Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Summary of the Course Algorithms and Data Structures

Soel Micheletti

HS 20

Department of Computer Science, ETH Zürich

Preface

This script is a summary of the lecture *Algorithmen und Datenstrukturen* held by the department of Computer Science at ETH Zurich. This is not official material and it does not cover all the exam relevant topic (e.g. dynamic programming is not covered). If you find any errors please notify me at *soel.micheletti@hotmail.it*. I wish you all the best for the semester :)

Contents

Chapter 1

Introduction and Preliminary Notions

In the famous textbook *Introduction to algorithms* we find the following (informal) definition of algorithm:

> *An algorithm is a well-defined computational procedure that takes some value as input and produces some value as output. An algorithm is thus a sequence of computational steps that transform the input into the output.* [1]

We represent the definition graphically for your convenience

INPUT
ALGORITHM
OUTPUT

The *algorithm box* can do a lot of different things: doing a mathematical computation, suggesting to the user of a social media the people he might know, computing the best combinations of public transport to go from ETH Zurich to Selnau...

In this course we will ”open” the algorithm box and appreciate the beauty of what it contains. We will do this in two different ways: on the one hand we will study ”famous” algorithms, on the other hand you will learn how to design your own algorithms to solve new problems. The second goal is definitely more challenging (but also more interesting) and can be performed in

[1] Cormen et al., *Introduction to algorithms*, MIT Press, 2009. This book is an excellent source, if you have problems in understanding a topic from the lecture, consider reading the corresponding pages in this book!

several ways: new algorithms can be developed by modifying "famous" algorithms, by exploiting some successful paradigm or.. with some phantasy!

In both cases, when studying algorithms, two fundamentals properties need to be considered:

- **Correctness:** given a problem specification, an algorithm has to return the correct answer. Imagine an algorithm $\mathcal{A}$ that it is supposed to sort a list of numbers. If $\mathcal{A}$ returns 3, 1, 2 for the input 2, 1, 3, then it does not solve the initial task and hence it is not correct. An incorrect algorithm is of course usually not very useful. Correctness can be proved via different techniques (induction, invariants, ...)
- **Efficiency:** informally, this means that an algorithm should not take too much time to solve the task. Here we assume that efficiency can be measured by the number of computational steps required by an algorithm. Later in this chapter, we will discuss the asymptotic notation, a convenient tool that allows one to asymptotically represent the number of operations performed by an algorithm with respect to the size of the input. Efficiency is important for two reasons. The first reason is the phenomenon of *combinatoric explosion* (*i.e.* in some cases correct algorithms require more operations than the number of atoms in the universe even for relative small input sizes, which makes those algorithms useless even though they are correct). The second reason is the fact that if an algorithm is asymptotically more efficient than another one, then it might perform significantly better.

So far so good: we have discussed an informal definition of algorithm and we have seen that good algorithms need to be both correct AND efficient. But what kind of algorithms are taught in this course? There will be three main categories of algorithms (see below), but the ultimate goal of this course is to teach you principles that are widely applicable.

- Sorting algorithms. Those algorithm get a list of object as input and return the sorted list (according to some order relationship) as output. Those kind of algorithms are a canonical example of introductory classes because they are very well suited to show students how algorithms are analysed (both in the sense of correctness and efficiency).
- Graph algorithms. Those algorithms get a graph as input and returns the answer to a *question of interest* as output. Example of problem you will be able to solve are: is it possible to reach node v from node u? What is the shortest path from u to v?
- Dynamic programming algorithms. Those algorithms use a table to efficiently solve various tasks. You will look at several examples and you will implement dynamic programming solutions very often (in this course and in the following semesters).

At this point, since this course is called *Algorithmen und Datenstrukturen*, you might wonder what data structures are. Often, also in the lecture, the two concepts are not clearly separated from each other. This happens because they are strictly connected. In the previous bullet list you can find the words *list*, *graph* and *table*. Those terms are quite abstract and are implemented in a programming language with a data structure. If you have a little bit of experience in programming you might know arrays (this is an example of data structure that can be used to implement a list). In general there are several different data structures to represent lists, graphs and tables. For examples you can represent graphs with adjacency matrices or adjacency lists, and there are trade-offs since for some operations adjacency matrices are more efficient, while for others adjacency lists suits better.

1.1 Induction

As we have already seen, correctness is a fundamental property of a useful algorithm. In order to prove the correctness of an algorithm there are different techniques. Here we introduce the proofs by induction, an important tool which is absolutely not restricted to this context, since it can be applied in several different scenarios.

Suppose you want to prove a formula $P(n)$ for all $n \in \mathbb{N}$. The proof by cases is not possible here (you would have to prove $P(0)$, $P(1)$, $P(2)$, ... for all the infinite cases). What is possible is to use the *domino principle* which is formally formulated by the *induction proof rule*. Intuitively you show that the statement you wish to prove holds for the first domino and then, under the assumption that the statement holds for an arbitrary domino, you show that the statement also holds for the next one.

The schema to prove $\forall n \in \mathbb{N}.\ P(n)$ is the following:

- **Base case:** show that $P(0)$ holds.
- **Induction hypothesis:** we assume $P(n)$ for an **arbitrary** n.
- **Induction step:** prove $P(n+1)$ under the assumption $P(n)$ for an *arbitrary* n.

Note that there exists also more involved forms of induction. For example the so called Noetherian induction proves $P(n)$ for all $n \in \mathbb{N}$ by proving $P(n)$ for an arbitrary n under the assumption that $P(m)$ holds for all $m < n$. Also here you have to prove a base case. We introduce Noetherian induction here because sometimes it is used in ETH lectures although it was never introduced before: you are not expected to be able to apply it by yourself (yet), but if you see an unusual form of induction do not be shocked!

We conclude this brief section with an example.

Example 1.1 *Consider the following recursive equation:*

$$T(n) = \begin{cases} 4 + 2T(\frac{n}{8}) & \textit{if } n > 1 \\ 1 & \textit{if } n = 1 \end{cases}$$

You may assume that n is a power of 8. Show that a closed form for the formula is given by $T(n) = 5n^{\frac{1}{3}} - 4$ by induction over n.

- ***Base case:*** $T(1) = 5 \cdot 1^{\frac{1}{3}} - 4 = 1$ *according to the closed formula and 1 is also the result of the recursive formula.*
- ***Induction hypothesis:*** *the statement holds for an n.*
- ***Induction step:*** $T(n) = 4 + 2T(\frac{n}{8}) = 4 + 2(5\frac{n}{8}^{\frac{1}{3}} - 4) = 5n^{\frac{1}{3}} - 4$ *where we used the induction hypothesis in the step from* $4 + 2T(\frac{n}{8})$ *to* $4 + 2(5\frac{n}{8}^{\frac{1}{3}} - 4)$.

1.2 Asymptotic Notation

In the lecture you have discussed Karatsuba's algorithm for the multiplication of two numbers. We don't want to repeat the technical details about the algorithm here, but we stress the reason why such an algorithm was introduced. Although it looks more involved than the method you learned in school, Karatsuba's algorithm is more efficient because it performs less elementary operations. In this course, when we talk about efficiency, we always mean it in a theoretical way. In other words, we assume that an algorithm that performs less elementary operations than another one is more efficient. Note that this might not always be true in practice, where other factors may play a role (*e.g.* memory performance).

As you may recall from the Karatsuba's algorithm example, it is not easy to exactly count the number of elementary operations. For this reason we introduce the asymptotic notation, a useful tool that puts all functions that behave roughly in the same way into the same equivalence class. Concretely, it puts functions which differ only in multiplicative/ additive constants into the same equivalence class. This is particularly useful to make the analysis of algorithms less tedious.

In this course we choose a set of elementary operations which can be executed fast (aka operations that can be executed in constant time or in $\mathcal{O}(1)$) and we count how many of such operations an algorithm does. However, instead of counting the exact number of operations, we consider an asymptotic estimation of the number of elementary operations where we omit the main multiplicative and additive constants (*i.e.* we can not omit those constants if they are in the exponent of a power!) For example, instead of saying *this algorithm does* $3n + 190$ *elementary operations* we will say *this algorithm has linear time*, noted $\mathcal{O}(n)$. Note that this model is an abstraction because an algorithm that does $1234567n + 19091$ operations is considered exactly as efficient as an algorithm that does n operations (they are both $\mathcal{O}(n)$, although they perform a different numbers of operations).

Examples of elementary operations are arithmetic and compare operations (addition, multiplication, power, $\leq$, $>$...), lookup in memory (for example accessing a given element in an array), declaration and value assignment of a variable.

Here we present you the formal definitions of the asymptotic notations.

Upper bound

$$\mathcal{O}(n) := \{f : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$
$$\mathcal{O}(f) \leq \mathcal{O}(g) \Leftrightarrow \exists c, n_0, \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

Lower bound

$$\Omega(n) := \{f : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$
$$\Omega(f) \geq \Omega(g) \Leftrightarrow \exists c, n_0, \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$

The upper bound is mostly used for the worst time of an algorithm. In facts several algorithms have a best case runtime which can be asymptotically better than the worst case runtime. Consider the following example: given a list of n numbers, return true if the list contains the element 5, false otherwise. A simple algorithm to solve this task is looking at every element of the list and stopping whenever we find a 5. In this case the best runtime is constant (if you are lucky the first element of the list is a 5), the worst case runtime is linear (if the list does not contain a 5 you have to look at all the n elements of the list). In this case we say that the algorithm has runtime $\mathcal{O}(n)$ because in worst case we have to look at the whole list.

The lower bound in mostly used to say that it is impossible to beat that runtime. There are examples of problems where the best known algorithm has runtime $\mathcal{O}(n \cdot 2^n)$ but it is theoretically possible that there exist a linear algorithm that nobody has found. In this case we can write that the algorithm has a lower bound $\Omega(n)$.

There exist also a definition for the **tight bound**, which is used when an algorithm has exactly that asymptotic runtime (in the best and in the worst case).

$$\Theta(f) = \Theta(g) \Leftrightarrow \mathcal{O}(f) \leq \mathcal{O}(g) \text{ and } \Omega(f) \geq \Omega(g)$$

1.3 Recursion

Imagine that you have to solve an algorithmic problem. In order to solve the problem you create a function ALGO(N) which solves the task for an input n. Solving the problem *recursively* means that, in your function ALGO(N), you use the function ALGO again, but with an input different than n. The idea is that the function ALGO(N) will call the function ALGO a finite number of times, until it reaches a base case which is solved non-recursively. A very important property that you have to consider when using recursion is *termination*. If your recursive solution does not reach a base case then you will call your function an infinite number of times (ideally, in practice your program would run out of memory) without reaching the solution.

Here a couple of hints on how to solve a problem with recursion:

- Consider the base case (or base cases). Those should not be too complicated but they are the heart of your solution: without them you can not solve the problem.
- Think in general: if you have a big input, how can you solve the problem if you already have the solution for a smaller input? Usually you do like this: you have to solve the problem for input n. Assuming you have the solution for the input $n-1$, how can you solve the problem?
- Think about termination. Does your solution always reach a base case?

The best way to understand recursion is solving exercises. I suggest you this ones:

1. Compute the sum 1+2+3+...+n recursively.
2. Implement the multiplication of two numbers a and b recursively.
3. Reverse a string recursively (e.g. "Hello" becomes "olleH").
4. Return the length of a string recursively.
5. Return whether a string is palindrome or not.
6. Given two string, return all possible interleavings (from a previous exam of *Einführung in die Programmierung*).
7. Print all possible anagrams of a given string.

Here we think it's worth to answer some frequently asked questions

What is the relation between recursion and induction?

They are strongly correlated (the domino principle applies also to recursion). Induction proves that the claim holds for a domino and then proves that if the claim holds for an arbitrary domino then it must also hold for the next domino. Recursion assumes that you have a solution for a smaller input and then it correctly solves the problem for the required input. The difference between recursion and induction are quite complicated, so we distinguish them functionally: induction is used for proofs, recursion for definitions and algorithm declarations.

I heard that recursion is "bad" and iteration is "good". What does it mean?

Recursion and iteration (for example for loops) can be used to solve the same problem. Recursion is a *top-down* approach (you start from the input of size n and then you go backwards to $n-1$, $n-2$ and so on until you reach the base case). Iteration is a *button-up* approach (you start from the base case and then you take more elements from the input until you reach the answer). In general the rule is: *use iteration instead of recursion* (this is the idea of dynamic programming). The reasons of this rule are multiple and you will get a complete insight of the problem later in the semester (but in general an important reason is that the number of recursive calls is limited, while iteration works also for bigger inputs). However recursion and iteration are strictly related and being able to think recursively is fundamental.

How is recursion related to the stack overflow problem?

Recursive calls are saved in a stack (this is not the data structure of this course, is a region in memory you will learn in the course *Systems Programming and Computer Architecture*). If you do too many recursive calls then the stack where recursive calls are saved will run out of memory and the computer will complain. This usually happens if you don't have a correct base case or if your input is too big.

Example 1.2 (Star Finding Algorithm) *There are n people in a room. We want to know whether there is a star or not (and when there is a star, which person is the star). A star is a person that is known by everybody, but he/ she does not know anybody. For example, if Lionel Messi enters the room we all know who he is, but he does not know us. For this problem we consider Yes/ No questions as elementary operations. Other kind of questions are not permitted. We observe that there must not always exist a star; for example, if everyone knows everybody else, then there is no star. Moreover, we show with a simple indirect argument that it is impossible to have two stars. Assume there are two stars S_1 and S_2. By definition of star, S_1 knows S_2 (otherwise S_2 would not be a star according to our definition). But in this case it does not hold that S_1 does not know anybody. This is a contradiction to the fact that S_1 is a star. Hence in general we have either no stars or exactly one star. Now we come to the big question, how do we solve the problem? A naive algorithm consists in collecting complete information, i.e. asking to each person x_i, whether it knows x_j for all choices of $j \neq i$. We can summarize the results in a table and we look if there is any person which is known by everybody else and does not know anyone. This algorithm requires $n \cdot (n-1)$ elementary operations and hence has asymptotic complexity $\Theta(n^2)$. Can we do better? The answer is yes, and we can use recursion to design a faster algorithm. The first step in order to design a recursive algorithm is considering the base case, i.e. when there are exactly two people A and B in the room. We proceed by asking to A whether he knows B and we ask B whether he knows A. If Aknows B but B does not know A, then B is the star. If B knows A but A does not know B, then A is the star. The base requires exactly 2 elementary operations. So far so good, but how can we recursively solve the problem with n people? In order to answer this question, we do the following fundamental observation. If we take two out of n people in the room, we can ask to the first person* Do you know anybody in the room? *If the answer is yes, then this people is definitely not a star. Otherwise, the other person is not a star, since the first person does not know him. This insight is particularly useful to design the following algorithm. Given n people in a room, we can pick a person which is not a star with a single elementary operation. We isolate this person. Then we can solve the problem recursively for the remaining $n-1$ persons and we can find a candidate star (a star between this $n-1$ persons). At the end we can ask two questions to the person that we have isolated at the beginning in order to know whether the candidate star between the $n-1$ persons is effectively a star or not. This algorithm is correct, but is it also efficient? In order to determine the number of elementary operations that this algorithm requires we write the following recursive formula:*

$$T(n) = \begin{cases} 2 & \textit{if } n = 2 \\ 1 + T(n-1) + 2 & \textit{otherwise} \end{cases}$$

Which we can expand as follows

$T(n) = 3 + T(n-1) = 3 + 3 + T(n-2) + \cdots + 3 + T(2) = 3(n-2) + 2 = 3n - 4.$

This suggest (but does not prove) that the runtime of this algorithm is in $\mathcal{O}(n)$. Intuitively we can not do better than this (asymptotically), since we must consider each person in the room in order to answer the question. Still, we have to prove our claim about the solution of the recursion. The following induction gives a formal argument for that.

- ***Base case:*** *We have $3 \cdot 2 - 4 = 2$ operations according to the close formula and also according to the recursive formula.*
- ***IH:*** *the proposition hold for an (arbitrary) n.*
- ***Induction step:***

  $$T(n+1) = 3 + T(n) = 3 + (3n - 4) = 3n - 1 = 3(n+1) - 4$$

  where we used the induction hypothesis.

1.4 Maximum Sub-Array Sum

Before moving to sorting algorithms, we examine another introductory problem with multiple algorithmic solutions.

Consider a list $a = a_1 \dots a_n$ of n numbers (positive and negative). The task is finding the indices i and j that maximize the following expression:

$$\sum_{k=i}^{j} a_k$$

In other words, we want to find the indices that maximize the sum of the elements between them. A naive algorithm tries all possible indices i and j and sums the elements of the list between those indices. What is the asymptotic runtime of such algorithm? There are $n-1$ choices for index 1, $n-1$ for index 2 and so on. This means that we have $\sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$ couples of indices to try, where we used Gauss formula. Of course, for each choice of the indices, we have to do the sum of the elements in the range. This operation takes, in the worst case, n operations. This gives an upper bound of $\mathcal{O}(n^3)$ operations for the algorithm. An important insight about this algorithm is that we repeat some computation several times. For example, we perform the sum of the elements between 1 and $n-1$ and the sum of the elements between 1 and n. Both sums takes $\mathcal{O}(n)$. Actually. it would be possible to remember the result of the sum between 1 and $n-1$ and then sum a_n. In this case, the second sum could be done in constant time instead of linear time. This observation suggest that we can do some pre-computation to improve the efficiency of our algorithm. Concretely, we compute an array b such that b_i is the sum of the elements between a_1 and a_i. For example for $a = 1, 4, -3, 12, 7, -3, 6$ we have $b = 1, 5, 2, 14, 21, 18, 24$. This new array b allow us to compute $\sum_{k=i}^{j} a_k = b_j - b_{i-1}$ in constant time in the worst-case scenario. A new algorithm can hence be designed by trying all possible indices i and j (in $\mathcal{O}(n^2)$, as before) and doing the sum of the elements in a between those indices (in constant time). The total asymptotic complexity is now $\mathcal{O}(n^2)$. We have improved the runtime of the naive algorithm, but we still can do better. A famous algorithm paradigm is called divide-and-conquer and it divides the problem in two (or more) sub problems, solves them recursively and then puts the solution together. We can apply this paradigm by splitting a in two parts of (roughly) the same length. Then we compute the maximum sum in both sub arrays. We have three possibilities:

- The maximum sum is on the left part of the array.
- The maximum sum is on the right part of the array.
- The maximum sum is between both parts of the array.

In order to address the third case we use maximum prefix/ suffix sum:

- We do an array with the prefix sum. Let p be the maximum element in this array.
- We do an array with the suffix sum. Let q be the maximum element in this array.
- The maximum sum *between the two sub arrays we have solved recursively* is $p + q$.

Hence the divide-and-conquer algorithm works as follows: we solve the sub arrays recursively (let m_1 be the maximum sum on the left array and let m_2 be the maximum sum on the right array). Find the maximum sum $p + q$ between the two arrays. Return the maximum between m_1, m_2 and $p + q$. The runtime is given by the following recursion:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + an & \text{otherwise} \end{cases}$$

By telescoping the formula we find $\mathcal{O}(n \log n)$ as total complexity (try to do it as exercise, but I suggest you to remember this recursive formula and its result because it is very common).

We have improved the runtime from the initial $\mathcal{O}(n^3)$ to $\mathcal{O}(n \log n)$. However, we can still do better.

We use two variables: *max* (which saves the maximum subarray sum) and *randmax* (which saves the maximum subarray sum starting from the last point where a maximum subarray sum could start). In facts if the sum of the elements between indices 1 and j is negative and $j < 0$, is impossible that a maximum subarray sum starts from j. We iterate through the array and we update the maximum when $randmax > max$ and we reset the value of *randmax* when it is less than zero.

Algorithm 1: MaxSubArraySum(a)

```
max ← 0 ;
randmax ← 0 ;
for t=1, ..., $n - 1$ do
    randmax ← randmax + $a_i$ ;
    if $randmax > max$ then
        max ← randmax ;
    end
    if $randmax < 0$ then
        randmax ← 0;
    end
end
```

This algorithm is asymptotically optimal since, In order to solve the problem, we need at least to look at each element of the array. We say that n is a lower bound of the algorithm (*i.e.* we can not do better) and we write it $\Omega(n)$.

Chapter 2

Searching and Sorting

In this chapter we will study two algorithmic problems that are strongly related: searching and sorting. These are classical problems of introductory classes, and there are several reasons to study them. First of all, searching and sorting are very important tasks: Google, for example, became very popular thanks to a particularly powerful searching algorithm. Moreover, they are an excellent example of algorithm design and they provide a neat framework to show some techniques of algorithm analysis. Last but not least, they give insights about a fundamental trade-off: it is clear that searching in a ″ordered″ (or sorted) environment is easier, but it is also clear that ″ordering″ (or sorting) a ″messy″ (or unsorted) environment has a cost. In this chapter we will see in which situations it is (not) worth it to pay the price of sorting in order to be more efficient in the process of searching.

2.1 Searching

We start from the first problem: searching. In this lecture, we will only consider the problem of searching a particular object in an array of that type of object. For example, we want to find a given integer in an array of integers. More formally, we have the following.

Input: an array $a[1], a[2], \dots a[n]$ and an element b

Output: k s.t. $a[k] = b$ or *not in the array* if the element is not in the array.

We observe that the input array can be either sorted or not. It is intuitively clear that searching in a sorted array is easier. Compare, for example, the tasks of searching the word *algorithm* in an English dictionary or in a business book: the former task seems much easier than the latter one. In the case of searching a word in a business book, we can not do much better than reading through the book until we either find the word or we finish the book (if the word is not contained). In the case of the dictionary, however, we can exploit the fact that the words are alphabetically sorted and, for example, open the dictionary in the middle and determine whether the word is in the first or in the second half. These two approaches are very similar to the ones we will discuss in this section. Reading the whole book until we find the word is called *linear search*, the searching algorithm in the dictionary is called *binary search*. Linear search is more general (it does not require the array to be sorted), but less efficient than binary search (which requires the array to be sorted). Note that, in the original problem of finding an element in an array, nothing prevents us from first sorting the array and then use binary search.

Now we will present the two algorithms more in detail, and then we will determine in which cases it is worth it to pay the cost of sorting in order to gain efficiency in searching (and when not).

2.1.1 Linear search

The idea is very simple. If you have to search a word in the book of algorithms and you don't have any idea of where it can be (e.g. the word *cat*), you simply read the book from the beginning and you stop when you read the work *cat*. The pseudocode of the algorithm is the following:

Algorithm 2: LinearSearch(a[], b)

```
for $i = 0, \ldots, n$ do
    if $a[i] = b$ then
        return i;
    end
end
return b is not in the array;
```

It is clear that, in the worst case scenario, we have to iterate through the whole array. Since each element of the array requires a constant amount of elementary operations (just a comparison), we have an asymptotic complexity of $\mathcal{O}(n)$.

2.1.2 Binary search

Recall that this algorithm, that only works on sorted arrays, can be remembered with the dictionary analogy. Imagine that you have to search a word in the dictionary (which is alphabetically sorted). In this case you don't search for the word sequentially, but you can do the following. You open the dictionary in the middle: if the word you are looking for is in that page you are over, otherwise you know if the word you are looking for is on the pages which comes before or after the middle page. Once you know this information you do the same kind of search in the previous/ following pages. This idea applies also to sorted arrays: you look at the element in the middle and (if this was not the element you were searching) then you look only at the left/ right side of the array. The pseudocode for binary search follows exactly this principle (in order to use it on an array you call BINARYSEARCH(A, 1, N, B)):

Algorithm 3: BinarySearch(a[], min, max, b)

```
while $min \leq max$ do
    mid ← $\frac{min+max}{2}$;
    if $a[mid]=b$ then
        return mid;
    end
    if $a[mid] < b$ then
        min ← mid+1;
    end
    if $a[mid] > b$ then
        max ← mid-1;
    end
end
return b is not in the array
```

Here the analysis of the runtime is less straightforward. In particular, the analysis of the recursive version of the algorithm (that simply applies binary search either on the left or on the right side of the array) comes to rescue. With the techniques of the first chapter, we can come up with the following recursive equation:

$$T(n) = \begin{cases} T(\frac{n}{2}) + c & \text{if } n > 1 \\ \tilde{c} & \text{if } n = 1 \end{cases}$$

Which gives (for example with telescoping and an induction proof) a worst case runtime of $\mathcal{O}(\log n)$.

In class you have seen variants of binary search (e.g. interpolation search and exponential search). This algorithms have the same worst case complexity of binary search, but they can have a better performance in some cases. As an intuition recall the dictionary example: if you have to search the word *cat*, you will most likely begin your search at the beginning of the dictionary and then apply the binary search. This works speeds-up a lot searches on dictionaries because you have an idea of how words are distributed. You can also assume a prior distribution of the elements in the array (for example a uniform distribution) and then use the same trick.

2.1.3 A fundamental trade-off

We have seen that linear search, a general algorithm that works also on usorted arrays, has a worst case complexity of $\mathcal{O}(n)$. We also know that, on sorted array, we have the binary search algorithm that requires $\mathcal{O}(\log n)$ elementary operations. Now the question is: what should we do if we have an unsorted array? Should we first sort it and then use binary search or should we directly go with linear search? In order to answer the question, we need to know the complexity of sorting. For example, if we were able to sort in $\mathcal{O}(\log n)$, then the strategy of sorting and using binary search would be better. On the other hand, if sorting would require $\mathcal{O}(n^2)$ operations, we should always go for linear search. In the next sections, we will show some algorithms to sort arrays with a runtime of $\mathcal{O}(n \log n)$. We will also show that we can not do better (in other words, we will show that $\Omega(n \log n)$ is a lower bound for comparison based sorting). This insight, which is just a black box for now, allows us to answer our question. If we have to search for a single value, then linear search is better (a cost of $\mathcal{O}(n)$ against $\mathcal{O}(n \log n)$), but if we have to search k elements, then we have the following.

- **k times linear search:** has runtime $k \cdot \mathcal{O}(n) = \mathcal{O}(kn)$.
- **Sorting + k times binary search:** has runtime $\mathcal{O}(n \log n + k \log n)$.

Depending on the value of k, one of the two strategies is better. In general, we have:

- $k < \mathcal{O}(\log n)$: in this case linear search is better ($\mathcal{O}(n)$ is better than $\mathcal{O}(n \log n)$).
- $k > \mathcal{O}(\log n)$ in this case is better to sort and then doing binary search (in this case in the factor $k \log n$ dominates in the runtime of the sorting+searching approach and hence it is more convenient than kn).

2.2 Basic Sorting Algorithms

We have indirectly introduced the problem of sorting in the previous section. In particular, we have seen that a sorted array allows a more efficient searching process. But, how can we actually sort an array? In this section we present a bunch of basic sorting algorithms. These algorithms are intuitively easy (they are probably the first algorithms one might come up with) but, as we will see, not particularly efficient.

2.2.1 Bubble Sort

Imagine to iterate once through an array a of length n and, every time you find two elements a_i, a_{i+1} such that $a_i > a_{i+1}$, swap this two elements. After one iteration, the largest element of the array is located at the right position (i.e. at the end). By repeating this process n times, we sort the array (at iteration i, we put the i-th largest element in the right position). This is an intuitive proof of the correctness of the following sorting algorithm:

Algorithm 4: BubbleSort(a[])

```
for $i = 1, \ldots, n-1$ do
    for $j = 1, \ldots, n-1$ do
        if $a[i] > a[i+1]$ then
            Swap($a[i], a[i+1]$);
        end
    end
end
```

It is easy to see from the pseudocode of the algorithm (two nested for loops which iterate through the array), that the worst time asymptotic complexity is $\mathcal{O}(n^2)$. Note that if, during an execution of the inner loop, we don't swap any elements, then the array is already sorted. Hence we can slightly modify our algorithm such that it stops when the array is sorted. This gives a best case runtime of $\Omega(n)$, but in the worst (and also in the average case), the runtime is still quadratic. We want to point out that, apart from a single temporary variable for the swap operation, this algorithm does not require any additional memory. This is a nice property and we will say that algorithms that satisfy it are *in place*.

2.2.2 Selection Sort

Selection sort is an algorithm that logically operates on two subsets S and $\bar{S}$. S contains the sorted part of the array, $\bar{S}$ the unsorted part. Of course, at the beginning, all elements of the array are in $\bar{S}$, Then we do n iterations, where at each iteration we take an element from $\bar{S}$ and we put it in S. How do we do it? We just iterate through all elements in $\bar{S}$. we find the largest one and we put it as smallest element in S. After n iterations all elements of the array are in S, *i.e.* we have sorted the array. More concretely, selection sort can be implemented as follows.

Algorithm 5: SelectionSort(a[])

```
for $k = n, \dots, 2$ do
    Iterate in the first k elements of the array. Let $a[m]$ be the
      maximum element in this set. ;
    Swap($a[m], a[k]$;
end
```

The number of comparisons is $n-1$ in the first iteration, $n-2$ in the second one, the $n-3$ and so on. We get a total of $\sum_{i=1}^{n-1} i$ which, according to the well known Gauss formula, is again in $\mathcal{O}(n^2)$. Similarly as bubble sort, also selection sort is in place.

2.2.3 Insertion Sort

Insertion sort follows a similar idea of how we usually sort a deck of cards. Similarly as in selection sort, also here we consider a set S of sorted elements and a set $\bar{S}$ of elements that still need to be sorted. The difference is how, at each of the n iterations, an element from $\bar{S}$ is added to S. Instead of searching the largest element in $\bar{S}$, we take an arbitrary one. Then we scan the sorted sequence $\bar{S}$ and we find the right place to insert it. In pseudocode, we have the following.

Algorithm 6: InsertionSort(a[])

```
for $i = 1, \dots, n-1$ do
    $j \leftarrow i$;
    while $j > 1$ AND $a[j] < a[j-1]$ do
        Swap($a[j]$, $a[j-1]$);
        $j \leftarrow j - 1$
    end
end
```

A careful analysis of the previous pseudocode would show that the runtime of insertion sort is the same we have encountered with bubble and selection sort: we have a worst case of $\mathcal{O}(n^2)$. The careful reader, at this point, might think that insertion sort can be implemented in $\mathcal{O}(n \log n)$: at each iteration, we have to pick an element from the unsorted set and put it in the right place in the sorted sequence. The pseudocode above uses linear search to find the right spot in S but, after all, one could use binary search and be more efficient. The idea might be true, but the problem is that, once we have found the right spot in S for the new element, we have to shift the larger elements to the right in order to make space for the new element. This operation requires linear time, and hence the runtime remains quadratic in the length of the array.

2.3 HeapSort

In the previous section we discussed three sorting algorithms with worst case complexity $\mathcal{O}(n^2)$. In the section where we introduced the trade-off between the cost of sorting and the consequent speed up on searching, however, we mentioned that it is possible to sort with $\mathcal{O}(n \log n)$ operations in the worst case. Heap sort is a first example of algorithm with such runtime and it is based on the heap data structure. We will discuss heaps in Chapter 3, but we summarize the operations that they support with their respective complexities here:

- Add/ remove an element to the heap can be done in $\mathcal{O}(\log n)$
- Given a set of number, heaps allow to find the maximum element in the set in $\mathcal{O}(1)$

This data structure is very convenient for our purposes. In facts, we can combine them with the idea of selection sort to obtain an $\mathcal{O}(n \log n)$ algorithm. Selection sort just looks for the largest element in the unsorted part $\bar{S}$ of the array in linear time and puts it as smallest element of the sorted part S. If we put the elements of the unsorted part in a heap, however, this operation can be done in logarithmic time. By repeating this operation until we have sorted the array, we get the following algorithm.

Algorithm 7: HeapSort(a[])

```
for $i = 1, \dots, n$ do
    Insert $a[i]$ in the heap;
end
for $i = 1, \dots, n$ do
    $max \leftarrow$ maximum element in the heap;
    Put $max$ as smallest element of $S$ and remove $max$ from the heap;
end
```

And from the specification of the operations (which are stated as black box in this chapter,but are explained in greater details in Chapter 3) it is clear that the asymptotic complexity is $\mathcal{O}(n \log n)$.

2.4 Merge Sort

In Chapter 1, we mentioned that divide and conquer is a useful paradigm for algorithm design. Can we also apply it to the sorting problem? The answer is yes, and the resulting algorithm is called merge sort. Concretely, the idea is the following:

1. Divide the array in two parts (left part and right part).
2. Sort the left part recursively.
3. Sort the right part recursively.
4. Merge the two sorted parts in a single sorted array.

For sure, we need a base case: this happens when we have either an empty array or an array containing a single element.

2.4.1 Warm-up: execution on paper

Consider the initial array:

5	2	7	-5	16	12	22	1

We partition the array in the middle and we get the two sub-arrays:

- The sub-array A:

5	2	7	-5

- The sub-array B:

16	12	22	1

This two arrays are sorted recursively. Here is where the magic of recursion comes into play: you can assume that your algorithm works when you write your algorithm. We obtain:

-5	2	5	7

and

1	12	16	22

Now the big question: how can we *conquer* the problem? How can we merge the two sorted arrays? The idea is the following: we keep a pointer to the first element of the A (now sorted) and another pointer to the first element of B (now sorted). Then we repeat the following task until we are finished:

Which element is smaller? The one indicated by the pointer on A or the one indicated by the pointer of B? We pick the smallest one and we put it in the solution. Then we move to the right the pointer which used to point to the smallest element. When one of the two pointers point out of the array (i.e. we have already put all the elements of A or B into your solution). At this point, we simply copy the other sub-array in the solution.

Let's apply the merge part to the sorted versions of A and B:

A (the bold element is the one the pointer is pointing to)	**B ((the bold element is the one the pointer is pointing to)**	**Sorted array**
-5, 2, 5, 7	**1**, 12, 16, 22	∅
-5, **2**, 5, 7	**1**, 12, 16, 22	-5
-5, **2**, 5, 7	1, **12**, 16, 22	-5, 1
-5, 2, **5**, 7	1, **12**, 16, 22	-5, 1, 2
-5, 2, 5, **7**	1, **12**, 16, 22	-5, 1, 2, 5
-5, 2, 5, 7	1, **12**, 16, 22	-5, 1, 2, 5, 7

At this point we are over since the pointer in A is over the last element. We just have to copy the part from the pointer to B to the end of B in the sorted array and we get

-5	1	2	5	7	12	16	22

as desired.

2.4.2 Analysis

We can analyse merge sort with the same method used for the previous divide-and-conquer algorithms. The first question is: what is the complexity of merging the two sorted sub-arrays? Since we do a constant number of operations for every element of both arrays (hence $c \cdot n$ operations for an

Algorithm 8: MergeSort(a[])

```
if $a$ has length 0 or length 1 then
    return a;
end
$left \leftarrow$MergeSort(first half of $a$);
$right \leftarrow$MergeSort(second half of $a$);
$leftPointer \leftarrow 1$;
$rightPointer \leftarrow 1$;
$S \leftarrow \emptyset$;
while $leftPointer < lengthofleft$ AND $rightPointer < lengthofright$
 do
    if $left[leftPointer] \leq right[rightPointer]$ then
        $S \leftarrow S \cup left[leftPointer]$;
        $leftPointer \leftarrow leftPointer + 1$;
    end
    else
        $S \leftarrow S \cup right[rightPointer]$;
        $rightPointer \leftarrow rightPointer + 1$;
    end
end
Add the array which still has a pointer inside into S
```

appropriate constant c), we have that the complexity of merging is $\mathcal{O}(n)$. The final runtime will be (under the assumptions that $T(1) = 1$ and that n is a power of two, i.e. $n = 2^k$):

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \mathcal{O}(n) \\ &= 2(2T(\frac{n}{4}) + \frac{n}{2}) + \mathcal{O}(n) \\ &= 4T(\frac{n}{4}) + 2\mathcal{O}(n) \\ &= \dots \\ &= kT(1) + k\mathcal{O}(n) \end{aligned}$$

From the last step, since $k = \log n$, we deduce that merge sort has a complexity of $\mathcal{O}(n \log n)$.

A very important observation is that MergeSort is not in place. In facts, for the merging part, we need extra space for storing the solution. This means that the algorithm has a complexity of $\mathcal{O}(n)$, but it needs $\Theta(n)$ extra memory.

For this reason we still want to do better: we want an $\mathcal{O}(n \log n)$ algorithm which is in place. We will illustrate a candidate algorithm that satisfies these goals in the next section.

2.5 Quick Sort

In the previous section we have seen that merge sort has a worst-case complexity of $\mathcal{O}(n \log n)$. However, to merge two sub-arrays of length $\frac{n}{2}$, we need $\Theta(n)$ extra place. Here we present an algorithm that is very similar to merge sort (it also follows the divide and conquer paradigm), but it does not have to perform a merge in order to be correct. In facts, if we know that all entries in the left sub-array are less equal than the entries in the right sub-array, we do not have to merge them (and hence we need only constant extra place). As we will see, quick sort is a solution in this sense. However quick sort has to face a performance challenge: merge sort has a worst-case complexity of $\mathcal{O}(n \log n)$ because we know that the left and right sub-arrays that we sort recursively have $\frac{n}{2}$ elements. As we will see, it is not easy to perform a partition and at the same time keeping the recursion tree balanced. Quick sort is great because we perform the partition to save the extra space and we expect the recursion tree to be *almost balanced*, i.e. we expect a complexity of $\mathcal{O}(n \log n)$. More concretely, the idea is the following.

1. We pick an arbitrary element p of the array (p is called pivot).
2. We create a sub-array A with all the elements $\leq p$ and sort this sub-array recursively.
3. We create a sub-array B with all the elements $> p$ and sort this sub-array recursively.
4. The final sorted array is the union of $A \oplus p \oplus B$[1].

Observe that this in only an intuition. The previous ”algorithm” does not provide any base-case and it is only a naive ”implementation” since we want quick sort to operate in place. In order to be more comfortable with the algorithm we consider a concrete example.

Consider the initial array:

5	2	7	-5	16	12	22

We chose 7 as the first pivot. After the first partition we get:

[1]With $\oplus$ we denote the union of two arrays

- The sub-array A:

5	2	-5

- The sub-array B:

16	12	22

First, we sort A. We chose 2 as pivot. The set AA of elements ≤ 2 contains -5 (base case). The set AB of elements > 2 contains 5 (base case). Now we know that $AA \oplus 2 \oplus AB$ is sorted. So we get that A sorted is:

-5	2	5

Then, we sort B. We chose 22 as pivot. The set BA of elements ≤ 22 contains 16 and 12. The set BB of elements > 22 is empty (base case). We have to sort BA recursively (we omit it here) and we obtain:

12	16

Now we know that $BA \oplus 22 \oplus BB$ is sorted. So we get that B sorted is:

12	16	22

Now we complete the task for A. The result is $A \oplus 7 \oplus B$.

-5	2	5	7	12	16	22

2.5.1 Implementation tips

In the official lecture notes, you can find a pseudocode for quick sort that works only on array with pairwise different elements. Here, we present an alternative implementation which works as well with duplicate elements (and it is also in place).

Algorithm 9: QuickSort(a[], left, right)

```
leftPointer ← left;
rightPointer ← right;
pivot ← a[right];
while leftPointer ≤ rightPointer do
    while a[leftPointer] < pivot do
        leftPointer ← leftPointer + 1;
    end
    while a[rightPointer] > pivot do
        rightPointer ← rightPointer - 1;
    end
    if leftPointer ≤ rightPointer then
        Swap(a[leftPointer], a[rightPointer]);
        leftPointer ← leftPointer + 1;
        rightPointer ← rightPointer - 1;
    end
end
if left < rightPointer then
    QuickSort(a, left, rightPointer);
end
if leftPointer < right then
    QuickSort(a, leftPointer, right);
end
```

The procedure QuickSort(a, left, right) sorts the array a from the index at the position left to the position at the position right. This means that when we want to sort an array a with n elements we call QuickSort(a, 1, n).

The idea is very similar to the one of the script. We have two bounds of the array (these are saved in the variables $left$ and *right*). Then we instantiate two pointers $leftPointer$ and *rightPointer* for which the following invariant is true: all elements left of $leftPointer$ are less equal the value of the pivot (analogously all elements right of *rightPointer* are greater equal the value of the pivot). The loop in the lines 4-12 performs the partition (i.e. it puts $leftPointer$ as right as possible and it puts *rightPointer* as left as possible). Note that after the execution of this loop between *rightPointer* and $leftPointer$ there is the pivot (or more instances of the pivot). From the fact that $rightPointer \leq leftPointer$ and the invariants we see that we have performed a correct partition. If there are still elements on the left of *rightPointer* we sort this part of the array and if there are still elements on the right of $leftPointer$ we sort this part of the array recursively.

2.5.2 Analysis

Quick sort is correct for an arbitrary choice of the pivot p. The performance, however, strongly depends on the choice of p. Quick sort on an array of length n has the best performance if it contains $\frac{n}{2}$ elements smaller equal the pivot and $\frac{n}{2}$ elements greater than the pivot (and of course all sub-arrays must choose the pivot *in the middle*). The intuition is that we want the sub-trees of the recursion tree to be as balanced as possible. But imagine that we take always the smallest (or the greatest) element of the list as pivot. What is the asymptotic complexity in this case? To do the partition we need to look at $n-1$ elements and since in this case every time we call quick sort on an array of $n-1$ element we obtain (consider $T(1) = 1$):

$$\begin{aligned} T(n) &= T(n-1) + \mathcal{O}(n) \\ &= T(n-2) + 2\mathcal{O}(n) \\ &= \dots \\ &= T(1) + n\mathcal{O}(n) \\ &\in \mathcal{O}(n^2) \end{aligned}$$

Let's now take a look at the complexity if we choose a good pivot (which partitions the initial array in two array of more or less the same length). Assume that $T(1) = 1$ and that n is a power of two, i.e. $n = 2^k$.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \mathcal{O}(n) \\ &= 2(2T(\frac{n}{4}) + \frac{n}{2}) + \mathcal{O}(n) \\ &= 4T(\frac{n}{4}) + 2\mathcal{O}(n) \\ &= \dots \\ &= kT(1) + k\mathcal{O}(n) \end{aligned}$$

From the last step, since $k = \log n$, we deduce that quick sort has a best case complexity of $\mathcal{O}(n \log n)$. But why should we consider quick sort (that has a worst complexity of $\mathcal{O}(n^2)$) if we have merge sort that has a worst case of $\mathcal{O}(n \log n)$? There are two main reasons:

- Quick sort has an average complexity of $\mathcal{O}(n \log n)$ and, as you will see in the next semester, choosing the pivot p uniformly at random leads to a very good performance. In practice, randomized quick sort is considered better than merge sort. In this particular case, the situation where the complexity is quadratic is very rare: in facts, an "unlucky" pivot must be chosen (too) many times.

- Merge sort needs $\Theta(n)$ extra space in order to merge the two sorted sub-arrays. Quick sort is in place and this is a desirable property.

Intuitively we see that quick sort needs only constants extra place because it uses only the initial array and some temporary variables. However, as you will see in *Systems Programming and Computer Architecture*, for every recursive call, some space is allocated in memory. In particular with quick sort, we need $\mathcal{O}(n)$ recursive calls, and each of them needs at least constant place. Actually there is a trick (see script for details) to avoid this issue, but this requires an even more involved implementation.

2.6 Lower Bound

In this chapter, we have introduced six sorting algorithms. Three of them (heap sort, merge sort and quick sort) have worst time complexity $\mathcal{O}(n \log n)$. In our discussion about the searching/ sorting trade-off, we mentioned that we can not find a sorting algorithm that uses less elementary operations than that in the worst case. A more precise statement would be that $\Omega(n \log n)$ is a lower bound for comparison based algorithms. This means that, in the general case, we can't do better. Of course, in some special situations, we can design faster algorithms. For example, if we would have to sort an array that contains just zeros and ones, we could iterate through the array, count how many zeros and ones we encounter, and create a new array with the number of counted zeros followed by the number of counted ones. This is just an example, but there are also additional situations where sorting in linear time is possible. If you are interested, we refer to the literature of algorithms such as bucket and radix sort. Note that when we speak about lower bounds, we usually mean the lower bound considering non parallel algorithms. In the second semester you will in facts see some parallel algorithms for sorting. A classical example is bitonic sort, which runs in $\mathcal{O}(\log^2 n)$ under the assumption, that there are infinite processors available.

But why can't comparison based sorting algorithms do better than $\Omega(n \log n)$? The first important observation towards a proof is that every algorithm can be represented as a decision tree. The number of leaves of this decision tree must be equal to the number of possible different sorting sequences. For an array of n number, there can be up to $n!$ sequences if all elements are pairwise different. Hence the decision tree can have $n!$ leaves.

Now we show that a binary tree with n leaves has height at least $\log n$. In order to do this we first prove that the number of leaves in a tree of height h is no more than 2^h leaves by induction on h.

- **h=0** The tree consists of a single node and hence has 2^0 leaves as required.

- **Induction hypothesis:** assume all trees of height h has $\leq 2^h$ leaves.
- **Induction step:** we have to show that trees of height $h+1$ have $\leq 2^(h+1)$ leaves. Consider the left and right subtrees of the root. These are trees of height no more than h, one less than the height of the whole tree. Therefore, by induction hypothesis, each has at most 2^h leaves. Since the total number of leaves is just the sum of the numbers of leaves of the subtrees of the root, we have $n = 2^h + 2^h = 2^(h+1)$, as required. This proves the claim.

Now that we have proven that a tree with height h has $\leq 2^h$ leaves, we prove that a tree with n leaves has height $\geq \log n$. Assume that there exist a tree with n leaves and height $\alpha < \log n$. By the previous lemma we now that the tree can't have more than 2^α leaves. Since we assumed $\alpha < \log n$ we have that this tree has less than n elements, which is a contradiction. Hence we have proven that a tree with n leaves has height $\geq \log n$.

Back to the original problem: we have a tree with $n!$ leaves and we know that the height of the decision tree is equal to the number of comparisons done by the algorithm. Since the tree has $n!$ leaves, it must have height $\geq \log(n!)$. We have $\log(n!) \leq \log(n^n) \leq n \log n$, hence the height of the tree must be greater equal than $n \log n$. Hence we have that $\Omega(n \log n)$ is a lower bound for comparisons based algorithms.

Chapter 3

Data Structures

This course is called *Algorithmen and Datenstrukturen*. In Chapter 2, we have discussed searching and sorting algorithms, a classical example of introductory classes that is useful to show what algorithms are. In a nutshell, algorithms are well-defined procedures that transform an input into an output according to the specification of an ”interesting” problem. In Chapter 1, we briefly mentioned that data structures are strictly connected to algorithms and that they are used to represent data in a convenient way. In this chapter we want to dive deeper into the concept of data structure. The first, crucial observation, is that it is useful to have a clear separation between the concept of abstract data type (ADT) and the concept of data structure. We have the following definitions.

- ADTs are analogous to an interface in Java. Every ADT declares the operations that will be correctly implemented by data structures.
- Data Structures are concrete implementations of the operations declared by ADT. The goal is to implement those operations as efficiently as possible.

ADTs and Data Structures are useful to achieve modularity. In facts, when we describe an algorithm, we can mention the use of a given ADT and use its operations as blackbox. Moreover, if we know a data structure that guarantees that the operations of the ADT are implemented with a certain runtime, we can assume that behaviour when we analyse the asymptotic complexity of the algorithm. This is particularly convenient, because we can have a pool of data structures and, when we have to face a design decision in the creative processing of coming up for an algorithm that solves a given problem, we can look for a convenient data structures that implements the required operations in an efficient manner. In this chapter we will take a look to some of the most important data structures that have proven themselves to be very useful in many situations.

3.1 Basic data structures

3.1.1 Arrays

Arrays are the first very simple data structure you have learned in the course *Introduction to programming*. Arrays can be imagined as a sequence of adjacent cells in memory, where all cells save elements of the same type. For example, an array of length 24 of type int, can be imagined as a list of 24 adjacent cells in memory, where each cell contains an integer number. Strictly speaking, this is an approximative view of what an array really is, but this goes beyond the scope of this course. All basic operations on array (read/ writing an element at a given offset) require $\mathcal{O}(1)$ time. Declaring an array of size n requires $\mathcal{O}(n)$ (both in space and in time).

3.1.2 Lists

A list is somehow similar to an array (basically an array does the same things as a list), but there are some slight differences:

- In an array two adjacent elements (e.g. the element at offset 0 and the element at offset 1) are also adjacent in memory.
- Adding an element in a list requires constant time (simply some work with pointers), while in an array linear time (because we have to declare another longer array).

3.1.3 Stack

The stack is a very important ADT (which is used for example in DFS, a graph algorithm that will be presented in Chapter 4). It can be easily implemented with a list and it supports three basic operations, which all take $\mathcal{O}(1)$:

- Push(x, S): pushes element x in top of the stack S.
- Pop(S): removes the element on top of the stack and returns it.
- Top(S): returns the element on top of the stack without returning it.

The stack is a LIFO data structure: this means that the last element pushed in the stack is the one that will be popped. An analogy would be the stack of trays at the *Polymensa*: the last tray is the first that will be picked again.

3.1.4 Queue

Another important basic data structure (used for example in BFS, another graph algorithm that we will study in Chapter 4) is the queue. It is analogous to a stack (i.e. all basic operations are analogous to the one of the stack and they also can be implemented with lists in constant time). The difference is that the queue is not a LIFO data structure, it is known as a FIFO data structure: the last element who enters the queue will be the last to be taken out. An analogy would be the queue at the *Polymensa*: the first who arrives will be the first to eat.

3.1.5 Union-Find

The union-find is an abstract data type which must implement the following operations:

- Make(v): add isolated node v.
- Union(u,v): put nodes u and v in the same component.
- Same(u,v): decide whether nodes u and v are in the same component or not.

We can implement it as follows: we save an array with the length of the total number of nodes and in $rep\,[v]$ we save the unique node which represents the component of v. In order to do make we initialize $rep\,[v] = v$ for all nodes. This operation requires linear time in the number of nodes. In order to check whether u and v are in the same component we check in constant time whether the unique nodes which represent u and v are the same or not. A naive way to implement the union would be to iterate through all nodes in the data structures and, for all nodes k with $rep\,[k] = u$, we set $rep\,[k] = v$. This would take linear time. A better way would be to keep a list of the nodes k for which $rep\,[k] = u$ so that, instead of iterating through all nodes in the data structure, we just need to iterate through the elements in this list. This trick (known as compression trick) is useful in order to get a more efficient version of Kruskal's algorithm, as we will see in Chapter 4.

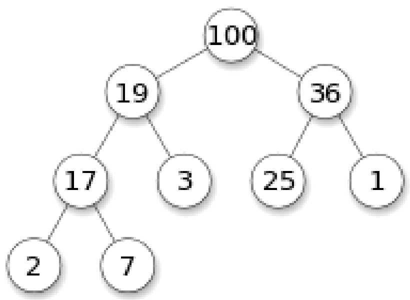
3.2 Priority Queue and Heaps

We have seen the Queue ADT, where we have (analogously to the queue at the *Polymensa*) a *first come first served* principle. Now we see a more particular kind of queue. Imagine that at the *Polymensa* each student arrives with a given priority (based for example on how much time he has for lunch). If a student arrives with better priority than all others student in the queue he will jump in the front and he will be the first to be served. Oppositely, if a student has a lot of time and arrives early in the queue, he might wait very long if the students that come afterwards have a better priority than him. This is exactly the idea behind the priority queue ADT. This ADT implements the following operations:

- Insert(P, x): inserts the elements x in the priority queue P.

- Maximum(P): returns the element with maximum priority in the queue P.
- ExtractMax(P): removes and returns the element of P with largest priority.
- IncreaseKey(S, x, k): increases the priority of element x to the value k.

Before we present how to implement such an ADT, we present a particularly convenient data structure: the max-heap. We begin with an example. An heap (although it is saved in an array) can be visualized as a tree:



The following statements should help you to fully understand the important properties of an heap:

- The height of a heap with n elements is $\mathcal{O}(\log n)$.
- All children (not only the direct ones, but all the children until the leafs) of n are smaller than the value of n. Intuitively: all keys that are "below" a given node have a smaller key than that node.
- Although heaps can be visualized as trees, they are stored in a normal array. To build the array from the tree we read the tree level by level from left to right. In this case we have: 100, 19, 36, 17, 3, 25, 1, 2, 7. More formally, we have:
  - Parent(i) = $\lfloor \frac{i}{2} \rfloor$
  - Left(i) = 2i
  - Right(i) = 2i + 1

  From this we can summarize the max heap property by: $A[parent(i)] \geq A[i]$ (the min heap property has $\leq$ instead of $\geq$).

Now we take a look at two very important functions in order to use heaps: Max-Heapify and Build-Max-Heap.

In order to maintain the max-heap property, we call MaxHeapify(A, i) with an array A and an index i as inputs. When we call this method, we assume that the trees at the left and at the right of i are max heaps, but the key $A[i]$ might be smaller than one of its children. After we call the procedure MaxHeapify we want the tree rooted at i to be a max heap.

Algorithm 10: MaxHeapify(A, i)

```
l ← Left(i) ;
r ← Right(i) ;
largest ← Argmax(A[l], A[r], A[i]) ;
if largest> i then
    Swap(A[i], A[largest]);
    MaxHeapify(A, largest);
end
```

The idea is the following: since the trees at the left of i and at the right of i are max heaps, the direct children of i are maxima of their sub-trees. This implies the element at index i (in order to satisfy the heap property), must contain the maximum between i, the element left of i and the element right of i. If i is the maximum, then we already have a max heap, otherwise we have to swap $A[i]$ with $A[max(left(i), right(i)]$ and to perform the procedure MaxHeapify again from $max(left(i), right(i)$. What about the complexity of this procedure? We know that heaps have logarithmic height and the single call to the procedure (without recursive calls) is constant. Hence we have a complexity of $\mathcal{O}(\log n)$.

What do we have to do if we have an array which does not satisfy the max heap property at all? We have to perform the MaxHeapify procedure starting from the bottom level and then iteratively upwards until we get to the top. Since the leafs trivially satisfy the heap property, we can start from the bottom level of inner nodes. This gives us the following pseudo-code:

Algorithm 11: BuildMaxHeap(A)

```
for $i$ = $\lfloor$ A.length/2 $\rfloor$, ..., 1 do
    MaxHeapify(A, i);
end
```

Note that we started from half of the length of the array because all elements which are after that index are leafs. Although at first sight you may think that the complexity of Build-Max-Heap is $\mathcal{O}(n \log n)$, it is actually only $\mathcal{O}(n)$. We don't go into details now because we use this procedure only in the context of HeapSort (not for priority queues) and hence we don't care if it is $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$. After all, in order to get the $\mathcal{O}(n \log n)$ complexity of heap sort, we don't require linear time for this operation.

Now that we have learned heaps we can implement the procedures of the priority Queue ADT. In order to do that, we can instantiate an array which we know it is long enough and we save a variable *heapLength* which tells us until which index of the array we have elements of the heap. This trick is useful, because if we want to add an element to the array we don't have to create a new one, but we simply change the value of this pointer. Getting the maximum simply means reading $A[1]$. so this goes in constant time. Let's take a look at the other three procedures.

ExtractMax We print the first element of the array, then we swap it with an arbitrary leaf and we restore the heap condition. After we extracted the max we have to reduce the heapLength by one.

Algorithm 12: ExtractMax(A)

```
max ← A[1];
Swap(A[1], A[heapLength];
MaxHeapify(A, 1);
```

This operation runs in $\mathcal{O}(\log n)$. Note that the implementation of MaxHeapify before has to be slightly be modified to support the heapLength pointer (the previous implementation considers that the heap goes from 1 to A.length instead that from 1 to heapLength).

IncreaseKey To implement this procedure we first increment the value of the node, than we go iteratively upwards in the tree and we exchange the node with the parent if the key of the node is greater than the one of the parent.

Algorithm 13: IncreaseKey(A, i, key)

```
$A[i]$ ← key;
while $i > 1$ AND $A[$Parent(i)$] < A[i]$ do
    Swap(A[i], A[Parent(i)]);
    i ← Parent(i);
end
```

This procedure also has complexity $\mathcal{O}(\log n)$.

Insert To insert a key in the heap we can increase the heap size by one, set the value at $A[heapLength]$ to minus infinity and then use the IncreaseKey procedure we have implemented. This also goes in $\mathcal{O}(\log(n))$

Algorithm 14: Insert(A, x)

```
heapLength ← heapLength + 1;
$A[heapLength] \leftarrow -\infty$ ;
IncreaseKey(A, heapLength, x);
```

3.3 Data Structures for Dictionary

In this section we aim to implement an efficient data structure for an ADT that performs the following operations:

- Search(x, V): is the element x in the data structure V?
- Insert(x, V): insert the element x in the data structure V.
- Remove(x, V): delete the element x from the data structure V.

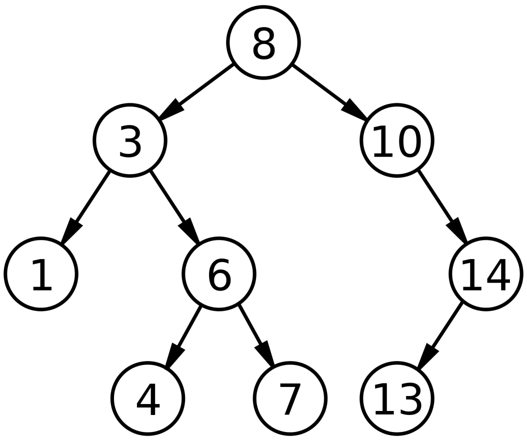
We are going to see several possible solutions, but the best data is called **AVL tree** and performs every operation in $\mathcal{O}(\log n)$. Of course, our portfolio of options includes simple data structures such as arrays (that perform the three operations in linear time if unsorted and support a logarithmic search if sorted), lists (where the runtime for the operations is strongly related on the implementation, but at least linear time is required for searching) and heaps (that also require linear time for all operations). In this section we are going to study two more interesting options: binary search trees and AVL trees, an improved version of binary search trees.

3.3.1 Binary search tree

Similarly as a simple unsorted array, binary search trees require $\mathcal{O}(n)$ for all three operations in the worst-case. Although this does not look encouraging, we need to study them carefully. The reason is not only that we need them to understand AVL trees, but also because they are a prominent data structure in computer science.

Let's take a look at the fundamental properties of binary search trees:

- In binary search trees we have the following properties. Given a node n with key k we have:
  - If l is the left child of n, the key of l is $\leq k$ and the keys of all children of l are $\leq k$.



  - If r is the right child of n, the key of r is $\geq k$ and the keys of all children of r are $\leq k$.

- 8 is the root of the tree. It has a left child with key 3 and a right child with key 10.
- The node n with key 10 has a right child with key 14 and has no left child (in a programming oriented way of seeing the problem, the leftNode of n is NULL).
- The node with key 13 is a Leaf. This means that both its leftNode and rightNode are NULL.

You can think at a NODE as an object in Java with a key attribute (of type double), a leftChild attribute (of type Node) and a rightChild attribute (also of type Node). When a node does not have a left child (or a right child), it simply stores NULL in the attribute. For example a leaf with key 13 will have two NULL attributes.

We now take a look at how to implement the three dictionary operations.

Search When we search for a value v in a tree starting from node n (at the beginning of the search n is equal to the root) we have three cases:

- The value of the key of n is equal to v. In this case we return *true*.
- The value of the key of n is greater than v. We know that a node with value v might be on the left of n, but not the right. For this reason we call our SEARCH method recursively on the left child of n.
- The value of the key of n is less than v. We know that a node with value v might be on the right of n, but not on the left. For this reason we call out SEARCH method recursively on the right child of n.

After this observation we can implement a Search routine. To search a value v on a binary search tree T we call Search(v, root(T)).

Algorithm 15: Search(v, N)

```
if $N = NULL$ then
    return false;
end
if $N.key < v$ then
    Search(v, N.left);
end
if $N.key = v$ then
    return true;
end
if $N.key > v$ then
    Search(v, N.right);
end
```

A single call to the search method goes in $\mathcal{O}(1)$. Since we call the method maximally h times (where h is the height of the tree), the asymptotic complexity of the Search method is $\mathcal{O}(h)$.

Insert The first key observation in order to implement the Insert operation is that, when we insert a node M in a tree T, M will be a leaf in the final tree T'. Insert is very similar to search (because we have to insert the new node in the right place in order to maintain the binary search tree property). Basically the implementation is equal to the one of the Search function, we just have to insert the value before calling the method on a NULL node. Concretely we have this implementation (where we assume that we do not insert a node with a key which is already in the tree). Note that to insert a node M in a tree T we have to call Insert(M, N):

The complexity of this implementation is $\mathcal{O}(h)$ for the same reasons we stated for Search.

Delete We have three possible cases about the node we want to delete:

- The node n is a leaf (i.e. both left and right children are NULL). In this case we just have to set the pointer of the parent to n to NULL (i.e. if n is a left child we set the leftChild attribute of its parent to NULL, otherwise we set the rightChild attribute of its parent to NULL).
- The node n is not a leaf, but either its left child or its right child is NULL. We call c the (only) child of n. In case n is the left child of its

Algorithm 16: Insert(M, N)

```
if $N.key < v$ AND $N.left \neq NULL$ then
    INSERT(M, N.LEFT);
end
if $N.key < v$ AND $N.left = NULL$ then
    N.left = M;
end
if $N.key > v$ AND $N.right \neq NULL$ then
    INSERT(M, N.RIGHT);
end
if $N.key > v$ AND $N.right = NULL$ then
    N.right = M;
end
```

parent, we set the left child of the parent of n to c. In case n is the right child of its parent, we set the right child of the parent of n to c.

- The node n in an inner node (i.e. both its left and right child are not NULL). In this case we have to replace the key of n with the key of its symmetric successor (i.e. ”go once right and then all the way left until we find a leaf”) or of its symmetric predecessor (i.e. ”go once left and then all the way right until we find a leaf”). The symmetric predecessor and successor of n are the only nodes which contain a key than can substitute the key of n without breaking the binary search tree property.

We use the previous observations to produce the following pseudo code for the DELETE method. To delete a value v from a root T we have to call DELETE(V, ROOT(T)). We don't provide an implementation of the method SYMMETRICSUCCESSOR(N): this should be simple as you just have to iteratively go left on N.right until you find NULL.

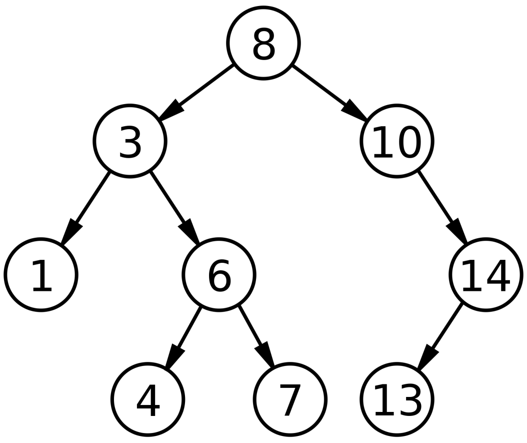
The asymptotic complexity of DELETE is also $\mathcal{O}(h)$.

We have seen that all dictionary operations on a binary search tree have complexity $\mathcal{O}(h)$. This is a problem because, if the tree consists of a list of strictly increasing (or decreasing) values inserted in ascending (or descending) order, the tree degenerates to a list. In this case the height is equal to the number of nodes and hence we get complexity $\mathcal{O}(n)$ for all three operations (which is bad, this is exactly the same complexity we have with unsorted arrays, the most naive data structure for dictionaries).

Before we move to a solution to this issue we explain the three standard ways to ”read” a tree: pre-order, post-order and in-order.

Algorithm 17: Delete(v, N)

```
if $N = NULL$ then
    return NULL;
end
if $v < N.key$ then
    N.left = DELETE(v, N.LEFT);
end
if $v > N.key$ then
    N.right = DELETE(v, N.RIGHT);
end
if $N.left = NULL$ then
    return N.right;
end
if $N.right = NULL$ then
    return N.left;
end
if $N.left \neq NULL$ AND $N.right \neq NULL$ then
    N.key = SYMMETRICSUCCESSOR(N).KEY;
    DELETE(N.KEY, N.RIGHT);
    return N;
end
```



- In order simply means: read the elements of the tree in ascending order. In the case of the figure we have 1, 3, 4, 6, 7, 8, 13, 14.
- Pre-order means: read the root first, then the left part of the tree in order and then the right part of the tree in order. In the example of the figure we get: 8, 3, 1, 6, 4, 7, 10, 14, 13.
- Post-order means: read (recursively) the left part in the post order

Algorithm 18: InOrder(N)

```
if $N.left \neq NULL$ then
    InOrder(N.left);
end
Print(N.key);
if $N.right \neq NULL$ then
    InOrder(N.right);
end
```

Algorithm 19: PreOrder(N)

```
Print(N.key);
if $N.left \neq NULL$ then
    PreOrder(N.left);
end
if $N.right \neq NULL$ then
    PreOrder(N.right);
end
```

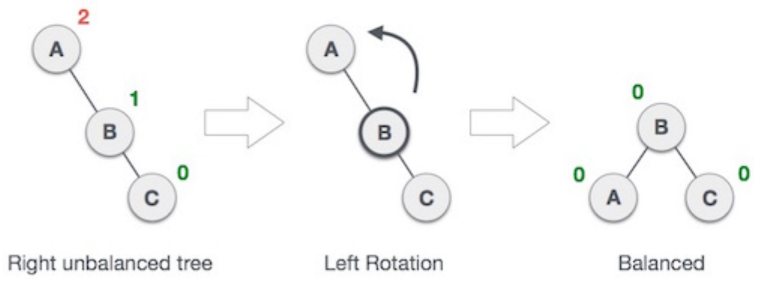
manner, then do the same with the right part and finally read the value of the root. In the example of the figure we get: 1, 4, 7, 6, 3, 13, 14, 10, 8.

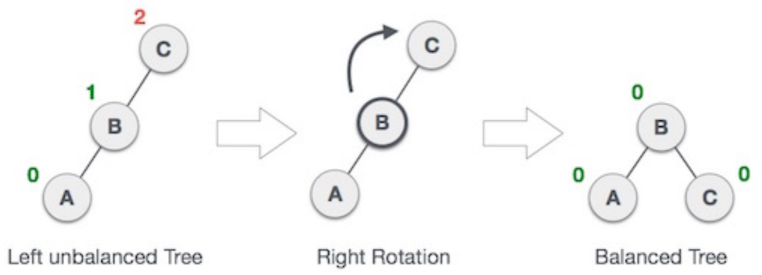
Algorithm 20: PostOrder(N)

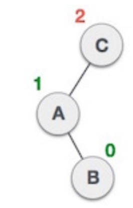
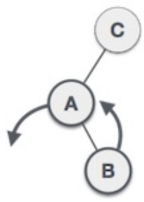
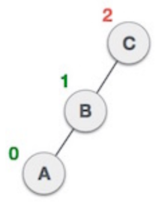
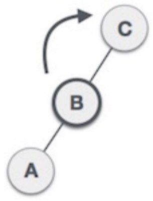
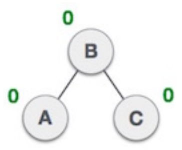
```
if $N.left \neq NULL$ then
    PostOrder(N.left);
end
if $N.right \neq NULL$ then
    PostOrder(N.right);
end
Print(N.key);
```

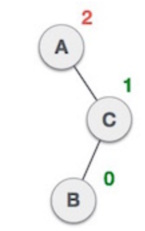
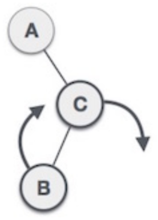
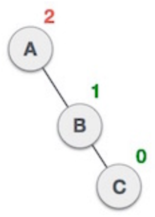
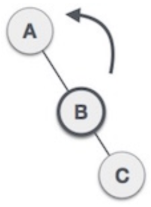
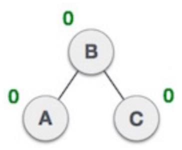
3.3.2 AVL tree

AVL trees are the most efficient dictionary data structure we study. They are very similar to binary search trees, but they have worst-case complexity $\mathcal{O}(\log n)$ for all operations. How is this possible? For binary search trees we saw that the complexity is $\mathcal{O}(h)$. AVL trees are balanced binary search trees or, in other words, we have $h \in \mathcal{O}(\log n)$. How can we achieve that? We say that AVL trees respect the AVL property, i.e. for every node in the tree the height of the left and right sub-tree differ by at most one. In order to prove that if the AVL condition is respected we have logarithmic height (in asymptotic notation) one could proceed by induction and show that the height of a binary search tree that respects the AVL property is $\leq 1.44 \log n$. We omit the details. It is more interesting to discuss how to maintain the AVL condition when inserting a new node in the tree. For every node, we save its balance, i.e. the difference of the height of its right sub-tree and its left sub-tree. We then call a function Upin that, if required, updates the content of the balance of each node. We observe that the balance of a node can change with the insertion of a new key in the AVL tree only if it is on the direct path from the root to the new node, and hence an upper bound for the number of calls of the Upin function is $\mathcal{O}(\log n)$ by induction hypothesis. But what happens if, when we call the Upin function, we break the AVL property on a given node? The key is to use rotations to repair it and to have a new AVL tree containing the new key. All rotations simply require to change some pointers, and hence can be performed in constant time. The following figures depict both how rotations are performed and in which case a certain rotation should be picked.





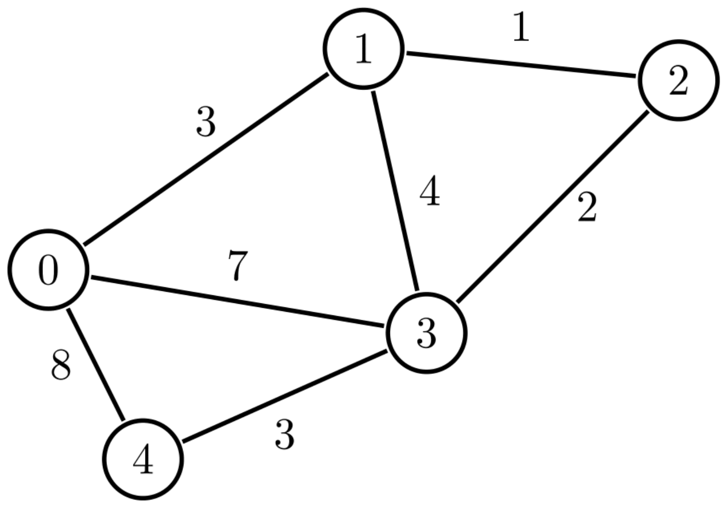
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Chapter 4

Graph Theory

We define a graph, denoted $G = (V, E)$, as an abstract data type composed by n nodes and m edges, where each edge links two nodes belonging to the nodes set. We can visualize a graph as follows:



4.1 Data Structures for Graphs

As we have seen in the previous chapter, we need data structures to concretely instantiate a graph. In this chapter, we present three canonical solutions: adjacency matrices, adjacency lists and a list of edges.

4.1.1 Adjacency matrix

We can represent a graph $G = (V, E)$ as a matrix with $|V|$ rows and $|V|$ columns. The entry (i, j) has value 0 if there is not edge between the nodes

i-j and has value w if there is an edge between nodes i and jJ with weight w (in the case of unweighted graphs, all edges have weight 1). In the case of the previous image we would have the following adjacency matrix:

$$\begin{bmatrix} 0 & 3 & 0 & 7 & 8 \\ 3 & 0 & 1 & 4 & 0 \\ 0 & 1 & 0 & 2 & 0 \\ 7 & 4 & 2 & 0 & 3 \\ 8 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Note that if the graph is undirected (i.e whenever we have an edge between i and j, we also have an edge between j and i), the matrix is symmetric, i.e. $A^T = A$.

As the reader might expect, we need to implement some operations on a data structure that represents a graph. Let's take a look at the asymptotic efficiency of some common operations on adjacency matrices.

- **Memory space:** $\mathcal{O}(|V|^2)$
- **Add vertex:** $\mathcal{O}(|V|^2)$
- **Remove vertex:** $\mathcal{O}(|V|^2)$
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(1)$
- **Are u and v adjacent?** $\mathcal{O}(1)$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(|V|)$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(|V|)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(|V|)$

4.1.2 Adjacency list

A major drawback of adjacency matrices is that they waste a lot of space in the case of sparse graphs, i.e. when $m << n^2$. A natural solution to avoid this waste is to store only the edges that are present in the graph as a list of lists. Of course, in the case of a complete graph, an adjacency list degenerates to an adjacency matrix. Concretely, an adjacency list provides a list for every node. The list of node u contains all v and w such that $(u, v, w) \in E$. In the case of the previous image we would have the following adjacency list:

$$\{[(0,1,3),(0,3,7),(0,4,8)],[(1,0,3),(1,3,4),(1,2,1)],[(2,1,1),(2,2,3)],$$
$$[(3,2,2),(3,1,4),(3,0,7),(3,3,4)],[(4,0,8),(4,3,3)]\}$$

Similarly as we did for adjacency matrices, we not look at the runtime of some common operations on adjacency lists.

- **Memory space:** $\mathcal{O}(|V| + |E|)$
- **Add vertex:** $\mathcal{O}(1)$
- **Remove vertex:** $\mathcal{O}(|E|)$. You can do it in $\mathcal{O}(1)$ if you implement it in a clever way, think about it and if you don't find the solution you can ask me ;)
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(|V|)$
- **Are u and v adjacent?** $\mathcal{O}(\min(deg(u), deg(v)))$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(deg(v))$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(1)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(deg(v))$

We observe a general trend: the efficiency of most operations is inversely proportional to the number of edges. In general, the choice between adjacency lists and adjacency matrices is critical, and we will see examples later on of algorithms that are slower on adjacency matrices than in adjacency lists.

4.1.3 List of edges

A third data structure for graphs is the list of edges. Here, we simply do a list of $|E|$ tuples (u, v, w), where u is the source node, v the destination node and w the weight of the edge. In the case of the previous image we would have the following list:

$$[(0,1,3),(0,3,7),(0,4,8),(1,0,3),(1,2,1),(1,3,4),(2,3,2),\\ (2,1,1),(3,2,2),(3,1,4),(3,0,7),(3,4,3),(4,3,3),(4,0,8)]$$

Note that if the graph is undirected we can either put directed edges in the list (as in the example) or undirected edges (i.e. instead of putting $(0,1,3)$ and $(1,0,3)$ we put only one of the two possibilities with the convention that we consider the graph undirected). Pay attention to the fact that the list does not have to be sorted in any way! Now we take a closer look of the asymptotic efficiency of some common operations on list of edges (in those cases the facts whether we put $|E|$ or $2|E|$ edges in the graph is irrelevant).

- **Memory space:** $\mathcal{O}(|E|)$
- **Add vertex:** $\mathcal{O}(1)$

- **Remove vertex:** $\mathcal{O}(|E|)$.
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(|E|)$
- **Are u and v adjacent?** $\mathcal{O}(|E|)$
- **Given a node v, find** $deg(v)$**:** $\mathcal{O}(|E|)$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(|E|)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(|E|)$

4.2 Searching on Graphs

Now that we introduced the concept of graph and three famous data structures to represent them, we can finally see some first examples of graph algorithms. Depth-first-search (DFS) and Breadth-first-search (BFS) are two very useful examples. Given a graph G, they give an answer to the question: *starting from node u, which nodes of the graph can I reach?* As we will see later, although the two algorithms solve the same problem, they are both necessary since they allow us to solve some more advanced problems.

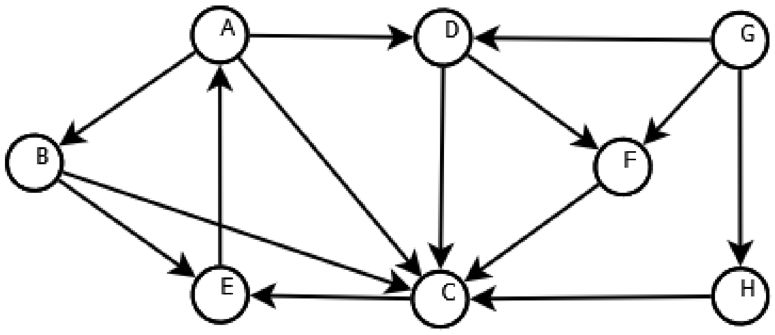
4.2.1 DFS

DFS exploits the idea of *going deep* in the graph. We start from a node and we follow an (arbitrary) path (although usually there is some convention, for example we do it in alphabetical order). After the path is finished, we go one step back and we start a recursive DFS from there. When we have visited every possible node, the search is over. With the help of an array that keeps track of all visited nodes in the graph, at the end of the search we will be able to discover which nodes are reachable from the starting node and which nodes are unreachable in the case of a disconnected graph.

We want to discuss four aspects of DFS: a concrete example; the pseudocode of the iterative version; an informal proof of the correctness and its runtime.

We introduce the example of DFS on the following graph:

We start DFS from node A and we visit nodes in alphabetical order. It is convenient to write the result in red. We start from node A. Starting from A, we can reach B, C and D in one step. Since we decided to go in alphabetical order the second node we visit is B. Now we go on from B and the next node (who wins the race against E) is C. Since C has not outgoing edges we can not continue the search from C, hence we go one step back (and hence we go back to node B) and we do DFS from B. B has two edges: one to C (which we have already visited) and one to E (which we haven't visited yet).



This means that we visit node E and we continue the DFS from there. E has only an outgoing edge to A (which we have visited since it is the starting point of this DFS). This means that we have to go one step back to B. All nodes reachable in one step from B have been visited, hence we do another step back to A. A has outgoing edges to B (already visited), to C (already visited) and to D (not visited yet). Hence we visit D and we continue the DFS from there. D can reach in one step nodes C (already visited) and F. We visit F and we continue from there. F can only reach C in a single step and, since C has already been visited, we go one step back to D. D can reach C and F (both already visited) so we go another step back to A. A can reach B, C and D in a single step and, since they all have been already visited, the DFS is over. Note that we have not visited G and H: this happened because in facts it is impossible to reach them!

Now we go on with our program and we propose a pseudocode for the algorithm. Note that the stack data structure is a very useful tool that comes at rescue to manage the nodes.

Algorithm 21: DFS(G, v)

```
S ← ∅ ;
Push(v, S) ;
while $S \neq \emptyset$ do
    w ← Pop(S);
    if w not yet visited then
        Visit w;
        for each (w, d)∈ E in the reverse order do;
        if d not yet visited then
            Push(d, S);
```

Before we discuss the runtime of DFS, we briefly discuss its correctness. Our goal is to show that if a node is reachable from the starting point, it will in fact be visited. We can argue that the algorithm terminates if and only if the stack is empty. Every node which goes into the stack will be marked as visited (because when the algorithm terminates the stack is empty and when we pop a node from the stack we mark it as visited). All elements that go into the stack are reachable from the starting point (because we use edges from a node in the stack to it and hence we have an inductive argument [1]) and if an element is not visited it means that is not reachable from the starting point. To proof this assume that there is a unique node k which is reachable from the starting point but never gets visited. If it is reachable from the starting point than there is another node t such that there is a path from the starting point to t and $(t, k) \in E$. However when t is visited, by the definition of the algorithm, we will push to the stack all nodes reachable from t and, since k is reachable from t, it will be pushed into the stack and hence visited.

The asymptotic complexity of the algorithm is $\Theta(|V| + |E|)$ if we use an adjacency list (and $\mathcal{O}(|V|^2)$ if we use an adjacency matrix). To see this consider that when we pop a node from the stack we do some work with it if and only if it was not yet visited. This can happen $|V|$ times (since we push at most $|E|$ elements in the stack, the pop operation will be executed at most $|E|$ times. For every node v we do $deg(v)$ operations. Hence we get:

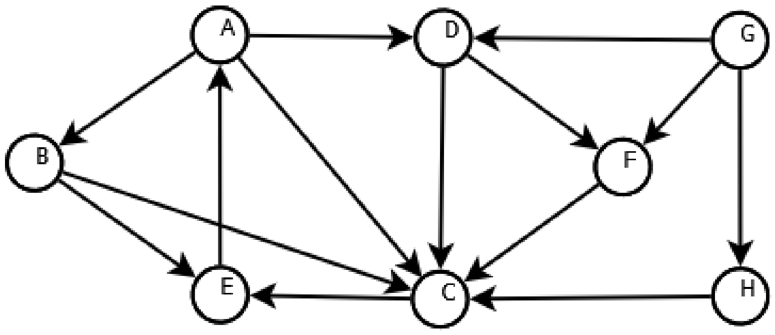
$$\begin{aligned}\Theta(\sum_{i=1}^{|V|}(1 + deg(v_i)) + |E|) &= \Theta(|V| + \sum_{i=1}^{|V|} deg(v_i) + |E|)\\ &= \Theta(|V| + 2|E| + |E|)\\ &= \Theta(|V| + |E|)\end{aligned}$$

4.2.2 BFS

In BFS, instead of going deep in the graph, we go in breadth. This means that we start from node v and we visit all nodes reachable from v. We save all those nodes in a queue and then we pick an arbitrary one from this queue and we visit all nodes reachable from there (and we add them at the end of the queue). Then we pick the second node reachable from v, we visit all its neighbours and we add them at the end of the queue. When all neighbours from v are de-queued we go one layer deeper and we continue in the same way.

We discuss the same aspects of DFS also for BFS.

[1] Recall that the first step is pushing the starting node in the stack and this would be the base case of a more formal induction proof.



We start from node A and we use the convention of alphabetical order to break ties. We visit all the neighbours of A (hence B, C and D) and we save them in a queue. Then we pick the first element of the queue which, by the alphabetical order convention, is B. We visit all neighbours of B (in this case only E since C was already visited) and we add E to the queue. Then we dequeue C and we do nothing (C has no edges going out). Afterwards we dequeue D and we visit its neighbour F (again, C has already been visited). We add F to the queue. At this point the queue contains F and E and since, both elements have no edges to elements which have not been visited, we dequeue them and we are over. The order is different, but the result is in principle the same we obtained with DFS (i.e. we see with both algorithms that G and H are unreachable from A). In general DFS and BFS are equivalent when we want an answer to the question *which nodes are reachable from u?*, but in other situations we might want to use a variant of either DFS or BFS to answer some other algorithmic questions.

As we see from the following pseudocode, the key data structure to implement BFS is the queue.

Algorithm 22: BFS(G, v)

```
Q ← ∅;
Mark v as active;
ENQUEUE(v, Q);
while $Q \neq \emptyset$ do
    w ← DEQUEUE(Q);
    Mark w as visited;
    for each (w, d)∈ E do if d not yet visited and not active then
        Mark d as active;
        ENQUEUE(D, Q);
```

The informal proof for the correctness is completely analogous to the one used for DFS.

The runtime is, again, $\Theta(|V| + |E|)$. We see that each node will be enqueued at most once and for each node v we do $deg(v)$ elementary operations. Hence the runtime is:

$$\Theta(\sum_{i=1}^{|V|}(1 + deg(v_i))) = \Theta(|V| + E|)$$

4.3 Topological Sort

Imagine that you have to solve n tasks $T_1, T_2, \dots, T_n$. However there are some constraints regarding the order in which the tasks have to be solved. That is, there are some constraints like *task T_i must be solved before T_j*. In a first instance, it is important to see that is not always possible to find a solution. Consider for example the following constraints:

- Solve T_1 before T_2
- Solve T_2 before T_1

In this case is not possible to find an order which satisfies both constraints. The problem of determining the existence of an order that satisfy all constraints can be modelled with a graph $G = (V, E)$ as follows: we create a node v_i for every task T_i (with $1 \leq i \leq n$) and for every constraint *solve T_i before T_j* we add an edge from node v_i to node v_j. If the graph contains a bijective function $ord : V \to 1, \dots, |V|$ such that $ord(i) < ord(j)$ for every $(i, j) \in E$, then we can solve the task. The function *ord* is called topological sort of the graph G. Here we present an algorithm to find a topological sort in a given graph (if a topological sort exists). A first important observation is the following:

$$\exists \text{ topological sort} \Leftrightarrow \neg\exists \text{ directed cycle in the graph}$$

Before we discuss an implementation of the algorithm it is worth discussing how to run topological sorting on paper with a small graph.

1. Find a node v that has no outgoing edges (i.e. $deg_{out}(v) = 0$).
2. Put v at **the end** of the topological sorting order.
3. Remove v and all its ingoing edges from the graph.
4. Go on recursively on the rest of the graph.

Of course, by a symmetry argument, you can also do the following:

1. Find a node v that has no ingoing edges (i.e. $deg_{in}(v) = 0$).
2. Put v at **the beginning** of the topological sorting order.
3. Remove v and all its outgoing edges from the graph.
4. Go on recursively on the rest of the graph.

Algorithm 23: TopologicalSort(G)

```
S ← $\emptyset$ ;
Save In-degree of every node in an array $A$ of length $|V|$ ;
for each $v \in V$ do if $A[v] = 0$ then
    Push(v, S);
    i ← 0;
while $S$ not empty do
    v ← Pop(S);
    $ord[v]$ ← i; i← i+1;
    for each $(v, w) \in E$ do A[w]← A[w]-1;
    if $A[w]=0$ then
        Push(w,S);
if $i = |V| + 1$ then
    return ord;
return cycle;
```

We observe that the stack contains all nodes v such that, in a given stage of the algorithm[2], have $deg_{in}(v) = 0$. At every iteration of the while loop we remove an element v from the stack (which has in-degree zero) and we put v as next element of our topological sorted sequence. Then, for all edges (v, w), we remove one from the in degree of w: this is equivalent to delete v from the graph and all edges that starts from v. The algorithm ends when the stack is empty and this happens if and only if we have considered all elements which could have been pushed in the stack (i.e. all nodes which, at a certain point, have in-degree 0). If the number of elements which has been stored in the stack is equal to $|V|$ then we have calculated a topological sorting, otherwise not.

The asymptotical complexity of the algorihtm in the pseudocode is $\Theta(|V| + |E|)$. In facts we see that:

- Step 2 takes $\Theta(\max(|V|, |E|))$.
- Steps 3-4 take $\Theta(|V|)$.

[2]Recall that, although we don't delete anything explicitly, we delete some nodes and edges when we decide the position in the topological sorting of an element.

- The while loop can be executed at most $|V|$ times.
- In the worst case (i.e. when we have a topological sort of the graph) the for loop inside the while loop is executed $|E|$ times **in total**.

With this observation we can conclude that the runtime of our algorithm to find a topological sort in a graph is $\Theta(|V| + |E|)$.

4.4 Shortest Path

In this section we consider a weighted graph $G = (V, E)$, i.e. every edge $e \in E$ has a cost $c(e)$. The problem of interest answers to queries where we give as input a starting vertex s, an ending vertex t, and we want to find the shortest path $W = (s, v_1, \ldots, v_k, t)$ in G, where all nodes in the path are elements of V and all transitions are elements of E. In particular, among all possible paths between s and t, we want to find the one that minimizes the sum of the costs associated to the edges in the path. We note with $d(s, t)$ the cost of the shortest path from s to t. In this section, we assume that G has no negative cycles, because in this case a shortest path would not exists (in facts, every tour along this cycle would give a shorter distance and hence the distance would tend to minus infinity). We also assume that the distance from a vertex to itself is zero.

At this point we mention a useful property of the shortest path problem that one can easily show by contradiction. Shortest paths satisfy the following optimality principle: if $(v_0, v_1, \ldots, v_{l-1}, v_l)$ is a shortest path in G, then also $(v_0, v_1, \ldots, v_{l-1})$ is a shortest path in G. We know that if a problem exhibits an optimality principle, then it can be solved with dynamic programming. In facts one can write the following recurrence:

$$d(s, v) = \min_{(u,v) \in E} d(s, u) + c(u, v)$$

The problem of this approach is that the order to fill the table is not always clear. However in some cases, as we will see, this is possible and we will take advantage of this. The shortest path problem has a peculiarity: there are multiple algorithms with different runtimes. Some algorithms are faster but less general, other are a bit slower but they work on more instances of the input graph. For this reason it is important to define the context where we want to solve a shortest path problem and pick the best algorithm accordingly. The following table gives an overview of the different possibilities that we have at hand.

Situation	Algorithm	Runtime
One-to-all, all edges with the same weight	Modified BFS	$\mathcal{O}(\|V\| + \|E\|)$
One-to-all, directed acyclic graph with arbitrary weights	DP and TopoSort	$\mathcal{O}(\|V\| + \|E\|)$
One-to-all, only with non-negative weights	Dijkstra	$\mathcal{O}(\|V\| \log \|V\| + \|E\|)$
One-to-all, also with negative weights	Bellman-Ford	$\mathcal{O}(\|V\|\|E\|)$
All-to-all	Floyd-Warshall	$\mathcal{O}(\|V\|^3)$
All-to-all	Johnson	$\mathcal{O}(\|V\|\|E\| + \|V\|^2 \log \|V\|)$

4.4.1 Modified BFS

If all edges have the same weight, we observe that the shortest path tree and the BFS tree are the same. In facts, both trees always visit adjacent vertices and if the weights are uniform the notions of adjacent and nearest are the same. Hence we can come up with the following algorithm with runtime $\mathcal{O}(|V| + |E|)$.

Algorithm 24: SP-BFS(G, s)

```
$d[s] \leftarrow 0$;
$d[v] \leftarrow \infty$ for $v \in V \setminus \{s\}$;
$Q \leftarrow \emptyset$;
Enqueue(s,Q);
while $Q \neq \emptyset$ do
    $u \leftarrow$Dequeue(Q);
    for each $(u, v) \in E$ do
    if $d[v] = \infty$ then
        $d[v] \leftarrow d[u] + c(u, v)$;
        Enqueue(v,Q);
```

4.4.2 DP and TopoSort

Now we move to an algorithm that works for directed acyclic graphs. The idea is to using the recurrence:

$$d(s,v) = \min_{(u,v)\in E} d(s,u) + c(u,v)$$

and filling the table in topological order. Concretely:

Algorithm 25: SP-DAG(G, s)

```
$d[s] \leftarrow 0$;
$d[v] \leftarrow \infty$ for $v \in V \setminus \{s\}$;
for $v \in V$ in topological order do
    for all $(v,w) \in E$ if $d[w] > d[v] + c(v,w)$ then
        $d[w] = d[v] + c(v,w)$;
```

We have already seen an algorithm that computes the topological sort in $\mathcal{O}(|V| + |E|)$. Afterwards, the for loop accesses all edges from a starting point v (and to do this for all vertices $v \in V$). If the graph is given with an adjacency matrix, we have a runtime of $\mathcal{O}(|V|^2)$, if we have an adjacency list, we get $\mathcal{O}(|V| + |E|)$.

4.4.3 Dijkstra

The modified BFS does not work if we have non uniform weights because it could happen that in the BFS we have a direct edge from a node u to a node v which is not the shortest. In facts, it is possible that going from u to another node u' and then to v is shorter than taking the direct way. The algorithm of Dijkstra is a generalized search that takes this issue into account. Concretely, we use an array d of length $|V|$ such that for all nodes $v \in V$ it always holds that $d[v]$ is greater equal than the minimum distance from the starting node s to v. Moreover, the algorithm saves a set S of the nodes for that we know that the correspondent entrance in d is equal to the shortest path from s to it. At the beginning of the algorithm we set $d[s] = 0$, $S = \{s\}$ and $d[v] = \infty$ for all nodes different from s. At the end of the algorithm we want to get $S = V$, because this would mean that d contains the right answer for every node. Dijkstra's algorithm does some computation at every iteration in order to include an additional node to S. Concretely we know that for all $v \in V \setminus S$ an upper bound of the final answer is given by:

$$d[v] = \min_{\substack{(u,v)\in E \\ u \in S}} d[u] + c(u,v)$$

Moreover let $v^* \in V \setminus S$ be the node such that $d[v^*]$ is minimal. We have that $d[v^*]$ is the shortest distance from s to v^*. This can be seen because every path W from s to v^* must leave S. For this reason we get:

$$c(W) \geq \min_{\substack{(u,v)\in E \\ u\in S \\ v\in S\setminus V}} d[u] + c(u,v) = d[v^*]$$

This means that every path from s to v^* has cost greater equal than $d[v^*]$ and hence this value is optimal.

By considering all these observation, we need to compute the following at every iteration:

$$d[v] = \min_{\substack{(u,v)\in E \\ u\in S \\ v\in V\setminus S}} d[u] + c(u,v)$$

and we include the minimal value of $d[v^*]$ such that $v^* \in V \setminus S$ into S. After $|V|$ iteration we have solved the task.

Algorithm 26: Dijkstra(G, s)

```
for each $v \in V \setminus \{s\}$ do $d[v] = \infty$;
$d[s] = 0$;
$V \setminus S \leftarrow V$;
while $V \setminus S \neq \emptyset$ do
    choose $u \in V \setminus S$ with minimal $d[u]$;
    $V \setminus S - \{u\}$;
    for $(u,v) \in E$ do
        if $d[v] > d[u] + c(u,v)$ then
            $d[v] = d[u] + c(u,v)$;
```

Now we analyse the runtime. We see that it depends on what data structure we use to extract the minimum element in the set $V \setminus S$. In general, we need a data structure to manage $V \setminus S$. In general, this data structure must support three operations: an ADD (at the beginning we add all nodes to the data structure), an EXTRACT-MIN (at every iteration of the while loop we extract the node with minimum distance from s) and a DECREASE-KEY (for the relaxation). Provided that we use an adjacency list he total runtime is given by:

$$\mathcal{O}(|V| \cdot \textsc{Add} + |V| \cdot \textsc{Extract-Min} + |E| \cdot \textsc{Decrease-Key})$$

Where we did an amortised analysis for the second nested loop (in facts, in total we take a look at every edge exactly once). Depending on the data structure we choose we get:

- **Heaps:** $\mathcal{O}((|V| + |E|) \log |V|)$
- **Fibonacci Heaps:** $\mathcal{O}(|V| \log |V| + |E|)$

4.4.4 Bellman-Ford

We know that negative edges might be a problem because we could have negative cycles. Before we design an algorithm that also addresses this type of issue, we first present the algorithm of Ford. The idea is the following: if we take an $(u, v) \in E$ with $d[v] > d[u] + c(u, v)$ then we relax the edges, i.e. we do $d[v] \leftarrow d[u] + c(u, v)$. The algorithm of Ford simply take a look at all edges and does a relaxation. If, after an iteration through all edges, no edge has been relaxed, then the algorithm terminates. This algorithm is correct when it terminates, but one can show that this algorithm terminates if and only if the graph has no negative cycles. In order to address this issue we use the fact that a graph contains a negative cycle if and only if after $|V| - 1$ there is still an edge that can be relaxed. We get the following algorithm:

Algorithm 27: Bellman-Ford(G, s)

```
for each $v \in V \setminus \{s\}$ do $d[v] = \infty$;
$d[s] = 0$;
for $i \leftarrow 1, 2, \ldots, |V| - 1$ do
    for each $(u, v) \in E$ do
    if $d[v] > d[u] + c(u, v)$ then
        $d[v] \leftarrow d[u] + c(u, v)$
for each $(u, v) \in E$ do
if $d[v] > d[u] + c(u, v)$ then
    return Negative cycle;
```

It is easy to see that the runtime of the algorithm is given by the nested loops: one iterates through every vertex and one through every edge. Hence the runtime of the Belman-Ford algorithm is $\mathcal{O}(|V||E|)$.

The algorithms we have seen so far are one-to-all algorithms, i.e. given a starting node s they calculate the shortest path from s to all other nodes in the graph. But what if we want to calculate the shortest path between all possible pair of nodes in the graph? A naive solution would be to repeat the one-to-all algorithms for all possible starting nodes. This is correct but it might not be efficient. The next two algorithms are a solution to this problem.

4.4.5 Floyd-Warshall

This algorithm uses the idea of dynamic programming. First, it gives a unique ID from 1 to $|V|$ to every node. Then it uses a three dimensional table where the entry $d(u,v,i)$ indicates the length of the shortest path from u to v which uses as intermediate node only the nodes with ID from 1 to i. Of course, at the end, the length of the shortest path from u to v will be in $d(u,v,|V|)$. At the beginning, we assign for all nodes $d(u,u,0) = 0$ and, for all edges $(u,v) \in E$, $d(u,v,0) = c(u,v)$. In order to update the value of an entry $d(u,v,i)$, we keep the minimum between $d(u,v,i-1)$ and $d(u,i,i-1) + d(i,v,i-1)$. We fill the table for ascending i (and then we consider of pairs of nodes). This gives us the following algorithm:

Algorithm 28: Floyd Warshall(G)

```
for $1 \leq u \leq |V|$ and $1 \leq v \leq |V|$ do
    $d(u,v,0) = \infty$;
for $u \in V$ do
    $d(u,u,0) = 0$ ;
for $(u,v) \in E$ do
    $d(u,v,0) = c(u,v)$;
for $i = 1, \ldots, |V|$ do
    for $u = 1, \ldots, |V|$ do
        for $v = 1, \ldots, |V|$ do
            $d(u,v,i) = \min(d(u,v,i-1), d(u,i,i-1) + d(i,v,i-1)$;
return $d$;
```

At it's easy to observe, the runtime of this algorithm is $\mathcal{O}(|V|^3)$.

4.4.6 Johnson

In principle, if we have a graph with only positive edges, we could run $|V|$ times Dijkstra's algorithm and get a better runtime than Floyd-Warshall's algorithm (provided that we don't have too many edges, in the limit case the runtimes are equivalent). However, we have a problem when we have negative edges. Johnson algorithm uses a trick to transform a given graph G into a graph G' with only positive weights and the same shortest paths between nodes such that the problem of the all-to-all shortest path can be solved (potentially) faster by applying $|V|$ times Dijkstra. The idea is to add a new node v_0 into the graph and then we add an edge with weight zero from v_0 to all other nodes. Then we define the new weights as: $c(u, v) + h(v) - h(u)$ where $h(v)$ and $h(u)$ are the length of the shortest path from v_0 to v and u respectively. We note that the shortest paths in the graph with the new weights are the same because the length of the shortest path from u to v in G' is equal to the one in G if we neglect a constant $h(v) - h(u)$ which is the same for all shortest paths from u to v. Moreover the new costs are non negative. Hence Johnson's algorithm provides the following steps:

1. Introduce v_0 in the graph and an edge with weight zero to all other nodes.
2. Use Bellman-Ford to calculate $h(v)$ for all $v \in V$.
3. Calculate the new weights using the function h.
4. Run $|V|$ times Dijkstra.
5. Use h to modify the distances found with Dijkstra.

If we use Fibonacci Heaps to implement Dijkstra we come up with a runtime of $\mathcal{O}(|E||V| + |V|^2 \log |V|)$ which can be better or equal than the one found with Floyd-Warshall.

4.5 MST

In this section we study another interesting graph problem algorithm, determining the minimum spanning tree (MST) in a graph. The formal specification of the problem is the following: given a connected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}$, we want to find a tree $T = (V, E')$ with $E' \subset E$ such that $\sum_{e \in E'} c(e)$ is minimized.

Before we give some concrete algorithms we give two useful *rules*:

- **Blue rule:** given a set $S \subset V$ with $\emptyset \neq S \neq V$ such that no edge in $E(S, V \setminus S)$ is blue, color an edge $e' \in E(S, V \setminus S)$ with $c(e') = \min\{c(e) | e \in E(S, V \setminus S)\}$ in blue.
- **Red rule:** given a circle C in G such that no edge in C is red, color in red an edge $e' \in C$ with $c(e') = \max\{c(e) | e \in C\}$.

One can show that if in a graph neither the blue nor the red rule is applicable, then all edges are colored and the blue edges are a minimum spanning tree.

With this idea in mind we present three different algorithms.

4.5.1 Boruvka's Algorithm

This algorithm begins with $|V|$ connected components S_i ($1 \leq i \leq |V|$) and contracts them together until a single connected component remains. While doing this contractions, we need to save in the set T the edges which are in the MST. Concretely:

Algorithm 29: Boruvka(G)

```
$T \leftarrow \emptyset$;
**while** $T$ *is not a spanning tree* **do**
    $(S_1, \dots, S_k) \leftarrow$ connected components of $T$;
    $(e_1, \dots, e_k) \leftarrow$minimum edges that leave $S_1, \dots, S_k$;
    $T \leftarrow T \cup (e_1, \dots, e_k)$;
```

Every iteration of the while loop takes $\mathcal{O}(|V| + |E|)$. In facts, we just need to iterate once through the adjacency list. Moreover, at every iteration, every connected component is contracted with another one. Since we start with $|V|$ connected components and we end with only one, we do $\mathcal{O}(\log |V|)$ iterations. Hence the total runtime of this algorithm is $\mathcal{O}((|V| + |E|) \log |V|)$.

4.5.2 Prim's Algorithm

Now we see an algorithm that applies the blue rule until convergence. It starts from an arbitrary node v and sets $S = \{v\}$. Then, it take a look from all edges that start in S and end in $V \setminus S$ and add the edge with minimum cost to the MST. Moreover it add the end of this edge into the set S. The algorithm goes on until S contains every node in the graph. In order to implement this we use the following algorithm:

Algorithm 30: Prim(G)

```
$S \leftarrow \emptyset$;
$Q$ with all nodes with priority $\infty$;
DECREASE-KEY(Q,s,0);
**while** $S \neq V$ **do**
    $u \leftarrow$ EXTRACT-MIN(Q);
    $S \cup \{u\}$;
    **for** $(u, v) \in E$, $v \notin S$ **do**
        DECREASE-KEY(Q,v,c(u,v);
```

The runtime of this algorithm is:

$$\mathcal{O}\left(|V| \cdot \textsc{Insert} + |V| \cdot \textsc{Extract-Min} + |E| \cdot \textsc{Decrease-key}\right)$$

We observe that we can chose multiple data structures to implement the operations required by the ADT we used in the algorithm. And this ADTs can have different costs. A good choice is using min-heaps, in order to get a runtime of $\mathcal{O}((|V| + |E|) \log |V|)$.

4.5.3 Kruskal's algorithm

The idea of this greedy algorithm is the following: we sort the edges with respect to their weights. Then we make a union find data structure where every node is a separated connected component. We iterate through every edge and, if the edge connects two different connected components, we add the edge to the MST and we union the connected components of the two edges.

Algorithm 31: Kruskal(G)

```
Sort the edges with respect to their weights. Let $e_1, \ldots, e_{|E|}$ be the
  sorted sequence (in ascending order). ;
Create a union-find data structure and create a separated
  component for every node.;
$T \leftarrow \emptyset$ ;
for i=1 to $|E|$ do
    Let $e_i = (u, v)$;
    if no Same(u,v) then
        $T \leftarrow T + e_i$;
        Union(u,v);
```

If we implement the union routine with the clever trick that we studied in Chapter 3, an amortized analysis gives us a total runtime of $\mathcal{O}(|V| \log |V|)$ for the union find operations. By combining this result with the sorting phase we get a total runtime of $\mathcal{O}(|E| \log |E| + |V| \log |V|)$.

Quiz

1. The idea of the proof for the lower bound of sorting is based on:
   (a) Looking at the minimum height of a binary search tree containing at least a certain number of nodes.
   (b) Looking at the minimum height of a binary search tree containing at least a certain number of leaves.
2. The runtime of Quick Sort is $\mathcal{O}(n \log n)$
   (a) True
   (b) False
3. For which data structure is the path compression technique useful?
   (a) AVL trees
   (b) Union-Find data structure
   (c) Min-max heaps
   (d) Multistack
4. We can efficiently find longest paths in graphs in some cases, but in general the problem is difficult and no polynomial algorithm is known.
   (a) True
   (b) False
5. What is a practical data structure that can be used to implement Dijkstra's and Prim's algorithms?
   (a) Queue
   (b) Priority Queue
   (c) Union-Find data structure

   (d) Stack

6. We can modify the binary search algorithms from the lecture such that it works on linked-lists.
   (a) True
   (b) False

7. We have seen an algorithm to decide whether a graph is bipartite. We can easily modify this algorithm to decide whether a graph is tripartite.
   (a) True
   (b) False

8. Which sorting algorithm is similar to how humans usually sort a deck of cards?
   (a) Bubble Sort
   (b) Insertion Sort
   (c) Heap Sort
   (d) Merge Sort

9. We can adapt Kruskal's algorithm to compute a maximum spanning tree.
   (a) True
   (b) False

10. The main advantage of dynamic programming over raw recursion is:
    (a) Avoiding unnecessary computation in the case of overlapping sub-problems.
    (b) Allowing the program to efficiently jump from the leaf of the recursion tree to its root.
    (c) Avoiding the program to efficiently jump from the root of the recursion tree to a corresponding leaf.
    (d) Saving memory in the case of a very deep recursion tree, but only if you use a bottom-up approach.

11. Quick Sort has quadratic runtime if and only if we always pick the greatest/ least element from the part of the array that still has to be sorted.
    (a) True

   (b) False

12. For the NP-complete problems it holds that...
    (a) We don't know a lower bound
    (b) We don't know an upper bound
    (c) Neither
13. Which of the following are data structures to store graphs?
    (a) Adjacency matrix
    (b) Adjacency heap
    (c) List of edges
14. Bubble Sort is slower than Merge Sort...
    (a) Never
    (b) In the best-case
    (c) In the worst-case
15. A Stack is known as a...
    (a) FIFO data structure
    (b) LIFO data structure
16. How many edges are contained in a tree with n nodes?
    (a) It depends
    (b) n-1
    (c) n
    (d) $\frac{n(n+1)}{2}$
17. AVL trees are a variant of binary search trees where each non-leaf node is exactly balanced.
    (a) True
    (b) False
18. The Theta notation can be used for...
    (a) Upper bounds of algorithms
    (b) Lower bounds of algorithms
    (c) Tight bounds of algorithms
19. Priority queues are usually implemented with...

   (a) Queues
   (b) Heaps
   (c) AVL trees
   (d) Union-Find data structure

20. If we have to find keys in an array it is better to fort sort it such that we can use binary search instead of linear search.
   (a) Always
   (b) Never
   (c) Sometimes

21. A graph can be topologically sorted if and only if it does not contain any cycle
   (a) True
   (b) False

22. Which of the following factors play a role in the performance of a program?
   (a) Asymptotic efficiency of the algorithm
   (b) Constants hidden by the asymptotic notation
   (c) Cache performance
   (d) Memory usage of the program

23. The most efficient implementation of Dijkstra uses a priority queue.
   (a) True
   (b) False

24. What are the main ingredients of AVL trees?
   (a) Rotations
   (b) Path compression
   (c) Avoiding overlapping sub-problems
   (d) A special mathematical property similar to the one of the Fibonacci sequence

25. A complete graph can not contain an Euler tour.
   (a) True
   (b) False

26. The best algorithm for the maximum sub-array sum uses the divide-and-conquer paradigm.
    (a) True
    (b) False
27. We can use binary search to speed up insertion sort and obtain an optimal algorithm for sorting.
    (a) True
    (b) False
28. There exist a polynomial algorithm to decide whether a graph contains an Euler tour or not.
    (a) True
    (b) False
29. What is the purpose of exponential search?
    (a) Searching through a collection containing an exponential number of elements
    (b) In the best case, it can speed up binary search
    (c) Helping evolutionary algorithms in properly exploring many regions of the solution space
30. No algorithms to exactly solve the travelling salesman problem is known.
    (a) True
    (b) False
31. As a rule of thumb, how many operations of your Java program can your computer perform each second?
    - 10^4
    - 10^7
    - 10^{15}
    - 10^{100}
32. Which of the following are types of heaps?
    (a) Min-heap
    (b) Max-heap
    (c) Min-max-heap

33. How many nodes can be contained at most in a binary search tree with height h (where the root is considered to be at height zero)?
    (a) 2^h
    (b) $2^h - 1$
    (c) 2^{h-1}
    (d) $2^{h+1} - 1$

Solutions

1. To prove the lower bound of sorting we consider a binary search tree. The height of the binary search tree corresponds to the number of operations that we have to perform and each node is a possible result. Hence we are interested in the height of a binary search tree with n elements and the answer is (a).
2. False, in the worse case Quick Sort has runtime $\mathcal{O}(n^2)$.
3. For Union-Find, it speeds up the FIND operation.
4. True, the longest path problem is NP-complete.
5. The answer is (c), priority queue.
6. False, linked list don't provide a framework for random access.
7. False, determining whether a graph is $k-$partite for $k > 2$ is NP-complete.
8. The answer is (b), Insertion Sort.
9. True, we can just multiply all edges by -1 and run Kruskal in order to compute a maximum spanning tree.
10. (a) and (d) are correct. Note that using memoization does not necessarily save memory because recursion occupy space in memory (but you will discuss this issue in great detail in the coming up semesters).
11. False, also if we pick large/ small elements as pivot most of the time.
12. The answer is (c). A lower bound is always known (e.g. the size of the input), and an upper bound is provided by a known algorithm (e.g. a brute force one).
13. (a) and (c).
14. In the best case. In facts it has a best case of $\Omega(n)$, as opposed to $\Omega(n \log n)$ of Merge Sort.

15. The stack is a LIFO data structures.
16. n - 1
17. False, nodes are almost balanced. This means that the absolute difference between the number of children on the left and the number of children on the right differs by at most one.
18. All of them. The Theta notation indicates that the bound is tight, but it can be used also for upper/lower bound of algorithms. For example, one could say that a problem has a lower bound of $\Theta(n)$, and this means that in the best case the algorithm will perform $c \cdot n$ operations for a constant c.
19. Priority queues are usually implemented with heaps.
20. The answer it depends. The threshold of the sorting-search trade-off is when we have to search $\Theta(\log n)$ keys: when we have to search more than that it is worth sorting, otherwise not.
21. True.
22. All of them.
23. False, with Fibonacci heaps (not discussed in the lecture, but you should know that they exist and which rumtime they provide) provide a better performance.
24. Rotations and also the answer (d), which is the basis of why AVL trees have logarithmic height.
25. True, for example consider two nodes with a single edge.
26. False, there exist a faster algorithm that runs in $\Theta(n)$ that uses the sliding window technique.
27. False, also if we find the position to put the element in logarithmic time, we still have to move all the successive elements in the array.
28. True.
29. Exponential search is used to (hopefully) speed up binary search.
30. False, there is a dynamic programming approach. It has exponential runtime, but the algorithm exists.
31. 10^7
32. All of them.
33. $2^{h+1} - 1$