Chapter 1: Algorithms and Design

1.1: Algorithms and Programs

1.1.1: Algorithm

An **algorithm** is a **well-ordered sequence** of **unambiguous** and **effectively computable** **operations**, which, when executed based on a **given set of initial conditions/inputs**, **produces the corresponding result** and **halts in a finite amount of time**.

Algorithms are a way to **solve problems**, especially those that are **repetitive** and **tedious** to do by hand.

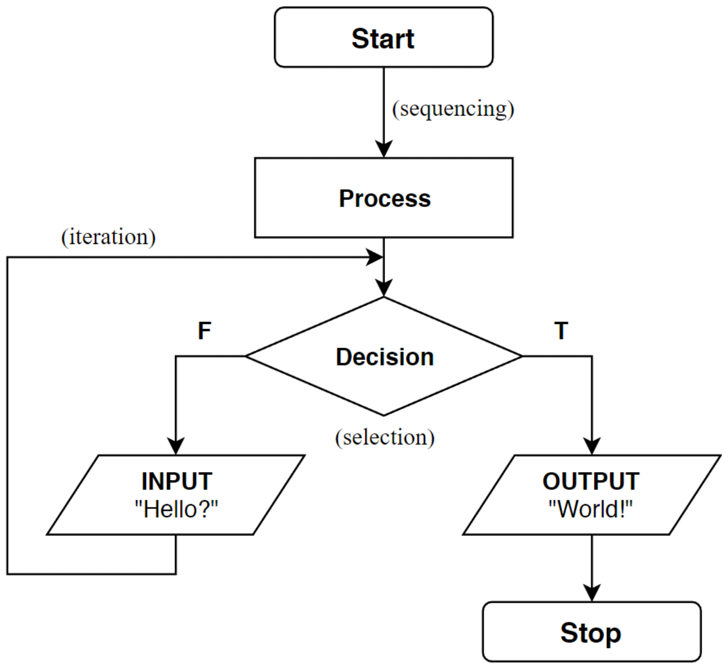
1.1.2: Program

A program is a **set of instructions** used by the computer to **perform a specific task**. It is the **actual expression of an algorithm** in a specific programming language.

1.1.3: Program Flowcharting

Program Flowcharting makes use of **formalised/standardised symbols** to represent different types of operations within a program. It essentially creates a visual representation of a program.

The diagram below shows the components of a **program flowchart**:



1.1.4: Pseudocoding

Pseudocoding makes use of the **English language** and **programming terminology** in a **code-like structure** in order to represent an algorithm. It is intended to display algorithms in a **friendly and understandable manner**, which is not dependent on the **strict rules** of programming languages.

Variables

To **declare** (create) a variable of a certain type, use this pseudocode:

```
DECLARE VariableName: INTEGER    // creates an integer VariableName
DECLARE VariableName2: STRING    // creates a string VariableName2
```

To **assign a value** to a variable that has been created, use this pseudocode:

```
VariableName ← 12
VariableName2 ← "Hello World!"
```

Receive and Output Information

To **input information** from a keyboard/file, use this pseudocode:

```
READ StudentName      // Gets StudentName from a records file
GET Number            // Gets Number from keyboard input from user
```

To **print to a printer**, use this pseudocode:

```
PRINT "Program Completed!"
```

To **write to a file**, use this pseudocode:

```
WRITE "Program Completed!"
```

To **write information to the screen**, use this pseudocode:

```
OUTPUT "Program Completed!"
DISPLAY "Hello World!"
PUT "Programming is FUN!!!!"
```

If-Then-Else Statements

To use an **if statement** using pseudocode:

```
IF <condition> THEN
      <statements>
ELSE
      <statements>
ENDIF
```

1.2: The Three Basic Control Structures

1.2.1: Sequence

The **sequence control structure** is defined as the **straightforward execution** of one processing step after another, with **no possibility** of **skipping** or **branching off** to another action.

Program Flowchart Representation: **PROCESS** (rectangle)

1.2.2: Selection

The **selection control structure** is defined as the **presentation of a condition**, where **control is diverted** to different parts of the program depending on whether the condition is **true** or **false**.

It is also known as an **If-Then-Else** statement.

Program Flowchart Representation: **DECISION** (diamond)

1.2.3: Iteration

The **iteration control structure** is defined as the **presentation of a set of instructions to be performed repeatedly**, given that **a certain condition** is **true**.

Iteration usually involves the use of **loops**.

While Loop/While-Do Loop

Checks for the **Boolean condition** **before** running any statement in the `WHILE` loop.

If the Boolean condition is **true**, the statements in the loop **will run**.
If the Boolean condition is **false**, the statements in the loop **will not run** (again).

Pseudocode for WHILE loop:

```
WHILE <Boolean condition>
      <statements>
ENDWHILE
```

Example of using WHILE loop:

```
i ← 0
WHILE i < 3
      OUTPUT "Number of times loop has been run", i
      i ← i + 1
ENDWHILE
```

Repeat-Until Loop

Checks for the **Boolean condition** **after** running all the statements in the loop. All the statements in the loop will **run at least once** when this loop is used.

If the Boolean condition is **true**, the statements in the loop **will continue to run**.
If the Boolean condition is **false**, the statements in the loop **will not run again**.

Pseudocode for REPEAT UNTIL loop:

```
REPEAT
      <statements>
UNTIL <Boolean condition>
```

Example of using REPEAT UNTIL loop normally:

```
i ← 0
REPEAT
      i ← i + 1
      OUTPUT "Number of times loop has been run", i
UNTIL i < 3
```

Example showing the difference between REPEAT UNTIL and WHILE loop:

```
i ← 4
REPEAT
      i ← i + 1
      OUTPUT "Number of times loop has been run", i
UNTIL i < 3
```

The statements within the REPEAT UNTIL loop will **run once** although the **condition** that `i < 3` is **not fulfilled**.

For Loop

A FOR loop uses an **explicit counter** for every iteration. Thus, the number of repetitions/iterations is **fixed** and **controlled by a variable** (the counter).

Pseudocode for FOR loop:

```
FOR <variable> = 0 TO 1:
      <statements>
NEXT <variable>
```

Example of using FOR loop:

```
FOR i = 0 TO 3
      OUTPUT "Number of times loop has been run", i
NEXT i
```

1.3: Sub-programs

Subroutines, functions and procedures are the **basic building blocks** of programs. These are small sections of code that **perform a particular task** within the program, and can be used within the program as much as needed.

- ✓ They avoid **repetition of commands** within the program, **shortening** the code and making it **easier to maintain**.
- ✓ They help to **define a logical structure** for the program, as the program is **broken down into smaller modules** with specific purposes.

Subroutines

A **sequence** of program instructions that can be **used and reused** to perform a **specific task** within the program. A subroutine ***may*** contain **input parameters** required for its processing.

Program Flowchart Representation: **SUBROUTINE** (rectangle w/ v. lines)

Procedures (do not return a value)

A **procedure** is a **self-contained subprogram** that is made up by an **ordered** set of coded instructions. Procedures can then be **called** from the main program.

When the procedure is passed, **control is given** to the procedure. Any **parameters** passed into the procedure will be **substituted** by their respective values, then the statements within the procedure get **executed**. When the procedure ends, **control is passed back** to the line that follows the procedure call.

Procedures **do not return a value** upon exiting.

Pseudocode for a PROCEDURE:

```
PROCEDURE ProcedureName(Parameter: <type>)
      <Statements>
ENDPROCEDURE
```

Example of using a PROCEDURE:

```
PROCEDURE CreateRecord(Name: STRING, PhoneNumber: INTEGER)
      <Statements>
ENDPROCEDURE
```

Functions (return a value)

Functions are **similar** to procedures, just that a function will **return a single value** to the point at which they are called. It is required that the **data type** of the value returned is **properly defined** in the function:

```
FUNCTION AddIntegers(Int1: INTEGER, Int2: INTEGER) RETURNS INTEGER
```

In pseudocode, the value returned is expressed like this:

```
RETURN ValueToReturn
```

1.3.1: Passing Parameters

The value of a **parameter** in procedures and functions can either be **changed** or **not changed**.

Passing by Value (ByVal)

Passing by value **creates a copy** of the original variable passed as the parameter. It **does not change the value** of the variable passed as the parameter.

Passing by Reference (ByRef)

Passing by reference allows variables in the procedure/function to **reference** the **memory address** of the original variable. This referencing **does change the value** of the variable passed as the parameter.

1.3.2: Scope of Variables and Constants

Variables and constants have a specific scope, which is the **region** within the program where the variable **is defined** and can be used.

Local Variables

Local variables are defined within the subroutine/function/procedure it is **declared in**. It exists only when the subroutine/function/procedure is run, and **cannot be accessed outside** the subroutine/function/procedure it is in.

Global Variables

Global variables are longstanding variables that is **accessible throughout the main program** and **all subroutines within** the program. It exists until the program terminates.

1.4: Structured Programming Concept

Structured programming is a **methodical approach** to designing a program that **emphasises** breaking a **large and complex task** into **smaller subtasks**.

Structured programming helps to **improve** the **clarity**, **quality**, **maintainability**, **and development efficiency** of a program.

Advantages of Structured Programming

- ✓ Subtasks can be **tested** individually and separately.
- ✓ Subtasks can be **reused** in the **main program** and used in **other programs**.
- ✓ **Improves** the **readability, debugging**, and **maintenance** of code.
- ✓ It allows programmers **working as a team** to work on **different subtasks**, **shortening the development time** for a large project.

Structured Program Theorem

It states that no matter how **complex** the task is, the task itself **can be solved** by splitting it into **subtasks**. These subtasks can be combined in 3 ways (**the 3 Control Structures**).

1.3: Recursion

Recursion is a way of programming where a function is able to **call itself** one or more times **in its body**, and then **terminates** when it reaches its **base case**.

```
FUNCTION Factorial(N: INTEGER) RETURNS INTEGER
      IF N = 0 THEN
            RETURN 1                      //base case, terminate here.
      ELSE
            RETURN N * Factorial(N – 1)//else, calls self & continue.
      ENDIF
ENDFUNCTION
```

1.3.1: Iteration vs Recursion

Iteration: Allows multiple blocks of instructions to be executed **repeatedly** and **in sequence** using loops, until a **condition is fulfilled**. **(iteration construct)**

Recursion: The function **calls itself** one or more times **in its body**, and then **terminates** when it reaches its **base case**. **(selection construct)**

1.3.2: Advantages/Disadvantages of Recursion

- ✓ Can **shorten** code
- ✓ Is **more intuitive** as it mimics humans' thought processes in problem solving
- ✓ More **mathematically abstract**
- ✓ More **readable** code, allowing for **more effective** maintenance, enhancement and development of code
- ✓ Some **complex problems** are done easier with recursion.
- ✗ May be **less appealing** to beginners
- ✗ It may be **more elegant**, but **more complex** to **design** and **test** at times.
- ✗ **Infinite recursion** occurs when programmer forgets to add a **base case**.
- ✗ Generally **less efficient** in terms of **time** and **memory** (call stack may overflow).

1.3.3: How Recursion Works

There is a **call stack** in the computer's **memory** that stores local variables, parameters passed to a function, and return values of a function.

PUSH: **Stores** an item into the **call stack**.
POP: **Removes** an item from the **call stack**.

For example:

```
1   FUNCTION S(N: INTEGER) RETURNS INTEGER
2       IF N = 0 THEN
3           RETURN 1                    //base case, terminate here.
4       ELSE
5           RETURN N + S(N - 1)         //else, calls self & continue.
6       ENDIF
7   ENDFUNCTION
```

1. When S(2) is executed, S(2) is **pushed** into the call stack.
2. 2 != 0, statement 5 is run. S(1) is **pushed** into the call stack.
3. 1 != 0, statement 5 is run. S(0) is **pushed** into the call stack.
4. 0 = 0, statement 3 is run. S(0) **returns** 1, and **popped** out of the call stack.
5. S(1) **returns** 1 + 0 = 1. S(1) is **popped** out of the call stack.
6. S(2) **returns** 2 + 1 = 3. S(2) is **popped** out of the call stack.
7. Function ends. 3 is **returned**.

Call Stack

③ S(0) = 0 ④
② S(1) = 1 ⑤
① S(2) = 2 ⑥

1.4: Time Complexity of Algorithms

Time complexity, also known as **order of growth**, is a ***rough*** measure of resources used in a **computational process**. It is represented using the **big O notation**, *O(n)*.

Time complexity allows us to know how **fast/efficient** an algorithm is when run on **large inputs** ($n \to \infty$), and is a measure of the **number of recursions** taken for the algorithm to **execute completely**.

For example, given the function `S(N)`:

```
FUNCTION S(N: INTEGER) RETURNS INTEGER
    IF N = 0 THEN
        RETURN 1                    //base case, terminate here.
    ELSE
        RETURN N + S(N - 1)         //else, calls self & continue.
    ENDIF
ENDFUNCTION
```

In the **best case scenario** (base case N = 0): time complexity is **O(1)**.
This is because 1 is **returned directly** after the function is run.

In the **worst case scenario** (N ≠ 0): time complexity is **O(N)**.
This is because S is **run a total of N + 1 times** before **returning a value** when run.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

1.5: Arrays/Lists

An **array/list** is a **fixed-length** **data structure**[1], with all data being of the **same type** (usually INTEGER or STRING). The elements of an array are a section of an array holding a piece of data. Each element also has an index that distinguishes one element of an array from another. It represents the **position** of an element in the array.

Pseudocode to implement an array

```
DECLARE ArrayName : ARRAY[<l>:<u>] OF <type>
```

Data	5	6	7	1
Index	0			3

The **iteration control structure (FOR loops)** should be used when accessing elements of a list/array.

1.5.1: Common Array Operations

- **Initialising values** into elements of an array
- **Processing elements** of an array
- **Searching** through an array using a **search key**
- **Printing** the contents of an array to a **report**

1.6: Searching

A **search algorithm** helps to **search** and **retrieve** data from the elements of an array, given a **search key**, which is the input to search within the array.

1.6.1: Linear Search

A **linear search algorithm** checks all elements of an array **one by one**, and **in sequence**, until the desired result is found. It can be used for both **sorted** and **unsorted** arrays. ....**Time complexity: O(N)**

Pseudocode for a Linear Search Algorithm

```
FUNCTION LinearSearch(InputArray: ARRAY, SearchKey: <TYPE>) RETURNS
BOOLEAN
        DECLARE flag : BOOLEAN
        flag ← FALSE
        FOR i = 0 TO size(ARRAY)
                IF ARRAY[i] = SearchKey THEN
                        flag ← TRUE
                ENDIF
        NEXT i
        RETURN flag
ENDFUNCTION
```

[1] A collection of elementary/primitive data types (such as Integer, Boolean, etc.)

1.6.2: Binary Search

Binary search is an algorithm that works on the principle of **divide and conquer**, that involves **iteration** or **recursion**. The array is split at the **middle of the array**, creating two **sub-arrays**. Depending on the condition, either the **left sub-array** or the **right sub-array** is chosen, essentially cutting the size of the array by half.

Binary search only works for **sorted arrays**. ....**Time complexity: $O(\log_2 N)$**

Pseudocode for a Binary Search Algorithm (Iteration)

```
FUNCTION BinarySearch(AR: ARRAY, InputValue: <type>) RETURNS BOOLEAN
      DECLARE ElementFound: BOOLEAN
      DECLARE LowElement, HighElement: INTEGER
      ElementFound ← FALSE
      LowElement ← 1
      HighElement ← size(AR)
      WHILE (NOT ElementFound) AND (LowElement <= HighElement)
            index ← INT((LowElement + HighElement)/2)
            IF AR[index] = InputValue THEN
                  ElementFound ← TRUE
            ELSE
                  IF InputValue < AR[index] THEN
                        HighElement ← index - 1
                  ELSE
                        LowElement ← index + 1
                  ENDIF
            ENDIF
      ENDWHILE
      RETURN ElementFound
ENDFUNCTION
```

1.7: Sorting

Sorting algorithms help to **rearrange** elements of an array **systematically** into a specified order based on the **sorting criterion** (such as **alphabetical order** or **numerical order**).

Sorting algorithms sort data using **2 basic operations**:

- **Comparison** operation – determines the **order** of an element
- **Swap** operation – **moves** the items, getting the array **closer** towards a sorted output

1.7.1: Advantages of Sorting

- ✓ It **optimises** the searching of data when **sorted** in a pre-defined order.
  - ➢ **Binary search** [*$O(\log_2 n)$*] is generally **faster** than **linear search** [*$O(n)$*]
  - ➢ **Binary search** only works for **sorted arrays**.
- ✓ It makes the information **more readable**.
- ✓ **Data processing** can be performed in a **defined order**.
  - ➢ e.g. to efficiently delete a data element from an array

1.7.2: Types of Sorting

1.7.2.1: Internal/In-place sorting

Internal sorting is performed when the number of elements is **small enough** to fit into the **main memory**.

1.7.2.2: External/Not-in-place sorting

External sorting is performed when the number of elements is **too large** to fit into the **main memory [RAM]** and data has to be **temporarily stored** in the **computer's storage** (e.g. in a temporary text file).

1.7.3: Bubble Sort

Bubble sort is a simple sorting algorithm that **compares adjacent elements** and **swaps them** depending on whether the elements are **out of order** in each pass.
After *N* passes, the last *N* elements are in the **correct position**.
Therefore, *N* - 1 passes are needed to sort *N* elements in their **correct positions**.

The **time complexity** for bubble sort is: **$O(n^2)$**.

Pseudocode for a Bubble Sort Algorithm

```
PROCEDURE BubbleSort(AR: ARRAY)
      DECLARE swapped: BOOLEAN
      swapped ← TRUE
      WHILE swapped = TRUE
            swapped ← FALSE
            FOR i = 1 TO size(AR)
                  IF AR[i] > AR[i + 1] THEN
                        DECLARE temp: <type>
                        temp = AR[i]
                        AR[i] = AR[i + 1]
                        swapped ← TRUE
                  ELSE
                        i ← i + 1  // increment i
                  ENDIF
            NEXT i
      ENDWHILE
ENDPROCEDURE
```

1.7.4: Insertion Sort

Insertion sort is a sorting algorithm that **partitions** the array into **two parts**: a **sorted sub-array** and an **unsorted sub-array**. Initially, the **sorted sub-array** consists of **the first element**, and the **unsorted sub-array** consists of the **rest of the elements**.

During each iteration, the first element of the unsorted sub-array is **compared** with the elements of the sorted sub-array, and **inserted** into the sorted sub-array. This **increases** the size of the sorted sub-array by 1, and **decreases** the size of the unsorted sub-array by 1.

The **time complexity** for insertion sort is: **$O(n^2)$**.

Pseudocode for an Insertion Sort Algorithm

```
PROCEDURE InsertionSort(AR: ARRAY)
      FOR j = 2 TO size(AR)
            DECLARE i: INTEGER
            i ← j - 1
            temp = AR[j]
            WHILE (i ⩾ 1) AND (temp < AR[i])
                  AR[i + 1] ← AR[i]
                  i ← i - 1
            ENDWHILE
            AR[i + 1] ← temp
      NEXT j
ENDPROCEDURE
```

1.7.5: Quick Sort

Quick sort is a sorting algorithm that uses the principle of **divide and conquer** to arrange elements of an array into their **correct positions**, using a **pivot** that divides the array into **two sub-arrays**.

1. The algorithm goes through the left sub-array and finds any element that **belongs** in the right sub-array by comparing with the **pivot**.
2. Then, the algorithm goes through the right sub-array and finds any element that **belongs** in the left sub-array by comparing with the **pivot**.
3. The algorithm then swaps the value of the elements belonging to the **wrong sub-array**.
4. As a result, after one pass, all the elements of the left sub-array are **less than** the value of the **pivot**, and all the elements of the right sub-array are **greater than** the value of the **pivot**. (depends on implementation)
5. This whole process **carries on** within the left sub-array, then within the right sub-array **recursively** (from steps 1 to 5).
6. In the end, a **sorted array** is obtained.

Pivot

The **pivot** can be any element of the array, although the **best** element to choose as the pivot is usually the **middle element**, with its index calculated as

$$\text{Pivot} = \text{INT}\left(\frac{\text{first} + \text{last}}{2}\right).$$

1.8: Data Validation and Verification

Data validation and **verification** techniques ensure that the data entered into a program/system is **accurate**, **reliable**, and **acceptable**.

1.8.1: Data Validation

Data validation is the **automated process** of checking the **value of input data** by the computer system/program to ensure that values entered are **acceptable/reasonable**. This is to ensure that the user **does not make a mistake** when entering data into a system.

It involves using the **properties of the data** to identify and inputs that are **obviously wrong**, and only checks whether the data is **reasonable enough** for the computer to accept.

- ✓ Allows the computer to **filter out obvious mistakes** when entering the data. The data **cannot be processed** until the validation succeeds.
- ✗ It **cannot prove** that the data entered is the **actual value** the user intended.

Some Data Validation Techniques

Type of Validation	Purpose
Presence Check	Checks whether data **has been entered** into a field or not.
Existence Check	Checks whether a **certain value is present** in a specified area.
Type Check	Ensures that the data value is of a **certain data type**.
Length Check	Ensures that the data has the **correct number of characters**. It can make sure either a **minimum** or **maximum** number of characters is entered.
Range Check	Ensures that the data value is **within a pre-determined range**.
Format Check	Ensures that the **individual characters** that make up the data is valid, and the data item matches a previously determined **format/pattern** with certain characters having certain values.

Check Digit

A **check digit** is an **extra digit** added to the end of a numeric code. It is determined by the **value** and the **positioning** of all other digits: any given code has only **one check digit**.

Modulo 11

A common method to use check digits for data validation is using **weighted modulus computation** for the check digit.

For example, the tens digit may have a weight of 2, etc.

For example: Check whether 123846 is a valid code.

Position	6	5	4	3	2	1
Digit	1	2	3	8	4	6
Position × Digit	**6**	**10**	**12**	**24**	**8**	**6**
Total	**60**					**6**

Since the total weighted sum of the digits is a **multiple of 11**, the code **123846** is valid.

For example: Find the check digit for 18956.

To find the check digit, find the weighted sum of all the digits like this:

Weighted Sum $= (1 \times 6) + (8 \times 5) + (9 \times 4) + (5 \times 3) + (6 \times 2)$
$= 6 + 40 + 36 + 15 + 12$
$= 109$

Then, find the weighed sum **modulo 11:**

Check Digit $= 109 \% 11$
$= 10 \Rightarrow$ **X**

Therefore, the check digit for 18956 is X, and **18956X** is the code.

1.8.2: Data Verification

Data verification is the **process** of ensuring that the data entered is **correct** and is **what the user intended**, such that there are no **transcription errors** or **transposition errors**.

Transcription Error

A **transcription error** is an error that is commonly made by **human operators** and **Optical Character Recognition (OCR)** programs, in which the **wrong character is entered** in a certain field. It can be caused by people **typing wrongly** or when OCR systems **wrongly recognise the characters** due to paper being crumpled or being in an unusual font.

eg: **Hello World!** vs **Helko Wirld!**

Transposition Error

A **transposition error** is an error whereby the **positions** of the characters entered in a field is **swapped** or **switched places**. It usually comes from people touch typing such that one character is entered before the other.

eg: **Hello World!** vs **Helol Wrold!**

Double Entry

(more information on **page 24**)

Double entry is a method of data verification where the data is **re-entered** into the same system, preferably by a different operator. This helps to spot the **transcription** and **transposition errors** that have been made when the data was **entered** into the system. If there is a discrepancy between the data entered between the first time and the second time, there is a transcription or transposition error that has been made by one of the two operators. The errors can then be **checked** and then **corrected** manually.

1.9: Character Sets

In order to store characters in a computer, it has to be represented as **binary data**[2] with a **specific binary number** representing a **specific character**.

ASCII & Unicode

ASCII is a character encoding system which is **7-bit**, and thus can encode **128 characters**. The characters encoded include the Latin alphabet, digits, and some symbols.

Unicode is a character encoding system with **32 bits** that can be used to encode characters. It can encode more than **4 billion (4,294,967,296) characters.** Thus, Unicode supports almost all characters and can represent many languages.

[2] Binary data refers to data only containing '1' and '0' bits,

1.10: Converting Bases 2, 8, 10, 16

1.10.1: Binary to Octal/Hexadecimal

To convert binary to **octal (3 bits)** or **hexadecimal (4 bits)**, split the bits up into **groups of 3 (octal)** or **groups of 4 (hexadecimal)**, then convert each group into the digit represented by the **group of 3** or **group of 4**.

Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7
001 000	10
001 001	11
001 010	12
001 011	13
001 100	14
001 101	15
001 110	16
001 111	17
010 000	20
010 001	21
010 010	22
010 011	23

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F
0001 0000	10
0001 0001	11
0001 0010	12
0001 0011	13

For example:

$1001100101010100_2 = 1001\ 1001\ 0101\ 0100_2$
$= \mathbf{9954_{16}}$

$1001100101010100_2 = 001\ 001\ 100\ 101\ 010\ 100_2$
$= \mathbf{114524_8}$

1.10.2: Decimal (Denary) to Other Bases

In order to convert from base 10 to other bases, two steps are required.

1. Convert the **decimal number** to binary
2. If converting to octal/hexadecimal, use the previous method above.

Converting from Decimal to Binary

To convert from **decimal** to **binary**, continually **divide by two**, keeping the quotient and remainder, using "long division":

Convert decimal 963 to hexadecimal.

2	963	
2	481	R 1
2	240	R 1
2	120	R 0
2	60	R 0
2	30	R 0
2	15	R 0
2	7	R 1
2	3	R 1
2	1	R 1
	0	R 1

Therefore $963_{10} = 1111000011_2$

$= 0011\ 1100\ 0011_2$

$= \mathbf{3C3_{16}}$

1.10.3: Converting Binary/Octal/Hexadecimal to Decimal

Digits have a **place value** that represents the value of the digit's position in the number. Hence, to convert **other bases** to decimal, just add up the sums of the place values of all the digits in the **binary**/**octal**/**hexadecimal** number, like this:

$$\begin{aligned}10110_2 &= (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0)\\ &= 16 + 4 + 2\\ &= \mathbf{22_{10}}\end{aligned}$$

$$\begin{aligned}1721_8 &= (1 \times 8^3) + (7 \times 8^2) + (2 \times 8^1) + (1 \times 8^0)\\ &= 512 + 448 + 16 + 1\\ &= \mathbf{977_{10}}\end{aligned}$$

$$\begin{aligned}7EF_{16} &= 7 \times 16^2 + 14 \times 16^1 + 15 \times 16^0\\ &= 1792 + 224 + 15\\ &= \mathbf{2031_{10}}\end{aligned}$$

Quick Computing Theory Notes (Part 2)

Systems Analysis (in a Nutshell)

Systems Development Cycle

The **systems development cycle** is made up of the various **stages/phases** that have to be completed to create a **new modified computer system**.

It is a cycle as after a period of time, the system might need to be **modified/replaced** and the process has to be repeated.

Step 0: Feasibility Study and Problem Definition

Feasibility Study

The **feasibility study** is the **preliminary investigation** of a problem to decide whether a **solution is possible** and how the solution **may be done**. It contains:

- **Context** of the problem
- **Evaluation/Simple analysis** of the problem
- **Ways** the problem can be solvable
- **Cost-benefit analysis** to determine whether the solution is affordable

Terms of Reference

The analyst must:

- **Investigate** and **report** on the existing system
- **Specify objectives** for the system and **whether** they will be met by the new system
- **Recommend** the most suitable system to achieve the objectives
- **Prepare** a **cost-benefit analysis**
- **Prepare** a plan for **implementing the new system** within a short time scale.

Factors for Feasibility

- **Technical** – is the technology feasible?
- **Economic** – is it economically feasible?
- **Social** – is the social effects likely to be damaging?
- **Availability** of **hardware/software**
- **Affordability** of **running** the solution
- **Time**
- **Skill** of workers
- **Effect** on customer

Cost-benefit Analysis

Costs

The **costs** of a new system may include:

- **Equipment costs** (computers and peripherals)
- **Installation costs**
- **Development costs** (of the system)
- **Personnel costs** (training, recruitment, salaries, etc.)
- **Operating costs** (consumables like disks, maintenance, etc.)

Benefits

The **benefits** of a new system may include:

- **Savings** in personnel costs, operating costs, etc.
- **Extra sales revenue** due to better marketing information
- **Improved cash flow position** since invoices can be sent faster, etc.

Why to computerise?

Some **manual systems** have characteristics that would be **more suited for computerisation**. These characteristics include:

- **Volume**
- **Requirement** for information to be **available** from several locations
- **Very accurate calculations**
- **Duplicated effort** involved (iteration)
- **Manual methods** are too **slow**
- **Data** has to be constantly **updated** and **accessible**

Problem Statement

These problems in the current system which require the use of computerisation can be listed in the **problem statement**.

Some other reasons may be:

- **Transcription/Transposition errors** from human input
- **Layout** of organisation of data
- Etc.

If the solution is found to be feasible using computerisation, the **system development cycle** can start and the more advanced **systems analysis** process can take place.

Systems Analysis

Systems analysis is the analysis of systems in businesses and organisations that help them run smoothly and efficiently. It is a **detailed look** at the current system and what the new system will be **required** to do. It is similar to the **feasibility study** but is **more detailed**.

Analysis – the **detailed** look at what the users require of the system that the project is to implement. A **requirements specification** is produced, which forms the **contract** between the **customer** and **the developer of the system**.

A person who analyses systems is known as a **systems analyst**. They are usually employed by organisations and businesses to help them **improve their systems** and become **more efficient** or **profitable**.

The Process of Systems Analysis

1. **Research** - **Collecting** information on how the present system works
2. **Analysis** - **Examining** how the present system works and **identifying problems** with it.
3. **Design** - Coming up with a **new system** that will **fix the problems** of the current system.
3. **Development** - **Creating** the new system from the design.
4. **Testing** - **Checking** if the new system **works as expected** (doesn't have any errors)
5. **Documentation** - **Creating** documents that describe **how to use** the new system and **how it works**
6. **Implementation** - **Replacing** the present system with the **new system**.
7. **Evaluation** - **Checking** that the new system **meets all expectations**.

Step 1: Research

Before the systems analyst can make any recommendations about a **new system**, they first have to understand how the **present system** works.

As much information about the current system has to be **gathered** as possible. The techniques that can be used are:

1.1a: Observation

The **systems analyst** walks around the **organisation** or **business**, watching **how things work** with their own eyes.

- ✓ Can gather **first-hand, unbiased** information
- ✗ People may **act differently** if they are aware they are being observed.

1.1b: Collecting Documents

The **systems analyst** can collect examples of documents to **gain an understanding** of the **type and quantity of data** that **flows** through the **business** or **organisation**.

- If the documentation is **poor quality/insufficient**, collecting documents may not be very helpful.

1.2: Interviews

The **systems analyst** can interview **key people** within the current system to find out **how it works**.

- ✓ Gather a lot of **very detailed** information
- ✗ Interviews can take **a long time**, thus may not be feasible, especially if **a lot of people** are involved in the **current system**.

1.3: Questionnaires

The **systems analyst** can create a questionnaire to **gather information** from **large groups of people**.

- ✓ Can gather data from **many people**
- ✗ People **may not answer the questions seriously**, making the information **less reliable**.
- ✗ Information gathered is **limited to the questions asked** in the questionnaire by the systems analyst.

NOTE: If the question states that 3 methods of data collection for the current system are required, **state all four**, just that **observation** and **collecting documents** can be put into the same point.

Step 2: Analysis

The **systems analyst** looks through the **information collected in Step 1** to **understand** how the system works, and to try and **identify problems** that need to be fixed.

2.1: Identifying Inputs, Outputs, and Processes

Every system has **inputs** and **outputs**, and the **system analyst** needs to identify the **data input** and **output** to the present system. This is because any **new system** that is designed will have to deal with similar inputs and outputs as the **present system**.

For similar reasons, the system analyst also has to identify the **processes** of the **current system**.

2.2: Identifying Problems

It is the job of the **systems analyst** to find out where the **problems** in a **system** are. If these problems are resolved, the system will work **more efficiently** and **smoothly**, and be more **profitable** for businesses.

2.3: Requirements Specification

The **requirements specification** is a list of requirements for the **new system**. The techniques for obtaining such requirements are:

- Interviewing
- Joint Application Design workshops
- Reviewing existing documents
- Analysing existing system
- Creating prototypes
- Observing current working practices

The **new system designed** must **meet these requirements**.

2.4: What software/hardware needed?

Hardware

What **computers/network/servers**?
Any **special input/output devices**? (e.g. barcode readers)

Software

Are there any **existing off-the-shelf applications**?
Does the software need to be **custom-made**?

2.5: Data Flow Diagrams

Data flow diagrams are diagrams that show **how data flows** through a system. These analysis tools show how the data is **input**, **output**, **stored**, and **processed** in a system.

Step 3a: Design

3.1: Systems Flowcharts

The **systems flowchart** is a diagram used to **describe** a **complete data processing system**.

It describes it at an **individual process level**, and the flow of data through the operations is **diagrammatically described**, down to the level of the **individual programs** using the system requirements.

The details of the programs themselves are **not included**, as they are included with the **program documentation (Step 5)**.

It shows:

- The **tasks** to be **carried out** in the new system
- The **devices** to be used
- The **input/output media**
- The **files** used in the system

3.2: Other Design Tools

Program Flowcharts

The **program flowchart** shows the operations involved in a **computer program**. It is part of the **permanent record** of a finished program for **maintenance (Step 7b)**.

Pseudocode

(covered in more detail on **page 2**)

Pseudocoding uses **control structures** and **keywords** like those in programming languages to describe a **program** or **system design**.

Decision Table

It is a table that specifies the **actions taken** when **specific conditions arise**.

3.3: User Interfaces

3.3.1: Good UI Design

A good UI design takes into consideration:

- Who **uses** the system
- The **tasks performed** by the system
- The **environment** where the system is used
- What is **technologically feasible**
- **SAVE BUTTON !!!!!!!!!!**

3.3.2: Types of UI

Some types of UI include:

- **Command line interface** (CLI)
- **Menu** interface
- **Graphical user interface** (GUI)
- **Form** interface
- **Touchscreen** interface
- ...and many more...

3.4: Data Inputs into a System

To get data into a system, data must first be **captured**, then **input** to a computer, either **manually** or using a **data capture device**.

Some Data Capture and Input Methods

1. **Paper Forms**
   Information is **written** into the forms, and **input** into the computer, either **manually** or using **machine-reading technology (OMR/OCR)**.

2. **Barcode Readers**
   Barcode readers capture the **numeric code** represented by the barcode.

3. **Card Reader**
   Card readers read data on the **magnetic strip/memory** on cards.

4. **Camera**
   Captures still or **moving images** that can be **input** to the computer for processing.

3.5: Data Validation and Verification

3.5.1: Validation

(covered in more detail on **page 15**)

Data validation checks whether the data input is **valid** or not.
The five types of **data validation checks** are:

1. **Presence** check - Is the data **present** within a field?
2. **Range** check - Is the data **within** the **specified range**?
3. **Length** check - Is the data **too short** or **too long**?
4. **Type** check - Is the data the right **type**?
5. **Format** check - Is the data the right **format**? (e.g. dates)

3.5.2: Verification

Data verification checks whether the data input is **correctly input** or not.
The two types of **data verification checks** are:

1. **Proof Reading**
   A person compares the **original data** with the **data** in the computer. If mistakes are spotted, they can be **corrected** by the person.
   - ✓ **Quick** and **simple**
   - ✗ **Doesn't catch** every mistake

2. **Double Entry**
   A person (preferably another person) re-enters the data into the system. If differences are spotted by the system, an **error** is generated and the person can **correct** the differences in the system.
   - ✓ Catches **almost every mistake**
   - ✗ **More time** and **effort** needed

3.6: Designing the System Processes

Any system has to **process** the data given. The **systems designer** has a number of things to consider:

Designing Data and File Structures

A **data structure** is an **organised collection** of data. It is usually a **database** in which data will be **stored** as it is being **processed**.

When designing a **database**, the **systems designer** must consider:

- **Type** of data stored
- **Size** of data (length)
- **Field names** to use
- **How many records** to be stored

The **designer** also must consider what **backing storage device** or **medium** to store the data in:

- **Frequency** of accessing data
- **Speed** of accessing data
- **Size** of data files

3.7: Algorithms

To process the data, the **systems designer** must design the actual steps to be followed to process the data (**algorithms**).

3.8: Designing System Outputs

There are usually **two** types of output from a system that needs to be designed: **on-screen reports** and **printed reports**.

On-screen reports

Designing an **on-screen report** is similar to designing an **on-screen form**.

When designing an **on-screen report**, the designer should:

- Show **all** necessary fields
- Have fields that are the **right size** for the data
- Have **easy-to-understand** instructions
- Make good use of **available screen area**
- Make good use of **colours** and **fonts** to make data clear

Printed reports

Designing a printed report is similar to designing an **on-screen report**, just that it is **printed** on a piece of paper.

Step 3b: Development

It is the process of **constructing** the **actual computer system** itself.
It includes:

- Identifying the **modules** to be used and **specifying** them
- Identifying the **main data structure** within the programs
- Identifying the **main algorithms** to use as **pseudocode** or **structure diagrams**
- **Producing** the program and any other elements of the system

3.9: Software Development Cycle

The **software development cycle** is the sequence of steps taken to **produce working software**.

The stages are:

1. **Overall design** - identifies **what is needed** and **splits** it into **self-contained modules**
2. **Module design** - decides how **each module** performs its task
3. **Module production** - programs **each module** using a programming language.
4. **Module testing** - ensures that each module **works independently**
5. **Combining modules** to form the **complete system**
6. **Integration testing** - ensures that modules **work together**

3.10: Program design

It involves **drawing structure charts** and writing **detailed program specifications**.

3.11: Prototyping

It is the building of a **working model** of the system to **evaluate** it, **test** it, or **have** it **approved** before building the **final product**.

While some prototypes get **developed** into the final product, others are **discarded**.

Step 4: Testing

Testing is the process of **detecting errors** in a system.

4.1: Test Plan, Test Data and Test Cases

Test Plan

It is a plan containing **details** on **every single thing** to be tested.
(e.g. does XXX work? / does this reject invalid data?)

It is **very detailed** and contain many **precisely specified tests**.

Test Cases and Test Data

Test data are the data to be tested.
Test cases are the **test data** and the **expected outcomes** from the test data.

4.2: Dry Run

A **dry run** (or desk checking) is a **manual check** through a program or system **step-by-step**. This is helpful in **locating errors** (especially run-time errors).

4.3: Unit and Integration Testing

Unit test	–	Each part of the system in **individually tested**.
Integration test	**-**	All parts are **put together** and the **complete system** is tested.

4.4: Bottom-up and Top-down Testing

Bottom-up Testing

- Components on the **lowest level** of the **hierarchy** are combined and tested first.
- The software is put together by including **successively higher-level** components.

Top-down Testing

- The **skeleton** of the **complete system** is tested, where **individual modules** are replaced by 'stubs'.
- These 'stubs' stand in for modules while they are **developed**. They may display a message stating that the module has been executed.
- In **subsequent tests**, the individual modules are included when they are **completed**.

4.5: White-box and Black-box Testing

White-box Testing

White-box testing refers to testing that is done by the **programmers of the system** with the **knowledge** of the **underlying code** that runs the method. This helps the developers to test **every possible route** through the methods in the program.

Black-box Testing

Black-box testing refers to testing that is done by **the system's test engineers** whereby **no assumption is made** about how the code of the system works and the **test data** is obtained from an examination of the **requirements statement** of the system.

4.6: Developmental Testing

Developmental testing is the **repeated testing** of a system such that the results can be used for **further design and development**.

Alpha Testing

Alpha testing is the issue of the software to a **restricted number of testers** within the **developer's own company**. The alpha version may be **incomplete** and **have some faults**.

Beta Testing

Beta testing is the issue of the software to a number of **privileged customers** in exchange for their **constructive comments**. The beta version are usually **similar to the finished product**. Beta testing takes place after the results of the alpha testing has been studied and **changes have been made**.

Acceptance Testing

Acceptance testing is the testing carried out to **prove** to the **customer** that the system works correctly. It is carried out **after** the system is completed, and **ready to be handed over** to the customer.

4.6: Test Data

Live Data

Live data is data that would normally be used in the current system.

Normal, Abnormal, and Extreme Data Values

Normal data is data that would **normally be entered** into the system.
Extreme data is normal data, but at the **absolute limits** of the normal range.
Abnormal data is data that **should not normally be accepted** be accepted into the system, as the values are invalid.

4.7: Debugging, Errors, and Breakpoints

Debugging

It is the **detection**, **location** and **correction** of faults/bugs that cause errors in a program. These errors are detected by **observing error messages** or by finding **unexpected results** in the test output.

Errors

Errors are **faults** or **mistakes** in a computer program or system that causes it to produce the wrong results or not work. A **bug** is a fault in the program that **causes errors**. **Error messages** are generated by the computer to help the user **locate the likely source** of the errors.

Some types of errors include:

- **Execution errors** - errors detected **during program execution**, such as **division by 0 errors**, or **overflow errors**.
- **Compilation errors** - errors detected **during compilation**, such as **syntax errors**.
- **Linking errors** - errors caused when a program is **linked to library routines**.
- **Syntax errors** - errors caused due to **incorrect program syntax**.
- **Logical errors** - **mistakes** in the **program design**, usually leading to program **displaying wrong results**.
- **Semantic errors** - errors caused by **violating rules** of the language.

Breakpoints

It is a position within the program where the **program is halted** to aid in debugging. When the program is halted, the programmer can **investigate the values** of **variables, memory locations, and registers**. This helps the programmers to locate errors, particularly **run-time errors**.

4.8: System Testing

There are several ways to test the entire system:

Functional Testing - **ensuring all parts of the system works correctly with test data.**

Recovery Testing - **ensuring that the system can cope and recover from failures (power, hardware, etc.)**

Performance Testing - **tests whether the system can cope with a realistic workload.**

Step 5: Documentation

5.1: User Documentation

User documentation is intended to **help the users** of the system.

As the users are **non-technical people**, they do not need to know about how the system works, just **how to use it**.

User documentations may include:

- **Minimum hardware** and **software** required
- How to **install**, **start** and **stop** the system
- How to **use the features** of the system
- **Screenshots** showing **typical usage** of the system
- Example **inputs** and **outputs**
- Explanations to any **error message** shown
- **Troubleshooting guide**

5.2: Technical Documentation

Technical documentation is intended to **help the maintainers** of the system.

It provides information on **how the system works**.

Technical documentations may include details on:

- **Hardware** and **software** required
- **Data structures** used in the system
- **Expected inputs**
- **Validation checks**
- How **data** is **processed**
- **Data flow diagram**
- **System flowchart**

5.3: Systems Documentation

Systems documentation describes the results of **systems analysis**, what is **expected** of the system, the **overall design decisions**, the **test plan**, and the **test data** with the **expected results**.

5.4: Systems Specification

Systems specification is a **complete description** of the **whole system**, containing **data flow diagrams**, **system flowcharts**, **inputs**, **files**, **outputs**, and **processing**.

5.5: Program Documentation

Program documentation is the **complete description** of the **software intended for use** when **altering** or **adapting** the software, including the **purpose** of the software, **restrictions** on use of the software, **input** and **output** data, **flowcharts**, **program listings** and **notes** to assist in future modifications.

Step 6: Implementation

The **implementation** of the new system occurs when the old system is replaced.

6.1: Direct changeover

The old system is **stopped immediately** and the new system **takes over**.

- ✓ New system can be **started immediately**
- If the new system fails, data is lost as there is **no back-up system**.

6.2: Parallel running

The new system is started but the **old system continues running for a short while in parallel** with the new system. After the new system is proven to work, the old system can stop operating.

- ✓ If the new system fails, no data is lost as there is a **back-up system**.
- ✓ The **outputs** of **both systems** can be **compared** to check that the new system is working correctly.
- **Entering data** into two systems and **running both systems** takes up **more time and effort**.

6.3: Phased implementation

The old system is replaced by the new system **gradually, in phases**.

- ✓ Allows users to **gradually get used** to the **new system**
- ✓ Staff training can be done **in stages**
- If the new system fails, data is lost as there is **no back-up system**.

6.4: Pilot running

The new system is **trialled** (pilot) in **one part** of the **business/organisation**.

- ✓ Features can be **fully trialled**
- ✓ Staff part of the pilot scheme can **train other staff**.
- If the new system fails, data is lost as there is **no back-up system**, for the section of the business/organisation trialling the new system.

Step 7a: Evaluation

The **evaluation** process **assesses** the system to see if:

- It does what it's **supposed to do**
- It is **working well**
- Everyone is **happy** with it

7.1: What does an Evaluation look for?

When the **systems analyst** evaluates the new system, the following questions will be asked:

Is the system…

- **Efficient**? Does it **save time** and **resources**? Does it operate **quickly** and **smoothly** with **minimal waste**?
- **Easy to use**? Can users use the system with **minimal training**?
- **Appropriate**? Is it **suitable** and **meets the needs** of the business/organisation?

7.2: How is a System Evaluated?

The systems analyst can use a number of techniques to evaluate the system:

Checking against the Requirements Specification

The systems analyst goes through the **requirements** in the Requirements Specification **one-by-one** and checks whether the new system meets them.

Checking the Users' Responses

They can obtain **feedback** from the users of the new system, like in **Step 1**, through **questionnaires**, **interviews**, and **observation**.

7.3: Post-implementation Review

Once the system is up and running, a **review** needs to be performed to **confirm** that the new system is **fulfilling expectations**, and to identify any **weaknesses** or **modifications** that need to be made.

Step 7b: Systems Maintenance

Systems maintenance involves:

- **Updating** the system to adapt it to **changing circumstances**, **legislation**, or **requirements**
- **Correcting** any errors that come to light
- **Documenting system updates** and **corrections**

There are several types of systems maintenance, including:

Perfective Maintenance	-	**making improvements, increasing ease of use**
Adaptive Maintenance	-	**take account of changes in business or legislation over time**
Corrective Maintenance	-	**correct any errors that may have arisen**