AC20 Report

AC20 Report

Algorithm complexity analysis

Reference:	AC20-P2016
Version:	2.0
Updated:	2016/06/13
Status:	Public

Authors: Jérôme BOULMIER , Benoît CORTIER

This document describes the AC20 Report project.

TEX is a trademark of the American Mathematical Society.

`tex-upmethodology` is owned by Stéphane Galland, *Arakhnê*.ORG, France.

This document was realised with LATEX and `tex-upmethodology`.

Reference : AC20-P2016

Contents

List of Figures

List of Algorithms

1

Introduction

This report was written as part of the AC20 course at the UTBM. The principle of this course is to acquire scientific knowledge by self-studying. Since we both intend to become computer scientists, we chose to study algorithm complexity, a very important and interesting area of computer science. The idea is to evaluate performances of algorithms.

In order to gather information on the subject, we used several approaches.

Firstly, we followed courses by using two differents MOOCs[1] about algorithms. One from Princeton university with lectures from [Sedgewick and Wayne, 2016] and one from Stanford university with lectures from [Roughgarden, 2016].

Then, we used scientific books and articles to go into specific algorithms or theorems in depth.

We also had the chance to help with actual research by designing an algorithm and program to exploit geometric data. It is an interesting situation where it is not possible to find pre-existing work to start with.

Chapter 2 exposes the fundamentals of the algorithm complexity study with the first section dedicated to theoretical aspects and then by examining two classical sorting algorithms: *selection sort* and *insertion sort*. Chapter 3 introduces and analyses a fast recursive algorithm based on merges called the *merge sort*. Next, Chapter 4 presents and proves a famous theorem called the *Master Theorem* which is used to quickly analyse a certain type of recursive algorithms so-called divide-and-conquer algorithms. Finally, Chapter 5 is dedicated to an application of what we learned to the actual geometrical problem as part of research at the UTBM.

[1] Acronym for *Massive Open Online Course*, an online course aimed at unlimited participation and open access via the web.

2

Elementary sorting algorithms

This chapter will introduce the fundamental approaches to analyse algorithms and use two fundamental sorting algorithms as examples: the selection sort and the insertion sort.

2.1/ Fundamentals

There are mainly two approaches to analyse an algorithm:

- The *empirical approach* where observations are made to predict.
- The *mathematical approach* where results are formally proved.

The latter is both more technical and more rigorous. Often, both of these approaches are used jointly.

2.1.1/ Empirical approach

One method to analyse algorithms is the so-called "Scientific method". This is the very same approach that scientists use to analyse the natural world. And this is effective for studying the running time of programs as well. The idea is:

- *Observe* and take measurements.
- *Hypothesize* a model that seems consistent with the observations (linear, quadratic, exponential, ...).
- *Predict* using the hypothesis.
- *Verify* the predictions with new observations.
- *Validate* by repeating until the hypothesis and observations mutually agree.

[Sedgewick and Wayne, 2011]

One key point is that designed experiments must be *reproducible* to allow other people to convince themselves. [Sedgewick and Wayne, 2011]

Observations: To make observations, one can simply run the program. Indeed, each run takes time to be performed. Thus it is possible to measure this time to answer the question "What is the running time of my program" which is a core question in algorithm's complexity study. This is an empirical analysis of the running time.

The size of the problem is often the main factor in the running time. Many algorithms are insensitive to data (running time doesn't depend on the input data itself but on the quantity of the given data). Yet, some algorithms are sensitive to data. In such case, it is required to go deeper into the analysis.

Here are the main steps to follow:

- measure the running time for different problem sizes (N = 1000, N = 2000, N = 4000, ...).
- make a standard plot.
- make a log-log plot. $\lg$ [1]$(T(N))$ on the ordinate and $\lg(N)$ on the abscissa. If the line is straight, the complexity is a power law and the slope is the key. With $slope = b$, $T(N) = aN^b$. Indeed, $\lg(T(N)) = b\lg(N) + c = \lg(N^b) + c \Leftrightarrow T(N) = 2^{\lg(N^b)+c} = 2^c N^b$ and $a = 2^c$.

Another way to deal with power laws is to take the ratio of the running times for N and $2N$ and so forth. Indeed, $b = \lg(ratio)$.

There are system independent and system dependent effects to take into account, too. The key effects are independent of the computer used:

- Algorithm
- Input data

Those determine the exponent in the power law.

Then, there are a lot of system dependent effects:

- the *Hardware*: what is the CPU, the memory, etc.
- the *Software*: what is the compiler, the interpreter, etc.
- the operating system
- ...

All those effects, independent and dependent ones, *determine the constant in the power law*.

2.1.2/ Mathematical approach

Mathematical models: D. E. Knuth postulated that it is possible, in principle, to build a mathematical model to describe the running time of any program. The total running time of a program is determined by two primary factors:

- Each instruction execution *cost*.
- Each instruction execution *frequency*.

The instruction execution cost depends on the computer (system dependent) and the instruction execution frequency depends on the algorithm and the inputed data. If one knows both for every instruction in the program, it is possible, for all instructions, to sum the cost multiplied by the frequency to get the *running time*.

Determining the frequency of certain instructions may require a more or less high-level reasoning and sometimes a probabilistic analysis.

Analysing instructions frequency can lead to expressions such as $\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}$. But lower order terms quickly become insignificant in comparison to the leading term. For instance, with $N = 1000$, $\frac{N^3}{6} \approx 1.66 \times 10^8$ and $-\frac{N^2}{2} + \frac{N}{3} \approx -5.00 \times 10^5$. 1.66×10^8 is around 322 times greater than 5.00×10^5. That's why the *tilde approximation* is used. The mathematical *tilde notation* ($\sim$) can be used, and is often used, to simplify the mathematical expression by ignoring low-order terms which are negligible. Thus, $\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}$ can be simplifieds $\sim \frac{N^3}{6}$. Indeed,

$$\lim_{n \to +\infty} \frac{\frac{N^3}{6}}{\frac{N^3}{6} - \frac{N^2}{2} + \frac{N}{3}} = 1.$$

[1] lg stands for log-2 (the binary logarithm).

Another important simplification is to *only count operations that are most expensive* in term of individual cost or frequency of occurrence. The typical behavior for many algorithm is to have a running time depending only *on a small subset of instructions*, especially for N getting larger.

Estimating a discrete sum is a critical way to determine the mathematical expression of loops without having to run the program.

Algorithm 2.1 Example algorithm for estimating a discrete sum

```
for i from 0 to N - 1 do
    for j from i + 1 to N - 1 do
        // statements
    end for
end for
```

Algorithm 2.1 is a classical situation. While i grows in the main loop, the inner loop iterates less. Basically, the statements in the inner loop are first executed N times, then $N-1$ times and so forth until they are executed only *once*. Overall, instructions in the inner loop are executed $1+2+\ldots+N-1+N=\sum_1^N i$ times. Finally, one can approximate the discrete sum:

$$\sum_1^N i = \frac{N(N+1)}{2} \sim \frac{N^2}{2}.$$

Remark: In a more complex situation, it may be easier to use an integral approximation.

Algorithm 2.2 Another example algorithm for estimating a discrete sum

```
for i from 0 to N - 1 do
    for j from i + 1 to N - 1 do
        for k from i + 1 to N - 1 do
            for l from i + 1 to N - 1 do
                // statements
            end for
        end for
    end for
end for
```

Algorithm 2.2 is another, more complex situation. Overall, instructions in the inner loop are executed $1+2^3+\ldots+(N-1)^3+N^3=\sum_1^N i^3$ times. One can approximate the discrete sum using an integral and then approximate the integral:

$$\sum_1^N i^3 \sim \int_1^N x^3\,\mathrm{d}x = \left[\frac{x^4}{4}\right]_1^N = \frac{N^4}{4} - \frac{1}{4} \sim \frac{N^4}{4}.$$

Common order-of-growth classifications: Often, the order of growth of the cost is one of just a few functions of N. The following list contains some common order-of-growth functions:

- *Constant* (1): when executing a fixed number of operations.
- *Logarithmic* $(\log N)$: when the problem size is divided in half like in a binary search.
- *Linear* (N): for a single *for* loop.
- *Linearithmic* $(N \log N)$: often when using a divide-and-conquer method.
- *Quadratic* (N^2): for two nested *for* loops.

- *Cubic (N^3)*: for three nested *for* loops.
- *Exponential (b^N)*: for an exhaustive search like checking all subsets in order to find all combinations of elements.

Commonly-used notations: To describe the order-of-growth of algorithms there are some important notations: Θ *(Big theta)*, O *(Big Oh)* and Ω *(Big Omega)*.

From a mathematical point of view, the running time can be expressed as a function from the set of integers (the input size) to the set of reals: $f : \mathbb{N} \to \mathbb{R}^{+*}$. Let g be another complexity function.

The following equivalences exist:

- $f \in O(g) \Leftrightarrow \exists\, c \in \mathbb{R}^{+*},\ f \leq c \times g$
- $f \in \Omega(g) \Leftrightarrow \exists\, c \in \mathbb{R}^{+*},\ f \geq c \times g$
- $f \in \Theta(g) \Leftrightarrow f \in O(g)$ and $f \in \Omega(g) \Leftrightarrow \exists\, (c,k) \in \mathbb{R}^{+*^2},\ c \times g \leq f \leq k \times g$.

For instance, with $f(N) = 5N^2 + 22N \log N + 3N$ one can say that $f \in \Theta(N^2)$, $f \in O(N^2)$, $f \in O(N^3)$, $f \in \Omega(N^2)$, $f \in \Omega(N)$ and so forth.

Table 2.1 summarises these different notations.

Notation	*Provides*	*Example*	*Shorthand for*	*Used to*
Big Theta	Asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}N^2$ $10N^2$ $5N^2+22N\log N+3N$ $\vdots$	Classify algorithms
Big Oh	Big Theta and smaller	$O(N^2)$	$10N^2$ $100N$ $22N \log N + 3N$ $\vdots$	Develop upper bounds
Big Omega	Big Theta and larger	$\Omega(N^2)$	$\frac{1}{2}N^2$ N^5 $N^3 + 22N \log N + 3N$ $\vdots$	Develop lower bounds

[Sedgewick and Wayne, 2011]

Table 2.1: Commonly used order-of-growth notations.

Running time: To evaluate the algorithm *performance*, one may count the basic operations (compares, exchanges, array accesses [read and write]). [Sedgewick and Wayne, 2011]

Extra memory: The amount of extra memory used by a sorting algorithm can be an important factor given the device. There are algorithms that perform *in-place* sorting. They do not require any extra memory that depends on the problem size. And there are algorithms that need extra memory to hold a copy of the array. [Sedgewick and Wayne, 2011]

Types of analyses: Since the input can cause the performances to vary very widely, there are several types of algorithm analyses:

- *Best case:* determined by the easiest input. The algorithm cannot be faster than the best case. It provides a goal for inputs.
- *Worst case:* determined by the most difficult input. The algorithm is guaranted to not be slower than the worst case. It provides a guarantee for inputs.
- *Average case:* expected for random input. It provides an estimate of average performance.

The running time has to be somewhere between the best case (lower bound) and the worst case (upper bound). Since actual data might differ from the models used, it's ideal to design an algorithm with the faster worst case possible to get an efficient upper bound guaranteed, an algorithm that always runs quickly. Otherwise, one can try to randomize the inputs to try to provide a probabilistic guarantee.

The next section shows how to determine the expression of the running time of the selection sort.

2.2/ SELECTION SORT

The "selection sort" is one of the simplest sorting algorithms. A total order relation between items is required. The same goes for almost every sorting algorithm. There are only a few exceptions like the "radix sort" which is not covered in this report. The principle is very simple: firstly, find the smallest item and exchange it with the first entry. Then, find the next smallest item and exchange it with the second entry and so forth.

Algorithm 2.3 Selection sort algorithm

```
N ← length(a)
for i from 0 to N - 1 do
    iMin ← i
    for j from i + 1 to N - 1 do
        if a_j < a_iMin then
            iMin ← j
        end if
    end for
    swap(a, i, iMin)
end for
```

- **swap:** a function that swaps two elements in the given array.

In algorithm 2.3, the inner loop performs one compare by item to check if the item is smaller than the previous smallest item found so far. Then, the exchange operation puts the smallest item found into its final position so the number of exchanges is N. Thus, the running time is dominated by the number of comparisons which is the most used basic operation in this algorithm.

It leads to the following proposition:

Proposition: Selection sort needs $\frac{N^2}{2}$ compares and N exchanges in order to sort an array of size N. Thus, the selection sort complexity is $\Theta(N^2)$.

Proof: By examining the algorithm one can tell that for each i from 0 to $N-1$, there is one exchange and $N-1-i$ compares. Thus the total is: N exchanges and $\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2} \sim \frac{N^2}{2}$ compares.

One should be able to convince oneself in a more graphical way by examining the trace (an N-by-N table) in Figure 2.1. In the trace, an unshaded letter is an item that is compared to the smallest one. Half of the items in the table are unshaded. A diagonal is clearly formed. Each item on this diagonal corresponds to an exchange.

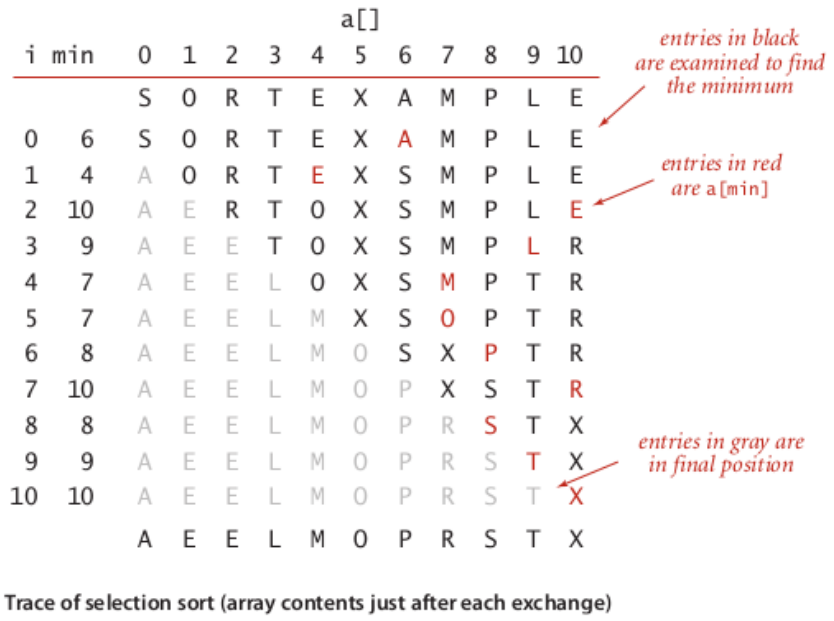


Figure 2.1: Selection sort trace.

[Sedgewick and Wayne, 2002-2014b]

Two properties:

- The *running time is insensitive to input*. Indeed, the process to find the smallest item always requires one pass through. It means that the running time is always the same even for a randomly-sorted array, a partially-sorted array and even a totally-sorted one.

- The *data movement is minimal* since the number of array accesses is a linear function of the array size. This is a rare property, every other sorting algorithm has a linearithmic or quadratic function for data movement.

This sorting algorithm is interesting when elements are easily comparable but slow to exchange. However, nowadays elements are mostly managed using references. And exchanging a reference is as simple as exchanging an integer. Both uses 32 bits or 64 bits given the architecture, but it may vary according to the programming language, interpreter, etc.

2.3/ Insertion sort

The ”insertion sort” is another simple sorting algorithm. The algorithm starts by checking the second element. If this element is smaller than the first one, the two are exchanged. Then, the next element is checked and moved to the left while there are greater elements on the left and so forth until the end of the array.

Algorithm 2.4 Insertion sort algorithm

```
N ← length(a)
for i from 1 to N - 1 do
    j ← i
    while j ≥ 1 and a_j < a_{j-1} do
        swap(a, j, j - 1)
        j ← j − 1
    end while
end for
```

- **swap:** a function that swaps two elements in the given array.

As one can see in algorithm 2.4, unlike the selection sort, the running time depends on the initial order of the elements in the array. It is very fast with inputs already in order or partially ordered and slow with a randomly ordered or reverse ordered input. It means that there are several cases: best, worst and average.

Typical examples of partially-sorted arrays are arrays where:

- each entry is not far from its final position.
- only a few entries are not in place.
- the array is large and a small array has been appended.

Proposition: To sort an array of size N, insertion sort needs $\frac{N^2}{4}$ compares and $\frac{N^2}{4}$ exchanges on the average, $\frac{N^2}{2}$ compares and $\frac{N^2}{2}$ exchages in the worst case and $N-1$ comparisons and 0 exchanges in the best case. Thus, selection sort complexity is $\Theta(N^2)$ in the worst and average cases and $\Theta(N)$ in the best case.

Proof: The number of compares and exchanges is easy to visualize in the trace in Figure 2.2. One simply counts all entries below the diagonal for the worst case, none for the best case and about half of them in the average case.

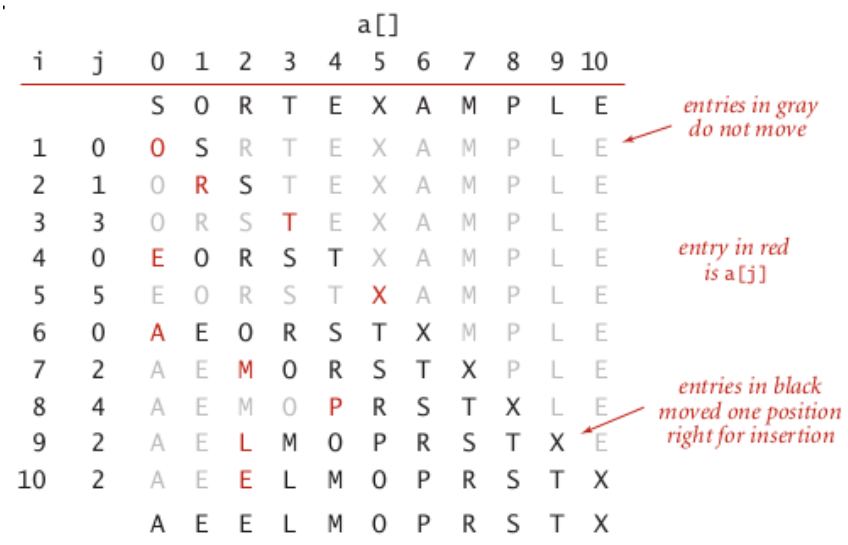


Figure 2.2: Insertion sort trace.

[Sedgewick and Wayne, 2002-2014a]

This sorting algorithm is excellent for partially-sorted arrays, but does far more exchanges than the selection sort in average and worst cases. Still, it is a fine method for tiny arrays. This algorithm is interesting since partially-sorted arrays frequently arise in practice and can even be used to speed up other, more efficient algorithms at some stages.

3

Merge sort

The *merge sort* is a fast algorithm based on a simple operation known as *merging*.

3.1/ Principle

The principle of *merging* is to combine two ordered arrays to make one larger ordered array. It directly leads to a simple recursive sort algorithm known as *mergesort*: to sort an array, divide it into two halves, sort the two halves (recursively), and then merge the results.

To begin, let's consider the *merge* operation which is the core of *mergesort*.

Merge operation: A straightforward approach would be to merge two disjoint ordered arrays into a third array. It requires to create a new output array of the requisite size each time (the third array), then choose successively the smallest remaining item from the two input arrays to be the next item added to the output array. But when mergesorting a large array, a huge number of merges are performed and the cost of creating a new array to hold the output every time is very consequent. So, it is much more desirable to have an in-place algorithm[1] to avoid using a significant amount of other extra space. It is already known that solutions that are known are quite complicated, especially by comparison to alternatives that use extra space. In this report, we're not going to cover the complicated solutions, but the *abstraction* of an in-place merge is still useful. The signature of an in-place merge would be *merge(Array, lower Index, middle Index, higher Index)* which is what we use in algorithm 3.1. It specifies a merge method that merges the subarray *a[lo..mid]* with another subarray *a[mid+1..hi]* and leaves the result in *a[lo..hi]*. However, this algorithm works by first copying the subarrays in an auxiliary array and then merging back to the original one. The auxiliary array is be the same for every merge operation: only one is created. The auxiliary array was added to the initial function signature to show that.

[1] An in-place algorithm works directly on the array to sort without creating a new one.

Algorithm 3.1 Merge algorithm

Require: a an array of size N.
Require: hi an integer such as $hi < N$, the higher bound index.
Require: mid an integer such as $mid < hi$, the middle bound index.
Require: lo an integer such as $lo < mid$, the lower bound index.
Require: aux an auxiliary array of size N.

```
function MERGE(a, lo, mid, hi, aux): ∅
    i ← lo
    j ← mid + 1
    for k from lo to hi do                    ▷ copy a[lo..hi] to aux[lo..hi]
        aux_k ← a_k
    end for
                                              ▷ merge back to a[lo..hi]
    while i ≤ mid and j ≤ hi do
        if aux_j < aux_i then
            a_k ← aux_j
            j ← j + 1
        else
            a_k ← aux_i
            i ← i + 1
        end if
    end while
    while i ≤ mid do
        a_k ← aux_i
        i ← i + 1
    end while
    while j ≤ hi do
        a_k ← aux_j
        j ← j + 1
    end while
end function
```

Next we're going to cover a recursive mergesort algorithm that use our merge function.

Top-down mergesort: The top-down mergesort is one of the best-known examples of the utility of the divide-and-conquer paradigm[2]. The idea is that if the algorithm sorts the two subarrays, then it sorts the whole array by merging together the subarrays. Algorithm 3.2 features two functions: *sort(Array)* and *sort(Array, lower Index, higher Index, auxiliary Array)*. The second one is designed to be called by the first one with the right initial parameters, then the function calls itself recursively until the trivial case is reached. The trivial case is when both selected subarrays contain at most 1 element. Merging two subarrays of size 1 result in an ordered array of size 2. And two sorted subarrays of size 2 result in an ordered array of size 4 and so forth until all the merges have been applied to reach a sorted array of size N.

[2] A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. Then to give a solution to the original problem, one can simply combine the sub-problems' solutions.

Algorithm 3.2 Top-down merge sort algorithm

Require: a an array of size N.
Require: hi an integer such as $hi < N$, the higher bound index.
Require: lo an integer such as $lo < mid$, the lower bound index.

```
function SORT(a) : ∅
    aux ← createArray(size(a))
    sort(a, 0, N − 1, aux)
end function
function SORT(a, lo, hi, aux) : ∅                    ▷ Sort a[lo..hi]
    if hi > lo then                                  ▷ Trivial case: hi ≤ lo, nothing to do
        mid ← lo + (hi − lo)/2
        sort(a, lo, mid, aux)                        ▷ Sort left half
        sort(a, mid + 1, hi, aux)                    ▷ Sort right half
        merge(a, lo, mid, hi, aux)                   ▷ Merge results
    end if
end function
```

The main advantages of the mergesort are:

- It is a *stable*[3] sorting algorithm.
- Its complexity function $f \in \Theta(n \log n)$.

That being said, the major drawback is that it uses extra space proportional to N and getting ride of this issue is not that easy (for this reason, shellsort or quicksort may used instead; it depends on the situation).

Another possible algorithm is the *bottom-up mergesort*. This is a way to design a mergesort algorithm so that all the merges of tiny subarrays are done on one pass, then a second pass to merge the resulting sorted subarrays in pairs is done, and so forth until a merge that includes the whole array is done. This version is not recursive, while still featuring the same properties.

Some optimizations are:

- *The use of insertion sort for small subarrays.* Most recursive algorithms can be improved by handling small cases differently. Indeed, recursion guarantees that small cases will often arise. Insertion sort is simple and, therefore, likely to be faster than mergesort for tiny subarrays. Thus switching to insertion sort for subarrays of length 15 or less will improve the running time.
- *Test whether the array is already in order.* The running time can be reduced to be linear for arrays already in order by adding a test to skip the call to the *merge* function if a_{mid} is less than or equal to a_{mid+1}.
- *Eliminate the copy to the auxiliary array.* It is indeed possible to eleminate the time (not the space) taken to copy the auxiliary array used for merging. The idea is to use two calls of the sort method in order to switch the roles of the arrays. One call would be taking its input from the given array and putting the sorted output in the auxiliary array, and the other would be taking its input from the auxiliary array and putting the sorted output in the original array.

3.2/ Analysis

Merge operation: Algorithm 3.1 merges by first copying into the auxiliary array. Let $n = hi - lo + 1$. This copy is made using a single *for* loop that runs n times and writes n times on the auxiliary array. Then

[3] A sorting method is stable if it preserves the relative order of equal keys in the array.

the actual merge operation runs n times too (by summing the 3 loops number of iterations), but the number of compares depends on the input array. There are at most n compares (only the first loop runs in the worst case). There are at least $\frac{n}{2}$ compares (all the elements of one of the subarrays are greater than the other). In every case it writes n times on the input array. That gives us n compares and $2n$ array writes for the worst case, and $\frac{n}{2}$ compares and $2n$ array writes for the best case. Thus the merge operation complexity is $\Theta(n)$.

Top-down mergesort: Let $C(N)$ be the number of compares needed to sort an array of length N.

We have $C(0) = C(1) = 0$ and for $N > 0$ one can write a recurrence relationship that directly translates the recursive *sort* method to establish an upper bound: $C(N) \leq C(\lfloor \frac{N}{2} \rfloor) + C(\lceil \frac{N}{2} \rceil) + N$. The first term is the number of compares to sort the left half of the array. The second term is the number of compares to sort the right half. The third term is the number of compares to merge in the worst case (see the the analysis of the merge operation above).

In the same way, the lower bound is: $C(N) \geq C(\lfloor \frac{N}{2} \rfloor) + C(\lceil \frac{N}{2} \rceil) + \frac{N}{2}$. This time, the third term is the number of compares to merge in the best case (see the the analysis of the merge operation above).

The exact solution to the recurrence is derived from when the equality holds and N is a power of 2. So, let $N = 2^n$. First, there is $\lfloor \frac{N}{2} \rfloor = \lceil \frac{N}{2} \rceil = 2^{n-1}$, so:

$$C(2^n) = 2C(n^{n-1}) + 2^n.$$

Dividing both sides by 2^n gives

$$\frac{C(2^n)}{2^n} = \frac{C(n^{n-1})}{2^{n-1}} + 1.$$

Applying the same equation to the first term on the right-hand side leads to:

$$\frac{C(2^n)}{2^n} = \frac{C(n^{n-2})}{2^{n-2}} + 1 + 1.$$

Repeating the previous step $n - 1$ additional times directly gives:

$$\frac{C(2^n)}{2^n} = \frac{C(n^0)}{2^0} + \sum_{k=0}^{n} 1 = n,$$

which after multiplying by 2^n gives:

$$C(2^n) = 2^n n.$$

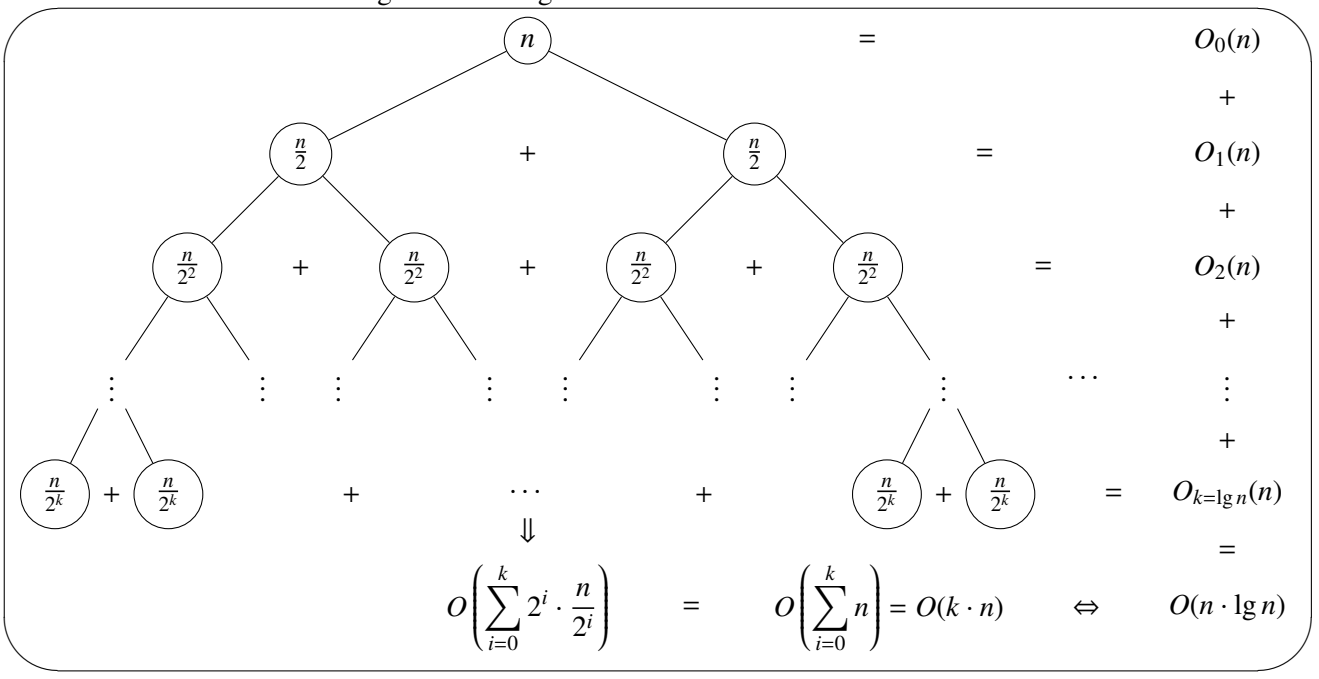
Finally, by using $N = 2^n \Leftrightarrow \log_2 N = n \log_2 2 = n$, we obtain:

$$C(N) = Nlog_2(N).$$

Exact solutions for general N are more complicated, but the strategy remains the same. This proof made no assumption on the input, thus it is valid no matter what the input is.

Another way to understand this proof is to examine the recursion tree in Figure 3.1. In this tree, each node is a subarray for which the *sort* function does a *merge operation*. The size of the array is n. The tree has precisely $k = \log_2 n$ levels. For i from 0 to $k - 1$, the ith level from the top depicts 2^i subarrays, each of length $2^{k-i} = \frac{2^k}{2^i} = \frac{n}{2^i}$, each of which thus requires at most $\frac{n}{2^i}$ compares for the merge. Thus we have $2^i \frac{n}{2^i} = n$ total cost for each of the k levels, for a total of $n \log_2 n$.

Figure 3.1: Merge sort recursion tree.



4

Master Theorem

The merge sort algorithm is an example of algorithm using a *divide-and-conquer* approach. The analysis of a divide and conquer type algorithm leads to a recursive relation for the study of the complexity function. In this chapter we prove a general theorem useful to deal with such recursive relations.

4.1/ Theorem

The Master Theorem assumes that the complexity function $T(n)$ of an algorithm satisfies the following recurrence:

$$T(n) = aT(\frac{n}{b}) + f(n),$$

where

- n is the size of the problem,
- $a \in \mathbb{R}$, $a \geq 1$ is the number of subproblems in the recursion,
- $\frac{n}{b}$ with $b \in \mathbb{R}$, $b > 1$, is the size of each subproblem (assuming that all subproblems are of the same size), and
- $f(n)$ is the cost of the work done outside the recursive calls.

Then:

1. If $f(n) \in O(n^c)$ where $c < \log_b a$ then $T(n) \in \Theta(n^{\log_b a})$.
2. If $f(n) \in \Theta(n^c \log^k n)$ where $c = \log_b a$ and $k \geq 0$ then $T(n) \in \Theta(n^c \log^{k+1} n)$.
3. If $f(n) \in \Omega(n^c)$ where $c > \log_b a$ and $af(\frac{n}{b}) \leq kf(n)$ with $k \in \mathbb{R}^+$, $k < 1$ and sufficiently large n then $T(n) \in \Theta(f(n))$.

Examples:

1. If an algorithm creates 8 subproblems, divides the problem size by 2 at each iteration and has $f(n) = 10n^2$, the expression is:

   $$T(n) = 8T(\frac{n}{2}) + 10n^2.$$

   Thus: $a = 8$, $b = 2$ and $c = 2$. Since $\log_b a = 3$, $c < \log_b a$ (1st case). The Master Theorem says that $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^3)$ (indeed, assuming $T(1) = 1$, the exact recurrence relation is $T(n) = 11n^3 - 10n^2$).

2. If an algorithm creates 2 subproblems, divides the problem size by 2 at each iteration and has $f(n) = 2n$, the expression is:

   $$T(n) = 2T(\frac{n}{2}) + 2n.$$

Thus: $a = 2$, $b = 2$ and $c = 1$. Since $\log_b a = 1$, $c = \log_b a$ and $f(n) = 2n^c \log^k n = 2n$, there is also $k = 0$ (2nd case). The Master Theorem says that $T(n) \in \Theta(n^c \log^{k+1} n) = \Theta(n \log n)$ (again, assuming $T(1) = 1$, the exact recurrence relation is $T(n) = 2n \log_2(n) + n$).

3. If an algorithm creates 2 subproblems, divides the problem size by 2 at each iteration and has $f(n) = 2n^2$, the expression is:

$$T(n) = 2T(\frac{n}{2}) + 2n^2.$$

Thus: $a = 2$, $b = 2$ and $c = 2$. Since $\log_b a = 1$, $c > \log_b a$ and $af(\frac{n}{b}) = n^2 \leq 2kn^2$) by choosing $k = \frac{1}{2}$ (or greater) (3rd case). The Master Theorem says that $T(n) \in \Theta(f(n)) = \Theta(n^2)$ (assuming $T(1) = 1$, the exact recurrence relation is $T(n) = 4n^2 - 3n$).

The next section proves these facts.

4.2/ Proof

Let's consider the recurrence relation:

$$T(n) = aT(\frac{n}{b}) + f(n) \ (*).$$

Graphical proof: The solution can be instinctively found using a graphical approach. Figure 4.1 is a tree showing how a typical divide-and-conquer algorithm divides itself. It shows the case with $a = 2$.

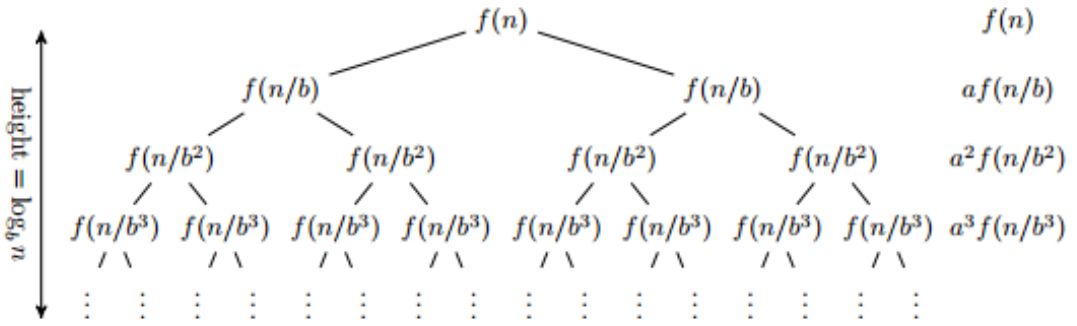


Figure 4.1: Master Theorem graphical proof.

It leads to the following complexity function:

$$T(n) = \overbrace{\sum_{i=0}^{log_b(n)-1} a^i f(\frac{n}{b^i})}^{\text{depends on } f} + \underbrace{a^{log_b(n)} f(1)}_{O(n^{log_b(n)}) \text{ depends on tree depth}}.$$

The first part is obtained by counting the number of nodes at each level (there are a^i nodes at level i). The last term $O(n^{\log_b a})$ is the sum across the leaves, which is $a^{\log_b n} f(1) = n^{\log_b a} f(1)$.

Algebraic proof: First, solve $(*)$. Let $k = log_b(n)$. In this proof we consider $n = b^k$, but it's generalizable. Solve recurrence by $n = b^k$.

$$\begin{aligned} T_k = T(n) &\Rightarrow T_k = aT_{k-1} + f(b^k) \text{ using } (*), \\ &\Rightarrow aT_{k-1} = a^2T_{k-2} + af(b^{k-1}), \\ &\vdots \\ &\Rightarrow a^{k-1}T_1 = a^kT_0 + a^{k-1}f(b). \end{aligned} \tag{4.1}$$

Finally, by summing up all the equations:

$$T_k = a^kT_0 + \sum_{i=0}^{k-1} a^i f(b^{k-1}) \Rightarrow T(n) = \overbrace{\sum_{i=0}^{log_b(n)-1} a^i f(\frac{n}{b^i})}^{\text{operations}} + \underbrace{O(n^{log_b(n)})}_{\text{tree depth complexity}}.$$

1st case: $f \in O(n^{log_b(a)-\varepsilon})$ (f complexity < tree depth complexity)

$$\begin{aligned} T(n) &= \overbrace{\sum_{i=0}^{log_b(n)-1} a^i f(\frac{n}{b^i})}^{log_b(n)} + O(n^{log_b(a)}), \\ &\leq \sum_{i=0}^{log_b(n)} a^i (\frac{n}{b^i})^{log_b(a)-\varepsilon} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \sum_{i=0}^{log_b(a)-1} a^i b^{-i\,log_b(a)} b^{i\varepsilon} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \sum_{i=0}^{log_b(a)-1} a^i a^{-i\,log_b(b)} b^{i\varepsilon} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \sum_{i=0}^{log_b(a)-1} a^i a^{-i} b^{i\varepsilon} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \sum_{i=0}^{log_b(a)-1} b^{i\varepsilon} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \frac{b^{\varepsilon\, log_b(n)} - 1}{b^\varepsilon - 1} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \frac{n^\varepsilon - 1}{b^\varepsilon - 1} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \frac{n^\varepsilon - 1}{b^\varepsilon - 1} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)-\varepsilon} \frac{n^\varepsilon}{b^\varepsilon - 1} + O(n^{log_b(a)}), \\ &\leq n^{log_b(a)} \frac{1}{b^\varepsilon - 1} + O(n^{log_b(a)}), \\ &\leq O(n^{log_b(a)}). \end{aligned} \tag{4.2}$$

So $T \in \Theta(n^{\log_b(a)})$.

2nd case: $f \in \Theta(n^{log_b(a)})$.

Let $(c, d) \in \mathbb{R}^{+*^2}$, with $c < d$. Then,

$$\begin{aligned}
T(n) &= \sum_{i=0}^{log_b(n)} a^i f(\frac{n}{b^i}) + O(n^{log_b(a)}), \\
&\leq \sum_{i=0}^{log_b(n)} a^i \times d \times (\frac{n}{b^i})^{log_b(n)} + O(n^{log_b(a)}), \\
&\leq d \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} a^i \times b^{-i \times log_b(a)} + O(n^{log_b(a)}), \\
&\leq d \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} a^i \times a^{-i \times log_b(b)} + O(n^{log_b(a)}), \\
&\leq d \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} a^i \times a^{-i} + O(n^{log_b(a)}), \\
&\leq d \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} 1 + O(n^{log_b(a)}), \\
&\leq d \times n^{log_b(a)} \times (log_b(a) + 1) + O(n^{log_b(a)}).
\end{aligned} \tag{4.3}$$

Moreover,

$$\begin{aligned}
T(n) &= \sum_{i=0}^{log_b(n)} a^i f(\frac{n}{b^i}) + O(n^{log_b(a)}), \\
&\geq \sum_{i=0}^{log_b(n)} a^i \times c \times (\frac{n}{b^i})^{log_b(n)} + O(n^{log_b(a)}), \\
&\geq c \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} a^i \times b^{-i \times log_b(a)} + O(n^{log_b(a)}), \\
&\geq c \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} a^i \times a^{-i \times log_b(b)} + O(n^{log_b(a)}), \\
&\geq c \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} a^i \times a^{-i} + O(n^{log_b(a)}), \\
&\geq c \times n^{log_b(a)} \times \sum_{i=0}^{log_b(a)} 1 + O(n^{log_b(a)}), \\
&\geq c \times n^{log_b(a)} \times (log_b(a) + 1) + O(n^{log_b(a)}).
\end{aligned} \tag{4.4}$$

So we have, $T(n) \in \Theta(n^{log_b(a)} \times log_b(a))$.

3rd case: $f \in \Omega(n^{log_b(a)+\varepsilon})$.

$$\begin{aligned}
a \times f(\frac{n}{b}) &\geq a \times (\frac{n}{b})^{log_b(a)+\varepsilon}, \\
&\geq a \times n^{log_b(a)+\varepsilon} \times b^{-log_b(a)-\varepsilon}, \\
&\geq a \times n^{log_b(a)+\varepsilon} \times a^{-log_b(b)} \times b^{-\varepsilon}, \\
&\geq n^{log_b(a)+\varepsilon} \times b^{-\varepsilon}, \\
&\geq f(n) \times b^{-\varepsilon}.
\end{aligned} \tag{4.5}$$

By using 4.5, we get

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{log_b(n)} a^i f(\frac{n}{b^i}) + O(n^{log_b(a)}),\\
&\geq \sum_{i=0}^{log_b(n)} a^i (\frac{n}{b^i})^{log_b(a)+\varepsilon} + O(n^{log_b(a)}),\\
&\geq n^{log_b(a)+\varepsilon} \sum_{i=0}^{log_b(n)} a^i \times b^{-i(log_b(a)+\varepsilon)} + O(n^{log_b(a)}),\\
&\geq n^{log_b(a)+\varepsilon} \sum_{i=0}^{log_b(n)} a^i \times a^{-i(log_b(b)} \times b^{-i\varepsilon} + O(n^{log_b(a)}),\\
&\geq n^{log_b(a)+\varepsilon} \sum_{i=0}^{log_b(n)} b^{-i\varepsilon} + O(n^{log_b(a)}),\\
&\geq n^{log_b(a)+\varepsilon} \sum_{i=0}^{log_b(n)} (\frac{1}{b^\varepsilon})^i + O(n^{log_b(a)}),\\
&\geq n^{log_b(a)+\varepsilon} \sum_{i=0}^{\infty} (\frac{1}{b^\varepsilon})^i + O(n^{log_b(a)}),\\
&\geq n^{log_b(a)+\varepsilon} \frac{1}{1-\frac{1}{b}} + O(n^{log_b(a)}),\\
&\geq f(n) \frac{1}{1-\frac{1}{b}} + O(n^{log_b(a)}),\\
&\geq O(f(n)).
\end{aligned}
\tag{4.6}
$$

So we have, $T(n) \in \Theta(f(n))$.

4.3/ Strassen algorithm

To illustrate the Master theorem, we now introduce a well-known algorithm for matrix multiplication, the Strassen algorithm.

To use the Strassen algorithm, we have to partition the matrix A and the matrix B into four blocks of equal size. If the size of the matrix isn't a power of 2, we fill the missing rows/columns with 0.

$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$, $B = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix}$.

So the resulting C is:

$$
\begin{aligned}
c_{1,1} &= a_{1,1}b_{1,1} + a_{1,2}b_{2,1},\\
c_{1,2} &= a_{1,1}b_{1,1} + a_{1,2}b_{2,1},\\
c_{2,1} &= a_{1,1}b_{1,1} + a_{1,2}b_{2,1},\\
c_{2,2} &= a_{1,1}b_{1,1} + a_{1,2}b_{2,1}.
\end{aligned}
$$

This equation is a basic product of two matrices. But if we use the Strassen algorithm, we can reduce the number of multiplications to 7. We define $M_1, M_2, M_3, M_4, M_5, M_6, M_7 \in \mathbb{R}^{2(n-1)\times 2(n-1)}$.

$$
\begin{aligned}
m_1 &= (a_{1,1} + a_{2,2})(b_{1,1} + b_{2,2}),\\
m_2 &= (a_{2,1} + a_{2,2})b_{1,1},\\
m_3 &= a_{1,1}(b_{1,2} - b_{2,2}),\\
m_4 &= a_{2,2}(b_{2,1} - b_{1,1}),\\
m_5 &= (a_{1,1} + a_{1,2})b_{2,2},\\
m_6 &= (a_{2,1} - a_{1,1})(b_{1,1} + b_{1,2}),\\
m_7 &= (a_{1,2} - a_{2,2})(b_{2,1} + b_{2,2}).
\end{aligned}
$$

And so, we have:

$$
\begin{aligned}
c_{1,1} &= m_1 + m_4 - m_5 + m_7,\\
c_{1,1} &= m_3 + m_5,\\
c_{1,1} &= m_2 + m_4,\\
c_{1,1} &= m_1 - m_2 + m_3 + m_6.
\end{aligned}
$$

And we repeat the process until A and B become ”small” (around 64). After that we use the basic product, which is used because the constant of the Strassen algorithm is so large, that the basic product is faster than the Strassen algorithm.

Algorithm 4.1 Strassen algorithm

```
function STRASSEN(m1 : Matrix, m2 : Matrix, matrixSize : Integer) : Matrix
    if matrixSize < MIN_MATRIX_SIZE then        ▷ Make a simple matrix multipliction using three loops.
    else
        newMatrixSize ← matrixSize / 2
        initialize matrices a11, a12, a21, a22, b11, b12, b21, b22
        initialize matrices p1, p2, p3, p4, p5, p6, p7
        for i from 0 to newMatrixSize do        ▷ Dividing the matrices into 4 sub-matrices
            for j from 0 to newMatrixSize do
                a11[i][j] ← m1[i][j]
                a12[i][j] ← m1[i][j + newMatrixSize]
                a21[i][j] ← m1[i + newMatrixSize][j]
                a22[i][j] ← m1[i + newMatrixSize][j + newMatrixSize]
                b11[i][j] ← m2[i][j]
                b12[i][j] ← m2[i][j + newMatrixSize]
                b21[i][j] ← m2[i + newMatrixSize][j]
                b22[i][j] ← m2[i + newMatrixSize][j + newMatrixSize]
            end for
        end for
        tempM1 ← a11 + a22                                  ▷ Compute p1 to p7
        tempM2 ← b11 + b22
        p1 ← strassen(tempM1, tempM2, newMatrixSize)    ▷ p1 = (a11 + a22) * (b11 + b22)
        tempM1 ← a21 + a22
        p2 ← strassen(tempM1, tempM2, newMatrixSize)    ▷ p2 = (a21 + a22) * (b11 + b22)
        tempM2 ← b12 - b22
        p3 ← strassen(a11, tempM2, newMatrixSize)           ▷ p3 = (a11) * (b12 - b22)
        tempM2 ← b21 - b11
        p4 ← strassen(a22, tempM2, newMatrixSize)           ▷ p4 = (a22) * (b21 - b11)
        tempM1 ← a11 + a12
        p5 ← strassen(tempM1, b22, newMatrixSize)           ▷ p5 = (a11 + a12) * (b22)
        tempM1 ← a21 - a11
        tempM2 ← b11 + b12
        p6 ← strassen(tempM1, tempM2, newMatrixSize)    ▷ p6 = (a21 - a11) * (b11 + b12)
        tempM1 ← a12 - a22
        tempM2 ← b21 + b22
        p7 ← strassen(tempM1, tempM2, newMatrixSize)    ▷ p7 = (a12 - a22) * (b21 + b22)
        initialize c11, c12, c21, c22               ▷ Computing the four parts of the matrix
        c12 = p3 + p5
        c21 = p2 + p4
        c11 = p1 + p4 - p5 -p7
        c22 = p1 + p3 -p2 + p6
        for i from 0 to newMatrixSize do            ▷ Assembling the four parts of the matrix
            for j from 0 to newMatrixSize do
                m3[i][j] ← c11[i][j]
                m3[i][j + newMatrixSize] = c12[i][j]
                m3[i + newMatrixSize][j] = c21[i][j]
                m3[i + newMatrixSize][j + newMatrixSize] c22[i][j]
            end for
        end for
    end if
end function
```

Performance analysis. *Using the Master Theorem,*

$$T(n) = aT(\frac{n}{b}) + f(n)\ (*),$$

$$T(n) = 7T(\frac{n}{4}) + n^2\ (*).$$

We have $2 < log_2(7)$*, so we are in the first case. That's why:* $T(n) \in O(n^{log_b(a)})$*, where* $b = 2$ *and* $a = 7$*.*

The complexity of the Strassen algorithm is $O(n^{log_2(7)}) \simeq O(n^{2.807})$*.*

Nevertheless, the constant is too large, that's why we add a min size for the matrix.

In the following chart, leafsize is the size of the matrix before using a simple product, without partitionnig.

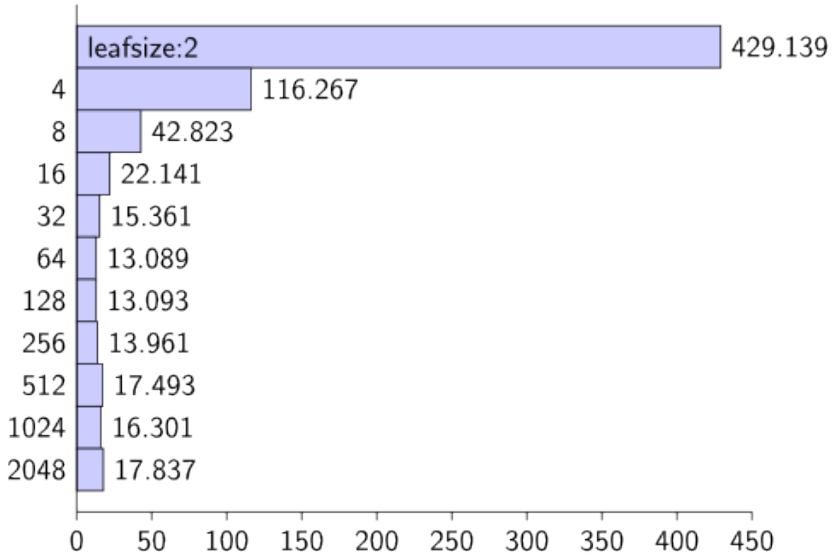


Figure 4.2: The speed of the algorithm for multiplication of two matrices of size 2000.

We can see that if we use the Strassen algorithm to compute the whole product it's slower that using Strassen until the size is smaller that 64 and after that one computes with a classical algorithm.

Moreover, the Strassen algorithm isn't stable numerically. It means that if you have a computer which doesn't use a computer algebra system in some parts of the matrix, the approximations will grow and, so, they aren't the same in the whole matrix.

5

Veldkamp spaces

In this chapter, we apply the techniques developped in this report to solve a geometrical problem. Given a geometry, i.e., a collection of points and lines (see 5.1), we provide an algorithm to define the Veldkamp space (i.e., a new geometry) associated to the first one. We will also analyse the complexity of the proposed algorithms. Moreover, this algorithm has already been used in a recent article [Saniga et al., 2016].

5.1/ Definitions

We start with a *point-line incidence structure* $\mathcal{C} = (\mathcal{P}, \mathcal{L}, I)$ where $\mathcal{P}$ and $\mathcal{L}$ are, respectively, sets of points and lines and where incidence $I \subseteq \mathcal{P} \times \mathcal{L}$ is a binary relation indicating which point-line pairs are incident (see, e. g., [Shult, 2011]). In what follows we shall deal with only specific point-line incidence structures where every line has just three points and any two distinct points are joined by at most one line.
The *order* of a point of $\mathcal{C}$ is the number of lines passing through it.
A *geometric hyperplane* of $\mathcal{C} = (\mathcal{P}, \mathcal{L}, I)$ is a proper subset of $\mathcal{P}$ such that a line from $\mathcal{C}$ either lies fully in the subset, or shares with it only one point. Given a hyperplane H of $\mathcal{C}$, one defines the *order* of any of its points as the number of lines through the point that are fully contained in H. If $\mathcal{C}$ possesses geometric hyperplanes, then one can define the *Veldkamp space* of $\mathcal{C}$ as follows [Buekenhout and Cohen, 2013]: (i) a point of the Veldkamp space (also called a Veldkamp point of $\mathcal{C}$) is a geometric hyperplane H of $\mathcal{C}$ and (ii) a line of the Veldkamp space (also called a Veldkamp line of $\mathcal{C}$) is the collection $H'H''$ of all geometric hyperplanes H of $\mathcal{C}$ such that $H' \cap H'' = H' \cap H = H'' \cap H$ or $H = H', H''$, where H' and H'' are distinct geometric hyperplanes. If each line of $\mathcal{C}$ has three points and $\mathcal{C}$ 'behaves well,' a line of its Veldkamp space is also of size three and can equivalently be defined as $\{H', H'', \overline{H' \Delta H''}\}$, where the symbol Δ stands for the symmetric difference of the two geometric hyperplanes and an overbar denotes the complement of the object indicated.

This geometric construction was used in quantum information theory to accommodate quantum operators and their commutation relations [Saniga et al., 2007].

5.2/ Algorithms

In this section, we will present three algorithms. One which computes the Veldkamp points by bruteforce, another one which computes the Veldkamp lines. The latter is an improvement of the first one, in the case where the geometry can be expressed as a Cartesian product $G = g' \times L_i$, where L_i is a line of i points and g' is a Cartesian product.

But in what follows, we consider only lines composed of three points.

5.2.1/ Bruteforce

In what follows, "bruteforce" means that the algorithm checks every possibility to determine the result. We will introduce an algorithm which works for every geometry.

5.2.1.1/ Veldkamp points

The following algorithm is used to compute Veldkamp points by bruteforce. This means that for each subset of points, the algorithm checks if this subset is a hyperplane.

Algorithm 5.1 ComputeHyperplanes

```
List<VeldkampPoints>veldkampPoints ← new List<VeldkampPoints>
for i ← 1 to numberOfPoints − 1 do
    for combination in combinations_i do                ▷ combinations of i points
        if isHyperplane(combination) then
            add(veldkampPoints, combination)
        end if
    end for
end for
return veldkampPoints
```

Algorithm 5.2 isHyperplane algorithm

```
function isHyperplane(potentialHyperplane)      ▷ potentialHyperplane is a subset of points
    for line in lines do
        intersect ← intersects(potentialHyperplane, line)
        if isEmpty(intersect) then
            return False
        end if                          ▷ If the intersection is a single point, we can't conclude.
        if isLine(intersect) then
            if not(isIncludeIn(potentialHyperplane, line)) then
                return False
            end if
        end if
    end for
    return True
end function
```

- isEmpty(line : Line) - checks if the intersection is null.
- isLine(line : Line) - checks if the intersection contains more than one point.
- isIncludeIn(first : Line, second : Line) - checks if the second line is included in the first one.

Performance analysis. *In this section we assume that all lines have n points, where n is the total number of points of the geometry. This approximation is here to compute the complexity of the worst case.*

To compute all the combinations, the complexity of the algorithm is $\sum_{k=1}^{n} k \times \binom{n}{k}$.

$$\begin{aligned}
\sum_{k=1}^{n} k \times \binom{n}{k} &= \sum_{k=1}^{n} \frac{n!}{(k-1)!(n-k)!}, \\
&= n \times \sum_{k=1}^{n} \frac{(n-1)!}{(k-1)!(n-1-(k-1))!}, \\
&= n \times \sum_{k=1}^{n} \binom{n-1}{k-1}, \\
&= n \times \sum_{k=0}^{n-1} \binom{n-1}{k}, \\
&= n \times (1+1)^{n-1}, \\
&= n \times 2^{n-1}.
\end{aligned} \tag{5.1}$$

For each combination, the algorithm checks whether the subset of points is a hyperplane (algorithm 5.2). For all lines, the algorithm computes the intersection (complexity $O(n^2)$ in worst case). The complexity of isHyperplane in the worst case is $O(m * n^2)$, where m is the number of lines.

This leads to a global complexity of $O(n^3 \times 2^{n-1} \times m)$ to find all the Veldkamp points.

Performance analysis. *In what follows, we assume that the considered geometry is the Segre variety S_N the Cartesian product of N lines of size 3, where N is the rank The number of points according to N is 3^N and the number of lines is equal to $N \times 3^{N-1}$. This leads to a complexity of $O(3^{N^3+N-1} \times 2^{3^N-1} \times N)$.*

5.2.1.2/ Veldkamp lines

The following algorithm is used to compute Veldkamp lines.

Algorithm 5.3 Find Veldkamp lines

```
List<VeldkampLines>veldkampLines ← new List<VeldkampLines>
for combination in combinations_2 do                ▷ combinations of 2 hyperplanes
    Hyperplane h1 = element 0 in combination
    Hyperplane h2 = element 1 in combination
    Hyperplane h3 = computeComplementOfTheSymmetricDifference(h1, h2)
    int indexH3 = indexOf h3 in hyperplanes
    if index of element 1 in combination < indexH3 then   ▷ Test if the Veldkamp lines have
already been computed
        VeldkampLine vl = new VeldkampLine(h1, h2, h3)
        put vl in veldkampLines
    end if
end for
return veldkampLines
```

Algorithm 5.4 Compute the complement of the symmetric difference of two hyperplanes

```
function computeComplementOfTheSymmetricDifference(h1 : Line, h2 : Line)
    List<Integer >result ← new List<Integer >
    i ← 0
    j ← 0
    for point in points do                          ▷ For each point of the geometry
        if i < number of points in h1 AND (getPoint i in h1 = element) then
            i ← i + 1
            if j < number of points in h2 AND (getPoint j in h2 = element) then
                j ← j + 1
                add element in result
            end if
        else
            if j < number of points in h2 AND (getPoint j in h2 = element) then
                j ← j + 1
            else
                add element in result
            end if
        end if
    end for
    return result
end function
```

Performance analysis. *In what follows, we assume that the list of hyperplanes is sorted out. The order of the hyperplanes doesn't have any mathematical sense. It's only to reduce the complexity of* indexOf. *For each combination of two hyperplanes, the complement of the symmetric difference of these two hyperplanes is computed, then if the Veldkamp lines have already been computed, the line isn't added to the list.*

The complexity of the main loop is $O(n^2)$. *The complexity of* computeComplementOfTheSymmetricDifference *is* $O(n)$, *where n is the number of points in the geometry. The complexity of* indexOf *is* $O(ln(m))$, *where m is the number of hyperplanes.*

This yields a global complexity of $O(ln(m) \times n^3)$.

Performance analysis. *In what follows, we assume that the considered geometry is* S_N, *where* N *is the rank.*

The number of hyperplanes in S_N *is* $2^{2^N} - 1$.

So the complexity in term of dimension is $O(3^{3N} \times ln(2^{2^N} - 1))$.

5.2.2/ Improvement

In this section we will present an improvement of the algorithm used to compute the Veldkamp points that employs some mathematical relation in order to reduce the complexity.

5.2.2.1/ Veldkamp points

This algorithm uses a geometric relation to compute the hyperplanes of $L_3 \times E$, where E is a point-line geometry; namely, the hyperplanes of $L_3 \times E$ are computed using the Veldkamp lines and the Veldkamp points of E [Saniga et al., 2014].

Algorithm 5.5 Find Veldkamp point algorithm

```
List<VeldkampPoints>veldkampPoints ← new List<VeldkampPoints>
for veldkampLine in veldkampLines do
    List<Line>hypers ← getHyperplanes(veldkampPoints, veldkampLine)
    List<List<Line>>permutations ← computePermutations(hypers)
    for permutation in permutations do
        Line veldkampPoint ← new Line
        for line in permutation do
            Line shiftedLine ← addScalar(line, i * nbrOfPointsInGeometry)
            addPoints(hyperplane, getPoints(shiftedLine))
        end for
        insert(veldkampPoints, hyperplane)
    end for
end for

Line fullGeometry ← new Line(geometryPoints)

for veldkampPoint in veldkampPoints do
    Line hyperplane ← new Line
    addPoints(hyperplane, getPoints(veldkampPoint))
    shiftedLine ← addScalar(veldkampPoint, nbrOfPointsInGeometry)
    addPoints(hyperplane, getPoints(shiftedLine))
    shiftedLine ← addScalar(fullGeometry, 2 × nbrOfPointsInGeometry)
    addPoints(hyperplane, getPoints(shiftedLine))
    insert(veldkampPoints, hyperplane)

    hyperplane ← new Line
    addPoints(hyperplane, getPoints(veldkampPoint))
    shiftedLine ← addScalar(fullGeometry, nbrOfPointsInGeometry)
    addPoints(hyperplane, getPoints(shiftedLine))
    shiftedLine ← addScalar(veldkampPoint, 2 × nbrOfPointsInGeometry)
    addPoints(hyperplane, getPoints(shiftedLine))
    insert(veldkampPoints, hyperplane)

    hyperplane ← new Line
    addPoints(hyperplane, getPoints(fullGeometry))
    shiftedLine ← addScalar(veldkampPoint, nbrOfPointsInGeometry)
    addPoints(hyperplane, getPoints(shiftedLine))
    shiftedLine ← addScalar(veldkampPoint, 2 × nbrOfPointsInGeometry)
    addPoints(hyperplane, getPoints(shiftedLine))
    insert(veldkampPoints, hyperplane)
end for
return veldkampPoints
```

First, for each permutation of VeldkampLines hyperplanes, the new hyperplanes are computed. After that a hyperplane is taken twice with the full geometry E, and for each permutation of those hyperplanes the new one is computed.

Performance analysis. *In this section, we assume that the hyperplanes and the Veldkamp lines of E are known. Moreover, we assume that a hyperplane has n point in the worst case, where n is the number of*

points in E.

First, for each Veldkamp line, every permutation is computed, thus for each permutation, the hyperplane is computed. After having been computed, the hyperplane is added to the list of Veldkamp points. The list used have an insertion cost of $O(1)$.

The complexity of computePermutations *is* $O(m!)$*, where m is the number of elements [Sedgewick, 1977]. But, since there are only 3 elements, the complexity is constant.*

Likewise, the complexity of the loop on the permutation is also constant. The complexity of addScalar *is* $O(n)$.

So, the complexity of this first loop is: $O(nbrOfVeldkampLinesOfE \times n)$.

Second, for each hyperplane of E, 3 hyperplanes of $L_3 \times E$ *are computed. The complexity of this loop is* $O(nbrOfHyperplanesOfE \times n)$ *since the complexity of* addScalar *is n.*

So, the global complexity of this algorithm is $O(n \times (nbrOfVeldkampLinesOfE + nbrOfHyperplanesOfE))$.

Performance analysis. *As seen before, in* S_N*, the number of hyperplanes and Veldkamp lines can be expressed according to the rank N. Moreover, the number of points in* S_N *can also be expressed in terms of N.*

$$numberOfPoints = 3^n.$$

So the complexity of this algorithm in terms of N is

$$O(3^N \times (2^{N-1} - 1) \times \frac{2^{(2^{N-1}-1)} + 2}{3}).$$

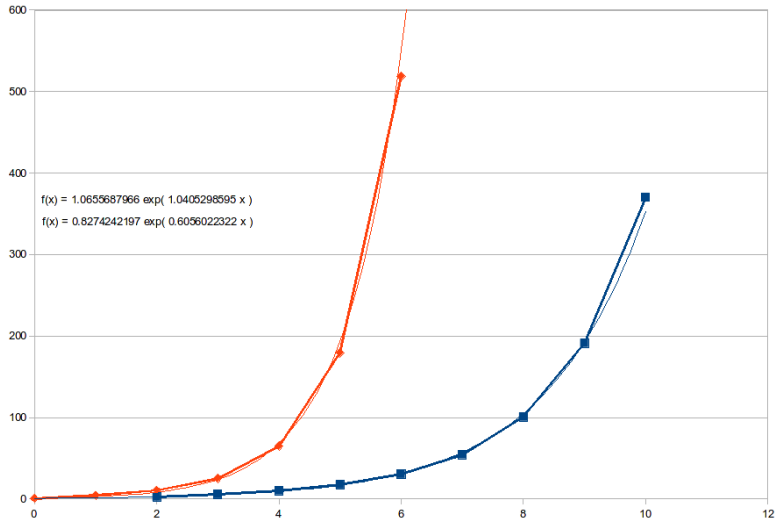


Figure 5.1: Comparison of the two algorithms. (*ln* scale)
Red — the first version, blue — the second one

Acknowledgements

We are extremely grateful to Frédéric Holweck and Metod Saniga for helping us to bring this report to a successful completion, introducing us to their ongoing research at the UTBM and for thoroughly reviewing our report.

Bibliography

Francis Buekenhout and Arjeh M. Cohen. *Diagram Geometry: Related to Classical Groups and Buildings*. Springer, 2013. ISBN 978-3-642-34453-4.

Tim Roughgarden. Coursera mooc, algorithms: Design and analysis, part 1, 2016. URL https://www.coursera.org/course/algo. [Online; accessed June 8, 2016].

Metod Saniga, Michel Planat, Petr Pracna, and Hans Havlicek. The veldkamp space of two-qubits. 2007. URL http://www.emis.ams.org/journals/SIGMA/2007/075/sigma07-075.pdf. [Online; accessed June 08, 2016].

Metod Saniga, Hans Havlicek, Frédéric Holweck, Michel Planat, and Pracna Petr. Veldkamp-space aspects of a sequence of nested binary segre varieties. 2014.

Metod Saniga, Frédéric Holweck, and Petr Pracna. Veldkamp spaces: From (dynkin) diagrams to (pauli) groups. 2016. URL https://www.researchgate.net/publication/302378003_Veldkamp_Spaces_From_Dynkin_Diagrams_to_Pauli_Groups.

Robert Sedgewick. *Permutation Generation Methods*. 1977.

Robert Sedgewick and Kevin Wayne. Insertion sort trace, 2002-2014a. URL http://algs4.cs.princeton.edu/21elementary/images/insertion.png. [Online; accessed May 16, 2016].

Robert Sedgewick and Kevin Wayne. Selection sort trace, 2002-2014b. URL http://algs4.cs.princeton.edu/21elementary/images/selection-2.1.1.png. [Online; accessed April 26, 2016].

Robert Sedgewick and Kevin Wayne. *Algorithms, fourth edition*. Addison-Wesley Professional, 2011. ISBN 978-0-321-57351-3.

Robert Sedgewick and Kevin Wayne. Coursera mooc, algorithms, part 1, 2016. URL https://www.coursera.org/course/algs4partI. [Online; accessed June 8, 2016].

Ernest Shult. *Points and Lines: Characterizing the Classical Geometries*. Springer, 2011. ISBN 978-3-642-15627-4.

INDEX

This report deals with the issue of the complexity of certain algorithms. First, the fundamentals of the study of algorithm complexity are addressed by examining theoretical aspects and classical sorting algorithms: *selection sort*, *insertion sort* and *merge sort*. Then, a famous theorem called the *Master Theorem* is introduced and proved, to be subsequently employed to tackle a certain type of recursive algorithms – so-called *divide-and-conquer* algorithms. As a particularly illustrative example, a well-known algorithm for matrix multiplication – the *Strassen algorithm* – is described and evaluated. Finally, the theory is applied to an intriguing geometrical problem as part of the ongoing international research at the UTBM.

Ce rapport traite d'analyse algorithmique. D'abord, les fondamentaux de l'étude de la complexité algorithmique sont abordés en examinant les aspects théoriques et des algorithmes de tri classiques: *tri par sélection*, *tri par insertion* et *tri par fusion*. De plus, le très connu *Master Theorem* est introduit et prouvé. Ce théorème est utilisé pour étudier un certain type d'algorithmes récursifs appelés algorithmes divide-and-conquer. Pour illustrer cela, un algorithm pour la multiplication matricielle est présenté, *l'algorithme de Strassen*. Enfin, une application à un réel problème géométrique dans le cadre de la recherche à l'UTBM est traitée.

Algorithms, Complexity, Sorting, Matrix multiplication, Veldkamp geometry, Master Theorem, Recursivity, LaTeX

Jérôme BOULMIER & Benoît CORTIER• University of Technology of Belfort-Montbéliard