Good Predictions Are Worth a Few Comparisons

Nicolas Auger[1], Cyril Nicaud[2], and Carine Pivoteau[3]

1 **Université Paris-Est, LIGM (UMR 8049), 77454 Marne-la-Vallée, France**
2 **Université Paris-Est, LIGM (UMR 8049), 77454 Marne-la-Vallée, France**
3 **Université Paris-Est, LIGM (UMR 8049), 77454 Marne-la-Vallée, France**

Abstract

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. We propose predictor-friendly versions of two classical algorithms: exponentiation by squaring and binary search in a sorted array. These variants result in less mispredictions on average, at the cost of an increased number of operations. These theoretical results are supported by experimentations that show that our algorithms perform significantly better than the standard ones, for primitive data types.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases branch misses, binary search, exponentiation by squaring, Markov chains

Digital Object Identifier 10.4230/LIPIcs.STACS.2016.12

1 Introduction

As an introductory example, 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. 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 to the current minimum and the greatest to the current maximum. This uses only $3n/2$ comparisons, which is optimal.[1]

In order to observe the benefit of this optimization, we implemented both versions (see Figure 3) and measured their execution time[2] for large arrays of uniform random `float` in $[0,1]$. The results are given in Figure 1 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 performances. 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. 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. Every time a conditional is used in a program, there is a mechanism that tries to predict whether the corresponding conditional jump will be taken or not. The cost of a misprediction can be quite large compared to a basic instruction, and 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 branch that is unpredictable (for an input taken uniformly

[1] 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 [11].
[2] We used a Linux machine with a 3.40 GHz Intel Core i7-2600 CPU.

© Nicolas Auger, Cyril Nicaud, and Carine Pivoteau;
licensed under Creative Commons License CC-BY
33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016).
Editors: Nicolas Ollinger and Heribert Vollmer; Article No. 12; pp. 12:1–12:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

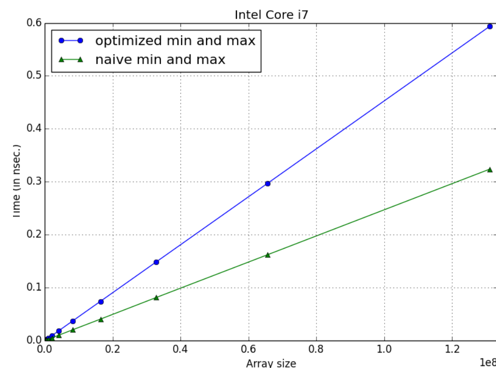


Figure 1 Execution time of simultaneous minimum and maximum searching.

at random) and will cause a substantial increase in the number of mispredictions. As we will see in the sequel, the expected number of mispredictions caused by the naive algorithm is $\Theta(\log n)$, whereas it is $\Theta(n)$ for the "optimal" one.

The influence of branch predictors over comparison based algorithms has already been studied, mostly to acknowledge the over-cost induced by mispredictions. Our approach is quite the opposite as we propose to take advantage of this feature, by proposing predictor-friendly versions of two classical algorithms.

Our contributions. After dealing with our introductory example using combinatorial arguments, we turn our attention towards the classical exponentiation by squaring and give a simple alternative algorithm, which reduces the number of mispredictions without increasing the number of multiplications. The analysis is based on the study of the Markov chains that describe the dynamic local predictors (see next section for a brief description of predictors). Finally, in the same vein, we propose biased versions of the binary search in a sorted array. We analyze the expected number of mispredictions for local predictors and we also give the (first to our knowledge) analysis of a global 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.

Related work. Over the past decade, several articles began to address the influence of branch predictors, and especially the cost of mispredictions, in comparison based algorithms. For instance, Biggar and his coauthors [1] 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 [3] and studied how the number of inversions in the data affects statistics such as the number of mispredictions [2]. 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 [13]. Elmasry, Katajainen and Stenmark then proposed a version of MergeSort that is not affected

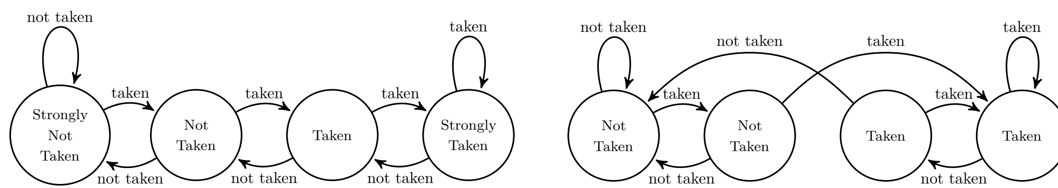


Figure 2 Two different 2-bit predictors (left: saturating counter, right: flip-on-consecutive).

by mispredictions [6], 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 [8]; 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 [10] implemented in Java's standard library.

Besides, Brodal and Moruz conducted an experimental study of skewed binary search trees in [4], 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. 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.

2 Elements of Computer Architecture

To analyze the complexity of searching or sorting algorithms, the standard model consists in counting the number of comparison operations performed. However most modern processors are pipelined. And to avoid stalling the pipeline when coming across a conditional jump, the processor tries to predict if the jump will occur and proceeds according to its prediction. A correctly predicted jump does not stall the pipeline whereas mispredictions lead this one to be flushed, causing a significant performance loss.[4] Therefore, the cost of a comparison in an "if" statement actually depends on the quality of the prediction.

For any conditional jump, a branch predictor will "guess" if the corresponding branch will be taken or not. For this purpose, many different strategies have been designed. The simplest one is a *static* branch predictor that does not use information from the code execution. It can, for example, predict that all branches will be taken. To improve its accuracy, a *dynamic* branch predictor uses the outcome of past branches to guess whether a particular branch should be taken or not. We now describe some techniques of dynamic branch prediction [7].

A *1-bit predictor* is a state buffer which remembers the last outcome of the branch; the guess is that the next outcome will be the same. As an improvement, the *2-bit predictors* try to avoid making two mispredictions when a branch takes an unlikely path. Two slightly different schemes are given by Figure 2. The saturating counter scheme can be further improved by keeping more information (*k-bit predictors* using 2^k states). All these predictors are *local*: there is one for every conditional (up to some limit in practice).

An *history table* has 2^n entries indexed by the sequence of the last n branches (1 for taken, 0 otherwise). The entries themselves are usually k-bit predictors. Such a table is said to be *local* when its entries correspond to the behavior of one specific branch and are used for

[3] Namely, conditional moves, which are now widely available on computers.
[4] For instance, on an Intel Core i7, a misprediction causes a penalty of about 15 cycles [7].

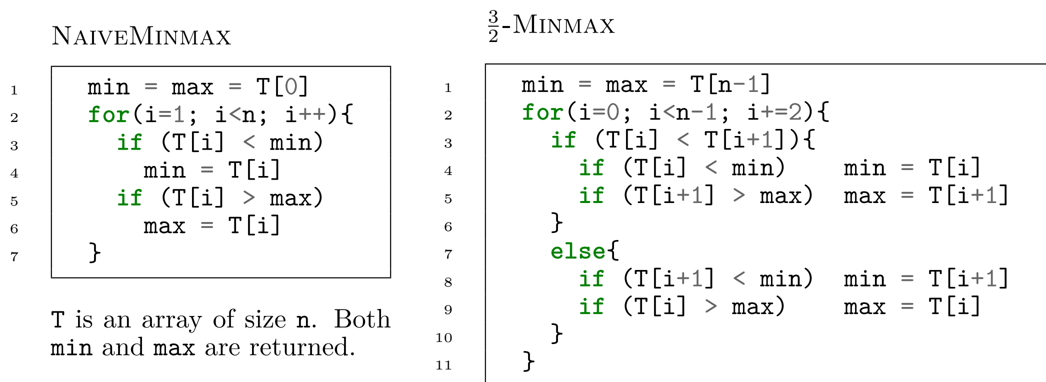


Figure 3 Naive and optimized implementations of simultaneous maximum and minimum finding.

this one only. On the contrary, in a *global* history table, the outcomes of the most recently executed branches are used to index the table, which is shared by all the conditionals.

To get the best of both worlds, *correlating branch predictors* use local and global information mixed together, and *tournament predictors* use an additional dynamic scheme to decide if they follow the local or the global prediction. These types of predictors are far beyond what we study in this article, but are worth mentioning for further analysis.

Strictly speaking, mispredictions can only be analyzed on a given assembly code, as they occur at conditional jumps. In the sequel, we use C-style pseudo code. We implicitly work on the non-optimized assembly code (compiled[5] with `gcc -O0`), where control structures are translated into conditional jumps in the standard way (i.e., not into conditional moves). For our experimental results, we checked that it was indeed the case. Furthermore, we remarked that our good results still hold for fully optimized binaries (compiled with `gcc -O3`).

3 Simultaneous Maximum and Minimum Finding

In this section, we go back to the example given in the introduction: We consider two algorithms that simultaneously compute the minimum and the maximum of an array of length n. These algorithms are given in Figure 3 (see also [5, Sec. 9.1]). For our analysis, we consider the local predictors presented in Section 2.

In the classical settings for the analysis, the algorithm $\frac{3}{2}$-Minmax is optimal (see the footnote[1] in the introduction), the number of comparisons performed being asymptotically equivalent to $\frac{3}{2}n$. Obviously, NaiveMinmax needs $2n - 2$ comparisons.

In order to give an explanation of the experimental results presented in Figure 1, 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 $\{1, \ldots, n\}$, 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,

[5] We used version `4.5.3` of `gcc`.

the first conditional at line 3 (resp. the one at line 5) is true for each min-record (resp. max-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 1.** *The expected number of mispredictions performed by* NaiveMinmax*, for the uniform distribution on arrays of size n, is asymptotically equivalent to $4\log n$ for the 1-bit predictor and to $2\log n$ for the two 2-bit predictors and the 3-bit saturating counter. The expected number of mispredictions performed by $\frac{3}{2}$-*Minmax *is asymptotically equivalent to $\frac{n}{4} + \mathcal{O}(\log n)$ for all the considered predictors.*

In light of these results, we observe that the mispredictions occurring in NaiveMinmax are negligible towards 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 (Figure 1), since we know that mispredictions can cost many CPU cycles and that comparisons are cheap operations. Of course, we are aware that other factors can influence the performances 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`[5] compiler (`-O3`) to check that these results withstand strong code optimization.[6] In this particular case, all the branches but the one at line 3 in $\frac{3}{2}$-Minmax are replaced by conditional moves that are not vulnerable to misprediction. Hence, $\frac{3}{2}$-Minmax still causes approximatively $\frac{1}{4}n$ mispredictions on average.

Of course, these results do not hold when considering a non-uniform distribution over the entries. It would be interesting to study the behavior of both algorithms on random permutations with, for instance, a given number of records.

4 Exponentiation by Squaring

We saw in the previous section that conditional branches with equal probabilities of going one way or another are particularly harmful when using branch prediction. Besides, several divide and conquer algorithms feature such branches, since they tend to split problems into parts of equal size to reach an optimal complexity. In the sequel, we explore two different ways of disrupting this balance, to end up with better performances for two classical algorithms: exponentiation by squaring and binary search.

4.1 Modified algorithms

The classical divide and conquer algorithm to compute x^n consists in 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 Figure 4. As expected, the conditional branch of line 3 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, Figure 4) using the decomposition $x^n = (x^4)^{\lfloor n/4 \rfloor} (x^2)^{n_1} x^{n_0}$. Still, both conditional 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 00 or not. This is the third algorithm

[6] Both algorithms are faster, as expected, but the naive version is still almost twice as fast.
[7] In our model, n is chosen uniformly at random between 0 and $4^k - 1$ for some positive k.

ClassicalPow (x,n)

```
1  r = 1;
2  while (n > 0) {
3    // n is odd
4    if (n & 1)
5      r = r * x;
6    n /= 2;
7    x = x * x;
8  }
```

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

UnrolledPow (x,n)

```
1  r = 1;
2  while (n > 0) {
3    t = x * x;
4    // n_0 == 1
5    if (n & 1)
6      r = r * x;
7    // n_1 == 1
8    if (n & 2)
9      r = r * t;
10   n /= 4;
11   x = t * t;
12 }
```

GuidedPow (x,n)

```
1  r = 1;
2  while (n > 0) {
3    t = x * x;
4    // n_1 n_0 != 00
5    if (n & 3) {
6      if (n & 1)
7        r = r * x;
8      if (n & 2)
9        r = r * t;
10   }
11   n /= 4;
12   x = t * t;
13 }
```

Figure 4 Three versions of the exponentiation by squaring, in C. The `&` denotes the bitwise `AND` in the C language.

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 5 Some parameters measured during 5.10^7 computations of x^n with the three algorithms of Figure 4, using the PAPI library.[8] The values of n are chosen uniformly at random between 0 and $2^{26}-1$. The number of branches is given excluding the ones caused by loops, as they do not yield mispredictions.

of Figure 4. Note that this conditional 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 at using 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 latest version of the PAPI library,[8] which give access, for instance, to the number of mispredictions occurring during the execution. These results are depicted in Figure 5. 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 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 conditionals of line 6 and 8, which leads to an overall decrease in the number of mispredictions). We are in similar settings as for the simultaneous minimum and maximum, where the increased number of comparisons is balanced by less mispredictions.

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

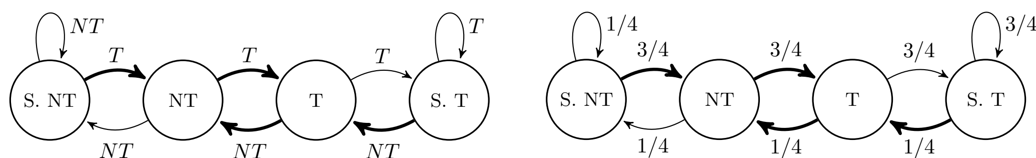


Figure 6 The saturating counter and its associated Markov chain for the first conditional of GuidedPow. The bold edges correspond to mispredictions.

We now proceed with the analysis of this phenomenon.

4.2 Analysis of the Average Number of Mispredictions for 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 2.

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

We now recall, using our algorithm as an example, why Markov chains are the key tools for that kind of analysis (as shown in [8, 10]). Let us consider the first conditional 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 is exactly described by the Markov chain obtained when changing the edges labels "taken" by $\frac{3}{4}$ and the labels "not taken" by $\frac{1}{4}$ (see Figure 6). A misprediction occurs whenever an edge labeled by "taken" (resp. "not taken") is used from a state that predicts "not taken" (resp. "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 classical Ergodic Theorem [9], which we restated bellow in order to fit our needs.

▶ **Theorem 2** (Ergodic Theorem). *Let (M, π_0) be a primitive and aperiodic Markov Chain on the finite set S. Let π 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 on 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: $\mathbb{E}[X_n] \sim \mathbb{E}[L_n] \sum_{(i,j)\in E} \pi(i)M(i, j)$.

When considering a given predictor, under the model where the condition is satisfied with probability p, we denote by M_p its transition matrix, by π_p its stationary vector and by $\mu(p)$ its *expected misprediction probability* defined by $\mu(p) = \sum_{(i,j)\in E} \pi_p(i)M_p(i, j)$, where E is the set of edges corresponding to mispredictions. As shown in [10], if we denote by $\mu_1(p)$, $\mu_2(p)$ and $\mu_2'(p)$ the expected misprediction probability of the 1-bit, 2-bit saturating counter and the flip-on-consecutive 2-bit, respectively, then we have:

$$\mu_1(p) = 2p(1-p); \quad \mu_2(p) = \frac{p(1-p)}{1-2p(1-p)}; \quad \mu_2'(p) = \frac{2p^2(1-p)^2 + p(1-p)}{1-p(1-p)}. \tag{1}$$

Similarly, the expected misprediction probability $\mu_3(p)$ of the 3-bit saturated counter is

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

Applying these mathematical tools 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 5 we show how to deal with the cases where we slightly deviate from the ideal case.

▶ **Theorem 3.** *Assume that n is taken uniformly at random in $\{0,\dots,N-1\}$. The expected number of conditional tests 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 is asymptotically equivalent to $\frac{1}{2}\log_2 N$ for* ClassicalPow *and* UnrolledPow*, for any kind of predictor. For* GuidedPow*, it is asymptotically equivalent to $\alpha \log_2 N$, where $\alpha = \frac{1}{2}\mu(3/4) + \frac{3}{4}\mu(2/3)$, where μ is the expected misprediction probability associated with the local predictor.*

Using Theorem 3 and Equations (1) and (2), we get that α is equal to $\frac{25}{48} \approx 0.52$, $\frac{9}{20} = 0.45$, $\frac{2045}{4368} \approx 0.47$ and $\frac{1095}{2788} \approx 0.39$ for the 1-bit, 2-bit saturated, flip-on-consecutive 2-bit and 3-bit saturated counter, respectively. These values are to be compared with the $\frac{1}{2}$ of the other two algorithms. In particular, for the 1-bit predictor, the expected number of mispredictions is greater for GuidedPow than for ClassicalPow or UnrolledPow. This predictor is not efficient enough to offset the mispredictions caused by the additional conditional. For the 3-bit saturated counter, GuidedPow therefore uses $\approx 0.25\log_2 n$ more comparisons than UnrolledPow, but $\approx 0.11\log_2 n$ less mispredictions.

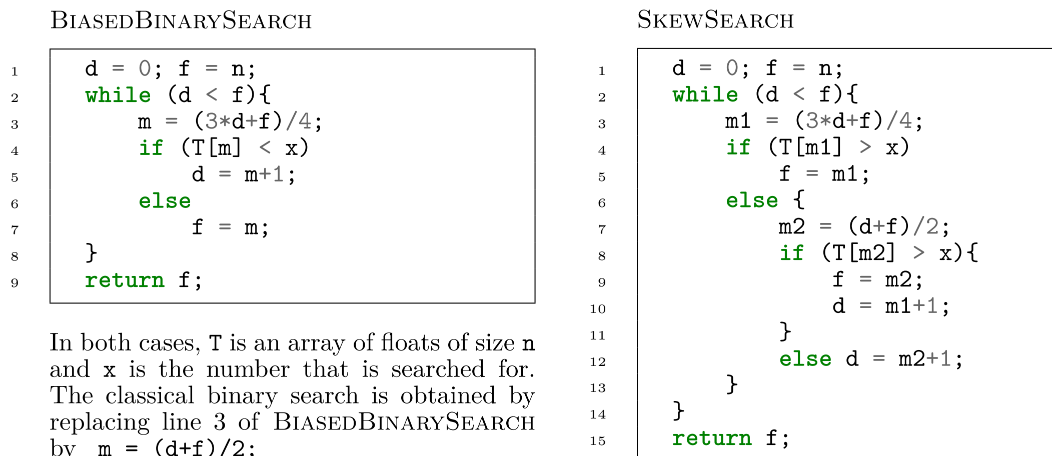
5 Binary Search and Variants

5.1 Unbalancing the Binary Search

We first consider the classical binary search which partitions a sorted array of size n into two parts of size $\frac{n}{2}$ and compares the value x that is searched for to the middle of the array in order to determine in which part of the array to continue the search. As before, if we consider arrays of uniform random floating-point numbers, we get a conditional branch that is taken with probability $\frac{1}{2}$. A simple way to change that is to partition another way, for instance with parts of size about $\frac{n}{4}$ and $\frac{3n}{4}$, as in the BiasedBinarySearch (see Figure 7). 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 costly in terms of 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 Figure 7).

5.2 Experiments

As expected at this point in our work, the BiasedBinarySearch experimentally performs better than the classical binary search and the SkewSearch performs much better. 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 is accessed. Thus we conducted experiments on arrays that fit in the last-level cache of our



BiasedBinarySearch

```
1  d = 0; f = n;
2  while (d < f){
3      m = (3*d+f)/4;
4      if (T[m] < x)
5          d = m+1;
6      else
7          f = m;
8  }
9  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

```
1  d = 0; f = n;
2  while (d < f){
3      m1 = (3*d+f)/4;
4      if (T[m1] > x)
5          f = m1;
6      else {
7          m2 = (d+f)/2;
8          if (T[m2] > x){
9              f = m2;
10             d = m1+1;
11         }
12         else d = m2+1;
13     }
14 }
15 return f;
```

Figure 7 Algorithms for the biased binary search and skew search. Both return the position where the element should be inserted.

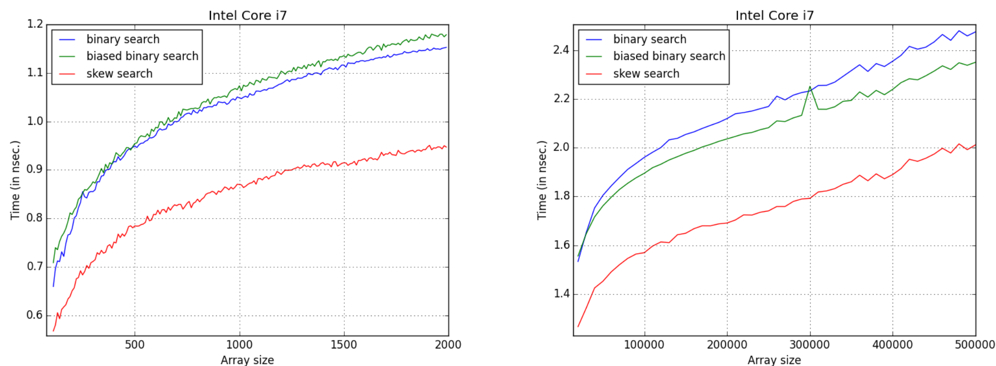


Figure 8 Execution time of the three searching algorithms of Figure 7 for small-size arrays (that fit in the first-level cache) and medium-size arrays (that fit in the last-level cache).

machine[2] in order to mostly measure the effects of branch prediction. The results are depicted in Figure 8: one can see that, for medium-size arrays, SkewSearch is up to 23% faster than the classical binary search (the programs were 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.

5.3 Local Predictors Analysis

As in Section 4, we aim at using the Ergodic Theorem (page 7) to obtain a good asymptotic estimate of the number of mispredictions. We therefore need to compute the expected number of times each given conditional is performed, in our different algorithms. We consider that

[9] Benchmark using jmh: `http://openjdk.java.net/projects/code-tools/jmh/`. Our algorithms are compared to `Arrays.binarySearch(double[] a, double key)`.

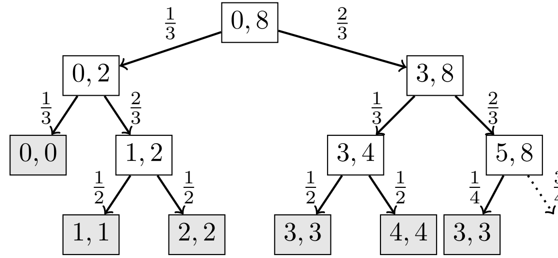


Figure 9 The decomposition tree of BiasedBinarySearch for $n = 8$.

each possible output is equally likely (*i.e.* the uniform distribution on $\{0, \ldots, n\}$).

A first order estimation of the expected number of times a given conditional is executed can be obtained using the following version of Roura's Master Theorem [12], which has been simplified for our specific case:[10]

▶ **Theorem 4** (Master Theorem). *Let $k \geq 1$, and $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$ be positive real numbers such that $\sum_{i=1}^{k} a_i = 1$. For every $i \in \{1, \ldots, k\}$, 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 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 4 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 transform the predictor into a Markov chain as we did in Section 4, because the probabilities $\frac{a_n}{n+1}$ and $\frac{b_n}{n+1}$ are not fixed anymore (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 yield a good approximation of the number of mispredictions with Theorem 2.

A convenient way to prove this formally is to introduce the *decomposition tree* $\mathcal{T}$ associated with the search algorithms, which is defined as follows. 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 in Figure 9.

By construction, following a path from the root to a leaf, by choosing between left and right according to the edge probability is exactly the same as choosing an integer uniformly

[10] For more general statements, we refer the reader to the seminal work of Roura [12].

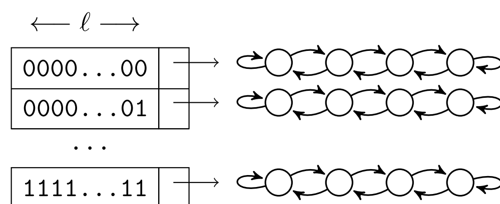


Figure 10 A fully global predictor scheme: The history table of size 2^ℓ keeps track of the outcomes of the last ℓ branches encountered during the execution, the last one corresponding to the rightmost bit. To each sequence of ℓ branches is associated a global 2-bit predictor (shared by all the conditional branches).

at random in $\{0, \ldots, n\}$. Let $u = (u_0, u_1, \ldots)$ be a infinite sequence of elements of $[0, 1]$ taken uniformly at random and independently. To u is associated its path $\mathbf{Path}_n(\mathcal{T}, u)$ in $\mathcal{T}$ where, at step i, we go to the left if u_i is smaller than the left child edge probability and to the right otherwise. Let $L_n(\mathcal{T}, u)$ be the length of $\mathbf{Path}_n(\mathcal{T}, u)$. Let also $\mathbf{Path}_n(\mathcal{I}, u)$ be the path following the values in u in the ideal (infinite) tree $\mathcal{I}$ where we go to the left with probability $\frac{1}{4}$ and to the right with probability $\frac{3}{4}$. Then the following result holds.

▶ **Lemma 5.** *The probability that $\boldsymbol{Path}_n(\mathcal{T}, u)$ and $\boldsymbol{Path}_n(\mathcal{I}, u)$ differ at one of the first $L_n(\mathcal{T}, u) - \sqrt{\log n}$ steps is $\mathcal{O}(\frac{1}{\log n})$.*

Hence, the algorithm BiasedBinarySearch behaves almost like the idealized version, for most of the iterations of its main loop, and we have a sufficiently precise estimation of the error term. This is enough to prove that the idealized version is a correct first order approximation of the number of mispredictions. The same construction can be done for all three algorithms, yielding Theorem 6. For instance, with a 2-bit saturated 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 6.** *Let C_n and M_n be the number of comparisons and mispredictions performed in our model of randomness. The following table give 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.

5.4 Analysis of the Global Predictor for SkewSearch

In this section we intend to give hints about the behavior of a global branching predictor, such as the one depicted in Figure 10 (see also Section 2), for the algorithm SkewSearch. Notice in particular that the predictor of each entry is a 2-bit saturated counter. This is not the only possible choice of a global predictor, but it is simple enough without being trivial. We make the analysis in the idealized framework that resemble 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 taken / not taken produced by the two conditional tests of SkewSearch. We deliberately do not consider the conditional induced by the test within the "while" loop, which would be always not taken in our settings

Figure 11 On the left, the automaton $\mathcal{A}_{\text{if}}$. On the right, the Markovian automaton $\mathcal{M}_{\text{if}}$ of transition probabilities $\mathbb{P}(1 \mid \texttt{main}) = \frac{1}{4}$, $\mathbb{P}(0 \mid \texttt{main}) = \frac{3}{4}$, $\mathbb{P}(0 \mid \texttt{nested}) = \frac{2}{3}$ and $\mathbb{P}(1 \mid \texttt{nested}) = \frac{1}{3}$.

(except for the very last step). Adding it would complicate the model without adding interesting information to the branch predictor.[11] We encode a taken conditional by a 1 and a not taken conditional by a 0. The trace of an execution of the algorithm is thus a nonempty word on the binary alphabet $B = \{0, 1\}$. Because of the way the two conditional tests are nested within the algorithm, we can keep track of the current "if" by the use of the simple deterministic automaton $\mathcal{A}_{\text{if}}$ with two states depicted in Figure 11: `main` stands for the first conditional and `nested` for the second one. In our model, `main` is taken with probability $\frac{1}{4}$ and `nested` with probability $\frac{1}{3}$. As done in Section 4, $\mathcal{A}_{\text{if}}$ can be changed into a Markov chain $\mathcal{M}_{\text{if}}$ using this transition probabilities. A direct computation shows that its stationary vector π_{if} satisfies $\pi_{\text{if}}(\texttt{main}) = \frac{4}{7}$ and $\pi_{\text{if}}(\texttt{nested}) = \frac{3}{7}$.

For the same reason as above, in the global table, we only record the history for the two conditionals `main` and `nested`. Let ℓ denote the history length, that is, the number of bits used in the history table of Figure 10. We assume that ℓ is even. An *history* h is thus seen as a binary word of length ℓ. Let 0^ℓ be the history made of 0's only.

When a conditional is tested at time t, the predictor uses the entry at position h_t to make the prediction, where h_t is the current history. To follow the evolution of the algorithm at time $t + 1$, we therefore only have to keep track of (1) the history table T_t, (2) the current history h_t and (3) which of the two conditionals IF_t is under consideration. Knowing IF_t is required in order to compute the probability that the next outcome is 0 or 1. 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 Theorem 2, 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. In the sequel, we therefore use the particular structure of $\mathcal{M}_{\text{up}}$ to directly compute the typical number of mispredictions.

Let $h \in B^\ell$ be an history that is not equal to 0^ℓ. There is at least one 1 in h. Since reading a 1 always send to state `main` in $\mathcal{A}_{\text{if}}$, we know for sure the conditional IF_t under consideration when an occurrence of h has just happened at time t. Hence, we know the probability to have a 0 or a 1 at time $t + 1$, given that $h_t = h$. As a consequence, each entry of $h \neq 0^\ell$ in the table T behaves like a fixed-probability local 2-bit saturating predictor, with probability $\frac{1}{4}$ (resp. $\frac{1}{3}$) for histories associated to `main` (resp. to `nested`). Therefore, $h = 0^\ell$ concentrates all the differences between the local and the global predictors.

What happens for the entry 0^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 conditional. 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.

[11] Also, most modern architectures have "loop detectors" that are used to identify such conditionals.

▶ **Theorem 7.** *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 6, if we use a local 2-bit predictor for each conditional, 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 0^ℓ. However, if there is a competition between a global predictor and a more accurate local predictor (a 3-bit saturated 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.

6 Conclusion

In this article we propose unbalanced predictor-friendly versions of two very classical algorithms, namely the exponentiation by squaring and the binary search. Using a precise estimation on the expected number of mispredictions, we show that our new algorithms are worth considering when the cost of a comparison is reasonable compared to the cost of a misprediction. This is typically the case for primitive data types.

We believe that these theoretical results, supported by experiments, advocate strongly for considering this particular feature of modern computers in the design and analysis of algorithms: we showed that taking branch prediction into account can yield significant improvements, even on very classical algorithms.

References

1. 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. doi:10.1145/1227161.1370599.
2. Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. On the adaptiveness of quicksort. *ACM Journal of Experimental Algorithmics*, 12, 2008. doi:10.1145/1227161.1402294.
3. Gerth Stølting Brodal and Gabriel Moruz. Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms. In *Algorithms and Data Structures*, volume 3608, pages 385–395. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
4. Gerth Stølting Brodal and Gabriel Moruz. Skewed Binary Search Trees. In *Algorithms – ESA 2006*, volume 4168, pages 708–719. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, third edition, 2009.
6. Amr Elmasry, Jyrki Katajainen, and Max Stenmark. Branch Mispredictions Don't Affect Mergesort. In *Experimental Algorithms*, volume 7276, pages 160–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
7. John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.
8. Kanela Kaligosi and Peter Sanders. How Branch Mispredictions Affect Quicksort. In *Algorithms – ESA 2006*, volume 4168, pages 780–791. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
9. David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2008. URL: http://pages.uoregon.edu/dlevin/MARKOV/markovmixing.pdf.

10 Conrado Martínez, Markus E. Nebel, and Sebastian Wild. Analysis of branch misses in quicksort. In *Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2015, San Diego, CA, USA, January 4, 2015*, pages 114–128, 2015. doi:10.1137/1.9781611973761.11.

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

12 Salvador Roura. Improved master theorems for divide-and-conquer recurrences. *Journal of the ACM*, 48(2):170–205, 2001. doi:10.1145/375827.375837.

13 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. doi:10.1007/978-3-540-30140-0_69.