Unit – II

Mathematical Aspects and Analysis of Algorithms

Asymptotic Notations

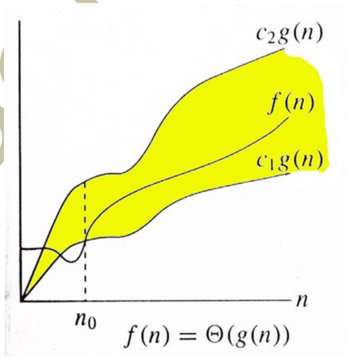
The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1) Θ Notation:

The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta n^2)$ irrespective of the constants involved. 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.

2) Big O Notation:

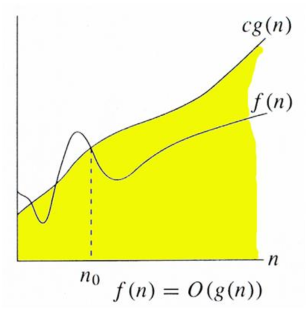
The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

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}

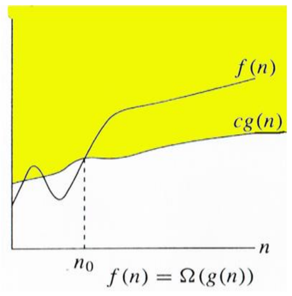


3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

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

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that

0 <= cg(n) <= f(n) for all n >= n0}.



Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Mathematical Analysis of Recursive Algorithms

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$. There are many other algorithms like Binary Search, Tower of Hanoi, etc.
There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.
For example consider the recurrence $T(n) = 2T(n/2) + n$
We guess the solution as $T(n) = O(nLogn)$. Now we use induction to prove our guess.
We need to prove that $T(n) <= cnLogn$. We can assume that it is true for values smaller than n.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &<= cn/2Log(n/2) + n \\ &<= cnLogn - cnLog2 + n \\ &<= cnLogn - cn + n \\ &<= cnLogn \end{aligned}$$

2) Recurrence Tree Method:
In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series. For example consider the recurrence relation
$T(n) = T(n/4) + T(n/2) + cn^2$

```
          cn^2
        /      \
    T(n/4)     T(n/2)
```

If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.

```
              cn^2
          /          \
      c(n^2)/16      c(n^2)/4
       /     \        /     \
  T(n/16)     T(n/8)  T(n/8)    T(n/4)
```

Breaking down further gives us following

```
               cn^2
           /            \
      c(n^2)/16          c(n^2)/4
       /      \           /      \
c(n^2)/256  c(n^2)/64  c(n^2)/64    c(n^2)/16
 /    \      /    \    /    \       /    \
```

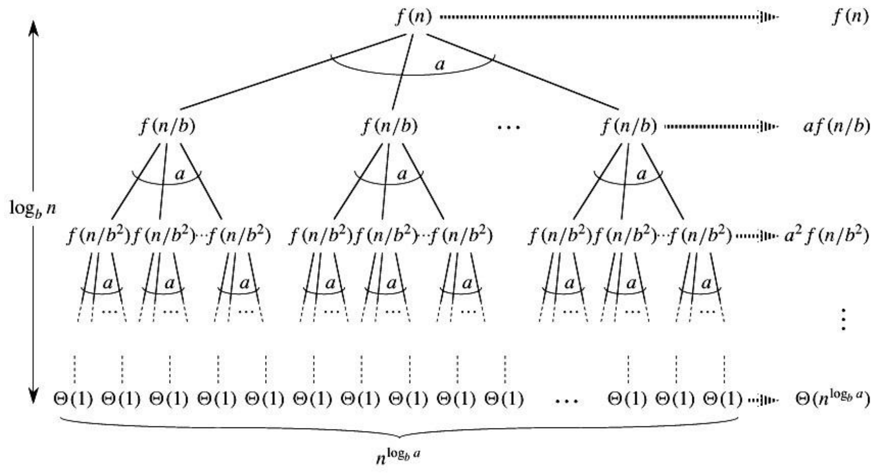
To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/16. To get an upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

3) Master Method:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
3. If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?
Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and $\text{Log}_b a$] is also 1. So the solution is Θ(n Logn)
Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and $\text{Log}_b a$ is also 0. So the solution is Θ(Logn)

Notes:

1) It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/\text{Log}n$ cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c \text{Log}^k n)$

If $f(n) = \Theta(n^c \text{Log}^k n)$ for some constant $k >= 0$ and $c = \text{Log}_b a$, then $T(n) = \Theta(n^c \text{Log}^{k+1} n)$

Practice Problems and Solutions on Master Theorem.(Refer Notes)

References:

http://en.wikipedia.org/wiki/Master_theorem
MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Mathematical Analysis of Nonrecursive Algorithms

The core of the algorithm analysis: to find out how the number of the basic operations depends on the size of the input.

There are four rules to count the operations:

Rule 1: for loops - the size of the loop times the running time of the body

The running time of a for loop is at most the running time of the statements inside the loop times the number of iterations.

```
        for( i = 0; i < n; i++)
                sum = sum + i;
```

a. Find the running time of statements when executed only once:

The statements in the loop heading have fixed number of operations, hence they have constant running time O(1) when executed only once. The statement in the loop body has fixed number of operations, hence it has a constant running time when executed only once.

b. Find how many times each statement is executed.

```
 for( i = 0; i < n; i++)          // i = 0; executed only once: O(1)
                                  // i < n; n + 1 times O(n)
                                 // i++ n times O(n)
                                 // total time of the loop heading:
                                 // O(1) + O(n) + O(n) = O(n)
        sum = sum + i;           // executed n times, O(n)
```

The loop heading plus the loop body will give: O(n) + O(n) = O(n).

Loop running time is: O(n)

Mathematical analysis of how many times the statements in the body are executed

$$C(n) = \sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

If

a. the size of the loop is n (loop variable runs from 0, or some fixed constant, to n) and

b. the body has constant running time (no nested loops)

then the time is O(n)

Rule 2: Nested loops – the product of the size of the loops times the running time of the body

The total running time is the running time of the inside statements times the product of the sizes of all the loops

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < n; j++)
                sum++;
```

Applying Rule 1 for the nested loop (the 'j' loop) we get O(n) for the body of the outer loop. The outer loop runs n times, therefore the total time for the nested loops will be $O(n) * O(n) = O(n*n) = O(n^2)$

Mathematical analysis:

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

What happens if the inner loop does not start from 0?

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = i; j < n; j++)
                sum++;
```

Here, the number of the times the inner loop is executed depends on the value of i

i = 0, inner loop runs n times

i = 1, inner loop runs (n-1) times

i = 2, inner loop runs (n-2) times

…

i = n – 2, inner loop runs 2 times

i = n – 1, inner loop runs once.

Thus we get: (1 + 2 + … + n) = $n*(n+1)/2 = O(n^2)$

General rule for nested loops:

Running time is the product of the size of the loops times the running time of the body.

Example:

```
sum = 0;
for( i = 0; i < n; i++)
         for( j = 0; j < 2n; j++)
                sum++;
```

We have one operation inside the loops, and the product of the sizes is $2n^2$

Hence the running time is $O(2n^2) = O(n^2)$

Note: if the body contains a function call, its running time has to be taken into consideration.

```
sum = 0;
        for( i = 0; i < n; i++)
                for( j = 0; j < n; j++)
                        sum = sum + function(sum);
```

Assume that the running time of function(sum) is known to be log(n).
Then the total running time will be $O(n^2*\log(n))$

Rule 3: Consecutive program fragments
The total running time is the maximum of the running time of the individual fragments

```
sum = 0;
        for( i = 0; i < n; i++)
                sum = sum + i;
sum = 0;
for( i = 0; i < n; i++)
         for( j = 0; j < 2*n; j++)
                sum++;
```

The first loop runs in O(n) time, the second - $O(n^2)$ time, the maximum is $O(n^2)$

Rule 4: If statement

```
if C
        S1;
else
        S2;
```

The running time is the maximum of the running times of S1 and S2.
Summary
Steps in analysis of nonrecursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of time the basic operation is executed depends on some additional property of the input. If so, determine worst, average, and best case for input of size n
- Count the number of operations using the rules above.

Exercise
a.

```
sum = 0;
for( i = 0; i < n; i++)
for( j = 0; j < n * n; j++)
sum++;
```

b.

```
sum = 0;
for( i = 0; i < n; i++)
for( j = 0; j < i; j++)
sum++;
```

c.

```
sum = 0;
for( i = 0; i < n; i++)
```

```
        for( j = 0; j < i*i; j++)
        for( k = 0; k < j; k++)
        sum++;
d.
        sum = 0;
        for( i = 0; i < n; i++)
        sum++;
        val = 1;
        for( j = 0; j < n*n; j++)
        val = val * j;
e.
        sum = 0;
        for( i = 0; i < n; i++)
        sum++;
        for( j = 0; j < n*n; j++)
        compute_val(sum,j);
```

The complexity of the function compute_val(x,y) is given to be O(nlogn)

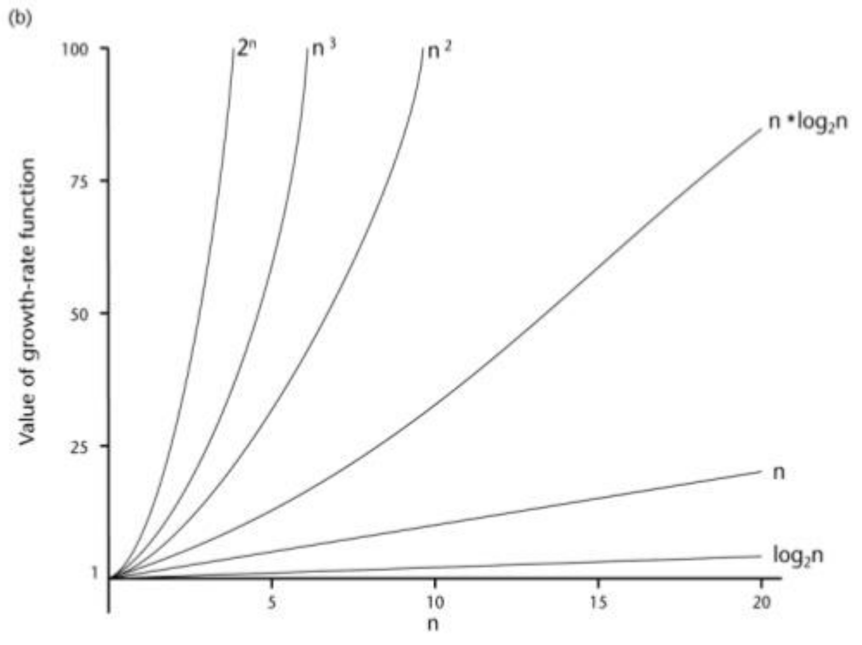
Solutions:

(a) $O(n^3)$
(b) $O(n^2)$
(c) $O(n^5)$
(d) $O(n^2)$
(e)$O(n^3\log(n)))$

Order of Growth Functions

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

A Comparison of Growth-Rate Functions (cont.)


(b)
Value of growth-rate function
n
2ⁿ
n³
n²
n *log₂n
n
log₂n