Chapter 1

Algorithms and Pseudocode

Area of Study 2: Algorithm design Outcome 2

Learning Intentions

- Key knowledge
  - basic structure of algorithms
  - pseudocode concepts, including variables and assignment, sequence, iteration, conditionals and functions
  - programming language constructs that directly correspond to pseudocode concepts
  - conditional expressions using the logical operations of AND, OR, NOT
- Key skills
  - interpret pseudocode and execute it manually on given input
  - write pseudocode

Introduction

Problem-solving often involves simplifying a real-world scenario by creating a model that captures essential information and relationships. In this model, the information about the problem is stored as **data** which is then manipulated to find a solution.

Two key concepts:

- **Data**: different data types are designed to store different kinds of information. Selecting the right data types helps optimize your solution by making data storage and retrieval more efficient
- **Algorithm**: Data is manipulated by algorithms to achieve the desired result.

Example: Baking a Cake

Data Types: Here, the types and quantities of ingredients represent the core data needed to bake the cake. Each ingredient's type (e.g., flour, sugar, eggs) and its specific measurement (e.g., 2 cups of flour, 1 cup of sugar) are different data elements that must be stored and managed accurately to ensure the cake turns out as expected.

Algorithm: The recipe acts as the algorithm, guiding you through the steps to transform the raw ingredients into a finished cake. This sequence includes mixing, baking, and cooling, with each step building upon the last.

In this subject:

- Area of Study 1 Data modelling with abstract data types
- Area of Study 2 Algorithm design

Algorithm

An algorithm is a step-by-step procedure for solving a problem or completing a task. The basic structure of an algorithm typically includes:

1. **Input**: Any data the algorithm requires to function.
2. **Process**: The core operations that transform the input into the output, often using flow control statements.
3. **Output**: The result of the algorithm's processing.

Flow control statements generally have the form:

```
keyword Condition
    Clause
end keyword
```

- **Keyword**: Specifies the type of flow control, such as `if`, `while`, or `for`.
- **Condition**: A Boolean expression that evaluates to `True` or `False`.
- **Clause**: Code to execute based on the condition.

Pseudocode

Pseudocode is a way to describe an algorithm using plain language mixed with programming like syntax. It's intended to be easily understood without focusing on syntax rules of a specific programming language. For written assessments such as SACs and exams, we will use pseudocode. For tasks involving implementing algorithms on a computer, we will use Python.

Programming Language Constructs that Directly Correspond to Pseudocode Concepts

There is a high level of correspondence between pseudocode and Python constructs. This makes it straightforward to translate algorithms written in pseudocode into Python code and vice versa.

Table 1.1: Differences between pseudocode and python

Pseudocode Concept	Pseudocode Syntax	Python Syntax
Variable Assignment	`x ← 5`	`x = 5`
Equality Check	`x = 5`	`x == 5`
Not Equal	`x` $\neq$ `5`	`x != 5`
Greater Than or Equal To	`x` $\geq$ `5`	`x >= 5`
Less Than or Equal To	`x` $\leq$ `5`	`x <= 5`
Conditional (If)	`if condition then`	`if condition:`
Colon :	Not required	required at the end of a flow control statement
Indentation	For readability	required to define the structure of code blocks
Else If / Elif	`else if` or `elif`	`elif condition:`
Else	`else`	`else:`
While Loop	`while condition`	`while condition:`
For Loop	`for i from 1 to 5`	`for i in range(1, 6):`
Function Definition	`Algorithm name(params)`	`def name(params):`

Boolean Expressions

A Boolean expression has a data type that can only hold two values: `True` and `False`. The concept is named after mathematician George Boole.

(a) Comparison Operators

Operator	Meaning
`=`	Equal to
$\neq$	Not equal to
`<`	Less than
`>`	Greater than
$\leq$	Less than or equal to
$\geq$	Greater than or equal to

(b) Logical NOT Expressions

Expression	Evaluates to
`not True`	`False`
`not False`	`True`

(a) Logical OR Expressions

Expression	Evaluates to
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

(b) Logical AND Expressions

Expression	Evaluates to
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

Example: Evaluate the expression `NOT`$(8 < 5)$ `AND` $2 \leq 3$

Example: Simplify the expression $(x$ `OR` $y)$ `AND` $(x$ `OR NOT`$(y))$

1.1 Exercise

1. Determine if each of the following Boolean expressions is `True` or `False`:
   (a) $5 > 3$ AND $7 < 10$
   (b) $4 = 4$ OR $6 \neq 6$
   (c) `NOT` $(8 < 5)$
   (d) $3 \neq 3$ OR $2 \geq 1$
   (e) $(6 < 9)$ AND $(7 > 8)$

2. Determine if the following expressions are `True` or `False` given $x = 4$ and $y = 10$:
   (a) $x < y$ AND $x \neq y$
   (b) $x \leq 5$ OR $y \neq 10$
   (c) `NOT` $(y < 8)$
   (d) $x = 4$ AND $y \leq 10$

3. Write Boolean expressions that represent the following conditions:
   (a) A number n is both greater than 10 and less than 20.
   (b) Two variables, a and b, are not equal to each other.
   (c) A variable age is greater than or equal to 18 and less than 65.
   (d) A value temp is less than 0 or greater than 100.
   (e) A student has passed if score ≥ 50.

4. Write Boolean expressions for the following scenarios:

   (a) A user receives a discount if they are either a member or have a coupon.

   (b) A room's air conditioning should be on if the temperature is above 25 degrees or if is above 70%.

   (c) The system displays a warning if either `battery_level` is less than 20% or `connection` is not available.

   ---

5. Simplify each of the following Boolean expressions:

   (a) `NOT (NOT x)`

   (b) `x AND (NOT x OR y)`

   (c) `(a OR b) AND a`

   (d) `(p OR q) AND (NOT p OR r)`

Flow Control in Pseudocode

Flow control in pseudocode usually takes the form of `if`, `while`, or `for` statements. Common types of flow control include:

- **Iteration**: Repeating a set of instructions until a condition is met. Examples include `while` and `for` loops.
- **Conditionals**: Decision-making structures that perform different actions based on specific conditions. This is typically done using `if`, `else`, and `else if` statements.

if Statement:

```
if temperature > 30:
    display "It's hot"
else:
    display "It's not hot"
end if
```

while Loop:

```
while temperature < 30:
    display "It's too cold"
    increase temperature
end while
```

for Loop:

```
for i from 1 to 10:
    display i
end for
```

Algorithm Structure in Pseudocode

Algorithms written in pseudocode generally have the form:

```
Algorithm name(argument1, argument2)
    perform operations on argument1 and argument2
    return result
```

For example:

```
Algorithm add(x, y)
    result = x + y
    return result
```

Example: Write a program that takes a single user input and prints the FizzBuzz pattern up to that number. Use the helper functions provided below:

```
Algorithm IsDivisibleByThree(number)
    if number % 3 = 0:
        return True
    else:
        return False
    end if
end Algorithm
```

```
Algorithm IsDivisibleByFive(number)
    if number % 5 = 0:
        return True
    else:
        return False
    end if
end Algorithm
```

Variables

A **variable** is a symbolic name or container that holds data or a value, which can be changed throughout the execution of a program. Variables make it possible to work with data dynamically, allowing algorithms to handle various inputs and produce different results.

Variable names should be descriptive and typically follow specific naming conventions. For example:

- Use meaningful names that reflect the purpose, like `age`, `total`, or `average_score`.
- Avoid spaces; if the name contains multiple words, use `camelCase` (e.g., `totalPrice`) or `snake_case` (e.g., `total_price`).

Data Types

Variables often have different **data types** depending on the kind of data they store. Common types include:

- **Integer**: Holds whole numbers (e.g., `5`, `-3`).
- **Float**: Holds decimal numbers (e.g., `3.14`, `-0.001`).
- **String**: Holds sequences of characters (e.g., `"hello"`, `"Algorithm"`).
- **Boolean**: Holds logical values, `True` or `False`.

Assignment

Assignment is the process of giving a variable a value. In pseudocode, we use the **arrow operator** (←) to assign a value to a variable. The value on the right side of the arrow is stored in the variable on the left side.

- **Syntax**: `variable_name ← value`
- **Example**:

```
x ← 5              // Assigns the integer 5 to variable x
name ← "Alice"     // Assigns the string "Alice" to variable name
temperature ← 25.5 // Assigns the decimal 25.5 to variable temperature
```

A variable can be updated by reassigning it a new value or by using its current value in calculations. For instance:

- **Incrementing**: `count ← count + 1` (adds 1 to the current value of `count`).
- **Using in Calculations**:

```
price ← 20
discount ← 5
final_price ← price - discount
```

Example Algorithm Using Variables and Assignment

Consider a simple algorithm that calculates the area of a rectangle:

```
Algorithm RectangleArea
    length ← 10               // Assigns 10 to the variable length
    width ← 5                 // Assigns 5 to the variable width
    area ← length * width     // Calculates area and assigns the result to area
    display area
end Algorithm
```

In this example:

- We use `length` and `width` as variables to hold the rectangle's dimensions.

- The **assignment** `area ← length * width` calculates the area and stores the result in the `area` variable.
- Finally, we display the `area` value.

1.2 Exercise

1. What value will be printed by the pseudocode?

```
a ← 5
while a < 50
    a ← a + 3
end while
print(a)
```

   A) 47

   B) 49

   C) 50

   D) 51

2. What value is printed by this pseudocode?

```
sum ← 0
for i from 1 to 4
    for j from 1 to 3
        sum ← sum + i + j
    end for
end for
print(sum)
```

   A) 30

   B) 36

   C) 54

   D) 60

3. This algorithm attempts to find an approximate root of the function $f(x) = x^2 - 9$. What will it output?

```
define f(x)
    return (x^2 - 9)

a ← 0
b ← 5
mid ← (a + b) / 2

while b - a > 0.1
    if f(a) * f(mid) < 0 then
        b ← mid
    else
        a ← mid
    end if
    mid ← (a + b) / 2
end while
print(mid)
```

4. The pseudocode below calculates a sum based on an iterative approach:

```
define g(x)
    return x + 1

a ← 1
b ← 3
n ← 4
h ← (b - a) / n
sum ← g(a) + g(b)
x ← a + h

for i from 1 to n - 1
    sum ← sum + 2 * g(x)
    x ← x + h
end for
print(sum)
```

What is the final value of `sum`?

A) 12

B) 14

C) 16

D) 18

5. Consider the following:

```
define f(x)
    return x^2
end define

define arc_length_segment(left, right, h)
    return sqrt((f(right) - f(left))^2 + h^2)
end define

define arc_length(a, b, h)
    sum ← 0
    left ← a
    right ← a + h

    while right <= b do
        arc ← arc_length_segment(left, right, h)
        sum ← sum + arc
        left ← left + h
        right ← right + h
    end while

    return sum
end define
```

What does `print(arc_length(1, 2, 0.5))` print?

Chapter 2

Data Types

Area of Study 1: Data modelling with abstract data types Outcome 1

Learning Intentions

- Key knowledge
  - the motivation for using ADTs
  - signature specifications of ADTs using operator names, argument types and result types
  - specification and uses of the following ADTs:
  - set, list, array, dictionary (associative array)
  - stack, queue, priority queue
- Key skills
  - explain the role of ADTs for data modelling
  - read and write ADT signature specifications

Primitive Data Types

We have already seen the primitive data types. These are the basic, low-level types directly supported by the programming language or underlying hardware. Primitives are usually predefined in programming languages and are the building blocks for more complex data types and structures.

- **Integer**: Holds whole numbers (e.g., `5`, `-3`).
- **Float**: Holds decimal numbers (e.g., `3.14`, `-0.001`).
- **Char** Represents a single character (e.g., `a`, `f`, `!`).
- **String**: Holds sequences of characters (e.g., `"hello"`, `"Algorithm"`).
- **Boolean**: Holds logical values, `True` or `False`.

Abstract Data Types

Primitive data types can be used to build Abstract Data Types such as

- set, list, array, dictionary
- stack, queue, priority queue
- graphs

Collection Data Types

Sets, lists, arrays, and dictionaries are all data types that store collections of primitives.

Sets

A **set** is a fundamental concept in mathematics, representing a collection of distinct objects or elements. Sets store a collection of **unordered** elements and support mathematical operations like union, intersection, and difference.

Sets are stored in memory in a unique way, allowing for faster operations but with limited functionality compared to lists; elements in a set cannot be indexed.

Sets do not contain duplicates.

Sets can be used for modelling Collections of unique items such as the days of the week, types of fruit, colours of paint.

Examples:

- $\{1, 2, 3\}$ – A set of numbers.
- $\{a, e, i, o, u\}$ – A set of vowels.

Curly braces $\{\}$ are used to denote a set. The symbol $\in$ is used to indicate that an element belongs to a set. For example:

$$2 \in \{1, 2, 3\}$$

This means that 2 is an element of the set $\{1, 2, 3\}$. Similarly, $-5 \in \mathbb{Z}$ indicates that -5 is an element of the set of integers.

Standard Sets

These are sets with a well-known and universally accepted definition, often represented by special symbols in mathematics. Examples include:

- $\mathbb{N}$ – The set of natural numbers.
- $\mathbb{Z}$ – The set of integers.
- $\mathbb{Q}$ – The set of rational numbers.
- $\mathbb{R}$ – The set of real numbers.
- $\mathbb{C}$ – The set of complex numbers.
- $\emptyset$ – The empty set, which contains no elements.

Excluding Elements from a Set

Sometimes, we want to define a set while specifically excluding certain elements. We use the backslash symbol $\setminus$ to indicate this. For example:

- $\mathbb{R} \setminus \{0\}$ – The set of all real numbers except 0.
- $\mathbb{Z} \setminus \{1, 2\}$ – The set of all integers except 1 and 2.

Sets in python

Sets can be implemented in Python.

```
# Creating a set
my_set = {1, 2, 3, 4}
print(my_set)  # Output: {1, 2, 3, 4}

# Adding elements
my_set.add(5)
print(my_set)  # Output: {1, 2, 3, 4, 5}

# Removing elements
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4, 5}

# Checking membership
print(2 in my_set)  # Output: True

# Union, intersection, and difference
another_set = {4, 5, 6, 7}
print(my_set | another_set)  # Union: {1, 2, 4, 5, 6, 7}
print(my_set & another_set)  # Intersection: {4, 5}
print(my_set - another_set)  # Difference: {1, 2}

#Iterating
for i in my_set:
    print(i)
```

Array

An array stores a collection of elements in a fixed-size structure, meaning the size of the array is defined when it is created and cannot be changed. Arrays can contain duplicate values, and each element is stored in a contiguous block of memory, allowing efficient access via indexing. Arrays are typically used when the number of elements is known in advance and does not change.

Arrays can be used for modelling Fixed-size data collections such as coordinates on a map, monthly temperatures or a matrix of pixel values in an image.

Arrays in Python

Arrays are not a built-in data type in Python. To use arrays, you need to import the array module. array() function requires two arguments, the first to specify the type of elements and the second the list of elements. In Python, the size of the array can be changed, but this can be slow.

```
import array
my_array = array.array('i', [1, 2, 3])  # 'i' specifies integer type
my_array.append(4)                      # Adds an integer
my_array[1]                             # Output: 2
```

List

A list is similar to an array but does not have a fixed size, meaning elements can be added or removed dynamically. Lists are more flexible than arrays and are typically used when the number of elements is not known in advance or may change over time.

Lists can be used for modelling Dynamic or growing data collections, such as a waiting list for or a shopping cart.

List in Python

```
#Creating a List:

countries = ['United States', 'India', 'China', 'Brazil']
my_list = []
#Indexing:
countries[0]        #Output: 'United States'
countries[3]        #Output: 'Brazil'

# Negative indexing
countries[-1]       #Output: 'Brazil'

#Slicing:
countries[0:3]      #Output: ['United States', 'India', 'China']
countries[1:]       #Output: ['India', 'China', 'Brazil']

#Adding Elements to a List:
countries.append('Canada')  # Adds 'Canada' to the end of the list
countries.insert(0, 'Canada')  # Inserts 'Canada' at the beginning

#Nested List:
nested_list = [countries, countries_2]

#Removing Elements from a List:
countries.remove('United States')  # Removes 'United States' by value
countries.pop(0)  # Removes and returns the element at index 0
```

Dictionaries

A **dictionary** (also known as an associative array or hash map) stores data in **key-value pairs**. Each key in a dictionary is unique, and each key maps to a specific value. Dictionaries are highly efficient for data retrieval when the key is known, making them useful for tasks like looking up information by a specific identifier. The keys in a dictionary are a set, so

- **Unique keys**: Each key in a dictionary must be unique. If a key is added more than once, the most recent value will overwrite the previous one.
- **Unordered**: Dictionaries do not maintain a particular order of elements, although in recent versions of Python (3.7+), dictionaries maintain insertion order.
- **Efficient Lookup**: Dictionaries provide efficient retrieval of values based on keys.

Dictionaries are useful for modelling data where each item is associated with a unique identifier such as a Phone book or student identification number and name.

Dictionaries are efficient for tasks that require quick lookup, insertion, and deletion of items based on unique keys, making them a powerful tool in data modeling.

Dictionaries in Python

A dictionary is created using curly braces {} with key-value pairs separated by colons.

```
# Creating a dictionary
student_grades = {
    "Alice": 85,
    "Bob": 90,
    "Charlie": 78 }

# Accessing a value
print(student_grades["Alice"])  # Output: 85
If a key does not exist you get an error, so use the get method
print(student_grades.get("David", "Not found"))  # Output: Not found

# Adding a new entry
student_grades["David"] = 92

# Modifying an existing entry
student_grades["Alice"] = 88

# Using pop() to remove an entry
student_grades.pop("Charlie")  # Removes "Charlie" and returns 78

# Using del to remove an entry
del student_grades["Bob"]

#Checking if a key exists
if "Alice" in student_grades:
    print("Alice is in the dictionary")

# Iterating over keys
for key in student_grades:
    print(key)

# Iterating over values
for value in student_grades.values():
    print(value)

# Iterating over key-value pairs
for key, value in student_grades.items():
    print(key, value)

#Getting all keys or values
keys = student_grades.keys()
values = student_grades.values()
```

2.1 Exercise

1. Complete the table to show the difference between sets, arrays, lists and dictionaries

Feature	Set	Array	List	Dictionary
Duplicates				
Order				
Indexed Access				
Data Type Consistency				
Common Use Cases				
Syntax Example				

2. For each of the following types of information, choose the most suitable data structure.

   (a) Temperature readings taken every hour for a week.

   (b) A student's grade point average (GPA), which is a single decimal number.

   (c) The days of the week on which a store is open.

   (d) A collection of unique tags assigned to a social media post.

   (e) The age of a person.

   (f) A set of answers (True/False) to a multiple-choice quiz.

   (g) A list of products in a shopping cart, where items may appear multiple times if selected more than once.

   (h) Mapping customer names to their phone numbers in a phone directory.

   (i) A large collection of measurements (e.g., sensor data) where each measurement is an integer and the total number of measurements is fixed.

   (j) A single letter grade assigned to a student, like 'A', 'B', 'C', etc.

   (k) A list of students' names, where order matters (e.g., sorted alphabetically).

   (l) A person's first name.

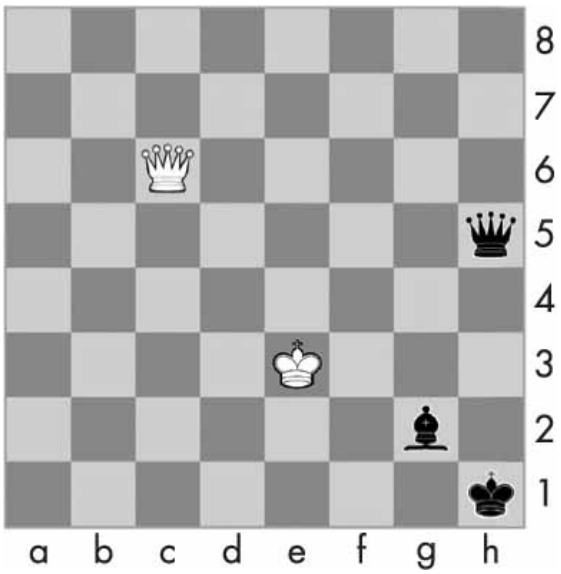
   (m) The presence or absence of an item in inventory, expressed as `True` or `False`.

   (n) Prices of items in a store, each represented by a decimal number.

(o) The top five highest scores in a game, with no duplicates allowed.

3. Chess Dictionary Validator (from Automate the boring stuff)

   A chess board modeled by the dictionary {'1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'}



   Write a function named isValidChessBoard() that takes a dictionary argument and returns True or False depending on if the board is valid.

A valid board will have:

- one black king and exactly one white king
- Each player can only have at most 16 pieces, at most 8 pawns
- all pieces must be on a valid space from '1a' to '8h'; that is, a
- piece can't be on space '9z'
- The piece names begin with either a 'w' or 'b' to represent white or black followed by 'pawn', 'knight', 'bishop', 'rook', 'queen', or 'king'.
- This function should detect when a bug has resulted in an improper chess board.

Abstract Data Types (ADTs)

"Abstract: existing in thought or as an idea but not having a physical or concrete existence." (Oxford English Dictionary, Oxford University Press, 2023).

Abstract Data Types (ADTs) specify how data structures are used and the behaviours they provide. ADTs do not define how the data structure is implemented or stored in memory; instead, they outline a minimal expected interface and a set of behaviours. ADTs are language-agnostic, meaning they can be implemented in any programming language without changing their fundamental properties.

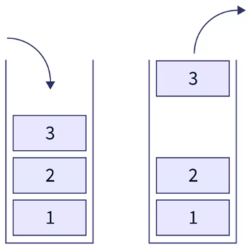
Collection Data Types can be used to implement abstract data types like stacks, queues, and priority queues. For example, a stack can be implemented using an array or a list. Since ADTs are abstract, the specific implementation details are not important at a high level of algorithm design—we only need to know the expected behaviour of the data type, not how it is implemented.

Stack

A **stack** is a list-like data structure in which data is added and removed from a single end, known as the **top** of the stack. It follows the principle of **Last In, First Out (LIFO)**, meaning the last item added is the first one to be removed.

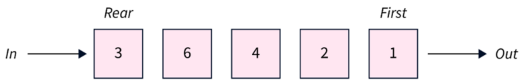
Stack Operations:

- `Push(data)` – Adds an item to the top of the stack.
- `Pop()` – Removes the item from the top of the stack.
- `Peek()` – Retrieves the value from the top of the stack without removing it.



Queue

A **queue** is a list-like data structure in which data is added at one end (the **back**) and removed from the other end (the **front**). It follows the principle of **First In, First Out (FIFO)**, meaning the first item added is the first one to be removed.



Queue Operations:

- `Enqueue(data)` – Adds an item to the back of the queue.
- `Dequeue()` – Removes the item from the front of the queue.
- `Peek()` – Retrieves the value from the front of the queue without removing it.

Signature Specifications

A **signature specification** is a formal way to define the inputs and outputs of an operation without detailing the internal implementation. Signature specifications describe what an operation does in terms of its input and output types, which helps in understanding the functionality of the operation without needing to know how it works internally.

A signature specification generally consists of:

- **Input Types**: The data types or structures required as inputs for the operation.
- **Output Type**: The data type or structure that the operation returns as output.
- **Arrow Notation** ($\rightarrow$): Used to indicate the transformation from input types to an output type.
- **Product Notation** ($\times$): Used to separate the inputs or outputs

For example, the signature specification for an operation might look like this:

$$\text{input type}_1 \times \text{input type}_2 \rightarrow \text{output type}$$

This notation means the operation takes two inputs of types `input type`$_1$ and `input type`$_2$, and returns a result of `output type`.

Example Write the signature specifications for a stack ADT:

- Initialise: init(): $\rightarrow$ stack
- Push: element $\times$ stack $\rightarrow$ stack
- Pop: stack $\rightarrow$ element $\times$ stack
- IsEmpty: Operation stack $\rightarrow$ boolean

2.2 Exercise

1. Write the signature specifications for a Queue:

Priority Queue

A **priority queue** is a data structure in which elements are ordered based on their priority rather than their order of insertion. Each element consists of a priority and an associated value, and elements with higher priority are typically removed before those with lower priority. If two elements have the same priority, they are usually processed in the order they were inserted (following a First In, First Out approach for items with equal priority).

Priority Queue Signature Specifications

- Initialise: init(): $\rightarrow$ priority queue
- Enqueue: element $\times$ priority $\times$ priority queue $\rightarrow$ priority queue
- Dequeue: priority queue $\rightarrow$ element $\times$ priority queue
- Peek: priority queue $\rightarrow$ element
- IsEmpty: priority queue $\rightarrow$ boolean

Example: In a hospital Emergency Department waiting room, patients are triaged based on the severity of their conditions. The priority levels are:

- **Priority 3**: Critical condition
- **Priority 2**: Serious condition
- **Priority 1**: Mild condition

The patients are managed using a priority queue named `ER_Waiting_List`. The current waiting list contains:

`ER_Waiting_List` = [Alice: Priority 1, Bob: Priority 1, Charlie: Priority 2, Daisy: Priority 2]

A new patient, Bill, arrives and is triaged as Priority 3. Write pseudocode to add Bill to the priority queue and provide the updated status of `ER_Waiting_List`.

The doctor is ready to see the next patient. Write pseudocode and provide the updated status of `ER_Waiting_List`.

The doctor is ready to see the next patient. Write pseudocode and provide the updated status of `ER_Waiting_List`.

2.3 Exercise

1. For each of the following scenarios, choose the most appropriate data structure.

   (a) A web browser keeps track of the pages a user has visited, allowing them to navigate back to previous pages in the exact reverse order they visited.

   (b) A task scheduler in an operating system needs to manage tasks based on their arrival time, ensuring that tasks are executed in the order they were received.

   (c) A school library needs to maintain a list of books checked out by each student, with each book associated with the student's name. The library should be able to efficiently look up which books a specific student has checked out.

   (d) An online gaming leaderboard shows the top scores, with no duplicate scores allowed. The leaderboard should always show the top five highest scores in order.

   (e) A train station has multiple platforms, with each platform having a queue of passengers waiting for the next train. Each queue should process passengers in the order they arrived.

   ---

2. Write Python code to implement a basic stack using a list. Implement and demonstrate the push, pop, and peek operations. (No methods allowed; you must use indexing, + and del)
3. Write Python code to implement a priority queue using a list. Implement and demonstrate the enqueue, dequeue, peek and isempty operations. (No methods allowed; you must use indexing, + and del)

4. 2015 ALGORITHMICS EXAM Question 2

   (a) Describe the difference between an array and a dictionary. (2 marks)

   (b) Justify, with two real-world examples, when a priority queue is the more suitable abstract data type than a queue. (2 marks)

5. 2016 ALGORITHMICS EXAM Question 4 (2 marks)
   When data is transferred across computer networks, it is first broken up into packets. Computer network traffic is normally processed in the order that packets arrive at each device along the path between communicating devices. Packets may be pieces of email, web content, voice or video. While some traffic, such as email or web content, can withstand delays in delivery, others, such as voice and video, cannot have delays; these packets cannot wait at each device for other traffic to be processed ahead of them. Describe a standard abstract data type (ADT) that could be used to manage the packets arriving at a computer.

6. 2020 ALGORITHMICS EXAM Question 1

   (a) Explain, with an example, the concept of the dictionary abstract data type (ADT). (2marks)

   (b) Write a complete signature specification for a dictionary ADT. (3 marks)

7. 2021 ALGORITHMICS EXAM Question 2

Alex is developing a computer simulation to model a chemical reaction. In the simulation, the reactor is initially empty and the reaction starts when one or more chemicals are added. More chemicals may be added while the reaction is in progress. A chemical reaction is characterised by what chemicals are added, how much of them are added and when they are added.

(a) To store a description of a chemical reaction for simulation, Alex needs the following information about each addition into the reactor:

- the name of the chemical added
- the time the chemical was added, in seconds, after the reaction begins
- the amount of the chemical added, in milligrams

Describe a combination of abstract data types (ADTs) that Alex could use to store a description of a chemical reaction. Justify your answer. (3 marks)

(b) Alex intends to use the combination of ADTs described in part a. often and decides to define the combination of ADTs as a new ADT called ChemEvents. For the ChemEvents ADT, as described in part a., write the signature specification of the following operations:

- Add a new chemical to the reaction.
- Look up what chemical was added to the reaction at a given time. (2 marks)

Chapter 3

Graphs

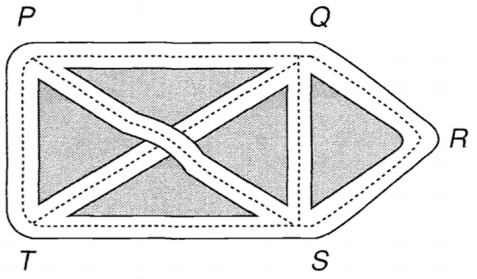
Area of Study 1: Data modelling with abstract data types Outcome 1

Learning Intentions

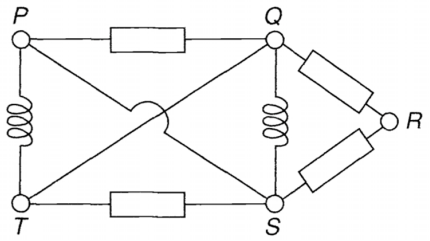
- Key knowledge
  - features of graphs, including paths, weighted path lengths, cycles and subgraphs
  - categories of graphs, including complete graphs, connected graphs, directed acyclic graphs and trees, and their properties
  - modularisation and abstraction of information representation with ADTs
- Key skills
  - model basic network and planning problems with graphs, including the use of decision trees and state graphs

Graphs

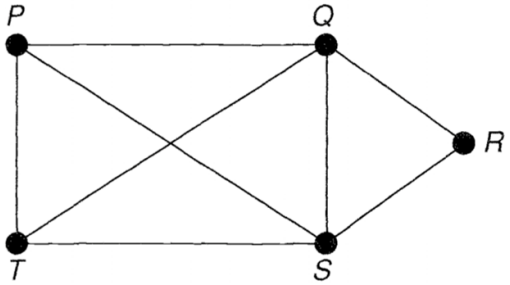
The Graph Abstract Data Type (ADT) is a data structure used to represent and model relationships between pairs of objects. In a graph, objects are represented by nodes (or vertices), and the connections between them are called edges. Graphs are widely used in computer science to model complex structures like networks, relationships, and paths.



(a) Road Network



(b) Electronics Schematic



(c) Abstract Representation

Graphs are written as an ordered pair $G(V, E)$. Where V represents the set of vertices and E represents the set of edges. The graph above can be written as:

Graph G=(V,E), where

- V={P,Q,R,S,T}
- E={PQ, PT, PS, TQ, TS, QR, SR}

Nomenclature

Walk: A sequence of vertices connected by edges. Example:

$$P \to S \to Q \to T \to S \to R$$

Path: A walk in which no vertex appears more than once. Example:

$$T \to S \to R$$

Cycle: A path that takes you back to the start. Example:

$$Q \to S \to T \to Q$$

Adjacent Nodes: Nodes connected by an edge. Example: P and Q.
Adjacent Edges: Edges connected by a common node. Example: PT and PQ.

Completed graph Complete graphs are graphs that have an edge between every single vertex in the graph.

Example

Connected Graph Connected Graph are graphs that have at least one path between all Nodes

Example

Graphs in python

Graphs can be created and visualized in Python using the **networkx** module.

See https://networkx.org/ for the full documentation.

Code Snippet 3.1: The following Python code draws the Example Graph

```
import networkx as nx  #for creating and working with graphs
import matplotlib.pyplot as plt  #graph visualization

# Define the graph using sets
nodes = {"P", "Q", "R", "S", "T"}
edges = {("P", "Q"), ("P", "S"), ("P", "T"), ("Q", "T"), ("Q", "S"),
         ("Q", "R"), ("T", "S"), ("Q", "T"), ("S", "R")}

# Explicitly set positions for each node in a 2D space
pos = {'P': (0, 1), 'Q': (1, 1), 'R': (2, 0.5), 'S': (1, 0), 'T': (0, 0)}

# Define visualization options for the graph
options = {
    "font_size": 20,         # Font size for node labels
    "node_size": 2000,       # Size of the nodes
    "node_color": "white",   # Color of the nodes
    "edgecolors": "black",   # Color of the border of the nodes
    "linewidths": 2,         # Width of the node border
    "width": 2,              # Width of the edges
}

# Create an undirected graph object using NetworkX
G = nx.Graph()
G.add_nodes_from(nodes)  # Add the defined nodes to the graph
G.add_edges_from(edges)  # Add the defined edges to the graph

# Draw the graph with the specified positions and options
nx.draw_networkx(G, pos, **options)

# Display the graph using Matplotlib
plt.show()
```

Code Snippet 3.2: Basic Graph Commands

```
import networkx as nx  # Required for creating and working with graphs
import matplotlib.pyplot as plt  # Required for graph visualization

# Creating the graph
G = nx.Graph()                # Create a graph called G
G.add_nodes_from(My_Nodes)    # Add nodes from the set/list My_Nodes
G.add_node('A')               # Add a node called 'A' to the graph
G.add_edges_from(My_Edges)    # Add edges from the set/list My_Edges
G.add_edge('A', 'B')          # Add an edge between node A and node B to the graph

# Removing nodes and edges
G.remove_node('D')            # Remove node 'D' from the graph
G.remove_edge('A', 'B')       # Remove the edge between 'A' and 'B'

# Drawing the graph
nx.draw(G)           # Draw the graph
plt.show()           # Show the graph
plt.ion()            # Enable interactive mode for updating graph
plt.clf()            # Clear the graph before updating
plt.ioff()           # Turn off interactive mode so that the
                     # graph remains when the program closes

# Listing nodes and edges
print(G.nodes)  # List all nodes in the graph
print(G.edges)  # List all edges in the graph

# Getting graph properties
print(G.number_of_nodes())  # Get the number of nodes
print(G.number_of_edges())  # Get the number of edges
print(G.degree('A'))        # Get the degree of node 'A'
print(list(G.adj['A']))     # Get a list of neighbors of node 'A'

# Changing node and edge attributes
G.nodes['A']['color'] = 'blue'      # Add a color attribute to node 'A'
G.edges[('B', 'C')]['weight'] = 2.5 # Add a weight attribute to the edge ('B','C')
print(G.nodes['A'])                 # Access attributes of node 'A'
print(G.edges[('B', 'C')])          # Access attributes of edge ('B', 'C')

#  iterate through the nodes in a Graph
for node in G.nodes:
    print(node)  # Prints 'A', 'B', 'C'

#  print Graph with colours
node_colors=[]                      # Make a list to hold node colours
for node in G.nodes:                # Set all nodes to Grey
    G.nodes[node]['color']='Grey'
G.nodes['A']['color'] = 'red'       # Set node A to red
node_colors.clear()
for node in G.nodes:                # Make a list of node colours
    node_colors.append(G.nodes[node]['color'])
nx.draw_networkx(Grap, node_color=node_colors)
plt.show()
```

3.1 Exercise

1. Draw the graph $G = (V, E)$, where

   $V = \{A, B, C, D, E\}$

   $E = \{AB, BC, CE, BD, BE\}$

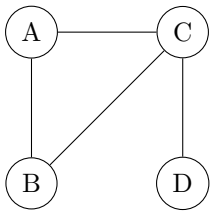
2. Install 'networkx' and 'matplotlib' by typing the following into the command prompt:

   ```
   pip install networkx
   pip install matplotlib
   ```

3. Verify the code provided in Code Snippet 3.1.

4. The 'options' and 'positions' are optional arguments. Delete them and observe how the graph looks by default.

5. Draw the following graph using Python:



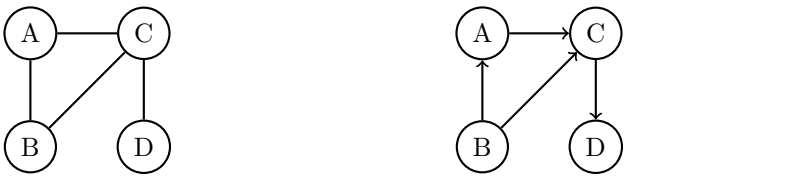
6. Add a new node 'E' to the graph and connect it to node 'C'.

7. Assign colors to nodes ('red' for A, 'blue' for B, etc.) and print out the node attributes.

8. Write a program that turns nodes red based on user input.

9. Write a program that adds nodes and edges to the graph based on user input.

Directed Graphs



(a) Undirected Graph (b) Directed Graph

Figure 3.2: Graphs can be undirected or directed

Undirected graphs represent a bidirectional relationship such as:

- Social networks, like Facebook where connections (friendships) are mutual.
- Network of roads connecting cities where roads can be traversed in both directions.

Directed graphs represent a directional relationship such as:

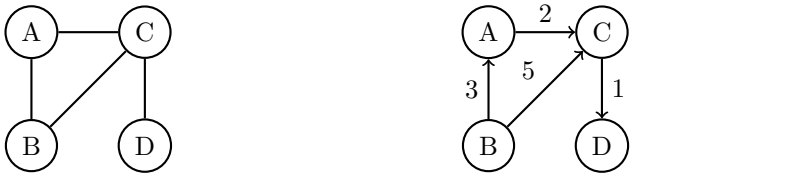
- Social networks, like X where the connection is one-way (a person can follow someone without being followed back)
- Web pages and links where the direction represents the hyperlink from one page to another.

Directed Graphs in Python

To create a directed graph in Python use `G = nx.DiGraph()` instead of `G = nx.Graph()`

Edges will be drawn in the direction specified i.e. `G.add_edge = ("A", "C")` will draw an arrow pointing to node `C`.

Weighted Graphs



(a) Unweighted Graph (b) Weighted Graph

Figure 3.3: Graphs can be unweighted or weighted

A weighted graph is used to represent quantifiable relationships such as costs, distances or capacities. Examples:

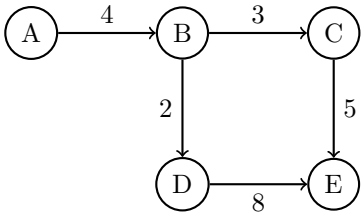
- Road networks where the weights represent distances or travel times.
- Network graphs where weights represent the bandwidth or capacity of the connections.

Unweighted graphs are used where the presence or absence of a connection is more important than the weight. Examples:

- A genealogy chart
- LAN (Local Area Network) topology
- University course dependency graph

Example

Given the following weighted graph, calculate:



1. Calculate the total weight of the graph.

---

2. The weighted path length from node A to node E via nodes B and D

---

3. The weighted path length from node A to node E via nodes B and C

Weighted Graphs in Python

```
# Specify edges in a list
edges = [
    ("A", "C", 2),  # Edge from A to C with weight 2.5
    ("B", "A", 3),  # Edge from B to A with weight 1.0
    ("B", "C", 5),  # Edge from B to C with weight 3.5
    ("C", "D", 1),  # Edge from C to D with weight 2.0
]

# Add weighted edges using the command:
G.add_weighted_edges_from(edges)

# Draw the directed graph with weights
edge_labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx(G, pos, **options)
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=15)
```

Tree

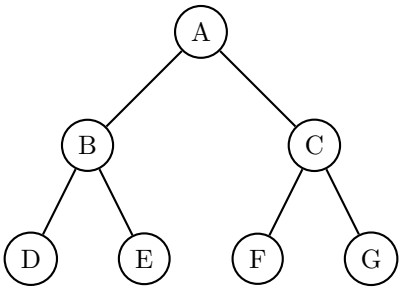


Figure 3.4: Graphs can be trees

A tree is an undirected graph that is connected and acyclic. Tree graphs represent hierarchical relationships such as:

- Family structures
- Computer file systems
- Taxonomy of Biological Classification
- Organizational Structures

Tree nomenclature:

- **Root Node:** The topmost node of the tree (node A in the diagrams).
- **Parent Node:** A node that has one or more child nodes (node B is a parent of nodes D and E).
- **Child Node:** A node that has a parent (nodes D and E are children of node B).
- **Leaf Node:** A node that has no children (nodes D, E, F, and G are leaves).
- **Subtree:** A tree formed by a node and all its descendants (the subtree rooted at B includes nodes B, D, and E).

A tree does not have to be drawn in the hierarchical representation. Figure 3.5 shows alternative ways to draw the tree from Figure 3.4.

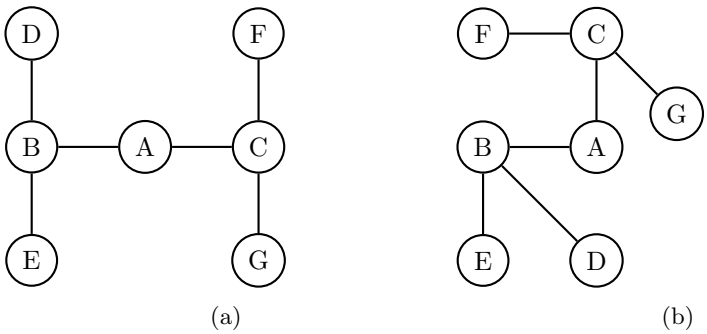


Figure 3.5: Different representations of the same tree structure in Figure 3.4

3.2 Exercise

1. Use python to draw the graphs from figure 3.3

Question 4 (9 marks)

a. Define the class of complete graphs and draw an example of a complete graph with five nodes. 2 marks

A graph that is formed from a subset of nodes of a graph and all the edges from the original graph that connect pairs of those nodes is called an induced subgraph.

b. The nodes of a complete graph are divided into two sets, U and V.

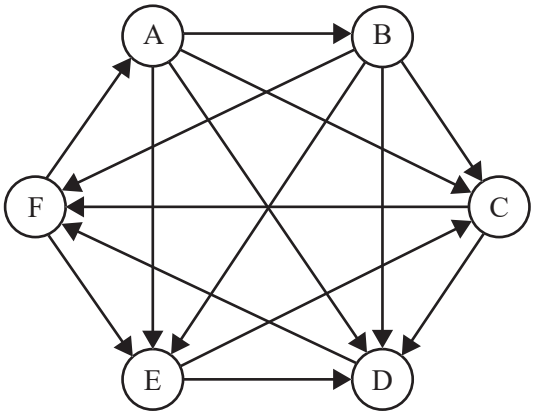
Explain why the two subgraphs induced by these two sets of nodes are also complete graphs. 2 marks

c. **i.** Consider the class of graphs created by taking an undirected complete graph with k nodes and then giving each edge in the graph a direction to create a directed graph. Let G be one such graph.

The following algorithm finds a path containing every node in G.

```
Algorithm findPath(G):
1   If G is empty Do
2     Return an empty list
3   If G has one node, v Do
4     Return a list containing v
5   Select a node v from G at random
6   Create a set V_in containing all nodes u for which
      (u,v) exists
7   Use the nodes V_in to create an induced subgraph
      called G_in
8   Create a set V_out containing all nodes u for which
      (v,u) exists
9   Use the nodes V_out to create an induced subgraph
      called G_out
10  in_path ← findPath(G_in)
11  out_path ← findPath(G_out)
12  path ← create a new list by joining together in_path,
      v and out_path in sequence
13  Return path
```

The *findPath* algorithm is executed on the graph shown below.



The node A is randomly selected in line 5 of the algorithm.

Find the values of *V_in* and *V_out* after the algorithm has executed up to the start of line 10. 1 mark

Graph Definition

Graphs can be defined as an ordered pair $G(V, E)$. Where V represents the set of vertices and E represents the set of edges. The graph above can be written as:

Graph G=(V,E), where

- V={P,Q,R,S,T}
- E={PQ, PT, PS, TQ, TS, QR, SR}

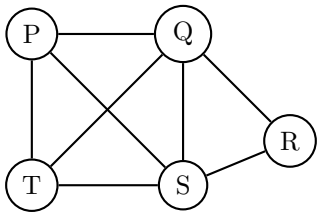


Figure 3.6: Example Graph

Directed Graphs

Directed graphs can be defined using a set E of ordered pairs to represent directed edges where the order of the pairs matters. For example, the edge (P, Q) is different from the edge (Q, P).

Graph G=(V,E), where

- V={P,Q,R,S,T}
- E={(P,Q), (P,T), (P,S), (T,Q), (T,S), (Q,R), (S,R), (Q,S)}

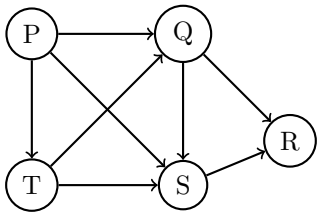


Figure 3.7: Directed Graph

Note: The set E may be written more simply as $E = \{PQ, PT, PS, TQ, TS, QR, SR, QS\}$

Weighted Graphs

Weighted graphs can be defined using a set E of of ordered triplets to represent the weights of the edges. For example, the edge (P, Q) with a weight of 3 can be represented as (P, Q, 3).

Graph G=(V,E), where

- V={P,Q,R,S,T}
- E={(P,Q,3), (P,T,2), (P,S,4), (T,Q,5), (T,S,6), (Q,R,1), (S,R,1), (Q,S,2)}

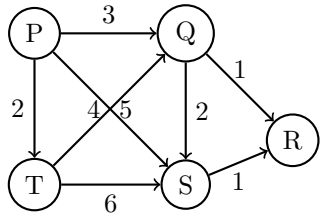


Figure 3.8: Weighted Graph

Matrix Representation

A graph can also be represented using an **adjacency matrix**. The adjacency matrix has a row and column for each vertex in the graph. The value in the matrix represents the weight of the edge between the vertices. If there is no edge between two vertices, the value is 0.

	P	Q	R	S	T
P	0	3	0	4	2
Q	0	0	1	2	5
R	0	0	0	0	0
S	0	0	1	0	6
T	0	0	0	0	0

(a) Weighted Graph

	P	Q	R	S	T
P	0	1	0	1	1
Q	0	0	1	1	1
R	0	0	0	0	0
S	0	0	1	0	1
T	0	0	0	0	0

(b) Directed Graph

Figure 3.9: Matrix Representation of Graphs

Adjacency List

An adjacency list is another way to represent a graph. It consists of a list of vertices and its adjacent vertices.

Example:

The adjacency list of the graph shown in figure 3.6 is:

E={(P, Q, T, S), (Q, P, T, R, S),(R, Q, S), (S,P, T, Q, R), (T, P, Q, S) }

Adjacency lists can be implemented using dictionaries where keys are vertices and values are lists of adjacent vertices.

```
E={
P: Q, T, S
Q: P, T, R, S
R: Q, S
S: P, T, Q, R
T: P, Q, S
}
```

Advantages of Adjacency Lists:

- Space Efficient: Uses less memory for sparse graphs.
- Easy to Traverse: Efficiently iterate over all neighbors of a vertex.

Disadvantages of Adjacency Lists:

- Less Efficient for Dense Graphs: Requires more memory for dense graphs compared to adjacency matrices.
- Slower Edge Lookup: Checking if an edge exists between two vertices can be slower compared to adjacency matrices.

3.3 Exercise

1. Draw the graph $G = (V, E)$ where:

   - $V = \{A, B, C, D, E\}$
   - $E = \{(A, B), (A, C), (B, D), (C, D), (D, E)\}$

2. Convert the graph defined above into an adjacency matrix.

3. Convert the graph defined above into an adjacency list.

4. Draw the graph $G = (V, E)$ where:

   - $V = \{A, B, C, D, E\}$
   - $E = \{(A, B, 3), (A, E, 2), (B, E, 4), (C, D, 1), (D, E, 5)\}$

5. Convert the weighted graph defined above into an adjacency matrix.

6. Convert the weighted graph defined above into an adjacency list.

7. Draw the graph represented by the following adjacency matrix

	P	Q	R	S	T
P	0	1	1	0	0
Q	1	0	0	1	1
R	1	0	0	1	0
S	0	1	1	0	1
T	0	1	0	1	0

8. Draw the weighted graph represented by the following adjacency matrix:

	P	Q	R	S	T
P	0	3	0	4	2
Q	0	0	1	2	5
R	0	0	0	0	0
S	0	0	1	0	6
T	0	0	0	0	0

9. Given the following adjacency list, draw the corresponding graph:

```
{
A: B, C
B: A, D, E
C: A, F
D: B
E: B, F
F: C, E
}
```

Spanning Tree

A spanning tree of a graph is a subgraph that includes all the vertices of the graph and is a tree. It spans all the vertices with the minimum number of edges needed to maintain connectivity.

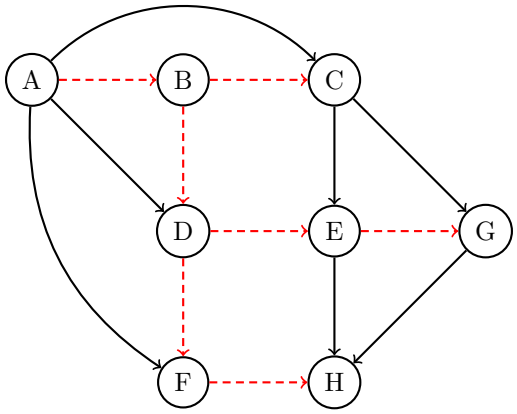
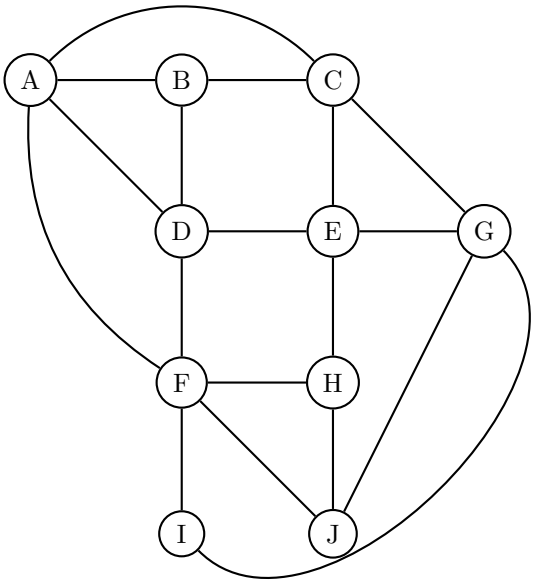


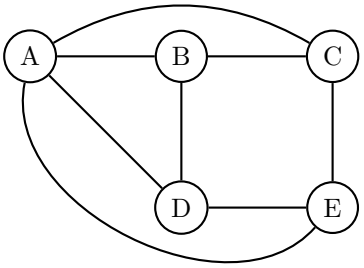
Figure 3.10: Graph with edges of the Spanning Tree highlighted in red.

3.4 Exercise

1. Highlight the edges of a spanning tree.



2. Given the following graph, find two possible spanning trees.



Decision Trees

Graphs can be used to model decision-making processes. A decision tree is a tree-like graph that represents a sequence of decisions and their possible outcomes. Decision trees are used in various fields, including computer science, data analysis, and artificial intelligence. Decision trees can be used to classify data. These classification trees are used in machine learning to predict outcomes based on input data.

Features of decision trees:

- Non-leaf nodes represent decisions to be made.
- Edges represent answers.
- Leaf nodes represent outcomes.

Example: Medical Diagnosis

Consider a decision tree for diagnosing a medical condition based on symptoms. The tree helps in making a decision about the diagnosis based on the presence or absence of certain symptoms.

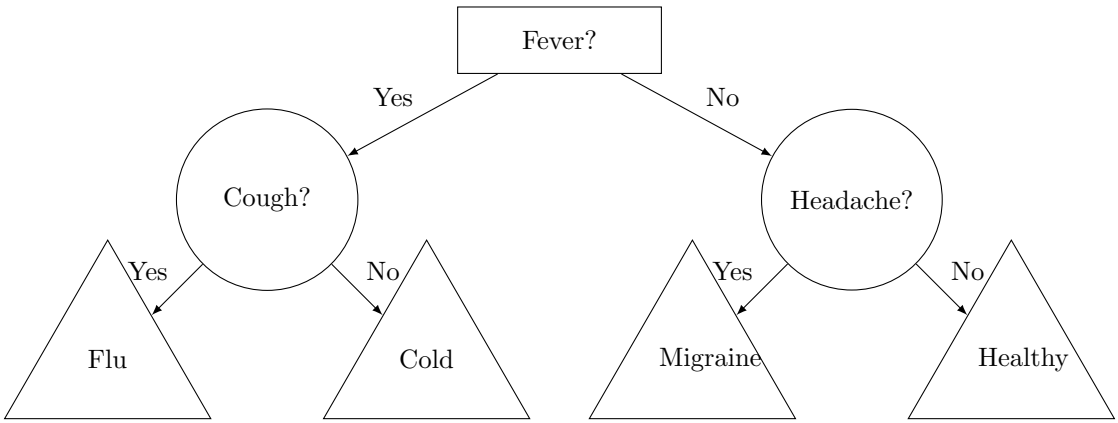


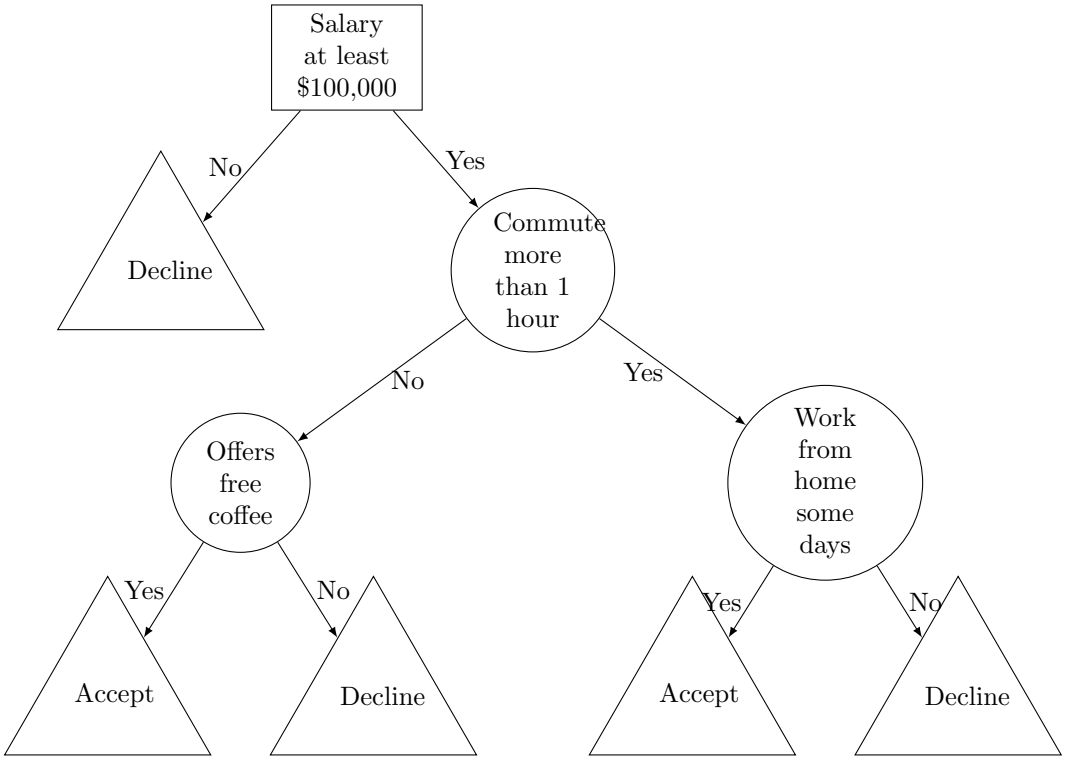
Figure 3.11: Decision Tree for Medical Diagnosis

In this example:

- If the patient has a fever and a cough, the diagnosis is Flu.
- If the patient does not have a fever and no headache, the diagnosis is Healthy.

Represent this decision tree using logical operations in pseudocode

Example: Job Offer Decision



1. What is the root node of the decision tree?

   _____

2. How many leaf nodes are there in the decision tree?

   _____

3. What decision is made at the root node?

   _____

4. What are the possible outcomes if the salary is $120,000?

   _____

5. What happens if the commute is more than 1 hour and the company offers free coffee?

   _____

6. How many levels does the decision tree have?

   _____

7. What is the significance of the leaf nodes in the decision tree?

   _____

- Let $D\{V, E\}$ be a decision tree for Job offers.
- Let o be a dictionary containing an offer.
- Let $P(u, x)$ be a function that returns Yes if the decision at node u is true for an offer with parameters x and no otherwise.

8. Write pseudocode for an algorithm that traverses the decision tree to accept or decline a job.

3.5 Exercise

1. Create a decision tree for diagnosing a computer problem based on symptoms. The tree should include the following decisions and outcomes:
    - Is the computer turning on?
        - Yes: Proceed to the next decision.
        - No: Check the power supply and connections.
    - Is the computer displaying any error messages?
        - Yes: Note the error message and look up the specific error code.
        - No: Proceed to the next decision.
    - Is the computer running slowly?
        - Yes: Check for high CPU or memory usage.
            - High CPU usage: Close resource-intensive applications.
            - High memory usage: Increase RAM.
        - No: Proceed to the next decision.
    - Is the computer connected to the internet?
        - Yes: Proceed to the next decision.
        - No: Ensure the network cable is connected or check Wi-Fi settings.
    - Is the computer making unusual noises?
        - Yes: Identify the source of the noise.
            - Fan noise: Replace the fan.
            - Hard drive noise: Backup data and replace the hard drive.
        - No: Proceed to the next decision.
    - Is the computer experiencing frequent crashes or freezes?
        - Yes: Update or reinstall the operating system.
        - No: The computer is functioning normally.
2. Draw a decision tree for the following scenario:

    A student is trying to decide what to do on a Saturday. The student's decision is based on the weather and whether they have homework to do. If it is raining, the student will stay home and do homework. If it is sunny and the student has homework, they will stay home and do homework. If it is sunny and the student does not have homework, they will go to the beach.
3. Write a function in Psudocode to determine what the student will do on a Saturday based on the weather and homework status.
4. 2023 ALGORITHMICS EXAM Question 3

    What do the non-leaf nodes in a decision tree represent?

    (A) the different answer options to decision questions

    (B) the results of the problem

    (C) the decision questions

    (D) the distance from the root node

Question 2 (10 marks)

a. Explain the concept of a decision tree. 2 marks

b. The table below identifies species of birds in terms of their main colours and typical size.

Bird	Colour	Size
mudlark	black and white	20 cm
rosella	red	25 cm
wattlebird	brown	35 cm
magpie	black and white	35 cm
noisy miner	grey	25 cm

In the space provided below, draw a decision tree for identifying the birds shown in the table above using colour as the first decision attribute. 4 marks

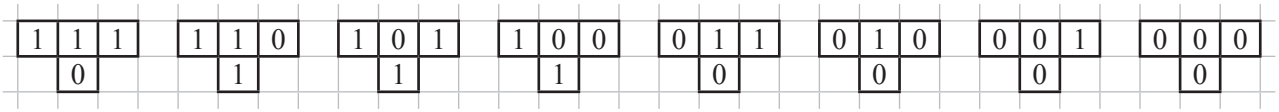
c. Let T be a decision tree for identifying birds and let x be a dictionary containing the information about a particular bird. For each non-leaf node u in T, let $P(u, x)$ be a function that returns True if the decision at node u is true for a bird with parameters x and False otherwise.

Write pseudocode for an algorithm that traverses the decision tree to identify a bird based on its features. 4 marks

Question 12 (8 marks)

A cellular automaton is a system in which each row is generated based on the row before it, in particular the cell above, the cell above to the left and the cell above to the right. The rules can vary, but for this question the rule is given as the following.

Rule



Assume the edges are considered 0, that is, the cells on the edge do not consider the cells on the other edge. For example, given the rule above with a row containing a single 1, the next few rows will be

0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

a. Given the input row | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |, determine the next row. 1 mark

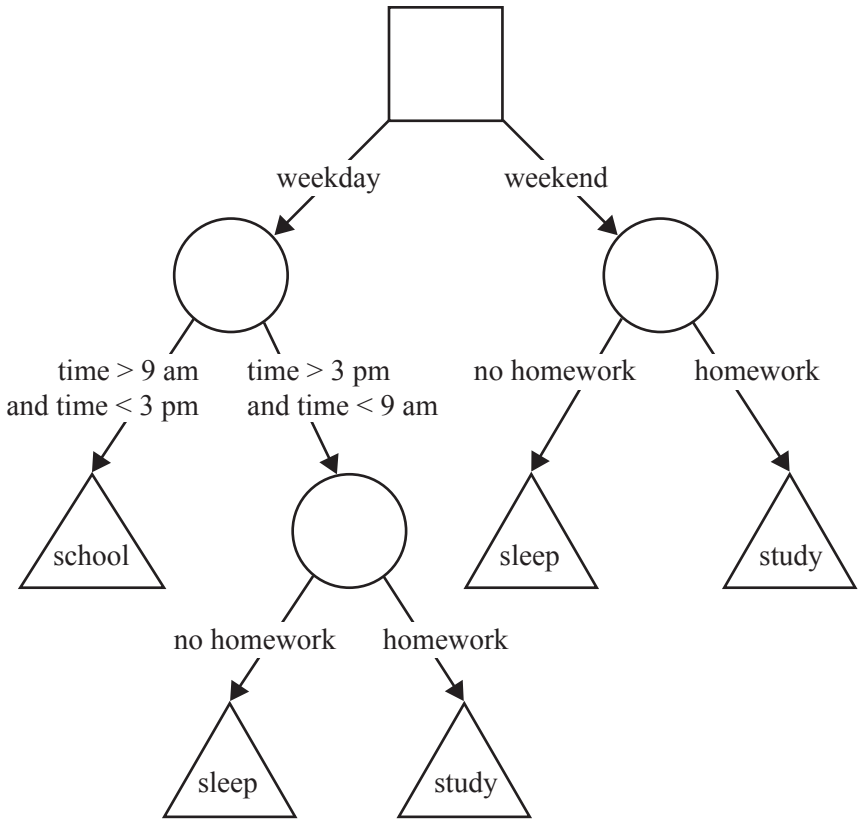
_____

b. Draw a decision tree to implement this cellular automata rule. 3 marks

c. Write pseudocode that takes an input array containing a combination of eight 0s and 1s, and generates *n*, the number of rows. Row 0 should contain the input row. 4 marks

Question 7 (4 marks)

Consider the decision tree below.



Represent this decision tree using logical operations in pseudocode.

State Graphs

State graphs, also known as state diagrams or finite state machines (FSM), are used to model the behavior of systems. They represent the states of a system and the transitions between those states based on inputs or events. State graphs are widely used in computer science, engineering, and various fields to design and analyze systems with a finite number of states.

State graphs are useful because they provide a clear and visual way to represent the dynamic behavior of systems. They help in understanding and designing systems by showing all possible states and transitions, making it easier to analyze and debug the system.

Once a state graph is created, graph algorithms can be used to solve problems related to the system's behavior, such as finding the shortest path between states or detecting cycles in the graph.

Key Components:

- **States:** Represented by nodes in the graph.
- **Inputs/Triggers:** Represented by edges in the graph.

Example: Marble Machine from Systems Engineering Class The Marchble machine from Systems Engineering 12 cab be modelled as a state graph with the following states and transitions

States:

- Idle
- move servo arm
- turn wheel

Transitions

- track wheel sennsor (idle to move servo arm)
- servo arm complete (move servo arm to turn wheel)
- wheel sensor (turn wheel to idle)

Draw a state graph for the vending machine scenario described above.

Example: Vending Machine A vending machine can be modeled as a state graph with the following states:

- Idle
- Read Coins
- Read customer selection
- Check stock
- Dispense item
- Return change
- Advise Customer Out of stock

When the vending machine is in the **Idle** state, inserting a coin transitions it to the **Read Coins** state. If valid coins are inserted, the machine moves to the **Read Customer Selection** state; otherwise, it returns to the **Idle** state, returning the coins. Once a selection is made, the machine transitions to the **Check Stock** state. In the **Check Stock** state, if the selected item is in stock, the machine transitions to the **Dispense Item** state; if the item is out of stock, it moves to the **Advise Customer Out of Stock** state. After dispensing the item, the machine transitions to the **Return Change** state, and once the change is returned, it goes back to the **Idle** state. If the machine advises the customer that the item is out of stock, the machine transitions to the **Return Change** state.

Draw a state graph for the vending machine scenario described above.

Example: Wolf, goat and cabbage problem.

A farmer with a wolf, a goat, and a cabbage must cross a river by boat. The boat can carry only the farmer and a single item. If left unattended together, the wolf would eat the goat, or the goat would eat the cabbage. How can they cross the river without anything being eaten?

Notation:

- M : man
- W : wolf
- G : goat
- C : cabbage

States: Pairs of subsets of {M,W,G,C} where the first subset of the pair represents which entities are on the initial side of the river and the second subset, the entities on the opposite side. For example, MGC-W means the man, goat and cabbage are on the initial side and the wolf is on the opposite side.

Transitions:

- m: the man crosses the river by himself
- w: the man crosses with the wolf
- g: the man crosses with the goat
- c: the man crosses with the cabbage

1. What is the start state?

   ---

2. What is the final state?

   ---

3. There are 16 possible "states", but some violate the constraints. List the violtaing states.

   ---

4. Construct the state diagram. You may omit the states and transitions that would violate the constraints.

3.6 Exercise

1. Consider a machine that has a keyboard as an input and prints letters to a screen. Draw a state diagram of a machine that produces this set of words, using a single start node and no more than 9 nodes total.
   - bat, back, sat, sack
   - boo, booo, boooo, ... [i.e. b followed by two or more o's]
   - moo, mooo, moooo, ... [i.e. m followed by two or more o's]

2. Suppose we're modelling an RC crane which is receiving a sequence of input commands, each of which is UP, DOWN, or BEEP. This crane has only two vertical positions, and starts in the high position. It should go into an ERROR end state if it is asked to go UP when it is in the high position or DOWN when it is in the low position. Once it is in the ERROR state, it stays in the ERROR state no matter what commands it receives. BEEP commands are legal at any point. Give a state diagram for this system, in which each edge corresponds to receiving a single command.

Node Attributes

Graphs can be combined with other ADTs to create more complex data structures. Graphs can be enhanced by adding attributes to nodes, providing additional information relevant to the nodes. These attributes are typically stored using arrays or dictionaries.

Arrays

- **Fixed Size**: Useful when the number of attributes is fixed and known in advance.
- **Efficient Access**: Fast access to elements, making it efficient for frequent operations.
- **Memory Usage**: More memory-efficient than dictionaries for small, fixed attributes.

Dictionaries

- **Flexibility**: Can store a variable number of attributes, useful when different nodes have different sets of attributes.
- **Ease of Use**: Easy addition, modification, and deletion of attributes.
- **Readability**: Attributes accessed by name, enhancing code readability.

Example Arrays Consider a Networks of sensors where nodes represent sensors deployed in a specific area, and edges represent communication links between them. Each sensor node might store a fixed set of values:

- Sensor ID (integer)
- Temperature reading (integer)
- Humidity reading (integer)
- Battery level (integer)

Describe the data structure that would be used to store the attributes of each sensor node and give an example.

Example Dictionaries: Consider a city map where nodes represent intersections, and edges represent roads. Each intersection (node) can have attributes such as

- Name (string)
- Coordinates (floats)
- Traffic management type (string: "sign", "lights", or "roundabout")
- Billboard sponsor (string)

Describe the data structure that would be used to store the attributes of each sensor node and give an example.

3.7 Exercise

1. Consider a graph where each node represents a city and each edge represents a direct flight between cities. Nodes store the following attributes: flight number, number of passengers, and departure time (in 24-hour format). What types of data structures would you use to store the attributes? Justify your answer.

2. Consider a graph where each node represents a student and each edge represents a friendship between students. Nodes store the following attributes: student ID, name, and grade level. What types of data structures would you use to store the attributes? Justify your answer.

3. Consider a graph where each node represents a task in a project and each edge represents a dependency between tasks. Nodes can store the following attributes: task ID, description, project team, task type, estimated completion time, ect. What types of data structures would you use to store the attributes? Justify your answer and provide an example.

4. Consider a Organizational chart where each employee has the following attributes: name, age, and job title and is connected to other employees based on their reporting structure.

   (a) Describe a data structure that could be used to store the attributes of the organizational chart.

   (b) Implement this data structure in python for the graph shown above.

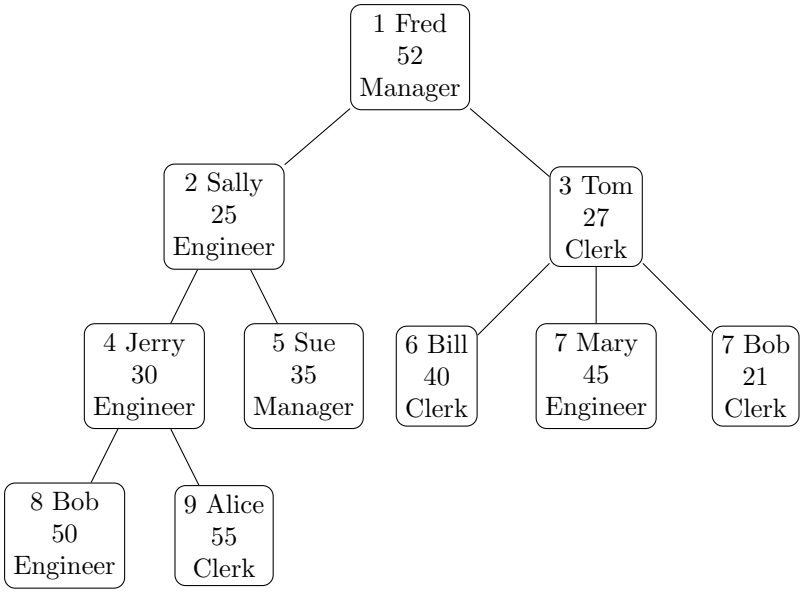


Figure 3.12: Organizational chart

5. Consider a website where each webpage has a unique URL, a title, and a last updated date and is linked to other webpages as shown below.

   (a) Describe a data structure that could be used to store the attributes of the organizational chart.

   (b) Implement this data structure in python for the graph shown above.

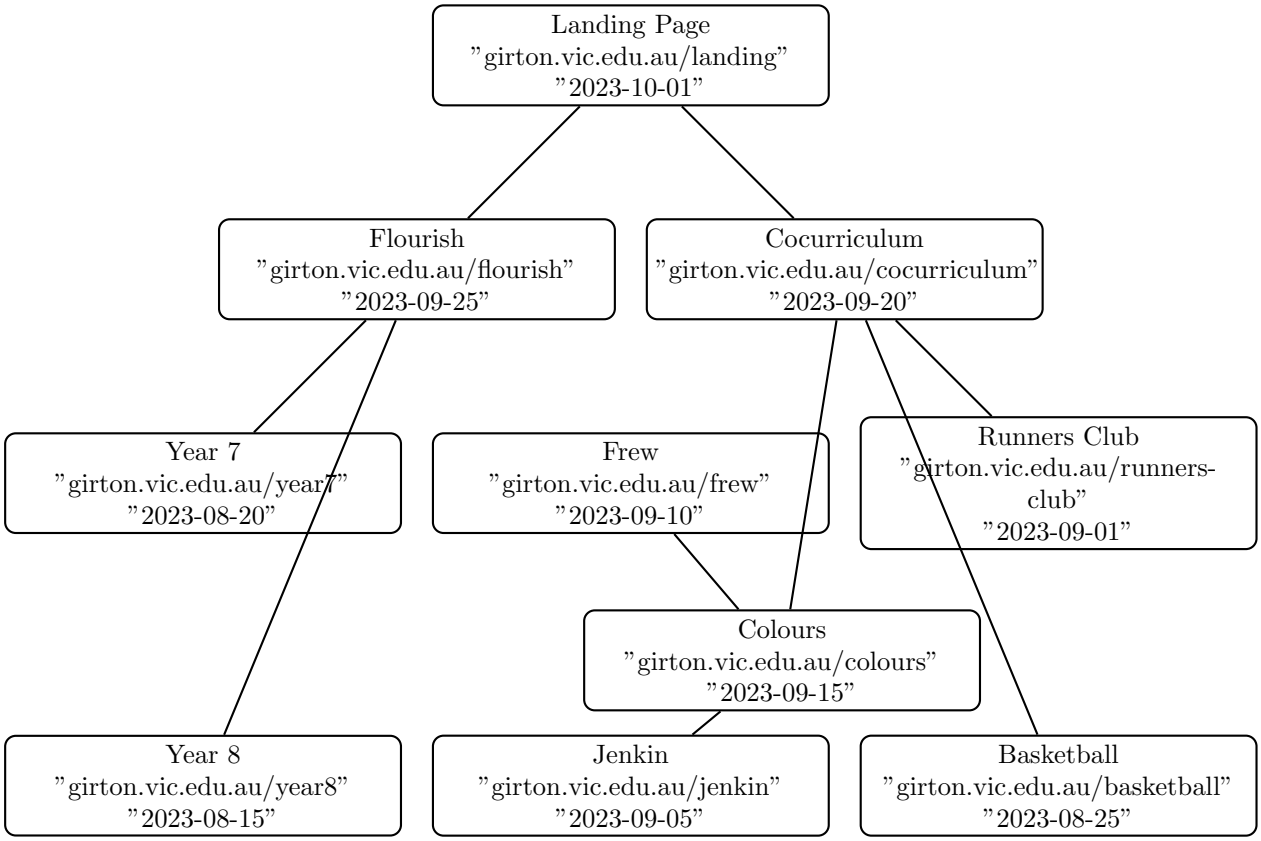


Figure 3.13: website

Chapter 4

Traversal Algorithms

Area of Study 2: Algorithm design Outcome 2

Learning Intentions

- Key knowledge
  - graph traversal techniques, including breadth-first search and depth-first search

Traversal Algorithms

"Traversal: the action or fact of moving, travelling, or extending through or across something."(Oxford English Dictionary, Oxford University Press, 2023)

Graph traversal algorithms are used to visit and process all vertices in a graph. These algorithms are essential for exploring the structure of a graph and can be used to solve a variety of problems such as finding certain properties of the graph. They are fundamental building blocks for other, more specialized graph algorithms like Prim's MST and Dijkstra's shortest path algorithms.

Depth-First Search (DFS)

- Starts at a chosen vertex (root) and explores as far as possible along each branch before backtracking.
- Uses a **stack** or recursion to keep track of vertices to visit.
- Can be prone to infinite loops if the graph contains cycles and no mechanism is used to detect and avoid revisiting visited vertices.

Algorithm:

```
Algorithm Depth First Search(Graph G, start node v):
    Create a list for visited nodes
    Create a stack for nodes to visit
    Push v to the stack
    while stack is not empty
        u <- Pop node from stack
        Add u to the visited list
        If nodes adjacent to u are not in the stack or visited list
            push adjacent nodes to the stack
    return visited list
```

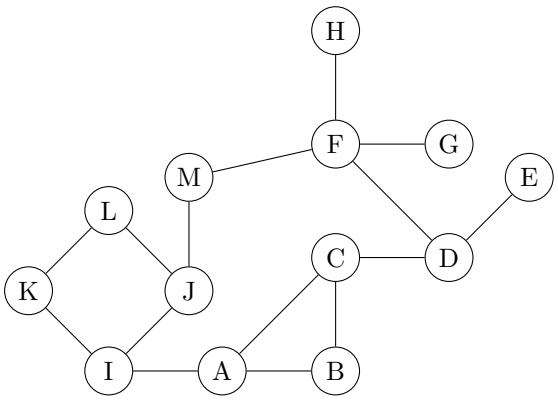
Breadth-First Search (BFS)

- Starts at a chosen vertex (root) and visits all its neighbors before moving on to the next level.
- Uses a **queue** to keep track of vertices to visit.
- Ensures that all vertices at a given distance from the root are visited before moving on to the next distance level.

Algorithm:

```
Algorithm Breadth First Search(Graph G, start node v):
    Create a list for visited nodes
    Create a queue for nodes to visit
    Enqueue start node
    while queue is not empty
        u <- Dequeue node
        Add u to the visited list
        If nodes adjacent to u are not in the queue or visited list
            enqueue nodes adjacent to u
```

Example Perform the Breadth-first search and Depth-first search algorithms on the following graph and list the order in which the nodes are visited. Start at node A. Where multiple options exist, travese the nodes in alphabetical order.

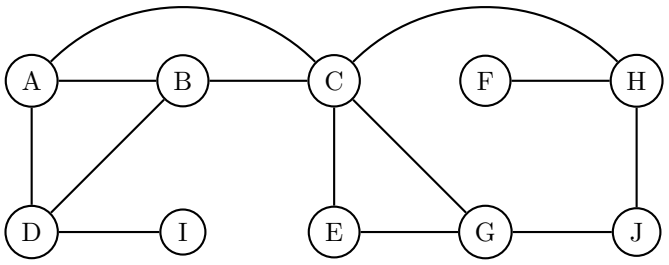


4.1 Exercise

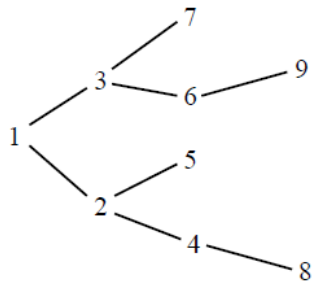
1. Starting at node A, list the order in which the nodes are visited using a

   a. **Depth-first** search algorithm, ensuring that nodes are visited in alphabetical order when multiple paths are possible:

   b. **Breadth-first** search algorithm, ensuring that nodes are visited in alphabetical order when multiple paths are possible:



2015 ALGORITHMICS EXAM Use the following information to answer Questions 2 & 3



2. 2015 ALGORITHMICS EXAM Question 2

   The first six nodes visited, starting at node 1, in a depth-fi rst search could be

   A. 123456

   B. 124637

   C. 124853

   D. 136798

3. 2015 ALGORITHMICS EXAMQuestion 3

   The first six nodes visited, starting at node 1, in a breadth-fi rst search could be

   A. 124653

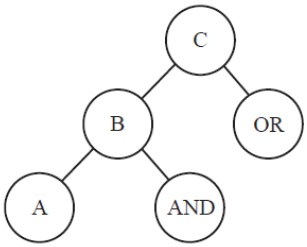
   B. 123456

   C. 124563

   D. 123458

4. 2017 ALGORITHMICS EXAM Question 6

   Consider the following graph representation of pseudocode. Starting at C, in which order would each of the nodes be first examined if they were traversed using depthfirst search? (Alphabetical order is used when there is more than one option.)



   A. A, AND, B, OR, C

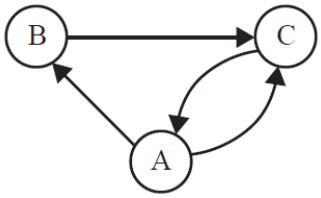
   B. A, AND, OR, B, C

   C. C, OR, B, AND, A

   D. C, B, A, AND, OR

5. 2018 ALGORITHMICS EXAM Question 14

   Sameera runs an algorithm on the graph below to compute a path from A to B. The algorithm becomes trapped in a cycle.



   A. PageRank algorithm

   B. Dijkstra's algorithm

   C. breadth-first search without marking previous nodes

   D. depth-first search without marking previous nodes

6. Implement the traversal algorithms on the graph from question 1 by

   (a) Defining a the data structure $G(V, E)$.

   (b) Write python code to implement the Breadth-first search algorithm.

   (c) Write python code to implement the Depth-first search algorithm.

7. Implement the traversal algorithms on the graph from Figure 3.12

   (a) Defining a the data structure $G(V, E)$.

   (b) Write python code to implement the Breadth-first search algorithm.

   (c) Write python code to implement the Depth-first search algorithm.

8. The depth-first search algorithm can be implemented using recursion. Write a recursive function to implement the depth-first search algorithm on the graph from question 1.

9. The breadth-first search algorithm can be modified to calculate the distances of all vertices from the source. Write a python function to calculate the distances of all vertices from the source on the graph from question 1.

Chapter 5

Graph Algorithms

Area of Study 2: Algorithm design Outcome 2

Learning Intentions

- Key knowledge
  - specification, correctness and limitations of the following graph algorithms
    - Prim's algorithm for computing the minimal spanning tree of a graph
    - Dijkstra's algorithm and the Bellman-Ford algorithm for the single-source shortest path problem
    - the Floyd-Warshall algorithm for the all-pairs shortest path problem and its application to the transitive closure problem
    - the PageRank algorithm for estimating the importance of a node based on its links
- Key skills
  - select appropriate graph algorithms and justify the choice based on their properties and limitations
  - explain the correctness of the specified graph algorithms
  - implement algorithms, including graph algorithms, as computer programs in a very high-level programming language that directly supports a graph ADT

Prim's minimal spanning tree algorithm

What does it do?

Prim's algorithm finds a Minimum Spanning Tree (MST) for a connected, weighted, undirected graph. A MST is a subset of the graph that is a tree with the minimal weight.

Prim's algorithm starts with a single node and grows the tree by adding the lowest-weight edge that connects the tree to a new node. This is a greedy algorithm.

What are its inputs?

A Graph and a starting node.

What are its outputs?

A MST.

Algorithm

```
Algorithm Prims(G,S)
Initialize a list MST for the edges of the MST
Initialize a set Visited for visited nodes
Initialize PQ a priority queue for edges (minimum first)

Enqueue adjacent edges of S to PQ
While PQ is not empty:
    u <- peek at PQ
    v <- destination node of u
    dequeue PQ
    If v is not in visited:
        Add v to Visited
        Add u the MST
        Add adjacent edges of v to PQ
End while
Return the MST
```

What are its limitations?

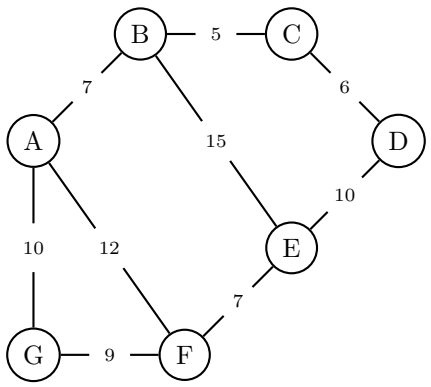
Prims requires a **Connected, Undirected** Graph. It can handle negative weights.

There can be more than one MST. The MST produced depends on how the priority queue is processed.

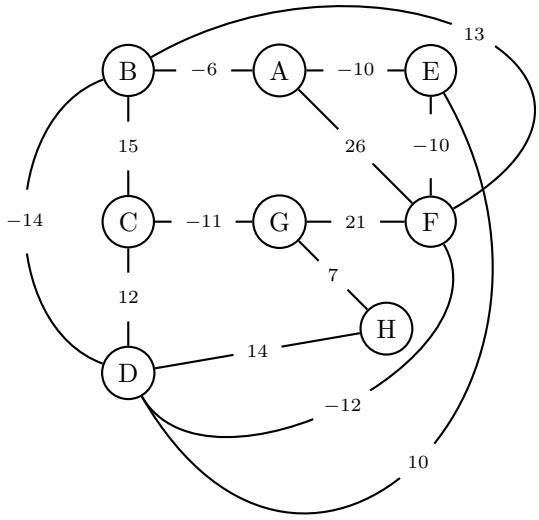
5.1 Exercise

1. Use Prims's algorithm to find the MST of the following graphs and calculate its weight.

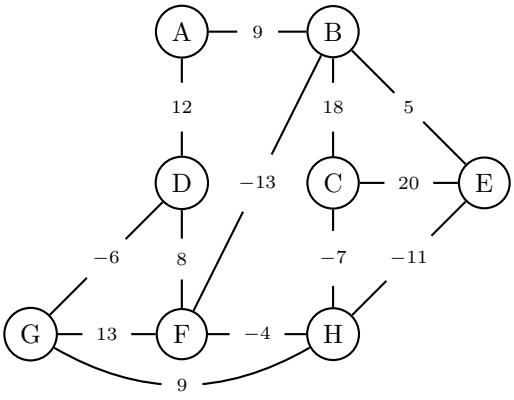
   (a)



(b)


B
A
E
C
G
F
H
D
13
−6
−10
15
26
−10
−14
−11
21
7
12
14
−12
10

(c)


A
9
B
12
18
5
D
−13
C
20
E
−6
8
−7
−11
G
13
F
−4
H
9

2. VCAA 2015 Question 6
To guarantee a unique minimal spanning tree in a graph

   A. there must be no cycles.

   B. running Prim's algorithm is sufficient.

   C. each edge must have a unique weight.

   D. repeated edge weights must be odd in number.

3. VCAA 2016 Question 9
A connected, undirected graph with distinct edge weights has maximum edge weight emax and minimum edge weight emin. Which one of the following statements is false?

   A. emax is not in any minimal spanning tree.

   B. Every minimal spanning tree of the graph must contain emin.

   C. Prim's algorithm will generate a unique minimal spanning tree.

   D. If emax is in a minimal spanning tree, its removal will disconnect the graph.

4. VCAA 2018 Question 7
When testing to find out whether a graph has multiple connected components, which one of the following modifications to a basic implementation of Prim's algorithm is the most appropriate?

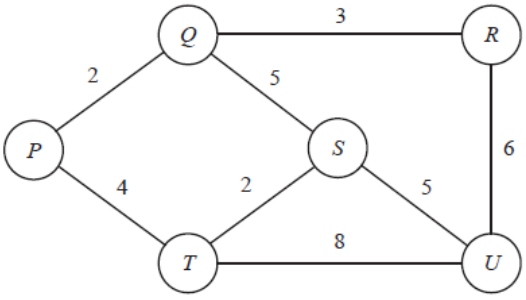
   A. Choose the maximum weighted edge to connect a node to the growing tree.

   B. Run Prim's algorithm on every node and keep track of unique sets of nodes.

   C. Delete all existing edges of the existing graph and create new edges to connect each node.

   D. No modifications need to take place as Prim's algorithm can already test for multiple disconnected components.

5. VCAA 2022 Question 5
Below is the graph of G



When Prim's algorithm is run on G to find its minimal spanning tree, the order in which the algorithm visits nodes could be

   A. P, Q, R, S, T, U

   B. R, Q, P, S, T, U

   C. S, T, P, U, Q, R

   D. T, S, P, Q, R, U

Question 6 (10 marks)

A building services engineer is designing the plumbing for a new apartment building. The building will have a central hot-water system serving all the apartments. This requires a network of pipes that form a spanning tree that connects every apartment to the central hot-water system.

Hot-water pipes are expensive, and so the overall length of pipe to be used in the design needs to be minimised. The distance from the central hot-water system to each apartment also needs to be minimised so that less water is wasted each time a resident needs hot water.

Consider a graph that has the central hot-water system and each apartment as nodes, and all possible pipe routes as edges, with edge weights indicating the length of each pipe.

a. Explain why Prim's algorithm would be appropriate for designing an optimal hot-water network of pipes if only the total cost of the pipes needed to be considered. 1 mark

b. Let *Prim*(*G*) be a function that takes as input a weighted graph, *G*, and returns a minimum spanning tree.

Write pseudocode for an algorithm that returns all the possible minimum spanning trees of any given weighted graph. 4 marks

c. Instead, if the only goal was to minimise the length of pipe from the central hot-water system to each apartment, what algorithm would be most suitable for designing the pipe network? Explain why your chosen algorithm best meets the requirements of this problem. 2 marks

d. Describe an algorithmic approach to finding which pipes to use for connecting the apartments that appropriately considers both of the desired goals: minimising the total amount of piping and minimising the length of pipe from the central hot-water system to each apartment. 3 marks

Dijkstra's shortest path algorithm

Dijkstra's algorithm finds the shortest path from a starting node to all other nodes in a connected, weighted, directed or undirected graph.

Dijkstra's algorithm maintains a set of visited nodes and iteratively selects the unvisited node with the smallest known distance, updating the distances to its neighbors. This is a greedy algorithm.

What are its inputs?

A graph and a starting node.

What are its outputs?

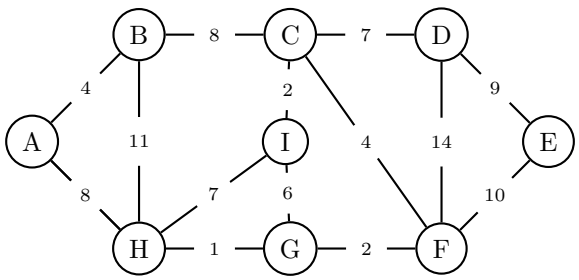
A table of shortest distances from the start node to all other nodes.

(Optional) A path reconstruction list that allows extracting the exact shortest path from the start node to any destination.

Algorithm

```
Algorithm Dijkstra(G, S)
Initialize a dictionary Distances with each node in G as a key and infinity as the value
Initialize a dictionary Previous with each node in G as a key and None as the value
Initialize a set Visited for tracking visited nodes
Initialize PQ a priority queue for nodes (minimum distance first)

Distances[S] <- 0
Enqueue (0, S) into PQ
While PQ is not empty:
    u <- peek at PQ
    dequeue PQ
    Add u to Visited

    For each unvisited neighbor v of u:
        Calculate new distance d <- Distances[u] + weight(u, v)
        If d is smaller than the current Distances[v]:
            Update Distances[v] <- d
            Update Previous[v] <- u
            Enqueue (d, v) into PQ

End while
Return Distances, Previous
```

Example Use Dijkstra's algorithm to find the shortest path from node A to all other nodes in the following graph.

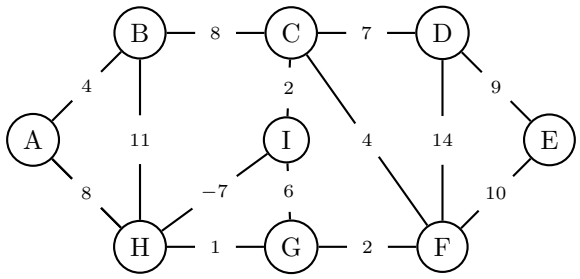


What are its limitations?

Dijkstra's does not work correctly with negative edge weights, as it assumes that once a node's shortest distance is found, it will not change.

If negative weights are present, Bellman-Ford's algorithm is required.

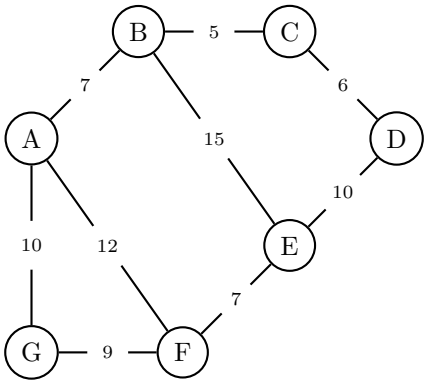
Example Use Dijkstra's algorithm to find the shortest path from node A to all other nodes in the following graph.



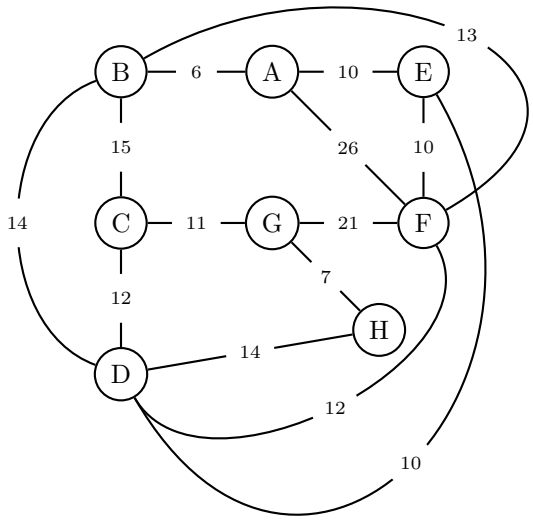
Exercise

1. Use Dijkstra's algorithm to find the shortest path from the given start node to all other nodes in the following graphs.

   (a) Start from node A



   (b) Start from node A



2. VCAA 2016 Question 10
   Dijkstra's single-source shortest path algorithm in an undirected graph reports distances from the source to each node.

   These distances

   A. are the shortest possible distances to every destination node.

   B. are never the shortest possible distances when negative edge weights are present.

   C. may be the shortest possible distances when negative edge weights are present.

   D. may not always be the shortest possible distances when all edge weights are positive.

3. Which of the following statements is true about Dijkstra's algorithm?

   A. It can handle negative edge weights.

   B. It always finds the shortest path in a graph with non-negative weights.

   C. It only works for undirected graphs.

   D. It requires an adjacency matrix to work correctly.

4. What happens if Dijkstra's algorithm encounters a node with a negative edge weight?

   A. The algorithm will still produce the correct shortest path.

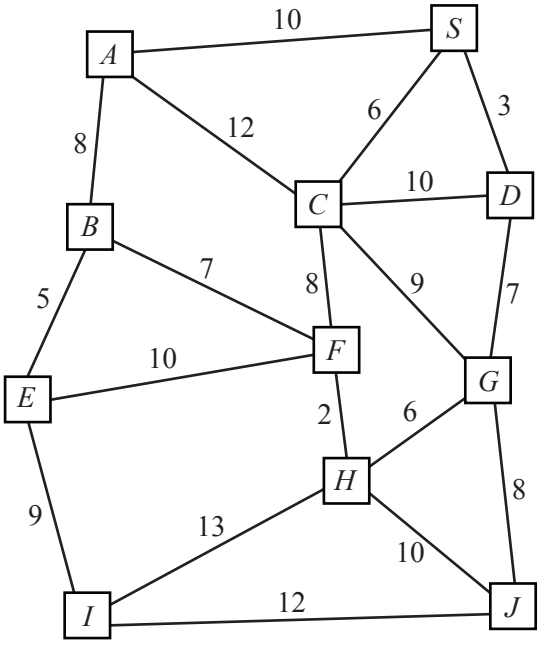
   B. The algorithm may produce incorrect results.

   C. The algorithm will run indefinitely.

   D. It will ignore the negative edge and continue.

Question 13 (7 marks)

Below is a graph representation of a possible way in which a collection of computers can be connected. Each computer is labelled with a letter and is a node in the graph. Cables that are used to connect the computers are shown as edges and the length of each cable is given as an edge weight.

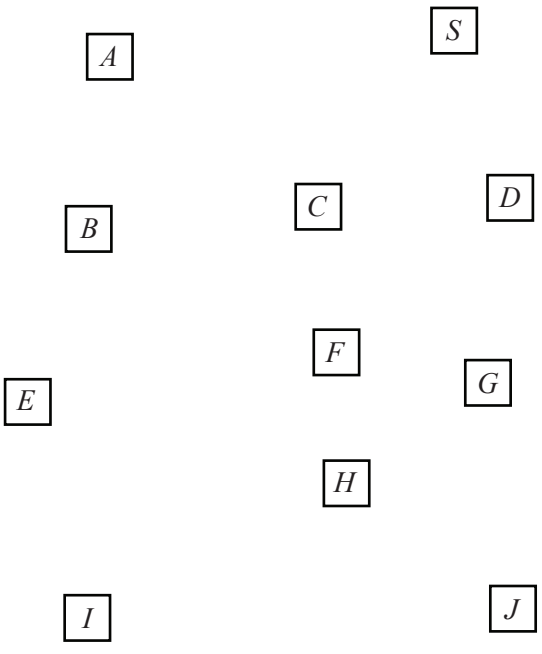


The collection of computers needs to be connected with cables such that the following conditions are met:

- Condition 1: There are no cycles.
- Condition 2: The shortest length of cabling is used from *S*, the source, to every other computer while the total cable length for the whole network is the smallest possible length.

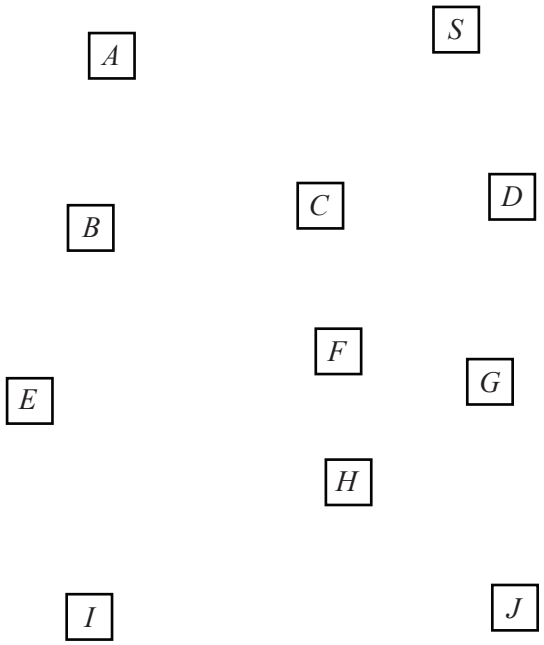
a. Draw the graph produced by Prim's algorithm and indicate the condition(s) that the graph meets. 2 marks

Condition(s) met _____



b. Draw the graph produced by Dijkstra's algorithm and indicate the condition(s) that the graph meets. 2 marks

Condition(s) met _____



c. Is there a modification to Dijkstra's algorithm that will allow for both Condition 1 and Condition 2 to be met? Explain your answer. 3 marks

_____

_____

_____

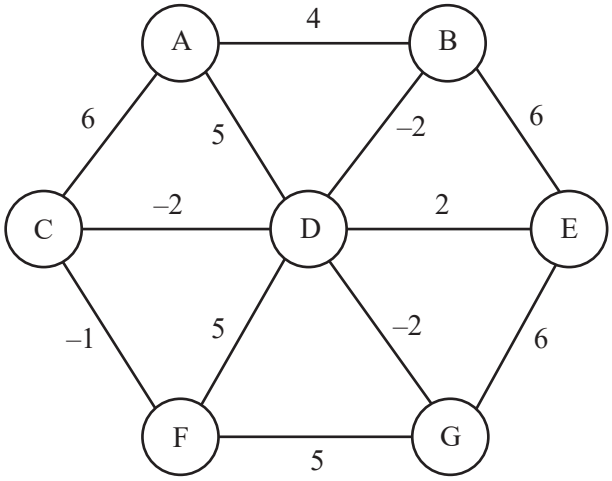
_____

_____

_____

Question 1 (5 marks)

Consider the following graph with edge weights as shown.



a. What is the distance of the C-D-E-G path? 1 mark

b. Write down the degree of node G. 1 mark

c. Consider running Dijkstra's algorithm, starting from node A, to find the shortest distance to all other nodes.

i. What property of this graph would make Dijkstra's algorithm unsuitable? 1 mark

ii. Find a node for which the distance from node A that Dijkstra's algorithm returns is incorrect. State the distance found by Dijkstra's algorithm and the true shortest distance, respectively, in your response. 2 marks

Bellman-Ford shortest path algorithm

What does it do?

The Bellman-Ford algorithm finds the shortest path from a starting node to all other nodes in a weighted graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs with negative edge weights and is also capable of detecting negative weight cycles.

The algorithm iterates through all edges V - 1 times, updating the shortest path estimate for each node.

An additional iteration is performed to check for the presence of negative weight cycles.

What are its inputs?

A graph with weighted edges and a starting node.

What are its outputs?

A list of shortest distances from the starting node to all other nodes or an indication of whether a negative weight cycle exists.

Algorithm

```
Algorithm Bellman-Ford(G(V,E), S)
Initialize Distances as a dictionary
For each node v in V:
    Distances[v] <- infinity
Distances[S] <- 0

Initialize Previous as a dictionary (to reconstruct paths)

For (V - 1) times:
    For each edge (u, v) with weight w:
        If Distances[u] + w < Distances[v]:
            Distances[v] = Distances[u] + w
            Previous[v] = u

// Check for negative weight cycles
For each edge (u, v) with weight w:
    If Distances[u] + w < Distances[v]:
        Return "Negative weight cycle detected"

Return Distances and Previous
```

The Bellman-Ford algorithm works regardless of the order in which edges are processed. The shortest path distances will converge within $V - 1$ iterations.

Why $V - 1$ Iterations? A shortest path in a graph with V vertices has at most $V - 1$ edges. If there were more than $V - 1$ edges in a shortest path, it must contain a cycle.

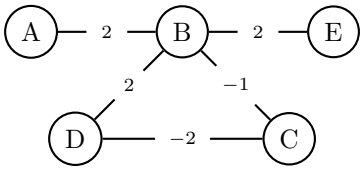
What are its limitations?

Bellman-Ford is slower than Dijkstra's algorithm for large graphs.

What if there are negative weighted cycles

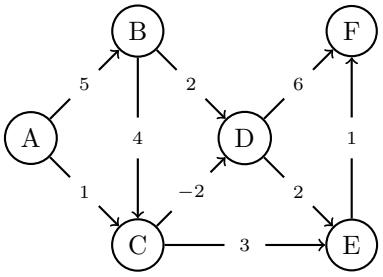
A negative weight cycle is a path that starts and ends at the same vertex, with a total negative weight. In a graph containing a negative weight cycle, the concept of a shortest path becomes meaningless.

Consider the graph below. What is the shortest path from A to D?



Example

Use Bellman-Ford's algorithm to find the shortest path from node A to all other nodes in the following graph.

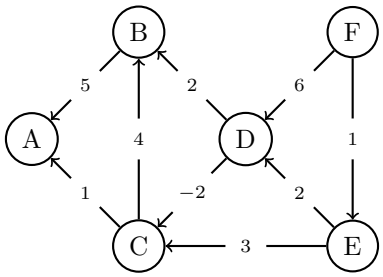


A	B	C	D	E	F
0					
0					
0					

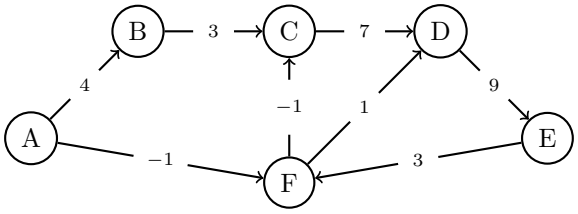
Edge List	Iteration 1	Iteration 2	Iteration 3
$(A, B, 5)$			
$(A, C, 1)$			
$(B, C, 4)$			
$(B, D, 2)$			
$(C, E, 3)$			
$(D, E, 2)$			
$(E, F, 1)$			
$(D, F, 6)$			
$(C, D, -2)$			

5.2 Exercise

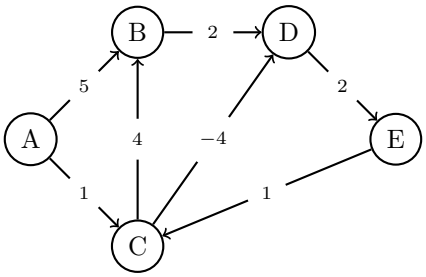
1. Use Bellman-Ford's algorithm to find the shortest path from node **F** to all other nodes in the following graphs.



2. Use Bellman-Ford's algorithm to find the shortest path from node **A** to all other nodes in the following graphs.



3. Use Bellman-Ford's algorithm to find the shortest path from node **F** to all other nodes in the following graphs.



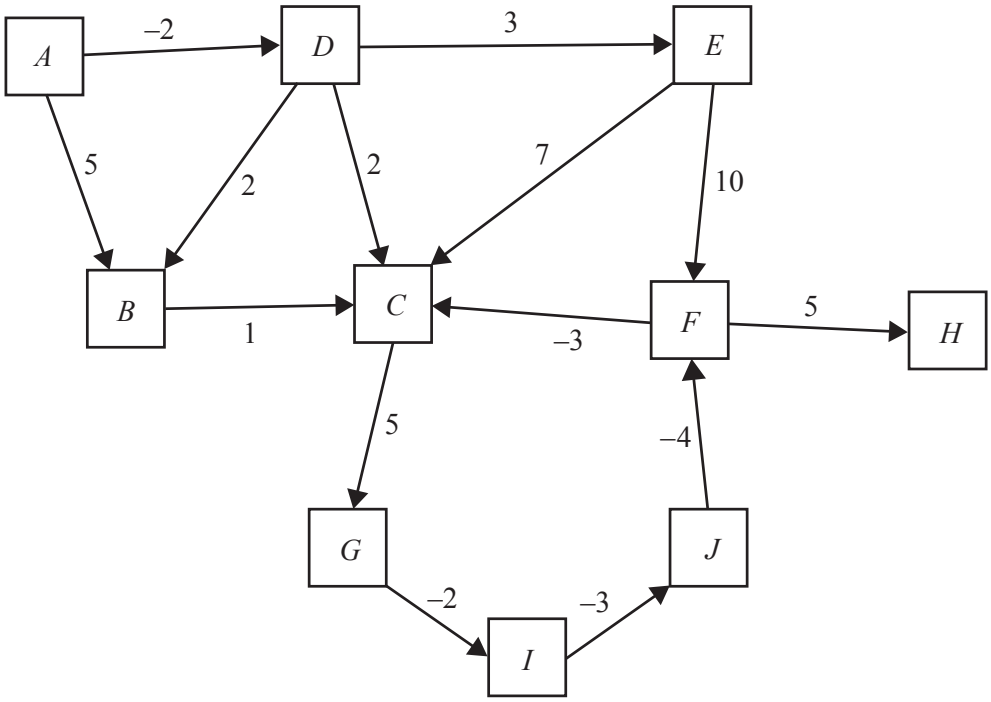
4. VCAA 2015 Q14 For a graph G with n nodes, a student runs the Bellman-Ford algorithm for $n-1$ iterations. She then runs one more iteration and notices the shortest path between two nodes has reduced.

   (a) What property of G has the testing established? (1 mark)

   (b) Explain whether the Bellman-Ford algorithm should be used to fi nd a shortest path solution in this example. (2 marks)

Question 7 (3 marks)

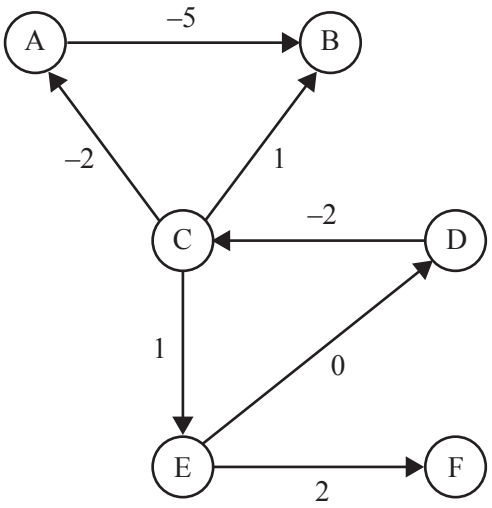
A student runs Bellman-Ford's single-source shortest path algorithm on the following directed graph using node *A* as the source. After nine iterations, she notes the distance from *A* to each of the other nodes. She then runs a tenth iteration of the algorithm and notes the distance from *A* to each of the other nodes.



a. Which nodes will show a change in distance from source node *A* between the ninth and tenth iterations? 1 mark

b. Explain why some nodes have remained the same distance from source node *A* while others have a new distance. 2 marks

Question 12 (2 marks)



Explain why the graph above is not a suitable input to a naive implementation of the Bellman-Ford algorithm.

Question 13 (3 marks)

Describe DNA computing and explain how it can be used as an alternative method of computation. Provide an example as part of your explanation.

Floyd-Warshall all-pairs shortest path algorithm

What does it do?

The Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in a weighted graph. The algorithm iteratively updates a distance matrix by considering each vertex as an intermediate step to check if a shorter path exists.

What are its inputs?

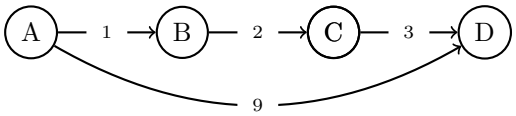
A weighted graph represented as an adjacency matrix.

What are its outputs?

An adjacency matrix containing the shortest distances between all pairs of vertices.

Algorithm

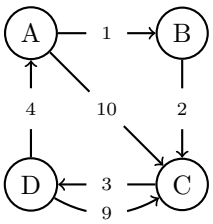
```
Algorithm Floyd-Warshall(G(V,A))
where V is the set of vertices and A is an adjacency matrix
Initialize a distance matrix D with the same dimensions as A
D <- A
For each k in V:
    For each i in V:
        For each j in V:
            if D[i][j] >  D[i][k] + D[k][j] then
                D[i][j] <- D[i][k] + D[k][j]
Return D
```



What are its limitations?

- The Floyd-Warshall algorithm has a time complexity of $O(V^3)$, making it inefficient for large graphs.
- The space complexity is $O(V^2)$, which may be impractical for graphs with a very high number of vertices.
- It does not work correctly if the graph contains negative weight cycles since it assumes that a shortest path exists.

Example Use the Floyd-Warshall algorithm to find the shortest paths between all pairs of vertices in the following graph.



The adjacency matrix for the graph is

D^0	A	B	C	D
A	0	1	10	∞
B	∞	0	2	∞
C	∞	∞	0	3
D	4	∞	9	0

For the first iteration k = A
The distance matrix is D^A

D^A	A	B	C	D
A	0	1	10	∞
B	∞	0		
C	∞		0	
D	4			0

now iterate through i and j for
$if D^0[i][j] > D^0[i][A] + D^0[A][j]$
Note: the row/column for A does not change.

$if D^0[A][B] > D^0[A][A] + D^0[A][B]$

$if D^0[B][C] > D^0[B][A] + D^0[A][C]$

$if D^0[B][D] > D^0[B][A] + D^0[A][D]$

$if D^0[C][B] > D^0[C][A] + D^0[A][B]$

$if D^0[C][D] > D^0[C][A] + D^0[A][D]$

$if D^0[D][B] > D^0[D][A] + D^0[A][B]$

$if D^0[D][C] > D^0[D][A] + D^0[A][C]$

For the second iteration k = B
The distance matrix is D^B

D^B	A	B	C	D
A	0			
B		0		
C			0	
D				0

Note: the row/column for B does not change

if $D^A[A][C] > D^A[A][B] + D^A[B][C]$

$if D^A[A][D] > D^A[A][B] + D^A[B][D]$

$if D^A[C][A] > D^A[C][B] + D^A[B][A]$

$if D^A[C][D] > D^A[C][B] + D^A[B][D]$

$if D^A[D][A] > D^A[D][B] + D^A[B][A]$

$if D^A[D][C] > D^A[D][B] + D^A[B][C]$

For the third iteration k = C
The distance matrix is D^C

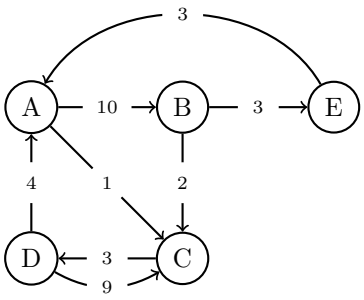
D^C	A	B	C	D
A				
B				
C				
D				

For the third iteration k = D
The distance matrix is D^D

D^D	A	B	C	D
A				
B				
C				
D				

5.3 Exercise

1. Consider the following graph.



   (a) Create the adjacency matrix for the graph.

   (b) Perform the first iteration of the Floyd-Warshall algorithm.

   (c) Find the final shortest path matrix.

2. A city's road network is represented as a weighted directed graph, where each vertex corresponds to a location, and each edge represents a one-way road with a given travel time in minutes. The adjacency matrix below represents the road network, where infinity means there is no direct road between two locations.

$$D_0 = \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix}$$

   (a) Draw the graph corresponding to the adjacency matrix.

   (b) Find the shortest paths between all pairs of vertices.

   (c) Identify the shortest travel time from:

   - A to C
   - D to B

Question 9 (9 marks)

Devices that are capable of wireless transmission are becoming cheaper and easier to integrate into existing technology, such as vacuum cleaners and refrigerators. These devices typically communicate using a network. The network below shows devices named A–O.

Grid	1	2	3	4	5	6	7	8	9
a									
b		O		_		F		M	
c			_	_	_			N	
d		_	_	A	_	E			
e			_	_	_			L	
f		C	B	_	K	I			
g							G		
h			D						
i	H		J						

The network above has the following conditions:

- Each device is aware of its coordinates and there is a complete list of devices.
- Each square represents a 50 m × 50 m area.
- For successful connection over wireless transmission, devices must be within range: two squares vertically or horizontally and one square diagonally. For example, device A can communicate with those squares with an underscore.
- Each device can relay data between nodes within range. For example, device A can communicate with device E, which can then communicate with device F.
- Each device can return temperature values measured within its square.

a. Which device(s) cannot successfully connect over wireless transmission with any other device? 1 mark

b. Describe how the Floyd-Warshall algorithm can be used to check if all devices can communicate with any other device. 2 marks

c. Write an algorithm to compute the average temperature of all devices in the network, given all temperatures in an input list, `temperature_list`. Some devices may be defective and return –255 as their temperature value. The algorithm should **not** include these temperatures in the calculation. 6 marks

Question 19

The following pseudocode for Floyd's all-pair shortest path algorithm is incomplete.

```
Let D be a |V| × |V| array of minimum distances initialised to ∞
For each edge (u,v) Do
      D[u][v] ← w(u,v)  // the edge weight (u,v)
  For each vertex v Do
      D[v][v] ← 0
  EndFor
  For k from 1 to |V| Do
      For i from 1 to |V| Do
         For j from 1 to |V| Do
         // this section is incomplete
         EndFor
      EndFor
  EndFor
EndFor
```

Which one of the following pseudocode extracts will complete the algorithm?

A.
```
If D[i][j] > D[i][k] + D[j][k] Then
      D[i][j] ← D[i][k] + D[k][j]
EndIf
```

B.
```
If D[i][j] < D[i][k] + D[j][k] Then
      D[i][j] ← D[i][k] + D[k][j]
EndIf
```

C.
```
If D[i][j] < D[i][k] + D[k][j] Then
      D[i][j] ← D[i][k] + D[k][j]
EndIf
```

D.
```
If D[i][j] > D[i][k] + D[k][j] Then
      D[i][j] ← D[i][k] + D[k][j]
EndIf
```

DO NOT WRITE IN THIS AREA

5.4 Transitive Closure and the Warshall Algorithm

Transitive Closure

The transitive closure of a directed graph $G = (V, A)$ is a graph G^* in which there is an edge from vertex u to vertex v if and only if there is a **path** from u to v in G. In other words, the transitive closure identifies all reachable nodes in a graph.

Example:

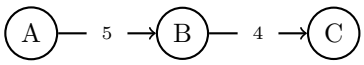


Figure 5.1: Directed graph G

	A	B	C
A	0	5	∞
B	∞	0	4
C	∞	∞	0

Figure 5.2: Adjacency Matrix of G

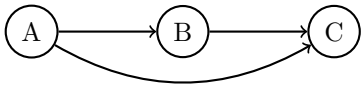


Figure 5.3: Transitive Closure of G

	A	B	C
A	1	1	1
B	0	1	1
C	0	0	1

Figure 5.4: Transitive Closure Matrix

Warshall Algorithm

What does it do?

The Warshall algorithm is a modified version of the Floyd-Warshall algorithm that computes the transitive closure of a directed graph. It uses Boolean operations (logical AND/OR) instead of numerical additions and comparisons.

What are its inputs?

A directed graph represented as an adjacency matrix A. The graph can be weighted or unweighted.

What are its outputs?

The output is a transitive closure matrix T. The entry $T[i][j]$ is 1 if there is a path from vertex i to vertex j in the original graph, and 0 otherwise. The matrix can be used to create the transitive closure graph.

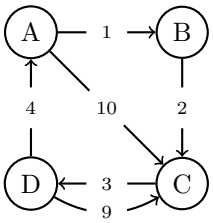
Algorithm

```
Algorithm Warshall(G(V,A))
where V is the set of vertices and A is an adjacency matrix
Initialize a Transitive Closure matrix T with the same dimensions as A

For each i in V:
    For each j in V:
        if i == j OR A[i][j] is not infinity:
            T[i][j] = 1
        else:
            T[i][j] = 0

For each k in V:
    For each i in V:
        For each j in V:
            T[i][j] = T[i][j] OR (T[i][k] AND T[k][j])

Return T
```

What are its limitations?

- The Warshall algorithm has a time complexity of $O(V^3)$, which makes it impractical for very large graphs.
- It requires an adjacency matrix representation, leading to a space complexity of $O(V^2)$.
- For sparse graphs, alternative methods like depth-first search (DFS) or breadth-first search (BFS) can compute reachability more efficiently.

Example Use the Warshall algorithm to find the transitive closure of the following graph.



The adjacency matrix for the graph is

A^0	A	B	C	D
A	0	1	10	∞
B	∞	0	2	∞
C	∞	∞	0	3
D	4	∞	9	0

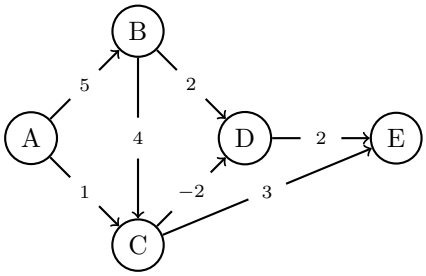
For the first iteration k = A
The matrix T is T^A

T^A	A	B	C	D
A				
B				
C				
D				

now iterate through i and j for

5.5 Exercise

1. Use the Warshall algorithm to find the transitive closure of the following graph.



2. Given the transitive closure matrix of a directed graph with 4 vertices:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

   (a) determine the possible original adjacency matrix of the graph **before** transitive closure was applied.

   (b) Draw the directed graph that corresponds to your adjacency matrix.

   (c) Explain why your solution is not necessarily unique and describe another possible original graph.

   (d) If a direct edge $(A \rightarrow D)$ was present in the original graph, how would the transitive closure change?

3. VCAA 2015 Q7

   To show that (u, v) is in the transitive closure of graph G, it is necessary to show that

   A. a u-v path exists in G.

   B. no u-v path exists in G.

   C. G is a connected graph.

   D. u and v have the same degree.

Question 4 (4 marks)

Consider an unweighted graph *G*.

a. Write a definition for the transitive closure of *G*. 2 marks

b. The Floyd-Warshall algorithm for transitive closure could be used to find the transitive closure of *G*. If the graph is assigned uniform edge weights of one unit, Floyd's algorithm for the all-pair shortest path problem could also be used to find the transitive closure of *G*.

Outline an alternative approach to finding the transitive closure of *G* that uses neither of the algorithms above. 2 marks

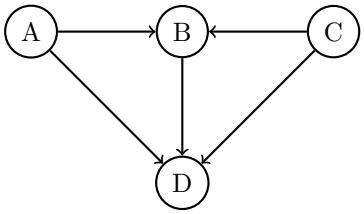
PageRank Algorithm

The PageRank score of a page represents the steady-state probability of a random web surfer being on that page at any given time.

Imagine a web site made of 4 web pages A, B, C, and D. The website is modeled as a directed graph where each page is a node and each hyperlink is an edge. The graph is shown below.

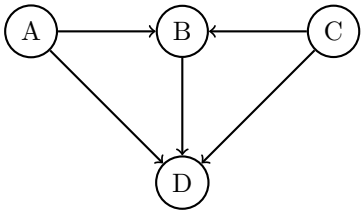
We will consider the probability of a web surfer visiting each page. Given the inital condition that the surfer is equally likely to start at any page, what is the probability that the surfer is on each page after one step?

Iteration 0: the surfer is equally likely to start at any page



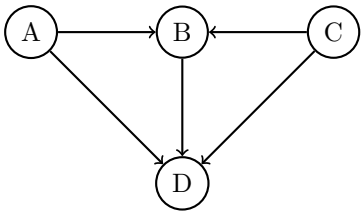
Iteration	PR(A)	PR(B)	PR(C)	PR(D)
1				

Iteration 1: the surfer is equally likely to follow any link from the current page (iteration 0)



Iteration	PR(A)	PR(B)	PR(C)	PR(D)
1				

Iteration 2: the surfer is equally likely to follow any link from the current page (iteration 1)



Iteration	PR(A)	PR(B)	PR(C)	PR(D)
2				

Convergence

If we continue this process, the probabilities will eventually converge (not change very much). This is the PageRank score for each page.

Mathematical Formulation

The PageRank of page A from our example can be calculated as follows:

$$PR(A) =$$

The PageRank value for any page u is:

$$PR(u) = \sum_{v \in B} \frac{PR(v)}{L(v)}$$

where:

- B represents the set of pages that **link to** u
- $L(v)$ is the number of links on page v

Damping Factor

The damping factor d takes into account the probability that the surfer will jump to a random page instead of following a link. At each step, the surfer either:

1. **Follows a link** from the current page with probability d.
2. **Jumps to a random page** in the network with probability $1 - d$.

The PageRank score of a node P is computed iteratively as:

$$PR(u) = \frac{1 - d}{N} + d \sum_{v \in B} \frac{PR(v)}{L(v)}$$

where:

- d is the **damping factor** (usually 0.85),
- B represents the set of pages that **link to** u
- $L(v)$ is the number of links on page v

Page Rank Algorithm

What does it do?

The PageRank algorithm is used to determine the importance of nodes in a directed graph based on their connections. Originally developed by Larry Page and Sergey Brin for ranking web pages, it assigns a numerical weight to each node, reflecting its significance based on incoming links.

What are its inputs?

- A directed graph representing nodes and edges (e.g., web pages and hyperlinks).
- A **damping factor** d (typically **0.85**).
- A convergence threshold for stopping iterations.

What are its outputs?

- A PageRank score for each node, indicating its importance in the network.
- PageRanks for all nodes sum to 1

Algorithm

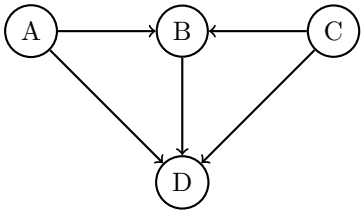
```
Algorithm PageRank(G, d, Convergence, max_iterations)
    Input:
        G: A directed graph with N nodes
        d: Damping factor (typically 0.85)
        Convergence: Convergence threshold
        max_iterations: Maximum number of iterations
    Output:
        PR: A dictionary containing PageRank values for each node

    Initialize PR, the PageRank of each node, to 1/N
    Initialize PR_new, a temporary storage for new PageRank values

    for i from 1 to max_iterations do
        for each node u in G do
            PR_new[u] ← (1 - d) / N
            for each node v pointing to u do
                PR_new[u] ← PR_new[u] + d * (PR[u] / out_degree(u))

        diff ← sum of absolute differences between PR and PR_new

        PR ← PR_new  // Update PR with new values

        if diff < Convergence then
            break  // Convergence achieved

    return PR
```

What are its limitations?

- **Does not handle disconnected graphs well**: Pages without incoming links (dangling nodes) cause rank sinks.
- **Webspam issues**: The algorithm can be manipulated with artificial link structures (e.g., link farms).

Example

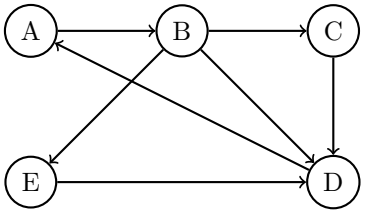
Using $d = 0.85$ find the PageRank values for the following graph.:



Iteration	PR(A)	PR(B)	PR(C)	PR(D)
0				
1				
2				

5.6 Exercise

1. Given the directed graph below,



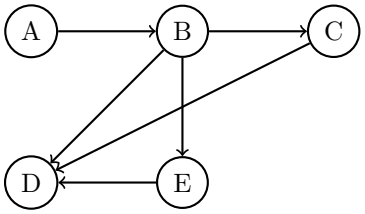
   (a) Compute the PageRank values with $d = 0.85$ until convergence at 0.01.

   (b) How many iterations were required to reach convergence?

   (c) Which node has the highest PageRank value?

   (d) How could you change the graph to increase the PageRank of node E?

2. Given the directed graph below,



   (a) Compute the PageRank values with $d = 0.85$ until convergence at 0.001.

   (b) How many iterations were required to reach convergence?

   (c) Which node has the highest PageRank value?

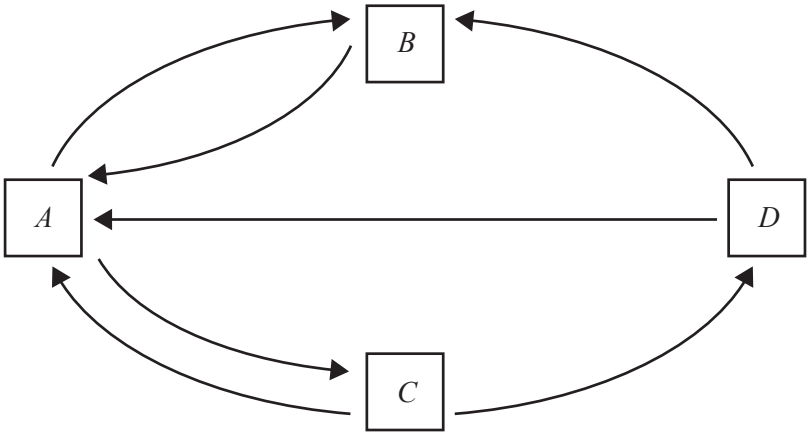
3. (a) What does the damping factor represent in the PageRank algorithm?

   (b) What would it mean if the damping factor were less than 0.5?

   (c) What impact would a very low damping factor (e.g., $d = 0.01$) have on the distribution of PageRank values across the network?

Question 6 (6 marks)

The following graph represents links between web pages.



The PageRank of Page *A* is given by

$$\mathrm{PR}(A) = \frac{(1-d)}{N} + d\left(\frac{\mathrm{PR}(B)}{\mathrm{L}(B)} + \frac{\mathrm{PR}(C)}{\mathrm{L}(C)} + \frac{\mathrm{PR}(D)}{\mathrm{L}(D)}\right)$$

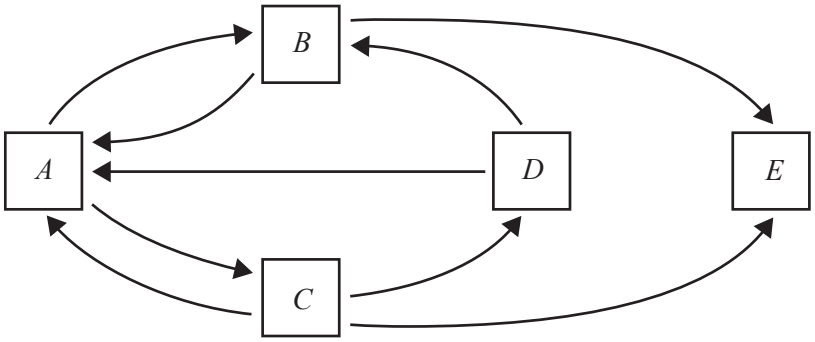
where PR(x) is the PageRank of Page x, N is the number of pages in this network and L(x) is the number of outgoing links from Page x.

a. Explain the purpose of d in the PageRank. 2 marks

b. What does $\frac{(1-d)}{N}$ represent in the PageRank? 1 mark

c. What does $d\left(\frac{\mathrm{PR}(B)}{\mathrm{L}(B)} + \frac{\mathrm{PR}(C)}{\mathrm{L}(C)} + \frac{\mathrm{PR}(D)}{\mathrm{L}(D)}\right)$ represent in the PageRank? 1 mark

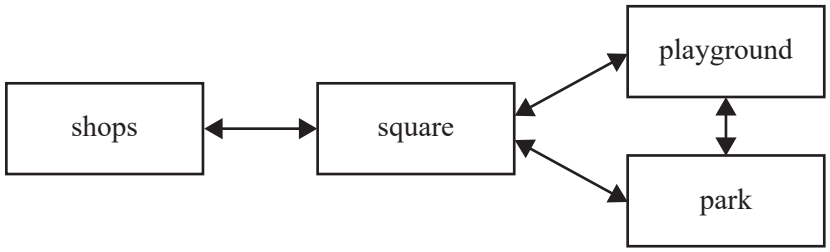
d. A new page, *E*, is added to the graph as a node, shown below.



Explain how the PageRank would include node *E* if there are no outbound links from Page *E*. 2 marks

Question 5 (7 marks)

The PageRank algorithm has been found to accurately rank urban spaces in terms of their human traffic density. Urban spaces can be modelled using graphs, with nodes representing urban spaces such as parks, squares, shops and playgrounds, and edges representing the connectivity between them. David decides to model the area around his local shops using the graph shown below.



a. Explain how the problem of estimating the importance of urban spaces based on their connectivity can be analogous to the problem of ranking the importance of web pages. 2 marks

b. What PageRank value would each node in David's graph be initialised to? 1 mark

c. Let *nodePR*(*G, r, u*) be a function with the following arguments:

- *G*, a network graph of urban spaces
- *r*, the PageRanks calculated in the previous iteration of the PageRank algorithm for each node in *G*, given as a dictionary
- *u*, a node in *G*

nodePR(*G, r, u*) returns the updated PageRank of node *u*.

Write pseudocode for an algorithm *pageranks*(*G*) that takes one input, *G*, a network graph of urban spaces. It returns a dictionary containing a (*node, PageRank*) pair for each node in the graph *G*. Your algorithm can assume that there are no nodes with zero outgoing edges, as urban spaces always have a way to leave. 4 marks

Chapter 6

Algorithm design

Area of Study 2: Algorithm design Outcome 2

Learning Intentions

- Key knowledge
  - recursion and iteration and their uses in algorithm design
  - modular design of algorithms and ADTs
- Key skills
  - identify and describe recursive, iterative, brute-force search and greedy design patterns within algorithms
  - design recursive and iterative algorithms
  - design algorithms by applying the brute-force search or greedy algorithm design pattern
  - write modular algorithms using ADTs and functional abstractions

Brute-force and Greed Algorithms

Brute-force Algorithm

Brute-force algorithms systematically check every possible solution or combination to find the optimal result. This exhaustive method ensures correctness by considering all scenarios but typically has very high computational complexity, making it inefficient for large-scale problems.

Characteristics and Suitability:

- Guarantees finding the optimal solution by exhaustive search.
- Easy to implement due to straightforward logic.
- Highly computationally intensive and impractical for large inputs.
- Suitable when computational resources are adequate and correctness is critical.

Example Applications:

- Password cracking through exhaustive search.
- Small-scale combinational optimization.
- Situations where computational cost is secondary to correctness.

Greedy Algorithm

Greedy algorithms build solutions by making the locally optimal choice at each step without reconsidering previous decisions. This approach significantly improves computational efficiency, though it does not guarantee global optimality unless specific conditions, such as optimal substructure and greedy-choice properties, are met.

Characteristics:

- Makes locally optimal decisions without revisiting past choices.
- Generally efficient with relatively low computational complexity.
- Does not always produce globally optimal solutions.

Suitability:

- Appropriate for problems exhibiting optimal substructure and greedy-choice properties.
- Effective in situations requiring fast, near-optimal solutions rather than exact solutions.

Example Applications:

- Scheduling tasks to optimize resource usage.
- Making change using the fewest coins possible.
- Graph algorithms such as Prim's algorithm (minimum spanning tree) and Dijkstra's algorithm (shortest path), under appropriate conditions.

In summary, brute-force algorithms guarantee optimal solutions at the expense of significant computational resources, whereas greedy algorithms efficiently produce quick solutions but may compromise on global optimality.

Example: Bob's To-Do List Problem

Bob has a list of tasks he wishes to complete today. Each task requires a different amount of time, and Bob wants to complete as many tasks as possible within his available 8-hour day. Here are Bob's tasks with their durations:

Task	Duration (hours)
Write Report	3
Clean Office	1
Update Software	4
Email Clients	2
Review Documents	2
Lunch Meeting	1

Bob must choose tasks to maximize the number completed within the 8-hour limit.

Brute-force Algorithm

A brute-force algorithm checks all possible combinations of tasks to find the maximum number Bob can complete:

```
max_tasks = 0
best_combination = []
for every possible subset of tasks:
    if total_duration(subset) <= 8 and len(subset) > max_tasks:
        update max_tasks and best_combination
return best_combination
```

Characteristics:

- Checks every subset of tasks.
- Always guarantees an optimal solution.
- Inefficient for a large number of tasks due to exponential complexity.

Greedy Algorithm

A greedy algorithm selects tasks based on a specific criterion (in this case, shortest duration first):

```
sort tasks by ascending duration
total_time = 0
selected_tasks = []
for each task in sorted_tasks:
    if total_time + task.duration <= 8:
        add task to selected_tasks
        total_time += task.duration
return selected_tasks
```

Characteristics:

- Selects shortest tasks first, aiming to maximize quantity completed quickly.
- Efficient and straightforward implementation.
- Usually, but not always, optimal.

Conclusion: The brute-force approach would need to examine every possible subset of tasks to guarantee finding the optimal solution. Since Bob has 6 tasks, there are $2^6 = 64$ possible subsets, including the empty set (no tasks selected).

In this example, both algorithms give the same optimal result. However, brute-force explicitly verifies optimality, whereas the greedy algorithm quickly provides a solution without such exhaustive verification. The greedy method is significantly more efficient and preferable when optimality is not strictly critical, especially for larger datasets.

Recusion and Iteration

Recursion and iteration are methods of repeating a set of instructions.

Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem.

Iteration is a method where a set of instructions is repeated in a sequence a specified number of times or until a condition is met. Both methods are used in algorithm design to solve computational problems.

Recursion

Recursion is a technique in which a function calls itself to solve smaller instances of the same problem until it reaches a base case. The recursive approach is particularly useful in problems that exhibit self-similarity or can be broken down into smaller subproblems of the same type.

A recursive function consists of the following components:

- **Base Case:** The condition that terminates recursion.
- **Recursive Case:** The part where the function calls itself with modified parameters to reduce the problem size.

Example: Factorial Calculation The factorial of a non-negative integer n, can be calculated using the following recursive function:

Factorial Calculation

```
function FACTORIAL(n)
    if n = 0 then
        return 1
    else
        return n× FACTORIAL(n − 1)
    end if
end function
```

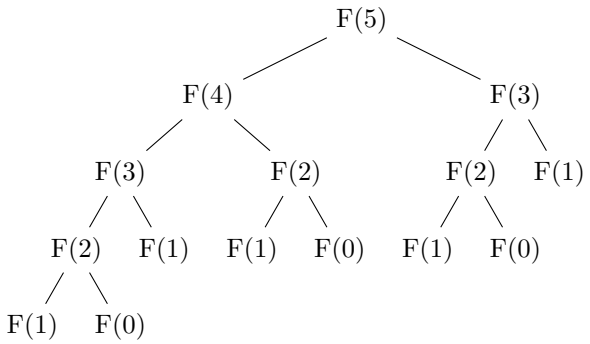
Recursive Tree Representation

Recursion can be visualized as a tree structure, where each node represents a function call and its children represent subsequent calls. The tree grows until it reaches the base case, at which point the recursion unwinds and the results are propagated back up the tree. Each recursive call creates a stack frame, where the computer must store the local variables specific to that function call and the return address, which tells the program where to continue execution once the function completes. The sequence of function calls and return addresses forms a recursion tree. This means that recursion consumes memory in proportion to the depth of the recursion tree. In a naive Fibonacci function, where each call branches into two more calls, the recursion depth grows exponentially, making it inefficient in terms of both time and space complexity.

Example: The Fibonacci sequence is defined recursively as:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n \geq 2 \end{cases}$$

The recursion tree for $F(5)$ looks like this:



Applications of Recursion

- Divide-and-conquer algorithms (e.g., Merge Sort, Quick Sort)
- Graph traversal (Depth-First Search)
- Tree-based problems (Binary Search Trees, Fibonacci sequence)

Advantages:

- Simplifies code for problems with a natural recursive structure.
- Reduces the need for explicit loops and state management.

Disadvantages:

- High memory usage due to function call stack.
- Risk of stack overflow if recursion depth is too high.

Iteration

Iteration refers to the process of executing a set of statements repeatedly using loops **(for, while)** until a specified condition is met.

Example: Iterative Factorial Calculation Instead of using function calls, iteration uses loops to compute the factorial.

Algorithm 1 Iterative Factorial Calculation

```
function FACTORIAL_ITERATIVE(n)
    result ← 1
    for i ← 1 to n do
        result ← result ×i
    end for
    return result
end function
```

Applications of Iteration

- Processing elements in arrays or lists
- Simulating repetitive tasks (e.g., clock cycles, loops in simulations)
- Iterative approximation methods (e.g., Newton's method)

Advantages:

- More efficient in terms of memory usage as it does not involve function call overhead.
- Avoids stack overflow issues inherent to deep recursion.

Disadvantages:

- Can be less intuitive for problems that naturally fit a recursive approach.
- Requires explicit loop control and state management.

When to Use Recursion vs. Iteration

- Use recursion when dealing with problems that naturally decompose into smaller subproblems (e.g., tree traversal, divide-and-conquer algorithms).
- Use iteration when performing simple, repetitive computations that do not require maintaining a call stack.
- Convert recursion to iteration when performance and memory efficiency are crucial.

6.1 Exercise

1. Write both recursive and iterative functions to calculate the Fibonacci sequence.
2. Write both recursive and iterative functions to calculate the sum of digits of a positive integer. Example: sum_digits(123) = 1 + 2 + 3 = 6.
3. Euclid's Algorithm provides an efficient way to compute the Highest Common Factor (HCF) of two numbers using the following steps:

   i If $b = 0$, then $\text{HCF}(a, b) = a$.

   ii Otherwise, replace a with b and b with $a \mod b$.

   iii Repeat step 2 until $b = 0$. The remaining value of a is the HCF.

   The modulus operator (mod) gives the remainder when one number is divided by another.

   For example:

   - $10 \mod 3 = 1$ because $10 \div 3 = 3$ remainder 1.
   - $18 \mod 5 = 3$ because $18 \div 5 = 3$ remainder 3.

   (a) Write a recursive function in Python to compute the HCF of two numbers using Euclid's Algorithm.

   (b) Write an iterative function in Python to compute the HCF of two numbers using Euclid's Algorithm.

Chapter 7

Proving Algorithm Correctness

Area of Study 2: Algorithm design Outcome 2

Learning Intentions

- Key knowledge
  - induction and contradiction as methods for demonstrating the correctness of simple iterative and recursive algorithms
- Key skills
  - explain the correctness of the specified graph algorithms
  - demonstrate the correctness of simple iterative or recursive algorithms using structured arguments that apply the methods of induction or contradiction

Common Algebraic Representations of Numbers

Type	**Algebraic Form**	**Conditions / Notes**
Even number	$n = 2a$	$a \in \mathbb{Z}$
Odd number	$n = 2a + 1$	$a \in \mathbb{Z}$
Multiple of k	$n = ka$	$a \in \mathbb{Z}$, for any integer k
Divisible by k	$n = ka$	Same as multiple: $k, a \in \mathbb{Z}$
Consecutive integers	$n, n+1, n+2, \ldots$	$n \in \mathbb{N}$
Perfect squares	$n = a^2$	$a \in \mathbb{Z}$
Perfect cubes	$n = a^3$	$a \in \mathbb{Z}$
Prime number	(No simple formula)	Assume: "Let p be prime"
Composite number	$n = ab$	$a, b \in \mathbb{N}$

Mathematical induction

Mathematical induction is a method for proving that a statement $P(n)$ is true for every natural number n. The natural numbers are the set of positive whole numbers used for counting and ordering. Zero may or may not be included in the set of natural numbers, depending on the context.

$$\mathbb{N} = \{0, 1, 2, 3, 4, 5, \ldots\}$$

That is, we want to prove that $P(1)$, $P(2)$, $P(3)$, $P(4)$, ..., are all true. Proof by induction relies on the fact that the natural numbers $\mathbb{N}$ are ordered. That is, each natural number has a unique successor and there is a smallest number (0 or 1).

Induction works by assuming that the statement is true for some arbitrary value k and then showing that it is true for the next value, i.e., $k + 1$ (the unique successor).

If we can show that the statement holds for the smallest value, then because the natural numbers are ordered, we can conclude that the statement is true for all natural numbers.

Steps:

1. **Base Case: let n=1** Verify that the statement holds for the smallest natural number (typically $n = 1$, but sometimes another value depending on the problem).
2. **Inductive Hypothesis: let n=k** Assume the statement is true for $n = k$.
3. **Inductive Step: let n=k+1** Show that if the statement is true for $n = k$, then it is also true for $n = k + 1$.

Example: Sum of First n Odd Numbers

Prove:

$$1 + 3 + 5 + 7 + \cdots + (2n - 1) = n^2, \quad \text{for all } n \in \mathbb{N}.$$

Example: Sum of Squares

Prove:

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}, \quad \text{for all } n \in \mathbb{N}.$$

7.1 Exercise

Prove the following propositions for all positive integers n.

1. $1 + 5 + 9 + 13 + \cdots + (4n - 3) = \frac{n(4n-2)}{2}$.
2. $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
3. $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$
4. $10^1 + 10^2 + 10^3 + \cdots + 10^n = \frac{10}{9}(10^n - 1)$
5. $\sum_{r=1}^{n} r(r + 1) = \frac{n(n+1)(n+2)}{3}$

Example: Exponential vs Quadratic Growth

Prove:

$$2^n > n^2 \quad \text{for } n \geq 5.$$

Example: Divisibility by 7

Prove that:

$$9^n - 2^n \text{ is divisible by } 7 \quad \text{for all } n \in \mathbb{N}.$$

7.2 Exercise

Prove the following by induction.

1. (a) $2^n \geq 1 + n$ for $n \geq 1$.

   (b) $3^n < (n + 1)!$ for $n \geq 4$.

2. Prove that $8^n - 3^n$ is divisible by 5 for all $n \in \mathbb{N}$.

3. Prove that $n^3 + 2n$ is divisible by 3 for all $n \in \mathbb{N}$.

4. Prove by induction that if p is any real number satisfying $p > -1$, then:

$$(1 + p)^n \geq 1 + np, \quad \text{for all } n \in \mathbb{N}.$$

Proof by Contradiction

Proof by contradiction is a method of mathematical proof that establishes the truth of a statement by showing that the opposite (negation) of the statement is false.

Proof by contradiction can be tricky because you often don't know what the contradiction will be until you reach it. A contradiction is a situation where two statements or facts cannot both be true at the same time. Some common contradictions include:

1. **A Logical Impossibility**

$$2 = 3, \quad 5 < 4, \quad a = b \text{ and } a \neq b$$

2. **A Value that Breaks a Definition**
   - A number is both even and odd
   - A rational number has non-simplified form
   - A prime number is divisible by another number
3. **A Conflict with Known Facts**
   - Assuming $\sqrt{2}$ is rational
   - A triangle having two right angles
4. **Too Many or Too Few Possibilities**
   - n nodes, but $n + 1$ different degrees
   - Two people but only one chair
5. **A Statement that Undoes Itself**
   - $x \neq x$
   - All numbers are even, including 3

Steps:

1. Assume the negation of the statement you want to prove.
2. Use algebra, logical reasoning or known facts to derive a contradiction.
3. Conclude that the original statement must therefore be true.

Example: Prove by contradiction that if n^2 is even, then n must also be even.

Example: Prove by contradiction that $\sqrt{2}$ is irrational (*classic*)

Example: Prove by contradiction that for all integers a and b, if a is even and b is odd, then $a^2 + b^2$ is not divisible by 4.

7.3 Exercise

Use proof by contradiction to prove each of the following statements:

1. Prove by contradiction that the difference of the squares of two consecutive odd numbers is divisible by 4.
2. Prove that for all integers a, b, c, if $a^2 + b^2 = c^2$, then at least one of a or b is even.
3. Prove that for all integers a and b, if a is even and b is odd, then 4 does not divide $a^2 + 2b^2$.
4. Prove that there is no integer x such that $x^3 - 4x^2 = 7$.
5. Prove that there is no integer solution to $x^2 = 2y^2 + 1$.
6. Prove that there do not exist integers m and n such that $15m + 25n = 1$.

Induction for Algorithm Correctness

Mathematical induction is not only used to prove number patterns — it is also used to prove that algorithms work correctly for inputs of any size.

In the context of algorithm correctness, we use induction to show that a recursive or iterative algorithm behaves as expected for all valid input sizes.

The inputs to the algorithm must be well ordered — for example, the natural numbers.

The steps required are similar to the steps used when proving a mathematical statement by induction:

1. **Base Case:** Show that the algorithm works for the smallest input - before a loop starts or at the base case of a recursive function.
2. **Inductive Hypothesis:** Assume that the algorithm works correctly for some arbitrary input size k.
3. **Inductive Step:** Show that if the algorithm works for input size k, then it also works for the next input $k + 1$.

Unlike algebraic proofs, we are not comparing a left-hand side and right-hand side. Instead, we must show that the algorithm produces the expected output for each input size. This may require using mathematics, logical reasoning, or plain English to describe the expected result.

Example: Recursive Sum Algorithm

Prove that the following recursive algorithm correctly computes the sum of the first n natural numbers.

Recursive Sum Algorithm

```
1: function SUM(n)
2:     if n == 1 then
3:         return 1
4:     else
5:         return n + sum(n − 1)
6:     end if
7: end function
```

We want to prove that:

$$\text{SUM}(n) = \frac{n(n+1)}{2}$$

Example: Iterative Exponentiation Algorithm

Prove that the following iterative algorithm correctly computes 2^n for all $n \in \mathbb{N}$.

Iterative Exponentiation Algorithm

```
1: function powerOfTwo(n)
2:     result ← 1
3:     for i from 1 to n do
4:         result ← result * 2
5:     end for
6:     return result
7: end function
```

We want to prove that after the loop completes, `result` equals 2^n.

7.4 Exercise

Use induction to prove the correctness of each of the following algorithms.

1. Prove that this algorithm returns $n!$ for all $n \geq 1$.

Factorial Algorithm

```
1: function FACTORIAL(n)
2:     if n == 1 then
3:         return 1
4:     else
5:         return n × factorial(n − 1)
6:     end if
7: end function
```

2. Prove that this algorithm returns $\frac{n(n+1)}{2}$ for $n \geq 1$.

Sum Algorithm

```
1: function SUM(n)
2:     total ← 0
3:     for i from 1 to n do
4:         total ← total + i
5:     end for
6:     return total
7: end function
```

3. **Product of first n even numbers** Prove that the following algorithm returns $2^n \cdot n!$.

Even Product Algorithm

```
1: function EVENPRODUCT(n)
2:     result ← 1
3:     for i from 1 to n do
4:         result ← result × (2 × i)
5:     end for
6:     return result
7: end function
```

4. **Sum of first n odd numbers** Prove that this algorithm returns n^2.

Odd Sum Algorithm

```
1: function ODDSUM(n)
2:     total ← 0
3:     for i from 1 to n do
4:         total ← total + (2 × i − 1)
5:     end for
6:     return total
7: end function
```

5. **Array sum** Given an array A[1...n], prove that this algorithm returns the sum of the first n elements.

Array Sum Algorithm

```
1: function arraySum(A, n)
2:     if n == 1 then
3:         return A[1]
4:     else
5:         return arraySum(A, n - 1) + A[n]
6:     end if
7: end function
```

6. **Factorial** Prove that the following algorithm correctly computes $n!$.

Factorial Algorithm

```
1: function factorial(n)
2:     result ← 1
3:     for i from 2 to n do
4:         result ← result × i
5:     end for
6:     return result
7: end function
```

7. **Sum of squares** Prove that this algorithm returns $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

Sum of Squares Algorithm

```
1: function sumSquares(n)
2:     total ← 0
3:     for i from 1 to n do
4:         total ← total + i × i
5:     end for
6:     return total
7: end function
```

Contradiction for Algorithm Correctness

Proof by contradiction is another method we can use to show that an algorithm behaves correctly — particularly useful when a direct or inductive proof is difficult or unclear.

In the context of algorithm correctness, proof by contradiction works by:

1. Assuming the algorithm *does not* produce the correct result.
2. Using logical reasoning, known properties of the problem, or assumptions about inputs/outputs to show that this assumption leads to a contradiction — something impossible, illogical, or inconsistent.
3. Concluding that the original assumption must be false, and therefore the algorithm must work correctly.

Example: Neighbor Sum

Neighbor Sum Algorithm

```
1: function SumNeighbors(n)
2:     result ← empty list
3:     for i ← 0 to n − 1 step 2 do
4:         append i + (i + 1) to result
5:     end for
6:     return result
7: end function
```

Find the output of the algorithm for $n = 8$.

Prove by contradiction that every element in the output array is **odd**.

Example:FindMax

FindMax Algorithm

```
1: function FindMax(inputs)
2:     result ← −∞                          ▷ initialised to the smallest possible integer
3:     for each n in inputs do
4:         if n > result then
5:             result ← n
6:         end if
7:     end for
8:     return result
9: end function
```

Prove by contradiction that at the end of the algorithm, `result` contains the maximum value in `inputs`.

7.5 Exercise

Use a proof by contradiction to demonstrate that each of the following algorithms behaves correctly.

1. Prove by contradiction that if the algorithm returns `False`, then all values in `inputs` are distinct.

ContainsDuplicate Algorithm

```
1: function ContainsDuplicate(inputs)
2:     seen ← empty set
3:     for each x in inputs do
4:         if x ∈ seen then
5:             return True
6:         else
7:             add x to seen
8:         end if
9:     end for
10:    return False
11: end function
```

2. Prove by contradiction that if the algorithm returns `True`, then `inputs` is sorted in increasing order.

IsSorted Algorithm

```
1: function IsSorted(inputs)
2:     for i ← 0 to length(inputs) − 2 do
3:         if inputs[i] > inputs[i + 1] then
4:             return False
5:         end if
6:     end for
7:     return True
8: end function
```

3. Prove by contradiction that `double(n)` always returns an even number.

RecursiveDouble Algorithm

```
1: function double(n)
2:     if n == 0 then
3:         return 0
4:     else
5:         return 2 + double(n − 1)
6:     end if
7: end function
```

4. Prove by contradiction that for all even n, the return value is an integer.

HalfSum Algorithm

```
1: function HalfSum(n)
2:     sum ← 0
3:     for i ← 1 to n do
4:         sum ← sum + i
5:     end for
6:     return sum ÷ 2
7: end function
```

Prim's Algorithm

Consider this version of Prim's MST algorithm.

Prim's Algorithm

```
1: function PRIM(G(V,E), start)
2:     MST is an empty graph with vertices V' and edges E'
3:     add start to V'
4:     while V' is not equal to V do
5:         find the lowest weight edge in E with one vertex in V' and one not in V'
6:         add the edge to E'
7:         add the new vertex to V'
8:     end while
9:     return MST
10: end function
```

Chapter 8

Algorithm Analysis

Area of Study 1: Formal Algorithm Analysis Outcome 1

Learning Intentions

- **Key knowledge**
  - The concept of classifying algorithms based on their time and space complexity with respect to their input.
  - Techniques for determining the time complexity of iterative algorithms.
  - The definition of Big-O notation and its application to the worst-case time complexity analysis of algorithms.
  - Examples and common features of algorithms that have time complexities of $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$ and $O(n!)$.
- **Key skills**
  - Formally analyse the time efficiency of algorithms using Big-O notation.
  - Estimate the time complexity of an algorithm by recognising features that are common to algorithms with particular time complexities.

Bibliography

Roughgarden, T. (2017) Algorithms illuminated. San Francisco: Soundlikeyourself Publishing.

Skiena, S.S. (2012) The algorithm Design Manual. London: Springer.

Introduction: Comparing Algorithms

Consider the problem of finding a name in a phone book. What is the best way to do this? What algorithm should we use?

- One option is to randomly flip to pages in the book, checking names at random until we happen upon the one we're looking for. This is called **random search**.
- Another option is to look through the book one name at a time, starting from the beginning and checking each name until we find the one we're after. This is called **linear search**.
- A third option is to open the book roughly in the middle, check the name there, and decide whether to look in the left or right half based on alphabetical order. This process is repeated, halving the remaining search space each time. This is called **binary search**.

What does "better" mean when comparing algorithms?

To evaluate and compare algorithms, we need a way to measure their performance. We typically consider two key aspects:

- **Time efficiency** — How long does the algorithm take to run?
- **Space efficiency** — How much memory does it require?

We can always get a faster computer or install more RAM — and these improvements can help speed up any program. However, when comparing algorithms, we want to evaluate the quality of the algorithm itself, not the performance of the hardware it's running on. An algorithm that runs in half the time on one machine may still be far less efficient than a better-designed algorithm that performs well on all machines.

We also care most about how algorithms perform on large datasets. Even poorly designed algorithms run quickly when the input is small — the difference in performance only becomes noticeable as the size of the input grows.

Consider the algorithms above for finding a name in a phone book. If there are only 10 names in the book, it doesn't really matter whether we use random search, linear search, or binary search — all of them will finish quickly. But if the phone book has 10 million names, the difference is dramatic:

- Linear search may take up to 10 million steps.
- Binary search would only take about 24 steps.
- Random search could take any number of steps and might never find the result.

That's why algorithm analysis focuses on how performance **scales** with input size — we want algorithms that remain efficient even when the problem grows large. Big-O notation is a formal way to describe and compare this growth.

8.1 Exercise

Read the excerpt from *The Algorithm Design Manual* by S.S. Skiena, 2nd ed., pp. 31–32.

2.1 The RAM Model of Computation

Machine-independent algorithm design depends upon a hypothetical computer called the **Random Access Machine (RAM)**. Under this model of computation, we are confronted with a computer where:

- Each simple operation ($+$, $*$, $-$, $=$, `if`, `call`) takes exactly one time step.
- Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations. It makes no sense for sort to be a single-step operation, since sorting 1,000,000 items will certainly take much longer than sorting 10 items. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.
- Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Under the RAM model, we measure run time by counting up the number of steps an algorithm takes on a given problem instance. If we assume that our RAM executes a given number of steps per second, this operation count converts naturally to the actual running time.

The RAM is a simple model of how computers perform. Perhaps it sounds too simple. After all, multiplying two numbers takes more time than adding two numbers on most processors, which violates the first assumption of the model. Fancy compiler loop unrolling and hyperthreading may well violate the second assumption. And certainly memory access times differ greatly depending on whether data sits in cache or on the disk. This makes us zero for three on the truth of our basic assumptions.

And yet, despite these complaints, the RAM proves an excellent model for understanding how an algorithm will perform on a real computer. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. We use the RAM model because it is useful in practice.

Every model has a size range over which it is useful. Take, for example, the model that the Earth is flat. You might argue that this is a bad model, since it has been fairly well established that the Earth is in fact round. But, when laying the foundation of a house, the flat Earth model is sufficiently accurate that it can be reliably used. It is so much easier to manipulate a flat-Earth model that it is inconceivable that you would try to think spherically when you don't have to.

The same situation is true with the RAM model of computation. We make an abstraction that is generally very useful. It is quite difficult to design an algorithm such that the RAM model gives you substantially misleading results. The robustness of the RAM enables us to analyze algorithms in a machine-independent way.

Take-Home Lesson: Algorithms can be understood and studied in a language and machine-independent manner.

- The **worst-case complexity** of the algorithm is the function defined by the maximum number of steps taken in any instance of size n.
- The **best-case complexity** of the algorithm is the function defined by the minimum number of steps taken in any instance of size n.
- The **average-case complexity** of the algorithm is the function defined by the average number of steps over all instances of size n.

8.2 Exercise

1. Consider a RAM implementing long multiplication of the problems given below. How many 'time steps' will it take to execute each?

$$\begin{array}{r} 33 \\ \times 72 \\ \hline \end{array} \qquad \begin{array}{r} 372 \\ \times 61 \\ \hline \end{array}$$

2. What is the best case for a 2 by 2 multiplication?

3. What is the worst case for a 2 by 2 multiplication?

4. What is the best case for a 2 by 3 multiplication?

5. What is the worst case for a 2 by 3 multiplication?

6. What is the best case of a n by n multiplication?

7. What is the worst case of a n by n multiplication?

8. What is the best case of a n by m multiplication?

9. What is the worst case of a n by m multiplication?

Worst is best

When comparing algorithms we are generally interested comparing their worst case performance over a large number of steps.

From: S.S. Skiena, The Algorithm Design Manual, 2nd ed., DOI: 10.1007/978-1-84800-070-4_2, Springer-Verlag London Limited 2008, pp. 31–32

The worst-case complexity proves to be most useful of these three measures in practice. Many people find this counterintuitive. To illustrate why, try to project what will happen if you bring n dollars into a casino to gamble. The best case, that you walk out owning the place, is possible but so unlikely that you should not even think about it. The worst case, that you lose all n dollars, is easy to calculate and distressingly likely to happen. The average case, that the typical bettor loses 87.32% of the money that he brings to the casino, is difficult to establish and its meaning subject to debate. What exactly does average mean? Stupid people lose more than smart people, so are you smarter or stupider than the average person, and by how much? Card counters at blackjack do better on average than customers who accept three or more free drinks. We avoid all these complexities and obtain a very useful result by just considering the worst case.

Big Oh: The Gist

Formal algorithm analysis uses **Big O notation** to classify algorithms based on their **worst-case time complexity**. This allows us to compare algorithms in a machine-independent way, focusing on how they scale with input size. It **suppresses constant factors** and **ignores lower-order terms**, focusing only on how the algorithm scales as the input size increases.

- An algorithm with runtime $2n^2$ is said to run in **"Big O of n squared** time", which is written as $O(n^2)$.
- An algorithm with runtime $6n^2 + 10n + 23$ also has **Big O of** n^2 time — we drop the lower order terms.
- An algorithm with runtime $6n \log_2 n + 6n$ has **Big O of** $n \log n$ time - $O(n \log_2 n)$.

We purposely ignore constant factors and lower-order terms in Big O notation because they:

1. **Depend on implementation details.**
   Factors like programming language, hardware, and compiler optimizations affect constants but not the underlying growth pattern. Big O provides a machine-independent way to compare algorithms.
2. **Do not affect long-term growth.**
   For large input sizes, the highest-order term dominates. For example, in the expression $6n^2 + 10n + 23$, the n^2 term grows much faster than the others as n increases.
3. **Don't help us compare algorithms.**
   Whether an algorithm takes $2n^2$ or $100n^2$ steps doesn't really matter compared to an algorithm that grows at 2^n steps.

Big-O classification: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$ and $O(n!)$.

8.3 Exercise

1. Graph each of the Big-O classifications on the same axes and write an inequality to express the order of increasing growth rate. Find the value of n that makes the inequality true.

   Big-O classification: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$ and $O(n!)$.

2. If an algorithm has a runtime of $3n^2 + 5n + 12$ what is its Big-O classification?
3. If an algorithm has a runtime of $7n^3 + 50n \log n + 20$ what is its Big-O classification?
4. If an algorithm has a runtime of $100n^2 + 50n + 10$ what is its Big-O classification?
5. If an algorithm has a runtime of $8n \log n + 4n + 60$ what is its Big-O classification?
6. If an algorithm has a runtime of $2^{n+1} + 100n^2$ what is its Big-O classification?
7. If an algorithm has a runtime of $1000 \log n + 4$ what is its Big-O classification?
8. If an algorithm has a runtime of $n^5 + n^3 + n$ what is its Big-O classification?
9. If an algorithm has a runtime of $40n + 0.5n^2$ what is its Big-O classification?
10. If an algorithm has a runtime of $3n! + 2^n$ what is its Big-O classification?
11. If an algorithm has a runtime of $n^3 + 20 \log n + 500$ what is its Big-O classification?
12. What, in big-O notation, is the running time of this algorithm?

Searching One Array

```
1: function SearchingOneArray(A list of n integers, t integer to find )
2:     for i ← 1 to n do
3:         if A[i] = t then
4:             return TRUE
5:         end if
6:     end for
7:     return FALSE
8: end function
```

13. What, in big-O notation, is the running time of this longer algorithm?

Searching Two Arrays

```
1: function SearchingTwoArrays(A and B lists of n integers, t integer to find )
2:     for i ← 1 to n do
3:         if A[i] = t then
4:             return TRUE
5:         end if
6:     end for
7:     for i ← 1 to n do
8:         if B[i] = t then
9:             return TRUE
10:        end if
11:    end for
12:    return FALSE
13: end function
```

14. What, in big-O notation, is the running time of this algorithm?

Checking for a Common Element

```
1: function CheckCommonElement(A, B: arrays of n integers)
2:     for i ← 1 to n do
3:         for j ← 1 to n do
4:             if A[i] = B[j] then
5:                 return TRUE
6:             end if
7:         end for
8:     end for
9:     return FALSE
10: end function
```

15. What, in big-O notation, is the running time of this algorithm?

Checking for Duplicates

```
1: function CheckDuplicates(A: array of n integers)
2:     for i ← 1 to n do
3:         for j ← i + 1 to n do
4:             if A[i] = A[j] then
5:                 return TRUE
6:             end if
7:         end for
8:     end for
9:     return FALSE
10: end function
```

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

The reason why we are content with coarse Big Oh analysis is provided by Figure 2.4, which shows the growth rate of several common time analysis functions. In particular, it shows how long algorithms that use $f(n)$ operations take to run on a fast computer, where each operation takes one nanosecond (10^{-9} seconds). The following conclusions can be drawn from this table:

- All such algorithms take roughly the same time for $n = 10$.
- Any algorithm with $n!$ running time becomes useless for $n \geq 20$.
- Algorithms whose running time is 2^n have a greater operating range, but become impractical for $n > 40$.
- Quadratic-time algorithms whose running time is n^2 remain usable up to about $n = 10,000$, but quickly deteriorate with larger inputs. They are likely to be hopeless for $n >$ 1,000,000.
- Linear-time and $n \lg n$ algorithms remain practical on inputs of one billion items.
- An $O(\lg n)$ algorithm hardly breaks a sweat for any imaginable value of n.

The bottom line is that even ignoring constant factors, we get an excellent idea of whether a given algorithm is appropriate for a problem of a given size. An algorithm whose running time is $f(n) = n^3$ seconds will beat one whose running time is $g(n) = 1{,}000{,}000 \cdot n^2$ seconds only when $n <$ 1,000,000. Such enormous differences in constant factors between algorithms occur far less frequently in practice than large problems do.

Big Oh: Formal Definitions

Big-O

Big-O notation concerns functions $T(n)$ defined on natural numbers where $T(n)$ is the worst case runtime of an algorithm. Big-O describes an **upper bound** on the growth rate of the worst-case runtime of an algorithm

$T(n) = O(f(n))$ if and only if $T(n)$ is eventually bounded above by a constant multiple of $f(n)$.

Big-O Mathematical Definition

$T(n) = O(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \leq c \cdot f(n)$$

for all $n \geq n_0$.

The two constants c and n_0 quantify 'constant multiple' and 'eventually.'

Note that $O(f(n))$ is actually a **set** — it contains all functions that grow no faster than $f(n)$, up to a constant multiple, for sufficiently large n. So the expression $T(n) = O(f(n))$ really means $T(n) \in O(f(n))$, but by convention, we write it using the equals sign. This means that $n^2 = O(n^3)$ is true and so is $n = O(n^3)$ for that matter. A useful way to read this is *'n^2 has an upper bound of n^3'* or *'n^3 grows faster than n^2'*

Big-Omega

Where Big-O describes an upper bound on the growth rate of the worst-case runtime of an algorithm, Big-Omega describes a **lower bound**.

Big-Ω Mathematical Definition

$T(n) = \Omega(f(n))$ if and only if there exist positive constants c and n_0 such that

$$T(n) \geq c \cdot f(n)$$

for all $n \leq n_0$.

So $n^4 = \Omega(n^3)$ and $n^3 = \Omega(n^2)$ are true statements, as is $n^2 = \Omega(n)$. A useful way to read this is *'n^4 has a lower bound of n^3'* or *'n^3 grows slower than n^4'*.

Big-Theta

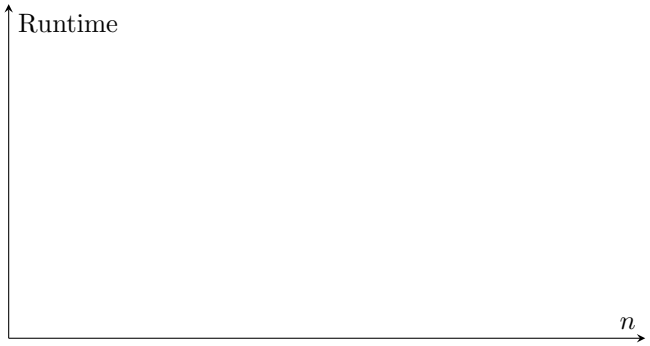
Big-Theta notation describes a **tight bound** on the growth rate of an algorithm's runtime. That is, it gives both an upper and a lower bound — the algorithm grows at the same rate as the function $f(n)$, up to constant factors.

Big-Θ Mathematical Definition

$T(n) = \Theta(f(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

for all $n \geq n_0$.



This means that $T(n)$ grows at the same rate as $f(n)$, asymptotically. The constants c_1 and c_2 define a range within which $T(n)$ stays for all sufficiently large n.

So while $n^2 = O(n^3)$ and $n^2 = \Omega(n)$, only $n^2 = \Theta(n^2)$.

This reflects the fact that:

- $O(n^3)$ is an upper bound — and n^2 grows no faster than n^3
- $\Omega(n)$ is a lower bound — and n^2 grows at least as fast as n
- $\Theta(n^2)$ is a tight bound — matching both the upper and lower growth rates of n^2

8.4 Excercise

1. VCAA Question 12 2020

   Which one of the following statements is false?

   A. The function $2n^3 + n^2 + 5$ is $O(n^4)$.

   B. The function $2n^3 + n^2 + 5$ is $O(n^3)$.

   C. The function $2n^3 + n^2 + 5$ is $\Omega(n^4)$.

   D. The function $2n^3 + n^2 + 5$ is $\Omega(n^3)$.

2. Show that $2^{n+1} = O(2^n)$

3. Let $T(n) = \frac{1}{2}n^2 + 3n$. Which of the following statements are true? (There might be more than one correct answer.)

   (a) $T(n) = O(n)$

   (b) $T(n) = \Omega(n)$

   (c) $T(n) = O(n^2)$

   (d) $T(n) = O(n^3)$

4. Let f and g be non-decreasing real-valued functions defined on the positive integers, with $f(n) \geq 1$ and $g(n) \geq 1$ for all $n \geq 1$. Assume that $f(n) = O(g(n))$, and let $c > 0$ be a constant. Is the following true?

   $$f(n) \cdot \log_2(f(n)^c) = O(g(n) \cdot \log_2(g(n)))?$$

   (a) Yes, for all such f, g, and c

   (b) Never, no matter what f, g, and c are

   (c) Sometimes yes, sometimes no, depending on the constant c

   (d) Sometimes yes, sometimes no, depending on the functions f and g

5. Assume two positive non-decreasing functions f and g such that $f(n) = O(g(n))$. Is the following true?

   $$2^{f(n)} = O(2^{g(n)})?$$

   (Multiple answers may be correct; choose all that apply.)

   (a) Yes, for all such f and g

   (b) Never, no matter what f and g are

   (c) Sometimes yes, sometimes no, depending on the functions f and g

   (d) Yes whenever $f(n) \leq g(n)$ for all sufficiently large n

6. Arrange the following functions in order of increasing growth rate, with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$:

   (a) $\sqrt{n}$

   (b) $10n$

   (c) $n^{1.5}$

   (d) $2^{\log_2 n}$

   (e) $n^{5/3}$

7. Arrange the following functions in order of increasing growth rate, with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$:

   (a) $n^2 \log_2 n$

   (b) 2^n

   (c) 2^{2n}

   (d) $n^{\log_2 n}$

   (e) n^2

8. Arrange the following functions in order of increasing growth rate, with $g(n)$ following $f(n)$ in your list if and only if $f(n) = O(g(n))$:

   (a) $2^{\log_2 n}$

   (b) $2^{2^{\log_2 n}}$

   (c) $n^{5/2}$

   (d) 2^{n^2}

   (e) $n^2 \log_2 n$

Common Growth Rates and Their Interpretations

- **Constant functions:** $f(n) = 1$

  These functions have no dependence on the input size n. They represent operations like adding two numbers or printing a fixed message. Even a function like $f(n) = \min(x, 100)$ behaves like a constant in the big picture.

- **Linear functions:** $f(n) = n$

  These measure the cost of looking at each item once in a dataset of size n, for example, to find the maximum, minimum, or compute an average.

- **Quadratic functions:** $f(n) = n^2$

  These arise when comparing all pairs of n items — for example, in *insertion sort* and *selection sort*.

- **Cubic functions:** $f(n) = n^3$

  These appear in algorithms that consider all triples of items or certain dynamic programming algorithms (e.g., matrix-chain multiplication).

- **Exponential functions:** $f(n) = c^n$, for some $c > 1$

  These emerge when enumerating all subsets of n items. For example, 2^n arises in brute-force solutions to NP-complete problems. These grow extremely quickly and become impractical even for moderate n.

- **Factorial functions:** $f(n) = n!$

  These represent the number of permutations of n items. Algorithms that examine all orderings — such as brute-force solutions to the Traveling Salesman Problem — can have factorial complexity.

- **Logarithmic functions:** $f(n) = \log n$

  These arise in divide-and-conquer algorithms, such as binary search. They represent the number of times you can halve n before reaching 1. Logarithmic growth is very slow compared to polynomial growth.

  ```
  Binary Search:
  Input: a sorted list or array and a target value t to find.

  Concept: At each step, binary search compares the target value to the middle
  element of the list:

  If the target is equal to the middle element, the search is successful.
  If the target is less than the middle element, search continues on the left half.
  If the target is greater than the middle element, search continues on the right half.

  This process continues, halving the search space each time, until the element is
  found or the search space is empty.
  ```

- **Superlinear functions:** $f(n) = n \log n$

  Found in efficient sorting algorithms like *Quicksort* and *Mergesort*. These grow slightly faster than linear functions, but enough to place them in a distinct complexity class.

8.5 Excercise

1. What is the Big-O classification of the following algorithms

Insertion Sort

```
for i ← 1 to n − 1 do
    j ← i
    while j > 0 and s[j] < s[j − 1] do
        Swap(s[j], s[j − 1])
        j ← j − 1
    end while
end for
```

2. What is the Big-O classification of the following algorithms

String Pattern Matching (Naive)

```
function FindMatch(pattern p, text t)
    m ← length of p
    n ← length of t
    for i ← 0 to n − m do
        j ← 0
        while j < m and t[i + j] = p[j] do
            j ← j + 1
        end while
        if j = m then
            return i                                  ▷ match found
        end if
    end for
    return −1                                         ▷ no match found
end function
```

3. What is the Big-O classification of the following algorithms

Matrix Multiplication

```
for i ← 1 to x do
    for j ← 1 to y do
        C[i][j] ← 0
        for k ← 1 to z do
            C[i][j] ← C[i][j] + A[i][k] · B[k][j]
        end for
    end for
end for
```

4. Question 12 VCAA 2023 Which one of the following families of functions has the largest Big-O complexity?

   A. $O(100n^2 + 1000000n)$

   B. $O(2n^{10})$

   C. $O(5000 \log_2 n)$

   D. $O(2000 \cdot (0.5)^n)$

5. What is the Big-O classification of the following algorithm:

Return Last Element

```
function LastElement(list)
    return list[length of list - 1]
end function
```

6. What is the Big-O classification of the following algorithm:

Count Letter

```
function CountLetter(word, letter)
    count ← 0
    for each character in word do
        if character = letter then
            count ← count + 1
        end if
    end for
    return count
end function
```

7. What is the Big-O classification of the following algorithm:

Brute Force Two Sum

```
function TwoSum(array, target)
    for i ← 0 to length(array) - 1 do
        for j ← i + 1 to length(array) - 1 do
            if array[i] + array[j] = target then
                return (i, j)
            end if
        end for
    end for
    return None
end function
```

Chapter 9

Recursion and Recurrence Relations

Area of Study 1: Formal algorithm analysis Outcome 1

Learning Intentions

- **Key knowledge**
  - recurrence relations as a method of describing the time complexity of recursive algorithms
- **Key skills**
  - Formally analyse the time efficiency of algorithms using Big-O notation.
  - Estimate the time complexity of an algorithm by recognising features that are common to algorithms with particular time complexities.

Recursion

Recursion is a method of solving problems where a function calls itself to solve smaller instances of the same problem. Each recursive function has:

- **Base case(s)**: Conditions under which the function returns a result directly, without recursion.
- **Recursive case(s)**: Rules that reduce the problem to simpler instances of itself.

Tracing Recursive Execution

Recursive calls can be visualised as a **tree**. Each call generates child calls until reaching the base case.

- For linear recursion (e.g., Factorial), the call stack forms a straight line.
- For tree recursion (e.g., Fibonacci), the number of calls grows exponentially.

Writing Recurrence Relations

A recurrence relation describes the time complexity of a recursive function based on the size of the input.

Step-by-step:

1. How many recursive calls are made per call?
2. What is the size of the input in each recursive call?
3. What extra work (outside the recursive calls) is done?

Let $T(n)$ be the total time to solve a problem of size n.

Then write:

$$T(n) = [\text{work in recursive calls}] + [\text{extra work}]$$

Estimating Time Complexity

The total time complexity of a recursive algorithm can often be estimated as:

$$\textbf{Time Complexity} \approx \text{Work per level} \times \text{Number of Nodes}$$

Common Patterns

- **Tail Recursion:** Recursive call is the last action.
- **Binary/Tree Recursion:** Multiple recursive calls per function.
- **Divide and Conquer:** Problem is split into smaller chunks, often equal halves.

Example: Factorial

Recursive: Factorial

```
1: function FACTORIAL(n)
2:     if n = 0 then
3:         return 1
4:     else
5:         return n× FACTORIAL(n − 1)
6:     end if
7: end function
```

1. Draw a recursion tree
2. Write the recurrence relation
3. Estimate the time complexity

Iterative: Factorial

```
1: function FACTORIAL(n)
2:     result ← 1
3:     for i ← 1 to n do
4:         result ← result × i
5:     end for
6:     return result
7: end function
```

Example: Compute the nth Fibonacci Number

Recursive: Fibonacci number

```
1: function FIB(n)
2:     if n = 0 or n = 1 then
3:         return 1
4:     else
5:         return FIB(n − 1) + FIB(n − 2)
6:     end if
7: end function
```

Iterative: Fibonacci number

```
1: function FIBONACCI(n)
2:     if n = 0 or n = 1 then
3:         return 1
4:     end if
5:     a ← 1
6:     b ← 1
7:     for i ← 2 to n do
8:         temp ← a + b
9:         a ← b
10:        b ← temp
11:    end for
12:    return b
13: end function
```

1. Draw a recursion tree
2. Write the recurrence relation
3. Estimate the time complexity

Example: Binary Search

Recursive: Binary Search

```
1: function BinarySearch(array, target, low, high)
2:     if low > high then
3:         return False
4:     end if
5:     mid ← ⌊(low + high)/2⌋
6:     if array[mid] = target then
7:         return True
8:     else if target < array[mid] then
9:         return BinarySearch(array, target, low, mid - 1)
10:    else
11:        return BinarySearch(array, target, mid + 1, high)
12:    end if
13: end function
```

1. Draw a recursion tree
2. Write the recurrence relation
3. Estimate the time complexity

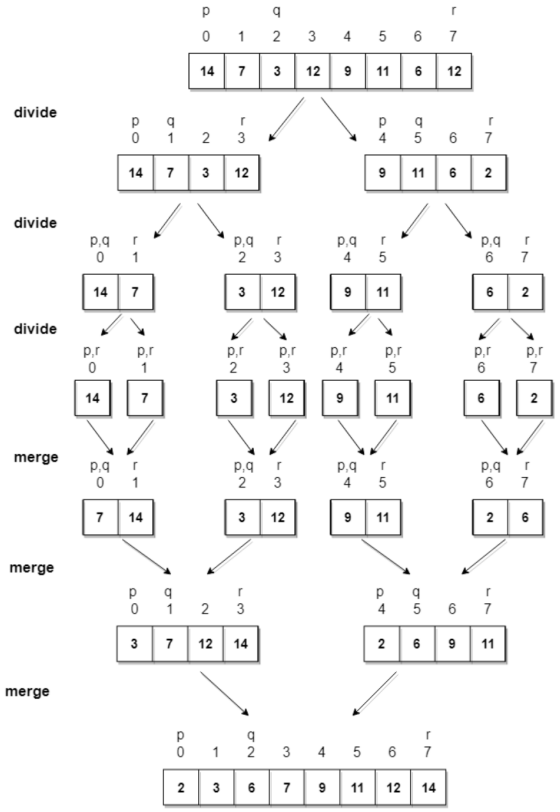
Example: Merge Sort

Recursive: Merge Sort

```
1: function MergeSort(array)
2:     if length(array) ≤ 1 then
3:         return array
4:     end if
5:     mid ← length(array) ÷ 2
6:     left ← MergeSort(array[0 : mid])
7:     right ← MergeSort(array[mid : end])
8:     return Merge(left, right)
9: end function
```

```
1: function Merge(left, right)
2:     result ← empty list
3:     while left and right are not empty do
4:         if left[0] ≤ right[0] then
5:             move left[0] to result
6:         else
7:             move right[0] to result
8:         end if
9:     end while
10:    append remaining left and right to result
11:    return result
12: end function
```

1. Draw a recursion tree
2. Write the recurrence relation
3. Estimate the time complexity



9.1 Exercise

For each of the following algorithms:

- Make sure you understand what the algorithm does
- Draw a recursion tree
- Write the recurrence relation
- Estimate the time complexity

Recursive: Mystery Function

1.
```
1: function MYSTERY(n)
2:     if n = 1 then
3:         return 1
4:     else
5:         return MYSTERY(n − 1) + n
6:     end if
7: end function
```

Recursive: Reverse an Array

2.
```
1: function REVERSE(A, left, right)
2:     if left ≥ right then
3:         return
4:     end if
5:     swap A[left] and A[right]
6:     REVERSE(A, left + 1, right - 1)
7: end function
```

Recursive: Sum of Array

3.
```
1: function SUM(A, n)
2:     if n = 0 then
3:         return 0
4:     else
5:         return A[n − 1]+ SUM(A, n - 1)
6:     end if
7: end function
```

Recursive: Power Function

4.
```
1: function POWER(x, n)
2:     if n = 0 then
3:         return 1
4:     else
5:         return x× POWER(x, n - 1)
6:     end if
7: end function
```

Recursive: Naive Power

5.
```
1: function POWER(x, n)
2:     if n = 0 then
3:         return 1
4:     else
5:         return x× POWER(x, n - 1)
6:     end if
7: end function
```

Recursive: Fast Power

6.
```
1: function POWER(x, n)
2:     if n = 0 then
3:         return 1
4:     else if n = 1 then
5:         return x
6:     else if n is even then
7:         return POWER(x × x, n/2)
8:     else
9:         return x× POWER(x × x, (n − 1)/2)
10:    end if
11: end function
```

Recursive: Maximum Element

7.
```
1: function MAX(A, n)
2:     if n = 1 then
3:         return A[0]
4:     else
5:         return max(A[n − 1], MAX(A, A[n − 1]))
6:     end if
7: end function
```

8. This algorithm counts the number of binary strings of length n that do not contain two consecutive 1s.

   E.g. for an input of n = 3, the valid strings are: 000, 001, 010, 100, 101. The algorithm returns 5.

Recursive: Count Binary Strings

```
1: function COUNTSTRINGS(n)
2:     if n = 0 then
3:         return 1
4:     else if n = 1 then
5:         return 2
6:     else
7:         return COUNTSTRINGS(n - 1) + COUNTSTRINGS(n - 2)
8:     end if
9: end function
```

Big-O Notation and Bases

Takeaway

- For logarithmic functions, the base does not matter in Big-O notation.
- For exponential functions, the base does matter in Big-O notation.

Logarithmic Functions

In Big-O notation, logarithmic bases are interchangeable. This is because any logarithm can be converted to another base using the change-of-base formula:

$$\log_b n = \frac{\log_k n}{\log_k b}$$

For example:

$$\log_3 n = \frac{\log_2 n}{\log_2 3} \approx 0.63 \times \log_2 n$$

So:

$$\log_3 n < c \cdot \log_2 n \quad \text{for some constant } c > 1$$

Since Big-O notation ignores constant factors:

$$O(\log_3 n) = O(\log_2 n) = O(\log n)$$

Therefore, for logarithmic time complexity, the base can be omitted.

Exponential Functions

For exponential functions, the base does matter. Different bases result in different growth rates. For example:

$$3^n < c \cdot 2^n \quad \text{is false for all constants } c$$

The change-of-base formula for exponentials is:

$$a^n = b^{n \cdot \log_b a}$$

So:

$$3^n = 2^{n \cdot \log_2 3} \approx 2^{1.58n}$$

$$2^{1.58n} = (2^n)^{1.58} \text{ not a constant multiplier}$$

This shows that:

$$3^n = \Theta(2^{1.58496n})$$

Unlike logarithms, exponential functions with different bases cannot be simplified into the same Big-O class. For example:

$$O(2^n) \neq O(3^n)$$

Therefore, for exponential time complexity, the base should always be specified.

Chapter 10

Solving Recurrence Relations

Area of Study 1: Formal algorithm analysis Outcome 1

Learning Intentions

- **Key knowledge**
  - recurrence relations as a method of describing the time complexity of recursive algorithms
  - the Master Theorem for solving recurrence relations of the form:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + kn^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases} \quad \text{where } a > 0,\ b > 1,\ c \geq 0,\ d \geq 0,\ k > 0$$

$$\text{and its solution } T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O\left(n^{\log_b a}\right) & \text{if } a > b^c \end{cases}$$

- **Key skills**
  - Formally analyse the time efficiency of algorithms using Big-O notation.
  - read off a recurrence relation for the running time of a recursive algorithm that can be solved by the Master Theorem or takes the form:

$$T(n) = \sum_{i=1}^{k} T(n - a_i) + b, \quad \text{where } a_i \in \mathbb{N}$$

  - use the stated Master Theorem to solve recurrence relations

The Master Theorem

The Master Theorem is a method used to solve recurrence relations for divide-and-conquer algorithms. It applies to algorithms that:

- Break a problem of size n into a subproblems,
- Each subproblem has size $\frac{n}{b}$,
- additional work is done to divide the problem and combine the results.

The 'Master Theorem' is provided during the exam in the form:

'Master Theorem'

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + kn^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases} \quad \text{where } a > 0,\ b > 1,\ c \geq 0,\ d \geq 0,\ k > 0$$

$$\text{and its solution } T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O\left(n^{\log_b a}\right) & \text{if } a > b^c \end{cases}$$

The definition of the parameters is not provided in the exam, but is as follows:

- a the number of recursive calls
- b the factor by which the input size shrinks
- kn^c is the cost of dividing and combining (non-recursive work done at each level)
- d is a constant time for the base case

Example: Merge Sort

Merge Sort is a classic example of a divide-and-conquer algorithm that can be analysed using the Master Theorem.

It works as follows:

- Divide the input array into two halves.
- Recursively sort each half.
- Merge the sorted halves into a single sorted array.

Use the Master Theorem to analyse the time complexity.

Example: BinarySearch

Binary Search is another example of a divide-and-conquer algorithm that can be analysed using the Master Theorem. It works as follows:

- Compare the target value to the middle element of the array.
- If the target is equal to the middle element, return its index.
- If the target is less than the middle element, search in the left half of the array.
- If the target is greater than the middle element, search in the right half of the array.
- Repeat the process until the target is found or the subarray size becomes zero.

Use the Master Theorem to analyse the time complexity.

Example: Recursive Integer Multiplication

This divide-and-conquer algorithm multiplies two n-digit numbers by:

- Splitting each number into high and low halves,
- Performing 4 recursive multiplications:

$$A = X_H \cdot Y_H, \quad B = X_L \cdot Y_L, \quad C = X_H \cdot Y_L, \quad D = X_L \cdot Y_H$$

- Combining the results using:

$$XY = A \cdot 10^n + (C + D) \cdot 10^{n/2} + B$$

The recurrence relation for the time complexity is:

$$T(n) \leq 4 \cdot T\left(\frac{n}{2}\right) + O(n)$$

The $O(n)$ term accounts for the cost of additions and digit shifts when combining partial results.

Use the Master Theorem to analyse the time complexity.

Example: Karatsuba Multiplication

Karatsuba's algorithm improves on the naive recursive method by reducing the number of recursive multiplications from 4 to 3.

- Splits each number into high and low halves,
- Recursively computes:

$$A = X_H \cdot Y_H, \quad B = X_L \cdot Y_L, \quad E = (X_H + X_L)(Y_H + Y_L)$$

- Uses the identity:

$$C + D = E - A - B$$

- Combines the result as:

$$XY = A \cdot 10^n + (E - A - B) \cdot 10^{n/2} + B$$

The recurrence relation for the time complexity is:

$$T(n) \leq 3 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Use the Master Theorem to analyse the time complexity.

10.1 Exercise

1. VCAA 2015 Q10

   A search algorithm satisfies the following recurrence relation.

   $$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + O(n), & n > 1 \\ O(1), & n = 1 \end{cases}$$

   Which one of the following is the time complexity of this search algorithm?

   A. $\Theta(n)$

   B. $\Theta(1)$

   C. $\Theta(\log n)$

   D. $\Theta(n \log n)$

2. VCAA 2020 Q13

   Consider the recurrence relation:

   $$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + n^2 & \text{if } n > 1 \\ 6 & \text{if } n = 1 \end{cases}$$

   $T(n)$ specifies a function that is:

   A. $O(n^2)$

   B. $O(n \log n)$

   C. $O(n^2 \log n)$

   D. $O(n^4)$

3. VCAA 2021 Q13

   Consider the recurrence relation:

   $$T(n) = \begin{cases} 5T\left(\frac{n}{2}\right) + 2n & \text{if } n > 1 \\ 2 & \text{if } n = 1 \end{cases}$$

   $T(n)$ describes a function that is:

   A. $O(n)$

   B. $O(n \log n)$

   C. $O(n^{\log_2 5})$

   D. $O(n^2)$

4. VCAA 2017 Q16

   What values of a, b, and c in the recurrence

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + kn^c & \text{if } n > 1 \\ 42 & \text{if } n = 1 \end{cases}$$

   would give the running time $T(n) = O(n^2 \log n)$?

   A. $a = 9, \quad b = 3, \quad c = 2$

   B. $a = 9, \quad b = 2, \quad c = 3$

   C. $a = 27, \quad b = 3, \quad c = 2$

   D. $a = 27, \quad b = 2, \quad c = 3$

5. VCAA 2018 Q4 Consider the following recurrence relations.

$$S(n) = \begin{cases} 2S\left(\frac{n}{3}\right) + 3\sqrt{n} & \text{if } n > 3 \\ O(1) & \text{if } n < 4 \end{cases}$$

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n) & \text{if } n > 1 \\ O(1) & \text{if } n < 2 \end{cases}$$

   a. What is the Big-O solution to $S(n)$? *1 mark*

   b. What is the Big-O solution to $T(n)$? *1 mark*

Additive Recurrence Relations

Many recursive algorithms do not divide the problem into fractions (as in divide-and-conquer), but instead reduce the input size by a fixed amount in each recursive call. These algorithms often lead to recurrence relations of the form:

$$T(n) = \sum_{i=1}^{k} T(n - a_i) + b, \quad \text{where } a_i \in \mathbb{N}$$

This expands to:

$$T(n) = T(n - a_1) + T(n - a_2) + \cdots + T(n - a_k) + b$$

Where:

- k is the number of recursive calls made at each step.
- a_i is the amount by which the input size n is reduced in the i-th recursive call. Each a_i is a natural number (i.e. a positive integer).
- b represents the amount of non-recursive work done in each call — for example, comparing values, copying data, or summing results. This could be a constant (e.g. 1), or a function of n, such as $\log n$ or n.

This form is common in recursive algorithms like:

- The naive Fibonacci algorithm: $T(n) = T(n - 1) + T(n - 2) + 1$
- Recursions that branch into several smaller subproblems, each reducing n by a constant

Understanding the shape of such recurrences helps estimate time complexity, even when the Master Theorem does not apply.

Solving Additive Recurrence Relations

Strategies

1. **Recursion Tree / Expansion**
   Write out a few steps of the recurrence to see how it expands and identify a pattern.
2. **Estimate Total Work**
   Track how many recursive calls are made in total and how much work is done at each level.
3. **Guess-and-Check**
   Try a guess such as $T(n) = O(n)$, $O(n^2)$, or $O(2^n)$, and verify by substitution.

Example Solve the following additive recurrence relation:

$$T(n) = T(n-1) + 3$$

Base case: $T(0) = 2$

Example Solve the following additive recurrence relation:

$$T(n) = T(n-1) + 2$$

Base case: $T(0) = 5$

Example Solve the following additive recurrence relation:

$$T(n) = T(n-2) + 1$$

Base case: $T(0) = 1$

Example Solve the following additive recurrence relation:

$$T(n) = T(n-1) + n$$

Base case: $T(0) = 1$

Examples

1. Linear Growth: Solve the following additive recurrence relations:

$$T(n) = T(n-1) + 1$$

Expands to:

2. Triangular Growth:

$$T(n) = T(n-1) + n$$

3. Fibonacci Recurrence:

$$T(n) = T(n-1) + T(n-2) + 1$$

4. General Case:

$$T(n) = T(n-1) + T(n-3) + 1$$

This grows faster than linear but slower than Fibonacci. Without an exact closed form, estimate an upper bound by trying $T(n) = O(2^n)$ and test if it fits.

Key Ideas

- These recurrences often lead to **exponential** or **polynomial** time complexity, depending on the number and size of the reductions.
- They model problems where the algorithm explores many combinations or paths, such as:
  - Brute-force search
  - Fibonacci number generation
  - Coin change and tiling problems

Question 3 (4 marks)

The following algorithm finds the maximum element in a list.

```
Algorithm FIND_MAX(L):
     Input: L, a non-empty list of elements

     If L has only one element
          return first element of L as the maximum
     EndIf

     m = FIND_MAX(L without the first element)
     If m is greater than the first element in L
          return m
     Else
          return the first element of L
     EndIf
```

a. Write a recurrence relation and its solution to describe the worst case running time of the algorithm when the list contains n elements. 3 marks

b. Would the Big-O for the best case running time of the algorithm be smaller than the worst case running time of the algorithm? Justify your answer. 1 mark

Question 7 (8 marks)

A machine is designed to organise balls of distinct sizes from smallest to largest. Given a bag containing an unknown number of balls, possibly zero, the machine follows these steps:

1. … <a missing step here> …
2. Randomly draw two balls from the bag. Let the smaller ball be A, having diameter d_A, and the larger ball be B, having diameter d_B.
3. Divide all remaining balls in the bag into three new bags, named *Bag1*, *Bag2* and *Bag3*, using the following criteria:
   a. All balls whose diameter is less than d_A are put into *Bag1*.
   b. All balls whose diameter is between d_A and d_B are put into *Bag2*.
   c. All balls whose diameter is greater than d_B are put into *Bag3*.
4. Recursively organise the balls in *Bag1*, *Bag2* and *Bag3*.
5. Output the balls in the following sequence: the sequence of balls in *Bag1*, *A*, the sequence of balls in *Bag2*, *B,* the sequence of balls in *Bag3*.

a. Describe the base case of the recursive ball-organising process that is missing in Step 1. 2 marks

The machine has a device to read the diameter of a ball in constant time. Once the diameter of a ball is known, the machine will then be able to put it into the appropriate bag.

b. Consider the case where every time the machine is asked to organise balls, in Step 3 it creates three bags with a similar number of balls.

i. Let $S(n)$ be the time complexity for the machine to organise a bag of n balls in this case.

Explain why $S(n)$ is given by $S(n) = 3S\left(\frac{n}{3}\right) + O(n)$ for $n > 1$. 1 mark

ii. Solve $S(n)$, giving your solution in Big-O notation. 2 marks

c. Consider the case where every time the machine is asked to organise balls, in Step 3 it puts all the balls into only one bag.

i. Let $T(n)$ be the time complexity for the machine to organise a bag of n balls in this case.

Explain why $T(n)$ is given by $T(n) = T(n - 2) + O(n)$ for $n > 1$. 1 mark

ii. Deduce the worst-case time complexity of the ball-organising process in this case and state it using Big-O notation. 2 marks

Question 14 (9 marks)

Joe finds it very time-consuming to perform the multiplication of two two-dimensional numeric arrays of size $n \times n$. He asks Alex, Betty and Chloe to help him write a program to perform the multiplication.

Alex first attempts to implement the multiplication according to the following pseudocode.

```
Algorithm multiply(A, B, n)
Begin
    For i = 1 to n Do
        For j = 1 to n Do
            Product[i][j] ← 0
            For k = 1 to n Do
                Product[i][j] ← Product[i][j] + A[i][k] × B[k][j]
            EndFor
        EndFor
    EndFor
    Return Product
End
```

Assume the multiplication and addition of two numbers can be performed in O(1) time.

a. What is the time complexity of Alex's pseudocode? Justify your answer. 2 marks

The pseudocode to add A and B, two $n \times n$ numeric arrays, is given below.

```
Algorithm add(A, B, n)
Begin
    For i = 1 to n Do
        For j = 1 to n Do
            Sum[i][j] ← A[i][j] + B[i][j]
        EndFor
    EndFor
    Return Sum
End
```

Betty comes up with the following recursive method of multiplying the arrays **when *n* is 1 or *n* can be divided by 2**:

- Step 1 – When n is 1, that is $A = A[1][1]$ and $B = B[1][1]$, just multiply the two numbers together to obtain the product, that is $C[1][1] = A[1][1] \times B[1][1]$, and return C.
- Step 2 – Otherwise, do the following:

  I. Split each two-dimensional array into four smaller two-dimensional arrays of size $(n/2) \times (n/2)$. Then, the two-dimensional arrays A and B will be denoted as

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

  where $A_{1,1}$, $A_{1,2}$, $A_{2,1}$ and $A_{2,2}$ are the two-dimensional arrays of size $(n/2) \times (n/2)$ split from A, and $B_{1,1}$, $B_{1,2}$, $B_{2,1}$ and $B_{2,2}$ are those split from B.

  II. Perform the following multiplications and additions.

$$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$$
$$C_{1,2} = A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2}$$
$$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1}$$
$$C_{2,2} = A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}$$

  III. Form the resultant two-dimensional array C using the following format and return it.

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

b. Assume both the multiplication and addition of two numbers can be performed in O(1) time.

$$T(n) = \begin{cases} 8T\left(\frac{n}{2}\right) + n^2 & \text{if } n > 1 \text{ and } n \text{ is even} \\ 1 & \text{if } n = 1 \end{cases}$$

Explain why the time complexity of Betty's algorithm can be obtained using the recurrence relation above. 3 marks

c. What is the time complexity of Betty's recursive algorithm? Explain your answer. 2 marks

d. Chloe says that she knows another recursive method for the multiplication that gives the following recurrence relationship.

$$T(n) = \begin{cases} 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Is this new method faster than the previous two? Justify your answer. 2 marks

Use the following information to answer Questions 10 and 11.

Consider a country where the only coins in circulation are 1 cent, 3 cent and 4 cent coins. A citizen would like to determine the fewest coins required to pay for a chocolate bar. The following algorithm has been provided for this purpose. The last line of the algorithm, which makes three recursive calls to itself, is incomplete.

```
Input: x, the price of the chocolate bar given in cents
Algorithm minCoins(x):
    If x < 5 Then
        If x = 2 Then
            Return 2
        Else
            Return 1
    Else
        Return _____
```

Question 10

Which one of the following correctly completes the algorithm above?

A. `minimum(minCoins(x-1)+1, minCoins(x-3)+1, minCoins(x-4)+1)`

B. `minimum(minCoins(x+1)-1, minCoins(x+3)-1, minCoins(x+4)-1)`

C. `minimum(minCoins(x-1)-1, minCoins(x-1)-3, minCoins(x-1)-4)`

D. `minimum(minCoins(x-1)+1, minCoins(x-1)+3, minCoins(x-1)+4)`

Question 11

Why is it infeasible to run this algorithm on large values of x?

A. This approach is greedy and would not return an optimal value when the value of x gets large.

B. Recursive algorithms will necessarily take longer to execute than iterative algorithms.

C. A very large number of repeated recursive calls would mean the algorithm is unlikely to terminate in a reasonable amount of time.

D. The minimum function is likely to be slow, so calling it repeatedly will mean the algorithm is unlikely to terminate in a reasonable amount of time.

1. Trace the execution of the algorithm in Question 10 for an input of $x = 6$ by drawing the recusion tree.
2. How many recursive calls are made in total when the algorithm is run with input $x = 6$?
3. Write a recurrence relation that describes the number of recursive calls made by the algorithm, assuming the `min` function runs in $O(1)$ time.
4. Estimate the time complexity of the algorithm using Big-O notation.

Term 3 Warm Up Exercise

1. What do the following notations measure in the context of algorithm analysis?
   (a) Big-O notation $O(f(n))$ — What does it describe?
   (b) Big-Omega notation $\Omega(f(n))$ — What does it describe?
   (c) Big-Theta notation$\Theta(f(n))$ — What does it describe?
2. What is the best-case and worst-case time complexity of this linear search algorithm?

```
for i in a list of length n
    if i is in the list
        return true
return false
```

3. Why does $O(n^2 + n) = O(n^2)$?
4. List the following time complexities in order from fastest to slowest growth:

$$O(n^2),\ O(1),\ O(n!),\ O(n \log n),\ O(2^n),\ O(n),\ O(\log n)$$

5. Given $T(n) = T(n-2) + 1$, estimate the time complexity.
6. Given $T(n) = T(n-2) + T(n-2) + 1$, estimate the time complexity.
7. Solve using Master Theorem:
   (a) $T(n) = 3T(n/2) + n$
   (b) $T(n) = 9T\left(\frac{n}{3}\right) + n^2$
   (c) $T(n) = 3T\left(\frac{n}{4}\right) + n$
   (d) $T(n) = 5T\left(\frac{n}{2}\right) + n^2$
   (e) $T(n) = T\left(\frac{n}{2}\right) + n$
   (f) $T(n) = 7T\left(\frac{n}{2}\right) + n^2$
8. Write a recurrence relation for a recursive Fibonacci function.
9. Give an example of an algorithm that has a time complexity of each of the following:
   (a) $O(1)$
   (b) $O(n)$
   (c) $O(\log n)$
   (d) $O(n^2)$
   (e) $O(2^n)$

Chapter 11

Analysing Algorithm

Area of Study 1: Formal algorithm analysis Outcome 1

Learning Intentions

- **Key skills**
  - Formally analyse the time efficiency of algorithms using Big-O notation.

Rules of Thumb for Time Complexity

Sequential Statements Add: If operations happen one after another their time complexities add.

```
for i in range(n):
    ...

for j in range(n):
    ...
```

Time complexity: _____

Nested Loops Multiply: If one loop is inside another their time complexities multiply.

```
for i in range(n):
    for j in range(n):
        ...
```

Time complexity: _____

Conditional Statements `if / else` branches do not add complexity; analyze the best/worst-case branch:

```
for i in range(n):
    if condition:
        do_something_constant()
    else:
        do_something_linear()
```

Worst-case time complexity: _____

Best-case time complexity: _____

Function Calls

If a function has a known complexity, substitute it directly:

```
for i in range(n):
    some_function(n)
```

Time complexity if some_function is $O(n)$: _____

Time complexity if some_function is $O(\log n)$: _____

Loops That Cut Input Size When a loop reduces the input size by a constant factor each iteration, it often leads to logarithmic complexity:

```
while n > 1:
    n = n // 2       # O(log n)
```

Time complexity: _____

Recursive Algorithms

Write a recurrence relation and solve it, use the Master Theorem, or use the recursion tree method.

Time Complexity of Graph Algorithms

Graph algorithm runtimes are usually written in terms of V, E, or both, since they define the size and structure of the problem e.g. $O(V + E)$, $O(E \log V)$, $O(VE)$, or $O(V^2)$. Keeping both V and E in the expression shows how performance changes with graph density.

Space Complexity of Graph Algorithms

Space complexity is expressed in terms of V and E. It includes space used by data structures the algorithm creates for computation. Input data (like the graph itself) is usually excluded unless it is copied.

Graph questions:

1. What is the minimum number of edges in a connected graph with V vertices? _____
2. How many edges does a tree with V vertices have? _____
3. How many edges are there in a complete graph with V vertices? _____

We will analyse the time and space complexity of the following algorithms:

- Bellman-Ford Algorithm
- Floyd-Warshall Algorithm
- Depth First Search (DFS)
- Breadth First Search (BFS)
- Prim's Minimum Spanning Tree Algorithm
- Dijkstra's Algorithm

Bellman-Ford Algorithm

Bellman-Ford Algorithm

```
1: function BellmanFord(G = (V, E), S)
2:     for all v ∈ V do
3:         Distances[v] ← ∞
4:     end for
5:     Distances[S] ← 0
6:     Previous ← empty dictionary
7:     for i = 1 to |V| − 1 do
8:         for all edge (u, v) with weight w in E do
9:             if Distances[u] + w < Distances[v] then
10:                Distances[v] ← Distances[u] + w
11:                Previous[v] ← u
12:            end if
13:        end for
14:    end for
15:    for all edge (u, v) with weight w in E do
16:        if Distances[u] + w < Distances[v] then
17:            return 'Negative weight cycle detected'
18:        end if
19:    end for
20:    return Distances, Previous
21: end function
```

1. What does the algorithm do?

2. Annotate each line or block of the algorithm with its time complexity (e.g. loops, function calls, and operations).

3. Calculate the worst case time complexity of the algorithm.

4. Calculate the best case time complexity of the algorithm.

5. List the data structures used in the algorithm and their space complexities.

6. What is the space complexity of the algorithm?

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm

```
1: procedure FloydWarshall(G = (V, E))
2:     dist ← matrix of size V × V
3:     for all i ∈ V do
4:         for all j ∈ V do
5:             if i = j then
6:                 dist[i][j] ← 0
7:             else if (i, j) ∈ E then
8:                 dist[i][j] ← weight of edge (i, j)
9:             else
10:                dist[i][j] ← ∞
11:            end if
12:        end for
13:    end for
14:    for all k ∈ V do
15:        for all i ∈ V do
16:            for all j ∈ V do
17:                if dist[i][k] + dist[k][j] < dist[i][j] then
18:                    dist[i][j] ← dist[i][k] + dist[k][j]
19:                end if
20:            end for
21:        end for
22:    end for
23:    return dist
24: end procedure
```

1. What does the algorithm do?

2. Annotate each line or block of the algorithm with its time complexity (e.g. loops, function calls, and operations).

3. Calculate the worst case time complexity of the algorithm.

4. Calculate the best case time complexity of the algorithm.

5. List the data structures used in the algorithm and their space complexities.

6. What is the space complexity of the algorithm?

Graph Traversal Algorithms

Depth First Search

```
1: procedure DFS(G, start node)
2:     visited ← empty list
3:     stack ← empty stack
4:     Push start node onto stack
5:     while stack is not empty do
6:         u ← Pop stack
7:         Add u to visited
8:         for all nodes w adjacent to u do
9:             if w not in stack and w not in
   visited then
10:                Push w onto stack
11:            end if
12:        end for
13:    end while
14:    return visited
15: end procedure
```

Breadth First Search

```
1: procedure BFS(G, start node)
2:     visited ← empty list
3:     queue ← empty queue
4:     Enqueue start node into queue
5:     while queue is not empty do
6:         u ← Dequeue queue
7:         Add u to visited
8:         for all nodes w adjacent to u do
9:             if w not in queue and w not in
   visited then
10:                Enqueue w into queue
11:            end if
12:        end for
13:    end while
14:    return visited
15: end procedure
```

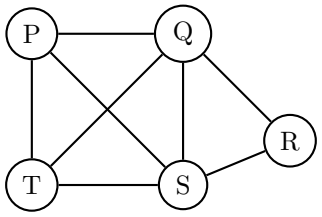


Figure 11.1: Example graph G

For the Depth First Search:

1. Trace the algorithm on G starting at P. What is the order of the nodes visited?

2. What does the algorithm do?

3. Annotate each line or block of the algorithm with its time complexity (e.g. loops, function calls, and operations).

4. What operations does the algorithm perform?

5. Calculate the worst case time complexity of the algorithm.

6. Calculate the best case time complexity of the algorithm.

7. List the data structures used in the algorithm and their space complexities.

8. What is the space complexity of the algorithm?

9. what is the time and space complexity of the Breadth First Search algorithm?

Prim's Minimum Spanning Tree Algorithm

Prim's algorithm can have different run times depending on the data structures and implementation used. In this section, we'll examine the classic version that uses an adjacency matrix, which has appeared on past exams, as well as a more efficient version using a binary heap. The binary heap data structure is not listed on the current study design.

Prim's Algorithm (Adjacency Matrix)

```
1: procedure PRIM(cost[V][V] - an adjacency matrix, start)
2:     for all v ∈ V do
3:         key[v] ← ∞
4:         parent[v] ← null
5:         inMST[v] ← false
6:     end for
7:     key[start] ← 0
8:     for i = 1 to |V| do
9:         u ← vertex not in MST with minimum key[u]
10:        inMST[u] ← true
11:        for all v ∈ V do
12:            if inMST[v] = false and cost[u][v] > 0 and cost[u][v] < key[v] then
13:                key[v] ← cost[u][v]
14:                parent[v] ← u
15:            end if
16:        end for
17:    end for
18:    return parent
19: end procedure
```

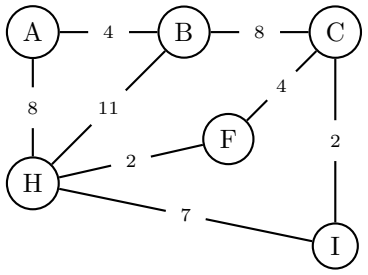


Figure 11.2: Graph G

	A	B	C	F	H	I
A	0	4	0	0	8	0
B	4	0	8	0	11	0
C	0	8	0	4	0	2
F	0	0	4	0	2	0
H	8	11	0	2	0	7
I	0	0	2	0	7	0

Figure 11.3: Adjacency matrix for graph G

1. Trace Prim's algorithm starting at A. What edges are selected for the MST?

	A	B	C	F	H	I
key						
parent						
inMST						

$u =$

2. What does the algorithm do?

3. Annotate each line or block of the algorithm with its time complexity (e.g. loops, function calls, and operations).

4. Calculate the worst case time complexity of the algorithm.

5. Calculate the best case time complexity of the algorithm.

6. List the data structures used in the algorithm and their space complexities.

7. What is the space complexity of the algorithm?

Prim's algorithm can be implemented using an adjacency list. Vertices are stored in a priority queue along with the weight of the edge connecting them to the growing MST. The priority queue is typically implemented using a binary heap, which supports insertions and key updates in $O(\log n)$ time.

Prim's Algorithm (Adjacency List, Priority Queue)

```
1:  procedure PRIM(G = (V, E), start)
2:      for all v ∈ V do
3:          key[v] ← ∞
4:          parent[v] ← null
5:          inMST[v] ← false
6:      end for
7:      key[start] ← 0
8:      pq ← priority queue ordered by key[v]
9:      Insert all v ∈ V into pq
10:     while pq is not empty do
11:         u ← extract-min from pq
12:         inMST[u] ← true
13:         for all neighbors v of u do
14:             if inMST[v] = false and weight(u, v) < key[v] then
15:                 key[v] ← weight(u, v)
16:                 parent[v] ← u
17:                 Update v's key in pq
18:             end if
19:         end for
20:     end while
21:     return parent
22: end procedure
```

The adjacency list representation of the graph previously shown is:

$V = \{A, B, C, F, H, I\}$
$E = \{(A, B, 4), (A, H, 8), (B, C, 8), (B, H, 11), (C, F, 4), (C, I, 2), (F, H, 2), (H, I, 7)\}$

1. Annotate each line or block of the algorithm with its time complexity (e.g. loops, function calls, and operations). Note that the priority queue operations are $O(\log V)$.
2. What operations does Prim's algorithm perform?

   _____

   _____

3. What is the worst-case time complexity of Prim's algorithm (adjacency list with heap)?

   _____

4. What is the best-case time complexity of Prim's algorithm?

   _____

5. What is the space complexity of Prim's algorithm?

   _____

Dijkstra's Algorithm

Like Prim's algorithm, Dijkstra's algorithm can be implemented using either an adjacency matrix or an adjacency list. The two algorithms have a similar structure and, when implemented with the same data structures, they have the same time complexity.

Dijkstra's Algorithm (Adjacency Matrix)

```
1:  procedure DIJKSTRA(cost[V][V], start)
2:      for all v ∈ V do
3:          dist[v] ← ∞
4:          parent[v] ← null
5:          visited[v] ← false
6:      end for
7:      dist[start] ← 0
8:      for i = 1 to |V| do
9:          u ← unvisited vertex with minimum dist[u]
10:         visited[u] ← true
11:         for all v ∈ V do
12:             if cost[u][v] > 0 and visited[v] = false and dist[u] + cost[u][v] < dist[v] then
13:                 dist[v] ← dist[u] + cost[u][v]
14:                 parent[v] ← u
15:             end if
16:         end for
17:     end for
18:     return dist, parent
19: end procedure
```

1. Annotate each line or block of the algorithm with its time complexity (e.g. loops, function calls, and operations). Priority queue operations are $O(\log V)$.
2. What does the algorithm compute?

   _____

   _____

3. What is the worst-case time complexity of Dijkstra's algorithm?

   _____

4. What is the best-case time complexity?

   _____

5. What data structures are used and what is the space complexity?

   _____

   _____

6. What kind of graphs is Dijkstra's algorithm suitable for?

   _____

Exercise

1. VCAA 2016 Q11
   Consider the following four algorithms, operating on a graph with V nodes and E edges:
   (a) Floyd–Warshall's algorithm for transitive closure
   (b) Bellman–Ford's algorithm for the single-source shortest path problem
   (c) Depth-first traversal algorithm
   (d) Dijkstra's algorithm for the single-source shortest path problem

   The time complexities of these algorithms, in order, are:
   A. $O(V^3),\ O(VE),\ O(V^2),\ O(V+E)$
   B. $O(V+E),\ O(V^3),\ O(VE),\ O(V^2)$
   C. $O(V^3),\ O(VE),\ O(V+E),\ O(V^2)$
   D. $O(VE),\ O(V+E),\ O(V^3),\ O(V^2)$
2. Consider the following algorithm.

Mystery Algorithm (Adjacency Matrix)

```
1: procedure MA(cost[V][V])
2:     for all u ∈ V do
3:         for all v ∈ V do
4:             if cost[u][v] = 100 then
5:                 return true
6:             end if
7:         end for
8:     end for
9:     return false
10: end procedure
```

(a) What does the algorithm do?

(b) What is the worst case time complexity of the algorithm?

(c) What is the best case time complexity of the algorithm?

(d) What is the space complexity of the algorithm?

3. The following algorithm uses a helper algorithm `Get_Neighbours(v)` which returns all the neighbours of node v.

Find_a_node (Adjacency List)

```
1: procedure FIND_A_NODE(G = (V, E), to_find)
2:     for all v in V do
3:         if v = to_find then
4:             return Get_Neighbours(v)
5:         end if
6:     end for
7:     return False
8: end procedure
```

(a) If `Get_Neighbours` runs in $O(n)$, what is the best and worst case time complexity of `Find_a_node`?

(b) If `Get_Neighbours` runs in $O(logn)$, what is the best and worst case time complexity of `Find_a_node`?

4. Consider the following algorithm.

Alg (V vertices list, E edges list)

```
1: procedure ALG(G = (V, E))
2:     for all v ∈ V do
3:         for all (u, v) ∈ E do
4:             print u and v
5:         end for
6:     end for
7:     return False
8: end procedure
```

(a) What is the time complexity of the algorithm?

(b) What is the space complexity of the algorithm?

(c) If G is a dense graph, what is the time complexity in terms of V?

(d) Write an algorithm that runs in $O(V + E)$ time.

Evaluating Time Complexity with Real Input Sizes

We often say:

> *"If a problem takes more than polynomial time, it's intractable."*

But this statement depends heavily on the size of the input.

- An algorithm with time complexity $O(2^n)$ may still be perfectly usable for small n
- An algorithm with time complexity $O(n^3)$ may be too slow when n is very large

In practice, whether an algorithm is **tractable** depends on:

- The expected input size
- Available computing resources
- Whether there is a strict time limit (e.g. real-time systems)

Time complexity (Big-O) tells us how the number of operations grows with input size. To evaluate performance:

1. Estimate the **number of operations** for a specific input.
2. Multiply by the **time per operation** (e.g. 1 microsecond).
3. Convert to seconds, minutes, or hours as needed.

Example

Suppose an algorithm has time complexity:

$$T(n) = O(n^2)$$

If $n = 5000$, then it will take roughly:

$$T(5000) = 5000^2 = 25\,000\,000 \text{ operations}$$

If each operation takes 1 microsecond:

$$25\,000\,000 \times 10^{-6} = 25\,000\,000 \ \mu\text{s} = 25 \text{ seconds}$$

So the algorithm would take about 25 seconds to run.

Exercise

1. An algorithm runs in $O(n \cdot \log_2 n)$. Complete the table assuming each operation takes 1 microsecond.

Input Size n	Estimated Operations	Time in Seconds
200		
20,000		
2,000,000		

   (a) Which input sizes can the algorithm handle in under 30 seconds?

   (b) What happens if the operation takes 10 microseconds instead of 1?

   (c) How does the time growth compare to an $O(n^2)$ algorithm?

2. Consider the following algorithm:

IndexCompareSum(A)

```
1: count ← 0
2: for i = 1 to n do
3:     for j = 1 to i do
4:         if A[i] > A[j] then
5:             count ← count + 1
6:         end if
7:     end for
8: end for
9: return count
```

   (a) What is the time complexity of this algorithm?

   (b) Estimate how long the algorithm would take to run if each basic operation takes 1 nanosecond.

Input size (n)	Estimated operations	Estimated time (in seconds)
500		
50 000		
500 000		

3. Consider the following algorithm:

```
MysteryProcess(n, e)
1: for i = 1 to n do
2:     for j = 1 to e do
3:         DoConstantWork(i, j) # O(1) work
4:     end for
5: end for
6: for k = 1 to n do
7:     DoConstantWork(k) # O(1) work
8: end for
```

   (a) What is the time complexity of this algorithm, in terms of n and e?

   (b) Estimate how long the algorithm would take to run if each basic operation takes 1 millisecond.

Input size (n)	Input size (e)	Estimated operations	Estimated time (in seconds)
50	20		
500	20		
50	2000		

   (c) What is the largest input size that can be processed in under 30 seconds?

4. An algorithm has time complexity $O(n^3)$. If it takes 2 seconds for $n = 100$, estimate how long it would take for $n = 1000$.

5. For each of the following algorithms, estimate the number of operations required for input sizes $n = 1000$, 2000, 4000, etc. How does the number of operations grow as the input size doubles? What does this reveal about the time complexity of each algorithm?

   - Algorithm A: $O(n^2)$
   - Algorithm B: $O(n \cdot \log_2 n)$

Chapter 12

Divide and Conquer

Area of Study 2: Advanced algorithm design Outcome 2

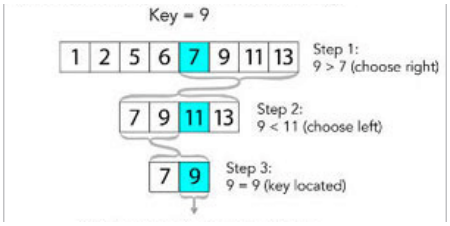
Learning Intentions

- **Key knowledge**
  - the binary search algorithm
  - divide and conquer algorithms that have linear time divide and merge steps, including mergesort and quicksort
- **Key skills**
  - Formally analyse the time efficiency of algorithms using Big-O notation.
  - Estimate the time complexity of an algorithm by recognising features that are common to algorithms with particular time complexities.

12.1 Exercise

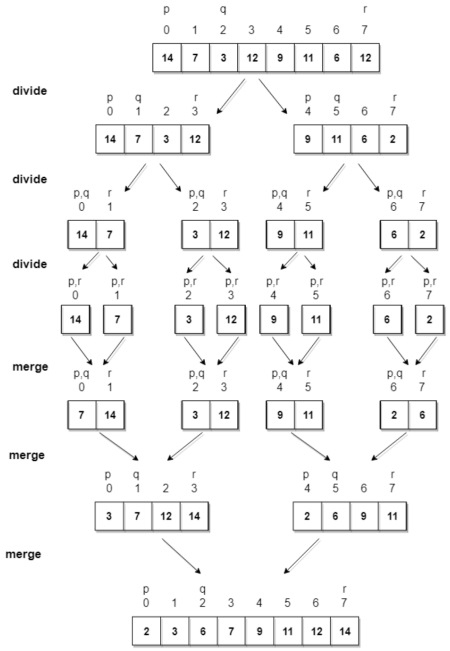
1. **Binary Search** - Write your own Python code for binary search. Write both an iterative and recursive version.

   Binary Search finds a target value in a **sorted** array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty.



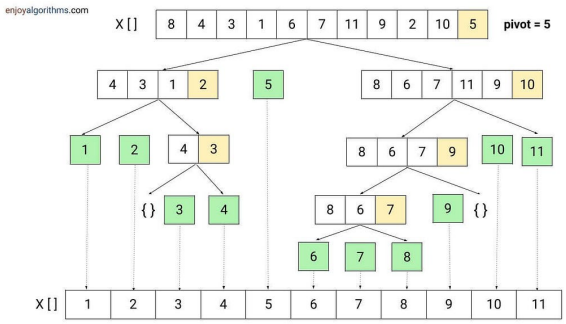
2. **Merge Sort** - Write your own Python code for merge sort. Use a **recursive** approach to implement the algorithm. Write a **helper function** to merge two sorted sub-arrays.

   Merge Sort sorts an array by repeatedly dividing it into halves, sorting each half, and then merging the sorted halves.



3. **Quick Sort** - Write your own Python code for quick sort. Use a **recursive** approach. You may choose your pivot to be the first element, last element, or a random element.

   Quick Sort sorts an array by selecting a *pivot* element (usually the last element), and then puts all the smaller elements before the pivot and all the larger elements after it. This process is repeated on each side of the pivot until the whole array is sorted.



Call Stacks

Recursive algorithms use the **call stack** to keep track of active function calls. Each call creates a **stack frame** containing parameters, local variables, and a return address. When the function finishes, its frame is removed.

The number of active calls determines the **stack depth**, which contributes to the algorithm's space complexity.

Lists

Lists are usually passed by reference, so the same list is shared across calls without increasing space usage.

If a new copy of the list is created in each call (e.g. via slicing), each call uses extra space, increasing total space complexity.

Binary Search

```
1: function BinarySearch(array, target, low, high)
2:     if low > high then
3:         return False
4:     end if
5:     mid ← ⌊(low + high)/2⌋
6:     if array[mid] = target then
7:         return True
8:     else if target < array[mid] then
9:         return BinarySearch(array, target, low, mid - 1)
10:    else
11:        return BinarySearch(array, target, mid + 1, high)
12:    end if
13: end function
```

1. Annotate each line or block of the algorithm with its time complexity (e.g. comparisons, recursive calls, and assignments).
2. What does the algorithm compute?

   _____

   _____

3. What is the worst-case time complexity of the algorithm?

   _____

4. What is the best-case time complexity?

   _____

5. Trace the algorithm on the array [1, 3, 5, 7, 9, 11, 13, 15] with a target value of 9 writing down the call stack at each step.

   _____

   _____

   _____

   _____

   _____

6. What is the space complexity of this recursive version?

   _____

   _____

Merge Sort

Recursive: Merge Sort

```
1: function MergeSort(array)
2:     if length(array) ≤ 1 then
3:         return array
4:     end if
5:     mid ← length(array) ÷ 2
6:     left ← MergeSort(array[0 : mid])
7:     right ← MergeSort(array[mid : end])
8:     return Merge(left, right)
9: end function
```

```
1: function Merge(left, right)
2:     result ← empty list
3:     i ← 0
4:     j ← 0
5:     while i < length(left) and j < length(right) do
6:         if left[i] ≤ right[j] then
7:             append left[i] to result
8:             i ← i + 1
9:         else
10:            append right[j] to result
11:            j ← j + 1
12:        end if
13:    end while
14:    append remaining elements of left to result
15:    append remaining elements of right to result
16:    return result
17: end function
```

1. What does the algorithm compute?

2. Annotate each line or block of the `Merge` algorithm with its time complexity (e.g. comparisons, assignments, and appends).

3. What is the time complexity of the `Merge` function?

4. Annotate each line or block of the `MergeSort` algorithm with its time complexity.

5. What is the worst-case time complexity of Merge Sort?

6. What is the best-case time complexity?

7. What is the space complexity of this recursive version?

Quick Sort

Quick Sort

```
1: function QuickSort(array)
2:     if length(array) ≤ 1 then
3:         return array
4:     end if
5:     pivot ← array[0]
6:     less ← empty list
7:     greater ← empty list
8:     for each x in array[1:] do
9:         if x ≤ pivot then
10:            append x to less
11:        else
12:            append x to greater
13:        end if
14:    end for
15:    return QuickSort(less) + [pivot] + QuickSort(greater)
16: end function
```

1. What does the algorithm compute?

2. Annotate each line or block of the `QuickSort` algorithm with its time complexity.

3. What is the worst-case time complexity of Quick Sort?

4. What is the best-case time complexity?

5. What is the space complexity of this recursive version?

SECTION B

Instructions for Section B

Answer **all** questions in the spaces provided.

Use the Master Theorem to solve recurrence relations of the form shown below.

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + kn^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases} \qquad \text{where } a > 0,\ b > 1,\ c \geq 0,\ d \geq 0,\ k > 0$$

$$\text{and its solution } T(n) = \begin{cases} O(n^c) & \text{if } \log_b a < c \\ O(n^c \log n) & \text{if } \log_b a = c \\ O(n^{\log_b a}) & \text{if } \log_b a > c \end{cases}$$

Question 1 (2 marks)

Describe the process of the merge step of the mergesort algorithm when sorting in ascending order.

Question 4 (4 marks)

Ada is currently studying array data structures. She comes up with the following way of comparing two numeric arrays of the same size. A numeric array is one where all of its entries are numbers.

Let A and B be two numeric arrays of size n. The array A is said to be greater than or equal to the array B, denoted as $A \geq B$, if for at least half of the values of i, the condition $A[i] \geq B[i]$ holds where $i = 1, \ldots, n$.

Ada wants to write an algorithm to determine whether A is greater than or equal to B. She has already implemented the following two algorithms:

1. an algorithm called `sortAscending` that will sort a numeric array of size n in ascending order with a worst case time complexity of $O(n^2)$
2. an algorithm called `median` that returns the median value of a numeric array with a time complexity of O(1)

Ada writes the following pseudocode to determine whether a numeric array A is greater than or equal to B, both of size n.

```
Algorithm isGreaterOrEqual(A, B, n)
Begin
  A ← sortAscending(A, n)
  B ← sortAscending(B, n)
  mA ← median(A, n)
  mB ← median(B, n)
  If mA >= mB Then
       Return true
  Else
       Return false
  EndIf
End
```

a. What is the worst case time complexity of `isGreaterOrEqual`? Explain your answer. 2 marks

b. Is Ada's pseudocode for `isGreaterOrEqual` correct? Explain your answer using an example. 2 marks

Chapter 13

Hard Limits of Computation

Area of Study 3: Computer science: past and present

Learning Intentions

- **Key knowledge**
  - the historical connections between the foundational crisis of mathematics in the early 20th century and the origin of computer science, including Hilbert and Ackermann's Entscheidungsproblem and its resolution by Church and Turing
  - characteristics of a Turing machine
  - the concept of decidability and the Halting Problem as an example of an undecidable problem
  - implications of undecidability for the limits of computation
- **Key skills**
  - explain the historical context for the emergence of computer science as a field
  - describe the general structure of a Turing machine
  - demonstrate the existence of hard limits of computability using the Halting Problem

Turing Machines

The Origins of Mechanical Logic

The Industrial Revolution transformed the world through machines: steam engines powered industry, railways linked cities, and looms automated production. As society adjusted to mechanisation, a natural question arose: **Could machines be made to think?**

Early mechanical systems, while limited in scope, revealed the possibility of embedding logic into physical devices.

In the 1830s, **Charles Babbage** designed the *Analytical Engine*, a general-purpose mechanical computer. It used punched cards like the Jacquard loom and featured a memory (the *store*) and processor (the *mill*).

Ada Lovelace, who wrote extensively about the machine, envisioned it not just calculating numbers, but manipulating *symbols*. She speculated that, with the right inputs, the machine could generate music, graphics, and language.

- Babbage — *Father of the Computer*
- Lovelace — *Mother of Programming*

Their ideas laid the intellectual foundations for Alan Turing's work in the 1930s.

The Turing Machine

In 1936, **Alan Turing** formalised the idea of a universal computing device, which came to be known as a **Turing Machine**. The Turing Machine is a *mathematical abstraction* of Babbage's Analytical Engine.

A **minimal universal model**:

- Minimal components
- Capable of performing any computation

Only the **tape is infinite** which means the Turing Machine is not bound by memory or speed limits. Everything else is finite so the concept is not relying on magic.

Key Components

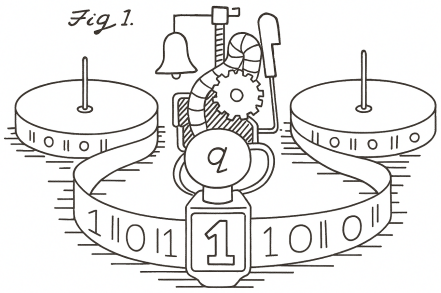


Figure 13.1: Turing Machine

- **Tape** — an infinite sequence of cells that can each hold a symbol from a finite alphabet.
- **Tape Head** — moves left or right, reads and writes symbols.
- **State Table** — a list of instructions - the *program/algorithm*.

The program is a set of states. Each state:

- Reads from the tape
- Writes to the tape
- Moves the head
- Transitions to the next state

Because it is *simple* and *universal* with *infinite memory*, anything that can be computed algorithmically can be done by a Turing machine.

- Any algorithm that can be executed can be simulated by a Turing Machine.
- If a problem cannot be solved by a Turing Machine, it cannot be solved by *any* algorithm.

This defines the boundary between the computable and the uncomputable.

Example: Incrementing a Binary Number

This Turing Machine adds one to a binary number.

- **Alphabet:** 0, 1, and blank ‘ ’
- **Input:** A binary number surrounded by blanks
- **Output:** The binary number incremented by one

Algorithm:

1. Move right to the end of the number.
2. Move left, flipping 1s to 0s.
3. When a 0 or blank is found, flip it to 1 and halt.

Transition Table:

State	Read	Write	Move	Next State
right	1	1	R	right
right	0	0	R	right
right	‘ ’	‘ ’	L	carry
carry	1	0	L	carry
carry	0	1	L	halt
carry	‘ ’	1	L	halt

Use the transition table to trace the machine’s steps on the input 10101011. Show how the tape and head change at each step.

Draw a state diagram to represent the transitions visually.

Example: Validating Input Format

This machine checks whether every `a` in a string is followed by a `b`.

- **Alphabet:** `a`, `b`, `X`, `Y`, `R`, `A`, and blank
- **Input:** A string
- **Output:** `A` (accept) or `R` (reject)

Algorithm

1. Scan for `a`, mark it as `X`.
2. Move right to find the next `b`, mark it as `Y`.
3. Return to the beginning and repeat.
4. If an `a` is unmatched, write `R`. If all matched, write `A`.

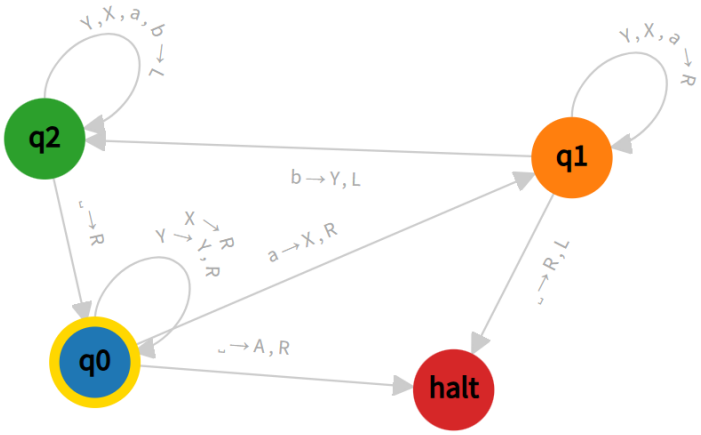


Figure 13.2: State diagram: Check input format

Write the state transition table for this Turing machine and trace its execution on the example input abaabb.

13.1 Exercise

Use https://turingmachine.io/ to test your solutions.

1. Write the state transition table for a Turing machine that adds two unary numbers. The tape will contain two unary numbers separated by a blank, like this:

   ␣111␣11␣

   The output should be the sum in unary, like this:

   ␣111111␣

2. Write the state transition table for a Turing machine that checks if a binary number is even. The tape will contain a binary number, like this:

   ␣1010␣

   The output should be a single symbol indicating even or odd, like this:

   ␣E␣ (for even) or ␣O␣ (for odd)

3. Write the state transition table for a Turing machine that checks if a string of parentheses is balanced. The tape will contain a string of parentheses, like this:

   ␣(())␣

   The output should be a single symbol indicating balanced or unbalanced, like this:

   ␣B␣ (for balanced) or ␣U␣ (for unbalanced)

4. Write the state transition table for a Turing machine that reverses a string. The tape will contain a string, like this:

   ␣hello␣

   The output should be the reversed string, like this:

   ␣olleh␣

Origins of Computer Science - A History Lesson

Vocabulary

- **Axiom** — foundational statement assumed true
- **Church–Turing thesis** — defines the boundary of what is computable
- **Completeness**
- **Computable** — can be solved by a Turing machine
- **Consistent** — no contradictions in the system
- **Decidable** — algorithm exists that always halts with yes/no
- **Entscheidungsproblem** - Ent-shy-dungs-problem - decision problem posed by Hilbert
- **Finite** — limited in size or length
- **Halting Problem** — classic undecidable problem
- **Paradox** — self-contradictory result (like Russell's Paradox or the Liar Paradox)
- **Turing machine** — model of computation
- **Undecidable** — no such algorithm exists

The Players

- **David Hilbert (1862–1943)** German mathematician. University of Göttingen in Germany
- **Wilhelm Ackermann (1896–1962)** Student and collaborator of Hilbert. University of Göttingen in Germany
- **Alonzo Church (1903–1995)** American logician. Princeton University in the USA.
- **Alan Turing (1912–1954)** British mathematician and founder of computer science. University of Cambridge (UK).

  From 1936 to 1938, Turing went to Princeton to do his PhD under Alonzo Church.

Timeline of publications

1928 Hilbert and Ackermann
Grundzüge der theoretischen Logik (*Principles of Mathematical Logic*). They formalised first-order logic and clearly defined the **Entscheidungsproblem** (decision problem): whether there exists a general mechanical procedure (algorithm) to decide if any given statement is provable.

1936 Alonzo Church
"An Unsolvable Problem of Elementary Number Theory" published. Church used **lambda calculus** to prove that the **Entscheidungsproblem** is undecidable — showing that there is no general algorithm to decide provability for all statements in first-order logic.

1936 Alan Turing
"On Computable Numbers, with an Application to the Entscheidungsproblem" published. Turing introduced the **Turing machine** as a new model of computation. He proved the **Halting Problem** is undecidable, which implies that the **Entscheidungsproblem** is undecidable too.

The Story

- In the early 20th century, mathematics faced a crisis due to logical paradoxes, such as Russell's Paradox.
- Hilbert proposed a solution: formalise all of mathematics using a set of axioms and rules. His goal was a symbolic system in which all mathematical truths could be derived mechanically — no intuition, just logic.
- Ackermann worked with Hilbert to formalise logic and pose the Entscheidungsproblem: to find a single algorithm that could decide if a mathematical statement was true or false
- In 1936, Alonzo Church proved that the Entscheidungsproblem is undecidable, using a formal model of computation called **lambda calculus**.
- Around the same time, Alan Turing independently proved the same result using his new model of computation — the **Turing machine** — by showing that the **Halting Problem** is undecidable.
- Church and Turing's work led to the formulation of the **Church–Turing Thesis**, the formal limit of what can be computed

Some details

Hilbert's Program

Hilbert's program aims to reduce all of mathematics to formal symbolic manipulations. Every line follows strictly from axioms and rules — no intuition, no diagrams, no external meanings. Just symbols and logic.

In his 1927 program to fully formalise mathematical reasoning, David Hilbert outlined the following three goals:

1. Mathematics should be **complete**: all true mathematical statements can be derived from a finite set of axioms.
2. Mathematics should be **consistent**: no contradictions.
3. Mathematics should be **decidable**: proofs can be verified by a single algorithm.

Example - not on study design - Peano and Addition Axioms

1. $0 \in \mathbb{N}$ Zero is a natural number.
2. $\forall x \in \mathbb{N},\ S(x) \in \mathbb{N}$ Every natural number has a successor.
3. $\forall x \in \mathbb{N},\ S(x) \neq 0$ Zero is not the successor of any number.
4. $\forall x, y \in \mathbb{N},\ S(x) = S(y) \Rightarrow x = y$ The successor function is injective (no two numbers share a successor).
6. $\forall x \in \mathbb{N},\ x + 0 = x$ Zero is the additive identity.
7. $\forall x, y \in \mathbb{N},\ x + S(y) = S(x + y)$ Defines addition recursively using the successor function.

The **Entscheidungsproblem** posed by Hilbert and Ackermann challenged mathmematicians to solve the third goal:

> Is there a general algorithm that can decide, for any mathematical statement, whether it is provable from the axioms?

In 1936, Alan Turing answered this question by proving that the **Halting Problem** is undecidable.

This serves as a **counterexample to Hilbert's Goal 3** — it shows that no single algorithm can decide, for every possible mathematical statement, whether it is provable. In other words, mathematics is not fully decidable.

The Halting Problem

Turing developed the concept of a **Turing machine** to demonstrate the Halting Problem. It is shown below algorithmically.

The Halting Problem is to decide whether a given program with a given input halts or loops forever. If a math statement can be written as a sequence of logical steps, then this is equivalent to a program running on a Turing machine.

Assumption: Suppose there exists an algorithm H that always solves this problem.

Algorithm 2 H(program, input)

```
if program(input) will loop forever then
    return "loops forever"
else
    return "halts"
end if
```

Algorithm 3 Paradox(program)

```
if H(program, program) == "loops forever" then
    return "halts"
else
    while True do
        do nothing                                  ▷ // loops forever
    end while
end if
```

Algorithm 4 Hello()

```
print("hello world")
```

Algorithm 5 Not_a_paradox()

```
while True do
    print("hello world")
end while
```

Evaluate `Paradox(Hello)`

Evaluate `Paradox(Not_a_paradox())`

Evaluate Paradox(Paradox)

13.2 Exercise

Question 9 VCAA 2019

The main goal of David Hilbert's 1927 program was to

A. prove that a system with a computable set of axioms could never be complete.

B. remove all paradoxes and inconsistencies from the foundations of mathematics.

C. prove that it is not possible to formalise all mathematical statements axiomatically.

D. construct a statement that can be derived from formal axiomatic rules and can be shown to be true.

Question 13 VCAA 2015

When considering computability, the Halting Problem is

A. complete when executed on a Turing machine.

B. decidable when executed on a Turing machine.

C. incomplete when executed on a Turing machine.

D. undecidable when executed on a Turing machine

Question 16 VCAA 2022

The Halting Problem was used to demonstrate that

A. we can never know whether a specific computer program will halt or not for a given input.

B. there exist some problems that cannot be solved by an algorithm.

C. we cannot know why a computer program has unexpectedly halted.

D. there exist some true mathematical statements that cannot be proved.

Question 10 (6 marks)

In his 1927 program to fully formalise mathematical reasoning, David Hilbert outlined the following three goals:

- Mathematics should be **complete**.
- Mathematics should be **consistent**.
- Mathematics should be **decidable**.

a. Define what is meant by any two of the goals above. 2 marks

1. _____

_____

_____

2. _____

_____

_____

b. Describe the Halting Problem. 2 marks

_____

_____

_____

_____

_____

_____

c. Explain why Alan Turing's formulation of the Halting Problem made Hilbert's program impossible. Include a reference to the specific goal in Hilbert's program that is contradicted by the Halting Problem. 2 marks

_____

_____

_____

_____

_____

Chapter 14

Soft Limits of Computation

Area of Study 1: Formal algorithm analysis Outcome 1

Learning Intentions

- **Key knowledge**
  - the concept of the P, NP, NP-Hard and NP-Complete time complexity classes for problems
  - consequences of combinatorial explosions and indicators for them
  - the feasibility of NP-Hard problems in real-world contexts
- **Key skills**
  - describe characteristics of problems in the P, NP, NP-Hard or NP-Complete time complexity classes, including the consequences for a problem's feasibility of it belonging to one of these classes
  - demonstrate how exponentially sized search and solution spaces impose practical limits on computability

Soft Limits of Computation

The soft limits of computation refer to the practical constraints on how useful an algorithm is for solving a problem. Just because an algorithm can solve a problem in theory doesn't mean it's practical in real-world scenarios. Take sudoku puzzles as an example.

Problem definition: Given a partially-filled 9 by 9 Sudoku grid, can you fill in the rest of the grid so that:

- Each row contains the digits 1 to 9 once
- Each column contains the digits 1 to 9 once
- Each 3x3 block contains the digits 1 to 9 once

A possible algorithm to solve sudoku would be a brute-force algorithm that tries every possible combination of numbers in the empty cells and checks if the resulting grid is valid.

Assuming there are 20 prefilled cells, let's work out how many combinations this would be.

Watch the video Computer Science's Biggest Mystery, by PurpleMind

https://youtu.be/rz1INSahE68?si=k3QiDf2cFaek8rct

P vs NP

Describe the following terms in your own words:

- **P**
- **NP**
- **NP-Hard**
- **NP-Complete**

Complete the following Venn diagram to show the relationships between the four classes of problems.

14.1 Exercise

1. Complete the Venn diagram to show the relationships between the four classes of problems.
2. Which of the following statements is true about problems in class P?
   A. They require exponential time in the worst case.
   B. They can only be solved using approximation algorithms.
   C. They can be solved in polynomial time.
   D. They cannot be verified efficiently.
3. Which of the following correctly describes an NP problem?
   A. A problem whose solution cannot be checked in polynomial time.
   B. A problem that is guaranteed to have an exact solution.
   C. A problem for which a solution can be verified in polynomial time.
   D. A problem that has no solution.
4. Which of the following is true about NP-complete problems?
   A. All NP-complete problems can be solved in polynomial time.
   B. If one NP-complete problem is solved in polynomial time, all problems in NP can be solved in polynomial time.
   C. NP-complete problems are a subset of P.
   D. NP-complete problems cannot be verified efficiently.
5. Which of the following is an example of an NP-complete problem?
   A. Finding a minimum spanning tree
   B. Solving a system of linear equations
   C. Boolean satisfiability problem (SAT)
   D. Binary search
6. Which statement best describes the relationship between NP and NP-hard problems?
   A. All NP problems are also NP-hard.
   B. NP-hard problems are always solvable in polynomial time.
   C. NP-hard problems may not be in NP.
   D. NP and NP-hard problems are disjoint sets.
7. Eliza and Mohan are discussing the classification of computational problems.

   Eliza says, "NP problems are the ones that cannot be solved efficiently, but NP-complete problems are easier because they just require checking if a given solution works."

   Mohan replies, "I thought NP-hard problems were the hardest problems in NP, and they include NP-complete problems."

   (a) Identify and explain any incorrect claims in Eliza's statement.

   (b) Identify and explain any incorrect claims in Mohan's statement.
8. Ava and Ben are working on a computational problem related to optimal scheduling.

   Ava says, "Our teacher said NP problems can be solved quickly if we try all possibilities in parallel, so that means NP problems are solvable in polynomial time with enough processors."

   Ben replies, "That must mean NP problems are just slow today, but in the future we'll be able to solve all of them quickly once we have faster computers."

   (a) Identify and explain any incorrect claims in Ava's statement.

   (b) Identify and explain any incorrect claims in Ben's statement.

Chapter 15

Advanced Algorithms

Area of Study 2: Advanced algorithm design

Learning Intentions

- **Key knowledge**
  - tree search by backtracking and its applications
  - the application of heuristics and randomised search to overcoming the soft limits of computation, including the limitations of these methods
  - hill climbing on heuristic functions, the A* algorithm and the simulated annealing algorithm
  - the graph colouring, 0-1 knapsack and travelling salesman problems and heuristic methods for solving them
- **Key skills**
  - apply the divide and conquer, dynamic programming and backtracking design patterns to design algorithms and recognise their usage within given algorithms
  - develop different algorithms for solving the same problem, using different algorithm design patterns, and compare their suitability for a particular application
  - apply heuristics methods to design algorithms to solve computationally hard problems
  - explain the application of heuristics and randomised search approaches to intractable problems, including the graph colouring, 0-1 knapsack and travelling salesman problems

Classic Problems

Now that we have considered the limits of computation, we can look at some classic problems that illustrate these limits and how advanced algorithms can help find solutions. Each of these problems has a **decision version** and an **optimisation version**. The **decision version** asks whether a solution exists that meets certain criteria, while the **optimisation version** seeks the best solution according to some metric.

Optimisation problem:

- Goal: Find the best solution according to some objective.
- Output: The actual solution and/or its optimal value.
- Nature: Search for the best among many possible solutions.

Decision problem:

- Goal: Answer yes/no — does a solution exist that meets a certain condition?
- Output: Boolean (yes or no).
- Nature: Confirm if a feasible solution exists that satisfies the constraint.

Why the distinction?

Decision problems are often easier to solve than optimisation problems because they require only a yes/no answer rather than finding the best solution.

You can transform an optimisation problem into a decision problem by introducing a threshold (or target value) for the objective function and asking whether there exists a feasible solution that meets or exceeds this threshold.

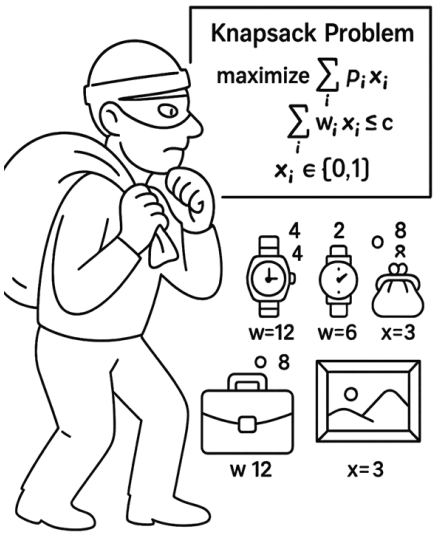
If you have an algorithm for the decision version, you can use it to solve the optimisation version by searching over possible thresholds (linear or binary search) to find the maximum (or minimum) value for which the answer is "yes". This approach can reduce the search space significantly and make the problem more manageable—particularly when the decision version can be solved in polynomial time.

Graph Colouring

- **Optimisation problem**: What is the minimum number of colours needed to colour a graph such that no two adjacent vertices share the same colour?
- **Decision problem**: Can a graph G be coloured with at most k colours so that no two adjacent vertices share a colour?
- **Complexity**:
  - Decision problem is **NP-complete** for $k \geq 3$.
  - Optimisation version is **NP-hard**.
  - Special cases (e.g., $k = 2$) solvable in polynomial time.
- **Applications**:
  - Scheduling (e.g., timetabling exams without conflicts).
  - Register allocation in compilers.
  - Frequency assignment in wireless networks.
- **Origins**:
  - Originates from the **Four Colour Problem** (1852, Francis Guthrie), asking whether four colours suffice to colour any map so that no adjacent regions share a colour.
  - Later reformulated in graph theory terms as a vertex colouring problem.
  - Four Colour Theorem proved in 1976 by Appel and Haken using computer assistance.

0–1 Knapsack

- **Optimisation problem**: Select a subset of items, each with a value and weight, to maximise total value without exceeding the weight capacity. Each item can be taken (1) or left (0).
- **Decision problem**: Given a set of items with values and weights, a maximum capacity W, and a target value V, is there a subset whose total weight $\leq W$ and total value $\geq V$?
- **Complexity**:
  - Decision version is **NP-complete**.
  - Optimisation version is **NP-hard**.
  - Solvable in $O(nW)$ via dynamic programming (pseudo-polynomial time).
- **Applications**:
  - Budget allocation.
  - Cargo loading.
  - Project selection with resource limits.
  - Investment portfolio optimisation.
- **Origins**:
  - Studied in the 19th and early 20th centuries in the context of resource allocation problems.
  - Became a classic combinatorial optimisation problem in operations research by mid-20th century.
  - Named "knapsack" because of the analogy to packing items in a backpack.



Travelling Salesman Problem (TSP)

- **Optimisation problem**: Find the shortest possible tour that visits each city exactly once and returns to the starting city.
- **Decision problem**: Given a set of cities, distances between them, and a maximum length D, is there a tour visiting each city exactly once with total length $\leq D$?
- **Complexity**:
  - Decision version is **NP-complete**.
  - Optimisation version is **NP-hard**.
  - Exact algorithms are exponential (e.g., Held–Karp DP in $O(n^2 2^n)$).
- **Applications**:
  - Route planning and logistics.
  - Manufacturing (e.g., circuit board drilling).
  - Genome sequencing.
  - Data clustering.
- **Origins**:
  - Dates back to 18th-century mathematical puzzles (e.g., Hamiltonian cycles in 1850s).
  - Named and popularised in the mid-20th century by operations research and the RAND Corporation.
  - Studied extensively as a benchmark for combinatorial optimisation and computational complexity.

0–1 Knapsack Algorithms

KnapsackBruteForce

```
1: procedure KnapsackBruteForce(W, V, C, i)
                                                          ▷ W = weights
2:                                                          ▷ V = values
3:                                                       ▷ C = capacity,
4:                                                          ▷ i = index
5:
6:     if i = length(W) then
7:         return 0
8:     end if
9:     best ← KnapsackBruteForce(W, V, C, i + 1)                ▷ Skip item i
10:    if W[i] ≤ C then
11:        includeValue ← V[i]+ KnapsackBruteForce(W, V, C − W[i], i + 1)
12:        if includeValue > best then
13:            best ← includeValue
14:        end if
15:    end if
16:    return best
17: end procedure
```

KnapsackBruteForceBitmask

```
1: procedure KnapsackBruteForceBitmask(W, V, C)
2:     n ← length(W)
3:     bestValue ← 0
4:     for mask ← 0 to 2^n − 1 do                          ▷ Loop over all subsets
5:         totalW ← 0
6:         totalV ← 0
7:         for i ← 0 to n − 1 do
8:             if mask[i] = 1 then
9:                 totalW ← totalW + W[i]
10:                totalV ← totalV + V[i]
11:            end if
12:        end for
13:        if totalW ≤ C then
14:            bestValue ← max(bestValue, totalV)
15:        end if
16:    end for
17:    return bestValue
18: end procedure
```

15.1 Exercise

1. Trace the execution of the algorithms
2. Modify the algorithms to answer the decision version of the problem

Dynamic Programming

Invented by Richard Bellman in the 1950s. Dynamic programming is an algorithm design technique that combines brute force and greedy ideas. It is used to solve problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations.

It makes use of:

- Recursion + dictionary (top-down memoization)
- Iteration (bottom-up tabulation)

An algorithm design technique for problems that have:

- **overlapping subproblems** (the same subproblems recur), and
- **optimal substructure** (an optimal solution is built from optimal subsolutions).

Key idea: solve each subproblem once, store the result in a table or dictionary, and reuse it to avoid recomputation.

Two main styles:

Top-down (memoization):

- write the natural recursive solution
- add a memo table to cache results as they are computed

Bottom-up (tabulation):

- identify the subproblem order from smallest to largest
- fill a table iteratively until the target answer is reached

Example: Fibonacci

Definition: $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$

Naive recursion (not DP)

```
function Fib(n):
  if n <= 1: return n
  return Fib(n-1) + Fib(n-2)
```

1. What is the time complexity of this function?

---

2. Draw the recursion tree for Fib(5)

This function is inefficient because it repeats the same calculation many times.

3. How many redundant function calls are made when fib(5) is called?

Recursion with Memoisation

```
memo = {}
function Fib(n):
  if n in memo: return memo[n]
  if n <= 1: return n
  memo[n] = Fib(n-1) + Fib(n-2)
  return memo[n]
```

Recursion with Memoisation

```
memo = {0: 0, 1: 1}

function FibM(n):
  if n in memo: return memo[n]
  memo[n] = FibM(n-1) + FibM(n-2)
  return memo[n]
```

4. What is the time complexity of FibM?

Iteration Bottom UP

Full table (classic bottom-up DP)

```
function FibTable(n):
  dp[0] = 0
  dp[1] = 1
  for i from 2 to n:
  dp[i] = dp[i-1] + dp[i-2]
  return dp[n]
```

- Stores **all values** $F(0) \dots F(n)$.
- Time O(n), space O(n).

Optimised bottom-up (rolling two values)

The "table" doesn't have to be an array — it can be a dictionary, or even just a couple of variables, as long as you're storing and reusing subproblem results.

```
function FibOptimised(n):
  if n <= 1: return n
  a = 0; b = 1       # F(0), F(1)
  for i from 2 to n:
  temp = a
  a = b
  b = temp + b
  return b
```

- Stores **only two values**: $F(i-2)$ and $F(i-1)$.
- Time O(n), space O(1).
- Same logic, just no big table.

Bottom-up and DP

- **Bottom-up DP:**
  - For problems with overlapping subproblems + optimal substructure
  - Builds solutions from small subproblems up to the final answer
  - Stores results (full table or compressed version) → avoids recomputation
  - Inherently **dynamic programming**
- **But not all bottom-up algorithms are DP**
  - Example: **Insertion sort**
    - Builds sorted array one element at a time
    - No reuse of subproblem results → not DP

Exercise

Consider the change-making problem with available coin denominations $\{1, 3, 5\}$. Let $dp[x]$ denote the minimum number of coins needed to make amount x.

Recurrence (problem structure)

$$dp[0] = 0, \qquad dp[x] = 1 + \min\big(dp[x-1],\, dp[x-3],\, dp[x-5]\big) \text{ for } x \geq 1,$$

ignoring any term with a negative index.

1. Write a recursive function that computes the minimum number of coins to make amount x using the recurrence above.
2. Draw the recursion tree for your recursive function when $x = 7$.
3. Modify your recursive algorithm to use a dictionary `memo` so that each subproblem x is solved at most once.
4. For $x = 7$, how many recursive calls are saved?
5. After computing $x = 7$ with your memoized algorithm, list the contents of `memo` (i.e., the x values that were cached and their $dp[x]$ values).
6. **Iterative algorithm.** Write an iterative bottom-up algorithm that fills $dp[0 \ldots n]$ and returns $dp[n]$.
7. Question 10 (VCAA2023)

   A dynamic programming algorithm is used to solve a change-making problem with the coin denominations of 1, 3 and 4. This algorithm iteratively solves the sub-problem of finding the fewest coins required to give k change. The algorithm is used to solve the problem for a total amount of 6.

   What are the final values stored in the array used by the algorithm? A. [0, 1, 2, 1, 1, 2, 2]

   B. [0, 1, 1, 3, 4, 4, 3]

   C. [0, 1, 1, 3, 4, 1, 3]

   D. [3, 3]
8. Question 20 (VCAA2019)

   Which one of the following descriptions of dynamic programming and divide and conquer is correct?

   A. Dynamic programming aims to solve the sub-problems once, whether they are overlapping or not, whereas divide and conquer does not care about how many times it needs to solve a sub-problem.

   B. Divide and conquer aims to solve the sub-problems once, whether they are overlapping or not, whereas dynamic programming does not care about how many times it needs to solve a sub-problem.

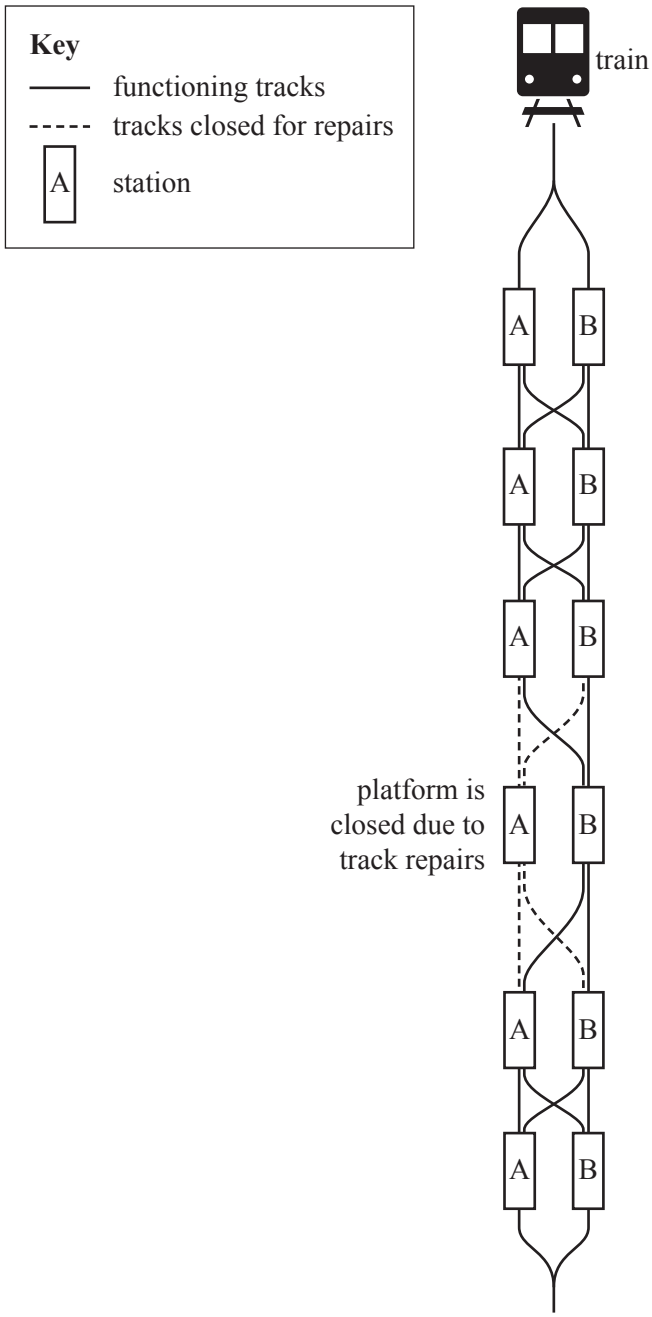
   C. Dynamic programming aims to find an optimal solution for a problem by splitting it into non-overlapping sub-problems, finding the optimal solutions for the sub-problems, and combining the optimal solutions for the sub-problems to form the optimal solution for the original problem.

   D. Divide and conquer aims to find an optimal solution for a problem by splitting it into overlapping sub-problems, finding the optimal solutions for the sub-problems with the intention of solving the overlapping sub-problems only once, and combining the optimal solutions for the sub-problems to form the optimal solution for the original problem.

Question 6 (7 marks)

A metropolitan train company has asked Maia to assist with scheduling trains travelling along a train network. Each station along the network has two platforms and interconnecting tracks. The expected wait time at each platform and the time taken to travel along that track depend on the number of staff allocated to assist with boarding and signalling. At times, platforms or tracks may be closed for repairs.

The proposed network is modelled below.



a. One approach to help with scheduling is to use a brute-force algorithm to reduce congestion for each train travelling through the network.

Explain whether or not this is feasible. Include the time complexity in your explanation. 4 marks

b. Maia suggests that a dynamic programming approach should be used for scheduling as the train network is likely to expand.

What properties of this problem make it suitable for a dynamic programming approach? 3 marks

Question 9 (10 marks)

A Melbourne property developer would like to buy a stretch of land to build some townhouses. She has identified a street on which she would like to build and has spoken to each of the property owners on the street to determine how much profit she would make by building on their land.

She has stored this information in an array, with each of the elements in the array representing the profit (or loss) for the purchase, in thousands of dollars.

For example, the array P = [80, −120, 50] would represent three houses, where buying the first would give the developer $80 000 in profit, buying the second would result in a $120 000 loss and buying the third would give a $50 000 profit.

The developer would like to solve the problem of determining the greatest profit she could make by buying a single, continuous stretch of land.

Three examples of continuous stretches of land with the greatest profit are provided below.

P = [80, −120, 50] : greatest profit is 80 thousand. ([**80,** −120, 50])

P = [20, −40, 90, 20, −50, 70] : greatest profit is 130 thousand. ([20, −40, **90, 20, −50, 70**])

P = [25, −10, −20, 50, 40, −30] : greatest profit is 90 thousand. ([25, −10, −20, **50, 40,** −30])

a. Describe a brute-force approach for solving this problem. Do **not** provide pseudocode in your answer. 3 marks

b. Discuss whether a divide and conquer approach could be used to solve this problem. Justify your answer. 3 marks

c. Write pseudocode for a dynamic programming algorithm that would solve this problem. 4 marks

Backtracking

Backtracking is an algorithm design technique that explores possible solutions step by step, abandoning ("backtracking") a partial solution as soon as it is determined that this path cannot lead to a valid complete solution.

It is often used for problems with:

- **Constraint satisfaction** (choices must obey rules),
- **Exponential search spaces** (too many possibilities to brute force blindly),
- **Combinatorial structures** (graphs, sets, paths, colourings).

The general pattern is:

1. Choose a candidate option.
2. Recurse to extend the partial solution.
3. If the choice leads to an invalid or dead end, backtrack.

—

Example 1: Maze Solving

Problem: Given a maze represented as a grid with walls and open cells, find a path from the start cell S to the goal cell G.

MazeBacktrack

```
 1: procedure MazeBacktrack(maze, pos)
 2:     if pos = G then
 3:         return true
 4:     end if
 5:     for each neighbour next of pos do
 6:         if next is unvisited and not a wall then
 7:             mark next as visited
 8:             if MazeBacktrack(maze, next) then
 9:                 return true
10:             end if
11:             unmark next                                  ▷ backtrack
12:         end if
13:     end for
14:     return false
15: end procedure
```

—

Example 2: Graph Colouring

Problem: Given a graph $G = (V, E)$ and k colours, assign colours to each vertex so that no two adjacent vertices share a colour.

GraphColour

```
1: procedure GraphColour(v)
2:     if v > |V| then
3:         return true                                  ▷ all vertices coloured successfully
4:     end if
5:     for each colour c in {1, ..., k} do
6:         if c is valid for vertex v (no adjacent vertex has c) then
7:             assign c to v
8:             if GraphColour(v + 1) then
9:                 return true
10:            end if
11:            remove colour from v                     ▷ backtrack
12:        end if
13:    end for
14:    return false
15: end procedure
```

—

Example 3: Hamiltonian Path

Problem: Determine whether a graph $G = (V, E)$ has a Hamiltonian path (a simple path visiting each vertex exactly once).

HamiltonianPath

```
1: procedure Hamiltonian(path)
2:     if length(path) = |V| then
3:         return true                                  ▷ all vertices included
4:     end if
5:     for each vertex u adjacent to last(path) do
6:         if u not in path then
7:             append u to path
8:             if Hamiltonian(path) then
9:                 return true
10:            end if
11:            remove u from path                       ▷ backtrack
12:        end if
13:    end for
14:    return false
15: end procedure
```

—

Exercise

1. Draw the recursion tree when solving a simple 4×4 maze using backtracking.
2. Trace the execution of GraphColour on a triangle graph (K_3) with $k = 2$ colours.
3. Explain why GraphColour can stop early if $k = 1$ and the graph has an edge.
4. Run the Hamiltonian path algorithm on a square (C_4). Show which paths are explored and which are abandoned.
5. Compare backtracking with brute force search. Why does pruning make it faster in practice?