COMP11012: Computational Thinking Unit Handbook

Sean Bechhofer Uli Sattler Andrea Schalk

V1.2, 16th January 2025

Contents

1 Introduction

This handbook contains information relating to the unit COMP11012: Computational Thinking. It provides an overview of the structure of the unit, the approach to teaching and introduces students to various concepts that are useful for this subject area and which appear in our workshops, videos and assessed coursework.

If you are just starting the unit, we suggest you read at least up to the end of Section 7. This will explain the mechanics of the unit. Section 10 onwards introduces some useful notions and concepts that we will be using.

2 What This Unit is About

Two of the key skills a student of computer science has to acquire are often conflated:

- devising, describing, adjusting and analysing *algorithms* and
- designing, implementing, testing and understanding *computer programs*.

It is important to note first of all that how a computer takes in and analyses data is completely different from how a human being goes about that task. The human brain is highly evolved when it comes to processing *visual information* and many people find illustrations a key aid in understanding and digesting information, whereas computers take in data one symbol at a time.

The latter is part of the reason why we have to learn languages that computers can understand, and why these languages are so unforgiving of any mistakes we make when giving the instructions we would like the computer to carry out.

While it is true that, in order to write computer programs, we must be able to develop, describe, understand and adjust algorithms, it is a separate skill. When we want to communicate with other human beings rather than a computer, the most appropriate form is therefore not a computer program, but something that is easier to understand for a human being.

This allows us to discuss algorithms and their properties in a way that is independent from any implementation. While presentations of algorithms that we consider in this unit are not written in the rigid framework of a programming languages, we can nonetheless make our descriptions, and our reasoning about those, rigorous.

2.1 Intended Learning Ourcomes

On successful completion of this unit, students should be able to:

- decompose a problem into a series of ordered steps,
- apply techniques such as decomposition, abstraction, and generalisation to problems,
- describe computational complexity in an informal way, and
- explain how self-referential problems relate to the limits of computation.

3 COMP11012 from 40,000 Feet (or 12,192 Meter)

The following narrative is based loosely on the final chapter of [1].

Computational Thinking is a set of problem solving skills, based largely around the creation of algorithms.

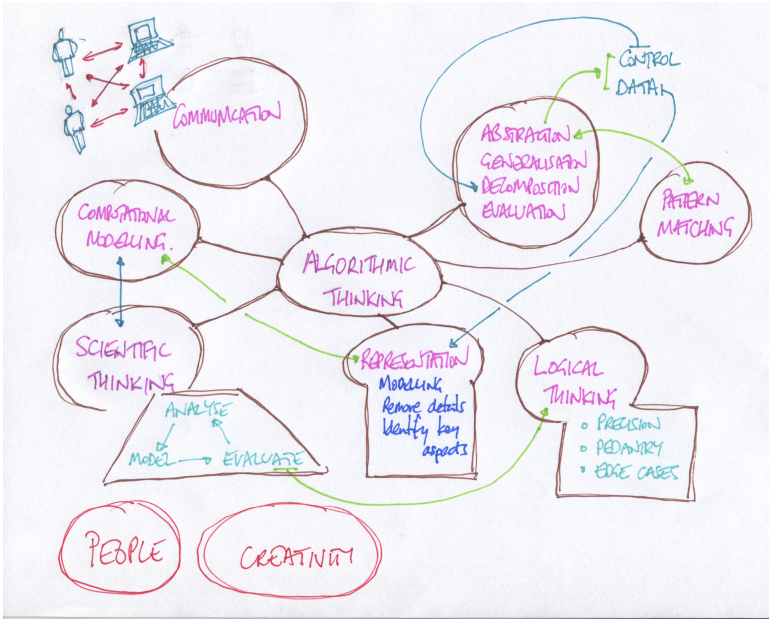


Figure 1: Mindmap of Concepts in COMP11012

During the course of the unit, we will go through a number of exercises and tasks that will be about trying to nurture these skills.

Algorithmic thinking is the ability to see the solution to a problem as an algorithm.

Writing an algorithm down allows us to

- check what it is doing
- perform it many times
- give it to another person
- give it to a machine

This is all about **Communication**. How do we communicate our ideas to other people, to machines, or indeed communicate between machines?

An important thing here is **Computational Modelling**. We take something in the real world: weather, maps, tourists and try and create representations and algorithms that will simulate that thing, or at least the parts that are relevant to our activity. Then we can mimic its behavious, run experiments, reason about it.

Scientific Thinking is about approaching problems using a particular approach. We analyse the problems, and use the results of this to generate a model. This model is then evaluated or tested, with the results of that evaluation informing our analysis and model. We should not lose sight of the fact that we are operating on a model though, which will be an approximation of the real world phenomena

we are interested in. **Evaluation** also requires us to engage in **Logical Thinking**: employing precision and pedantry in order to be very careful and precise about our details, models and observations.

Pattern Matching is something we do all the time. It saves us time, allowing us to apply known solutions to a new problem. It will often involve a level of abstraction: we need to get past the details in order to determine whether a known solution will work. So this is also closely linked to questions of representation.

Choosing appropriate **Representations** can make problems easier to solve: we will see examples of this in the unit. A good representation will also make it much easier to code up a solution, once we get to the point of actually doing some programming.

Abstraction is something that we will discuss a lot in this unit, and it is a very important concept in Computer Science. One example of abstraction, which is very closely linked to the notion of **Decomposition** is to group lots of smaller instructions together into a bigger instruction that *hides* the detail of what is going on. This is sometimes referred to as **Control Abstraction**. **Data Abstraction** on the other hand tries to hide the details of how information or data is actually represented: we might not actually care how, for example numbers are represented. We just need to know that we can do it. **Abstraction** is also useful during **Evaluation**. When we are trying to understand how fast an algorithm performs, we can look at counting, for example, how many *set operations* or *comparisons* it involves for a given input or an input of a given size. Ignoring irrelevant details makes this an easier task – without compromising the results.

Generalisation takes a problem that we have solved and adapts it so that we can solve other problems. This may involve **Pattern Matching** and the use of a suitable **Representation**.

Decomposition involves breaking something down into smaller problems. It is related to the notion of **Control Abstraction** that we described above. **Pattern Matching** and **Abstraction** also play a part here. The idea of **Recursion** is a particular example of Decomposition, where the smaller problem is similar to the larger problem.

Evaluation is about checking that a solution is fit for purpose. Does it do the right thing: is it functionally correct? Does it do the right thing in a usable way: will it actually run in a sensible amount of time even if given a large input? Testing our algorithms requires **Logical Thinking**: pedantry and precision. We can also evaluate through rigorous arguments. Can we reason about the performance or behaviour of our algorithm or solution? We will see that this is not always possible in the general case. Such reasoning may take place on an **Abstraction** of the problem. **Decomposition** might allow us to evaluate sections of the solution independently.

Running through all this are two strands that you might not immediately think of as relating to Computational Thinking: **Understanding People** and **Creativity**. Our solutions and algorithms need to be usable, so we need to understand how those solutions might be used and the problems that users will face. We need to

be creative in coming up with solutions, and looking for the problems that need to be solved. But this requires the right environment and conditions: head space to work in, supportive colleagues and peers, and the willingness and opportunity to get things wrong.

4 Course Structure

The unit runs over one semester, with eleven weeks of scheduled activities.

We have split the semester into five two week blocks, each of which covers a particular topic (with short initial, and final periods to start and end the semester). At the start of each block we have an introductory workshop which provides you with learning activities for a first encounter with the topic for the block. You then have just under two weeks to submit your related coursework, which is carried out in groups. All the tasks could be carried out by two or three engaged students taking the above ideas to heart, so you do not have to worry too much about illnesses or absences among your group members.

The blocks look at the following topics:

- **Block 1** is concerned with *how to describe an algorithm*: Using different formalisms such as flowcharts and pseudo code, and thinking about suitable primitive actions.
- In **Block 2** we look at further *tools that help us create and describe algorithms*, such as decomposing problems and algorithms, different kinds of loops, stepwise refinement, the idea of generalizing an algorithm that allows it to be applied to a related problem, and we also ask you to think about how to test and scrutinize your algorithm and its description.
- *Abstraction* is our topic for **Block 3**. This is a technique that allows us to strip irrelevant detail from the problem we are trying to solve, only retaining what is required to describe a solution. Doing this also allows us to apply the same algorithm to many different problems by realizing that they have a common abstraction. We make use of graphs (see Section 10.4).
- In **Block 4** we look at the *complexity* of both problems and algorithms, and we look at how we can count the number of steps our algorithm is going to take, for both particular examples as well as in general.
- Our final topic of *recursion* is covered in **Block 5**. This is a very powerful method for creating algorithms and data structures, but it is one that most people find difficult to understand when they first encounter it. Scroll down for a simple recursive algorithm if you would like a sneak preview of that idea.

5 How we teach

This unit is delivered using a *Blended Learning* approach. A brief definition of Blended Learning[1] is as follows:

> *Blended learning is an approach to education that combines online educational materials and opportunities for interaction online with physical place-based classroom methods.*
>
> *Blended learning requires the physical presence of both teacher and student, with some elements of student control over time, place, path, or pace.*

For us, this means that the unit is delivered using a mix of so-called *Asynchronous* and *Synchronous* activities. For asynchronous activites, academic staff create or curate materials that students access at their own pace. For synchronous activities, students and staff are present at the same time, and activities happen "live".

For COMP11012, we have videos, quizzes and directed reading that are intended for asynchronous consumption. You can choose how and when to watch the videos or take the quizzes, although we will give some guidance. Coursework tasks have a deadline, but it is up to you when the work towards the submission is done. Synchronous activities include workshops, where key concepts are introduced through group activities; and lecture sessions, which will typically be Q&A style, driven by questions from students.

There is much evidence in the literature that students learn most when they work in a mode called *active learning*, which means they go through activities that are designed to have every learner carry out tasks that help them to explore new topics. There are also many studies that demonstrate that "from the front teaching", where students listen to somebody explain material, does not have much of a lasting impact on the learner's understanding.

5.1 Misconceptions about active learning

In order to make the most of your time at university we encourage you to have a look at the literature yourself rather than take our word for it. However, there are two issues that interfere with active learning working in practice:

- Students often feel uncomfortable when they are asked to engage with active learning tasks. It pushes them out of their comfort zone, and being asked to solve tasks when students don't already know the material creates stress because students don't want to make mistakes. Active learning, on the other hand, assumes that learners will make some mistakes because it's normal to have misconceptions when first encountering new ideas, and what is considered important is that learners get *the feedback required* to

[1] from https://en.wikipedia.org/wiki/Blended_learning

allow them to self-correct such misconceptions, leaving them with a much richer understanding of the underlying notions.

- Students often come out of active learning sessions feeling unsure about what, if anything, they have learned, and maybe they even feel more confused than before the session, while their subconscious is still busy digesting everything that happened. In contrast, consider someone giving a coherent presentation on a topic that students passively follow. Students often come out of such a presentation with the (misguided) impression that they have learned a lot because they followed the presentation and it made sense to them.

A consequence of these two observations is that students may drop out of attending active learning sessions, while feeling some resentment that they are not "being taught". Actually, coming up with good learning activities is harder, and takes more time, than designing a traditional lecture – and students learn more engaging in these activities than in a traditional lecture (even if they do not notice this straight away).

If you only want to look at one study about learning that supports the above claims, we suggest that you make it the following one:

Deslauriers, Louis, et al. Measuring actual learning versus feeling of learning in response to being actively engaged in the classroom. In *Proceedings of the National Academy of Sciences* 116.39, 2019, pages 19251-19257.
https://www.pnas.org/doi/epdf/10.1073/pnas.1821936116

On this unit we embrace active learning, with workshops with carefully chosen tasks acting to introduce new ideas, and group coursework serving as a follow-up activity where we ask students to go deeper into a topic.

5.2 Teaching activities on this unit

In the workshop, feedback is provided both, by other students having contrasting ideas, and by unit staff answering questions or giving hints, and also by us bringing the whole group together to compare notes on various solutions to the given tasks.

If you are looking for some evidence that our workshops are instructive for learners, then here is an observation: One of the free answer questions in the 2021-22 exam was closely connected with one of our workshops. The students who attended the workshops on average got more than twice the number of points than students who weren't at that workshop, despite the fact that the relevant materials were available to all students for revision purposes..

For the coursework, while unit staff provide feedback when marking, it is really important that students *provide feedback to each other* while the group works on solving the given tasks. We are asking students to learn how to scrutinize solutions, or partial solutions, suggested by others, so that the group can then improve on

the ideas until they have an answer that all group members believe works well and is presented well.

5.3 Developing your independent learning skills

In addition to expectations about engaging as active learners, we expect students to be developing themselves as *independent* learners. An independent learner has control and ownership of the learning process, managing workload, commitments, deadlines. This can be a difficult shift to make if you have been used to a lot of hands-on guidance. The Foundation Year is an ideal opportunity to acquire independent learning skills: if mastered they can be the key to a successful academic career, and beyond that serve you through your professional life.

6 Teamwork

In COMP11012, you will work in two kinds of teams: you will work in

- *sets* during the (timetabled) workshops; sets are teams that are assembled afresh for each workshop.
- *groups* on your coursework; groups are fixed for the whole course unit, so you should be able to develop a good working relationship with the other members of your group during the course unit. Group members will need to meet outside the timetabled activities to collaborate on their coursework tasks.

Working in teams will enable students to design complex solutions to complex problems and present these in an understandable way. In particular, they

- share their ideas for solutions to problems
- work on algorithms together
- build on top of each others' ideas
- improve other students' solutions
- scrutinize, test, and run other students' solutions, thereby identifying points for improvement
- gain experience in teamwork, which is of central importance for computer scientists and software engineers

The coursework in COMP11012 is designed to be solvable well by a team of 3 students, and the coursework teams (groups) contain about double that many students.

We expect all students to contribute to the best of their ability to their teams, both during the workshop and in their coursework teams. Being an efficient member (or leader) of a team requires a wide range of skills; we do not expect students to master these skills yet but that they are keen to further develop and practice them. In particular, a good team player is able to do the following:

- Keep an eye on "the process": goals, deliverables, tasks, etc. via suitable means like agenda, minutes, to-do lists, etc.
- Communicate clearly with everybody in the team. We want students to work on their ability to do the following:
  - Explain their own ideas or thoughts. This is often difficult when the other members of your team don't initially understand what you mean, and you may find using an example can help you with getting your ideas across.
  - *Listen actively* to other team members: this includes paying attention and showing that you are paying attention to the speaker, concentrating on what they are saying and trying to understand what they mean. It is important that you give others space to express their ideas, even if you don't think they are very good ideas. Expressing your judgement will stifle creativity in the team. After you have heard them out, respond with constructive feedback: Can you see ways of improving on their original suggestion? This is often difficult, in particular when you have your own thoughts/ideas and are eager to explain these rather than listening.
  - Practice *equality in the distribution of conversational turn taking*: in your team meetings, get into the habit of having all team members talk for roughly the same amount of time, and make sure all team members feel happy to contribute their ideas. Google has carried out some interesting research and found that this is extremely important for your team's success [2].
- Split a task into suitable sub-tasks.
- Combine different pieces of work/solutions into a whole
- Plan their time
- Contribute to the team behaving in a respectful, fair, inclusive, and professional manner
- Address issues your team encounters in a transparent, open way with your team; if you experience serious concerns, consult with the COMP11012 team and see the next section.

Also, we suggest you bring pen and paper to each learning session so that you can sketch solutions and examples, work through algorithms together, take notes, etc.

6.1 Marks and Teamwork

Many of the activities in this course unit involve teamwork, and we expect all students to make a suitable, consistent effort and contribute to their team tasks. For group coursework, the same mark will be given to all students of a group—unless in extreme situations as described in this section.

Should a student not contribute to their team's effort, in particular in the coursework groups, we first expect the other group members to address this in an open, professional way; in particular, they can calmly point out that/where they found the student's contributions to be lacking.

In case a student continues to fail to contribute to their group's effort, please consult with the COMP11012 team. We will try to improve the situation.

In the rare and unlikely case where a student has consistently failed to contribute to their group's effort (despite the group's efforts to bring them back into the team and suitable discussions of the issues arising with the COMP11012 team), the student's coursework mark can be reduced to reflect this lack of contribution.

7 Thinking Computationally

This course unit is concerned with algorithms, and various skills related to those, and not with programming. You learn something about the latter in the project course unit, and there are two introductory programming course units in the first year of the computer science degree.

What we want to do in this unit is to think about algorithms *without worrying about how to express them in a particular programming language.* In the second year, when students on the computer science degree study algorithms in more detail, this is never done looking at programs (although there are practical lab exercises where students are asked to *implement* and analyse particular algorithms).

When we are concerned with algorithms we typically use *pseudo code*, or maybe *flow charts*, to describe the procedures under consideration. For more complicated sets of instructions, pseudo code provides the more flexible format. There is no fixed syntax for pseudo code, and we do not insist that you use a particular set of commands on this unit either. Section 8 discusses this in more detail.

7.1 Tips for describing algorightms

When you describe algorithms you should bear the following in mind:

- Your instructions should be unambiguous, and somebody who follows them should always carry out the same steps for the same input (although the order of some of the steps may vary).

- Your instructions should give sufficient detail so that you do not leave it up to an agent following those instructions to decide how to interpret more complex actions. We sometimes talk about choosing appropriate *primitive actions* for the task in hand.
- Your instructions should not require any advanced knowledge for somebody to carry them out. Could you give them to a younger sibling and they would know what to do? (You may have to explain to them what a graph, or an array, or a set is first.)
- If your instructions include some randomized task such as"select something", or "choose an item", have you made sure that the procedure will terminate if carried out, or could it be that the same random action is taken over and over again?
- Have you made sure that you cover every situation encountered by an agent following your instructions? If you say "choose an item", but there are no such items to choose from, what should the agent do?

You may find the following useful when working on your group's coursework submission:

- When scrutinizing your group's work you may find it useful to imagine that you are trying to find a way of following the given instructions which will not give the desired result. That will allow the group to eliminate any ambiguities.
- You should ask whether you have chosen appropriate primitive actions. Is your current version hiding some of the details in instructions that need to be expanded? For example, take "take the closest intersection and ...": is it clear how your agent is supposed to make that decision? Does it already have the information of the required distances? How does it determine which one is the closest? What should it do if there is more than one intersection satisfying the criterion?
- It is usually helpful to try an algorithm on small examples of suitable input data, and then to also think about whether making changes to that input data might reveal a situation where your algorithm does not have any instructions, or whether in some weird situation, the instructions do not have the desired effect. There is some further discussion about testing in Section 9.

Describing an algorithm precisely and with sufficient detail is an *iterative process*, and the quality of your submissions to the coursework is likely to depend on how thoroughly you scrutinize each iteration to find possible improvements.

8 Pseudocode

Throughout the unit we will be asking you to describe algorithms. This will typically be done using **pseudocode**.

Wikipedia [4] describes pseudocode as:

"a description of the steps in an algorithm using a mix of conventions of programming languages (like assignment operator, conditional operator, loop) with informal, usually self-explanatory, notation of actions and conditions.".

Goodrich [3] summarises pseudocode as follows:

"Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. There really is no precise definition of the pseudocode language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudocode mixes natural language with standard programming language constructs.

When we write pseudocode, we must keep in mind that we are writing for a human reader, not a computer. Thus, we should strive to communicate high-level ideas, not low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice."

Thus there are various "flavours" of pseudocode using different conventions that often correspond closely to syntax used in programming languages. But note that unlike languages like python or Java, there is no definitive standard for pseudocode – it is intended to be read by humans rather than machines, and so does not necessarily need to conform to a rigorous specification. We also might not expect the reader to have a deep understanding of particular programming languages. Within this unit we do not mandate a particular notational convention that we expect you to follow. We do, however expect that any pseudocode you use follows some basic principles.

Clarity If there are inputs or outputs in a description, be clear about what they are. Variables do not necessarily require *declaration*;

Scoping When you are describing, for example a loop, it should be clear which statements are included in the loop, i.e. which statements get repeated. This could be done through explicit `BEGIN/END` structures, indentation or some other mechanism;

Consistency Don't mix styles within a description;

Brevity Include as much information as you need, but no more. Descriptions should not be verbose.

```
FOR x = 1 TO N {
  a = a + x
  FOR y = 1 to M DO
    b = b + a
  ENDFOR
  }
```

(a) Undesirable

```
FOR x = 1 TO N DO
  a = a + x
  FOR y = 1 to M DO
    b = b + a
  ENDFOR
ENDFOR
```

(b) Better

Figure 2: Pseudocode Consistency

8.1 Pseudocode Examples

In Figure 2a we see the use of two different ways to indicate the scope of the `FOR` loop: braces and explicit `DO` and `ENDFOR` keywords. While these would be acceptable individually, mixing them is undesirable. In Figure 2b we see a more consistent approach.

Figure 3a illustrates an issue with scoping. It is unclear which statements sit within the `WHILE`, `FOR` or `IF` statement. Figure 3b is better, with indentation denoting the scope, but could be further improved by explicit use of braces or `END` keywords.

The code in Figure 4a is undesirable. It is using some constructs from a particular programming language (C's `for` loop and the ternary conditional operator). Without some knowledge of this a user might have difficulty understanding what this code is doing. Figure 4b improves this by giving an explicit description of what is intended by the description.

9 Testing

A key question when designing an algorithm is to ask whether the algorithm does what we intend it to do for valid inputs. *Testing* can help us to answer this question. If we have a pseudocode description, we can't necessarily execute the algorithm (using a machine), but we can run through the steps by hand, checking what happens at each stage. This can then give us some confidence that our algorithm will behave as we want.

When testing, we want to look at a range of different inputs, ideally covering different cases. In particular, we should look at what are often called *edge cases*. Edge cases typically occur at the extremes of data. For example, if we have a `FOR` loop, we might want to look at what happens when we are considering the values at the start and end of the loop. Consider also extremes of input data. If our input is, say a number between 0 and N, what happens when the input is 0, or N?

Testing will not necessarily be *exhaustive* (it may be impossible to check all

```
WHILE carryOn DO
FOR x in nodes DO
current = x
IF x has an outgoing edge THEN
count = count + 1
carryOn = false
total = total + count
```

(a) Undesirable

```
WHILE carryOn DO
  FOR x in nodes DO
    current = x
    IF x has an outgoing edge THEN
      count = count + 1
    carryOn = false
total = total + count
```

(b) Better

Figure 3: Pseudocode Scoping

```
for(x=1;x<y;x++) {
output(values[x] < 10 ? "good" : "bad";
}
```

(a) Undesirable

```
FOR x = 1 TO y
  IF the xth entry in values is less than 10 THEN
    output "good"
  ELSE
    output "bad"
  ENDIF
}
```

(b) Better

Figure 4: Pseudocode Language Agnosticity

possible inputs), but it should be *comprehensive*, with a selection drawn across the range of possible inputs.

When testing, try and put yourself in the position of someone who is reading the description as written, rather than using your own interpretation or idea of what the algorithm should be doing. This may help in identifying assumptions that you've made or places where there is ambiguity. When working in a team you could split the work – one sub team describes the algorithm, the other tests it. Such an approach can again help to identify assumptions or ambiguity.

If our algorithm involves iteration using `WHILE` or `REPEAT` loops, we should be looking to determine whether or not the algorithm will terminate. Is it possible that the condition in a `WHILE` loop will always be true? Again, we may not be able to determine this with absolute certainty, but some judicious testing may allow us to identify if there is an issue with our description.

10 Notions that help us describe algorithms

When we describe an algorithm we usually need a way that captures the structure of the data we are concerned with for the problem at hand. We will not be looking at advanced data structures, such as ordered binary treas, heaps, queues, stacks, and the like, and all our tasks can be carried out using some fairly basic structures which we introduce in this section.

In order to decide what a suitable structure for a given problem might be, we need to think about

- how can we access each item of the given data and
- how can we move between different items of data.

Answering those questions will usually point us towards a suitable structure on which to base our algorithm. In order to describe an algorithm we have to refer to the data of some specific situation we may find ourselves in, and by choosing a particular representation for that purpose we take an important step towards making sure that our description is clear and unambiguous because the given structure tells the agent how to interact with the given data.

It is no accident that many of the concepts we give below also play an important role in mathematics or theoretical computer science—having these precisely defined notions, which gives us clarity for how we interact with our data, is just what makes them useful for other tasks requiring precision, and the structures we describe here capture some quite simple ideas regarding structure of data. This makes them useful *abstractions* which apply in a variety of situations.

10.1 Basic values

Before we look at some structures that allow us to put together several items of data we begin by briefly discussing instances of such data that will be used on this unit.

10.1.1 Booleans

Sometimes we are interested in a question that has a simple true/false answer, or when describing a more complicated algorithm we need to check for a particular property. For these purposes we may use **boolean** values, which may be either `true` or `false`.

For example, we might define a recursive algorithm `even` that checks whether a number is even or odd, and which only needs to know about subtraction (rather than division) .

```
name of algorithm: even
input: natural number n
output: boolean

if (n = 0)
   then return true
   else {if (n = 1)
     then return false
     else return even(n − 2)
     }
```

Do not worry if you are unsure how this algorithm works—we discuss recursion in more detail in Block 5. But you can still try to get a first impression how it goes about its task:

Activity 1. Put yourself into the role of an agent following that algorithm and calculate `even(5)`. If you find this confusing, this may be a good activity to come back to later.

If you have any questions about this, please ask us on the Blackboard discussion board.

10.1.2 Numbers

You have been familiar with various sets of numbers (and their operations, such as addition and multiplication) for a very long time, but note that there may be some unexpected differences when you encounter these again in your university studies.

The **set of natural numbers**, $\mathbb{N}$, consists what are also called the *counting numbers*,[2] 0, 1, 2, 3, 4, and so on.

A key observation is that there is a smallest number, 0, and that we may construct every other natural number by adding 1 to that number again and again and again.

[2] They are called this as they capture the number of elements in finite sets, i.e., 0 is the number of elements in the empty set, 1 in the next larger set, 2 in the next larger set, and so on.

This is the reason why we are able to write recursive programs whose inputs are natural numbers, by telling the computer what do do if the input is the number 0, and what to do if the number is the result of adding 1 to a previously constructed number. In the example from above, the algorithm specifies what the answer should be if the input is 0, and also if the input is 1, and then determines the answer for an input of the form $n + 1 + 1$ via its output for the input n.

We have addition and multiplication as key operations for the natural numbers.

Note that you may have been told previously that 0 is not a natural number—it is possible to define the natural numbers as starting form one. However, in computer science, and also in most university-level maths units, you will find that it is standard to include 0.

The **set of integers**, $\mathbb{Z}$, extends the natural numbers by also allowing their negative, so you might think of it as $\ldots, -3, -2, -1, 0, 1, 2, 3, 4, \ldots$. The operations of addition and multiplication are defined for the integers in such a way that they agree with the corresponding operations for the natural numbers in the cases where both inputs are elements of $\mathbb{N}$. On the integers we are able to use subtraction, which is an operation closely related to addition.

The **set of rational numbers**, $\mathbb{Q}$, extends the integers by allowing fractions of those, where we have to bear in mind that division by 0 is undefined. Again we have addition and multiplication extending the corresponding operations for the integers, and we gain the operation of division as long as we do not divide by 0. Division is closely related to multiplication.

The **set of real numbers**, $\mathbb{R}$, allows us to express all numbers that we may write in decimal notation, and again we have extensions of all the operations previously defined.

In this course unit, you will mostly work with the natural numbers and the integers but when we graph a function we are implicitly using the real numbers.

10.2 Sets

Modern mathematics is based on sets, but for the purposes of using sets as a concept in computer science we thankfully do not have to worry about the deeper foundations of the subject, known as *set theory*.

10.2.1 Sets and elements

We may think of a set as a way of putting together a bunch of items into an entity, and we can then refer to these items communally via that entitty.

A **set** has **elements**, and a set is determined by knowing all its elements. We may write, for example

$$\{a, b, c\}$$

for the set containing the first three letters of the alphabet. We may then ask whether some entity is an element of a given set and, if we set

$$A = \{a, b, c\},$$

we would then say

$$c \in A \qquad \text{but} \qquad g \notin A,$$

where "$\in$" is read as "is an element of" or "is in" for short.

An easy way to specify a set is to enumerate its elements, as we have done above for A. Alternatively, we can describe it. For example, we can say "let B be the set of natural numbers that are even" or "let C be the set of letters in the latin alphabet that are consonants". Of course we need to make sure that our descriptions are very clear so that we all agree on which elements are/are not in the set described.

10.2.2 Operations for sets

If we already have some sets we may construct new sets by applying various *set operations*.

The **union** of two sets is the collection of elements that appear in at least one of the two sets, that is

$$S \cup T = \{s \mid s \in S \text{ or } s \in T\}.$$

Here we see a more formalized version of the above description of a set: in the description of $S \cup T$ above, we use the vertical bar $\mid$ to mean "such that" and the whole line reads as "S union T is the set of all those s (which appears before the vertical line) such that s is in S or s is in T".

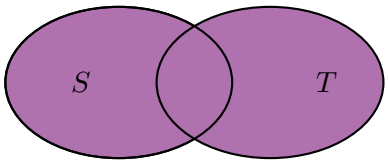
The **intersection** of two sets is the collection of elements that appear in both sets, that is

$$S \cap T = \{s \mid s \in S \text{ and } s \in T\}.$$

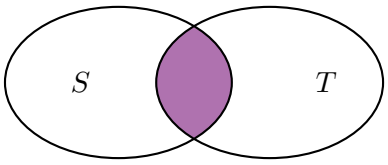
The **set difference** of two sets is the set of all elements that occur in the first, but not in the second set, that is

$$S \setminus T = \{s \mid s \in S \text{ and } s \notin T\}.$$

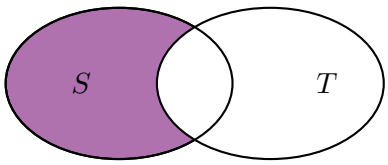
We may picture these operations as follows:



$S \cup T$



$S \cap T$



$S \setminus T$

10.2.3 Comparing sets and equality of sets

We are also able to *compare sets* by asking whether one is entirely contained in the other, that is

$$S \subseteq T$$

if and only if

$$\text{for all } s \in S \text{ it is the case that } \ s \in T,$$

in which case we say that S **is a subset of** T, or that T **is a superset of** S.

Because the only relation we have between a set, and potential elements, is whether or not they occur in the set, a set does not have a notion of an element appearing several times. In other words

$$\{a\} \cup \{a, b\} = \{a, b\},$$

and it does not really make sense to write $\{a, a, b\}$.

Note also that a set does not come with a way of *ordering its elements*, and while we tend to write $\{a, b, c\}$ rather than $\{b, a, c\}$, these two expressions describe the same set. It does not make sense to talk about "the first element of a set", so if we need to scrutinize the elements of a set one by one, we have to do so in an *arbitrary order*, unless we have some additional information available to us.

10.2.4 Sample algorithms using sets

We typically use a set to describe the data we have if there is no structure to the data, and if we do not have duplicates among the data.

If we want to identify all the books in a library by their title, then a set may not be a good structure for these data unless we are sure that no title appears more than once.

But if we have a pile of some items, and we have to check the items one by one to put them onto two different piles depending on some property, say X,[3] we may use sets for the piles, and describe a simple algorithm as follows, assuming we have a set S of items, and we are creating two new sets, P with items that have our property, and N with items that do not have the property.

```
name of algorithm: extract
input: a set S
output: sets P and N

set P to ∅
set N to ∅
while S ≠ ∅ do {
```

[3] This property X will depend on what kind of items are in S and what we want to extract. Examples could be "is even" for sets S of integers or "is expensive" for sets of products S.

```
  choose i ∈ S
  set S to S \ {i}
  if (i has the property X)
    then set P to P ∪ {i}
    else set N to N ∪ {i}
}
return P and N
```

The output of this algorithm is given by the sets P and N (or if we were only interested in items that have the given property, the output would be just the set P, and we might then drop N as an output, and remove the instruction that updates N).

Activity 2. Again, imagine you are the agent carrying out that algorithm. Using pencil and paper to keep track of how the sets S, P and N change,, use a step by step process to calculate the output of this algorithm for the set $S = \{0, 1, 2, 3, 4, 5\}$ where the property X we are looking for is for the number to be even, that is, divisible by 2 leaving no remainder.

Is there just one way of following the instructions, or can you think of more?

If you have any questions about this, please ask us on the Blackboard discussion board.

An agent following the given instructions will choose an arbitrary element from the input pile, check whether it has the given property, and then put it onto the appropriate output pile. They will stop when there is nothing left on the input pile.

If checking the given property is a complex task, we could now *refine* our instructions.

Maybe we have a separate algorithm that checks for the procedure, say `check` which might look like this:

```
name of algorithm: check
input: item i
output: boolean

if (instructions that
    check for the property X)
  then return true
  else return false
```

Our original algorithm could then be written as

```
name of algorithm: extract
input: a set S
output: sets P and N

set P to ∅
set N to ∅
while S ≠ ∅ do {
  choose i ∈ S
  set S to S \ {i}
  if (running check with input i returns true)
    then set P to P ∪ {i}
    else set N to N ∪ {i}
}
```

By giving names to our algorithms we talk about them more easily, and we are also able to use them as part of bigger algorithms, which is useful. Also note that it can help to organize our information by describing the input and output for each algorithm

When describing an algorithm it is often a good idea to first concentrate on the general shape of what we need to do, and then give more details for specific parts of the overall procedure as we did with the algorithm that provides the checking functionality in the previous example.

Here is another sample algorithm for sets. This one takes as input two sets and produces the intersection of the two.

```
name of algorithm: intersect
input: two sets S and T
output: a set O

set O to ∅
while (S ≠ ∅) do {
  select s ∈ S
  set S to S \ {s}
  if s ∈ T then set O to O ∪ {s}
}
return O
```

Note that this algorithm (like the algorithm `extract` above) destroys the input set S as it carries out its calculations. If you want to use this as part of a bigger algorithm you may want to start by copying the set S and then you could work

with the copy.

Activity 3. Describe an algorithm that takes as input two sets and outputs the union of the two.

You may want to start with the following:

```
name of algorithm: union
input: two sets S and T
output: a set
```

Do you have to start with the empty output set?

If you have any questions about this, please ask us on the Blackboard discussion board.

Note that when using a set to provide the data of an algorithm, you should not use primitive instructions that assume your agent can see all the elements of the set at once, and, for example, find the smallest one in a set of natural numbers. The set might be very large, and your algorithm needs to work for all sets, so you need to ensure that you give your agent simple steps that can be carried out in any possible situation that might arise.

10.2.5 The product of two sets

An important operation for sets is that of the **product of two sets**.

When graphing functions or using vectors, you have probably encountered $\mathbb{R} \times \mathbb{R}$ (also written as $\mathbb{R}^2$), the product of the set of real numbers with itself, where elements are given as a pair (x, y), where

- x is the x-coordinate given by a real number and
- y is the y-coordinate, also given by a real number.

We may apply the same idea to all sets, by formally taking all the pairs where the first coordinate comes from the first, and the second coordinate from the second set.

Definition 1: product of two sets

The **product of two sets S and T** is the set

$$\{(s, t) \mid s \in S, t \in T\}.$$

For example, for the set $\{a, b, c\}$ we get that

$$\{a, b, c\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c)\}.$$

An example you will have come across is that of *playing cards*, where the four suits clubs ♣, spades ♠, hearts ♥ and diamonds ♦ are combined with values, for

example 7, 8, 9, 10, J, Q, K, A, and we may think of a set of cards as being given by the product of the sets

$$\{\clubsuit,\spadesuit,\heartsuit,\diamondsuit\} \times \{7, 8, 9, 10, J, Q, K, A\}.$$

We need the product of two sets to define the notion of a graph, see Section 10.4.

Activity 4. Describe an algorithm that takes as input two sets and produces as output the product of the two sets.

If you have any questions about this, please ask us on the Blackboard discussion board.

10.3 Arrays

Most imperative programming languages support the notion of an array.

10.3.1 Indexed data storage

An **array** is an entity that allows us to interact with data in a slightly more structured way. Here is an example of an array:

0	1	2	3	4	5	6	7
9	35	6	5	58	5	1	42

An array is defined for indices, here they go from 0 to 7. The data is stored in the array with one data item at each index. In the example, the entry at index 4 is 58, and if the array is called a we would write this as $a[4] = 58$.

In many programming languages it is standard that indexing starts at 0, but when writing pseudo code you are free to start, for example, with index 1.

An array allows us to

- access an entry at any index and
- provides us with a systematic way of going through all the entries.

Here is an algorithm `min` that finds the smallest entry in an array of natural numbers.

```
name of algorithm: min
input: an array a with natural number entries and
       maximal index maxindex
output: a natural number

set m to a[0]
for i = 1 to maxindex {
```

```
  if a[i] < m
    then set m to a[i]
}
return m
```

Activity 5. For the example array from the start of this section, use a step by step process to calculate the output of `min`. Use paper and pencil to keep track of how m changes as the algorithm progresses.

If you have any questions about this, please ask us on the Blackboard discussion board.

Activity 6. Describe an algorithm that takes as input an array of integers and whose output is an integer, namely the sum of all the entries in the array.

If you have any questions about this, please ask us on the Blackboard discussion board.

Activity 7. Describe an algorithm that takes as input two arrays and defines a new array, which has as entries all the entries of the first input array, followed by all the entries of the second array.

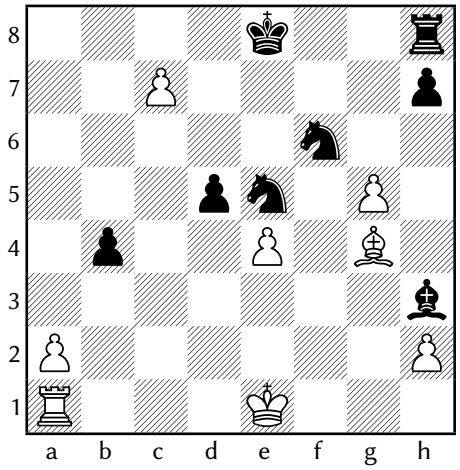
If you have any questions about this, please ask us on the Blackboard discussion board.

Note that there is no way of "looking at all the entries in the array"—we have to access them one by one. Also, while it is possible to *sort* the entries in an array so that, for example, they appear in ascending order as the indices increase, unless you are told that your data has already been sorted, you should not assume that you have what is called an *ordered array* as the input.

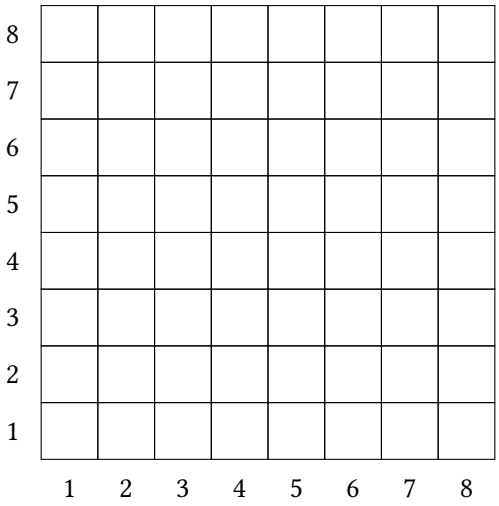
A disadvantage of having an array is that its size is fixed, so if you need to be able to add or remove data, you will want to use something more flexible.

10.3.2 Higher dimensional arrays

In some situations it is preferable to have a different indexing system. For example, we might want to use an array with boolean entries to express whether a square on a chessboard is occupied by a piece.



We could number the squares from 1 to 64 (or from 0 to 63), but we probably find it easier to work out which square is which if instead we use a **2-dimensional array** a. Instead of illustrating such an array as a line, we would draw this one as a table.



In order to encode the fact that the square $g4$ is occupied in our given chessboard we would set

$$a[7, 4] = \texttt{true}.$$

The array a is said to be *2-dimensional*. We ask you to use a 2-dimensional array in an algorithm in Activity 10

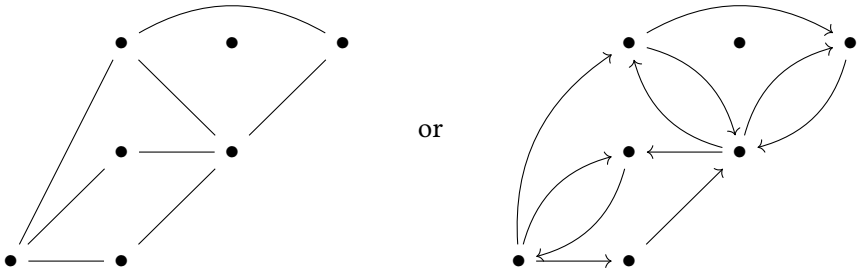
You might imagine wanting to use 3-dimensional arrays for blocks in a 3-dimensional space.

10.4 Graphs

So far we have only looked at data where there are no connections between items.

10.4.1 Data with connections

Sometimes we want to be able to describe data that we may picture as follows:



Here we have data items (there may be names associated with the • items in the picture) that have *connections* between them. Sometimes these connections only work in one direction, and sometimes they go both ways. There are many examples of this kind, and maybe the most natural one is that of some kind of *map*, for example for a transport system, but these might also be the description for moves in a game. Or, for example, the first picture might describe which people in a small group know each other, and the second might describe whether a person likes another. The possibilities are endless.

We need to make formal what we mean by these pictures in such a way that we are able to use them in an algorithm. We say that a graph has

- **nodes**, which are indicated as • in the picture and
- **edges**, which are the connections between the nodes.

Our pictures suggest that we can treat all the nodes as the same somehow since they are all marked by •, but any description of a graph will have to have distinct names for the nodes, and we find it useful to think of them as forming a set.

We may have even more data to include in our picture, and we talk about that after looking at formal definitions.

10.4.2 Directed graphs

We have to make a distinction between the situation where the edges are *directed* and that where they work both ways.

If there is an edge from a node x to a node y we say that y **is a direct neighbour of** x.

Definition 2: directed graph

A **directed graph** is given by a set N of nodes and a set

$$E \subseteq (N \times N) \setminus \{(n, n) \mid n \in N\}$$

of edges.

In the above definition, we say that the edges in E are *pairs* of nodes (via $E \subseteq (N \times N)$) and we exclude edges from a node to itself (via $\setminus\{(n, n) \mid n \in N\}$).

Based on this definition, the following defines a directed graph:

$$N = \{a, b, c, d, e\}, \qquad E = \{(a, b), (b, a), (b, c), (c, b), (d, a), (d, b)\}.$$

We may picture this graph as shown in Figure 5.

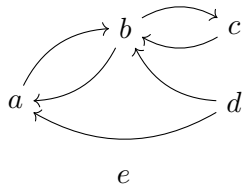


Figure 5: A directed graph

The example should make it clear that we interpret an edge of the form (x, y) as a connection *from* x *to* y.

This leaves us with explaining why we are excluding edges of the form (n, n) where $n \in N$. This would be an edge that goes from a node to itself

This is not allowed according to our definition. If we wanted this to be a feature we would have to talk about a *directed graph with loops* but we will not use those in this course unit.

Also note that by saying that the set of edges forms a subset of the set $N \times N$ we ensure that there is *at most one edge* from a node x to a node y.

There may be reasons you would like to allow there to be more than one edge, in which case we would speak of a (directed) *graph with multi-edges*. For this we need to change the formal definition, but we will not be concerned with that possibility in this unit so we do not look into the details.

10.4.3 Undirected graphs

What is the difference between a directed and an undirected graph? We think of all the connections as working both ways, but what does that mean for our formal definition?

There are two solutions one might adopt here. We might either still use a set $E \subseteq N \times N$ to give the edges, but demand that $(x, y) \in E$ implies $(y, x) \in E$, to ensure that all edges go both ways. But that would make the set E twice as big as it needs to be.

Instead we use *2-element subsets of* N to describe the edges of our graph, making use of one of the features of sets, namely that the order of elements does not matter.

Definition 3: undirected graph

An **undirected graph** is given by a set $N$of nodes and a set

$$E \subseteq \{\{x, y\} \mid x, y \in N, x \neq y\}$$

of edges.

If $\{x, y\} \in E$, we say that x and y are neighbours, i.e., x is a neighbour of y and y is a neighbour of x.

Again we forbid loops by not allowing one-element sets of the form $\{x\}$ to define an edge (which would have to be from x to itself). We again do not allow multi-edges in our definition since we demand that all the edges form a set, which rules out duplicates..

We may now consider the graph

$$N = \{a, b, c, d, e\}, \qquad E = \{\{a, b\}, \{a, d\}, \{b, c\}, \{b, d\}\}$$

which we may picture as shown in Figure 6.

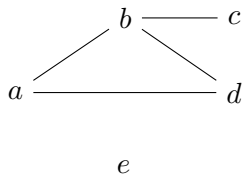


Figure 6: An undirected graph

Activity 8. For the figures given at the start of this section, formally define graphs whose picture looks like those figures.

Hint: You may want to start by naming the nodes in the pictures.

If you have any questions about this, please ask us on the Blackboard discussion board.

10.4.4 Algorithms using graphs

Graphs provide us with a very flexible structure that is used in many algorithms and you will see this in some of our workshops and coursework.

Activity 9. Consider the following algorithm.

```
name of algorithm: algoA
```

```
input: a (directed or undirected) graph with
       set of nodes N and a node n ∈ N
output: a set of nodes

set S to {n}
set O to ∅
while S ≠ ∅ do {
  choose m in S
  set S to S \ {m}
  if (m ∉ O) then {
    set O to O ∪ {m}
    for every direct neighbour m' of m
      if (m' ∉ O) then set S to S ∪ {m'}
  }
}
return O
```

Using pencil and paper to keep track of the sets S and O, apply the algorithm `algoA` to the two graphs from Figures 5 and 6 for node a as the second input.

Compare the two outputs you obtain. Can you explain why they are different?

Can you work out what this algorithm computes in general?

If you have any questions about this, please ask us on the Blackboard discussion board.

Activity 10. Describe an algorithm that takes as input a directed graph with

$$N = \{0, 1, \ldots, n\}$$

and edge set E, and which produces a 2-dimensional array whose entries are booleans with the property that an entry in the array is **`true`** if and only if there is an edge from i to j in the graph. This is known as the *incidence matrix* of the graph.

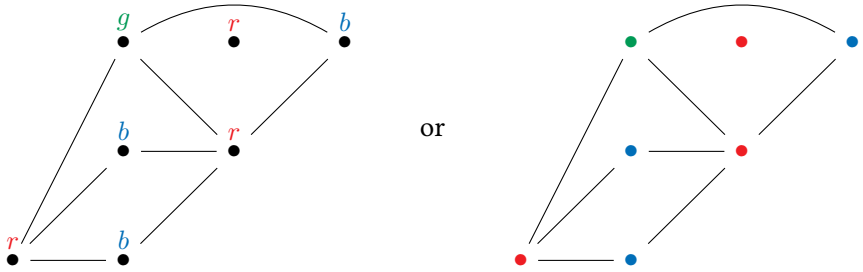
If you have any questions about this, please ask us on the Blackboard discussion board.

10.4.5 Labelling graphs

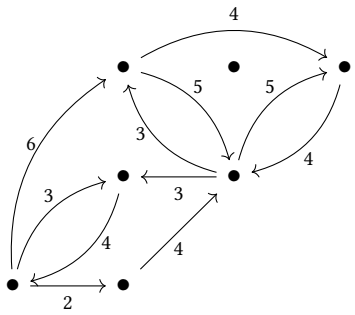
It is not unusual for us to have additional data available that we may want to include in the information of the graph. For example, we may want to associate a colour with each node in the graph. We would say that we want to *label each node*

with a colour. Note that we can not use the labels to name the nodes unless every node gets a different colour!

We could picture this labelling in different ways:



Alternatively we might want to label the edges of the graph, for example with distances or costs of following a connection from one node to another.



This additional information allows us then to create algorithms that use this data, for example by computing the shortest path between two nodes in a graph where edges are labelled with distances, or to produce labels as outputs, for example by colouring the nodes in a graph in such away that no two connected nodes have the same colour.

10.5 Strings

So far we were mostly concerned with either using boolean values, or numbers, as the data that might provide inputs or outputs to our algorithms. But, of course, we also want to be able to compute with text.

10.5.1 Sticking symbols together

When we look at text we can see that it is composed from individual symbols, stuck together. Dealing with symbols one by one is not very efficient, so in computer science we have a notion of sticking symbols together, but we first need to fix which symbols we allow.

When we have a set of symbols (often called[4] Σ), we talk of an **alphabet**.

[4] This is the capital version of the Greek letter known as "sigma".

Definition 4: string over a given alphabet

Let Σ be a set of symbols. A **string over the alphabet** Σ is given by

$$s_1 s_2 \cdots s_n$$

where $n \in \mathbb{N}$ and $s_i \in \Sigma$ for $1 \leq i \leq n$.

In other words, a string just consists of sticking together symbols of the given alphabet, so

$$aabbcabccccccca$$

is a string over the alphabet

$$\{a, b, c\}$$

(but also a string over the alphabet $\{a, b, c, \ldots, x, y, z\}$). Sometimes strings are also called **words over the given alphabet**. When we perform operations on strings we often need to know which symbols are allowed in strings, which is why we always specify the underlying alphabet.

A subtle point is that our definition says that $n \in \mathbb{N}$, and our definition of $\mathbb{N}$ says that 0 is a natural number—so what does a string look like for $n = 0$? How can we take symbols indexed from 1 to 0? The answer is that this is a string that contains no symbols at all. Writing an empty space is not particularly helpful, so in many areas of computer science the Greek letter[5] ϵ is used for the **empty string**, but one could also use quotation marks to put around an empty space " " – but then one has to use quotation marks to enclose all strings for a consistent definition if one wants to add symbols to the empty string...

Note that since the number of symbols contained in a string is finite, all our strings are finite, and an infinite sequence of symbols from the alphabet is not a valid string.

We can stick strings together, known as **concatenation of string**, so

$$\text{the concatenation of } abbcca \quad \text{with} \quad bbbca \quad \text{is } abbccabbbca.$$

An example that appears quite a bit in computer science is that of a **binary string** which is a string made from the symbols 0 and 1, that is, a string over the alphabet $\{0, 1\}$.

Since digital data is stored using two values, which are often taken to be 0 and 1, there are a lot of relevant ideas that can be described using binary strings.

Here is a simple algorithm that involves binary strings.

```
name of algorithm: length
input: a binary string w
output: an natural number n
```

[5] This is the letter with the name "epsilon".

```
set n to 0
while (w ≠ ϵ do {
  set w to the string resulting from removing
           the first symbol from w
  set n to n + 1
}
```

Activity 11. For the binary string 001101,as the input what is the output of the algorithm `length`? Can you say what happens if the input is an arbitrary binary string?

What if we wanted to have strings over a different alphabet as the inputs to this algorithm? What changes would we have to make?

If you have any questions about this, please ask us on the Blackboard discussion board.

In the algorithm above we treated a string as something that we have to take apart in order to study it. Instead it is not unusual to make the assumption that we can have access to the ith symbol in a string. That idea is required for the following algorithm.

```
name of algorithm: algo1
input: a binary string w
output: a natural number

set n to 0
if w is the empty string
  then return n
  else {
    for i = 1 to length of w {
      if (the ith symbol of w is 1) then set n to n + 1
    }
  return n
  }
```

Activity 12. What happens if you apply the algorithm `algo1` to the binary string 00101? How would you describe the output for an arbitrary input? s Why do we have to treat the empty string separately? If we look at strings

for different alphabets, what would be a sensible way of generalizing this algorithm?

If you have any questions about this, please ask us on the Blackboard discussion board.

We may also use binary strings as outputs of algorithms, not just as inputs.

```
name of algorithm: algo2
input: a binary string w
output: a binary string

set v to the empty string
if w is the empty string
  then return v
  else {
    for i = 1 to length of w {
      if (the ith symbol of w is 1)
        then add 1 to v
    }
    return v
  }
```

Activity 13. How would you describe what `algo2` outputs for a given input?

Can you find a connection between the algorithms `length`, `algo1` and `algo2`?

If you have any questions about this, please ask us on the Blackboard discussion board.

Activity 14. Describe an algorithm that takes as input a binary string and which outputs the binary string where the 0s and 1s have been swapped.

If you have any questions about this, please ask us on the Blackboard discussion board.

References

[1] Paul Curzon and Peter William Mcowan. *The Power Of Computational Thinking: Games, Magic And Puzzles To Help You Become A Computational Thinker.* World Scientific Publishing, 2020.

[2] Charles Duhigg. What Google learned from its quest to build the perfect team: New research reveals surprising truths about why some work groups

thrive and others falter. *The New York Times Magazine*, 2016. Available at https://nyti.ms/2ExlQcJ.

[3] Michael T Goodrich and Roberto Tamassia. *Algorithm Design and applcations*. John Wiley Inc, 2015.

[4] Wikipedia contributors. Pseudocode — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Pseudocode&oldid=1196593773, 2024. [Online; accessed 22-January-2024].