University of Trieste

International School for Advanced Studies

The Abdus Salam International Centre for Theoretical Physics

Algorithmic Design

Lectures Notes

Author:
Marco Sciorilli

Gennaio 2021

Abstract

This document contains my notes on the course of Algorithmic Design held by Prof. Alberto Casagrande for the Master Degree in Data Science and Scientific Computing at Trieste University in the year 2020/2021. As they are a work in progress, every correction and suggestion is welcomed. Please, write me at: marco.sciorilli@gmail.com .

Contents

Chapter 1

Fundamentals

1.1 Algorithm and Computational Model

Definition 1. *Algorithm: a sequence of well-defined steps that transforms a set of inputs into a set of outputs in a finite amount of time.*

The output should be consistent with the given definition of the problem. In some specific times, there could be actually an algorithm that goes on indefinitely. But that case is not studied in this course.

Definition 2. *Computational model: a mathematical tool to perform computations.*

A function described by an algorithm is calculable(i.e. I can give an high level set of instructions to turn the input of the function into the output of the function, I can provide and algorithm for it).
A function is implementable in a computational model is computable (i.e. Given a formal model, it provides a set of instructions which are fixed, making me able to write a program that turn the input of the function in the output of the function using only that proposed rules).
Notion of calculable and computable are related? Algorithms would not guarantee implementability! An interesting example is the "Halting problem": we want to write a program that takes as input any other program and it establish whether that ends in a finite amount of time or not.

Example 1. *Halting Problem: Let h be the function that establish whether any program p eventually ends its execution ($\downarrow$) on an input i or runs forever ($\uparrow$)*

$$h(p,i) \overset{\text{def}}{=} \begin{cases} 0 & \textit{if } p(i) \textit{ never ends} \\ 1 & \textit{otherwise} \end{cases} \tag{1.1}$$

It is not possible to implement h.

Proof. For any computable function $f(a, b)$ we can define the function

$$g_f(, i) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } f(i,i) = 0 \text{(if it ends)} \\ \uparrow & \textit{otherwise} \text{(it goes on forever)} \end{cases} \tag{1.2}$$

If we assume that f is computable, so is g_f, cause the first part can be implemented using a conditional statement, while the second can be implemented using a "while" forever loop. If I can evaluate the former, I can evaluate the latter. Let's call G_f the program implementing the function g_f.
Can f be the Halting function? No

- If $f(G_f, G_f) = 0 \Rightarrow g_f(G_f) = 0$ and $h(G_f, G_f) = 1$: a.k.a. if G_f applied on G_f gives 0, then g_f ends, so the halting function return 1.
- If $f(G_f, G_f) \neq 0 \Rightarrow g_f(G_f) \uparrow$ and $h(G_f, G_f) = 0$: a.k.a. if G_f applied on G_f is not 0, then g_f goes on forever, so the halting function return 0.

So, any computable function must be different from h, because any f with a given result must be different from its associated h. □

Luckily, there is a thesis that gives a correlation between computable and calculable.

Thesis 1. *Church-Turing Thesis: Every effectively calculable function is a computable function.*

So, if we are able to describe an algorithm for a function f, then f can be formally computed. This has some direct consequences

- All the "reasonable" computation models are equivalent.(i.e. we are able to write a program in every different programming language)
- We can avoid "hard-to-be-programmed" models.(i.e. we can avoid models which are not easy to be handled)

Example 2. *Random-Access Machine: an easy computational model.*
It allows to

- *Variables to store data (no types)*
- *Arrays*
- *Integer and floating point constatns*
- *Algebraic functions*
- *Assignments*
- *Pointers (no pointer arithmetic)*
- *Conditional and loop statements*
- *Procedure definitions and recursion*
- *Simple "reasonable" functions*

This is the computational model used in this course, not a real machine. Now let's look at an example of an algorithm, assuming to work on this computational model (so thanks to the Church-Turing Thesis every conclusion on this machinery will work on real life machine)

Algorithm 1: Algorithm example: find the maximum in an array

Input: An array A of numbers $< a_1, ..., a_2 >$
Output: The maximum among $a_1, ..., a_2$

```
def find_max(A)
    max_value← A[1]
    for $i$← $2..|A|$ do
        if A[i]>max_value then
            max_value← A[i]
        end
    end
    return  max_value
enddef
```

Ram is not a real hardware: any variable in a ram algorithm can store any possible number. It doesn't have memory hierarchy, and no difference in instruction execution time (for example, same time for all the operations).

1.2 Time Complexity

We want to asses the efficiency of the algorithm. Execution time is not an effective way to do so, because is not applicable in many cases. Counting the number of operations could be a good idea, but it depends on the size of the input.

Definition 3. *Scalability: effectiveness of a system in handling input growth.*

It means that the system has to keep the execution time low even if the input is big. We want to estimate the relation between the input size and the execution time, we use a function to do so.
The important part of this function is the asymptotic behavior (its exponent). Constants are not relevant. A theorem called "linear time speedup theorem", support this idea, as it asses that for every Touring Machine, it is possible to built an equivalent one scaled by a constant.

Big * notation

These are ways in which one can collect all the functions that behave in the same way.

Definition 4. *Big-O notation:*

$$O(f) \overset{\text{def}}{=} \{g | \exists c > 0 \; \exists n_0 > 0 \; s.t. \; m \geq n_0 \; \Rightarrow g(m) \leq cf(m)\} \tag{1.3}$$

It give the upper bound.

Example 3. *Examples of functions' $O(n)$ appartenance:*

- $n \in O(n)$
- $2n \in O(n)$
- $500n \in O(n)$
- $2n + 7 \in O(n)$
- $n \in O(n^2)$
- $n \in O(2n) = O(n)$

Definition 5. *Big-Ω notation:*

$$\Omega(f) \overset{\text{def}}{=} \{g | \exists c > 0 \ \exists n_0 > 0 \ s.t. \ m \geq n_0 \ \Rightarrow cf(m) \leq g(m)\} \tag{1.4}$$

It gives a lower bound.

Example 4. *Examples of functions' $\Omega(n)$ appartenance:*

- $n \in \Omega(n)$
- $2n \in \Omega(n)$
- $500n \in \Omega(n)$
- $2n + 7 \in \Omega(n)$
- $n \notin \Omega(n^2)$
- $n \in \Omega(2n) = \Omega(n)$

Properties of Big-O notation, for any $c_1, c_2 \in \mathbb{N}$ and for any $k \in \mathbb{Z}$:

- $f(n) \in O(f(n))$
- $O(f(n)) = O(c_1 f(n) + k)$
- if $c_1 \geq c_2$ then $O(f(n)^{c_1} + kf(n)^{c_2}) = O(f(n)^{c_1})$
- $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$
- if $h(n) \subseteq O(f(n))$ and $h'(n) \in O(g(n))$, then
  - $h(n) + h'(n) \in O(g(n) + f(n))$
  - $h(n) \times h'(n) \in O(g(n) \times f(n))$

Definition 6. *Big-Θ notation:*

$$\Theta(f(n)) \overset{\text{def}}{=} \{g(n) | \exists c_1, c_2 > 0 \ \exists n_0 > 0 \ s.t. \ m \geq n_0 \ \Rightarrow c_1 f(m) \leq g(m) \leq c_2 f(m)\} \tag{1.5}$$

Theorem 1.

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n)) \tag{1.6}$$

1.3 Cost Criteria

We are interested in how many time instructions are executed, not in the time of the single instruction.
Uniform cost criterion states that time cost is:

- 1 for any Boolean and algebraic expression evaluation.
- 0 for assignments and control instructions.

Any expression, no matter how difficult it is, cost the same. The aim is not to evaluate the execution time, but how many time an instruction is executed.

Example 5. *Uniform Cost Criterion Example*

Algorithm 2: Uniform Cost Criterion Example

```
def test(n):
    Z ← 2 // costs 1
    for i← 1..n do // loop n time
        Z←Z*Z // costs 1
    end
    return Z
enddef
```

Cost functions is $T(n) = 1 + n * 1 \in \Theta(n)$

However, $test(n) = 2^{2^n}$, so the function in linear time, according to the uniform cost criterion, return a very large number, that could not be stored in a linear space.
Logarithmic Cost Criterion states that:

- $\max_{i\in[0,c]}(\log a_i)$ for any Boolean and algebraic expression involving $a_0, ..., a_c$ as operands.
- 0 for assignments and control instructions.

Example 6. *Logarithmic Cost Criterion Example*

Algorithm 3: Logarithmic Cost Criterion Example

```
def test(n):
    Z ← 2 // costs 2
    for i← 1..n do // loop n time
        Z←Z*Z // costs log Z
    end
    return Z
enddef
```

Cost functions is $T(n) = \sum_{i=1}^{n} \log 2^{2^i} = \sum_{i=1}^{n} 2^i = 2 * (2^n - 1) \in \Theta(2^n)$

Logarithmic cost better represents big-number algorithms. If instead the representation space is bounded (as for CPU arithmetic), uniform cost is enough. Where not otherwise declared, we will use uniform cost criterion.

1.4 Some Useful Notions

Definition 7. *Arrays: indexed collections of values which is fixed in length.*

Definition 8. *Single-Linked Lists: sequences of values supporting `head` (the first element) and `next` operations.*

Definition 9. *Double-Linked Lists: sequences of values supporting `head` (the first element), `next` and `previous` operations.*

Definition 10. *Queues: Collections of values ruled according to the FIFO policy (First In First Out:the first element inserted in the list, should be the first one to be taken out). They support `head`, `is_empty`,`insert_back`, `extract_head` operations.*

Definition 11. *Stacks: Collections of values ruled according to the LIFO policy (Last In First Out:the last element inserted in the list, should be the first one to be taken out). They support `head`, `is_empty`,`insert_back`, `extract_head` operations*

Definition 12. *Graphs: set of pairs (V,E) where:*

- *V is a set of nodes*
- *E is a set of edges*

If the edges are (un)directed, the graph itself is (un)directed.

Definition 13. *Path of length n between $a, b \in V$: a sequence $e_1, ..., e_n$ s.t.*

- *e_1 involves a*
- *e_n involves b*
- *e_i and e_{i+1} involve a common node n_i*

A Cycle is a node in which the fist node is also the last one.
A graph is connected if there is a path between every pairs of nodes.
A graph is acyclic if it does not contains cycles.
A tree is a connected and acyclic undirected graph. Tree are useful to organize data, and store it in a smart way. The `root` is the node from which all the tree originate.
The `depth` the length of the path going from the node considered to the `root`.
A `level` is a set of nodes having the same `depth`.
The parent of a node is a node one step closer to the root. The children of a node have the node as a parent. Two nodes are siblings if they have the same parent. Leaves are nodes without children, while internal nodes have children.
The height of a tree is the max depth among those of its leaves.
Every node of a n-ary tree can have up to n children. A n-ary tree is complete if the nodes in all the levels but the last one have n children.

Chapter 2

Strassen's Algorithm for Matrix Multiplication

2.1 Problem definition

Definition 14. *Row-Column Multiplication: let A be a $n \times n$ matrix and let B be a $m \times l$ matrix. $A \times B$ is a $n \times l$ matrix s.t.*

$$(A \times B)[i, j] = \sum_k A[i, k] * B[k, j] \tag{2.1}$$

Input: Two $n \times n$ matrices A and B (where n is a power of 2)
Output: The $n \times n$ matrix $A \times B$

2.2 Motivation

- In Deep Neural Network, both training and evaluation heavily rely on matrix multiplication.
- Good example to learn how to compute the complexity of an algorithm.

2.3 Naive Solution

Do exactly as you would do by hand.

Algorithm 4: Naive matrix multiplication

```
def naive_mult(C,A,B):
    for $i \leftarrow$ 1..rows(A) do
        for $j \leftarrow$ 1...cols(B) do
            a$\leftarrow$ 0
            for $k \leftarrow$ 1..cols(A) do
                a$\leftarrow$ a + $A[i,k] * B[k,j]$
            end
            $C[i,j] \leftarrow$ a
        end
    end
    return C
enddef
```

The algorithm is composed by 3 nested loop with indexes in $[1, n]$. The inner block takes $O(1)$, so the overall execution complexity takes time $\Theta(n^3)$. Is it possible to find a better algorithm using recursion?

2.4 Divide-and-Conquer Strategy

We try to use recursion by splitting the matrix into 4 quadrants, where

$$C_{ij} = (A_{i1} \times B_{1j}) + (A_{i2} \times B_{2j}) \tag{2.2}$$

We can than use the standard technique on the blocks. Let's find the complexity of this new matrix.
We can find the complexity required to find a quadrant by finding the complexity of the matrix product between matrices a quarter of the original size, plus the cost of the sum of the resulting matrices. Defining some symbols:

- $+$ is the element-wise matrix sum (with time complexity $\Theta(n^2)$)
- $\times$ is the usual row-column moltiplication
- A_{ik} and B_{kj} are $\frac{n}{2} \times \frac{n}{2}$ matrices

So it is possible to write the equation

$$T(n) = \begin{cases} 1 & n = 1 \\ 4(2 \cdot T(\frac{n}{2}) + \Theta(\frac{n^2}{4})) = 8 \cdot T(\frac{n}{2}) + \Theta(n^2) & otherwise \end{cases} \tag{2.3}$$

To solve it we can use a recursion tree. Each node is labeled by the cost of that specific call. In particular since we are saying that the cost at each step of the recursion is quadratic, we may select one specific constant such that the cost we are dealing with belong to $\Theta(n^2)$.

The first call cost $c \cdot n^2$. After the first call the matrix is split in four parts, and for each of the four new call we need to go through recursive call, so we have four child node of cost $c \cdot (\frac{n^2}{2})$ and we go on splitting in new children nodes until $n = 1$.
The height of the tree is $\log_2 n$. So

$$\begin{aligned}
T(n) =& c \cdot n^2 + 2 \cdot c \cdot \frac{4}{2^2} \cdot n^2 + ... \\
&= \sum_{i=0}^{\log_2 n} 8^i \cdot c \cdot \frac{n}{n^i}^2 = \sum_{i=0}^{\log_2 n} (\frac{8}{4})^i \cdot c \cdot n \\
&= c \cdot n^2 \cdot \frac{(2^{(\log_2 n+1)} - 1)}{2-1} \\
&= c \cdot n^2 \cdot (2 \cdot n - 1) \\
&= 2 \cdot c \cdot n^3 - c \cdot n^2 \in O(n^3)
\end{aligned}$$

Which is the same as the naive algorithm. It would be nice to reduce the recursive calls. It is possible to do so.

2.5 Strassen's Algorithm

This algorithm consists of doing a bunch of sums that ends up in 10 different matrices, than perform on them 7 matrix multiplications, and from them, doing other sums, we are able to end up with the quadrants of the final matrix. Strassen achieve better results by doing more sums. The recursive equation associated with Strassen's algorithm

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 7 \cdot T(\frac{n}{2}) + \Theta(n^2) & otherwise \end{cases} \tag{2.4}$$

The recursion tree this time is So the final time complexity is

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n} 7^i \cdot c \cdot \frac{n}{n^i}^2 \\
&= c \cdot n^2 \cdot \sum_{i=0}^{\log_2 n} (\frac{7}{4})^i \\
&= c \cdot n^2 \cdot \frac{((\frac{7}{4})^{\log_2 n+1} - 1)}{\frac{7}{4} - 1} \\
&= c' \cdot n^2 \cdot ((\frac{7}{4})^{\log_2 n+1} - 1) \quad \text{for} \quad c' = \frac{4}{3}c \\
&= c'' \cdot n^2 \cdot (\frac{7}{4})^{\log_2 n} - \frac{4}{7} \\
&= c'' \cdot 4^{\log_2 n} \cdot (\frac{7}{4})^{\log_2 n} - c' \cdot n^2 \\
&= c'' \cdot 7^{\log_2 n} - c' \cdot n^2 \\
&= c'' \cdot n^{\log_2 7} - c' \cdot n^2 \in \Theta(n^{\log_2 7})
\end{aligned}$$

So we are able to reduce the complexity to

$$O(n^{log_2 7}) \subseteq O(n^3) \tag{2.5}$$

Chapter 3

Matrix Chain Multiplication Problem

The problem is focused on the parenthesization of the product, the order in which we want to apply the matrix multiplication.

3.1 Problem definition

We are dealing with a sequence of matrices. As the matrix product is associative, we can compute multiplication in the order we prefer. Even thought the result is the same, the number of operation required by it can vary a lot.

Example 7. *Let's consider three matrices: A_1, A_2 and A_3*

- *A_1 has dimensions* 50×5
- *A_2 has dimensions* 5×100
- *A_3 has dimensions* 100×10

The number of operation required for their product is:

- *For $(A_1 \times A_2) \times A_3$*

$$50 * 100 * 5 = 25000$$
$$50 * 10 * 100 = 50000$$

- *For $A_1 \times (A_2 \times A_3)$*

$$5 * 10 * 100 = 5000$$
$$50 * 10 * 5 = 2500$$

So the first case require 75000 operations, the second 7500, 10 times less.

Consider a chain of matrices $\langle A_1, ..., A_n \rangle$ where A_i has dimension $p_{i-1} \times p_i$ for all $i \in [1, n]$. The aim is to compute a parenthesization that minimizes the number if scalar product for the chain multiplication considered. This problem is usually faced during the design part of the coding.
Chain matrix multiplication are often used in Deep neural networks. It is also useful because it gives a relevant speedup in data preparation pipeline.

3.2 A naive approach

Build up every possible parenthesization and compute the cost of the parenthesization. As we are doing it at design time, there usually are no time constrains.

- If $n = 1$, the parenthesization is obvious
- If $n > 1$, the chain can be parenthesization as

  $$(A_1 \times ...A_k) \times (A_{k+1} \times ...A_n) \tag{3.1}$$

  for any $k \in [1, n-1]$. I can then recursively produce the parenthesization for $\langle A_1, ..., A_k \rangle$ and $\langle A_{k+1}, ..., A_n \rangle$

But for a sequence $1..n$ the number of parenthesization is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & \text{if } n > 1 \end{cases} \tag{3.2}$$

Trying to give a lower bound to the second term of the previous expression

$$\text{1st term } P(1) \cdot P(n-1) \tag{3.3}$$

$$\text{Last term } P(n-1) \cdot P(n-(n-1) \tag{3.4}$$

So we can derive that

$$P(n) \geq 2 \cdot P(1) \cdot P(n-1) = 2 \cdot P(n-1) \tag{3.5}$$

So finally

$$P(n) \in \Omega(2^n) \tag{3.6}$$

And computing all of that is basically impossible.
We notice however that:

- If the given parenthesization $(A_1 \times ...A_k) \times (A_{k+1} \times ...A_n)$ is the optima chain, than we may conclude that
  - The first part is optimal for $\langle A_1 \times ...A_k \rangle$
  - The second part is optimal for $\langle A_{k+1} \times ...A_n \rangle$
- So many branches of the naive recursive approach preform the same computation (once the parenthesization of a section is done, there is no need to re-evaluate it in a different recursion of the algorithm)

Idea:
Recursively compute optimal parenthesization and use dynamic programming.
This means that starting from the smallest parenthesization then built it up to the top, ending up with a possible solution.

Definition 15. *Dynamic programming: a strategy in which the program store the partial result in a place, and use it any time it is needed.*

3.3 A dynamic programming solution

Store the minimum number of products for all the sub-chains in m.
To do so, we first consider the subset of the sequence of matrix that goes from i to j. At this stage we are not storing the parenthesization, i.e. the position in which we are going to place the parenthesis, but just evaluating the minimal number of product required. So we recursively compute $m[i, j]$ as:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{k\in[i,j-1]}\{m[j,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases} \tag{3.7}$$

The second term is well defined, and allow a minimum, because the two references to the matrix m are references in which the distances between the two indexes are smaller than the distance between i and j. So every time I am evaluating something in the matrix $m[i, j]$, I am reducing the distance between the indexes of the other two matrices $m[i, k]$ and $m[k + 1, j]$, ending up in a situation in which $i = j$ For each i, j also store in $s[i, j]$ the k that minimizes

$$m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \tag{3.8}$$

i.e. the parenthesization for the current level.

Example 8. *Consider $A_1(3 \times 5)$,$A_2(5 \times 510)$,$A_3(10 \times 52)$ and $A_4(2 \times 53)$ Each color on the matrix on the left represents different steps of the evaluation of parenthesization cost of the matrix chain. The matrix on the right is populated with the parenthesization once the minimal cost of product for a parenthesization placement is found. Both m and s can be computed iteratively from the shortest sub-chain to the longest: we can proceed from the mid diagonal to the top right corner.*

Algorithm 5: Dynamic Programming Solution

```
def MatrixChain(P):
    m$\leftarrow$ allocate (1..n, 1..n)
    s$\leftarrow$ allocate (1..n-1, 2..n)
    for i$\leftarrow$ 1..n do
        m[i, i] $\leftarrow$ 0
    end
    for l$\leftarrow$ 2..n do
        for i$\leftarrow$ 1..(n-l+1) do
            j $\leftarrow$ i + l -1
            MatrixChainAux(P, m, s, i, j)
        end
    end
    return  (m, s)
enddef
```

Algorithm 6: Dynamic Programming Solution

```
def MatrixChainAux(P, m, s, i, j):
    m[i,j] $\leftarrow$ INFINITY
    for k$\leftarrow$ 1..(j-1) do
        q $\leftarrow$ m[i,k] + m[k+1,j] + P[i-i] $*$ P[k] + P[j]
        if q<m[i,j] then
            m[i,j] $\leftarrow$ q
            s[i,j] $\leftarrow$ k
        end
    end
    return
enddef
```

The computation of $m[i,j]$ takes time:

$$\sum_{k=i}^{j-1} \Theta(1) = \Theta(j-i) \tag{3.9}$$

Since $i \in [1,n]$ and $j \in [i,n]$

$$T_C(n) = \sum_{i=n}^{n} \sum_{j=i}^{n} \Theta(j-i) = \Theta\Big(\sum_{i=1}^{n} \Big(\sum_{j=i}^{n} j\Big) - n*i\Big) \tag{3.10}$$

$$= \Theta\Big(\sum_{i=1}^{n} \frac{n*(n+1)}{2} - \frac{i*(i+1)}{2} - n*i\Big) = \Theta(n^3) \tag{3.11}$$

Chapter 4

Heaps

Definition 16. *Heaps: Abstract data types which store totally ordered values with respect to a given total order ($\preceq$)*

Example 9. *If I have a bunch of data regarding students, and I order it based on the students ID, that domain can be stored into a Heap*

Heaps efficiently support the following tasks:

- Built a heap from a set of data
- Finding the minimum with respect to $\preceq$
- Extracting the minimum with respect to $\preceq$
- Decreasing the one of the values with respect to $\preceq$
- Inserting a new value

Example 10. *A min-heap: a heap in which the relation between elements is "less" or "equal to" ($\preceq$ is $\leq$).*
A max-heap: a heap in which the relation between elements is "more" or "equal to" ($\preceq$ is $\geq$).

Heaps can be used to implement priority queues.
Queues elements respects the FIFO policy: the first object inserted in the queues will be the first object to exit from the queue. Sometimes however it is useful to establish a priority in the order to extract elements from a queue. To do so, the next element to be extracted from the queue is the one that minimizes a priority criterion.

Example 11. *In emergencies, more serious patient must be served first, and their condition may change and become more and more serious over time.*

4.1 Binary Heaps

Definition 17. *Binary heaps: A nearly completed binary tree. i.e. it is complete up to the second-last level and all leaves of the last level are on the left.*

The relation $parent(p) \preceq p$ holds for any node (head property).
A possible representation for a binary heap is using an array: the first position stores the root key.

Example 12. *Let's assume that the emergency room aforementioned can contain a maximum number of people. In that case we can organize the priority queues using an array, whose size corresponds to the number of places available.*

The values stored in the array would be the keys of the tree, while its position corresponds to the node. The first position of the array is the root.
The i-th position of the array represents a node whose:

- Left child has index $2 * i$
- Right child has index $2 * i + 1$
- Parent has index $i/2$

(In pseudo code the index starts from 1)

Following, the pseudo code for some useful function:

Algorithm 7: Array-based representation: useful functions

```
def LEFT(i):
    return 2∗ i
enddef
```

Algorithm 8: Array-based representation: useful functions

```
def GET_ROOT():
    return 1
enddef
```

Algorithm 9: Array-based representation: useful functions

```
def RIGHT(i):
    return 2 ∗ i + 1
enddef
```

Algorithm 10: Array-based representation: useful functions

```
def IS_ROOT(i):
    return i = 1
enddef
```

Algorithm 11: Array-based representation: useful functions

```
def PARENT(i):
    return floor (i/2)
enddef
```

Algorithm 12: Array-based representation: useful functions

```
def IS_VALID_NODE(H, i):
    return H.size ≤ i
enddef
```

4.2 Finding the Minimum

For minimum we mean the minimum with relation to the relation which parameterize our heap.
The minimum with relation to the $\preceq$ is in the root of the heap. If this is not the case, the heap property did not hold.

Algorithm 13: Array-based representation: minimum

```
def HEAP_MIN(H):
    return H.root.key
enddef
```

Which for array based representation became

Algorithm 14: Array-based representation: minimum (array)

```
def HEAP_MIN(H):
 |  return H[ 1]
enddef
```

In both case the complexity is $\Theta(1)$.

4.3 Removing the minimum

When removing the minimum, we have to

- Preserve the heap topological structure of the heap itself. To do so, we have to remove a leaf on the last level, the rightest more.
- Preserve the property. Removing the minimum means to remove the key to the root, which could create some problem.

The solution to achieve the removal is to replace the root's key with the rightmost leaf of the last level, then delete the rightmost leaf of the least-level, as its key is already stored. Doing so, the heap property may be lost (but only in one point).
To restore the property, we use a procedure called heapify.If consist in

- Find the node n, among the root and its children, whose key is minimum with relation to $\preceq$
- If the root's key is the minimum, we are done
- Otherwise, swap n's and root's keys
- Repeat on the sub-tree rooted on n

Correctness of HEAPIFY

Before the iteration: the heap property holds in T_1 and T_2.
After the iteration:

- The heap property still holds on T_2 and between a and b.
- T_1 has been messed up, but it is shorter than the original tree and all the keys on T_1 are greater than a.

We are pushing the problem constantly deeper, and at some point we end up either solving the problem, or in a leaf (where the problem is also solved).
Replacing the root's key cost $\Theta(1)$, because finding the rightest most node is simply finding the value which is in the last position of the array in this representation. Assigning the root key to that value is again easy, we just have to assign the value of the last position of the array, to the first position.
For each iteration of HEAPIFY:

- 2 comparison to find the minimum.

- 1 swap at most.

The distance from a leaf is decreased is decreased by one at each iteration. The total cost of HEAPIFY is the height of the heap $O(\log n)$

Algorithm 15: HEAPIFY

```
def HEAPIFY(H, i):
    m ← i
    for i in [LEFT(i), RIGHT(i)] do
        if IS_VALID_NODE(H, j) and H[j] $\preceq$ H[m] then
            m ← j
        end
    end
    if  i != m then
        swap(H, i, m)
        HEAPIFY(H, m)
    end
enddef
```

Algorithm 16: Array-based representation: remove minimum

```
def REMOVE_MIN(H):
    H[1] ← H[H.size]
    H.size ← H.size-1
    HEAPIFY(H, 1)
enddef
```

4.4 Building a Heap

Building the topology is easy because it can be done automatically once we decide to go for the array implementation of the heap. Once we have the heap topology, we can try to call the functions already implemented, which should preserve the heap topology. We cannot however call the `heapify` function on the root, because it can happen that the array is not yet ready to satisfy the precondition of the function.
Instead, we can call `heapify` from the bottom-up, starting from the leafs:

- Fix the heaps rooted on the second-last level with children.
- Fix the heaps rooted on the third-last level, and so on.

Complexity of Build_Heap

Let us focus on complete binary tree. How many nodes do they have at level i of the tree?
The answer is 2^i. If n is the number of nodes in our tree, then $n = \sum_{i=0}^{l} 2^i = 2^{l+1} - 1$. The number of leaves in a complete binary tree is then $\lceil \frac{n}{2} \rceil$ (ceiling of the fraction). This quantity holds also for nearly complete binary tree.
Let us consider the nodes at height h, how many h-heighted nodes has a nearly complete

binary tree having n nodes?
For height h we have $\lceil \frac{n}{2^{h+1}} \rceil$ nodes. What about the complexity of build_heap?
At each height h we have $\lceil \frac{n}{2^{h+1}} \rceil$ nodes. We want to fix the heap property by calling heapify on all the nodes of the tree. So the overall complexity, a.k.a. the complexity of invoking Build_Heap on n nodes

$$\begin{aligned} T_{BH}(n) &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot O(h) \\ &\leq c \cdot n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \\ &\leq c \cdot n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} \\ &\leq c \cdot n \cdot \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 \cdot c \cdot n \end{aligned}$$

So

$$T_{BH}(n) \in O(n) \tag{4.1}$$

4.5 Decreasing a Key

In a Min-heap we are talking about a decrease in the key of a node, in a Max-heap we are talking about an increase in the key of a node.
The heap property doesn't held anymore (in relation to the parent, for both cases): it preserve the heap property only on the sub-tree rooted to the node. Heapify solve the heap property on a sub-tree rooted on the node we are dealing with.
Swapping the keys of the node and its parent solves the problem on the sub-tree rooted on the parent (it preserve the relation with the sibling node). We could end up breaking the relation between the new parent node and its parent. In this case, we just need to reiterate the same procedure, repeating until the heap property is restored.
At each iteration we have to either:

- Ends the computation in time $\Theta(1)$ or
- Pushes the problem one step closer to the root in time $\Theta(1)$

Since the heap height is $\lfloor \log_2 n \rfloor$, the complexity is $O(\log n)$

Algorithm 17: Array-based representation: decreasing a key

```
def DECREASE_KEY(H, i, value):
    if H[i] $\preceq$ value then
        error(value+ "is not smaller than H["+i+"]"
    end
    H[i]$\leftarrow$ value
    while not (IS_ROOT(i) or H[PARENT(i)] $\preceq$ H[i]) do
        swap(H, i, PARENT(i))
        i$\leftarrow$ PARENT(i)
    end
enddef
```

4.6 Inserting a Value

Add the new value as the rightmost node (where we would remove a node). A new node N has to be added preserving the heap topology. We could set the key of N to the maximum value with relation to $\preceq$ (∞ or $-\infty$). Decrease than the key of N to the desired value.

Algorithm 18: Array-based representation: inserting a new value

```
def INSERT_VALUE(H, value):
    H.size $\leftarrow$ H.size + 1
    H[H.size]$\leftarrow$ $\infty_{\preceq}$
    DECREASE_KEY(H, H.size, value)
enddef
```

Has the same complexity of DECREASE_KEY: $O(\log n)$

Chapter 5

Retrieving Data and Sorting

5.1 Retrieving Data

If we have a batch of data, a very large database, and we want to extract data from it, because we are interested in only a specific instance, how can we do it?
Let's say we have and array $A =< a_1, ..., a_n >$ that contains some data. Each element of the array is associated to an **identifier**, that we will indicate as: $A[i].id$ (it is unique and different for all the values stored in the array). How can we find the data associated to the identifier id_1?

A Naive Solution

The easiest solution is to scan all the database searching for $A[i].id = id_1$. We start from the first value, we check if its id correspond to the one we are interested in, and if it is not, we go to the next element and so on.
Its asymptotic complexity in term of big-O is $O(n)$.

Dichotomic Search

If $A =< a_1, ..., a_n >$ is sorted with relation to the id's (i.e. if $i < j$, then $A[i].id \leq A[j].id$), then we can look at the element in the middle $A[n/2]$.

- If $A[n/2].id = id_1$ then we have done
- If $A[n/2].id > id_1$ then we focus on the 1st half of A$< a_1, ..., a_{n/2-1} >$
- If $A[n/2].id < id_1$ then we focus on the 2st half of A$< a_{n/2+1}, ..., a_n >$

Repeat until the array is empty, or the element is discovered.

Example 13. *Search for 2 in* $< -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 >$

Algorithm 19: Dichotomic Search

```
def di_find(A, a):
    (l,r) ← (1, $|A|$)
    while $r \geq l$ do
        m ← (l+r)/2
        if $A[m] = a$ then
            return m
        end
        if $A[m] > a$ then
            r ← m-1
        end
        else
            l← $m + a$
        end
    end
    return 0
enddef
```

At each iteration, $l - r$ is halved. So, if $|A| = 2^m$, `di_find` ends after m iteration at most (we for sure reduce the array to one single element).
The while-block takes time $\Theta(1)$.
The `di_find`'s complexity is $O(\log n)$.
However, let's remember that the dichotomic approach only works on sorted data. So the problem of sorting becomes of great importance.

5.2 Sorting

Input: An array A of numbers.
Output: The array A sorted i.e. if $i < j$, then $A[i] \leq A[j]$

Example 14.

A naive solution would be checking all possible permutations of the elements of the array, with complexity $\Omega(n!)$. There exist however many strategies to solve this very same problem.

5.3 Insertion Sort

We assume that an initial part of the array is already sorted. We want to increase the number of sorted elements in the array, and to do so we pick the next value after the already sorted part of the array, and put it in the right position. It is not so easy to do it in an array, because it is an immutable object. We can implement the insertion by swapping the considered value with the one in the previous in the array, and continue to do so until it reach the right position (comparing the value at each step).

Algorithm 20: Insertion Sort

```
def insertion_sort(A):
    for i in 2...$|A|$ do
        j ← i
        while (j>1 and $A[j] < A[j-1]$) do
            swap(A,j-1,j)
            j← j-1
        end
    end
enddef
```

The while-loop block costs $\Theta(1)$, swapping constant, and the following operation, takes constant time. The while-loop however is repeated a number of time up to i. The maximum number of iteration is when the while stop at its first condition, while the minimum is when the value is already in its correct position. The algorithm iterates $O(i)$ and $\Omega(1)$ times for all $i \in [2, n]$, the complexity is:

- The upper bound: $\sum_{i=2}^{n} O(i) * O(1) = O\Big(\sum_{i=2}^{n} i\Big) = O(n^2)$
- The lower bound: $\sum_{i=2}^{n} \Omega(1) * \Omega(1) = \Omega\Big(\sum_{i=2}^{n}\Big) = \Omega(n)$

5.4 Quick Sort

Quick Sort aim to improve the asymptotic complexity of the Insertion sort. It exploit the same approach of the dichotomic search. It

- Select one element of the A: the pivot (randomly).
- Split the array in two part based on the pivot (partition):
  - Sub-array S of the values smaller or equal to the pivot.
  - The pivot itself.
  - Sub-array G of the values greater than the pivot.
- Repeat in the sub-arrays having more than 1 elements (recursively).
- At the end of the iterations above:
  - The values in S stay in S even after sorting A
  - The values in G stay in G even after sorting A
  - The pivot is in its "*sorted*" position
  - S and G are shorter than A

An iteration places at least one element in the correct position. It prepares A for two recursive calls on S and G.

Algorithm 21: Quick Sort

```
def QUICKSORT($A$, $l = 1$, $r = |A|$):
    if l<r then
        p ← PARTITION(A, l, r, l)
        QUICKSORT(A, l, p-1)
        QUICKSORT(A, p+1, r)
    end
enddef
```

The last recursion call is a tail recursion (which can be substituted with a loop) An alternative version with less recursion is (with the same asymptotic complexity):

Algorithm 22: Quick Sort

```
def QUICKSORT($A$, $l = 1$, $r = |A|$):
    while l<r do
        p ← PARTITION(A, l, r, l)
        QUICKSORT(A, l, p-1)
        l← p+1
    end
enddef
```

From the execution time point of view, if it is possible to avoid recursion, it is better to do so.
The time complexity T_Q of quick sort will be

$$T_Q(|A|) = \begin{cases} \Theta(1) & \text{if } |A| = 1 \\ T_Q(|S|) + T_Q(|G|) + T_P(|A|) & \text{otherwise} \end{cases} \tag{5.1}$$

Where T_P is the complexity of the partition.

Partition

An In-Place Algorithm to partition the array A is:

- Switch the pivot p and the first element in A, then (in a closing maneuver) :
  - If $A[i] > p$, swap $A[i]$ and $A[j]$ and decrease j
  - Else ($A[i] \leq p$), increase i by 1
- Repeat until $i \leq j$
- Swap p and $A[j]$

The lower bound on the Partition, as it has to pass through all the elements of the array, is $\Omega(n)$. The general complexity is $\Omega(|A|)$ (or of the slice of the array we are considering).

Algorithm 23: Partition

```
def PARTITION($A$, $i$, $j$, $p$):
    swap(A, i, p)
    (p, i) ← (i, i+1)
    while i≤j do
        if $A[i] > A[p]$ then
            swap(A, i, j)
            j← j-1
        end
        else
            i← i+1
        end
    end
    swap(A, p, j)
    return j
enddef
```

Complexity

General Case:

$$T_Q(|A)| = T_Q(|S|) + T_Q(|G|) + \Omega(|A|) \tag{5.2}$$

But the complexity of the first two part depends on the size of S and G.
The worst case scenario is the one in which the two sub-arrays are unbalanced. After the partition S or G doesn't contain any value.
In that case $|G| = 0$ or $|S| = 0$ for all recursive call, so

$$\begin{aligned} T_Q(n) &= T_Q(n-1) + \Omega(n) \\ &= \sum_{i=0}^{n} \Omega(i) = \Omega\Big(\sum_{i=0}^{n} i\Big) \\ &= \Omega(n^2) \end{aligned}$$

Which is a telescopic equation.
The best case scenario is instead the one of a balanced partition a.k.a. the content of S and G preserve a given ration. To compute the complexity, we may use a recursion tree, in which every node, which represent a recursive call, specifying the cost of the call, taking into account the split ratio of the array. Each branch has different length, the smallest has length $\log_{ratio_1} n$ and the longest $\log_{ratio_2} n$ (with $ratio_2 > ratio_1$ and $ratio_1 + ratio_2 = 1$).
Every level until the length of the smallest branch has cost cn, while all the subsequent one has a decreasing cost $\leq cn$.

Example 15. *Recursive tree with fixed ratio:*

We want to sum up the cost of all levels.
The total is $\Theta(n \log n)$.

Finally, the average case depend on the ordering of A. If all the permutation are equally likely, the partition has the ration more balanced than $1/d$ with probability

$$\frac{d-1}{d+1} \tag{5.3}$$

However, if "good" and "bad" partition alternate, then we can conclude that the one consider is also a good scenario, because the average is $\Theta(n \log n)$

5.5 Finding the Maximum

In consist in:

- Find the maximum
- Move the maximum at the end of the array, then:
- If $|A| > 1$, repeat on the initial fragment of A

Example 16.

But how can we find the maximum? And how do we move it to the end of the array? The complexity is $\sum_{i=1}^{|A|}(T_{\mathtt{max}}(i) + \Theta(1))$, and depends heavily on the complexity required by the algorithm which find the maximum, which by the way depend on the length of the array it is considering at the moment, which decrease by one at each iteration of the algorithm (that's why it $T_{\mathtt{max}}$ is summed over i).

Bubble Sort

The first solution to the finding-the-maximum problem is the bubble sort algorithm. It work by pair-wise swapping the maximum to the right: it find the maximum by comparing each value with the one on its right, and swap the pair if the value on the right is smaller. Every iteration, bubble sort bring the highest value to the end of the array.

Algorithm 24: Bubble sort

```
def BUBBLE_SORT($A$):
    for i in $|A|$..2 do
        for j in 1..i-1 do
            if $A[j] > A[j+1]$ then
                swap(A, j, j+1)
            end
        end
    end
enddef
```

One swap-block costs $\Theta(1)$, so the nested for-loop costs $\Theta(i)$.

So the total cost of this operation would be:

$$\begin{aligned} T_B(n) &= \sum_{i=2}^{n} \Theta(i) * \Theta(1) \\ &= \Theta\Big(\sum_{i=2}^{n} i\Big) = \Theta(n^2) \end{aligned}$$

Best and worst case scenario are the same, and require the same time-complexity.

Selection Sort

The second solution to the finding-the-maximum problem is the selection sort algorithm. It work by linear scanning the unsorted part: it find the maximum among all the values by doing a linear search (the naive algorithm shown at the beginning of the chapter), and perform one single swap.

Algorithm 25: Bubble sort

```
def SELECTION_SORT($A$):
    for i in $|A|..2$ do
        max_j ← 1
        for j in 2..i do
            if $A[j] > A[max\_j]$ then
                max_j ← j
            end
        end
        swap(A, i, j+1)
    end
enddef
```

One if-block costs $\Theta(1)$, so the nested for-loop costs $\Theta(i)$.
So the total cost of this operation would be:

$$\begin{aligned} T_S(n) &= \Theta(1) \sum_{i=2}^{n} \Theta(i) * \Theta(1) \\ &= \Theta\Big(1 + \sum_{i=2}^{n} i\Big) = \Theta(n^2) \end{aligned}$$

Best and worst case scenario are the same, and require the same time-complexity. We decreased the number of swap necessary in a constant way, so overall selection sort has the same complexity of the bubble sort.

Heap Sort

The second solution to the finding-the-maximum problem is the heap sort algorithm. It works by using the max-heap $H_{\texttt{max}}$:

- Store the elements of A in $H_{\texttt{max}}$: building up an heap which stores all the values of the array (which has linear complexity).
- Extract the min(i.e. the maximum) and plate in in A. It takes time constant respect to the heap itself.
- Repeat from 2 until $H_{\texttt{max}}$ in not empty.

The array-based representation of heaps is an in-place algorithm.

Algorithm 26: Heap sort

```
def HEAP_SORT($A$):
    H← BUILD_MAX_HEAP(A) // the root is the max
    for i in $|A|..2$ do
        A[i] ← EXTRACT_MIN(H)
    end
enddef
```

Let's now look at the heap sort complexity.
BUILD_MAX_HEAP procedure costs $\Theta(n)$. EXTRACT_MIN costs $O(\log i)$ per iteration, in which i is the number of elements in the heap (decreasing every time by 1). The total is

$$\begin{aligned} T_H(n) &= \Theta(n) + \sum_{i=2}^{n} O(\log i) \\ &\leq O(n) + O\Big(\sum_{i=2}^{n} \log n\Big) = O(n \log n) \end{aligned}$$

This is an upper-bound, for the worst case scenario. The actual number of iteration is different for every case.
The overall complexity of heap sort is $O(n \log n)$.
Heap sort is overall faster than quick sort, because in its worst scenario quick sort takes a quadratic time.

5.6 Sorting in Linear Time

Is it possible to improve again the efficiency of sorting?
To find the lower bound, we need to consider all the possible computation made up by an algorithm. For this reason, the execution of a sorting-by-comparison algorithm can be modeled as a decision-tree model: given an algorithm and an array, a decision tree represent all the possible computation of the algorithm over that array. Any comparison between a and b elements of the array corresponds to a node, which branches the computation according to whether $a \leq b$ or $a > b$

Example 17.

In this way we can model every possible algorithm, dealing with every possible array. Since every leaves represent a permutation of the original array, the number of leaves in a decision tree is always factorial with respect to the number of elements in the array. The height h is the maximum number of comparisons required by the algorithm. Since a binary tree has no more than 2^h leaves,

$$h \geq \log_2(n!) \in \Omega(n \log n) \tag{5.4}$$

The lower bound for comparison-based sorting is then $\Omega(n \log n)$. For this reason, there is no general algorithm to sort in linear time by using comparisons.
However, we can introduce some minor ad-hoc assumptions that allows us to lift this bound. Those are:

- Bounded domain for the array values.
- Uniform distribution of the array values.

Counting Sort

Counting Sort is an algorithm which doesn't use any comparison, but is nevertheless able to sort values in the interval $[1, k]$. The idea is

- Count the occurrences of A's values and place them in C. This means: take the unsorted array A, and count the number of repetition of values in each array by counting an auxiliary matrix C, whose length is the difference between the first (min) value of the array A, and the last (max) value of the array A. Each of the indexes in C represent a value in A. To populate C we can just do a linear scan of the array A, updating every time we find a new value.
- Sums the values in C and get the number of elements smaller of equal to the C's indexes. This means we are performing a pulling operation: accumulate in each entries of C the number of values smaller then the indexes. In this way, for each value in the array A, the we have the number of values in A which are smaller or equal to the value itself. From the update C we know the correct position of the value in the array.
- Use C to place the elements of A in the correct positions in B. To avoid change in position due to swapping, we start from the end of the array A, building a third array B (to maintain the relative order between equal values) . Also, every time we place a value of A in B, we decrease by 1 the corresponding value in the updated C.

Example 18.

Counting sort is not an in-place algorithm, the auxiliary space necessary is not constant, and we and up with a new array separated from the original A. Stability is the preservation of relative order of equivalent elements. We place the value backwardly to preserve it.

Generalizing the algorithm to deal with any interval $[k_1, k_2]$ domain is easy.

Algorithm 27: Counting sort

```
def COUNTING_SORT($A$, $B$, $k$):
    C$\leftarrow$ ALLOCATE_ARRAY(k, default_value = 0)
    for i $\leftarrow$ 1 upto $|A|$ do
        C[A[i]] $\leftarrow$ C[A[i]] + 1
    end
    // C[j] is now the number of j in A
    for j $\leftarrow$ 2 upto $|C|$ do
        C[j] $\leftarrow$ C[j-1] + C[j]
    end
    // C[j] is now the number of j in A's value $\leq$j
    for i $\leftarrow$ $|A|$ downto 1 do
        B[C[A[i]]] $\leftarrow$ A[i]
        C[A[i]]$\leftarrow$ C[A[i]]-1
    end
enddef
```

Let's consider the complexity of the algorithm:

- Allocating C and setting all its element to 0: $\Theta(k)$
- Counting the instances of A's values: $\Theta(n)$
- Setting in $C[j]$ the number of A's values $\leq j$: $\Theta(k)$
- Copying A's values into B by using C: $\Theta(n)$

So the total is: $\Theta(n + k)$

Radix Sort

It perform the sorting of numbers which have the same number of digits. An array A of d-digit values can be sorted digit-by-digit:

- For each digit i from the rightmost down to the leftmost.
- Use a stable algorithm and sort A according to the digit i.

The way which you sort the number digit-by-digit is stable, i.e. assuming that the relative order of the equal values are not sorted during the sort.
This is the ideal case to use counting sort. If therefore the digit sorting is in $\Theta(|A| + k)$, radix sort takes time:

$$\Theta(d(|A| + k)) \tag{5.5}$$

where d is the number of digits in each of A's values.

Bucket Sort

If we are able to provide the uniformity assumption to the distribution of the values in the array, i.e. the probability of getting a number is always the same, we can use the Bucket sort. The idea is to:

- Split $[0,1)$ in n buckets: $\left[\frac{i-1}{n}, \frac{i}{n}\right)$ for $i \in [1,n]$. This means that we create an array of buckets which contains all the values of the given array in a given interval. We divide the original interval in some sub-interval, and place every value in the corresponding bucket.
- Add each value of A to the correct bucket.
- Sort the bucket: we have to sort all elements in the same buckets, but as we have assumed an uniform distribution, the probability of having more than one value in each bucket is very low.
- Reverse the content of buckets in bucket order on A.

The complexity is:

- Allocating an initializing B: $\Theta(n)$
- Filling the buckets: $\Theta(n)$
- Sorting each bucket(expected): $O(n)$
- Reversing buckets content into A: $\Theta(n)$

Total expected complexity: $O(n)$