Data Structures and Algorithms in Python

Notes based on the Udacity online course

Omar Chehaimi

October 15, 2020

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
https://github.com/fmarotta/kaobook
(You are welcome to contribute!)

Felix qui potuit rerum cognoscere causas.

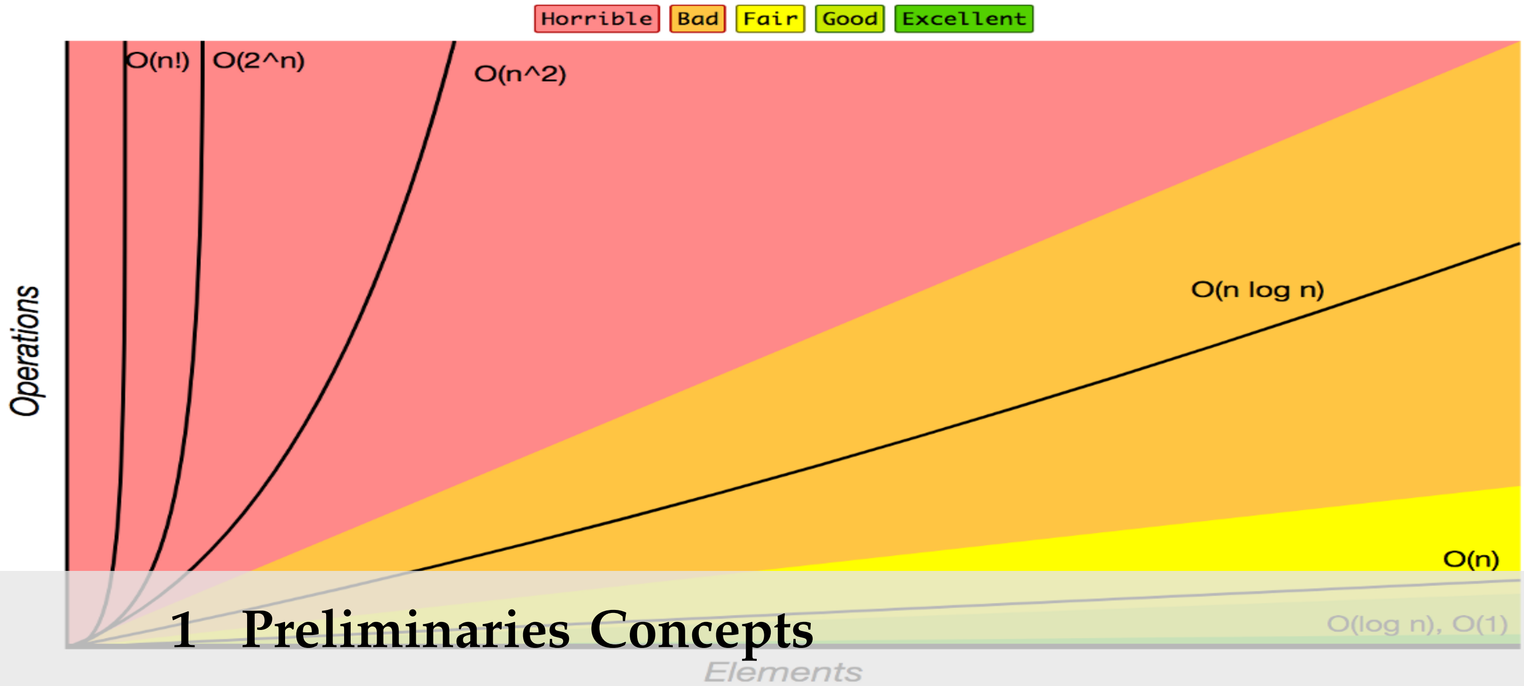
– Publius Vergilius Maro

Preface

The following notes are based on the Udacity online course Intro to Data Structures and Algorithms. These notes are not to intended to be used as book, nor as formal introduction to data structures and algorithms, they are just a my personal attempt to have a small but comprehensive notes about the huge world of data structures and algorithms.

Omar Chehaimi

Contents



1 Preliminaries Concepts

1.1 Computational Complexity

The computational complexity of an algorithm is the amount of resources needed for solving it. The computational complexity is the minimum complexity of all possible implementations for solving a given algorithm, included also the unknown ones.

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

When the kind of the complexity is not explicitly indicated, generally is meant to be the **Time Complexity**, which is different from computer to computer, and is generally expressed as as the number of elementary operations required to solve a given algorithm. It is assumed that these elementary operations take the same time for being solved. The computational complexity can be related also to the memory consumption.

Synonymous for Computational Complexity are: Complexity, and Efficiency.

1.2 Big O Notation

The Big O notation [1] (Big O Notation, Wikipedia) describe the behavior of a function when its argument tents to infinity or a particular value.

Definition 1.2.1 *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 $x \longrightarrow \infty$ if $\exists$ $M \in \mathbb{R}$, $M > 0$ and $x_0 \in \mathbb{R}$ such that $|f(x)| \leq Mg(x)\ \forall x \geq x_0$.*

The credits for the image above the chapter title go to: `https://www.bigocheatsheet.com/`

Usually $f(x) = O(g(x))$ is used as $x \longrightarrow \infty$, but it can be defined also for the case $x \longrightarrow a$, 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 became irrelevant.

Example 1.2.1 In $f(x) = 6x^4 - 2x^3 + 5$ as $x \longrightarrow \infty$, $6x^4$ it the fastest growing term. 6 is a constant and can be omitted, thus $f(x) = O(x^4)$.

Properties

Here are listed some simple properties about Big O notation.

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: $O(k|g) = O(g)$, $f = O(g) \Rightarrow kf = O(g)$

Logarithm and Exponential

Let c be a nonzero constant, then: $O(\log n^c) = O(\log n)$ $(\log n^c = c \log n)$.
$O(n^c)$ 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 1.2.2 $f = 9 \log n + 5(\log n)^4 + 3n^2 + 2n^2 = O(n^3)$ as $n \longrightarrow \infty$

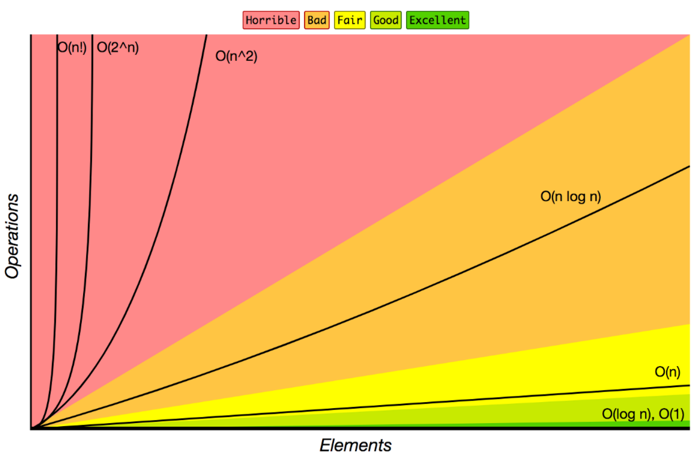


Figure 1.1: Plots of the main functions and their evaluation for computational complexity. Credits: bigocheatsheet.com

1.3 Time complexity evaluation example

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

```python
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 += 1 # -> O(1)
    return total # -> O(1)
```

Listing 1.1: Sum of all elements of a matrix.

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 [2] (Recursion, Wikipedia) a function calls itself again on a smaller size input, until the exit condition stops this self calling (recursive calling). There are three fundamentals elements in a recursive function:

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

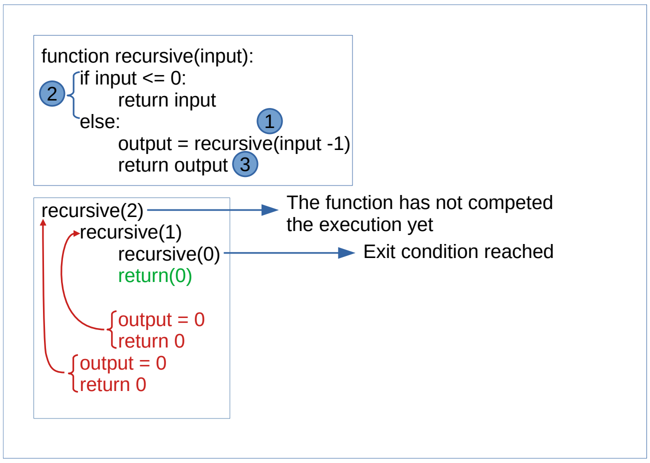


Figure 1.2: Pseudocode of a recursive function and its internal execution.

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

Python Implementation Examples

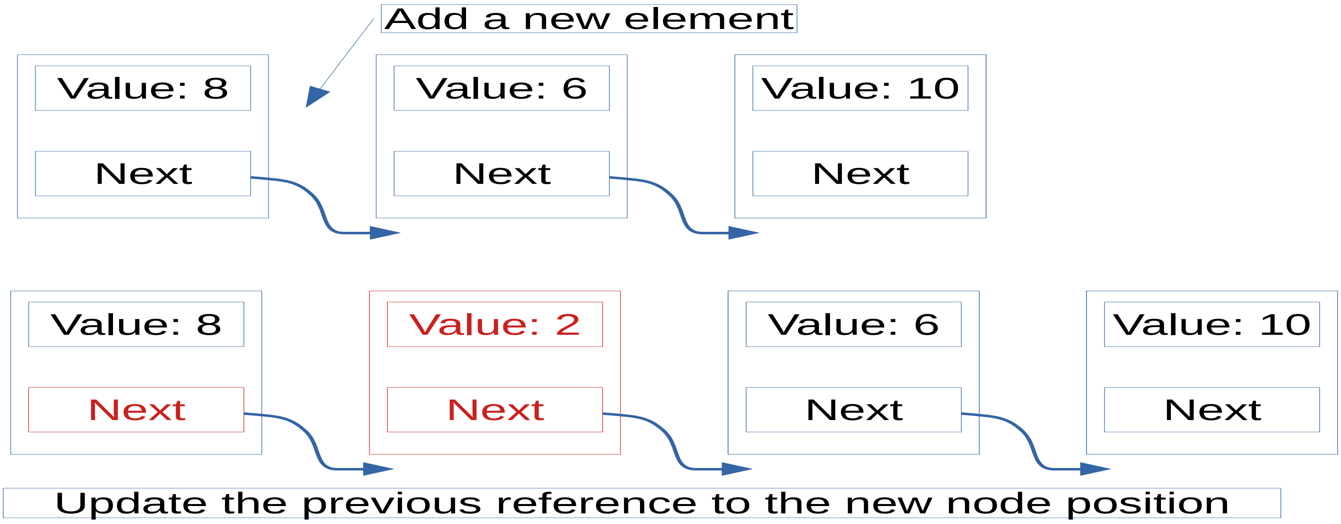
```python
def fibonacci_iterative(position):
    if position == 0:
        return 0
    elif position == 1:
        return 1
    else:
        first = 0
        second = 1
        next = first + second
        for i in range(2, position):
            first = second
            second = next
        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

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

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

Listing 1.3: Implementation of calculating the factorial of a number using the recursive way.



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 collected data, their properties, the operations that can be done on them, are all properties of the data structure [3] (Data Structure, Wikipedia). Data structure is the basis for **Abstract Data Type (ADT)** a mathematical model for **Data Type** [4] (Data Type, Wikipedia), which is defined by its behavior from the poin of view of a user, its type of data, specifically in terms of possible values, by possible operations on these data, and by the behavior of these operations. This mathematical model contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user [5] (Abstract Data Type, Wikipedia.
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 only single unit. Collections are used to save, to find, to manipulate, and to communicate grouped data. [6] (Collection (abstract data type), Wikipedia) 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 → phone number).

2.2 List

A **List** is a collection which represents a set of **ordered** elements, which can also be of different type. Same value elements can be repeated several times. Lists don't 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 ($O(1)$).

2.3 Array

An **Array** is a collection of elements, of the same or also different type, in which each of them is identified with at least one **array index** or **key** [7] (Array data structure, Wikipedia). In some programming languages arrays have a fixed size, not in others.

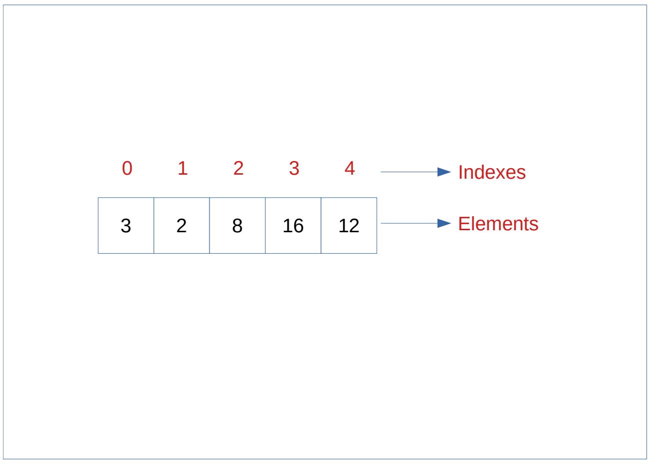


Figure 2.1: An example of an array with elements and indexes.

For doing any allowed operation to an element of an array it is enough to know its index.
Adding or removing an element in an array could be a very expensive operation. This is because when a change take place to an element, all the following indexes to the modified element must be updated (Figure 2.2). The worst case complexity is $O(n)$, where n is the size of the array.

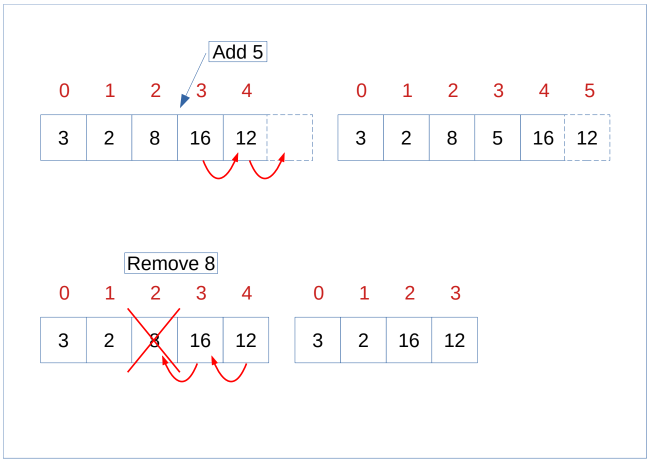


Figure 2.2: Removing or adding an element from an array and the indexes update.

2.4 Linked List

A **Linked List** is a linear collection in which each element has the data and a reference to the next element (a link) [8] (Linked List, Wikipedia). In a linked list the order of the elements is not assured.

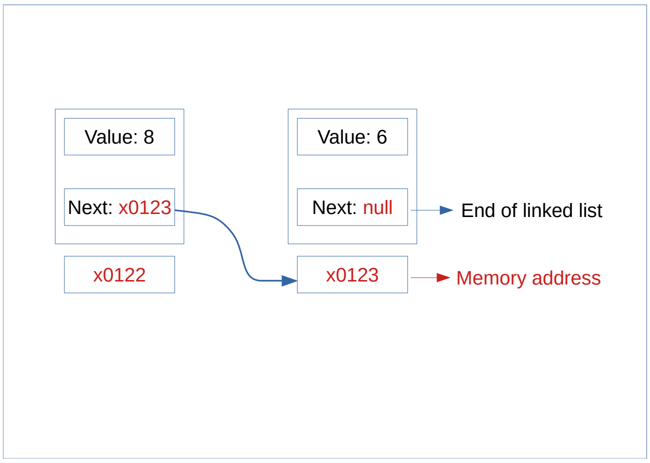


Figure 2.3: An example of a linked list with the data and the reference to the next element.

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 for adding a new element is enough to change the previous element reference to the just added element, and change the reference of the new element to the next element (Figure 2.4). In case of removing an element it is enough to update the previous element's reference to the next element of the removed one (Figure 2.5).

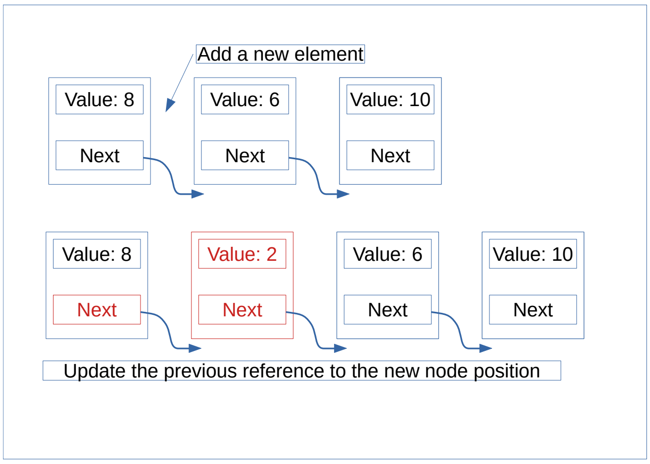


Figure 2.4: Adding a new element to the linked list. As explained before this operation is very efficient because it is enough to change the references of the new element and the previous element to the one just added.

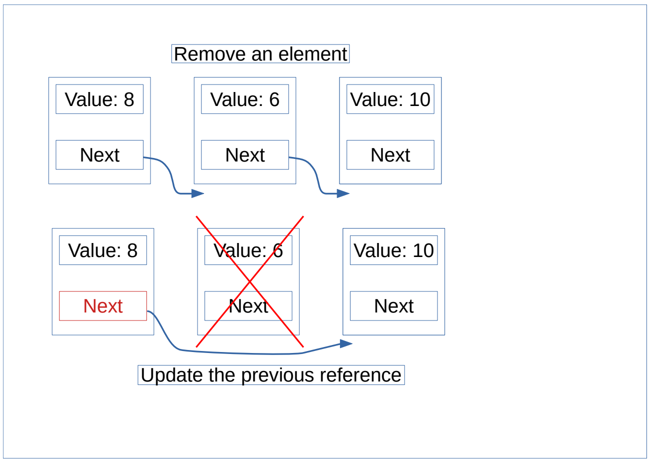


Figure 2.5: Removing an element to the linked list.

The complexity for adding or removing an element is constant $O(1)$. Linked lists can be also **Doubly Linked List** (Figure 2.6), which evry element has the reference to the previous and next element of the collection [9] (Doubly linked list, Wikipedia).

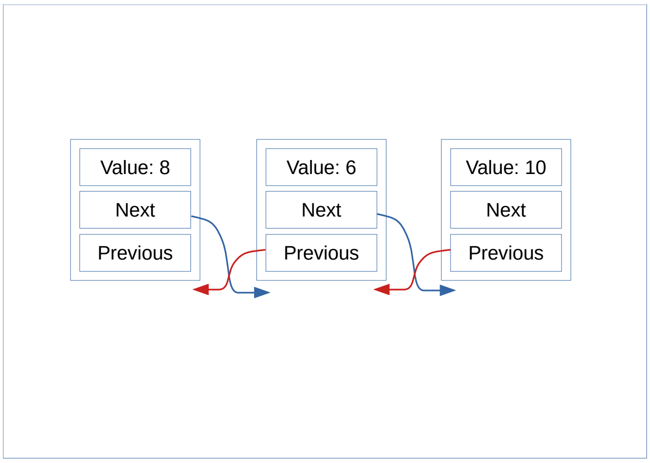


Figure 2.6: Doubly Linked List.

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

Linked List Implementation

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

Listing 2.1: Linked List implementation.

```
class Element(object):

    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList(Object):

    def __init__(self, head=None):
        self.head = head

    def append(self, new_element):
        current = self.head
        if self.head:
            while current.next:
                current = current.next
            current.next = new_element
        else:
            self.head = new_element

    def get_position(self, position):
        counter = 0
        current = self.head
        if position < 1:
            return None
        while current and counter <= position:
            if counter == position:
                return current
            current = current.next
            counter += 1
        return None

    def insert(self, new_element, position):
        counter = 1
        current = self.head
        if position > 1:
            while current and counter < position:
                if counter == position - 1:
                    new_element.next = current.next
                    current.next = new_element
                current = current.next
                counter +=1
        elif position == 1:
            new_element.next = self.head
            self.head = new_element
```

2.5 Stack

A **Stack** is abstract data type belonging to collections, in which only two operations are allowed [10] (Stack, Wikipedia):

- **Push**: add an element at the top of the stack
- **Pop**: remove the newest element of the stack

In the stacks only the element at the top of the stack can be modified. The complexity remains constant for adding and removing operations. The stacks are also called **Last In, First Out** (**LIFO**).

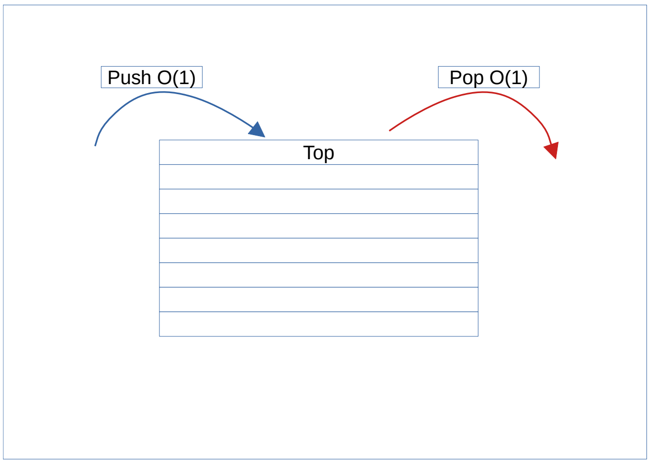


Figure 2.7: In a stack only the element at the top is modified.

Stack Implementation

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

Listing 2.2: Stack implementation.

```
class Element(object):
    ...

class LinkedList(object):
    ...

    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(Object):

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

```

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

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

2.6 Queue

A **Queue** is abstract data type belonging to collections very similar to the stacks. For a queue the admitted operations are only on the oldest element, thus the first element to be added [11] (Queue, Wikipedia). The allowed operations on a queue are:

- **Enque**: adding an element at the bottom of the queue
- **Deque**: removing the element at the head of the queue
- **Pick**: observing the element at the head of the queue

For adding and removing operations the complexity remain constant. Queues are also called **First In, First Out** (**FIFO**).

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

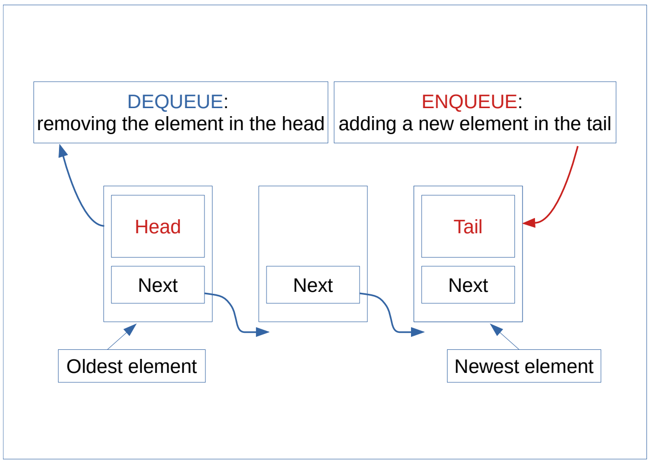


Figure 2.8: In a queue only the first element to be added can be removed or where a new element can be added.

A generalization of queues and linked lists are **Deques**, in which is possible to perform deque and enque operations on both the **Head** and **Tail**.

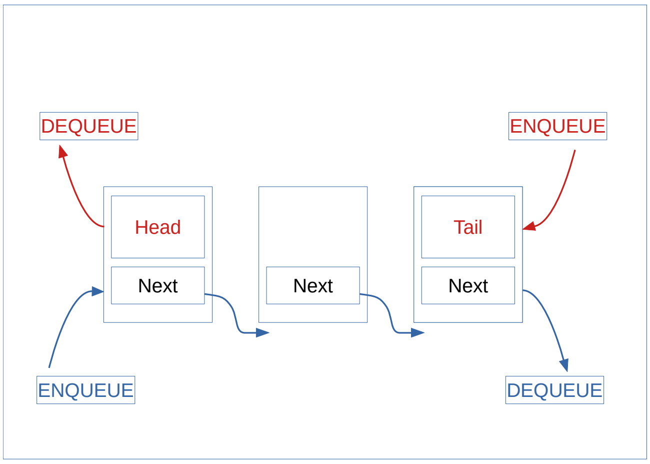


Figure 2.9: In a deque the operations can be done on both head and tail.

Another modification of queue are **Priority Queue**, in which each element has a priority (a numeric value that indicates its importance). When an element of the queue is removed (deque), 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 element is removed.

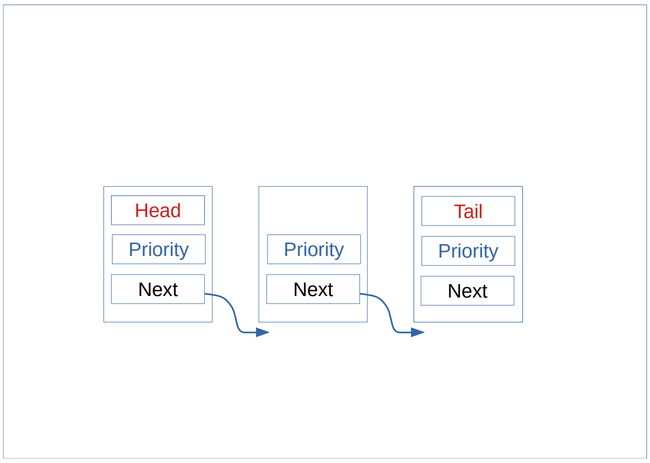


Figure 2.10: A priority queue.

Queue Implementation

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

Listing 2.3: Queue implementation.

```
class Queue(object):
    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 the unique values are stored without a particular order [12] (Set, Wikipedia).

2.8 Map

A **Map**, **Associative Array**, **Symbol Table**, or **Dictionary** is an abstract data type composed of a collection of (key, value) pairs [13] (Hash Map, Wikipedia). The group of the key is a set, in which each element has a unique value. Map is a very useful data type, used in a lot of situations.

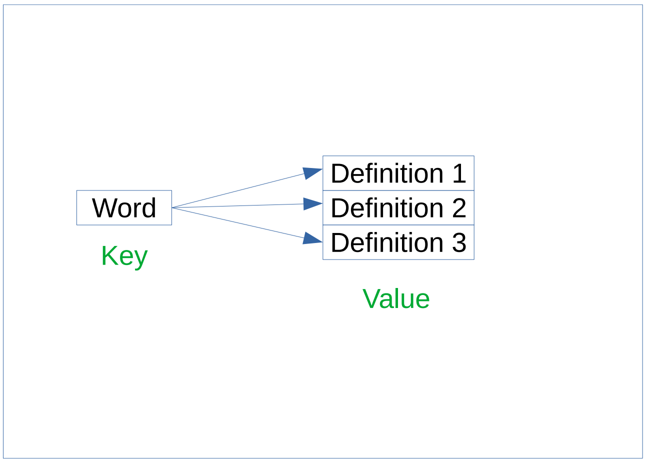


Figure 2.11: An example of a map.

2.9 Hash Table

A **Hash Table** or **Hash Map** is a data structure that implements the map abstract data type [14] (Hash Table, Wikipedia). Hash table uses the **Hash Function** for working [15] (Hash Function, Wikipedia). Using the hash function is called hashing.

Hashing

Hashing is an operation that allows to transform the value of a variable to another one which is much easier to find within a collection. If we want to find a value inside a collection (list, stack, queue, etc) the time requested for the search is linear with the size of the collection. In fact we have to check all the elements until the searched one is found (in case of stack and queue this is not true is only the last or the first element respectively is checked). For solving this problem and thus to find an element in a constant time hash function are used.

Let us consider the follow example, where we have an array in which we would like to store big random numbers. A simple way for hashing these numbers is to consider only the last numbers (56 and 17) and divide them for a fixed number. The reminder of the division is used as the new index of the array associated to the number (Figure).

What happen when the hash function transforms two different number in the same? In this case we have a **Collision** (Figure 2.12). In case of a collision there are several strategies for solving this issue: one way could be to change the the hash function, another one could be to use an array for storing different values associated to the same key (**Bucket**). For the last occurrence the worst case complexity for a search is $O(n)$ (Figure 2.12).

It does not exist a perfect hash function and a trade off must always be reached. A way to analyze a hash function is by using the **Load Factor** defined as follow: $Load\ Factor = \#of\ entries/\#of\ buckets$. For example, if we want to save 10 values using 1000 buckets the Load Factor is equal to 0.01, and most of the buckets is empty. In this case is convenient to change the hash function by using less buckets. More the Load Factor is closed to 0 more the hash table will be empty (**Sparse**), and more the Load

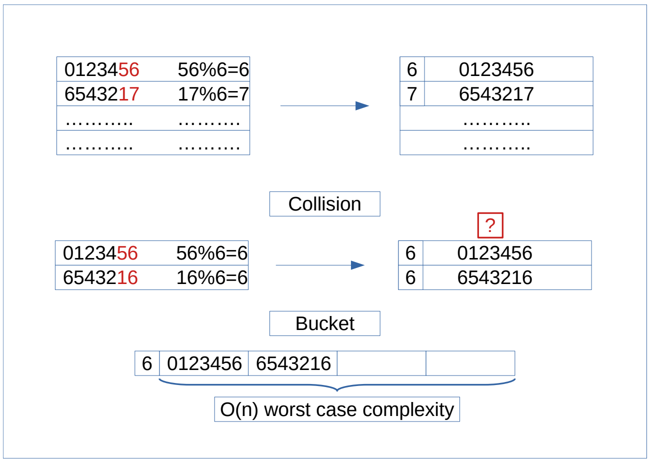


Figure 2.12: An example of collision and a possible way to solve this issue by using the bucket method.

Factor is closed to 1 more the has table will be full and efficient. In case the Load Factor is bigger than 1 there is the certainty there will happen some collisions. Another example: let us consider a hash function that divide the numbers by 100. If we consider 100 values all multiple of 5, the Load Factor will be: $Load\ Factor = \#of\ entries/\#of\ buckets = 100/100 = 1$. But this way is very slow and a different configuration should be used to make this more efficient. For example if we use more bucket, for example 107, 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. For hash function we can use the following: $S[0] * 31^{(n-1)} + S[1] * 31^{(n-2)} + \ldots + S[n-1]$, 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.

String Keys Implementation

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

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

Listing 2.4: String key implementation.

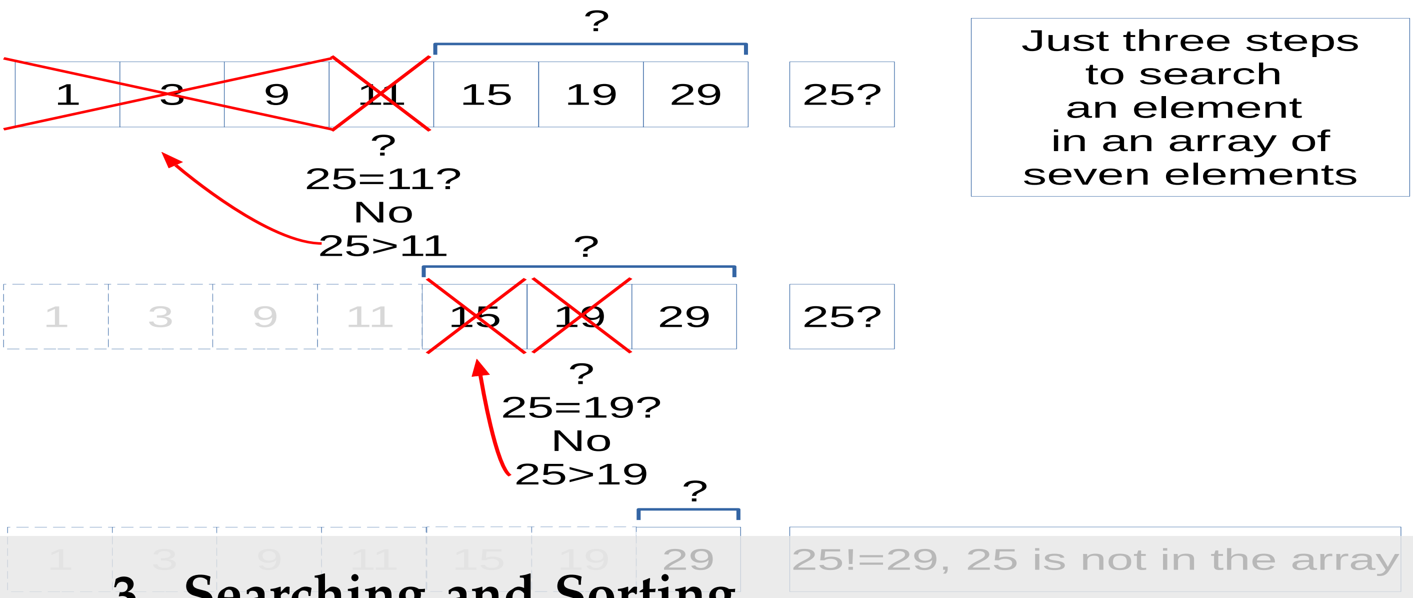
```
class HashTable(object):
    def __int__(self):
        self.table = [None]*10000
```

```

    def store(self, string):
        hv = self.calculate_hash_value(string)
        if hv != -1:
            if self.table[hv] != None:
                self.table[hv].append(string)
            else:
                self.table[hv] = [string]

    def lookup(self, string):
        hv = self.calculate_hash_value(string)
        if hv != -1:
            if self.table[hv] != None:
                if string in self.table[hv]:
                    return hv
        return -1

    def calculate_hash_value(self, string):
        value = ord(string[0]*100) + ord(string[1])
        return value
```



3 Searching and Sorting

In this chapter are introduced the most important and used algorithms about **Searching**, which retrieve some data stored in a particular data structure [16] (Search algorithm, Wikipedia), and **Sorting**, which put in a certain order some data stored in a list [17] (Sorting algorithm, Wikipedia).

3.1 Binary Search

Let us consider the problem of finding a number stored in an array with an **ascending** order (Figure 3.1).

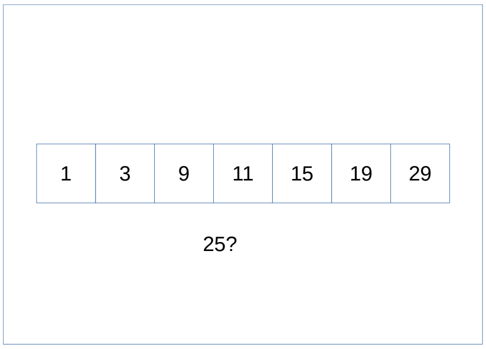


Figure 3.1: An array with numeric values ordered in an ascending order.

A first way to tackle the problem could be to check all the numbers of the array, in others words this method consists in performing a loop all over the elements of the array and check one by one if the number where is the element we are looking for. The complexity of this method is $O(n)$ because in the worst case we have to look at all the elements of the array.

There is a more efficient way to search an element in an ascending ordered array then the method showed previously. This method is called **Binary Search** [18] (Binary Search Algorithm, Wikipedia). Let us start to check as the first element 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). If the central element is the one we are looking for the search ends, otherwise, because the array is ascending ordered, we can ignore one half of the array. If the central number is bigger than the number we are looking for thus we will ignore the right half of the array, if the central number is lower than the number we are looking for thus we will ignore the left half of the array. This procedure is repeated: we consider the central value of the new array, which has half of the size of the previous step, and we check is that value is the one we are looking for, and we repeat the procedure until or we find or we do not find the number we are looking for (Figure 3.2).

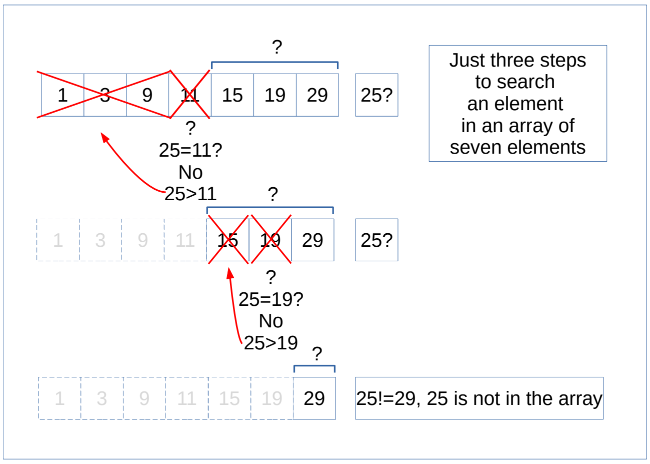


Figure 3.2: Binary Search Algorithms steps.

Efficiency of the Binary Search Algorithm

For evaluating the efficiency of this algorithm we have to evaluate the number of steps required for solving the problem. We already know that the efficiency of this algorithm will not be $O(n)$, because in this algorithm not all the elements of the array are checked. The way for evaluating the complexity of an algorithm is to execute the algorithms 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 searched element at the end of the array splitting process.

		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

Table 3.1: The number of iterations grows of one every power of two, in others words it grows as $log(n)$.

We observe that the exponent of the power of two is the number of iteration minus one, or at the contrary the number of iteration is the exponent of a power plus one.

$$log(power\ of\ exponent\ 2+1) = log_2(n) + 1$$

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

Binary Search Implementation

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

```python
def binary_search(array_input, value):
    low = 0
    high = len(array_input) - 1
    while low <= high:
        mid = (low + high) // 2
        if array_input[mid] == value:
            return mid
        elif value > array_input[mid]:
            new_low = mid + 1
        else:
            new_high = mid - 1
    return -1
```

Listing 3.1: Binary search python implementation.

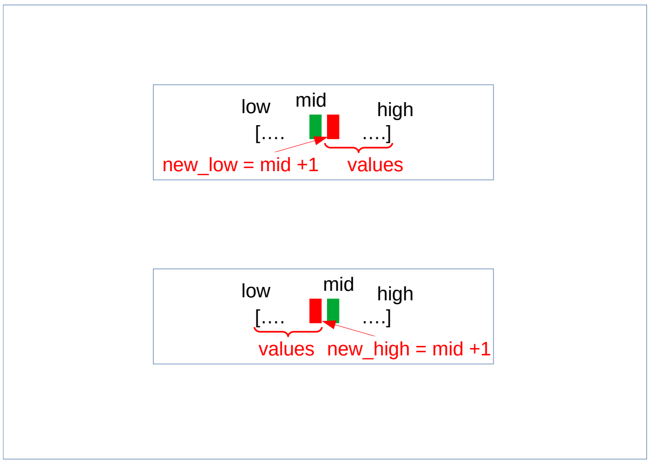


Figure 3.3: Array splitting in the implementation of the binary search algorithm.

3.2 Bubble Sort

Bubble Sort is the easiest sorting algorithm working on arrays. Bubble sort works by swapping two element at each step if they are in the wrong order, repeating this process until all the array is not completely ordered [19] (Bubble Sort, Wikipedia). In this algorithm the bigger elements tend to move at the bottom of the array, like bubbles that move at the top of a water bottle.

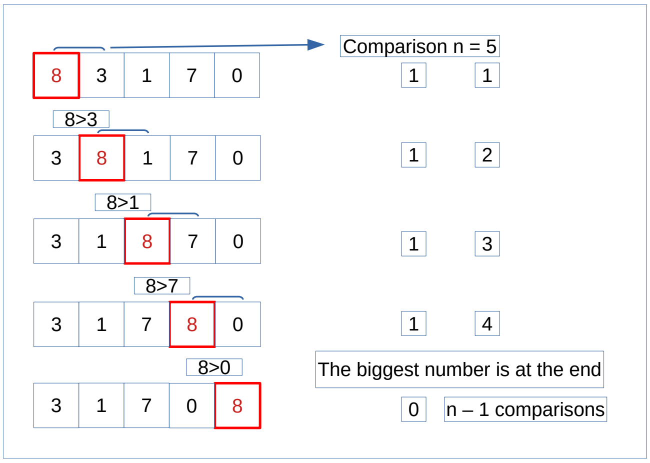


Figure 3.4: Bubble sort algorithm. The biggest element goes at the and of the array.

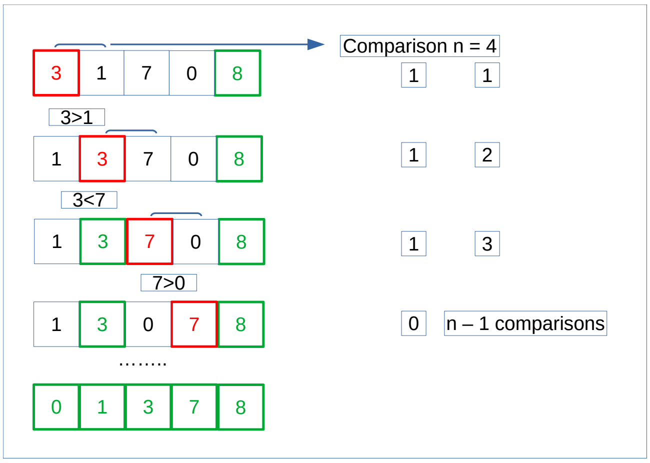


Figure 3.5: The swapping process is repeated until the array is completely ordered.

Efficiency of the Bubble Sort Algorithm

For ordering an array using the bubble sort $n-1$ iterations are required, 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 completely ordered and it is enough to cycle all the elements.

Bubble Sort Implementation

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

```python
def bubble_sort(array_input):
    index = len(array_input) - 1
    sorted = False

    while not sorted:
        sorted = True
        for i in range(0, index):
            if array_input[i] > array_input[i + 1]:
                sorted = False
                array_input[i], array_input[i + 1] = array_input[i + 1], array_input[i]
    return array_input
```

Listing 3.2: Bubble Sort python implementation

3.3 Merge Sort

The **Merge Sort** algorithm works by dividing the array in single elements at first, 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 [20] (Merge Sort, Wikipedia). This way of reducing a big problem in several smaller is called **Divide et Impera** (Divide and Conquer).

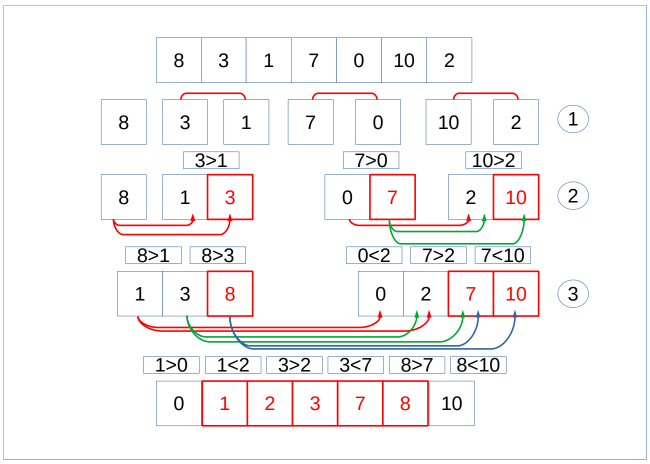


Figure 3.6: Merge sort algorithm. The merging and sorting process is repeated until the array is again unified with all the elements sorted.

The steps are (Figure 3.6):

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 comparison 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 comparison for this step is 5.**

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

Efficiency of the Merge Sort Algorithm

For evaluating the efficiency of this algorithms we have to count the number of iterations and comparisons are being done. By using the example showed in Figure 3.6 we will try to extrapolate a general pattern for an array of dimension n.

The number of comparisons depends by the array size. 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 efficiency 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. From the subprocess of our example 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:

	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

Table 3.2: The number of iterations grows of one every power of two, in others words it grows as $log(n)$.

In conclusion the efficiency if $O(n\ log(n))$, which is better than $O(n^2)$ of the bubble sort.

The memory efficiency 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.

Merge Sort Implementation

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

Listing 3.3: Merge Sort python implementation.

```python
def merge_sort(array_input):

    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
```

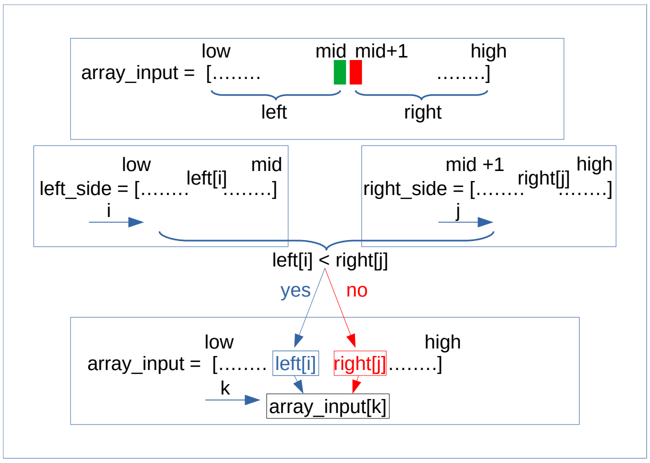


Figure 3.7: Merge sort algorithm implementation.

3.4 Quicksort

The **Quicksort** is a sorting algorithm of the divide et impera type. It works by randomly choosing an element of the array, called pivot, and putting all the bigger and lower values on its left or on its right respectively. This procedure is repeated recursively on the two new subarrays until all the elements have been a pivot [21] (Qicksort, Wikipedia).

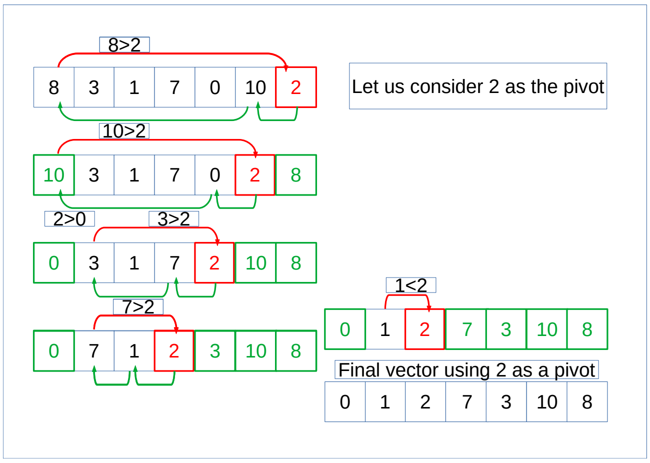


Figure 3.8: Quicksort algorithm steps, part one.

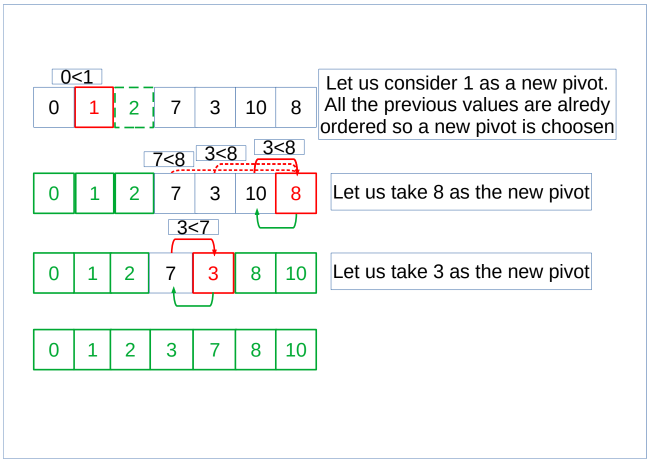


Figure 3.9: Quicksort algorithm steps, part two.

Here are the steps of quicksort algorithm based on the example of Figure 3.8 and Figure 3.9:

1 Let us consider the last element as pivot, two in this case, and let us compare it with all the elements to 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, but going to the next element, 3 in this case.
4 For the pivot 2 all the elements to the left are less than it, and all the element to the right are bigger than it. 2 is not moved anymore.

We can change the pivot and repeat all the steps for the new pivots until all the elements have been a pivot.

Efficiency of the Quicksort Algorithm

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

Let us consider first the worst case. 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 3.10.

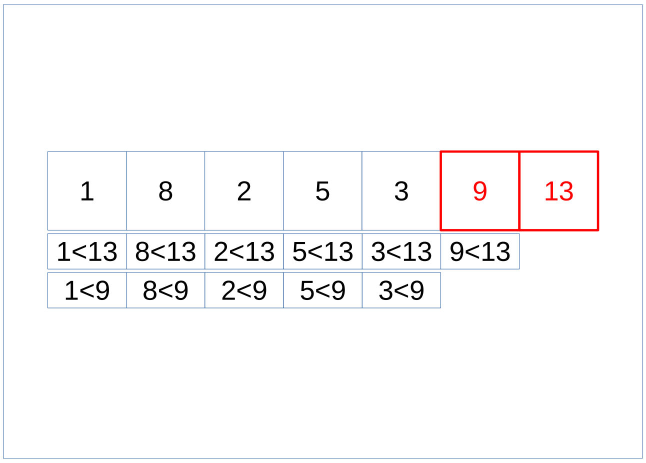


Figure 3.10: Quicksort algorithm worst case.

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

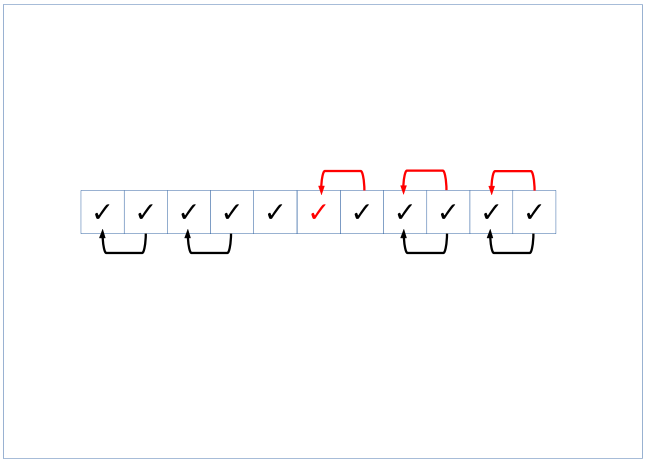


Figure 3.11: Quicksort algorithm best and average case.

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 considering as pivot always the last elements.

Quicksort and Merge Sort Comparison Quicksort is often a better solution than merge sort, because even if its worst case performance 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 an 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 [22] (Heapsort, Wikipedia) when the sorting is done on linked lists that do not require big auxilary space and on very large data sets stored on slow-to-access media, such as disk storage or network-attached storage [21].

In summary:

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

Quicksort Implementation

The following code is the python implementation of the quicksort algorithm [23] (Quicksort Python Implementation).

Listing 3.4: Quicksort python implementation.

```
def merge_sort(array_input):

    elements = len(array_input)

    # Base case
    if elements < 2:
        return array_input

    # Position of the partitioning element
    current_position = 0

    # Partitioning loop
    for i in range(1, elements):
        if array_input[i] <= array_input[0]:
            current_position += 1
            temp = array_input[i]
            array_input[i] = array_input[current_position]
            array_input[current_position] = temp

    # Brings pivot to its appropriate position
    temp = array_input[0]
    array_input[0] = array_input[current_position]
    array_input[current_position] = temp

    # Sorts the elements to the left of pivot
    left = SortingAlgorithms.quicksort(array_input[0:
     current_position])
    # Sorts the elements to the right of pivot
    right = SortingAlgorithms.quicksort(array_input[
     current_position+1:elements])

    # Merging everything together
```

```python
    array_input = left + [array_input[current_position]] + right

    return array_input
```

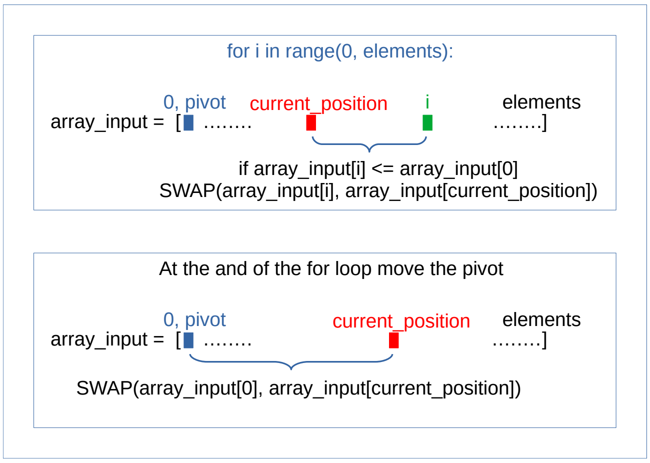
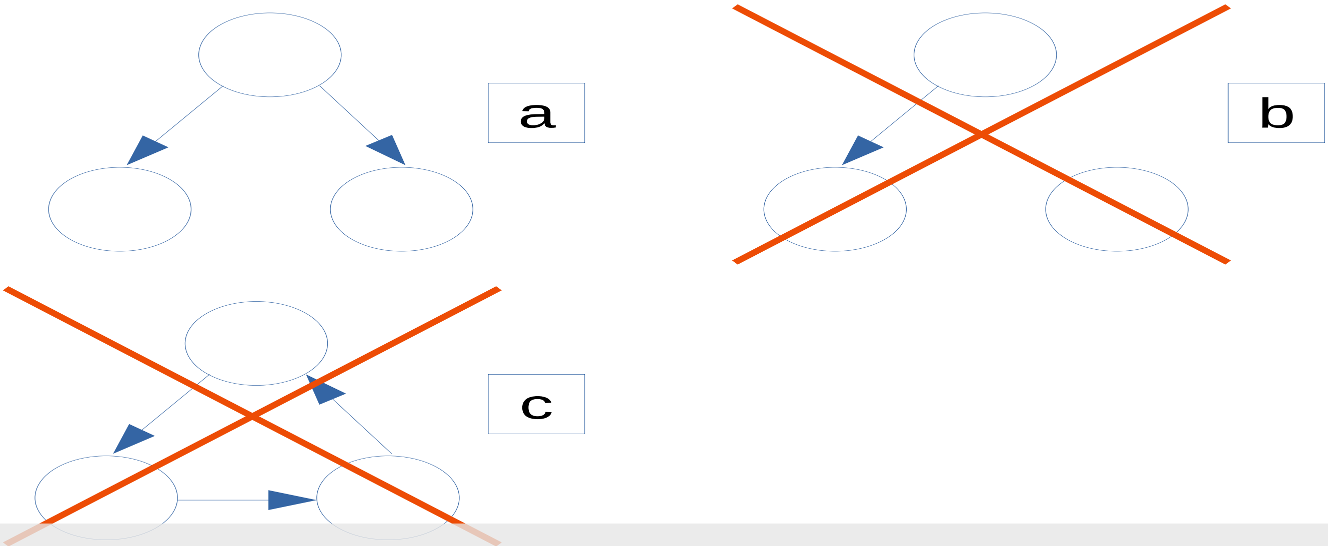


Figure 3.12: Quicksort algorithm implementation.



4 Trees

In this chapter are introduced the fundamentals concepts of tree as abstract data type, and the most used algorithms related to trees [24] (Trees, Wikipedia).

4.1 General Trees

Appendix

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

[13] *Hash Map*. URL: https://en.wikipedia.org/wiki/Associative_array.

[14] *Hash Talbe*. URL: https://en.wikipedia.org/wiki/Hash_table.

[15] *Hash Talbe*. URL: https://en.wikipedia.org/wiki/Hash_function.

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

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

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

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

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

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

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

[23] *Quicksort python implementation*. URL: https://www.educative.io/edpresso/how-to-implement-quicksort-in-python.

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