# Analysis of Algorithms

João Augusto Costa Branco Marado Torres

December 22, 2024

**Abstract**

This will be a summary the basics that you can find in *Introduction to Algorithms* [H C+09] and *Introduction to The Design & Analysis of Algorithms* [Lev11] plus some of the things shown during classes.

The source code for this paper is available at:
https://github.com/Marado-Programmer/alganalysis [Cos24a]

## License



## Contents

# Listings

# 1 Introduction

Analysis of Algorithms is the study of how algorithms perform based on efficiency and resource consumption.

# 2 Algorithms

An algorithm is a group of instructions for something (human or machine) to do by receiving a set of values (the input) and returning a set of values (the output). Those instructions are unambiguous and are abstract in a way that it can be implemented using any programming language, and if the machine permits it.

Like a recipe, any person can follow the recipe (even if they do some steps in different ways), but the kitchen needs to have all the resources (furniture, ingredients) to be able to complete the recipe.

An algorithm solves a problem, and that problem can be solved in various ways by different algorithms. Some will fit better than others and that can depend on a lot of things. You want to find the best algorithm for your needs.

Algorithms often work alongside with data structures **??**.

Some other things you can take into account when creating an algorithm is about **finiteness**, **effectiveness** and **generality**.

## 2.1 Analysis and Design

An algorithm can be analyzed using its time complexity.

The time complexity is the amount of time an algorithm takes to complete, given an input of size `N` using asymptotic notation such as Big $O$ notation that shows how time grows with the input size.

Time complexity can be analyzed in terms of:

- worst case scenario – maximum time it can take, maximum steps on input of size `N`;
- average case scenario – expected time it can take over typical inputs, average steps on input of size `N`;
- best case scenario – minimum time it can take, minimum steps on input of size `N`.

The space complexity is the amount of memory an algorithm takes to complete, given an input of size `N`. Helps understand storage requirements when working with big data or memory limited environments.

Algorithm analysis finds the most efficient time/space complexity solution for a given problem for a specific situation.

There will always be various ways to solve a problem. But only a few of them will have the characteristics, performance and time complexity and space complexity efficiency that fits your needs. So you need to learn the best, average and worst case of the algorithm and the algorithm behavior as the input gets larger (asymptotic notation, unbounded behavior).

Inefficient algorithms can use unnecessary computing power or storage which costs time. A problem for some places when time is precious.

We always estimate the complexity of an algorithm in an asymptotic sense (for an arbitrarily large input), when the input size approaches infinity.

This is the determination of the amount of time and space resources required to execute the algorithm.

One way to measure the amount of time is to know which instructions from a certain ISA will execute and how many instructions of each will be executed. Then by knowing each CPI for every instruction, calculate the total of cycles of your program will have and then divide the number of cycles by the CPU frequency. Some ISA can have instructions that other don't and need to be implemented based on more basic instructions which can take up more CPU cycles.

We can also count how much memory it's used by all instructions to have our space complexity.

We also need to take into the account the **input size** here which can vary depending on the problem. For searching or sorting a list, it can be the number of elements in a list, for multiplication of two numbers it can be the total number of bits to represent both inputs, for graphs it can be the number of edges and vertices in it.

The **running time** is the number of "steps" executed by the algorithm. For simplicity normally we treat every operation as constant time (1 "step") except loops.

But what really is important it's to know the order/rate of grow of the running time. We want to know what the running time will be as the input size approaches $\infty$, where the leading term of the running time will be the only one that we care.

Asymptotic Notation2.4 it's often used for this.

## 2.2 Stages of algorithm analysis

### 2.2.1 *A Priori*

This happens before implementation. Assuming that factors external to the algorithm logic such as processor speed, memory read and writes, compiler optimizations, etc..., have no effect.

If you want to distribute a product you should rely on this since you don't know the user machine.

### 2.2.2 *A Posterior*

After implementation by measuring the actual running time and space required of an implementation on a machine (benchmark).

## 2.3 Main factors of the efficiency of an algorithm

- Time factor – number of key operations;
- Space factor – maximum memory space required.

## 2.4 Asymptotic Notation

Measure efficiency that does not depend on the machine.

Being $f(n)$ and $g(n)$ two running time functions for an input size of n.

### 2.4.1 Upper bound of a function

Represents the worst case scenario.

**Definition 2.1 – Big $O$**

$$\exists c, n_0 \in \mathbb{R}^+ : \forall n \geq n_0, f(n) \leq cg(n) \implies f(n) = O(g(n))$$

### 2.4.2 Lower bound of a function

Represents the best case scenario.

**Definition 2.2 – Big $\Omega$**

$$\exists c, n_0 \in \mathbb{R}^+ : \forall n \geq n_0, f(n) \geq cg(n) \implies f(n) = \Omega(g(n))$$

### 2.4.3 Average bound of a function

Represents the average case scenario.

**Definition 2.3 – Big $\Theta$**

$$\exists c_1, c_2, n_0 \in \mathbb{R}^+ : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \implies$$
$$\implies f(n) = \Theta(g(n))$$

## 2.5 Time/Space tradeoff

You can not have both. For example, the factorial:

$$0! = 1$$

$$x! = x(x-1)!$$

To calculate the factorial of a number, you can calculate it recursively every time which is slower than preprocess the result for some inputs and store them for when needed that depending on the data structure used to store those preprocessed values, will be faster, but it uses more storage.

## 2.6 Design Strategies

### 2.6.1 Brute Force

It's the most straightforward approach to solving a problem.

If you want to calculate $x^6$ you can calculate it as it is defined: $x \times x \times x \times x \times x \times x$. That would be the brute force approach. But you can think of it as $x^3 \times x^3$, where now you can calculate $x^3$ one time and then multiply it by itself. But that's not the straightforward approach, so it's not considered brute force.

### 2.6.2 Decrease and Conquer

1. **Decrease** the problem into sub problems which the result has relationship with the final result;
2. **Conquer** by keeping on dividing until a solution is found;
3. **Accumulate/Increment** the sub problems solution into the actual final solution.

### 2.6.3 Divide and Conquer

1. **Divide** the problem into sub problems;
2. **Conquer** by keeping on dividing until a solution is found;
3. **Combine** the sub problems solution into the actual final solution.

The main pros of applying this design strategy is that it supports parallelism. But you need to be careful with the implementation in case of recursion (probably will be the case). If tail call optimization isn't used, you might have problems with stack overflow.

### 2.6.4 Transform and Conquer

Here the goal is to transform the input into a simpler representation, or just a simpler one. Then we resolve the problem with the transformed input or use an algorithm already existent for the transformed input.

### 2.6.5 Dynamic Programming

It's more like a technique for solving problems with overlapping sub problems that rather than solving them again and again, you compute them only once by using storage

### 2.6.6 Greedy Technique

It's a way of constructing the solution little by little. Each step towards the solution needs to be locally optimal, so between all possible steps that you can take, there will be one that is better than the others. Also, each step is irrevocable, once made you can not come back, but if you already took the optimal step, there will be no reason to come back.

For some algorithms, a sequence of the locally optimal steps might not give the optimal solution.

# 3 Analyzing my algorithm

Since 2021 I've been doing the Advent of Code[Was24b], first 2 years I did it on the JavaScript console in the browser with the input page opened while in lab classes. Last year and this year I'm using it as a way to learn new programming languages, and I leave my solutions on GitHub [Cos24b].

Let's analyze my solution for part 1 of day 5.

Listing 1: solution for day 5, part 1 of the Advent of Code 2024

```
-module(day_05_impl).

-export([part_1/1]).

-spec part_1(string()) -> integer().
part_1(In) ->
    {Rules, Lists} = get_parts(In),
    Sort = create_sort_from_rules(Rules),
    Filtered = lists:filter(
        fun(X) -> X =:= lists:sort(Sort, X) end,
        Lists
    ),
    sum_middles(Filtered).

get_parts(In) ->
    [Rules, Lists] =
        lists:map(
            fun(X) -> string:split(X, "\n", all) end,
            [X || X <- string:split(In, "\n\n", all),
                  string:length(X) > 0]
        ),
    {lists:map(
        fun(X) ->
            [First, Last] = string:split(X, "|", all),
            {First, Last}
        end,
        Rules
    ), lists:map(
        fun(X) -> string:split(X, ",", all) end,
        Lists
    )}.

sum_middles(Lists) ->
    Middles = lists:map(
        fun(X) -> lists:nth(
            (list_length(X) + 1) div 2,
            X
        ) end,
        Lists
```

```
    ),
    lists:sum(lists:map(
        fun(X) -> binary_to_integer(X) end,
        Middles
    )).

create_sort_from_rules(Rules) ->
    fun(X, Y) -> not lists:any(
        fun(Z) -> Z == {Y, X} end,
        Rules
    ) end.

list_length(List) ->
    list_length(List, 0).

list_length([_ | T], Acc) ->
    list_length(T, Acc + 1);
list_length([], Acc) ->
    Acc.
```

It receives the input string, and it parses it to extract its rules and lists.

Let's pretend the separated rules and lists are already given. So we have as an input a list of pairs of tokens (these tokens are string of numbers that are not parsed as numbers because they do not need to), those are the rules, and a list of lists of numbers.

In this case, a list is a singly-linked list. It's one of the basic structures provided by Erlang, the other being tuples and maps, but the language also has bit strings/binaries, strings and records.

So first we create the predicate for a sort function, which takes constant time $O(1)$. The execution of that predicate however will take $O(n)$ as it uses lists:any

Now we filter from the list of lists those how are already sorted. The implementation of lists:sort is some time of divide and conquer algorithm that I am not sure if it is mergesort but let's assume it is, so time complexity $O(n \log(n))$. The filtering itself has for sure time complexity $O(n)$. The comparing of the list with the operator `=:=` is possibly $O(n)$ too as it is comparing each element of both lists one by one.

So to create the variable **Filtered** has runtime function of

$$\begin{aligned} f(n,m) &= n(n + (n\log(n))(m)) = \\ &= n(n + n\log(n)m) = \\ &= n^2 + n^2\log(n)m = \\ &= O(n^2\log(n)m \end{aligned} \tag{1}$$

Where n is the amount of lists and m the amount of rules.

# References

[H C+09] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. Cambridge, Massachusetts and London, England: The MIT Press, July 2009. ISBN: 9780262033848 and 9780262533058.

[Lev11] Anany Levitin. *Introduction to The Design & Analysis of Algorithms*. Ed. by Pearson. 3rd ed. Sept. 2011. ISBN: 0132316811 and 9780132316811.

[Cos24a] João Augusto Costa Branco Marado Torres. *alganalysis*. Dec. 2024. URL: https://github.com/Marado-Programmer/alganalysis.

[Cos24b] João Augusto Costa Branco Marado Torres. *aoc2024*. git. Dec. 2024. URL: https://github.com/Marado-Programmer/aoc2024.

[Was24b] Eric Wastl. *Advent of Code*. 2024. URL: https://adventofcode.com/.

# Other Resources

[Ahm24a] Kamran Ahmed. *Computer Science*. Dec. 22, 2024. URL: https://roadmap.sh/computer-science (visited on 12/22/2024).

[Ahm24b] Kamran Ahmed. *Data Structures & Algorithms*. Dec. 22, 2024. URL: https://roadmap.sh/datastructures-and-algorithms (visited on 12/22/2024).

[Was24a] John Washam. *Coding Interview University*. git. manual. Dec. 5, 2024. URL: https://github.com/jwasham/coding-interview-university (visited on 12/20/2024).