Analysis of Algorithms

Algorithm

Algorithm

An algorithm is a finite set of instructions which if followed accomplish a particular task. Every algorithm must satisfy the following criteria

1) Input: There are zero or more quantities which are externally supplied
2) Output: At least one quantity is produced
3) Definiteness: Each instruction must be clear & unambiguous
4) Finiteness: If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps
5) Effectiveness: Every instruction must be sufficiently basic

Analysis of Algorithms

Algorithm Analysis

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

A Priori Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

A Posteriori Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Analysis of Algorithms

Algorithm Complexity

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.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.

Analysis of Algorithms

Algorithm Complexity

Time Complexity

Usually, 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.

Analysis of Algorithms

Time Complexity

Worst Case Complexity

Let D_n be the set of inputs of size n for the problem under consideration and let I be an element of D_n. Let t(I) be the number of basic operations performed by the algorithm on input I. Then worst case complexity is

$$W(n) = \max\{t(I) / I \varepsilon D_n\}$$

Average Case Complexity

Let Pr(I) be the probability that input I occurs. Let t(I) be the number of basic operations performed by the algorithm on input I. Then average case complexity is

$$A(n) = \sum Pr(I).t(I)$$

Analysis of Algorithms

Frequency count

Consider the code

```
sum(int n)
{    sum=0;
     for (i=1;i<=n;i++)
     {
          sum=sum+i;
     }
print sum;
}
```

Analysis of Algorithms

Frequency count

Consider the code

```
sum(int n)
{    sum=0;
     for (i=1;i<=n;i++)
     {
          sum=sum+i;
     }
print sum;
}
```

- Frequency count , the number of times the statement is executed in the program.

Analysis of Algorithms

Frequency count

```
sum(int n)
{    sum=0;                       : 1 time
     for (i=1;i<=n;i++)           : n+1 times
     {
          sum=sum+i;              : n times
     }
print sum;                        : 1 time
}
```

- Total: 2n+3
- O(n)

Analysis of Algorithms

Asymptotic Notation

- Θ, O, Ω, o, ω
- Used to describe the running times of algorithms
- Instead of exact running time, say $\Theta(n^2)$
- Defined for functions whose domain is the set of natural numbers, N
- Determine *sets* of functions, in practice used to compare two functions

Intuition for Asymptotic Notation

Big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

little-oh

- f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

little-omega

- f(n) is ω(g(n)) if is asymptotically **strictly greater** than g(n)

Big-O: Common Names

- constant: $O(1)$
- logarithmic: $O(\log n)$ $(\log_k n, \log n^2 \in O(\log n))$
- linear: $O(n)$
- log-linear: $O(n \log n)$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$ (k is a constant)
- exponential: $O(c^n)$ (c is a constant > 1)

Analysis of Algorithms

- Big O notation
- Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.
- Big O can be used to describe the execution time required by an algorithm.

Analysis of Algorithms

- <u>Big O notation – formal definition</u>

Given f, $g:N \rightarrow R^+$, we say that $f(n) \in O(g(n))$ if there exists some constants $c > 0$, $n_0 \geq 0$ such that for every $n \geq n_0$, $f(n) \leq cg(n)$.

That is, for sufficiently large n, the rate of growth of *f* is bounded by *g*, up to a constant c. *f*, *g* might represent arbitrary functions, or the running time or space complexity of a program or algorithm

Analysis of Algorithms

- Big O notation

- O(1) - O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

- O(log N) - The iterative halving of data sets as in the binary search produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase . Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

Analysis of Algorithms

- Big O notation

For example, when analyzing some algorithm, one might find that the time (or the number of steps) it takes to complete, a problem of size *n* is given by $T(n)=4n^2-2n+2$.

If we ignore constants (which makes sense because those depend on the particular hardware the program is run on) and slower growing terms,we could say *"T(n) grows at the order of* n^2*"* and *write:* $T(n)=O(n^2)$.

Analysis of Algorithms

Big O notation

$O(n^c)$ and $O(c^n)$ are very different. The latter grows much,much faster, no matter how big the constant c is.

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

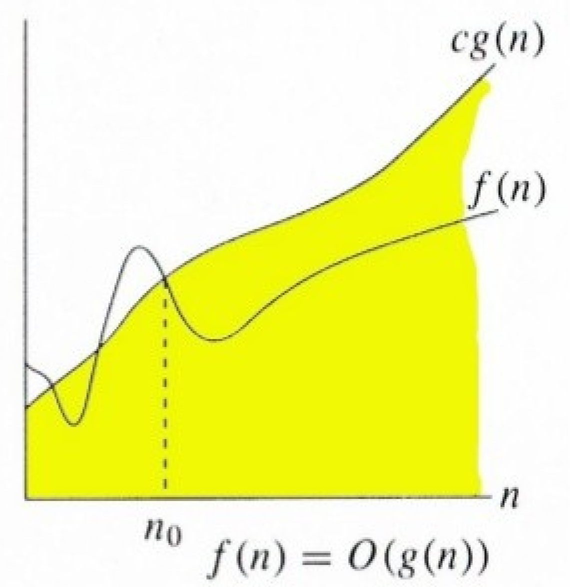
O-notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$0 \leq f(n) \leq cg(n)$,

for all $n \geq n_0\}$



We say $g(n)$ **is an** ***asymptotic upper bound*** **for** $f(n)$

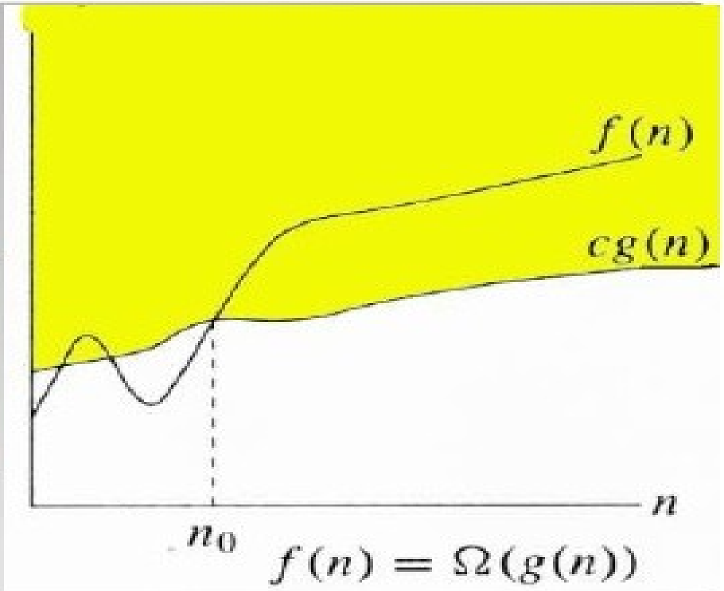
Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$\Omega(g(n)) = \{f(n)$:
$\exists$ **positive constants c and n_0, such that $\forall n \geq n_0$,**
we have $0 \leq cg(n) \leq f(n)\}$



Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$.
$\Theta(g(n)) \subset \Omega(g(n))$.

Theta Notation, θ

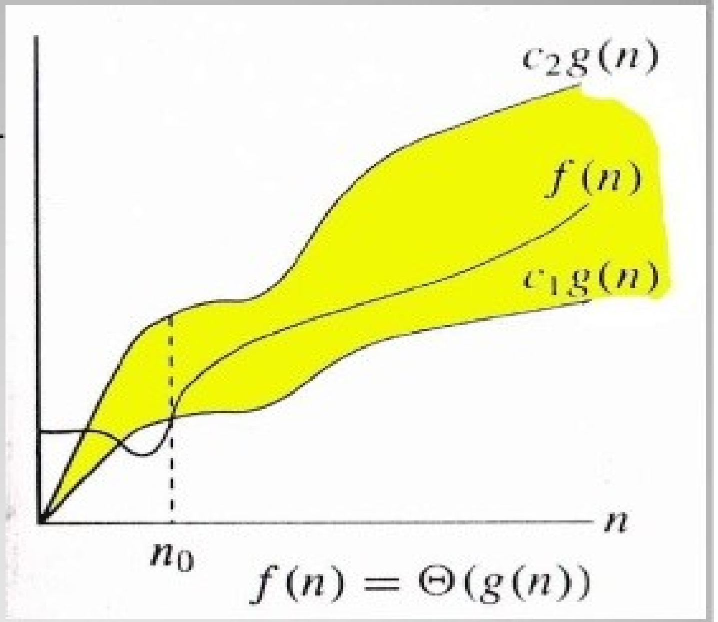
The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$:
∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\}$

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.



$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Analysis of Algorithms

<u>Little-o notation</u>

$f(n) = o(g(n))$ means for all $c > 0$ there exists some $k > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge k$.

<u>Little- ω notation</u>

$f(n) = \omega\ (g(n))$ means that for any positive constant c, there exists a constant k, such that $0 \le cg(n) < f(n)$ for all $n \ge k$.

Analysis of Algorithms

Thank You