⇒ Amortized Cost Analysis:-

- It is also known as average cost Analysis. It is the method of analysing the cost associated with the data structure that averages the worst operations out over time. This is required because one worst case cannot signify the overall performance of an algorithm. So, Amortize analysis is used to average out the costly operations in the worst case.

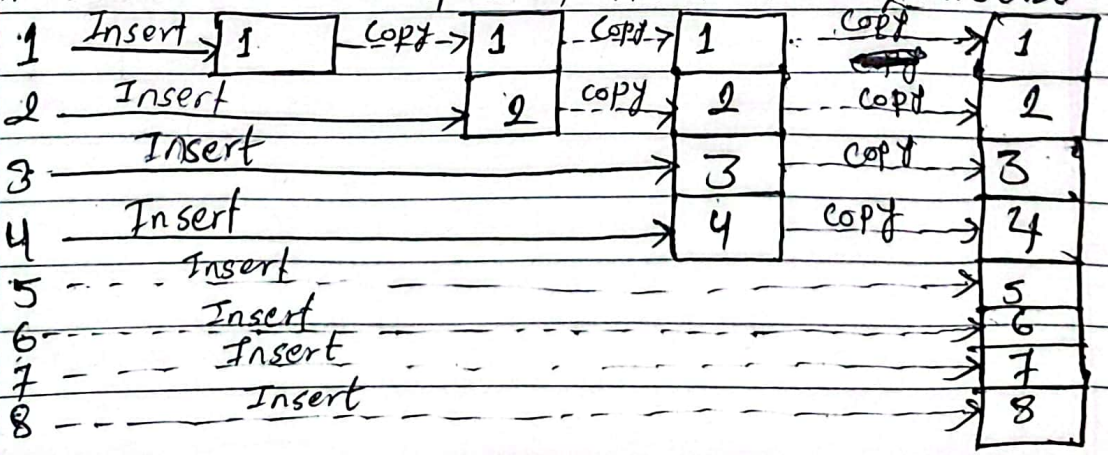
- There are three main types of amortized analysis:

① Aggregate Analysis.
② Accounting Analysis.
③ Potential Analysis.

1) Aggregate Analysis:-

In aggregate analysis, there are two steps, first, we must show that the sequence of 'n' operations takes T(n) time in the worst case Then, we show that each operation takes $\frac{T(n)}{n}$ time on average. Therefore, in aggregate analysis each operation has some cost.

Consider an example of Hash Table Insert.



Insert	i	1	2	3	4	5	6	7	8	9	10
size	i	1	2	4	4	8	8	8	8	16	16
cost	c_i	1	2	3	1	5	1	1	1	9	1

Here, c_i is the cost involved, which is the sum of copy + insert

now,

$$c_i = \begin{cases} i & \text{if } (i-1) \text{ is the exact power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

Again,

$$c_i = n + \sum_{j=0}^{\lfloor \log_2 i \rfloor} 2^j$$

or, $c_i = [1+1+1+1+\cdots) + (1+2+4+8+\cdots)$

As this is the geometric series, it can be generalized as,

$$c_i = n + 2n - 3$$
$$= 3n - 3$$
$$O(n) = n$$

Average cost $O(n) = \frac{O(n)}{n} = 1$

Amotized cost Analysis of hash table, if the table doubling is also included.

→ While using dynamic table as array, following are the step needed to be followed, when table become full.

i) Allocate memory for twice the old table size.
ii) copy the contents of old table to new table.
iii) free the old table.

If the table doubling cost is also included then,

Item	size	operation cost	copying cost	Doubling cost	Total
1	1	1	0	0	1
2	2	1	1	1	3
3	4	1	2	1	4
4	8	1	4	0	1
5	8	1	4	1	6
6	8	1	0	0	1
7	8	1	0	0	1
8	8	1	0	0	1
9	16	1	8	1	10
10	16	1	0	0	1

Here, C_i = operation cost + copying cost + Doubling cost

i.e. $C_i = n + \sum_{i=0} 2^i + \sum_{j=0} 1^j$

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

$= 2n + 2n - 3$

$= 4n - 3$

$= O(n)$

$\therefore \hat{C}_i = O(n) = O(n)/n = 1$ #

2) Accounting Analysis:-

The accounting method is apply named because it borrows ideas and terms from accounting. This method is burroweed from finance where there is the concept of ~~storing~~ storing excess amount in the bank for future use. Here, each operation is assigned a

charge called the amotized cost. Some operation can be charged more or less than they actually cost, we assign the difference called credit. Creadit can be never be negative in any sequence of operations. This method requires setting the cost of $\hat{c}_i$ (Amotized Cost), which should be equal for all inserts such that. Bank ≥ 0 (cost) which is turn will satisfy the inserts of all the input $\hat{c}_i = 3$.

Q.N-1

$\leftarrow$ Bank$_i$ = ~~Bank$_i$~~ + ($\hat{c}_i - c_i$)

$\hat{c}_i$	3	3	3	3	3	3	3	3	3	3
Bank$_i$	2	3	3	5	3	5	7	9	3	1
c_i	1	2	3	1	5	1	1	1	9	1

$\hat{C}_i = 3n$

~~Q~~

i	1	2	3	4	5	6	7	8	9	10
size$_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

$\hat{c}_i$	3	3	3	3	3	3	3	3	3	3
Bank	2	2	3	5	3	5	7	9	3	5

if guess = 2

$\hat{c}_i$	2	2	2	2	2	2	2	2	2	2
Bank	1	1	0	1	-1					

which is invalid.

So, $\hat{c}_i = 3$ $\leftarrow$ limit is correct guess.

If every 1 operation cost is 3 then, n operation cost is 3n.

$\therefore O(n) = \frac{3n}{n} = 3 = 1$ (constant).

3) Potential Analysis :-

Here, we have to define potential function (ϕ).

$\phi(D_0)$

$\phi(D_i)$

$\phi(D_{i-1})$

potential difference $= \phi(D_i) - \phi(D_{i-1})$

Lets assume $\phi(D_i) = 2i - 2^{\lceil \log i \rceil}$

$\phi(D_{i-1}) = 2(i-1) - 2^{\lceil \log (i-1) \rceil}$

$\hat{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$

Here,

$$C_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

$$\therefore \hat{C}_i \begin{cases} i, & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

$\cdot 2i - 2^{\lceil \log i \rceil} - [2(i-1) - 2^{\lceil \log (i-1) \rceil}]$

Case I :-

$\hat{C}_i = i - 2i - 2^{\lceil \log i \rceil} - 2i - 2 - 2^{\lceil \log (i-1) \rceil}$

$= i - 2(i-1) + 2 + (i-1)$ generalization.

$= i - 2i + 2 + 2 + i - 1$

$= 4 - 1$

$= 3$

Case II :-

$\hat{C}_i = 1 - 2^{\lceil \log i \rceil} + 2 + 2^{\lceil \log (i-1) \rceil}$

$= 1 - (i-1) + 2 + (i-1)$ (generalization)

$= 1 - i + 1 + 2 + i - 1$

$= 3$

⇒ Randomized Algorithm -:

- A probabilistic algorithm is an algorithm where the result and/or the way the result is obtained depend on chance. These algorithms are also sometimes called randomized algorithms.
- It makes use of randomizer (random number generator) to make some decision in a algorithm.
- The output and/or execution time of a randomized algorithm could also differ from run to run for the same input.
- It can be categorized into:
  1) Las Vegas Algorithm
  2) Monte Carlo Algorithm

⇒ Las Vegas Algorithm -:

- The algorithm always same (correct) output for the same input.
- Execution time of this algorithm is characterized as a random variable.
- The execution time depends on the output of randomizer.
- Type of Randomized quick sort.
- It guarantees the correctness of the solution.
- Upper bound cannot fixed.

⇒ Monte Carlo Algorithm -:

- It is a algorithm whose output might differ from run to run for the same input.
- It does not guarantees the correctness of the solution.
- Type of primility Testing.
- Upper bound can be fixed.
- The execution time is fixed.

⇒ Las Vegas Algorithm

```
void LasVegas()
{
  while (1) {
    int i = random()%2;
    if (i) return;
  }
}
```

```
LasVegasSearch(A,n) {
  while (true)
  {
    randomly choose an
    element out of n.
    if (a found)
      return true;
  }
}
```

⇒ monte Carlo Algorithm -:

```
MonteCarloSearch(A, n, x) {
  p=0
  while (p≤x) {
    randomly choose an element out of n.
    If (a found) {
      flag = true;
    }
    return flag;
  }
```

⇒ primality Testing -:

- to check whether the given number is prime or not

⇒ Bruth force Algorithm

If n is the no to be checked

```
i = 2
for i = 2 to n-1
n % i = 0
  n == i
    prime
  not prime.
```

→ If $a^{n-1} \% n == 1$

prime

else

not prime.

n is the number to be checked.

$a < n$ (the numbers)

let $n = 2$

$a = 1$

$\Rightarrow 1^{2-1} \% 2 \Rightarrow 1 \% 2 \neq 1$

$n = 3$

$a = 1, 2$ and $2^{3-1} \% 3 = 4 \% 3$

$\Rightarrow 1^{3-1} \% 3 = 1 \% 3 = 1$ $= 1$

$n = 4$

$a = 1, 2, 3$

If $a = 1 \Rightarrow 1^{4-1} \% 4 = 1$

If $a = 2 \Rightarrow 2^{4-1} \% 4 = 0$ ⇒ complexity - O(1)

If $a = 3 \Rightarrow 3^{4-1} \% 4 = 3$ // · monte carlo

⇒ Primality Testing :-

- Any integer greater than one is said to be a prime if its only divisors are 1 and the integer itself. By convention, we take 1 to be a non-prime, then 2, 3, 5, 7, 11, and 13 are the first six primes. Given an integer n, the problem of deciding whether n is a prime is known as primality testing.

```
Las Vegas () {
    while (true) do {
        i := Random () mod 2;
        If (i >= 1) then return;
    }
}
```

Unit-1.2 : Advance Algorithm Design Techniques :-

⇒ Greedy Algorithms :-

- It is a simplest and straight forward method (approach).
- It take decision on the basis of current available information without worrying about the effect of the current decision in future.
- It works in phases. At each phase :
  - You take the best you can get right now, without regard for future consequences.
  - You hope that by choosing a local optimum at each step.
  - You will end up at a global optimum.
- At each stages decision is made regarding whether particular input is an optimal solution.

```
Greedy ( A[], n ) {
    for (i=1 ; to n ) {
        x = select (A)
        if ( feasible x ) {
        solution = solution + x ;
    }
    return solution ;
}
```

eg Huffman coding, Shannon Fano Algorithm Job scheduling etc.

→ lossless data encoding schemes is disadvantage.
→ Relatively slower process.

⇒ Huffman coding :-

- By David A. Huffman (1949)
- Huffman coding is a very popular algorithm for encoding data.
- It is greedy (best for greedy approach) algorithm.
- It reduces the average access time of codes as much as possible.
- It is a unique code generator algorithm.
- It generate always prefix code (optimal prefix).
- It is a lossless data compression algorithm.
- It use variable length code words to represent character's in a message.
- Shorter code words are assign to more frequent characters, and longer code words to less frequent characters.
- This reduces the total no of bits required to represent the message.
- Algorithm construct the binary tree of nodes where each leaf node represents a character in the message.
- The frequency of each character is used to determine the position of nodes in the tree.
- character with higher frequency are closer to the root, while those with lower frequency are farther from the root.
- Root to leaf node to generate binary code.
- The resulting code words are prefix-free, ensuring unique decoding.
- Huffman coding are used in data compression, image and video compression, and file archiving.
- eg best is Huffman coding for greedy.

Example-1

String = { "aaaa bbb ·e f ;;;;;;;" }

Frequency = {a=4, b=3, e=1, f=1, & ;=6}

Sol^n

A character is 1 byte long = 8-bit

∴ 8 × 19 = 152 bit (∵19 → total string)

code = a = 01 = 2

b = 110 = 3

Total bits = (frequency of ';') * length of code for ;) + (frequency of 'a') * length of code for a) + ------

= 6×2 + 4×2 + 4×2 + 3×3 + 4×1 + 4×1 = 45 bits.

Average code length = Total no of bits / Total no of symbol

= 45 / 19 = 2.37.

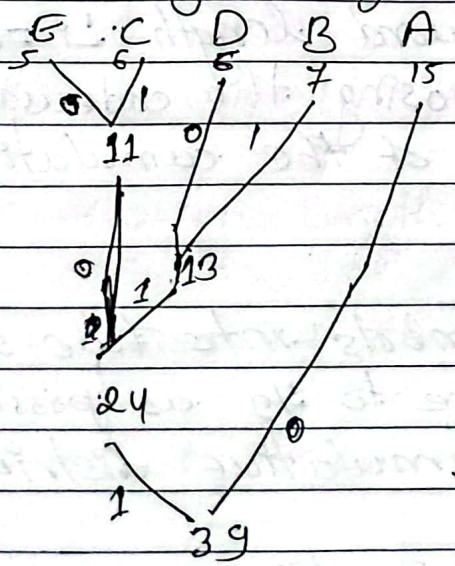
∴ Average code length is 2.37 bits per symbol as compared to 8-bit per symbol before encoding.

Example-2 ↓

symbol	A	B	C	D	E
count	15	7	6	6	5

Sol^n

Sorting all symbol first and use huffman coding.



⇒ A = 0 = 1

B = 111 = 3

C = 101 = 3

D = 110 = 3

E = 100 = 3

∴ Total bit length = 1×15 + 3×7 + 3×6 + 3×6 + 3×5

= 87

Average bit length = 87 / 39 = 2.231 bits

Entrophy $(H) = \sum_{i=1}^{n} P_i \log(1/n)$.

or $- \sum_{i=1}^{n} P_i \log(P_i)$

eg Hello $\Rightarrow$ $H = 1/5$, $e = 1/5$, $l = 2/5$, $o = 1/5$

$\therefore H = -(1/5)\log_2(1/5) - (1/5)\log_2(1/5) - (2/5\log_2 2/5) - (1/5)\log_2(1/5)$.

$\Rightarrow H = \ldots$

$\Rightarrow$ Shannon-Fano coding:-

- Related to field of data compression.
- Cloud shannon and Robert Fano - scientist
- Related techniques for constructing a prefix code based on a set of symbols and their probabilities. (estimated or measured).

$\rightarrow$ Shannon's method:-

- Chooses a prefix code where a source symbol 'i' is given the codeword length $L_i = \lceil -\log_2 P_i \rceil$
- one common way of choosing the codewords uses the binary expansions of the cumulative probabilities.

$\rightarrow$ Fano's method:-

- Divides the source symbols into two sets (0 & 1) with probabilities as close to 1/2 as possible.
- It is based on the cumulative distribution function.
- It provides low optimization.

Example :-

m.	a	b	c	d	e
P	0.3	0.15	0.15	0.2	0.2

Sol:-

	0.3	0.15	0.15	0.2	0.3
0.3 (a)	0	0			
0.2 (d)	0	1			
0.2 (e)	1	0			
0.15 (b)	1	1	0		
0.15 (c)	1	1	1		

Example :- Applying Shannon Fano coding for following message.

$[m] = [\, m_1 \quad m_2 \quad m_3 \quad m_4 \quad m_5 \quad m_6 \,]$

$[P] = [\, 0.30 \quad 0.25 \quad 0.15 \quad 0.12 \quad 0.08 \quad 0.10 \,]$

Sol:-

m	p		stage 1	stage 2	stage 3	codeword	length
m_1	0.30	0.55	0	0		00	2
m_2	0.25		0	1		01	2
m_3	0.15	0.45	1	0	0	100	3
m_4	0.12		1	0	1	101	3
m_6	0.10		1	1	0	110	3
m_5	0.08		1	1	1	111	3

Average length $L = \sum_{K=1}^{n} p_n h_K = \sum_{K=1}^{6} p_6 h_K$

$= (0.30 \times 2) + (0.25 \times 2) + (0.15 \times 3) + (0.12 \times 3) + (0.10 \times 3) + (0.08 \times 3) = 2.48$ bits

$\therefore \Rightarrow L = 2.45$ bits

⇒ Job scheduling with deadlines:-

- It has unique processor machine.
- Each job 'Ji' requires one unit time to complete.
- Each job 'Ji' has di deadlines and P_i - profits.
- Ji is - associated with P_i iff it meets its deadline
- Find the subset of jobs and its sequence, so that its profit is maximum.
- A machine can provide service to any one job at a time.
- The greedy approach to this problem involves sorting the job by their profits. in descending order and then scheduling them in the order in which they appear in the sorted list, as long as their deadlines have not been exceeded.

Algorithm:-

- Arrange all job J_i in decending order with respect to P_i.
- Allocate timeslot with respect to maximum deadlines.
- k = min (max (deadline), di)
- Allocate the time slot for J_i with respect to value of k.
- If the time slot is already occupied then search for the positions before it.
- If all the time slot is occupied publish the set and sequence and max profit.

• It has uniprocessor, non-preemption, and every job takes one unit of time.

n = maximum deadline & m = total job.

∴ total time = n x m.

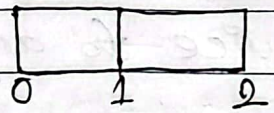
Example-1

	1	2	3	4
Job (J_i)	J_1	J_2	J_3	J_4
Profit (P_i)	100	10	15	17
deadline (d_i)	2	1	2	1

Soln

Time slot = max deadline → max deadline = 2

Since time slot = 2.

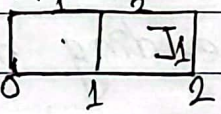


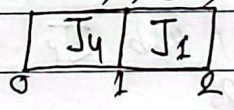
descending order is:

J_i	J_1	J_4	J_3	J_2
P_i	100	27	15	10
d_i	2	1	2	1

Iteration-1 $i=1$

$K = \min(\max(\text{deadline}), d_i) = \min(2,2) = 2$.

① 

② 

$\Rightarrow$ set = $\{J_1, J_4\}$

Sequen = $\{J_4, J_1\}$ $\Rightarrow$ profit max = 27+100 = 127 #

Iteration-2 : $i=2$

$k = \min(\max(\text{deadline}), d_i) = \min(2,1) = 1$

Example :

J_i	J_1	J_2	J_3	J_4
P_i	30	15	75	80
d_i	1	2	3	2

Soln descending order - sorting:

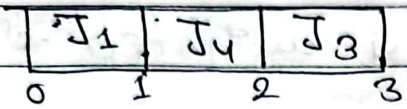
J_i	J_4	J_3	J_1	J_2
P_i	80	75	30	15
d_i	2	3	1	2

Total time slot = 3

for i=1

k = min (max (deadline), di) = max(3,2) = 2

⇒ J4 in slot 2



⇒ 30 + 80 + 75 = 185 # profit max.

for i=2

k = min (max (deadline), di) = min (3,3) = 3 #

for i=3

k = min (max (deadline), di) = min (3,1) = 1 #

⇒ set = { J1, J3, J4 }

sequence = { J1, J4, J3 }

profit max = 30 + 80 + 75 = 185 #

⇒ Analysis :-

Case-I :- Each time slot calculated is vaccant, so just insert is required which is equal to the no of insertion. ∴ O(n) = n. #

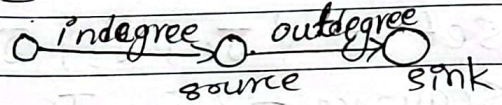
Case-II :- Time slot calculated is not vacant, so required another iteration to find the vacant slot. ∴ O(n) = n(n-1) ⇒ n^2 #

Algorithm

```
Greedy Job (d, J, n) {
    J = {1}
    for i = 2 to n do {
        if all deadlines in J ∪ {i} are feasible then
            J = J ∪ {i}
    }
    return J
}
```

⇒ Tree vertex splitting - (TVS) -:

- Consider directed binary tree each edge is leveled with weights.
- TVS can be defined as:
  Let $T = (V, E, W)$ be a weighted directed tree. .
  where,
  $V \rightarrow$ set of vertices
  $E \rightarrow$ set of edges
  $W(i,j) \rightarrow$ weight of edge $\langle i,j \rangle \in E$. and
- Source vertex has in-degree '0' and sink vertex has out degree.



- we define delay of path 'p' denoted as $d(P)$ = sum of weight of path.
  ⇒ Delay of Tree, $d(T) = \max(d(P))$ ∀
- Let $T = (V, E, W)$ be a directed tree. A weighted tree can be used to model a distribution network in which electrical signal are transmitted.
- The transmission of power from node to another may result in some loss.
- Each edge in the tree is labeled with the loss that occurs in traversing that edge.
- The network may not be able to tolerate losses beyond a critical limit.
- In places where the loss exceeds the tolerance level, boosters have to be placed.
- Given a network and a loss tolerance level, the tree vertex splitting problem is to

determine an optimal placement of boosters.

- A greedy approach to solve this problem is to compute for each node $u \in V$, the maximum delay $d(u)$ from u to any other node in its subtree.
- If u has a parent v such that
$d(u) = \max\{d(v) + w(parent(u), v)\}$
$d(u) \rightarrow$ delay of node.
v - set of all edges & u belongs to child(u)
δ tolerance value.

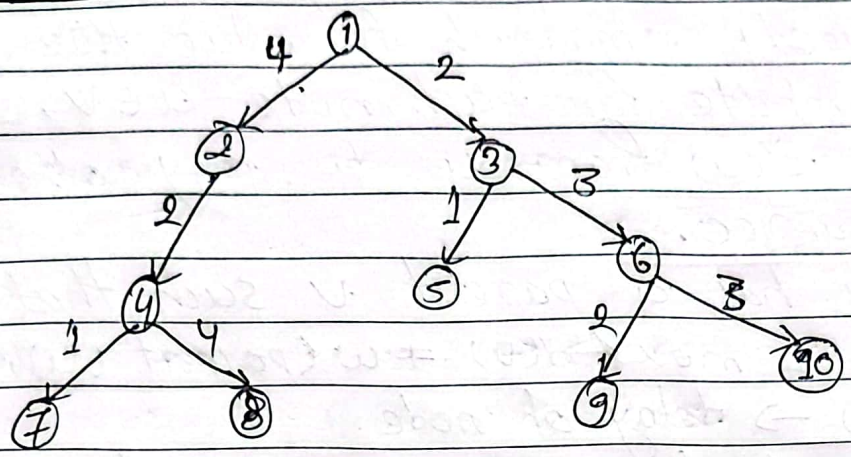
- **Algorithm -:**

```
Algorithm TVS(T, δ){
    if (T ≠ 0) then
    {
        d[T] = 0;
        for each child v to T do.
        {
            TVS(v, δ);
        d[T] = max{d[T], d[v] + W[T, v]};
    }
    if ((T is not the root) and (d[T] +
            W(parent(t), T) → δ)) then.
    {
        write [T];
        d[T] = 0;
    }}}
```

Analysis -:

- visit each vertex n;
- In each vertex there is constant time required for calculation and computing it with δ. ∴ $O(n) = n$ #

Example:-



If $d(u) >= \delta$; then place the booster.

Sol:

Here all leaf nodes delay = 0 and leaf nodes are 7, 8, 9, 10, 5

$\delta = 5$

$d[7] = \max\{0 + w(4,7)\} = 1$

$d[8] = \max\{0 + w(4,8)\} = 4$

$d[9] = \max\{0 + w(6,9)\} = 2$

$d[10] = \max\{0 + w(6,10)\} = 3$

$d[5] = \max\{0 + w(3,5)\} = 1$

$\Rightarrow d[7], d[8], d[9], d[10], d[5] < \delta \Rightarrow$ 0 ✓

$d[4] = \max\{1 + w(2,4) + 4 + w(2,4)\}$

$= \max\{1+2, 4+2\} = 6 > \delta$ booster ✓

$d[6] = \max\{2 + w(3,6) + 3 + w(3,6)\}$

$= \max\{2+3, 3+3\} = 6 > \delta$ booster ✓

$d[2] = \max\{6 + w(1,2)\}$

$= \max\{6+4\} = 10 > \delta$ booster ✓

$d[3] = \max\{1 + w(1,3), 6 + w(1,3)\}$

$= \max\{3, 8\} = 8 > \delta$ booster ✓

∴ No need to find tolerance value for node 1 because from source only power is transmitting.

⇒ Dynamic Programming :-

- It is a technique in compute programming used to solve optimized problems by breaking down a large problem into smaller problems and solving each sub-problem only once.
- principle of optimal substructure which means that the optimal solution to a large problem can be found by combining the optimal solutions to smaller sub-problems.
- Dynamic programming is efficient in finding optimal solutions for cases with lots of overlapping sub-problems.
- It solves problems by recombining solutions to sub-problems, when the sub-problems themselves may share sub-sub-problems.

eg. fibonacci number -

1, 1, 2, 3, 5, 8, 13, 21, 34, - - -.

Sol^n

```
def fib(n):
    if n==0 or n==1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decision.

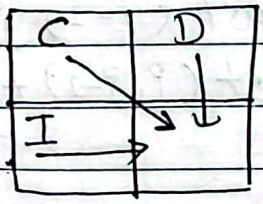
⇒ Difference between Greedy vs Dynamic Algorithm :-

Greedy Algorithm	Dynamic Algorithm
→ makes locally optimal choices at each step	→ Breaks down a problem in smaller subproblems.

→ Does not guarantee an optimal solution.	→ Guarantee an optimal solution.
→ ~~Does~~ Faster than dynamic programming.	→ slower than greedy programming.
→ may not consider all possible choices	→ consider all possible choices.
→ Works well when the problem ~~has~~ the greedy choice property.	→ can solve a wide range of a problems.
→ memory efficient as it only needs to store the current solution.	→ memory intensive, as it needs to store solution to a sub-problems.
Example: Huffman coding, kruskal's algorithm etc	example: knapsack problem, fibonacci sequence. etc.

⇒ String Editing :-

- There are two string source and destination. The goal of the problem is to convert source to destination by applying minimum edit operations on string 'source'.
- we have given two string $x = x_1, x_2, x_3, \dots, x_n$ and $y = y_1, y_2, y_3, \dots, y_m$; where $x_i = 1 \le i \le n$ and $y_j = 1 \le j \le m$ are number of finite set of symbols known as the alphabets, we want's to transform x to y using a sequence of edit operation on x.
- The permissible edit operations are:
  Delete — 1 cost → denote D(x_i)
  change/copy — 2 cost → denote C(x_i y_j)
  Insert — 1 cost → denote I(y_j)
- The problem of string editing is to identify a minimum cost sequence of edit operations that will transform x into y.



Example :-

String-1 → a a b a b (source)
String-2 → b a b b (destination)

Total cost = 5(source) + 4(destination) = 9 #

Technique-1 : Remove all characters within string-1, Insert the required string:

d d d d d → b → a → b → b
D D D D D I I I I ⇒ cost = 9 #

Technique-2:

a ∅ a b a b → b → First Insert

D D I ⇒ cost = 3

Technique-3:

a b a b ∅ b ← Replace with b

R D ⇒ cost = 2+1 = 3 #

Technique-4:

∅ b a b ∅ b b

R R D ⇒ cost = 5 #

Technique-5:

b ∅ a b ∅ b

I D D ⇒ cost = 3

$$cost(i,j) = \begin{cases} 0 & i=j=0 \\ cost(i-1,0) + D(x_i) & i>0;\ j=0 \\ cost(0,j-1) + I(y_j) & i=0;\ j>0 \end{cases}$$

or

$$cost(i,j) = \min \begin{cases} cost(i-1,j) + D(x_i) & i>0 \\ cost(i-1,j-1) + c(x_i,y_j) & i>0,\ j>0 \\ cost(i,j-1) + I(y_j) & j>0 \end{cases}$$

Example:

Source = a a b a b

destination = b a b b

Sol^n

i \ j	Null 0	a_1	a_2	b_3	b_4
Null 0	0 →	1	2	3	4
b_1	1 →	2	1 →	2	3
b_2	2	3	2 →	3	4
b_3	3	2	3	2 →	3
b_4	4	3	2	3	4
b_5	5	4	3	2 →	3

Step-1 :- Case · i=0 ; j=0 , cost (0,0) = 0

Case · i > 1 , j = 0

cost (0,1) = cost (0, 1-1) + I (y1) = 0+1 = 1

cost (0,2) = cost (0, 2-1) + I (y2) = 1+1 = 2

cost (0,3) = cost (0, 3-1) + I (y3) = 2+1 = 3

cost (0,4) = cost (0,4-1) + I (y4) = 3+1 = 4

Step2

cost (1,0) = cost (1-1, 0) + D (x1) = 0+1 = 1

cost (2,0) = cost (2-1, 0) + D (x2) = 1+1 = 2

cost (3,0) = cost (3-1, 0) + D (x3) = 2+1 = 3

cost (4,0) = cost (4-1, 0) + D (x4) = 3+1 = 4

cost (5,0) = cost (5-1, 0) + D (x5) = 4+1 = 5

Row-1

$$cost(1,1) = min \begin{cases} cost(1-1,1) + D(x_1) \\ cost(1-1,1-1) + c(x_1,y_1) \rightarrow \text{change cost} \\ cost(1,1-1) + I(y_1) \end{cases}$$

= min { 1+1, 0+2, 1+1 } = 2 #

$$cost(1,2) = min \begin{cases} cost(1-1,2) + D(x_1) \\ cost(1-1,2-1) + c(x_1,y_2) \\ cost(1,2-1) + I(y_2) \end{cases}$$

= min { 2+1, 1+0, 2+1 } = 1 #

$$cost(1,3) = min \begin{cases} cost(1-1,3) + D(x_1) \\ cost(1-1,3-1) + c(x_1,y_3) \\ cost(1,3-1) + I(y_3) \end{cases}$$

= min { 3+1, 2+2, 1+1 } = 2 #

$$cost(1,4) = min \begin{cases} cost(1-1,4) + D(x_1) \\ cost(1-1,4-1) + c(x_1,y_4) \\ cost(1,4-1) + I(y_4) \end{cases}$$

= min { 4+1, 3+2, 2+1 } = 3 #

Row-2 $cost(2,1) = \min \begin{cases} cost(2-1,1) + D(x_2) \\ cost(2-1,1-1) + C(x_2, y_1) \\ cost(2,1-1) + I(y_1) \end{cases}$

$\min\{2+1, 1+2, 2+1\} = 3$ #

$cost(2,2) = \min \begin{cases} cost(2-1,2) + D(x_2) \\ cost(2-1,2-1) + C(x_2, y_2) \\ cost(2,2-1) + I(y_2) \end{cases}$

$= \min\{1+1, 2+0, 3+1\} = 2$ #

$cost(2,3) = \min \begin{cases} cost(2-1,3) + D(x_2) \\ cost(2-1,3-1) + C(x_2, y_3) \\ cost(2,3-1) + I(y_3) \end{cases}$

$= \min\{2+1, 1+2, 2+1\} = 3$ #

$cost(2,4) = \min \begin{cases} cost(2-1,4) + D(x_2) \\ cost(2-1,4-1) + C(x_2, y_4) \\ cost(2,4-1) + I(y_4) \end{cases}$

$= \min\{4+1, 2+2, 3+1\} = 4$ #

Row-3 $cost(3,1) = \begin{cases} cost(3-1,1) + D(x_3) \\ cost(3-1,1-1) + C(x_3, y_1) \\ cost(3,1-1) + I(y_1) \end{cases}$

$= \min\{3+1, 2+0, 3+1\} = 2$ #

$cost(3,2) = \min \begin{cases} cost(3-1,2) + D(x_3) \\ cost(3-1,2-1) + C(x_3, y_2) \\ cost(3,2-1) + I(y_2) \end{cases}$

$= \min\{2+1, 3+2, 2+1\} = 3$ #

$$cost(3,3) = \min \begin{cases} cost(3-1, 3) + D(x_3) \\ cost(3-1, 3-1) + c(x_3, y_3) \\ cost(3, 3-1) + c(y_3) \end{cases}$$

$$= \min \{ 3+1, 2+0, 3+1 \} = 2 \#$$

$$cost(3,4) = \min \begin{cases} cost(3-1, 4) + D(x_3) \\ cost(3-1, 4-1) + c(x_3, y_4) \\ cost(3, 4-1) + I(y_4) \end{cases}$$

$$= \min \{ 4+1, 3+0, 2+1 \} = 3 \#$$

Row-4

$$cost(4,1) = \min \begin{cases} cost(4-1, 1) + D(x_4, y_1) \\ cost(4-1, 1-1) + c(x_4, y_1) \\ cost(4, 1-1) + I(y_1) \end{cases}$$

$$= \min \{ 2+1, 3+2, 4+1 \} = 3 \#$$

$$cost(4,2) = \min \begin{cases} cost(4-1, 2) + D(x_4) \\ cost(4-1, 2-1) + c(x_4, y_2) \\ cost(4, 2-1) + I(y_2) \end{cases}$$

$$= \min \{ 3+1, 2+0, 3+1 \} = 2 \#$$

$$cost(4,3) = \min \begin{cases} cost(4-1, 3) + D(x_4) \\ cost(4-1, 3-1) + c(x_4, y_3) \\ cost(4, 3-1) + I(y_3) \end{cases}$$

$$= \min \{ 2+1, 3+2, 2+1 \} = 3 \#$$

$$cost(4,4) = \min \begin{cases} cost(4-1, 4) + D(x_4) \\ cost(4-1, 4-1) + c(x_4, y_4) \\ cost(4, 4-1) + I(y_4) \end{cases}$$

$$= \min \{ 3+1, 2+2, 3+1 \} = 4 \#$$

Row 5

$$\text{Cost}(5,1) = \min \begin{cases} \text{cost}(5-1,1) + D(x_5) \\ \text{cost}(5-1,1-1) + C(x_5, y_1) \\ \text{cost}(5,1-1) + I(y_1) \end{cases}$$

$= \min \{ 3+1, 4+0, 5+1 \} = 4$ #

$$\text{cost}(5,2) = \min \begin{cases} \text{cost}(5-1,2) + D(x_5) \\ \text{cost}(5-1,2-1) + C(x_5, y_2) \\ \text{cost}(5,2-1) + I(y_2) \end{cases}$$

$= \min \{ 2+1, 3+0, 4+1 \} = 3$ #

$$\text{cost}(5,3) = \min \begin{cases} \text{cost}(5-1,3) + D(x_5) \\ \text{cost}(5-1,3-1) + C(x_5, y_3) \\ \text{cost}(5,3-1) + I(y_3) \end{cases}$$

$= \min \{ 3+1, 2+0, 3+1 \} = 2$ #

$$\text{cost}(5,4) = \min \begin{cases} \text{cost}(5-1,4) + D(x_5) \\ \text{cost}(5-1,4-1) + C(x_5, y_4) \\ \text{cost}(5,4-1) + I(y_4) \end{cases}$$

$= \min \{ 4+1, 3+0, 2+1 \} = 3$ #

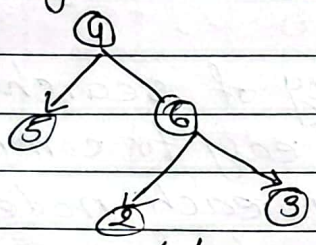
= minimum cost $= 1+1+0+0+0+1 = 3$

optimal cost (3) #

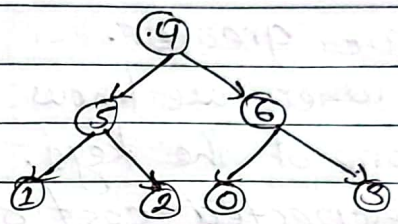
⇒ Optimal Binary Search Tree - (OBST) -:

1) Incomplete Binary Tree
2) Complete Binary Tree
3) Left skewed
4) Right skewed.

⇒ Binary Tree (BT) -:

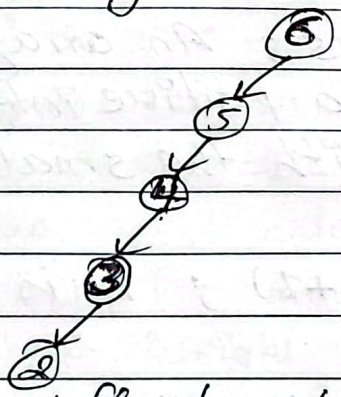


Eg: Incomplete Binary Tree

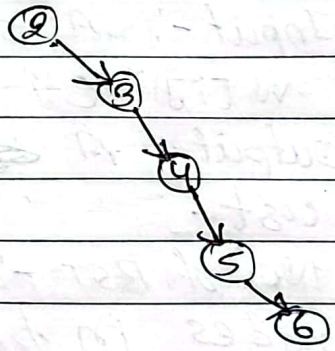


Eg: Complete Binary Tree.

⇒ Binary Search Tree -:



Eg Left skewed BST



Eg Right skewed BST.

Binary Tree	Binary Search Tree
- A tree data structure in which each node has at most two children without any ordering restrictions.	- A tree data structure in which each node follows a specific ordering rule, left subtree contain only nodes with keys less than parent nodes & right subtree contains only node with key generate then the parent node.
- Ordering of node not required.	- ordering of nodes are sorted.
- Efficiency : O(log n)	- Efficiency : O(log n).

Definition:-

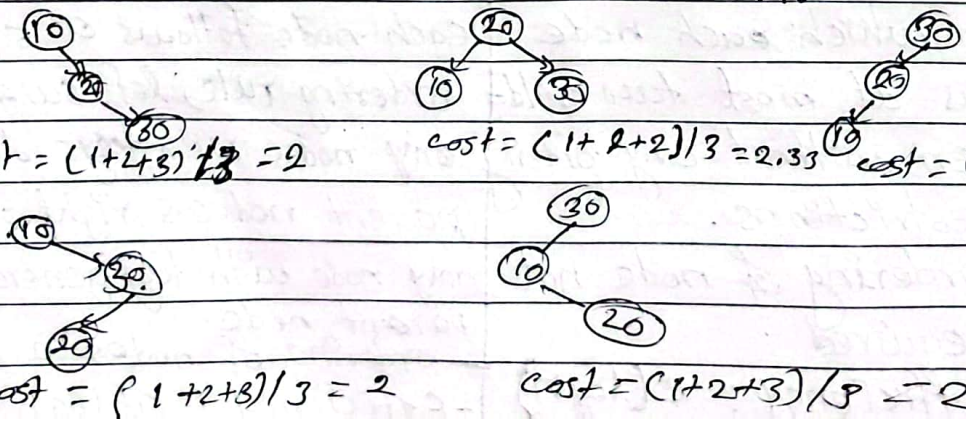
- Binary search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null node. The keys are stored lexicographically, i.e. for each internal node all the key in the length sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.
- When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node.
- Input:- A set of n integers; An array W where; W[i] (1 ≤ i ≤ n) stores a positive integer weight.
- Output:- A BST on S with the smallest average cost.
- No. of BST :- = $^{2n}C_n / (n+1)$; n is no of nodes in tree.

  eg $n=4 \Rightarrow \frac{8!}{4!5!} = 14$ #

Example:- given node: 10, 20, 30

The possible BST are:

for $n=3 \Rightarrow$ no of BST $= \frac{6!}{3!4!} = 5$ #



Example :-

X :	10	20	30	40
freq" :	4	2	6	3

Soln solving with dynamic programming :

i ↓ j →	0	1	2	3	4
0	0	$4^{(1)}$	$8^{(1)}$	$20^{(3)}$	$26^{(3)}$
1		0	$2^{(1)}$	$10^{(3)}$	$16^{(3)}$
2			0	$6^{(1)}$	$12^{(3)}$
3				0	$3^{(1)}$
4					0

Case I : $j-i=0$

$0-0=0 \Rightarrow c[0,0]=0$

$1-1=0 \Rightarrow c[1,1]=0$

$2-2=0 \Rightarrow c[2,2]=0$

$3-3=0 \Rightarrow c[3,3]=0$

$4-4=0 \Rightarrow c[4,4]=0$

Case II : $j-i=1$

$1-0=1 \Rightarrow c[0,1]=10 \Rightarrow 4$

$2-1=1 \Rightarrow c[1,2]=20 \Rightarrow 2$

$3-2=1 \Rightarrow c[2,3]=30 \Rightarrow 6$

$4-3=1 \Rightarrow c[3,4]=40 \Rightarrow 3$

now, cost calculate ;.

$$\cdot\ c[i,j] = \min_{k=i+1}^{j} \{c[i,k-1]+c[k,j]\} + w[i,j]$$

where weight $w[i,j] = \sum_{i=1}^{n} f(i)$.

then,

Case III $j-i=2$

$2-0=2 \Rightarrow c[0,2] = \min_{k=1,2} \begin{Bmatrix} c[0,0]+c[1,2], \\ c[0,1]+c[2,2] \end{Bmatrix} + w[0,2]$

$= \min \begin{Bmatrix} 0+2, \\ 4+0 \end{Bmatrix} + 4+2 = 2+4+2 = 8^{(1)}$ #

$3-1=2 \Rightarrow c[1,3] = \min_{k=2,3} \begin{Bmatrix} c[1,1]+c[2,3], \\ c[1,2]+c[3,3] \end{Bmatrix} + w[1,3]$

$= \min \begin{Bmatrix} 0+6, \\ 2+0 \end{Bmatrix} + 2+6 = 2+2+6 = 10^{(3)}$ #

$4-2=2 \Rightarrow c[2,4] = \min_{k=3,4} \{ c[2,2]+c[3,4],\ c[2,3]+c[4,4] \} + w[2,4]$

$= \min \{ 0+3,\ 6+0 \} + 6+3 = 3+6+3 = 12^{(3)}$ #

Case IV : $j-i=3$

$3-0=3 \Rightarrow c[0,3] = \min_{k=1,2,3} \{ c[0,0]+c[1,3],\ c[0,1]+c[2,3],\ c[0,2]+c[3,3] \} + w[0,3]$

$= \min \{ 0+10,\ 4+6,\ 8+0 \} + 4+2+6 = 8+4+2+6 = 20^{(3)}$ #

$4-1=3 \Rightarrow c[1,4] = \min_{k=2,3,4} \{ c[1,1]+c[2,4],\ c[1,2]+c[3,4],\ c[1,3]+c[4,4] \} + w[1,4]$

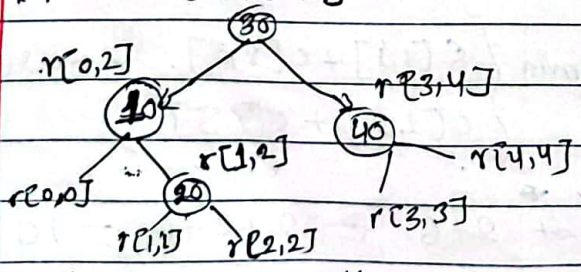
$= \min \{ 0+12,\ 2+3,\ 10+0 \} + 2+6+3 = 5+2+6+3 = 16^{(3)}$ #

Case-V : $j-i=4$

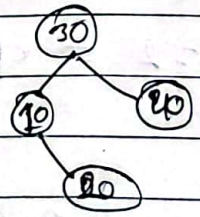
$4-0=4 \Rightarrow c[0,4] = \min_{k=1,2,3,4} \{ c[0,0]+c[1,4],\ c[0,1]+c[2,4],\ c[0,2]+c[3,4],\ c[0,3]+c[4,4] \} + w[0,4]$

$= \min \{ 0+16,\ 4+12,\ 8+3,\ 20+0 \} + 4+2+6+3 = 11+4+2+6+3 = 26^{(3)}$ #

$\therefore\ r[0,4] = 3$



$\Rightarrow$



#

Analysis : Algorithm requires $O(n^3)$ time, since we have three (3) level of nested for loop, each of these loops take an at most n value #

Example :- Optimal Binary Search Tree (OBST) :-

x_i	—	10	20	30	40
f_i	—	4	2	6	3
p_i	—	3	3	1	1
q_i	2	3	1	1	1

Sol^n Now we construct value of cost, weight and root and fill the following table :

$w_{0,0} = 2$	$w_{1,1} = 3$	$w_{2,2} = 1$	$w_{3,3} = 1$	$w_{4,4} = 1$
$c_{0,0} = 0$	$c_{1,1} = 0$	$c_{2,2} = 0$	$c_{3,3} = 0$	$c_{4,4} = 0$
$r_{0,0} = 0$	$r_{1,1} = 0$	$r_{2,2} = 0$	$r_{3,3} = 0$	$r_{4,4} = 0$
$w_{0,1} = 8$	$w_{1,2} = 7$	$w_{2,3} = 3$	$w_{3,4} = 3$	
$c_{0,1} = 8$	$c_{1,2} = 7$	$c_{2,3} = 3$	$c_{3,4} = 3$	
$r_{0,1} = 1$	$r_{1,2} = 2$	$r_{2,3} = 3$	$r_{3,4} = 4$	
$w_{0,2} = 12$	$w_{1,3} = 9$	$w_{2,4} = 5$		
$c_{0,2} = 19$	$c_{1,3} = 12$	$c_{2,4} = 8$		
$r_{0,2} = 1$	$r_{1,3} = 2$	$r_{2,4} = 3$		
$w_{0,3} = 14$	$w_{1,4} = 11$			
$c_{0,3} = 25$	$c_{1,4} = 19$			
$r_{0,3} = 2$	$r_{1,4} = 2$			
$w_{0,4} = 16$				
$c_{0,4} = 32$				
$r_{0,4} = 2$				

$$c[i,j] = \min_{i<k\le j} \{ c[i,k-1] + c[k,j] \} + w[i,j]$$

and weight $w[i,j] = w[i,j-1] + p_j + q_j$

$w[0,0], w[1,1], w[2,2], w[3,3], w[4,4]$ = Initial value of q_i = 2, 3, 1, 1, 1, then root all zero. and initial cost is also all zero.

$j-i=0 \Rightarrow c[0,0], c[1,1], c[2,2], c[3,3], c[4,4] = 0$

$j-i=1 \Rightarrow c[0,1], c[1,2], c[2,3], c[3,4]$

$j-i=2 \Rightarrow c[0,2], c[1,3], c[2,4]$

$j-i=3 \Rightarrow c[0,3], c[1,4]$

$j-i=4 \Rightarrow c[0,4]$.

Case-I $j-i=0$

$w[0,0] = q_0 + \cdots = 2 \quad c[0,0]=0 \quad \& \quad r[0,0]=0$

$w[1,1] = q_1 + \cdots = 3 \quad c[1,1]=0 \quad \& \quad r[1,1]=0$

$w[2,2] = q_2 + \cdots = 1 \quad c[2,2]=0 \quad \& \quad r[2,2]=0$

$w[3,3] = q_3 + \cdots = 1 \quad c[3,3]=0 \quad \& \quad r[3,3]=0$

$w[4,4] = q_4 + \cdots = 1 \quad c[4,4]=0 \quad \& \quad r[4,4]=0$

Case-II $j-i=1$

$w[0,1] = w[0,0] + \overset{p_1+q_1}{3+3} \Rightarrow 2+3+3 = 8$

$w[1,2] = w[1,1] + 3+1 \Rightarrow 3+3+1 = 7$

$w[2,3] = w[2,2] + 1+1 \Rightarrow 1+1+1 = 3$

$w[3,4] = w[3,3] + 1+1 \Rightarrow 1+1+1 = 3$ and

$c[0,1] = \min_{k=1} \{c[0,0] + c[1,1]\} + w[0,1] = \min\{0+0\}+8 = 8^{(1)}$

$c[1,2] = \min_{k=2} \{c[1,1] + c[2,2]\} + w[1,2] \Rightarrow \min\{0+0\}+7 = 7^{(2)}$

$c[2,3] = \min_{k=3} \{c[2,2] + c[3,3]\} + w[2,3] \Rightarrow \min\{0+0\}+3 = 3^{(3)}$

$c[3,4] = \min_{k=4} \{c[3,3] + c[4,4] + w[3,4] \Rightarrow \min\{0+0\}+3 = 3^{(4)}$

Case-III $j-i=2$

$w[0,2] = w[0,1] + P_2 + q_2 \Rightarrow 8+3+1 = 12$

$w[1,3] = w[1,2] + P_3 + q_3 \Rightarrow 7+1+1 = 9$

$w[2,4] = w[2,3] + P_4 + q_4 \Rightarrow 3+1+1 = 5$

$$c[0,2] = \min_{k=1,2} \begin{Bmatrix} c[0,0]+c[1,2], \\ c[0,1]+c[2,2] \end{Bmatrix} + w[0,2]$$

$$= \min \begin{Bmatrix} 0+7 \\ 8+0 \end{Bmatrix} + 12 = 7+12 = 19 \text{ and}$$

$r_{0,2} = 1$

$$c[1,3] = \min_{1<k\le 2,3} \{ c[1,1] + c[2,3], \; c[1,2] + c[3,3] \} + w[1,3]$$

$$= \min \{ 0+3, \; 7+0 \} + 9 \Rightarrow 3+9 = 12 \quad \& \quad r_{1,3} = 2$$

$$c[2,4] = \min_{1<k\le 3,4} \{ c[2,2] + c[3,4], \; c[2,3] + c[4,4] \} + w[2,4]$$

$$= \min \{ 0+3, \; 3+0 \} + 5 \Rightarrow 3+5 = 8 \quad \& \quad r_{2,4} = 3 \text{ or } 4$$

Case IV $j - i = 3$

$$w[0,3] = w[0,2] + P_3 + q_3 = 12+1+1 = 14$$

$$w[1,4] = w[1,3] + P_4 + q_4 = 9+1+1 = 11$$

$$c[0,3] = \min_{1<k\le 1,2,3} \{ c[0,0] + c[1,3], \; c[0,1] + c[2,3], \; c[0,2] + c[3,3] \} + w[0,3]$$

$$= \min \{ 0+12, \; 8+3, \; 19+0 \} + 14 = 11 + 14 = 25 \quad \& \quad r_{0,3} = 2$$

$$c[1,4] = \min_{1<k\le 2,3,4} \{ c[1,1] + c[2,4], \; c[1,2] + c[3,4], \; c[1,3] + c[4,4] \} + w[1,4]$$

$$= \min \{ 0+8, \; 7+3, \; 12+0 \} + 11 = 8 + 11 = 19 \quad \& \quad r_{1,4} = 2$$

Case V $j - i = 4$

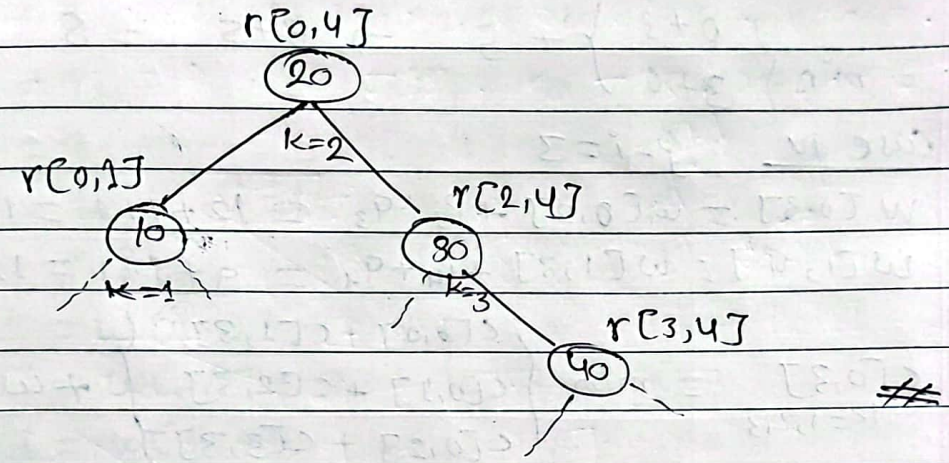
$$w[0,4] = w[0,3] + P_4 + q_4 \Rightarrow 14+1+1 = 16$$

$$\therefore c[0,4] = \min \{ c[0,0] + c[1,4], \; c[0,1] + c[2,4], \; c[0,2] + c[3,4], \; c[0,3] + c[4,4] \} + w[0,4].$$

$$= \min \begin{Bmatrix} 0+19, \\ 8+8, \\ 19+3, \\ 25+0 \end{Bmatrix} +16 \quad = 16+16 = 32 \;\&\; r_{0,4} = 2$$

$\Rightarrow$ minimum cost $= 32/16 = 2$ #

and OBST root is 2 or $r_{0,4} = 2$ then,



⇒ **Backtracking -:**

- Backtracking is a general algorithmic technique that involves exploring all possible solutions to a problem by incrementally building solution and backtracking when we determine that it cannot be complete further.
- It is used in situations, where we need to explore a large, but finite, search space to find a solution.
- It is particularly useful when the search space is too large to explore exhaustively by using heuristics ~~too large to explore e~~ eliminate paths that can't lead to valid solution.

Forward
- General solution vector one at a time.
- Add other feasible alternatives to the vector until the criteria is met.

Backward
- The criteria is not meet traverse backward to immediate node & change the direction.

- The concept of coming backward is also known as pruning.

Algorithm -:

1) Sum of subsets
2) knapsack problem.

⇒ **Sum of Subsets -:**

- The sum of subset problem is a combinational optimization problem that involves finding all possible subset of a given set of elements whose

sum equal a target value. this problem uses solved by using backtracking method.:

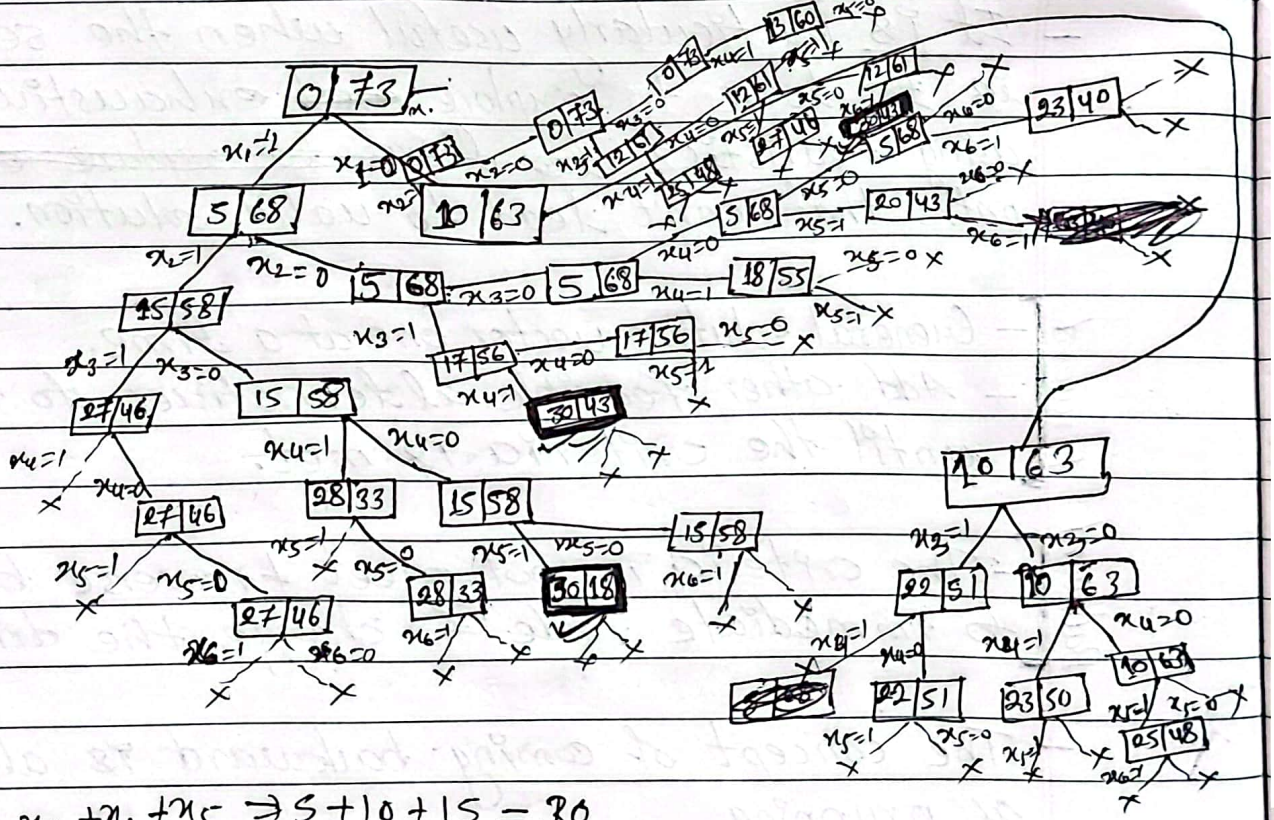
Example :- $w[1:6] = \{5, 10, 12, 13, 15, 18\}$

$n = 6$ and Sum of subset $m = 30$.

Sol

we make total path $2^n = 2^6 = 64$ possible construct tree using backtracking method.

$x_1 = 5$, $x_2 = 10$, $x_3 = 12$, $x_4 = 13$, $x_5 = 15$ & $x_6 = 18$



$\Rightarrow x_1 + x_2 + x_5 \Rightarrow 5 + 10 + 15 = 30$

$x_1 + x_3 + x_4 \Rightarrow 5 + 12 + 13 = 30$

$x_3 + x_6 \Rightarrow 12 + 18 = 30$ //

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

$$= \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

⇒ **Knapsack Algorithm-:**

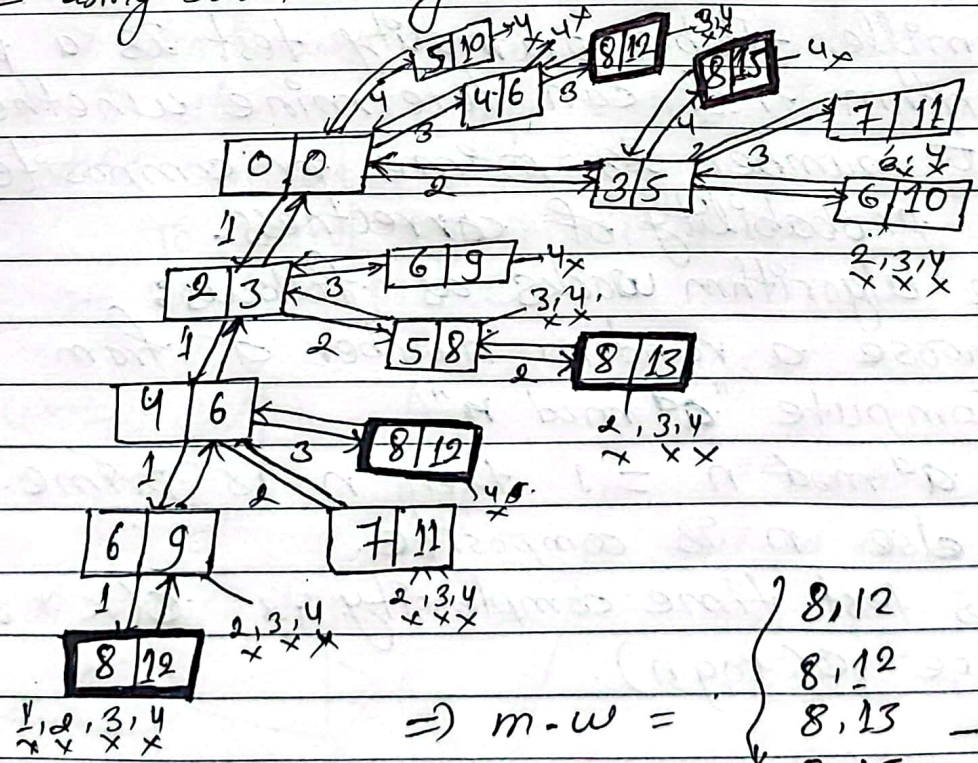
- The knapsack problem is a type of optimization problem where a set of items with different weights and values need to be selected to fill a knapsack of limited capacity.
- The objective is to maximize the total value of the selected items while not exceeding the capacity of the knapsack.
- This knapsack problem using technique is back-tracking.

Example-:

i	1	2	3	4	
w	2	3	4	5	kg
p	3	5	6	10	

max weight $m = 8$ kg.

Sol^n using backtracking on tree;



⇒ m - w = 8, 12 / 8, 12 / 8, 13 / 8, 15 / 8, 12

→ 2, 4 ⇒ 0 1 0 1

⇒ 3 + 5 = 8

∴ p = 5 + 10 = 15 max

Analysis-:

$O(2^n)$ worstcase

⇒ Randomized Algorithm -:

1. Average cost is minimum than other algorithm
2. It might not provide the correct output.
   - To table point 2, we should embed some deterministic approach within Randomization.

eg. • monte Carlo.
• Las Vegas
   - primality Test.
      ↳ miller Robins Algorithm.

⇒ Primality Testing -:

- To perform primality testing using randomized algorithm is called the miller-Robin primality.
- The miller-Robin primality test is a probabilistic algorithm that can determine whether an input number is prime or composite with a high probability of correctness.
- The algorithm works as follows:
  • Choose a random number a from n.
  • Compute "a^d mod n"
  • If $a^d \bmod n = 1$ then n is prime.
    else n is composite.

This test time complexity is $O(k * \log^3 n)$ & space $O(\log n)$.

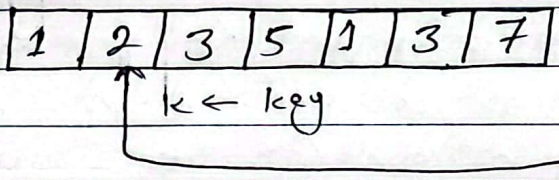
⇒ Identify the Repeated element -:

- Create an empty hash table.
- for each element in the empty array.

- check element in hash table return element is repeated else element inserted in hash table and not repeated.
- If no repeated element found after processing all element in the array, return null.

example:

n = {1, 2, 3, 5, 1, 3, 7}



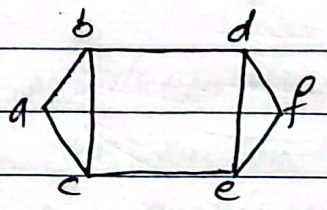
1	
2	
3	
4	
5	
6	
7	

Hash Table.

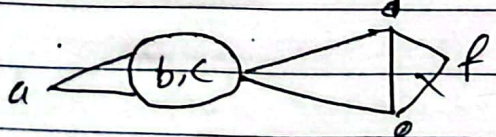
- Karger's Algorithm:-
- Randomized Algorithm
- provide Graph G = (V, E)
- Create sub-graph with minimum cut of edge.
- It is also known as min-cut problem.
- It is also called monte carlo algorithm.

  used in computer Network.
  - To control ~~cong~~ congestion control
  - To set protocol.

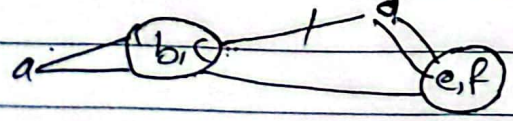
eg G =



cut b to e then



cut e to f then



cut b, c to d then

cut a to b, c, d.

$\Rightarrow \boxed{V_1 \cap V_2 = \phi}$ It

Unit-2 : Computational Complexity Theory :

⇒ Complexity Theory :

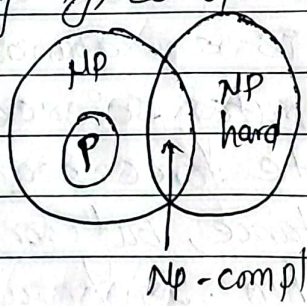
- In a computer science or mathematical problem is computable. If it can be solved in principle by a computing device. It is also known as solvable, decidable.
- There is an extensive classification of computable problems into computational complexity classes, according to how much computational - as a function of the size of problem instance is needed to answer that instance.
- It concerned with the resources, such as time and space, needed to solve computational problem.
- There is an abundance of problems from mathematics & computer science that are trivially solvable by brute force search of an exponential number of instance, but for which currently no efficient algorithm is known.
- Complexity theory is the appropriate setting for the study of such problems.

- Given a problem, two situations exist, either we can solve the problem or there exists no solution at all. If any problem can be solved the question rises above its complexity. Any problem in computer science is classified mainly in terms of computability theory and complexity theory.

⇒ Complexity classes -!

- In a computer science there exist some problems, whose solutions are not yet found, the problems are divided into classes known as complexity classes.
- In a complexity theory, a complexity class is a set of problems with related complexity. These classes help scientists to group problems based on how much time and space they require to solve problems and verify the solution.
- Time complexity requires how much number of steps to completion a problem. and space complexity requires how much memory are useful in organizing similar type of problem.
- There are following types of complexity classes

- P class
- NP class
- NP - hard
- NP - Complete.



⇒ P - class -!

- The P contains decision problems (problem with yes or No answer) that are solved by deterministic turing machine in polynomial time.
- It is an types of problem an algorithm would take a polynomial amount of time to solve the problem (easy).
- The Big-O notation of these problems is always a polynomial (i.e. $O(1)$, $O(n)$, $O(n^2)$) in general. then $T(n) = O(c * n^k)$ for all
- P problem, where $c > 0$ & $k > 0$ and both c & k are constant in n input size.

$\therefore O(n^k)$ #

- This class contains many natural problems:
  - Calculating the greatest common divisor.
  - Finding a maximum matching.
  - Decision versions of linear programming.
- problem are easy & quick to solve in polynomial time.

⇒ NP-class:-
- NP - class consists of those problems that are verifiable in polynomial time.
- NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information.
- Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.
- In this class every problem can be solved in exponential time using exhaustive search.

- The NP in NP class stands for Non-deterministic polynomial time. It is the collection of decision problem that can be solved by a non-deterministic machine in polynomial time.
- The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- problems of NP can be verified by a Turing machine in polynomial time.

eg. Boolean satisfiability problem (SAT)
- Hamiltonian path problem
- Graph coloring.

- problems are quick to verify but slow to solve.

⇒ NP-hard :

- An NP-hard problem is at least as hard as the hardest problem in NP and it is a class of problems such that every problem in NP reduces to NP-hard.
- All NP-hard problems are not in NP.
- It takes a long time to check them. This means if a solution for an NP-hard problem is given then it take a long time to check whether it is right or not.
- A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial time reduction from L to A.
- NP-hard problems are slow to verify, slow to solve and can be reduced to any other NP problem
- All NP-hard problems are not in NP & it takes a long time to check them.

eg.
- Halting problem
- Qualified Boolean formulas
- No Hamiltonian cycle.

⇒ NP-complete :

- A problem is NP-complete, if it is both NP and NP hard. NP-complete problems are the hard problems in NP.
- NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
- If one could solve an NP-complete problem in polynomial time, then one could also solve any NP-problem in polynomial time.

- Decision version of 1/0 knapsack
- Hamiltonian cycle
- Satisfiability
- Vertex cover.

- NP-complete problems are also quick to verify slow to solve and can be reduced to any other NP-complete problem.
- A problem that is NP and NP-hard is NP-complete.

⇒ Decision Problems:-

- A decision problem is the problem of determining an answer to a class of yes/no questions about some objects of interest (graph, DFA's, CFG's, TM's; Integers, Boolean formulas, etc.) (0/1, T/F or accept & reject).

⇒ Language Recognition Problems:-

- observe that a decision problem can also be thought of as a language recognition problem. we could define a language L.

  $L = \{\langle G, k \rangle : G$ has a MST of weight at most $k\}$

- This set consists of pairs, the elements is graph (e.g. the adjacency matrix encoded as a string) followed by an integer k encoded as a binary number.
- any thing broken into some how string of bits.
- when presented with an input string (G, k), the algorithm would answer "yes" if $(G, k) \in L$. Implying that G has a spanning tree of weight at most k, and "No" otherwise.

2.2. Problem Reduction :-

⇒ Reduction :-

- A reduction is a transformation of one problem into another.
- A reduction of decision problem Q_1 to decision problem Q_2 is a mapping of every instance q_1 of problem Q_1 to an instance q_2 of problem Q_2 such that q_1 is 'yes' if and only if q_2 is 'yes'.
- If Q_1 reduces to Q_2 in polynomial time and Q_2 reduces to Q_3 in polynomial time, then Q_1 reduces to Q_3 in polynomial time.

eg. An instance of the Hamiltonian problem is a graph G with vertices v_1 and v_n.
∴ construct a weighted graph G' by adding weight 1 to all edges of G.
eg Travelling salesman problem.

⇒ polynomial Time Reduction :-

- A polynomial time reduction is a mapping reducts that can be computed in polynomial time
- Suppose that A and B are two language.
- function f is a polynomial time reduction from A to B if both of the following are true:
  a) f is computable in polynomial time.
  b) for every x, $x \in A \Leftrightarrow f(x) \in B$.
- say that $A \leq_p B$ if there exists a polynomial time reduction from A to B.

eg we saw the trivial reduction $f(x) = x+1$ from the set of even integers to the set of odd integers. that is a polynomial time reduction because.

- There is an algorithm that adds 1 to n digit number in time $O(n)$.
- For every integer x, x is even $\Leftrightarrow$ $x+1$ is odd.

$\Rightarrow$ Cook's Theorem:-

- Cook's Theorem states that the Boolean satisfiability problem is NP-complete. that is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of deterministic whether a Boolean formula is satisfiable.

Corollary: SAT $\in$ P if and only if $P = NP$.

Proof There are two parts to proving that the Boolean satisfiability problem (SAT) is NP-complete.

- One is to show that SAT is an NP problem.
  - SAT is NP because any assignment of Boolean values to Boolean variables that is claimed to satisfy the given expression can be verified in polynomial time by a deterministic Turing machine.
- The other is to show that every NP problem can be reduced to an instance of a SAT problem by a polynomial-time many-one reduction.
  - Suppose that a given problem in NP can be solved by the nondeterministic Turing machine $M = (Q, \Sigma, s, F, \delta)$ where.
    $Q \rightarrow$ set of states
    $\Sigma \rightarrow$ Alphabet

$s \in Q$ → set of initial states.
$F \subseteq Q$ → set of accepting states.
and $\delta \subseteq ((Q|F) * \Sigma) \times (Q \times \Sigma \times \{-1, +1\})$ is transition relation. and M accepts or rejects an instance of the problem in time $p(n)$ p-polynomial with size of instance n.
For each input, I, we specify a Boolean expression which is satisfiable if and only if the machine M accept I.
- Boolean expression uses the variables set such that, $q \in Q$, $-p(n) \leq i \leq p(n)$, $j \in \Sigma$, and $0 \leq k \leq p(n)$
- If there is an accepting computation for M on input I, then B is satisfiable by assigning their intended interpretations. On the other hand, if B is satisfiable, then there is an accepting computation for M on input I that follows the steps indicated by the assignments to the variables.
- There are $O(p(n)^2)$ Boolean variables, each encodeable in space $O(\log p(n))$. The number of clauses is $O(p(n)^3)$ so the size of B is $O(\log(p(n)) p(n)^3)$. Thus the transformation is certainly a polynomial-time many-one reduction, as required.

⇒ SAT:-
- SAT is also called Boolean satisfiability problem.
- A propositional logic formula φ is called satisfiable if there is some assignment to its variable that makes it evaluate to true.
- $p \wedge q$ is satisfiable i.e. for $p \Rightarrow 1$ & $q \Rightarrow 1$ $p \wedge q$ is true.
i.e. satisfiable.

- $p \wedge \neg p$ is not satisfiable, But no value of p could make boolean function ($p \wedge \neg p$) true so not satisfiable.
- SAT is NP complete.

⇒ Prove that formula satisfiable is a NP class:

Proof SAT ∈ NP

- Given an assignment for $x_1, x_2, \ldots x_n$. you can check if: $\phi(x_1, x_2, \ldots x_n) = 1$ in polynomial-time by evaluating a formula on a given assignment. Hence SAT is in NP.

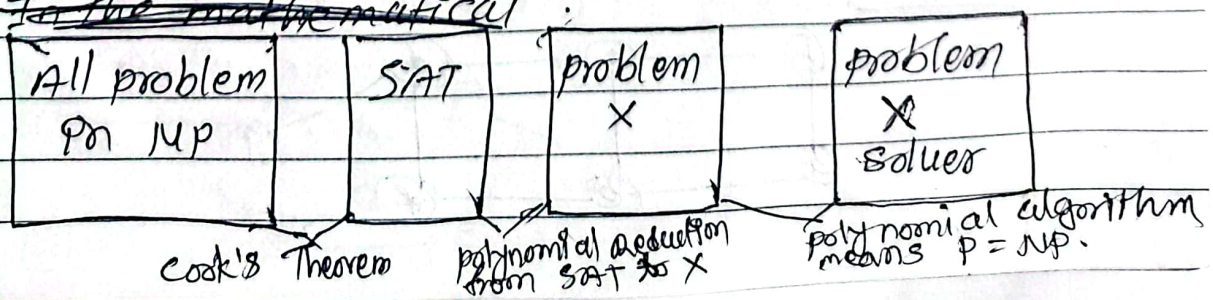
Proof SAT ∈ NP-hard-.

⇒ 3SAT -:

- problem: Given a CNF where each clause has 3 variables; decide whether it is satisfiable or not.
- $(x1 \vee x_2 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$
- A language 3SAT is $\{\psi / \psi$ is satisfiable 3 CNF formula $\psi\}$.
- 3 SAT is NP complete.

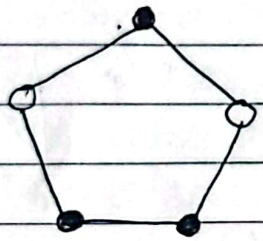
⇒ ~~Steps:~~ ~~Verification~~ -:

- ~~For the mathematical~~ :



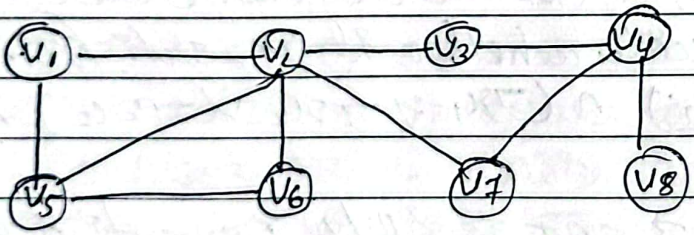
⇒ Vertex cover :-

- In the mathematical discipline of graph theory, "A vertex cover (sometime node cover) of a graph is a subset of vertices which "covers" every edge.
- An edge is covered if only of its endpoint is chosen.
- In other words "A vertex cover for a graph G is a set of vertices incident to every edge in G."
- The vertex cover problem: what is the maximum size vertex cover in G?
- problem: Given graph $G=(V,E)$, find smallest $V' \subseteq V$ such that if $(u,v) \in E$, then $u \in V'$ or $v \in V'$ or both.



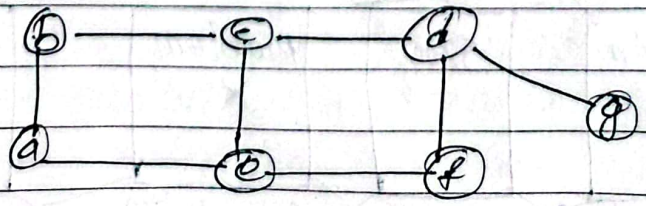
This problem can be solved by
① Greedy Approach.
② Approx optimal solution.

eg



vertex set $\{v_1, v_3, v_5, v_6, v_7, v_8\}$ is a vertex cover.
vertex set $\{v_2, v_4, v_5\}$ is vertex cover.
vertex set $\{v_2, v_3, v_8\}$ is not a vertex cover since edge $\{v_4, v_7\}$ is not covered.

Example :



Sol: $C = \phi$

$E' = \{(a,b), (b,c), (c,d), (c,e), (e,f), (d,f), (d,e), (d,g)\}$

Initially let select (b,c) edge, where,

$u = b$

$v = c$

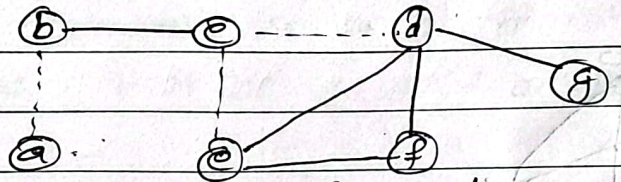
$C = \phi \cup \{b,c\}$

$= \{b,c\}$

Now, remove from E' every edge incident

$E' = \{(d,e), (e,f), (d,f), (d,g)\}$

~~Again, let's select (e,f) vertex.~~



Again, let's select (e,f) vertex where.

$u = e$

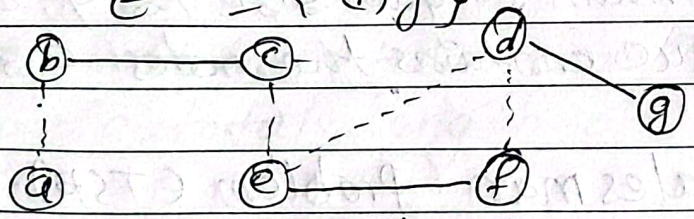
$v = f$

$C = \{b,c\} \cup \{e,f\}$

$= \{b,c,e,f\}$

Now, remove E' ~~edge~~ every edge incident.

$E' = \{d,g\}$



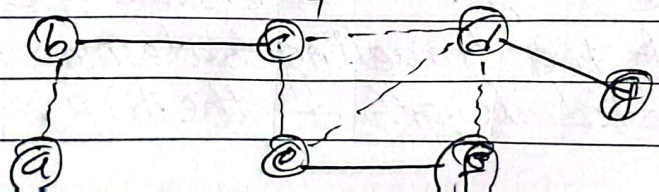
Again let select (d,g) vertex where,

$u = d$

$v = g$

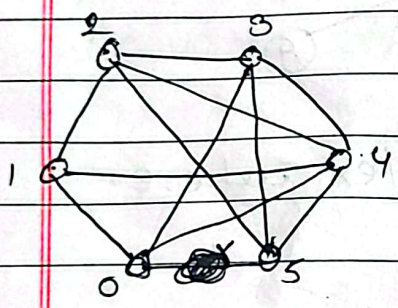
$C = \{b,c,e,f\} \cup \{d,g\} = \{b,c,d,e,f,g\}$

$E' = \phi$



⇒ maximum clique problem:-

- In a graph G, a subset of vertices fully connected to each other, i.e. a complete subgraph of G is called clique
- The maximum clique problem is how large is the maximum-size clique in a graph?
- In another words, given a graph of vertices some of which have edges in between them, the maximal clique is the largest subset of vertices in which each point is directly connected to every other vertex in the subset.

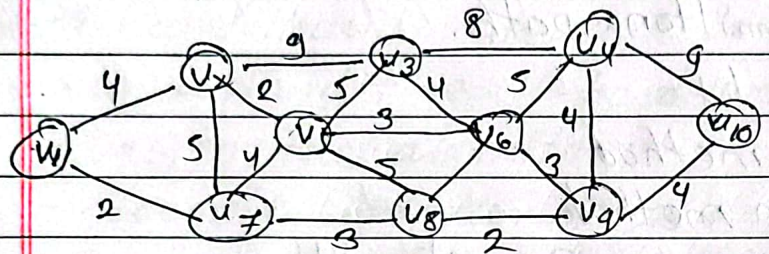


- Given a graph with vertices 0, 1, 2, 3, 4, 5. Here, the maximum clique problem is the largest subset of graph which is complete.
- And the maximum clique size is 4, and the maximum clique contains the nodes 2, 3, 4, 5.

⇒ Travelling Salesman problem (TSP):-

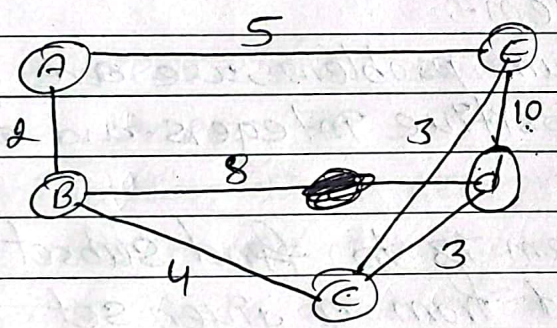
- The travelling salesman problem consists of a salesman and a set of cities. the salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and return to the same city. the challenge of the problem is that the travelling salesman wants to minimize the total length of the trip.

- It find shortest possible route.
- Given a weighted graph G, find the minimum cost path from v_1 to v_n which visits each vertex exactly once.
- No known polynomial time algorithm for solving this problem.



Find the minimum cost path from v_1 to v_{10} which visits each vertex exactly once.

Eg

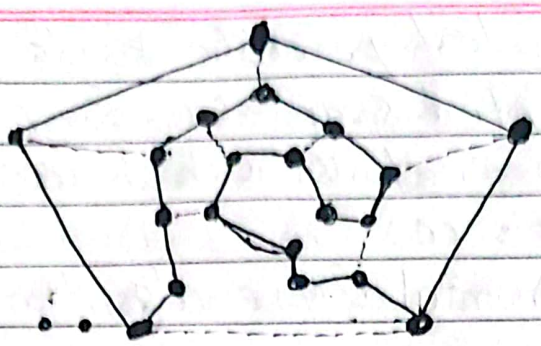


From path-1 : {A, B, C, D, E, A} and path-2 - {A, B, C, E, D, A} pass all the vertices but path-1 has total length of 24 and path-2 has a total length of 31 #

- It solve brute-force approach & Dynamic approach.

⇒ Hamiltonian cycle :

- Hamiltonian cycle or circuit is a graph cycle (i.e. closed loop) through a graph that visits each node exactly once (except for the vertex that is both the start & end, which is visited twice).



Eg: Hamilton path.

- It can solved by
  - Brute force method
  - Frank Rubin method
  - Dynamic programming algorithm
  - monte carlo Algorithm

⇒ Subset Sum problem:-

- In the subset-sum problem, we are given a finite set S of positive integers and an integer target $t > 0$.
- subset sum problem is to find subset of elements that are selected from a given set S whose sum adds up to a given number k.
- mathematically, subset sum: {(S, t) : there exists a subset S' . $S' \subseteq S$ such that $\sum_{s \in S'} S$.

Eg $a = \{12, 1, 3, 8, 20, 50\} \Rightarrow t = 44 \rightarrow$ true, $t = 12 \rightarrow$ False

2.3:- NP-hard Code Generation problems:-

- The function of a compiler is to translate programs written in some source language into an equivalent assembly language or machine language program.
- A translation of an expression into the machine or assembly language of a given machine is opti-

mal if and only if it has a minimum number of instructions.

- A binary operator ⊙ is commutative in the domain D iff $a \odot b = b \odot a$ for all a & b in D.
- It refers to the task of generating source code that solves an NP-hard problem.
- It has following steps:
  - understand the NP-hard problem.
  - Design the algorithm.
  - choose a programming language.
  - Implement the algorithm
  - Test and validate
  - optimize if needed and refine and iterate.

⇒ Code generation with common subexpression:

- Code generation with common subexpression elimination is a technique used to optimize the generated code by identifying and eliminating redundant computations.
- The goal is to identify expressions that are computed multiple times and replace them with a single computation, storing the result in a temporary variable.

⇒ ~~Parallel~~ Implementing Parallel Assignment Instructions:

- It execution of multiple assignment statements simultaneously, taking advantage of parallelism to improve performance.
- the format $(v_1, v_2, \ldots v_n) = (e_1, e_2, \ldots, e_n)$ where v_i's are distinct variable name & the e's are expressions.
- multiple independent assignments can be executed concurrently.

2.4: Coping with NP-completeness:

⇒ performance ratios for approximation algorithm:

- The performance ratio ~~for approximation algorithm~~ are used to measure the quality of approximation algorithms. An approximation algorithm is design to solve optimization problems with the goal of finding a near-optimal solution that is close to optimal solution but can be computed more ~~effectively~~ efficiently.
- It defined as worst-case ratio between the solution produced by the algorithm and the optimal solution. It quantifies how close the approximation algorithm's solution is to the optimal solution. The smaller the performance ratio, the better the approximation algorithm.

⇒ Approximated Algorithms:

- It is also known as approximation algorithms, are algorithms designed to find near-optimal solutions for optimization problems in a computationally efficient manner.
- It strike the balance between solution quality and computational complexity.
- performance guarantee
- Greedy algorithms
- Heuristic algorithms.
- Randomized algorithm
- Polynomial time complexity

Unit-3 : Online and PRAM Algorithms :

⇒ Online Algorithms :

- An online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the start.
- An offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand. eg selection sort requires that the entire list be given before it can sort it, while insertion sort doesn't).
- Online algorithm is one that can process its input piece by piece in a serial fashion. i.e. in the order that the input is fed to the algorithm, without having the entire input available from the start.
- An offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand.
- selection sort eg is this one.
- Because it doesn't know the whole input, an online algorithm is forced to make decisions that may later turn out not to be optimal, and the study of online algorithm has focused on the quality of decision making that is possible in this setting.
- Competitive analysis formalises this idea by comparing the relative performance of an online and offline algorithm for the same problem instance.

- An on-line algorithm A is c-competitive if there is a constant b for all sequence S of operations:

$$A(S) \leq c \; OPT(S) + b.$$

- where A(S) is the cost of A on the sequence S and OPT(S) is the optimal off-line cost for the same sequence.
- Competitive ratio is a worst case bound.

⇒ Ski Rental problem:-

- Assume that you are taking ski lessons.
- It is also known as Buy-Rent problem.
- The training period is unknown.
- Decision has to be made whether to continue rent or buy at a particular instance.
- After each lesson you decide (depending on how much you enjoy it, and what is your bones status) whether to continue to ski or to stop totally.
- You have the choice of either renting skis for 1$ a time or buying skis for y$.
- Will you buy or rent?
- If you knew in advance how many times you would ski in your life then the choice of whether to rent or buy is simple. If you will ski more than y times then buy before you start, otherwise always rent.
- The cost of this algorithm is min(t, y).
- This type of strategy, with perfect knowledge of the future, is known as an offline strategy.

- In practice, you don't know how many times you will ski. what should you do?
- An online strategy will be a number k such that after renting k-1 times you will buy skis (just before your k^{th} visit).
- claim: setting k = y guarantees that you never pay more than twice the cost of the offline strategy.
- Example: Assume y = 7$ thus, after 6 rents, you buy. Your total payment: 6+7 = 13$.

⇒ Theorem:
- setting k = y guarantees that you never pay more than twice the cost of the offline strategy.
- proof: When you buy skis on your k^{th} visit, eve if you quit right after this time, t >= y.
  - Your total payment is k-1 + y = 2y-1.
  - The offline cost is min(t, y) = y.
  - The ratio is (2y-1)/y = 2-(1/y).
- we say that this strategy is [2-(1/y)] -compe titive.

⇒ conclusion:
- when balancing small incremental costs against a big one-time cost, you want to delay spending the big cost until you have accumulated roughly the same amount in small costs.

⇒ Disk scheduling Algorithm:
- Access time = Seek Time + latency time.
- Seek Time → Head positioning
- Latency Time → Disk position.

1) FCFS (First come First Serve)
2) SSTF (Shortest Seek Time First)
3) SCAN (ELEVATOR)
4) C-SCAN
5) LOOK
6) C-LOOK.

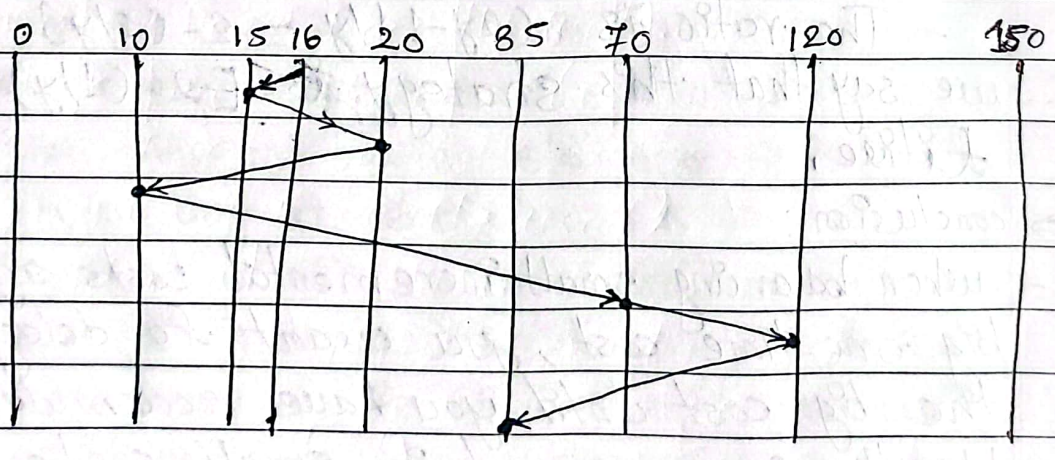
← Left Inward
→ Right Outward

Example:- suppose the provided sequence of requests are:

Sector	S_1	S_1	S_2	S_3	S_1	S_4
Track	15	20	10	70	120	35

Latency Time = 0.2 ms
seek Time = 0.1 ms
max Track = 150
Head position = 16

FCFS :-

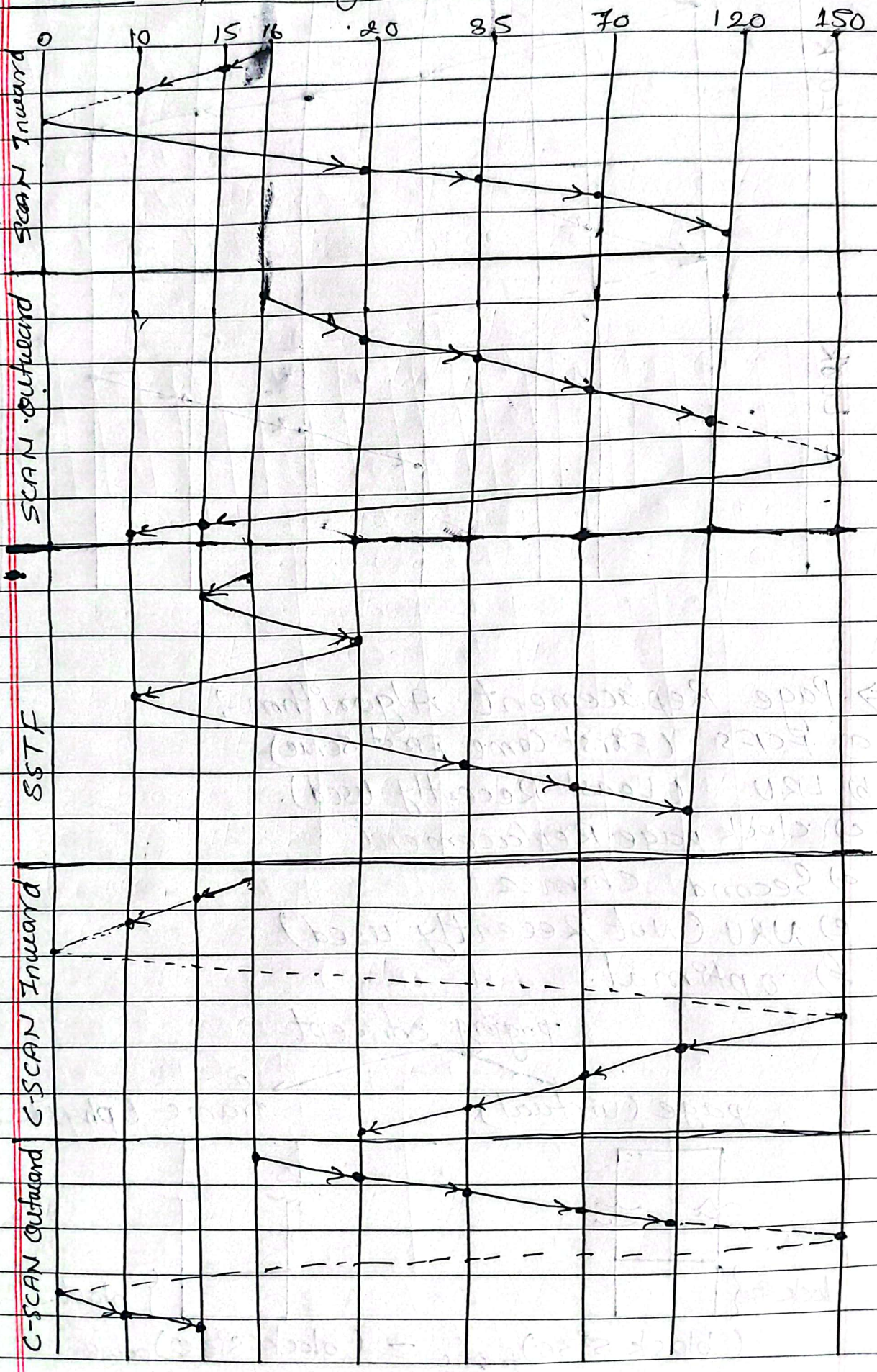


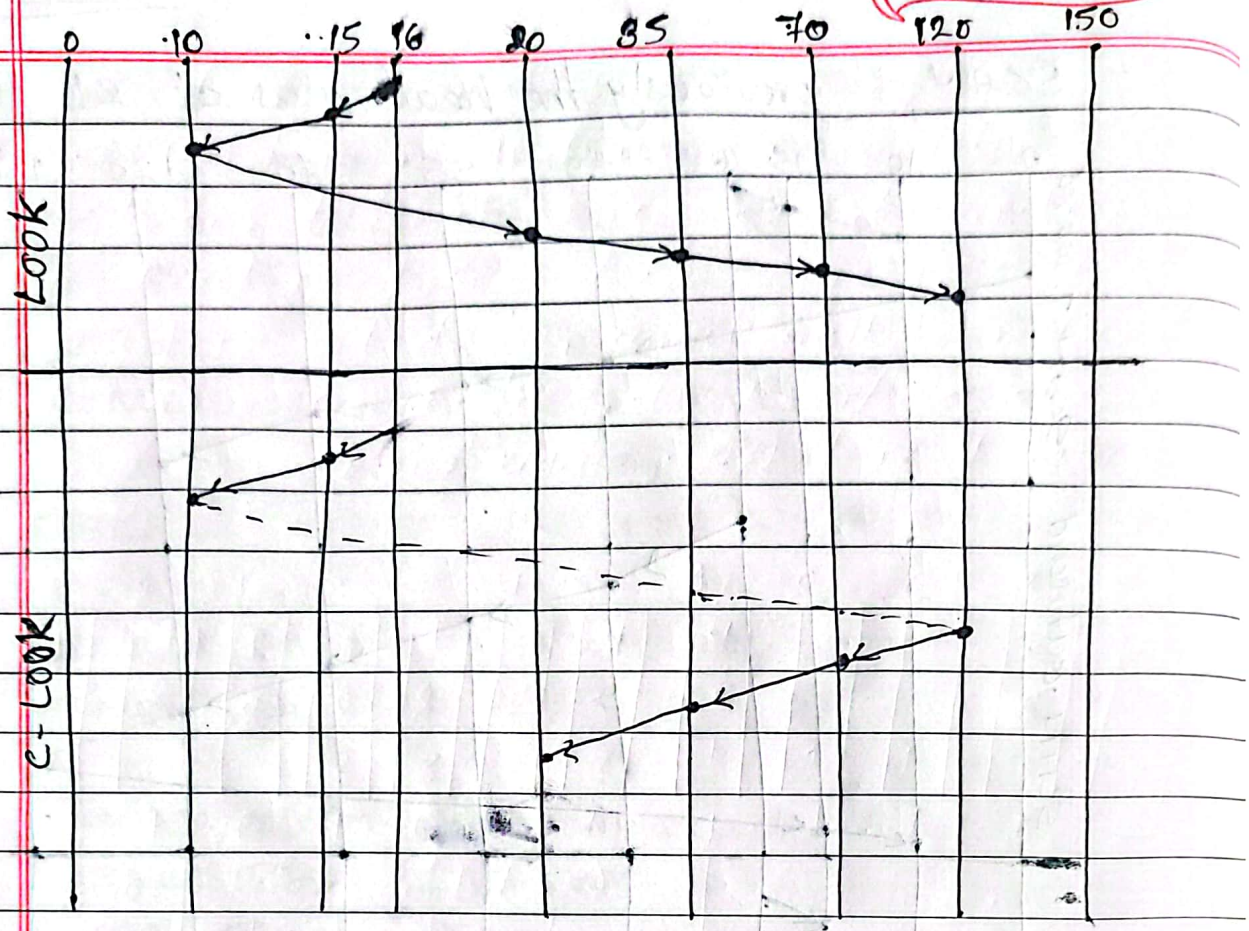
Access Time = ((16-15) × 0.1) + ((20-15) × 0.1) + (((20-10) × 0.1 + 0.2) + (((70-10) × 0.1) + 0.2) + (((120-70) × 0.1) + 0.4) + (((120-35) * 0.1) + 0.6

= 22.5.

~~SCAN → previously the head was at 21.~~

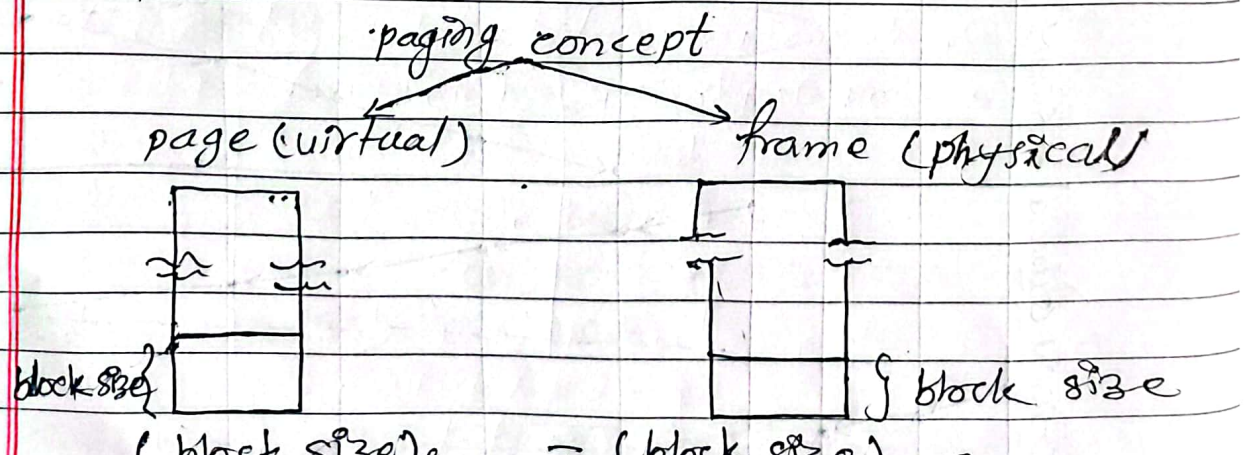
SCAN :- previously the head was at 21.





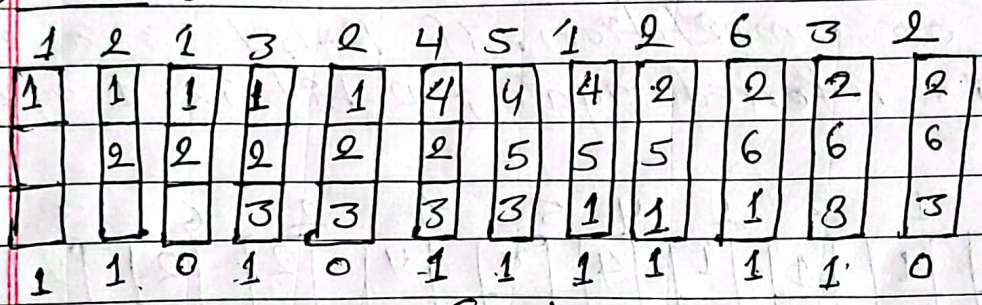
⇒ Page Replacement Algorithm :-

a) FCFS (First Come First Serve)
b) LRU (Least Recently Used).
c) clock page Replacement
d) Second chance
e) NRU (Not Recently used)
f) optimal.



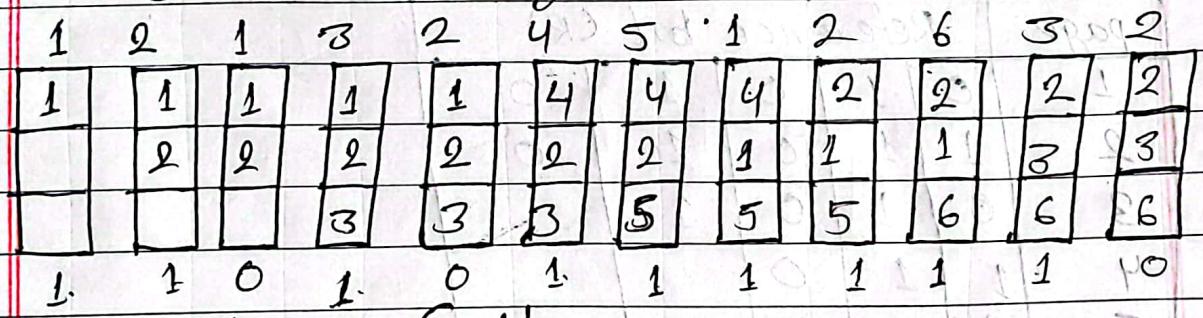
Example :- provided sequence.
Request — 1, 2, 1, 3, 2, 4, 5, 1, 2, 6, 3, 2
Size of page 3

a) FCFS - (First Come First Serve)



Total page fault = 9 #

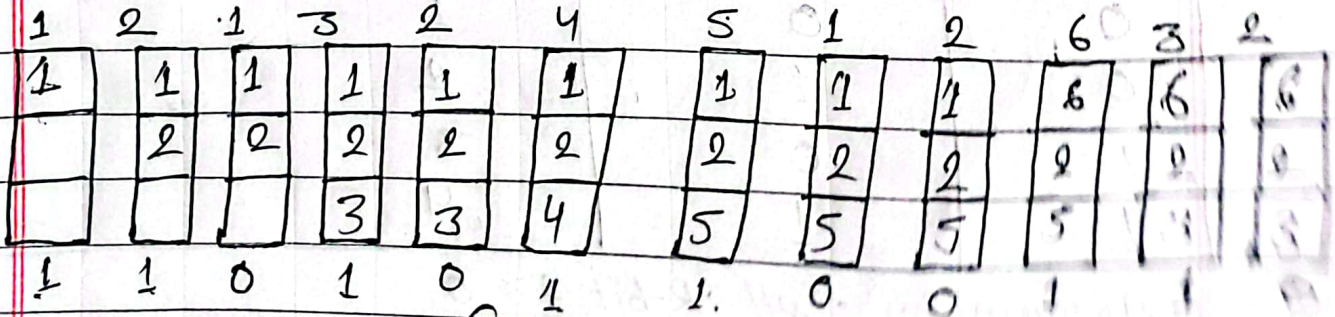
b) LRU (Least Recently used) :-



Total page fault = 9

- Implemented Linux family because it is free from Balady's Anamoly provided stack data structure is used. (page size increased condition → Balady's Anamoly
((LFD) → Longest Forward Distance).

A) Optimal :- (Replace the page not in demand in the future
Assumption based on nearest request.



Total page fault = 7 //

d) Second chance :-

- If R-bit is 0, that page is candidate for replacement which is carried out w.r.t. FCFS. If R-bit is 1, second chance is provided & R-bit is set to 0, which will be replaced based on competition. If R-bit is already 1 and is referred again, R-bit is 1 again.

1	2	1	3	2	4	5	1	2	6	3	2
1	1	1	1	1	4	4	4	2	2	2	2
	2	2	2	2	2	5	5	5	6	6	6
			3	3	3	3	1	1	1	3	3
1	1	0	1	0	1	1	1	1	1	1	0

page	Reference bit (R)				
1	0	1	0	1	0
2	1	1	0	1	0
3	0	1	0	1	
4	1	1	0		
5	1	1	0		
6	0	1	0		

∴ Total page fault = 9 #

e) NRU (Not Recently used) :-

It use two bit status framework of R & m.

Read (R)	modify (m)	priority of page Replacement
0	0	1
0	1	2
1	0	3
1	1	4

Assumption = All R-bit is 0
m bit of 3, 5, 6 is 1.

Arrival time is after every 2 sec. starting from 0. Sec. the R-bit of page will reset to 0 after every 3 sec.

1	2	1	3	2	4	5	1	2	6	3	2
1	1	1	1	1	4	4	1				
	2	2	2	2	2	5	2				
			3	3	3	3					
0	0	0	0	0	1	1					

Page	R	M	Timings (s)
1	0	0	0
2	0	0	2
1	0	0	4
3	1	1	6
2	0	0	8
4	0	0	10
5	1	1	12
1	0	0	14
2	0	0	16
6	1	1	18
3	1	1	20
2	0	0	22

#

→ NRU algorithm removes a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick than a clean page that is in heavy use.

- It is easy to understand, moderately efficient to implement, and gives a performance that, not opt

⇒ Load Balancing Algorithms:-

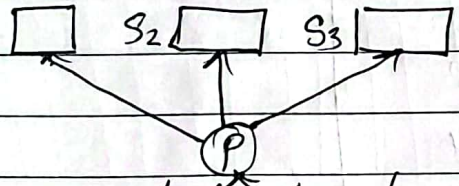
- Distribution of processes within servers, so that the stress is controlled, it is also known as stress testing.

Type :- a) Round Robin
b) weighted Load Balancing.

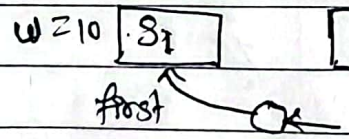
- A set of m identical machines,
- A sequence of jobs with processing times p_i ...
- Each job must be assigned to one of the machines,

Goal: schedule the jobs on machines in a way that
- minimize the makespan.

a) Round Robin
- Timing factor is used
- Selection of servers is carried out based on timing.



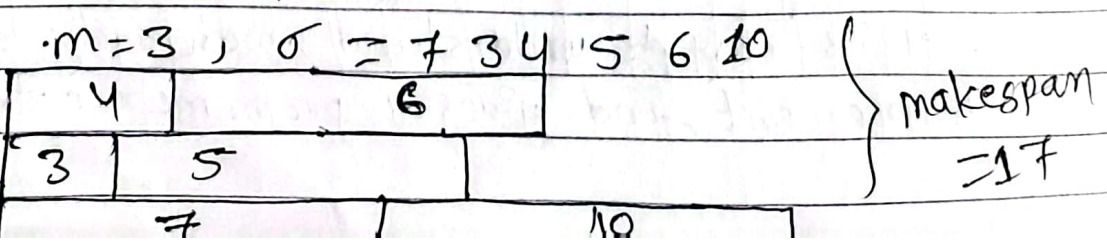
b) weighted Load Balancing:
- Every servers is provided with some weights.
- The server with highest weight is chosen first.



- How to minimize the delay of context switch between servers in Load Balancing algorithm?

⇒ List Ranking:
A greedy algorithm: always schedule a job on the least loaded machine.

m = 3, σ = 7 3 4 5 6 10



⇒ 3.2 PRAM Algorithms :-

- Let there be 100 numbers to be added.

a) How much time will a person take to add?

1 ——— 100

99

b) How much time will two person take to add?

1 ——— 50 . 51 ——— 100

49 — 1 — 49

i.e. 50 (provide 1 sec is required to add two no's)

- The addition of processors does not always reduce the run time. therefore for a system to be work optimal the number of processors have to carefully chosen.
- For this eg -

Asymptotic speed up = 99/50 ≈ 1.98 ≈ 2

Asymptotic speed up = $S(n)/T(n,p)$

where,

$S(n)$ = Sequential Run time

$T(n,p)$ = PRAM Run time

n is number of inputs

p is number of processors

Total work done $= \Theta(S(n)) = p * T(n,p)$

⇒ Efficiency :- Θ :-

$\Theta = S(n) / (p * T(n,p))$

$= S(n) / \Theta(S(n))$.

- To be work optimal it requires the efficiency to be $\Theta(1)$. ⇒ 1 //

- eg:- optional run time to sort n keys ~~$\Theta(n \log n)$~~ $= \Theta(n \log n)$. Let A be an n-processor parallel algorithm that sorts n keys in $\Theta(\log n)$ time. Let B be n^2-processor algorithm that also sorts n keys in $\Theta(\log n)$ time.
Find out which algorithm is works optimal?

I). $(\text{speed up})_A = \frac{\Theta(n \log n)}{\Theta(\log n)} = \Theta(n)$

$(\text{speed up})_B = \frac{\Theta(n \log n)}{\Theta(\log n)} = \Theta(n)$.

II) $(\text{work done})_A = n \cdot \Theta(\log n)$

$(\text{work done})_B = n^2 \Theta(\log n)$

III). $\text{efficiency } (\Theta)_A = \frac{\Theta(n \log n)}{n \cdot \Theta(\log n)} = 1$

$\text{Efficiency } (\Theta)_B = \frac{\Theta(n \log n)}{n^2 \Theta(\log n)} = \frac{1}{n}$ ∴

⇒ Prefix Computation:-

- Let Σ be the domain ~~of~~ in which binary associative operator $\oplus$ is defined.
- Any operator is binary associative if for any of three elements x, y, z from.
- $\Sigma((x \oplus y) \oplus z = (x \oplus (y \oplus z)))$ [$\oplus$ can be +, -, /, max, min, avg].

eg $\Sigma = (5, 8, -2, 7, -11, 12)$

$\oplus$ = minimum.

output = 5, 5, -2, -2, -11, -11.

- The prefix computation problem on Σ_i has an input 'n' elements from Σ_i, say $x_1, x_2, x_3, \ldots x_n$.
The problem is to compute the n elements.
$x_1, x_1 \oplus x_2, \ldots x_1 \oplus x_2 \oplus x_3 \ldots \oplus x_n$.
The output elements are preferred to as prefixes.

Example:- Let Σ_i be the set of integers and $\oplus$ be the usual addition operation.
Input = 3, −5, 8, 2, 5, 4
output = 3, −2, 6, 8, 13, 17

Example:- Let $n = 8$ and $p = 8$
Let the input to the prefix computation problem be. 12, 3, 6, 8, 11, 4, 5, 7, & $\oplus$ be addition.

step.1
processor P_1 to P_4 computes the prefix sum of 12, 3, 6, 8 to arrive at 12, 15, 21, 29.
and,
processor P_5 to P_8 computes the prefix sum of 11, 4, 5, 7 to arrive at 11, 15, 20, 27

step.2
processor P_1 to P_4 sits idle.
processor P_5 to P_8 will update their results by adding 29 to prefix to obtain 40, 44, 49, 56.

step.1

12	12, 3	12, 3, 6	12, 3, 6, 8	11	11, 4	11, 4, 5	11, 4, 5, 7
12	12, 15	12, 15, 21	12, 15, 21, 29	11	11, 15	11, 15, 20	11, 15, 20, 27

12, 15, 21, 29	11, 15, 20, 27

step.2

12, 15, 21, 29	40, 44, 49, 56

#

⇒ Analysis :-

step 1 takes $T(n/2)$

step 2 takes $O(1)$

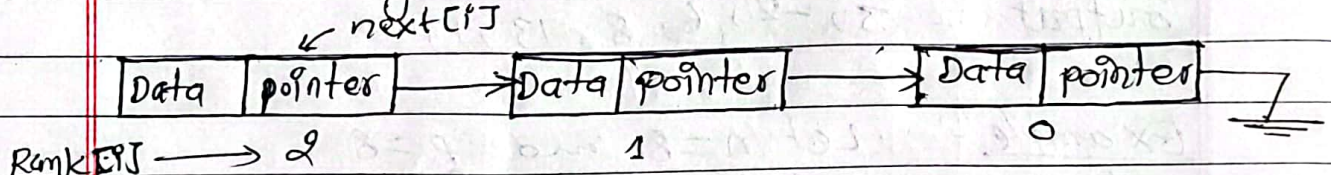
Recurrance relation is,

$T(n) := T(n/2) + O(1)$, $T(1) = 1$

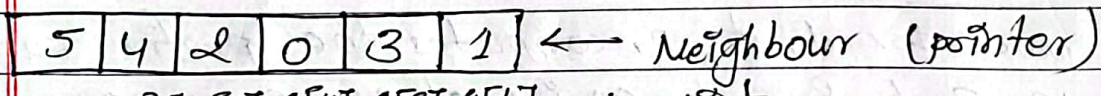
This solves to,

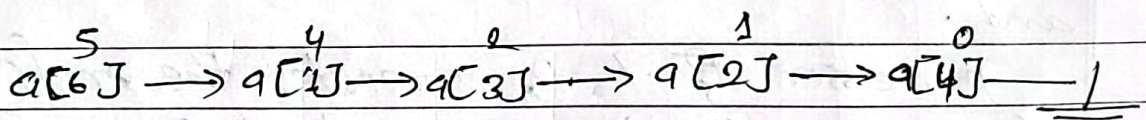
$T(n) = O(\log n)$ #

⇒ List Ranking :-



pointer jumping.





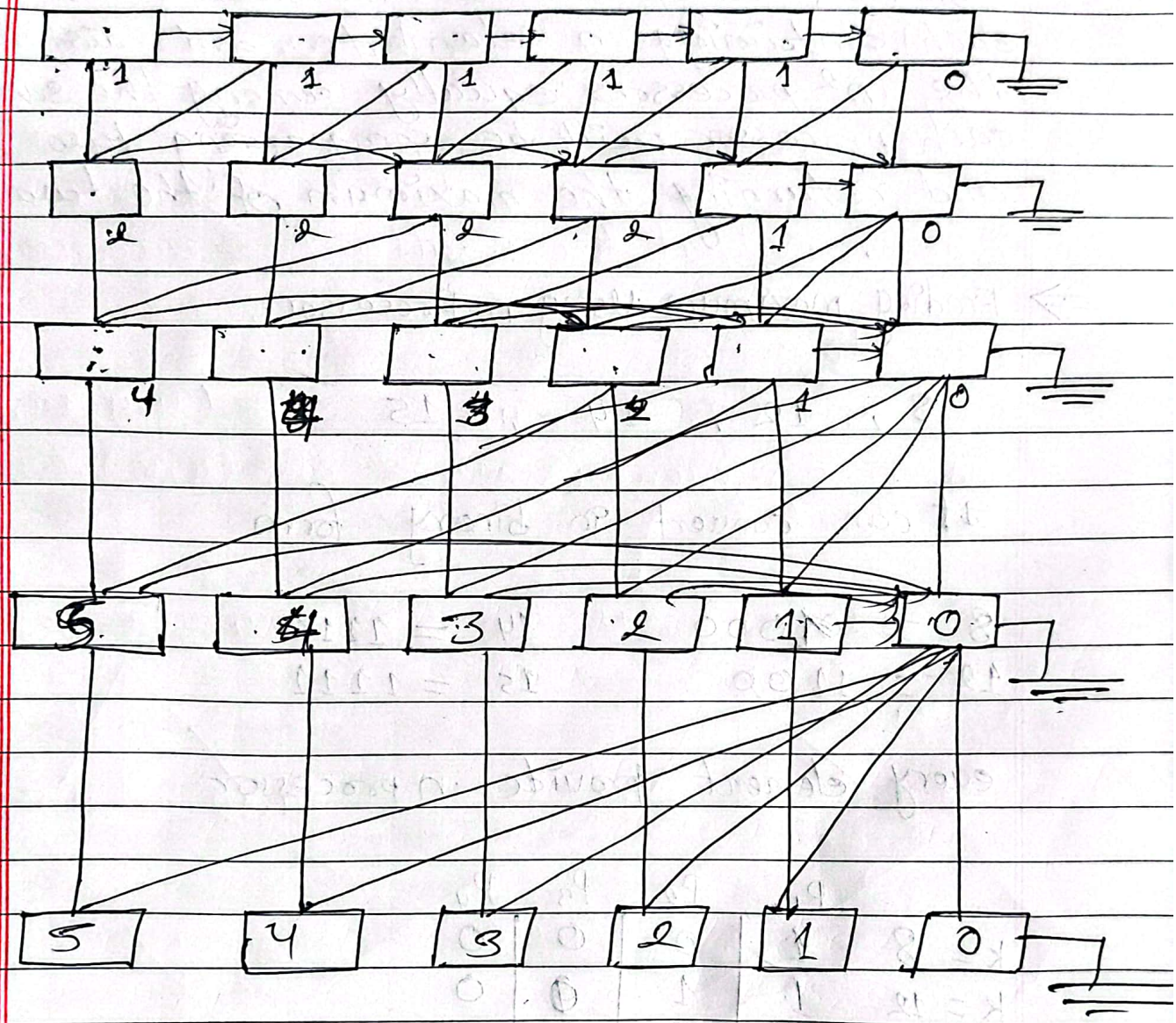
Rank	4	1	2	0	3	5
	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Algorithm :-

```
# Initialization
for (each processor i)
    if (next[i] == 0)
        Rank[i] = 0
    else
        Rank[i] = 1
```

```
# Finding Rank
while (next[i] != null)
for(each processor i)
    if (next[i] != null)
        next[i] = next[i] + (next(next[i]
        Rank[i] = Rank[i] + Rank(next
                                [i]
```

Example :-



Ther sorting rank is 5, 4, 3, 2, 1, 0 (if

sequential algorithm - O(n) - Total running time = O(log n), Total work = O(n log n)
Efficiency = O(n)/O(n log n) = 1/log n · not work optimal

⇒ maximal selection with n^2 processor :-

- maximal selection with n^2 processor refers to a parallel algorithmic problem that involves selecting the maximum element from a given set using a parallel computing system with n^2 processors. The problem assumes that the elements of the set are stored in a distributed manner across the processor.

- It is find the maximum element efficiently.
- It has array divided into n subarrays. each containing n elements. and distribute the n^2 processors equally among the subarrays. each processor will be comparing two elements and returning the maximum of the two so-on.

⇒ Finding maximum using n processor:-

8, 12, 14, 15

It can convert in binary form.

8 = 1000 14 = 1110
12 = 1100 15 = 1111

every element provide n processor.

	P_1	P_2	P_3	P_4	
K = 8	1	0	0	0	
K = 12	1	1	0	0	
K = 14	1	1	1	0	
K = 15	1	1	1	1	→ K_1, K_2 & K_3 → eliminate

P_1 → No eliminate. P_2 → K_1 only eliminate. P_3 → K_1 & K_2 eliminate.

K or $n^{1/2}$ processor
$n^{1/2}$ elements.

K_1
K_2
K_3
K_4

$T(n, p) = 1$

$T(2) = O(1)$

which is $T(n) = (\log \log n)$

Note $n = 2^{2^p} \Rightarrow 2^q = \log n \Rightarrow q = (\log \log n)$ #

Total work not optimal $O(n \log \log n)$ #

⇒ Merging

- problem: Given 2 sorted sequences $X_1 = k_1, k_2, \ldots, k_m$ and $X_2 = k_{m+1}, k_{m+2}, \ldots, k_{2m}$.
  Assume each sequence has m distinct elements, and m is an integral power of 2.
- The goal: To produce a sorted sequence of 2m elements.
- Best sequential algorithm is $O(m)$.
- For each $k_i \in X_1$, we know that it is the rank #i element in X_1. We allocate a single processor to perform a binary search on X_2 and figure out q (the number of elements in X_2 that are less than k_i). Then we know that k_i is the rank #(i+q) element in $X_1 \cup X_2$.
- For each element in X_2, a similar procedure can be used to compute its rank in $X_1 \cup X_2$.
- We can use 2m processors, one for each element. An overall rank can be found for each element using binary search in $O(\log m)$ time. Merging can be done in $O(\log m)$ time.

⇒ Odd even merge

- step 1: If $m = 1$, merge two sequence with 1 comparision.
- step 2: partition X_1 into their odd and even parts, i.e. $X_1^{odd} = k_1, k_3, k_5, \ldots k_{m-1}$ and $X_1^{even} = k_2, k_4, \ldots k_m$ similarly, partition X_2 into X_2^{odd} and X_2^{even}.
- step 3: Recursively merge X_1^{odd}, X_2^{odd} (and X_1^{even}, X_2^{even}) using m processors.

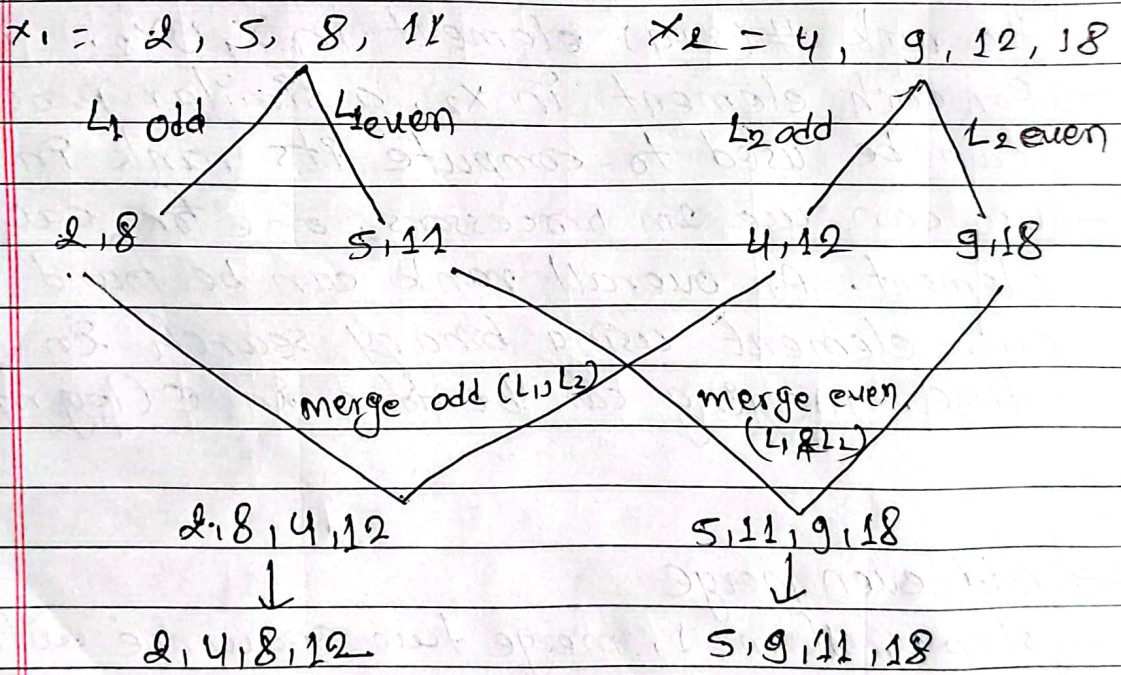
Let $L_1 = U_1, U_2, \ldots, U_m$ $\{L_2 = U_{m+1}, U_{m+2}, \ldots U_{2m}\}$ be the result.

- step 4: Form a sequence $L = U_1, U_{m+1}, U_2, U_{m+2}, U_3, U_{m+3}, \ldots, U_m, U_{2m}$ Compare every pair (U_{m+1}, U_{1+1}), i.e. $(U_{m+1}, U_2), (U_{m+2}, U_3), \ldots$ Interchange elements if they are out of order. Output the resultant sequence.

Example:-

Let $X_1 = (2, 5, 8, 11)$ & $X_2 = (4, 9, 12, 18)$

Sol^n odd = (2, 8), even = (5, 11) & odd = (4, 12), even = (9, 18) so that



Shuffling

2, [5, 4], [9, 8], [11, 12], 18

If required interchange, ll

2, 4, 5, 8, 9, 11, 12, 18

⇒ The final result = 2, 4, 5, 8, 9, 11, 12, 18 in sorted order #

⇒ Odd even merge Sort -:

- step 1: if $n <= 1$ return x
- Step 2: use n CPUs to partition input x of n elements into two lists x_1 and x_2. each with n/2 elements
- step 3: use n/2 CPUs to sort x_1 recursively and n/2 CPUs to sort x_2 recursively. Let x_1* and x_2* be the result sorted lists.
- step 4: use odd even merge to merge two sorted lists using n CPUs.

⇒ Analysis -:

$T(n) = O(1) + T(n/2) + O(\log n)$

$= \log(n) + \log(n/2) + \log(n/4) + \dots + \log(n/2^i)$; $i = \lg n$

$= \log(n) + [\log(n) - \log 2] + [\log(n) - \log 4] + \dots + [\log(n) - \log(n)]$

$<= \log(n) * \log(n)$

~~$T(n) = \log(n) * \log(n)$~~

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

Total work $O(n \log^2 n)$

It is not work optimal.

⇒ preparata's Algorithm -:

- It is known as the closest pair algorithm, is an algorithm used to find the pair of closest points in a two-dimensional plane. It was developed by franco preparata and michael Ian Shamos in 1975.
- algorithm works by dividing the plane into smaller regions and recursively finding the closest pair of of points in each region.
- short
  - divide into two parts
  - find closest
  - find minimum distance & combine result.

Unit-4:- Mesh and Hypercube Algorithms:-

⇒ Mesh Algorithm:-

- A mesh is an axb grid in which there is a processor at each grid point. The edges correspond to communication lines and are bidirectional. Each processor of the mesh can be labeled with a tuple (i, j) where $1 \le i \le a$ and $1 \le j \le b$.
- Every processor ~~can perform any of the~~ of the mesh is a RAM with some local memory. Hence, each processor can perform any of the basic operations such as subtraction, addition, multiplication, comparision, local memory access and so on. The computation is assumed to be synchronous.
- We consider square meshes, that is meshes for which $a=b$. A $\sqrt{p} \times \sqrt{p}$ mesh is shown in figure(a).
- A closely related model is the linear array. A linear array consists of p processors named $(1, 2, \ldots, p)$ connected as follows:

a) Processor i is connected to the processors $i-1$ & $i+1$ for $2 \le i \le p-1$.

b) Processor 1 is connected to processor P2 and processor p is connected to processor $p-1$.

processor 1 and p are known as boundary processors processor $i-1$ and $i-5$ are known as left neighbour (right neighbour) of i.

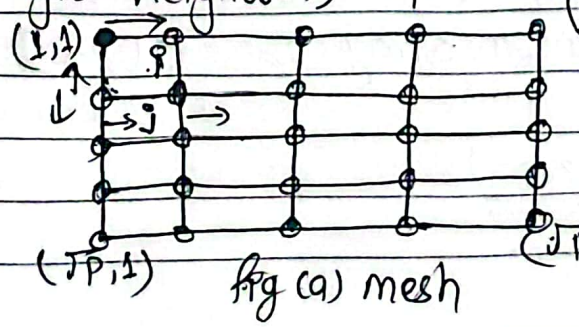


fig (a) mesh

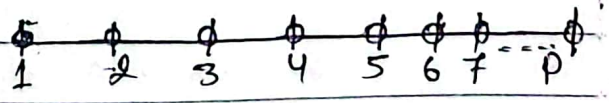
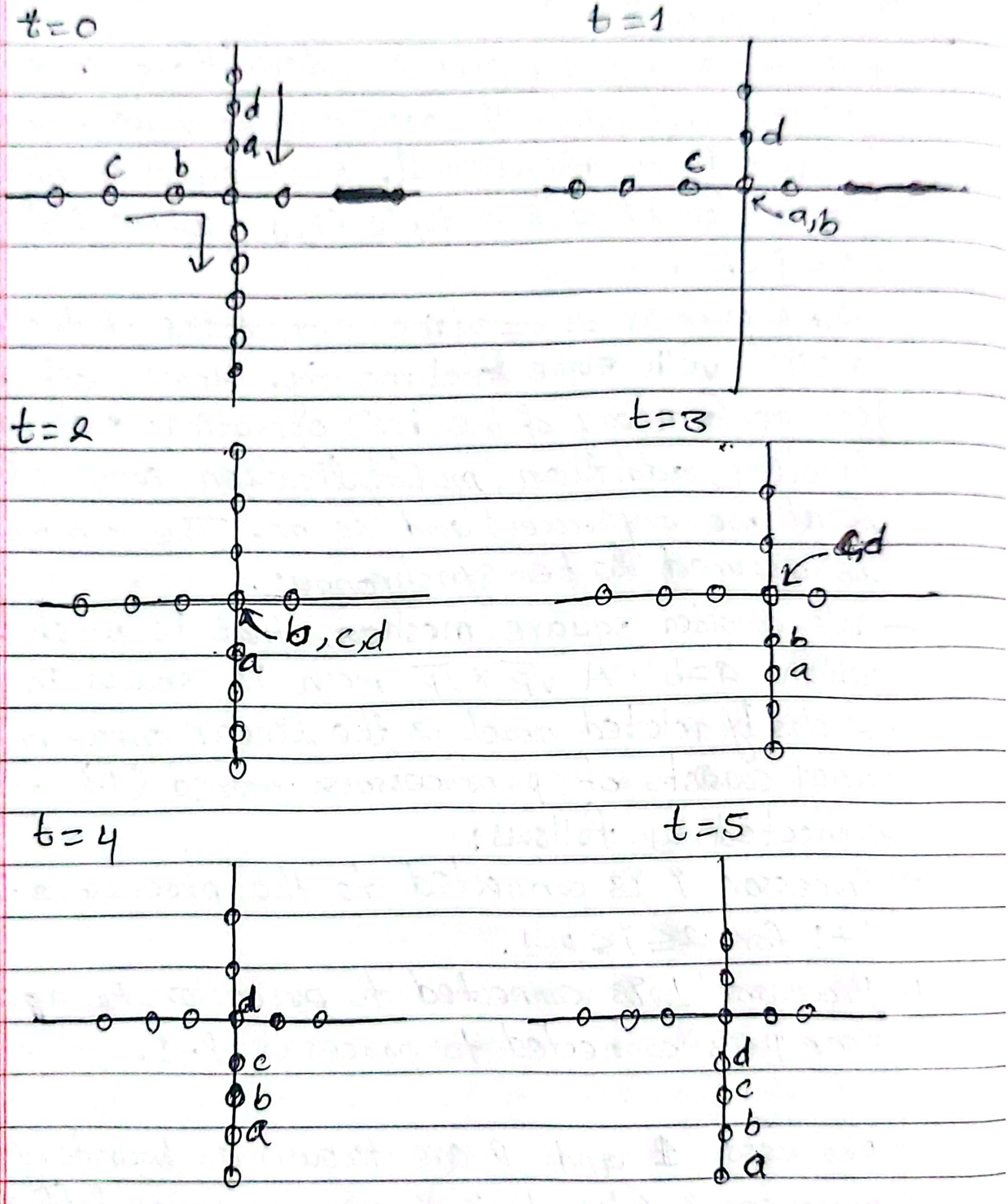


fig (b) linear array

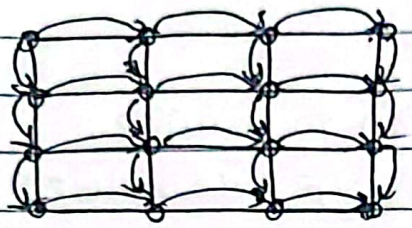
Consider a packets a, b, c and d their final destination are shown in figure below:



since it needs p-1 steps to complete the linear array the worst case of linear array is O(p-1).

(12 possible changes)

⇒ Broadcast in mesh :-



- In the case of a $\sqrt{p} \times \sqrt{p}$ mesh broadcasting can be done in two ~~bi-direction~~ phases.

1) If (i, j) is the processor of message origin in ~~that~~ phase 1, m could be broadcast to all the processors in row j.

2) In phase 2, broadcasting of m is done in each column

- This algorithm takes $\leq 2(\sqrt{p}-1$ steps) which is expressed as :
The broadcasting on a p processor linear array can be completed in p steps or less on a $\sqrt{p} \times \sqrt{p}$ mesh, the same can be performed in $\leq 2(\sqrt{p}-1) = \Theta(\sqrt{p})$ time.

Broadcasting can be done in the following ways:

a) Broadcasting without constraints (Redundancy).
b) Broadcasting with constraints (No redundancy).

⇒ Prefix computation in mesh :-

- consider a $\sqrt{p} \times \sqrt{p}$ mesh. The problem of computing prefix sums on the mesh can be reduced to three phases in each of which computation is local to the individual rows and columns. This algorithm assumes the row major indexing scheme.

The prefix computation on mesh can be done in following step:

step-1: prefix computation row-wise.
Row i (for $i = 1, 2, \ldots \sqrt{p}$) computes the prefix of its p elements.

step-2: prefix computation column wise,
Last column only:
only $\sqrt{p}$ column computes the prefix of sum computed in step 1.

step-3: shift 1 down to column wise last column only & broadcast in its respective row (column-wise) & perform prefix computation.
shifting $\Rightarrow (i, \sqrt{p})$ shifting $(i+1, \sqrt{p})$
Broadcast: Broadcast (i, p) in row $i+1$ (for $i = 1, 2, \ldots \sqrt{p}-1$)
Node j in row i updates final results.

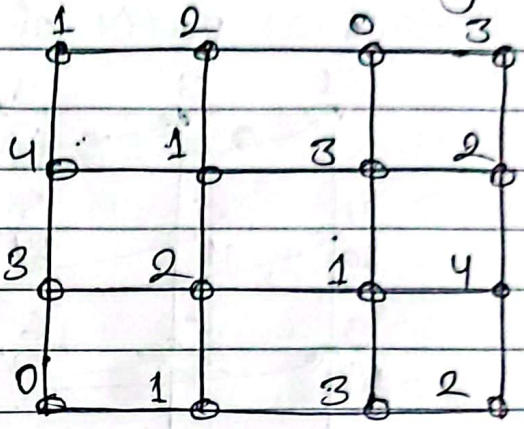
or simply

step 1: Each rows computes its prefix
step 2: only last column computes its prefix
step 3: shift the prefix of last column one down
step 4: for second row to last row broadcast the prefixes to each row and add with the individual prefix of step 3.

The worst case complexity analysis for prefix computation is $O(2\sqrt{p} - 1)$.

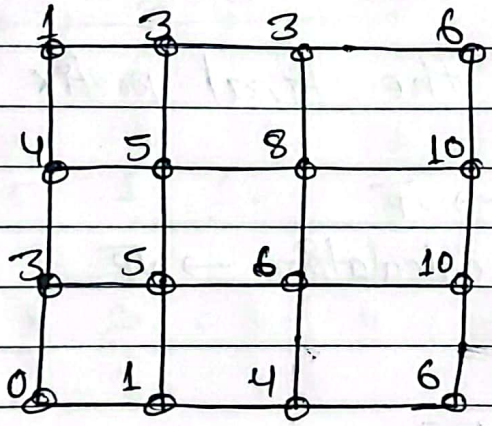
consider the data on the 3×3 mesh and the problem of prefix sums under the row major indexing scheme.

Given example is :

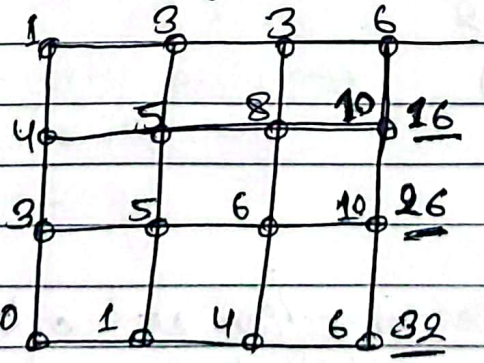


Sol

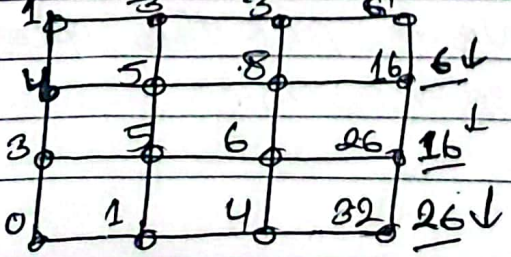
Step-1 :- prefix compute each ~~column~~ rows :



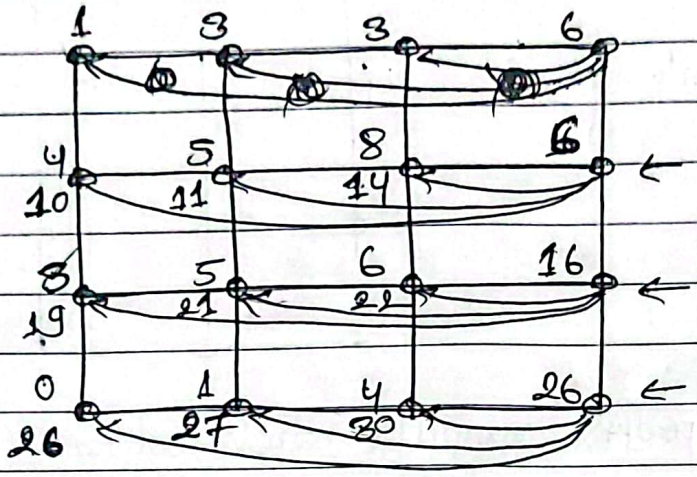
step-2 :- Only last column computes its prefix.



step-3 :- shift the prefix of last column one down.



Step-4 :- For second row to last row broadcast the prefixes to each row and add with the individual prefix of step 3.



Hence this is the final prefix computation.

prefix calculation → $\sqrt{p}$
last column prefix calculation → $\sqrt{p}$
shifting → 1
updating → 1
Broadcasting → $\sqrt{p}$

$3\sqrt{p} + 2$
$= \Theta(\sqrt{p})$ #

⇒ Data Concentration :-
- In a p-processor interconnection network. Let there be $d < p$ data items distributed arbitrarily with atmost one data item per processor.
- Problem is to move or pack the given elements into the first d processors.

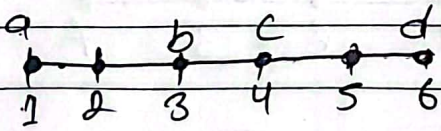
⇒ Linear Array :-

Let L be a p-processor linear array with d data items.

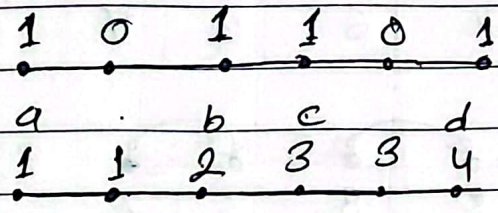
Assumptions :-

- If processor i has a data item, then set $x_i = 1$ otherwise $x_i = 0$, Let prefixes of the sequences $x_1, x_2, \dots x_n$ be $y_1, y_2, \dots y_n$
- If processor i has data item, then the ~~destination~~ destination of this item is y_i, once calculation is over, there are routed.

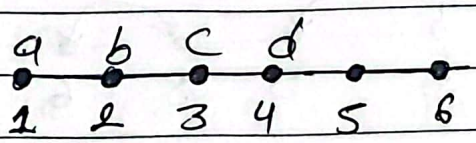
eg



$\therefore d < p$ satisfies.



Routing :-



Complexity Analysis :-

prefix calculation :- p

Routing :- p

2p

$\therefore O(p)$ //

⇒ Sorting :-

- Given a sequence of n-keys, the problem of sorting is to rearrange the sequence is either ascending or descending order.

⇒ Linear Array :-

For (i = 1; i <= p ; i++)

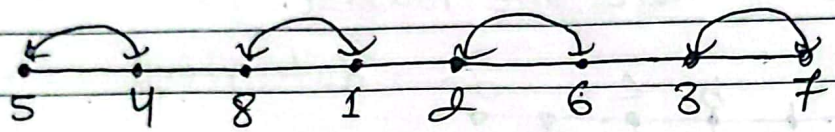
If i is odd, compare and exchange keys at processor 2i - 1 and 2i

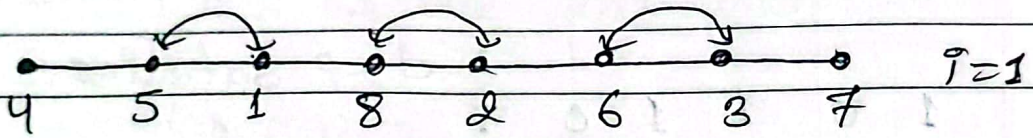
else

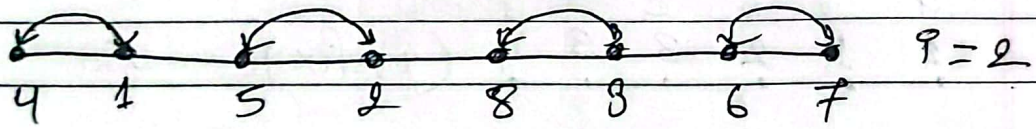
compare and exchange keys at processor 2i & 2i + 1.

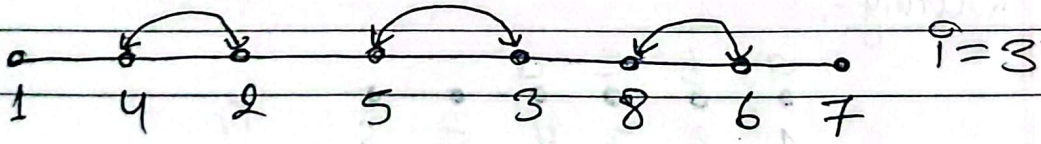
This algorithm is also known as odd-even transposition.

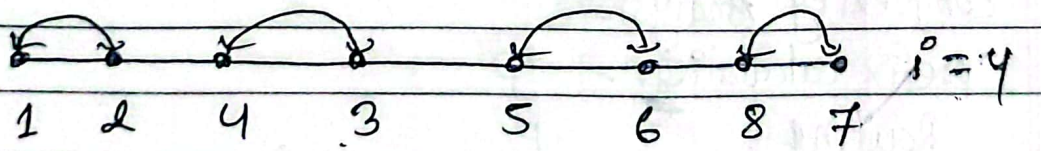
Example :-

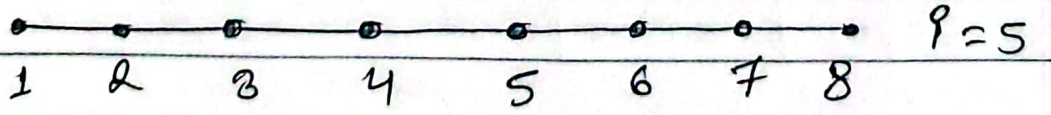












complexity : O(p) as in every execution of 'for loop' the sorting takes 1 unit of time.

⇒ Sorting on mesh :-

Algorithm

```
for (i = 1; i ≤ log p + 1 ; i++)
{
    If i as even
    sort the columns (top to bottom)
     else
        sort the rows (alternative order)
}
```

Example :-

$i = 1$

15	12	8	32	
7	13	6	17	
2	16	19	25	
18	11	5	3	

→

8	12	15	32	Asc
17	13	7	6	Desc
2	16	19	25	Asc
18	11	5	3	Desc

$i = 2$ (col Asc)

2	11	5	3
8	12	7	6
17	13	15	25
18	16	19	32

→

$i = 3$

2	3	5	11	Asc
12	8	7	6	Desc
13	15	17	25	Asc
32	19	18	16	Desc.

$i = 4$ (col Asc)

2	3	5	6
12	8	7	11
13	15	17	16
32	19	18	25

→

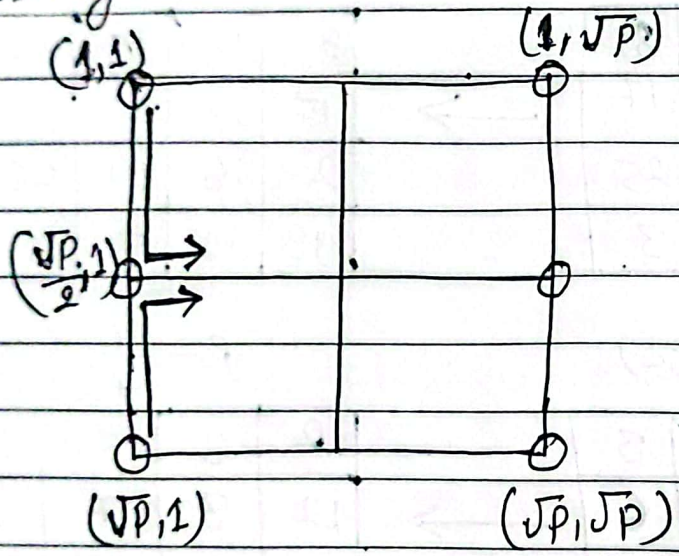
$i = 5$

2	3.	5.	6.	Asc
12	11	8.	7	Desc
13	15	16	17	Asc
32	25	19	18	Desc.

Complexity : $O(\sqrt{p} \log p)$ #

⇒ A Greedy Algorithm for PPR on a Mesh:-

- For the PPR (Parallel Prefix Reduction) problem on a $\sqrt{p} \times \sqrt{p}$ mesh, we see that if a packet at processor $(1,1)$ has $(\sqrt{p}, \sqrt{p})$ as its destination, then it has to travel a distance of $2(\sqrt{p}-1)$. Hence $2(\sqrt{p}-1)$ is a lower bound on the worst-case routing time of any packet routing algorithm.
- It used to compute the prefix sum of a given input array or list. this is one approach to solve this problem efficiently in a parallel setting.



- It use packet routing algorithms with linear array.
- Let 'q' be an arbitary packet with (i,j) as its origin and (k,l) as its destination.
- packet q use a two phase algorithm.
  - one It travels along column j to row k shortest.
  - two It traverses along row k to its correct destination using shortest path. So on repeatedly.

⇒ ~~selection~~ (n=p)

⇒ Randomized Algorithm for n = p :- (selection) :-

- Given a sequence of n keys and an integer i, $1 \le i \le n$, the problem of selection is to find the ith smallest key from the sequence.
- If $X = k_1, k_2, \ldots, k_n$ is the input, the algorithm chooses a random sample (call it S) from X and identifies two element l_1 and l_2 from S. the elements choosen are such that they braeket the element to be selected with high probability and also the number of input keys that are in range $[l_1, l_2]$ is small.
- After choosing l_1 and l_2 we determine whether the element to be selected is in the range $[l_1, l_2]$. If this is the case, we proceed further and the element to be selected is the $(i - |X_1|)$ the element of X_2. If the element to be selected is not in range $[l_1, l_2]$, we start all over again.
- selection from n=p keys can be performed in $O(\sqrt{p})$ time on a $\sqrt{p} \times \sqrt{p}$ mesh.

⇒ Randomized Selection for n > p :-

- when the number of key 'n' is larger than ~~select~~ the network size 'p' or n > p.
- In a particular, assume that $n = p^c$ for some constant $c \ge 1$.

Each processor has n/p keys to begin with. The condition for the while loop is changed to (N > D). (~~con~~ where D is constant).

In step 1 a processor includes each of

Date
Page

its keys with probability $\frac{1}{N^{1-(1/3c)}}$;
so, this step now takes time n/p.
- The number of keys in the sample is $O(N^{1/3c}) = \tilde{O}(\sqrt{p})$.

step 2 remains the same and still takes $O(\sqrt{p})$ time..
- since there are only $\tilde{O}(N^{1/3c})$ sample keys they can be concentrated and sorted in step 3. in time $O(\sqrt{p})$.

then each step takes time $O(n/p + \sqrt{p})$.

⇒ 4.2 :- Hypercube Algorithm :-

⇒ Hypercube :-
A hypercube of dimension d, denoted by H_d has $P = 2^d$ processors. Each processor in H_d can be labelled with d-bits.

- we define hypercube as following terms:
  - Dimension
  - Coding of processors.
  - Hamming Distance
  - Diameter.

Example

for d = 3 ; i.e. dimension = 3
no. of vertex = $2^d = 2^3 = 8$
no. of processors = 8
coding of processors = 000, 001, 011, 010, 110, 100, 101, 111

- coding should be carried out in a such a way that the difference of code of two adjacent processors should be 1, which is called hamming distance.
- The diameter of a d-dimensional hypercube is d.
- The bisection width of a d-dimensional hypercube is $2^{(d-1)}$.
- The hypercube is highly scalable architecture. Two d-dimensional hypercubes can be easily combined to form a d+1 -dimensional hypercube

⇒ Characteristics :-

- Each processor number should alter by 1 bit only.
- Degree of H_d = No of interconnected processor of reference processor.
- Calculate by hamming distance with no of bit position change.

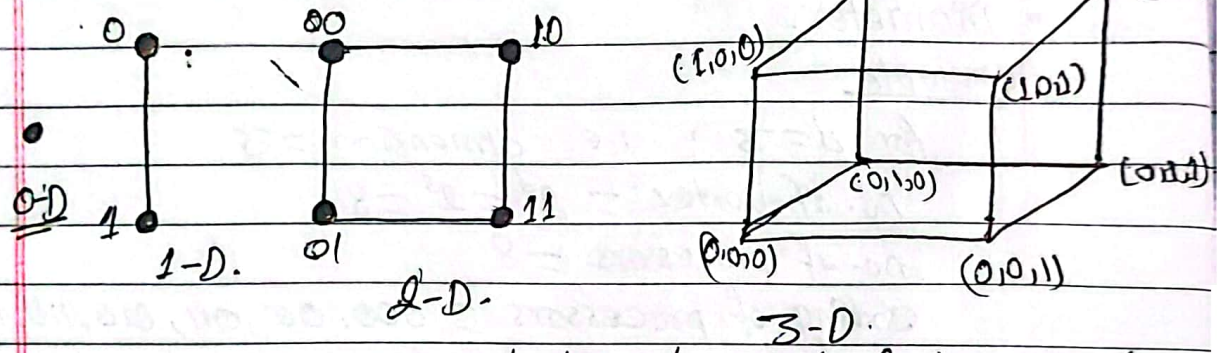
⇒ Variables :- ~~Hypercube~~ :

- Sequential – at one unit of time a processor can communicate with only one of its neighbour.
- Parallel – at one unit of time a processor can communicate with all of its neighbour.

⇒ Features :-

- The dimeter of hypercube is log p.
- A hypercube can be generated by combining different hypercubes.

$Hd = Hd_1 \times Hd_2$

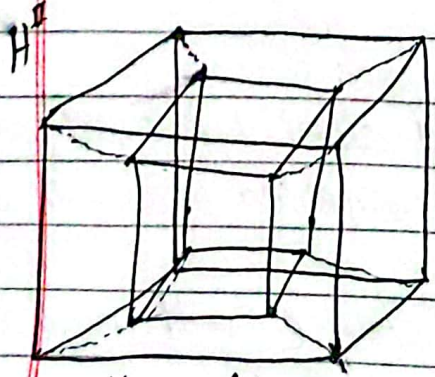


Eg computational model of hypercube.

Let's take 0,0,0

0, 0, 0	0, 0, 0	0, 0, 0
0, 0, 1	0, 1, 0	1, 0, 1

only one bit changed then we can see that all points are hamming distance a part from p processor.



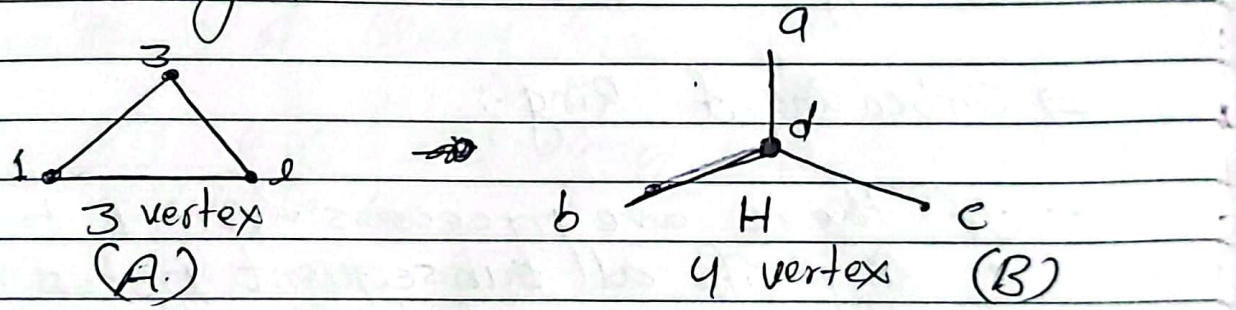
~~prefix (0)~~ ~~prefix (1)~~

0'

The hypercube has several variants like butterfly, ~~Shuf~~ shuffle-exchange network and cube-connected cycle.

prefix (0)		prefix (1)	
0	0 0 0	1	0 0 0
0	0 0 1	1	0 0 1
0	0 1 0	1	0 1 0
0	0 1 1	1	0 1 1
0	1 0 0	1	1 0 0
0	1 0 1	1	1 0 1
0	1 1 0	1	1 1 0
0	1 1 1	1	1 1 1

⇒ Embedding of Network :-



1 → b , 2 → c , 3 → a

1 − 2 → bd − dc

1 − 3 → bd − da

1) Expansion = $\frac{|V_2|}{|V_1|} = \frac{4}{3} = 1.33$

2) Dilation :- Longest path value or max length of path that any link of G is mapped.
⇒ Dilation = 2

3) Congestion :- maximum number of times any node of 'B') involves while mapping path of A with 'B') = 2.
⇒ congestion of bd = 2.

∴ Changing one network structure into another Network (N) structure.

- changing Network (M) structure to Network (N) structure.
- perform calculations on network structure (N).
- Result obtained is mapped to the original network structure (M).
- It helps in choosing the best alternative structures.
- The algorithm that has been analysed on one structure can be used on another structure that is transformable without testing.

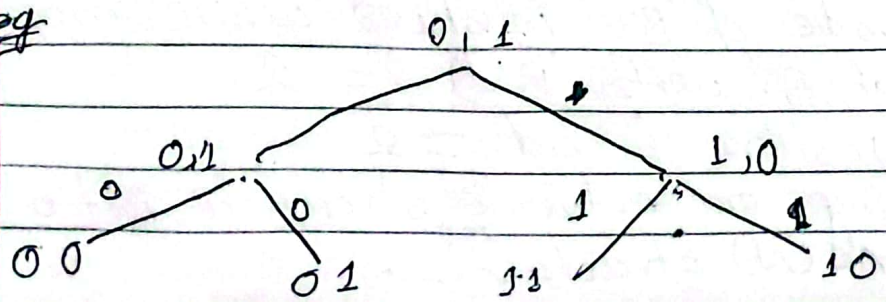
⇒ Embedding of Ring :-

- If there are processors from 1 to p, its a ring. if all subsequent processors are linked and 1 is linked with p. the transformation is carried out with the help of gray code.
- The graycode of order k, denoted by G_k defines an ordering among all the k bit binary numbers.
- G_k can be defined recursively as

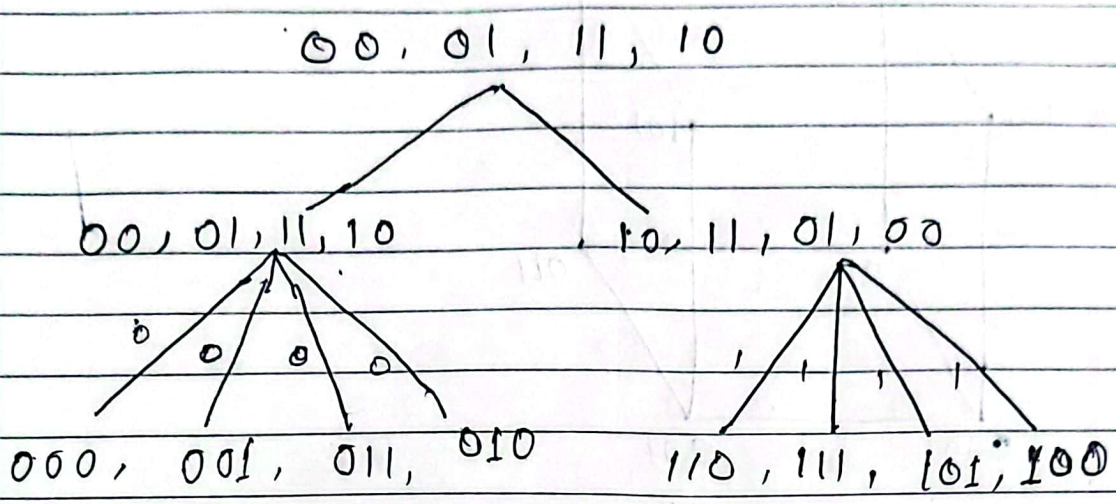
1) G_1 is 0 1
2) G_k is $0[G_{k-1}]$, $1[G_{k-1}]^r$

$[G_{k-1}]^r$ is reverse.

eg

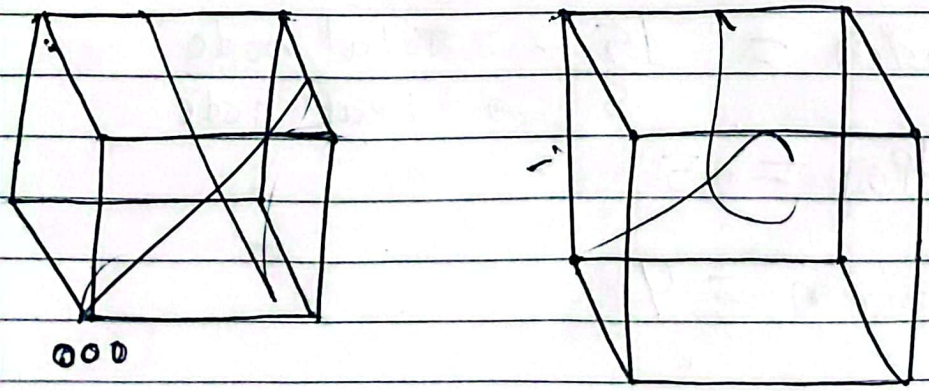


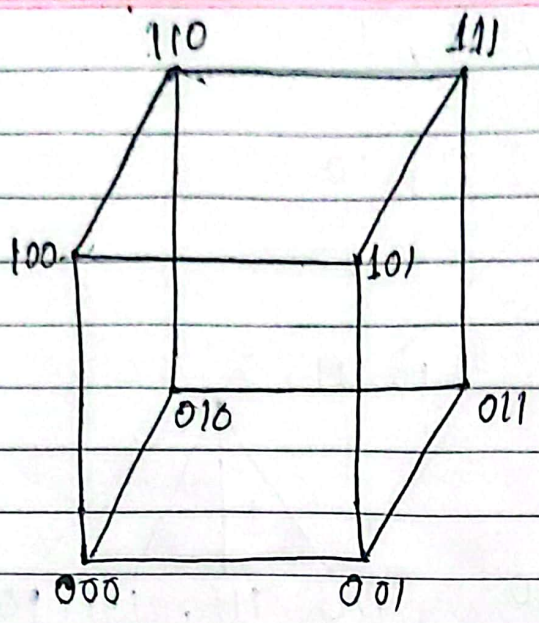
Example :- 00, 01, 11, 10



⇒ Embidding of Binary Tree :-

- A p-leaf ($P = 2^d$) binary tree T can be embidded into H_d.
- more than one processor of T have to be mapped into same processor of H_d.
- If tree leaves are 0, 1, 2, ..., P-1 then leaf is mapped to i^{th} processor of H_d.
- Each processor of T is mapped to the same processor of H_d as its leftmost descendant leaf.





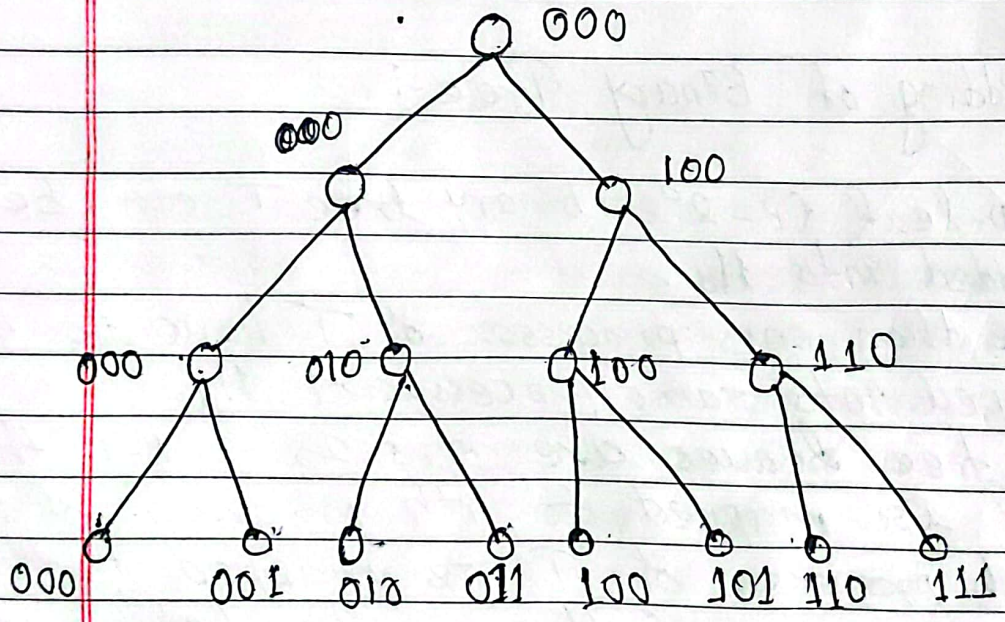


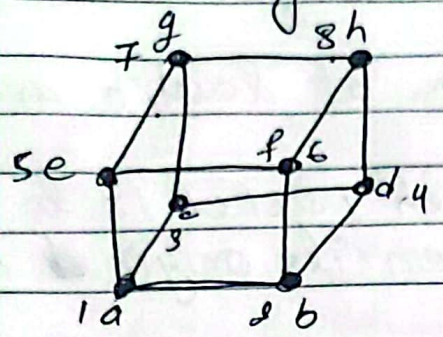
Fig Embedding of hypercube.

Expansion = $\frac{15}{8}$ → Total node / Leaf node.

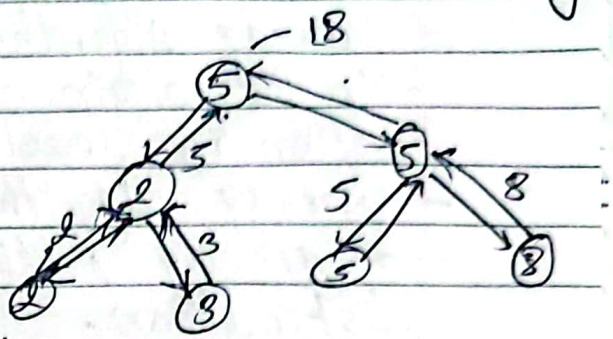
Dilation = 5

Congestion = 4

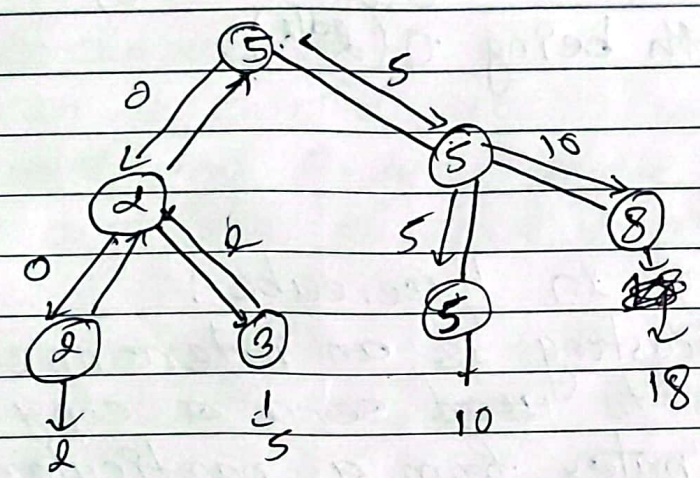
⇒ Embedding of Hypercube to tree Backtracking



⇒

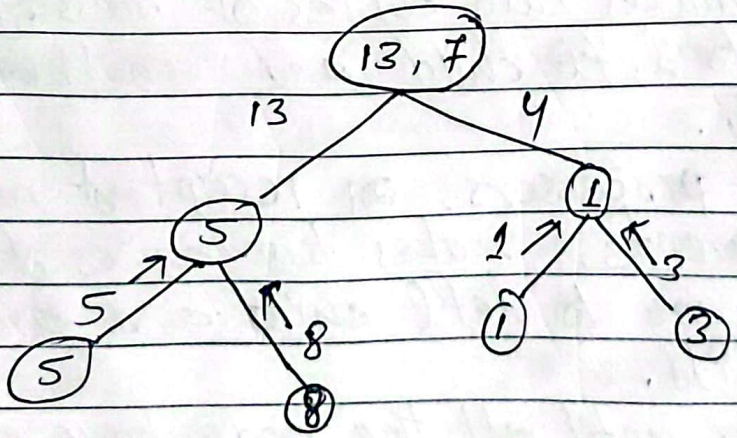


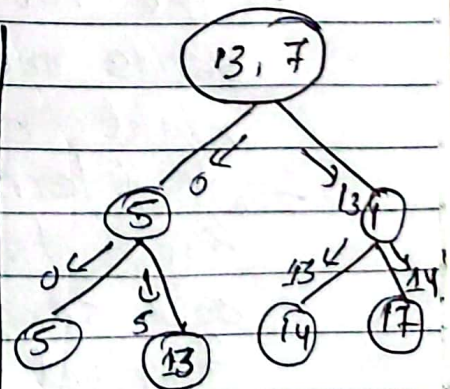
Forward pass



∴ Complexity : $O(\log n) = \log 2 \cdot \#$

⇒ perform prefix computation of : 5, 8, 1, 3 on full binary tree representation.





complexity : $O(2d)$
$\Rightarrow O(d)$ #

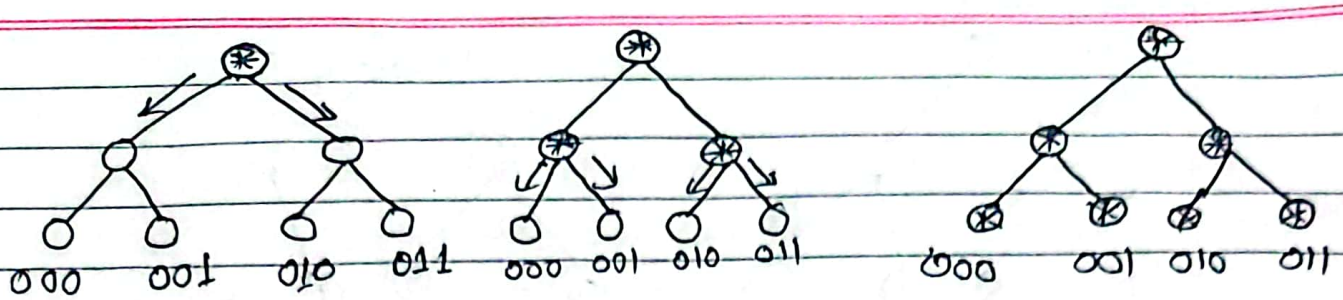
⇒ PPR Routing -:

⇒ Greedy Algorithm -:

- In B_d, origin of packet is at level 0 and destination is level d.
- Greedy algorithm for each packet is to choose a greedy path between its origin and distination.
- Distance travelled by any packet is d. Algorithm runs in $O(d^2)$ time, the average queue length being $O(2^{d/2})$.

⇒ Broadcasting in Hypercube -:

- The broadcasting is an interconnection networks. It is to send a copy of message that originates from a particular processor to a subset of other processors. It is widely used in the design of several algorithm.
- To perform broadcasting on H_d, we employ the binary tree embedding.
- Assume a message M, at root of the tree is to be broadcast.
- The root makes two copies of message and sends one to left child and another to right child.
- On internal processors, on receipt of message from its parents, makes two copies of message and sends one to left child and another to right child.
- This proceed until all the node have a copy of M.



Step-1 Step-2

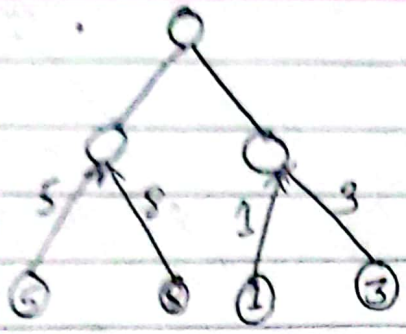
Fig: Broadcasting in hypercube.

⇒ Prefix computation -:

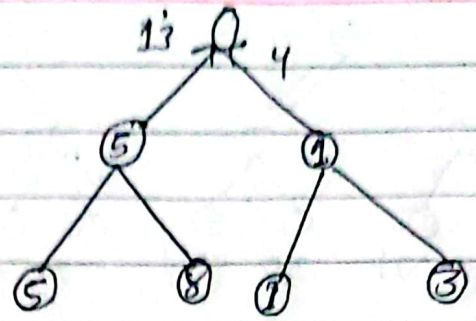
- It can be carried out in two phase. i.e. forward and Reverse phase. At each step only one level of tree is active.

Forward phase -: The leaves start by sending their data up to their parents. Each internal processor on receipt of two items (y is left child and z is right child) computes $w = y \oplus z$, stores a copy of y and w and send w to its parent. At the end of d steps, each processor in the tree has stored in its memory the sum of all data items in the subtree rooted at this processor. The root sum of all elements in tree.

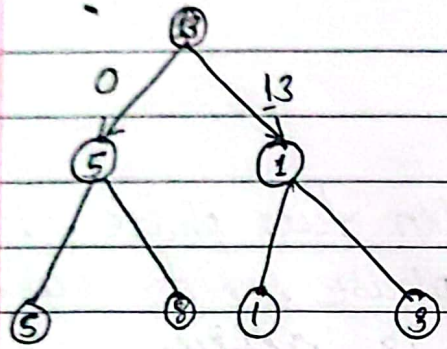
Reverse Phase -: The root starts by sending zero to its left child and y to its right child. each internal processor on receipt of a datum (say q) from its parent q to its left child and q+y to its right child. When the left gets a datum q from its parent, it computes q+x and stores it as final result.



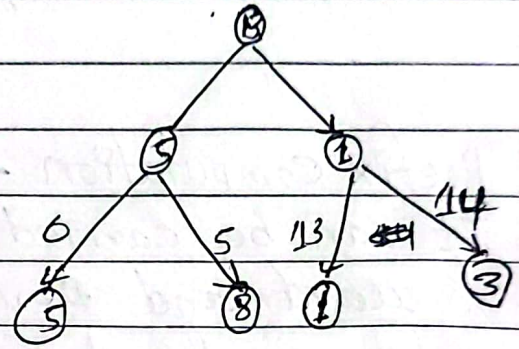
(a)



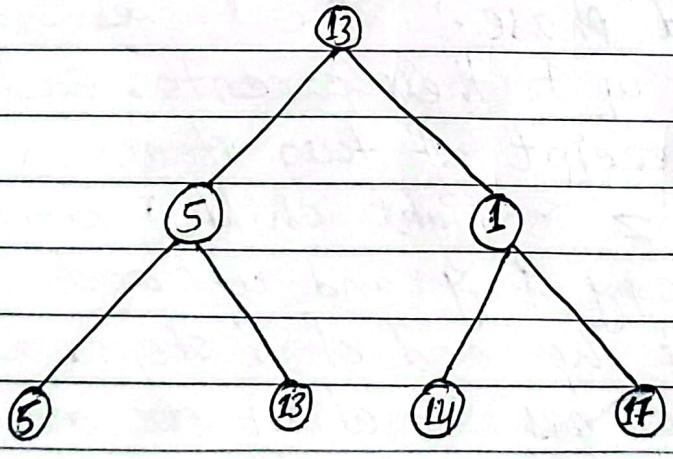
(b)



(c)



(d)



(e)

Fig prefix computation on a binary tree.

- prefix computation on 2^d - leaf binary tree ~~are k<p data item~~ as well as Hd can be done in O(d) steps.

⇒ Data concentration -:

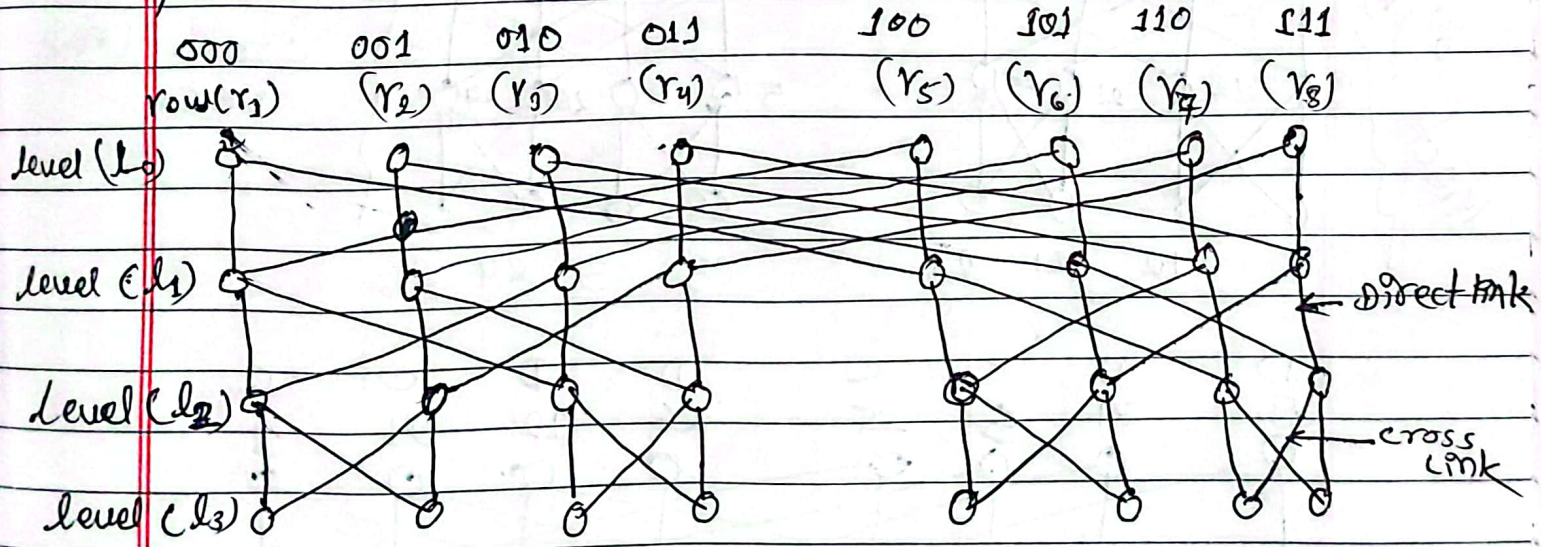
- On H_d, there are $k < p$ data items distributed arbitrarly with atmost one data per processor. The problem of data concentration is to move the data into the processor 0, ----k-1 of H_1.

I. prefix computation by using tree (binary).

II. Route by using butterfly network.

Imp ⇒ Butterfly Network (B_d) -:

B_d is also one of the embedding technique to solve problems of sorting data concentration, prefix computation, selection etc. In a massive parallel connected Network.



1) vertical line
2) cross link
3) $1^{st} \rightarrow 2^{nd}$, $3^{rd} \rightarrow 1^{st}$ cross

processor (p) = 32 $\longrightarrow$ $(d+1)2^d$

Total links = $d \cdot 2^{d+1}$

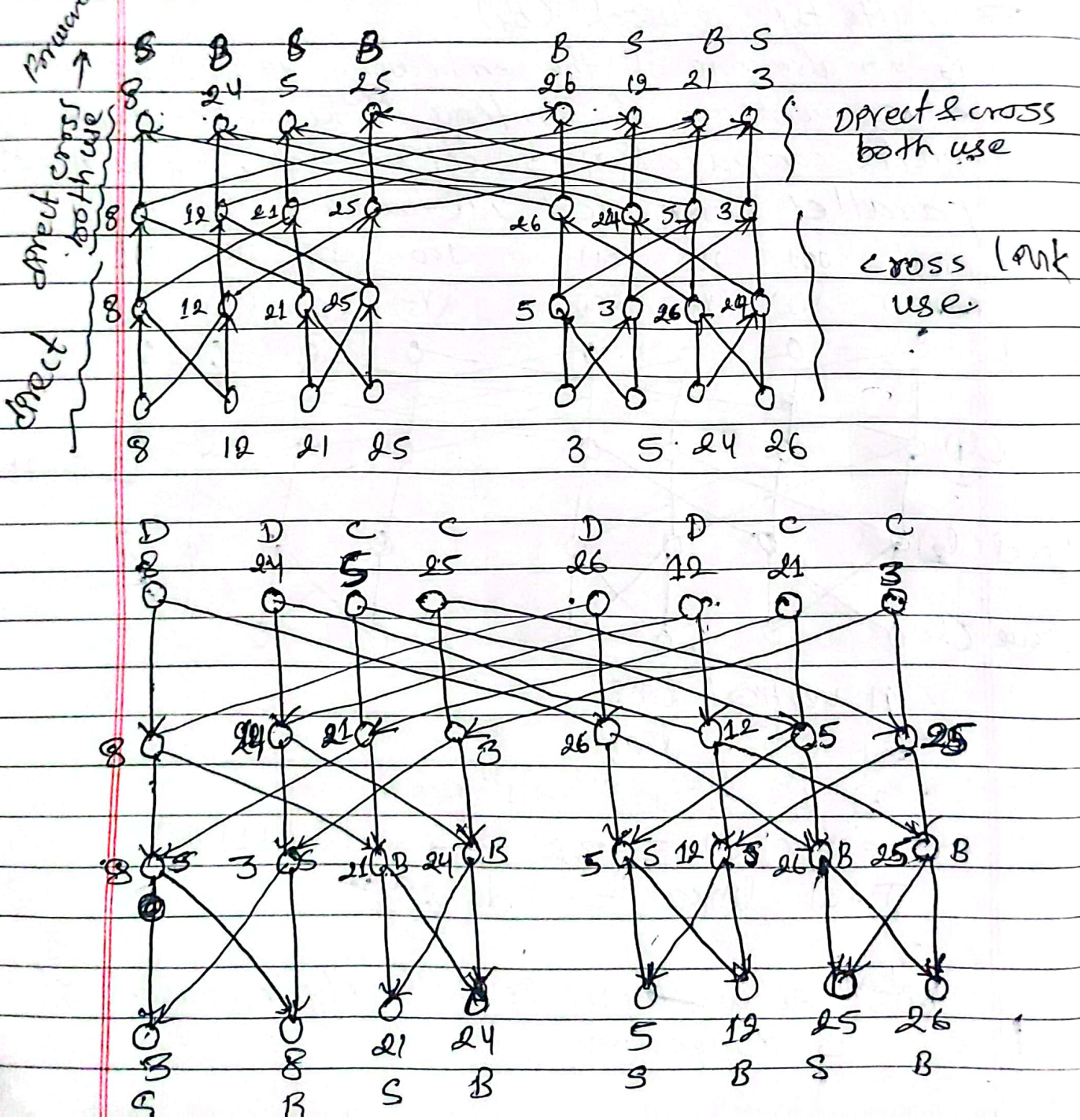
Fig Above figure is Butterfly network of 3 dimension.

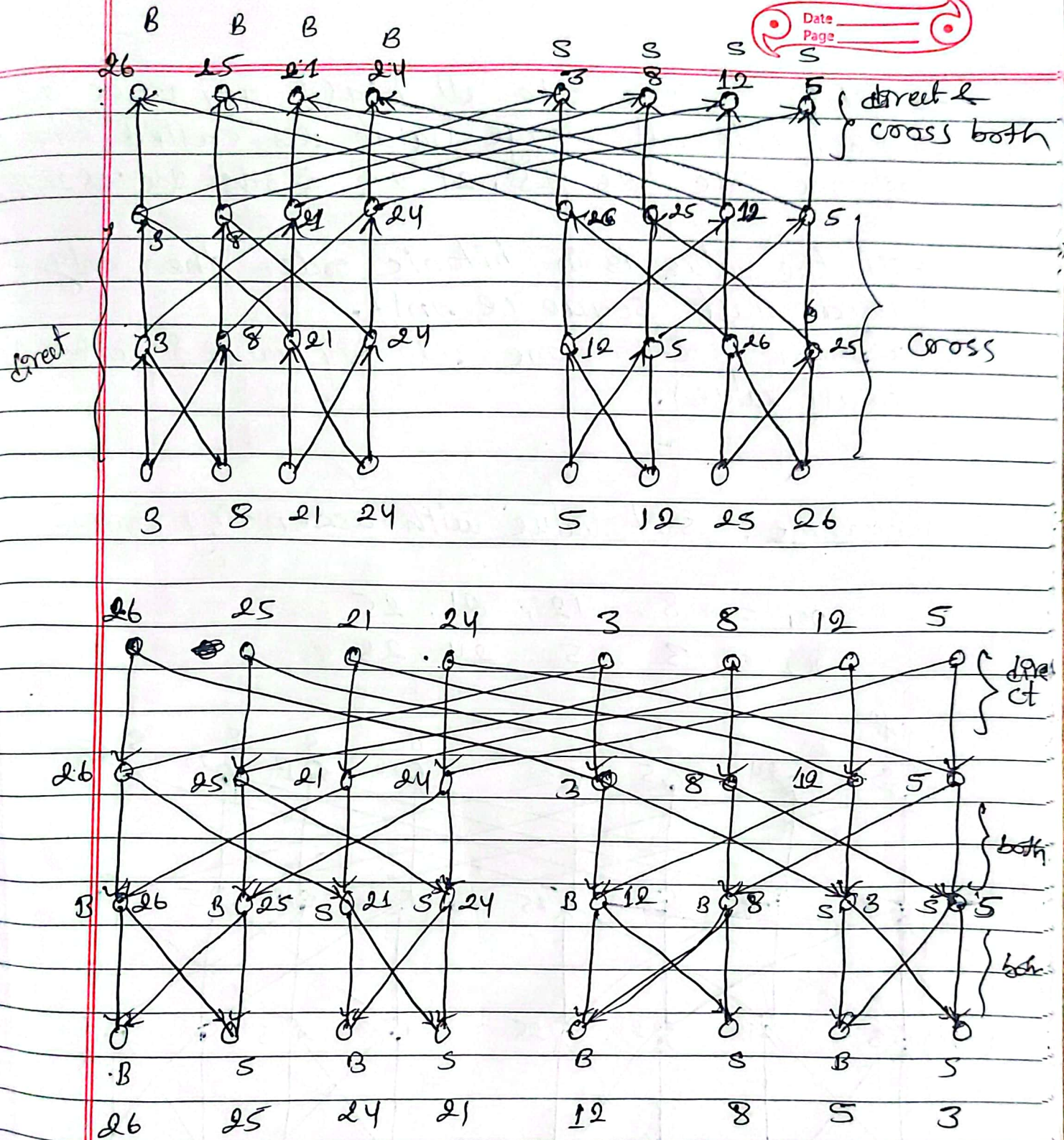
⇒ Odd-even merge sort (Butterfly network)

- It consists of two phases.
- I-phase: separate odd and even into [X_1 & X_2 given sequence to O_1; E_1, O_2, E_2]
- II-phase: Recursively merge O_1 with O_2 and E_1 with E_2.

Example :- x_1 = 8 12 21 25

x_2 = 3 5 24 26





∴ sorted are TS 26, 25, 24, 21, 12, 8, 5, 3.

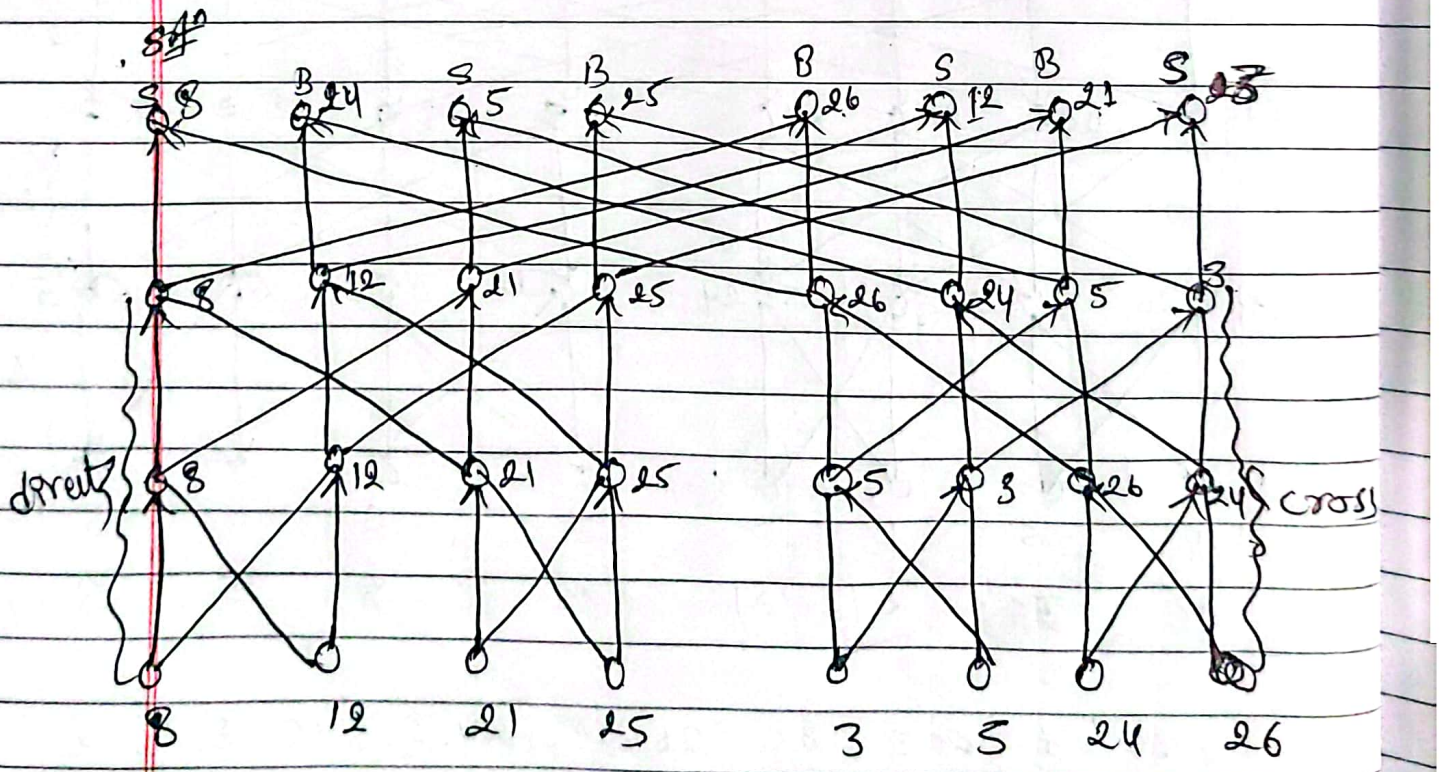
⇒ Sorted order data given example trace only last two steps. #

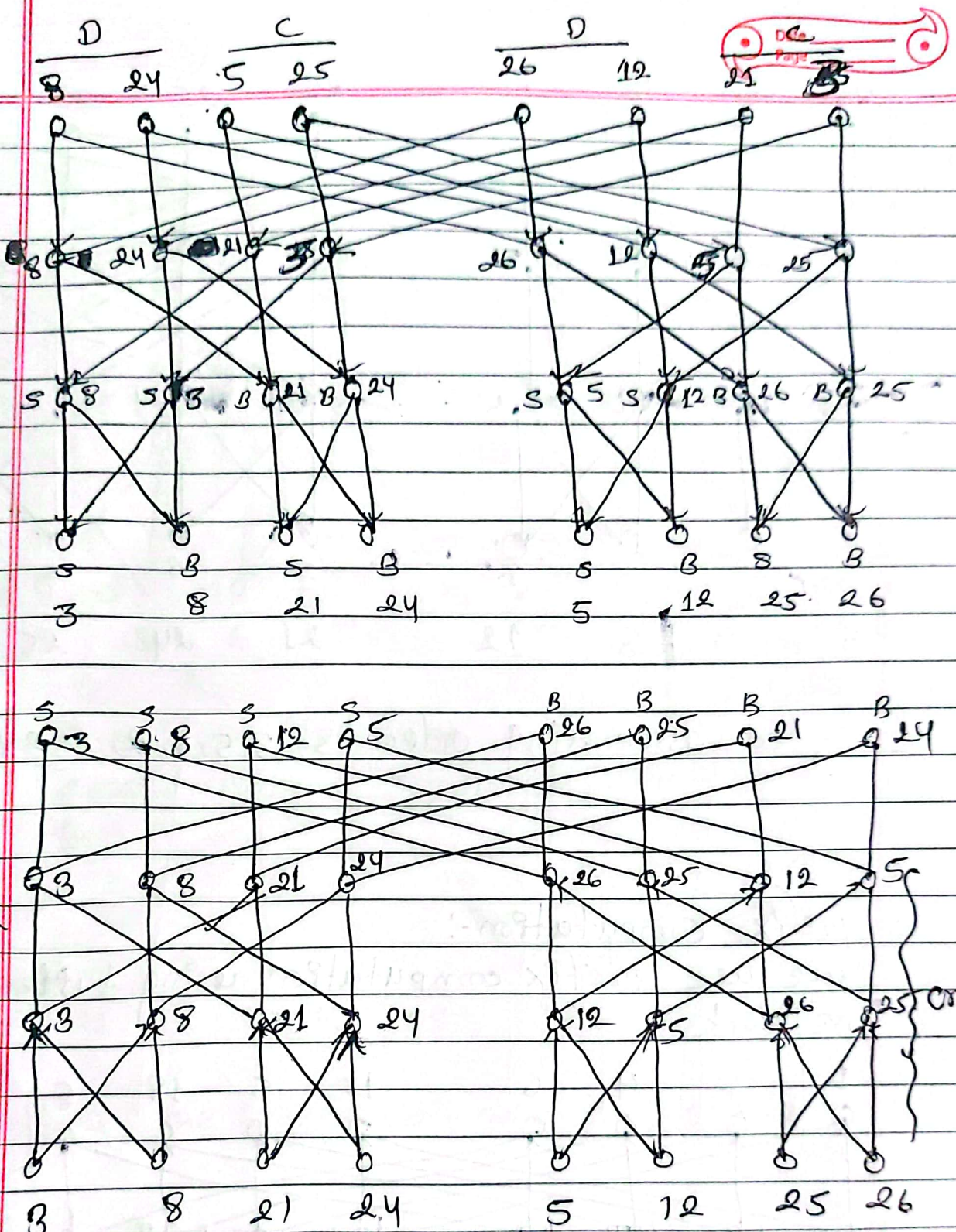
⇒ Bitonic → one side all big (large) value and other side all large value is called bitonic
eg one side 26, 25, 21, 24 & other side 3, 8, 12, 5

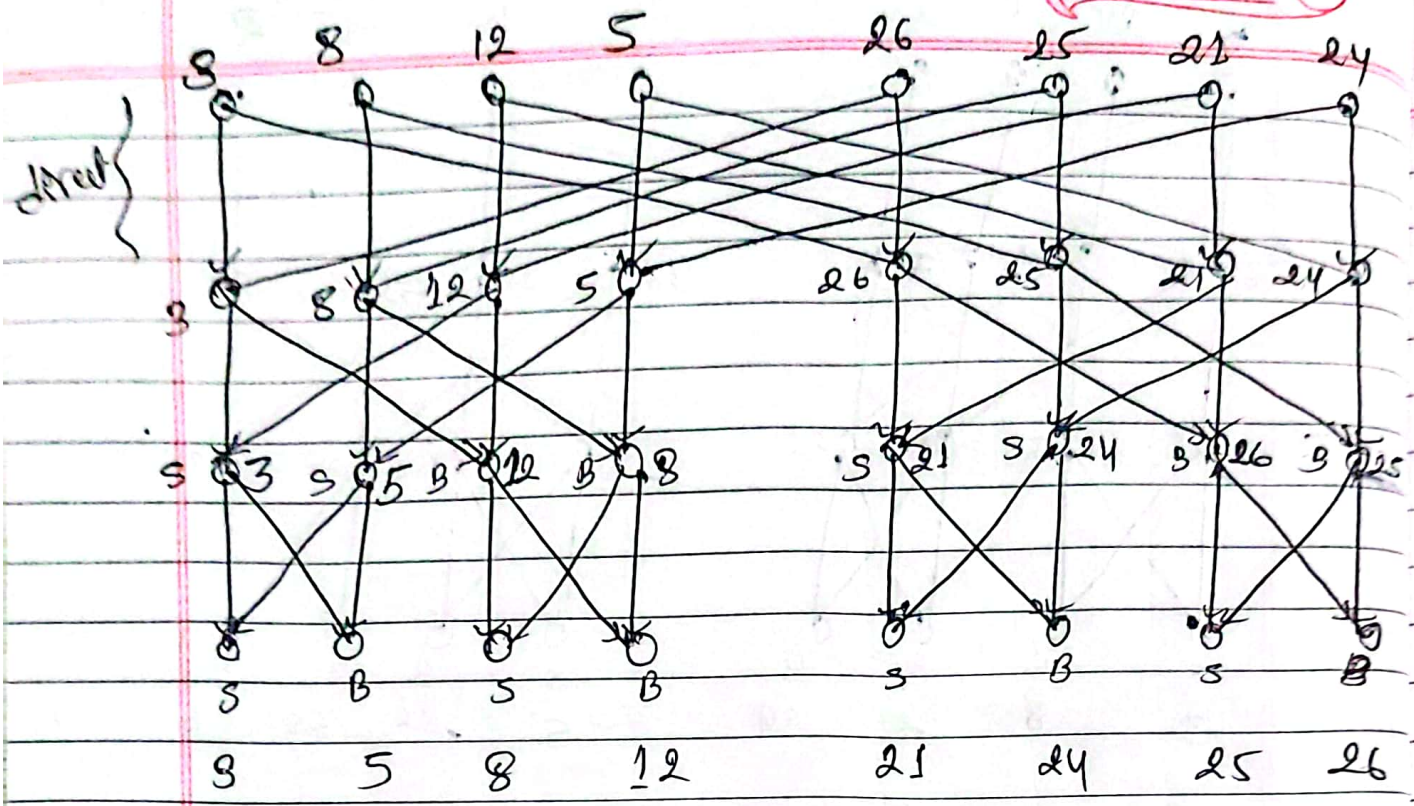
- If the value is in bitonic order then only used last sequence only.
*(Both section one side big value & other side small value).

Example: Sort value with ascending order.

x_1 = 8, 12, 21, 25
x_2 = 3, 5, 24, 26

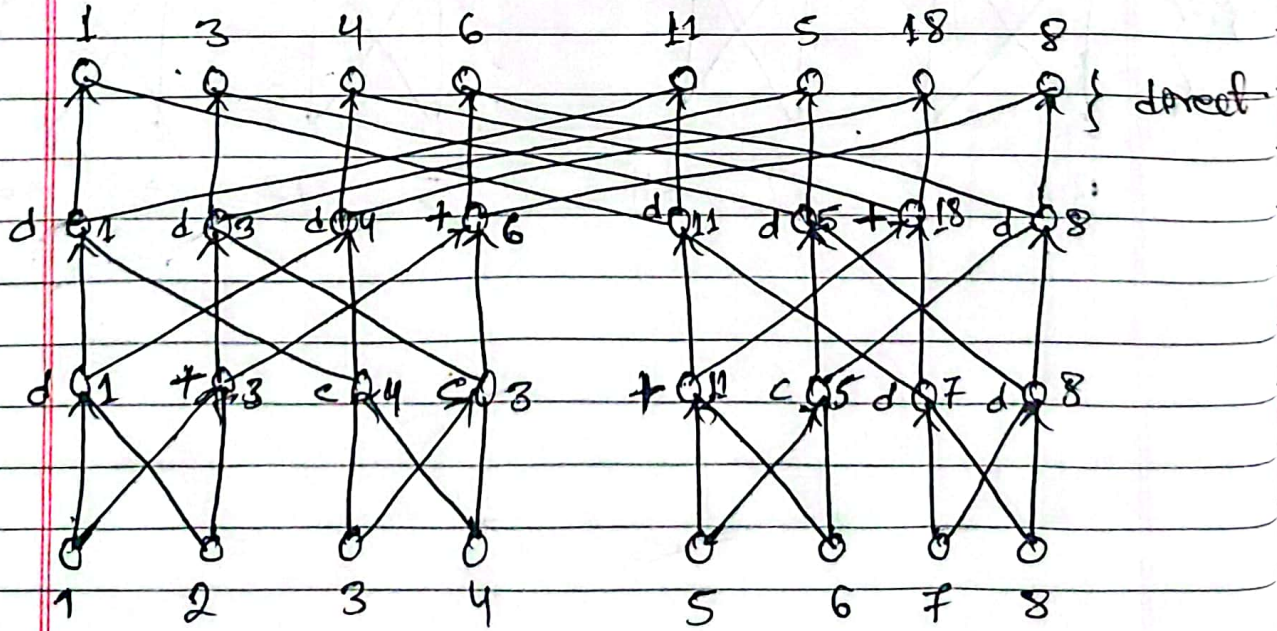


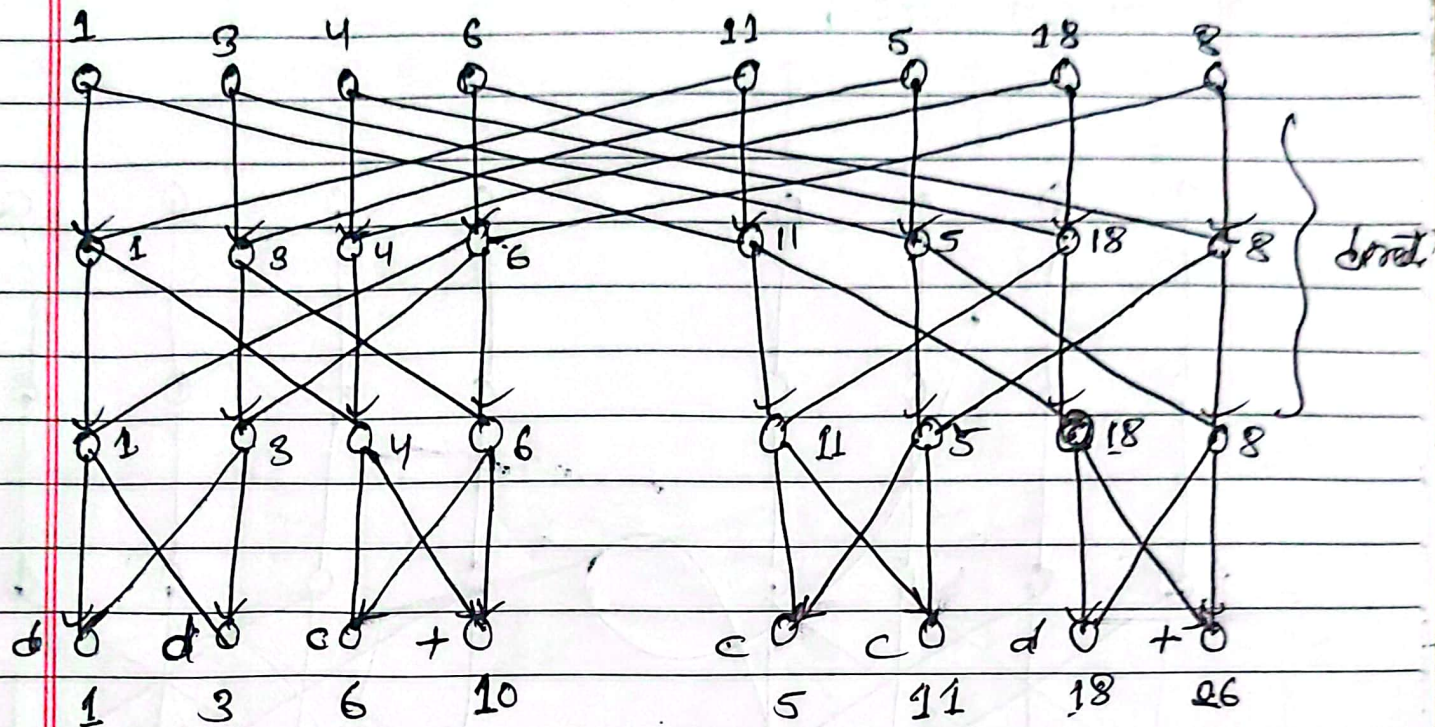

D
C
D
8 24 5 25
26 12 21 13
8 24 21 3
26 12 5 25
S 8 S B B 21 B 24
S S S 12 B 26 B 25
S B S B
3 8 21 24
S B B B
5 12 25 26
S S S S
3 8 12 5
B B B B
26 25 21 24
3 8 21 24
26 25 12 5
3 8 21 24
12 5 26 25
Direct
Cross
3 8 21 24
5 12 25 26



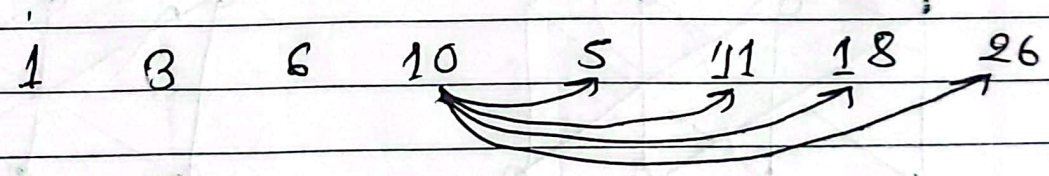
∴ Sorted asscending order is 3, 5, 8, 12, 21, 24, 25, 26

⇒ prefix computation -:
we use prefix computation using butterfly network :



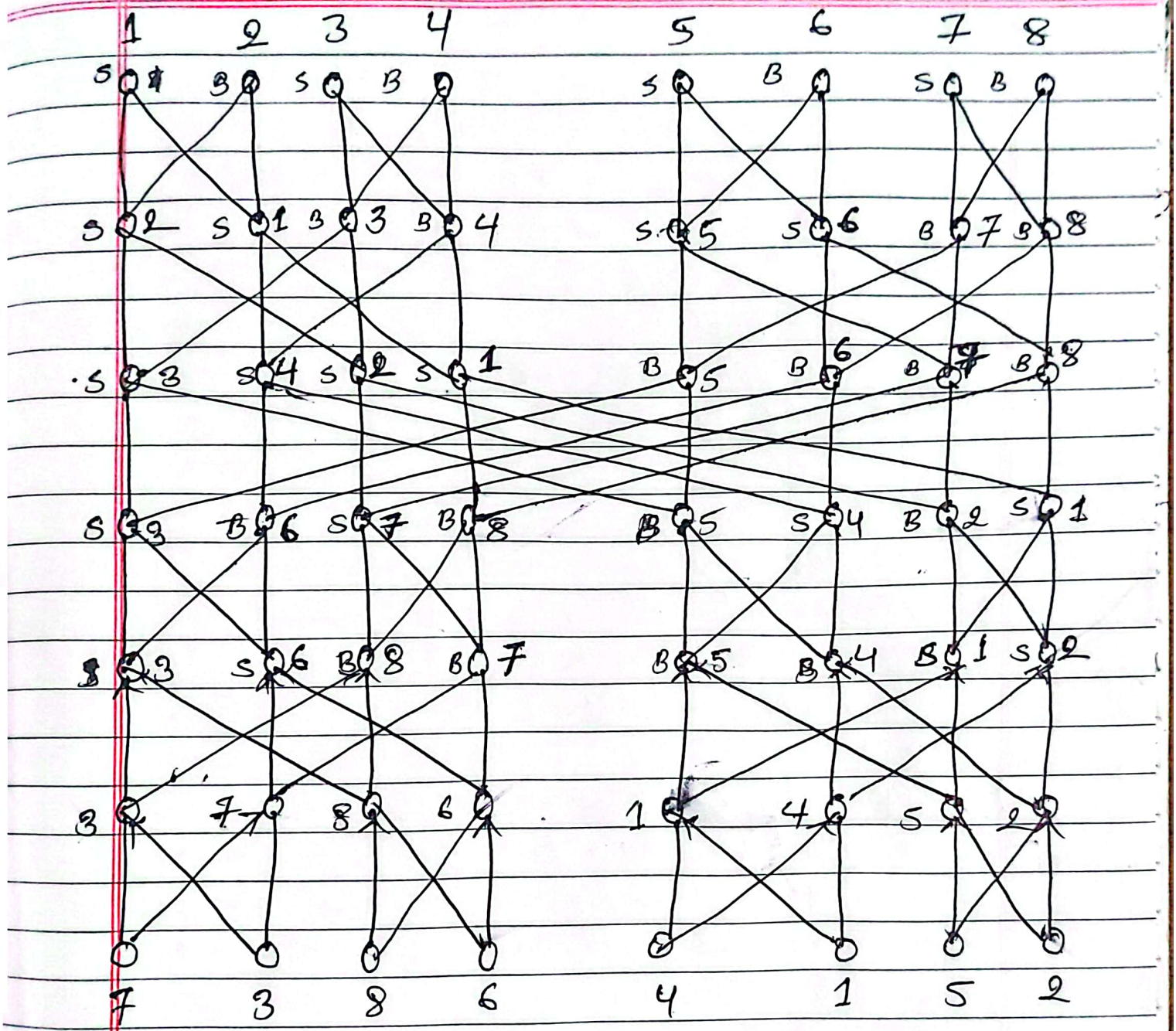


⇒ First part = 1, 3, 6, 10
Second part = 5, 11, 18, 26
Then,



⇒ 1, 3, 6, 10, 15, 21, 28, 36

Hence the prefix output ~~is~~ 1,3,6,10,15,21,28,36



⇒ selection sort output = 1, 2, 3, 4, 5, 6, 7, 8 ∎