ETH Zurich

Department of Computer Science

Algorithms and Data Structures

Author:
Soel Micheletti

2019

Preface

This script is a summary for the lecture *Algorithmen und Datenstrukturen* of the department of Computer Science at ETH Zurich. This material is not official and does not cover all the material which is important for the exam (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 exams :)

Contents

Part I

Introduction

Chapter 1

Starting Point

In the famous textbook *Introduction to algorithms* there is 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]

The concept is graphically represented in the following picture:

The *algorithm box* can do various things: doing a mathematical computation, suggesting to the user of a social media the people he might know, indicating to the first year students which bus they should take to go to the ESF...

In this course you will learn several algorithms and you will have to solve real world problems by designing (and implementing in Java) your own algorithms. The following properties are fundamental:

- **Correctness:** given a problem specification, your algorithm has to return the correct answer. Imagine that you have to design an algorithm that sorts a list of numbers. If your algorithm returns 3, 1, 2 when the input is 2, 1, 3, then your algorithm does not solve the task and hence is useless in the context of the initial task. Correctness can be proved (you will do it often in your exercise sheets).
- **Efficiency:** informally, this means that an algorithm should not take too much time to solve the task. During the course you will learn that efficiency can be measured by the number of computational steps required by your algorithm (in a few weeks you will learn the *Big Oh* notation, which asymptotically evaluates the number of operations your algorithm requires 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

[1] Cormen et al., *Introduction to algorithms*, MIT Press, 2009.

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 an algorithm which is more efficient than another one (in the sense of the *Big Oh* notation) is significantly better. In order to be successful in this course you will have not only to learn to design correct algorithm, but also efficient ones (if you design an algorithm which is slightly less efficient than the optimal one, you might lose many points in the exam).

So far so good: you have understood an informal definition of algorithm and you know that algorithms have to be both correct AND efficient in order to be useful. 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 and not restricted to those categories.

- 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 what is an array (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). If you are still confused about the concept of data structure don't worry: at the beginning (and maybe also in the middle) of the course, it's absolutely normal, but it's important that you realise they are present from the beginning of the course!

Chapter 2

Induction

Correctness is a fundamental property of every useful algorithm. In order to prove the correctness of an algorithm there are different techniques, a very important one (which you will use often also in other courses) is the proof by induction.

Suppose you wish 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 domino.

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 a more involved form of induction (called Noetherian induction) which 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.

Let's take a look at a first example from the exam of HS11.

Consider the following recursive equation:

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

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 $4 + 2(5\frac{n}{8}^{\frac{1}{3}} - 4)$.

Chapter 3

Asymptotic Notation

In class you have seen the Karatsuba algorithm for the multiplication of two numbers. You have seen that this algorithm is more efficient than the canonical multiplication algorithm you learned at the primary school because it *performs less elementary operations*. The asymptotic notation you will use for the rest of the course follows exactly the same principle: it chooses 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 counts how many of such operations an algorithm does. However, instead of counting the exact number of operations, we will 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 approximation 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 I present you the formal definitions of the asymptotic notations. A few very useful theorems are presented in the exercise sheet 0 and in the script. Although at the beginning it may seem a very abstract and theoretical concept, with the time you will develop a sixth sense and you will not think at the formal definition anymore.

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).

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

In the exercise sheet 0 and in the script there are some rules that may be very useful (e.g. the limit trick and the formal expression that additive and multiplicative constant do not matter). Do the exercise 0 more than once and take a look at old exams, you will find a lot good exercises to practice the asymptotic notation. In order to be faster I also suggest you to train to isolate the worst term according to the following list:

$$\mathcal{O}(1) \leq \mathcal{O}(\log n) \leq \mathcal{O}(n) \leq \mathcal{O}(n \log n) \leq \mathcal{O}(n^2) \leq \mathcal{O}(2^n) \leq \mathcal{O}(3^n) \leq \mathcal{O}(n!)$$

Chapter 4

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 (then it should reach a base case which is solved non-recursively) and then the problem is solved.

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 (ideally) an infinite number of times 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 I answer to a list of questions you may ask yourself during the semester.

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.

4.1 Star finding algorithm

Describe the problem

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 which everybody else know, but he/ she does not know nobody. For example if Lionel Messi enters the room we all know who he is, but he does not know us.

What are the elementary operations in this problem?

Every Yes/No question is an elementary operation. Other kind of questions are not permitted.

Is there always a star?

No. For example if everyone knows everybody else, then there is no star.

Is it possible that there are two stars?

No. 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 contraddiction to the fact that S_1 is a star. Hence in general we have either no stars or exactly one star.

Describe a naive algorithm to solve the problem

We ask to every person in the room if he knows person A, person B... for every other person in the room. We summarize the results in a table and we search if there are persons which are known by everybody and don't know anyone.

What is the asymptotic complexity of the naive solution?

We have to ask $n-1$ to every person in the room. Hence we have $n(n-1)$ elementary operations and the asymptotic complexity is $\mathcal{O}(n^2)$.

We solve the problem recursively. How is the base case of the problem (i.e. when there are exactly two people in the room)? How many elementary operations are needed?

The base case is when there are only persons A and B in the room. We ask A whether he knows B and we ask B whether he knows A. If A knows 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. We need 2 elementary operations.

If we take two out of the n people in the room, at least one of them is not a star. Is there a question we can ask in order to be sure which of the two people is definitely not a star?

Yes. We could ask to the first person: *Do you know anybody in the room?*. If he say yes, then he is definitely not a star. If he says no, the other person is not a star (because the first person does not know him).

Describe a (clever) recursive solution of the problem

We have n persons in the room. With one elementary operation we can pick a person who is not a star (see question above). 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 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.

Write a recursive formula for the number of operations

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

Guess a closed formula for the recursive formula

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

Prove your claim by induction

- **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.

What is the asymptotic complexity of the algorithm?

$\mathcal{O}(n)$

Chapter 5

Maximum Sub-Array Sum

Given a list $a = a_1 \dots a_n$ of n numbers (positive and negative), find the indices i and j that maximize the following expression:

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

Intuitively, find the indices that maximize the sum of the elements between them.

Describe a naive algorithm

Take all possible indices i and j and do the sum of the elements of the list between them. Remember the maximum and return it.

What is the asymptotic complexity of the algorithm?

The possible choices of the indices are: $n-1$ for index 1, $n-2$ for index 2, ..., 1 for index $n-1$. This is the Gauss formula and we know the asymptotic behaviour is $\mathcal{O}(n^2)$. For all choices of the indices we have to do the sum of the elements between them, which in the worst case takes n operations. Hence the total asymptotic complexity is $\mathcal{O}(n^3)$.

With the approach above we are doing several computations more than once. Which computations?

In the algorithm above we do, for example, 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 would be done in constant time instead of linear time.

How can we improve the first cubic algorithm?

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 $b = 1, 5, 2, 14, 21, 18, 24$.

How can we do the sum of the elements of a between the indices i and j with the help of the array b? What is the asymptotic complexity?

We get:

$$\sum_{k=i}^{j} a_k = b_j - b_i$$

The asymptotic complexity is now $\mathcal{O}(1)$.

How can we improve the first cubic algorithm? What is the new asymptotic complexity?

We try all possible indices i and j (in $\mathcal{O}(n^2)$) and we do the sum of the elements in a between those indices (in constant time). The total asymptotic complexity is $\mathcal{O}(n^2)$.

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. Can you imagine such an algorithm for this problem?

Yes. We split the array a in two parts of 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 $p + q$.

The algorithm is: 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 $p + q$.

What is the asymptotic complexity of this algorithm?

Finding the maximum sum between the two arrays goes in constant time, hence we have:

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

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).

After the lecture you should remember that there is an $\mathcal{O}(n)$ algorithm. Describe it!

We use two variable: 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. In pseudocode we have:

Algorithm 1 MaxSubArraySum(a)

```
1: max ← 0
2: randmax ← 0
3: for i ← 1 to n − 1 do
4:     randmax ← randmax + a_i
5:     if randmax > max then
6:         max ← randmax
7:     if randmax < 0 then
8:         randmax ← 0
```

Is it possible to do asymptotically better than $\mathcal{O}(n)$?

No. In order to solve the problem we need at least to look at every element of the array, hence we can not do asymptotically better. We say that n is a lower bound of the algorithm (i.e. we can not do better) and we write it $\Omega(n)$.

Part II

Data Structures

This course is called *Algorithms and Data Structures*. At this point of the course you should have a very concrete idea of what is an algorithm (you have seen several sorting algorithms, DFS, BFS, ...), but maybe you are still a little bit confused about Data Structures. Examples of Data Structures you already know are arrays, linked lists (see the course *Introduction to Programming*), adjacency matrices and adjacency lists. Now we are going to go a little bit deeper in the topic and in particular it is important to understand the difference between **Abstract Data Types (ADT)** and **Data Structures**.

- ADTs are analogous to an interface in Java. Every ADT has a name and a set of operations it guarantees to perform.
- Data Structures implement ADTs concretely. The goal is to implement the operations guaranteed by the ADT as efficient as possible.

ADTs and Data Structures are useful for modularity. Let's see an example which explains what this last statement means. The pseudo code for DFS is given by:

Algorithm 2 DFS(G, v)

```
1: S ← ∅
2: Push(v, S
3: mentre  S ≠ ∅ fai
4:     w ← Pop(S)
5:     se w not visited allora
6:         Mark w as visited
7:         for each (w, x) ∈ E in reverse order fai
8:             se x not visited allora
9:                 Push(x, S)
```

In the algorithm we write things such as Push(v, S) and Pop(S). Those are operations of the abstract data type Stack which are provided by a concrete implementation of the Stack (for example using the list Data Structure). When we say that ADTs are useful for the modularity of the program we simply mean that we can use the operation in an algorithm without caring about the concrete implementation. Modularity is good in order to achieve abstraction. Usually concrete implementations of ADTs are provided by libraries (e.g. *util* library of Java), but since this is an introductory course you have to implement ADTs by yourself.

Now I present you a list of the ADTs and the Data Structures covered in this course:

- **ADT**: stack, queue, priority queue, graph, ADT for dictionary.
- **Data Structures**: array, list (linked lists), heap, binary search tree, AVL tree, adjacency matrix, adjacency list, union-find data structure.

Chapter 6

Simple Data Structures

6.1 Array

This is the first very simple data structure you have known in the course *Introduction to programming*. You can imagine an array as a sequence of adjacent cells in memory, where each cell contain an element with the same type of the array type. For example if you declared an array of length 24 of type int you can imagine that you have a list of 24 cells in memory and each cell contains an element of type int. This is an approximative view of what an array really is (as you will see in later courses): if you want to know more details come to me, but this abstraction works for 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).

6.2 List

You should have seen lists (single and double linked) in *Introduction to Programming*. A list is somehow similar to an array (basically an array does the same things as a list), but there are some 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).
- Inserting an element between two elements in a list requires constant time, in an array linear time.

When you talked about InsertionSort, you may have thought: *the complexity of the algorithm is* $\mathcal{O}(n \log n)$*, we have to do* n *times a binary search.* This is false because if we want to sort an array we can't simply insert an element in the right place, but we have to copy the whole array in linear time. Hence, if we use insertion sort with binary search on an array we need $\mathcal{O}(n^2)$ time, while on a list $\mathcal{O}(n \log n)$ is enough. This is a first neat example that shows why data structures are important: in order to efficiently solve a task not only a good algorithm is required, but also appropriate data structures!

6.3 Stack

This is a very important data structure (e.g. it is used in DFS). It can be easily implemented with a list and it supports three basic operations, which takes $\mathcal{O}(1)$:

- 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.

6.4 Queue

Another important basic data structure (used for example in BFS) 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 queue is 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.

6.5 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 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 constant 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 the list. This trick (known as compression trick) is useful in order to get a more efficient version of Kruskal's algorithm.

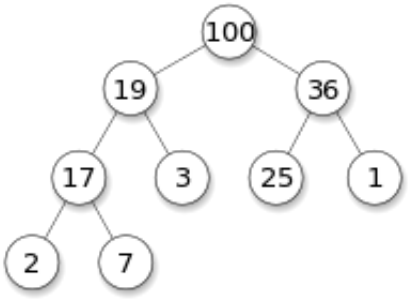
Chapter 7

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 a priority queue ADT. This ADT implements the following functions:

- 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 useful data structure: the max-heap. The data-structure is completely analogous. We begin with an example. An heap (although it is saved in an array) can be seen with a tree structure:



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.

MaxHeapify 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 3 MaxHeapify(A, i)

```
1: l ← LEFT(I)
2: r ← RIGHT(I)
3: largest ← MAX(L, R, I)
4: se largest ≠ i allora
5:     SWAP(A[I], A[LARGEST])
6:     MAXHEAPIFY(A, LARGEST)
```

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 $\mathcal{O}(\log n)$.

Build-Max-Heap(A) 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 4 BuildMaxHeap(A)

```
1: for i = ⌊ A.length/2 ⌋ downto 1
2:     MAXHEAPIFY(A, I)
```

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 $\mathcal{O}(n)$ because the complexity of HeapSort is $\mathcal{O}(n \log n)$ anyway.

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 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 5 ExtractMax(A)

```
1: max ← A[1]
2: SWAP(A[1], A[HEAPLENGTH]
3: 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 6 IncreaseKey(A, i, key)

```
1: A[i] ← key
2: mentre i > 1 AND A[PARENT(I)] < A[i] fai
3:     SWAP(A[I], A[PARENT(I)])
4:     I ← PARENT(I)
```

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 7 Insert(A, x)

```
1: heapLength ← heapLength + 1
2: A[heapLength] ← −∞
3: INCREASEKEY(A, HEAPLENGTH, X)
```

That's it about Priority Queues. You have learned about a very useful ADT and its implementation. This is useful in several situations, we'll see some example later in the course in Graph Theory!

Chapter 8

Data Structures for Dictionary

We want to implement an efficient data structure for an ADT which 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 (which you should learn very well) is called **AVL tree** and performs every operation in $\mathcal{O}(\log n)$.

8.1 Unsorted array

- SEARCH(x, V) goes in $\mathcal{O}(n)$ (you have to do a linear search).
- INSERT(x, V) goes in $\mathcal{O}(n)$ (since arrays are of fixed length you have to instantiate a new array which can contain an additional element and then copy the old array).
- REMOVE(x, V) goes in $\mathcal{O}(n)$ since you first have to search and then create a new shorter array.

8.2 Sorted array

- SEARCH(x, V) goes in $\mathcal{O}(\log n)$ (you have to do a binary search).
- INSERT(x, V) goes in $\mathcal{O}(n)$ (since arrays are of fixed length you have to instantiate a new array which can contain an additional element and then copy the old array).
- REMOVE(x, V) goes in $\mathcal{O}(n)$ since you first have to search and then create a new shorter array ($\mathcal{O}(\log n + n) \in \mathcal{O}(n)$).

8.3 List (sorted or unsorted)

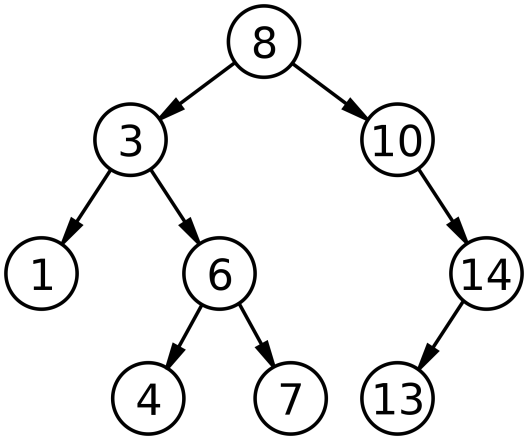
The complexity of the operations depends on the implementation, but you need at least $\mathcal{O}(n)$ for search (since you can not access the element at index i in constant time).

8.4 Heap

Search in a heap goes in $\mathcal{O}(n)$ so this is not the most efficient way to implement our ADT for dictionary.

8.5 Binary search tree

As mentioned, this is not the best data structure (in the worst-case all operations need $\mathcal{O}(n)$). However I suggest you to study binary trees carefully because otherwise you will not understand AVL trees and because there are often questions about them in the exam! Let's take a look at a binary tree.



Now I will explain some concepts about binary trees in a little bit verbose way, but I think that this might be useful for some of you.

- 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 8 Search(v, N)

```
1: se p = NULL allora
2:     return false
3: se p.key < v allora
4:     SEARCH(V, N.LEFT)
5: else if p.key = v then
6:     return true
7: else if p.key > v then
8:     SEARCH(V, N.RIGHT)
```

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):

Algorithm 9 Insert(M, N)

```
1: se p.key < v AND p.left ≠ NULL allora
2:     INSERT(M, N.LEFT)
3: else if p.key < v AND p.left = NULL then
4:     N.left = M
5: else if p.key > v AND p.right ≠ NULLthen
6:     INSERT(M, N.RIGHT)
7: else if p.key > v AND p.right = NULLthen
8:     N.right = M
```

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 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.

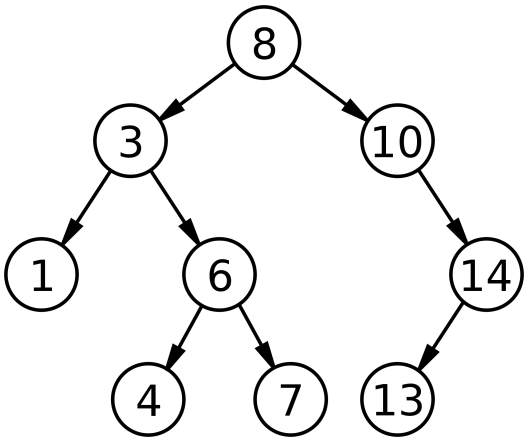
Algorithm 10 Delete(v, N)

```
1: se N = NULL allora
2:     return NULL
3: se v < N.key allora
4:     N.left = DELETE(v, N.LEFT)
5: else if v > N.key then
6:     N.right = DELETE(v, N.RIGHT)
7: else
8:     if N.left = NULL then
9:         return N.right
10:    if N.right = NULL then
11:        return N.left
12:    N.key = SYMMETRICSUCCESSOR(N).KEY
13:    DELETE(N.KEY, N.RIGHT)
14: return N
```

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.



- 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. If you execute InOrder(root(T)) on a tree T you give this ordering:

Algorithm 11 InOrder(N)

```
1: se N.left ≠ NULL allora
2:     InOrder(N.left)
3: Print(N.key)
4: se N.right ≠ NULL allora
5:     InOrder(N.right)
```

- 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. If you execute PreOrder(root(T)) on a tree T you give this ordering:

Algorithm 12 PreOrder(N)

```
1: Print(N.key)
2: se N.left ≠ NULL allora
3:     PreOrder(N.left)
4: se N.right ≠ NULL allora
5:     PreOrder(N.right)
```

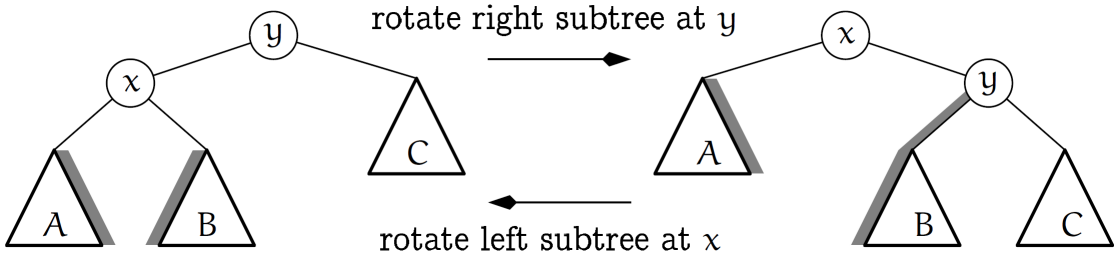
- Post order means: read (recursively) the left part in the post order manner, then do the same with the right part and finally read the value of the root. In the example of the figure we get: 1, 4, 7, 6, 3, 13, 14, 10, 8. If you execute PostOrder(root(T)) on a tree T you give this ordering:

Algorithm 13 PostOrder(N)

```
1: se N.left ≠ NULL allora
2:     PostOrder(N.left)
3: se N.right ≠ NULL allora
4:     PostOrder(N.right)
5: Print(N.key)
```

8.6 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, so that $h \in \mathcal{O}(\log n)$. How can we achieve that? We say that AVL trees respect the AVL condition, i.e. for every node in the tree the height of the left and right subtree differ by at most one. In order to prove that if the AVL condition is respected we have logarithmic height (in asymptotic notation) we proceed by induction: this is what you have done in the lecture where you came up with the fact that the height of an AVL tree is $\leq 1.44 \log n$, but I don't think it is useful to report this very theoretical proof here. It is more interesting to discuss how to maintain the AVL condition when you insert a new node in the tree. The key is to use **rotations** when needed. Particularly, you insert the new node in the same way we have seen with binary search trees and then we check the AVL condition: if the insertion of the new node violates the AVL conditions we perform some rotations such that the tree obtained after the rotations is an AVL tree (i.e. a binary search tree with logarithmic height). The figure below illustrates how to perform rotations:



I recommend you to solve the programming exercise 4 of HS17, it is very useful to understand how rotations really work!

Part III

Searching and Sorting

Chapter 9

Searching

In this chapter we see how to search an element in an array. This problem is very important and comes over and over in practical situations. Moreover we explain the importance of sorting, the problem which we will examine in the following chapters.

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

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

The array given as input can be sorted or not sorted. If the array is sorted we can use more efficient algorithms (intuitively, searching a word in the dictionary is faster than searching a word in the algorithm's book). If the array is not sorted we can look at every element or sort it and then using the faster algorithms. The following table summarizes the algorithms we can pick in the case the array is already sorted and the possible strategies we have when the array is not sorted.

Case	**Algorithms**
Sorted array	Binary Search Interpolation search Exponential search
Unsorted array	Linear search Sort the array and use binary/ interpolation/ exponential search

First we take a look at linear search and binary search separately, then we investigate, in the case we have an unsorted array, which algorithm suits best.

9.1 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* (I suggest you to do it since, probably, you will read the whole book and you will learn a lot of amazing stuffs in between)! The pseudocode of the algorithm is the following:

Algorithm 14 LinearSearch(a[], b)

```
1: for i ← 1 to n do
2:     if a[i] = b then return i
3: return b is not in the array
```

The worst-case complexity is $\mathcal{O}(n)$.

9.2 Binary search

This algorithm works only on sorted array. You shouldn't be surprised that searching in a sorted set is easier than searching in an unsorted set. Imagine 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 15 BinarySearch(a[], min, max, b)

```
1: while min≤max do
2:     mid ← (min+max)/2
3:     if a[mid]=b then return mid
4:     if a[mid]>b then max← mid-1
5:     if a[mid]<b then mid← min+1
6: return b is not in the array
```

The worst-case runtime of the algorithm is:

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

Which gives $\mathcal{O}(\log n)$.

In class you have seen variants of binary search (e.g. interpolation search and exponential search). This algorithms have the same worst-case complexity as binary search but on average they perform less elementary operations (but the asymptotic behaviour is the same). 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, but on arrays is not always the case (e.g. if you have to search the number 53, is it a big number or not?)

9.3 Trade-offs

Before answering this question you should know the cost of sorting an array. Take as a black-box that the best algorithm for sorting has runtime $\mathcal{O}(n \log n)$. Imagine that you have to search k different elements in an array of n elements. We have two possibilities:

- **Linear search:** you can do k times a linear search. In this case you have runtime $k \cdot \mathcal{O}(n) = \mathcal{O}(kn)$.
- **Sorting + binary search:** you can sort the array and then do k times a binary search. In this case you have runtime $\mathcal{O}(n \log n + k \log n)$.

Depending on the value of k, one of the two strategies is better (e.g. if you have to search only one element linear search is faster because you don't have to pay the cost of sorting). 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).

Chapter 10

Basic Sorting Algorithms

In the previous chapter we have seen that sorting can be useful if we have to search several elements in a collection of data. Sorting is useful in this and several other practical situations. In this chapter we see a bunch of sorting algorithms.

10.1 BubbleSort:

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

Algorithm 16 BubbleSort(a)

```
1: for i ← 1 to n − 1 do
2:     for j ← 1 to n − 1 do
3:         if a[i] > a[i + 1] then Swap(a[i], a[i + 1]
```

It is easy to see from the form of the algorithm (two nested for loops which iterate through the array), that the asymptotic complexity is $\mathcal{O}(n^2)$. Note that if during an execution of the inner loop we never swap two 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 $\mathcal{O}(n)$, but in the worst (and also in the average case), the runtime is still quadratic.

10.2 SelectionSort

This is, in my opinion, what one (who does not have expertise in algorithms) would do if he/she has to design a sorting algorithm: you iterate through the array, you take the largest element and you put it at the end of the array. Then you iterate through the array once again (ignoring the last element which is now in the right position), you take the largest element and you put in the right position (before the last element). You repeat this process and you have sorted the array. The pseudocode is the following:

Algorithm 17 SelectionSort(a)

1: **for** k $\leftarrow$ n **down to** 2 **do**
2: Iterate in the first k elements of the array. Let $a[m]$ be the maximum element in this set.
3: SWAP$(a[m], a[k]$

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 $\mathcal{O}(n^2)$.

10.3 Insertion Sort

This algorithm follows the same idea one usually uses to sort a deck of card. It proceeds by keeping a sequence of already sorted elements, and always considering the next element in the sequence. It then inserts this element in the right place between the sorted sequence by shifting all the bigger elements right. This is the pseudocode (note that this is different from the one in the script, but it may help you to implement the algorithm):

Algorithm 18 InsertionSort(a)

1: **for** i $\leftarrow$ 1 **to** n - 1 **do**
2: j $\leftarrow$ i
3: **while** $j > 1$ AND $a[j] < a[j-1]$ **do**
4: SWAP$(a[j],\ a[j-1])$
5: j$\leftarrow$ j-1

The runtime analysis is analogous to the one of SelectionSort.

10.4 HeapSort

We have seen some first sorting algorithms which go in $\mathcal{O}(n^2)$. Can we do better? The answer to this question is yes, and HeapSort is a first example of sorting algorithms which has complexity $\mathcal{O}(n \log n)$. In order to understand this algorithm you have to understand the heap Data Structure (see summary in the Data Structure part). Have you read it? Perfect, now HeapSort will be just a game for you ;)

We can combine heaps and the idea of SelectionSort to obtain an $\mathcal{O}(n \log n)$ algorithm. In facts, if we are able to get the maximum in an unsorted array in $\mathcal{O}(\log n)$ we can take n times the maximum from the unsorted part of the array and then we have completed the sorting task. Recall that heaps provide a BUILDHEAP procedure which transforms an arbitrary array in an array with the heap property in $\mathcal{O}(n)$ and a MAXHEAPIFY which restores in $\mathcal{O}(\log n)$ the heap condition in a tree where left and right sub trees are heaps but the whole tree not necessarily. With this two procedures we can write a recursive version of HeapSort.

Algorithm 19 HeapSort(A)

```
1: BuildHeap(A)
2: heapSize ← A.length
3: for i = A.length downto 2
4: Swap(A[1], A[i])
5: heapSize ← heapSize - 1
6: MaxHeapify(A, 1)
```

We see that the asymptotic complexity is $\mathcal{O}(n + n \log n) \in \mathcal{O}(n \log n)$

Chapter 11

Merge Sort

11.1 Idea

We have seen several examples of divide-and-conquer algorithms. This kind of paradigm suits very well also to sorting. In facts we can do 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. Merging the two sorted parts in a single sorted array.

Obviously we need a base case: this happens when we have 0 or 1 element in the array (in this case we are done).

11.2 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. Remember the magic of recursion: **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: you have a pointer to the first element of the A (now sorted) and another pointer to the first element of B (now sorted). Then you repeat until you have finished the following task:

Which element is smaller? The one indicated by the pointer on A or the one indicated by the pointer of B? You pick the smallest one and you put it in your solution. Then you move to the right the pointer which used to point to the smallest element. You will arrive at a point where one of the two pointer points out of the array (i.e. you have already put all the elements of A or B into your solution). When you get to this point you simply copy the other sub-array in your 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	$\emptyset$
-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.

11.3 Pseudocode

The following is a possible pseudocode for MergeSort. I let it relatively vague on purpose (this gives the idea, but it is quite far from a Java Implementation). Try to code it yourself and ask (Google or me), if you have problems.

Algorithm 20 MergeSort(a)

```
1: if a has length 0 then return a
2: if a has length 1 then return a
3: left ← MergeSort(first half of a)
4: right ← MergeSort(second half of a)
5: leftPointer ← 1
6: rightPointer ← 1
7: S ← ∅
8: while leftPointer < length of left AND rightPointer < length of right do
9:     if left[leftPointer] ≤ right[rightPointer] then
10:        S ← S+left[leftPointer]
11:        leftPointer ← leftPointer + 1
12:
13:    else do          S ← S+right[rightPointer]
14:        rightPointer ← rightPointer + 1
15: Add the array which still has a pointer inside into S
```

11.4 Analysis

We can analyse MergeSort 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 cn operations for an appropriate constant c), we have that the complexity of merging is $\mathcal{O}(n)$. The final runtime will be (under the assumptions that $T(1) = 0$ and that n is a power of two, i.e. $n = 2^k$):

$$T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) = 2(2T(\frac{n}{4}) + \frac{n}{2}) + \mathcal{O}(n) = 4T(\frac{n}{4}) + 2\mathcal{O}(n) = \cdots = kT(1) + k\mathcal{O}(n)$$

From the last step, since $k = \log n$, we deduce that MergeSort 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 MergeSort has a complexity of $\mathcal{O}(n \log n)$, but it needs $\Theta(n)$ extra memory. MergeSort is not bad, but it is possible to do better (i.e. using only a constant ammount of memory): the algorithm which does this is called QuickSort.

Chapter 12

Quick Sort

12.1 Why QuickSort?

MergeSort 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 MergeSort (it follows the divide and conquer paradigm) but it does not have to perform a merge in order to be correct. In fact 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, QuickSort is a solution in this sense. However QuickSort has to face a performance challenge: MergeSort 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 keep the recursion tree balanced. QuickSort 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 $\mathcal{O}(n \log n)$.

12.2 Idea

1. Pick an arbitrary element p of the array (p is called pivot).
2. Create a sub-array A with all the elements $\leq p$ and sort this sub-array recursively.
3. 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].

NB: this in only an intuition. The previous "algorithm" does not provide a base-case and is only a naive "implementation" since we want QuickSort to be in place.

12.3 Execution on paper

Consider the initial array:

5	2	7	-5	16	12	22

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

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

- 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

Than 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

12.4 Implementation tips

In the script you can find a pseudo-code for QuickSort which works only on array with different elements (i.e. there can not be two entries with the same value). I strongly recommend you to read it and try to implement it yourself. Here I present you an alternative implementation which works as well with duplicate elements (and is also in place).

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).

Algorithm 21 QuickSort(a, left, right)

```
1:  leftPointer ← left
2:  rightPointer ← right
3:  pivot ← a[right]
4:  mentre leftPointer ≤ rightPointer fai
5:      mentre a[leftPointer] < pivot fai
6:          leftPointer ← leftPointer + 1
7:      mentre a[rightPointer] > pivot fai
8:          rightPointer ← rightPointer - 1
9:      se leftPointer ≤ rightPointer allora
10:         Swap(a[leftPointer], a[rightPointer])
11:         leftPointer ← leftPointer + 1
12:         rightPointer ← rightPointer - 1
13: se left < rightPointer allora
14:     QuickSort(a, left, rightPointer)
15: se leftPointer < right allora
16:     QuickSort(a, leftPointer, right)
```

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.

12.5 Analysis

QuickSort is correct for an arbitrary chose of the pivot p. The performance, however, strongly depends on the chose of p. QuickSort 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 subtrees 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 QuickSort on an array of $n-1$ element we obtain (consider $T(1) = 0$):

$$T(n) = T(n-1) + \mathcal{O}(n) = T(n-2) + 2\mathcal{O}(n) = \cdots = T(1) + n\mathcal{O}(n) \in \mathcal{O}(n^2)$$

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 $T(1) = 0$ and that n is a power of two, i.e. $n = 2^k$.

$$T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n) = 2(2T(\frac{n}{4}) + \frac{n}{2}) + \mathcal{O}(n) = 4T(\frac{n}{4}) + 2\mathcal{O}(n) = \cdots = kT(1) + k\mathcal{O}(n)$$

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

- QuickSort has an average complexity of $\mathcal{O}(n \log n)$ and, as you will see in the next semester, if you chose the pivot p uniformly at random in the array you have to sort, the performance of QuickSort very good (in praxis randomized QuickSort is considered better than MergeSort). The case where the complexity is quadratic is very rare (because you have to choose an "unlucky" pivot many times).
- MergeSort needs $\Theta(n)$ extra space in order to merge the two sorted sub-arrays. QuickSort is in place and this is a desirable property.

Intuitively we see that QuickSort 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. When we execute QuickSort we do $\mathcal{O}(n)$ 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. If in the programming assignments you have to sort something you can chose whether to implement HeapSort, MergeSort or QuickSort (altough I suggest you MergeSort since it is probably the easiest to implement).

Chapter 13

Lower Bound

We have seen three sorting algorithms with complexity $\mathcal{O}(n \log n)$. The question is: can we do better? The short answer is no. The precise answer is: comparison based algorithms (i.e. all algorithms that are relevant for this lecture) can't do better that $\Omega(n \log n)$, but there are some specialised sorting algorithms that can sort in linear time (e.g. counting sort, radix sort, bucket sort). However, all this algorithms require special assumptions about the sequence to be sort. When we speak about lower bound we usually mean the lower bound considering non parallel algorithms. In the second semester you will in facts see some parallel algorithms for sorting and a famous 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 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 contraddiction. 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.

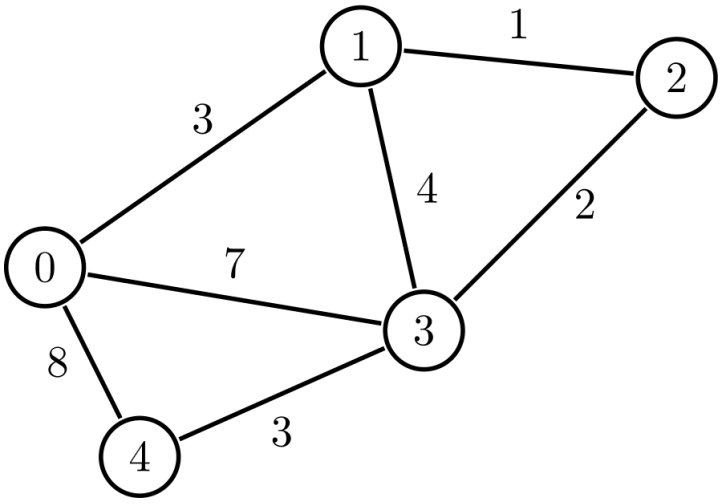
Part IV

Graph Theory

Chapter 14

Data Structures for Graphs

Until now a graph $G = (V, E)$ is an abstract data type which can be represented as follows:



But how can you represent a graph in a computer? How can you code algorithms that works with graphs? In order to do this you need a **data structure** which represent the graph. Here there are the three most popular solutions.

14.1 Adjacency matrix

We simply do a matrix with $|V|$ rows and $|V|$ columns. The entry (i, j) has value 0 if there is not edge between nodes i and 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 the matrix is symmetric, i.e. $A^T = A$. Now we take a closer look of 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:** $\mathcal{O}(|V|)$

14.2 Adjacency list

We do a list of lists. The list has 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:

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

Let's take a closer look of the asymptotic efficiency 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:** $\mathcal{O}(deg(v))$

Obviously, if you have a complete graph (i.e. every node is connected to all other nodes), the runtimes of many operations becomes the same of the ones for adjacency matrices. The intuition behind adjacency matrices is that the efficiency of many operations is inversely proportional to the number of edges (which, you should remind, is $\leq |V|^2$ in the graphs you consider in this

course): if the graph has zero edges the operations are very efficient, if the graph has many edges the operations have the same efficiency they would have in adjacency matrices. As you'll see during the semester, some algorithms on graphs have different complexity if we use adjacency matrices or adjacency lists (e.g. Dijkstra and Prim's algorithms).

14.3 List of edges

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:** $\mathcal{O}(|E|)$

Chapter 15

DFS+BFS

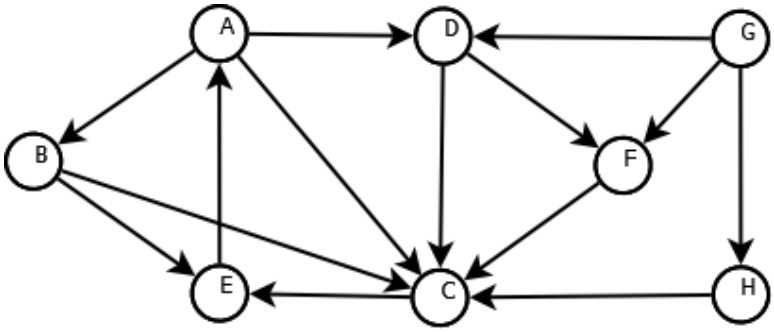
Depth-first-search (DFS) and Breadth-first-search (BFS) are two crucial algorithms in graph theory. Given a graph they answer to the question: *starting from node u, which nodes of the graph can I reach?*

15.1 DFS

The idea is *going deep* in the graph. You start from a node and you follow an (arbitrary) path (although usually there is some convention, for example do it in alphabetical order). After the path is finished, you go one step back and you start DFS from there. When you visited every possible node the search is over. If you keep an array with all the nodes in the graph, at the end of the search you will see which nodes are reachable from the starting node (the one that have been visited) and which nodes are unreachable [1].

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

Let's begin with an example based on the following graph:



We start DFS from node A and we visit nodes in alphabetical order. I write the result in red. We start from node A and 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

[1] Since you are at the beginning of your experience with graphs remember that not every graph is connected and this is a clear example where some nodes are unreachable!

the search from C, but we have to 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!

The key data structure to implement an iterative DFS is the stack. The following is a pseudocode for DFS:

Algorithm 22 DFS(G, v)

```
1: S ← ∅
2: Push(v, S)
3: while S ≠ ∅ do
4:     w ← Pop(S)
5:     if w not yet visited then
6:         Visit w
7:         for each (w, d)∈ E in the reverse order do
8:             if d not yet visited then
9:                 Push(d, S)
```

Before we discuss the runtime we briefly discuss the correctness (in the sense 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 [2]) 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:

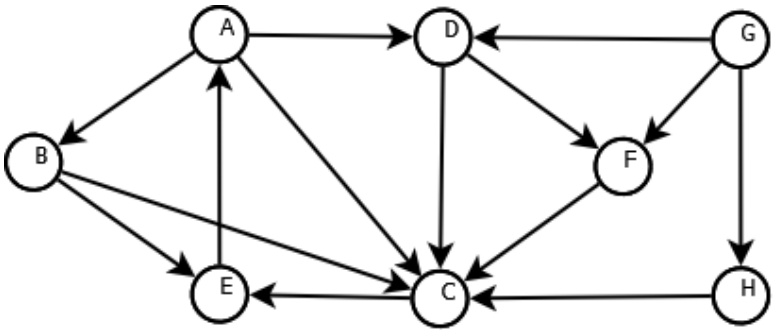
[2] Recall that the first step is pushing the starting node in the stack and this would be the base case of a more formal induction proof.

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

15.2 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 dequeued we go one layer deeper and we continue in the same way.

We discuss the same aspects of DFS also for BFS.



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 questions.

The key data structure to implement BFS is the queue. The pseudocode is:

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:

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

Algorithm 23 BFS(G, v)

```
1: Q ← ∅
2: Mark v as active
3: ENQUEUE(v, Q)
4: while Q ≠ ∅ do
5:     w ← DEQUEUE(Q)
6:     Mark w as visited
7:     for each (w, d)∈ E do
8:         if d not yet visited and not active then
9:             Mark d as active
10:            ENQUEUE(d, Q)
```

Chapter 16

Topological Sort

Imagine that you have n tasks $T_1, T_2, \ldots, T_n$ that you have to solve. However you have some constraints regarding the order in which you have to solve the tasks. That is, you have some constraints like *you have to solve T_i before T_j*. In a first instance it is important to see that is not always possible to find a solution, consider for example that you have to solve the tasks T_1 and T_2 with 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 can be modelled with a graph $G = (V, E)$ as follows: we do 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 $ord : V \to 1, \ldots, |V|$ 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 (which you have proven formally in the lecture, but it should be easy to understand intuitively with the introductory example of task scheduling) 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 worthy to discuss how to run topological sorting on paper with a small graph. The algorithm that I use is the following:

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 24 TopologicalSort(G)

```
1: S ← ∅
2: Save In-degree of every node in an array A of length |V|
3: for each v ∈ V do
4:     if A[v] = 0 then PUSH(v, S)
5: i ← 0
6: while S not empty do
7:     v ← POP(S)
8:     ord[v] ← i; i← i+1
9:     for each (v, w) ∈ E do
10:        A[w]← A[w]-1
11:        if A[w]=0 then PUSH(w,S)
12: if i = |V| + 1 then return ord else return cycle
```

We observe that the stack contains all nodes v such that, in a given stage of the algorithm[1], 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 $\Theta(|V|)$.
- 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 finding a topological sort in a graph is $\Theta(|V| + |E|)$.

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

Chapter 17

Shortest Path

Given a graph $G = (V, E)$ where every edge $e \in E$ has a cost $c(e)$, a starting vertex s and an end vertex t, we want to find the shortest path $W = (s, v_1, \ldots, v_k, t)$ in G, i.e. we must have $(s, v_1) \in E$, $(v_i, v_j) \in E$ (for all $1 \leq i \leq k-1$ and $2 \leq j \leq k$), $(v_k, t) \in E$ and the sum of the costs associated to those edges must be the minimum among all paths from s to t. We note with $d(s, t)$ the cost of the shortest path from s to t. We assume that G has no negative cycles because in those case a shortest path does not exists (in facts, every tour along this cycle would give a shorter distance and hence the distance would be minus infinity). We also assume that the distance from a vertex to itself is zero.

One can show (by contraddiction) the optimality principle of shortest path, i.e. 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 has 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 (but in some cases, as we will see, it is possible to take advantage of this).

Now we are going to see different kind of algorithms to solve the shortest path problem. Note that there are different algorithms with different runtimes but that can be applied in different contexts. The following table summarizes different algorithms:

17.1 Modified BFS

If all edges have the same weight we can see 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 notion of adjacent and nearest is the same). Hence we can come up with the following algorithm (which has runtime $\mathcal{O}(|V| + |E|)$.

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

Algorithm 25 SP-BFS(G, s)

```
1:  d[s] ← 0
2:  d[v] ← ∞ for v ∈ V \ {s}
3:  Q ← ∅
4:  ENQUEUE(s,Q)
5:  while Q ≠ ∅ do
6:      u ←DEQUEUE(Q)
7:      for each (u, v) ∈ E do
8:          if d[v] = ∞ then
9:              d[v] ← u
10:             ENQUEUE(v,Q)
```

17.2 DP and TopoSort

We can use this algorithm for directed acyclic graphs and the idea is to use the recurrence:

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

and we fill the table in topological order. Concretely:

Algorithm 26 SP-DAG(G, s)

```
1: d[s] ← 0
2: d[v] ← ∞ for v ∈ V \ {s}
3: for v ∈ V in topological order
4:     for all (v, w) ∈ E
5:         if d[w] > d[v] + c(v, w)
6:             d[w] = d[v] + c(v, w)
```

It is clear that we can compute the topological sort in $\mathcal{O}(|V| + |E|)$. Afterwards the for loop requires to access all edges for 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 $\mathcal{O}(|V| + |E|)$.

17.3 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). 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 all nodes. 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 and hence 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.

Hence, at every iteration, we compute:

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

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.

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 have two 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:

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

Algorithm 27 Dijkstra(G, s)

```
1: for each v ∈ V \ {s} do
2:     d[v] = ∞
3: d[s] = 0
4: V \ S ← V
5: while V \ S ≠ ∅ do
6:     choose u ∈ V \ S with minimal d[u]
7:     V \ S − {u}
8:     for (u, v) ∈ E do
9:         if d[v] > d[u] + c(u, v) then
10:            d[v] = d[u] + c(u, v)
11:
```

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

17.4 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 edges 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 28 Bellman-Ford(G, s)

```
1: for each v ∈ V \ {s} do
2:     d[v] = ∞
3: d[s] = 0
4: for i ← 1, 2, ..., |V| − 1 do
5:     for each (u, v) ∈ E do
6:         if d[v] > d[u] + c(u, v) then
7:             d[v] ← d[u] + c(u, v)
8: for each (u, v) ∈ E do
9:     if d[v] > d[u] + c(u, v) then
10: 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.

17.5 Floyd-Warshall

This first 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 29 Floyd Warshall(G)

1: **for** $1 \leq u \leq |V|$ and $1 \leq v \leq |V|$ **do** $d(u, v, 0) = \infty$
2: **for** $u \in V$ **do** $d(u, u, 0) = 0$
3: **for** $(u, v) \in E$ **do** $d(u, v, 0) = c(u, v)$
4: **for** $i = 1, \ldots, |V|$
5: **for** $u = 1, \ldots, |V|$
6: **for** $v = 1, \ldots, |V|$
7: $d(u, v, i) = \min(d(u, v, i-1), d(u, i, i-1) + d(i, v, i-1)$
8: **return** d

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

17.6 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, otherwise the runtimes are equivalent). The problem happens 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.

Chapter 18

MST

The MST is a very famous problem of graph theory and has a lot of real world applications (e.g. deciding how to build a network). Formally the problem is:

Given a connected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}$, 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.

18.1 Boruvka's Algorithm

In this algorithm we begin with $|V|$ connected components S_i $(1 \leq i \leq |V|)$ and we contract them together until we have only one connected component. While doing this contraction we save in the set T the edges which are in the MST. Concretely:

Algorithm 30 Boruvka(G)

1: $T \leftarrow \emptyset$
2: **while** T is not a spanning tree
3: $\quad (S_1, \ldots, S_k) \leftarrow$ connected components of T
4: $\quad (e_1, \ldots, e_k) \leftarrow$minimum edges that leave $S_1, \ldots, S_k$
5: $\quad T \leftarrow T \cup (e_1, \ldots, e_k)$

Every iteration of the while loop takes $\mathcal{O}(|V| + |E|)$ because 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 $\mathcal{O}((|V| + |E|) \log |V|)$.

18.2 Prim's Algorithm

Now we see an algorithm which applies the blue rule. 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 31 Prim(G)

1: $S \leftarrow \emptyset$
2: Q with all nodes with priority ∞
3: DECREASE-KEY(Q,S,0)
4: **while** $S \neq V$
5: $u \leftarrow$ EXTRACT-MIN(Q)
6: $S \cup \{u\}$
7: **for** $(u, v) \in E$, $v \notin S$
8: DECREASE-KEY(Q,V,C(U,V)

The runtime of this algorithm is:

$$\mathcal{O}\left(|V| \cdot \text{INSERT} + |V| \cdot \text{EXTRACT-MIN} + |E| \cdot \text{DECREASE-KEY}\right)$$

If we use min-heaps we get a runtime of $\mathcal{O}((|V| + |E|) \log |V|)$.

18.3 Kruskal's algorithm

The idea of this 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. Concretely:

Algorithm 32 Kruskal(G)

1: Sort the edges with respect to their weights. Let $e_1, \ldots, e_{|E|}$ be the sorted sequence (in ascending order).
2: Create a union-find data structure and create a separated component for every node.
3: $T \leftarrow \emptyset$
4: **for** i=1 to $|E|$ **do**
5: Let $e_i = (u, v)$
6: **if no** SAME(U,V) **then**
7: $T \leftarrow T + e_i$
8: UNION(U,V)

If we use the clever trick to implement the union routine, we can do an amortised analysis and we get 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 $\mathcal{O}(|E| \log |E| + |V| \log |V|)$.