Note Junction

Best Note Provider

UNIT=1
Foundations of Algorithm Analysis:

⊛ Algorithm and its properties:

An algorithm is a finite set of instructions where each instructions can be executed in finite time to perform computation by taking some value as input and to produce value(s) as output.

Characteristics/Properties:

i) **Input/Output:** Algorithm must take input value(s) and it must produce output value(s).

ii) **Correctness:** It should produce the output according to the requirement of the algorithm.

iii) **Finiteness:** Algorithm must complete after a finite number of instructions have been executed.

iv) **Feasibility:** It must be feasible to execute each instruction.

v) **Flexibility:** It should also be possible to make changes in the algorithm without putting so much effort on it.

vi) **Efficient:** The algorithm should use lesser running time and memory space as much as possible.

⊛ RAM model: [Imp]

Random Access Machine (RAM) model is a model for counting the steps in algorithm in order to analyze the complexity. In this model we count:

- Basic operations (+, −, *, /) as 1 step.
- Memory refrence (read & write) as 1 step.
- Loops, function calls are not basic operations. Hence not counted.

$$Avg = \frac{a+b}{2}$$

Example: Algorithm to find the factorial of given numbers.

```
factorial (int n){
    if (n<0)
        fact=1
        for(i=1; i<n; i++)
            fact = fact * i;
    } return fact;
```

Step count according to RAM model:
$7n+3$

⊛. Time and Space Complexity:- [Imp]

Time complexity of an algorithm is the total time it takes to solve a problem. Time complexity is measured in terms of no. of computational steps.

Space complexity of an algorithm is the total memory space it takes to store data required to solve a problem. The space complexity is measured in terms of number of data variables used during the computation.

Best case complexity: It gives lower bound on the running time of the algorithm for any instance of input (s). This indicates that the algorithm can never have lower running time than best case for particular class of problems.

Worst case complexity: It gives upper bound on the running time of the algorithm for all the instances of input (s). This indicates that no input can overcome the running time limit posed by worst case complexity.

Average case complexity: It gives average number of steps required on any instance of the input(s).

⊛ Detailed Analysis of Algorithms:-

(no need to remember each line just understand. Writing this is not asked in exam but concept is important)

1) Time Complexity: (Analysis)

→ Time complexity of simple operations that takes 1 step time like assignment (e.g. i=0), addition (e.g. a=b+c), simple statements like printf, scanf, return etc. take very small constant time, which does not affect time complexity of our algorithm much so we can neglect them.

(Space or time complexity analyze गर्दा ठूलो term हेरेर [illegible] complexity निकाल्छ i.e, worst case)

→ We mainly analyze time complexity of algorithms based on the loops like for loop, while loop etc. We may have many condition in this case some of the simple cases are as follows:

i) If loop is like for(i=0; i<=n; i++) i.e, loop is simply running from 0 to n and incrementing simply by 1. In this case time complexity will be O(n).

ii) For nested loops e.g. two loops running simply as in (i) which are nested. In this case time complexity will be product of time complexity of each loop.

for e.g.
```
for(i=0;i<=n;i++)          —— O(n)
{
  for(j=0;j<=n;j++)        —— O(n)
    printf("Hello");
}
```

Time complexity $= O(n) \times O(n) = O(n^2)$.

iii) For loops like for(i=0; i<=n; i*5) i.e, incrementing by multiplication. In this case time complexity $= O(\log_{\text{constant multiplier}} n) = O \log_5 n$.

2) Space Complexity: (Analysis)

Space complexity is the total memory refrences used by the algorithm.

→ If total memory refrences used by the algorithm is constant like 1, 2, 3, 4, 5, etc. then the space complexity will be O(1).

→ If Array is taking n memory refrences then the space complexity will be O(n).

Example:- Find detailed analysis of following factorial algorithm.

```
#include <stdio.h>
int main()
{
    int i, n, fact=1;
    printf("Enter a number to calculate its factorial \n");
    scanf("%d", &n);
    for (i=1; i<=n; i++)
        fact = fact*i;
    printf("Factorial of %d = %d \n", n, fact);
    return 0;
}
```

Time Complexity Analysis

The declaration statement takes 1 step time
Printf statement takes 1 step time.
Scanf statement takes 1 step time.
In for loop
- i=1 takes 1 step.
- i<=n takes (n+1) step
- i++ takes n step.
- fact = fact*i takes n step

Printf statement takes 1 step.
return statement takes 1 step.

⇒So Total time complexity = 1+1+1+1+n+1+n+n+1+1
= 2n+7
= O(1) × O(n) + O(1) ← Since time complexity of constant is O(1)
= O(n) + O(1) ← Smaller terms are always neglected when there are higher ones like n, n^2 etc.
= O(n).

Space Complexity Analysis

Total memory refrences used = 3 ← i.e. Memory space is needed 1 for i, 1 for n and 1 for fact.

Hence, Space complexity = O(1) ← for constants O(1)

⊛. Aggregate analysis:-

It determines the upper bound $T(n)$ on the total cost of a sequence of n operations, then calculates the average cost to be $T(n)/n$. In aggregate analysis, there are two steps. First we must show that a sequence of n operations takes $T(n)$ time in the worst case. Then, we show that each operation takes $T(n)/n$ time on average. Therefore, in aggregate analysis, each operation has the same cost.

Example:- A common example of aggregate analyers is a modified stack. In stack Push(element) puts an element on the top of the stack, and Pop(element) takes the top element off of the stack and returns it. The operations are both constant-time, so a total of n operations will result in $O(n)$ time. So, using aggregate analysis,

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1).$$

⊛. Asymptotic Notations: [Imp]

Complexity analysis of an algorithm is done in terms of bound (upper bound & lower bound). For this purpose we need the concept of asymptotic notations.

Big Oh (O) notation: When we have only asymptotic upper bound then we use O notation. Mathematically, a function $f(x)$ is said to be Big Oh of another function $g(x)$ [i.e, $f(x) = O\,g(x)$], iff there exist two constants x_0 and c such that

$$f(x) \leq c * g(x) \quad \forall x \geq x_0.$$

When $f(x) = O\,g(x)$ then we say that $g(x)$ is the upper bound of $f(x)$.

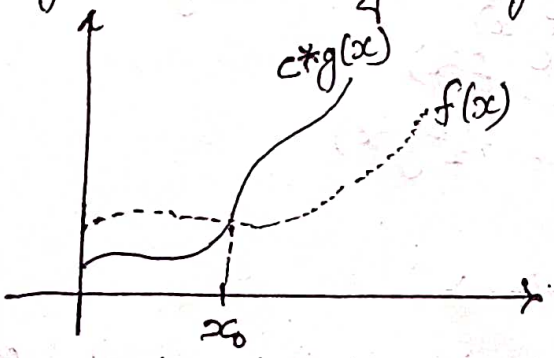


Fig: Geometrical interpretation of $f(x) = O\,g(x)$.

Example:- Find big oh of given function $f(n) = 3n^2 + 4n + 7$

Solution:- we have, $f(n) = 3n^2 + 4n + 7 \leq 3n^2 + 4n^2 + 7n^2 \leq 14n^2$

$$\Rightarrow f(n) \leq 14n^2$$

where, $c = 14$ and $g(n) = n^2$, thus $f(n) = O(g(n)) = O(n^2)$.

ii) Big Omega (Ω) notation:- Big omega notation gives asymptotic lower bound. If f and g are any two functions from set of integers to set of integers, the function f(x) is said to be big omega of g(x) i.e, $f(x) = \Omega(g(x))$ if and only if there exists two positive constants c and x_0 such that

For all $x >= x_0$, $f(x) >= c * g(x)$.

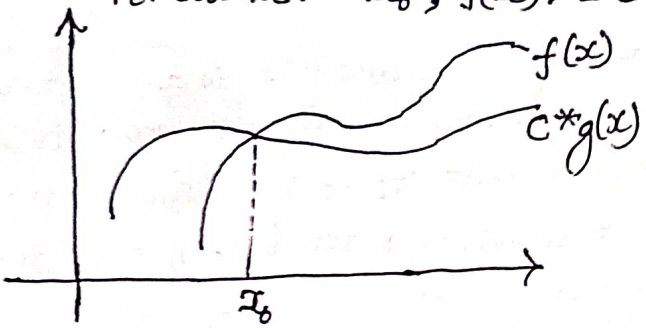


Fig: Geometric intrepretation of Big Omega notation.

When $f(x) = \Omega g(x)$ then we say that g(x) is the lower bound of f(x).

Example:- Find big omega of $f(n) = 3n^2 + 4n + 7$

Solution:

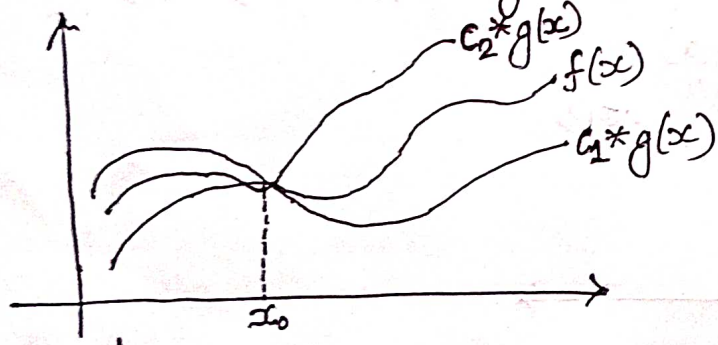
Since we have $f(n) = 3n^2 + 4n + 7 >= 3n^2$

$\Rightarrow f(n) >= 3n^2$

where, $c = 3$ and $g(n) = n^2$, thus $f(n) = \Omega(g(n)) = \Omega(n^2)$.

iii) Big Theta (Θ) notation:-

When we need asymptotically tight bound then we use this notation. If f and g are any two functions from set of integers to set of integers then function f(x) is said to be big theta of g(x) [i.e, $f(x) = \Theta(g(x))$] if and only if there exists three positive constants c1, c2 and x_0 such that for all $x >= x_0$, $c1 * g(x) <= f(x) <= c2 * g(x)$.



Example:- If $f(n) = 3n^2 + 4n + 7$ $g(n) = n^2$, then prove that $f(n) = \Theta(g(n))$.

Proof: Let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have, $f(n) <= c_1 * g(n)$, $n >= n_0$ as $3n^2 + 4n + 7 <= 14 * n^2$

and $f(n) >= c_2 * g(n)$, $n >= n_0$ as $3n^2 + 4n + 7 >= 1 * n^2$.

For all $n >= 1$ (in both cases).

So, $c_2 * g(n) <= f(n) <= c_1 * g(n)$ is trivial.

Hence, $f(n) = \Theta(g(n))$

⊛. Recursive Algorithms:

Recursion is the process of defining a problem in terms of itself. It is the process in which a function calls itself directly or indirectly. Using recursive algorithm complex problems are divided into smaller problems and solved then the solution of smaller problems are combined to get the solution of original problem. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), tree traversals, DFS of graph etc. We need base case for our recursive algorithm.

Example: Recursive algorithm for calculating factorial:

```
int fact (int n)
{ if (n<=1) //base case
    return 1;
  else
    return n*fact (n-1);
}
```

⊛. Recurrence Relations:-

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a recurrence relation means to obtain a function defined on the natural numbers that satisfy the recurrence. To solve recursive algorithms we need to define their recurrence relation and by using any one of the recurrence relation solving method we calculate their complexity.

Example: Recursive algorithm for finding factorial.

$T(n) = 1$ when $n = 1$

$T(n) = T(n-1) + O(1)$ when $n > 1$.

⊛. Solving Recurrences: [Imp]

Numericals are important. Recursion tree & Master method are more important

The process of finding solution of given recurrence relation in terms of big oh notation is called solving recurrences. There are a lot of methods for solving recurrence relations some of the popular methods are: Iteration method, Recursion Tree, Substitution and Master method.

Some Important formulas that may be using while solving Numericals Related to Recurrences. These are Mathematical Foundations:

1) Exponents:

i) $x^a x^b = x^{a+b}$

ii) $x^a / x^b = x^{a-b}$

iii) $(x^a)^b = x^{ab}$

iv) $x^n + x^n = 2x^n$

v) $2^n + 2^n = 2^{n+1}$

2) Series:

i) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

ii) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

iii) $\sum_{i=0}^{n} a^i \leq \frac{1}{1-a}$; if $0 < a < 1$.

iv) $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

3) Logarithms:-

i) $\log_a b = \log_c b / \log_c a$; $c > 0$ [making base same].

ii) $\log ab = \log a + \log b$.

iii) $\log a/b = \log a - \log b$.

iv) $\log (a^b) = b \log a$

v) $\log x < x$ for all $x > 0$.

vi) $\log 1 = 0$, $\log 2 = 1$, $\log 1024 = 10$.

vii) $a^{\log_b n} = n^{\log_b a}$

1) Iteration method:

Here we expand the given relation until the boundry is not met. Expand the relation so that summation independent on n is obtained. It uses an initial guess to generate a sequence of improving approximate solutions for a class of problems, in which the nth approximation is derived from the previous ones.

Example: Solve following recurrence relation by using iterative method.

$T(n) = 2T(n/2) + 1$ when $n > 1$

$T(n) = 1$ when $n = 1$.

To maintain length of note I wrote single example here, please refer to kec book examples for more numerical problems since solving recurrences related numericals are imp.

Solution:

$$T(n) = 2T(n/2) + 1$$
$$= 2\{2T(n/4) + 1\} + 1$$
$$= 2^2 T(n/2^2) + 2 + 1$$
$$= 2^2\{2T(n/2^3) + 1\} + 2 + 1$$
$$= 2^3 T(n/2^3) + 2^3 + 2 + 1$$
$$\vdots$$
$$= 2^k T(n/2^k) + 2^{k-1} + \ldots + 4 + 2 + 1$$

For simplicity assume:

$n/2^k = 1$

or, $n = 2^k$

Taking log on both sides,

$\log n = \log 2^k$

$\log n = k \log 2$

$k = \log n$ [since $\log 2 = 1$].

Now, $T(n) = 2^k T(n/2^k) + 2^{k-1} + \ldots + 4 + 2 + 1$

$T(n) = 2^k T(1) + 2^{k-1} + \ldots + 2^2 + 2^1 + 2^0$

$T(n) = (2^{k+1} - 1)/(2-1)$

$T(n) = 2^{k+1} - 1$

$T(n) = 2 \cdot 2^k - 1$

$T(n) = 2n - 1$

$T(n) = O(n)$.

2) Recursion Tree:

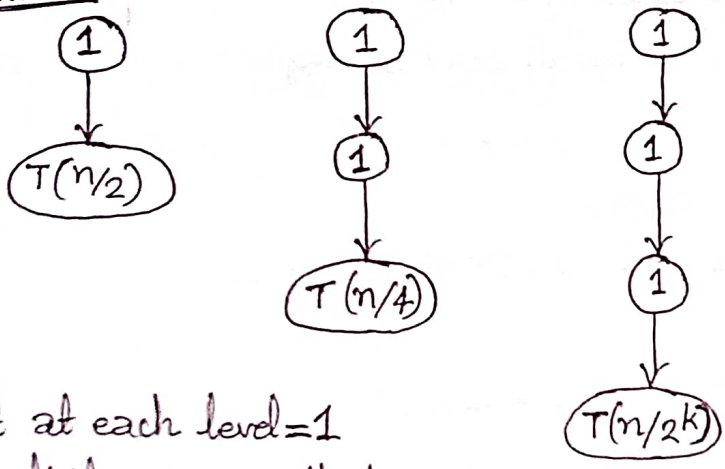
Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded. In general, we consider second term in recurrence as root. Each root and child represents cost of single sub-problem. Summing the cost at each level we determine the total cost.

Example:- Solve the following recurrence relation by using recursion tree method.

$T(1)=1$ when $n=1$

$T(n)=T(n/2)+1$ when $n>1$.

Solution:



Cost at each level=1

For simplicity assume that $n/2^k=1$.

$\Rightarrow n=2^k$

Taking log on both sides,

$\Rightarrow \log n = \log 2^k$

$\Rightarrow k \log 2 = \log n$

$\Rightarrow k = \log n$

Summing the cost at each level,

Total cost $= 1+1+1+\ldots\ldots+T(n/2^k)$

$= 1+1+\ldots\ldots+1 \text{ (k times)} + T(1)$

$= k*1+1$

$= (k+1)$

$= \log n+1$

$\Rightarrow T(n) = O(\log n)$.

3) Substitution method:-

The substitution method for solving recurrences is described using two steps:

i) Guess the form of the solution.

ii) Use induction to show that the guess is valid.

Note: Initially guessing the solution of a problem depends on your practices.

Example: Solve the following recurrence relation by using substitution method.

OR

Show the $O(n^3)$ the complexity of following recurrence relation by using substitution method.

$T(n)=1$ when $n=1$

$T(n)=4T(n/2)+n$ when $n>1$.

Solution:

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

$\Rightarrow T(n)\leq cn^3$, for all $n>=n_0$ ——(i).

Now prove this by mathematical induction as,

Base step: For $n=1$

$T(n)=c*1^3$ Definition.

$1\leq c$ which is true for all +ve values of c.

Inductive step:

Lets assume that it is true $\forall k<n$.

Then $T(k)\leq ck^3$ ——(ii)

It is also true for $k=n/2$

Now equation (ii) becomes,

$T(n/2)\leq c(n/2)^3$

$=cn^3/8$

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

$\leq 4cn^3/8+n$

$=cn^3/2+n$

$=cn^3-cn^3/2+n$

$=cn^3-n(cn^2/2-1)\leq cn^3$

Hence, $T(n)\leq cn^3$ for $\forall n>0$.

Thus, $T(n)=O(n^3)$ proved.

4) Master Method:

Master Method is a direct way to get solution. The master method works only for recurrences that can be transformed to following type:

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

where, $a\geq 1$, $b>1$ are constant, $f(n)$ asymptotically positive function. If the recurrence relation is in this form then there are following four possible cases occurred:

i) Master Method Case 1:

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

$$T(n) = O(n^{\log_b a})$$

ii) Master Method Case 2:

If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constants $\varepsilon > 0$ then,

$$T(n) = O(f(n)).$$

iii) Master Method Case 3:

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

$$T(n) = O(f(n) \cdot \log n)$$

iv) Master Method Case 4:

In this case the master method cannot be applied.

Example:- Solve the following recurrence relation by using Master's method.

$$T(n) = 3T(n/2) + n$$

Solution:

We have, $a = 3$, $b = 2$ and $f(n) = n$.

Now, $n^{\log_b a} = n^{\log_2 3} = n^{(\log 23 / \log 22)} = n^{1.584}$

Also, $f(n) = n^1$

Since $f(n) \leq n^{\log_b a - \varepsilon}$ where choose $\varepsilon = 0.1$

Thus it satisfy the first case of Master's method.

Thus its complexity,

$$T(n) = \Theta(n^{\log_b a}).$$
$$= \Theta(n^{\log 2^3})$$
$$= \Theta(n^{1.58})$$

Thus, $T(n) = \Theta(n^{1.58})$