2019

Computational Thinking

Aniel Bhulai

Vrije Universiteit Amsterdam

Contents

1. Introduction

In the Computational Thinking course, you will be introduced to different solution strategies such as modeling, formulation, guess and check to solve problems. Also, different algorithms such as search algorithms, sorting algorithms and graph algorithms will be discussed during the lectures to solve problems. You will learn to solve problems by reasoning and by using knowledge from other disciplines. In the practical sessions, you will solve various problems using the different solution strategies and algorithms that have been discussed in the lectures. Since there are many ways to solve a problem, you will also start thinking about developing algorithms by yourself.

The aim of the Computational Thinking course is to teach you

- ✓ how to analyze problems,
- ✓ how to choose a right solution strategy or algorithm to solve problems,
- ✓ how to create algorithms, and
- ✓ how to translate algorithms into a flowchart.

For this course, we do not expect any specific foreknowledge. However, a little mathematical understanding may be to your advantage.

2. Solution strategies

2.1. Easy vs. efficient solution strategy

There are various strategies to solve everyday problems. Often a problem can be solved in different ways and there is not always a "best way". However, sometimes one way is more efficient than the other, or you find one approach easier or more pleasant than the other.

Consider the following problem of a tennis club which has financial problems (see Problem 1).

Problem 1

A tennis club has financial problems. The members of this tennis club want to help the tennis club by donating money. In total one hundred members were prepared to donate money to the tennis club. Each of the hundred persons donates an amount of money. Person 1 donates 1 Euro, person 2 donates 2 Euros, person 3 donates 3 Euros, person 4 donates 4 Euros, etc.

Calculate (without a calculator) how much money the members donate together.

Problem 1: Members donate money to the tennis club.

To solve this problem without a calculator we could sum up all the donated Euros of the one hundred members. This means we would have to make the following calculation:

$$\text{The total amount of donated money } = 1 + 2 + 3 + \cdots + 98 + 99 + 100 \qquad \text{eq. (1)}$$

This is an easy solution to solve the problem, but it is a time-consuming solution. Besides that, it is not an efficient solution. And what if we would have one thousand members or one million members instead of one hundred members. Would you still use the same solution strategy?

It is obvious that we should use another solution strategy to solve this problem more efficiently. As previously stated, there are several strategies to solve a problem. What we need is to find a proper solution strategy to solve the problem efficiently. And to find out what the proper solution strategy is we need to know how we should approach a problem. Therefore, we discuss a scheme in the next Section to tackle problems. This will help us in approaching a problem systematically.

2.2. How do we approach a problem?

To solve a problem efficiently it is necessary to approach the problem systematically. This means that we must follow certain steps to solve the problem. In Figure 1 we see the steps which are required to solve a problem.

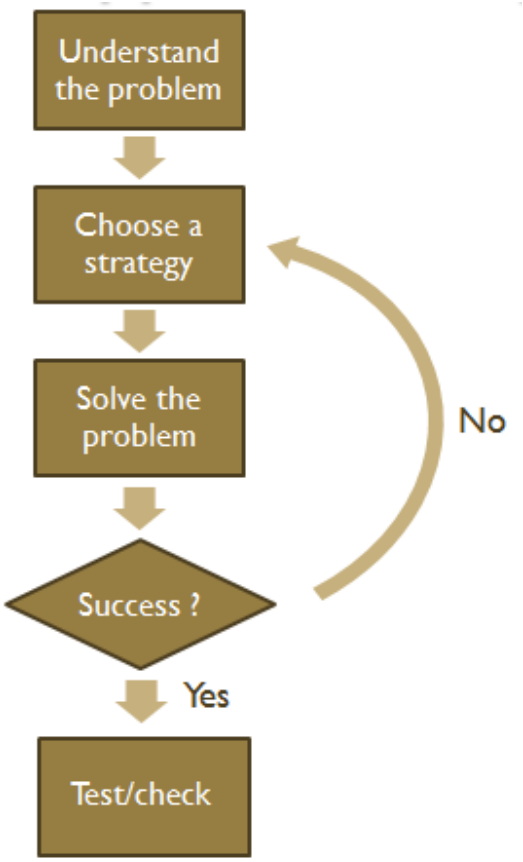


Figure 1: Steps to approach a problem

The *first step* in approaching the problem is to have a good understanding of the problem. This means can we formulate the problem in our words, do we understand the problem properly, do we know what we have to solve, have we seen such a problem before, which information does the problem provide us, and what information do we need to solve the problem.

In the *second step,* we choose the right solution strategy to solve the problem in the *third step*. Sometimes we must combine a few solution strategies to solve the problem.

The *fourth step* is to check whether the solution strategy worked or not to solve the problem. If the solution strategy did not work, then we must go back to the second step and choose another solution strategy to solve the problem.

If we think that we have found the solution, then we should check the solution in the *fifth and last step*. We might, for example, make miscalculations or fallacies. Sometimes we cannot check the solution. In those cases, we should check whether the solution we have found is reasonable or not (e.g., check the size of the number).

In the next Section, we will discuss some solution strategies. In the lectures, we will apply the solution strategies to some examples to see how the different solution strategies work.

3. Examples of some solution strategies

There are many solution strategies to solve problems. We will discuss a few of them. How the solution strategies work is left as an exercise for the reader.

In the next sections we will discuss the following solution strategies briefly:

1) The guess and check solution strategy
2) The go through all the possibilities solution strategy
3) The divide the problem into several subproblems solution strategy
4) The use of formulas or equations solution strategy
5) The discover a structure or pattern solution strategy
6) The make a model solution strategy
7) The brute force solution strategy
8) The divide-and-conquer solution strategy

3.1. The guess and check solution strategy

The guess and check solution strategy is also known as the “Try something” solution strategy. This solution strategy can be used if the problem is not to complex and if we have an overview of the problem.

3.2. The go through all the possibilities solution strategy

The go through all the possibilities solution strategy can be used when there are not so many possibilities to solve a problem. If, for example, there are only four possibilities to solve a problem, then we can easily try all the four possibilities. The go through all the possibilities solution strategy is only suitable if the number of possibilities is limited.

3.3. The divide the problem into several subproblems solution strategy

Sometimes a problem looks very complex. But when you divide the problem into subproblems, the problem becomes easier to solve. We use the divide the problem into several subproblems solution strategy to divide a complex or big problem into subproblems. There are several approaches to divide a problem into subproblems. We could, e.g., use the approaches simplifying, back reasoning or exclusion to divide a problem into subproblems. We could also combine a few of these approaches to reduce the complexity of the problem. By reducing the complexity of the problem, the problem will be clear(er) and hence better to solve.

3.4. The use of formulas or equations solution strategy

Sometimes it is easy to use formulas and equations to solve a problem. By using formulas, we sometimes have a better overview of the problem. Complex problems become easier to handle and to solve.

3.5. The discover a structure or pattern solution strategy

Some problems can be solved by discovering a structure or a pattern in similar problems. Sometimes it is necessary to extend the problem or to look at other similar problems to discover a structure or pattern. Once the structure or pattern is known it becomes easier to solve the problem.

3.6. The make a model solution strategy

Often mathematical models are used to solve complex problems. When we make a model of a problem, we omit many aspects, which make the problem complex. By doing so we get a simpler problem or situation, which can often be solved by known and simple techniques.

When making a model we could also think of, for example, making a diagram or drawing a picture. By using such a model, we can discover things that may help us to solve the problem.

3.7. The brute force solution strategy

The brute force solution strategy is a simple approach to solve problems. It relies on sheer computing power to try all possibilities until the solution to the problem is found. This means that brute force is not using any algorithm or heuristics[1] to speed up the calculation to solve the problem. Brute force is often used if no algorithm is known that is faster or more efficient which leads to a solution. The linear search algorithm and the bubble sort algorithm are examples which use the brute force solution strategy. We will discuss these algorithms in detail in Section 6 and Section 7.2.

3.8. The divide-and-conquer solution strategy

The divide-and-conquer solution strategy (D&C) is an important and probably the best-known general technique to design algorithms. The D&C technique consists of three major steps (see also Figure 2). In the first step, we divide a problem of n size into a number of small subproblems of the same type and ideally of about the same size.

[1] A heuristic is a rule or method that helps us solve problems faster than we would if we did all the computing. We can think of heuristic as a mental shortcut that allows people to solve problems and make judgments quickly and efficiently [29]. These rule-of-thumb strategies shorten decision-making time and are helpful in many situations, but they can also lead to biases.

When we are trying to solve a problem or make a decision, we often turn to these mental shortcuts when we need a quick solution.

While heuristics can speed up our problem and decision-making process, they can introduce errors. Just because something has worked in the past does not mean that it will work again, and relying on an existing heuristic can make it difficult to see alternative solutions or come up with new ideas [28].

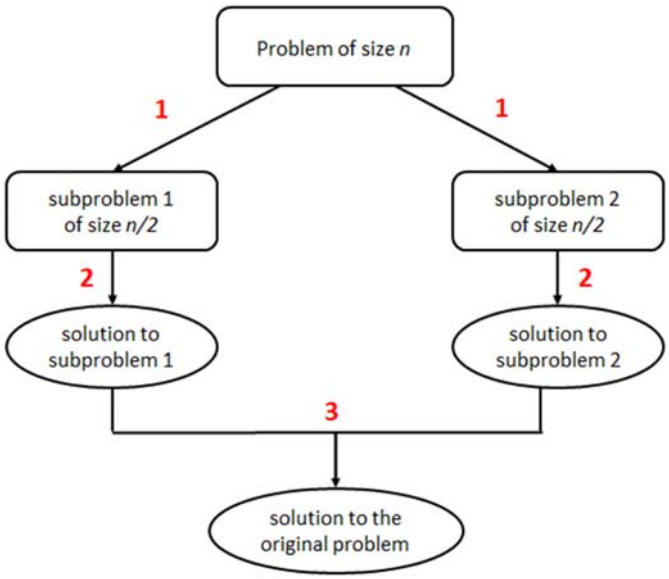


Figure 2: Divide-and-Conquer technique

We divide the original problem into subproblems until these become simple enough to be solved directly. In the example, in Figure 2 the problem of n size is divided into subproblem 1 and subproblem 2, each of $n/2$ size. In the next step of the D&C technique, we solve each subproblem recursively. And in the third step, we combine all these solutions to get a solution to the original problem.

A typical divide-and-conquer algorithm solves a problem using the following three steps:

1. Divide: Break the given problem into subproblems of the same type.
2. Conquer: Recursively solve these subproblems.
3. Combine: Appropriately combine the solutions.

It is good to know that the D&C technique is ideally suited for parallel computations. In parallel computations, each subproblem can be solved simultaneously by its own processor.

In Section 5, Section 7.3.1, and Section 7.5 we will discuss the binary search, the merge sort, and the quick sort algorithms respectively which make use of the D&C technique.

3.9. Exercises

1) Solve the problem mentioned in exercise 1 by using formulas and equations.

2) The bubble sort algorithm belongs to which solution strategy?

3) What is the brute force solution strategy?

4) Name two algorithms which belong to the brute force solution strategy.

5) What is the divide-and-conquer solution strategy?

6) Name two algorithms which belong to the D&C technique.

7) **How many coins?**
Suppose you have an amount of €5.20 in coins of 5 and 20 Euro cents. You have three times as many coins of 20 Euro cents as 5 Euro cents.
Calculate how many coins you have without a calculator. Use the scheme to approach a problem and use the given solution strategies in this Chapter.

8) **Hourglasses**
Suppose you have an hourglass of 7 minutes and an hourglass of 11 minutes. Now, you want to measure exactly 15 minutes with these hourglasses.
How can you measure exactly 15 minutes with these two hourglasses? Use the scheme to approach a problem and use the given solution strategies in this Chapter.

9) **What number has Alex in mind?**
Alex has a number in mind. He subtracts 99 from this number and he multiplies the result by 55. The final result of the calculation is an odd number.
Did Alex have an odd or an even number in mind? Use the scheme to approach a problem and use the given solution strategies in this Chapter.

10) **Last digit of 7^{77}**
Determine the last digit of 7^{77}? Use the scheme to approach a problem and use the given solution strategies in this Chapter.

11) **Stone slab**
Suppose we have a heavy stone slab which must be transported. For this purpose, the slab is on three rollers. Each roller has a circumference of one meter.
Determine how far the slab in total has moved forward if the rollers have made exactly one rotation? Use the scheme to approach a problem and use the given solution strategies in this Chapter.

12) **Two snails**
Two snails (each weighs 29 grams) are located at the bottom of a dry well, exactly 20 meters deep. Snail number one is unhappy and decides to move out of the well. Every

day he climbs 5 meters upwards the wall, but at night he sleeps and slowly slides 4 meters down because of the gravity.

a) How many days does it take snail number one to reach the edge of the well? Motivate your answer and explain which strategy you have chosen to solve this problem.

b) Given the same circumstances as before, how many days does it take both snails to reach the edge of the well when they climb up together?
Motivate your answer.

13) **Pheasants and rabbits**

William is a huge animal lover. This time he has bought a cage with both pheasants and rabbits in the market. But what he does not know is how many pheasants and rabbits are in the cage; the seller could not tell him that. However, the seller could tell William that there were 35 heads and 94 legs in the cage.

a) How many pheasants are there in the cage William has bought?
Motivate your answer and explain which strategy you have used.

b) How many rabbits are there in the cage?
Motivate your answer.

14) **Wine**

A wine vendor is having a final sale. He has only three barrels left; one with a capacity of 10 liters, one of 7 liters and a small one of 3 liters. Only the biggest barrel is filled with red wine, the two smaller barrels are empty. Two customers want to share the last wine in equal parts of 5 liters each. The vendor has no scales or any other way to measure the wine, he can only pour the contents of the barrel into the other barrels. He divides the wine into equal parts without estimating, just by pouring the wine from barrel to barrel. He manages to do so in a limited number of moves only.

Show how the vendor can manage this (using the lowest number of moves possible). Motivate your answer and explain which strategy you have used.

15) **Box of oranges**

Three travelers arrive late at night at the inn, but the kitchen is already closed. The innkeeper mutters: all I have is a box of old oranges... The travelers say that is fine since the alternative is no food at all. The innkeeper disappears and the three travelers have fallen asleep by the time he comes back. The innkeeper does not want to wake the travelers, so he leaves the box on the table and goes to bed.

The first traveler wakes up and finds the box of oranges. Not wishing to make a pig of himself and not knowing if anyone else has eaten something already, he throws away one orange that looks bad and eats a third of what is left. He goes back to sleep.
The second traveler wakes up. This traveler wants to be fair too. He throws out two ugly, dried out oranges, eats a third of what remains and goes back to sleep.
The third traveler wakes up. He throws out three half rotten oranges and eats a third of what remains.
It is almost morning and the innkeeper returns and takes away the box. Only six left, mutters the innkeeper.

Calculate how many oranges there were in the box to start with?
Motivate your answer and explain which strategy you have used.

16) **Job hunt**

Peter applies for a job at the company "Car Service". He must pass a test in order to get through to the next round. The test is the following:
A Ferrari 599XX (usage 16 liters/100 kilometers) and a Mercedes CLS 300 (usage 10 liters/100 kilometers) drive towards each other. The Ferrari barrels down the road at 130 kilometers per hour; the Mercedes goes 110 kilometers per hour. What is the distance between the two cars exactly one minute before the collision?

Help Peter pass this test. Calculate the distance exactly one minute before the collision between the Ferrari 599XX and the Mercedes CLS 300. Give the solution strategy you have used.

17) **Sharing pizza**

Karen has invited three friends, Molly, Judy, and Carol, to have dinner at her place. Since Karen does not like to cook, she calls for pizza. She orders three pizzas at *Trattoria Bellissima*: a pizza with mozzarella and salami, a pizza with tuna and a pizza with mushrooms. Since the friends all want to try every pizza, Karen divides the three pizzas into twelve slices each. Carol does not like salami but wants to have a slice with only mozzarella. On every slice of this pizza, there is either salami, mozzarella or both. On six slices there is salami, on ten slices mozzarella. Molly, on the other hand, does not like mozzarella and wants to eat every slice with salami only. Molly also wants a single slice of every other pizza.

a) How many slices of the mozzarella and salami pizza can Carol eat?
   Motivate your answer.

b) How many slices have both mozzarella and salami?
Explain which solution strategy you have used.

c) How many slices does Molly get?

18) **Pooling resources**

Alice and Bob have two market stalls next to each other. Both were selling cheap USB sticks. Alice decided to price hers at 2 for 10 euro, while Bob was thinking of asking 20 euro for 3. Each of them has 30 USB sticks so together they can make 150 + 200 = 350 when every USB stick is sold. Worried about the competition they decided to pool their resources and reasoned that together they can sell their products 5 for 30 euro. At that price, if they sell all USB sticks, their income would be 360 euro, 10 euro more.
Just across the street, Chris and Diane are also selling USB sticks; they also have 30 USB sticks each to sell. Chris was thinking of selling his at 2 for 10 euro, while Diane was thinking about undercutting the competition by selling hers at 3 for 10 euro. When they heard about Alice and Bob combining forces, they also decided to pool their resources, selling their USB sticks at 5 for 20 euro.

a) Why do Alice and Bob get a higher result in working together than selling their products by themselves? Explain your answer.

b) Is it a good idea for Chris and Diane to combine their resources too? Explain your answer.

19) **Nim**

Nim is a two-player game. The players take turns to remove a number of objects (for example matchsticks) from a pile. There are many variations to the game. The differences are the number of objects you are allowed to remove and the number of objects you start with.

In this version, we start with a pile of 21 matchsticks. The two players take turns and can remove at least 1 and maximal 5 matchsticks. The players cannot repeat each other's moves (so if the first player removes 2 matchsticks, the second player can only remove 1, 3, 4 or 5 matchsticks). The player who removes the last matchstick wins the game.

A *winning strategy* is a way of playing, which always leads to profit, i.e., the player using this strategy always wins the game, whatever moves the opponent is choosing to prevent it.

a) Does a winning strategy exist for this game and if so, who can use this strategy, the first player (who can remove the first matchsticks) or the second player? Explain your answer.

b) Explain which solution strategy you have chosen and why.

20) **Multiplication arrays**

The square array given below uses each of the digits 1 through 9:

192
384
576

The second row is twice the first row and the third row is three times the first row. There are three other ways to rearrange the digits to do this. You can use each digit one time only. Find them and explain which solution strategy you have used to solve this problem.

21) **Decrypt the hidden message**

An archeologist has found a piece of paper with some strange symbols in a secret room in an old castle from the seventeen hundred. He suspects it is a hidden message, but he cannot decipher it. The symbols on the piece of paper are pictured below.

Help the archeologist to decrypt this message. Explain which solution strategy and/or algorithm(s) you have used.

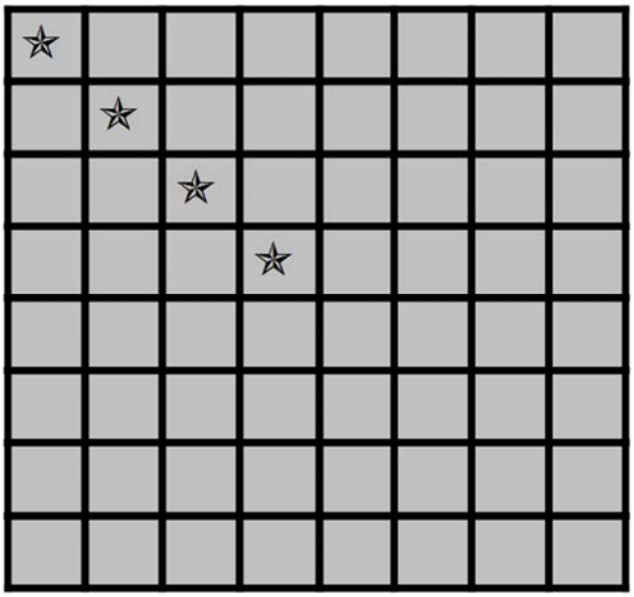
22) **The Asterian Temple**

Indy and Lara have penetrated the inner sanctum of the Asterian star temple. They face the final obstacle to the hidden chamber of secrets. On the door is a square array of 64

tiles, with golden star disks on four tiles.

“What do we have to do this time?”, asked Lara.

“According to the Lost Papyrus of the Goddess Asteria, we must divide all the tiles into four non-overlapping connected regions, each with a star disk and each of the same size.”, replied Indy. “Then the door to the secret chamber will open for us. But: each of the four regions has to be the same shape as well!”



a) Draw a figure showing the answer and explain how you got your answer.

b) Which solution strategy have you used to solve this problem?

23) **Puzzling cubes**

Karen has received a box with cubes for her birthday. The box contains nine cubes with the numbers 1 through 9, and three cubes with plus and minus sign on the different faces of the cube. With the box comes an instruction manual with a few games and the rules explained. One of the games is as follows:

- Place all the cubes in a single row with the numbers in ascending order.
- Now place the three cubes with the two possible arithmetic symbols between the number cubes (do not change the order of the number cubes!) in such a way that the result of the formula is 100.

Help Karen solve this puzzle. Place the three arithmetic cubes between the nine number cubes.

Note: you can place three symbols between the numbers, three minus signs, three plus signs or any other combination of plus and minus signs. Adhere strictly to the rules. Show and explain how you solved this puzzle.

4. Algorithms

4.1. The importance to know something about algorithms

Why is it so important to know something about algorithms? There are both practical and theoretical reasons to study algorithms if you are thinking to work later as a computer professional. From a practical point of view, you should know a standard set of important algorithms of computer science. Besides having knowledge of the different algorithms, you should be able to design new algorithms and analyze their efficiency. From a theoretical point of view, the study of algorithms has come to be recognized as the pillar of computer science. But even if you are not a student in a computing-related program, there are several reasons for having some knowledge of algorithms. One of the reasons is that nowadays computer applications are integrated into many things, like mobile phones, tablets, watches, and wearables. No computer program would exist without algorithms, and without a computer program, we would not be able to make those nice devices. Computer programs are becoming indispensable in many aspects of our personal and professional lives. So, studying algorithms becomes a necessity for more and more people.

Another reason for studying algorithms is that we develop analytical skills, which is necessary to solve problems. Algorithms can be considered as special kind of solutions to problems – not just answers but precisely defined procedures for getting answers [1, p. 27]. That's why specific algorithm design techniques can be interpreted as problem-solving strategies. Donald Knuth, one of the most prominent computer scientists in the history of algorithms, put it as follows: "A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a computer, i.e., expressing it as an algorithm… An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way." [2, p. 9].

4.2. What is an algorithm?

What is an algorithm? Formally we can say that

> An algorithm is an effective method, which consists of a finite number of steps or instructions to solve a problem or to accomplish a task.

In everyday life, you might have an algorithm for getting from home to school,

Step 1: Walk to the bus station
Step 2: Take the bus
Step 3: Walk to school

Figure 3: Getting from home to school

for efficiently going through your shopping list in a grocery store,

Step 1: Scan the list from top to bottom
Step 2: Find the item that you want to buy
Step 3: Cross out the item that is bought

Figure 4: Shopping list

or for preparing for your exam.

Step 1: Attend lectures
Step 2: Study slides
Step 3: Make assignments
Step 4: Study textbooks
Step 5: Make summaries

Figure 5: Preparing for your exam

One important thing about an algorithm is that it always works if we follow the steps. It is like a recipe for baking a cake. If you follow the recipe correctly you will always end up with the same

cake. An algorithm gives us, in fact, a roadmap to solve problems or to accomplish a task. A solutions strategy, on the other hand, does not necessarily always work. That is the main difference between an algorithm and a solution strategy. Note that Figure 1, the steps to approach a problem, is in fact, an algorithm, namely an algorithm to approach and to solve problems.

4.3. Properties of algorithms

An algorithm is useful if it helps us to find a solution to a specific problem. For that to happen, we should comply with the properties of an algorithm if we want to design an algorithm. An algorithm must satisfy the following properties.

a) **Input:** Usually, an algorithm takes input values from a specified set of elements, where the amount and type of inputs are specified.
b) **Output**: The algorithm produces the output values for the specified set from the input values. These output values are the solution for the specific problem.
c) **Definiteness**: The steps of the algorithm must be defined precisely. This means that there is no ambiguity.
d) **Correctness**: The algorithm should produce the correct output values for each set of input values.
e) **Finiteness**: The output values should be produced after a finite number of steps for any input values in the specified set.
f) **Effectiveness**: It must be possible to perform each step of the algorithm correctly and in a reasonable amount of time.
g) **Generality**: The algorithm should work for all problems of the desired form, not just for a particular set of input values.

When an algorithm satisfies these properties, it is a fail-proof way to solve the problem for which it was designed.

4.4. Pseudocode

Suppose we want to design an algorithm for counting the number of people in a classroom. You probably would point at each person one at a time and start counting from 0, 1, 2, 3, 4, 5 and so forth. Well, that's an algorithm. When we design an algorithm, it is a good habit to express the algorithm a bit more formally in what we call *pseudocode*. Pseudocode is an English-like syntax that resembles a programming language. It is a detailed yet readable description of what a computer program or algorithm must do. The pseudocode is expressed in a formally styled natural language rather than in a programming language. A pseudocode for counting people in a classroom could be

```
1.    Let NumberOfPerson = 0
2.    for each person in the classroom
3.          set NumberOfPerson = NumberOfPerson + 1
```

How we interpret this pseudocode? In line 1 we declare, so to speak, a variable called NumberOfPerson, and we initialize its value to zero. This just means that at the beginning of our algorithm, the thing with which we are counting has a value of zero. After all, before we start counting, we haven't counted anything yet. Calling this variable NumberOfPerson is just a convention. We could have called it almost anything. In line 2 we demark the start of a *loop*. A loop is a sequence of steps that is repeated some number of times. In our example, the step we are taking is counting people in the classroom. Line 3 describes exactly how we are counting. The indentation implies that it is line 3 that will be repeated. So, what the pseudocode is saying is that after starting at zero, for each person in the classroom, we will increase NumberOfPerson by 1.

Writing pseudocodes has some advantages. One of the advantages is that catching errors at the pseudocode stage is less costly than catching them later in the development process. Another advantage is that the pseudocode can be inspected by the team of designers and programmers as a way to ensure that actual programming is likely to match the design specifications.

4.5. Exercises

1) What is an algorithm?

2) Give at least three reasons why we should know something about algorithms.

3) Give three examples (not mentioned in this reader) for which you may have an algorithm and name the steps in the algorithms.

4) We should comply with the properties of an algorithm when we design an algorithm. Name seven properties to which an algorithm should comply with.

5) Explain what a pseudocode is.

6) Give a pseudocode for counting *each pair* of people in a room.

7) Give an example of a loop in pseudocode.

5. Big O

5.1. Notation

In computer science, *big O notation* (with a capital letter O, not a zero), also called *Landau's symbol*, is used to classify algorithms by how they respond (e.g., in their processing time or working space requirements) to changes in input size. Big O notation characterizes functions according to their rates of growth: different functions with the same rate of growth may be represented using the same big O notation (e.g., $f(x) = \mathrm{x}^2 + 2x + 8$ and $g(x) = 5\mathrm{x}^2 + 10x + 100$). *Landau's symbol* comes from the name of the German number theoretician Edmund Landau who invented the notation [6]. The letter O is used because the rate of growth of a function is also called its *order*. A description of a function in terms of big O notation usually only provides an upper bound on the rate of growth of the function.

Big O notation is useful when analyzing algorithms for efficiency. For example, one might find that the time (or the number of steps) it takes to complete a problem of size n is given by $T(n) = 4n^2 - 2n + 2$. As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected; for instance when $n = 500$, the term $4n^2$ is 1000 times as large as the $2n$ term. Ignoring the latter would have a negligible effect on the expression's value for most purposes. Further, the coefficients become irrelevant if we compare to any other order of expressions, such as an expression containing a term n^3 or n^4. So, for our example, we could say that "$T(n)$ grows at the *order* of n^2" and write: $T(n) = \mathrm{O}(n^2)$. Note that the "=" sign is not meant to express "is equal to" in its normal mathematical sense. The "=" sign is a more colloquial "is". We can write $T(n) = \mathrm{O}(n^2)$ also as $T(n) \in \mathrm{O}(n^2)$ which is sometimes considered more accurate.

Table 1 shows a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. The slower growing functions are listed first. In each case, c is an arbitrary constant and n increases without bound (i.e., $n \to \infty$).

Running time $(T(n))$	Name	Examples of running times	Example algorithms
O(1)	Constant time	10	Determining if an integer (represented in binary) is even or odd.
O$(log(n))$	Logarithmic time	$log(n^2)$, $log(n)$	Binary search
O$((log(n))^c)$, $c > 1$	Polylogarithmic time	$(log(n))^2$	Matrix chain ordering can be solved in polylogarithmic time on a Parallel Random Access Machine.
O(n)	Linear time	n	Finding the smallest item in an unsorted array.
O$((n^2)$	Quadratic time	n^2	Bubble sort; Insertion sort
O(n^c), $c > 1$	Polynomial time	$n, nlogn, n^{10}$	Tree-adjoining grammar parsing; maximum matching for bipartite graphs
O(c^n), $c > 1$	Exponential time	2^n, 2^{n^2}	Finding the (exact) solution to the traveling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search.

Table 1: Big O notations [7] [6]

Note that O(n^c) and O(c^n) are very different. The latter grows much, much faster, no matter how big the constant *c* is. A function that grows faster than any power of *n* is called *superpolynomial*. One that grows slower than an exponential function of the form c^n is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential. Examples of this include the fastest algorithms known for integer factorization.

Note, too, that O$(log(n))$ is exactly the same as O$((log(n))^c$. The logarithms differ only by a constant factor, and the big O notation ignores that. Similarly, logs with different constant bases are equivalent.

5.2. Understanding Big O

How efficient is an algorithm or a piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- Memory usage
- Disk usage
- Network usage

They are all important but we will mostly talk about *time complexity* (CPU usage) which is expressed by using the big O notation. We want to stress here the difference between performance and time complexity of an algorithm. The performance deals with how much time, memory, disk, etc., is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code. Time complexity, on the other hand, deals with how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger [8]? Note that complexity affects performance but not the other way around.

The time required by a function or procedure is proportional to the number of "basic operations" that it performs. Some examples of basic operations are:

- One arithmetic operation (e.g., +, *, /).
- One assignment (e.g., $x := 0$)
- One test (e.g., $x = 0$)
- One read (of a primitive type: integer, float, character, Boolean)
- One write (of a primitive type: integer, float, character, Boolean)

Some functions or procedures perform the same number of operations every time they are called. We say that these functions or procedures take *constant time*. Other functions or procedures may perform a different number of operations, depending on the value of a parameter. For example, in the bubble sort algorithm, the number of elements in the list determines the number of operations performed by the algorithm. This parameter (i.e., the number of elements) is called the *problem size* or *input size* [8].

When we are trying to find the complexity of an algorithm (program, function, or procedure), we are not interested in the exact number of operations to the problem size. Instead, we are interested in the relation of the number of operations to the problem size.

Typically, we are usually interested in the worst-case time complexity, denoted as $T(n)$. This means we are interested in the maximum number of operations that might be performed for a given problem size. For example, when we are inserting an element into an array, we have to move the current element and all of the elements that come after it one place to the right in the array. In the worst case, inserting at the beginning of the array, all of the elements in the array must be moved. Therefore, in the worst case, the time for insertion is proportional to the number of elements in the array, and we say that the worst-case time for the insertion operation is linear in the number of elements in the array. For a linear-time algorithm, if the problem size doubles, the number of operations also doubles [8].

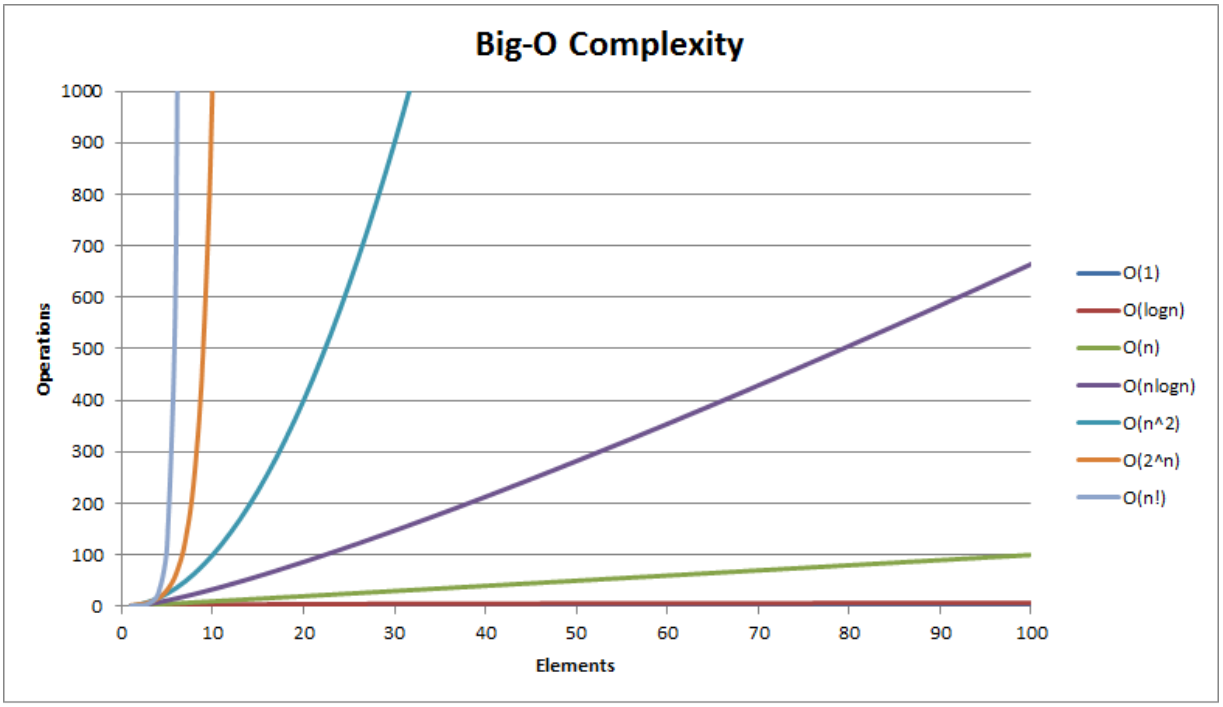


Figure 6: Big-O complexity [9]

5.3. Determining complexities

In general, how can we determine the running time of a piece of code? The answer to this question is that it depends on what kinds of statements are used. Below we give some examples of statements and the running time.

5.3.1. Sequence of statements

Suppose we have a sequence of statements, like the following:

statement 1;
statement 2;
...
statement k;

Then the total time is found by adding the times for all the statements:

$$total\ time = time(statement\ 1) + time(statement\ 2) + \cdots + time(statement\ k)$$

If each statement is "simple" (i.e., it only involves basic operations) then the time for each statement is constant and the total time is also constant, which means that $T(n) = O(1)$.

5.3.2. If-Then-Else

Suppose we have the following if-condition

```
if (condition) then
        block 1 (sequence of statements)
else
        block 2 (sequence of statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time complexity is the slower one of the two possibilities:

$$\max(time(block\ 1), time(block\ 2))$$

If block 1 takes O(1) and block 2 takes O(N), then the if-then-else statement would be O(N).

5.3.3. Loops

Suppose we have a loop like

```
for I in 1 .. N loop
        sequence of statements
end loop;
```

In the example, the loop executes N times, so the sequence of statements also executes N times. If we assume the statements are O(1), then the total time for the for-loop is N * O(1), which is O(N) overall.

5.3.4. Nested loops

Suppose we have the following nested loop

```
for I in 1 .. N loop
        for J in 1 .. M loop
                sequence of statements
        end loop;
end loop;
```

In the example, the outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the time complexity is O(N * M).

5.3.5. Statements with function or procedure calls

When a statement involves a function or procedure call, the time complexity of the statement includes the time complexity of the function or procedure. Assume that we know that a function or procedure f takes constant time, and that a function or procedure g takes time proportional to (linear in) the value of its parameter k. Then the statements below have time complexities as indicated.

$f(k)$ has O(1)
g(k) has O(k)

Then when a loop is involved, the same rule applies, e.g.,

```
for J in 1 .. N loop
        g(J);
end loop;
```

The above example has a time complexity of $O(N)^2$. The loop executes N times and each function or procedure call $g(N)$ has a time complexity of O(N).

5.4. Exercises

1) What is the difference between the performance and the time complexity of an algorithm?

2) What do we mean with the worst-case time complexity of an algorithm?

3) What is the time complexity of $f(n) = 10\log(n) + 5(\log(n))^3 + 7n + 3n^2 + 6n^3$?

6. Search algorithms

6.1. Introduction

Searching for a keyword or value is the basis of many computing applications, whether it is in an internet search engine or in a contact list of your mobile phone. In all those situations we use consciously or unconsciously a search algorithm or some strategy to search. In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items [10]. The items may be stored individually as records in a database or may be as elements of a search space[2]. There are many search algorithms. For searching too there is no single algorithm that fits all situations best. Some algorithms are faster than others but require more memory. Some algorithms are fast but can only be applied to sorted arrays, and so on. Unlike with sorting algorithms, there are no stability problems with searching algorithms [1].

6.2. Linear search

Linear search is also known as *sequential search* and is the simplest search algorithm for finding a particular element or value in a list. It is the most basic search algorithm that we can have. Linear search belongs to the Brute force strategy and is only efficient for small lists. For big lists (n > 8) binary search is more efficient. Linear search sequentially moves through a list and checks each element until the desired element (the so-called *key*) is found or the list is exhausted. When linear search finds the desired element, it returns the location of the element and stops searching further. For linear search, the list does not need to be sorted. Figure 7 shows the linear search pseudo code.

```
FOR each element in the list
        check if element is equal to the element being searched
            IF element found
                    stop with searching and return the element's location
            ELSE continue
stop
```

Figure 7: Linear search pseudo code

6.2.1. Analysis

If we look at the performance of linear search, we see that for the best case the algorithm needs only one comparison to search the key element in a list with *n* elements. The best case occurs when the key element is at the first position of the list. In the worst case, the key

[2] A search space is the feasible region defining the set of all possible solutions.

element is either at the last position or not present in the list. This means that in the worst case the algorithm needs *n* comparisons if there are *n* elements in the list.

If the value being sought occurs *k* times in the list, and all orderings of the list are equally likely, then the expected number of comparisons is

$$\begin{cases} n, & \text{if } k = 0 \\ \dfrac{n+1}{k+1}, & \text{if } 1 \le k \le n. \end{cases} \qquad \text{eq. (1)}$$

The worst case cost and the expected cost of the linear search are both O(*n*).

6.3. Binary search

The *binary search algorithm* (also called the *half-interval search algorithm*) belongs to the Divide-and-Conquer solution strategy, which we discussed in Section 3.8. Binary search is an efficient algorithm to search the position of an element in a sorted list. A sorted list means that the elements in the list are arranged either in increasing order or in decreasing order. The average speed of binary search is faster than the linear search.

The binary search algorithm is deceptively simple. Pretend someone is thinking of a number between 1 and 100. Every guess we take, the person says higher or lower. The most efficient way to discover the person's number is to first guess 50. The person says "higher". We guess now 75. The person says "lower". We guess now 63. The person says "higher". We guess 69. The person says "Yes!".

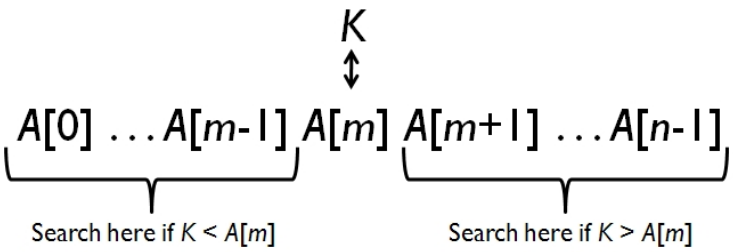


Figure 8: Value being sought by binary search.

Consider the list in Figure 8. The binary search algorithm starts by comparing a search key *K* with the middle element A[*m*] of the list. If they match, the algorithm stops. If the search key *K* is larger than the middle element A[*m*], then the first half of the list will be eliminated and the algorithm will continue with the same operation recursively for the second half of the list. If the search key *K* is less than the middle element, then the second half of the list will be eliminated and the algorithm will continue with the same operation recursively for the first half of the list. In this way, the binary search algorithm reduces the number of elements needed to be checked

by a factor of two each time and finds the sought value if it exists in the list or if not, it determines “not present”, in logarithmic time.

Thanks to its performance characteristics over large collections the binary search algorithm is frequently used. The only time binary search does not make sense is when the collection is being frequently updated (relative to searches) since re-sorting will be required.

6.3.1. Analysis

The standard way to analyze the efficiency of binary search is to count the number of times the search key *K* is compared with an element of the list. Moreover, for the sake of simplicity, we will count the so-called three-way comparisons [1]. This assumes that after one comparison of *K* with A[m], the algorithm can determine whether *K* is smaller, equal to, or larger than A[m].

When we count the three-way comparisons, we see that in the best case we just need 1 comparison to find the search key *K*. In the best case, the search key *K* is equal to the middle element A[m]. In the worst case, we need $\lceil \log_2(n+1) \rceil$ comparisons and in the average case we need about $\log_2(n)$ comparisons. Note that we take the ceiling of $\log_2(n+1)$ in the worst case[3].

The runtime complexity of the binary search algorithm is O(1) in the best case. For both in the worst case and in the average case, the runtime complexity of binary search is $O(\log(n))$.

6.4. Exercises

1) What is the expected number of comparisons if the value being sought occurs once in the list, and all orderings of the list are equally likely?

2) We want to search a list with *n* elements with the linear search algorithm. Explain what $k = 0$ means.

3) For binary search we need $\lceil \log_2(n+1) \rceil$ comparisons in the worst case. Explain what this means and what the value of *k* is in this worst case of binary search.

4) What do we mean by the three-way comparisons?

5) **Recipe collection**
   Below you can see a small part of the recipe collection of Grandma. The page numbers

[3] The floor and ceiling functions are defined as $floor(x) = \lfloor x \rfloor$ is the largest integer not greater than x and $ceiling(x) = \lceil x \rceil$ is the smallest integer not less than x [27].

have been left out here.

Recipe Collection

(1) Apple pie
(2) Baguette
(3) Banana cream pie
(4) Brownies
(5) Carrot cake
(6) Cheesecake
(7) Chocolate chip cookies
(8) Cornbread
(9) Crumble
(10) Cupcakes
(11) Gingerbread
(12) Key Lime pie
(13) Oatmeal cookies
(14) Pancakes
(15) Pizza
(16) Pudding
(17) Quiche
(18) Raisin bread
(19) Sourdough bread
(20) Waffles

a. Show with binary search how you would look up the recipe for Brownies in this list.
b. Demonstrate that linear search, in this case, is faster than binary search.

6) **Comparisons of keys**

When you look up a key (e.g., value or element) with binary search, what is the largest possible number of key comparisons in the following list?
Show how you got your answer.

4	13	25	33	38	41	55	71	73	84	86	92	97

7) **Estimating performance***

Estimate how many times faster an average successful search will be in a sorted array of 100,000 elements if it is done by binary search versus sequential search.

7. Sorting algorithms

7.1. Introduction

Sorting as a concept is deeply embedded in a lot of things that we do. It's quite often that we would like to arrange things or data in a certain order. We do this to improve the readability of that data or to be able to search in the data or to extract some information quickly out of that data.

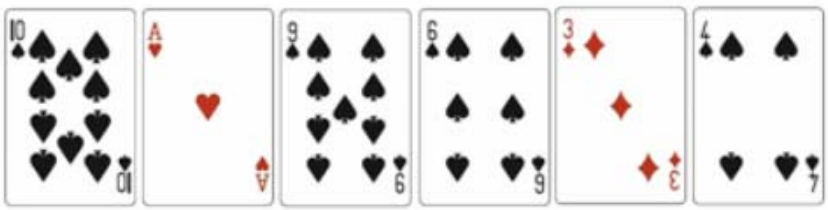


Figure 9: Unsorted hand of cards

Take the simple example of playing a card game. When we are playing a card game, even though the number of cards in our hands is less, we like to keep our hands of cards sorted by rank or suit. So, if our hand of cards is as in Figure 9 and we would like to keep it in increasing order of rank, then the arrangement will be something like in Figure 10.

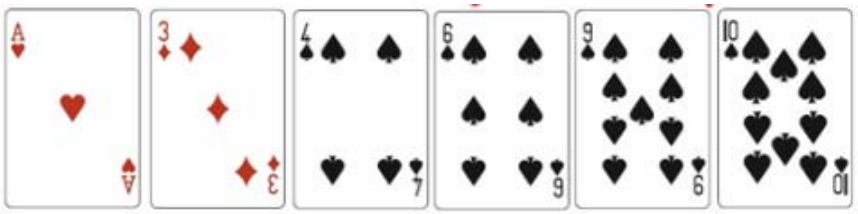


Figure 10: A sorted hand of cards

Sorting is a helpful feature. Be it a dictionary where we want to keep the word sorted so that searching a word in the dictionary is easy or something like a table with salary to see who has the highest salary.

If we want to define sorting formally, then we could define sorting as follow:

Sorting is arranging the elements in a list or collection in increasing or decreasing order of some property. The list should be homogeneous, that is all the elements in the list should be of the same type.

Efficient sorting is important to optimize the use of other algorithms that require sorted lists to work correctly. Efficient sorting is also important to normalize and standardize data that can be presented in more than one way. This is often the case when we are working with databases. Sorting makes searching easier. Furthermore, sorting produces a human-readable output.

When we study sorting algorithms, we use most of the times a list of integers, and typically we sort the list of integers in increasing order of value. Suppose we have a list of integers like in Figure 11.

3	4	10	5	7

Figure 11: Integers in unsorted order

Then sorting it in increasing order of value means rearranging the elements like in Figure 12.

3	4	5	7	10

Figure 12: Integers in an increasing order of value

Sorting the list in decreasing order of value means an order like in Figure 13.

10	7	5	4	3

Figure 13: Integers in decreasing order of value

And as we have said in the definition, we can sort the list on any property. What if we want to sort this list on the basis of let's say, increasing number of factors. This means the integer with a lesser number of factors is towards the beginning of the list.

3	5	7	4	10

Figure 14: Integers in an increasing order of number factors

As we can see in Figure 14, 3 has got only 2 factors, 1 and 3 itself. 4 has got 3 factors, 1, 2 and 4 itself. 5 has also got 2 factors, 1 and 5 itself. 7 has got two 2 factors, 1 and 7 itself, 10 has got 4 factors, 1, 2, 5, and 10 itself.

As we can see, a sorted list is a permutation of the original list. When we sort a list, we just rearrange the elements. Sorted data is good, not just for presentation or manual retrieval of information, even when we are using the computational power of machines, sorted data is really helpful.

7.2. Classification

Sorting algorithms used in computer science are often grouped into classes like the computational complexity of element comparisons in terms of the size of the list. Computational complexity, in this case, refers to the worst, average, and the best behavior of element comparisons. The other classes are memory usage, the use of other computer resources, and recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).

Stability is another class. Stable sorting algorithms maintain the relative order of records with equal keys (i.e., the values). We say that a sorting algorithm is *stable* if the relative order of the elements with equal values in the input is maintained in the output. If all values are different then this distinction is not necessary. But if there are equal values, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

A sorting algorithm is said to be *in place* if the algorithm does not need extra memory to store temporary data. These algorithms do not need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.

7.3. Bubble sort

Bubble sort is also known as the *ripple* sort and belongs to the Brute force strategy. Bubble sort is a straightforward and simplistic method of sorting data that is used in computer science education. It is an in place and stable algorithm. The bubble sort gets its name because elements tend to move up into the correct order like bubbles rising to the surface. Bubble sort compares the first two adjacent elements of an array and swaps them if they are out of order. It continues doing this for each adjacent element until it ends up to the last entry. Formally we can say that the bubble sort algorithm starts at the beginning of a data set. It then compares the first two adjacent elements, and if the first element is greater than the second one, then it swaps them. The algorithm continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.

7.3.1. Analysis

When we consider the performance of bubble sort for a list with n elements (this is a list with the length n), we see that the number of key comparisons is $n-1$ for one pass to get the largest element at the end of the list. The next pass bubbles up the second largest element. The performance is then $(n-1)+(n-2)$ for two passes. For n passes, the performance is the following sum

$$(n-1)+(n-2)+\cdots+1. \qquad \text{eq. (1)}$$

This is a mathematical series which sums to

$$\frac{n(n-1)}{2} \qquad \text{eq. (2)}$$

which is equal to

$$\frac{n^2}{2} - \frac{n}{2} \in O(n^2). \qquad \text{eq. (3)}$$

For a list with a large number of elements (i.e., n is large) the number of key comparison is in the order of n^2, because n^2dominates. In the best case, we need just one pass to sort the list. So the number of key comparisons is then in the order of n. We say that the time complexity in the best case is $O(n)$. The time complexity in the worst case and the average case is $O(n^2)$.

7.4. Merge sort

Merge sort is a sorting algorithm that sorts data items into ascending or descending order, which comes under the category of comparison-based sorting. The algorithm was invented by John von Neumann in 1945 [11] and belongs to the divide-and-conquer technique.

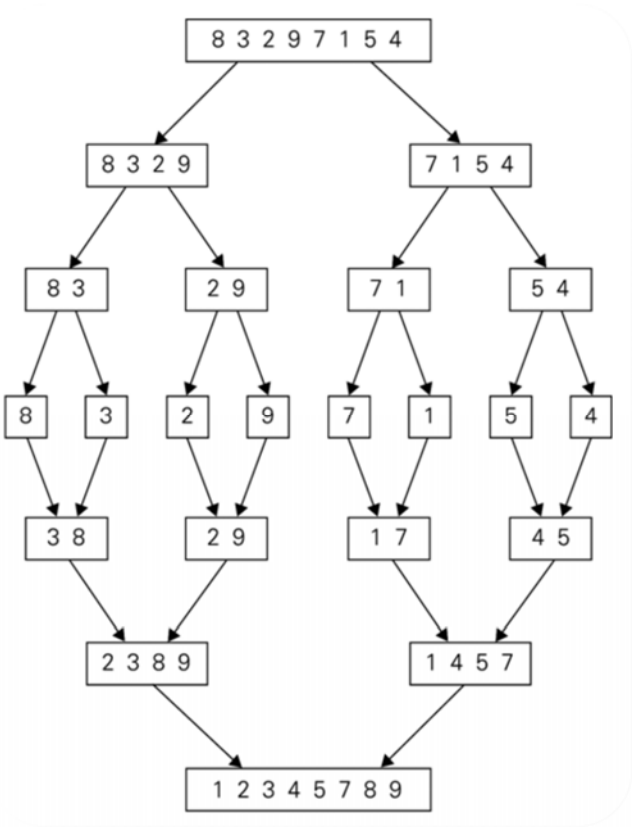


Figure 15: Example of merge sort [1]

In the first step, merge sort divides an unsorted list into two halves or two lists if the number of elements is odd. This is repeated until there are *N* sub-lists, each having one element because a list of one element is considered sorted. In the second step, each of them is sorted recursively

to produce newly sorted sub-lists. And in the last step, the two smaller sorted lists are merged into a single sorted list.
The last step is the merging step in which two sorted arrays or lists are being merged. This merging step can be done as follows. Two pointers, i and j, are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed. After that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted. The remaining elements of the other array are then copied to the end of the new array, which results in a sorted array.

7.4.1. Analysis

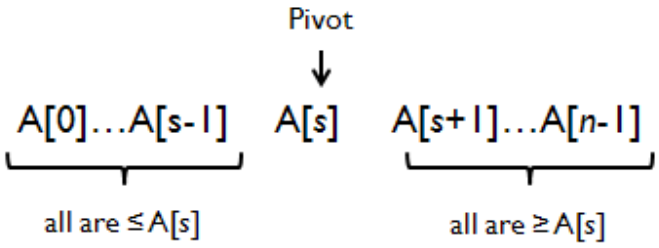
When we consider the performance of merge sort we see that the time complexity in the best case and the worst case is $O(n\ log(n))$. This is faster than bubble sort, insertion sort, and selection sort, which have a $O(n^2)$. This does not mean that merge sort will always be faster for all lists. For example, insertion sort might be the fastest sort for all lists smaller than 5 elements. While in practice, merge sort is usually faster for lists as small as 50 elements. But this extra speed does not come without a price. Unlike other sorts, which take a list and modify the list in place, until we get a sorted list, merge sort needs some additional space to merge 2 lists together. This space is a bit of a price.

7.5. Quicksort

Quicksort (sometimes called *partition-exchange sort*) is another important sorting algorithm besides merge sort and bubble sort. The algorithm was developed by Tony Hoare in 1959 [12] and is still a commonly used algorithm for sorting. The quicksort algorithm is based on the divide-and-conquer technique and is one of the fastest and efficient sorting algorithms in practical scenarios. It can be about two or three times faster than merge sort and heapsort when it is implemented well. Unlike merge sort, which divides its input's element according to their *position* in the array, quicksort divides the elements according to their *value*. It can sort items of any type for which a "less than" relation is defined. The quicksort sorting algorithm is an in-place sorting algorithm, requiring small additional amounts of memory to perform the sorting.

Quicksort rearranges elements of a given array $A[0..n-1]$ to achieve its *partition*. Partition is a situation where all the elements before some position s are smaller than or equal to $A[s]$ and all elements after position s are greater than or equal to $A[s]$:

eq. (4)



Obviously, after a partition has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two sub-arrays of the elements preceding and following $A[s]$ independently (e.g., by the same method). Because of its guiding role, we call the element $A[s]$ the *pivot*.

7.5.1. Partitioning

As we mentioned before, the partition is a re-arrangement of the list's elements so that all elements to the left of the pivot are smaller than or equal to the pivot, and all the elements to the right of the pivot are greater than or equal to the pivot. There are several alternative procedures for rearranging elements to achieve a partition. We use here an efficient method based on two scans of the sub-array: a left-to-right scan and a right-to-left scan, each comparing the sub-arrays element with the pivot.

We start the partition step by selecting an element to be the pivot. Ideally, the pivot should be the median element. But it is chosen randomly to make the selection process quicker. Then we create two index pointers, e.g., i and j. Index pointer i starts from the beginning of the array and index pointer j starts from the end of the array. These pointers are used to mark the positions at the elements that we are scanning. Now we will scan the sub-arrays from both ends. The index pointer i does the left-to-right scan. It scans elements smaller than the pivot to be in the first part of the sub-array. This index pointer skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The index pointer j does the right-to-left scan. It scans elements greater than the pivot to be in the second part of the sub-array. This index pointer skips over elements that are greater than the pivot and stops on encountering the first element smaller than or equal to the pivot. We swap the elements after both the index pointers stops. We continue doing this until the index pointers pass each other. Once that happens, we have the array divided into two partitions: one partition with the elements smaller than the pivot, and one partition with the elements greater than the pivot. That finishes the partition step. We repeat the partition procedure for the two separate partitions until there remains only one element in the list.

7.5.2. Analysis

With quicksort, we have an efficient in-place algorithm to partition a list. The partition can be done in the constant amount of extra memory using only some temporary variables. Note that unlike merge sort we do not need to create auxiliary arrays and to copy the elements in the new arrays with quicksort. In quicksort, we can work on the same arrays. The entire work happens in the division stage, with no work required to combine the solutions to the subproblems. We just have to keep track of the start and the end index of the segments.

The time complexity of the quicksort algorithm is $O(n\ log(n))$ in average case and best case for an array with length n. The time complexity in the worst-case is $O(n^2)$. But as we said before, quicksort is an in place sorting algorithm, which take almost constant amount of extra memory. Despite having a worst-case running time of $O(n^2)$ quicksort is pretty fast and efficient in practical scenarios. The worst-case running time of quicksort is almost always avoided by using what we call a randomized version of quicksort (i.e., a quicksort version where we take a random element as a pivot). Randomized quicksort gives us $O(n\ log(n))$ running time with very high probability.

7.6. Exercises

1) What do we mean by sorting?

2) Why is sorting useful?

3) What do we mean by “A sorted list is a permutation of the original list”?

4) What is ripple sort?

5) Explain how bubble sort works.

6) Explain how merge sort works.

7) Explain why the extra speed of merge sort does not come without a price.

8) Explain the difference between merge sort and quicksort when these algorithms divide the elements in an array?

9) **How to use merge sort?**
   Use merge sort to alphabetize the following list: E, X, A, M, P, L, E.

Show how you sorted this list using tree notation.

10) **Sort and search***

Given is the following list: t, f, w, c, q, g, p, r, o.

a. Use quicksort to sort this list. Show how you got your answer.

b. Show using binary search how you would look up the letter f.

c. Explain why linear search is faster in this particular case and show this.

8. Greedy technique

The *Greedy technique* is a straightforward and a simple technique, which suggests to make a best choice at each stage from the available alternatives in the hope that a series of local optimal choices ultimately lead to a global optimal solution to the big problem. This is the so-called *Greedy choice property*. A Greedy algorithm does not, in general, produce an optimal solution, but nonetheless, a Greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

Consider the following *change-making problem* faced by millions of cashiers all over the world: give change for a specific amount n with the least number of coins of the denominations $d_1 > d_2 > \ldots > d_m$ used in that locale [1]. For example, let $d_1 = 50$, $d_2 = 20$, $d_3 = 10$, and $d_4 = 5$ be Euro cents. How would you give change with coins of these denominations of, say, 95 Euro cents? Most people would come up with the following answer: 1 x 50 Euro cents, 2 x 20 Euro cents, and 1 x 5 Euro cents. This strategy of making a sequence of best choices among the current available alternatives is called the *Greedy thinking*. In the first step "Greedy" thinking leads to giving one coin of 50 Euro cents because it reduces the remaining amount the most, namely, to 45 Euro cents. In the second step, we had the same coins at our disposal, but we could not give 50 Euro cents because it would have violated the problem's constraints. So, the best selection in this step was two 20 Euro cents. This reduced the remaining amount to 5 Euro cents, which can be reduced to zero by giving 5 Euro cents. The question is now, is this the optimal solution for this problem? And the answer is "Yes, it is".

To use the Greedy technique the algorithm should meet the following requirements for each step, which is the central point of the Greedy technique.

- Feasible: i.e., the choice made must satisfy the problem's constraints.
- Local optimal: i.e., it must be the best local choice among all feasible choices available on that step.
- Irrevocable: i.e., once the choice is made, it cannot be changed on subsequent steps of the algorithm. In other words, a greedy algorithm never reconsiders its choices, whatever situation may arise later.

Despite the Greedy technique is straightforward and simple, it has one disadvantage. It mostly (but not always) fails to find the globally optimal solution, because it usually does not operate exhaustively on all the data. Greedy algorithms are best suited for simple problems (for example for the change-making problem).

If a Greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods

like dynamic programming. Examples of such algorithms are the Dijkstra's algorithm for finding single-source shortest path and Prim's algorithm for finding minimum spanning trees.

Greedy algorithms are often used in ad hoc mobile networking to efficiently route packets with the fewest number of hops and the shortest delay possible. They are also used in business intelligence (BI), artificial intelligence (AI), machine learning, and programming.

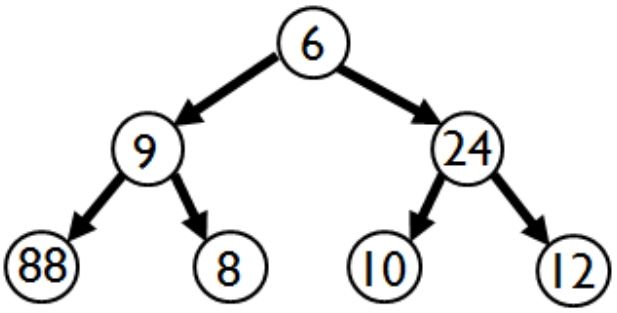
8.1. Exercises

1) What is the disadvantage of the Greedy technique and explain why it is a disadvantage?

2) Give an example of the change-making problem for which the Greedy algorithm does not yield an optimal solution.

3) Explain how you would give change (making use of a Greedy algorithm) of 36 cents using only coins with the values 1, 5, 10, and 20.

4)



Explain which path the Greedy algorithm will choose if the goal is to reach the largest sum in the tree.

9. Introduction to graph theory

9.1. Introduction

In computer science, graphs are used to represent networks of communication, data organization, computational devices, the flow of computation, etc. One practical example: The link structure of a website could be represented by a directed graph. The vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B [13]. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science.

9.2. Graphs

Informally a graph can be thought of as a set of *vertices* or *nodes* and a set of *edges* or *arcs* which connect pairs of vertices. See Figure 16 for an example of a graph.

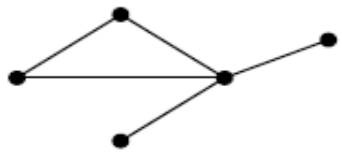


Figure 16: An undirected graph

Formally, a graph $G = (V, E)$ is defined by a pair of two sets:

1) a finite set V of items called *vertices,* and
2) a set E of pairs of these items called *edges.*

Now suppose we have two vertices u and v, i.e., $V = \{u, v\}$. Then the pair of vertices (u, v) is the same as the pair (v, u) if these pairs of vertices are unordered. We say that the vertices u and v are *adjacent* to each other and that they are connected by the *undirected edge* (u, v). We call the vertices u and v *endpoints* of the edge (u, v). We write $e = (u, v)$ if edge $e \in E$ joints $u, v \in V$. A graph G is called *undirected* if every edge $e \in E$ is undirected. This means that every edge in graph G is undirected. An undirected graph is *symmetric*. Figure 16 is an example of an undirected graph.

We say that the edge $e = (u, v)$ is *directed* from the vertex u to vertex v if a pair of vertices (u, v) is not the same as the pair (v, u). Vertex u is called the edge's *tail* and vertex v is called the edge's *head*. A graph G whose every edge e is directed is called a *directed graph*. Figure 17 shows an example of a directed graph G. Directed graphs are also called *digraphs* and are *asymmetric*.

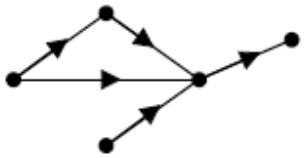


Figure 17: A directed graph

A graph G with both directed and undirected edges is called a *mixed graph.* Figure 18 is an example of a mixed graph.

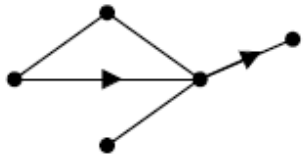


Figure 18: A mixed graph

Vertices of a graph are normally labeled with letters, integer numbers, or, if an application calls for it, character strings. The graph shown in Figure 19 has six vertices and seven undirected edges:

$V = \{a, b, c, d, e, f\},$
$E = \{(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)\}.$

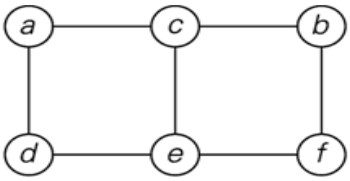


Figure 19: An undirected graph with labeled vertices

9.3. Representations of graphs

Graphs in the computer science are usually represented by the so-called *adjacency matrices*. The adjacency matrix of a graph with n vertices is a $n \times n$ Boolean matrix with one row and one column for each of the graph's vertices. In this matrix the element in the i-th row and the j-th column is equal to 1 if there is an edge from the i-th vertex to the j-th vertex, and equal to 0 if there is no such edge. For example, the graph of Figure 19 is represented by the adjacency matrix given in Figure 20.

	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

Figure 20: Adjacency matrix

Note that the adjacency matrix of an undirected graph is always symmetric. This means that $A[i,j] = A[j,i]$, where $A[i,j]$ is a way to represent a matrix mathematically.

9.4. Some examples of graphs

In this section, we will briefly discuss some examples of graphs.

9.4.1. Weighted graphs

We call a graph a *weighted graph* if the graph has numbers assigned to its edges. In Figure 21 we see an example of a weighted graph.

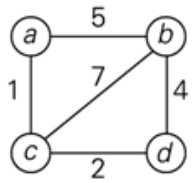


Figure 21: Weighted graph

The numbers assigned to the edges are called *weights* or *costs*. Weighted graphs are important for a lot of real-life applications, such as finding the shortest path between two points in a transportation or communication network. Dijkstra's and Prim's algorithms use weighted graphs. We will discuss these algorithms in Section 10. The matrix of a weighted graph is called a *weighted matrix* or a *cost matrix* (see Figure 22 for an example).

	a	b	c	d
a	∞	5	1	∞
b	5	∞	7	4
c	1	7	∞	2
d	∞	4	2	∞

Figure 22: Weighted matrix

A weighted matrix simply contains the weight of the edge from the i-th to the j-th vertex if there is such an edge and an infinity symbol (∞) if there is no such edge. So, for example, in Figure 21 the edge from vertex a to vertex b has a weight of 5 in the graph, then we write 5 in

the matrix from vertex a to vertex b (see Figure 22). There is no edge from vertex a to vertex d in the graph, so we write an infinity symbol, ∞, in the matrix from vertex a to vertex d. We say that Figure 22 is the weighted matrix of the weighted graph of Figure 21.

9.4.2. Simple graphs

A graph is called a *simple* or *strict graph* (see Figure 23) if it is an undirected graph with no weights, no loops, and no more than one edge between any two different vertices. A simple graph may be either connected or disconnected.

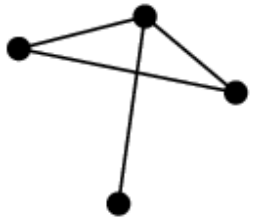


Figure 23: Simple graph

An edge of a graph which joins a vertex to itself is called a *loop*.

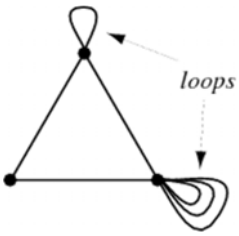


Figure 24: Graph with loops [14]

A simple graph cannot contain any loops. Unless stated otherwise, the unqualified term “graph” usually refers to a simple graph.

9.4.3. Multigraph

Multiple edges are two or more edges connecting the same two vertices within a simple graph. Such a simple graph with multiple edges is called a *multigraph* (see Figure 25).

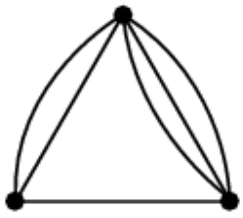


Figure 25: Multigraph

9.4.4. Connected and disconnected graphs

A graph is called *connected* if there is, for any two given vertices u and v, a path from u to v. Otherwise, it is called *disconnected*. The graph in Figure 27 is disconnected as there is no path from, for example, vertex e to vertex a or from vertex a to vertex d.

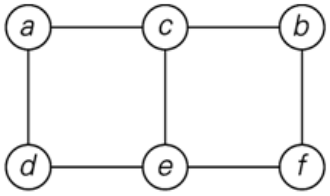


Figure 26: Undirected connected graph

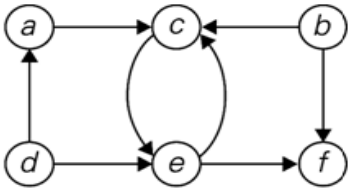


Figure 27: Directed disconnected graph

A *path* in the graph theory is defined as a finite or infinite sequence of edges which connect a sequence of vertices which are all distinct from one another [15].

9.4.5. Connected component

A *connected component* of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph [16]. The graph in Figure 28 consists of three components and is called the supergraph. Note that the components are also graphs which are called the subgraphs. The path from, for example, u to v is shown in red.

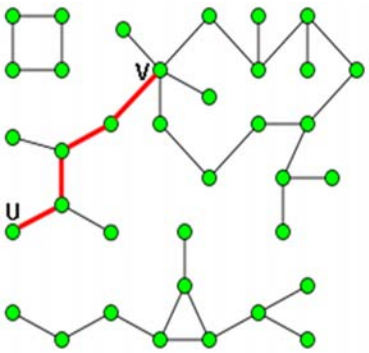


Figure 28: Graph with three connected components

9.4.6. Complete graphs

A *complete graph* on n vertices (denoted as K_n) is a graph in which all vertices are mutually connected by means of direct edges. A complete graph is simple and undirected. To make a graph with n vertices complete you need $n-1$ edges on each vertex. So, for example, to make a graph with 3 vertices complete, you need 2 edges from each vertex. Likewise, you need 3 edges from each vertex to make a graph with four vertices complete (see Figure 29 for examples). We say that K_n is a graph of *degree* $n-1$.

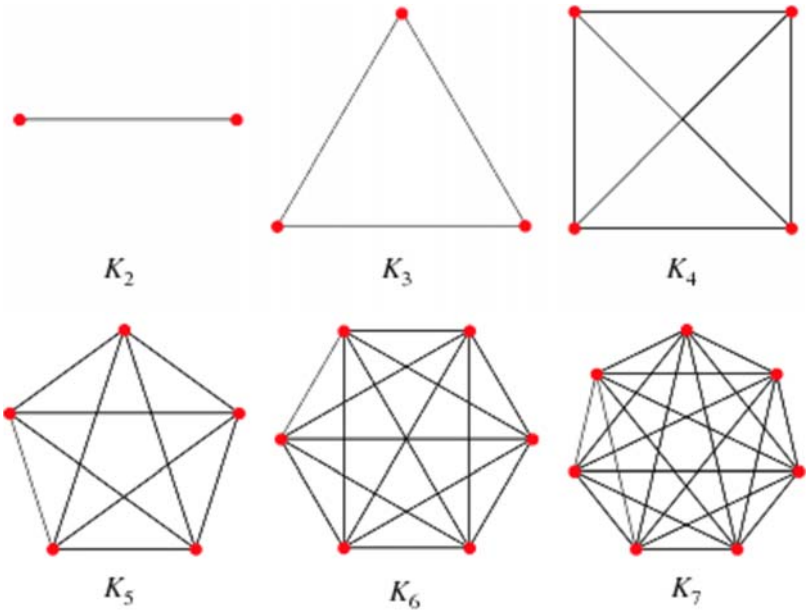


Figure 29: Complete graphs

A complete graph with n vertices has $\frac{n(n-1)}{2}$ edges.

9.4.7. Cycle graphs

A *cycle graph* or a *circular graph* on n vertices (denoted as C_n) is a simple connected graph with n vertices where each vertex is connected with two other vertices. Note that the cycle graph has a close path that starts and ends at the same vertex and does not traverse the same edge more than once. Figure 30 illustrates a cycle graph in which the vertices f, h, i, g, f is a cycle. A graph with no cycles is said to be *acyclic*.

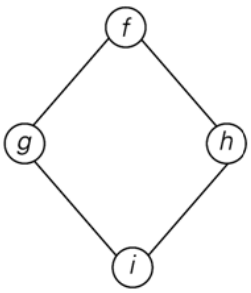


Figure 30: Cycle graph

The number of vertices in C_n equals the number of edges. Every vertex has degree 2, which means that every vertex has exactly two edges with it.

9.4.8. Trees

A *tree* is a simple, undirected, connected, acyclic graph (see Figure 31). In a tree, any two vertices are connected by exactly one path. A tree with n vertices has $n - 1$ edges. A graph that has no cycles but is not necessarily connected is called a *forest* (see Figure 32). This means each of its connected components is a tree.

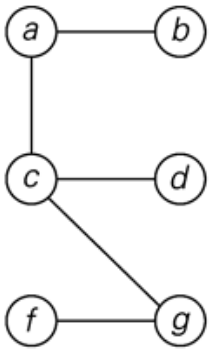


Figure 31: Tree

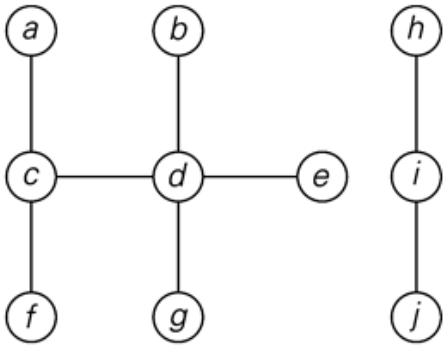


Figure 32: Forest

9.4.9. Eulerian graph

We say that a path is an *Eulerian path* or *Eulerian trail* if we can find a path in a connected graph by visiting every edge exactly one time. To put it in other words, an Eulerian path is a path in a connected graph which contains all the edges of the graph. An *Eulerian cycle* or an *Eulerian circuit* is an Eulerian path which starts and ends on the same vertex, which means that we have a circuit in a graph which contains all of the edges of the graph. We call a graph an *Eulerian graph* if it has an Eulerian cycle. The pentatope graph in Figure 33 is an example of an Eulerian graph because it comprises an Eulerian cycle as we can find a circuit in this graph which contains all the edges of the graph.

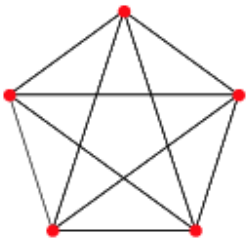


Figure 33: Pentatope graph

The Eulerian graph is a special kind of graph which was identified by the Swiss mathematician and physicist Leonhard Euler (1707 - 1783) [17] following his solution to the problem of the

famous Seven Bridges of Königsberg in 1736. Mathematically the problem can be stated like this:

Given the graph of the Königsberg Bridges (Figure 34), is it possible to construct a path (or a cycle) which visits each edge exactly once?

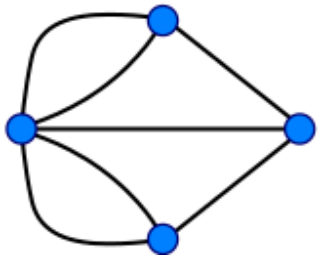


Figure 34: Graph of the Königsberg Bridges

Euler proved that if all vertices in a graph have an even degree, then the graph contains an Eulerian circuit. He stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit.

9.4.10. Hamiltonian graph

A *Hamiltonian path,* also called a *Hamilton path* or *traceable path*, is defined as a path in an undirected or directed graph that visits each vertex exactly once. A *Hamiltonian cycle*, also called *Hamiltonian circuit*, *Hamilton cycle*, or *Hamilton circuit*, is a graph cycle (i.e., closed loop) through a graph that visits each vertex exactly once [18]. To put it in other words, the Hamiltonian cycle is a path that visits all the graph's vertices exactly once before returning to the starting vertex for this graph. We call a graph a *Hamiltonian graph* or a *Hamilton graph* if it possesses a Hamiltonian cycle.

Hamiltonian paths and cycles are named after the Irish physicist, astronomer, and mathematician Sir William Rowan Hamilton (1805 - 1865) [19] who invented the *Icosian Game* in 1857 [20]. The mathematical game was played on a circular wooden game board on which a graph (See Figure 35) was carved.

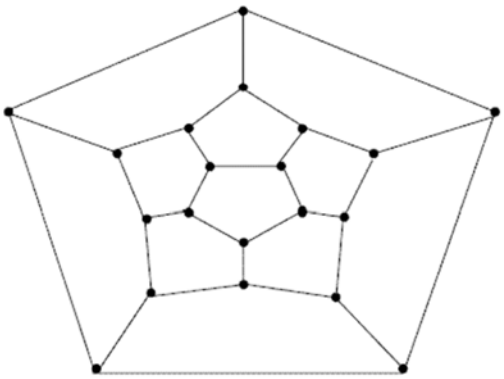
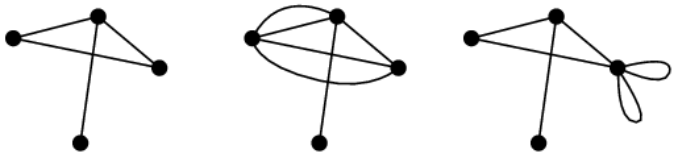


Figure 35: Graph for the Icosian Game

The game's object was finding a Hamiltonian cycle along the edges of a dodecahedron (See Figure 35) such that every vertex was visited a single time, and the ending point is the same as the starting point.

9.5. Exercises

1) Given the following graphs



a b c

Explain what kind of graph these are?

2) Explain whether the graph of Königsberg Bridges is an Eulerian graph or not.

3) Find a Hamiltonian circuit for the graph of the Icosian Game.

4) What is the difference between a connected graph and a connected component?

5) **Weighted matrix to weighted graph**

Given is the following weighted matrix:

	A	B	C	D	E	F
A	-	2	3	4	-	-
B	-	-	1	-	-	7
C	3	2	-	-	-	6
D	-	-	-	-	8	-
E	-	-	-	7	-	4
F	-	6	6	-	-	-

a) Draw the corresponding weighted graph.

b) Explain what kind of graph this is.

c) Show and explain what the shortest path is from A to F.

d) Is this a connected graph? Motivate your answer.

e) Is there a path to get from D to A? Motivate your answer.

10. Graph algorithms

10.1. Dijkstra's algorithm

In this section, we consider the single-source shortest path problem. The single-source shortest path problem is to find the shortest path (i.e., path with the lowest cost) from one designated node (vertex), called the source, to *every* other node in a weighted connected graph (e.g., a road network) such that the sum of the weights of its constituent edges is minimized. Dijkstra's algorithm is a graph search algorithm which solves the single-source shortest path problem and belongs to the Greedy technique. It works in directed or undirected graphs and only works in graphs with non-negative edge weights (e.g., distance or time). The algorithm is developed by the Dutch computer scientist Edsger W. Dijkstra (1930 – 2002) in 1956 [21].

Dijkstra's algorithm exists in many variants. The original variant found the shortest path between two nodes [22], but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a *shortest path tree* [4]. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path algorithm is widely used in network routing protocols and as a subroutine in other graph algorithms [23].

Dijkstra's algorithm works as follows [23]: let the node at which we are starting be called the *initial node*. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the currently assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then update the distance to 8. Otherwise, keep the current value.

[4] Given a connected, undirected graph G, a shortest path tree rooted at node v is a spanning tree T of G, such that the path distance from root v to any other node u in T is the shortest path distance from v to u in G.

4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

10.2. Prim's algorithm

We need to define some terms before we consider Prim's algorithm in this section.

DEFINITIONS

A ***spanning tree*** of a connected graph is its connected acyclic sub-graph (i.e., a tree) that contains all the vertices of the graph.

A ***minimum spanning tree*** of a weighted connected graph is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges.

The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

Suppose we have the following problem which may arise in several practical situations: connect n points in the cheapest possible way so that there will be a path between every pair of points. In this problem the points can be represented by vertices of a graph, the possible connections can be represented by the graph's edges, and the connection costs can be represented by the edge weights. Then the problem can be posed as the minimum spanning tree problem.

Prim's algorithm finds a *minimum spanning tree* for a connected weighted undirected graph and belongs to the Greedy technique. It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was originally discovered in 1930 by Czech mathematician Vojtěch Jarník and later independently rediscovered and republished by computer scientists Robert Clay Prim in 1957. It was later rediscovered by Edsger W. Dijkstra in 1959. The algorithm is sometimes referred to as the *DJP algorithm* or the *Jarník algorithm* [24].

Prim's algorithm operates by constructing the minimum spanning tree through a sequence of expanding sub-trees. The initial sub-tree in such a sequence consists of a single vertex, which is chosen arbitrarily from the set V of the graph's vertices. In each iteration, we expand the current tree by adding the cheapest possible connection from the tree to another vertex.

The algorithm may informally be described as performing the following steps:

1. Select any vertex to be the first of the tree T.
2. Consider which edge connects vertices in T to vertices outside T. Pick the edge with the minimum weight (if there are more than one, then choose any). Add this edge and vertex to T.
3. Repeat step 2 until T contains every vertex of the graph.

For any connected weighted undirected graph, Prim's algorithm will always yield a minimum spanning tree [25].

10.3. Kruskal's algorithm

In the previous section, we considered Prim's algorithm that "grows" a minimum spanning tree through a greedy inclusion of the nearest vertex to the vertices already in the tree. In this section, we will consider Kruskal's algorithm.

Kruskal's algorithm is named after the American mathematician, statistician, computer scientist and psychometrician Joseph Bernard Kruskal (1928 - 2010). He discovered the algorithm when he was a second-year graduate student [26]. Kruskal's algorithm is, like Prim's algorithm, also a greedy algorithm in the graph theory that finds a minimum spanning tree for a connected weighted graph. If the graph is not connected, then it finds a minimum spanning forest (i.e., a minimum spanning tree for each connected component). The algorithm looks at a minimum spanning tree of a connected weighted graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but not necessarily connected on the intermediate stages of the algorithm [1].

The algorithm begins by sorting the graph's edges in non-decreasing order of their weights, creating a sorted list of edges. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

The algorithm may informally be described as performing the following steps:

1. Remove all loops.

2. Remove all parallel edges between two vertices except the one with least weight.
3. Create an edge table. An edge table consists of all the edges along with their weight in an ascending order, i.e. a sorted list of edges.
4. Select the edge with the minimum weight.
5. Select the next edge with the minimum weight that does not form any cycle with previously selected edges. You may have a forest now.
6. Repeat step 5 until T contains every vertex of the graph.

10.3.1. Comparison of Prim's and Kruskal's algorithm

At first sight, both Prim's and Kruskal's algorithm seems to operate in the same way. Both Prim's algorithm and Kruskal's algorithm are greedy algorithms for finding the minimum spanning tree. But when we compare both algorithms we see some differences between Prim's and Kruskal's algorithm.

a) For Prim's algorithm, the graph *must* be connected, but that is not true in the case of Kruskal's algorithm.
b) Prim's algorithm grows only one tree, while Kruskal's algorithm grows a collection of trees (a forest).
c) In Prim's algorithm, the next edge in the minimum spanning tree shall be the cheapest edge in the *current vertex*. In Kruskal's algorithm, we shall choose the cheapest edge, but it may not be in the current vertex.
d) Prim's algorithm is found to run faster in dense graphs with more number of edges than vertices, whereas Kruskal's algorithm is found to run faster in sparse graphs.
e) The running time of Kruskal's algorithm is $O(E\ log\ E)$, or equivalently, $(E\ log\ V)$, where E is the number of edges and V is the number of vertices. This can be reached by simple data structures[5]. The time complexity of Prim's algorithm depends on the data structures used for the graph and for ordering the edges by weight. A simple implementation of Prim's algorithm, using an adjacency matrix, requires $O(|V|^2)$ running time.

10.4. Exercises

1) Name three conditions for the graph on which Dijkstra's algorithm works.

[5] A data structure is a particular way of organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications. Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Examples of data structures are *arrays* or *lists*, *graphs*, *trees*, *class*, and *records* [30].

2) What is the shortest path tree?

3) What are the differences between Dijkstra's and Prim's algorithm?

4) What is the single-source shortest path problem?

5) Name two variants of the Dijkstra's algorithm.

6) Give a practical example of the Dijkstra's algorithm.

7) **The shortest route from Mazalan to Aasteria***
On the planet, Sukhi in the land of Hyster two travelers want to go from Mazalan to Aasteria. But as their flying credits are fully depleted, they must travel over land by car. There are different routes they can take to Aasteria. They have an almanac with the distances between some of the cities of Hyster. The distances below are shown in Hysterian units (Hu). The travelers must use these distances to find the shortest route possible between Mazalan and Aasteria.

Mazalan – Dusit	118 Hu
Mazalan – Rachburi	136 Hu
Mazalan – Eulesy	78 Hu
Eulesy – Bramin	136 Hu
Eulesy – Rachburi	110 Hu
Rachburi – Dusit	48 Hu
Rachburi – Aasteria	370 Hu
Rachburi – Bramin	96 Hu
Dusit – Aasteria	416 Hu
Bramin – Aasteria	240 Hu

a) Show using Dijkstra's algorithm what the shortest route is between Mazalan and Aasteria.

b) Draw the minimum spanning tree of the graph given in the table above using Prim's algorithm.

c) What is the weight of the minimum spanning tree?

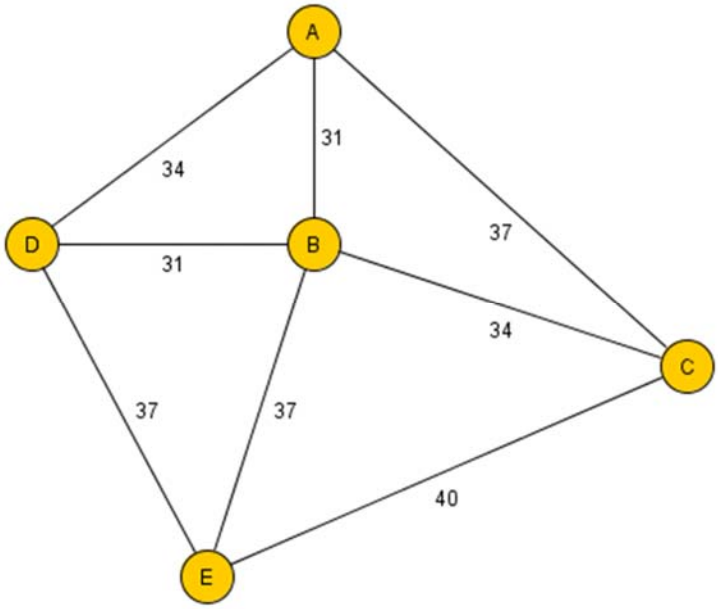
d) Draw the minimum spanning tree of the graph given in the table above using Kruskal's algorithm.

8) **The traveling salesman problem ***

A traveling salesman has to go to Amsterdam, Alkmaar, The Hague, Hilversum, Utrecht, and IJmuiden. What is the shortest route he can take visiting all these places? This question is an old problem known as the *traveling salesman problem*. Using brute force and trying every single solution is an option, but if the number of places to visit grows, the number of solutions to go through grows as well and becomes prohibitive. Also choosing the route so you visit every place once only is not easy to solve. As a matter of fact, this problem is an unsolved problem. But finding an answer to this question is a matter of importance in modern technology, think for example of routing on a microchip.

It is possible though to solve a simplified form of the traveling salesman problem with the Greedy algorithm.

Assume an agent of a commercial courier service has to deliver five packages to five different addresses. The agent wants to be home as early as possible to celebrate his daughter's birthday, so he needs to find the shortest route to deliver his freight. See the graph pictured below.



In the graph A to E denote the addresses, with A being the home of the agent and B to E the delivery places. The edges between the addresses represent the roads, the numbers are the distances. The agent delivers the packages in a certain order and tries to travel the shortest distance between the addresses.

a) Determine the shortest route to travel for the agent if he wants to be home early. Show and explain your answer.

b) Are there other routes the agent can travel which are of the same length or shorter than the route mention in a) in order to be home early? If so, show and explain.

c) Assume the agent must deliver 42 packages to 25 addresses. How many different routes can he travel? Explain your answer.

References

[1] A. Levitin, in *Introduction to the Design & Analysis of Algorithms*, 3rd ed., London, Pearson Education Limited, 2012.

[2] D. E. Knuth, in *Selected Papers on Computer Science*, Stanford, California: Cambridge University Press, 1996.

[3] R. Kiriluk-Hill, "Fairness Of Hunterdon Central High School's Courtesy Busing Questioned," [Online]. Available: http://www.nj.com/hunterdon/index.ssf/2009/03/fairness_of_hunterdon_central.html. [Accessed 11 03 2015].

[4] W. G. G. Centre, "Farm Shop," [Online]. Available: http://www.woodcotegreen.com/Garden-Centre/farm-shop.aspx. [Accessed 11 03 2015].

[5] A. Adam, "Study Habits," [Online]. Available: http://www.study-habits.com/how-to-prepare-for-an-exam. [Accessed 11 03 2015].

[6] "Big O notation," [Online]. Available: http://en.wikipedia.org/wiki/Big_O_notation. [Accessed 05 05 2015].

[7] "Time complexity," [Online]. Available: http://en.wikipedia.org/wiki/Time_complexity. [Accessed 05 05 2015].

[8] I. K. Lundqvist, "Massachusetts Institute of Technology," [Online]. Available: http://web.mit.edu/. [Accessed 12 05 2015].

[9] E. Rowell, "Know Thy Complexities!," [Online]. Available: http://bigocheatsheet.com/. [Accessed 16 05 2015].

[10] "Search algorithm," [Online]. Available: http://en.wikipedia.org/wiki/Search_algorithm. [Accessed 24 04 2015].

[11] "Merge Sort," [Online]. Available: https://en.wikipedia.org/wiki/Merge_sort. [Accessed 29 08 2017].

[12] "Tony Hoare," [Online]. Available: https://en.wikipedia.org/wiki/Tony_Hoare. [Accessed 29 08 2017].

[13] A. Bhulai, Dynamic website optimization through autonomous management of design patterns, Amsterdam: Universal Press, 2011, pp. 190-194.

[14] "Graph loop," Wolfram Mathworld, [Online]. Available: http://mathworld.wolfram.com. [Accessed 07 06 2015].

[15] "Path," [Online]. Available: http://en.wikipedia.org/wiki/Path_%28graph_theory%29. [Accessed 08 06 2015].

[16] "Connected component," [Online]. Available: http://en.wikipedia.org/wiki/Connected_component_%28graph_theory%29. [Accessed 08 06 2015].

[17] "Leonhard Euler," [Online]. Available: https://en.wikipedia.org/wiki/Leonhard_Euler. [Accessed 12 06 2015].

[18] S. Skiena and S. Pemmaraju, Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, United States of America: Cambridge University Press, 2003.

[19] "William Rowan Hamilton," [Online]. Available: https://en.wikipedia.org/wiki/William_Rowan_Hamilton. [Accessed 12 06 2015].

[20] "Icosian game," [Online]. Available: https://en.wikipedia.org/wiki/Icosian_game. [Accessed 12 06 2015].

[21] "Edsger W. Dijkstra," [Online]. Available: http://en.wikipedia.org/wiki/Edsger_W._Dijkstra. [Accessed 19 05 2015].

[22] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* vol. 1, p. 269–271, 1959.

[23] "Dijkstra's algorithm," [Online]. Available: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. [Accessed 24 05 2015].

[24] "Robert C. Prim," [Online]. Available: http://en.wikipedia.org/wiki/Robert_C._Prim. [Accessed 29 05 2015].

[25] "Prim's_algorithm: Proof_of_correctness," [Online]. Available: http://en.wikipedia.org/wiki/Prim%27s_algorithm#Proof_of_correctness. [Accessed 30 05 2015].

[26] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem.," *Proceedings of the American Mathematical Society,* vol. 7, pp. 48-50, 1956.

[27] R. L. Graham, D. E. Knuth and O. Patashnik, Concrete Mathematics, 2nd, Ed., Massachusetts: Addison-Wesley, 1994, p. Ch. 3.1.

[28] K. Cherry, "What is a heuristic?," About, [Online]. Available: http://psychology.about.com/od/hindex/g/heuristic.htm. [Accessed 19 05 2015].

[29] "Heuristic," vocabulary.com, [Online]. Available: http://www.vocabulary.com/dictionary/heuristic. [Accessed 19 05 2015].

[30] "Data structure," [Online]. Available: https://en.wikipedia.org/wiki/Data_structure. [Accessed 25 05 2015].

Index