APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY
Sixth Semester B.Tech Degree Examination June 2022 (2019 Scheme)
Course Code: CST306
Course Name: ALGORITHM ANALYSIS AND DESIGN
PART A

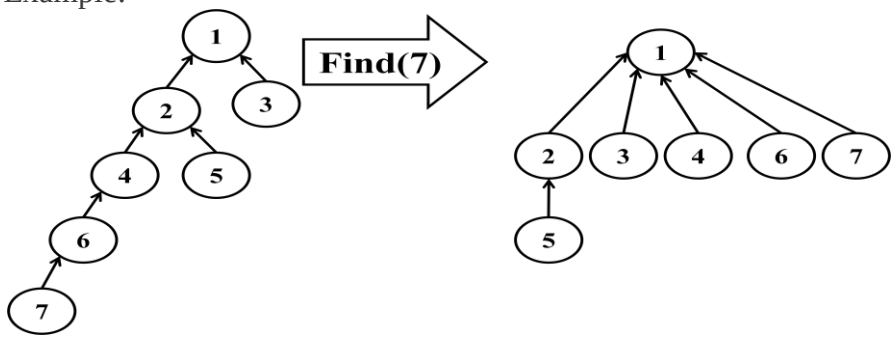
1. Let f(n)=3n^3+2n^2+3 for an algorithm, Let $g(n)=n^3$. Prove that f(n) of this algorithm is in $O(n^3)$
   Ans:

   $3n^3+2n^2+3 \leq 4\ n^3$ for all n≥3
   Here f(n)= $3n^3+2n^2+3$ g(n)= n^3 c=4 n_0=3
   Therefore $3n^3+2n^2+3$ = **$O(n^3)$**

2. Solve the recurrence T(n)=3T(n/4)+$nlogn$. using Master theorem
   Ans:

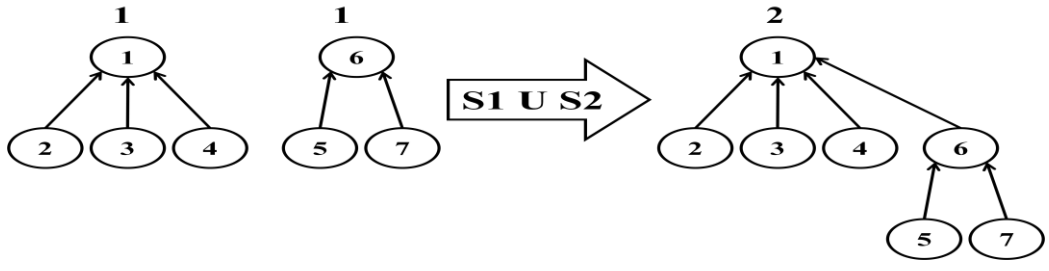
   a=3 b=4 $nlogn$ =Θ ($n^1 \log^1(n)$) k=1 p=1
   $b^k = 4^1 = 4$
   Here $a < b^k$ and p≥0
   $T(n) = \Theta(n^k \log^p(n))$
   $= \Theta(n^1 \log^1(n))$
   = **Θ(**n log(n)**)**

3. Discuss briefly the heuristics, union by rank and path compression, to improve the running time of disjoint set data structure
   Ans:
   - **Path Compression(collapsing rule)**
     - Path compression is a way of flattening the structure of the tree whenever Find is used on it.
     - Since each element visited on the way to a root is part of the same set, all of these visited elements can be reattached directly to the root.
     - Example:

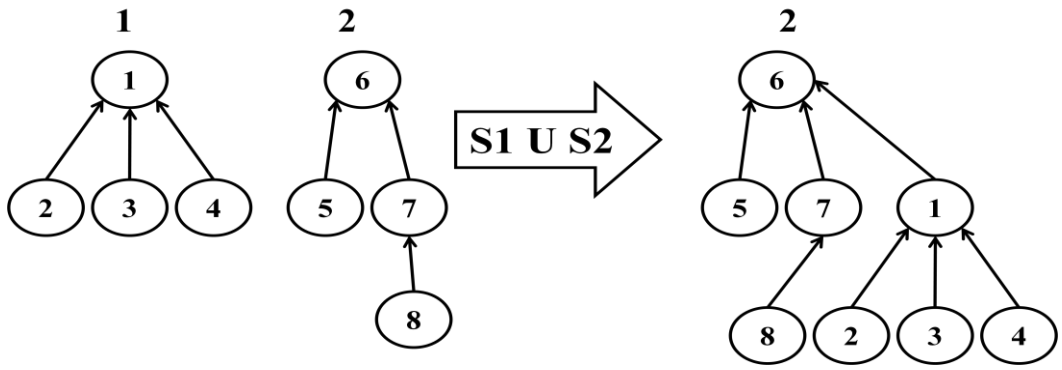


     - Next time we perform Find(6) it will give us the answer in two steps instead of four prior to the optimisation.

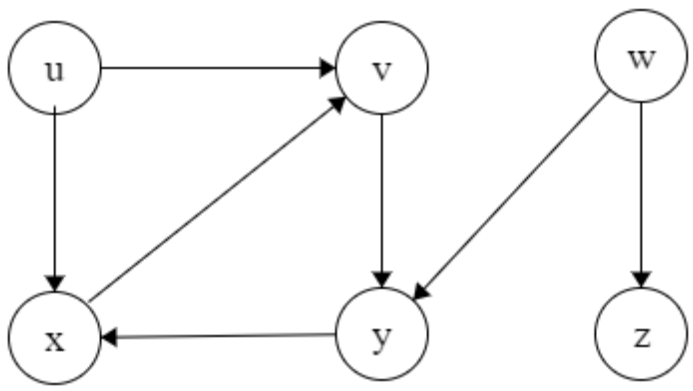
   - **Union by Rank(weighted rule)**
     - In this operation we decide which tree gets attached to which.
     - The basic idea is to keep the depth of the tree as small as possible.
     - We assign a new value **Rank** to each set. This rank represents the depth of the tree that the given set represents.
     - If we union two sets and :
       - Both sets have same rank → then the resulting set's rank is 1 larger.
       - Both sets have the different ranks — the resulting set's rank is the larger of the two.
     - Example 1:



- Example 2:



4. Consider the directed acyclic graph G=(V,E) given in the following figure.



Find any topological ordering of G.

Ans:

There is a cycle in the graph. So there is no topological ordering.

5. Give the control abstraction of Greedy strategy

Ans:

```
Greedy(a, n)   //a[1..n] contains n inputs
{
        solution = Φ;
        for i=1 to n do
        {
                x = Select(a);
                if Feasible(solution, x) then
                        solution = Union(solution, x);
        }
        return solution;
}
```

6. Why Strassen's matrix multiplication algorithm is better than traditional divide and conquer algorithm for multilpying two square matrices? What is the recurrence for the number of computational steps taken by Strassen's algorithm and its time complexity?

Ans:

Divide and Conquer Matrix multiplication

- For multiplying two matrices of size n x n using divide and conquer matrix multiplication strategy, we make 8 recursive calls, each on a matrix with size n/2 x n/2.
- Addition of two matrices take $O(n^2)$ time.
- Time complexity of divide and conquer matrix multiplication= $8\ T(n/2) + O(n^2)$
  $= \mathbf{O(n^3)}$
- For multiplying two matrices of size n x n using strassen's matrix multiplication strategy, we make 7 matrix multiplications and 10 matrix additions and subtractions
- Addition/Subtraction of two matrices takes $O(n^2)$ time.

- Time complexity = $7\ T(n/2) + O(n^2)$ = $O(n^{\log 7})$ = **$O(n^{2.81})$**

- Therefore strassen's matrix multiplication strategy is better than traditional divide and conquer matrix multiplication strategy.

- The recurrence for strassen's matrix multiplication strategy is

$$\mathbf{T(n)} = \begin{cases} \mathbf{b} & \mathbf{if\ n<2} \\ \mathbf{7\ T(n/2) + c\ n^2} & \mathbf{Otherwise} \end{cases}$$

7. Discuss briefly the elements of dynamic programming with a suitable example

Ans:

Dynamic programming is a technique used in computer science and mathematics to solve problems by breaking them down into simpler subproblems and solving each subproblem just once, storing the results to avoid redundant computations. The key elements of dynamic programming include:

1. Optimal Substructure: Here the problem can be broken down into smaller subproblems, and the solution to the larger problem can be found by combining solutions to the smaller subproblems.

2. Overlapping Subproblems: This property means that the same subproblems are solved multiple times in the process of solving the larger problem. Dynamic programming avoids redundant calculations by storing the solutions to subproblems in a table and reusing them when needed.

3. Memoization or Tabulation: Dynamic programming can be implemented using either memoization or tabulation techniques.

- **Memoization** involves storing the results of solved subproblems in a data structure like an array or hash table. When a subproblem needs to be solved, its result is looked up in the table, and if it's already solved, the stored result is returned.
- **Tabulation** involves solving the subproblems iteratively and building up the solution from smaller subproblems until the larger problem is solved.

4. State Transition: This refers to defining how solutions to larger problems can be built from solutions to smaller subproblems. It involves defining the recurrence relation or formula that relates the solution to a problem with the solutions to its subproblems.

Example:

Consider the problem of finding the n-th Fibonacci number, where Fibonacci sequence is defined as follows: $F(0) = 0$, $F(1) = 1$, and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.
Dynamic programming can be applied to efficiently compute the n-th Fibonacci number using memoization or tabulation:

- Optimal Substructure: The Fibonacci sequence exhibits optimal substructure because the solution to the n-th Fibonacci number can be constructed from the solutions to the (n-1)-th and (n-2)-th Fibonacci numbers.

- Overlapping Subproblems: Without dynamic programming, the naive recursive solution to compute Fibonacci numbers would result in exponential time complexity due to redundant calculations. Dynamic programming avoids this by storing the results of solved subproblems.

- Memoization/Tabulation: Dynamic programming can be implemented using memoization (top-down approach) by storing the results of computed Fibonacci numbers in a table and looking up previously computed values to avoid redundant calculations. Tabulation (bottom-up approach) involves solving the subproblems iteratively and building up the solution from smaller subproblems until the larger problem is solved.

- State Transition: The state transition for computing Fibonacci numbers is straightforward: F(n) = F(n-1) + F(n-2), where F(n) depends on the solutions to the (n-1)-th and (n-2)-th Fibonacci numbers. This recurrence relation forms the basis for dynamic programming solution.

By applying dynamic programming techniques, the time complexity of computing Fibonacci numbers can be reduced from exponential to linear or linearithmic, making it much more efficient.

8. Compare backtracking and branch-and-bound design techniques

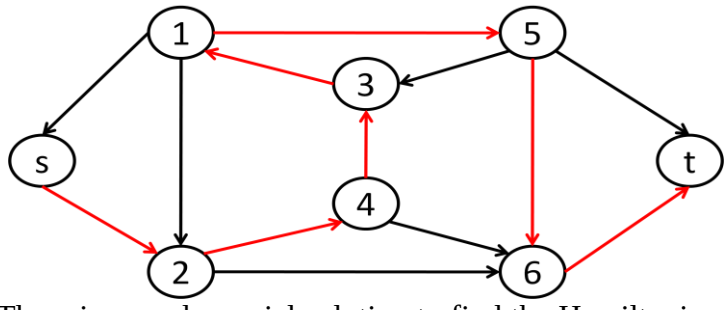
Ans:

Backtracking	Branch and Bound
Backtracking is a problem-solving technique so it solves the decision problem.	Branch n bound is a problem-solving technique so it solves the optimization problem.
Backtracking uses a Depth first search.	Branch and bound uses Depth first search/D Search/Least cost search.
In backtracking, all the possible solutions are tried. If the solution does not satisfy the constraint, then we backtrack and look for another solution.	In branch and bound, based on search; bounding values are calculated. According to the bounding values, we either stop there or extend.
Applications of backtracking are n-Queens problem, Sum of subset.	Applications of branch and bound are knapsack problem, travelling salesman problem, etc.
Backtracking is more efficient than the Branch and bound.	Branch n bound is less efficient.

9. Define P, NP and NP complete domains.

Ans:

- **Class P**
  - Class P consists of those problems that are solvable in polynomial time.
  - P problems can be solved in time $O(n^k)$. Here n is the size of input and k is some constant.
  - Example:
    - PATH Problem
- **Class NP**
  - Some problems can be solved in exponential or factorial time. Suppose these problems have no polynomial time solution. We can verify these problems in polynomial time. These are called NP problems.
  - NP is a class of problem that having only non-polynomial time algorithm and a polynomial time verifier.
  - Example:
    - **Hamiltonian path(HAMPATH) Problem**
      - A Hamiltonian path in a directed graph G is a directed path that goes through each node exactly once.



      - There is no polynomial solution to find the Hamiltonian path from s to t in a given graph. But there is a verification algorithm. So HAMPATh is a NP problem

  - **NP- Hard**
    - If a decision problem X is NP-Hard if every problem in NP is polynomial time reducible to X.

$$Y \leq_p X \qquad \text{for every Y in NP}$$

- It means that X is as hard as all problems in NP.
- If X can be solved in polynomial time, then all problems in NP can also solved in polynomial time.

▪ **Class NP-Complete**
  ▪ If the problem is NP as well as NP-Hard, then that problem is NP Complete.
  ▪ Example:
    - 3-CNF-SAT
    - CLIQUE problem
    - VERTEX COVER problem

10. Compare Las Vegas and Monte Carlo algorithms

Ans:

- **Randomized Las Vegas Algorithms**
  - Output is always correct and optimal.
  - Running time is a random number
  - Running time is not bounded
  - Example: Randomized Quick Sort
- **Randomized Monte Carlo Algorithms:**
  - May produce correct output with some probability
  - A Monte Carlo algorithm runs for a fixed number of steps. That is the running time is deterministic

    - **Example**: Finding an '*a*' in an array of *n* elements
      - **Input**: An array of $n \geq 2$ elements, in which half are '*a*'s and the other half are '*b*'s
      - **Output**: Find an '*a*' in the array

    - **Las Vegas algorithm**

```
Algorithm findingA_LV(A, n)
{
        repeat
        {
                Randomly choose one element out of n elements
        }until('a' is found)
}
```

- The expected number of trials before success is 2.
- Therefore the time complexity = O(1)

- **Monte Carlo algorithm**

```
Algorithm findingA_MC(A, n, k )
{
        i=0;
        repeat
        {
                Randomly select one element out of n elements
                i=i+1;
        }until(i=k or 'a' is found);
}
```

- This algorithm does not guarantee success, but the run time is bounded. The number of iterations is always less than or equal to k.
- Therefore the time complexity **= O(k)**

PART B

11.

a) Illustrate best case, average case and worst-case complexity with insertion sort algorithm.

Ans:

```
Algorithm InsertionSort(A,n)
{
        for i=1 to n-1 do
        {
                j=i
                while j>0 and A[j-1] > A[j] do
                {
                        Swap(A[j], A[j-1])
                        j=j-1
                }
        }
}
```

Best case complexity: Best case situation occurs when the array itself is in sorted order. In this case the while loop will not execute successfully. So the time complexity is proportional to number of times the for loop will execute. It will execute O(n) time.

The best case time complexity = **Ω(n)**

Worst case complexity: Worst case situation occurs when the array itself is in reverse sorted order. In this case the while loop will execute 1+2+3+ . .. .+(n-1)=n(n+1)/2 times.

The time complexity is proportional to number of times the while loop will execute. It will execute $O(n^2)$ time.

The worst case time complexity = **$O(n^2)$**

Average case complexity: Average case situation occurs when the while loop will iterate half of its maximum iterations. In this case the while loop will execute [1+2+3+ . .. .+(n-1)]/2=n(n+1)/4 times.

The time complexity is proportional to number of times the while loop will execute. It will execute $O(n^2)$ time.

The average case time complexity = **$\Theta(n^2)$**

b) Give the general idea of the substitution method for solving recurrences. Solve the following recurrence using substitution method. $T(n)=2T(n/2)+n$

Ans:

Guess that T(n) = O(n log n)

As per O notation definition T(n) <= c n log n

By mathematical induction

If n=1, T(1) <= c x 1 x log 1 ➔ 1 <= 0 It is false

If n=2, T(2) <= c x 2 x log 2 ➔ 2 T(1) + 2 <= 2 c ➔ 4 <= 2c It is true

If n=3, T(3) <= c x 3 x log 3 ➔ 2 T(1) + 3 <= c x 3 x log 3

5 <= 3 c log 3 It is true

This relation is true when n= 2, 3, 4 . . . . ., k

T(k) <= c k log k, where 2<= k <= n

T(n/2) <= c (n/2) log (n/2) ——— (1)

The recurrence relation is

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

Apply equation number 1

T(n) <= 2 c (n/2) log (n/2) + n

<= c n log (n/2) + n

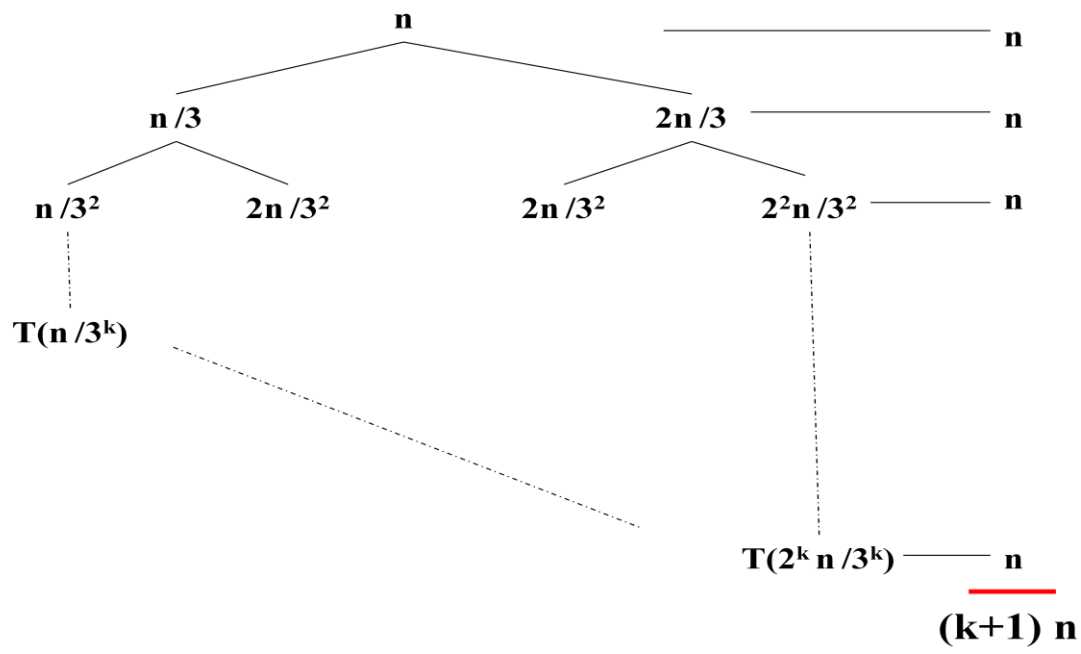
<= c n log n – c n log 2 + n

<= c n log n

Therefore **T(n) = O(n log n)**

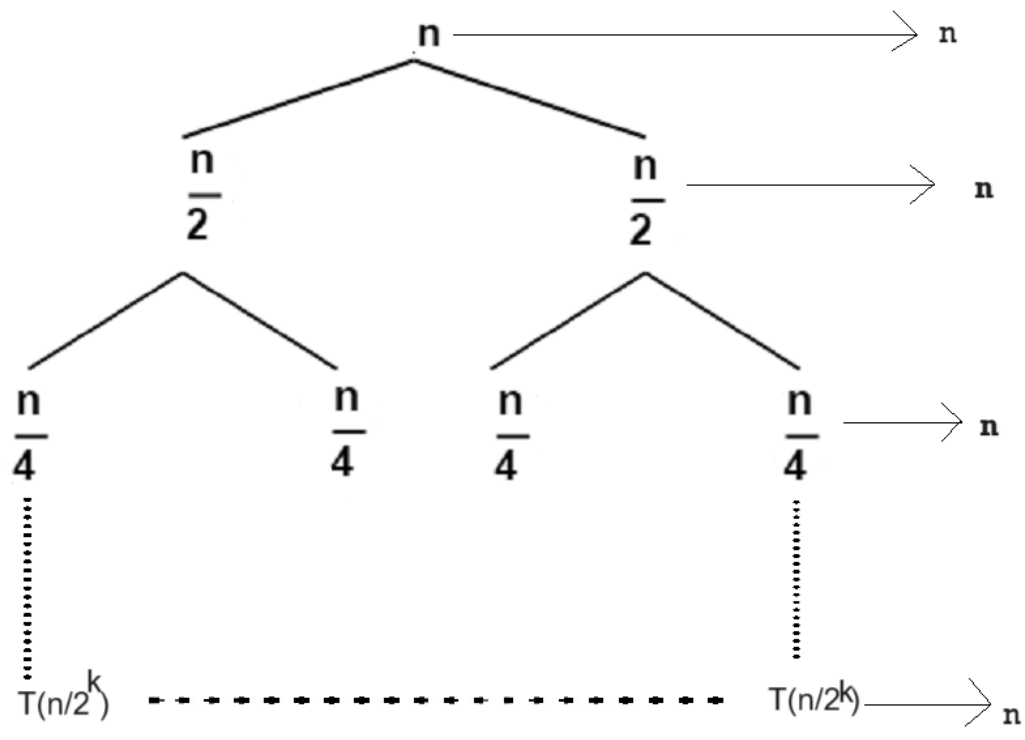
12.

a) Solve the following recurrence using recursion tree method
1) T(n)=T(n/3)+T(2n/3)+cn
2) T(n)=2T(n/2)+n
Ans:
1)



Assume that $2^k n/3^k = 1$ ➔ $(3/2)^k = n$ ➔ $k = \log_{(3/2)} n$
T(n) = (k+1) n
= ($\log_{(3/2)} n + 1$) n
= n $\log_{(3/2)} n$ + n
= **O(n $\log_{(3/2)}$ n)**

2)



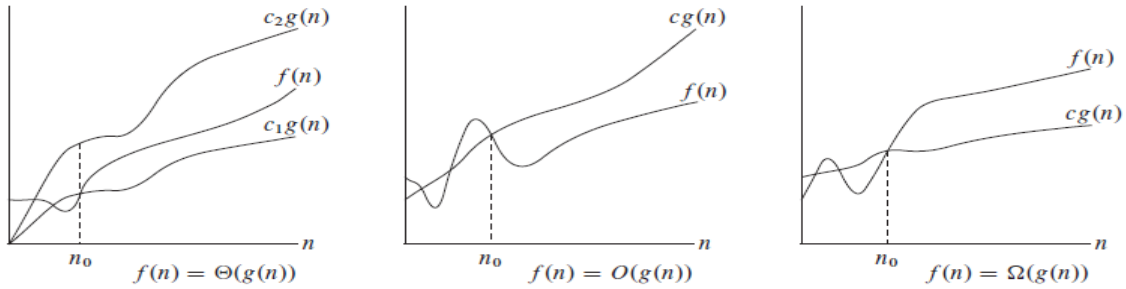
$n/2^k = 1$ $2^k = n$ $\boldsymbol{k = log_2(n)}$

T(n)= n + n + n + ……
=kn

= log_2n xn
= nlog_2n
= **O (nlog_2n)**

b) Define Big Oh, Big Omega and Theta notations and illustrate them graphically
Ans:

- **Asymptotic Notations**
  - It is the mathematical notations to represent frequency count.
  - **Big Oh (O)**
    - The function f(n) = O(g(n)) iff there exists 2 positive constants c and n_0 such that $0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$
    - It is the measure of longest amount of time taken by an algorithm(Worst case).
    - It is asymptotically tight upper bound
  - **Omega (Ω)**
    - The function f(n) = Ω (g(n)) iff there exists 2 positive constant c and n_0 such that $f(n) \geq c\ g(n) \geq 0$ for all $n \geq n_0$
    - It is the measure of smallest amount of time taken by an algorithm(Best case).
    - It is asymptotically tight lower bound
  - **Theta (Θ)**
    - The function f(n) = Θ (g(n)) iff there exists 3 positive constants c_1, c_2 and n_0 such that $0 \leq c_1\ g(n) \leq f(n) \leq c_2\ g(n)$ for all $n \geq n_0$
    - It is the measure of average amount of time taken by an algorithm(Average case).



13.

a) Give Breadth First Search algorithm for graph traversal. Perform its complexity analysis.
Ans:

Algorithm BFS(G, u)
1. Set all nodes are unvisited
2. Mark the starting vertex u as visited and put it into an empty Queue Q
3. While Q is not empty
   3.1 Dequeue v from Q
   3.2 While v has an unvisited neighbor w
      3.2.1 Mark w as visited
      3.2.2 Enqueue w into Q
4. If there is any unvisited node x
   4.1 Visit x and Insert it into Q. Goto step 3

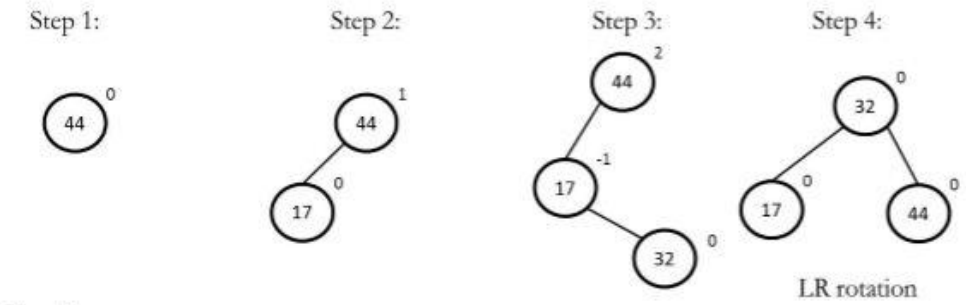
- **Complexity**
  - If the graph is represented as an adjacency list
    - Each vertex is enqueued and dequeued atmost once. Each queue operation take O(1) time. So the time devoted to the queue operation is **O(V).**

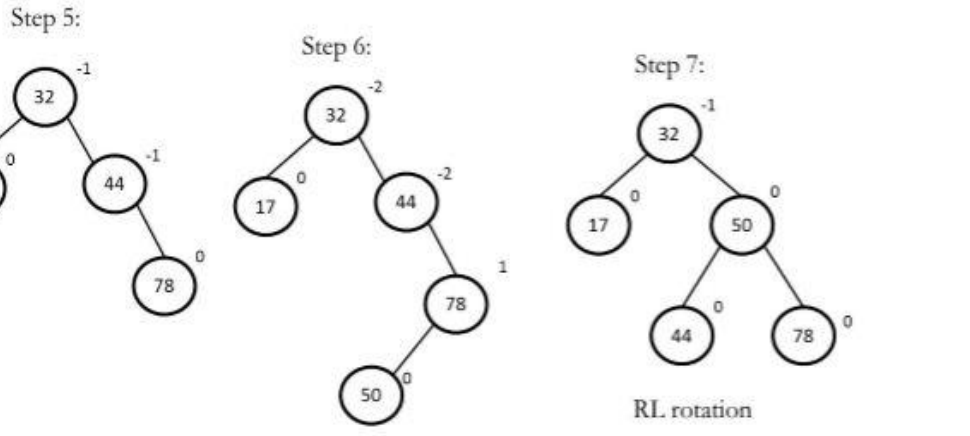
- The adjacency list of each vertex is scanned only when the vertex is dequeued. Each adjacency list is scanned atmost once. Sum of the lengths of all adjacency list is |E|. Total time spend in scanning adjacency list is **O(E).**
- Time complexity of BFS = O(V) + O(E) = **O(V+ E).**
- **In a dense graph:**
  - $E=O(V^2)$
  - Time complexity= $O(V) + O(V^2) = \mathbf{O(V^2)}$

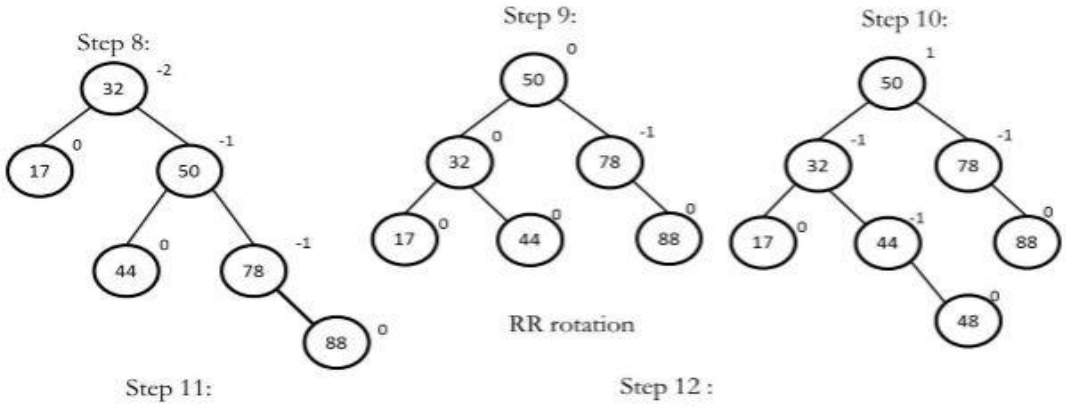
- If the graph is represented as an adjacency matrix
  - There are V^2 entries in the adjacency matrix. Each entry is checked once.
  - Time complexity of BFS = $\mathbf{O(V^2)}$

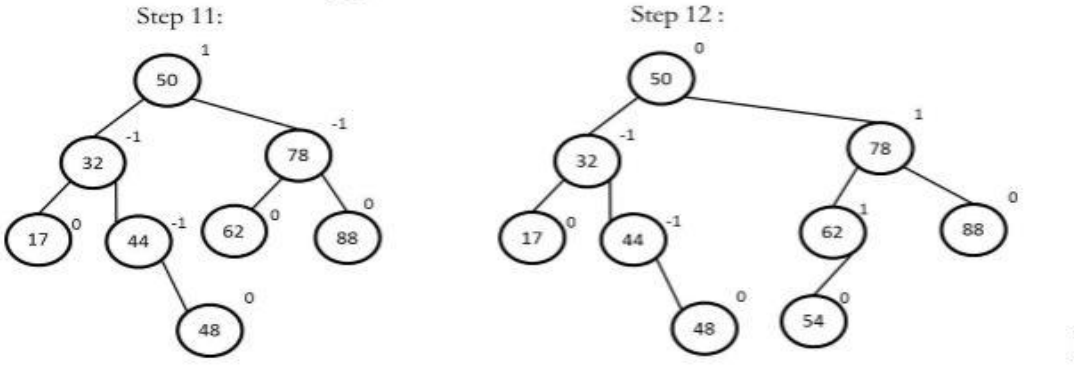
b) Define AVL tree. Construct an AVL tree by inserting the keys: 44, 17, 32, 78, 50, 88, 48, 62, 54 into an initially empty tree. Write clearly the type of rotation performed at the time of each insertion.

Ans:









14.

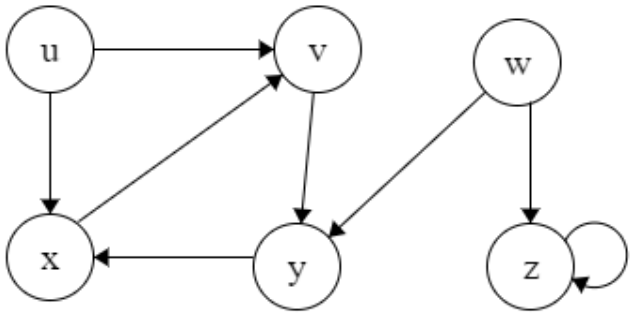
a) Give Depth First Serach algorith for graph traversal. Perform its time complexity analysis.
Ans:

Algorithm DFS(G, u)
1. Mark vertex u as visited
2. For each adjacent vertex v of u
   2.1 if v is not visited
      2.1.1 DFS(G, v)

Algorithm main(G,u)
1. Set all nodes are unvisited.
2. DFS(G, u)
3. For any node x which is not yet visited
   3.1 DFS(G, x)

- **Complexity**
  - If the graph is represented as an adjacency list
    - Each vertex is visited atmost once. So the time devoted is **O(V)**
    - Each adjacency list is scanned atmost once. So the time devoted is **O(E)**
    - Time complexity of DFS = **O(V + E)**.
  - If the graph is represented as an adjacency matrix
    - There are V^2 entries in the adjacency matrix. Each entry is checked once.
    - Time complexity of DFS = **$O(V^2)$**

b) Perform DFS traversal on the following graph starting from node A. When multiple nodes are available for next traversal choose nodes in alphabetical order. Classify the edges of the graph into different category.



Ans:

The DFS traversal of the above graph :u v y x w z

Tree Edge: It is an edge in tree obtained after applying DFS on the graph.

Eg.(u,v),(v,y),(y,x),(w,z)

Forward Edge: It is an edge(u,v) such that v is descendant but not part of the DFS tree.

Eg.(u,x)

Backward Edge: It is an edge(u,v) such that v is ancestor of edge u but not part of theDFS tree.

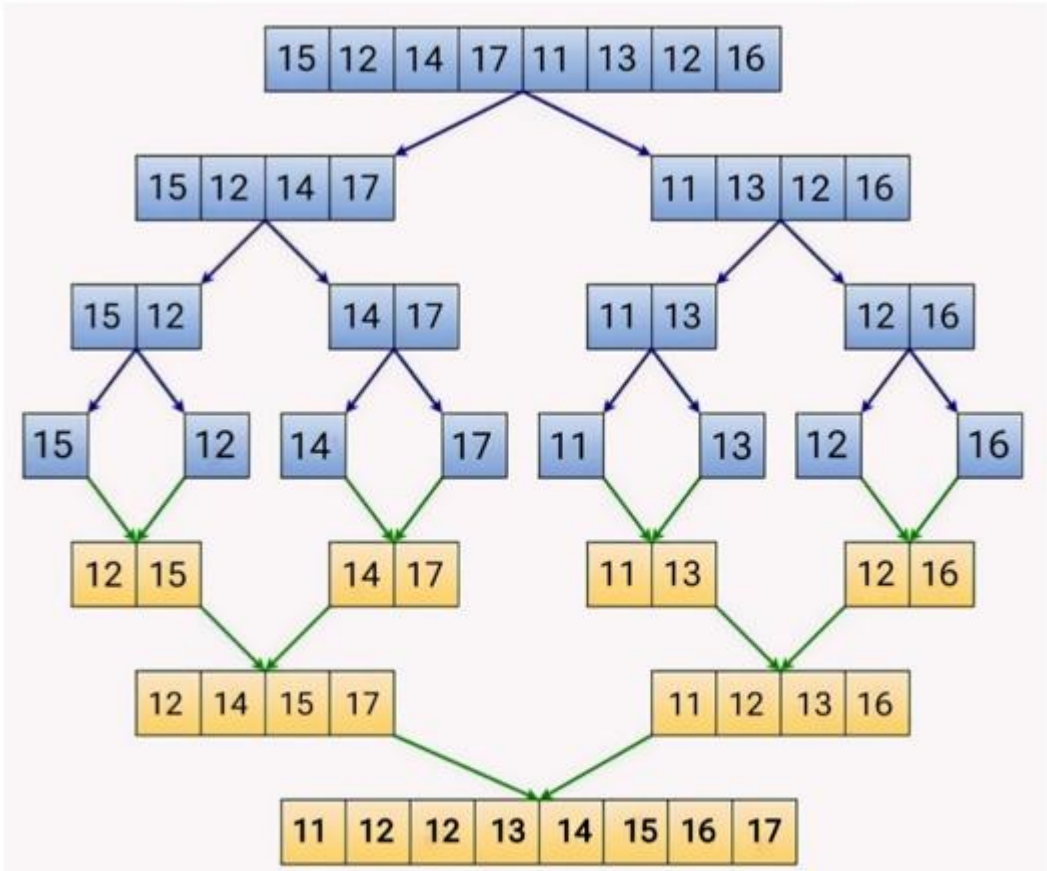
Eg. (x,v)

Cross Edge:It is an egde which connects two node such that they do not have any ancestor and descendant relationship between them.

Eg.(w,y)

15.

a) Illustrate the divide and conquer approach by applying 2 way merge sort for the input array: [15,12,14,17,11,13,12,16]. Write the recurrence for merge sort and give the complexity.
Ans:



The sorted array using the two-way merge sort approach is: [11, 12, 12, 13, 14, 15, 16, 17].

Recurrence Relation of Merge Sort

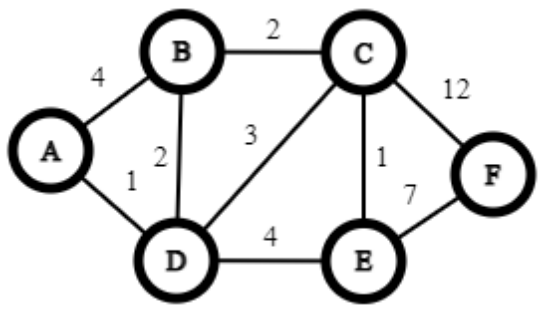
$$T(n) = \begin{cases} a & \text{if } n=1 \\ 2\,T(n/2) + cn & \text{Otherwise} \end{cases}$$

a is the time to sort an array of size 1
cn is the time to merge two sub-arrays
2 T(n/2) is the complexity of two recursion calls

$T(n) = 2\,T(n/2) + c\,n$
$= 2(2\,T(n/4)+c(n/2)) + c\,n$
$= 2^2T(n/2^2) + 2\,c\,n$
$= 2^3T(n/2^3) + 3\,c\,n$
. . . . . .
$= 2^kT(n/2^k) + k\,c\,n$ [Assume that $2^k = n \Rightarrow k=\log n$]
$= n\,T(1) + c\,n \log n$
$= a\,n + c\,n \log n$
$=$ **O(n log n)**

Best Case, Average Case and Worst Case Complexity of Merge Sort = **O(n log n)**

b) Apply Dijkstra's algorithm for single source shortest path to solve the following graph. Assume the source as node A.



Ans:

	A	B	C	D	E	F
A	0	∞	∞	∞	∞	∞
D		4	∞	1	∞	∞
B		3	4		5	∞
C			4		5	∞
E					5	16
F						12
		D	D	A	D	E

PATH	SHORTEST PATH	SHORTEST DISTANCE
A → B	A → D → B	3
A → C	A → D → C	4
A → D	A → D	1
A → E	A → D → E	5
A → F	A → D → E → F	12

16.

a) Consider the following instance of Fractional Knapsack problem with 3 objects. The capacity of the knapsack is 20 units. The weights and profits of the 3 items respectively are represented by the vectors (w1,w2,w3) = (18,15,10) and (p1,p2,p3)=(25,24,15). Using a greedy strategy compute the optimal solution to this instance.

Ans:

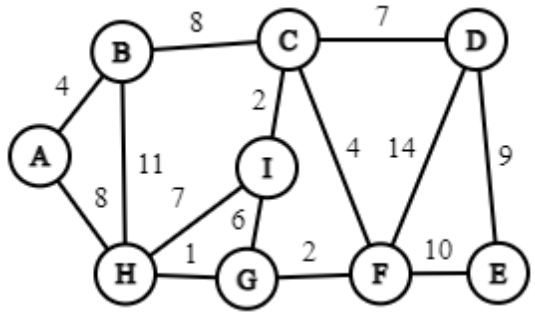
m = 20
n = 3
i ➔ { 1, 2, 3 }
P = {25, 24, 15 }
W = {18, 15, 10 }
P/W= {1.39, 1.6, 1.5 }

Sort p[i]/w[i] ≥ p[i+1]/w[i+1]
i ➔ { 2, 3, 1 }
P = {24, 15, 25 }
W= {15, 10, 18 }

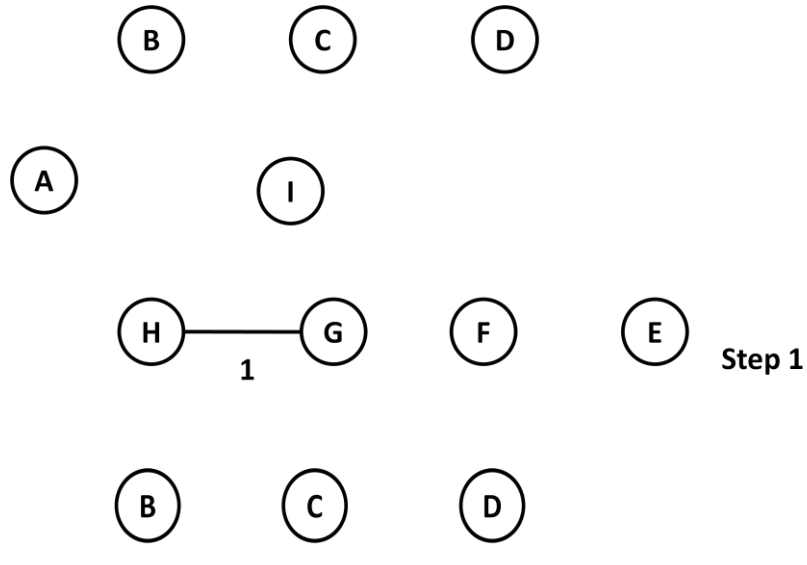
Item	**Pi**	**Wi**	**Xi**	**U = U-Wi**
2	24	15	1	5
3	15	10	5/10=1/2	0
1	25	18	0	

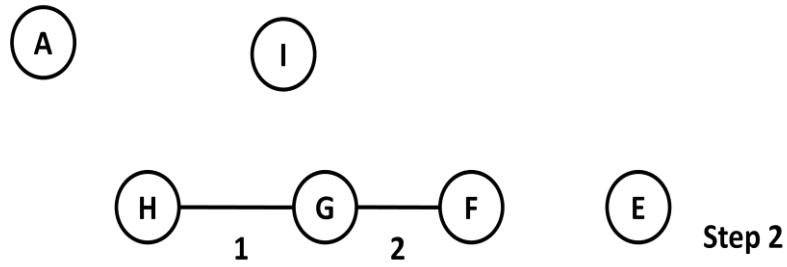
Total Profit = Σ Pi * Xi
=25x0 + 24x1 + 15x1/2 = **31.5**
Solution vector **X ={0,1,1/2}**

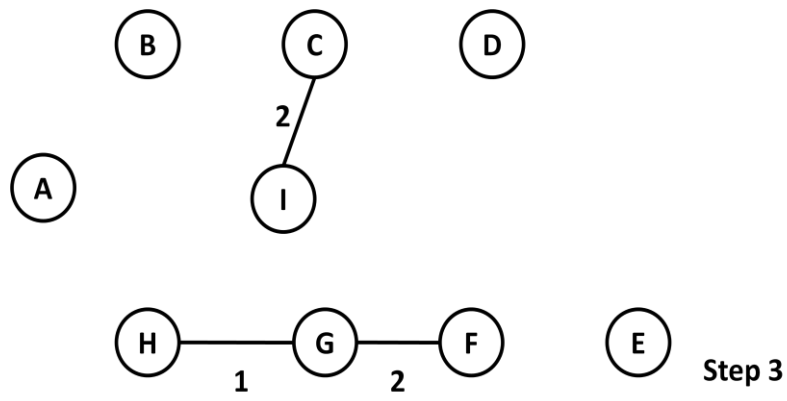
b) Apply Kruskal algorithm to find minimum cost spanning tree for the following graph

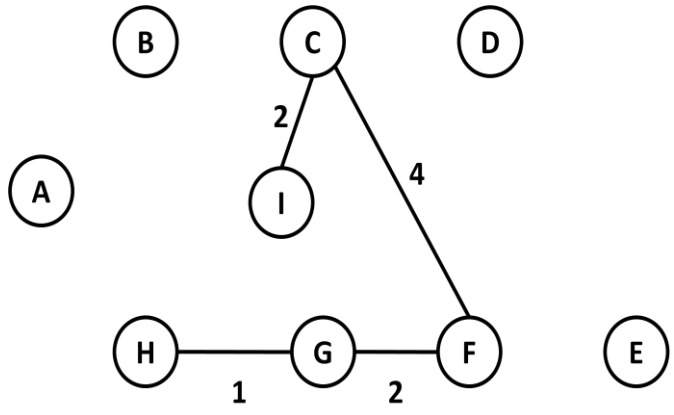


Ans:

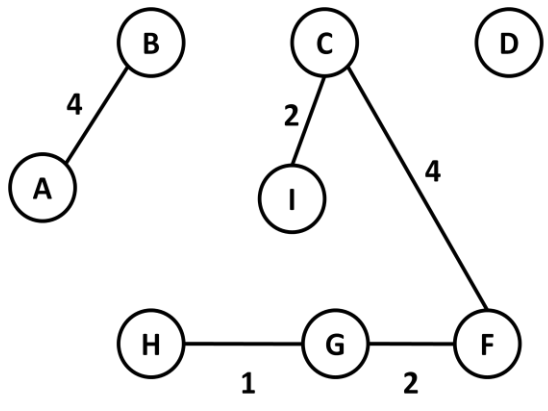




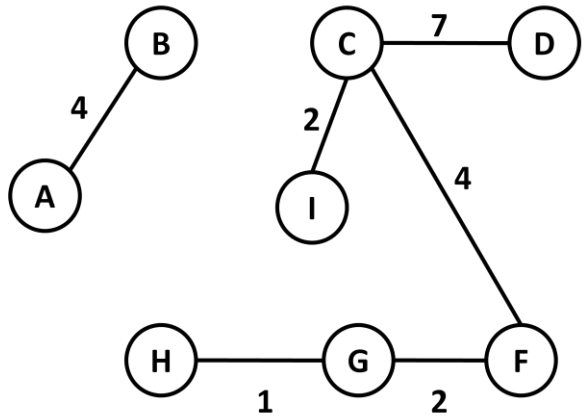




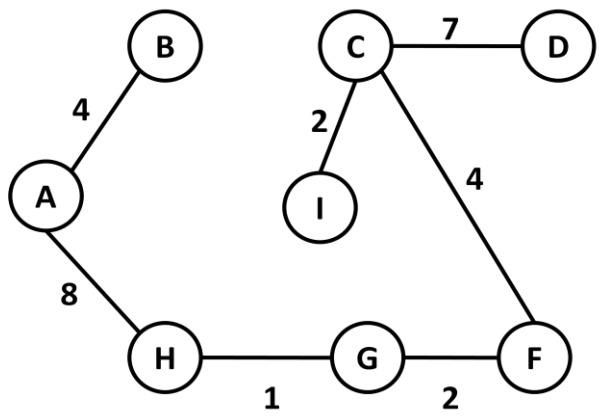
Step 4



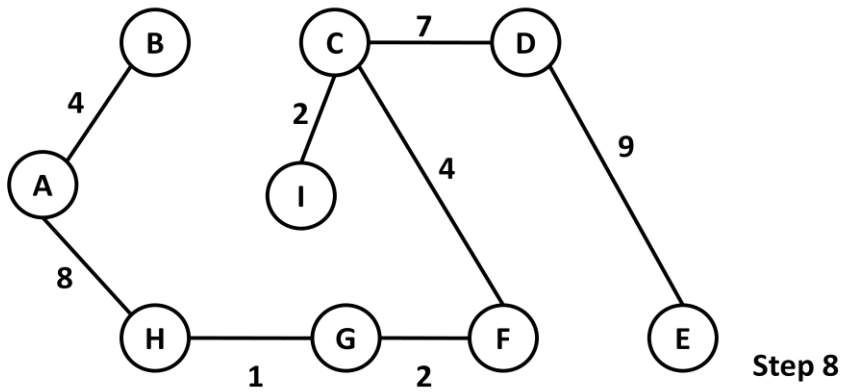
Step 5



Step 6



Step 7

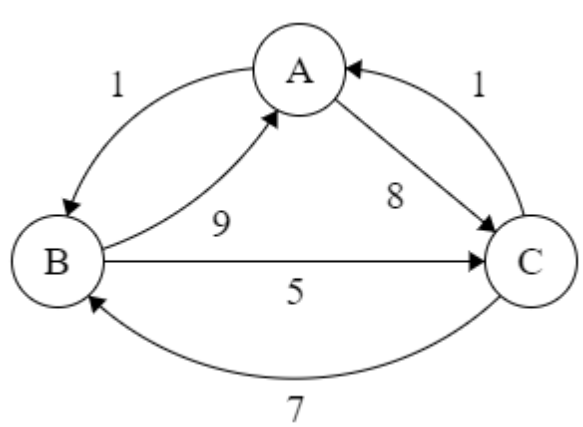


Step 8

This is the minimum cost spanning tree and its cost = 37

17.

a) Discuss Floyd-Warshall algorithm for all pair shortest path problem. Solve the following instance using the algorithm



Ans:

Initially, the adjacency matrix is

$$\mathbf{D^0} = \begin{pmatrix} 0 & 1 & 8 \\ 9 & 0 & 5 \\ 1 & 7 & 0 \end{pmatrix}$$

Take node A as intermediate node

Keep the 1st row and 1st column and diagonal elements of D_0 assuch

$D_1(2,3) = \min\{ D_0(2,3), D_0(2,1) + D_0(1,3) \} = \min\{ 5, 9+8 \} = 5$

$D_1(3,2) = \min\{ D_0(3,2), D_0(3,1) + D_0(1,2) \} = \min\{ 7, 1+1 \} = 2$

$$\mathbf{D^1} = \begin{pmatrix} 0 & 1 & 8 \\ 9 & 0 & 5 \\ 1 & 2 & 0 \end{pmatrix}$$

Take node B as intermediate node

Keep the 2nd row and 2nd column and diagonal elements of D_1 assuch

$D_2(1,3) = \min\{ D_1(1,3), D_1(1,2) + D_1(2,3) \} = \min\{ 8, 1+5 \} = 6$

$D_2(3,1) = \min\{ D_1(3,1), D_1(3,2) + D_1(2,1) \} = \min\{ 1, 2+9 \} = 1$

$$\mathbf{D^2} = \begin{pmatrix} 0 & 1 & 6 \\ 9 & 0 & 5 \\ 1 & 2 & 0 \end{pmatrix}$$

Take node C as intermediate node

Keep the 3rd row and 3rd column and diagonal elements of D_2 assuch

$D_3(1,2) = \min\{ D_2(1,2), D_2(1,3) + D_2(3,2) \} = \min\{ 1, 6+2 \} = 1$

$D_3(2,1) = \min\{ D_2(2,1), D_2(2,3) + D_2(3,1) \} = \min\{ 9, 5+1 \} = 6$

$$\mathbf{D^3} = \begin{pmatrix} 0 & 1 & 6 \\ 6 & 0 & 5 \\ 1 & 2 & 0 \end{pmatrix}$$

This matrix shows the shortest distance between every pair of vertice

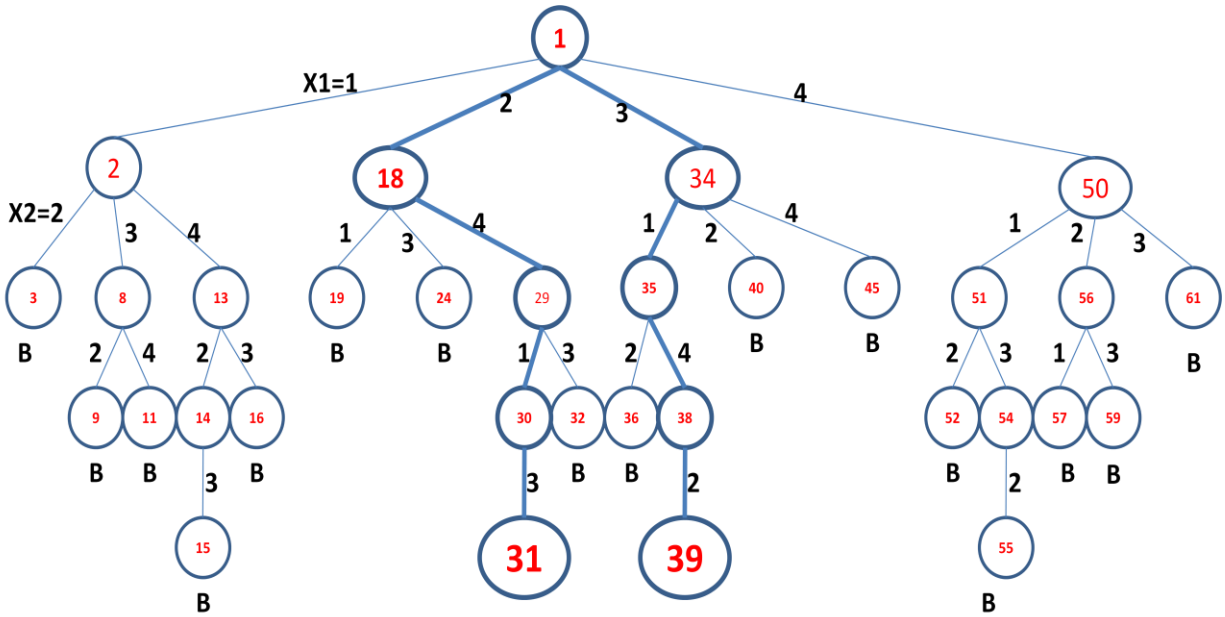
b) Discuss the control abstraction used in backtracking design technique. Draw the state space tree for 4-queens problem.

Ans:

Backtracking Control Abstraction

- $(x_1, x_2.,\ldots\ldots x_i)$ be a path from the root to a node in a state space tree.
- **Generating Function** $T(x_1, x_2.,\ldots\ldots x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2.,\ldots\ldots x_{i+1})$ is also a path to a problem state.
- $T(x_1, x_2.,\ldots\ldots x_n) = \varphi$
- **Bounding function** $B_{i+1}(x_1, x_2.,\ldots\ldots x_{i+1})$ is false for a path $(x_1, x_2.,\ldots\ldots x_{i+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node.
- Thus the candidates for position i+1of the solution vector $(x_1, x_2.,\ldots\ldots x_n)$ are those values which are generated by T and satisfy B_{i+1}.
- The recursive version is initially invoked by **Backtrack(1)**.

```
Algorithm Backtrack(k)
{       for (each x[k] ϵ T(x[1], . . . .x[k-1])
        {
            if(Bk(x[1], x[2], . . . . . ., x[k]) != 0) then
            {
                if(x[1], x[2], . . . . . ., x[k] is a path to an answer node)
                        then write(x[1:k])
                if(k<n) then Backtrack(k+1)
            }
        }
}
```



State Space Tree of 4 Queens Problem

18.

a) Discuss the elements of dynamic programming by considering the matrix chain multiplication problem.

Ans:

- Step 1: The structure of an optimal parenthesization
  - $A_{i..j}$ denote the matrix that results from evaluating the product $A_iA_{i+1} \ldots A_j$ where i<= j
  - If i< j, we must split the problem into two subproblems ($A_i A_{i+1} \ldots A_k$ and $A_{k+1}A_{i+1} \ldots A_j$), for some integer k in the range i<= k < j.
  - That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$. Then multiply them together to produce the final product $A_{i..j}$.
  - Total cost = Cost of computing the matrix $A_{i..k}$+ Cost of computing $A_{k+1..j}$+ Cost of multiplying them together.

- Step 2: A recursive solution
  - We can define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems.
  - Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$
  - For the full problem, the lowest cost way to compute $A_{1..n}$ would thus be m[1, n]
  - $A_{i..i} = A_i$ so m[i,i] = 0 for i=1,2,. .. . n

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

  - s[i,j] be a value of k at which we split the product $A_iA_{i+1} \ldots A_j$ in an optimal parenthesization.
  - This will take exponential time

- Step 3: Computing the optimal costs
  - Compute the optimal cost by using a tabular, bottom-up approach

Algorithm Matrix_Chain_Order(p)

```
{       n = p.length – 1
        Let m[1..n, 1..n] and s[1..n-1 , 2..n] be new tables
        for i=1 to n do
                m[i,i] = 0
        for l=2 to n do
        {       for i=1 to n-l+1 do
                {       j=i+l-1
                        m[i,j] = α
                        for k=i to j-1 do
                        {       q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
                                if q<m[i,j] then
                                {       m[i,j] = q
```

```
                                    s[i,j] = k
                                }
                            }

                        }
                    }
                    return m[][] and s[][]
                }
```

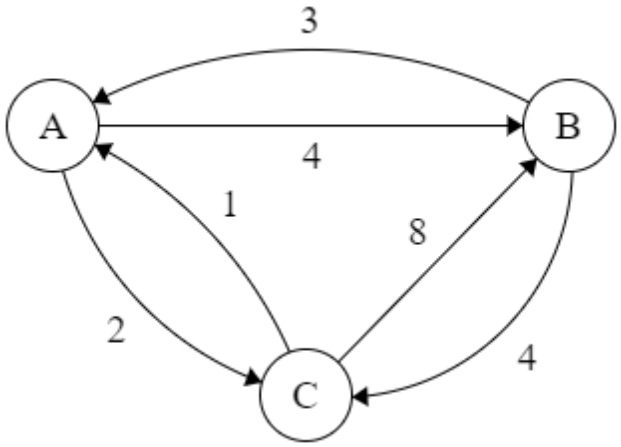
- Matrix A_i has dimensions p_{i-1} X p_i for i =1,2, . . . n.
- Input to this algorithm is a sequence p = (p_0, p_1, ... p_n), where p.length = n + 1.
- The procedure uses 2 auxiliary tables
- m[1 .. n , 1...n] for storing the cost of matrix multiplication
- s[1..n-1 , 2 ...n] records which index of k achieved the optimal cost in computing m[i,j] .

- Step 4: Constructing an optimal solution

```
Algorithm Print_Optimal_Parens(s,i,j)
{
        if i==j then
                print "A"i
        else
                print "("
                print_Optimal_Parens(s,i,s[i,j])
                print_Optimal_Parens(s,s[i,j]+1,j)
                print ")"
}
```

- Initial call is PRINT-OPTIMAL-PARENS(s,1,n)

b) Define Travelling Salesman Problem (TSP). Apply branch and bound technique to solve the following instance of TSP. Assume that the starting vertex as A. Draw the state space tree for each step.



Ans:

The adjacency matrix is

$$\begin{array}{c} \\ A \\ B \\ C \end{array}\begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} \infty & 4 & 2 \\ 3 & \infty & 4 \\ 1 & 8 & \infty \end{bmatrix} \end{array}$$

Perform row reduction, then column reduction

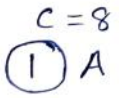
$$\begin{bmatrix} \infty & 4 & 2 \\ 3 & \infty & 4 \\ 1 & 8 & \infty \end{bmatrix} \begin{matrix} -2 \\ -3 \\ -1 \end{matrix} \xrightarrow{\text{row reduction}} \begin{bmatrix} \infty & 2 & 0 \\ 0 & \infty & 1 \\ 0 & 7 & \infty \end{bmatrix} \xrightarrow{\text{column reductn}} \begin{bmatrix} \infty & 0 & 0 \\ 0 & \infty & 1 \\ 0 & 5 & \infty \end{bmatrix}$$

$0 \quad -2 \quad 0$

M_1

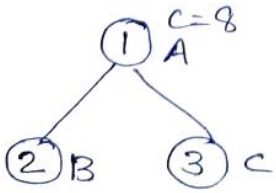
Total cost reduced $= 2+3+1+2 = 8$

The state space tree is



M_1 is the matrix for node 1

Generate the child node of node 1



Find the matrix and cost of node 2

* set row A and column B elemts are ∞
* Set $M_1[B,A] = \infty$
* The resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & 1 \\ 0 & \infty & \infty \end{bmatrix}$$

* Perform row reduction, then column reductio

$$\begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & 1 \\ 0 & \infty & \infty \end{bmatrix} \xrightarrow[\text{redtn}]{\text{row}} \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & 0 \\ 0 & \infty & \infty \end{bmatrix} \xrightarrow[\text{redtn}]{\text{column}} \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & 0 \\ 0 & \infty & \infty \end{bmatrix}$$

$0 \quad 0 \quad 0$

M_2

Cost reduced $= r = 1$

M_2 is the matrix for node 2

cost of node 2 = cost of node 1 + $M_1[A,B]$ + r

$= 8 + 0 + 1 = 9$

Find the matrix and cost of node 3

* Set row A and column C elmts are ∞
* Set $M_1[C,A] = \infty$
* The resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ 0 & 5 & \infty \end{bmatrix}$$

* Perform row reduction and column reduction

$$\begin{bmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ 0 & 5 & \infty \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \xrightarrow{\text{row redtn}} \begin{bmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ 0 & 5 & \infty \end{bmatrix} \xrightarrow{\text{column redtn}} \begin{bmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ 0 & 0 & \infty \end{bmatrix} = M_3$$

$0 \quad -5 \quad 0$

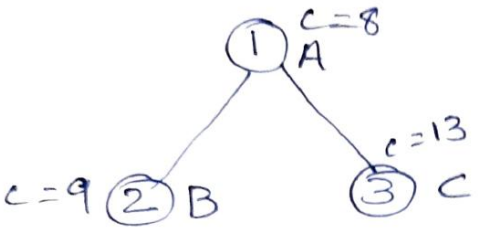
Cost reduced $= r = 5$

M_3 is the matrix for node 3

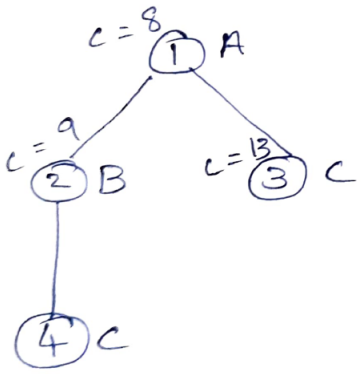
cost of node 3 = cost of node 1 + $M_1[A,C]$ + r

$= 8 + 0 + 5 = 13$

Now the state space tree is



Minimum cost live node is node 2.
Now node 2 will be the E-node.
Generate the child node of node 2



Find the matrix and cost of node 4

* Set row B and column C elmts are ∞
* Set $M_2[C, B] = \infty$
* The resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ 0 & \infty & \infty \end{bmatrix}$$

* Perform row reduction and column reduction

$$\begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ 0 & \infty & \infty \end{bmatrix} \xrightarrow[\text{redtn}]{\text{row}} \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ 0 & \infty & \infty \end{bmatrix} \xrightarrow[\text{redtn}]{\text{column}} \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ 0 & \infty & \infty \end{bmatrix}$$

M_4

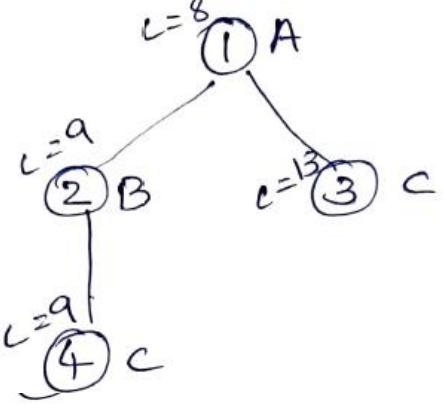
cost reduced = $r = 0$

M_4 is the matrix for node 4

cost of node 4 = cost of node 2 + $M_2[B, C]$ + r

$= 9 + 0 + 0 = 9$

Now the state space tree is



Now node 3 and 4 are the live nodes
Among them node 4 is the E-node.
This node having no children. So we can conclude that the TSP path is A-B-C-A.

TSP cost is the cost of node 4.

∴ TSP cost = 9

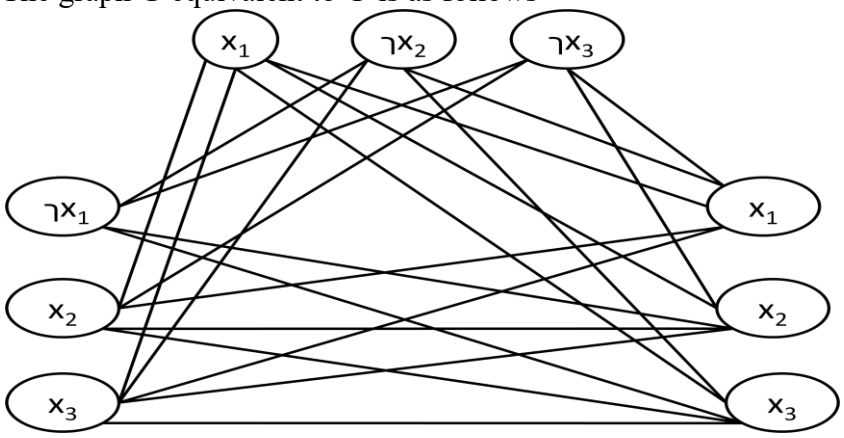
19.

a) Prove that CLIQUE problem is NP Complete
Ans:

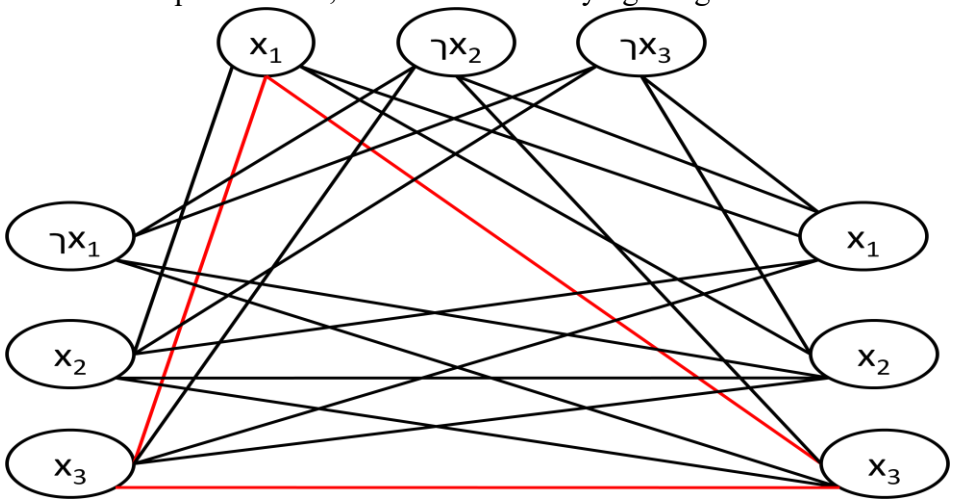
- **CLIQUE problem is NP Complete: Proof**
  - Step 1: Write a polynomial time verification algorithm to prove that the given problem is NP
    - Algorithm: Let G= (V,E), we use the set V' ⊆ V of k vertices in the clique as a certificate of G
      1. Test whether V' is a set of k vertices in the graph G
      2. Check whether for each pair (u,v) ∈ V', the edge (u,v) belongs to E.
      3. If both steps pass, then accept. Otherwise reject.
    - This algorithm will execute in polynomial time. Therefore **CLIQUE problem is a NP problem.**
  - Step 2: Write a polynomial time reduction algorithm from 3-CNF-SAT problem to CLIQUE problem(3-CNF-SAT ≤p CLIQUE)
    - Algorithm

- Let $\Phi = C_1 \wedge C_2 \ldots\ldots \wedge C_k$ be a Boolean formula in 3CNF with k clauses
- Each clause C_r has exactly three distinct literals l^r_1, l^r_2, l^r_3.
- Construct a graph G such that Φ is satisfiable iff G has a click of size k.
- The graph G is constructed as follows
  - For each clause $C_r = (l^r_1 \vee l^r_2 \vee l^r_3)$ in Φ, we place a triple of vertices V^r_1, V^r_2 and V^r_3 in to V.
  - Put an edge between V^r_i to V^s_i if following two conditions hold
    - V^r_i and V^s_i in different triples(that is r!=s)
    - l^r_i is not a negation of l^s_i.

- Example: $\Phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
  - The graph G equivalent to Φ is as follows



  - If G has a clique of size k, then Φ has a satisfying assignment. Here k=3.



  - G can easily be constructed from Φ in polynomial time.
  - So **CLIQUE problem is NP Hard**.
- Conclusion
  - CLIQUE problem is NP and NP Hard. So it is NP-Complete

b) Write randomized quicksort algorithm and perform its expected running time analysis.
Ans:

Algorithm randQuickSort(A[], low, high)

1. If low >= high, then EXIT
2. While pivot 'x' is not a Central Pivot.
   2.1. Choose uniformly at random a element from A[low..high]. Let the randomly picked element be x.
   2.2. Count elements in A[low..high] that are smaller than x. Let this count be sc.
   2.3. Count elements in A[low..high] that are greater than x. Let this count be gc.
   2.4. Let n = (high-low+1). If sc >= n/4 and gc >= n/4, then x is a central pivot.
3. Partition A[low..high] into two subarrays. The first subarray has all the elements of A that are less than x and the second subarray has all those that are greater than x. Now the index of x be pos.
4. randQuickSort(A, low, pos-1)
5. randQuickSort(A, pos+1, high)

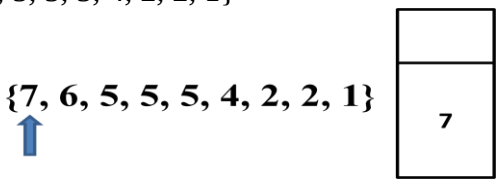
Number times while loop runs before finding a central pivot?

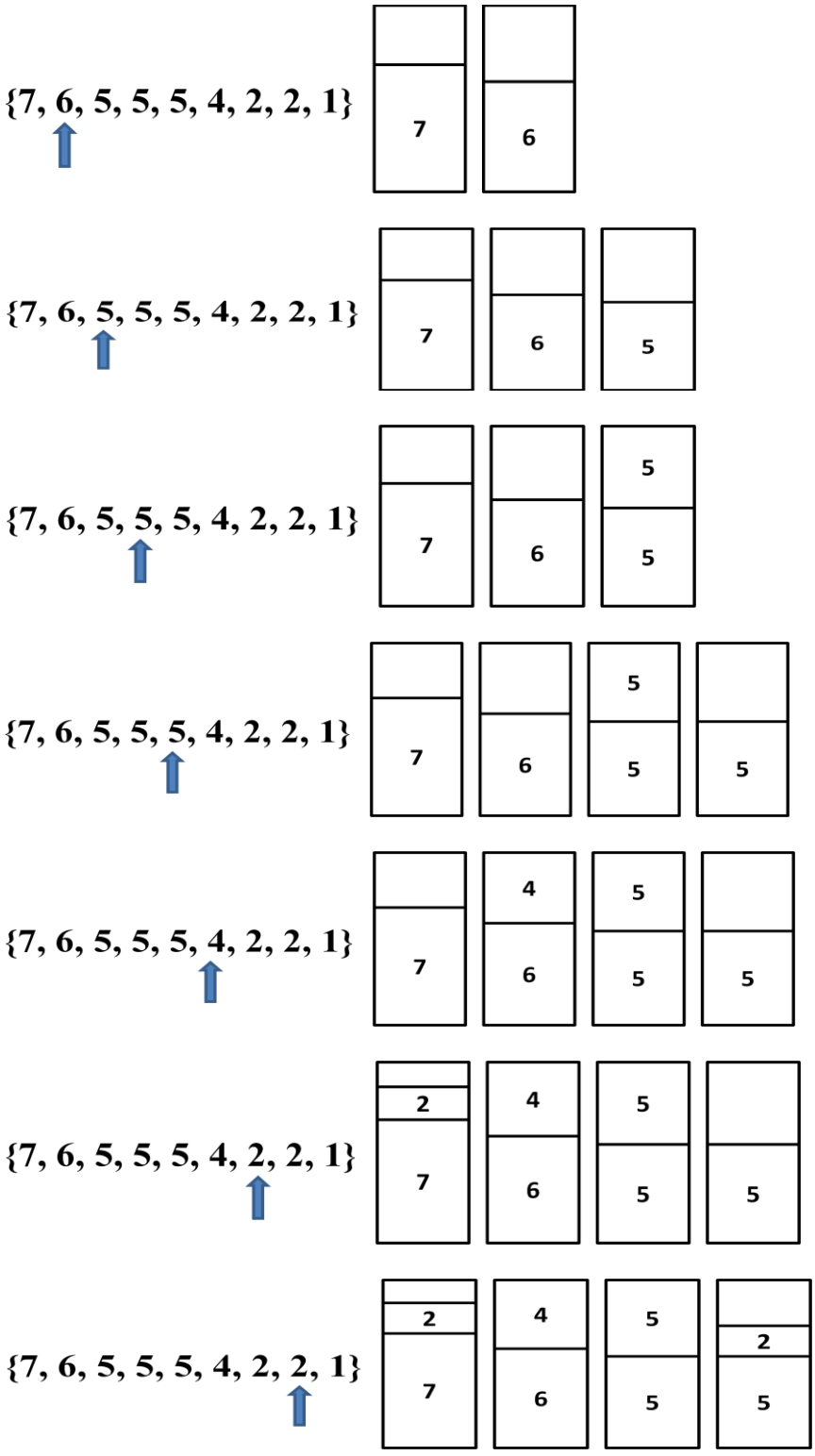
- The probability that the randomly chosen element is central pivot is 1/n.
- Therefore, expected number of times the while loop runs is n.
- Thus, the expected time complexity of step 2 is O(n).

20.

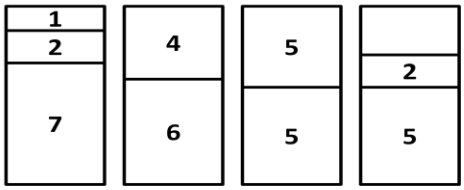
a) Define approximation algorithm. Give an approximation algorithm for bin packing using first fit heuristic and give its approximation ratio.
Ans:

- **First Fit Decreasing Algorithm**
  - Sort the items in the descending order of their size
  - Apply First fit algorithm
  - Time Complexity
    - Best case Time Complexity = $\theta(n \log n)$
    - Average case Time Complexity = $\theta(n^2)$
    - Worst case Time Complexity = $\theta(n^2)$
- **Example:** bin capacity=10, sizes of the items are {5, 7, 5, 2, 4, 2, 5, 1, 6}.
  - Arrange the items in the decreasing order of the weight
    {7, 6, 5, 5, 5, 4, 2, 2, 1}




{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
6
{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
6
5
{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
6
5
5
{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
6
5
5
5
{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
6
4
5
5
5
{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
2
6
4
5
5
5
{7, 6, 5, 5, 5, 4, 2, 2, 1}
7
2
6
4
5
5
2
5

{7, 6, 5, 5, 5, 4, 2, 2, 1}

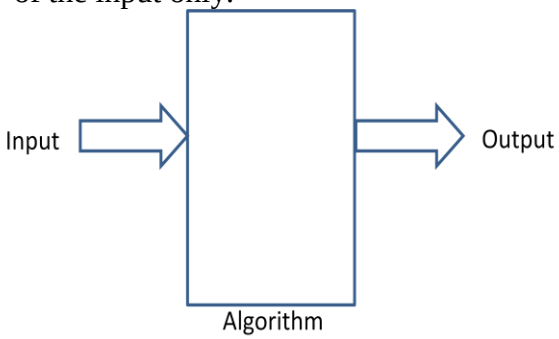


Number of bins required = 4

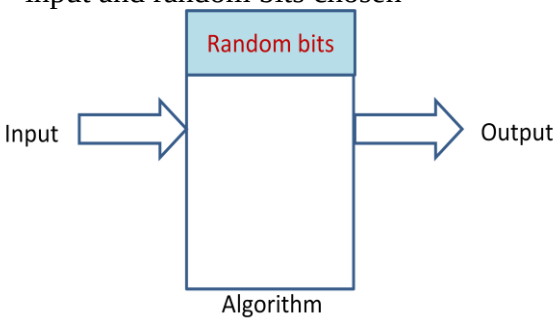
b) Discuss the advantages of randomized algorithms over deterministic algorithms. Discuss Las Vegas and Monte Carlo algorithms with a suitable example

Ans:

- **Deterministic Algorithm**: The output as well as the running time are functions of the input only.



- **Randomized Algorithm**: The output or the running time are functions of the input and random bits chosen



  - An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm
  - Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms

- **Type of Randomized Algorithms**
  - **Randomized Las Vegas Algorithms**
    - Output is always correct and optimal.
    - Running time is a random number
    - Running time is not bounded
    - Example: Randomized Quick Sort
  - **Randomized Monte Carlo Algorithms:**
    - May produce correct output with some probability

- A Monte Carlo algorithm runs for a fixed number of steps. That is the running time is deterministic

- **Example**: Finding an '*a*' in an array of *n* elements
  - **Input**: An array of $n \geq 2$ elements, in which half are '*a*'s and the other half are '*b*'s
  - **Output**: Find an '*a*' in the array

  - **Las Vegas algorithm**

```
Algorithm findingA_LV(A, n)
{
        repeat
        {
                Randomly choose one element out of n elements
        }until('a' is found)
}
```

    - The expected number of trials before success is 2.
    - Therefore the time complexity = O(1)

  - **Monte Carlo algorithm**

```
Algorithm findingA_MC(A, n, k )
{
        i=0;
        repeat
        {
                Randomly select one element out of n elements
                i=i+1;
        }until(i=k or 'a' is found);
}
```

    - This algorithm does not guarantee success, but the run time is bounded. The number of iterations is always less than or equal to k.
    - Therefore the time complexity **= O(k)**