DAA

Unit-1

References:

1. "Introduction to algorithms", Cormen, Leiserson, Rivest, Stein, PHI

2. "Introduction to The Design and Analysis of Algorithms", levitin, Pearson

Introduction to Algorithm

An algorithm is a sequence of unambiguous instructions for solving a problem. It takes some valid input and produces the desired output in a finite amount of time.

Criterion to write an algorithm:

1. Definiteness: Each instruction of an algorithm should be clear and written unambiguously.

2. Finiteness: Any repetition of statements must be terminated after a finite number of steps.

3. Effectiveness: Every instruction must have some effect towards the solution of the problem.

Fundamentals of problem solving using algorithm:

1. Understanding the problem: This means one must know and understand the problem in hand clearly. This includes objective of the problem, output requirements, input instance and range of input instance of the problem

2. Ascertaining the capabilities of the machine: It is very much important to know the technical characteristics of the machine, where the algorithm going to be tested. The machine may be a sequential or parallel computer, hand held device or any other device.

3. Choosing between the exact solution and the approximate solutions: Most of the problems need exact solution, like sorting an array of n numbers.

But there exist some kind of problems, we need approximate solution. We need approximate algorithm for the problem, whose algorithm for exact solution is

unacceptable slow or the problem cannot be solved for most of the instances. Examples are *travelling salesman problem*, *finding square root of a number.*

4. Deciding appropriate data structure: Data structures are the logical and mathematical model of organization of data. Some of the examples of data structures you come across are: array, linked list, stack, queue, various tree data structures, and graph. Choosing appropriate data structure is important as affects the efficiency of the algorithm.

5. Algorithm design techniques: Design techniques are the strategy on which algorithms are written. Some of the design techniques are: Brute force, divide and conquer approach, dynamic programming, greedy approach, iterative increment approach, backtracking, branch and bound techniques.

6. Method of specifying an algorithm: Two widely used approaches used to specify an algorithm are: Natural Language representation and pseudo code.

Natural language representation causes ambiguity and difficulty to code in any programming language because of unclear definition.

Pseudo code is a mixture of natural language and programming language constructs. It is precise and gives clear description of the algorithm.

7. Correctness of algorithm: This is utmost required to check the whether the algorithms give required output for every possible set of valid input.

8. Analyzing the algorithm: For one problem, many algorithms exist. So it is required to have a comparison among them to find efficient one. This is achieved by analyzing the algorithms in terms of number of time, amount of memory space, amount of hardware resources used by the algorithm to solve a problem.

9. Coding: It is the transformation of algorithm to program, that can be executed in a computing device.

Example1 : Write an algorithm to add all the elements of an array with n elements.

ADDELEMENTS(A,n):

//This algorithm computes the sum of all elements of an array A[1….n] of n elements and //returns the sum.

1. sum = 0
2. **for** k = 1 to n STEP 1
3. sum = sum + A[k]
4. **return** sum.

Example 2 : Write an algorithm to search number in an n-element array.

LSEARCH(A, n, number):

//This algorithm search the location of number in an array A[1…n] of n elements. It returns the location of number in array, If it is present .It returns 0 is the number is not present in array.

1. loc = 0
2. **while** loc <= n AND number ≠ A [loc]
3. loc = loc + 1
4. **return** loc.

Fundamentals of Algorithm analysis

1. The Analysis Framework: *(Input Size, unit of measurement of time, Efficiency cases)*

Algorithm analysis includes the time, space requirement by an algorithm in order to solve a particular problem.

Amount of time needed by an algorithm to solve a problem, is called **time complexity**. Time here means not the clock time, but number of execution steps involved in the algorithm. Similarly, amount of memory spaces used by an algorithm is called **space complexity.**

Input Size:

This is the size of input data or input instance that an algorithm takes to process. The running time of an algorithm is directly proportional to the size of input data to the algorithm. Consider the algorithm of computing sum of an array. If the array size is more, number of addition is more.

Unit of measuring running time:

There are different approaches to measure the running time. We can count total number of steps of the algorithm by targeting all the steps and the loops. Another approach is to count the number of times algorithm spent on executing **basic operation**.

Basic operation is most important operation of the algorithm and the operation that contribute the most to the running time.

Measuring the running time of the algorithm

Approach -1

ADDELEMENTS(A,n):

This algorithm computes the sum of all elements of an array A[1….n] of n elements and returns the sum.

1. sum = 0	- -	1
2. FOR k = 1 to n STEP 1	- - - - - - - - - - - - - - - - - - - -	n+1
3. sum = sum + A[k]	- - - - - - - - - - - - - - - - - - - -	n
4. RETURN sum.	- - - - - - - - - - - - - - - - - - - -	1

		T(n) = 2n + 3

Approach -2

Here the basic operation is the line where the addition operation is performed, i,e line-3. We can see this line is executed n times. So we can say that T (n) = n.

The running time calculated in approach-1 and approach-2 seems to be different, but they come under same class of efficiency. [We will see this next section]

Best- case, worst-case and average- case efficiency:

There are some kinds of problem, where same algorithm behaves differently. This different behavior is due to the input instances. For some input instance, the algorithm computes the problem quickly and for some other input instance algorithm runs slow.

Best- case of the algorithm occurs when an algorithm, for an input instance, takes minimum amount of time to compute the problem.

Worst- case of the algorithm occurs when an algorithm, for an input instance, takes maximum amount of time to compute the problem

Average- case of the algorithm occurs when an algorithm, for an input instance, takes average amount of time to compute the problem. It is difficult to calculate average running time of an algorithm.

2.2. Asymptotic Notation

Asymptotic notations are used to describe the running time of an algorithm and compare the order growth of two functions. Following are the notations used to specify the running time.

Notations	Symbols	Gives
Big Oh notation	O	Upper bound (tight)
Big Omega notation	Ω	Lower Bound (Tight)
Theta notation	θ	Tight bound
Little Oh notation	o	Upper bound
Little Omega notation	ω	Lower Bound

Big Oh Notation:

This notation gives an upper bound to the asymptotic growth of the function.

Definitions: If f(n) and g(n) are two functions defined for non-negative integers, then f(n) =O(g(n)) if and only if there exists two positive constants c and n0 such that $0 \leq f(n) \leq c \times g(n)$ for all n >= n0.

***Problem**: The running time of an algorithm is described by a function f(n) = 3n+4. Find the Big Oh Notation and find the value of the constant c and n0.*

Solution:

$f(n) = 3n + 4$

$\Rightarrow f(n) \leq 3n + n$

$\Rightarrow f(n) \leq 4n$

$\Rightarrow f(n) = O(n),$ and c =4

n	f(n)=3n+4	c.g(n)=4xn	f(n)≤c.g(n)
0	4	0	False
1	7	4	False
2	10	8	False
3	13	12	False
4	16	16	True
5	19	20	True

The relation f() ≤ c* g(n) is true at n = 4 and continued to be true for any value of n ≥ 4. That means the growth rate of f(n) is less than equal to growth rate of c*g(n) for all value of n that is greater than or equal to 4. So value of $n_0 = 4$.

So the running time of the algorithm whose running time is expressed by linear function f(n) = 3n + 4 is O(n).

Exercise: *Find asymptotic growth of following functions by using Big Oh notation.*

1. $f(n) = 2n^2 + 3n + 5$
2. $f(n) = 2^n + 5n$
3. $f(n) = 5000$

Some Rules on Big Oh

1. If $f1(n) = O(g1(n))$ and $f2(n) = O(g2(n))$, then

$$f1(n) + f2(n) = O(\max(g1(n), g2(n)))$$

$$f1(n) + f2(n) = O(g1(n) \times g2(n))$$

2. $O(k \times f(n)) = O(f(n)$, k is any constant.,

3. $f1(n) - f2(n) = f1(n) + (-f2(n)) = O(\max(g1(n), -g2(n)))$

4. If $f(n) = a_m \times n^m + a_{m-1} \times n^{m-1} + \ldots\ldots\ldots + a_{1-} \times n + a_0$, Then $f(n) = O(n^m)$

Big Omega Notation:

This notation gives a lower bound to the asymptotic growth of the function.

Definitions: If f(n) and g(n) are two functions defined for non-negative integers, then f(n) = Ω(g(n)) if and only if there exists two positive constants c and n0 such that $f(n) \geq c \times g(n)$ for all n >= n0.

$f(n) = 3n + 4$

$\Rightarrow f(n) \geq 3n$

$\Rightarrow f(n) = \Omega(n),$

$Where \cdot c = 3$

Theta Notation:

This notation gives both tighter upper bound and tighter lower bound to the asymptotic growth of the function.

Definitions: If f(n) and g(n) are two functions defined for non-negative integers, then f(n) = θ(g(n)) if and only if there exists three positive constants c1,c2 and n0 such that $c1 \times g(n) \leq f(n) \leq c2 \times g(n)$ for all n >= n0.

To find time complexity of a function in terms of theta notation, we need to find a lower bound i ,e $c1 \times g(n) \leq f(n)$ and an upper bound i,e $f(n) \leq c2 \times g(n)$

Lets the function given is f(n) =3n+4

To find a tighter lower bound,

$f(n) = 3n + 4$

$\Rightarrow f(n) \geq 3n$ …… ……..(eq. 1)

So $f(n) = \Omega(n),$ and the constant c1 =3.

To find a tighter upper bound,

$f(n) = 3n + 4$

$\Rightarrow f(n) \leq 3n + n$

$\Rightarrow f(n) \leq 4n$ ……(eq. 2)

So f(n) = O(n) and the constant c2 = 4

Form eq. 1 and eq.2 we get

$3 \times n \leq f(n) \leq 4 \times n$

So we can write f(n) = θ(g(n)) and the constant c1 = 3 and c2 = 4.

Little Oh Notation:

Like Big Oh, this notation also gives an upper bound to the asymptotic growth of the function. But this notation does not give the tighter bound.

Definitions: If f(n) and g(n) are two functions defined for non-negative integers, then f(n) =o(g(n)) if and only if there exists two positive constants c and n0 such that $0 \leq f(n) < c \times g(n)$ for all n >= n0.

Little Omega Notation:

Like Big Omega, this notation also gives a lower bound to the asymptotic growth of the function. But this does not give a tighter lower bound.

Definitions: If f(n) and g(n) are two functions defined for non-negative integers, then f(n) = ω (g(n)) if and only if there exists two positive constants c and n0 such that $f(n) > c \times g(n)$ for all n >= n0.

Tight bound and loose bound/Big Oh and Little Oh

Both the notations give an upper bound to the asymptotic growth of a function. But the difference between the two is that, Big Oh notation is used to denote the

upper bound that is asymptotically tight, but the little Oh is used to denote the upper bound that is not asymptotically tight.

For example:

$5n = o(n^2)$: Correct , as growth rate of $5n \le n^2$, and not tight
$5n = O(n^2)$: Correct, though not night, growth rate of $5n \le n^2$
$5n^2 = o(n^2)$: Incorrect , as this is tight bound.
$5n^2 = O(n^2)$: Correct , as this is tight bound

Suppose there are two function f(n) and g(n), Then g(n) is not asymptotic tight bound to the function f(n) I,e we can say f(n) = o(g(n)) , if following condition holds.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Similarly little omega notation (ω) can be used if you want to denote a lower bound that is not asymptotically tight.

That means f(n) = ω(g(n)) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Using Limit theory to compare order of growth

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & \text{If f(n) has smaller growth rate than g(n)} \\ c & \text{If f(n) has same growth rate as g(n)} \\ \infty & \text{if f(n) has higher growth rate than g(n)} \end{cases}$$

Exercise: Compare following functions using limit.

1. nlogn and n
2. $\frac{1}{2}n(n-1)$ and n2
3. $Log_2 n$ and $\sqrt{n}$
4. n! and 2^n Hints: Use Stirling's approximation

Relational properties, applicable to asymptotic comparison of functions.

1. Transitive Property:

i. If $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$, Then $f(n) = \theta(h(n))$

ii. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, Then $f(n) = O(h(n))$

iii. If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, Then $f(n) = \Omega(h(n))$

iv. If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, Then $f(n) = o(h(n))$

v. If $f(n) = \omega(g(n))$ and $g(n) = \omega\theta(h(n))$, Then $f(n) = \omega(h(n))$

2. Reflexive property

i. $f(n) = \theta(f(n))$

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

iii. $f(n) = \Omega(f(n))$

3. Symmetry Property:

i. $f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$

4. Transpose property

i. $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

ii. $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Basic Algorithm Efficiency class

Efficiency	Algorithm name
1	Constant time algorithm
log n	logarithmic time
n	linear time algorithm
n log n	log linear time algorithm
n^2	Quadratic time algorithm
n^3	Cubic time algorithm
2^n	Exponential time
n!	Factorial time algorithm

Analysis of Some Non-recursive algorithm

Analysis-1: Following algorithm determines whether all elements in an array are distinct.

Algorithm: CheckDistinctElement(A,n)

//This algorithm return TRUE, if all elements in A are unique and return

// FALSE , otherwise.

```
1. for i = 1 to n
2.   for j = i+1 to n
3.       if A [i] = A [j]
4.           return FALSE
5. return TRUE
```

Here basic operation is the comparison operation at line number 3. This line is executed for each value of j from i +1 to n. and this is repeated for each value of i from 1 to n.

So number of steps, the basic operation is exceed is as follows

$$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (1)$$

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

$$= n - 1 + n - 2 + \ldots\ldots + 1 \quad = \frac{n(n-1)}{2} \quad = \frac{1}{2}n^2 - \frac{1}{2}n \quad = O(n^2)$$

Analysis-2: Insertion sorts

Insertion sort is an efficient algorithm for sorting of the elements in an array. It is based on playing cards sorting technique. We start with an empty hand and the cards (numbered) are on the table. Then we remove one card from the table and insert it into its correct position in the hand of sorted cards. To find the correct

position of a card, we compare it with each of the already sorted cards from right to left.

Algorithm: `InsertionSort(A,n)`

//This algorithm sorts all the elements in the array A with n elements by using insertion techniques.

1.	**for** j = 2 to n	.....	n
2.	key = A [j]	.....	n-1
3.	i =j-1	.....	n-1
4.	**while** i ≥ 1 and A[i] >key	.....	$\sum_{j=2}^{n}(t_j)$
5.	A[i+1] = A[i]	.....	$\sum_{j=2}^{n}(t_j-1)$
6.	i = i-1	.....	$\sum_{j=2}^{n}(t_j-1)$
7.	A [i + 1] = key	.....	n-1

Adding the frequency of all the lines, We get,

$$T(n) = n+n-1+n-1+\sum_{j=2}^{n}(t_j)+\sum_{j=2}^{n}(t_j-1)+\sum_{j=2}^{n}(t_j-1)+n-1$$

Worst-case running time of insertion sort.

In insertion sort, worst case occurs, if the elements in the array are present in reversely sorted order. In this case, we must compare each element A[j] with each element in the already sorted sub-array A [a ... j – 1]. this means a total number of 'j' number of comparison is required to sort A[j]. So t_j = j for j = 2 to n.

so $$\sum_{j=2}^{n}(t_j) = \sum_{j=2}^{n} j = 2+3+.....+n = \frac{n(n+1)}{2}-1 = \frac{n^2}{2}+\frac{n}{2}-1. \quad \text{..... (1)}$$

and $\sum_{j=2}^{n}(t_j-1)=\sum_{j=2}^{n}(j-1)=1+2+3.....+n-1=\frac{n(n-1)}{2}=\frac{n^2-n}{2}$ .....(2)

Putting the value of (1) and (2) in T(n), we get

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

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

$$=3n-4+\frac{n^2}{2}+\frac{n}{2}+n^2 \quad =\frac{3}{2}n^2+\frac{7}{2}n-4 \qquad \mathbf{=O(n^2)}$$

Best-case running time of insertion sort.

In insertion sort, best case occurs, if the elements in the array are already present in sorted order. In this case for each element A [j] for j = 2 to n , we find A[i] ≤ key in inner while loop. So the loop in line number 4 is checked only once. So t_j =1 for j = 2 to n.

S0 $\sum_{j=2}^{n}(t_j)=\sum_{j=2}^{n}1=1+1+.....+1=n-1.$ ..... (3)

and $\sum_{j=2}^{n}(t_j-1)=\sum_{j=2}^{n}0=0+0+.....+0=0$ ..... (4)

Putting value of (3) and (4) in T(n) , we get

$T(n)=n+n-1+n-1+n-1+0+0+n-1 \quad =5n-4$ **=O(n)**

Exercise:

1. Write linear search algorithm. Compute running time

2. Write the algorithm for selection sort. Compute the running time.

3. Write the algorithm for bubble sort. Compute the running time.

4. write the algorithm to find minimum distance between two elements in an array. Compute the running time.

5. write the algorithm to multiply two nXn matrices. Compute the running time.

Analysis of Recursive algorithm

To analyze the time complexity of a recursive algorithm, first we need to obtain the recurrence from the algorithm. **A recurrence** is an equation that describes function in terms of its value on smaller input. Following are some example of recurrences.

1. $T(n) = \begin{cases} 1, & \text{if } n=1 \\ 2T(n/2)+1, & \text{otherwise.} \end{cases}$
2. $T(n)=T(n/4)+1$
3. $T(n)=T(n/3)+T(2n/3)+n$

To obtain the recurrence, we need to identify the running time of the statement under base condition and we need to know the number of recursive call and the size of each sub-problem for each recursive call.

Example: The recursive algorithm to find the sum of all elements of an array.

Algorithm: `recursiveAdd (A, n)`

```
//This algorithm recursively add each element of the
array A[1….n] and n is the size of array.
```

		n=0	n>0
1. if n = 0	.....	1	1
2. return 0	.....	1	--
3. return A[n]+RecursiveAdd(A,n-1)	.....	--	1 + T(n-1)
	T(n) =	2	2 + T(n – 1)

Now the recurrence for above algorithm is obtained as follows.

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

Now the solving the recursive part of the equation using **iteration method**,

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

$= 4 + T(n-2)$ . . . . step 2 substituting T(n-1) with 2 + T(n-1-1)

$= 6 + T(n-3)$ . . . . step 3 substituting T(n-2) with 2 + T(n-2-1)

Now we can guess not only the next step but he emerging step also.

$= 2 \times i + T(n-i)$ . . . step i

When i =n the base condition meets.

$= 2 \times n + T(n-n)$

$= 2n + T(0) = 2n + 2$ **=O (n)**

Exercises:

1. Write a recursive algorithm to find factorial of a number. derive recurrence and find the time complexity,

2. Write the recursive algorithm for binary search and find the time complexity.

3. Write an algorithm to find nth number in a Fibonacci series.

4. Solve the recurrence T(n)= $\begin{cases} 1, & \text{if } n = 1 \\ 2T(n/2) + n, & \text{if } n > 0 \end{cases}$

5. write algorithm to solve Tower of Hanoi problem

Other Methods to solve recurrences

1. Iteration Method (See above)
2. Master method
3. Recursion tree
4. Substitution Method

2. Master method:

This method is used to solve the recurrences that have the following form.

$T(n) = aT(\frac{n}{b}) + f(n)$ where a≥1 , b>1 are constant

Above recurrence says that, the problem is divided into 'a' number of sub-problems, each of the sub-problem has size (n/b). The cost of the problem is described by f (n).

Master theorem: *Let T (n) be defined on non=negative integers by the recurrence* $T(n) = aT(\frac{n}{b}) + f(n)$ *where a ≥ 1, b > 1 are constant .f(n) be a function and (n/b) may be* $\lfloor n/b \rfloor$ *or* $\lceil n/b \rceil$*. Then T (n) is bound asymptotically on*

case -1 : If f(n) = O ($n^{(\log_b a)-k}$) for some constant k > 0. Then

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

case-2 : If f(n) = $\Theta(n^{\log_b a})$ Then

$$T(n) = \Theta(n^{\log_b a} \times \log n)$$

case-3 : If f(n) = $\Omega(n^{(\log_b a)+k})$ for some constant k >0 and $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant c<1 and for sufficiently large n. Then

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

Example-1: Solve $T(n) = 2T(n/2) + n$

Solution: Here, a = 2, b = 2, f (n) =n.

$n^{\log_b a} = n^{\log_2 2} = n$

Given f(n) = n, which is $\Theta(n^{\log_b a})$, So case-2 of master theorem is applicable.

By case-2, $T(n) = \Theta(n^{\log_b a} \times \log n) = \Theta(n^{\log_2 2} \times \log n) = \Theta(n \log n)$

Example-2: Solve $T(n) = 4T(n/2) + n$

Solution: Here, a = 4, b = 2, f (n) =n.

$n^{\log_b a} = n^{\log_2 4} = n^2$

Given f(n) = n = $O(n^{\log_b a - k})$, So case-1 is applicable.

By case-1 , $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

Example-3: Solve $T(n) = 3T(n/4) + n\log n$

Solution: Here, a = 3, b = 4, f (n) =nlogn.

$n^{\log_b a} = n^{\log_4 3} = n^{0.79}$

Given, f (n) = nlogn = $\Omega(n^{\log_b a - k})$,

So here, case-3 may be applicable, subject to the regularity condition

$a \cdot f(n/b) \le c \cdot f(n)$ for any value of c < 1.

So , $a \cdot f(n/b) = 3 \cdot \frac{n}{4} \log \frac{n}{4} \le \frac{3}{4} n \log n \le c \cdot n \log n,$

where, f (n) = nlogn and c =3/4, which is <1. Therefore, case 3 is applicable.

By case-3, $T(n) = \Theta(f(n)) = \Theta(n \log n)$

Exercises:

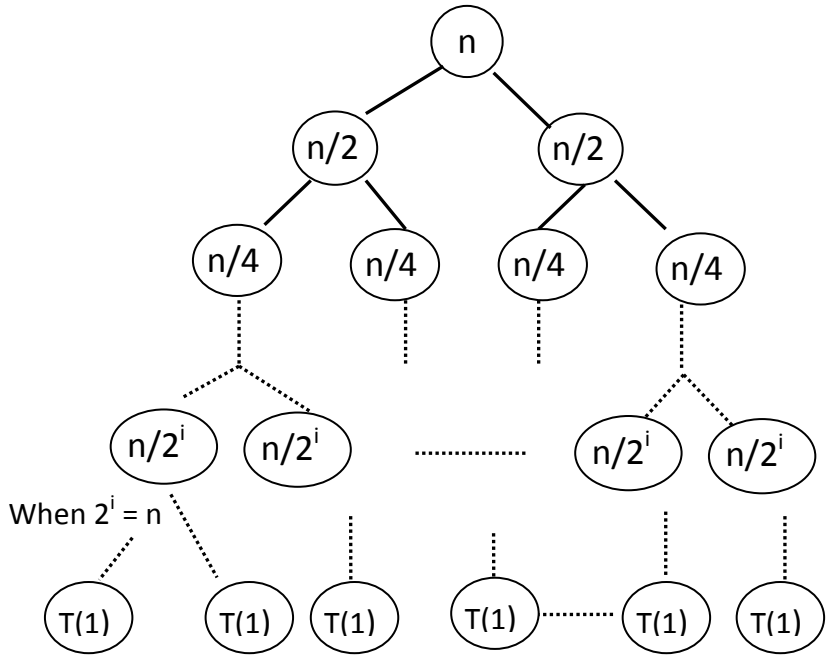
1. T(n) = 2T(n/2) +1
2. T(n) = T(3n/4) +1
3. T(n) = 9T(n/3) + n
4. T(n) = T(2n/3) + n
5. T(n) = 2T(n/2) + nlogn

3. Recursion Tree:

Recursion tree is used to generate an accurate guess to the form of solution for a recurrence and this guess can be verified by using mathematical induction.

In this method, a tree is to be made where, each node represent the cost of the single sub-problem in recursive function. Then the cost of all the nodes are added level wise to obtain per-level cost. Then all per-level costs are added to determine the total cost of the problem.

Example: Solve the recurrence $T(n) = 2T(n/2) + n$



Level	No.of nodes	cost
0	1	n
1	2^1	n
2	2^2	n
i	2^i	n
i	2^i	$2^i \times T(1)$

Adding the cost,

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

$$= 2^{\log_2 n} \times 1 + in = n^{\log_2 2} + \log_2 n \times n = n + n\log_2 n = O(n\log_2 n)$$

Exercises:

1. $T(n) = 3T(n/4) + cn^2$
2. $T(n) = T(n-1) + cn$
3. $T(n) = T(n/10) + T(9n/10) + O(n)$
4. $T(n) = T(n/3) + T(2n/3) + cn$

4. Substitution Method:

It involves in to steps.

Step-1 : Guess a form of the solution

Step-2: Using mathematical induction, show that the guess is correct.

Example: Solve the recurrence T(n) = $\begin{cases} 2, & \text{if } n = 0 \\ 2T(n-1)+1, & \text{if } n > 0 \end{cases}$

Solution:

Step-1: Make an intelligent guess by computing few steps.

n	T(n) = 2T(n – 1) + 1	
0	0	2^0-1
1	1	2^1-1
2	3	2^2-1
3	7	2^3-1
:	:	:
n	2^n-1	<- Guess

So our guess is T(n) = 2^n-1 for n ≥ 0.

Step-2: Using mathematical induction, prove that the guess is correct.

The guess is correct for n= 0,1,2,. . . .

Let it be correct for n = k. i,e T(k) = **2^k-1** .

Then we have to prove that it is also correct for n= k+1, i,e **T(k+1) = 2^{k+1}-1**

Given , T(n) = 2T(n-1) +1

T(k+1) = 2T(k+1-1) +1 = 2T(k) + 1 = 2(2^k-1) + 1 = 2^{k+1} – 1 . Hence proved

Therefore, the guess is correct. So T(n) = 2^n-1 = O(2^n)

Exercises:

1. For the recurrence T(n) =2T(n/2) + n ,prove that the solution T(n) = O(nlogn)
2. Show that T(n/2) +1 is O(logn)
3. Solve $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$