Syllabus codes

Quicksort

```
#include <iostream>
using namespace std;
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
int main() {
    int n = 10; // Number of elements in the array
    int arr[] = { 10, 7, 8, 9, 1, 5, 4, 6, 6 }; // Initialize the
array

    quickSort(arr, 0, n - 1);

    cout << "Sorted array using Quick Sort:" << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    } return 0;
}
```

Mergesort

```
#include <iostream>
#include <algorithm>
using namespace std;

int b[100];

void merge(int a[], int low, int
middle, int high) {
    int i = low;
    int j = middle + 1;
    int k = low;

    while (i <= middle && j <= high)
{
        if (a[i] <= a[j]) {
            b[k] = a[i];
            k++;
            i++;
        } else {
            b[k] = a[j];
            k++;
            j++;
        }
    }

    while (i <= middle) {
        b[k] = a[i];
        k++;
        i++;
    }
while (j <= high) {
        b[k] = a[j];
        k++;
        j++;
    }
 for (int p = low; p <= high; p++) {
        a[p] = b[p];
    }
}

void mergeSort(int a[], int low, int
high) {
    if (low < high) {
        int middle = (low + high) / 2;
        mergeSort(a, low, middle);
        mergeSort(a, middle + 1, high);
        merge(a, low, middle, high);
    }
}

int main() {
    int n = 6; // Number of elements
in the array
    int a[] = { 12, 11, 13, 5, 6, 7 }; //
Initialize the array

    mergeSort(a, 0, n - 1);
cout << "Sorted array using Merge
Sort:" << endl;
    for (int i = 0; i < n; i++) {
        cout << a[i] << " ";
    }

    return 0;
}
```

Heap

```
///Max Heap

#include<bits/stdc++.h>
using namespace std;

class Heap {
private:
    int a[101], size;

public:
    Heap() {
        size = 0;
    }

private:
    void bottomTopAdjust(int i) {
        while (i > 1 && a[i] > a[i / 2]) {
            swap(a[i], a[i / 2]);
            i = i / 2;
        }
    }

    void topBottomAdjust(int i) {
        int child;
        while (2 * i <= size) {
            child = 2 * i;
            if (child + 1 <= size && a[child + 1] > a[child])
                child++;
            if (a[i] >= a[child])
                break;
            swap(a[i], a[child]);
            i = child;
        }
    }

public:
    bool insert(int val) {
        if (size >= 100)
            return false;
        size++;
        a[size] = val;
        bottomTopAdjust(size);
        return true;
    }

    int showMax() {
        if (size == 0)
            return -1; // Assuming -1 is an invalid value
        return a[1];
    }

    int showSize() {
        return size;
    }

    bool deleteRoot() {
        if (size == 0)
            return false;
        swap(a[1], a[size]);
        size--;
        topBottomAdjust(1);
        return true;
    }

    void buildHeap() {
        for (int i = size / 2; i >= 1; i--) {
            topBottomAdjust(i);
        }
    }

    void sort() {
        int heapSize = size;
        while (size > 1) {
            swap(a[1], a[size]);
            size--;
            topBottomAdjust(1);
        }
        size = heapSize; // Restore the original size
    }

    void bfs() {
        if (size == 0)
            return;
        int level = 1;
        queue<int> q;
        q.push(1);

        while (!q.empty()) {
            int parent = q.front();
            q.pop();
            if (parent == level) {
                cout << endl;
                level = level * 2;
            }
            cout << a[parent] << " ";
            if (2 * parent <= size) q.push(2 * parent);
            if (2 * parent + 1 <= size) q.push(2 * parent + 1);
        }
    }
};

int main() {

    Heap heap;

    while (true) {
        cout << "1. Insert    2. Show Max    3. Delete Max    4. Sort    5. Level    6. Build Heap    7. End" << endl << endl;
        int choice;
        cin >> choice;

        if (choice == 1) {
            cout << "Insert Value: ";
            int y;
            cin >> y;
            bool b = heap.insert(y);

            if (b)   cout << y << " is inserted in the heap" << endl;
        }

        else if (choice == 2) {
            if (heap.showSize() != 0) cout << "Max Element: " << heap.showMax();
            else    cout << "No element in the heap" << endl;
        }

        else if (choice == 3) {
            bool b = heap.deleteRoot();
            if (b)   cout << "Root deleted from heap";
            else    cout << "Heap is empty";
            cout << endl;
        }

        else if (choice == 4) {
            heap.sort();
        }

        else if (choice == 5) {
            cout << "Level Wise Traversal of the heap:" << endl;
            heap.bfs();
            cout << endl;
        }
        else if (choice == 6) {
            if (heap.showSize() == 0)
```

```
                cout << "Heap is Empty!" << endl;
            else
                heap.buildHeap();
        }else if (choice == 7) {
            break;
        }else {
            cout << "Invalid Choice" << endl;
        }
        cout << endl;
    }

    return 0;
}


/*
1 2
1 9
1 8
1 16
1 3
1 7
1 10
1 1
1 4
1 14
```

Priority Queue

```
///Max Heap

#include<bits/stdc++.h>
using namespace std;

class Heap {
private:
    int a[101], size;

public:
    Heap() {
        size = 0;
    }

private:
    void bottomTopAdjust(int i) {
        while (i != 1) {
            if (a[i] > a[i / 2])
                swap(a[i], a[i / 2]);
            else
                break;
            i = i / 2;
        }
    }

    void topBottomAdjust(int i) { /// HEAPIFY!
        int pseudoRoot = a[i];
        int pseudoIdx = i;
        while (i <= size / 2) {
            int leftVal = a[2 * i];
            int maxIdx = 2 * i;
            if ((2 * i + 1) <= size && a[2 * i + 1] > leftVal)
                maxIdx = 2 * i + 1;
            if (a[i] < a[maxIdx]) {
                swap(a[i], a[maxIdx]);
            }
            else {
                break;
            }
            i = maxIdx;
        }
    }

public:
    bool insert(int val) {
        if (size >= 100)
            return false;
        size++;
        a[size] = val;
        bottomTopAdjust(size);
        return true;
    }

    bool increaseKey(int x, int k) {
        if (x < 1 || x > size || k <= a[x])
            return false;
        a[x] = k;
        bottomTopAdjust(x);
        return true;
    }

    int showMax() {
        if (size == 0)
            return -1; // Assuming -1 is an invalid value
        return a[1];
    }

    int showSize() {
        return size;
    }

    int extractMax() {
        if (size == 0)
            return -1; // Assuming -1 is an invalid value
        int maxVal = a[1];
        swap(a[1], a[size]);
        size--;
        topBottomAdjust(1);
        return maxVal;
    }

    void bfs() {
        if (size == 0)
            return;
        int level = 1;
        queue<int> q;
        q.push(1);

        while (!q.empty()) {
            int parent = q.front();
            q.pop();
            if (parent == level) {
                cout << endl;
                level = level * 2;
            }
            cout << a[parent] << " ";
            if (2 * parent <= size) q.push(2 * parent);
            if (2 * parent + 1 <= size) q.push(2 * parent + 1);
        }
    }
};

int main() {

    Heap heap;

    while (true) {
        cout << "1. Insert    2. Increase Key    3. Show Max    4. Extract Max  5. Level Order Traversal 6. End" << endl << endl;
        int choice;
        cin >> choice;

        if (choice == 1) {
            cout << "Insert Value: ";
            int y;
            cin >> y;
            bool b = heap.insert(y);

            if (b)   cout << y << " is inserted in the heap" << endl;
        }
        else if (choice == 2) {
            cout << "Which node you
want to increase?" << endl;
            int nodeNo;
            cin >> nodeNo;
            cout << "What will be the
new value?" << endl;
            int value;
            cin >> value;
            bool b =
heap.increaseKey(nodeNo, value);
            if (b) cout << "Node value
increased successfully!" << endl;
            else cout << "Unsuccessful
Operation :(" << endl;
        }
        else if (choice == 3) {
            if (heap.showSize() != 0)
cout << "Max Element: " <<
heap.showMax();
```

```
            else    cout << "No element
in the heap" << endl;
        }
        else if (choice == 4) {
            if (heap.showSize() != 0)
cout << "Max element extracted: "
<< heap.extractMax();
            else    cout << "No element
in the heap" << endl;
        }
        else if (choice == 5) {
            cout << "Level Wise
Traversal of the heap:" << endl;
            heap.bfs();
            cout << endl;
        }
        else if (choice == 6)
            break;
        else {
            cout << "Invalid Choice" <<
endl;
```

```
        }
        cout << endl;
    }

    return 0;
}


/*
1 2
1 9
1 8
1 16
1 3
1 7
1 10
1 1
1 4
1 14
*/
```

TRIE

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int EoW;
    Node* children[26];
    Node() {
        EoW = 0;
        for (int i = 0; i < 26; i++) {
            this->children[i] = NULL;
        }
    }
};

void trie_insert(Node* root, string
s) {
    Node* current = root;
    for (char c : s) {
        int index = c - 'A'; // Assuming
uppercase letters only
        if (!current->children[index]) {
            current->children[index] =
new Node();
        }
        current = current-
>children[index];
    }
    current->EoW++;
}
```

```
int trie_search(Node* root, string s,
int k = 0) {
    Node* current = root;
    for (char c : s) {
        int index = c - 'A'; // Assuming
uppercase letters only
        if (!current->children[index]) {
            return 0; // Not found
        }
        current = current-
>children[index];
    }
    return current->EoW;
}

bool trie_delete(Node* root, string
s, int idx = 0) {
    if (!root) return false;

    if (idx == s.length()) {
        if (root->EoW > 0) {
            root->EoW--;
            return true;
        }
        return false;
    }

    int index = s[idx] - 'A'; //
Assuming uppercase letters only
    if (!root->children[index]) {
        return false; // Word not found
```

```
    }

    bool canDelete = trie_delete(root-
>children[index], s, idx + 1);

    if (canDelete && root-
>children[index]->EoW == 0) {
        delete root->children[index];
        root->children[index] =
nullptr;
    }

    return canDelete;
}

void printTRIEUtil(Node* root,
string s) {
    if (root->EoW > 0) {
        cout << s << " (" << root-
>EoW << ")" << endl;
    }
    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming
uppercase letters only
            printTRIEUtil(root-
>children[i], s + c);
        }
    }
}
```

```
void printTRIE(Node* root, string s = "") {
    printTRIEUtil(root, s);
}

void printStringsZA(Node* root, string s = "") {
    if (root->EoW > 0) {
        cout << s << " (" << root->EoW << ")" << endl;
    }
    for (int i = 25; i >= 0; i--) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printStringsZA(root->children[i], s + c);
        }
    }
}

void printPrefixStrings(Node* root, string prefix, string s = "") {
    if (prefix.length() > 0 && s != prefix) return;

    if (root->EoW > 0) {
        cout << s << " (" << root->EoW << ")" << endl;
    }

    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printPrefixStrings(root->children[i], prefix, s + c);
        }
    }
}

void printDuplicateStrings(Node* root, string s = "") {
    if (root->EoW > 1) {
        cout << s << " (" << root->EoW << ")" << endl;
    }

    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            char c = i + 'A'; // Assuming uppercase letters only
            printDuplicateStrings(root->children[i], s + c);
        }
    }
}

int main() {
    Node* root = new Node();

    while (1) {
        cout << "1. Insert    2. Search    3. Delete   4. Lexicographical Sorting  5. Display Strings (Z to A)"
                << "  6. Print Strings with Prefix  7. Print Duplicate Strings  8. End"
                << endl
                << endl;
        int choice;
        string x;
        cin >> choice;
        if (choice == 1) {
            cout << "Insert String: ";
            cin >> x;
            trie_insert(root, x);
            cout << x << " is inserted in the trie" << endl;
        } else if (choice == 2) {
            cout << "Enter string to search: ";
            cin >> x;
            if (trie_search(root, x) > 0)
                cout << x << " FOUND " << endl;
            else
                cout << x << " NOT FOUND " << endl;
        } else if (choice == 3) {
            cout << "Enter string to delete: ";
            cin >> x;
            if (trie_delete(root, x))
                cout << x << " DELETED " << endl;
            else
                cout << x << " NOT FOUND " << endl;
        } else if (choice == 4) {
            printTRIE(root);
        } else if (choice == 5) {
            printStringsZA(root);
        } else if (choice == 6) {
            cout << "Enter prefix: ";
            cin >> x;
            printPrefixStrings(root, x);
        } else if (choice == 7) {
            printDuplicateStrings(root);
        } else if (choice == 8) {
            break;
        } else {
            cout << "Invalid Choice" << endl;
            break;
        }
        cout << endl;
    }

    return 0;
}
```

Knapsack

```
#include<bits/stdc++.h>
using namespace std;

int dp[2005][2005];
int c, n;
int p[2005],w[2005];

int knapsack(int i, int j)
{
   if(i<0 || j<=0) return 0;
   if(dp[i][j]!=-1)    return dp[i][j];
   int v1 = knapsack(i-1,j), v2=-1;
   if(w[i]<=j) v2 = p[i] + knapsack(i-1,j-w[i]);
   return dp[i][j] = max(v1, v2);
}

int main()
{
   cin>>c>>n;
   for(int i=0; i<n; i++) cin>>w[i]>>p[i];
   for(int i=0; i<2005; i++)
      for(int j=0; j<2005; j++)
         dp[i][j] =  -1;

   cout<<knapsack(n-1,c)<<endl;
   for(int i=0; i<=n; i++)
   {
        for(int j=0; j<=c; j++)
        {
            cout<<dp[i][j]<<" ";
        }
        cout<<endl;
    }
}
```

```
/*
4 5
1 8
2 4
3 0
2 5
2 3

-1  8  8 -1  8
-1  8  8 -1 12
-1 -1  8 -1 12
-1 -1  8 -1 13
-1 -1 -1 -1 13
-1 -1 -1 -1 -1
*/
```

Dijkstra

```
#include <iostream>
#include <vector>
#include <set>
#include <climits>

using namespace std;

const int INF = INT_MAX; // Represents infinity

// Custom data structure to represent an edge
struct Edge {
    int v, weight;
    Edge(int _v, int _weight) : v(_v), weight(_weight)
{}
};

// Dijkstra's algorithm function
void dijkstra(vector<vector<Edge>>& graph, int
source) {
    int V = graph.size(); // Number of vertices
    vector<int> distance(V, INF);
    set<pair<int, int>> pq; // Priority queue (min-heap)

    distance[source] = 0;
    pq.insert({0, source});

    while (!pq.empty()) {
        int u = pq.begin()->second;
        pq.erase(pq.begin());

        for (const Edge& edge : graph[u]) {
            int v = edge.v;
            int weight = edge.weight;

            // Relaxation step
            if (distance[u] + weight < distance[v]) {
                pq.erase({distance[v], v});
                distance[v] = distance[u] + weight;
                pq.insert({distance[v], v});
            }
        }
    }

    // Print the shortest distances from the source
    for (int i = 0; i < V; i++) {
        cout << "Shortest distance from " << source <<
" to " << i << ": " << distance[i] << endl;
    }
}

int main() {
    int V, E; // Number of vertices and edges
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;

    vector<vector<Edge>> graph(V);

    // Input the graph (edge details)
    for (int i = 0; i < E; i++) {
        int u, v, weight;
        cout << "Enter edge (u v weight): ";
        cin >> u >> v >> weight;
        graph[u].emplace_back(v, weight);
    }

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;

    dijkstra(graph, source);

    return 0;
}
```

Bellman ford

```
#include<bits/stdc++.h>
using namespace std;
```

```
struct edge {
    char v;
    int w;
};

struct node {
    char vertex;
    int d;
    node *parent;
    vector<edge> adj;
};

class Graph {
public:
    int V;
    node *nodes;

    Graph(int v) {
        V = v;
        nodes = new node[V];
        for (int i = 0; i < V; i++) {
            nodes[i].vertex = 'A' + i;
            nodes[i].d = INT_MAX;
            nodes[i].parent = NULL;
        }
    }

    void AddWeightedEdge(char u,
char n, int w) {
        edge e;
        e.v = n;
        e.w = w;
        for (int i = 0; i < V; i++) {
            if (nodes[i].vertex == u)
                nodes[i].adj.push_back(e);
        }
    }

    void printPath(node *s) {
        if (s->parent != NULL) {
            this->printPath(s->parent);
            cout << " ----> ";
        }
        cout << s->vertex << " ";
```

```
    }

    void
INITIALIZE_SINGLE_SOURCE(c
har s) {
        for (int i = 0; i < V; i++) {
            if (nodes[i].vertex == s)
                nodes[i].d = 0;
        }
    }

    int relax(char u, char v, int w) {
        int uIndex = u - 'A';
        int vIndex = v - 'A';
        if (nodes[uIndex].d !=
INT_MAX && nodes[uIndex].d +
w < nodes[vIndex].d) {
            nodes[vIndex].d =
nodes[uIndex].d + w;
            nodes[vIndex].parent =
&nodes[uIndex];
            return 1;
        }
        return 0;
    }
};

bool BELLMAN_FORD(Graph G,
char s) {

G.INITIALIZE_SINGLE_SOURC
E(s);
    for (int i = 0; i < G.V - 1; i++) {
        for (int u = 0; u < G.V; u++) {
            for (edge e : G.nodes[u].adj)
{

G.relax(G.nodes[u].vertex, e.v,
e.w);
            }
        }
    }
```

```
    // Check for negative weight
cycles
    for (int u = 0; u < G.V; u++) {
        for (edge e : G.nodes[u].adj) {
            if
(G.relax(G.nodes[u].vertex, e.v,
e.w))
                return false; // Negative
weight cycle detected
        }
    }

    return true; // No negative weight
cycle
}

int main() {
    Graph G(5);

    G.AddWeightedEdge('A', 'B', 3);
    G.AddWeightedEdge('A', 'C', 1);
    G.AddWeightedEdge('B', 'D', 1);
    G.AddWeightedEdge('B', 'E', 2);
    G.AddWeightedEdge('C', 'B', 1);
    G.AddWeightedEdge('C', 'D', 4);
    G.AddWeightedEdge('D', 'E', 3);

    if (BELLMAN_FORD(G, 'A')) {
        for (int i = 0; i < G.V; i++) {
            cout << "\nShortest Path
from A to " << G.nodes[i].vertex <<
endl;
            G.printPath(&G.nodes[i]);
            cout << " (Distance: " <<
G.nodes[i].d << ")" << endl;
        }
    } else {
        cout << "Negative Weighted
cycle present. No solution!!" <<
endl;
    }

    return 0;
}
```

LCS

```
//LCS in tabulation method
#include <iostream>
#include <vector>
#include <string>

using namespace std;
```

```
pair<vector<vector<int>
>,vector<vector<string> > >
LCS(string X, string Y) {
    int m = X.length();
    int n = Y.length();

    vector<vector<int> > c(m + 1,
vector<int>(n + 1, 0));
```

```
    vector<vector<string> > b(m + 1,
vector<string>(n + 1, ""));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (X[i - 1] == Y[j - 1]) {
                c[i][j] = c[i - 1][j - 1] + 1;
                b[i][j] = "";
            } else if (c[i - 1][j] >= c[i][j -
1]) {
                c[i][j] = c[i - 1][j];
                b[i][j] = "1";
            } else {
                c[i][j] = c[i][j - 1];
                b[i][j] = "2";
            }
        }
    }

    return make_pair(c, b);
}

void
printAllLCS(vector<vector<string>
>& b, string X, int i, int j, string
currentLCS) {
    if (i == 0 || j == 0) {
        cout << currentLCS << endl;
        return;
    }
    if (b[i][j] == "") {
        printAllLCS(b, X, i - 1, j - 1,
X[i - 1] + currentLCS);
    } else if (b[i][j] == "1") {
        printAllLCS(b, X, i - 1, j,
currentLCS);
    } else {
        printAllLCS(b, X, i, j - 1,
currentLCS);
    }
}

int main() {
        string X, Y;
        cout << "Enter the first string:
";
        cin >> X;
        cout << "Enter the second
string: ";
        cin >> Y;
        pair<vector<vector<int> >,
vector<vector<string> > > result =
LCS(X, Y);
        vector<vector<int> >& c =
result.first;
        vector<vector<string> >& b =
result.second;

        int lengthOfLCS =
c[X.length()][Y.length()];
        cout << "Length of Longest
Common Subsequence: " <<
lengthOfLCS << endl;

        cout << "Longest Common
Subsequences:" << endl;
        printAllLCS(b, X, X.length(),
Y.length(), "");

        return 0;


}
```

Problem sheet by Ramisa Maam

Priority queue

```
/*
You're managing an emergency room, and patients arrive with different levels of urgency.
Let's say we have a list of patients with their names and levels of urgency. The urgency level
is a numerical value, where a higher number indicates a more critical condition. Devise a
solution using an appropriate data structure to process these patients in order of urgency by modifying your recent assignment.
Input
Number of Patients: 5
Name of Patient 1: John
Urgency Level of Patient 1: 3
Name of Patient 2: Sarah
Urgency Level of Patient 2: 5
Name of Patient 3: Emily
Urgency Level of Patient 3: 4
Name of Patient 4: Michael
Urgency Level of Patient 4: 2
Name of Patient 5: David
Urgency Level of Patient 5: 1

Output
Emergency Room Treatment Order:
1. Sarah - Urgency Level: 5 (Most Critical)
2. Emily - Urgency Level: 4
3. John - Urgency Level: 3
4. Michael - Urgency Level: 2
5. David - Urgency Level: 1 (Least Critical)

Input
Number of Patients: 3
Name of Patient 1: John
Urgency Level of Patient 1: 3
Name of Patient 2: Sarah
Urgency Level of Patient 2: 1
Name of Patient 3: Michael
Urgency Level of Patient 3: 2

Output
Emergency Room Treatment Order:
1. John - Urgency Level: 3 (Most Critical)
2. Michael - Urgency Level: 2
3. Sarah - Urgency Level: 1 (Least Critical)

Note: To simplify the problem, assume that all urgency levels are distinct.
*/

#include <iostream>
#include <queue>
#include <string>

using namespace std;

// Custom data structure to represent a patient
struct Patient {
    string name;
    int urgency;

    // Define comparison operator for max-heap
    bool operator<(const Patient& other) const {
        return urgency < other.urgency; // Max-heap order
    }
};

int main() {
    int numPatients;
    cout << "Number of Patients: ";
    cin >> numPatients;

    // Create a max-heap to store patients based on their urgency
    priority_queue<Patient> emergencyRoom;

    for (int i = 0; i < numPatients; i++) {
        cout << "Name of Patient " << i + 1 << ": ";
        string name;
        cin >> name;

        cout << "Urgency Level of Patient " << i + 1 << ": ";
        int urgency;
        cin >> urgency;

        // Create a patient and add to the max-heap
        emergencyRoom.push({name, urgency});
    }

    cout << "Emergency Room Treatment Order:" << endl;
    int order = 1;

    while (!emergencyRoom.empty()) {
        Patient patient = emergencyRoom.top();
        emergencyRoom.pop();
        cout << order << ". " << patient.name << " - Urgency Level: " << patient.urgency;
        if (order == 1) {
            cout << " (Most Critical)";
        } else if (order == numPatients) {
            cout << " (Least Critical)";
        }
        cout << endl;
        order++;
    }

    return 0;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
Facetook is a well known social network website, and it will launch a new feature called Facetook Priority Wall. This feature will sort all posts from your friends according to the priority factor (it will be described).

This priority factor will be affected by three types of actions:

1. "X posted on Y's wall" (15 points),
2. "X commented on Y's post" (10 points),
3. "X likes Y's post" (5 points).
X and Y will be two distinct names. And each action will increase the priority factor between X and Y (and vice versa) by the above value of points (the priority factor between X and Y is the same as the priority factor between Y and X).

You will be given n actions with the above format (without the action number and the number of points), and you have to print all the distinct names in these actions sorted according to the priority factor with you.

Input
The first line contains your name. The second line contains an integer n, which is the number of actions (1 ≤ n ≤ 100). Then n lines follow, it is guaranteed that each one contains exactly 1 action in the format given above. There is exactly one space between each two words in a line, and there are no extra spaces. All the letters are lowercase. All names in the input will consist of at least 1 letter and at most 10 small Latin letters.

Output
Print m lines, where m is the number of distinct names in the input (excluding yourself). Each line should contain just 1 name. The names should be sorted according to the priority factor with you in the descending order (the highest priority factor should come first). If two or more names have the same priority factor, print them in the alphabetical (lexicographical) order.

Note, that you should output all the names that are present in the input data (excluding yourself), even if that person has a zero priority factor.

The lexicographical comparison is performed by the standard "<" operator in modern programming languages. The line a is lexicographically smaller than the line b, if either a is the prefix of b, or if exists such an i (1 ≤ i ≤ min(|a|, |b|)), that ai < bi, and for any j (1 ≤ j < i) aj = bj, where |a| and |b| stand for the lengths of strings a and b correspondently.

Examples
inputCopy
ahmed
3
ahmed posted on fatma's wall
fatma commented on ahmed's post
mona likes ahmed's post
outputCopy
fatma
mona
inputCopy
aba
1
likes likes posted's post
outputCopy
likes
posted
*/

#include <iostream>
#include <string>
#include <map>
#include <set>
#include <vector>
#include <sstream>
#include <algorithm>

using namespace std;

int main() {
    string yourName;
    cin >> yourName;

    int n;
    cin >> n;
    cin.ignore(); // Consume the
newline character

    map<string, int> priorities;
    map<string, set<string>>
actions;

    for (int i = 0; i < n; i++) {
        string action;
        getline(cin, action);

        stringstream ss(action);
        string X, act, on, Y;
        ss >> X >> act >> on >>
Y;

        if (act == "posted") {
            priorities[Y] += 15;
            actions[Y].insert(Y);
        } else if (act ==
"commented") {
            priorities[Y] += 10;
            actions[Y].insert(X);
        } else {
            priorities[Y] += 5;
            actions[Y].insert(X);
        }
    }

    vector<pair<int, string>>
sortedNames;

    for (const auto& it : actions)
{
        if (it.first != yourName) {

sortedNames.push_back({-
priorities[it.first], it.first});
        }
    }

    sort(sortedNames.begin(),
sortedNames.end());

    sort(sortedNames.begin(),
sortedNames.end(), [](const
pair<int, string>& a, const
pair<int, string>& b) {
        if (a.first == b.first) {
            return a.second <
b.second;
        }
        return a.first > b.first;
    });

    for (const auto& name :
sortedNames) {
        cout << name.second <<
endl;
    }

    return 0;
}
```

++

/*
In computer science, a priority queue is an abstract data type which is like a regular queue, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. - Wikipedia

In this problem we will test your knowledge on Java Priority Queue.

There are a number of students in a school who wait to be served. Two types of events, ENTER and SERVED, can take place which are described below.

ENTER: A student with some priority enters the queue to be served.
SERVED: The student with the highest priority is served (removed) from the queue.
A unique id is assigned to each student entering the queue. The queue serves the students based on the following criteria (priority criteria):

The student having the highest Cumulative Grade Point Average (CGPA) is served first.
Any students having the same CGPA will be served by name in ascending case-sensitive alphabetical order.
Any students having the same CGPA and name will be served in ascending order of the id.
Create the following two classes:

The Student class should implement:
The constructor Student(int id, String name, double cgpa).
The method int getID() to return the id of the student.
The method String getName() to return the name of the student.
The method double getCGPA() to return the CGPA of the student.
The Priorities class should implement the method List<Student> getStudents(List<String> events) to process all the given events and return all the students yet to be served in the priority order.
Input Format

The first line contains an integer, , describing the total number of events. Each of the subsequent lines will be of the following two forms:

ENTER name CGPA id: The student to be inserted into the priority queue.
SERVED: The highest priority student in the queue was served.
The locked stub code in the editor reads the input and tests the correctness of the Student and Priorities classes implementation.

Constraints
Constraints
2 ≤ n ≤ 1000
0 ≤ CGPA ≤ 4.00
1 ≤ id ≤105
2≤ |name| ≤ 30

Output Format

The locked stub code prints the names of the students yet to be served in the priority order. If there are no such student, then the code prints EMPTY.

Sample Input 0

```
12
ENTER John 3.75 50
ENTER Mark 3.8 24
ENTER Shafaet 3.7 35
SERVED
SERVED
ENTER Samiha 3.85 36
SERVED
ENTER Ashley 3.9 42
ENTER Maria 3.6 46
ENTER Anik 3.95 49
ENTER Dan 3.95 50
SERVED
```

Sample Output 0

```
Dan
Ashley
Shafaet
Maria
```

Explanation 0
In this case, the number of events is 12. Let the name of the queue be Q.

John is added to Q. So, it contains (John, 3.75, 50).
Mark is added to Q. So, it contains (John, 3.75, 50) and (Mark, 3.8, 24).
Shafaet is added to Q. So, it contains (John, 3.75, 50), (Mark, 3.8, 24), and (Shafaet, 3.7, 35).
Mark is served as he has the highest CGPA. So, Q contains (John, 3.75, 50) and (Shafaet, 3.7, 35).
John is served next as he has the highest CGPA. So, Q contains (Shafaet, 3.7, 35).
Samiha is added to Q. So, it contains (Shafaet, 3.7, 35) and (Samiha, 3.85, 36).
Samiha is served as she has the highest CGPA. So, Q contains (Shafaet, 3.7, 35).
Now, four more students are added to Q. So, it contains (Shafaet, 3.7, 35), (Ashley, 3.9, 42), (Maria, 3.6, 46), (Anik, 3.95, 49), and (Dan, 3.95, 50).
Anik is served because though both Anil and Dan have the highest CGPA but Anik comes first when sorted in alphabetic order. So, Q contains (Dan, 3.95, 50), (Ashley, 3.9, 42), (Shafaet, 3.7, 35), and (Maria, 3.6, 46).
As all events are completed, the name of each of the remaining students is printed on a new line.

Generate the code in c++, make sure to get it accepted in vjudge.
*/

```
#include <iostream>
#include <queue>
#include <string>
#include <vector>
#include <sstream>
using namespace std;

int main() {
    int totalEvents;
    cin >> totalEvents;
    cin.ignore();  // Consume
the newline character


priority_queue<pair<double,
pair<string, int>>> pq;

    vector<string> events;
    for (int i = 0; i <
totalEvents; i++) {
        string event;
        cin.ignore();  // Consume
the newline character
        getline(cin, event);
        events.push_back(event);
    }

    for (int i = 0; i <
totalEvents; i++) {
        stringstream ss(events[i]);
        string cmd, name;
        double cgpa;
        int id;
        ss >> cmd;
        if (cmd == "SERVED") {
            if (!pq.empty())
pq.pop();
        } else {
            ss >> name >> cgpa >>
id;
            pq.push({-cgpa,
{name, id}});
        }
    }
    if (pq.empty()) {
        cout << "EMPTY" <<
endl;
    } else {
        vector<pair<double,
pair<string, int>>> students;
        while (!pq.empty()) {

students.push_back(pq.top());
            pq.pop();
        }
        for (int i = students.size()
- 1; i >= 0; i--) {
            cout <<
students[i].second.first <<
endl;
        }
    }
    return 0;

}
```

++

TRIE

```
/*
Given a list of phone numbers,
determine if it is consistent in
the sense that
no number is the prefix of
another. Let’s say the phone
catalogue listed these
numbers:
• Emergency 911
• Alice 97 625 999
• Bob 91 12 54 26
In this case, it’s not possible to
call Bob, because the central
would direct
your call to the emergency line
as soon as you had dialled the
first three digits of
Bob’s phone number. So this
list would not be consistent.
Input
The first line of input gives a
single integer, 1 ≤ t ≤ 40, the
number of test cases. Each test
case starts
with n, the number of phone
numbers, on a separate line, 1
≤ n ≤ 10000. Then follows n
lines with
one unique phone number on
each line. A phone number is a
sequence of at most ten digits.
Output
For each test case, output
‘YES’ if the list is consistent,
or ‘NO’ otherwise.
Sample Input
2
3
911
97625999
91125426
5
113
12340
123440
12345
98346
Sample Output
NO
YES
*/
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

struct TrieNode {
    bool isEnd;
    TrieNode* children[10];

    TrieNode() {
        isEnd = false;
        for (int i = 0; i < 10; i++)
{
            children[i] = nullptr;
        }
    }
};

bool insert(TrieNode* root,
const string& phone) {
    TrieNode* current = root;
    for (char digit : phone) {
        int index = digit - '0';
        if (current-
>children[index] == nullptr) {
            current-
>children[index] = new
TrieNode();
        }
        current = current-
>children[index];
        if (current->isEnd) {
            return false; // Prefix
found
        }
    }
    current->isEnd = true;
    for (int i = 0; i < 10; i++) {
        if (current->children[i]) {
            return false; // More
digits in this number
        }
    }
    return true; // Successfully
inserted
}

bool
isConsistent(vector<string>&
phoneNumbers) {
    TrieNode* root = new
TrieNode();
    sort(phoneNumbers.begin(),
phoneNumbers.end());
    for (const string& phone :
phoneNumbers) {
        if (!insert(root, phone)) {
            return false;
        }
    }
    return true;
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        int n;
        cin >> n;
        vector<string>
phoneNumbers(n);
        for (int i = 0; i < n; i++) {
            cin >>
phoneNumbers[i];
        }
        if
(isConsistent(phoneNumbers))
{
            cout << "YES" <<
endl;
        } else {
            cout << "NO" << endl;
        }
    }
    return 0;
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
Mr. A, a faculty member of
CSE department, MIST, has
been noticing that there are
some
students who have been
consistently late to his classes
for a few days. Hence, he has
decided to
keep a Late student list to note
down the name of the students
being late.
After listing the names of the
students for three consecutive
days he has decided to call
upon the
students who have been late to
two or more of his classes and
has requested that they submit
written explanations for their
tardiness. Though Mr. A
initially decided to penalize
these
students, after receiving the
students' explanations, he has
decided to be lenient and give
the
students a second chance.
Consequently, he has removed
the names of these students
from the
late student list.
Now , help Mr. A by
implementing this Late
Students List using TRIE Data
structure.
Input
Name of the students who
were late in the class:
Day 1:
Number of Late Students: 7
Enter Names:
JOHN
SARAH
EMILY
MICHAEL
DAVID
SARA
EMA
Day 2:
Number of Late Students: 4
Enter Names:
EMA
DAVID
SARAH
ARAF
Day 3:
Number of Late Students: 2
Enter Names:
FARAH
SARAH

Output
List of students who were late
in two or more
classes:
DAVID
EMA
SARAH
After deleting these names, list
of the late
students in Lexicographical
order:
ARAF
EMILY
FARAH
JOHN
MICHAEL
SARA
*/

#include <iostream>
#include <vector>
#include <map>
using namespace std;

struct TrieNode {
    map<char, TrieNode*>
children;
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
    }
};

void insert(TrieNode* root,
const string& word) {
    TrieNode* node = root;
    for (char c : word) {
        if (node->children.find(c)
== node->children.end()) {
            node->children[c] =
new TrieNode();
        }
        node = node-
>children[c];
    }
    node->isEndOfWord = true;
}
```

```
bool search(TrieNode* root,
const string& word) {
   TrieNode* node = root;
   for (char c : word) {
      if (node->children.find(c)
== node->children.end()) {
         return false;
      }
      node = node-
>children[c];
   }
   return node->isEndOfWord;
}

int main() {
   TrieNode* root = new
TrieNode();

   int totalDays;
   cin >> totalDays;

   vector<string> lateStudents;
   vector<string>
secondChanceStudents;

   for (int day = 1; day <=
totalDays; day++) {
      int n;
      cin >> n;
      vector<string> names(n);

      for (int i = 0; i < n; i++) {
         cin >> names[i];
         insert(root, names[i]);
      }

      for (const string& name :
names) {
         if (search(root, name))
{

lateStudents.push_back(name)
;
         } else {

secondChanceStudents.push_b
ack(name);
         }
      }
   }

   cout << "List of students
who were late in two or more
classes:" << endl;
   for (const string& student :
lateStudents) {
      cout << student << endl;
   }

   cout << "After deleting
these names, list of the late
students in Lexicographical
order:" << endl;

sort(secondChanceStudents.be
gin(),
secondChanceStudents.end());
   for (const string& student :
secondChanceStudents) {
      cout << student << endl;
   }

   return 0;
}
```

++

Knapsack

```
/*
The famous knapsack
problem. You are packing for a
vacation on the sea side and
you are going to carry only
one bag with capacity S (1 <=
S <= 2000). You also have N
(1<= N <= 2000) items that
you might want to take with
you to the sea side.
Unfortunately you can not fit
all of them in the knapsack so
you will have to choose. For
each item you are given its
size and its value. You want to
maximize the total value of all
the items you are going to
bring. What is this maximum
total value?

Input
On the first line you are given
S and N. N lines follow with
two integers on each line
describing one of your items.
The first number is the size of
the item and the next is the
value of the item.

Output
You should output a single
integer on one like - the total
maximum value from the best
choice of items for your trip.

Example
Input
4 5
1 8
2 4
3 0
2 5
2 3
Output
13
*/
#include <iostream>
#include <vector>
using namespace std;

int knapsack(int S, int N,
vector<int>& sizes,
vector<int>& values) {
   vector<vector<int>> dp(N +
1, vector<int>(S + 1, 0));

   for (int i = 1; i <= N; i++) {
      for (int size = 0; size <=
S; size++) {
         dp[i][size] = dp[i -
1][size]; // Initialize with the
previous row's value.
```

```
            if (sizes[i - 1] <= size)
{
                dp[i][size] =
max(dp[i][size], dp[i - 1][size -
sizes[i - 1]] + values[i - 1]);
            }
        }
    }

    return dp[N][S];
}

int main() {
    int S, N;
    cin >> S >> N;
    vector<int> sizes(N);
    vector<int> values(N);

    for (int i = 0; i < N; i++) {
        cin >> sizes[i] >>
values[i];
    }

    int result = knapsack(S, N,
sizes, values);
    cout << result << endl;

    return 0;
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*
Taro's summer vacation starts
tomorrow, and he has decided
to make plans for it now.

The vacation consists of N
days. For each i (1≤i≤N), Taro
will choose one of the
following activities and do it
on the i-th day:

A: Swim in the sea. Gain ai
points of happiness.
B: Catch bugs in the
mountains. Gain bi points of
happiness.
C: Do homework at home.
Gain ci points of happiness.
As Taro gets bored easily, he
cannot do the same activities
for two or more consecutive
days.
Find the maximum possible
total points of happiness that
Taro gains.

Constraints
All values in input are
integers.
1≤N≤10
1≤ai, bi, ci≤10^4

Input
Input is given from Standard
Input in the following format:


N
a1 b1 c1
a2 b2 c2
. . .
. . .
. . .
aN bN cN
Output
Print the maximum possible
total points of happiness that
Taro gains.

Sample 1
Inputcopy
3
10 40 70
20 50 80
30 60 90
210
If Taro does activities in the
order C, B, C, he will gain
70+50+90=210 points of
happiness.

Sample 2
Inputcopy
1
100 10 1
Outputcopy
100
Sample 3
Inputcopy
7
6 7 8
8 8 3
2 5 2
7 8 6
4 6 8
2 3 4
7 5 1
Outputcopy
46
Taro should do activities in the
order C, A, B, A, C, B, A.
*/
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int N;
    cin >> N;

    vector<vector<int>>
happiness(N, vector<int>(3));
    vector<vector<int>> dp(N,
vector<int>(3, 0));

    for (int i = 0; i < N; i++) {
        cin >> happiness[i][0] >>
happiness[i][1] >>
happiness[i][2];
    }

    for (int i = 0; i < 3; i++) {
        dp[0][i] =
happiness[0][i];
    }

    for (int i = 1; i < N; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3;
k++) {
                if (j != k) {
                    dp[i][j] =
max(dp[i][j], dp[i - 1][k] +
happiness[i][j]);
```

```
                }
            }
        }
    }
    int maxHappiness =
max(dp[N - 1][0], max(dp[N -
1][1], dp[N - 1][2]));
    cout << maxHappiness <<
endl;

    return 0;
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
Given an array of integers and
a target sum, determine the
sum nearest to but not
exceeding the target that can
be created. To create the sum,
use any element of your array
zero or more times.
For example, if arr = [2, 3, 4]
and your target sum is 10, you
might select [2, 2, 2, 2, 2], [2,
2, 3, 3] or [3, 3, 3, 1]. In this
case, you can arrive at exactly
the target.

Function Description
Complete the
unboundedKnapsack function
in the editor below. It must
return an integer that
represents the sum nearest to
without exceeding the target
value.
unboundedknapsack has the
following parameter(s):
  k: an integer
  arr: an array of integers

Input Format
The first line contains an
integer t, the number of test
cases,
Each of the next t pairs of lines
are as follows:
-The first line contains two
integers n and k, the length of
arr and the target sum. -The
second line contains n space
separated integers arr[i].

Constraints
1 ≤ t≤ N
1≤ n, k, arr[i] ≤ 2000

Output Format
Print the maximum sum for
each test case which is as near
as possible, but not exceeding,
to the target sum on a separate
line.
Inputcopy
2
3 12
1 6 9
5 9
3 4 4 4 8
Outputcopy
12
9
Explanation
In the first test case, one can
pick {6, 6}. In the second, we
can pick {3,3,3}.
*/
#include <iostream>
#include <vector>
using namespace std;

int unboundedKnapsack(int k,
vector<int>& arr) {
    int n = arr.size();
    vector<int> dp(k + 1, 0);

    for (int i = 1; i <= k; i++) {
        for (int j = 0; j < n; j++) {
            if (arr[j] <= i) {
                dp[i] = max(dp[i],
dp[i - arr[j]] + arr[j]);
            }
        }
    }

    return dp[k];
}

int main() {
    int t;
    cin >> t;

    while (t--) {
        int n, k;
        cin >> n >> k;
        vector<int> arr(n);

        for (int i = 0; i < n; i++) {
            cin >> arr[i];
        }

        int result =
unboundedKnapsack(k, arr);
        cout << result << endl;
    }

    return 0;
}
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
Entering into the cave with
treasures, Aladdin did not take
an old blackened lamp. He
rushed to
collect the gold coins and
precious stones into his
knapsack. He would, of
course, take
everything, but miracles do not
happen - too much weight the
knapsack can not hold. Many
times he laid out one thing and
put others in their place, trying
```

to raise the value of the jewels as
high as possible.

Now, help Aladdin to determine the maximum value of weight that Aladdin can put in his
knapsack.

We will assume that in the cave there are objects of n different types, the number of objects of
each type is not limited. That means, an item can be taken multiple times. The maximum weight that a knapsack can hold is s. Each item of type i has the weight wi and cost
vi (i = 1, 2, ... , n).

Input data:
First line contains two integers s and n (1≤ s ≤250, 1 ≤n ≤35) — the maximum possible weight of items in the knapsack and the number of types of items. Each of the next n lines
contains two numbers wi and vi (1≤ wi ≤ 250, 1 ≤ vi ≤250) — the weight of an item of type i
and its cost.

Output data:
Print the maximum value of the loading, which weight does not exceed s.

Input
Knapsack size and number of item:
10 2
Weight and value of each item:
5 10
6 19
Output
Maximum profit: 20

```
*/

#include <iostream>
#include <vector>
using namespace std;

int knapsack(int s, int n, vector<pair<int, int> >& items) {
    vector<int> dp(s + 1, 0);

    for (int i = 0; i < n; i++) {
        int weight = items[i].first;
        int value = items[i].second;
        for (int j = weight; j <= s; j++) {
            dp[j] = max(dp[j], dp[j - weight] + value);
        }
    }

    return dp[s];
}

int main() {
    int s, n;
    cin >> s >> n;
    vector<pair<int, int> > items(n);

    for (int i = 0; i < n; i++) {
        int weight, value;
        cin >> weight >> value;
        items[i] = make_pair(weight, value);
    }

    int result = knapsack(s, n, items);
    cout << "Maximum profit: " << result << endl;

    return 0;
}
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
/*
```

Vasya is going to hike with fellow programmers and decided to take a responsible approach to the choice of what he will take with him. Vasya has n things that he could take with him in his knapsack. Every thing weighs 1 kilogram. Things have different "usefulness" for Vasya.

The hiking is going to be very long, so Vasya would like to carry a knapsack of weight no more than w kilo.

Help him to determine the total "usefulness" of things in his knapsack if the weight of backpack can be no more than w kilo.

Input data
The first line contains integers w и n (1 ≤ w, n ≤ 20). The second line contains n integers c[i] (1 ≤ c[i] ≤ 1000) - the "usefulness" for each thing.

Output data
Print the total "usefulness" of things that Vasya can take with him.

Examples
Inputcopy
2 3
1 5 3
Outputcopy
8
Inputcopy
3 2
3 2
Outputcopy
5

```
*/

#include <iostream>
#include <vector>
using namespace std;

int knapsack(int w, int n,
vector<int>& c) {
    vector<vector<int>> dp(n +
1, vector<int>(w + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int weight = 1;
weight <= w; weight++) {
            if (weight >= i) {
                dp[i][weight] =
max(dp[i - 1][weight], dp[i -
1][weight - i] + c[i - 1]);
            } else {
                dp[i][weight] = dp[i
- 1][weight];
            }
        }
    }

    return dp[n][w];
}

int main() {
    int w, n;
    cin >> w >> n;
    vector<int> c(n);

    for (int i = 0; i < n; i++) {
        cin >> c[i];
    }

    int result = knapsack(w, n,
c);
    cout << result << endl;

    return 0;
}
```

+++

LCS

```
/*
You are given strings s and t.
Find one longest string that is
a subsequence of both s and t.

Notes
A subsequence of a string x is
the string obtained by
removing zero or more
characters from x and
concatenating the remaining
characters without changing
the order.

Constraints s and t are strings
consisting of lowercase
English letters. 1≤|s|,|t|≤3000
Input
Input is given from Standard
Input in the following format:
s
t

Output
Print one longest string that is
a subsequence of both s and t.
If there are multiple such
strings, any of them will be
accepted.

Sample 1
Input
axyb
abyxb
Output
axb
The answer is axb or ayb;
either will be accepted.
Sample 2
Input
aa
xayaz
Output
aa
Sample 3
Input
a
z
Output
The answer is (an empty
string).
Sample 4
Input
abracadabra
avadakedavra
Output
aaadara
*/
#include <iostream>
#include <vector>
#include <string>
using namespace std;

string
findLongestCommonSubseque
nce(string s, string t) {
    int m = s.length();
    int n = t.length();

    // Create a table to store the
lengths of longest common
subsequences
    vector<vector<int>> dp(m
+ 1, vector<int>(n + 1, 0));

    // Fill the table using
dynamic programming
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++)
{
            if (s[i - 1] == t[j - 1]) {
                dp[i][j] = dp[i - 1][j -
1] + 1;
            } else {
                dp[i][j] = max(dp[i -
1][j], dp[i][j - 1]);
            }
        }
    }

    // Reconstruct the longest
common subsequence
    int length = dp[m][n];
    string lcs(length, ' ');

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (s[i - 1] == t[j - 1]) {
            lcs[length - 1] = s[i -
1];
            i--;
            j--;
            length--;
        } else if (dp[i - 1][j] >
dp[i][j - 1]) {
            i--;
        } else {
            j--;
        }
    }

    return lcs;
}

int main() {
    string s, t;
    cin >> s >> t;

    string lcs =
findLongestCommonSubseque
nce(s, t);

    cout << lcs << endl;

    return 0;
}
```

+++

```
/*
A subsequence of a given
sequence is the given sequence
with some elements (possible
none) left out. Given a
sequence X = <x1, x2, ..., xm>
another sequence Z = <z1, z2,
..., zk> is a subsequence of X
if there exists a strictly
increasing sequence <i1, i2, ...,
ik> of indices of X such that
for all j = 1,2,...,k, xij = zj. For
example, Z = <a, b, f, c> is a
subsequence of X = <a, b, c, f,
b, c> with index sequence <1,
2, 4, 6>. Given two sequences
X and Y the problem is to find
the length of the maximum-
length common subsequence
of X and Y.
The program input is from a
text file. Each data set in the
file contains two strings
representing the given
sequences. The sequences are
separated by any number of
white spaces. The input data
are correct. For each set of
data the program prints on the
standard output the length of
the maximum-length common
subsequence from the
beginning of a separate line.
Input
abcfbc abfcab
programming contest
abcd mnp
Output
4
2
0
Sample
Input
abcfbc abfcab
programming contest
abcd mnp
Output
4
2
0
*/
#include <iostream>
#include <vector>
#include <string>
using namespace std;

pair<vector<vector<int>,
vector<vector<string>>>
LCS(string X, string Y) {
    int m = X.length();
    int n = Y.length();

    vector<vector<int>> c(m +
1, vector<int>(n + 1, 0));
    vector<vector<string>> b(m
+ 1, vector<string>(n + 1, ""));

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++)
{
            if (X[i - 1] == Y[j - 1])
{
                c[i][j] = c[i - 1][j - 1]
+ 1;
                b[i][j] = "";
            } else if (c[i - 1][j] >=
c[i][j - 1]) {
                c[i][j] = c[i - 1][j];
                b[i][j] = "1";
            } else {
                c[i][j] = c[i][j - 1];
                b[i][j] = "2";
            }
        }
    }

    return make_pair(c, b);
}

int main() {
    string X, Y;
    while (cin >> X >> Y) {
        vector<vector<int> > c;
        vector<vector<string>>
b;
        tie(c, b) = LCS(X, Y);

        int length =
c[X.length()][Y.length()];
        cout << length << endl;
    }

    return 0;
}
```

++

BF

```
/*
Given a directed graph, that can contain multiple edges and loops. Each edge has a weight that is expressed by a number (possibly negative). It is guaranteed that there are no cycles of negative weight.

Calculate the length of the shortest paths from the vertex number 1 to all other vertices.

Input data
First the number of vertices n (1 ≤ n ≤ 100) is given. It is followed by the number of edges m (0 ≤ m ≤ 10000). Next m triples describe the edges: beginning of the edge, the end of the edge and its weight (an integer from -100 to 100).

Output data
Print n numbers - the distance from the vertex number 1 to all other vertices of the graph. If the path to the corresponding vertex does not exist, instead of the path length print the number 30000.

Examples
Input example #1
content_copy
4 5
1 2 10
2 3 10
1 3 100
3 1 -10
2 3 1
Output example #1
content_copy
0 10 11 30000
*/
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
struct Edge {
  int from, to, weight;
};

void bellmanFord(int n, int m, vector<Edge>& edges, vector<int>& distances) {
  const int INF = 30000;

  distances[0] = 0; // The source vertex has a distance of 0

  // Relaxation step for (n-1) times
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < m; j++) {
      if (distances[edges[j].from - 1] < INF) {
        if (distances[edges[j].to - 1] > distances[edges[j].from - 1] + edges[j].weight) {
distances[edges[j].to - 1] = distances[edges[j].from - 1] + edges[j].weight;
        }
      }
    }
  }

  // Check for negative weight cycles
  for (int i = 0; i < m; i++) {
    if (distances[edges[i].from - 1] < INF) {
      if (distances[edges[i].to - 1] > distances[edges[i].from - 1] + edges[i].weight) {
        distances[edges[i].to - 1] = -INF; // Indicates a negative cycle
      }
    }
  }
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<Edge> edges(m);

  for (int i = 0; i < m; i++) {
    cin >> edges[i].from >> edges[i].to >> edges[i].weight;
  }

  vector<int> distances(n, 30000); // Initialize distances to a large value

  bellmanFord(n, m, edges, distances);

  for (int i = 0; i < n; i++) {
    if (distances[i] == 30000) {
      cout << "30000 ";
    } else if (distances[i] == -30000) {
      cout << "-1 ";
    } else {
      cout << distances[i] << " ";
    }
  }

  cout << endl;

  return 0;
}
```

++

Quicksort

```
/*

Given an integer array nums,
return an integer array count
where count[i] is the number
of
smaller elements of the
nums[i].

Input Output

nums = [5,2,6,1] [2,1,3,0]
nums = [-1] [0]
nums = [-1,-1] [0,0]

* Consider you have enough
resources so you don't need to
take the risk of having higher
time complexity than
O(nlogn).*/
#include <iostream>
#include <vector>
#include <utility>

using namespace std;

// Function to perform merge
step and count smaller
elements
vector<int>
mergeAndCountSmaller(vecto
r<int>& nums,
vector<pair<int, int>>&
indices, int low, int high,
vector<int>& smallerCounts)
{
    if (low >= high) return
indices;

    int mid = low + (high - low)
/ 2;

mergeAndCountSmaller(nums
, indices, low, mid,
smallerCounts);

mergeAndCountSmaller(nums
, indices, mid + 1, high,
smallerCounts);

    vector<pair<int, int>>
merged;
    int i = low;
    int j = mid + 1;
    int rightCount = 0;

    for (; i <= mid; i++) {
        while (j <= high &&
nums[indices[i].second] >
nums[indices[j].second]) {
            j++;
            rightCount++;
        }

smallerCounts[indices[i].secon
d] += rightCount;
    }

    i = low;
    j = mid + 1;
    while (i <= mid && j <=
high) {
        if
(nums[indices[i].second] <=
nums[indices[j].second]) {

merged.push_back(indices[i]);
            i++;
        } else {

merged.push_back(indices[j]);
            j++;
        }
    }

    while (i <= mid) {

merged.push_back(indices[i]);
        i++;
    }
    while (j <= high) {

merged.push_back(indices[j]);
        j++;
    }

    for (i = low; i <= high; i++)
{
        indices[i] = merged[i -
low];
    }

    return indices;
}

vector<int>
countSmaller(vector<int>&
nums) {
    int n = nums.size();
    vector<int>
smallerCounts(n, 0);

    vector<pair<int, int>>
indices;
    for (int i = 0; i < n; i++) {

indices.push_back({nums[i],
i});
    }


mergeAndCountSmaller(nums
, indices, 0, n - 1,
smallerCounts);

    return smallerCounts;
}

int main() {
    vector<int> nums1 = {5, 2,
6, 1};
    vector<int> result1 =
countSmaller(nums1);
    for (int count : result1) {
        cout << count << " ";
    }
    cout << endl;

    vector<int> nums2 = {-1};
    vector<int> result2 =
countSmaller(nums2);
    for (int count : result2) {
        cout << count << " ";
    }
    cout << endl;

    vector<int> nums3 = {-1, -
1};
    vector<int> result3 =
countSmaller(nums3);
    for (int count : result3) {
        cout << count << " ";
    }
    cout << endl;

    return 0;
}
```