

# **VCE Algorithmics (2017 Edition)**

**An introduction to Computer Science**

**Bernd Meyer, Steven Bird**

## **VCE Algorithmics (2017 Edition)**

*An introduction to Computer Science*

Bernd Meyer, Steven Bird

License: Copyright © Monash University, unless otherwise stated. All Rights Reserved. (No derivatives)

Generated by [Alexandria](https://www.alexandriarepository.org) (<https://www.alexandriarepository.org>) on November 23, 2018 at 3:50 pm AEDT

---

### **COPYRIGHT WARNING**

Copyright © Monash University, unless otherwise stated. All Rights Reserved. This ebook is protected by copyright. For use within Monash University only. Users must not modify any content nor remove the copyright notice from this content under this licence. Other than as permitted under the Copyright Act 1968 (Cth), no covered material may be reproduced without the prior written permission of Monash University, except that you may save an electronic copy or print out parts of this website solely for your own information, research or study, provided that: you do not modify the copy as it appears on the Alexandria Repository platform; and you include the University's copyright notice and/or acknowledge any relevant third party sources. NOT FOR RESALE.

Disclaimer:

<https://www.monash.edu/disclaimer-copyright>

# Synopsis

Materials for the new [VCE Algorithmics](http://www.vcaa.vic.edu.au/Documents/vce/algorithmics/AlgorithmicsSD-2015.pdf) (<http://www.vcaa.vic.edu.au/Documents/vce/algorithmics/AlgorithmicsSD-2015.pdf>) study under revision for 2017 delivery.

# Contents

<b>Title</b> .....	i
<b>Copyright</b> .....	ii
<b>Synopsis</b> .....	iii
<b>1 Algorithmics - The heart of computer science</b> .....	1
<b>1.1 Why Study Algorithmics</b> .....	2
<b>1.2 What is an Algorithm?</b> .....	4
<b>1.3 The Muddy City Problem</b> .....	6
<b>1.4 Algorithmics versus Coding</b> .....	8
<b>1.5 Data: from messy to beautiful</b> .....	11
<b>2 Algorithmic Problem Solving</b> .....	15
<b>2.1 Graphs and Networks</b> .....	17
2.1.1 Modelling with Graphs .....	18
2.1.2 Modelling States of the World .....	25
2.1.3 Decision Trees .....	28
2.1.3.1 Sorting with Decision Trees .....	30
2.1.4 Formal Definition of Graphs .....	31
<b>2.2 Graph Algorithms: Revisiting the Muddy City</b> .....	35
<b>2.3 Searching in networks: a first look</b> .....	43
<b>2.4 How to write an algorithm</b> .....	48
2.4.1 Sequences .....	53
2.4.2 Variables and Assignments .....	54
2.4.3 Decisions .....	55
2.4.4 Repetition and Iteration .....	56
2.4.5 Blocks .....	59
2.4.6 Nesting .....	61
2.4.7 Abstraction and Modularization .....	62
2.4.7.1 Parameters .....	63
2.4.7.2 Return Values .....	65
2.4.8 Collections .....	67
2.4.9 Converting Pseudocode into Edgy .....	69
<b>2.5 Edgy: Revisiting the Muddy City</b> .....	84
<b>2.6 Breadth First Search and Depth First Search (2016)</b> .....	87
2.6.1 Browsing a social network using BFS .....	93
<b>2.7 Abstract Data Types</b> .....	94
2.7.1 Why ADTs Matter .....	95
2.7.2 The Graph ADT .....	97
2.7.3 Collection ADTs: Lists and Arrays .....	100
2.7.4 More Collection ADTs .....	105
2.7.5 Application: Graph Traversal .....	111
2.7.6 An extreme example: integers .....	113
<b>2.8 Networks of Actions: Planning and Decision Making</b> .....	114
2.8.1 Suspicious Boyfriends .....	115
2.8.2 Building the graph .....	118
2.8.3 Planning: putting it all together .....	121
<b>2.9 Path Finding</b> .....	123
2.9.1 Dijkstra's Shortest Path Algorithm .....	127

2.9.2 Bellman-Ford's Shortest Path Algorithm .....	131
2.9.3 Warshall's Transitive Closure Algorithm .....	134
2.9.4 Floyd's Algorithm for All-Pair Shortest Paths .....	137
<b>2.10 PageRank</b> .....	140
<b>2.11 Algorithmic Complexity: How fast is my algorithm?</b> .....	143
<b>2.12 Recursion</b> .....	144
2.12.1 What is recursion? A brief introduction .....	145
2.12.2 What is recursion: simple examples .....	146
2.12.3 Decrease and Conquer .....	149
2.12.4 How to draw a tree .....	151
2.12.5 Recursive tree search (video) .....	154
2.12.6 Recursive Graph Traversal by DFS .....	155
2.12.7 Analysing recursive algorithms .....	159
<b>2.13 Best First Search</b> .....	164
<b>2.14 Applied Algorithms (DRAFT)</b> .....	173
<b>3 Principles of Algorithm Design</b> .....	175
<b>3.1 Big-O notation: a brief introduction</b> .....	176
<b>3.2 Algorithm analysis: a primer</b> .....	177
<b>3.3 Divide and Conquer</b> .....	178
3.3.1 Binary Search Trees .....	189
<b>3.4 Complexity of Recursive Algorithms</b> .....	191
3.4.1 Solving a Recurrence Relation by Telescoping .....	194
3.4.2 The Master Theorem for Divide and Conquer .....	196
3.4.3 Recursive Complexity: An Example .....	198
<b>3.5 Dynamic Programming</b> .....	201
<b>3.6 Game Trees and the Minimax Algorithm</b> .....	214
<b>3.7 Computationally Hard Problems</b> .....	219
3.7.1 Travelling Salesman Problem .....	220
3.7.2 NP-Hard Problems: Soft Limits of Computation .....	224
3.7.3 Heuristics .....	227
3.7.4 Heuristics for the Graph Colouring Problem .....	233
<b>4 Turing Machines and the Origin of Computer Science</b> .....	242
<b>4.1 Hilbert's Program</b> .....	243
<b>4.2 Alan Turing and the Turing Machine</b> .....	245
<b>4.3 Busy Beavers</b> .....	246
<b>4.4 Undecidability and The Halting Problem: Hard Limits of Computation</b> .....	247
<b>4.5 A Weird Computational Formalism</b> .....	248
<b>4.6 A short Introduction to the General History of Computing</b> .....	249
<b>5 Searle's Chinese Room Argument</b> .....	250
<b>6 DNA Computing a la Adleman</b> .....	254
<b>7 Appendix A: Extension materials</b> .....	257
<b>7.1 Being Harry Houdini</b> .....	258
<b>7.2 Semantic Specification of Abstract Data Types</b> .....	260
<b>7.3 Tail Recursion</b> .....	261
7.3.1 Exercises: recursive list idioms in Edgy .....	266
7.3.1.1 Solutions: recursive list idioms in Edgy .....	268
7.3.1.2 Solutions: Append and Reverse in Edgy .....	271
<b>7.4 Testing Algorithm Correctness</b> .....	273
<b>8 Appendix B: The VCE study "Algorithmics"</b> .....	280

<b>8.1 VCE “Algorithmics” .....</b>	281
<b>8.2 Textbooks for VCE “Algorithmics” .....</b>	283
<b>8.3 Learning for Algorithmics and Computer Science .....</b>	284
<b>8.4 Recommeded Progression .....</b>	285
8.4.1 Recommended Progression – Unit 3 .....	286
8.4.2 Recommended Progression – Unit 4 .....	288
<b>9 Test Your Knowledge .....</b>	289
<b>9.1 Review Unit 3, Outcome 1 .....</b>	290
<b>9.2 Review Unit 3, Outcome 2 .....</b>	291



# 1

# Algorithmics - The heart of computer science

- [1.1 Why Study Algorithmics](#)
  - [1.2 What is an Algorithm?](#)
  - [1.3 The Muddy City Problem](#)
  - [1.4 Algorithmics versus Coding](#)
  - [1.5 Data: from messy to beautiful](#)
-

# 1.1

## Why Study Algorithmics

There is very simple answer to the question "why study algorithms?" Because we need to get things done! Algorithms are really just precise descriptions how a specific task has to be performed, or a particular type of problem can be solved.

It is easy to see why we need to have a precise description, in the case where we get a computer to perform the hard work for us. As you know, it needs a program, i.e. a description of how to perform the task. A program is what you get when you adapt an algorithm to a particular programming language and a particular kind of machine.

But the situation is no different when a human is doing the work. At some point, the human needs to be taught how to tackle this particular type of task, and to do so the procedure needs to be described precisely.

### Some examples

You have encountered many examples of this already. One of the earliest examples would have been learning addition in primary school, even though your teacher probably did not use the word "algorithm".

$$\begin{array}{r} 5 \quad 6 \quad 8 \\ 1 \quad 2_1 \quad 4 \\ \hline 6 \quad 9 \quad 2 \end{array}$$

The standard method for addition is to align the numbers according to place value and then to perform column-wise single digit additions working right-to-left. For the student to learn this method it needs to be described in full detail, step by step. In other words, it needs to be expressed as an algorithm.

It is generally straightforward to describe such problem-solving procedures. Or is it? Try the following activity and decide for yourself: [Being Harry Houdini](#)

Clearly, we would prefer to have reproducible procedures or "recipes" for problems of all levels of complexity. Consider a project manager who has to figure out how long a new project will take, given all the sub-projects, and the time required for each one. Some can be done in parallel, but maybe not all. An informed guess is not good enough when there's a firm delivery date. Surely the project manager would be happier to follow a set procedure to find the answer rather than having to figure out a way of solving this problem every time a new project has to be handled.

### Is it correct? Are you sure? Could it be faster?

It is crucial that our algorithms allow us to perform our tasks reliably. We'd like to be confident that the algorithm delivers the correct solution in all cases. Mistakes could be costly, or even fatal in extreme cases. If we want to be assured that the algorithm will deliver the correct result in all cases, we have to be able to reason mathematically about an algorithm before we trust it. It is simply impossible to test all possible cases even with a lot of time at disposal: there are usually infinitely many. Thus, we need to prove correctness. Algorithmics provides a formal way to do this.

But we are not only interested in correctness. We would also like to perform our tasks *efficiently*. A large part of Algorithmics is thus concerned with studying the resources that the algorithm requires, such as the time that it takes to run, or the amount of memory that it will consume. Here too, we need to study the efficiency of an algorithm mathematically, without having to test all possible cases. Complexity theory, another core part of Algorithmics, deals with these questions.

Abstraction and abstract reasoning are thus at the core of Algorithmics.

## Algorithmic problem solving

This new approach to reasoning is a powerful way to approach problem solving in general. When viewed abstractly like this, seemingly unrelated problems turn out to have deep structural similarities. This enables us to solve them in similar ways. Such general approaches to problem solving are called algorithm design patterns or design paradigms. They provide powerful tools that often allow us to construct efficient algorithms for problems that may appear extremely difficult at first sight.

Algorithmics could thus be defined as the art and science of constructing efficient formal procedures for problem solving and the study of their properties central properties: correctness and efficiency.

It should be evident that Algorithmics is at the heart of Computer Science. However, it is of more general interest than this even. Whenever we face a new type of problem, Algorithmics provides us with a powerful conceptual framework to tackle it.

There is another reason to study Algorithmics. It is fun, and it is intellectually challenging-just like solving the logical puzzles in the Sunday paper only much more useful!

---

## 1.2

# What is an Algorithm?

### The basic idea

This video by Alan Dorin explains the basic concept without getting too technical.



(<https://www.alexandriarepository.org/wp-content/uploads/WhatIsAlgorithm-Wi-Fi-High.mp4>)

License: Copyright © Monash University, unless otherwise stated. All Rights Reserved. (No derivatives)

**Activity:** Think of some other topic that interests you, and find an algorithm or create one of your own. Can you write down the sequence of steps?

### Official definition

An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. [[Wikipedia](http://en.wikipedia.org/wiki/Algorithm) (<http://en.wikipedia.org/wiki/Algorithm>)]

This means that there is a finite sequence of steps that can be followed mechanically to process specified inputs and correctly produce the expected output.

**Activity:** Consider the situation of withdrawing money from an ATM, and hypothetical algorithms

for doing this. (Imagine you have written careful instructions for an interstellar visitor who has never interacted with an ATM before.)

- One of the steps of the algorithm is: "raise your arm at a 60 degree angle, and move your body 10cm forwards, and slowly release your grip on your ATM card"
- You interpret "card" to mean a business card or playing card and try to insert that into the ATM
- The ATM displays a message: "No connection, please try again later". Slavishly following the algorithm, you inspect your watch and see that it is now later (by one second) and re-start the transaction, over and over again
- The ATM displays a message: "Sorry, you have reached your withdrawal limit". However, you proceed with your transaction and manage to withdraw \$100.

## Further Reading

- Chapter 2 of J. Hromkovic. Algorithmic Adventures: From Knowledge to Magic. Springer-Verlag, 2009.
- Chapter 1 of A.K. Dewdney. The New Turing Omnibus: 66 Excursions in Computer Science. W.H. Freeman, 1993.
- Chapter 1 of D. Harel. Computers Ltd. - What They Really Can't Do. Oxford University Press, 2000.
- [Wikipedia entry on Algorithms](https://en.wikipedia.org/wiki/Algorithm) (<https://en.wikipedia.org/wiki/Algorithm>)

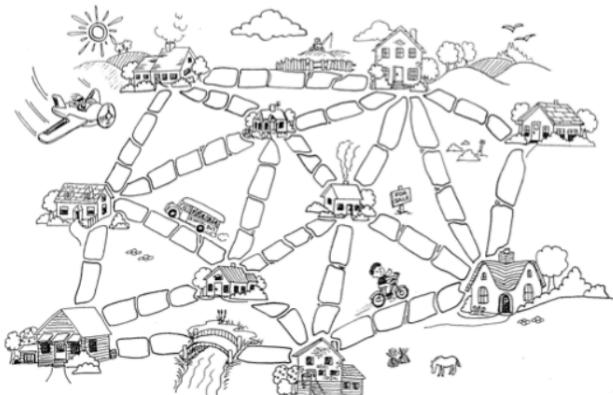
# 1.3

## The Muddy City Problem

[draft for preview only; BM]

Let's get started with designing an algorithm. In this problem, we want to build footpaths to connect the houses of a village.

Muddy City



(c) csunplugged.org 2010

In the picture above, you see the houses of the village and the paths that could be built between the houses. They do not exist yet; they are just possible paths at this stage. The cost of building a path is just the number of stepping stones that it takes to build it, as shown in the picture.

**Activity:** Using pen and paper, your task is to choose a subset of the paths that need to be constructed, so that it is possible to get between any pair of houses without getting muddy. However, we don't just want any solution, but the cheapest one, measured in terms of the total number of paving stones required.

1. Construct a network of paths that makes it possible to get from every house to every other.
2. Find the cheapest possible such network (count the number of paving stones it takes to build it).

### Writing down the algorithm

How did you solve the muddy city problem?

**Activity:** Write down the sequence of steps that was required. Now, working with a partner, give a sequence of instructions to construct the minimum cost network of paths that connects all the houses. Once you're confident about your method, write down the sequence of steps. Invent a new

village map and check that your algorithm gives the expected result.

Look at your steps again. Is each step detailed enough that it could be followed mechanically? Now exchange algorithms with another pair of students. Make up another muddy village map and apply the other team's algorithm to your map. Observe the other team as they do the same. No one is permitted to explain anything verbally or give any hints along the way! The written algorithm is the only source of information about how to solve the problem.

After doing this, you might want to improve the wording of your steps.

## A closer look

Inspect the resulting muddy village maps with their paths, and think about the following questions:

1. For an arbitrarily chosen pair of houses  $A$  and  $B$ , how many different routes exist between  $A$  and  $B$  along paved paths?
2. What happens to the network if one of the paved paths is removed?
3. Can you come up with a starting map for which your algorithm cannot provide a solution?

Supposing that you were to add one more paved path to your network. Try this, and observe that it creates a loop, or *cycle*. Look at the other paved paths that make up this loop. Pick one at random and remove it. Is it still possible to reach every house from every other house?

Look at that loop once more. Which path in the loop was the highest cost? Was it the very one that your algorithm left out?

The Wikipedia definition of algorithm talks about "calculating a function". What function is being calculated by your muddy city algorithm? Be careful to define the input(s) and output(s) explicitly.

*This module was adapted with permission from [csunplugged.org](http://csunplugged.org) (<http://csunplugged.org>), an exciting educational program from New Zealand that creates education resources for 'accessible' computer science.*

# 1.4

## Algorithmics versus Coding

### Coding is not Computer Science

As you may already know, computers can be instructed to carry out their work using a variety of different programming languages. Some well-known examples are Python, Java, and C++. In fact there are hundreds of programming languages; more are being developed all the time.

Computers are not like humans. When a computer follows a set of instructions, it does not know what it is doing. It cannot guess what the coder had in mind and work out how best to interpret the instructions to get a good result. If you have ever used the Quadratic Formula to solve a quadratic equation you will already know what this is like. You plug in the values and do some simplifications, painstakingly applying the rules until an answer pops out at the end. You do not need to know why the Quadratic Formula works. You can simply apply it, and "turn the handle" (a metaphor that comes from old-style adding machines).

Coding is just like that. We start with a method for solving a problem, i.e. the "algorithm". Coding is the task of translating the algorithm into the language that the computer can understand. Unlike coding, Computer Science is concerned with the algorithm itself, regardless of how it is expressed in a particular programming language. Computer Science can help us identify several different algorithms for solving the same problem, and help us understand why one algorithm might be more effective in a particular situation.

### Coding: An Example

Suppose that our problem is to process text documents and search for all words that end with "ing". We might come up with the following simple algorithm:

```

read the input
break each line into its words
for each word
    if the word ends with "ing"
        print the word

```

Although these steps are written in plain English, each step is precise, and each one is written on its own line, sometimes with indentation. This format for writing down an algorithm is known as **pseudocode**.

If we were to express this algorithm in Python, it would look like this:

```

import sys
for line in sys.stdin:
    for word in line.split():
        if word.endswith('ing'):
            print word

```

Don't worry if you do not understand the code above. It is just here to illustrate the increased complexity

of a program compared with an algorithm. The Python program starts with a line that tells the computer to use (`import`) the system library (a collection of standard methods, including one for reading input called `sys.stdin`). Even though the rest is somewhat similar to the pseudocode, there are many idiosyncrasies. You need to get the layout and punctuation exactly right, and you need to know when to use parentheses, and so on.

Here's the same algorithm expressed in Java. It looks even more complicated.

```
import java.io.*;
public class IngWords {
    public static void main(String[] args) throws Exception {
        BufferedReader in = new BufferedReader(new
            InputStreamReader(
                System.in));
        String line = in.readLine();
        while (line != null) {
            for (String word : line.split(" ")) {
                if (word.endsWith("ing"))
                    System.out.println(word);
            }
            line = in.readLine();
        }
    }
}
```

Although coding is important, and essential if you want to make a computer do something, it involves a lot of tedious detail and this detail often gets in the way of understanding the algorithm.

If we want to see what is going on, if we want to effectively communicate our ideas about how to solve a problem, it is better to use pseudocode. In Algorithmics, our challenge is to come up with the pseudocode for solving a problem. We will also want to be confident that our solution is correct and efficient. Once we have it, the interesting work has been done, and we can give the solution to a coder who can translate it into a form that a computer can use directly. For this reason, a subtitle for Algorithmics might be "Think Before You Code".

## Edgy

To enable us to experiment with algorithms and to execute them we will use an algorithm design environment that has been developed for VCE Algorithmics called *Edgy*. Edgy is a high-level programming language. It is so high-level that it looks almost like pseudocode. You can build programs in Edgy by snapping various building blocks together. The blocks only fit together when it makes sense to do so. For more information, please see the Edgy tutorials: [Programming with Edgy](https://www.alexandriarepository.org/syllabus/programming-with-edgy/) (<https://www.alexandriarepository.org/syllabus/programming-with-edgy/>).

## Summary

Computer Science is not the same as coding, or programming, or software engineering. Computer Science is the underlying body of knowledge to be mastered in order to be a competent programmer. It also provides a generic approach to problem solving that does not have to lead to programs at all.

The difference is stark. It's like the difference between mechanics (physics) and mechanics (car repair). Or, as one of the most famous computer scientists (Dijkstra) said:

"Computer Science is no more about computers than astronomy is about telescopes."

---

## 1.5

# Data: from messy to beautiful

You're probably used to thinking about data as messy. If so, you're probably thinking of experimental data, the result of making measurements, measurements which do not fit the theory as perfectly as you might hope. Sometimes, we can work out nice ways to present data (e.g. <http://www.informationisbeautiful.net/>), but when we see diagrams like these, we still know that the underlying data is likely to be complex and messy.

Computer Science gives us a rather different notion of data, one which is more elegant and mathematical. We don't mean mathematical in the sense of multiplication or division, or other ways to *operate* on data. We mean a precise way to structure data. Let's consider some examples to see what is going on here, and why it matters.

## Examples of Types and Operations

### Dates

Suppose you are writing a program to analyse historical events, and for each event you would like to incorporate the date of that event. What is a date? At one level it is a sequence of numbers and slashes, like "21/9/1997" or "21 September 1997". In the US this may be written "9/21/1997", and in China "1997/9/21". Of course, even the word "September" may be spelt differently in different languages. This string representation is not too satisfying because we need to know some context in order to interpret a date string. It's hard to work out if a pair of date strings refer to the same date. It's extra difficult to work out the interval between a pair of dates. Simply working out the date of the following day is tricky!

Other way to think about a date is as a triple of three integers, like (21, 9, 1997), along with conventions about how to print the date for different audiences. Similarly, it could be a dictionary that maps keys to values: {"day":21, "month":9, "year":1997}. Another popular method is to represent a date as a single integer, the number of seconds since midnight on 1 January 1970. Can you think of any advantages of this last representation?

### Cards

Suppose you have an algorithm for playing blackjack and you'd like to use it to win some money. Since your algorithm only improves your chances of winning by 1%, you need to play a lot in order to get the return that justifies all your work developing the algorithm. So you want to write a program that can play hundreds of online blackjack games simultaneously. You need to keep track of the cards in your hand. How would you represent even a single card, like the Four of Spades or the Queen of Hearts? You will already have guessed that a string like "Four of Spades" is not a good representation. Why not?

In blackjack, a card like the Four of Spades is worth 4 points. The same is true for the Four of Diamonds. The suit does not matter. Suppose we represent both of these cards as the integer 4. Then it would be easy to total the cards in a hand to check whether they had exceeded 21, the ideal score. However, your algorithm involves keeping track of the other cards that have been seen and the order in which they were seen, so you realise that you do need to pay attention to the suit. What would you do? How can you capture the full identity of the card, while making it easy to sum the values of cards?

## Primitive Types and Operations

The above examples illustrate that seemingly-simple types of data can be surprisingly complex! So we need to go back to basics.

There is a small set of so-called "primitive types" such as integers and strings. Notice how these are the types we referred to above when we tried to break down date and card information into more basic components. We can immediately formalise the operations on primitive types. For instance, when we add two integers, the result must be an integer:

$+ : \text{int} \times \text{int} \rightarrow \text{int}$

This is a *syntactic* rule. It uses the times symbol for "Cartesian Product", the set of all combinations of pairs of integers. It tells us that  $1 + 2$  is a syntactically well-formed expression and that it produces an integer:

$x = 1 + 2$

Since  $x$  is an integer, we also know that the following expression is well-formed too, and that  $y$  is also an integer:

$y = x + 3$

None of this tells us about the meaning (or "semantics") of addition. For this we need further statements known as axioms, such as the following:

$$\begin{aligned}x + 0 &= x \\x + (y + z) &= (x + y) + z \\x + y &= y + x\end{aligned}$$

This tells us that zero is the additive identity, and that addition is associative and commutative.

Let's think about another primitive type, namely strings. Can you think of a kind of addition operation that makes sense for strings?

```
x = "Monty"
y = "Python"
z = x + y
```

[What's the resulting value of  \$z\$ ?<sup>1</sup>](#)

This operation is known as concatenation. Can you write down the syntax and semantics for string concatenation?

[Check your answers.<sup>2</sup>](#)

## It's Not What They Look Like

Recall how we considered representing a date string "21 September 1997" using a triple of numbers (21, 9, 1997). This is a possible logical representation to be used by a computer program that is probably more

convenient for processing than a string. The string has its uses: it is probably a more effective visualisation of the date.

## Data Types

A data type is a set of elements (which may not be finite in size), along with various well-defined operations on those elements.

The integer data type is the set of integers  $\{\dots, -2, -1, 0, 1, 2, \dots\}$  along with operations like addition and subtraction.

The string data type is the set of possible strings  $\{\ "", "a", "b", \dots, "aa", "ab", \dots\}$  along with various operations like concatenation. What else do we need to know about strings? Length is useful; it reports an integer result, and we would write:

`len: str → int`

This doesn't tell us the semantics of `len`, just the type of result it produces. Consider how `len` and concatenation interact: take two strings  $x$  and  $y$ , then the length of the concatenation is the sum of the lengths, i.e.  $\text{len}(x + y) = \text{len}(x) + \text{len}(y)$ . This is an example of an axiom which ensures that any implementation of `len` behaves as we would expect.

By now you should be able to write down the definitions for other string operations. Try this for the operations in bold below:

1. **reverse**("abracadabra") = "arbadacarba"
2. **find**("rat", "scratch") = 2
3. "python"**[2]** = "t"
4. "foo" + "bar" == "foobar" = True

[Check your answers](#)<sup>3</sup>

## More Data Types

We can build up more complex data types out of these elementary ones. For instance, a playing card could be represented as a pair consisting of a suit (e.g. "hearts") and a value (e.g. 6). This is the complex type `str × int`. We will come back to types later on, under the heading of "Abstract Data Types".

(Note that we would have to explicitly include a space if we wanted one, i.e. "Monty" + " " + "Python")

<sup>1</sup> "MontyPython"

<sup>2</sup>  $x + "" = x$   
 $x + (y + z) = (x + y) + z$   
 $+ : \text{str} \times \text{str} \rightarrow \text{str}$

<sup>3</sup>  $\text{reverse}: \text{str} \rightarrow \text{str}$   
 $\text{find}: \text{str} \times \text{str} \rightarrow \text{int}$   
 $[]: \text{str} \times \text{int} \rightarrow \text{str}$

`==: str × str → bool`

# 2

# Algorithmic Problem Solving

## [2.1 Graphs and Networks](#)

### [2.1.1 Modelling with Graphs](#)

### [2.1.2 Modelling States of the World](#)

### [2.1.3 Decision Trees](#)

#### [2.1.3.1 Sorting with Decision Trees](#)

### [2.1.4 Formal Definition of Graphs](#)

## [2.2 Graph Algorithms: Revisiting the Muddy City](#)

## [2.3 Searching in networks: a first look](#)

## [2.4 How to write an algorithm](#)

### [2.4.1 Sequences](#)

### [2.4.2 Variables and Assignments](#)

### [2.4.3 Decisions](#)

### [2.4.4 Repetition and Iteration](#)

### [2.4.5 Blocks](#)

### [2.4.6 Nesting](#)

### [2.4.7 Abstraction and Modularization](#)

#### [2.4.7.1 Parameters](#)

#### [2.4.7.2 Return Values](#)

### [2.4.8 Collections](#)

### [2.4.9 Converting Pseudocode into Edgy](#)

## [2.5 Edgy: Revisiting the Muddy City](#)

## [2.6 Breadth First Search and Depth First Search \(2016\)](#)

### [2.6.1 Browsing a social network using BFS](#)

## [2.7 Abstract Data Types](#)

### [2.7.1 Why ADTs Matter](#)

### [2.7.2 The Graph ADT](#)

### [2.7.3 Collection ADTs: Lists and Arrays](#)

### [2.7.4 More Collection ADTs](#)

### [2.7.5 Application: Graph Traversal](#)

### [2.7.6 An extreme example: integers](#)

## [2.8 Networks of Actions: Planning and Decision Making](#)

### [2.8.1 Suspicious Boyfriends](#)

### [2.8.2 Building the graph](#)

### [2.8.3 Planning: putting it all together](#)

## [2.9 Path Finding](#)

### [2.9.1 Dijkstra's Shortest Path Algorithm](#)

### [2.9.2 Bellman-Ford's Shortest Path Algorithm](#)

### [2.9.3 Marshall's Transitive Closure Algorithm](#)

[2.9.4 Floyd's Algorithm for All-Pair Shortest Paths](#)

[2.10 PageRank](#)

[2.11 Algorithmic Complexity: How fast is my algorithm?](#)

[2.12 Recursion](#)

[2.12.1 What is recursion? A brief introduction](#)

[2.12.2 What is recursion: simple examples](#)

[2.12.3 Decrease and Conquer](#)

[2.12.4 How to draw a tree](#)

[2.12.5 Recursive tree search \(video\)](#)

[2.12.6 Recursive Graph Traversal by DFS](#)

[2.12.7 Analysing recursive algorithms](#)

[2.13 Best First Search](#)

[2.14 Applied Algorithms \(DRAFT\)](#)

---

## **2.1**

# **Graphs and Networks**

[2.1.1 Modelling with Graphs](#)

[2.1.2 Modelling States of the World](#)

[2.1.3 Decision Trees](#)

[2.1.3.1 Sorting with Decision Trees](#)

[2.1.4 Formal Definition of Graphs](#)

---

## 2.1.1

# Modelling with Graphs

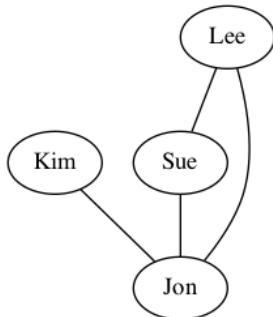
Graphs are one of the most fundamental and most widely applicable ways of modelling data that exists. A **graph** is the mathematical term used to describe a network consisting of a set of entities (or **nodes**) and the links between them (or **edges**). You will already be familiar with various kinds of networks:

- social networks, where each person is represented by a node, and where the friendship between two people is represented by an edge connecting the nodes
- the web, consisting of websites (the nodes) and hyperlinks (the edges)
- railway networks, consisting of stations and the railway lines that connect them

A very important aspect of modelling with graphs is that a node does not need to represent a physical entity as in the above examples. A node can also represent an abstract entity, such as a concept, or a state of affairs. Likewise edges can represent abstract relationships. You are probably already familiar with some types of such networks:

- semantic hierarchies, consisting of concepts (e.g. vertebrate) and the is-a relationship between them (a vertebrate is a kind of mammal)
- [mind-maps](https://en.wikipedia.org/wiki/Mind_map) ([https://en.wikipedia.org/wiki/Mind\\_map](https://en.wikipedia.org/wiki/Mind_map)), in which each node represents an idea, and edges represent an is-related-to relationship

We visualise a graph by writing down some nodes (here the names of a few people), and drawing lines between them.



Here are some more examples of graphs. Can you identify the nodes and edges?

- the brain
- the food web
- a computer network
- the airline network

**Activity:** Can you come up with more examples of graphs or networks? Think about the spread of an infectious disease. Or the sequence of subjects required to complete the VCE. Or how water gets to your kitchen tap. Or what is happening when you look up a reference in the back of a book.

Graphs are everywhere!

Activity: try to find something that you think can not be represented as a graph. Discuss with a friend the reasons why you think your example cannot be represented as a graph and see whether you can find ways around these (perceived) limitations.

## Representing the world

A graph is actually a mathematical structure, consisting of a set (the nodes) and a set of links between pairs of nodes (the edges). So the above graph could be written down as the pair of two sets: <{Lee, Kim, Sue, Jon}, {<Lee, Sue>, <Lee, Jon>, <Kim, Jon>}>. That's all there is to a graph.

We use graphs to model something of interest in the world, such as a social network. It's important to distinguish the mathematical model from what it represents. For example, the above graph contains four nodes, including one called "Kim". The node for Kim corresponds to a person in the real world, but it is not that person, just a mathematical abstraction. We use the label "Kim" to remind us of what it models. Similarly, the edge <Kim, Jon> is a mathematical abstraction, and tells us nothing about the nature of the friendship but just that a friendship exists. Notice that there's not a lot of detail in our model, compared with the real life situation it represents.

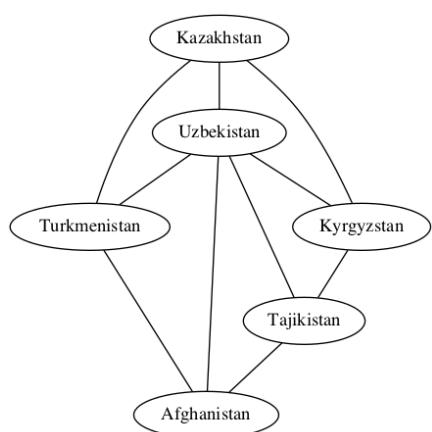
In general, when we solve an real world problem using an algorithm, we start by modelling salient aspects of the problem using a mathematical structure. Graphs happen to be a general purpose model for representing real world information. It can be an interesting and non-trivial task deciding how to construct the abstract mathematical representation of real world problem.

Let's take a quick look at part of the real world ([Al-Khwarizmi](#)

([https://en.wikipedia.org/wiki/Mu%e1%b8%a5ammad\\_ibn\\_M%C5%Ab%C4%81\\_al-Khw%C4%81rzm%C4%AB](https://en.wikipedia.org/wiki/Mu%e1%b8%a5ammad_ibn_M%C5%Ab%C4%81_al-Khw%C4%81rzm%C4%AB)) from whose name the term Algorithm is derived came from this area). Central Asia consists of six countries. Here they have been coloured with six different colours.



We can represent the countries of Central Asia, and their geographical arrangement, using the following graph:



In this graph, there is a node for each country.

### What do the edges represent?<sup>1</sup>

Now that we have this representation, which can be treated mathematically, we are in a position to think algorithmically about the geography of these countries and to define algorithms that process it. For example, we could apply a "graph colouring algorithm" which tries to assign a colour to each node in such a way that no two adjacent nodes (nodes connected by an edge) have the same colour. This, by the way, is one of the most famous algorithmic problems, and we will return to it later.

Activity: How many colours do you need at least to colour the map above such that no two neighbouring countries have the same color. How many colours do you think will be required in general for a map with  $n$  countries?

Note that our representation would not be useful for other problems, such as ordering the countries by land area, or working out which country contains the town of Khiva (al-Khwarizmi's birthplace). Our model excluded the information we would need in order to perform this task.

## Modelling the world with graphs

The following video explains the basic concepts of a graph and how graphs can be used as a versatile modelling tool.



(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-modelling-the-world-Wi-Fi-High.mp4>)

NB. the distance between two nodes in a graph is sometimes called *topological distance*. This is to distinguish it from actual distance in the real world. In general, we will use the term *distance* to mean *topological distance*.

## Review

After viewing this video, you should be able to answer the following questions.

Given a pair of nodes n<sub>1</sub> and n<sub>2</sub> in a graph, what is the maximum number of edges that can exist between n<sub>1</sub> and n<sub>2</sub>?<sup>2</sup>

What does it mean to say that graphs abstract away from irrelevant detail?<sup>3</sup>

What is the degree of a node?<sup>4</sup>

What is the distance between two nodes in a graph?<sup>5</sup>

What is a weighted graph?<sup>6</sup>

What is the distance between two nodes in a weighted graph?<sup>7</sup>

What does it mean for a graph to be connected?<sup>8</sup>

What is a directed graph (digraph)?<sup>9</sup>

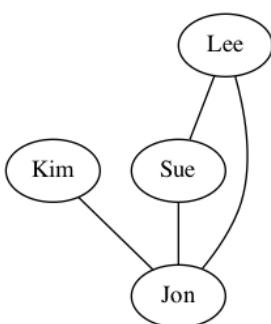
What is a multigraph?<sup>10</sup>

What is the difference between a graph and a visualisation of a graph?<sup>11</sup>

Activity: Can you identify some real-world structure and "read off" a graph? (We saw a road map, a maze, and a ferry route map in the video; so try to think of something different.)

## Choosing what to represent: social and antisocial networks

Earlier, we saw a diagram for a simple social network. Here it is again:



Can you write down this graph using the mathematical notation we saw at the start? Remember, a graph is defined using two sets, a set of nodes, and a set of edges. Edges can be specified using a pair of nodes, like  $\langle \text{Kim}, \text{Jon} \rangle$  or  $\langle \text{Jon}, \text{Kim} \rangle$  (the order doesn't matter). Write down the above graph using mathematical notation then refer back to the start of this module to check your work.

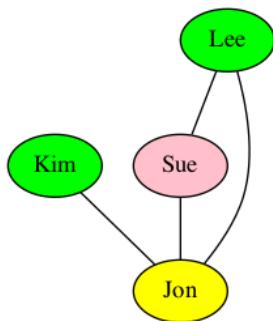
For the following you may want to review your understanding of the definition of a **set**.

Which of the following are sets?  $\{1, 2, 3\}$ ,  $\{\}$ ,  $\{1, 2, 2\}$ <sup>12</sup>

Suppose that all edges were removed from the above graph. What does this look like in the mathematical notation?<sup>13</sup>

Does it still meet the definition of graph?<sup>14</sup>

As we saw, the edges are intended to indicate the existence of a friendship. But it does not have to be that way. Suppose we used the edges to mean that two people *dislike* each other. We could call this an antisocial network. What does the *absence* of an edge mean exactly? (If your answer is "friendship", think again.) Now, let's use the colouring algorithm to colour this graph:



Now we can interpret the colours as table assignments. Everyone with the colour green (Lee and Kim) is assigned to the same table, while the others (Sue and Jon) are each assigned to separate tables. Our antisocial network representation has allowed us to solve the problem of allocating tables to people at a social event, where we want to avoid assigning certain people to the same table. Most interestingly, it allows us to do so with a known standard algorithm (graph colouring), so we have reduced a new problem to a solved one, simply by change of the representation.

Key points to note here are that:

- the same representation was used to mean two different things (friendship vs non-friendship, or "antipathy")
- our choice of representation affects what problems we can solve using the representation
- our choice of representation can also affect *how* we can solve the problem

## Another example: the Movie Database

The Movie Database is a collection of movies and TV shows. For each movie or show, it also has information about the directors and cast members. For each person, you can find out what show they are in. You can access it at [themoviedb.org](http://themoviedb.org) (<http://themoviedb.org/>). Take a look, and try to navigate between

people and shows.

If you were representing this information in a database table, it might look like this, and it would go on for hundreds of thousands of rows.

Show	Person
Wild (2014)	Reese Witherspoon
Wild (2014)	Laura Dern
Gone Girl (2014)	Ben Affleck
Gone Girl (2014)	Reese Witherspoon
...	...

Note that we're simplifying here, e.g. by not specifying the role of the person in the movie. For instance, Reese Witherspoon was an actor in *Wild* and a producer for *Gone Girl*.

Notice that the above table is really just a set of ordered pairs. We can represent this as a graph having two types of nodes. Can you draw a diagram for the fragment of the movie database that is shown in the above table? To finish, please take a look at [the interface to the social network of the Movie Database](http://marvl.infotech.monash.edu/webcola/examples/browsesmovies.html) (<http://marvl.infotech.monash.edu/webcola/examples/browsesmovies.html>).

- D<sup>1</sup>** The fact that the two countries connected by the edge have a common border.
- D<sup>2</sup>** A single edge if the graph is undirected, or two if it is directed.
- D<sup>3</sup>** It means that we capture the information that we need to solve a given problem. This may involve summarizing the original information/data, dropping aspects of it that are not required, etc
- D<sup>4</sup>** The number of edges that are incident on it.
- D<sup>5</sup>** The lowest number of edges that need to be traversed to get from one node to the other
- D<sup>6</sup>** A graph in which each edge is labelled with a (numeric) weight. This may, for example, represent a distance between two cities in a road network, or the cost of getting from one state to another in a state diagram.
- D<sup>7</sup>** The lowest possible sum of the weights of all edges that need to be traversed to get from one node to the other.
- D<sup>8</sup>** There is a path between every possible pair of nodes
- D<sup>9</sup>** A graph in which the edges have directions (usually depicted by arrows). Thus an edge (A, B) in a directed graph, also written as A->B, can only be traversed from A to B but not in the opposite direction.
- D<sup>10</sup>** A graph that can have multiple edges between the same pair of nodes. In a road network this could, for example, be used to represent different routes with the same start and end point.
- D<sup>11</sup>** A graph is a mathematical structure consisting of two sets: the set of nodes, and the set of edges between them. Its visualization is a convenient form of writing it in a readable form, but it adds additional (arbitrary) information. For example, the exact position of nodes and edges are meaningless, so is are the color and shape, and label of a node (if we give it one) etc.
- D<sup>12</sup>** Only the first two. A set can be empty, but it can never contain the same element more than once.
- D<sup>13</sup>** The set representing the edges would be an empty set.
- D<sup>14</sup>** Yes, it does. A graph can have no edges. It can even have no nodes! (In that case it can however never have edges, because a graph can only have edges between nodes that it contains).

## 2.1.2

# Modelling States of the World

### Modelling States

Many real-world problems involve a collection of entities, along with relationships between those entities. By now you will be quite familiar with these examples:

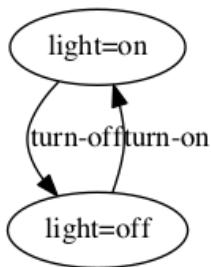
- social networks (node=people, edge=friendship)
- movie network (node=movie or person, edge=person is associated with movie)
- railway network (node=station, edge=railway line)

In these cases, it is easy to see a correspondence between real-world entities and nodes in the graph, and the correspondence between real-world relationships and edges in the graph.

However, graphs have much broader applicability.

### State Diagrams

A state diagram is a graph that represents the behaviour of a system. State diagrams use different types of graphical notations, but their common essence always is that the nodes represent a possible state that the system can be in and edges represent transitions between these states. As a simple example, consider the following system which has two states, and the *transitions* between them:



It is easy to find examples of more complex systems. For example, a washing machine allows you to select from several available "cycles", and each cycle consists of a sequence of steps, such as: wash, rinse, spin. Some machines have buttons that you can press repeatedly to switch between, say, fast-spin, normal-spin, and slow-spin. You can see some interesting examples if you search the web for "washing machine state diagram".

A node in a state diagram represents a particular state of a system, such as the fact that a washing machine is currently in its rinse state. However, this concept of "state of a system" is more general, and can refer to any particular state of affairs in the world. For instance, we can think of "it is Wednesday" or "it is sunny" as states.

**Important:** When deciding on what states to use, we need to be careful to model the fact that the real-world system can only be in one particular configuration at any one time. Imagine that instead of just one light you now have two that can be switched independently. To draw a state diagram for this "system" of lights you cannot just duplicate the diagram above. Why? Because we cannot be in two states at the

same time. The current node has to capture the entire state of the system (i.e. the state of both lights). The solution is to have four states:

1. (light 1 on, light 2 on),
2. (light 1 off, light 2 on),
3. (light 1 on, light 2 off),
4. (light 1 off, light 2 off).

Supposing the system we are modelling has two independent dimensions, e.g. day of the week (7 possibilities) and whether it is sunny or not that day (2 possibilities). How many different states do we need to model such a system?<sup>1</sup> How many reasonable edges would this state diagram have?<sup>2</sup>

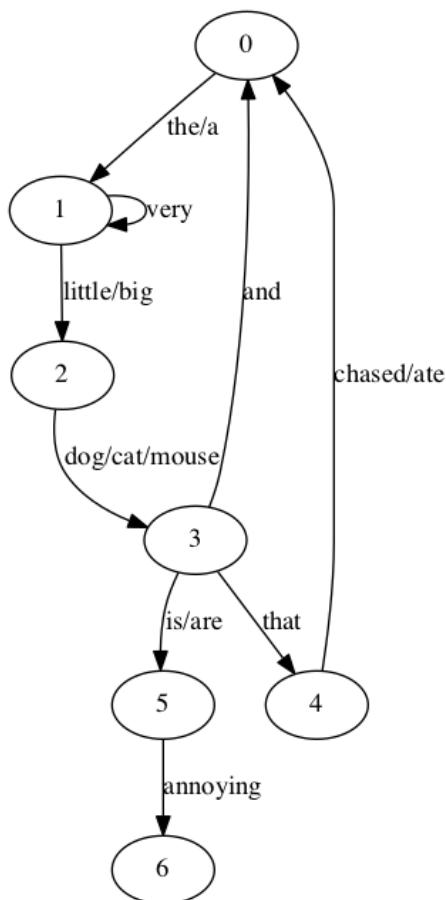
Activity: There are many more examples of state diagrams. Please take a few moments to research these and identify what the nodes and edges correspond to. Take care to identify the "system", the "particular state of a system", and the transitions between states.

1. a university degree consists of a series of courses that need to be done in sequence, where some are prerequisites of others;
2. a construction project involves a series of tasks, some of which can be done in parallel, and some of which must be done in series;
3. a game of tic-tac-toe begins with an empty  $3 \times 3$  board, and the first player places an O in one of the 9 available spaces, and so on;
4. an elevator can be called to one or more floors, and people get in and select the floors they want to go to.

Can you think of any more examples of state diagrams?

## A linguistic example

To give you an idea of the generality of state diagrams, consider the following example. It can be used to generate sentences. Each node can be thought of as representing the state you are in when you've just uttered a particular word, and are about to utter another word that can follow the previous word (according to the rules of English grammar). The edges are labelled with one or more words that can be uttered between the given states.



(This is a particular kind of state graph called a "finite state machine", and it is widely used for language processing and speech recognition by computer).

<sup>1</sup>  $7 \times 2 = 14$  states. You must have a separate state for each possible configuration: Sunny Monday, Rainy Monday, Sunny Tuesday, Rainy Tuesday, etc.

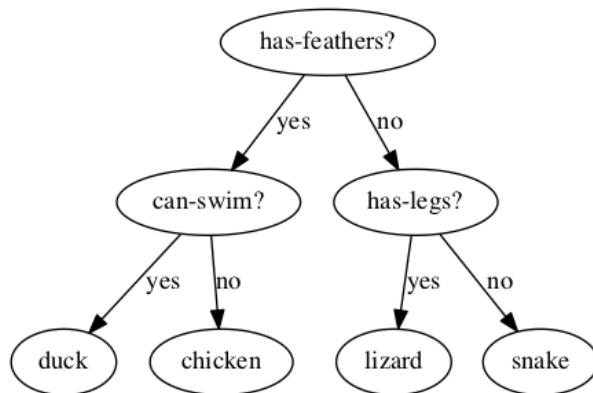
<sup>2</sup> If the weather does not change on any given day, you have  $7 \times 4 = 28$  transitions. For each pair of consecutive days you need four transitions: Sunny Monday -> Sunny Tuesday, Sunny Monday -> Rainy Tuesday, Rainy Monday -> Sunny Tuesday, Rainy Monday, Rainy Tuesday, etc. If the weather can change on a day, you need to add another  $7 \times 2 = 14$  edges: Sunny Monday -> Rainy Monday, etc.

## 2.1.3 Decision Trees

### Modelling the Decision Process

How do you decide what to do with your disposable time? Suppose you have a spare evening. What will you do? Is it more homework, going out with friends, reading a book? Reflect on a recent time when you made such a decision. What are the factors that influenced your decision? (e.g. mood, weather, whether a good movie just came out, ...). Did you need to weigh up any factors? (e.g. convenience of transport vs enthusiasm of a friend).

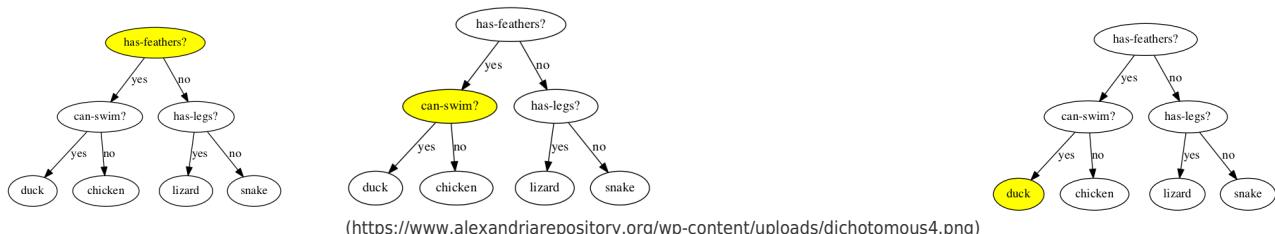
You have probably seen a decision tree in connection with games like 20 Questions, or a dichotomous key in biology. Here's an example:



Activity: Write down a decision tree to express a process for deciding what to do with a spare evening. It might have nodes with names like "school-night?" or "went-out-last-night?" Try to model your decision process as accurately as you can, within the constraints of the decision tree format. Hint: first enumerate the factors in your decision making, then prioritise them.

### Using a decision tree

If we have been given a decision tree, it is a simple matter to apply it. We start at the top, answer the questions, and work our way down to the "leaves", as shown in the following series of diagrams.



Note that only one state can be "active". This models the fact that there can only be one answer to the

yes-no question at each stage. (If you wanted to model the possibility that a user doesn't know the answer to a particular question, you need to add a third "don't know" option at every level, rather than allow multiple states to be active.)

## The size of the tree

Suppose that you're asked to guess a number between 1 and 100, and you can only ask yes-no questions like "is it less than 50?", or "is it the number 17?" A possible solution would be to ask a series of questions: "is it the number 1? is it the number 2? is it the number 3?" and so on, a total of 100 questions. What do we call this kind of solution?

A better solution is to halve the size of the problem at each step, a method known as "divide-and-conquer". (We'll see more about this in Unit 4). How many yes-no questions do you need in order to correctly identify the number?

Can you generalise this, and come up with a mathematical expression involving  $n$ , where  $n$  is the number of items in the set?

---

## 2.1.3.1 Sorting with Decision Trees

### Extension topic: How many decisions do we need to sort

We can use the concept of decision trees to calculate how much work a sorting algorithm must do if it is only based on direct comparison of two numbers, i.e. the only fundamental type of question we can ask is "Is  $x$  less than  $y$ ?"

Suppose you were given three unique integers,  $a$ ,  $b$ , and  $c$ , and needed to sort them into increasing order. The leaves of the decision tree are all the possible permutations, i.e.  $abc$ ,  $acb$ ,  $bac$ , etc. The interior nodes of a decision tree for this problem will represent questions like " $a < b$ ?".

Construct the decision tree for sorting a list of three unique integers.

How many questions do you need to ask, at most, in order to get the three integers into the correct order?

In general, with  $n$  items to sort, we will have  $n!$  ( $n$  factorial) possible orderings. In order to pick the correct one, we need to ask enough yes-no questions. The number of questions we will need to ask is going to be  $\log_2(n!)$ .

With the help of [Stirling's Approximation](http://en.wikipedia.org/wiki/Stirling%27s_approximation) ([http://en.wikipedia.org/wiki/Stirling%27s\\_approximation](http://en.wikipedia.org/wiki/Stirling%27s_approximation)), we can simplify  $\log_2(n!)$  approximately to  $k n \log(n)$ , where  $k$  is a constant. Thus, we know that, in principle, there can be no algorithm for sorting  $n$  items that takes less than  $k n \log(n)$  steps if it is based on pairwise comparison. If someone ever claims to have a faster algorithm, it must either contain a bug, or it must rely on a different operation to make comparisons. [Bucket Sort](https://en.wikipedia.org/wiki/Bucket_sort) ([https://en.wikipedia.org/wiki/Bucket\\_sort](https://en.wikipedia.org/wiki/Bucket_sort)) is one such "faster" sorting algorithm, it gains speed, but is limited in other ways. What precisely are the limitations of Bucket Sort?

---

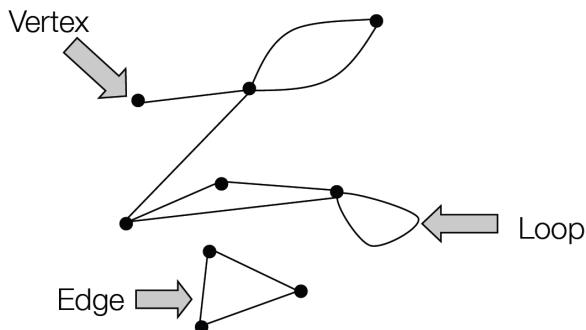
## 2.1.4

# Formal Definition of Graphs

[draft for preview only; BM]

### Terminology for Graphs

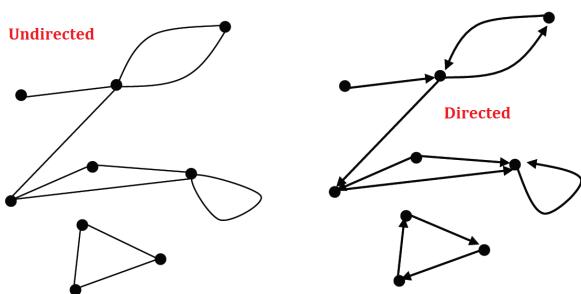
A graph can be considered as being made up of **nodes** (also known as **vertices**) and **edges** joining some nodes. It is possible for an edge to have the same source and destination nodes, making it a **loop**.



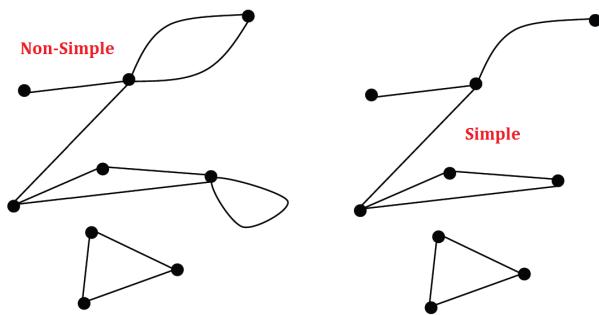
There are a number of important specific types of graphs distinguished by their properties:

**Undirected** graphs make no distinction between direction of travel along edges (both directions permissible). They simply indicate that two nodes are connected. For instance, if a graph depicts members of a social network and their friend relationships this would be an undirected graph. If A is a friend of B then B is also a friend of A (we call this type of relationship symmetric). We typically use simple lines as edges for undirected graphs.

**Directed** graphs are used when the direction of traversal of an edge is important. A typical example would be a road network which contains one way roads: Just because you can get from A to B does not mean that you can also go from B to A. Directed edges are typically shown by arrows. Note that we can always convert an undirected graph into a directed graph: we simply have to replace each undirected edge by two directed edges.



**Simple** graphs only have a single edge between any pair of nodes and no self-loops. **Non-simple** graphs can include multiple edges between two nodes, and they can include self-loops. We will mostly be concerned with simple graphs.



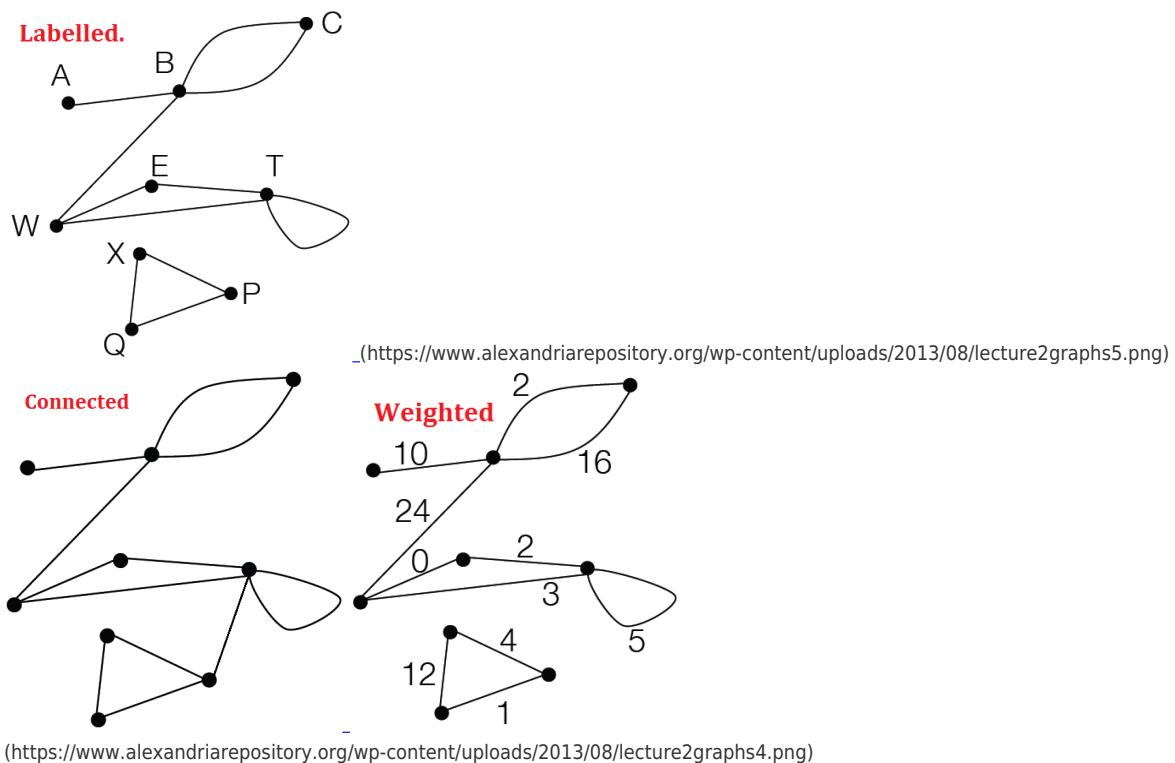
A **path** between two nodes  $A$  and  $B$  is a sequence of nodes that are connected so that we can move from  $A$  to  $B$  in the graph. The sequence must thus start with  $A$  and end with  $B$  such that all adjacent nodes in the sequence are also **adjacent** in the graph, i.e. connected by an edge (if this edge is directed, it must obviously lead from the first to the second). Note that this way of specifying a path is restricted to simple graphs. In a non-simple graph we would have to include the edges in the path.

A graph is **connected** if we can find a path between any pair of nodes. In a **disconnected** graph, at least two nodes will exist for which this is not possible.

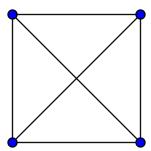
**Weighted** graphs have a numerical value associated with edges, which may correspond to distances or costs. For example, if the graph models a road network, a weight may denote distance or travel time.

**Unweighted** graphs exclude these numerical values.

**Labelled** (as opposed to **unlabelled**) graphs use letters or symbols to label nodes or edges such that specific references can be made.



A **complete** graph is a graph that has edges between each pair of nodes. For  $n$  nodes we usually abbreviate it as  $K_n$ . For example,  $K_4$  is the complete graph with four nodes:



**Trees** are graphs that are defined with the following properties:

1. Simple
2. Connected
3. Contain no circuits

**Note:** A circuit is present in a graph if there exist at least two distinct paths that can be used to go between the same two nodes. A simple but very crucial property of a tree is the following: whenever you add an edge between two nodes in a tree you generate a circuit.

## Mathematical Notation for Graphs

Mathematically, we can specify a simple, unlabelled graph  $G$  as a pair of two sets, a set of nodes  $N$  and a set of edges  $E$ :

$$G=(N, E)$$

Each node can be specified by an arbitrary unique identifier, for example letters or numbers. Edges can simply be specified as a pair of node identifiers. In connected graphs we use the convention that the pair is ordered and the edge is directed from the first node in the pair to the second one.

For example, the complete graph  $K_4$  from above can be given as:  $G=(N, E)$  with

$$N=\{1, 2, 3, 4\} \text{ and } E=\{(1,2), (1,3), (1,4), (2, 3), (2, 4), (3,4)\}.$$

It could also be specified with any other set of identifier for the nodes, for example

$$N=\{a, b, c, d\} \text{ and } E=\{(a,b), (a,c), (a,d), (b, c), (b, d), (c,d)\}.$$

You may have seen other ways to specify graphs mathematically, for example with a so-called incidence matrix. For its simplest form we identify the nodes of a directed graph with numbers  $1\dots n$  and give a matrix  $M$  of zeros and ones such that

$$M_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are connected} \\ 0 & \text{otherwise} \end{cases}$$

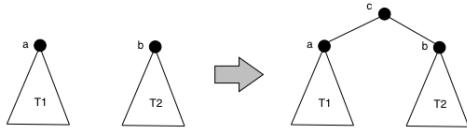
This is all the information we need for a directed, unweighted graph.

[How would you represent a weighted graph with an incidence matrix?](#)<sup>1</sup>

[What can you say about the incidence matrix of an undirected graph?](#)<sup>2</sup>

## Formal Definition of Trees

Above we have already specified trees as a special type of graph by listing their properties: simple, connected, and circuit-free. We can give a very elegant alternative definition in the following way: If we have two trees  $T_1, T_2$ , we can construct a new, bigger one by adding a new root and connecting the roots of  $T_1$  and  $T_2$  to the new root.



1. A graph  $G = (a,)$  is a tree with root  $a$ .
2. If  $T_1 = (V_1, E_1)$  is a tree with root  $a$ ,  $T_2 = (V_2, E_2)$  is a tree with root  $b$ , and  $c$  is a unique new node, then  

$$G = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(c, a), (c, b)\})$$
is a tree with root  $c$ .

Such a definition is called *inductive*. Such definitions are often very useful when thinking about algorithms, because they are often *constructive*: the definition above basically tells us how to build a tree (and thus gives rise to algorithms that need to construct trees). We will return to this later.

<sup>1</sup>

Instead of zero/one entries the entry for the edge from  $i$  to  $j$  would be its weight.

<sup>2</sup>

Since an undirected graph  $G$  is identical to a directed graph  $H$  in which each edge of  $G$  is twice (once for each direction), the incidence matrix of an undirected graph must be symmetric (along its diagonal). We thus really only need to represent either the upper or lower triangular matrix.

## 2.2

# Graph Algorithms: Revisiting the Muddy City

[draft for preview only; BM]

## Graph Algorithms

We have looked at the questions 'What is an algorithm' and 'what is a graph', so we are now in a position to put these two together and lay the foundation for the rest of the unit. The question obviously is, "what is a graph algorithm". We can apply two interpretations: (1) a systematic method to answer a question *about* a network or graph or more generally to solve a problem involving a graph; (2) a systematic method that solves a problem *using* a network or graph representation. Most graph algorithms exhibit both interpretations.

Note that some algorithms exhibit one interpretation but not the other. For example, algorithms to interpret natural language expressions or visual scenes routinely use graph representations internally but they are not *about* networks. On the other hand, an algorithm that solves a problem for a network without using some form of graph representation is difficult to imagine, since the information about the problem must be captured somewhere after all.

The example that we will discuss now fulfils both criteria: it solves a real-world (toy world) network problem and it uses a graph representation to do so.

## Minimum Spanning Trees

Recall the [Muddy City Problem](#) that we have solved informally earlier.

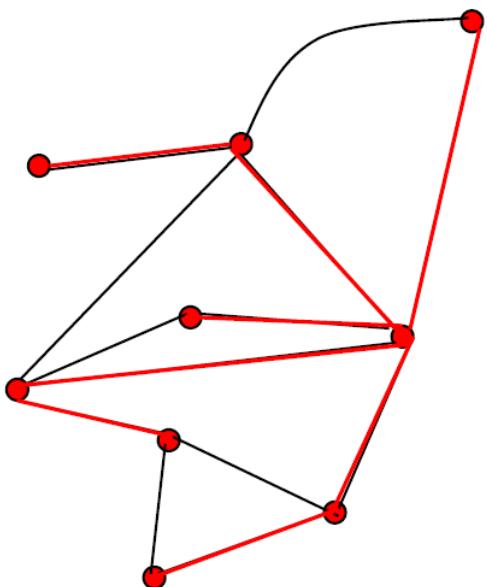
The type of network or graph that we have constructed in the *Muddy City* example is known as a spanning tree. A tree that allows us to reach every node in the graph from every other node.

Let us make this precise (and before we proceed, make sure you know the formal definitions for graphs set out in the module [Formal Definitions of Graphs](#).

A **spanning tree** of a simple, connected graph  $G$  is a tree that:

1. Contains all the nodes of  $G$ , and
2. All the edges of the tree are edges of  $G$

**Figure 3.** A sample spanning tree (in red) of a graph (in black).



**Start Quiz** (<https://www.alexandriarepository.org/app/WpProQuiz/3>)

Generally a spanning tree is not unique. There can be many spanning trees for a given graph.

An important form of spanning tree is the **Minimum Spanning Tree (MST)**, which is only defined for weighted graphs. An MST  $T$  is a spanning tree such the sum of all edge weights in the graph is as small as possible.

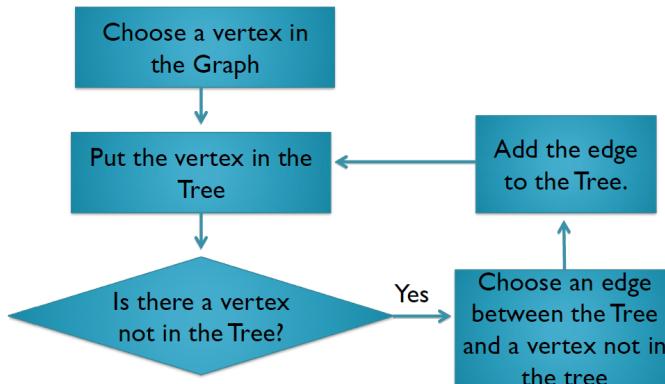
Strictly speaking mathematicians and computer scientist would not accept this as a strict definition because "as small as possible" is not well defined. A better definition is the following.

Let  $w(X)$  be a function defined on trees that returns the sum of all edge weights in  $X$ . Let  $T$  be a spanning tree of  $G$ .  $T$  is a *minimum* spanning tree of  $G$  if there is no other spanning tree  $T_1$  of  $G$  with  $w(T_1) < w(T)$ .

Given a graph, how do we find a minimum spanning tree?

## Prim's Algorithm

Consider the following algorithm:



It works by growing a spanning tree node-by-node, so that it maintains a partial spanning tree at any point of time.

This partial spanning tree starts from an arbitrary node. In every step of the algorithm, one new node is added to the partial spanning tree. This can be done by considering all nodes that are not yet in the partial spanning tree and that are directly connected to some node in the spanning tree by an edge (i.e. that are adjacent to some node in the spanning tree). The algorithm picks one such new node and one of the edges that connect it to the tree and adds these to the partial spanning tree. It proceeds in the same way until all nodes of the graph have been added to the tree.

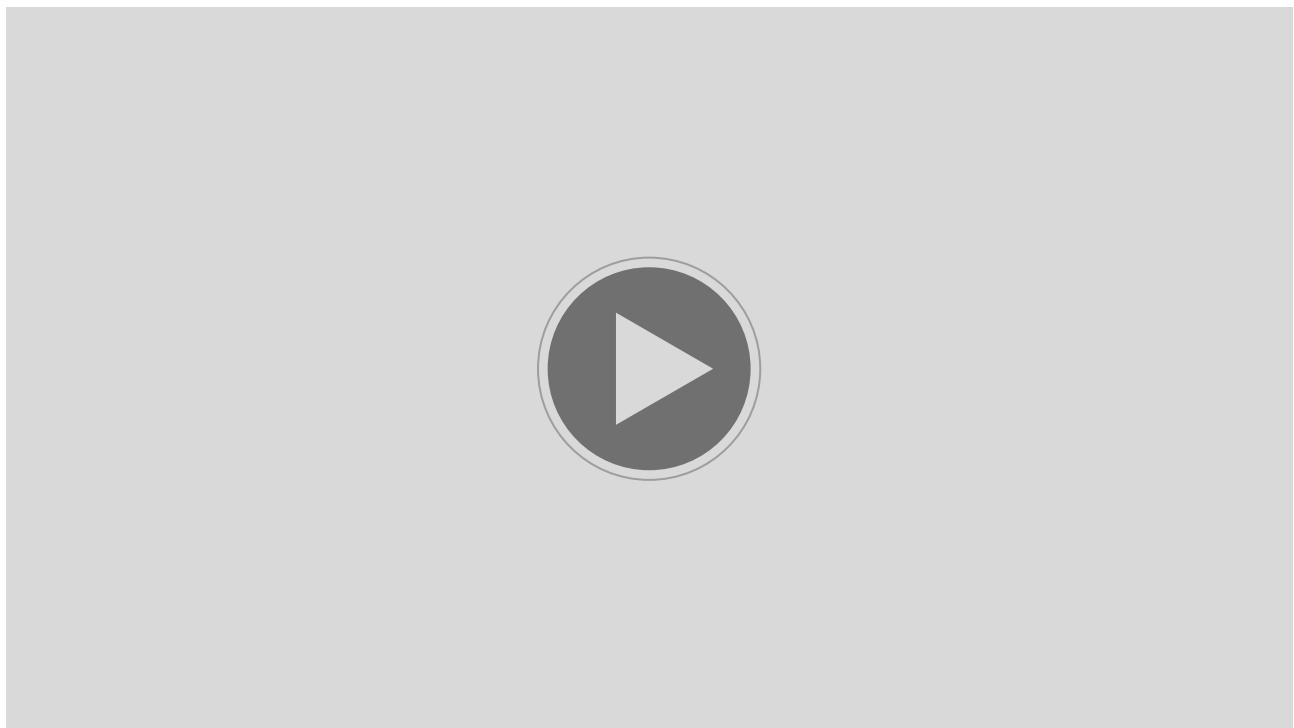
The selection of nodes in this algorithm is arbitrary. Different choices may lead to different spanning trees.

Note that at each stage of the loop we have a tree. Each time through the loop we add a node to the tree. Since the algorithm only terminates when there are no unconnected nodes, the final result will be a tree containing all nodes from the original graph.

The visualization below illustrates the construction of a spanning tree. Select a node to see all possible edges. Click the right arrow symbol to trigger the next step (note that you need to click twice to before the next edge is added).

 **Click to open**

We can modify this algorithm in a very simple way to obtain the minimum spanning tree. Instead of picking an arbitrary new node to add to the partial spanning tree, we consider all edges that connect nodes in the partial spanning tree to nodes outside of it and pick the edge with the lowest edge weight to extend the partial tree.



(<https://www.alexandriarepository.org/wp-content/uploads/Prim-2-Wi-Fi-High.mp4>)

This revised Algorithm will always construct the minimum cost spanning tree. It is known as [Prim's](#)

[Algorithm](http://en.wikipedia.org/wiki/Prim's_algorithm) ([http://en.wikipedia.org/wiki/Prim's\\_algorithm](http://en.wikipedia.org/wiki/Prim's_algorithm)) or as the Prim-Jarník algorithm. It was discovered by the Czech mathematician Vojtěch Jarník in 1930, later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959.

You can explore the algorithm further with a very nice interactive animation available at <http://www.cs.usfca.edu/~galles/visualization/Prim.html>.

Intuitively, the algorithm seems to make sense, but how can we be sure that it always and without fail constructs the minimum spanning tree? Isn't it possible that the choice of a somewhat longer edge will later be rewarded by allowing us to use a much cheaper edge at some future point in the construction? This question, which can be rephrased as "Is the algorithm correct" (i.e. does it always compute what it is supposed to compute) is the most important question we can ask about an algorithm. The only way to gain certainty is to prove that the algorithm is correct.

## Is Prim's Algorithm Correct?

Prim's method belongs to a special type of algorithms, which are called **greedy** Algorithms. Such algorithms proceed by trying to achieve the maximum gain at every single step. They rely on locally optimal choices and never attempt to work for a "delayed reward", i.e. they never accept a reduced gain at some step in order to be able to achieve a higher gain later. In the case of Prim's algorithm this means: we take the cheapest edge immediately (maximum savings). We do not even consider any other edge in order to potentially make a bigger gain later. Greedy algorithms are short-sighted.

Where they work, greedy algorithms are often very good: making only very few and simple choices they are typically fast. But how can we be sure that the greedy method works for a given problem?

We can of course test the method on any number of examples actively trying to construct an example that makes it fail. We can even implement the algorithm as a computer program and do large numbers of such tests automatically. In this way we can gain confidence that the method works (unless we find an example to show that it does not always work), but we still cannot be absolutely sure. Maybe we just didn't find a counter example. This, unfortunately, applies to all forms of testing.

Generally, however, we would like to be sure that our methods work, and in some cases we must be sure: imagine you are writing a control algorithm for a nuclear power plant or an aircraft landing system.

The only way to be sure is to give a proof. The video below sketches a proof that Prim's algorithm is definitely correct, i.e. that it produces a Minimum Spanning Tree (don't worry, the video streaming works, the whiteboard is blank until 0:55). Of course, the video only sketches the proof idea, it is not a full formal proof. Since we have not yet even learned how to formally write down an algorithm (let alone how to formally prove a property of an algorithm), we will leave it at that. The important aspect to understand is that it is possible to prove correctness of an algorithm, and that this is quite different from "testing" it (or the corresponding program) for a number of cases. So whenever the correctness of our algorithms is critical (for example, when they control a medical equipment), we will want to be certain that they function right and only a proof can give us this certainty.



(<https://www.alexandriarepository.org/wp-content/uploads/Proof-Sketch-Prim-Wi-Fi-High.mp4>)

## Other Minimum Spanning Tree Algorithms

There are a number of other [minimum spanning tree algorithms](http://en.wikipedia.org/wiki/Minimum_spanning_tree) ([http://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree)), most importantly [Kruskal's algorithm](http://en.wikipedia.org/wiki/Kruskal%27s_algorithm) ([http://en.wikipedia.org/wiki/Kruskal%27s\\_algorithm](http://en.wikipedia.org/wiki/Kruskal%27s_algorithm)) and the Reverse-Deletion Algorithm.

The Reverse Deletion Algorithm is essentially just the inverse of Prim's algorithm: It starts from the full graph under consideration and removes edges from it one-by-one.

### Which edge should be chosen for removal at each step?<sup>1</sup>

If this edge would disconnect the graph it is never considered again and the algorithm proceeds with the next lower weight.

Kruskal's algorithm grows several independent parts of the graph separately. Thus it does not grow a single tree but a forest. It picks edges one by one in increasing order starting from the cheapest one. Unless an edge introduces a circuit, it and the two nodes it connects are added to the growing forest. The algorithm ends when all nodes have been added and the structure is fully connected, i.e. when the forest has turned into a single tree.

Kruskal's algorithm is much more involved to implement, but it can be made very fast if implemented cleverly.

## An Inductive Definition of MCSTs Derived from Prim's Algorithm

When we formally defined the notion of a tree in the module Formal Definitions of graphs, we used an

inductive definition and noted how an inductive definition can often give direct rise to an algorithm.

Let us illustrate this with a (semi-formal) inductive definition of a minimum spanning tree MST for a graph  $G=(V1,E1)$ .

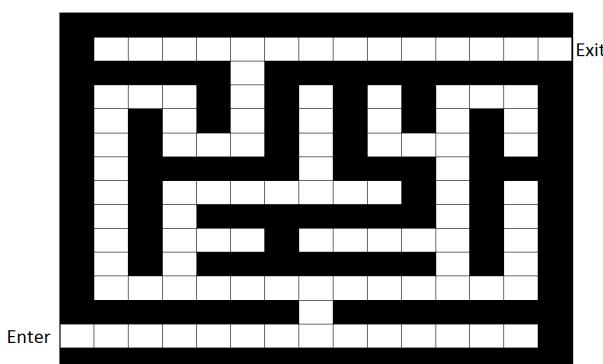
1. If  $x$  is a node in  $V$ , then  $T=(\{x\}, \{\})$  is a partial MST for  $G$ .
2. Let  $T=(V2, E2)$  be a partial MST of  $G$ .  
Let  $E3$  be the subset of edges  $E1$  that connects nodes in  $V2$  with nodes in  $V1-V2$ .  
Let  $e=(x,y)$  be the edge in  $E3$  with the lowest edge weight.  
Then  $T1=(V2 \cup \{x, y\}, E2 \cup \{e\})$  is a partial MST for  $G$ .
3. Let  $T=(V2, E2)$  be a partial MST for  $G=(V1, E1)$ . If  $V1=V2$  then  $T$  is a Minimum Cost Spanning Tree of  $G$ .

You can see that this definition directly tells us how Prim's algorithm works! Of course we only know that this correctly defines a minimum spanning tree, because we have already proven that Prim's algorithm is correct.

## An Application of Minimum Spanning Trees: Creating Mazes

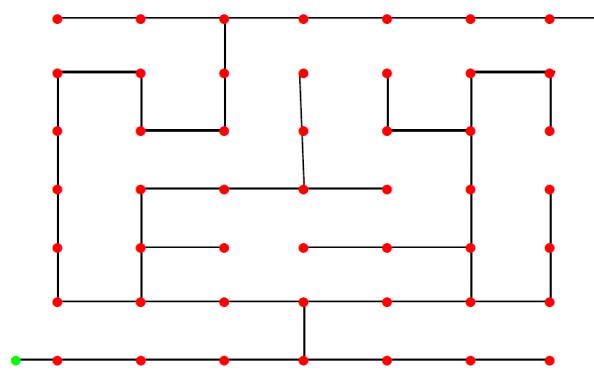
There are many relatively obvious applications of spanning trees that focus on creating fully connected networks of some type (traffic, social, etc). However, spanning trees also have many less obvious applications. An entertaining one is the construction of mazes...

**Figure 1.** A sample maze



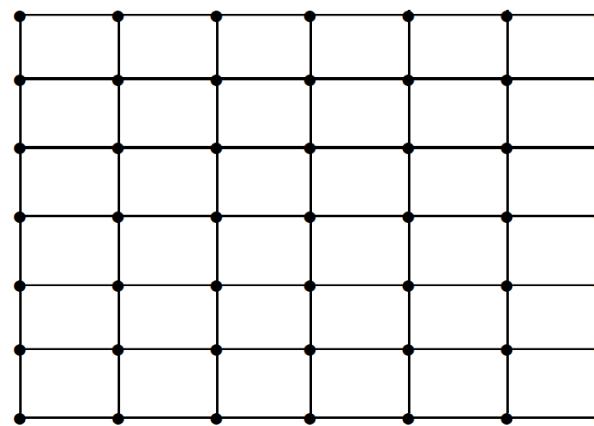
Consider the sample maze given in **figure 1**. This can be simplified (or abstracted) by mapping out the nodes, as well as the paths between each node. This representation is a **tree**.

**Figure 2.** The sample maze now represented as a tree.

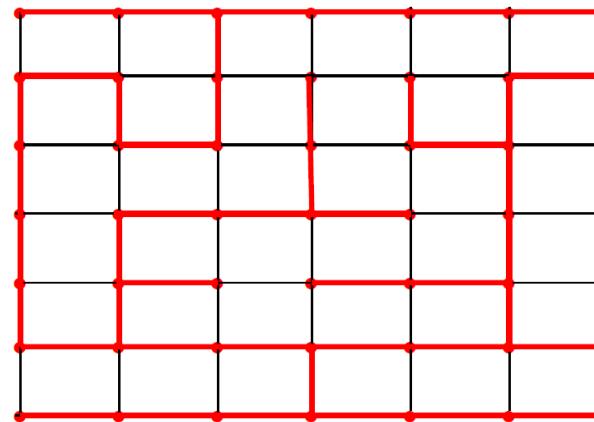


Using the concepts of graphs and spanning trees, it is simple to create a procedure for constructing mazes:

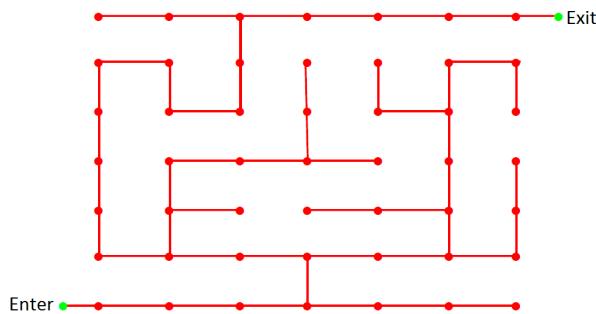
1. Start off with a full grid as below. This is just another type of graph...



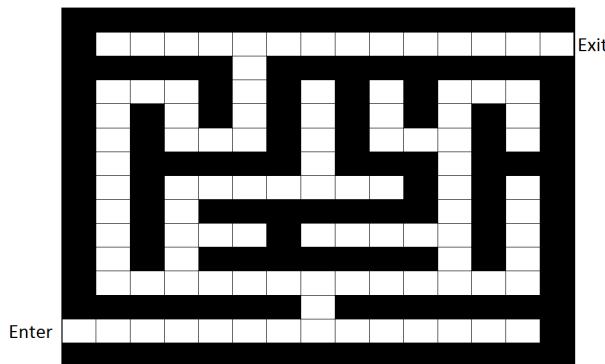
2. Find a spanning tree of the grid



3. Mark entrance and exit vertices.



4. Add in walls and remove the spanning tree



Although this algorithm cannot create mazes of all types (e.g. cannot include a circuit) and is not very detailed here, it demonstrates the usefulness of graphs as well as how a seemingly difficult problem (generating a maze) can be reduced to quite a simple algorithm.

**Start Quiz** (<https://www.alexandriarepository.org/app/WpProQuiz/4>)

Even though we have explained in natural language (and using drawings) how Prim's Algorithm works, what we have not done yet is to write it down in a precise way. However, we will delay this a little bit because we need to develop a language to do so first. Before we do this, let us look at another example of a very fundamental graph algorithm.

<sup>1</sup>

Since the algorithm is the dual of Prim's algorithm, it should remove the edge with the highest edge weight provided that this does not disconnect the graph.

## 2.3

# Searching in networks: a first look

### Connectivity

The most basic questions that we can ask about a real-world network are about connectivity: Is there a bus connection from home to school? Are two points in an electric circuit connected? Is there a chain of friends on facebook that links me to the person I just met?

The same goes for a graph as an abstract model of information. Its essence is to capture information about the connections between entities, whether the entities are people, objects, ideas, concepts, ... Its nodes can stand for physical objects (such as cities in a transport network), states (such as in a game tree) or conceptual entities (such as in a governance chart). The edges always model a physical or abstract relationship between the nodes.

There are two ways for a pair of nodes,  $A, B$ , to be connected. In the simplest case of a direct connection, we only need to check if there is an edge between  $A$  and  $B$ . In general, however, we may need to follow a path through other nodes in order to get from  $A$  to  $B$ .

Our notion of connectivity here is *transitive*. This means that if  $A$  is connected to  $B$ , and  $B$  is connected to  $C$ , then  $A$  is also connected to  $C$ . One point of a social network is to connect people transitively. Thus, if Chris is a friend of Anna, and Anna is a friend of James, then James belongs to the (extended) network of Chris' friends, and at some point, they may be introduced and become friends as well.

Suppose that you are the employee of a large company, Exploitation Inc., but you are not very happy in your job. Having already looked around for other positions you have decided that you would like to work for Gooditwoshoes & Co. Unfortunately you do not know anyone at Gooditwoshoes who could help you get a job there. Luckily though, you are a user of a large social network that links professionals (similar to linkedin.com). The network captures information about people (its users), the companies they work for, and their roles in the company. Most importantly it also captures links between people (friendships, professional relationships).

### Paths

The theory of "[six degrees of separation](http://en.wikipedia.org/wiki/Six_degrees_of_separation)" lets you suspect that there must be a relatively short chain of direct connections that links you to the CEO of Gooditwoshoes. If only you knew this chain, you could ask the first person (who you know directly) to introduce you to the second. Then, once you've met the second you can ask them to introduce you to the third, and so on until you get to meet the influential CEO herself. (If you are a member of LinkedIn you will notice that they suggest these types of introductions in the sidebar when you access someone's profile).

Such a chain of connections is called a *path*. You are already used to solving path finding problems in real life. You do this when you use a map to navigate from one city to another: To get from Melbourne to Albury you first drive from Melbourne to Kilmore, then from Kilmore to Euroa, then from Euroa to Wangaratta, and finally from Wangaratta to Albury. However, there are some important differences when you find a path on a map. First, you can use some clever way of guessing the right direction, because the cities have a location in space. In most road networks it is usually a good guess to go in the approximate direction of your destination if you want to get there quickly.

In a social network this is not possible, because the nodes have no physical position. "Distance" in the immediate (geometric) sense is simply a concept that does not apply. This makes the task much harder. Secondly, when a human looks at a map he or she always has some kind of global view of the connections on the map even if it may be partial. You do not have this when you are dealing with a social network. The only information you can directly obtain are answers to questions of the type: "Are X and Y friends" (i.e. is there an edge between X and Y) and "Who are the friends of X" (i.e. to which other nodes Y is X adjacent - connected via an edge). Finding a path using only this information is like trying to find a route on a map of which you can only inspect a small window at a time. To make things more difficult, we often have to deal with huge networks.

What we need is a precise, automated method to automatically find a path in networks of any size based on such localised information, or else to report that none exists: an algorithm for path finding. How can we go about designing one?

## Path finding

It seems reasonable to start the search for a path from  $A$  to  $B$  at node  $A$  and to first check all immediate neighbours of  $A$ . If we find  $B$  as one of the neighbours, our job is done. Otherwise we simply do the same thing for all the neighbours, i.e. we check their neighbours. If there is a path from  $A$  to  $B$  we must ultimately arrive at  $B$ .

[Why would it be difficult to use the method we have just described to search for a path by physically moving through a network of paths, for example to search for the exit in a maze? <sup>1</sup>](#)

This seems easy if we are dealing with a *directed acyclic graph*.

[What additional difficulty would we encounter in an undirected graph? <sup>2</sup>](#)

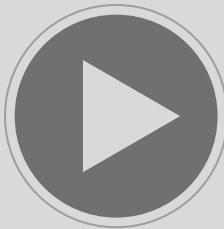
The essential question is thus: how do we keep track of which nodes we have already investigated and which we need to investigate next, so that we can sequence the visits and do not end up going around in circles?

We can observe that there are three types of nodes:

- Nodes for which we have already processed all neighbours. We will call these "visited" nodes.
- Nodes which we have seen as neighbours of other nodes, but that we still have to process. We will call these "to-be visited" nodes.
- Nodes that we have not yet considered at all. We will call these "unvisited" nodes.

We could simply mark these nodes with colours to keep track of what we have done. Let us mark visited as red, to-be-visited as green, and leave unvisited unmarked.

Let's give it a go:



(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-first-traversal-Wi-Fi-High.mp4>)

Now let's try to express this as an algorithm. It is quite easy to do this, as we are just expressing the steps in natural language, albeit a slightly formalised style of natural language. Our pseudocode here is quite verbose, and we will soon see more concise ways of expressing our ideas in pseudocode.

**Algorithm search**

input: two nodes A, B

output: "yes" if there is a path from A to B, "no" otherwise

mark A green

repeat everything between begin and end

    until there are no green nodes left:

        begin

            let C be an arbitrarily chosen green node

            if C is the same as B answer "yes" as output and stop

            otherwise mark all neighbours of C green

                unless they are red already

                remove the green mark from C

                mark C red

        end (\* of what is repeated \*)

    answer "no" as output and stop

Note that there is a minor difference between this algorithm and what we have done in the video above. The algorithm explicitly removes the green mark from a node when it is marked as red. In the video we didn't do this (because it is too messy on the whiteboard). Thus, whenever we picked a new green node, we actually had to pick a "green one that is not also marked red".

You could try to rewrite the algorithm above so that it behaves exactly like what we did in the video.

The first three lines of the description give the algorithm a name (so that we can refer to it by name), and define what it needs to know in order to run (the input) and what it is expected to produce as output. The remainder details the steps that the algorithm takes. The only tricky bit in the definition are to know what exactly is repeated, which is why we "bracket" it with the words begin and end. The other important thing

is that the instructions tell us to stop the execution as soon as an answer is generated. This is indeed crucial!

[Can you see why this is crucial? <sup>3</sup>](#)

This algorithm is an example of a so-called **graph traversal**. This term simply refers to a systematic method of visiting all the nodes in a graph in a particular order. Of course, if we terminate the search as soon as we find the node B, we will not necessarily visit all nodes. Simply omitting the test and proceeding until no more green nodes are left we are, however, guaranteed to visit all nodes that are reachable from A.

[We had earlier discussed some important properties that graphs can have. Which of these can you test using this algorithm? <sup>4</sup>](#)

[How can you extend the algorithm to check whether the graph has cycles? <sup>5</sup>](#)

We can even use the algorithm to compute a spanning tree for the graph as the following video shows:



(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-Spanning-Tree-from-Traversal-Wi-Fi-High.mp4>)

The video has shown that for one particular instance of a graph, the algorithm will find a spanning tree. But is this always the case? We can make the following argument:

In a connected graph, there is a path from every node to every other node. The algorithm visits all nodes to which a path exists from a single, arbitrarily chosen starting node and marks these as green. Thus, in a connected graph it will mark all nodes in green and it will add one edge for each new node that it colours such that it is connected to the green nodes so far. This will generate a connected graph that contains all nodes of the original graph and some (but not all) edges of the original graph. It will never add an edge that forms a cycle to a node already visited. Thus the graph formed by the purple edges must be a tree. As it contains all the nodes of the graph it is a spanning tree.

Alternatively you could look back at the inductive definition of the minimum spanning tree. If you remove the requirement to pick the cheapest edge from that definition, the algorithm follows exactly the construction procedure that the inductive definition prescribes!

A shortcoming of our algorithm is that tagging the nodes with colour attributes would in practice be a very inefficient way to keep track of which nodes we still have to expand.

#### [Why would it be inefficient to use just colour attributes to keep track of visited nodes? <sup>6</sup>](#)

Luckily it is easier to make this more efficient and more elegant at the same time.

However, before we can discuss this, we should really need to prepare some groundwork and figure out how to write algorithms precisely.

---

- ☒<sup>1</sup> The method requires us to go to all the neighbours of a node, then to all the neighbours of these neighbours. We must thus either be able to be at multiple locations simultaneously (unlikely) or we must somehow memorize and sequence these visits.
- ☒<sup>2</sup> If there are cycles or undirected edges, we could run into trouble, because we would arrive at locations that we have already explored before. Unless we take precautions to memorize where we have been already we would thus end up going around in circles.
- ☒<sup>3</sup> If the algorithm would not stop after the answer "yes" is produced, it would eventually always answer "no". What the answer "no" really means in this algorithm is that we have run out of options to check. But this only means that there is no path if we hadn't found one earlier. Thus the validity of the "no" answer depends on stopping as soon as we have found a path so that we will never get to this instruction if there is a path.
- ☒<sup>4</sup> You can use this algorithm to test whether a graph is fully connected: Start at an arbitrary node. If at the end of the algorithm all nodes are red, the graph is fully connected. If there are white nodes left, it isn't. The algorithm will visit all nodes in the connected component of the start node. You can also use the algorithm to test whether the graph has any cycles.
- ☒<sup>5</sup> You only need to check when adding the neighbours of a current node C if one of these is red already. If so, you had processed this node before, so you have discovered a cycle.
- ☒<sup>6</sup> There is no way to directly access a node of a specific colour. Thus to find the next green node (and to find out whether any green nodes are left) the algorithm would have to search all nodes and check the colour for each of them. Because this potentially means that we will have to check each and every node this is not an efficient solution.

## 2.4

# How to write an algorithm

[draft for preview only; BM]

## What is pseudocode and why do we use it?

Computer scientists like to use pseudocode for expressing algorithms. Here's an example of pseudocode:

Algorithm mystery1

  input: a list of numbers L  
  output: a number, but what does it mean?

```
let x := 0
let n be the number of elements in L
repeat until the list L is empty
    remove the first element of L and add it to x
return x/n as the result
```

[Can you guess what this pseudocode does? <sup>1</sup>](#)

Here is an example that is slightly more complex.

Algorithm mystery2

  input: a list of numbers L  
  output: a number, but what does it mean?

```
sort L in ascending order of elements
let n be the number of elements in L
if n is odd then
    return the element in position (n/2+0.5) of L
if n is even then
    let a be the element in position (n/2) of L and
    let b be the element in position (n/2)+1 of L and
    return (a+b)/2 as the result
```

[Can you still find out what the algorithm does? <sup>2</sup>](#)

You probably answered both questions above correctly, which shows that pseudocode does indeed serve a purpose: it lets us communicate with one another, and this is one of the most important considerations when we are writing algorithms. To quote two famous computer scientists and the authors of one of the best-known books on programming:

"...a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."

from: Structure and Interpretation of Computer Programs,  
by Harold Abelson, Gerald Jay Sussman with Julie Sussman

Of course no computer will be able to interpret pseudocode. Which begs the question, why do we not communicate in *real* code straight away? There is a seemingly simple answer to this: when writing real

code we would simply get lost in detail. Consider the [Python](https://www.python.org/) (<https://www.python.org/>) code (or rather one possible form of Python code) for the first example above:

```
def mean(listOfNumbers):
    if len(listOfNumbers) > 0:
        return float(sum(listOfNumbers)) / len(listOfNumbers)
    else:
        return float('nan')
```

Even though this is a very simple piece of code, it is much harder to understand.

[What do you think, for example, does the "return float\(nan\)" do?](#)<sup>3</sup>

What is worse, the following bit of code, almost the same as the one above, is simply incorrect.

```
def mean_incorrect(listOfNumbers):
    if len(listOfNumbers) > 0
        return sum(listOfNumbers) / len(listOfNumbers)
    else:
        return float('nan')
```

[Can you spot the differences?](#)<sup>4</sup>

Assume the second program (with the syntax error fixed) is called as `mean_incomplete([1,2])` to compute the mean of a list with just the two numbers 1 and 2.

[Can you guess what the program would return as a result?](#)<sup>5</sup>

"Why", you might object, "this is a contrived example! Who cares whether there is a colon or a semicolon, I understand the code anyway"... and of course you still know how to read this piece of code even though the computer wouldn't. "So", you might argue, "we don't need to worry about the character at the end of the instruction, we still have almost valid code and can communicate with it". But this is a slippery slope. Real code is at a very well-defined level: A real, existing, precisely defined bit of machinery can execute it. We can also exactly (!) write down what the code means mathematically (this is an advanced field of computer science called "programming language semantics"). However, both the attempt to execute our slightly sloppy bit of "real" Python code on a computer and the mathematically defined semantics would fail to give us a result for this code fragment. Both need the level of rigour that only a completely and rigidly defined programming language gives us. If allow we ourselves to choose another character to terminate a statement, where do we stop? Why not write "As long as" instead of "While"? Why not "Unless  $x=0$ " instead of "If  $x!=0$ "? Once we start to take liberties with the code, where do we stop?

This is exactly why pseudocode exists. There is no defined point where the liberties stop, and there can't be. If we could precisely define which changes to the code are allowed, we would again have a precisely defined language (which defeats the purpose). Thus, in pseudocode we only use a vaguely defined level of language that is based on common sense and the common background of computer scientists. The idea of writing pseudocode is simply to write things down precisely enough so that the intended reader can interpret it without any ambiguity, yet to do so at a sufficiently high level of abstraction that allows us to focus on the salient aspects of the algorithm. There is judgement involved in how precisely we need to specify something, and as all human judgement, this can fail. This is the price we have to pay for using pseudocode, and there is no way around it. Yet, most computer scientists agree that it is worthwhile paying this price, and to write pseudocode as the first step.

To illustrate the point try to find out what the following piece of Python code does:

```
def find(limit):
    limitn = limit+1
    not_selected = [False] * limitn
    selection = []

    for i in range(2, limitn):
        if not_selected[i]:
            continue
```

```

for f in xrange(i*2, limitn, i):
    not_selected[f] = True

selection.append(i)

return selection

```

If you understood this piece of code, it's a reasonably safe assumption that you are not new to programming. If you could not find out consider the following piece of pseudocode instead that describes the same function:

**Input:** an integer  $n > 1$

Let  $A$  be a list of truth values, indexed by integers 2 to  $n$ , initially all set to **true**.

**for**  $i = 2, 3, 4, \dots$ , not exceeding  $\sqrt{n}$  repeat the following  
**if**  $A[i]$  is **true** **then**

**set**  $A[j]$  to **false** **for** all  $j = i^2, i^2+i, i^2+2i, \dots$ , up to  $n$

**Output:** all  $i$  for which  $A[i]$  is **true**.

In identifying the function of this code it may help you to simply execute it yourself for some intermediate number  $n$  (e.g.  $n=20$ ) and analyse the result.

[Can you now guess what the function of the code is?](#)<sup>6</sup>

We will also often use pseudocode in this book. However, since readers will have varying backgrounds, this can be dangerous territory - what has an obvious meaning for one person may be unclear to another. Therefore, we will often make use of Edgy code instead of pseudocode. Edgy, as you may know, is a special educational programming language that is very close to pseudocode. Consider the simple example of finding the largest element in a list of numbers. In pseudocode we might write the solution like this:

*maximum of list(list):*

*if (length (list) = 1)*

*return first item of list*

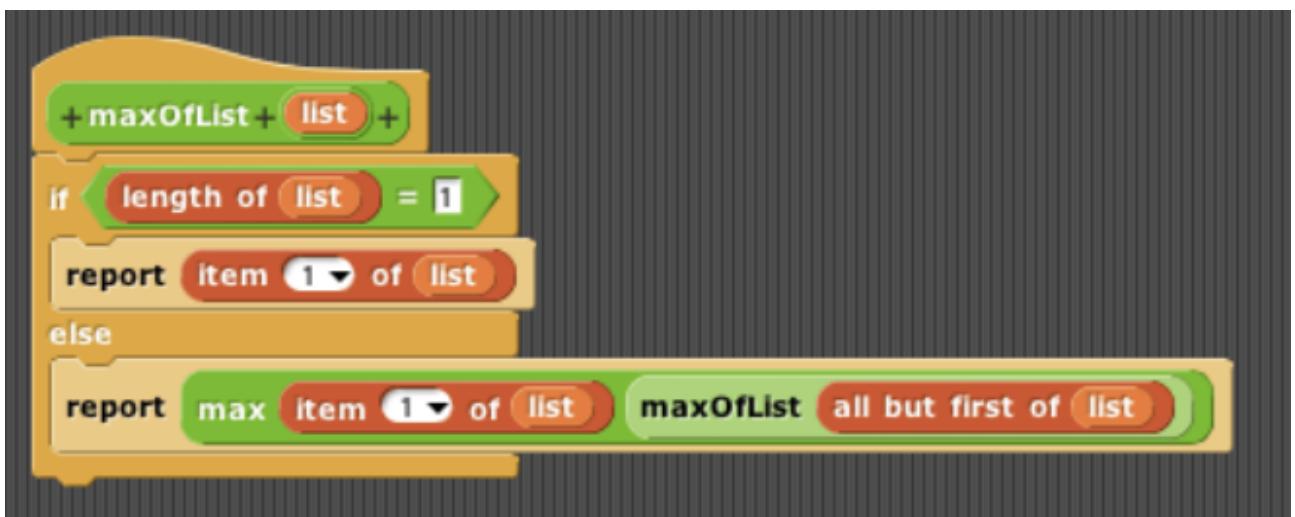
*else*

*return the greater of*  
*first item of list*  
*and*

*maximum of list (rest of list)*

*end .*

In Edgy it looks almost the same:



Despite its similarity to pseudocode, Edgy is a real programming language. This has a number of clear advantages: First, the code is executable, so that you can experiment with it directly. Secondly, the level of code is entirely well-defined. There is no question about what we are allowed to write down and what not. However, being real code, it also has the usual disadvantages: Sometimes we do not have the flexibility to express some new concept at a very high level (this is when we will use pseudocode) and even though Edgy uses a high level of abstraction, we can still get bogged down in details. However, using Edgy (or another high level educational language) is a valid middle ground, as you will see. Three factors make it possible to reach this middle ground:

1. The visual syntax frees us from the tyranny of having to memorise syntactic details, such as where a semi-colon belongs. The essential rule is "if it snaps it is syntactically correct".
2. Much of the functionality of an industrial strength programming environment is stripped from Edgy, to let us focus on algorithm design. You will not be able to build a user interface or to implement fast numerical methods. Yet, everything that we need for the discussion of network and graph algorithms is there, and this reduction to the essentials lets us focus on algorithm design.
3. Some functionality that you will need but for which there is no point now in worrying about the how it is done in detail, is provided in the form of pre-defined libraries and so-called "abstract data types". We will come back to this point (modularisation, abstraction, and abstract data types) in detail in a later module.

It is worth noting that the transition to coding in an industrial-strength programming language will be straightforward once you have understood how to code things in Edgy. Look at the example above and mentally strip it of the colours and block shapes. What is left is structurally close to the corresponding Python code...

```
def maxOfList(list):
    if len(list)==1:
        return list[0]
    else:
        return max(list[0], maxOfList(list[1:]))
```

For now, let us take a rough and tumble tour of the basic elements of pseudocode. This should enable you to understand the basics very quickly.

To do so we will incrementally built up an algorithm for the traversals we have developed informally above.

<sup>1</sup> It computes the mean (i.e. the numerical average) of the elements in list. To do so it first sums up the elements in the list one by one and then divides the result by the numbers of elements.

<sup>2</sup> This example computes the median of the list which is defined as the middle element of all elements in the list. If the

number of elements is even there is no well-defined middle element. In this case the usual convention is to take the average of the two elements closest to the middle.

- ☒<sup>3</sup> This line is only executed when the list is empty, i.e. when the mean cannot be computed. "float(nan)" generates an object that can be used like a number but indicates that in truth it is "Not A Number" (NAN). This is exactly an entirely straightforward concept! This is used to return the message that no result could be computed.
- ☒<sup>4</sup> The first difference is that there is a colon missing at the end of line 2. A python compiler (or interpreter) would simply report an error. It would not understand the program and would refuse to even run it. Such a mistake is called a syntax error - we did not use the syntax defined for Python correctly and therefore the program cannot be run. But there is also another difference: in line 3 I omitted to indicate "float" in the division. This causes a problem that represent a semantic error, an error in the meaning of the program. The program will run just fine, but it will sometimes return a result that is not what we expect.
- ☒<sup>5</sup> Strangely it would not return the correct result 1.5 but instead 1. This is because Python thinks we are only working with whole numbers (integers) and essentially decides for that reason to round down the result. If we indicate that we mean to work with float and call the program with `mean_incomplete([1.0,2.0])` everything works fine! Likewise, everything works fine in the first version of the code because we indicated in line 3 using the word float that we want to work with real values (so-called floats). Oh the joys of coding!
- ☒<sup>6</sup> The code computes all prime numbers up the integer n. It is a version of the so-called "Sieve of Erastosthenes" that you probably now already. If not you find a detailed description on Wikipedia. The pseudocode above was adapted from this Wikipedia page.

## 2.4.1

# Sequences

### [draft for preview only; BM]

The fundamental unit of an algorithm are *statements* or *instructions* that are put into sequence. Consider how we visit the next node in the network:

```
Algorithm next-node1
    find a green node
    mark this node red
    mark all its neighbours green unless they are red already
```

Each line in this algorithm is a single statement. Statements are put into sequence by which we indicate that they have to be executed strictly in the order in which they are written down.

What is a statement is not strictly fixed in pseudocode. In principle any instruction may be used that is *well-defined* and can be executed in a *finite amount of time*. While pseudocode gives us the freedom to invent new statements or statement types as we need them, doing so only results in a meaningful algorithm if these "made-up" statements can be interpreted and executed *unambiguously and without guessing* on the basis of reading the algorithm alone and of agreed upon (!) prior knowledge.

---

## 2.4.2

# Variables and Assignments

### [draft for preview only; BM]

The first aspect of the above algorithm description that we need to revise is that we have referred to some entity by using the pronouns "this" or "it". This worked fine above because there was clearly only one object to refer to, the node that we are marking red. In general, though, if there are multiple objects this is not a clean way to reference an object. In pseudocode we use *variables* to refer to specific entities. You can think of variables as placeholders for objects and values. Alternatively you can think of them as names for containers that contain such an object or value. Wherever we write the name for the container (the variable name) we actually mean the object that is currently in the container. There are two fundamental operations on variables:

1. We can *assign* an object or a value to a variable ("put an object into the container") . This is usually written as " $x := a$ " meaning that the variable  $x$  is assigned the value  $a$ . Before the first assignment to it, a variable holds no value (i.e. the first use of a variable must always be an assignment).
2. We can *use* the object or the value that the variable stands for ("that is in the container") by using the name of the variable instead.

Let's use this to clean up the code above:

```
Algorithm next-node2
  n := any green node
  mark n red
  mark each neighbour of n green unless it is red already
```

Much better already!

---

## 2.4.3 Decisions

### [draft for preview only; BM]

The second aspect to refine is the instruction "unless it is green already". This statement describes a decision and we need a well-defined and structured way to express decision - otherwise it will be impossible to express complex decisions unambiguously.

For example, consider the statement "mark x unless it is round unless it is blue". Does this mean that all objects will be marked except for the round blue ones? Or that all objects will be marked except for the round ones and the blue ones? Or will the round blue ones be marked but none of the other blue ones? How many possible interpretations can you think of? How do these differ? It is plain to see that we need a well-defined way to express decisions. Pseudo-code usually uses "if-then-else" statements for this. Such a statement is written as:

```
if condition then statement A else statement B
```

It means that first the condition will be tested. If the condition is true, statement A will be executed, and if it is false statement B will be executed. The else branch may be dropped completely if it is not required. But we quickly encounter a difficulty if we try to use an "if-then-else" to rewrite the code above:

```
Algorithm next-node3
    n := any green node
    mark n red
    let m be a neighbour of n
    if m is not red then mark m green
```

How do we proceed from here? We need to repeat the same process for each and every neighbour of m. This can be made precise using the concept of iteration.

---

## 2.4.4

# Repetition and Iteration

### [draft for preview only; BM]

Pseudocode usually allows a range of different forms of iteration. The cleanest and simplest one is the "while" statement. It takes the form

```
while condition do statement
```

and indicates that the statement must be repeated for as long as the condition remains true. First the condition must be tested and if it is true the statement must be executed. After this, the test is repeated and if it is true the statement is repeated. This continues until the test is false. Thus, the condition must be something that changes through the execution of the statement... otherwise this will process will never stop. Using a while statement we can revise the code further and write

```
Algorithm next-node4
  n := any green node
  mark n red
  while (n has a neighbour that is not red) do
    mark green a neighbour of n that is not red
```

The effect of this while statement is that the non-red neighbours of n will be marked green until all have been marked. At this point the condition, which is evaluated each time before the statement is executed again will be false and the repetition will be terminated.

The code above is correct, but the way we have used "a neighbour that is not red" twice clearly lacks elegance. We will shortly improve on this...

## Other forms of repetition and iteration

It would really be completely sufficient to have a while loop. With this we can express everything we need. However, simply for convenience we often use other types of iteration as well. A useful variant of the while loop is a repeat-until loop. The only difference between this and the while loop is that the termination condition of a while loop is checked before the loop is entered and each time the body of the loop is executed before this is done, whereas in a repeat-until loop this test happens at the end of the loop (i.e. after the body is executed).

Look at the following two versions of a loop that counts

```
x := 1
while (x<10) do
  begin
    print x
    x := x+1
  end
```

and

```
x := 1
repeat
  begin
    print x
    x := x+1
```

```

    end
until x=10

```

### [Will the two loops print the same numbers?](#)<sup>1</sup>

Armed with repeat loops you may now be tempted to rewrite the version of next-node 4 given previously in this form. *But this is not correct!*

```

Algorithm next-node4
  n := any green node
  mark n red
  repeat
    mark green a neighbour of n that is not red
    until n has no more neighbours that are green or red

```

### [Why is this not correct?](#)<sup>2</sup>

A nice alternative shorthand form of loop is the "foreach" statement. This is particularly well-suited for the situation at hand. This type of iteration takes the form

```
foreach x in choices do statement
```

where choices is the set of all items that have to be handled by statement. This will be executed by first assigning the first item in choices to x and performing the statement with this value for x, then assigning the next value in choices to x and performing the statement again. This process is repeated until all items have been handled. We could write:

```

Algorithm next-node5
  n := any green node
  mark n red
  foreach (x in neighbours of n) do
    if x is not red then mark x green

```

This is starting to look better!

The final loop type that we need to consider is the "for" loop. Like the "foreach" loop, it is not really required, because the while loop alone can already express all iterations, but it is sometimes convenient nevertheless. For loops are generally used for iterations that need to be executed for an increasing or decreasing regular sequence of numbers. The prototypical case are indexes. A for loop takes the form:

```
for x from a to b step n do statement
```

where x is a variable name and a, b, n are numbers (or variables that have numerical values). The loop will bind x to the value a and then execute the statement. Subsequently, it will increment x by n and test whether this value is less than or equal to b. If this is the case, the statement will be executed again with the new value for x. The process repeats until x is greater than b. If n=1 we may drop the "step n" from the statement. A simple example is:

```
for x from 1 to 10 step 2 do print x
```

which will print out the odd numbers from 1 to 9. After the loop has finished, x has the value 11. Imagine we had indexes for the nodes in our graph. The statement:

```
for i from 1 to (number of nodes in the graph) do
  colour node with index i green
```

would colour all nodes in the graph green, whereas:

```
for i from 1 to (number of nodes in the graph) step 2 do
```

colour node with index  $i$  green  
would colour only every other node.

---

- ➊ No! The while loop will print 1...9 (because the loop won't be re-entered when  $x=10$ ) while the repeat loop will print 1...10 (because the termination test for  $x=10$  only takes place after the body has been executed).
- ➋ Because the test is only executed at the end of the loop, there is no guarantee that  $n$  even has a neighbour when the loop is entered.

## 2.4.5 Blocks

### [draft for preview only; BM]

Above we have glossed over an important question: How do we know which statements are dependent on a condition (in the case of a decision) or which statements are repeated (in the case of an iteration)? Our pseudocode needs to indicate this somehow. For this, we need the concept of a "block".

A block "groups" several statements so that they are treated as a single statement. The pseudocode indicates a block by bracketing the statements in some way. Commonly a "begin" is used to indicate the start of a block and an "end" to indicate where it finishes. Sometimes braces are used. Sometimes only indentation is used, and all statements that are at the same level of indentation are considered as belonging to the same block (this is also done in the programming language Python). We will use "begin"- "end". This may be more verbose, but it is also very clear. Let's illustrate this with the simple example of the for loop above:

```
for x from 1 to 10 step 2 do
    print x
print "more"
```

will print "1 3 5 9 more". The numbers 1...9 are printed by the loop, as discussed above. When the loop finishes the next statement after the loop is executed, which prints "end". If we would bracket both print statements with "begin" and "end" they would form a single block and thus be treated as a single statement.

```
for x from 1 to 10 do
    begin
        print x
        print "more"
    end
```

### Can you figure out what this would print? <sup>1</sup>

So far our little algorithm "next node" has only considered the nodes around a given start node n. We do, of course, want to traverse the whole graph! Recall that this simply means that, once we have marked all neighbours, we need to restart with one of the nodes that are still green as a new start node. In other words, we simply repeat the execution of our algorithm next-node 5 as soon as it has finished. For that, we do of course use iteration. Since we want to repeat the whole sequence of instructions that makes up algorithm next-node 5, we need to group these into a block by using begin and end. The only remaining question is when to terminate the repetition. But this is easy: we simply terminate when there are no more nodes to be processed, i.e. when no more green nodes are left.

```
Algorithm next-node-6-incomplete
    while (there are green nodes in the graph)
        begin
            n := any green node
            mark n red
            foreach (x in neighbours of n) do
                if x is not red then mark x green
        end
```

But there is a trap: In the beginning not a single node is green. So when the execution tries to enter the loop for the first time the loop condition is false. In consequence, the loop will never be executed at all! To fix this, we need to mark a "seed" node before the loop is entered so that the loop condition is true.

```
Algorithm next-node-6
```

---

```
marks an arbitrary node green
while (there are green nodes in the graph)
begin
    n := any green node
    mark n red
    foreach (x in neighbours of n) do
        if x is not red then mark x green
end
```

---

- <sup>1</sup> This piece of code would repeat both print instructions 10 times. It would thus print "1 more 2 more 3 more" ... and so forth until "10 more".

## 2.4.6 Nesting

### [draft for preview only; BM]

Note that we have implicitly used a new concept in version 6 of our algorithm. This is called "nesting". We have nested a loop inside another loop. We are allowed to do this because the whole loop is simply a type of statement and statements are exactly what a loop executes repeatedly.

The situation in our algorithm is more complex than that: above we are not just repeating a loop but a whole sequence of instructions including a loop. We can do this because we have wrapped the entire sequence into a "block" using the "begin" and "end" brackets. This turns everything in the block into a single (compound) statement so that the block can be put anywhere where a statement can be put.

There is another case where we have already applied nesting above: We have nested an if-statement inside a foreach-loop. The same thinking applies: the whole if instruction is indeed a single statement and thus the foreach loop can repeat it.

In the same way other types of statements can be nested. For example, we can nest an "if" statement inside another if statement:

```
if (x>0) then
    if (x<5) then
        print x
```

[Can you figure out what this fragment of code will do? <sup>1</sup>](#)

---

<sup>1</sup> It will print the value of the variable x if and only if this value is between 0 and 5 (but excluding the values 0 and 5). The nesting of the two if statements effectively causes a conjunction (AND) of the conditions to be applied for the print statement. Thus, the code has the same effect as if  $(x>0 \text{ AND } x<5)$  then print x

## 2.4.7

# Abstraction and Modularization

### [draft for preview only; BM]

Meanwhile we are dealing with quite a few lines in our algorithm and even with several levels of nesting. The concepts we have introduced so far are powerful - so powerful that we would not need anything else to write our algorithms. However, this could quickly become unwieldy. We need a better way to structure our algorithms.

Recall how above we have wrapped the code of one one algorithm (next-node5) into a loop to obtain the algorithm next-node6. Why did we have to copy all this text? Instead we could write:

```
Algorithm next-node-6
    marks an arbitrary node green
    while (there are green nodes in the graph)
        next-node-5
```

thus calling upon another algorithm when we define a new one. This is called *modularisation*. The beauty of modularisation is that it allows us to write things in a far more compact form. In particular if we need the same functionality more than once in an algorithm. We can simply encapsulate this part of the algorithm into a separate algorithm and use it twice.

---

## 2.4.7.1 Parameters

### [draft for preview only; BM]

Now let us consider a related but different kind of traversal - we want to execute two of the traversals we have performed so far in parallel starting at two specified nodes A, B. Thus one wave of green nodes will start to spread from A and one wave will start to spread from B. Since colour is the only property that we are currently using to distinguish nodes, we will have to use two different colours for the two waves, let us say green and yellow.

Our first attempt to write the algorithm might look like this

```
Algorithm next-node-6
    mark node A green
    mark node B yellow
    while (there are green or yellow nodes in the graph)
        begin
            if (there is a green node in the graph)
                begin
                    n := any green node
                    mark n red
                    foreach (x in neighbours of n) do
                        if x is not red then mark x green
                end
            if (there is a yellow node in the graph)
                begin
                    n := any yellow node
                    mark n red
                    foreach (x in neighbours of n) do
                        if x is not red then mark x yellow
                end
        end
    end
```

Note that we have essentially just written down a loop that executes two copies of algorithm next-node-5. However, we can not simply call next-node-5 twice because we need to execute it with different colours. If we had a mechanism to pass the information to the algorithm which colour to use, we could simply execute it twice but with different colours. This is what *parameter passing* is for. An algorithm can have any number of parameters that are supplied to it as input when the algorithm is called. When we specify the algorithm we write the parameters in brackets behind the algorithm name, and we use variable names for these. The convention is that these variables will be bound to the parameter values that are supplied when the algorithm is called by another algorithm. A version of next-node-5 with a colour parameter would thus look like this:

```
Algorithm next-node5-p(c)
    if (there is a node with colour c in the graph) then
        begin
            n := any node with colour c
            mark n red
            foreach (x in neighbours of n) do
                if x is not red then mark x with colour c
        end
```

"c" is the single parameter of this algorithm. The name that we give to the parameter has no function. It is simply a variable name. At the time that we execute the algorithm c will be bound to whatever value we

have supplied. We could also write a version with more than one parameter. Let us say, we also wanted to specify the colour of the completed nodes. We would then write

```
Algorithm next-node5-p2(c1, c2)
  if (there is a node with colour c1 in the graph) then
    begin
      n := any node with colour c1
      mark n with colour c2
      foreach (x in neighbours of n) do
        if x is not of colour c2 then mark x with colour c1
    end
```

When we call an algorithm we supply the values for the parameters by listing them in brackets behind the name of the called algorithm. The convention to identify which parameter is which is simply that the order of the parameter values in the call must be the same as that of the parameters in the definition of the called algorithm.

Now that we can tell the algorithm which colour to use we can write our double-traversal significantly more elegantly.

```
Algorithm next-node-7
  mark node A green
  mark node B yellow
  while (there are green or yellow nodes in the graph)
    begin
      execute next-node5-p2(red, green)
      execute next-node5-p2(blue, yellow)
    end
```

This algorithm will spread two wavefronts through the graph. From A we spread a green wavefront that leaves red nodes in its wake and from B we spread a yellow wavefront that leaves blue nodes in its wake. We have written down our algorithm much more compactly and elegantly in this form and any change that we make to the base algorithm next-node5-p2 will automatically be used consistently in both places. But the true beauty of writing modular algorithms is that it allows us a top down development approach. We don't need to think about how to do every little detail straight away. If we know that some functionality can be achieved but we do not want to bother with the details, we can simply assume that an algorithm (yet to be defined) exists for it and use this as a module in higher-level abstractions. This allows us to postpone worrying about the details of how to exactly perform this function and to stay mentally focussed on the level of abstraction that we are currently trying to tackle.

---

## 2.4.7.2

# Return Values

### [draft for preview only; BM]

Let us illustrate this top-down development. First observe that if the two waves that our last algorithm propagates through the graph ever meet we have identified that a path from A to B exists. Let us extend our algorithm to verify whether there is such a path in the graph. Surely we can check whether the two waves have met, but we don't want to worry about the details at this point. We want to modify the algorithm along the following lines:

```
Algorithm next-node-8-incomplete
    mark node A green
    mark node B yellow
    while (there are green or yellow nodes in the graph)
        begin
            execute next-node5-p2(red, green)
            execute next-node5-p2(blue, yellow)
            if "the two waves have met" then
                print "A path has been found"
        end
    print "no path has been found"
```

Clearly, we need to specify how to check whether "the two waves have met" with a new algorithm. There is only one problem: So far we have no way for the new algorithm to signal the outcome of the test to the if statement. To do this we need to use one further concept: *return values*. We allow our algorithms to use a "return" statement that defines the result of the algorithm. This result essentially replaces the call of the algorithm in the code. You can think of it like a read-only variable. Using this we can write

```
Algorithm next-node-8
    mark node A green
    mark node B yellow
    while (there are green or yellow nodes in the graph)
        begin
            execute next-node5-p2(red, green)
            collision := test-collision(green, yellow)
            if collision(green, yellow) then print "A path has been found"
            collision := test-collision(green, yellow)
            if collision then print "A path has been found"
        end
    print "no path has been found"
```

The effect of this is that the assignment will call the algorithm test-collision (which we yet have to define) with the parameters green and yellow.

What does the algorithm test-collision need to do? A simple way to perform the test would be:

```
Algorithm test-collision(c1, c2)
    if (there is a pair of green and yellow nodes
        that are adjacent) then
        return true
    else return false
```

Let us write this in a little more detail to make explicitly how finding the pair would have to happen. Essentially we need to check all possible pairs of nodes and this can be done by using two nested loops each of which iterates over all possible nodes:

---

```

Algorithm test-collision(c1, c2)
    foreach n1 in all-the-nodes
        foreach n2 in all-the-nodes
            if (n1 is of colour c1)
                and (n2 is of colour c2)
                and (there is an edge from c1 to c2) then
                    return true
    return false

```

There are a number of subtleties here how both the algorithm as well as the way of writing it down could be improved, but it would be too early to discuss them here. You may however want to think about the following: You could save your algorithm a lot of work by integrating the collision test directly in next-node-5-p2.

[How would you do this and how would it save work?](#)<sup>1</sup>

---

<sup>1</sup> If you integrated the propagation to the next node and the test tightly instead of modularising them in the form shown above, the test could profit from knowing what the last node expanded was. It would not have to search for a pair of adjacent yellow and green nodes but instead would only need to test whether the just expanded node touches the waveform of the opposite colour. This test could be done in fewer steps (because not all pairs of nodes would have to be checked). Striking the right balance between clean modularisation and efficiency is not always easy, but we will postpone this discussion until after we have developed to discuss efficiency in a meaningful way.

## 2.4.8 Collections

### [draft for preview only; BM]

Let us return to our basic traversal algorithm (next-node6). It works fine, but there is an important difference from the two traversals that we discussed earlier. Recall that breadth-first traversal (BFS) had an important property: it visits the nodes in increasing distance from the start node and in this way it always finds a node on the shortest possible path (provided short is defined as the number of edges traversed). The algorithm above does not do this: it visits the nodes in some random order. This is because we have used the statement "n := any node with colour" to find the next point to expand from. What we need is to maintain a specific order in which to process the nodes: we want to do this on a first-in-first-out basis, i.e. if node a was marked green earlier than another node b then a should also be expanded earlier than b.

A simple way to keep track of this is by memorising a sequence of nodes adding each node to the end of this sequence as it is marked green and always taking the next node from the front of this sequence. This sequence will then be served in a first-in-first-out manner.

The only trouble is that so far we have no way of memorising such sequence. This is what *lists* are for. A list is an object that can hold any number of other objects in a defined order. Variables can be bound to lists just like to any other value or object. We write lists by including the elements in square brackets and separating them by commas. Thus

```
x = [4, 3, 2, 1]
```

binds x to the list that contains the values four to one in decreasing order. It is worth noting that a list can be empty! This is written as

```
y = []
```

There are only a few fundamental operations necessary to work with lists: you need operations to

1. check whether a list contains any elements at all or is empty. We will do this by writing "is empty(x)" which evaluates to the corresponding truth value.
2. get the first element of a list. We will do this by writing "first(x)". For example "a=first(x)". Note that you cannot get the first element from an empty list.
3. remove the first element from a list. We will do this by writing "rest(x)". Note that this does not actually mean that we are changing x. We are simply getting a list that contains all elements of x bar the first. If we want to change x itself, we would have to write an explicit assignment "x := rest(x)"
4. add a new element to the list. Strictly speaking, you only need a single operation `prepend(l, a)` which adds an element a at the front of the list l. However, for convenience sake we will also agree on a second operation `append(l, a)` which adds an element a to the end of the list l.

For convenience you will, sometimes want to have different kinds of access to a list. For example, you may want to look at the *n*-th element in sequence directly. Many programming languages will offer you operations to do this and many other things with lists, but note that these are just convenience operations. If you have the above four you can do everything already, but you may need a bit more writing. To illustrate the point let us write an algorithm that returns the *n*-th element in a list. It will have the list and the index position as parameters

```
Algorithm element(l, n)
  temp-list := l
  while (n>1) do
    temp-list := rest(temp-list)
  return first(temp-list)
```

Other convenience operations could be described in similar ways and it is a good exercise to think of a few convenience operations on lists and to specify them using only the four operations given above.

---

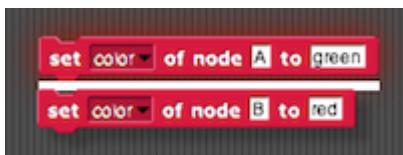
## 2.4.9

# Converting Pseudocode into Edgy

In this module, we're going to convert the pseudocode algorithms that we discussed in the previous modules into code in Edgy. You will see how to snap together blocks in Edgy just like the sequences of statements that make up an algorithm.

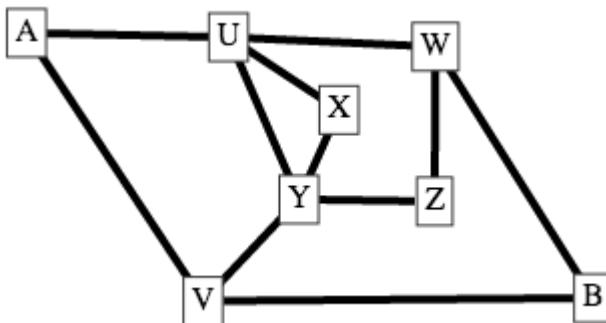
### Statements in Sequence

Programs in Edgy consist of blocks, which are the statements or instructions for your program. The blocks are grouped into categories listed in the top left section of the Edgy window (the palette). To select a category, simply click on the category name and the blocks will be listed in the panel underneath. To build a program, select a block and drag it into the centre of the Edgy window (the canvas). Blocks can be "snapped" together in sequence to make a program. The shape of the blocks indicates how they fit together, like the pieces in a jigsaw puzzle. When you drag blocks together in Edgy, a white line will appear indicating how the blocks fit together.



### Variables and Assignment

We're going to build the traversal algorithm that we discussed in modules 3.3.1 - 3.3.8 in Edgy. The file below called "example\_graph", contains an example network similar to the one that we used in module 3.2.



Please take a moment to load it into your own copy of Edgy so that you can follow along with this module and build algorithms in Edgy. First, download the file to your computer, then open the Edgy window, right-click on the stage and select "import from file" to load this file into Edgy.

[example graph](https://www.alexandriarepository.org/wp-content/uploads/20150331094302/3.3.9.txt) (<https://www.alexandriarepository.org/wp-content/uploads/20150331094302/3.3.9.txt>)

Now, let's create a variable in Edgy. Try this in your own copy of Edgy as we go along. Click on "Make a Variable" under the variables tab, then give your variable a name, "n".

**Make a variable**

This will create a variable block named "n", that will be listed in under the variables tab. You can use the checkbox next to the variable block to show or hide the variable on the stage in Edgy.



Now, let's set the value of our variable "n". You can set or update the value of a variable in Edgy, using a "set ... to ..." block. Select the name of the variable you want to set from the pre-populated drop-down list of variables and type in a value for that variable:- let's give our variable "n" the value "A".



To use the variable, drag the orange block under the variables tab, named "n", inside other blocks. You can use a variable block anywhere where you can type in a value.



Now drag and snap your blocks together to make the following block of code, and click on this block to run it on the example graph and see the node named "A" turn green.



## Decisions

In Edgy, we can use the "if... else..." block to make decisions in our code.



If the "else" branch is not required, an "if ..." block can be used. Once again, let's try this out on our example graph in Edgy. Build the block below and then try it out before and after changing the colour of the node X to red to see it execute. (You can change the colour of a node manually, by right-clicking on it on the stage and selecting "set color").



## Repetition and Iteration

We can create a "while.. do..." statement using the "repeat until ..." block in Edgy. Whereas the "while condition do action" statement repeatedly performs an action until some condition no longer holds, the "repeat (action) until condition" statement performs an action until some condition is met. Thus, the condition of a while...do..." statement is negation of the condition of an equivalent "repeat until...." statement.

So, the following pseudocode using a "while ... do ..." loop to add 6 nodes to an empty graph....

```
while (number of nodes is not equal to 6) do
    add node (new node)
```

... can be written equivalently in Edgy using a "repeat until ..." loop as follows:-



Edgy also has a "foreach item of ..." block, which iterates over a list of possible values and executes the blocks inside once for each item in the list. Please try this out on our example graph, using the following block to colour all the neighbours of one node. You can rename your loop variable (named "item" by default), by clicking on it and entering the new name in the "Script variable name" dialog box.



When you ran the blocks above on your example graph, it changed the colour of some of the nodes in the graph. To change the colour of the nodes back to white in your example graph, you can use another "for each item in ..." loop to do this, like the one below.



There is also a "for i = 1 to 10" block in Edgy, which can be used for iterating over an sequence of numbers. The numbers change by 1 each iteration, and can be set to wither increment or decrement (by setting the start number as either higher or lower than the end number). For example:-



If you wanted to alter the amount to increment the number between each iteration (the "step"). You could write this this in Edgy as follows:-



## Blocks

Whereas in pseudocode we need to use use "begin" - "end" (or another strategy such as indentation or braces) to indicate which statements should be grouped together to form a block, in Edgy this is indicated by the shape of the blocks. As you have seen, conditional or iterative blocks in Edgy are shaped like a "C" and wrap around the block or blocks that are dependent on them.

So, where the following program in Edgy, with the "say end for 1 secs" block outside the loop, prints (or "says") the numbers 1 through 10 and then "end"...

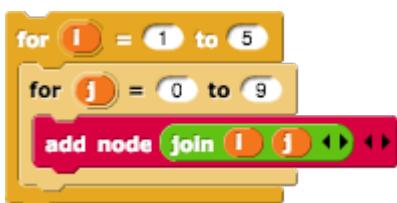


... the program below, with the "say" blocks inside the loop, prints "1", "more", "2", "more", "3", "more"..... "10", "more".



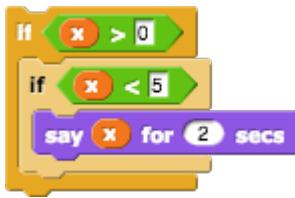
## Nesting

You can also nest C-shaped blocks inside one another in Edgy just as you would other blocks. So you can nest loops inside loops:-



If the name of your nested loop variables are duplicated, you should re-name one or both of them to avoid confusion.

Or you can nest an "if..." block inside another "if..." block:-



Or, an "if.." block inside a loop:-



Build the block above in your own Edgy environment and try it out on the example graph above. All of the nodes in the graph will turn green.

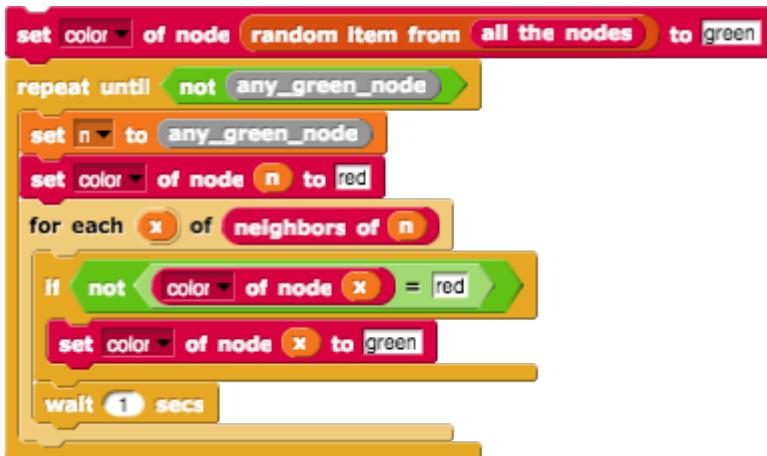
## Abstraction and Modularisation

Now, we've seen enough blocks in Edgy, to start to build the "next-node" algorithm, which we created in pseudocode in modules 3.3.1 - 3.3.5. Before you read further, please have a go at converting the pseudocode for next-node-6 (below) into blocks in Edgy.

```

Algorithm next-node-6
  marks an arbitrary node green
  while (there are green nodes in the graph)
    begin
      n := any green node
      mark n red
      foreach (x in neighbours of n) do
        if x is not red then mark x green
    end
  
```

Here's the "next-node'6" algorithm converted into Edgy:-

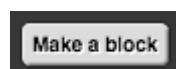


However, you may have noticed the grey block called "any\_green\_node": where did this block come from? Well, our pseudocode includes statements to check whether "there are green nodes in the graph" and to

find "any green node". And, checking whether there are green nodes in the graph is the same as checking whether we can find "any green node". So, to build our "next-node-6" algorithm, we need a block to find a green node in our graph, if one exists. As no such block exists in Edgy, we're going to build our own block or "module" to find a green node if there is one. Let's see how we can do this:

## Making a Block in Edgy

To create a module (or block) in Edgy, click on the button labelled "Make a block" at the bottom of the Variables palette.



This will bring up a dialog like the one below for you to select the type of block that you want to create and which palette you would like it to be listed under in the Edgy environment. We'll type in a name for our block, "any\_green\_node"...



... and then choose a block type. There are 3 block types:

- Command blocks: A command block is a block that performs an action or actions. They are shaped like bricks or jigsaw puzzle pieces that can be slotted together.



- Reporter blocks: A reporter block will report a value that can be used as an input to another block. They are an oval shape.



If you drag a reporter block onto the canvas by itself and click on it, the value that it reports will

appear in a speech bubble next to it.



- Predicate blocks: A predicate is a special kind of reporter that always reports either *true* or *false*. Predicate blocks are a hexagonal shape:

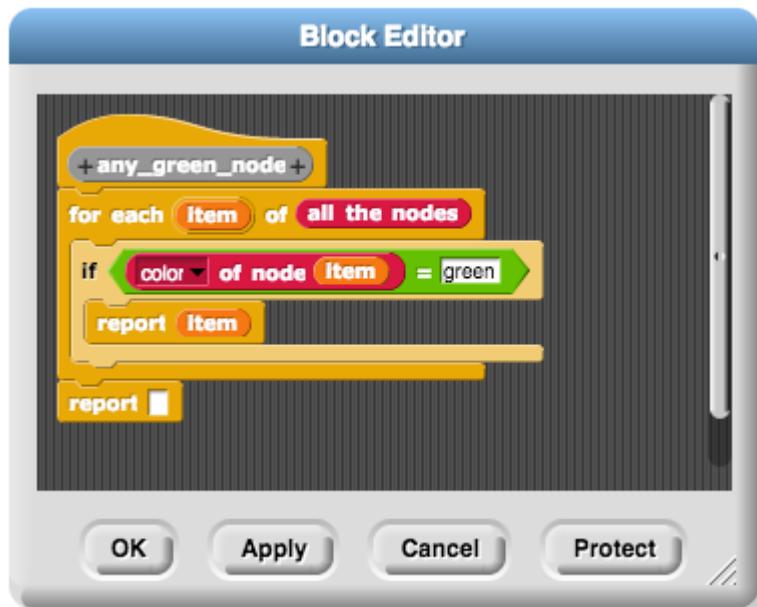


They fit inside special hexagonal input slots like this one:



We're going to make the "any\_green\_node" block a reporter, because it needs to report the name of a green node if there is one in the graph. Once you have selected a category and type for your block, type in a name for it and hit "Enter". This will bring up the "Block Editor" window with a template for your new block or "module" where you can drag and snap blocks together to define your block or module.

Here's the "any\_green\_node" block below. It loops through all the nodes in the graph, checking whether they are green. As soon as it finds a green node, it will report the name of this node. If it has looped through all the nodes and not found a green node, it will report back without a node name.

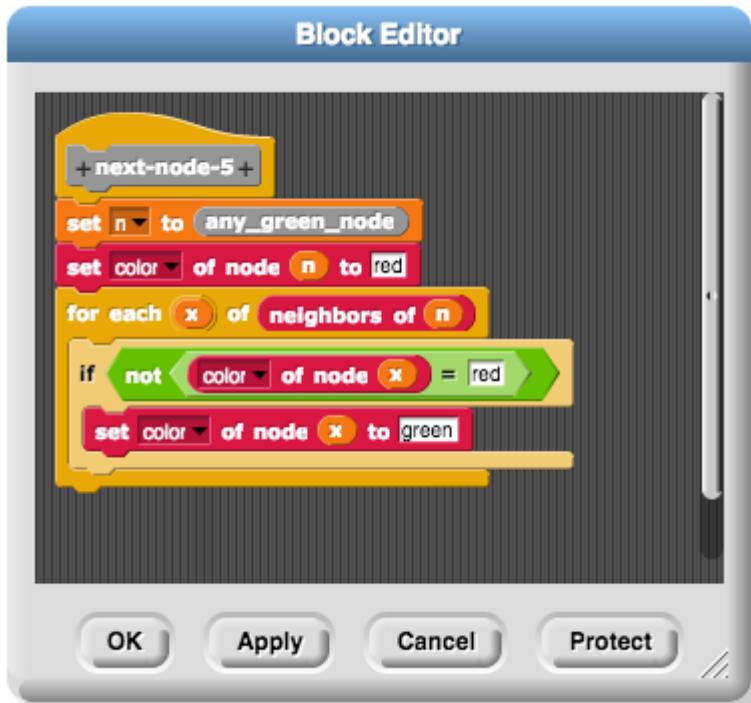


Once you are happy with your block in the Block Editor, click on "OK" to create it. Your block will then appear at the bottom of the palette that you selected. Grey blocks, categorised as "Other", will appear at the bottom of the "Variables" tab. You can modify your block by right-clicking on it and selecting "edit..." .

Once you've created your block, you can use it just like you would use any other block in Edgy.

In module 3.3.7 we discussed how to modularise our algorithm "next-node-6" by using the algorithm "next-node-5" from module 3.3.4. We can do this in Edgy by making another block for "next-node-5", and using this block in our next-node-6 algorithm.

Let's make a new command block in Edgy for "next-node-5":-



And now we can use this block in "next-node-6" to create a modularised version of the same algorithm.



## Parameters

In Edgy, we can also add input parameters to blocks that we have made. As in module 3.3.7.1, we're going to modify the "next-node-6" algorithm to execute two traversals, one starting from node A and one from node B. A wave of green nodes will spread from node A, and a wave of yellow from node B. Here's the pseudocode that we created in module 3.3.7.1 as a reminder of our algorithm.

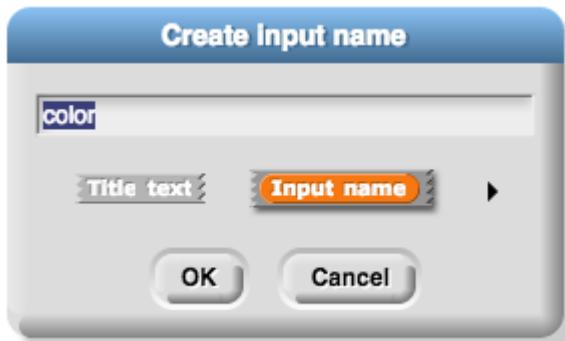
```

Algorithm next-node-7
  mark node A green
  mark node B yellow
  while (there are green or yellow nodes in the graph)
    begin
      execute next-node5-p2(red, green)
      execute next-node5-p2(blue, yellow)
    end
  
```

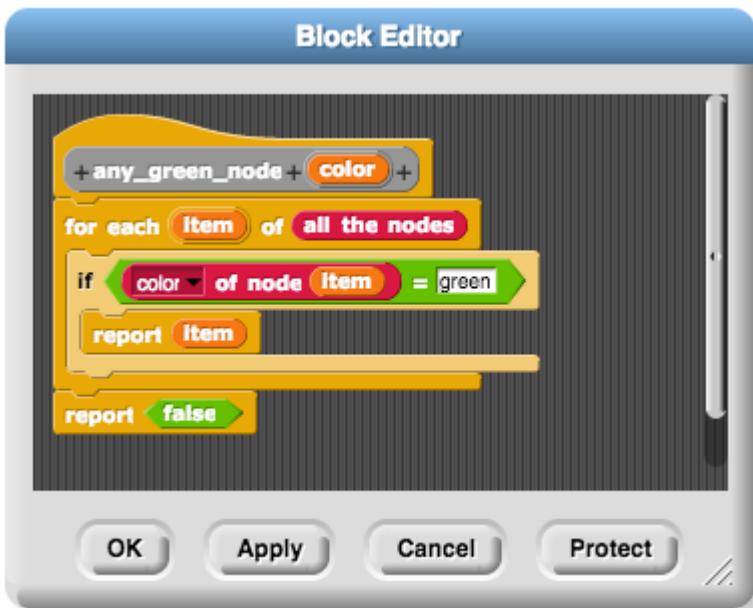
When we start converting this into code in Edgy, we'll need to add some parameters to the blocks (or modules) that we previously created in Edgy: "any\_green\_node" and "next\_node\_5". The parameters will

allow us to specify which different node colours for the two wavefronts.

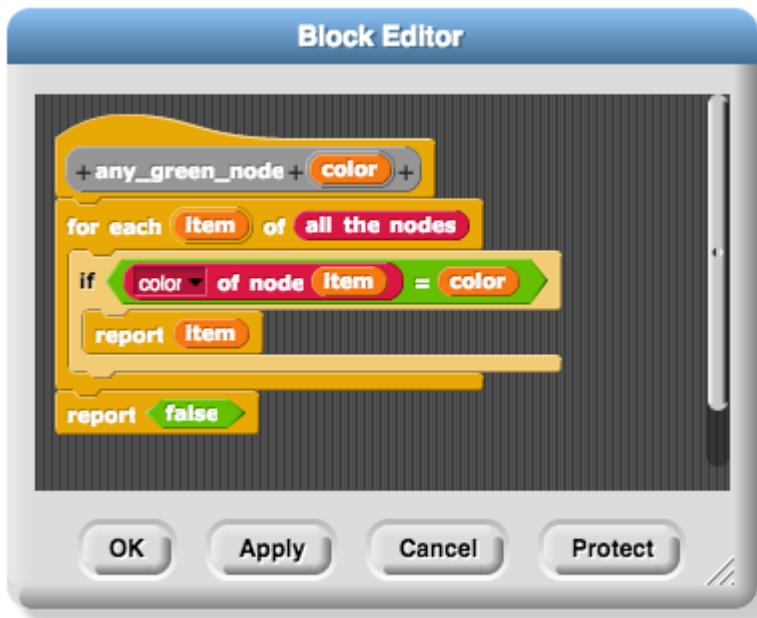
First, let's edit the "any\_green\_node" block, to test for the presence of a node colored with a color that we specify. Right click on the "any\_green\_node" block and select "edit..." to add a parameter for color to this block. Then, in the Block Editor, click on the plus sign (+) next to the name of the block, which will bring up a "Create Input name" dialog. Using this dialog, you can edit the name of your block (by selecting "Title text"), or add input parameters to your block (by selecting "Input name"). We're going to add an input parameter for color as follows.



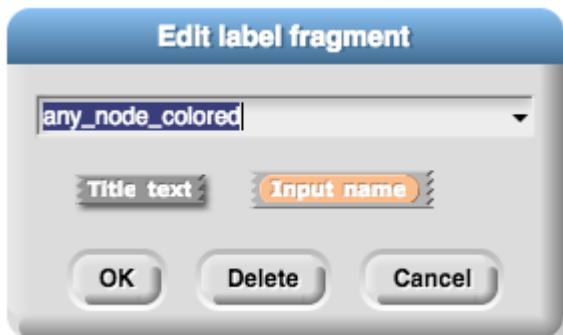
And here's our modified "any\_green\_node" block.



Now, we'll need to replace the text string "green" with the "color" variable that we created as an input parameter. To do this simply drag and drop the color parameter into the spaces where the text "green" was.



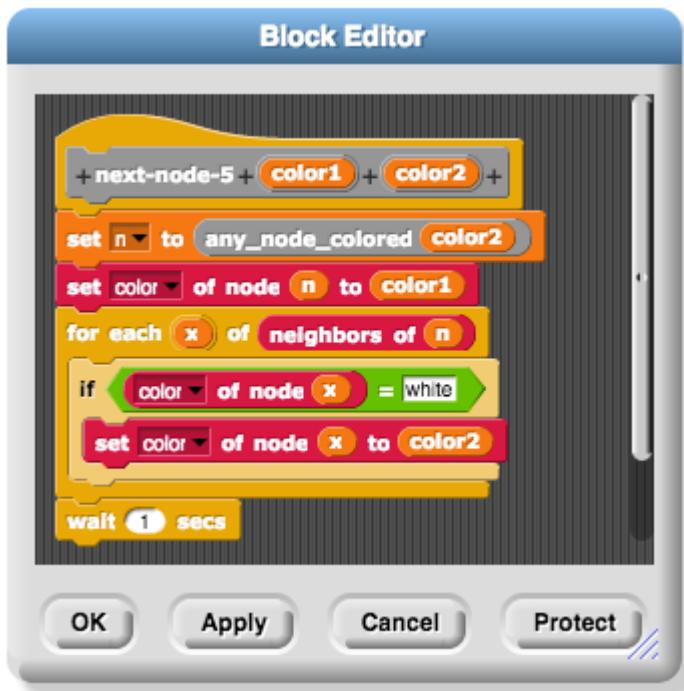
You'll notice that the name of the block no longer makes sense now that it takes a colour as input, so we're going to change the name of the block as well. To do this, click on the name of the block in the "Block editor" dialog, and this will bring up a dialog called "Edit label fragment". Select "Title text", then enter a new name for the block ("any\_node\_colored"), and click "OK".



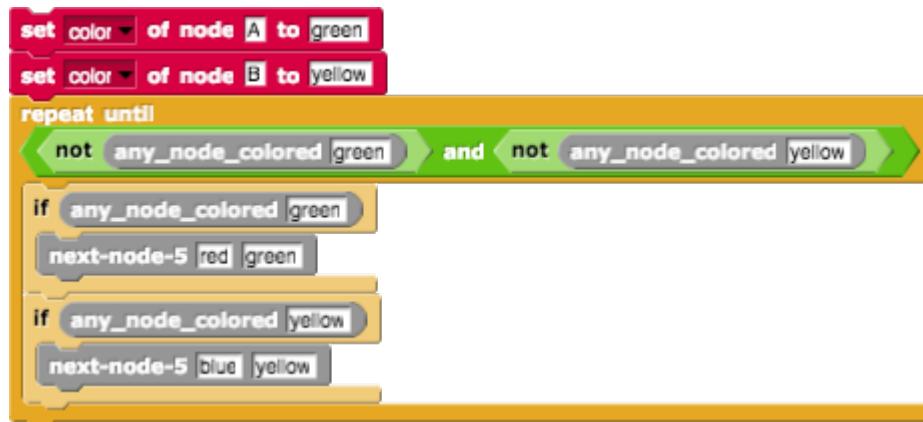
Now we have an "any\_node\_coloured" block that checks whether there are any nodes of a specified color in the graph. We can use this in our "next-node-7" algorithm.

However, we'll also need to add some parameters to our "next-node-5" block, which will use two colors to color the nodes it has visited and next nodes to visit.

Once again, right click on the "next-node-5" block and select "edit...". Then click on the plus sign to add the necessary parameters to the block (one at a time), we'll call them color1 and color2. Then replace the text "red" and "green" with color1 and color2 respectively. (Note: I have changed the if condition to check if the node is white, in order not to colour over yellow / blue nodes from the other waveform).



And now we can build the "next-node-7" algorithm as Edgy using the parameterised blocks that we've created. The calls to "next-node-5" are inside individual "if..." conditions, to check whether only green or only yellow nodes exist in the graph (not both).



## Return values

When we created our "any\_green\_node" block, we made it report (or return) the value of a green node. Reporter and predicate blocks in Edgy return a value to the calling block.

Now, if you run the the "next-node-7" algorithm above, you will notice that it colors the nodes from the 2 wavefronts. If we want to verify whether the two wavefronts have met, we need to add a test for collisions, as we discussed in the "next-node-8" algorithm in module 3.3.7.2. Here's the pseudocode for "next-node-8". Before you read on, have a go at converting this into Edgy yourself.

```

Algorithm next-node-8
  mark node A green
  mark node B yellow
  while (there are green or yellow nodes in the graph)
    begin

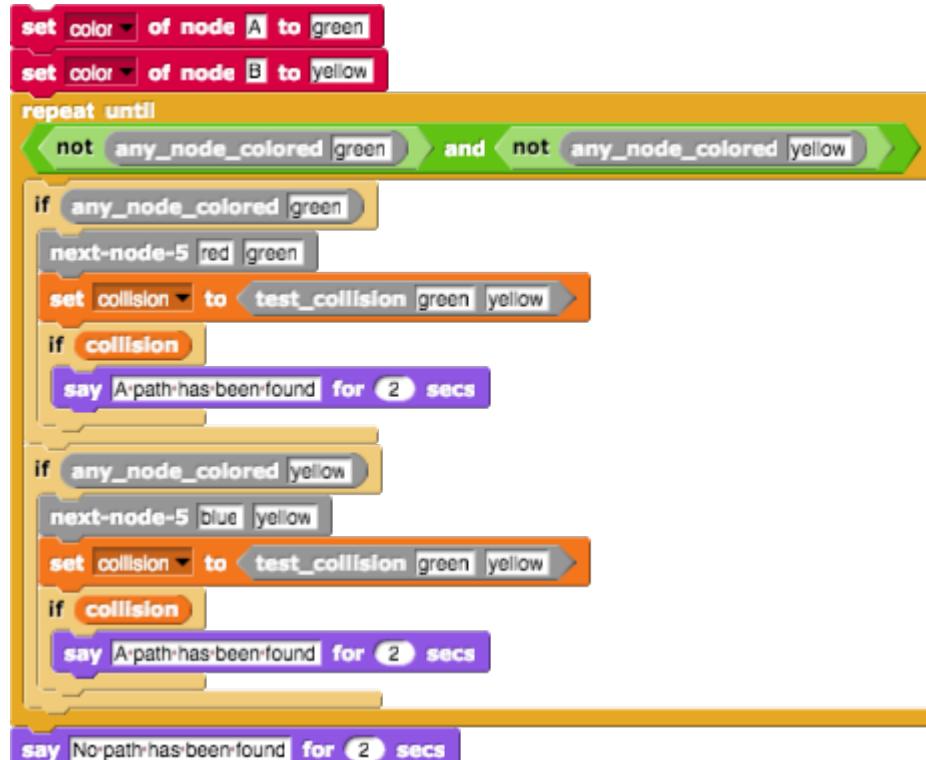
```

```

execute next-node5-p2(red, green)
collision := test-collision(green, yellow)
if collision(green, yellow) then print "A path has been found"
collision := test-collision(green, yellow)
if collision then print "A path has been found"
end
print "no path has been found"

```

And here's the corresponding Edgy code:-

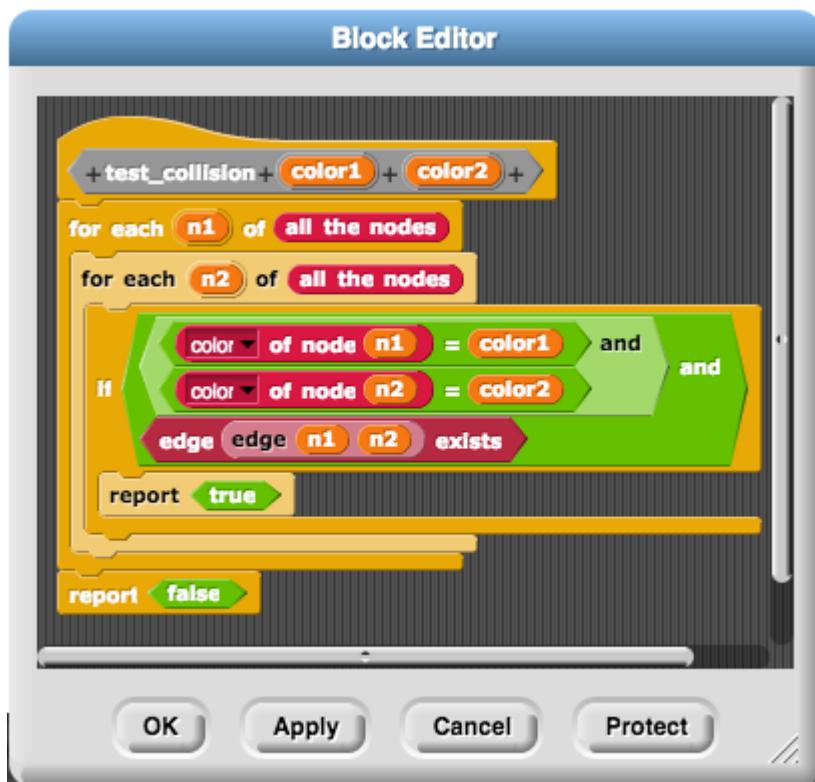


We'll also need to create a predicate block for "test-collision", as follows, which returns either true or false, depending on whether the two wavefronts have met.

```

Algorithm test-collision(c1, c2)
  foreach n1 in all-the-nodes
    foreach n2 in all-the-nodes
      if (n1 is of colour c1)
        and (n2 is of colour c2)
        and (there is an edge from c1 to c2) then
          return true
  return false

```



## Collections

To create a list in Edgy, use the "list ..." block. A lists with no input spaces is an empty list. You can use the black arrows to increase or decrease the number of elements in the list.

**list 1 3 7 13 ↪** or **list ↪**

You can bind a variable to a list in Edgy just as you do with any other value:-

**set x ↴ to list 1 3 7 13 ↪**

List blocks and blocks to perform various list operations are found under the "Variables" tab in Edgy. The fundamental list operations can be performed as follows:-

1. Checking if a list is empty
 


or

2. Getting the first element from a list:
 


1



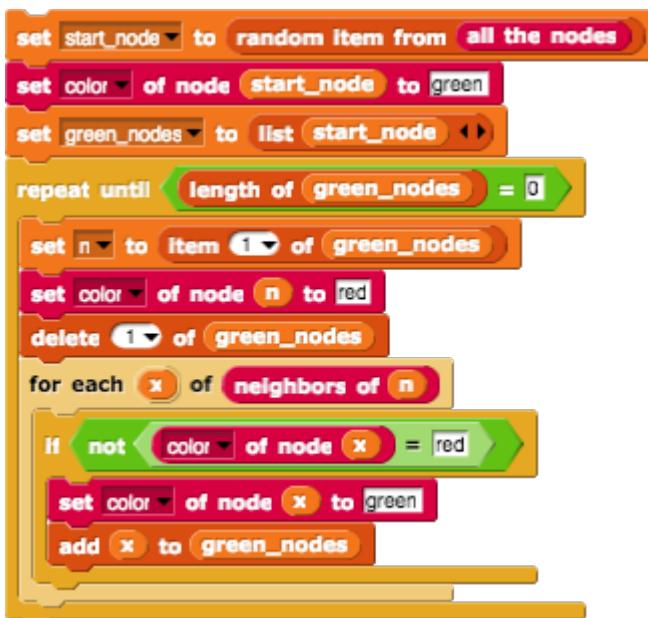
3. Getting the rest (all but the first element) of the list:-
4. Adding an element to the end of a list, use the "add thing to ..." block:-



And the resulting list:-



As discussed in module 3.3.8, we're going to modify our traversal algorithm (next-node-6) to ensure that it visits the nodes in order of increasing distance from the start node. To do this, we're going to store the nodes that we have marked green in a list. The nodes towards the front of the list, will have been marked green before those at the end of the list. Here is the algorithm in Edgy:-



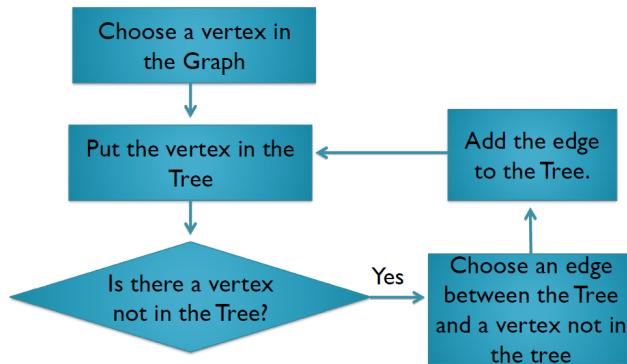
Notice that we are using a list "green\_nodes" to keep track of which nodes are coloured green. Nodes are appended to this list as they are coloured green, and removed from the front of the list as they are coloured red. And the green nodes are visited in list order: in each iteration, we take the first green node from the list as our next node.



## 2.5

# Edgy: Revisiting the Muddy City

Recall Prim's algorithm from our earlier discussion of the Muddy City Problem.



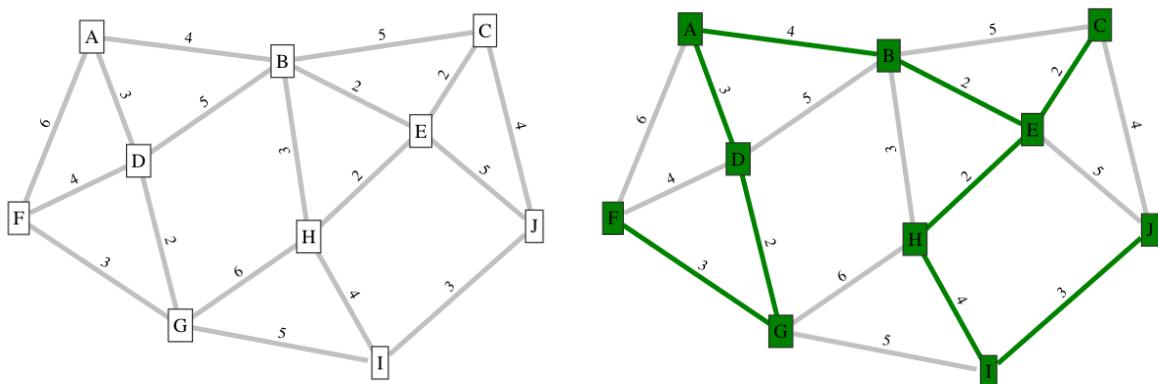
Remember that we refined this so that we chose the closest node each time through the loop. We also need to add a branch for the case when there are no remaining nodes (the "no" case missing from the conditional in the flowchart).

Let's begin by writing down a sequence of steps in plain English.

```

initialize a tree with a single node, chosen arbitrarily
while there is still a node not incorporated into the tree
    find the closest node not already in the tree
    add an edge to link this node to the tree
  
```

Now let's flesh it out more, and write proper pseudocode. We will assume that the cost of an edge is stored as the label of the edge. We will add two attributes to each node, one to hold the cost of adding that node to the graph, and one to keep track of which node in the current MST is closest. We will mark MST nodes and edges in green as we go.



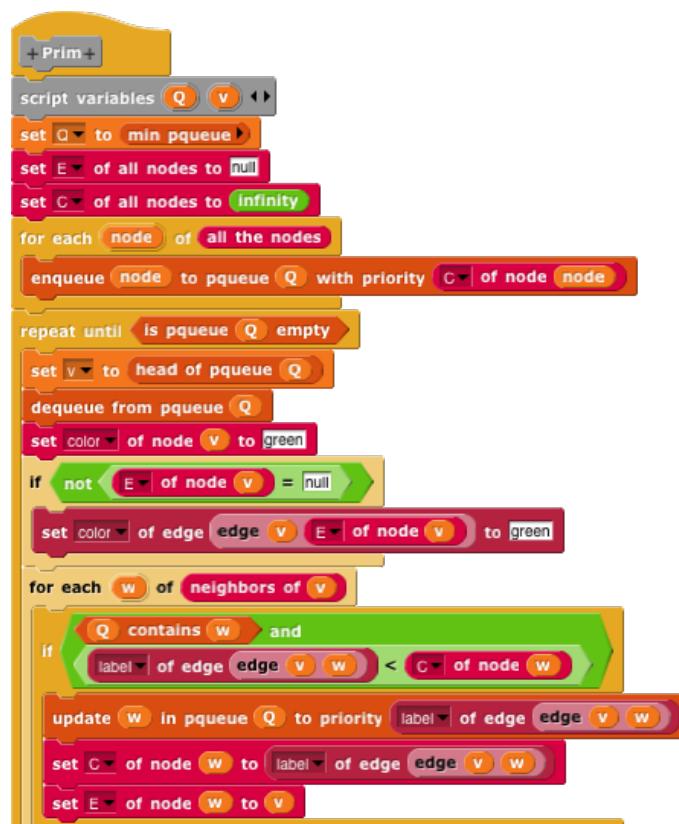
The following is an adaption of the pseudocode for [Prim's algorithm from Wikipedia](https://en.wikipedia.org/wiki/Prim%27s_algorithm) ([https://en.wikipedia.org/wiki/Prim%27s\\_algorithm](https://en.wikipedia.org/wiki/Prim%27s_algorithm)). For ease of comparison we use the same variable names. We will use the notation label(a, b) to refer to the label of the edge (a, b).

```

Q := new priority queue
E := new dictionary
C := new dictionary
foreach node n:
    E[n] := null
    C[n] := infinity
foreach node n:
    enqueue n in Q with priority C[n]
repeat until Q is empty:
    v := head of Q
    dequeue from Q
    set colour of v to green
    if E[v] ≠ null:
        set colour of edge (v, E[v]) to green
    foreach neighbour w of v:
        if Q contains w and label(v, w) < C[w]:
            update w in Q to have priority (label(v, w))
            C[w] := label(v, w)
            E[w] := v

```

NB we represent E and C using node attributes. This is just a matter of convenience, and we could have used dictionaries.



Prim's Algorithm in Edgy



## 2.6

# Breadth First Search and Depth First Search (2016)

### [draft for preview only; BM]

We can now return to the search algorithm that we had discussed earlier and phrase it in a precise way. We will make it precise in two ways: firstly, by using the algorithmic pseudo-code notation that we settled on earlier, second by using lists instead of marking the nodes with colours. This brings the algorithm very close to an actual implementation as a program, as you will soon discover.

As a first step, let us rewrite the algorithm - still using colour marking - in proper pseudocode. This is really just a minor rewrite.

```
Algorithm search(A, B)
/* input: two nodes A, B
/* output: true if there is a path from A to B, false otherwise

mark A green
found := false
while (there is a green node)
begin
    C := an arbitrarily chosen green node
    if (C=B) then found := true
    else
        foreach n in neighbours(C) do
            if (n is not marked red) then mark n green
        remove the green mark from C
        mark C red
end
return found
```

Note that this algorithm will continue to traverse the graph even after it has arrived at node B. For what we are trying to achieve this is a complete waste. We can stop the algorithm from doing so by incorporating explicitly in the condition of the while loop that it should only continue if B has not yet been found:

```
Algorithm search(A, B)
/* input: two nodes A, B
/* output: true if there is a path from A to B, false otherwise

mark A green
found := false
while (there is a green node) and not found
begin
    C := an arbitrarily chosen green node
    if (C=B) then found := true
    else
        foreach n in neighbours(C) do
            if (n is not marked red) then mark n green
```

```

        remove the green mark from C
        mark C red
    end
return found

```

Lists give us an easy and efficient way to keep the nodes that we still need to expand directly accessible (the green nodes). Using lists, we can immediately access the next green node without any search. (We assume you are familiar with the basic concept of a list, if not, please study [the corresponding module](#) before you proceed).

We will use one list to keep track of all the green nodes and one list to keep track of the red nodes that we have visited already. The "green" list will initially only contain the start node. In every repetition of the loop we take one node off the list (the one being marked red) and add its neighbours to the list (the new green ones). Thus, this corresponds exactly to what we have done with the colours before.

We will use the list and the graph ADTs that we have defined earlier. One consequence of this is that we will need the whole graph as input to the algorithm, because the *neighbours* operation needs it as an argument. Of course, this makes perfect sense. Without knowing the graph we cannot determine the neighbours of any node! We have just been sloppy about this so far and assumed implicitly that there is a specific graph in the context of which we evaluate our operations.

```

Algorithm BFS(G, A, B)
/* input: a graph G and two nodes A, B of G
/* output: true if there is a path from A to B in G, false otherwise

Green := new_list()
Green := append(A, Green)

Red := new_List()
found := false

while not(is_empty(Green) or found)
begin
    C := first(Green)
    if (C=B) then found := true
    else
        foreach n in neighbours(G, C) do
            if not(is_member(n, Red)) then Green := append(n, Green)
        Green := rest(Green) /* remove the green mark from C
        Red := append(C, Red) /* mark C red
    end
return found

```

Now that there is no arbitrary choice of next node in the algorithm anymore the visit order is clearly defined . We are using a list, take the nodes to be processed from the front, and insert new nodes to be processed at the end.

The resulting algorithm is called **Breadth First Search** or **BFS** for short. The name of the algorithm is explained by the order in which the nodes are visited.

[What can you observe about the order in which BFS visits the nodes? What does this mean for the path to the target node B?](#)<sup>1</sup>

The following video gives an example of the visit order that you can use to check your understanding.



(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-BFS-visit-order-Wi-Fi-High.mp4>)

### [Is the visit order uniquely defined? Can you say exactly in which order the nodes will be visited?](#)<sup>2</sup>

The way in which we have used the list is a very special (and important) type of lists, which is called a *Queue*. It behaves exactly as a (well-behaved) waiting queue should: newcomers queue up at the end, and the people at the front of the queue are served first. An alternative name for a queue is thus also *FIFO*, which stands for first-in-first-out.

Now consider one tiny change to the algorithm: We add new elements at the front of the list rather than at the end. Before we proceed we need to define one more detail. In BFS we avoided adding nodes to the list twice by checking whether we had visited them already. Let's make a different choice this time: we always add the nodes to the front of the list (even if they are already in the list) risking duplicates. Instead we avoid revisiting nodes by checking them as we pull them off the list.

```

Algorithm DFS(G, A, B)
/* input: a graph G and two nodes A, B of G
/* output: true if there is a path from A to B in G, false otherwise

Green := new_list()
Green := append(A, L)

Red := new_List()
found := false

while not(is_empty(Green) or found)
begin
    repeat

```

```

C := first(Green)
Green := rest(Green)
until not(is_member(C, Red)) or is_empty(Green)
if not(is_member(C, Red)) /* did we actually find a next node to work
with?
begin
  if (C=B) then found := true
  else
    foreach n in neighbours(G, C) do
      if not(is_member(n, Red)) then Green := cons(n, Green)
    Red := append(C, Red) /* mark C red
  end
end
return found

```



(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-DFS-visit-order-Wi-Fi-High1.mp4>)

Assume we would not have crossed out the edge between the nodes numbered 2 and 8. Would the algorithm ever expand the node numbered 7 in this case?<sup>3</sup>

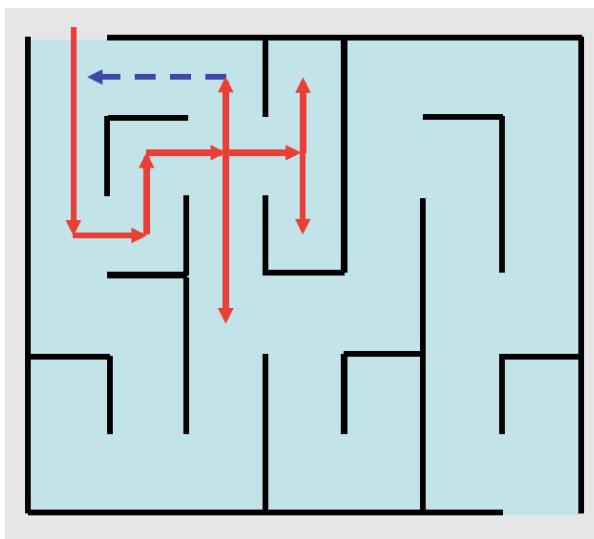
The way in which we have used the list this time is another special type of lists, which is called a *Stack*. It behaves like a stack of dishes that is waiting to be washed: new dishes are stacked on top (at the front), and dishes are taken from the top (front) to be washed. If dirty dishes come flooding in faster than they are cleaned, the first one will never be cleaned (a common share-house phenomenon)!

An alternative name for a stack is *LIFO*, which stands for last-in-first-out.

The name of this algorithm is **Depth First Search**, or **DFS** for short. It is another example of a traversal method. The name derives from the fact that it first follows a path in full depth (until the path ends or until the target node B is found). If the path ends, the algorithm backs up to the previous node in the path and expands another neighbour instead. This is exactly the method you should use to find your path out of a maze (but only if your memory is good

enough to keep track of all the places you have visited already). You trail a thread behind yourself and every time you get stuck you back up to the last point where you had a choice to explore where another neighbour takes you to. You do this until you find the exit or run out of options.

Use the picture of the maze below to find your way from the entry to the exit using DFS. Recall that a maze can be represented as a graph by turning each branching point as well as entry and exit into nodes, and creating edges for each two such points that are directly connected by a corridor. Note that you could not effectively use BFS if you were physically caught in a maze. This would require you to keep the list of unexpanded "green" nodes explicitly and to jump between them as you are finding your path. This jumping would really make your problem worse instead of helping to solve it.



You probably found it hard to keep track of what you had already explored at a previous node. This is exactly because it is difficult to keep a stack in your head! But it is very easy for a computer.

There is a much better way to phrase DFS than the above. We can formulate DFS very concisely using a programming concept called recursion. We will revisit DFS therefore after we have studied recursion in a later module. This will also help us to define powerful extensions of DFS. (For those of you already familiar with recursive programming: the magic of the recursive DFS is that we don't need to keep the stack explicitly, because the call stack handling the recursive calls takes care of all of this automatically in the background!)

The final problem that we need to solve to make our search for the path from A to B useful is to actually return the actual path, i.e. the sequence of nodes we have to take to get from A to B.

[How would you extend BFS to explicitly yield the path from A to B? <sup>4</sup>](#)

<sup>4</sup>

The nodes are visited in increasing order of topological distance from the start node. You can imagine the graph as being divided into levels (or layers): Layer 0 contains the start node; Layer 1 all nodes that can be directly reached from the start node; Layer 2 all the nodes which are at distance two from the start node (i.e. for which you must pass through at least one other node to reach them), and so on. With this layering, breadth first search will first visit all nodes in Layer 1,

then all nodes in Layer 2, and so on.

- ◻<sup>2</sup> No! The nodes will always be visited in increasing topological distance from the start node A, but for nodes at the same distance it is not specified which one will be visited first. This is because the order of neighbours(G, C) is not specified.
- ◻<sup>3</sup> This question cannot be answered without further information. It depends on the order in which the algorithm handles the neighbours of node 2 when it expands this. In the case that it prepends node 4 to L first, and only then node 3, node 3 (the target B) will be pulled off the list in the next step and the algorithm will stop. However, if node 3 is prepended first and only then node 4 is prepended, the algorithm will continue the path with node 3 and will eventually expand node 7. Thus, we have found one spot where the algorithm is not (yet) fully and precisely specified.
- ◻<sup>4</sup> This is relatively easy to do, as BFS always visits a node using the shortest path to it. You simply create a new attribute for each node. Call this "parent". As you expand a node A, you set the attribute "parent" of all of its neighbours to A. You will never have to change this, as the node won't ever be reconsidered. Then, when you have found the node B, you can read off the "parent" of B. Call this C. You then read off the parent of C and so on until you have reached A. This gives you the path from A to B in reverse order. Try it out manually on a piece of paper! For the so-called "iterative" version of DFS that we have written, it is not quite as easy to see how we would keep the path. However, this will be exceedingly clear to see in the more elegant recursive version that we will look at later when we introduce recursion. Stay tuned!

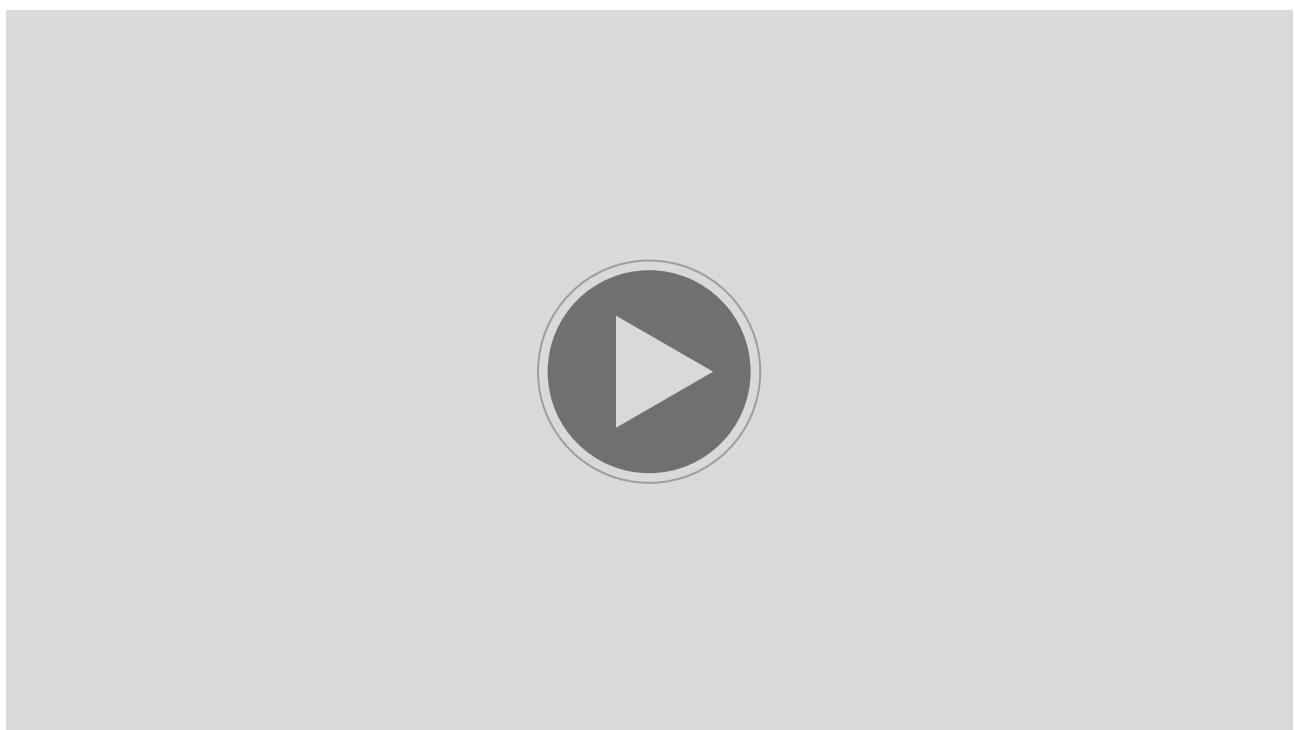
## **2.6.1**

# **Browsing a social network using BFS**

**[draft for preview only; BM]**

We can now use our knowledge of BFS to implement a mini "friends-browser" for the [last.fm social network](http://www.last.fm/social-network) (<http://www.last.fm/>).

The following screencast walks you through the complete implementation of BFS and this browser in Edgy (30 min).



(<https://www.alexandriarepository.org/wp-content/uploads/Last-FM-BFS-Browser-mp4.mp4>)

---

## **2.7**

# **Abstract Data Types**

---

## 2.7.1

# Why ADTs Matter

### Cakes and Parties

By now you are familiar with the idea of an algorithm as a kind of recipe, a sequence of steps for completing a task. We can think of the instructions for baking a cake as pseudocode that can be interpreted by a human with the support of appropriate hardware (the kitchen).

Popping up a level, we can think of higher level tasks in the same way. Suppose you were organising a birthday party for a friend. This could involve steps like:

1. pick a venue and date
2. make a list of invitees and invite them
3. bake a cake
4. put up some decorations
5. host the party
6. clean up the mess

Notice that "bake a cake" is a single step here. However, we can expand "bake a cake" into a more detailed series of steps. The same is true for the other steps above. Imagine a more comprehensive algorithm for a party, in which each of the above steps is broken down into several steps. This would include steps like "identify possible dates for a party", or "invite Kim", or "buy cocoa". In such a way it would be easy to come up a list of 30-40 low-level steps. However, this becomes impossible to manage. Try writing these low-level steps and see just how unwieldy it is.

### Modularity and Top-Down Design

Consider again the short list of tasks for organising a birthday party. Notice how *modular* it is. We didn't give the low level details. At this high level of abstraction, we simply presumed that we would know how to carry out the steps when the time came. For instance, at the point of writing down the algorithm, we anticipated the need to clean up, but perhaps we didn't bother thinking it through in detail. Thus, we set about organising the party in a *top-down* fashion.

The same idea is widespread in programming. For example, if you were writing a program to process meteorological data to find Australia's windiest city, you might write something like the following:

```
wind_data = load_data("wind.csv")
city = analyse(wind_data)
print_result(city)
```

Then you could zero in on the individual steps. If you were working in a team of three, you might assign each person a different step. Everyone would go away and implement his or her module, and then come back to put them together. You would just need to agree on the interface between the parts (i.e. what kind of structure is `wind_data` or `city`).

This modularity also helps later on when it comes to testing. If we are searching for a bug, we can more

quickly narrow our search to a particular module. This can lead to huge time savings.

## Going further with modularity

Let's think about graphs again. Remember that a graph consists of a set of nodes and a set of edges. There are various basic operations that we can perform on a graph, such as adding an edge. Each operation is a separate little algorithm. When we want to perform higher-level tasks such as finding the shortest path between a pair of nodes, we can write a larger algorithm that is built from these basic operations. It's the same situation that we had above with the algorithm for a party that required an algorithm for baking a cake.

This idea is so important that we will spend the next several modules exploring it.

In the case of graphs, we will *encapsulate* graph data (nodes and edges) and graph operations (connectivity, add\_edge, etc) into a single entity, namely the Graph Abstract Data Type, or more simply, the *Graph ADT*. The same will go for other kinds of data and operations, e.g. stacks, with push and pop.

## What do we mean by abstract?

Why are we calling these *abstract* data types? Simply because we are providing a definition without an implementation. We are specifying enough about the type that we can be sure any implementation is correct.

---

## 2.7.2

# The Graph ADT

In general, we are going to formalise data types in three steps: (a) specifying a set of objects; (b) specifying the syntax of the operations that can be applied to this set; (c) specifying the meaning (or semantics) of the operations in terms of the items in this set.

So, how do we specify a graph? At one level, we do it using a node set and an edge set, e.g.  $(\{1, 2, 3\}, \{(1,2), (2,3)\})$ . But this just specifies an *instance* of a graph. How do we specify the abstract data type of graph?

For starters, we need to specify the possible objects under consideration. A graph is a collection of nodes and edges. In other words, it is a pair  $(N, E)$  where  $N$  is a set, and where  $E$  is a set of ordered pairs of items from  $N$ . (We're going to assume that we have a directed graph for now.) According to this definition, here are some instances of graphs:

1.  $(\{\}, \{\})$
2.  $(\{1\}, \{\})$
3.  $(\{1, 2\}, \{(1,2)\})$
4.  $(\{1, 2, 3\}, \{(1,2), (2,3)\})$
5.  $(\{1, 2, 3\}, \{(1,2), (2,3), (3,1)\})$

There's obviously a large space of possible graphs!

### Defining some graph operations

Notice the similarity between the above graphs. We get from the first to the second by adding node 1; we get from the fourth to the fifth by adding edge (3,1). So our Graph ADT needs to provide the operations of `add_node` and `add_edge`, i.e.

`add_node: Graph × node → Graph`  
`add_edge: Graph × edge → Graph`

### What is the meaning of the Cartesian product notation $A \times B$ ?<sup>1</sup>

These operations of `add_node` and `add_edge` enable us to build up all the graphs listed above. However, they do not provide us a way to get started! How do we create a graph in the first place? We need a special operation just for that. It doesn't take any arguments at all, but it gives us a graph:

`new_graph: → Graph`

Now we can provide the operations that would have produced the earlier sequence of graphs. Note that, in order to perform an operation that involves no arguments, we still provide parentheses, i.e. `new_graph()`. Also, in order to apply an operation to an object, we use a dot, as you can see in line 2.

1. `G = new_graph()`
2. `G.add_node(1)`

*Can you provide the rest?*

Finally, we should give ourselves a way to access the contents of the graph we have been creating:

`all_the_nodes: Graph → set(node)`  
`all_the_edges: Graph → set(edge)`

These operations are useful if we want to process all nodes or all edges in some way. Thus, if we perform the operations to create the above series of graphs, then do `all_the_nodes(G)`, we would get the set {1, 2, 3}.

Each of these lines is called a *signature specification*. A signature consists of the name of an operation, the type(s) of its input(s) and the type of its output.

Although the details are beyond the scope of the course, we can already provide some axioms to ensure that graphs behave as expected. For example, if we add a node to a graph, it had better be present in the resulting graph. We can formalise this as follows:  $n \in \text{all\_the\_nodes}(\text{add\_node}(G, n))$ . This is an example of an axiom that any implementation of the Graph ADT must satisfy.

## Navigating a Graph

When we move around a graph, we assume we are located at a particular node  $n$ . So the starting node must be specified for any operation that involves navigation. We want to know which nodes can be reached from  $n$  by following an edge that starts at  $n$ . How many such edges might there be? Any number, zero, one, or more. Thus, the result of our operation must be a set. We can now write it down:

`neighbours: Graph × node → set(node)`

The semantics for the neighbours operation needs to take care of the fact that edges are undirected by default. In other words,  $n$  is a neighbour of  $m$  in some graph  $G$  if and only if  $(m,n)$  or  $(n,m)$  is in the edge set of  $G$ . It is possible to go further and write down the semantics using algebraic notation as follows:

$$n \in \text{neighbours}(G, m) \text{ iff } (m,n) \in \text{all\_the\_edges}(G) \text{ or } (n,m) \in \text{all\_the\_edges}(G)$$

However, this is not necessary. Carefully worded statements in English are sufficient for specifying semantics.

## Some Housekeeping

It is usually a good idea to include several more operations. For example, we just added the `outgoing_nodes` operation. So there should be a corresponding `incoming_nodes` operation in order to navigate in the reverse direction. We also included `add_node` and `add_edge`, and so there should be corresponding `remove_node` and `remove_edge` operations.

For convenience, it is good to define an operation that reports the order of a graph, and one that tells us whether a graph is empty. Note that these are redundant, since they can be specified using operations that have already been provided. (There is no requirement that the operations are minimal.)

Finally, it's a good idea to add operations that report whether a graph is connected, or

whether it contains a cycle.

To consolidate your knowledge, we encourage you to write down signature specifications for:

- incoming\_nodes, outgoing\_nodes
- remove\_node, remove\_edge
- is\_empty, is\_connected, is\_cyclic
- order

Now we have defined a variety of operations. The definitions tell us what type of input and output to expect for each operation. However, we have held back from mathematically defining the meaning of these operations using statements like:  $n \in G.add\_node(n).all\_the\_nodes()$ . Such statements are required for a complete formal definition of an abstract data type, and they go beyond the scope of this course.

## Back to the Basics

This is about as difficult as it gets. In the following modules we will go back to looking at some simpler data types, such as lists. If you found the above material a little challenging, we encourage you to press on regardless, then revisit the Graph ADT once you've seen a few other ADTs.

---

<sup>1</sup> The Cartesian product  $A \times B$  is the set of all ordered pairs  $(a, b)$  where  $a \in A$  and  $b \in B$ . So the add\_node operation takes any combination of Graph and node.

## 2.7.3

# Collection ADTs: Lists and Arrays

### Introduction

You may have noticed that computers seem to be at their best when doing the same thing over and over again, ad infinitum. Of course, we know this as *iteration*, the act of performing the same operation(s) repeatedly. (And of course, if a computer could speak, it would point out that programmers are at their best when writing simple loops. For anything more complicated, programmers struggle write what they mean!)

Consider the following pseudocode, which generates the first 100 Fibonacci Numbers:

```
fib = [0, 1]
for i = 3..100:
    next = fib[i-2] + fib[i-1]
    fib = append(fib, next)
```

The result is a sequence of integers, 0, 1, 1, 2, 3, 5, ..., which we have represented as a list. A *list* is a sequence which may include repeats.

Now consider a hand of 10 cards:



In order to play a card game, we might begin by sorting the cards in your hand, first into suits, then by value, resulting in a list like: ['KH', '6H', 'KD', 'QD', '7D', 'AC', '10C', '2C', 'JS', '6S'].

As our final example, you may have a set of tasks to work through. You make a list, then put stars beside the items according to importance:

```
**fix bike
*call Lee
***feed cat
***write essay
*read chapter
**visit gym
```

Rather than getting started on the tasks, you think about how you would go about writing an app to help you manage tasks. With income from such an app, you could probably just pay someone to do the tasks. Anyway, you see that there's a collection of items, each with a numerical priority in the range 1-3, and express the information as a list of pairs: [('fix bike', 2), ('call Lee', 1), ('feed cat', 3), ...]

At this point, we've seen collections of integers representing a numerical sequence, collections of strings

representing playing cards, and collections of pairs of strings and integers, representing tasks with priorities. In all cases, we have collections, in which each element is of the same type.

## Operations on collections

Several data types exist purely for helping us to manage collections that consist of items having the same type. Those items could be simple things like integers, strings, or pairs, but in principle they could be anything. They could represent people, companies, cities, stars, or animal species. On a more mundane level, they could represent toothpaste brands, unallocated phone numbers, or the paper dimensions known to a particular brand of photocopier.

Given that we have a collection of things, what operations might we want to perform? In the card and task examples above, the obvious operation is sorting. Let's try to come up with some more operations.

Suppose that you've finally got underway with your tasks and are writing the essay. You decide to stop for lunch. You hadn't bothered to list eating as a task. (You realise that the task-planning app needs to be smarter than you originally thought.) While having lunch, an important message arrives. While you deal with that, and while your lunch gets cold, the doorbell rings... With interruption upon interruption, you're worried that you'll lose track of what you were doing. So you keep a new collection of ongoing tasks: ['write essay', 'have lunch', 'deal with message', 'answer door']. What are the operations you need to work with this collection, in order to be able to resume the most recently suspended task?

Think of some other real-world problem that involves collections, and that motivates some new operation, such as finding the maximum, or processing items in the order they arrived, or ensuring there are no repeats. Identify the operations you need for this problem.

For good measure, try to formalise these operations, writing down signature specifications. Don't forget the operation which creates the original, empty instance of the type. What more do you need in order to make this into an official ADT?

Given the prevalence of collections in algorithmic problem solving, several standard collection types have been established. Here we will take a look at the List and Array ADTs.

## The List ADT

Recall our first example above, the iteration that produced the Fibonacci Sequence. Each time through the loop, it appended an item to a list. We can formalise this operation as: append: List  $\times$  item  $\rightarrow$  List. However, there's a more general meaning of append, in which two lists are joined or concatenated together.

[What is the signature specification for the append operation, which joins two lists together to make a new list?<sup>1</sup>](#)

Once we have built up a list, we probably want to access its contents. In the case of lists, we can access the first element, or we can get the rest of the list (everything but the first element). Here are the signature specifications:

first: List  $\rightarrow$  item  
rest: List  $\rightarrow$  List

(At this point, you're probably wondering why we don't allow ourselves direct access to the contents of a list using indexing, e.g. using `mylist[i]` to get the  $i$ 'th element. However, this is the Array ADT, and we will discuss it below.)

The List ADT standardly includes a special method called "cons", short for construct, with the following signature:

`cons: item × List → List`

The behaviour of this operation is the inverse of the first and rest operations above. In other words, for any list  $l$ , if we "cons" the head of  $l$  with the rest of  $l$ , we get  $l$ . (Formally, for all non-empty lists  $l$ ,  $\text{cons}(\text{first}(l), \text{rest}(l)) = l$ .)

To round out the definition, we provide a `new_list` operation, an operation to test whether a list is empty, and an operation to report the length of the list. Here is the complete definition:

```
new_list: → List
cons: item × List → List
first: List → item
rest: List → List
is_empty: List → boolean
len: List → int
is_member: item × List → boolean
append: item × List → List
```

## The Array ADT

An array is a multi-dimensional structure accessed using coordinates. For instance, here is a  $3 \times 2$  array:

$$x = \begin{bmatrix} 27 & 31 & 16 \\ 12 & 15 & 42 \end{bmatrix}$$

We access a cell using indexes like `x[2, 1]` and this returns the value 42.

For simplicity, we will look at 1D arrays here. We can write down the following signature specifications:

```
new_array: int → Array
set: Array × int × item → Array
get: Array × int → item
```

Notice how we need to specify a size when we initialise an array. Here's an example of the use of these operations to produce the array `["three", "blind", "mice"]`

```
x = new_array(3)
x = set(x, 0, "three")
x = set(x, 1, "blind")
x = set(x, 2, "mice")
```

For convenience, a programming language would usually allow us to perform the above operations by typing the following instead:

```
x = new_array(3)
x[0] = "three"
x[1] = "blind"
x[2] = "mice"
```

The meaning of the array operations is fairly easy to state. We just want to ensure that when we set an array index  $i$  to have some value  $val$ , then when you access  $i$ , you get  $val$ . In other words:  $\text{get}(\text{set}(x, i, val), i) = val$ .

## Lists vs 1D arrays

The List and Array ADTs look very different, even though both can be used to represent a sequence of items. One source of confusion is that programming languages usually provide a mash-up of the two operations, and call it a list!

The key to understanding the difference is that a list can expand or shrink as needed, while an array has a fixed size that is specified when it is created. This difference might seem arbitrary in the case of 1D arrays, but for arbitrary dimension arrays (or matrices), it makes perfect sense. There are deeper reasons for making this distinction that have to do with efficient storage in the memory of a computer, that go beyond the scope of this course.

## What is this *item* you speak of?

We have been slightly careless in writing formal definitions that involve an undefined thing that we simply refer to as "item". What is an item? In reality, it is anything we need it to be in order to model the problem. But since we envisage collections of elements all having the same type, we should do more than this.

We can think of "item", as used in our signature specifications, as standing for any type that we care to name. Thus, the List ADT, defined over integers, would be as follows, where the only changes from the above definition are shown in bold:

```
new_list: → List
cons: int × List → List
first: List → int
rest: List → List
append: List × List → List
is_empty: List → boolean
```

Thus, our collection ADTs are templates. Strictly speaking, we should specify the type of the contained item at the same time as specifying the type of the container. For example, we might have `List(int)` or `Array(string)`.

[How would you write the type for an array of lists of integers? <sup>2</sup>](#)

## More collection ADTs

In this module we've seen the List and Array ADTs. These are just two of several standard collection ADTs, and we will look at the rest in the next module.

<sup>1</sup> append: List × List → List

<sup>2</sup> Array(List(int))

## 2.7.4

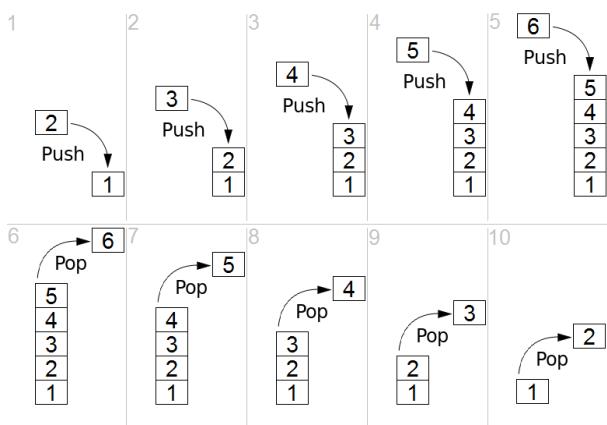
# More Collection ADTs

Abstract Data Types are a fairly straightforward idea, once you get the hang of them. In this module, we will cover the rest of the collection data types: Stack, Queue, Priority Queue and Dictionary.

### The Stack ADT

We already encountered a popular motivation for stacks, namely keeping track of uncompleted tasks during interruptions. If you pause your work on an essay in order to have lunch, once you finish your lunch, you will hopefully get back to that essay. However, if an important message arrives during lunch, you might deal with that before finishing your lunch (and then getting back to the essay). And if the doorbell rings while you're dealing with that message...

The stack is a last-in first-out (LIFO) structure. We add and remove items using the push and pop operations:



The contents of a stack, like a list, is just a sequence of items. However, unlike a list, we are constrained in how we can manipulate those items. There's only one place where we can operate: the top.

The operation of push adds an element to the stack. So, given a stack and an item, it pushes the item onto the stack and returns the new stack.

[What is the signature specification for the push operation? <sup>1</sup>](#)

There are three more operations. Top accesses the first item on the stack. It allows us to inspect this item, without modifying the stack. It is an error to use this operation when the stack is empty.

[What is the signature specification for top? <sup>2</sup>](#)

Pop returns the stack that results from removing the top element. The resulting stack is one element smaller. It is also an error to use this operation on an empty stack.

[What is the signature specification for pop? <sup>3</sup>](#)

Is\_empty returns True just in case the stack is empty.

[What is the signature specification for is\\_empty? <sup>4</sup>](#)

[Is our Stack ADT specification complete, in terms of operations? <sup>5</sup>](#)

## Aside: Implementing stacks using lists

We could write down an implementation of the Stack ADT using lists. In fact, this is how it is done in Edgy (see the [Javascript code](https://github.com/snapapps/edgy/blob/master/edgy/collections.js#L595) (<https://github.com/snapapps/edgy/blob/master/edgy/collections.js#L595>) for Edgy's stack implementation). Here's the pseudocode:

```
def new_stack():
    return new_list

def push(s, item):
    return cons(item, s)

def top(s):
    return first(s)

def pop(s):
    return rest(s)

def size(s):
    return len(s)

def is_empty(s):
    return is_empty(s)
```

This is implementation detail, the "how". It demonstrates that a stack is a list with restricted operations. *This implementation is not part of the ADT specification.* With a little ingenuity, we can also implement stacks using arrays, as shown below.

```
def new_stack():
    s = new_array(1000000000)
    s[0] = 0
    return s
def push(s, item):
    s[0] += 1
    size = s[0]
    s[size] = item
    return s
def top(s):
    size = s[0]
    return s[size]
def pop(s):
    s[0] -= 1
    return s
def size(s):
    return s[0]
def is_empty(s):
    if s[0] == 0:
```

```

    return True
else:
    return False

```

We've now seen two implementations for stacks. Remember that the Stack ADT is just the definition, i.e. the syntax (signature specification) and the semantics (axioms). There can be more than one implementation of an ADT; but these are not the ADT!

Crucially, we are able to use an ADT based on its definition, without paying attention to the implementation. We can use the axioms to test whether an implementation is correct, without inspecting the implementation (i.e. black box testing).

## Summary

Here is the syntactic specification for the Stack ADT:

```

new_stack: → Stack
push: Stack × item → Stack
top: Stack → item
pop: Stack → Stack
is_empty: Stack → boolean
size: Stack → int

```

## The Queue ADT

Recall that a stack is a LIFO structure. A queue, on the other hand, is FIFO.

### What does FIFO stand for?<sup>6</sup>

The queue operations are enqueue and dequeue. Enqueue adds a new item to the queue, while dequeue reports the item at the head of the queue, and removes it from the queue. Can you write down the complete signature specification for a queue? Give it a go now.

### Check your definition of the Queue ADT<sup>7</sup>

You might like to test your understanding of the queue operations by writing down some axioms.

### What are the axioms for is\_empty?<sup>8</sup> What are the axioms for length?<sup>9</sup>

Here's some more Queue axioms. You are not expected to remember them, but they are good to think about. The first one says that if we enqueue an item to an empty queue, then it must be at the front of the queue:

$\text{front}(\text{enqueue}(\text{new\_queue}, \text{val})) = \text{val}$

Here's a more complex axiom, which states that so long as the queue has at least one item in it, you can do enqueue then dequeue, or dequeue then enqueue, to get the same result:

$\text{dequeue}(\text{enqueue}(Q, \text{val})) = \text{enqueue}(\text{dequeue}(Q), \text{val}) \ (\text{size}(Q) > 0)$

For example, both of the following produce the same result, namely a queue of length 1 containing "Mary".

```

Q = new_queue()
Q = enqueue(Q, "John")
Q = enqueue(Q, "Mary")
Q = dequeue(Q)

```

```

Q = new_queue()
Q = enqueue(Q, "John")
Q = dequeue(Q)
Q = enqueue(Q, "Mary")

```

## The Priority Queue ADT

A priority queue is like a queue, only you can specify the priority for an element when you add it, and you can modify the priority of an element that's in the queue. If you've ever waited in line at the airport and then had your flight called out, and been asked to jump to the head of the queue, you have experienced a priority queue. Similarly, if you have ever waited in the emergency department of a hospital for hours, continually bypassed by more urgent cases, you know what it feels like to be an element of a priority queue.

For concreteness suppose that we have three priorities: high (3), medium (2), and low (1). And suppose the following events unfold in the emergency department, in sequence:

- Kim arrives with a grazed knee from falling off her skateboard while attempting a new trick. The staff assess her as low priority and ask her to take a seat.
- Lee is suffering trauma from a car accident and needs urgent surgery. The staff assess him as high priority and he is sent to an operating theatre.
- Sue arrives with a fractured arm. The staff assess her as medium priority and ask her to wait, and promise it won't be for long.
- Alan has concussion and is assessed as medium priority.
- Sue is called for treatment

Here is the corresponding sequence operations and the resulting priority queue after each one:

1. `Q = new_priority_queue() → []`
2. `enqueue(Q, Kim, 1) → [(Kim, 1)]`
3. `enqueue(Q, Lee, 3) → [(Kim, 1), (Lee, 3)]`
4. `front(Q) → Lee`
5. `dequeue(Q) → [(Kim, 1)]`
6. `enqueue(Q, Sue, 2) → [(Kim, 1), (Sue, 2)]`
7. `enqueue(Q, Alan, 2) → [(Kim, 1), (Sue, 2), (Alan, 2)]`
8. `front(Q) → Sue`
9. `dequeue(Q) → [(Kim, 1), (Alan, 2)]`

Extend your definition of the Queue ADT to the Priority Queue ADT. Don't forget to define the set of objects it applies to. Just one operation needs a different signature specification.

[What is the modified signature specification required for the Priority Queue ADT? <sup>10</sup>](#)

One further operation is needed. Suppose that the status of the patients waiting in the emergency room can change over time. For example, perhaps Alan's concussion is worse than originally expected and he faints, and his priority is changed from medium to high. What operation would be needed for this?

[Suggest an operation for updating the priority of an existing item? <sup>11</sup>](#)

## The Dictionary ADT

We're familiar with the idea of a dictionary, which allows us to look up a word to get information such as pronunciation or meaning. The Dictionary ADT generalises from this idea, allowing us to access any kind

of information (the "value"), using any fixed lookup "key". Here are some examples:

Phone List	
Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution	
aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
python.org	18.21.3.144
sourceforge.net	51.98.23.53

Word Frequency Table	
computational	25
language	196
linguistics	17
natural	56
processing	57

(<https://www.alexandriarepository.org/wp-content/uploads/20151025214652/maps03.png>) Note that the lookup is from key to value (e.g. name to phone number, domain name to IP address, etc), and not the other way around.

So the most basic operations on a dictionary are to create an empty one, assign a value to a key, and look up that value.

new\_dict: → Dict

set: Dict × key × value → Dict

get: Dict × key → value

It is common to use shorthand for dictionaries. So instead of writing: mydict = set(mydict, "Dana", "x642"), we typically write mydict["Dana"] = "x642".

Note also that the type of key and value is not specified, though it's usual to assume that the key is not something that can ever be modified once it has been created (for this you'd delete it and create a new key to hold the same value).

Dictionaries typically support the following additional operations:

del: Dict × key → Dict

in: Stack × key → Boolean

size: Dict → int

keys: Dict → List

 <sup>1</sup> push: Stack × item → Stack

 <sup>2</sup> top: Stack → item

 <sup>3</sup> pop: Stack → Stack

 <sup>4</sup> is\_empty: Stack → boolean

new\_stack: → Stack

It is also a good idea to include a size operation:

size: Stack → int

 <sup>5</sup> It is not complete. We need to provide an operation that creates a new empty stack:

 <sup>6</sup> first-in first-out

 <sup>7</sup> new\_queue: → Queue

enqueue: Queue × item → Queue

front: Queue → item  
dequeue: Queue → Queue  
is\_empty: Queue → boolean  
len: Queue → int

➊<sup>8</sup> is\_empty(new\_queue) = True  
is\_empty(enqueue(Q, val)) = False

➋<sup>9</sup> len(new\_queue) = 0  
len(enqueue(Q, val)) = len(Q) + 1  
len(dequeue(Q)) = len(Q) - 1

➌<sup>10</sup> enqueue: PriorityQueue × item × priority → PriorityQueue

➍<sup>11</sup> update\_priority: PriorityQueue × item × priority → PriorityQueue

## 2.7.5 Application: Graph Traversal

In an earlier module we saw how to traverse a graph using Breadth First Search (BFS) and Depth First Search (DFS). Recall how we used lists to keep track of the unvisited (red) and to-be-visited (green) nodes.

We return to the graph traversal problem, in order to demonstrate the List, Queue and Stack ADTs.

### BFS with the List ADT

We need to keep track of the nodes that are still to be visited. Instead of colouring them green, we will simply maintain an extra list Q.

```
Algorithm BFS_original
input: two nodes A, B
output: True if there is a path from A to B, False otherwise

Q := new_list
Q := append(Q, A)

repeat until is_empty(Q)
    C := first(Q)
    Q := rest(Q)
    mark C as visited
    if C = B
        return True
    foreach neighbour N of C
        if N is not marked as visited
            Q = append(Q, N)

return False
```

### BFS using the Queue ADT

Here we will replace the list Q with the Queue ADT. Notice that the structure of the program is the same. We just replace List operations (new\_list, append, first, rest) with the corresponding Queue operations (new\_queue, enqueue, front, dequeue).

```
Algorithm BFS
input: two nodes A, B
output: True if there is a path from A to B, False otherwise

Q := new_queue()
Q := enqueue(Q, A)

repeat until is_empty(Q)
    C := front(Q)
    Q := dequeue(Q)
```

```

mark C as visited
if C = B
    return True
foreach neighbour N of C
    if N is not marked as visited
        Q = enqueue(Q, N)

return False

```

## DFS using the Stack ADT

Here we will replace the queue Q with a stack, and use the stack operations (new\_stack, push, top, pop).

```

Algorithm DFS
input: two nodes A, B
output: True if there is a path from A to B, False otherwise

S := new_stack()
S := push(S, A)

repeat until is_empty(Q)
    C := top(S)
    S := pop(S)

    mark C as visited
    if C = B
        return True
    foreach neighbour N of C
        if N is not marked as visited
            S = push(S, N)

return False

```

---

## 2.7.6

# An extreme example: integers

It may surprise you, but we can extend our concept of abstract data types to something as fundamental as integers.

### The Integer ADT

The Integer ADT applies to the set of integers  $\mathbf{Z}$ , and includes the following operations. (The first operation could have been called zero or new\_int.)

```
0: → int
+: int × int → int
-: int × int → int
-: int → int
++: int → int
--: int → int
is_zero: int → bool
```

This is the *syntax* for integers. It consists of a list of *signature specifications*, which specify the available operations and how they are used. Thanks to these syntactic definitions, we know that the following expressions are well-formed:  $x+y$ ,  $-x$ ,  $x+(y+z)$ ,  $x++$ ,  $\text{is\_zero}(x)$ . We also know that the following are ill-formed:  $x+$ ,  $xy$ ,  $\text{is\_zero}.$

The semantics is expressed using axioms, e.g.:

```
x + 0 = x
x + -x = 0
x + y = y + x
x + (y + z) = (x + y) + z
is_zero(0) = True
is_zero(0++) = False
```

---

## **2.8**

# **Networks of Actions: Planning and Decision Making**

[2.8.1 Suspicious Boyfriends](#)

[2.8.2 Building the graph](#)

[2.8.3 Planning: putting it all together](#)

---

## 2.8.1 Suspicious Boyfriends

### A warm up problem: suspicious boyfriends

It's after midnight... two couples emerge from a club. They want to get across town to another club. They have a single motorbike between them, which they can all drive. It carries at most two people at a time. The two guys are suspicious and will not tolerate their girlfriend being left alone with the other guy. Can they use the bike to get across town from the origin to the destination, or must they walk? (This is a restatement of the [jealous husbands problem](https://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem) ([https://en.wikipedia.org/wiki/Missionaries\\_and\\_cannibals\\_problem](https://en.wikipedia.org/wiki/Missionaries_and_cannibals_problem))).

### Data modelling activity

Try to represent this problem using physical objects, e.g. coins, pencil, and paper. Answer the following questions for your representation:

1. What real-world entities are modelled in your representation? What represents what?
2. What are the start and end configurations?
3. How many different configurations are there?

Can you solve the problem using this representation?

Terminology: a configuration of real world entities is also known as a *state*. Think of this as a "state of affairs in the world".

This is a *physical* model. The next step is come up with an *abstract* model, where each state is represented by a label. For instance, we might write the initial state as " $g_1=\text{start}$ ,  $g_2=\text{start}$ ,  $b_1=\text{start}$ ,  $b_2=\text{start}$ ". However, there's any number of ways you might like to represent the state of people, bike, and locations. For example, a state could be represented using a little diagram, like a picture of a domino block, with two halves representing the two locations, and figures to represent the people and the bike.

### Discussion of illegal states

The problem talked about suspicion. We want to rule out some of the states as "illegal". Using the notation you devised above, write out the illegal states, and discuss with a small group, to make sure you're agreed.

Try to express these illegal states more compactly. E.g. we could talk about  $g_1$  being present with  $b_2$ , regardless of whether that is at the origin or the destination, by writing  $g_1=b_2$ .

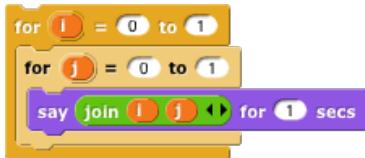
Now try to write down a boolean expression for an illegal state, combining such expressions using  $\wedge$  (and),  $\vee$  (or), and  $\neg$  (not) operators.

(As you think about this, you might need to decide whether being on the bike at the origin or destination counts as being in that location.)

Let's put this on hold for now. We will use it later when we're generating the states, in order to rule out illegal states.

## Aside: generating all possible states

How will we enumerate all of the states? This requires generating all possible combinations, where each combination looks something like:  $g_1=0, g_2=1, b_1=0, b_2=1, \text{bike}=1$ . Here's a hint:



Try to write code to generate all 32 nodes. Use variable names that correspond to the way you labelled each entity, e.g. girl1 or g1.

It is a good idea to store the location of each person and the bike as a separate attribute. So make five node attributes corresponding to each of your variable names. They can be the same name, so you might end up with something like the following:



That leaves the question of how to label the nodes. The simplest way is to use the join block, to combine the variables into a string that serves as the node label, e.g. 10101.

**Putting it together:** Now that you can generate the states, bring in your boolean expression for detecting illegal states, and colour all those states red.

## Building the graph

At this point, you'll have a collection of states, and the problem is how to link them up. Which states are reachable from which states, in a single "move" (or *transition*), i.e. in a single motorbike trip from origin to destination or back?

Here are some things to think about:

- what happens to the location of the bike after each transition?
- what is the capacity of the bike? i.e. how many people can change location with each transition?
- will the transitions (edges) be directed? i.e. will you use an undirected graph or a directed graph?

It's easy enough to check whether someone moved, using the following condition, assuming that we stored the locations using node attributes.



So the problem becomes one of checking that the bike moved appropriately, and that at least one person moved (i.e. there was a driver), but not too many people moved.

You will need to do that for all possible pairs of nodes. How do you generate all possible pairs of something? Hint: Look back to our discussion of generating all possible states, but this time, you'll want to use the "all the nodes" block, with a suitably-chosen control block.

## Pruning the graph

Once you've added in all of the edges, you can simply delete all of the illegal states, and any transitions that involve those states will disappear at the same time.

Note that it was convenient to generate all the states (including illegal ones), then add all the transitions before deleting the illegal states. Instead, we could have generated the states, deleted illegal states, then added transitions. The former turns out to be considerably easier to code, even though it is slightly less efficient in terms of the total number of computational steps (creating nodes only to remove them later). Can you think why this is?

## Search

The final task is to perform breadth first or depth first search, to try to find any paths from the start node to the end node.

What is the length of the shortest path, and how many shortest paths are there? Are there longer paths? What's the longest path?

What algorithm design strategy have we used to solve the suspicious boyfriends problem?

## 2.8.2 Building the graph

### Setting up the Graph

In planning and decision making, a graph node is used to model a particular state of affairs in the world, such as the configuration of pieces on a game board. If one state is reachable from another (e.g. by the move of a game), then we add an edge between them. Under what circumstances would such an edge be directed or undirected?

Now, some states may turn out to be unreachable. We need to decide if we're going to generate them all then link them up (as we did for the suspicious boyfriends case) or if we're going to start with an initial state and generate just the states that are reachable (as we saw for tic-tac toe). Could we have done things differently? Could we have applied the generate-all-then-link method for tic-tac-toe? Could we have applied the build-out-from-start-state for suspicious boyfriends case? Why might it be preferable to use one of these methods over the other?

### Labelling the nodes

Remember that the nodes of a graph are just a set. We give them names or "identifiers" in order to have a convenient way of referring to them. These identifiers could just be integers, but it's often more helpful to assign meaningful names to nodes, names that remind us what they represent. Thus, we came up with names like -x- | xo- | ox- for tic-tac-toe.

Such names are strings, and it is almost always a bad idea to break such strings back down in order to analyse a state in our program. Instead, it's better to have separate human-readable and machine-readable ways to represent information about a state.

For example node -x- | xo- | ox- could have an attribute called "board" whose value is a list of rows, where each row is a list of integers, as we see below:



Now when it comes to identifying which player has won, we can conveniently iterate over cells by accessing the board attribute, instead of trying to split apart the label into x's, o's and -'s.

### Generating all nodes then linking

Generating nodes in a state graph may require nested iteration. Suppose we are solving a problem that involves three entities (such as a farmer, goat and chicken, or three light globes), and each entity can be in one of several locations (such as river-side or boat, or on or off). Then we would need the following nested for loops:

```
for entity1 in <possibilities for entity1>:
    for entity2 in <possibilities for entity2>:
```

```
for entity3 in <possibilities for entity3>:
    new_node(entity1_entity2_entity3)
```

This is a general-purpose pattern for generating all combinations. For example, we can put words together to generate simple sentences as follows. You might like to try this for sentences of 3 or 4 words in length.



Once we have generated all the nodes, we need to link them. The most natural way to do this is to identify the precise conditions for linking a pair of nodes, based on the information available in just those nodes. For example, if we were applying this method to tic-tac-toe, we would add an edge just in case the nodes differ by a single piece, e.g.

step 5: -x- | xo- | ox-  
step 6: -xo | xo- | ox-

So, how do we detect that exactly one position has changed? We can define a function that counts up the changes as follows. It processes all the cells, comparing what the two states have for each cell:

```
function count_changes(n1, n2):
    num_changes = 0
    for i = 1 to 3:
        for j = 1 to 3:
            if n1.board[i][j] = 0 and n1.board[i][j] ≠ n2.board[i][j]:
                num_changes = num_changes + 1
    return num_changes
```

Note that we only consider changes that involve an empty cell that becomes something other than empty, using the test: `n1.board[i][j] = 0`

Now we can consider all pairs of nodes, and use the above test to see if they should be connected or not.

```
for node1 in <node set>:
    for node2 in <node set>:
        if count_changes(node1, node2) = 1:
            new_edge(node1, node2)
```

## Building out from the start state

Our other way of creating nodes is to begin with a start state, such as an empty tic-tac-toe board, and consider all possible ways to transition out of that state, such as placing an x on the board. In this approach, we will create nodes and edges at the same time.

As before, we need to devise a way to label our nodes, and might want to put information into one or more attributes for ease of access later.

Then we need to generate all "next states", i.e. enumerate all possible moves and create a state for each one, linked from the start state. For example, in a tic-tac-toe board, there's nine places for the first player to place their piece. (Of course, some of these are duplicates because of the inherent symmetries of the game, but we will ignore that for now.)

In general, a move consists of replacing an empty cell (value zero) with an x piece (value 1) or an o piece (value 2). Given a particular state, we want to scan it for empty cells. For each empty cell, we want to create a new state which has the piece in that cell.

```
function generate_next_states(state, piece):
    for i = 1 to 3:
        for j = 1 to 3:
            if state.board[i][j] = 0:
                new_state = state
                new_state.board[i][j] = piece
                new_edge(state, new_state)
```

Each time we create a new state, we also create an edge that goes to that state.

## 2.8.3

# Planning: putting it all together

### Suspicious Boyfriends Again

It's time to put together the ideas in this series of modules on planning and decision making, to provide a complete solution of the Suspicious Boyfriends problem. First we give the algorithms: generating legal states, generating the edges between them, and finding a path from the initial state to the final state. At the end of this module we give a link to a complete solution in Edgy for you to try.

### Generating the Legal States

We will let 0 and 1 represent the origin and destination. We use nested iteration to "multiply out" all the possibilities.

```
function make_states()
    for B1 in 0 to 1:
        for G1 in 0 to 1:
            for B2 in 0 to 1:
                for G2 in 0 to 1:
                    for M in 0 to 1:
                        if not (B2=G1 and B1≠G1 or B1=G2 and B2≠G2):
                            add node B1_G1_B2_G2_M
```

An illegal state occurs when G1 is in the presence of B2 ( $G1=B2$ ) and B1 is absent ( $B1\neq G1$ ), and conversely for G2.

### Adding the Edges

We consider all pairs of legal nodes ( $n_1, n_2$ ), and check whether there is a legal move that takes people (and the motorbike) from location 0 to location 1.

```
function add_edges()
    foreach node n1:
        foreach node n2:
            num_moves = 0
            legal = True
            if motorbike moves from 0 to 1:
                foreach person:
                    if person moves from 0 to 1:
                        increment num_moves
                    if person moves from 1 to 0:
                        legal = False
            if legal and 1 <= num_moves <= 2:
                add an edge from n1 to n2
```

Notice how we count the number of people who move, and we make sure they move in the right direction, before adding an edge.

## Iterative DFS

We will need to apply DFS to search the graph for a path from start to finish. We use an additional stack T which records the fact that we have visited a previously unvisited node (C). If no unvisited nodes can be reached from here, we pop C from the stack. It is not part of a path to the solution. (If it was, we would have already found a path via one of C's neighbours that did not involve going via C.) An Edgy implementation of this algorithm is available, and defines a reporter block called dfs (see [dfs-iterative](https://www.alexandriarepository.org/wp-content/uploads/20150414005907/dfs-iterative.xml) (<https://www.alexandriarepository.org/wp-content/uploads/20150414005907/dfs-iterative.xml>)).

```
function dfs(A, B)
    mark all nodes as unvisited
    S = newStack(A)
    T = newStack()

    repeat until isEmpty(S)
        C = top(S)
        S = pop(S)
        T = push(T, C)
        mark C as visited
        if C == B
            return T
        viable = False
        foreach N in neighbours(C)
            if N is not marked as visited
                S = push(S, N)
                viable = True
        if not viable
            T = pop(T)
    return []
```

## Edgy Implementation

A complete implementation of the above pseudocode is available: [Suspicious Boyfriends](https://www.alexandriarepository.org/wp-content/uploads/20150414005907/Suspicious-Boyfriends.xml) (<https://www.alexandriarepository.org/wp-content/uploads/20150414005907/Suspicious-Boyfriends.xml>)

## 2.9 Path Finding

### [draft for preview only; BM]

You should by now know how to find a path between two nodes in a graph using one of the two basic patterns for graph traversal, breadth first search and depth first search.

A more interesting problem than just finding any path is to find the *best* path. Best, of course, can have many different interpretations. Probably the most basic one is that of the shortest path.

Imagine you want to travel from Melbourne to Sydney driving the shortest distance. To solve this problem, you can model the road map as an (undirected) graph that has cities as its nodes and roads as its edges. The driving distances between the cities can be captured as weights on the edges. Finding the shortest route between the two cities now amount to finding the shortest path between two nodes in the network, where the length of a path is defined as the sum of all edge weights on this path.





[image from <http://en.wikipedia.org/wiki/File:GA20891.pdf> under [Creative Commons](#)

([http://en.wikipedia.org/wiki/en:Creative\\_Commons](http://en.wikipedia.org/wiki/en:Creative_Commons)) [Attribution 3.0 Australia](#) (<http://creativecommons.org/licenses/by/3.0/au/deed.en>) license.]

But finding shortest path lets us solve a much broader class of problems. For example, if we want to fly from A to B our definition of *best* path may be the cheapest connection. In that case we would label the edges with the prices for flight segments. Finding shortest paths even lets us solve important industrial problems, for example, how the processing steps in a production facility should be sequenced to optimize its throughput and efficiency.

It should not be too hard to sketch a simple algorithm to find such a shortest path.

[Describe a basic idea for a naive algorithm to solve the shortest path problem](#)<sup>1</sup>

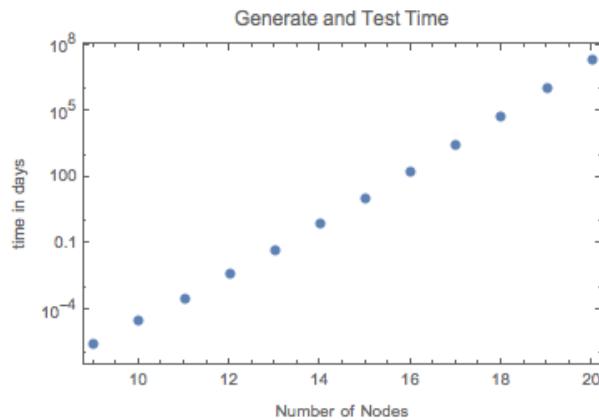
Surely this would work. But let us stop for a second. Is this a good way to go about the problem? The core question is, how many different paths would we have to check? Let's just look at what could happen in the worst case.

[Which graph with n nodes has the maximum number of possible paths between two nodes A, B.](#)<sup>2</sup>

[How many paths between two nodes A, B, can exist at most in a graph with n nodes and m edges?](#)<sup>3</sup>

This is bad news. Look at the plot of this function below. Let's assume that generating and checking a single path takes just 1 microsecond, then checking all 2606501 possible paths for K10 would then already take 2.6 seconds! And that's just 10 cities! You may think that is not so bad, but the function

grows incredibly fast. There are 1747509302894800001 possible paths between any pair of cities in a complete network of 20 cities, which means that finding the shortest path would now take at least 1747509302894800001 microseconds or 55413.2 years!!



This is typical of naive generate-and-test algorithms. Often the number of possible solutions that have to be tested grows so fast with the problem size that generate and test is prohibitively expensive and we need to find a faster algorithm (hoping that one exists).

[Can you compare the number of possible paths for a graph with n nodes to the exponential function  \$2^n\$ <sup>4</sup>](#)

[Can you imagine why the exponential function  \$2^n\$  often enters into the number of candidate solutions that have to be checked by a generate-and-test algorithm?<sup>5</sup>](#)

In general, if a generate-and-test algorithms has to check all possible subsets, its runtime will grow (at least) exponentially with the size of the problem.

Surely, we should be able to find a better solution for the shortest path problem. If we look at a map to find a shortest path, we might be tempted to employ a heuristic (a method of informed guessing), for example, "keep moving in the general direction of the destination", but this does not work in an arbitrary graph that is not embedded into an actual physical space (more mathematically speaking, into a metric space, where we can rely on the usual laws of distances, for example that the direct line from A to B is never longer than going from A to C and then from C to B).

Since we want a general method, we need a better idea. [Edsger W. Dijkstra](#)

([http://en.wikipedia.org/wiki/Edsger\\_W.\\_Dijkstra](http://en.wikipedia.org/wiki/Edsger_W._Dijkstra)) found this idea in 1956. His algorithm to solve the shortest path problem is known as [Dijkstra's algorithm](#) ([http://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)).

<sup>1</sup> Enumerate all paths from A to B, store them in a list or other collection and then search for the minimum.

<sup>2</sup> A complete graph. It allows us to take any path with n nodes, (n-1) nodes, (n-2) nodes, ..., down to two nodes (the direct connection). On each of these paths we can visit the nodes in any order.

<sup>3</sup> Lots! If we assume that the graph is complete (as above) we can take intermediate paths of any length between 0 and (n-2) nodes to get from A to B. The order of the nodes on these paths is arbitrary. Since n items can be arrange in  $n!$  ways, we obtain as the total number of paths

$$paths(n) = \sum_{k=0}^{(n-2)} \binom{n}{k} \cdot k!$$

<sup>4</sup>

$2^n$  grows much more slowly than the number of possible paths. This should be obvious, because  $2^{(n-2)}$  is the number of combinations of intermediate nodes that we can pick between A and B in a complete graph and the formula needs to account for the reordering of nodes on top of this!

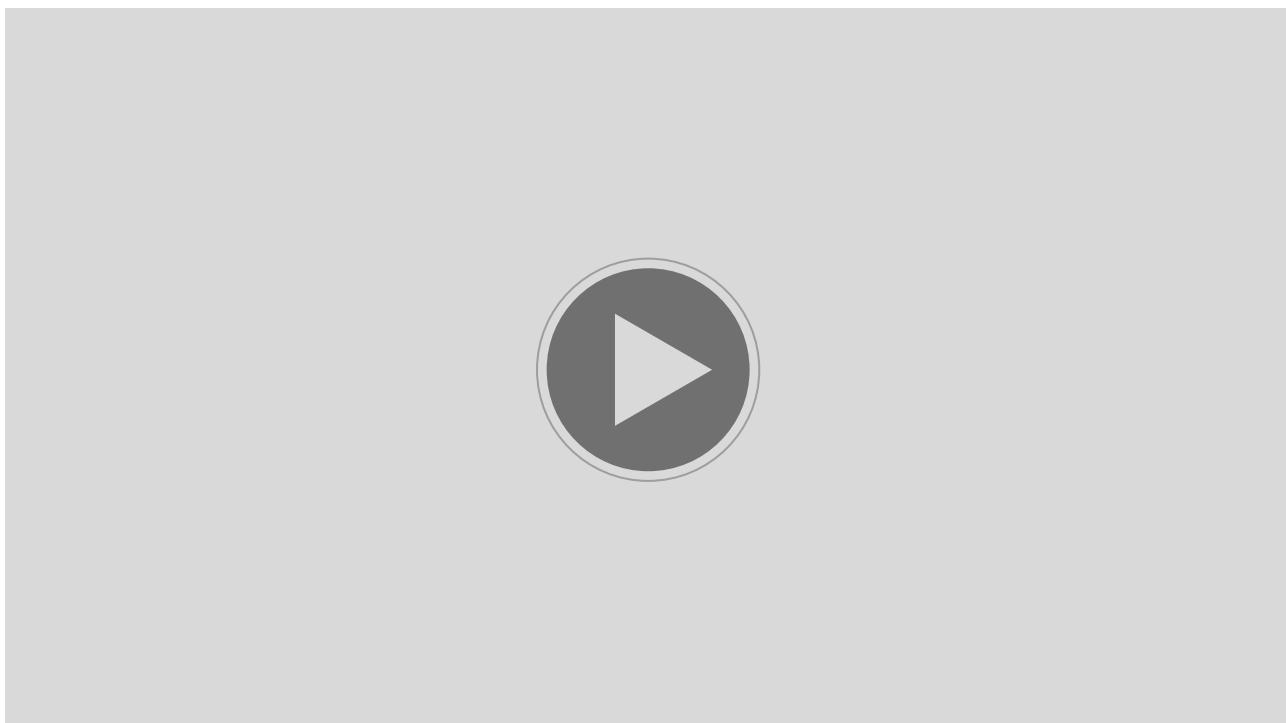
- 5  $2^n$  is the number of subsets that can be picked from a set of n items (the size of the powerset on n items). Generate and test algorithms often have to find just such a subset. For example, if we ignored the fact that we can reorder the nodes on a path in the above problems, there are  $2^{(n-2)}$  ways to pick paths.

## 2.9.1

# Dijkstra's Shortest Path Algorithm

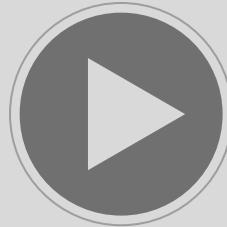
Dijkstra's algorithm is the best solution to finding a shortest path between two nodes  $A, B$  in a weighted graph  $G$  that only has positive edge weights (this is important, as you will see later). It can also be used to find the shortest paths from a given node to all other nodes in the graph. Finding the shortest paths between *all possible pairs* of nodes in a given graph is a related problem, and Dijkstra can be used to solve this, but in many cases another algorithm is a better choice. We will look at this later. For now we focus on the problem of finding the shortest connection between two given nodes  $A, B$ .

The following video explains the basic idea and logic behind Dijkstra's Algorithm.



(<https://www.alexandriarepository.org/wp-content/uploads/20150326132651/Dijkstra.mp4.mp4>)

In summary, the algorithm incrementally builds a tree  $T$  of shortest paths from  $A$  to all nodes. At all points of time all nodes in the graph have an estimate on their shortest distance from  $A$ . This estimate is an *upper bound*. In the beginning, there is no knowledge of about these distances, so the estimate is initialised to infinity. In each iteration a single node  $C$  is added to  $T$ . This is the closest node to  $A$  that is not yet in  $T$ . Immediately after adding a new node  $C$  to the tree, the distance estimates of the neighbours  $X$  of  $C$  that are not yet in  $T$  are reconsidered. If they can be reached via  $C$  on a shorter distance than the previous estimate, their distance will be updated. This step is called a "relaxation". Its result is a better (tighter) upper bound on the shortest distance of  $X$  from  $A$ . The relaxation step is detailed separately in this video.



(<https://www.alexandriarepository.org/wp-content/uploads/20150326130003/Relaxation-in-Dijkstra.mp4.mp4>)

When  $X$  is the node outside of  $T$  with the lowest distance estimate, we know that its upper bound is tight and cannot be relaxed further.

[Can you explain why this is the case?](#)<sup>1</sup>

This is the reason why Dijkstra's algorithm can work in greedy fashion. We know when a node can be safely added to  $T$  and we know that we never need to reconsider such a node.

[Can you explain why this algorithm is not guaranteed to work if there are negative weights \(distances\) in the graph?](#)<sup>2</sup>

The upshot of this consideration is that we cannot use a greedy approach a la Dijkstra if there are negative weights. Instead we will have to revise the distance estimates of all nodes continuously. This is, of course, much more costly (i.e. requires more computational effort). The corresponding algorithm is called Bellman-Ford's shortest path algorithm and will be discussed later.

[Which ADT would be suitable to keep track of the distance estimates?](#)<sup>3</sup>

Here is the high-level pseudocode for Dijkstra's algorithm and an implementation in Edgy (NB: the edgy implementation currently does not use the priority queue ADT due to a bug in its implementation. This will be fixed soon.)

### Algorithm Dijkstra

```

input: an edge weighted graph G; two nodes A, B;
output: the shortest path distance from A to B
assumption: B is reachable from A,
            all edge weights w(x,y) are positive

```

```
Q=newPrioQueue() (* the queue of open nodes not in T *)
```

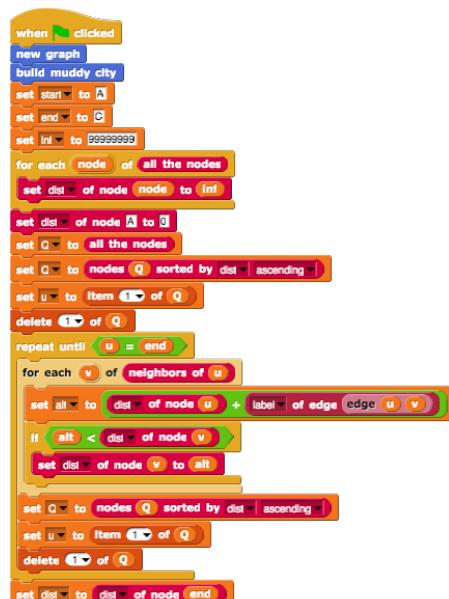
```
(* Q is a minimum priority queue *)

foreach node x in G
    initialize distance estimate d(x) to infinity
initialize d(A) to 0

foreach node x in G
    Q := enqueue(Q,x)
current = minimum(Q)
Q := dequeue(Q)

while current is not B do
    foreach neighbour x of B
        update d(x) := min(d(x), d(B)+w(B,x))
        Q := updatePriority(Q, x, d(X))

return d(B)
```



**Exercise:** Change the pseudo-code and edgy implementation such that it computes the shortest distance to all nodes from A

**Exercise:** Extend the pseudo-code and Edgy implementation such that it memories the shortest path to each node and returns the shortest path to B. *Hint:* The basic idea is the same that we used for strong paths in BFS, by using a predecessor attribute at each node.

<sup>1</sup> X has the shortest distance from any node in T (as its estimate is the lowest). If we were go via some other node Y outside of T, the lowest possible distance from A to X on such a path would be the distance of A to Y. Since the distance of A to Y is greater than the distance from A to X we know that there can be no alternative shorter path from A to X involving nodes outside of T. Thus the distance estimate to X is now tight and it being the lowest, we can safely add X to T.

<sup>2</sup> Consider the explanation of the relaxation step above. Assume that X is a neighbour of Y and that the distance from Y to X is negative. Let the current distance estimate of X be  $d(X) > 0$ . Let the shortest distance from A to Y be  $d(Y) > 0$  and let the

edge weights (distances)  $d(Y,X) = -d(A,Y)$ . The distance estimate from  $A$  to  $X$  can thus be bettered (decreased) by going via a node that is further from  $A$  than  $X$ . Thus, we cannot guarantee that a node does not need to be reconsidered when it is added to the tree.

- <sup>3</sup> We always want to handle the node with the shortest distance estimate next and once we have handled it we do not need to keep track of it anymore. Distance estimates are continuously updated. Thus an adaptive (updatable) priority queue that sorts priorities in ascending (i.e. a min priority queue) is ideal.

## 2.9.2

# Bellman-Ford's Shortest Path Algorithm

[draft for preview only; BM]

*Note: This module assumes familiarity with Dijkstra's Algorithm.*

Dijkstra's Algorithm is clearly the best choice for computing a shortest path between two nodes if there are no negative edge weights in the graph. However, if there are negative weights it does not work and we need to proceed differently.

Before we discuss how to find a shortest path in the presence of negative edge weights, note that the presence of negative weights implies that the graph must be directed.

[Why is it not meaningful to look for a shortest path in an undirected graph with negative edge weights?](#)<sup>1</sup>

The essentially advantage of Dijkstra's algorithm is that it never reconsiders an edge once it has been processed. This is what makes it fast. However, if there are negative edge weights we know that we may still find a better (shorter) distance to the node that this edge leads to at a later point of time so that we may have to reconsider it. The edge relaxation step itself still works perfectly fine, of course. We simply may have to repeat it several times even for the same edge.

Our first shot at the problem, keeping the same edge relaxation, may look like this

Algorithm not-quite-Bellman-Ford

```

input: an edge weighted directed graph G; two nodes A, B;
output: the shortest path distance from A to B
assumption: B is reachable from A

foreach i from 1 to number of nodes node x in G
    initialize distance estimate d(x) to infinity
initialize d(A) to 0

repeat until d(x) remains unchanged for all nodes x
    foreach edge e in allEdges(G)
        d(endNode(e)) := min(d(endNode(e)), d(startNode(e))+w(e))
    end
end

return d(B)
```

We could implement the algorithm in this form, but maybe there is a better way to terminate the relaxation?

[Can you find an upper bound on the number of times an edge has to be relaxed?](#)<sup>2</sup>

Based on this consideration we can simply apply a fixed limit on the number of edge relaxations. The resulting algorithm is known as Bellman-Ford's algorithm.

Algorithm Bellman-Ford

input: an edge weighted directed graph G; two nodes A, B;  
 output: the shortest path distance from A to B  
 assumption: B is reachable from A

```

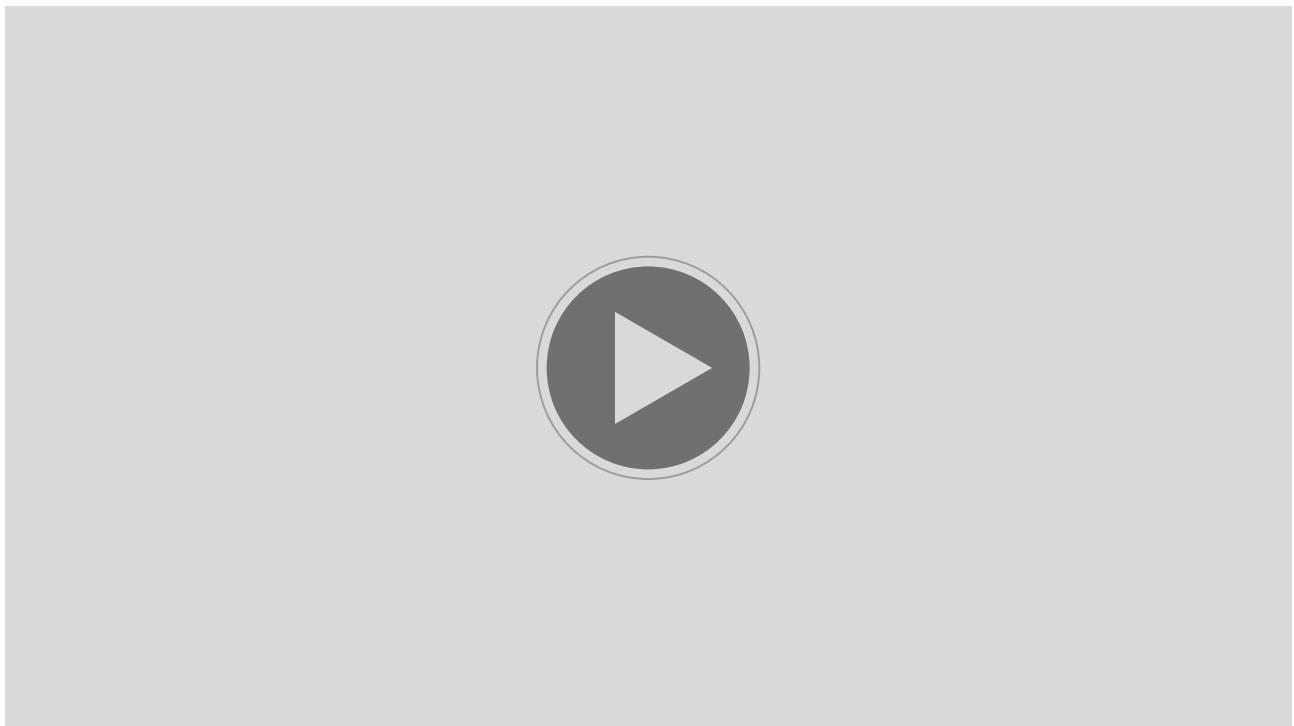
foreach i from 1 to number of nodes node x in G
    initialize distance estimate d(x) to infinity
initialize d(A) to 0

for i from 1 to length(allNodes(G))-1
    foreach edge e in allEdges(G)
        d(endNode(e)) := min(d(endNode(e)), d(startNode(e))+w(e)))
    end
end

return d(B)
```

Clearly, Bellman-Ford does more work than Dijkstra (it performs the same edge relaxations, but more often). We have to postpone a detailed quantitative analysis of how much slower until later, since we do not yet have the appropriate tools to analyze this.

The following video summarizes the function of Bellman-Ford's algorithm:

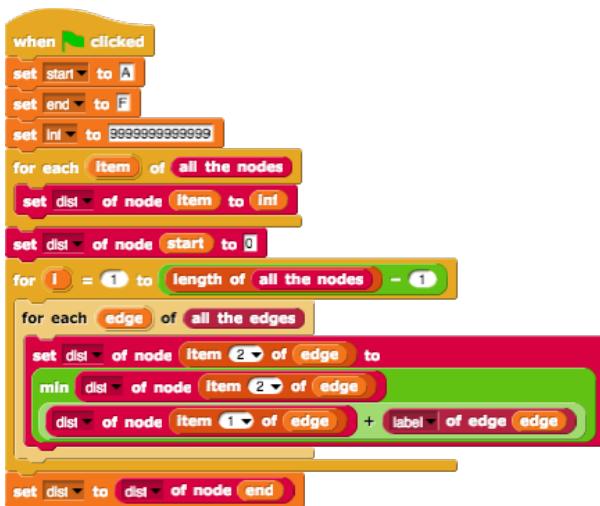


(<https://www.alexandriarepository.org/wp-content/uploads/20150408173936/Bellman-Ford.mp4>)

In a directed graph negative edge weights do not necessarily lead to negative weight cycles, but the fact that edges are directed by itself is no guarantee that there is no negative weight cycle. The final consideration thus is how to test for negative weight cycles.

[Can you find a simple test to check whether there are negative cycles that can be run after Bellman-Ford has been run?](#)<sup>3</sup>

Below is an Edgy implementation of Bellman-Ford's algorithm.



- ➊ If an undirected graph has a negative edge weight it automatically contains a cycle of negative length (consider the undirected edge A-B with negative weight. We can cycle around A-B-A-B accumulating more and more "negative length"). Thus the shortest path problem is ill defined because the possible path length is not bounded from below.
- ➋ Each edge needs to be considered at most  $(n-1)$  times where  $n$  is the number of nodes in the graph. This is for the following reason: After the  $i$ -th repetition of the outer loop, all paths of length  $i$  have been computed correctly. The longest (cycle-free) path contains at most  $(n-1)$  edges. Thus  $(n-1)$  repetitions of the outer loop all shortest paths have been computed correctly.
- ➌ All that needs to be done is to run one further round of edge relaxation. If the distance estimate  $d(x)$  for any node  $x$  can still be reduced there must be a negative weight cycle in the graph. This is because all cycle-free shortest paths (of length  $n-1$ ) had already been considered correctly before.

## 2.9.3

# Marshall's Transitive Closure Algorithm

### [draft for preview only; BM]

The algorithms considered so far compute the shortest paths between two given nodes or from a given node to all reachable nodes in a graph. A somewhat more complex problem is to compute the shortest distances between all possible pairs of nodes A, B in a given graph.

Of course, given that we have solutions to compute the shortest distance from A to all nodes in the graph, we have at least one way to do this already.

[Can you give a simple \(naive\) way to compute the distances between all possible pairs of nodes \(A, B\) in a given graph G? <sup>1</sup>](#)

We will return to this later. Under some additional assumptions on the structure of the graph, the naive solution from above can even be the best way to solve this task. But often it is not. There is a more direct way to solve this problem.

Before we consider the computation of all pairs shortest paths, let's tackle a simpler problem: the computation of reachability between all pairs of nodes (A, B). We simply want to answer the question whether B is reachable from A or not. This problem is known as the *transitive closure* of a graph.

The transitive closure of a graph is a graph which contains an edge between A and B whenever there is a directed path from A to B. In other words, to generate the transitive closure every path in the graph is directly added as an additional edge.

[Can you think of a naive way to construct a transitive closure? <sup>2</sup>](#)

But there is a more direct way to add the additional edges. We can simply iterate the process of concatenating two edges (either originally in G or already inserted earlier) by iterating over all possible combinations of start node, end node, and a middle node at which the two edges connect. In other words, we look for a combination of edges (A,C) and (C, B) and insert a new edge (A,B) wherever we find such a pair. The resulting algorithm is known as Marshall's transitive closure algorithm.

### Algorithm Marshall

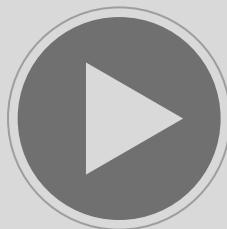
```

input: a graph G;
output: the transitive closure of G

foreach mid in allNodes(G)
    foreach start in allNodes(G)
        foreach end in allNodes(G)
            if not edgeExists(G, newEdge(start, end))
                if edgeExists(G, newEdge(start, mid)) and
                   edgeExists(G, newEdge(mid, end)) then
                       addEdge(G, newEdge(start, end))
            end (* of foreach end *)
        end (* of foreach start *)
    end (* of foreach mid *)
return G

```

The video below summarizes how Marshall's algorithm for the transitive closure works.



(<https://www.alexandriarepository.org/wp-content/uploads/20150408173958/Warshall.mp4>)

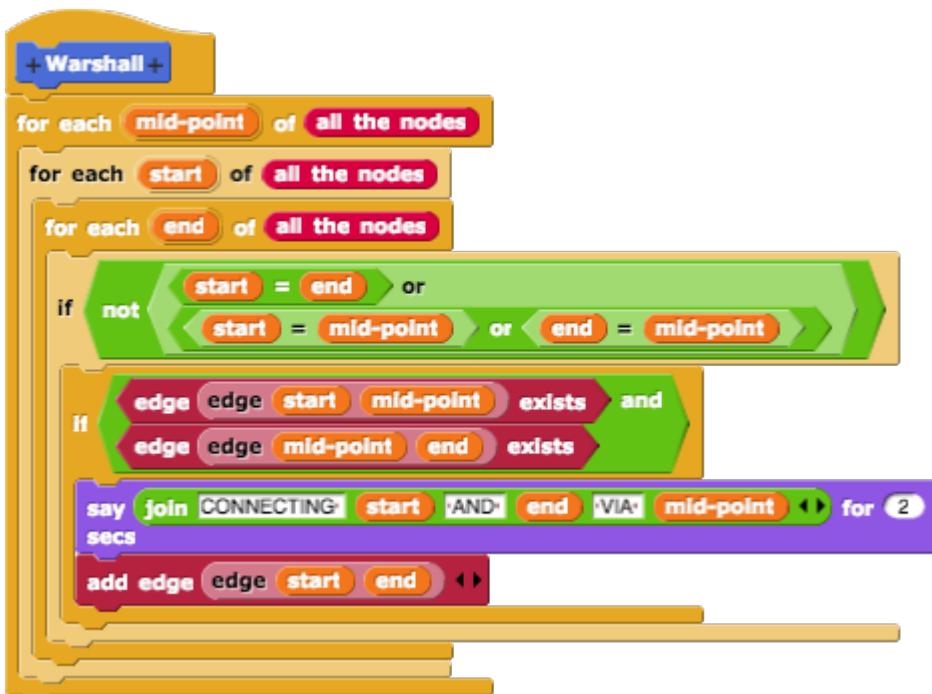
## Correctness of Marshall's algorithm

But how do we know that the algorithm has exhausted all options when it terminates? To see this use the fact that the nodes are ordered in some (arbitrary) sequence from  $i=1\dots n$ . This is, after all, the basis of the outer loop.

It is easy to see (by induction) that the  $k$ -th iteration of the outer loop establishes all paths that only uses nodes  $1\dots k$  as *intermediate* nodes (even though the start and end can be outside of the range  $1\dots k$ ).

This is obvious for the first iteration. At the  $(k+1)$ -st iteration all possible path from  $A$  to  $B$  that only use nodes  $1\dots k$  as intermediate nodes have already been established earlier (by induction). A new (cycle-free) path using  $(k+1)$  can only arise if there is an edge from  $A$  to  $(k+1)$  and one from  $(k+1)$  to  $B$  that can be joined at  $(k+1)$ . Each of these edges is either an original edge or arose from a path that only uses nodes  $1\dots k$ . By inserting edges from  $A$  to  $B$  for all such cases the  $k$ -th iteration establishes all possible paths that use only nodes  $1\dots k$  as intermediate nodes. Thus the algorithm correctly generates the transitive closure after  $n$  iterations of the outer loop, where  $n$  is the number of nodes in the graph.

Below is an Edgy implementation of Marshall's transitive closure algorithm.



- <sup>1</sup> Simply repeat the computation of Dijkstra (or Bellman-Ford) with each node of the graph as a start node.
- <sup>2</sup> One possibility would be simply to start a BFS from every possible node  $x$  in the graph, inserting edges from  $x$  to every node that is reached along the way.

## 2.9.4

# Floyd's Algorithm for All-Pair Shortest Paths

**[draft for preview only; BM]**

*Note: This module presumes knowledge of Marshall's transitive closure algorithm.*

The problem of computing the shortest path between all pairs of nodes in a graph is almost the same as computing the transitive closure. The extension that is needed is simply to keep track of whether a newly emerging connection between two nodes provides a shorter path or not. Marshall's transitive closure algorithm provides a good starting point. All that we need to do is to keep record of the current best distance estimate between  $d(X,Y)$  between nodes X and Y and to introduce a relaxation step in the innermost loop that checks whether a shorter connection has been found and that updates  $d(X,Y)$  accordingly.

The resulting algorithm is known as Floyd-Warshall's algorithm or simply as Floyd's algorithm.

**Algorithm Floyd-Warshall**

```

input: an edge-weighted graph G;
output: the shortest path distances of all pairs of nodes in G

foreach start in allNodes(G)
    foreach end in allNodes(G)
        set D(start, end) to infinity

foreach e in allEdges(G)
    set D(startNode(e), endNode(e)) := w(e)

foreach mid in allNodes(G)
    foreach start in allNodes(G)
        foreach end in allNodes(G)
            D(start, end) :=
                min(D(start, end), D(start, mid)+D(mid, end))
        end (* of foreach end *)
    end (* of foreach start *)
end (* of foreach mid *)

return D

```

The following video summarizes the algorithm and illustrates how it works.



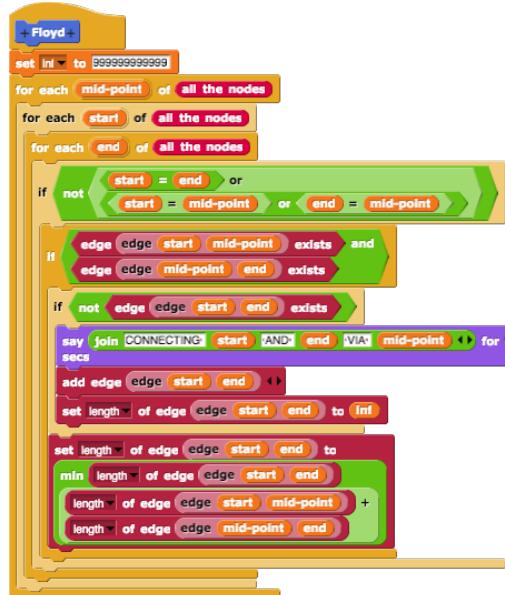
(<https://www.alexandriarepository.org/wp-content/uploads/20150408173947/Floyd-Warshall.mp4.mp4>)

Finally, we need to consider whether this Algorithm works in the presence of negative weights.

[Will the Floyd-Warshall algorithm run correctly if there are negative weights in the graph? <sup>1</sup>](#)

[Can you think of a simple way how the presence of negative weight cycles can be detected with this algorithm? Hint: this is best done after it has terminated. <sup>2</sup>](#)

Below is an implementation of Floyd's algorithm in Edgy.

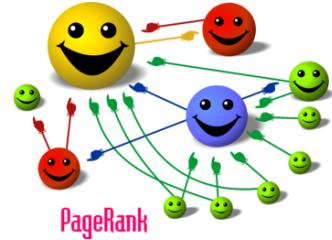


- <sup>1</sup> Yes, it will obviously terminate, since it has a fixed number of iterations. The argument applied for Warshall's algorithm still applies, i.e. all cycle free paths are considered. Furthermore, the relaxation step is in no way restricted to positive weights. In summary, the algorithm will work correctly with negative weights.
- <sup>2</sup> We simply need to check whether there is a negative entry  $d(X,X)$  for some node X. Since the algorithm considers all possible simple paths and correctly computes their length, there must be such an entry if a negative cycle exists.

## 2.10 PageRank

### Google PageRank

(<https://en.wikipedia.org/wiki/PageRank>)



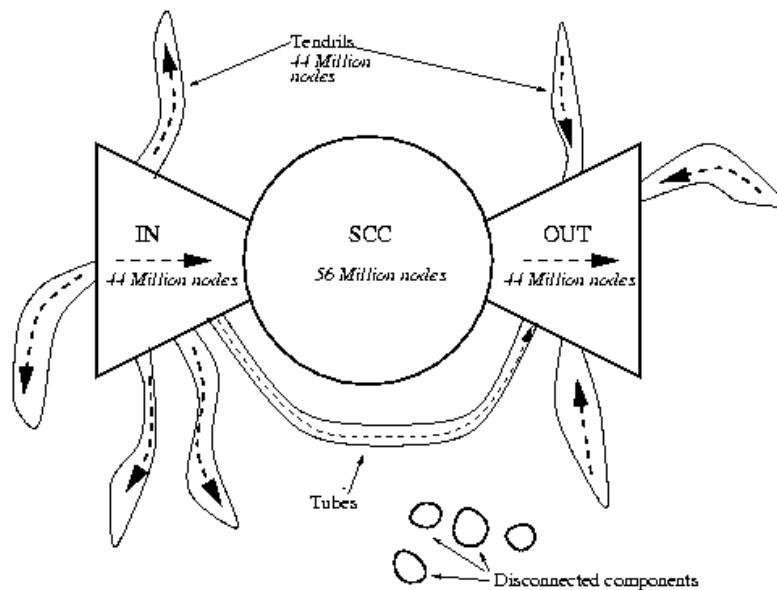
Have you ever wondered how web search engines deliver just the right results? It is one thing to quickly locate pages that contain a particular search term, but when there are millions of matching pages, how does a search engine know which one to put first?

The early success of Google as a search engine can be put down to the so-called PageRank Algorithm. The PageRank of a page is an estimate of the importance of that page. The intuition behind the algorithm is that if some page is linked from many important pages, then it is probably an important page too (note that this is a circular definition). As you will see, we can compute the PageRank of a set of pages iteratively, starting with an initial value for each page, and updating it several times until the PageRank values converge.

The resulting PageRank values amount to a probability distribution over the set of pages. The distribution corresponds to the likelihood that, starting on a random page and clicking random links to navigate the web for a sufficiently long period of time, we end up at the given page.

### The Web as a Graph

The Web can be considered as a graph in which the web pages are the nodes, and the hyperlinks are the (directed) edges. This so-called "Webgraph" has a complicated structure, organised around a central "strongly connected component" in which there exists a path in both directions between any pair of nodes.



Kumar et al (2000) *The Web as a Graph*.  
<http://cs.brown.edu/research/webagent/pods-2000.pdf>

## The PageRank Formula

At each stage of the iteration, we take the PageRank of a page  $PR(\text{page})$  and divide it by the number of outgoing links from that page  $L(\text{page})$  in order to calculate the contribution of that page to each of the pages it links to, i.e.  $PR(\text{page}) / L(\text{page})$ .

For a given page A, we sum these weighted PageRank values for the incoming pages B, C, ... Finally, we "dampen" the contribution of incoming links by some factor d (usually set to 0.85).

$$PR(A) = \frac{1-d}{N} + d \left( \frac{PR(B)}{L(B)} + \frac{PR(C)}{L(C)} + \dots \right)$$

Given the circularity of this definition, we get started by assigning the PageRank of every page to  $1/N$  (this is equivalent to setting  $d=0$  in the above formula).

What PageRank is assigned to a page with no incoming edges?

## PageRank and the Random Surfer (NEW)

The PageRank algorithm simulates the behaviour of a random web surfer, who starts at a random page (selected with uniform probability) and clicks a link at random to go to another page, and so on. Rather than getting stuck on a page that has no outgoing links, the surfer picks a new page at random. Thus, the PageRank value of a page is an estimate of the probability that the random surfer ends on that page.

It is easy to see that the initial PageRank values for the nodes in the graph sum to one, and constitute a uniform probability distribution over the nodes. After each iteration, we distribute the "probability mass" from a node to all of its outgoing nodes. No mass is lost in this process.

In subsequent iterations, we assign a new value to a node based on the following weighted sum:

$$(1 - d) \cdot A + d \cdot B$$

Since the values of A sum to one across all nodes, and the values of B sum to one across all nodes, their weighted sum does as well.

However, there is just one problem; if the graph contains one or more sink nodes, corresponding to pages with no outgoing links, this mass is lost to the system. It does not stay at the present node. (Why not?) So in order to be a well-behaved probability distribution, and faithfully simulate the behaviour of our random surfer, we need to add edges from any sink nodes to every other node, before running the algorithm.

## The PageRank Algorithm

In the following algorithm, we assume that nodes have a PageRank property called "PR", and we assume that we can access the incoming (resp. outgoing) nodes of some node using `node.incoming()` (resp. `node.outgoing()`).

```

N = length(all_the_nodes)
d = 0.85

foreach node:
    node.PR = 1 / N

foreach node:
    if length(node.outgoing()) == 0:
        foreach node2:
            add_edge(node, node2)

repeat:
    foreach node:
        total = 0
        foreach incoming in node.incoming():
            total += incoming.PR / length(incoming.outgoing())
        node.PR = (1 - d) / N + d * total
until convergence has been reached

```

## Implementing PageRank with Edgy

In order to better understand the PageRank algorithm, you might like to try implementing it with Edgy. You can use the PageRank of a page as the basis for setting the colour or diameter of each node (as you can see in the [Wikipedia entry for PageRank](#) (<https://en.wikipedia.org/wiki/PageRank>)).

Instead of testing for convergence, we recommend that you perform the outer loop a 10-20 times. You will be able to observe convergence thanks to your visualisation of the PageRank values, and adjust this figure.

NB. There is [another method for computing PageRank](#) (<https://en.wikipedia.org/wiki/PageRank#Algebraic>) that uses linear algebra, starting with the matrix representation of the webgraph.

## **2.11**

# **Algorithmic Complexity: How fast is my algorithm?**

### **First steps in algorithm analysis**

This video by Alan Dorin is a gentle introduction to algorithm complexity analysis. It explains in simple terms what is a meaningful notion of runtime for an algorithm.



(<https://www.alexandriarepository.org/wp-content/uploads/RunningTime-Wi-Fi-High.mp4>)

---

## **2.12 Recursion**

[2.12.1 What is recursion? A brief introduction](#)

[2.12.2 What is recursion: simple examples](#)

[2.12.3 Decrease and Conquer](#)

[2.12.4 How to draw a tree](#)

[2.12.5 Recursive tree search \(video\)](#)

[2.12.6 Recursive Graph Traversal by DFS](#)

[2.12.7 Analysing recursive algorithms](#)

---

## **2.12.1**

### **What is recursion? A brief introduction**

This video by Alan Dorin explains the general concept of recursion. It requires you to know about algorithms and specifically about functions (depending on what you have done before you may know functions as procedures, methods, or - in Edgy and SNAP - user defined blocks.)



---

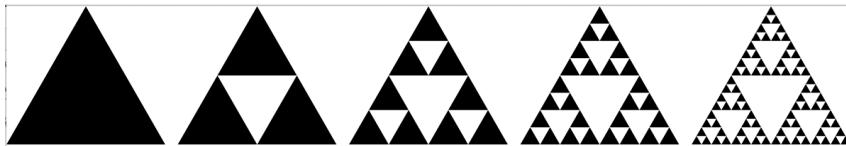
(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-recursion-1-Wi-Fi-High.mp4>)

---

## 2.12.2

# What is recursion: simple examples

Consider the following series of diagrams, starting with a triangle on the left. At each step, we make three copies of the diagram on the left, shrink them down, and arrange the three into a triangle again, ad infinitum.



You will have experienced recursion if you ever stood between two almost parallel mirrors, or shared your computer screen during video chat, and seen an infinite series of images of yourself.

Activity: view the following videos.

- [merge sort](https://www.youtube.com/watch?v=EeQ8pwjQxTM) (<https://www.youtube.com/watch?v=EeQ8pwjQxTM>)
- [triomino tiling](https://www.youtube.com/watch?v=kVq9QJA36tl) (<https://www.youtube.com/watch?v=kVq9QJA36tl>)

Once you've done this, write down what it means to solve an instance of these problems of size  $2^0$  and  $2^1$ . Now explain how the solution to the problem of size  $2^n$  is related to problems of size  $2^{n-1}$ . Try to write down a concise explanation of recursion.

## Some examples in Edgy

In the Edgy file menu, open the examples folder, and load the Sierpinski example. Try it out for some small values of the size parameter (up to 5). Inspect the contents of the sierpinski block, and observe the following structure:

```
function sierpinski(a, b, c, depth):
    if depth > 1:
        sierpinski(? , ?, ?, depth-1)
        sierpinski(? , ?, ?, depth-1)
        sierpinski(? , ?, ?, depth-1)
    else:
        add edges
```

Suppose we started by invoking the sierpinski function with depth=3. It would invoke the same function three times with depth=2. For each of these, it would call sierpinski three times with depth=1 (a total of 9), and this would finally call sierpinski three times with depth=0, which would add the edges.

You might like to add a "wait 1 secs" block to the top of the definition of the sierpinski block, and watch the Sierpinski Triangle take shape in stages.

Now load the Towers of Hanoi example, and again try it out for some small values of the size parameter. Inspect the "Solve Hanoi" block, and observe the following structure:

```

function hanoi(n, start, spare, end):
    if n > 1:
        hanoi(n-1, ?, ?, ?)
        move a piece
        hanoi(n-1, ?, ?, ?)
    else:
        move a piece

```

Compare the two blocks of pseudocode above, to see what they have in common. Review your earlier written explanation of recursion in light of your new understanding of recursion.

## Recursion vs Iteration

Recursion offers a way to perform the same task repeatedly. In this respect, it is similar to iteration. Consider the case of factorial. Here is the iterative version:

```

function factorial(n):
    product = 1
    for i = 1 to n
        product = product * i
    return product

```

Let's use our evolving schema for a recursive function (see the previous section), to write a template for the recursive factorial function:

```

function factorial(n):
    if n > 1:
        do some operations involving factorial(n-1)
    else:
        handle the case where n=1

```

Flesh this out into a complete definition of factorial, and satisfy yourself that it is correct. Now study the iterative and recursive versions. Are both of these "effective methods expressed as a finite list of well-defined instructions" for calculating factorial?

## Infinite Recursion

Looking back at all the examples of recursion we have seen, observe that there is always a depth bound. Our function has a parameter, and this is reduced by one for the recursive call(s):

```

function f(n, ?, ?):
    if n > 1:
        do some operations involving f(n-1, ?, ?)
    else:
        handle the case where n=1

```

In most programming languages, it is possible to get an error "maximum recursion depth exceeded". For example, this happens in Python after 1000 recursive calls:

```

>>> def factorial(n):
...     if n == 1:

```

(This is an arbitrary limit, set by the programming language, to help programmers avoid errors with runaway recursion. It is possible to raise this limit, but ultimately, we will run out of computer memory. In such cases, it is sometimes necessary to translate a recursive function into an iterative version.)

This depth-bounded recursion is different to the infinite recursion you experience when standing between a pair of (almost) parallel mirrors. However, both kinds are still known as recursion.

## 2.12.3

# Decrease and Conquer

We just saw a simple example of the recursive template:

```
function factorial(n):
    if n > 1:
        do some operations involving factorial(n-1)
    else:
        handle the case where n=1
```

Let's consider a particular instance, for 6!, and expand out the steps of the computation:

```
factorial(6)
-> 6 * factorial(5)
-> 5 * factorial(4)
-> 4 * factorial(3)
-> 3 * factorial(2)
-> 2 * factorial(1)
-> 1
-> 2
-> 6
-> 24
-> 120
-> 720
```

Notice how our method for computing the factorial of  $n$  involves reducing the task to a slightly smaller task of computing the factorial of  $n-1$ . This is a recognised algorithm design strategy called Decrease-and-Conquer. The most famous example of this strategy is binary search.

## Binary Search

Suppose you want to look up a word in a printed dictionary. Instead of starting at page 1, you would be more likely to dive into the middle somewhere. So if you were looking up *synecdoche* you might first land on a page containing words that start with the letter *n*. Since *s* comes after *n*, you would dive into the middle of the second half of the dictionary, and so on. Let's formalise the process using pseudocode:

```
function binary_search(word, start, end):
    if word is found at the start position:
        access the definition
    else if end < start:
        report word not found
    else:
        mid = start + (end - start) / 2
        if word && the word found at the start position:
            binary_search(word, start, mid-1)
        else:
            binary_search(word, mid, end)
```

Suppose our dictionary contained 10,000 entries. Then we might go through the following steps:

```
binary_search("synecdoche", 1, 10000)
-> binary_search("synecdoche", 5000, 10000)
  -> binary_search("synecdoche", 7500, 10000)
    -> binary_search("synecdoche", 7500, 8749)
      -> ...
```

See how each step of the process results in a smaller instance of the problem?

Decrease-and-Conquer is similar to its more famous cousin Divide-and-Conquer (to be covered later). Both involve solving a problem through solving smaller instances of the same problem. However, for Decrease-and-Conquer we only create a *single instance* of the smaller problem. It could be half the size of the original problem (as in binary search) or one less than the size of the original problem (as in factorial above).

## Topological Sort

TO BE WRITTEN

## Tail Recursion (extension)

There is a special case of recursion, known as tail recursion, which is another example of the Decrease-and-Conquer design strategy. For details, please see the extension materials in the Appendix.

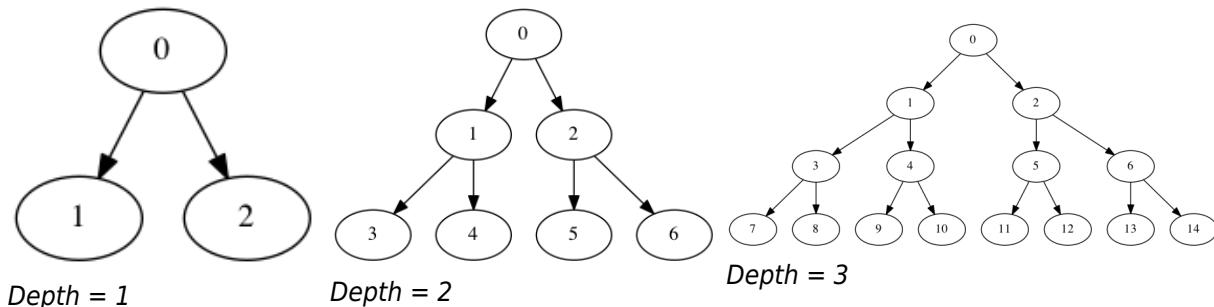
---

## 2.12.4

# How to draw a tree

### Trees and recursion

Trees offer us an easy way to understand recursion. Consider the following trees.

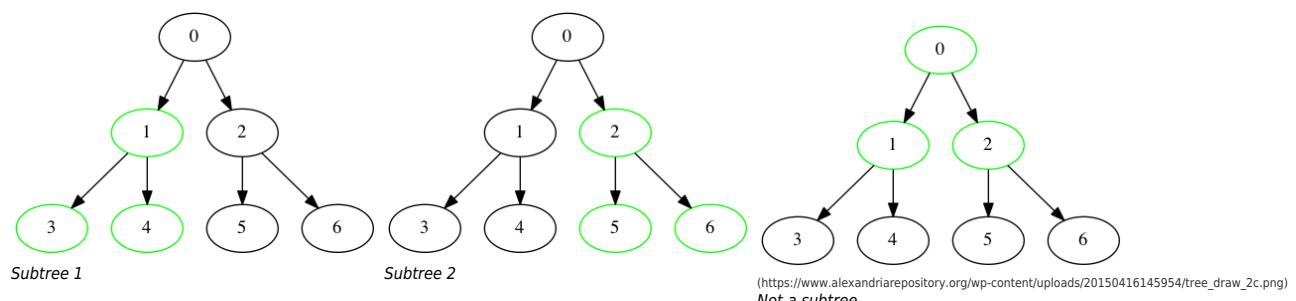


Can you think of an iterative method for generating these trees? Try to do this on your own before consulting the solution below. When you're ready, swap your work with a partner and see if you are both convinced that it is correct.

Here's a possible solution. Why is it called iterative? What do you think about the efficiency and readability of this solution?

```
depth = 3
root = new_node()
while depth > 0:
    initialise leaves to the empty list
    foreach node:
        if node has no children:
            add it to the list of leaf nodes
    foreach leaf node:
        create two new nodes
        link them to the leaf node
    depth = depth - 1
```

Consider the above tree diagrams again. Notice that a tree of depth  $d$  contains two trees of depth  $d-1$ . Let's look more closely at the depth 2 tree, and identify the two depth 1 subtrees:



Note that the green nodes 0, 1, 2 in the third tree above do not form a subtree of this tree, since nodes 1

and 2 are not leaf nodes.

## Generating trees recursively

Now we are ready to try to generate trees recursively. The recursive insight is that to draw a tree of depth  $d$ , we need to draw two trees of depth  $d-1$ , and link them up somehow.

Without any further thought than this, we can apply our general purpose recursion schema to write down a template for a `draw_tree` function as follows.

```
function draw_tree(node, depth):
    if depth > 0:
        one or more operations involving draw_tree(?, depth-1)
    else:
        what to do when depth=0, if anything
```

Now it's up to you to work out how to fill this out into a complete algorithm. Start by asking yourself how many times you need to call `draw_tree()`. It's going to be called on a specific node (its first parameter). Where will that node come from? How will you join everything up.

What do you have to do when drawing a tree of depth 0, if you've been given the root node for that tree?

Note that there will need to be an outer program that calls this function, as follows:

- `new_digraph()`
- `root = new_node()`
- `depth = 3`
- `draw_tree(root, depth)`

## Generalising for trees of arbitrary degree

The above trees have all been binary.

The final step is to lift this restriction, to generate trees of arbitrary degree  $> 0$ . Modify the above algorithm accordingly.

Finally, you might like to implement your algorithm using Edgy, and compare your work with the built-in block:

generate balanced tree of degree 2 and height 3 numbered from 0

## Depth-bounded recursion

Notice that we have used the depth parameter to limit how much recursion we do. This was also done in the code for generating Sierpinski Triangles. Why is this necessary?



## **2.12.5**

### **Recursive tree search (video)**

This video by Alan Dorin explains how to apply recursion to search in a tree.



(<https://www.alexandriarepository.org/wp-content/uploads/FIT1042-recursion-2-tree-dfs-Wi-Fi-High.mp4>)

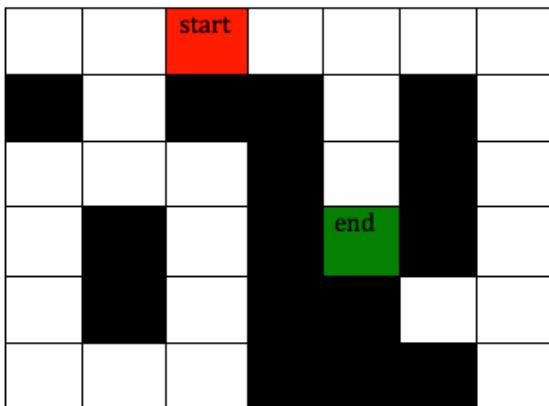
---

## 2.12.6 Recursive Graph Traversal by DFS

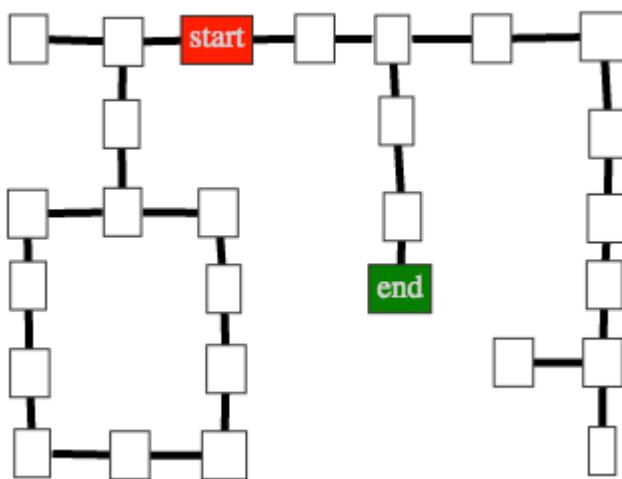
We're going to return the problem of finding a path between two nodes in a graph. In previous modules we looked at two algorithms to solve this problem: breadth-first-search and depth-first search. It turns out that we can write a particularly elegant version of depth-first search (DFS) using recursion which can then be easily extended to yield the nodes that form the actual path.

### Solving a Maze using Recursion

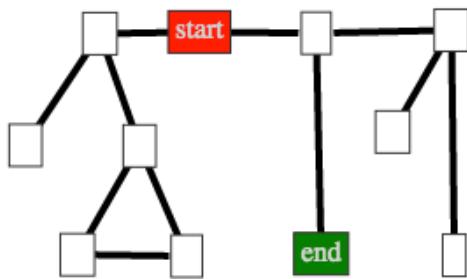
So, let's look at the problem of finding a path through a maze. Imagine we want to find a path between the "start" and "end" cells of the maze shown below, where the white cells are part of the path and black ones are blocked with an obstacle.



We can represent this maze as a graph, where each node corresponds to a cell in the maze, and the edges correspond to paths between these points (i.e. where there is no obstacle between cells).



We can simplify the graph representation above, to represent only the choice points or ends of paths in the maze. This gives the graph below:



## The Depth-First-Search Algorithm

Now consider the problem of finding a path between the "start" and "end" nodes in the graph above. We have seen that we can use a graph traversal algorithm to search our graph starting from the "start" node until we either reach the "end" node or fail to find a path to it. We're going to use the DFS (depth-first-search) algorithm that we discussed previously, but this time we're going to write a recursive version of this algorithm.

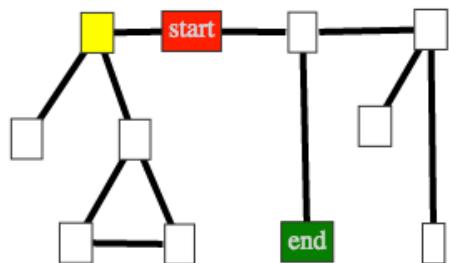
## Writing the Algorithm using Recursion

So, what might a recursive version of the DFS algorithm look like? Can you use the recursion schema from the previous module to write a template for the algorithm? The function will need to be given a start node and an end node, and will need to call itself recursively.

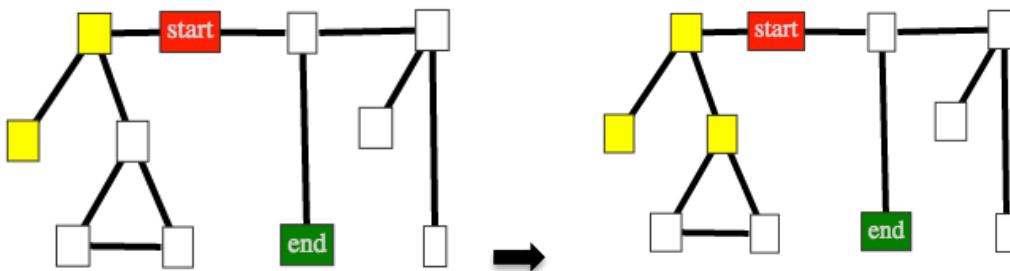
### A First Attempt

The recursive DFS algorithm will start at the "start" node, then check one of its neighbours,  $n$ , to see if it is the "end" node. If it is not, then we repeat the DFS algorithm to check whether a path exists between node  $n$  and the "end" node. Where a path ends without reaching the "end" node, the algorithm backs up to the previous node in the path and checks another neighbour instead.

In our example maze, we first choose a neighbour of the start node (coloured yellow as in the graph below). Then we check recursively using DFS if there is a path from this neighbour to the end node.



If a node is the end of a path, we backtrack to the previous node and check for a path from a different neighbour, and so on.



When we have checked all the neighbours of a node, we back up to the previous node, to check its neighbours.

Let's use our recursion schema from the previous module, and write down a template for our recursive depth-first-search function:

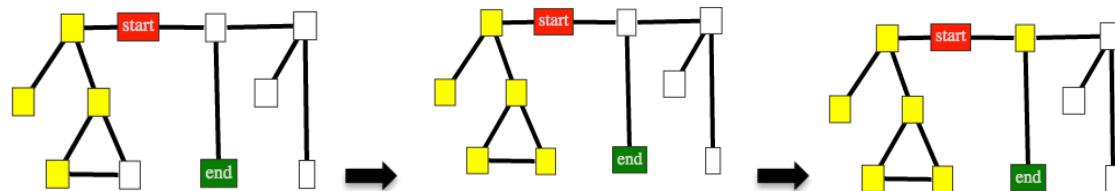
```
function depth_first_search(start_node, end_node):
    if (start_node != end_node):
        do depth_first_search on neighbours of start_node that haven't already
        been visited
        if a path is found:
            return true
    else:
        return true
    return false
```

Now we have defined the basic structure of our algorithm. First, a recursive call, if we have not found the "end" node yet. And then, a termination case (or base case) that returns "true" when we have reached the "end" node. The algorithm will also terminate and return "false", if the "end" node has not been found and there are no more new paths to search.

## Refining the Algorithm

However, it is also necessary to keep track of which nodes have already been visited in order to avoid:

- re-checking the previous node in the path (which is a neighbour of the current node in an undirected graph).
- checking neighbouring nodes repeatedly where a graph has a cycle of connected nodes. For example, in the graph below, the final white node in the (triangle-shaped) cycle, will be checked only once, despite being the neighbour to two yellow (visited) nodes. By marking nodes that have already been visited (e.g. using colour as in the graphs below) you can avoid checking the same node repeatedly.



Refine the pseudo-code above to give a complete definition of the DFS algorithm. Note that you will need to mark the nodes once they have been visited, in order to avoid repeatedly following the same paths through the maze. You will also need to check whether a node has been visited before searching for a

path from it to the "end" node.

## A Recursive DFS Algorithm

Flesh out the pseudo-code above into a complete definition of the DFS algorithm.

Satisfy yourself that it is correct by working through your algorithm using the example graph. You might like to compare it with the iterative version of DFS from previous modules.

Once you have done this, modify your algorithm to colour "visited" nodes and nodes that form part of the final path different colours. For example, we could mark visited nodes yellow, and nodes on the final path to green. To do this, mark the nodes that are currently being visited by the DFS algorithm as green. Then, once the algorithm has finished visiting them, and if it has not found a path, mark them yellow.

## Getting the Path

To make our search for the path from "start\_node" to "end\_node" useful, can you modify your algorithm to produce the actual path, i.e. the sequence of nodes we have to take to get from "start\_node" to "end\_node"? Here are some hints:-

- use a local variable to build up a list of the nodes on the path.
- instead of returning true or false, your algorithm will need to return a list of nodes on the path.
- you can add nodes to the path list each time a recursive call returns successfully.

You will notice that the DFS algorithm does not necessarily return the shortest path between two nodes. Another algorithm would be required for this - do you know which one?

---

## 2.12.7

# Analysing recursive algorithms

Once we've written a recursive algorithm, how can we work out how efficient it is? When deciding between various algorithms for solving a particular problem, it is useful to be able to compare the performance of different solutions. For small problems, efficiency usually doesn't matter much. However, it's important to know what happens as the size of the problem increases. In this module we will see how to analyse the efficiency of recursive algorithms.

Recall from module 3.9 that we can analyse the time complexity of an algorithm by counting the number of times it performs some basic operation. The basic operation differs, depending on the algorithm, it might be multiplication, appending to a list, or creating a node. For iterative algorithms, we can look at the number of times the loop (or loops) are executed, and work out how many times the basic operation is performed.

```
for i = 1 to n:
    for j = 1 to n:
        perform some operation
```

In the above case, simple inspection tells us that the operation will be performed  $n^2$  times. This approach won't work for recursive algorithms. (Why not?)

### Example: Generating a binary tree

Consider the algorithm for generating a binary tree of a specified depth, discussed in module 3.10.3:

```
function draw_tree(root_node, depth):
    if depth > 0:
        left_child = new_node()
        create an edge from root_node to left_child
        right_child = new_node()
        create an edge from root_node to right_child
        draw_tree(left_child, depth-1)
        draw_tree(right_child, depth-1)

root_node = new_node()
draw_tree(root_node, 4)
```

We'd like to modify this algorithm to count the number of "steps" that it takes to generate the binary tree. What's the basic operation? Let's opt for node creation, and add a counter to keep track of the number of times this operation is performed:

```
function draw_tree(root_node, depth):
    if depth > 0:
        left_child = new_node()
        create an edge from root_node to left_child
        step_counter = step_counter + 1
        right_child = new_node()
        create an edge from root_node to right_child
        step_counter = step_counter + 1
```

```

draw_tree(left_child, depth-1)
draw_tree(right_child, depth-1)

set step_counter = 0
root_node = new_node()
step_counter = step_counter + 1
draw_tree(root_node, 4)

```

Now, let's see what happens to the value of the step counter:

- depth=0: we create a root node (1 step)
- depth=1: we create the root node (1 step), then add left and right child nodes to our tree (2 more steps)
- depth=2: we create the root node (1 step), and then add left and right child nodes to our tree (2 more steps), then call draw\_tree for each of these 2 child nodes resulting in 4 further leaf nodes (2 x 2 more steps)
- depth=3: we create the tree of depth 2 as above, and then add 2 child nodes for each of the 4 leaf nodes

Let's tabulate these values:

depth 0:	1 step
depth 1:	$1+2 = 3$ steps
depth 2:	$1+2(1+2) = 7$ steps
depth 3:	$1+2(1+2(1+2)) = 15$ steps

Can you see the pattern? What is the relationship between the depth of the tree and the number of steps that the algorithm executes? Is there way to express the number of steps as a function of the depth?

## Example: Summing Numbers

Now, let's look at the algorithm for summing the numbers from 1 to n (cf 3.10.6). Here's the iterative version:

```

function sum_iter(n):
    set total = 0
    for i = 1 to n:
        total = total + i
    return total

```

You can see that it loops through values of i, from its initial value of 1 to its final value of n. Can you analyse its time complexity by counting how many times the loop is executed? Remember, time complexity is often expressed relative to the size of the input.

Now, let's look at the corresponding tail-recursive version of this function, with an added variable "step\_count", to count the number of times that the basic operation (addition) is performed.

```

set step_count = 0

function sum_tail_rec(n, running_total):
    if n != 0:
        step_count = step_count + 1
        return sum_tail_rec(n - 1, running_total + n)

```

```

else:
    return running_total

sum_tail_rec(20, 0)

```

Can you see the relationship between the value of the input  $n$  and `step_count`? Work through some examples with different input values, and see the number of steps that the algorithm takes.

If we run this algorithm with a number of different inputs, we get the following results:

x	step_count
1	1
2	2
3	3
4	4
5	5
6	6

How can we express the relationship between  $n$  and `step_count`? Well, we can say that the time complexity of '`sum_tail_rec`' is directly proportional to the size of the input  $n$ , or, that our function performs  $n$  basic operations.

## Deciding what to count

You might have noticed that we made some arbitrary choices in what to count in the recursive examples above. When creating the binary tree, we counted the number of node creations (not edge creations), which was one less. And for the addition function, we counted the addition of each new value to the running total, and not the decrementing of  $n$ .

If we had included this extra work in our calculations, the total value of the counts would have been increased by a factor of 2. (You might like to check this for yourself.) However, when measuring the complexity of an algorithm, we are not concerned with constant factors. We just want to identify the growth rate, as something like  $n$ ,  $n^2$ ,  $2^n$  or whatever, since this rate is more significant than the constant factor, as  $n$  gets large.

## Experimenting in Edgy

Now that we've seen how to calculate the time complexity for recursive functions, let's do some more systematic experiments with different input values. We're going to look at the Sierpinski Triangle example in Edgy. In the Edgy file menu, open the examples folder, and load the Sierpinski example.

Your task is to identify the "basic operation", and then add a `step_counter` variable to count the number of times the basic operation is performed.

This time, we're not going to inspect the code in any detail. Instead, we're going to treat this function as a black box and simply run it with different input values to log the number of steps that it takes for each. We want to systematically study the behaviour of the function as the input size changes. Create a block of code in Edgy based on the pseudocode below to run Sierpinski with values from 1 to 5, and calculate the

number of steps it takes to run.

```
results = []
for size = 1 .. 5:
    step_counter = 0
    sierpinski(new_node(), new_node(), new_node(), size)
    results.append(step_counter)
```

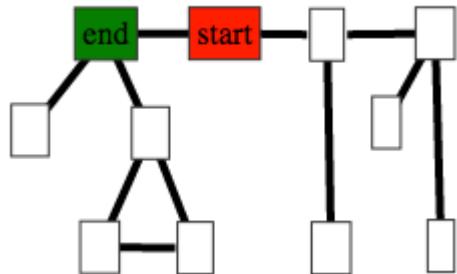
Run your block and examine the values in the 'results' list. Can you see a relationship between the input size and the number of basic operations that the Sierpinski function performs? Observe the rate of growth of the 'step\_count' as the input size gets larger.

Using your programming skills, you have written code to collate data about the performance of other code. This can help you to analyse algorithms and understand their performance.

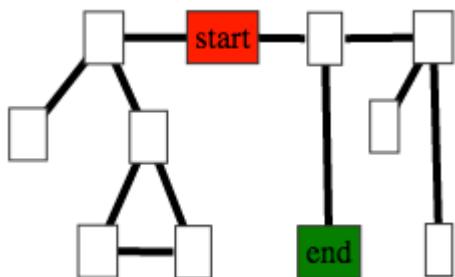
## The Effect of Input

Some algorithms do a different amount of work depending on the order or structure of their input. For these algorithms, the time complexity is not only affected by the size of the input, but also other conditions under which the algorithm is run. For example, the performance of the DFS algorithm for finding a path between a start and an end point in a given maze, will depend on which start node we have chosen, the structure of our maze (or graph) and which end node we are trying to reach.

If we search the children of a node from left to right, then an end node that is left-most neighbour of the start node will be found immediately.



In contrast, an end node that is towards the bottom of a branch will take much longer to find using the same DFS algorithm.



For algorithms such as DFS, in which the structure of the input or the particular problem affect the time to solve it, we can talk about best-case performance, worst-case performance and average-case performance.

For the DFS algorithm in a maze, which we can represent as a graph, the best case would be when the end node is the first neighbour of the start node. In this case it only requires 1 recursive step to reach the end node, so its time complexity is constant time (1 step). The worst case for DFS in a maze is if the end node is at the end of the last path to be searched. So, the worst case time complexity is proportional to the size of the number of nodes in the graph, or  $n$ .

What do these two cases tell us about the performance of the DFS algorithm? It seems that neither is very representative of the maze problem, so we'd really like to work out the average case time performance. How could we do this? Well, one way would be to run lots of trials of the algorithm, just as we did above, to see the effect of the size of the input. Our trials , however, would need to run our algorithm on representative maze structures and start and end points.

---

## 2.13 Best First Search

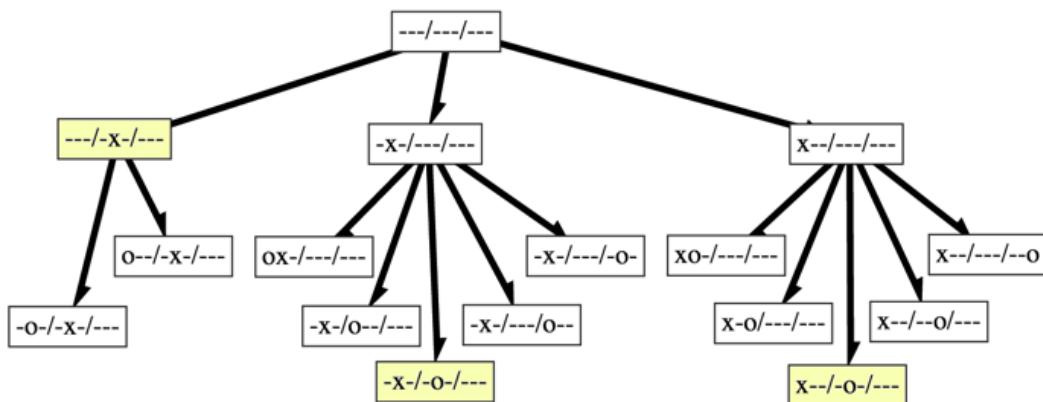
### Heuristic Search

When solving real-world problems, we often rely on simple "rules of thumb". For example, if we needed to cross town on foot and didn't have a map, we would probably be inclined to set off in the general direction of our goal. In some cases that choice might not be optimal, e.g. we might be in a dead-end street; the quickest route might involve starting off in the opposite direction.

Consider the game of tic-tac-toe. Do you have any rules of thumb about where to put your X or O pieces? Obviously, if you can win on the next move, you move accordingly. But what if you can't see a win right away? And what about the opening move? Try to identify a couple of these rules of thumb before continuing. You might like to do the same for other board games, like checkers or chess. E.g. in chess it's good to protect your pieces, rather than leave them undefended, even if they're not in any immediate danger.

In computer science, such rules of thumb are known as *heuristics*.

A possible heuristic in tic-tac-toe is to claim the centre square if possible. In terms of a game tree, where each node represents a configuration of the board, that means favouring the highlighted nodes:



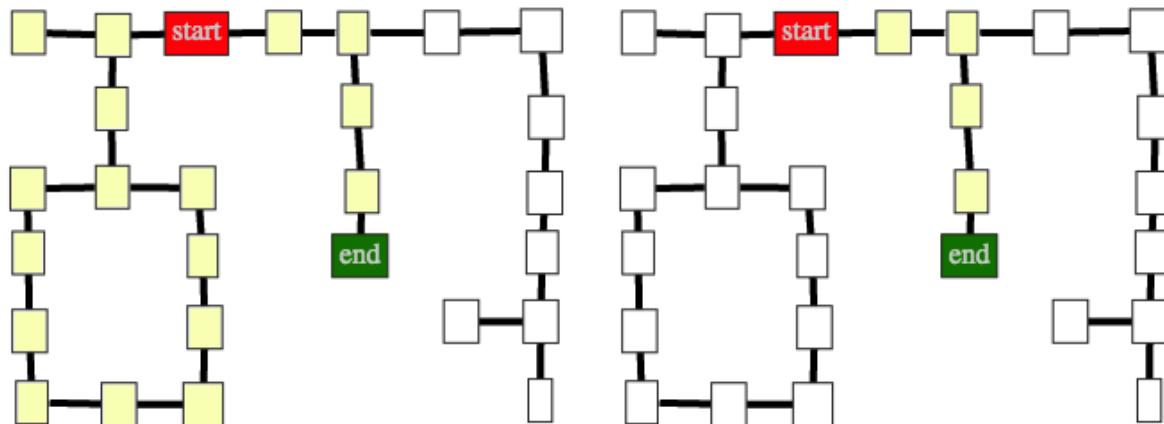
If the center square is taken, we might favour corner squares over side squares, other things being equal. In this way, whatever node we might be at in our game tree, we have some way to prioritise the available moves.

Now, if we are writing a program to analyse all options and find the best one, there is no prospect for saving time. We'll need to visit the entire subtree. (Why?) The heuristic is only useful if we think we can make a good decision without exhaustive search, without visiting the entire tree. Our goal is to improve the average speed of our algorithm.

### An example: Searching in a Maze

Imagine that you are wandering around a hedge maze, trying to find the way out. By standing on your

toes, you're able to look over the walls and see a flag that marks the exit. Now, suppose you come to an intersection in the maze, and one of the options will take you in the direction of the flag. That seems to be a good option to explore first, even though it's not guaranteed to be the right one. If we applied this heuristic to our previous maze example, we would make fewer wrong turns and find the solution much more quickly, as shown in the highlighted graphs below (yellow is used to mark visited nodes).



We can write down the heuristic as follows (shown in green):

```
function best_first_search(start_node, end_node):
    if (start_node != end_node):
        work out the compass direction
            from start_node to end_node
        sort the unvisited neighbours of start_node
            for similarity with that direction
        foreach unvisited neighbour:
            if best_first_search(neighbour, end_node):
                return true
    else:
        return true
    return false
```

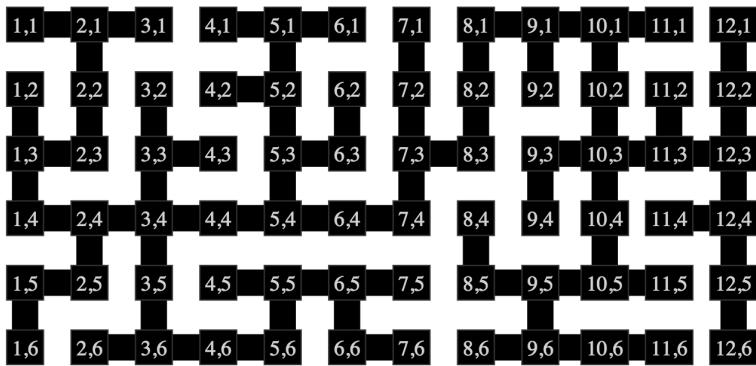
Examine the above pseudocode carefully. If the best choice at a given node turns out to be wrong, will the best-first search algorithm still find a solution? In other words, is best-first search guaranteed to find a solution if one exists?

How much time does this save on average? Answering this question would require some experimentation, in which we run a series of trials (see the next section).

## Implement best-first search for the maze problem

**Step 1:** Load the [generate maze block](#)

(<https://www.alexandriarepository.org/wp-content/uploads/20150510123424/generate-maze.xml>) into Edgy. You can find it at the bottom of the network tab. Set the layout to manual before running this block. It will generate  $m \times n$  mazes like the following, where the available paths are black (here  $m=12$  and  $n=6$ ).



(Can you figure out how the maze generation block works? You might like to read about the [origin of this algorithm](#) ([http://en.wikipedia.org/wiki/Maze\\_generation\\_algorithm#Randomized\\_Prim.27s\\_algorithm](http://en.wikipedia.org/wiki/Maze_generation_algorithm#Randomized_Prim.27s_algorithm)).

**Step 2:** Specify the start and end nodes. This could be as simple as 1,1 and m,n.

**Step 3:** Apply depth-first search to this graph, marking visited nodes, until the end is found. Then count how many nodes were visited. This number is the result of the run.

**Step 4:** Apply best-first search to this graph, again marking the visited nodes, and counting how many nodes were visited.

**Step 5:** Run a series of trials, and work out the average-case complexity for each algorithm when applied to an n\*n maze.

Note: for step 4, you'll need to work out how to score any node, in terms of how promising it is as part of the path to the end node. One simple way would be to calculate the Hamming distance from the node to the end node:  $\text{hamming}(a, b) = \text{abs}(a.x - b.x) + \text{abs}(a.y - b.y)$ . Then you will need to order the candidate nodes according to their score.

## Designing the heuristic

An important feature of a heuristic is that it is quick to calculate. Typically, a heuristic would only use locally-available information.

For example, the Hamming distance score was calculated based only on the child nodes of the current node and their distance from the end node. In theory, we could have performed complete look-ahead, searching the whole maze to work out the correct node to pick at any stage. But this would be a strange thing to do: we effectively solve the maze in order to take the first move, then solve it again to take the next move, and so on. The problem with such a heuristic is that it repeatedly visits the whole graph. If we're trying to save effort and not search the whole graph in order to find the best path, there's no sense in visiting the whole graph in order to make every single move!

## Best first vs greedy

Best-first search is related to greedy search. Recall Prim's algorithm, and how, at each step, it extends a partial minimal spanning tree by linking the nearest unconnected node, successively growing a partial solution into a complete solution. This is a greedy design pattern, which happens to find the optimal

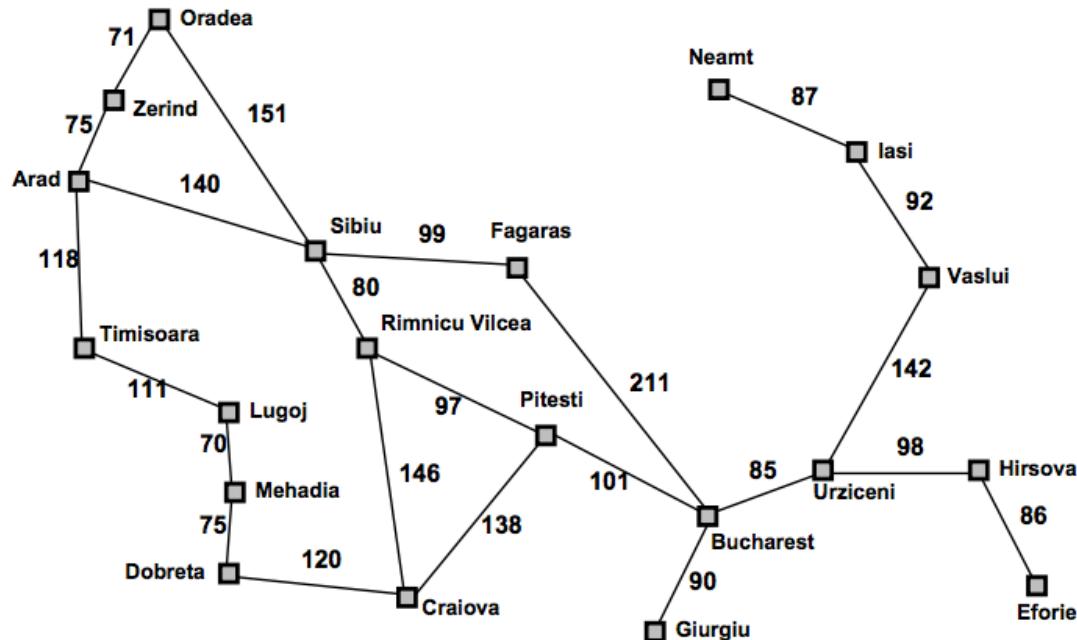
solution in the case of Prim's algorithm.

Imagine we applied the greedy strategy to solving the maze. This means that we would always commit to taking the move that brings us closer to the end. We would never be able to back up and try other branches.

Best-first search is different. It is guided by our greedy heuristic, but it can always back out and consider other alternatives. Eventually, it will consider all possibilities, just as depth-first and breadth-first search do.

## Example: Route Finding in Romania

Imagine you need to get from point A to point B in a road network. Just for fun, suppose you were in Romania, and wanted to travel from Arad to Bucharest. You have a map showing which cities have roads connecting them and the distances of these roads, as shown below.



Note that this is a map, not just a graph, given that the cities are plotted in a way that corresponds to their geographical location.

Think for a moment how you would go about the task of getting from Arad to Bucharest. You're currently at Arad. Do you dig out your implementation of Dijkstra's algorithm and transcribe the map into a graph? No, your intuition is probably to consider the physical layout, and go to a neighbouring city that is closer to Bucharest, or in the same direction as Bucharest. How can we exploit ideas like this?

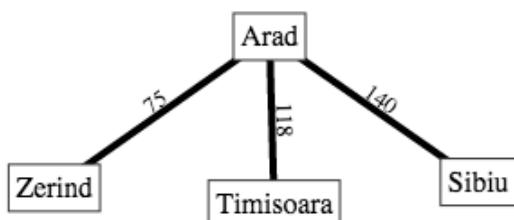
Now, observe that the map isn't perfectly to scale: there's a 99km edge (Sibiu-Faragash) that is more than 25% longer than the nearby 80km edge (Sibiu-Rimnicu Vilcea). So perhaps the cities are not accurately mapped, and we might not make a good choice as we set out from Arad.

Let's think about this another way. Suppose you could redraw the above map so that edges are to scale,

but so that the cities are moved around. For instance we could rotate the Arad-Oradea-Sibiu triangle 90° anti-clockwise, and stretch Lugoj and Mehadia out westwards, while making sure that the Arad-Timisoara edge pointed directly towards Bucharest. By playing around with the layout like this, a human reader of our map might make different choices about the first step to take from Arad. All of this serves to demonstrate that there is more going on in the above diagram than just a collection of nodes and edges.

## Getting started

How would you go about finding a route? If you were to choose which road to take from Arad, how would you decide which one? Dijkstra's algorithm feels like a fair amount of work. What about the brute-force approach to finding the optimal path, by looking at every possible route and calculating the shortest one? Let's try to formulate a heuristic, basing our decision on locally-available information. Starting at Arad, there are three roads we could possibly take:



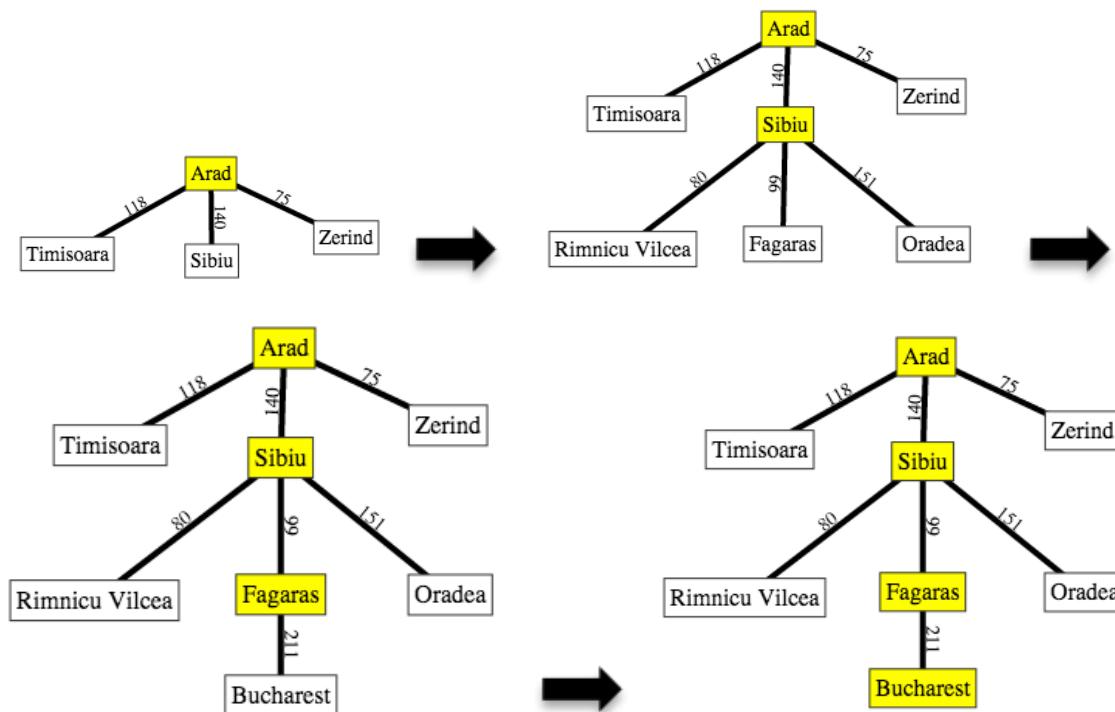
How could we choose between them?

One possible heuristic is to choose the neighbouring city that is closest to Bucharest. That is, we could estimate the distance of the shortest path to Bucharest, using the straight-line distance from each of these cities to Bucharest. This would be possible if we had a map that is to scale, allowing us to measure the straight-line distance between each of these cities and Bucharest. Let's assume we have such a map, and the straight-line distances are as follows:

Straight-line distance  
to Bucharest

<b>Arad</b>	366
<b>Bucharest</b>	0
<b>Craiova</b>	160
<b>Dobreta</b>	242
<b>Eforie</b>	161
<b>Fagaras</b>	178
<b>Giurgiu</b>	77
<b>Hirsova</b>	151
<b>Iasi</b>	226
<b>Lugoj</b>	244
<b>Mehadia</b>	241
<b>Neamt</b>	234
<b>Oradea</b>	380
<b>Pitesti</b>	98
<b>Rimnicu Vilcea</b>	193
<b>Sibiu</b>	253
<b>Timisoara</b>	329
<b>Urziceni</b>	80
<b>Vaslui</b>	199
<b>Zerind</b>	374

Below are the steps that an algorithm that uses the shortest straight-line distance heuristic would take. From Arad, we choose to go to Sibiu, because Sibiu is closer to Bucharest than Timisoara or Zerind. The next city chosen would be Fagaras, for the same reason, and then we reach Bucharest.



For this example, our heuristic has taken a route straight to Bucharest, without ever choosing a node that is not on a route to Bucharest.

However, it is possible that our heuristic might choose a route that does not actually lead to our target city. For example, if the road from Arad to Sibiu went no further, our heuristic would still chose this road, as it takes us closest to Bucharest, but from there we would have to backtrack. In this case, the heuristic would have caused us to expand an unnecessary node.

### [Does our algorithm find the shortest path from Arad to Bucharest?](#)<sup>1</sup>

It is also worth noting that our algorithm does not find the "shortest" route from Arad to Bucharest. The route it finds, is 450km long, but in fact, the route from Arad, via Sibiu, Rimnicu Vilcea and Pitesti is shorter at 418km. Another algorithm would be required for this - do you know which one?

We can write down the algorithm for finding a path from Arad to Bucharest as follows:

```
function find_route(start_node, goal_node):
    if start_node != goal_node:
        mark start_node visited
        sort the unvisited neighbours of start_node by distance to goal_node
        (closest first)
        for each unvisited neighbour of start_node:
            if find_route(neighbour, goal_node):
                return true
    else:
        return true
    return false
```

Why does the algorithm need to mark nodes that have been visited already?

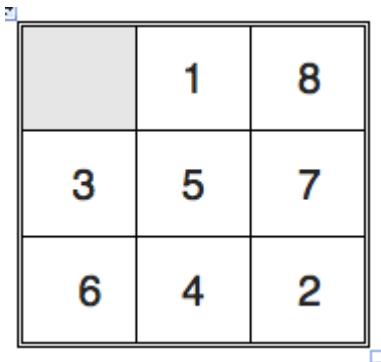
Can you flesh out this pseudocode to create an program in Edgy? Mark the successful route in a different colour. Download the attached file below ([romania\\_route\\_finding](#)), containing a graph of the cities and roads between them, corresponding to the map above. Each node in the graph also has an attribute "kms to Bucharest", with the straight-line distance from this city node to Bucharest. You can then right click on the stage in the Edgy window and select "import from file" to import the graph into your copy of Edgy.

[romania\\_route\\_finding](#) ([https://www.alexandriarepository.org/wp-content/uploads/20150510032137/romania\\_route\\_finding.txt](https://www.alexandriarepository.org/wp-content/uploads/20150510032137/romania_route_finding.txt))

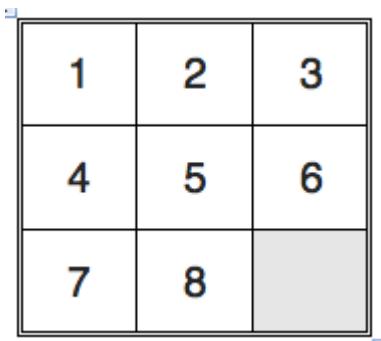
## Example: The 8 Puzzle

Now, let's look at a puzzle that consists of a  $3 \times 3$  board of square tiles with one tile missing. The tiles are numbered from 1 to 8. They can slide against one another, and a tile can move into the empty space if it is horizontally or vertically adjacent to it (i.e. not diagonally adjacent). The aim of the puzzle is to place the tiles in numbered order. This puzzle is known as the 8 puzzle, because it consists of 8 tiles and one empty square inside a 9-square frame.

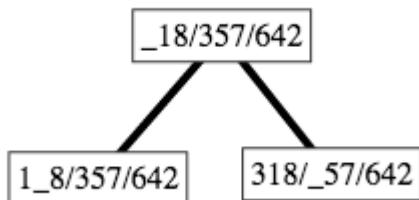
Here's an example, in which the tiles are unordered:-



And we aim to reach a solution, in which the numbers are ordered correctly, like this:-



If we create a graph representing the 8 puzzle game, we can represent each configuration of tiles as a node in a graph. An edge between two nodes, represents that it is possible to move between two configurations by sliding a single tile. Here's an example of some possible moves to and from the unordered tile configuration above:-



Imagine we want to generate a graph to represent every possible configuration of tiles (using nodes), and the possible moves between them (using edges). What might this graph look like? Well, it wouldn't be a tree, it would contain cycles and the edges would be undirected. Additionally, there are  $9! = 362880$  possible board configurations, so it would contain a huge number of nodes. This graph would be very hard to generate!

Instead, we want to consider how we can generate the graph as we go about solving the 8 puzzle. We can use a best first heuristic to save generating too many new nodes and generate the nodes that are most likely to lead to a solution. But what sort of heuristic could we use? Can you think of a rule of thumb that we can use to judge which of the possible next moves might be best?

Well, we could count the number of tiles that are in the wrong position in each of the possible next moves. Then, we could choose our next move to be the one with most tiles in the correct position.

Another heuristic we could use is to calculate the sum of the taxicab (or city-block) distances between each tile and its position in the goal configuration, for each of the possible next moves. The taxicab distance is the the distance between two points located on a grid, where the only path allowed is a combination of vertical and horizontal lines. So, the taxicab distance, describes the number of vertical and horizontal moves, that the tile must make to get to its final position.

We can write down an algorithm for solving the the 8 puzzle as follows:-

```
function eight_puzzle(configuration):
    if not solved:
        identify the possible moves
        create the nodes for these moves
        apply the heuristic to each of the unvisited nodes
        sort the unvisited nodes according to the heuristic
    foreach move in the sorted list of possible moves:
        eight_puzzle(move)
```

But, how will we know which nodes have already been visited? We need some way of marking them visited to avoid re-visiting them and repeating work that we have already done. Can you refine the above algorithm (in pseudocode) to mark 'visited' nodes? You might like to satisfy yourself that it is correct by working through your algorithm using a few moves from an example 8 puzzle.

Once you have done this, modify your algorithm to colour 'visited' nodes and nodes that form part of the final solution different colours. For example, we could mark visited nodes yellow, and solution nodes green. To do this, mark the nodes that are currently being visited by the eight-puzzle algorithm as green. Then, once the algorithm has finished visiting them, and if it has not found a solution, mark them yellow.

---

<sup>1</sup> No. The route it finds is 450km long, but the route via Sibiu, Rimnicu Vilcea and Pitesti is shorter at 418km. Best first search is a heuristic approach which is not guaranteed to find the best solution.

## 2.14

# Applied Algorithms (DRAFT)

### Justifying a choice of data type

When choosing a data type, we need to consider the information that needs to be represented, and how it will be accessed and possibly modified.

For example, in the module on graph traversal, we saw several versions of the algorithm, using lists, queues, and stacks. The BFS algorithm needed to add nodes to one end of a list, and remove them from the other end. In contrast, the DFS algorithm needed to add and remove nodes from the same end of the list. Given these strong restrictions on how we needed to manipulate our data, we could choose the most restrictive data type.

This has the benefit of helping us not to perform unintended operations. For example, if we chose the stack, and tried to apply a queue operation to it, there would have been an error, and we would have been alerted to a problem with our algorithm. No such error would have occurred if we used the more flexible List ADT.

In other situations we have more flexibility. For instance, suppose we want to maintain a calendar of people's birthdays. It will need to contain dates and names. How will we represent these? There are many formats for date, such as a string "25 September", or "25/9", or a pair of integers (25, 9), or an integer and a string (25, "Sep"), a floating point number 9.25, etc. There are various choices for names as well, e.g. "Usain Bolt", ("Usain", "Bolt"), ("Bolt", "Usain"). We need to allow for the fact that two people can share a birthday.

Once we have decided on the data types to use for names and dates, we need to decide how to represent the correspondence between names and dates. For instance, we could have a list of (name, date) pairs:

```
[ ("Usain Bolt", "AUG21")), ("Roger Federer", "AUG08"), ... ]
```

Is this a good choice? It depends on how we want to use it. For instance, do we want to be able to look up the birthday of a particular person, or do we want to be able to find out who has a birthday today? Do we want to find who has a birthday in the next week so we have time to buy presents? The above list representation is a poor choice if we want to be able to do look ups by name or date.

To rigorously justify our choice, we write down the proposed representation, then list the operations we would like to perform and show how they would be carried out using our representation. For example:

Proposed representation: a dictionary mapping month names (represented as strings) to a dictionary mapping days (represented as integers) to a list of people's names (represented as strings), e.g.

```
{
  "August" -> {
```

```

21 -> ["Usain Bolt", "Count Basie", ...],
8 -> ["Roger Federer", ...],
...
},
"March" -> {
    14 -> ["Albert Einstein", ...],
    ...
}
}

```

1. Add someone to the birthday calendar, with name n, month m, and date d:  
if calendar[m][d] does not exist, initialise it to the empty list; append n to calendar[m][d]
2. Find all people who have birthday m/d:  
look up calendar[m][d]
3. Find all people who have a birthday tomorrow:  
find today's date, increment it (requires another algorithm, but we have made it easier by keeping the month and day separate), then apply method 2 above.

Supposing at this point, we realised we wanted to also be able to list today's birthday people in order of decreasing age. We would need to work out where to add the birth year information. Perhaps it would go in the above nested dictionary structure, or it could go in a separate dictionary that maps from names to birth years. Suppose further, it occurs to us that two people can have the same name, how would we deal with this?

When justifying the choice of data type, it is also good to consider a couple of alternatives, and explain why these would be inferior. For example, using a floating point number to represent the date makes it harder to do various operations (work out the next day, convert it to the "day month" format, and so on). Using an array to hold all days of a month (where day d is stored in position d), makes it a little cumbersome to initialise our data structure, and more wasteful if we don't have many birthdays to store.

# 3

# Principles of Algorithm Design

[3.1 Big-O notation: a brief introduction](#)

[3.2 Algorithm analysis: a primer](#)

[3.3 Divide and Conquer](#)

[3.3.1 Binary Search Trees](#)

[3.4 Complexity of Recursive Algorithms](#)

[3.4.1 Solving a Recurrence Relation by Telescoping](#)

[3.4.2 The Master Theorem for Divide and Conquer](#)

[3.4.3 Recursive Complexity: An Example](#)

[3.5 Dynamic Programming](#)

[3.6 Game Trees and the Minimax Algorithm](#)

[3.7 Computationally Hard Problems](#)

[3.7.1 Travelling Salesman Problem](#)

[3.7.2 NP-Hard Problems: Soft Limits of Computation](#)

[3.7.3 Heuristics](#)

[3.7.4 Heuristics for the Graph Colouring Problem](#)

---

## **3.1**

# **Big-O notation: a brief introduction**

This video by Alan Dorin introduces the main mathematical formalism that is used to analyse the runtime complexity of an algorithm, i.e. to determine how much runtime an algorithm needs to solve a problem: The Big-O notation.



(<https://www.alexandriarepository.org/wp-content/uploads/BigOComplexity-Wi-Fi-High.mp4>)

It is recommended that you still read the text version (the "primer" in the next module) after watching this video as it provides a bit more formal detail.

---

## **3.2**

# **Algorithm analysis: a primer**

Chapter 1 of *Introduction to Algorithms and Data Structures* by Michael J. Dinneen, Georgy Gimelfarb and Mark C. Wilson gives a good first introduction to algorithm analysis and runtime complexity ([download link](https://drive.google.com/file/d/0Bz0_PKKD2CDJRW96R1MzSGVUQjg/edit?usp=sharing) ([https://drive.google.com/file/d/0Bz0\\_PKKD2CDJRW96R1MzSGVUQjg/edit?usp=sharing](https://drive.google.com/file/d/0Bz0_PKKD2CDJRW96R1MzSGVUQjg/edit?usp=sharing))).

[Show "AlgorithmAnalysisExcerptDinneen.pdf - Google Drive"](#)

([https://docs.google.com/file/d/0Bz0\\_PKKD2CDJRW96R1MzSGVUQjg/preview](https://docs.google.com/file/d/0Bz0_PKKD2CDJRW96R1MzSGVUQjg/preview))

---

## 3.3

# Divide and Conquer

### a quick introduction

## A First Example: Sorting Nuts and Bolts

Imagine yourself as the remote operator of the latest orbital probe. You are in the middle of assembling a mission-critical piece of equipment with the remote-controlled manipulators. Unfortunately, you have just spilt the entire set of bolts and nuts for this piece of equipment and you have no way to tell which bolt fits which nut except for trying.

What is worse, the only information you get back from the remote manipulator when you try to fit a particular bolt and nut pair is whether the nut fits the bolt, is smaller, or is larger than it. It cannot directly report an absolute size to you.

You are in a hurry, so you want to sort the set of bolt and the set of nuts into matching pairs in as few tries as possible. How would you go about this systematically?

In other words, you need to design an algorithm that uses the minimum number of comparisons to sort two sets  $N$ ,  $B$  into matching pairs, where the only information you get from a comparison is whether  $x$  matches  $y$  ( $x$  in  $N$ ,  $y$  in  $B$ ) and where you can assume that each element of  $N$  has exactly one matching element in  $B$ .

### Naive Solution

The naive way to solve the problem is quite easy to figure out: we simply try all possible combinations of nuts and bolts. This will certainly allow us to sort the set completely.

```
Algorithm nuts-and-bolts-naive(N, B)
    input: two lists of integers, representing sizes of nuts and bolts

    foreach x in N
        foreach y in B
            if (x=y) then
                report "x matches y" (* or use the pair x,y *)
            end
        end
    end.
```

But how long will it take us to do this? Let us assume the time a comparison takes is the dominant contribution to the time required. We can thus take the number of comparisons executed as a proxy for the runtime of the algorithm (we will make the general notion of runtime more precise later on). In the form that we have written down the algorithm above it clearly checks all possible pairs. If the size of  $N$  and  $B$  is  $n$ , we have  $n^2$  possible pairs, so this is the number of comparisons executed.

## A first improvement

Can we do better than that? One might object that the algorithm above proceeds to check other bolts for a given nut  $x$  even after it has found a match for  $x$ . This is, of course, completely wasted work. Let us assume we modified the algorithm so that it does not do that and proceeds to the next nut straight away (you should do this as an exercise). How many comparisons would we execute now?

Clearly, this depends on the order of nuts and bolts in  $N$  and  $B$ . What would happen if the two lists  $N$  and  $B$  are already sorted? For every nut we would find the matching bolt on the first try and thus we would only execute  $n$  comparisons! Of course, this would be very lucky. It is actually the best case that could happen. Usually, when we analyse an algorithm, we need to look at the worst case.

What would be the worst case?

If one of the lists is sorted in ascending order and the other one in descending order, we would need  $n$  comparisons to find the first pair,  $(n-1)$  comparisons for the second pair,  $(n-2)$  comparisons for the third pair, and so on. Thus, the total number of comparisons is:

$$\sum_{i=0}^{n-1} (n - i) = \frac{n}{2}(n + 1) = \frac{n^2+n}{2}$$

While this was certainly an improvement, it has not even made the algorithm twice as fast. The number of comparisons still grow quadratically with the number of nuts. We need to think of something more fundamental.

## A real improvement

We can make one fundamental observation when we look closely at how the algorithm proceeds: it is actually not using all the information it can have without doing more work. Each comparison can not only tell us whether a nut  $x$  matches a bolt  $y$ , it can also tell us (for free!) if the nut is bigger or larger than the bolt. We should not throw this information away. For a given nut  $x$  it would be easy to split the list of bolts as a byproduct of finding the matching bolt  $y$  into one list  $BS$  of those bolts that are smaller than  $x$  and another list  $BL$  of those that are larger than  $x$ . But how does this help us? If we try to find a bolt for the next nut, we would still have to look through all of  $BS$  and  $BL$ , so nothing seems to be gained.

However, if we managed to also split the list of nuts into one list  $NS$  of those nuts that are smaller than  $x$  and one list  $NL$  of those that are larger than  $x$ , our job would become simpler: in all further checks we would only have to compare nuts in  $NS$  with bolts in  $BS$  and nuts in  $NL$  with bolts in  $BL$ . This splitting easily be done: The matching nut  $x$  and bolt  $y$ , of course, have the same size. So we can easily split the list of nuts using the matching bolt  $y$ . Note that in the algorithm below both of our list-splitting operations (the one for the nuts and the one for the bolts) return a matching element as well as the list of smaller and larger elements. The second split (the one for the nuts) would not really need to do this! Of course, the returned element  $z$  has to be the same as  $x$ , since bolt  $y$  fits nut  $x$  (first split) and in turn nut  $z$  fits bolt  $y$ .

Algorithm nuts-and-bolts( $N$ ,  $B$ )

input: two lists of integers, representing sizes of nuts and bolts

```
if ( $N$  is not empty) then begin
    let  $NS$ ,  $NL$ ,  $BS$ ,  $BL$  be empty lists
    let  $x$  be the first nut in  $N$ 
```

```

(BS, y, BL) = split(B, x)
(NS, z, NL) = split(N, y)
report "z matches y" (* or use the pair (z,y) which is the same as
(x,y) *)
    nuts-and-bolts(NS, BS)
    nuts-and-bolts(NL, BL)
end
end.

```

Note that the algorithms proceeds recursively: It splits the sets and calls itself with the smaller subsets as arguments. The problem always remains unchanged with each recursive call, just the size of the problem (the number of nuts and bolts) becomes smaller until it is of trivial size. This is the base case, where there is nothing left to match.

Of course we also need the algorithm for splitting the lists:

```

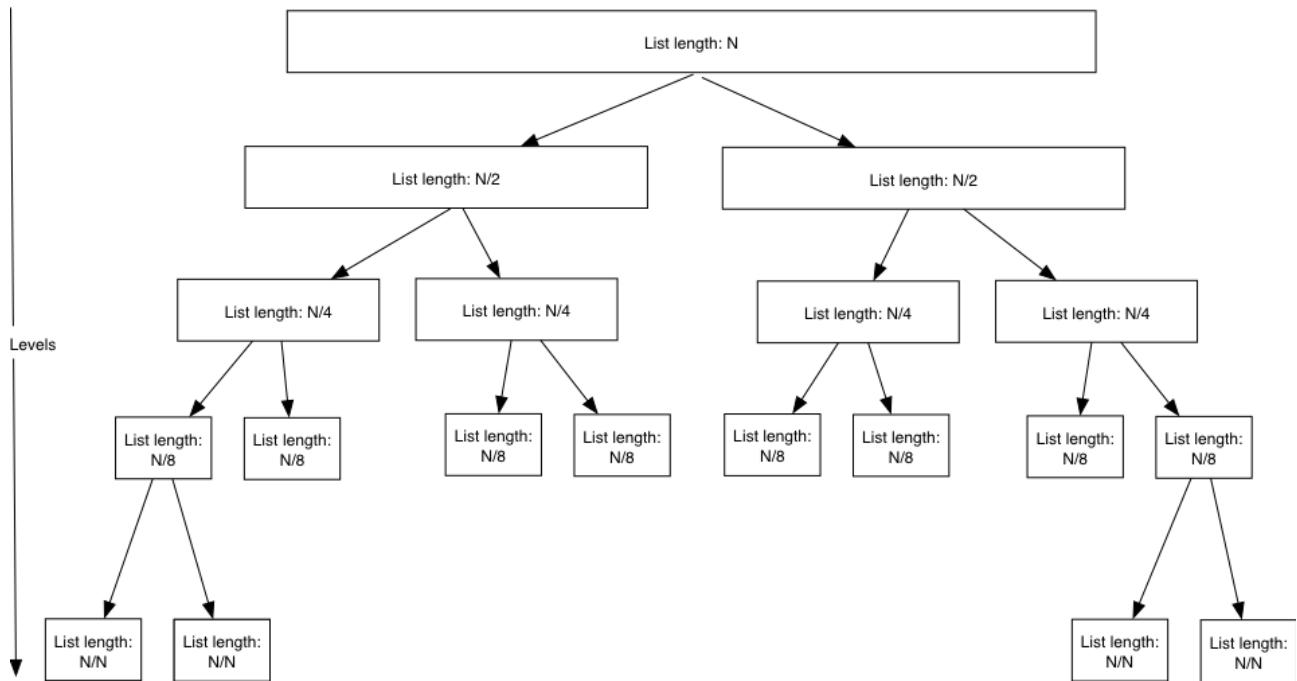
Algorithm split(L, x)
    input: a list of integers, representing sizes of nuts or bolts
    output: a triple consisting of
            a lists with all elements of L that are smaller x,
            the element y of x that matches x
            a list with all elements of L that are larger than x

    let LS, LL be two empty lists
    foreach a in L
        if (a<x) then add a to LS
        else if (a>x) then add a to LL
        else let y=a
    end
    return (LS, y, LL)
end.

```

It seems clear that this must save us some comparisons, but how much exactly? Above we have looked at the *worst case* and the *best case*. We now start by looking at the *average case*.

When nuts-and-bolts is called the first time, both its argument lists have length n. The lists that the algorithm passes on to the next recursive call are clearly shorter. On average we could expect that half of the elements of N belong to NS and half to NL (and accordingly for BS, BL). In this case the next recursive call would be executed with two lists of length n/2. This, in turn, would execute calls with lists of length n/4 and so on. Here is a picture of the call tree.



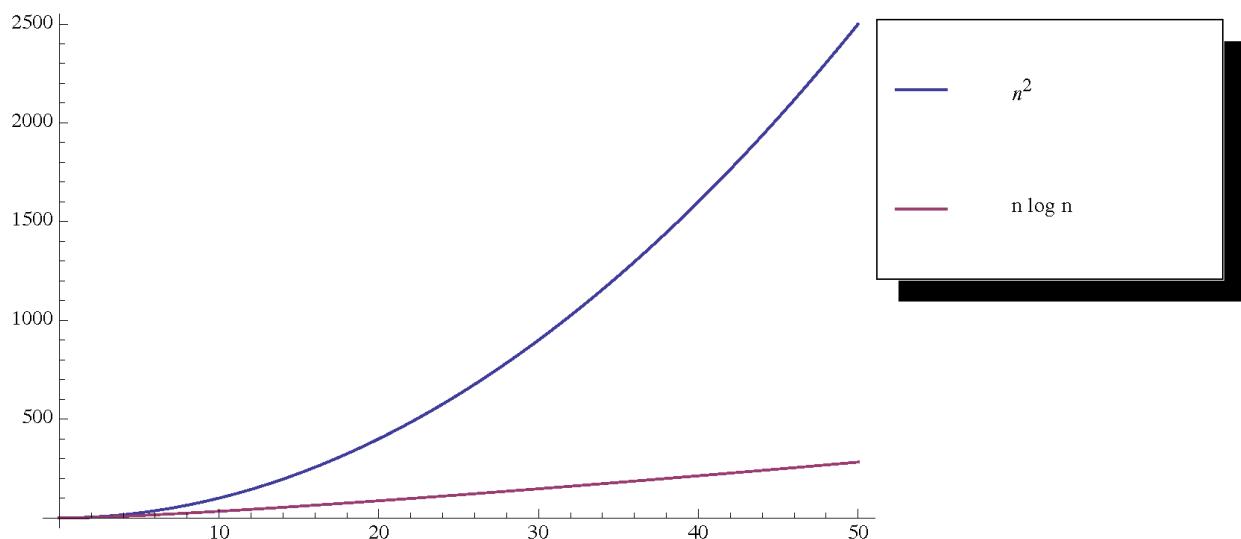
When do we reach the base case? The length of the list is  $(n/2^i)$  for the  $i$ -th level of recursive calls (we count the original call, the root of the call tree, as the first level). Thus, when  $i = \log_2 n$  the list length of the recursive call is 1 and we reach the base case with the next call. Thus there are  $1 + \log_2 n$  levels in the call tree.

How many comparisons do we execute on each level? This is simple: the only comparisons are executed in split, and split performs a linear scan over the list, thus it executes  $n$  comparisons for a list of length  $n$ . We execute two calls to split for lists of equal length. Thus, on each level of the call tree, we have a total of  $2n$  comparisons. (Observe that at each level of the call tree, all the different sub-lists NS, NL that  $N$  has been split into together have exactly the same elements as the original  $N$ . The same holds for B and BS, BL).

In summary, we have  $O(\log_2 n)$  levels of calls and  $n$  comparisons on each level. The total amount of comparisons executed is

$$O(\sum_{i=1}^{\log_2 n} n) = O(n \cdot \log_2 n)$$

This is a very significant saving over our previous algorithm. In the diagram below you can see how much slower the number of comparisons grows as the number of elements  $n$  increases.



Now think about what the worst case and the best case would be and try to work out the complexity. Do we gain anything in these cases? (Hint: the number of comparisons on each level of the call tree always stays the same, the crucial question is: how deep will the call tree grow in the different cases).

## A Second Example: Finding the Fake Coin

The following video describes a classical logic puzzle: the fake coin problem.

In summary, you are given a set of  $n$  gold coins and you are being told that exactly one of them is a fake coin. The only tool that you have at disposal is a scale. You know that all the coins have the same weight except for the fake coin. How do you find the fake with the least number of weighings?

The naive solution is again quite obvious: you can pick each coin in turn and compare it to all other coins. All pairs of coins that you compare must be equal unless they contain a fake. Thus, if you find a difference in one pair you know that one of two coins is a fake and it is quite easy to determine which one. As you know there is only one fake, all that is left to do is to compare one the coins to an arbitrary third one. If there is again a difference this coin is the fake one. If there is none the other coin of the suspicious pair is the fake. Here is the algorithm:

```

Algorithm fake-coin-naive(L)
    input: a list of integers, representing the weight of the coins

    foreach x in L
        foreach y in L
            if (not x=y) then begin
                let z be an arbitrary coin in L different from x and y
                if (not x=z) report "x is the fake"
                else report "y is the fake"
            end
        end
    end
end.

```

As in the previous problem, we are performing comparisons for all possible pairs of coins and thus have

$O(n^2)$  comparisons. Likewise, as for the previous problem, we could easily save quite a few comparisons: the way the naive algorithm is written above, it considers each pair twice! However, even if we did this, we would still need on the order of  $(n^2)/2 = O(n^2)$  comparisons, so this would not save us much.

At this point we might have the hunch that the same technique as before would help us to derive a more efficient algorithm.

To illustrate this, we shall consider a somewhat easier (but not fundamentally different) version of the problem. The simplification in version is that we have more information: We know that the fake coin is lighter than a real coin - gold is exceptionally heavy, and the fake is just made from brass.

The idea for a divide-and-conquer solution for this problem is simple: split the set of coins in half, and compare the total weight of both sets. One of the sets must be lighter. Discard the heavier set - these are all real coins. (On second thoughts, don't. One should never discard real gold coins. put them in your piggy bank instead). Continue recursively with the lighter set. The base case clearly is where there is only a single coin left. This must be the fake coin.

```
Algorithm fake-coin(L)
    input: a list of integers, representing the weight of the coins

    if (length(L)=1) then return first(L)
    else begin
        if (length(L) is odd) then let a := first element of L
        let L1 := first half of (L without its first element)
        let L2 := second half of (L without its first element)
        if (total-weight(L1)>total-weight(L2)) then
            let b := fake-coin(L2)
        else let b := fake-coin(L1)
        if length(L) is even then return b
        else return min(a,b)
    end
end.
```

A fully precise description of the algorithm as Edgy code is given further below.

From what we have seen so far, the analysis of the number of comparisons this algorithm executes is straight forward: the length of the list is halved with each level of recursive calls, so we have  $\log_2 n$  levels of calls. On each call level we have to perform a single comparison of the total sum of the sub-lists. (If we want to be more precise, we should actually say 3 comparisons: one of the total weights, one to determine whether the list length is odd, and one for the  $\min(a,b)$  operation. We shall disregard this for now, and only count one. This is justified because we are only interested in the fact that this is  $O(1)$ , not the exact number.

The total number of comparisons executed is thus simply  $O(\log(n))$ . This is a huge saving over the naive algorithm!

If you had previously come across the Binary Search Algorithm for finding an element, you will notice how the call tree of the fake coin algorithm is very similar to that of the binary search method. As a reminder: Binary search finds an element in a sorted list or another sorted structure. Assume you want to find a name in a phone book. A linear scan through all names would be very inefficient. Instead you could start

by comparing a name (somewhere) in the middle of the phone book to the one you are looking for. If the one you are looking for comes earlier in a lexicographical ordering (alphabetical ordering) you proceed (recursively!) to look in the first half of the phone book, otherwise you proceed to search in the second half.

The considerations for the number of comparisons that you need to find the entry (or to ascertain that the name is not in the book) are exactly the same as for the fake coin algorithm.

We have still not squeezed every last bit out of the fake coin algorithm: Instead of splitting the list into two halves, we could split it into three parts A, B, C. We could then compare just two of these, say A and B. If they have the same total weight, C must contain the fake. If they have different ones, the lighter of A and B must contain the fake. The question for you to work out as an exercise is: What is the number of comparisons needed with this modification. Does the modification reduce the previous number of comparisons by more than a constant amount?

## A harder version of the problem

The full version of the fake coin problem is more complex. It does not assume that the fake coin is lighter, it only assumes that the fake coin is the only one with a different weight. It could be heavier. This problem is left for you as an exercise. A good way to go about solving it is to first contemplate the problem for a fixed, small number of coins.



(<https://www.alexandriarepository.org/wp-content/uploads/FakeCoin-Wi-Fi-High.mp4>)

## A Third Example: Finding Minimum and Maximum

Let us consider another example. Given a list L of numbers, you have to find the smallest and the largest number in it. We might be tempted to use the same technique that we have just applied to the other problems expecting spectacular savings: split the list and treat the sublists recursively.

More concretely, we could split the original list (containing n numbers) in the middle, find the minimum

and maximum of the two sublists, and compare the two candidate minima (maxima, respectively). If n is odd, we can simply let one of the two lists be one element longer. The algorithm follows:

```

Algorithm min-and-max(L)
    input: a list of numbers
    output: a tuple consisting of the minimum and the maximum
            number in the list

    let n=length(L)
    let L1 be a list with the first (n/2) elements of L
    let L2 be a list with the remaining elements of L (* L-L1 *)
    let (min1, max1) = min-and-max(L1)
    let (min2, max2) = min-and-max(L2)
    if (min1<min2) min=min1 else min=min2
    if (max1>max2) max=max2 else max=max1
    return (min, max)
end.
```

[Did the application of the technique save us a lot of comparisons in this case? <sup>1</sup>](#)

The call tree looks exactly like the one we have looked at above. Each time we split the list in halves.

Thus, we clearly have  $\log_2 n$  levels of recursive calls again. The comparison which we have to count happen when we compare the tentative minima and maxima. At level i we have  $2^i$  different instances of the recursive call (remember that we count the first call, the root of the call tree as Level 0). On Level i we thus have to compare  $2^i/2=2^{(i-1)}$  different minima pairs and correspondingly many different maxima pairs, thus we need  $2^i$  comparisons. Summing up the number of comparisons for all levels of the call tree we arrive at:

$$\sum_{i=1}^{\log_2 n} 2^i = 2 \cdot (n - 1)$$

This is definitely less than for the previous nuts-and-bolts problem, where we do not have a shrinking number of comparisons per level rather than a constant amount of n comparisons per level.

But did it save us much? The answer is no. The most naive solution for the problem is to simply perform two linear sweeps over the list: one to find the minimum and one to find the maximum. Each linear sweep executes  $(n-1)$  comparisons, so the naive solution would only have needed the same amount of comparisons.

Whether or not this design principle is useful clearly depends on a number of factors: How deep the call tree will grow, how much effort we spend on each level, and also on how costly a naive solution is.

## The General Principle

Let us briefly recap the basic principle of the divide and conquer paradigm:

- Divide and Conquer is a recursive approach. The two fundamental steps are to
  1. Divide the problem into a number of subproblems that are in structure identical to the original problem and then solve these recursively.
  2. To combine the solutions of the subproblems into a solution of the whole problem on each level of

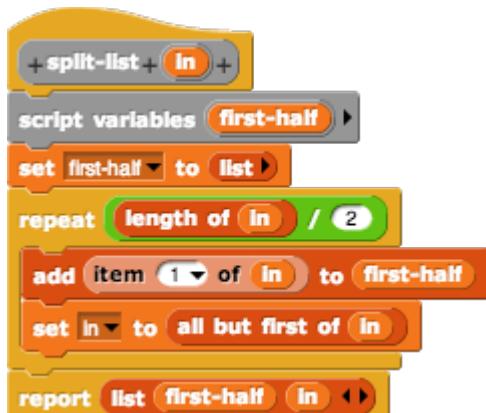
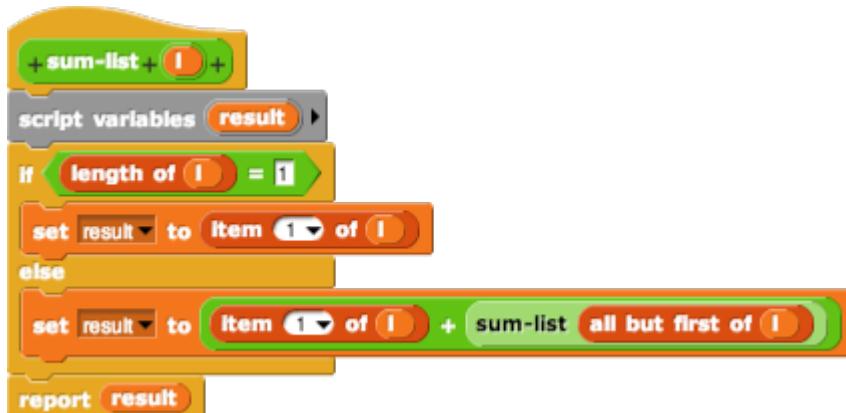
recursion.

- Divide and Conquer is only useful if the subproblems can be solved independently of each other.
- The cost of a divide and conquer algorithm is determined by the depth of the call tree and the amount of work performed at each level.
- If the size of the subproblems is reduced by a constant factor for each recursive level of calls, the resulting call tree has  $O(\log n)$  levels, where  $n$  is the size of the original problem. This is typically the case when a problem is split into sub-problems of equal size.
- If the subproblem is only reduced by a constant amount (as opposed to factor), the call tree will have  $O(n)$  levels.
- If the divide and merge phases are trivial (i.e. only need a constant amount of work that is independent of the problem size), and the problems are split into subproblems of equal size the overall runtime of the algorithm will be of order  $O(\log n)$ .
- If the divide and merge phases require and amount of work that is linear in the problem size, the overall runtime of the algorithm will be of order  $O(n \log n)$ .
- Divide-and-conquer is often a good approach where a naive solution has to consider all possible pairs so that its runtime scales quadratically with the size of the problem.
- For the typical cases of divide and conquer, there is a beautiful theorem, the [Master Theorem](http://en.wikipedia.org/wiki/Master_theorem) ([http://en.wikipedia.org/wiki/Master\\_theorem](http://en.wikipedia.org/wiki/Master_theorem)) which you can use to derive the overall runtime complexity of the algorithm.

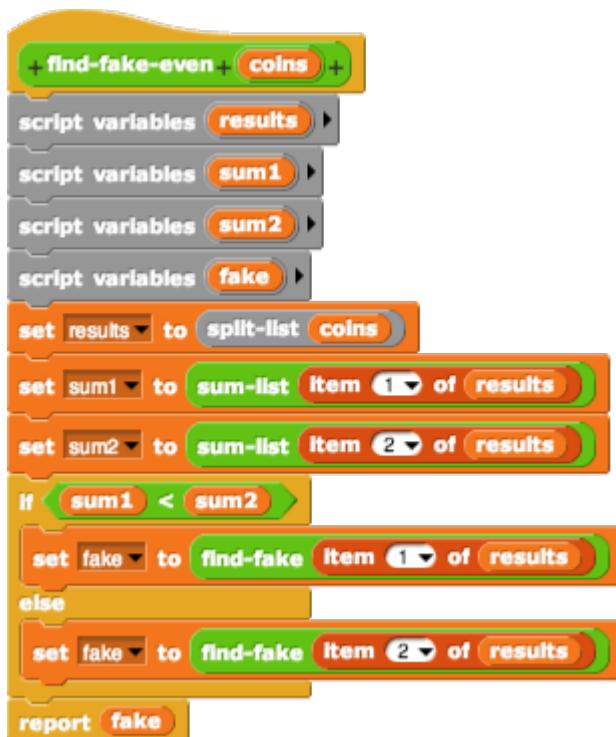
## Edgy Code for the Fake Coin Problem

Below is the edgy code for the fake coin problem as outlined in the algorithms above.

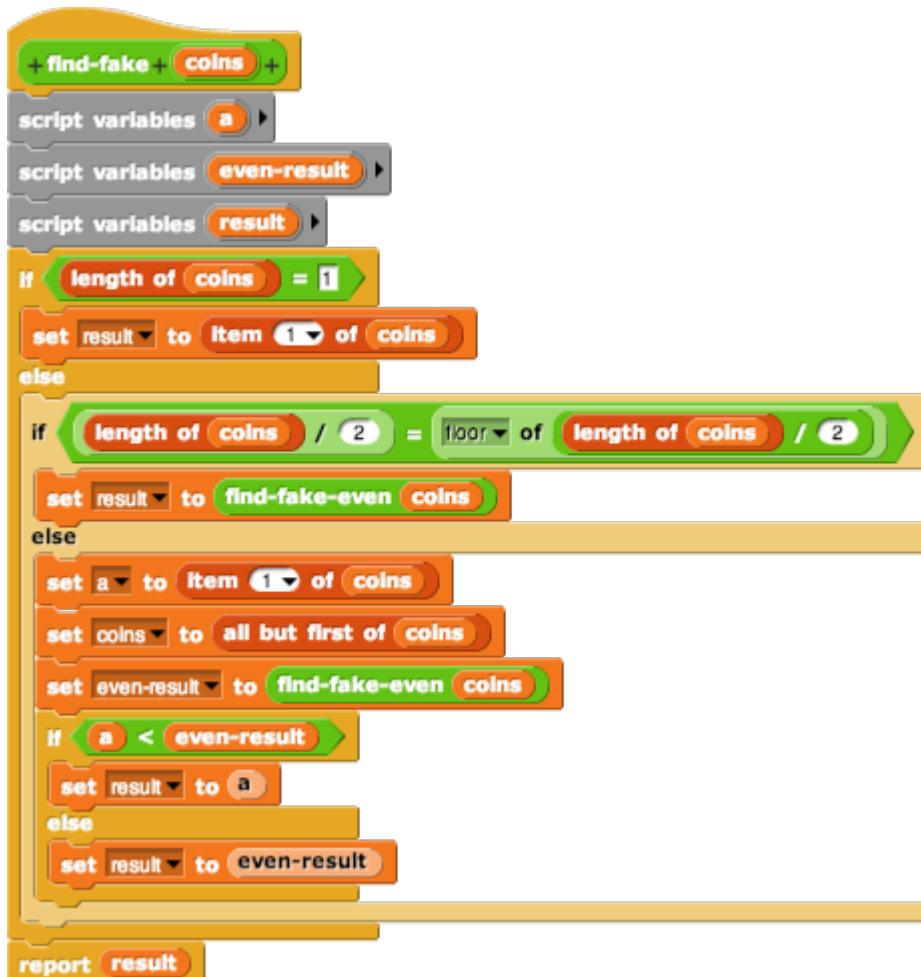
First, we have to define two helper function to split the list and to sum up all weights in a list:



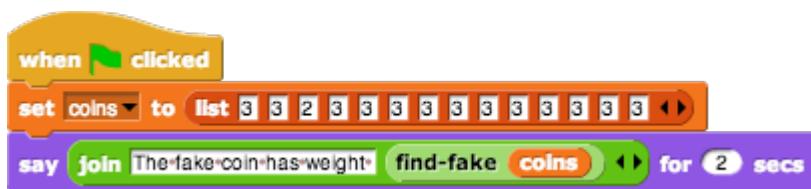
The core functionality is in the block "find-fake-even". this will only work for even list lengths.



We thus need another auxiliary function that takes care of odd list lengths by isolating one element.



The main function simply sets up a test list and c



<sup>1</sup> Somewhat surprisingly not. Here is why...

## 3.3.1

# Binary Search Trees

A special case of divide and conquer is the case of binary search. If you have ever looked up a word in a printed dictionary, you will probably have experienced binary search, or something close to it.

This algorithm takes a sorted list of items (here, integers) along with a particular item that we are searching for. If the item is found, then it returns its position in the list. If it is not found, it returns the value -1 (an impossible position).

```
Algorithm binary_search(L, x)
    input: a sorted list of integers
    output: the index of x in L, or -1 if x was not found

    left := 0
    right := len(L) - 1

    while left <= right
        mid := floor( (left + right) / 2 )
        if L[mid] < x
            left = mid + 1
        else if L[mid] > x
            right = mid - 1
        else
            return mid
    return -1
```

To see how this works, let's step through the algorithm for particular inputs:

$L = [7, 16, 29, 52, 85, 136, 184]$   
 $x = 85$

iter	left	right	mid	$L[mid]$
1	0	6	3	52
2	4	6	5	136
3	4	4	4	85
4	4	4	4	

The algorithm finishes by returning the value 4.

Notice what happens to the size of the task each time through the while loop. Initially, we are dealing with a list of size  $n$ . The item we are seeking could be anywhere in the list. After one iteration, we have eliminated roughly half of the candidates, and the size of the problem has halved.

Notice also that we could have been lucky, and found the item on the first or second time through the while loop (how?). But if the item is not in the list at all, then there are no short-cuts. The loop has to run until  $left > right$ .



## 3.4

# Complexity of Recursive Algorithms

In the last discussion of Divide and Conquer we have seen an informal (or semi-formal) argument for the improved complexity of Divide and Conquer over naive solutions for some problems.

Hopefully this was convincing, but convincing is, of course, not good enough. We want to *know precisely* and *with certainty*, and the only way to get to this point is to conduct the argument formally.

We have already discussed the complexity of iterative algorithms a fair bit, and you have probably developed a good intuition for the complexity of such algorithms based on their structure, for example that an algorithm that uses nested loops typically has (at least) polynomial complexity with a degree that depends on the depth of nesting (provided they iterate over collections of the same size): linear for a single loop, quadratic for two nested loops, cubic for three nested loops and so forth.

For recursive algorithms intuition often fails. But we could at least make a start with thinking about the problem by drawing analogies to how we handled iterative algorithms. To determine the complexity of an iteration, the central questions to ask are:

1. how often is the loop executed and
2. how much work is done in each iteration?

Equivalent questions can be asked for a recursive algorithm:

1. how often is the recursive call executed and
2. how much work is done at each level of recursion.

In both cases we obtain a series (a summation formula) that describes the complexity.

This was exactly how the informal analysis for the nuts-and-bolts problem proceeded: We drew the call tree, determined its depth (ie. how often the recursive call is executed), and determined the complexity of each level of the call tree. The same approach will also form the basis of the formal analysis that we conduct now.

## Deriving a Recurrence Relation for a Recursive algorithm

The central tool to formally derive the complexity of a recursive algorithm are [Recurrence relations](https://en.wikipedia.org/wiki/Recurrence_relation) ([https://en.wikipedia.org/wiki/Recurrence\\_relation](https://en.wikipedia.org/wiki/Recurrence_relation)). Recurrence relations are equations that define a sequence recursively, ie. in terms of itself, and that take integers as arguments. A typical simple example is the definition of the [Fibonacci numbers](https://en.wikipedia.org/wiki/Fibonacci_number) ([https://en.wikipedia.org/wiki/Fibonacci\\_number](https://en.wikipedia.org/wiki/Fibonacci_number)):

$$F(n) = \begin{cases} F(n - 1) + F(n - 2)) & \text{if } n > 2 \\ 1 & \text{if } n \leq 2 \end{cases}$$

Effectively, a recurrence relation defines how to fill out a table (or, for a single argument, a list) with the

corresponding values. To fill out entry  $n$  lookup entries  $n-1$  and  $n-2$  and add them...

[You may now be wondering how to fill the table starting with the lower entries first. What should that remind you of?](#)<sup>1</sup>

The fact that these are recursive definitions makes it easy to define the complexity of a recursive algorithm. Take  $T(n)$  to be the complexity of the recursive algorithm when applied to problems of size  $n$ . For a divide-and-conquer algorithm the principle how we can define  $T(n)$  is easy to see:

$$T(n) = \begin{cases} \text{work required to divide the problem} \\ +T(n_1) + \dots + T(n_k) \\ +\text{work required to synthesize solution} \\ \quad \text{from the solutions of the subproblems} \end{cases}$$

where  $n_1, \dots, n_k$  are the sizes of the subproblems generated.

## A Worked Example: Nuts-and-Bolts

Recall the algorithm for the nuts-and-bolts problem. To make the discussion a little bit simpler, we will modify this slightly by stating the base case for list length 1 instead of the empty list. This does not really change anything.

```
Algorithm nuts-and-bolts(N, B)
    input: two lists of integers, representing sizes of nuts and bolts

    if (length(N)>0) then begin
        let NS, NL, BS, BL be empty lists (* 0(1) *)
        let x be the first nut in N (* 0(1) *)
        (BS, y, BL) = split(B, x) (* ? *)
        (NS, x, NL) = split(N, y) (* ? *)
        report "x matches y" (* 0(1) *)
        nuts-and-bolts(NS, BS) (* T(?) *)
        nuts-and-bolts(NL, BL) (* T(?) *)
    end
    else begin
        x=first(B) (* 0(1) *)
        y=first(N) (* 0(1) *)
        report "x matches y" (* 0(1) *)
    end
end.
```

[The first question to ask is, what is the independent variable that we should use to measure problem size?](#)

<sup>2</sup>

-

[next we need to know, what is the complexity of the calls to split?](#)<sup>3</sup>

The next question is a really difficult one. What is the size of the subproblems? To determine this we first need to know which case we want to look at. It is not easy for us to determine what the *best* and the *worst* case are! To determine what is best and what is worst you first need to understand the solution of the runtime recurrence relation - This sounds dangerously like circular reasoning.

We shall postpone this and worry about the average case first.

[what would be the size of the lists NS, BS \(and NL, BL, respectively\) on average?](#)<sup>4</sup>

Armed with this information it is not difficult to write down a recurrence relation for the runtime of *nuts-and-bolts*:

$$T(n) = \begin{cases} O(1) + O(1) + O(1) + 2 * O(n) + 2 * T(n/2) & \text{if } n > 0 \\ O(1) & \text{otherwise} \end{cases}$$

Since we are working with O-notation, we can easily summarize all the  $O(n)$  terms:

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 0 \\ O(1) & \text{otherwise} \end{cases}$$

<sup>1</sup> Dynamic Programming

<sup>2</sup> As in the other cases, this should be the length of the list, ie. the number of nuts (or bolts) as this clearly determines how long the algorithm will run.

<sup>3</sup> split is a linear algorithm, as it performs a single linear sweep across the list. This is of course only true if adding an element to the output lists is  $O(1)$ . You may want to think about the case where this operation itself is linear in the length of the list to which the element is added. What is the complexity in this case?

<sup>4</sup> In the absence of additional knowledge we should assume that on average half of the elements are smaller and half of the elements are larger than the splitting bolt or nut. So their size would be  $n/2$ .

## 3.4.1

# Solving a Recurrence Relation by Telescoping

Recurrence relations that arise from Divide-and-Conquer, like the one for our example, are relatively easy to solve in many cases. However, in general solving a recurrence relation can be a very complex (or even infeasible) task. We will only look at two techniques that are frequently used for applications in algorithm design:

- Telescoping and the
- Master Theorem.

For further information you may want to consult [Concrete Mathematics by R. Graham, D. Knuth, and O. Patashnik \(Addison Wesley 1994\)](#) ([https://en.wikipedia.org/wiki/Concrete\\_Mathematics](https://en.wikipedia.org/wiki/Concrete_Mathematics)), one of the most useful books for computer science maths ever published.

*Telescoping* sounds like a complex process, but it is actually quite straight forward. We simply start with the definition of  $T(n)$  and repeatedly replace the definition of  $T(n)$  into itself until we reach the base case. With any luck we will discover the general pattern that the expansion follows and solve the equation in this way.

Let's do this for the nuts-and-bolts runtime equation:

$$T(n) = 2 * T(n/2) + O(n)$$

Note how we have already summed up the  $O(1)$  and  $O(n)$  terms to make life easier. We now telescope by replacing exactly the same definition (recursively). With

$$T(n/2) = 2 * T(n/4) + O(n/2)$$

we obtain

$$T(n) = 2 * (2 * T(n/4) + O(n/2)) + O(n)$$

Now repeat the process until you see the pattern:

replacing  $T(n/4) = 2 * T(n/8) + O(n/4)$  gives us

$$T(n) = 2 * (2 * (2 * T(n/8) + O(n/4)) + O(n/2)) + O(n)$$

This could go on forever, when do we stop? As with all recursions, when we have reached the base case.

[When do we reach the base case?](#)<sup>1</sup>

You can now see why we have modified the base case slightly!

Let us write the pattern that we are always replacing in a general form:

$$T(n/2^i) = 2 * T(n/2^{i+1}) + O(n/2^i)$$

We can now write the expansion in a generalized form:

$$\begin{aligned} & 2 * (2 * (\dots 2 * T(n/2^{(\log n)})) + O(n/2^{(\log n)-1})) \\ & + \quad O(n/2^{(\log n)-2})) \\ T(n) = & + \quad O(n/2^{(\log n)-3})) \\ & + \dots \\ & + O(n) \end{aligned}$$

Now observe that the innermost term is simply  $T(1) = O(1)$  (the base case) and that all the remaining terms have just the form  $2^{(\log n)-i} * O(\frac{n}{2^{(\log n)-i}})$ . There are  $\log n$  of these terms.

Thus the sum reduces to

$$T(n) = n + n + \dots n = O(n \log n)$$

This exactly corresponds to what we derived before informally: Our call tree has  $\log n$  levels and we perform  $O(n)$  work on each level.

This was not really difficult, but somewhat tedious. Luckily, there is a simpler way (that also avoids us making small errors in the solution!): the Master Theorem.

<sup>1</sup> After we have divided the list  $\log(n)$  times into halves the list length will be 1, so  $\log(n)$  expansions we arrive at  $T(1)$ , the base case.

## 3.4.2

# The Master Theorem for Divide and Conquer

### Levitin version

Luckily, we don't need to perform all the work for telescoping every time we encounter a Divide-and-Conquer algorithm.

There is a general solution to most runtime recurrence relations that we would derive from a Divide-and-Conquer algorithm which is known as the [Master Theorem](https://en.wikipedia.org/wiki/Master_theorem) ([https://en.wikipedia.org/wiki/Master\\_theorem](https://en.wikipedia.org/wiki/Master_theorem)).

The Master Theorem gives us a general solution to recurrence equations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ where } a > 1, b > 1$$

All that we have to do is to match  $a$  and  $b$  to the equations that we are trying to solve.

[What are the factors  \$a, b\$  for the nuts-and-bolts problem?](#)<sup>1</sup>

### Master Theorem

If  $f(n) \in O(n^d)$ , the above recurrence relation has the solution:

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Note that there are stronger versions of the Master Theorem, but we won't discuss these here, as this is sufficient for our purposes.

Deriving or proving the master theorem would be significantly beyond the scope of this course, so we shall simply use it as a (very convenient) tool. But with this tool we can very simply determine the runtime complexity of many recursive algorithms that are relevant to us. All that we have to do is to construct the runtime recurrence relation, read off the factors from its definition, and plug these into the master theorem.

For our nuts-and-bolts problem, the factors are simply  $a = 2, b = 2, d = 1$

We thus have Case 2 ( $a = b^d$ ) and the solution is, as we already know from solving by telescoping

$$T(n) = O(n \log n)$$

For full details regarding the Master Theorem, see [here](https://en.wikipedia.org/wiki/Master_theorem) ([https://en.wikipedia.org/wiki/Master\\_theorem](https://en.wikipedia.org/wiki/Master_theorem)).

1 a=2, b=2

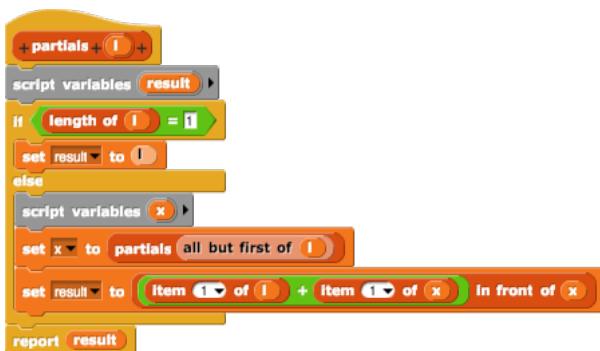
## 3.4.3 Recursive Complexity: An Example

### Partial Sums

Below is a simple example of a recursive algorithm. It computes the partial sums of a list of numbers. The first partial sum of a list is the sum of the whole list, the second partial sum is the sum of all elements but the first one, the third partial sum is the sum of all elements but the first two, and so forth. For example, the partial sums of the list [1, 2, 3, 4, 5] are [15, 14, 12, 9, 5].

Before you continue to read, you may want to construct iterative and recursive versions of algorithms to compute these partial sums yourself.

The Edgy program below is an implementation of the most straight forward way to compute partial sums.



[What is the runtime recurrence relation for this version of the program?](#)<sup>1</sup>

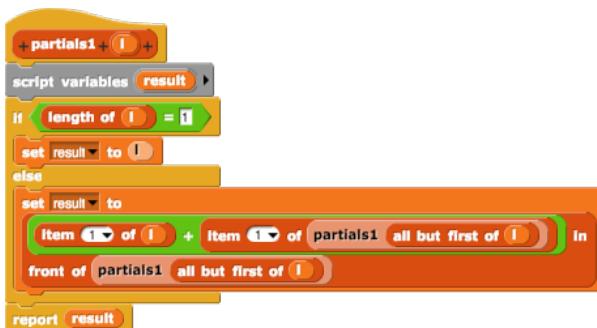
[What, apart from input size, does your calculation of the runtime complexity depend on?](#)<sup>2</sup>

[Can you solve this runtime recurrence with the Master Theorem?](#)<sup>3</sup>

[Can you work out what the runtime complexity of this implementation is?](#)<sup>4</sup>

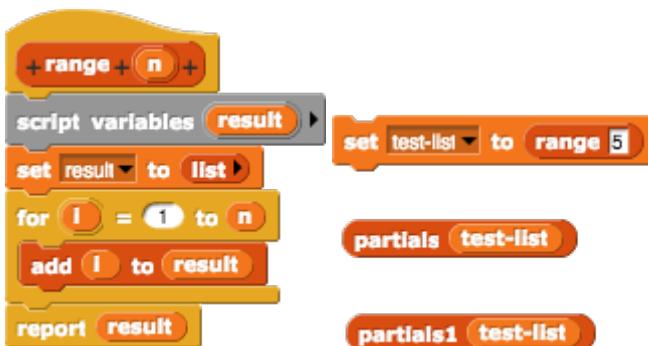
### A "minor" variation of the algorithm

Here is a second implementation of partial sums. Comparing it to the first implementation you will notice that there is hardly any difference; only a single very subtle one: in the new version we were too lazy to define a local variable to hold the value of the intermediate recursive computation. Instead we are computing it directly where it is needed in the formula.



Do you think this will make a major difference for the runtime? Think about this abstractly at first, then explore it by using the implementation. Start with a very short list (say, 2 or 3 elements) and then increase the list length step by step: 5 elements, 7 elements, 10, elements, 15 elements...

To do this you may want to use a little helper function so that you don't have to construct your lists manually.



[Can you give the runtime recurrence relation for the second implementation?](#)<sup>5</sup>

[Can you already make an informed guess what the runtime complexity class is?](#)<sup>6</sup>

[How would you solve this equation?](#)<sup>7</sup>

You see how even minor differences can make a massive difference to an algorithm's complexity. So massive, indeed, that it can change from fast to practically unusable. Careful design and structured (formal) reasoning about the complexity of an algorithm design can thus be a real life saver!

<sup>1</sup>  $T(1)=1; T(n)=T(n-1)+O(1)$

<sup>2</sup> All list ADT operations, including in-front-of and all-but-first-of, are assumed to be  $O(1)$

<sup>3</sup> No. The recurrence relation does not have a form to which the Master Theorem is applicable: The size of the problem reduces linearly not geometrically.

<sup>4</sup> The recursion is clearly executed  $n-1$  times for a list of size  $n$ , i.e. the call tree is linear and of depth  $O(n)$ . At each level of the call tree we perform  $O(1)$  work under the above assumptions. The total runtime is  $T(n)=O(n)$ .

<sup>5</sup> This time we are generating two calls on each level of the recursion. The runtime recurrence relation thus is  $T(n)=2*T(n-1)+O(1); T(1)=1$

<sup>6</sup> The call tree will be a binary tree with  $n$  levels (it branches into two subtrees on each level and the problem size reduces linearly). The number of nodes on each level is thus  $2^n$  (starting with 0 at the root) and the total number of nodes up to level  $n$  is  $O(2^n)$ . At each node we are performing  $O(1)$  work. The total runtime is thus  $O(2^n)$ .

- 7 As before, the Master Theorem does not apply. You have to solve this recurrence relation by telescoping. Another technique that is always an option is to guess the solution and then to prove it by induction.

# 3.5

## Dynamic Programming

### a quick introduction

#### A seemingly simple problem

Imagine your task is to write the control program for a new vending machine. While this may at first not appear as a particularly demanding intellectual exercise, it will turn out to be trickier than expected!

We want to look at one particular sub-task that needs to be solved. This is the algorithm that determines how to give change. As we would like to keep a minimal amount of coins in the machine, we will need to develop an algorithm that pays out a given amount in the smallest number of coins possible.

For example, to pay out \$7 when the available coin denominations are \$1, \$2, \$5 we should only use two coins: \$7=\$5+\$2.

In general form, the coin system at hand has  $n$  different denominations  $d_1, \dots, d_n$ . We assume that we have an unlimited supply of each coin type. The problem we have to solve is how to pay out a given amount  $C$  using the minimum number of coins.

A most naive way to approach the problem is to determine the coins to use step by step and to always use the highest valued coin possible.

```

Algorithm change(C)
  input: an integer C (the amount to be paid out),
  output: the coins to be used to pay out C
  assumptions:
    the available denominations are d1, ..., dn
    an unlimited number of coins are available
      for each denomination

  let coins be an empty list
  repeat until (C=0) begin
    let temp=0
    foreach d in [d1, ..., dn]
      if (d<=C and d>temp) then let temp := d
    let C := C - temp
    append temp to coins
  end
  return coins
end.

```

Clearly, this would solve the above example correctly and it determines the coins to be used very efficiently because there is no search involved.

[What type of algorithm is the method suggested above?](#)<sup>1</sup>

Let us investigate whether this algorithm is safe to use.

### [Does this algorithm work in any possible coin system? <sup>2</sup>](#)

The greedy approach will work for almost all monetary system that exist in reality. But how can we make sure that the algorithm chooses the minimal number of coins for any possible coin system?

## A brute force solution

As usual, for some inspiration a naive solution is a good starting point. Obviously, generating all possible combinations of coins that sum up to  $C$  and checking which of these has the smallest number of coins is guaranteed to solve the problem correctly.

We could implement this algorithm elegantly using recursion. Instead of explicitly generating all sets first, the recursive algorithm can do this implicitly "on-the-fly". The idea is simply for each denomination to test whether using it vs not using it at a given step yields the better result.

To simplify the discussion we will only let our algorithm compute *how many* coins are needed (rather than explicitly determining which ones have to be used). This does not really change the problem as we still have to determine implicitly which ones to use, it will simply allow us not to worry about recording the denominations used. In this way it will allow us to focus on the important design aspects and to write the relevant aspects of the algorithm more compactly.

```

Algorithm change(C)
    input: an integer C (the amount to be paid out),
    output: the minimum number of coins needed to pay out C
    assumptions:
        the available denominations are d1, ..., dn
        an unlimited number of coins are available
        for each denomination

    if (C=0) return 0
    else begin
        min = +infinity
        foreach d in [d1, ..., dn] begin
            if (change(C-d)+1 < min) then min = change(C-d)+1
        end
    end
    return min
end.

```

Note that the algorithm calls itself recursively. For each denomination, it reduces the amount required by this denomination and computes recursively how many coins are required to change the reduced amount. It then compares this result to all other possibilities of taking out another denomination first.

We can easily extract a compact (recursive) formula from this algorithm that computes the minimum number of coins needed:

$$\text{change}(c) = \begin{cases} 0 & \text{if } c = 0 \\ \min_i(1 + \text{change}(c - d_i)) & \text{otherwise} \end{cases}$$

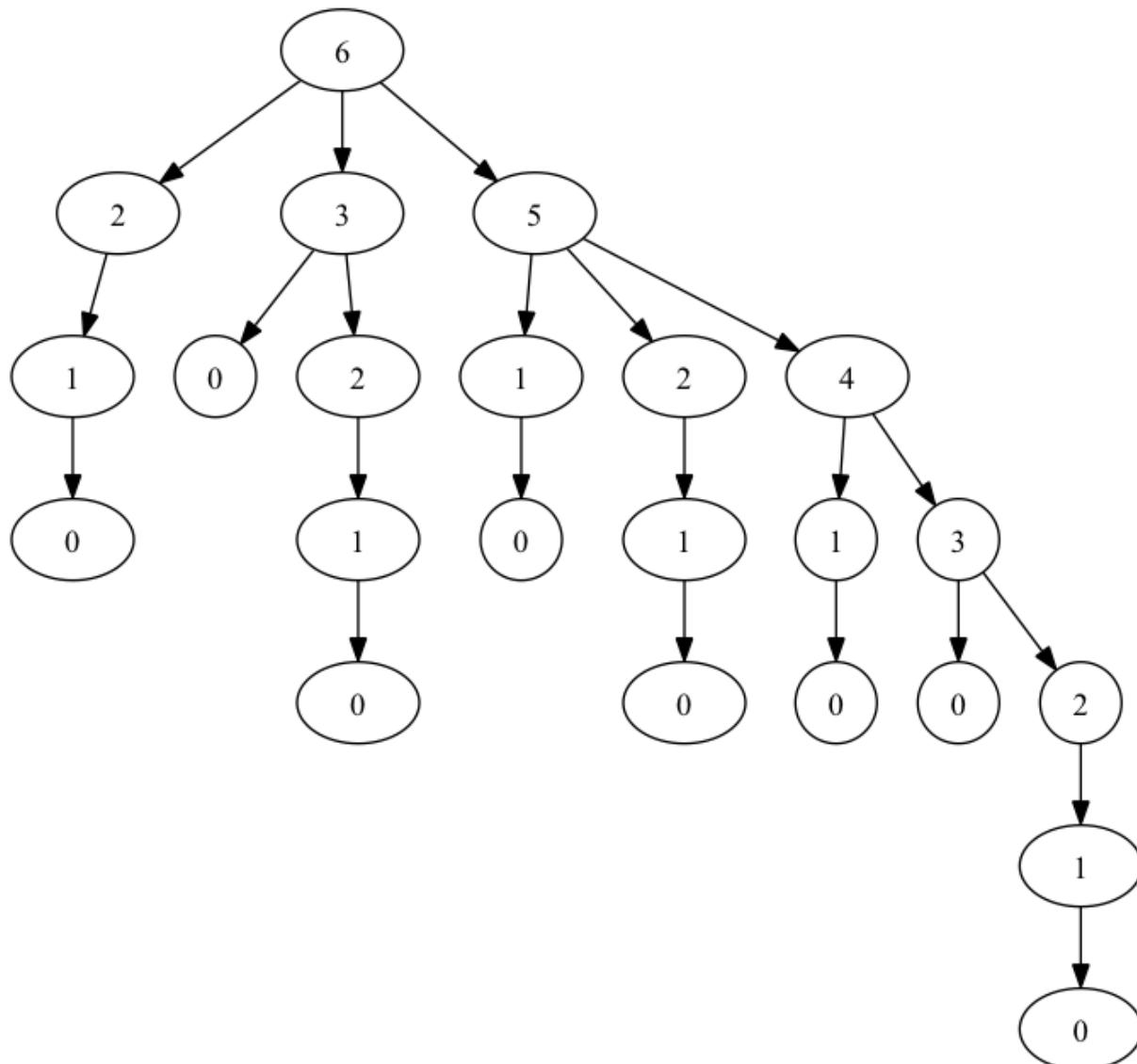
This is elegant, but is it an efficient way to solve the problem? Unfortunately, the answer is no.

[To analyze the runtime complexity of this algorithm, what is the relevant variable that measures problem size?](#)<sup>3</sup>

[How many possible combinations of denominations must the above algorithm test?](#)<sup>4</sup>

To convince yourself of this, consider a coin system that has  $n$  different denominations with  $d_{\max}$  being the highest one. To pay out the amount  $C$  you will clearly need at least  $m = \frac{C}{d_{\max}}$  coins (and most likely more). Consider just these  $m$  coins. As there are  $n$  denominations (possible coin types) you already have  $n^m$  combinations to consider. And this is just a lower bound.

We can easily see how the problem manifests in the recursive call tree of the algorithm. Consider the tree below for the example that we started with:



## An efficient solution: Dynamic Programming

We are obviously performing a lot of redundant computations! Many calls, such as `change(3)` are evaluated multiple times! To construct a more efficient solution we need to avoid these redundant double computations. The basic idea is not complicated at all. Consider the multiple calls to `change(3)`. We need the results of these calls within the computation of `change(4)`, `change(5)`, and so forth... What if we computed `change(3)` first and stored the solution so that we do not need to recompute it? This would clearly avoid the double computations.

*The core idea is to compute the nodes of the call tree above bottom-up rather than top-down and to store intermediate results so that they can be reused without having to recompute them.*

[What type of data structure would you use to store the intermediate results?](#)<sup>5</sup>

We initialize `A` at all spots where the solution is trivial. These typically correspond to the base cases of the recursion. All other positions are computed in increasing order, reusing values that have been computed previously.

```

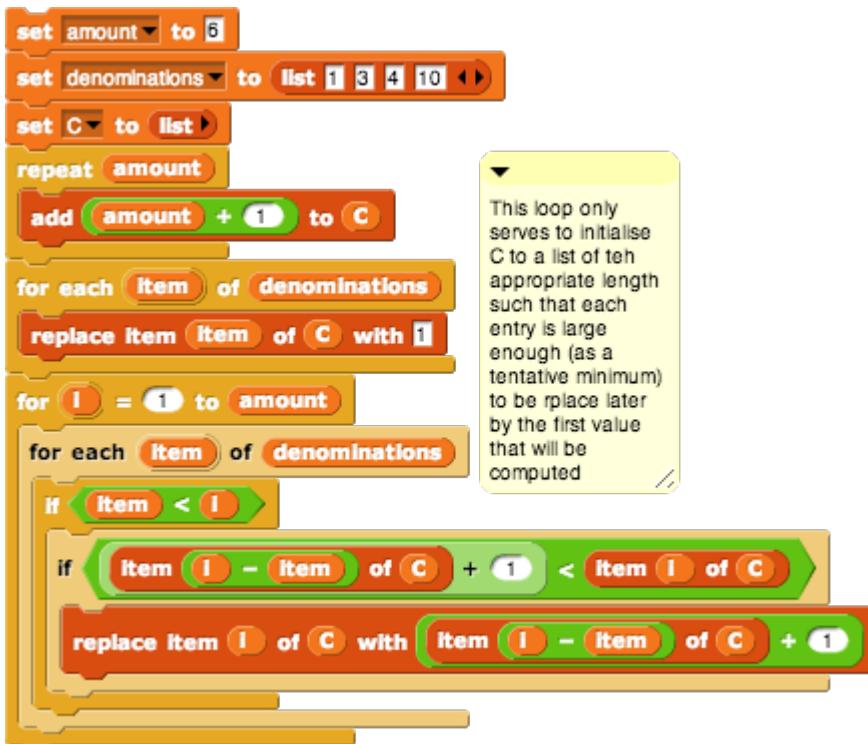
Algorithm change-dp(C)
  input: an integer C (the amount to be paid out),
  output: the minimum number of coins needed to pay out C
  assumptions:
    the available denominations are d1, ..., dn
    an unlimited number of coins are available
      for each denomination

  A[0]=0
  for i from 1 to C
    foreach d in [d1, ..., dn] begin
      if (d < C)
        if (1+A[i-d] < A[i]) then A[i] := 1+A[i-d]
    end
  end
  return A[C]
end.

```

## Implementation in Edgy

below is the full implementation of the coin change problem in edgy. It directly mirrors the pseudo-code given above.



## Runtime complexity of the coin change algorithm

The runtime complexity of the dynamic programming algorithm is easily determined: We have two nested loops and perform only a constant time computation in the innermost loop body. The outer loops runs from 1 to  $C$  and the inner loop executes once for each of  $n$  denominations.

[Based on these observations, what is the asymptotic runtime complexity of change-dp?](#)<sup>6</sup>

These are massive savings over the runtime of the naive algorithm which was exponential in  $C$ !

## The general schema

This "bottom-up" computation is an instance of a algorithm design pattern called *Dynamic programming*.

- Dynamic Programming is often useful where we have a straight forward recursive solution to a problem, but the direct recursive implementation is inefficient because it leads to redundant computations.
- The redundant computations are caused by what is often called "overlapping sub-problems": The computations that need to be performed to compute the solutions of sub-problems contain identical parts so that these are replicated in the call tree.
- The terminology "overlapping sub-problems" clarifies an important distinction between dynamic programming and divide-and-conquer. Dynamic programming is often useful where the problem can only be split into overlapping subproblems. Divide-and-Conquer is usually only useful when the problem can be split into non-overlapping (independent) sub-problems.
- Implementing a dynamic programming solution for a recursive definition essentially amounts to computing the call tree bottom up in some order that avoids double-computation and storing

intermediate results for reuse. To define the DP algorithm from a recursive definition two fundamental decision have to be made:

- What is an appropriate storage structure? Typically the answer is a  $k$ -dimensional matrix or array.
- What is the appropriate order to traverse (or generate) the entries? This needs to ensure that all entries have been computed before they are needed in other computations.
- A subproblem of the traversal order is the question which results can be determined trivially and used to initialize the storage structure (typically the base cases of the corresponding recursion).
- In many cases the direct recursive implementation will execute an exponential number of calls.
- Dynamic programming usually has polynomial runtime complexity. The basic algorithm structure typically consists of a number of nested loops. A common form is an algorithm that fills a one dimensional array step by step and at each step all previously computed elements are considered. The complexity of such an algorithm is  $O(n^2)$ .
- Dynamic programming is often used for problems where we find a solution among a large number of candidate solutions that is in some sense optimal.

## Variation: number of ways to give change for the sum

Let us consider a simple variation of the problem: Instead of computing the minimum number of coins needed we now want to compute the number of different coin combinations that can be used to pay out the amount.

The recursive solution for this problem is only a minor variation on the above and likewise very simple to obtain:

$$\text{change}(c) = \begin{cases} 1 & \text{if } c = d_i \text{ for some } d_i \\ \sum_i (\text{change}(c - d_i)) & \text{otherwise} \end{cases}$$

The required change to the algorithm directly mirrors the change in the formula. We leave it as an exercise for you to rewrite the above algorithm so that it computes the number of ways change can be given.

Note that if your coin system only has two denominations \$1 and \$2, the solution is exactly the  $c$ -th [Fibonacci number](http://en.wikipedia.org/wiki/Fibonacci_number) ([http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number))!

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

So the computation of Fibonacci numbers is clearly a candidate for dynamic programming! Here is a dynamic programming algorithm that computes the Fibonacci numbers up to  $n$ .

```
Algorithm fib-dp(N)
  input: an integer N>0
  output: the N-th Fibonacci number
```

```
F[1]=1
F[2]=1
for i from 3 to N
  F[i]=F[i-1]+F[i-2]
```

```

end
return F[N]
end.
```

It is well known that the closed-form solution for the Fibonacci numbers is

$$F_n = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

with the [Golden Ratio](http://en.wikipedia.org/wiki/Golden_ratio) ([http://en.wikipedia.org/wiki/Golden\\_ratio](http://en.wikipedia.org/wiki/Golden_ratio))

$$\phi = \frac{1+\sqrt{5}}{2}$$

This fact can be used as a simple proof that the recursive algorithm has exponential runtime complexity (executes an exponential number of calls), since the recursive formula does not only describe the number of ways to pay out the amount, but also the number of recursive calls made ("F(n) requires all the recursive calls that are needed to execute F(n-1) plus all the recursive calls needed to execute F(n-2)")

## Another example: cutting for profit

Let us apply the same technique for a different problem. We want to solve a planning problem for a company that buys steel rods, cuts them into varying lengths, and resells them again. The question that arises is how to best cut a rod of a given length  $n$  in order to achieve the maximum profit.

The answer is not entirely straight forward because the price for each rod length is set individually, i.e. it is not a simple per meter price. Otherwise the problem would be rather boring: it clearly would not matter how to cut!

The company only sells rods in multiples of full meters, and the cost for cutting is negligible so that it does not need to be taken into account.

You have to write an algorithm that determines the maximum total selling price that can be achieved for the pieces cut from a rod of length  $n$ . Our algorithm is given as input the total length  $n$  and the prices  $p_i$ , where  $p_i$  is the price charged for a single piece of length  $i \in 1 \dots n$ .

To simplify the discussion, we only explicitly compute the maximum profit we can make. Our algorithm will not output the cutting pattern itself!

As before, a recursive formula for the maximum profit that can be achieved is simple to find. The formula essentially considers at each recursive call all possible ways of cutting one piece off the rod.

$$p(n) = \begin{cases} 0 & \text{if } n = 0 \\ p_i & \text{if } n = l_i \text{ for some } i \\ \max_{i \in \{1 \dots k | l_i \leq n\}} p_i + p(n - l_i) & \text{otherwise} \end{cases}$$

We can again use a one-dimensional matrix or array for the intermediate results, and at index  $i$  of the matrix we will store the maximum profit that we can achieve from a rod of length  $i$ . The algorithm uses two nested loops. The outer loop considers rods of increasing length until we reach the target length, and the inner loop considers all possible ways of cutting a single segment from the currently considered

length. If we cut one segment we end with two shorter segments for which we know the maximum profit already, so these two profits can simply be added up.

```

Algorithm cutting-dp(L)
    input: an integer L (the rod length),
    output: maximum profit from selling a rod of length L in segments
    assumptions:
        the profits for the lengths of rods that are sold uncut are
        p1, ..., pk and their corresponding lengths are l1, ..., lk
    for i from 0 to L: P[i] := 0
    for i from 1 to k: P[lk] := pk
    for i from 1 to L
        for j from 1 to k begin
            if (lj < i) AND (pj + P[i - lj] > P[i]) then P[i] := pj + P[i - lj]
        end
    return P[L]
end.

```

## Dynamic Programming Graph Algorithms

Dynamic programming is also used with great benefit in graph algorithms. We have indeed already looked at the two best-known examples: the computation of the *transitive closure* and the *all-pairs shortest path problem*.

### Transitive closure algorithm: The Floyd-Warshall Algorithm

Recall that in graph theory the transitive closure of a DAG  $G$  is an augmented version of  $G$  that makes the reachability in  $G$  explicit. It contains an edge from  $a$  to  $b$  for every pair of nodes  $a, b$  where there is a directed path from  $a$  to  $b$ , ie. where  $b$  is reachable from  $a$ .

A trivial way to compute the transitive closure would be to deploy either DFS or BFS for each possible start node and to insert an edge from the start node to each node that is visited. Since both traversal methods visit each node that is reachable from the start node this is guaranteed to generate the transitive closure.

Let's view this problem from the perspective of dynamic programming.

As usual, we start from a recursive definition of the problem for which we then find a suitable "bottom-up" implementation. The definition is simple: Node  $C$  is reachable from node  $A$  if either there is an edge from  $A$  to  $B$  or if there is an intermediate node  $C$  such that  $C$  is reachable from  $A$  and  $B$  from  $C$ .

As before we could implement this directly as a recursive algorithm, but as before this would be very inefficient because of redundant calls, and a dynamic programming solution is preferable. Since we want all partial results (all paths!) anyway, it is useful to use the graph itself as the storage structure for all the intermediate results. We simply need to try all the possible pairs with all intermediate nodes.

```

Algorithm transitive-closure-DP(G)
    input: a graph G
    output: the transitive closure of G

```

```

let G1 := G
for hop in AllNodes
    for start in AllNodes
        for end in AllNodes
            if not (start=end or start=hop or hop=end)
                if ( edge(start, hop) exists in G1
                    and edge(hop, end) ) exists in G1 then
                        insert edge(start, end) into G1
end.

```

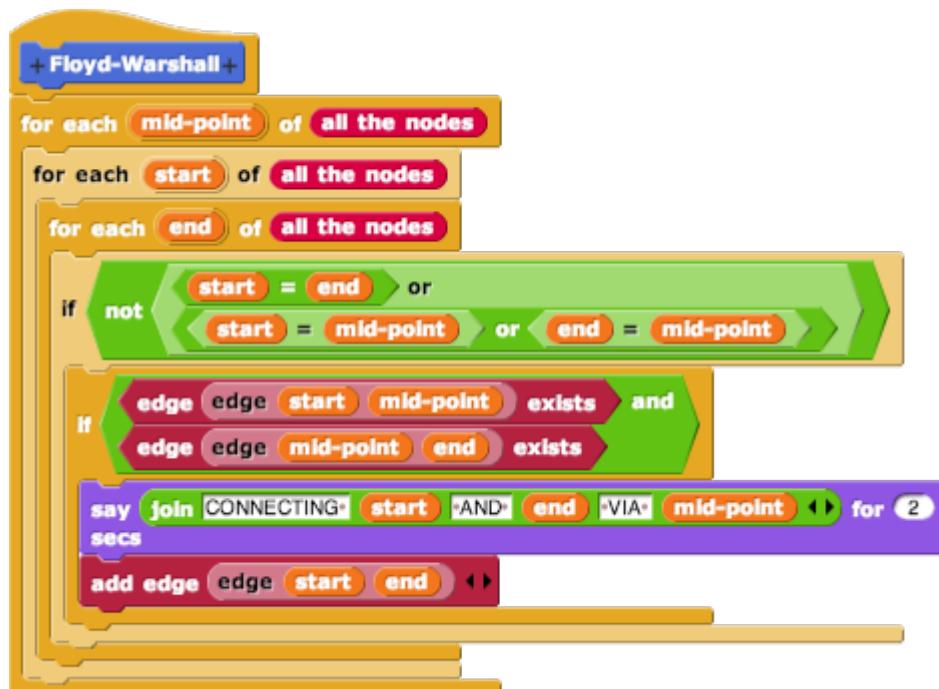
We have arrived at [Warshall's transitive closure algorithm](#)

(<https://www.alexandriarepository.org/reader/vce-algorithmics/53766>)!

The stopping condition deserves a further look. Why are we justified in testing every hop only once? We are effectively iterating the test for each pair of nodes  $n$  exactly times. Clearly any cycle-free path can have at most  $n$  nodes on it, and all paths tested in each iteration ( $i+1$ ) will be at least one node longer than the paths previously considered in iteration  $i$ . Thus, no new paths can be generated after  $n$  iterations.

## Implementation in Edgy

Below is the full implementation of the coin change problem in edgy. It directly mirrors the pseudo-code given above.



## Computational Cost of Warshall's Algorithm

Let us have a brief look at the runtime complexity of the transitive closure algorithm. As often is the case for Dynamic Programming algorithms, it is fairly simple to determine.

[What is the asymptotic runtime complexity for the Warshall algorithm for transitive closures?](#)<sup>7</sup>

We could also have solved the transitive closure problem as outlined above with  $n$  successive calls to a DFS or BFS. As we had determined in an earlier module, the runtime complexity of DFS and BFS is  $O(n+m)$  for a graph with  $n$  nodes and  $m$  edges. How do the two approaches compare? There can be at most  $O(n^2)$  edges in a simple graph, thus  $m=O(n^2)$ . Deploying DFS or BFS  $n$  times thus takes time

$$n \cdot O(n + m) = O(n^2 + mn) = O(n^2 + n \cdot n^2) = O(n^3)$$

Both approaches have the same runtime complexity! However, due to its simpler structure, Floyd-Warshall's algorithm will generally run faster if the graph does indeed have close to  $n^2$  edges. However, if the graph is sparse, i.e. if it has significantly fewer than  $n^2$  edges, it may be faster to run DFS/BFS repeatedly.

## All-pair shortest path: Floyd's algorithm

We have already seen how the same idea can be used to compute the shortest paths between all pairs of nodes in a graph. The structure of the dynamic programming algorithm for this problem is almost identical to the Warshall algorithm for transitive closures. The fundamental difference is that instead of just inserting an edge we keep checking whether a shorter connection has come into existence in a later iteration. You will recognize this algorithm as [Floyd's algorithm for the all-pair shortest path problem](#) (<https://www.alexandriarepository.org/reader/vce-algorithmics/53768>).

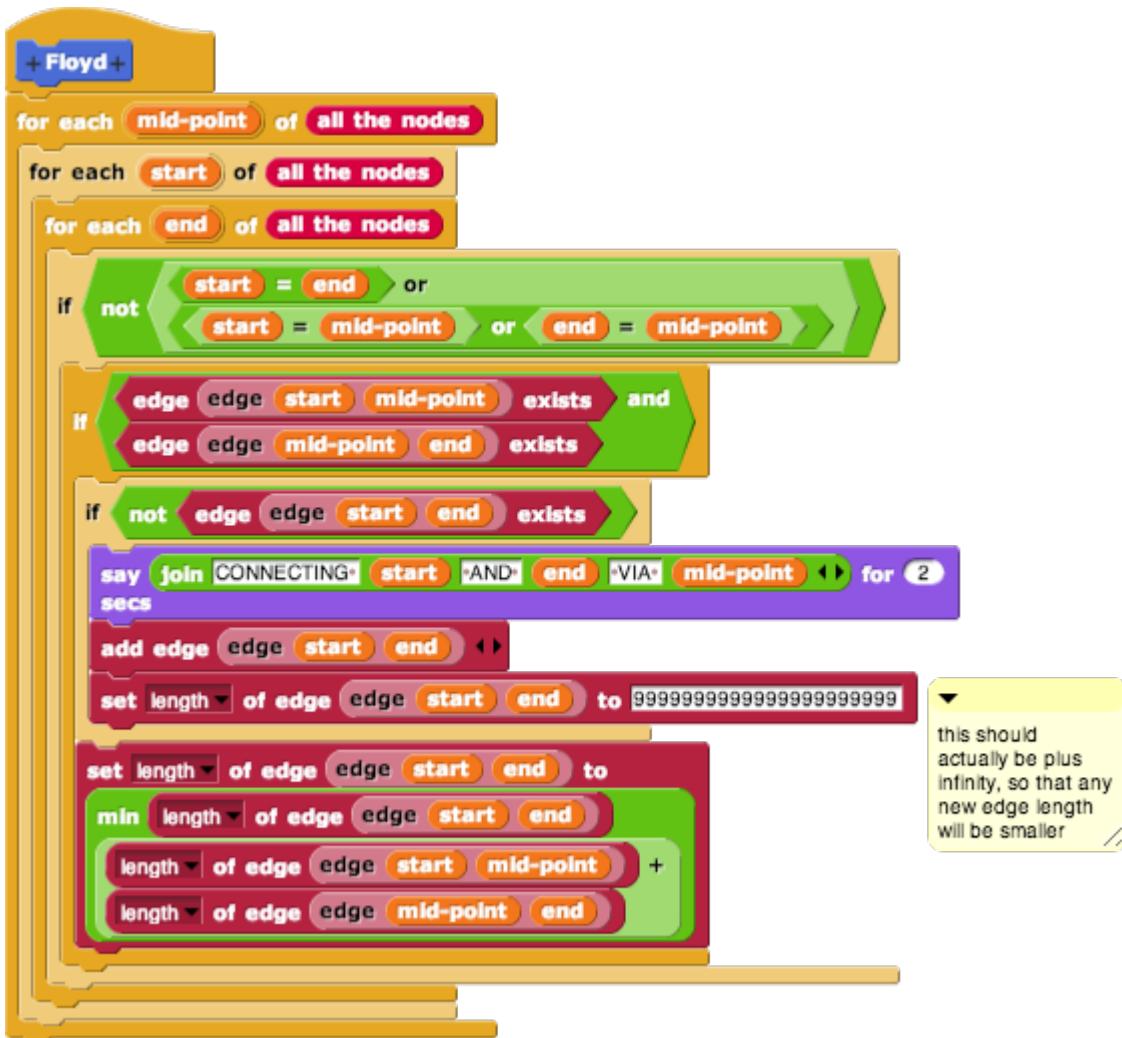
```

Algorithm all-pairs-shortest-path-DP(G)
    input: a graph G
    output: the transitive closure of G
           with shortest path lengths as edge attribute "length"
    let G1 := G
    for hop in AllNodes
        for start in AllNodes
            for end in AllNodes
                if not (start=end or start=hop or hop=end)
                    if ( e1=edge(start, hop) exists in G1 begin
                        and e2=edge(hop, end) ) exists in G1 then
                            if (edge(start, end) does not exist in G1 then begin
                                insert e3=edge(start, end) into G1
                                let length(e3)=+infinity
                            end
                            let length(e3)=
                                min(length(e3), length(e2)+length(e2))
                        end
                    end
    end.

```

## Implementation in Edgy

below is the full implementation of Floyd's algorithm in edgy. It directly mirrors the pseudo-code given above.



## Computational Cost of Floyd's Algorithm

Unsurprisingly, the same considerations as for Floyd-Warshall's algorithm apply to Floyd's algorithm. We are justified in stopping after  $n$  iterations because all possible intermediate connections that are not cyclic have been considered.

The runtime complexity of Floyd's algorithm is obviously likewise  $O(n^3)$ .

We could solve the problem with a different approach, since we know an algorithm that solves the shortest path problem for a single start node: Dijkstra's algorithm. Simply running Dijkstra's algorithm for every possible start node would clearly solve the all-pair shortest path problem.

We had earlier determined that the runtime complexity of Dijkstra's algorithm for a connected graph (if implemented wisely) is  $O(m \log n)$ . Running Dijkstra  $n$  times and assuming  $m=O(n^2)$  edges, the total runtime complexity is

$$n \cdot O(m \cdot \log n) = O(n^3 \cdot \log n)$$

But for large sparse graphs we cannot assume that we have on the order of  $n^2$  edges and would have to compare  $O(n^3)$  with  $O(nm \log n)$ . Thus, for large sparse graphs repeated runs of (a good implementation

of) Dijkstra's algorithm may actually be preferable.

## Memoization

*Nota bene:* Memoization is not formally a part of the study design of VCE Algorithmics.

*Nota benissimo:* It is still useful to know and in many ways easier to understand than full dynamic programming.

There is a "sneaky" way to get around having to devise a Dynamic Programming algorithm for a recursively defined problem and to still avoid redundant computations. This approach is called [Memoization](http://en.wikipedia.org/wiki/Memoization) (<http://en.wikipedia.org/wiki/Memoization>). It essentially amounts to maintaining the original recursive call structure but to explicitly store and reuse the intermediate results while the recursion is executed.

The following algorithm illustrates the technique for the rod cutting problem that we considered above:

```

Algorithm cutting(L)
  input: an integer L (the rod length),
  output: maximum profit from selling a rod of length L in segments
  assumptions:
    the profits for the lengths of rods that are sold uncut are
    p1, ..., pk and their corresponding lengths are l1, ..., lk

  for j from 1 to L: P[j]:=-infinity
  return cutting-main(L)
end.

Algorithm cutting-main(L)
  (* this performs the real computation
     after the initialisation has finished *)
  max = -infinity
  for j from 1 to k begin
    if (lj < L) AND (pj+cutting-memo(L-lj) > max) then
      max := pj+cutting-memo(L-lj)
    end
  return max
end.

Algorithm cutting-memo(L)
  (* this is just an auxilliary function that checks *)
  (* whether this value has already been computed *)
  (* if not it computes it and stores it as an intermediate result *)
  if P[L] <> -infinity then return P[L]
  else begin
    P[L] = cutting-main(L)
    return P[L]
  end
end.

```

There is no real advantage to using memoization instead of an iterative dynamic programming approach. The theoretical runtime complexity of both approaches is the same in these cases, but the simple

iterative structure of the dynamic programming version (rather than recursive calls) makes it faster than the memoization version.

However, memoization has an advantage that is conceptually very neat: It automatically takes care of generating the values in the order in which they are needed without relying on a pre-defined order. Thus the approach comes in handy if for some reason it is too difficult (or impossible) to determine in which order the results have to be computed .

---

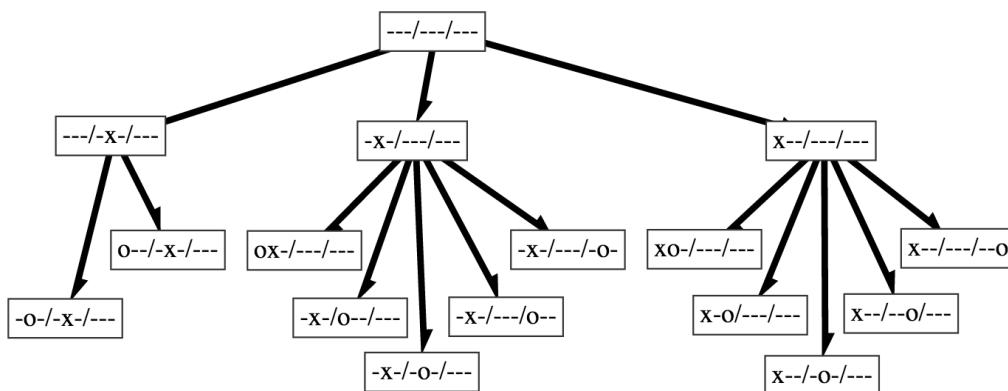
- ☒<sup>1</sup> The algorithm is another example of a *greedy* approach: it tries to make the maximum progress in each step and it never revises a decision previously made. As always we have to be suspicious: greedy algorithms are fast when they work, but they don't always work.
- ☒<sup>2</sup> The answer turns out to be no! Consider a coin system with denominations \$1, \$3, \$4 and  $C=6$ . The greedy algorithm would use three coins ( $\$6=\$4+\$1+\$1$ ) but it could have used only two coins ( $\$6=\$3+\$3$ ). Admittedly, this is an unusual coin system.
- ☒<sup>3</sup> The total mount to be changed.
- ☒<sup>4</sup> You can easily convince yourself that the number of possible combinations that are tested must grow exponentially.
- ☒<sup>5</sup> For our example, a one-dimensional array (matrix) indexed with the amount  $C$  to be changed is the ideal storage structure.  $A[i]$  will store the number of coins needed to pay out the amount  $i$ .
- ☒<sup>6</sup> The total runtime complexity (number of potential updates of the array) is  $O(nC)$ .
- ☒<sup>7</sup> We have three nested loops iterating over all nodes, so for  $n$  nodes we perform  $n^3$  repetitions of the test and assignment in the innermost loop. The total runtime complexity is thus cubic -  $O(n^3)$  .

# 3.6 Game Trees and the Minimax Algorithm

## Deciding which turn to take

Many games involve a series of turns, in which opponents decide on their optimal move. A typical move might involve placing or moving a piece on a board. At each turn, there are one or more moves that could be made. Which one is the best? We need to make a decision, and we can use a special kind of decision tree, called a game tree, to help.

Here's part of a game tree for tic-tac-toe. The root node represents the start state with an empty board: `- - / - - / - -` (three empty rows). The child node `- - - / - x - / - -` represents a board with an X in the middle.

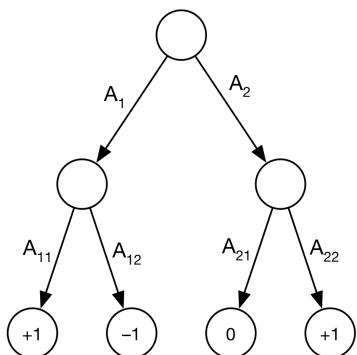


Notice that the above tree only shows three options for the first move by player X, even though there are nine cells where the first piece can be played. Why are only three options shown?

This graph only includes the first two moves. Deeper in the tree, we can expect to find nodes that represent a win, e.g. `ox-/xo-/x-o`, a win by player O. On the previous turn, player X should be able to detect that O can win on the next turn by looking further down in the tree, and choose a move that blocks the win. We can do this systematically using the Minimax Algorithm.

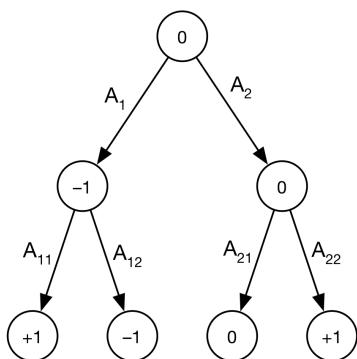
## The Minimax concept illustrated

To understand the core idea of the Minimax Algorithm, consider the following game tree. The game itself consists of just two turns. Player 1 has two choices  $A_1$  and  $A_2$ . For each of these moves, there are two possible responses by player 2.

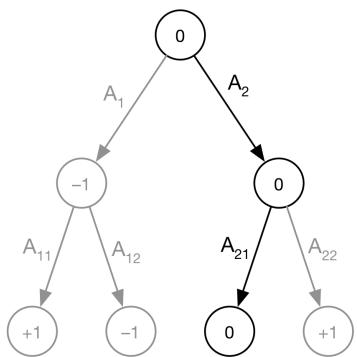


(<https://www.alexandriarepository.org/wp-content/uploads/game-tree-3.png>)

The desirability (or "utility") of each final state is shown as an integer in the bottom row. This number represents the benefit to player 1, as either a win (+1), loss (-1), or draw (0). Assuming that player 2 will act to minimise the benefit to player 1, we can predict player 2's behaviour. Accordingly, we write the minimum value of the pair of leaf nodes on their parent:  $\min(+1, -1) = -1$ , and  $\min(0, +1) = 0$  (see the two internal nodes below). Now, player 1 will decide on a course of action that maximises the result, so we write the *maximum* value of the internal nodes on their parent (the root of the tree below):  $\max(-1, 0) = 0$ .



So player 1 will choose move  $A_2$ , and player 2 will respond with  $A_{21}$ .



Notice how our approach involved working up the tree from the leaves, alternately writing minimum and maximum values on the parent nodes at each level. However, when it came to using the tree, we worked down from the root (the start). In this way, the minimax algorithm gives us unlimited look-ahead. If some branch could only take us to -1 leaves (5 or 20 moves later perhaps) we will never take that branch.

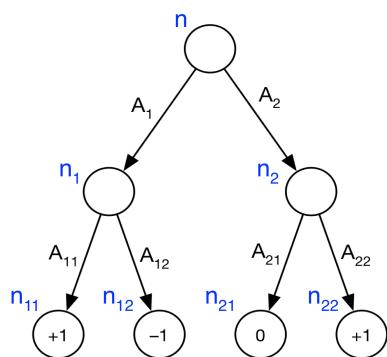
## The Minimax Algorithm

The Minimax Algorithm begins from the root of the game tree, the initial state of the game, e.g. `minimax(root, True)`. The second argument is a Boolean value that indicates whose turn it is. The function recursively calculates the utility of node using the utility of its children.

```
function minimax(node, maximizingPlayer)
    if node is a terminal node
        return the value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, False)
            bestValue := max(bestValue, val)
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, True)
            bestValue := min(bestValue, val)
        return bestValue
```

(The concept of recursion will be covered later in the unit.)

Let's step through the above example using the Minimax Algorithm. We start with Player 1, who is the player who we want to win (the "maximising player"). It will be useful to have a way to refer to the nodes, so here is a version of the game tree with node labels:



```

→ minimax(n, True)
    compute highest value of children
    → minimax(n1, False)
        compute lowest value of children
        → minimax(n11, True)
            the value of n11 is +1
            +1
        → minimax(n12, True)
            the value of n12 is -1
            -1
        lowest value of +1 and -1 is -1
        -1
    → minimax(n2, False)
```

```

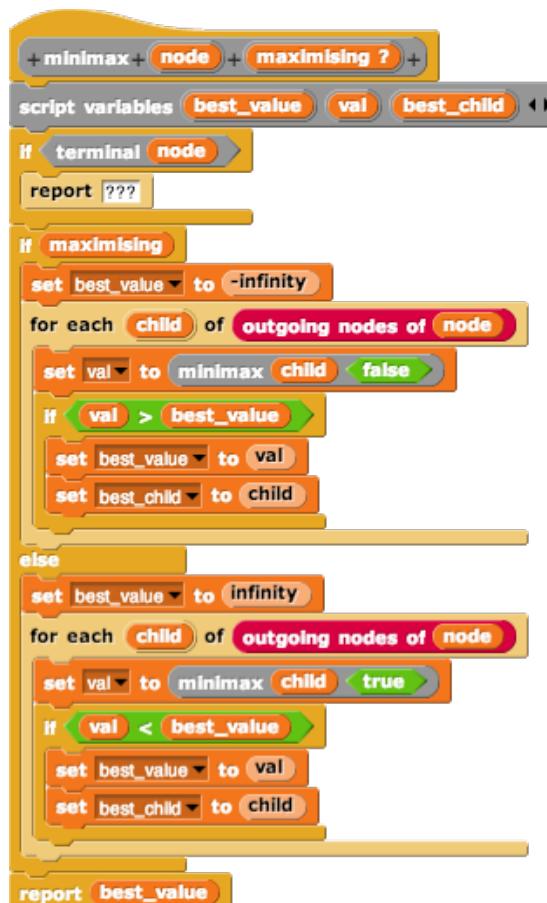
compute lowest value of children
→ minimax(n21, True)
    the value of n21 is 0
    0
    → minimax(n22, True)
        the value of n22 is +1
        +1
lowest value of 0 and +1 is 0
    0
highest value of -1 and 0 is 0
0

```

Using the Minimax Algorithm, we have determined that the best result Player 1 can hope for is a draw. Of course, Player 2 could make a mistake in which case Player 1 might be able to do better. But assuming that Player 2 is playing sensibly, and is as capable as Player 1 in considering the downstream consequences of any move, then it will be a draw.

## Doing it in Edgy

It's not hard to translate the Minimax Algorithm into Edgy code. Here we go a step further to store the utility of a node as an attribute of the node, and to keep track of the best child of a node.



We haven't specified the value to report for a terminal node. We assume there is some other process that does the calculation and stores a utility value in a node attribute. For instance, it might examine the tic-tac-toe board and determine whether it is a win for x, a win for o, or a draw (a completed board where neither player is a winner).

(Is it possible to have a board which is a win for both x and o? If so, is this a problem?)

## Types of game

We can classify games according to the following system. Can you identify the category of game that we are able to model using a game tree?

**deterministic      chance**

**perfect information** chess      backgammon

**imperfect information** battleship scrabble

Would it be true to say that a game like tic-tac-toe is deterministic? What assumption do we make about the players? Is there any room for chance in the way the game is played? Explain.

---

## **3.7 Computationally Hard Problems**

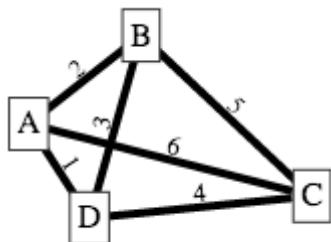
- [3.7.1 Travelling Salesman Problem](#)
  - [3.7.2 NP-Hard Problems: Soft Limits of Computation](#)
  - [3.7.3 Heuristics](#)
  - [3.7.4 Heuristics for the Graph Colouring Problem](#)
-

## 3.7.1 Travelling Salesman Problem

### The Travelling Salesman Problem

Let's consider a route-finding problem, in which a salesman needs to visit a number of cities. Imagine that he has a map with these cities and the distances between each pair of cities that are directly connected by a road. He needs to find the shortest possible route that visits each city exactly once and then returns to his start city.

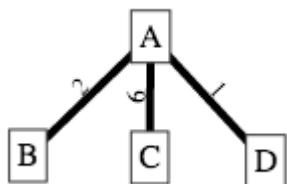
Well, this sounds quite similar to some of the other path-finding problems that we've encountered before. So, let's first create a weighted graph with nodes representing the cities and weighted edges representing the roads that connect them and their distances. Here's an example with four cities, A, B, C and D:-



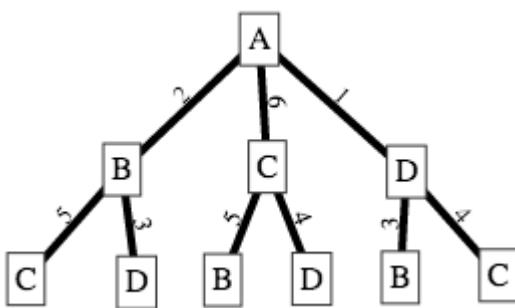
### Naive Solution

Can you think of a way to solve this?

The brute-force solution is one naive place to start. We simply try all possible routes. So, if the salesman starts at A, he has 3 options A-B, A-C, or A-D.



And then from each of these cities, he has two further unvisited cities that he could visit:-



... and so on.

We can write an algorithm to do this:- it will simply find all possible routes, and take the shortest one.

```

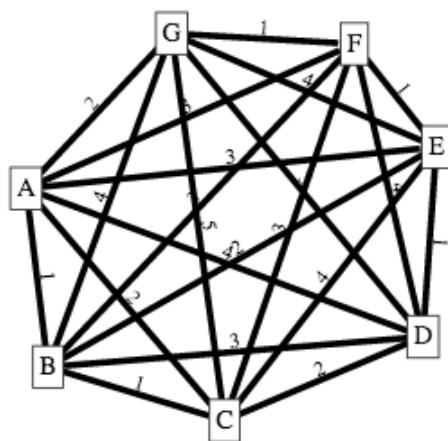
function brute_force_tsp(node, path, home_node, min_path):
    inputs: node - the next city to add to the path
            path - a list of cities in the current path (or route)
            home_node - the start and end city for the path
            min_path - the completed path of minimum length

    add node to path
    if path contains all nodes in the graph and there is an edge between node
    and home_node:
        add home_node to path
        if length of path < length of min_path:
            set min_path = path
        return min_path
    else:
        foreach neighbour of current_node:
            if neighbour is not in path:
                set min_path = brute_force_tsp(neighbour, path, home_node)
    return min_path
  
```

So, how long will this brute-force algorithm take to run for our example? Let's count the number of times the basic operation (creating a possible route) is performed. For our example graph with 4 cities, our algorithm will need to create routes from the start city to 3 possible next cities, then for each of these it will create a route to 2 further cities, and then to the 1 remaining unvisited city. So, it will create  $3 \times 2 \times 1 = 6$  possible routes, and take the shortest of these to be the best route.

The 6 possible routes are A-B-C-D-A, A-B-D-C-A, A-C-B-D-A, A-C-D-B-A, A-D-B-C-A and A-D-C-B-A. There are two shortest routes of length 12: A-B-C-D-A and A-D-C-B-A. And, if you look closely you may notice that these two shortest routes are simply the same route in reverse.

So, what about a larger example? How many possible routes would a brute force algorithm need to examine in this case? Let's take a look at an example with 7 cities:-



How many routes would the brute force algorithm need to examine when it is run on this graph?

If we start at node A, we have 6 possible next cities to visit. Then, for each of these 6 cities, we have a further 5 unvisited cities.... and so on. And, this gives a total of  $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$  possible routes that our brute force algorithm would need to examine. You can see that, even though we only added 3 extra cities into our graph, the number of possible routes has increased significantly.

So now, let's think about the time complexity of the brute-force approach. From the examples above we have the following results:-

4 nodes:  $3 \times 2 \times 1 = 6$  possible routes

7 nodes:  $6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$  possible routes

Can you see a pattern? Remember that the time complexity is often expressed relative to the size of the input (the number of nodes in this case). Is there a way to represent that number of possible routes as a function of the number of cities (or nodes) in the graph?

Well, it turns out that the brute force algorithm has a time complexity of  $O(n-1!)$ . That is, for a graph with  $n$  nodes, the brute force algorithm will need to check  $n-1 \times n-2 \times n-3 \dots \times 2 \times 1$  (or  $n-1$  factorial) possible routes.

We have seen that even for a small graph, the number of possible routes get's quite large. For a slightly bigger graph containing, say, 11 cities, we would need to check  $10! = 3,628,800$  possible routes. This is starting to look unmanageable. As we add cities to our graph, we get a combinatorial explosion in the number of possible routes:- and although, it is possible, in theory, to check all possible routes, the number of routes is so enormous that it is completely impractical, not only at present but for any conceivable computer in the future.

## Other Possible Approaches

At first glance, the travelling salesman problem looks very similar to the problem of finding a minimum cost spanning tree (MCST) for a given graph. Recall the problem of finding a spanning tree involves finding a subgraph that connects all the edges together and is a tree. And, if we assign weights to the edges, then the MCST, is the spanning tree with edge weights that sum to less than those of any other spanning tree.

We've seen that there are lots of algorithms for finding the MCST in linear time - for example Kruskal's algorithm or Prim's algorithm. So, could we use one of these algorithms to solve TSP? Well, the MCST and TSP problems are subtly but very importantly different. While both aim to find minimal cost subgraphs of a graph which connect all vertices, a MCST is a tree, where the Travelling Salesman Problem aims to find a path or cycle (a route in which the tree is not allowed to branch). This restriction on the tree results in a much harder problem and for this reason, the algorithms that can be used to efficiently solve the MCST, are not suitable for the TSP.

In fact, the Travelling Salesman Problem has been studied by numerous researchers for more than 60 years, and no one has yet found an exact solution that runs in a reasonable amount of time. The Travelling Salesman Problem belongs to a group of problems in computer science for which there is no known "efficient" solution. But, what exactly do we mean by an "efficient" solution? An algorithm is considered "efficient" if its time complexity grows as a polynomial function of the size of the input (such as  $n$ ,  $n^2$  or  $n^3$ ). We will discuss this in more detail in the next module.

## Hard Problems in Computer Science

The travelling salesman problem is not just applicable to route-finding between cities. It has numerous different applications such as vehicle routing, the order-picking problem in warehouses, the manufacture of microchips and DNA sequencing.

But, why are we learning about a problem that cannot be solved efficiently? Well, when you come across a problem in computer science, it is useful to understand whether it *can* be solved efficiently. In the next few modules, we are going to look at more of these "hard" problems, how they are defined and some strategies for finding good (if not optimal) solutions to them.

---

## 3.7.2

# NP-Hard Problems: Soft Limits of Computation

## Easy versus Hard Problems

During this course, we have seen various problems and algorithms to solve them. The vast majority of these have been solvable in polynomial time (or less). For example, Dijkstra's algorithm for finding the shortest path between two nodes in a graph with positive-weight edges or Prim's algorithm for computing the Minimum Cost Spanning Tree of a weighted graph.

However, we have also seen the Travelling Salesman Problem:- a problem for which there is no known polynomial-time solution. And, in fact, there are a number of other problems for which this is also the case. Broadly speaking, the difference between problems that can be solved in polynomial-time and those that cannot (to our knowledge), corresponds to the difference between computationally tractable ("easy") and intractable ("hard") problems.

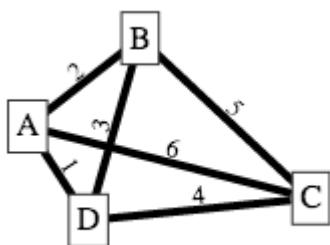
So, if you were faced with a problem that you hadn't seen before, how would you know whether it is tractable? How can you create an appropriate solution to it? Bear in mind that "hard" problems are not unsolvable, but may require the use of alternative strategies to solve in reasonable time. We're going to look at how we define "easy" and "hard" problems in computer science, how we can go about identifying hard problems and where we define the boundary between the two categories.

## Complexity Classes: P and NP

Let's look at some more formal definitions of easy and hard problems. In computer science, we divide problems with related time-complexity into complexity classes. The first complexity class which we're going to look at is called **P**. It contains what we can think of as "easy" problems - those that can be solved in polynomial time. That is, problems for which there is a solution, whose worst case complexity is polynomial time or less (e.g.  $n$ ,  $n^2$ ,  $n^3$  etc, where  $n$  is the size of the input). So, problems in **P** include the examples that were mentioned above:- finding the shortest path between two nodes in a graph with positive-weight edges, or finding the Minimum Cost Spanning Tree of a weighted graph.

Another complexity class is the a group of problems for which a given solution can be checked for correctness in polynomial time. This means that, if we are given a solution to a problem in **NP**, there is an algorithm that can verify the validity of this solution in polynomial time. This group of problems is known as **NP**. Clearly, the class **NP** contains all the problems in **P**, since any solvable problem in polynomial time can also be checked in polynomial time.

An example of a problem in **NP** is the decision version of the Travelling Salesman Problem:- where, given a length  $x$ , the task is to decide whether the graph has any tour shorter than  $x$ . Given the previous example network of cities, as shown below, and a target tour length of 13:-



... we can write an algorithm to check that the solution A-B-C-D-A is a valid tour with length less than 13. The algorithm would simply need to iterate through the cities in the tour checking that each city is visited exactly once, that edges exist connecting all consecutive cities visited, and calculating the tour's length to check whether it is less than x. Such an algorithm would have a time complexity of  $O(n)$ , where n is the number of cities in the network, as it iterates through every city, .

#### Why are they called P and NP?<sup>1</sup>

So, we know that all problems in **P** are also in **NP**. However, people are still uncertain about the opposite - are all **NP** problems also in **P**, or do some **NP** problems exist that are not in **P**? In other words, does **P** = **NP**? Do any problems exist that can be verified, but not solved in polynomial time? Well, we don't know. This remains one of the most important open questions in computer science. Despite decades of research, no problem has been proven to be in **NP** but not in **P** (that is can be verified but not solved). For example, although there is no known polynomial time solution for the 3decision version of the TSP, no one has proven that no such solution exists.

Although we do not know for sure, most scientists and mathematicians think that there are problems in **NP** that do not have polynomial time solutions (i.e. that **P** ≠ **NP**). It is hard to believe that there may be a simple trick that no one has yet discovered in decades of research, that will allow "hard" problems to be solved in polynomial time. And, finding a solution to some problems in **NP**, seems intuitively to be much harder than verification.

## NP-Complete Problems

We know that there are numerous problems for which no known polynomial-time solution exists. So, when you encounter a problem, how do you know whether it is one of these problems and whether you should even attempt to find an efficient solution? Well, there is a subset of **NP** problems that are known to be the "hardest" problems in **NP**, for which no known polynomial time solutions exist. These are called **NP-complete** problems. If we can show that a given **NP** problem is "at least as hard" as any other **NP** problem, then we know that this problem is **NP-complete**. To show that a problem is "at least as hard" as a known **NP** problem, we use a technique called "reduction".

So, what do we mean by "reduction"? A problem  $P_1$  can be reduced to  $P_2$  if you can transform an instance of  $P_1$  to an instance of  $P_2$ , solve  $P_2$ , and then map the answer back to the original problem. For example, if we want to solve the problem of finding the median of a list of numbers, we can reduce this to the problem of sorting the list of numbers. To do this, we can simply sort the list, then find the middle element of the sorted list and this middle element is the solution to the problem of finding the median.

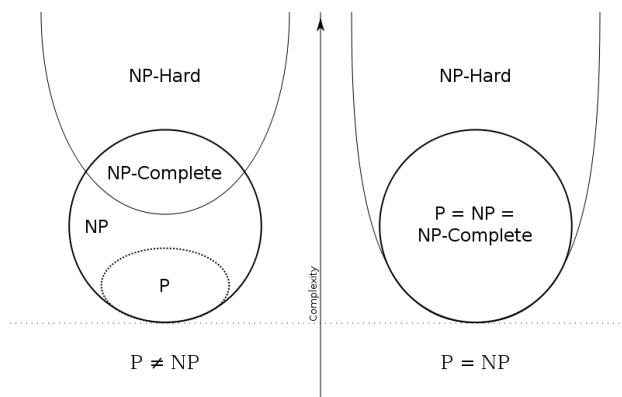
More formally, we can say that a problem,  $P'$ , is **NP-complete** if,

- $P'$  is in **NP**, and
- every other problem in **NP**, can be reduced to  $P'$  in polynomial time (or  $P'$  is **NP-hard** - see below).

We have seen above how to check whether a problem is in **NP**, which covers the first condition of **NP-completeness**. But how can we show that every other problem in **NP** can be reduced to a given problem? Well, we can take advantage of the transitivity of polynomial reductions, and use a known **NP-complete** problem in place of 'every other **NP** problem'. So, we would need to show that a known **NP-complete** problem can be reduced in polynomial time to the problem we are trying to solve. There are hundreds of known NP-complete problems.

## NP-Hard Problems

But what about hard problems that are not in **NP**? For example, the optimisation version of the Travelling Salesman Problem - in which we need to find the tour of minimal length (not just a tour of less than a given length). Such problems are known as **NP-hard** problems, and satisfy only the second condition of **NP-completeness** stated above. These are problems that are "at least as hard as the hardest problems in **NP**". More formally, this means that there is a reduction from every problem in **NP** to a given **NP-hard** problem. Or equivalently that there is a reduction from a known **NP-complete** problem to a given **NP-hard** problem. Unlike **NP-complete** problems, and despite their name, **NP-hard** problems do not have to be in **NP**. The diagram below provides an illustration of how the various classes of problems which we have discussed are related.



There are quite a number of important problems in computer science that are known to be **NP-hard**. If you can show that a problem is **NP-hard**, that is strong evidence of computational intractability. This doesn't mean that there is no solution to the problem, but rather that there is no known efficient means of finding such a solution. Understanding whether a problem is **NP-hard**, means you can have realistic expectations about the efficiency of a solution, and you can use appropriate strategies to find a solution. We'll be looking at some strategies for solving **NP-hard** problems in the next few modules.

<sup>1</sup> Well, **P** obviously stands for polynomial time. But what about **NP** (a common abbreviation for "no problem")? Well, **NP** actually stands for "non-deterministic polynomial". This means that a solution to an **NP** problem can be found in polynomial time by a special (and quite unrealistic) kind of algorithm called a non-deterministic algorithm. Such an algorithm would have the power to always make correct choices at every step of a solution. Solving the problem with a non-deterministic algorithm, is equivalent to verifying a given solution.

## 3.7.3 Heuristics

We've seen that there are quite a number of problems in computer science that are NP-hard. In fact, you are very likely to encounter them in your own projects. And, when you're faced with a problem that is NP-hard, even the best known solutions may not be efficient. We saw that even finding the solution to a reasonable-sized Travelling Salesman Problem (TSP), of say 100 cities, is beyond the capabilities of any conceivable computer in reasonable amount of time. So, should we just give up and say the problem is too hard? What would the salesman do then? He still needs to visit the cities in his network. Is there a way he can decide on a good, if not optimal, route?

Let's think about some strategies that we could apply to solve this and other NP-hard problems.

### Random Guessing

If generating all possible routes is too slow in practice, perhaps we could generate random routes. Then, once we have run out of time, we can just take the shortest one that we have found. This would be a fairly straightforward algorithm. We use a variable to store the shortest route so far, and we repeatedly generate random permutations of the cities. If a permutation (or route) is shorter than the best route so far, then we store this route instead. Once a certain number of random guesses have been made, we can simply stop and take the shortest route we have found to be the best.

The trouble with this solution is that for large numbers of cities, there is no guarantee that we will find a reasonable solution. There are just so many possible permutations. It's like searching for a needle in a haystack, by pulling out one random straw at a time.

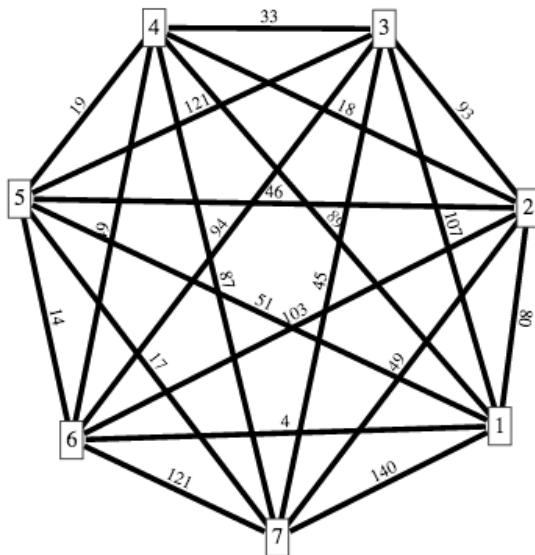
### Heuristics

So, what might be a better approach to finding a good solution? Perhaps we could use heuristics to help? Recall from [module 3.10.9](#), that heuristics are 'rules of thumb' that can be used to solve a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. Let's see how we can use heuristics to help us find an approximate solution to the TSP and other NP-hard problems.

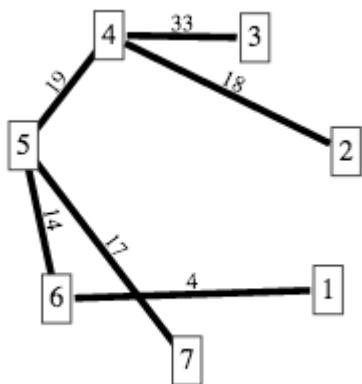
### Simple Approximations

What about the similarity between the TSP and the problems of finding a Minimum Cost Spanning Tree (MCST)? Perhaps we can take advantage of this similarity to help find a solution for the TSP? Although we have noted that the cycle that the TSP aims to find differs from a MCST tree, the problems are similar in that they both require visiting every node in a graph using a minimal path. So, could we use a MCST to generate a tour (or cycle) that solves the TSP? It might not guarantee to be a minimum cost route, but would it be a reasonable approximation?

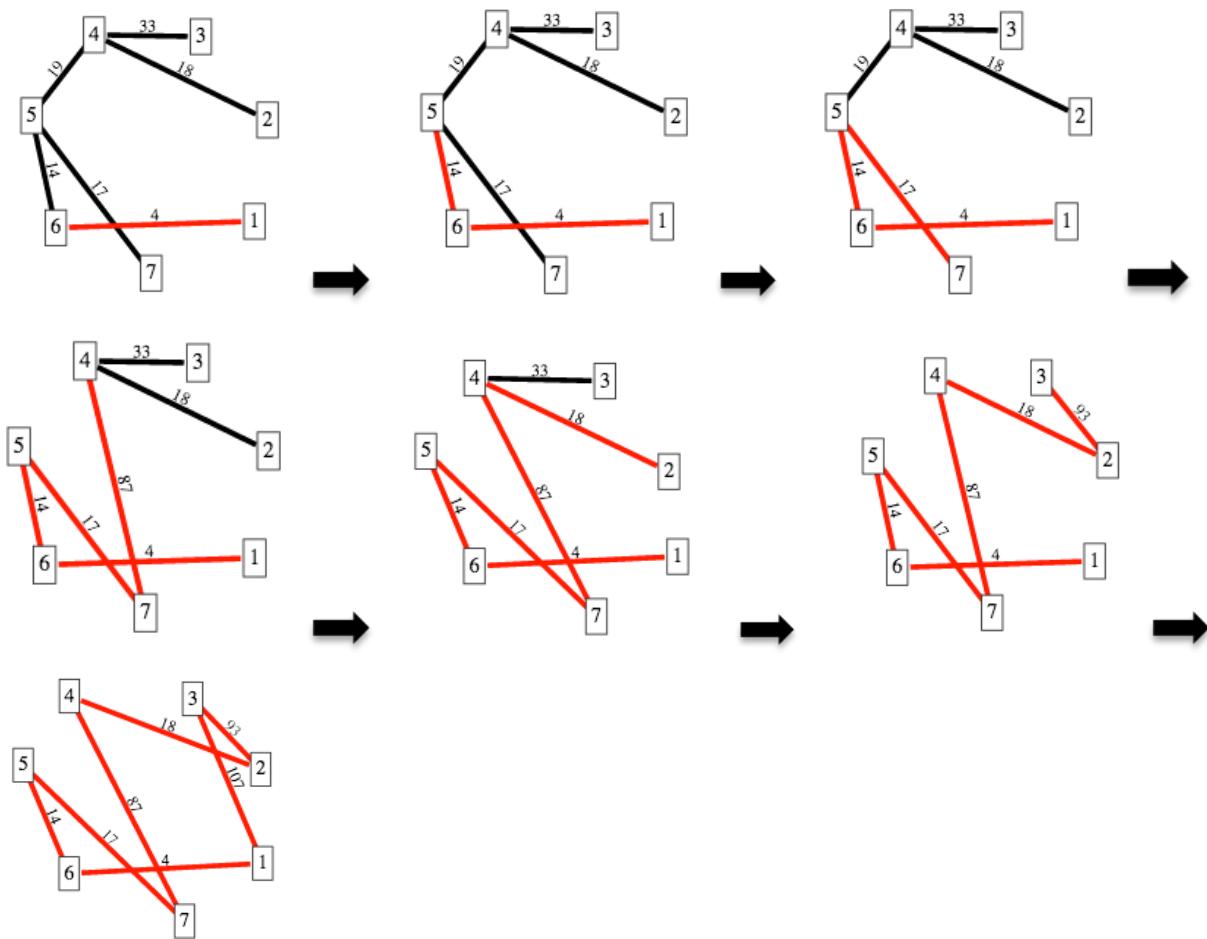
Let's look at an example network with 7 cities:



We can find a minimum cost spanning tree for this network efficiently, using one of the algorithms discussed in [module 3.1](#): Prim's Algorithm, Kruskal's Algorithm or the Reverse-Deletion Algorithm. A MCST for our example network looks like this:-



Now that we have a MCST, how can we convert it to an non-optimal solution to the TSP (an approximately minimum cost tour or cycle)? Well, a tour is like a tree, except that it cannot have branches. So, we need to walk through the tree converting any branch structures into simple paths when we find them. More precisely, if the walk takes to you node already visited, continue walking the tree until you encounter an unvisited node, then draw a shortcut from the last node visited to the newly discovered unvisited node. Once, we have visited every node, we create take a direct route from the final node back to our original start node. If we colour the edges in our example MCST red as we walk the tree, our tour can be created, starting from node 1, with the steps below:-



We can write an algorithm to do this as follows:-

```

function approximate_tsp(start_node):
    create a MCST (using Reverse-Deletion, Prim's, Kruskal's algorithm etc.)
    let tour = []
    tour = create_tour(start_node, MCST, tour)
    add start_node to tour
    return tour

function create_tour(start_node, MCST, tour)
    add start_node to tour
    foreach neighbour of start_node in MCST:
        if tour does not contain neighbour:
            create_tour(neighbour, MCST, tour)

```

Look carefully at the example above. Does this approximation method create an optimal solution to the TSP?

Well, no, in fact it doesn't create an optimal solution. The tour that it finds has length 340. However, the minimum length tour is 211 long (taking the path 1-2-4-3-7-5-6-1). Using a MCST gives us a good solution to the TSP, which we are able to find in "reasonable" time.

So, let's look a bit closer at this approximation algorithm and check that it does in fact run in "reasonable" time. There are 2 steps to our algorithm:

1. create a MCST, for which there are a number of different known polynomial time algorithms.
2. convert the MCST to a path (or tour). This is also a polynomial (linear) time algorithm, as we simply need to walk the MCST, passing over every node at most twice (if it is the branch point for a tree) and ensuring there are edges connecting them.

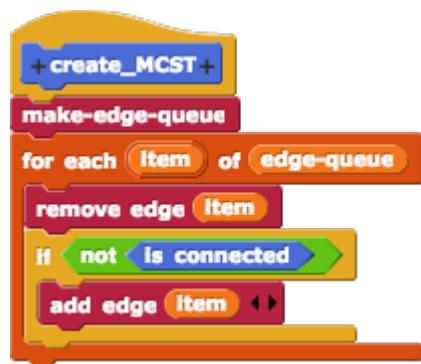
As both steps run in polynomial time, the approximation algorithm as a whole also runs in polynomial time.

We have seen a way to use the MCST as a heuristic for finding a solution to the TSP. This heuristic is a way to find a good solution to the TSP efficiently. So, by using a heuristic, we have traded off the optimality of our solution in return for a reduced time complexity to find it.

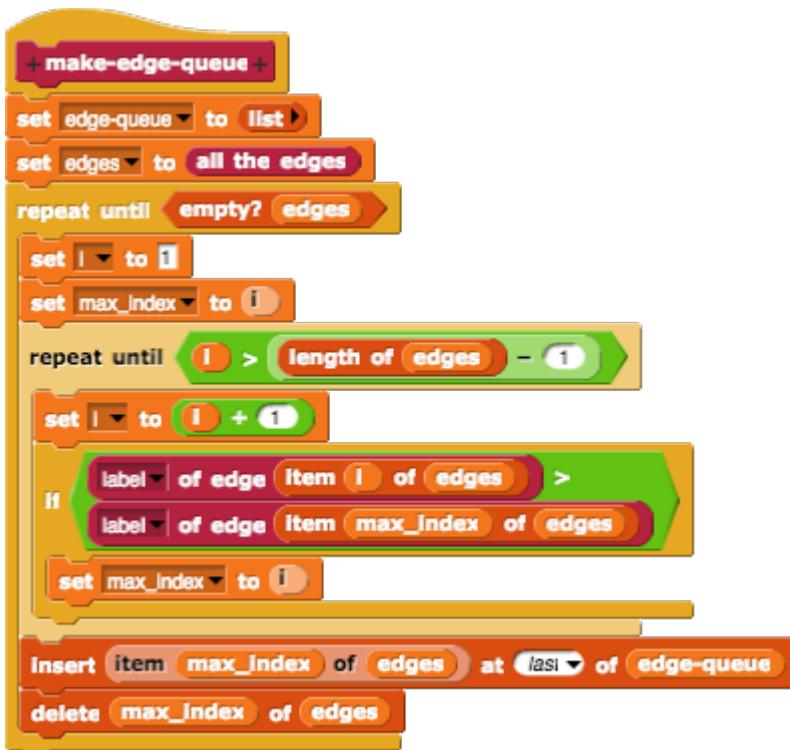
## Edgy code for Approximating the TSP using MCST

Below is some Edgy code for approximately solving the TSP using a MCST, as described above.

First, we need to define some helper functions. The first creates an MCST using the Reverse-Deletion algorithm:-



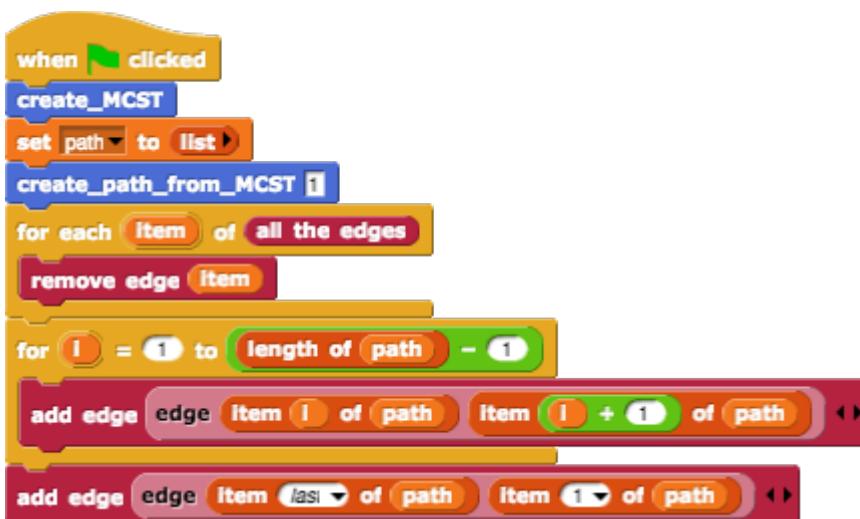
And this requires another helper function to create an ordered list of edges in descending order by weight:-



And a third helper function, will create a path or cycle from a MCST:-



Then, we can put them all together, as follows, to create and then display a cycle that is a good, but non-optimal, solution to the TSP:-

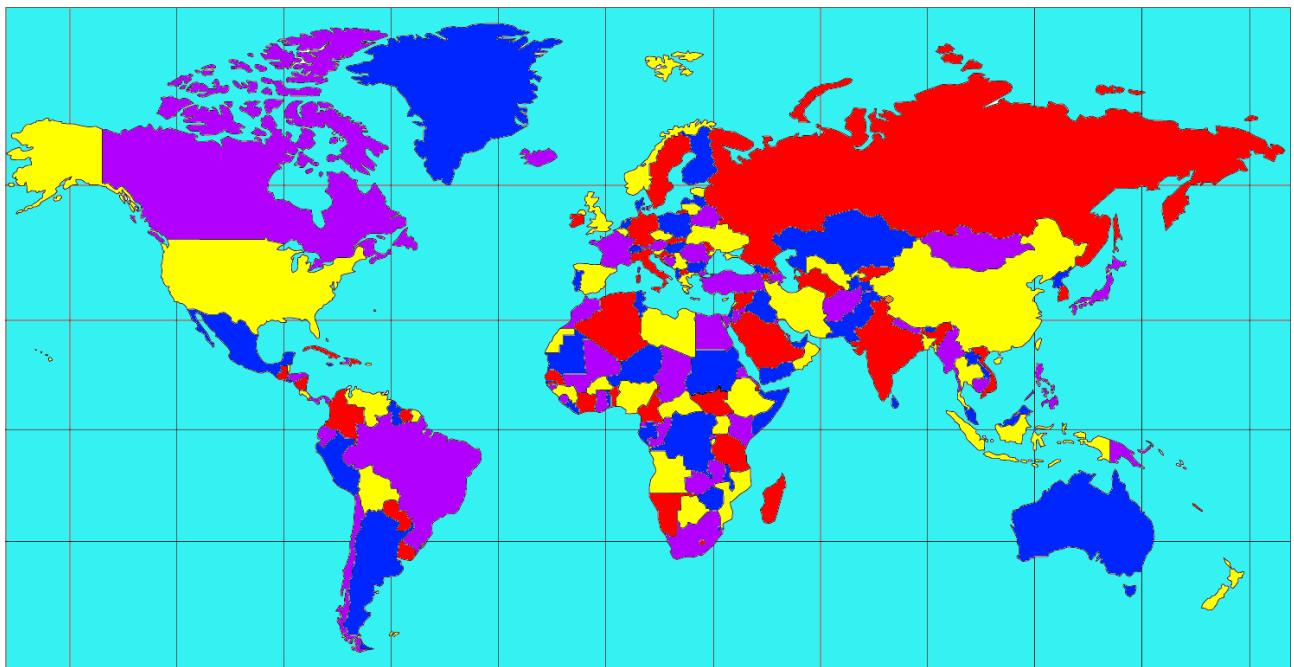




## 3.7.4 Heuristics for the Graph Colouring Problem

### The Graph Colouring Problem

Now we're going to look at a heuristic for solving another known NP-complete problem. Imagine that you need to colour the countries on a map such that no two neighbouring countries are the same colour, like this:-



If you had the map below of various countries in Europe, how could it be coloured using 4 colours?



We can represent this map as a graph, in which the nodes represent the countries on the map, and the edges connect neighbouring countries.



And we can restate our map-colouring problem as a graph colouring problem as follows:- colour the  $n$  nodes in a graph, such that no two adjacent nodes are the same colour. There is an obvious solution that uses  $n$  colours and simply gives each node its own colour, but we want to see if there is a possible solution using only 4 colours.

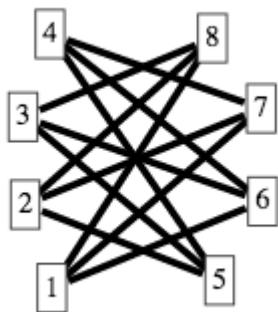
In fact, this is a well-known and well studied NP-hard problem in computer science known as the "graph colouring" problem. The graph colouring problem can also be used to model a variety of other everyday scenarios. For example, job scheduling, in which a set of jobs must be assigned to time slots, with some pairs of jobs being in conflict (for example because they rely on a shared resource). In this case nodes are

used to represent jobs, and edges exist where jobs conflict. The colours assigned to the nodes represent each represent a time slot. Further applications of graph colouring include timetabling, pattern matching and flight and other scheduling problems. Even soduku puzzles can be modelled as a variation on the graph colouring problem. Each grid square is a node in the graph, and those that share a row, column, or sub-grid are connected by edges in a graph. You need to complete a partially complete puzzle by putting the numbers 1 - 9 in the grid squares - equivalent to finding a colouring with 9 colours.

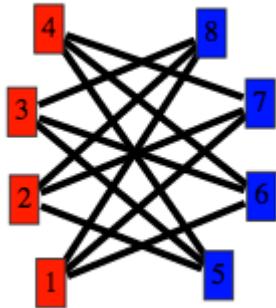
## Approaches to solving the Graph Colouring Problem

Can you think of how we might write an algorithm to solve the graph colouring problem? We could adopt a brute force approach - try every possible colouring and see whether we can find a valid colouring. But, for any reasonable size graph this is going to take too long. We've already seen that brute-force algorithms are not efficient for solving NP-complete problems, as their running time grows exponentially large with the size of the problem.

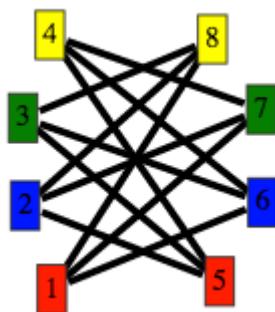
Another possible approach is to use a greedy algorithm: we consider each node in sequence and assign each node the first available colour that does not conflict with any previously coloured nodes. If necessary, use a new colour. Do you think that this greedy algorithm will guarantee to find a solution if one exists? Imagine we applied this algorithm to the following graph:-



If we colour the nodes in numbered order as shown below, we get a valid colouring with 2 colours:-



However, if we colour the nodes in the order 1,5,2,6,3,7,4,8, a valid colouring requires 4 colours, as shown below:-



As you can see, the greedy algorithm is not guaranteed to generate a valid solution for a given number of colours. The number of colours used in the resulting colouring depends on the order in which the nodes are visited. Theoretically, there exists an ordering that leads to a greedy colouring with a minimal number of colours. However, greedy colourings can also be arbitrarily bad as we can see in the second colouring above.

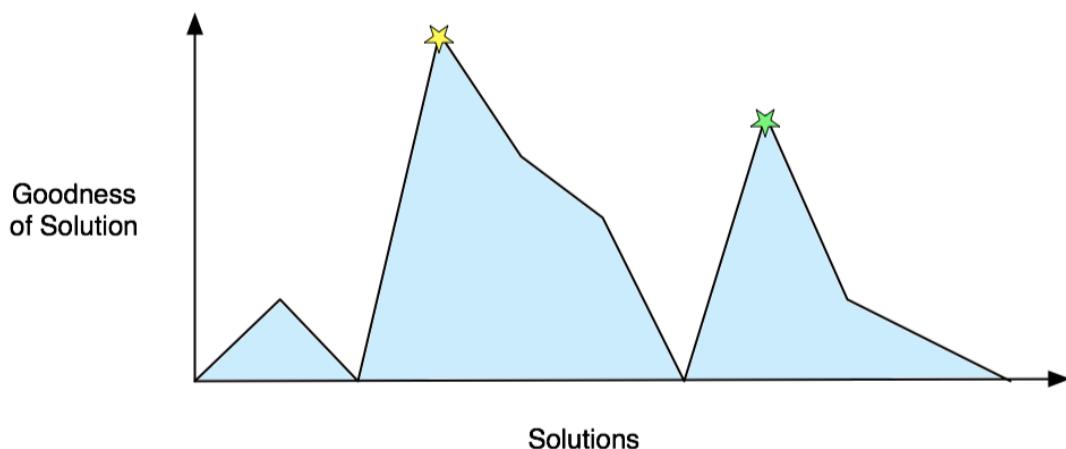
## Iterative Improvement Algorithms

Let's think about another approach to the graph colouring problem. What if we start with a random colouring using  $n$  colours, and then we change the colour of one individual node at a time, to reduce the number of adjacent nodes that are coloured the same?

This is the idea behind a group of algorithms known as iterative improvement algorithms - we start with a complete, random configuration and modify this to improve its quality. It may be helpful to think of all possible configurations as forming the surface of a landscape - configurations that are better (according to some evaluation score) are higher points in the landscape. A basic iterative improvement algorithm chooses a random point in the landscape as its starting point. It then moves around the landscape trying to find the highest point, which corresponds to the optimal solution. Generally, iterative improvement algorithms only keep track of the current solution and look at neighbouring configurations, and because of this, it has been said that they resemble "trying to find the top of Mount Everest in a thick fog whilst suffering from amnesia". Nonetheless they are useful for solving hard problems.

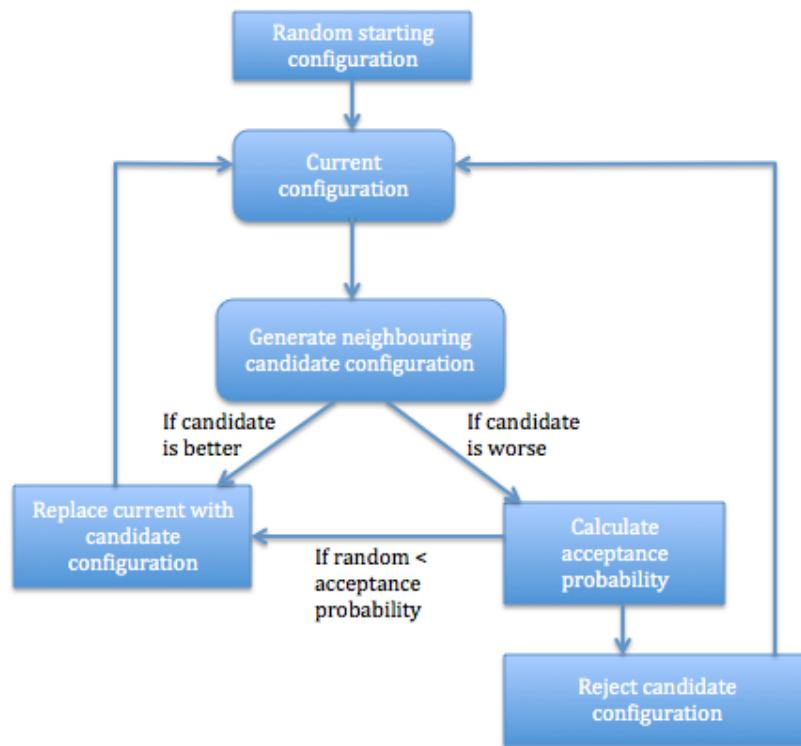
## Randomized Search - Simulated Annealing

So, if you imagine that you are moving through a "landscape" of possible solutions like an iterative improvement algorithm. You come to a point from which there are no better neighbouring solutions. You can probably imagine that you're at the top of some kind of peak. Does this mean you've found the best solution? Not necessarily. In fact, this kind of local search algorithm can lead to a situation where you are stuck at a sub-optimal solution (also known as a local maxima). For example, in the diagram below, the best solution is at the yellow star. However, a simple algorithm might reach the green star and never move away from it. How could you know whether there is another larger peak elsewhere?



Well, this is where we can use an iterative improvement algorithm called 'Simulated Annealing'. In simulated annealing, we move to a neighbouring solution if it is better, however, we may also sometimes move to a neighbouring solution that is worse. As the algorithm continues to run, we decrease the size and frequency of "bad" moves that are acceptable. This approach avoids getting stuck on local maxima.

Here's a high-level overview of the simulated annealing algorithm:-



The flowchart above gives you a basic idea of how simulated annealing works, but it leaves out some important details:- how do we generate a neighbouring configuration, how do we judge if one candidate is better than another, when does the algorithm terminate and what is the acceptance probability and how can we calculate it?

Let's look at these details one by one:-

- Generating a neighbouring configuration: A neighbouring configuration is generated by only

changing one thing at random in the current configuration. In our graph colouring example, this means changing the colour of one node.

- Judging which configuration is better: We need a way of calculating a "score" for each configuration, such that a better configuration has a higher score. In the case of graph colouring, this could be simply calculating the proportion of the edges in the graph that connect different coloured nodes.
- Termination: The simulated annealing algorithm is based on the process of heating and cooling metals known as "annealing". As a result, the algorithm uses a parameter which is known as the "temperature" and is gradually reduced for each iteration. This is sometimes referred to as the cooling "schedule". Once the temperature reaches a certain minimum value, the algorithm terminates. Usually the temperature starts at 1.0 and is decreased after each iteration by multiplying by a constant, alpha, which is usually between 0.8 and 0.99.
- Calculating the acceptance probability: The acceptance probability is used to decide whether to move to a worse configuration or not. It is a number between 0 and 1 which is then compared to a random number between 0 and 1 in order to decide whether to accept a new configuration or not. It is usually calculated using the formula below. So, as the temperature is reduced, the algorithm is less likely to accept a worse configuration.

$$\text{acceptance\_probability} = e^{((\text{new\_score} - \text{old\_score}) / \text{temperature})}$$

## Simulated Annealing for the Graph Colouring Problem

Let's see how we could use simulated annealing to solve the graph colouring problem. We can write an algorithm to do this as follows:-

```
function find_colouring(colours):
    temp = 1.0
    alpha = 0.8
    min_temp = 0.01
    colourings_per_temp = 20

    current_colouring = generate random initial colouring using colours
    current_score = score(colouring)
    while temp > min_temp:
        repeat colourings_per_temp:
            new_colouring = change the colour of one node in current_colouring
            at random
            new_score = score(new_colouring)
            if new_score = 1:
                // stop if we have found a valid colouring
                return new_colouring
            else if (new_score > current_score):
                // apply the new colouring
                current_colouring = new_colouring
                current_score = new_score
            else:
                acceptance_probability = e^((new_score - current_score) / temp)
                if acceptance_probability > random(0..1):
                    // apply the new colouring
                    current_colouring = new_colouring
                    current_score = new_score
                else:
                    // keep the current colouring
```

```

temp = temp * alpha

function score(colouring):
    valid_edges = 0
    for each edge in colouring:
        if edge connects 2 node coloured different colours:
            valid_edges = valid_edges + 1
    return valid_edges / number of edges in graph

```

Consider the simulated annealing algorithm detailed above. Do you think it will find a solution to the graph colouring problem of using 4 colours to colour the map of central Europe above?

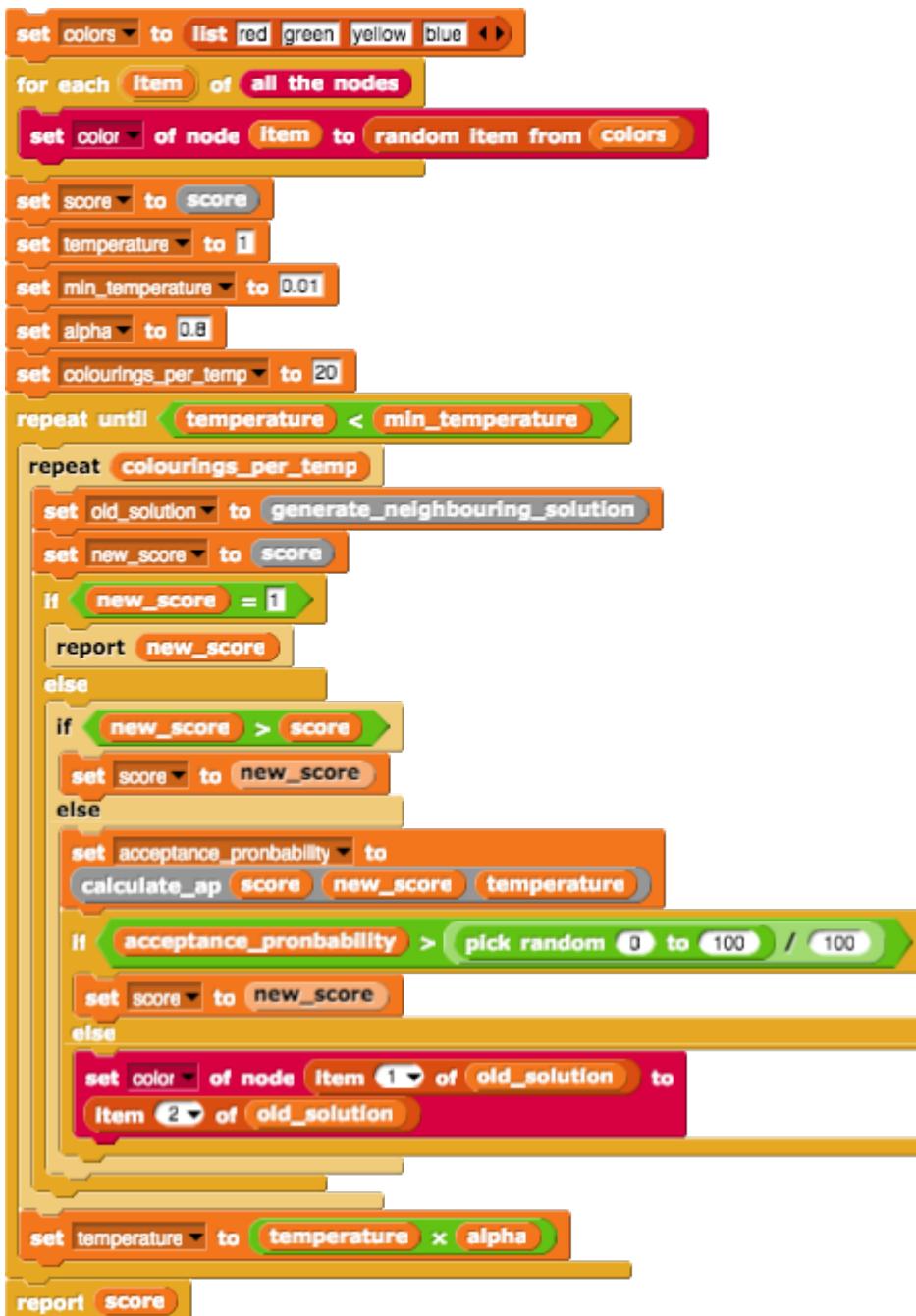
Well, there is no guarantee that it will find a solution (if one exists). However, like other heuristic approximation methods that we've seen, the search that it performs should find a 'good' solution provided the cooling schedule is chosen appropriately. For the graph colouring problem, you might want to modify the algorithm to store details of the best colouring that it has found so far (i.e. the colouring with the best score).

Here's one possible solution to colouring the map of Europe above with 4 colours:-

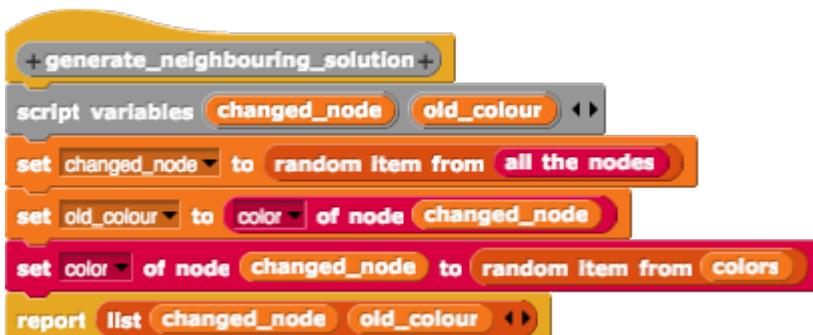


## Edgy code for using Simulated Annealing to Solve the Graph Colouring Problem

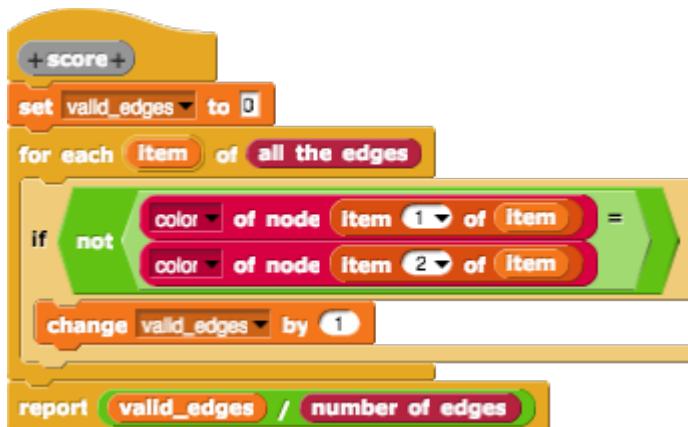
Here is the Edgy code for solving the graph colouring problem using simulated annealing, as described above:-



This algorithm requires some helper functions. The first of these, generates a neighbouring configuration by randomly changing the colour of one node. Note that this function also reports which node was changed and from what colour. This is in case the configuration is rejected and the random change must be undone.



Another helper function calculates a score for the current colouring:-



And, here's the final helper function, that calculates the acceptance probability for a newly generated colouring:-



## **4**

# **Turing Machines and the Origin of Computer Science**

[4.1 Hilbert's Program](#)

[4.2 Alan Turing and the Turing Machine](#)

[4.3 Busy Beavers](#)

[4.4 Undecidability and The Halting Problem: Hard Limits of Computation](#)

[4.5 A Weird Computational Formalism](#)

[4.6 A short Introduction to the General History of Computing](#)

---

## **4.1 Hilbert's Program**

These videos explain a "simple" question in set theory



(<https://www.alexandriarepository.org/wp-content/uploads/20150904205248/Russell-Zermelo-Paradox.mp4.mp4>)  
and how it triggered a fundamental crisis in mathematics that lead to the birth of computer science.



(<https://www.alexandriarepository.org/wp-content/uploads/20150904205102/Hilberts-Program.mp4.mp4>)



## **4.2**

# **Alan Turing and the Turing Machine**

Alan Turing's response to Hilbert's challenge



(<https://www.alexandriarepository.org/wp-content/uploads/20150904205513/Turing-Machines.mp4.mp4>)

---

## **4.3 Busy Beavers**

**A Fun Game with Turing Machines that has Deep Meaning**



(<https://www.alexandriarepository.org/wp-content/uploads/20150904200401/Busy-Beavers.mp4.mp4>)

---

## **4.4**

# **Undecidability and The Halting Problem: Hard Limits of Computation**

Why some things just cannot be computed (and why Busy Beavers are very unpredictable animals)



(<https://www.alexandriarepository.org/wp-content/uploads/20150904204530/Halting-Problem.mp4.mp4>)

---

## **4.5**

# **A Weird Computational Formalism**

You will not believe how little you need to compute!

(Note: this is purely optional and not formal contents of VCE Algorithmics)



(<https://www.alexandriarepository.org/wp-content/uploads/20150904215055/GameOfLifeAndComputation.mp4.mp4>)

---

## **4.6**

# **A short Introduction to the General History of Computing**

A very brief introduction to the broader history of computation beyond Turing machines.

(Note: this is not a formal part of VCE Algorithmics)



(<https://www.alexandriarepository.org/wp-content/uploads/20150904202137/Computing-History.mp4.mp4>)

---

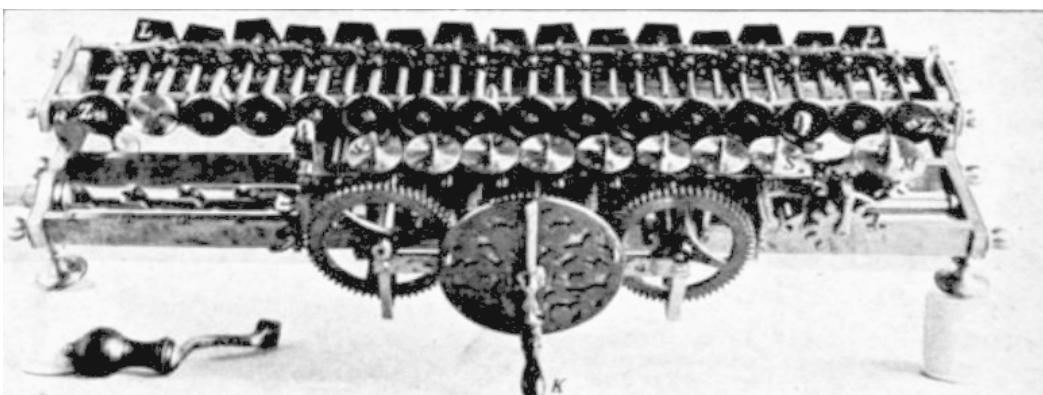
## 5

# Searle's Chinese Room Argument

Is artificial intelligence (AI) possible? Can computers think? How can we even address this question? In this module we consider a famous thought experiment, the *Chinese Room*, and how it has helped our understanding of this question. First, for some historical perspective, we review an earlier thought experiment due to Leibniz.

## Leibniz's Mill

Godfried Leibniz lived just prior to the Industrial Revolution, in a period when factories were beginning to replace solo craftsmen as the means for manufacturing things. With manufacturing came the need for more complex calculations, e.g. what size of stream and water wheel would be needed to power what size of mill? With manufacturing also came machines. Devices were already being made for measuring distances and angles with greater precision. Greater precision led to more intricate machines, such as this multiplying machine:



Leibniz Calculator ([https://en.wikipedia.org/wiki/Stepped\\_Reckoner](https://en.wikipedia.org/wiki/Stepped_Reckoner))

Leibniz's interest was in formal reasoning. When solving a mathematical problem for instance, you could simply capture it in a formula, and then apply the rules of calculus. This could be applied to compute the trajectory of a missile. With this reasoning "technology", there was no longer any need to see who had the most convincing argument for the solution; you could just write it down and compute. As machines grew in complexity, Leibniz was able to speculate about building ever more complex machines, to the point where a machine could manipulate not numbers, but *ideas*.

Leibniz's Mill is a thought experiment. A mill is a water-powered grinding machine, typically housed in a building that contains heavy grinding stones, used for turning grain into flour. Other mills performed different tasks, such as cutting timber (a sawmill). The mill building is large enough that you could enter and walk around inside (<http://www.angelfire.com/journal/millrestoration/mill.html>). By referencing a mill, Leibniz raised the prospect that a human would not need to turn the handle to power the work (as in the figure above); the machine would manipulate ideas on its own, powered by the environment.

Suppose that the mill contained a complicated version of the above machine. And suppose that, from the outside, it appeared that this machine was capable of performing thought, i.e. it was capable of representing information about the environment, manipulating it, and making decisions. Leibniz imagined walking around inside the machine, and watching the cogs as they turned during the process of thinking. He wondered, where would the thinking be happening? Where are the ideas? Clearly, the thinking must

be taking place somewhere else.

Aside: Leibniz's idea of building machines that could think did not take hold. (What's the main reason why it could not?)

## Searle's Chinese Room Argument

Jump forward 200 years to the middle of the 20th Century and we were now starting to build analogue computers with a new level of complexity, and it was now conceivable that a computer could process natural language. Writing in 1950, Turing predicted we would have computers that could engage in a conversation with us by the turn of the century, although this task has turned out to be harder than Turing expected. The machines we access each day are vastly more powerful than Turing anticipated, and so the question still arises for us: at what point can we say that the computer on my desk, or in my pocket, is a "thinking" thing?

Discussion: What would you set up as a Turing test? Consider this [xkcd cartoon](https://xkcd.com/632/) (<https://xkcd.com/632/>). Recaptchas are quite effective at discriminating humans from machines. If a machine could reliably solve these visual challenges, would you credit it as being able to think? (Aside: have you noticed that recaptchas are getting more difficult for humans to solve? Perhaps we will get to the point where machines surpass humans at solving them.) What would it take to convince you that a computer could think?

If you have seen either of the movies *Her* or *Ex Machina*, you may have new intuitions about this. How quickly do you anthropomorphise with computers? At one level, you might do this when complaining about a machine "that has a mind of its own". But usually, you're not serious when you say things like this. Could you genuinely anthropomorphise the computers in these movies? And could this be genuine when the the computer roles were played by humans?

Searle, a professor at Berkeley, wrote a paper in 1980 called *Minds, Brains and Programs*, in which he proposed the Chinese Room thought experiment.

## The Chinese Room

Suppose you find that you have been put in a room. Looking around you, you see a large book, and it contains detailed instructions written in English. You also see a large number of boxes of papers, written in squiggles that you do not recognise. There is a slot, and sheets of paper are coming in through the slot.

You open up the instruction book and it tells you to examine the sheets of paper coming in through the slot. You see more unrecognisable squiggles on those sheets. Following the instructions, you open a particular box of papers and find one that matches the squiggles. Continuing to follow the instructions, you make some notes in English, examine other papers, and eventually, you carefully draw some squiggles on a fresh piece of paper and put it back through the slot.

More papers come in, and as before, they only contain unrecognisable marks. Your only way of dealing with them is to follow the instructions in the book, matching shapes, making and reading your own notes written in English, and drawing new shapes to put back through the slot. At no stage, do you understand if the squiggles coming in or going out mean anything.

As a well-educated student, you are highly trained at following instructions without knowing what they mean, and without even thinking to question why the instructions exist. You happily follow along,

correlating symbols, and drawing squiggles.

Unbeknownst to you, the people outside the room speak Chinese. (If you speak Chinese, you need to pick some other language that you don't know, like Basque, Quechua, Uzbek, or Zulu. Chinese was the most exotic language that Searle could think of at the time). Those people refer to the paper they put in as "questions" and the paper that comes back as "answers". They refer to the instruction book as the "program" and the boxes of papers as "real world knowledge".

We do not know whether your answers are interpreted as direct answers to the questions, or as requests like "please stop asking questions and let me out of this room". We just know that no-one who looks at your "answers" can tell that you don't speak a word of Chinese.

NB. If you're concerned about speed, and that the people outside would know you were not a speaker of Chinese simply because of the slow speed of your response, then let's assume they don't know whether you're awake or sleeping, or preoccupied with something else so unable to answer right away.

## Searle's Question

Do you understand Chinese by doing this? Of course not. You are just following English instructions. However, as the result of your efforts, Chinese questions are being answered, in Chinese. Searle asks: *if there is understanding going on in the Chinese Room, where is it?*

Discussion: what do you think? Is there understanding going on in a system like this? If so, where?

(Extension: would it be different if you had managed to fully internalise the contents of the instruction book and boxes of notes, and get to the point where you could answer Chinese questions as effortlessly as a speaker of Chinese?)

## The Argument

Searle sets up the Chinese Room to argue against the position of "Strong AI". This is the view that computers that display intelligent behaviour such as question-answering really are intelligent, that such computers really do understand. With this definition, his argument is as follows:

1. If Strong AI is true, then there is a program for Chinese such that if any computing system runs that program, that system thereby comes to understand Chinese.
2. I could run a program for Chinese without thereby coming to understand Chinese.
3. Therefore Strong AI is false.

## Responses to the Chinese Room Argument

Searle considers a range of responses to the Chinese Room Argument:

1. The Systems Reply: the person in the room doesn't understand, s/he is just like a CPU; it's the system that understands.
2. The Virtual Mind Reply: the functioning room evokes a separate virtual agent which is the locus of understanding.
3. The Robot Reply: you need to put the room into a robot and add sensors and actuators so that the robot can engage with the world, and then you will have a machine that understands.

4. The Brain Simulator Reply: the activity inside the room could replicate the firing of nerves inside the brain of a Chinese person who understands the questions and gives answers, so therefore there is understanding in the room.
5. The Other Minds Reply: how do you know that anyone else who answers Chinese questions actually understands Chinese? Presumably from their behaviour. Since the room is behaving the same way, you should credit it with understanding Chinese too.
6. The Intuition Reply: the Chinese Room Argument depends on Searle's unwarranted intuitions that certain things are incapable of thought.

Discussion: do you find any of these responses particularly compelling? Why? Can you refute any of them?

## How Searle answers the responses

You are now ready to read the discussion of the Chinese Room Argument that appears in the Stanford Encyclopedia of Philosophy: <http://plato.stanford.edu/entries/chinese-room/>

If you're game you might also like to try reading Searle's original article.

## Finally

What is *your* position - do you believe in Strong AI?

## Readings

Cole, David (2014), [The Chinese Room Argument](http://plato.stanford.edu/entries/chinese-room/) (<http://plato.stanford.edu/entries/chinese-room/>), *Stanford Encyclopedia of Philosophy*

Searle, John (1980), [Minds, brains and programs](http://philpapers.org/rec/SEAMBA) (<http://philpapers.org/rec/SEAMBA>), *Behavioral and Brain Sciences*

## Acknowledgements

The authors wish to thank Greg Restall at the University of Melbourne, whose lecture on the Chinese Room Argument was used as the basis for some of the discussion in this module.

# 6

## DNA Computing a la Adleman

Computing takes place in many forms and media, from the digital computers we all use to biological wetware. As a short taster, and to give our study of 'conventional' algorithms some broader context, we look at a short example of one specific form of computing with biological wetware. Let's get a bit of an overview first.



(<https://www.alexandriarepository.org/wp-content/uploads/20150907223133/DNA-Computing-Context.mp4.mp4>)

The following video contains the essential redux of Adleman's seminal experiments with DNA computing, in which he devised a new solution to the traveling salesman problem.



(<https://www.alexandriarepository.org/wp-content/uploads/20150907222104/DNA-Computing-Adleman-TSP.mp4.mp4>)

Interestingly, we will find that much of our previous discussion of the TSP based on 'conventional' algorithmics remains useful, and in fact provides the basis for understanding the implications of the DNA solution.



(<https://www.alexandriarepository.org/wp-content/uploads/20150907223107/DNA-Computing-Complexity.mp4.mp4>)

Of course, what these videos have illustrated is just one form of DNA computing (and only an exceedingly small slice of bio computing in general) to give some context to our study of standard algorithmics. But I hope you have seen that at least for some of these forms what we have learned about conventional algorithmics can still give us a conceptual framework to understand what is going on.

## Further Reading

- Computing with DNA Molecules, or Biological Computer Technology on the Horizon in *Algorithmic Adventures*. Juraj Hromkovic. Springer-Verlag, 2009.
  - Computing with DNA. Leonard M. Adleman in *Scientific American*, August 1998, p. 54-61.
-

# 7

## Appendix A: Extension materials

[7.1 Being Harry Houdini](#)

[7.2 Semantic Specification of Abstract Data Types](#)

[7.3 Tail Recursion](#)

[7.3.1 Exercises: recursive list idioms in Edgy](#)

[7.3.1.1 Solutions: recursive list idioms in Edgy](#)

[7.3.1.2 Solutions: Append and Reverse in Edgy](#)

[7.4 Testing Algorithm Correctness](#)

---

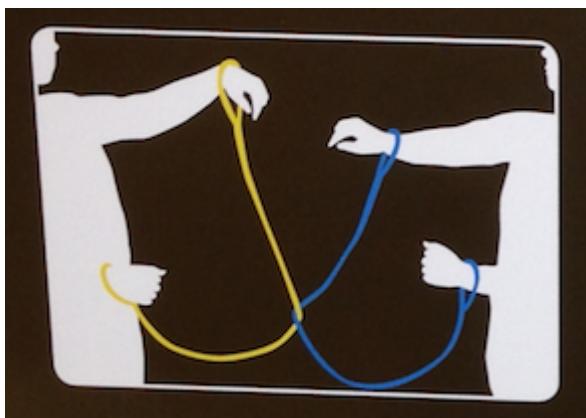
# 7.1

## Being Harry Houdini

### standalone version

### How to free yourself systematically

Imagine yourself being having your hands 'tied' together with another person as shown in the following picture.



At first, it seems impossible to disentangle yourself without cutting the rope (or an arm!). Surprisingly it is possible!

Your task is to find out how to do this and then describe how it can be done.

What does this have to do with Computer Science, you ask? What you have to write is effectively an algorithm. You will see how difficult this can be if the problem doesn't just involve numbers and letters!

### Steps

1. Find a partner to perform the 'experiment' with
2. Prepare two separate pieces of rope (about 1m-1.5m).
3. Tie loops at both ends of each rope (these must be big enough to fit your hands through, but not much bigger).
4. Fit one rope over both your partners hands
5. Fit one end of the rope over one of your hands, pass the rest of this rope behind the rope that your partner already holds, and then fit the free rope over your other hand. You should now be entangled as on the picture shown above.
6. Try to find a way to disentangle this (without taking the ropes off your hands or cutting it!)
7. Don't give up! It is possible.
8. After you have figured out how to do this, write down a precise set of instructions that allows someone who hasn't found the solution to free themselves.
9. Now find two people who are not familiar with this problem.
10. Entangle them according to steps 1-5 (provided they agree!)
11. Read your instructions back to them step by step. You can read as slowly as necessary, you can even repeat the instructions, but you are not allowed to add anything to them, to answer

questions, or to provide further explanation. You can only read out what is on the paper. Likewise, your entangled friends are not allowed to do anything that they are not told to do in your instructions. They must follow the instructions precisely.

12. Did your friends manage to disentangle themselves? Most likely not. Why did this happen?

Of course, The Great Houdini would not have passed on his solution ! And he probably would have done this while submerged in a tank of water...

---

## 7.2 Semantic Specification of Abstract Data Types

### Semantics of the Stack operations

Specifying the meaning of the stack operations requires some axioms. Consider the following:

$$\text{top}(\text{push}(S, \text{val})) = \text{val}$$

This just says that if we push some value *val* onto the stack, then inspect the top of the stack, we see that value. Here are some axioms that express what push and pop do to the size of the stack:

$$\begin{aligned}\text{size}(\text{push}(S, \text{val})) &= \text{size}(S) + 1 \\ \text{size}(\text{pop}(S)) &= \text{size}(S) - 1\end{aligned}$$

And here is what we need to say about the `is_empty` operation:

$$\begin{aligned}\text{is\_empty}(\text{new\_stack}) &= \text{True} \\ \text{is\_empty}(\text{push}(S, \text{val})) &= \text{False}\end{aligned}$$

Finally, recall that it is an error to apply `top` or `pop` to an empty stack. Here's how we can express that:

$$\begin{aligned}\text{top}(\text{new\_stack}) &= \text{Error} \\ \text{pop}(\text{new\_stack}) &= \text{Error}\end{aligned}$$

(Note that you are not required to know these axioms. However, it will help your understanding of the Stack ADT if you are able to interpret each of the above statements.)

---

## 7.3 Tail Recursion

### What is Tail Recursion?

There are often various different algorithms for solving a problem in computer science. We can think about solving one problem in different ways.

For example, if we want to calculate factorial of a number  $x$ , we can think of it mathematically as a product:-

$$n! = n \cdot (n-1) \cdots 2 \cdot 1$$

or recursively:-

$$n! = 1,$$

*if ( $n=1$ ), or*

$$n \cdot (n-1)!$$

*if ( $n>1$ )*

Both are correct ways of thinking about the problem. In fact, if we look at the example of 5 factorial, we can see that the recursive version can be expanded as follows, to produce the iterative version.

5!  
5.4!  
5.4.3!  
5.4.3.2!  
5.4.3.2.1!  
5.4.3.2.1

Similarly, if we want to find the maximum value in a list, we can think about this iteratively or recursively. Let's represent a list using the "cons" operator, which adds an item to the front of a list. So, for example the list [3,5,1] can be written cons(3, cons(5, cons(1, nil))). Then we can think of a procedure to find the maximum item in the list recursively as follows:-

`max_of_list(cons(head, tail)) = head,`

*if  $tail=nil$ , or*

`max(head, max_of_list(tail))`

*otherwise*

or iteratively (where  $tail[n]$  is the  $n$ th item of the tail list):-

`max_of_list(cons(head, tail)) = max(nil, max(tail[length_of_tail], ...`

```
max(tail[2], max(tail[1], head)) . . .)
```

In fact, iterative functions are equivalent to recursive functions and it is always possible to convert one to the other, using a specific kind of recursion called "tail recursion". So, what is tail recursion? To find out, let's revisit our recursion schema and look at it a bit more closely. Specifically, we're going to look at where the recursive call is positioned.

In tail recursion, the recursive call is the last thing that the function does (a "tail" call). Any calculations are performed first, and then passed to the recursive call if necessary. This sometimes means that extra parameters are required for a tail recursive function. These parameters store the values of any intermediate calculations that are necessary.

Tail recursion is also useful, because it is often more efficient than traditional recursion. When the computer evaluates a recursive function, it usually has to remember (or store) any remaining calculations to be completed once the recursive call has been evaluated. If the recursion is deep or the calculations are complicated, this can take up a lot of space. In tail recursion, however, because the final call in a tail recursive function is the recursive call, there is no requirement to store any remaining incomplete calculations.

NB. Tail recursion is an example of the Decrease-and-Conquer algorithm design pattern.

## An example of Tail Recursion

As an example, let's look at the "sum" function, that sums up all the numbers from 0 to a given positive number, x.

Here is the recursive version, below. Can you see that recursive call is performed and then the value that it returns is added to x? This version is not tail recursive as the recursive call to sum\_rec is not the last thing in the function (its result is added to x).

```
function sum_rec(x):
    if x != 0:
        return x + sum_rec(x - 1)
    else:
        return 0
```

Calling sum\_rec(4), results in the following sequence of calls when the function is evaluated. You can see that the calculations cannot be performed until the base case is reached and every recursive call has completed. This means that evaluating this function requires more and more space to store the necessary calculations for each recursive call.

```
sum_rec(4)
(4 + sum_rec(3))
(4 + (3 + sum_rec(2)))
(4 + (3 + (2 + sum_rec(1))))
(4 + (3 + (2 + (1 + sum_rec(0)))))
(4 + (3 + (2 + (1 + 0))))
10
```

Here's is a tail-recursive version of the same sum function which we saw above. Notice that in this version the recursive call is the final (or "tail") step in the recursive case, and no further calculations are performed using its return value.

```
function sum_tail_rec(x, running_total):
    if x != 0:
        return sum_tail_rec(x - 1, running_total + x)
    else:
        return running_total
```

And here is the corresponding sequence of calls when the `sum_rail_rec(4, 0)` is evaluated. In this version, with each evaluation of the recursive call, the variable "running\_total" is updated, so the space required to store the calculations does not increase as recursive calls are made.

```
sum_tail_rec(4, 0)
sum_tail_rec(3, 4)
sum_tail_rec(2, 7)
sum_tail_rec(1, 9)
sum_tail_rec(0, 10)
10
```

## Iteration vs Tail Recursion

There is a direct correspondence between an iterative function and a tail recursive version of the same function. In fact, an iterative function can be converted to a tail recursive function and vice versa.

Here is an iterative version of the sum function that we discussed above:-

```
function sum_iter(x):
    set running_total = 0
    while (x != 0):
        running_total = running_total + x
        x = x - 1
    return running_total
```

Compare this with the tail recursive version, initially invoked with a `running_total` of 0:-

```
function sum_tail_rec(x, running_total):
    if x != 0:
        return sum_tail_rec(x - 1, x + running_total)
    else:
        return running_total
```

Do you notice correspondences between the two versions above?

The iterative version uses a local variable `running_total` to store the results of the sum calculations it has already made, as it loops over all the integers from `x` down to 0. Whereas in the tail recursive version, the variable `running_total` is passed as a parameter to each recursive call and used to store an intermediate value as each recursive call is executed.

In general, to convert an iterative function to a tail recursive function, you can use the loop termination condition as the base case, and the body of the loop as the recursive step. The local variables in the iterative version turn into parameters in the recursive version.

## Binary Tree Search Using Iteration and Tail Recursion

A binary search tree is a tree in which each node has at most 2 children: a left child and a right child. The nodes of a binary search tree are in sorted order, such that a left child is always less than its parent node and a right child is always greater than its parent node. We're going to look at a function to find a particular item or "key" in a binary search tree.

Here is an iterative version of this function:-

```
function binary_tree_search_iter(node, key)
    set current_node to node
    while (current_node != null):
        if current_node == key:
            return current_node
        else if (key < current_node):
            current_node = left child of node
        else:
            current_node = right child of node
    return null
```

Starting at the root node of the tree, if the `root_node` is null then the key does not exist in the tree. If the `root_node` equals the key, then the key has been found. Otherwise, if the key is less than the `current_node`, we check the left child node in the same way as the `root_node`. And similarly, we check the right child node if the key is greater than the current node.

If we apply the method above for converting from an iterative function to a tail recursive function, we produce the corresponding tail recursive version:-

```
function binary_tree_search_tail_rec(current_node, key):
    if current_node != null:
        if current_node == key:
            return current_node
        else if (key < current_node):
            return binary_tree_search_tail_rec(left child of node, key)
        else:
            return binary_tree_search_tail_rec(right child of node, key)
    else:
        return null
```

Take a few moments to study the correspondence between then two versions above.

The loop termination condition in the iterative version, when the `current_node` is null, has become the base case of the recursive version. And the body of the loop, in which the current node is compared to the value of the key, has become the recursive case.

## Exercises

### 1. Searching in a List: Iteration to Tail Recursion

The function below searches for a particular value (or key) in a list. It returns the position of the key, if it is found, and -1 otherwise.

```
find(list, key):
    for(i in 1 to length_of_list):
        if ith_item_of_list == key:
            return i
    return -1
```

Use the method for converting between an iterative function and a tail recursive one, to rewrite this iterative function using tail recursion.

## 2. Checking for Palindromes: Tail Recursion to Iteration

Below is a tail recursive function to check whether a string is a palindrome (a string that reads the same backwards and forwards).

```
function is_palindrome(input_string):
    if length of input_string > 1:
        if first character of input_string == last character of input_string:
            set shortened_string = input_string minus first & last characters
            return is_palindrome(shortened_string)
        else:
            return false
    else:
        return true
```

Use the conversion method described above in reverse to write an iterative version of the same function.

---

## 7.3.1

# Exercises: recursive list idioms in Edgy

**Note that this material is beyond the scope of the unit, but useful for deepening your understanding of the material.**

In the previous module we have already encountered a couple of tail recursive functions on lists that you would probably have written as iterative functions before.

It is a good exercise to define all of the typical simple functions on lists as recursive functions. The solutions are given in the following submodules, but you should try to solve this by yourself before you look up any solutions. Define each of the following functions on lists in Edgy, first as an iterative function, then as a recursive function.

- ***min : list → int*** (finds the minimal element in a list of numbers)
- ***max : list → int*** (finds the maximal element in a list of numbers)
- ***length : list → int***: (computes the number of elements of a list of numbers)
- ***total : list → int***: (computes the total sum of a list of numbers)
- ***last : list → elem***: (returns the last element of a list)

It is more natural to think about the following functions as recursive straight away rather than defining iterative versions first:

- ***zip : list × list → int***: this function returns a list that contains the elements from both argument list in alternating order, for example,  $\text{zip}([1, 2, 3], [4, 5, 6])=[1, 4, 2, 5, 3, 6]$ . This should not be too tricky to work out. Each recursive level takes the first element from each of the lists simultaneously.
- ***merge : list × list → int***: this function returns a list that contains all elements from both argument lists in sorted order provided that each argument list by itself is sorted already. For example,  $\text{zip}([1, 3, 4], [2, 5, 6])=[1, 2, 3, 4, 5, 6]$ . This works essentially like the *zip* above, but each recursive call just removes an element from one of the list (the next one in sequence). Working this one out may take you a little bit longer.

Finally, the following two are rather tricky.

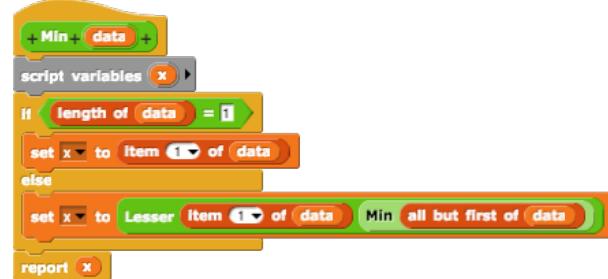
- ***append : list × list → list***: returns a list that is the concatenation of the two argument lists. To define this you need to think about the base case first. The result of appending a list  $x$  to an empty list is just that list  $x$ . So the recursion has to reduce one of the arguments until it becomes an empty list.
- ***reverse : list → list***: returns a list that contains all the elements of the argument list in inverse order. This one is particularly tricky. There are two different solutions for the definition of *reverse*. The simpler one reduces one of the arguments element by element until it becomes an empty list and relies on the above defined *append* function to construct the result of each recursive step. The second solution is much more efficient and elegant but looks confusing at first. It relies on passing an additional argument, a so-called *accumulator*, down the recursion to construct the result. A nice explanation can be found [here](http://www.siddharta.me/2006/04/recursion-part-2-tail-recursion.html) (<http://www.siddharta.me/2006/04/recursion-part-2-tail-recursion.html>). Accumulators are a somewhat advanced method in recursion, and **the use of accumulators is beyond the scope of this unit**.



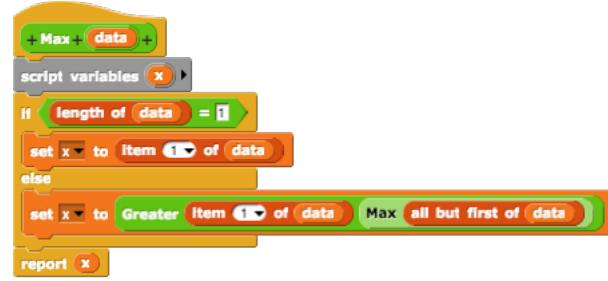
## 7.3.1.1 Solutions: recursive list idioms in Edgy

Note that this material is beyond the scope of the unit, but useful for deepening your understanding of the material.

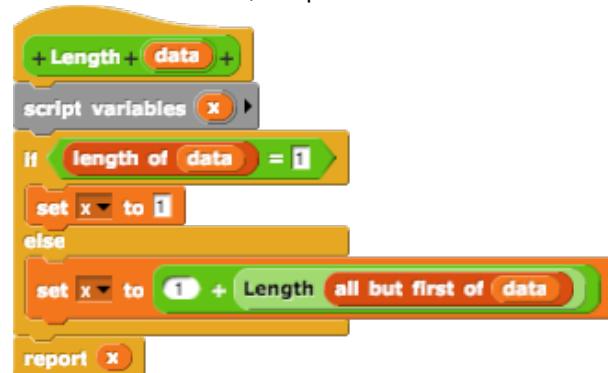
- *min : list → int* (finds the minimal element in a list of numbers)



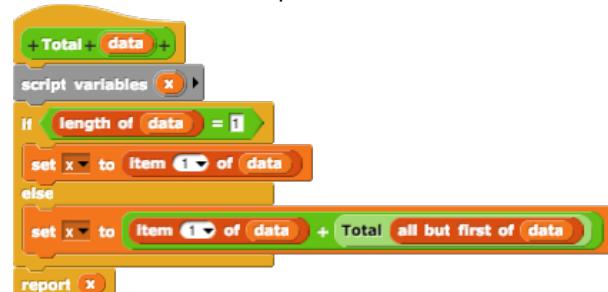
- *max : list → int* (finds the maximal element in a list of numbers)



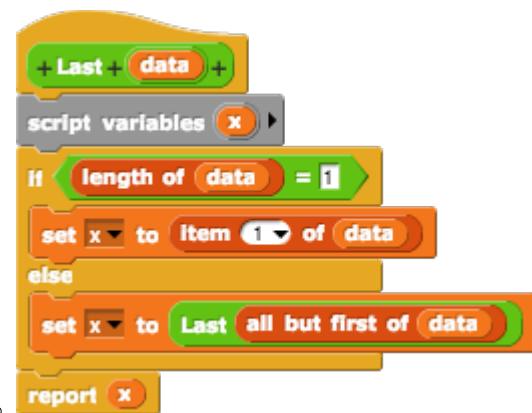
- *length : list → int*: (computes the number of elements of a list of numbers)



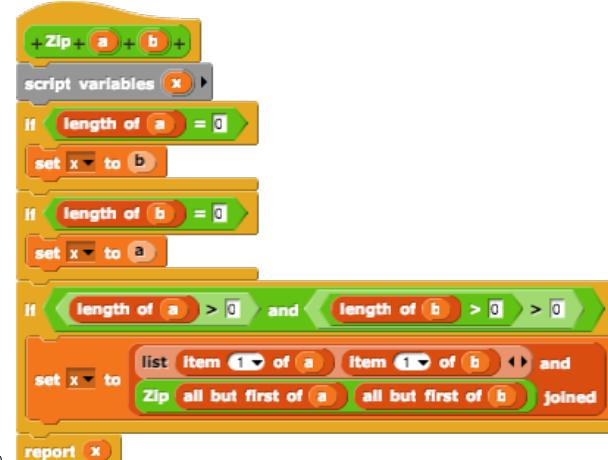
- *total : list → int*: (computes the total sum of a list of numbers)



- *last : list → elem*: (returns the last element of a list)



- ***zip*** :  $list \times list \rightarrow int$ : this function returns a list that contains the elements from both argument list in alternating order, for example,  $\text{zip}([1, 2, 3], [4, 5, 6])=[1, 4, 2, 5, 3, 6]$ . This should not be too tricky to work out. Each recursive level takes the first element from each of the lists simultaneously.



- ***merge*** :  $list \times list \rightarrow int$ : this function returns a list that contains all elements from both argument lists in sorted order provided that each argument list by itself is sorted already. For example,  $\text{zip}([1, 3, 4], [2, 5, 6])=[1, 2, 3, 4, 5, 6]$ . This works essentially like the *zip* above, but each recursive call just removes an element from one of the list (the next one in sequence). Working this one out may take you a little bit longer.

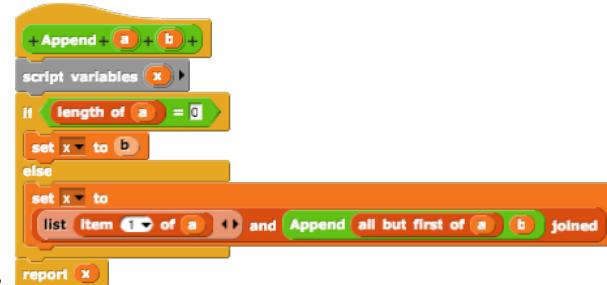




## 7.3.1.2 Solutions: Append and Reverse in Edgy

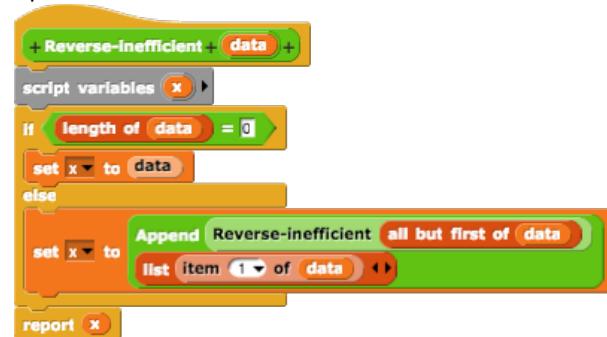
Note that this material is beyond the scope of the unit, but useful for deepening your understanding of the material.

- *append : list × list → list*: returns a list that is the concatenation of the two argument lists.

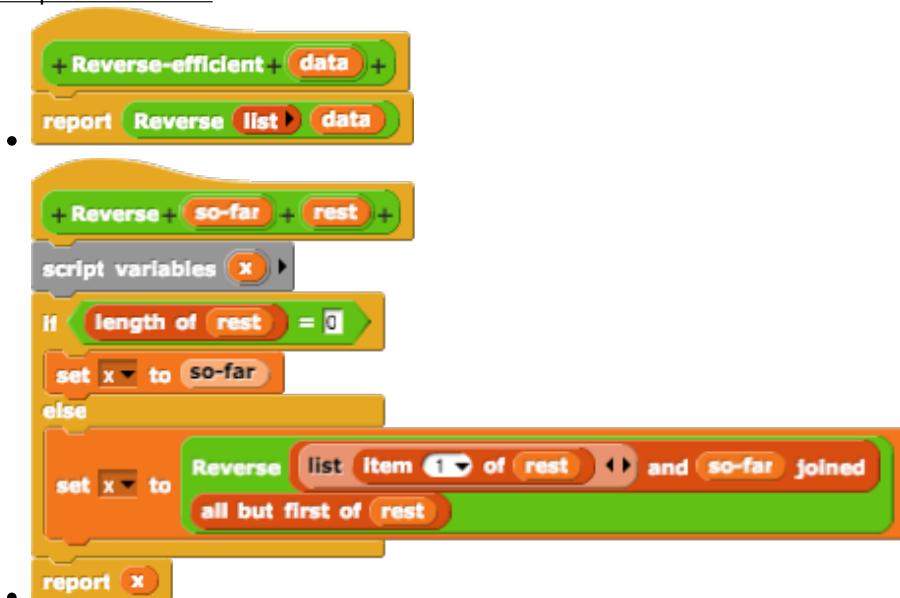


- *reverse : list → list*: returns a list that contains all the elements of the argument list in inverse order.

- The simple inefficient solution.



- The elegant, efficient solution that uses an *accumulator*. The use of accumulators is beyond the scope of this unit.





## 7.4 Testing Algorithm Correctness

### Introduction to Testing

A common way to design an algorithm is to write down something that seems reasonable and try it out. For example, suppose we want to check whether a graph is connected. After thinking about it for a minute, we decide to pick a node at random, then check whether the whole graph can be reached from that node. Here's a first attempt at defining a function to perform this work:

```
function connected():
    reachable = [random_node]
    while size(reachable) < size(graph)
        for n in reachable:
            foreach neighbour of n:
                add neighbour to reachable if not already there
    return size(reachable) == order
```

The algorithm returns True if and only if we were able to reach all of the nodes by following edges from a random node. However, it only works for *connected* graphs. [What does this algorithm do for unconnected graphs?<sup>1</sup>](#)

If you came up with this algorithm, you would surely think of testing it on at least one *disconnected* graph. You would think of this because connectedness is mentioned in the task. But would you have considered the case of disconnected graphs when testing, say, a shortest path algorithm? Would that be a useful thing to do or a waste of effort?

Suppose another student designed an algorithm for checking connectedness, and demonstrated that their algorithm handled the following two cases correctly:

$$\begin{aligned} G_1 &= (\{a\}, \{\}) - \text{connected} \\ G_2 &= (\{a, b\}, \{\}) - \text{disconnected} \end{aligned}$$

Would you be convinced that their algorithm was correct?

Neither case has any edges. However, the algorithm itself involves following edges, so it seems reasonable to include a test case that has an edge. For example:

$$G_3 = (\{a, b\}, \{(a, b)\}) - \text{connected}$$

If the algorithm behaved correctly for  $G_1$ ,  $G_2$ , and  $G_3$ , would you be satisfied that the other student's algorithm is correct?

Notice how the algorithm involves iterating over all the neighbours of a node. Accordingly, we'd like to see that it works when a node has more than one neighbour. And notice how the algorithm adds nodes to a collection of reachable nodes, and successively builds out from that. Accordingly, we'd like to see that it works when the graph contains paths. In other words, the algorithm should be tested on some non-trivial

cases, i.e. including connected and disconnected graphs involving several nodes, e.g.:

$$\begin{aligned} G_4 &= (\{a, b, c, d, e\}, \{(a, b), (a, c), (b, c), (c, d), (c, e)\}) - \text{connected} \\ G_5 &= (\{a, b, c, d, e\}, \{(a, b), (a, c), (b, c), (c, d)\}) - \text{disconnected} \end{aligned}$$

By now, you would probably be feeling fairly confident that the other student's algorithm is correct.

However, we run into a new problem. Testing the algorithm is time consuming! We need to test it after each modification. Thankfully, we can implement our algorithm in a programming language, and then we can automatically run the tests after each change to our program to check that it produced the expected values, i.e.:

$$\begin{aligned} \text{connected}(G_1) &= \text{True} \\ \text{connected}(G_2) &= \text{False} \\ \text{connected}(G_3) &= \text{True} \\ \text{connected}(G_4) &= \text{True} \\ \text{connected}(G_5) &= \text{False} \end{aligned}$$

The main purpose of testing, then, is to convince ourselves and others that the algorithm or implementation is correct. However, testing has other uses. We can test performance, e.g. see how slow it gets for large inputs. We can check that it does something sensible for erroneous inputs, e.g. discover that it enters an infinite loop when given a disconnected graph as input, even if the problem statement specified that the input was connected. A good set of tests can be useful more generally, for testing *any* algorithm that has been proposed for solving a particular problem.

Before going on, reflect back on the axioms of an ADT. For example, the stack ADT satisfies the axiom  $\text{pop}(\text{push}(x, s)) = x$ . I.e if you push an item  $x$  on stack  $s$ , then pop  $s$ , you get  $x$  back. The axioms for the stack ADT can be used to test the correctness of any implementation of the stack ADT. This is important: we will take some implementation, not inspect it, but merely check that it passes our tests, and then trust that it is correct. This approach is known as "black box testing".

## Black-box Testing

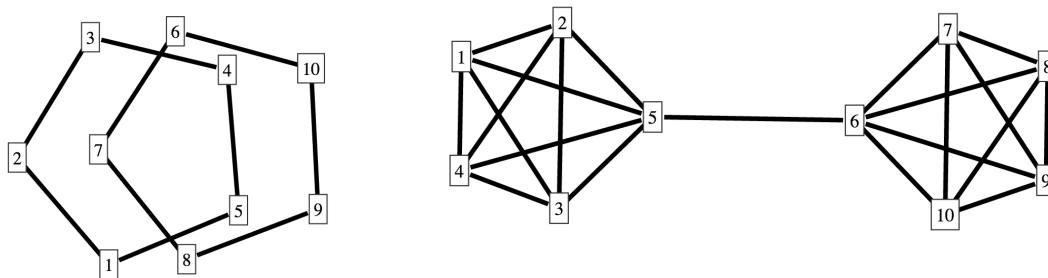
Black-box testing is an approach where we do not "look inside the box", i.e. we assume we've been given an implementation in a form that we cannot inspect; we can only run it. In this case, the only thing we have to go on when devising our test cases is the problem statement.

In the case of our algorithm for checking connectedness, we would start from the definition of connectedness: a graph is connected if there is a path between every pair of nodes; a graph consisting of a single node is connected. The definition mentions "single node" and "pair of nodes", so we would write down the same three cases as before:

$$\begin{aligned} G_1 &= (\{a\}, \{\}) \\ G_2 &= (\{a, b\}, \{\}) \\ G_3 &= (\{a, b\}, \{(a, b)\}) \end{aligned}$$

Then, we might focus on the use of the term "path", and think of cases where the path is longer than a single step, and devise larger test cases, covering both connected or disconnected graphs. We might even be a little creative, and try to think up some pathological examples which we think the implementer

might not have considered, for example:



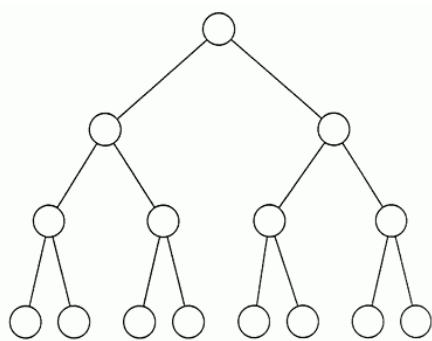
Here, we are looking for so-called "edge cases", cases which look like they might be near the boundary. Of course, they are not. Let's consider edge cases more carefully.

## Edge Cases / Boundary Values

Consider the following algorithm which generates a full binary tree of specified depth.

```
function generate_binary_tree(depth):
    node = new_node()
    if depth >= 0:
        child1 = generate_binary_tree(depth - 1)
        child2 = generate_binary_tree(depth - 1)
        add_edge(node, child1)
        add_edge(node, child2)
    return(node)
```

The depth of a tree is defined to be the length of the longest path from the root to a leaf. However, it's easy to forget whether we should be counting edges or counting nodes when working out the depth of a tree. Our tree of depth 3 contains paths of length 3, that pass through 3 edges. But the drawing of this tree has 4 levels of nodes:



Will the above algorithm generate this tree, when called with a depth of 3? Or does it contain a bug?

Any time we're given an algorithm that has an integer parameter, there's a risk of an "off-by-one" error: we generate a tree of depth 3 when asked to generate a tree of depth 4. Or is it, we generate a tree of depth 4 when asked to generate a tree of depth 3. That's right, it's hard to be sure which direction the off-by-one error will go.

Off-by-one errors occur all the time in algorithms. The above algorithm has a boundary value at zero: we can only create a tree of depth  $\geq 0$ . We need to test that it produces the correct result for this value, i.e.:

```
generate_binary_tree(0)
```

We should also check the values either side of this, i.e.:

```
generate_binary_tree(1)
generate_binary_tree(-1)
```

Notice that we have deliberately applied the algorithm to a value that is one step outside the boundary.

[What does the above pseudocode do for depth=-1?](#)<sup>2</sup>

## Error Guessing

Another approach to black-box testing is to draw on our experience of the sorts of cases that tend to cause problems. There are special cases of inputs that are easy to overlook. For example, our connectedness algorithm said to start with a random node. It made the assumption that the graph has at least one node. What does it do for the null graph, i.e. a graph having no nodes at all?

If our problem was to sort a list of integers, here are some cases we might want to check, that tend to cause problems:

- [] - the empty list
- [1] - a singleton list
- [-3] - a negative value
- [3, 3, 3, 3] - a list of identical values
- [1, 2, 3, 4, 5] - a list that is already sorted
- [5, 4, 3, 2, 1] - a list that is as far from being correctly sorted as possible

Here we are poking at the black box, seeing if we can break it by giving it inputs that its creator might not

have considered. So, for example, if we were testing the implementation of a stack, we might call the pop operation on an empty stack.

When we think of graphs, what might be some pathological cases to test? Here are some you might like to consider:

- $(\{\}, \{\})$  - the null graph
- $(\{a\}, \{\})$  - a singleton graph
- $(\{a\}, \{(a, a)\})$  - a graph with a self-loop
- $(\{a, b, c, d, e\}, \{\})$  - a graph with no edges
- $K_5$  - the complete graph on five edges
- a graph which is already a solution to the problem (e.g. for computing the minimal spanning tree, we give an input which is a tree, and expect to get the same tree back, not some new tree which also happens to be a minimal spanning tree)

## Pairwise Testing

Some functions have more than one parameter. For instance, the function to generate a tree of depth  $d$  and branching factor  $b$ . The function has two integer parameters where  $d \geq 0$  and  $b > 0$ . Now suppose we extended this function to allow us to specify the size  $s$  of nodes. A further boolean parameter allows us to specify whether the graph is directed. So our function is called like this:

`generate_tree( $d, b, s, directed$ )`

Notice that each parameter has two regions. For example, depth can be non-negative or negative, so we pick an exemplar for each. Now we do this for all the parameters:  $d = 1$  or  $-1$ ;  $b = 2$  or  $-2$ ;  $s = 3$  or  $-3$ ;  $directed = True$  or  $False$ . There are 16 possible combinations of these values.

In pairwise testing, also known as all-pairs testing, we test each combination of pairs, i.e.:

- `generate_tree(1, 2, 3, True)`
- `generate_tree(1, -2, -3, False)`
- `generate_tree(-1, 2, -3, False)`
- `generate_tree(-1, -2, 3, False)`
- `generate_tree(-1, -2, -3, True)`

Notice that all pairs of parameters have all possibilities tested. To verify this, pick any pair of parameters, e.g. the second and fourth, and observe the combinations:  $(2, True)$ ,  $(-2, False)$ ,  $(2, False)$ ,  $(-2, True)$ .

We constructed the above set of pairwise tests by holding one value fixed (in boldface) and enumerating all possibilities for the other values.

[EXAMPLE with a ternary variable, e.g. colour = red or yellow or green]

Pairwise testing is based on the observation that some bugs only exist for a combination of parameter values, but it is very unlikely that it will take an interaction of more than two parameters in order to expose the bug. Using this method on the above example, we reduced 16 test cases down to 5. As the number of parameters increases, and the number of test values per parameter increases, the reduction in effort becomes significant.

Our final step is to permit ourselves to inspect the code. Instead of a black box, we have a "white box" (or

"glass box") that allows us to see inside.

## White-box Testing

Unlike black-box testing, which starts from the specification, white-box testing starts from the pseudocode or the code. We inspect the code and devise a variety of inputs that causes every branch to be followed at least once.

[EXAMPLE with multiple paths]

The pseudocode or code may include components that were not explicitly stated in the problem definition. White box testing enables us to examine these components as well.

[EXAMPLE with auxiliary function]

## Voodoo Programming

Suppose that you have created a comprehensive test set and written code to run your implementation on every test case. Now, each time you modify your implementation, you can test it in a few seconds. Now suppose that all tests are passing except for one. So, you tweak your program and the same test fails. You tweak it again, and it still fails. Various friends make suggestions and you follow all of them, to no avail. After 20 minutes of this, your program has evolved in an almost random way. By this time, you're practicing "voodoo programming", making changes that you do not fully understand in the hopes that one of them will solve the problem. It is as if you're searching for the right magical incantation that will make the computer do what you want. This behaviour isn't so common when the tests are performed by hand.

Instead, it is better to step through the implementation with the input values from the failed test case, to locate the source of the problem. It could be that you used the wrong inequality symbol ( $<$  vs  $\leq$ ). At this point, you formulate a hypothesis that a certain modification to the implementation will fix the problem. If it does not, you need to understand why not, and reverse the change you made. It seems that you incorrectly located the source of the problem, and need to step through the implementation more carefully.

The temptation into voodoo programming occurs when it is difficult to understand the behaviour of the implementation, and when tests are easy to perform. A good approach is to test the sub-components of the implementation on their own. If there are no sub-components, you may need to break down your implementation into components that you can understand and test individually.

## Delving deeper (optional)

### Algorithms as Functions

An elegant approach to testing is to make sure that the algorithm or implementation is presented as a function (not a program), with clearly defined parameters and return values. Then we have rather simple schema for testing:

```
test_cases = [(G1, True), (G2, False), (G3, True), (G4, True), (G5, False)]
foreach (case, expected_result) in test_cases:
```

```

if connected(case) != expected_result:
    report failed test for case

```

Consider the case of a shortest path program that marks the path in the graph by colouring the edges. In order to test this program, we would need to run it, then inspect the resulting graph, checking if particular edges were the right colour (and that no other edges were that colour). It is more elegant to write our program as a function that constructs the path and returns it. The returned path can then be checked for (in)equality with the expected path.

[Note that it is (currently) difficult to do this in Edgy, because it is set up to modify a global graph. MORE]

[TODO: functions that modify an object in place, ie methods]

## Multiple Solutions

Sometimes there is not a unique solution. For instance, there can be more than one minimum spanning tree for a graph. We can deal with these cases in a variety of ways:

- avoid test cases that involve multiple solutions
- test that the result produced by the algorithm belongs to the set of known correct solutions
- require the algorithm to produce all solutions
- check some invariant property of the solution (such as the cost of the minimum spanning tree, which will be the same minimal value regardless of which tree is produced)

## Testing and Software Development

When we modify a program, it is tempting to only test the parts that have changed. We might have found that our program gave the incorrect result for a particular input, and we naturally check that the modified program gave the correct result for this input. However, it often happens that changing one part of a program effects other parts in unpredictable ways. It is better to run the entire set of tests after every change, in order to make sure that we haven't "regressed", introducing new bugs that have taken us further away from the desired solution. This is known as "regression testing".

In regression testing, each time we fix a problem, we add a new test to our test suite. Now, when we fix a new bug, we can be confident that we didn't re-introduce an earlier bug.

It is even possible to establish a series of tests before starting to do any algorithm design or implementation. We can write down test cases by reading the problem definition, then begin the development work with the most naive possible solution, and test it immediately. This is known as "test-driven development".

---

<sup>1</sup> It goes into an infinite loop.

<sup>2</sup> It generates a tree of depth=0 for all negative values of depth.

# **8**

## **Appendix B: The VCE study "Algorithmics"**

[8.1 VCE "Algorithmics"](#)

[8.2 Textbooks for VCE "Algorithmics"](#)

[8.3 Learning for Algorithmics and Computer Science](#)

[8.4 Recommended Progression](#)

[8.4.1 Recommended Progression - Unit 3](#)

[8.4.2 Recommended Progression - Unit 4](#)

---

## 8.1

# VCE "Algorithmics"

In the Victorian Certificate of Education, the new *Algorithmics* study examines how information about the world can be systematically represented and processed. Ultimately, this gives rise to computer programs. However, the focus of the study is not on coding as such, but on *algorithmic thinking*. Computer Science is as distinct from software development as theoretical Physics is from mechanical engineering. The study explores rigorous methods for analysing real world problems. A key part of this is identifying salient aspects of the real world that must be understood and represented to solve the problem. This is called building a computational model. Modelling is thus concerned with how we represent a problem in a formal way.

To solve a problem we then design algorithms that process these models. *Algorithm Design* is the central aspect of VCE Algorithmics. This is a complex and challenging topic. The same problem usually has a range of algorithmic solutions, and these can have markedly different properties. Sometimes, the addition of a small amount of memory will dramatically speed up an algorithm. Sometimes, presenting the inputs in a different sequence will dramatically slow down an algorithm. You will learn some powerful mathematical techniques to help you understand the behaviour of different algorithms and how to design an efficient algorithm.

Unit 3 sets out the basics of computational modelling and algorithm design. Unit 4 then expands on this with advanced paradigms for "better" algorithm design, i.e. clever techniques that allow us to make algorithms more efficient, often substantially so, and in some cases even to find solutions for problems that seem impossible to solve.

Unit 4 rounds this off with a look into deeper topics in computer science. It discusses how some problems are intrinsically harder than others, and shows some ways to identify and attack such hard problems. It also discusses general limits of computation, specifically that some information problems cannot be solved algorithmically at all. This naturally leads to a discussion of the possibility of artificial intelligence, and the prospects for creating new models of computation that are inspired by physical and biological systems.

Students will develop practical skills in algorithm development, that are independent of specific computer coding languages and transferrable between these. To achieve this, a very high level algorithm development environment will be used that abstracts from the technical details of industrial strength coding languages. This enables students to test their algorithms interactively and study their performance empirically.

### Aims of VCE Algorithmics

This study enables students to:

- understand the mathematical foundations of computer science and software engineering
- design algorithms to solve practical information problems, using suitable abstract data types and algorithm design patterns
- investigate the efficiency and correctness of algorithms through formal analysis and empirically through implementation as computer programs
- reason about the physical, mathematical and philosophical limits of computability.

## Presumed knowledge

The study will presume that students are already confident in the following areas:

- matrix addition, subtraction and multiplication
- sets and set operations (union, intersection, power sets)
- substitution and transposition in linear and non-linear relations
- the construction of tables of values from a given formula
- development of formulas from word descriptions
- sequences and linear relations generated by recursion
- logarithms and exponents
- the ability to produce and interpret numerical plots
- propositions, connectives and truth tables.

Students are expected to be currently enrolled in, or have successfully completed VCE Mathematical Methods (CAS) Units 1 and 2.

## Secondary and tertiary IT study

VCE Algorithmics provides the foundation for studying Computer Science and Software Engineering at tertiary level, and some universities will offer accelerated pathways to students who have completed this study. VCE Algorithmics also provides a conceptual framework for structured problem solving in Science, Engineering, and Technology (STEM), and other disciplines that involve formal reasoning.

Pathways at Monash University and The University of Melbourne are outlined in the following documents:

- Monash University: <http://it.monash.edu/algorithmics>
- The University of Melbourne: <http://www.cis.unimelb.edu.au/schools/algorithmics.html>

VCE Algorithmics complements VCE Software Development by providing the theoretical framework for designing and analysing algorithms which may ultimately be implemented in software systems. In VCE Algorithmics, programming plays a secondary role as a means of verifying and evaluating algorithms.

## 8.2

# Textbooks for VCE "Algorithmics"

There is no real a textbook for VCE Algorithmics. The online resources are for VCE Algorithmics in front of you are the equivalent of a traditional textbook.

However, there are some books that you might find interesting as supplementary materials to study the broader context. Chapters of these books will be referenced in some learning modules as further reading, however they are not required to study VCE Algorithmics and their contents is not in general aligned with this study.

- Algorithmic Adventures - From Knowledge to Magic. Juraj Hromkovic. Springer, Berlin, 2009.
- The New Turing Omnibus - 66 Excursions into Computer Science. A.K. Dewdney. W.H. Freeman, New York, 2001.

If you want to take your study of Algorithmics further there is a host of standard textbooks on the topic. However, these are all aimed at tertiary level and much of the materials is beyond the scope of the VCE study. The most accessible book on the formal details of Algorithmics is the following.

- Introduction to the Design and Analysis of Algorithms. Anany Levitin. Addison-Wesley, Upper Saddle River, 2012.

## 8.3

# Learning for Algorithmics and Computer Science

Computer science is very different from more familiar subjects such as English or Geography. CS concepts have been described as *tightly integrated* (Anthony Robins, CS Professor, University of Otago). What does this mean?

In CS, early achievement is critical: if you don't "get" early concepts then your misunderstandings are quickly magnified, and you're soon left far behind. Each new concept in CS provides an 'edge' where you can attach new pieces of knowledge.

### We learn at the edges of what we know

In computer science, ideas connect to each other like jigsaw pieces, more so than in other subjects you might be studying. As you join the pieces together, you expand your skills. If you want to avoid having your brain in a jumble, like the contents of jigsaw box emptied on floor, you need to start building things up one piece at a time.

These are the study skills you'll need:

- Check your own understanding constantly
- Don't "fool yourself" that you understand an idea, concept or topic: doing the exercises is crucial
- Help each other: explaining the idea to someone else is a great way to test your own understanding
- There's more than one way to solve a problem: observe how others do it and learn new problem solving strategies
- Sometimes new knowledge needs to incubate in your head for a while, so be patient while you wait for that eureka! moment

### Getting Help

There are several sources of help available to you:

- the subject materials and classes are the starting point and you should do your best not to fall behind because it is difficult to catch up
- your teach
- your peers
- the Web - there are many excellent resources on the Web and we will use some of them in classes, but you might find others as well

## **8.4**

# **Recommended Progression**

---

## 8.4.1 Recommended Progression - Unit 3

1. [1 week] Algorithmics - the heart of computer science
  1. What is an algorithm? (KK 2.1)
  2. Why Graphs and Networks (KK 1.5)
  3. The Relation Between Algorithmics and Coding (KK 2.1)
  4. Scope of the Subject and Overview
2. [15 weeks total] Algorithmic Problem Solving
  1. [2 weeks] Modelling Networks (KK 1.4, 1.5)
    1. What is modelling
    2. Modelling with graphs
    3. properties of graphs, including cyclicity, connectedness and distance
    4. example domains for graph modelling, especially social networks
    5. Which operations do we need to handle graphs (informal graph ADT)
    6. State Diagrams
  2. [1 week] First Network Algorithms (KK 1.5)
    1. Illustration of simple graph algorithms
    2. Following a given path
    3. Executing a state diagram
  3. [2 weeks] Traversal and Search in Networks (KK 2.5)
    1. Breadth First Version
    2. Depth First Search (DFS, iterative version only)
  4. [2 weeks] How to Write Algorithms (KK 1.1-1.3, 1.6-1.8, 2.1, 2.2, 2.10)
    1. Basic Concepts: Statements, Sequences, Variables, Loops, Decisions
    2. Modularization
    3. Collections
      1. When to use which collection type
    4. Abstract Data Types
      1. Concept
      2. ADTs for collections (list, stack, queue)
      3. ADT for graphs
      4. BFS, DFS revisited using proper ADT descriptions
  5. [1 week] Networks of Actions: Planning and Decision Making (KK 1.5, 2.5)
    1. Decision Trees
    2. river-crossing puzzle
    3. tic-tac-toe
  6. [1 week] Algorithmic Complexity: How Fast is my Algorithm - Informal Complexity (KK 2.4, 2.7)
    1. step counting
    2. experience complexity hands-on (e.g. shortest path: naive exponential solution vs Dijkstra)
  7. [3 weeks] Path Finding (KK 1.3, 4, 1.8, 2.6, 2.7, 3.2)
    1. Shortest Path (Dijkstra's algorithm and Bellman-Ford's algorithm)
    2. Minimum Spanning Tree (Prim, Reverse Deletion)
    3. Floyd's algorithm for the all-pair-shortest path problem
    4. Floyd-Warshall's algorithm for transitive closure
  8. [3 weeks] Recursion (KK 1.5, 2.3, 2.5)
    1. The Concept of Recursion
    2. What is Tail Recursion?
    3. Recursive DFS

4. Extensions of DFS
  1. Best First Search
  2. Iterated depth-limited DFS
  3. Minimax
3. Revision and Exam Preparation

## Remaining Key Knowledge Points

- KK 2.8: to be woven into all components by going through the design process with the students (rather than presenting the solution) and reflecting on the structure of the design process.
- KK 2.9, 3.1: to be attached to the discussion of selected algorithms by incorporating implementation and testing when these are discussed

Tests and Revision are assumed to be interleaved with the above.

---

## 8.4.2 Recommended Progression - Unit 4

NB: No learning outcomes mapping yet

1. [2 weeks] Formal Algorithm Analysis & Complexity
  1. [0.5 week] Working with Big-O
  2. [1.5 weeks] Applied Runtime Analysis  
(apply formal methods to Algorithms from Unit 3)
2. [5 weeks] Advanced Algorithm Design
  1. [2 weeks] Advanced Algorithm Paradigms
    1. [1 week] Divide & Conquer
    2. The Master Theorem
    3. [1 week] Dynamic Programming
  2. [2.5 weeks] Computationally Hard Problems
    1. [0.5 weeks] The Travelling Salesman Problem
    2. [0.5 weeks] NP-completeness: A firm concept of "hard"
    3. [1 week] Heuristics
      1. [0.5 weeks] Simple approximations: Minimum Cost Spanning Tree (MCST)-based Travelling Salesman Problem (TSP) solution
      2. [0.5 weeks] Randomized Search: Simulated Annealing for Graph Colouring
    4. [0.5 weeks] DNA Computing
    3. [0.5 weeks] Minimax & Game Trees
  3. [3 weeks] Universality of Computation and Algorithms
    1. [0.5 weeks] History of Computer Science
    2. [2 weeks] Hard Limits of Computation
      1. [1 week] Turing Machines and the Church-Turing Thesis
      2. [1 week] Undecidability and the Halting Problem
      3. [0.5 weeks] Searle's Chinese Room Argument
    4. [1 week + 1 week buffer] Revision

# **9**

## **Test Your Knowledge**

The review materials are designed to test your knowledge of each outcome.

---

[9.1 Review Unit 3, Outcome 1](#)

[9.2 Review Unit 3, Outcome 2](#)

## **9.1**

# **Review Unit 3, Outcome 1**

On completion of this unit the student should be able to devise formal representations for modelling various kinds of information problems using appropriate abstract data types, and apply these to a real-world problem.

~VCE *Algorithmics (HESS) Study Design 2017-2019*

### **Test Your Knowledge**

**Start Quiz** (<https://www.alexandriarepository.org/app/WpProQuiz/231>)

### **Modelling a real-world problem**

**Show "ReviewTask-open-ended.docx"**

([https://docs.google.com/viewer?url=https%3A%2F%2Fwww.alexandriarepository.org%2Fwp-content%2Fuploads%2F20170130150103%2FRviewTask-open-ended.docx&hl=en\\_GB&embedded=true](https://docs.google.com/viewer?url=https%3A%2F%2Fwww.alexandriarepository.org%2Fwp-content%2Fuploads%2F20170130150103%2FRviewTask-open-ended.docx&hl=en_GB&embedded=true))

**Download (DOCX, 22KB)** (<https://www.alexandriarepository.org/wp-content/uploads/20170130150103/ReviewTask-open-ended.docx>)

---

## 9.2 Review Unit 3, Outcome 2

On completion of this unit the student should be able to design algorithms to solve information problems using basic algorithm design patterns, and implement the algorithms.

~VCE Algorithmics (HESS) Study Design 2017-2019

### Test Your Knowledge

**Start Quiz** (<https://www.alexandriarepository.org/app/WpProQuiz/235>)

### Real-world Problem

A **palindrome** is a word that is spelled the same forward and backward. For example, *rotor* and *redder* are palindromes, but *motor* is not.

Construct a recursive algorithm that when given a string of any size as input, the algorithm will return True if the string is a palindrome, otherwise False.

---