NOTES

Sub: Analysis of Algorithm &Design

Class:-IV Comp

smitasankhe@somaiya.edu

Chapter -1 Introduction to Analysis of Algorithm

- ✓ **Design and analysis fundamentals.**
- ✓ **Performance analysis, space and time complexity.**
- ✓ **Growth of function-Big –Oh, Omega, theta notation.**
- ✓ **Randomized and recursive algorithm.**

Definition of Algorithm

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

The most famous algorithm is Euclid's algorithm for calculating the greatest common divisor of two integers.

Properties:

1. It must be **correct.**
2. It must be composed of **concrete steps.**
3. There can be **no ambiguity** as to which step will be performed next.
4. It must be composed of **finite number** of steps.
5. It must be **terminate.**

Design and analysis fundamentals

Algorithmic is a branch of computer science that consists of designing and analyzing computer algorithms

1. The "design" pertain to
   i. The description of algorithm at an abstract level by means of a pseudo language, and
   ii. Proof of correctness that is, the algorithm solves the given problem in all cases.
2. The "analysis" deals with performance evaluation (complexity analysis).

We start with defining the model of computation, which is usually the Random Access Machine (*RAM*) model, but other models of computations can be use such as *PRAM*. Once the model of

computation has been defined, an algorithm can be describe using a simple language (or pseudo language) whose syntax is close to programming language such as *C* or java.

Performance Analysis

Two important ways to characterize the effectiveness of an algorithm are

1. Space complexity 2.Time complexity.

1. Space Complexity

Amount of computer memory required during the program execution, as a function of the input size

2. Time complexity

Running time of the program as a function of the size of input

Worst Case, Average Case, and Best Case

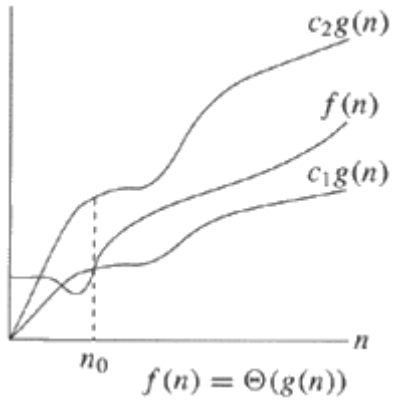
- **Worst case Running Time**: The behavior of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input.
- **Average case Running Time**: The expected behavior when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.
- **Best Case Running Time:** Minimum time required to execute.

Example :

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

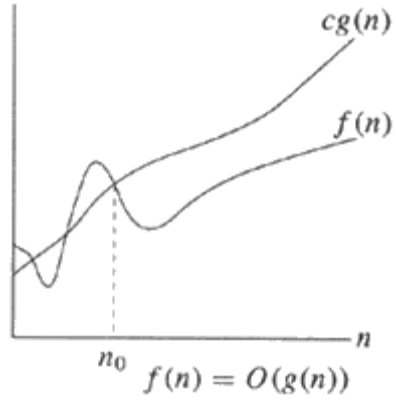
Θ-Notation (Same order)

This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.



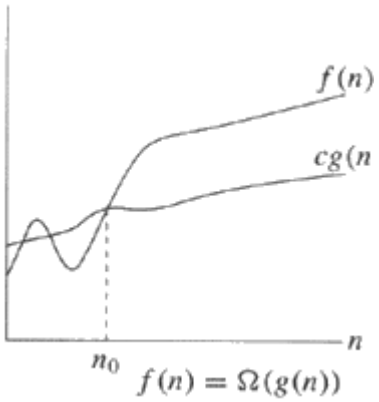
O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.



Ω-Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.



Growth of function-Big –Oh, Omega, theta notation.

Big Oh Notation (asymptotic notation)

-

A convenient way of describing the growth rate of a function and hence the time complexity of an algorithm.

Let n be the size of the input and $f(n)$, $g(n)$ be positive functions of n.

DEF.Big Oh. $f(n)$ is $O(g(n))$ if and only if there exists a real, positive constant $C$and a positive integer n_0 such that

$$f(n) \leq Cg(n) \ \forall \ n \geq n_0$$

- Note that $O(g(n))$ is a class of functions.
- The "Oh" notation specifies asymptotic upper bounds
- $O(1)$ refers to constant time. $O(n)$ indicates linear time; $O(n^k)$ (k fixed) refers to polynomial time; $O(\log n)$ is called logarithmic time; $O(2^n)$ refers to exponential time, etc.

Time complexity		Example
$O(1)$	*constant*	Adding to the front of a linked list
$O(\log N)$	*log*	Finding an entry in a sorted array
$O(N)$	*linear*	Finding an entry in an unsorted array
$O(N \log N)$	*n-log-n*	Sorting n items by 'divide-and-conque
$O(N^2)$	*quadratic*	Shortest path between two nodes in a
$O(N^3)$	*cubic*	Simultaneous linear equations
$O(2^N)$	*exponential*	The Towers of Hanoi problem

Randomized and recursive algorithm.

Recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

If a set or a function is defined recursively, then a recursive algorithm to compute its members or values mirrors the definition. Initial steps of the recursive algorithm correspond to the basis clause of the recursive definition and they identify the basis elements. They are then followed by steps corresponding to the inductive clause, which reduce the computation for an element of one generation to that of elements of the immediately preceding generation.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Randomized Algorithm

A randomized algorithm or probabilistic algorithm is an algorithm which employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

In common practice, randomized algorithms are approximated using a pseudorandom number generator in place of a true source of random bits; such an implementation may deviate from the expected theoretical behavior.

As a motivating example, consider the problem of finding an 'a' in an array of *n* elements, given that half are 'a's and the other half are 'b's. The obvious approach is to look at each element of the array, but this would take very long (n/2 operations) if the array were ordered as 'b's first followed by 'a's. There is a similar drawback with checking in the reverse order, or checking every second element. In fact, with any strategy at all in which the order in which the elements will be checked is fixed, i.e, a *deterministic* algorithm, we cannot guarantee that the algorithm

will complete quickly *for all possible inputs*. On the other hand, if we were to check array elements *at random*, then we will quickly find an 'a' *with high probability, whatever be the input*.

Randomized algorithms are particularly useful when faced with a malicious "adversary" or attacker who deliberately tries to feed a bad input to the algorithm (see worst-case-complexity and competitive analysis (online algorithm)). It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore either a source of truly random numbers or a cryptographically secure pseudo-random number generator is required. Another area in which randomness is inherent is quantum computing.

In the example above, the randomized algorithm always outputs the correct answer, but its running time is a random variable. Sometimes we want the algorithm to complete in a fixed amount of time (as a function of the input size), but allow a *small probability of error*. The former type are called Las Vegas algorithms, and the latter are Monte Carlo algorithms (related to the Monte Carlo method for simulation). Observe that any Las Vegas algorithm can be converted into a Monte Carlo algorithm (via Markov's inequality), by having it output an arbitrary, possibly incorrect answer if it fails to complete within a specified time. Conversely, if an efficient verification procedure exists to check whether an answer is correct, then a Monte Carlo algorithm can be converted into a Las Vegas algorithm by running the Monte Carlo algorithm repeatedly till a correct answer is obtained.