





# Course Book

---

## Algorithmics

DLMCSA01

## Masthead

Publisher:

IU Internationale Hochschule GmbH  
IU International University of Applied Sciences  
Juri-Gagarin-Ring 152  
D-99084 Erfurt

Mailing address:

Albert-Proeller-Straße 15-19  
D-86675 Buchdorf

media@iu.org  
www.iu.de

DLMCSA01

Version No.: 001-2022-0817

© 2022 IU Internationale Hochschule GmbH

This course book is protected by copyright. All rights reserved.

This course book may not be reproduced and/or electronically edited, duplicated, or distributed in any kind of form without written permission by the IU Internationale Hochschule GmbH.

The authors/publishers have identified the authors and sources of all graphics to the best of their abilities. However, if any erroneous information has been provided, please notify us accordingly.





## Module Director

### **Prof. Dr. Paul Libbrecht**

Mr. Libbrecht has been a lecturer in Computer Science at IU International University of Applied Sciences since 2020. His main areas are the World Wide Web, data management, and general computer science.

Mr. Libbrecht studied Mathematics at the University of Lausanne (Switzerland) and the Université du Québec à Montréal (Canada). He received his doctorate in Computer Science from Saarland University (Germany). He has been a substitute professor at the University of Education Weingarten (PH Weingarten) and senior developer at the Leibniz Institute for Research and Information in Education. He is a member of the W3C Math Working Group and has been active in the OpenMath Society.

Mr. Libbrecht's research focusses on the technology of learning systems, often with a focus on mathematics. He has published in international conferences and journals. Since 2010, he has worked as a web development consultant for German, French, and US companies.

# Table of Contents

## Algorithmics

Module Director ..... 3

### **Introduction**

Algorithmics ..... 7

Signposts Throughout the Course Book ..... 8

Learning Objectives ..... 9

### **Unit 1**

Introduction to Algorithmics ..... 12

1.1 Basic Concepts and Historical Overview ..... 12

1.2 Algorithms, Programming Languages, and Data Structures ..... 15

1.3 Quality Algorithms: Correctness, Accuracy, Completeness, and Efficiency ..... 20

1.4 The Role of Algorithms in Society ..... 21

### **Unit 2**

Algorithm Design ..... 30

2.1 Data Structures ..... 30

2.2 Recursion and Iteration ..... 56

2.3 Divide-and-Conquer ..... 60

2.4 Balancing, Greedy Algorithms, and Dynamic Programming ..... 60

### **Unit 3**

Some Important Algorithms ..... 68

3.1 Searching and Sorting ..... 68

3.2 Pattern Matching ..... 81

3.3 The RSA Algorithm ..... 89

3.4 The K-Means Data Clustering Algorithm ..... 91

**Unit 4**

Correctness, Accuracy, and Completeness of Algorithms	100
4.1 Partial Correctness	100
4.2 Total Correctness	104
4.3 Ensuring Correctness in Day-to-Day Programming	110
4.4 Accuracy, Approximation, and Error Analysis	124

**Unit 5**

Computability	134
5.1 Models of Computation	134
5.2 The Halting Problem	146
5.3 Undecidable Problems	147

**Unit 6**

Efficiency of Algorithms: Complexity Theory	152
6.1 Models of Complexity	152
6.2 NP-Completeness	168
6.3 $P = NP?$	171

**Unit 7**

Advanced Algorithmics	174
7.1 Parallel Computing	174
7.2 Probabilistic Algorithms	182
7.3 Quantum Computing and the Shor Algorithm	197

**Appendix 1**

List of References	208
--------------------	-----

**Appendix 2**

List of Tables and Figures

212

# Introduction

## Algorithmics



# Signposts Throughout the Course Book



## Welcome

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

# Learning Objectives



The course **“Algorithmics”** aims to present the theoretical and the historical foundations of computer science and explore key concepts on the design and programming of algorithms. Moreover, it looks at the societal implications of the use of the computing applications that are born from the programming of algorithms. It also presents some of the futuristic paradigms of algorithms.

The first part of the course starts with the above mentioned historical and societal perspectives. It then focuses on searching, sorting, pattern-matching, clustering, and encryption algorithms, and on the use of key data structures and algorithm design strategies. The theoretical foundations of computer science are presented in the second part of the course where models of computation, complexity, completeness, and correctness are discussed. In the last part of the course, we single out parallelism, probabilistic algorithms, and quantum computing as some of the innovative paradigms for the future of algorithms. Links to the [IUBH GitHub repository for this course](#) have been provided throughout this script. You are encouraged to use this resource for practice.





# Unit 1



## Introduction to Algorithmics

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... basic concepts of algorithmics.
- ... key issues in the algorithm quality and the difficulty levels of computational problems.
- ... important features of the interplay between algorithms, programming languages, and data structures using the JavaScript programming language.
- ... the historical and the contemporary role of algorithmics.

# 1. Introduction to Algorithmics

## Introduction

Computer programming is widely perceived as the holding block of the following three main technologies of the Fourth Industrial Revolution: digital, biological, and physical. However, a computer program is nothing more than an algorithm written in a given programming language. In other words, algorithms and algorithmics (the study of algorithms) are the real holding blocks of Industry 4.0 (also known as the Fourth Industrial Technology). One of the highlights of this unit will be to single out this cornerstone role of algorithmics in the Fourth Industrial Revolution.

In addition to the discovery of the history of algorithmics, we study the definitions of basic algorithmic concepts, as well as how to code algorithms in the JavaScript programming language.

Some of the concepts introduced in this unit are related to the design of algorithms and their evaluation in terms of their correctness and of their complexity. Finally, some of the theoretical components of algorithmics such as the complexity theory and the computability theory are introduced.

## 1.1 Basic Concepts and Historical Overview

Before looking at the history of algorithmics, it seems important to first briefly define the concept. An algorithm is simply a sequence of steps toward the resolution of a computational problem. After a historical overview, this section discusses computational problems and computability and explains how algorithms differ from algorithmics and programs.

### History of Algorithmics

Many perceive Ancient Iraq and Ancient Egypt as the cradle of civilization. Consequently, it is not surprising that they are also considered the cradle of algorithmics. Ancient Greek and Italian mathematicians also played an important role. Fibonacci was key in the dissemination of the work of al-Khwarizmi, the Persian mathematician from Khorasan (present-day Uzbekistan and Iran) whose name gave birth to the term “algorithm.” Turing and other contemporary Western mathematicians such as Gödel and Hilbert were also instrumental in the development of algorithmics.

#### **Ancient Iraq and Ancient Egypt**

The Sumerian people of Ancient Iraq are known as the first people to have used algorithms around 2500 BCE, for example, in the teaching of geometry and basic operations such as division on clay tablets.

## Introduction to Algorithmics

For these calculations, their number system used decimal numbers within a sexagesimal system. Thereafter, between 2000 and 1650 BCE, the Babylonians, another Ancient Iraq people, also wrote algorithms on their tablets to teach other mathematical concepts, such as inverting numbers, squaring roots, or solving algebraic quadratic equations. Similarly, Ancient Egypt used papyrus between 1900 and 1800 BCE to write algorithms for the teaching of geometrical concepts, such as surface areas, geometrical series, and volumes.

### **Ancient Greek civilization**

Around 500 BCE, Ancient Greece ruled the world both militarily, politically, and academically. We know it as the “cradle of democracy.” Military commanders such as Alexander the Great expanded the Greek empire to Egypt. Moreover, it was home to legendary philosophers such as Plato, Aristotle, and Socrates, as well as brilliant mathematicians such as Pythagoras, Thales, Euclid, and Eratosthenes. The renowned Hellenistic mathematicians Eratosthenes and Euclid residing in Libya and in Alexandria, respectively, found the first algorithms on the identification of prime numbers and on the calculation of the greatest common divisor of two numbers.

### **Ancient Islamic translations**

In 820, the House of Wisdom was built by Muslim rulers, such as al-Mamun in Ancient Baghdad, with the initial purpose of translating Hellenistic and other manuscripts into Arabic (Ausiello, 2013). The house ultimately became a vibrant, intellectual, and scientific hub that hosted a wide range of scholars from diverse origins, languages, and religions.

This is how al-Khwarizmi introduced the number zero from the Indian numbers system to the Arabic numbers system. He is also credited for the first detailed specification of the main four basic arithmetic operations: addition, subtraction, multiplication, and division.

### **The Byzantine Empire**

The Roman Empire split into the Western Empire and the Eastern Empire in 390. The Western Empire ended around 480, but the Eastern (or Byzantine) Empire only ended around 1453. History teaches us that Pisa (present-day Tuscany) held a key maritime position in the life of the Byzantine Empire. This is where the young Italian Fibonacci was born around 1170. He studied at Bugia (in what is now Algeria) but also traveled intensively, both as a trader and as a scholar, in many Mediterranean countries where he discovered the work of Hindu-Arabic scholars such as al-Khwarizmi. Fibonacci became quite skillful at merging al-Khwarizmi’s Hindu-Arabic algorithms with Euclidian mathematics. His work contains algorithms and mathematical solutions for various application domains such as mathematics, accounting, and games, even though his name is usually associated in the history of algorithmics with the “Rabbit Problem” and the “Fibonacci Sequence.” Interested readers may find an account of the history of mathematical notations and how they traveled in Mazur (2014).

### **Post-Renaissance and contemporary Western mathematicians**

Historical hardware devices sustained the vibrancy of algorithmic activities, starting with the previously mentioned clay tablets and the papyrus used in Ancient Iraq and in Ancient Egypt, respectively. The invention of “algorithmic mechanical machines” (e.g., Joseph Marie Jacquard, Charles Babbage) also constitutes a key moment in the history of algorithmics. Jacquard was an eighteenth-century French inventor who created programmable sewing machines for the silk industry. His contemporary, Charles Babbage, was an English mechanical engineer and mathematician whose groundbreaking inventions included how to use punch cards to automate astronomical calculations.

Around 1900, the German mathematician Hilbert undertook the ambitious task of proposing an algorithm or method that could decide on the veracity of any given mathematical proposition or statement. However, the Austrian mathematician Gödel published a clear proof in 1931 that no algorithm or method can decide the veracity of any given mathematical proposition or statement. In 1936, Turing confirmed Gödel's work and hypothesized that if there is a proof of the veracity of a given mathematical proposition or statement then that proof can be confirmed by a Turing machine (Ausiello, 2013).

With its rich history in mind, we move to the overview of the present state of algorithmics.

### **Algorithmics Basic Concepts**

This subsection is a presentation of the definition of basic algorithmics concepts such as computational problems, computability, as well as the difference between algorithms and algorithmics.

#### **Computational problems**

The purpose of algorithms is to solve computational problems that are made up of three components: inputs with their preconditions, outputs with their post-conditions, and a set of relationships between inputs and outputs. The problem of the identification of the highest common divisor between two strictly positive numbers is described in the next paragraph as an example of a computational problem.

The inputs of the aforementioned computational problem are two natural numbers that must be strictly positive (precondition). The output will be a number with the post-condition of being the highest common divisor of the two inputs.

As for the relationship between the inputs and the output, it simply portrays the output as the biggest natural number that is both a divisor of the first input and a divisor of the second input. Similarly, it is not difficult to specify the primality test problem of deciding whether or not a given strictly positive natural number is prime.

From a Turing machine's perspective, it is hypothesized that we can always run the algorithm of a computational decision problem once we are in possession of such an algorithm.

Decision problems are computational problems whose output simply yields Boolean answers such as yes/no or true/false as opposed to non-decision problems. One can easily identify the above described primality test and highest common divisor problems as a decision problem and as a non-decision problem, respectively.

### Computability and decidability

A computational problem is said to be computable if there exists an algorithm that can solve it, i.e., if a Turing machine exists for it. For example, if the above specified problem of the identification of the highest common divisor between two strictly positive numbers is computable, so is the one of the primality test of a number. However, many other computational problems, for example, the classical Halting Problem, are not computable. The dichotomy between decision problems and non-decision problems also makes it possible to speak of decidability as a synonym of computability for decision problems; this does not make sense for non-decision problems for which computability is only synonymous with solvability. Thus, we say the primality test problem is decidable and the highest

### Algorithms and algorithmics

An **algorithm** is a finite sequence of doable (by a Turing machine) steps aimed at solving a given computational problem. Consequently, algorithmics is the study of algorithms and of computational problems since many computational problems are known to be non-computable. While algorithmics examines the computability, solvability, and decidability of computational problems as highlighted above, it also studies the design and the analysis of algorithms as introduced below.

**Algorithm**  
An algorithm solves a computational problem.

## 1.2 Algorithms, Programming Languages, and Data Structures

It seems natural to present algorithms in the form of pseudocodes so that they can be easily understood by humans in their natural languages. However, in order for an algorithm to solve its computational problem with the help of a computer, the pseudocode of that algorithm must first be translated into a program that will then be executed by a computer, together with the relevant control and data structures. In contrast to pseudocodes that are written in natural languages, computer programs are written in **programming languages**.

### Programming Languages

Many programming languages are available algorithms, depending on the nature of the computational problems. This course has adopted JavaScript as its programming language mainly because of its ease of deployment on Web browsers. These were written and tested in NodeJs, as visible in the following example.

**Programming language**  
A programming language automates the processes leading to the execution of algorithms.

### “Hello There”

Please follow the steps below to write and execute our “Hello There” JavaScript program. This program asks its users to enter their name and it displays on the screen a “Hello” message to that name followed by the “Kind Regards,” with each output on a separate line. It is assumed that readers are familiar with the basic concepts of programming and efforts will be made to always write self-explanatory JavaScript examples. Non-trivial code segments will always be explained.

1. Open a text editor (e.g., Notepad, Notepad++, Edit, GEdit, BBEdit, and Atom), type the following code, and save your file with a name of your choice and with a `js` extension (Use the option `All files` for the dropdown list `Save as type`).

#### “Hello There” Example

```
let readline = require('readline-sync');

while (true){
  let name = readline.question('\nWhat is your name? ');
  // \n is the character to go to a new line
  console.log('Hello ' + name);
  // The plus sign stands here for the concatenation operator
  console.log('Kind Regards');
}
```

We see from that the characters `//` are used for short comments not exceeding one line. The `readline.question` instruction is also very important here because of its role in the identification of the inputs of the program, and a similar role is played by the `console.log` instruction with regards to the display of the outputs of the program.

2. Assuming that `HelloThere.js` is the name that you have given to your program, type the `node HelloThere.js` command on the command prompt to run your program. If that command complains that it does not recognize the `readline-sync` module, then type the `npm install readline-sync` command on the command prompt to install it. You are encouraged to run each program and even modify the program with your own ideas and use a debugger (e.g., in Eclipse, VisualStudio Code, netbeans WebStorm) to see it walk through its steps. These are two excellent methods for understanding the work of an algorithm.

### Greatest Common Divisor (GCD)

As previously stated, the greatest common divisor (GCD) problem consists in identifying the highest natural positive integer that divides two given strictly positive natural numbers.

A simple algorithm for the GCD problem is to start by assuming that the smallest of the two given numbers is the highest divisor of these two numbers. Thereafter, that assumption must be tested: If it is true, then the highest common divisor would have been found. Otherwise, we have to reduce the value of currently assumed GCD by one and test that new assumption. That iterative process of assumption and testing will

ultimately end with a GCD value greater or equal to one because one divides all strictly positive natural numbers. N. B. Euclid is credited as the inventor of the first GCD algorithm. The code below can be found [here](#).

### GCD Naive Algorithm

```
let readline = require('readline-sync');

let gcd = function(n1, n2) {
  let r = Math.min(n1, n2);
  while ( !( ((n1%r)===0) && ((n2%r)===0) ) ) {r--;}
  return r;
}

let strictlyPosInt = function(s){
  let n = Number(s);
  return((s!=='') && (Number.isInteger(n)) && (n>0))
}

while (true){
  let mess1 = '\nEnter the first strictly positive integer ';
  let mess2 = 'Enter the second strictly positive integer ';
  let mess3 = 'Only strictly positive integers please!';
  let ns = readline.question(mess1);
  let ms = readline.question(mess2);
  if ((strictlyPosInt(ns)) && (strictlyPosInt(ms))){
    let n = parseInt(ns); let m = parseInt(ms);
    console.log('GCD between '+n+' and '+m+' is '+gcd(n,m));
  }
  else {console.log(mess3);}
}
```

By default, it is usually assumed that users enter strings from the keyboard, so we have to use `parseInt` to tell JavaScript that the value that is entered is an integer. The definition and the use of the function `gcd` also deserve our attention. Similarly, from this program we learn how to calculate the minimum between two numbers (`Math.min`), calculate the remainder or modulus between two numbers (`%`) (e.g., `26 % 7` is equal to 5 in the sense that when we divide 26 by 7, we get 3 but there is a remainder of 5), test the equality of two entities (`===`), negate a Boolean value (`!`), and decrement the value of a variable (`--`). The syntax of `while` loops and `if` function can also be learned from this example, especially with the use of caliber brackets (`{}`). The use of the keyword `return` is also very important for functions.

## Data Structures

The `HelloThere` and `GCD` examples deal mainly with one or two integers or strings whose values are collected from users. Those values are the data of the program. However, there are many situations where an algorithm or a program has to collect data of

a more diverse nature either in terms of quantity or in terms of varieties. Consequently, this forces algorithms and programs to organize or structure their data into relevant data structures, such as arrays, lists, sets, trees, or graphs. For example, for the computational problem of sorting a sequence of numbers in ascending order, it would be appropriate to use an array for the storing and the processing of these numbers, as in the [selection sort JavaScript](#) example below.



## Selection Sort Algorithm

```
let readline = require('readline-sync');

let captureNumbers = function (n1,nbs){
  let i=0;
  while(i < n1){
    let mess = 'Please enter your next number ';
    let v = readline.question(mess);
    if ((isNaN(v)===false) && (v!=='')){
      nbs.push(Number(v));
      i++;
    }
  }
  return;
}

let displayNumbers = function (mess, nbs){
  console.log(mess);
  console.log(nbs);
  return;
}

let sortNumbers = function (nbs){
  for (let i = 0; i < nbs.length; i++){
    for (let j = i; j < nbs.length; j++){
      if (nbs[j]<nbs[i]){
        v = nbs[i];
        nbs[i] = nbs[j];
        nbs[j] = v;
      }
    }
  }
  return;
}

let strictlyPosInt = function(s){
  let n = Number(s);
  return((s!=='') && (Number.isInteger(n)) && (n>0))
}

while (true){
  let numbs = [];
  let mess = '\nPlease how many numbers do you have? ';
  let ns = readline.question(mess);
  if (strictlyPosInt(ns)===true){
    let n = parseInt(ns);
    captureNumbers(n,numbs);
    displayNumbers("Unsorted sequence of numbers",numbs);
    sortNumbers(numbs);
    displayNumbers("Sorted sequence of numbers", numbs);
  }
}
```

The selection sort algorithm swaps the smallest value from the first position of the array with the value in the first position, then it swaps the smallest value of the array from the second position with the value in the second position, then it swaps the smallest value of the array from the third position with the value in the third position, and so on, until it reaches the end of the array.

In the JavaScript program above, the array `numbs` has been initialized with the empty value `[]`. In order for the `captureNumbers` function to fill it with values, it must push those values one by one at the end of the array. Moreover, the `captureNumbers` function uses the `Number` function instead of the `parseInt` function simply because this computational problem accepts both integers and real numbers as inputs. It is also important to note that all three functions of this JavaScript program are marking use of the return keyword without returning any value, simply because their purpose is to accomplish their intended tasks without necessarily returning one specific value. In fact, two of these functions are modifying the array that is passed to them as a parameter either by filling it with values, or by sorting it, and the third function simply displays the content of its array on the screen. When an array is passed as a parameter to a function, its content can be changed by that function if it contains the necessary instructions to do so. This is possible because, in JavaScript, most parameters are passed to functions by value, except for objects parameters that are passed to functions by address, and JavaScript arrays are actually objects. Readers are reminded that parameters that are passed by value to a function cannot be changed by that function, as opposed to the ones that are passed by reference or by address.

## Control Structures

From an algorithm's perspective, the two main control structures are conditions and loops. Conditions include different forms of `if` statements and loops are used for the iteration or repetition of sequences of instructions. The purpose of these control structures is to indicate how instructions will follow one another. For example, the aforementioned `sortNumbers` function has a condition inside a double loop.

## 1.3 Quality Algorithms: Correctness, Accuracy, Completeness, and Efficiency

When proposing an algorithm for a given computational problem, we have to ensure that such an algorithm is correct, accurate, complete, and efficient.

### Correctness

The correctness of an algorithm can only be established through a mathematical correctness proof of its partial or total correctness. An algorithm is totally correct if there is a mathematical proof that the algorithm is partially correct and, for all its inputs that fulfill its preconditions (correct inputs), the algorithm will always terminate. An algo-

rithm is said to be partially correct if one can prove mathematically that it has two types of inputs fulfilling its preconditions: the ones for which the algorithm does not terminate and whose outputs are therefore unknown, and the ones that terminate and whose outputs fulfill its post-conditions (correct outputs).

## Completeness

An algorithm is said to be complete if, for all its correct inputs (inputs that fulfill its preconditions), it always renders the correct outputs (outputs that fulfill its post-conditions) for the given inputs when there is a solution for such inputs, or it terminates with a “no solution found” message when there is no solution for such inputs. Let’s consider the computational problem of looking for the position of the first occurrence of a decimal digit (0..9) in a given non-null string. Any algorithm that will ignore the fact that certain strings do not contain decimal numbers will not be complete.

## Accuracy

When inputting a value to a computational problem or when calculating the value of one of its outputs, it is important for such values to have the desired level of accuracy, i.e., to be as close as possible to the reality that those values are supposed to represent. The issue of accuracy is important for approximation algorithms whose purpose is to make an approximation of an expected value. Such algorithms will be considered inaccurate if their approximations are too different from the expected value.

## Efficiency and Complexity

Let us recall that an algorithm is a finite sequence of doable (by a Turing machine) steps aimed at solving a given computational problem, using appropriate data structures. This is why the efficiency of an algorithm can be assessed either in terms of space or in terms of time. Space efficiency has to do with the amount of memory that is required for the data structures of an algorithm. As for **time efficiency**, it has to do with the number of steps of an algorithm. Complexity has more to do with the level of difficulty of a given computational problem.

Time efficiency  
This quality indicates the speed of a given algorithm.

## 1.4 The Role of Algorithms in Society

It is important to note that the previous definition of algorithms is different from the one that has widely been adopted by modern societies. Let’s first define algorithms from a societal viewpoint before discussing the opportunities that they create and their associated challenges. Modern societies consider algorithms powerful elements of computing devices and applications whose mission it is to assist as much as possible in the management of our lives in decision-making choices as well as the execution of activities.

## Application Domains

Algorithms have so far mostly influenced modern life: search engines, public-key cryptography and digital signatures, errors correction, patterns matching, data compression, and databases (MacCormick, 2013).

### Search engines

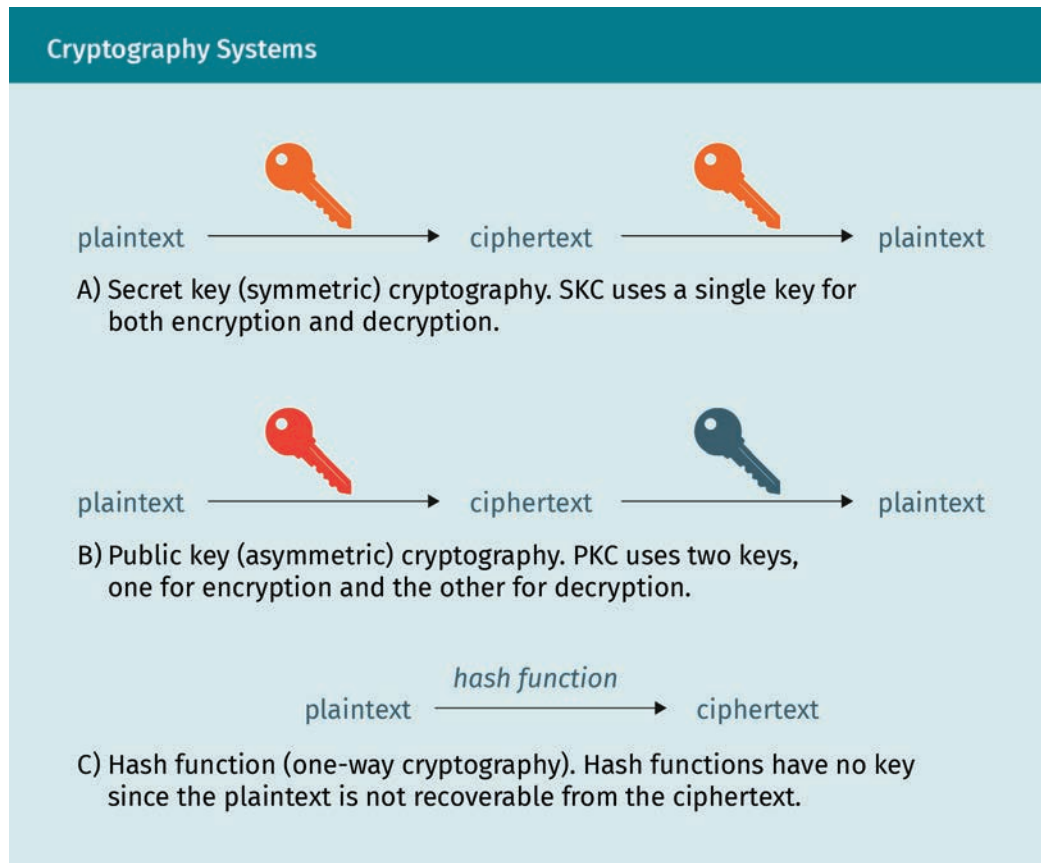
The World Wide Web is perceived by many as a gigantic collection of pages that are deployed in a worldwide distribution and contain valuable information for its users. Search engines allow their users to query the Web in order to select the right information from all those pages. Currently, the most popular search engines are owned by Google and Microsoft.

### Public-key cryptography and digital signatures

Public-key cryptography and digital signatures are mainly used to secure information. These two concepts are illustrated using the example of a box whose role is to transmit messages between senders and receivers (Vryonis, 2013).

In private-key or symmetric cryptograph, we have a single private key that locks and unlocks the box. Any key holder can hide the content of the box by locking it (encryption) and only the other holders of a copy of that private key can unlock it and see its content (decryption).

Public-key or asymmetric cryptography works a bit differently. Suppose we have an initially closed single box with a single clockwise-turning private key held by only one person. There are multiple copies of an anticlockwise-turning public key with the middle vertical position ( $\uparrow$ ) of the lock being the unlocked position and both the left ( $\leftarrow$ ) and the right ( $\rightarrow$ ) horizontal positions being the locked ones. Anyone who puts something in the box or takes something from it must close it immediately after, and two people cannot use the box at the same time. When the lock of the box is in the right ( $\rightarrow$ ) horizontal position, anyone (including the private key holder) can open the box with the public key, put things inside, and lock it with the public key (encryption); however, only the private key holder can unlock it (decryption). Similarly, if a public key owner opens that box and finds a packet in it, then they are certain that the packet was sent by the private key owner. That private key serves here as the digital signature of its owner. A representative diagram of different cryptography systems is displayed below from Kessler (2020).



### Error correction

Computing devices, their applications, and algorithms handle large quantities of data that sometimes contain errors. The purpose of error correction algorithms is to automatically detect and correct such data errors. Let's illustrate this purpose by checking the validity of a 13-digit ISBN code.

First, we have to multiply its even positions digits by three and add all these results to the sum of all the other digits except for the thirteenth. The second step is to calculate the remainder from the division of that total by ten. If the addition of that remainder to the thirteenth digit gives ten, then the ISBN code is valid; otherwise, it is not.

Let's consider the ISBN code 978-3-540-48663-3. Its odd positions digits are 7, 3, 4, 4, 6, and 3. Their multiplications by three will give 21, 9, 12, 12, 18, and 9, whose sum is 81. When adding 81 to the other digits except for the 13th one (9, 8, 5, 0, 8, and 6), we get a total value of 117. The natural division of 117 by 10 yields a value of 110 with a remainder of 7, and the addition of 7 to the 13th digit gives 10. So this ISBN code is valid.

Error detection and correction algorithms are used in various application domains, such as fixed and mobile storage devices or discs, internet packets transmissions, and cell phone calls.

### Data compression

Data compression consists of taking advantage of the existence of redundancies in data in order to present them in a more compact form. For example, the Run Length Encoding (RLE) compression algorithm takes a sequence of binary digits and transforms it to another sequence of binary digits based on the length of the longest sequence of repetitive bits.

We will use the binary sequence 111111111111111100001111111111 to illustrate the concept of data compression. In this bit sequence, there are seventeen 1s followed by four 0s and eleven 1s. We must use the number of bits of seventeen for our compression: 17 is converted to 10001 which consists of five bits. So we must also convert 4 and 11 to binary on five digits: 4 will be converted to 00100 and 11 will be converted to 01011. It is now time to say that our binary sequence has seventeen 1s followed by four 0s and by eleven 1s and obtain the compressed binary sequence 110001000100101011 of the initial binary sequence. Note that the initial sequence had 32 bits, but its compressed version has 18.

### Pattern matching

Pattern matching algorithms aim to check the occurrence(s) of a given pattern in a given object, using, for example, the regular expressions (RegExp) facilities that are available in current programming languages. For instance, the [following JavaScript code](#) illustrates the matching of two patterns in a short present tense sentence input by users in order to check whether such sentences are grammatically correct. This example assumes that such sentences are simply made up of a pronoun (I, you, he, she, it, we, and they) that is followed by a space and a verb. Pattern matching algorithms are used in several applications, such as search engines, machine learning, information security, virtual reality, DNA sequencing, and pattern recognition (characters and images).

#### Third Person Present Tense Pattern

```
let readline = require('readline-sync');

while (true){
  let m = '\nShort present tense sentence with a pronoun and a
  verb';
  m= m+'. Start with a capital letter but do not end with a dot\n';
  let t = readline.question(m);
  let pat3s = /^(He|She|It)\s([a-z]*)s$/;
  let patOp = /^(I|You|We|They)\s([a-z]*)[^\s]$/;
  if ((t.search(pat3s)!=0)&&(t.search(patOp)!=0)){
    console.log('Sentence ' + t + ' is grammatically incorrect');
  }
  else{
    console.log('Sentence ' + t + ' is grammatically correct');
  }
}
```

### Databases

In a database, data is structured and kept in a repository making it easy for users and programmers to store data and retrieve information. The repositories are currently almost omnipresent in the management of all organizational activities dating back to 1960. Popular Database Management Systems (DBMS) include Oracle, SQL Server, DB2, MySQL, Access, MongoDB, Solr, and CouchDB.

### Challenges

In modern societies, algorithms tend to use personal data to help them manage their activities and decision-making choices. However, such algorithmic societies are exposed to several challenges such as the applicability of current legal and ethical frameworks, the promotion of unfair and opaque practices, the lack of accountability, privacy breaches, and the disruption of the labor system.

#### Applicability of current legal and ethical frameworks

The British Academy, the Royal Society, the House of the Lords, the French Parliament, and the Association for Computing Machinery are all in agreement on the following straightforward ethical principle for the use of data by algorithms: Algorithms should only use data if it is for the general benefit of society (Olhede & Wolfe, 2018). Moreover, since mid-2018, the European Union legal system follows a General Data Protection Regulation (GDPR) law that requires all algorithms to give an explanation to anyone who is requesting such an explanation when their data were used by the algorithms without their consent or when they are affected by the processing and outcome of the algorithms. However, critics doubt the ability of algorithms to generate meaningful explanations in human natural languages, let alone in the legal terminology.

#### Promotion of unfair and opaque practices

Suppose we have an algorithm on a mobile application that allows any resident to report street light problems in their neighborhood for the assessment of the state of street light in the country and for the consequent allocation of a budget for their maintenance (Crawford, 2013). Unfortunately, this collection method of massive data for the algorithm is biased in that it excludes the data of the non-users of the mobile application. Consequently, all non-users of the mobile application will not have a budget for the maintenance of their street lights even if those lights are not working, and that can be seen as unfair. Another serious concern is that the development of algorithms is so complex that they are a black box for most people who simply have to put their trust in the hands of the few algorithm specialists. Unfortunately, the betrayal of this trust is possible and can have serious negative consequences in people's lives, as in the example of the mistaken arrest of an African American due to a "faulty facial recognition match" (Hill, 2020). Search engines are another example that illustrates the opacity of algorithms. It is difficult for an end-user to understand how a search keyword can return pages that correspond to their personal profile even when such a keyword does not contain their personal data.



**Lack of accountability**

Both the Association for Computing Machinery (ACM) and the European General Data Protection Regulation (GDPR) are adamant that individuals are entitled to question algorithmic decisions. It is the initiators of the use of automated algorithms in such decision-making processes who are responsible and accountable for those decisions. This is, however, difficult to enforce mainly because the typical end-user of automated algorithms is usually a non-specialist who does not have the ability to understand and to explain in natural languages the details leading to the outcomes of algorithms.

**Privacy breaches**

The sharing of social media data (sometimes for financial motives) by different stakeholders without the prior knowledge and prior consent of the owners and generators of such data is a good example of a privacy breach in algorithmic societies. It is a major concern because people usually make use of social media to share their life experiences with their friends and families, and discuss sensitive matters on their health, love lives, or political opinions. This, unfortunately, sometimes leads to situations where managers and government authorities use social media applications to invade people's privacy with negative consequences to their work and their lives.

**Disruption of the labor system**

As stated above, the development of algorithms is so complex that they are a black box for most people who simply have to put their trust in the hands of the few specialists who are working on them. Moreover, there are perceptions that, because algorithms will ultimately take full and exclusive control of the management of people lives in terms of their decision-making choices as well as the execution of their activities, the labor force currently in charge of these decision-making choices and of these activities may eventually become redundant. The main concern is that we may end up with an algorithmic society where algorithmics' workers are the only active population and all others are unemployed.

**Summary**

This unit identified Euclid, al-Khwarizmi, Fibonacci, Gödel, Hilbert, and Turing as some of the key players in the history of algorithms within their respective historical contexts. Computational problems were introduced before the presentation of key algorithmic concepts, such as algorithm quality and the difference between algorithms and programming. Finally, we reviewed modern algorithmic societies with a few examples on some of their predominant algorithms, such as search engines and error detection, and we identified societal issues resulting from algorithms. Some of those issues include, for example, the opacity of algorithms and their lack of accountability despite the existence of international laws for the promotion of a transparent and fair algorithmic society.



## Introduction to Algorithmics

**Knowledge Check**

---

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!



# Unit 2



## Algorithm Design

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... how to organize the data of your algorithms into suitable data structures.
- ... the use of iteration and recursion in algorithm design.
- ... key algorithm design strategies.

## 2. Algorithm Design

### Introduction

When faced with the task of creating an algorithm for a computational problem, it is useful to consider existing algorithm design techniques as a source of inspiration. It is also important to have first-hand experience with other computational problems whose algorithms are based on design techniques in order to be aware of the similarities and differences between various computational problems. Similarities and differences between computational problems may arise from the nature of the data. This fact calls for a clear understanding of the different data structures that are available to algorithms, while knowing that the use of data structures may differ from one algorithm design technique to another.

The concepts that are covered by this unit are intended to give you a sound understanding of the use of data structures by key design techniques, as visible in relevant algorithms.

### 2.1 Data Structures

**Data structure**  
A data structure organizes data into a usable format to which operations can be applied.

Algorithms use several **data structures** that are presented in this section, including arrays, lists, heaps, queues, stacks, trees, and graphs.

#### Arrays

An array can be described as a finite sequence of elements which may be empty. The length of an array is simply equal to the length of its sequence of elements. These elements are accessible through their indices. It is possible to insert and remove elements in or from any position in the array. Current programming languages have many more methods and functions for arrays, as is visible in the following JavaScript example for the insertions' operations:

1. at the beginning: `arrayVariable.unshift (element(s) to add)`
2. somewhere at the middle: `arrayVariable.splice (position where the added element(s) will stay, 0, element(s) to add)`
3. at the end: `arrayVariable[arrayVariable.length] = element(s) to add;` or, `arrayVariable.push (element(s) to add)`

It is also possible to initialize an array simply by stating its number of elements even though other operations can be used later to increase or decrease that initial length. That is what is done in the JavaScript program below.

In this program, an array is created with an initial size of  $n_1$ , making it possible for the function `captureElements` to fill those  $n_1$  positions of the array simply by accessing them with their indices. However, because the other  $n_2$  elements were not planned as being part of the initialization of the array, the use of one of the methods identified above is necessary for their insertion.

### Array's Input and Output Example

```
let readline = require('readline-sync');

let captureElements = function (n,m,arr){
  let mess = 'Please enter your next element ';
  for (let i = 0; i < n; i++) {
    arr[i] = readline.question(mess);
  }
  for (let i = 0; i < m; i++){
    arr.push(readline.question(mess));
  }
}

let displayArray = function (mess, arr){
  console.log(mess);
  console.log(arr);
}

let posInt = function(s){
  let n = Number(s);
  return((s!='') && (Number.isInteger(n)) && (n>=0))
}

while(true){
  let mess1 = '\nInitial size of the array please ';
  let mess2 = 'How many additional elements please? ';
  let mess3 = 'Only positive integers please!';
  let ns = readline.question(mess1);
  let ms = readline.question(mess2);
  if ((posInt(ns)) && (posInt(ms))){
    let n1 = parseInt(ns);
    let arr1 = new Array(n1);
    let n2 = parseInt(ms);
    captureElements(n1,n2,arr1);
    displayArray('Content of the array',arr1);
  }
  else{
    console.log(mess3);
  }
}
```

## Stacks

A stack is a sequence of elements where new elements can only be inserted at the end of the sequence, and it is only the element at the end of the sequence that can be removed. Thus, stack data structure has a limited number of operations: creating an empty stack, putting a new element on top of the stack, getting the value of the element currently on top of the stack, or removing it from the stack. All of these operations can easily be implemented by a stack class with an array as a private attribute. On the other hand, implementing stacks with linked nodes is a little bit more complicated, as is visible in the following JavaScript program. Many points deserve our attention here.

## Stack Implementation with Linked Nodes (Start)

```
let readline = require('readline-sync');
class NodeClass {
  constructor (top, prevNode){
    this.topElement = top;
    this.prevTopNode = prevNode;
  }

  getTopElement(){
    return this.topElement;
  }

  putMeOnTopOf(oldTopNode){
    this.prevTopNode = oldTopNode;
  }

  removeMyTop(){
    if (this.prevTopNode !== undefined){
      this.topElement = (this.prevTopNode).topElement;
      this.prevTopNode = (this.prevTopNode).prevTopNode;
    }
    else{
      this.topElement = undefined;
      this.prevTopNode = undefined;
    }
  }

  displayMe(){
    let txt = 'Displaying the stack';
    txt = txt + '\nTop of the stack';
    let currTopEl = this.topElement;
    let currPredNode = this.prevTopNode;
    while (currTopEl !== undefined){
      txt = txt + '\n' + currTopEl;
      if (currPredNode !== undefined){
        currTopEl = currPredNode.topElement;
        currPredNode = currPredNode.prevTopNode;
      }
      else {currTopEl = undefined;}
    }
    txt = txt + '\nBottom of the stack\n';
    return(txt);
  }
}
```

## Stack Implementation with Linked Nodes (End)

```

let captureElements = function (n1){
  let oldStack = new NodeClass(undefined,undefined);
  let finalStack = new NodeClass(undefined,undefined);
  let mess = 'Please enter your next element ';
  for (let i = 0; i < n1; i++){
    let e = readline.question(mess);
    let newNode = new NodeClass(e,undefined);
    newNode.putMeOnTopOf(oldStack);
    oldStack = newNode;
    finalStack = newNode;
  }
  if (n1>0){return finalStack;}
  else {return oldStack;}
}

let posInt = function(s){
  let n = Number(s);
  return((s!='') && (Number.isInteger(n)) && (n>=0))
}

while (true){
  let mess1 = '\nPlease how many elements do you have? ';
  let mess2 = 'Only positive integers please!';
  let ns = readline.question(mess1);
  if (posInt(ns)===true){
    let n = parseInt(ns);
    let s = new NodeClass(undefined,undefined);
    s = captureElements(n);
    console.log('\nCurrent stack');
    console.log(s.displayMe());
    s.removeMyTop();
    console.log('\nOne element removed from the stack');
    console.log(s.displayMe());
  }
  else {console.log(mess2);}
}

```

## Lists

A list is a sequence of elements, just like an array or a stack. Although all three have a beginning (head) position and an end (tail) position, for a list, we will also always want to know the current position of the cursor of the list.

In fact, the operations of a list are usually carried out only from this unique position of the cursor. In the case of a stack, the position of the cursor always points to the top of the stack such that all stacks' operations can only apply to the top of the stack. The



## Algorithm Design

identified arrays' and stacks' operations still apply to lists for the position of their cursor. Moreover, lists offer the possibility to change the value of the position of their cursor.

As is the case for stacks, because lists can easily be implemented by a class with an array as a private attribute, we will show you how to implement them by using linked nodes, as illustrated in the following JavaScript program.

Always trace [this program](#) with a handful of test cases for a better understanding of the implementation of the list data structure so that missing methods can be added (e.g., insertion of elements).

## List Implementation with Linked Nodes (Start)

```
let readlineSync = require('readline-sync');

class ListNodeClass{
  constructor (e,newNode){
    this.valueInNode = e;
    this.NodeSucc = newNode;
  }

  getValueInNode(){
    return this.valueInNode;
  }

  getElementInPosition(p){
    let i = -1;
    let curr = new ListNodeClass(undefined,undefined);
    curr = this;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++;
      curr = curr.NodeSucc;
    }
    if (i===p) {return (curr.valueInNode);}
    else {return undefined;}
  }

  putMeAtEndAfter(oldList){
    let curr = new ListNodeClass(undefined,undefined);
    curr = oldList;
    while (curr.NodeSucc !== undefined){
      curr = curr.NodeSucc;
    }
    curr.NodeSucc = this;
  }

  removeElementInPosition(p){
    let i = -1;
    let curr = new ListNodeClass(undefined,undefined);
    let prev = new ListNodeClass(undefined,undefined);
    curr = this;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++;
      prev = curr;
      curr = curr.NodeSucc;
    }
    if (i===p){
      prev.NodeSucc = curr.NodeSucc;
    }
  }
}
```

## List Implementation with Linked Nodes (Cont'd)

```

displayMe(){
  let txt = 'Displaying the list';
  txt = txt + '\nBeginning of the list';
  let currSucc = this.NodeSucc;
  while (currSucc !== undefined){
    if (currSucc.valueInNode !== undefined){
      txt = txt + '\n' + currSucc.valueInNode;
    }
    currSucc = currSucc.NodeSucc;
  }
  txt = txt + '\nEnd of the list ';    return(txt);
}
}

class ListClass{
  constructor (c,l){
    this.cursor = c;
    this.list = l;
  }

  getValueOfCursor() {return this.cursor;}
  changeValueOfCursor(nc) {this.cursor = nc;}

  getElement() {
    return ((this.list).getElementInPosition(this.cursor));
  }

  removeCurrentElement(){
    (this.list).removeElementInPosition(this.cursor);
  }

  displayMe(){
    let txt = this.list.displayMe();
    txt = txt + '\nCurrent cursor value : ' + this.cursor;
    return(txt);
  }
}

let captureElements = function (n1){
  let rootNode = new ListNodeClass(undefined,undefined);
  let prevList = new ListNodeClass(undefined,undefined);
  prevList = rootNode;
  for (let i = 0; i < n1; i++){
    e = readlineSync.question('Please enter your next element ');
    let newNode = new ListNodeClass(e,undefined);
    newNode.putMeAtEndAfter(prevList);
  }
  return prevList;
}

```

### List Implementation with Linked Nodes (End)

```
let posInt = function(s){
  let n = Number(s);
  return((s!='') && (Number.isInteger(n)) && (n>=0))
}

while(true){
  let mess1 = '\nPlease how many elements do you have? ';
  let mess2 = 'Only positive integers please!';
  let ns = readlineSync.question(mess1);
  if (posInt(ns)===true){
    let n = parseInt(ns);
    let ln = new ListNodeClass(undefined,undefined);
    ln = captureElements(n);
    let ls = new ListClass(0,ln);
    console.log(ls.displayMe());
    console.log('Current element: ' + ls.getElement() + '\n\n');
    let cp=parseInt(readlineSync.question('New position of cursor
please? '));
    ls.changeValueOfCursor(cp);
    console.log(ls.displayMe());
    console.log('Current element: ' + ls.getElement() + '\n\n');
    cp=parseInt(readlineSync.question('Another position of cursor
please? '));
    ls.changeValueOfCursor(cp);
    console.log(ls.displayMe());
    console.log('Current element: ' + ls.getElement() + '\n\n');
    ls.removeCurrentElement();
    console.log('\nElement at the current position has been removed!');
    console.log(ls.displayMe());
  }
  else {console.log(mess2);}
}
```

### Non-Priority and Priority Queues

For stacks, the top of the stack contains the element that entered last. It is that top of the stack that will always be the first element to leave the stack. This is why stacks are said to have a last-in-first-out (LIFO) policy. The code below can be found [here](#).

## Non-Priority Queues Implementation with Linked Nodes (Start)

```
let readlineSync = require('readline-sync');
class QueueNodeClass {
  constructor (e,newNode){
    this.valueInNode = e;
    this.NodeSucc = newNode;
  }
  getValueInNode() {return this.valueInNode;}
  getElementInPosition(p){
    let i = -1;
    let curr = new QueueNodeClass(undefined,undefined);
    curr = this;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++; curr = curr.NodeSucc;
    }
    if (i==p) {return (curr.valueInNode);}
    else {return undefined;}
  }
  putMeAtEndAfter(oldQueue){
    let curr = new QueueNodeClass(undefined,undefined);
    curr = oldQueue;
    while (curr.NodeSucc !== undefined){
      curr = curr.NodeSucc;
    }
    curr.NodeSucc = this;
  }
  removeElementInPosition(p){
    let i = -1;
    let curr = new QueueNodeClass(undefined,undefined);
    let prev = new QueueNodeClass(undefined,undefined);
    curr = this;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++; prev = curr; curr = curr.NodeSucc;
    }
    if (i==p) {prev.NodeSucc = curr.NodeSucc;}
  }
  displayMe(){
    let s = 'Displaying the queue';
    let currSucc = this.NodeSucc;
    while (currSucc !== undefined){
      if (currSucc.valueInNode !== undefined){
        s = s + " <---- " + currSucc.valueInNode;
      }
      currSucc = currSucc.NodeSucc;
    }
    return(s);
  }
}
```



## Non-Priority Queues Implementation with Linked Nodes (End)

```

class NpQueueClass{
  constructor (npq){
    this.queue = npq;
  }
  getElement(){
    return ((this.queue).getElementInPosition(0));
  }
  removeCurrentElement(){
    (this.queue).removeElementInPosition(0);
  }
  displayMe(){
    return(this.queue.displayMe());
  }
}
let captureElements = function (n1){
  let rootNode = new QueueNodeClass(undefined,undefined);
  let prevQueue = new QueueNodeClass(undefined,undefined);
  prevQueue = rootNode;
  let mess = 'Please enter your next element ';
  for (let i = 0; i < n1; i++){
    let e = readlineSync.question(mess);
    let newNode = new QueueNodeClass(e,undefined);
    newNode.putMeAtEndAfter(prevQueue);
  }
  return prevQueue;
}
let posInt = function(s){
  let n = Number(s);
  return((s!=='') && (Number.isInteger(n)) && (n>=0))
}

while(true){
  let mess1 = '\nPlease how many elements do you have? ';
  let mess2 = 'Only positive integers please!';
  let ns = readlineSync.question(mess1);
  if (posInt(ns)===true){
    let n = parseInt(ns);
    let npqn = new QueueNodeClass(undefined,undefined);
    npqn = captureElements(n);
    let npq = new NpQueueClass(npqn);
    console.log(npq.displayMe());
    console.log('\nLet us remove first element of the queue');
    npq.removeCurrentElement(); console.log(npq.displayMe());
    console.log('\nLet us remove one more element of the queue');
    npq.removeCurrentElement(); console.log(npq.displayMe());
  } else {console.log(mess2);}
}

```

## Algorithm Design

In the first-in-first-out (FIFO) policy of non-priority queues, the first element to have entered the queue is always the first element to leave. Non-priority queues behave like lists with cursor positions always equal to zero, making their implementation very similar to the one of lists except for the position. A non-priority queue that is implemented as a list with a cursor position always equal to zero is visible in the above JavaScript program. Readers are advised that the word list has simply been replaced by the word queue. The cursor attribute has disappeared, and it has been replaced by the zero value where necessary.

As in the case of a queue where the elderly are sometimes helped first, priority queues allow their elements to have different levels of priority in order to first serve the ones with the highest level of priority. This requires the `QueueNodeClass` class to have a third attribute on the priority level of each of its nodes so that, periodically, we can scan through the queue, identify the position of the first node with the highest level of priority, and remove it from the queue in order to attend to it. This is done with the `getPriorityPosition()` method of the `PQueueNodeClass` class in the [JavaScript implementation](#) of a priority queue, shown below.

The first difference between the `PQueueNodeClass` class and the `QueueNodeClass` class is the inclusion of the priority attribute in the `PQueueNodeClass` class to indicate that each node of a priority queue has a priority level that is represented by a priority number. Recall that a priority number is a strictly positive natural number and that the smaller the priority number, the higher the priority level.

## Priority Queues Implementation with Linked Nodes (Start)

```

let readlineSync = require('readline-sync');

class PQueueNodeClass {
  constructor (e,pr,newNode){
    this.valueInNode = e;
    this.priority = pr;
    this.NodeSucc = newNode;
  }
  getValueInNode() {return this.valueInNode;}
  getPriority() {return this.priority;}
  getPriorityPosition(){
    let curr = new PQueueNodeClass(undefined,undefined,undefined);
    curr = this;
    if (curr.NodeSucc === undefined){return -1;} else {
      curr = curr.NodeSucc; let hprl = curr.priority;
      let i = 0; let prp = i;
      while (curr.NodeSucc !== undefined){
        i++; curr = curr.NodeSucc;
        if (curr.priority<hprl){hprl=curr.priority; prp=i;}
      } return prp;
    }
  }
  getElementInPosition(p){
    let curr = new PQueueNodeClass(undefined,undefined,undefined);
    curr = this; let i = -1;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++; curr = curr.NodeSucc;
    }
    if (i===p){return(curr.valueInNode);} else {return undefined;}
  }
  getPriorityInPosition(p){
    let curr = new PQueueNodeClass(undefined,undefined,undefined);
    curr = this; let i = -1;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++; curr = curr.NodeSucc;
    }
    if (i===p){return (curr.priority);} else {return undefined;}
  }
  putMeAtEndAfter(oldQueue){
    let curr = new PQueueNodeClass(undefined,undefined,undefined);
    curr = oldQueue;
    while (curr.NodeSucc !== undefined) {curr = curr.NodeSucc;}
    curr.NodeSucc = this;
  }
}

```



## Priority Queues Implementation with Linked Nodes (Cont'd)

```

removeElementInPosition(p){
  if (p !== -1){
    let curr=new PQueueNodeClass(undefined,undefined,undefined);
    let prev=new PQueueNodeClass(undefined,undefined,undefined);
    curr = this; let i = -1;
    while ((i<p) && (curr.NodeSucc !== undefined)){
      i++; prev = curr; curr = curr.NodeSucc;
    }
    if (i===p) {prev.NodeSucc = curr.NodeSucc;}
  }
}
displayMe(){
  let s = 'Displaying the queue';
  let currSucc = this.NodeSucc;
  while (currSucc !== undefined){
    if (currSucc.valueInNode !== undefined){
      s=s+'<----('+currSucc.getValueInNode()+':'+currSucc.priority+')';
    }
    currSucc = currSucc.NodeSucc;
  }
  return(s);
}
}
class PQueueClass {
  constructor (pqp) {
    this.queue = pqp;
  }
  getElement() {return ((this.queue).getElementInPosition(0));}
  removeCurrentElement(){(this.queue).removeElementInPosition(0);}
  getHighestPriorityElement(){
    let hprp = (this.queue).getPriorityPosition();
    return ((this.queue).getElementInPosition(hprp));
  }
  getPriorityLevelOFHighestPriorityElement(){
    let hprp = (this.queue).getPriorityPosition();
    return ((this.queue).getPriorityInPosition(hprp));
  }
  removeHighestPriorityElement(){
    let hprp = (this.queue).getPriorityPosition();
    (this.queue).removeElementInPosition(hprp);
  }
  displayMe() {return(this.queue.displayMe());}
}

```

### Priority Queues Implementation with Linked Nodes (End)

```

let captureElements = function (n1) {
  let rootNode=new PQueueNodeClass(undefined,undefined,undefined);
  let prevQueue=new PQueueNodeClass(undefined,undefined,undefined);
  prevQueue=rootNode; let mess = 'Please enter your next element ';
  for (let i = 0; i < n1; i++){
    let prtl = Math.floor((Math.random() * 100) + 1);
    let e = readlineSync.question(mess);
    let newNode = new PQueueNodeClass(e,prtl,undefined);
    newNode.putMeAtEndAfter(prevQueue);
  } return prevQueue;
}
let posInt = function(s){
  let n = Number(s);
  return((s!=='') && (Number.isInteger(n)) && (n>=0))
}

while(true){
  let mess1 = '\nPlease how many elements do you have? ';
  let mess2 = 'Only positive integers please!';
  let ns = readlineSync.question(mess1);
  if (posInt(ns)===true){
    let n = parseInt(ns);
    let pqn = new PQueueNodeClass(undefined,undefined,undefined);
    pqn = captureElements(n); let pq = new PQueueClass(pqn);
    console.log(pq.displayMe());
    console.log('\nLet us remove first element of the queue');
    pq.removeCurrentElement(); console.log(pq.displayMe());
    let pl = pq.getPriorityLevelOFHighestPriorityElement();
    console.log('\nHighest priority element');
    console.log(pq.getHighestPriorityElement()+ ' Priority no '+ pl);
    console.log('\nLet us remove highest priority element');
    pq.removeHighestPriorityElement();console.log(pq.displayMe());
    console.log('\nLet us remove first element of the queue');
    pq.removeCurrentElement(); console.log(pq.displayMe());
  } else {console.log(mess2);}
}

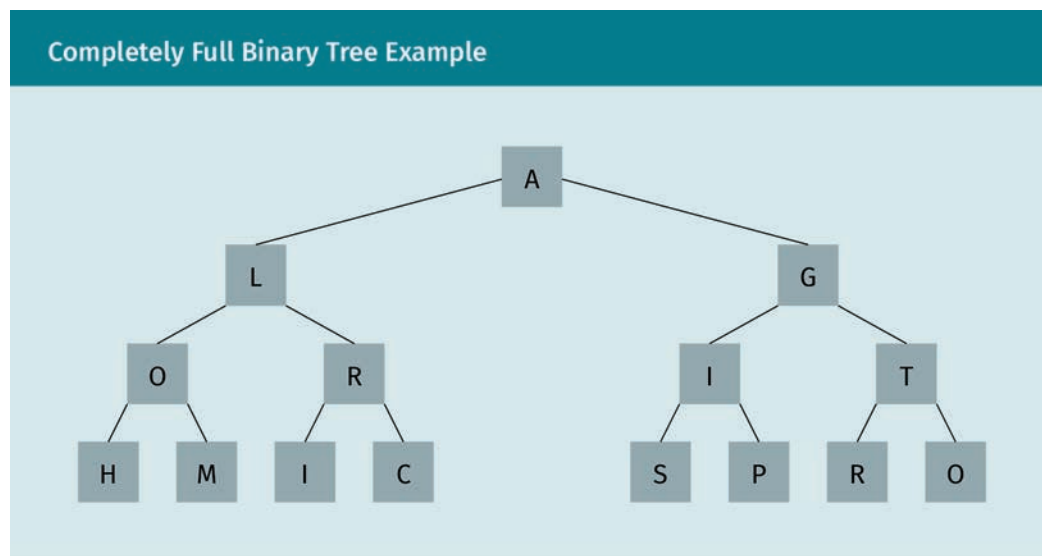
```

The second difference between these two classes is the inclusion of the `getPriorityPosition()` method in the `PQueueNodeClass` class so that we can use that method to find the position of the first element with the highest priority and remove the associated element from the queue, if needed, in order to attend to it.

This is precisely what is achieved by the `removeHighestPriorityElement()` method of the `PQueueClass` class. The use of the `Math.random()` method by the `captureElements()` function for the generation of random priority numbers is also worth noting.

## Binary Trees and Binary Search Trees

A tree is made of up a root node that has one or many children or subtrees. In the case of binary trees, every root node has one or two direct sub-nodes except for the leaves that do not have any nodes. Let us suppose that we have a binary tree in which every node has two sub-nodes, except for the leaves that do not, as represented in the following diagram.



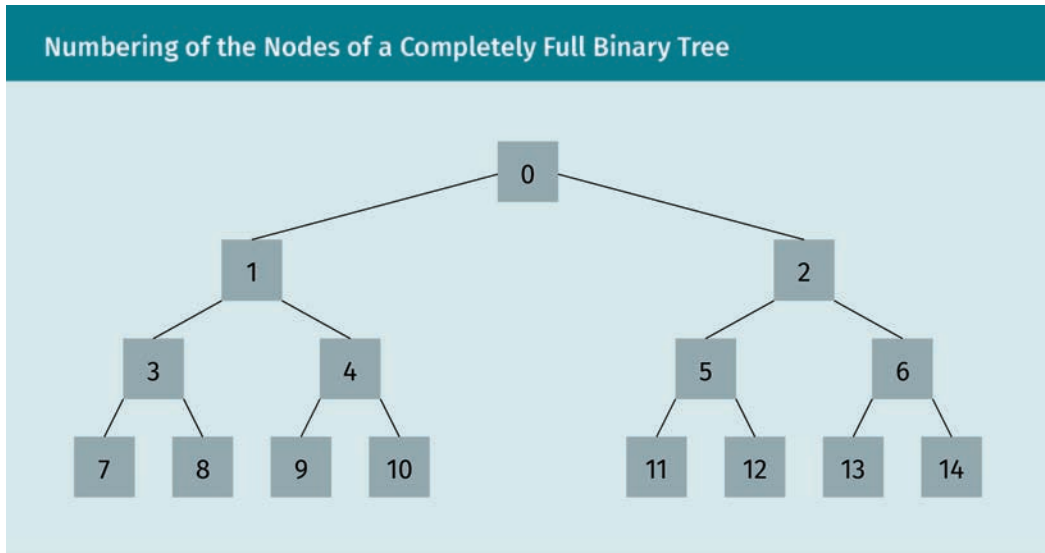
Such a completely full  $L$  levels tree has a number of nodes equal to the sum of the terms of the geometrical sequence that has 1 as its first term and 2 as its ratio.

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \dots + 2^{L-1}$$

The application of the following formula for the sum of the first  $n$  terms of a geometric series with an initial value  $a$  and with a ratio  $r$  yields a value equal to  $2^{L-1}$ .

$$ar^0 + ar^1 + ar^2 + ar^3 + ar^4 + \dots + ar^{n-1} = \frac{a(1-r^n)}{(1-r)}$$

This means that if we want to store all nodes of a completely full binary tree of  $L$  levels, we must keep  $2^L - 1$  slots for that storage. Such slots can be numbered from 1 to  $2^L - 1$  or from 0 to  $2^L - 2$ . As the figure below illustrates: the numbers represent the positions of the tree above, and the root of the tree is in position 0.



It is interesting to note in this tree of positions that all the left nodes are located in odd positions and all the right nodes are located in even positions. Moreover, the position of a left child node is the addition of one to the double of the position of its father, and the position of a right child is the addition of two to the double of the position of the father. Conversely, the position of a father is the half of the position of its left child minus one, but that position of the father is also the subtraction of one from half of the position of the right child. In other words, the position of the two children can easily be calculated based on the position of the parent, and the position of the parent can also be easily calculated once we know the position of one of the children. Thus, it is possible to assign values to the different nodes of the binary tree based on the following rule: assigning a value to the root of the tree that is located in position 0 is always allowed, and, for all the other nodes, assigning a value to a node in a position where the parent does not have a value is not allowed. This is precisely what is done by the following JavaScript program where the values of the nodes of binary tree are kept in an array.

The following points can be singled out from this JavaScript implementation of binary trees:

- The `BinTreeClass` class that represents binary trees is quite simple as it is only made up of two attributes: (1) the number of levels in the tree and (2) the array that will store all the elements of the array. It has a `getTTNbOfElements()` method whose purpose is to calculate the total number of slots required for the array to store all the elements of the binary tree, even when such a tree is completely full. That method is called by the `initializeBinTree()` that assigns all these array's slots to `undefined` as a way of stating that none of the elements of the tree has so far been given a value. The role of the `putElementInNode()` method is to place an element in a given position on the binary tree provided that its parent's position

## Algorithm Design

does not contain an **undefined** object. Other instructions are included in the JavaScript program, and their *modus operandi* will easily become apparent by tracing the program with a few representative test cases.

- It is worth noting that the JavaScript program presented below does not have a method on how to remove an element from a binary tree. Suppose we want to remove the element that is currently located in a given position on the binary tree. The first thing to do is to set the value in that position to the **undefined** object. Afterwards, one will have to scan through all the nodes that have a higher index value than the current position and that are located below the node of that position. All those higher index values will be set to **undefined**.

Just as there are many species of trees in nature, there are different sizes and shapes of binary trees in algorithmics. This variation has led to the identification of several types of binary trees such as full, complete, perfect, and balanced trees. Other noticeable types of binary trees include the Huffman and binary search trees, heaps, and AVLs (Adel'son-Vel'sky & Landis, 1962), in addition to other types of multi-node trees where a node is allowed to have more than two children. Similarly, there are different ways to systematically scan through the different nodes of a tree: the breadth-first approach, where the nodes that are the nearest to the root are the first ones to be scanned, and the depth-first approach, where the nodes that are the furthest from the root are the first ones to be scanned. The implementation of binary trees can be done either with the use of an array or linked nodes, as was done for lists and queues. The following [JavaScript program](#) uses arrays for its implementation of binary trees.



## Binary Trees Implementation with Arrays (Start)

```

let readlineSync = require('readline-sync');

class BinTreeClass {
  constructor (lev) {
    this.levels = lev;
    this.slots = [];
  }
  getTTNbOfElements(){
    let pw = Math.pow(2,this.levels);
    let space = (Math.floor(pw))-1;
    return space;
  }
  initializeBinTree(){
    let n = this.getTTNbOfElements();
    for(let i=1; i<=n; i++){this.slots.push(undefined);}
  }
  getElementInNode(p) {
    if((p>=0)&&(p<(this.slots).length)){return this.slots[p];}
    else {return undefined;}
  }
  putElementInNode(e,cp){
    if (cp===0){this.slots[cp] = e;}
    else {
      if ((cp>0) && (cp<(this.slots).length)){
        let parPos = 0;
        if ((cp%2)===0){parPos=(Math.floor(cp/2))-1;}
        else {parPos = Math.floor((cp-1)/2);}
        if (this.getElementInNode(parPos)!==undefined){
          this.slots[cp] = e;
        }
      }
    }
  }
  displayMe(){
    let s = 'Displaying the tree\n';
    for (let i = 0; i<(this.slots).length; i++) {
      if (((this.slots)[i]) !== undefined){
        s = s + '| |' + i + ': ' + ((this.slots)[i]);
      }
    }
    return(s + '| |');
  }
}

```

## Binary Trees Implementation with Arrays (End)

```

let captureElements = function (bt,n1){
  let m = bt.getTTNbOfElements()-1;
  let mess1 = 'Node position between 0 and ' + m + ' please ';
  let mess2 = 'Node element please? ';
  for (let i = 0; i < n1; i++){
    let myp = parseInt(readlineSync.question(mess1));
    let e = readlineSync.question(mess2);
    bt.putElementInNode(e,myp);
  }
}

let posInt = function(s){
  let n = Number(s);
  return((s!='') && (Number.isInteger(n)) && (n>=0))
}

while(true){
  let ms0='\nRoot of the binary tree: Position 0';
  let ms1='Left child of the root: Position 1';
  let ms2='Right child of the root: Position 2';
  let ms3='Left child of the left child of the root: Position 3';
  let ms4='And so on';
  console.log(ms0);
  console.log(ms1);
  console.log(ms2);
  console.log(ms3);
  console.log(ms4);
  let mess1 = '\nPlease how many levels do you have? ';
  let mess3 = 'Only positive integers please!';
  let ns = readlineSync.question(mess1);
  if (posInt(ns)===true){
    let n = parseInt(ns);
    let btn = new BinTreeClass(n);
    let nm = btn.getTTNbOfElements();
    let mess2='Number of non empty nodes between 1 and '+nm+' ';
    let ms = readlineSync.question(mess2);
    if (posInt(ms)===true){
      let m = parseInt(ms);
      if ((m>0)&&(m<=nm)){
        btn.initializeBinTree();
        captureElements(btn,m);
        console.log(btn.displayMe());
      } else {console.log('Too few or too many nodes!')}
    } else {console.log(mess3);}
  } else {console.log(mess3);}
}
}

```

Let us briefly return to the concepts of full trees and complete trees. In a full tree, only the leaves are allowed not to be full. For a binary tree to be complete, it must satisfy the following two conditions: (a) be full, and (b) have all leaves at the same height from

the root. It is also possible for a binary tree to be considered semi-complete, i.e., it is complete up to one level before the leaves, but few of the parents of the leaves on the right side are not full. One of the advantages of complete and almost-complete binary trees is that, because they do not have many internal holes, their implementation by an array effectively makes use of almost all the spaces of that array. As for binary search trees, the value of the left child of each node must be smaller than the one of the node itself, which in return must be smaller than the value of its right child. This makes it easy for these trees to be searched.

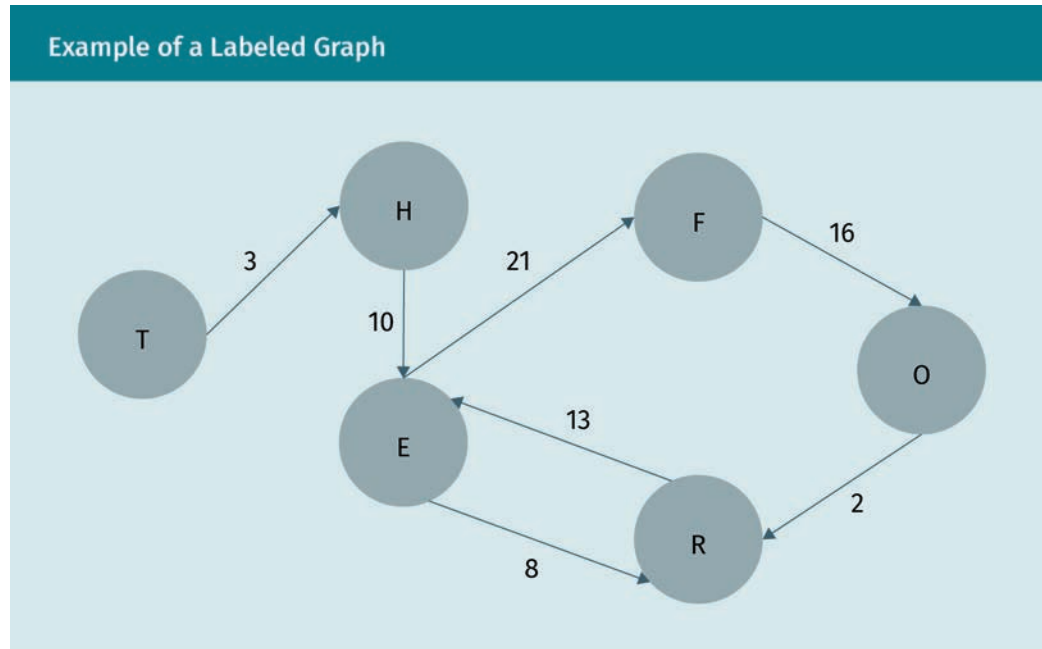
A heap is an almost complete binary tree that also keeps an order between each node and its children. There are two types of heaps: the min-heap and the max-heap. In a max-heap, the value of each parent node is greater than or equal to the one of each of its direct children nodes. For a min-heap, however, the value of each parent node is smaller than or equal to the one of its direct children nodes. The root of a max-heap always holds the maximum value of the tree, and the root of the min-heap also holds the minimum value of the tree. This is what makes heaps suitable for the representation of priority queues.

## Graphs

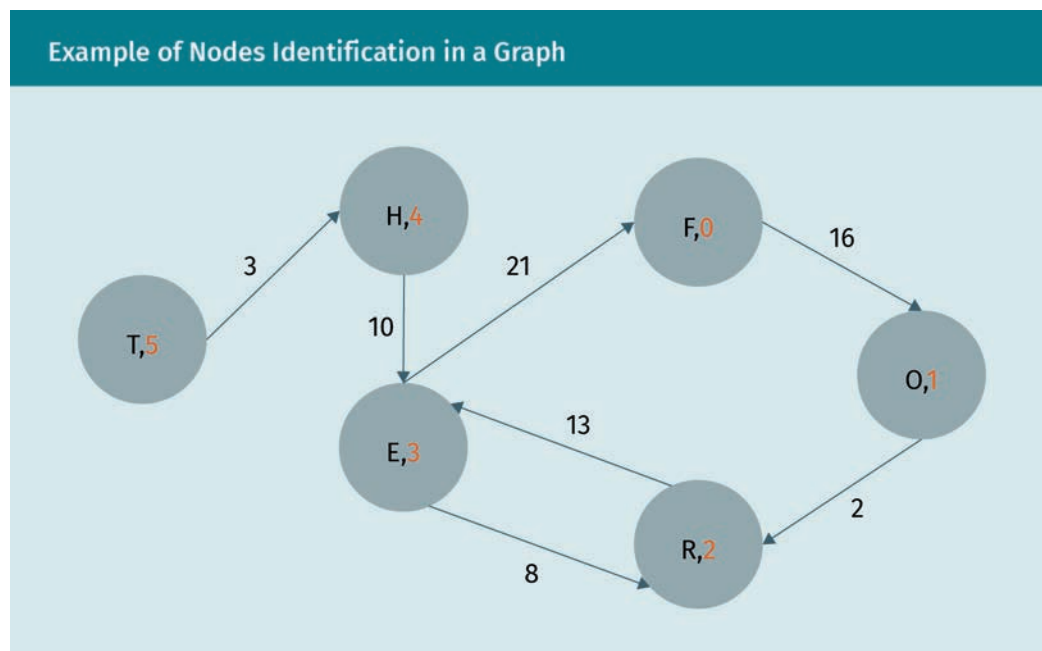
Much like binary trees, graphs can either be scanned with a breadth-first approach or a depth-first approach. Their application domains include communication systems, hydraulic systems, integrated computer circuits, mechanical systems, and transportation (Ahuja et al., 1993).

A graph can be seen as a set of connected nodes. In a directed graph, the direction of the connection between two nodes is well indicated as opposed to general graphs where all connections between nodes are bi-directional. Thus, we focus on directed graphs because they also allow for the possibility of representing bi-directions in nodes' connections, as illustrated in the example below. This example is a labeled graph where weights are assigned to the nodes' connections. In any case, this also accommodates unlabeled graphs where weights are Boolean values.

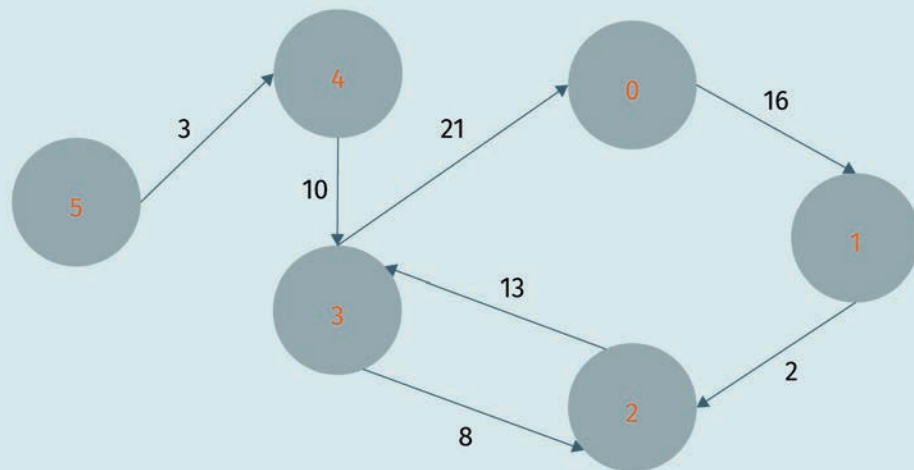




In a graph with a given number of nodes, each node can be identified by a unique natural positive number less than the total number of nodes. Thereafter, it will become easy to identify each node and implement graphs either as two-dimensional arrays or as an array of linked nodes. This is what is done below for the above graph.



### Representation of the Nodes of a Graph in an Array



0	1	2	3	4	5
F	O	R	E	H	T

Let's now represent the previous graph example first as a two-dimensional array and as an array of linked nodes.

### Graph's Representation as a Two-Dimensional Array and an Array of Linked Nodes

	0	1	2	3	4	5
0		16				
1			2			
2				13		
3	21		8			
4				10		
5					3	

0	→	(1;16)
1	→	(2;2)
2	→	(3;13)
3	→	(0;21) → (2;8)
4	→	(3;10)
5	→	(4;3)

## Algorithm Design

Certain graphs' configurations are more suitable for representation as a two-dimensional array while others are better represented as an array of linked nodes. The [JavaScript example](#) below is an implementation of graphs with two-dimensional arrays so that readers can also be introduced to the programming of these types of arrays in JavaScript.

In this JavaScript program, the `nbOfNodes` attribute of the `GraphClass` class represents the total number of nodes of the graph, and the names of those nodes are stored in the `nodesNames` array attribute. It is the `weights` array attribute that stores the values of the weights of the links between the different nodes of the graph, as illustrated in the two-dimensional array (above). Unfortunately, there are no two-dimensional arrays in JavaScript. Hence, we must store arrays inside another array as a way to implement a two-dimensional array. This is visible in the `initializeGraph()` method whose purpose is to initialize each box of the `nodesNames` array with the `undefined` object. That method also initializes each box of the `weights` array with a sequence of `undefined` objects.

Suppose we have a graph with three nodes. In this case, the `nodesNames` array is made up of three boxes and each of these three boxes is initialized with the `undefined` object. Similarly, the `weights` array is made up of three boxes. Each of these three boxes is also an array of three boxes that are each initialized with the `undefined` object by the `initializeGraph()` method. The rest of the program is not too difficult to understand, and, again, it is always useful and recommended to trace your programs with a few test cases in order to unearth all their details.

## Graph Implementation with Two Arrays (Start)

```

let readlineSync = require('readline-sync');

class GraphClass {
  constructor (nbn) {
    this.nbOfNodes = nbn;
    this.nodesNames = [];
    this.weights = [];
  }
  getNbOfNodes(){return this.nbOfNodes;}
  initializeGraph(){
    let n = this.getNbOfNodes();
    for (let i=0; i<n; i++){
      this.nodesNames.push(undefined);
      let w = [];
      for (let j=0; j<n; j++){w.push(undefined);}
      this.weights.push(w);
    }
  }
  getElementInNode(p){
    if ((p>=0) && (p<(this.nodesNames).length)){
      return this.nodesNames[p];
    } else {return undefined;}
  }
  putElementInNode(e,p){
    if ((p>=0) && (p<(this.nodesNames).length)){
      this.nodesNames[p] = e;
    }
  }
  putWeightBtwNodes(w,o,d){
    let n = this.getNbOfNodes();
    let c1 = ((o>=0) && (o<n));
    let c2 = ((d>=0) && (d<n));
    if (c1 && c2){this.weights[o][d] = w;}
  }
  displayNames(){
    let s = 'Displaying the names\n';
    let n = this.getNbOfNodes();
    for (let i=0; i<n; i++){
      if (((this.nodesNames)[i]) !== undefined) {
        s = s + i + ' : ' + ((this.nodesNames)[i]) + '\n';
      }
    }
    return(s.slice(0,-1));
  }
}

```

## Graph Implementation with Two Arrays (End)

```

displayWeights(){
  let s = 'Displaying the weights\n';
  let n = this.getNbOfNodes();
  for (let i=0; i<n; i++){
    //s = s + "\n";
    for (let j=0; j<n; j++){
      if (((this.weights)[i][j]) !== undefined){
        s=s+'('+i+'->'+j+' : '+((this.weights)[i][j])+')';
      }
    }
  }
  return(s.slice(0,-1));
}
}
let captureNames = function (g){
  let mess1 = 'Node position please? ';
  let n1 = g.getNbOfNodes();
  let mess = 'Name of node number ';
  for (let i = 0; i < n1; i++) {
    let e = readlineSync.question(mess + i + ' please? ');
    g.putElementInNode(e,i);
  }
}
let captureWeights= function (m,g){
  let mess1 = ' origin position? ';
  let mess2 = ' destined position ';
  let mess3 = 'Weight between the two nodes please? ';
  for (let i = 1; i <= m; i++) {
    orp = parseInt(readlineSync.question("Weight " + i + mess1));
    dsp = parseInt(readlineSync.question("Weight " + i + mess2));
    let wgt = readlineSync.question(mess3);
    g.putWeightBtwNodes(wgt,orp,dsp);
  }
}
while(true){
  let ms1 = '\nPlease how many nodes do you have? ';
  let ms2 = 'How many weights please? ';
  let n = parseInt(readlineSync.question(ms1));
  let grp = new GraphClass(n);
  grp.initializeGraph();
  captureNames(grp);
  console.log(grp.displayNames());
  let m = parseInt(readlineSync.question(ms2));
  captureWeights(m,grp);
  console.log(grp.displayWeights());
}

```

## 2.2 Recursion and Iteration

Let's briefly put ourselves in the situation where we have to fulfill the important mission of climbing a cement stairway. One way to approach that mission is to step on the first stair, then move to the second, then the third, the fourth, the fifth, and so on, up to the top. We would call this an iterative approach. It usually takes the form of a **for**, a **while**, or a **repeat** loop when it is adopted in an algorithm. Each of its steps is identified as an iteration. The other way to approach the mission is to step on the first stair and simply consider the remaining steps as a new but shorter mission that, when fulfilled, will also successfully end the initial mission. This new approach is **recursive** in the sense that the completion of the initial mission on a given object relies on the completion of the same mission but for a simpler or smaller object. Let us now illustrate these two concepts of recursion and iteration with the following two JavaScript programs on the factorial of numbers and on the enumeration of prime numbers, respectively. More precisely, the [first program](#) calculates the factorial of a given strictly positive natural number, both iteratively and recursively. The [second program](#) enumerates the set of prime numbers smaller than or equal to a given strictly positive natural number, also both iteratively and recursively.

### Recursion

This approach allows a function to call itself with different arguments.

This is a gentle reminder of the formula of the factorial of a strictly positive natural number  $n$ .

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdot (n - 4) \cdot \dots \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

### Iterative and Recursive Versions of Factorial

```

let readline = require('readline-sync');
let factIter = function(n1) {
  if (n1>=0) {
    let r = 1;
    for (let i=1; i<=n1; i++) {r = r * i;}
    return r;
  }
}
let factRec = function (n1){
  if (n1>=0) {
    if ((n1===0) || (n1===1)) {return 1;}
    else {return (n1 * factRec(n1-1));}
  }
}
while (true){
  let mess='\nPositive integer please ';
  let s = readline.question(mess);
  let n = Number(s);
  if (((s!=='') && (Number.isInteger(n)) && (n>=0)) === true)
  {
    n=parseInt(s);
    let i=factIter(n);
    let r=factRec(n);
    let ms='Factorial of '+n+' iterative is '+i+'; recursive is '+r;
    console.log(ms);
  }
}

```

The iterative calculation of the factorial of  $n$  can be illustrated as follows.

### Iterative Illustration of Factorial

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot \dots \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$



As for the recursive calculation of the factorial of  $n$ , it is clearly visible in the formula of factorial itself, as illustrated below.



$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdot (n - 4) \cdot \dots \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$n! = n \cdot (n - 1)! \text{ i.e. } \text{Factorial}(n) = n \cdot \text{Factorial}(n - 1)$$

In other words, the recursive calculation of the factorial of  $n$  involves the calculation of another factorial, the factorial of  $n-1$ .

Both the iterative and recursive approaches are illustrated in the JavaScript program presented below. It enumerates the set of prime numbers less than or equal to a given strictly positive natural number, starting with the primality testing function for which `isPrimeIter()` uses the iterative approach, and `isPrimeRec()` uses the recursive approach. The `isPrimeIter()` function loops from 2 to the value of its parameter to check if the later one has a divisor. As for `isPrimeRec()`, if its first parameter is divisible by its second, then it concludes on the non-primality of the first parameter; otherwise, it reduces the value of its second parameter by one before calling itself back into action. The `primesIter()` function uses an iterative approach by testing the primality of each number for its possible inclusion in the final array. On the other hand, the `primesRec()` function checks the primality of its parameter and decides its inclusion in the set of the prime numbers that are less than that parameter.



## Naive Iterative and Recursive Primality Test Algorithms

```

let readline = require('readline-sync');
let isPrimeIter = function (n1) {
  if (n1 >= 2) {
    let r = true; let i = 2;
    while ((r === true) && (i < n1)) { r = ((n1 % i) !== 0); i++; }
    return r;
  } else { return false; }
}
let isPrimeRec = function (n1, d) {
  if (n1 >= 2) {
    if (d === 1) { return true; }
    else {
      if ((n1 % d) === 0) { return false; }
      else { return isPrimeRec(n1, d-1); }
    }
  } else { return false; }
}
let primesIter = function (n1) {
  if (n1 >= 2) {
    let r = [];
    for (let i=2; i <= n1; i++) {
      if (isPrimeIter(i) === true) { r.push(i); }
    }
    return r;
  } else { return ([]); }
}
let primesRec = function (n1) {
  if (n1 >= 2) {
    if (n1 === 2) { let rn1 = []; rn1.push(n1); return rn1; }
    else {
      let rpredn1 = []; rpredn1 = primesRec(n1-1);
      if (isPrimeRec(n1, n1-1) === true) { rpredn1.push(n1); }
      return rpredn1;
    }
  } else { return ([]); }
}
while (true) {
  let mess = '\nAn integer greater than 1 please ';
  let s = readline.question(mess); let n = Number(s);
  if (((s !== '') && (Number.isInteger(n)) && (n >= 2)) === true) {
    n = parseInt(s); let fi = []; let fr = [];
    fi = primesIter(n); fr = primesRec(n);
    console.log('Prime numbers less or equal to ' + n);
    console.log('Iterative method: '); console.log(fi);
    console.log('Recursive method: '); console.log(fr);
  }
}

```

## 2.3 Divide-and-Conquer

Divide-and-Conquer  
This strategy results  
in faster algorithms  
by breaking the  
input data into  
many, almost equal-  
sized inputs.

The **divide-and-conquer** algorithm design strategy consists of sub-dividing a given problem into similar sub-problems of almost equal sizes, solving those sub-problems, and aggregating the solutions of those sub-problems into an overall solution for the initial problem. According to Kao (n.d.), the divide-and-conquer strategy is linked to Herbert Simon's mathematical concept of near decomposability. Simon (2002) defines near decomposability as the "boxes-within-boxes" hierarchical multilevel organization of a system.

This algorithm design strategy is usually recursive and time-efficient. This strategy is illustrated in the following algorithm that multiplies two long integers with an equal length that is a power of two. Suppose that we have the long integer  $L1 = 74983152$  whose length is 8. We can divide this long integer into two parts: the upper part 7498, and the lower part 3152. Similarly, let's consider the example of another long integer  $L2 = 54926813$  of length 8 whose upper half is 5492 and lower half is 6813. This is where the trick is.

$$\begin{aligned} L1 &= 74983152 = (7498 \cdot 10^4) + 3152 \\ L2 &= 54926813 = (5492 \cdot 10^4) + 6813 \\ L1 \cdot L2 &= (6813 \cdot 3152) + ((6813 \cdot 7498) \cdot 10^4) + ((5492 \cdot 3152) \cdot 10^4) + ((5492 \cdot 7498) \cdot 10^8) \end{aligned}$$

This equation shows how the problem of multiplying integers of length 8 has been reduced to the problem of multiplying integers of length 4 which itself will be reduced to the easier problem of multiplying integers of length 2. The efficiency of this method comes from the number of steps taken to move from 8 to 2 (three steps on the 8, 4, 2 trip) compared to the 8 steps on the 8, 7, 6, 5, 4, 3, 2, 1 trip.

## 2.4 Balancing, Greedy Algorithms, and Dynamic Programming

The following algorithm design techniques are usually used for optimization problems where the aim is typically to find the best option for a given purpose.

### Balancing

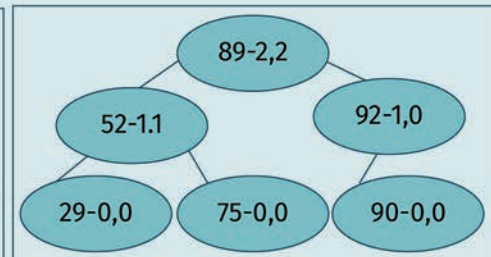
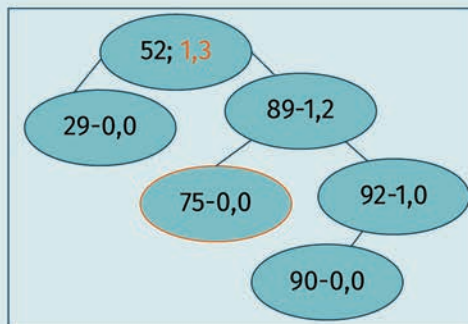
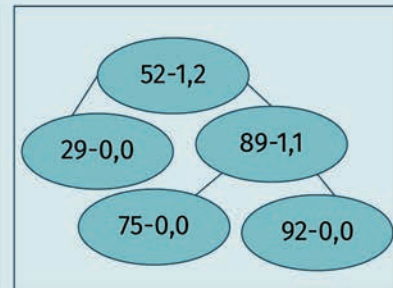
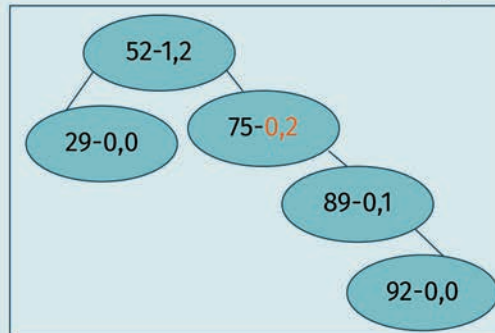
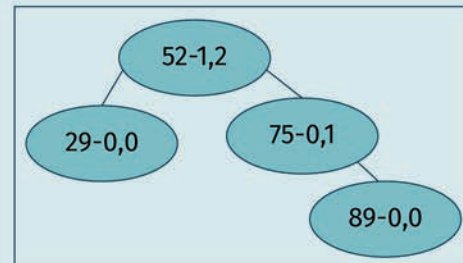
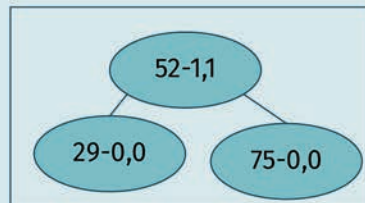
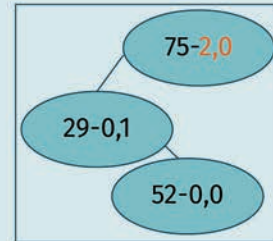
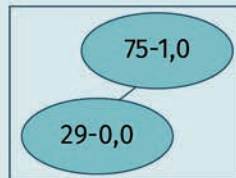
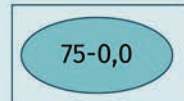
The divide-and-conquer algorithm design technique teaches us that an algorithm can become significantly faster by subdividing its input into two smaller equal-sized sub-inputs. This is the case, for example, for an algorithm on binary trees where it can decide to consider the left and the right children of the tree separately in its quest to solve its computational problem. Hence, ensuring that the left and the right children of the tree have an almost equal size for the metric that the algorithm is interested in is vital. It is in that perspective that AVL trees always stay balanced (heights of left and

## Algorithm Design

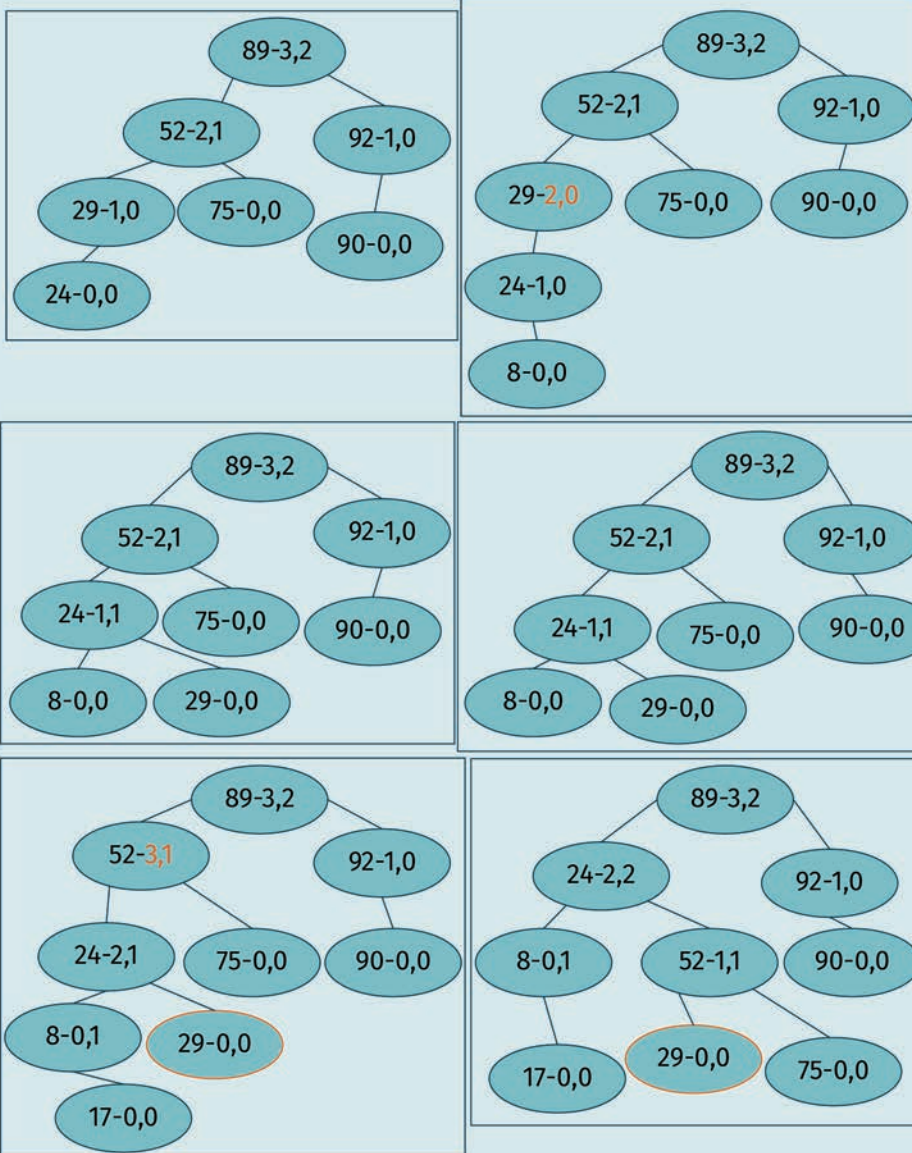
right children never differ by more than one). The AVL acronym comes from the names of Adel'son-Vel'sky and Landis (1962) who are credited for the discovery of the AVL data structure.

Let us now illustrate the concept of tree-balancing by successively inserting the following numbers in an initially empty AVL, having in mind the two main properties of AVLs: (1) the value of any parent is greater than the one of any left child but less than the one of any right child, and (2) the difference in height between any left child and any right child cannot exceed one. The sequence of numbers to be inserted is 75, 29, 52, 89, 92, 90, 24, 8, 17, 27.

Insertion of the Sequence 75, 29, 52, 89, 92, 90, 24, 8, 17, 27 in an AVL (Start)

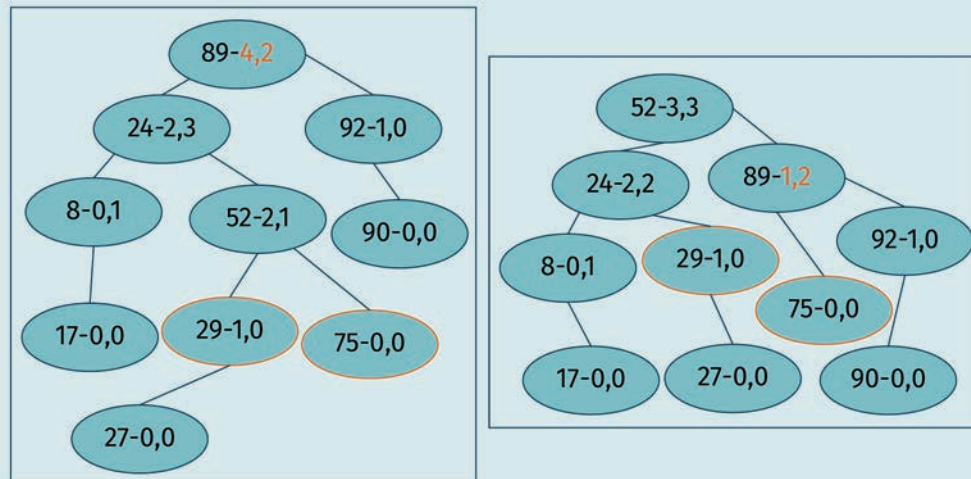


Insertion of the Sequence 75, 29, 52, 89, 92, 90, 24, 8, 17, 27 in an AVL (Cont'd)





### Insertion of the Sequence 75, 29, 52, 89, 92, 90, 24, 8, 17, 27 in an AVL (End)



### Greedy Algorithms

Dating back to the 1970s, the greedy algorithm design technique consists of always making the best possible choice in the moment (local choice), even if the final or global outcome from these successive local choices happens not to be the best one (Ye, 2013). This is how this technique works for the knapsack problem. Suppose that we have to store a set of valuable fruits and vegetables in a bag with a given weight so that the bag can be as full as possible with the most valuable items. We have mangoes, avocados, yams, maize, cassava, potatoes, and apples. The different weights of these different items are 20, 15, 10, 30, 5, 50, and 25, and their total costs are 60, 60, 90, 60, 80, 75, and 75, respectively. If we want to know the value of a given item, we must calculate the unit cost of that item by dividing its total cost by its weight.

The greedy approach starts by assuming that mangoes are the most valuable and compares their unit cost to the other fruits and vegetables until it finds that avocados are more valuable than mangoes. Once avocados are assumed to be the most valuable, their unit cost will be compared against that of yams, which will become the assumed most valuable item. This process continues until we learn that cassava is actually the most valuable.

### Greedy Algorithm Example

Items	Mangoes	Avocadoes	Yams	Maize	Cassava	Potatoes	Apples
Weight	20	15	10	30	5	50	25
Total Costs	60	60	90	60	80	75	75
Unit Cost	3	4	9	2	16	1.5	3

### Dynamic Programming

This algorithm design technique looks for the best possible solution to a combinatory problem by dividing the initial problem into relevant sub-problems whose solutions are then stored in arrays before being aggregated toward the final solution. Currency exchange illustrates this technique. Suppose we have 1, 7, and 11 coins denominations for a given currency, and we want to change a cash value of 21 in that currency with the smallest possible number of coins. A greedy approach will first look at the eleven (11), then the seven (7), and finally the one (1). It will result in a single eleven (11) currency denomination coin, a single seven (7) currency denomination coin, and 3 one (1) coins, for a total of 5 coins. A quick look at this example shows that it is also possible to change the same amount of cash with only 3 seven (7) coins.

In other words, the greedy approach fails to find the best or smallest possible number of coins for this problem. Let's look at the dynamic programming solution of this example.

### Dynamic Programming of the Currency Exchange Problem

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
7	0	1	2	3	4	5	6	1	2	3	4	5	6	7	2	3	4	5	6	7	8	3
11	0	1	2	3	4	5	6	1	2	3	4	1	2	3	2	3	4	5	2	3	4	3

The first row in the table represents the cash value from 0 to 21. The three different currency denominations are represented in the first column. The entire second column is filled with zeros because a zero cash value is changed into zero coins for any denomination. Let's start by showing how to fill the rest of the second row where it is assumed that we only have a currency denomination of value 1 and where the cash value and the number of coins are equal since we only have coins of 1s in the second row. When we are on the third row, we now have coins of 1s and coins of 7. Cash values from one

to six can only use coins of 1s, but a cash value of seven will use one coin of seven that can be calculated as the minimum value between the number on top of that box and the successor of the value located in the seventh position on the left of that box. When this process is carried out for all the other rows of the table, it will result in the minimum number of coins for this problem: three.

### Summary

This unit explained how to organize data in relevant data structures, such as arrays, lists, queues, graphs, and trees. We reviewed how an iterative algorithm can be transformed to its recursive equivalent. Important algorithm design techniques were presented, including the divide-and-conquer strategy, the greedy approach, dynamic programming, and data balancing with the example of balanced binary search trees.

### Knowledge Check

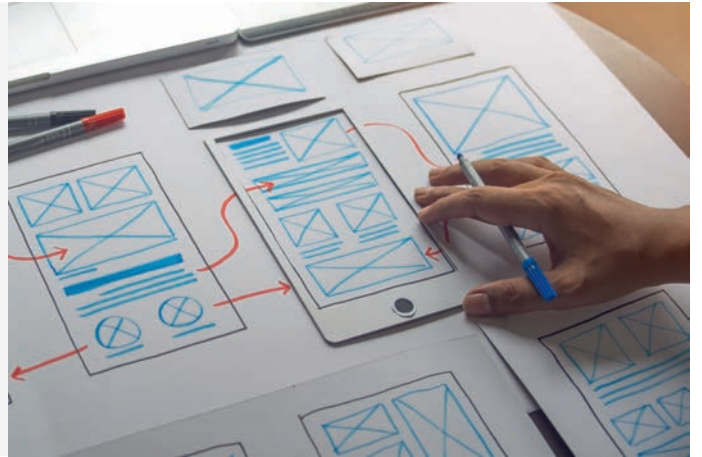
Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!



# Unit 3



## Some Important Algorithms

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... key details about the main searching and sorting algorithms.
- ... jq commands for pattern matching.
- ... fundamental cryptography concepts as applied to the RSA algorithm.
- ... essential data clustering concepts as visible in the k-means algorithm.

## 3. Some Important Algorithms

### Introduction

Algorithms are designed and developed for computational problems from a myriad of application domains and knowledge areas. However, a good proportion of algorithms relies on searching and sorting tasks in the process of crafting their own solutions. This is why searching and sorting algorithms are important, for example, for the task of matching patterns from a given text.

Digital security is a critical domain for the protection of the integrity of computing systems. It mainly relies on data encryption and decryption protective techniques offered by cryptography algorithms such as the RSA (Rivest–Shamir–Adleman) algorithm.

This unit is dedicated to searching and sorting algorithms, pattern matching, and the RSA algorithm. It also presents the k-means algorithm that plays an important role in data clustering. This is the task of subdividing a given set of objects into groups or clusters based on the analysis of their data.

### 3.1 Searching and Sorting

This section is dedicated to the presentation of searching and sorting algorithms because of their omnipresence in algorithmic tasks. An effort will be made to always compare the efficiency of the different algorithms.

#### Searching Algorithms

The following searching algorithms are presented in this sub-section: linear, binary, and hash search. A search problem consists simply of locating the position of a given value within a sequence of values. We will assume that the sequence of values is stored in an array. We will also assume that the search for a value in the array will return a position where that searched value is stored in the array, or it will return  $-1$  when a search did not find the expected value.

##### Linear search

The linear search algorithm simply goes through each element of a sequence of values in search of the expected one and returns with its position, if found; otherwise, it concludes that it did not find it. The number of comparisons of the linear search algorithm can be as high as the length of the sequence in situations where the value being searched is not found in the sequence (Subero, 2020).

##### Binary search

The binary search algorithm assumes that its sequence of elements is already sorted. It consists in dividing that sequence into two halves with the assumption of having only three alternatives: (i) the value being searched is located at the middle of the sequence

### Some Important Algorithms

and the search is over, (ii) the value being searched is located in the first half of the sequence and the search should continue only inside that chunk of the sequence, and (iii) the value being search is located in the second half of the sequence and the search should continue only inside that second half of the sequence. It is important to note that the size of the sequence being searched always shrinks by half until it eventually reaches the size of one in the worst case. At the beginning, three comparisons (equal, less than, greater than) are made on a value of the full size sequence. Then the same comparisons are made on a value of the appropriate half-size sub-sequence, and thereafter the same comparisons are made on a value of the consequent quarter-size sub-sequence, etc.

Let's consider the worst case scenario where the value being searched is not in the sequence. The following series represents the sizes of the successive sub-sequences being searched.

$$n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \frac{n}{2^4}, \frac{n}{2^5}, \frac{n}{2^6}, \frac{n}{2^7}, \dots, \frac{n}{2^{s-1}}, \frac{n}{2^s}$$

In other words, for a sequence with a length  $n$ , the binary search algorithm will successively consider  $s + 1$  sub-sequences and make three comparisons for each of them. The algorithm will ultimately stop when its input is reduced to a sub-sequence with a length of one, in other words, when the following equation is fulfilled.

$$\frac{n}{2^s} = 1$$

or

$$2^s = n$$

or

$$s = \log_2(n)$$

This equation gives us an idea about the logarithmic efficiency of the binary search algorithm (Subero, 2020).

### Hash search

The hash search algorithm assumes that there is a sequence of elements where each element is identified by a unique key that is either a natural number or a string. Searching strings is possible here because it is always possible to transform any string into its integer equivalent using ASCII codes. The hash search algorithm uses these keys to store the different elements in a hash table that can easily be searched later. In fact, each element of the sequence is stored in the hash table in the position of the hash function of its key. So, if we are looking for an element, we simply have to calculate the hash function of its key and locate it at that position. The first problem is that it is possible for many different keys to have the same hash function. This is called a collision, and it leads to an extra search in the position of the collided keys. The second

problem is that there are no standard hash functions for all types of keys. This subsection uses the Multiplication, Addition, and Division (MAD) hash function to illustrate the concept of hash search.

Let's start from the assumption that there are  $n$  keys to search from, and  $p$  is the smallest prime number that is greater or equal to  $n$ . Two random numbers  $a$  and  $b$  are then chosen between 0 and  $p - 1$  with  $a$  being strictly positive. The MAD hash function  $h$  is

$$h(k) = ((ak + b) \bmod p) \bmod n$$

Let us suppose that we have the following 24 keys to search from: 19, 20, 30, 31, 67, 125, 189, 192, 267, 357, 388, 393, 428, 435, 483, 513, 574, 592, 645, 744, 794, 916, 954, and 980.

Twenty-nine is the smallest prime number that is greater than 24; and we will use 4 and 15 as the respective values of  $a$  and  $b$ . In other words, the values of  $n$ ,  $p$ ,  $a$ , and  $b$  are respectively 24, 29, 4, and 15. The first column of the table below contains the values of the different keys. The fourth column contains the calculated hash function. It is the index of the place where values are stored. When searching for a key, the hash search algorithm will calculate the hash function of that key and locate it in the matching index of the hash table. In case of collusion, the matching index will contain many keys that must be searched with a different search algorithm. The speed of the hash search algorithm mainly depends on the speed of the algorithm of the hash function and on the number of collisions.

### Sorting Algorithms

The history of sorting algorithms includes radix sort, merge sort, insertion sort, counting sort, bubble sort, bucket sort, quicksort, introsort, timesort, library sort, and burst sort (Attard Cassar, 2018). In this section, we will only focus on the fundamental sorting algorithms: radix sort, merge sort, insertion sort, bubble sort, bucket sort, and quicksort. We will not dwell on how these fundamental sorting algorithms were extended by other sorting algorithms.

## Some Important Algorithms

## Sorting Algorithms

Key (k)	ak + b	(ak+b) mod p	((ak+b) mod p) mod n	Index	Keys	Keys	Keys
19	91	4	4	0	192		
20	95	8	8	1	388	794	916
30	135	19	19	2			
31	139	23	23	3	954		
67	283	22	22	4	19	483	744
125	515	22	22	5	592		
189	771	17	17	6			
192	783	0	0	7			
267	1083	10	10	8	20	513	
357	1443	22	22	9			
388	1567	1	1	10	267		
393	1587	21	21	11			
428	1727	16	16	12			
435	1755	15	15	13			
483	1947	4	4	14	645		
513	2067	8	8	15	435		
574	2311	20	20	16	428		
592	2383	5	5	17	189		
645	2595	14	14	18			
744	2991	4	4	19	30		
794	3191	1	1	20	574	980	
916	3679	25	1	21	393		
954	3831	3	3	22	67	125	357
980	3935	20	20	23	31		

**Radix sort**

Although the radix sort algorithm was invented by Hollerith in the 1880s, the first computerized version was proposed by Seward (Attard Cassar, 2018). It works well with natural numbers where it makes use of the different digits of these numbers starting from the least significant ones to the most significant ones. This algorithm uses ten buckets numbered from zero to nine because it knows that each decimal digit has a value between these two numbers.

At the beginning of the algorithm, each number is put in the bucket of its least significant digit (digit 0, see the second column in the table below). Then each number is put in the bucket of its second least significant digit (digit 1, see the third column in the table below). This process ends when all the digits' positions are covered. This is illustrated below for the numbers 7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, and 174.

### Radix Sort Algorithm Example

Bucket	Digit 0	Digit 1	Digit 2	Digit 3	Digit 4	Digit 5
0	500	500, 6, 7	6, 7, 26, 28, 45	6, 7, 26, 28, 45, 174, 500, 578	6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648	6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, 65947
1			174			
2		26, 28				
3				3974		
4	3974, 174	45, 6648, 65947				
5	45		500, 578	65947		
6	26, 6		6648	6648	65947	
7	7, 65947	3974, 174, 578				
8	6648, 578, 28					
9			65947, 3974			

The total number of operations performed by the radix sort algorithm depends on the number of elements to be sorted and on the highest number of significant digits for those elements. In fact, each element is checked for each level of significance.

### Bucket sort

The bucket sort algorithm assumes that it is given different sorted ranges or buckets to which the various values to be sorted belong. It then simply places each value inside its range or bucket before sorting each bucket with another sorting algorithm. Thereafter, the sorted buckets are concatenated to get the final sorted sequence of values. This is illustrated below for the numbers 7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, and 174. This example uses the following buckets or ranges: [0...9], [10...99], [100...999], [1000...9999], and [10000...99999].

## Some Important Algorithms

## Bucket Sort Algorithm

Bucket	Values	Bucket	Values sorted with another algorithm
[0...9]	7, 6	[0...9]	6, 7
[10...99]	26, 45, 28	[10...99]	26, 28, 45
[100...999]	500, 578, 174	[100...999]	174, 500, 578
[1000...9999]	6648, 3974	[1000...9999]	3974, 6648
[10000...99999]	65947	[10000...99999]	65947

The final sorted sequence for this example is 6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, and 65947. This algorithm relies on the uniform distribution of values in the buckets even though such a distribution is not guaranteed, and its total number of operations depends on the other sorting algorithm that it is using.

**Insertion sort**

The insertion sort algorithm searches for the smallest value in a given sequence and exchanges it with the value at the beginning of that sequence. It then ignores this newly updated first element of the sequence in order to restart the above process with the rest of the sequence, up to the point where the entire sequence is sorted. The following example is an illustration of the insertion sort algorithm for the numbers 7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, and 174.



## Insertion Sort Algorithm Example

Input values	ei	mi	Output values
7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, 174	0	3	6, 500, 26, 7, 6648, 578, 45, 65947, 28, 3974, 174
6, 500, 26, 7, 6648, 578, 45, 65947, 28, 3974, 174	1	3	6, 7, 26, 500, 6648, 578, 45, 65947, 28, 3974, 174
6, 7, 26, 500, 6648, 578, 45, 65947, 28, 3974, 174	2	2	6, 7, 26, 500, 6648, 578, 45, 65947, 28, 3974, 174
6, 7, 26, 500, 6648, 578, 45, 65947, 28, 3974, 174	3	8	6, 7, 26, 28, 6648, 578, 45, 65947, 500, 3974, 174
6, 7, 26, 28, 6648, 578, 45, 65947, 500, 3974, 174	4	6	6, 7, 26, 28, 45, 578, 6648, 65947, 500, 3974, 174
6, 7, 26, 28, 45, 578, 6648, 65947, 500, 3974, 174	5	10	6, 7, 26, 28, 45, 174, 6648, 65947, 500, 3974, 578
6, 7, 26, 28, 45, 174, 6648, 65947, 500, 3974, 578	6	8	6, 7, 26, 28, 45, 174, 500, 65947, 6648, 3974, 578
6, 7, 26, 28, 45, 174, 500, 65947, 6648, 3974, 578	7	10	6, 7, 26, 28, 45, 174, 500, 578, 6648, 3974, 65947
6, 7, 26, 28, 45, 174, 500, 578, 6648, 3974, 65947	8	9	6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, 65947
6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, 65947	9	9	6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, 65947
6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, 65947	10	10	6, 7, 26, 28, 45, 174, 500, 578, 3974, 6648, 65947

For the table above, ei is the abbreviation of Exchange Index, and mi is the abbreviation of Minimum Index. When the insertion sort algorithm is looking for its minimal value for the first time, it checks almost the entire sequence of numbers. But, in the second time, its number of checks is reduced by one, by one again for the third time, and so on until there is nothing to check. The total number of checks and swaps made by the insertion sort algorithm in the worst case scenario is, thus, almost equal to the following formula where  $n$  represents the number of elements in the sequence to be sorted.

$$(n - 1) + (n - 2) + (n - 3) + \dots + 4 + 3 + 2 + 1 = \frac{n(n - 1)}{2}$$

**Bubble sort**

The bubble sort algorithm consists of continuously swapping neighboring values in a sequence of elements so as to shift the highest value at the end of the sequence. The following example is an illustration of how the bubble sort algorithm shifts the highest value of the following numbers to the end of the sequence: 7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, and 174.

## Some Important Algorithms

## Bubble Sort Algorithm Example (Start)

Before swapping the two yellow neighbours	After swapping the two yellow neighbours
7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, 174	7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, 174
7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, 174	7, 26, 500, 6, 6648, 578, 45, 65947, 28, 3974, 174
7, 26, 500, 6, 6648, 578, 45, 65947, 28, 3974, 174	7, 26, 6, 500, 6648, 578, 45, 65947, 28, 3974, 174
7, 26, 6, 500, 6648, 578, 45, 65947, 28, 3974, 174	7, 26, 6, 500, 6648, 578, 45, 65947, 28, 3974, 174
7, 26, 6, 500, 6648, 578, 45, 65947, 28, 3974, 174	7, 26, 6, 500, 578, 6648, 45, 65947, 28, 3974, 174
7, 26, 6, 500, 578, 6648, 45, 65947, 28, 3974, 174	7, 26, 6, 500, 578, 45, 6648, 65947, 28, 3974, 174
7, 26, 6, 500, 578, 45, 6648, 65947, 28, 3974, 174	7, 26, 6, 500, 578, 45, 6648, 65947, 28, 3974, 174
7, 26, 6, 500, 578, 45, 6648, 65947, 28, 3974, 174	7, 26, 6, 500, 578, 45, 6648, 28, 65947, 3974, 174
7, 26, 6, 500, 578, 45, 6648, 28, 65947, 3974, 174	7, 26, 6, 500, 578, 45, 6648, 28, 3974, 65947, 174
7, 26, 6, 500, 578, 45, 6648, 28, 3974, 65947, 174	7, 26, 6, 500, 578, 45, 6648, 28, 3974, 174, 65947
7, 26, 6, 500, 578, 45, 6648, 28, 3974, 174, 65947	7, 26, 6, 500, 578, 45, 6648, 28, 3974, 174, 65947

The same process is carried out for the following remaining sequence of numbers as is illustrated below: 7, 26, 6, 500, 578, 45, 6648, 28, 3974, and 174.

## Bubble Sort Algorithm Example (End)

Swapping position	Before swapping the two yellow neighbours	After swapping the two yellow neighbours
0	7, 26, 6, 500, 578, 45, 6648, 28, 3974, 174	7, 26, 6, 500, 578, 45, 6648, 28, 3974, 174
1	7, 26, 6, 500, 578, 45, 6648, 28, 3974, 174	7, 6, 26, 500, 578, 45, 6648, 28, 3974, 174
2	7, 6, 26, 500, 578, 45, 6648, 28, 3974, 174	7, 6, 26, 500, 578, 45, 6648, 28, 3974, 174
3	7, 6, 26, 500, 578, 45, 6648, 28, 3974, 174	7, 6, 26, 500, 578, 45, 6648, 28, 3974, 174
4	7, 6, 26, 500, 578, 45, 6648, 28, 3974, 174	7, 6, 26, 500, 45, 578, 6648, 28, 3974, 174
5	7, 6, 26, 500, 45, 578, 6648, 28, 3974, 174	7, 6, 26, 500, 45, 578, 6648, 28, 3974, 174
6	7, 6, 26, 500, 45, 578, 6648, 28, 3974, 174	7, 6, 26, 500, 578, 45, 28, 6648, 3974, 174
7	7, 6, 26, 500, 45, 578, 28, 6648, 3974, 174	7, 6, 26, 500, 45, 578, 28, 3974, 6648, 174
8	7, 6, 26, 500, 45, 578, 28, 3974, 6648, 174	7, 6, 26, 500, 45, 578, 28, 3974, 174, 6648
9	7, 6, 26, 500, 45, 578, 28, 3974, 174, 6648	7, 6, 26, 500, 45, 578, 28, 3974, 174, 6648

Now that the highest value 6648 has been shifted to the end of its sequence, the bubble sort algorithm can continue with the remaining sequence of numbers 7, 6, 26, 500, 45, 578, 28, 3974, 174 until the entire sequence is sorted. The number of swapping positions is almost equal to the number of elements in the sequence, and the lengths of the sequences where swapping happens successively decrease by one starting from the length of the full sequence. In other words, in the worst case, the number of swaps done by the bubble sort algorithm is similar to the number of operations that are performed by the insertion sort algorithm.

### Merge sort

Merge sort algorithm  
The merge sort algorithm efficiently joins two already sorted arrays into a newly sorted array.

The **merge sort algorithm** starts with the partitioning of its sequence of values into a set of singleton arrays (single-element arrays). Each of these singleton arrays is sorted since they only contain one value. This allows the merge sort algorithm to continuously merge all neighboring pairs of arrays until the entire array is sorted. Before illustrating the merge sort algorithm itself, it is important to first understand the following example on how to merge two sorted arrays: 1, 3, 5, 6 and 2, 5, 5, 6, 9.





$$\frac{n}{2^s} = 1$$

or

$$2^s = n$$

or

$$s = \log(n)$$

This means that the merge sort algorithm roughly changes its merging size  $\log(n)$  times, and it makes almost  $n$  comparisons for each of those merging sizes for a total number of comparisons of almost  $n \log(n)$  for the entire merge sort algorithm.

### Quicksort

The quicksort algorithm (Hoare, 1961) chooses a pivot value (possibly initialized with the first element of the sequence to sort) and places it in a suitable position so that every value on the left of that position is less than or equal to the pivot value, and every value on the right of that position is greater than or equal to the value of the pivot. This process is carried out many times for the sequence on the left of the pivot and for the one on its right until the length of the sequence to be sorted is equal to one.

The quicksort algorithm has two cursors, a left-to-right  $\rightarrow$  cursor that only moves from the left to the right, and a right-to-left  $\leftarrow$  cursor that only moves from the right to the left. At the start of the algorithm, the  $\rightarrow$  cursor points to the beginning of the sequence being sorted while the  $\leftarrow$  cursor points to its last element. The  $\rightarrow$  cursor stops moving when it finds a value that is greater than the value of the pivot, but the  $\leftarrow$  cursor continues moving as long as it is meeting values that are greater than the one of the pivot.

Two cases are possible when both cursors stop moving. The first is that the position of the  $\leftarrow$  cursor is higher than the one of the  $\rightarrow$  cursor. In this case, the two cursors will simply exchange the values in their positions and continue with their moves. The second case is that the position of the  $\leftarrow$  cursor is lower than the one of the  $\rightarrow$  cursor. In this second case, the following two steps are carried out:

1. The position of the  $\leftarrow$  cursor exchanges its value with the one in the position of the current pivot.
2. The entire quicksort algorithm starts all over again for the sub-sequence on the left side of the  $\leftarrow$  cursor and for the one on its right side until both sub-sequences are left with one element maximum.

There is no guarantee that the found correct place of the pivot will divide the sequence being sorted into two almost equal-sized left and right sub-sequences. This is why, in the worst case scenario, the total number of operations performed by the quicksort algorithm is roughly equal to the following ( $n$  is the length of the sequence being sorted).

## Some Important Algorithms

$$(n - 1) + (n - 2) + (n - 3) + \dots + 4 + 3 + 2 + 1 = \frac{n(n - 1)}{2}$$

The following table illustrates the quicksort algorithm for the sequence of numbers 7, 500, 26, 6, 6648, 578, 45, 65947, 28, 3974, and 174.

## Quicksort Algorithm Example

→ Stop if value > than pivot						Carry on if value > than pivot ←				
7	500	26	6	6648	578	45	65947	28	3974	174
	→		←							
7	6	26	500	6648	578	45	65947	28	3974	174
	←	→								
6	7	26	500	6648	578	45	65947	28	3974	174

→ Stop if value > than pivot						Carry on if value > than pivot ←				
6	7	26	500	6648	578	45	65947	28	3974	174
		←	→							
6	7	26	500	6648	578	45	65947	28	3974	174

→ Stop if value > than pivot						Carry on if value > than pivot ←				
6	7	26	500	6648	578	45	65947	28	3974	174
				→						←
6	7	26	500	174	578	45	65947	28	3947	6648
					→			←		
6	7	26	500	174	28	45	65947	578	3974	6648
						←	→			
6	7	26	45	174	28	500	65497	578	3974	6648

			→		←		→			←
6	7	26	45	174	28	500	65497	578	3974	6648
				→	←					←→
6	7	26	45	28	174	500	6648	578	3974	65947
				←	→					←→
6	7	26	28	45	174	500	6648	578	3974	65497

							→		←	
6	7	26	28	45	174	500	6648	578	3974	65947
									←→	
6	7	26	28	45	174	500	3974	578	6648	65947

							→	←		
6	7	26	28	45	174	500	3974	578	6648	65947
								←→		
6	7	26	28	45	174	500	578	3974	6648	65947



## 3.2 Pattern Matching

This section focuses on two different applications of pattern matching. First, it looks at how to define a pattern of characters so that we can check if and where it is matched by a given string. Such patterns are known as regular expressions, and many modern programming languages process them under the RegExps abbreviation. This section gives an overview of RegExps in JavaScript. It is also possible to define patterns and search them in JavaScript objects. Such objects are described with the use of the JavaScript Object Notation (JSON). The second part of this section is dedicated to the `jq` utility where it is possible to define a pattern and match it against a JSON object.

### Regular Expressions or RegExps

**Regular expressions** (RegExps) are constructed from the following building blocks: basic operations, ranges, the escape feature, anchors, and commonly used classes. A character is the most basic unit of a RegExp, and the dot (`.`) stands for any character. Regular expressions are concatenated by simply placing them one next to the other, while piping (`|`) two expressions means that we are interested in either of the expressions. The same role is played by the square-bracketing of regular expressions inside, where it is possible to insert a caret (`^`) to indicate that none of the expressions should be matched. The round-bracketing of a regular expression forces it to be evaluated.

Regular expression  
A regular expression describes the format of how a given text should be written.

The `*` operator repeats its regular expression zero or more times, while the `+` operator repeats its expression one or more times, and the `?` operator simply checks whether its regular expression appears at most once. The escape character `\` must be applied to special characters, such as those previously identified, in order to cancel their special meaning. The minus sign `-` when placed between two characters inside square brackets `[]` represents any value in the range between these two characters, but it loses that special meaning if it is located at the beginning of the left square bracket `[`.

The following expressions are commonly used by regular expressions. `\d` stands for any numeric digit, `\w` stands for any alphanumeric character or for `_`, and `\s` stand for the white space character. The capitalization of these expressions leads to their negation: `\D` stands for any character that is not a numeric digit, `\W` stands for any character that is not an alphanumeric character nor `_`, and `\S` stands for any character that is not a white space.

Four anchoring tags are used to indicate the place where we want our expression to be: The caret character (`^`) places the regular expression at the start of the sentence while the dollar sign (`$`) places it at the end. Similarly, the `\<` tag (`\b` in JavaScript) places its regular expression at the beginning of the word while `\>` (`\b` in JavaScript) places it at the end.

## Jq and Json Objects

While regular expressions are used to match desired patterns in strings, they can also be used to match desired patterns in objects. Here, we focus on how to use `jq` commands to search specific patterns in objects.

In `jq`, the dot (`.`) symbol simply refers to the entire set of objects being searched, but placing the name of an attribute after that symbol restricts the listed values to that attribute. The `keys` keyword lists all the attributes' names but that keyword can be given an index to specify a specific attribute. Square brackets are used for arrays. The comma sign can be used to apply different filters to the same set of objects. The `length` function gives the number of elements in an array, the number of attributes in an object, or the length of a string, depending on the nature of its parameter. This parameter is passed to it though the `|` pipe character. Some of these `jq` filtering patterns are presented in the table below.

jq Filtering Patterns (Selection)	
Command Line	Job Description
<code>jq .</code>	This command lists all the objects being searched. This is useful to format the json input simply.
<code>jq .attrName</code>	This command lists the values of all the objects for the attribute <code>attrName</code> .
<code>jq .arrayAttr[p]</code>	This command lists the values of all the objects at the position <code>p</code> of the array attribute <code>arrayAttr</code> .
<code>jq .[p]</code>	This command gives the value in position <code>p</code> of the object being searched, provided that such object is an array
<code>jq filt   length</code>	This command gives the number of elements or the length of the result obtained from filtering the objects being searched with the filter <code>filt</code> .

We will restrict ourselves to the following `jq` general command to look for regular expressions in the object located in the `json` file `f.json`.

```
jq 'fltPat | fct ("regExp" ; "flg")' f.json
```

## Some Important Algorithms

In this command, `fltPat` represents the filtering patterns that follow the `jq` keyword in the first column of the above table. `fct` stands for any of the following functions: `match`, `test`, or `capture`. The `match` function gives a four-field description of the occurrences of the `regex` regular expression in the filtered object(s), while the `test` function simply gives a `true` or `false` answer. The `capture` function is used for the naming of the matched patterns, for example, to identify the different parts of a string. Here, `flg` stands for optional flags.

We now illustrate these different `jq` commands' parameters with the following Json object. We will not use the command line version of `jq` because of its issues in the processing of single and double quotes for different operating systems. Instead, we will use the online version of `jq` by simply specifying the designed filters with the assumption that the content of the following object is already present in the designated text-box of the `jq` online tool.

```

Json Object First Example

{
  "RespId":["0028","50706","0109"],
  "ContId":["ZA","AF","EU"],
  "Gender":["m","Fe","bfm"],
  "Issues":
    {
      "Voting":["20","1000","19"],
      "Internet":["free","paid","paid"]
    }
}

```

The `RespId` field is an array where each element is assumed to be a sequence of four decimal digits. The `ContId` and the `Gender` fields are also arrays. `ContId` represents the name of a continent assuming that there are six continents known under the six abbreviations AF, AS, AU, EU, NA, and SA.

The following filter will check each element of the `RespId` array to find if it respects the above described format.

```
.RespId[] | test("^[0-9]{4}$")
```

Let's examine this `jq` filter from left to right, starting with the full stop sign that simply states that `RespId` is an attribute of the `Json` object being filtered. The square brackets `[]` are an indication that `RespId` is an array and that we are filtering each of its elements. The first pipe `|` separates the input string (ending with `[]`) from the function of the filter (`test` in this case). A regular pattern expression is inside the `test` function and is formatted as a sequence of exactly four digits. The presence of the caret (`^`) and the dollar sign (`$`) at the beginning and at the end of the `test` function, respectively, is

an indication that the given pattern is expected to match the entire word, not only its beginning nor end. The output of the `jq` filter for our displayed illustrative `json` object (above): `true`, `false`, `true`. It shows that the second element did not match the specified pattern because it has five digits instead of four.

The following filter will check each element of the `ContId` array to find if it respects the format described above. The pipes characters `|` inside the `test` function represent the OR operator.

```
.ContId[] | test("^(AF|AS|AU|EU|NA|SA)$")
```

The output of this `jq` filter for our displayed illustrative `json` object is: `false`, `true`, `true`. It shows that the first element did not match the specified pattern because its value `ZA` is not a recognized continent. The `Gender` field is also an array. It is assumed that the recognized genders are `M`, `F`, `U`, `m`, `f`, and `u`, where `U` and `u` both stand for the unspecified gender. Here, a gender is valid if and only if it starts with any of the above listed six characters.

The following command checks each element of the `Gender` array to find if it respects the format described above.

```
.Gender[] | test("^[FMUfmu]")
```

Here is the output of the above `jq` filter for our above displayed illustrative `json` object: `true`, `true`, `false`. It shows that the third element did not match the specified pattern because its value `bfm` does not start with any of the six recognized alphabetical characters.

The `Issues` field is an object with two sub-fields `Voting` and `Internet` that are both arrays. Let's assume that the `Voting` field represents the voting age as a natural number between 0 and 999. The following command will check the conformity to that specification.

```
.Issues.Voting[] | test("^[0-9]{1,3}$")
```

This time, the output of the `Jq` command is "`false`, `false`, `true`," for the reasons you have surely already identified

We will now show how these four `jq` commands can be transformed from being `test` functions into becoming `match` functions so that answers can be shown as matched objects instead of being mere `true` or `false` outputs. For ease of reference, the above `jq` filters are denoted by `C1`, `C2`, `C3`, and `C4` as seen below.

Notation	Jq Filter
C1	<code>.RespId[]   test("^[0-9]{4}\$")</code>

## Some Important Algorithms

Notation	Jq Filter
C2	<code>.ContId[]   test("^(AF AS AU EU NA SA)\$")</code>
C3	<code>.Gender[]   test("^[FMUfmu]")</code>
C4	<code>.Issues.Voting[]   test("^[0-9]{1,3}\$")</code>

The `match` equivalent of the above `test` table is the following.

Notation	Jq Filter
C5	<code>.RespId[]   match("^[0-9]{4}\$")</code>
C6	<code>.ContId[]   match("^(AF AS AU EU NA SA)\$")</code>
C7	<code>.Gender[]   match("^[FMUfmu]")</code>
C8	<code>.Issues.Voting[]   match("^[0-9]{1,3}\$")</code>

The outputs for the above `match` commands for our above illustrative `Json` object example are presented by the following table.

## Outputs

C5	C6
<pre> {   "offset": 0,   "length": 4,   "string": "0028",   "captures": [     {       "offset": 0,       "length": 4,       "string": "0028",       "name": null     }   ] } {   "offset": 0,   "length": 4,   "string": "0109",   "captures": [     {       "offset": 0,       "length": 4,       "string": "0109",       "name": null     }   ] } </pre>	<pre> {   "offset": 0,   "length": 2,   "string": "AF",   "captures": [     {       "offset": 0,       "length": 2,       "string": "AF",       "name": null     }   ] } {   "offset": 0,   "length": 2,   "string": "EU",   "captures": [     {       "offset": 0,       "length": 2,       "string": "EU",       "name": null     }   ] } </pre>
C7	C8
<pre> {   "offset": 0,   "length": 1,   "string": "m",   "captures": [] } {   "offset": 0,   "length": 1,   "string": "F",   "captures": [] } </pre>	<pre> {   "offset": 0,   "length": 2,   "string": "19",   "captures": [     {       "offset": 0,       "length": 2,       "string": "19",       "name": null     }   ] } </pre>

## Some Important Algorithms

You may have noticed that the results of the `match` commands are quite verbose. We will now use the `capture` functionality to extract the information that we would like to focus on, as shown by the following new commands.

Ref.	Jq Filter
C9	<code>.RespId[]   capture("^(?&lt;idNo&gt;([0-9]{4}))\$")</code>
C10	<code>.ContId[]   capture("^(?&lt;ctn&gt;(AF AS AU EU NA SA))\$")</code>
C11	<code>.Gender[]   capture("^(?&lt;Gender&gt;([FMUfmu]))")</code>
C12	<code>.Issues.Voting[]   capture("^(?&lt;vote&gt;([0-9]{1,3}))\$")</code>

The results of the above `capture` commands are presented below.

C9	C10	C11	C12
<pre>{   "idNo":   "0028" } {   "idNo":   "0109" }</pre>	<pre>{   "ctn": "AF" } {   "ctn": "EU" }</pre>	<pre>{   "Gender": "m" } {   "Gender": "F" }</pre>	<pre>{   "vote": "19" }</pre>

For the sake of completeness, let us try our `test` and `capture` commands on a different Json object (displayed below), and check if we get the expected results.



### Json Object Second Example

```

{
  "RespId":["0000","07","109"],
  "ContId":["AF","ASIA","EU"],
  "Gender":["not known","u","Male"],
  "Issues":
  {
    "Voting":["16 age","180","teen"],
    "Internet":["free","paid service","paid"]
  }
}

```

The respective outputs for the eight tests and capture commands above are presented by the following table for the new json object.

Output for Eight tests and capture Commands			
Ref.	First element	Second element	Third element
C1	true	false	false
C2	true	false	true
C3	false	true	true
C4	false	true	false
C9	{ "idNo": "0000" }		
C10	{ "ctn": "AF" }		{ "ctn": "EU" }
C11		{ "Gender": "u" }	{ "Gender": "M" }

## Some Important Algorithms

Output for Eight tests and capture Commands			
Ref.	First element	Second element	Third element
C12		{ "vote": "180" }	

Readers are invited to check the veracity of the above table.

### 3.3 The RSA Algorithm

The RSA algorithm is currently used worldwide to secure the transmission of data and information. It is a public-key asymmetric encryption algorithm that uses both private and public keys. The RSA acronym stands for Rivest, Shamir, and Adleman who are the three scholars who authored the RSA algorithm in 1978. This section presents the strengths and weaknesses of this algorithm.

#### Encryption, Decryption, and Signatures

Put yourself for a moment in the situation where you have to choose two different prime numbers  $p_1$  and  $p_2$ . Let's say that you have chosen for example 11 and 23. Let us denote by  $m_1$  the multiplication of  $p_1$  by  $p_2$  ( $m_1 = 11 \cdot 23 = 253$ ), and  $m_2$  the multiplication of  $p_1 - 1$  by  $p_2 - 1$  ( $m_2 = 10 \cdot 22 = 220$ ). You are now requested to choose a strictly positive number  $e$  less than  $m_2$  and coprime with it ( $e$  and  $m_2$  should not share a common divisor except for 1, for example,  $e = 9$ ). Let us now calculate a value  $d$  such that  $e \cdot d$  is the immediate successor of a multiple of  $m_2$  (for example,  $e \cdot d = 1 + (2 \cdot 220) = 441$ . So,  $d = 49$ ). We can now **encrypt** and decrypt messages. The encryption of a number  $n$  can be achieved with the following formula.

$$\text{Encrypt}(n) = n^e \bmod m_1$$

The decryption of the above encrypted message is achieved with the following formula.

$$\text{Decrypt}(n) = n^d \bmod m_1$$

Let us now take the example of non-case-sensitive English text messages made up exclusively of alphabetical characters. There are 26 letters in the English alphabet, so each character can be converted into its numeric equivalent as a number between 01 and 26, representing the characters a and z, respectively.

#### Encryption

To encrypt a message is to transform it so that it cannot be understood without having been decrypted.

Suppose we want to send the text *Yes* in its encrypted format. First, we must convert *Yes* into its digits' format 25, 05, 19. We can now use  $e = 9$ ,  $d = 49$ , and  $m_1 = 253$  to encrypt 25, 05, 19 and decrypt it later as seen in the following table.

$\text{Encrypt}(25) = 25^9 \bmod 253 = 213$	$\text{Decrypt}(213) = 213^{49} \bmod 253 = 25$
$\text{Encrypt}(05) = 5^9 \bmod 253 = 218$	$\text{Decrypt}(218) = 218^{49} \bmod 253 = 5$
$\text{Encrypt}(19) = 19^9 \bmod 253 = 194$	$\text{Decrypt}(194) = 194^{49} \bmod 253 = 19$

The encrypted message is thus equal to 213, 218, 194, and its decryption will return the values 25, 05, 19. Even when we know the values of  $e$  and  $m_1$ , we will still not be able to decrypt messages without knowing the value of  $d$ . This explains why  $e$  and  $m_1$  are considered public keys, and  $d$  is considered the private key. Conversely, if a message can be decrypted by the public keys  $e$  and  $m_1$ , it is because it was encrypted by the private key  $d$ . In other words, a private key can be used to encrypt messages that can be decrypted by public key holders with the assurance that these messages were encrypted with the use of the private key. This is how a private key is used as the digital signature of its owner.

We can reverse the above example to illustrate this concept of digital signature by allowing the private key owner to use their private key  $d = 49$  to encrypt and sign their message 25, 05, 19 so that public key holders can decrypt it with the public keys  $e = 9$  and  $m_1 = 253$ .

$\text{Encrypt}(25) = 25^{49} \bmod 253 = 147$	$\text{Decrypt}(147) = 147^9 \bmod 253 = 25$
$\text{Encrypt}(05) = 5^{49} \bmod 253 = 020$	$\text{Decrypt}(20) = 20^9 \bmod 253 = 5$
$\text{Encrypt}(19) = 19^{49} \bmod 253 = 172$	$\text{Decrypt}(172) = 172^9 \bmod 253 = 19$

The RSA encryption system can be summarized in the following illustrated situation whereby secret messages are exchanged between the members of a group and their leader. Each group member knows the values of the public keys  $e$  and  $m_1$  but only the group leader knows the value of the private key  $d$ . When a person wants to send a message to the group leader, the sender must encrypt their message with the public keys  $e$  and  $m_1$  so that only the group leader can decrypt it with the private key  $d$ . On the other

## Some Important Algorithms

hand, when the group leader wants to send a message to the group members, they must encrypt it with the private key  $d$  so that any member can use the public keys  $e$  and  $m_1$  to decrypt it with the assurance that the message is coming from the group leader. The RSA encryption system is asymmetric in the sense that the encryption and the decryption keys are different.

### Strengths and Weaknesses of the RSA Algorithm

The strength of the RSA algorithm comes from the fact that  $m_1$  is the product of two prime numbers  $p_1$  and  $p_2$ , and for large values of  $m_1$  it is very difficult to identify  $p_1$  and  $p_2$  even when we know the value of  $m_1$ . That makes it equally difficult to find the value of  $m_2$  and, consequently, that of  $d$  even when we know the value of  $e$ .

Choosing large values for  $m_1$  lowers the speed of the RSA algorithm. However, the RSA cryptography system is less secure with smaller values of  $m_1$  because it is easier to factorize them. The choice of the values of  $m_1$  is the Achilles heel of the RSA algorithm: On one hand, large values of  $m_1$  make the RSA algorithm more secure but they reduce its speed, and on the other hand, smaller values of  $m_1$  speed up the RSA algorithm but they make it less secure. Let us also mention that the message needs to be unknown; otherwise, the key becomes known.

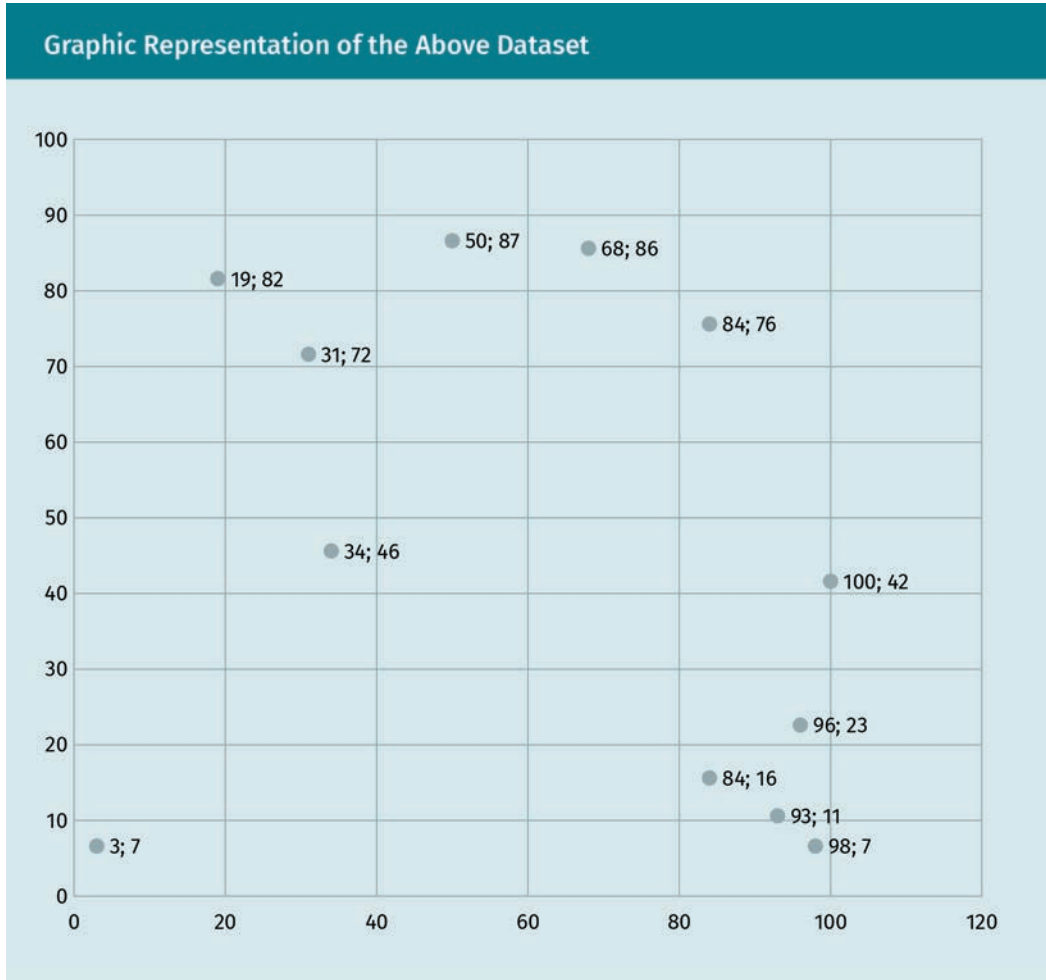
## 3.4 The K-Means Data Clustering Algorithm

The concept of data clustering refers to the process of dividing a dataset into clusters or groups so that the closest elements of the dataset are assigned to the same cluster. Data clustering is part of data mining because its clustering process only relies on the data itself. It has several applications in image processing, market research, and geographical information systems (GIS).

This section will restrict itself to two-dimensional coordinate datasets where it is simple to calculate the distance between two data elements. Invented by Lloyd (1982), the k-means name comes from the fact that the k-means algorithm divides its dataset into  $k$  clusters or groups. These clusters are built based on the distance to the mean value of each cluster. Here is an illustration of the k-means algorithm for the following two-dimensional dataset.

### K-Means Algorithm Example

E	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12
X	34	3	100	50	96	19	98	68	84	31	93	84
Y	46	7	42	87	23	82	7	86	76	72	11	16



The first step of the k-means algorithm is to choose the value of  $k$ . For our example, we choose  $k = 4$  to state that we want our data to be partitioned into 4 clusters. This algorithm also requires the random choice of  $k$  elements from the dataset as the initial mean values to start working. We can, for example, choose E1, E6, E7, and E11 as our respective initial mean values  $M_1$ ,  $M_2$ ,  $M_3$ , and  $M_4$ . We then have to calculate the Euclidean distance of each element to each mean value and assign each element to the cluster of its closest mean (see the first step table below). The first step table shows that so far, the dataset has been divided into the following clusters: {E1, E2, E9}, {E4, E6, E8, E10}, {E7}, and {E3, E5, E11, E12}.

We will now calculate the mean values of each of these clusters both for the X and Y components, and we will get the following four means-elements:  $M_1(40.33; 43)$ ,  $M_2(42; 81.75)$ ,  $M_3(98; 7)$ , and  $M_4(93.25; 23)$ . It is now time to check each element again in order to identify which of the  $M_1$ ,  $M_2$ ,  $M_3$ , or  $M_4$  mean-elements it is closest to and assign it to the corresponding cluster (second step table).

## Some Important Algorithms

## K-Means Algorithm Example First Step

Elements	X	Y	DistToM1	DistToM2	DistToM3	DistToM4	Minim
E1	34	46	0	39	74,94665	68,60029	0
E2	3	7	49,81967	76,68768	95	90,08885	49,81967
E3	100	42	66,1211	90,33825	35,0571	31,7805	31,7805
E4	50	87	44,01136	31,40064	93,29523	87,32125	31,40064
E5	96	23	66,12866	97,00515	16,12452	12,36932	12,36932
E6	19	82	39	0	108,9312	102,5524	0
E7	98	7	74,94665	108,9312	0	6,403124	0
E8	68	86	52,49762	49,16299	84,50444	79,05694	49,16299
E9	84	76	58,30952	65,27634	70,40597	65,62012	58,30952
E10	31	72	26,1725	15,6205	93,34881	86,97701	15,6205
E11	93	11	68,60029	102,5524	6,403124	0	0
E12	84	16	58,30952	92,63369	16,64332	10,29563	10,29563

The following table shows that, so far, the dataset has been divided into the following clusters: {E1, E2}, {E9, E4, E6, E8, E10}, {E7, E11}, and {E3, E5, E12}.

## K-Means Algorithm Example Second Step

Elements	X	Y	DistToM1	DistToM2	DistToM3	DistToM4	Minim
E1	34	46	7,007932	36,63417	74,94665	63,55755	7,007932
E2	3	7	51,86307	84,31229	95	91,65731	51,86307
E9	84	76	54,7337	42,39177	70,40597	53,80114	42,39177
E4	50	87	45,04936	9,56883	93,29523	77,24353	9,56883
E6	19	82	44,45347	23,00136	108,9312	94,83703	23,00136
E8	68	86	51,13164	26,34507	84,50444	67,87166	26,34507
E10	31	72	30,46492	14,69906	93,34881	79,2216	14,69906
E7	98	7	67,98121	93,40001	0	16,69019	0
E3	100	42	59,67505	70,31403	35,0571	20,1634	20,1634
E5	96	23	59,15047	79,79701	16,12452	2,75	2,75
E11	93	11	61,62611	87,21561	6,403124	12,0026	6,403124
E12	84	16	51,33983	78,01963	16,64332	11,60011	11,60011

## K-Means Algorithm Example Third Step

Elements	X	Y	DistToM1	DistToM2	DistToM3	DistToM4	Minim
E1	34	46	24,90984	38,28995	71,77221	62,30124	24,90984
E2	3	7	24,90984	87,54268	92,52162	92,52087	24,90984
E9	84	76	82,10055	33,91342	67,97978	49,88097	33,91342
E4	50	87	68,20924	6,412488	90,30089	74,01201	6,412488
E6	19	82	55,50225	31,43119	105,7414	92,46861	31,43119
E8	68	86	77,39832	18,40978	81,76338	64,20886	18,40978
E10	31	72	47,1858	21,22074	90,16235	76,87941	21,22074
E7	98	7	81,85658	87,65113	3,201562	20,53723	3,201562
E3	100	42	82,96083	62,84998	33,3054	16,41476	16,41476
E5	96	23	77,57899	73,46509	14,00893	4,807402	4,807402
E11	93	11	76,09533	81,60221	3,201562	16,00347	3,201562
E12	84	16	66,33626	72,81566	13,46291	14,42606	13,46291

We will now calculate the mean values for each of these clusters both for the X component and for the Y component, and get the following four means-elements: M1(18.5; 26.5), M2(50.4; 80.6), M3(95.5; 9), and M4(93.333333; 27). We check each element in order to identify which of the M1, M2, M3, or M4 mean-element it is closest to and assign it to the corresponding cluster (third step table). The third step table shows that so far, the dataset has been divided into the following clusters: {E1, E2}, {E9, E4, E6, E8, E10}, {E7, E11, E12}, and {E3, E5}.

We will now calculate the mean values for each of these clusters both for the X and Y components, and get the following four means-elements: M1(18.5; 26.5), M2(50.4; 80.6), M3(91.66; 11.33), and M4(98; 32.5). We check each element once again to identify which of the M1, M2, M3, or M4 mean-element it is closest to and assign it to the corresponding cluster. This is the purpose of the next table.



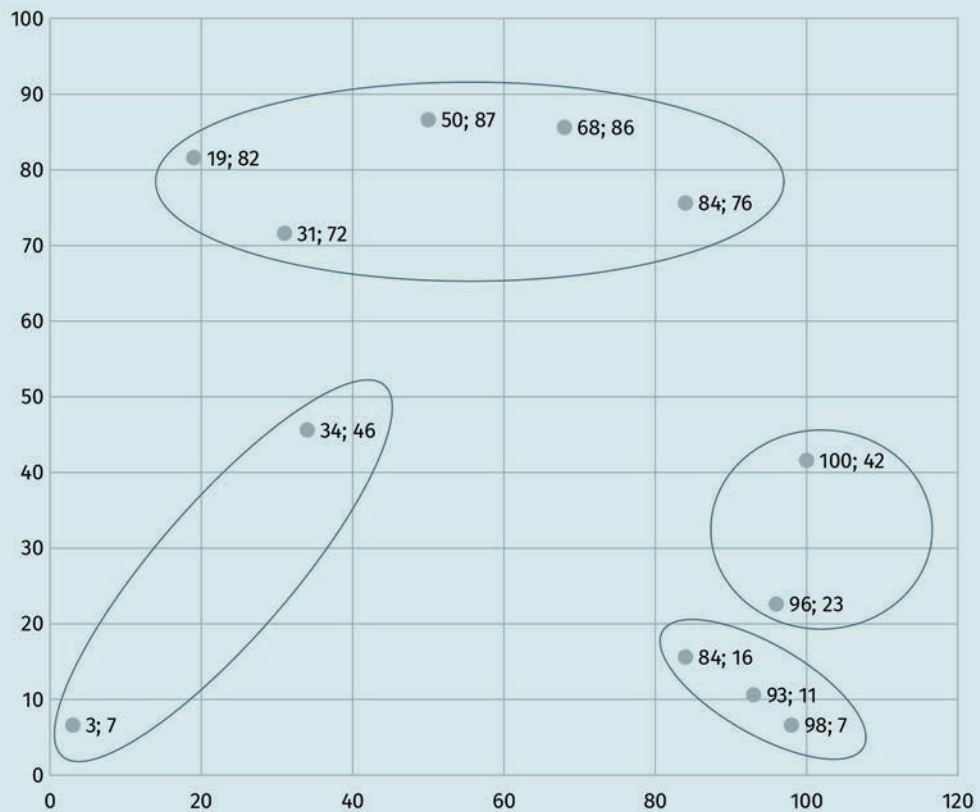
## Some Important Algorithms

## K-Means Algorithm Example Fourth Step

Elements	X	Y	DistToM1	DistToM2	DistToM3	DistToM4	Minim
E1	34	46	24,90984	38,28995	67,28464	65,40833	24,90984
E2	3	7	24,90984	87,54268	88,77249	98,36285	24,90984
E9	84	76	82,10055	33,91342	65,11955	45,69737	33,91342
E4	50	87	68,20924	6,412488	86,3803	72,62403	6,412488
E6	19	82	55,50225	31,43119	101,3618	93,22687	31,43119
E8	68	86	77,39832	18,40978	78,32766	61,33718	18,40978
E10	31	72	47,1858	21,22074	85,79562	77,77692	21,22074
E7	98	7	81,85658	87,65113	7,67391	25,5	7,67391
E3	100	42	82,96083	62,84998	31,77875	9,708244	9,708244
E5	96	23	77,57899	73,46509	12,44544	9,708244	9,708244
E11	93	11	76,09533	81,60221	1,374369	22,07374	1,374369
E12	84	16	66,33626	72,81566	8,975275	21,63909	8,975275

This time, we have the same clusters as in the previous step. Therefore, the same mean values will prevail, and the algorithm has to stop in acknowledgment that the current clusters are the final ones. These clusters are graphically represented below: {E1, E2}, {E3, E5}, {E4, E6, E8, E9, E10}, and {E7, E11, E12}.

### K-Means Algorithm Example Diagram



### Summary

This unit began with the presentation of classical searching and sorting algorithms. Searching and sorting are very important because they sometimes appear as the basic tasks of many algorithms. Moreover, an idea about the number of steps in these searching and sorting algorithms in the worst case scenario was given. This unit explained how to write basic regular expressions for pattern matching with the use of jq commands. The RSA algorithm and the k-means algorithm were also presented by this unit.

Some Important Algorithms

### Knowledge Check

---

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!



# Unit 4



## Correctness, Accuracy, and Completeness of Algorithms

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... how to write correctness proofs.
- ... the difference between total correctness and partial correctness.
- ... the practical side of a program's correctness.
- ... how to analyze the accuracy of approximate algorithms.

## 4. Correctness, Accuracy, and Completeness of Algorithms

### Introduction

We have all endured, at least for a few moments in our lives, the discomfort of having to scrutinize each and every line of one of our programs to find out why it was not working according to plan. This is simply because an algorithm or a program is of no use if it produces the wrong output for a given legitimate input.

There are also instances where programs and algorithms fail to produce the expected output simply because their execution is unable to reach an end point. This is why it is important to verify the correctness of programs and algorithms during their development in anticipation of their later testing.

The concepts covered in this unit are intended to give to the readers a sound understanding of how to ensure that their algorithms and programs are correct. These correctness concepts will be illustrated in this unit with suitable algorithms written in JavaScript.

### 4.1 Partial Correctness

The partial correctness of an algorithm is checked with the help of correctness proofs that are themselves based on the mathematical induction proof method. We first briefly present the mathematical induction proof method before explaining how to prove the partial correctness of an algorithm.

#### Mathematical Induction

The basic principle of mathematical induction states that in order to show that a statement is true for a sequence of objects, we first have to prove that it is true for the first object. We then have to assume that the statement is true for all the objects up to a certain one and prove that the statement is true for the next object. This can be expressed in the mathematical language as follows. In order to prove that a statement  $S$  is true for a sequence of objects  $O_0, O_1, O_2, O_3, O_4, \dots, O_{n-4}, O_{n-3}, O_{n-2}, O_{n-1}, O_n$ , we must first prove that  $S(O_0)$  is true. We must then prove that  $S(O_{k+1})$  is true when  $0 \leq k \leq n - 1$  and  $S(O_i)$  is assumed to be true for any  $i \leq k$ . For example, let us prove that the following formula is true for any whole number  $n$ :

$$1 + 2 + 3 + 4 + \dots + (n - 4) + (n - 3) + (n - 2) + (n - 1) + n = \frac{n(n + 1)}{2}$$

Let us start with the base case where  $n = 0$ . The corresponding statement is the following and is true:

Correctness, Accuracy, and Completeness of Algorithms

$$0 = \frac{0(0+1)}{2}$$

Let us now assume that the following statement is true for any  $i$  less or equal to  $k$  with  $k$  itself being between 0 and  $n - 1$ :

$$1 + 2 + 3 + 4 + \dots + (i - 4) + (i - 3) + (i - 2) + (i - 1) + i = \frac{i(i+1)}{2}$$

Let us now calculate the following sum:

$$1 + 2 + 3 + \dots + (k - 4) + (k - 3) + (k - 2) + (k - 1) + k + (k + 1)$$

The sum can also be written as follows:

$$(1 + 2 + 3 + \dots + (k - 4) + (k - 3) + (k - 2) + (k - 1) + k) + (k + 1)$$

This is also equal to the following expression because of the above assumption:

$$\frac{k(k+1)}{2} + (k+1)$$

Further calculations on the expression will lead to the following:

$$\frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

Thus, the equation confirms what we had to prove:

$$1 + 2 + 3 + 4 + \dots + (k - 3) + (k - 2) + (k - 1) + k + (k + 1) = \frac{(k+1)(k+1+1)}{2}$$

## Partial Correctness Proof of Iterative Algorithms

Correctness proofs of iterative algorithms are based on the use of the following three algorithmic features: preconditions, loop invariants, and post-conditions. Preconditions are a description of the criteria to be met by the inputs of an algorithm, while post-conditions are a description of the criteria to be met both by its output and by some of its key internal variables. As for loop invariants, they are a statement that must stay true for each and every instance of a loop while ensuring that the loop is contributing to the computation of the output.

The previous example on the sum of the first  $n$  natural numbers is implemented by the following Node.js [JavaScript program](#) to illustrate partial correctness concepts.



### Example of the Sum of Numbers

```

let readline = require('readline-sync');

let sum = function (n){
  if (n>=0) {
    let i = 0; let s = 0; for (let i=1; i<=n; i++){s = s + i;}
    return s;
  }
}

while (true){
  let ms = readline.question("A whole number please: ");
  let v = Number(ms);
  if ((ms!=='') && (Number.isInteger(v)) && (v>=0)){
    let m = parseInt(ms); let sm = sum(m); let o = 'first whole';
    console.log('The +(m+1)+ ' +o+' numbers added : '+sm+'\n');
  }
}

```

This JavaScript function is intended to calculate  $s$  as the sum of all the whole numbers from 0 to  $n$ . Let us now give a partial correctness proof, starting with the definition of the precondition, the invariant, and the post-condition. The precondition is simply that  $n$  must be an integer greater than or equal to zero. The loop invariant is the following:  $s$  is the sum of all the whole numbers between 0 and  $i$ . The post condition is that  $s$  is the sum of all the whole numbers between 0 and  $n$ .

The partial correctness proof itself consists in proving that

- the invariant is true at the initialization of the loop.
- if we assume that the invariant is true for all the values of the iterator  $i$  up to a given value  $k$ , then we have to show that the invariant remains true when the value of the iterator  $i$  becomes equal to  $k + 1$ .
- the algorithm ultimately yields the expected result after its final exit of the loop.

When the precondition is met, can we confirm that the loop invariant is true even before entering the loop? Yes. Indeed, before entering the loop, we have  $i = 0$ ,  $s = 0$ , and, in this case,  $s$  (whose value is equal to 0) is as a matter of fact, the sum of all the natural positive numbers between 0 and  $i$  (whose value is equal to 0).

Can we now prove that the loop invariant is true? Yes. We will do so with the help of the second step of the mathematical induction technique. We assume that the loop invariant is true for any value of  $i$  less than or equal to a given whole number  $k$ . We have to prove that the loop invariant remains true for  $i = k + 1$ , with  $k$  being of course less than  $n$ . When  $i$  exits the loop with the value  $i = k$ , the above assumption implies that  $s$  is the sum of all the natural numbers between 0 and  $k$ . When  $i$  re-enters the loop this time with the value  $i = k + 1$ , it will allow the assignment instruction  $s = s + i$  to

## Correctness, Accuracy, and Completeness of Algorithms

replace the  $s$  on its right side with its above indicated sum value. This replacement will update the value of  $s$  as follows:  $s = (1 + 2 + 3 + \dots + k) + k + 1$ . This clearly shows that  $s$  is the sum of all the natural numbers from 0 to  $k + 1$ , which is what we had to prove.

The final step of the partial correctness proof is to show that after the last iteration of the loop,  $s$  will yield the expected final result of the algorithm, which is the sum of all the natural numbers from 0 to  $n$ . This can be proven by the case of  $k = n - 1$  in the previous step of the proof that implies that the loop invariant is also true for  $k + 1$  which is  $n$ . In other words,  $s$  is the sum of all the natural numbers from 0 to  $n$ .

### Partial Correctness Proof of Recursive Algorithms

Recursive algorithms are not too different from iterative algorithms as far as partial correctness proofs are concerned, especially because these proofs are both based on the mathematical induction method. In fact, recursive algorithms also have preconditions, invariants, and post-conditions. The following main differential feature of recursive functions is however worth noting for the conceptualization of their correctness proofs.

Recursive functions always recall themselves into action with different parameters, except when they reach their base case. The partial correctness proof of a recursive algorithm therefore consists of the following three steps:

1. Proving that the base case of the invariant is true when the precondition is met
2. Proving that, if the invariant is true for all the parameter-values  $i$  less than or equal to a given value  $k$ , then the invariant is also true when the parameter-value  $i$  becomes equal to  $k + 1$
3. The algorithm ultimately yielding the expected result after its final recursive call. Let us, for example, prove the partial correctness of the Node.js [JavaScript `facto` function](#) written below for the calculation of the factorial of  $m$ .

### Example of the Recursive Version of Factorial

```

let readline = require('readline-sync');
let facto = function (m){
  if (m>=0){
    if (m===0){let f = 1;return f;}
    else {let f = m*facto(m-1); return f;}
  }
}

while (true){
  let ms = readline.question("A whole number please: ");
  let v = Number(ms);
  if ((ms!=='') && (Number.isInteger(v)) && (v>=0)){
    let n = parseInt(ms); let fn = facto(n);
    console.log(n + ' factorial is: ' + fn + '\n');
  }
}

```

The precondition for the recursive `facto` function is that the parameter `m` should be a whole number. The invariant is that, for any whole number `i` between 0 and `m`, the `facto(i)` function should yield the value `i!`, and the post-condition is that the `facto(m)` function should yield the value `m!`.

Let us start with the first step of the partial correctness proof. Is the invariant true for the base case when the precondition is met? Yes. Indeed, the base case is when `m===0`. In that case, the `facto(0)` function yields the value 1 which indeed is the value of `0!`. Assuming that `facto(i)` is equal to `i!` for any value of `i` less than or equal to a given `k` between 0 and `m - 1`, let us prove that `facto(k + 1) = (k + 1)!`. The fact that `k` is between 0 and `m - 1`, and that `m` is greater than or equal to 0 implies that `k + 1` is at least equal to 1. In other words, it is the `else` part of the above `if` condition that will be used for the calculation of `facto(k + 1)` as being equal to `(k + 1) · facto(k)`, which is ultimately equal to `(k + 1) · k!` because of the induction assumption on `facto(k)`. We have now shown that `facto(k + 1)` is equal to `(k + 1) · k!` which is clearly the same as `(k + 1)!` which we had to prove. The last step of the proof is to show that `facto(m) = m!` by simply referring to the previous step with `k` being equal to `m - 1`.

## 4.2 Total Correctness

Proving the total correctness of an algorithm consists of two parts, namely, its partial correctness proof, and its termination proof.

## Termination Proofs

The termination of an iterative algorithm can be proven by demonstrating that its loop is made up of a finite number of steps that are always going to come to an end. As for the termination of a recursive algorithm, it is proven by demonstrating that the parameters of that recursive function will be subjected to a finite number of variations that are always going to come at the end to the base case.

For example, the above iterative algorithm on the sum of the first  $n$  natural numbers has the following for loop: `for(let i=1; i<=n; i++)`. Can we prove that this algorithm will always terminate? Yes. This algorithm will always terminate because, when  $n$  is meeting the precondition ( $n \geq 0$ ), the loop of this algorithm will always come to an end after being executed  $n$  times ( $n$  is a finite number).

Let us likewise prove that the above `facto` function will also always terminate. It is obvious that the `facto` function will terminate when its parameter is equal to zero. Let us now prove that `facto(m)` will also always terminate for all the other whole numbers  $m$  that meet the precondition ( $m \geq 0$ ). For all such  $m$  values, the following successive calls are made for the calculation of `facto(m)`: `facto(m-1)`, `facto(m-2)`, `facto(m-3)`, and so on, up to `facto(0)`. This clearly shows that the recursive `facto` function will come to an end after recursively calling itself  $m$  times ( $m$  is a finite number).

## Total Correctness Proofs

An algorithm (or a program) is correct if and only if it is **totally correct**. For an algorithm to be declared totally correct, its termination (for all the inputs that are fulfilling its preconditions) and its partial correctness must be proven. The following table summarizes the relationship between algorithms' partial correctness, their termination, and their total correctness.

Total correctness  
This categorization entails partial correctness and termination.

Summary		
Partial Correctness	Termination	Total Correctness
False	False or Unclear or Unproven	False
False	True	False
True	False or Unclear or Unproven	False or Unclear or Unproven
True	True	True

The table above seems clear, but it does not really explain why it is crucial to differentiate partial correctness from total correctness: there are many algorithms for which the termination proof is unknown. For these algorithms, only the concept of partial correctness can be applied. Here are two examples of such algorithms whose termination proofs are unknown even though their partial correctness is proven.

Let's consider the problem of determining whether the Collatz sequence of a given strictly positive integer contains the value 1. For that sequence, the next number is equal to half of the current number when the current number is even, else the next number is the immediate successor of the triple of the current number. The Collatz sequence of a strictly positive integer always starts with that number and it ends when it lands on a repetitive or cyclic sub-sequence. Let's assume that we are looking for the Collatz sequence of 6. That sequence will start with 6, then it will go to 3, then 10, 5, followed by 16, 8, 4, 2, and finally 1, 4, 2, and 1. In other words,  $\text{Collatz}(6)$  is 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1. Similarly,  $\text{Collatz}(18)$  is equal to the sequence 18, 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1. It does not seem too difficult to write a naive JavaScript function to check whether the Collatz sequence of a given strictly positive integer contains the value 1.

All that is needed is to loop up to the last value which is equal to one and change the value for each iteration either to half of its previous value, or to the successor of its triple, depending on its parity. The [JavaScript program](#) below computes Collatz sequences containing the number 1.

## Collatz Sequence Code

```

let readline = require('readline-sync');

let collatz = function (n1){
  if (n1>0){
    let m = n1; let c = new Array(); c.push(m);
    while (m!=1) {
      if ((m%2)===0){m = m / 2;} else {m = (3*m) + 1;}
      c.push(m);
    }
    c.push(4,2,1);
    return c;
  }
}

while (true){
  let mes = "\nA strictly positive integer please: ";
  let s = readline.question(mes); let n = Number(s);
  if (((s!='') && (Number.isInteger(n)) && (n>=1))===true){
    n = parseInt(s); let cn = new Array(); cn = collatz(n);
    console.log("Collatz sequence of " + n);
    console.log(cn);
    console.log("Sequence's length: " + cn.length);
  }
}

```

The partial correctness of this JavaScript `collatz` function can be proven based on the following specifications. The precondition is that the input `n1` of the `collatz` function should be a strictly positive integer. The invariant stipulates that, with `lc` being considered as the length of the array `c`, this array is the sub-sequence of the first `lc` elements of the Collatz sequence of `n1`. The post-condition is that the array returned by the `collatz` function is the Collatz sequence of `n1`.

Is the invariant true for the base case when the pre-condition is met? Yes. Indeed, in that case, `lc=1` since `c` is only made up of `n1`, and `c` is the sub-sequence of the first element of the Collatz sequence of `n1` (`c=[n1]` and `lc =1`).

Let us now assume that, for all the arrays `c` with a length `lc` less than or equal to a given length `k*lc`, the array `c` is the sub-sequence of the first `lc` elements of the Collatz sequence of `n1`. Can we prove that, when the length of `c` becomes equal to `k*lc + 1`, `c` will also be the sub-sequence of the first `k*lc + 1` elements of the Collatz sequence of `n1`? Yes. The length of `c` can only become equal to `k*lc + 1` because it has added a new Collatz element to the already calculated Collatz sub-sequence with `lc` elements.

Does the above function return the Collatz sequence after exiting its loop? Certainly. The loop invariant is always fulfilled, up to when `m` becomes equal to one.

It is now time to look at the total correctness of the `collatz` JavaScript function by proving its termination now that we have proven its partial correctness. Does it always terminate for all the inputs that are fulfilling its precondition? Unfortunately, this is an open conjecture without an unanimously accepted mathematical proof judging by Paul Erdos's comment that "Mathematics is not yet ready for such problems" (Lagarias, 1985, p. 3). In other words, presently there is no termination proof for the `collatz` function. No one knows if the above `collatz` function is totally correct since it does not have a termination proof. All that can be said of this algorithm is that it is partially correct as proven above. The example of the twin primes problem can also help us to further understand why it is sometimes necessary to restrict oneself to proving the partial correctness of an algorithm instead of trying to prove its total correctness.

By definition, the numbers  $p$  and  $p + 2$  are twin primes if and only if  $p$  and  $p + 2$  are both primes. An apparently simple problem will be to find, for a given whole number  $n$ , the smallest possible value  $p$  greater than or equal to  $n$  such that  $p$  and  $p + 2$  are twin primes. For example, for  $n$  in  $\{0, 1, 2, 3\}$ ,  $p$  and  $p + 2$  are equal to 3 and 5, respectively; but for  $n$  in  $\{4, 5\}$ ,  $p$  and  $p + 2$  are respectively equal to 5 and 7. The naive algorithm for this twin primes problem will simply consist of starting an upwards loop on  $n$  in search of the prime number  $p$  for which  $p + 2$  is also prime. Such a naive algorithm is available in the [JavaScript program](#) below.



## Twin Primes Code

```
let readline = require('readline-sync');

let isPrime = function (m){
  let d = true;
  if (m===2){d = false;}
  if (m>=3){
    let i = 2; d = ((m % i) === 0);
    while ((d===false) && (i<m-1)){i++; d = ((m % i) === 0);}
  }
  return !d;
}

let twinP = function (m) {
  if (m>=0) {
    let p = m;
    while ((isPrime(p)===false) || (isPrime(p+2)===false)){
      p++;
    }
    return p;
  }
}

while (true){
  let ms = readline.question("A whole number please: ");
  let v = Number(ms);
  if ((ms!=='') && (Number.isInteger(v)) && (v>=0)){
    let n = parseInt(ms); let p = twinP(n);
    console.log("First twin primes from " + n);
    console.log(p + " and " + (p+2));
  }
}
```

The proof of the partial correctness of the above JavaScript program is twofold in the sense that one has to prove the partial correctness of the `isPrime` primality test function prior to proving the partial correctness of the `twinP` function itself. However, we can move directly to the partial correctness proof of the `twinP` function since the partial correctness proof of the `isPrime` primality test function was given as a self-check question in the previous section.

The precondition of the `twinP` function is that each of its inputs  $m$  should be a whole number. The loop invariant states that: For each value of the loop iterator  $i$ , there is no integer value  $p$  from  $m$  to  $i - 1$  such that  $p$  is prime and  $p + 2$  is also prime. As for the post-condition, it requires that the `twinP` function returns the value of the smallest possible prime number  $p$  greater than or equal to  $m$  such that  $p + 2$  is also prime.

Is the invariant true before the start of the loop? Yes. In this case,  $i = m$ , there is no single number from  $m$  to  $m - 1$  (is equal to  $i - 1$ ), and there can't be any prime number from  $m$  to  $i - 1$ .

Let us now suppose that for each value of the loop iterator  $i$  less than or equal to a given value  $k$  there is no integer value  $p$  from  $m$  to  $i - 1$  such that  $p$  is prime and  $p + 2$  is prime. Is it true that, when  $i = k + 1$ , there is no integer value  $p$  from  $m$  to  $k$  such that  $p$  is prime and  $p + 2$  is prime? Yes. Indeed, when  $i$  becomes equal to  $k + 1$ , it is because  $k$  is not prime or  $k + 2$  is not prime. The assumption of the induction implies that there is no integer value  $p$  from  $m$  to  $k - 1$  such that  $p$  is prime and  $p + 2$  is prime. We just saw now that at least one value between  $k$  and  $k + 2$  is not prime. This proves that there is no integer value  $p$  from  $m$  to  $k$  such that  $p$  is prime and  $p + 2$  is prime.

The last step of this partial correctness proof is to demonstrate that the algorithm yields the expected result at the final exit of its loop. This seems obvious because the loop will only stop when it finds the first value  $i$  such that `(isPrime(i)===true)` and `(isPrime(i+2)===true)`, and that is the expected output since  $i$  was initialized to  $m$  and the value of  $i$  is always increased by one.

It is now time to turn our attention to the termination proof of the `twinP` algorithm. Can we prove that the `while` loop of the `twinP` algorithm will always terminate? Unfortunately not because it has not yet been proven that there are infinitely many twin primes, and one is not sure whether, given a random extra-large integer  $m$ , the above algorithm will ultimately find a prime couple greater than  $m$ . This is why the JavaScript program sometimes seems to loop forever in the face of an extra-large input value  $m$ , and ultimately fails to output the expected result, even though it seems to work perfectly with smaller input values.

In summary, this is an algorithm that has been proven as partially correct but without any termination proof; sometimes, the algorithm does not seem to terminate. For example, this happened to us when we ran this program with the input value 1000000000000; and, after five minutes, we gave up on seeing the output that was still not there. This example illustrates a different perspective on how the definitions of partial correctness and total correctness can be tied down to the issue of termination. An algorithm is said to be totally correct if and only if, for all the inputs that fulfill its precondition, the algorithm always terminates and returns the correct output as defined by the post-condition. On the other hand, an algorithm is said to be partially correct if and only if, for all the inputs that fulfill its precondition, the algorithm returns the correct output as defined by the post-condition whenever it terminates.

### 4.3 Ensuring Correctness in Day-to-Day Programming

It is generally acknowledged that the writing of correctness proofs is perceived by most programmers as a difficult exercise that is only worthwhile for special software development projects such as the ones on the verification of security protocols. This might

## Correctness, Accuracy, and Completeness of Algorithms

explain why manual correctness proofs are rarely done in day-to-day programming. Instead, correctness is ensured in day-to-day programming by different mechanisms both during and after coding.

### Ensuring Correctness during Coding

Programmers can ensure the correctness of their code by making use of exceptions handling and assertions mechanisms, by modularizing their code, for example, through the use of existing libraries, and by programming in teams. Code analysis tools are also very valuable for the detection of errors.

#### Modules and libraries

It is easier to detect errors in a program that is divided into modules compared to a program that presents itself in a single block, especially for modules from tried and tested libraries. This is the case because modularization allows programmers to isolate problematic modules and focus their energy on the mitigation of their errors. For example, [here](#) is a main program that makes use of an already written `isPerfect` Boolean function to test whether given numbers are perfect (equal to the sum of their divisors excluding themselves, e.g., 6 since  $6 = 3 + 2 + 1$ ). This example assumes that `isPerfect` is from the `number-isperfect` library that can be installed with the `npm install number-isperfect` command.

#### Perfect Numbers Caller Example

```
const isPerfect = require('number-isperfect');
let readline = require('readline-sync');
while (true){
  let num = parseInt(readline.question("\nA whole number please: "));
  let pf = isPerfect(num); console.log(num + ' is perfect: '+pf+'\n');
}
```

In this code, the loop of the main program is only made up of three instructions:

1. The first one collects the input from the user and stores it in the `num` variable.
2. The second one calls the `isPerfect(num)` function and stores its result in the `pf` variable.
3. The last one displays the `pf` result on the screen.

The actual task of checking whether a number is perfect is not done by this main program but by the `isPerfect()` function. In fact, the main program simply divides the job to be done into relevant sub-tasks and coordinates their interactions. The `isPerfect()` function's sole role is to check if a given number is perfect.

### Exceptions and assertions

Modern high-level programming languages allow programmers to test an expected precondition for the purpose of handling any related exception. There are also instances where programmers make use of assertion instructions to test an expected precondition and ultimately halt the program for any negative test. Exception handling and assertions are available in JavaScript and Node.js. These two mechanisms are also an attempt to counter the Garbage In Garbage Out (**GIGO**). This is, for instance, the case of a program that might tell you that 11.5 is a prime number simply because it has converted its input from the console into an integer without first rejecting all non-integer values. Exceptions use keywords such as `try` and `catch` to avoid certain instructions to be executed with wrong values, and assertions use the `assert` keyword to ensure that a given condition is fulfilled prior to the execution of certain instructions. Let's see how to use exceptions and assertions to improve our primality test algorithm. The code below can be found [here](#).

#### GIGO

This concept describes the challenge of programs outputting the wrong answers because they have been fed with inappropriate inputs.

**Primality Test Example with Exceptions (First Version)**

```
let readline = require('readline-sync');

let isPrime = function (s){
  if ((s==='') || (Number.isInteger(Number(s))===false)){
    return new Error("Sorry! Only integers please!");
  }
  else {
    let m = Number(s);
    if (m<0){
      return new Error("Sorry! No negatives please!");
    }
    else {
      let d = true;
      if (m===2) {d = false;}
      if (m>=3){
        let i = 2;
        d = ((m % i) === 0);
        while ((d===false) && (i<m-1)){
          i++;
          d = ((m % i) === 0);
        }
      }
      return !d;
    }
  }
}

while (true){
  let ns = readline.question("\nA whole number please: ");
  let pri = isPrime(ns);
  if (pri instanceof Error){
    console.log(ns + ' is prime: ' , pri, '\n');
  }
  else{
    console.log(ns + ' is prime: ' + pri + '\n');
  }
}
```

A [second version](#) of the program can be found below with `try`, `throw`, and `catch` instructions.

## Primality Test Example with Exceptions (Second Version)

```

let readline = require('readline-sync');

let isPrime = function (s){
  try {
    if ((s==='') || (Number.isInteger(Number(s))===false)){
      throw new Error("Sorry! Only integers please!");
    }
    else {
      let mn = Number(s);
      if (mn<0){
        throw new Error("Sorry! No negatives please!");
      }
    }
  }

  catch(err) {return(err);}

  let m = Number(s);
  let d = true;
  if (m===2) {d = false;}
  if (m>=3){
    let i = 2;
    d = ((m % i) === 0);
    while ((d===false) && (i<m-1)){
      i++;
      d = ((m % i) === 0);
    }
  }
  return !d;
}

while (true){
  let ns = readline.question("\nA whole number please: ");
  let pri = isPrime(ns);
  if (pri instanceof Error){
    console.log(ns + ' is prime: ' , pri, '\n');
  }
  else{
    console.log(ns + ' is prime: ' + pri + '\n');
  }
}

```

An [equivalent program](#) can be found below where exception handling is replaced by assertion instructions.

### Primality Test Example with Assertions

```
let readline = require('readline-sync');

let isPrime = function(s) {
  let assert = require('assert');
  let er = 'Sorry! Only integers please!';
  assert(((s!='') && (Number.isInteger(Number(s))))===true,er);
  let m = parseInt(s);
  assert(m>=0,"Sorry! No negatives please!");
  let d = true;
  if (m===2) {d = false;}
  if (m>=3) {
    let i = 2;
    d = ((m % i) === 0);
    while ((d===false) && (i<m-1)){
      i++;
      d = ((m % i) === 0);
    }
  }
  return !d;
}

while (true){
  let ns = readline.question("\nA whole number please: ");
  let pri = isPrime(ns);
  console.log(ns + ' is prime: ' + pri);
}
```

### Code analysis tools

Code analysis automated tools are used by programmers to quickly detect programming errors instead of spending long hours manually debugging. Such error identification is relatively easy in the first version of a program but becomes increasingly difficult with the changes made by its newer versions. This tool is becoming essential since a growing number of programs are being “transpiled” from one language to another in order to keep their qualities in newer execution environments.

We will now illustrate the use of code analysis tools by executing Jshint on the following [JavaScript program](#) (Jshint is a JavaScript code analysis tool that can be installed by the `npm install jshint` command).



### Primality Test Example for Jshint Demo

```

let readline = require('readline-sync');
while (true){
  let n = readline.question("An integer please: "); let d = true;
  if (n=2){d = false;}
  if (n>2){
    let i=2; d=((n % i)===0);
    while((d===false)&&(i<n-1)){i++; d =((n%i)===0);}
  }
  console.log(n + ' is prime: ' + (d!) + '\n');
}

```

Trying to execute the above JavaScript program with the `node` command (`node nameOf-program.js`) will output the following error.

```

while((d===false)&&(i<n-1){i++; d=((n%i)===0);}
                                     ^

```

SyntaxError: Unexpected token '{'

However, running the same code with the Jshint code analysis tool (`jshint nameOf-program.js`) will output the following errors.

```

line 1, col 1, 'let' is available in ES6 (use 'esversion: 6') or Mozilla
JS extensions (use moz).
line 3, col 3, 'let' is available in ES6 (use 'esversion: 6') or Mozilla
JS extensions (use moz).
line 3, col 53, 'let' is available in ES6 (use 'esversion: 6') or Mozilla
JS extensions (use moz).
line 4, col 8, Expected a conditional expression and instead saw an assignment.
line 6, col 5, 'let' is available in ES6 (use 'esversion: 6') or Mozilla
JS extensions (use moz).
line 7, col 31, Expected ')' to match '(' from line 7 and instead saw '{'.
line 9, col 37, Expected an operator and instead saw '!'.
line 9, col 47, Unrecoverable syntax error. (90% scanned).

```

### Team programming

The traditional image of a programmer is a computing geek always sitting alone in their corner staring at the computer or hitting the keyboard in search of a solution for a computing problem. It is what is known as solo programming as opposed to team programming where many programmers work together on the same program. One form of collaboration is pair programming. According to Saltz and Shamshurin (2017), pair programming improves code quality by approximately 15 percent, and its significant increases in code development time usually lead to important decreases in debugging and testing times. The same authors also report that pair programming significantly enhan-

## Correctness, Accuracy, and Completeness of Algorithms

ces the thinking abilities, programming knowledge, and communication skills of programmers whose job satisfaction and team spirit levels are likewise meaningfully increased.

### Ensuring Correctness after Coding

In day-to-day programming, it is common practice to enforce programs' correctness after coding through code reviews and through testing.

#### Code reviews

Once a code has been written, it is not unusual to ask a fellow programmer to review it in order to assess its quality, just like research articles are peer-reviewed in the publication process.

According to Alami et al. (2019), code review is currently extensively practiced in the software industry. The same study presents the following benefits of code review, based on the case study of 21 code reviewers from four open-source communities (Allura, CKAN, FOSSASIA, and Kernel):

- Negative feedback constitutes the main quality assurance mechanism of code reviews.
- Negative feedback from a code review is an opportunity to learn and become a better coder.
- Code reviewers are passion driven and always on the quest for excellence and quality.
- The primary trading currencies in the code review world are reputation and status.

#### Testing

It is unthinkable that a program could be put into use without having been tested. Testing consists of running a program with various inputs or test cases in order to assess the behavior of the program compared to its requirements. Testing happens at different levels such as unit testing, integration testing, and system testing. Unit testing is restricted to individual units or modules, integration testing checks the interactivity of these units, and system testing extends to the assessment of the behavior of the system as a whole. We can reasonably say that testing contributes to the improvement of the quality of a program, but what about the quality of the test cases themselves? According to Kochhar et al. (2019), there are five main test case characteristics that software testers use as guidelines for the design of quality test cases.

Their study confirmed these characteristics by surveying 21 respondents and interviewing 261 software practitioners both from the open source and from the proprietary software industry. The participants of this study were distributed over 29 countries; China and the USA had the highest number of participants. Moreover, these respondents were from reputable organizations such as Google, Facebook, Apache, and Microsoft. To examine the perceptions of the respondents on the quality characteristics of test cases, 26 Likert scale items were created. These items were classified into the following five

themes: the content of test cases, their size and complexity, coverage, maintainability, and their bug detection requirements. Readers are invited to refer to Kochhar et al. (2019) for a closer look at the above listed quality requirements for test cases.

In the meantime, we present an example of how to automate test cases for a NodeJS program using the JEST testing tool. It is assumed that you have installed JEST on your machine, for instance, with the `npm install --save-dev jest` command and that you have added the following object in the list of objects in your `package.json` file inside your home folder.

#### Jest Object for package.json File

```
"scripts": {  
  "test": "jest"  
}
```

The NodeJS function to be tested is the `isPrime` primality test function. However, we had to modify it slightly as follows and save it in a dedicated file named `isPrime1.js` for its testing by JEST.

#### First Primality Test Example for Jest Demo

```
let isPrime1 = function(s){  
  let assert = require('assert');  
  let er = 'Sorry! Only integers please!'  
  assert(((s!='') && (Number.isInteger(Number(s))))===true,er);  
  let m = parseInt(s);  
  assert(m>=0,"Sorry! No negatives please!");  
  let d = true;  
  if (m===2){d = false;}  
  if (m>=3){  
    let i = 2; d = ((m % i) === 0);  
    while((d===false) && (i<m-1)){i++; d=((m % i) === 0);}  
  }  
  return !d;  
}  
module.exports = isPrime1;
```

## Correctness, Accuracy, and Completeness of Algorithms

The test cases themselves can be found in the `isPrime1.test.js` below written file where for each integer `i` between 0 and 16, the expected primality `r` of the integer `i` is compared to its calculated primality `t` by the `isPrime1()` function. The `isPrime1()` function will fail the test whenever `t` and `r` are different, and this automated testing process is enacted by the `npm run test` command.

## First Set of Jest Test Cases for the Primality Test Program

```
const isPrime1 = require('./isPrime1');

for (let i=0; i<17; i++){
  let r = false;
  if ((i===2)||(i===3)||(i===5)||(i===7)||(i===11)||(i===13)){
    r = true;
  }
  let t = isPrime1(i.toString());
  test('Check primality of ' + i + ' as ' + r, () => {
    expect(t).toBe(r);
  });
}
```

It is also possible to test whether a given test case can trigger the expected exceptions as visible in the updated NodeJS program below (`isPrime2.js`).

## Second Primality Test Example for Jest Demo

```
let isPrime2 = function (s) {
  if ((s==='')||(Number.isInteger(Number(s)))===false){
    throw new Error('Sorry Only Integers');
  }
  else{
    m=Number(s)
    if (m<0){
      throw new Error('Sorry Positives Only');
    }
  }
  let d = true;
  if (m===2) {d = false;}
  if (m>=3){
    let i = 2; d = ((m % i) === 0);
    while ((d===false) && (i<m-1)){i++; d = ((m % i) === 0);}
  }
  return !d;
}
module.exports = isPrime2;
```

The following code (`isPrime2.test.js`) contains three test cases (abc, 10.3, and -7) for the testing of the handling of exceptions by the program above.

#### Second Set of Jest Test Cases for the Primality Test Program

```
const isPrime2 = require('./isPrime2');

test('throws for abc as non integer', () => {
  expect(() => {
    isPrime2('abc');
  }).toThrowError(new Error ('Sorry Only Integers'));
});

test('throws for 10.3 as non integer', () => {
  expect(() => {
    isPrime2('10.3');
  }).toThrowError(new Error ('Sorry Only Integers'));
});

test('throws for -7 as negative integer', () => {
  expect(() => {
    isPrime2('-7');
  }).toThrowError(new Error ('Sorry Positives Only'));
})
```

The automated testing process is enacted by the `npm run test` command.

We end this section by presenting a different side of testing where the behavior of a program significantly differs from what was expected because its input is a code instead of data. This is known as code injection. We show an example of SQL injection in a NodeJs code that interacts with a MySQL Database. Install MySQL for the execution of the following programs.

The [following program](#) creates a `Credentials` database assuming that the user name of the administrator of the MySQL server is `root`, and they use `pswd!@#$$%^` as the password in MySQL. Successfully running this program on the command line with NodeJs leads to the creation of the `Credentials` database in MySQL.

### Code for the Creation of a MySQL Database

```
let mysql = require('mysql');

let con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "pswd!@#$%^"
});

con.connect(function(err){
  if (err) throw err;
  console.log("Connected!");
  con.query("CREATE DATABASE Credentials", function (err, result){
    if (err) throw err;
    console.log("Database created");
  });
});
```

The following code creates the `login` table in the above referred `Credentials` database.

This `login` table is made up of two fields that are both strings, the email address field (`email`), and the password field (`pswd`). The email address field is the primary key. This program must be run with NodeJS on the command line after execution of the one above in order for this `login` table to be created in the `Credentials` database with its two `email` and `pswd` fields.

### Code for the Creation of a MySQL table

```
let mysql = require('mysql');

let con = mysql.createConnection({
  host: "localhost", user: "root", password: "pswd!@#$%^",
  database: "Credentials"
});

con.connect(function(err){
  if (err) throw err;
  console.log("Connected!");
  let sql = "CREATE TABLE login (email VARCHAR(255), pswd VARCHAR(255), PRIMARY KEY (email))";
  con.query(sql, function (err, result){
    if (err) throw err;
    console.log("Table created");
  });
});
```

The coding and the execution of the [following program](#) will allow a user to input their email address and password in order for them to be stored in the `login` table in the `Credentials` database. Let's suppose that one user has entered `Algo2020[.]example[.]com` as a username and `yu2?&!me` as a password; another user has entered `Correct100[.]example[.]com` as their username and `me241&!u` as the password.



### Code for the Storage of a MySQL Record

```
let readlineSync = require('readline-sync');
let emad = readlineSync.question('Email address please: ');
let secret = readlineSync.question('Your code please: ', {
  hideEchoBack: true
});
let mysql = require('mysql');
let con = mysql.createConnection({
  host: "localhost", user: "root", password: "pswd!@#$$%^",
  database: "Credentials"
});
con.connect(function(err) {
  if (err) throw err; console.log("Database connected!");
  let sqlq = "INSERT INTO login (email, pswd) VALUES ('" + emad +
  "','" + secret + "')";
  con.query(sqlq, function (err, result) {
    if (err) throw err;
    console.log(result.affectedRows + " record inserted");
  });
});
```

This final JavaScript will allow a user that has forgotten their password to see that password after inputting their email address.

## Code for the Querying of a MySQL Record

```

let readlineSync = require('readline-sync');
let emad = readlineSync.question('Email address please: ');
let mysql = require('mysql');
let con = mysql.createConnection({
  host: "localhost", user: "root", password: "pswd!@#$$%^",
  database: "Credentials"
});
con.connect(function(err) {
  if (err) throw err;
  const sql1 = `SELECT * FROM login WHERE email = ${emad}`;
  con.query(sql1, function (err, result) {
    if (err) {
      const sql2 = `SELECT * FROM login WHERE email = "${emad}"`;
      con.query(sql2, function (err, result) {
        if (err) throw err; console.log(result);
      });
    }
    else {console.log(result);}
  });
});

```

Readers can now have a first-hand experience with SQL injection by executing the program above using the following usernames or email addresses as inputs:

- Email address: Algo2020[@]example[.]com. The above program correctly outputs the password of the user as `yu2?&!me`. Everything seems normal since the program is behaving as expected.
- Email address: Correct100[@]example[.]com. The above program correctly outputs the password of the user as `me241&!u`. Here, everything also seems normal since the program is behaving as expected.
- Email address: "" OR 1=1. The above program displays the email addresses and passwords of each and every user in the `login` table. Something is terribly wrong with the program that has now suffered an SQL injection attack where a user has input a code instead of a bona fide email address.

## 4.4 Accuracy, Approximation, and Error Analysis

This section is dedicated to nondeterministic algorithms as opposed to the deterministic ones that we have been dealing with. For a given input, a deterministic algorithm will always yield the same output no matter how many times the algorithm is executed. But for a nondeterministic algorithm, it is possible for the outcome of the execution of the algorithm to change from one execution to another one, still with the same input, and sometimes with the wrong output.

## Rationale and Consequences

The main idea behind the design of nondeterministic algorithms is fourfold:

1. Random choices can sometimes lead to the solution.
2. Many problems do not require to be solved with a one hundred percent level of accuracy.
3. The use of approximations simplifies solutions.
4. A fast approximate solution to a problem is sometimes preferable to a delayed exact answer for the same problem.

The random and approximate nature of **nondeterministic algorithms** implies that those algorithms do not always give the correct answer and are prone to errors. Nevertheless, they are useful as long as their level of accuracy is acceptable. The next section is an illustration of these concepts using the example of the middle rank problem. In that example, simple probability calculations will be used for the estimation of the level of accuracy of an approximate algorithm.

Nondeterministic algorithms  
These algorithms tend to yield different output values when executed many times with the same input value.

## Approximate Algorithm for the Middle Rank Problem

The middle rank problem assumes that there is a sequence  $U$  of  $n$  unsorted distinct numbers, and we simply want to identify any element  $S[k]$  with  $S$  being the sorted version of  $U$  and with  $k$  fulfilling the following condition.

$$k \in \left[ \frac{(1 - \epsilon)n}{2}, \frac{(1 + \epsilon)n}{2} \right], 0 \leq \epsilon \leq \frac{1}{2}$$

Let's suppose that the unsorted sequence is made up of the following 16 numbers. Assuming that  $\epsilon = \frac{1}{8}$ , it is required for  $k$  to be equal to seven, eight, or nine. In other words, we are simply looking for the seventh, eighth, or ninth element in the sorted version of the following sequence of numbers: 51, 54, and 59.

26	71	65	43	60	95	54	72	74	42	51	85	49	46	16	59
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

This problem can be solved by first sorting the array and thereafter getting access to the seventh, eighth, or ninth element of the sorted array. But sorting takes times. So why not try our luck with any element from any position in the above array and calculate its ranking? We might be lucky to land with a number whose ranking is between 7 and 9; for example, what is the ranking of the element in position 10 whose value is 42? This can be calculated by first assuming that 42 is the smallest element of the array, and its rank is thus equal to 1; by comparing each element of the array with 42 and incrementing the ranking of 42 each time, an element is less than 42.

This gives a ranking of 2 for the value 42, which does not belong to the required interval between 7 and 9. Why not try our luck one last time with the 16th element of the sequence, 59? The ranking of 59 is 8: Bingo! We found our number in the required range after two random choices only and without sorting the array.

This algorithm is formalized in the following JavaScript program that assumes that the numbers of the unsorted sequence are stored line by line in a text file whose name is input by the user together with the value of  $\epsilon$ . Before going into the details of the JavaScript program itself, let us calculate a few mid rank intervals for the different values of epsilon that we will use for the testing of our program, assuming that there are 100 different numbers in the text file ( $n = 100$ ).

#### Mid Rank Intervals for Epsilon

$\epsilon$	Formula of lower bound for the rank	Formula of higher bound for the rank	Value of lower bound (n=100)	Value of higher bound (n=100)
$\frac{1}{2} = 0.5$	$\frac{n}{2}$	$\frac{3n}{2}$	25	75
$\frac{1}{4} = 0.25$	$\frac{3n}{8}$	$\frac{5n}{8}$	37,5	62,5
$\frac{1}{8} = 0.125$	$\frac{7n}{16}$	$\frac{9n}{16}$	43,75	56,25
$\frac{1}{16} = 0.0625$	$\frac{15n}{32}$	$\frac{17n}{32}$	46,875	53,125
$\frac{1}{32} = 0.03125$	$\frac{31n}{64}$	$\frac{33n}{64}$	48,4375	51,5625

The [following program](#) uses the `n-readlines` module that should be installed on your machine using the `npm install n-readlines` command. This module contains the necessary functions for the line-by-line reading of a text file. Once the numbers are read from the text file, they are immediately transferred into an array for their processing by the algorithm designed for the mid rank problem.

## Mid Rank Problem Algorithm Example

```

let read = require('readline-sync');
const lineReader = require('n-readlines');

let midRangeV0 = function (Ar,epsi) {
  let n = Ar.length; let lb = ((1-epsi)*n)/2;
  let hb = ((1+epsi)*n)/2; let r = Math.floor(Math.random()*n);
  let x = Ar[r]; let k = 1;
  for (let i=0; i<n; i++){if(Ar[i] < x){k++;}}
  if ((k >= lb) && (k<=hb)){return k;} else{return undefined;}
}

let midRangeV = function(Ar,epsi,nmaxt){
  let r = midRangeV0(Ar,epsi); let c = 1;
  while((r===undefined) || (c<nmaxt)){r=midRangeV0(Ar,epsi); c++;}
  return r;
}

let readFile = function(tfn){
  const lrdr = new lineReader(tfn); let line; let s = "";
  let Ar = new Array(); Ar = [];
  while (line = lrdr.next()){
    s = line.toString('ascii');
    if (s[s.length-1]==='\n'){Ar.push(Number((s.slice(0, -1))));}
    else {Ar.push(Number(s));}
  }
  return Ar;
}

let tf=read.question('Text file name please: ');

while (true){
  let epss=read.question('\nStrictly positive epsilon please: ');
  if ((epss!=='' ) && (Number(epss)>0)){
    let A = new Array(); A = readFile(tf); let lNo = A.length;
    let eps = Number(epss); let lbo = ((1-eps)*lNo)/2;
    let hbo = ((1+eps)*lNo)/2;
    console.log('Epsilon: '+eps+' Mid range: ['+lbo+' , '+hbo+']);
    let r1 = midRangeV0(A,eps);
    let ms = 'Rank and value found after attempt ';
    console.log(ms + ' 1 : ' + r1 + ' , ' + A[r1]);
    if (r1 === undefined){
      let r2 = midRangeV(A,eps,10);
      console.log(ms + ' 10 or even less : '+r2+' , '+A[r2]);
    }
  }
}
}

```

In this program, the `midRangeV0` function is an implementation of the algorithm that randomly picks up a number and reports it as a successful answer if its rank is in the required mid-range while acknowledging a failure if that is not the case. The main pro-

gram above always starts by trying its luck with a first attempt of the `midRangeV0` function (`let r1 = midRangeV0(A,eps)`) with the hope of it being successful. We have experimented with this program using the following 100 values stored in a text file with one value per line.

8722, 3420, 4548, 5637, 5927, 5078, 3739, 6338, 8362, 4617, 3980, 2680, 6264, 8329, 1815, 6119, 9179, 8015, 9235, 3161, 8453, 8469, 3917, 2944, 7502, 6514, 4025, 8678, 8820, 6988, 7214, 6463, 7506, 2042, 7176, 3762, 9577, 5902, 5109, 4441, 9127, 2271, 3726, 2018, 8272, 9629, 8693, 5772, 3185, 6663, 3644, 7668, 1667, 3757, 2969, 6626, 6074, 3861, 2913, 7566, 2257, 3705, 1353, 3868, 9133, 8921, 8368, 8307, 4331, 1092, 6495, 8175, 6472, 9238, 8987, 2838, 1012, 6521, 2779, 2028, 3677, 7394, 2582, 2978, 7930, 7274, 2272, 4015, 3678, 6991, 1962, 9652, 5097, 3277, 4532, 6607, 9203, 4945, 5708, 9954

We executed this program four times with an epsilon value  $\epsilon = \frac{1}{2} = 0.5$  for a required mid-range of [25, 75] and got the following different results.

Execution	Result
First	Rank and value found after first attempt: undefined, undefined Rank & value found after more attempts: 68 , 4331
Second	Rank and value found after first attempt: 31 , 6463
Third	Rank and value found after first attempt: 72 , 6472
Fourth	Rank and value found after first attempt: undefined, undefined Rank & value found after more attempts: 60 , 2257

The table above shows that, out of four executions, the first attempt of the `midRangeV0` function failed to do its job twice (first and fourth execution) in trying to find a mid-range value ranked between the 25th and the 75th rank, but it yielded different good answers when it succeeded (second and third execution). The main reason behind this nondeterministic behavior is the use of a random number by the `midRangeV0` function. We then executed the same program another four times but with an epsilon value  $\epsilon = \frac{1}{8} = 0.125$  for a required mid-range of ranks in the [43.75, 56.25] interval. This time, we got the following different results.

Execution	Result
First	Rank and value found after first attempt: undefined, undefined Rank & value found after more attempts: 51 , 7668



## Correctness, Accuracy, and Completeness of Algorithms

Execution	Result
Second	Rank and value found after first attempt: undefined, undefined Rank & value found after more attempts: 47 , 5772
Third	Rank and value found after first attempt: 49 , 6663
Fourth	Rank and value found after first attempt: undefined, undefined Rank & value found after more attempts: 56 , 6074

The table above shows that, out of four executions, the first attempt of the `midRangeV0` function failed to do its job three times (first, second, and fourth execution) in trying to find a mid-range value ranked between the 44th and the 56th rank, but it yielded a good answer when it succeeded (third execution). The success rate of the `midRangeV0` function is 50 percent for the first table, but it is now 25 percent for the second. It is important to note that each successful answer of the `midRangeV0` function is a correct answer, but each failure of the same function is a wrong answer.

In this case, the terms' success rate and accuracy levels are equivalent, and it is only natural for us to want to know the value of the estimated accuracy level of the `midRangeV0` function. Is it 50 percent as experienced by the results of the first table? Is it 25 percent as experienced by the results of the second? Or does it depend on certain parameters? The rest of the section is dedicated to these questions.

What are the chances for the `midRangeV0` function succeeding? It succeeds when, at the end of the iteration of  $i$ ,  $k$  ends with a value that fulfills the following condition.

$$k \in \left[ \frac{(1 - \epsilon)n}{2}, \frac{(1 + \epsilon)n}{2} \right]$$

How many possible values are there in this interval?  $\epsilon n + 1$  as shown below

$$\frac{(1 + \epsilon)n}{2} - \frac{(1 - \epsilon)n}{2} + 1 = \frac{(1 + \epsilon)n - (1 - \epsilon)n}{2} + 1 = \frac{(n + \epsilon n) - (n - \epsilon n)}{2} + 1 = \frac{(2\epsilon n)}{2} + 1$$

How many possible values can  $k$  have in general?  $n$  values, since  $k$  can end up with any value between 1 and  $n$ .

What are the chances of the `midRangeV0` function to succeed? The answer to this question is the following in accordance with basic probability laws:  $\frac{\epsilon n + 1}{n} \simeq \epsilon$ . This translates into the following estimated accuracy levels of the `midRangeV0` function when  $n$  is equal to 100.



$\epsilon$	Estimated Accuracy Levels of the <code>midRangeV0</code> Function When $n = 100$	Estimated Failure Rate
$\frac{1}{2} = 0.5$	50%	50%
$\frac{1}{4} = 0.25$	25%	75%
$\frac{1}{8} = 0.125$	12.5%	87.5%
$\frac{1}{16} = 0,0625$	6.25%	93.75
$\frac{1}{32} = 0,03125$	3.125%	96.875

The table above shows that the estimated accuracy level of the `midRangeV0` function is quite low. The best that algorithm can do is to give us a fifty-fifty chance between a correct answer and an erroneous one when we execute it. Worse, there is almost no chance of getting a good answer when the value of epsilon becomes smaller. But what happens when the `midRangeV0` function is executed a few times, let's say  $c$  times? The probability of the `midRangeV0` function to fail is equal to  $(1 - \epsilon)$  since its probability to succeed is equal to  $\epsilon$ . Therefore, the probability of the `midRangeV0` function to fail  $c$  times is equal to  $(1 - \epsilon)^c$ , and its probability to succeed after  $c$  times is equal to  $1 - (1 - \epsilon)^c$ . This translates into the following accuracy levels of a `midRangeV` function that tries the `midRange0` function  $c$  times with  $n$  equal to 100 and  $c$  equal to 10.

$\epsilon$	Estimated failure rate of <code>mid-Range0</code> : $(1 - \epsilon)$	Estimated failure rate of 10 <code>mid-Range0</code> attempts: $(1 - \epsilon)^{10}$	Estimated success rate of 10 <code>midRange0</code> attempts: $1 - (1 - \epsilon)^{10}$
$\frac{1}{2} = 0.5$	0.5	0,00098 = 0.098%	0,99902 = 99.902%

## Correctness, Accuracy, and Completeness of Algorithms

$\epsilon$	Estimated failure rate of <code>midRange0</code> : $(1 - \epsilon)$	Estimated failure rate of 10 <code>midRange0</code> attempts: $(1 - \epsilon)^{10}$	Estimated success rate of 10 <code>midRange0</code> attempts: $1 - (1 - \epsilon)^{10}$
$\frac{1}{4} = 0.25$	0.75	0,05631 = 5.631%	0,94369 = 94.369%
$\frac{1}{8} = 0.125$	0.875	0,26308 = 26.308%	0,73692 = 73.692%
$\frac{1}{16} = 0,0625$	0.9375	0,52446 = 52.446%	0,47554 = 47.554%
$\frac{1}{32} = 0,03125$	0.96875	0,72798 = 72.798%	0,27202 = 27.202%

It is important to note that, even when the `midRangeV` function runs the `midRange0` function a few number of times, it is still achieved in a linear time. In fact, it is still faster by far than the time that it would have taken to sort the array, which in the worst case is quadratic.

### Summary

This unit presented mathematical induction as the cornerstone of partial correctness proofs both for iterative algorithms and for recursive functions where we have to show that the invariant is always true. It covered termination proofs as the additional verification step beyond the partial correctness proof in order to determine total correctness. Day-to-day programming correctness checking mechanisms, such as testing and code reviews, together with other approaches such as the use of code analysis tools, libraries, modules, assertions, and exceptions, were reviewed. The end of the unit was dedicated to the presentation of an example of an approximate and randomized algorithm. The presentation of this algorithm includes key details about the probabilistic calculation of its accuracy in an effort to explain the rationale behind such nondeterministic algorithms despite the fact that they sometimes yield erroneous results.

**Knowledge Check**

---

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

# Unit 5



## Computability

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... how to compute an algorithm in a given model of computation.
- ... the specification and characteristics of the halting problem.
- ... key details about some well-known undecidable problems.

# 5. Computability

## Introduction

Today's computing machinery seems so effective judging by its overwhelming success both for traditional applications and the newest ones. In the midst of such an impressive computing power, it is a legitimate question to ask whether a computational problem that cannot be solved by a computer exists. This is the question at the core of this unit. But that question cannot be answered without a clear understanding of the different models of computation.

Classical models of computation are presented first. Thereafter, the specification and the characteristics of the halting problem are described as an introduction to the concepts of uncomputability and undecidability. The last section of the unit is dedicated to the presentation of some well-known undecidable problems.

## 5.1 Models of Computation

Model of computation  
A model of computation fully describes a conceptual computer.

A **model of computation** is a mathematical description of how a conceptual computer processes the inputs of computational problems towards the output of their results. These models of computation are usually benchmarked against the Turing machine that was invented around 1930.

### Traditional Models

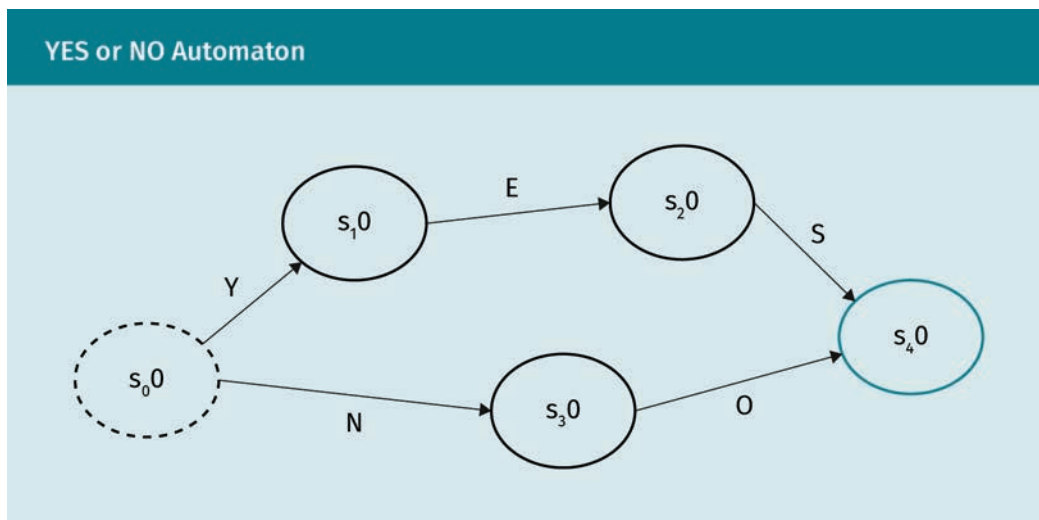
This subsection is dedicated to five models of computation: automata, Turing machines, lambda calculus, recursive functions, and first order predicate calculus. An effort will be made to present these models with the help of suitable understandable examples. Although this book does not include JavaScript examples of each of these computational models, they are all feasible and, in fact, have been done by various packages. If they use these packages, motivated readers will grasp the concepts in question in a rich and applied way. Let us recall that a model of computation is a description of how a conceptual computer processes its instructions written in a given formalized or formal language. As in the case for natural languages, the formal language of each conceptual computer is made up of semantically and syntactically approved words from a given alphabet. We will later see that each model of computation covers its own family of languages, with certain families being more powerful than the others.

### Automata

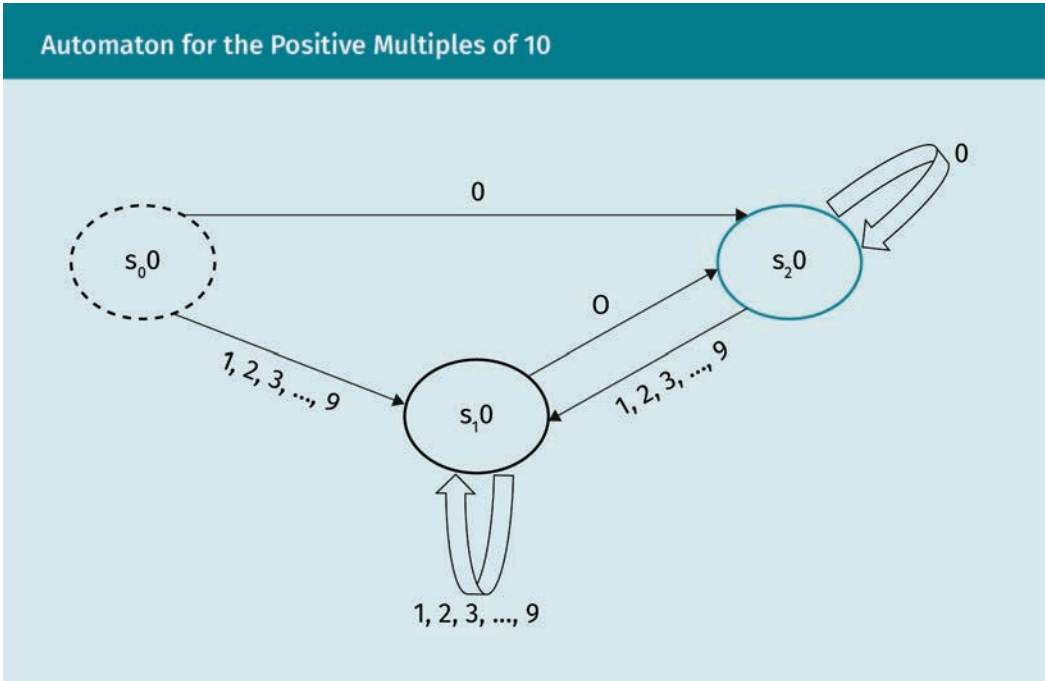
There are two types of automata: finite automata, which are the model of computation for regular languages based conceptual computers, and pushdown automata, which are the model of computation for context-free language-based conceptual computers. Let's

## Computability

suppose that we want to build a small conceptual computer whose task is simply to check whether an input from the user is solely made up of the two strings YES or NO. This conceptual computer is equivalent to the following finite deterministic automaton.



Assuming that all inputs are starting from the initial state ( $s_0$ ), it is easy to observe that all valid inputs will end up at the  $s_4$  acceptance state. It is important to note that the above automaton implicitly assumes that any non-valid transition will land in the dustbin state. For example, for the input string NOT, the automaton will be in  $s_4$  without having read the character T, and reading T from  $s_4$  is considered as a non-valid transition since such a transition is not explicitly stated. In other words, the input string NOT will end up in the dustbin state. Let us now build a finite automaton for a conceptual computer whose sole task is to check whether an input from the user in the decimal numbering system is a positive multiple of 10.



We can now formally define a finite automaton as a model of computation that includes the family of conceptual computers. These are made up of the following components: a finite number of states with one of them being the initial state (dotted line state) and one or more of them being the final accepting (green state) state(s), an alphabet of characters, and a transition function or table on how to move from one state to another state for each character of the alphabet. Any input leading to a non-authorized move by the machine is considered invalid. For instance, the above automaton is formally specified as follows:

Alphabet = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; states = {s<sub>0</sub>, s<sub>1</sub>, s<sub>2</sub>} with s<sub>0</sub> as the initial state and s<sub>2</sub> the final accepting state. The transition function is represented by the following table.

State	Alphabet's elements c	
	c = 0	c in {1,2,3,4,5,6,7,8,9}
s <sub>0</sub>	s <sub>2</sub>	s <sub>1</sub>
s <sub>1</sub>	s <sub>2</sub>	s <sub>1</sub>
s <sub>2</sub>	s <sub>2</sub>	s <sub>1</sub>

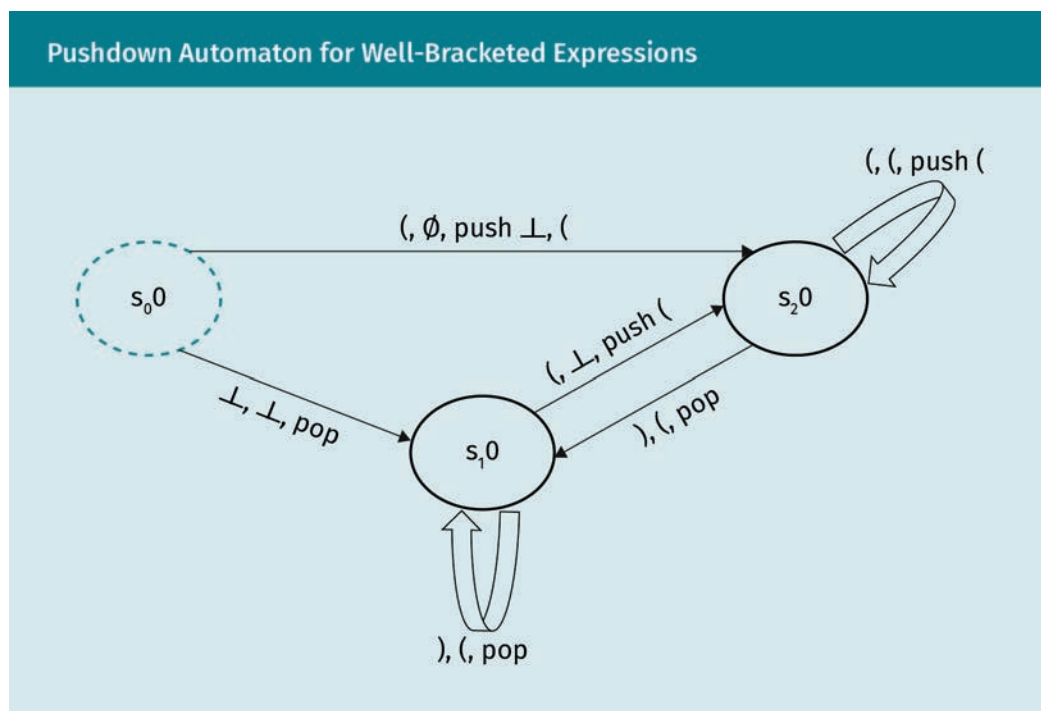


## Computability

Certain automata restrict their transition function to less than two moves per character from each state. Those are deterministic automata as opposed to nondeterministic automata where characters are allowed any number of moves from any state. It is worth mentioning that both finite deterministic and finite nondeterministic automata are part of the same model of computation as the one of regular languages.

One of the characteristics of regular languages is that the union of two regular languages remains a regular language; the same applies for their intersection. Another property of regular languages, the one proven by the pumping lemma, is that a word of a given regular language can only be arbitrarily long if and only if it is of the form  $pm^ns$ , where  $p$ ,  $m$ ,  $s$  are words and  $n$  is a positive integer. In other words, because the finiteness of the number of states of automata and their lack of memory, an arbitrary long word will have to go through a loop on its way to a final accepting state. As a result, many languages cannot be modeled by finite automata.

Let's consider a small machine whose sole task is to check whether a sequence of brackets entered by a user is well-bracketed in the sense that each open bracket is suitably closed. Such a machine needs a memory to keep track of the number of consecutively open brackets so that it can check if that number matches the one for the closed brackets.



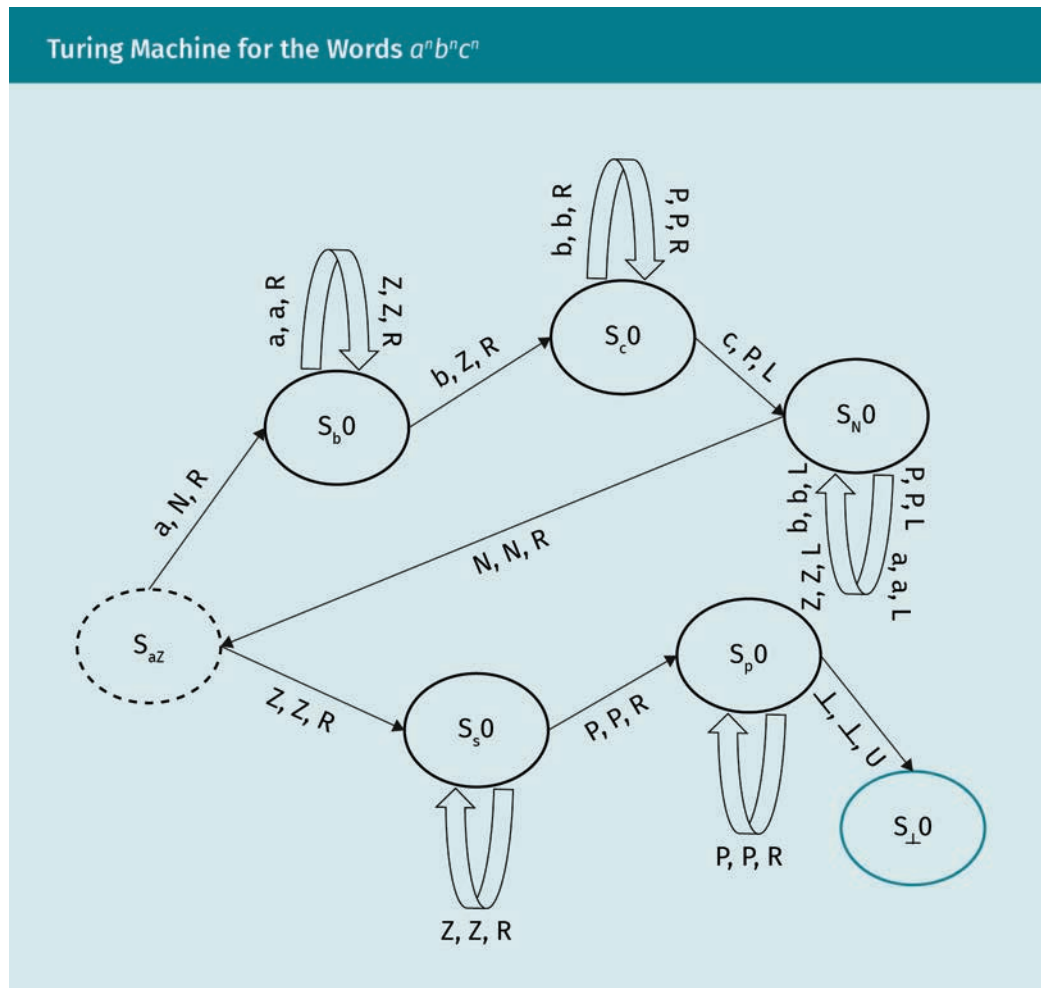
The above pushdown automaton uses a stack whose push operation increases the count of the number of consecutive open brackets.

The pop operation of the stack decreases the above count. All in all, a final state with an empty stack will correspond to a well-bracketed sequence. The concepts of states, transitions, and alphabets are similar to the ones of the finite automata except that, for pushdown automata, stacks may have an alphabet of their own in addition to the alphabet of the inputs. However, these two alphabets might sometimes coincide. It is also important to note that the transition arrow of a pushdown automaton is made up of three elements: the current letter of the input word being checked, the letter currently on the top of the stack, and the action to be performed on the stack (push or pop). For example, there is a transition between  $s_0$  and  $s_2$  and that transition has the “(,  $\emptyset$ , push , (” label. This means that from  $s_0$ , if the open bracket ( is the current letter of the input word, and the stack is currently empty  $\emptyset$ , then we have to push “ ” followed by ( on top of the stack and move to  $s_2$ .

Readers are advised that the above pushdown automaton assumes that well-formatted input words end with the “ ” character, and the stack is initially empty. Readers are also invited to test the machine using, for example,  $((()))()$  as an input word. They are also reminded that pushdown automata are the model of computation of context-free language-based conceptual computers where the stack serves as the memory device.

### **Turing machines**

The use of a stack as a memory device has its own limitations. This calls for other memory devices that make use of a less restrictive data structure as is the case of the tape of a Turing machine. Such tapes allow their read/write head to move to the left (L), to stay unchanged (U), or to move to the right (R).



The purpose of the Turing machine above is to recognize words of the form  $a^n b^n c^n$ , where  $n$  is a strictly positive integer. The transition from one state to another is made up of three components. For example, the  $a, N, R$  transition between the  $S_{az}$  state and the  $S_b$  state stipulates that from state  $S_{az}$ , if the head of the tape is currently pointing to  $a$ , then the letter  $a$  has to be replaced by  $N$ , the head of the tape must be moved to the position on its immediate right, and the Turing machine must go to state  $S_b$ . It is important to note that the input word is always copied on the tape at the beginning of the processing with the head of the Turing machine pointing to the first letter of that input. Here again, there is a terminating character  $\perp$  as was the case for pushdown automata. The Turing machine is tested here for the  $aabbcc$  input, but readers are invited to test it with other inputs.

## Turing Machine (aabbcc Input)

State	Tape						
$S_{aZ}$	↓a	A	b	b	c	c	⊥
$S_b$	N	↓a	b	b	c	c	⊥
$S_b$	N	A	↓b	b	c	c	⊥
$S_c$	N	A	Z	↓b	c	c	⊥
$S_c$	N	A	Z	b	↓c	c	⊥
$S_N$	N	A	Z	↓b	P	c	⊥
$S_N$	N	A	↓Z	b	P	c	⊥
$S_N$	N	↓a	Z	b	P	c	⊥
$S_N$	↓N	A	Z	b	P	c	⊥
$S_{aZ}$	N	↓a	Z	b	P	c	⊥
$S_b$	N	N	↓Z	b	P	c	⊥
$S_b$	N	N	Z	↓b	P	c	⊥
$S_c$	N	N	Z	Z	↓P	c	⊥
$S_c$	N	N	Z	Z	P	↓c	⊥
$S_N$	N	N	Z	Z	↓P	P	⊥
$S_N$	N	N	Z	↓Z	P	P	⊥
$S_N$	N	N	↓Z	Z	P	P	⊥
$S_N$	N	↓N	Z	Z	P	P	⊥
$S_{aZ}$	N	N	↓Z	Z	P	P	⊥
$S_Z$	N	N	Z	↓Z	P	P	⊥
$S_Z$	N	N	Z	Z	↓P	P	⊥
$S_p$	N	N	Z	Z	P	↓P	⊥
$S_p$	N	N	Z	Z	P	P	↓⊥
$S_{\perp}$	N	N	Z	Z	P	P	↓⊥

What is amazing about Turing machines is that they are the model of computation for all possible language-based conceptual computers. In other words, we define an algorithm to be computable if and only if it can be modeled by a Turing machine.

### Lambda calculus

Lambda calculus is a model of computation for conceptual computers whose language is made up of the terms defined below:

## Computability

- A variable such as  $x, y, z, a, b, c, d$ , and so on
- An anonymous lambda function that assigns the term  $t$  to a given variable  $x$ , denoted by  $\lambda x.t$
- An application  $ts$  that computes the term  $t$  using another term  $s$  as an argument

In other words, a term is either a variable, an anonymous lambda function, or an application. However, the use of brackets is sometimes unavoidable for the sake of clarification even though the left priority rule  $[t_1 t_2 t_3 = (t_1 t_2) t_3]$  applies by default. Here are a few examples of simple terms. The identity function for which each element is assigned to its own value is represented by the anonymous lambda function  $\lambda x.x$ , which is the same as  $\lambda y.y$  since  $x$  is a variable whose name is changeable.

Let's also look at the example of how functions are composed where  $f \circ g$  represents the composition of the functions  $f$  and  $g$ . What is the meaning of  $f \circ f$ ? For  $f$ , every variable is assigned to the value of its  $f$ -mapping, which is the value of its mapping by the function  $f$ . For  $f \circ f$ , each variable is assigned to its  $f \circ f$ -mapping which is the value of its mapping, by the function  $f \circ f$  (i.e., the  $f$ -mapping of its  $f$ -mapping). We can carry on with  $f \circ f \circ f$ ,  $f \circ f \circ f \circ f$ , and so on, up to eternity, but, let us pause for a while to look at how to translate these examples of function compositions into the lambda calculus language:

- The anonymous function that assigns any given variable  $f$  ( $f$  is a function) to its own value  $f$  is denoted in lambda calculus by  $\lambda f.(\lambda x.fx)$ .
- The anonymous function that assigns any given variable  $f$  ( $f$  is a function) to the value of its own double composition  $f \circ f$  is denoted in lambda calculus by  $\lambda f.(\lambda x.f(fx))$ .
- An anonymous function that assigns any given variable  $f$  ( $f$  is a function) to the value of its own triple composition  $f \circ f \circ f$  is denoted in lambda calculus by  $\lambda f.(\lambda x.f(f(fx)))$ .

These three bullets are the respective lambda calculus representations of the numbers 1, 2, and 3, with the convention that the lambda term  $\lambda f.(\lambda x.x)$  is a representation of 0.

It is possible to reduce a lambda calculus expression to a simpler form as is the case for other mathematical expressions. Such simplifications are done with the help of the following three rules: alpha-conversions, beta-reductions, and eta-reductions.

1. Alpha-conversion: For application terms, argument names should be changed in order to avoid the name clashing with the names of the variables of mapping functions.
2. Beta-reduction: An application term  $ts$  can be reduced by replacing the variable of the term  $t$  with the argument  $s$ .
3. Eta-reduction or conversion: The term  $\lambda x.(tx)$  can be simplified, reduced, or converted to the term  $t$  when the variable  $x$  is not used by the term  $t$ .

We will now illustrate the use of these simplification rules on the following lambda calculus term:  $\lambda n.(\lambda f.(\lambda x.f((nf)x)))$ .

Let us try to apply this function with the variable  $n$  equal to  $\lambda f.(\lambda x.x)$ . This will give the application term  $(\lambda n.(\lambda f.(\lambda x.f((nf)x))))(\lambda f.(\lambda x.x))$ .

$$\begin{aligned} & (\lambda n.(\lambda f.(\lambda x.f((nf)x))))(\lambda f.(\lambda x.x)) = (\lambda n.(\lambda f.(\lambda x.f((nf)x))))(\lambda f.(\lambda x.x)) \\ & = \lambda f.(\lambda x.f(((\lambda f.(\lambda x.x))f)x)) = \lambda f.(\lambda x.f((\lambda f.(\lambda x.x))f)x)) \text{ by beta-reduction} \\ & = \lambda f.(\lambda x.f((\lambda f.(\lambda x.x))f)x)) = \lambda f.(\lambda x.f((\lambda x.x)x)) \text{ by Eta-conversion} \\ & = \lambda f.(\lambda x.f((\lambda x.x)x)) = \lambda f.(\lambda x.f((x))) = \lambda f.(\lambda x.f(x)) = \lambda f.(\lambda x.fx) \end{aligned}$$

We have just applied  $\lambda f.(\lambda x.fx)$  to the  $\lambda n.(\lambda f.(\lambda x.f((nf)x)))$  lambda calculus term and landed on the term  $\lambda f.(\lambda x.fx)$ . Readers are also invited to apply  $\lambda f.(\lambda x.fx)$  to the  $\lambda n.(\lambda f.(\lambda x.f((nf)x)))$  lambda calculus term, and they will land on the term  $\lambda f.(\lambda x.f(fx))$  in confirmation of the fact that  $\lambda n.(\lambda f.(\lambda x.f((nf)x)))$  is the lambda calculus representation of the increment function that adds 1 to its parameter  $n$ .

Lambda calculus was discovered by Church and Kleene around 1930 (Turner, 2018). It is both **Turing-complete** and **Turing-equivalent** (Adams, 2018; Lyman, 2016) in the sense that any Turing machine can be modeled as a lambda calculus-based machine and vice-versa.

### Recursive functions

Discovered by Gödel and Herbrand around 1930, general recursive functions are Turing-complete (Wang, 1990). There are two types of recursive functions for a tuple variable  $X = (x_1, x_2, x_3, \dots, x_n)$ : primitive recursive functions and general recursive functions.

For a function to be declared a primitive recursive function, it has to fulfill one of the following conditions:

- It is a Zero function that assigns the value zero to each tuple variable  $X$ .
- It is a projection function that assigns a selected tuple coordinate to each tuple variable  $X$ .
- It is a projection successor function that assigns the successor of a selected tuple coordinate to each tuple variable  $X$  (e.g.,  $X$  is mapped to  $1 + x_3$ ).
- It is the composition of primitive recursive functions.
- It is the recursion of two primitive recursive functions  $g$  and  $h$ , as expressed below:

$$\begin{cases} f(X, 0) = g(X) \\ f(X, S(n)) = h(X, n, f(X, n)) \end{cases}$$

Let us consider a few examples of functions to check whether they are primitive recursive, starting with the increment function that simply adds 1 to its parameter. The increment function  $f(x) = x + 1$  is a primitive recursive function since  $x + 1$  is the successor of the projection on the first and unique variable  $x$ . Let's look at the function  $f$  that adds its two parameters  $x$  and  $n$ . This function is recursively written as follows even though it is not yet the formal primitive recursive language.

$$\begin{cases} f(x, 0) = x \\ f(x, S(n)) = f(x, n) + 1 \end{cases}$$

Turing-complete computational models  
These models are able to model any Turing machine.  
Turing-equivalent computational models  
Turing machines can simulate any Turing equivalent computational model.

## Computability

The workings of the formulation of the  $x + n$  function in the formal primitive recursive language are as follows:

- $f(x, 0) = g(x) = x$  with  $g$  being the projection on its first and unique variable
- $f(x, S(n)) = h(x, n, f(x, n))$  with  $h$  being the successor of the projection on its third attribute

This leads to the following formal formulation of the  $x + n$  function as a primitive recursive function

$$\begin{cases} f(x, 0) = g(x) \\ f(x, S(n)) = h(x, n, f(x, n)) \end{cases}$$

with

$$\begin{cases} g(t) = t \\ h(u, v, w) = S(w) \end{cases}$$

It is also possible to write the function  $f$  that multiplies its two parameters  $x$  and  $n$ . The function is written recursively as follows even though it is not yet the formal primitive recursive language.

$$\begin{cases} f(x, 0) = 0 \\ f(x, S(n)) = f(x, n) + x \end{cases}$$

The workings of the formulation of the  $x \cdot n$  function in the formal primitive recursive language are as follows:

- $f(x, 0) = g(x) = 0$  with  $g$  being the Zero function
- $f(x, S(n)) = h(x, n, f(x, n))$  with  $h$  being the addition of the first parameter with the third one.

This leads to the following formal formulation of the  $x \cdot n$  function as a primitive recursive function, assuming that we have already shown that the addition function is a primitive recursive function.

$$\begin{cases} f(x, 0) = g(x) \\ f(x, S(n)) = h(x, n, f(x, n)) \end{cases}$$

with

$$\begin{cases} g(t) = 0 \\ h(u, v, w) = \text{add}(u, w) \end{cases}$$

Readers are invited to convert other arithmetic operations such as subtraction, the power on one number to another one, and even the factorial to their formal primitive recursive formats. For now, we are going to direct our attention to general recursive functions.

A function  $f(X)$  is said to be a general recursive function if and only if it fulfills either of the following conditions:

- It is a primitive recursive function.
- It is the unbounded minimization of another general recursive function  $g$  and can be expressed as

$$f(X) = \text{Minimum } \{z \text{ such that } g(X, z) = 0\}$$

Here is an example of a general recursive function that is not primitively recursive, the Ackermann function  $A$ , defined as follows:

$$\begin{cases} A(0, y) = y + 1 \\ A(x + 1, 0) = A(x, 1) \\ A(x + 1, y + 1) = A(x, A(x + 1, y)) \end{cases} .$$

Below are the values of  $A(x, y)$  when  $x$  and  $y$  belong to the  $\{0, 1, 2, 3\}$  set.

	$y = 0$	$y = 1$	$y = 2$	$y = 3$
$x = 0$	1	2	3	4
$x = 1$	2	3	4	5
$x = 2$	3	5	7	9
$x = 3$	5	13	29	61

We can see from the table that any couple of natural numbers has a value for the Ackermann function. This function is thus recognized as a total general recursive function. In contrast, the following general recursive function only has a value when  $x = 0$  and not for any other  $x$ . It is recognized as a partial general recursive function:  $f(x) = \text{Min } \{z \mid \text{add}(x, z) = 0\}$ .

### First order predicate calculus

Discovered by Frege (1879/1990), first order predicate calculus is proven to be Turing-complete. It is a model of computation that consists in representing facts either as true or not true, with the help of quantifiers such as “for all” and “there exists.” Here is an illustration of the use of the first order predicate calculus model for the increment operation  $i$ . This example simply says that for any number  $n$ , it is true that the increment  $i$  of  $n$  is simply its successor  $s(n)$ :  $\forall n. i(n, s(n))$ .



## Computability

Similarly, the addition operation  $a$  can be expressed by the following first order predicate calculus expression according to which it is true that the addition of any number  $x$  with 0 is equal to  $x$ . According to the second part of that expression, it is true that if  $z$  is the result of the addition of  $x$  and  $y$ , then it is also true that  $z + 1$  is the result of the addition of  $x$  and  $y + 1$ .

$$\begin{cases} \forall x . a(x, 0, x) \\ \forall x . \forall y . \forall z . (a(x, y, z) \Rightarrow a(x, s(y), s(z))) \end{cases}$$

Let's try to compute the addition of 3 and 2 with this predicate calculus model of computation:  $a(3,0,3) \Rightarrow a(3,1,4) \Rightarrow a(3,2,5)$ .

## New Models and the von Neumann Machine

The above models have led to the creation of new models of computation such as the von Neumann machine, object calculus, and interaction nets. Object-oriented programming is one of the relatively new programming approaches whose model of computation is known as imperative object calculus as proposed by Abadi and Cardelli (1996). It is based on lambda calculus and it has been proven to be Turing-complete. Interaction nets are a graphical model of computation that was proposed by Lafont (1989) even though they also have a textual version. This model of computation is based on a linear logic model proposed by Girard (1987) as a combination of classical logic and constructive logic. It is important to note that interaction nets are Turing-complete. Computing is also currently being explored from the perspective of many other knowledge domains such as humanities, physics, chemistry, biology, biochemistry, and mathematics, with the hope of proposing new models of computation.

We would like to end this section with an overview of the von Neumann computer because of its central role towards the physical implementation of several abstract concepts from many of the aforementioned models of computation.

The von Neumann computer is made up of the following main components that are connected by a bus system: control unit, processing unit, memory, hard disk, and input/output devices. The role of the control unit is to coordinate the activities of the other components by continuously updating its instruction register and its program counter. The processing unit is made up of an arithmetic-logic unit and of registers for the processing of instructions. As for the memory, its role is to store instructions and data prior to their execution by the processing unit. The memory is organized as a sequence of addresses whose role is to identify the locations of program data and instructions.

Such addresses are used by the Fetch-Decode-Execute cycle of the control unit for its continuous coordination of the activities of the components of the computer. Let's not forget to mention input and output devices for their central role at the interface between the computer and its user. We must also remember that data and programs are transferred to the memory when it is their turn to be processed; otherwise, they are stored in the hard disk.

## 5.2 The Halting Problem

Having in mind that a function is supposed to be fed with an input for which it will yield an output, let us consider the following function denoted by `hp`:

- The input of `hp` is made up of two arguments `ta` and `ti` where `ta` represents any algorithm in its JavaScript textual form, and `ti` represents any text input of that algorithm `ta`. Please note that JavaScript is not mandatory here and can be replaced with any other language.
- The output of `hp` for the `(ta, ti)` input is a Boolean value that is equal to `true` if and only if the algorithm `ta` terminates for the text input `ti`.

The `hp` function portrays what is known as the halting problem, and it is well established that there is no possible `hp` function that can decide in advance whether any given algorithm `ta` will terminate for any given input `ti`.

Proof: Let us suppose that there is an algorithm `hpa` for the halting problem whereby `hpa(ta, ti)` always returns `true` if the algorithm `ta` terminates for the input text `ti` and returns `false` otherwise. We also have another function with the name of `happyCrazyLooper` that works as follows for a given input text `t`: It is happy to return `true` when `hpa(t, t)` is false; otherwise, it loops forever.

```
function happyCrazyLooper(t) {  
  if (hpa(t,t) == false) {return true}  
  if (hpa(t,t) == true) {loop forever}  
}
```

Having in mind that the `happyCrazyLooper` algorithm is also a text that we can denote, for example, by `hcl`, an interesting question is to find out if `happyCrazyLooper(hcl)` terminates. The simple example of a program that counts the number of words in a given text can help us understand that a program can use itself as an input, i.e., a program can count its own number of words.

If we first assume that `happyCrazyLooper(hcl)` terminates, then this will imply that `hpa(hcl, hcl)` is false. In other words, `hcl` does not terminate with `hcl` as an input, which also tells us that `happyCrazyLooper(hcl)` does not terminate. And that contradicts the initial first assumption made.

If we now assume that `happyCrazyLooper(hcl)` does not terminate, then this will imply that `hpa(hcl, hcl)` is true. In other words, `hcl` does terminate with `hcl` as an input, which also tells us that `happyCrazyLooper(hcl)` does terminate. It is a contradiction of the initial second assumption made. These contradictory conclusions point to the fact that the existence of the `hpa` algorithm is not possible. In other words, there is no possible algorithm for the halting problem.

### 5.3 Undecidable Problems

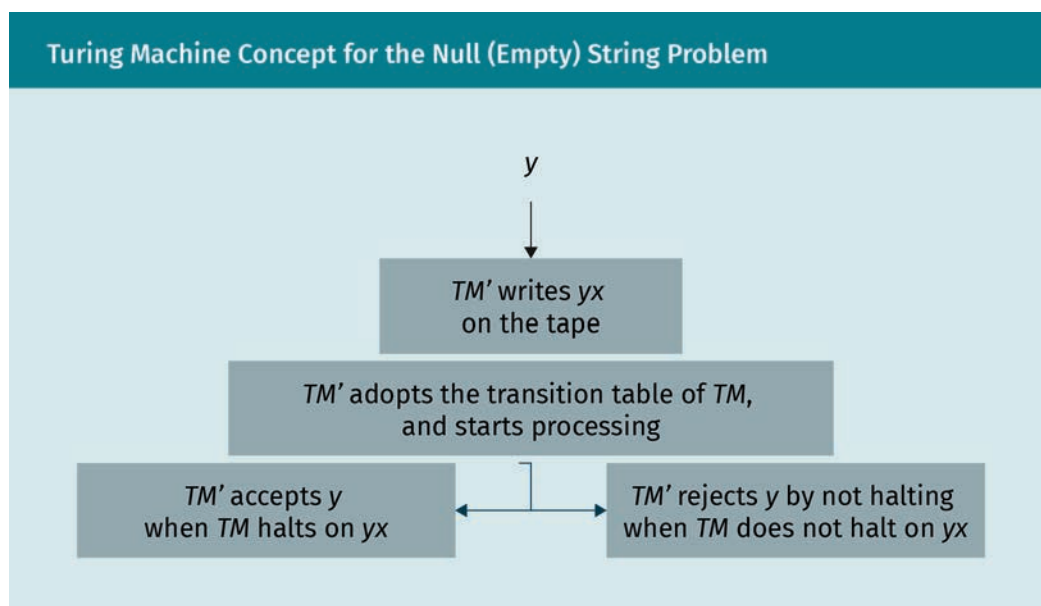
The previous section ended with the confirmation that it is not possible to have an algorithm for the halting problem. In other words, there is no possible algorithm that can decide in advance if a given algorithm will terminate for any given input. Otherwise, the halting problem is **undecidable**, and so are so many other problems from diverse mathematical sub-domains such as logic, numbers theory, differential equations, and Turing machines.

This section will present two undecidable Turing machine problems, but let us first take a few minutes to mention that the previous halting problem was initially described by Turing himself as a Turing machine problem. That initial formulation of the halting problem states that there is no algorithm that can decide in advance whether any randomly given Turing machine will terminate for a randomly given input.

It is now time to look at two other undecidable Turing machine problems: the null (empty) string problem and the problem of the membership to a recursively enumerable set.

#### The Null (Empty) String Problem

Let us suppose that we have a randomly chosen Turing machine,  $TM$ , as well as a randomly chosen input,  $x$ . We can then build the following Turing machine,  $TM'$ , whose behavior for each of its inputs  $y$  is based on  $TM$  and  $x$ .



Let us suppose for a moment that we found an algorithm NSA (Null String Algorithm) that can determine in advance whether any given Turing machine accepts the null(empty) string. This will imply that the NSA algorithm is able to determine in

Decision problems  
These aim at returning a binary answer such as true or false, yes or no, 0 or 1, etc.

advance whether the above described Turing machine 'TM' accepts the null string. In other words, having in mind that  $y\epsilon = \epsilon$  when  $y$  is the null(empty) string, the NSA algorithm is able to determine in advance whether or not TM halts on  $x$ . This shows that we have now found an algorithm that, for any randomly chosen Turing machine TM and any randomly chosen input,  $x$  is able to decide in advance whether TM halts on  $x$ .

This contradicts the undecidability of the halting problem, and we conclude that there is no algorithm that can determine in advance whether a given Turing machine accepts the null string.

### The Problem of the Membership in a Recursively Enumerable Set

It is possible to represent each imaginable algorithm and each conceivable algorithm's input by a natural number. Let us assign ourselves the task of writing a JavaScript pseudocode `listSHX` to print SHX. We assume that SHX is the list of all the algorithms (represented by numbers) that are halting on a randomly chosen input X (also represented by a number), having in mind that an algorithm is a sequence of many steps.

The `listSHX` JavaScript pseudocode shows that SHX is a recursively enumerable set, i.e., a set for which "there is a computer program that when left running forever eventually prints out exactly" its elements (Poonen, 2014, p. 3).

#### Recursively Enumerable Set Membership JavaScript Pseudocode

```
function listSHX() {
  res = {}; s = 1; // s represents the number of steps
  while (s >= 1){
    for (var an=1; an<=s; an++){
      if an halted on X after s steps {update res with an}
    }
    s++
  }
}
```

Is there a possible program that can determine whether an algorithm represented by a number  $n$  belongs to SHX? In other words, is there a program that can determine in advance whether any algorithm  $a$  will halt on a randomly chosen input X? The answer to that question is negative because of the undecidability of the halting problem. In other words, the problem of the membership of an algorithm to the recursively enumerable set SHX is undecidable. This also shows that the problem of determining whether or not a randomly chosen natural number belongs to a randomly chosen recursively enumerable set is undecidable.

### Summary

This unit presented five traditional models of computation including automata, Turing machines, lambda calculus, recursive functions, and first order predicate calculus. With the exception of automata, these classic models of computation are all Turing-complete. They are the foundation of contemporary programming language paradigms, such as imperative, functional, and logic. Newer models of computation, such as the von Neumann machine, object calculus, interaction nets, and nature-based computing models, were briefly introduced.

The concept of undecidability was presented and exemplified with the halting problem, the null (empty) recognition problem, and the problem of the membership to a recursively enumerable set. These examples are a living proof that Turing machines are powerful enough to compute all possible algorithms because of the equivalence between the concepts of algorithms and Turing machines. But there are still many computational problems that cannot be solved by any algorithm.

### Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!



# Unit 6



## Efficiency of Algorithms: Complexity Theory

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... how to measure the efficiency of an algorithm.
- ... the different computational complexity classes.
- ... key perceptions as to whether  $P = NP$ .

## 6. Efficiency of Algorithms: Complexity Theory

### Introduction

Let us recall that an algorithm is nothing more than a step-by-step method proposed as a solution to a computational problem. It is important for an algorithm to have the smallest possible number of steps for the fastest resolution of its computational problem. Apart from this time efficiency requirement, the quality of an algorithm also depends on the amount of space that it is using. Both complexity dimensions are important to assess if an implemented algorithm will be able to process an expected class of inputs.

The purpose of this unit is to present the main computational complexity models that are available for the measurement of the efficiency of algorithms and for the analysis of the complexity of computational problems. This unit will also present the different classes of computational complexity, in anticipation of the discussion of the question as to whether  $P = NP$ .

### 6.1 Models of Complexity

Model of complexity  
A model of complexity encompasses metrics used to measure the efficiency of algorithms and the difficulty of computational problems.

A **model of complexity** is an evaluation framework for the assessment of the complexity of computational problems or for the analysis of the efficiency of algorithms.

#### Time Efficiency Analysis with the Big O Approximation Model

Big O time complexity  
The big O time complexity model measures the upper bound of the execution time of an algorithm when its input size is considered to be infinitely large.

Let us recall that an algorithm is nothing more than a sequence of steps to solve a given computational problem. From that perspective, it is possible to consider that the number of steps of an algorithm is a suitable metric for the measurement of its efficiency. Let us also remember that different inputs do not necessarily yield the same number of steps for a given algorithm. One of the most used complexity models for the analysis of the time (and the space) efficiency of algorithms is the big O time and space approximation model. The big O model consists of approximating the **time** (or the **space**) efficiency of algorithms in the form of a function of their infinitely large input size. It is a measurement of an upper bound of the execution time (or the space requirement) of an algorithm for an infinitely large input size.

The formal definition of the big O notation stipulates that  $h(n) = O(f(n))$  if and only if two constant values  $b$  and  $c$  exist such that  $0 \leq h(n) \leq bf(n)$  for all values of  $n \geq c$ . In other words, the definition of the big O is that  $bf(n)$  is an upper bound of  $h(n)$  for high values of  $n$ . Below are a few examples of how to calculate the big O efficiency of an algorithm.

#### The biggest subsequence problem

Let us assume that we have a sequence of numbers and are looking for an algorithm to identify the first subsequence with the biggest sum among all the other subsequences.



## Biggest Subsequence Problem JavaScript Code (Start)

```

let read = require('readline-sync');

let captureElements = function(n){
  let a = new Array(); a = []; let i=0;
  while (i<n){
    let s = read.question('Next number please ');
    if ((s!=='')&&(isNaN(s)===false)){a.push(Number(s)); i++}
  }
  return(a)
}

let biggestSubSeq = function(a){
  let imax0b = 0; let imax0e = 0; let smax0 = a[0];
  for (let i = 0; i < a.length; i++){
    let localSum = 0;
    for (let j=i; j < a.length; j++){
      localSum = localSum + a[j];
      if (localSum>smax0){imax0b=i; imax0e=j; smax0=localSum;}
    }
  }
  a.push(imax0b); a.push(imax0e); a.push(smax0);
}

let Kadane = function(a){
  let imax0b = 0; let imax0e = 0; let smax0 = a[0];
  let localSum = a[0]; let localStart = 0; let localEnd = 0;
  for (let i = 1; i < a.length; i++){
    if ((localSum + a[i]) < a[i]){
      localSum = a[i]; localStart = i; localEnd = i;
    }
    else {localSum = localSum + a[i]; localEnd = i;}
    if (localSum>smax0){
      imax0b = localStart; imax0e = localEnd;
      smax0 = localSum;
    }
  }
  a.push(imax0b); a.push(imax0e); a.push(smax0);
}

```

Big O space complexity

The big O space complexity model measures the upper bound of the space used by an algorithm when its input size is considered to be infinitely large.

### Biggest Subsequence JavaScript Code (End)

```

while(true){
  let s = read.question('\nInput sequence size (Integer > 0) ');
  let n = Number(s);
  if ((s!=='') && (Number.isInteger(n)) && (n>0)){
    let m = parseInt(s); let ar = new Array();
    ar=captureElements(m); console.log('\nSequence captured\n'+ar);
    for (let i=0; i<2; i++){
      if (i==0){biggestSubSeq(ar); console.log('\n\n1st Version');}
      if (i==1){Kadane(ar); console.log('\n\nSecond Version');}
      let s = ar.pop(); let e = ar.pop(); let b = ar.pop();
      let r = ar.slice(b,e+1);
      console.log('Starting index of subsequence\n' + b);
      console.log('Ending index of subsequence\n' + e);
      console.log('Subsequence with biggest sum\n' + r);
      console.log('Sum of above subsequence\n' + s);
    }
  }
}

```

The biggest subsequence problem can be illustrated with the example of the sequence 3, -4, 8, -1, 6, -1 where it is visible that the subsequence with the biggest total (of 13) is 8, -1, 6. The above NodeJs JavaScript code presents two different algorithms for the biggest subsequence problem. These two algorithms are traced for the input sequence 3, -4, 8, -1, 6, -1, starting with the first one denoted by `biggestSubSeq`.

### Traced Algorithm

		a					
		3	-4	8	-1	6	-1
i ↓	j →	0	1	2	3	4	5
0		<u>3</u>	-1	<u>7</u>	6	<u>12</u>	11
1			-4	4	3	9	8
2				8	7	<u>13</u>	12
3					-1	5	4
4						6	5
5							-1

Sum of the elements of a from i to j

## Efficiency of Algorithms: Complexity Theory

This algorithm simply scans through each element of the sequence and calculates the sums of all the possible subsequences that are starting from that element. This process assigns a new value to the biggest subsequence whenever it finds a new subsequence that is bigger than the one that was previously considered the biggest.

Let us estimate the efficiency of this algorithm with the help of the big O notation, assuming that the input sequence has a length of  $n$ . When  $i = 0$ ,  $j$  loops from 0 to  $n - 1$ . In other words,  $j$  loops  $n$  times; each of these times a fixed number of basic steps are executed by the algorithm. When  $i = 1$ ,  $j$  loops from 1 to  $n - 1$ , i.e.,  $j$  loops  $n - 1$  times, and each of these times a fixed number of basic steps is executed by the algorithm. We can carry on with that pattern until we reach the point where  $i = n - 1$  and, in that case, the instructions inside the inner loop are only executed once. This leads to the following formula for the number of basic operations carried out by the above traced algorithm when the value of  $n$  is assumed to be infinitely big.

$$n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1 = \frac{n(n + 1)}{2} = O(n^2)$$

The equation shows that this algorithm has a quadratic big O time efficiency ( $n^2$  is the square of  $n$ ). Let us recall the following formal definition of the big O notation:  $h(n) = O(f(n))$  if and only if two constant values  $b$  and  $c$  exist such that  $0 \leq h(n) \leq bf(n)$  for all values of  $n \geq c$ . This definition of big O also clearly shows that  $bf(n)$  is an upper bound of  $h(n)$  for high values of  $n$ . For instance, one can see in the above example that  $\frac{1}{2} \cdot n(n + 1)$  is less or equal to  $n^2$  when  $n \geq 1$  (for this example,  $b = c = 1$ , and  $n^2$  is an upper bound of  $\frac{1}{2} \cdot n(n + 1)$ ).

The following table traces Kadane's algorithm for the subsequence problem with the same input sequence: 3, -4, 8, -1, 6, -1.

## Tracing Kadane's Algorithm

Before the loop					
3	-4	8	-1	6	-1
imax0b=0	imax0e=0	smax0=a[0]=3	localSum=a[0]=3	localStart=0	localEnd=0

Inside the loop with i=1, a[i]=a[1]=-4, localSum=3, and localSum+a[i]=-1					
3	-4	8	-1	6	-1
imax0b=0	imax0e=0	smax0=a[0]=3	localSum=-1	localStart=0	localEnd=1

Inside the loop with i=2, a[i]=a[2]=8, localSum=-1, and localSum+a[i]=7					
3	-4	8	-1	6	-1
imax0b=2	imax0e=2	smax0=8	localSum=8	localStart=2	localEnd=2

Inside the loop with i=3, a[i]=a[3]=-1, localSum=8, and localSum+a[i]=7					
3	-4	8	-1	6	-1
imax0b=2	imax0e=2	smax0=8	localSum=7	localStart=2	localEnd=3

Inside the loop with i=4, a[i]=a[4]=6, localSum=7, and localSum+a[i]=13					
3	-4	8	-1	6	-1
imax0b=2	imax0e=4	smax0=13	localSum=13	localStart=2	localEnd=4

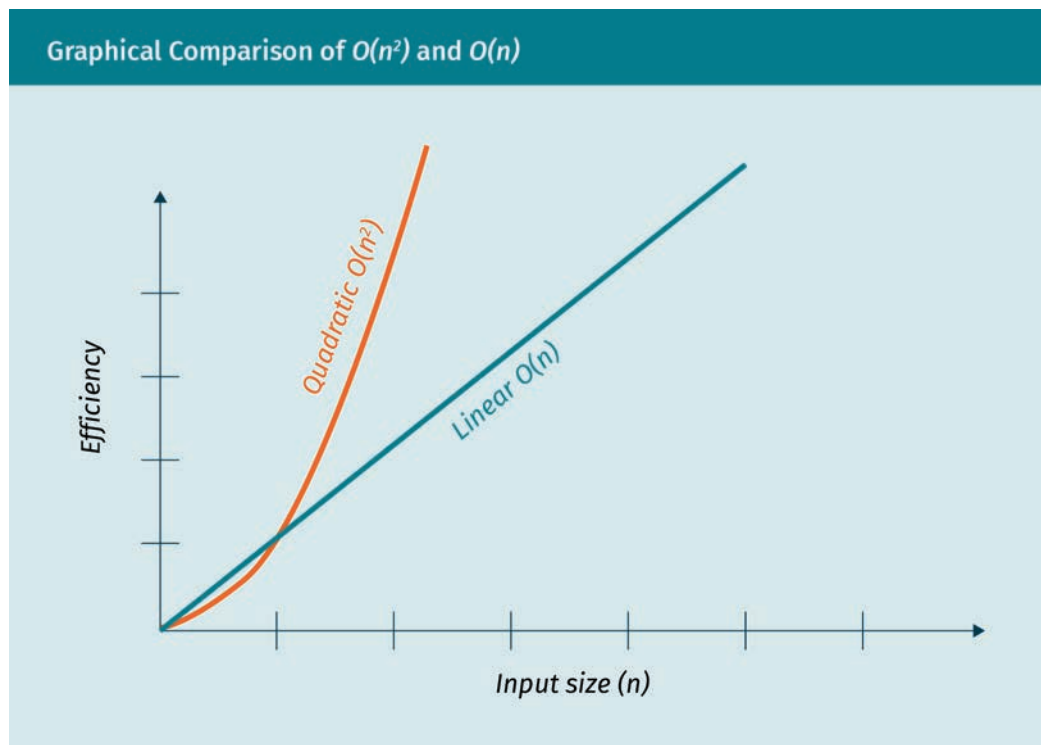
Inside the loop with i=5, a[i]=a[5]=-1, localSum=13, and localSum+a[i]=12					
3	-4	8	-1	6	-1
imax0b=2	imax0e=4	smax0=13	localSum=12	localStart=2	localEnd=5

The main characteristic of the traced algorithm is that, when we try to add a value  $a[i]$  to the current local sum, it is better to start a new local sum from the element  $a[i]$  if that element is greater than the addition. For example, if the current local sum is  $-3$ , and we are trying to add  $-2$  to it, then the addition will give  $-5$ , which is less than  $-2$ . Therefore, it is better to start a new sum from  $-2$ . Another key characteristic of this algorithm is that it is made up of a single loop instead of a double loop, as was the case for the first one `biggestSubSeq`.

It is clear that the traced Kadane's algorithm (above) loops  $n$  times, and for each of these times, a fixed number of operations is executed. If we denote that fixed number of operations with  $c$ , then the total number of operations executed by that algorithm can be estimated by the following formula when  $n$  is assumed to be infinitely big.

$$cn \approx n = O(n)$$

Both algorithms are a typical illustration of the fact that for the same computational problem, it is possible to have different algorithms, but the most valuable ones are the ones that are proven to be the most efficient. In this case, Kadane's algorithm with a linear big O time efficiency function (a linear function of  $n$ ) is by far more efficient than the other algorithm that has a quadratic big O time efficiency function. This difference is visible in the graph below comparing the evolution of the quadratic function to a linear function.



### Fibonacci numbers

Fibonacci numbers are quite popular in computing and mathematics. They are defined as follows:

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(i) = F(i-1) + F(i-2) \text{ for } i \geq 2 \end{cases}$$

The following NodeJS JavaScript code contains two different algorithms for the computation of the Fibonacci number  $F(n)$  when  $n$  is considered a given positive integer.

In the [following program](#), the `usualFib` function is an implementation of the usual calculation of Fibonacci numbers. It is a loop from 2 to  $n$  as an iterative computation of  $F(0)$ ,  $F(1)$ ,  $F(2)$ ,  $F(3)$ ,  $F(4)$ , ..., and so on, up to  $F(n)$ . That loop is executed almost  $n$  times with a fixed number of operations for each instance of the loop. We can say that the big O approximation of the `usualFib` algorithm is the following when the value of  $n$  is assumed to be infinitely big.

$$cn \approx n = O(n)$$

### Fibonacci Algorithms (Start)

```
let read = require('readline-sync');

let usualFib = function (n){
  if (n>=0){
    if (n===0){return(0);}
    if (n===1){return(1);}
    if (n>=2){
      let fp = 0; let fi = 1;
      for (let i = 2; i <= n; i++) {
        let f = fi; fi=fp+fi; fp = f;
      }
      return(fi);
    }
  }
}
```



## Fibonacci Algorithms (Cont'd)

```

let multMatr = function (a,b){
  let c = new Array(4);
  c[0]=(a[0]*b[0]) + (a[1]*b[2]); c[1]=(a[0]*b[1]) + (a[1]*b[3]);
  c[2]=(a[2]*b[0]) + (a[3]*b[2]); c[3]=(a[2]*b[1]) + (a[3]*b[3]);
  return (c);
}

let powerMatr = function (m, n){
  if (n>=1){
    if (n===1) {return m;}
    if (n>1){
      if((n%2)===0){let m1=powerMatr(m,n/2);return multMatr(m1,m1);}
      if ((n%2)!==0) {let m1 = powerMatr(m,(n-1)/2);
        return multMatr(multMatr(m1,m1),m);
      }
    }
  }
}

let fastFib = function (n){
  if (n>=0){
    if (n===0) {return 0;}
    if (n>=1) {let mf = [1, 1, 1, 0]; let mp = powerMatr(mf, n);
      return mp[1];
    }
  }
}

let powerMatrI = function (m, n){
  if (n>=1){
    if (n===1) {return m;}
    if (n>1){
      let s = new Array(); let i = n;
      while(i>1){
        if ((i%2)===0) {s.push(1); i=i/2;}
        else {s.push(2); i=(i-1)/2;}
      }
      let mp = new Array(4); mp[0] = m[0]; mp[1] = m[1];
      mp[2] = m[2]; mp[3] = m[3];
      while((s.length)>0){
        i = s.pop(); if (i===1) {mp = multMatr(mp, mp);}
        if (i===2) {mp = multMatr(multMatr(mp, mp), m);}
      }
      return(mp);
    }
  }
}

```

## Fibonacci Algorithms (End)

```

let fastFibI = function (n){
  if (n>=0){
    if (n===0) {return 0;}
    if (n>=1){
      let mf=[1, 1, 1, 0]; let mp=powerMatrI(mf, n); return mp[1];
    }
  }
}

while (true){
  let s = read.question('\nInput sequence size (Integer >= 0) ');
  let n = Number(s);
  if ((s!=='') && (Number.isInteger(n)) && (n>=0)){
    let m = parseInt(s); let ms = ' Algorithm. Fibonacci number F';
    console.log('Usual' + ms + m + ' : ' + usualFib(m));
    console.log('Recursive Fast' + ms + m + ' : ' + fastFib(m));
    console.log('Iterative Fast' + ms + m + ' : ' + fastFibI(m));
  }
}

```

The rationale behind the `fastFib` function as another way to compute Fibonacci numbers is less obvious. It deserves a few words, starting with the presentation of the following matrix  $M$ :

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Let us multiply the matrix  $M$  by itself a few number of times:  $M \cdot M = M^2$ ;  $M \cdot M \cdot M = M^3$ ;  $M \cdot M \cdot M \cdot M = M^4$ ;  $M \cdot M \cdot M \cdot M \cdot M = M^5$  and so on.

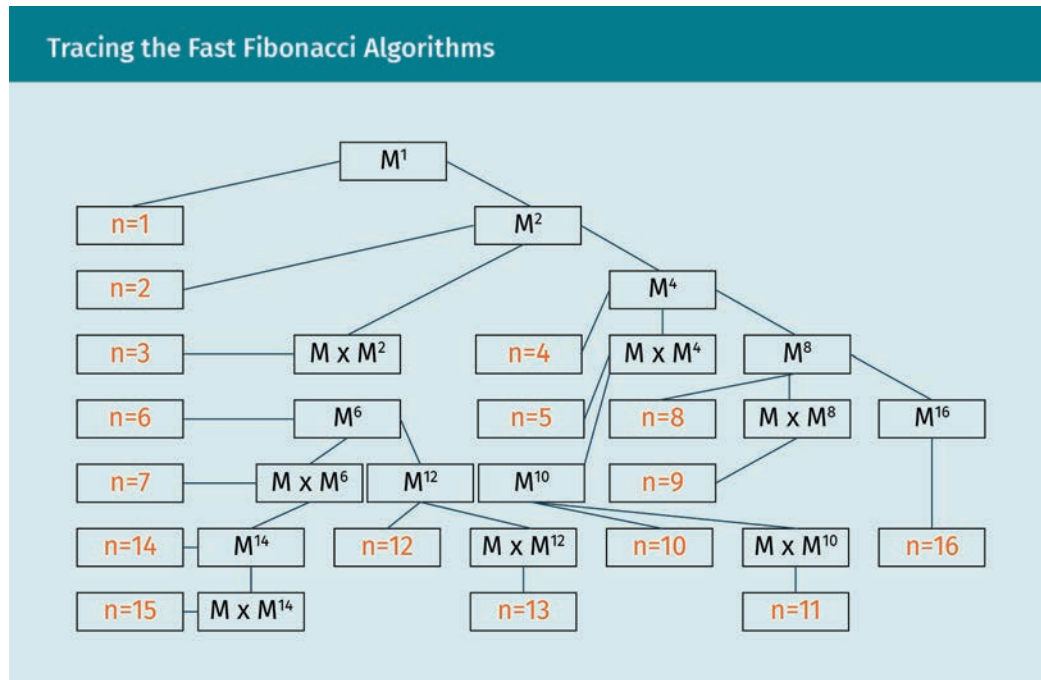
$M^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$	$M^2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$	$M^3 = \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}$	$M^4 = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}$
$M^5 = \begin{pmatrix} 8 & 5 \\ 5 & 3 \end{pmatrix}$	$M^6 = \begin{pmatrix} 13 & 8 \\ 8 & 5 \end{pmatrix}$	$M^7 = \begin{pmatrix} 21 & 13 \\ 13 & 8 \end{pmatrix}$	$M^8 = \begin{pmatrix} 34 & 21 \\ 21 & 13 \end{pmatrix}$

The above table can be generalized by stating that

$$M^n = \begin{pmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{pmatrix}$$



This formula shows that the calculation of  $M^n$  directly leads to the value of  $F(n)$ . This is precisely what is exploited by the `fastFib` function.



A closer look at the `fastFib` function shows how the matrix  $M$  is raised to the power  $n$ , visible above for all the numbers  $n$  from 1 to 16. Let us start by tracing the above tree for  $n = 16$ . The calculation of  $M^{16}$  ( $= M^8 \cdot M^8$ ) calls for the calculation of  $M^8$  ( $= M^4 \cdot M^4$ ), which itself calls for the calculation of  $M^4$  ( $= M^2 \cdot M^2$ ), and then  $M^2$  ( $= M^1 \cdot M^1$ ). This shows that the calculation of  $M^{16}$  succeeds with only four ( $4 = \log_2(16)$ ) multiplications of two matrices. The calculation of  $M^{32}$  will succeed with only five ( $5 = \log_2(32)$ ) multiplications of two matrices. In general, when  $n$  is a power of 2, the calculation of  $M^n$  will succeed with only  $\log_2(n)$  multiplications of two matrices. This process works well when the division by two always lands on an integer from  $n$  down to 1. The situation is, however, slightly nuanced when the number to be halved is odd.

Let us trace the tree above for  $n = 25$ . The calculation of  $M^{25}$  ( $= M^{12} \cdot M^{12} \cdot M$ ) calls for the calculation of  $M^{12}$  ( $= M^6 \cdot M^6$ ) which itself calls for the calculation of  $M^6$  ( $= M^3 \cdot M^3$ ) and the one of  $M^3$  ( $= M^1 \cdot M^1 \cdot M$ ). This shows that the calculation of  $M^{25}$  succeeds with only four ( $4 \leq \log_2(25)$ ) matrices multiplication requests, and each of these requests consists of one or two multiplications. It can be generalized that when  $n$  is not a power of 2, the calculation of  $M^n$  succeeds with less than  $\log_2(n)$  matrices multiplication requests, with each request being made up of one or two multiplications. Let us recall that when  $n$  is a power of 2,  $M^n$  succeeds with exactly  $\log_2(n)$  matrices multiplication requests, with each request being made up of one multiplication. Having in mind that the multiplication of two  $2 \times 2$  matrices involves few arithmetic operations, we can conclude that the total number of operations executed by the `fastFib` algorithm can be estimated with the following formula when  $n$  is assumed to be infinitely big.

$$c \log_2(n) = O(\log_2(n))$$

Once again, we have two different algorithms for the same computational problem where one of them, `fastFib`, is more efficient than the other one, `usualFib`. In fact, the big O approximation function of the time efficiency of `fastFib` is logarithmic ( $\log_2(n)$ ) while the one of `usualFib` is linear.

### Space Efficiency Analysis with the Big O Approximation Model

Until now, we have focused on the approximation of the time efficiency of algorithms using the big O model. Let's briefly turn our attention to the big O approximation of the space efficiency of the previous two algorithms.

#### The biggest subsequence problem

The two algorithms that were proposed for this problem both make use of a small number of integer variables ( $c$ ). This number does not depend on the size of the input array. We are thus able to conclude that the total amount of space used with each of these algorithms can be estimated by the following formula when the size  $n$  of the input array is assumed to be infinitely big.

$$f(n) = c = O(1)$$

This equation shows that the big O approximation function of the space efficiency of each of these two algorithms is constant. It might still be necessary to consider the size  $n$  of the input array as part of the space requirements of these algorithms, for example, when the array is read from a network.

#### Fibonacci numbers

The `usualFib` algorithm that was proposed for Fibonacci the problem makes use of a small number of integer variables. This number does not depend on the value of the input  $n$ . We can conclude that the total amount of space used by that algorithm can be estimated with the following formula when  $n$  is assumed to be infinitely big:  $f(n) = c = O(1)$ . This formula shows that the big O approximation function of the space efficiency of the `usualFib` algorithm is constant.

The `fastFib` algorithm uses a recursive version of the `powerMatr` function; recursive calls need to store all local variables of each of the functions that they visit until either it is finished, or the variable is not necessary anymore. Therefore, "tail-recursive" functions have an added advantage. Let us look at `powerMatrI`, which is the iterative version of the `powerMatr` algorithm, in order to estimate the amount of space used by these two algorithms. It is visible that the first loop of the `powerMatrI` algorithm builds a stack  $s$  to record the parity of the successive numbers when they are halved because that parity determines how the second loop multiplies its matrices (one multiplication or two multiplications). As was the case with the big O function of the time efficiency of the `fastFib` algorithm because of the halving of numbers, we can conclude that the stack  $s$  does not contain more than  $\log_2(n)$  integers. Moreover, the total amount of space used by the `fastFibI` (also by the `fastFib`) algorithm can be estimated by the following formula when  $n$  is assumed to be infinitely big.

$$c \log_2(n) \simeq \log_2(n) = O(\log_2(n))$$

## Worst, Average, and Best Case Efficiency Analysis

There are many situations where the efficiency of an algorithm depends on the choice of the value of its input. This calls for the need to analyze the efficiency of algorithms for the best possible, worst possible, and average scenarios. It also helps provide an estimate of the waiting time for the termination of an algorithm that is run by a user. Below is an example of a search algorithm that seeks to identify the position of the first occurrence of a given element in a given sequence of elements.

### Search Algorithm Example

```
let r = require('readline-sync');

let captureElements = function (n, a){
  for(let i=0;i<n;i++){a.push(r.question('Next element please? '));}
}

let findPosition = function (e, a){
  let n = a.length; let i = 0;
  while ((i<n) && (a[i]!==e)) {i++;}
  if ((i<a.length)&&(a[i]==e)) {return (i);} else {return (-1);}
}

while (true){
  let s = r.question('\nNb of elements? '); let n = Number(s);
  if ((s!=='') && (Number.isInteger(n)) && (n>0)){
    let m=parseInt(s); let ar=new Array(); captureElements(m, ar);
    console.log('Sequence captured ' + ar);
    let es = r.question('Element searched please? ');
    let p = findPosition(es, ar);
    console.log('Position of ' + es + ' in the sequence ' + p);
  }
}
```

### Worst case scenario

The worst case scenario is the situation whereby the input of the algorithm pushes it to perform the highest possible amount of work. Let's consider the search algorithm that scans through its input sequence from its first element to its last. Here, the worst case scenario corresponds to the situation where the element that is being sought is either the last element of the sequence or does not even belong to the sequence. In that case, the algorithm would have checked each and every element of the sequence. This means that the total number of comparisons made in the worst possible case by the search algorithm can be estimated with the following formula when  $n$  (the number of elements in the input sequence) is assumed to be infinitely big.

$$cn \simeq n = O(n)$$

This formula shows that the worst case big O approximation function of the time efficiency of the above search algorithm is linear.

### Best case scenario

The best case scenario is the situation where the input of the algorithm pushes that algorithm to perform the smallest possible amount of work. Let's consider the search algorithm that scans through its input sequence from its first element to the last one. Here, the best case scenario corresponds to the situation where the element that is being sought is the first element of the sequence. In this case, the algorithm would have only checked one element of the sequence. This means that the total number of comparisons made in the best possible case by the search algorithm can be estimated with the following formula when  $n$  (the number of elements in the input sequence) is assumed to be infinitely big.

$$f(n) = c = O(1)$$

This formula shows that the best case big O approximation function of the time efficiency of the above search algorithm is constant.

### Average case scenario

The average case scenario assumes that algorithms' inputs have random values. We assume a uniform random distribution of data here to avoid data repeats and biases that can influence the evaluation of the efficiency of an algorithm. For such randomly distributed data, the big O approximation notation can still represent the calculated average efficiency when the value of the input size is considered to be infinitely big. If we again consider the search algorithm, the general case is the one where the element being sought can be found in any position in the sequence.

If the element being sought is the first element of the sequence, then only one comparison is made by the search algorithm. If the element being sought is the second element of the sequence, then only two comparisons are made by the search algorithm. We can easily see that if the element being sought is the  $p^{\text{th}}$  element of the sequence, then only  $p$  comparisons are made by the search algorithm. The probability of an element to be in position  $p$  is  $\frac{1}{n}$  because of the hypothesized uniform distribution principle, and  $p$  can take any value between 1 and  $n$ . In other words, the total number of comparisons made in the average case by the search algorithm can be estimated by the following formula when  $n$  (the number of elements in the input sequence) is assumed to be infinitely big. The formula shows that the average case big O approximation function of the time efficiency of the above search algorithm is linear.

$$\frac{1}{n} + \frac{2}{n} + \frac{3}{n} + \dots + \frac{n-2}{n} + \frac{n-1}{n} + \frac{n}{n} = \frac{1}{n}(1 + 2 + \dots + n) = \frac{n(n+1)}{2n} = O(n)$$

Here is a summary of the names of the most common big O approximation functions.

Big O approximation function name	Big O Approximation function notation
Constant	$O(1)$
Logarithmic	$O(\log_2(n))$
Linear	$O(n)$
Log linear	$O(n\log_2(n))$
Quadratic	$O(n^2)$
Exponential	$O(c^n)$ with $c$ constant

## Complexity Classes

It is one thing to estimate the efficiency of a given algorithm, but it is entirely another issue to estimate the level of complexity or difficulty of a given computational problem. In fact, certain problems are intrinsically more difficult to resolve than others. This is why it is possible to classify computational problems into distinct complexity classes according to their different levels of difficulty or complexity. Such a classification implies that two different problems will belong to the same complexity class if and only if their best possible algorithms have the same big O approximation function. A complexity class is formally defined by the following four elements: a computational model (i.e., Turing machine), a computational mode (deterministic or nondeterministic), a resource (time or space), and a lowest upper bound (big O approximation of the best possible algorithms). The deterministic term is abbreviated to  $D$ , and the nondeterministic term to  $N$  for the naming of computational classes. For example,  $DTime(n^k)$  is the complexity class of all the computational problems whose best possible algorithm has a big O time efficiency approximation function of  $n^k$  when using a deterministic Turing machine as the model of computation. Similarly,  $NSpace(\log_2(n))$  is the complexity class of all the computational problems whose best possible algorithm has a big O space efficiency approximation function of  $\log_2(n)$  when using a nondeterministic Turing machine as the model of computation. We see from these two examples that the name of a complexity class is of the form  $MTime(f)$  or  $MSpace(f)$  where  $M$  is either  $D$  or  $N$  and  $f$  is a big O approximation function of  $n$ . The following table gives a list of the most common complexity classes.

Most Common Complexity Classes		
Acronym	Notation	Name
REG	$DSPACE(1) = NSPACE(1)$	Regular languages problems
L	$DSPACE(\log_2(n))$	Deterministic logarithmic space problems
NL	$NSPACE(\log_2(n))$	Nondeterministic logarithmic space problems
PSPACE	$DSPACE(n^{O(1)}) = NSPACE(n^{O(1)})$	Polynomial space problems ( $n, n^2, n^3$ , etc.)
EXPSPACE	$DSPACE(2^{n^{O(1)}}) = NSPACE(2^{n^{O(1)}})$	Exponential space problems ( $2^n, 2^{n^2}, 2^{n^3}$ , etc.)
CONSTTIME	$DTime(1) = NTime(1)$	Constant time problems
DLOGTIME	$DTime(\log_2(n))$	Deterministic logarithmic time problems
NLOGTIME	$NTime(\log_2(n))$	Nondeterministic logarithmic time problems
P	$DTime(n^{O(1)})$	Deterministic polynomial time problems
NP	$NTime(n^{O(1)})$	Nondeterministic polynomial time problems
EXPTIME	$DTime(2^{n^{O(1)}})$	Deterministic exponential time problems
NEXPTIME	$NTime(2^{n^{O(1)}})$	Nondeterministic exponential time problems

It is proven that the following relationships are true for any big O approximation function  $f(n)$ .

$$\begin{aligned} \text{DTIME}(f(n)) &\subseteq \text{NTIME}(f(n)) \\ \text{DSPACE}(f(n)) &\subseteq \text{NSPACE}(f(n)) \\ \text{DTIME}(f(n)) &\subseteq \text{DSPACE}(f(n)) \\ \text{NTIME}(f(n)) &\subseteq \text{NSPACE}(f(n)) \\ \text{NTIME}(f(n)) &\subseteq \text{DSPACE}(f(n)) \\ \text{NSPACE}(f(n)) &\subseteq \text{DTIME}(2^{f(n)}) \\ \text{NTIME}(f(n)) &\subseteq \text{DTIME}(2^{f(n)}) \\ \text{NSPACE}(f(n)) &\subseteq \text{DSPACE}((f(n))^2) \end{aligned}$$

These properties also lead to the following hierarchy of complexity classes.

$$\text{CONSTTIME}(1) \subseteq \text{REG} \subseteq \text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}$$

The attention of readers is drawn to the use of the  $\subseteq$  symbol above. This is because it is still an open question whether that  $\subseteq$  symbol should be replaced by the  $\subset$  symbol or even by the  $=$  symbol in certain instances. For example, it is well proven that  $\text{P} \subseteq \text{NP}$ , but it is an open question whether  $\text{P} \subset \text{NP}$  or if  $\text{P} = \text{NP}$ . On the other hand, Vega (2016) recently claimed that  $\text{NL} = \text{P}$ .

Each of the identified complexity classes has several computational problems that cannot be explicitly listed. The following table is an attempt to describe a few examples of computational problems for some of the above identified complexity classes.

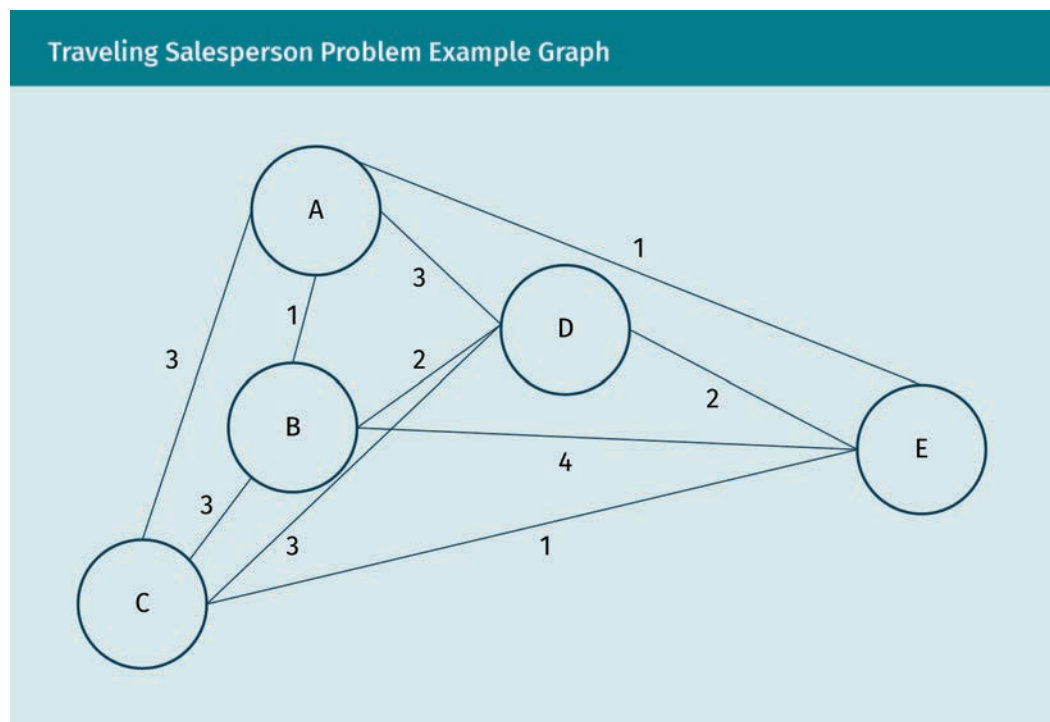
Computational Problems (Examples)	
Class	Problem
REG	Deciding whether or not a given natural number (in the decimal number system) is a multiple of ten
L	Deciding whether or not all opened brackets are well closed in a string of brackets
NL	Deciding whether or not there is a path between two given points of a given directed graph
P	Identifying the complement of a given graph
NP	Factorizing a given natural number

## 6.2 NP-Completeness

It seems opportune to introduce this section with the example of the well-known traveling salesperson and Hamilton cycle problems because of the central role of the notion of reducibility in the conceptualization of NP-completeness.

### The Traveling Salesperson Problem (TSP)

The formulation of the traveling salesperson problem is quite simple. It seeks to determine whether a person can travel from a point of origin and back to the same point after visiting each city of a given network exactly once, and without exceeding a given maximum total distance. The following figure is an example that represents five cities denoted by A, B, C, D, and E and their distances.

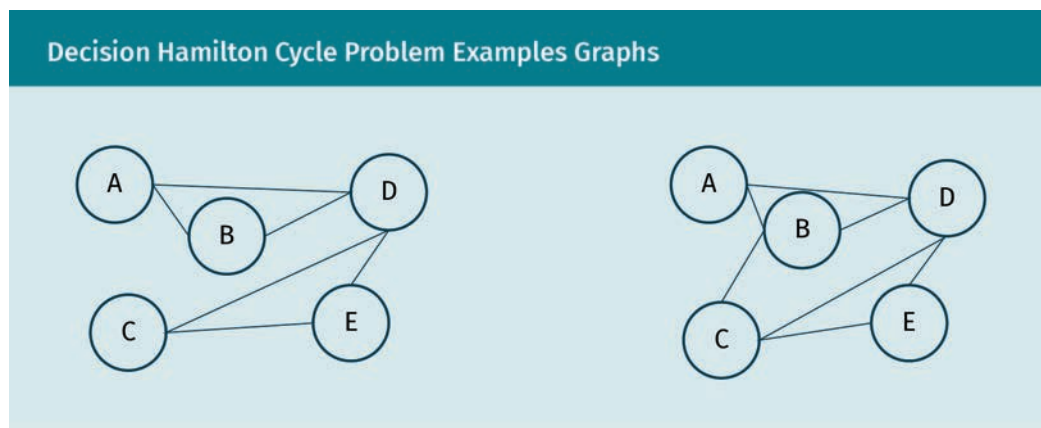


The ABCDEA route has a total distance of 10 while the ACEBDA has a total distance of 13, but the total distance of the AECDBA route is only 8. This shows that there is a solution here for a maximum distance of 8.



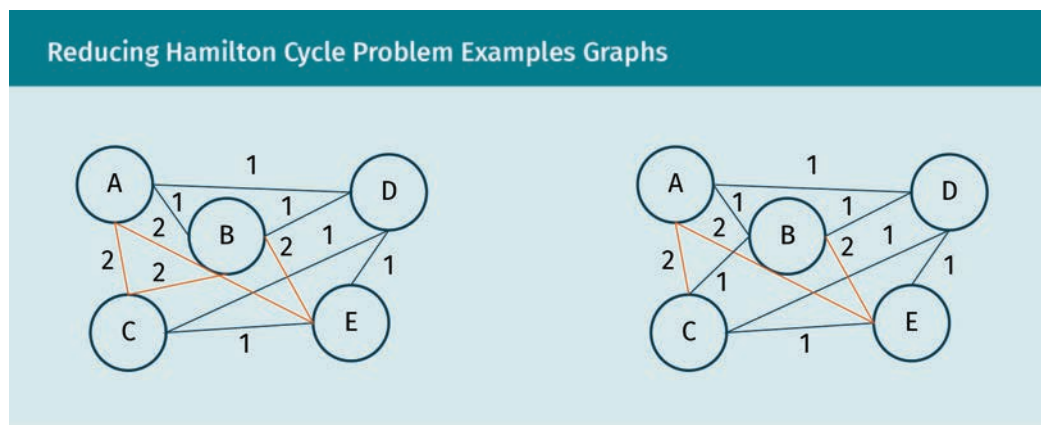
## The Hamilton Cycle Problem (HCP)

The Hamilton cycle problem seeks to determine whether a given graph has a path from a point of origin and back to the same point (cycle) after visiting each node of the graph exactly once. For example, the first graph below does not have a Hamiltonian cycle while ADECBA, DECBA D, ECBAD E, CBADCE, and BADECD are Hamiltonian cycles for the second graph.



## The Concept of Reducibility

Although they both deal with graphs, TSP and HCP are two different computational problems. In fact, the graph of the former is a fully connected graph but that is not necessarily the case for the latter. Similarly, the first problem aims to optimize distances while the second one does not involve any distance concept. Let us transform the two graphs of the HCP by assigning a value of one to the existing links and by creating new links with a value of two so that each of these graphs can be fully connected.



Having in mind that each of the graphs has five nodes, let us determine for both whether a person can travel from a point of origin and back to the same point after visiting each node exactly once, without exceeding a maximum total distance of five. The answer is no for the first graph but yes for the second.

What has just happened? An instance  $H$  of the HCP has been transformed or reduced into an instance  $T$  of the TSP so that the answer of  $H$  can be directly derived from the answer of  $T$ . In general terms, any instance of the HCP can be reduced into an instance of the TSP by assigning a value of one to the existing links of the graph of the HCP and by creating new links with a value of two in order for that graph to become fully connected. The big  $O$  approximation of the time efficiency of the above described transformation or reduction is equal to  $O(n^2)$  (polynomial reduction) where  $n$  is the number of nodes of the graph of the HCP. This is because such a reduction simply assigns relevant values in the  $n \times n$  matrix that represent the HCP graph. The general definition of reducibility states that a problem  $PRO$  is reducible to another problem  $QUE$  if and only if each instance  $p$  of the problem  $PRO$  can always be transformed into an instance  $q$  of the problem  $QUE$  such that the solution of  $p$  is directly derived from the solution of  $q$ . We have described how to reduce any instance of the HCP into an instance of the TSP. There is also an algorithm to reduce each instance of the TSP into an instance of an HCP.

### NP-Complete and NP-Hard Problems

Having in mind that the NP complexity class represents the class of nondeterministic polynomial time computational problems, NP-complete problems are the subclass of the most difficult decision problems (with a yes or no answer) of that class. For a given complexity class  $C$ , we define the  $C$ -complete complexity class as the subclass of the most difficult problems of  $C$ .

It is well established, for example, that the TSP is a NP-complete problem in the sense that it is one of the most difficult NP problems. An interesting question is to determine whether a given new problem is NP-complete. This requires proving that every instance of such a problem can be reduced in polynomial time to an instance of a well-known NP-complete problem and vice versa. This is, for instance, the case of the HCP that was proven to be reducible to TSP in polynomial time, while it is also established that TSP is reducible to HCP in polynomial time. In other words, the HCP is also NP-complete. Moreover, once it is proven that for a given problem  $H$ , an existing NP-complete problem is reducible in polynomial time to  $H$ , that problem  $H$  is said to be **NP-hard**.

NP-hard problems  
The polynomial time  
reduction of an  
existing NP-com-  
plete to another  
problem makes that  
other problem NP-  
hard.

For instance, the previous version of the TSP is a decision problem in the sense that its answer is a yes or a no. However, another version of the TSP exists. It is the optimization version that consists of determining the shortest path from a city of origin and back to the same point after visiting each city only once for a given network. The optimization version of the TSP is a NP-hard problem, even though it is neither a decision problem nor a NP-complete problem. Similarly, most optimization versions of existing

NP-complete problems are NP-hard problems. Here are a few examples of some well-known NP-complete problems (Aho, 1977; Varadharajan, 2020; Zapata-Rivera & Aranzazu-Suescun, 2020):

- the satisfiability problem (SAT) to determine whether a given Boolean formula can be satisfied
- the partitioning problem to determine whether a given set of positive integers can be partitioned into two complementary subsets such that the sum of the elements of one subset is equal to the sum of the elements of the other subset
- the graph coloring problem to determine whether it is possible to assign different colors to each node of a graph while ensuring that two directly connected nodes do not have the same color and the total number of colors used does not exceed a given value

### 6.3 P = NP?

A hierarchy of inclusion of the various complexity classes was presented in the first section of this unit where we saw that  $P \subseteq NP$ . What remains an open question is whether  $P = NP$ . That question is so interesting for computer scientists that there is even a one million dollar reward for the first person to solve it (Blank, 2002).

The P versus NP question is also interesting because it is currently approached on the basis of the “educated” perceptions and beliefs of computer theory scientists instead of through systematic facts and proofs. In fact, surveys conducted by Gasarch (2002; 2012) found that the majority of eminent computer theory scientists think that  $P \neq NP$ . More precisely, Gasarch (2012) found that 83 percent of the surveyed scientists thought that  $P \neq NP$ , an increase from 61 percent in Gasarch (2002). Similar proportions are reported in both surveys about the beliefs of the respondents that a proof will be found for this question before the beginning of the twenty-second century. Interestingly, in 2002, 22 percent of the respondents reported that they did not know whether or not  $P = NP$ , but that number shrunk to a mere 0.6 percent by 2012. The 2002 survey also revealed that almost a fifth of respondents reported that they did not know when this problem would be solved, and almost the same proportion of respondents thought that it will be solved after 2100. In contrast, 41 percent of the respondents in 2012 believed that a solution would be found for this problem after 2100.

These results seem to indicate that it is currently widely believed that  $P \neq NP$ , but one of the follow-up questions is about the consequences of the opposite statement. In other words, how will a  $P = NP$  proven statement challenge the status quo? If there is a proof that  $P \neq NP$ , then such a proof can be used to solve other relevant open questions. Nevertheless, a  $P = NP$  proof will release a gigantic algorithmic power in the sense that the entire existing class of NP problems that are currently considered as intractable (not realistically solvable by an efficient algorithm) will become tractable (realistically solvable by an efficient algorithm). A  $P = NP$  proof will, however, destroy

existing cryptography systems that currently rely on the fact that it is hard to break the primary and private keys because the factorization problem is an NP problem (Rodó, 2010).

### Summary

This unit began with the presentation of the big O time and space approximation model as the commonly used complexity model for the measurement of the efficiency of algorithms and for the analysis of the complexity of computational problems. This presentation is illustrated with relevant examples for the best, average, and worst case scenarios, as well as for the identification of the complexity classes of computational problems. The rest of the unit is dedicated to the concept of NP-completeness, with the presentation of the central concept of reducibility with examples of both NP-complete problems and NP-hard problems. The unit ends with a presentation of the perceptions of eminent computer theory scientists on the question of whether  $P = NP$ .

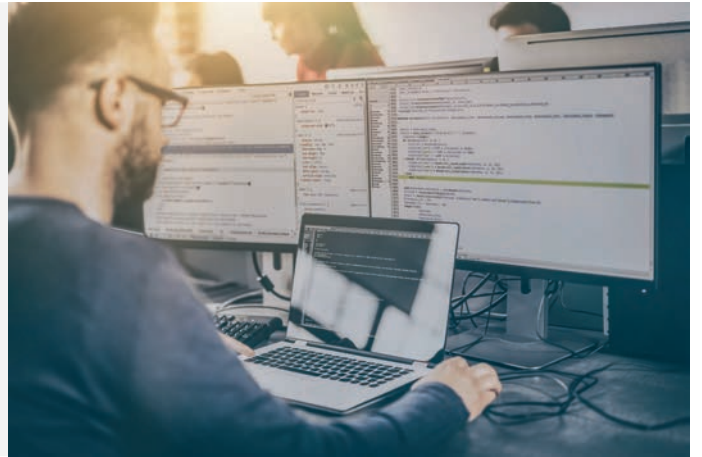
### Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

# Unit 7



## Advanced Algorithmics

### STUDY GOALS

On completion of this unit, you will have learned ...

- ... the fundamental constructs of parallel algorithms.
- ... how to program basic probabilistic algorithms.
- ... core quantum computing concepts.
- ... the five steps of Shor's algorithm.

# 7. Advanced Algorithmics

## Introduction

The world of computing is an ever evolving domain where novel and diverse paradigms are regularly proposed in the quest for new solutions to existing computational problems and for the advancement of computing sciences and technologies. This unit is an introduction to alternative advanced algorithmic approaches such as parallel algorithms, probabilistic algorithms, and quantum computing. These interesting approaches are considered important alternative solutions to some of the most difficult computational problems. The unit will present Shor's algorithm as an important application of quantum computing. It will also use other relevant examples to outline the fundamental concepts of parallel algorithms and introduce probabilistic programming.

## 7.1 Parallel Computing

This section is an introduction to parallel algorithms with an overview of their general concepts together with suitable NodeJS parallel programming examples.

### Parallel Algorithms General Concepts

Let us recall that an algorithm is a set of steps to be executed as a solution to a given computational problem, and let us denote such steps as  $s_0, s_1, s_2, \dots$ , and  $s_n$ . The most common approach for the execution of such steps is the sequential or serial execution whereby  $s_0$  is executed first, then  $s_1, s_2$ , and so on. The sequential approach works well in instances where the execution of one step depends on the outcome of the execution of the preceding one. There are, however, other instances where two steps can be executed at the same time or in parallel simply because they use different data inputs. The main advantage of this parallel approach is its time efficiency. In fact, the time taken for the parallel execution of two steps is equal to the time taken for the execution of the longest of these two steps. On the other hand, the sequential execution of two steps is equal to the addition of the execution times of the two steps. It is also important to note that sequential execution is unavoidable in situations where the executing computer only has one processor as was the case in the early days of computing. On the contrary, the emergence of multiprocessor computers and of distributed computing makes it possible for the different processors of the same computer (or from different machines) to execute different instructions in parallel. Nowadays, these multiprocess infrastructures are cleverly used by the parallel features of many modern programming languages.

The parallel execution of instructions by the different processors of the same computer can be referred to as multiprocessor parallel programming. This is different from multi-computer parallel programming where different computers execute in parallel the different processes of the same task. In both cases, parallel programming must take into account situations where parallel tasks are not totally independent. In fact, parallel

tasks sometimes rely on, synchronize with, and communicate with one another either through message-passing or through the use of a shared memory. Such a synchronization often requires the use of **semaphores** or locks that are acquired and orderly released for the management of interesting situations such as mutual exclusion, critical sections, and **deadlocks**, especially for the integrity of a shared memory. Besides synchronization, partitioning and scheduling are two other important aspects of parallel algorithms.

### Data partitioning versus function partitioning

Let us suppose that we have some input data  $d$  for a given algorithm  $f$  that we would like to parallelize with a certain number of parallel tasks. A key question is to decide what will constitute a parallel task. One method is the data partitioning approach which consists of dividing the original dataset  $d$  into many data partitions or parts such that a parallel task can be created for each data partition. Here, the different parallel tasks are doing the same job but on different data. Another method, the function partitioning approach, consists of dividing or partitioning the original algorithm  $f$  into different sub-functions that can be executed in parallel on the original dataset  $d$ . In both cases, the number of partitions determines the granularity of the parallel algorithm which can either be classified as coarse-granularity for large parallel tasks, or as fine-granularity for small parallel tasks.

### Scheduling

Once the different parallel tasks have been designed, they must be assigned to different processors or computers for their parallel execution. This assignment process is also known under the name of scheduling where each processor or computer is assigned a schedule of parallel tasks. This assignment process can either be dynamic or static. In dynamic scheduling, the assignment of tasks to processors or to computers is decided during the execution of the parallel tasks. In contrast, static scheduling allows program designers to specify within their parallel programs how the parallel tasks will be assigned to the different processors or computers.

## Parallel Programming Building Blocks and Examples

There are a number of parallel building blocks or features that are available in modern programming languages for the design of parallel programs. The parallel programming features presented here are mostly applicable to data parallel algorithms. Some of these features include elementwise operations, reductions, and broadcasting. These features will be presented in the following subsections with relevant NodeJs illustrative examples using the `ParallelJs` library. Other parallel JavaScript libraries include `Hamsters.js`, `Threads.js`, and `Parallel.es` (Reiser, 2017). `ParallelJs` has a relatively small number of functions but these functions are powerful enough to implement key parallelism features in JavaScript. Readers are advised to install the `ParallelJs` library, for instance, with the help of the `npm install paralleljs` command should they want to explore and test the given examples in NodeJs.

### Semaphore

A semaphore is a data structure that is acquired and on which one can wait for the management of a shared resource.

### Deadlocks

These occur when a process is holding a shared resource while waiting for a shared resource held by another process that is also waiting for a shared resource.

### Elementwise operations

Elementwise operations allow the parallel execution of the same operation on each of the data partitions. This is visible in the [following ParallelJs example](#) where the `cubeOperation` operation is carried out in parallel by each element of the `lst` sequence. The final output of the program appears below it.

#### ParallelJs Map Operation First Example with Output Screenshot

```
let Parallel = require('paralleljs');
let lst = [1, 3, 5, 7];
let p = new Parallel(lst);
let log = function () {console.log(arguments[0]);};
let cubeOperation = function (n) {return n*n*n;};
p.map(cubeOperation).then(log);
```

```
[ 1, 27, 125, 343 ]
```

The first line of this program simply indicates that the program requires the use of the ParallelJs library. The second line declares an array `lst` with four integers. The list constitutes the data of the program, and it is partitioned into four elements. The third line of the program creates a ParallelJs object `p` for the parallel processing of the `lst` data. The fourth line is a modification of the behavior of the `log` method so that it can simply print the content of the `arguments` array that contains the current results of the parallel processing of the partitioned data. The `cubeOperation` function simply returns the value of the cube of its parameter. The last line of the program uses the ParallelJs `map` function for the parallel execution of the `cubeOperation` function on each of the elements of the `p` ParallelJs object such that the subsequent array of results can be printed by the `log` function.

Let us consider another little more elaborate example to illustrate the use of the concept of elementwise operations by the ParallelJs `map` function. It is the example of the pairwise multiplication of a row of values by a column of values. Assuming, for example, that the elements of a row are presented from the left to the right and the ones of the column are presented from the top to the bottom, the pairwise multiplication of the row `[1,2,3,4]` by the column `[5,6,7,8]` will yield the value `[(1*5), (2*6), (3*7), (4*8)]` which is equal to the array `[5, 12, 21, 32]`.

A screenshot of the output of the [following program](#) is displayed below that program for input row `[1,2,3,4]` and for input column `[5,6,7,8]`. Readers are advised to test the program and trace it.



### ParallelJs Map Operation Second Example

```
let readlineSync = require('readline-sync');
let Parallel = require('paralleljs');
let pairRoAndColValues = function(ro,cl){
  let rcpa = new Array();
  for (let i=0; i<ro.length; i++){
    let rcv = {rowV: Number(ro[i]),colV: Number(cl[i]),}
    rcpa.push(rcv);
  }
  return(rcpa);
}
let pMultRC = function(xy) {return((xy.rowV)*(xy.colV));}
let log = function () {console.log(arguments[0]);}
let row = new Array(); let col = new Array();
let mess = 'values separated by commas please: ';
row = (readlineSync.question('Row ' + mess)).split(',');
col = (readlineSync.question('Column ' + mess)).split(',');

if (row.length === col.length){
  let rcpairs = new Array(); rcpairs = pairRoAndColValues(row,col)
  console.log('Row and column paired'); console.log(rcpairs);
  let p = new Parallel(rcpairs);
  console.log('Pairwise multiplication of the row and the column');
  p.map(pMultRC).then(log);
}
else {
  console.log('Rows and columns with different nb of elements');
}
```

### Output Screenshot of the Second ParallelJs Map Operation Example

```
Row values separated by commas please: 1,2,3,4
Column values separated by commas please: 5,6,7,8
Row and column paired
[
  { rowV: 1, colV: 5 },
  { rowV: 2, colV: 6 },
  { rowV: 3, colV: 7 },
  { rowV: 4, colV: 8 }
]
Pairwise multiplication of the row and the column
[ 5, 12, 21, 32 ]
```

The first two lines of the above program simply declare the two libraries that are used by the program. Let us recall that the `ParallelJs` object's constructor uses an array of values as its main parameter, but our data is made up of two arrays which respectively represent a row of values and a column of values. One way to merge these two arrays into a single array is to pair their elements such that the first element of the row is paired with the first element of the column, the second element of the row is paired with the second element of the column, the third element of the row is paired with the third element of the column, and so on. Such pairs can be constructed as objects with two attributes where the first attribute is the value from the row and the second one is the value for the column.

Returning to our example on the pairwise multiplication of the row [1,2,3,4] by the column [5,6,7,8], the pairwise alignment of these two arrays is equal to the array `[{rowV:1, colV:5}, {rowV:2, colV:6}, {rowV:3, colV:7}, {rowV:4, colV:8}]`.

This array has four objects which will be processed in parallel by the `ParallelJs` `map` elementwise operation as applied to the `pMultRC` function. It is easy to see that the `pMultRC` function takes in an object that is made up of a `rowV` attribute and a `colV` attribute, and it returns the multiplication of the value of `rowV` by the value of `colV`. As for the `pairRoAndColValues` function, its role is to perform and return the pairwise alignment of its two input arrays. Readers are also invited to write the sequential versions of the different examples as a way of comparing parallel programming with sequential programming.

### Reductions

The result of the parallel execution of partitioned data is usually stored in an array (seen above), and the purpose of the reduction operation is to perform an action that can cumulate or reduce all the values of that array into a single scalar or atomic value. For example, the result of the sum reduction of the array [10, 20, 30, 40] is 100. We use the term sum reduction here to say that we are reducing the array [10, 20, 30, 40] to a single scalar value by summing all its values.

The combination of the reduce operation with the map operation is highly credited for its ability to optimize parallelism, for example, for graphics programming: "Map Reduce is an ideal abstraction for programming general purpose computations on the graphics processor. Structuring a computation as stages of Map Reduce operations ensures that maximal parallelism is expressed" (Catanzaro et al., 2008, Section 3).

Returning to our example of the multiplication of a row by a column, and keeping in mind that the program has already shown us how to store the pairwise multiplication of a row and a column into an array, the sum reduction of that array is the final result of the multiplication of the row by the column. This is demonstrated by making the following changes in the above program:

- Add the following function to your program:  

```
function addAll(d) {return d[0] + d[1];}
```
- Replace `console.log('Pairwise multiplication of the row and the column')` with the following instruction:

```
console.log('Multiplying row and column: The final result is');
```

- Replace `p.map(pMultRC).then(log)` with the following instruction:  
`p.map(pMultRC).reduce(addAll).then(log);`

Below is a screenshot of the output of the updated program using the input row [1,2,3,4] and the input column [5,6,7,8]. This example shows that the multiplication of the row [1,2,3,4] by the column [5,6,7,8] is the sum of all the values of the array [(1\*5), (2\*6), (3\*7), (4\*8)], in other words 5 + 12 + 21 + 32, which is ultimately equal to 70. The sum is the outcome of the `addAll` sum reduction operation as applied to the array resulting from the `pMultRC` pairwise multiplication operation.

#### Output Screenshot of the Paralleljs Map Reduce Example

```
Row values separated by commas please: 1,2,3,4
Column values separated by commas please: 5,6,7,8
Row and column paired
[
  { rowV: 1, colV: 5 },
  { rowV: 2, colV: 6 },
  { rowV: 3, colV: 7 },
  { rowV: 4, colV: 8 }
]
Multiplying row and column: The final result is
70
```

### Broadcasting

The purpose of broadcasting operations is to distribute a given value to different parallel processes for their internal use. We will illustrate that concept with the [following example](#) of the multiplication of two matrices.

## ParallelJs Broadcasting Operation Example (Start)

```
let readlineSync = require('readline-sync');
let Parallel = require('paralleljs');
let assert = require('assert');

let log = function(){
  console.log('Result column after column');
  console.log(arguments[0]);
}

let printMatr = function (A){
  for (let r=0; r<A.length; r++){
    let rs = "";
    if (r!==0) {rs = '\n';}
    for (let c=0; c<A[r].length; c++){
      if (c===0) {rs = rs + A[r][c];}
      else {rs = rs + "  " + A[r][c];}
    }
    console.log(rs);
  }
}

function leftMatr(){
  return([[0,1,2,3,4],[5,6,7,8,9],[1,2,3,4,5]]);
}

let multByLeftMatr = function(C){
  let P = new Array(); let L = new Array(); L = leftMatr();
  let s = 0;
  for (let i=0; i<L.length; i++){
    s=0;
    for (let j=0; j<C.length; j++){
      s = s + (L[i][j]*C[j]);
    }
    P.push(s);
  }
  return P;
}

console.log('Multiplying the left matrix below with yours');
printMatr(leftMatr());
```

### ParallelJs Broadcasting Operation Example (End)

```

let mess1 = 'Nb of columns of your matrix please : ';
let lc = (leftMatr()[0]).length;
let mess2 = lc + ' values separated by commas please : ';

let cn = parseInt(readlineSync.question(mess1));
let cols = new Array();
for (let i=0; i<cn; i++){
  let col = new Array();
  let mess3 = 'For column no' + (i+1) + ' , ' + mess2;
  col = (readlineSync.question(mess3)).split(',');
  assert(lc===col.length, lc.toString() + " values please!");
  cols.push(col);
}

let r = new Parallel(cols);

r.require(leftMatr).map(multByLeftMatr).then(log);

```

In the example above, the first (left) matrix is constant, and it is broadcast to the different processes that will perform the multiplication of that matrix by the different columns of the second (right) matrix. This broadcasting operation is performed by the `require ParallelJs` method that distributes the outcome of the `leftMatr` function to all future parallel processes for their internal use. It is important to note that the purpose of the `leftMatr` function is simply to return the value of the left matrix in order for it to be broadcast to the parallel processes that will use it in the `multByLeftMatr` function. The use of the ParallelJs elementwise `map` operation allows the creation of a set of parallel processes where the left array multiplies each column of the right array:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \times \begin{pmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 7 & 9 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 37 & 54 \\ 132 & 184 \\ 56 & 80 \end{pmatrix}.$$

Below is a screenshot of the output of the above program for the matrices multiplication example. Readers are advised to test and trace the program with this example as well as with the others.

### ParallelJJs Broadcasting Operation Example

```

Multiplying the left matrix below with yours
0  1  2  3  4
5  6  7  8  9

1  2  3  4  5
Nb of columns of your matrix please : 2
For column no1 , 5 values separated by commas please : 2,4,6,7,0
For column no2 , 5 values separated by commas please : 3,5,7,9,2
Result column after column
[ [ 37, 132, 56 ], [ 54, 184, 80 ] ]

```

## 7.2 Probabilistic Algorithms

Probabilistic algorithms  
These algorithms create and process random data for various probability distributions.

The purpose of **probabilistic algorithms** is to resolve computational problems on the creation and the processing of random data. Such data randomizations are based on the use of common probability distributions. This section is an introduction to the programming of probabilistic algorithms using the WebPPL JavaScript probabilistic language. Other probabilistic programming languages include Hakaru, Augur, R2, Figaro, IBAL, PSI, Church, Anglican, BLOG, Turing.jl, BayesDB, Venture, Probabilistic-C, CPProb, Biips, LibBi, Birch, STAN, JAGS, and BUGS (van de Meent et al., 2018).

WebPPL can be installed in NodeJS with the `npm install -g webppl` command, and we will use the following [HelloTest.wppl](#) to illustrate how to test it. Readers are advised to install WebPPL, type the following few lines in a text editor, and save their file under the name `HelloTest.wppl` before executing the `webppl HelloTest.wppl` command.

### WebPPL Hello Mam or Dad Example

```

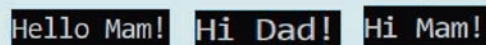
let greeting = function () {return flip(.5) ? "Hello" : "Hi"}
let aParent = function () {return flip(.5) ? "Mam" : "Dad"}
let phrase = greeting() + " " + aParent() + "!"
phrase;

```



The following three screenshots of the output of three executions of the above program illustrate the randomness of the data generated by that program. In fact, the `greeting` function randomly returns either “Hello” or “Hi” with an equal “fifty-fifty” probability or chance. Similarly, the `aParent` function randomly returns either “Mam” or “Dad” with an equal “fifty-fifty” probability or chance.

#### Output Screenshot of the WebPPL Hello Mam or Dad Example



Three screenshots of program output, each in a black box with white text: "Hello Mam!", "Hi Dad!", and "Hi Mam!".

First, let us review the programming of the listed probability distributions. We apply these distributions later to the example of the visualization of an algorithm of average time efficiency. Next, we introduce the programming of common descriptive statistics used by probabilistic algorithms and hypothesis testing.

### Generation of Random Data for Common Probability Distributions

The following probability distributions are introduced in this subsection: Bernoulli, binomial, geometric, and uniform distributions.

#### The Bernoulli distribution

The Bernoulli distribution is that of a single event whose outcome is randomly chosen with a given probability between two possible values. A common example of such a random choice is the flip of a coin that has a 50 percent chance (probability) to land on either side. It is, of course, always possible to set the value of the probability of a given distribution to any number between zero and one. The `flip` function allows WebPPL programmers to generate data according to the Bernoulli distribution as was the case for the above `HelloTest.wppl` example.

That `flip` function receives a probability value as a parameter but that value is assumed to be equal to 0.5 when absent. The [following example](#) uses the `flip` function inside the `random0To4` function to generate a random number between 0 and 4.

#### WebPPL Bernoulli Distribution First Example

```
let random0To4 = function(){
  return flip(0.4) + flip(0.5) + flip(0.4) + flip(0.5)
};
random0To4();
```

Here, it is acknowledged that the `flip` function can only return 0 or 1 and adding four calls of the `flip` function can yield any natural number between 0 and 4.

This program can be slightly amended to generate six random natural numbers that are each between 0 and 4.

#### WebPPL Bernoulli Distribution Second Example

```
let random0To4 = function(){
  return flip(0.4) + flip(0.5) + flip(0.4) + flip(0.5)
};
repeat(6, random0To4);
```

#### The binomial distribution

The binomial distribution represents the number of successes in a series of Bernoulli events assuming that the outcome of each Bernoulli event is either a success (with a given probability) or a failure (with the complementary probability). In the [following program](#), the `b0To10` variable uses the Binomial constructor to represent the numbers of successes for a series of ten “fifty-fifty” chance Bernoulli events.

The purpose of the `randomSampleB0To10` function is to generate a sample of the binomial object `b0To10` as a single random value between 0 and 10. Seven of such random values are generated by the `randomSampleOf7ValuesB0To10` function.

#### WebPPL Binomial Distribution Example

```
let b0To10 = Binomial({p: 0.5, n:10})
let randomSampleB0To10 = function(){return sample(b0To10)};
let randomSampleOf7ValuesB0To10 = function() {
  return repeat(7,randomSampleB0To10)
};

randomSampleOf7ValuesB0To10();
```

#### The geometric distribution

The geometric distribution represents the number of attempts by a series of Bernoulli events before its first success, assuming that the outcome of each Bernoulli event is either a success (with a given probability) or a failure (with the complementary probability). WebPPL does not have a constructor for the geometric distribution. Instead, that distribution can be translated into a recursive function that uses the `flip` function of the Bernoulli distribution. The code can be found [here](#).



### WebPPL Geometric Distribution Example

```
let geomF = function (p) {flip(p) ? 0 : 1 + geomF(p)};;  
let randomG = function() {return geomF(0.1)};;  
  
repeat(10, randomG);
```

The `geomF` function of the program returns 0 when the Bernoulli event is successful; otherwise, it increments the number of unsuccessful attempts by one before starting a new Bernoulli event.

We have biased each Bernoulli event by making the `randomG` function call the `geomF` function with an argument of 0.1 so that it can only have a 10 percent probability of success. This was done for us to witness higher values for the `geomF` function. Here is a screenshot of an example of execution of the above program as yielded by its `repeat(10, randomG)` instruction.

### Output Screenshot of WebPPL Geometric Distribution Example

```
[  
  14, 2, 19, 9, 45,  
  6, 4, 13, 15, 8  
]
```

### The uniform distribution

The uniform distribution can either be discrete or continuous. A discrete uniform distribution represents a single event whose outcome is a randomly chosen integer or value from a given range of values. As for the continuous uniform distribution, it represents a single event whose outcome is a randomly chosen real number from a given range of values. WebPPL uses the `RandomInteger` constructor for discrete uniform distributions while the `Uniform` constructor is used for continuous uniform distributions as visible.

### WebPPL Uniform Distribution Example

```
let du = RandomInteger({n:50})
let cu = Uniform({a:-3, b:1})
let rUDSf = function() {return sample(du)};
let rCDSf = function() {return sample(cu)};
let mess1 = 'Randomly chosen integer below fifty : '
let mess2 = '\nAnd randomly chosen real from -3 to one : '
let mess3 = mess1 + rUDSf() + mess2 + rCDSf()

mess3
```

This [program](#) chooses two random numbers: one random integer between 0 and 50 and one random real value between  $-3$  and 1.

### Output Screenshot of WebPPL Uniform Distribution Example

```
Randomly chosen integer below fifty : 23
And randomly chosen real from -3 to one : 0.8341691450519422
```

## Descriptive and Inferential Statistics

This subsection is an introduction to the programming of basic descriptive statistics such as the computation of means, variances, and standard deviations. Inferential hypothesis testing is also introduced in this subsection with the example of the computation of the Bayes factor.

### Descriptive statistics

Means, variances, and standard deviations are common in probabilistic algorithms. The [following program](#) is an illustration of how to compute these in WebPPL.

### WebPPL Descriptive Statistics Example (Start)

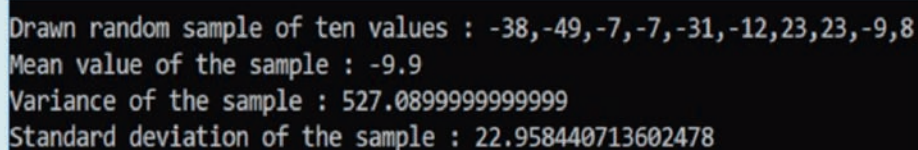
```
let du = RandomInteger({n:50})
let rUDSf = function() {
  return (flip(.5) ? sample(du) : ((-1)*sample(du)))
};
let rAr10 = function() {return repeat(10,rUDSf)};
let a = rAr10()
```

### WebPPL Descriptive Statistics Example (End)

```
let ms1 = '\nDrawn random sample of ten values : ' + a
let ms2 = '\nMean value of the sample : ' + listMean(a)
let ms3 = '\nVariance of the sample : ' + listVar(a)
let ms4 = '\nStandard deviation of the sample : ' + listStdev(a)
let ms5 = ms1 + ms2 + ms3 + ms4
```

```
ms5
```

### Output Screenshot of WebPPL Descriptive Statistics Example



```
Drawn random sample of ten values : -38,-49,-7,-7,-31,-12,23,23,-9,8
Mean value of the sample : -9.9
Variance of the sample : 527.0899999999999
Standard deviation of the sample : 22.958440713602478
```

### Hypothesis testing and Bayesian inference

It is possible to customize a given probability distribution according to the context. For example, in the [following program](#), the `nullHypM` variable simply represents a binomial distribution for 200 attempts with a probability of 0.5. As for the `altHypM` variable, it represents a customized binomial distribution for 200 attempts but whose probability is a randomly chosen real number between 0 and 1 (See `p = uniform(0, 1)`). The value of the `altHypM` variable is inferred from 100 samples of the customized binomial distribution.

### WebPPL Inferential Statistics Example

```

let attempts = 200; let successes = 115;
let nullHypM = Binomial({p: 0.5, n: attempts})
let altHypM = Infer({method: "forward", samples: 100000},
  function(){let p = uniform(0, 1);return binomial(p, attempts)}
)
let nullHypBEF = Math.exp(nullHypM.score(successes))
let altHypBEF = Math.exp(altHypM.score(successes))
let bayesFactor_01 = "Bayes Factor : " + (nullHypBEF / altHypBEF)
bayesFactor_01

```

This program represents the simple scenario whereby a coin is flipped 200 times with 115 successes. Such observed data may lead to the null hypothesis `nullHypM` that the probability of one of the 200 flips to succeed is 0.5, as represented by the `Binomial({p: 0.5, n: attempts})` distribution where the value of the `attempts` variable is 200. An alternative hypothesis with the name of `altHypM` counters the above `nullHypM` hypothesis by stating that the probability of one of the 200 flips to succeed can take any value between 0 and 1, as explained above. The Bayes factor can be used to decide which of the hypotheses is supported by the observed data, i.e., the actual observed number of successes. A Bayes factor above 1.6 indicates that the hypothesis on its numerator (null hypothesis) is the one that should be accepted, while a Bayes factor below 1.6 indicates that the hypothesis on its denominator (alternative hypothesis) is the one that should be accepted. This program was executed a few times, and its output always yielded a value below 1.6 (see screenshots below), indicating that the null hypothesis of  $p = 0.5$  cannot be accepted.

### Output Screenshot of WebPPL Inferential Statistics Example

Bayes Factor : 1.1840739940828027	Bayes Factor : 1.2105471931374985
Bayes Factor : 1.1475707495638716	Bayes Factor : 1.2356622801320534
Bayes Factor : 1.198368649946981	Bayes Factor : 1.246002550258681

### Visualizing the Effect of Uniform Distributions on Efficiency Estimates

The concept of the average efficiency of an algorithm can be suitably understood under the assumption that its input data is uniformly randomly distributed. We will illustrate the suitability of the uniform random distribution with the example of a naive linear

algorithm that seeks to identify the first number that is greater than or equal to 50 in an array  $A$  of  $n$  whole numbers. It is assumed that the value of each element of  $A$  is less than 100. In the best case, the first element of  $A$  is greater than or equal to 50, and it will only take one step for the algorithm to find it. In the worst case, no element of  $A$  has a value greater than or equal to 50, and it will take  $n$  steps for the algorithm to see that. In the average case, the data of  $A$  will be uniformly randomly distributed between 0 and 99. So, how many steps will the algorithms take to find the searched item in the average case? In other words, what is the efficiency of the algorithm in the average case scenario? This question is answered by the following `webppl` programs. The first program gives a textual output, and the second uses `plotly` for the visualization of its graphical output. Both of these programs collect the value of  $n$  in the form of an argument (`argv.n`) on the command line when running `webppl` for their execution.

### Program with a textual output

This program creates array of  $n$  uniformly distributed random whole numbers with none of the numbers greater than 99 (See `udpns`, `udp1s`, and `udp`). The `stepsToFind1stPassF(a)` function returns the first position in the array `a` where a value greater or equal to 50 is found. This function returns 0 if `a` does not have any value greater than or equal to 50. Apart from the  $n$  argument, this program also uses a second argument denoted by `mode` that can take three possible values: `ad`, `mn`, or `mr`. In the `ad` mode, the program outputs the details of each randomly generated array together with the number of steps used to find its first number with a value greater than or equal to 50. The code below can be found [here](#).

## WebPPL Text Version Search Example with Uniformly Distributed Input (Start)

```

let udp = RandomInteger({n:100});

let udp1s = function() {
  return (sample(udp))
};

let udpns = function() {
  return repeat(argv.n,udp1s)
};

let stepsToFind1stPassF = function(a){
  if(a.length===0){return(0)}
  if(a.length>0){
    if (a[0]>=50){return(1)}
    else {
      if (stepsToFind1stPassF(a.slice(1))===0){return(0)}
      else {return(1+stepsToFind1stPassF(a.slice(1)))}
    }
  }
};

let randomSearchRun = function(){
  let a = udpns();
  if(argv.mode === 'ad'){
    if(stepsToFind1stPassF(a)===0){
      return ([a.toString(),1+a.length]);
    }
    else {return ([a.toString(),stepsToFind1stPassF(a)]);}
  }
  if((argv.mode === 'mr') ||(argv.mode === 'mn')){
    if(stepsToFind1stPassF(a)===0) {return (1+a.length);}
    else {return (stepsToFind1stPassF(a));}
  }
};

let cRandomSearchRuns = function(c){
  let steps21stPass = function() { randomSearchRun();}
  if(argv.mode === 'ad'){return (repeat(c,steps21stPass))}
  if(argv.mode === 'mr'){
    return [c, listMean(repeat(c,steps21stPass))]
  }
  if(argv.mode === 'mn'){
    return [c, Math.round(listMean(repeat(c,steps21stPass)))]
  }
};

```



## WebPPL Text Version Search Example with Uniformly Distributed Input (End)

```

let emptyArray = function(){return([])}

let builtArray = function(n,nmax){
  if (n===nmax){return(emptyArray().concat([n]))}
  else{
    return((emptyArray().concat([n])).concat(builtArray(n+10,nmax)))
  }
}

let randomSearchRunsOnMoreAndMore = function(){
  if(argv.mode === 'ad'){return(cRandomSearchRuns(argv.n))}
  if((argv.mode === 'mr') ||(argv.mode === 'mn')){
    map(cRandomSearchRuns,builtArray(10,200))
  }
}

randomSearchRunsOnMoreAndMore();

```

Let us trace the above program when it is in the ad mode. The following functions are called one after the other.

- `randomSearchRunsOnMoreAndMore()` simply calls the following function.
- `cRandomSearchRuns(argv.n)` simply calls the following function.
- `repeat(argv.n, steps21stPass)` calls `steps21stPass` `n` times
- `steps21stPass`: simply calls the following function.
- `randomSearchRun()` creates a random array `a` of `n` values by calling `udpns (a=udpns())`, calls the `stepsToFind1stPassF(a)` function to locate the first element of `a` with a value greater than or equal to 50, and returns the value of the array and the found location.

Assuming that the program is named `unin.wppl`, we can call it in the ad mode with the `webppl unin.wppl -- --n 10 --mode ad` command that will yield an output similar to the following screenshot.

## Detailed Array Output of WebPPL Uniform Input Distribution Text Version Example

```
[
  [ '19,27,17,49,36,50,44,35,42,82', 6 ],
  [ '57,45,12,83,20,82,33,81,79,33', 1 ],
  [ '40,98,17,59,9,47,41,43,38,75', 2 ],
  [ '0,48,94,37,42,3,89,79,12,73', 3 ],
  [ '94,33,22,73,11,86,89,55,29,77', 1 ],
  [ '25,44,96,42,6,26,91,38,58,68', 3 ],
  [ '44,15,28,52,42,35,67,93,16,10', 4 ],
  [ '19,16,5,34,50,30,90,58,48,17', 5 ],
  [ '13,30,70,27,17,8,31,67,2,85', 3 ],
  [ '45,26,38,92,83,82,69,46,1,44', 4 ]
]
```

The `mr` and `mn` modes are slightly different from the `ad` mode, stemming mainly from the following instructions:

- `map(cRandomSearchRuns,builtArray(10,200))`: The function `cRandomSearchRuns` is called for each multiple of 10 between 1 and 200 and executes the following instruction where `c` is a multiple of 10 between 1 and 200.
- `return[c,listMean(repeat(c,steps21stPass))]`: Here, `c` random arrays of `n` elements are generated. Each of these will have a calculated value of the position of the first element greater than or equal to 50. It is the average value of these positions that is returned here together with the value of `c`.

The only difference between `mr` and `mn` is that `mr` rounds its results to the nearest natural value while `mn` does not.

Assuming that the above program is named `unin.wppl`, we can call it in the `mr` or in the `mn` mode with the `webppl unin.wppl -- --n 10000 --mode mr` or `unin.wppl -- --n 10000 --mode mn` command, respectively.



## Real Mean Output of WebPPL Uniform Input Distribution Text Version Example

```
[
  [ 10, 1.4 ],
  [ 20, 1.8 ],
  [ 30, 1.5666666666666667 ],
  [ 40, 2 ],
  [ 50, 2.24 ],
  [ 60, 1.85 ],
  [ 70, 1.7857142857142858 ],
  [ 80, 2.025 ],
  [ 90, 1.7 ],
  [ 100, 2.21 ],
  [ 110, 2.1454545454545455 ],
  [ 120, 1.9416666666666667 ],
  [ 130, 2.1538461538461537 ],
  [ 140, 1.9285714285714286 ],
  [ 150, 1.9466666666666668 ],
  [ 160, 2.1125 ],
  [ 170, 1.9588235294117646 ],
  [ 180, 2.2277777777777778 ],
  [ 190, 2.0105263157894737 ],
  [ 200, 1.96 ]
]
```

## Natural Mean Output of WebPPL Uniform Input Distribution Text Version Example

```
[
  [ 10, 2 ], [ 20, 2 ],
  [ 30, 2 ], [ 40, 2 ],
  [ 50, 2 ], [ 60, 2 ],
  [ 70, 2 ], [ 80, 2 ],
  [ 90, 2 ], [ 100, 2 ],
  [ 110, 2 ], [ 120, 2 ],
  [ 130, 2 ], [ 140, 2 ],
  [ 150, 2 ], [ 160, 2 ],
  [ 170, 2 ], [ 180, 2 ],
  [ 190, 2 ], [ 200, 2 ]
]
```

### Program with a graphical output

This subsection exclusively considers the `mr` mode of the above program and modifies it with the purpose of showing a visual representation of the final array generated by that program. Each element of that array is made up of two numbers that will be considered the *x* and *y* coordinates of a point. These points are visualized here. In the program above, the data of the final array targets the multiples of 10 less than or equal to 200, but the program below extends its data to the sequence of natural numbers less than or equal to 1000.

#### WebPPL Graphic Version Search Example with Uniformly Distributed Input (Start)

```

let udp = RandomInteger({n:100});

let udp1s = function(){
  return (sample(udp))
};

let udpns = function(){
  return repeat(argv.n,udp1s)
};

let stepsToFind1stPassF = function(a){
  if(a.length===0){return(0)}
  if(a.length>0){
    if (a[0]>=50){return(1)}
    else {
      if (stepsToFind1stPassF(a.slice(1))===0){return(0)}
      else {return(1+stepsToFind1stPassF(a.slice(1)))}
    }
  }
};

let randomSearchRun = function(){
  let a = udpns();
  if(stepsToFind1stPassF(a)===0) {return (1+a.length);}
  else {return(stepsToFind1stPassF(a));}
};

let cRandomSearchRuns = function(c){
  let steps21stPass = function(){randomSearchRun();}
  return [c, listMean(repeat(c,steps21stPass))]
};

let emptyArray = function(){return([])}

```

**WebPPL Text Version Search Example with Uniformly Distributed Input (End)**

```
let builtArray = function (n,nmax){
  if (n===nmax){return(emptyArray().concat([n]))}
  else {return((emptyArray().concat([n])).concat(builtArray-
(n+1,nmax)))}
}

let randomSearchRunsOnMoreAndMore = function() {
  map(cRandomSearchRuns,builtArray(1,999))
}

"let pts = " + JSON.stringify(randomSearchRunsOnMoreAndMore())
+ ";\nmodule.exports.exPts = pts;";
```

The concept of mode does not exist in the above program because it exclusively focuses on the `mr` mode. The `webppl 12-probabilistic-efficiency-graph.wbpl -- -- n 100 > pts.js` command will allow the above program (`u12-probabilistic-efficiency-graph.wbpl`) to save its `pts` generated array of points in the `pts.js` file whose `pts` variable can be imported by another JavaScript program. It is precisely this importation mechanism that is used by the JavaScript program within the following HTML Web page. For the Web page to run, the `plotly` open source library must be loaded with `npm install plotly.js`.

This importation mechanism is used by the following program for the online `plotly` drawing of the content of the `pts` array. The online use of `plotly` requires a sign up on the `plotly nodeJs` website where you are issued a username and an API key. Those are the two `plotly` credentials that are used in the `plotly require` function in the following program. The other `require` instruction of that program simply imports the `pts` array from the above described `pts.js` file so that it can transfer that array to a sequence of points (`data`) to be drawn by `plotly`. The graph generated by this program is denoted by `uniDisAvgStp`, and the actual drawing is accomplished by the `plotly plot` function. Our [program code](#) is visible below.

### Plotly Uniform Input Distribution Example

```

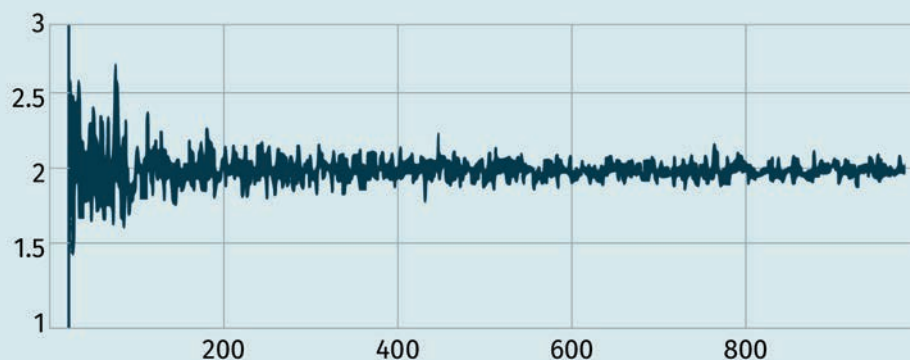
<!DOCTYPE html>
<html>
<head><title>Graphic</title><meta charset="utf-8"/>
  <script src="node_modules/plotly.js-dist/plotly.js"></script>
  <script src="pts.js"></script>
</head>
<div id="graphDiv"></div>
<script type="module">
  let x = [], y = []
  console.log("pts has " + pts.length + " entries.");
  pts.forEach(e => { x.push(e[0]); y.push(e[1]); });
  Plotly.newPlot('graphDiv', [{x: x, y: y, type: 'line'}],
    {title: "Search through a uniformly distributed dataset",
      xaxis: {title:"size of sample",
        yaxis: {title:"execution time", range:[0,5]} } );
</script>
</html>

```

The program above is simply executed with the `node plotlyDraw.js` command if one assumed that `plotlyDraw.js` is its name.

The url should show your plotly username on your screen. Please copy that url and paste it on the address bar of your Web browser and you will see a graph that is similar to the following.

### Output of Plotly Uniform Input Distribution Example (Screenshot)



The y-axis of the graph represents the average number of steps that it took to find the first element greater than or equal to 50 in an array of 100 values uniformly distributed between 0 and 99. The x-axis represents the number of times that an array was generated and searched. An interesting question is to find out why the above graph tends to follow the line with the equation  $y = 2$ .

## 7.3 Quantum Computing and the Shor Algorithm

A quantum computer is a computing machine that uses distinctive quantum physics and mathematics features to solve computational problems. A historical overview of quantum computing is hereby presented, followed by an introduction to some of the key concepts of quantum computing. The rest of the subsection is dedicated to the overview of the five phases of Shor's algorithm (Lomonaco, 2000).

### Genesis of Quantum Computing

According to Hidary (2019), the genesis of quantum computing can be traced back to 1979 when Paul Benioff suggested the idea of a theoretical paradigm for the construction of quantum machines. Yuri Manin was another of the quantum computing pioneers with the publication of his book on this topic in 1980 (Hidary, 2019). In 1981, the field of quantum computing came to the public eye when Nobel Prize winner Feynman revealed that the usual computing paradigm was not able to model many quantum physics features and proposed a set of desirable functionalities for quantum computers. Some quantum computing pioneers include David Deutsch, Richard Jozsa, Umesh Vazirani, Ethan Bernstein, Daniel Simon, Seth Lloyd, and Peter Shor.

### Complex Numbers, Vectors, and Matrices

Let us start by recalling a few fundamentals of complex numbers and matrices because of their importance in quantum computing. We use the example of the complex number  $c = 4 + 3i$  which is made up of the real part 4 and the complex part 3. The complex conjugate  $\bar{c}$  of  $4 + 3i$  is the complex number  $4 - 3i$ , and its modulus  $|c|$  is the square root of  $(4^2 + 3^2)$  which is equal to 5. Let's not forget that  $i^2 = -1$ . Quantum computing uses matrices of complex numbers to model solutions for computational problems. Here is an example of a matrix  $M$  of complex numbers followed by its transpose  $M^T$  (the first row becomes the first column, the second row becomes the second column, etc.), its conjugate  $\bar{M}$  (each element is replaced by its conjugate), and its adjoint  $M^\dagger$  (the conjugate of its transpose, or the transpose of its conjugate).

$$\begin{array}{cccc} \begin{pmatrix} -5 + 4i & 6 - 7i & 9 \\ 1 - 2i & -3 & -8i \end{pmatrix} & , & \begin{pmatrix} -5 + 4i & 1 - 2i \\ 6 - 7i & -3 \\ 9 & -8i \end{pmatrix} & , & \begin{pmatrix} -5 - 4i & 6 + 7i & 9 \\ 1 + 2i & -3 & 8i \end{pmatrix} & , & \begin{pmatrix} -5 - 4i & 1 + 2i \\ 6 + 7i & -3 \\ 9 & 8i \end{pmatrix} \\ \text{Matrix } M & & \text{Transpose } M^T & & \text{Conjugate } \bar{M} & & \text{Adjoint } M^\dagger \end{array}$$

The tensor product is also intensively used in quantum computing and can be defined as follows. Assuming that  $A$  is a matrix with  $m$  rows and  $n$  columns, we can denote by  $a_{i,j}$  the element on the  $i^{\text{th}}$  row and column  $j^{\text{th}}$  of  $A$ . The tensor product of  $A$  by another matrix  $B$  is a matrix denoted by  $A \otimes B$  and it is calculated as follows.

$$A \otimes B = \begin{pmatrix} a_{1,1}B & a_{1,2}B & a_{1,3}B & \cdots & a_{1,n-2}B & a_{1,n-1}B & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & a_{2,3}B & \cdots & a_{2,n-2}B & a_{2,n-1}B & a_{2,n}B \\ a_{3,1}B & a_{3,2}B & a_{3,3}B & \cdots & a_{3,n-2}B & a_{3,n-1}B & a_{3,n}B \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ a_{m-2,1}B & a_{m-2,2}B & a_{m-2,3}B & \cdots & a_{m-2,n-2}B & a_{m-2,n-1}B & a_{m-2,n}B \\ a_{m-1,1}B & a_{m-1,2}B & a_{m-1,3}B & \cdots & a_{m-1,n-2}B & a_{m-1,n-1}B & a_{m-1,n}B \\ a_{m,1}B & a_{m,2}B & a_{m,3}B & \cdots & a_{m,n-2}B & a_{m,n-1}B & a_{m,n}B \end{pmatrix}$$

It also seems important to recall the following formula and Dirac kets notations for the calculation of  $\|V\|$  which is the norm of a column vector  $V$  of complex numbers.

$$V = |V\rangle = [c_0, c_1, c_2, c_3, \dots, c_{n-3}, c_{n-2}, c_{n-1}]^T = \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-3} \\ c_{n-2} \\ c_{n-1} \end{pmatrix}$$

$$\langle V| = \overline{V}^T = [\overline{c_0}, \overline{c_1}, \overline{c_2}, \overline{c_3}, \dots, \overline{c_{n-3}}, \overline{c_{n-2}}, \overline{c_{n-1}}]$$

$$\|V\| = \||V\rangle\| = \sqrt{\langle V|V\rangle} = \sqrt{[\overline{c_0}, \overline{c_1}, \overline{c_2}, \overline{c_3}, \dots, \overline{c_{n-3}}, \overline{c_{n-2}}, \overline{c_{n-1}}] \times \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-3} \\ c_{n-2} \\ c_{n-1} \end{pmatrix}}$$

The formula is similar to the following from Strubell (2011, pp. 6–7):

$$\|V\| = \sqrt{|c_0|^2 + |c_1|^2 + |c_2|^2 + |c_3|^2 + \dots + |c_{n-3}|^2 + |c_{n-2}|^2 + |c_{n-1}|^2}.$$

## States and Dynamics

A quantum system is defined as a structure with a finite number of positions or states with the understanding that at the time when that structure is observed, only one of these positions or states is occupied by the proton, and each of these positions or states has its own probability to be occupied. This does not cancel out the main feature of quantum computing according to which the proton simultaneously occupies all its states when it is not being observed. If we assume that our quantum system has  $n$  positions or states, then those positions or states can be represented by the following column vector using the Dirac ket notation.

$$|\varphi\rangle = [c_0, c_1, c_2, c_3, \dots, c_{n-1}, c_{n-2}, c_{n-1}]^T$$

The above states can be normalized where it is clear that each value represents a probability between 0 and 1:

$$\frac{|\varphi\rangle}{\| |\varphi\rangle \|} = \left[ \frac{c_0}{\| |\varphi\rangle \|}, \frac{c_1}{\| |\varphi\rangle \|}, \frac{c_2}{\| |\varphi\rangle \|}, \frac{c_3}{\| |\varphi\rangle \|}, \dots, \frac{c_{n-3}}{\| |\varphi\rangle \|}, \frac{c_{n-2}}{\| |\varphi\rangle \|}, \frac{c_{n-1}}{\| |\varphi\rangle \|} \right]^T$$

The normalized observed state values show that at the time when the quantum system is being observed the sum of all the probabilities is equal to 1 with the probability of the proton being in state  $i$  calculated as:

$$\frac{|c_i|^2}{\| |\varphi\rangle \|^2}$$

It is important to note that the classical 0 and 1 bits are translated as follows in quantum computing so that any qubit  $[c_0, c_1]^T$  can be expressed in terms of the qubit of 0 and the qubit of 1.

$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$	$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$	$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = c_0 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + c_1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
$ 0\rangle$	$ 1\rangle$	$ c\rangle$	$ c\rangle = c_0  0\rangle + c_1  1\rangle$
Qubit of 0	Qubit of 1	A qubit $c$	Expression of $c^\dagger$



The observed values of the states of a quantum system continuously change as specified by an operator that is represented by a unitary square adjacency matrix of complex numbers whose number of columns (or rows) is equal to the number of states of the quantum system. A matrix  $M$  is said to be unitary when the  $M \times M^\dagger$  multiplication and the  $M^\dagger \times M$  multiplication are both equal to the identity matrix. This change process and example is presented below.

$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$	$\begin{pmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{pmatrix}$	$\begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \rightarrow \begin{pmatrix} U_{0,0} & U_{0,1} \\ U_{1,0} & U_{1,1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$	$\begin{pmatrix} U_{0,0}c_0 + U_{0,1}c_1 \\ U_{1,0}c_0 + U_{1,1}c_1 \end{pmatrix}$
$ \varphi\rangle$	$U$	$ \varphi\rangle \rightarrow U \varphi\rangle$	$U \varphi\rangle$
Original qubit	Adjacency	Qubit change process	Changed qubit

$\begin{pmatrix} \frac{-1+3i}{\sqrt{14}} \\ \frac{-2i}{\sqrt{14}} \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	$\begin{pmatrix} \frac{-1+3i}{\sqrt{14}} \\ \frac{-2i}{\sqrt{14}} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} \frac{-1+3i}{\sqrt{14}} \\ \frac{-2i}{\sqrt{14}} \end{pmatrix}$	$\begin{pmatrix} \frac{-1+3i}{\sqrt{14}} \\ \frac{2}{\sqrt{14}} \end{pmatrix}$
$ \varphi\rangle$	$U$	$ \varphi\rangle \rightarrow U \varphi\rangle$	$U \varphi\rangle$
Original qubit	Adjacency matrix	Qubit change process	Changed qubit

### An Overview of Shor's Algorithm

The purpose of Shor's algorithm is to solve the problem of the factorization of natural numbers. The efficiency of that algorithm is seen as a serious threat to the survival of current encryption systems that are based on the difficulty of computing the prime factors of numbers. Shor's algorithm is usually divided into five steps but it is only one of these steps that makes use of quantum computing paradigms. These five steps are briefly reviewed below for an input value  $N$ .

1. Randomly choose a natural number  $m$  between 2 and  $N - 1$  and calculate the gcd or greatest common divisor between  $m$  and  $N$ . If the value of the gcd is different from 1, then the algorithm should terminate with the answer that the gcd is a factor of  $N$ . If not, then the algorithm should move to the second step.
2. Use quantum computing to calculate the period  $p$  of the function  $f(n) = m^n \bmod N$ .

3. Determine whether  $p$  is even or odd. If  $p$  is even, then the algorithm should move to the fourth step, otherwise it should move to the first step.
4. Determine whether  $\frac{p}{2} + 1 = 0 \pmod{N}$ . If so, then the algorithm should move to step one, otherwise it should move to step five.
5. Calculate  $d$  as the highest common divisor between  $\frac{p}{2} - 1$  and  $N$ . The algorithm must then terminate by returning  $d$  as a non-trivial factor of  $N$ .

### Use of Quantum Computing by Shor's Algorithm

Let us end this section by briefly illustrating how Shor's algorithm uses quantum computing to calculate the period  $p$  of the function  $f(n) = m^n \pmod{N}$  with the example of the calculation of the period of the function  $f(x) = x \pmod{2}$ . A value  $r$  is a period of  $f$  if and only if  $f(x) = f(x + r)$  for all  $x$  values. First, we need to introduce Quantum Fourier Transformation (QFT) matrices because of their use by this algorithm. We also must remember that  $e^{pi} = -1$ . Let us also denote the imaginary number  $e^{\frac{2\pi i}{N}}$  by the name  $w$  where  $N$  is a natural number.

The  $\text{QFT}_N$  matrix is defined as follows:

$$\text{QFT}_N = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & w & w^2 & w^3 & \dots & w^{N-2} & w^{N-1} \\ 1 & w^2 & w^4 & w^6 & \dots & w^{2N-4} & w^{2N-2} \\ 1 & w^3 & w^6 & w^9 & \dots & w^{3N-6} & w^{3N-3} \\ \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 1 & w^{N-3} & w^{2N-6} & w^{3N-9} & \dots & w^{(N-2)N-3(N-2)} & w^{(N-1)N-3(N-1)} \\ 1 & w^{N-2} & w^{2N-4} & w^{3N-6} & \dots & w^{(N-2)N-2(N-2)} & w^{(N-1)N-2(N-1)} \\ 1 & w^{N-1} & w^{2N-2} & w^{3N-3} & \dots & w^{(N-2)N-(N-2)} & w^{(N-1)N-(N-1)} \end{pmatrix}$$

Let us review the examples of  $N = 2$ ,  $N = 4$ , and  $N = 8$ .

$$\text{QFT}_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & w \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & e^{\frac{2\pi i}{2}} \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

For  $N = 4$ , the value of  $w$  is equal to  $e^{\frac{2\pi i}{4}} = e^{\frac{\pi i}{2}}$ . The square of  $w$  will therefore yield the value  $e^{pi}$  which is also equal to  $-1$ . This implies that  $w = i$ . The  $\text{QFT}_4$  matrix can now easily be filled knowing that  $w = i$ ,  $w^2 = -1$ ,  $w^3 = -i$ ,  $w^4 = 1$ ,  $w^5 = i$ ,  $w^6 = -1$ ,  $w^7 = -i$ ,  $w^8 = 1$ ,  $w^9 = i$ .

$$\text{QFT}_4 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w^9 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

For  $N = 8$ , the value of  $w$  is equal to  $e^{\frac{2\pi i}{8}} = e^{\frac{\pi i}{4}}$ . Raising  $w$  to the power 4 will yield the value  $e^{\pi i}$  which is also equal to  $-1$ . This implies that  $w$  is equal to the square root of  $i$  which is also equal to  $\frac{1+i}{\sqrt{2}}$ , and  $\text{QFT}_8$  can be calculated as follows:

$$\text{QFT}_8 = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & w^8 & w^{10} & w^{12} & w^{14} \\ 1 & w^3 & w^6 & w^9 & w^{12} & w^{15} & w^{18} & w^{21} \\ 1 & w^4 & w^8 & w^{12} & w^{16} & w^{20} & w^{24} & w^{28} \\ 1 & w^5 & w^{10} & w^{15} & w^{20} & w^{25} & w^{30} & w^{35} \\ 1 & w^6 & w^{12} & w^{18} & w^{24} & w^{30} & w^{36} & w^{42} \\ 1 & w^7 & w^{14} & w^{21} & w^{28} & w^{35} & w^{42} & w^{49} \end{pmatrix} = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \sqrt{i} & i & w^3 & -1 & w^5 & -i & w^7 \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & w^3 & -i & w^9 & -1 & w^{15} & i & w^{21} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & w^5 & i & w^{15} & -1 & w^{25} & -i & w^{35} \\ 1 & -i & -1 & i & 1 & -i & -1 & -i \\ 1 & w^7 & -i & w^{21} & -1 & w^{35} & -i & w^{49} \end{pmatrix}.$$

We can now multiply  $\text{QFT}_8$  by the qubit  $|0\rangle |0\rangle = |0\rangle \otimes |0\rangle$  as shown below.

$$|0\rangle |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \end{pmatrix}$$

$$\frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \sqrt{i} & i & w^3 & -1 & w^5 & -i & w^7 \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & w^3 & -i & w^9 & -1 & w^{15} & i & w^{21} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & w^5 & i & w^{15} & -1 & w^{25} & -i & w^{35} \\ 1 & -i & -1 & i & 1 & -i & -1 & -i \\ 1 & w^7 & -i & w^{21} & -1 & w^{35} & -i & w^{49} \end{pmatrix} \times \begin{pmatrix} 1|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \\ 0|0\rangle \end{pmatrix} = \frac{1}{\sqrt{8}} \begin{pmatrix} 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \end{pmatrix}$$

$$\begin{pmatrix} 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \\ 1|0\rangle \end{pmatrix} = \begin{pmatrix} 1|0\rangle \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1|0\rangle \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1|0\rangle \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1|0\rangle \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1|0\rangle \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1|0\rangle \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1|0\rangle \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1|0\rangle \end{pmatrix}$$

The equation can be rewritten as

$$\begin{pmatrix} |1\rangle|0\rangle \\ |1\rangle|0\rangle \\ |1\rangle|0\rangle \\ |1\rangle|0\rangle \\ |1\rangle|0\rangle \\ |1\rangle|0\rangle \\ |1\rangle|0\rangle \\ |1\rangle|0\rangle \end{pmatrix} = |0\rangle|0\rangle + |1\rangle|0\rangle + |2\rangle|0\rangle + |3\rangle|0\rangle + |4\rangle|0\rangle + |5\rangle|0\rangle + |6\rangle|0\rangle + |7\rangle|0\rangle \sum_{x=0}^7 |x\rangle|0\rangle$$

These calculations show that we landed on the following qubit after multiplying QFT<sub>8</sub> by the qubit  $|0\rangle|0\rangle$ .

$$\frac{1}{\sqrt{8}} \sum_{x=0}^7 |x\rangle|0\rangle$$

We now have to apply the following unitary transformation  $U_f$  to the above qubit:

$$\frac{1}{\sqrt{8}} \sum_{x=0}^7 |x\rangle|0\rangle \rightarrow U_f \rightarrow \frac{1}{\sqrt{8}} \sum_{x=0}^7 |x\rangle|f(x)\rangle = \frac{1}{\sqrt{8}} \sum_{x=0}^7 |0\rangle \otimes |f(x)\rangle$$

The above unitary transformed qubit can be rewritten as follows:

$$\begin{aligned} \frac{1}{\sqrt{8}} \sum_{x=0}^7 |x\rangle|f(x)\rangle &= \frac{1}{\sqrt{8}} (|0\rangle|f(0)\rangle + \\ &|1\rangle|f(1)\rangle + |2\rangle|f(2)\rangle + |3\rangle|f(3)\rangle + |4\rangle|f(4)\rangle + |5\rangle|f(5)\rangle + |6\rangle|f(6)\rangle + |7\rangle|f(7)\rangle) \end{aligned}$$

We now have to measure  $|f(x)\rangle$  whose value can either be equal to  $|0\rangle$  or to  $|1\rangle$  because  $f(x)$  is a modulo of 2, having in mind that we are using modulus 2.

Let us assume that  $|f(x)\rangle$  is equal to  $|0\rangle$ . This implies that  $x$  is even, and the measurement of the above unitary transformed qubit is the following where  $p = 2$  (even or odd).

$$\frac{1}{\sqrt{8}} \sum_{x=0}^7 |0\rangle \otimes |f(x)\rangle \rightarrow \text{measure} \rightarrow \sqrt{\frac{p}{N}} \sum_{x=0}^{\frac{N}{p}-1} |x\rangle|f(x)\rangle = \frac{1}{2} (|0\rangle + |2\rangle + |4\rangle + |6\rangle) \otimes |0\rangle$$

The QFT of the above measurement yields

$$\frac{1}{2}(|0\rangle + |0\rangle + |0\rangle + |0\rangle) \rightarrow \text{QTF}_8 \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |4\rangle)$$

This transformation tells us that we must measure  $|4\rangle$ . Moreover, the division of  $N$  by 4 is equal to 2, since  $N = 8$ , and the period of our function is 2.

### Summary

Three advanced algorithmic paradigms were introduced in this unit: parallel, probabilistic, and quantum computing. Parallel algorithms are classified either as data partitioning or function partitioning algorithms. They are based on fundamental mechanisms such as elementwise operations, reductions, and broadcasting. This unit contains practical basic examples on how to program fundamental probability distributions. It also showed how to compute descriptive statistics and test a hypothesis using the Bayes factor. We provided a brief historical perspective on quantum computing before recalling core mathematical concepts on complex numbers, vectors, and matrices. Finally, we reviewed key quantum computing system components, such as states and dynamics, qubit, and Shor's algorithm.

### Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

**Congratulations!**

You have now completed the course. After you have completed the knowledge tests on the learning platform, please carry out the evaluation for this course. You will then be eligible to complete your final assessment. Good luck!





# Appendix 1

## List of References



# List of References

- Abadi, M., & Cardelli, L. (1996). *A theory of objects*. Springer.
- Adams, P. (2018). *Undecidability and the structure of the Turing degrees* [REU Paper, University of Chicago].
- Adel'son-Vels'ky, G. M., & Landis, E. M. (1962). An algorithm for organization of information. *Doklady Akademii Nauk SSSR*, 146(2), 263–266.
- Aho, A. V. (1977). Algorithms and computational complexity. *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography*, 33(1), 5–12.
- Ahujia, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network flows: Theory, algorithms and applications*. Prentice Hall.
- Alami, A., Cohn, M. L., & Wąsowski, A. (2019, May 25–31). Why does code review work for open source software communities? *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)* (pp. 1073–1083). IEEE.
- Attard Cassar, E. (2018). In search of the fastest sorting algorithm. *Symposia Melitensia*, 14, 63–77. <https://www.um.edu.mt/library/oar/handle/123456789/30001>
- Ausiello, G. (2013). Algorithms, an historical perspective. In A. Giorgio & R. Petreschi (Eds.), *The Power of Algorithms* (pp. 3–26). Springer.
- Blank, B. E. (2002). *The millennium problems: The seven greatest unsolved mathematical puzzles of our time*. Basic Books.
- Catanzaro, B., Sundaram, N., & Keutzer, K. (2008, April). A map reduce framework for programming graphics processors. *Workshop on Software Tools for MultiCore Systems*.
- Crawford, K. (2013, April 1). The hidden biases in big data. *Harvard Business Review*. <https://hbr.org/2013/04/the-hidden-biases-in-big-data>
- Frege, G. (1990). Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. In J. van Heijenoort (Ed.), *Frege to Gödel: A source book in mathematical logic* (pp. 1–82). Harvard University Press. (Original work published 1879)
- Gasarch, W. I. (2002). The P=?NP poll. *ACM SIGACT News*, 33(2), 34–47.
- Gasarch, W. I. (2012). Guest column: The second P=?NP poll. *ACM SIGACT News*, 43(2), 53–77.
- Girard, J. Y. (1987). Linear logic. *Theoretical computer science*, 50(1), 1–101.
- Hidary, J. D. (2019). *Quantum computing: An applied approach*. Springer.

## List of References

Hill, K. (2020, June 24). Wrongfully accused by an algorithm. *The New York Times*. <https://www.nytimes.com/2020/06/24/technology/facial-recognition-arrest.html>

Hoare, C. A. R. (1961). Algorithm 64: quicksort. *Communications of the ACM*, 4(7), 321.

Kessler, G. C. (2020, June 1). *An overview of cryptography*. Gary Kessler. <https://www.gary-kessler.net/library/crypto.html>

Kao, Y. F. (n. d.). *Computable foundations of bounded rationality* [Lecture notes]. <https://pdfs.semanticscholar.org/bc65/587fe1409cc4a152bdaefedec0c6e2020100.pdf>

Kochhar, P. S., Xia, X., & Lo, D. (2019, May). Practitioners' views on good software testing practices. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 61–70). IEEE.

Lafont, Y. (1989, December). Interaction nets. *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on principles of programming languages* (pp. 95–108). Association for Computing Machinery.

Lagarias, J. C. (1985). The  $3x + 1$  problem and its generalizations. *The American Mathematical Monthly*, 92(1), 3–23.

Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2), 129–137.

Lomonaco, S. J. (2000). A lecture on Shor's quantum factoring algorithm [Lecture notes]. arXiv. <https://arxiv.org/pdf/quant-ph/0010034.pdf>

Lyman, J. (2016). Blossom: A language built to grow. *Mathematics, Statistics, and Computer Science Honors Projects*, 45.

MacCormick, J. (2013). *Nine algorithms that changed the future: The ingenious ideas that drive today's computers*. Princeton University Press.

Mazur, J. (2014). *Enlightening symbols: A short history of mathematical notation and its hidden powers*. Princeton University Press.

Olhede, S. C., & Wolfe, P. J. (2018). The growing ubiquity of algorithms in society: Implications, impacts and innovations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 376(2128), 3–26.

Poonen, B. (2014). Undecidable problems: A sampler. In J. Kennedy (Ed.), *Interpreting Gödel: Critical essays* (pp. 211–241). Cambridge University Press.

Reiser, M. (2017). *Parallelize JavaScript computations with ease* [Projektarbeit, Hochschule für Technik Rapperswil]. Hochschule für Technik Rapperswil.

- Richardson, K. (2020, May). *Number theory meets computability theory*. Kyle Richardson. <https://www.nlp-kyle.com/files/h10.pdf>
- Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 120–126.
- Rodó, C. (2010). *Efficiency in quantum key distribution protocols using entangled Gaussian states* [Master thesis, Universitat Autònoma de Barcelona]. arXiv:1005.2291.
- Saltz, J. S., & Shamshurin, I. (2017, July). Does pair programming work in a data science context? An initial case study. *2017 IEEE International Conference on Big Data (Big Data)* (pp. 2348–2354). IEEE.
- Simon, H. A. (2002). Near decomposability and the speed of evolution. *Industrial and corporate change*, 11(3), 587–599.
- Strubell, E. (2011). *CSE 301: An introduction to quantum algorithms* [Lesson content].
- Subero, A. (2020). *Codeless data structures and algorithms: Learn DSA without writing a single line of code*. Apress.
- Turner, R. (2018). Towards a philosophy of computer science. *Computational Artifacts* (pp. 13–19). Springer.
- van de Meent, J. W., Paige, B., Yang, H., & Wood, F. (2018). An introduction to probabilistic programming. arXiv:1809.10756.
- Varadharajan, S. (2020). *Hard mathematical problems in cryptography and coding theory* [Doctoral dissertation, The University of Bergen]. The University of Bergen.
- Vega, F. (2016). NL versus P. hal-01354989. <https://hal.archives-ouvertes.fr/hal-01354989/document>
- Vryonis, P. (2013, August 27). Public-key cryptography for non-geeks. *Vrypan*. <https://blog.vrypan.net/2013/08/28/public-key-cryptography-for-non-geeks/>
- Wang, H. (1990). The concept of computability. In *Computation, Logic, Philosophy* (pp. 13–29). Springer. <https://doi.org/10.1007/978-94-009-2356-0>
- Ye, Y. (2013). *Generalizing contexts amenable to greedy and greedy-like algorithms* [Doctoral dissertation, University of Toronto]. TSpace.
- Zapata-Rivera, L. F., & Aranzazu-Suescun, C. (2020). Enhanced virtual laboratory experience for wireless networks planning learning. *IEEE Revista Iberoamericana de Tecnologías del Aprendizaje*, 15(2), 105–112.

# Appendix 2



## List of Tables and Figures

# List of Tables and Figures

**“Hello There” Example**

Source: Author.

---

**GCD Naive Algorithm**

Source: Author.

---

**Selection Sort Algorithm**

Source: Author.

---

**Cryptography Systems**

Source: Kessler, 2016.

---

**Third Person Present Tense Pattern**

Source: Author.

---

**Array’s Input and Output Example**

Source: Author.

---

**Stack Implementation with Linked Nodes (Start)**

Source: Author.

---

**Stack Implementation with Linked Nodes (End)**

Source: Author.

---

**Linked List Representation of the YES String**

Source: Author.

---

**Removal of an Element from a List**

Source: Author.

---

**List Implementation with Linked Nodes (Start)**

Source: Author.

---

**List Implementation with Linked Nodes (Cont’d)**

Source: Author.

---

## List of Tables and Figures

**List Implementation with Linked Nodes (End)**Source: Author.  
.....**Non-Priority Queues Implementation with Linked Nodes (Start)**Source: Author.  
.....**Non-Priority Queues Implementation with Linked Nodes (End)**Source: Author.  
.....**Priority Queues Implementation with Linked Nodes (Start)**Source: Author.  
.....**Priority Queues Implementation with Linked Nodes (Cont'd)**Source: Author.  
.....**Priority Queues Implementation with Linked Nodes (End)**Source: Author.  
.....**Completely Full Binary Tree Example**Source: Author.  
.....**Numbering of the Nodes of a Completely Full Binary Tree**Source: Author.  
.....**Binary Trees Implementation with Arrays (Start)**Source: Author.  
.....**Binary Trees Implementation with Arrays (End)**Source: Author.  
.....**Example of a Labeled Graph**Source: Author.  
.....**Example of Nodes Identification in a Graph**Source: Author.  
.....

**Representation of the Nodes of a Graph in an Array**Source: Author.  
.....**Graph's Representation as a Two-Dimensional Array and an Array of Linked Nodes**Source: Author.  
.....**Graph Implementation with Two Arrays (Start)**Source: Author.  
.....**Graph Implementation with Two Arrays (End)**Source: Author.  
.....**Iterative and Recursive Versions of Factorial**Source: Author.  
.....**Iterative Illustration of Factorial**Source: Author.  
.....**Naive Iterative and Recursive Primality Test Algorithms**Source: Author.  
.....**Insertion of the Sequence 75, 29, 52, 89, 92, 90, 24, 8, 17, 27 in an AVL (Start)**Source: Author.  
.....**Insertion of the Sequence 75, 29, 52, 89, 92, 90, 24, 8, 17, 27 in an AVL (Cont'd)**Source: Author.  
.....**Insertion of the Sequence 75, 29, 52, 89, 92, 90, 24, 8, 17, 27 in an AVL (End)**Source: Author.  
.....**Greedy Algorithm Example**Source: Author.  
.....**Dynamic Programming of the Currency Exchange Problem**Source: Author.  
.....



## List of Tables and Figures

**Sorting Algorithms**Source: Author.  
.....**Radix Sort Algorithm Example**Source: Author.  
.....**Bucket Sort Algorithm**Source: Author.  
.....**Insertion Sort Algorithm Example**Source: Author.  
.....**Bubble Sort Algorithm Example (Start)**Source: Author.  
.....**Bubble Sort Algorithm Example (End)**Source: Author.  
.....**Merging Two Sorted Sequences and Merge Sort Algorithm Examples**Source: Author.  
.....**Quicksort Algorithm Example**Source: Author.  
.....**jq Filtering Patterns (Selection)**Source: Author.  
.....**Json Object First Example**Source: Author.  
.....**Outputs**Source: Author.  
.....**Json Object Second Example**Source: Author.  
.....

**Outputs for Eight tests and capture Commands**

Source: Author.

---

**K-Means Algorithm Example**

Source: Author.

---

**Graphic Representation of the Above Dataset**

Source: Author.

---

**K-Means Algorithm Example First Step**

Source: Author.

---

**K-Means Algorithm Example Second Step**

Source: Author.

---

**K-Means Algorithm Example Third Step**

Source: Author.

---

**K-Means Algorithm Example Four Step**

Source: Author.

---

**K-Means Algorithm Example Diagram**

Source: Author.

---

**Example of the Sum of Numbers**

Source: Author.

---

**Example of the Recursive Version of Factorial**

Source: Author.

---

**Summary**

Source: Author.

---

**Collatz Sequence Code**

Source: Author.

---

## List of Tables and Figures

**Twin Primes Code**Source: Author.  
.....**Perfect Numbers Caller Example**Source: Author.  
.....**Primality Test Example with Exceptions (First Version)**Source: Author.  
.....**Primality Test Example with Exceptions (Second Version)**Source: Author.  
.....**Primality Test Example with Assertions**Source: Author.  
.....**Primality Test Example for Jshint Demo**Source: Author.  
.....**Jest Object for package.json File**Source: Author.  
.....**First Primality Test Example for Jest Demo**Source: Author.  
.....**First Set of Jest Test Cases for the Primality Test Program**Source: Author.  
.....**Second Primality Test Example for Jest Demo**Source: Author.  
.....**Second Set of Jest Test Cases for the Primality Test Program**Source: Author.  
.....**Code for the Creation of a MySQL Database**Source: Author.  
.....

**Code for the Creation of a MySQL Table**

Source: Author.

---

**Code for the Storage of a MySQL Record**

Source: Author.

---

**Code for the Querying of a MySQL Record**

Source: Author.

---

**Mid Rank Intervals for Epsilon**

Source: Author.

---

**Mid Rank Problem Algorithm Example**

Source: Author.

---

**YES or NO Automaton**

Source: Author.

---

**Automaton for the Positive Multiples of 10**

Source: Author.

---

**Alphabet's Elements  $c$** 

Source: Author.

---

**Pushdown Automaton for Well-Bracketed Expressions**

Source: Author.

---

**Turing Machine for the Words  $a^n b^n c^n$** 

Source: Author.

---

**Turing Machine (aabbcc Input)**

Source: Author.

---

**Turing Machine Concept for the Null (Empty) String Problem**

Source: Author.

---

## List of Tables and Figures

**Recursively Enumerable Set Membership JavaScript Pseudocode**Source: Author.  
.....**Biggest Subsequence Problem JavaScript Code (Start)**Source: Author.  
.....**Biggest Subsequence JavaScript Code (End)**Source: Author.  
.....**Traced Algorithm**Source: Author.  
.....**Tracing Kadane's Algorithm**Source: Author.  
.....**Graphical Comparison of  $O(n^2)$  and  $O(n)$** Source: Author.  
.....**Fibonacci Algorithms (Start)**Source: Author.  
.....**Fibonacci Algorithms (Cont'd)**Source: Author.  
.....**Fibonacci Algorithms (End)**Source: Author.  
.....**Tracing the Fast Fibonacci Algorithms**Source: Author.  
.....**Search Algorithm Example**Source: Author.  
.....**Most Common Complexity Classes**Source: Author.  
.....

**Computational Problems (Examples)**

Source: Author.

---

**Traveling Salesperson Problem Example Graph**

Source: Author.

---

**Decision Hamilton Cycle Problem Examples Graphs**

Source: Author.

---

**Reducing Hamilton Cycle Problem Examples Graphs**

Source: Author.

---

**ParallelJs Map Operation First Example with Output Screenshot**

Source: Author.

---

**ParallelJs Map Operation Second Example**

Source: Author.

---

**Output Screenshot of the Second ParallelJs Map Operation Example**

Source: Author.

---

**Output Screenshot of the ParallelJs Map Reduce Example**

Source: Author.

---

**ParallelJs Broadcasting Operation Example (Start)**

Source: Author.

---

**ParallelJs Broadcasting Operation Example (End)**

Source: Author.

---

**ParallelJs Broadcasting Operation Example**

Source: Author.

---

**WebPPL Hello Mam or Dad Example**

Source: Author.

---

## List of Tables and Figures

**Output Screenshot of the WebPPL Hello Mam or Dad Example**Source: Author.  
.....**WebPPL Bernoulli Distribution First Example**Source: Author.  
.....**WebPPL Bernoulli Distribution Second Example**Source: Author.  
.....**WebPPL Binomial Distribution Example**Source: Author.  
.....**WebPPL Geometric Distribution Example**Source: Author.  
.....**Output Screenshot of WebPPL Geometric Distribution Example**Source: Author.  
.....**WebPPL Uniform Distribution Example**Source: Author.  
.....**Output Screenshot of WebPPL Uniform Distribution Example**Source: Author.  
.....**WebPPL Descriptive Statistics Example (Start)**Source: Author.  
.....**WebPPL Descriptive Statistics Example (End)**Source: Author.  
.....**Output Screenshot of WebPPL Descriptive Statistics Example**Source: Author.  
.....**WebPPL Inferential Statistics Example**Source: Author.  
.....

**Output Screenshot of WebPPL Inferential Statistics Example**

Source: Author.

---

**WebPPL Text Version Search Example with Uniformly Distributed Input (Start)**

Source: Author.

---

**WebPPL Text Version Search Example with Uniformly Distributed Input (End)**

Source: Author.

---

**Detailed Array Output of WebPPL Uniform Input Distribution Text Version Example**

Source: Author.

---

**Real Mean Output of WebPPL Uniform Input Distribution Text Version Example**

Source: Author.

---

**Natural Mean Output of WebPPL Uniform Input Distribution Text Version Example**

Source: Author.

---

**WebPPL Graphic Version Search Example with Uniformly Distributed Input (Start)**

Source: Author.

---

**WebPPL Text Version Search Example with Uniformly Distributed Input (End)**

Source: Author.

---

**Plotly Uniform Input Distribution Example**

Source: Author.

---

**Output of Plotly Uniform Input Distribution Example (Screenshot)**

Source: Author.











**IU Internationale Hochschule GmbH**  
**IU International University of Applied Sciences**

Juri-Gagarin-Ring 152  
D-99084 Erfurt



Mailing address:  
Albert-Proeller-Straße 15-19  
D-86675 Buchdorf