Algorithm Analysis

Unit-1

Why performance analysis?

- There are many important things related to any sysytem that should be taken care of, like user friendliness, modularity, security, maintainability, etc.
- Why to worry about performance?
- The answer to this is simple, we can have all the above things only if we have performance.
- So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

How to measure performance

- Time Complexity

- Space Complexity

Given two algorithms for a task, how do we find out which one is better?

- One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.

2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis

- Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms.

- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time).

- We calculate, how does the time (or space) taken by an algorithm increases with the input size.

Example

- For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic).

- To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer.

- For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine.

- The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

We can have three cases to analyze an algorithm:

1) Worst Case
2) Average Case
3) Best Case

Implementation of Linear Search.

```
#include <stdio.h>


// Linearly search x in arr[].  If x is present then
return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
       if (arr[i] == x)
         return i;
    }
    return -1;
}
```

```
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr);
    printf("%d is present at index %d", x,
search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- We must know the case that causes maximum number of operations to be executed.
- For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array.
- When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

Average Case Analysis (Sometimes done)

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.

- Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases.

- For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

$$\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta((n+1)*(n+2)/2)}{(n+1)}$$

$$= \Theta(n)$$

Best Case Analysis

- In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.

- In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$

- Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

Asymptotic Notations

- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.

- There are 3 asymptotic notations used to represent time complexity of algorithms.

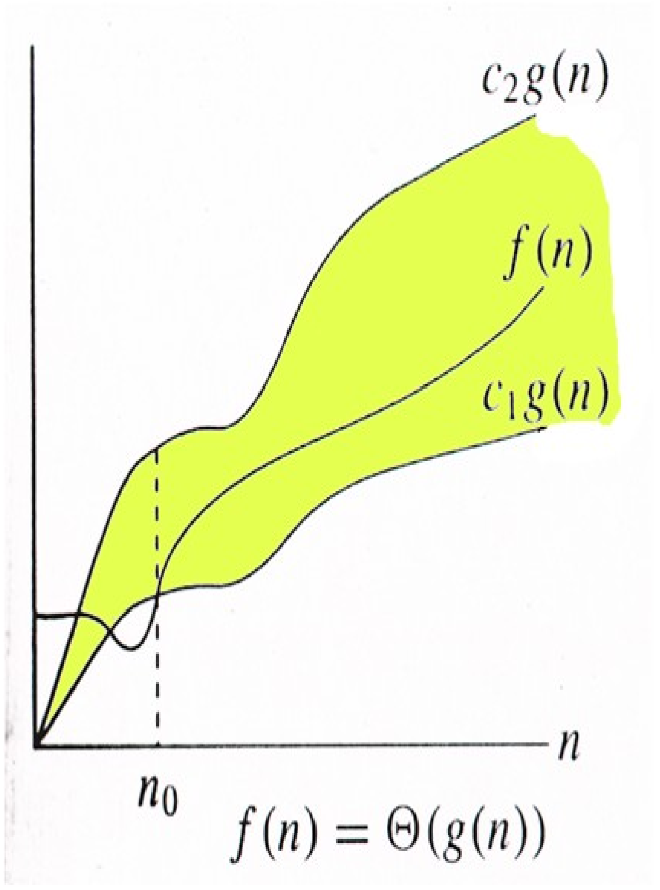
Θ Notation

- The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.

- For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

- The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



Big O Notation

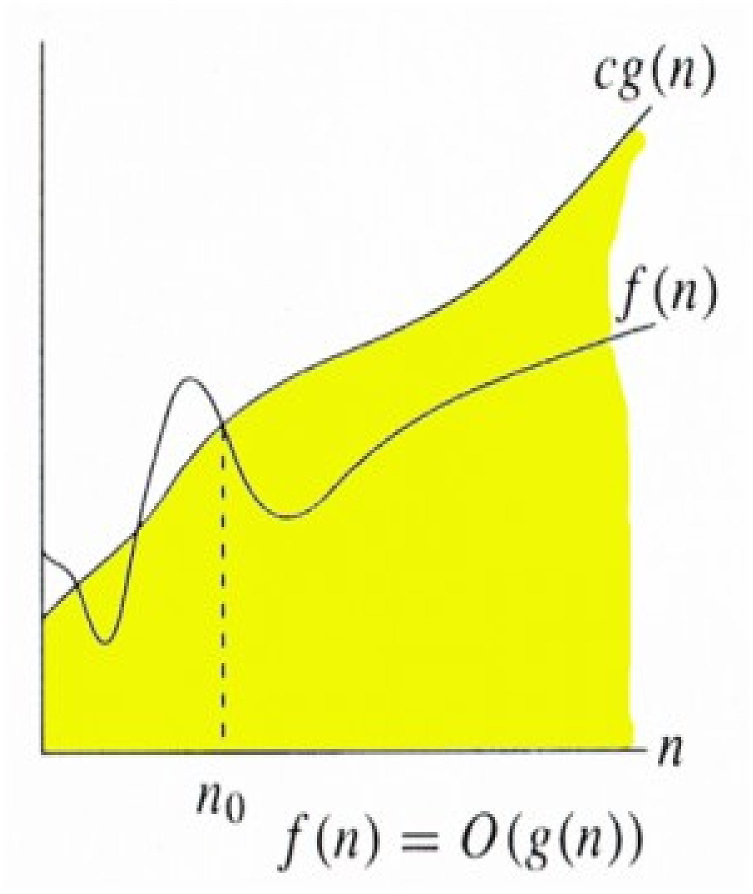
- The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.

- The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and

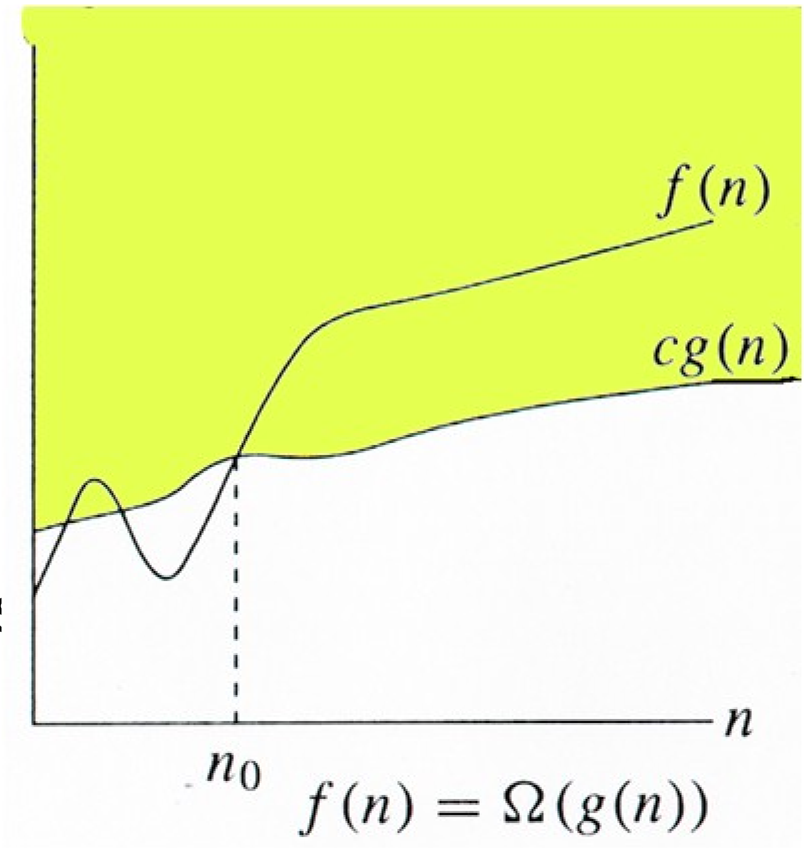
n0 such that 0 <= f(n) <= cg(n) for

all n >= n0}



Ω Notation

- Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- For a given function g(n), we denote by Ω(g(n)) the set of functions.



Ω (g(n)) = {f(n): there exist positive constant
and

n0 such that 0 <= cg(n) <= f(n)

for

all n >= n0}.

Simple examples for analysizing the algorithm

O(1)

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
   // set of non-recursive and non-loop statements
```

- For example swap() function has O(1) time complexity.
- A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
  for (int i = 1; i <= c; i++) {
      // some O(1) expressions
  }
```

O(n)

Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```