كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

1. Introduction

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms:

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

1.2 Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics:

- **Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input**: An algorithm should have 0 or more well-defined inputs.
- **Output**: An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness**: Algorithms must terminate after a finite number of steps.
- **Feasibility**: Should be feasible with the available resources.

- **Independent**: An algorithm should have step-by-step directions, which should be independent of any programming code.

1.3 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is a problem and resource-dependent. Algorithms are never written to support a particular programming code.

Algorithm writing is a process and is executed after the problem domain is well-defined. So it should know the problem domain, for which we are designing a solution.

The common constructs such as like loops (do, for, while), flow-control (if-else), etc are all programming languages share these basic codes .so, it can be used to write an algorithm.

Example:

Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c** ← Input
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c** ← Output
Step 7 − STOP ← terminate

1.4 Algorithm Analysis

The efficiency of an algorithm can be analyzed at two different stages, before implementation, and after implementation. They are the following

1. Priori Analysis :(execution time) this is a theoretical analysis of an algorithm. The efficiency of an algorithm is measured such as processor speed, are constant and has no effect on the implementation.

2. Posterior Analysis :(running time) this is an empirical analysis of an algorithm. The selected algorithm is implemented using a programming language. This is then executed on the target computer machine. In this analysis, actual statistics like running time and space required, are collected.

1.5 Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

1.5.1Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

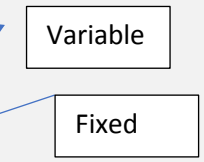
- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory
- allocation, recursion stack space, etc.

Space complexity S(x) of any algorithm x is S(x) = C + S(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Following is a simple example that tries to explain the concept.

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop



S(x) = C + S(I) → s(x)=1+2→=3

1.5.2 Time complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

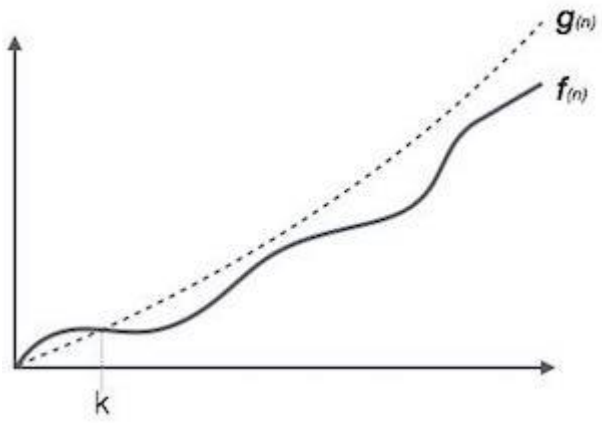
the time required by an algorithm falls under three types :

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.

- **Worst Case** − Maximum time required for program execution.

1.5.3 Big O Notation

Big O Notation is commonly used asymptotic notations to calculate the running time complexity of an algorithm**.** It is the formal way to express the upper bound of an algorithm's running time. Moreover, it measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



Following is a list of some common asymptotic notations:

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$