Name- Jitendra kumar
Roll No.- Crs1914
Course- Design & Analysis of Algorithm
Ph. No. - 8709359481

(7) (a) Basic Idea of Greedy Algorithm:-

→ A Greedy Algorithm is algorithm that always makes the choice that looks best at the moment.

That is, It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

→ Greedy Algorithm do not always yield optimal solutions, but for many problems they do.

→ A greedy Algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the Algorithm, the choice that seems best at the moment is chosen

(b) Minimum Spanning Tree for an Undirected weighted graph:

Let G be a connected, Undirected graph with vertex set V and edge set E.

That is $G=(V,E)$ with a weight function $w: E \to R$

We wish to find a minimum spanning Tree:

For this, We will use Kruskal's Algorithm.

Kruskal's Algorithm is a greedy Algorithm, because at each step it adds to the forest an edge of least possible weight.

It uses a disjoint ~~set~~ set data structure to maintain several disjoint set of elements.

Each set contains the vertices in a tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v).

Hence the Algorithm is given as:

```
MST_KRUSKAL(G, w)
1.  A ← φ
2.  for each vertex v ∈ V[G]
3.      do MAKE-SET(v)
4.  sort the edges of E by non-decreasing weight w.
5.  for each edge (u,v) ∈ E, in order by non-decreasing weight.
6.      do if FIND-SET(u) ≠ FIND-SET(v)
7.          then A ← A ∪ {(u,v)}
8.               UNION(u,v)
9.  return A.
```

(C) Proof of Kruskal's Algorithm:

That is, Kruskal's Algorithm produces a minimum spanning tree.

Proof:- Let G be a connected, undirected weighted graph, and let x be the subgraph of G produced by the Algorithm. Now, first we prove that Spanning Tree for x:

Clearly, X can not form a cycle and X can not be disconnected, because the first encountered edge that joins two components of X would have been added by the algorithm. Thus, X is a spanning tree of G.

Now, we prove that Minimality of X. That is X has minimum weight.

We prove this by contradiction. Let us suppose that X is not minimal spanning tree and among all minimum weight spanning trees, we select X_1 which has the smallest number of edges which are not in X. Consider the edge e which was first to be added by the algorithm to X of those which are not in X_1.

$X_1 \cup e$ forms a cycle.

Since X is a tree, so X can not contain all edges of this cycle. Therefore this cycle contains an edge f which is not in X.

The graph

$X_2 = X_1 \cup e \setminus f$ is also a spanning tree and therefore its weight can not be less than the weight of X_1 because X_1 is one of the minimal spanning tree with the smallest number of edges and hence the weight of e can not be less than the weight f. ie $e \geq f$.

However the edge e is selected at the first step of Kruskal's Algorithm which implies that is of the minimal weight of all edges ie $e \leq f$.

Thus, the weight of e and f are equal ie $e = f$.

and Hence, X_2 is also minimal spanning tree. But X_2 has one more edge in common with X and X_1. which contradicts to the choice of X_1.

Thus, X is a minimal spanning tree.

(d) Time Complexity of Kruskal's Algorithm:

The Running-time of Kruskal's Algorithm for Graph $G=(V,E)$ depends on the implementation of the disjoint-data structure. Let us assume the disjoint-set forest implementation with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known.

Initialization takes time $O(V)$, and time to sort the edges in line 4 is $O(E \log E)$. Since there are $O(E)$ operations on the disjoint-set forest, which takes total $O(E \cdot \alpha(E,V))$ time, where α is the functional Inverse of Ackermann's function. Since $\alpha(E,V) = O(\log E)$.

Hence, the total Running time of Kruskal's Algorithm is

$$\boxed{O(E \cdot \log E)}$$

⑥ (a) Longest Common subsequence problem:-

Let X and Y be two given sequences then find the length of their LCS Z. where Z is longest Common Subsequence (LCS), that is Z is a subsequence of both X and Y, any sequence longer than Z is not a subsequence of at least one of X or Y.

Example:-

(i) LCS for input sequences ABCDGHIKL and AEDFHRKPQS is

ADHK

(ii) LCS for input sequences

PQRSTUVW and PRTVWABC

is PRTVW.

(b) Algorithm In Dynamic programming Approach to solve LCS Problem:

Let $X = x_1 x_2 x_3 \ldots x_m$ &

$Y = y_1 y_2 y_3 \ldots y_n$ be two sequences so that length of X be m and length of Y be n.

Let X[1:K] denote the prefix

$X[1:K] = x_1 x_2 \ldots x_K$.

Now, Two cases arise:

Case 1: If $x_m = y_n = l$ (say)

Then clearly any LCS Z has l as its last symbol.

Thus, If $|Z| = K$ and $x_m = y_n = l$

We can write,

$Z[1:K-1] = LCS(X[1:m-1], Y[1:n-1])$

and $Z = Z[1:k-1] \circ l$,

where $\circ$ denote the concatenation operation.

Case 2: - If $x_m \neq y_n$.

Let Z be the LCS of X and Y.

In this case, last letter of Z (say z_k) is either not equal to x_m or it is not equal to y_n.

Hence

$$LCS(X,Y) = LCS(X[1:m-1], Y)$$

or $$LCS(X,Y) = LCS(X, Y[1:n-1])$$

whichever is longer

Thus, length of LCS(X,Y) is given by

$$\text{len } LCS(X,Y) = \max\{\text{len } LCS(X[1:m-1], Y), \text{len } LCS(X, Y[1:n-1])\}$$

Let us keep a table C, so that

$$C[i,j] = \text{length of } LCS(X[1:i], Y[1:j])$$

Then we have:

$$C[i,j] = \begin{cases} 0 & \text{If } i=0 \text{ or } j=0 \\ c[i-1, j-1]+1 & \text{If } x[i] = Y[j], \; i,j>0 \\ \max\{C[i-1, j], c[i, j-1]\} & \text{If } x[i] \neq Y[j], \; i,j>0. \end{cases}$$

Or

$$c[i,j] = \begin{cases} 1 + c[i-1, j-1] & \text{If } x[i]=y[j] \\ \max\{c[i-1,j], c[i, j-1]\} & \text{otherwise} \end{cases}$$

The Recursive relationship defined above gives Dynamic programming Algorithm for filling out the table C:

Algorithm 1: len LCS(X, Y)

Initialize an $(n+1) \times (m+1)$ zero-indexed array C.

Set $C[0, j] = C[i, 0] = 0$ for all $i, j \in \{1, \ldots m\} \times \{1, \ldots n\}$

```
for i = 1, ... m do
    for j = 1, ... n do
        if X[i] = Y[j] then
            C[i,j] ← C[i-1, j-1] + 1
        else
            C[i,j] ← max{C[i-1, j], C[i, j-1]}
Return C
```

→ Algorithm 1 only gives the length of the LCS of X and Y.

If we want to recover the actual Longest Common Subsequence, then following algorithm is given:

Algorithm 2: LCS(X, Y, C)
// C is filled out already in Algorithm 1

```
L ← ∅
i ← m
j ← n
while i > 0 and j > 0 do
    if X[i] = Y[j] then
        append X[i] to the beginning of L
        i ← i - 1
        j ← j - 1
```

```
else if C[i,j] = C[i,j-1] then
    L j ← j-1
else
    L i ← i-1
```

(C) Time Complexity of Algorithm:

Since C[i,j] only depends on three possible prior values:

C[i-1,j], C[i,j-1] and C[i-1,j-1]

This means that each step we compute a new value C[i,j] from previous entries. It takes O(1) time.

For Algorithm 1: In order to fill each entry, we only need to perform a constant number of lookups and additions. Thus, we need to do a constant amount of work for each of the mxn entries, giving a running time O(mn)

For Algorithm 2: In each step, the sum i+j is decremented by at least one (maybe two) and stops as soon as one of i, j is equal to zero, this is at least before i+j=0. Thus the number of times we decrement i+j at most m+n, which was their total value to start

Because at each step of Algorithm 2, the work is O(1), the total Running time is thus O(m+n) which is subsummed by the runtime O(mn) necessary to fill in the table.

CONCLUSION:- The Running time of LCS(X,Y) of a sequence of length m and a sequence of length n is time

O(mn).

⑤ (a) Algorithm to solve the All Pair Shortest Path Problem in a directed weighted Graph :-

Let $G = (V, E)$ be a directed graph (weighted). Here Negative-weight may be present, but we shall assume that there are no negative-weight cycle.

Let W be an $n \times n$ matrix representing the edge weights of an n-vertex directed graph $G = (V, E)$

That is

$$W = (w_{ij}), \text{ where}$$

$$w_{ij} = \begin{cases} 0 & \text{If } i = j \\ \text{the weight of directed edge } (i,j) & \text{If } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{If } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

We are going to use FLOYD-WARSHALL ALGORITHM to find all pair shortest path in a directed graph.

The Algorithm considers the "Intermediate vertex of a shortest path, where an Intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots v_k \rangle$ is any vertex of p other than v_1 or v_k. that is any vertex in the set

$$\{v_2, v_3, \ldots v_{k-1}\}$$

The Floyd-Warshall algorithm is based on the following Observation:-

Let the vertices of G be
$V = \{1, 2, 3, \ldots n\}$ and consider a subset $\{1, 2, 3, \ldots k\}$ of vertices for some k. For any pair of vertices $i, j \in V$, consider all paths from i to j whose intermediate vertices are all drawn from $\{1, 2, 3 \ldots k\}$, and let p be a minimum-weight path from among them.

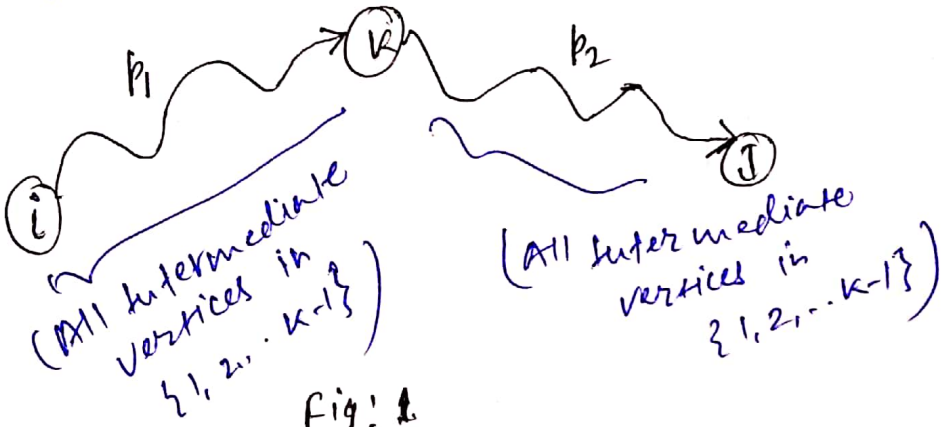


Fig: 1

The Floyd-Warshall Algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set $\{1, 2, \ldots k-1\}$.

The Relationship depends on whether or not k is an intermediate vertex of path p.

→ If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, 3, \ldots k-1\}$. Thus, a shortest path from vertex i to j with all intermediate vertices in the set $\{1, 2 \ldots k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, 3 \ldots k\}$.

→ If k is an intermediate vertex of path p, then we break p down into

$i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$

as shown in Fig: 1.

Now, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, 3 \ldots k\}$.

In fact, vertex k is not an intermediate vertex of path p_1, and so p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, 3, \ldots k-1\}$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, 3 \ldots k-1\}$

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, 3, \ldots k\}$

when $k=0$, A path from vertex i to vertex j with no intermediate vertex.

Hence $d_{ij}^{(0)} = w_{ij}$

A Recursive definition is given by:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{If } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{If } k \geq 1 \end{cases}$$

The matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer. where

$d_{ij}^{(n)} = \delta(i,j)$ for all $i, j \in V$.

where $\delta(i,j)$ denotes the shortest-path weight from vertex i to vertex j.

Hence, Algorithm is given as:

INPUT: $n \times n$ matrix W, defined as above

OUTPUT:- matrix $D^{(n)}$ of shortest-path weights.

```
FLOYD-WARSHALL (W)
1.  n ← rows[W]
2.  D^(0) ← W
3.  for k ← 1 to n
4       do for i ← 1 to n
5.          do for j ← 1 to n
6.              dij ← min( dij^(k-1), dik^(k-1) + dkj^(k-1) )
7.  return D^(n).
```

(5) (b) Time-Complexity of FLOYD-WARSHALL Algorithm:

Since, Floyd-warshall Algorithm consists of three loops over all the vertices of graph. The Inner most loop consist of only constant complexity operations.

Hence, The time complexity of Floyd-warshall Algorithm is

$O(n^3)$, where n is the number of vertices in a graph.

(3) (a) C Program to implement the Merge Sort - Algorithm :-

```
#include <stdio.h>
#include <stdlib.h>
// Merges two subarrays of arr[].
// First subarray is arr[l...m]
// Second subarray is arr[m+1...r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m-l+1;
    int n2 = r-m;
    /* create temp arrays */
    int L[n1], R[n2];
    for (i=0; i<n1; i++)
        L[i] = arr[l+i];
    for (j=0; j<n2; j++)
        R[j] = arr[m+1+j];
    /* Merge the temp arrays back
       into arr[l...r] */
    i=0;
    j=0;
    k=l;
    while (i<n1 && j<n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++
        }
        else {
            arr[k] = R[j];
            j++;
```

```
        }
        k++
    }
/* Copying the remaining elements of
        L[], if there are any */
while (i < n1) {
    arr[k] = L[i]
    i++;
    k++;
}
/* Copying the remaining elements of
        R[] if there are any */
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}
void mergesort (int arr[], int l, int r)
{
    if (l < r) {
        int m = l+(r-l)/2;
        // sort first and second halves
        mergesort (arr, l, m);
        mergesort (arr, m+1, r);
        merge (arr, l, m, r);
    }
}
```

Example:

```
/* function to print an array */
void printArray (int A[], int size)
{
    int i;
```

```
for (i=0 ; i<size ; i++)
    Printf ("%d", A[i]);
    Printf ("\n");
}
int main ()
{
    int arr[ ] = {10, 15, 12, 8, 6, 2}
    int arr_size = sizeof (arr) / sizeof
                                (arr[0]);
Printf ("Given array is \n");
 Printf Array( arr, arr_size);
 mergesort (arr, 0, arr_size-1);
 Printf ("\n sorted array is \n");
  Printf Array (arr, arr_size);
  return 0;
}
```

OUTPUT: Given array is
15, 12, 10, 8, 6, 2.

3. (b) Program to Generate sequence :

```
#include <stdio.h>
#define BDAY 1
 int main ()
{
    int arr[20];
     arr[0] = -1;
     arr [1] = BDAY
     for (int i=2; i<20; i++)
     {
         arr[i] = (arr[i-1] +1) * (arr[i-1]+1);
         arr[i] = arr[i]%(9 7+ arr [i-2]);
```

```
}
    for (int i=0; i<16; i++)
      printf ("%d\t", arr[i]);
      printf ("\n");
      return 0;
}
```

OUTPUT: My birthday falls on 25th sept.

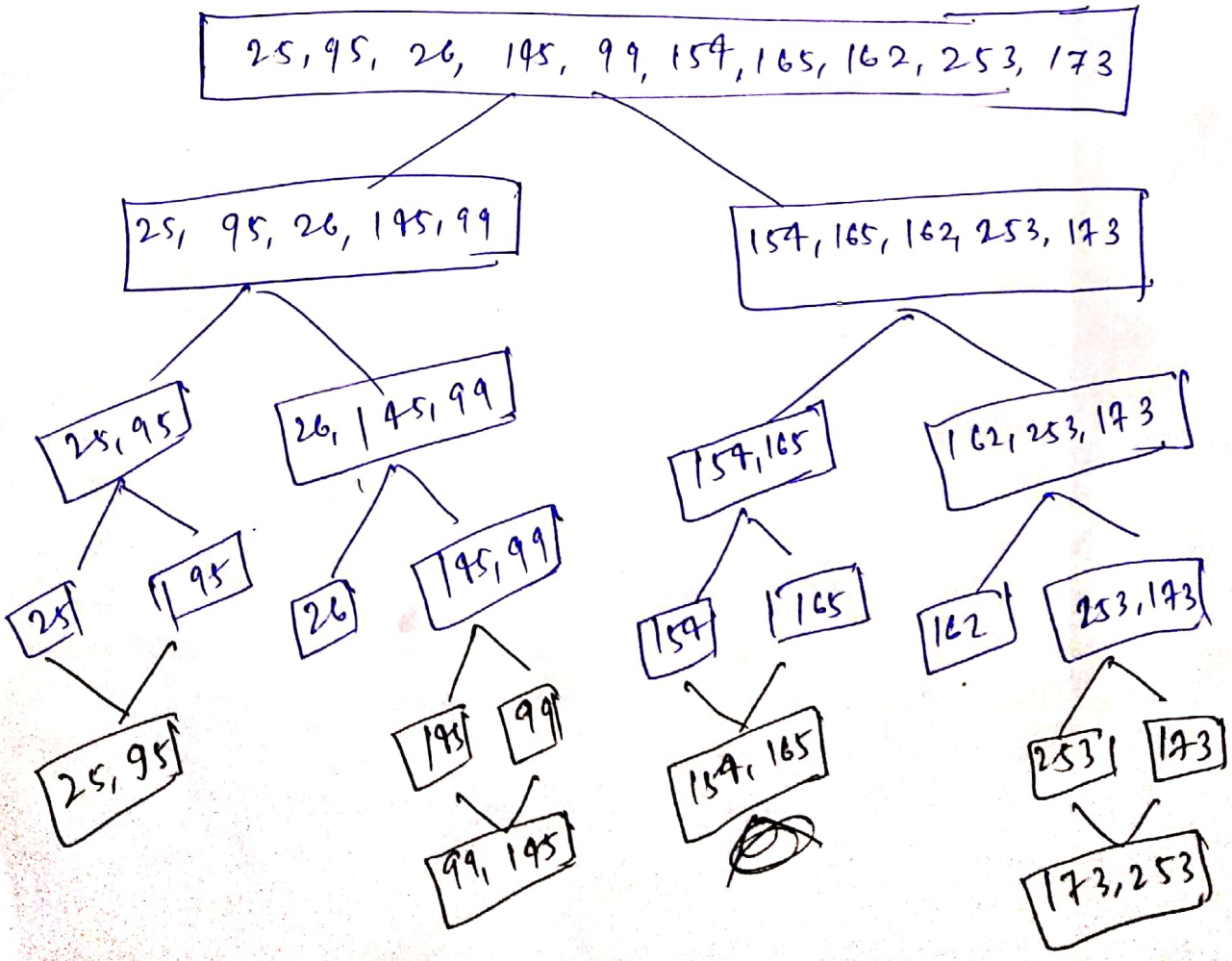
So, out put of the program is

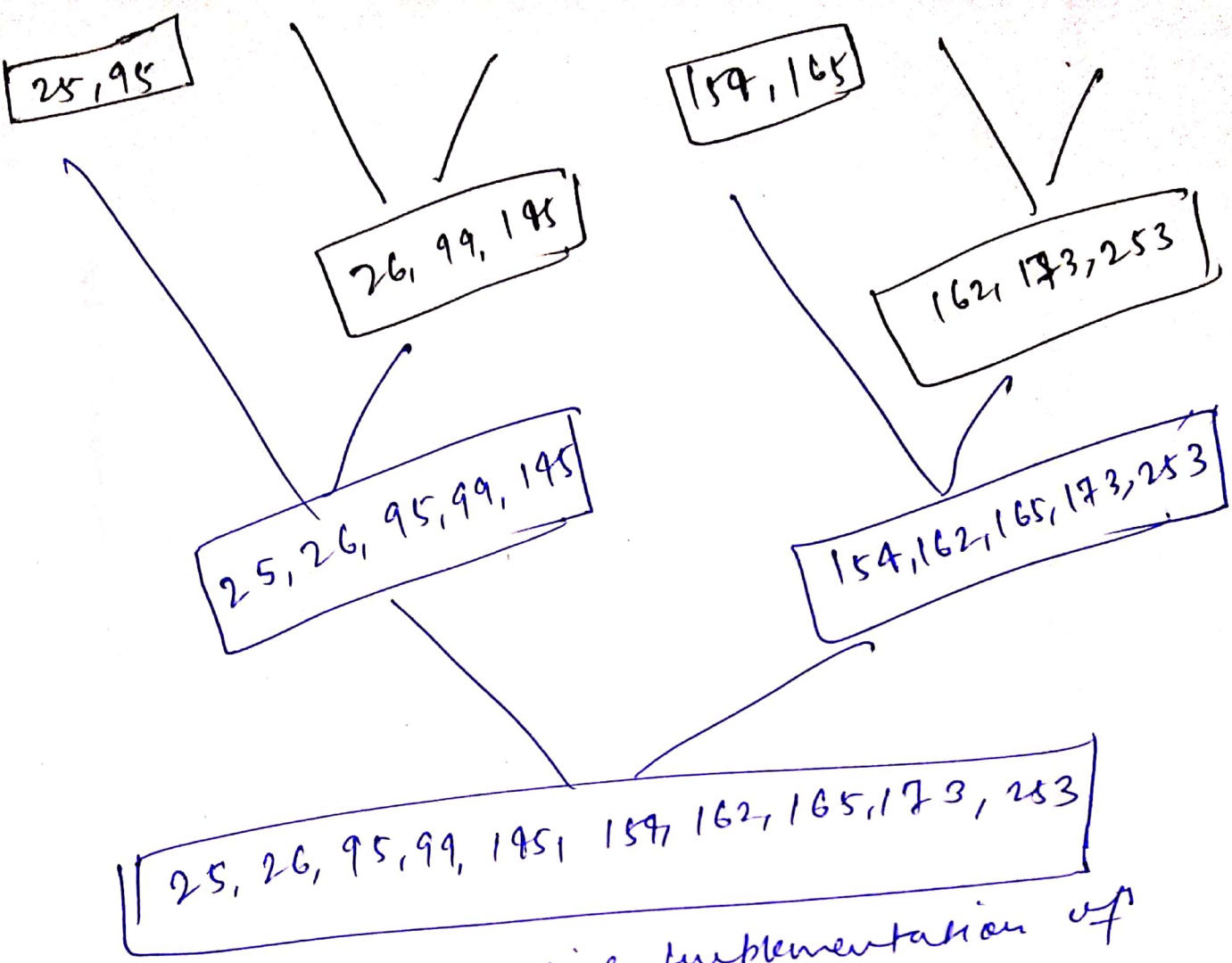
25, 95, 26, 145, 99, 154, 165, 162, 253, 173,
265, 216, 309, 286, 325, 347

After Sorting, we get:

25, 26, 95, 99, 145, 154, 162, 165, 173,
216, 253, 265, 286, 309, 325, 347

Now, Implementation executes on first 10 output ie a[1], a[2]... a[10] to sort them in ascending order:





Merge sort: Recursive implementation of first 10 elements.

⑨ (a) We have to explain 3-SAT and vertex cover problems!

3-SAT Problem:

First, We want to define the terminology which is related to this problem:

Literal: Boolean variable and its negation

Clause: Combination of several literals connected with V.

Conjunctive normal form or CNF: Several clause is connected with ∧.

Now, let us consider a boolean function ϕ in three variables which is in CNF form. Let the three variables be x_1, x_2 and x_3.

Then 3-SAT problem states that "To check whether there is a combination of (x_1, x_2, x_3) which satisfies to given Boolean function".

for example:

$$\phi(x_1, x_2, x_3) = (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_1 \vee x_3) \wedge (x_1 \vee \bar{x}_3 \vee x_2)$$

- Vertex-Cover problem:

First we want to define the terminology related to this problem:

Vertex-Cover:- Let $G=(V, E)$ be an undirected graph. Then A vertex cover of G is a subset of vertices V' such that $V' \subseteq V$ and if edge (u,v) is an edge of G, then either u in V' or v in V' or both.

Then vertex-cover problem says that "For a given graph and a number k whether there is a vertex cover of size k in G".

(9)

(9) (b) Considering 3-SAT problem as NP-complete, We have to prove that vertex-cover problem is also NP-complete.

First, we want to define some terminology related to this proof.

NP-problem:- NP is the set of all those decision problem for which the problem is solvable in polynomial time by a deterministic turing Machine.

NP-Hard:- Let L be a given problem or Language. Then L will be ~~hard~~ NP-hard If $\forall L' \in NP$,

$$L' \leq_p L$$

ie L' is Reducible to L in polynomial time.

NP-Complete:- Let L be a given Language or problem. Then We say that L is NP-complete If

→ $L \in NP$

and → L is NP-hard.

Now, We will show that vertex-cover problem is NP-complete.

Let G be an undirected graph and k is given positive integer.

Now, Define the set

$$VCOV := \{ \langle G, k \rangle \mid G \text{ has } k \text{ many node vertex-cover} \}$$

Proving vertex cover is ~~NP~~ in NP, the vertex cover of size k will work as certificate.

To prove it is NP-hard, we will show that 3SAT is Reducible to Vertex-Cover problem.

ie $3SAT \leq_p$ Vertex-Cover problem

For this, we will show that a mapping reduction from 3SAT to Vertex-Cover

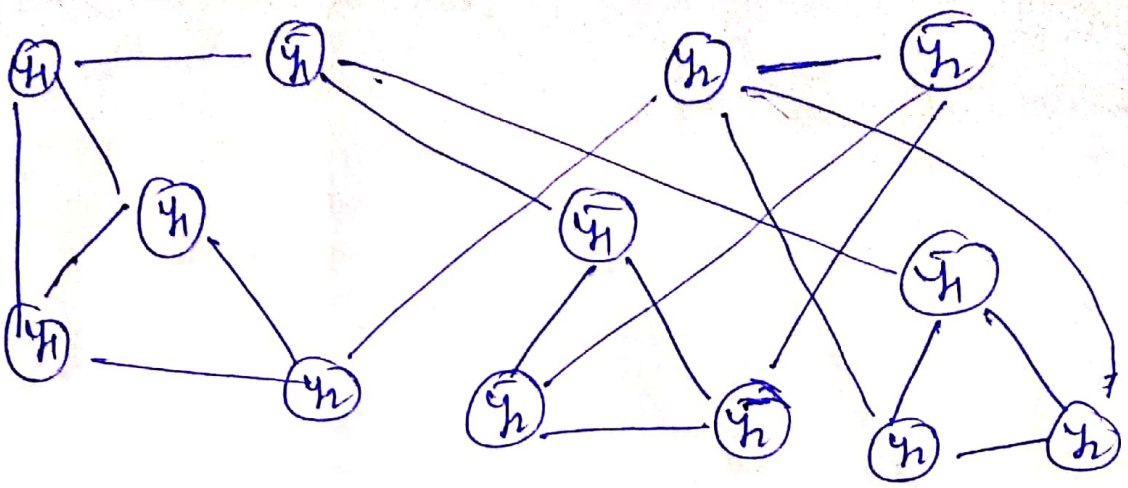
For this, we convert a 3cnf formula into a graph and an inter G and k respectively in such a manner that whenever there is a vertex-cover of k in G, the 3cnf Boolean formula will be satisfied.

Let us consider the Boolean function as:

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$$

Now, we set-up Gadget for variable as follows:

For each variable x in ϕ, we draw two edges into gadget. We level the two nodes in the gadget as x and $\bar{x}$. If the node is included in vertex-cover x will be TRUE else FALSE.



To set-up gadget for ~~variable~~ clause, we do following:

We combine three inter connected nodes to make a gadget, and connect to variable gadget whenever the label matches.

Now, If ϕ be an δ variable λ clause, Boolean function then total number of nodes in our constructed graph will be

$$2\delta + 3\lambda.$$

Let $\quad k := 2\delta + 3\lambda.$

Now, we will show the Reduction:

For this, we Assume that ϕ is satisfiable, so we put ~~one~~ the nodes of variable gadget corresponding to the true literals in assigned ~~vertex~~ vertex-cover and we put one true literal from every clause gadget into vertex-cover. So the Cardinality of our vertex cover is k.

Hence

3SAT $\Rightarrow$ Vertex-Cover
satisfiability

Again on the contrary, we assume that G has a vertex-cover of size k.

Now, this cover must contain one node from every clause gadget, as it has to cover our graph both consisting of both type of gadgets.

In order to make ϕ satisfiable we assign TRUE to the corresponding literals.

Hence

Vertex-Cover $\Rightarrow$ 3 SAT satisfiability

Hence vertex-cover problem is NP-Complete.