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.

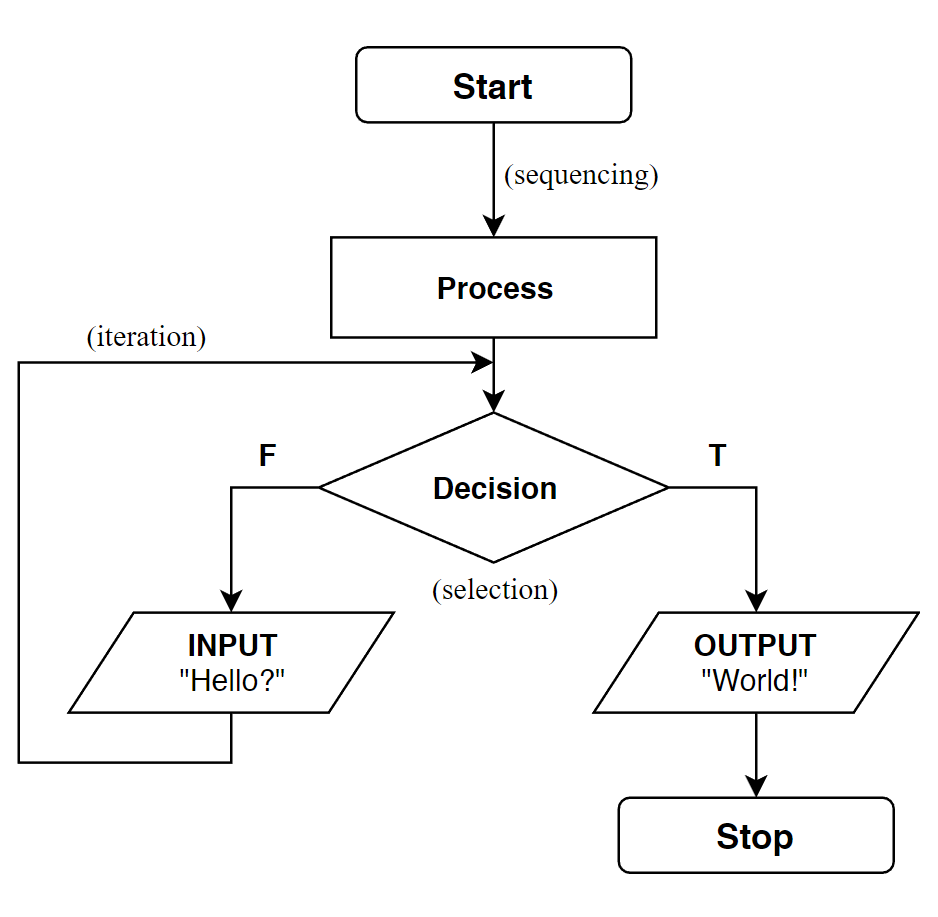
## 1.1.2 Program

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 (ByValue)

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** breakinga **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.
* Subtaskscan 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).**

# 1.5 Recursion

**Recursion** is a way of programming where a function is able to **call itself** one or moretimes **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

## 1.5.1 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)**

## 1.5.2 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).

## 1.5.3 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

|  |  |
| --- | --- |
| **Call Stack** | |
|  | |
| S(0) | = 0 |
| S(1) | = 1 |
| S(2) | = 3 |

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**.

# 1.6 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 → ∞), and is a measure of the **number of recursions** taken for the algorithm to **execute completely**.

For example, given the function S(N):

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

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(n2) < O(n3) < O(2n)**

# 1.7 Arrays/Lists

An **array/list** is a **fixed-length** **data structure**[[1]](#footnote-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 | 1 | 2 | 3 |

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

## 1.7.1 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**

# 1.8 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.

## 1.8.1 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

## 1.8.2 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(log2 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

# 1.9 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

## 1.9.1 Advantages of Sorting

* It **optimises** the searching of data when **sorted** in a pre-defined order.
  + **Binary search** [*O(log2 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

## 1.9.2 Types of Sorting

### 1.9.2.1 Internal/In-place sorting

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

### 1.9.2.2 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).

## 1.9.3 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(n2)**.

**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

## 1.9.4 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(n2)**.

**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

## 1.9.5 Quicksort

**Quicksort** 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**.

The average **time complexity** for quicksort is **O(n log n)**.

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

.

# 1.10 Data Validation and Verification

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

## 1.10.1 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 × 6) + (8 × 5) + (9 × 4) + (5 × 3) + (6 × 2)  
 = 6 + 40 + 36 + 15 + 12  
 = 109

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

**Check Digit** = 109 % 11  
= 10 ⇒ **X**

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

## 1.10.2 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.

# 1.11 Character Sets

In order to store characters in a computer, it has to be represented as **binary data[[2]](#footnote-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.

# 1.12 Converting Bases 2, 8, 10, 16

## 1.12.1 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** |  |  | **Binary** | **Hexadecimal** |
| 000 | 0 |  |  | 0000 | 0 |
| 001 | 1 |  |  | 0001 | 1 |
| 010 | 2 |  |  | 0010 | 2 |
| 011 | 3 |  |  | 0011 | 3 |
| 100 | 4 |  |  | 0100 | 4 |
| 101 | 5 |  |  | 0101 | 5 |
| 110 | 6 |  |  | 0110 | 6 |
| 111 | 7 |  |  | 0111 | 7 |
| 001 000 | 10 |  |  | 1000 | 8 |
| 001 001 | 11 |  |  | 1001 | 9 |
| 001 010 | 12 |  |  | 1010 | A |
| 001 011 | 13 |  |  | 1011 | B |
| 001 100 | 14 |  |  | 1100 | C |
| 001 101 | 15 |  |  | 1101 | D |
| 001 110 | 16 |  |  | 1110 | E |
| 001 111 | 17 |  |  | 1111 | F |
| 010 000 | 20 |  |  | 0001 0000 | 10 |
| 010 001 | 21 |  |  | 0001 0001 | 11 |
| 010 010 | 22 |  |  | 0001 0010 | 12 |
| 010 011 | 23 |  |  | 0001 0011 | 13 |

**For example:**

10011001010101002 = 1001 1001 0101 01002

= **995416**

10011001010101002 = 001 001 100 101 010 1002

= **1145248**

## 1.12.2 Decimal (Denary) to Other Bases

In order to convert from base 10 to other bases, there are two methods:

### Method 1

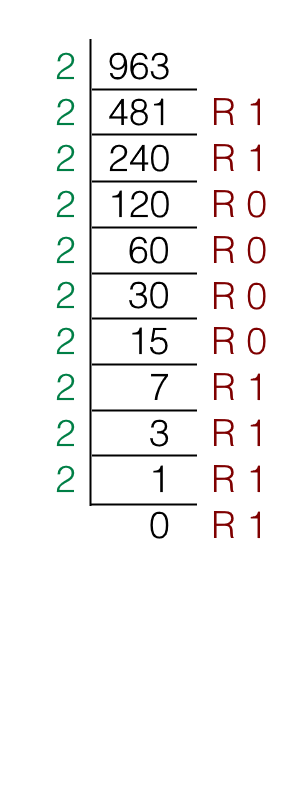
1. Convert the **decimal number** to **binary/octal/hexadecimal** directly by repeatedly dividing by the base (**2/8/16**).

### Method 2

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 binary, then convert it to hexadecimal.**

Therefore 96310 = 11110000112

= 0011 1100 00112

= **3C316**

#### (NOTE: alternatively, you can also “long divide” by 16 directly.)

## 1.12.3 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:

101102 = (1 × 24) + (0 × 23) + (1 × 22) + (1 × 21) + (0 × 20)

= 16 + 4 + 2

= **2210**

17218 = (1 × 83) + (7 × 82) + (2 × 81) + (1 × 80)

= 512 + 448 + 16 + 1

= **97710**

7EF16 = 7 × 162 + 14 × 161 + 15 × 160

= 1792 + 224 + 15

= **203110**

1. A collection of elementary/primitive data types (such as Integer, Boolean, etc.) [↑](#footnote-ref-1)
2. Binary data refers to data only containing ‘1’ and ‘0’ bits, [↑](#footnote-ref-2)