13/12/2024.

DAA:

performance measured by Time & space complexity.

Algorithm:

sequence of steps indicating how to solve a problem.

(or)

step by step process of solving a problem.

Algorithm -> problem -> execution -> soln.

why Algorithm:

To easily solve a prblm.

characteristics / Properties: we need them to write an Algorithm.

1) Input to prblm

Ex: sorting

1) List of values
2) Type of values (easier if it is same data type)

2) Output of the algorithm

3) Definiteness - each step should be clear and unambiguous.

4) Finiteness - should have an ending point.

Ex: for(; ;) - infinite loops shouldn't be there
{
}

5) Effectiveness - should be easily convertable into any programming languages.

6) Correctness - should generate expected output.

7) Generality - should be independent of programming language & OS.

Algorithmic problem solving:

For solving we need to follow no. of steps

1) Understanding the prblm.
input and output expected

2) Assertaing the capabilities of computing device / computer system.
we focus on 1) no. of processors (1 processor - 1 program)
if 1 processor then we develop Sequential Algo
if * processors then parallel algo.
2) capacity of memory (Main memory)

3) Deciding b/w exact & approximate algorithm.

exact - give exact output

Ex: prime no's

approximate - gives approximate output.

Ex: square root (for some values)

4) Selecting suitable algorithm design method.

(among 5 methods)

* Solns - Greedy & dynamic method.

* Solns & need to find all - Backtracking

5) Design / develop the algorithm & deciding the datastructures.

Select datastructure which gives better efficiency.

6) Specifying the algorithm.

Diff representations can be used lik flow chart (Drawback: not suitable for lot of steps) so we use pseudo code (uses C / Java)

7) Verifying correctness of algorithm.

we see the expected output if not then we go to step 4.

8) Analyse the algorithm.

we concentrate on 1) space 2) time complexity 3) simplicity 4) Generality.

if not satisfied track back to step 4.

9) Coding the algorithm

we write equivalent progrm for algorithm.

10) Testing (for verifying the code) & optimality (reducing time & space for expected output. for progrm).

18/12/24

Analysis of Algorithms:

for any prblm, if there exist many algorithm, to find the best (optimal) we use analysis of algorithms.

Algorithm with less time & less memory space is chosen to make some decisions about algorithms, we use analysis of algorithms.

performance measure of algorithms:-

It is based on 2 factors (parameters):-

1) Time complexity / efficiency

2) Space complexity / efficiency

Space complexity : it indicates memory requirement for the algorithm.

h) Analyze the Algorithm
Space Complexity (or) Space efficiency
Time Complexity
Simplicity
Generality
If not satisfied backtrack to step 4

i) Coding the algorithm
Write the program equivalent to the algorithm

j) Testing & Optimality
Verify applying the testing techniques
Optimality :- Reducing memory space or run time

18/12/24

Analysis of Algorithms :-

For any problem, if there exist many algorithm, to find the best (optimal), we use analysis of algorithms.

Algorithm with less time & less memory space is chosen
To make some decisions about algorithms, we use analysis of algorithms

Performance measure of Algorithms :-

It is based on 2 factors (parameters) :-

1) Time Complexity / Efficiency
2) Space Complexity / Efficiency

Space Complexity :- It indicates memory requirement for the algorithm (for its equivalent program)

Time complexity :- It indicates total CPU time required to execute the algorithm or its equivalent program.

CALCULATION OF COMPLEXITY

i) The complexity of algorithm is measured in terms of size of input to the algorithm

Eg- 1) Sorting problem :-

Input :- list of values (n)

2) Searching problem :-

Input :- list of values (n)

Target value

$\therefore$ complexity of these algorithm is calculated in terms of n

3) Matrix Addition :-

Input :- list of values $(m \times n)$

complexity is calculated

ii) To calculate time complexity, identity basic operations & how many times the basic operations are carried out (executed).

Linear search - No of times, comparison is being carried on = No. of elements

Binary search

iii) Best case, Worst-case, Avg case - Time complexity

Eg:- Searching :- Linear Search

Bestcase - $O(1)$

Avg - $O(\frac{n}{2})$

Worst - $O(n)$

CALCULATE THE SPACE COMPLEXITY OF AN ALGORITHM

Space Complexity = Space for fixed part of the algorithm + Space for variable part of the algorithm

Fixed part :- for storing
1) Code of the algorithm.
2) Simple & local variables — Which can hold one value at a time
3) Defined constants

Variable part :- 1) Variable whose size varies from one particular instance of algorithm to another instance. (like lists, arrays)
2) Global variables
3) Recursive stack
   i) Values of formal parameters
   ii) Values of local variables
   iii) return value.

What are actual & formal parameters?
actual ↓ parameters in function invocation or calling
formal → parameters in function definition

Q) Calculate space complexity of following algorithm:- (Pseudo code format)

```
Algorithm Add(a, b)
//a & b are simple variables
{
      c = a + b;
      return c;
}
```

Space complexity = Space for fixed part + Space for variable part
= C words + 3 words + 0 words   0 + 0 + 0
= (C+3) words

For any algorithm, space complexity is always calculated in terms of words

1 word =

Space for storing code = c words

For every variable memory req = 1 word

19/12/24.

Ex 3 : Algorithm MatAdd (a, b, m, n)

// a & b are matrices of size m by n

```
{
    for (i:0; i<m; i++)  -> rows
    {   for (j:0; j<n; j++)  -> columns.
        {
            c[i][j] : a[i][j] + b[i][j];
            write c[i][j];
        }
    }
}
```

Space complexity : Fixed + variable part

4 simple variables : m, n, i, j

: Code + 4 simple variables + 0 defined variables + 3mn + 0 global variables + 0 recursive stack.

: (C + 4 + 3mn) words. For multidimensional array - multiply.

Ex 4: Algorithm Factorial(n) - Recursive Algo

// 'n' is a number (integer)

```
{ if (n::1)
      return 1;
  else
      return  n* Factorial(n-1);
}
```

Space complexity : FP + UP

so 1

: Code + n simple variable + 0 defined v/constants + 0 arrays + 0 global variables + (1 word + 0 word + 1 word) x n

as it is recursive stack.

Space for recursion stack :

: (C + 1 + 2n) words.

i) Formal parameters

1 - n

ii) local variables : 0

iii) return path - 1

Ex 5 : Algorithm RSum (a, n) -> calling the Algorithm.

// a is an array of size n

sum of values in an array.

```
{
    if (n::0)
      return 0;
    else
      return a[n] + RSum (a, n-1);
}
```

Space complexity : code + 1 simple variable (n) + 0 constants + 1 x n array + 0 global variable +

Formal parameters - parameters used in definition.

⇒ only starting address of array is stored.

+ (1 for n + 1 for starting address + 0 local variable + 1 return value) (n+1)

For recursion stack.

= c + 1 + n + 3(n+1) words

= c + 4n + 4 words.

Time complexity of an Algorithm:

Total CPU time required for execution of an algorithm.

2 methods to calculate : 1) counting method / step count method.

2) Tabular / Frequency method.

Counting method:

global variable count = 0

After every executable stmnt count++

Ex 1 : Algorithm sum(a, n)

// a is an array of size n. count = 0

```
{   s = 0; -> count = count + 1;
    for (i = 0; i<n; i++) -> count = count + 1;
    {   s = s + a[i]; -> count = count + 1;
    }
    write s; -> count = count + 1;
}
```

s = 0 executed only once

Time complexity = 1 + n + 1 + n + 1

= 2n + 3

Time complexity is calculated here in terms of size.

Ex 2 : Algorithm MatAdd (a, b, m, n)

```
{ for (i = 0; i<m; i++) -> count = count + 1;
  { for (j = 0; j<n; j++) -> count = count + 1;
    {  c[i][j] = a[i][j] + b[i][j]; -> count = count + 1;
       write c[i][j]; -> count = count + 1;
    }
  }
}
```

Time complexity = (m+1) + (n+1)m + mn + mn

= m + 1 + mn + m + mn + mn = 3mn + 2m + 1

Ex 3 : Algorithm matmul(a, b, m, n)

```
{ for (i:0; i<m; i++)
  { for (j:0; j<n; j++)
    { c[i][j] :0;
      for (k:0; k<m; k++)
      { c[i][j] : c[i][j] + a[i][k] * b[k][j];
      }
      write c[i][j];
    }
  }
}
```

Time complexity : $(m+1) + (n+1)m + mn + mn(m+1) + m^2n + mn$

$: m+1 + mn + m + mn + m^2n + mn + m^2n + mn$

$: 2m^2n + 4mn + 2m + 1$

20/12/24.

Ex 4 : Algorithm uit()

3 executable statements.

```
{ for(i:1; i<:n; i++)                 - n+1 times
                   j:j*2
  { for (j:1 ; j<:n ; j++)            - n[(log_2 n + 1) + 1] times
    { write "UIT-AP" ;  n(log_2 n + 1) times
    }
  }
}
```

$n : 10$

$1 + 1 + 1 + 1 + 1$

$④ + 1$

$\log_2^n + 1 + 1$

↳ cause j is incrementing by multiplication

Time complexity :

$: n+1 + n[(\log_2 n + 1) + 1] + n(\log_2 n + 1)$

Ex 5 : Algorithm uit()

3 executable statements.

```
{ for (i:n/2 ; i<:n; i++)              - n/2 + 1 + 1
                                          True   False
  { for (j:1 ; j<:n ; j:j*2)           - [(log_2 n + 1) + 1](n/2 + 1)
    { write "UIT-AP" ;                 - (n/2 + 1)(log_2 n + 1)
    }
  }
}
```

Time complexity : $[(n/2 + 1) + 1] + (n/2 + 1)[(\log_2^n + 1) + 1] + (n/2 + 1)(\log_2^n + 1)$

Ex 6 : Algorithm uit()

```
{ for(i:1 ; i<:n ; i++)            - n+1        , if increments by multiplication/
                                                  division.
  { for(j: n; j>:1; j:j/2)         - [(log_2 n + 1) + 1]n
    {
```

write "UIT- AP" $\quad - n\,(\log_2^n + 1)$
}
} }

Time complexity : $(n+1) + n\,[(\log_2^n + 1) + 1] + n\,(\log_2^n + 1)$

Ex 7 : Algorithm Armstrong(n)

```
{ // n is an integer number.
    s := 0;
    m := n;
    while(n>0)
    {   r := n%10;
        s := s + r*r*r;
        n := n/10;
    }
    if (s := m)
        write "yes";
    else
        write "no";
}
```

k : no. of digits in number

Time complexity :
$1 + 1 + (k+1) + k + k + k + \underline{1} + 1$
$= 4k + 5$

for if condition.

Recursive Algorithm : we need to consider 2 cases

Ex : Algorithm Factorial (n)

```
{ // n is a number
    if (n:=1)
        return 1;
    else
        return n* factorial (n-1);
}
```

$n = 153$
$r = 3$
$s = 0 + 3^3 = 27$
$n = 15$

$r = 5$
$s = 27 + 5^3 = 27 + 125$
$s = 152$
$n = 1$

$r = 1$
$s = 152 + 1^3 = 153$
$n = 0$

Time complexity :

case 1 : For terminating condition.

$n = 1$

if return stmnts.

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

Time complexity

case 2 : $n > 1$

$\rightarrow$ Algorithm is again invoked for $n-1$ times.

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

we need to repeat for $n-1$ times

$T(n) = 2 + T(n-1)$
$= 2 + 2 + T(n-2) = 2 + 2 + 2 + T(n-3) = 2 + 2 + 2 + 2 + T(n-4)$
$\vdots$ after $n-1$ times,
$2 + 2 + \dots + T(n-(n-1))$

$: 2+2+2+\cdots+T(1) \quad : \quad 2+2+\cdots 2$

$: 2*n$

$T(n) : 2n$

Ex: Algorithm RSum(a,n)

// a is array of size n.

```
{   if (n::0)
      return 0;
    else
      return a[n] + RSum(a, n-1);
}
```

case 1: $n:0$

$T(n) : T(0) : 1+1 : 2$

case 2: $n>0$

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

$: 2+T(n-1)$ — we need to repeat this for n times as n should be 0.

$T(n) : 2(n+1)$.

03/01/2025

Ex: Algorithm uit() - non recursive Algorithm.

```
{ for (i:1; i<:n; i++)              - n+1
  { for (j:i+1; j<:n; j++)          - (n(n+1))/2
    { write "UIT-AP";               - (n-1)+(n-2)+(n-3)+...+1
    }
  }
}
```

$n+(n-1)+(n-2)+(n-3)+\cdots+1$

$T_c : n+1+n+(n-1)+(n-2)+\cdots+1+(n-1)+(n-2)+\cdots+1$

Ex: Algorithm uit( )

```
{ for (i:1; i<:n; i++)              - n+1
  { for(j:n; j>:i+1; j--)
    { write "UIT-AP"
    }
  }
}
```

we get same time complexity as above.

Ex: Algorithm vit( )

{ for (i = 1; i <= 50; i++) $\quad - 50 + 1 = 51$

{ for (j = 1; j <= n; j = j*2) $\quad = \{[\log_2^n + 1] + 1\} 50$

{ for (k = n; k >= 1; k = k/2) $\quad = \{(\log_2^n + 1)[(\log_2^n + 1) + 1]\} 50$

{ write "VIT-AP"; $\quad = [(\log_2^n + 1)(\log_2^n + 1)] 50$

} }

} }

$$TC = 51 + 50\{(\log_2^n + 1) + 1\} + 50\{(\log_2^n + 1)[(\log_2^n + 1) + 1]\} +$$
$$50[(\log_2^n + 1)(\log_2^n + 1)]$$

Ex: Algorithm Recursive (n) - For recursive consider 2 cases.

// 'n' is an Integer number.

{ if (n == 1)

return 1;

else

return n* Recursive (n/2);

}

case 1 : n == 1

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

parameter of algorithm.

case 2 : n > 1

$T(n) = 1 + 1 + T(n/2)$ → For recursive part.

if stmt, return in else part

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

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

$= 2 + 2 + 2 + T(n/8)$

$= 2 + 2 + 2 + \cdots$

: After $\log_2^n + 1$ times

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

$= 2 + 2 + 2 + \cdots 2 + 2.$

$T(n) = 2(\log_2^n + 1)$

08/01/2025

Ex: Algorithm Fibonacci (a,b,n)

// 'a' & 'b' are two previous fibonacci series

// n is numbers that we want to display

```
{   if(n>0)
    { c = a+b;
      write c;
      a = b;
      b = c;
      Fibonacci(a,b,n-1);
    }
}
```

Time complexity:

case 1: $n=0$

$T(n) = T(0)$: only if stmnt is executed

$= 1$

case 2: $n>0$

$T(n) = 1\ (\text{if}) + 1+1+1+1+T(n-1)$

$= 5+T(n-1)$

$= 5+5+T(n-2)$

$= 5+5+5+T(n-3)$

⋮ after n times

$= 5+5+5+5+\dots+T(n-n)$

$= 5+5+5+\dots+5+1$

$T(n) = 5n+1$

Tabular / Frequency method:

Algorithm	Frequency.
Algorithm sum(a,n)	
// 'a' is array of size 'n'	
{ s=0;	1
for(i=0; i<n; i++)	n+1
{ s=s+a[i];	n
}	
write s;	1
}	$2n+3$ = Time complexity $T(n)$

Generally for non recursive method & mostly we use the step count method.

Asymptotic Notations :

To represent complexity (space & time) of algorithms.

1) Big oh (O)
2) omega (Ω)
3) Theta (Θ)

Small oh & small omega are rarely used.

5 types of Asymptotic notations.

Big oh (O) notation :

if f(n) & g(n) are funxns defined in terms of n, then we can write f(n) = O g(n) if & only if there exists 2 positive constants c & n_0 such that $f(n) \le c * g(n)$ for all $n \ge n_0$.

f(n) is complexity of algorithm

g(n) is identified based on largest component in f(n).

$f(n) = 2n+3$ $\quad g(n) = n$

$f(n) = 4n^2 + 10n + 6$ $\quad g(n) = n^2$.

$f(n) = n\log_2^n + n + 20$ $\quad g(n) = n\log_2^n$.

Ex : Represent complexity $2n+3$ using O notation.

$f(n) = 2n+3$ $\quad f(n) \le c * g(n)$

$g(n) = n$ $\quad 2n+3 \le c * n.$ $\quad$ c should be min 3.

$2n+3 \le 3n.$ $\quad$ n should start from 3 any value should satisfy the condition.

$n_0 = 3$ $\quad$ n_0 depends on c.

$f(n) = O(g(n))$

$O(n) = 2n+3.$

Represent a complexity $10n^2 + 4n + 6$ using O notation.

$f(n) = 10n^2 + 4n + 6$ $\quad g(n) = n^2$. $\quad$ n_0 depends on c.

$f(n) \le c * g(n)$ $\quad 10n^2 + 4n + 6 \le n^2 * c.$

$c = 11$

$10n^2 + 4n + 6 \le 11n^2.$

$n_0 = 6.$

$O(n^2) = 10n^2 + 4n + 6.$

09/01/2025

Ex: Represent the complexity $6*2^n+n^2$ using Big oh notation.

$f(n) \le c \cdot g(n)$ $f(n) = O(g(n))$

c & n are +ve constants.

$f(n) = 6*2^n+n^2$

$g(n) = 2^n$

$f(n) \le c \cdot g(n)$

$6*2^n+n^2 \le c \cdot 2^n$

$c = 7$ (min)

$n = 1, 2, 3, 4, 5, 6, 7$

$6*2^n+n^2 \le 7 \cdot 2^n$

starting from that value n should satisfy for all the values.

$n_0 = 4$.

$f(n) = O\ g(n)$

$6*2^n+n^2 = O(2^n)$.

Ex: Represent the complexity $n\log_2^n+n+10$ using Big oh notation.

$f(n) = n\log_2^n+n+10$ $g(n) = n\log_2^n$.

$n\log_2^n+n+10 \le c \cdot n\log_2^n$.

$c = 2$ (min)

$n\log_2^n+n+10 = O(n\log_2^n)$

$n\log_2^n+n+10 \le 2 \cdot n\log_2^n$.

$n_0 = 7$

Omega (Ω) notation:

If f(n) & g(n) are funxns defined in terms of n, then we can write $f(n) = \Omega(g(n))$ if & only if there exist 2 positive constants c & n_0 such that $f(n) \ge c*g(n)$ for all $n \ge n_0$.

g(n) - biggest value in f(n) → complexity.

Ex. $2n+3 = O(n)$

$f(n) = 2n+3$ $g(n) = n^2$.

$f(n) \le c*g(n)$.

$2n+3 \le c*n^2$.

$c = 3$

$2n+3 \le 3n^2$.

$n_0 = 2$.

$f(n) = O(g(n))$

$2n+3 = O(n^2)$.

$g(n) = n^3$

$f(n) \le c*g(n)$

$2n+3 \le c*n^3$

$c = 2$

$2n+3 \le 2n^3$

$n_0 = 2$

$2n+3 = O(n^3)$.

There exists no. of ways to represent the complexity in O. Correct representation is least value.

Ex: $f(n) : 2n+3$

$f(n) \geq c * g(n)$.

$g(n) : n$

$2n+3 = \Omega(n)$

$2n+3 \geq c * n$

$c = 2$ (or 1).

$2n+3 \geq 2n$

$n_0 : 1$

Ex: $f(n) : 10n^2 + 4n + 6$

$g(n) : n^2$

$10n^2 + 4n + 6 \geq c * n^2$

$c : 1$ (1 to 10)

$10n^2 + 4n + 6 \geq n^2$

$n_0 = 1$

$10n^2 + 4n + 6 = \Omega(n^2)$.

Ex: $f(n) : 6 * 2^n + n^2$.

$g(n) : 2^n$

$6 * 2^n + n^2 \geq c * 2^n$

$c :$ (1 to 6) 6

$6 * 2^n + n^2 \geq 6 * 2^n$

$n_0 : 1$

$6 * 2^n + n^2 = \Omega(2^n)$.

Ex: $n\log_2^n + n + 10 : f(n)$

$g(n) : n\log_2^n$

$n\log_2^n + n + 10 \geq c * n\log_2^n$.

$c : 1$

$n\log_2^n + n + 10 \geq n\log_2^n$

$n_0 : 1$

$n\log_2^n + n + 10 = \Omega(n\log_2^n)$.

Ex: $f(n) : 2n+3$ $g(n) : 1$.

$2n+3 \geq c * 1$

$c :$ 1 to 2n

$: 1$

$2n+3 \geq 1$

$n_0 : 1$

$2n+3 = \Omega(1)$

In omega, out of all possibilities we select the highest one.

$10n^2 + 4n + 6 = \Omega(n)$

$10n^2 + 4n + 6 = \Omega(1)$

these are also possible.

Theta (θ) notation:

If $f(n)$ & $g(n)$ are constants defined in terms of n then we can write $f(n) = \theta(g(n))$, if & only if there exists 3 +ve constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$.

O & Ω combination.

Ex: $f(n) : 2n+3$

$g(n) : n$

$c_1 * n \leq 2n+3 \leq c_2 * g(n)$

$c_1 : 1$ & $c_2 : 3$

$n \leq 2n+3 \leq 3n$

$n_0 : 3$

$2n+3 = \theta(n)$

10/01/2025

$g(n) : n^2$

$C_1 * g(n) \le f(n) \le C_2 * g(n)$

$C_1 * n^2 \le 2n+3 \le C_2 * n^2$ × not possible.

* It is not possible to represent theta in no. of representations.

$f(n) = 10n^2 + 4n + 6$

$g(n) : n^2$

$C_1 * n^2 \le 10n^2 + 4n + 6 \le C_2 * n^2$.

$C_1 : 1$ to $10 \Rightarrow 10$

$C_2 : 11$

$10n^2 \le 10n^2 + 4n + 6 \le 11n^2$

$n_0 : 6$

$10n^2 + 4n + 6 : \theta(n^2)$.

$f(n) : 6 * 2^n + n^2$.

$g(n) : 2^n$

$C_1 * 2^n \le 6 * 2^n + n^2 \le C_2 * 2^n$

$C_1 : 1$ to $6 \Rightarrow 6$

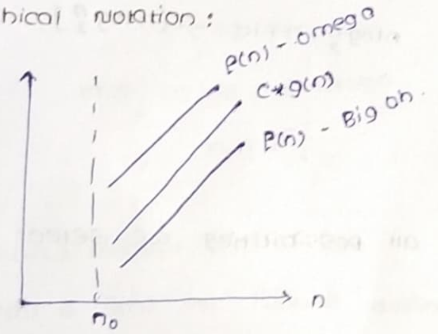
$C_2 = 7$

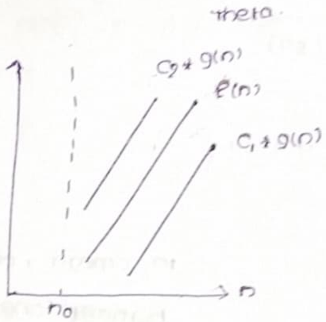
$6 * 2^n \le 6 * 2^n + n^2 \le 7 2^n$.

$n_0 : 4$.

$6 * 2^n + n^2 : \theta(2^n)$.

Graphical notation :





Show that $2n^3 + 4n^2 + 6 : \theta(n^3)$

$f(n) : 2n^3 + 4n^2 + 6$ $\quad g(n) : n^3$

$C_1 * n^3 \le 2n^3 + 4n^2 + 6 \le C_2 * n^3$

$C_1 : 1$ to $2 : 2$

$C_2 : 3$

$2n^3 \le 2n^3 + 4n^2 + 6 \le 3n^3$

$n_0 : 3$ (It can be represented).

Show tha $3^n \neq O(2^n)$

$f(n) : 3^n$ $\quad g(2^n)$

$f(n) \le C * g(n)$ $\quad 3^n \le C * 2^n$

no C & n can satisfy the above eqn.

methods to solve Recurrence relation :

1) Substitution method

2) master's theorem

3) Recursion Tree method.

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

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

$= 2 + 2 + 2 + T(n-3)$ ⇒ Substitution method of factorial.

⋮ After n-1 times,

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

$= 2n$

master's theorem :

$T(n) = aT(n/b) + n^k \log^p n$

$a \geq 1$, $b > 1$, $k \geq 0$, ~~p≥0~~, p is a real number. (both +ve & -ve values)

compare a with b^k.

case 1 : $a > b^k$

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

case 2 : $a = b^k$

$p < -1$ then $T(n) = \theta(n^{\log_b^a})$

$p = -1$ then $T(n) = \theta(\log^2 n \cdot n^{\log_b^a})$

$p > -1$ then $T(n) = \theta(n^{\log_b^a} \cdot \log^{p+1} n)$

case 3 : $a < b^k$

$p < 0$ $T(n) = \theta(n^k)$

$p \geq 0$ $T(n) = \theta(n^k \log^p n)$

Solve the following Recurrence relation :

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

$\log_n^p = 0$

$a = 4$, $b = 2$, $k = 2$, $p = 0$

$b^k = 2^2 = 4$

$a = b^k$ $p > -1$ then $T(n) = \theta(n^{\log_b^a} \cdot \log^{p+1} n)$

$= \theta(n^{\log_2^4} \cdot \log^{0+1} n)$

$= \theta(n^2 \cdot \log^1 n) = \theta(n^2 \cdot \log n)$

16/01/2025

Ex:- $T(n) = 3T(n/2) + n^2$

$a = 3$, $b = 2$, $k = 2$ $p = 0$

$b^k = 2^2 = 4$

$a < b^k$ & $p = 0$ then $T(n) = \theta(n^2 \log_n^0) = \theta(n^2)$.

Ex: $T(n) = 2T(n/2) + n\log n$

$a = 2 \quad b = 2 \quad k = 1 \quad p = 1$

$a = b^k$

$T(n) = \theta(n^{\log_2^2} \cdot \log^{1+1} n)$

$\therefore \theta(n \cdot \log^2 n)$.

Ex: $T(n) = \sqrt{2} \cdot T(n/2) + \log n$

$a = \sqrt{2} \quad b = 2 \quad k = 0 \quad p = 1$

$a > b^k$

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

$= \theta(n^{\log_2^{\sqrt{2}}})$

$= \theta(n^{1/2}) = \theta(\sqrt{n})$

Ex: $T(n) = 2T(n/2) + n^2/\log n$

$a = 2 \quad b = 2 \quad k = 2 \quad p = -1$

$a < b^k \qquad T(n) = \theta(n^k)$

$= \theta(n^2)$

$T(n) = 3T(n/3) + n/2. \ (2^{-1} n)$

$a = 3 \quad b = 3 \quad k = 1 \quad p = 0$

$a = b^k$

$T(n) = \theta(n^{\log_3^3} \cdot \log^{0+1} n)$

$= \theta(n \cdot \log n)$

Recursion Tree method to solve recurrence relations:

Factorial (4) -> Factorial (3) -> Factorial (2) -> Factorial (1). $T(n) = C + T(n-1)$

will invoke Algorithm once.

Algorithm mergeSort (a, s, e)

```
{ if (s<e)
  { m = (s+e)/2;
    mergesort (a, s, m);
    MergeSort (a, m+1, e);
    merge (a, s, m, e);
  }
}
```

MS(1, 8)

MS(1, 4) MS(5, 8)

invokes mergesort twice.
Algorithm

2/ more invokation of Algorithm -> use Recursion Tree.

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

Solve the following using Recursion Tree method:

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

$T(n) = 2T(n/2) + n \quad = T(n/2) + T(n/2) + n$.

Recursion tree:

consider non recursive part of relation as root node.

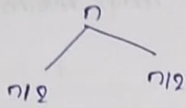
n

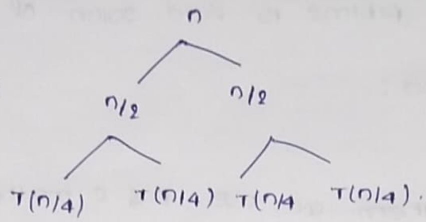
T(n/2) T(n/2)

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

Replace T(n/2) with its equivalent value.

$T(n) = 2T(n/2) + n \qquad T(n/2) = 2T(n/2^2) + n/2$

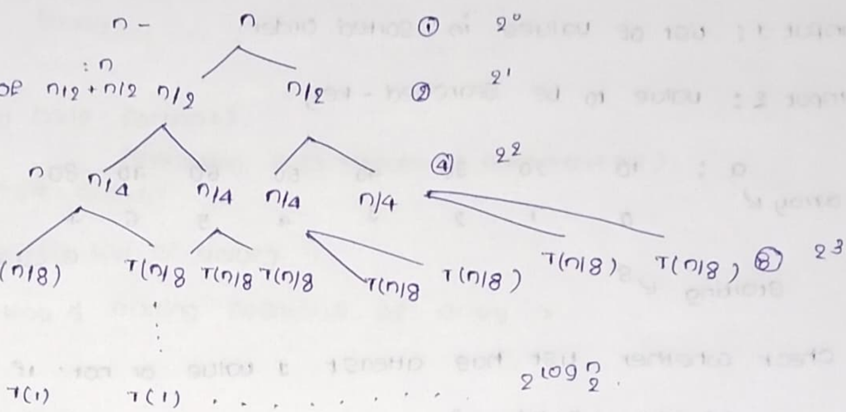




After equivalent part place non recursive part

$T(n/4) = 2\,T(n/2^3) + n/2^2/4$

identify after no. of levels it read $T(1)$.



no. of levels $= \log_2^n$

calculate cost of each level.

Time Complexity

$T(n)$ = cost of internal levels + cost of last level.

$$a^{\log_c^b} = b^{\log_c^a}$$

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

cost of last level $= 1 * n = n$.

cost of internal levels

$$= \underbrace{n + n + n + \cdots n}$$

repeats for $\log_2^n - 1$

$$= n * (\log_2^n) = n\log_2^n$$

$T(n) = n\log_2^n + n$ $\quad$ $T(n) = O(n\log_2^n)$.

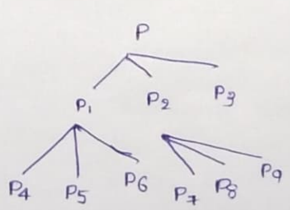
17/01/2025

MODULE - 2 (Divide & Conquer Method)

used when we can divide prblm into no. of sub prblms.

if prblm is small enough - directly solve it.

if prblm is large - it should be divided into sub prblms. & verify whether it can be directly solved.



continued until the prblm is directly solvable.

Combine solutions of sub prblms to find soln of main prblm.

Applications of D&C method :

1) Binary search

2) merge sort   To solve them we use D & C method.

3) Quick sort

Binary search : Ex, Algorithm, Time complexity.

input 1 : list of values in sorted order.

input 2 : value to be searched - key

a :	10	20	30	40	50	60	70	80
array	0	1	2	3	4	5	6	7

key = 70

starting s   e ending

Check whether list has atleast 1 value or not. if list has >1 values then find middle value.

m : (s+e)/2 : 0+7/2 : 3

key > a[m] : a[3] : 40

s : mid+1 : 3+1 : 4

m : 4+7/2 : 5

arr[m] : arr[5] : 60

key > arr[m]

s : mid+1 : 5+1 : 6.

$m : \frac{6+7}{2} : 13/2 : 6$

arr[m] : arr[6] : 70   (Target found).

no need to solve all subprblms one is enough to find whole soln.

key : 15

$mid : \frac{s+e}{2} : 0+7/2 : 3$

arr[mid] : arr[3] : 40

key < a[mid]

e : mid-1 : 3-1 : 2.

$mid : \frac{0+2}{2} : 1$

arr[mid] : arr[1] : 20

key < arr[mid]

end = mid - 1

end = 1-1 = 0

$\left\{ mid = \frac{0+1}{2} = 1/2 = 0.5 \right.$

= 0

arr[0] = 10

key > arr[mid]

there's no second part here so Target isn't found.

Algorithm: (in pseudo code format).

(indicating size through 2 parameters).

Algorithm BinarySearch (a, s, e, k)

{ // 'a' is array containing list of values.

// 's' & 'e' are starting & ending positions of array 'a'.

// 'k' is key value.

if (s>e) //list doesn't have more than 1 value.

return -1; - // indicates key is not present

else

{ m = (s+e)/2;

~~(if key = arr)~~ if (k == a[m])

return m;

else

{ if (k<a[m])

BinarySearch (a, s, m-1, k);

else

BinarySearch (a, m+1, e, k);

}

}

}

Time complexity:

case 1: s>e (terminating condition)

n = 0 (no values in list)

T(n) = T(0) = 1+1 = 2

case 2: s ≤ e.

n>0

when k == a[m]

T(n) = 1+1+1+1 = 4

when k ≠ a[m]

T(n) = 1+1+1+1+ T(n/2)

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

master's theorem

a = 1 b = 2 k = 0 p = 0

$a = b^k$

$T(n) = \Theta(n^{\log 2} \cdot \log^{0+1} n) \Rightarrow T(n) = \Theta(1 \cdot \log n)$

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

22/1/25

MERGE SORT

Input: A list of values

Algorithm MergeSort(a, s, e)

// 'a' is an array containing list of values

// 's' and 'e' are starting & ending points of an array

```
{
   if(s<e)
   {
      m=(s+e)/2;
      MergeSort(a, s, m);
      MergeSort(a, m+1, e);
      Merge(a, s, m, e);
   }
}
```

0	1	2	3	4	5	6	7
40	20	80	10	70	50	60	30

s e

$m = \frac{s+e}{2} = \frac{0+7}{2} = 3$

0	1	2	3
40	20	80	10

s to m

4	5	6	7
70	50	60	30

m+1 e

$m = \frac{s+e}{2} = \frac{0+3}{2} = 1$

$m = \frac{s+e}{2} = \frac{4+7}{2} = 5$

40	20		80	10		70	50		60	30
0	1		2	3		4	5		6	7

$\frac{0+1}{2} = 0$ $\frac{2+3}{2} = 2$ $\frac{4+5}{2} = 4$ $\frac{6+7}{2} = 6$

40 20 80 10 70 50 60 30

20	40		10	80		50	70		30	60

10	20	40	80		30	50	60	70

b=

10	20	30	40	50	60	70	80

for storing merged list

Algorithm Merge(a, s, m, e)

```
{
   i=s;
   j=m+1;
   k=s;
   while(i<=m and j<=e){
   if(a[i]<=a[j])
   {
      b[k] = a[i];
      i=i+1;
   }
   else{ b[k]= a[j];
      j=j+1;
   }
   k=k+1;
   }
   while (i<=m)
   {
      b[k] = a[i];
      k=k+1;
      i=i+1;
   }
   while (j<=e)
   {
      b[k] = a[j]
      k=k+1;
      j=j+1;
   }
   for (x=s; x<=e; x++)
   {  a[x]=b[x];
   }
}
```

1 class

23/01/2025

Time complexity of mergesort algorithm:

Time complexity of merge algorithm : use stepcount method :

$1+1+1+(n/2+1) + \underbrace{n/2 + n/2 + n/2}_{\text{if stmt}} + \underbrace{n/2}_{\text{last stmt}} + (n/2+1) + 3 \cdot n/2$

while

only if/else part any one can be added.

either one of the while loops is executed.

n → n/2, n/2 ; i -> m, j -> e

$+ (n+1) + n$

for loop

$= 6 + \frac{9n}{2} + 2n = 6 + \frac{13n}{2}$

if n = 1000, 6 is very less compared to n. so omit it

$= 13/2\, n$

$= cn$

easy method :

n → n/2, n/2

max. no. of comparisions $= n/2 + n/2$

$= n$

Time complexity of Mergesort Algorithm: step count method :

$1+1 + T(n/2) + T(n/2) + cn. \quad = 2 + 2T(n/2) + cn.$

if

∵ 2 is very small omit it

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

compare it with master's theorem

$a = 2 \quad b = 2 \quad k = 1, \; P = 0$

$a = b^k$:

$T(n) = \Theta(n\log_2^2 \cdot \log^1 n) \quad = T(n) = \Theta(n\log_2 n)$.

Quick sort :

Arranges lists of values in ascending order.

	P						
a :	40	20	70	10	50	30	60
	0	1	2	3	4	5	6
	i						

// store values into array

if list has > 1 value then we start sorting.

select any element as pivot but mostly 1st value.

$a[i] \le P$ then move i towards right by 1.

move until $a[i] > P$

a[j] > P then move j by 1 to left

stop it when a[j] < P

verify relation b/w i & j

if i <= j then a[i] swap a[j]

if i > j then swap Pivot with a[j]

P						
40	20	70	10	50	30	60
0	1	2	3	4	5	6
i						j

i i then move j

j

now i <= j 2 <= 5

P						
40	20	30	10	50	70	60
0	1	2	3	4	5	6

i →

i →

i

j swap pivot with j

			P			
10	20	30	40	50	70	60
0	1	2	3	4	5	6
			j	i		

Based on j divide list into 2 parts until j-1

P
10 0 i→ 20 1 30 2 j

i

j

j swap pivot with a[j]

40 3

P
50 4 i j 70 5 60 6 j

i

P
20 1 i j 30 2 j

P
70 5 i 60 6 j

i

j

i i>j

60 5 P 70 6

60 5

Algorithm QuickSort (a, s, e)

// 'a' is array containing list of values.

// 's' & 'e' are starting & ending positions of array 'a'.

```
{   if (s<e)
    {
        j = partition (a, s, e);
        QuickSort (a, s, j-1);
        QuickSort (a, j+1, e);
    }
}
```

Algorithm partition (a, s, e)

```
{   p = a[s];
    i = s;
    j = e;
->  while (i<=j) {
      while (a[i] <= p)
          i = i+1;

      while (a[j] > p)
          j = j-1;

      if (i<=j)
      {   t = a[i];
          a[i] = a[j];
          a[j] = t;
      }
    } <-
    t = a[s];
    a[s] = a[j];
    a[j] = t;
    return j;
}
```