Design and Analysis of Algorithms

Unit - I

Introduction - Algorithm Design - Fundamentals of Algorithms - Correctness of algorithm - Time Complexity Analysis - Insertion Sort - Line count, Operation count - Algorithm Design paradigms - Designing an Algorithm and its Analysis - Best, Worst and Average Case - Asymptotic notations based on growth functions - $O, o, \Theta, \omega, \Omega$ - Mathematical Analysis - Induction - Recurrence relations - Solution of recurrence relations - Substitution method - Recursion tree - Examples

Lab:
1) Simple Algorithm - Insertion Sort
2) Bubble Sort
3) Recurrence Type - Merge Sort, Linear Search.

Algorithm Design Paradigms: - General approach to the Construction of efficient solutions to problems

1) Brute Force
2) Divide and Conquer - Merge sort, Quick sort
3) Greedy approach
4) Dynamic Programming - Optimizing the recursive approach.
5) Back Tracking
6) Branch and Bound

Algorithm Design Paradigms:

1) **Divide and Conquer:** Given an instance of the problem to be solved, split into several, smaller sub instances, solve each of the sub instances independently and then combine the sub instance solutions so as to yield a solution for the original instance.

eg: Binary Search, Merge Sort.

2) **Dynamic Programming:** It is an algorithm design method used when the solution to a problem can be viewed as a result of sequence of decisions.

eg. Shortest path algorithm, Traveling salesman problem

We try to find optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision.

3) **Greedy method:** The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step, the choice made must be

1) feasible – Has to satisfy the problem's constraints
2) locally optimal – It has to be the best local choice among all feasible choices available on that step
3) irrevocable – Once made, it cannot be changed on subsequent steps of the algorithm.

– "Greedy" grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a optimal solution to the entire problem.

eg: Prim's algorithm for MST, Kruskal's algorithm for MST, Dijkstra's Shortest path algorithm.

4) Backtracking: The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows: If a partially constructed solution can be developed further without violating the problem's constraints, it can be done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

Eg. n-Queens problem, Subset sum problem

5) Branch and Bound: It is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. It is based on the principle that the total set of feasible solutions can be partitioned into smaller subset of solutions. These smaller subsets can then be evaluated systematically until the best solution is found.

Branch and bound is a stepwise enumeration of possible candidate solutions by exploring the entire search space tree. Before constructing the space tree, we set an upper and lower bound for a given problem based on the optimal solution. At each level, we need to make a decision about which node to include in the solution set (choose a node with best bound).

eg: Assignment Problem, Knapsack problem and Traveling Salesman problem.

Time Complexity Analysis:

1) `for(i=n; i>0; i--)`
`{ stat; }` — n

$O(n)$

2) `for(i=0; i<n; i++)`
`{ stat; }` — n

$O(n)$

3) `for(i=1; i<n; i+2)`
`{ stat; }` — $n/2$

Same for increment of 5, 10 or 20.

Even though it's $n/2$

$f(n) = \frac{n}{2} = O(n)$

4) `for(i=0; i<n; i++)` — $(n+1)$
`{`
`for(j=0; j<n; j++)` — $n*(n+1) = n^2+n$
`{ stat; }` — $n*n = n^2$
`}`
`}` $(n+1)(n+1) = 2n^2+2n+1$

$= O(n^2)$

5) `for(i=0; i<n; i++)` —
`{`
`for(j=0; j<i; j++)`
`{ stmt; }`
`}`

$1+2+3+4+\dots+n = \frac{n(n+1)}{2}$

$O(n^2)$.

i	j	no. of times
0	0	0
1	0	1
	1	1 x
2	0	
	1	2
		2 *
3	0	
	1	
	2	3
	3	3 x
n		n

Total no. of execution.

Finding Time Complexity of an Algorithm:

1) Instruction Count

2) Operation count (Basic Operations) – BODMAS

3) (Table method) Step count – A program step is defined as a syntatically / semantically meaningful segment of a program that has an execution time independent of the instance characteristics

Steps assigned to any program segment depends on the kind of statement

1) Comment – Zero

2) Assignment – One

3) for, while, repeat – until – Consider step counts only for the control part of the statement.

The table contains S/e and frequency

S/e – amount by which the count changes as a result of execution of that statement

freq – total no. of times each statement is executed.

eg:1

```
Algorithm Sum(a,n)
{
    s = 0.0;
    for i=1 to n do
        s = s + a[i];
    return s;
}
```

S/e	frequency	total steps
0	—	0
0	1	0
1	1	1
1	n+1	n+1
1	n	n
1	1	1
0	0	0
		2n+3

Eg:2

	s/e	freq n=0	freq n>0	total steps n=0	total steps n>0
Algorithm Rsum(a,n)	0	-	-	0	0
{	0	-	-	-	-
if (n ≤ 0) then	1	1	1	1	1
return 0.0	1	1	0	1	0
else return					
Rsum(a,n-1)+a[n]	1+x	0	1	0	1+x
}	0	-	-	0	0
				2	2+x

Correctness of an algorithm:

1) Proof by induction (Direct Proof)
2) Proof by contradiction
3) Proof by Loopinvarient
4) Proof by Counter example (Indirect Proof)
5) Proof by cases
6) Proof by chain of iffs
7) Proof by contrapositive

Proof by induction:

1) Prove base case is TRUE
2) Assume that for 'n', it is TRUE
3) Try to prove for case 'n+1'

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

case n=1: $\sum_{i=1}^{1} i = \frac{1(1+1)}{2} = 1 = \text{TRUE}$

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

Try to prove for case 'n+1'

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

W.K.t $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = 1+2+3+\cdots+(n-1)+n$

$$\therefore \sum_{i=1}^{n+1} i = \frac{(n+1)(n+1+1)}{2} = 1+2+3+\cdots+(n+1-1)+(n+1)$$

$$\frac{(n+1)(n+2)}{2} = 1+2+3+\cdots+n+(n+1)$$

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

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

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

L.H.S = R.H.S

Hence proved.

Proof by Loop invarient:

Invarient : Something that is always TRUE

After finding loop invarient, we have to proove

1) Initialization – How does the invarient get initialized?
2) Loop Maintenance – How does the invarient changes at each pass of the loop?
3) Termination – Does the loop stop? when?

Proof by contrapositive:

Form the contrapositive statement, proove it and thereby proove original as TRUE

eg: if A then B $A \Rightarrow B$

Then if $\neg B \Rightarrow \neg A$

if x^2-6x+5 is even, then x is odd

Contrapositive: If x is even, then x^2-6x+5 is odd

Let $x = 2a$ then $(2a)^2 - 6(2a)+5$

$= 4a^2 - 12+5 = 2(2a^2-6a+2)+1$

Thus x^2-6x+5 is odd.

Proof by counter example:

Identify a case for which our assumption is not TRUE.

eg. Prove or disprove

$\lceil x+y \rceil = \lceil x \rceil + \lceil y \rceil$, $\forall x \forall y (xy \geq x)$

if $x = \frac{1}{2}$ and $y = \frac{1}{2}$

Soln: Lets assume if $x = -1$, $y = 3$, $xy = -3$; $-3 \not\geq -1$.

Hence proved.

Proof by Cases:

Splitting the problem into subparts and try to prove them one by one.

Proof by chain of iffs:

Using more than one conditions to proove.

Proof by contradiction:

Try to prove that the given statement is FALSE and if the result is not FALSE, then say that the statement is TRUE.

Recurrence relations: An equation or inequality that describes a function in terms of its value on smaller inputs. ← Basic step

How to write recurrence relations for a given code ← Recursive step

1)
```
void Test(int n)          — T(n)
{
    if(n>0)               — 1
        printf("%d",n);   — 1
    Test(n-1);            — T(n-1)
}
```

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+2, & n>0 \end{cases}$$

$$T(n) = T(n-1) + 2 \Rightarrow T(n) = T(n-1) + 1$$

2)
```
void Test(int n)          — T(n)
{
    if(n>0)               — 1
    {
        for(i=0; i<n; i++)  — n+1
        {
            printf("%d",n);  — n
        }
        Test(n-1);          — T(n-1)
    }
}
```

$$T(n) = T(n-1) + 2n + 2$$

$\Downarrow$ can be approximated to

$$T(n) = T(n-1) + n$$

$$T(n) = \begin{cases} 1 & , n=0 \\ T(n-1)+n & , n>0 \end{cases}$$

3)
```
void Test(int n)          — T(n)
{
    if(n>0)               — 1
    {
        for(i=0; i<n; i=i*2)  — logn+1
        {
            printf("%d",i);  — logn
        }
        Test(n-1);          — T(n-1)
    }
}
```

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+\log n, & n>0 \end{cases}$$

$$T(n) = T(n-1) + 2\log n + 2$$

$\Downarrow$ can be approximated to

$$T(n) = T(n-1) + \log n$$

4) Void Test(int n) — T(n)

```
{ if (n>1)                    — 1
  {
    for(i=0; i<n; i++)        — n+1
    {
      stmt;                   — n
    }
    Test(n/2);                — T(n/2)
    Test(n/2);                — T(n/2)
  }
}
```

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

⇓ approximated to

$$\boxed{T(n) = 2T(n/2) + n}$$

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

Solving recurrence relations:

1) Iteration method
2) Substitution method
3) Recursion tree method
4) Master's theorem.

Substitution method:

Procedure:

1. Guess the solution
2. Use the mathematical induction to find the boundary condition and show that the guess is correct.

1) $T(n) = T(n-1)+1$

To guess the solution, we can use forward substitution method

Given $T(n) = T(n-1)+1$ — (1)

$T(n-1) = T(n-2)+1$ — (2)

Substituting (2) in (1) we get

$T(n) = T(n-2)+1+1$

$T(n) = T(n-2)+2$ — (3)

Substituting (4) in (3) we get

$T(n) = T(n-3)+3$ — (5)

Looking at (1), (3) and (5), it is understood that,

after K times $T(n) = T(n-k)+k$ — (6)

w.k.t $T(0) = 1$

∴ if $n-k = 0$, $n = k$ or $\boxed{k=n}$

∴ (6) becomes $T(n) = T(n-n)+n$

$T(n) = T(0)+n = n+1$

∴ $T(n) = O(n)$

Step 1: Guess the solution, $T(n) = O(n)$

Step 2: We have to show that for constant, c

$T(n) \leq c.n$

Induction Base: $T(0) = 1$

Induction Hypothesis: $T(n-1) \leq c(n-1)$

To prove: $T(n) \leq c.n$

$T(n) = T(n-1)+1$

$\leq c(n-1)+1$

$= cn - c + 1 \leq cn.$

Hence proved.

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

To guess the solution,

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad -(1)$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right) \quad -(2)$$

Substituting (2) in (1)

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)\right] + n$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 2n \quad -(3)$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right) \quad -(4)$$

Substituting (4) in (3)

$$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)\right] + 2n$$

$$T(n) = 8T\left(\frac{n}{8}\right) + 3n \quad -(5)$$

Comparing (1), (3) and (5)

General equation is $T(n) = 2^k T\left(\frac{n}{2^k}\right) + k.n$

W.K.T $T(1) = 1$

i.e. $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$$\therefore T(n) = 2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + n\log_2 n$$

$$\boxed{T(n) = O(n \log n)}$$

To poove that $T(n) \leq c.n\log n$

Induction Base : $T(1) = 1$

Induction Hypothesis : $T\left(\frac{n}{2}\right) \leq c.\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)$

W.K.t. $T(n) = 2T\left(\frac{n}{2}\right) + n$

$$\leq 2c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + n$$

$$= cn\log\left(\frac{n}{2}\right) + n$$

$= cn\log n - cn\log 2 + n$

$= cn\log n - cn + n \leq cn\log n$

Hence proved.

3) $T(n) = \begin{cases} 1 & , \quad n = 1 \\ 2T\left(\frac{n}{2}\right) + 1 & , \quad n > 1 \end{cases}$

Step 1: Guess the solution:

$T(n) = O(n)$

Step 2: Prove that the guess is correct using induction.

Induction Base: $T(1) = 1$

Induction Hypothesis: $T\left(\frac{n}{2}\right) \leq c.\left(n/2\right)$

To prove: $T(k) \leq c.k$ is true, $\forall k < n$

$T(n) = 2T\left(\frac{n}{2}\right) + 1$

$\leq 2c.\left(\frac{n}{2}\right) + 1$

$= cn + 1 \leq cn.$

Hence proved.

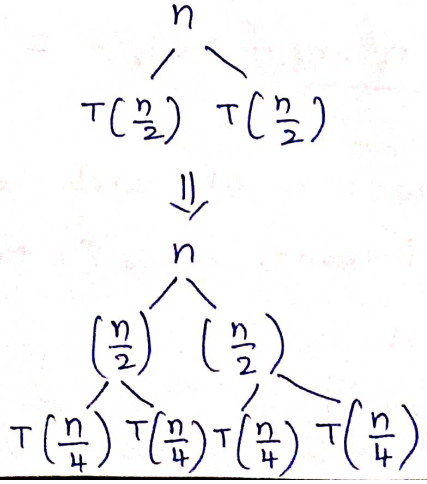
Recursion Tree Method:

- Making a good guess is sometimes difficult with the substitution method
- Recursion Tree method can be used to devise a good guess
- Recursion trees show successive expansions of recurrences using trees.
- Recursion Trees model the cost of a recursive algorithm that is composed of two parts
  1. Cost of non recursive part
  2. Cost of recursive call on smaller input size.
- A Tree node represents the cost of a sub-problem
- To determine the total cost of the Recursion Tree, evaluate
  a) Cost of individual node at depth "i"
  b) Sum up the cost of all nodes at depth "i"
  c) Sum up all per level costs of the Recursion Tree

Example 1:

$$T(n) = \begin{cases} 1 & n=1 \\ 2T\left(\frac{n}{2}\right)+n & n>1 \end{cases}$$

Root = Cost of non recursive part



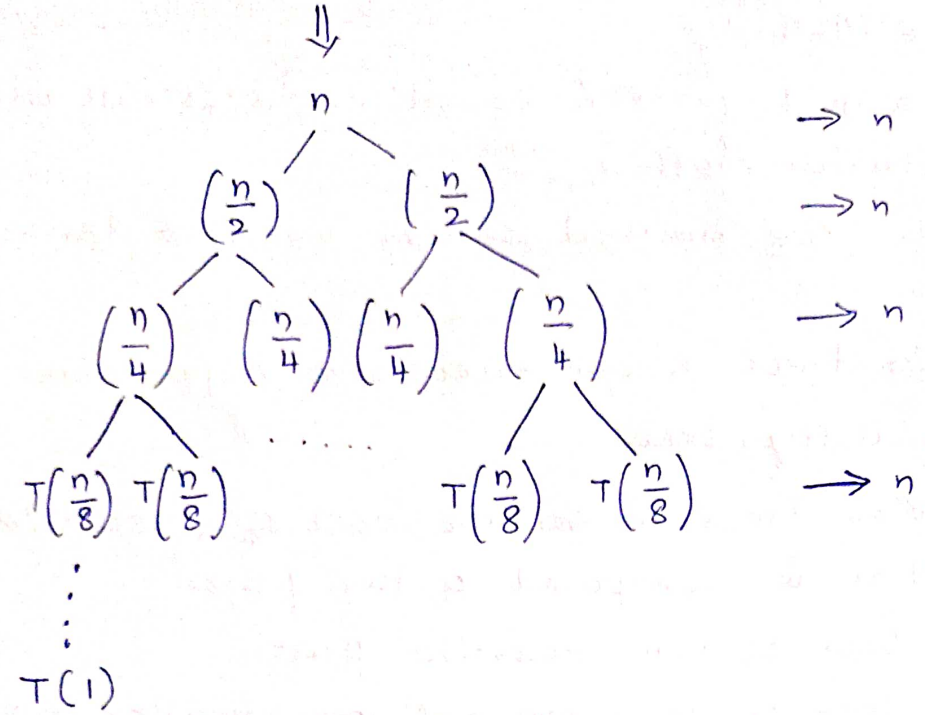
$$T(n) = 2T\left(\frac{n}{2}\right)+n$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right)+\frac{n}{2}$$

$$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}$$

$$\vdots$$

$$T\left(\frac{n}{2^k}\right) = T(1)$$



Let T(1) occurs at depth 'K'. ie. Problem size reduces to 1 at k^{th} level.

Total cost = Cost of Leaf nodes + Cost of Internal nodes.

Total cost = (cost of a leaf node × Total no. of Leaf nodes) + Sum of costs at each level of Internal nodes

$= L_c + I_c$

For this, we need to find the value of K in terms of n

$$\frac{n}{2^K} = 1 \Rightarrow \boxed{K = \log n}$$

L_c:

The cost of a leaf node = $T(1) = 1$

No. of leaf nodes at 'k^{th}' level = 2^k nodes

$\therefore$ Cost of leaf nodes $(L_c) = 2^{\log n} = 'n'$

$$\boxed{L_c = n}$$

I_c:

Cost of internal node = n at each level

No. of levels of internal nodes (0 to k-1) = K levels

$I_c = K \cdot n$

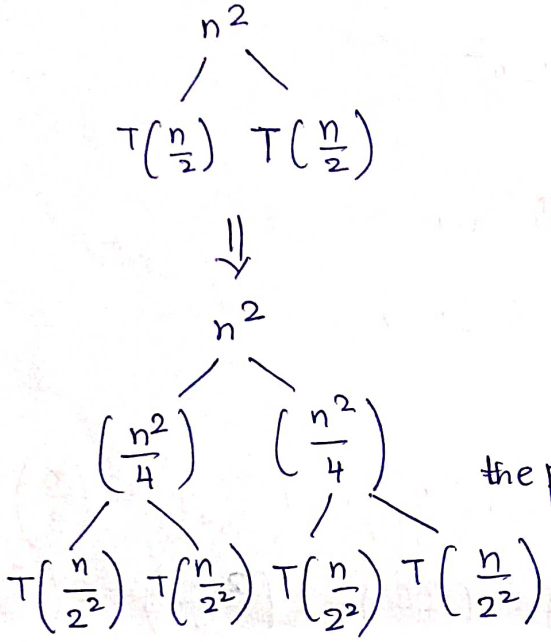
$$\boxed{I_c = n \log n}$$

Total cost = $L_c + I_c$

$T(n) = n \log n + n$

$\therefore T(n) = O(n \log n)$

2) $$T(n) = \begin{cases} 1 & n=1 \\ 2T\left(\frac{n}{2}\right) + n^2 & n>1 \end{cases}$$



$T(n) = 2T\left(\frac{n}{2}\right) + n^2$

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n^2}{2^2}$

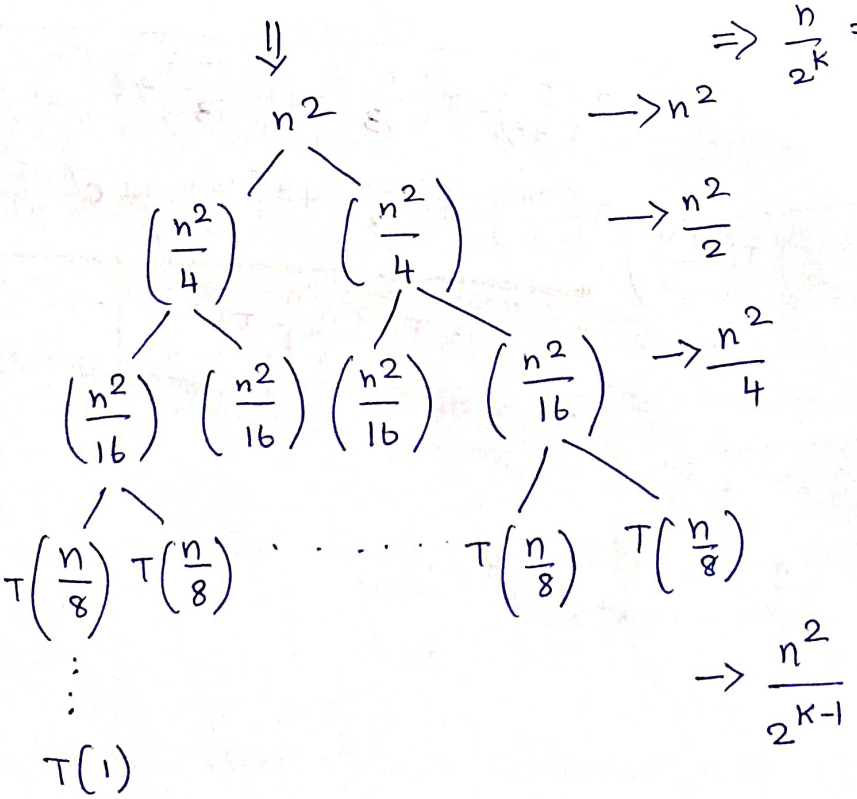
$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n^2}{4^2}$

Let us assume at K^{th} level, the problem reduces to $T(1)$

i.e. $T\left(\frac{n}{2^K}\right) = T(1)$

$\Rightarrow \frac{n}{2^K} = 1 \Rightarrow n = 2^K \Rightarrow$

$\boxed{K = \log_2 n}$



Cost of a leaf node = $T(1) = 1$

Total no. of leaf nodes at k^{th} level $= 2^k$

$$L_C = 2^k = 2^{\log n} = n^{\log 2} = n$$

$$I_C = n^2\left[\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^{k-1}\right]$$

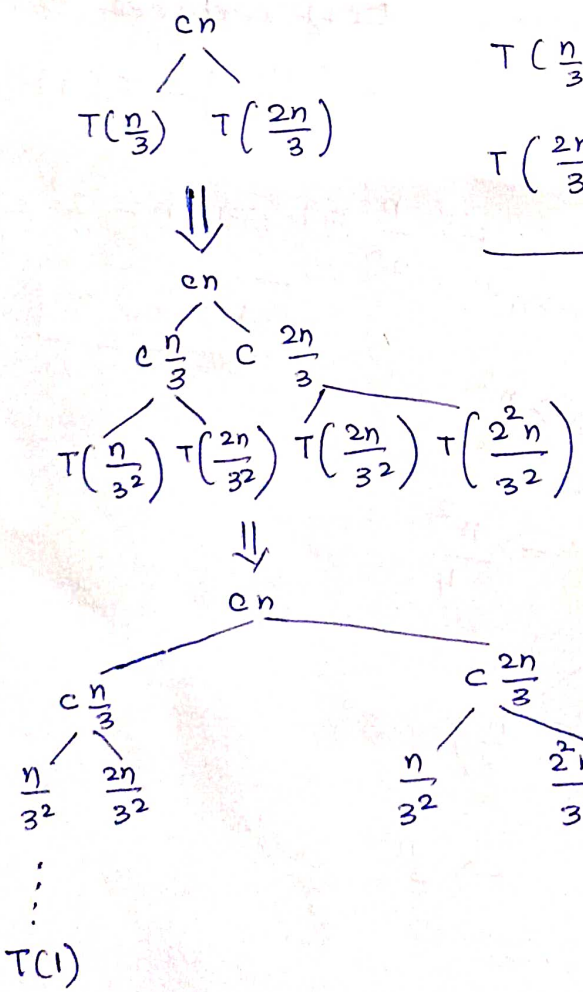
Since $x < 1$

$$I_C = n^2\left[\frac{1}{1-1/2}\right] = 2n^2$$

$\therefore$ Total cost $= 2n^2 + n$

$\therefore T(n) = O(n^2)$

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



$$T\left(\frac{n}{3}\right) = T\left(\frac{n}{3^2}\right) + T\left(\frac{2n}{3^2}\right) + c\left(\frac{n}{3}\right)$$

$$T\left(\frac{2n}{3}\right) = T\left(\frac{2n}{3^2}\right) + T\left(\frac{2^2n}{3^2}\right) + c\left(\frac{2n}{3}\right)$$

$$T\left(\frac{n}{3^2}\right) = T\left(\frac{n}{3^3}\right) + T\left(\frac{2n}{3^3}\right) + c\left(\frac{n}{3^2}\right)$$

$$T\left(\frac{2n}{3^2}\right) = T\left(\frac{2n}{3^3}\right) + T\left(\frac{2^2n}{3^3}\right) + c\left(\frac{2n}{3^2}\right)$$

$$T\left(\frac{2^2n}{3^2}\right) = T\left(\frac{2^2n}{3^3}\right) + T\left(\frac{2^3n}{3^3}\right) + c\left(\frac{2^2n}{3^2}\right)$$

$\longrightarrow cn$

$\longrightarrow cn$

$\longrightarrow cn$

For finding the cost of the nodes at last level, we have to consider the deeper path from root node.

The path from $cn \to c\frac{2n}{3} \to c\frac{2^2 n}{3^2}$ is deeper.

Let k^{th} level is the last level where $T(1) = 1$

$cn \to$ level 0 $\to c\frac{2^0 n}{3^0}$

$c\frac{2n}{3} \to$ level 1 $\to c.\frac{2^1 n}{3^1}$

$c\frac{2^2 n}{3^2} \to$ level 2 $\to c.\frac{2^2 n}{3^2}$

$\therefore$ At k^{th} level $T\left(\frac{2^k n}{3^k}\right) = 1$

$$\frac{2^k n}{3^k} = 1 \Rightarrow$$

$$n = \frac{3^k}{2^k}$$

$$\log n = \log\left(\frac{3^k}{2^k}\right) \Rightarrow \log 3^k - \log 2^k$$

$$\Rightarrow k \log 3 - k \log 2$$

$$\log n \Rightarrow k \log\left(\frac{3}{2}\right)$$

$$k = \frac{\log_2 n}{\log\left(\frac{3}{2}\right)}$$

$$k = \log_{\left(\frac{3}{2}\right)} n$$

$\because \log_a m = \frac{\log_b m}{\log_b a}$

Total no. of nodes at last level $= 2^k$

Total cost of last level $= 2^{\log_{\left(\frac{3}{2}\right)} n} \cdot 1$

$$= n^{\log_{\left(\frac{3}{2}\right)} 2}$$

Total cost of Internal nodes $= n \cdot k$

$$= n \log_{\left(\frac{3}{2}\right)} n$$

$\therefore$ Total cost of the tree $= n^{\log_{\left(\frac{3}{2}\right)} 2} + n \log_{\left(\frac{3}{2}\right)} n$

Master's Theorem:

- Applicable for recurrence relations of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \quad a \geq 1, b > 1$$

There are 3 cases

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$, $\epsilon > 0$, then

$T(n) = \Theta(n^{\log_b a})$ → when no. of leaves/nodes dominate the work done at any level

Case 2: If $f(n) = \Theta(n^{\log_b a})$ then

$T(n) = \Theta(n^{\log_b a} \log n)$ → Both no. of nodes and work at any level are equal

Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ &&

$af(n/b) \leq c f(n)$, $\epsilon > 0, c < 1$, then

$T(n) = \Theta(f(n))$ → Work on a level dominates

Eg: 1) $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$.

Comparing with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $a \geq 1$, $b > 1$

$a = 8$, $b = 2$, $f(n) = 1000n^2$

Substituting the above in case 1

$f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow \log_b a = \log_2 8 = 3$

$= O(n^{3-\epsilon})$

If $\epsilon = 1$, then $f(n) = O(n^2) = 1000n^2$

So, Case 1 holds

$\therefore T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$.

$$\boxed{T(n) = \Theta(n3)}$$

Eg:2: $T(n) = 2T\left(\frac{n}{2}\right) + 10n$

When compared with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $a \geq 1$, $b > 1$

here $a = 2$, $b = 2$, $f(n) = 10n$

$$\log_b a = \log_2 2 = 1$$

Substituting the above values in case 2

$$f(n) = \Theta(n^{\log_b a})$$

$$f(n) = \Theta(n^1) = \Theta(n)$$

$$10n = \Theta(n)$$

This case holds.

$\therefore$ $T(n) = \Theta(n \log n)$

Eg: 3: $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

When compared with $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, $a \geq 1$, $b > 1$

here $a = 2$, $b = 2$, $f(n) = n^2$

$$\log_b a = \log_2 2 = 1$$

Substituting the above values in case 3

$$f(n) = \Omega(n^{\log_b a + \epsilon})$$

$$= \Omega(n^{1+\epsilon}) \quad \text{if } \epsilon = 1$$

then $f(n) = \Omega(n^2)$

Hence Case 3 holds

Checking Second condition

$$2f(n/2) \leq c \cdot n^2$$

$$2 \cdot \left(\frac{n}{2}\right)^2 \leq c \cdot n^2$$

$$\frac{n^2}{2} \leq c \cdot n^2$$

If we choose 'c' as $\frac{1}{2}$, then

$\frac{n^2}{2} \leq \frac{n^2}{2}$. It is True

$\therefore T(n) = \Theta(f(n))$

i.e. $T(n) = \Theta(n^2)$

Other examples:

Case 1: $T(n) = 9T(\frac{n}{3}) + n$

Case 2: $T(n) = T(\frac{2n}{3}) + 1$

Case 3: $T(n) = 3T(\frac{n}{4}) + n\log n$