Design and Analysis of Algorithms (DAA)

Algorithm – sequence of steps.

Design – developing algorithms

Analysis – Measure Performance
↳ of algorithms

2 parameters → space → = space Complexity
→ Time → = Time Complexity

Text Books

1) Introduction to Design and Analysis of Algorithms (Levitin)
2) Fundamentals of computer Algorithms (Sahani, Horowitz)

Mod-1

1) Def of algorithm what is algorithm? purpose of alg
2) Charecteristics/properties of algorithm → (Algorithmic problem solving)
3) Measure performance of alg — space, time complexities
procedures for calculating
4) Representing the complexity values
ASymptotic notations (O, Ω, Θ)

Recurrence relations – methods to solve

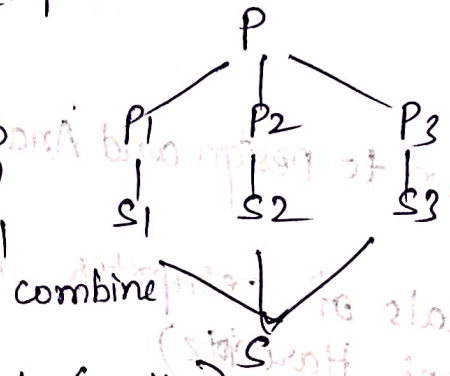
Mod-2 to Mod-6

Algorithm Design Methods

1) Divide and Conquer — Mod-2
2) Greedy
3) Dynamic Programming — Mod-3
4) Backtracking — Mod-5
5) Branch and Bound — Mod-6

I) Divide and Conquer ⇒ applied to prob which gets — one soln / one sorting order

(smaller sub prob)



Adv

1) Big prob to no of sub prob (smaller)
2) single prob — more time, solve sub prob concurrently (less time)

*) Binary Search
*) QuickSort
3) Merge Sort

II) Greedy — is applied to prob which has no of soln's → identify best soln (optimal)

1) shortest path problem (Graph) — (least cost path from no of soln)
2) Minimum cost Spanning Tree

Adv — no need to find all possible soln
only find best soln (solve), quick

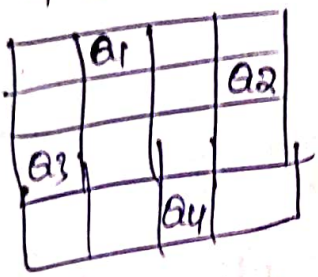
Disadv — fails sometimes in finding optimal soln

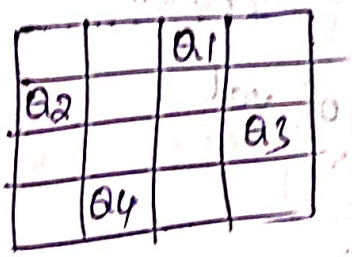
III) Dynamic Programming — when prob has no of soln's
(generating all pos soln and then selects best soln)
(compares)
Drawback

Adv - guarantees generation of optimal soln

IV) Backtracking — when problem has no of soln's
Finds all possible solutions

1) N-Queens
4-Queens





2) Graph Coloring problem
Call diff ways to color the graph

V) Branch and Bound — (prob which has no of soln's)
To identify best
(generates all soln — while generating we check with the bound, if it is not best we stop generating that soln)

14/12/24

Module-1

Algorithm

An Algorithm is sequence of steps which indicates how to solve a problem.

Need of Algorithm

Algorithm → program (built based on alg) → executed → solves a problem

Properties / Characteristics of an Algorithm (7)

1) Input (parameters)
   ex:- sorting — list of val
   prime no — integer

2) Output

3) Definiteness
   (each step of alg should be clear and unambiguos)
   1) Add 5 or 6 to 10 — ambiguous

4) Finiteness
   (should end after finite no of steps)
   should not be in an infinite loop

5) Effectiveness
   Each step can be easily converted into a program

6) Correctness
   after executing correct output

7) Generality
   steps of algorithm should be independent of
   - PL (programming language)
   - OS (operating system)

Algorithmic problem solving / Steps to be followed to develop an algorithm.

1) Understanding the problem
   (input, output)

2) Ascertaining the capabilities of computing device / computer system
   1) No of processors / CPU's
   2) Storage Capacity
   ↓ will indicate
   no of programs can be executed concurrently

① 4 processors – 4 pgms at a time can be executed
   ↓ (parallel algorithms)
   1 processor – 1 pgm at a time executed
   ↳ (sequential algorithms)
   ↓ step by step

② Storage capacity (space available in RAM)
   ↓ assess space in main memory before loading pgm from sec memory to main memory

3) Deciding between Exact and Approximate Algorithm
   exact o/p — approx o/p

Exact
   prime no — yes / no

Approx
   square root — (√3) — 1.73 approx o/p not exact o/p

4) Deciding Algorithm design method — Div & Conq, Greedy, DP, BT, B&B

5) Design the Algorithm & Decide the Data Structures

6) Ex:- sort list of values
   Mergesort, Quicksort } Div & Conq — arrays, list (ArrayList)
   ↓ best less time

6) Specifying the Algorithm

(1) Flowcharts — less no of steps (Representing Algorithm)

(2) Pseudocode — java code format

(In this format steps should be similar to pgm)

7) Verifying the correctness of Algorithm

correct o/p

↓ manually tracing the steps

✓ → next step → correct o/p

× (wrong o/p)

↓

~~Move~~ backtrack to step 4 select diff method & proceed

8) Analyze the Algorithm

1) Space Complexity
2) Time Complexity
3) Simplicity — steps are simple or not
4) Generality — follows generality principles/ property (indp of PL, OS)

follows ✓ → next step

↓

× → step (4)

takes more time / more space

9) Coding the Algorithm

(equivalent pgm in any programming language)

10) Testing

sample test cases → output check

18/12/24

Analysis of Algorithms

to make decisions regarding algorithm
(To choose best alg) → (less time, less space)

Performance Measurement of algorithms
1) Space Complexity / space Efficiency — Memory space required by the alg/pgm
2) Time Complexity / Time Efficiency — CPU time required by the alg/pgm

Imp pts while cal SC or TC
1) Complexity of algorithm is calculated interms of size of input to the algorithm. — (n-size the TC, SC cal interms of n)
2) Time Complexity — identify basic operations — no of times they are executed

Searching — list of val, Key val
(Linear/Binary search)
basic op ↓ comparision → executed n times

3) Cal Time Complexity using / by 3 cases — for same alg
Best Case
Average Case (Avg no of steps to get o/pt)
Worst Case

Searching problem
↓
list of values
Key
↓
Linear search

Best case — 1 comp operation
Worst Case — n
Avg case — $1 < a < n$ at any position

4)

Calculating space complexity of an algorithm

- Space Complexity of an algorithm = Space req for storing fixed part of alg + Space req for storing variable part of alg

fixed part

1) code of alg → steps in alg
2) simple local var → storing only one val (simple)
3) defined const
   ↳ x = 100

Variable part (arrays)

1) variables where size varies from one distance to another distance of alg/pgm.
2) Global var
3) Recursion stack (stores info about fact calls)

Recursion

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

fact(3)
↓
3 * fact(2)
↓
fact(1)

* for each invocation of recursion call 3 values are stored in recursion stack → 'n'
  1) Values of formal parameters
  2) value of local variables
  3) return values

- parameters used in function calling
  actual par (fact(3))

- parameters used in f^n definition
  ↓
  formal parameters
  'n' → in factorial pgm

Ex-1: Cal space comp of foll alg ? → (start with Algorithm for writing pseudocode format)

Alg Add (a,b) → Name
// 'a' and 'b' are single variables
{
  c = a+b; — code (no need to declare var)
  write c ;
}

↓ (Pseudocode)

FP → for code memory req (1 word = 2 bytes / 4 bytes)

SC = FP + VP = C words + [simple var (a, b, c) — 3 words]
= 3 words + 0 words (no defined const) + 0 words (no global var) + 0 words (not rec) ⟹ 0
VP

SC = (C+3+0+0+0+0) / (FP + VP) = (C+3) words = SC

(C+3+n)

Ex-2: Algorithm sum(a,n)
// 'a' is an array of size 'n'
{
  s=0;
  for(int i=0; i<n; i++)
  {
    s = s+a[i];
  }
  write s;
}

SC = FP+VP
= C words + 3 words + 0 words
{n, s, i} simple var
FP
+ n words + 0 words + 0 words
VP

SC = (C+3+n) words

↓ (Non-Recursive)

→ not a simple var as we can store more val

20/12/24

Matrix addition

q) cal sc of foll alg.?

Algorithm MatAdd (a, b, m, n)

// 'a' & 'b' are matrices/arrays of size 'm' x 'n'

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

$SC = FP + VP \quad (m, n, i, j)$

$FP = 0 \text{ words} + 4 \text{ words} + 0 \text{ words} = 0 + 4 \text{ w}$

$VP = \underset{(a,b,c)}{3 \text{ words}} + 0 + 0\text{w} = 3mn\,\text{w}$ (m x n x 3)

$SC = 0 + 4 + 3mn$ words

$SC = (0 + 4 + 3mn)$ words //

* Matrix Multiplication

Non Recursive Alg. ⇒ Add of no's, ∞ sum of val in array, Matrix Add

Recursive Alg

ex:- Algorithm Factorial (n) — Factorial

```
// 'n' is an integer number
{ if(n==1) return 1;
  else return n * Factorial(n-1);
}
```

3 times

fact(3) → fact(3), fact(2), fact(1)

n times → fact(n) invoked

$3! = 3 * 2! = 3 \times 2 \times 1! = 3 \times 2 \times 1 = 6$

SC ⇒ FP + VP

$= \underbrace{C+1+0}_{FP} + 0 + 0 + (1 + 0 + 1) * n$

(no local var; 1 formal par; returning a val)

$= C+1+2n = C+2n$

~~SC = (C+2) words~~

SC = (C+1+2n) words

Ex:-

```
Algorithm RSum(a,n)
// 'a' is an array of size 'n'
{
  if(n==0) return 0;
  else return a[n-1] + RSum(a,n-1)
}
```

n=3
a[1] = 10
a[2] = 20
a[3] = 30
} 60

RSum(a,3)
↓
30 + RSum(a,2)
↓
30 + 20 + RSum(a,1)
↓
30 + 20 + 10 + RSum(a,0)
30 + 20 + 10 + 0 = 60

n=3 ↓ 4 times (n+1)

$SC = FP + VP = \underbrace{c+1}_{FP} + 0 + n + 0 + (\underbrace{n}_{n\ a} + \underbrace{0}_{local\ var} + 1) * (n+1)$

$= c+1+n+(3)*(n+1)$

$= c+1+n+3n+3$

$SC = (c+4+4n)$ words //

(n a formal par)

only stack add of array is stored (because val of arr are stored in consequent memory locations)

* Time Complexity of an Alg

(Total CPU time req to execute a pgm)

1) Counting Method
2) Tabular/Frequency Method

① Counting Method

count = 0; → global var

executable statements → identify after each exec stat increment count val by 1

TC = count → end of alg

Examples

ex-1: Cal TC of foll alg?

```
Algorithm Sum(a,n)
// 'a' is an array of size 'n'
{
  s=0 ------> count=count+1; (1)
  for(i=0; i<n; i++) ------> c=c+1; (n+1)
  {
    s=s+a[i]; ------> c=c+1; (n)
  }
  write s; ------> c=c+1; (1)
}
```

count = 0

count at end of alg

$TC = 1 + (n+1) + (n) + (1) = 2n+3$

(for stat)

ex-2:- Matrix Addition Alg

```
Algorithm MatAdd (a, b, m, n)
// 'a' and 'b' are matrices/arrays of size 'm' & 'n'
{
  for (i=0; i<m; i++)          ⟶ (m+1)
  {
    for (j=0; j<n; j++)        ⟶ (n+1) * m
    {
      c[i][j] = a[i][j] + b[i][j];   ⟶ (n) * m
    } write c;                 ⟶ (n) * m
  }
}
```

TC = (m+1) + (n+1)*m + mn + mn

= m+1 + mn + m + mn + mn

TC = 3mn + 2m + 1 //

ex-3:- Matrix Multiplication Alg

```
Algorithm MatMul (a, b, m, n)
// 'a' and 'b' are matrices/arrays of size 'm' & 'n'
{
  for(int i=0; i<m; i++)                ⟶ m+1
  {  for(int j=0; j<n; j++               ⟶ m*(n+1)
     {                                   ⟶ m*n
       c[i][j]=0;
       for(int k=0; k<m; k++)            ⟶ m*n*(m+1)
       {  c[i][j] = c[i][j] + a[i][k] * b[k][j];
                                         ⟶ m*n*(m+1)
       } write c[i][j];                  ⟶ m*n
  }
}
```

* $\underbrace{(m \times n)\ (m \times n)}_{m=n}$

$TC = m+1 + m*(n+1) + mn + (1+m)n + 1+n$

$= m+1 + mn+m+n+1+mn+1+n$

$TC = 2mn + 4m + 2n + 3$ //

21/12/24

* for (i=1; i<=n; i++) → (n+1) (true / false)
{
}

* for(i=1; i<=n; i=i*2) → $\log_2 n + 1$ (true / false)
{
}

n=8

↳ 1 + 1 + 1 + 1 → 5 times
(n=2, n=4, n=8 true; n=16 false)

$4+1 \Rightarrow 3+1+1 \Rightarrow \log_2 8 + 1 + 1$

* for (i=n; i>=1; i=i/2) → $\log_2 n + 1$ (true / false)
{
}

n=8 ⇒ 1 + 1 + 1 + 1 + 1 (n=8, n=4, n=2, n=1, n=1/2=0; true true true true false)

$= 4+1 = 3+1+1$

$= \log_2 n + 1 + 1$

n=8: 1+1+1+1+1 (5)

* for(i=n/2 ; i<=n; i++) → $\frac{n}{2}+1$ + 1
true false
{
}

$n = 8$

$\frac{8}{2} = 4$

4, 5, 6, 7, 8 + 1

4+1 (n_2+1) true — false

*
Algorithm Example()
{
for (i=1; i<=n; i++) → n+1
{
for (j=1; j<=n; j=j*2) → $n * [\log_2 n+1+1]$
{
write "VIT-AP"; → $n * [\log_2 n+1]$
}
}
}

$TC = (n+1) + n * [\log_2 n+1] + n * [\log_2 n+1]$

*
Algorithm Example()
{
for(i=1; i<=n; i++) → n+1
{
for(j=i+1; j<=n; j++) → ~~n*~~(
{
write "VIT-AP"; → n*(
}
}
}

$i=1 \Rightarrow j=2 \text{ --- } n \hookrightarrow n$

$i=2 \Rightarrow j=3 \text{ --- } n \rightarrow n-1$

$i=3 \Rightarrow j=4 \text{ --- } n \rightarrow n-2$

$\vdots$

$i=n \rightarrow j=n+1 \rightarrow 1$

$n+(n-1)+ \text{ ----- } 1$

$$TC = (n+1) + (n+n-1+n-2+ \text{ --- } 1) + (n-1+n-2+ \text{ --- } 1)$$

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

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

$$= \frac{2n^2+n+1}{2}$$

$TC \Rightarrow n^2+n+1$

TC of Recursive Algorithm:

* Ex-1: Algorithm Factorial (n)

```
// 'n' is an integer number
{
  if(n==1) return 1
      return 1;
  else
      return n * Factorial(n-1);
}
```

case 1: (n==1) → terminating condition

case 2: (n>1)

Case 1: $n=1$

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

(1 ↓ if, 1 ↓ return stat in if cond)

Case 2: $n>1$

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

(1 ↓ if, 1 ↓ return stat in else cond)

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

Recurrence Relation / Base equation

$T(n) = 2 + T(n-1)$ — 1

$= 2 + 2 + T(n-2)$ — 2

$= 2 + 2 + 2 + T(n-3)$ — 3

(n-1) times

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

(n-1) times

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

$= 2 + 2 + \dots + 2 + 2$

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

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

$$T(n) = 2n$$

//

Ex 1:

```
Algorithm Rsum(a,n)
// 'a' is an array of size 'n'
{
    if (n==0)
        return 0
    else
        return a[n]+Rsum(a,n-1);
}
```

case 1: $n=0$

$T(n) \Rightarrow T(0) = 1+1 = 2$

case 2: $n>0$

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

$= 2+T(n-1)$

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

⋮ after n times

$= \underbrace{2+2+---2}_{n\text{times}} + T(n-(n-1))$

$= \underbrace{2+---2}_{n\text{times}} + T(0)$

$= \underbrace{2+---2}_{n\text{times}} + 2$

$= 2(n+1)$

$= 2(n+1)$

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

Ex1:

```
Algorithm Recur(n)
// 'n' is an integer number
{   if(n==1)
        return 1
    else return n*Recur(n/2);
}
```

case 1: $n==1$

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

case 2: $n>1$

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

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

⋮ after $\log_2 n$ times

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

$= 2+2+---2+T(1) \rightarrow 2$

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

8/01/25

9) Cal TC of foll Alg?

Algorithm Armstoong (n)
// 'n' is an integer number

```
{
    s=0;        → 1
    m=n;        → 1
    while(n>0)  → K+1
    {
        r = n%10;   → K
        s = s+r*r*r;  → K
        n = n/10;   → K
    }
    if (s==m)   → 1
        write "number is armstoong"
    else
        write "number is not armstoong"
}
```

any one write stat → (1)

(K is no of digits in 'n')

either if inside or else inside statement

Count Method

↳ TC = 1+1+ (K+1) + K+ K+K + 1+(1) = 4K+5

no of digits

n = 26437 //

(*) n=153 → 3+1

(1+1+)+ (1) ⇒ while(n>0)

True false

5times +1 time

↓

6 times

5+1

(no of digits +1)

9) Algorithm Fibonacci (a,b,n)

// 'a' and 'b' are previous two fibonacci numbers

// 'n' is the no of fibonacci numbers

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

Recursion

Case-1 : n=0

TC ⇒ n==0

TC = T(n) = T(0) = 1

Case-2 : n>0

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

T(n) = 5+T(n-1)

= 5+5+T(n-2)

after n times:

= 5+5+5+ -- -- T(0)

= 5+5+ -- -- +5

= 5n+1

= 5(n+1)

T(n) = 5n+1

SC ⇒ PP + VP

= (C+4+0) + [0+0+(3+1+0)n]

= C+4+4n

SC = (C+4+4n) words

Cal Space Complexity:

q) Algorithm vit()

```
{
  for(i=1; i<=50; i++)                    → 50+1
  {
    for(j=1; j<=n; j=j*2)                 → 50*(log2n+1+1)
    {
      for(k=n; k>=1; k=k/2)               → 50*(log2n+1)*(log2n+1+1)
      {
        write "VIT-AP";                   → 50*(log2n+1)*(log2n+1)*(log2n+1)
      }
    }
  }
}
```

TC = $50+1+50*(\log_2 n+1+1)+50*(\log_2 n+1)*(\log_2 n+1+1)$

$+50*(\log_2 n+1)*(\log_2 n+1)*(\log_2 n+1)$

~~TC = 51+~~ Tabular Method

$= 51+50\log_2 n+100+50*((\log_2 n)^2+3\log_2 n+2)$

$+50*((\log_2 n)^2+1+2\log_2 n)$

$= 151+50\log_2 n+100(\log_2 n)^2+250\log_2 n+150$

$TC = 100(\log_2 n)^2+300\log_2 n+301$

Tabular Method

Algorithm	Frequency

Algorithm Sum(a,n)
//'a' is an array of size 'n'
{
s=0 ;
for(i=0; i<n; i++)
{ s=s+a[i];
}
write s;
}
	1
	n+1
	n
	1
	2n+3

Total = TC => 1+n+1+n+1

TC = 2n+3

* step-count method is more preferred

Symptotic Notations

4/1/25

Asymptotic Notations

To Represent complexity of algorithm

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

Bigoh (O) Notation

If $f(n)$ and $g(n)$ are functions interms of n, the we can write $f(n) = O(g(n))$.

If and only if there exist two positive ~~function~~ constants c and n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$

$f(n)$ is the complexity of the algorithm

1) Identify $g(n)$ — is selected based on largest component in $f(n)$

Alg
TC = 2n+6
f(n) = 2n+6
(largest component → from this 'n' is selected)
g(n) = n

TC = $2n^2+4n+10$
$f(n) = 2n^2+4n+10$
$g(n) = n^2$

2) Identify 2 positive constants c and no satisfying $f(n) \leq c * g(n)$

3) ~~Satisfied~~ found 2nd one then we can represent in Big Oh Notation.

Ex 1: Represent the complexity of $2n+3$ using Big oh Notation

$f(n) = 2n+3$

① $g(n) = n$

② $f(n) \leq c * g(n)$

$2n+3 \leq c * n$

$c = 3$ — satisfies above cond

$2n+3 \leq 3n$

$n_0 = 1$ ✗

2 ✗

3 ✓

$n_0 = 3$

for all val starting from 3 satisfied

↓

To select no Imp cond ⓐ

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

$2n+3 = O(n)$

$f(n) = 2n+3$

$g(n) = n^2$ n^3

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

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

$c = 3$

$2n+3 \leq 3 * n^2$ n^3

$n_0 = 2$

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

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

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

$2n+3 = O(n) / O(n^2) / O(n^3) / \text{-----}$

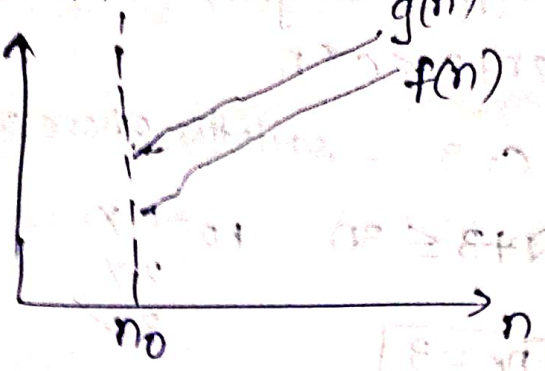
↓ correct representation

$g(n)$ – upperbound of $f(n)$

↓ out of all possible UB select Least Upper Bound

$O(n)$

Least Upper Bound



$f(n) \leq g(n)$

Ex1: Represent the complexity of 100n+6 using Big Oh Notation.

$f(n) = 100n+6$

$g(n) = n$

~~$f(n) \leq O(g(n))$~~

~~$100n+6$~~

$f(n) \leq c * g(n_0)$ $\boxed{c=101}$

$100n+6 \leq 101n$

$\boxed{n_0 = 6}$

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

$\boxed{100n+6 = O(n)}$

Ex2: Represent the complexity $10n^2+4n+6$ using O notation

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

$g(n) = n^2$

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

$10n^2+4n+6 \leq c * g(n)$

$\boxed{c = 11}$

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

$\boxed{n_0 = 6}$

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

$\boxed{f(n) = O(n^2)}$

$\boxed{10n^2+4n+6 = O(n^2)}$

Ex-3: Represent the complexity $6*2^n+n^2$ using Big Oh Notation.

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

$g(n)=2^n$

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

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

$c=7$

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

$n_0=4$

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

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

Ex-4: Represent the complexity $n\log n+\log n+10$ using O notation.

$f(n)=n\log n+\log n+10$

$g(n)=n\log n$

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

$n\log n+\log n+10 \le c*n\log n$

$c=2$

$n\log n+\log n+10 \le 2n\log n$

$n_0=11$

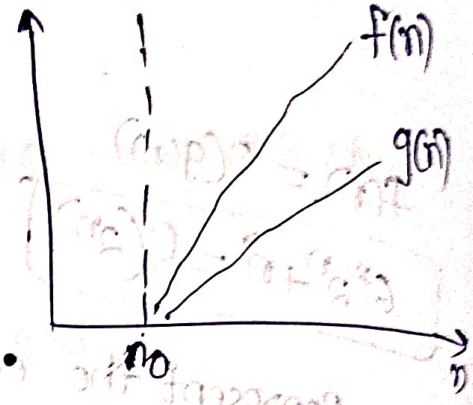
$f(n)=O(g(n))$

$n\log n+\log n+10=O(n\log n)$ //

$(3\log 2+10 \le 4\log 2)$

② Omega (Ω) Notation

If $f(n)$ and $g(n)$ are functions defined in terms of n, then we can write $f(n) = \Omega(g(n))$ if and only if there exist two positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all $n \geq n_0$



Ex:- $2n+3$

$f(n) = 2n+3$

$g(n) = n$

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

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

can also be

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

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

$2n+3 \geq c * n$

$c = 1, 2$

$c = 2$

$2n+3 \geq 2n$

$n_0 = 1$

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

↓ correct as we should consider highest val

Ex:- $100n+6$

$f(n) = 100n+6$

$g(n) = n$

~~ff~~

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

$100n+6 \geq c * (n)$

$c = 100$

$100n+6 \geq 100n$

$n_0 = 1$

$100n+6 = \Omega(n)$

$100n+6 = \Omega(n)$

Ex:- $10n^2+4n+6$

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

$g(n) = n^2$

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

$10n^2+4n+6 \leq c * (n^2)$

$c = 10$

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

$n_0 = 1$

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

Ex:- $6*2^n+n^2$

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

$g(n)=2^n$

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

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

$c=6$

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

$n_0=1$

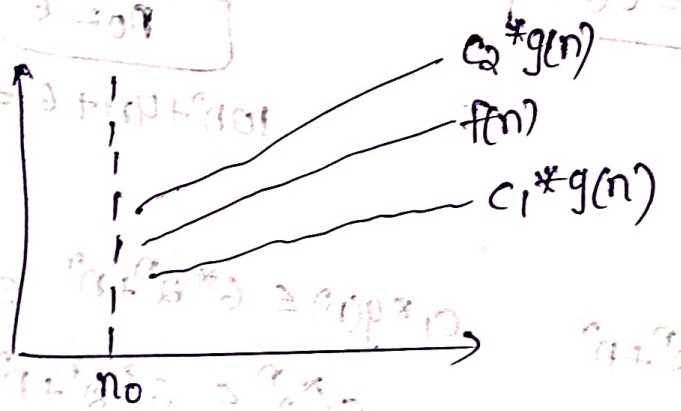
$6*2^n+n^2=\Omega(2^n)$

$\Omega(n), \Omega(1) \Omega(2^n)$

$=6*2^n+n^2$

Theta (Θ) Notation :- better than O, Ω

If f(n) and g(n) are functions defined interms of n, then we can write $f(n)=\Theta(g(n))$ if and only if there exists three postive constants c_1, c_2 and n_0 such that $c_1*g(n) \leq f(n) \leq c_2*g(n)$ for all $n \geq n_0$



Ex:- $2n+3$

$f(n)=2n+3$

$g(n)=n$

$c_1*g(n) \leq f(n) \leq c_2*g(n)$

$c_1 n \leq 2n+3 \leq c_2*n$

$c_1=2$, $c_2=3$

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

$n_0=3$

$\therefore 2n+3=\Theta(n)$

only exists, only possibility

$2n+3=O(n) / O(n^2) / O(n^3)$ ... ——

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

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

Ex:- $100n+6$

$f(n) = 100n+6$

$g(n) = n$

$c_1 * g(n) \le f(n) \le c_2 * g(n)$

$c_1 * n \le 100n+6 \le c_2 * n$

$c_1 = 100$ $\quad$ $c_2 = 101$

$100n \le 100n+6 \le 101n$

$n_0 = 6$

$100n+6 = \Theta(n)$

Ex:- $10n^2+4n+6$

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

$g(n) = n^2$

$c_1 * g(n) \le f(n) \le c_2 * g(n)$

$c_1 * n^2 \le 10n^2+4n+6 \le c_2 * n^2$

$c_1 = 10$ $\quad$ $c_2 = 11$

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

$n_0 = 6$

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

Ex:- $6*2^n+n^2$

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

$g(n) = 2^n$

$c_1 * g(n) \le 6*2^n+n^2 \le c_2 * g(n)$

$c_1 * 2^n \le 6*2^n+n^2 \le c_2 * 2^n$

$c_1 = 6$ $\quad$ $c_2 = 7$

$6*(2^n) \le 6*2^n+n^2 \le 7*(2^n)$

$n_0 = 4$

$6*2^n+n^2 = \Theta(2^n)$

* Prove the statement $4n^3 + 6n^2 + 20 = O(n^3)$

$f(n) = 4n^3 + 6n^2 + 20$

$g(n) = n^3$

$c_1 * n^3 \leq 4n^3 + 6n^2 + 20 \leq c_2 * n^3$

let $c_1 = 4,\ c_2 = 5$

$4n^3 \leq 4n^3 + 6n^2 + 20 \leq 5n^3$

$\Rightarrow n_0 = 7$

$\therefore 4n^3 + 6n^2 + 20 = O(n^3)$

Hence proved

* Prove that $3^n \neq O(2^n)$

$f(n) = 3^n$

$g(n) = 2^n$

$3^n \leq c * 2^n$

not possible to find c and n_0 that satisfies $f(n) \leq c * g(n)$

$\therefore 3^n \neq O(2^n)$

10/1/25

Methods to solve Recurrence Relations

$T(n) = 2 + T(n-1)$ — factorial recurrence rel

1) Substitution Method
2) Masters Theorem Method
3) Recursive Tree Method

1) Substitution Method

format - $T(n) = 2 + T(n-1)$ → substituting

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

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

$\vdots$ After $(n-1)$ times

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

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

$T(1) \downarrow 2$

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

2) Masters Theorem

↳ format ⇒ $T(n) = a\,T(n/b) + n^k \log^p n$

$a \geq 1,\ b > 1,\ k \geq 0,\ p \geq 0$ p is a real no

compare a with b^k

case 1: $a > b^K$ $T(n) = \theta(n^{\log_b a})$

case 2: $a = b^K$ $T(n) = \theta(n^{\log_b a})$ — $p < -1$

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

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

case 3: $a < b^K$

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

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

~~$p > -1$~~ ✗

Examples

q) Solve the foll recurrence relation

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

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

$b^K = 2^2 = 4$

$a < b^K$

$p = 0 \Rightarrow T(n) = \theta(n^2 \log^0 n) = \theta(n^2)$ (case-3)

$\boxed{T(n) = \theta(n^2)}$

q) $T(n) = 2T(n/2) + n \log n$

$a = 2, b = 2, K = 1, p = 1$

$b^K = 2^1 = 2$

$a = b^K$ (case 2)

$p > -1 \Rightarrow T(n) = \theta(n^{\log_2 2} \cdot \log^2 n)$

$\boxed{T(n) = \theta(n \log^2 n)}$ //

q) $T(n) = \sqrt{2}\,T(n/2) + \log n$

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

$b^k = 2^0 = 1$

$a > b^k$ — case-1, $p = 0$

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

$T(n) = \theta(n \log n)$ //

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

$T(n) = \theta(\sqrt{n})$ //

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

$a = 3$, $b = 3$, $k = 1/2$, $p = 0$

$b^k = 3^{1/2} = 3$

$a = b^k$, $p > -1$ — case 3

$T(n) = \theta(n^{\log_b a}) = \theta(n^{\log_3 3})$

$T(n) = \theta(n)$ //

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

$T(n) = \theta(n \cdot \log^1 n)$ //

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

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

$b^k = 2^2 = 4$

$a < b^k$ — (case 3)

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

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

Recursion Tree Method

Recursive algorithms → invokes once

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

↳ 1 time

Factorial(4)
↓
Factorial(3)
↓
Fact(2)
↓
Fact(1)

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

↓ Substitution method

Mergesort

→ invokes 2 times inside

```
Algorithm Mergesort(a,s,e)
{ m = (s+e)/2;
  Mergesort(a,s,m);
  Mergesort(a,m+1,e);   } 2 times
  Merge(a,s,m,e);
}
```

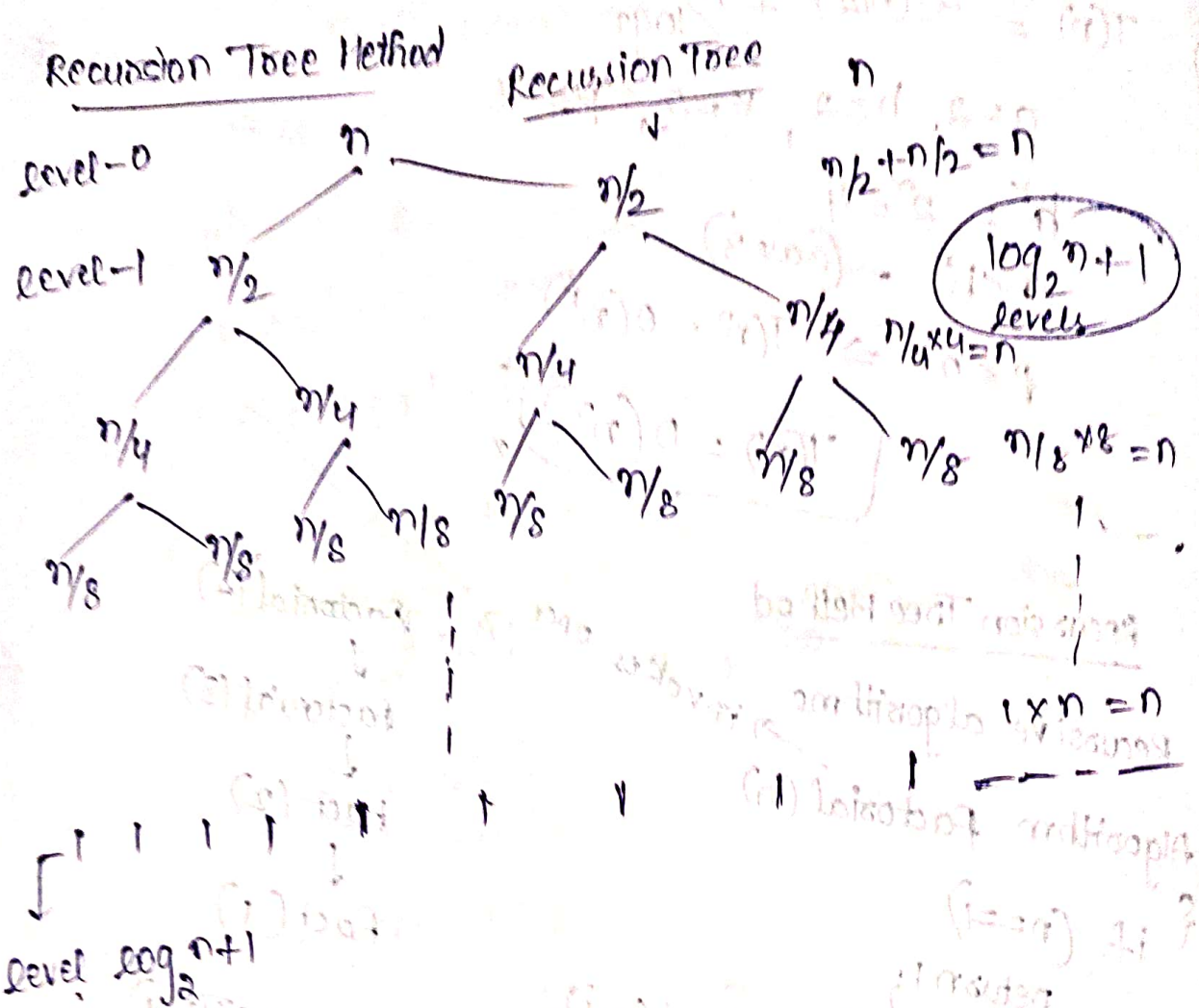
Recursion Tree Method

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

↑

for each invocation, recursive algorithm is invoked more than once.

Recursion Tree Method



TO cal TC — we cal cost of each level

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

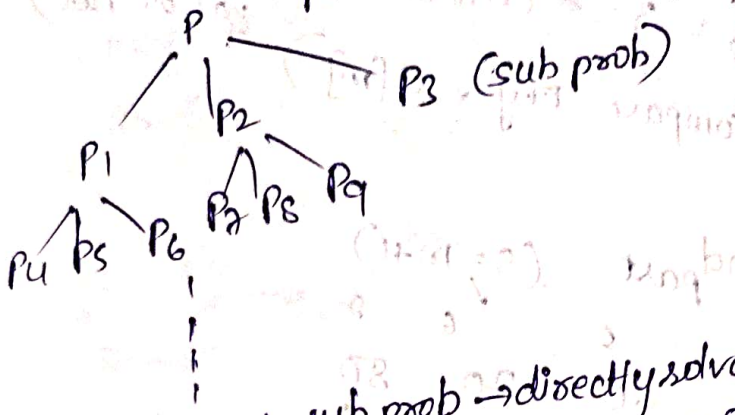
$T.C = \Theta(n * (\log_2 n)) = T(n)$

17/11/25

Mod-2

(Divide and Conquer Method)

problem - not directly solvable



until we get sub prob → directly solvable → solution

① combining sol^n's of sub prob get sol^n of P

② single sub prob, 1 sol^n enough to get sol^n of P

Applications of Divide and Conquer Method

1) Binary search
2) Merge sort
3) Quick sort

1) Binary search

To search for a val in a list of values.

Input1: List of values in sorting order

Input2: val to be searched - key

a:	10	20	30	40	50	60	70	80	key = 70
	0	1	2	3	4	5	6	7	
	↑ s			↑ m				↑ e	

$m = (s+e)/2 = (0+7)/2 = 3$ ← (list contains atleast one val or not)

key a[m] (Compare key & a[m])

70 40

key > a[m] – 2nd part (s = m+1)

	4	5	6	7
	50	60	70	80
	↑ s	↑ m		↑ e

$m = (s+e)/2 = (4+7)/2 = 5$

key a[m]

70 60

key > a[m] – 2nd part (s = m+1)

6	7	
70	80	
↑ s	↑ m	↑ e

$m = (6+7)/2 = 6$

key a[m]

70 70

key = a[m]

① Verify list has atleast one val

② Cal mid val

③ Compare key & mid and check whether to proceed or not.

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

a: ↑s ↑m ↑e key = 25

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

key < a[m] first part

25 < 40

e = mid − 1 = 2

0	1	2
10	20	30
↑s	↑m	↑e

$m = \frac{0+2}{2} = 1$

key a[m]

25 > 20 second part

s = m + 1 = 1 + 1 = 2

2
30
↑s ↑e ↖m

$m = \frac{2+2}{2} = 2$

key < a[m]

25 < 30 no first part → [list empty (noval)]

val is not present in list

e =

s = m + 1 = 2 + 1 = 3

25 < 30

m − 1

e = 2 − 1 = 1, s = 2

s > e — return −1

```
Algorithm Binarysearch (a, s, e, key)
{ //'a' is an array containing list of val in sorting order
  //'s' and 'e' are starting & ending positions of the array
      (or 'n' ——— size can be passed of array
  //'key' val is the val to be searched.
                                          → case-2
  if( s>e)  →1
      return -1;
  else {
      m = (s+e)/2;  →1
      if (key==a[m])  →1
          return m;  →1
      else if (key<a[m])
          Binarysearch (a, s, m-1, key);
      else
          Binarysearch (a, m+1, e, key);
  }
}
```

Time Time Complexity

TC

case-1: s>e

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

$T(0) = 2$

if (s>e) → 1

return -1; → 1

case-2: s≤e

$T(n) = 1+1+1+1 = 4$ — Best Case, $T(1)^2$

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

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

Avg case, Worst case

Unsuccessful search

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

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

$a=1,\ b=2,\ K=0,\ p=0$

$b^K = 2^0 = 1$

$a = b^K,\ p > -1$

(case-2)

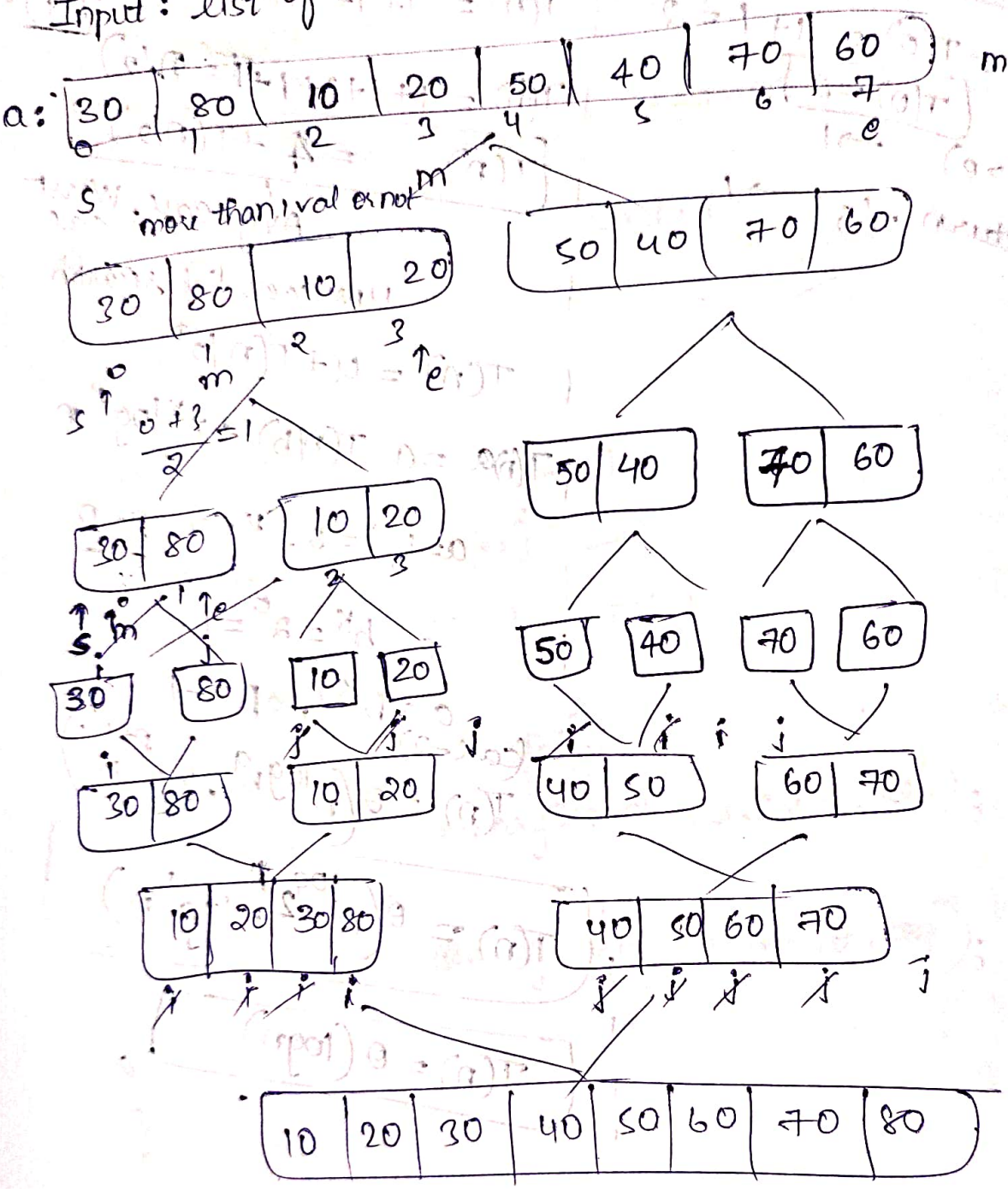
$T(n) = \theta\left(n^{\log_b a} \cdot \log^{p+1} n\right)$

$T(n) = \theta\left(n^{\log_2 1} \cdot \log^1 n\right)$

$T(n) = \theta(\log n)$ //

② Merge Sort

Input : list of val



```
Algorithm MergeSort(a, s, e)
// 'a' is an array containing list of values
// 's' and 'e' are starting and ending positions of array 'a'
{
   if(s<e)
   {
      m=(s+e)/2;
      MergeSort(a,s,m);
      MergeSort(a,m+1,e);
      Merge(a,s,m,e);
   }
}
Algorithm Merge (a,s,m,e)
{while(i<=m && j<=e){
    i=s;
    j=m+1;
    k=s;
    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];
    i=i+1;
    k=k+1;
}
while(j<=e){
    b[k]=a[j]
    j=j+1;
    k=k+1;
}
for(int x=s; x<=e; x++){
    a[x]=b[x];
}
```