**Week 9**

**Q1) Given a graph, design an algorithm and implement it using a program to implement Floyd-**

**Warshall all pair shortest path algorithm.**

**Input Format:**

**The first line of input takes number of vertices in the graph.**

**Input will be the graph in the form of adjacency matrix or adjacency list. If a direct edge is not**

**present between any pair of vertex (u,v), then this entry is shown as AdjM[u,v] = INF.**

**Output Format:**

**Output will be shortest distance matrix in the form of V X V matrix, where each entry (u,v)**

**represents shortest distance between vertex u and vertex v.**

**Solution:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n, i, j, k, w;

cin >> n;

int graph[n][n];

string temp;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

cin >> temp;

if (temp != "INF")

{

graph[i][j] = stoi(temp);

} else {

graph[i][j] = 1e8;

}

}

}

for (k = 0; k < n; k++)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if (graph[i][k] + graph[k][j] < graph[i][j])

{

graph[i][j] = graph[i][k] + graph[k][j];

}

}

}

}

cout << "The shortest path matrix: " << endl;

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if(graph[i][j] >= 1e8) cout << "INF";

else cout << graph[i][j];

cout << " ";

}

cout << endl;

}

return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Q2) Given a knapsack of maximum capacity w. N items are provided each having its own value and weight. You have to design and algorithm and implement it using a program to find the list of the selected items such that the final selected content has weight w and has maximum value. You can take fractions of items i.e. the items can be broken into smaller pieces so that you have to carry only a fraction xi of items i, where 0<xi<1.**

**Input Format:**

**First input line will take number of items N which are provided.**

**Second input line will contain N space-separated array containing weights of all N items.**

**Third input will contain N space-separated array containing values of all N items.**

**Last line of input will take the maximum capacity w of knapsack.**

**Output Format:**

**First output line will give maximum value that can be achieved.**

**Next line of output will give list of items selected along with their fraction of amount which has been taken.**

**Solution:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin >> n;

vector<double> items(n);

vector<double> val(n);

vector<vector<double>> job;

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

{

cin >> items[i];

}

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

{

cin >> val[i];

job.push\_back({val[i] / items[i], items[i], (double)(i + 1)});

}

double k;

cin >> k;

sort(job.rbegin(), job.rend());

vector<pair<double, double>> ls;

float profit = 0;

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

{

if (job[i][1] >= k)

{

profit += k \* job[i][0];

ls.push\_back(make\_pair(k, job[i][2]));

break;

}

else

{

profit += job[i][1] \* job[i][0];

}

ls.push\_back(make\_pair(job[i][1], job[i][2]));

k = k - job[i][1];

}

cout << "Maximum Value : " << profit << endl;

cout << "Item - Weight" << endl;

for (auto it : ls)

cout << it.second << " - " << it.first << endl;

return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Q3) Given an array of elements. Assume arr[i] represents the size of file i. Write an algorithm and a program to merge all these files into single file with minimum computation. For given two files A and B with sizes m and n, computation cost of merging them is O(m+n). (Hint: use greedy approach)**

**Input Format:**

**First line will take the size n of the array.**

**Second line will take array s an input**

**Output Format:**

**Output will be minimum computation cost required to merge all elements of array.**

**Solution:**

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin >> n;

vector<int> a(n);

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

{

cin >> a[i];

}

priority\_queue<int, vector<int>, greater<int>> minheap;

for (int i = 0; i < n; i++) {

minheap.push(a[i]);

}

int ans = 0;

while (minheap.size() >1)

{

int e1 = minheap.top();

minheap.pop();

int e2 = minheap.top();

minheap.pop();

ans += e1 + e2;

minheap.push(e1 + e2);

}

cout << ans;

return 0;

}

**OUTPUT**

**Graphical user interface, text

Description automatically generated**