OMAR CHEHAIMI

NOTES ON DATA STRUCTURES AND ALGORITHMS IN PYTHON

July 13, 2021

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*.

Felix qui potuit rerum cognoscere causas.

– Publius Vergilius Maro

PREFACE

In these notes I collected all the material I studied during the last years on the topic of data structures and algorithms. The main sources are the Udacity online course Intro to Data Structures and Algorithms in Python, and the book of M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, *Data Structures and Algorithms in Python* (John Wiley & Sons, Incorporated, 2013). I also used some additional material from webpages and YouTube educational videos.

These notes are not intended to be a book, nor a formal introduction to data structures and algorithms, they are just my personal attempt to have concise notes of the huge world on this topic.

Omar Chehaimi

CONTACTS

☞ omar.chehaimi@outlook.it

CONTENTS

Preface iv

1 PRELIMINARIES CONCEPTS 1
- 1.1 Computational Complexity 1
- 1.2 Big O Notation 1
  - 1.2.1 Properties 2
- 1.3 Time complexity evaluation example 3
- 1.4 Recursion 3
  - 1.4.1 Python Implementation Examples 3

2 DATA STRUCTURES 5
- 2.1 Collection 5
- 2.2 List 5
- 2.3 Array 5
- 2.4 Linked List 6
  - 2.4.1 Linked List Implementation 7
- 2.5 Stack 8
  - 2.5.1 Stack Implementation 9
- 2.6 Queue 9
  - 2.6.1 Queue Implementation 11
- 2.7 Set 11
- 2.8 Map 11
- 2.9 Hash Table 12
  - 2.9.1 Hashing 12
- 2.10 String Keys 13
  - 2.10.1 String Keys Implementation 13

3 SEARCHING AND SORTING 15
- 3.1 Binary Search 15
  - 3.1.1 Complexity of the Binary Search Algorithm 16
  - 3.1.2 Binary Search Implementation 16
- 3.2 Bubble Sort 17
  - 3.2.1 Complexity of the Bubble Sort Algorithm 17
  - 3.2.2 Bubble Sort Implementation 18
- 3.3 Merge Sort 19
  - 3.3.1 Complexity of the Merge Sort Algorithm 20
  - 3.3.2 Merge Sort Implementation 20
- 3.4 Quicksort 22
  - 3.4.1 Complexity of the Quicksort Algorithm 23
  - 3.4.2 Quicksort Implementation 24

4 TREES 26
- 4.1 General Definitions 26
- 4.2 Tree Traversal 27
  - 4.2.1 Depth-first search 27
- 4.3 Binary Trees 29
  - 4.3.1 Search 29

4.3.2 Delete 30
4.3.3 Insert 30
4.3.4 Perfect binary tree 31
4.3.5 Binary Tree Implementation 31
4.4 Binary Search Trees (BST) 32
4.4.1 Binary Search Tree Implementation 33
4.5 Heaps 34
4.5.1 Heapify 34
4.5.2 Heap Implementation 35
4.6 Self-balancing Binary Search Trees 35
4.7 Red-Black Trees 36
5 GRAPHS 38
5.1 General Definitions 38
5.1.1 Connectivity 38
5.2 Graph Representations 39
5.2.1 Edge List 39
5.2.2 Adjacency List 40
5.2.3 Adjacency Matrix 40
5.3 Graph Traversal 40
5.3.1 Depth-first Search (DFS) 41
5.3.2 Breadth-first Search (BFS) 41
5.3.3 Eulerian Path and Circuit 43
5.3.4 Hamiltonian Path and Circuit 44
5.3.5 Shortest Path Problem 44
5.3.6 Dijkstra's Algorithm 44
6 DYNAMIC PROGRAMMING 48
6.1 General Definitions 48
6.2 Knapsack Problem 49
6.3 Travelling Salesman Problem 51

Appendix 52
A BINARY SEARCH COMPLETE IMPLEMENTATION 53
A.1 Pre-order Traversal 53
A.2 In-order Traversal 55
A.3 Post-order Traversal 57
A.4 Breadth-first search Implementation 60
B IMPLEMENTATION OF GRAPH REPRESENTATION 63
C IMPLEMENTATION OF GRAPH TRAVERSAL 65
C.1 Depth-first search 65
C.2 Breadth-first search 66
D DIJKSTRA'S ALGORITHM IMPLEMENTATION 68
E COMPLEXITY SUMMARY 71

BIBLIOGRAPHY 72

LIST OF FIGURES

Figure 1 Plot of the main functions and their evaluation for computational complexity. 2
Figure 2 Factorial recursive execution. 4
Figure 3 An example of an array with elements and indexes. 6
Figure 4 Addition 4a and deletion 4b of an element of an array, and the subsequent update of the indexes. 6
Figure 5 An example of a linked list with the data and the reference to the next one. 6
Figure 6 Add a new element to a linked list. 7
Figure 7 Remove an element of a linked list. 7
Figure 8 A doubly linked list. 7
Figure 9 In a stack only the element at the top can be modified. 9
Figure 10 Allowed operations on a queue. 10
Figure 11 Allowed operations on a deque. 10
Figure 12 A priority queue. 11
Figure 13 An example of a map. 11
Figure 14 Has map and collisions. 12
Figure 15 An array with numeric values ordered in ascending order. 15
Figure 16 Binary search algorithm. 16
Figure 17 Array splitting in the implementation of the binary search algorithm. 17
Figure 18 The swapping process is repeated until the array is completely ordered. 18
Figure 19 Merge sort algorithm. 19
Figure 20 Merge sort algorithm implementation. 22
Figure 21 Quicksort algorithm. 23
Figure 22 Quicksort algorithm worst case. 23
Figure 23 Quicksort algorithm best and average case. 24
Figure 24 Quicksort algorithm implementation. 25
Figure 25 A linked list and a tree. 26
Figure 26 Possible structures of a tree. 26
Figure 27 A tree and its fundamental elements. 27
Figure 28 The depth-first search (28a) and the breadth-first search (28b). 27
Figure 29 Pre-order search. 28
Figure 30 In-order search. 28
Figure 31 Post-order search. 29
Figure 32 Example of tree search and traversal. 29
Figure 33 Deletion of an element of a tree. 30
Figure 34 Addition of an element to a tree. 30
Figure 35 Perfect binary tree. 31

Figure 36 Search on an ordered binary tree. 32
Figure 37 Addition on an ordered binary tree. 32
Figure 38 An unbalanced binary tree. 33
Figure 39 Max heap (39a) and min heap (39b) example. 34
Figure 40 Addition of a new node to a heap. 35
Figure 41 Implementation example of a heap using an array. 35
Figure 42 Example of an unbalanced (42a), and a balanced (42b) tree. 36
Figure 43 A red-black tree. 36
Figure 44 Insertion of a new node and the following updating of the nodes' color. 37
Figure 45 An example of right rotation. 37
Figure 46 An undirected and a directed graph and its elements. 38
Figure 47 A weakly connected graph. 39
Figure 48 Edge list. 39
Figure 49 Adjacency list. 40
Figure 50 Adjacency matrix. 40
Figure 51 Depth-first search example. 41
Figure 52 Breadth-first search example. 42
Figure 53 Spanning tree. 42
Figure 54 An Eulerian path (54a) and a not Eulerian path (54b). 43
Figure 55 Eulerian circuit. 43
Figure 56 Hamiltonian path (56a) and Hamiltonian circuit (56b). 44
Figure 57 Shortest path. 44
Figure 58 Relaxation condition. 45
Figure 59 Dijkstra's algorithms example. 45
Figure 60 Calculation of the fifth number of the Fibonacci sequence. 48
Figure 61 Knapsack problem. 49
Figure 62 Knapsack problem example. 50
Figure 63 Selection of the optimal subset. 51
Figure 64 Pre-order iterative example. 55
Figure 65 In-order iterative example. 57
Figure 66 Post-order search. 60
Figure 67 Breadth-first search example. 62
Figure 68 Depth-first search example. 66
Figure 69 Breadth-first search example. 67
Figure 70 Dijkstra's algorithms example. 68
Figure 71 Complexities of the most important algorithms. 71

LIST OF TABLES

Table 1 Binary search complexity. 16
Table 2 Merge Sort Complexity. 20

Table 3 Traversal paths for the tree in Figure 32. 29
Table 4 Eulerian path and circuit existence rules. 43
Table 5 Number of computation for each value of the Fibonacci sequence. 49
Table 6 Dijkstra's algorithms example for the graph in Figure 70. The shortest path from a to b is also shown. 68

LISTINGS

Listing 1.1 Sum of all the elements of a matrix. 3
Listing 1.2 Pseudocode of a recursive function and its internal execution. 3
Listing 1.3 Implementation of the Fibonacci series with both iterative and recursive way. 3
Listing 1.4 Implementation to calculate the factorial of a number using the recursive way. 4
Listing 2.1 Linked List implementation. 7
Listing 2.2 Stack implementation. 9
Listing 2.3 Queue implementation. 11
Listing 2.4 String key implementation. 13
Listing 3.1 Binary search Python implementation. 16
Listing 3.2 Bubble Sort Python implementation. 18
Listing 3.3 Merge sort Python implementation (the recursive part is taken from Merge Sort, GeeksforGeeks). 20
Listing 3.4 Quicksort Python implementation. 24
Listing 4.1 Class definition for a node and a tree. 31
Listing 4.2 Recursive pre-order traversal and search implementation. 31
Listing 4.3 implementation of insert and search operation for a binary search tree. 33
Listing 5.1 Recursive implementation of a depth-first search. 41
Listing 5.2 Recursive implementation of a depth-first search. 42
Listing A.1 Class definition for a node and a tree. 53
Listing A.2 Recursive and iterative implementation of pre-order traversal. 54
Listing A.3 Recursive and iterative implementation of in-order traversal. 56
Listing A.4 Recursive and iterative implementation of post-order traversal. 58
Listing A.5 Breadth-first search implementation. 61
Listing B.1 Graph representation and fundamental operations. 63
Listing C.1 Recursive and iterative implementation of depth-first search on graphs. 65
Listing C.2 Breadth-first search implementation on graphs. 66

Listing D.1 Dijkstra's algorithm implementation. Credits: stackoverflow. 69

ACRONYMS

ADT Abstract data type
LIFO Last In, First Out
FIFO First In, First Out
DFS Depth-first search
BFS Breadth-first search
BST Binary Search Trees

1 PRELIMINARIES CONCEPTS

1.1 COMPUTATIONAL COMPLEXITY

The **computational complexity**, or **complexity**, of an algorithm is the amount of resources (time and memory) required to run it, and it is the minimum complexity of all possible implementations for solving that given algorithm [1].

The amount of needed resources for solving an algorithm varies with the size of the input n. The computational complexity in general is a function $n \to f(n)$, and it could represent the worst case complexity (the maximum amount of required resources) or the average complexity (the average amount of required resources) over all the inputs of size n.

When the type of the complexity is not explicitly indicated, typically is meant to be the **time complexity**, and it is generally expressed as the number of elementary operations required to run a given algorithm. It is assumed that these elementary operations take the same amount of time for being solved on the same computer. The computational complexity can be related also to the memory consumption. In this case we talk about **space complexity**, which is generally expressed as the amount of memory required by an algorithm over all the inputs of size n.

1.2 BIG O NOTATION

The **big O notation** [2] describes the behaviour of a function when its argument tents to infinity or to a particular value.

Definition 1.1
Let f be a real or complex value function and let g be a real function. Let both functions be defined on the same positive and real unbounded interval, and let $g(x)$ be strictly positive for all large enough x values: $g(x) > 0, \forall x$ large enough. Thus: $f(x) = O(g(x))$ as $x \to \infty$ if $\exists\ M \in \mathbb{R}$, $M > 0$ and $x_0 \in \mathbb{R}$ such that $|f(x)| \leqslant Mg(x)\ \forall x \geqslant x_0$.

Usually $f(x) = O(g(x))$ is used as $x \to \infty$, but it can be also defined for the case $x \to a$, where a is a real number.

O notation is asymptotic for big x, so the important terms are the ones which grow faster than the others, which become irrelevant.

Example 1.1
In $f(x) = 6x^4 - 2x^3 + 5$ as $x \to \infty$, $6x^4$ is the fastest growing term. 6 is a constant and can be omitted, thus we have $f(x) = O(x^4)$.

1.2.1 Properties

Here are listed some simple properties about big O notation.

Definition 1.2 Product-Sum-Multiplication by a constant
Product
$f_1 = O(g_1)$ *and* $f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
Sum
$f_1 = O(g_1)$ *and* $f_2 = O(g_2) \Rightarrow f_1 f_2 = O(max(f_1, f_2))$
Multiplication by a constant
Let k *be a nonzero constant, then:* $O(k|g) = O(g)$, $f = O(g) \Rightarrow kf = O(g)$

Definition 1.3 Logarithm and Exponential
Let c *be a nonzero constant, then:* $O(\log n^c) = O(\log n)$, *because:* $(\log n^c = c \log n)$.
$O(n^c)$ *and* $O(c^n)$ *are very different: if* $c > 1$ *the latter grows much faster.*
Let c *be a nonzero constant.* $(cn)^2 = c^2 n^2 = O(n^2)$, *but* 2^n *and* 3^n *are not of the same order. In general* $2^{cn} = (2^c)^n$ *is not of the same order of* 2^n.

Example 1.2
$f = 9 \log n + 5(\log n)^4 + 3n^2 + 2n^3 = O(n^3)$ *as* $n \to \infty$

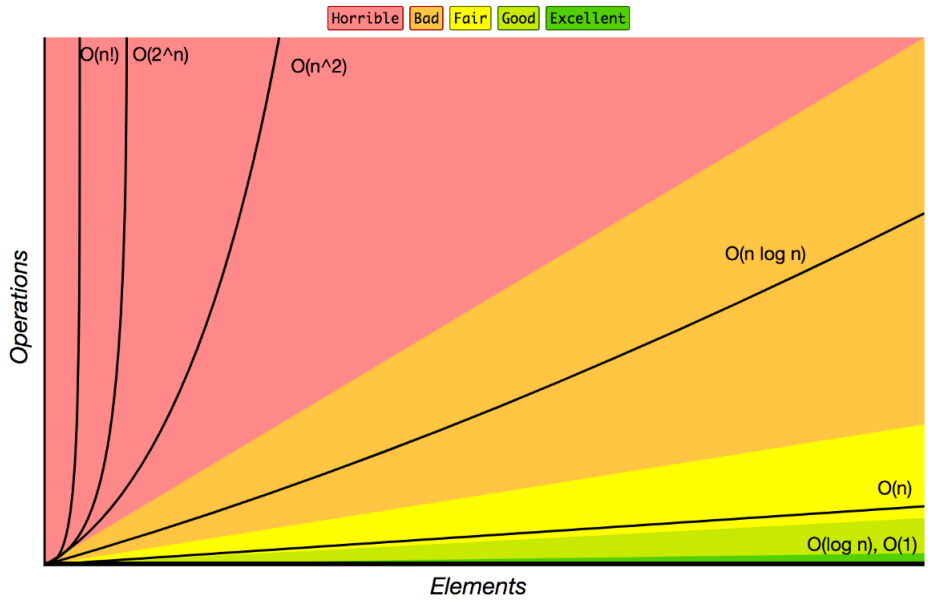


Figure 1: Plot of the main functions and their evaluation for computational complexity. Credits: bigocheatsheet.com.

1.3 TIME COMPLEXITY EVALUATION EXAMPLE

Let us suppose we want to calculate the sum of all the elements of a $n \times n$ matrix. For calculating the time complexity of this function we have to identify all the elementary operations and calculate their complexity based on how many times they are repeated. Here is the pseudocode for the function that calculates the sum of all the elements of a matrix.

Listing 1.1: Sum of all the elements of a matrix.

```
def find_sum_2d(array_2d):
      totoal = 0 # -> O(1)
      for each row in array_2d: # -> repeated n times
            for each column in array_2d: # -> repeated n times
                  total += array_2d[column][row] # -> O(1)
      return total # -> O(1)
```

The total time complexity is:

$$T = O(1) + n^2 O(1) + O(1) = O(n^2)$$

Where $O(1)$ is a constant value.

1.4 RECURSION

In recursion a function calls itself again on a smaller size input, until the exit condition stops this self calling (recursive calling) [3]. There are three fundamental elements in a recursive function:

1. A function that calls itself.
2. An exit condition. Without this condition a recursive function would call itself forever without an end.
3. An input alteration. When the function is called again the input is changed to a smaller dimension than the previous call.

Listing 1.2: Pseudocode of a recursive function and its internal execution.

```
function recursive(input):
      if exit condition: # Element 2
            return input
      else:
            output = recursive(input-1) # Element 1
            return output # Element 3
```

In recursion the exit condition is fundamental since if it contains some errors the recursion will never end, resulting in an infinite process.

1.4.1 Python Implementation Examples

Listing 1.3: Implementation of the Fibonacci series with both iterative and recursive way.

```
def fibonacci_iterative(position):
      if position == 0:
            return 0
      elif position == 1:
            return 1
      else:
            first = 0
            second = 1
            next_value = first + second
            for i in range(2, position):
                  first = second
                  second = next
                  next_value = first + second
            return next_value

def fibonacci_recursive(position):
      if position <= 1: # Exit condition
            return position
      return fibonacci(position - 1) + fibonacci(position - 2) # Calling
          the function on a smaller size input
```

Listing 1.4: Implementation to calculate the factorial of a number using the recursive way.

```
def factorial(n):
      if n <= 1: # Exit condition
            return n
      else:
            return n*factorial(n - 1) # Calling the function on a
                smaller input
```

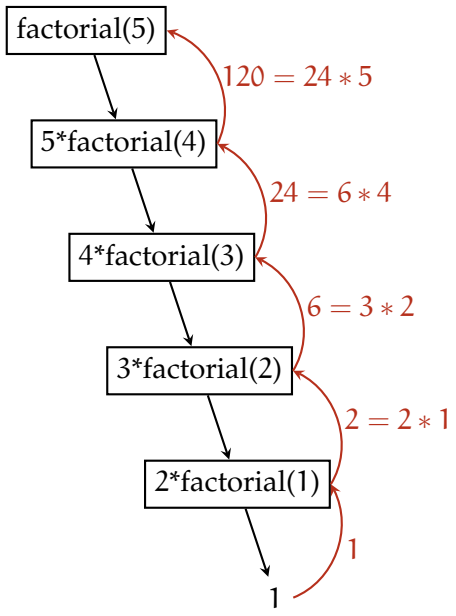


Figure 2: Factorial recursive execution.

2 DATA STRUCTURES

A **data structure** is a data organization, management, and storage format that enables efficient access and modification of the collected data. The relationship among the data, their properties, and the operations that can be done on them, are all properties of the data structure [4]. Data structures are the basis for the **abstract data types (ADT)**. An abstract data type is a mathematical model for **data types** [5], and it is defined by its behaviour, by the type of data it can have, by the possible operations that can be done on these data, and by the behaviour of these operations. This mathematical model contrasts with data structures, which indeed are concrete representations of data and are the point of view of an implementer, not of a user [6].

In this chapter the most important data structures like **collections**, **lists**, **arrays**, **linked lists**, **stacks**, and **queues** are introduced. For each data structure are shown all the most important properties, operations, and implementations.

2.1 COLLECTION

A **collection** is an object that groups several different elements in a single unit. Collections are used to save, to find, to manipulate, and to communicate grouped data [7]. Usually the elements belonging to a collection are of the same type, such as a poker hand (a collection of playing cards), a folder containing emails (a collection of emails), or a phone book (a map *name* $\rightarrow$ *phone number*).

2.2 LIST

A **list** is a collection which represents a set of **ordered** elements, which can be also of different type. Same value elements can be repeated several times. Lists do not have a fixed size, and it is possible to add, to remove, and to modify all the elements in the list. The complexity of adding or removing an element is constant ($O(1)$) [8].

2.3 ARRAY

An **array** is a collection of elements of the same or different type, in which each of them is identified with at least one **array index** or **key** [9]. In some programming languages arrays have a fixed size, while in others, instead, it can be variable.

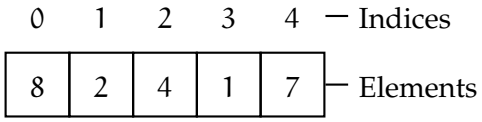
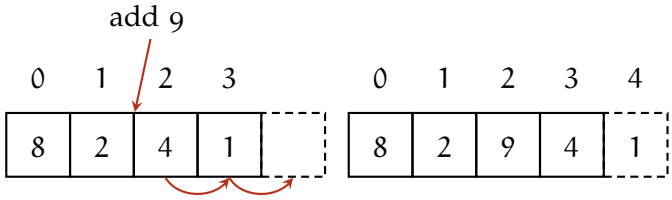


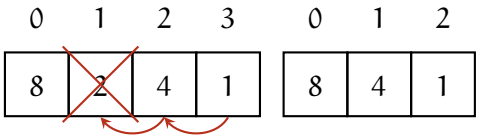
Figure 3: An example of an array with elements and indexes.

For doing any allowed operation to an element of an array it is enough to know its index.

Adding or removing an element of an array could be a very expensive operation. This is because when a change takes place to an element, all the following indexes must be updated (Figure 4). The worst case complexity for this operation is $O(n)$, where n is the size of the array.



(a) Add an element to an array.



(b) Remove an element of an array.

Figure 4: Addition 4a and deletion 4b of an element of an array and the subsequent update of the indexes.

2.4 LINKED LIST

A **linked list** is a linear collection in which each element has the data and the reference to the next one (a link) [10].

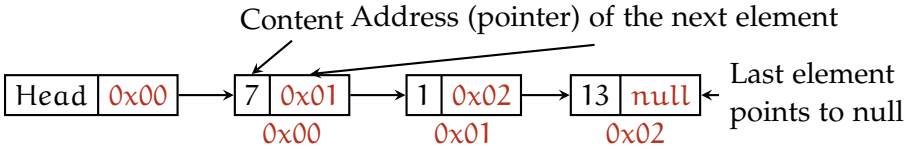


Figure 5: An example of a linked list with the data and the reference to the next one.

In a linked list the order of the elements is not assured.

Operations like adding or removing an element in a linked list are very efficient, because it is enough to change the references of the elements involved in the operation. For example, to add a new element it is enough to

change the previous element's reference to the new one, and set the reference of the new element to the next one (Figure 6). Instead, when an element is removed it is enough to update the previous element's reference to the next element of the removed one (Figure 7).

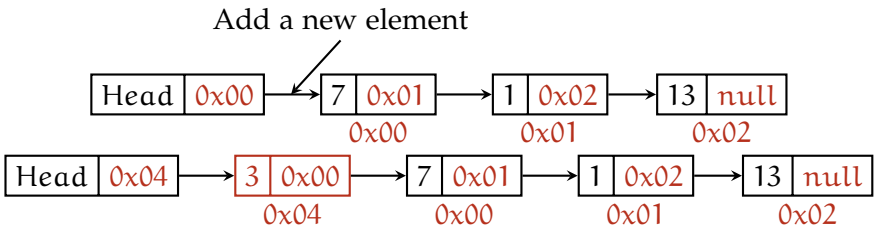


Figure 6: Add a new element to the linked list. This operation is very efficient since it is enough to change the references of the previous element to the one just added, and set the reference of the new one to its next element.

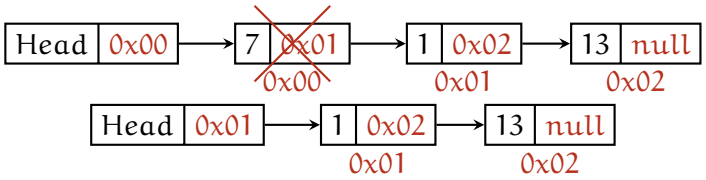


Figure 7: Remove an element of a linked list.

The complexity for adding or removing an element is constant $O(1)$.

Linked lists can be also **doubly linked lists** (Figure 8), in which every element has the reference to the previous and the next one of the collection [11].

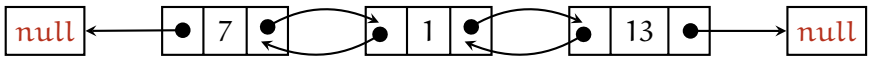


Figure 8: A doubly linked list.

Linked lists are used for implementing other data structures such as lists, stacks, queues, and associative arrays.

2.4.1 Linked List Implementation

The following code is the implementation of a linked list in Python.

Listing 2.1: Linked List implementation.

```
1 class Element():
2
3       def __init__(self, value):
4             self.value = value
5             self.next = None
6
7 class LinkedList():
8
9       def __init__(self, head=None):
```

```
10            self.head = head
11
12      def append(self, new_element):
13            current = self.head
14            if self.head:
15                  while current.next:
16                        current = current.next
17                  current.next = new_element
18            else:
19                  self.head = new_element
20
21      def get_position(self, position):
22            counter = 0
23            current = self.head
24            if position < 1:
25                  return None
26            while current and counter <= position:
27                  if counter == position:
28                        return current
29                  current = current.next
30                  counter += 1
31            return None
32
33      def insert(self, new_element, position):
34            counter = 1
35            current = self.head
36            if position > 1:
37                  while current and counter < position:
38                        if counter == position - 1:
39                              new_element.next = current.next
40                              current.next = new_element
41                        current = current.next
42                        counter +=1
43            elif position == 1:
44                  new_element.next = self.head
45                  self.head = new_element
```

2.5 STACK

A **stack** is an abstract data type belonging to collections in which only two operations are allowed [12]:

- addition of an element at the top of the stack (**push**),
- deletion of the newest element of the stack (**pop**).

In stacks only the element at the top can be modified. The complexity remains constant while adding or removing the element. Stacks usually use the **Last In, First Out** (**LIFO**) methodology.

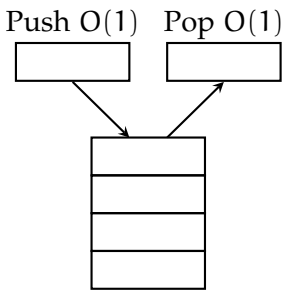


Figure 9: In a stack only the element at the top can be modified.

2.5.1 Stack Implementation

The following is the implementation in Python of a stack using the linked list.

Listing 2.2: Stack implementation.

```
class Element():
      ...

class LinkedList():
      ...

      def insert_first(self, new_element):
            new_element.next = self.head
            self.head = new_element

      def delete_first(self):
            if self.head:
                  delete_element = self.head
                  temp = deleted_element.next
                  self.head = temp
                  return deleted_element
            else:
                  return None

class Stack():

      def __init__(self, top=None):
            self.ll = LinkedList(top)

      def push(self, new_element):
            self.ll.insert_first(new_element)

      def pop(self):
            self.ll.delete_first()
```

2.6 QUEUE

A **queue** is an abstract data type belonging to collections and very similar to the stacks. For a queue the admitted operations are only on the oldest

element, thus the first added element [13]. The allowed operations on a queue are:

- addition of an element at the back of the queue (**Enque**),
- deletion of the element at the front of the queue (**Deque**),
- observation of the element at the front of the queue (**Pick**).

For addition and deletion operations the complexity remain constant. Queues usually use the **First In, First Out** (**FIFO**) methodology.

The most efficient way to implement a queue is by using linked lists.

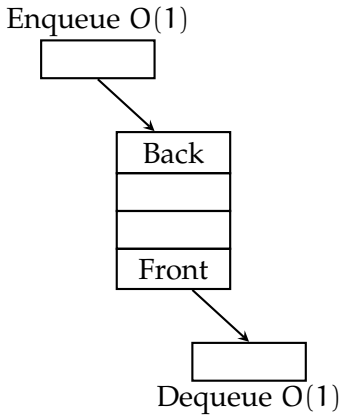


Figure 10: In a queue an element can be added only at the back (enqueue) and can be removed only at the front (dequeue).

A generalization of queues and linked lists are **deques**, in which it is possible to perform dequeue and enqueue operations on both the **back** and **front** element.

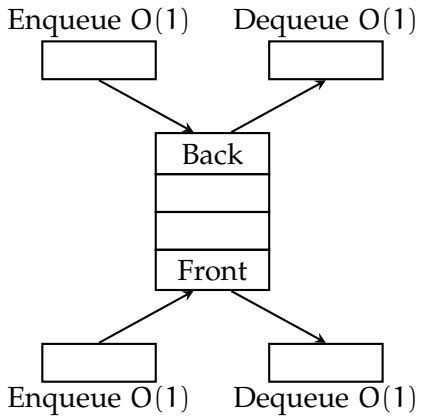


Figure 11: In a deque the operations can be done on both the back and the front.

Another modification of queue is the **priority queue**, in which each element has a priority (a numeric value that indicates its importance). When an element of the queue is removed (dequeue), the element to be removed is the one which has the highest priority. In case two or more elements have the same priority the oldest one is removed.

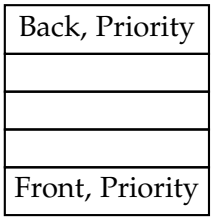


Figure 12: A priority queue.

2.6.1 Queue Implementation

The following is the implementation of a queue using the list of Python.

Listing 2.3: Queue implementation.

```
class Queue():
      def __init__(self, head=None):
            self.storage = [head]

      def enqueue(self, new_element):
            self.storage.append(new_element)

      def peek(self):
            return self.storage[0]

      def dequeue(self):
            return self.storage.pop(0)
```

2.7 SET

A **set** is an abstract data type in which unique values are stored without a particular order [14].

2.8 MAP

A **map**, **associative array**, **symbol table**, or **dictionary** is an abstract data type composed of a collection of (key, value) pairs [15]. The collection of the keys is a set, then each element is a unique value. The map is a very useful data type, and it is used in a lot of different situations.

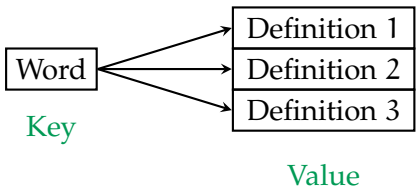


Figure 13: An example of a map.

2.9 HASH TABLE

A **hash table** or **hash map** is a data structure that implements the map abstract data type [16]. The hash table uses the **hash function** for working [17]. The usage of the hash function is called hashing.

2.9.1 Hashing

Hashing is an operation that allows to map data of an arbitrary size to a fixed-size values, called **hash values**. The necessary time for finding a value inside an ordered or unordered collection grows linearly with its size. By using a hash function to index the values with more efficient ones, the search of an element can be done in nearly constant time. Moreover, hashing avoids the exponential storage requirements of direct access of state spaces of large or variable-length keys.

Let us consider an array which stores big random numbers. A simple way for hashing these numbers, for example, is to consider only the last digits (56 and 17) and divide them for a fixed number. The reminder of the division is used as the new index for the hash table where the numbers are stored (Figure 14).

But what happen when the hash function transforms two different numbers into the same one? In this case we have a **collision** and for solving this problem we have several strategies. One way might be to use a better hash function, or, alternatively, we might use an array, called **bucket**, for storing different values associated to the same key. The worst case complexity for a search in the bucket is $O(n)$ (Figure 14).

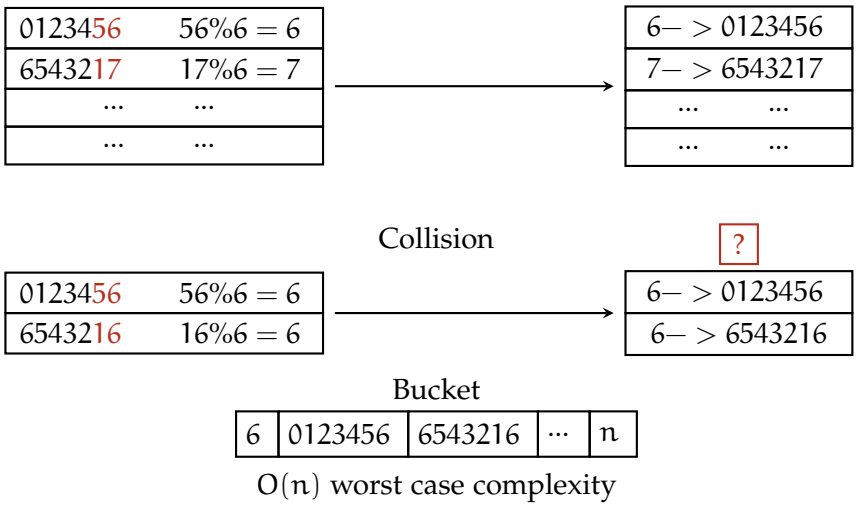


Figure 14: A hash map, and an example of collision and a possible way to solve it using the bucket method.

It does not exist a perfect hash function and a trade off must always be reached. A way to analyse a hash function is by using the **Load Factor** defined as: $\texttt{Load Factor} = \#\ \texttt{of entries} / \#\ \texttt{of buckets}$. For example, if we want to save 10 values using 1000 buckets the $\texttt{LoadFactor}$ is equal to 0.01,

leaving most of the buckets empty. In this case it is convenient to change the hash function by using less buckets. More the LoadFactor is closed to 0 more the hash table will be empty (**sparse**), and more the LoadFactor is closed to 1 more the hash table will be full and efficient. In case the LoadFactor is bigger than 1 there is the certainty that there will be some collisions.

Another example of hash function: let us consider a hash function that divide the numbers by 100. If we consider 100 values all multiple of 5, we have $\text{Load Factor} = \#\text{ of entries}/\#\text{ of buckets} = 100/100 = 1$. This way is very slow, and a different configuration should be used to make this more efficient. For example, if we use 107 buckets, we still avoid collisions, and we do not have the hash map too much sparse.

2.10 STRING KEYS

Let us consider a hash function that associates a word to a numerical value. For example, we can use the ASCII character encoding of the first two letters of the string as numerical value. Thus, in case of *UDACIY* we have *U*=85 and *D*=68. The hash function we can use might be: $S[0] * 31^{(n-1)} + S[1] * 31^{(n-2)} + \ldots + S[n-1]$, where n is the length of the string. In this way for the word in the previous example, in which only the first two letters are encoded in a numerical value for simplicity, we have $85 * 31^1 + 68 = 2703$.

This hash function works very well because it assures a very low probability of collision. 31 is a number empirically obtained by several researches and showed good results in hashing strings.

2.10.1 String Keys Implementation

The following is the implementation of the string key map in Python.

The hash function of this implementation is: $\text{Hash Function} = (\text{ASCII}[0] * 100) + \text{ASCII}[1]$. In the following code *ord()* and *char()* of the standard library of Python are used. *ord()* takes a char as an argument and returns the respective ASCII code ($\text{ord}('U') = 85$), and *char()* takes a numeric value as ASCII code and returns the respective char ($\text{char}(85) = 'U'$).

Listing 2.4: String key implementation.

```
1  class HashTable():
2        def __init__(self):
3              self.table = [None]*10000
4
5        def store(self, string):
6              hv = self.calculate_hash_value(string)
7              if hv != -1:
8                    if self.table[hv] != None:
9                          self.table[hv].append(string)
10                   else:
11                         self.table[hv] = [string]
12
13       def lookup(self, string):
14             hv = self.calculate_hash_value(string)
```

```
15            if hv != -1:
16                  if self.table[hv] != None:
17                        if string in self.table[hv]:
18                              return hv
19            return -1
20
21      def calculate_hash_value(self, string):
22            value = ord(string[0]*100) + ord(string[1])
23            return value
```

3 SEARCHING AND SORTING

In this chapter the most important and used algorithms about **searching** (retrieve some data stored in a particular data structure) [18], and **sorting** (put in a certain order some data) [19] are discussed.

3.1 BINARY SEARCH

Let us consider the problem of finding a number in an array sorted in **ascending** order (Figure 15).

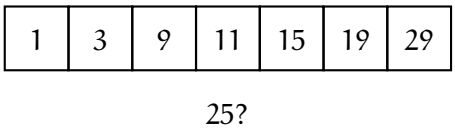


Figure 15: An array with numeric values ordered in ascending order.

A first way to tackle this problem might be to check all the numbers of the array. In other words, this method consists in performing a loop all over the elements of the array and check one by one each number. The complexity of this method is $O(n)$, since in the worst case we have to look at all the elements of the array.

A more efficient way to search an element in an array sorted in ascending order is the **binary search algorithm** [20]. In this algorithm the first element to be checked is the central value of the array (in case the array has an even number of elements there are two central values, and one can choose the bigger or the lower equally). If the central element is the one we are looking for the search ends, otherwise, since the array is sorted with an ascending order, we can ignore one half of the array. If the central number is bigger than the number we are looking for, we will ignore the right half of the array, instead, if the central number is lower than the number we are looking for we will ignore the left half of the array. This procedure is repeated: we consider the central value of the new subarray, which has half of the size of the previous step, and we check if that value is the one we are looking for. If not, the process is repeated until or we find or we do not find the desired element (Figure 16).

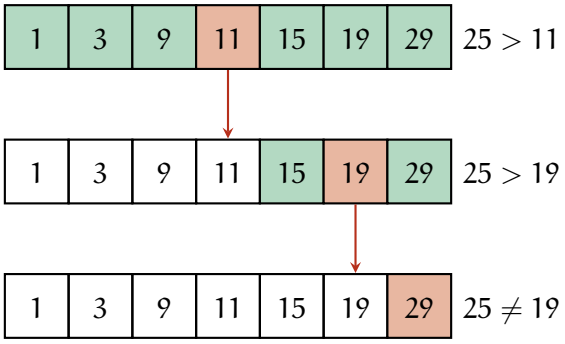


Figure 16: Binary search algorithm.

3.1.1 Complexity of the Binary Search Algorithm

For evaluating the complexity of this algorithm we have to evaluate the number of steps required for solving the problem. We already know that the complexity will not be $O(n)$, since in this algorithm not all the elements of the array are checked. Typically, the way for evaluating the complexity of an algorithm is to execute its implementation by varying the size of the input, and evaluating how many operations are done in the worst case. In the binary search algorithm the worst case is when the size of the array becomes one, so we find or we do not find the desired element in the last step.

Table 1: The number of iterations grows of one every power of two of the array size, in other words it grows as $\log(n)$.

		2^0	2^1		2^2				2^3
Array Size	0	1	2	3	4	5	6	7	8
Iterations (worst case)	0	1	2	2	3	3	3	3	4

We observe that the exponent is the number of iterations minus one, then the logarithm base two of the array size plus one is the number of iterations:

$$\#iterations = \log_2(array\ size) + 1 = \log_2(n) + 1$$

In general it is used $\log$, and is said that the binary search algorithm has a complexity of $\log(n)$.

3.1.2 Binary Search Implementation

The following code is the Python implementation of the binary search algorithm.

Listing 3.1: Binary search Python implementation.

```
1 def binary_search(array_input, value):
2       low = 0
3       high = len(array_input) - 1
4       while low <= high:
5             mid = (low + high) // 2
6             if array_input[mid] == value:
```

```
                return mid
            elif value > array_input[mid]:
                new_low = mid + 1
            else:
                new_high = mid - 1
    return -1
```

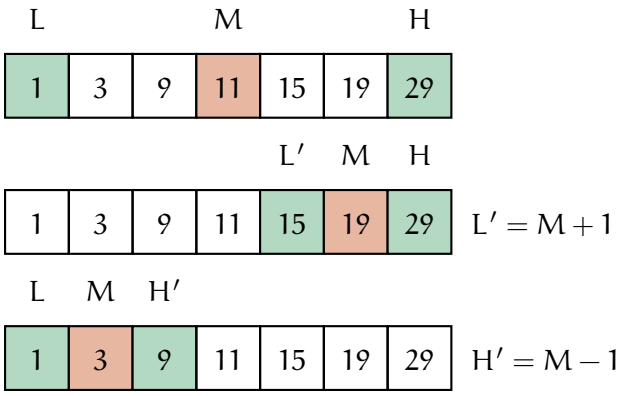


Figure 17: Array splitting in the implementation of the binary search algorithm.

3.2 BUBBLE SORT

Bubble sort is the easiest sorting algorithm working on arrays. Bubble sort works by swapping two elements at each step if they are in the wrong order, and it repeats this process until all the array is completely ordered [21]. In this way, the bigger elements tend to move at the bottom of the array, like bubbles that move at the top of a water bottle (Figure 18).

3.2.1 Complexity of the Bubble Sort Algorithm

For ordering an array using the bubble sort $n-1$ iterations are required for every step. The number of total steps are $n-1$, thus the total number of operations to be executed for ordering an array is $(n-1)*(n-1) = n^2-2n+1 = O(n^2)$.

In summary:

- **Worst Case**: $O(n^2)$
- **Average Case**: $O(n^2)$
- **Best Case**: $O(n)$. The array is already ordered, and it is enough to cycle over all the elements.

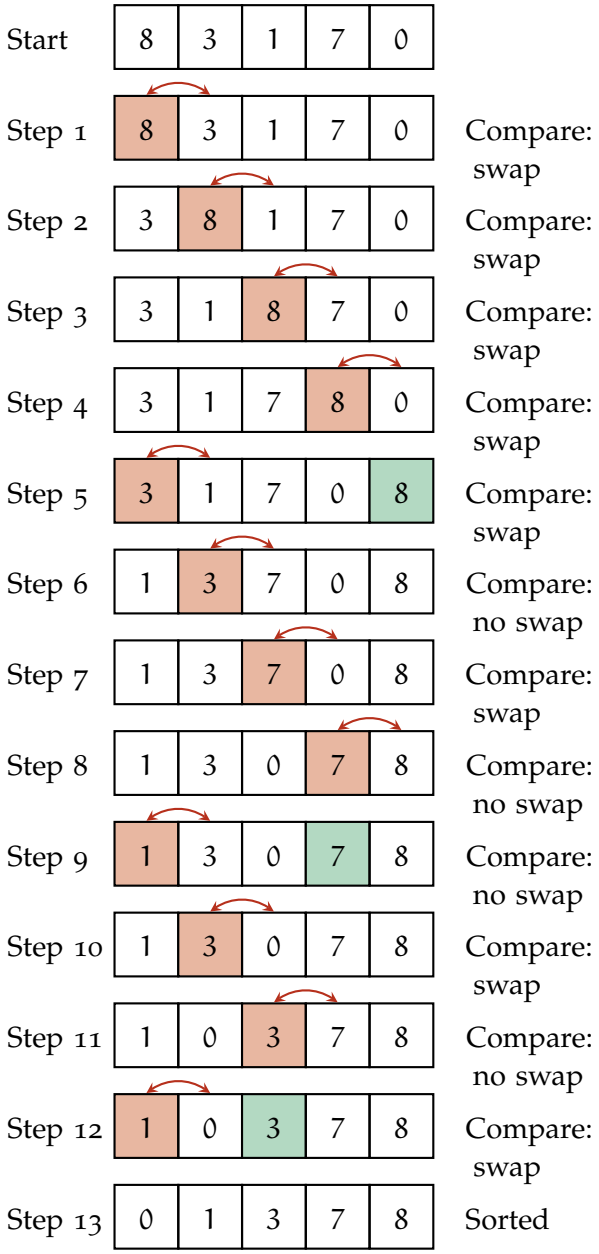


Figure 18: The swapping process is repeated until the array is completely ordered.

3.2.2 Bubble Sort Implementation

The following code is the Python implementation of the bubble sort algorithm.

Listing 3.2: Bubble Sort Python implementation.

```
1 def bubble_sort(array_input):
2       index = len(array_input) - 1
3       sorted = False
4
5       while not sorted:
6             sorted = True
7             for i in range(0, index):
```

```
8                   if array_input[i] > array_input[i + 1]:
9                         sorted = False
10                        array_input[i], array_input[i + 1] = array_input
                              [i + 1], array_input[i]
11      return array_input
```

3.3 MERGE SORT

The **merge sort** algorithm works by dividing the array in single elements at first and grouping and ordering all the elements two by two. After the first step we will have a lot of subarrays of two elements. The next step is to merge all these subarrays and to order the elements. The merging and ordering process is repeated until the array is unified again [22]. This way of reducing a big problem in several smaller problems is called **Divide et impera** (Divide and conquer).

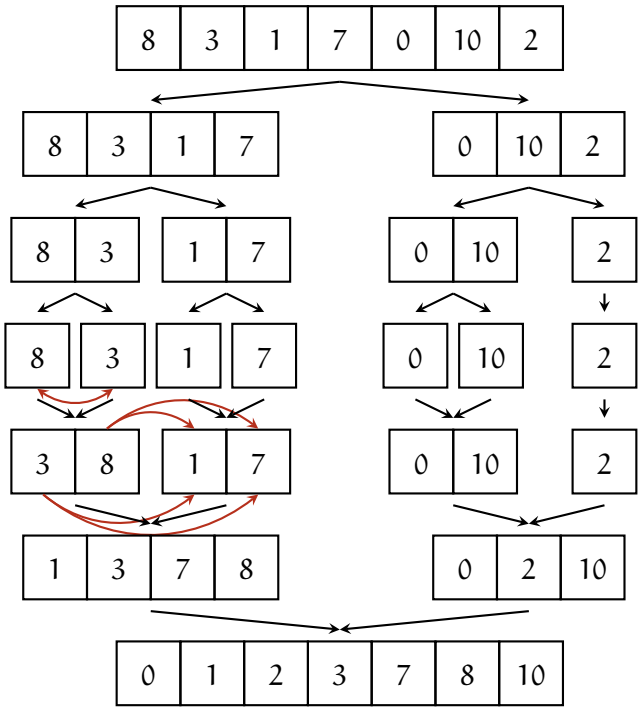


Figure 19: In merge sort algorithm the merging and sorting process is repeated until the array is again unified with all the elements sorted.

The steps are (Figure 19):

1 Divide the array in subarryas of one element. Merge two by two and order the elements. The number of the subarrays is odd, so one array at this step is not merged. **The number of comparisons for this step is 3**.

2 Merge and order again the new subarrays. For ordering in this case we start from the first element of the array on the left, and we compare this value with all the elements of the array on the right. If the first element is bigger than the picked one from the array on the right is

moved, otherwise is not moved and we go to the next element. **The number of comparisons for this step is 5.**

3 Merge and order again as in the previous step. **The number of comparisons for this step is 6.**

3.3.1 Complexity of the Merge Sort Algorithm

For evaluating the complexity of this algorithm we have to count the number of iterations and comparisons done. By using the example showed in Figure 19 we will try to extrapolate a general pattern for an array of dimension n.

The number of comparisons depends by the size of the array. For an array of two elements the number of comparisons is one, for one of three elements are two, for one of four are three, and for one of seven are six. It is impossible to calculate in general the number of comparisons, but it is possible to calculate the worst case given the array dimension. From the previous example we see that for each step the maximum number of comparisons is seven, the size of the array. The reason is that the sum of all subarrays is seven. In general the sum of all subarrays is always the size of the array. Thus, the total complexity is $O(\#\ of\ comparison\ *\ \#\ of\ iterations\ )$.

How many iterations are required? In our example for an array of seven elements the iterations required are three. While from the subprocess we observe that for an array of size four the number of iterations are two, for one of size three are two, and for one of size two is one. Thus, we can create the following table:

Table 2: The number of iterations grows of one every power of two, in others words it grows as $\log(n)$.

	2^0	2^1		2^2				2^3	
Array Size	1	2	3	4	5	6	7	8	9
Iterations (worst case)	0	1	2	2	3	3	3	3	4

By considering also the number of comparisons, the complexity is $O(n\ \log(n))$, which is better than $O(n^2)$ of the bubble sort.

The memory complexity in this case is bigger than the bubble sort algorithm. For the merge sort some subarrays (in the worst case are n) are used, and they need to be stored in the memory.

3.3.2 Merge Sort Implementation

The following code is the Python implementation of the merge sort algorithm.

Listing 3.3: Merge sort Python implementation (the recursive part is taken from Merge Sort, GeeksforGeeks).

```
1 def merge_sort_iterative(array_input):
2
3       if len(array_input) > 1:
```

```
            mid = len(array_input)//2
            left_side = array_input[:mid]
            right_side = array_input[mid:]

            merge_sort(left_side)
            merge_sort(right_side)

            i = 0 # Left side index
            j = 0 # Right side index
            k = 0 # Sorted array index

            while i < len(left_side) and j < len(right_side):
                  if left_side[i] < right_side[j]:
                        array_input[k] = left_side[i]
                        i+= 1
                  else:
                        array_input[k] = right_side[j]
                        j+= 1
                  k+= 1

            # Adding all elements if some of
      # them have been left behind
      while i < len(left_side):
            array_input[k] = left_side[i]
         i+= 1
         k+= 1

      while j < len(right_side):
            array_input[k] = right_side[j]
         j+= 1
         k+= 1

      return array_input

# Recursive implementation
def merge(left, right):
   if not len(left) or not len(right):
      return left or right

   result = []
   i, j = 0, 0
   while (len(result) < len(left) + len(right)):
      if left[i] < right[j]:
         result.append(left[i])
         i+= 1
      else:
         result.append(right[j])
         j+= 1
      if i == len(left) or j == len(right):
         result.extend(left[i:] or right[j:])
         break

   return result

def mergesort(list):
```

```
59    if len(list) < 2:
60       return list
61
62    middle = len(list)/2
63    left = mergesort(list[:middle])
64    right = mergesort(list[middle:])
65
66    return merge(left, right)
```

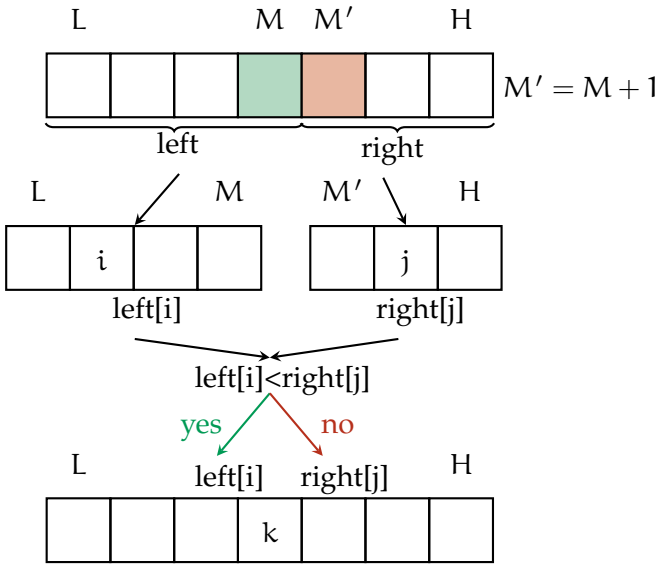


Figure 20: Merge sort algorithm implementation.

3.4 QUICKSORT

The **quicksort** is a sorting algorithm of the divide et impera class. It works by randomly choosing an element of the array, called pivot, and putting all the bigger and lower values on its right and on its left respectively. This procedure is repeated recursively on the two new subarrays until all the elements have been a pivot [23].

Here are the steps of the quicksort algorithm based on the example of Figure 21:

1 Let us consider the last element as pivot, 2 in this case, and let us compare it with all the elements on its left. Let us start from the first element of the array, and let us compare their values. In this case $8 > 2$, so 8 is moved on the position of the pivot (2) which is moved of one position to the left. The number to be removed, 10 in this case, is moved in the first position.

2 Let us repeat the process. Now we have to compare the pivot (2) in the new position with the first element (10), and because $10 > 2$ we repeat the previous step of moving the elements.

3 In this case $0 < 2$ so we do not have to do anything, and then we move to the next element, 3 in this case.

4 For the pivot 2 all the elements on the left are less than it, and all the elements on the right are bigger than it. 2 is not moved anymore. At this point we can change the pivot and repeat all the steps for the new pivots until all the elements have been a pivot.

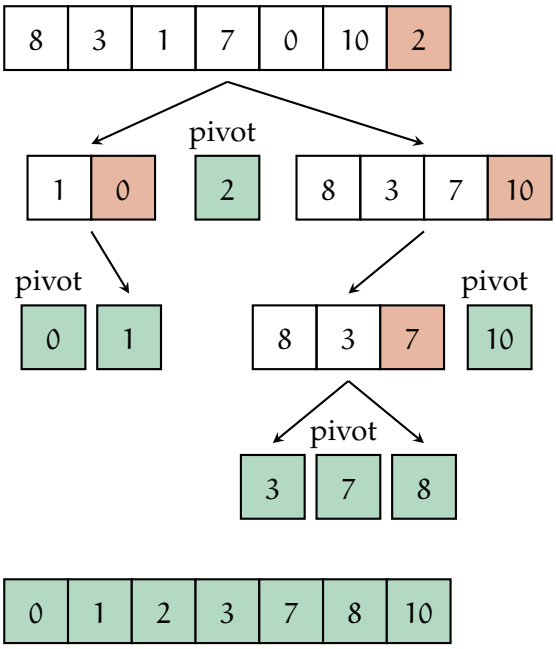


Figure 21: Quicksort algorithm.

3.4.1 Complexity of the Quicksort Algorithm

Evaluating the complexity of quicksort is very hard. In the following there are some justifications for the worst, the best, and the average case complexity.

Let us consider first the worst case complexity. In this situation the last elements of the array are the bigger ones, so it is necessary to check all the previous elements, by doing n^2 comparisons Figure 22.

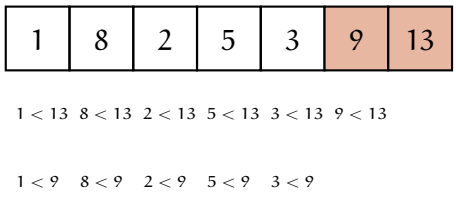


Figure 22: Quicksort algorithm worst case.

In the best and in the average case the complexity is $O(n\log(n))$. The reason is because the first pivot tends to move at the center of the array, obtaining in this way two subarrays. In turn the pivots of these two subarrays will tend to move at their center (Figue 23), and the process is repeated until all the elements have been a pivot, and eventually obtaining an ordered array.

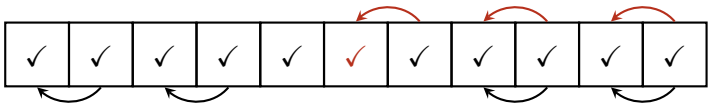


Figure 23: Quicksort algorithm best and average case.

The space complexity of quicksort is constant, $O(1)$.

This algorithm can be optimized in several ways. For example it is possible to order at the same time two half of the array, or consider as pivot always the last elements.

3.4.1.1 Quicksort and Merge Sort Comparison

Quicksort is often a better solution than merge sort, because even if its worst case complexity is $O(n^2)$, this problem can be solved by using the randomized quicksort. If the right pivot is chosen the problem related to having a worst case performance is solved. Moreover, the quicksort algorithm dose not require any auxiliary memory, which is a big advance in a lot of situations.

On the other hand, merge sort is a better solution than quicksort and heapsort [24] when the sorting is done on linked lists that do not require big auxiliary space, and on very large data sets stored on slow-to-access media, such as disk storage or network-attached storage [23].

In summary:

- **Worst Case**: $O(n^2)$
- **Average Case**: $O(n \log(n))$
- **Best Case**: $O(n \log(n))$
- **Space**: $O(1)$

3.4.2 Quicksort Implementation

The following code is the Python implementation of the quicksort algorithm [25].

Listing 3.4: Quicksort Python implementation.

```
def partition(arr, low, high):
    i = (low-1)    # Index of smaller element
    pivot = arr[high] # pivot

    for j in range(low, high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # Increment index of smaller element
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

```

```
15    arr[i+1], arr[high] = arr[high], arr[i+1]
16    return (i+1)
17
18 # Function to do Quick sort
19 def quick_sort(arr, low, high):
20    if len(arr) == 1:
21       return arr
22    if low < high:
23
24       # pi is partitioning index, arr[p] is now
25       # at right place
26       pi = partition(arr, low, high)
27
28       # Separately sort elements before
29       # partition and after partition
30       quick_sort(arr, low, pi-1)
31       quick_sort(arr, pi+1, high)
```

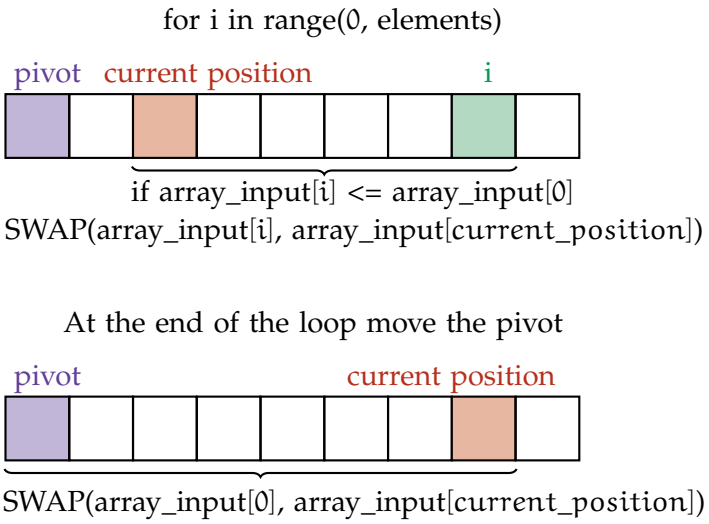


Figure 24: Quicksort algorithm implementation.

4 TREES

In this chapter are introduced the fundamental concepts and the most used algorithms about the **trees** as abstract data type.

4.1 GENERAL DEFINITIONS

A **tree** is an abstract data type that simulate a hierarchical structure [26]. A tree is a particular kind of linked list where there are more than one next elements. The base element of a tree is called **node**, and the first element is called **root**.

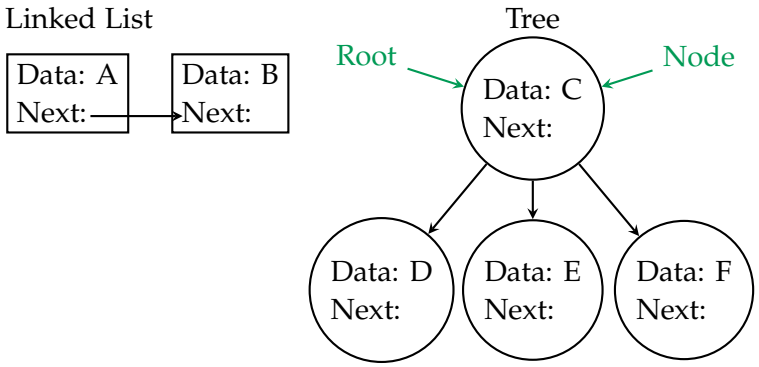


Figure 25: A linked list and a tree.

Trees must be completed connected structures (Figure 26 case (a)). This means that there are not any nodes which are not connected to anything (Figure 26 case (b)) and there must not be any cycles (Figure 26 case (c)).

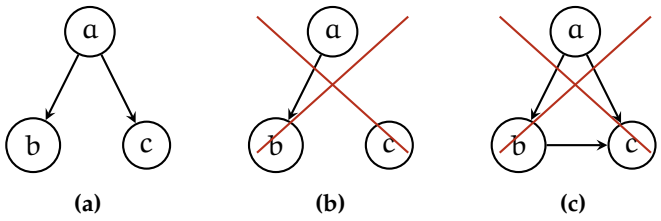


Figure 26: Completed connected structure (26a), a non completed connected structure (26b), and a cycle (26c).

Trees are hierarchical structures divided into layers: the first layer is the one belonging to the root node, the first node of a tree. The next elements of the root are called children, which in turn become parents in case they have one or more nodes connected. The last nodes of a trees do not have any children, and they are called **leaf**. The numbers of connections is called **height**. A set of connections creates a **path**. The numbers of edges starting from the root to a node is called **depth**.

In Figure 27 are reported all the previous definitions.

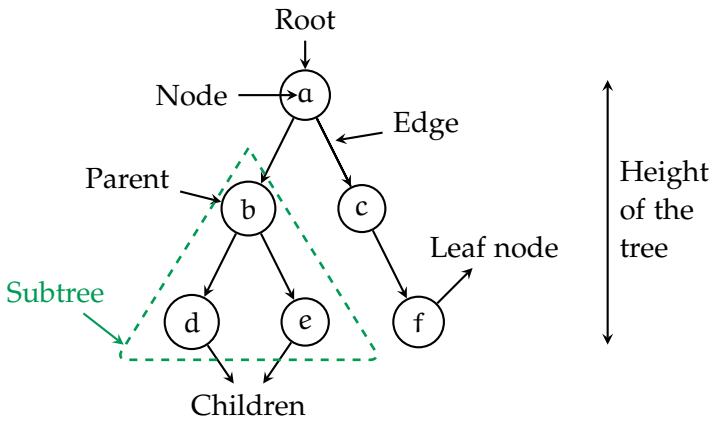


Figure 27: A tree and its fundamental elements.

4.2 TREE TRAVERSAL

Which way is the most efficient for visiting all the nodes of a tree? Is it more efficient to look at layer by layer or to look at subtrees? There are two different approaches to traverse a tree: the **depth-first search** (**DFS**), and the **breadth-first search** (**BFS**). In the first one the priority is to look at the children of a node, while in the second one is to look at the nodes of the same layer (Figure 28).

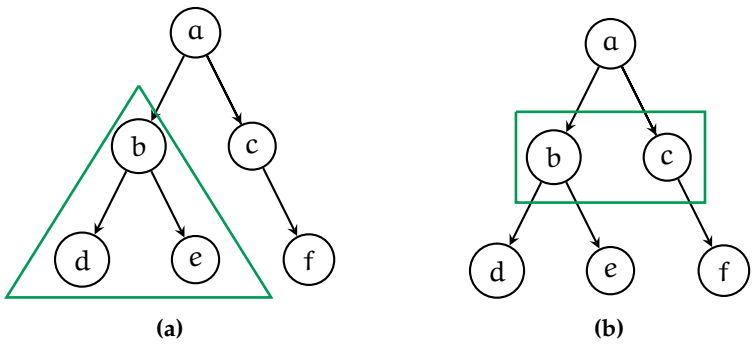


Figure 28: The depth-first search (28a) and the breadth-first search (28b)

For both ways the average complexity in case of a search is $O(|E| + |V|)$, where $|E|$ and $|V|$ are the number of edges and the number of vertices respectively. If all the tree is traversed the complexity is $O(|V|)$. The full implementation and explanation of the two methods can be found in the appendix A.

4.2.1 Depth-first search

The depth-first search has different ways to perform the search on a tree. Example of applications of this search are for solving mazes, and for scheduling problems.

PRE-ORDER SEARCH In the **pre-order search** the first node to be checked as visited is the root. The following node to be checked is the left child by convention. Once checked the left child the process is repeated until the first node without any children is reached. At this point the same process is applied to the right side: all the nodes on the right are checked until all the nodes are checked as visited (Root-Left-Right) (Figure 29).

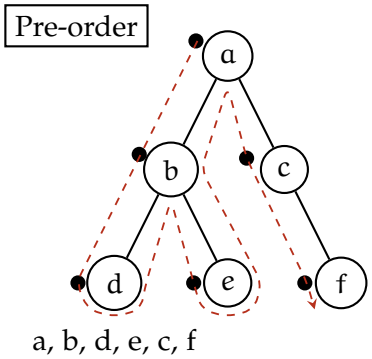


Figure 29: Pre-order search.

IN-ORDER SEARCH In the **in-order search** the first node to be checked is the first node without children on the left side (always by convention). Once checked this node the next one to be checked is its parent, and the process is repeated again on the right side nodes, until all the nodes are checked (Left-Root-Right) (Figure 30).

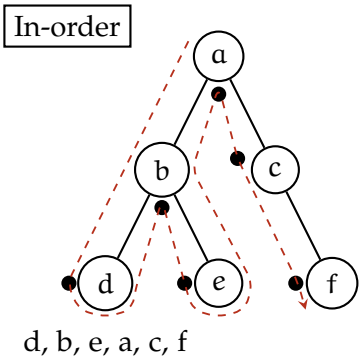


Figure 30: In-order search.

POST-ORDER SEARCH In the **post-order search** the first node to be checked is the first node without children on the left side (always by convention). Once checked this node the next one to be checked is the one which does not have any children. Once all the nodes of the current left subtree without any children are checked, the parents can be checked, and the whole process is repeated until all the nodes are checked (Left-Right-Root) (Figure 66).

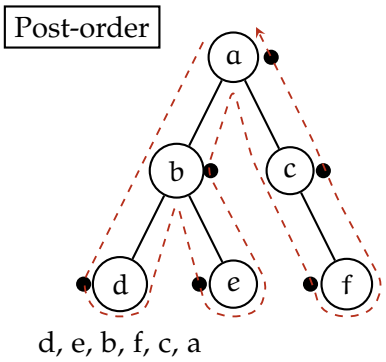


Figure 31: Post-order search.

4.3 BINARY TREES

Binary trees are trees in which the parent has at most two children (0, 1, 2 are the only number of admitted children). In binary trees some operations like searching, deleting, and inserting can be easily and efficiently done.

4.3.1 Search

To search an element in this kind of tree one of the previous methods can be used. Typically, in a tree there is no ordering, and this means that potentially all the elements of the tree can be visited. Then the complexity is $O(n)$.

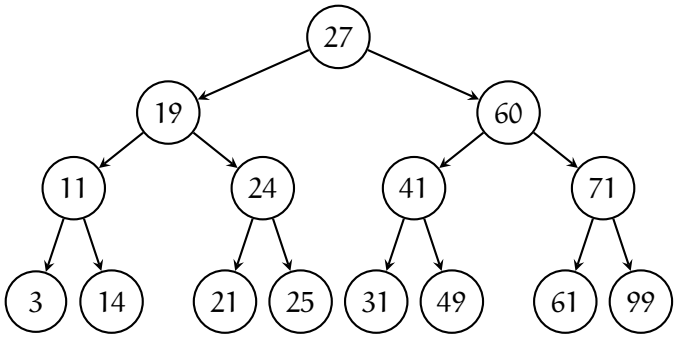


Figure 32: Example of tree search and traversal.

DFS	
Pre-order traversal	[27, 19, 11, 3, 14, 24, 21, 25, 60, 41, 31, 49, 71, 61, 99]
In-order traversal	[3, 11, 14, 19, 21, 24, 25, 27, 31, 41, 49, 60, 61, 71, 99]
Post-order	[3, 14, 11, 21, 25, 24, 19, 31, 49, 41, 61, 99, 71, 60, 27]
BRF	[27, 19, 60, 11, 24, 41, 71, 3, 14, 21, 25, 31, 49, 61, 99]

Table 3: Traversal paths for the tree in Figure 32.

4.3.2 Delete

Before the deletion of an element there is always the search for finding it in the tree. If the element to be deleted is a leaf, the deletion is not a problem, since there are not any children to be moved. Instead, if the node to be removed has some children there are several ways to reorganize the nodes, since the condition of a binary tree must remain valid (a node can have at most two children). Let us consider the tree in Figure 33.

- If the node to be removed has only one child it can be replaced by the child node (case 1 Figure 33).
- If the node to be removed has two children it can be replaced with one of the children and reorganize the others (case 2 Figure 33).
- If the node to be removed has two children, which in turn have two children as well, there are several possible options. For example, one of the child can be moved to the position of removed node (case 3 Figure 33).

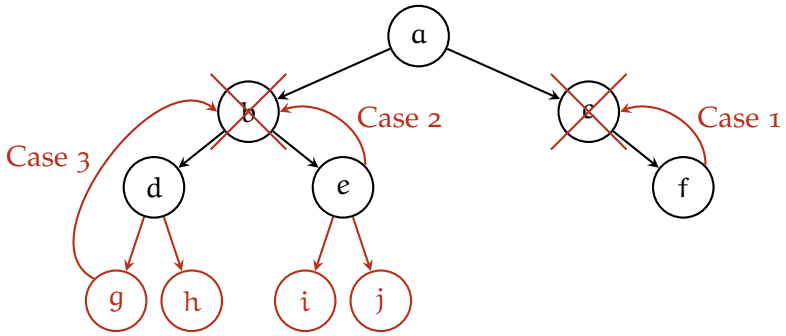


Figure 33: Deletion of an element of a tree.

4.3.3 Insert

Insert a new element to a binary tree is not a hard operation, since it is enough to find a node which can take a new children. In the worst case the farthest leaf must be reached (case 1 Figure 34).

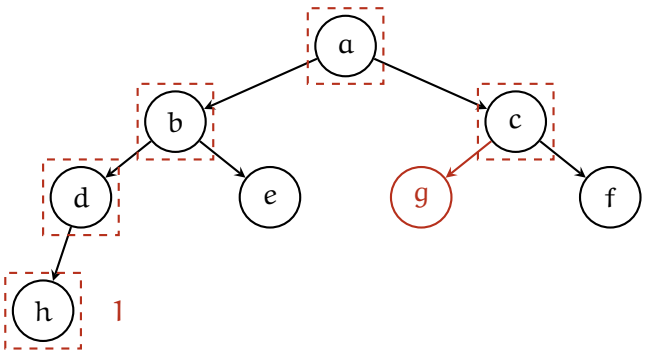
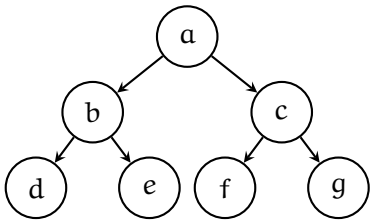


Figure 34: Addition of an element to a tree.

4.3.4 Perfect binary tree

A perfect binary tree is a binary tree in which all the nodes have two children except the leaf which do not have any children. In this case the results in Figure 35 are valid.



Level	# of nodes	Total nodes
1	2^0	1
2	2^1	3
3	2^2	7

Figure 35: Perfect binary tree.

When a new row is added, the number of nodes double. Given a level n the total number of nodes is $2n + 1$.

4.3.5 Binary Tree Implementation

The following code is the recursive Python implementation of the pre-order search and traversal (Figure 29). The recursive and iterative implementations of the other ways of search and traverse are in appendix A.

Listing 4.1: Class definition for a node and a tree.

```
class Node():

      def __init__(self, value):
            self.value = value
            self.left = None
            self.right = None

class BinaryTree():

      def __init__(self, root):
            self.root = Node(root)
```

Listing 4.2: Recursive pre-order traversal and search implementation.

```
class Node():
      ...

class BinaryTree():
      ...

      def print_tree(self):
            return self.preorder_print(tree.root, "")[:-1]

      def preorder_search_recursive(self, start, find_val):
            if start:
                  if start.value == find_val:
                        return True
                  self.preorder_search_recursive(start.left, find_val)
```

```
15                self.preorder_search_recursive(start.right, find_val)
16
17      def preorder_print(self, start, traversal):
18            if start:
19                  traversal += (str(start.value) + "-")
20                  traversal = self.preorder_print(start.left, traversal)
21                  traversal = self.preorder_print(start.right,traversal)
22            return traversal
```

4.4 BINARY SEARCH TREES (BST)

Performing operations such as search, addition or deletion of an element is a very efficient process if the binary tree is ordered. A binary tree is ordered if for all nodes the left child has a minor value, and the right child has a greater value than the considered node [27].

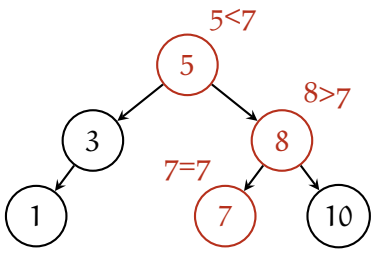


Figure 36: Search on an ordered binary tree.

For example, the tree in Figure 36 is ordered. Let us consider we would like to find the node which value is 7. Because the tree is ordered for efficiently finding 7 is enough to compare 7 with the value of the node, and choose every time if to search on the right or on the left. In this way for finding a value is not necessary to look at all the nodes, and the complexity for this operation is $\log(n)$.

Let us consider instead we would like to add a new node of value 4 to the tree of Figure 36. Because the tree is ordered, there is only one place in which the node can be added. In the case the node can be easily added since there is an empty space that can correctly take it (Figure 37). The complexity for this operation is again $\log(n)$.

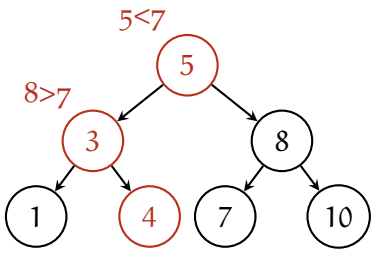


Figure 37: Addition on an ordered binary tree.

In case the position of the node is already occupied, adding the node could be a very difficult task. For a more complete and formal description of this topic refer to [28] in the chapter about trees.

When all the possible nodes are present at each level the tree is said **balanced**. In Figure 37 there is an example of a balanced tree. When instead a tree has all its children nodes with always a bigger value, all the tree is on the right, and it is said to be **unbalanced** (Figure 38). In this case the search of an element has a $O(n)$ complexity in the worst case.

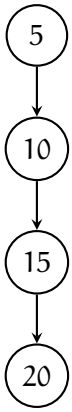


Figure 38: An unbalanced binary tree.

4.4.1 Binary Search Tree Implementation

The following code is the Python implementation of the search and addition operations performed on an ordered binary tree.

Listing 4.3: implementation of insert and search operation for a binary search tree.

```
class BST():

      def __init__(self, root):
            self.root = Node(root)

      def insert(self, new_val):
            self.insert_helper(self.root, new_val)

      def insert_helper(self, current, new_val):
            if current.value < new_val:
                  if current.right:
                        self.insert_helper(current.right, new_val)
                  else:
                        self.current.right = Node(new_val)
            else:
                  if current.left:
                        self.insert_helper(current.left, new_val)
                  else:
                        current.left = Node(new_val)

      def search(self, find_val):
            return self.search_helper(self.root, find_val)

```

```
24      def search_helper(self, current, find_val):
25            if current:
26                  if current.value == find_val:
27                        return True
28                  elif current.value < find_val:
29                        return self.search_helper(current.right,
                              find_val)
30                  else:
31                        return self.search_helper(current.left, find_val
                              )
32            return False
```

4.5 HEAPS

Heaps are a particular tree-based data structure in which the elements are ordered in increasing or decreasing order [29]. Heaps are a very efficient implementation of the abstract data type (Chapter 2) priority queue (Section 2.6). The value with the highest (or lowest) priority is always stored in the root. Heaps are very useful when an element with a high (or low) priority must be repeatedly removed from the tree. Moreover, heaps are not sorted structure, but are regarded as partially ordered.

A heap is said *max heap* (Figure 39a) if for every node its children have the values less or equal than the one of the parent (the root has always the highest value). Instead, a heap is said *min heap* (Figure 39b) if for every node its children have the values higher or equal than the of the parent (the root has the lowest value).

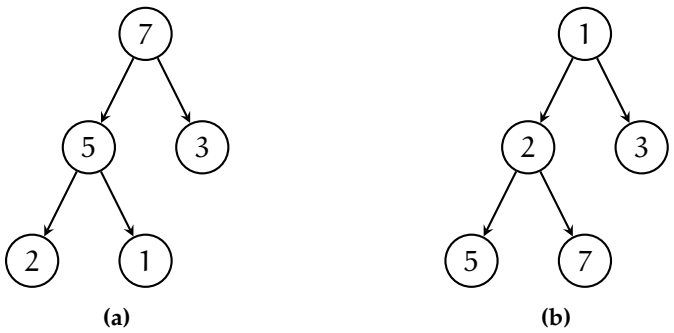


Figure 39: Max heap (39a) and min heap (39b) example.

In general heaps are not binary trees so they can have more than two nodes.

In a max heap tree the search of the biggest value is called **peek**, and it has a constant complexity ($O(1)$). For searching other values, instead, we have to check node by node, since the ordering of the left and right side is not guarantee. The average case is $O(\frac{n}{2}) = O(n)$.

4.5.1 Heapify

Let us consider we would like to add a new node to a heap. In this case the node to be added must respect the heap condition. For doing so, the

node is added in the first available position, and later the node of the heap are swapped in order to satisfy the heap condition.

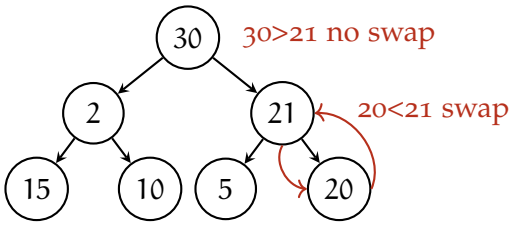


Figure 40: Addition of a new node to a heap.

In the example in Figure 40 only one swap is necessary. The complexity of adding a new element, and swap all the necessary nodes, is $\log(n)$, and in the worst case the highest number of operations correspond to the height of the tree.

4.5.2 Heap Implementation

To implement a heap we can use an array. An example of this is shown in Figure 41.

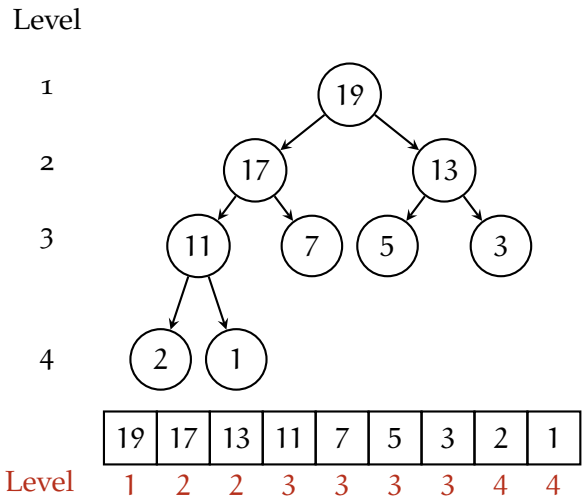


Figure 41: Implementation example of a heap using an array.

4.6 SELF-BALANCING BINARY SEARCH TREES

Self-balancing binary trees are binary tree-based structure that automatically maintain the lowest height (number of levels below the root) when an insertion or a deletion is done [30]. This structure is an efficient way to implement abstract data types like lists, priority queues, and sets.

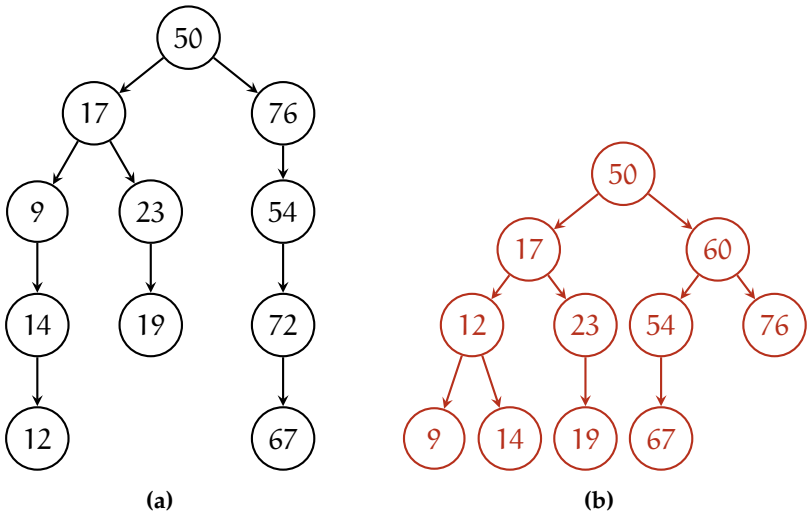


Figure 42: Example of an unbalanced (42a) and a balanced tree (42b). Credits: Self-balancing binary search tree, Wikipedia

In Figure 42 is show an unbalanced tree (Figure 42a) in which the average path to reach the leaf is 3.27 nodes. If the same tree is balanced (Figure 42b) the average path to reach any leaf from the root decreases to 3.

4.7 RED-BLACK TREES

The **red-black trees** are particular kind of self balancing trees. In this case an extra bit is used to store the color of the node, which can be only **red** or **black**. Using a color for each node assures that the tree remains balanced during insertion and deletion operations [31].

The rules that a red-black tree must respect are:

1 Nodes can be only of two types: **red** or **black**.

2 There must exist the **null leaf nodes**. Every node which does not have two leaf must have the children **null**.

3 If a node is **red** all its children must be **black**.

4 Usually the root is **black**, but this is not mandatory.

5 Every path from a the upper nodes to the lower ones must contain the same number of **black** nodes. This property is very important when inserting new nodes.

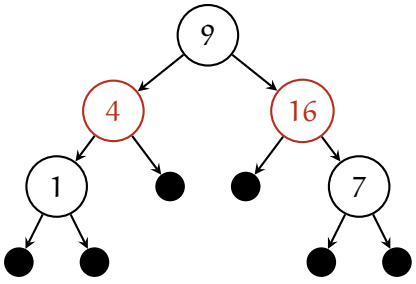


Figure 43: A red-black tree.

Usual only red nodes are added and the color of the other nodes are changed accordingly in order to respect the rules of the red-black trees.

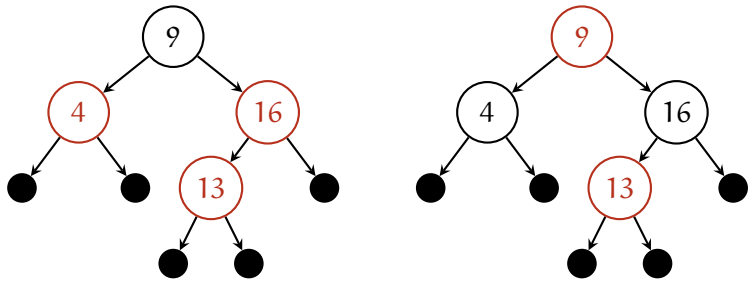


Figure 44: Insertion of a new node and the following updating of the nodes' color.

In case the insertion or the deletion does nor respect the rules of the red-black tree and BST, some more complex operations must be done as in the following figure. These operations are called rotations and can be done on the right or on the left.

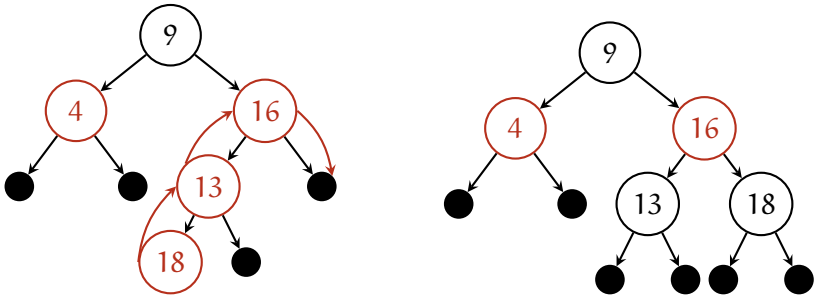


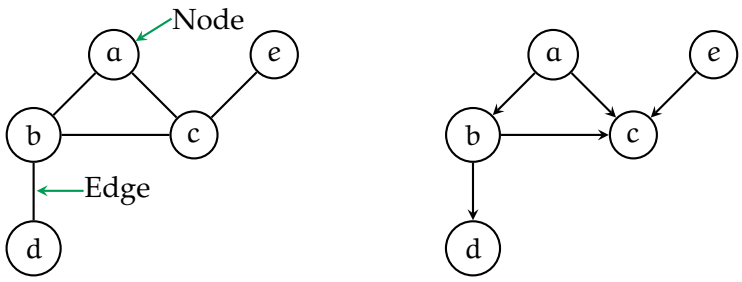
Figure 45: An example of right rotation.

5 GRAPHS

In this chapter are introduced the fundamental concepts of **graph** as a mathematical object and as an abstract data type, its applications to computer science, and the most important and notable algorithms with their implementations.

5.1 GENERAL DEFINITIONS

A graph is a discrete mathematical structure in which the connections between its elements and their relationship are highlighted. A graph is made up by two different elements: **nodes** or **vertices**, and **links** or **edges**. In case the links do not have any direction the graph is called **undirected graph** (Figure 46a), in case, instead, the links have a direction the graph is called **directed graph** (Figure 46b).



(a) Undirected graph. **(b)** Directed graph.

Figure 46: An undirected and a directed graph and its elements.

Formally a graph is the pair $G = (V, E)$, where V is the set of all vertices, and E is the set of paired vertices, whose elements are called edges [32].

A **tree** (Chapter 4) is a special kind of graph.

Graphs are very useful to describe a lot of real situations like: connections between people, computers (internet), web pages (world wide web), airports, cities, and gene inside the DNA.

In graphs, unlike trees, closed loops can exist. These kind of closed loops can be dangerous for the algorithms since they could lead to infinite executions.

5.1.1 Connectivity

Connectivity is a measure that describes how much the nodes of a graph are connected. It is defined as the minimum number of elements (nodes or edges) that need to be removed to separate the remaining nodes into isolated subgraphs [33].

A graph is said to be **connected** if every pair of nodes are connected. Thus it always exists at least one path that connects every pair of nodes. If

an undirected graph is not connected then it is **disconnected**: in this case there is one or more nodes that can not be reached by any paths.

STRONGLY CONNECTED A directed graph is said to be **strongly connected** if every pair of nodes can be reached by one or more path.

WEAKLY CONNECTED A directed graph is said to be **weakly connected** if, by replacing all the directed edges with undirectd ones, the new graph is connected. In a directed graph some nodes can not be reached if all the edges exit or enter from them. The graph in Figure 47 is weakly connected since the node g has only entering edges.

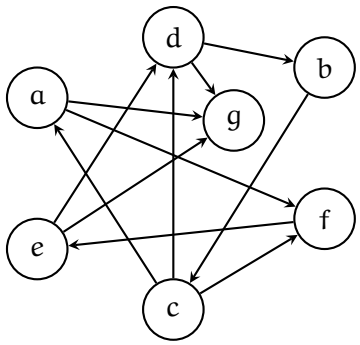


Figure 47: A weakly connected graph.

5.2 GRAPH REPRESENTATIONS

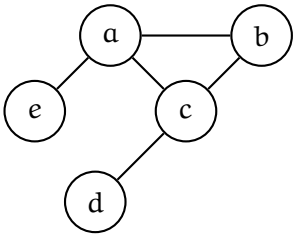
There are several ways to represent graphs using data structures. For example in an object oriented programming language a way could be to define a type for the vertex, and a type for the edges.

The most common data structures used for representing a graph are: **edge list**, **adjacency list**, and **adjacency matrix** [28].

In appendix E there is a summary of the complexities for the most common operations performed on graphs for the different data structures.

5.2.1 Edge List

The **edge list** is an unordered list of all the pairs of nodes that form an edge. This representation is minimal but it does not allow to locate a specific edge, or the set of edges incident to a particular node [28].



[[a, b], [a, c], [a, e], [b, c], [c, d]]

Figure 48: Edge list.

5.2.2 Adjacency List

The **adjacency list** is a list containing a separate list for each node containing all the incident edges to that node. In this representation identify all the edges incident to a node is easy [28]. In case of directed graphs for each node there are two different lists: one for the entering edges, and another one for the outgoing ones.

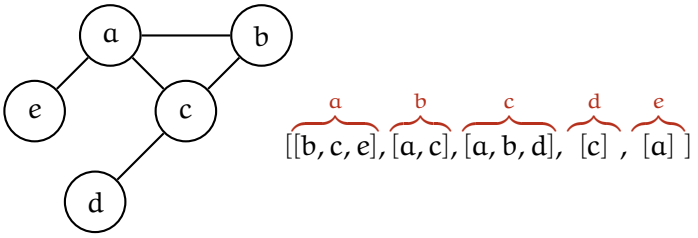


Figure 49: Adjacency list.

5.2.3 Adjacency Matrix

The **adjacency matrix** is a matrix A in which each element $A[i, j]$ represents the relationship between the edge i with the edge j. If between i and j there is an edge $A[i, j] = 1$, otherwise 0. In case an edge go out and enter in the same node $A[i, i] = 1$. For undirected graphs this matrix is symmetric, and in case there are not edges going in and out the same node the principal diagonal is all of 0.

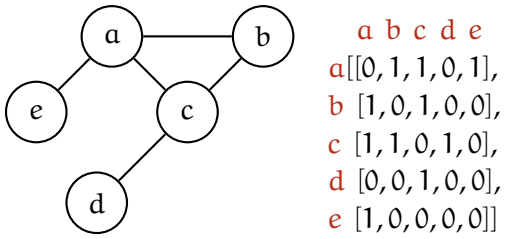


Figure 50: Adjacency matrix.

In appendix B there is the full Python implementation of the graph representations. There are defined the classes for nodes, edges, and graphs objects. Moreover, all the main operations on graphs like: insert a new node, a new edge, get the edge list, the adjacency list, the adjacency matrix, and find the max index are also implemented.

5.3 GRAPH TRAVERSAL

As for trees (Chapter 4), also for graphs there are two different ways of traversal: the **depth-first search**, and the **breadth-first search**. But for graphs, differently from trees, there is not a privileged way to traverse, and it is arbitrarily chosen a node where to start.

5.3.1 Depth-first Search (DFS)

In the **depth-fist search** the search starts from an arbitrary node, and traverse one of the connected node. This process is repeated until there are not any new nodes on that path, and starts again with a new node, until the node we are looking for is found or all the nodes have been traversed [34]. This traversal can be implemented both recursively, and iteratively by using a stack.

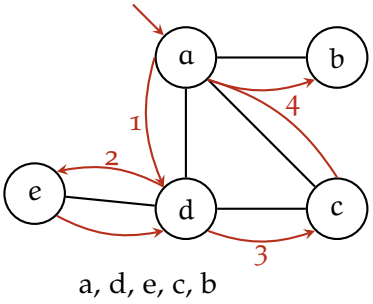


Figure 51: Depth-first search example.

Listing 5.1: Recursive implementation of a depth-first search.

```
class Graph():
      ...

      def depth_first_search_recursive(self, start_node):
            ret_list = [start_node.value]
            start_node.visited = True
            edges_out = [e for e in start_node.edges
                              if e.node_to.value != start_node.value]
            for e in edges_out:
                  if not edge.node_to.visited:
                        ret_list.extend(depth_first_search_recursive(
                            edge.node_to)
            return ret_list
```

The complexity in this case is $O(|E| + |V|)$, where E is the number of edges and V the number of vertexes. For more details on the complexities evaluation on graphs refer here [28].

In appendix C there is a detailed recursive and iterative implementation, with also an additional example.

5.3.2 Breadth-first Search (BFS)

In the **breadth-first search** the traversal is done by starting from a node, and the first visited nodes are all its neighbors. Once visited all the neighbors the traversal keeps going in the same previous way, looking to all the node's neighbors first.

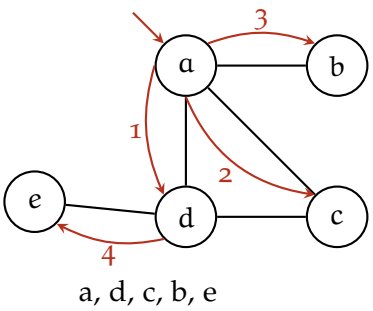


Figure 52: Breadth-first search example.

The only possible implementation is the iterative one, and it is implemented by using a queue.

Listing 5.2: Recursive implementation of a depth-first search.

```
class Graph():
      ...

      def breadth_first_search(self, start_node):
            queue = []
            queue.append(start_node)
            while queue:
            q = queue.pop(0)
            edges_out = [e for e q.edges if e.node_to.value != q.value]
            for neighbour in edges_out:
                  if neighbour not neighbour.visited:
                        neighbour.visited = True
                        queue.append(neighbour)
```

The complexity in this case is $O(|E| + |V|)$, where E is the number of edges and V the number of vertexes. For more details on the complexities evaluation on graphs refer here [28].

SPANNING TREE Using a tree it is possible to visualize how the graph is traversed. For the graph in Figure 52 the spanning tree is the following.

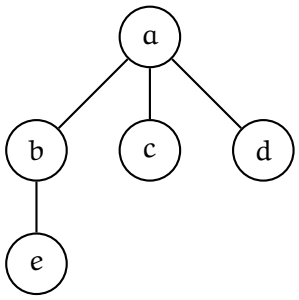


Figure 53: Spanning tree.

In appendix C there is the detailed implementation, with also an additional example.

5.3.3 Eulerian Path and Circuit

An **Eulerian path** (or **Eulerian trail**) is a path of edges that visits all the edges in a graph exactly once [35]. Not every graph has an Eulerian path and even if a graph has an Eulerian path it could be found only at specific starting nodes (Figure 54).

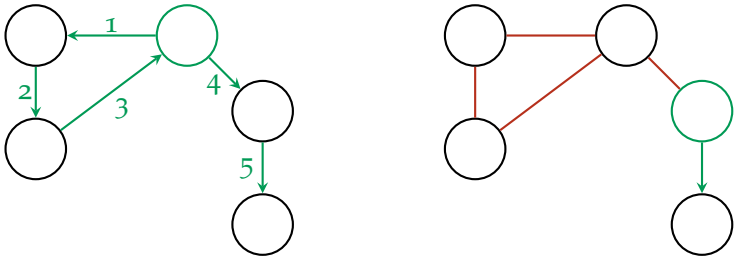


Figure 54: An Eulerian path (54a) and a not Eulerian path (54b).

An **Eulerian circuit** (or **Eulerian cycle**) is an Eulerian path which starts and ends on the same node. As before not every graph has an Eulerian circuit.

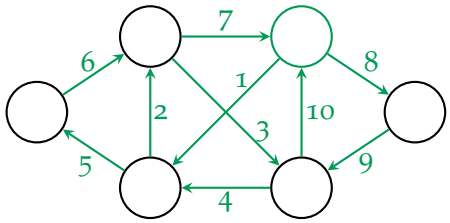


Figure 55: Eulerian circuit.

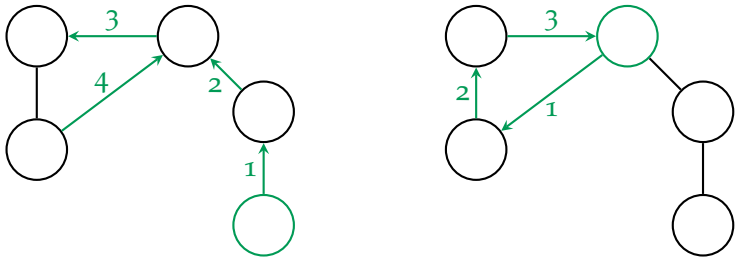
In the following table are summarized some rules for finding if a graph has Eulerian path and/or circuit [36].

Table 4: Eulerian path and circuit existence rules.

	Eulerian Path	**Eulerian Circuit**
Undirected Graph	Either every vertex has even degree or exactly two vertices have odd degree.	Every vertex has an even degree.
Directed Graph	At most one vertex has (outdegree)-(indegree)=1 and at most one vertex has (indegree)-(outdegree)=1 and all other vertices have equal in and out degrees.	Every vertex has equal indegree and outdegree.

5.3.4 Hamiltonian Path and Circuit

A **Hamiltonian path** (or **traceable path**) is a path in which all the nodes are visited exactly once. A **Hamiltonian circuit** (or **Hamiltonian Cycle**) is a Hamiltonian path that starts and ends in the same node [37].



(a) (b)

Figure 56: Hamiltonian path (56a) and Hamiltonian circuit (56b).

5.3.5 Shortest Path Problem

The **shorted path problem** is the problem of finding a path between two nodes such that the sum of the weights is minimized. In case of an undirected graph the shortest path is the path that pass through the minimum number of nodes [38]. The breadth-first search can be used for finding the shortest path from a node to all the others, but it expects that all the edges are of the same nature, and in several situations this approach is not useful. Let us image we would like to find the best path connecting two cities. In this case the links between the nodes are not the same, since some streets are faster or slower than others. Another interesting application of finding the shortest path in graphs, is about routing packages from a computer to another in order to be as fast as possible.

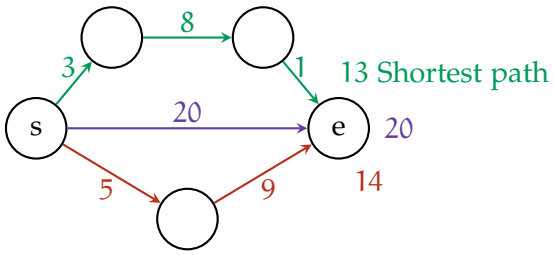


Figure 57: Shortest path.

5.3.6 Dijkstra's Algorithm

The Dijkstra's single source shortest path algorithm for a non-negative weighted graphs, finds all the shortest paths from a given node to all the others. The distance between two nodes is the sum of the weights of the edges of the path [39].

This algorithm is part of a broader class of algorithms which are called **greedy algorithms**. In the greedy algorithms the optimal solution is found

by finding the optimal solution at every step without considering the previous ones. Anyway, this approach does not always find the optimal solution of a problem. A more advanced technique to greedy algorithm is **dynamic programming**, which will be the topic of the next chapter.

RELAXATION Before introducing the Dijkstra's algorithm it is important to introduce the **relaxation condition**. Let us consider the weighted directed graph of Figure 58. The cost for going from node a to node u is 2, and ∞ from a to v, since there is not any direct link between that nodes. The relaxation condition is then:

Definition 5.1 Relaxation condition
if (d[u] + w(u, v) < d[v])
d[v] = d[u] + w(u, v)
Where $d[u]$ *is the distance of node* u *from the starting node (*1 *in this case), and* $w(u, v)$ *is the weight from the node* u *to node* v.

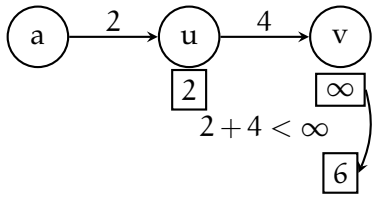


Figure 58: Relaxation condition.

In other words the relaxation condition says that if there is a shorter path connecting two nodes, this distance should be used as the new distance.

In the Dijkstra's algorithm at each step the new node is chosen based on the lowest value of the distance at that given step, and eventually the distances for all the adjacent nodes are updated accordingly the relaxation condition.

Let us consider the graph in Figure 59, and let us start from the node a [40].

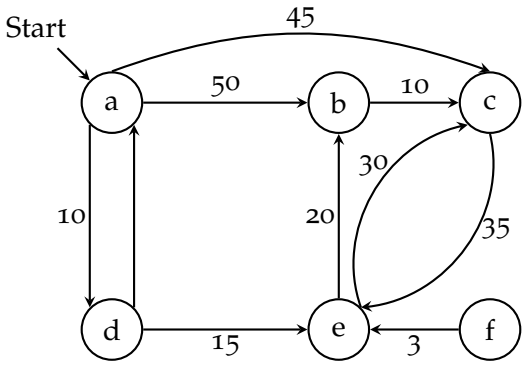


Figure 59: Dijkstra's algorithm example.

For describing how the algorithm works we use the following table, in which for each column there is a node, and at every step a new row is

added for the node for which the shortest path has just been found. At the first step the table looks like:

explored vertex	a	b	c	d	e	f
a	0_a	50_a	45_a	10_a	∞	∞

All the nodes are listed as columns, and all the distances from the node a to all the others are reported in the table. The nodes e and f do not have any directed link to the node a, then their distance is infinity. Once listed all the distances the node with the lowest one is selected: at the first step it is the node a.

At the second step the lowest distance is chosen: in this case it is for the node d, then it is added in the visited node as the shortest path from a to d is fund. The node e is now reachable and it has a distance of 15 from d, then 25 from a ($10_a + 15_{b,e}$).

visited node	a	b	c	d	e	f
a	0_a	50_a	45_a	10_a	∞	∞
d	|	50_a	45_a	10_a	25_d	∞

The process is repeated again: the lowest distance is chosen, in this case is the distance to reach the node e. As for the previous step we check for all nodes if it is shorter to pass through the new shortest path found, and in case true the weight are updated (relaxation condition). At this step the weight to be updated is the one for the node b because for reaching it from a the weight is 50 (50_a), while for reaching it passing through e (which in turn for be reached with its shortest path we have to go through d, which in turn again is reached with its shortest path from a) is 45 ($25_d + 20_{e,b} = 45_e < 50_a$).

visited node	a	b	c	d	e	f
a	0_a	50_a	45_a	10_a	∞	∞
d	|	50_a	45_a	10_a	25_d	∞
e	|	45_e	45_a	|	25_d	∞

The same process is repeated again, the new shortest path found is tor the node b passing through e with a weight of 45. In this case there are two shortest path, and for this example is the same choosing one of them.

visited node	a	b	c	d	e	f
a	0_a	50_a	45_a	10_a	∞	∞
d	|	50_a	45_a	10_a	25_d	∞
e	|	45_e	45_a	|	25_d	∞
b	|	45_e	45_a	|	|	∞

In the last step all the shortest paths are found. Only the node f has been left out since it can not be reached in any ways from a.

visited node	a	b	c	d	e	f
a	0_a	50_a	45_a	10_a	∞	∞
d	|	50_a	45_a	10_a	25_d	∞
e	|	45_e	45_a	|	25_d	∞
b	|	45_e	45_a	|	|	∞
d	|	|	45_e	|	|	∞

All the shortest paths from a are:

- b: a $\rightarrow$ d $\rightarrow$ e $\rightarrow$ b, W=45;
- c: a $\rightarrow$ c, W=45;
- d: a $\rightarrow$ d, W=10;
- e: a $\rightarrow$ d $\rightarrow$ e, W=25;
- f: ∞.

For example for finding the shortest path for reaching the node b the steps to follow are:

visited node	a	b	c	d	e	f
a	0_a	50_a	45_a	10_a	∞	∞
d	|	50_a	45_a	10_a	25_d	∞
e	|	45_e	45_a	|	25_d	∞
b	|	45_e	45_a	|	|	∞
d	|	|	45_e	|	|	∞

In appendix D the full implementation of the Dijkstra's algorithm is reported.

The complexity of this algorithm is $O(|V^2|)$ where V is the number of nodes. This is because all the nodes are relaxed with respect all the others, and in the worst case all nodes must be checked. If the implementation of the algorithm is optimized the complexity in this case is $O(|E| + |V|\log(|V|))$, where E is the number of edges.

6 DYNAMIC PROGRAMMING

In this chapter are introduced the fundamental concepts of **dynamic programming**.

6.1 GENERAL DEFINITIONS

Dynamic programming is a mathematical optimization and a computer programming method [41].

In computer science a problem which can be solved with dynamic programming must have an optimal substructure (the problem can be broken down in smaller parts), and an overlapping sub-problems (the broken down problems share some results). If a problem can be solved with an optimal substructure but with non-overlapping sub-problems, then the strategy is called **divide and conquer**.

When a problem is solved with dynamic programming, the starting point is to find the **base case**, which is the sub-problem with the lowest computational cost. Once solved this problem its result is saved in a table called **lookup table**, and the same general formula is used recursively. At each step the result is saved in the lookup table, since in the following steps might be used for calculating new results. Instead of calculating the same value again, we look at the lookup table and retrieve it. This aspect of dynamic programming makes this technique extremely powerful, since most of the calculations must not be repeated several times.

For example, in the calculation of the Fibonacci numbers we have to use the same function several times in different sub-problems. Let us suppose we want to calculate the fifth number of the Fibonacci sequence:

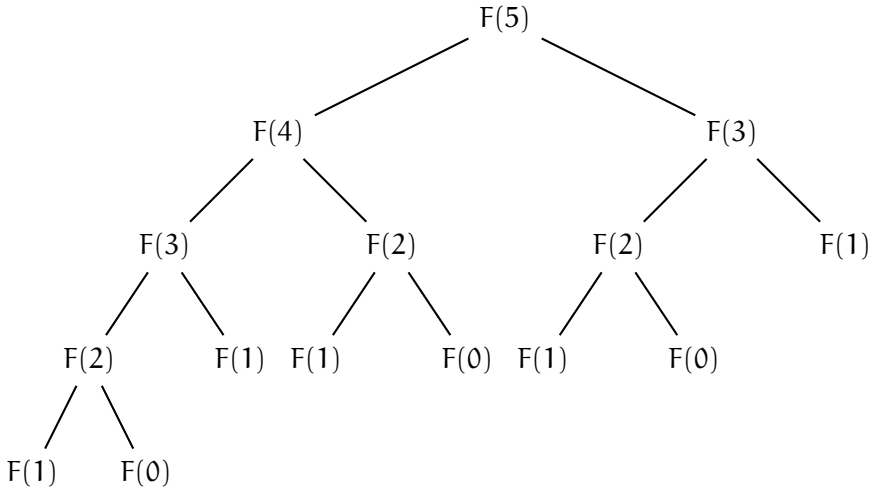


Figure 60: Calculation of the fifth number of the Fibonacci sequence.

Some values are used several times in different sub-problems as shown in the following table:

function	F(0)	F(1)	F(2)	F(3)	F(4)	F(5)
number of computation	3	4	3	2	1	1

Table 5: Number of computation for each value of the Fibonacci sequence.

If the values of the functions which are repeated are stored in a lookup table, instead of calculating them again, they are retrieved from the table and used.

6.2 KNAPSACK PROBLEM

Let us consider a knapsack with a limited capacity of weight and number of objects, which are defined by a value and a weight. Let us also consider that not all the available objects can be carried. Which objects we can put in the knapsack in order to have the optimal weight-value in the limits of the maximum weight [42]? In this case we consider to take or not the an entire object, and it is said the 0/1 **knapsack problem** . In some variants it is possible to take a fraction of an object.

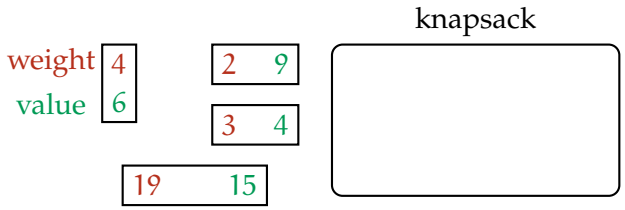


Figure 61: Knapsack problem.

The easiest way to solve this problem would be to test all the combinations of possible objects, and select the optimal value (brute force). The complexity in this case is $O(2^n)$, where n is the number of objects. The complexity of this approach is very high.

Let us try another approach instead. The idea is to maximize the overall value and have the maximum transportable weight. What we do at first is to find the combination with the lowest weight and the highest value, and add objects until the allowed weight is reached. For doing so let us create a **table of values** where in each row there is an item and in each column the maximum capacity which increases. Let us consider the following example, where we have five elements ($n = 5$), with the maximum capacity of the knapsack of 7 kg ($m = 7$kg). The values and the weights are respectively: $V = \{2, 3, 2, 4, 5\}$[\$], and $W = \{1, 2, 3, 3, 4\}$[kg].

		0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0	0
$v_1 = 2, w_1 = 1$	1	0	2	2	2	2	2	2	2
$v_2 = 3, w_2 = 2$	2	0	2	3	5	5	5	5	5
$v_3 = 2, w_3 = 3$	3	0	2	3	5	5	5	7	7
$v_4 = 4, w_4 = 3$	4	0	2	3	5	6	7	9	9
$v_5 = 5, w_5 = 4$	5	0	2	3	5	6	7	9	10

Figure 62: Knapsack problem example.

The green arrows show where the state is update, while the red ones where the state is not update (only the arrows for the last row have been drown for clarity in Figure 62). In case of a tie it is equivalent to take one of the two values (in Figure 62 for the item 5 at the capacity 4 there are two green arrow). To decide if update or not a value of the table we have to check the previous state (at $K[i-1, w]$, where K is the knapsack, i the index for the row, and w the maximum capacity) and the previous optimal state (at $K[i-1, w-w[i]]$, where $w[i]$ is the weight of the current item). The formula to be used is the following $K[i, w] = max\{K[i-1, w], K[i-1, w-w[i]] + V[i]\}$, where $V[i]$ is the weight of the current item. For example, for calculating the value at $5, 3$ the formula becomes: $K[5, 3] = max\{K[4, 3], K[4, 3-2] + V[3]\} = max\{4, 3+2\} = max\{4, 5\} = 5$, then the state is updated. In case there is some undefined positions in the previous formula the state is not updated and it is kept the previous one.

With the table in Figure 62 we found the best value, but what about the the best elements to form the subset? Starting from the last element of previous table ($K[5, 7] = 10$) the idea is to check whether the previous element is different to the previous optimal solution. If the value is different (red arrows of 63), then the previous step of the best solution is chosen, instead if the previous value is the same (green arrows of 63) we move to the previous step, and we keep doing so until a different value is found, and we repeat the same thing as before. The process stops when we reach the first element of the table.

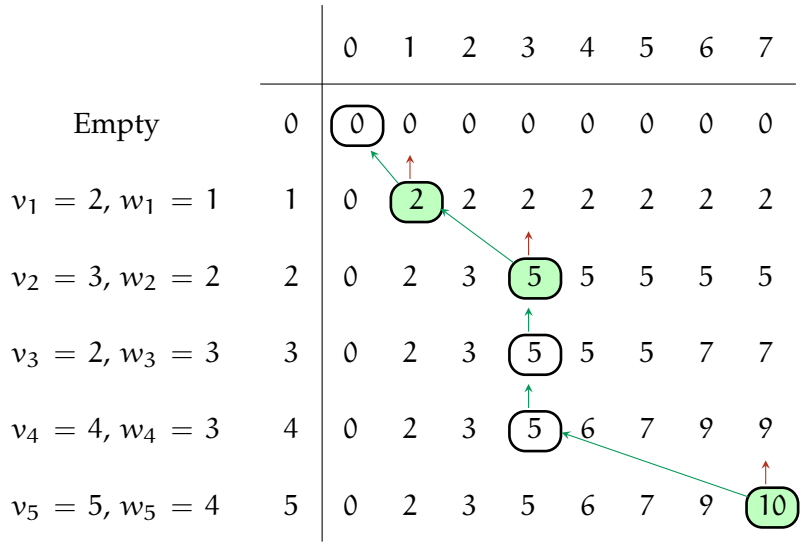


Figure 63: Selection of the optimal subset.

6.3 TRAVELLING SALESMAN PROBLEM

Given a graph and the distances between all its nodes, what is the shortest possible rout that connects all the nodes and visits them exactly once and returns to the starting node [43]?

This is a very hard problem and it belongs to the NP-Hard problems [44]. NP-Hard problems can not be solved in a polynomial time ($O(n^2)$, $O(n)$, $O(2)$, ...). The complexity of the travelling salesman problem is said to be **pseudo polynomial** [45].

There are two ways to solve this problem: the exact ones, which find an exact solution, and the approximated ones, which find an approximated solution. The latests are much faster in the execution that the first ones. If we try to solve this problem with the brute force by testing all the possible routes, we would have a complexity of $O(n!)$. There are also solutions which use the dynamic programming, such as the **Held-Karp algorithm** whose complexity is $O(n^2 2^n)$. The most common approximated algorithm is the **Christofides–Serdyukov algorithm** [46].

APPENDIX

A

BINARY SEARCH COMPLETE IMPLEMENTATION

The best way to implement the tree data structure is to create a class for the nodes (**Node**) and a class for the binary tree (**BinaryTree**). Since all the nodes of a binary trees has at most two children, in the class **Node** is enough to have three attributes: the value of the node, the left, and the right child, in turn of **Node** type as well. This way of implement a tree is very similar to the one used for the linked list (Section 2.4).

Listing A.1: Class definition for a node and a tree.

```
1  class Node():
2
3        def __init__(self, value):
4              self.value = value
5              self.left = None
6              self.right = None
7
8  class BinaryTree():
9
10       def __init__(self, root):
11             self.root = Node(root)
```

In this appendix there is a detailed implementation, both recursive and iterative, of all possible ways to perform a tree traversal in the case of **depth-first search**: pre-order traversal A.1, in-order traversal A.2, and post-order traversal A.3. The convention followed here is to look up first at the left child always, and later look up the right one. The opposite approach is equivalent.

The pseudocode of the iterative implementations is taken from [47].

A.1 PRE-ORDER TRAVERSAL

In the pre-order traversal every new node is first checked as visited, and the left subtree is traversed first, and later the right one. This process is repeated until all the nodes of the tree are visited (Section 4.2.1).

For implementing the pre-order traversal iteratively a stack (Section 2.5) is used. The steps are [47]:

Algorithm 1: Pre-order pseudocode.

```
1  Function Recursive-Preorder(node):
2      if (node == null) then
3          return
4      visit(node)
5      Recursive-Preorder(node.left)
6      Recursive-Preorder(node.right)
7
8  Function Iterative-Preorder(node):
9      s ← empty stack
10     s.push(node)
11     while (not s.isEmpty()) do
12         node ← s.pop()
13         visit(node)
14         /* Right child is pushed first so that left is
               processed first (LIFO) */
15         if (node.right ≠ null) then
16             s.push(node.right)
17         if (node.left ≠ null) then
18             s.push(node.left)
```

Listing A.2: Recursive and iterative implementation of pre-order traversal.

```
class BinaryTree():
      ...

      def recursive_print_tree(self):
            return self.preorder_recursive_print(tree.root, "")[:-1]

      def iterative_print_tree(self):
            return self.preorder_search_iterative(self, tree.root, "")

      def preorder_search_recursive(self, start, find_val):
            if start:
                  if start.value == find_val:
                        return True
                  self.preorder_search_recursive(start.left, find_val)
                  self.preorder_search_recursive(start.right, find_val)

      def preorder_search_iterative(self, start, find_val):
            if start == None:
                  return None
            visited = []
            stack = []
            stack.push(start)
            while stack: # Keep cycle until the stack is empty
                  node = stack.pop()
                  visited.append(node)
```

```
26                  if node.value == find_val:
27                        return True
28                  # Right child is pushed first so that the left one is
                        processed first
29                  if node.right:
30                        stack.append(node.right)
31                  if node.left:
32                        stack.append(node.right)
33            return visited
34
35      def preorder_recursive_print(self, start, traversal):
36            if start:
37                  traversal += (str(start.value) + "-")
38                  traversal = self.preorder_recursive_print(start.left,
                        traversal)
39                  traversal = self.preorder_recursive_print(start.right,
                        traversal)
40            return traversal
```

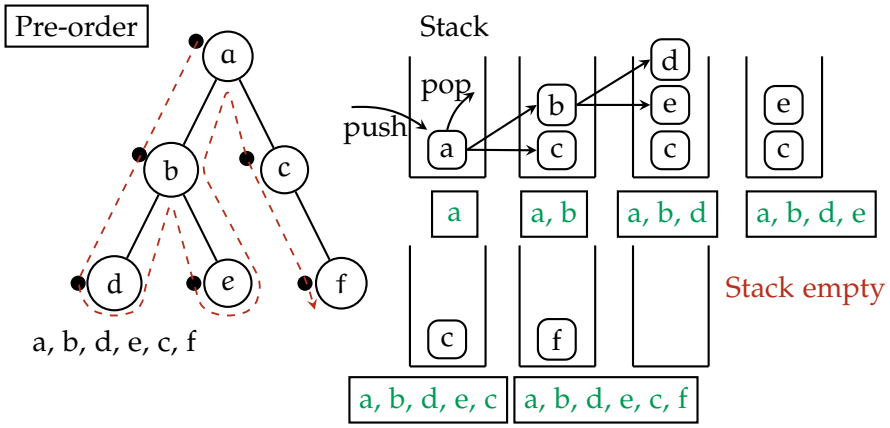


Figure 64: Pre-order iterative example.

A.2 IN-ORDER TRAVERSAL

In the in-order traversal the first node to be checked is the first leaf of the left subtree. The second one is the parent of the first checked node, and then the right subtree is checked repeating the previous process. All this process is repeated until all the nodes are visited or the value is found (Section 4.2.1).

For implementing the in-order traversal iteratively a stack is used. The steps are [47]:

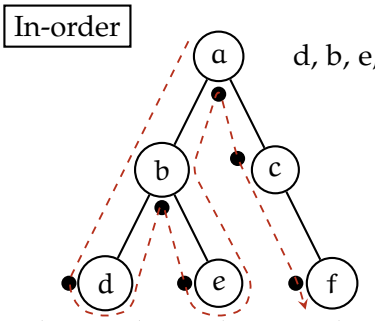
Algorithm 2: In-order pseudocode.

```
1  Function Recursive-Inorder(node):
2      if (node == null) then
3          return
4      Recursive-Inorder(node.left)
5      visit(node)
6      Recursive-Inorder(node.right)
7
8  Function Iterative-Inorder(node):
9      s ← empty stack
10     while (not s.isEmpty() or node ≠ null) do
11         if (node ≠ null) then
12             s.push(node)
13             node ← node.left
14         else
15             node ← s.pop()
16             visit(node)
17             node ← node.right
```

Listing A.3: Recursive and iterative implementation of in-order traversal.

```
class BinaryTree():
      ...

      def recursive_print_tree(self):
            return self.inorder_recursive_print(tree.root, "")[:-1]

      def iterative_print_tree(self):
            return self.inorder_search_iterative(self, tree.root, "")

      def inorder_search_recursive(self, start, find_val):
            if start:
                  self.inorder_search_recursive(start.left, find_val)
                  if start.value == find_val:
                        return True
                  self.inorder_search_recursive(start.right, find_val)

      def inorder_search_iterative(self, start, find_val):
            if start == None:
                  return None
            stack = []
            stack.append(start)
            current = start
            visited = []
            while stack or current != None:
                  if current != None:
                        stack.append(current)
                        current = current.left
```

```
28                else:
29                      current = stack.pop()
30                      if current.value == find_val:
31                            return True
32                      visited.append(current)
33                      current = current.right
34          return visited
35
36    def inorder_recursive_print(self, start, traversal):
37          if start:
38                traversal = self.inorder_recursive_print(start.left,
                      traversal)
39                traversal += (str(start.value) + "-")
40                traversal = self.inorder_recursive_print(start.right,
                      traversal)
41          return traversal
```



Step	Node	push	pop	Stack	New Node	Visit
1	a	a		a	a.left(b)	
2	b	b		b,a	b.left(d)	
3	d	d		d,b,a	d.left(null)	
4	null		d	b,a	d.right(null)	d
5	null		b	a	b.right(e)	b
6	e	e		e,a	e.left(null)	
7	null		e	a	e.right(null)	e
8	null		a		a.right(c)	a
9	c	c		c	c.left(null)	
10	null		c		c.right(f)	c
11	f	f		f	f.left(null)	
12	null		f		f.right(null)	f
stop	null			empty		

Figure 65: In-order iterative example.

A.3 POST-ORDER TRAVERSAL

In the post-order traversal the first node to be checked is the first leaf of the left subtree. The second node is the right node of the parent of the first checked node. Once checked the right side of the subtree also the parent is checked, and the process is repeated until all the node are visited or the value is found (Section 4.2.1).

Also in this case for implementing the post-order traversal iteratively a stack is used. The steps are [47]:

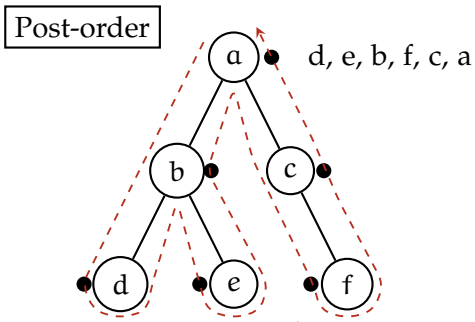
Algorithm 3: Post-order pseudocode.

```
1  Function Recursive-Postorder(node):
2      if (node == null) then
3          return
4      Recursive-Postorder(node.left)
5      Recursive-Postorder(node.right)
6      visit(node)
7
8  Function Iterative-Postorder(node):
9      s ← empty stack
10     lastNodeVisited ← null
11     while (not s.isEmpty() or node ≠ null) do
12         if (node ≠ null) then
13             s.push(node)
14             node ← node.left
15         else
16             peekNode ← s.peek()
17             /* If right child exists and traversing node from
                left child, then move right */
18             if (peekNode.right ≠ null and lastNodeVisited ≠
                peekNode.right) then
19                 node ← peekNode.right
20             else
21                 visit(peekNode)
22                 lastVisitedNode ← s.pop()
```

Listing A.4: Recursive and iterative implementation of post-order traversal.

```
class BinaryTree():
      ...

      def recursive_print_tree(self):
            return self.postorder_recursive_print(tree.root, "")[:-1]

      def iterative_print_tree(self):
            return self.postorder_search_iterative(tree.root, "")

      def postorder_search_recursive(self, start, find_val):
            if start:
                  self.postorder_search_recursive(start.left, find_val)
                  self.postorder_search_recursive(start.right, find_val)
                  if start.value == find_val:
                        return True

      def postorder_search_iterative(self, start, find_val):
            if start == None:
                  return None
```

```
20            stack = []
21            stack.push(start)
22            last_visited_node = None
23            visited = []
24            current = start
25            while stack or current != None:
26                  if current != None:
27                        s.push(current)
28                        current = current.left
29                  else:
30                        # Take the last element of the stack
31                        peeked_node = stack[-1]
32                        # If right child exists and traversing node from
                              left child, then move right
33                        if (peeked_node.right != null and
                              last_visited_node != peeked_node.right):
34                              current = peeked_node.right
35                        else:
36                              if peeked_node.value == find_val:
37                                    return True
38                              visited.append(peeked_node)
39                              last_visited_node = stack.pop()
40            return visited
41
42      def postorder_recursive_print(self, start, traversal):
43            if start:
44                  traversal = self.postorder_recursive_print(start.left,
                        traversal)
45                  traversal = self.postorder_recursive_print(start.right
                        ,traversal)
46                  traversal += (str(start.value) + "-")
47            return traversal
```



Step	Node	New Node	Last Visited	peek	push	pop	Stack	Visit
1	a	a.left (b)			a		a	
2	b	b.left (d)			b		b, a	
3	d	d.left (null)			d		d, b, a	
4	null	d.right (null)	d	d		d	b, a	d
5	null	b.right (e)	d	b			b, a	
6	e	e.left (null)	d		e		e, b, a	
7	null	e.right (null)	e	e		e	b, a	e
8	null	b.right (e) == last visited	e	b		b	a	b
9	null	a.right (c) $\neq$ last visited	e	a			a	
10	c	c.left (null)	e		c		c,a	
11	null	c.right (f) $\neq$ last visited	e	c			c,a	
12	f	f.left (null)	e		f		f,c,a	
13	null	f.right (null)	f	f		f	c,a	f
14	null	c.right (f) == last visited	f	c		c	a	c
15	null	a.right (c) == last visited	a	a		a		a
16	null	null	null	null	null		empty	
stop								

Figure 66: Post-order search.

A.4 BREADTH-FIRST SEARCH IMPLEMENTATION

In the case of the **breadth-first search** (Chapter 4) the implementation uses a queue (Section 2.6) instead of a stack. Also here the convention is to check

the left node first, and later the right ones. The pesudocode is adapted from the general case of the breadth-first search taken from [48].

Algorithm 4: Breadth-first search pseudocode.

```
1  Function Breadth-first-search(node):
2      q ← empty queue
3      visit(node)
4      q.enqueue(node)
5      while (not q.isEmpty()) do
6          node ← q.dequeue()
7          visit(node)
8          /* Left child is enqueue first so that left is
               processed first (FIFO) */
9          if node.left ≠ null then
10             q.enqueue(node.left)
11         if node.right ≠ null then
12             q.enqueue(node.right)
```

Listing A.5: Breadth-first search implementation.

```
1  class BinaryTree():
2        ...
3
4        def breadth_first_search(self, start, find_val):
5              if start == None:
6                    return None
7              visited = []
8              queue = []
9              queue.append(start)
10             while queue:
11                   node = queue.pop(0)
12                   visited.append(node)
13                   if node.value == find_val:
14                         return True
15                   # Left node is added first so that the left side nodes
                           are visited first
16                   if node.left:
17                         queue.append(node.left)
18                   if node.right:
19                         queue.append(node.right)
20             return visited
```

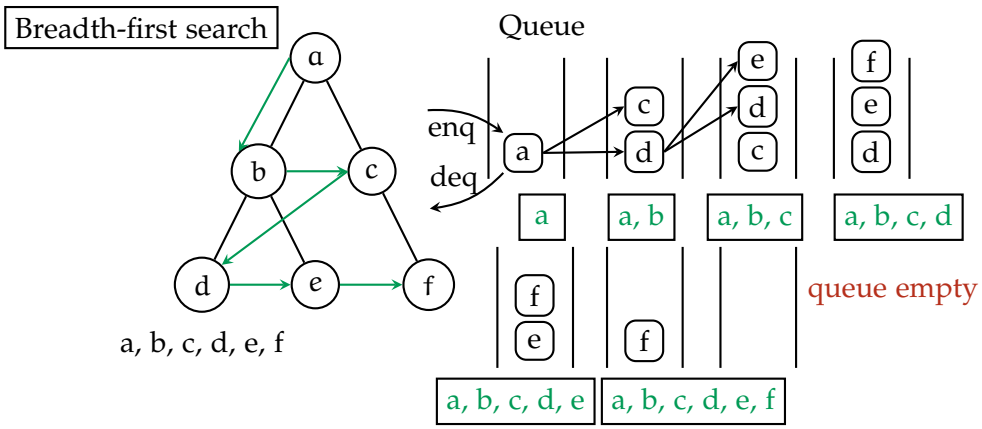


Figure 67: Breadth-first search example.

B

IMPLEMENTATION OF GRAPH REPRESENTATION

In the following python code there is the implementation for the all the different graph representations, and operation of insertion of nodes and edges.

Listing B.1: Graph representation and fundamental operations.

```
class Node():

      def __init__(self, value):
            self.value = value
            self.edges = []

class Edge():

      def __init__(self, value, node_from, node_to):
            self.value = value
            self.node_from = node_from
            self.node_to = node_to

class Graph():

      def __init__(self, nodes=[], edges=[]):
            self.nodes = nodes
            self.edges = edges

      def insert_node(self, new_node_value):
            new_node = Node(new_node_value)
            self.nodes.append(new_node)

      def insert_edge(self, new_edge_val, node_from, val, node_to_val):
            from_found = None
            for node in self.nodes:
                  if node_from_val == node.value:
                        from_found = node
                  if node_to_val == node.value:
                        to_found = node
                  if from_found == None:
                        from_found = node(node_from_val=
                        self.nodes.append(from_found)
                  if to_found == None:
                        to_found = Node(node_to_val)
                        self.nodes.append(to_found)
            new_edge = Edge(new_edge_val, from_found, to_found)
            from_found.edges.append(new_edge)
            to_found.edges.append(new_edge)
            self.edges.append(new_edge)

      def get_edge_list(self):
            get_list = []
```

```
44            for edge_object in self.edges:
45                  edge = (edge_object.value, edge_object.node_from_value.
                        value, edge_object.node_to.value)
46                  edge_list.append(edge)
47            return edge_list
48
49      def get_adjacency_list(self):
50            max_index = self.find_max_index()
51            dajacency_list = [None]*(max_index + 1)
52            for edge_object in self.edges:
53                  if adjacency_list[edge_object.node_from.value]:
54                        adjacency_list[edge_object.node_from.value].
                              append((edge_object.node_to.value,
                              edge_object.value))
55                  else:
56                        adjacency_list[edge_object.node_from.value] = [(
                              edge_object.node_to.value, edge_object.value
                              )]
57            return adjacency_list
58
59      def get_adjacency_matrix(self):
60            max_index = self.find_max_inde()
61            adjacency_matrix = [[0 for i in range(max_index + 1)] for j
                  in range(max_index + 1)]
62            for edge_object in self.edges:
63                  adjacency_matrix[edge_object.node_from.value][
                        edge_object.node_to.value] = edge_object.value
64            return adjacency_matrix
65
66      def find_max_index(self):
67            max_index = 1
68            if len(self.node):
69                  for node in self.nodes:
70                        if node.value > max_index:
71                              max_index = node.value
72            return max_index
```

C IMPLEMENTATION OF GRAPH TRAVERSAL

In the following chapter **depth-first search** and **breadth-first search** are implemented, with both recursive and iterative implementation.

C.1 DEPTH-FIRST SEARCH

In the following there are the pesudocode, the python implementation, and an example of execution of the depth-first search on a graph [34].

Algorithm 5: Depth-first search pseudocode.

```
1  Function Recursive-Depth-first-search(graph, vertex):
2      visit(vertex)
3      for all adjacent nodes adj_vertices to vertex do
4          Recursive-Depth-first-search(graph, adj_vertices)
5
6  Function Iterative-Depth-first-search(graph, vertex):
7      s ← empty stack
8      s.push(vertex)
9      while not s.isEmpty() do
10         vertex ← s.pop()
11         if vertex not visited then
12             visit(vertex)
13             for all adjacent nodes adj_vertices to vertex do
14                 s.push(adj_vertices)
```

Listing C.1: Recursive and iterative implementation of depth-first search on graphs.

```
1  class Graph():
2        ...
3
4        def depth_first_search_recursive(self, start_node):
5              ret_list = [start_node.value]
6              start_node.visited = True
7              edges_out = [e for e in start_node.edges
8                                  if e.node_to.value != start_node.value]
9              for e in edges_out:
10                   if not edge.node_to.visited:
11                         ret_list.extend(depth_first_search_recursive(
                               edge.node_to)
12             return ret_list
13
14       def depth_first_saerch_iterative(self, start_node):
15             stack = []
16             stack.append(start_node)
17             while stack:
```

```
                    s = stack[-1]
                    stack.pop()
                    if not s.visited:
                          s.visit = True
                    edges_out = [e for e s.edges if e.node_to.value != s.
                        value]
                    for neighbour in edges_out:
                          if not neighbour.visited:
                                stack.append(neighbour)
```

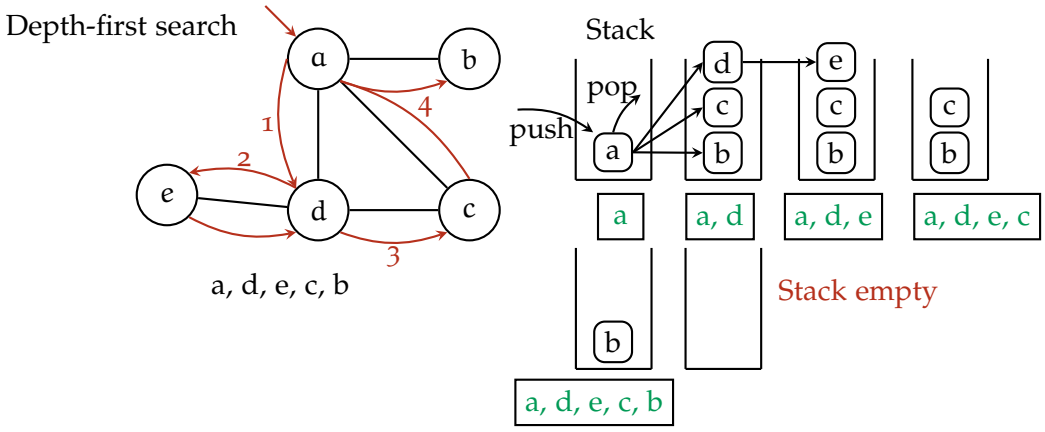


Figure 68: Depth-first search example.

C.2 BREADTH-FIRST SEARCH

In the following there are the pesudocode, the python implementation, and an example of execution of the breadth-first search on a graph [48].

Algorithm 6: Breadth-first search pseudocode.

```
1  Function Breadth-first-search(graph, vertex):
2      q ← empty queue
3      visit(vertex)
4      q.enqueue(vertex)
5      while not q.isEmpty() do
6          vertex ← s.dequeue()
7          for all adjacent nodes adj_vertices to vertex do
8              if adj_not visited then
9                  visit(adj_vertices)
10                 q.enqueue(adj_vertices)
```

Listing C.2: Breadth-first search implementation on graphs.

```
class Graph():
      ...

      def breadth_first_search(self, start_node):
            queue = []
            queue.append(start_node)
            while queue:
```

```
8             q = queue.pop(0)
9             edges_out = [e for e q.edges if e.node_to.value != q.value]
10            for neighbour in edges_out:
11                  if neighbour not neighbour.visited:
12                        neighbour.visited = True
13                        queue.append(neighbour)
```

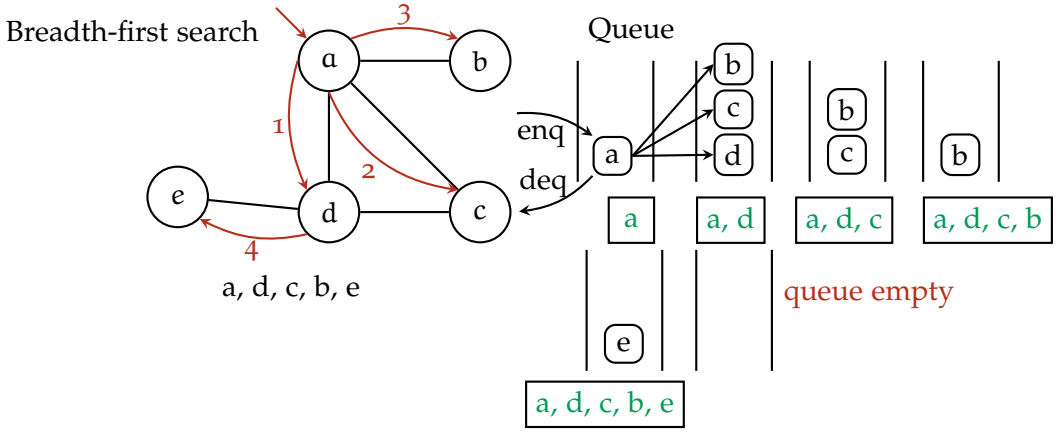


Figure 69: Breadth-first search example.

D

DIJKSTRA'S ALGORITHM IMPLEMENTATION

As described in Section 5.3.6, the Dijkstra's algorithm is a greedy algorithm that find the optimal solution by finding the optimal solution at each step. The implementation of the Dijkstra's algorithm is based on priority queue.

Let us consider the following graph (Figure 70).

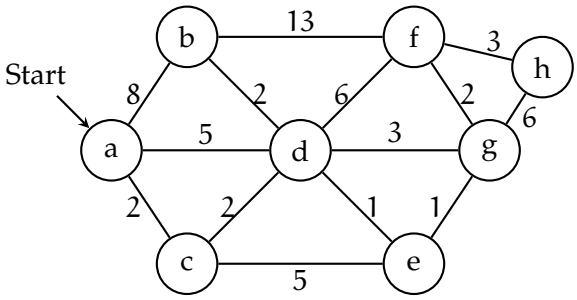


Figure 70: Dijkstra's algorithms example.

Solving the shortest problem for this graph starting from the node a leads to the following table:

visited node	a	b	c	d	e	f	g	h
a	0_a	8_a	2_a	5_a	∞	∞	∞	∞
c	|	8_a	2_a	4_c	7_c	∞	∞	∞
d	|	6_d	|	4_c	5_d	10_d	7_d	∞
e	|	6_d	|	|	5_d	10_d	6_e	∞
b	|	6_d	|	|	|	10_d	6_e	∞
g	|	|	|	|	|	8_g	6_e	12_g
f	|	|	|	|	|	8_g	|	11_f
h	|	|	|	|	|	|	|	11_f

Table 6: Dijkstra's algorithms example for the graph in Figure 70. The shortest path from a to b is also shown.

All the shortest paths from a are:

- b: a → c → d → b, W=6;
- c: a → c, W=2;
- d: a → c → d, W=4;
- e: a → c → d → e, W=5;
- f: a → c → d → e → g → f, W=8 ;

- g: a → c → d → e → g, W=6;
- h: a → c → d → e → g → f → h, W=11.

For implementing this algorithm there are several ways. The one presented here uses the priority queues. In fact, each line of the table where are reported the updated distances with the shortest path can be see as a priority queue in which at every step the shortest value (then the shortest path for the current visited node) is selected.

In the following the pseudocode [39] and the Python implementation [49] are reported.

Algorithm 7: Dijkstra's algorithm pseudocode.

```
1  Function Dijkstra(graph, source):
2      dist[source] ← 0 // Initialization
3      create vertex priority queue q
4      for all vertex v in graph do
5          if v ≠ source then
6              dist[v] ← infinity // Unknown distance from source to v
7              prev[v] ← undefined // Predecessor of v
8          q.add_with_priority(v, dist[v])
9      while not q.isEmpty() do
10         u ← q.extract_min() // Remove and return best vertex
11         for all neighbor v of u do
12             // Only v that are still in q
13             alt ← dist[u] + length(u, v)
14             if alt < dist[v] then
15                 dist[v] ← alt
16                 prev[v] ← u
17                 q.decrease_priority(v, alt)
18     return dist, prev
```

Listing D.1: Dijkstra's algorithm implementation. Credits: stackoverflow.

```python
import heapq

def dijkstra(graph, start):
    """Visit all nodes and calculate the shortest paths to each from
        start"""
    queue = [(0, start)]
    distances = {start: 0}
    visited = set()
    while queue:
        _, node = heapq.heappop(queue) # (distance, node), ignore distance
        if node in visited:
            continue
        visited.add(node)
        dist = distances[node]
        for neighbour, neighbour_dist in graph[node].items():
            if neighbour in visited:
                continue
```

```
17          neighbour_dist += dist
18          if neighbour_dist < distances.get(neighbour, float('inf')):
19              heapq.heappush(queue, (neighbour_dist, neighbour))
20              distances[neighbour] = neighbour_dist
21    return distances
```

E

COMPLEXITY SUMMARY

In the following table are reported the complexities of the most important data structure and algorithms cited in this notes [50].

BIG-O COMPLEXITIES

OF COMMON ALGORITHMS USED IN COMPUTER SCIENCE

DATA STRUCTURE OPERATIONS

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	–	O(1)	O(1)	O(1)	–	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	–	O(log(n))	O(log(n))	O(log(n))	–	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	–	O(log(n))	O(log(n))	O(log(n))	–	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

ARRAY SORTING ALGORITHMS

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

LEGEND

Excellent
Good
Fair
Bad
Horrible

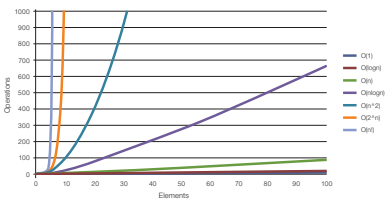
GRAPH OPERATIONS

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	O(|V|+|E|)	O(1)	O(1)	O(|V|+|E|)	O(|E|)	O(|V|)
Incidence list	O(|V|+|E|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Adjacency matrix	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Incidence matrix	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|V| · |E|)	O(|E|)

HEAP OPERATIONS

Heap Type	Time Complexity						
	Heapify	Find Max	Extract Max	Increase Key	Insert	Delete	Merge
Linked List (sorted)	–	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Linked List (unsorted)	–	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Binary Heap	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Binomial Heap	–	O(1)	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))
Fibonacci Heap	–	O(1)	O(log(n))	O(1)	O(1)	O(log(n))	O(1)

BIG-O COMPLEXITY CHART



WWW.BIGOPOSTER.COM

Figure 71: Complexities of the most important algorithms. Credits: Big-O Poster, GitHub.

BIBLIOGRAPHY

[1] Wikipedia. *Computational complexity*. URL: https://en.wikipedia.org/wiki/Computational_complexity.

[2] Wikipedia. *Big O notation*. URL: https://en.wikipedia.org/wiki/Big_O_notation.

[3] Wikipedia. *Recursion (computer science)*. URL: https://en.wikipedia.org/wiki/Recursion_(computer_science).

[4] Wikipedia. *Data structure*. URL: https://en.wikipedia.org/wiki/Data_structure.

[5] Wikipedia. *Data type*. URL: https://en.wikipedia.org/wiki/Data_type.

[6] Wikipedia. *Abstract data type*. URL: https://en.wikipedia.org/wiki/Abstract_data_type.

[7] Wikipedia. *Collection (abstract data type)*. URL: https://en.wikipedia.org/wiki/Collection_(abstract_data_type).

[8] Wikipedia. *List (abstract data type)*. URL: https://en.wikipedia.org/wiki/List_(abstract_data_type).

[9] Wikipedia. *Array data structure*. URL: https://en.wikipedia.org/wiki/Array_data_structure.

[10] Wikipedia. *Linked list*. URL: https://en.wikipedia.org/wiki/Linked_list.

[11] Wikipedia. *Doubly linked list*. URL: https://en.wikipedia.org/wiki/Doubly_linked_list.

[12] Wikipedia. *Stack (abstract data type)*. URL: https://en.wikipedia.org/wiki/Stack_(abstract_data_type).

[13] Wikipedia. *Queue (abstract data type)*. URL: https://en.wikipedia.org/wiki/Queue_(abstract_data_type).

[14] Wikipedia. *Set*. URL: https://en.wikipedia.org/wiki/Set_(abstract_data_type).

[15] Wikipedia. *Hash map*. URL: https://en.wikipedia.org/wiki/Associative_array.

[16] Wikipedia. *Hash talbe*. URL: https://en.wikipedia.org/wiki/Hash_table.

[17] Wikipedia. *Hash function*. URL: https://en.wikipedia.org/wiki/Hash_function.

[18] Wikipedia. *Search algorithm*. URL: https://en.wikipedia.org/wiki/Search_algorithm.

[19] Wikipedia. *Sorting algorithm*. URL: https://en.wikipedia.org/wiki/Sorting_algorithm.

[20] Wikipedia. *Binary search algorithm*. URL: https://en.wikipedia.org/wiki/Binary_search_algorithm.

[21] Wikipedia. *Bubble sort*. URL: https://en.wikipedia.org/wiki/Bubble_sort.

[22] Wikipedia. *Merge sort*. URL: https://en.wikipedia.org/wiki/Merge_sort.

[23] Wikipedia. *Quicksort*. URL: https://en.wikipedia.org/wiki/Quicksort.

[24] Wikipedia. *Heapsort*. URL: https://en.wikipedia.org/wiki/Heapsort.

[25] GeeksforGeeks. *Quicksort python implementation*. URL: https://www.geeksforgeeks.org/python-program-for-quicksort/.

[26] Wikipedia. *Trees*. URL: https://en.wikipedia.org/wiki/Tree_(data_structure).

[27] Wikipedia. *Binary search tree*. URL: https://en.wikipedia.org/wiki/Binary_search_tree.

[28] M.T. Goodrich, R. Tamassia, and M.H. Goldwasser. *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated, 2013. ISBN: 9781118476734. URL: https://books.google.it/books?id=2UccAAAAQBAJ.

[29] Wikipedia. *Heap*. URL: https://en.wikipedia.org/wiki/Heap_(data_structure).

[30] Wikipedia. *Self-balancing binary search tree*. URL: https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree.

[31] Wikipedia. *Red-Black tree*. URL: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree.

[32] Wikipedia. *Graph*. URL: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).

[33] Wikipedia. *Connectivity*. URL: https://en.wikipedia.org/wiki/Connectivity_(graph_theory).

[34] Wikipedia. *Depth-first search*. URL: https://en.wikipedia.org/wiki/Depth-first_search.

[35] Wikipedia. *Eulerian path*. URL: https://en.wikipedia.org/wiki/Eulerian_path.

[36] William Fiset. *Existence of Eulerian Paths and Circuits | Graph Theory*. URL: https://www.youtube.com/watch?v=xR4sGgwtR2I.

[37] Wikipedia. *Hamiltonian path*. URL: https://en.wikipedia.org/wiki/Hamiltonian_path.

[38] Wikipedia. *Shortest path problem*. URL: https://en.wikipedia.org/wiki/Shortest_path_problem.

[39] Wikipedia. *Dijkstra's algorithm*. URL: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[40] Abdul Bari. *3.6 Dijkstra Algorithm - Single Source Shortest Path - Greedy Method*. URL: https://www.youtube.com/watch?v=XB4MIexjvY0.

[41] Wikipedia. *Dynamic programming*. URL: https://en.wikipedia.org/wiki/Dynamic_programming.

[42] Wikipedia. *Knapsack problem*. URL: https://en.wikipedia.org/wiki/Knapsack_problem.

[43] Wikipedia. *Travelling salesman problem*. URL: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[44] Wikipedia. *HP-hardness*. URL: https://en.wikipedia.org/wiki/NP-hardness.

[45] Wikipedia. *Pseudo-plynomial time*. URL: https://en.wikipedia.org/wiki/Pseudo-polynomial_time.

[46] Wikipedia. *Christofides algorithm*. URL: https://en.wikipedia.org/wiki/Christofides_algorithm.

[47] Wikipedia. *Trees traversal*. URL: https://en.wikipedia.org/wiki/Tree_traversal.

[48] Wikipedia. *Breadth-first search*. URL: https://en.wikipedia.org/wiki/Breadth-first_search.

[49] Martijn Pieters. *StackOverflow answer*. URL: https://stackoverflow.com/a/57234618.

[50] Roman Pushkin. *Big O Poster*. URL: https://github.com/ro31337/bigoposter.