**LIS:**

#include <bits/stdc++.h>

**using** **namespace** std;

#define ll long long int

**int** find\_lis(vector<**int**> a)

{

vector<**int**> dp;

**for** (**int** i : a)

{

**int** pos = lower\_bound(dp.begin(), dp.end(), i) - dp.begin();

**if** (pos == dp.size())

{

dp.push\_back(i);

}

**else**

{

dp[pos] = i;

}

}

**return** dp.size();

}

**int** main()

{

**int** n;

cin >> n;

vector<**int**> num(n);

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

{

cin >> num[i];

}

**int** ans = find\_lis(num);

cout << ans << endl;

**return** 0;

}

**0/1 Knapsack:**

#include <iostream>

#include <vector>

**using** **namespace** std;

#define ll long long int

**int** main()

{

**int** n, ks;

cin >> n >> ks;

vector<**int**> item(n), price(n);

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

{

cin >> item[i];

}

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

{

cin >> price[i];

}

vector<vector<**int**>> ans(n + 1, vector<**int**>(ks + 1, 0));

**for** (**int** i = 1; i <= n; i++)

{

**for** (**int** j = 0; j <= ks; j++)

{

**if** (j < item[i - 1])

{

ans[i][j] = ans[i - 1][j];

}

**else**

{

ans[i][j] = max(ans[i - 1][j], ans[i - 1][j - item[i - 1]] + price[i - 1]);

}

}

}

ll cnt = 0;

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

{

cnt += price[i];

}

**if** (cnt == ans[n][ks])

{

cout << "My King, I am successful in capturing the big fish. Immortality is few steps away." << endl;

}

**else**

{

cout << "My King, I have captured " << ans[n][ks] << " followers till now and I need more soldiers asap." << endl;

}

**return** 0;

}

**BFS Traversal:**

#include <bits/stdc++.h>

**using** **namespace** std;

**void** bfs(vector<vector<**int**>> &adjMatrix, vector<**bool**> &visited, vector<**int**> &ans, **int** vertex, **int** start)

{

queue<**int**> q;

q.push(start);

visited[start] = **true**;

**while** (!q.empty())

{

**int** frontNode = q.front();

q.pop();

ans.push\_back(frontNode);

**for** (**int** i = 1; i <= vertex; i++)

{

**if** (adjMatrix[frontNode][i] == 1 && !visited[i])

{

q.push(i);

visited[i] = **true**;

}

}

}

}

vector<**int**> BFS(vector<vector<**int**>> &adjMatrix, **int** vertex, **int** start)

{

vector<**int**> ans;

vector<**bool**> visited(vertex+1, **false**);

bfs(adjMatrix, visited, ans, vertex, start);

**return** ans;

}

**int** main()

{

**int** vertex;

cin >> vertex;

vector<vector<**int**>> adjMatrix(vertex+1, vector<**int**>(vertex+1, 0));

// Input the adjacency matrix

**for** (**int** i = 1; i <= vertex; i++)

{

**for** (**int** j = 1; j <= vertex; j++)

{

cin >> adjMatrix[i][j];

}

}

**int** start;

cout << "Enter the starting point for BFS: ";

cin >> start;

vector<**int**> ans = BFS(adjMatrix, vertex, start);

cout << "BFS traversal from " << start << ": ";

**for** (**int** i = 0; i < ans.size(); i++)

{

cout << ans[i] << " ";

}

cout << endl;

**return** 0;

}

**Closet Pair of Points:**

// A divide and conquer program in C++

// to find the smallest distance from a

// given set of points.

#include <bits/stdc++.h>

**using** **namespace** std;

**class** Point

{

**public**:

**int** x, y;

};

**int** compareX(**const** **void** \*a, **const** **void** \*b)

{

Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

**return** (p1->x - p2->x);

}

**int** compareY(**const** **void** \*a, **const** **void** \*b)

{

Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

**return** (p1->y - p2->y);

}

**float** dist(Point p1, Point p2)

{

**return** sqrt((p1.x - p2.x) \* (p1.x - p2.x) +

(p1.y - p2.y) \* (p1.y - p2.y));

}

**float** bruteForce(Point P[], **int** n)

{

**float** min = FLT\_MAX;

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

**for** (**int** j = i + 1; j < n; ++j)

**if** (dist(P[i], P[j]) < min)

min = dist(P[i], P[j]);

**return** min;

}

**float** min(**float** x, **float** y)

{

**return** (x < y) ? x : y;

}

**float** stripClosest(Point strip[], **int** size, **float** d)

{

**float** min = d;

qsort(strip, size, **sizeof**(Point), compareY);

**for** (**int** i = 0; i < size; ++i)

**for** (**int** j = i + 1; j < size && (strip[j].y - strip[i].y) < min; ++j)

**if** (dist(strip[i], strip[j]) < min)

min = dist(strip[i], strip[j]);

**return** min;

}

**float** closestUtil(Point P[], **int** n)

{

**if** (n <= 3)

**return** bruteForce(P, n);

**int** mid = n / 2;

Point midPoint = P[mid];

**float** dl = closestUtil(P, mid);

**float** dr = closestUtil(P + mid, n - mid);

**float** d = min(dl, dr);

Point strip[n];

**int** j = 0;

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

**if** (abs(P[i].x - midPoint.x) < d)

strip[j] = P[i], j++;

**return** min(d, stripClosest(strip, j, d));

}

**float** closest(Point P[], **int** n)

{

qsort(P, n, **sizeof**(Point), compareX);

**return** closestUtil(P, n);

}

**int** main()

{

Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};

**int** n = **sizeof**(P) / **sizeof**(P[0]);

cout << "The smallest distance is " << closest(P, n);

**return** 0;

}

**Huffman Decoding:**

// C++ program to encode and decode a string using

// Huffman Coding.

#include <bits/stdc++.h>

#define MAX\_TREE\_HT 256

**using** **namespace** std;

map<**char**, string> codes;

map<**char**, **int**> freq;

**struct** MinHeapNode

{

**char** data; // One of the input characters

**int** freq; // Frequency of the character

MinHeapNode \*left, \*right; // Left and right child

MinHeapNode(**char** data, **int** freq)

{

left = right = NULL;

**this**->data = data;

**this**->freq = freq;

}

};

**struct** compare

{

**bool** **operator**()(MinHeapNode \*l, MinHeapNode \*r)

{

**return** (l->freq > r->freq);

}

};

**void** printCodes(**struct** MinHeapNode \*root, string str)

{

**if** (!root)

**return**;

**if** (root->data != '$')

cout << root->data << ": " << str << "\n";

printCodes(root->left, str + "0");

printCodes(root->right, str + "1");

}

**void** storeCodes(**struct** MinHeapNode \*root, string str)

{

**if** (root == NULL)

**return**;

**if** (root->data != '$')

codes[root->data] = str;

storeCodes(root->left, str + "0");

storeCodes(root->right, str + "1");

}

priority\_queue<MinHeapNode \*, vector<MinHeapNode \*>, compare>

minHeap;

**void** HuffmanCodes(**int** size)

{

**struct** MinHeapNode \*left, \*right, \*top;

**for** (map<**char**, **int**>::iterator v = freq.begin();

v != freq.end(); v++)

minHeap.push(**new** MinHeapNode(v->first, v->second));

**while** (minHeap.size() != 1)

{

left = minHeap.top();

minHeap.pop();

right = minHeap.top();

minHeap.pop();

top = **new** MinHeapNode('$',

left->freq + right->freq);

top->left = left;

top->right = right;

minHeap.push(top);

}

storeCodes(minHeap.top(), "");

}

**void** calcFreq(string str, **int** n)

{

**for** (**int** i = 0; i < str.size(); i++)

freq[str[i]]++;

}

string decode\_file(**struct** MinHeapNode \*root, string s)

{

string ans = "";

**struct** MinHeapNode \*curr = root;

**for** (**int** i = 0; i < s.size(); i++)

{

**if** (s[i] == '0')

curr = curr->left;

**else**

curr = curr->right;

**if** (curr->left == NULL **and** curr->right == NULL)

{

ans += curr->data;

curr = root;

}

}

**return** ans + '\0';

}

**int** main()

{

string str = "geeksforgeeks";

string encodedString, decodedString;

calcFreq(str, str.length());

HuffmanCodes(str.length());

cout << "Character With there Frequencies:\n";

**for** (**auto** v = codes.begin(); v != codes.end(); v++)

cout << v->first << ' ' << v->second << endl;

**for** (**auto** i : str)

encodedString += codes[i];

cout << "\nEncoded Huffman data:\n"

<< encodedString << endl;

decodedString = decode\_file(minHeap.top(), encodedString);

cout << "\nDecoded Huffman Data:\n"

<< decodedString << endl;

**return** 0;

}

**Min Cost Path:**

// Min cost path

#include <bits/stdc++.h>

**using** **namespace** std;

#define row 3

#define col 3

**int** minCost(**int** cost[row][col])

{

**for** (**int** i = 1; i < row; i++)

cost[i][0] += cost[i - 1][0];

**for** (**int** j = 1; j < col; j++)

cost[0][j] += cost[0][j - 1];

**for** (**int** i = 1; i < row; i++)

**for** (**int** j = 1; j < col; j++)

cost[i][j] += min(cost[i - 1][j - 1],

min(cost[i - 1][j], cost[i][j - 1]));

**return** cost[row - 1][col - 1];

}

**int** main(**int** argc, **char** **const** \*argv[])

{

**int** cost[row][col] = {{1, 2, 3}, {4, 8, 2}, {1, 5, 3}};

cout << minCost(cost) << endl;

**return** 0;

}

binary\_search(startaddress, endaddress, valuetofind)