Design Analysis and Algorithms

INTRODUCTION

Algorithm: An algorithm is a step by step procedure to represent the solution for a given problem. It consists of a sequence of steps representing a procedure defined in a simple language to solve the problem.

It was proposed by Persian mathematician, ABU JAFAR MOHAMMED IBN MUSA AL KHOWARIZMI IN 825 A.D.

Characteristics of an Algorithm:

1) An algorithm must have a finite number of steps.
2) An algorithm must be simple and must not be ambiguous.
3) It must have zero or more inputs.
4) It must produce one or more outputs.
5) Each step must be clearly defined and must perform a specific function.
6) There must be a relationship between every two steps.
7) It must terminate after a finite no. of steps.

How to develop an algorithm

1. The problem for which an algorithm is being devised has taken precisely and clearly defined.
2. Develop a mathematical model for the problem. In modeling mathematical structures that are best suited are selected.

3. Data structures and program structures used to develop a solution are planned.

4. The most common method of designing an algorithm is stepwise refinement. Step wise refinement breaks the logic into a series of steps. The process starts from converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements.

5. Once an algorithm is designed, its correctness should be verified. The most common procedure to correct an algorithm is to run the algorithm on various test cases. An ideal algorithm is characterized by its run time and space occupied.

How to analyze an algorithm:

Analysis of algorithms means estimating the efficiency of an algorithm in terms of time and storage an algorithm requires.

1. First step is to determine what operations must be done and what their relative cost is. Basic operations such as addition, subtraction, comparison have constant time.

2. Second task is to determine the number of data sets which cause the algorithm to exhibit all possible patterns. It requires understanding the best and worst behavior of algorithms for different data configurations.

The analysis of algorithm two phases,

1) Priori Analysis: In this analysis we find some functions which bind algorithm time and space complexities. By the help of these functions it is possible to compare the efficiency of algorithms.

2) Posterior Analysis: Estimating actual time and space when an algorithm is executing is called posterior analysis.

Prior analysis depends only on number of inputs and operations done, whereas posterior analysis depends on both machine and language used, which are not constant. Hence analysis of most of the algorithms is made through priori analysis.

How to validate an algorithm:

The purpose of the validation of an algorithm is to assure that this algorithm will work correctly independently of the issues such as machine, language, platform etc. If an algorithm is according to given specifications and is producing the desired outputs for a given input then we can say it is valid.

TIME & SPACE COMPLEXITIES

Space complexity:

It is defined as the amount of memory need to run to completion by an algorithm.

The memory space needed by an algorithm has two parts

1. One is the fixed part, such as memory needed for local variables, constants, instructions etc.
2. Second one is the variable part, such as space needed for reference variables, stack space etc. They depend upon the problem instance.

The space requirement S(P) of an algorithm P can be written as S(P) = c + Sp, where c is a constant(representing fixed part). Thus while analyzing an algorithm Sp (variable part) is estimated.

Time Complexity:

The time T(P) taken by a program is the sum of compile time and run time. The compile does not depend upon the instance characteristics, hence it is fixed. Whereas run time depends on the characteristics and it is variable. Thus while analyzing an algorithm run time Tp is estimated.

Order of magnitude:

It refers to the frequency of execution of a statement. The order of magnitude of an algorithm is the sum of all frequencies of all statements. The running time of an algorithm increases with size of input and number of operations.

Ex1: In linear search, if the element to be found is in the first position, the basic operation done is one. If it is somewhere in the array, basic operation (comparison) is done n times.

If in an algorithm, the basic operations are done the same number of times every time then it is called 'Every time complexity analysis" T(n).

Ex1:

Addition of n numbers

Input : n

Alg: for 1 to n

Sum = sum+n;

T(n) = n;

Ex:2:

Matrix multiplication

Input : n

Alg: *for i = 1 to m*

for j = 1 to n

*multiplying a[i] * a[j]*

T(n) = n^2;

Worst case: w(n)

It is defined as the maximum number of times that an algorithm will ever do.

Ex: Sequential search

Inputs: n

Alg: comparison

If x is the last element, basic operations are done n times.

w(n) = n

Best case: B(n)

It is defined as the minimum number of times the algorithm will do its basic operation.

Ex: Sequential search

If x is first element B(n) = 1

Average case: A(n)

It is the average number of times an algorithm does basic operations for n.

Ex: Sequential search

Inputs: n

Case 1:

If x is in array

Let probability of x to be in Kth slot = 1/n

If x is in the Kth slot, no. of times basic operation done is K.

$$A(n) = \sum_{k=1}^{n}\left(KX\frac{1}{n}\right) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Case 2:

If x is not in array

Probability that x is in Kth slot = p/n

Probability that x is not in array = 1-P

$$A(n) = \sum_{k=1}^{n}\left(KX\frac{p}{n}\right) = n(1\text{-}P) = \frac{P}{n}\left(\frac{n(n+1)}{2}\right) + n(1\text{-}P) = n\left(1-\frac{p}{2}\right)+\frac{p}{2}$$

Order

It is the sum of all frequencies of execution of statements in a program.

1. Algorithms with time complexities of O(n) and 100n are called linear time algorithms since time complexity is linear with input size.
2. Algorithms such as $O(n^2)$, $0.01n^2$ are called quadratic time algorithms.
3. Any linear time algorithm is more efficient than a quadratic time algorithm.

Set of all complexity functions that can be classified with pure quadratic functions is called $\theta(n^2)$.

$\theta(n^2)$ is called the quadratic time algorithm.

In priori analysis, all factors regarding machine and long are ignored and only inputs and no. of operations are taken into account. For this purpose O-notation can be used.

Big O:

O(f(n)): For a function f(n), O(f(n)) is the set of complexity functions g(n) for which there exist a constant C and N such that for all $n \geq N$.

$G(n) \leq C*f(n)$

Where g(n) is computing time

n-number of inputs

O(f(n)) – algorithm complex time

If g(n) $\in$ O(f(n)) then g(n) is Big O(f(n)). It puts an asymptotic upper bound on a function.

Ex: Let $g(n) = n^2 + 10n$

$F(n) = n^2$

$G(n) \in O(f(n))$

When C=2 and N=10

For c=2

$G(n) = n^2+10n$ *and* $c\,f(n) = 2n^2$

Time complexity of $g(n) \geq c\,f(n)$

For $n \leq 10$

If $n>10$ *then* $g(n) \leq f(n)$

Thus N = 10 keeps an upper bound complexity of a 40 function.

If the algorithm is run on the same complexity on the same type of data, but with a higher magnitude of n, the resulting time is less than some constant time f(n).

Thus

$O(1) \leq O(\log n) \leq O(n) \leq O(n\log n) \leq O(n^2) \leq O(2^n)$ *for a given N.*

O(f(n)) – stands for a quantity that is not explicitly known.

Every appearance of O(f(n)) means, there are positive constants C and N such that there occurs a number x_n represented by O(f(x)) such that for all n≥N

$X_n \leq cf(n)$

Ex: $1^2+2^2+3^2+\text{----------}n^2 = \frac{1}{3}n(n+\frac{1}{2})(n+1)$

$= \frac{1}{3}n^3+\frac{1}{2}n^2+\frac{1}{6}n$

so

$1^2+2^2+3^2+\text{----------}n^2 = O(n^4)$ *may be*

$1^2+2^2+3^2+\text{----------}n^2 = O(n^3)$ *strong*

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

Theorem:

If $P(n) = a_0+a_1n+\text{----------}a_mn^m$ show that $P(n) = O(n^m)$

Proof:

$P(n) \leq |a_0| +|a_1|n+\text{----------}|a_m|n^m$

$\leq \left(|a_0|/n+|a_1|/n^{m-1}+|a_m|\right) n^m$

$\leq |a_0|+|a_1|n+\text{----------}|a_m|n^m$

when n≥1

let $C= |a_0|+|a_1|+\text{----------}|a_m|$

$P(n) \leq Cn^m$

$P(n) = O(n^m)$

The symbol O(f(n)) stands for set of all functions g of integers such that

$g(n) \leq cf(n)$

Contest of O-notation identifies the variable that is involved and the range of variables.

Ex: Show that $n^2 + 10n \in O(n^2)$

C = 11 for N=1

$n^2+10n \in O(n^2)$

Big Ω

Ωf(n)): It is the set of complexity functions g(n) for which there exists C and N for all $n \geq N$ so that $g(n) \geq cf(n)$. Always it represents the lower boundary.

Big θ

$\theta(f(n)) = O(f(n)) \cap \Omega(f(x))$

It is the set of complexity functions g(n) for which there exists some +ve real constants c, d and N such that

$c(f(x) \leq g(n) \leq d\ f(x)$

Ex: $T(n) = \frac{n(n-1)}{2} = O(n^2) = \Omega(n^2)$

$= \theta(n^2)$

If $g(n) = \theta(f(n))$ then $f(n)$ is both upper and lower bounds.

Small O: $O(f(n))$

$g(n) \sim O(f(n))$

If the exact computing time of $g(n)$ is known we can find $f(n)$, such that g is asymptotic to f.

Ex: $f(n) = a_k n^k$+ --------a_0

$= O(n^k)$

and $f(n) \sim O(a_k n^k)$