des mathématiques, avec une spécialisation en informatique théorique. Pour ma part, j'ai suivi une Licence puis un Master d'Informatique. Cette formation m'a apporté de bonnes bases en théorie des langages et en algorithmique, tout en intégrant les aspects appliqués de l'informatique, notamment à travers une pratique étendue de la programmation dans divers paradigmes. J'ai découvert la combinatoire et l'analyse en moyenne assez tard, en dernière année, notamment grâce au cours d'Analyse d'Algorithmes de Philippe Flajolet. Je me suis alors orientée vers ces domaines, tout en conservant l'influence de mes premières années de formation. Cela m'a naturellement conduite vers la génération aléatoire, qui exploite la connaissance des objets combinatoires pour concevoir des générateurs destinés, entre autres, à fournir des simulateurs pour tester les performances des algorithmes. J'ai en particulier étudié la méthode de Boltzmann, qui permet de construire automatiquement des générateurs à partir de spécifications combinatoires (similaires à des grammaires hors contexte). J'ai poursuivi cette ligne de recherche avec différents groupes de co-auteurs, en automatisant les traitements pour certaines classes d'objets (notamment les permutations à motifs exclus) et pour les séries génératrices, qui sont au cœur des méthodes de combinatoire analytique.

En parallèle, avec l'un de mes principaux co-auteurs, Cyril Nicaud, j'ai commencé à approfondir la question de l'analyse d'algorithmes. En tant qu'enseignants, nous intervenons dans le même type de cursus que celui que j'ai suivi, ce qui nous amène à couvrir des sujets variés, tels que la programmation, le système ou encore l'architecture des ordinateurs. Même si la plupart de ces cours ne sont pas directement orientés vers la recherche, ils nourrissent nos réflexions et entretiennent notre intérêt pour les aspects appliqués de l'informatique. C'est probablement l'une des raisons pour lesquelles nous concevons l'analyse d'algorithmes comme un moyen d'expliquer de manière pertinente les phénomènes observés en pratique. L'analyse en moyenne, qui constitue notre spécialité, offre un premier éclairage théorique sur les performances effectives des algorithmes. Nous souhaitons toutefois aller plus loin et proposer des éléments permettant de faire évoluer le cadre théorique dans lequel s'inscrivent nos résultats. Dans cette perspective, nous avons exploré plusieurs pistes, dont deux que je présente ici. La première consiste à adopter un point de vue alternatif sur les données, en considérant par exemple des distributions non uniformes et calibrées pour mieux représenter les données réelles. Une autre approche est d'enrichir les paramètres de l'analyse en y intégrant des informations sur la structure des données, et pas uniquement sur leur taille. La deuxième piste concerne le modèle de d'ordinateur sur lequel les algorithmes sont implémentés : elle vise à intégrer dans les paramètres d'analyse certains aspects de l'architecture des processeurs.

C'est notre collègue Rémi Forax, avec qui j'enseigne la programmation en `Java`, qui nous a parlé le premier de TimSort. Il s'agit d'un algorithme de tri relativement récent, intégré à l'API standard de `Java` après avoir été introduit dans `Python` au début des années 2000. Le choix des implémentations retenues dans les bibliothèques standard étant généralement validé par des benchmarks, il paraissait évident que cet algorithme devait être particulièrement performant pour avoir supplanté celui précédemment en usage. En y regardant de plus près, nous avons découvert qu'il s'agissait d'un algorithme assez différent de ceux que nous avions l'habitude d'étudier, et que les développeurs de `Python` avaient su innover pour un problème que nous considérions comme traité depuis longtemps. En particulier, ce tri utilise des heuristiques originales pour améliorer ses performances. D'une part, son mécanisme principal cherche à exploiter l'idée que les données manipulées dans les applications sont souvent

plus structurées qu'on ne l'imagine. Il est en effet fréquent que les données soient déjà partiellement triées, parce qu'elles ont simplement été mises à jour ou bien agrégées à partir de sources elles-mêmes triées. D'autre part, les sous-routines employées par TimSort sont conçues pour tirer pleinement parti des caractéristiques des processeurs modernes, notamment en optimisant l'utilisation du cache. Suite à ces observations, nous avons naturellement envisagé d'appliquer nos outils habituels afin de proposer une analyse en moyenne, dans le but de mieux comprendre les raisons des bonnes performances observées en pratique. Cependant, nous avons constaté que, bien que l'algorithme ait été annoncé avec une complexité en $\mathcal{O}(n \log n)$ dans le pire cas, aucune preuve n'existait dans la littérature scientifique. Nous avons donc commencé par établir ce résultat, au moyen d'une analyse en complexité amortie [AJNP18]. Au cours de ce travail, nous avons rapidement réalisé qu'un des atouts majeurs de l'algorithme réside dans son exploitation intelligente des séquences d'éléments déjà triés (*runs*) présentes dans les données. Cela le place dans la catégorie des algorithmes de tri adaptatifs, puisqu'il exploite une certaine "mesures de désordre" [Man85], qui permet de quantifier à quel point la séquence d'entrée est déjà partiellement triée.

La plupart des mesures de désordre proposées dans la littérature sont directement reliées à des statistiques sur les permutations, puisque seul l'ordre relatif des éléments importe pour l'analyse des comparaisons effectuées par un algorithme de tri. Cela nous a conduits à considérer un modèle de permutations aléatoires non uniformes capable de capturer cette notion de mesure de désordre. De telles distributions existent déjà (a priori sans lien avec le contexte considéré ici), parmi lesquelles la distribution d'Ewens [Ewe72] et celle de Mallows [Mal57], qui introduisent un biais exponentiel par rapport au nombre de cycles et au nombre d'inversions, respectivement. Nous avons choisi de nous inspirer de ces modèles pour définir une distribution biaisée vis-à-vis du nombre de records (maximums de gauche à droite). Dans ce modèle, la probabilité d'une permutation dépend d'un paramètre θ qui permet de contrôler, dans une certaine mesure, la proportion de records qu'elle contient [ABNP16, BNP25]. Les permutations présentent alors des caractéristiques différentes de celles suivant une distribution uniforme. Nous avons analysé certaines de ces statistiques, comme le nombre de descentes ou le nombre d'inversions, et obtenu des formules explicites pour leur espérance en fonction de θ, en particulier dans le régime linéaire où $\theta = \lambda n$, avec n la taille de la permutation. Nous avons également établi la distribution asymptotique de ces statistiques lorsque n tend vers l'infini. Du point de vue des algorithmes de tri, le nombre de descentes présente un intérêt particulier puisqu'il correspond au nombre de runs croissants exploités par certains algorithmes de tri comme le NaturalMergeSort de Knuth. Une piste de recherche dans la même direction serait de concevoir un modèle permettant d'analyser les runs alternants (croissants et décroissants), utilisés notamment par TimSort.

En ce qui concerne TimSort, notre première analyse de complexité en pire cas nous a permis de mieux comprendre le fonctionnement de l'algorithme. Nous avons ensuite affiné cette étude et montré que la complexité de TimSort est en $\mathcal{O}(n \log \rho)$, où ρ désigne le nombre de runs. Cependant, pour réellement expliquer ses performances, il est nécessaire d'aller plus loin et d'intégrer la structure des runs dans l'analyse. Nous avons ainsi établi que la complexité de TimSort en pire cas est en $\mathcal{O}(n + n\mathcal{H})$, où $\mathcal{H}$ représente l'entropie de la distribution des longueurs de runs, un paramètre inspiré de l'entropie de Shannon qui permet de prendre en compte simultanément le nombre et la taille des runs. Un tel résultat permet, par exemple, d'expliquer pourquoi TimSort est plus efficace pour trier un tableau composé d'un grand run de longueur $n - 2(\rho - 1)$ suivi de $\rho - 1$ runs de longueur 2, que pour trier une séquence constituée de ρ runs chacun de longueur n/ρ.

Lorsque j'ai pris mes fonctions à l'Université de Marne-la-Vallée, le premier cours que j'ai eu à enseigner était une introduction à l'architecture des ordinateurs pour les étudiants de DUT Informatique. Malheureusement, cet aspect de l'informatique avait été relativement peu abordé dans mon cursus. J'ai donc découvert énormément de choses en préparant ce cours et, de fil en aiguille, j'ai approfondi mes connaissances bien au-delà de ce qui était nécessaire pour une simple introduction. J'ai en particulier pris conscience de l'existence de mécanismes d'architecture moderne, à la fois ingénieux et de plus en plus sophistiqués, conçus pour améliorer sensiblement les performances des programmes. C'est d'ailleurs ce que j'enseigne désormais dans un cours avancé d'architecture des ordinateurs destiné aux étudiants de troisième année de Licence. Parmi les éléments d'architecture moderne abordés, figurent notamment la hiérarchie de mémoire et le fonctionnement du cache, ainsi que la vectorisation des instructions, deux techniques fondamentales pour l'optimisation des performances : la seconde réduit directement le nombre d'opérations de base par un facteur constant, tandis que la première a déjà été étudiée en profondeur dans le cadre des algorithmes en mémoire externe [AV88] et des algorithmes *cache-oblivious* [Dem02, FLPR12]. Nous avons donc là une autre façon d'enrichir le modèle utilisé pour l'analyse d'algorithmes, en considérant cette fois des paramètres non pas liés aux données, mais au modèle de machine sur lequel les algorithmes sont implémentés. Dans mes cours, j'aborde également un troisième sujet : la prédiction de branchement. Dans un processeur moderne, les instructions sont exécutées dans un *pipeline*, ce qui signifie qu'elles se superposent de manière à ce qu'une nouvelle instruction puisse commencer avant que la précédente soit terminée. C'est une technique très efficace pour améliorer l'*ILP*[1] du processeur, mais elle souffre d'un inconvénient majeur : tant que le flot d'instructions est séquentiel, le pipeline reste rempli, mais certaines instructions, telles que les sauts conditionnels, doivent être exécutées intégralement avant de déterminer la suite du programme. Le pipeline se retrouve alors bloqué en attente de cette information. Pour limiter ce problème, le processeur tente de prédire la prochaine instruction. Si la prédiction est correcte, l'exécution se poursuit sans ralentissement. En revanche, si elle est incorrecte, le pipeline doit être vidé, ce qui pénalise fortement les performances. Il en résulte que plus un algorithme contient d'instructions de branchement (notamment des comparaisons), plus son exécution est sensible à la qualité du prédicteur. C'est en particulier le cas des algorithmes de tri et de recherche basés sur les comparaisons d'éléments.

Dans cette optique, nous avons initié un travail visant à intégrer le nombre d'erreurs de prédiction comme paramètre pour l'analyse en moyenne des algorithmes, afin de mettre en évidence les compromis possibles avec le nombre de comparaisons et d'expliquer certaines observations empiriques. Nous avons commencé par étudier trois algorithmes classiques reposant principalement sur les comparaisons [ANP16]. La recherche simultanée du minimum et du maximum dans un tableau constitue un exemple particulièrement frappant de l'existence de tels compromis : bien qu'il existe un algorithme optimal qui effectue $3n/2$ comparaisons pour une entrée de taille n, l'algorithme naïf qui en effectue $2n$ se révèle en pratique deux fois plus rapide pour des entrées aléatoires uniformes. Nous avons montré que ce résultat contre-intuitif peut s'expliquer par une proportion beaucoup plus faible d'erreurs de prédiction pour le second : $O(\log n)$ contre $\mathcal{O}(n)$ pour l'algorithme optimal. Ce constat nous a conduits à examiner d'autres algorithmes fondés sur les comparaisons, tels que l'exponentiation rapide et la recherche dichotomique. Pour chacun d'eux, nous avons proposé des variantes spécifiquement conçues pour réduire le nombre d'erreurs de prédiction. Dans les deux cas, en déséquilibrant volontairement les probabilités des branches suivies lors de l'exécution, nous avons pu améliorer les performances en pratique.

Dans nos analyses, nous avons considéré un modèle de prédicteur local, c'est-à-dire qu'à chaque instruction de branchement du programme est associé son propre prédicteur, lequel utilise l'information des résultats des branchements précédents pour établir ses prédictions.

[1]Instruction-Level Parallelism, ou parallélisme au niveau des instructions

Une des particularités des algorithmes que nous venons d'examiner est que les branchements conditionnels qu'ils impliquent sont presque indépendants les uns des autres, ce qui rend ce type de prédicteur particulièrement efficace. Parmi les algorithmes qui reposent fortement sur les comparaisons, on trouve aussi les algorithmes de traitement de texte, et en particulier ceux de recherche de motif dans un mot. Contrairement aux exemples précédents, ils induisent souvent des branchements auto-corrélés, soit en raison de la structure interne de l'algorithme, soit à cause des auto-corrélations présentes dans le motif lui-même. Nous avons donc poursuivi notre étude avec trois algorithmes de ce type : le plus naïf (celui utilisé pour les `String` en `Java`), qui fonctionne comme une fenêtre glissante en testant le motif à toutes les positions possibles du texte, ainsi que les algorithmes de Morris–Pratt et Knuth–Morris–Pratt, plus performants en termes de nombre de comparaisons [NPV24, NPV25]. Pour chacun, nous avons obtenu des résultats de complexité en moyenne pour le nombre d'erreurs de prédiction, dont certains dépendent du motif, avec parfois des observations surprenantes. Par exemple, pour un modèle de mots aléatoires uniformes (chaque lettre ayant la même probabilité d'apparaître), il arrive que le prédicteur se trompe avec une probabilité supérieure à $1/2$, ce qui est moins bon qu'un choix aléatoire à pile ou face. Heureusement, dans la pratique, les prédicteurs intégrés aux processeurs modernes se comportent bien mieux. Il est en effet possible de mesurer le nombre d'erreurs de prédiction à l'aide de compteurs matériels, même si le modèle exact de prédicteur reste inconnu, car il est presque toujours tenu confidentiel par les constructeurs. Nos résultats, confrontés à ces observations empiriques, mettent en évidence l'inadéquation du prédicteur local dans ce contexte et suggèrent l'utilisation d'autres techniques, en particulier les prédicteurs globaux, qui exploitent l'information issue de l'ensemble des instructions conditionnelles au moyen de tables d'historique, par exemple. Les processeurs modernes recourent d'ailleurs le plus souvent à des prédicteurs hybrides, combinant plusieurs approches, incluant à la fois la prédiction locale et globale. Dans cette perspective, la poursuite naturelle de nos travaux sera d'analyser ces autres modèles de prédicteurs, afin d'obtenir des résultats en meilleure adéquation avec le comportement réel des processeurs.

Ce mémoire d'habilitation présente ces résultats et fournit ainsi une synthèse d'une partie des travaux que j'ai menés depuis ma thèse sur l'analyse d'algorithmes dans un modèle réaliste. Ces travaux m'ont conduit à co-encadrer un doctorat et deux mémoires de master 2, ainsi qu'à contribuer au dépôt d'un projet de recherche dont le financement vient d'être accepté par l'ANR[2]. Je n'aborderai que brièvement, dans la conclusion, les autres travaux menés dans la continuité de ma thèse, en particulier sur le traitement systématique des spécifications combinatoires, qui s'est prolongé pendant près de dix ans, avec Bruno Salvy, par le développement d'une bibliothèque Maple intégrant ces résultats. Nous avons ensuite poursuivi cette collaboration, qui a abouti, après un travail de longue haleine [PS25], à une chaîne de traitement algorithmique complète permettant de calculer automatiquement le développement asymptotique de toutes les séries génératrices issues de la méthode symbolique de Flajolet et Sedgewick [FS09]. J'ai également poursuivi mes travaux sur des thématiques connexes en collaboration avec plusieurs collègues de mon équipe, notamment Florent Koechlin (désormais au LIPN), Pablo Rotondo et Éric Fusy.

[2]PLASMA (Programming Languages, Algorithms and Structures: Models and Analysis) is an ANR research project that will begin in 2026. See https://protondo.github.io/anr-plasma/

Introduction

Analysis of algorithms is the field that studies the efficiency of computer programs through a variety of approaches, providing theoretical estimates of the resources required by an algorithm. Its systematic development began with the work of Donald Knuth, who introduced techniques for both worst-case and average-case scenarios. These two perspectives are essential and complementary in the design and study of efficient algorithms: worst-case analysis provides an upper bound on running time, while average-case analysis offers a more accurate measure of performance when worst cases are rare. A well-known example is QuickSort, which, despite its quadratic worst-case complexity, was often preferred in standard libraries over algorithms with a guaranteed $\mathcal{O}(n \log n)$ bound, such as Knuth's NaturalMergeSort. This preference can be explained in part by QuickSort's $\mathcal{O}(n \log n)$ average complexity (its expected running time over all $n!$ permutations of size n), and by the fact that a good strategy to choose the pivot can mitigate the worst case. This suggests good practical behavior, assuming comparisons have unit cost and inputs resemble uniformly random permutations.

But this may not be the most accurate model of reality: algorithms are implemented as programs that run on real machines and process real data. For example, many algorithms operate on sequences of letters, yet typical DNA sequences look nothing like Shakespeare's words, which in turn differ greatly from the binary sequences produced by compiled programs, and none of these resemble uniformly random words. Likewise, a computer is not a Turing machine: while that model is a useful abstraction, it ignores the complexity of modern processors, which include sophisticated features such as highly efficient caches that can strongly affect performance, as well as the impact of the programming language and compiler optimizations. To account for these factors, one would need details about the implementation, the target hardware, and the actual input data. In practice, this is what benchmarks are for, and engineers rely on them extensively to guide algorithmic choices. Examining the algorithms and data structures in the standard libraries of modern programming languages reveals a clever mix of textbook methods and original heuristics, often diverging from expected designs to create new algorithms or combine existing techniques in innovative ways. A prominent example is the sorting algorithm TimSort, which we will discuss in more detail later. Designed by Tim Peters to perform particularly well on nearly sorted inputs, TimSort was introduced in `Python` in the early 2000s, accompanied by a brief note explaining its rationale and claiming that "it has supernatural performance on many kinds of partially ordered arrays" [Pet]. To illustrate this, the note discusses empirical average complexity on a few examples of almost sorted data, but offers no formal proof. Nevertheless, the algorithm actually performs very well in practice. It gradually gained popularity and is still used in `Java` and `Rust`, yet for over a decade it attracted little attention from the academic community and still lacked a proper complexity analysis despite its widespread adoption. This gap has since been filled, and the techniques and metrics developed for its study have inspired new sorting algorithms, including PowerSort [MW18], which has recently replaced TimSort in `Python`.

This is just one of many examples showing how incorporating real-world insight can lead to significant improvements in algorithm design. To foster and expand this approach, the framework for analyzing algorithms should evolve accordingly. While the classical analysis model is deliberately kept independent of specific implementations and use cases, the perspective I present here takes the opposite approach, aiming to provide complementary information. My main question is: how can we adapt our models to produce a more realistic analysis of algorithms? Among the possible directions, I will focus on two in particular. The first concerns the nature of the input. The uniform distribution is a reasonable starting point and provides valuable general insights, but, as the case of TimSort illustrates, uniformly random permutations seems to be far from representative of typical sorting inputs. Can we propose alternative models that more accurately reflect the kinds of data encountered in practice? Section 1 is devoted to one such model: a parametrized, biased probabilistic distribution on permutations that allows us to control certain structural properties and to analyze various parameters within this framework.

Often, what sets uniform random input apart from real-world data is the internal structure of the objects themselves. For example, the typical height of a uniformly random tree can differ greatly from the height of trees encountered in applications or drawn from another distribution. One way to account for such differences is to incorporate these structural features as parameters in the analysis. In Section 2, the case of TimSort is examined in greater depth. After providing the missing full classical worst-case analysis, which yields an $\mathcal{O}(n \log n)$ running time, an alternative complexity result is presented: TimSort runs in $\mathcal{O}(n + n\mathcal{H})$, where $\mathcal{H}$ is the entropy of the distribution of runs (i.e., maximal monotonic sequences). While not an average-case analysis, this result offers a convincing explanation of why TimSort can be an excellent practical choice, as it quantifies its performance on partially sorted inputs, which is a pattern frequently observed in real-world data.

To enrich the model, it is also sometimes relevant to consider multiple parameters in the analysis. For instance, an algorithm that exploits cache locality effectively may outperform one that accesses less memory but in a random pattern. In the second part of this study, I adopt this perspective for a specific architectural feature: the branch predictor. On modern processors, instructions are pipelined, meaning they overlap in execution so that they can begin before the previous one finishes. This does not hold for branching instructions such as `if` statements: when encountering a branch, the next instruction cannot be executed until the branch outcome is known, stalling the pipeline. To avoid this, the processor predicts the outcome of the branch and preloads the corresponding instruction into the pipeline. If the prediction is correct, execution proceeds smoothly; if not, the pipeline must be flushed, which can significantly slow down the program. This can be particularly challenging for algorithms that rely heavily on comparisons, such as searching or sorting. To account for this effect, I will present work in which this mechanism is incorporated into the analysis model for several algorithms, highlighting compromises between classical cost measures (such as comparisons) and the number of mispredictions, thereby offering a potentially more accurate account of the performance observed in practice. In Section 4, three simple comparison-based algorithms, including binary search, are analyzed with a classical branch predictor model. Evaluating the average misprediction rate alongside the number of comparisons reveals surprising trade-offs, suggesting ways to adjust branch probabilities to improve performance. In particular, variants of the algorithms are proposed that can outperform the classical versions through uneven partitioning. Another broad class of algorithms that can benefit from branch-prediction analysis is pattern matching. Also comparison-based, these algorithms often feature correlated branches, caused both by their structure and the autocorrelation within the pattern, which present a particularly challenging case for the predictors studied here. An initial analysis is presented in Section 5 for three such algorithms, including Knuth–Morris–Pratt, under the same predictor model, while also opening the path to the study of more sophisticated ones.

Overall, the purpose of the work presented here is to develop a framework for the realistic analysis of modern algorithms and data structures, with the aim of both deepening our understanding and guiding their future evolution.

Note All the results presented are drawn from collaborations with various colleagues. Most are already published, with a few exceptions appearing only in preprints. As my aim is to provide a survey of these results, some technical details, more intricate proofs, and results beyond the intended scope will be omitted.

Part I

Real-World Data and Implementations

Over the past two and a half decades, new sorting algorithms have been introduced and adopted, breathing new life into a fundamental problem in computer science that had long seemed settled [Knu98]. Since the introduction of TimSort in `cpython` (the standard implementation of `Python`) in the early 2000s, many modern programming languages have embraced new approaches to sorting. In 2009, Yaroslavskiy proposed a new implementation [Yar] of the Dual-Pivot QuickSort from the 1970s (see, e.g., [Sed77]), which proved fast enough to be included in `Java` 7 for sorting arrays of primitives. TimSort is now used in languages such as `Java` and `Rust`; PatternDefeatingQuicksort, introduced in 2021 [Pet21], is implemented in both `Rust` and `C++`; and PowerSort [MW18] has recently replaced TimSort in `cpython`. Even the classic SampleSort [FM70], often used in parallel systems, has seen renewed interest since 2004 [SW04], with one of the most recent variants being the IPS^{4}o[1] [AWFS17], designed to work in place, run in parallel, be cache-efficient, and avoid branch mispredictions.

These algorithms go beyond simple heuristic modifications of classical techniques. Their development has been driven by two main factors: advances in computer architecture (such as caches, branch prediction, vectorization, etc.), explored in the second part of this study, and the expected structure of the data, which is the focus here. To account for this structure, one can either consider a non-uniform probabilistic model of the input, which is the motivation to introduce record-biased permutations in Section 1, or incorporate additional parameters, beyond input size, into complexity analyses to capture the non-uniformity of real-world data. This latter approach is the one used in the study of TimSort in Section 2.

Real-world data is often too complex for direct mathematical analysis and must be simplified by removing non-essential heuristic details and focusing on key structural features. This is why we[2] focus on permutations when analyzing sorting algorithms, and why we began our exploration of non uniformity by adapting a classical probabilistic model for our record-biased permutations. These provide a model that is both meaningful with respect to presortedness (a possible way to describe the structure of the data), while still being mathematically tractable.

In the case of TimSort, the same idea of presortedness guided the design of an algorithm particularly well suited to partially ordered inputs, a feature that appears to be common in practical applications. As shown later, this behavior can be analyzed by introducing a measure that quantifies presortedness based on Shannon entropy, which allows the traditional worst-case complexity analysis to be refined by incorporating a structural parameter.

1 Record-Biased Permutations

Studying the average running time of algorithms under the uniform model usually gives a quite good understanding of their behavior. However, it is not always clear that the uniform model is relevant, when the algorithm is used on a specific data set. In some cases, the uniform distribution arises by construction, from the randomization of a deterministic algorithm, as in QuickSort when the pivot is chosen uniformly at random, for instance. In other situations, the uniformity assumption may not reflect the data accurately, yet it remains a reasonable first step in modeling it, making the analysis mathematically tractable.

In practical applications where the data consists of sequences of values, it is not uncommon for the input to be partially sorted, depending on its origin. As a result, assuming that the input is uniformly distributed, or deliberately shuffling the input as in the case of QuickSort, may not be appropriate. Indeed, among the new sorting algorithms that have

[1]IPS^{4}o stands for: In-Place Parallel Super Scalar SampleSort.

[2]I began this document using the pronoun "I" to give my own personal perspective on the work presented here. However, as already noted, this work is the result of collaborations. From this point on, I will use "we" to reflect that.

made their way into standard libraries, some are specifically designed to exploit the partial ordering of real-world data, such as TimSort, used in `Python` from 2002 to 2021 and in `Java` since version 7, or PowerSort [MW18], which replaced TimSort in `Python` in 2021. These algorithms are particularly efficient on inputs containing long increasing or decreasing subsequences (see Section 2 for further discussion).

In the context of sorting algorithms, the idea of taking advantage of some bias in the data towards sorted sequences dates back to Knuth [Knu98]. It has been formalized by the notion of *presortedness* [Man85], which quantifies how far a sequence is from being fully sorted (see Section 1.1.4). Given a measure of presortedness m, one can then ask whether a sorting algorithm is optimal for m, that is, whether it minimizes the number of comparisons as a function of both the input size and the value of m. For instance, Knuth's NaturalMergeSort [Knu98] is optimal for the number of runs r, with a worst case running time of $\mathcal{O}(n \log r)$ for an array of length n.

Most measures of presortedness studied in the literature are directly tied to basic statistics on permutations, since inputs to sorting algorithms can be modeled as permutations: only the relative order of elements matters, not their actual values. Consequently, it is natural to define biased distributions on permutations that depend on such statistics, and to analyze classical algorithms under these non-uniform models. *A priori* unrelated to the computer science motivation, the study of non-uniform distributions on permutations has been an active research topic in discrete probability for several decades. Among the most studied models, one can cite the Ewens distribution [Ewe72] and the Mallows distribution [Mal57], both introducing an exponential bias according to a statistic on permutations (specifically, the number of cycles for the Ewens model and the number of inversions for the Mallows model).

We define another non-uniform distribution on permutations, which we call *record-biased* of parameter $\theta > 0$. It also has an exponential bias: this time, the probability of a permutation σ is proportional to $\theta^{\mathrm{rec}(\sigma)}$ where $\mathrm{rec}(\sigma)$ is the number of records (a.k.a. left-to-right maxima) of σ. This model is meaningful in the context of presortedness, which naturally arises when studying algorithms that are designed to be efficient for almost sorted sequences. Moreover, the record-biased distribution is the image of the Ewens distribution under the Foata bijection, a property that can be exploited to derive results on record-biased permutations.

In the second part of this study (see Section 4.1), we present a fine-grained analysis of a simple algorithm under the record-biased distribution. Specifically, we compare two algorithms (one naive and one more clever) for finding both the minimum and the maximum in an array, focusing on the number of mispredictions made by a branch predictor in each case. This analysis helps explain why, in practice, the naive algorithm outperforms the clever one despite performing a lot more comparisons.

We begin by presenting several generative processes for the record-biased distribution on permutations of any given size, and show how they can be used for efficient random generation under this distribution. We then analyze the behavior of classical permutation statistics in this setting. First, we derive explicit formulas for their expectations and examine their behavior in various regimes for the parameter θ, in particular when $\theta = \lambda n$ is linear w.r.t. the size n. We next focus on the regime where θ is constant, establishing the asymptotic distributions of these statistics as the size goes to infinity: three are Gaussian, and the last one follows a beta distribution, asymptotically. The work in [BNP25] also provides the description of the typical limit shape of the diagrams of record-biased permutations, via their *permuton* limit, but this lies beyond the scope of the present study.

This section is based on joint work with Nicolas Auger, Mathilde Bouvel, and Cyril Nicaud [ABNP16, BNP25].

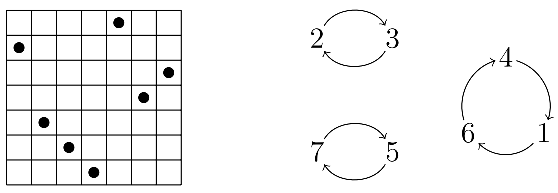


Figure 1: The diagram and the set of cycles representations of $\tau = 6321745$.

1.1 Background and definition of record-biased permutations

We review here some basics on permutations and some known results on the Ewens distribution [Ewe72]. We define record-biased permutations and present the link between Ewens permutations and record-biased permutations through Foata's bijection [Lot97, §10.2].

1.1.1 Permutations

For any integers a and b, let $[a, b] = \{a, \dots, b\}$ and for every integer $n \geq 1$, let $[n] = [1, n]$. By convention $[0] = \emptyset$. If E is a finite set, let $\mathfrak{S}(E)$ denote the set of all permutations on E, i.e., of bijective maps from E to itself. For convenience, we write $\mathfrak{S}([n])$ as $\mathfrak{S}_n$. For a permutation σ in $\mathfrak{S}_n$, the integer n is called the size of σ and is denoted by $|\sigma|$. Permutations in $\mathfrak{S}_n$ can be viewed in several ways (see, e.g., [Bón12]); in what follows, we alternate between their representations as words, diagrams, and sets of cycles.

A permutation σ in $\mathfrak{S}_n$ can be represented as a word $w_1 w_2 \cdots w_n$, where each symbol in $[n]$ appears exactly once, by setting $w_i = \sigma(i)$ for all $i \in [n]$. The *diagram* of a permutation σ of size n is the set of points $(i, \sigma(i))$ in an $n \times n$ grid. An example is given in Fig. 1. A cycle of size k in a permutation $\sigma \in \mathfrak{S}_n$ is a subset $\{i_1, \dots, i_k\}$ of $[n]$ such that $i_1 \overset{\sigma}{\mapsto} i_2 \dots \overset{\sigma}{\mapsto} i_k \overset{\sigma}{\mapsto} i_1$, written as $(i_1, i_2, \dots, i_k)$. Any permutation can be decomposed into its cycles. For instance, the cycle decomposition of $\tau = 6321745$ is $(32)(641)(75)$, as illustrated in Fig. 1.

1.1.2 Ewens model

The Ewens distribution on permutations (see, e.g., [ABT03, Ewe72]) is a generalization of the uniform distribution on $\mathfrak{S}_n$: the probability of a permutation depends on its number of cycles. Let $\mathrm{cyc}(\sigma)$ denote the number of cycles of a permutation σ, the Ewens distribution of parameter θ on $\mathfrak{S}_n$ (where $\theta > 0$) assigns to each $\sigma \in \mathfrak{S}_n$ the probability

$$\frac{\theta^{\mathrm{cyc}(\sigma)}}{\sum_{\rho \in \mathfrak{S}_n} \theta^{\mathrm{cyc}(\rho)}}. \tag{1}$$

Setting $\theta = 1$ yields the uniform distribution on $\mathfrak{S}_n$. The normalization constant $\sum_{\rho \in \mathfrak{S}_n} \theta^{\mathrm{cyc}(\rho)}$ is equal to the *rising factorial* $\theta^{(n)}$, where for any real number x, the rising factorial $x^{(n)}$ is defined by $x^{(n)} = x(x+1) \cdots (x+n-1)$, with $x^{(0)} = 1$. Forgetting the normalizing constant, we sometimes refer to the numerator $\theta^{\mathrm{cyc}(\sigma)}$ in Eq. (1) as the *weight* of the permutation σ; this weight can be interpreted as a product of weights, one from each element of σ. Here, we assign weight θ to each element that is the smallest in its cycle, and weight 1 to all others.

For fixed θ, the expected number of cycles in a random permutation of size n under the Ewens distribution is $\sum_{j=0}^{n-1} \frac{\theta}{\theta+j}$, which is asymptotically equivalent to $\theta \log n$, and it is asymptotically normal.

1.1.3 Generative processes for the Ewens distribution

We call *generative process* a way to incrementally build random permutations that follow a given distribution. They are conveniently depicted by trees, where the branches correspond to the random choices performed during the generation.[3] As we are interested in distributions on $\mathfrak{S}_n$ for every positive n, we consider two kinds of generative processes: either n is known in advance and is used for the construction, or the generative process is in essence infinite and one halts when a size-n permutation is produced.

Generative processes can be directly transformed into random generation algorithms, whose complexity primarily depends on the data structures used throughout the process. They also prove to be very useful for analyzing the behavior of statistics as those studied in Section 1.3, as they describe how a random permutation can be viewed as a sequence of independent random choices. We recall two classical generative processes for the Ewens distribution (see [ABT03] for instance).

The Chinese restaurant process. This works on permutations expressed as sets of cycles and was originally designed for uniform random permutations. A straightforward variant of this process can be used to derive any permutation according to the Ewens distribution.

Starting from the empty permutation σ_0 at step 0, a random permutation $\sigma_n \in \mathfrak{S}_n$ is constructed after n steps. At each step $i \geq 1$, the new permutation σ_i is obtained by modifying σ_{i-1} as follows. First, set $\sigma_i(j) = \sigma_{i-1}(j)$ for $j \in [i-1]$. Then, insert i as described below, all the random choices being independent:

- with probability $\frac{\theta}{\theta+i-1}$, create a new cycle containing i only, i.e., set $\sigma_i(i) = i$;
- with probability $\frac{i-1}{\theta+i-1}$, insert i at a uniformly chosen position (among the $i-1$ available across all existing cycles). Equivalently, for each $j \in [i-1]$, with probability $\frac{1}{\theta+i-1}$, set $\sigma_i(i) = j$ and $\sigma_i(\sigma_{i-1}^{-1}(j)) = i$, that is, insert i before j in its cycle.

This process is infinite and does not depend on n. However, stopping after n steps produces a random permutation of size n according to the Ewens distribution of parameter θ. Indeed, with the notation $c_\sigma(i) = 1$ if the i is the minimum in its cycle in σ and $c_\sigma(i) = 0$ otherwise, it generates the permutation $\sigma \in \mathfrak{S}_n$ with probability

$$\prod_{i=1}^{n} \frac{\theta^{c_\sigma(i)}}{i-1+\theta} = \frac{\theta^{\mathrm{cyc}(\sigma)}}{\theta^{(n)}}.$$

This process can be represented by a tree, which is also infinite in this case. Its root, at height 0, is the empty permutation and has one child: the permutation consisting of a single cycle containing 1. Then, each node at height i has $i+1$ children obtained as described above.

Trimming the tree at height n creates leaves corresponding to all permutations of size n. For example, Fig. 2 shows the tree for $\mathfrak{S}_3$. In this figure (and similar ones later), the weight of each permutation (a leaf of the tree) is shown in red. As explained earlier, it is the product of the element weights, indicated in blue on the edges corresponding to each element insertion.

The Feller coupling. The *Feller coupling* is another classical generative process that can be used for the Ewens distribution. Contrary to the Chinese restaurant process, the size n has to be known in advance, and there is one (finite) tree associated with each n. This generative process is described as follows. At any given step, except for the initialization, the partial

[3]This is similar to the way algorithms are translated into decision trees when establishing lower bounds in complexity, but in addition with probabilities (or weights) on the edges of the trees.

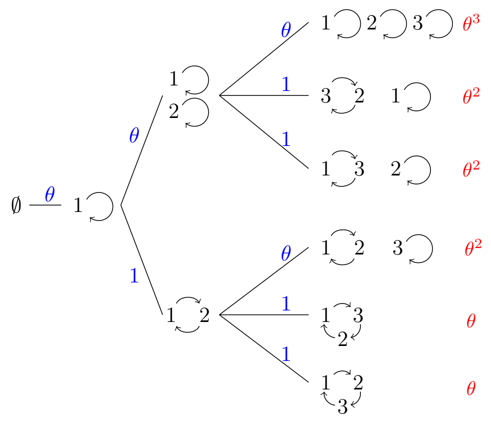


Figure 2: the Chinese Restaurant process for permutations in $\mathfrak{S}_3$ in the Ewens model.

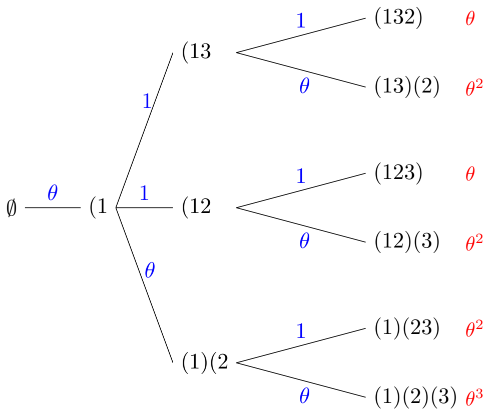


Figure 3: Feller coupling for permutations in $\mathfrak{S}_3$ in the Ewens model.

representation of the permutation consists of a set of cycles and a sequence, called the *open cycle*, which represents the current cycle under construction. The other cycles are complete: they remain untouched in the sequel and will be cycles of the generated permutation at the end. At each step i, we perform an (independent) random choice as follows:

- with probability $\frac{\theta}{\theta+n-i}$, close the open cycle, adding the newly formed cycle to the list, and start a new open cycle containing the smallest unused value;
- for each unused value $j \in [n]$, with probability $\frac{1}{\theta+n-i}$, add j at the end of the open cycle.

The process starts with no cycle and no open cycle, so the first step deterministically produces a open cycle containing only the value 1.

In the tree representation shown in Fig. 3, cycles are written in the order of their creation, which naturally orders them by increasing values of their smallest element. The open cycle is written using a single open parenthesis; for example, (13 represents the open cycle $1 \to 3 \to$. Each edge is labeled with a weight, either θ or 1, depending on the chosen action. One can readily verify that the probability of generating a given size-n permutation σ is:

$$\prod_{i=1}^{n} \frac{\theta^{c_\sigma(i)}}{n-i+\theta} = \frac{\theta^{\mathrm{cyc}(\sigma)}}{\theta^{(n)}}.$$

where $c_\sigma(i) = 1$ if i is the minimum in its cycle in σ and $c_\sigma(i) = 0$ otherwise.

1.1.4 Records and record-biased permutations

For a permutation $\sigma \in \mathfrak{S}_n$ and for $i \in [n]$, there is a *record* at position i in σ (and $\sigma(i)$ is called a record) if $\sigma(i) > \sigma(j)$ for all $j \in [i-1]$. The number of records in σ is denoted by $\mathrm{rec}(\sigma)$. We refer to any element that is not a record as a *non-record*. For permutations represented as words, records are elements that have no larger elements to their left. And in the diagrams of a permutation, a record is an element that have no higher element to its left.

Foata's bijection: the link between cycles and records [Lot97] Also called the *fundamental bijection*, it is a bijection from $\mathfrak{S}_n$ to $\mathfrak{S}_n$ that maps $\mathrm{cyc}(\sigma)$ to $\mathrm{rec}(\sigma)$, for any permutation σ. This bijection φ is the following transformation:

1. given σ a permutation of size n, consider the cycle decomposition of σ;

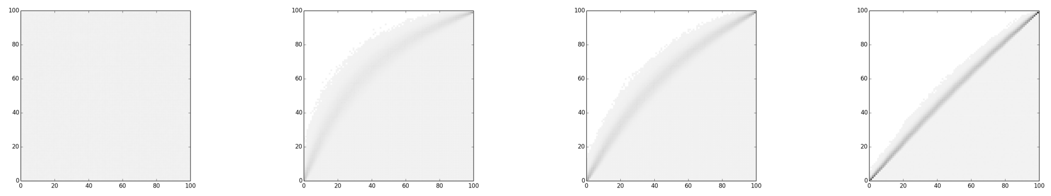


Figure 4: Random record-biased permutations. From left to right, $\theta = 1$ (uniform), $50, 100$, and 500. For each diagram, the darkness of a point (i, j) is proportional to the number of permutations σ such that $\sigma(i) = j$, for a sampling of 10000 permutations of size 100.

2. write every cycle starting with its maximal element, and write the cycles in increasing order of their maximal (i.e., first) element;
3. erase the parenthesis to get the word representation of $\varphi(\sigma)$.

For instance $\varphi\big((57)(32)(416)\big) = 3264175$. This transformation is a bijection, with each cycle of σ corresponding to a record in $\varphi(\sigma)$, and vice versa.

Our model: record-biased permutations In a similar fashion as for Ewens distribution, we consider the non-uniform distribution of parameter θ that gives the probability

$$\frac{\theta^{\mathrm{rec}(\sigma)}}{\sum_{\rho \in \mathfrak{S}_n} \theta^{\mathrm{rec}(\rho)}}$$

to any permutation $\sigma \in \mathfrak{S}_n$. We call them *record-biased* permutations. We define the weight of a permutation σ as $\mathrm{w}(\sigma) = \theta^{\mathrm{rec}(\sigma)}$. Like for the Ewens distribution, the normalization constant $\sum_{\rho \in \mathfrak{S}_n} \theta^{\mathrm{rec}(\rho)}$ is equal to the rising factorial $\theta^{(n)}$.

Foata's bijection provides a very tight connection between Ewens permutations and record-biased permutations. Indeed, σ is a random Ewens permutation in $\mathfrak{S}_n$ if and only $\varphi(\sigma)$ is a random record-biased permutation in $\mathfrak{S}_n$. Figure 4 shows random record-biased permutations, for various values of θ.

The number of non-records as a measure of presortedness The concept of presortedness, formalized by Mannila [Man85], naturally arises in the study of sorting algorithms that efficiently handle nearly sorted sequences. We denote by $E^\star$ the set of all nonempty sequences of distinct elements of a totally ordered set E, and by $\cdot$ the concatenation on $E^\star$. A mapping m from $E^\star$ to $\mathbb{N}$ is is called a *measure of presortedness* if it satisfies:

1. if $X \in E^\star$ is sorted, then $m(X) = 0$;
2. if $X = (x_1, \cdots, x_\ell)$ and $Y = (y_1, \cdots, y_\ell)$ are sequences in $E^\star$ of the same length, and for every $i, j \in [\ell]$, $x_i < x_j \Leftrightarrow y_i < y_j$, then $m(X) = m(Y)$;
3. if X is a subsequence of Y, then $m(X) \leq m(Y)$;
4. if every element of X is smaller than every element of Y, then $m(X \cdot Y) \leq m(X) + m(Y)$;
5. for every symbol $a \in E$ not occurring in X, $m(a \cdot X) \leq |X| + m(X)$.

Classical measures of presortedness include the number of inversions and the number of ascending runs. One can easily verify, by checking conditions 1 to 5, that the number of non-records in a sequence s, is a valid measure of presortedness for sequences of distinct integers. Note that, due to condition 2, restricting to permutations in $\mathfrak{S}_n$ is equivalent to studying sequences of distinct integers. Given a measure of presortedness m, one can define the corresponding notion of m-optimality for sorting algorithms. For example, Knuth's NaturalMergeSort is optimal with respect to the number of runs (see [Man85, PM95] for details).

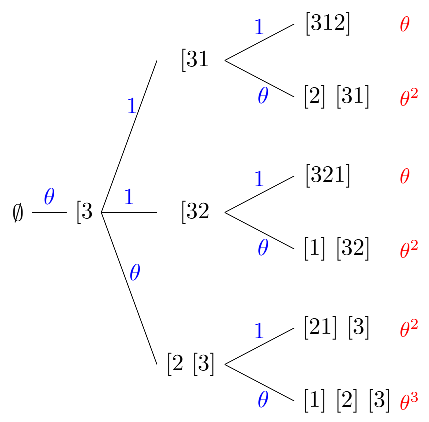


Figure 5: Generative process for record-biased permutations in $\mathfrak{S}_3$, viewed as sequences of sequences.

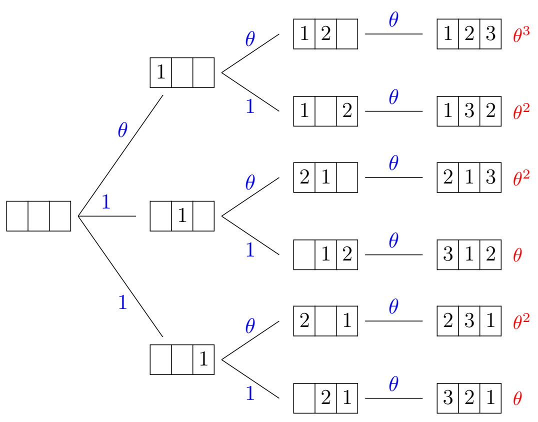


Figure 6: Generative process for record-biased permutations in $\mathfrak{S}_3$, viewed as words.

1.2 Generative processes for record-biased permutations

As in Section 1.1.3, we introduce generative processes for record-biased permutations, which will be used to compute expected values and derive limit laws for several statistics of interest.

1.2.1 Two generative processes for the word representation

The *Feller coupling* presented above can readily be transformed into a generative process for record-biased permutations. The modifications are the following:

- we work directly on sequences of values concatenated at the end, instead of cycles;
- when a new open sequence is created, it contains the largest available value;
- the sequences are created from right to left.

This construction directly ensures that the largest value of each sequence is a record. The tree associated with this generative process on $\mathfrak{S}_3$ is depicted in Fig. 5.

The tree depicted in Fig. 6 represents a different generative process for record-biased permutations viewed as sequences. Given the target size n, the idea is to start with a sequence made of n empty slots, and then to incrementally place the values ranging from 1 to n, one by one. At each step i an independent random choice is performed:

- with probability $\frac{\theta}{\theta+n-i}$, the value i is placed at the leftmost empty slot;
- for every index j in the array which corresponds to an empty slot but the leftmost one, i is placed at the j-th slot with probability $\frac{1}{\theta+n-i}$.

The construction is readily seen to associate with each permutation σ the probability $\theta^{\mathrm{rec}(\sigma)}/\theta^{(n)}$ which characterizes record-biased permutations.

1.2.2 One generative process for diagrams

For permutations viewed as diagrams, one can also design a generative process, starting with an empty diagram and inserting points from left to right. Note that this generative process does not require the target size n to be known in advance.

The root of the tree is the empty diagram. At each step i, we add a new row and a new column in the diagram, and we put the new point at their crossing. The added column will

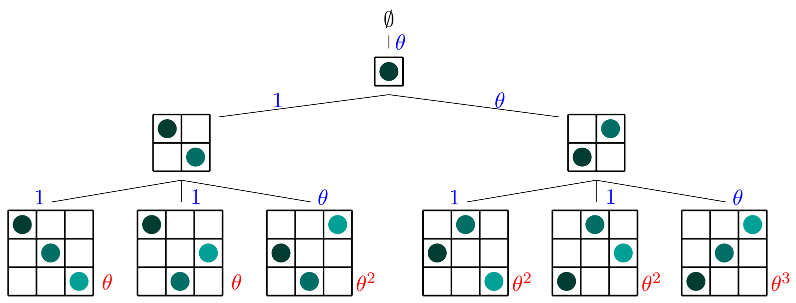


Figure 7: Generative process for record-biased permutations in $\mathfrak{S}_3$, viewed as diagrams.

always be the rightmost (i.e., the i-th column). The placement of the point in the newly added column is chosen randomly and independently at each step i:

- it is on the highest row with probability $\frac{\theta}{\theta+i-1}$;
- every other possibility has probability $\frac{1}{\theta+i-1}$.

The associated tree is depicted on Fig. 7, where the weights on the edges are such that the total weight of a permutation σ, i.e., the product of the weights along the path from the root to the diagram of σ, is precisely $\sigma^{\mathrm{rec}(\sigma)}$.

1.2.3 Random generation

Among other advantages, the generative processes described above translate naturally into random samplers, making it possible to formulate conjectures and test algorithms. We briefly explain how to construct efficient samplers (see [BNP25] for details).

As noted in [Fér13], it is easy to obtain a linear time and space algorithm for random permutations according to the classical Ewens distribution, using the variant of the Chinese restaurant process. Applying Foata's bijection, this yields a linear random sampler for permutations according to the record-biased distribution (see [ABNP16] for details).

Building on the same ideas, we can design linear samplers that generate record-biased permutations directly, based on our three generative processes, provided each step can be implemented in constant time. Here, we assume that it is possible to sample a uniform real number in $[0, 1]$ in constant time, which in turn allows us to generate a random integer in $[1, n]$ in constant time as well.[4]

Sequence representation To implement a linear-time sampler following the process in Fig. 6, we maintain a set S, initially the interval $[n]$, representing the positions still empty in the array σ and we must be able to perform the following operations in constant time:

(i) remove the minimum from S, which will be needed when inserting a record in σ;

(ii) choose uniformly at random one value different from the minimum in S, and remove it from S, which will be needed when inserting a non-record in σ.

This is achieved by maintaining three linear-space data-structures: a linked list L, and two arrays A and $invA$. Both L and A store the positions of σ which still need to be filled (i.e., the set S), but with different structures to support different operations efficiently. The array $invA$ stores the functional inverse of A and allows constant-time lookup of the position

[4] An analysis for a more accurate complexity model is doable, e.g., considering that we can only generate uniform random bits in constant time, but it is not the topic of the present study.

of any given value in A. The data structure L is a doubly linked list of the values in the set S, in increasing order. Its cells are kept in an array such that the cell containing the value i is located at index i, which gives direct access to any value. This setup provides efficient retrieval of the minimum (always the first element in the list), and any value can be removed from the set S in constant time by updating the pointers in L. Meanwhile, the array A keeps track of all positions still empty in σ, excluding the current minimum. This array is maintained contiguous upon removals by filling any gap with the last element (and updating $invA$), which allows to choose the position of a non-record uniformly at random in constant time. When the minimum is needed, its value is obtained from L, and its index k in A can be found in constant time using $invA$.

The generative process of Fig. 5 can be transformed into a linear sampler for record-biased permutations, using almost the same data structure as above, replacing the minimum by the maximum. The only slight difference is that the maximum of the remaining values can be picked for action (ii). This is easy to implement by keeping the maximum in A and removing it (instead of the next maximum) from A during action (i). The sequence of sub-sequences is a simple list of lists in which insertions can be performed in constant time. Flattening the list in the end yields another linear time sampler for record-biased permutations.

Diagram representation To get a linear time sampler following the process of Section 1.2.2, we first note that choosing the height of the new row for a non-record is equivalent to choosing uniformly at random a column j and placing the new row just under the point in column j. Thus, it is enough to maintain a linked list of the indices of the columns ordered by decreasing height of the point they contain, starting with the highest. When a non-record point is added, its height is determined by choosing a column in $[i-1]$. At each step, the insertion of the index i of the last column into the list can also be done in constant time, provided we use a linked list with direct access to its cells, as we did before. Viewed as a word, the resulting list (in reverse order) corresponds to σ^{-1}, from which σ can be recovered in linear time.

1.3 Behavior of some classical statistics

In this section, we study the behavior of several statistics on record-biased permutations with parameter θ, which may be either a fixed positive real number or a function of the size n. These results can be derived almost directly from the generative processes described in Section 1.2.

We use the properties of the so-called *digamma* function [OLBC10]. The digamma function is defined by $\Psi(x) = \Gamma'(x)/\Gamma(x)$, with Γ the classical gamma function generalizing the factorial. It satisfies the recurrence $\Psi(x+1) = \Psi(x) + \frac{1}{x}$ which leads to the identity

$$\sum_{i=0}^{n-1} \frac{1}{x+i} = \Psi(x+n) - \Psi(x). \tag{2}$$

As $x \to +\infty$, it admits the asymptotic expansion $\Psi(x) = \log(x) - \frac{1}{2x} - \frac{1}{12x^2} + o\left(\frac{1}{x^2}\right)$.

1.3.1 Four statistics and their expectations

Throughout this section, we denote by $\mathbb{E}_n[\cdot]$ the expectation of a random variable on record-biased permutations of size n for the parameter θ.

Number of records We naturally begin by examining how the value of θ affects the expected number of records.

Theorem 1. *Among record-biased permutations of size n for the parameter θ, for any $i \in [n]$, the probability that there is a record at position i is:* $\mathbb{P}_n(\textit{record at } i) = \frac{\theta}{\theta+i-1}$.

Proof. A record occurs at position i if and only if the point inserted in the i-th column by the process of Section 1.2.2 is the highest so far, which happens with probability $\frac{\theta}{\theta+i-1}$. □

Corollary 2. *Among record-biased permutations of size n for the parameter θ, the expected value of the number of records is:* $\mathbb{E}_n[\text{rec}] = \sum_{i=1}^{n} \frac{\theta}{\theta+i-1} = \theta(\Psi(\theta+n) - \Psi(\theta))$.

Proof. The expression of $\mathbb{E}_n[\text{rec}]$ is simply obtained summing over i the result of Theorem 1, and applying Eq. (2) for the right hand term. □

Observe that this can also be recovered from known results on the Ewens distribution. Indeed, as we have seen in Section 1.1.4, the record-biased distribution on $\mathfrak{S}_n$ is the image of the Ewens distribution on $\mathfrak{S}_n$ by the fundamental bijection φ mapping cycles to records. Consequently, Corollary 2 is a consequence of the well-known expectation of the number of cycles under the Ewens distribution.

Number of descents Recall that a permutation σ of $\mathfrak{S}_n$ has a *descent* at position $i \in \{2, \dots, n\}$ if $\sigma(i-1) > \sigma(i)$. We denote by $\text{desc}(\sigma)$ the number of descents in σ; we are interested in descents as they are directly related to the number of increasing runs in a permutation: every run except the last one is immediately followed by a descent, and conversely. Some merge-based sorting algorithms, such as Knuth's NaturalMergeSort, begin by decomposing the input into such runs.

Theorem 3. *Among record-biased permutations of size n for the parameter θ, $\forall i \in \{2, \dots, n\}$, the probability that there is a descent at position i is:* $\mathbb{P}_n(\textit{descent at } i) = \frac{(i-1)(2\theta+i-2)}{2(\theta+i-1)(\theta+i-2)}$.

Proof (sketch). For a permutation σ of size n and any $a \leq n$, we denote by $\text{norm}(\sigma(a))$ the rank of $\sigma(a)$ in $\sigma(1)\sigma(2)\dots\sigma(a)$, that is to say $\text{norm}(\sigma(a)) = j$ exactly when $\sigma(a)$ is the j-th smallest element in $\{\sigma(1), \sigma(2), \dots, \sigma(a)\}$. Splitting according to the possible values of $\text{norm}(\sigma(i-1))$ and $\text{norm}(\sigma(i))$, we have

$$\mathbb{P}_n(\text{descent at } i) = \sum_{j=1}^{i-1} \sum_{k=1}^{j} \mathbb{P}_n\big(\text{norm}(\sigma(i-1)) = j \text{ and } \text{norm}(\sigma(i) = k)\big).$$

From the generative process for diagrams of Section 1.2.2, we know that the rank of $\sigma(i-1)$ is independent from the rank of $\sigma(i)$, and the probabilities of these ranks are explicit. This makes it possible to compute the probability that there is of a descent at position i. □

Corollary 4. *Among record-biased permutations of size n for the parameter θ, the expected value of the number of descents is:* $\mathbb{E}_n[\text{desc}] = \frac{n(n-1)}{2(\theta+n-1)}$.

Proof. The partial fraction decomposition of the formula for $\mathbb{P}_n\big(\sigma(i-1) > \sigma(i)\big)$ in Theorem 3 gives: $2\mathbb{P}_n(\sigma(i-1) > \sigma(i)) = 1 + \frac{\theta(\theta-1)}{\theta+i-1} - \frac{\theta(\theta-1)}{\theta+i-2}$. Hence, we have a telescopic series when summing for i from 2 to n, which yields: $2\mathbb{E}_n[\text{desc}] = (n-1) + \frac{\theta(\theta-1)}{\theta+n-1} - \frac{\theta(\theta-1)}{\theta}$. This gives the announced expression for $\mathbb{E}_n[\text{desc}]$ after some elementary simplifications. □

Number of inversions An *inversion* in a permutation $\sigma \in \mathfrak{S}_n$ is a pair $(i, j) \in [n] \times [n]$ such that $i < j$ and $\sigma(i) > \sigma(j)$. In the word representation of permutations, this corresponds to a pair where the larger element appears to the left of the smaller one. For any $\sigma \in \mathfrak{S}_n$, let $\mathrm{inv}(\sigma)$ denote the number of inversions in σ, and for any $j \in [n]$, define $\mathrm{inv}_j(\sigma)$ as the number inversions of the form (i, j). Formally, $\mathrm{inv}_j(\sigma) = \big|\{i \in [j-1] : (i, j) \text{ is an inversion of } \sigma\}\big|$.

Note that the number of inversions is directly related to the complexity of InsertionSort (see, e.g., [CLRS09]). This algorithm maintains a sorted prefix of its input: at each step $i \in 2, \ldots, n$, the first $i-1$ elements are already sorted, and the i-th element is inserted into its correct position through a sequence of adjacent swaps. It is well known that the number of swaps performed on a permutation σ is exactly its number of inversions. Moreover, the number of comparisons $C(\sigma)$ satisfies the inequality $\mathrm{inv}(\sigma) \le C(\sigma) \le \mathrm{inv}(\sigma) + n - 1$.

Theorem 5. *Among record-biased permutations of size n for the parameter θ, for any $j \in [n]$ and $k \in [0, j-1]$, the probability that there are k inversions of the form (i, j) is: $\mathbb{P}_n\big(\mathrm{inv}_j(\sigma) = k\big) = \frac{1}{\theta+j-1}$ if $k \neq 0$ and $\mathbb{P}_n\big(\mathrm{inv}_j(\sigma) = k\big) = \frac{\theta}{\theta+j-1}$ if $k = 0$.*

Proof. By definition, the value of $\mathrm{inv}_j(\sigma)$ depends only on how $\sigma(j)$ compares to the $\sigma(i)$ for $i < j$. More precisely, $\mathrm{inv}_j(\sigma) = k$ if and only if $\sigma(j)$ is the $(j-k)$-th largest element of σ among the first j. From the generative process of Section 1.2.2, the probability that $\sigma(j)$ is the largest element of σ among the first j is $\frac{\theta}{\theta+j-1}$, and this proves the statement of the theorem in the case $k = 0$. And for any value $k \neq 0$ with $k < j$, the probability that $\sigma(j)$ is the $(j-k)$-th largest element of σ among the first j is $\frac{1}{\theta+j-1}$, concluding the proof. □

Corollary 6. *Among record-biased permutations of size n for the parameter θ, the expected value of the number of inversions is: $\mathbb{E}_n[\mathrm{inv}] = \frac{n(n+1-2\theta)}{4} + \frac{\theta(\theta-1)}{2}\left(\Psi(\theta+n) - \Psi(\theta)\right)$.*

Proof. We first compute $\mathbb{E}_n[\mathrm{inv}_j]$ for any $j \in [n]$. With Theorem 5 and noting that the maximum possible value of inv_j is $j-1$, we have

$$\mathbb{E}_n[\mathrm{inv}_j] = \sum_{k=0}^{j-1} k \cdot \mathbb{P}_n(\mathrm{inv}_j = k) = \sum_{k=1}^{j-1} k \cdot \frac{1}{\theta+j-1} = \frac{j(j-1)}{2(\theta+j-1)}. \tag{3}$$

Consequently, $\mathbb{E}_n[\mathrm{inv}] = \sum_{j=1}^{n} \mathbb{E}_n[\mathrm{inv}_j] = \sum_{j=1}^{n} \frac{j(j-1)}{2(\theta+j-1)}$. Using the properties of the digamma function Ψ reviewed at the beginning of Section 1.3, and noting that $\frac{j(j-1)}{\theta+j-1} = j - \theta + \frac{\theta(\theta-1)}{\theta+j-1}$, this sum may be expressed as $\mathbb{E}_n[\mathrm{inv}] = \frac{n(n+1-2\theta)}{4} + \frac{\theta(\theta-1)}{2}\left(\Psi(\theta+n) - \Psi(\theta)\right)$. □

As a direct consequence of Corollary 6 and the asymptotic estimates of the fourth row of Table 1 (see below), we get the expected running time of InsertionSort:

Corollary 7. *Applied to record-biased permutations of size n for the parameter $\theta = \mathcal{O}(n)$, the expected running time of* InsertionSort *is $\Theta(n^2)$, like under the uniform distribution. If $\theta = n^\delta$ with $1 < \delta < 2$, it is $\Theta(n^{3-\delta})$. If $\theta = \Omega(n^2)$, it is $\Theta(n)$.*

First value Naturally, the behavior of the first value $\sigma(1)$ in record-biased permutations differs completely from the three statistics studied above.

Theorem 8. *Among record-biased permutations of size n for the parameter θ, for any $k \in [n]$, the probability that a permutation starts with k is: $\mathbb{P}_n(\sigma(1) = k) = \frac{(n-1)!\,\theta^{(n-k)}\theta}{(n-k)!\,\theta^{(n)}}$.*

	$\theta = 1$ (uniform)	fixed $\theta > 0$	$\theta = n^\varepsilon$, $0 < \varepsilon < 1$	$\theta = \lambda n$, $\lambda > 0$	$\theta = n^\delta$, $\delta > 1$	See Cor.
$\mathbb{E}_n[\text{rec}]$	$\log n$	$\theta \cdot \log n$	$(1-\varepsilon) \cdot n^\varepsilon \log n$	$\lambda \log(1+1/\lambda) \cdot n$	n	2
$\mathbb{E}_n[\text{desc}]$	$n/2$	$n/2$	$n/2$	$n/2(\lambda+1)$	$n^{2-\delta}/2$	4
$\mathbb{E}_n[\text{inv}]$	$n^2/4$	$n^2/4$	$n^2/4$	$n^2/4 \cdot f(\lambda)$	$n^{3-\delta}/6$	6
$\mathbb{E}_n[\sigma(1)]$	$n/2$	$n/(\theta+1)$	$n^{1-\varepsilon}$	$(\lambda+1)/\lambda$	1	9

Table 1: Asymptotic behavior in expectation of some permutation statistics on record-biased permutations for the parameter θ. We use the shorthand $f(\lambda) = 1 - 2\lambda + 2\lambda^2 \log(1 + 1/\lambda)$. All the results in this table are asymptotic equivalents.

Proof. The statement is proved using the generative process of Section 1.2.1 for permutations viewed as words (Fig. 6). In this process, the event that the first element equals k corresponds to the first $k-1$ insertions (of the values 1 through $k-1$) being non-records, followed by the k-th insertion (of the value k) being a record. This happens with the following probability:

$$\mathbb{P}_n(\sigma(1) = k) = \prod_{i=1}^{k-1} \frac{n-i}{\theta+n-i} \cdot \frac{\theta}{\theta+n-k} = \frac{(n-1)!\,\theta^{(n-k)}\theta}{(n-k)!\theta^{(n)}}. \qquad \square$$

Note that the above theorem was also proved by B. Corsini in [Cor22]. Both our and his proof rely on the generative process of Section 1.2.1.

Corollary 9. *Among record-biased permutations of size n for the parameter θ, the expected value of the first element of a permutation is:* $\mathbb{E}_n[\sigma(1)] = \frac{\theta+n}{\theta+1}$.

Proof. From Theorem 8, we have

$$\mathbb{E}_n[\sigma(1)] = \sum_{k=1}^{n} k\,\mathbb{P}_n(\sigma(1) = k) = \frac{(n-1)!\theta}{\theta^{(n)}} \sum_{k=1}^{n} k \frac{\theta^{(n-k)}}{(n-k)!} = \frac{(n-1)!\theta}{\theta^{(n)}} \sum_{j=0}^{n-1} (n-j) \frac{\theta^{(j)}}{j!}.$$

By the binomial formula, $[z^m](1-z)^{-\alpha} = \frac{\alpha^{(m)}}{m!}$, where as usual $[z^m]F(z)$ denotes the coefficient of z^m in the series expansion of $F(z)$. Observe also that $(n-j) = [z^{n-j-1}](1-z)^{-2}$. Therefore

$$\mathbb{E}_n[\sigma(1)] = \frac{(n-1)!\theta}{\theta^{(n)}} \sum_{j=0}^{n-1} [z^{n-j-1}](1-z)^{-2} \cdot [z^j](1-z)^{-\theta} = \frac{(n-1)!\theta}{\theta^{(n)}} \frac{(\theta+2)^{(n-1)}}{(n-1)!},$$

which yields the announced result after elementary simplifications. $\square$

Different regimes for different θ We compute asymptotic equivalents of the expectations derived above for different regimes of θ: constant, sublinear $\theta = n^\varepsilon$ with $\varepsilon \in (0,1)$, linear $\theta = \lambda n$ for $\lambda > 0$ and superlinear $\theta = n^\gamma$ with $\gamma > 1$. The computations are straightforward, using the asymptotic behavior of Ψ when necessary. The results are summarized in Table 1.

1.3.2 Limit laws for fixed θ

In this section, and only here, we consider the case where $\theta > 0$ is fixed. Under this classical assumption, the number of records, the number of descents and the number of inversions

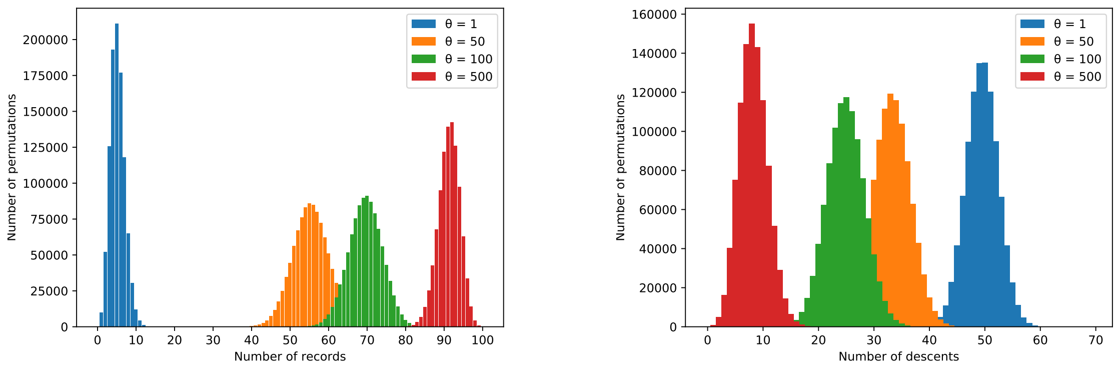


Figure 8: Empirical distribution of the number of records (left) and the number of descents (right) in 10^6 record-biased permutations of size 100 for several values of θ.

are asymptotically Gaussian, whereas the first value tends to a beta distribution of parameters $(1, \theta)$. For the first two results, we rely on known properties of the Ewens distribution. These convergences are clearly illustrated by simulations using the random generator described in Section 1.2.3, as shown in Figs. 8 and 9.

As usual, the notation $\mathcal{N}(0,1)$ below denotes the normal distribution with mean 0 and variance 1, and $\xrightarrow{d}$ denotes the convergence in distribution.

Theorem 10. *The number of records in record-biased permutations is asymptotically normal. More precisely, letting R_n be the random variable which denotes the number of records in a record-biased permutation of size n for any fixed value of the parameter θ, we have*

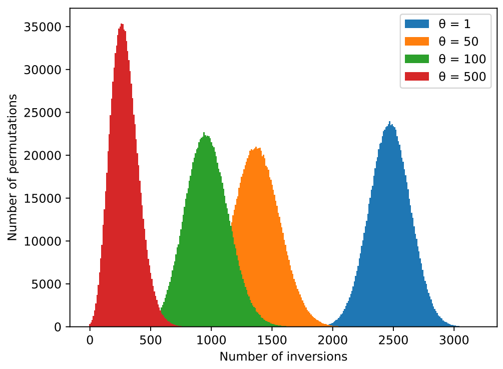
$$\frac{R_n - \theta\log(n)}{\sqrt{\theta\log(n)}} \xrightarrow{d} \mathcal{N}(0,1).$$

Proof. As before, it follows immediately from the distribution of the number of cycles in Ewens permutations. □

Theorem 11. *The number of descents in record-biased permutations is asymptotically normal. More precisely, letting D_n be the random variable which denotes the number of descents in a record-biased permutation of size n for any fixed value of the parameter θ, we have*

$$\frac{D_n - n/2}{\sqrt{n/12}} \xrightarrow{d} \mathcal{N}(0,1).$$

Proof (sketch). Similarly to cycles, the number of descents in record-biased permutations of size n is mapped by the fundamental bijection φ^{-1} to the number of anti-excedances in Ewens permutations of size n. An *anti-excedance* of $\sigma \in \mathfrak{S}_n$ is an index $i \in [n]$ such that $\sigma(i) < i$. The fact that descents of σ are equinumerous with anti-excedances of $\varphi^{-1}(\sigma)$ is a simple adaptation of a proof from [Bón12]. A *weak excedance* of $\sigma \in \mathfrak{S}_n$ is an index $i \in [n]$ such that $\sigma(i) \geq i$. Clearly, the numbers of anti-excedances and of weak excedances in a permutation of size n sum up to n, so results for one translate directly to the other. Since the distribution of D_n matches that of the number of anti-excedances in Ewens permutations, D_n is therefore distributed like $n - W_n$, where W_n denotes the number of weak excedances in a random Ewens permutation of size n. As W_n is asymptotically normal [Fér13], the same holds for D_n. As shown in Corollary 4, the expectation of D_n is asymptotically $n/2$, and its variance, which equals that of W_n, can be computed from the results of [Fér13]. □



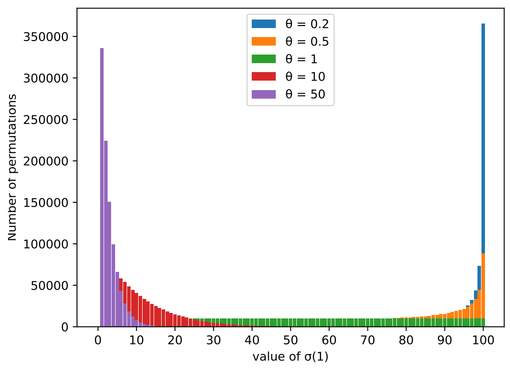


Figure 9: Empirical distributions of the number of inversions (left) and the value of $\sigma(1)$ (right) in 10^6 record-biased permutations of size 100, for several values of θ.

Theorem 12. *The number of inversions in record-biased permutations is asymptotically normal. More precisely, letting I_n be the random variable which denotes the number of inversions in a record-biased permutation of size n for any fixed value of the parameter θ, we have*

$$\frac{I_n - n^2/4}{\sqrt{n^3/36}} \xrightarrow{d} \mathcal{N}(0,1).$$

Proof (sketch). Fix a positive real number θ. From Section 1.2.2, we observe that, for σ a record-biased permutation of size n for the parameter θ, the random variables $\mathrm{inv}_j(\sigma)$ and $\mathrm{inv}_{j'}(\sigma)$ are independent for $j \neq j'$. Therefore I_n is the sum of the independent random variables $\mathrm{inv}_j(\sigma)$ for j from 1 to n, whose distributions are given by Theorem 5. Recall from the proof of Corollary 6 that $\mathbb{E}_n[\mathrm{inv}_j] = \frac{j(j-1)}{2(\theta+j-1)}$, which implies $\mathbb{E}_n[I_n] \sim n^2/4$. Similarly,

$$\mathbb{E}_n[\mathrm{inv}_j^2] = \sum_{k=0}^{j-1} k^2 \cdot \mathbb{P}_n(\mathrm{inv}_j = k) = \sum_{k=1}^{j-1} k^2 \cdot \frac{1}{\theta+j-1} = \frac{j(j-1)(2j-1)}{6(\theta+j-1)}.$$

Denoting $\mathbb{V}_n(\cdot)$ the variance of a random variable on record-biased permutations of size n, it follows that $\mathbb{V}_n(\mathrm{inv}_j) = \frac{j(j-1)(j^2+(4\theta-3)j+2-2\theta)}{12(\theta+j-1)^2}$, so that $\mathbb{V}_n(I_n) \sim n^3/36$ as $n \to \infty$.

Using Theorem 27.3 in [Bil12] it suffices to show that

$$\sum_{j=1}^{n} \frac{1}{v}\mathbb{E}_n[\mathrm{inv}_j^3] \to 0 \text{ as } n \to \infty, \qquad \text{where } v = \sqrt{\sum_{j=1}^n \mathbb{V}_n(\mathrm{inv}_j)}^3.$$

With similar computations as before, we find that $\mathbb{E}_n[\mathrm{inv}_j^3]$ is of order j^3, so $\sum_{j=1}^n \mathbb{E}_n[\mathrm{inv}_j^3]$ is of order n^4. And, since $v = \mathbb{V}_n(I_n)^{3/2}$ is of order $n^{9/2}$, the result follows. $\square$

Theorem 13. *Fix θ a positive real number. For σ a record-biased permutation of size n for the parameter θ, the random variable $\sigma(1)/n$ is asymptotically distributed following a beta distribution of parameters $(1,\theta)$.*

Proof (sketch). For any positive integer r, the r-th moment of a beta distribution of parameters $(1,\theta)$ is $\frac{r!}{(\theta+1)^{(r)}}$ (see, e.g., [JKB95]). In particular, the associated moment generating function has positive radius of convergence, so that the beta distribution is determined by its moments [Bil12]. Therefore, it is enough to compute the limits of the moments of $\sigma(1)/n$, for σ a record-biased permutation of size n for the parameter θ, and to observe that these limits are indeed $\frac{r!}{(\theta+1)^{(r)}}$.

For any $r > 0$, we denote by $A_r(z) = a_{r,0}+a_{r,1}z+\cdots+a_{r,r-1}z^{r-1}$ the Eulerian polynomial of degree $r-1$ (their properties can be found, e.g., in [Pet15]). First, $a_{r,i}$ is the number of permutations of size r with i descents, so that we have $a_{r,0}+a_{r,1}+\cdots+a_{r,r-1} = r!$. Second, the following identity (known as the Carlitz identity) holds, for any $r > 0$: $\sum_n n^r z^n = \frac{zA_r(z)}{(1-z)^{r+1}}$. This allows us to compute the r-th moment of $\sigma(1)$, for $r > 0$ (see [BNP25] for the details):

$$\mathbb{E}_n[\sigma(1)^r] = \sum_{j=1}^{r} a_{r,j-1}\frac{n^r(1+o(1))}{(\theta+1)^{(r)}} \text{ so that } \mathbb{E}_n[\sigma(1)^r] \sim_{n\to\infty} \frac{r!n^r}{(\theta+1)^{(r)}}.$$

Therefore, dividing by n^r, the r-th moment of $\sigma(1)/n$ coincides with that of a beta distribution of parameters $(1,\theta)$, concluding the proof. □

1.3.3 Conclusion on record-biased permutations

We introduced a non-uniform distribution on permutations, biased by their number of records, inspired by the classical Ewens model. Since the number of non-records serves as a measure of presortedness, this provides a meaningful model for analyzing algorithms on partially sorted data, potentially offering a more realistic alternative to the uniform distribution.

We studied the behavior of several classical permutation statistics, such as the number of records, inversions, and descents, for record-biased permutations. We derived explicit formulas for the expectations of these statistics and analyzed their behavior across different regimes for the parameter θ, in particular when $\theta = \lambda n$ is linear w.r.t. the size n. In the regime where θ is constant (corresponding to a logarithmic number of records), we further established the asymptotic distributions of the statistics as the size tends to infinity: three converge to Gaussian distributions, while the last converges to a beta distribution.

It would be interesting to extend this work to other permutation statistics, particularly those related to presortedness, such as the number of runs (i.e., maximal monotonic sequences) or the minimum number of elements that must be removed to obtain a sorted sequence (see [Man85]). We are especially interested in the number of alternating runs, as this statistic plays a central role in TimSort, which we study in the next section. Unfortunately, runs are difficult to handle directly within our model. A natural direction for future work is to design alternative biased models that better capture such statistics. As we have seen, generative processes provide powerful tools for exploring and defining such models[5] and they also provide a practical way to design efficient (linear-time) random samplers. As a simple application, we used our model to analyze the expected complexity of InsertionSort under the record-biased distribution. In the second part of this study, we also examine the impact of branch prediction (see Section 3) on two variants of simultaneous minimum and maximum search in arrays, assuming inputs are distributed according to our record-biased model.

Finally, although we did not present it here, as it lies beyond the scope of algorithm analysis, the typical shape of the diagram of a record-biased permutation (see Fig. 4) can be described via the permuton limit of random record-biased permutations of size n as n tends to infinity, in the linear regime where $\theta = \lambda n$. For these results, we refer the reader to [BNP25].

[5]Notably, while we were able to establish some results on expectations without generative processes, the proofs required more tedious case analysis.

2 Complexity Analysis of TimSort

In our University's Master program, we teach a lot of `Java`. This choice obviously aligns with industry demand, as `Java` remains one of the most widely used languages in practice. However, it also comes from a certain opportunism. One of our colleagues, Rémi Forax, is part of the team that develops the language, which means we are always up to date with its newest features. It was in fact he who first introduced us to TimSort, expressing his interest in better understanding its inner mechanics. This curiosity led us to investigate the algorithm, which made us realize that the implementations in standard libraries can deviate significantly from those presented in traditional textbooks. In the case of TimSort, its practical design reflects specific assumptions about the structure of real-world data: in particular, it is tailored to the presortedness (as discussed in the previous section) that is often present in real inputs.

$$\text{A}=(\underbrace{12,10,7,5,}_{\text{first run}}\ \underbrace{5,7,10,14,25,36,}_{\text{second run}}\ \underbrace{3,5,11,14,15,21,22,22,}_{\text{third run}}\ \underbrace{20,15,10,8,5,1}_{\text{fourth run}})$$

Figure 10: A sequence of values and its *run decomposition* by TimSort: for each run, the first two elements determine whether it is increasing or decreasing; the run then continues with the longest possible sequence of consecutive elements that preserve the monotonicity.

In a nutshell, TimSort processes the input array by greedily computing maximal monotonic runs on the fly (see Fig. 10). Each time a new run is identified, it is pushed onto a stack, followed by a possible sequence of merges, as in MergeSort, until the stack is stabilized. Historically, the merge conditions involved only the top three runs, and only two consecutive runs can be merged to favor cache locality. More precisely, if the runs are R_1, R_2, …, R_h, from top to bottom of the stack, with respective sizes r_1, r_2, …, r_h, merges occur as long as $r_1 \geq r_2$ or $r_1 + r_2 \geq r_3$, each involving either R_1 and R_2, or R_2 and R_3. Once the stack satisfies $r_1 < r_2$ and $r_1 + r_2 < r_3$, the next run is identified and added. When no runs remain to be computed, the ones in the stack are merged iteratively from top to bottom.

In an unpublished preprint [ANP15], we established that the running time of TimSort is $\mathcal{O}(n \log n)$, as initially stated without proof in the document accompanying its first implementation in `cpython`. Shortly after, De Gouw and co-authors [DGRdB+15] attempted a formal proof of TimSort using the verification tool `Keys` [BHS07]. They discovered that an important invariant stated in the algorithm's description ($r_i + r_{i+1} < r_{i+2}$ holds anywhere in the stack once stabilized) is actually incorrect. This had no impact on the correctness of the algorithm or on our proof, which does not rely on that invariant. By that time, however, TimSort has already been adopted in `Java` and the invariant was used to statically allocate the stack size. As a consequence, De Gouw et al. were able to construct an example array causing a stack overflow. They proposed two solutions: (i) resize the stack, using the fact that the invariant cannot fail twice in a row, or (ii) modify the algorithm to also trigger a merge when $r_2 + r_3 >= r_4$. The first solution was adopted in `Java`, while `Python` chose the second (although no change was necessary since `Python` uses a dynamic stack implementation).

Later, we showed that the claim that the invariant cannot be broken twice in a row is incorrect [AJNP18]. Consequently, patch (i) implemented in `Java` did not fully fix the algorithm, and we were able to trigger another stack overflow using a carefully constructed array. After we brought this to their attention, `Java` developers eventually adopted solution (ii), as in `Python`. Since this corrected version is now the standard implementation, we will refer to it simply as TimSort, and use the term *legacy* TimSort for the original algorithm.

In parallel with the work on the invariant $r_i + r_{i+1} < r_{i+2}$, we extended the analysis of TimSort's complexity by introducing parameters that quantify the pre-sortedness of the

Algorithm 1: TimSort, from `Python` up to 3.10

Input: a sequence A to sort
Result: the sequence A is sorted into a single run, which remains on the stack
Note: the function `merge_force_collapse` repeatedly pops the last two runs on the stack $\mathcal{S}$, merges them and pushes the resulting run back on the stack

```
1 runs ← a run decomposition of $A$
2 $\mathcal{S}$ ← an empty stack
3 while runs $\neq \emptyset$ do                    // main loop of TimSort
4     remove a run $R$ from runs and push $R$ onto $\mathcal{S}$
5     merge_collapse($\mathcal{S}$)
6 if height($\mathcal{S}$) $\neq 1$ then
7     merge_force_collapse($\mathcal{S}$)
```

input. A natural such parameter is the number of runs, denoted as ρ. It was conjectured for several years that TimSort's complexity could be refined to $O(n \log \rho + n)$, which we establish here as a byproduct of an even stronger result, using a finer parameter than ρ. The *entropy* of the run lengths (denoted by r_i, for $i \leq \rho$) is defined as $\mathcal{H} := H(r_1/n, \ldots, r_\rho/n)$, with H the binary Shannon entropy (see Eq. (6)). This measure generalizes the number of runs by capturing the disparity among run lengths. We prove that TimSort runs in $\mathcal{O}(n\mathcal{H} + n)$, which is optimal since a matching lower bound exists [BN13].

Buss and Knop [BK19] established a tight lower bound of $1.5n\mathcal{H} + \mathcal{O}(n)$ for TimSort's running time, conjecturing it was also an upper bound. We confirmed this, though we will not detail the proof here. These advances in the analysis inspired new first-order optimal algorithms, such as AdaptiveShiverSort by Jugé [Jug20] and PowerSort by Munro and Wild [MW18], the latter adopted in 2021 as `cpython`'s default sorting algorithm.

This work presented in the next section was done with Nicolas Auger, Vincent Jugé, and Cyril Nicaud [ANP15, AJNP18].

2.1 TimSort core algorithm

The idea behind TimSort was to design a merge sort capable of exploiting possible non-randomness in the input, without having to detect it beforehand and without degrading the performances on random-looking data. To achieve this, the first feature of TimSort is to use the natural decomposition of the input sequence into maximal runs. To capture longer subsequences, TimSort allows both nondecreasing and decreasing runs, unlike most merge sort algorithms. The merging strategy of TimSort (Algorithm 1) is simple yet effective: runs are considered in the order given by the run decomposition and successively pushed onto a stack. If certain conditions on the size of the topmost runs in the stack are not satisfied after a new run has been pushed, this triggers a sequence of merges involving pairs of runs at the top or right under. Once all initial runs have been pushed, the remaining ones are merged pairwise from the top of the stack to produce the final sorted sequence. These stack conditions and merging rules are handled by the `merge_collapse` subroutine (Algorithm 2), which forms the core mechanism of the original TimSort design.

Another strength of TimSort lies in its use of heuristics to improve efficiency. For example, it ensures that initial runs are not too short by applying insertion sort, and it employs a technique known as *galloping* to accelerate merges. These optimizations do not affect our

Algorithm 2: The `merge_collapse` procedure (from `Python` 3.4.4 up to 3.10)

```
Input: a stack of runs $\mathcal{S}$
Result: the invariant of Equations (4) and (5) is established
Note: the runs on the stack are $R_1, R_2 \ldots R_h$ (top to bottom), with lengths denoted by $r_i$
1 while height($\mathcal{S}$) $> 1$ do
2     if (height($\mathcal{S}$) $> 2$ and $r_3 \leqslant r_2 + r_1$) or (height($\mathcal{S}$) $> 3$ and $r_4 \leqslant r_3 + r_2$) then
3         if $r_3 < r_1$ then merge runs $R_2$ and $R_3$ on the stack
4         else merge runs $R_1$ and $R_2$ on the stack
5     else if $r_2 \leqslant r_1$ then merge runs $R_1$ and $R_2$ on the stack
6     else break
```

analysis and will not be discussed further here (see [GJK22, GJKY25] for an analysis of the galloping heuristic).

Algorithm 2 is a pseudo-code transcription of the `merge_collapse` procedure from `Python`. We keep the convention that the topmost run has index 1 and the length of run i is r_i. An example of execution of the main loop of TimSort (lines 3-5 of Algorithm 1) is presented in Fig. 11. As stated in its note [Pet], Tim Peter's idea was that:

> "The thrust of these rules when they trigger merging is to balance the run lengths as closely as possible, while keeping a low bound on the number of runs we have to remember."

To achieve this, the merging conditions in `merge_collapse` are designed to ensure that the following invariant[6] holds at the end of the procedure:

$$r_{i+2} > r_{i+1} + r_i, \tag{4}$$
$$r_{i+1} > r_i. \tag{5}$$

This implies that the runs lengths r_i on the stack grow at least as fast as Fibonacci numbers, and consequently, that the stack height remains logarithmic (see Lemma 25, Section 2.2).

Note that the bound on the stack height alone does not ensure the $\mathcal{O}(n \log n)$ running time of TimSort. Without the efficient merge strategy applied on the fly, one can easily construct an example with at most two runs on the stack that leads to $\Theta(n^2)$ complexity: for instance, if all runs have size two and are merged pairwise as soon as there are two runs on the stack. In contrast, the merges triggered by line 3 allow very large runs to be pushed and absorbed into the stack without being merged all the way down, effectively collapsing the stack below the new run instead. Meanwhile, the role the other merges is primarily to restore the invariant from Eqs. (4) and (5), ensuring an exponential growth of the run lengths within the stack. The cost of maintaining the stack's structural invariant is offset by the absorption of these large runs, naturally calling for an amortized complexity analysis.

2.2 TimSort runs in $\mathcal{O}(n \log n)$

Our main objective here is to provide an insightful proof of the complexity of TimSort, in order to illustrate how well designed its strategy for choosing the merge order is. To obtain precise bounds on the running time, we follow the standard approach of defining the *merge cost* of two runs of lengths r and r' as the size of the resulting run, $r + r'$. Henceforth, we

[6] Actually, in [Pet], the invariant is only stated for the 3 topmost runs of the stack.

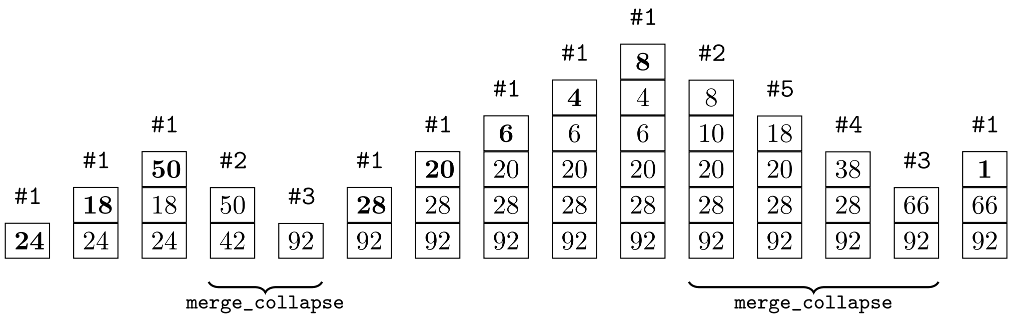


Figure 11: Successive states of the stack $\mathcal{S}$ (the values are the run lengths) during an execution of the main loop of TimSort (Algorithm 1), with run length $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. The label `#1` marks that a run has just been pushed onto the stack. Other labels correspond to the different merge cases of `merge_collapse`, as defined in Algorithm 3.

will consider the time spent merging two runs to be equal to their merge cost. Under this cost model, we are only one comparison above the lower bound in the comparison model, according to the following folklore result (it also matches the upper bound of the classical merging algorithm used in MergeSort).

Lemma 14. *For any algorithm that compares only pairs of elements, merging two ordered sequences of lengths n and m requires at least $n + m - 1$ comparisons in the worst case.*

We begin with our initial, least refined yet important result, followed by a detailed proof to offer a first glimpse into why TimSort performs so well in practice.

Theorem 15. TimSort *runs in $\mathcal{O}(n \log n)$.*

We start the proof with an alternative formulation of the TimSort algorithm, presented in Algorithm 3, which is easier to analyze and performs the same comparisons in the same order as Algorithm 1. The only difference is that Algorithm 2 is replaced by the `while` loop in lines 5 to 10 of Algorithm 3. By examining the different cases, it is easy to verify that merges involving the same runs occur in the same order in both algorithms.

The run decomposition in the first line can be computed using a simple greedy algorithm, in linear time. We can therefore turn to the main part of Algorithm 3, that is the `while` loop of lines 3-10. The proof proceeds using standard arguments from *amortized complexity*. Tokens are credited to the elements of the initial sequence at some point during the algorithm and are spent when comparisons are performed. An upper bound on the number of credited tokens then gives an upper bound on the running time of the algorithm. This technique allows us to pay in advance for costly successions of merges that can be more complicated to track than the credit times. Recall that the merge cost of two runs R_i and R_{i+1} is $r_i + r_{i+1}$, the sum of their lengths, which is an upper bound on the number of comparisons needed. Elements of the input array are easily identified by their starting position in the array, so we consider them as well-defined and distinct entities (even if they have the same value). The *height* of an element in the stack of runs is the number of runs that are below it in the stack: the elements belonging to the run R_i in the stack $\mathcal{S} = (R_1, \ldots, R_h)$ have height $h - i$.

Two $\diamondsuit$ tokens and one $\heartsuit$ token are credited to an element when its run is pushed onto the stack (when case `#1` is triggered) or when its height decreases because of a merge: in the latter case, all the elements of R_1 are credited when R_1 and R_2 are merged, and all the elements of R_1 and R_2 are credited when R_2 and R_3 are merged. Tokens are spent to pay for comparisons, depending on which case is triggered (the different cases are depicted in Fig. 12):

Algorithm 3: TimSort, translation of Algorithm 1 and Algorithm 2

Input: a sequence to A to sort
Result: the sequence A is sorted into a single run, which remains on the stack
Note: at any time, we refer to the height of the stack $\mathcal{S}$ as h

```
runs ← the run decomposition of $A$
$\mathcal{S}$ ← an empty stack
while runs $\neq \emptyset$ do                                          // main loop of TimSort
    remove a run $R$ from runs and push $R$ onto $\mathcal{S}$                      // #1
    while true do
        if $h \geqslant 3$ and $r_1 > r_3$ then merge the runs $R_2$ and $R_3$            // #2
        else if $h \geqslant 2$ and $r_1 \geqslant r_2$ then merge the runs $R_1$ and $R_2$     // #3
        else if $h \geqslant 3$ and $r_1 + r_2 \geqslant r_3$ then merge the runs $R_1$ and $R_2$  // #4
        else if $h \geqslant 4$ and $r_2 + r_3 \geqslant r_4$ then merge the runs $R_1$ and $R_2$  // #5
        else break
while $h \neq 1$ do merge the runs $R_1$ and $R_2$
```

- case #2: every element of R_1 and R_2 pays 1 $\diamondsuit$. This is enough to cover the cost of merging R_2 and R_3, because $r_1 > r_3$ in this case, and therefore $r_2 + r_1 \geqslant r_2 + r_3$.
- case #3: every element of R_1 pays 2 $\diamondsuit$. In this case $r_1 \geqslant r_2$, and the cost is $r_1 + r_2 \leqslant 2r_1$.
- cases #4 and #5: every element of R_1 pays 1 $\diamondsuit$ and every element of R_2 pays 1 $\heartsuit$. The cost $r_1 + r_2$ is exactly the number of tokens spent.

Lemma 16. *The balances of $\diamondsuit$ tokens and $\heartsuit$ tokens of each element remain non-negative throughout the main loop of* TimSort.

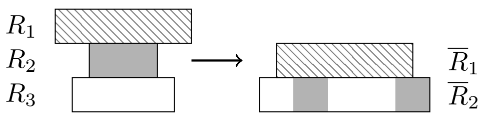
Proof. In all four cases #2 to #5, because the height of the elements of R_1 and possibly the height of those of R_2 decreases, the number of $\diamondsuit$ credited after the merge is at least the number of $\diamondsuit$ spent. The $\heartsuit$ tokens are spent only in cases #4 and #5: each element of R_2 pays one $\heartsuit$; after that, it belongs to the topmost run $\overline{R}_1$ of the new stack $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{h-1})$ obtained by merging R_1 and R_2. Since for all $i \geqslant 2$, $\overline{R}_i = R_{i+1}$, the conditions of case #4 and #5 imply respectively that $\overline{r}_1 \geqslant \overline{r}_2$ and $\overline{r}_1 + \overline{r}_2 \geqslant \overline{r}_3$. In both scenarios, the next operation on the stack $\overline{\mathcal{S}}$ is another merge.

This subsequent merge reduces the height of $\overline{R}_1$, which in turn lowers the height of the elements originally in R_2. These elements then regain one $\heartsuit$ without losing any, since the topmost run of the stack never pays in $\heartsuit$. Therefore, every time an element pays one $\heartsuit$, the very next modification is another merge that restores its $\heartsuit$. This concludes the proof by direct induction. □

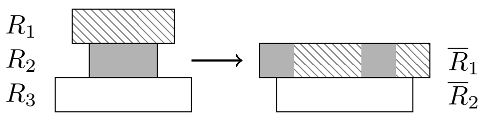
Since tokens are credited to an element upon entering the stack and whenever its height decreases, the running time of the `while` loop can be bounded by analyzing the stack height. To this end, we establish the invariants (4) and (5), ensuring an exponential growth of run lengths from top to bottom in the stack. Note that this is already established in [DGRdB+15]

Lemma 17. *At any point in the main loop of* TimSort, *$r_i + r_{i+1} < r_{i+2}$, for $i \in \{3, \ldots, h-2\}$.*

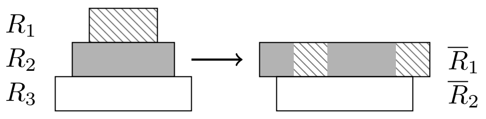
Proof. We proceed by induction, verifying that, if the invariant holds at some point, it continues to hold after a stack update in any of the five cases #1 to #5 of the algorithm. This

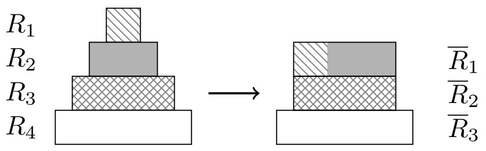


case #2: Since $r_1 > r_3$, runs R_2 and R_3 are merged into $\overline{R}_2$. Each element of R_1 and R_2 pays one $\diamondsuit$ token, covering the merge cost, as $r_2 + r_3 \leqslant r_2 + r_1$. These tokens are immediately regained due to the resulting decrease in height.



case #3: Since $r_1 \geqslant r_2$, runs R_1 and R_2 are merged into $\overline{R}_1$. Each element of R_1 pays two $\diamondsuit$, covering the merge cost, as $r_1 + r_2 \leqslant 2r_1$. These tokens are immediately regained due to the resulting decrease in height.





cases #4 & #5: Since $r_1 + r_2 > r_3$ (case #4) or $r_2 + r_3 > r_4$ (case #5), runs R_1 and R_2 are merged into $\overline{R}_1$. Each element of R_1 pays one $\diamondsuit$, and each element of R_2 pays one $\heartsuit$, covering the merge cost. The $\diamondsuit$ are immediately regained due to the decrease in height, while the $\heartsuit$ are restored during the following merge, as explained in the proof.

Figure 12: Schematic representations of the merge cases considered in the proof of Lemma 16.

follows from a straightforward case analysis. Let $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_{\overline{h}})$ be the new state of the stack after the update, and let $\overline{r}_i = |\overline{R}_i|$ denote the size of $\overline{R}_i$ for each i:

- If case #1 just occurred, a new run $\overline{R}_1$ was pushed onto the stack. This implies that none of the conditions of cases #2 to #5 held in $\mathcal{S}$, otherwise additional merges would have followed. In particular, if $h \geqslant 4$, we have $r_2 + r_3 < r_4$. Since $\overline{r}_i = r_{i-1}$ for all $i \geqslant 2$, and the invariant holds for $\mathcal{S}$, it also holds for $\overline{\mathcal{S}}$.
- If one of the cases #2 to #5 just occurred, $\overline{r}_i = r_{i+1}$ for all $i \geqslant 3$. Since the invariant holds for $\mathcal{S}$, it necessarily holds for $\overline{\mathcal{S}}$ as well.

$\square$

Corollary 18. *During the main loop of* TimSort*, whenever the inner* `while` *loop terminates, we have* $r_i \leqslant \sqrt{2}^{(i+1-j)} r_j$ *for all integers* $i \leqslant j \leqslant h$.

Proof. Since the inner loop has finished, none of the conditions of cases #2 to #5 hold in the stack $\mathcal{S}$. Combined with Lemma 17, this implies that $r_i + r_{i+1} < r_{i+2}$ for all $i \in \{1, \ldots, h-2\}$, and $r_1 < r_2$ if $h \geqslant 2$. In particular, $r_i < r_{i+1}$ for all $i \leqslant h-2$, and thus $2r_i \leqslant r_i + r_{i+1} \leqslant r_{i+2}$. It follows that $r_i \leqslant 2^{-k} r_{i+2k} \leqslant 2^{-k} r_{i+2k+1}$ for all $k \geqslant 0$ for which the runs exist. $\square$

This covers the main part of Theorem 15, as a consequence of Lemma 16, Corollary 18, and the fact that tokens are credited to an element at most its height plus one times.

Proposition 19. *The outer* `while` *loop of* TimSort *runs in* $\mathcal{O}(n \log n)$.

Proof. By taking $j = h$ in Corollary 18, we see that the stack height is in $\mathcal{O}(\log n)$ whenever the inner `while` loop finishes. Consequently, the height remains in $\mathcal{O}(\log n)$ at any point during the outer loop: each iteration adds one element to the stack before the inner `while` loop completes, and during this inner loop, the height only decreases.

Since elements are credited a constant number of tokens upon entering the stack and each time their height decreases, the total number of credited tokens is $\mathcal{O}(n \log n)$. By Lemma 16, the total number of comparisons performed during the outer `while` loop is therefore bounded from above by the number of credited tokens, which concludes the proof. □

Only the final sequence of merges remains, and due to the exponential decrease in size given by Corollary 18, it follows directly that this sequence completes in linear time. Note that the analysis of line 11 could also be omitted by adding a fictitious run of length $n+1$, consisting of elements greater than any in the original array, and appending it to the end of the run decomposition.

Lemma 20. *Line 11 of Algorithm 3 runs in linear time.*

Proof. Let $\mathcal{S} = (R_1, \ldots, R_h)$ be the stack just before Line 11. Each element in R_i will be involved in at most $h+1-i$ merges during the process, except those in R_1, which participate in only $h-1$ merges. Thus, the total number of comparisons is bounded by $\sum_{i=1}^{h}(h+1-i)r_i$. Since $r_h \leqslant n$, applying Corollary 18 with $j = h$ yields

$$\sum_{i=1}^{h}(h+1-i)r_i \leqslant n\left(\sum_{i=1}^{h}(h+1-i)\sqrt{2}^{i+1-h}\right) = 2n\sum_{\ell=1}^{h}\ell\sqrt{2}^{-\ell},$$

which is in $\mathcal{O}(n)$ since the sum converges as h tends to infinity, completing the proof. □

This concludes the proof of Theorem 15: computing the runs takes $\mathcal{O}(n)$ time, the main loop runs in $\mathcal{O}(n \log n)$ by Proposition 19, and the final merges complete in $\mathcal{O}(n)$ by Lemma 20.

2.3 TimSort runs in $\mathcal{O}(n\mathcal{H} + n)$

In this section, we refine the analysis by introducing some measures of presortedness [Man85], as defined in Section 1.1.4, to capture properties of the input that more accurately describe the performance of the algorithm. In our setting, the number of runs ρ is a natural measure: as we shall see below, TimSort runs essentially in $\mathcal{O}(n \log \rho)$, highlighting its efficiency on inputs with relatively few runs. Our main result is more precise, showing that TimSort finds inputs consisting of ρ runs each of length n/ρ harder to process than inputs made up of one large run of length $n-2(\rho-1)$ together with $\rho-1$ runs of length 2. This is to be expected for an efficient algorithm: in the latter case, each element of the long run can be placed correctly using binary search with very few comparisons.

Shannon entropy is a classical and natural way to account for this without introducing too many parameters to describe the run structure. Recall that if $p_1, \ldots p_\rho$ are non-negative real numbers that sum to 1, their binary Shannon entropy is the quantity defined by

$$H(p_1, \ldots, p_\rho) = -\sum_{i=1}^{\rho} p_i \log_2(p_i). \qquad (6)$$

For any integer vector $\vec{r} = (r_1, \ldots, r_\rho)$, with each $r_i \geqslant 2$, let $\mathcal{C}(\vec{r})$ denote the class of arrays of length n, whose run decomposition consists of ρ monotonic runs of lengths $r_1, \ldots, r_\rho$. The *entropy* of the run lengths is defined as $\mathcal{H}(\vec{r}) := H(r_1/n, \ldots, r_\rho/n)$. For instance,

- if $\vec{r} = (\frac{n}{\rho}, \ldots, \frac{n}{\rho})$ then $\mathcal{H}(\vec{r}) = \log_2 \rho$;
- if $\vec{r} = (n-2(\rho-1), 2, \ldots, 2)$ then $\mathcal{H}(\vec{r}) \approx \frac{2(\rho-1)\log_2 n}{n}$.

Thus, to exploit presortedness, it is natural to consider sorting algorithms that perform well on inputs with low entropy, such as TimSort, as evidenced in Theorem 21 below.

Note that, when it is clear from context (as below), we omit the parameter $\vec{r}$ and simply write $\mathcal{H} := \mathcal{H}(\vec{r})$ for the run entropy of arrays in the class $\mathcal{C} := \mathcal{C}(\vec{r})$.

Theorem 21. *For inputs of length n,* TimSort *runs in time $\mathcal{O}(n + n\mathcal{H})$.*

Since Shannon entropy is maximal for the uniform distribution ($p_1 = p_2 = \ldots = p_\rho$), we immediately obtain the following, less precise result.

Corollary 22. *For inputs of length n with ρ runs* TimSort *runs in time $\mathcal{O}(n + n\log\rho)$.*

Proof. The function $f : x \mapsto -x\ln(x)$ is concave on the positive real numbers $\mathbb{R}_{>0}$, since its second derivative is $f''(x) = -1/x$. Hence, for any positive $p_1, \ldots, p_\rho$ summing to one, we have $H(p_1, \ldots, p_\rho) = \sum_{i=1}^{\rho} f(p_i)/\ln(2) \leqslant \rho f(1/\rho)/\ln(2) = \log_2(\rho)$. In particular, this implies that $\mathcal{H} \leqslant \log_2(\rho)$, so TimSort runs in $\mathcal{O}(n + n\log\rho)$ time. Note that, since $\rho \leqslant n$, we also recover the result of Theorem 15: $\mathcal{O}(n + n\log\rho) \subseteq \mathcal{O}(n + n\log n) = \mathcal{O}(n\log n)$. □

Before proving Theorem 21, we first show that it is optimal up to a multiplicative constant (a related, slightly less precise proof for ascending runs appears in [BN13, Th. 2]).

Proposition 23 (lower bound)**.** *For every algorithm comparing only pairs of elements, there exists an array in the class $\mathcal{C}$ whose sorting requires at least $n\mathcal{H} - 3n$ element comparisons.*

Proof. In the comparison-based model, for any given run length vector $\vec{r} = (r_1, \ldots, r_\rho)$, at least $\log_2(|\mathcal{C}(\vec{r})|)$ element comparisons are required for sorting all arrays in $\mathcal{C}(\vec{r})$, by the classical counting argument for lower bounds. We prove below that $\log_2(|\mathcal{C}(\vec{r})|) \geqslant n\mathcal{H}(\vec{r}) - 3n$.

In this model, only the relative order of elements matters; therefore, it suffices to consider permutations of $\{1, \ldots, n\}$. Let $\mathcal{P}$ be the set of partitions π of $\{1, \ldots, n\}$ such that $|\pi_i| = r_i$ for all $i \leqslant \rho$. We say that π is *nice* if $\max\pi_i > \min\pi_{i+1}$ for all $i \leqslant \rho - 1$: this ensures that listing the elements of $\pi_1, \ldots, \pi_\rho$ in order corresponds to an input whose run length vector is exactly $\vec{r}$. On the contrary, if π is not nice, the run decomposition does not match with $\vec{r}$. We denote by $\mathcal{N} \subseteq \mathcal{P}$ the subset of nice partitions.

Let us transform any partition $\pi \in \mathcal{P}$ into a nice partition as follows. Since the sizes $|\pi_i|$ correspond to run lengths, each π_i except the last contains at least two elements, so for all $i \leqslant \rho - 1$ we have $\min\pi_i < \max\pi_i$. Then, for all $i \leqslant \rho - 1$, if $\max\pi_i < \min\pi_{i+1}$, we swap the elements $\max\pi_i$ and $\min\pi_{i+1}$ between their respective parts,[7] and denote the resulting partition by π^*, which is guaranteed to be nice. Since each partition π^* can result from at most $2^{\rho-1}$ different partitions $\pi \in \mathcal{P}$, it follows that $2^{\rho-1}|\mathcal{N}| \geqslant |\mathcal{P}|$. Now, identify each partition π^* with an array in $\mathcal{C}(\vec{r})$, which starts with the elements of π_1^* (in increasing order), followed by those of π_2^*, and so on, up to π_ρ^*. Since these partitions are nice, this mapping from $\mathcal{N}$ to $\mathcal{C}(\vec{r})$ is injective, and we conclude that $|\mathcal{C}(\vec{r})| \geqslant |\mathcal{N}|$.

Finally, using variants of the Stirling formula which give $(k/e)^k \leqslant k! \leqslant e\sqrt{k}(k/e)^k$ for all $k \geqslant 1$, and since $|\mathcal{P}| = \binom{n}{r_1,\ldots,r_\rho} = \frac{n!}{r_1!\cdots r_\rho!}$, we obtain

$$\log_2(|\mathcal{C}(\vec{r})|) \geqslant n\mathcal{H}(\vec{r}) + (1 - \rho - \rho\log_2(e)) - 1/2\sum_{i=1}^{\rho}\log_2(r_i).$$

[7]For instance, if $\pi = (\{1, 2, 3, 4\}, \{5, 6\})$, then $\pi^* = (\{1, 2, 3, 5\}, \{4, 6\})$.

By concavity of the function $x \mapsto \log_2(x)$, we have $\sum_{i=1}^{\rho} \log_2(r_i) \leqslant \rho \log_2(n/\rho)$. One checks readily that the function $x \mapsto x \log_2(n/x)$ reaches its maximum at $x = n/e$. Since $n \geqslant \rho$, it follows that $\log_2(|\mathcal{C}(\vec{r})|) \geqslant n\mathcal{H}(\vec{r}) - (1 + \log_2(e) + \log_2(e)/e)n \geqslant n\mathcal{H}(\vec{r}) - 3n$. □

Elements of the proof of Theorem 21 It follows the same ideas as in Section 2.2, but we need to treat separately the sequence of cases #2 that occur immediately after the insertion of each run. For any input, we consider the sequence of cases #1 to #5 triggered during the execution of the main loop of TimSort. This sequence fully encodes the execution of the algorithm. Splitting this sequence at each occurrence of #1 yields segments corresponding to iterations of the main loop. Each such segment is further divided into two parts at the first occurrence (if any) of a #3, #4, or #5. The first part, called a *starting sequence*, consists of a #1 followed by a maximal number of #2's. The second part, called an *ending sequence*, starts with #3, #4, or #5 (or is empty), and contains no occurrence of #1 (see Fig. 13).

$$\underbrace{\texttt{\#1 \#2 \#2 \#2}}_{\text{starting seq.}}\quad \underbrace{\texttt{\#3 \#2 \#5 \#2 \#4 \#2}}_{\text{ending seq.}}\quad \underbrace{\texttt{\#1 \#2 \#2 \#2 \#2 \#2}}_{\text{starting seq.}}\quad \underbrace{\texttt{\#5 \#2 \#3 \#3 \#4 \#2}}_{\text{ending seq.}}$$

Figure 13: A trace of cases during execution, decomposed into starting and ending sequences.

We use two new ingredients to prove Theorem 21. The first one is that the total number of comparisons required to merge the starting sequences is at most linear.

Lemma 24. *The total cost of all merges performed during the starting sequences is* $\mathcal{O}(n)$.

Proof. For a stack $\mathcal{S} = (R_1, \ldots, R_h)$, we prove that any starting sequence initiated by pushing a run R of size r onto $\mathcal{S}$ uses at most γr comparisons in total, with $\gamma = 2\sum_{\ell \geqslant 1} \ell \sqrt{2}^{-\ell}$. After the push, the stack becomes $\overline{\mathcal{S}} = (R, R_1, \ldots, R_h)$. If the starting sequence consists of $k \geqslant 1$ steps (i.e., $k-1$ applications of case #2), then this sequence merges the runs $R_1, R_2, \ldots, R_k$. Since no merge occurs if $k = 1$, we assume $k \geqslant 2$. The final application of case #2 ensures that $r > r_k$, and by Corollary 18 applied to the stack $\mathcal{S} = (R_1, \ldots, R_h)$, we have $r \geqslant r_k \geqslant \sqrt{2}^{k-1-i} r_i$ for all $i = 1, \ldots, k$. As in the proof of Lemma 20, it follows that:

$$C \leqslant r \sum_{i=1}^{k} (k+1-i)\sqrt{2}^{i+1-k} = 2\sum_{\ell=1}^{k} \ell \sqrt{2}^{-\ell} < \gamma r.$$

This concludes the proof, since each run initiates exactly one starting sequence, and the total sum of their lengths is n. □

Now, we handle the merges that occur during ending sequences. We adapt the amortized analysis technique from Section 2.2, restricting it to merges performed within these sequences:

- 2 ♢ tokens and 1 ♡ token are credited to an element when it first enters the stack, and when its height decreases *because of a merge belonging to an ending sequence* (no token is credited during the case #2 merges of starting sequences);
- tokens are used to pay for comparisons *during ending sequences only.*

Since token credits and debits occur only during ending sequences, the balances of ♢ tokens and ♡ tokens remain non-negative throughout the main loop of TimSort, by a direct adaptation of Lemma 16. It remains to establish a bound h on the height of a newly inserted element *after* its starting sequence, in order to bound the number of tokens it is credited: at most $2h + 2$ ♢ tokens and $h + 1$ ♡ tokens. This is the purpose of the following lemma.

Lemma 25. *The height of the stack when the starting sequence of a run of length r is over satisfies the inequality $h \leqslant 4 + 2\log_2(n/r)$.*

Proof. Let $\mathcal{S}$ be the stack just before pushing a run R of length r, and let $\overline{\mathcal{S}} = (\overline{R}_1, \ldots, \overline{R}_h)$ be the stack just after the starting sequence of R (i.e., the starting sequence initiated when R is pushed onto $\mathcal{S}$) is over. Since none of the runs $\overline{R}_3, \ldots, \overline{R}_h$ has been merged during the starting sequence of R, applying Corollary 18 to the stack $\mathcal{S}$ proves that $\overline{r}_3 \leqslant 2^{2-h/2}\overline{r}_h \leqslant 2^{2-h/2}n$. The run R has not yet been merged either, which means that $r = \overline{r}_1$. Moreover, at the end of this starting sequence, the conditions of case #2 no longer hold, which means that $\overline{r}_1 \leqslant \overline{r}_3$. It follows that $r = \overline{r}_1 \leqslant \overline{r}_3 \leqslant 2^{2-h/2}n$, which entails the desired inequality. ☐

Collecting the above results suffices to prove Theorem 21: the starting sequences of the main loop have a total merge cost of $\mathcal{O}(n)$ by Lemma 24, and the ending sequences have a total merge cost of $\mathcal{O}(\sum_{i=1}^{\rho}(4+2\log_2(n/r_i))r_i) = \mathcal{O}(n + n\mathcal{H})$ by Lemma 25. The other parts of the algorithm are treated exactly as in Theorem 15 and contribute to $\mathcal{O}(n)$ time.

2.4 Further developments

2.4.1 Refined analysis and precise worst-case running time

The analysis performed in Section 2.2 proves that TimSort runs in time $\mathcal{O}(n+n\mathcal{H})$. Looking more closely at the constants hidden in the $\mathcal{O}$ notation, one can in fact prove that the cost of merges performed during an execution of TimSort is never greater than $6n\mathcal{H} + \mathcal{O}(n)$. However, the lower bound provided by Proposition 23 only proves that the cost of these merges must be at least $n\mathcal{H} + \mathcal{O}(n)$. In addition, as mentioned in the introduction, there do exist sorting algorithms [MW18, Jug20] whose merge cost is exactly $n\mathcal{H} + \mathcal{O}(n)$, which is therefore tight for the first order asymptotic.

Hence, TimSort is optimal only up to a multiplicative constant. In an extended, still unpublished version of [AJNP18], we refined this estimate by determining the smallest real constant κ such that the merge cost of TimSort is at most $\kappa n\mathcal{H} + \mathcal{O}(n)$, thereby proving a conjecture posed in [BK19]. The result is as follows.

Theorem 26. *The merge cost of* TimSort *on arrays of length n is at most $\kappa n\mathcal{H} + \mathcal{O}(n)$, where $\kappa = 3/2$. Furthermore, $\kappa = 3/2$ is the least real constant with this property.*

Theorem 26 consists of two parts: an asymptotic upper bound of $\frac{3}{2}n\mathcal{H}$, and a matching lower bound of the same order, the latter established in [BK19]. Our full proof appears in the arXiv version of [AJNP18], but we do not include it here, as it is somewhat tedious.

2.4.2 About the legacy version of TimSort

Algorithm 2 (and thus Algorithm 3) differs slightly from the original TimSort. Prior to `Python` release 3.4.4, the second part of the condition at line 2 of `merge_collapse` (and therefore merge case #5 of Algorithm 3) was missing. This earlier variant, shown in Algorithm 4, is what we refer to as *legacy* TimSort. Although it correctly sorted arrays, the invariant in Equation (4) could fail. Figure 14 illustrates the impact of the missing condition on the same input as in Fig. 11. Legacy TimSort was also used in `Java` to sort arrays of objects until September 2018 when `Java` 11 was released, at which point it was changed to the new TimSort, following the bug report we made while publishing our results [AJNP18].

Algorithm 4: legacy TIMSORT, translation of the `Java` code (up to `Java` 10)

```
Input: a sequence to $A$ to sort
Result: the sequence $A$ is sorted into a single run, which remains on the stack
Note: at any time, we refer to the height of the stack $\mathcal{S}$ as $h$

1  runs ← the run decomposition of $S$
2  $\mathcal{S}$ ← an empty stack
3  while runs $\neq \emptyset$ do                                          // main loop of TIMSORT
4      remove a run $R$ from runs and push $R$ onto $\mathcal{S}$                   // #1
5      while true do
6          if $h \geqslant 3$ and $r_1 > r_3$ then merge the runs $R_2$ and $R_3$          // #2
7          else if $h \geqslant 2$ and $r_1 \geqslant r_2$ then merge the runs $R_1$ and $R_2$    // #3
8          else if $h \geqslant 3$ and $r_1 + r_2 \geqslant r_3$ then merge the runs $R_1$ and $R_2$  // #4
9          else break
10 while $h \neq 1$ do merge the runs $R_1$ and $R_2$
```

After the first bug in `Java`'s implementation was discovered by De Gouw *et al.* [DGRdB+15], the static maximum size of the run stack was adjusted based on the assumption that the invariant could not be broken for two consecutive runs on the stack. This also turned out to be incorrect,[8] as illustrated in Fig. 15. Consequently, up to `Java` 10, it was still possible to cause the `Java` implementation to fail: it uses a stack of runs of size at most 49, while we constructed an example requiring a stack of size 50 (see http://igm.univ-mlv.fr/~pivoteau/Timsort/Test.java), causing an error at runtime in `Java`'s sorting method.

Even if the bug we identified in `Java`'s TIMSORT is very unlikely to occur in practice, it still needed to be fixed.[9] Following our recommendation, this correction was finally made in the release of `Java` 11. However, since legacy TIMSORT is still present in earlier versions of `Java` that are still in use (and possibly widespread), two questions remain: Does the complexity analysis still hold without the missing condition? And, can we compute an actual bound on the stack size? It turns out that the missing invariant was a key ingredient in obtaining simple and elegant proofs. Deriving similar results for the legacy TIMSORT requires more involved (and less insightful) arguments. We only summarize the results here (full proofs can be found

[8]This results from a small error in the proof of their Lemma 1. The constraint $C_1 > C_2$ is unfounded. Indeed, in our example, we have $C_1 = 25$ and $C_2 = 31$.

[9]See the related discussion for Python at: https://bugs.python.org/issue23515.

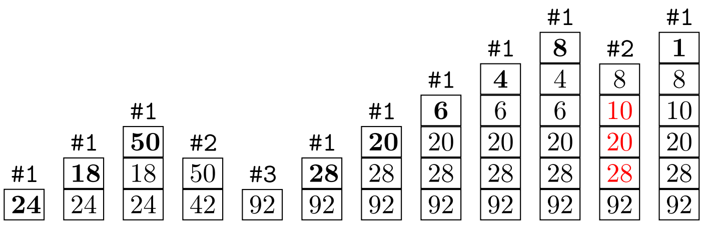


Figure 14: Execution of the main loop of legacy TIMSORT (Algorithm 4), with run lengths $(24, 18, 50, 28, 20, 6, 4, 8, 1)$. When the second to last run (of length 8) is pushed onto the stack, the while loop of line 5 stops after only one merge, breaking the invariant (in red), in contrast with the behavior of the `Python` version shown in Fig. 11.

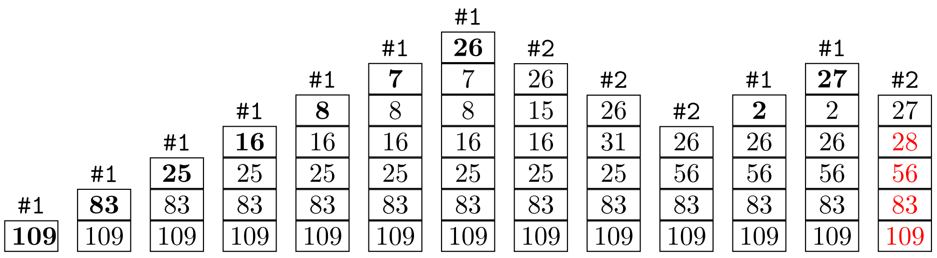


Figure 15: Execution of the main loop of legacy TimSort, with run lengths $(109, 83, 25, 16, 8, 7, 26, 2, 27)$. When it stops, the invariant is broken at two consecutive runs (in red).

in the arXiv version of [AJNP18]): the complexity of legacy TimSort on inputs of size n with ρ runs is indeed $\mathcal{O}(n + n \log \rho)$. Moreover, on an input of size n, the algorithm creates a stack of at most $h_{\max} \leqslant 7 + \log_\delta(n)$ runs, where $\delta = \big(5/(2+\sqrt{7})\big)^{1/5}$.

Unfortunately, for integers smaller than 2^{31}, this only guarantees that the stack size never exceeds 347. However, in the comments of `Java`'s implementation of TimSort,[10] it is mentioned that keeping the stack short is important, for practical reasons, and that `Python`'s bound of 85 is considered "too expensive." Thus, we went so far as to compute the optimal bound. It turns out that this bound is $h_{\max} \leqslant 3 + \log_\Delta(n)$, where $\Delta = (1+\sqrt{7})^{1/5}$, and it never exceeds 86 for such integers. This bound might be slightly improved, but not to the extent of competing with the one obtained if the invariant of Eq. (4) were valid, so we stopped our efforts at this point.

2.5 Conclusion on TimSort

When we discovered that `Java` had adopted TimSort in place of the highly regarded QuickSort, we saw an opportunity to validate a novel algorithm with a compelling average-case analysis that could explain this choice. While such an analysis should be most reflective of real-world behavior, it remains only a complement to a proper worst-case complexity analysis, which serves as a guarantee on execution time. At the time, this was lacking for TimSort, so we first addressed this gap by providing a short, simple, and self-contained proof of its worst-case running time. In doing so, we came to appreciate the true nature of the algorithm and developed a clearer view of how it operates in practice, particularly the central role played by the merge strategy. We came to realize that the excellent performance, which likely motivated the adoption of TimSort in widely used languages such as `Python` and `Java`, can be explained through a worst-case analysis, albeit from a different perspective on the input data. While TimSort is not worse than traditional algorithms on uniformly random inputs, it is particularly well suited to partially sorted inputs. Even stripped of its fine-tuned heuristics, the dynamics of its merge process, driven by a few local rules, yield globally efficient behavior. This is captured in the bound $\mathcal{O}(n + n\mathcal{H})$, where $\mathcal{H}$ denotes the entropy of the run lengths.

As we have seen, the story of TimSort is rich in developments and highlights the importance of careful algorithm analysis in avoiding errors and misconceptions. This work also stands as a successful example of fruitful collaboration between engineers and academics and this is the kind of interaction we hope to continue fostering in the future.

[10]Comment at line 168: http://igm.univ-mlv.fr/~pivoteau/Timsort/TimSort.java.

Summary of Our Results on Data Oriented Analysis

At first, when we learned that `Java`'s implementation of QuickSort had been replaced by a variant of MergeSort, we assumed that this new algorithm, TimSort, must be particularly efficient in practice. Naturally, we wanted to study its average-case complexity to confirm this from a theoretical perspective. However, we soon realized that its worst-case complexity had not yet been formally established. This led us to first focus on proving that TimSort runs in $\mathcal{O}(n \log n)$, as was announced by its creator.

In parallel, we began investigating the notion of presortedness, aiming to characterize the types of inputs that could explain why an algorithm like TimSort would be favored over more obvious alternatives, i.e., algorithms already known to perform well in practice. Given our interest in experimenting to test our ideas, it quickly became clear that having a suitable model (and ideally random generators) would be valuable for putting TimSort to the test. This naturally led us to the question of non-uniform distributions for random permutations.

While the worst-case analysis clarified aspects of TimSort's behavior, it was not sufficient to explain why it had been brought to the forefront. We refined our analysis and showed that TimSort runs in $\mathcal{O}(n + n\mathcal{H})$, where $\mathcal{H}$ is the entropy of the run lengths, a natural parameter that accounts for its strong performance on partially sorted inputs.

To complete the picture, a natural next step would be to develop a probabilistic model for biased permutations that allows us to derive results on the expected number of runs, thereby enabling a meaningful average-case analysis.

Part II

Enhancing the Model with Computer Architecture Features

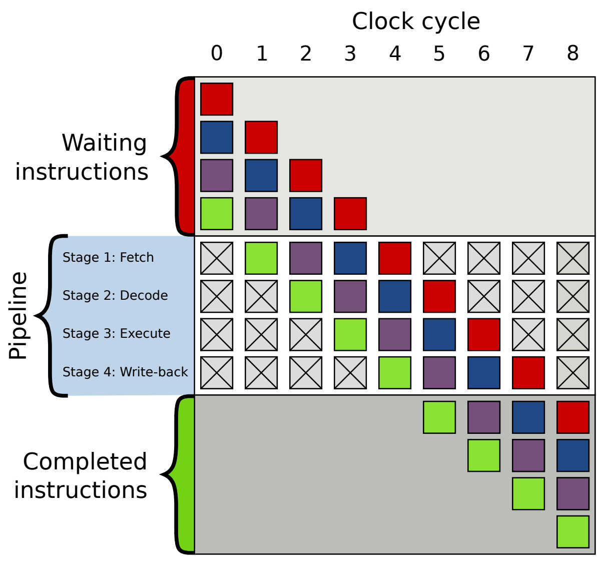
In pursuit of a more realistic model for algorithm analysis, the second part of our work turns to modern architectural features. We try to design models that capture phenomena observed in practice, aiming to provide theoretical results that align with empirical behavior. As computing systems grow increasingly complex, continuous efforts to improve performance have led to the introduction of various hardware mechanisms for accelerating data access, predicting control flow, and enabling parallel execution. Among these, caching stands out as one of the most impactful innovations, giving rise to an entire field of research. This led to the development of two major algorithmic models: external memory algorithms [AV88] and cache-oblivious algorithms [Dem02, FLPR12]. As a result, cache-related behavior is now well understood from an algorithmic perspective, although detailed, fine-grained analysis remains technically challenging. In our work, we instead focus on another architectural feature: branch prediction, which plays a crucial role in improving instruction-level parallelism (see next section). This topic began attracting attention in the 2000s, with both experimental and theoretical approaches, primarily targeting worst-case analyses. To deepen our understanding of its impact, we shift focus to the average case. We concentrate mainly on relatively simple models as a starting point, aiming to later explore more complex ones that are more representative of real processors. Proposing an adequate model is a challenge in itself, especially since the exact designs of processors are proprietary and sparsely documented. Whenever possible, we compare our theoretical results with experimental data; some results align well, while others suggest the presence of more refined mechanisms at play. We explore two main directions in this context: exploiting branch prediction to improve algorithm performance when execution paths are simple, and analyzing the behavior of branch predictors when algorithms present more complex branching patterns. Our results highlight trade-offs between architectural events and operations traditionally assumed to have unit-time cost, in the spirit of the work on cache misses in [LL99], and they pave the way for studying more sophisticated and accurate models of branch predictors.

3 Branch Prediction

Pipelining is a fundamental technique in modern processors that improves performance by allowing multiple instructions to execute in parallel rather than strictly sequentially. Typically, the execution of an instruction is divided into distinct stages, so that several instructions can be processed simultaneously, much like an assembly line in a factory. Fig. 16 illustrates how this works on a four-stage pipeline. Today, pipelining is ubiquitous, even in low-cost processors priced under a dollar (see [HP17] for a detailed presentation).

The sequential execution model assumes independence between instructions, since each completes before the next begins; however, this assumption no longer holds in a pipelined processor. Specific conditions, known as hazards, may prevent the next instruction in the pipeline from executing during its designated clock cycle. Hazards introduce delays that undermine the performance benefits of pipelining and may stall the pipeline, thus reducing the theoretical speedup:

- *Structural hazards* occur when resource conflicts arise, preventing the hardware from supporting all possible combinations of instructions during simultaneous executions.
- *Control hazards* arise from the pipelining of conditional jumps (arising from loops or `if-then-else` structures, for instance) and other instructions that modify the order in which instructions are processed, by updating the Program Counter (PC).
- *Data hazards* occur when an instruction depends on the result of a previous instruction, and this dependency is exposed by the overlap of instructions in the pipeline.



Instructions are represented as colored boxes, and clock cycles correspond to the processor's discrete time units. At cycle 0, four instructions are waiting to be executed. Over the following four cycles, the green instruction progresses through the CPU's four pipeline stages: it is fetched from memory, decoded, executed, and finally its result is written back to either the register file or memory. At cycle 2, the purple instruction enters the pipeline's first stage (fetch), as the green instruction has already advanced to the second stage (decode). The third and fourth instructions follow the same pattern. Note that this parallelism is only possible if the instructions are independent (i.e., there are no data or control dependencies between them). As a result, only eight clock cycles are needed to complete all four instructions, compared to sixteen cycles in a fully sequential four-stage CPU.

Figure 16: Example of the evolution of a four-stage pipeline on independent instructions.[1]

We focus here on control hazards from branching instructions. Two-way branching is typically implemented using a conditional jump instruction, which can either be *taken*, updating the Program Counter with the target address specified by the jump instruction and redirecting the execution path to a different location in memory, or *not taken*, allowing execution to continue sequentially. Note that when a conditional instruction is compiled, it results in a jump that may correspond to either a taken or a not-taken branch.[2] For consistency, we define a successful condition (when a test evaluates to true) as always leading to a taken branch.

During the execution of a program, the outcome of any conditional jump remains unknown until the evaluation of its condition reaches the actual execution stage, which may cause stalls in the pipeline. To improve efficiency, modern computer architectures incorporate a mechanism to predict the branch to take: a *branch predictor* is a digital circuit that anticipates the outcome of a branch (due to an `if-then-else` statement, for instance) before it is resolved. Without branch prediction, the processor would be forced to wait until the conditional jump instruction reaches its last stage before the next instruction can enter the first stage in the pipeline. The branch predictor aims to reduce this delay by predicting whether the conditional jump is likely to be taken or not. The instruction corresponding to the predicted branch is then fetched and speculatively executed. If the prediction is later found to be incorrect (that is, a *misprediction* has occurred), the speculatively or partially executed instructions must be cancelled: the instructions after the jump are discarded, the pipeline is flushed, and execution resumes along the correct path, incurring a small delay. A misprediction costs little more than the stall that would have occurred without branch prediction, whereas a correct prediction avoids the stall entirely, which is where the real bonus lies.

The effectiveness of a branch prediction scheme depends on both its accuracy and the

[1]Illustration from Wikipedia, by en:User:Cburnett - This W3C-unspecified vector image was created with Inkscape, CC BY-SA 3.0.

[2]Most assembly languages provide both `jump-if-true` and `jump-if-false` instructions.

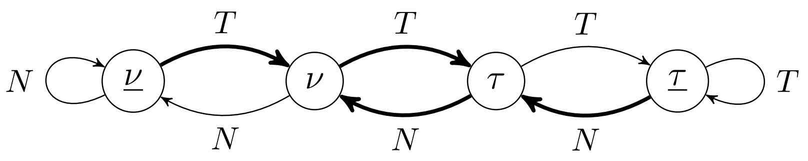


Figure 17: The 2-bit saturating predictor consists of four states: $\underline{\nu}$ and ν predict that the branch will not be taken, while τ and $\underline{\tau}$ predict that it will. The predictor updates at each condition evaluation, transitioning via T when the branch is taken (i.e., the condition is true) and via N when it is not. Bold edges indicate mispredictions.

frequency of conditional jumps. *Static branch prediction* is the simplest technique, as it does not depend on the code execution history, with the drawback that it cannot adapt to program behavior. In contrast, *dynamic branch prediction* takes advantage of runtime information (specifically, branch execution history) to determine whether branches were taken or not, allowing it to make more informed predictions about future branches.

A vast body of research is dedicated to dynamic branch prediction schemes. At the highest level, branch predictors are classified into two categories: global and local. A *global branch predictor* does not maintain separate history records for individual conditional jumps. Instead, it relies on a shared history of all jumps, allowing it to capture their correlations and improve prediction accuracy. In contrast, a *local branch predictor* maintains an independent history buffer for each conditional jump, enabling predictions based solely on the behavior of that specific branch. Modern processors typically employ a combination of local and global branch prediction techniques, often incorporating even more sophisticated designs. For a deeper exploration of this topic, see [Mit18], and for a comprehensive discussion of modern computer architecture, refer to [HP17].

We primarily focus on local branch predictors implemented with *saturating counters*. A *1-bit saturating counter* (essentially a flip-flop) records the most recent branch outcome. Although this is the simplest form of dynamic branch prediction, it offers limited accuracy. A *2-bit saturating counter* (see Fig. 17), by contrast, operates as a state machine with four possible states: Strongly not taken ($\underline{\nu}$), Weakly not taken (ν), Weakly taken (τ), and Strongly taken ($\underline{\tau}$). When the 2-bit saturating branch predictor is in the Strongly not taken or Weakly not taken state, it predicts that the branch will not be taken and execution will proceed sequentially. Conversely, when the predictor is in the Strongly taken or Weakly taken state, it predicts that the branch will be taken, meaning execution will jump to the target address.

Each time a branch is evaluated, the corresponding state machine updates its state. If the branch is not taken, the state shifts toward Strongly not taken; if taken, it moves toward Strongly taken. A misprediction (corresponding to a bold edge in Fig. 17) occurs when:

- a branch is not taken, while the predictor is in either of the Taken states (τ or $\underline{\tau}$);
- a branch is taken, while the predictor is in either of the Not Taken states (ν or $\underline{\nu}$).

This mechanism gives the 2-bit saturating counter an advantage over the simpler 1-bit scheme: a branch must deviate twice from its usual behavior (i.e., a Strongly state) before the prediction changes, reducing the likelihood of mispredictions. Note that this saturating counter scheme can be further improved by keeping more information (*k-bit* predictors using 2^k states).

Branch prediction and it influence over algorithms efficiency in the literature
Since the 2000s, several articles began to address the influence of branch predictors, and es-

pecially the cost of mispredictions, in comparison based algorithms. For instance, Biggar and his coauthors [BNWG08] investigated the behavior of branches for many sorting algorithms, in an extensive experimental study. Brodal, Fagerberg and Moruz reviewed the trade-offs between comparisons and mispredictions for several sorting algorithms [BM05] and studied how the number of inversions in the data affects statistics such as the number of mispredictions [BFM08]. Moreover, these works introduced the first theoretical analysis of static branch predictors.

Also interested by the influence of mispredictions on the running time of sorting algorithms, Sanders and Winkel considered the possibility to dissociate comparisons from branches in their SampleSort, which allows to avoid most of the misprediction cost [SW04]. Elmasry, Katajainen and Stenmark then proposed a version of MergeSort that is not affected by mispredictions [EKS12], by taking advantage of some processor-specific instructions.[3] The influence of mispredictions was also studied for Quicksort: Kaligosi and Sanders gave an in-depth analysis of simple dynamic branch predictors to explain how mispredictions affect this classical algorithm [KS06]; however, Martínez, Nebel and Wild pointed out that this is not enough to explain the "better than expected" performances of the dual-pivot version of QuickSort [MNW15] implemented in Java's standard library, while Edelkamp and Weiß gave an alternative called BlockQuicksort [EW19] which limit the impact of branch prediction by avoiding certain conditional jumps.

Besides, Brodal and Moruz conducted an experimental study of skewed binary search trees in [BM06], highlighting that such data structures can outperform well-balanced trees, since branching to the right or left does not necessarily have the same cost, due to branch prediction schemes. Some of our work follows the same line, as we also want to take advantage of the branch predictions, but we focus on algorithms rather than on data structures. To that end, we examine classical divide-and-conquer algorithms, adjust their structure to unbalance the conditional jumps, and analyze the trade-offs between the number of comparisons and the number of mispredictions in the average case.

Markov chains in the analysis of branch prediction Since our main focus is on average-case analysis, we often enrich the algorithm's input with a probability distribution, which in turn leads to a decoration of the branch predictor with probabilities on its edges, effectively yielding a Markov chain. In our context, a *Markov chain* is defined by a non-empty finite set $\mathcal{S}$ of states, and a *transition matrix* M, where, for every $x, y \in \mathcal{S}$, $M(x, y)$ is the probability to go from state x to state y. In particular, it requires that for all $x \in \mathcal{S}$, $\sum_{y \in \mathcal{S}} M(x, y) = 1$. Usually, the definition is supplemented with an initial probability vector π_0 of the same dimension, with non-negative coefficients that sum to 1. One can follow a (random) *trajectory* $(x_t)_{t \geq 0}$ in the Markov chain by starting at a random position following the probability π_0 for x_0, at time $t = 0$, and then at each discrete step from time t to $t+1$, going from the current state x_t to x_{t+1} with probability $M(x_t, x_{t+1})$.

We shall rely on classical results on Markov chains in the sequel, which we now briefly introduce. A Markov chain is said to be *irreducible* when its underlying directed graph is strongly connected, and *aperiodic* when the gcd of its cycle lengths is 1. The key result we use is that, under these conditions, the chain admits a unique stationary distribution [LPW08]:

Theorem 27. *If (M, π_0) is a Markov chain on $\mathcal{S}$ that is irreducible and aperiodic, then there exists a unique probability line vector $\hat{\pi}$ on $\mathcal{S}$, called the* stationary vector*, such that $\hat{\pi} \cdot M = \hat{\pi}$ and if $(x_t)_{t \geq 0}$ is a trajectory then $\lim_{t \to \infty} \mathbb{P}(x_t = s) = \hat{\pi}(s)$, for all $s \in \mathcal{S}$.*

[3]Namely, conditional moves, such as `cmov`, which are now widely available on computers.

Classically, if the chain is not irreducible and has only one terminal strongly connected component $\mathcal{C}$ that is aperiodic, Theorem 27 still applies by considering the restriction to $\mathcal{C}$, as a random trajectory quickly ends in $\mathcal{C}$ with high probability.

Some of our results involve counting the number of mispredictions that occur during the execution of various algorithms. This task can often be reduced to counting how many times specific edges are taken in a Markov chain, when we perform a random walk of (random) length L_k. When this happens, we may be able to conclude using the classical Ergodic Theorem [LPW08], which we restated below in order to fit our needs.

Theorem 28 (Ergodic Theorem)**.** *Let (M, π_0) be a primitive and aperiodic Markov chain on the finite set S. Let $\hat{\pi}$ be its stationary distribution. Let E be a set of edges of M, that is, a set of pairs $(i, j) \in S^2$ such that $M(i, j) > 0$. For any nonnegative integer n, let L_n be a random variable taking values in the nonnegative integers such that $\lim_{n\to\infty} \mathbb{E}[L_n] = +\infty$. Let X_n be the random variable that counts the number of edges in E that are used during a random walk of length L_n in M (starting from the initial distribution π_0). Then the following asymptotic equivalence holds:*

$$\mathbb{E}[X_n] \sim \mathbb{E}[L_n] \sum_{(i,j)\in E} \hat{\pi}(i) M(i, j).$$

Now that our setting is established and our main tools have been introduced, we can proceed first with variants of classical comparison-based algorithms, and then with a study of pattern matching.

```
NaiveMinMax
Input: array $T$ of length $n$
1  $min \leftarrow T[0]$
2  $max \leftarrow T[0]$
3  for $i = 1$ to $n - 1$ do
4      if $T[i] < min$ then
5          $min \leftarrow T[i]$
6      if $T[i] > max$ then
7          $max \leftarrow T[i]$
8  return $min, max$
```

```
$\frac{3}{2}$-MinMax
Input: array $T$ of length $n$
1  $min \leftarrow T[n-1], max \leftarrow T[n-1]$
2  for $i \leftarrow 1;\ i < n-2;\ i \leftarrow i+2$ do
3      if $T[i] < T[i+1]$ then
4          if $T[i] < min$ then $min \leftarrow T[i]$
5          if $T[i+1] > max$ then $max \leftarrow T[i+1]$
6      else
7          if $T[i+1] < min$ then $min \leftarrow T[i+1]$
8          if $T[i] > max$ then $max \leftarrow T[i]$
9  return $min, max$
```

Figure 18: Naive and optimized algorithms for simultaneous minimum/maximum searching.

4 Unbalancing the Jumps

Branches with equally likely outcomes can naturally occur during the execution of an algorithm. For example, if the input follows a uniform distribution or is randomized, branches arising from element comparisons will be balanced, meaning they are equally likely to be taken or not. Divide-and-conquer algorithms often produce such branches as well, since they typically split problems into equally sized parts to achieve optimal complexity. However, these perfectly balanced conditions are particularly challenging for the type of branch predictor considered here: past information offers no guidance in predicting which side will prevail, and the predictor will be wrong roughly half of the time.

Our idea is to mitigate this effect by deliberately breaking this symmetry. We explore two strategies for disrupting the balance, aiming to improve performance on two classical algorithms. We take a look at exponentiation by squaring and give a simple alternative, slightly faster algorithm, which reduces the number of mispredictions without increasing the number of multiplications, by adding an unnecessary test. And in the same vein, we propose biased versions of the binary search in a sorted array, for which we analyze the expected number of mispredictions for the local predictor. For these two different problems, we manage to significantly lower the number of mispredictions by breaking the perfect balance usually favored in the divide and conquer strategy. In practice, the trade-off between comparisons and mispredictions allows a noticeable speed-up in the execution time, when the comparisons involve primitive data types, which supports our theoretical results.

This section presents a synthesis of work published jointly with Nicolas Auger, Mathilde Bouvel, and Cyril Nicaud [ANP16, ABNP16].

4.1 A case study : simultaneous minimum and maximum searching

We start with an introductory example using combinatorial arguments. Let us consider the simple problem of computing both the minimum and the maximum of an array of size n. The naive approach is to compare each entry to the current minimum and maximum, which uses $2n$ comparisons. This is Algorithm NaiveMinMax. A better solution, in terms of number of comparisons, is to look at the elements of the array two by two, and to compare the smallest

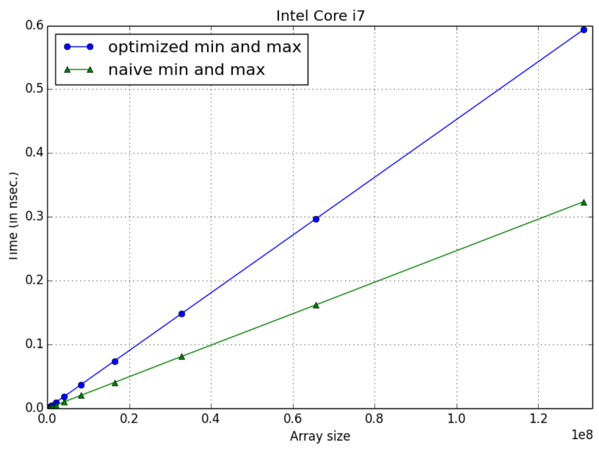


Figure 19: Execution time of simultaneous minimum and maximum searching.

to the current minimum and the greatest to the current maximum. This corresponds to Algorithm $\frac{3}{2}$-MinMax.

In the classical analysis setting, $\frac{3}{2}$-MinMax is optimal.[4], performing a number of comparisons asymptotically equivalent to $\frac{3}{2}n$. By contrast, NaiveMinMax needs $2n-2$ comparisons.

To observe the benefit of this optimization, we implemented both versions and measured their execution time[5] for large arrays of uniform random `float` in $[0, 1]$. The results are given in Fig. 19 and are very far from what was expected, since the naive implementation is almost twice as fast as the optimized one. Clearly, counting comparisons cannot explain these counterintuitive results. An obvious explanation could be a difference in the number of cache misses. However, both implementations make the same memory accesses, in the same order. Instead, we turn our attention to the comparisons themselves. As we have seen, most modern processors are heavily parallelized and use predictors to guess the outcome of conditional branches in order to avoid costly stalls in their pipelines. Since the cost of a misprediction can be quite large compared to a basic instruction, we believe this should be taken into account in order to explain accurately the behavior of algorithms that use a fair amount of comparisons.

In this matter, our example is quite revealing since the trick used to lower the number of comparisons relies on a conditional jump that is unpredictable (for an input taken uniformly at random) and will cause a substantial increase in the number of mispredictions. As we will see, the expected number of mispredictions caused by the naive algorithm is $\Theta(\log n)$, whereas it is $\Theta(n)$ for the "optimal" one.

In order to give an explanation of the experimental results presented in Fig. 19, where NaiveMinMax outperforms $\frac{3}{2}$-MinMax, we estimate the expected number of mispredictions for both algorithms. Our probabilistic model is the following: we consider the *uniform random distribution on arrays of size n*, where each element is chosen uniformly and independently in $[0, 1]$. Up to an event of probability 0 (when the elements of the input are not pairwise distinct), this is the same as choosing a uniform random permutation of $\{1, \dots, n\}$, since we only use comparisons on the elements in both algorithms.

Recall that a *min-record* (resp. *max-record*) in an array or a permutation is an element that is strictly smaller (resp. greater) than any element to its left. Obviously, in NaiveMinMax, the first condition at line 4 (resp. the one at line 5) is true for each min-record (resp. max-

[4] More precisely, an adversary argument can be used to establish a lower bound of $\lfloor \frac{3n}{2} \rfloor - 2$ comparisons, in the "decision tree with comparisons" model of computation [Poh72].

[5] We used a Linux machine with a 3.40 GHz Intel Core i7-2600 CPU.

record), except for the first position. The number of records in a random permutation is a well-known statistics, which we can use to establish the following proposition.

Proposition 29. *The expected number of mispredictions performed by* NaiveMinMax*, for the uniform distribution on arrays of size n, is asymptotically equivalent to $2\log n$ for the 2-bit saturating counter. The expected number of mispredictions performed by* $\frac{3}{2}$-MinMax *is asymptotically equivalent to $\frac{n}{4} + \mathcal{O}(\log n)$.*

Proof (sketch). We consider the 2-bit predictor in Fig. 17. Let σ be a permutation in $\mathfrak{S}_n$, whose cycles, ordered by decreasing order of their smallest element, are $\mathcal{C}_1, \ldots, \mathcal{C}_m$. We use Foata's bijection φ, described in Section 1.1.4, from $\mathfrak{S}_n$ onto itself, such that the number of min-records in $\varphi(\sigma)$ equals the number of cycles of σ. Hence, the expected number of records in a uniform random permutation is asymptotically equivalent to $\log n$ (see [FS09]). Our goal is to estimate the number of mispredictions $\xi(\varphi(\sigma))$ triggered by line 3 of NaiveMinMax applied to σ. Since $\varphi(\sigma) = \varphi(\mathcal{C}_1)\cdot\varphi(\mathcal{C}_2)\cdots\varphi(\mathcal{C}_m)$, we count the mispredictions cycle by cycle.

Assume that the predictor is in state ν just before processing $\varphi(\mathcal{C}_i)$. This state predicts that the branch is not taken. As the first element of $\varphi(\mathcal{C}_i)$ is a min-record, it causes a misprediction and the predictor switches to state τ (which predicts taken). If $\mathcal{C}_i$ has length at least 2, then the next element also causes a misprediction and the predictor goes back to ν. If it has length at least 3, then the predictor is set to $\underline{\nu}$ with no further misprediction. All useful cases are depicted in the following table:

starting	cycle of length 1		cycle of length 2		cycle of length 3		cycle of length ≥ 4	
state	mispred.	ending	mispred.	ending	mispred.	ending	mispred.	ending
$\underline{\nu}$	1	ν	1	$\underline{\nu}$	1	$\underline{\nu}$	1	$\underline{\nu}$
ν	1	τ	2	ν	2	$\underline{\nu}$	2	$\underline{\nu}$
τ	0	$\underline{\tau}$	1	τ	2	ν	2	$\underline{\nu}$
$\underline{\tau}$	0	$\underline{\tau}$	1	τ	2	ν	2	$\underline{\nu}$

From this, we readily get that the number $\chi(\varphi(\sigma))$ of mispredictions caused by the first conditional jump satisfies

$$\mathrm{cyc}_{\geq 4}(\sigma) \leq \chi(\varphi(\sigma)) \leq \mathrm{cyc}_{\geq 4}(\sigma) + 3\,\mathrm{cyc}_{\leq 3}(\sigma),$$

where $\mathrm{cyc}_i(\sigma)$ is the number of cycles of length i of σ, since there can be two mispredictions caused by $f(\mathcal{C}_i)$ only if $\mathcal{C}_{i-1}$ has length at most 3. We conclude by summing the contribution of both conditional jumps, as $\mathbb{E}_n[\mathrm{cyc}_{\leq 3}] = \mathcal{O}(1)$.

We now consider the algorithm $\frac{3}{2}$-MinMax. Using the model of n random numbers in $[0,1]$, it is straightforward to see that the first test in the loop of $\frac{3}{2}$-MinMax (line 3) causes a misprediction with probability $\frac{1}{2}$, for any saturating counter. Hence, this first test causes around $\frac{n}{4}$ mispredictions in average, when the algorithm ranges through the whole input. Moreover, the inner tests are true only when a min-record or max-record occurs. Using the same kinds of arguments as for Proposition 29, the expected number of mispredictions caused by these inner tests is in $\mathcal{O}(\log n)$, concluding the proof. □

In light of these results, we observe that the mispredictions occurring in NaiveMinMax are negligible compared to the number of comparisons. On the other hand, the additional test used to optimize $\frac{3}{2}$-MinMax (line 3) causes the number of mispredictions to be comparable to the number of comparisons performed. We believe this is enough to explain why the naive implementation performs better (Fig. 19), since we know that mispredictions can cost many CPU cycles and that comparisons are cheap operations. Of course, we are aware that other factors may influence the performance of such simple programs, including cache effects. In our implementation, we took care to fetch each element of the array only once and in the same order, so that the cache behavior should not interfere with our results. We also tried the most commonly used optimization of the `gcc` compiler (`-O3`) to check that these results withstand strong code optimization. In this particular case, all the branches but the one at line 3 in $\frac{3}{2}$-MinMax are replaced by conditional moves, which are not vulnerable to misprediction. Hence, $\frac{3}{2}$-MinMax still causes approximately $\frac{1}{4}n$ mispredictions on average; in practice, both algorithms are faster, as expected, but the naive version is still almost twice as fast.

Of course, these results no longer hold under a non-uniform distribution. We now examine the behavior of both algorithms on record-biased permutations, as introduced in Section 1.

4.1.1 Expected number of mispredictions on record-biased permutations

We have just seen that, for uniform permutations, NaiveMinMax outperforms $\frac{3}{2}$-MinMax, as the latter suffers more mispredictions, offsetting the benefit of performing fewer comparisons. This corresponds to the case of record-biased permutations with $\theta = 1$. However, as θ varies, the distribution of records changes accordingly, which in turn affects the performance of both algorithms. In particular, when $\theta = \lambda n$, the expected number of records becomes linear in n, in contrast to the logarithmic behavior observed when $\theta = 1$. We now analyze the number of mispredictions in NaiveMinMax and $\frac{3}{2}$-MinMax, applied to record-biased permutations, with a special focus on the case $\theta = \lambda n$, which exhibits significantly different behavior compared to the uniform distribution (see Fig. 20).

Expected number of mispredictions in NaiveMinMax Recall from the first part of this study that, among record-biased permutations of size n with parameter θ, the expected number of records is given by $\mathbb{E}_n[\mathrm{rec}] = \theta(\Psi(\theta+n) - \Psi(\theta))$, where Ψ is the digamma function.[6]

Theorem 30. *For record-biased permutations of size n with parameter θ, the expected numbers of mispredictions occurring at lines 4 and 6 of NaiveMinMax are respectively given by $\mathbb{E}_n[\mu_4] \leq \frac{2}{\theta}\mathbb{E}_n[\mathrm{rec}]$ and $\mathbb{E}_n[\mu_6] = 2\theta(\Psi(\theta+n-1) - \Psi(\theta)) - \frac{(2\theta+1)(n-1)}{\theta+n-1}$.*

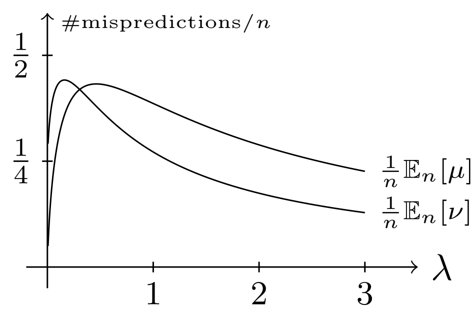
Consequently, the expected number of mispredictions at line 4 is $\mathcal{O}(\log n)$ when $\theta = \Omega(1)$, that is, when $\theta = \theta(n)$ is constant or larger. Furthermore, using the asymptotic expansion of the digamma function, we obtain the following asymptotic behavior for the expected number of mispredictions at line 6 (again, for $\lambda > 0$, $0 < \epsilon < 1$, and $\delta > 1$):

	fixed $\theta > 0$	$\theta := n^\epsilon$	$\theta := \lambda n$	$\theta := n^\delta$
$\mathbb{E}_n[\mu_6]$	$\sim 2\theta \cdot \log n$	$\sim 2(1-\epsilon) \cdot n^\epsilon \log n$	$\sim 2\lambda(\log(1+1/\lambda) - 1/(\lambda+1)) \cdot n$	$o(n)$

In particular, asymptotically, the expected total number of mispredictions of NaiveMinMax is given by $\mathbb{E}_n[\mu_6]$ (up to a constant factor when θ is constant).

Expected number of mispredictions in $\frac{3}{2}$-MinMax Mispredictions in $\frac{3}{2}$-MinMax may occur at any of the three `if` statements. We analyze the expected number of mispredictions at each of them independently. We start with the `if` statement at line 3, which compares $T[i-1]$ and $T[i]$. For a 1-bit saturating counter, a misprediction occurs whenever there is a

[6] As mentioned in Section 1.3, the digamma function is defined by $\Psi(x) = \Gamma'(x)/\Gamma(x)$, where Γ is the classical gamma function. As $x \to +\infty$, it admits the asymptotic expansion $\Psi(x) = \log(x) - \frac{1}{2x} - \frac{1}{12x^2} + o\left(\frac{1}{x^2}\right)$.



We have $\mathbb{E}_n[\mu] \sim \mathbb{E}_n[\mu^\star]$ for $\lambda_0 = \frac{\sqrt{34}-4}{6} \approx 0.305$. Algorithm $\frac{3}{2}$-MinMax incurs fewer mispredictions on average than NaiveMinMax as soon as $\lambda > \lambda_0$. However, since $\frac{3}{2}$-MinMax performs $\frac{n}{2}$ fewer comparisons than the naive algorithm, it becomes more efficient before λ_0. For instance, if a misprediction is worth 4 comparisons, $\frac{3}{2}$-MinMax becomes more efficient as soon as $\lambda > 0.110$.

Figure 20: The expected number of mispredictions produced by NaiveMinMax (μ) and for $\frac{3}{2}$-MinMax ($\mu^\star$), when $\theta := \lambda n$.

descent at $i-2$ and an ascent at i, or an ascent at i and a descent at $i-2$. For the 2-bit saturating counter, a tedious case analysis yields a precise estimate of $\mathbb{E}_n[\mu_3^\star]$, the expected number of mispredictions at line 3 (the first `if`) of $\frac{3}{2}$-MinMax, for any parameter θ. The full statement of this intermediate result is omitted for conciseness. As an illustrative example, when $\theta = \lambda n$, we obtain $\mathbb{E}_n[\mu_3^\star] \sim \frac{6\lambda^2+8\lambda+3}{12(\lambda+1)^3}\, n$.

For the second `if`, the statement is simpler: the expected number of mispredictions at line 7 of $\frac{3}{2}$-MinMax satisfies $\mathbb{E}_n[\mu_7^\star] \leq \frac{2}{\theta}\mathbb{E}_n[\text{rec}]$. As a result, if $\theta = \lambda n$, then $\mathbb{E}_n[\mu_7^\star] = \mathcal{O}(1)$.

We finally turn to the third `if` (line 8) of Algorithm $\frac{3}{2}$-MinMax. If there is a record (resp. no record) at position $i-3$ or $i-2$, then a misprediction occurs when there is no record (resp. a record) at position $i-1$ or i. A case-by-case analysis of all possible configurations at these four positions yields, once again, a precise estimate for $\mathbb{E}_n[\mu_8^\star]$; we omit the details for brevity. From this, one has, for example, that if $\theta = \lambda n$, then $\mathbb{E}_n[\mu_8^\star] \sim \left(2\lambda \log\left(1+\frac{1}{\lambda}\right) - \frac{\lambda(6\lambda^2+15\lambda+10)}{3(\lambda+1)^3}\right) n$.

Combining these results yields the following.

Theorem 31. *For record-biased permutations of size n with parameter $\theta = \lambda n$, the total number of mispredictions of $\frac{3}{2}$-MinMax is*

$$\mathbb{E}_n[\mu^\star] \sim \left(2\lambda \log\left(1+\frac{1}{\lambda}\right) - \frac{24\lambda^3+54\lambda^2+32\lambda-3}{12(\lambda+1)^3}\right) n.$$

Fig. 20 shows that, unlike in the uniform case ($\theta = 1$), $\frac{3}{2}$-MinMax is more efficient than NaiveMinMax on record-biased permutations with $\theta := \lambda n$, as soon as λ is large enough.

4.2 Exponentiation by squaring with a twist

The example in the previous section shows that reducing the average number of mispredictions during the execution of an algorithm can significantly improve efficiency. Here, we explore a first application of this idea by using a small trick to intentionally unbalance the jump conditions in exponentiation by squaring. This also leads to a trade-off: fewer mispredictions at the cost of performing slightly more comparisons, again resulting in a faster implementation.

4.2.1 Modified algorithms

The classical divide-and-conquer algorithm to compute x^n is based on rewriting

$$x^n = (x^2)^{\lfloor n/2 \rfloor} x^{n_0},$$

where $n_k \dots n_1 n_0$ is the binary decomposition of n, in order to divide the size n of the problem by two. This is the Algorithm ClassicalPow of Fig. 21. As expected, the branch of line 3

```
ClassicalPow

Input: $x, n$
1  $r \leftarrow 1$
2  while $n > 0$ do
       // $n$ is odd
3      if $n$ & 1 then
4          $r \leftarrow r * x$
5      $n \leftarrow n/2$
6      $x \leftarrow x * x$
7  return $r$
```

For each algorithm, x is a floating-point number, n is an integer and the returned value r is x^n.

```
UnrolledPow

Input: $x, n$
1  $r \leftarrow 1$
2  while $n > 0$ do
3      $t \leftarrow x * x$
       // $n_0 = 1$
4      if $n$ & 1 then
5          $r \leftarrow r * x$
       // $n_1 = 1$
6      if $n$ & 2 then
7          $r \leftarrow r * x$
8      $n \leftarrow n/4$
9      $x \leftarrow t * t$
10 return $r$
```

```
GuidedPow

Input: $x, n$
1  $r \leftarrow 1$
2  while $n > 0$ do
3      $t \leftarrow x * x$
       // $n_1 n_0 \neq 00$
4      if $n$ & 3 then
5          if $n$ & 1 then
6              $r \leftarrow r * x$
7          if $n$ & 2 then
8              $r \leftarrow r * t$
9      $n \leftarrow n/4$
10     $x \leftarrow t * t$
11 return $r$
```

Figure 21: Three versions of exponentiation by squaring. The `&` operator is a bitwise `AND`.

is taken with probability $\frac{1}{2}$, which is what we want to avoid.[7] In order to introduce some imbalance in the algorithm, we first unroll the loop (UnrolledPow, Fig. 21) using the decomposition $x^n = (x^4)^{\lfloor n/4 \rfloor}(x^2)^{n_1}x^{n_0}$. Still, both branches are taken with probability $\frac{1}{2}$, but we can now guide the algorithm by injecting the test that determines whether the last two bits of n are 11 or not. This is the third algorithm of Fig. 21. Note that this branch (line 4) is absolutely unnecessary in the algorithm, as it is redundant with the tests of line 5 and 7. But on the other hand, this branch is taken with probability $\frac{3}{4}$ and the branches of line 5 and 7 are now both taken with probability $\frac{2}{3}$. This is how we aim to use the branch predictions.

To compare their performances experimentally, we computed the floating-point value of x^n using each of the algorithms 5.10^7 times, with n chosen uniformly at random in $\{0, \dots, 2^{26}-1\}$. We measured the execution time, as well as some other parameters given by the `PAPI` library,[8] which give access, for instance, to the number of mispredictions occurring during the execution. These results are depicted on Fig. 22. The first observation is that GuidedPow is 14% faster than UnrolledPow and 29% faster than ClassicalPow and yet, the number of multiplications performed is essentially the same for the three algorithms. The main explanation we have come across for the speed-up between UnrolledPow and ClassicalPow is that the number of loops is divided by two. As for GuidedPow, the number of loops is the same as for UnrolledPow and it uses 25% more comparisons, but still the guided version is faster. The main difference between the two is that the test added at line 4 allows us to decrease the number of mispredictions by about a quarter (this test causes additional mispredictions, but it also modifies the probabilities associated to the inner `if` instructions of line 5 and 7 (corresponding to line 4 and 6 in UnrolledPow), which leads to an overall decrease in the number of mispredictions). We are in similar setting to the simultaneous minimum and maximum, where the increased number of comparisons is balanced by fewer mispredictions. We now proceed with the analysis of this phenomenon.

[7]In our model, n is chosen uniformly at random between 0 and $4^k - 1$ for some positive k.
[8]`PAPI 5.4.1.0` , see http://icl.cs.utk.edu/papi.

Pow	time (in sec.)	loops $\times 10^9$	mult. $\times 10^9$	branches $\times 10^9$	mispred. $\times 10^9$
classical	7.230	1.250	1.900	1.300	0.674
unrolled	6.316	0.633	1.917	1.317	0.683
guided	5.606	0.633	1.917	1.658	0.554

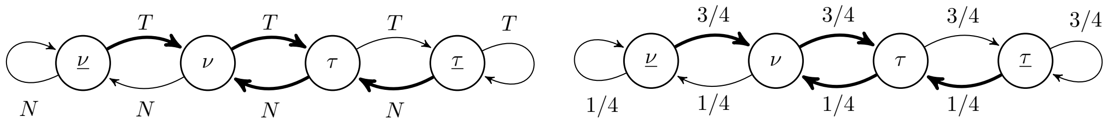
Figure 22: Statistics measured during 5.10^7 computations of x^n with the three algorithms of Fig. 21, using the `PAPI` library. The values of n are chosen uniformly at random in $[0, 2^{26}[$. The number of branches is given excluding the loop ones, as they do not yield mispredictions.

4.2.2 Analysis of the average number of mispredictions in GuidedPow

For the analysis, we consider that n is taken uniformly at random in $\{0, \dots, N-1\}$, for $N = 4^k$ and with $k \geq 1$. This model is exactly the same as choosing each of the $2k$ bits of the binary representation of n uniformly at random and independently. We consider the local predictors presented in Section 3.

Let $L_k(n)$ be the number of loop iterations of GuidedPow. This is a random variable, which is easy to analyze since it is equal to the smallest integer ℓ such that 4^ℓ is greater than n. In particular, we have $\mathbb{E}[L_k] = k - \frac{1}{3} + o(1) \sim k$.

As stated earlier, Markov chains are the key tools for that kind of analysis (as done in [KS06, MNW15]). Let us consider the first conditional jump of line 4. In our model, at each iteration, the condition is true with probability $\frac{3}{4}$, as it is not satisfied when the last two bits are 00. It yields that the behavior of the predictor associated to this conditional jump is exactly described by the Markov chain obtained when changing the edges labels T by $\frac{3}{4}$ and the labels N by $\frac{1}{4}$ in the 2-bit saturating counter:



A misprediction occurs whenever an edge labeled by N (resp. T) is used from a state that predicts the branch will be taken (resp. not taken). We also need to know the initial state of the predictor, but it has no influence on our asymptotic results, as we shall see.

Hence, we reduced our problem to counting the number of times some particular edges are taken in a Markov chain, when we perform a random walk of (random) length L_k. We can therefore conclude using the version of the Ergodic Theorem (Th. 28) given in Section 5.2.3.

We consider the model where the condition is satisfied with probability p, we denote by M_p the transition matrix of the Markov chain associated to the predictor, by $\hat{\pi}_p$ its stationary vector and by $\mu(p)$ its *expected misprediction probability* defined by $\mu(p) = \sum_{(i,j)\in E} \hat{\pi}_p(i) M_p(i,j)$, where E is the set of edges corresponding to mispredictions. As shown in [MNW15], the expected misprediction probability of the 2-bit saturating counter is

$$\mu(p) = \frac{p(1-p)}{1-2p(1-p)}. \tag{7}$$

Applying this to GuidedPow yields the following results. The theorem is stated for values of N that are not powers of 4, which is more complicated since the bits are not exactly 0's and 1's with probability $\frac{1}{2}$ (and not independent). In Section 4.3 we show how to deal with the cases where we slightly deviate from the ideal case.

Theorem 32. *Assume that n is taken uniformly at random in $\{0,\dots,N-1\}$. The expected number of conditional jumps in* ClassicalPow *and* UnrolledPow *is asymptotically equivalent to $\log_2 N$, whereas it is asymptotically equivalent to $\frac{5}{4}\log_2 N$ for* GuidedPow.
The expected number of mispredictions for a 2-bit saturating counter is asymptotically equivalent to $\frac{1}{2}\log_2 N$ for ClassicalPow *and* UnrolledPow. *For* GuidedPow, *it is asymptotically equivalent to $\alpha\log_2 N$, where $\alpha = \frac{1}{2}\mu(3/4) + \frac{3}{4}\mu(2/3)$.*

These results can easily be extended to other local predictors described as automata. For instance, the expected misprediction probability $\mu_1(p)$ and $\mu_3(p)$ of the 1-bit and 3-bit saturating counters are

$$\mu_1(p) = 2p(1-p) \quad \text{and} \quad \mu_3(p) = \frac{p(1-p)\,(1-3p(1-p))}{1-2p(1-p)\,(2-p(1-p))}. \tag{8}$$

Using Theorem 32 and Eq. (7), we get that for the 2-bit saturating counter, α is equal to $\frac{9}{20} = 0.45$, instead of $\frac{1}{2}$ for the conditional jumps of the other two algorithms. With a 1-bit predictor, the expected number of mispredictions for GuidedPow is $\frac{25}{48} \approx 0.52$, which is greater than for ClassicalPow or UnrolledPow. This predictor is not efficient enough to offset the mispredictions caused by the additional conditional. And with a 3-bit saturating counter, GuidedPow therefore uses $\approx 0.25\log_2 n$ more comparisons than UnrolledPow, but $\frac{1095}{2788} \approx 0.39$ mispredictions, which is $\approx 0.11\log_2 n$ less mispredictions.

These results might suggest that the branch predictor used on our machine behaves like a 3-bit saturating counter. However, we cannot confirm this with certainty. Additionally, since we introduced slight dependency between the outcomes of our conditional jumps, the observed performance gains may in fact be due to the influence of a global predictor. We will revisit this line of investigation at the end of the study. For now, we explore a different approach to unbalancing the branches, this time by breaking the symmetry in binary search.

4.3 Skew binary search

Consider the classical binary search, which partitions a sorted array of size n into two parts of size $\frac{n}{2}$ and compares the target value x with the middle element of the array to determine in which part of the array the search should continue. As before, when the array consists of uniform random floating-point numbers, the corresponding branch is taken with probability $\frac{1}{2}$. Once again, we explore ways to break this perfect balance in order to give the local branch predictor an advantage.

4.3.1 Unbalancing the binary search

A simple way to change the probabilities in the binary search is to partition the input differently, for instance into parts of size roughly $\frac{n}{4}$ and $\frac{3n}{4}$, as in BiasedBinarySearch (see Fig. 23). Carrying on with the divide-and-conquer strategy but partitioning the array into three parts of size about $\frac{n}{3}$, gives a ternary search. The main issue with this approach is that, in practice, the division by 3 is expensive in hardware. Thus, to limit the cost of partitioning, we choose to slice the array into two parts of size $\frac{n}{4}$ and one part of size $\frac{n}{2}$. This can be done using only divisions by powers of two, which are simple binary shifts, as in the initial binary search (see SkewSearch in Fig. 23).

As expected at this point in our work, BiasedBinarySearch experimentally performs slightly better than the classical binary search and SkewSearch is significantly faster. Unlike our previous examples, the changes we brought in the binary search are quite sensitive to cache effects, since the way we partition the array influences the location where the memory

```
BiasedBinarySearch
Input: $T, x$
1  $d \leftarrow 0, f \leftarrow n$
2  while $d < f$ do
3      $m \leftarrow (3 * d + f)/4$
4      if $T[m] < x$ then
5          $d \leftarrow m + 1$
6      else
7          $f \leftarrow m$
8  return $f$
```

In both cases, T is an array of floats of size n and x is the number that is searched for. The classical binary search is obtained by replacing line 3 of BiasedBinarySearch by $m = (d + f)/2$.

```
SkewSearch
Input: $T, x$
1  $d \leftarrow 0, f \leftarrow n$
2  while $d < f$ do
3      $m_1 \leftarrow (3 * d + f)/4$
4      if $T[m_1] > x$ then
5          $f \leftarrow m_1$
6      else
7          $m_2 \leftarrow (d + f)/2$
8          if $T[m_2] > x$ then
9              $f \leftarrow m_2$
10             $d \leftarrow m_1 + 1$
11         else
12             $d \leftarrow m_2 + 1$
13 return $f$
```

Figure 23: Algorithms for the biased binary search and skew search. Both return the position where the element should be inserted.

is accessed. Thus we conducted experiments on arrays that fit in the last-level cache of our machine[5] in order to mostly measure the effects of branch prediction. The results are given by Fig. 24 and we can see that, for medium-size arrays, SkewSearch is up to 23% faster than the binary search (program compiled with `gcc` without optimization, in order to keep track of what really happens during the execution). Experiments in Java using a dedicated micro-benchmarking library[9] gave roughly the same results (but with a lesser speedup of about 12%), when comparing our skew search to the implementation of the binary search on doubles in the standard library.

4.3.2 Average number of mispredictions in biased variants of binary search

As in Section 4.2, we rely on the Ergodic Theorem (page 42) to derive an accurate asymptotic estimate of the number of mispredictions. To this end, we first need to compute the expected number of times each conditional jump is executed in the different algorithms. We assume that each possible output is equally likely i.e., that the distribution is uniform on $\{0, \dots, n\}$).

A first-order estimate of the expected number of executions of a given conditional jump can be obtained using the following adaptation of Roura's Master Theorem [Rou01], specialized to our setting.[10]

Theorem 33 (Master Theorem). *Let $k \geq 1$, and $a_1, \dots, a_k$ and $b_1, \dots, b_k$ be positive real numbers such that $\sum_{i=1}^{k} a_i = 1$. For every $i \in \{1, \dots, k\}$, let also $\varepsilon_i(n)$ be a real valued sequence such that $b_i n + \varepsilon_i(n)$ is a positive integer and $\varepsilon_i(n) = \mathcal{O}(\frac{1}{n})$. Let $T(n)$ be the real*

[9]Benchmark using jmh: http://openjdk.java.net/projects/code-tools/jmh/ Our algorithms are compared to the method `Arrays.binarySearch(double[] a, double key)` of the Java standard API.
[10]For more general statements, see the seminal work of Roura [Rou01].

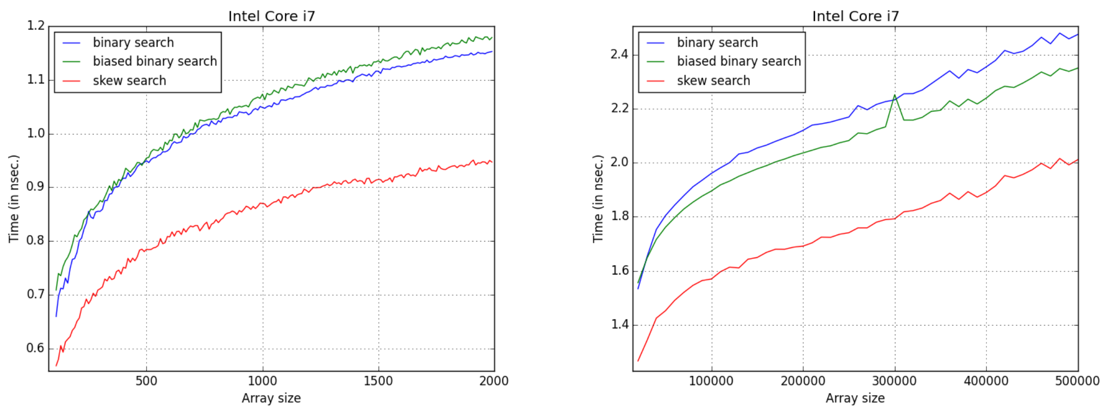


Figure 24: Execution time of the three searching algorithms of Fig. 23 for small-size arrays (that fit in the first-level cache) and medium-size arrays (that fit in the last-level cache).

valued sequence that satisfies, for some positive constants c and d,

$$T(0) = c \quad \textit{and} \quad T(n) = d + \sum_{i=1}^{k} a_i T\left(b_i n + \varepsilon_i(n)\right) + \mathcal{O}\left(\frac{\log n}{n}\right) \textit{ for } n \geq 1.$$

Then $T(n) \sim \frac{d}{h}\log n$, with $h = -\sum_{i=1}^{k} a_i \log b_i$.

Before stating our main result, we describe the main steps of our analysis on the algorithm BiasedBinarySearch. The expected number of iterations $L(n)$ of BiasedBinarySearch satisfies the relation

$$L(n) = 1 + \frac{a_n}{n+1} L\left(a_n\right) + \frac{b_n}{n+1} L\left(b_n\right), \text{ with } a_n = \left\lfloor \frac{n}{4} \right\rfloor + 1, b_n = \left\lceil \frac{3n}{4} \right\rceil \text{ and } L(0) = 0.$$

Thus, Theorem 33 applies and $L(n) \sim \lambda \log n$, with $\lambda = \frac{4}{4\log 4 - 3\log 3} \approx 1.78$.

Unfortunately, we cannot directly model the predictor as a Markov chain, as we did in Section 4.2, because the probabilities $\frac{a_n}{n+1}$ and $\frac{b_n}{n+1}$ are no longer fixed (they slightly depend on n). However, since $\frac{a_n}{n+1} = \frac{1}{4} + \mathcal{O}(\frac{1}{n})$ and $\frac{b_n}{n+1} = \frac{3}{4} + \mathcal{O}(\frac{1}{n})$, this Markov chain should still provide a good approximation of the number of mispredictions with Theorem 28.

To formalize this idea, we introduce the *decomposition tree* $\mathcal{T}$ associated with the search algorithms. If the input has size n, its root is labeled by the pair $(0, n)$, and each node corresponds to the possible values of d and f during one loop of the algorithm. The leaves are the pairs (i, i), for $i \in \{0, \ldots, n\}$; they are identified with the output of the algorithm in $\{0, \ldots, n\}$. There is a direct edge between (d, f) and (d', f') whenever the variables d and f can be changed into d' and f' during the current iteration of the loop. Such an edge is labeled with the probability $\frac{f'-d'+1}{f-d+1}$, which is the probability that this update happens in our model. An example of such a decomposition tree for BiasedBinarySearch is depicted on Fig. 25.

By construction, following a path from the root to a leaf, choosing between left and right according to the edge probability is exactly the same as choosing an integer uniformly at random in $\{0, \ldots, n\}$. Let $u = (u_0, u_1, \ldots)$ be an infinite sequence of elements of $[0, 1]$ taken uniformly at random and independently. To u is associated its path $\mathrm{Path}_n(\mathcal{T}, u)$ in $\mathcal{T}$ where, at step i, we go left if u_i is smaller than the left child edge probability and right otherwise. Let $L_n(\mathcal{T}, u)$ be the length of $\mathrm{Path}_n(\mathcal{T}, u)$. Let also $\mathrm{Path}_n(\mathcal{I}, u)$ be the path following the values in u in the ideal (infinite) tree $\mathcal{I}$ where we go left with probability $\frac{1}{4}$ and right with probability $\frac{3}{4}$. The following lemma shows that these constructions agree for almost all steps.

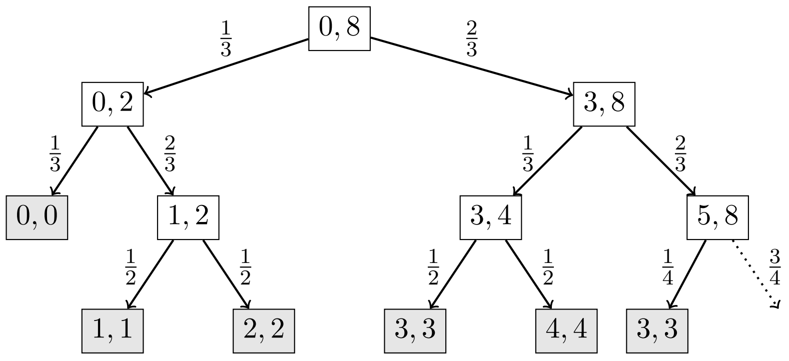


Figure 25: The decomposition tree of BiasedBinarySearch for $n = 8$.

Lemma 34. *The probability that* $\mathrm{Path}_n(\mathcal{T}, u)$ *and* $\mathrm{Path}_n(\mathcal{I}, u)$ *differ at one of the first* $L_n(\mathcal{T}, u) - \sqrt{\log n}$ *steps is* $\mathcal{O}(\frac{1}{\log n})$.

Hence, for most iterations of its main loop, Algorithm BiasedBinarySearch behaves almost exactly like its idealized version, and the error term can be estimated with sufficient precision. This is enough to prove that the idealized version is a correct first order approximation of the number of mispredictions. The same construction applies to all three algorithms, leading to Theorem 35. For instance, with a 2-bit saturating counter, $\mu(\frac{1}{4}) = \frac{3}{10}$ and $\mu(\frac{1}{3}) = \frac{2}{5}$, thus $\mathbb{E}[C_n]/\log(n)$ is around 1.44, 1.78 and 1.68 for the binary, biased and skew search respectively, while $\mathbb{E}[M_n]/\log(n)$ is around 0.72, 0.53 and 0.58.

Theorem 35. *Let* C_n *and* M_n *be the number of comparisons and mispredictions performed in our model of randomness. The following table gives asymptotic equivalents,*

	BinarySearch	BiasedBinarySearch	SkewSearch
$\mathbb{E}[C_n]$	$\log n/\log 2$	$4\log n/(4\log 4 - 3\log 3)$	$7\log n/(6\log 2)$
$\mathbb{E}[M_n]$	$\log n/(2\log 2)$	$\mu(1/4)\mathbb{E}[C_n]$	$\big(4\mu(1/4)/7 + 3\mu(1/3)/7\big)\mathbb{E}[C_n]$

where μ *is the expected misprediction probability associated with the predictor.*

4.3.3 Analysis of a global predictor for SkewSearch

In this section we intend to give hints about the behavior of a global branch predictor, such as the one depicted on Fig. 26, for the algorithm SkewSearch. Notice in particular that the predictor of each entry is a 2-bit saturating counter. This is not the only possible choice for a global predictor, but it is simple enough without being trivial. We make the analysis in the idealized framework that resembles the real case sufficiently well, by ignoring the rounding effects of dealing with integers. We saw in the previous section why these approximations still give the correct result for the first order asymptotic.

In our idealized model we only consider the sequence of choices produced by the two conditional jumps of SkewSearch. We deliberately do not consider the branch induced by the `while` loop, which would never be taken in our setting (except for the very last step). Adding it would complicate the model without adding interesting information for the branch predictor.[11] The trace of an execution of the algorithm, in terms of jumps, is a non-empty word on the binary alphabet $B = \{N, T\}$. Because of the way the two conditional jumps are nested within the algorithm, we can keep track of the current `if` instruction using the simple

[11]Also, most modern architectures have *loop detectors* that are used to identify such conditional jumps.

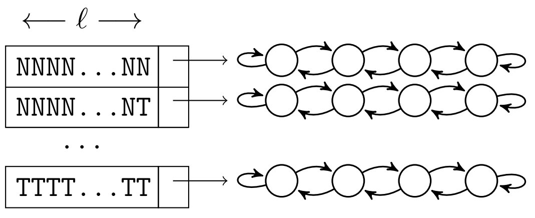


Figure 26: A fully global predictor: the history table of size 2^ℓ keeps track of the outcomes of the last ℓ branches encountered during the execution, the last one corresponding to the rightmost bit. To each sequence of ℓ branches is associated a global 2-bit predictor (shared by all the conditional jumps).

deterministic automaton $\mathcal{A}_{\text{if}}$ with two states depicted in Fig. 27: `main` stands for the first `if` and `nested` for the second one. In our model, the branch arising from `main` is taken with probability $\frac{1}{4}$ and the one from `nested` with probability $\frac{1}{3}$. As done in Section 4.2, $\mathcal{A}_{\text{if}}$ can be changed into a Markov chain $\mathcal{M}_{\text{if}}$ using these transition probabilities. A direct computation shows that its stationary vector $\hat{\pi}_{\text{if}}$ satisfies $\hat{\pi}_{\text{if}}(\texttt{main}) = \frac{4}{7}$ and $\hat{\pi}_{\text{if}}(\texttt{nested}) = \frac{3}{7}$.

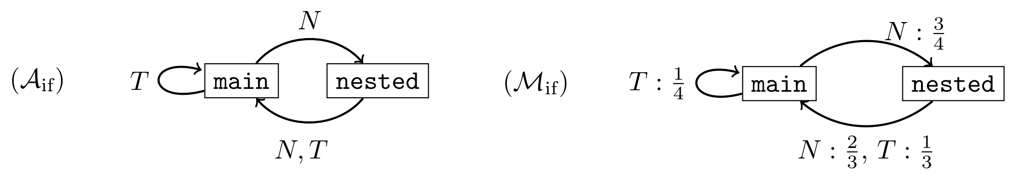

($\mathcal{A}_{\text{if}}$)
T
main
nested
N
N, T
($\mathcal{M}_{\text{if}}$)
$T : \frac{1}{4}$
main
nested
$N : \frac{3}{4}$
$N : \frac{2}{3}$, $T : \frac{1}{3}$

Figure 27: On the left, the automaton $\mathcal{A}_{\text{if}}$. On the right, the Markov chain $\mathcal{M}_{\text{if}}$ of transition probabilities $\mathbb{P}(T \mid \texttt{main}) = \frac{1}{4}$, $\mathbb{P}(N \mid \texttt{main}) = \frac{3}{4}$, $\mathbb{P}(N \mid \texttt{nested}) = \frac{2}{3}$ and $\mathbb{P}(T \mid \texttt{nested}) = \frac{1}{3}$.

For the same reason as above, in the global table, we only record the history for the two conditional jumps `main` and `nested`. Let ℓ denote the history length, that is, the number of bits used in the history table of Fig. 26. We assume that ℓ is even. An *history* h is thus seen as a binary word of length ℓ. Let N^ℓ be the history made of N's only.

When a test is performed at time t, the predictor uses the entry at position h_t to make the prediction, where h_t is the current history. To track the evolution of the algorithm at time $t+1$, we therefore to maintain (1) the history table H_t, (2) the current history h_t and (3) which of the two conditional jumps $\texttt{IF}_t$ is currently under consideration. Knowing $\texttt{IF}_t$ is required in order to compute the probability that the next outcome is N or T. This defines a Markov chain $\mathcal{M}_{\text{up}}$ for the updates in the history table. From $\mathcal{M}_{\text{up}}$, one can theoretically estimate the expected number of mispredictions using the Ergodic Theorem, as we did for local predictors. The main issue with this approach is that computing π_{up} is typically in $\mathcal{O}(m^3)$, where m is the number of states of $\mathcal{M}_{\text{up}}$. Since the number of states is exponential in ℓ, the computations are completely intractable for reasonable history lengths (such as $\ell \geq 6$), even if we first remove the unreachable states. However, $\mathcal{M}_{\text{up}}$ has a particular structure that we can use to directly compute the typical number of mispredictions.

Let $h \in B^\ell$ be a history that is not equal to N^ℓ. There is at least one T in h. Since reading a T always sends to state `main` in $\mathcal{A}_{\text{if}}$, we know for sure the conditional $\texttt{IF}_t$ under consideration when an occurrence of h has just happened at time t. Hence, we know the probability of obtaining N or T at time $t+1$, given that $h_t = h$. As a consequence, each entry of $h \neq N^\ell$ in the table H behaves like a fixed-probability local 2-bit saturating predictor, with probability $\frac{1}{4}$ for histories associated with `main` and $\frac{1}{3}$ for those associated with `nested`. Therefore, $h = N^\ell$ concentrates all the differences between the local and the global predictors.

What happens for the entry N^ℓ is well described by considering the automaton on pairs (s, i), where s is a state of the predictor and i is the current `if` instruction. This automaton can be turned into a Markov chain, and the Ergodic Theorem yields a precise estimation of the number of mispredictions. Following this idea yields the following result.

Theorem 36. *For the global predictor, the average number of mispredictions caused during* SkewSearch *on an input of size n is asymptotically equivalent to* $(\frac{12}{35} + \frac{1}{595 \cdot 2^\ell})\mathbb{E}[C_n]$.

By Theorem 35, if we use a local 2-bit predictor for each conditional jump, the expected number of mispredictions is asymptotically equivalent to $\frac{12}{35}\mathbb{E}[C_n]$. The difference with the global predictor is therefore extremely small, which is not surprising as there is a difference only when the history is N^ℓ. However, if there is a competition between a global predictor and a more accurate local predictor (a 3-bit saturating counter for instance), then the local predictor performs better; it is probably slightly disrupted by the global one, as the dynamic selector between both predictors can choose to follow the global predictor from time to time.

We have seen three examples of algorithms that rely heavily on conditional jumps, most of which are independent, making them well-suited for analyzing the behavior of local branch predictors. But what happens when the outcomes of these conditional jumps are correlated? We address this question in the next section and then we will conclude with a more in-depth discussion of global predictors.

5 Pattern Matching

To obtain meaningful results on the cost of branch prediction, we studied divide-and-conquer algorithms, as they make frequent use of branching instructions. These algorithms are natural candidates for such an analysis; however, when assuming a uniform distribution over the input, most of the conditional jumps they perform are independent. As a result, they do not pose a significant challenge for local branch predictors, which are well-suited for handling uncorrelated branches. Different branch behavior could be induced by refining the probabilistic model to incorporate dependencies, using Markov chains for instance. Here, however, we take a different approach: we explore another family of algorithms that inherently give rise to correlated branch conditions, namely, pattern matching algorithms.

As an illustration, consider the most naive approach to pattern matching: the pattern is compared to the text character by character, starting at the first position. If a mismatch occurs, the pattern is shifted by one position and the process starts over. For example, with the pattern *aabaab*, if a match occurs at the beginning of the text, then by the third alignment, we already know that the fourth to sixth characters in the text will match the beginning of the pattern again. One might imagine that a branch predictor capable of exploiting this kind of correlation could perform very well on this type of algorithm.

This is the question we aim to address in this section. We begin with the most basic pattern matching algorithm, examine the impact of branch prediction on its performance, and then move on to more advanced algorithms, namely, Morris-Pratt and Knuth-Morris-Pratt.

This work was conducted with Cyril Nicaud and Stéphane Vialette [NPV24, NPV25].

Probabilistic model for the analysis of branch prediction We study textbook solutions to the pattern matching problem, that is, counting the number of occurrences of a pattern X in a text W [CR94, Gus97, CHL07]. We consider a probabilistic model where the pattern is of length $m \geq 2$, and the text W, of length n, is generated using a memoryless source of non-degenerated probability distribution π on a fixed alphabet A. In our setting, throughout this study, X is fixed while n tends to infinity.

Let π be a probability measure on A such that for all $\alpha \in A$, $0 < \pi(\alpha) < 1$ (we also use the notation $\pi_\alpha := \pi(\alpha)$ in formulas when convenient). For each $n \geq 0$ and each $W \in A^n$, we define $\pi_n(W) := \prod_{i=0}^{n-1} \pi(W_i)$. For any n, the measure π_n is a probability on A^n, where all letters are chosen independently following π. We obtain the uniform distribution on A^n if π is the uniform distribution on A, with $\pi(\alpha) = \frac{1}{|A|}$ for all $\alpha \in A$.

When considering texts and patterns, indices start at 0: if u is a word of length $|u| = n$ over the alphabet A, we represent it as $u = u_0 \dots u_{n-1}$, where each u_i is a letter of u. We also use $u[i]$ to denote the letter u_i. If $u = xyz$ where x, y and z are words, then x is a *prefix* of u, y is a *factor* and z is a *suffix*. A prefix (resp. suffix) of u is *strict* if it differs from u. For any $i \in \{0, \dots, n\}$, we denote by $\mathrm{Pref}(u, i)$ the prefix of u of length i. A *(strict) border* of u is a word v that is both a (strict) prefix and a (strict) suffix of u.

As before, our analysis depends on the type of predictor used (see Section 3). All results are stated for the local 2-bit saturating counter (see Fig. 17).

5.1 The sliding window algorithm

We begin with the most elementary pattern matching algorithm: the sliding window algorithm. Not only is this still the algorithm used for simple pattern searches in a `String` in Java,[12] but analyzing it, even under a model restricted to local branch predictors, already

[12]Specifically, in the `String.indexOf(String str)` method of the Java standard API.

Natural Sliding Window

```
Input: pattern $X$ of size $m$,
       text $W$ of size $n$
1  $i, nb \leftarrow 0, 0$
2  while $i \leq n - m$ do
3      $j \leftarrow 0$
4      while $j < m$ and
5              $X[j] = W[i + j]$ do
6          $j \leftarrow j + 1$
7      if $j = m$ then $nb \leftarrow nb + 1$
8      $i \leftarrow i + 1$
9  return $nb$
```

On the left is a natural way to write the algorithm and SlidingWindow is a variation that splits the condition of the window loop.

SlidingWindow

```
Input: pattern $X$ of size $m$,
       text $W$ of size $n$
1  $i, nb \leftarrow 0, 0$
2  while $i \leq n - m$ do
3      $j \leftarrow 0$
4      while $True$ do
5          if $j = m$ then
6              $nb \leftarrow nb + 1$
7              break
8          if $X[j] = W[i + j]$ then
9              $j \leftarrow j + 1$
10         else break
11     $i \leftarrow i + 1$
12 return $nb$
```

Figure 28: Two versions of the sliding window algorithm to count the number of occurrences of a pattern X in the text W.

leads to nontrivial results. There are two main reasons for this. First, since we are interested in average-case behavior, we must estimate the number of mispredictions on random inputs: in our setting, the pattern is fixed while the text is randomly generated. Second, the instructions exhibit inherent correlations, depending on autocorrelations within the pattern or its prefixes, which makes accurate prediction more difficult.

The sliding window algorithm is given by Fig. 28, along with the implementation variant we study. Natural Sliding Window is the plain algorithm, where the pattern is checked letter by letter against the window at position i in the text, for each eligible position. The window loop condition is a dual check on the length of the pattern and a matching of letters. Usually, this pseudo-code is sufficient for analysis. However, for the purpose of our study, we are interested in the exact branching structure of the code. In particular, the condition $j < m$ `and` $X[j] = W[i + j]$ can be evaluated as a single test (as in Natural Sliding Window) or split into two consecutive tests by moving the check $X[j] = W[i+j]$ inside the loop. It is easy to see that, after that change, the `while` condition becomes redundant with the one at line 7, which leads to the reorganization in SlidingWindow, where one conditional instruction is eliminated. This matters in our model, as there will now be two distinct local predictors: one for $j < m$ and one for $X[j] = W[i + j]$.

Observe that when using a language like C, the compiler will almost surely translate `and` conditions into two jumps, since the second part should not be evaluated if the first part is false. This lazy behavior can be implemented with only one jump, using a `cmov` instruction, but unless explicitly forced, the (slightly optimized) compiled version will more likely resemble SlidingWindow, which is why we focus on this version of the algorithm from now on. Finally, we recall that, for consistency, we define a successful condition (i.e., when a test evaluates to true) as always leading to a taken branch.

5.1.1 Average number of comparisons in SlidingWindow

Let us begin with the average number of letter comparisons performed by the algorithm.All the results presented here are more or less folklore, we include them as an introductory step toward the analysis of mispredictions.

Assuming that the text is composed of letters chosen uniformly at random from an alphabet A, the sliding window algorithm runs in expected linear time, as stated in the following proposition which follows from linearity of expectation.

Lemma 37 ([CHL07])**.** *For any pattern, if the text W is chosen uniformly at random, then the average number of letter comparisons performed by the two algorithms is less than $\frac{|A|}{|A|-1}\,|W|$.*

The upper bound given by Lemma 37 is obtained by assuming that the nested `while`, line 4, halts only when two letters differ, i.e., by considering an infinite pattern X. This can be refined by taking into account the halting condition $j < m$ in the analysis, yielding that the expected number of letter comparisons is $\frac{|A|}{|A|-1}\,(n-m+1)\,(1-|A|^{-m})$. The upper bound from Lemma 37 remains sufficiently tight, except for small alphabets and very short patterns.

We refine our model with different probabilities for each letter: the text is generated by a so-called *memoryless source*, where each letter is drawn independently of the others according to a fixed, non-degenerate probability distribution $\pi : A \to (0,1)$. The expected number of comparisons performed by the algorithms is still linear for a text generated by a memoryless source, but it now depends directly on the pattern: a pattern composed mostly of frequent letters induces more comparisons in expectation than a pattern with rare letters only.

Lemma 38. *For any pattern X of length m, if the text is generated by a memoryless source of probability distribution π, then the average number of letter comparisons performed by the two algorithms, for a text of length n, is $\gamma(\pi,X)\,(n-m+1)$, with $\gamma(\pi,X) = \sum_{j=0}^{m-1}\pi(\mathrm{Pref}(X,j))$.*

Consider the pattern $X = aba$ over the alphabet $A = \{a,b,c\}$. For any starting index i in the text, there is always at least one comparison, i.e., with probability 1. Two comparisons occur when the first character matches, with probability $\pi(a)$, and three when the first two characters match, with probability $\pi(a)\pi(b)$. Since there cannot be more than three comparisons for a given index i, and there are $n-m+1$ possible values for i, linearity of expectation yields $\gamma(\pi,X) = 1+\pi(a)+\pi(a)\pi(b)$ expected comparisons.

A possible way to handle the pattern-dependent results is to choose the pattern randomly as well. This approach is standard in the analysis of text algorithms (see, for instance, the analysis of KMP [Rég89]). However, this is not the direction we pursue here, as our goal is to study how the pattern itself influences the results.

5.1.2 Average number of mispredictions in SlidingWindow

First, if we look at the `if` instruction at line 5 in SlidingWindow, the condition is tested at each increment of j: if the pattern is found at position i, this condition evaluates to false m times then true once; if not, it remains false throughout the loop (at least one iteration). As a result, the predictor quickly settles in the subset of states $\{\underline{\nu},\nu\}$ that predict the branch as not taken, and it produces a misprediction if and only if the pattern is found:

Lemma 39. *The average number of mispredictions caused by the counter update in* Sliding-Window *is asymptotically equivalent to $\pi(X)\,|W|$.*

Then, the analysis of the total number of mispredictions of SlidingWindow amounts to considering the condition of the `while` loop of line 8.

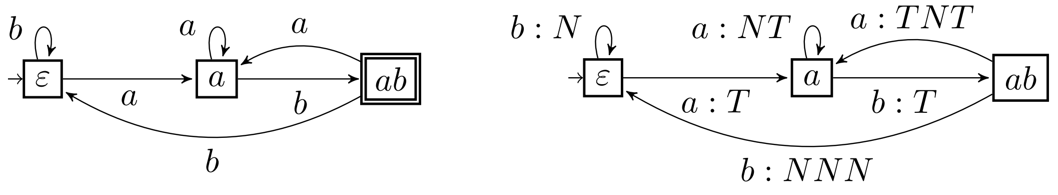


Figure 29: On the left, the finite-state automaton $\mathcal{A}_{aba}$ recognizing the language A^*ab; on the right, its associated transducer $\mathcal{T}_{aba}$, enriched to indicate whether the branch is taken or not for the condition $W[i+j] = X[j]$ in SlidingWindow.

Our main result here is an algorithm that computes the expected number of mispredictions for the condition $W[i+j] = X[j]$, given a fixed pattern $X = x_0 \cdots x_{m-1}$. The approach relies on the construction of two transducers $\mathcal{T}_X$ and $\mathcal{P}_X$: while reading a text W, the transducer $\mathcal{T}_X$ outputs the corresponding sequence of true/false evaluations of the condition (i.e., T/N, depending on whether the branch is taken or not), and $\mathcal{P}_X$ processes this output to compute the number of mispredictions.

If $w \in A^+$ is a non-empty word written as $w = w_0 \cdots w_{\ell-1}$, let $s(w) = w_1 \dots w_{\ell-1}$ denote the suffix of w obtained by removing its first letter, and $p(w) = w_0 \dots w_{\ell-2}$ denote the prefix of w obtained by removing its last letter. The first step of the algorithm consists in building the minimal finite-state automaton $\mathcal{A}_X$ that recognizes the language $A^* \cdot p(X)$. Let δ_X denote the transition function of $\mathcal{A}_X$, which is classically extended to words. The construction is standard [CHL07] and can be carried out in $\mathcal{O}(m)$ time, based on the following properties:

- The set of states Q_X consists of the m prefixes of $p(X)$; the initial state is ε, and the final state is $p(X)$.
- For any prefix w of $p(X)$ and any letter $\alpha \in A$:
  - if $w\alpha$ is a prefix of $p(X)$ then $\delta_X(w, \alpha) = w\alpha$, which is called a *forward transition*,
  - else it is a *backward transition* $\delta_X(w, \alpha) = \delta_X(\varepsilon, s(w)\alpha)$ if $w \neq \varepsilon$, and $\delta_X(\varepsilon, \alpha) = \varepsilon$.

For example, the automaton $\mathcal{A}_{aba}$ is depicted in Fig. 29, on the left.

When reading the path labeled by W in $\mathcal{A}_X$, after each letter of W is read in the algorithm, the current state q encodes the part of the sliding window which has already been parsed immediately before reading the next letter of W, i.e., the prefix of length j where j is the variable used in SlidingWindow (see Fig. 30 for an illustration):

- following a forward transition $w \xrightarrow{\alpha} w\alpha$ corresponds to a match $W[i+j] = X[j]$. Thus the `if` condition is true at line 8, hence j is incremented;
- otherwise the window is shifted to the right either due to a letter mismatch or because the pattern X has been found, and the algorithm continues until the next letter of W is queried. At that precise moment, the value of j is the length of the longest suffix of $w\alpha$ that belongs to Q_X, which by construction corresponds exactly to $\delta_X(w, \alpha)$.

The second step consists in transforming the automaton $\mathcal{A}_X$ into a transducer $\mathcal{T}_X$ that maps a word $w \in A^*$ to a binary word of $\{N, T\}^*$. Let $\gamma_X : Q_X \times A \to \{N, T\}^*$ be the output function of $\mathcal{T}_X$, which associates a binary word with each transition as follows:

1. If $w \xrightarrow{\alpha} w\alpha$ is a forward transition, then $\gamma_X(w, \alpha) = T$.
2. If $w \xrightarrow{\alpha} v$ is a backward transition and $w\alpha \notin \{\varepsilon, X\}$, then $\gamma_X(w, \alpha) = N \cdot \gamma_X(\varepsilon, s(w)\alpha)$.
3. If $w\alpha = X$ then $\gamma_X(w, \alpha) = T \cdot \gamma_X(\varepsilon, s(w)\alpha)$, and if $|X| > 1$ and $\alpha \neq x_0$, $\gamma_X(\varepsilon, \alpha) = N$.

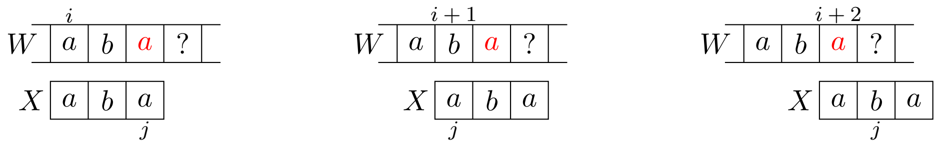


Figure 30: The link between the sliding window algorithm for $X = aba$ and the automaton $\mathcal{A}_X$ of Fig. 29 is illustrated through three successive steps. On the left, the prefix ab has already been matched and the red letter a at position $i + 2$ is queried for the first time in W. Since the pattern is found, the window shifts one position to the right, as shown in the center. There, an immediate mismatch occurs at the first position ($j = 0$), so the window advances once more. On the right, the first character matches, and the next letter of W needs to be queried to proceed. Before reading the red a, the window matched with ab, which corresponds to the state ab of the automaton. When the red a is read, the algorithm continues until it queries the unknown $W[i + 3]$, and at this point, it reaches the state a of $\mathcal{A}_X$. This part of the algorithm's execution corresponds to the transition $ab \xrightarrow{a} a$ in $\mathcal{A}_X$.

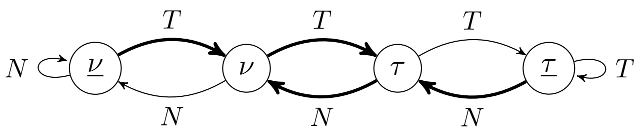
This construction can be computed in $\mathcal{O}(m^2)$, from left to right. It is designed so that each occurrence of T (resp. N) corresponds to the evaluation of $W[i+j] = X[j]$ as true (resp. false). Indeed, one can easily check that:

- If $w \xrightarrow{\alpha} w\alpha$ is a forward transition, the interpretation is that after having matched w in the text, we read α in W, which is also the next letter in X, so the condition is true.
- If $w \xrightarrow{\alpha} v$ is a backward transition and $w\alpha \neq X$, then we have a mismatch (producing a N) then we move the window one step to the right, aligning it with $s(w)$ in W.
- If $w\alpha = X$ then we have a match (producing a T) immediately followed by $j = m$ which implies moving the window as in the previous case.

If w is a word on A, let $\phi_X(w)$ denote the concatenation of the outputs produced when following the path labeled by w in $\mathcal{T}_X$. From the observation above, $\phi_X(W)$ is almost the sequence of evaluations of $W[i + j] = X[j]$ when calling SlidingWindow(X, W), the only difference being that we possibly do a few more steps in the transducer output because the algorithm halts as soon as $i > n - m$. This is formalized as follows.

Lemma 40. *Let w be the binary sequence of the evaluations of the condition $W[i+j] = X[j]$ in SlidingWindow(X, W). Then w is a prefix of $\phi_X(W)$ and $|\phi_X(W)| - |w| \leq |X|^2$.*

Recall that we use the 2-bit saturating counter below as our local branch predictor. Let ξ denote its transition function, extended to binary words. For example, $\xi(\nu, NNNTT) = \tau$. Additionally, let $\mu(\lambda, s)$ denote the number of mispredictions encountered when following the path in the predictor starting from state $\lambda \in \{\underline{\nu}, \nu, \tau, \underline{\tau}\}$ and labeled by $s \in \{N, T\}^*$. For instance, $\mu(\nu, NNNTT) = 2$.



We now build a kind of product of the transducer $\mathcal{T}_X$ and the predictor. This is a transducer $\mathcal{P}_X$, whose states are of the form (w, λ) where w is a state of $\mathcal{T}_X$ and $\lambda \in \{\underline{\nu}, \nu, \tau, \underline{\tau}\}$. Its transition function ζ_X is defined by $\zeta_X((w, \lambda), \alpha) = (\delta_X(w, \alpha), \xi(\lambda, \gamma_X(w, \alpha)))$. In other words, the first coordinates change along with the states in $\mathcal{T}_X$, and the second ones come from the update of the current state of the predictor according to the output of the transition in $\mathcal{T}_X$. From the example of Fig. 29, for instance $\zeta_{aba}((ab,\underline{\tau}),a) = (a,\underline{\tau})$, as the transition in $\mathcal{T}_{aba}$ is $ab \xrightarrow{a:TNT} a$ and $\xi(\underline{\tau}, TNT) = \underline{\tau}$.

The output function ψ_X of $\mathcal{P}_X$ accounts for the number of mispredictions performed during the updates. Formally, we have $\psi_X((w,\lambda),\alpha) = \mu(\lambda, \gamma_X(w,\alpha))$. To follow our example, as reading TNT from $\underline{\tau}$ produces one misprediction, we have $\psi_X((ab,\underline{\tau}),a) = 1$. Finally, we restrict $\mathcal{P}_X$ to the states accessible from $(\varepsilon,\underline{\nu})$: we fix the predictor's initial state to $\underline{\nu}$ and only retain states that can be reached in $\mathcal{P}_X$ during an execution of SlidingWindow. Choosing to start the predictor in state $\underline{\nu}$ is arbitrary, but does not change the asymptotic results (as Theorem 27 does not depend on the initial conditions). It simply makes $\mathcal{P}_X$ smaller, while still yielding the correct asymptotic estimates.

$$\begin{tikzcd}
& \boxed{a,\nu} \arrow[loop below, "a:1"] \arrow[r, "b:1"] & \boxed{ab,\tau} \arrow[dll, bend right, "b:1"'] \arrow[ddl, "a:1"] \\
\boxed{\varepsilon,\underline{\nu}} \arrow[loop left, "b:0"] \arrow[ur, "a:1"] & & \\
& \boxed{a,\underline{\tau}} \arrow[loop left, "a:1"] \arrow[r, bend left, "b:0"] & \boxed{ab,\underline{\tau}} \arrow[l, bend left, "a:1"] \arrow[ull, bend left, "b:2"]
\end{tikzcd}$$

$$\begin{array}{c|ccccc}
 & \varepsilon,\underline{\nu} & a,\nu & a,\underline{\tau} & ab,\tau & ab,\underline{\tau} \\
\hline
\varepsilon,\underline{\nu} & \pi(b) & \pi(a) & 0 & 0 & 0 \\
a,\nu & 0 & \pi(a) & 0 & \pi(b) & 0 \\
a,\underline{\tau} & 0 & 0 & \pi(a) & 0 & \pi(b) \\
ab,\tau & \pi(b) & 0 & \pi(a) & 0 & 0 \\
ab,\underline{\tau} & \pi(b) & 0 & \pi(a) & 0 & 0
\end{array}$$

Figure 31: On the left, the transducer $\mathcal{P}_{aba}$. Its states encode both the letters seen in the window and the state of the predictor just before each letter of W is read. The output is the number of mispredictions made by SlidingWindow when reading the letter. On the right, the transition matrix of the associated Markov chain, obtained directly from the probabilities of the letters labeling each transition.

The transducer $\mathcal{P}_{aba}$ is depicted in Fig. 31. Let $\xi_X(W)$ denote the sum of the outputs produced while following the path that starts from $(\varepsilon,\underline{\nu})$ and labeled by W in $\mathcal{P}_{aba}$. Then $\xi_X(W)$ gives a good estimation of the number of mispredictions produced on W by SlidingWindow, by Lemma 40 (and by Theorem 27, to ignore the predictor's starting state).

Proposition 41. *For a fixed pattern X, the expected number of mispredictions performed by* SlidingWindow *at line 8 for a random text W is asymptotically equivalent to $\xi_X(W)$.*

At this point, completing the computations is straightforward: $\xi_X(W)$ is obtained from a long trajectory in the Markov chain M_X obtained from $\mathcal{T}_X$ by changing the transitions $p \xrightarrow{\alpha:t} q$ into $p \xrightarrow{\pi(\alpha)} q$, or by making the sum of the probabilities if several letters label a transition between p and q. Computing the stationary distribution $\hat{\pi}_X$ of M_X gives the asymptotic number of times each state is visited with high probability. This, in turn, yields the asymptotic number of times each transition is taken: a transition $p \xrightarrow{\alpha:t} q$ is typically taken $\hat{\pi}_X(p)\cdot\pi(\alpha)\,n$ times, producing asymptotically $t\cdot\hat{\pi}_X(p)\cdot\pi(\alpha)\,n$ mispredictions. For instance, the stationary distribution $\hat{\pi} := \hat{\pi}_{aba}$ of the transition matrix of M_{aba} (Fig. 31) is:

$$\hat{\pi}(\varepsilon,\underline{\nu}) = (1-p)^2;\ \hat{\pi}(a,\nu) = (1-p)p;\ \hat{\pi}(a,\underline{\tau}) = p^2;\ \hat{\pi}(ab,\tau) = (1-p)^2p;\ \hat{\pi}(ab,\underline{\tau}) = (1-p)p^2,$$

where $p := \pi(a)$ is the probability of generating the letter a. From this, we obtain that the expected number of mispredictions is asymptotically $(p^4 - p^3 - p^2 + 3p)n$ for $A = \{a,b\}$.

The bottleneck in the running time of this construction lies in computing the stationary vector, which amounts to solving the equation $M_X^t \cdot \hat{\pi} = \hat{\pi}$. This can be done in $\mathcal{O}(|Q|^3)$ time

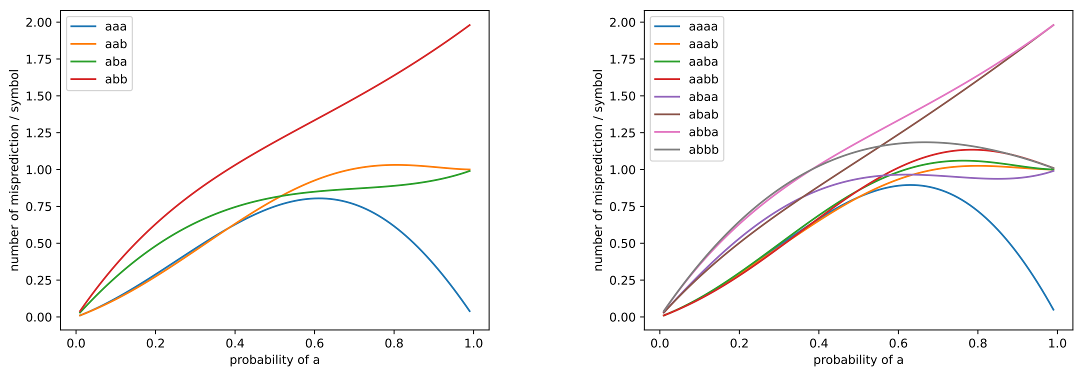


Figure 32: Variations of the expected number of mispredictions per symbol of W, depending on $\pi(a)$, for all pattern shapes of size 3 and 4 and $A = \{a, b\}$.

using Gauss elimination. Since $|Q| \leq 4m$, the following result summarizes what has been done so far.

Proposition 42. *There exists an algorithm of cost $O(|X|^3)$, which given a pattern X computes the asymptotic equivalent of the expected number of mispredictions $\xi_X(W)$ performed by line 8 of* SlidingWindow*(X, W), where W is produced by a memoryless source.*

Together with Lemma 39, this yields our main result for the sliding window algorithm:

Theorem 43. *Let X be a pattern of length $m \geq 1$. As n tends to infinity, the expected numbers of mispredictions produced by line 8 and line 5 in* SlidingWindow*(X, W) are asymptotically equivalent to $\mu_X n$ and $\pi(X)\,|W|$, respectively, when W is a text of size n generated by a memoryless source of probability π.*

The value of μ_X depends solely on the pattern and can be computed using the procedure described above. Here are some examples of the expression of μ_X for small patterns, assuming the alphabet is $\{a, b, c\}$, with $p := \pi(a)$, $q := \pi(b)$ and $r := \pi(c)$:

X	μ_X	X	μ_X
a	$\frac{p(1-p)}{1-2p(1-p)}$	aa	$p(1+2p-2p^2-3p^3+2p^4)$
ab	$\frac{p(1+2q-2p-4pq+p^2-3pq^2+2p^2q+3p^2q^2+p^2q^3)}{1-2p-pq+p^2+p^2q+p^2q^2}$	aaa	$\frac{p(1+3p-5p^2-p^3+4p^4-2p^5)}{1-p^2+p^3}$
aab	$\frac{p(1+3p-pq-3p^2+2p^2q-2p^3+p^3q-p^3q^2+2p^4-2p^4q)}{1-p^2+p^2q+p^3}$	abb	$\frac{p(1+2q+q^2-p-3pq-2pq^2-2pq^3+pq^4)}{1-p-pq}$
aba	$\frac{p((1-p)^2+2q-3pq+p^2q-2pq^2+p^2q^2(p+q+pq+p^2-1)}{(1-p)^2}$	abc	$\frac{p(1-p+2q+qr-3pq-2pq^2(1+r)-pq^2r^2)}{1-p-pq}$

For instance, the computations for the uniform distribution on $\{a, b\}$ gives $\mu_{aba} \approx 0.813$, $\mu_{abaa} \approx 0.938$ and $\mu_{abab} \approx 1.063$. The table below shows the numerical values of the ratio of mispredictions per letter of W for patterns of size up to 4 assuming a uniform distribution for W. Fig. 32 presents the same ratios for a non-uniform distribution. In both cases, the number of mispredictions clearly depends on the pattern. For some patterns, such as *abb* or *abab* for instance, the number of mispredictions even exceeds n, which is quite an underperformance. Indeed, since the expected number of letter comparisons is $2n$, the predictor performs worse than flipping a coin, suggesting that the local 2-bit saturating counter is misled by the dependencies arising in SlidingWindow.

X	$\lvert A\rvert=2$	$\lvert A\rvert=4$	$\lvert A\rvert=26$	X	$\lvert A\rvert=2$	$\lvert A\rvert=4$	$\lvert A\rvert=26$
aa	0.6265	0.3355	0.04278	$aaab$	0.8125	0.3883	0.04273
ab	0.9181	0.3910	0.04296	$aaba$	0.8594	0.4011	0.04279
aaa	0.7501	0.3766	0.04273	$aabb$	0.8594	0.4011	0.04279
aab	0.7970	0.3894	0.04279	$abaa$	0.9375	0.3951	0.04153
aba	0.8126	0.3814	0.04153	$abab$	1.0625	0.3971	0.04153
abb	1.1876	0.3946	0.04154	$abba$	1.1875	0.3945	0.04148
$aaaa$	0.8125	0.3883	0.04273	$abbb$	1.1250	0.3906	0.04148

Since this is the algorithm used for `Strings` in `Java`, it would be interesting to further investigate whether it actually performs poorly in practice, especially given that modern branch predictors are likely far more sophisticated. However, as much more efficient pattern matching algorithms exist in the literature, we will first turn our attention to these before exploring the possible impact of global branch prediction.

5.2 Morris-Pratt and Knuth-Morris-Pratt algorithms

We proceed with the study of less naive pattern matching algorithms, and we start with the classical Morris-Pratt (MP) and Knuth-Morris-Pratt (KMP) algorithms [MP70, KJP77].[13] Once again, we focus on quantifying mispredictions for random text inputs for the 2-bit saturating counter of Fig. 17 (page 40).

5.2.1 Algorithms and their encoding using transducers

Unlike the naive algorithm discussed in the previous section, MP and KMP algorithms both rely on precomputing a *failure function*, which helps identify candidate positions for the pattern X within the text W. The general strategy consists in scanning W from left to right, one character at a time. Before advancing to the next character in W, the algorithm determines the longest prefix of X that is also a suffix of the discovered prefix of W. The failure function is the key component that allows this step to be performed efficiently.

The function mp_X maps each prefix of X to its longest strict border, with the convention that $\mathrm{mp}_X(\varepsilon) = \bot$. The function kmp_X is a refinement of the function mp_X, defined by $\mathrm{kmp}_X(X) = \mathrm{mp}_X(X)$, $\mathrm{kmp}_X(\varepsilon) = \bot$, and for all prefixes $u\alpha$ of X, where $u \in A^*$ and $\alpha \in A$, $\mathrm{kmp}_X(u)$ is the longest strict suffix of u that is also a prefix of u but such that $\mathrm{kmp}_X(u)\alpha$ is not. If no such strict suffix exists, then $\mathrm{kmp}_X(u) = \bot$. For a more detailed discussion of these failure functions, see [Gus97]. In the following, we only require that Algorithm Find remains correct regardless of which failure function is used. Within the algorithm, the function $b := \mathrm{mp}_X$ (or $b := \mathrm{kmp}_X$) is transformed into a precomputed integer-valued array B, defined as $B[i] = |b(\mathrm{Pref}(X, i))|$ for $i \in \{0, \ldots, |X|\}$, with the convention that $|\bot| = -1$.

Algorithm Find utilizes the precomputed table B from either mp_X or kmp_X to efficiently locate potential occurrences of X in W. The indices i and j are the current positions in X and W, respectively. Each iteration of the main `while` loop (line 2) corresponds to the discovery of exactly one letter in W. At the beginning of each iteration, index i stores the length of the longest matching prefix of X. The inner `while` loop (line 3) updates i using the table B. Finally, the `if` statement (line 6) is executed when an occurrence of X is found, updating i

[13]We could also look at Boyer-Moore [CHL07] since they both have interpretations in terms of automata that we should be able to exploit in a similar way

```
Find
Input: pattern $X$ of size $m$, text $W$ of size $n$, border table $B$
1 $i, j, nb \leftarrow 0, 0, 0$
2 while $j < n$ do
3     while $i \geq 0$ and $X[i] \neq W[j]$ do
4         $i \leftarrow B[i]$
5     $i, j \leftarrow i + 1, j + 1$
6     if $i = m$ then
7         $i, nb \leftarrow B[i], nb + 1$
8 return $nb$
```

accordingly. For both algorithms, the table B can be computed in $\mathcal{O}(m)$ time and Find runs in $\mathcal{O}(n)$ time. In the worst case, it performs at most $2n - m$ character comparisons [CR94].

As already mentioned, in any programming language supporting short-circuit evaluation of boolean operators, the condition $i \geq 0$ `and` $X[i] \neq W[j]$ at line 3 of Algorithm Find is evaluated as two separate jumps by the compiler. As a result, Algorithm Find contains a total of four branches: one at line 2, two at line 3, and one at line 6. In our model, each of these four branches is assigned a local predictor, and all may potentially lead to mispredictions. Recall finally that our convention is that a successful condition (when the test evaluates to true) always leads to a taken branch.

Associated automata At the beginning of each iteration of the main `while` loop in Algorithm Find, the prefix of W of length j (from $W[0]$ to $W[j-1]$) has been discovered, and i is the length of the longest suffix of $\text{Pref}(W, j)$ that is also a strict prefix of X: we cannot have $i = m$, because finding the pattern immediately triggers the update of i to $B[i]$ in line 6.

The evolution of i at each iteration of the main `while` loop is encoded by a deterministic and complete automaton $\mathcal{A}_X$. Its set of states is the set Q_X of strict prefixes of X, identified by their unique lengths if necessary. Its transition function δ_X maps a pair (u, α) to the longest suffix of $u\alpha$ which is in Q_X. Its initial state is ε. If $X = Y\alpha$, where $\alpha \in A$ is a letter, then when following the path labeled by W starting from the initial state of $\mathcal{A}_X$, there is an occurrence of X in W exactly when the transition $Y \xrightarrow{\alpha} \delta_X(Y, \alpha)$ is used. This variant of the classical construction is more relevant here than the usual one [CR94, Sec. 7.1] which also has the state X. The transition $Y \xrightarrow{\alpha} \delta_X(Y, \alpha)$ is the accepting transition to identify occurrences of X. The automaton $\mathcal{A}_X$ tracks the value of i at the beginning of each iteration of the main loop, where i corresponds to the length of the current state label. This value remains the same for both MP and KMP. An example of $\mathcal{A}_X$ is depicted in Fig. 33.

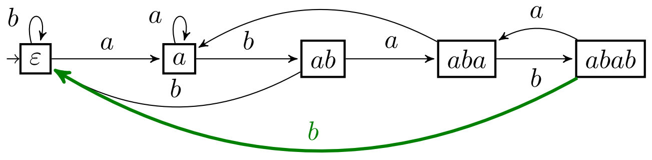


Figure 33: The deterministic and complete automaton $\mathcal{A}_X$ for $X = ababb$.

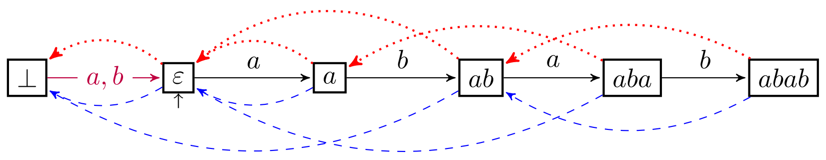


Figure 34: The automata $\mathcal{F}_X^{\mathrm{mp}}$ and $\mathcal{F}_X^{\mathrm{kmp}}$ for $X = ababb$; the failure transitions of $\mathcal{F}_X^{\mathrm{mp}}$ are shown as dotted red lines above, while those of $\mathcal{F}_X^{\mathrm{kmp}}$ appear as dashed blue lines below. To read the letter a from state aba in $\mathcal{F}_X^{\mathrm{mp}}$, we follow the failure transition $aba \to a$, then $a \to \varepsilon$ until finally reaching the transition $\varepsilon \xrightarrow{a} a$ where the letter a can be read. In $\mathcal{F}_X^{\mathrm{kmp}}$, only one failure transition $aba \to \varepsilon$ is needed, instead of two.

To refine the simulation of Algorithm Find using automata, we incorporate the failure functions. This is achieved by constructing the *failure automaton*. Specifically, for Algorithm MP, let $\mathcal{F}_X^{\mathrm{mp}}$ be the automaton defined by:

- A state set $Q_X \cup \{\perp\}$ and an initial state ε.
- Transitions $\perp \xrightarrow{\alpha} \varepsilon$ for every $\alpha \in A$.
- Transitions $u \xrightarrow{\alpha} u\alpha$ for every $u \in Q_X$ such that $u\alpha \in Q_X$.
- A failure transition $u \to \mathrm{mp}_X(u)$ for every $u \in Q_X$, used when attempting to read a letter α where $u\alpha \notin Q_X$.

The automaton $\mathcal{F}_X^{\mathrm{kmp}}$ associated with KMP is identical to $\mathcal{F}_X^{\mathrm{mp}}$, except that its failure transitions are defined as $u \to \mathrm{kmp}_X(u)$ for every $u \in Q_X$. Both automata serve as graphical representations of the failure functions mp_X and kmp_X, structured in a way that aligns with $\mathcal{A}_X$. An example of $\mathcal{F}_X^{\mathrm{mp}}$ and $\mathcal{F}_X^{\mathrm{kmp}}$ is illustrated in Fig. 34.

When reading a letter α from a state u in $\mathcal{F}_X^{\mathrm{mp}}$ or $\mathcal{F}_X^{\mathrm{kmp}}$, if the transition $u \xrightarrow{\alpha} u\alpha$ does not exist, the automaton follows failure transitions until a state with an outgoing transition labeled by α is found. This process corresponds to a single backward transition in $\mathcal{A}_X$. Importantly, using a failure transition directly mirrors the execution of the nested `while` loop in Algorithm Find (line 3) or triggers the `if` statement at line 6 when an occurrence of X is found. This construction captures what we need for the forthcoming analysis.

5.2.2 Expected number of letter comparisons for a given pattern

We begin by analyzing the expected number of letter comparisons made by Algorithm Find for a given pattern X of length m and a random text W of length n. The average-case complexity of classical pattern matching algorithms has been explored before, particularly in scenarios where both the pattern and the text are randomly generated. Early studies [Rég89, RS98] examined the expected number of comparisons in Algorithms MP and KMP under memoryless or Markovian source models, employing techniques from analytic combinatorics. Prior attempts based on Markov chains introduced substantial approximations, limiting their accuracy compared to more refined combinatorial methods [Rég89, RS98]. Here, we refine and extend this Markov chain-based methodology, providing a more precise foundation for analyzing the expected number of mispredictions (see Section 5.2.3).

Recall that π is a probability measure on A such that for all $\alpha \in A$, $0 < \pi(\alpha) < 1$, and for each $n \geq 0$ and each $W \in A^n$, $\pi_n(W) := \prod_{i=0}^{n-1} \pi(W_i)$.

Encoding the letter comparisons with transducers Letter comparisons occur at line 3 only if $i \geq 0$, due to the lazy evaluation of the `and` operator (when $i < 0$, $W[j]$ is not compared

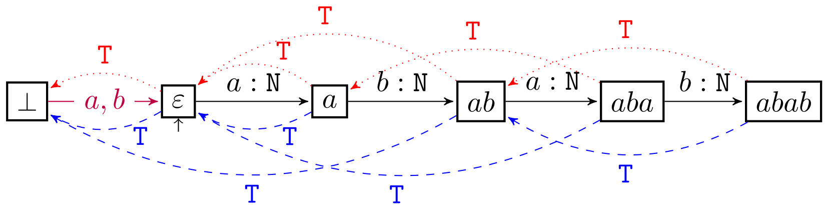


Figure 35: The automata $\mathcal{F}_X^{\text{mp}}$ and $\mathcal{F}_X^{\text{kmp}}$ transformed into transducers by adding the result of letter comparisons in Find as output of each transition.

to $X[i]$). We encode these comparisons within the automata of Section 5.2.1 by adding outputs to the transitions, thereby transforming them into transducers. In both $\mathcal{F}_X^{\text{mp}}$ and $\mathcal{F}_X^{\text{kmp}}$, each transition $u \xrightarrow{\alpha} u\alpha$ corresponds to matching letters, meaning the test $X[i] \neq W[j]$ evaluates to false. As before, we denote this with the letter N for a *not taken* branch. Conversely, following a failure transition indicates that the test $X[i] \neq W[j]$ is true, which we denote by T for *taken*. Transitions from $\perp$ correspond to cases where $i \geq 0$ is false, meaning no letter comparisons occur, as noted above. This construction is illustrated in Fig. 35.

We keep track of the results of the comparisons $X[i] \neq W[j]$ in $\mathcal{A}_X$ by simulating the reading of each letter in the transducer associated with $\mathcal{F}_X^{\text{mp}}$ and concatenating the outputs. This transforms $\mathcal{A}_X$ into the transducer $\mathcal{T}_X^{\text{mp}}$ for MP, by adding an output function $\nabla_{\mathcal{T}_X^{\text{mp}}}$ to $\mathcal{A}_X$ as follows (see Fig. 36 for an example).

$$\nabla_{\mathcal{T}_X^{\text{mp}}}\left(u \xrightarrow{\alpha}\right) = \begin{cases} N & \text{if } u\alpha \in Q_X \text{ or } u\alpha = X, \\ T & \text{if } u\alpha \notin Q_X \text{ and } \text{mp}(u) = \perp, \\ T \cdot \nabla_{\mathcal{T}_X^{\text{mp}}}\left(\text{mp}(u) \xrightarrow{\alpha}\right) & \text{otherwise.} \end{cases} \tag{9}$$

Instead of $\nabla_{\mathcal{T}_X^{\text{mp}}}$ we can use $\nabla_{\mathcal{T}_X^{\text{kmp}}}$ defined as $\nabla_{\mathcal{T}_X^{\text{mp}}}$ except that mp is changed into kmp. This yields the transducer $\mathcal{T}_X^{\text{kmp}}$ associated with KMP.

Recall that the output of a path in a transducer is the concatenation of the outputs of its transitions. As the transducers $\mathcal{T}_X^{\text{mp}}$ and $\mathcal{T}_X^{\text{kmp}}$ are (input-)deterministic and complete, the output of a word is the output of its unique path that starts at the initial state. From the classical link between $\mathcal{A}_X$ and Algorithm Find [CR94] we have the following key statement.

Lemma 44. *The sequence of results of the comparisons $X[i] \neq W[j]$ when applying Algorithm* Find *to the pattern X and text W is equal to the output of the word W in the transducer $\mathcal{T}_X^{mp}$ for* MP*, and in the transducer $\mathcal{T}_X^{kmp}$ for* KMP *.*

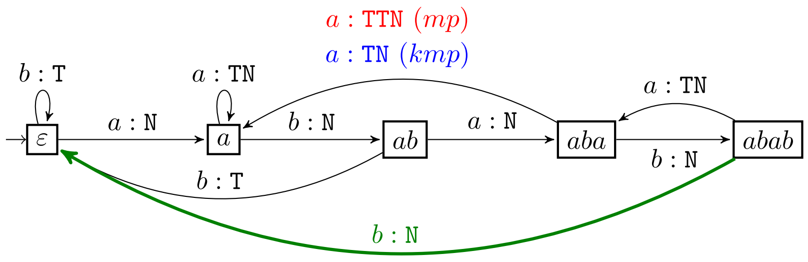


Figure 36: The transducers $\mathcal{T}_X^{\text{mp}}$ and $\mathcal{T}_X^{\text{kmp}}$ for $X = ababb$. The only difference between them lies in the transition $aba \xrightarrow{a} a$, for which MP uses one more letter comparison.

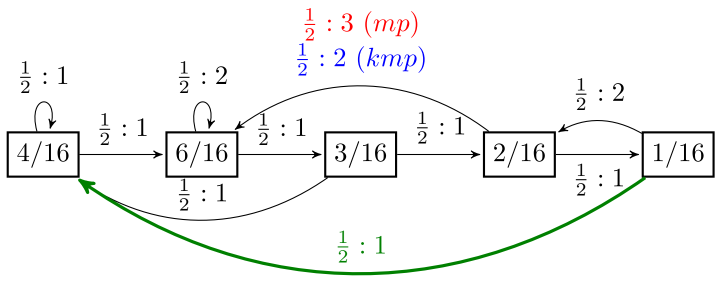


Figure 37: A graphical representation for the computation of the expected number of comparisons for the uniform distribution on $\{a, b\}$: in $\mathcal{T}_X^{\text{mp}}$ and $\mathcal{T}_X^{\text{kmp}}$ the state labels u have been replaced by their probabilities $p_X(u)$, the letters by their probabilities $1/2$, and the output by their lengths. For instance, the transition $aba \xrightarrow{a} a$ has probability $\frac{2}{16} \cdot \frac{1}{2} = \frac{1}{16}$ yielding 3 comparisons for MP or 2 for KMP. Thus, its contribution to C_X in Proposition 46 is $\frac{3}{16}$ or $\frac{1}{8}$.

State probability and expected number of comparisons Since we can use exactly the same techniques, from now on we focus on KMP in our presentation. Recall that if we reach the state u after reading the first j letters of W in $\mathcal{A}_X$, and hence in $\mathcal{T}_X^{\text{kmp}}$, then at the next iteration of the main `while` loop, index i contains the value $|u|$. For $u \in Q_X$ and $j \in \{0, \ldots, n-1\}$, we are thus interested in the probability $p_X(j, u)$ that after reading $\text{Pref}(W, j)$ in $\mathcal{T}_X^{\text{kmp}}$ we end in a state u. Slightly abusing notation, we write $\pi(u) = \pi_{|u|}(u)$. For any $u \in Q_X$ let $\text{bord}(u)$ denote the longest strict border of u, with the convention that $\text{bord}(\varepsilon) = \perp$.

Lemma 45. *For any $u \in Q_X$ and any $j \geq m$, $p_X(j, u)$ does not depend on j and we have $p_X(j, u) = p_X(u)$ with $p_X(u) := \pi(u) - \sum_{\substack{v \in Q_X \\ bord(v) = u}} \pi(v)$.*

From Lemma 45 we can easily estimate the expected number of comparisons for any fixed pattern X, when the length n of W tends to infinity. Indeed, except when $j < m$, the probability $p_X(j, u)$ does not depend on j. Moreover, if we are in state u, from the length of the outputs of $\mathcal{T}_X^{\text{kmp}}$ we can directly compute the expected number of comparisons during the next iteration of the main `while` loop. See Fig. 37 for a graphical representation.

Proposition 46. *As $n \to \infty$, the expected number of letter comparisons performed by Algorithm* Find *with* KMP *(or* MP *with $\mathcal{T}_X^{mp}$) is asymptotically equivalent to $C_X \cdot n$, where*

$$C_X = \sum_{u \in Q_X} p_X(u) \sum_{a \in A} \pi(a) \cdot \left| \nabla_{\mathcal{T}_X^{kmp}} \left(u \xrightarrow{a} \right) \right|, \text{ and } 1 \leq C_X \leq 2.$$

Observe that Lemma 45 can also be derived by transforming $\mathcal{T}_X^{\text{kmp}}$ into a Markov chain and computing its stationary distribution [LPW08]. However, Lemma 45 provides a more direct and simpler formula, which appears to have gone unnoticed in the literature. Markov chains will also prove very useful in Section 5.2.3.

5.2.3 Expected number of mispredictions

We now turn to our main objective: a theoretical analysis of the number of mispredictions for a fixed pattern X of length m and a random text W of length n.

Recall that we use the following local branch predictor and that ξ denote its transition function extended to binary words. As before, $\mu(\lambda, s)$ is the number of mispredictions encountered when following the path in the predictor starting from state $\lambda \in \{\underline{\nu}, \nu, \tau, \underline{\tau}\}$ and labeled by $s \in \{N, T\}^*$.

$$\begin{tikzcd}
\underline{\nu} \arrow[loop left, "N"] \arrow[r, bend left, "T"] & \nu \arrow[l, bend left, "N"] \arrow[r, bend left, "T"] & \tau \arrow[l, bend left, "N"] \arrow[r, bend left, "T"] & \underline{\tau} \arrow[l, bend left, "N"] \arrow[loop right, "T"]
\end{tikzcd}$$

As previously noted, Algorithm Find contains four branches in total: one at line 2, two at line 3, and one at line 6. Each of these branches is assigned a local predictor, and all have the potential to generate mispredictions. The mispredictions generated by the main `while` loop (i.e., line 2) are easily analyzed. Indeed, the test holds true for n times and then becomes false. Hence, the sequence of taken/not taken outcomes for this branch is T^nN. Therefore, starting from any state of the 2-bit saturating predictor, at most three mispredictions can occur. It is asymptotically negligible, as we will demonstrate that the other branches produce a linear number of mispredictions on average.

Mispredictions of the counter update We analyze the expected number of mispredictions induced by the counter update at line 6. The sequence s of taken/not-taken outcomes for this `if` statement is defined by $s_j = T$ if and only if $\mathrm{Pref}(W, j)$ ends with the pattern X, for all $j \in \{0, \dots, n-1\}$. This is easy to analyze, especially when the pattern X is not the repetition of a single letter. Proposition 47 establishes that, on average, there is approximately one misprediction for each occurrence of the pattern in the text.

Proposition 47. *If X contains at least two distinct letters, then the expected number of mispredictions caused by the counter update is asymptotically equivalent to $\pi(X) \cdot n$.*

Proof (sketch). Since X contains at least two distinct letters, it cannot be a suffix of both $\mathrm{Pref}(W, j)$ and $\mathrm{Pref}(W, j + 1)$. Hence, the sequence s is of the form $(N^+T)^*N^*$. This means that every step to the right in the local predictor (for every T in sequence s), which corresponds to a match, is followed by a step to the left, except possibly for the last step. Thus, if the local predictor reaches state $\underline{\nu}$, it remains in ν or $\underline{\nu}$ forever. Having three consecutive positions in W without an occurrence of X is sufficient to reach state $\underline{\nu}$. This happens in fewer than $\mathcal{O}(\log n)$ iterations with high probability, and at this point there is exactly one misprediction each time the pattern is found. This concludes the proof, as the expected number of occurrences of X in W is asymptotically equivalent to $\pi(X) \cdot n$. □

The analysis of the case $X = \alpha^m$, where X consists of a repeated single letter, is more intricate. We present the proof sketch for $X = \alpha\alpha$, which captures all the essential ideas. Let $A' = A \setminus \{\alpha\}$ and write $W = \beta_1\beta_2 \dots \beta_\ell\alpha^x$, where $\beta_i = \alpha^{k_i}\overline{\alpha}$ with $k_i \geq 0$ and $\overline{\alpha} \in A'$. Depending on the value of k_i, one can compute the sequence of taken/not taken outcomes induced by a factor $\alpha^{k_i}\overline{\alpha}$, which is either preceded by a letter $\overline{\alpha}$ or nothing: $\overline{\alpha}$ yields N, $\alpha\overline{\alpha}$ yields NN, $\alpha^2\overline{\alpha}$ yields NTN, and so on. Thus, more generally, $\overline{\alpha}$ yields N and $\alpha^{k_i}\overline{\alpha}$ yields $NT^{k_i-1}N$ for $k_i \geq 1$. We then examine the state of the predictor and the number of mispredictions produced after each factor β_i is read. For instance, if just before reading $\beta_i = \alpha^3\overline{\alpha}$ the predictor state is ν, then the associated sequence $NTTN$ produces three mispredictions and the predictor ends in the same state ν, which can be seen on the path $\nu \xrightarrow{N} \underline{\nu} \xrightarrow[\textit{misp.}]{T} \nu \xrightarrow[\textit{misp.}]{T} \tau \xrightarrow[\textit{misp.}]{N} \nu$. Since $\underline{\tau}$ cannot be reached except at the very beginning or at the very end, it has a negligible

$$\begin{tikzcd}
\boxed{\tilde{\nu}} \arrow[loop left, "k \leq 3"] \arrow[r, bend left, "k \geq 4"] & \boxed{\tau} \arrow[loop right, "k \geq 3"] \arrow[l, bend left, "k \leq 2"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\boxed{\tilde{\nu}} \arrow[loop left, "1-p^4"] \arrow[r, bend left, "p^4"] & \boxed{\tau} \arrow[loop right, "p^3"] \arrow[l, bend left, "1-p^3"]
\end{tikzcd}$$

Figure 38: On the left, the transition system determined by the factor $\alpha^k\overline{\alpha}$; on the right, the corresponding Markov chain. For clarity, we denote $p := \pi(\alpha)$.

contribution to the expectation, and we can list all the relevant possibilities as follows:

	$k=0$		$k=1$		$k=2$		$k=3$		$k \geq 4$	
	N		NN		NTN		$NTTN$		$NT^{k-1}N$	
$\underline{\nu}$	$\to \underline{\nu}$	0 misp.	$\to \underline{\nu}$	0 misp.	$\to \underline{\nu}$	1 misp.	$\to \nu$	3 misp.	$\to \tau$	3 misp.
ν	$\to \underline{\nu}$	0 misp.	$\to \underline{\nu}$	0 misp.	$\to \underline{\nu}$	1 misp.	$\to \nu$	3 misp.	$\to \tau$	3 misp.
τ	$\to \nu$	1 misp.	$\to \underline{\nu}$	1 misp.	$\to \nu$	3 misp.	$\to \tau$	3 misp.	$\to \tau$	3 misp.

In the table above, the states $\underline{\nu}$ and ν produce identical outcomes and can therefore be merged into a single state, denoted as $\tilde{\nu}$, for the analysis. The resulting transitions form a graph with two vertices, which is then converted into a Markov chain by incorporating the transition probabilities $\alpha^k\overline{\alpha}$, as illustrated in Fig. 38.

The stationary distribution $\hat{\pi}$ of this Markov chain is straightforward to compute, yielding $\hat{\pi}(\tilde{\nu}) = \frac{1-p^3}{1-p^3+p^4}$ and $\hat{\pi}(\tau) = \frac{p^4}{1-p^3+p^4}$, where $p := \pi(\alpha)$. From each state, the expected number of mispredictions can be computed using the transition table. For instance, starting from $\tilde{\nu}$, a misprediction occurs when $k = 2$ with probability $(1-p)p^2$, and three mispredictions occur when $k \geq 3$ with probability p^3. Therefore, the expected number of mispredictions when reading the next factor $\alpha^k\overline{\alpha}$ from $\tilde{\nu}$ is given by $(1-p)p^2 + 3p^3$. Finally, with high probability there are around $(1-p)n$ factors of the form $\alpha^*\overline{\alpha}$ in the decomposition of W, which corresponds to roughly the same number of steps in the Markov chain. The general statement for $X = \alpha^m$ is as follows.

Proposition 48. *If $X = \alpha^m$, the expected number of mispredictions caused by the counter update is asymptotically $\kappa_m(\pi(\alpha)) \cdot n$, with $\kappa_m(p) = p^m(1-p)(1+p)^2$ for $m \geq 3$, and*

$$\kappa_1(p) = \frac{p(1-p)}{1-2p(1-p)}, \qquad \textit{and} \qquad \kappa_2(p) = \frac{p^2(1-p)\left(1+2p+p^2-p^3\right)}{1-p^3+p^4}.$$

Expected number of mispredictions during letter comparisons In this section, we analyze the expected number of mispredictions caused by letter comparisons in KMP (similar results can be derived for MP).

According to Lemma 44, the outcome of letter comparisons in KMP is encoded by the transducer $\mathcal{T}_X^{\text{kmp}}$. More precisely, following a transition $u \xrightarrow{\alpha:s} v$ in this transducer simulates a single iteration of the main loop of Algorithm Find, starting with $i = |u|$ and processing the letter $\alpha := W[j]$. At the end of this iteration, $i = |v|$, and $s \in \{N, T\}^*$ is the sequence of taken/not-taken outcomes for the test $X[i] \neq W[j]$.

The mispredictions occurring during this single iteration of the main loop depend on the predictor's initial state λ and the sequence s which is computed using $\mathcal{T}_X^{\text{kmp}}$. The number of mispredictions $\mu(\lambda, s)$ is retrieved by following the path starting from state λ and labeled by s in the predictor, corresponding to the transition $\xi(\lambda, s)$. This is formalized by using a coupling of $\mathcal{T}_X^{\text{kmp}}$ with the predictor in Fig. 17, forming a product transducer $\mathcal{P}_X^{\text{kmp}}$, defined as follows (see Fig. 39 for an example):

- the set of states is $Q_X \times \{\underline{\nu}, \nu, \tau, \underline{\tau}\}$,
- there is a transition $(u, \lambda) \xrightarrow{\alpha:\mu(\lambda,s)} (\delta_X(u), \xi(\lambda, s))$ for every state (u, λ) and every letter α, where s is the output of the transition $u \xrightarrow{\alpha:s} \delta_X(u)$ in $\mathcal{T}_X^{\text{kmp}}$.

By construction, at the beginning of an iteration of the main loop in Algorithm Find, if $i = |u|$, λ is the initial state of the 2-bit saturating predictor, and $\alpha = W[j]$, then, during the next iteration, $\mu(\lambda, s)$ mispredictions occur, and the predictor terminates in state $\xi(\lambda, s)$, where $u \xrightarrow{\alpha:s} \delta_X(u)$ in $\mathcal{T}_X^{\text{kmp}}$. This leads to the following statement.

Lemma 49. *The number of mispredictions caused by letter comparisons in* KMP*, when applied to the text W and the pattern X, is given by the sum of the outputs along the path that starts at (ε, λ_0) and is labeled by W in $\mathcal{P}_X^{kmp}$, where $\lambda_0 \in \{\underline{\nu}, \nu, \tau, \underline{\tau}\}$ is the initial state of the local predictor associated with the letter comparison.*

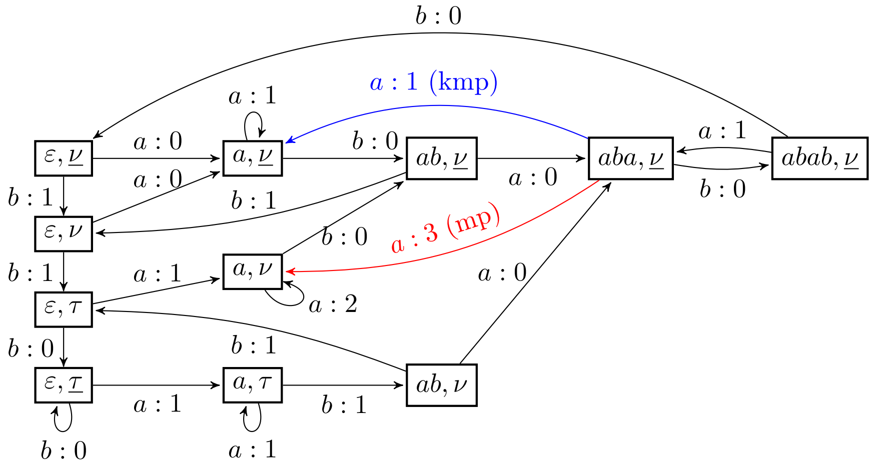


Figure 39: The strongly connected terminal component of $\mathcal{P}_X^{\text{kmp}}$ in black and blue, for $X = ababb$. In black and red, the variant for $\mathcal{P}_X^{\text{mp}}$.

We can then proceed as in Proposition 48: the transducer $\mathcal{P}_X^{\text{kmp}}$ is converted into a Markov chain by assigning a weight of $\pi(\alpha)$ to the transitions labeled by a letter α. From this, we compute the stationary distribution $\hat{\pi}$ over the set of states, allowing us to determine the asymptotic expected number of mispredictions per letter of W. This quantity, L_X, satisfies

$$L_X = \sum_{u \in Q_X} \sum_{\lambda \in \{\underline{\nu}, \nu, \tau, \underline{\tau}\}} \hat{\pi}(u, \lambda) \times \sum_{\alpha \in A} \pi(\alpha) \cdot \nabla_{\mathcal{P}_X^{\text{kmp}}}((u, \lambda) \xrightarrow{\alpha}). \tag{10}$$

Observe that when processing a long sequence of letters different from $X[0]$, the letter comparisons produce a sequence of T's, causing the 2-bit saturating predictor to settle in state $\underline{\tau}$ while $i = 0$ in the algorithm. Consequently, the state $(\varepsilon, \underline{\tau})$ is reachable from every other state. Hence, the Markov chain has a unique terminal strongly connected component (i.e., there are no transitions from any vertex in this strongly connected component to any vertex outside of it), which includes $(\varepsilon, \underline{\tau})$ along with a self-loop at this state. Thus, our analysis focuses on this component, allowing us to apply classical results on primitive Markov chains [LPW08], ultimately leading to Eq. (10). Notably, this result is independent of the predictor's initial state. The computation of L_X can be easily carried out using computer algebra, since computing the stationary probability reduces to inverting a matrix.

Proposition 50. *The expected number of mispredictions caused by letter comparisons in* KMP *on a random text of length n and a pattern X, is asymptotically equivalent to $L_X \cdot n$.*

Expected number of mispredictions of the test $i \geq 0$ We conclude the analysis by examining the mispredictions caused by the test $i \geq 0$ at line 3 of Algorithm Find. To this end, we use the previously constructed transducer $\mathcal{T}_X^{\text{kmp}}$ (or equivalently $\mathcal{T}_X^{\text{mp}}$, as the approach remains the same) to capture the behavior of this test through a straightforward transformation of the outputs. Recall that a transition $u \xrightarrow{\alpha:s} v$ in $\mathcal{T}_X^{\text{kmp}}$, with $s \in \{N, T\}^*$ indicates that when reading the letter α, the inner `while` loop performs $|s|$ character comparisons, with the result encoded by the symbols of s. Due to the loop structure, s always takes one of two forms:

- T^*N and the loop terminates because $X[i] = W[j]$ eventually, or
- T^+ and the loop terminates because $i = -1$ eventually.

In the first case, the condition $i \geq 0$ holds for $|s|$ iterations. In the second case, it also holds for $|s|$ iterations, before failing once. Thus, we define the transducer $\widetilde{\mathcal{T}}_X^{\text{kmp}}$ identically to $\mathcal{T}_X^{\text{kmp}}$, except for its output function:

$$\nabla_{\widetilde{\mathcal{T}}_X^{\text{kmp}}}\left(u \xrightarrow{\alpha}\right) = \begin{cases} T^{|s|} & \text{if } s = \nabla_{\mathcal{T}_X^{\text{kmp}}}\left(u \xrightarrow{\alpha}\right) \in T^*N, \\ T^{|s|}N & \text{if } s = \nabla_{\mathcal{T}_X^{\text{kmp}}}\left(u \xrightarrow{\alpha}\right) \in T^+. \end{cases} \tag{11}$$

The same transformation can be applied to $\mathcal{T}_X^{\text{mp}}$ for MP. At this stage, we could directly reuse the framework from Section 5.2.3 to compute the asymptotic expected number of mispredictions for any given pattern X. However, a shortcut allows for a simpler formulation while offering deeper insight into the mispredictions caused by the test $i \geq 0$.

Since each output is either T^kN for some $k \geq 1$ or T^k, the local predictor state generally moves toward $\underline{\tau}$, except in the case of TN. In this latter case, the predictor either remains in the same state or transitions from $\underline{\tau}$ to τ. Moreover, from any state s of $\mathcal{A}_X$, there always exists a letter α such that $s \xrightarrow{\alpha:T}$ in $\widetilde{\mathcal{T}}_X^{\text{mp}}$ or $\widetilde{\mathcal{T}}_X^{\text{kmp}}$ (for instance, the transition that goes to the right or when the pattern is found). As a result, with high probability, the predictor reaches the state $\underline{\tau}$ in at most $\mathcal{O}(\log n)$ iterations of the main loop of Algorithm Find. Once in $\underline{\tau}$, the predictor remains confined to the states τ and $\underline{\tau}$ indefinitely. Thus, with high probability, except for a small number of initial steps, the predictor consistently predicts that the branch is taken. At this point, a misprediction occurs if and only if the output belongs to T^*N, which happens precisely when a non-accepting transition in $\widetilde{\mathcal{T}}_X^{\text{kmp}}$ leads to the state ε. Since $\widetilde{\mathcal{T}}_X^{\text{kmp}}$ and $\widetilde{\mathcal{T}}_X^{\text{mp}}$ differ only in their output functions, this result holds for both MP and KMP, allowing us to work directly with $\mathcal{A}_X$. Applying Lemma 45, we obtain the following statement.

Proposition 51. *When Algorithms* MP *or* KMP *are applied to a random text W of length n with a given pattern X, the expected number of mispredictions caused by the test $i \geq 0$ is equal to the expected number of times a transition ending in ε is taken along the path labeled by W in $\mathcal{A}_X$, up to an error term of $\mathcal{O}(\log n)$. As a result, the expected number of such mispredictions is asymptotically equivalent to $G_X \cdot n$, where $G_X = \sum_{u \in Q_X} p_X(u) \sum_{\substack{u \xrightarrow{\alpha} \varepsilon \\ u\alpha \neq X}} \pi(\alpha)$.*

5.2.4 Results for small patterns, discussion and perspectives

We conducted a comprehensive study of local branch prediction for MP and KMP and provide the code[14] that allows to quantify mispredictions for any alphabet size, any given pattern and any memoryless source for the input text (as for the examples given in Table 2).

[14]Python notebook (using `sympy`), available at https://github.com/vialette/branch-prediction/

X	$i = m$	$i >= 0$	Algo.	$X[i] \neq W[j]$
aa	$\kappa_2(p)$	$1-p$	MP	$p(1-p)(1+2p)/(1-p^2+p^3)$
			KMP	$p(1-p)/(1-2p+2p^2)$
ab	$p(1-p)$	$(1-p)^2$	both	$p(3-7p+7p^2-2p^3)/(1-p+2p^2-p^3)$
aaa	$\kappa_3(p)$	$1-p$	MP	$p(1-p)(1+p)^2$
			KMP	$p(1-p)/(1-2p+2p^2)$
aab	$p^2\,(1-p)$	$(1-p)^2(1+p)$	MP	$p(1+2p-p^2-8p^3+6p^4+5p^5-5p^6+p^7)$
			KMP	$p(1-2p^2-p^3+5p^4-3p^5+p^6)/(1-2p+3p^2-2p^3+p^4)$
aba	$p^2\,(1-p)$	$(1-p)^2$	MP	$p(3-7p+8p^2-4p^3+p^4)/(1-p+p^2)$
			KMP	$p(3-7p+7p^2-2p^3)/(1-p+2p^2-p^3)$
abb	$p(1-p)^2$	$(1-p)^3$	both	$p(4-13p+21p^2-16p^3+6p^4-p^5)$

Table 2: Asymptotic expected number of mispredictions per symbol in the text for each branch of Algorithm Find, for all normalized patterns of length 2 and 3 over the alphabet $A = \{a, b\}$. For readability, we set $p := \pi(a) = 1 - \pi(b)$. Notably, for patterns ab and abb, the failure functions used by MP and KMP coincide, resulting in identical behavior. The functions κ_2 and κ_3 are defined in Proposition 48.

Notably, the expressions for the number of mispredicted letter comparisons become increasingly complex as the pattern length grows and as the alphabet size increases. For instance, for the pattern $X = abab$, with $\pi_a := \pi(a)$ and $\pi_b := \pi(b)$, we obtain:

$$L_{abab} = \frac{\pi_a(-\pi_a^3\pi_b + 2\pi_a^2\pi_b^3 + 4\pi_a^2\pi_b^2 + 3\pi_a^2\pi_b + \pi_a^2 - 5\pi_a\pi_b^2 - 4\pi_a\pi_b - 2\pi_a + 2\pi_b + 1)}{(1-\pi_a)(\pi_a^2\pi_b^2 + \pi_a^2\pi_b - \pi_a\pi_b - \pi_a + 1)}.$$

The results given in Table 3 illustrate this for the uniform distribution, for small patterns and alphabets. In particular, the branch $i \geq 0$, which is poorly predicted by its local predictor, exhibits a very high number of mispredictions when $|A| = 4$, while the branch that comes from letter comparisons, $X[i] \neq W[j]$, experiences fewer mispredictions. This trend becomes more pronounced as the size of the alphabet increases: for $X = abb$ and $|A| = 26$, the misprediction rate for the test $i \geq 0$ reaches 0.96, whereas for $X[i] \neq W[j]$, it drops to 0.041.

Our work presents an initial theoretical exploration of pattern matching algorithms within computational models enhanced by local branch prediction. However, modern processors often employ hybrid prediction mechanisms that integrate both local and global predictors, with global predictors capturing correlations between branch outcomes across different execution contexts. In our simulations with `PAPI`[15] on a personal computer, the actual number of mispredictions is roughly divided by $|A|$ in practice. A key direction for further research is to develop a theoretical model that incorporates both predictors, allowing for more precise measurement in real-world scenarios. Another important line of research is to account for more sophisticated probabilistic distributions for texts, as real-world texts are often badly modeled by memoryless sources. For instance, Markovian sources should be manageable within our model and could provide a more accurate framework for the analysis.

In the first series of four figures below (Fig. 40), we used our formulas to compute the expected number of mispredictions for each branch, as well as the total number of mispredic-

[15]`PAPI 5.4.1.0` , see http://icl.cs.utk.edu/papi.

	$\lvert A\rvert = 2$					$\lvert A\rvert = 4$				
X	`i=m`	`i>=0`	algo	X `[i]!=`W`[j]`	Total	`i=m`	`i>=0`	algo	X `[i]!=`W `[j]`	Total
aa	0.283	0.5	MP	0.571	1.353	0.073	0.75	MP	0.295	1.117
			KMP	0.5	1.283			KMP	0.3	1.123
ab	0.25	0.25	both	0.571	1.321	0.062	0.688	both	0.375	1.186
aaa	0.14	0.5	MP	0.563	1.202	0.018	0.75	MP	0.293	1.06
			KMP	0.5	1.14			KMP	0.3	1.068
aab	0.125	0.375	MP	0.605	1.23	0.015	0.734	MP	0.322	1.086
			KMP	0.542	1.166			KMP	0.322	1.086
aba	0.125	0.25	MP	0.708	1.083	0.015	0.688	MP	0.367	1.068
			KMP	0.571	0.946			KMP	0.375	1.076
abb	0.125	0.125	both	0.547	0.921	0.015	0.672	both	0.397	1.098

Table 3: Asymptotic expected number of mispredictions per input symbol in a random text W, using Algorithm Find, assuming a uniform distribution over alphabets of size 2 and 4.

tions. We ran our code for the patterns *aaaa*, *aaab*, *abab*, and *abbb*, each plot being generated in a few seconds on a standard laptop. The results displayed are the expected numbers of mispredictions per text symbol as $\pi(a)$ varies. In the last two figures (Fig. 41), we consider all prefixes of length at least 2 of *abababb* and compute the variation of the expected total number of mispredictions per text symbol, for both the MP and KMP. The results suggest a form of convergence as the length of the prefix increases, which is expected since reaching the rightmost states of $\mathcal{A}_X$ becomes increasingly unlikely.

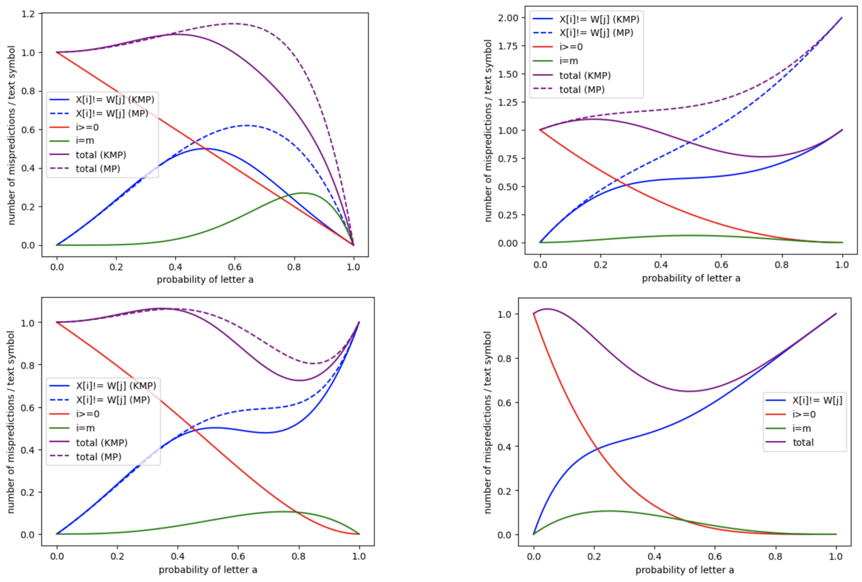


Figure 40: Expected number of mispredictions per text symbol as $\pi(a)$ varies, shown per branch and in total, for the patterns *aaaa* (top left), *abab* (top right), *aaab* (bottom left), and *abbb* (bottom right).

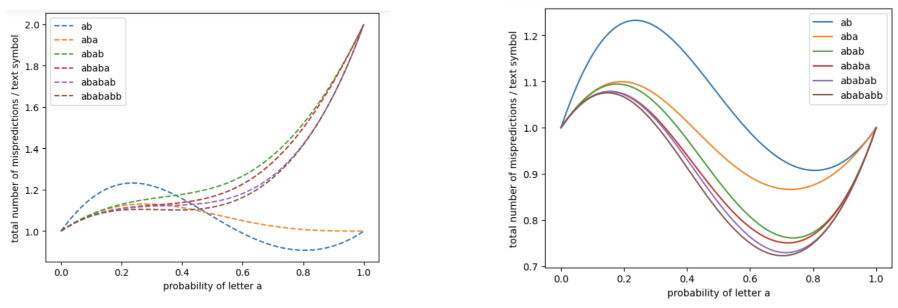


Figure 41: Expected number of mispredictions of MP (left) and KMP (right) in total, for the prefixes of *abababb*.

Summary of Our Results on Branch Prediction Analysis

Our goal was to refine the model used for algorithm analysis to gain deeper insights into real-world performance. To this end, we presented a set of results that explicitly integrate features of modern computer architecture. We studied variants of fundamental algorithms designed to improve their performance by accounting for branch prediction mechanisms. We obtained surprising theoretical improvements, such as an altered binary search where the split is not made exactly at the midpoint. These enhancements were validated through experiments implemented in both C and Java. Additionally, we measured the impact of branch prediction on classical pattern matching algorithms, starting with the naive sliding window method and moving on to the more sophisticated Morris-Pratt and Knuth-Morris-Pratt algorithms. The conditional instructions in these algorithms display strong correlations, making their analysis both challenging and distinct from previous theoretical work. In this second part of our study, a common feature is the reliance on probabilistic methods, exploiting Markov chain properties that emerge when 2-bit saturating predictors interact with the algorithmic steps under study. To our knowledge, existing theoretical analyses have not yet addressed more advanced prediction schemes, which are those currently implemented in modern computers. As an example among many others, the global two-level predictor of Fig. 26 maintains a history of recent branch outcomes across the entire program, enabling it to capture interactions between distinct conditional instructions and autocorrelations such as those present in pattern-matching algorithms. One of our objectives is to develop methods and results applicable to these more involved prediction mechanisms. We provided an initial glimpse of techniques relevant for such predictors in our analysis of binary search (see Section 4.3.3), and we intend to explore them further. In particular, we expect that these developments will yield theoretical insights more closely aligned with our experimental observations, especially in scenarios involving correlated branching instructions.

Ongoing Work

By choosing to focus here on realistic analysis of algorithms, I set aside another significant part of my work, namely analytic combinatorics and random generation. These topics are not entirely unrelated: random generation appeared in the discussion on record-biased permutations, and analytic combinatorics often provides powerful tools for average-case analysis. Nevertheless, the work presented here leans more toward a probabilistic approach. To acknowledge this other line of research, I will briefly comment on it here.

In the early stages of my work, I was naturally drawn to combinatorics. However, with a strong background in computer science, I always kept the practical aspects in mind. I began by working on automatic methods for the random generation of combinatorial structures, with the goal of providing ready-to-use libraries for researchers and developers. In particular, I focused on the Boltzmann method [DFLS04], which is based on results from *Analytic Combinatorics* [FS09], a field that studies the asymptotic properties of families of combinatorial objects using complex analysis on their generating functions.

Implementing the Boltzmann method requires evaluating these generating functions numerically at points inside their disk of convergence. Since they are often defined by implicit systems of equations, computing their values is nontrivial and requires computer algebra techniques such as Newton's method. Together with Bruno Salvy and Michèle Soria, we obtained significant results on this topic [PSS12], most notably the so-called Boltzmann oracle, which also turned out to be the first step toward automatically computing the radius of convergence of the generating series, a missing ingredient for much broader applications.

In their reference book *Analytic Combinatorics* [FS09], Flajolet and Sedgewick present a general approach that starts from a combinatorial specification, translates it into equations satisfied by generating functions, interprets these generating functions as analytic objects, and exploits their singular behavior to deduce asymptotic properties of the underlying combinatorial structures as their size grows. Continuing this line of work with Bruno Salvy, we developed computational tools to automate large parts of this approach. The outcome is an almost complete algorithmic chain that takes a combinatorial system and produces asymptotic expansions [PS25]. Below is an outline of these results.

Effective Asymptotics of Combinatorial Systems We begin by precisely characterizing the set of well-founded combinatorial systems that actually define combinatorial structures, and we use the dictionary from [FS09] to translate them into systems of equations on exponential generating functions. We then determine the radius of convergence of these generating functions, viewed as analytic functions, as this governs the exponential growth of their coefficients. For a given $a > 0$ inside the domain of convergence of $\boldsymbol{Y}$, the system $\boldsymbol{Y} = \boldsymbol{H}(a, \boldsymbol{Y})$ has $\boldsymbol{Y}(a)$ as a solution, i.e., the value of the generating function at a. It may also admit other solutions with nonnegative coordinates inside the domain of convergence of $\boldsymbol{H}$. However, for a larger than the radius of convergence of $\boldsymbol{Y}$, we prove that no such solution exists. This is a basis for computing of the radius of convergence by dichotomy, provided one can detect the existence of nonnegative solutions to such systems.

The systems we consider have strong positivity properties. This has an important consequence for Newton's iteration: for $a \in [0, \rho)$ where ρ is the radius of convergence of $\boldsymbol{Y}$, Newton's iteration started at $\boldsymbol{Y} = \boldsymbol{0}$ converges to the solution we want. We establish a converse of this result: if Newton's iteration started at $\boldsymbol{Y} = \boldsymbol{0}$ converges to a point $\boldsymbol{B}$ with nonnegative coordinates inside the domain of convergence of $\boldsymbol{H}$, then $a \leq \rho$ and $\boldsymbol{B} = \boldsymbol{Y}(a)$. In practice, one still needs to determine when to stop the iteration, but we can compute *a posteriori* bounds that guarantee correctness.

Newton's iteration can likewise be used to find the radius of convergence. In this case, however, the system no longer has the strong positivity property, so unconditional convergence from the origin is not guaranteed. Nevertheless, we prove that quadratic convergence holds in a neighborhood of the solution.

While the exponential growth of the coefficients of a generating function is determined by its radius of convergence ρ, subexponential terms in their asymptotic behavior are governed by the local behavior of the generating function at all the singularities on the circle $|z| = \rho$, called *dominant singularities*. The point ρ is always one such singularity, but others may also occur on this circle. Generating functions arising from combinatorial systems have a nice behavior: the arguments of their dominant singularities are rational multiples of π, and these rational numbers are induced by periodicity properties that can be computed from the combinatorial equations.

Once the dominant singularities are located, the next step is to compute the expansions of the generating functions in their neighborhood. For positive systems involving only polynomial or entire functions, the possible singular behaviors at finite singularities have been classified [BD15]. In that case, the exponents appearing in the expansions are rational numbers whose denominators are powers of 2. For combinatorial systems that may involve sets an cycles, the possibilities are much more diverse and we extends the classification to the general setting. There is a *gap* property: roughly speaking, as $z \to \rho$, either the generating function grows at least as fast as $\exp(c \ln^2(1 - z/\rho))$ for some $c > 0$, or it has an algebraic-logarithmic behavior. In the latter case, as in [BD15], the exponent $1/r$ is a power of 2; in the former, less regular exponents may occur. Another difference with the case of systems of polynomial or entire functions is that the exponents at nonreal dominant singularities can be nonreal as well. Despite these seemingly intricate exponents, we provide algorithms that compute the singular expansions of generating functions, provided they are of algebraic-logarithmic type.

Expansions of algebraic-logarithmic type are precisely those to which the transfer theorems of Flajolet and Odlyzko [FO90] apply. The end result of our work is that asymptotic expansions at arbitrary precision can be computed for the coefficients of all constructible generating functions that have algebraic-logarithmic dominant singularities. The only assumption is that Schanuel's conjecture [MW96] holds, or that the system contains neither sets nor cycles.

Other related work The automation of the treatment of combinatorial systems has been a recurring thread throughout my research. In particular, it led me to collaborate with Florent Koechlin and Pablo Rotondo on a computational approach to evaluating the expressiveness of systems of equations that generate random regular expressions (viewed as trees) [KPR25]. More concretely, we established bounds on the proportion of *universal* expressions, i.e., those that recognize every word over the alphabet, produced by such systems. The motivation behind this result, directly linked to the first part of the present study, is to provide concrete evidence that the uniform model may be inadequate for randomly testing algorithms that process regular expressions, due to the limited variety of languages it generates. This follows earlier work by my co-authors showing that, under the uniform model, universal expressions are overabundant. The bounds we obtained build on heuristics developed by my co-authors to refine the system of expression trees so as to distinguish, within each class, those that can be identified as universal. We extend this to more general systems involving several equations, which are more challenging to study, in a spirit similar to the results I obtained with Bruno Salvy, as described above.

My work on the Boltzmann method is another example of my ongoing effort to automate processes in combinatorics. Shortly after my PhD, I began working on Boltzmann samplers for classes of pattern-avoiding permutations [BBP+17]. These permutation classes have the particularity that, in general, they do not admit a proper combinatorial specification in the sense of [FS09]. With Frédérique Bassino, Mathilde Bouvel, Adeline Pierrot, and Dominique Rossin, we developed a methodology that automatically derive such a specification for a permutation class, given its basis of excluded patterns and the set of simple[16] permutations in the class, when both sets are finite. Only a small proportion of these classes, called substitution-closed classes, can be translated directly into specifications. In this case, we define a specification for the trees coming from the substitution decomposition of the permutations, that is, trees whose nodes correspond to permutations of size 2 or to the simple permutations in the class. For classes that are not substitution-closed, the process yields an ambiguous system of equations. We resolve this by transforming ambiguous unions into disjoint unions of terms that involve both pattern avoidance and pattern containment constraints. This somehow allows to interpret, on the combinatorial objects themselves, the result of applying the inclusion-exclusion to their generating functions. The outcome is a fully algorithmic approach that produces a positive system, from which a random sampler can be obtained directly via the Boltzmann method. This allowed us to observe various asymptotic behaviors in permutation diagrams, for which an explicit limit shape result was later established [BBF+22].

Recently, with Éric Fusy, we revisited the idea of extending the Boltzmann method from yet another perspective. Boltzmann samplers can generate random objects of any size, with the property that all objects of the same size are equally likely. A uniform random sampler for a given size n can therefore be obtained by rejection sampling: keep only the objects of size n and discard the others. Naturally, the efficiency of this approach is governed by the number and sizes of the discarded objects, i.e., by the size distribution in the Boltzmann model. For tree-like structures, this distribution heavily favors very small trees, so generating an object of size n typically requires discarding others whose total size is of order n. A remarkable aspect of the method is that all subtrees of a tree follow the same distribution as the tree itself. This means that the trees discarded during rejection have the same distribution as the nodes of any accepted tree. This naturally suggests reusing the rejected trees to help build a uniform tree of size n rather than discarding them. Using the idea behind the efficient generator for super-critical sequences from [DFLS04] and building on [BD24] by combining labelled and unlabelled trees via a bijection based on Prüfer code, we implemented an approach for designing almost-uniform samplers for several classes of unlabeled tree-like structures. This is still an ongoing project, but our experiments so far are very promising.

Implementations One common denominator across all these works is that they involved a substantial amount of coding. From the very beginning, I have implemented numerous random samplers, particularly Boltzmann samplers, tested them extensively, and used them to observe the behavior of other algorithms. And more recently, with Éric Fusy, we have been experimenting with our new samplers to precisely quantify how close our samples are to the uniform distribution. As explained earlier, doing so required the Boltzmann oracle. Together with Bruno Salvy, we spent almost ten years on its implementation, resulting in the Maple library NewtonGF[17]. While the algorithmic core is based on our theoretical results [PSS12], a major part of the effort went into automatically detecting whether Newton's iteration converges numerically. This aspect goes beyond the "simple" scope of computer algebra and proved to be particularly challenging. In the end, our library also computes the radius of convergence of the generating series for all systems treated in [FS09], which is the basis needed for implementing our new results on their asymptotic expansions.

In a different direction, our work on integrating architectural features into the analysis

[16]A permutation is simple if it has size at least 4 and contains no interval other than the trivial ones.
[17]Available at https://perso.ens-lyon.fr/bruno.salvy/software/the-newtongf-package/

of algorithms led to another kind of development. We spent a significant amount of time simulating and benchmarking branch predictor behavior across various algorithms. Using `Python`, we modeled different types of branch predictors, both local and global, with automata or history tables such as the one shown in Fig. 42. Together with Cyril Nicaud, we supervised two Master's internships: Victor Veillerette focused on pattern matching algorithms, and Tom Redon worked on classifying all possible k-bit predictors (viewed as Markov chains) by their misprediction rate in the stationary distribution, in an effort to explain empirical design choices made by processor manufacturers. When running benchmarks, a major difficulty is that many architectural features influence the speed during the execution of a program, including cache behavior, vectorization, and even the specific machine instructions generated by the compiler. Most of our code was written in `C` and was relatively straightforward, but we also had to examine large amounts of assembly code, in particular that produced by the `gcc` compiler, to study jump structures and propose a realistic predictor model. Another challenge was the lack of public information about the actual predictor implementations, which are proprietary. We began with a local branch prediction model, as it is easier to analyze, but it is very likely that our hardware uses global predictors or hybrids of local and global schemes. Using the PAPI library, we could access dynamic counters for the total number of branches and mispredictions, but this data is only partial: it does not reveal which branch caused a misprediction, whether a cache miss was linked to a mispredicted branch, or even which type of predictor was in use, for instance. While this experimental work does not appear directly in our theoretical results, they provided important groundwork toward a better understanding of modern processors, which we view as essential for building accurate models for algorithm analysis.

Future Work

Real-World Data and Implementations The works presented here open up many research directions. In the area of data-oriented analysis, our study of record-biased permutations has already led to a full characterization of the limit shape via the definition of a permuton (see [BNP25]). In addition, the methodology based on generative processes that we developed appears promising for designing and analyzing other models of biased distributions on permutations. As new sorting algorithm implementations employing heuristics to exploit input structure continue to emerge in widely used programming languages, developing corresponding analytical models appears both natural and necessary.

Focusing on the number of records was a good starting point, as it provided a way to obtain biased permutations toward nearly sorted inputs, which was a central concern in the design of TimSort. However, as shown by its analysis and by those of ShiversSort [Jug24] and PowerSort [MW18], it is even more natural to consider distributions in which the number of runs can be directly controlled. These algorithms aim to minimize the time complexity as a function of both the input length and the run-length entropy, achieving worst-case guarantees and supported by empirical evidence that the entropy is typically small. Therefore, developing and studying biased distributions over permutations with weights depending on the entropy of the run lengths, and potentially favoring permutations with low entropy, would be a valuable extension to our analytical framework.

Almost-sorted inputs are not the only examples of non-uniform inputs frequently encountered in practice. In his work introducing PatternDefeatingQuicksort [Pet21], Peters observes that inputs with many equal keys, and thus few distinct elements, occur frequently, as data are often sorted according to shared attributes (for example, car color). The galloping routine analyzed in [GJKY25] was, for instance, introduced in TimSort to improve efficiency on arrays with few distinct values. A natural extension of our work would be to model this regime explicitly by considering distributions over permutations with repetitions, i.e., multiset permutations, governed by a parameter controlling the number of distinct elements.

Since the models proposed above aim at reflecting the structure of real-world inputs, designing efficient random samplers within these models (following our approach using generative processes for record-biased permutations) would also provide useful tools for testing and potentially developing new sorting algorithms tailored to exploit such structure in their input. This would also make it possible to assess whether algorithms that are well-behaved in terms of complexity achieve good performance with respect to architectural measures such as cache hits or branch mispredictions, for instance.

Regarding TimSort, significant progress has already been achieved by Vincent Jugé and collaborators, who refined its analysis [GJKY25] and proposed a new variant [Jug24]. At the same time, as we already mentioned, new algorithms and heuristics, such as PatternDefeatingQuicksort which is implemented in both `Rust` and `C++`, continue to appear in modern programming languages. Existing libraries (algorithms and data structures) are also regularly refined, updated, or fully re-engineered. We believe that the choices made by the developers of these languages are often guided by their understanding of the application context and by benchmarks that are not necessarily publicly available. As the case of TimSort has shown, some of these choices could be particularly interesting to investigate.

For instance, if one takes a close look at `Java`'s implementation of the dual-pivot QuickSort for primitive arrays, it is actually a variant of IntroSort [Mus97], which switches to HeapSort when the recursion stack becomes too deep. As noted by Musser, a key parameter for this approach to be effective is the depth at which the switch occurs, which he determined empirically, observing that it may strongly depend on the underlying hardware architecture. A fine-grained theoretical analysis of HeapSort, in both the worst and average cases, could

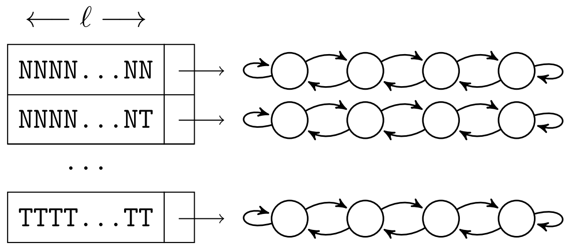


Figure 42: Two-level predictor: a history table of size 2^ℓ records the outcomes of the last ℓ branches, with the most recent outcome in the rightmost bit. Each sequence of ℓ branch outcomes is associated with a 2-bit saturated counter.

shed light on these questions. Moreover, incorporating architectural considerations, as we did in the second part of this study, could further enrich such an analysis, since cache handling is one of the main drawbacks of HeapSort, due to the underlying data structure.

These kinds of questions are in fact among the main objectives of a new ANR Project in which I am involved, PLASMA[18], which focuses on developing relevant models for realistic inputs and computer architecture features.

Enhancing the Model with Computer Architecture Features Obviously, I am also particularly interested in the second aspect addressed by our PLASMA project, as it aligns with the focus of the second part of this presentation, namely branch prediction analysis. The work we have carried out has provided a solid theoretical understanding of local predictors, as illustrated here on classical divide-and-conquer algorithms such as binary search, as well as on more intricate pattern-matching algorithms like KMP, for instance. Even though the technical steps for a full analysis can be quite involved for the latter, due to the high correlation between its branches, we established fundamental tools for analyzing such mechanisms. In particular, Markov chains naturally emerge as the appropriate framework for this type of analysis. We consider these first results as preliminary, in the sense that branch predictors found in contemporary processors, even the simplest ones, employ far more sophisticated schemes. One natural next step is therefore to extend our results on local predictors to other, more realistic types of predictors.

Two-level predictors make their predictions using a limited history of recent branch outcomes (see Fig. 42). They are generally classified as either local or global, depending on whether each branch has its own history-based prediction table or whether all branches share the same table. In both cases, unlike a simple saturated-counter predictor, a sufficiently long history table can accurately capture correlations between branches, such as those occurring in pattern matching. This ability could explain the differences we observed between our theoretical results and our empirical measurements. In principle, two-level predictors can be analyzed using Markov chains, but with realistic history sizes the resulting chains become prohibitively large. During his internship, Victor Veillerette studied this approach and found that with an 8-bit history (a size that appears to be standard), even a small pattern of length 6 leads to a chain with roughly 10^5 states. Our preliminary work suggests, however, that such chains often have a rich symmetry structure. The objective is to systematically identify these symmetries to make the analysis tractable. Since such predictors are implemented in current processors, the results should be directly observable and have practical impact.

Many other prediction schemes are either already in use or have been proposed in the literature. Some, such as neural-network–based predictors [ESQ+03], are probably beyond mathematical tractability. Others, like schemes using multiple history sizes [SM06], are natural generalizations of the predictors described above and should be amenable to our methodology.

[18]PLASMA (Programming Languages, Algorithms and Structures: Models and Analysis) is an ANR research project that will begin in 2026. See https://protondo.github.io/anr-plasma/

The results we obtained so far concern isolated cases in which the analysis of branch predictions was tractable. A longer-term objective would be to develop a more general framework encompassing these results. This could be achieved by analyzing broader classes of algorithms within a unified setting, allowing for the derivation of more general conclusions. For instance, many pattern-matching algorithms share the property of inducing correlations between branches. One could aim to quantify algorithmic complexity with respect to a measure of this correlation. As an example, it would be interesting to determine the expected number of mispredictions when the probability of correlation between successive branches decreases exponentially. Another promising direction would be to generalize the type of analysis we conducted for binary search by defining an ideal path in the branching process of an algorithm and bounding its deviation from actual execution paths. Pursuing such ideas could lead to results of broader applicability and deeper theoretical insight.

Finally, since our work focused on branch predictor behavior, we isolated it from other architectural effects, particularly its interactions with the cache. However, during our experiments on binary search, we observed that the algorithm's performance was strongly affected by the cache accesses occurring during the execution of mispredicted branches, which calls for further investigation in this direction.

More broadly, pursuing realistic analyses would naturally involve cache-aware modeling. A still not so much looked direction would be to develop analyses that more faithfully capture cache behavior by incorporating additional parameters beyond cache and block size. Much of the existing work on cache-oblivious and cache-aware algorithms assumes a fully associative cache, meaning that each memory block can be loaded into any cache line (see, for example, [Dem02, FLPR12, BDE+16]). While this model has produced valuable theoretical insights, it fails to capture the actual behavior of modern hardware, where caches typically have low associativity, i.e., each memory block can be stored only in a limited subset of cache lines (see, for example, [SCD02, MS03]). When associativity is taken into account, another common simplifying assumption is the use of a fully random hash function to determine the destination subset, whereas in practice, this mapping is usually achieved through direct addressing based on specific bits of the memory address. Beyond being more realistic, this mechanism imposes a strong structural constraint on data placement that could be fruitfully captured and studied using combinatorial methods. Finally, the cache replacement policy used in analyses involving associativity is almost always LRU (Least Recently Used), meaning that when the cache is full, the block that has not been accessed for the longest time is evicted and replaced by a new one. Although LRU is conceptually simple and analytically convenient, it is rarely implemented in its pure form on modern processors due to hardware complexity. Instead, real systems often employ pseudo-LRU or alternative strategies such as random or FIFO, whose influence on algorithmic behavior remains largely unexplored.

To conclude, I would like to emphasize how strongly my research directions are influenced by my teaching. As is probably clear from the preceding discussion, I teach algorithmics, programming in various languages, and computer architecture, all of which give a distinctive orientation to my work. In addition, my courses on network programming (including the design of client–server applications) and concurrent programming constitute a continuous source of real-world problems and applications (such as parallel cache management, scheduling strategies, or data management for servers, for instance) that will, I hope, continue to inspire and enrich my future research.

Bibliography

[ABNP16] Nicolas Auger, Mathilde Bouvel, Cyril Nicaud, and Carine Pivoteau. Analysis of Algorithms for Permutations Biased by Their Number of Records. In *27th International Conference on Probabilistic, Combinatorial and Asymptotic Methods for the Analysis of Algorithm (AofA)*, Cracovie, Poland, July 2016. Available at http://hal.archives-ouvertes.fr/hal-01838692.

[ABT03] Richard Arratia, A. D. Barbour, and Simon Tavaré. *Logarithmic combinatorial structures: a probabilistic approach*. EMS Monographs in Mathematics. EMS, Zürich, 2003.

[AJNP18] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. In *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 4:1–4:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. Extended version available at: http://arxiv.org/abs/1805.08612.

[ANP15] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Merge strategies: From Merge Sort to TimSort. Research Report hal-01212839, hal, 2015. Available at http://hal.science/hal-01212839.

[ANP16] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. Good predictions are worth a few comparisons. In *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 12:1–12:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. Available at: http://doi.org/10.4230/LIPIcs.STACS.2016.12.

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[AWFS17] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BBF$^+$22] Frédérique Bassino, Mathilde Bouvel, Valentin Féray, Lucas Gerin, and Adeline Pierrot. Scaling limits of permutation classes with a finite specification: a dichotomy. *Advances in Mathematics*, 405:Article 108513, 2022.

[BBP$^+$17] Frédérique Bassino, Mathilde Bouvel, Adeline Pierrot, Carine Pivoteau, and Dominique Rossin. An algorithm computing combinatorial specifications of permutation classes. *Discret. Appl. Math.*, 224:16–44, 2017. Available at http://arxiv.org/abs/1506.00868.

[BD15] Cyril Banderier and Michael Drmota. Formulae and asymptotics for coefficients of algebraic functions. *Combinatorics, Probability and Computing*, 24:1–53, 1 2015.

[BD24] Laurent Bartholdi and Persi Diaconis. An algorithm for uniform generation of unlabeled trees (pólya trees), with an extension of cayley's formula, 2024. To appear Forum of Math. Sigma, available at: http://arxiv.org/abs/2411.17613.

[BDE$^+$16] Michael A. Bender, Erik D. Demaine, Roozbeh Ebrahimi, Jeremy T. Fineman, Rob Johnson, Andrea Lincoln, Jayson Lynch, and Samuel McCauley. Cache-adaptive analysis. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 135–144, New York, NY, USA, 2016. Association for Computing Machinery.

[BFM08] Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. On the adaptiveness of quicksort. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-Oriented Software. The KeY Approach: Foreword by K. Rustan M. Leino*, volume 4334. Springer, 2007.

[Bil12] Patrick Billingsley. *Probability and Measure.* John Wiley and Sons, anniversary edition, 2012.

[BK19] Sam Buss and Alexander Knop. Strategies for stable merge sorting. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, page 1272–1290, USA, 2019. Society for Industrial and Applied Mathematics.

[BM05] Gerth Stølting Brodal and Gabriel Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *Algorithms and Data Structures, 9th International Workshop (WADS), Waterloo, Canada, August 15-17, 2005, Proceedings*, volume 3608 of *Lecture Notes in Computer Science*, pages 385–395. Springer, 2005.

[BM06] Gerth Stølting Brodal and Gabriel Moruz. Skewed Binary Search Trees. In *Algorithms ESA 2006*, volume 4168, pages 708–719. Springer Berlin Heidelberg, 2006.

[BN13] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theor. Comput. Sci.*, 513:109–123, 2013.

[BNP25] Mathilde Bouvel, Cyril Nicaud, and Carine Pivoteau. Record-biased permutations and their permuton limit. *Combinatorics, Probability and Computing*, 2025. To appear.

[BNWG08] 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.

[Bón12] Miklós Bóna. *Combinatorics of permutations.* Chapman-Hall and CRC Press, second edition, 2012.

[CHL07] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings.* Cambridge University Press, 2007.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 4th edition, 2009.

[Cor22] Benoît Corsini. The height of record-biased trees. *Random Structures and Algorithms*, on-line first, August 2022.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, 1994.

[Dem02] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets.* BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[DFLS04] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4–5):577–625, 2004.

[DGRdB$^+$15] 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.

[EKS12] Amr Elmasry, Jyrki Katajainen, and Max Stenmark. Branch Mispredictions Don't Affect Mergesort. In *Experimental Algorithms*, volume 7276, pages 160–171. Springer Berlin Heidelberg, 2012.

[ESQ$^+$03] Colin Egan, Gordon Steven, Patrick Quick, Rubén Anguera, Fleur Steven, and Lucian Vintan. Two-level branch prediction using neural networks. *Journal of Systems Architecture*, 49(12–15):557–570, December 2003.

[EW19] Stefan Edelkamp and Armin Weiß. Blockquicksort: Avoiding branch mispredictions in quicksort. *Journal of Experimental Algorithmics (JEA)*, 24:1–22, 2019.

[Ewe72] Warren J Ewens. The sampling theory of selectively neutral alleles. *Theoretical population biology*, 3(1):87–112, 1972.

[Fér13] Valentin Féray. Asymptotics of some statistics in Ewens random permutations. *Electronic Journal of Probability*, 18(76):1–32, 2013.

[FLPR12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. 8(1), January 2012.

[FM70] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970.

[FO90] Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM Journal on Discrete Mathematics*, 3(2):216–240, 1990.

[FS09] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics.* Cambridge University Press, 2009.

[GJK22] Elahe Ghasemi, Vincent Jugé, and Ghazal Khalighinejad. Galloping in fast-growth natural merge sorts. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 68:1–68:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[GJKY25] Elahe Ghasemi, Vincent Jugé, Ghazal Khalighinejad, and Helia Yazdanyar. Galloping in fast-growth natural merge sorts. *Algorithmica*, 87(2):242–291, 2025.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 6th edition, 2017.

[JKB95] Norman L. Johnson, Samuel Kotz, and N. Balakrishnan. *Continuous Univariate Distributions.* John Wiley and Sons, second edition, 1995.

[Jug20] Vincent Jugé. Adaptive shivers sort: An alternative sorting algorithm. In *Proceedings of the 31th ACM-SIAM Symposium on Discrete Algorithms, (SODA), Salt Lake City, UT, USA, January 5-8, 2020*, pages 1639–1654. SIAM, 2020.

[Jug24] Vincent Jugé. Adaptive shivers sort: An alternative sorting algorithm. *ACM Trans. Algorithms*, 20(4):31:1–31:55, 2024.

[KJP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publish. Co., Redwood City, CA, USA, 1998.

[KPR25] Florent Koechlin, Carine Pivoteau, and Pablo Rotondo. Heuristic universality detection over regular expressions specified by systems. In *Developments in Language Theory (DLT) - 29th International Conference, Seoul, South Korea, August 19–22, 2025, Proceedings*, volume 16036 of *Lecture Notes in Computer Science*. Springer, 2025. Available at http://hal.science/hal-05211135.

[KS06] Kanela Kaligosi and Peter Sanders. How Branch Mispredictions Affect Quicksort. In *Algorithms ESA 2006*, volume 4168, pages 780–791. Springer Berlin Heidelberg, 2006.

[LL99] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999.

[Lot97] Monsieur Lothaire. *Combinatorics on words*, volume 17. Cambridge university press, 1997.

[LPW08] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times.* American Mathematical Society, 2008.

[Mal57] Colin L Mallows. Non-null ranking models. I. *Biometrika*, 44(1/2):114–130, 1957.

[Man85] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Computers*, 34(4):318–325, 1985.

[Mit18] Sparsh Mittal. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience*, 31, 2018.

[MNW15] 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), San Diego, CA, USA, January 4, 2015*, pages 114–128, 2015.

[MP70] James H Morris, Jr and Vaughan R Pratt. A Linear Pattern-Matching Algorithm. Technical report, University of California, Berkeley, CA, 01 1970.

[MS03] Kurt Mehlhorn and Peter Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35(1):75–93, 2003.

[Mus97] David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8):983–993, August 1997.

[MW96] Angus Macintyre and A. J. Wilkie. On the decidability of the real exponential field. In *Kreiseliana*, pages 441–467. A K Peters, Wellesley, MA, 1996.

[MW18] J. Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In *26th Annual European Symposium on Algorithms (ESA 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 63:1–63:15, 2018.

[NPV24] Cyril Nicaud, Carine Pivoteau, and Stéphane Vialette. Theoretical Analysis of Branch Prediction in the Sliding Window Algorithm. Working paper, available at http://hal.science/hal-05170281, February 2024.

[NPV25] Cyril Nicaud, Carine Pivoteau, and Stéphane Vialette. Branch prediction analysis of morris-pratt and knuth-morris-pratt algorithms. In *36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, June 17-19, 2025, Milan, Italy*, volume 331 of *LIPIcs*, pages 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. Version with appendices available at: http://hal.science/hal-05147509.

[OLBC10] Frank W. J. Olver, Daniel W. Lozier, Ronald F. Boisvert, and Charles W. Clark, editors. *NIST Handbook of Mathematical Functions*. Cambridge University Press, 2010.

[Pet] Tim Peters. Timsort description, accessed july 2025. http://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[Pet15] T. Kyle Petersen. *Eulerian Numbers*. Birkhäuser Advanced Texts Basler Lehrbücher, Springer, New York, first edition, 2015.

[Pet21] Orson RL Peters. Pattern-defeating quicksort. *arXiv preprint arXiv:2106.05123*, 2021.

[PM95] O. Petersson and A. Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, may 1995.

[Poh72] Ira Pohl. A sorting problem and its complexity. *Communications of the ACM*, 15(6):462–464, 1972.

[PS25] Carine Pivoteau and Bruno Salvy. Effective asymptotics of combinatorial systems, 2025. 78 pages. Submitted. Available at: http://arxiv.org/abs/2508.20008.

[PSS12] Carine Pivoteau, Bruno Salvy, and Michèle Soria. Algorithms for combinatorial structures: Well-founded systems and newton iterations. *Journal of Combinatorial Theory, Series A*, 119(8):1711 – 1773, 2012. 62 pages. Available at http://arxiv.org/abs/1109.2688.

[Rég89] Mireille Régnier. Knuth-Morris-Pratt algorithm: An analysis. In *Mathematical Foundations of Computer Science 1989, MFCS'89, Porabka-Kozubnik, Poland, August 28 - September 1, 1989, Proceedings*, volume 379 of *Lecture Notes in Computer Science*, pages 431–444. Springer, 1989.

[Rou01] Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *Journal of the ACM*, 48(2):170–205, 2001.

[RS98] Mireille Régnier and Wojciech Szpankowski. Complexity of sequential pattern matching algorithms. In *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, volume 1518 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 1998.

[SCD02] Sandeep Sen, Siddhartha Chatterjee, and Neeraj Dumir. Towards a theory of cache-efficient algorithms. *J. ACM*, 49(6):828–858, November 2002.

[Sed77] Robert Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7(4):327–355, 1977.

[SM06] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8:23, February 2006.

[SW04] Peter Sanders and Sebastian Winkel. Super scalar sample sort. In *Algorithms – ESA 2004*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796. Springer Berlin Heidelberg, 2004.

[Yar] Vladimir Yaroslavskiy. Dual-pivot quicksort algorithm, 2009. URL: http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf.