Algorithms & Complexity:

Q.1: Explain amortized algorithm analysis technique with example of each.

Answer: Data structures typically support several different types of operations, each with its own cost (space or time cost)

- often, a data structure has one particular costly operation, but it doesnot get performed very often. That data structure must not be labeled a costly structure just because that one operation, that is seldom performed is costly.

- Amortized analysis is a method of analylizing the costs associated with a data structure that averages the worst operations out over time.

- ~~Data~~ In general, Amortized analysis is used for algorithms where an ocasional operation is very slow (costly) but most of the other operations are faster.

- The idea here is to compute the worst case average time where it is lower than the worst case time of a particular expensive operation.

- The only requirement in amortized analysis is that the sum of the amortized complexities of all operations in any way must be greater

than or equal to their sum of actual complexities

i.e. $\sum_{i=1}^{n} \text{amortized}(i) \geq \sum_{i=1}^{n} \text{actual}(i)$

where,

n = total number of operations
amortized(i) = amortized cost of i^{th} operation
actual(i) = actual cost of i^{th} operation.

- There are three approaches to amortized analysis:
  - Aggregate method
  - Accounting method
  - Potential method

(i) Aggregate Method:

- In this approach, we determine an upper bound for a sequence of n operations, say $T(n)$, then the amortized cost of n operations is set equal to $T(n)/n$. Hence in aggregate analysis, each operation has same cost.

* Example of Aggregate Analysis: Modified Stack

- stacks are linear data structure with 2 operations:
  PUSH(S,x): push object x on stack
  POP(S) : pop object from stack.

- These both operations are constant time oper ations with time $O(1)$. So a total of n operations in any order will result $O(n)$ total time.

- Now lets introduce a new operation to the stack Multipop(S, K), that pops top K elements from the stack and it it runs out of element it will pop all the elements & stop.

- The pseudo code will look like,

```
multipop(S, K)
    while S is not empty and K > 0:
        K = K - 1
        stack.pop()
```

Analysis of n sequence of operations:

(a) One multipop() can take $O(n)$ time
n multipop() takes $O(n^2)$ time [n operation of multipop]

(b) Hoover, this is not the case, since multipop() cannot function until there's been a push to the stack because there would be nothing to pop

(c) Any sequence of multipop, push, pop can take at most $O(n)$ time, since multipop(), the only non constant operation, can only take $O(n)$ time, if there have been n, constant time push operations.

(d) In worst case there are n constant-time operations and one operation taking O(n) time.

So, using Aggregate analysis,

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$$

so, this stack has an amortized cost of O(1) per operation.

(ii) Accounting Method:

- The accounting method assigns a different charge/cost to each type of operation called the amortized cost.

- some operations can be charged more or less then they actually cost.

- If an operation's amortized cost exceeds it's actual cost, we assign the difference called credit and accumulate it.

- credit can later be used to aid other operations whose amortized cost is less than their actual cost.

- Here we start by guessing the amortized cost

and then proceed to show that it satisfies the equation $P(n) - P(0) \geq 0$.

- credit can never be negative in any sequence of operations.

- Unlike in aggregate analysis, each operation can have a different amortized cost, but the cost must always be same for a specific operation.

- In symbols, given an operation whose actual cost is 'c' we assign amortized cost '$\hat{c}$' such that for any sequence of n operations with actual costs $c_1, c_2, c_3 \ldots c_n$ and amortized costs $\hat{c}_1, \hat{c}_2, \hat{c}_3 \ldots \hat{c}_n$, we have,

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

as long as this condition is met, we know that the amortized cost provides an upper bound on the actual cost of any sequence of operations.

* Example: Multipop stack

From aggregate method we found, the actual costs of modified stack operations,

PUSH : 1
POP : 1
Multipop: $\min(|S|, k)$

Now, using accounting method, we can findout the amortized costs as.

PUSH : 2
POP : 0
Multipop: 0

The final table is;

Operation	Actual cost	Amortized cost
PUSH	1	2
POP	1	0
Multipop	Min(|S|, K)	0

Proof :
For sequence of operations PUSH → PUSH → PUSH → POP → PUSH → POP → POP → PUSH → Multipop(4), show the above amortized cost is suitable.

operation	PUSH	PUSH	PUSH	POP	PUSH	POP	POP	PUSH	Multipop(4)
Actual cost	1	1	1	1	1	1	1	1	2
Amortized	2	2	2	0	2	0	0	2	0
P(i)	1	2	3	2	3	2	1	2	0

The actual cost of multipop(4) = min(|S|, K)
= min(2, 4)
= 2

from above calculation, we can verify that the chosen amortized cost is suitable.

(iii) Potential Method:

- The potential method is similar to accounting method but instead of thinking about the analysis in terms of cost and credit, potential method develops a potential function.

- The potential function refers to the credit assigned to the entire data structure as a whole.

- The potential method starts with an initial data structure D_0. Then n operations are performed turning the initial data structure to $D_1, D_2, D_3 \cdots D_n$.
  - c_i is the actual cost associated with the i^{th} operation and D_i is the structure resulting from i^{th} operation.

- $P()$ is the potential function that maps data structure D to number $P(D)$, the potential associated with that structure.

- The amortized cost of the i^{th} operation is,

$$a_i = c_i + P(D_i) - P(D_{i-1})$$

for n operations, total amortized cost is,

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + P(D_i) - P(D_{i-1}))$$

which equals,

$$\sum_{i=1}^{n} c_i + P(D_n) - P(D_0)$$

Thus the total credit in the data structure is $P(D_n) - P(D_0)$ and $P(D_n) - P(D_0) \geq 0$.

- Any function $P()$ that satisfies above property can be used as potential function

* Example: Modified stack with Multipop().

- Let the potential of stack be defined as $P(S) = |S|$, the number of elements in the stack.

- An empty stack has zero potential and $P(S)$ is non-negative

- Then the costs of stack operations are as follows.

(a) PUSH operation increases size of S by 1 and has actual cost 1. Thus the amortized cast for PUSH is,

$$\begin{aligned} a_{PUSH} &= a_{ctual}(PUSH) + P(D_{new}) - P(D_{old}) \\ &= 1 + (|S|_{new} + 1) - (|S|)_{old} \\ &= 1 + 1 \\ &= 2 \end{aligned}$$

(b) POP decreases s by 1 and actual cost = 1, Thus amortized cost of POP is,

$$a_{pop} = \text{actual}(POP) + P(D_{new}) - P(D_{old})$$
$$= 1 + (|S_{old}| - 1) - |S_{old}|$$
$$= 1 - 1$$
$$= 0$$

(c) Multipop(s, k) decreases size of s by $\min(|s|, k)$ and actual cost = $\min\{|s|, k\}$ Thus amortized cost is,

$$a_{multipop} = \text{actual}(multipop) + (D_{new}) - (D_{old})$$
$$= \min\{|s|, k\} + (|S_{new}| - |S_{old}|)$$
$$= \text{min}\{|s|, \quad = 0$$

for $|s| = 5$, multipop (3).

$$= \min\{|s|, k\} + (|S_{new}| - |S_{old}|)$$
$$= \min\{5, 3\} + |s| \text{ after pop}(3) - |S_{old}|)$$
$$= 3 + (5 - 3) - 5$$
$$= 3 + 2 - 5$$
$$= 0$$

Hence we have,

operation	Actual cost	Amortized cost
PUSH	1	2
POP	1	0
Multipop	$\min\{|s|, k\}$	0

Q. 2: What are probabilistic and randomized algorithms? compare and contrast between Monte Carlo and Las Vegas algorithm.

Answer: A randomized algorithm is one that makes use of randomizer and some of the decisions made in the algorithm depend on the output of the randomizer.

- The idea of probabilistic (randomized) algorithms is to build an algorithm using a random element so as to gain improved performance.

- In some cases the performance is very dramatic, moving from intractable to tractable. often however, there might be loss in the reliability of results: either no result at all may be produced or an incorrect result may be returned or for numerical results an approximate answer may be produced.

- Because of the random element, different runs may produce different results and therefore the reliability of results may be improved with multiple runs

- Since a randomizer gives varying output on each different runs, a randomized algorithm can be viewed as a family of algorithm.

- for a given input, some of the algorithms in this family may run indefinitely for long periods of time (or may give incorrect answers)

- The objective in the design of a randomized algorithm is to ensure that the number of such bad algorithms in the family is very low on comparision of total nuber of algorithms.

- If for any input, we can show that $(1-\epsilon)$, (ϵ is very close to 0), fractions of algorithms run quickly, respectively and give correct answer on that input, then a random algorithm in the family will run quickly on any input with probability $\geq 1-\epsilon$, where ϵ is the error probability.

Classification of probabilistic algorithms

- The randomized algorithm can be categorized into two classes:
  (i) Las vegas algorithms
  (ii) Monte Carlo algorithms

(i) Las Vegas Algorithms:
- They produce the same (correct) output for the same input
- These algorithms never return incorrect result, but may not produce results at all on some runs.

- The execution time of Las vegas algorithm depends on the output of the randomizer.
- In some cases, the algorithm terminates very fast but in worst cases it might run very long time.
- The running time is not fixed in Las vegas algorithm. for eg. Randomized quick sort always produces a correctly sorted array as its output. However it takes $O(n \log n)$ time on average but can be as bad as $O(n^2)$ in the worst case.

(ii) Monte carlo Algorithms:
- These algorithms might produce different result from run to run for the same input
- The Monte carlo algorithms might produce incorrect result depending on the output of randomizer.
- The running time of these algorithms is fixed. typically for a fixed input, monte carlo algorithms donot show much variation in the execution time between runs.
- Most of the Monte carlo algorithms can be converted

to Las Vegas algorithm by adding a check to see if the given answer is right and if not running the Monte carlo algorithm again till the answer is right.

Q.No.3 Explain the Tree Vertex splitting solution with the Greedy based approach of its solution:

Answer:

- Given a weighted directed binary tree that can be used in distribution network such as power lines and oil. Loss occurs in the process of transmission.
- Each edge is labelled with the loss that occurs during transmission via the edge.
- The network cannot tolerate losses beyond a certain level 'δ'. A booster needs to be placed where the loss exceeds the tolernance level 'δ'.
- The booster can only be placed in the nodes of the tree. TVSP has a solution if the maximum edge weight $\leq \delta$.

Then the Tree vertex splitting problem can be defined as:

Given a network and a loss tolerance level, the tree vertex splitting problem is to determine the optimal placement of boosters

solution:

- Our objective is to minimize the number of booster operations (X)
- computation proceeds from leaves to root
- Delay for each leaf node is zero.
- Then, for each node $u \in V$,
  (a) compute the maximum delay $d(u)$ from u to any other node in its subtree
  (b) If u has a parent v such that $d(u) + w(v,u) > \delta$, split u and set $d(u) = 0$
  (c) The delay for each node v is computed from the delay for the set of its children $c(v)$,

$$d(v) = \max_{u \in c(v)} \{d(u) + w(v,u)\}$$

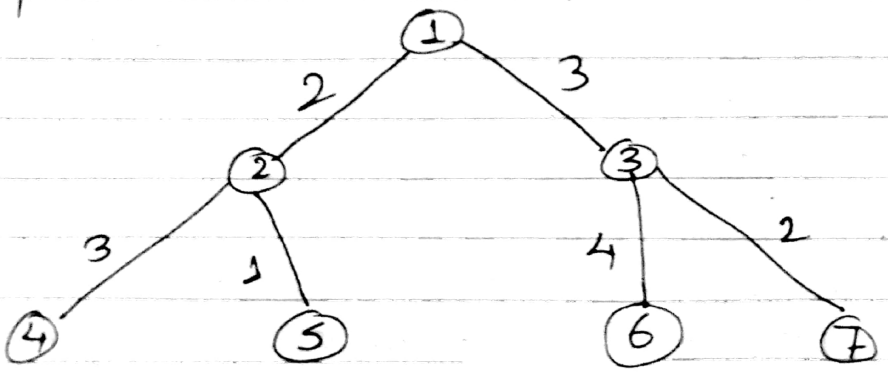
  (d) if $d(v) > \delta$, split v

Algorithm:

```
TVS (T, δ){
if (T != 0){
        d[T] = 0;
        for each child v of T do{
                    TVS (v, delto);
                    d[T] = max{d[T], d[V] + w(T,V)};
                    }

        if (T is not root) && (d[T] + w(Parent(T), T) > δ))
            {
                write (T);
                d[T] = 0
               }
}]
```

Example. For δ = 5, Perform Tree vertex splitting.



Solution:

we start at leaf nodes. the delay at leaf nodes is zero.

$d[4] = d[5] = d[6] = d[7] = 0$

Now, for node 2,

$$d[2] = \max\{d[T], d[v] + w(T,v)\}$$
$$= \max\{0, 0+1, 0+3\}$$
$$= \max\{0, 1, 3\}$$
$$= 3$$

Since 2 is not root and compute,

$d[T] + w(\text{Parent}(T), T)$

or, $3 + w(1, 2)$

or, $3 + 2$

or, $5 = \delta$, no splitting necessary

for node 3

$$d[3] = \max\{d[T], d[v] + w[T,v)\}$$
$$= \max\{0, 0+4, 0+2\}$$
$$= \max\{0, 4, 2\}$$
$$= 4$$

Since 3 is not root, we compute.

$d[T] + w(\text{Parent}(T), T)$

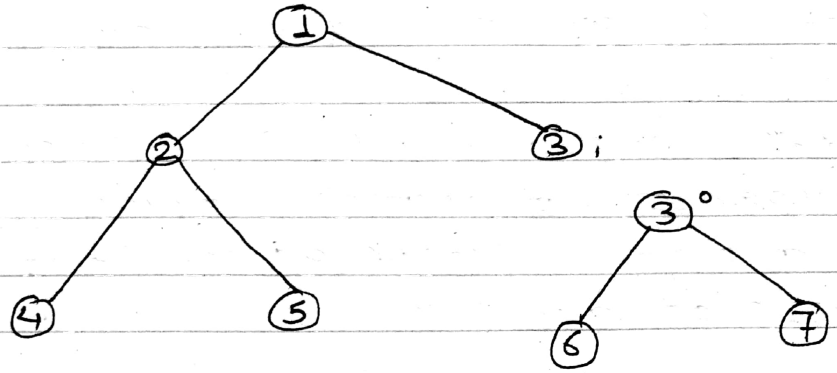
or, $4 + w(1, 3)$

or, $4 + 3$

or, $7 > (\delta = 5)$

Hence, we split the vertex 3.

Since, remaining vertex 1 is the root, we stop. Then the tree formed is,



Since, booster needs to be placed at vertex 3.

4. What is string editing problem? Discuss the approaches to solve this problem using greedy and dynamic approach.

Answer: String editing problem:

- There are two strings source & destination. The goal of the problem is to convert source to destination by applying minimum edit operations on string 'source'.

- The edit operations are:
  (a) insert a character - 1 cost
  (b) delete a character - 1 cost
  (c) Replace a character - 2 cost

solution:

- This problem can be solved optimally using buttom up dynamic programming. we will first calculate minimum operations required to transform prefix of the string source to the prefix of string destination.

- Then we use this information to calculate minimum nuber of operations required to transform a bigger prefix of source to a bigger prefix of destination.

- The time complexity of this approach will be $O(len_1 * len_2)$

Process of solution:

- Let $X_b = x_1, x_2, x_3 \ldots x_n$ and $Y = y_1, y_2 \ldots y_m$ be source & destination strings respectively.
- Then let $cost(i,j)$ be minimum cost of any edit sequence for transforming $x_1 x_2 \ldots x_i$ into $y_1, y_2 \ldots y_j$ for $(0 \leq i \leq n$ and $0 \leq j \leq m)$.
- Now we compute $cost(i,j)$ for each i & j. Then $cost(n,m)$ is the cost of an optimal edit sequence.
- Then following conditions exist.
  (i) For $i = j = 0$, $cost(i,j) = 0$, since the two sequences are identical & empty.
  (ii) if $j = 0$ and $i > 0$, we can transform x to y by sequence of deletes and $cost(i,0) = cost(i-1, 0) + D(x_i)$

(iii) if $i=0$ & $j>0$, $cost(0,j) = cost(0,j-1) + I(y_j)$

(iv) if $i \neq 0$ and $j \neq 0$ then following 3 conditions exist

(a) Transform $x_1, x_2 \ldots x_{i-1}$ into y_1 $y_2 \ldots y_j$ using minimum cost edit and then delete x_i.

$$cost(i,j) = cost(i-1,j) + D(x_i)$$

(b) Transform $x_1, x_2 \ldots x_i$ into $y_1, y_2 \ldots y_{j-1}$ using minimum cost edit sequence and then insert y_j

$$cost(i,j) = cost(i,j-1) + I(y_i)$$

(c) Transform $x_1, x_2 \ldots x_{i-1}$ into $y_1, y_2 \ldots y_{j-1}$ using a minimum cost edit sequence and change symbol x_i to y_j. The cost is.

$$cost(i-1, j-1) + c(x_i, y_j)$$

Summarizing all we have:

$$cost(i,j) = \min \begin{cases} cost(i-1,j) + 1 \\ cost(i,j-1) + 1 \\ cost(i-1,j-1) + \begin{cases} 2 & \text{if } X(i) \neq Y(j) \\ 0 & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

Example: compute the minimum cost to convert string abbcab to acbbab.

		a	b	b	c	a	b
	0	1	2	3	4	5	6
a	1	0 ↑	1	2	3	4	5
c	2	1 ↖	2	3	2	3	4
b	3	2	1 ↖	2	3	4	3
b	4	3	2	1 ←	2 ↖	3	4
a	5	4	3	2	3	2 ↖	3
b	6	5	4	3	4	3	2

LEFT → insertion

up ~~DOWN~~ → deletion

DIAG → substitution / No replacement.

Hence, the number of operations = 2.

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

Q.NO5: Explain how dynamic programming approach can be used to solve optimal BST.

- A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes.

- The keys are ordered lexicographically i.e. for each internal node all the keys in left subtree 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 BST is a BST, which has minimal expected cost of locating each node.

- Hence the optimal BST problem is of finding the BST tree structure that minimizes the cost of searching each node.

Dynamic programming solution:

- we use an auxiliary array $c[n][n]$ to store the solutions of subproblems. $c[0, n]$ will hold the final result.

- we start by filling all the diagonal values first, then we go for the values which lie on the line just above the diagonal.

- In other words we must fill all cost[i][i] values, then all cost[i][i+1] values and then all cost[i][i+2] values.

- The cost is computed as,

$$W[0,i] = sum(q_i)$$

$$c[i,j] = W[i,j] + \min\{c[i,k-1] + c[k,j]\}$$

where k is root and $i < k \leq j$

Example:

x :	10	20	30	40
Freq:	4	2	6	3

solution: we construct the table for n+1 levels where n = no of nodes.

- since n = 4, we have 5 levels, 0, 1, 2, 3, 4

	L=0	L=1	L=2	L=3	L=4
j-i =	0	1	2	3	4
	[0,0]	[0,1]	[0,2]	[0,3]	[0,4]
	[1,1]	[1,2]	[1,3]	[1,4]	
	[2,2]	[2,3]	[2,4]		
	[3,3]	[3,4]			
	[4,4]				
	1st	2nd	3rd	4th	5th

Now, we construct a table using values for each level.

i \ j	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

- The Level-0 (diagonal) values are 0.
- Now, compute level-1 values.

$c[0,1] = 4$ {frequency of 1st element.
$c[1,2] = 2$ { 2nd
$c[2,3] = 6$ { 3rd
$c[3,4] = 3$ { 4th
then fill in table.

- Level 2 values

$k = 1, 2$ $\quad c[0,2] = w[i,j] + \min\{c[i,k-1] + c[k,j]\}$

Since $i=0$ & $j=2$ & $i < k \le j$, $k = 1, 2$

$= w[0,2] + \min\{c[0,0] + c[1,2], c[0,1] + c[2,2]\}$

$= 4 + 2 + \min\{0+2, 4+0\}$

$= 4 + 2 + 2$

$= 8$ {The value minimum at k is choosen}

k=2,3 $C[1,3] = w[1,3] + \min\{ c[1,1] + c[2,3],\ c[1,2] + c[3,3] \}$

$= 8 + \min\{6, 2\}$

$= 10$

k=3,4 $C[2,4] = w[2,4] + \min\{ c[2,2] + c[3,4],\ c[2,3] + c[4,4] \}$

$= 9 + \min\{3, 6\}$

$= 12$

For Level 3:

k=1,2,3 $C[0,3] = w[0,3] + \min\{ c[0,0] + c[1,3],\ c[0,1] + c[2,3],\ c[0,2] + c[3,3] \}$

$= 12 + \min\{10, 4+6, 8\}$

$= 12 + 8$

$= 20$

k=2,3,4 $C[1,4] = w[1,4] + \min\{ c[1,1] + c[2,4],\ c[1,2] + c[3,4],\ c[1,3] + c[4,4] \}$

$= 11 + \min\{12, 5, 10\}$

$= 11 + 5$

$= 16$

for Level 4:

k=1,2,3,4 $C[0,4] = w[0,4] + \min\{ c[0,0] + c[1,4],\ c[0,1] + c[2,4],\ c[0,2] + c[3,4],\ c[0,3] + c[4,4] \}$

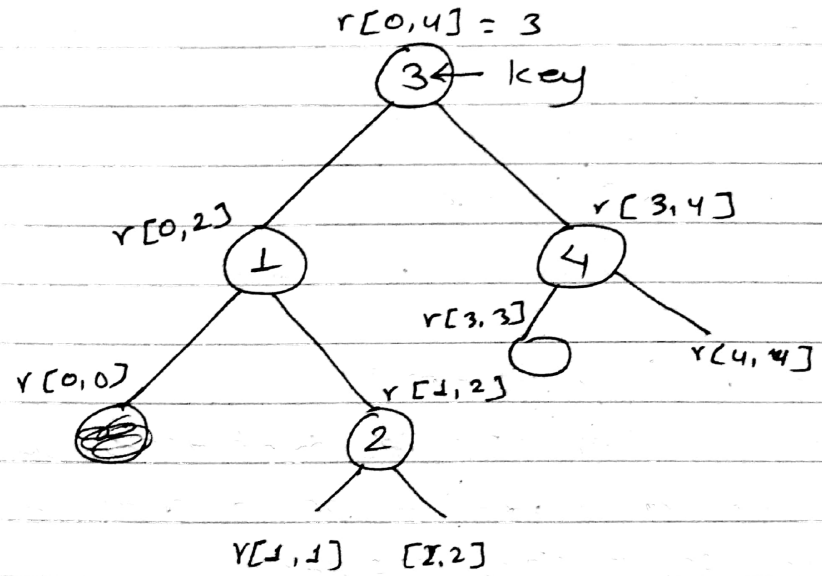
$= 15 + \min\{16, 16, 11, 20\}$

$= 15 + 11$

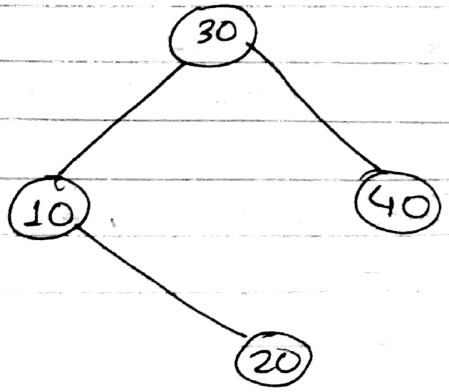
$= 26.$

Now, the result is on $C[0,n] = C[0,4]$ i.e. cost of optimal BST is 26 and root is 3.

The tree becomes.



Hence the tree is.



6. Differentiate between P, NP, NP-hard, NP-complete with example of each.

Answer: Given a problem, two situations exist, either we can solve the problem or there exists no solution. solvable problems are often referred as computable problems.

All the computable problems can be placed in any one of the classes according to their hardness:

- Easy → P
- Medium → NP
- Hard → NP-complete
- Hardest → NP-Hard

(i) P (Polynomial) problems:

- P problems refer to the problems where an algorithm would take a polynomial amount of time to solve the problem
- P class problems can be solved in polynomial time by deterministic Turing machine
- The Big-Oh notation of these problems is always a polynomial (i.e. $O(1)$, $O(n)$, $O(n^2)$). In general, $T(n) = O(c * n^k)$ for P problems where $c > 0$ and $k > 0$ also c & k are the constant and n is input size.

(ii) NP (Non-deterministic Polynomial) Problems:

- These problems satisfy the following:
  (a) solution can be verified in polynomial time by deterministic Turing machine.
  (b) solution can be found by non-deterministic Turing machine.

- These class of problems can be verified in polynomial time but cannot be solved in polynomial time.

- The time complexity of these problems tend to be exponential or factorial. In general,
  $T(n) = O(c_1 * k^{(c_2 * n)})$
  where $c_1 > 0$, $c_2 > 0$ and $k > 0$ also are constants
  n is the input size.

(iii) NP-Complete problems:

- The NP-complete class includes all the problems in class NP that satisfy an additional property of completeness.

- Completeness states 'for any problem that is NP-complete, there exists a polynomial time algorithm that can transform the problem into any other NP-complete problem.

- Simply we can say that, NP-complete problems belong to NP, but are among the hardest in the set

- The NP problems proven to be complete are:
  - Travelling salesman problem
  - Knapsack
  - Graph coloring

(iv) NP-Hard problems:

- This class contains the hardest, most complex problems in computer science. They are not only hard to solve but are hard to verify as well.

- These problems can all be reduced to any problem in NP and are at least as hard as any other problem in NP.

- Simply, A problem is classified as NP-Hard when an algorithm for solving it can be reduced to solve any NP algorithm.

Examples of P class:
One a game of checkers, what is the complexity of determining the optimal move on a given turn, if the size of board is 8×8. Then this is a polynomial-time problem.

NP-class: Soduku

- To solve the soduku problem, wouldnot have a polynomial runtime. However it we fed this puzzle with a possible solution, it would be much less complex to check with the polynomial run-time.

7. State cook's theorem. Discuss the cook's method to show ~~SAT~~ SAT is NP-hard and happens to be NP-complete.

- cook's theorem states that SAT is NP-complete.

- To show SAT is in NP-complete we claim that SAT $\in$ NP and SAT $\in$ NP-hard

Proof: SAT $\in$ NP.

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

Proof: SAT $\in$ NP-Hard

- we now need to show that any language $A \in NP$, is efficiently reducible to SAT.

- $A \in NP$, means that there exists a non-deterministic turing machine N running in $O(n^k)$ the time that decides A. We will construct a Boolean formula ϕ that is satisfiable iff some branch of N's computation accepts a given input w

Idea:

(i) construct a table for non-deterministic TM, N listing its configurations on some branch of its computation tree.

(ii) so determining if $w \in A$ is equivalent to whether or not there is a tabelau configuration such that N accepts input w

#	q_0	w_1	w_2	w_n #
#	w_1'	q_1	w_2	w_n #
#				#
#				

Encoding Tabelau as a formula:

- Each entry of tabelau T can be
  (a) state q_i of Turing Machine N, $q_i \in Q$
  (b) element of tape alphabet Γ or #
  (c)

Let $C = Q \cup \Gamma \cup \{\#\}$, then we define a propositional variable $x_{i,j,s}$ for every cell in row i, column j

and element $s \in C$

- we interpret $x_{i,j,s} = 1$ iff cell $[i,j]$ has the value s

Then, M accepts w iff:

(i) Each cell is well-defined or legal (i.e. it only contains one value type: a state, a letter or a #)

(ii) The first row is an initial configuration with w as the input

(iii) Each row follows from the previous row using the transition function given by M

(iv) Some row has a cell that includes an accepting state q_{accept}.

condition (i)

- In propositional logic cell $T[i,j]$ being filled with exactly one element is equivalent to

$$\phi_{ij} = \left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right)$$

- we have well defined tableau iff,

$$C_{cell} = \bigwedge_{i,j} \phi_{ij} \quad \text{is true.}$$

condition (ii): The formula $\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \ldots x_{1,O(n^k),\#}$ is true iff $w = w_1 \ldots w_n$ is given as input

condition 3(iii):

- Each row in the tabelau is a configuration following the previous row according to N iff each window in the tabelau is legal

- Hence the condition that each row follows from the previous according to N can be expressed as

$$\phi_{move} = \bigwedge_{1 \le i, j < O(n^k)} \phi_{window, i, j}$$

and, $\phi_{window, i, j} = \bigvee_{(a_1 \ldots a_6) \text{ legal}} (x_{i, j-1, a_1} \wedge x_{i, j, a_2} \wedge x_{i, j+1, a_3} \wedge x_{i+1, j-1, a_4} \wedge x_{i+1, j, a_5} \wedge x_{i+1, j+1, a_6})$

condition (iv)

- The tabelau is accepting iff some cell in the tabelau contains an accepting state.

$$\phi_{accept} = \bigvee_{ij} x_{i, j, q_{accept}}. \quad \text{if}$$

Combining all, Given a non-deterministic Turing machine M and some input w, we have shown that there exists a propositional formula ϕ as,

$\phi_{M,w} = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$ that is satisfiable iff M accepts w.

Polynomial Time Reduction:

- we have assumed 'N' runs in $O(n^k)$ time on inputs of length n, so the tabelau has $O(n^k)$ rows & $O(n^k)$ columns.
- The formula constructed by reduction has $O(2n^{2k})$ literals, since there is a constant size formula for each cell of the tabelau.
- The formula for each cell can be generated efficiently from a description of NDTM M.
- This together gives a reduction with runtime poly(n)
- This completes the reduction $A \leq_p SAT$, since, we can produce a formula $\phi_{M,w}$ in polynomial time that is satisfiable iff $w \in A$

Hence both the conditions $SAT \in NP$ & $SAT \in NP\text{-Hard}$ are satisfied, we can say that SAT is NP-complete.

8 where do you think online algorithms can be implemented? Explain with a suitable example of your own.

- Ato online algorithms can be implemented for problems where difficulty is not necessarily 'computationally hard' but rather the algorithm doesnot have all the information it needs to solve the problem

- online algorithms need to make decisions without full knowledge of the input. They have full knowledge of the past but no/partial knowledge of the future

- online algorithms are designed for seltings where inputs/data is arriving over time and we need to make decisions on the fly without knowing what will happen in the future.

The page replacement algorithm is an example of online algorithms. Since one cannot predit the pages system is going to demand earlier, the page replacement algorithm has no knowledge about the future.

Example: optimal page replacement Vs LRU algorithm.

consider following series of page requests:
a, b, c, d, a, b, e d e a e c b a
with frame = 3 for Optimal & LRU algorithm

i) Offline algorithm: optimal page replacement.

a	b	c	d	a	b	e	d	e	a	e	c	b	a
a	a	a	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	e	e	e	e	e	c	b	b
		c	d	d	d	d	d	d	d	d	d	d	d
M	M	M	M	H	H	M	H	H	H	H	M	M	H

Page faults = 7

ii) online algorithm: Least Recently used.

a	b	c	d	a	b	e	d	e	a	e	c	b	a
a	a	a	d	d	d	e	e	e	e	e	e	c	a
	b	b	b	a	a	a	d	d	d	d	c	c	c
		c	c	c	b	b	b	b	a	a	a	b	b
M	M	M	M	M	M	M	M	H	M	H	M	M	M

Page faults = 12

- In case of offline algorithms the future knowledge was known and hence only 7 page faults occured. However in the online algorithm: Least Recently used there occured 12 page faults but the algorithm did its job.

- In the LRU algorithm, we couldnot take the decision by considering the future knowledge and we were only allowed to use the past knowledge.

- However in first algorithm the page faults were minimal but one cannot implement that algorithm since no one can predict the future request of pages.

- In such case, we take decisions based on current & past references. i.e. current/past knowledge. That is the beauty of al online algorithms.

9 Prove that formula satisfiability is a NP-class problem:

Solution: for a problem to be in NP-class it satisfies following two conditions:

(i) The problem must be solveable by non-deterministic turing machine in polynomial time.
(ii) But the ~~problem~~ solution can be veritied by deterministic turing machine in ~~solved~~ polynomial time.

Hence if we prove these two conditions are satisfied by the formula satisfiability then we can say that formula satisfiability is a NP problem.

let us consider a formula $F=(\neg x) \wedge (y \vee z)$ then formula satisfiability states to find out either there exist some combination of inputs such that the value of F is true(1).

→ Then we can attempt to find the solution by brute-force approach.

x	y	z	$F=(\neg x) \wedge (y \vee z)$
0	0	0	-
0	0	1	-
0	1	0	-
⋮	⋮	⋮	-
1	1	1	-

we try every possible input and determine the value of formula F.
Since there are two possible values for each input either 0 or 1 this gives 2^n possible input patterns. This.

- This means the brute-force approach takes exponential time to obtain the solution. $O(2^n)$

- Since it takes exponential time for solution, we say that our condition (i) is not met.

Now, lets say that we choose an input pattern at random.

say $x = 0$ $y = 1$ $z = 0$

then, our formula becomes: $\neg x \wedge (y \vee z)$

or, $(\neg 0) \wedge (1 \vee 0)$

or, $1 \wedge 1$

$= 1$ (True)

- However, verifying either the given input satisfies the formula is of polynomial time. and this can be simulated by a deterministic turing machine. Hence our (ii) condition is also met.

- The formula satisfiability problem can be solved in exponential time and verified in polynomial time by a deterministic turing machine, we say formula satisfiability is a NP-class problem.

10. Explain work optimal PRAM algorithm to solve prefix computation problem with an example.

olution: The work optimal PRAM algorithm for prefix computation is,

step 1: Processor i ($i = 1, 2 \ldots n/\log n$) in parallel computes the prefixes of its $\log n$ assigned elements. ~~of~~ $x_{(i-1)\log n + 1}, x_{(i-1)\log n + 2} \ldots x_{i \log n}$ This takes $O(\log n)$ time. Let the results be $z_{(i-1)\log n + 1}, z_{(i-1)\log n + 2}, \ldots z_{i \log n}$

Step 2: A total of $\frac{n}{\log n}$ processors collectively Perform following.

~~(a)~~ recursively compute the prefixes of $n/\log n$ elements say $z_{\log n}, z_{2\log n}, z_{3\log n} \ldots z_n$ and results be $w_{\log n}, w_{2\log n} \ldots w_n$.

Step 3: Each processor updates the prefix it computed in step 1 as follows.
Processor i computes and outputs $w_{(i-1)\log n} \oplus z_{(i-1)\log n + 1}, w_{(i-1)\log n} \oplus z_{(i-1)\log n + 2} \ldots w_{(i-1)\log n} \oplus z_{i \log n}$, for $i = 2, 3 \ldots \frac{n}{\log n}$

Processor 1 outputs $z_1, z_2 \ldots z_{\log n}$ without any modifications.

optimality of Algorithm

- The prefix computation is solved by linear algorithm in $O(n)$ time.
- Parallel algorithm uses $n/\log n$ processors and work done is $O(\log n)$ then,

$$\text{efficiency}(\Theta) = \frac{S(n)}{P.T(n,P)} = \frac{O(n)}{\frac{n}{\log n} * \log n} = \frac{O(n)}{O(n)} = O(1)$$

or, $\Theta = 1$

Hence above algorithm is work optimal.

example:

2, 1, 3, 6, -2, 4, -3, 5

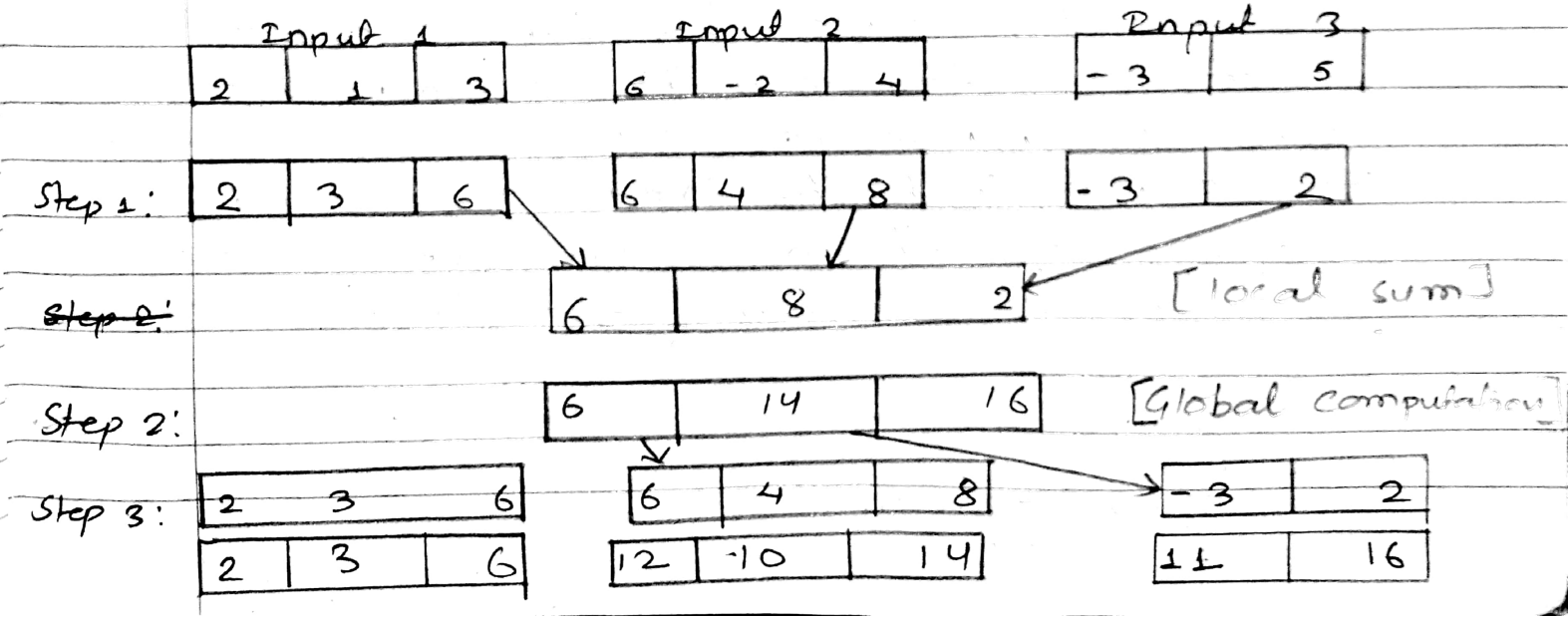
solution:

Since we have 8 elements $(n) = 8$

no. of processors $= n/\log n = 8/\log 8 = \frac{8}{3}$

$= 3$ processors

each, Processor has $n/\log n$ inputs $= 3$ inputs.



Hence the output is:

2, 3, 6, 12, 10, 14, 11, 16

11. Explain how you can implement back tracking algorithm to calculate sum of subsets with an example.

Answer: Subset sum problem is to find subset of elements from the given set such that the sum of these elements is equal to a given number k.

The two considerations in this problem are:

(i) The set contains non-negative values

(ii) The input set has unique values (NO duplicates are present)

- To obtain the solution for sum of subsets problem we start with the root node with value 0 and include all the values one by one and expand the tree in DFS order, find sum at each node until the solution is found or the bounding condition is met.

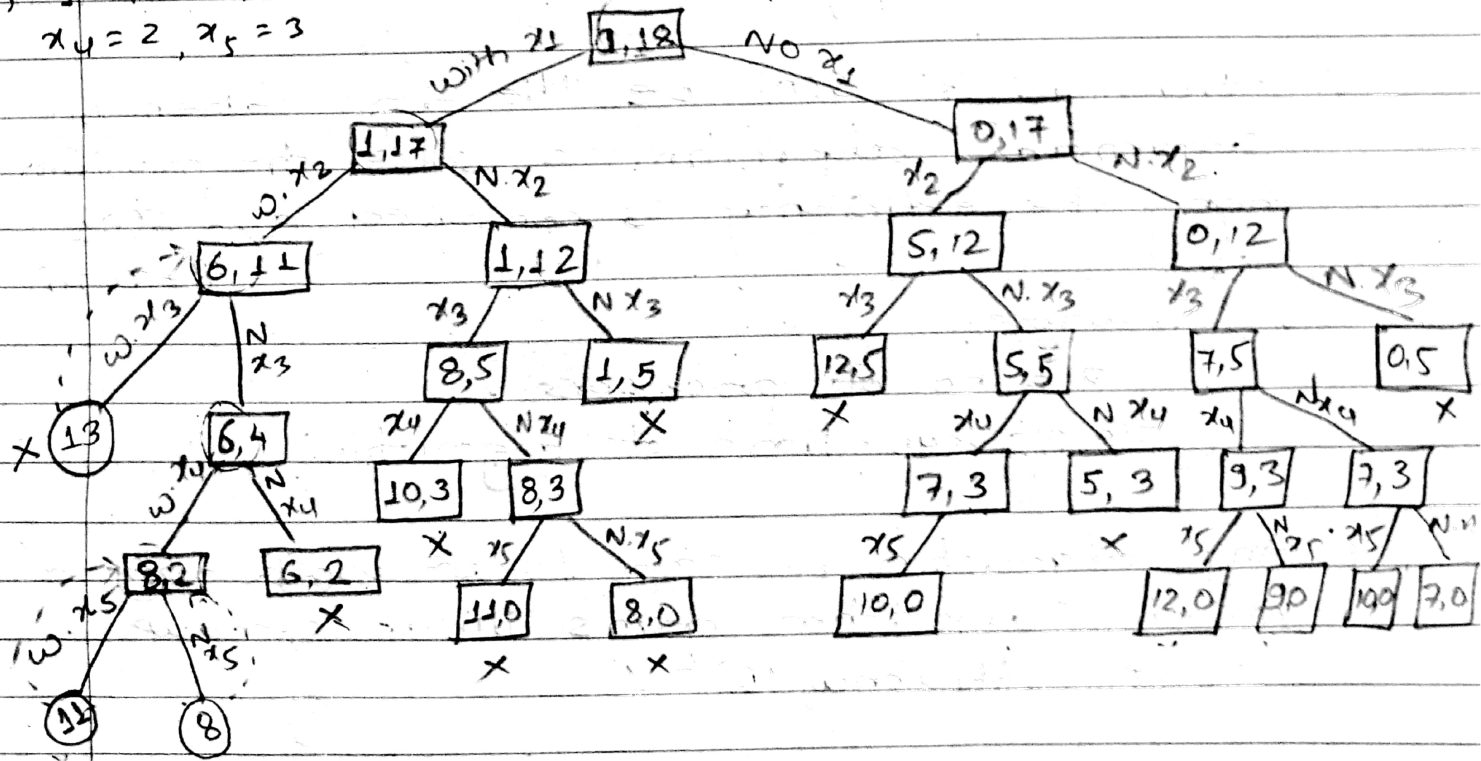
The bounding condition is,

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

when the bounding condition is met, we stop expanding the tree and backtrack to the upper level and include other elements. In many scenarios, it saves considerable amount of processing time.

Example: Find the subsets whose sum is 10 where set = {1, 5, 7, 2, 3}

Here, $x_1 = 1$, $x_2 = 5$, $x_3 = 7$
$x_4 = 2$, $x_5 = 3$



Using the algorithm we had to backtrack from 9 nodes without reaching the maximum depth of tree and the derived subsets are found as.

$S_1 = \{1, 7, 2\}$ $S_2 = \{5, 2, 3\}$ $S_3 = \{7, 3\}$.

This backtracking has saved computational resources and time too.

Q NO: 16: Explain work done and its efficiency with a suitable example of your own for PRAM. when do you confirm that the algorithm is work optimal.

Solution: For any PRAM algorithm,

(i) work done: if P-procesor parallel algorithm for a problem runs in $T(n,p)$, the total work done by this algorithm is $P * T(n,p)$

(ii) efficiency: if $S(n)$ be the time taken by sequential algorithm to solve the problem then the efficiency is computed as $\frac{S(n)}{P * T(n,P)}$

where,

P = no of processors

$T(n,P)$ = run-time of algorithm.

(iii) Any algorithm is said to be work optimal if the algorithm gains linear speedup or it efficiency $(\theta) = 1$

To, shaw this lets take prefix-sum computation in parallel environment.

Algorithm:

Step 1: Procesor i ($i = 1, 2 \ldots n/\log n$). compute the prefix of ($\log n$) sized elements.

this takes $O(\log n)$ time.

Step2: $(n/\log n)$ procesors compute prefixes of $(n/\log n)$ elements.
This step takes $O(\log n)$ time.

Step3: Each processor updates the prefix computed in step 1 with the prefix computed in Step 2 ~~of bottoms~~.

example: 1, 2, -2, 4, 3, 5, 7, -8

1 2 -2 4 3 5 7 -8

1 3 1 4 7 12 7 -1

1 12 -1

1 13 12

1 3 12 4 37 512 7 -1

1 3 7 5 8 13 20 12

Since, we have used $n/\log n$ procesors and it takes $O(\log n)$ time, the work done is,

$$P * T(n, P) = (n/\log n) * O(\log n) = O(n)$$

The prefix sum can be computed using sequential algorithm in O(n) time. and the efficiency can be computed as,

$$\theta = \frac{S(n)}{P * T(n,P)} = \frac{O(n)}{\frac{n}{\log n} * \log n} = \frac{\theta(n)}{n} = O(1)$$

Since the efficiency is linear i.e. O(1), we can say that the algorithm is work optimal.

Q. NO.17: Highlight on the implementation of the dynamic algorithm. Use dynamic algorithm to solve string editing problem.

Solution: Dynamic programs are implemented where a given problem can be divided into smaller sub-problems and their results can be reused to obtain the solution to larger problem.

Dynamic programming is an approach to optimize plain recursion. whenever we see a recursive solution with same inputs, we can optimize it using Dynamic Programing.

The problems that can be solved using DP are:

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi

String editing problem:

- Given two strings 'src' and 'dest', the string editing problem is to obtain minimum number of operations so as to convert 'src' to 'dest'. The operations permitted are:

(i) Insert a character - 1
(ii) Delete a character - 1
(iii) Replace a character - 2

Solution

- Let x and y be source & destination strings respectively where $x = x_1, x_2 \ldots x_n$ & $y = y_1, y_2 \ldots y_m$
- Let cost(i, j) is the minimum cost for transforming 'src' to 'dest' Then,

if $i = j = 0$, $c(i,j) = 0$
if $i > 0$ & $j = 0$, then $c(i,j) = cost(i-1, 0) + D(x_i)$
if $i = 0$ & $j > 0$, then $c(i,j) = cost(0, j-1) + I(x_i)$
else,

$$c(i,j) = \min \begin{cases} cost(i-1, j) + 1 \\ cost(i, j-1) + 1 \\ cost(i-1, j-1) + \begin{cases} 2 & \text{if } x(i) \neq y(j) \\ 0 & \text{if } x(i) = y(j) \end{cases} \end{cases}$$

Convert string ababoa to ~~ebabac~~ aebbab with minimum cost possible.

		a	b	a	b	a	b
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
c	2	1	2	3	4	5	6
b	3	2	1	2	3	4	5
b	4	3	2	3	2	3	4
a	5	4	3	2	3	2	3
b	6	5	4	3	2	3	2

Hence the minimum edit distance is 2

1 → insertion

1 → deletion

Q.NO.18: Explain Aggregate method of complexity analysis with an example of your own.

Answer: Aggregate method of complexity analysis is an approach of amortized analysis.

- In aggregate method, we determine an upperband for a sequence of n operations say, T(n), then we compute the average and set the amortized cost of n operations equal to T(n)/n. Hence in aggregate analysis each operation has same cost.

Example: Modified stack

- stack are linear data structure with 2 operations:
  Push() and Pop(), both operations are constant time operations and any sequence of n operations will result to O(n) total time.

- Let's introduce a new operation to the stack, Multipop(S, k), that pops top k elements from stack and if runs out of elements it empties the stack.

The pseudo code looks like:

```
multipop(S, k)
    while S is not empty & k > 0;
    k = k - 1
    stack.pop()
```

Analysis:

(a) one multipop can take O(n) time.
'n' multipop takes $O(n^2)$ time.
However this is not the case, since multipop() cannot function until there has ~~not~~ been a Push to the stack because there would be nothing to pop.

(b) Any sequence of Push, Pop and Multipop opera

tions can take at most $O(n)$ time, because the only non constant operation multipop can take at most $O(n)$ time, if there have been n constant time push operations.

(c) In the worst case, there are n-constant time operations and a single operation with $O(n)$ time.
so using aggregate analysis,

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$$

Hence, this stack has amortized cost of $O(1)$ per operation.

Q. No9: Demonstrate odd-even merge sort in Butterfly Network. calculate its time complexity.

Answer: consider two arrays $x_1 = 5, 8$ and $x_2 = 1, 12$ to be merged in butterfly network.

Then, we can seperate the odd and even parts of these arrays by placing these elements on level d of a butterfly network. The Butterfly network we are going to need is, $2m \leq 2^d$ or $2 \times 2 \leq 2^d$ $4 \leq 2^d$ and $d = 2$,

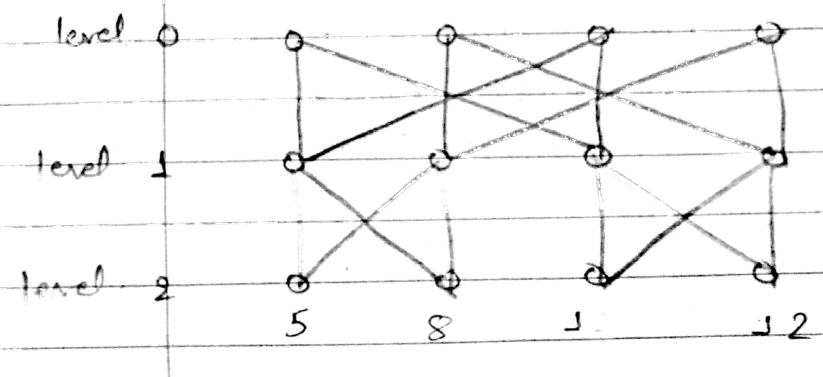


Fig: Initial network.

Now, we send these elements to level $(d-1)$ by using direct links for the 1st half of rows & using cross links for 2nd half of rows. Then we have.

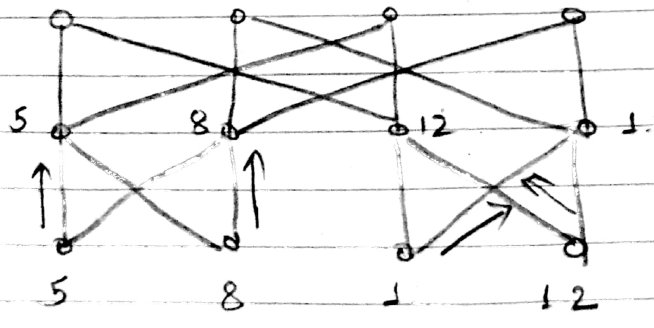


Fig: seperation of E_1 O_2 & E_2, O_1

Now, we have E_1 & O_2 in the even sub butterfly and E_2 and O_1 in the odd sub butterfly

and $E_1 = 5$ $O_2 = 12$

$E_2 = 1$ $O_1 = 8$

Now we recursively merge in the similar fashion to obation $A = E_1 O_2$ & $B = O_1 E_2$ and shuffle them to obtain,

$A = 5, 12$ $B = 1, 8$

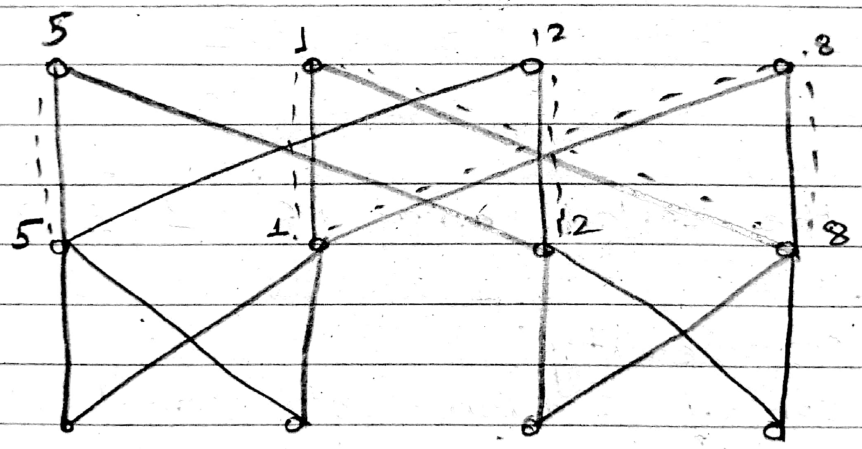
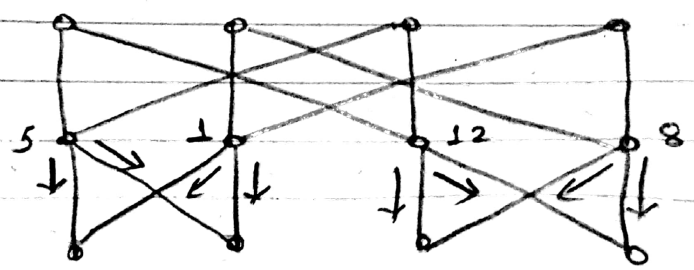
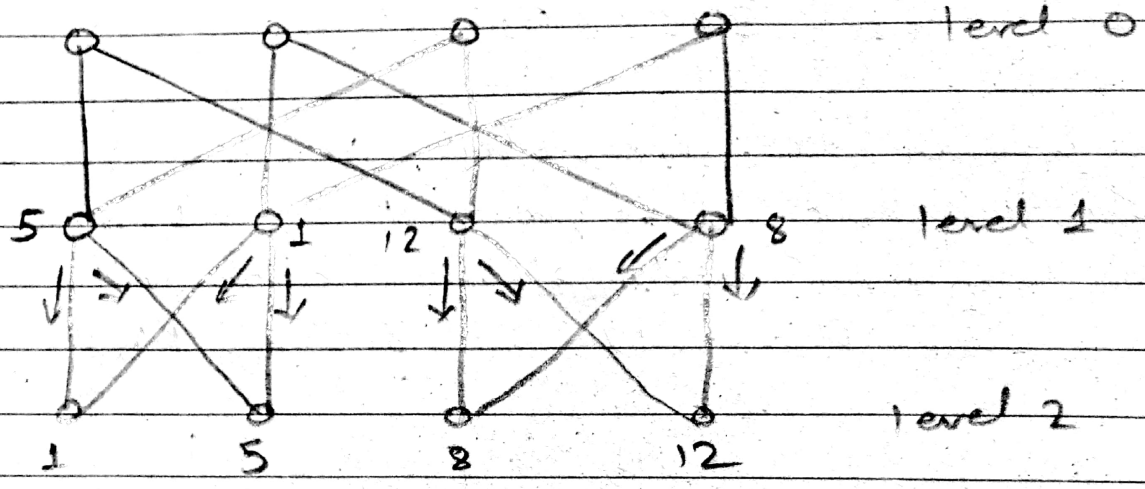


Fig: Shuffling of $a_0 b_0, a_1 b_1 \ldots a_m b_m$.

Now each node at level $(d-1)$ transmits data at both direct & cross links where the even row stores minimum value & odd row stores maximum value.



After the values are compared the final level 'd' nodes contain the sorted values starting from row 000 to ~~row~~ right row



Hence the sorted array is 1, 5, 8, 12.

Time complexity:

→ Two sorted sequences of length m each can be merged on a butterfly network of level d in O(d) time since $2m = 2^d$

→ Similarly merging also can be done in O(d) time.

Hence the time complexity is $O(d) + O(d)$
$= O(d)$

Q. No. 20: Explain the Mesh algorithm for maximum selection with n processor. will it work optimal?

- The mesh algorithm for maximum selection with n processor is a randomized algorithm, the steps can be given as,

(i) The randomized algorithm chooses a random sample, S from X of size $n^{1-\epsilon}$ where, $\epsilon = 0.6$ is sufficient.

(ii) Two elements l_1 & l_2 are identified from S, l_1 & l_2 are also known as splitter

(iii) The keys l_1 & l_2 are such that the element to be selected has a value between l_1 & l_2 with high probability and the number of keys in range $[l_1, l_2]$ is small.

(iv) Now, we partition X into X_1, X_2 & X_3 where $X_1 = \{x \in X \mid x < l_1\}$ $X_2 = \{x \in X \mid l_1 \leq x \leq l_2\}$ and $X_3 = \{x \in X \mid x > l_2\}$

(v) we also count $|X_1|$, $|X_2|$, $|X_3|$, and if $|x_1| < i \leq |X_1| + |X_2|$, the element to be selected lies in X_2. if this is true, we proceed further or else we start all over again.

(vi) The element to be selected will be $(i - |X_1|)^{th}$ smallest element of X_2

- The above process of sampling and elimination is repeated until the appropriate number of remaining keys is $\leq n^{0.4}$.
- After this, we perform an appropriate selection from out of the remaining keys using the sparse enumeration sort.

Analysis:
- The step 1 takes $O(1)$ time on mesh
- prefix computation can be done in $O(\sqrt{p})$ time
- concentration & sparse enumeration sort takes $O(\sqrt{p})$ time.

Hence: selection from $n = p$ keys can be performed in $O(\sqrt{p})$ time on $\sqrt{p} * \sqrt{p}$ mesh.

Q 21: How can you implement data concentration on Hypercube? Explain with an example of your own.

Answer: A hypercube is a d-dimensional structure, numbered using d-bits. Hence there are 2^d processors in a d-dimensional hypercube, represented as H_d.

- A hypercube has several variants like butterfly, shuffle-exchange network and cube-connected cycles

- Algorithms for hypercube can be adapted for butterfly network and vice-versa. Hence algorithm for butterfly network can be applied on hypercube too.

Data Concentration on Hypercube:

- on a hypercube H_d assume there are $k < p$ data items distributed arbitrarily with at most one datum per processor. then the problem of data concentration is to move the data into processors $0, 1 \ldots k-1$ of H_d.

- for this, we map the given hypercube to a butterfly network and then apply the concentration algorithm of butterfly.

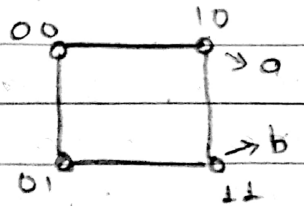


Fig: a 2-D hypercube.

consider a 2-D hypercube that has two data items at procesor 10 & 11 be a & b respectively.

Then we map the 2-D hypercube to 2-d butterfly network.

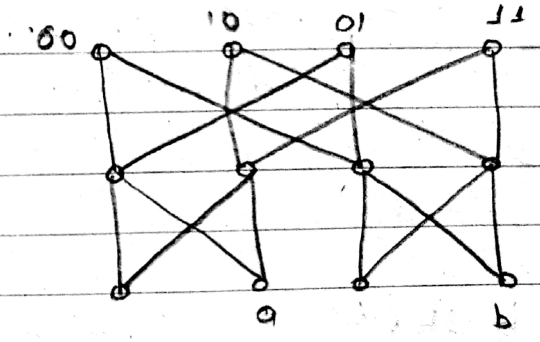


Fig: Mapping 2-D hypercube to 2-d butterfly network

Step1: The first step in data concentration is to compute prefix sum and to do so, we map the butterfly network B_2 to binary network of depth 2.

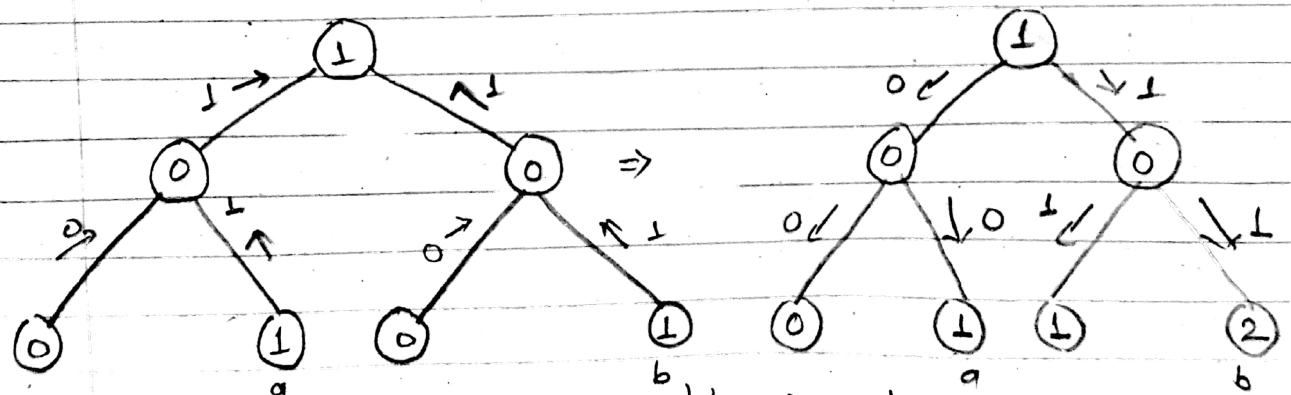


fig: Prefix computation on binary-tree.

The prefix sum is {0, 1, 1, 2}. This means that the element a must be moved to position 1 and the element b must be moved to position 2. ie. row 0 & row 1.

In the second phase, using the greedy approach the packets are routed to their destination. The path followed is shown dashed lines.

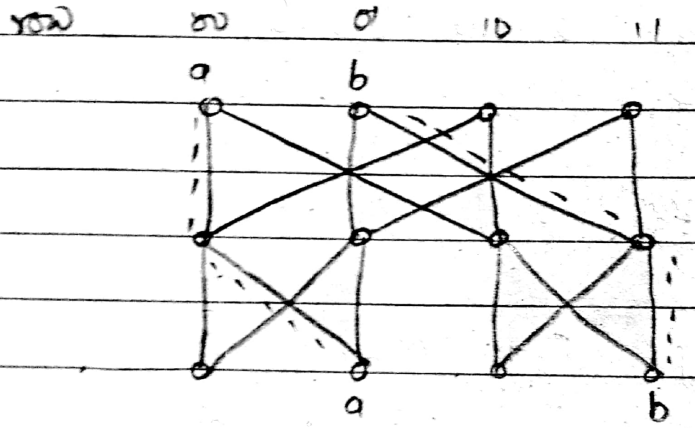


Fig: PPR - greedy routing algorithm applied on butterfly network.

Finally we map the butterfly network to hypercube.

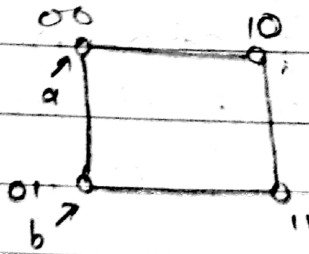


Fig: Mapping of butterfly network to hypercube.

Q.22: what do you understand by embedding of networks? Explain with an example of your own.

- Many networks such as ~~rig~~ ring, mesh and binary tree can be shown to be subgraphs of a hypercube. In general, mapping of one network into another is called an embedding.

- Let ~~G(G1,~~ $G(V_1, E_1)$ and $H(V_2, E_2)$ are any two connected networks, an embedding of G into H is a mapping of V_1 into V_2

- Embedding are important, to simulate one network on another. Using an embedding of G to H, an algorithm designed for G can be simulated on H

- Terminologies used for embedding:
  Expansion: is obtained as V_2/V_1
  Dilation: Length of longest path any link of one network structure is mapped with another.
  Congestion: Number of paths on the H, corresponding to the links of G that it is on.

Example: Embedding of a binary-tree

- we can embed a binary-tree to a hypercube in many ways.

- A ~~leaf~~ p-leaf full binary tree, where $p = 2^d$, can be embedded into H_d.
- A ~~p-leaf~~ full binary tree with levels d has $(2^{d+1} - 1)$ nodes / processors, however a d-dimension hypercube has 2^d processors.
- Hence, In a clear sense $(2^{d+1} - 1) > 2^d$, for any value of d and the mapping cannot be one-to-one
- More than one processor of a binary tree has to be mapped to a single processor of hypercube.

(i) If tree leaves are 0, 1, 2, ... P-1, then leaf i is mapped to i^{th} processor of H_d.

(ii) Each internal processor (except leaf node) is mapped to the same processor of H_d as its leftmost descendent leaf.

consider a level-2 full binary tree,

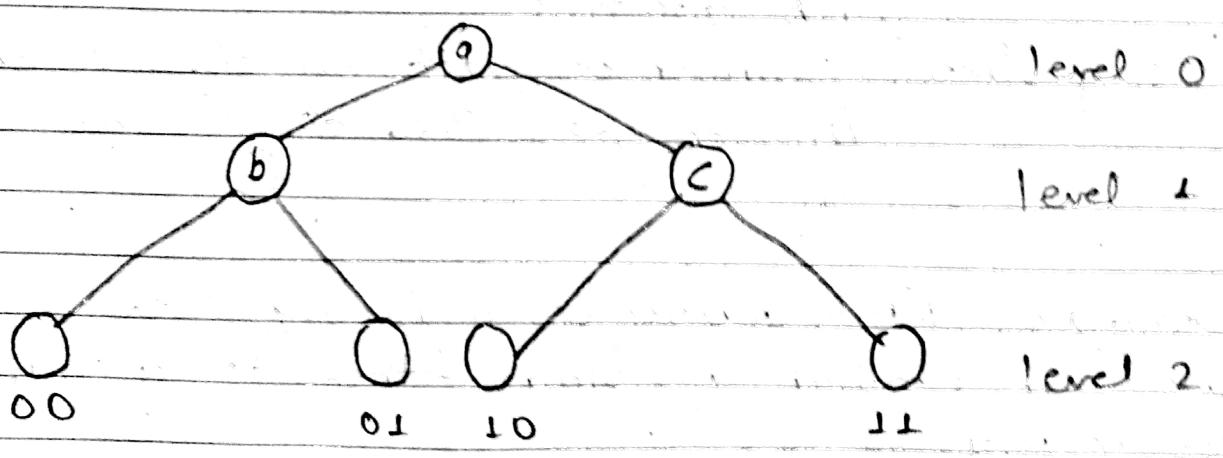


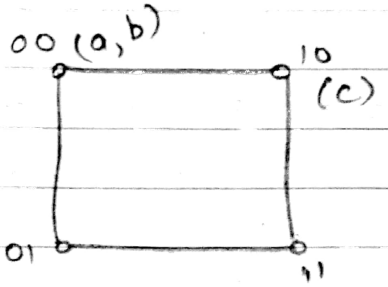
fig: level-2 binary tree.

Since the binary tree has level 2, it has $2^{d+1} - 1$ processors

$= 2^{2+1} - 1 = 2^3 - 1 = 7$ nodes,

However, the 2-d hypercube only has $2^d = 2^2 = 4$ processors. Then we have to map 3-additional nodes to the same node of hypercube.

The embedding looks like.



The internal node b is mapped to the processor 00, since it's leftmost descendent is node 00. Similarly, the node a is also mapped to the node 00 and node c is mapped to node 10.

Node	No. of mappings.
00	3 (1-leaf & 2-internal)
01	2 (1-leaf & 1 internal
10	1 (1-leaf)
11	1 (1 leaf)

Total node = 4 in hypercube.

Total node = 7 (in binary tree)

Q.25: Write short notes on:

a) Maximum CLIQUE problem:

- In an undirected graph, a clique is a complete sub-graph of the given graph. complete sub-graph means, each vertex in the subgraph is directly connected with all other vertices of this sub-graph.

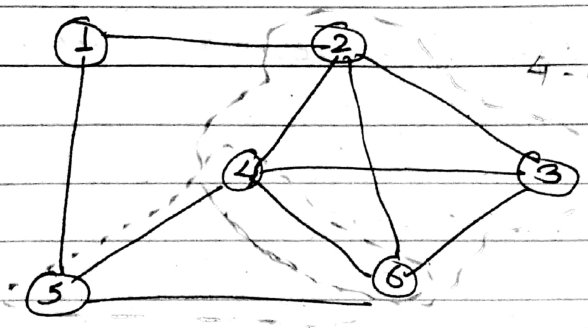
- The maximal clique problem is to find the maximum sized clique of a given graph G i.e. a complete sub-graph of G that contains the maximum number of vertices.

- Every time a new point is added, the number of total cliques that must be searched at least doubles; hence we have an exponentially growing problem.

- consider a social networking application, where vertices represent people's profile and the edges represent mutual acquaintance in a graph. In this graph, a clique represents a subset of people who know each other.

Analysis:

- Max-CLIQUE problem is a non-deterministic algorithm.
- Here, we try to determine a set of k distinct vertices and then we try to test whether these vertices form a complete graph.
- The problem is Np-complete.



3-Clique:-

In the graph above, the vertices 2, 3, 4, 6 form a complete graph. Hence, this sub-graph is a clique. Hence it is a 4-clique.

b) Subset Sum Problem

- Given a set of integers $S = (S_1, S_2, S_3 \dots S_n)$ where s are n-positive integers and a integer I, the subset sum problem is to choose/find a subset s' of given sets such that sum of elements of subset s' is equal to integer ⊗ I.

- The two considerations in this problem are:
  (i) The set contains non-negative values
  (ii) The input set has unique values (NO duplicates are present)

- Solution to the 'sum of subsets' problem can be obtained using the backtracking algorithm

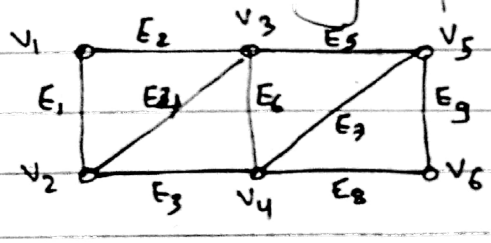
- The subset sum problem takes $O(2^n)$ time

complexity but is significantly faster than the recursive approach.

c) Vertex covering problem

- Given a graph $G=(V, E)$, a set of vertices $S \subseteq V$ is a vertex cover, if every edge is incident on a vertex in S.

- consider the graph below:



if we choose vertex V_2, V_3, V_5 & V_6 then all the edges are covered / are incident in the vertex V_2, V_3, V_5, V_6. and it is a vertex cover of size 4. $V_2: E_1, E_4, E_3$, $V_3: E_2, E_5, E_6$

$V_5: E_7$ E_9, $V_4: E_8$

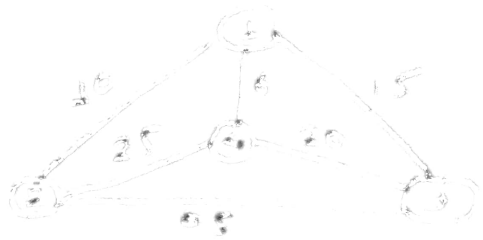
However, if we try to find the vertex cover of size less than four, we cant find it, since all the edges are not covered.

d) Travelling salesman problem:

- Given a set of cities and distance between every pair of cities, the problem is to find

the shortest possible route that visits every city and returns to the starting point.

Consider the given set,



For the given graph, the solution is 1 2 3 4 and the cost is minimum cost = 41

TSP can be solved using the dynamic programming approach and has exponential time. The approach is infeasible even for slightly higher number of vertices.

Sparse Enumeration Sort

- The approach of sorting in which the number of keys to be sorted is much less than the network size is referred as the sparse enumeration sort.

- If the network size is P, the number of keys n to be sorted is typically assumed to be n^x for some constant $x \leq 1/2$

- Sparse enumeration sort can be done by computing the rank of each key & sending the key to

its correct position in sorted order.

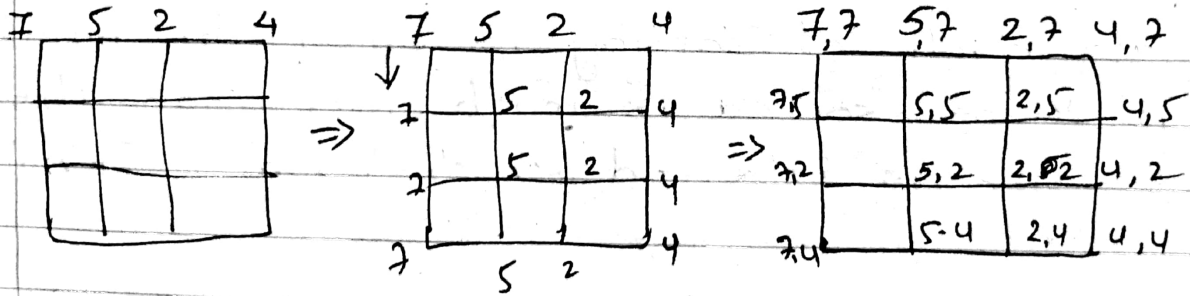
Algorithm:

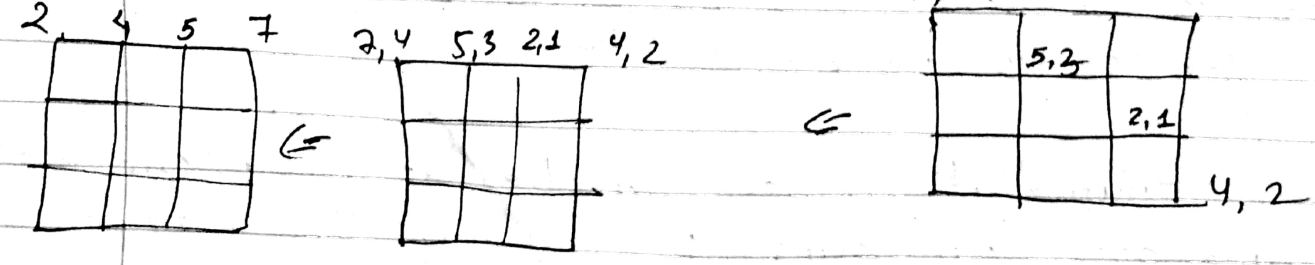
1) In parallel
   (a) for $1 \le j \le \sqrt{p}$ broadcast k_j along column j
   (b) for $1 \le i \le \sqrt{p}$ broadcast k_i along row i

2) In parallel,
   (a) for $1 \le i \le \sqrt{p}$, compute rank of k_i in each row i using prefix sum computation.
   (b) for $1 \le j \le \sqrt{p}$, send the rank of key k_j to $(1, j)$

3) In parallel for $1 \le r \le \sqrt{p}$ route the key whose rank is r to node $(1, r)$.

eg.





1) Pretix computation in Mesh

- consider a $\sqrt{p} \times \sqrt{p}$ mesh at which there is an element at each processor and a binary associative unit time computable operator $\oplus$ is defined.
- Then for any element e_i; $i=1$ to p, the pretix computation is the problem of determining the pretix at each operator such that at each processor i, pretix is $\sum_{j=1}^{i} e_i$
- The pretix computation on mesh can be done in three phases:

(i) Row i (for $i = 1$ to $\sqrt{p}$) computes the pretix of its $\sqrt{p}$ elements. At end the processor (i, j) has $y(i,j) = \sum_{q=1}^{j} z(i,q)$

(ii) only $\sqrt{p}$ column computes pretixes of sums computed in step 1. & sends the result of processor $(i, \sqrt{p})$ to row $(i+1, \sqrt{p})$ for $i = 1, 2 \ldots \sqrt{p}-1$

(iii) Broadcast the sum at processor $(i, \sqrt{p})$ to row $i+1$ & finally update at each processes $\Leftarrow z(i, \sqrt{p}) + y(i+1, j)$.

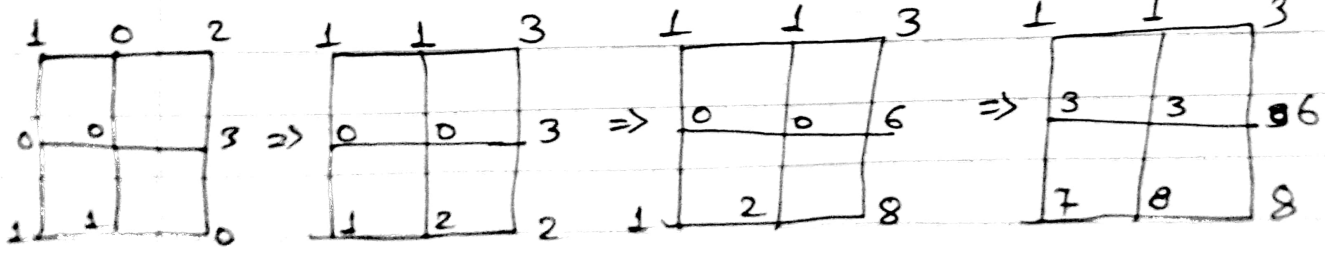


Fig: pretix computation in Mesh.

Q.N.12: Explain odd-even merge sort in a butterfly network.

Answer: We are given with two sorted sequences to be merged. Let $x_1 = k_0, k_1 \ldots k_{m-1}$ and $x_2 = k_m, k_{m+1} \ldots k_{2m-1}$ be the two sorted sequences to be merged, where $2m = 2^d$.

Then there exists an butterfly network of depth d, that can merge x_1 & x_2.

Process:

i) we separate the odd and even parts of x_1 and x_2, let them be O_1, E_1, O_2, E_2.

ii) we recursively merge E_1 with O_2 to obtain $A = a_0, a_1 \ldots a_{m-1}$ also O_1 & E_2 are recursively merged to obtain $B = b_0, b_1 \ldots b_{m-1}$.

iii) Now A & B are shuffled to form $c = a_0 b_0 a_1 b_1 a_2 b_2 \ldots a_{m-1} b_{m-1}$ and compare a_i with b_i for $(0 \le i \le m-1)$ and interchange them if they are out of order. The resulting sequence is in sorted order.

Example: Let $x_1 = 4, 8, 13, 17$ and $x_2 = 9, 11, 16, 18$

For this case:

$O_1 = 8, 17$ $\quad O_2 = 11, 18$

$E_1 = 4, 13$ $\quad E_2 = 9, 16$

Now we merge E_1 with O_2 to obtain A and
A = 4, 11, 13, 18
Similarly Merge O_1 with E_2 to obtain B and
B = 8, 9, 16, 17

iii) Now shuffling A with B to obtain C we have,
C = 4 8 11 9 13 16 18 17

compare a_i with b_i for $(0 \le i \le m-1)$ and interchange them if needed to get sorted order.

C = 4, 8, 9, 11, 13, 16, 17, 18

Implementation in a butterfly network:

(1) consider a buttertly network with depth d, B_d
(2) we have to separate the odd and even parts of X_1 and X_2. this can be done in following way:

(i) feed input X_1 & X_2 at at level d
(ii) we route the keys in the first m rows using direct links and other keys using cross links.

after this, we have E_1, O_2 in even sub-butterfly and E_2, O_1 in odd sub-butterfly

(3) As in step 2, we recursively merge E_1 & O_2, E_2 & O_1, to obtain A & B.

(4) After the recursive calls are over A, will be ready in the even sub-butterfly at level (d-1) and B will be ready in odd sub-butterfly.

(5) To shuffle & compare, each processor at level (d-1) sends its results along the cross links as well as the direct links

when processor in row i at level d receives two data, it keeps minimum value if i is even; otherwise it keeps the maximum.

There are two phases in the overall algorithm, In the first phase, data flows from bottom to top and in second phase data flows from top to bottom.

In the first phase, when any data progresses toward level 0, it enters butterflies of smaller and smaller dimension, when all the data reach at level 0, 1st phase is complete.

In the second phase, data flow from top to bottom one at a time, when data are at level l, ~~both~~ each processor at level l sends its datum both along the direct link and along the cross link. In the next step, processors in level l+1 keep either the minimum or the maximum of two items received from above

depending on the even row or odd row. When the data reaches level d, final result is computed.

Q. NO 13: What is packet routing in Mesh? Demonstrate the broadcasting algorithm on Mesh.

Answer: A mesh is an 'm x n' grid where there is a processor at each grid point. Each processor is connected via edges which are bi-directional communication links.

Each processor can be labelled with tuple (i, j) and has some local memory. There also exists a global clock which synchronizes all processors.

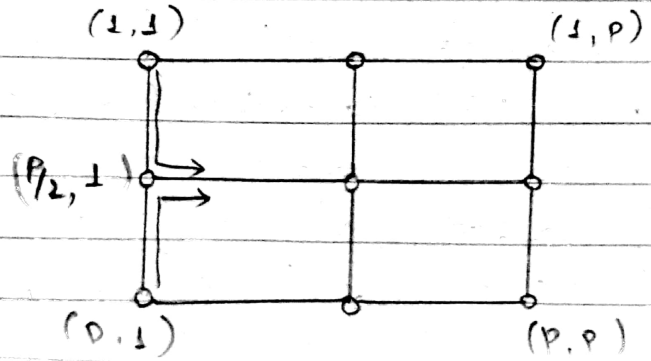
- Packet routing is a primitive Inter-process communication operation. In each packet has it's origin and its destination, then, In a mesh of p x p processors, q can be an arbitrary packet with (i, j) as its origin and (u, v) as its destination. Then packet can travel in two steps.
  (i) Travel along column j to row u
  (ii) Travel along row u to destination (u, v)

- The PPR problem on a mesh of $P \times P$ dimension, if a packet with origin $(1,1)$ has destination (P,P), then step 1 is done in $(P-1)$ steps and step 2 can be done in $(P-1)$ steps. Hence the lower bound on the worst case routing time of any algorithm is

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

- However, there is a severe drawback in this algorithm, consider all the packets originated at ~~row 1~~ column 1 are destined for row $P/2$.

For this problem, the processor $(P/2, 1)$ gets two packets (one from above, one from below at every time step. since both of these want to use the same link, only one can be sent and other packet has to be queued.
Hence, the queue size needed is as large as $P/2$.



The greedy approach to PRR in mesh needs large queues.

Broadcasting in Mesh:

- The problem of broadcasting in an interconnection network is to send a copy of a message that originates from a particular processor to a specified subset of other processors. Unless specified, this subset is assumed to consist of every other processor.

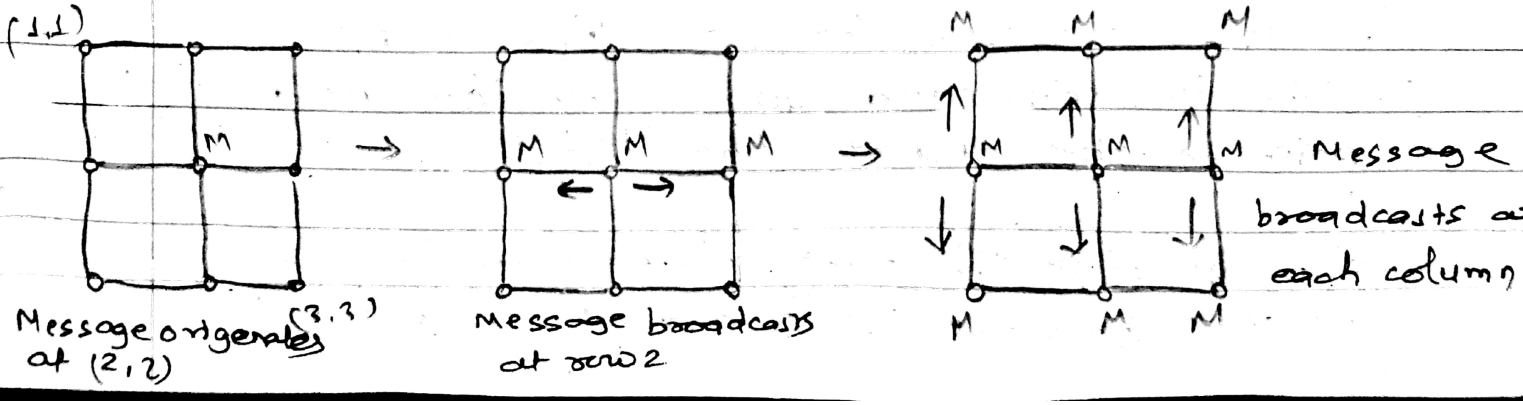
- In case of a P x P mesh, broadcasting can be done in two phases. If (i, j) is the processor of message(M) origin then broadcasting can be done in following way:

  (i) M could be broadcast to all the processors in row i.

  (ii) In phase 2: broadcasting of M is done in each column. This algorithm takes $\leq P-1 + P-1$ steps $\Rightarrow 2(P-1) = O(P)$.

Example: consider a 3 x 3 mesh with message M, originating at (2, 2) then broadcast is done as:

i) broadcast M to nodes of row 2
ii) broadcast M to nodes in each column 1, 2 & 3.



.NO:14: How can you compute Rank in linear array? Explain

swer: The list ranking problem is of identifying the distance for each object in a linked list from the end of the list.

Given a Linked list L with n, objects, we compute for each object n in L, its distance from the end of the list

The input to the problem is a list, given in form of array of nodes. A node holds some data and a pointer to its right neighbour in the list. Then we have to compute rank for each node.

Rank can be computed by use of the parallel algorithm with idea of pointer 'jumping'

Process:

(i) Initially each node points to immediate right neighbour.

(ii) In one step of pointer jumping, the right neighbour of every node is modified to be right neighbour of its right neighbour.

(iii) This is continued until all the nodes point to the end of the list

also in step (ii) each node computes the distance between itself and the new node it

points to. The steps are:

(i) set rank of each node to 1 except the rightmost node is set to 0.

(ii) then at each step (ii) we modify Rank of i^{th} item as.

Rank[i] += Rank[Neighbour[i]]

(iii) continue step (ii) until every node points to end of list.

Algorithm:

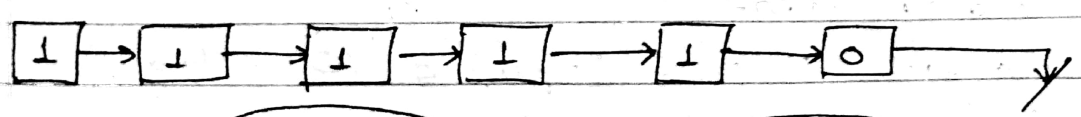
```
for (int n = 1 ; n <= [log n] ; q ++)

    Processer i (in parallel for 1 ≤ i ≤ n) do:
        if (Neighbour [i] ≠ 0) {
            Rank [i] += Rank [Neighbour [i]];
            Neighbour [i] += Neighbour [Neighbour [i]];
        }
```

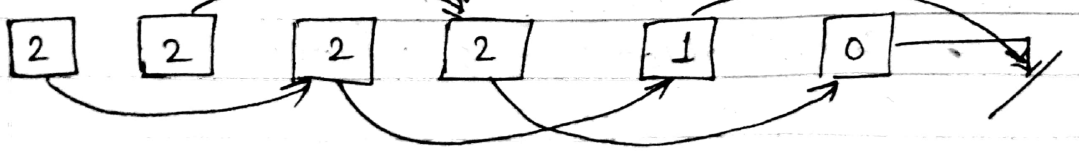
Example: given the linked list compute rank:

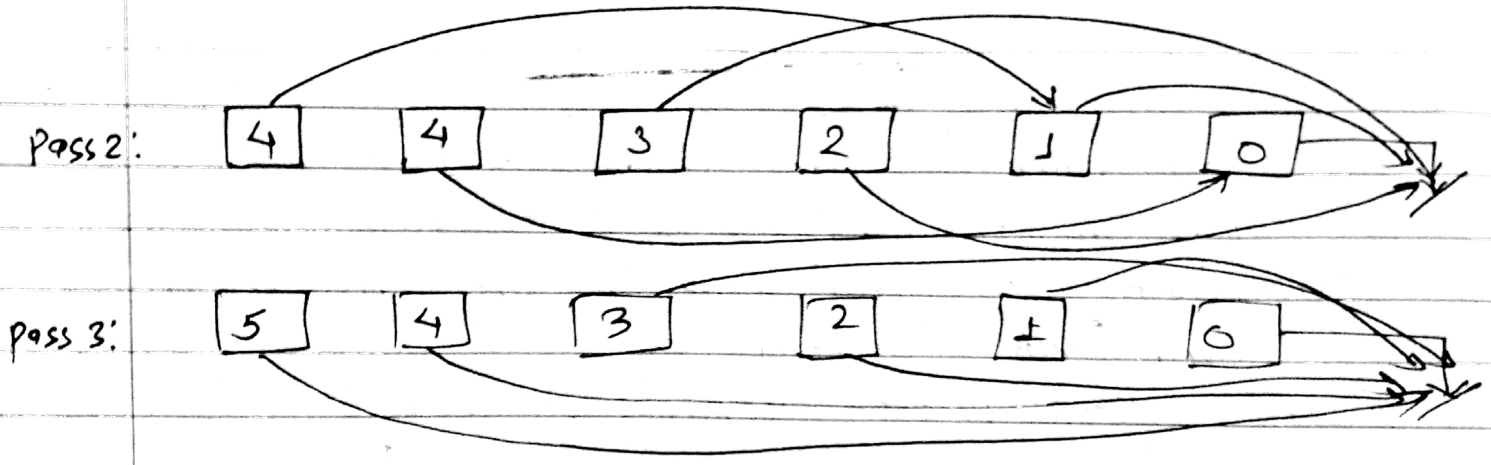
A[2] → A[6] → A[5] → A[4] → A[1] → A[3]

Then: Initially:



Pass1:





After logn iterations we have the
(log n = 6)
(n = 3)
respective rank and,

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

Analysis:

- Total running time : O(log n)
- Total work n * O(log n) = O(n log n)
- Sequential algorithm : O(n)
- Speed up = O(n)/O(log n) = O(n/log n)
- Efficiency = O(n) / O(n log n) = 1/log n

Is not work optimal.

Q.15: Explain data concentration in butterfly network with an example.

- In a p-processor interconnection network, there can be 'd' data items such that $d < p$, distributed arbitrarily with at most one data item per processor.

- The problem of data concentration is to move the data into the first 'd' processors of the network. The problem is also known as packing.

- Consider a butterfly network B_d, on which $k < 2^d$ data items are arbitarily distributed in level d of B_d.

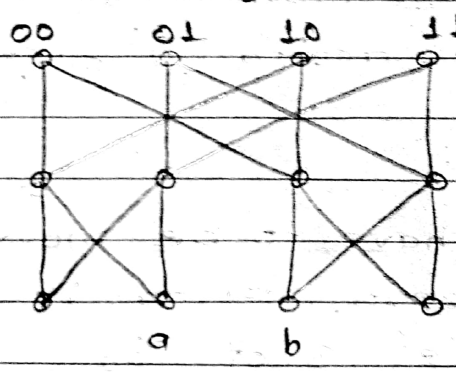
- All these data items have to be moved to successive rows of level zero. For eg. if there are five items in level 3 of B_3,
  001 → a, 010 → b 100 → c 110 → d 111 → e
  then at the end these items will be in level zero at row 000 → a 001 → b 010 → c 011 → d 100 → e

- Data concentration in any network is obtained using two phase algorithm.
  (i) In the first phase the prefix sum is computed to determine destination address of each data item.
  (ii) In the second phase data is routed to its destination from its origin.

- To compute prefix we can use binary tree and for routing we can use any approach discussed in the PPR algorithm.

example:
consider a butterfly network of depth 2 (B_2) with elements a & b at rows 01 and 10 respectively. Perform data concentration on the network.

Solution: The butterfly network for above scenario is,



Step 1: we compute the prefix sum to determine destination, To compute prefix sum we embed the butterfly network B_2 to binary network of depth 2

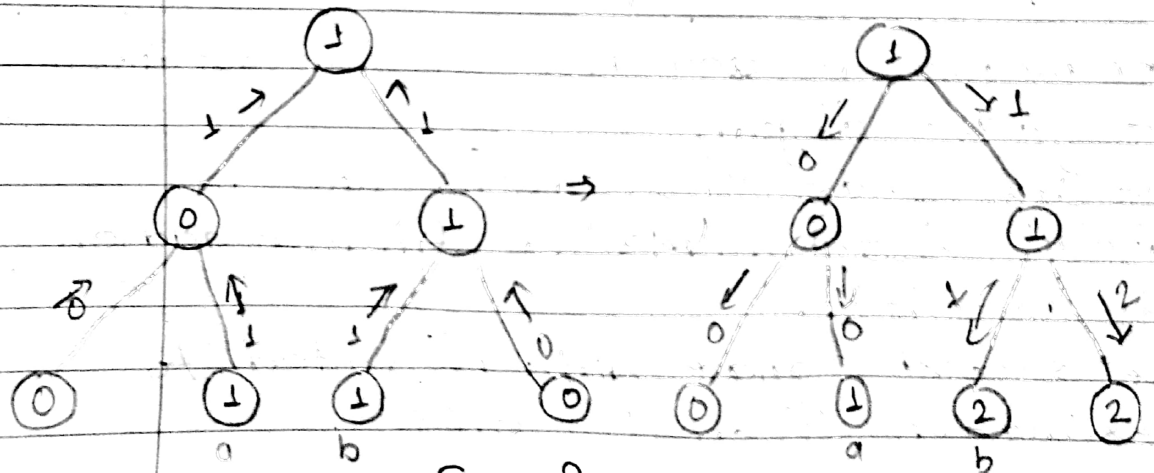
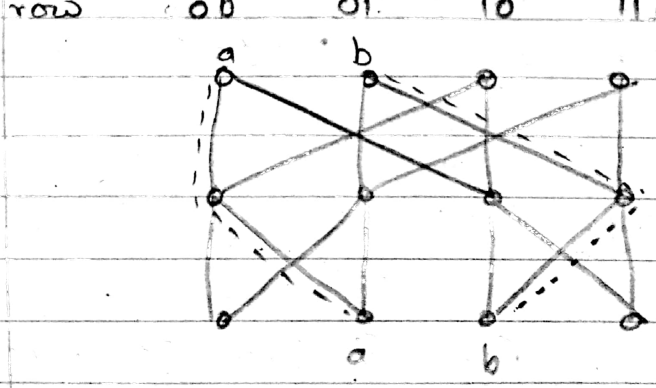


Fig: Prefix computation

=> Then the prefix is computed as {0, 1, 2, 2}. This implies the element a has to be moved to the first row i.e. row 000 & b has to be moved to row 001.

=> In the second phase, using the greedy approach the packets are routed to their destination. The path followed can be seen as.



Here, dotted line represents the path followed by each data and that is equal to level d i.e. 2.

Q.23: Explain the job sequencing with deadline problem and its solution using greedy approach.

- Given a set of dead jobs associated with their respective deadline and their profit, job sequencing problem is to find out the best jobs to perform within the deadline so as to maximize the profit.

Solution:

(i) Initially we order the jobs according to their profit $P_1 \geq P_2 \geq P_3 \ldots \geq P_n$ in decreasing order.

(ii) Pseudocode:

```
for i = 1 to n do,
    k = min(dmax, D[i]) // D[i] -> deadline of ith job
                        // dmax -> max value of deadline
while k >= 1 do,
    if timeslot[k] = empty then,
        timeslot[k] = job[i]
        break;
    end if
    k = k-1
end while
end for.
```

we, Initially plan to choose the job that provides the best profit hence this is an greedy approach.

Given the job, deadline & profit, choose the jobs so as to maximize the profit.

job	J_1	J_2	J_3	J_4	J_5
deadline	2	1	3	1	2
Profit	20	60	50	100	40

Step1: arrange jobs in ascending order of profit

Index	1	2	3	4	5
job	J_4	J_2	J_3	J_5	J_1
deadline	1	1	3	2	2
profit	100	60	50	40	20

then,

for i = 1 to 5

i = 1

k = min(dmax, d[i])

= min(3, d[1] = 1)

= 1

if timeslot[1] = empty => True

timeslot[1] = J[1] = J_4

break;

* Job J_4 is selected

i = 2

k = min(dmax, d[i])

= min(3, 1)

= 1

Since timeslot[1] is not empty

we break.

k = k - 1 = 1 - 1 = 0

exit;

i = 3	i = 4
k = min (3, 3) = 3	k = min (3, 2) = 2
Since timeslot[3] = empty timeslot[3] = J[3] = J_3 break	Since timeslot[2] = empty timeslot[2] = J[4] = J_5 break.
* Job J_3 is selected	* J_5 is selected.

Since all the slots are full we stop and the selected jobs that maximize the profit are:

Job	J_4	J_5	J_3
deadline	1	2	3
profit	100	40	50

and maximum profit = 100 + 40 + 50
= 190

Q. No. 24: Explain the ski-rental problem with necessary calculation:

Answer: The ski rental problem is defined as: Assume you are taking ski lessons, after each lesson you decide whether to continue to ski or to stop totally.

You have two choices of either renting ski's for 1 $ at a time or buy ski for y $.
The problem is to decide "will you buy or rent?"

Offline Strategy:

- if we knew in advance how many times 't' 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)

Problem:

- However, you never know how many times you will ski in prior.
- Then in online strategy, we choose a number k such that after ~~rent~~ renting k-1 times you will by skis (just before your k^{th} visit)
- we can claim that setting k = y guarantees that you never pay more than twice the cost of the offline strategy.
- example: assume y = 7 $, Then after 6 rents, you buy. Then total cost = 6 + 7 = 13 $

competitiveness:

Theorem: setting $k = y$ guarantees that you never pay more than twice the cost of the offline strategy.

Proof: when you buy skis in your k^{th} visit, even if you quit right after this time, $t \geq y$

- Your total payment is $k - 1 + y = 2y - 1$
- The offline cost is $\min(t, y) = y$
- The ratio is $(2y - 1)/y = 2 - \frac{1}{y}$

Hence, this strategy is $(2 - 1/y)$ competitive, every strategy is at least $(2 - 1/y)$ competitive.