1.State the elements of greedy algorithm.

1. **Greedy choice property**

***Global optimal solution can be achieved by making a local optimal solution***

1. **Optimal Substructure**

**A problem exhibits optimal substructure if an optimal solution to the problem contain within it optimal solution to the subproblems.**

**2.use of relaxation function in shortest path algorithm**

**RELAX (u, v, w)**

**1. If d [v] > d [u] + w (u, v)**

**2. then d [v] ← d [u] + w (u, v)**

**3. π [v] ← u**

**It is used to decrease an upper bound on the actual shortest path weight of each vertex until the upper bound equivalent the shortest - path weight.**

**3.Augmenting path and residual capacity:-** **An augmenting path is a simple path from source to sink which do not include any cycles and that pass only through positive weighted edges.**

**Residual Capacity:- Original capacity of the edge – flow**

**4.** Define decision problem and optimization problem.

**Decision Problem:**

**A decision problem is a type of computational problem that requires a yes/no answer based on a given input. It is a problem that can be formulated as a question, and the goal is to determine whether the answer to the question is "yes" or "no**

**Optimization Problem:**

**An optimization problem is a type of computational problem that involves finding the best solution from a set of possible solutions which focus on a binary yes/no answer, optimization**

**problems seek to find the best possible solution, which may involve maximizing profits,**

**minimizing costs, optimizing efficiency, or achieving an optimal arrangement.**

**.**

**5.** Briefly explain divide and conquer approach for problem solving.

**Divide and conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy.**

**Divide: This involves dividing the problem into smaller sub-problems.**

**Conquer: Solve sub-problems by calling recursively until solved.**

**Combine: Combine the sub-problems to get the final solution of the whole problem.**

**Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property.**

**F) Define Time Complexity and Space Complexity:-**

**Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.**

**g)** Define recurrence with example.

**Recurrence is a way in which a function can be described in terms of its value on smaller inputs.**

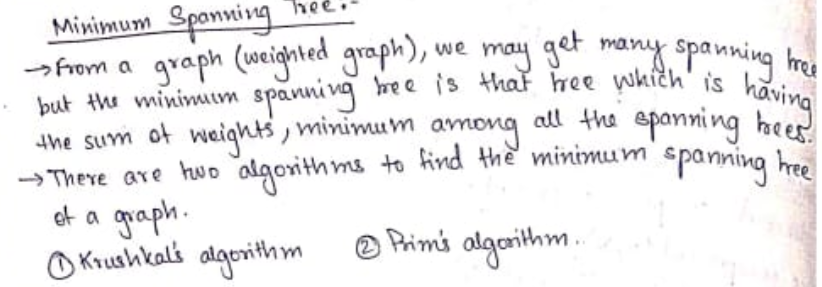
**Example:-**

**S(n) = n + S(n-1)**

**h) Different graph representation techniques are:-**

**To represent a graph in the memory, we have two ways**

1. **Linked list representation:- for sparse graph**
2. **Adjacency matrix:- For Dense graph**

**I ans Minimum Spanning tree:-** 

**J ans State Branch and bound technique:-**

**Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems.**

**k)**

**NO as f(n) = O(g(n))**

**f(n)<=c.f(n)**

**l)** **Tractable problems, also known as solvable or efficiently solvable problems, are those for which there exists an algorithm that can solve them within a reasonable amount of time. These algorithms have polynomial time**

**intractable problems are those for which there is**

**no known algorithm that can solve them efficiently for all possible inputs. These problems have superpolynomial time complexity.**

**Focused Short**

**Q1)Algorithm and time complexity of Merge Sort:-**

1. MERGE\_SORT(arr, beg, end)
2. **if** beg < end
3. set mid = (beg + end)/2
4. MERGE\_SORT(arr, beg, mid)
5. MERGE\_SORT(arr, mid + 1, end)
6. MERGE (arr, beg, mid, end)
7. end of **if**
9. END MERGE\_SORT

void merge(int a[], int beg, int mid, int end)

{

int i, j, k;

int n1 = mid - beg + 1;

int n2 = end - mid;

int LeftArray[n1], RightArray[n2]; //temporary arrays

/\* copy data to temp arrays \*/

for (int i = 0; i < n1; i++)

LeftArray[i] = a[beg + i];

for (int j = 0; j < n2; j++)

RightArray[j] = a[mid + 1 + j];

i = 0, /\* initial index of first sub-array \*/

j = 0; /\* initial index of second sub-array \*/

k = beg; /\* initial index of merged sub-array \*/

while (i < n1 && j < n2)

{

if(LeftArray[i] <= RightArray[j])

{

a[k] = LeftArray[i];

i++;

}

else

{

a[k] = RightArray[j];

j++;

}

k++;

}

while (i<n1)

{

a[k] = LeftArray[i];

i++;

k++;

}

while (j<n2)

{

a[k] = RightArray[j];

j++;

k++;

}

}

Time complexity of MergeSort

STEP-1 Is to divide the array into two parts of equal size .

2 \* T(n/2) --> Part 1

STEP-2 Now to merge baiscall traverse through all the elements.

constant \* n --> Part 2

STEP-3 --> COMBINE 1 + 2

T(n) = 2 \* T(n/2) + constant \* n --> Part 3

Hence,

T(N) = N \* T(1) + N \* logN

= O(N \* log(N))

**Q2)Ans**

**Our Adjacency Matrix is:**

**0 1 1 0**

**0 0 1 0**

**1 0 0 1**

**0 0 0 0**

**After making diagonal element equals to 1**

**1 1 1 0**

**0 1 1 0**

**1 0 1 1**

**0 0 0 1**

**after k = 0 Our Matrix is**

**1 1 1 0**

**0 1 1 0**

**1 1 1 1**

**0 0 0 1**

**after k = 1 Our Matrix is**

**1 1 1 0**

**0 1 1 0**

**1 1 1 1**

**0 0 0 1**

**after k = 2 Our Matrix is**

**1 1 1 1**

**1 1 1 1**

**1 1 1 1**

**0 0 0 1**

**after k = 3 Our Matrix is**

**1 1 1 1**

**1 1 1 1**

**1 1 1 1**

**0 0 0 1**

**c) ans**

**Algorithm Of Fractional Knapsack**

**Consider all the items with their weights and profits mentioned respectively.**

**Calculate Pi/Wi of all the items and sort the items in descending order based on their Pi/Wi values.**

**Without exceeding the limit, add the items into the knapsack.**

**If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.**

**Hence, giving it the name fractional knapsack problem.**

**D)ans Circuit Satisfiability, also known as the Boolean Circuit Satisfiability problem or the SAT**

**problem, is a decision problem in computer science. It involves determining if there exists an input**

**assignment to the inputs of a Boolean circuit that makes the circuit output a true value.**

**To prove: -**

**1. Concept of 3CNF SAT**

**2. SAT≤ρ 3CNF SAT**

**3. 3CNF≤ρ SAT**

**4. 3CNF ϵ NPC**

**1. CONCEPT: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have**

**almost 3 literals or constants.**

**2. SAT ≤ρ 3CNF SAT:- In which firstly you need to convert a Boolean function created in**

**SAT into 3CNF either in POS or SOP form within the polynomial time**

**F=X+YZ**

**= (X+Y) (X+Z)**

**= (X+Y+ZZ') (X+YY'+Z)**

**= (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)**

**= (X+Y+Z) (X+Y+Z') (X+Y'+Z)**

**3. 3CNF ≤p SAT: - From the Boolean Function having three literals we can reduce the**

**whole function into a shorter one.**

**F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)**

**= (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)**

**= (X+Y+ZZ') (X+YY'+Z)**

**= (X+Y) (X+Z)**

**= X+YZ**

**4. 3CNF ϵ NPC: - As you know very well, you can get the 3CNF through SAT and SAT**

**through CIRCUIT SAT that comes from NP.**

**E)ans**

**lets T(n) ne total time complexity for worst case**

**n = total number of elements**

**T(n) = T(n-1) + constant\*n**

**as we are dividing array into two parts one consist of single element and other of n-1**

**and we will traverse individual array**

**T(n) = T(n-2) + constant\*(n-1) + constant\*n = T(n-2) + 2\*constant\*n - constant**

**T(n) = T(n-3) + 3\*constant\*n - 2\*constant - constant**

**T(n) = T(n-k) + k\*constant\*n - (k-1)\*constant ..... - 2\*constant - constant**

**T(n) = T(n-k) + k\*constant\*n - constant\*[(k-1) .... + 3 + 2 + 1]**

**T(n) = T(n-k) + k\*n\*constant - constant\*[k\*(k-1)/2]**

**put n=k**

**T(n) = T(0) + constant\*n\*n - constant\*[n\*(n-1)/2]**

**removing constant terms**

**T(n) = n\*n - n\*(n-1)/2**

**T(n) = O(n^2)**