University of Trieste

International School for Advanced Studies

The Abdus Salam International Centre for Theoretical Physics

Algorithmic Design

Lectures Notes

Author:
Marco Sciorilli

Friday 4th June, 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. We can give an high level set of instructions to turn the input of the function into the output).
A function which 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 ($\uparrow$)*

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

It is not possible to implement h.

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

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

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 constants*
- *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:*

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

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:*

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

It gives a lower bound.

Example 4. *Examples of functions' Ω(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:*

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

Theorem 1.

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

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. In directed graphs, all edges has a direction: e.g. if we can go from one node to another, it may not be the case the other way around. In an undirected graph there are no such limitations.

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

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

Definition 14. *Some definitions:*

- *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 connected component of a undirected graph G is a maximum connected sub-graph of G.*
- *A graph is acyclic if it does not contains cycles.*
- *Directed Acyclic Graphs are also known ad DAGs: i.e. a graph in which every edge is directed, and there is no cycle.*

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 15. *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← 1..rows(A) do
        for j ← 1...cols(B) do
            a← 0
            for k← 1..cols(A) do
                a← a + A[i,k] * B[k,j]
            end
            C[i,j] ← 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 multiplication
- 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

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

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

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

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 operations 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 operations 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 of 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 it. As we are doing it at design time, when usually there are no time constrains.

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

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

  for any $k \in [1, n-1]$. We 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 16. *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 matrices 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:

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

The second term is well defined, and allows 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 we are evaluating something in the matrix $m[i, j]$, we are 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

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

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← allocate (1..n, 1..n)
    s← allocate (1..n-1, 2..n)
    for i← 1..n do
        m[i, i] ← 0
    end
    for l← 2..n do
        for i← 1..(n-l+1) do
            j ← 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] ← INFINITY
    for k← i..(j-1) do
        q ← m[i,k] + m[k+1,j] + P[i-1] * P[k] + P[j]
        if q<m[i,j] then
            m[i,j] ← q
            s[i,j] ← 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 17. *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 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 18. *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 most.
- Preserve the property. Removing the minimum means to remove the key to the root, which could create some problems.

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 last-level. 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 comparisons to find the minimum.
- 1 swap at most.

The distance from a leaf 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 j in [LEFT(i), RIGHT(i)] do
        if IS_VALID_NODE(H, j) and H[j] ⪯ 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 trees. 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] ⪯ value then
        error(value+ "is not smaller than H["+i+"]"
    end
    H[i]← value
    while not (IS_ROOT(i) or H[PARENT(i)] ⪯ H[i]) do
        swap(H, i, PARENT(i))
        i← 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 ← H.size + 1
    H[H.size]← ∞_⪯
    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≥ 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
```

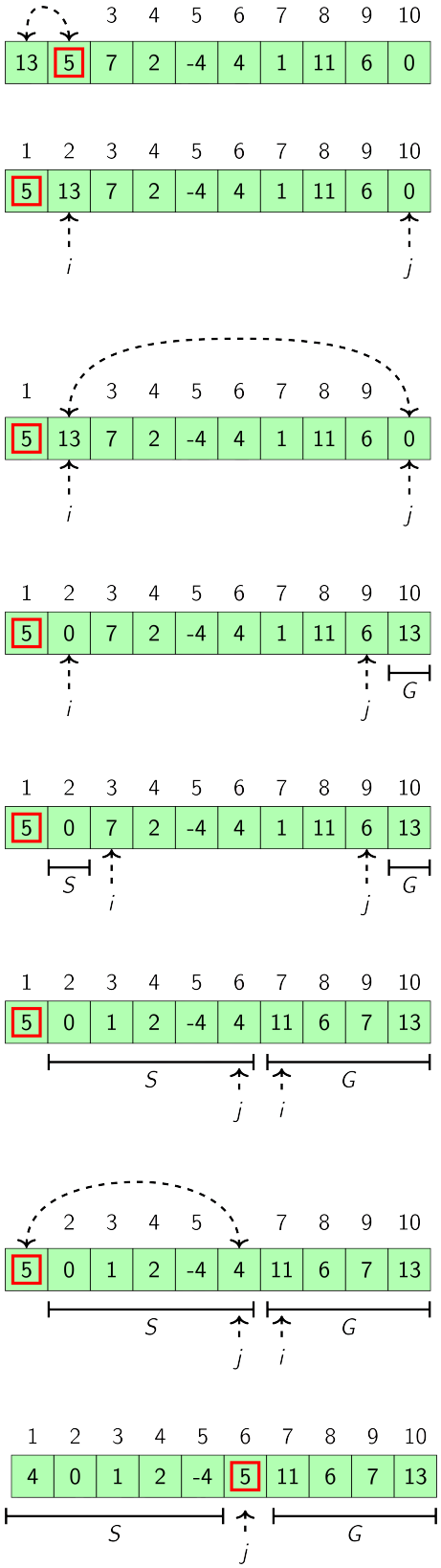


Figure 5.1: Partition example

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 $\sum_{i=1}^{|A|}(T_{\texttt{max}}(i) + \Theta(1))$, and depends heavily on the complexity required by the algorithm which find the maximum, which by the way depends 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 $T_{\texttt{max}}$ 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 works by linear scanning the unsorted part: it finds the maximum among all the values by doing a linear search (the naive algorithm shown at the beginning of the chapter), and

performs 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 third 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 place it in A. It takes constant time 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, we have the number of values in A which are smaller or equal to the value itself. From the updated 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 changes 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.

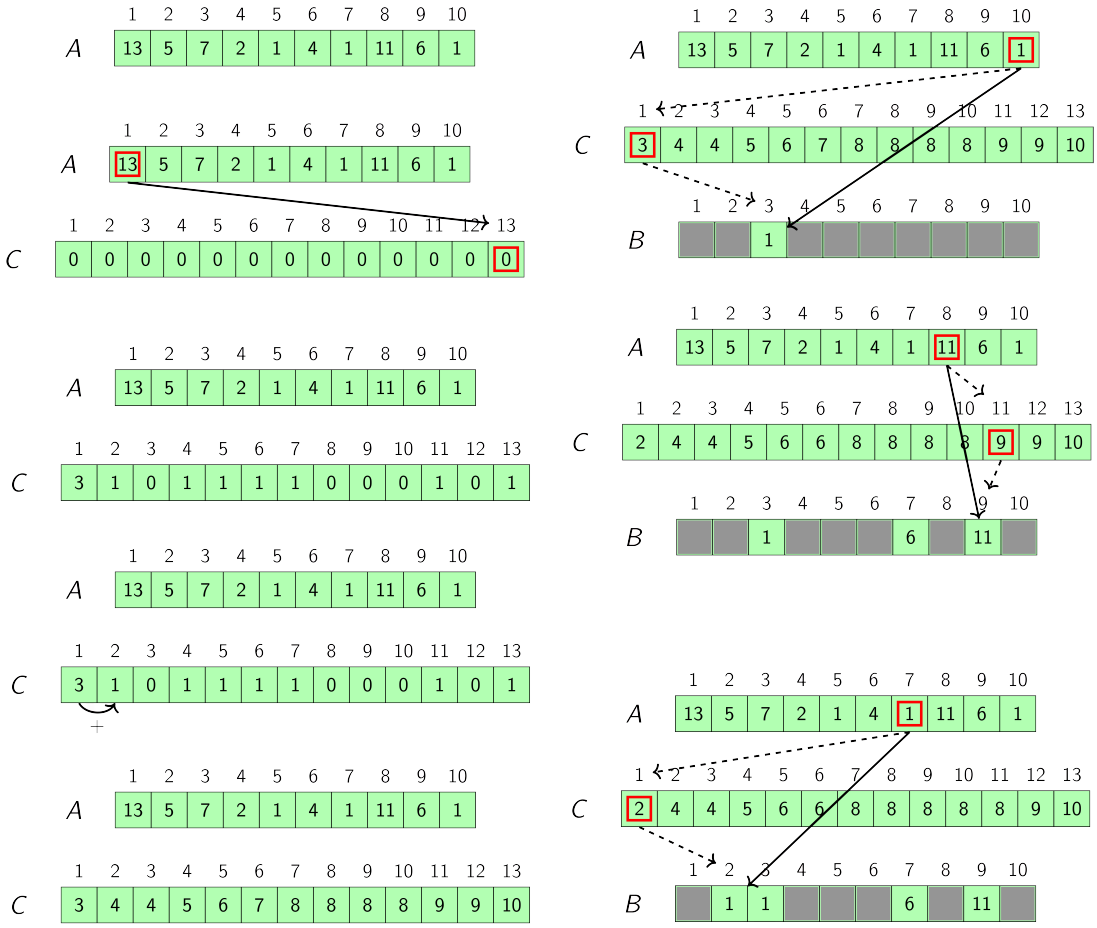


Figure 5.2: Counting sort example

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← ALLOCATE_ARRAY(k, default_value = 0)
    for i ← 1 upto |A| do
        C[A[i]] ← C[A[i]] + 1
    end
    // C[j] is now the number of j in A
    for j ← 2 upto |C| do
        C[j] ← C[j-1] + C[j]
    end
    // C[j] is now the number of j in A's value ≤j
    for i ← |A| downto 1 do
        B[C[A[i]]] ← A[i]
        C[A[i]]← 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 performs 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-intervals, 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 and 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: $O(n)$

5.7 Select

Let A be an array which may be unsorted. How could we find the value that, if A was sorted, would be in position:

- 1 i.e. the minimum value in A. We could scan all the array and search for the minimum. This requires to scan all the elements of A, so its minimum complexity is $\Theta(n)$.
- n i.e. the length of the array itself. In this case it means to search the maximum value, so as before the complexity is $\Theta(n)$.
- $i \in [1, n]$ i.e. any other value in the array. A straight forward strategy would be to sort the array A itself, and return the value in position i. Its complexity is then $O(n \log n)$.

Is it possible to deliver an algorithm with a better complexity?

Definition 19. *The Select Problem:*
Input: *a potentially unsorted array A and an index $i \in [1, |A|]$*
Output: *the value $\overline{A}[i]$ where $\overline{A}$ is the sorted version of A.*

Which can be reformulated in a different, more easy to handle form:

Definition 20. *The Select Problem:*
Input: *a potentially unsorted array A and an index $i \in [1, |A|]$*
Output: *an index j s.t. $\tilde{A}[j] = \overline{A}[i]$ where $\tilde{A}$ and $\overline{A}$ are A after the computation and the sorted version of A, respectively.*

We do not want to build and index: we are considering an una-tantum query (i.e. one specific position in A, not the case of answering the selection problem many consecutive times.)
We will also assume that A does not contains multiple instances of the same value (this is not necessary, but simplify things when dealing with the complexity of the algorithm).

A possible strategy

We could try be using a PARTITION (Algorithm 23) approach, inherited from the dichotomic approach. Let's say we want to find the value stored in position i, the algorithm consists in:

- Select a pivot $A[j]$
- Compute $k \leftarrow PARTITION(A, 1, |A|, j)$, and gets S and G
- Compare i and k:
  - If $i = k$, then return k
  - If $i < k$, then $\overline{A}[i]$ is in S
  - If $i > k$, then $\overline{A}[i]$ is in G

Here anyway we are facing the same problem already saw in QUICKSORT: the pivot choice makes the difference. What if we end up in a situation in which the pivot end up with a partition which is always the first or last element in the array?
In this case the complexity of the algorithm is $\Theta(n^2)$, that is not what we want to do. The best pivot is the one that is median in $\overline{A}$, but in our case A may be unsorted. But constant ratio partitions are QUICKSORT's best scenarios as well. We want an alternative to the median of $\overline{A}$ which guarantees us a constant ratio between the partition of A. We can find an almost-median in this way (getting the median of the medians):

- Split A in $[n/5]$ chunks $C_1, ...C_{n/5}$ each of size 5. We know that we can sort each of this array in constant time, since the length is constant.
- Find the median m_i of C_i, e.g., by sorting C_i itself. Sorting all of that takes a time linear to the number of chunks, which is linear to the length of the array. But once we have sorted each of the chunks, we can find the median of the chunk in a constant time. This operation is done in time $\Theta(n)$.
- Recursively compute the median m of the m_i's. Let's assume we have now all the medians in an array, we want to find the median of them. We have decrease the size of the array to $\lceil n/5 \rceil$. We can apply recursion.

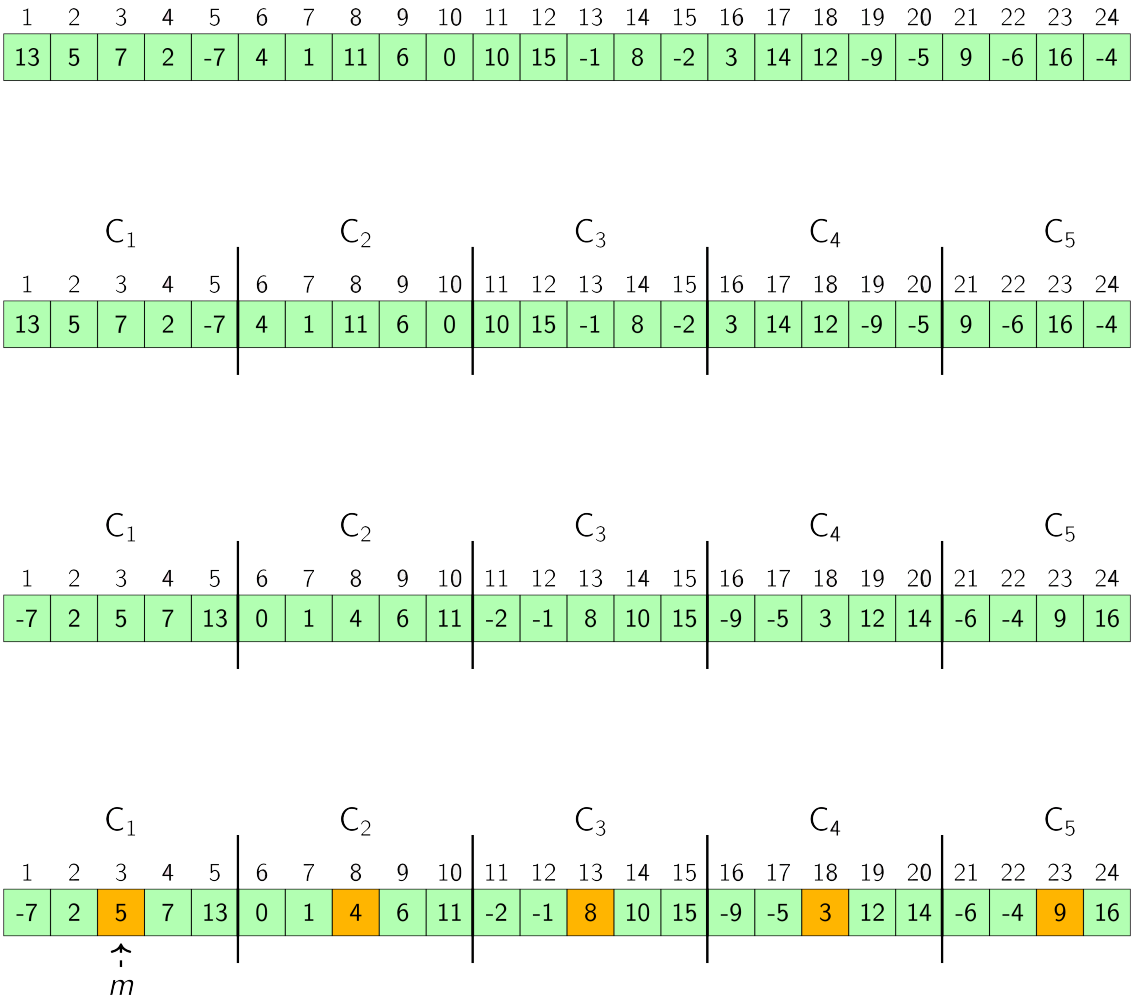


Figure 5.3: Select example

An algorithm which exploits this trick has a complexity which is below $n \log(n)$?
The worst case scenario is the one in which the median of the median is not a good enough pivot for PARTITION. Let's think the chunks as they were the columns of a matrix.
We can then sort the chunks according to the median. This representation lets us count how many values end up in G and in S.

- The number of chunks are of course $\lceil \frac{n}{5} \rceil$.
- How many medians are greater than the median of the medians? $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$
- How many chunks at least have 3 elements greater than m? $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2$
- How many elements at least are greater than m? $3\left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right)$
- This is the number of values that for sure are greater than the median of the median: m? $3\left(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \right) \geq \frac{3n}{10} - 6$

Figure 5.4: Select matrix

The same calculations can be carried out for S. An upper bound for the number of

elements smaller or equal to m (the worst case scenario) is: $3n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$.
Let's compute the complexity to get the pivot for the partition (the median of the median):

$$T_s(n) = \Theta(n) + T_s\Big(\lceil \frac{n}{5} \rceil\Big) + \Theta(n) + T_s\Big(\frac{7n}{10} + 6\Big) \tag{5.6}$$

The last two terms are the parts that account for the partition + recursion. We can solve $T_s(n)$ by using the Substitution method:

- We guess that $T_s(n) \in O(n)$
- Replace all the asymptotic classes by representative. So

$$T_s(n) = T_s\Big(\lceil \frac{n}{5} \rceil\Big) + T_s\Big(\frac{7n}{10} + 6\Big) + c' \cdot n \tag{5.7}$$

- Prove by induction:
  - Assume that $\exists c > 0$ such that

$$\forall m < n \;\; T_s(m) \leq c \cdot \Big(\lceil \frac{m}{5} \rceil\Big) + c \cdot \Big(\frac{7}{10}m + 6\Big) + c'm \tag{5.8}$$

  - Need to prove that holds for m

$$\begin{aligned} T_s(n) &\leq T_s\Big(\lceil \frac{m}{5} \rceil\Big) + T_s\Big(\frac{7}{10}n + 6\Big) + c'm \\ &\text{if } n > \frac{7}{10}n + 6 \\ &\leq c \cdot \lceil \frac{n}{5} \rceil + c \cdot \Big(\frac{7}{10}n + 6\Big) + c'n \\ &\leq c \cdot (n/5 + 1) + c \cdot \frac{7}{10}n + (c \cdot 7 + c'n) \\ &\leq \frac{9}{10} \cdot c \cdot n + c'n + 7 \cdot c \end{aligned}$$

    So at the end

$$\Big(\frac{9}{10} + c'\Big) \cdot n + 7c \leq c \cdot n \tag{5.9}$$

    And this holds for $c \geq 20c'$ and $n \geq 140$. So $T_s(n) \leq cn$

Algorithm 28: SELECT

```
def SELECT(A, l=1, r=|A|, i):
    if r-l≤10 then // base case
        SORT(A, l, r)
        return i
    end
    j←SELECT_PIVOT(A, l, r)
    k←PARTITION(A, l, r, j)
    if i=k then // dichotomic approach
        return k
    end
    if i<k then // search in S
        return SELECT(A, l, k-1, i)
    end
    // search in G
    return SELECT(A, k+1, r, i)
enddef
```

Algorithm 29: SELECT_PIVOT

```
def SELECT_PIVOT(A, l=1, r=|A|):
    if r-l≤10 then // base case
        SORT(A, l, r)
        return (l+r)/2
    end
    chunks←(r-l)/5
    for c in 0..chunks-1 do // for each chunk
        (c_l, c_r)←(1, 5) + c ∗ 5
        SORT(A, c_l, c_r) // sort it
        SWAP(A, c_l+2, l+c) // place the middle element at the beginning
         of A
    end
    // recursive step
    return SELECT(A, l, l+chunks-1, chunks/2)
enddef
```

Chapter 6

Binary Search Trees and Red-Black Trees

6.1 Motivation

Example 19. *Let us consider the registry office, for which for every newborn we want to record a set of data (e.g. name, birthday, parents, etc.). The registry change quite often. What if we want to perform a search based on one of this data?*

If we use the array based scenario, but every time we update the data we have to build up a new array, and erase the precedent one. For searching we can:

- Scan all the data-set at each query, this takes time $\Theta(n)$
- Sort the data-set by the chosen data (e.g. birthday, in the example) at each insertion. This takes time $\Theta(n)$ per insertion (using the Radix Sort, because the birthday has a fixed number of digits) + $\Theta(\log n)$ (using a dichotomic search) per query. But what if we are interested in the order of a different data (e.g. search by name)?
- Sort the data-set by the chosen data (e.g. birthday, in the example) at each query. This takes time $\Theta(n+\log n) = \Theta(n)$ per query (using the Radix Sort and dichotomic search)

So there each strategies always lands on a complexity of approximately $\Theta(n)$. For this kind of registry, the array is not the best data structure. What we are looking for is a data structure that provides:

- Adding new data
- Searching data
- Removing data

We are aiming to build an index i.e. an auxiliary data structure to **efficiently** perform those operations.

6.2 Binary Search Trees

We already encountered trees

Definition 21. *Binary Search Trees (BST) are an abstract data type (ADT). The following notation holds*

- `x` *is a node*
- `x.left` *and* `x.right` *are a node* `x` *left and right child respectively*
- `x.parent` *is the parent of the node* `x`
- `x.key` *is the value sorted in* `x`*, i.e. its key*
- `x.left=NIL`*, then* `x` *misses the left child (i.e. the reference is not present)*
- `T.root` *is the root of the tree* T *and* $T.root.parent = NIL$ *(i.e. the root has no parent, its reference is missing)*

Utility function

Some useful functions with complexity $\Theta(1)$

Algorithm 30: BST: IS_ROOT

```
def IS_ROOT(x):
    return x.parent=NIL
enddef
```

Algorithm 31: BST: IS_RIGHT_CHILD

```
def IS_RIGHT_CHILD(x):
    return ¬ IS_ROOT and x.parent.right=x
enddef
```

Algorithm 32: BST: SIBLING

```
def SIBLING(x):
    if IS_RIGHT_CHILD(x) then
        return x.parent.left
    end
    return x.parent.right
enddef
```

Algorithm 33: BST: CHILDHOOD_SIDE

```
def CHILDHOOD_SIDE(x): // get x's side w.r.t its parent
    if IS_RIGHT_CHILD(x) then
        return RIGHT
    end
    return LEFT
enddef
```

Algorithm 34: BST: REVERSE_SIDE

```
def REVERSE_SIDE(side): // reverse the side
    if side=LEFT then
        return RIGHT
    end
    return LEFT
enddef
```

Algorithm 35: BST: GET_CHILD

```
def GET_CHILD(x, side): // get x's child on side
    if side=LEFT then
        return x.left
    end
    return x.right
enddef
```

Algorithm 36: BST: SET_CHILD

```
def SET_CHILD(x, side, y): // set x's child
    if side=LEFT then
        return x.left ← y
    end
    else
        return x.right← y
    end
    if y≠NIL then
        y.parent← x
    end
enddef
```

Algorithm 37: BST: UNCLE

```
def UNCLE(x):// get x's uncle
    return SIBLING(x.parent)
enddef
```

Algorithm 38: BST: UNCLE

```
def GRANDPARENT(x):// get x's granpa
    return x.parent.parent
enddef
```

Algorithm 39: BST: NEW_NODE

```
def NEW_NODE(v):// build a new node
    x.key←v
    x.parent ←NIL
    x.right←NIL
    x.left←NIL
enddef
```

A BST is a tree s.t.:

- All the keys belong to a totally ordered set with relation to $\preceq$.
- If x_l is the left sub-tree of x, then $x_l.key \preceq x.key$: all the nodes on the left sub-tree of a node should be smaller or equal respect to the total order to the node itself .
- If x_r is the right sub-tree of x, then $x.key \preceq x_r.key$: all the nodes on the right sub-tree of a node should be greater or equal respect to the total order to the node itself.

Binary search tree are used to represent an index, so we usually avoid to have copies of the same key. To have multiple objects with the same key, the best thing is to attach a list to every key with the corresponding object.

Printing, Finding Min/Max

How can we print in order all the nodes of a BST? Using a recursive algorithm: for every node, all keys which are smaller or equal then the key of the node considered at the moment belongs to the left subtree, so we first need to visit the left subtrees, until we find the smallest. Then we keep going right until all nodes are printed.

Algorithm 40: BST: INORDER_WALK:INORDER_WALK_AUX

```
def INORDER_WALK_AUX(x):
    if x≠NIL then
        INORDER_WALK_AUX(x.left)
        print x.key
        INORDER_WALK_AUX(x.right)
    end
enddef
```

Algorithm 41: BST: INORDER_WALK:INORDER_WALK

```
def INORDER_WALK(T):
    INORDER_WALK_AUX(T.root)
enddef
```

Due to the BST property:

- The minimum key is contained by the first node on the leftmost branch that has not a left child (we always have 2 child nodes on the tree, so we just has to follow, starting from the root, the left branch until no node is found).
- The maximum key is contained by the first node on the rightmost branch that has not a right child (we always have 2 child nodes on the tree, so we just has to follow, starting from the root, the right branch until no node is found).

Algorithm 42: BST: MINIMUM_IN_SUBTREE

```
def MINIMUM_IN_SUBTREE(x):
    while x.left ≠ NIL do
        x ← x.left
    end
    return x
enddef
```

Algorithm 43: BST: MAXIMUM_IN_SUBTREE

```
def MINIMUM_IN_SUBTREE(x):
    while x.right ≠ NIL do
        x ← x.right
    end
    return x
enddef
```

The complexity depends on the length of the tree itself: for both algorithms, the maximum number of steps we need to perform is equal to the length of the longest branch on the tree itself. So it is $O(h_T)$.

Find the successor of a node

We want to find the node whose key is the next key in the total order with respect to the key we are considering at the moment. Due to the BST property, the successor with relation to $\preceq$ of node n is either

- The node containing the minimum in the right sub-tree of n.
- The nearest "right-ancestor" (i.e. go up until we end up with a root whose previous node was a right child of the node we are dealing with) of n, if n has not right child.

Algorithm 44: BST: SUCCESSOR

```
def SUCCESSOR(x):
    if x.right≠NIL then
        return MINIMUM_IN_SUBTREE(x.right)
    end
    y←x.parent
    while y≠NIL and IS_RIGHT_CHILD(x) do
        x←y
        y←x.parent
    end
    return y
enddef
```

The complexity is $O(h_T)$, for the worst case scenario, where we follow all the path up to the root.

Searching for a value

To search for a value v, we apply a dichotomic approach from the root x

- If n is NIL or $x.key = v$, return n: if the root has the value we are looking for, we are good.
- If $x.key \preceq v$, search on the right sub-tree
- If $x.key \npreceq v$, search on the left sub-tree

Algorithm 45: BST: SEARCH_SUBTREE

```
def SEARCH_SUBTREE(x, v):
    while x≠NIL do
        if x.key⪯v then
            if v⪯x.key then
                return x
            end
            x← x.right
        end
        else
            x←x.left
        end
    end
enddef
```

Remark. The strategy is the same of the dichotomic search.

As before, the worst scenario is the one in which we follow the route of the longest branch. Each iteration performs $\Theta(1)$ operations. The number of iterations depends on the height h_T of T and on v, so the algorithm takes time $O(h_T)$.
What is the relations between the number of nodes in tree and its height? We first need to see how to insert a value in a BST.

Insert a value

Browse a branch of T and add a new leaf having v as a key. Once we discover that the corresponding node is absent, we insert the node with that key value.

Algorithm 46: BST: INSERT_BST

```
def INSERT_BST(T, v): // v is the new value
    x←T.root
    y←NIL // y is x's parent
    // search the right place for z
    while x≠NIL do
        y←x
        if x.key⪯v then
            if v⪯x.key then
                return HANDLE_MULTI_INSERT(x,v)
            end
            x← x.right
        end
        else
            x←x.left
        end
    end
    // attaching the new node
    x←NEW_NODE(v)
    if T.root≠NIL then
        if v ⪯ y.ket then
            SET_CHILD(y, LEFT, x)
        end
        else
            SET_CHILD(y, RIGHT, x)
        end
    end
    else
        T.root←x
    end
    return x
enddef
```

As before, the complexity is $O(h_T)$.

Removing a Key

Search the node x containing the key. We end up in two cases:

- x has one child at most. In this case we erase the node itself and replace with its only child. This operation is named transplant. The cost of this operation is constant, we simply need to re-assign the parent of the only child of the node, and

the child of the parent node.

- x has two children. In this case we can search for the successor of this node (the minimum on the right subtree). It surely does not have a left child. So we can substitute the value of the key of the new found node to the one we want to erase, and this preserve the order of the BST. We need then to erase the redundant node, and we can do it by using the transplant function.

Algorithm 47: BST: TRANSPLANT

```
def TRANSPLANT(T, x, y):// replace x by y
    if IS_ROOT(x) then
        T.root←y
    end
    if y≠NIL then
        y.parent←NIL
        // else x has no parent
    end
    else
        x_side←CHILDHOOD_SIDE(x)
        // attach y in place of x
        SET_CHILD(x.parent,x_side,y)
    end
enddef
```

Algorithm 48: BST: REMOVE_BST

```
def REMOVE_BSTT(T, x):// remobe x.key from T and return a removed
 node
    if x.left=NIL// if x lacks of left child then
        TRANSPLANT(T, x, x.right)
        return x
    end
    if x.right=NIL// if x lacks of right child then
        TRANSPLANT(T, x, x.left)
        return x
    end
    y← MINIMUM_IN_SUBTREE(x.right)
    x.key←y.key
    return REMOVE_BST(T,y)// if y lacks of left child
enddef
```

As before, the complexity of removing a node x takes time $O(h_T)$.
So, again, what are the relations between the number of nodes in the tree and its height? h_T may be equal to the number n of nodes (e.g. keeping inserting the maximum). So BTSs costs more than single-linked list (insertion $\Theta(1)$).
Are we able to provide some guidelines able to reduce the height of a binary search tree? Or better, are we able to balance the height of a binary search tree in a way in which

the ration between the longest and shortest branch in the tree is always below a given threshold?
Yes, with the use of Red-black trees.

6.3 Red-Black trees

Red-Black trees (RBTs) are a datatype meant to produce a dynamic index for a set of data, built as the BST, but balanced: the ratio between the longest and shortest branch in the tree is upper-bounded by 2.

Definition 22. *RBTs are BSTs satisfying the following conditions:*

- *Each node is either a RED or BLACK node.*
- *The tree's root is BLACK.*
- *All the leaves are BLACK NIL nodes.*
- *All the RED nodes must have BLACK children.*
- *For each node x, all the branches from x (down to a leaf) contain the same number of black nodes.*

Definition 23. *Black height (BH(x)): the number of BLACK nodes below x (down to a leaf) in any branch. Thanks to the last condition on RBTs, we do not need to specify anything on the branch.*

To have a totally balanced tree, we have to deal with a complete tree (at most 2^{h_T} nodes). But we also have to deal with the case in which the tree is not complete: the maximum length of a branch is always constrained from above by some linear bound on the logarithm of the number of the nodes. Let's compute the bound on the height of RBT:

- $n \geq 2^{BH(x)} - 1$: if we have a RBT rooted on the node x, and we consider exclusively the internal nodes (all but the leaves), the number of nodes n contained in the tree must be $n \geq 2^{BH(x)} - 1$. Let's consider the case in which we count all the nodes but the RED ones. Since all branches has the same BH, we end up with a complete tree, whose height is the one to the root. The height of the all-black tree is $2^{BH(x)} - 1$.
- $h(x) \leq 2 \cdot BH(x)$: this is caused by the definition. i.e. every time we end up on a red node, it has to have black children, but it might be the case we do not have any red node.

So we deduce that

Theorem 2. *Any RBT with n internal nodes has height at most $2\log_2(n+1)$:*

$$
\begin{aligned}
& n \geq 2^{\frac{h(x)}{2}} - 1 \\
& \Leftrightarrow \log_2(n+1) \geq \frac{h(x)}{2} \\
& \Leftrightarrow h(x) \leq 2 \cdot \log_2(n+1)
\end{aligned}
$$

The height of any node $h(x) \in \log(n)$, because in the worst case scenario the ratio between the shortest and longest branch is 2 (because BH is always equal).
Function to get the color of a node

Algorithm 49: RBT: COLOR

```
def COLOR(x)
    if x=NIL then
        return BLACK
    end
    return x.color
enddef
```

We want to insert nodes in the RBT preserving the properties of the tree.
We needs some useful operations which are meant to re-balance the tree after the insertion.

Definition 24. *Rotations: change the length of branches while preserving the BST properties.*

e.g.

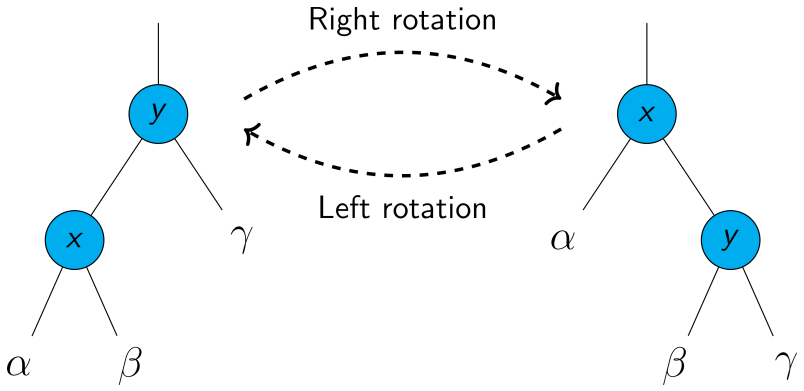


Figure 6.1: Rotate example

The cost of the rotation operation is always the same and take constant time $\Theta(1)$.

Algorithm 50: RBT: ROTATE

```
def ROTATE(T, x, side)
    r_side←REVERSE_SIDE(side)
    y←GET_CHILD(x, r_side)
    TRANSPLANT(T, x, y)
    beta←GET_CHILD(y, side)
    TRANSPLANT(T, beta, x)
    SET_CHILD(x, r_side, beta)
enddef
```

Insertion requires:

- Inserting as in BST (cost depends on the height of the tree: logaritmic for a RBT).
- RED-coloring the node: we cannot insert directly black nodes because it would change the BH. But that could put a RED node as a child of another RED node, so we need to..
- Fixing up RB-Tree properties:
  - Case 1: x's uncle (y) is RED, so we RED-color x's granpa and BLACK-color x's parent and y.
  - Case 2: x's uncle (y) is BLACK and y and x are on the same side. In this case we rotate x's parent on the opposite side. The new x is the former x's parent.
  - Case 3: x's uncle (y) is BLACK and y and x are on different side. In this case we invert x's parent and grandparent colors and rotate on x's granpa on y's side.

All this case takes constant time. For the total complexity, let's consider all the possible situations:

- If x parent is BLACK: done.
- If x has no uncle, either x or $x.parent$ are the root: BLACK-color it.
- If x has an uncle, we can always choose between Cases 1,2, or 3.
  - Case 1 either removes the problem or pushes it torwards the root (so we are upper bounding the cases in which we can apply Case 1).
  - Case 2 brings to Case 3.
  - Case 3 solves the problem

  So in the worst case scenario the algorithm keeps repeating Case 1 steps along the insertion branch and the complexity is $O(\log n)$.

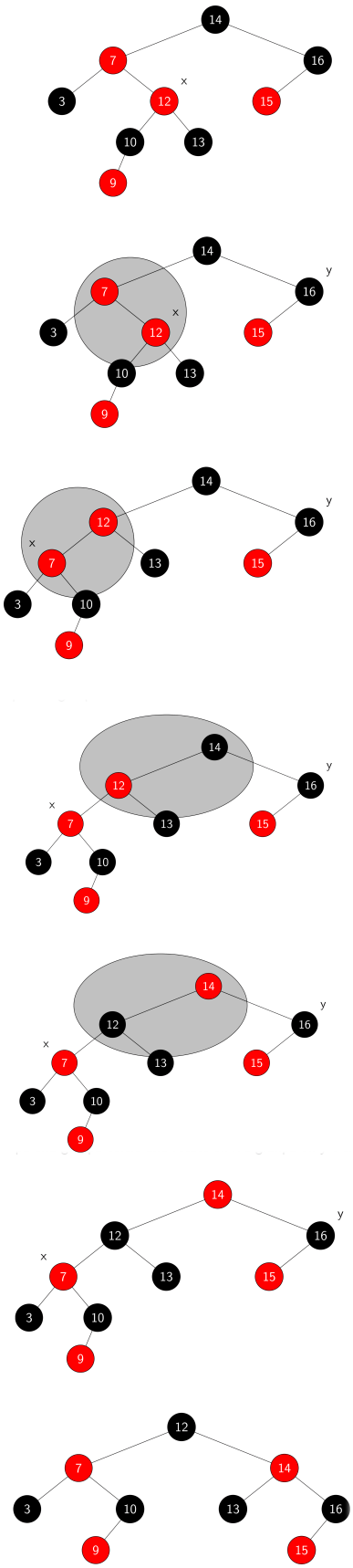


Figure 6.2: Case 1, 2 and 3 example

Algorithm 51: RBT: INSERT_RBTREE

```
def INSERT_RBTREE(T, v)
    x←INSERT_BST(T, v)
    x.color←RED
    FIX_INSERT_RBTREE(T, x)
enddef
```

Algorithm 52: RBT: FIX_INSERT_RBT_CASE1

```
def FIX_INSERT_RBT_CASE1(T, x)
    UNCLE(x).color←BLACK
    x.parent.color←BLACK
    GRANDPARENT(x).color←RED
    return GRANDPARENT(x)
enddef
```

Algorithm 53: RBT: FIX_INSERT_RBT_CASE2

```
def FIX_INSERT_RBT_CASE2(T, x)
    x_side←BCHILDHOOD_SIDE(x)
    p←x.parent
    ROTATE(T, p, REVERSE_SIDE(x_side))
    return p
enddef
```

Algorithm 54: RBT: FIX_INSERT_RBT_CASE3

```
def FIX_INSERT_RBT_CASE3(T, x)
    g←GRANDPARENT(z)
    x.parent.color←COLOR
    g.color←RED
    x_side←CHILDHOOD_SIDE(x)
    ROTATE(T, g, REVERSE_SICE(x_side))
enddef
```

Algorithm 55: RBT: FIX_INSERT_RBTREE

```
def FIX_INSERT_RBTREE(T, x)
    while ¬IS_ROOT(x) and (¬IS_ROOT(x.parent) or x.parent.color=RED) do
        if COLOR(UNCLE(x))=RED then
            z←FIX_INSERT_RBT_CASE1(T,x)
        end

        else
            if CHILDHOOD_SIDE(x)≠CHILDHOOD_SIDE(x.parent) then
                z← FIX_INSERT_RBT_CASE2(T,x)
            end
            FIX_INSERT_RBT_CASE3(T,x)
        end
    end
    T.root.color←BLACK
enddef
```

Removing a key

Removing a key as in BST remove also a node y which is replaced by its former child x

- If y was RED, the RBT properties are preserved: not breaking any rules concerning both red and black key.
- If y was BLACK, the branches through x lost 1 BLACK node: $BH(x) = BH(w) - 1$ where w is the sibling of y before the removal. That is a problem, because we now have two siblings with different heights.

There are different cases:

- Case 0: x is RED. We just need to color x BLACK. $BH(x)$ is increased by 1 and the tree has been fixed.
- Case 1: x's sibling is RED.
  - Invert colors in x's parent and sibling.
  - Rotate x's parent on x's side.

  $BH(x)$ does not change. By applying this transformation we moved the problem towards the leafs. We increased the distance from the root down to the root we are considering (the opposite of the insertion).
- Case 2: x's sibling and nephews are BLACK. In this case we RED-color x's sibling. $BH(x)$ does not change, while the BLACK height of both x's parent and sibling are decreased by 1.
- Case 3: among x's sibling and nephews, only the nephew on x's side is RED. In this case:
  - Rotate x's sibling on the opposite side with relation to x.

  - Invert colors in both old and new siblings of x.

  The BLACK height of both x and x's parent does not change.

- Case 4: The x's nephew on the opposite side with relation to x is RED. In this case:
  - Switch colors of x's parent and sibling.
  - BLACK-color the x's nephew on the opposite side with relation to x.
  - Rotate x's parent on x's side.

  The BLACK height of both x and x's parent does not change. In this case, we solved the problem.

All the case transformation procedure takes time $\Theta(1)$. All the transformation based on switching colors, applying a rotation, which takes constant times, are obviously constant. Times for different cases are:

- Both Case 0 and Case 4 transformation procedures fix the RBT.
- Case 1 cannot occur twice in-row: once we apply case 1, we are guaranteed that the sibling of the node we are interested in is BLACK. After it, we could apply either case 2, case 3 or case 4.
- Case 2 we move toward the root. We may end up again in case 1, but for sure after it we end up in a situation in which we can apply case 3.
- Worst case scenario: we keep allying case 2 without solving the issue until we push the problem to the root ($O(\log n)$ times).

This data-structure may be used to represent set. If we want to represent set of integers, the best way to do it is to use a RBT (depending on the application), because it is efficient both form the computational and space memory point of view.

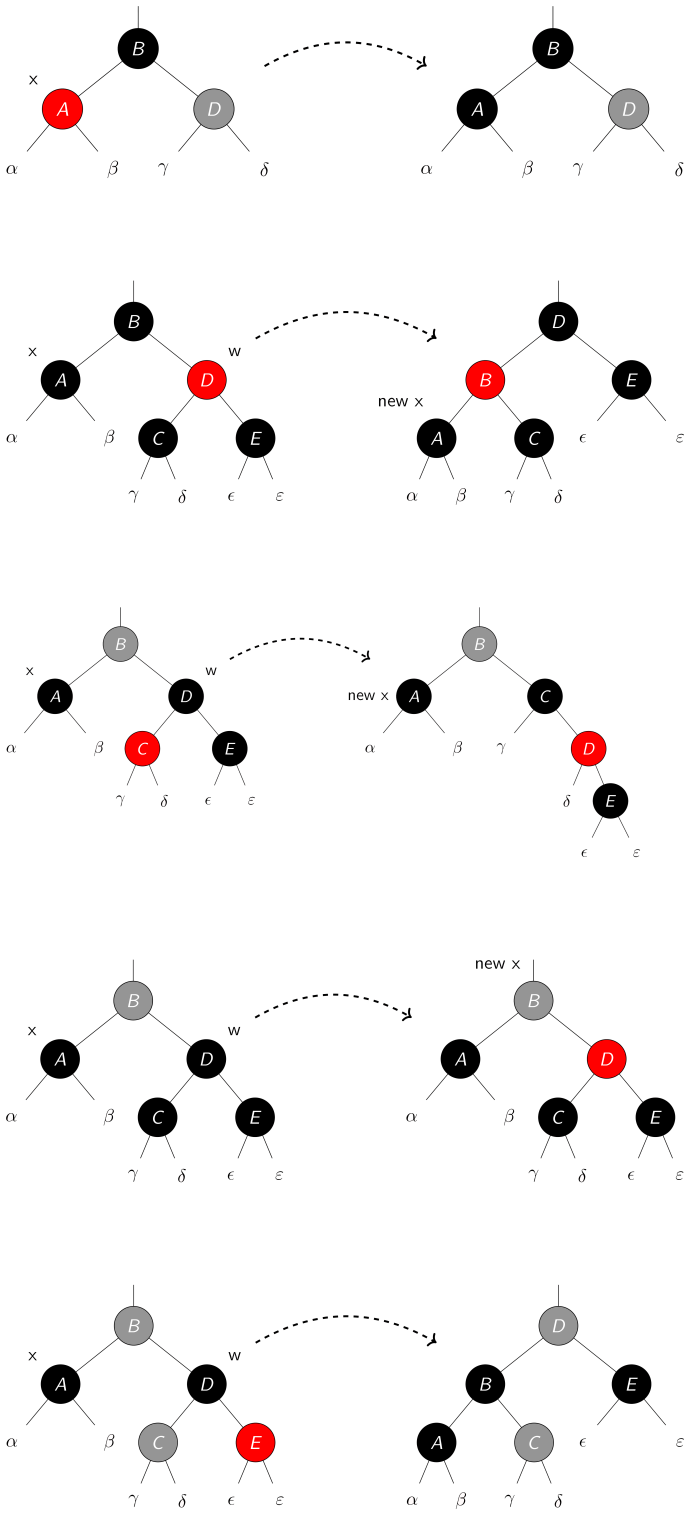


Figure 6.3: Example of all case of key removal

Chapter 7

Graphs

7.1 Basics

Graph is an unique way to represent a lot of different things. We already defined them in section 1.4.
There are two ways to represent a graph:

- Adjacency list (usually for sparse graph, i.e. the number of edges is at most linear with the number of nodes). For each of the node in my data structure we have an index in our array which store the list of its neighbors. What if we want to know if there is an edge between two nodes? We can go to see the list of nodes reachable using one single edge from the one we are interested in, and look for our ending point node.
  To detect if an edge is present in my graph, we are going to spend a time linear to the number of nodes in the graph (from the definition of sparse graph itself).
- Adjacency matrix (usually for dense graphs): a square matrix in which each row and each column represent the sources and the destination of edges. It is however less memory efficient.

The most trivial task to be performed on a graph is to visit it (i.e. search all the node on the graph).
There are two main ways to do it in both directed and undirected graphs:

- Breadth-First-Search (BFS): visit the graph as a "wave" from the source, and moving along all the nodes.
- Deapth-First-Search: search "deeper" in the graph whenever possible (i.e. follow a branch until you get to the wanted node).

7.2 Breadth-First Search

Visiting order is related to the distance from a source s: the lesser the distance of a node, the sooner it will be visited. For this reason BFS is used to compute source-node

distances. It also produces the breadth-first tree, i.e. the tree of shortest paths from s, and returns the shortest path from s to any reachable node. So BST is used to evaluate the distance of a node, and the shortest path.
Let's try to see how BFS works: at the beginning of computation the distance from any possible source to any other node to the graph is infinity, because we have not started the computation and moreover we do not know the starting point. So the initialization is meant to assigns infinity (the guessed minimal distance) from the starting point to any node in the graph. The initialization procedure also assign a color to any node in the graph: the color is meant to tell whether the node has already been reached during the computation, or whether it has been reached but we still lacks some of the computation it concerns (every possible evaluation).
Nodes are WHITE, GRAY or BLACK:

- WHITE nodes have not been discovered yet. So the initialization assigns all the distances to infinity, and all the nodes to white.
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered. As soon as the real computation start, we have to select a starting node. Moving from the starting node (point 0) to a new node following an edge, we increase by 1 the distance of the new node just visited. We do the same for all the nodes directly connected. Once we have colored all the neighbors, the original node has no more WHITE neighbors, and we can color it BLACK.
- BLACK nodes have been discovered and all their neighbors have been discovered too.

Doing this operation we are not only changing the color of the nodes, but also evaluating the predecessor on the shortest path (i.e. as a node turn GRAY, we know that the shortest path from the origin to it pass trough its already BLACK colored predecessor, and we know it because we discovered that new node passing by the aforementioned BLACK node).
We continue with the computation by doing the same passages to the first node discovered (preserving the discovery time). Eventually we BLACK color all the nodes reachable from the starting one. It may happen that some nodes are note reachable from the starting

graph. It remains signed with infinity distance.

Algorithm 56: BFS_SET

```
def BFS_SET(v, color, d, pred)
    v.color←color
    v.d←d
    v.pred←pred
enddef
```

Algorithm 57: BUILD_INIT

```
def BUILD_INIT(G, s)
    for v in G.V do
        BFS_SET(v, WHITE, ∞, NIL)
    end
    BFS_SET(s, GRAY, 0, s)
    return BUILD_QUEUE([s])
enddef
```

Algorithm 58: BFS

```
def BFS(G, s)
    Q←BFS_INIT(G,s)
    while Q≠ ∅ do
        u←DEQUEUE(Q)
        for v in G.Adj[u] do
            if v.color=WHITE then
                BFS_SET(v, GRAY, u.d+1,u)
                ENQUEUE(Q,v)
            end
        end
        u.color← BLACK
    end
    return G.pred, G.d
    )
enddef
```

Initialization takes time linear with the number of nodes in the graph ($\Theta(n)$, where n is the set of nodes, i.e. the cardinality of v). That is because we have to do a for loop across all the nodes in the graph.
Let's focus on the block of the while loop in BFS, how many time we are dequeueing one single node from Q(the queue)?
To insert a node in the queue, the node itself must be WHITE. Once inserted the node turns GRAY. We cannot insert twice the same node in the queue, so the time we are going through the while loop is at least proportional to the number of nodes in the graph, at most linear (proportional because it could happen that some nodes are not inserted in the queue).

Now let's look at the for loop. We go through it a number of time equivalent to the number of edges moving from u. In total the number of times we repeat the for loop among all the repetition of the while is equivalent to the number of edges in the graph (i.e. for each node u we are repeating the for loop a number of time equal to the edges from u, and we may end up doing it for all nodes)
In summary:
An iteration of while extract a u from Q and GRAY colors it. The for loop cost $\Theta(|Ad_j[u]|)$ per while iteration. Each iteration of the for enqueues $v \in Ad_j[u]$ only if it is WHITE. Every node can be inserted in Q at most once. Cumulatively, the while costs $O(|V|)$ and the for $O(|E|)$.
BFG has asymptoptic complexity $O(|V| + |E|)$

Lemma 3. *Let $Q = [v_1, ..., v_n]$ be the queue during BFS. Then $v_i.d \leq v_{i+1}.d$ for all $i \in [1, n-1]$ and $v_n.d \leq v_1.d + 1$*

So, $u_1.d + 1 \leq u_i.d + 1$ for all $i \in [2, n]$. If v is a successor of u_1, any other path reaching V through a node in Q is longer than $u_1.d + 1$

Theorem 4. *Let $\delta(s, v)$ be the distance from s to v. After BFS:*

- *$v.d \neq \infty$ iff v is reachable from s*
- *If $v.d \neq \infty$, then $v.d. = \delta(s, v)$*
- *The shortest path from s to v ends with $(v.pred, v)$*

7.3 Depth-First Search

All the nodes of the graph are visited:

- A non-visited node s is selected as source.
- A path is extended as much as it is possible by adding non-visited nodes.
- The process is repeated on all the branches left behind by previous step.
- If some nodes remain non-visited, a node among them is selected as new source.

At the end of the execution of a DFS, we have a depth-first forest of predecessor. A forest because the source of the DFS visit is made by many nodes, all the ones used to start the execution (the ones not visited before), each with its corresponding tree.
DFS labels all the nodes with discovery time (i.e. time at which we reach a new node) and finishing time (i.e. time at which a GRAY node turns BLACK, in the endpoint).
Nodes are WHITE, GRAY or BLACK colored:

- WHITE nodes have note been discovered yet. In the initialization all nodes are WHITE.
- GRAY nodes have been discovered, but not finished yet (i.e. some of their neighbors have not been discovered). We pick randomly a node in the graph, turn it GRAY and set its discover time at 1. We than move on to the following node picking an edge, and we turn the following node GRAY and discover time 2, and so on.

- BLACK nodes have been finished(i.e. nodes discovered and all their neighbors also discovered). When we have reached an endpoint, we go back to the previous node, and look for a different path.

The search exclusively evolves from GRAY nodes. Their processing order is handled by a FIFO queue.

Algorithm 59: DFS

```
def DFS(G)
    for v in G.V do
        v.color←WHITE
        v.pred←NIL
    end
    time←0
    for v in G.V do
        if v.color=WHITE then
            time←DFS_VISIT(G, v, time)
        end
    end
    return G.pred, G.d, G.f
enddef
```

Algorithm 60: DFS_VISIT

```
def DFS_VISIT(G, v, time)
    // discovery
    time←time + 1
    v.d←time
    v.color←GRAY
    // search for WHITE neighbors
     for u in G.Adj[v] do
        if u.color=WHITE then
            u.pred←v
            time←DFS_VISIT(G, u, time)
        end
    end
    // finalization
    time←time+1
    v.f←time
    v.color=BLACK
    return time
enddef
```

Each DFS_VISIT call GRAY color the node parameter. The for loop costs $\Theta(|Adj[u]|)$ per DFS_VISIT call. Each iteration of the for calls DFS_VISIT on $v \in Adj[u]$ if and only if it is WHITE. Every node is used as DFS_VISIT parameter exactly once. Cumulatively, the for loop in DFS is repeated $\Theta(|V|)$ time and produces $\Theta(|E|)$ iterations of the DFS_VISIT's

for loop.
DFS has asymptotic complexity $\Theta(|V| + |E|)$.
Let's now classify the edges of the graph according to the product of the DFS:

- Tree edges: belong to the depth first forest (i.e. discovery time of the destination node of an edge is greater than the source).
- Back edges: connect a node to an ancestor or self-loop (i.e. the discovery time of the node source of that edge is greater of the discovery time of the destination).
- Forward Edges: connect a node to a non-direct descendant.
- Cross Edges: the remaining edges (two nodes on different branches of the same tree).

Theorem 5. *Parenthesis Theorem:*
For any pair of nodes $v, u \in V$, either:

- $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ *and neither u is a descendant of v nor v of u.*
- $[u.d, u.f] \subsetneq [v.d, v.f]$ *and u is a descendant of v.*
- $[v.d, v.f] \subsetneq [u.d, u.f]$ *and v is a descendant of u.*

From that, we can deduce another theorem

Theorem 6. *White-Path Theorem:*
For any pair of nodes v, $u \in V$, u is a descendant of v iff at time $v.d - 1$ there exists a WHITE-only path from v to u.

7.4 Topological Sort

Example 20. *Trying to install a software can be a "sorting" problem. To install software A we have to solve the following dependency:*

- *Software A needs software B,C, D*
- *Software B depends on libraries E and F and software G*
- *...*

How to figure out what is needed and in which order?

Dependency relations can be modeled by using direct graphs. Nodes represent sub-tasks, edges the needs.
On DAG's, there exists a topological order $\preceq_T$ on nodes satisfying:

$$\text{If } (u, v) \in E, \text{ then } u \preceq_T v$$

How can we compute the total order? Using DFS

Algorithm 61: TOPOLOGICAL_SORT

```
def TOPOLOGICAL_SORT(G)
    call DFS(G)
    as a node is finalized, push it into a stack S
    return S
enddef
```

Its asymptotic complexity is $\Theta(|V| + |E|)$.

Correctness

Lemma 7. *Let $[v_1, ..., v_n]$ be the output of TOPOLOGICAL_SORT(G). Then $v_i.f > v_{i+1}$ for all $i \in [1, n-1]$.*

Since $[v_1, ..., v_n]$ is the output of topological sort, we know for sure that the finalization time of $v_i.f$ is greater than $v_{i+1}.f$ because a passage from one to the next is a step.

Lemma 8. *G contains cycles iff DFS(G) yields back edges.*

Back edges are those which allows us to move from a node to a node in the same branch which has been sort before. `DFS` is a good way to look if there are cycles in a graph (checking if there are back edges),

Theorem 9. *If G is a DAG, then TOPOLOGICAL_SORT(G) produces G's topological sort.*

7.5 Strongly Connected Components

If we have a cycles inside the graph we are considering, we cannot apply topological sort. That is because when a graph is not DAG, establishing a topological ordering is not possible (there is not a total order for the nodes in the graph).
From a reachability point of view, all the nodes in a loop behave in the same way (i.e. whenever from a node in a cycle we can reach another node, that node can be reached from any other node in the cycle). All the nodes in a cycle belong to the same kind of class, a partition of the overall nodes in the graph such that if you are able to reach a node from any node in the cycle, than all the nodes in the cycle are able to reach that node.

Definition 25. *Strongly Connected Components (SCCs): maximum sub-graphs (of a directed graph) so that for every pair of their nodes, there is a path from one to the other and vice versa.*

By maximal we mean that it contains the maximal number of nodes and edges in the original graph. Picking two nodes at will, they are mutually reachable.
If we build all the SCCs in the graph, we are basically partitioning all the nodes in the graph itself. So if a node belong to a SCC, it cannot belongs to another.
Let's say we can crunch all the nodes of a SCC into a single node (like assuming we are

not able to say which nodes in it must come first, but we know that at some moment we have to pass through all of them, all together). By doing this operation we end up with a new graph, which is for sure a DAG, because we removed all the cycles. At that point we are able to apply topological sort. How can we identify an SCC?
DFS is able to identify SCC loops, as saw in a previous lemma.
But how to mark that two nodes belong to the very same cycle?
Minimum Discovery Time from the Sub-Tree (lowlink): if it is smaller than the node discovery time, then a back edge must be reachable from it (i.e. it could be the case that by crossing a back edge we are able to reach a node whose discovery time is lower than the one in which we already were). Basically the lowlink is decreased with respect to the discovery time of each node by crossing back edges (whenever we reach a node we update its discovery time and its lowlink, because we know that the lowest among all the possible discovery times reachable from that point is for sure at least equivalent to its discovery time). As soon as we are able to reach from that node another node which time is lower that the one just visited, we know that we have a cycle.
Another crucial intuition is that by calling `DFS_VISIT` once, we end up for sure visiting all the nodes of the same SCC.
Also `DFS_VISIT` do not interleave nodes of two distinct SCC.
We than have a way to:

- Perform a `DFS_VISIT` call and update lowlinks when possible.
- Identify a SCC as soon as all its nodes have been visited. All the nodes in a cycle end up having the same lowlink.
- Label the SCC nodes as "not available for lowlink updates". If we discover a node which is able to reach another node, which for sure belong to a different SCC, we can in some sense highlight this situation by marking all the nodes in the graph belonging to the same SCC, and saying something like "this nodes are not available to update your lowlink".

How are the discovery time and the lowlink of the first node S_f in a SCC S to be visited, related?
Let's assume we have never seen any node in my SCC by using `DFS_VISIT`. When we visit the first node, what is the lowest possible values among all the possible discovery times which would be assigned to that node? Is it possible that its lowlink is lower than the discovery time of the node itself?
No, because if it was the case, we end up in a situation in which we are able to reach from that node, another node which has been already discovered. So if we know that the previous node was not the first node in the SCC, by contradiction this is not possible.
Is it possible that the lowlink for all the nodes reachable from that one is greater than its discovery time?
Since we are looking for the minimum, and a node can for sure reach itself, the lowlink has to be at least the discovery time of the node reached at that iteration. The the first node visited by `DFS_VISIT` has the same discovery time as the lowlink.

So for sure once S_f has been finished, all the nodes in S have too.

Algorithm 62: TARJAN_SCC

```
def TARJAN_SCC(G)
    for v in G.V do
        v.color←WHITE
    end
    time←0
    S←BUILD_STACK()
    for v in G.V do
        if v.color=WHITE then
            time←TRAJAN_SCC_VISIT(G, v, S, time)
        end
    end
enddef
```

Algorithm 63: TARJAN_SCC_VISIT

```
def TARJAN_SCC_VISIT(G, v, S, time)
    time ← time+1
    v.d ← time
    v.color=GRAY
    v.lowlink←time
    S.push(v)
    v.onStack←True
    for u in G.Adj[v] do
        if u.color=WHITE then
            time ← TRARJAN_SCC_VISIT(G, u, S, time)
            v.lowlink← min(v.lowlink, u.lowlink)
        end
        if u.onStack then
            v.lowlink←min(v.lowlink, u.d)
        end
    end
    if v.lowlink=v.d then
        yield EXTRACT_SCC_FROM_STACK(S, v)
    end
    return time
enddef
```

Example 21. *Nodes are labeled by "Discovery Time"/"Lowlink".*

The algorithm performs a DFS-like visit + stack handling + lowlink handling (which is assigning a value to the lowest between two values, so it take constant time).
How many times are we going to sort an algorithm inside the stack?
We know that the Tarjan Algorithm inserts a node in the stack as soon as it is GRAY

colored.
How many time are we going to extract a node from the stack?
At most the number of time we are inserting the nodes in the stack.
One single node is pushed in S during each `TRAJAN_SCC_VISIT` call.
`TRAJAN_SCC_VISIT` is called on WHITE nodes and sets them to GRAY.
So the number of node inserted in S at some point is $|V|$.
All `EXTRACT_SCC_FROM_STACK` calls cumulatively cost $\Theta(|V|)$.
So the Trajan's algorithm costs $\Theta(|V| + |E|)$.

7.6 Transitive Closure

For each pair of nodes v and w, we would like to know whether there is a path from v to w.
A naive solution: evaluate BFS from all the graph nodes (i.e. using each of the node of the graph as source). That is because if we already visited a node, an edge connecting one still not visited to it could be not considered by the BFS, even if it actually exist. The complexity of applying this strategy is the number of nodes multiplied by the time require to apply the BFS to each of the nodes, so $|V|* = (|V| + |E|) = O(|V|^2 + |V| * |E|)$.
Let's now look at the graph the the matrix representation point of view.

- w has distance 1 from v if and only if $A[v][w] \neq 0$. To have this property, we need to have the diagonal (we may have a node which does not have a self loop). How can i decide if there is path from a node to another?
- w has distance 2 from v if and only if there exists z so that $A[v][z] \neq 0$ and $A[z][w] \neq 0 \Leftrightarrow (A \times A)[v][w] > 0$.
- w has distance $\leq n$ from v if and only if $A^n[v][w] > 0$.

Every acyclic path has at most length $|V|$ (we are only focusing on simple paths, as we are interested in reachability). We can solve the problem using Strassen's algorithm: $(|V| - 1) * \Theta(|V|^{\log_2 7})$. Which is even worst than the one of the naive solution.
Let's try to look for something better. We start from noticing that for concerns reachability, all the nodes in a SCC are the same. So we can collapses the nodes in SCCs in a single node and build the SCCs graph $\overline{G}$. We may have reduced the number of nodes in the original graph. We it could be the case in which in our graph there are no SCC. However, it is important because after this procedure we are dealing with a graph which is a DAG (all the cycle has been removed and replaced by a single node).
On a DAG we can apply Topological sort (already done by the Tarjan's algorithm), but why?
Let's have a look on the adjacency matrix of $\overline{G}$. The effects of re-sorting the adjacency matrix by the order provided by the topological sort: the matrix becomes upper-triangular. But why triangular matrixes are good for our problem?
We can split the matrix into quadrants: upper-left quadrant A, upper-right quadrants C, lower-right quadrant B (and the remaining one, made of zeroes). In this operation we are partitioning the edges in the graph: there are three kind of edges, the ones which moves from the first half of the nodes, to the first half of the nodes (A); the ones which moves from the second half of the nodes, to the second half of the nodes (B); then quadrant C

which tells us when we can move from the first half of the nodes to the second half of the nodes.

$$\overline{G} = \left(\begin{array}{c|c} A & C \\ \hline 0 & B \end{array} \right)$$

For this situation, any path in the graph can have at most one single edge in C, because whenever we are crossing an edge in C, we are moving from the first half to the second half of the nodes, and since all the edges in C are guaranteed to have a source in the first half of the nodes, we cannot end up in A anymore. So any path in $\overline{G}$ can contain at most one of the edges in C.

$$\overline{G}^* = \left(\begin{array}{c|c} A^* & A^* \times C \times B^* \\ \hline 0 & B^* \end{array} \right)$$

We can compute the matrix multiplication of $\overline{G}$ times itself, as many time as we want, simply by reducing the problem to a set of three sub-problems:

- The repetition of taking the edges from submatrix A in itself.
- The repetition of taking the edges from B in itself.
- If we want to move from the first half to the second half, we can keep repeating edges in the first half, cross an edge in C, then keep repeating in the second half.

Let's see the complete algorithm:

- Trajan's SCCs algorithm on the input $G : \Theta(|V| + |E|)$
- Build the SCCs graph $\overline{G}$ of $G : \Theta(|E|)$
- Topological sort of $\overline{G} : \Theta(|V| + |E|)$
- Compute
$$\overline{G}^* = 2 * T(n/2) + 2 * \Theta(n^{\log_2 7}) \tag{7.1}$$
It can be proved that $T(|V|) \in \Theta(|V|^{\log_2 7})$
- Extend the transitive closure of $\overline{G}$ to $G : O(|V|^2)$

So the overall asymtotic complexity is $\Theta(|E| + |V|^{\log_2 7})$.

Chapter 8

Weighted Graphs

Definition 26. *Weighted Graph: a graph with weighted edges. More formally,* (V, E, W) *are structures where:*

- (V, E) *is an (un)directed graph.*
- W *is a function mapping edges into weights.*

The length of a path is the sum of all its edge labels.

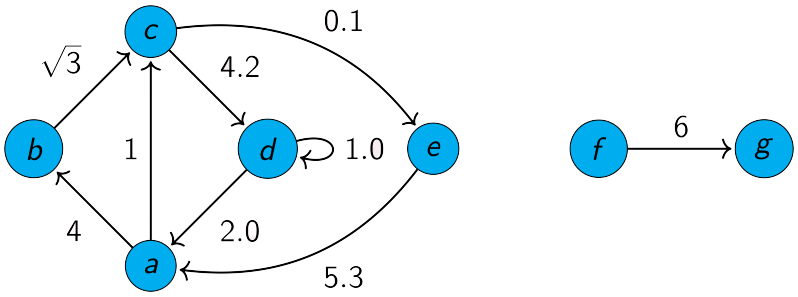


Figure 8.1: Weighted graph example

Weighted graph can be represented in two main ways:

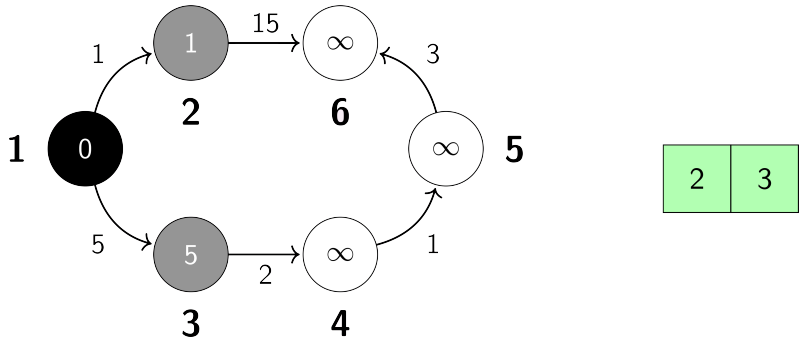
- Adjacency lists (usually, for sparse graph). In this case, also the weight of the edges is appended after the arriving node.
- Adjacency matrix(usually, for dense graphs). Instead of 0s and 1s, the matrix store the weight of the edges + a label for when the edge does not exist.

8.1 Single Source Shortest Paths

As before, we want to compute all the shortest paths from a single node s. Computing the shortest paths from s in a non-weighted graph was solved using BFS in time $O(|V|+|E|)$. So we can try to adapt BFS to SSSP. Their processing order is handled by a FIFO queue.

The main requirements for BFS was explained in 7.2.
BFS set v's distance to $u.d + 1$ where u is queue head. We could, instead of updating the distance by 1, set v's distance to $u.d + W[(u, v)]$. But in the lemma what guarantees the working of BFS(Lemma), the distance is set to 1. From it we remember that v is a successor of u_1, any other path reaching v through a node in Q is longer than $u_1.d + 1$. This does not really apply to BFS. Even if $u_{i-1}.d \leq u_i.d$ for all $i \in [2, n]$, there may be $(u_k, \overline{v})$ so that

$$u_1.d + W[(u_1, v)] > u_k.d + W[(u_k, \overline{v})] \tag{8.1}$$



$$2.d + W[(2, 6)] = 1 + 15 > 5 + 2 = 3.d + W[(3, 4)]$$

Figure 8.2: BFS inapplicability example

Dijkstra's algorithm

We cannot trust anymore the fact the queue is already sorted. A possible idea to overcome this problem is enqueuing not-discovered nodes in place of the just discovered. All this nodes are "pre-labeled" with a candidate distance (which at the beginning is ∞). At each step a node having minimal candidate distance is extracted and "finalized". Its outgoing neighbors are updated (since all the other distances are at least equivalent to the smallest distance of the node from the source, it is then not possible that we are able to reach the node we are aiming to passing through those nodes, and using a shorter path).
BFS queues has became a priority queue with reference to candidate distance (i.e. we do not want to extract the first element of the queue, but counting the minimum distance among all the nodes).
It does not need coloring:

- BFS WHITE nodes correspond to nodes in Q.
- Nodes are finalized as soon as extracted from Q.

Paths are treated as distances: the predecessor of a node is updated every time a new possible minimum distance arises.
It is a kind of meta-algorithm: it does not specify how to handle the queue (just that we need to find the minimum).
Also, the algorithm only work if we are dealing with positive weights. If they were allowed,

the minimal path to the node extracted from Q could be not discovered (if we are going through a negative cycle, we could end up with a minimal distance not defined, because we can keep going through the cycle continuously decreasing the distance).

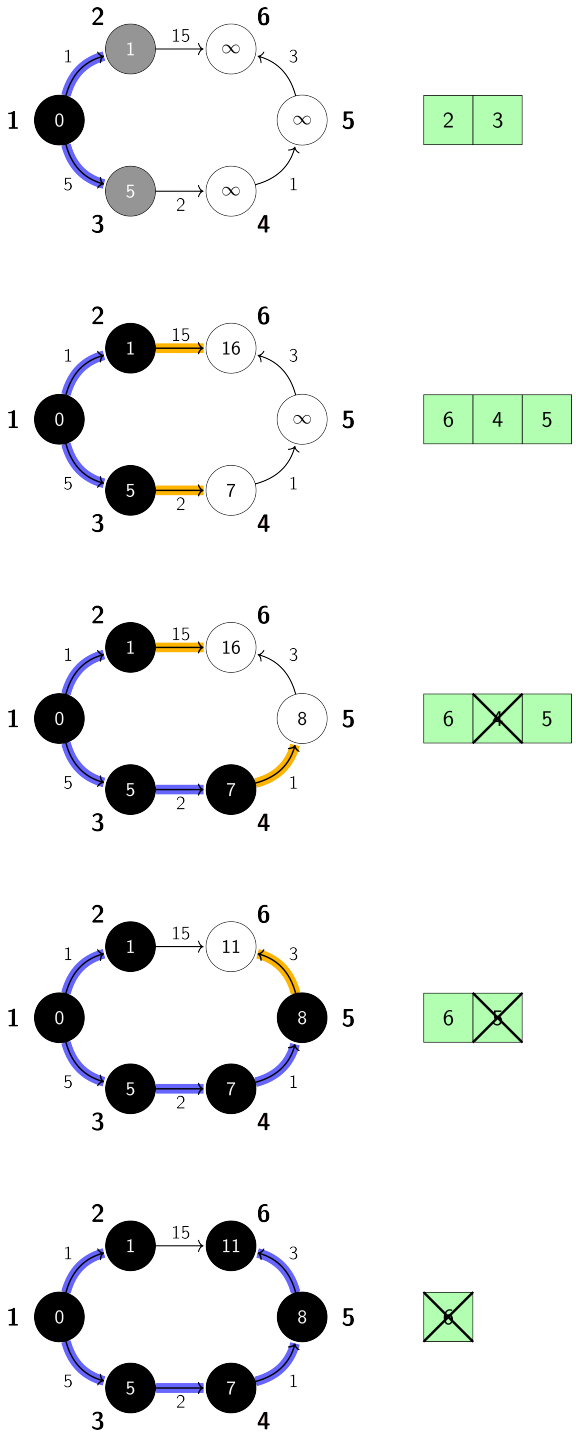


Figure 8.3: Dijkstra's algorithm example

Algorithm 64: INIT_SSSP

```
def INIT_SSSP(G)
    for v in G.V do
        v.d ←∞
        v.pred← NIL
    end
enddef
```

Algorithm 65: RELAX

```
def RELAX(Q, u, v, w)
    if u.d+w<v.d then
        UPDATE_DISTANCE(Q, v, u.d+w)
        v.pred←u
    end
enddef
```

Algorithm 66: DIJKSTRA

```
def DIJKSTRA(G, s)
    INIT_SSSP(G, s)
    s.d←0
    Q←BUILD_QUEUE(G.V)
    while not IS_EMPTY(Q) do
        u←EXTRACT_MIN(Q)
        for (v, w) in G.Adj[u] do
            RELAX(Q, u, v, w)
        end
    end
    return G.d, G.pred
enddef
```

Complexity:

- All the nodes are in Q at the beginning of the computation.
- One node u is extracted at each while-loop iteration.
- The for-loop iterates on the adjacency list of u.
- Globally, the for-loop performs $|E|$ iterations.

So the overall complexity of the Dijkstra's algorithm is:

$$T_D(G) = \Theta(|V|) + T_B(|V|) + |V| * T_E(|V|) + |E| * T_U(|V|) \tag{8.2}$$

where T_B, T_E and T_U are the complexities of `BUILD_QUEUE`, `EXTRACT_MIN` and `UPDATE_DISTANCE`. The final complexity of the algorithm then depends on the data structure we use to implement it. While the complexity for binary heap implementation is better for non dense-graph, the naive array approach is better for dense graph.

Queue Data Structure	$T_B(n)$	$T_E(n)$	$T_U(n)$	$T_D(G)$
Arrays	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\|V\|^2 + \|E\|)$
Binary Heaps	$\Theta(n)$	$O(\log n)$	$O(\log n)$	$O((\|V\| + \|E\|) * \log \|V\|)$
Fibonacci Heaps[1]	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	$O(\|E\| + \|V\| * \log \|V\|)$

Figure 8.4: Complexity tables for different data structure

8.2 All Pairs Shortest Paths

We want to compute the shortest paths between all the pairs of nodes. We can use the Dijkstra's algorithm from each node in the graph. The complexity is then $O(|V| * (|V| + |E|) * \log |V|)$ (for a binary heap). The same conditions for Dijkstra's algorithm have to held, so no negative edges.
Let us consider graphs whose nodes are natural numbers. Let $p = e_1, ..., e_h$ be the shortest path from i to j (source and destination). Let k be the "greatest" internal node in the path. Since for sure we have the node k which is the greatest (greatest node touched by the path p and different from i and j). We can split the path into two subpath: there exists $\overline{h}$ so that $e_{\overline{h}-1}$ and $e_{\overline{h}}$ that have k as destination and source.
So, $e_1, ..., e_{\overline{h}-1}$ and $e_{\overline{h}}, ..., e_h$ are shortest paths between i and k and between k and j. If it is not the case, we could consider two shorter paths, join them, and get a path smaller the the original one, which is a contradiction. All the nodes in the two paths are surely smaller than k, for a similar reasoning.
We can than incrementally build up the shortest paths by admitting new untouched internal nodes at each step (at each step we ask our self whether the smallest path from i to j has been already discovered, or is the union of the path from i to k and k to j).
If $D^{(k-1)}$ (a matrix) contains the lengths of the shortest paths whose internal nodes are smaller than k, we can compute $D^{(k)}$

$$D^{(k)}[i,j] = \min(D^{(k-1)}[i,k] + D^{(k-1)}[k,j], D^{(k-1)}[i,j]) \tag{8.3}$$

i.e. let's consider a square matrix, for each cell in the matrix we have the shortest path from i to j which only touches nodes smaller than k. We can compute the smaller path from i to j, admitting also nodes equal to k, by asking our self if the new node is relevant (comparing with the path without the new node).
The same criterium applies to $\prod^{(k)}$ where $\prod^{(k)}[i,j]$ is the predecessor of j in the smallest path between i and j.

Floyd-Warshall's Algorithm

An operative example:

Algorithm 67: FLOYD_WARSHALL_STEP

```
def FLOYD_WARSHALL_STEP(old_D, old_P, k)
    D←COPY_MATRIX(old_D)
    P←COPY_MATRIX(old_P)
    for i ← 1 upto |G.V| do
        for j ← 1 upto |G.V| do
            if old_D[i][j]>old_D[i][k]+old_D[k][j] then
                D[i][j]←old_D[i][k]+old_D[k][j]
                P[i][j]←old_P[k][j]
            end
        end
    end
    return (D, P)
enddef
```

Algorithm 68: FLOYD_WARSHALL

```
def FLOYD_WARSHALL(G)
    D[0]←INIT_MATRIX_D0(G.W)
    P[0]←INIT_MATRIX_P0(G.W)
    for k←1 upto |G.V| do
        D[k], P[k] ← FLOYD_WARSHALL_STEP(D[k-1], P[k-1])
    end
    return (D[|G.V|], P[|G.V|])
enddef
```

The number of steps we have to perform at most are exactly the number of nodes in the graph. So the number of step is equal to the cardinality of edges in the graph itself, so the total complexity of the algorithm is $\Theta(|Q|)$.

8.3 Routing

Identify the shortest path from a source, up to a destination.
Given a weighted graph G, a source s, and a destination d, we aim for the shortest path in G from s to d. A possible solution is to apply the Daijstra's algorithm and discharge any solution which does not involve the destination (but its an extreme overkill, even stopping the algorithm as soon as d has been finalized).
The asymptotic complexity of this algorithm is the one of the Daijstra's algorithm general case, has it may be the case that d is the last node reached by the algorithm.
The idea to get a better algorithm is to use an heuristic distance h, not embedded in the graph.

Definition 27. *Heuristic distance: a distance which estimates the shortest distance from*

a source to the destination.

e.g. in a map an heuristic distance could be the euclidean distance.
If the shortest path length from s to u is $u.d$, then

$$u.d + W(u, v) + h(v, d) \tag{8.4}$$

estimates the length of the path between s and d. Estimation accuracy depends than both on h and on G topology.

A^*

The A^* algorithm has the same structure of the Dijkstra's algorithm, but with the addiction of the heuristic distance: Q is sorted according to

$$u.d + W(u, v) + h(v, d) \tag{8.5}$$

where $u.d + W(u, v)$ is the guessed shortest path length to v

Algorithm 69: RELAX_ASTAR

```
def RELAX_ASTAR(Q, u, v, w, d, h)
    if u.d+w<v.d then
        UPDATE_DISTANCE(Q, v, u.d+w+h(v, d))
        v.pred←u
    end
enddef
```

Algorithm 70: ASTAR

```
def ASTAR(G, s, d, h)
    INIT_SSSP(G, s)
    s.d←h(s, d)
    Q←BUILD_QUEUE(G.V)
    while not IS_EMPTY(Q) do
        u←EXTRACT_MIN(Q)
        for (v, w) in G.Adj[u] do
            RELAX(Q, u, v, w, d, h)
        end
    end
    return G.d, G.pred
enddef
```

Routing on World Scale Graphs

Graphs are huge and cannot be completely stored in memory. Neither Dijkstra nor A^* can be applied. To solve this problem, we could try to reduce the complexity of the graph, my eliminating all the nodes which are not important.
Let $V = \{1, ..., n\}$ be sorted in ascending "importance", from our problem point of view.

E.g. we can remove a node by trying to preserve the shortest path.
The shortest paths $(i, 1), (1, j)$ are replaced by shortcuts(i,j) so that

$$W(i, j) = W(i, 1) + W(1, j) \tag{8.6}$$

The new graph resulting from the substitution of a node with a shortcut is called contraction.

Definition 28. *The contraction of node k consists in:*

- *Adding the needed shortcuts.*
- *Removing k.*

The resulting graph is called "overlay graph".

The sequence of the overlay graphs is a contraction hierarchy (CH) (i.e. we have a hierarchy of graphs representing the original graph).
Now, let $v_1, ..., v_d$ be the shortest path on the CH with $w_i = W[v_1, v_d]$, and let's assume the importance of a node is represented by the y axes.
Every time we are removing a node from the graph which is less important, we add a shortcut (so in the image removing v_i or v_j just create a straight line). We can see than looking for the shortest path, for what importance concern, we have to firstly move upward, than downward. The shortest paths on CH have the form $v_1, ..., v_k, ..., v_d$ where:

- $v_{i-1} < v_i$ for all $i \leq k$
- $v_{i-1} > v_i$ for all $i > k$

We can work level by level, in this way we are able to focus only on parts of the original graph, instead of maintaining the space for the all of it.
We can build up two different graphs: the original graph, in which we take care exclusively of the edges moving from a node whose index is smaller than the next one, and a second graph in which we exclusively take care of those edges moving from a node whose index is greater than the one of the next one.
A.k.a. building $G\uparrow = (V, E\uparrow)$ and $G\downarrow = (V, E\downarrow)$ where:

- $E\uparrow = \{(v, w) \in E | v < w\}$
- $E\downarrow = \{(v, w) \in E | v > w\}$

and using a bidirectional version of Dijkstra on $G\uparrow$ and $G\downarrow$.
It search forward on $G\uparrow$ and backward on $G\downarrow$ until the two searches finalize the same node.
Many overlay graphs share a large set of edges, we can store only those that are about to disappear and the involved nodes. Looking at the overlay graph from the top, once again we get the original graph.
By merging subsequent layers, we endup with graphs which have high probability to the disconnected. Huge graphs can be parted into subgraphs at the lowest levels and connect them by using highest levels.

Chapter 9

Algorithms on Strings

9.1 Strings

An alphabet is a set of symbols (e.g. $\{0, 1\}$, or $\{a, ..., z\}$).
A string $S[1...|S|]$ is a finite sequence of symbols in an alphabet.
Σ^* is the set of all strings built on Σ.
ϵ is the empty string and belongs to Σ^*.
If $x \in \Sigma^*$ and $y \in \Sigma^*$, then $xy \in \Sigma^*$ is their concatenation.
If $y = xw$:

- x is a prefix of y and we write $x \sqsubset y$
- w is a suffix of y and we write $w \sqsupset y$

If $x \in \Sigma^*$ and $q \in \mathbb{N}$, x_q will be the x's prefix of length q.

Lemma 10. *Let x, y and w s.t. $x \sqsupset w$ and $y \sqsupset w$.*

- *If $|x| > |y|$, then $y \sqsupset x$*
- *If $|x| = |y|$, then $y = x$*

Given:

- A finite alphabet Σ
- A text $T[1, ...n]$
- A pattern $P[1...m]$ with $m \leq n$ (usually we mean that this is the string that we have to search)

P occurs with shifts s in T means $T[s+1...s+m] = P$ If P occurs with shift s in T, then s is a valid shift.

9.2 String-Matching

The problem: find all the valid shifts for P in T.
A naive solution is to try all the possible shifts for P in T.

Algorithm 71: NAIVE_STRING_MATCHING

```
def NAIVE_STRING_MATCHING(T, P)
    valid←[]
    for s← upto |T| − |P| + 1 do
        i←1
        while i≤ andT[i+s]=P[i] do
            i←i+1
        end
        if i > |P| then
            valid.append(s)
        end
    end
    return valid
enddef
```

A match is tested for all the possible $|T| - |P| + 1$ shifts.
Each match test costs $O(|P)|$.
Since $|P| \leq |T|$, the overall complexity is $O(|P| * |T|)$.

A Better Idea

Let's assume that at some point of the repetition we were able to match in the text a prefix of length q of the pattern.
Is there a way to shift the pattern, while preserving the matches found so far in the text?
We want to shift the most we can while still preserving the matches done so far. This can be done by searching the largest prefix in the patter which is also a suffix for the prefix.
Thus, $P_k \sqsupset P_q$ beacuse $P_q \sqsupset T[2..q+1]$ and $P_k \sqsupset T[2..q+1]$, i.e. the prefix of length k of the pattern is also a suffix of prefix q of the pattern.

Definition 29. *Prefix function for P:*

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\} \tag{9.1}$$

$\pi[q]$ is the longest prefix of P that is a proper suffix of P_q.

Let $\pi^*[q]$ be $\{\pi[q], \pi^2[q], ..., \pi^{(t)}[q]\}$

Lemma 11.

$$\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\} \tag{9.2}$$

Lemma 12. *If $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q-1]$*

Let E_q be $\{k \in \pi^*[q] : P[k+1] = P[q+1]\}$

Theorem 13.

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \\ 1 + \max\{k \in E_{q-1}\} & \text{otherwise} \end{cases} \tag{9.3}$$

Algorithm 72: COMPUTE_PREFIX_FUNCTION

```
def COMPUTE_PREFIX_FUNCTION( P)
    π ← INIT_ARRAY(|P|)
    π[1]←0
    k←0
    for q←2 upto |P| do
        while w>0 and P[k+1]≠P[q] do
            k=π[k]
        end
        if P[k+1]=P[q] then
            k=k+1
        end
        π[q]←k
    end
    return π
enddef
```

The while-loop condition holds only if $k > 0$.
However, each iteration of then while-loop decreases k. k is initialized to 0 and is increased in the for-loop. So, the while-loop can be repeated $|P| - 1$ time at most.
The overall asymptotic complexity is than $\Theta(|P|)$.
Once a mismatch has been identified after q matches, the algorithm uses the prefix func-

tion to avoid $\pi[q]$ useless character comparisons.

Algorithm 73: KMP

```
def KMP(T, P)
    valid=[]
    π ←COMPUTE_PREFIX_FUNCTION(P)
    q←0
    for i←1 upto |T| do
        while q>0 and P[q+1]≠T[i] do
            q=π[q]
        end
        if P[q+1]=P[i] then
            q=q+1
        end
        if q=|P| then
            valid.append(i-q+1)
            q=π[q]
        end
    end
    return valid
enddef
```

As for the prefix function computation, the while-loop condition holds only if $q > 0$. However, each iteration of the while-loop decreases q. q is initialized to 0, and is increased i the for-loop, so the while-loop can be repeated $|T|$ times at most.
The overall asymptotic complexity is $\Theta(|P| + |T)|$

9.3 Multiple Patterns String Matching

For our problem we have:

- A text T
- A large set of patterns $\mathcal{P} = \{P_1, ..., P_l\}$

We want to find a valid shift for each P_i. Using the `KMP` algorithm, the complexity is $O(|T| * I + \sum_{i=1}^{l} |P_i|)$.
A possible better solution is using a tree-base process. Once we have built the tree, performing the string matching takes time equivalent to the overall number of character in the pattern. The complexity is $\Theta(\sum_{i=1}^{l} |P_i|)$.

A Formal Definition

Let $\sigma(T)$ be the set of all the substrings of T.
$STrie(T)$ of T is a tuple $(Q \cup \{\perp\}, \overline{\epsilon}, L, g, f)$ where:

- $Q = \{\overline{x} | x \in \sigma(T)\}$. i.e. all the nodes in the tree represent a substring in the original text.

- $\perp \notin Q$. i.e. we need a further node, which does not represent any substring.
- $L : Q \to [1...|T|$ is the shift label. i.e. the label for the shift for that specific node.
- $g : (Q \cup \{\perp\}) \times \Sigma \to Q$ is the transition function (the edges in the tree):
  - $g(\overline{x}, a) = \overline{xa}$ for all $xa \in \sigma(T)$
  - $g(\perp, a) = \overline{\epsilon}$ for all $a \in \Sigma$
- $f : Q \to Q \cup \{\perp\}$ is the suffix function (the suffix of the string which was represented by the previous node):
  - $f(\overline{ax}) = \overline{x}$ for all $ax \in \sigma(T)$
  - $f(\overline{\epsilon}) = \perp$

Let T^i be $T[1...i]$.
The boundary path of $STrie(T^i)$ is the sequence (i.e. the path we build up by starting from the node which represent the overall string we are representing at the moment by using the suffix array and by crossing the suffix link until we end up with the bottom node)

$$\overline{T^i} = s_1, s_2, ..., s_{i+1} = \perp \tag{9.4}$$

where $s_k = f^k(\overline{T^i})$
i.e. it is the path which is on the boundary of the new character we are going to introduce in our text.
The active point is the first s_j that is not a leaf.
The end point is the first $s_{j'}$ having a $T[i+1]$-transition (i.e. node which already contains an outgoing transition labeled by the character we are adding now at the end of the transition).

Algorithm mechanics

- Adds a $T[i+1]$-transition from s_h for all $h \in [1, j'-1]$ (i.e. we are adding an update transition labeled by the character we are trying to add to our text from any node which belong to the said interval: from the first node of the boundary path up to the end point).
  - If $h \in [1, j-1]$ (dealing with a node in the interval 1... active point), then it extends a branch (not splitting the branch).
  - If $h \in [j, j'-1]$, then it creates a new branch.

Algorithm 74: UPDATE_STRIE

```
def UPDATE_STRIE(S, T, i, top) // top is the node corresponding to
 T[1...i-1]
    r←top
    old_s←None
    while S.g(r, T[i])=None do
        s←CREATE_NEW_NODE()
        S.add_node(s)
        S.g(r, T[i])←s
        if old_s≠None then
            S.f(old_s)←s
        end
        old_s ←s
        r← S.f(r)
    end
    f(old_s)←S.g(r, T[i])
    return S.g(top, T[i])
enddef
```

Complexity:

- Each node is visited at most twice.
- Constant steps per node.
- $|Q| = |\sigma(T)|$

So building a $STrie(T)$ costs $\Theta(\sigma(T))$.
Unfortunately the number of sub strings for a text is up to quadratic with respect to the size of the test itself.

Lemma 14.

$$|\sigma(T)| \in O(|T|^2) \tag{9.5}$$

Theorem 15. *Building a $STrie(T)$ costs $O(|T|^2)$*

So this is really a non-optimal solution.

Reduce complexity

We can try to reduce redundancy in the suffix tries. We can in some sense remove those nodes (remove non-branching nodes):

- Q' containing branching nodes Q_b and leaves Q_l
- $g' : ((Q_b \cup \{\perp\}) \times \Sigma^* \to Q')$
- $f' : Q_b \to Q_b$

Let's count nodes now:

- The leaves represent some of the suffixes of T and they are at most $|T|$.
- All the internal nodes are branching and they are at most $|T| - 1$

So this kind of trees has $\Theta(|T|)$ nodes. This tree also represent the same thing the previous suffix tries was representing. The space previously wasted in the suffix tries is now embedded in the transition itself, because we need a way to represent the label in the transition, that now is no more constant (so we may need space for each transition equal to the length of the label). The label itself can be represented by using two index in the text: the initial and final position of the label.
So, to save space g' labels are represented as T-index intervals.
If $\Sigma = \{a_1, ..., a_k\}$, then:

- The string a_i labeling the edge $(\perp, \overline{\epsilon})$ is encoded as $[-i, -i]$
- $g'(\perp, [-i, -i]) = \overline{\epsilon}$ for all $i \in [1, k]$

We can also avoid L: look at the last matching label to infer shifts.
Not all the node of the suffix trees are explicitly represented.
We can represent implicit nodes by reference pairs explicit node/substring. The implicit nodes can be represented as a pair, whose first element is an explicit ancestor of the node, and the second element is the string we have to read from that ancestor to reach the implicit node (e.g. if $T = cac$, then $\overline{ca}$ is encoded as $(\overline{\epsilon}, [1, 2])$).
Also explicit nodes can be represented by reference pairs as (x, ϵ). (x, ϵ) is encoded as $(x, [p + 1, p])$.
If x is the closed ancestor of (x, w), then (x, w) is canonical.
By using this representation we see that every time we extend the path (from a suffix tree to an updated suffix tree in which we increase the text we are interested in by one character), done by moving along the suffix link (starting from the node representing the overall string, end up in an active point, than move to the end point), we do not need to deal with the nodes moving from the node representing the all string to the active point. That is because all this nodes are the leaves, and in the new representation we say that all the leaves are reachable by reading a string which starts from the original character and correspond to the string that you can reach until you end up in the final part of the string.
I.e. branch extensions is avoided by labeling transition to leaf as $[h, \infty]$.
What is the active point can be represent exclusively as the reference pair?
Let's assume s_j has canonical reference pair representation $(s, [k, i])$. If it is implicit, it should become explicit (branching, splitting from the root up to the text of interest, and building up a new node).
If s_j is explicit, add a new branch labeled $[i - 1, \infty]$
If $(s, [k, i])$ is on boundary path, the next node is $(f'(s), [k, i])$. A nuance observation here is that the active point of one iteration of this process is the endpoint of the previous one. Every time we are adding a new character, we perform all the operation from the active point to the end point, and the active point of the next character is the ending point of the previous one.
$(s', [k, i])$, where $s' = f'(s)$, may be non-canonical. Before any other procedure, we must canonize it (everything works only in the canonical representation: we are not admitted

to have a reference pair in which the first node of the reference pair is not the closest ancestor to the implicit node we are representing).
Let $[k', i']$ be the label of the $T[k]$-transition from s' (we can try to match the second part of the reference pair as much as we can):

- If $[k, i]$ is shorter than $[k', i']$, it is canonical.
- Otherwise replace:
  - s' with $g'(s', [k', i'])$
  - $[k, i]$ with $[k + (i' - k') + 1, i]$

  and repeat.

$\overline{T[j...i]}$ is the active point of $STree(T^i)$ if and only if $T[j...i]$ is the longest suffix of T^i that occurs twice.
$\overline{T[j'...i]}$ is the end point of $STree(T^i)$ if and only if $T[j'...i]$ is the longest suffix of T^i so that $T[j'...i+1]$ is a substring of T^i.

Theorem 16. *If $(s, [k, i])$ is the end point of $STree(T^i)$, then $(s, [k, i+1])$ is the active point of $STree(T^{i+1})$*

Algorithm 75: UPDATE_STREE

```
def UPDATE_STREE( root, T, s, k, i) // (s,[k,i-i]) is the canonical
 reference to active point
    old_r ← root
    (end_point,r)←TEST_AND_SPLIT((s,[k,i-1]), T, i)
    while not end_point do
        s← CREATE_NEW_BRANCH(r,i)
        ADD_A_SUFFIX_TRANSITION(root, old_r, r)
        old_r←r
        (s,k)←CANONIZE((s.f,[k,i-1]))
        (end_point, r)←TEST_AND_SPLIT((s,[k,i-1]), T, i)
    end
    ADD_A_SUFFIX_TRANSITION(root, old_r, r)
    return (s,k)
enddef
```

Algorithm 76: CREATE_NEW_BRANCH

```
def CREATE_NEW_BRANCH(r,i)
    // add a T[i]-transition from r to s
    s.g[T[i]]←(r,[i,∞])
    return s
enddef
```

Algorithm 77: ADD_A_SUFFIX_TRANSITION

```
def ADD_A_SUFFIX_TRANSITION( root, s, r)
    // if s = root, then s, f =⊥
    if s≠root then
        s.f ←r
    end
enddef
```

Algorithm 78: TEST_AND_SPLIT

```
def TEST_AND_SPLIT( (s, [k,p]), T, i)
    if k>p then // if (s, [k, p]) is explicit
        return HANDLE_EXPLICIT_NODE(s, T, i)
    end
    // if (s, [k, p]) is implicit
    return HANDLE_IMPLICIT_NODE((s,[k,p]), T, i)
enddef
```

Algorithm 79: HANDLE_EXPLICIT_NODE

```
def HANDLE_EXPLICIT_NODE( s, T, i)
    if s.g[T[i]]≠ NIL then // s has a T[i]-transition
        return (False, s)
    end
    return (True, s)
enddef
```

Algorithm 80: HANDLE_IMPLICIT_NODE

```
def HANDLE_IMPLICIT_NODE( (s,[k,p]), T, i)
    (dst,[sk, sp])← s.g[T[k]] // get T[k]-transition's dst and label
    rk←sk+p-k+1
    if T[i]= T[rk] then
        return (True, s)
    end
    r←CREATE_NEW_NODE()
    // split the T[k]-transition in (s,r) and (r, dst)
    s.g[T[sk]]←(r, [sk,rk-1])
    r.g[T[rs]]←(dst, [rk,sp])
    return (False, r)
enddef
```

Algorithm 81: CANONIZE

```
def CANONIZE( (s,[k,p]))
    if k> p then // if (s, [k.p]) is explicit
        return (s, k)
    end

    (dst, [sk, sp])← s.g[T[k]] // get T[k]-transition's dst and label
    while sp-sk≤p-k do
        k← k+sp-sk+1
        s←dst
        if k≤p then
            (dst, [sk,sp])←s.g[T[k]]
        end
    end
    return (s,k)
enddef
```

Algorithm 82: BUILT_STREE

```
def BUILT_STREE(T, Sigma)
    root ← CREATE_NEW_NODE()
    ⊥←CREATE_NEW_NODE()
    root.f ←⊥
    for i←1 upto |Sigma| do
        ai ← Sigma[i]
        ⊥.g[ai]←(root, [-i,-i])
    end
    k←1
    s← root
    for i←1upto|T| do
        (s,k) ← UPDATE_STREE(root, T, s, k, i)
        (s, k) ← CANONIZE((s, [k,i]))
    end
    return root
enddef
```

The algorithm is split into two components:

1. The canonize calls: takes time $O(|T|)$ in total.
2. The remaining computation: again, it is $\Theta(|T|)$.

So, for the general complexity:

- Solving the MPSM problem takes time

$$\Theta(|T|) + \Theta(\sum_{i=1}^{l} |P_i|) \tag{9.6}$$

- Suffix trees take space $\Theta(|T|)$

However:

- It handles only one of the pattern valid shifts.
- It requires up $(2 + 1 + |\Sigma|) * |T|$ words of RAM (non-feasable for genome).