Design and Analysis of Algorithms (18CS42)

MODULE – I

- Introduction to Design and analysis of algorithms
- Growth of Functions (Asymptotic notations)
- Recurrences, Solution of Recurrences by substitution
- Recursion tree method
- Master Method

Module 1 - Introduction to Design and analysis of algorithms

What is an algorithm?

Algorithm is a *set of steps to complete a task.*

For example,

Task: to make a cup of tea.

Algorithm:

- add water and milk to the kettle,
- boilit, add tea leaves,
- Add sugar, and then serve it in cup.

What is *Computer algorithm*?

''a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it''.

Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.

These algorithmsrun on computers or computational devices.Forexample, GPS in our smartphones, Google hangouts.

GPS uses *shortest path algorithm.* Online shopping uses cryptography which uses RSA algorithm.

Characteristics of an algorithm:-

- Must take an input.
- Must give some output (yes/no,valueetc.)
- Definiteness –each instruction is clear and unambiguous.
- Finiteness –algorithm terminates after a finite number of steps.
- Effectiveness –every instruction must be basic i.e. simple instruction.

Expectation from an algorithm

- Correctness:-

  Correct: Algorithms must produce correct result.

Produce an incorrect answer:Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. Eg.Rabin-Miller PrimalityTest
(Used in RSA algorithm): It doesn't give correct answer all the time.1 out of 2^{50} times it gives incorrect result.

Approximation algorithm: Exact solution is not found, but near optimal solution can be found out. (Applied to optimization problem.)

- Less resource usage:

  Algorithms should use less resources (time and space).

Resource usage:

Here, the time is considered to be the primary measure of efficiency .We are also concerned with how much the respective algorithm involves the computer memory.But mostly time is the resource that is dealt with. And the actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into: 1.Memory (space) 2.Time

Time taken by an algorithm?

performance measurement or Apostoriori Analysis: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.

Performance Evaluation or Apriori Analysis. Before implementing the algorithm in a system. This is done as follows

1. How long the algorithm takes :-will be represented as a function of the size of the input.

   f(n)→how long it takes if 'n' is the size of input.

2. How fast the function that characterizes the running time grows with the input size.

   "Rate of growth of running time".
   The algorithm with less rate of growth of running time is considered better.

How algorithm is a technology ?

Algorithms are just like a technology. We all use latest and greatest processors but we need to run implementations of good algorithms on that computer in order to properly take benefits of our money that we spent to have the latest processor. Let's make this example more concrete by pitting a faster computer(computer A) running a sorting algorithm whose running time on n values grows like n^2 against a slower computer (computer B) running asorting algorithm whose running time grows like n lg n. They eachmust sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than anysingle sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is1000 times faster than computer B in raw computing power. To makethe difference even more dramatic, suppose that the world's craftiestprogrammer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose furtherthat just an average programmer writes for computer B, using a high-level language with an inefficient compiler, with the resulting code taking 50n lg n instructions.

Growth of Functions (Asymptotic notations)

Before going for growth of functions and asymptotic notation let us see how to analyase an algorithm.

How to analyse an Algorithm

Let us form an algorithm for Insertion sort (which sort a sequence of numbers).The pseudo code for the algorithm is give below.

Let C_i be the cost of i^{th} line. Since comment lines will not incur any cost $C_3=0$.

Cost	No. Of times Executed
C_1	n
C_2	n-1
$C_3=0$	n-1
C_4	n-1
C_5	$\sum_{j=2}^{n} t_j$
C_6	$\sum_{j=2}^{n} (t_j-1)$
C_7	$\sum_{j=2}^{n} (t_j-1)$
C_8	n-1

Running time of the algorithm is:

$T(n)=C_1n+C_2(n-1)+0(n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n} t_j)+C_6(\sum_{j=2}^{n} t_j - 1)+C_7(\sum_{j=2}^{n} t_j - 1)+ C_8(n-1)$

Best case:

It occurs when Array is sorted.

Worst case:

It occurs when Array is reverse sorted, and $t_j = j$

$T(n)=C_1n+C_2(n-1)+0 (n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n} j)+C_6(\sum_{j=2}^{n} j-1)+C_7(\sum_{j=2}^{n} j-1)+ C_8(n-1)$

$=C_1n+C_2(n-1)+C_4(n-1)+C_5\left(\frac{n(n+1)}{2}-1\right)+C_6\left(\sum_{j=2}^{n}\frac{(j-1)(j+1)}{2}\right)+C_7\left(\sum_{j=2}^{n}\frac{(j-1)(j+1)}{2}\right)+ C_8(n-1)$

which is of the form an^2+bn+c

Quadratic function. So in worst case insertion set grows in n^2.

Why we concentrate on worst-case running time?

- The worst-case running time gives a guaranteed upper bound on the runningtime for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it's often about as bad as the worst case.

Order of growth:

It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

Example: We found out that for insertion sort the worst-case running time is of the form $an^2 + bn + c$.

Drop lower-order terms. What remains is an^2.Ignore constant coefficient. It results in n^2.But we

cannot say that the worst-case running time T(n) equals n^2 .Rather It grows like n^2 . But it doesn't equal n^2.We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Asymptotic notation

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.

- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

$O \approx \leq$

$\Omega \approx \geq$

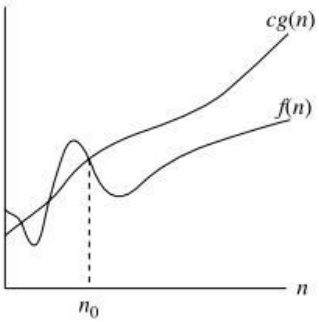
$\Theta \approx =$

$o \approx <$

$\omega \approx >$

O-notation

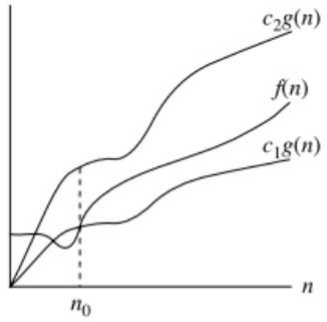
$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.



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

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



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

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

***o*-notation**

$$o(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} .$$

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

***ω*-notation**

$$\omega(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} .$$

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

Recurrences, Solution of Recurrences by substitution, Recursion Tree and Master Method

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

• If the given instance of the problem is small or simple enough, just solve it.

• Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g.the worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n>1 \end{cases}$$

whose solution can be found as $T(n)=\Theta(n \log n)$

There are various techniques to solve recurrences.

1. SUBSTITUTION METHOD:

The substitution method comprises of 3 steps

i. Guess the form of the solution
ii. Verify by induction
iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name “subst itution method”. This method is powerful, but we must be able to guess the form of the answer in order to apply it.

e.g.recurrence equation: T(n)=4T(n/2)+n

step 1: guess the form of solution

$T(n)=4T(n/2)$

$F(n)=4f(n/2)$

$F(2n)=4f(n)$

$F(n)=n^2$

So, T(n) is order of n^2 Guess $T(n)=O(n^3)$

Step 2: verify the induction

Assume $T(k)<=ck^3$

$T(n)=4T(n/2)+n$

$<=4c(n/2)^3+n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 –n)$ is always positive

So what we assumed was true.

$T(n)=O(n^3)$

Step 3: solve for constants

$Cn^3/2-n>=0$

$n>=1$

$c>=2$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound

Assume,$T(k)<=ck^2$

so,we should prove that $T(n)<=cn^2$

$T(n)=4T(n/2)+n$

$4c(n/2)^2+$

$n\ cn^2+n$

So,T(n) will never be less than cn^2. But if we will take the assumption of $T(k)=c_1\ k^2-c_2k$, then we can find that $T(n) = O(n^2)$

2. BY ITERATIVE METHOD:

e.g. $T(n)=2T(n/2)+n$

$=> 2[2T(n/4) + n/2]+n$

$=>2^2T(n/4)+n+n$

$=> 2^2[2T(n/8)+ n/4]+2n$

$=>2^3T(n/2^{3)} +3n$

After k iterations ----------

$,T(n)=2^kT(n/2^k)+kn$ (1)

Sub problem size is 1 after $n/2^k=1$ => k=logn

So,afterlogn iterations ,the sub-problem size will be 1.

So,when k=logn is put in equation 1

T(n)=nT(1)+nlogn

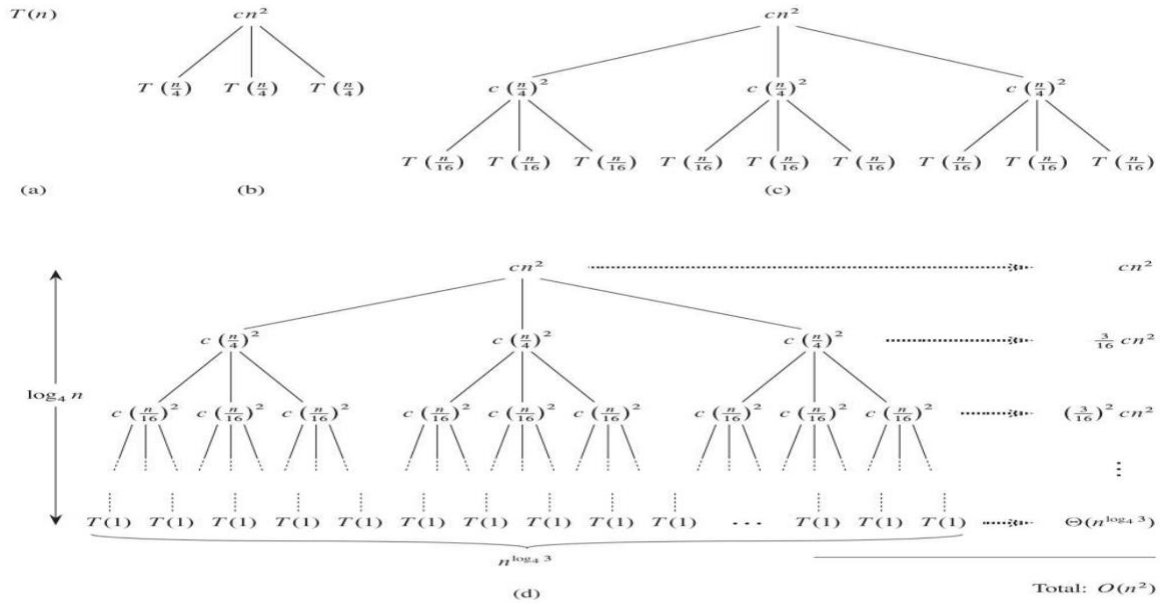
nc+nlogn (say c=T(1))

O(nlogn)

3. BY RECURSSION TREE METHOD:

In a recursion tree ,each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost,and then we sum all the per level cost to determine the total cost of all levels of recursion .

Constructing a recursion tree for the recurrence $T(n)=3T(n/4)+cn^2$



Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn^2.. Part (a) shows T (n), which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part

(d) has height $\log_4 n$ (it has $\log_4 n + 1$

levels). Sub problem size at depth i =$n/4^i$

Sub problem size is 1 when $n/4^i=1$ => $i=\log_4 n$

So, no. of levels =$1+ \log_4 n$

Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth i=3^i

Cost of each node at depth i=$c\ (n/4^i)^2$

Cost of each level at depth i=$3^i\ c\ (n/4^i)^2 = (3/16)^i cn^2$

T(n)= $\sum_{i=0}^{\log_4 n} cn^2(3/16)^i$

T(n)= $\sum_{i=0}^{\log_4 n\ -1} cn^2(3/16)^i$ + cost of last level

Cost of nodes in last level =$3^i T(1)$

logn (at last level
c3 4 i=$\log_4 n$)
log3
cn 4

T(n) = $\sum_{i=0}^{\log_4 n\ -1}$ [illegible] + c n $^{\log 3}_{4}$

<= $cn^2 \sum_{i=0}^{\log_4 n\ -1}$ [illegible]

log
2*(16/13)+ 3
<= cncn 4 => T(n)=O(n) 2

4. BY MASTER METHOD:

The master method solves recurrences of the form

T(n)=aT(n/b)+f(n)

where a>=1 and b>1 are constants and f(n) is a asymptotically positive function .

To use the master method, we have to remember 3 cases:

1. If $f(n)=O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon > 0$, then $T(n)=\Theta(n^{\log_b a})$

2. If $f(n)=\Theta(n^{\log_b a})$ then $T(n)=\Theta(n^{\log_b a}\log n)$

3. If $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$, and if $a*f(n/b)<=c*f(n)$ for some constant $c<1$ and all sufficiently large n, then $T(n)=\Theta(f(n))$

e.g. $T(n)=2T(n/2)+n\log n$

ans: $a=2$ $b=2$

$f(n)=n\log n$

using 2nd formula

$f(n)=\Theta(n^{\log_2 2}\log^k n)$

$=>\Theta(n^1 \log^k n)=n\log n$ $=>K=1$

$T(n)=\Theta(n^{\log_2 2}\log^1 n)$

$=>\Theta(n\log^2 n)$