GREEDY :

MERGE SORT:

#include <iostream>

using namespace std;

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

L[i] = arr[l + i];

}

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

R[j] = arr[m + 1 + j];

}

int 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++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

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;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

int main() {

int n;

cout << "Enter the number of elements: ";

cin >> n;

int arr[n];

cout << "Enter the elements: ";

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

cin >> arr[i];

}

mergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

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

cout << arr[i] << " ";

}

cout << endl;

return 0;

}

INSERTION SORT:

#include <bits/stdc++.h>

using namespace std;

void insertionSort(int arr[], int n) {

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

int n = sizeof(arr) / sizeof(arr[0]);

insertionSort(arr, n);

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

cout << arr[i] << " ";

}

cout << endl;

return 0;

}

FRACTIONAL KNAPSACK

#include <iostream>

#include <algorithm>

using namespace std;

struct Item {

int value;

int weight;

};

bool cmp(Item a, Item b) {

double r1 = (double) a.value / a.weight;

double r2 = (double) b.value / b.weight;

return r1 > r2;

}

double fractionalKnapsack(int W, Item arr[], int n) {

sort(arr, arr + n, cmp);

double res = 0.0;

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

if (arr[i].weight <= W) {

res += arr[i].value;

W -= arr[i].weight;

}

else {

res += (double) arr[i].value / arr[i].weight \* W;

break;

}

}

return res;

}

int main() {

int W = 50;

Item arr[] = {{60, 10}, {100, 20}, {120, 30}};

int n = sizeof(arr) / sizeof(arr[0]);

double maxVal = fractionalKnapsack(W, arr, n);

cout << "Maximum value that can be obtained is: " << maxVal << endl;

return 0;

}

0/1 KNAPSACK

#include <iostream>

using namespace std;

int knapsack(int W, int wt[], int val[], int n) {

int dp[n + 1][W + 1];

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

for(int w = 0; w <= W; w++) {

if(i == 0 || w == 0) dp[i][w] = 0;

else if(wt[i - 1] <= w) dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);

else dp[i][w] = dp[i - 1][w];

}

}

return dp[n][W];

}

int main() {

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

int maxVal = knapsack(W, wt, val, n);

cout << "Maximum value that can be obtained is: " << maxVal << endl;

return 0;

}

JOB SELECTON:

#include <iostream>

#include <algorithm>

using namespace std;

struct Job {

char id;

int deadline;

int profit;

};

bool cmp(Job a, Job b) {

return a.profit > b.profit;

}

void jobSequence(Job arr[], int n) {

sort(arr, arr + n, cmp);

int result[n];

bool slot[n];

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

slot[i] = false;

}

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

for (int j = min(n, arr[i].deadline) - 1; j >= 0; j--) {

if (slot[j] == false) {

result[j] = i;

slot[j] = true;

break;

}

}

}

cout << "Job sequence: ";

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

if (slot[i]) {

cout << arr[result[i]].id << " ";

}

}

cout << endl;

int totalProfit = 0;

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

if (slot[i]) {

totalProfit += arr[result[i]].profit;

}

}

cout << "Total profit: " << totalProfit << endl;

}

int main() {

int n;

cout << "Enter the number of jobs: ";

cin >> n;

Job arr[n];

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

cout << "Enter the ID, deadline, and profit of job " << i+1 << ": ";

cin >> arr[i].id >> arr[i].deadline >> arr[i].profit;

}

jobSequence(arr, n);

return 0;

}

PRIMS ALGO:

#include <iostream>

#include <climits>

using namespace std;

#define V 5

int minKey(int key[], bool mstSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (mstSet[v] == false && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

void printMST(int parent[], int graph[V][V]) {

cout << "Edge Weight" << endl;

for (int i = 1; i < V; i++) {

cout << parent[i] << " - " << i << " " << graph[i][parent[i]] << endl;

}

}

void primMST(int graph[V][V]) {

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++) {

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][v];

}

}

}

printMST(parent, graph);

}

int main() {

int graph[V][V];

cout << "Enter the adjacency matrix of the graph:" << endl;

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

for (int j = 0; j < V; j++) {

cin >> graph[i][j];

}

}

primMST(graph);

return 0;

}

KRUSALS ALGO:

#include <iostream>

#include <algorithm>

#include <climits>

using namespace std;

#define V 5

#define E 7

struct Edge {

int src, dest, weight;

};

bool cmp(Edge a, Edge b) {

return a.weight < b.weight;

}

int find(int parent[], int i) {

if (parent[i] == -1) {

return i;

}

return find(parent, parent[i]);

}

void Union(int parent[], int x, int y) {

parent[x] = y;

}

void kruskalMST(Edge edges[]) {

Edge result[V-1];

int parent[V];

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

parent[i] = -1;

}

sort(edges, edges + E, cmp);

int i = 0, j = 0;

while (i < V-1 && j < E) {

Edge next\_edge = edges[j++];

int x = find(parent, next\_edge.src);

int y = find(parent, next\_edge.dest);

if (x != y) {

result[i++] = next\_edge;

Union(parent, x, y);

}

}

cout << "Edge Weight" << endl;

for (i = 0; i < V-1; i++) {

cout << result[i].src << " - " << result[i].dest << " " << result[i].weight << endl;

}

}

int main() {

Edge edges[E];

cout << "Enter the source, destination, and weight of each edge:" << endl;

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

cin >> edges[i].src >> edges[i].dest >> edges[i].weight;

}

kruskalMST(edges);

return 0;

}

DIJKSTARS ALGO:

#include <iostream>

#include <climits>

using namespace std;

#define V 5

int minDistance(int dist[], bool sptSet[]) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++) {

if (sptSet[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printSolution(int dist[]) {

cout << "Vertex Distance from source" << endl;

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

cout << i << " " << dist[i] << endl;

}

}

void dijkstra(int graph[V][V], int src) {

int dist[V];

bool sptSet[V];

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

dist[i] = INT\_MAX;

sptSet[i] = false;

}

dist[src] = 0;

for (int count = 0; count < V-1; count++) {

int u = minDistance(dist, sptSet);

sptSet[u] = true;

for (int v = 0; v < V; v++) {

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX &&

dist[u] + graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph[u][v];

}

}

}

printSolution(dist);

}

int main() {

int graph[V][V];

cout << "Enter the adjacency matrix of the graph:" << endl;

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

for (int j = 0; j < V; j++) {

cin >> graph[i][j];

}

}

int src;

cout << "Enter the source vertex: ";

cin >> src;

dijkstra(graph, src);

return 0;

}

DYNAMIC PROGRAMMING :

DIVIDE AND CONQUER :

#include <iostream>

using namespace std;

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

L[i] = arr[l + i];

}

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

R[j] = arr[m + 1 + j];

}

int 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++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

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;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

void printArray(int arr[], int size) {

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

cout << arr[i] << " ";

}

cout << endl;

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Enter the elements of the array:" << endl;

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

cin >> arr[i];

}

mergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

MAXIMUM SUM SUBARRAY:

#include <iostream>

using namespace std;

int maxSubArraySum(int arr[], int n) {

int max\_so\_far = arr[0];

int curr\_max = arr[0];

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

curr\_max = max(arr[i], curr\_max + arr[i]);

max\_so\_far = max(max\_so\_far, curr\_max);

}

return max\_so\_far;

}

int main() {

int arr[] = {-2, -3, 4, -1, -2, 1, 5, -3};

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Enter the elements of the array:" << endl;

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

cin >> arr[i];

}

int max\_sum = maxSubArraySum(arr, n);

cout << "Maximum subarray sum is " << max\_sum << endl;

return 0;

}

KARATSUBA ALGO:

#include <iostream>

#include <cmath>

using namespace std;

int getLength(int num) {

int len = 0;

while (num != 0) {

len++;

num /= 10;

}

return len;

}

int karatsuba(int x, int y) {

if (x < 10 || y < 10) {

return x \* y;

}

int n = max(getLength(x), getLength(y));

int m = ceil(n / 2.0);

int p = pow(10, m);

int a = x / p;

int b = x % p;

int c = y / p;

int d = y % p;

int ac = karatsuba(a, c);

int bd = karatsuba(b, d);

int abcd = karatsuba(a + b, c + d);

return ac \* pow(10, 2\*m) + (abcd - ac - bd) \* pow(10, m) + bd;

}

int main() {

int x = 1234;

int y = 5678;

cout << "Enter two numbers to multiply:" << endl;

cin >> x >> y;

int result = karatsuba(x, y);

cout << "Result: " << result << endl;

return 0;

}

MATRIX CHAIN MULTILICATION:

#include <iostream>

using namespace std;

#define ROW 2

#define COL 3

void matrixMultiplication(int mat1[][COL], int mat2[][COL], int res[][COL]) {

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

for (int j = 0; j < COL; j++) {

res[i][j] = 0;

for (int k = 0; k < COL; k++) {

res[i][j] += mat1[i][k] \* mat2[k][j];

}

}

}

}

void printMatrix(int mat[][COL]) {

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

for (int j = 0; j < COL; j++) {

cout << mat[i][j] << " ";

}

cout << endl;

}

}

int main() {

int mat1[ROW][COL] = {{1, 2, 3}, {4, 5, 6}};

int mat2[ROW][COL] = {{7, 8, 9}, {10, 11, 12}};

int res[ROW][COL];

cout << "Enter the elements of the first matrix:" << endl;

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

for (int j = 0; j < COL; j++) {

cin >> mat1[i][j];

}

}

cout << "Enter the elements of the second matrix:" << endl;

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

for (int j = 0; j < COL; j++) {

cin >> mat2[i][j];

}

}

matrixMultiplication(mat1, mat2, res);

cout << "Resultant matrix:" << endl;

printMatrix(res);

return 0;

}

LONGEST COMMON SUBSEQUENCE:

#include <iostream>

#include <cstring>

using namespace std;

int max(int a, int b) {

return (a > b) ? a : b;

}

int lcs(char\* X, char\* Y, int m, int n) {

int L[m+1][n+1];

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

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

if (i == 0 || j == 0) {

L[i][j] = 0;

}

else if (X[i-1] == Y[j-1]) {

L[i][j] = L[i-1][j-1] + 1;

}

else {

L[i][j] = max(L[i-1][j], L[i][j-1]);

}

}

}

return L[m][n];

}

int main() {

char X[] = "AGGTAB";

char Y[] = "GXTXAYB";

int m = strlen(X);

int n = strlen(Y);

cout << "Enter two strings to find the longest common subsequence:" << endl;

cin >> X >> Y;

m = strlen(X);

n = strlen(Y);

cout << "Length of LCS is " << lcs(X, Y, m, n) << endl;

return 0;

}

0/1 KNAPSACK USING DP:

#include <iostream>

using namespace std;

int max(int a, int b) {

return (a > b) ? a : b;

}

int knapsack(int W, int wt[], int val[], int n) {

int K[n+1][W+1];

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

for (int w = 0; w <= W; w++) {

if (i == 0 || w == 0) {

K[i][w] = 0;

}

else if (wt[i-1] <= w) {

K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

}

else {

K[i][w] = K[i-1][w];

}

}

}

return K[n][W];

}

int main() {

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

int n = sizeof(val)/sizeof(val[0]);

cout << "Enter the capacity of the knapsack:" << endl;

cin >> W;

cout << "Enter the number of items:" << endl;

cin >> n;

cout << "Enter the weights of the items:" << endl;

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

cin >> wt[i];

}

cout << "Enter the values of the items:" << endl;

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

cin >> val[i];

}

cout << "Maximum value that can be put in a knapsack of capacity " << W << " is " << knapsack(W, wt, val, n) << endl;

return 0;

}

BACKTRACKING

N-QUEENS :

#include <iostream>

#include <cmath>

using namespace std;

void printSolution(int\*\* board, int n) {

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

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

cout << board[i][j] << " ";

}

cout << endl;

}

}

bool isSafe(int\*\* board, int row, int col, int n) {

int i, j;

for (i = 0; i < col; i++) {

if (board[row][i]) {

return false;

}

}

for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j]) {

return false;

}

}

for (i = row, j = col; j >= 0 && i < n; i++, j--) {

if (board[i][j]) {

return false;

}

}

return true;

}

bool solveNQUtil(int\*\* board, int col, int n) {

if (col == n) {

return true;

}

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

if (isSafe(board, i, col, n)) {

board[i][col] = 1;

if (solveNQUtil(board, col + 1, n)) {

return true;

}

board[i][col] = 0;

}

}

return false;

}

void solveNQ(int n) {

int\*\* board = new int\*[n];

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

board[i] = new int[n];

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

board[i][j] = 0;

}

}

if (solveNQUtil(board, 0, n) == false) {

cout << "Solution does not exist" << endl;

return;

}

printSolution(board, n);

}

int main() {

int n = 4;

cout << "Enter the number of queens:" << endl;

cin >> n;

solveNQ(n);

return 0;

}

SUBSET SUM:

#include <iostream>

using namespace std;

bool isSubsetSum(int\* set, int n, int sum) {

if (sum == 0) {

return true;

}

if (n == 0) {

return false;

}

if (set[n-1] > sum) {

return isSubsetSum(set, n-1, sum);

}

return isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1]);

}

int main() {

int set[] = {3, 34, 4, 12, 5, 2};

int sum = 9;

int n = sizeof(set)/sizeof(set[0]);

cout << "Enter the size of the set:" << endl;

cin >> n;

int\* set = new int[n];

cout << "Enter the elements of the set:" << endl;

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

cin >> set[i];

}

cout << "Enter the target sum:" << endl;

cin >> sum;

if (isSubsetSum(set, n, sum)) {

cout << "Found a subset with given sum" << endl;

}

else {

cout << "No subset with given sum" << endl;

}

return 0;

}

NAIVE STRING MATCHING :

#include <iostream>

#include <cstring>

using namespace std;

void search(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

for (int i = 0; i <= N - M; i++) {

int j;

for (j = 0; j < M; j++) {

if (txt[i+j] != pat[j]) {

break;

}

}

if (j == M) {

cout << "Pattern found at index " << i << endl;

}

}

}

int main() {

char txt[] = "AABAACAADAABAABA";

char pat[] = "AABA";

cout << "Enter the text:" << endl;

cin >> txt;

cout << "Enter the pattern:" << endl;

cin >> pat;

search(pat, txt);

return 0;

}

KMP STRING MATCHING:

#include <iostream>

#include <cstring>

using namespace std;

void computeLPS(char\* pat, int M, int\* lps) {

int len = 0;

lps[0] = 0;

int i = 1;

while (i < M) {

if (pat[i] == pat[len]) {

len++;

lps[i] = len;

i++;

}

else {

if (len != 0) {

len = lps[len-1];

}

else {

lps[i] = 0;

i++;

}

}

}

}

void search(char\* pat, char\* txt) {

int M = strlen(pat);

int N = strlen(txt);

int lps[M];

computeLPS(pat, M, lps);

int i = 0;

int j = 0;

while (i < N) {

if (pat[j] == txt[i]) {

i++;

j++;

}

if (j == M) {

cout << "Pattern found at index " << i-j << endl;

j = lps[j-1];

}

else if (i < N && pat[j] != txt[i]) {

if (j != 0) {

j = lps[j-1];

}

else {

i++;

}

}

}

}

int main() {

char txt[] = "AABAACAADAABAABA";

char pat[] = "AABA";

cout << "Enter the text:" << endl;

cin >> txt;

cout << "Enter the pattern:" << endl;

cin >> pat;

search(pat, txt);

return 0;

}

RABIN KARP STRING MAYCHING:

#include <iostream>

#include <cstring>

using namespace std;

#define d 256

void search(char\* pat, char\* txt, int q) {

int M = strlen(pat);

int N = strlen(txt);

int i, j;

int p = 0;

int t = 0;

int h = 1;

for (i = 0; i < M - 1; i++) {

h = (h\*d) % q;

}

for (i = 0; i < M; i++) {

p = (d\*p + pat[i]) % q;

t = (d\*t + txt[i]) % q;

}

for (i = 0; i <= N - M; i++) {

if (p == t) {

for (j = 0; j < M; j++) {

if (txt[i+j] != pat[j]) {

break;

}

}

if (j == M) {

cout << "Pattern found at index " << i << endl;

}

}

if (i < N - M) {

t = (d\*(t - txt[i]\*h) + txt[i+M]) % q;

if (t < 0) {

t = (t + q);

}

}

}

}

int main() {

char txt[] = "AABAACAADAABAABA";

char pat[] = "AABA";

int q = 101;

cout << "Enter the text:" << endl;

cin >> txt;

cout << "Enter the pattern:" << endl;

cin >> pat;

cout << "Enter the prime number:" << endl;

cin >> q;

search(pat, txt, q);

return 0;

}

BELLMAN-FORD ALGO:

#include <iostream>

#include <climits>

using namespace std;

struct Edge {

int src, dest, weight;

};

struct Graph {

int V, E;

struct Edge\* edge;

};

struct Graph\* createGraph(int V, int E) {

struct Graph\* graph = new Graph;

graph->V = V;

graph->E = E;

graph->edge = new Edge[E];

return graph;

}

void printArr(int dist[], int n) {

cout << "Vertex Distance from Source" << endl;

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

cout << i << "\t\t" << dist[i] << endl;

}

}

void BellmanFord(struct Graph\* graph, int src) {

int V = graph->V;

int E = graph->E;

int dist[V];

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

dist[i] = INT\_MAX;

}

dist[src] = 0;

for (int i = 1; i < V; i++) {

for (int j = 0; j < E; j++) {

int u = graph->edge[j].src;

int v = graph->edge[j].dest;

int weight = graph->edge[j].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

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

int u = graph->edge[i].src;

int v = graph->edge[i].dest;

int weight = graph->edge[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

cout << "Graph contains negative weight cycle" << endl;

return;

}

}

printArr(dist, V);

}

int main() {

int V = 5;

int E = 8;

struct Graph\* graph = createGraph(V, E);

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = -1;

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 4;

graph->edge[2].src = 1;

graph->edge[2].dest = 2;

graph->edge[2].weight = 3;

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 2;

graph->edge[4].src = 1;

graph->edge[4].dest = 4;

graph->edge[4].weight = 2;

graph->edge[5].src = 3;

graph->edge[5].dest = 2;

graph->edge[5].weight = 5;

graph->edge[6].src = 3;

graph->edge[6].dest = 1;

graph->edge[6].weight = 1;

graph->edge[7].src = 4;

graph->edge[7].dest = 3;

graph->edge[7].weight = -3;

cout << "Enter the number of vertices:" << endl;

cin >> V;

cout << "Enter the number of edges:" << endl;

cin >> E;

struct Graph\* graph = createGraph(V, E);

cout << "Enter the edges: source destination weight" << endl;

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

cin >> graph->edge[i].src >> graph->edge[i].dest >> graph->edge[i].weight;

}

int src = 0;

cout << "Enter the source vertex:" << endl;

cin >> src;

BellmanFord(graph, src);

return 0;

}

FLOYD WARSHALL ALGO

#include <iostream>

#include <climits>

using namespace std;

#define V 4

void printSolution(int dist[][V]) {

cout << "Shortest distances between every pair of vertices:" << endl;

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

for (int j = 0; j < V; j++) {

if (dist[i][j] == INT\_MAX) {

cout << "INF\t";

}

else {

cout << dist[i][j] << "\t";

}

}

cout << endl;

}

}

void FloydWarshall(int graph[][V]) {

int dist[V][V];

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

for (int j = 0; j < V; j++) {

dist[i][j] = graph[i][j];

}

}

for (int k = 0; k < V; k++) {

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

for (int j = 0; j < V; j++) {

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {

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

}

}

}

}

printSolution(dist);

}

int main() {

int graph[V][V] = {{0, 5, INT\_MAX, 10},

{INT\_MAX, 0, 3, INT\_MAX},

{INT\_MAX, INT\_MAX, 0, 1},

{INT\_MAX, INT\_MAX, INT\_MAX, 0}};

cout << "Enter the graph (in the form of an adjacency matrix):" << endl;

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

for (int j = 0; j < V; j++) {

cin >> graph[i][j];

}

}

FloydWarshall(graph);

return 0;

}

FORD FULKERSON :

#include <iostream>

#include <cstring>

#include <queue>

using namespace std;

#define V 6

bool bfs(int rGraph[][V], int s, int t, int parent[]) {

bool visited[V];

memset(visited, 0, sizeof(visited));

queue<int> q;

q.push(s);

visited[s] = true;

parent[s] = -1;

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v = 0; v < V; v++) {

if (!visited[v] && rGraph[u][v] > 0) {

q.push(v);

parent[v] = u;

visited[v] = true;

}

}

}

return (visited[t] == true);

}

int fordFulkerson(int graph[][V], int s, int t) {

int u, v;

int rGraph[V][V];

for (u = 0; u < V; u++) {

for (v = 0; v < V; v++) {

rGraph[u][v] = graph[u][v];

}

}

int parent[V];

int max\_flow = 0;

while (bfs(rGraph, s, t, parent)) {

int path\_flow = INT\_MAX;

for (v = t; v != s; v = parent[v]) {

u = parent[v];

path\_flow = min(path\_flow, rGraph[u][v]);

}

for (v = t; v != s; v = parent[v]) {

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

max\_flow += path\_flow;

}

return max\_flow;

}

int main() {

int graph[V][V] = {{0, 16, 13, 0, 0, 0},

{0, 0, 10, 12, 0, 0},

{0, 4, 0, 0, 14, 0},

{0, 0, 9, 0, 0, 20},

{0, 0, 0, 7, 0, 4},

{0, 0, 0, 0, 0, 0}};

cout << "Enter the graph (in the form of an adjacency matrix):" << endl;

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

for (int j = 0; j < V; j++) {

cin >> graph[i][j];

}

}

int s = 0;

int t = 5;

cout << "Enter the source and sink vertices:" << endl;

cin >> s >> t;

cout << "The maximum possible flow is " << fordFulkerson(graph, s, t) << endl;

return 0;

}

EDMOND KARP:

#include <iostream>

#include <cstring>

#include <queue>

using namespace std;

#define V 6

bool bfs(int rGraph[][V], int s, int t, int parent[]) {

bool visited[V];

memset(visited, 0, sizeof(visited));

queue<int> q;

q.push(s);

visited[s] = true;

parent[s] = -1;

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v = 0; v < V; v++) {

if (!visited[v] && rGraph[u][v] > 0) {

q.push(v);

parent[v] = u;

visited[v] = true;

}

}

}

return (visited[t] == true);

}

int edmondsKarp(int graph[][V], int s, int t) {

int u, v;

int rGraph[V][V];

for (u = 0; u < V; u++) {

for (v = 0; v < V; v++) {

rGraph[u][v] = graph[u][v];

}

}

int parent[V];

int max\_flow = 0;

while (bfs(rGraph, s, t, parent)) {

int path\_flow = INT\_MAX;

for (v = t; v != s; v = parent[v]) {

u = parent[v];

path\_flow = min(path\_flow, rGraph[u][v]);

}

for (v = t; v != s; v = parent[v]) {

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

}

max\_flow += path\_flow;

}

return max\_flow;

}

int main() {

int graph[V][V] = {{0, 16, 13, 0, 0, 0},

{0, 0, 10, 12, 0, 0},

{0, 4, 0, 0, 14, 0},

{0, 0, 9, 0, 0, 20},

{0, 0, 0, 7, 0, 4},

{0, 0, 0, 0, 0, 0}};

cout << "Enter the graph (in the form of an adjacency matrix):" << endl;

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

for (int j = 0; j < V; j++) {

cin >> graph[i][j];

}

}

int s = 0;

int t = 5;

cout << "Enter the source and sink vertices:" << endl;

cin >> s >> t;

cout << "The maximum possible flow is " << edmondsKarp(graph, s, t) << endl;

return 0;

}

CONVEX HULL:

#include <iostream>

#include <stack>

#include <cmath>

using namespace std;

struct Point {

int x, y;

};

Point p0;

Point nextToTop(stack<Point> &S) {

Point p = S.top();

S.pop();

Point res = S.top();

S.push(p);

return res;

}

int distSq(Point p1, Point p2) {

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

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

}

int orientation(Point p, Point q, Point r) {

int val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

if (val == 0) {

return 0;

}

return (val > 0) ? 1 : 2;

}

int compare(const void \*vp1, const void \*vp2) {

Point \*p1 = (Point \*)vp1;

Point \*p2 = (Point \*)vp2;

int o = orientation(p0, \*p1, \*p2);

if (o == 0) {

return (distSq(p0, \*p2) >= distSq(p0, \*p1)) ? -1 : 1;

}

return (o == 2) ? -1 : 1;

}

void convexHull(Point points[], int n) {

int ymin = points[0].y;

int min = 0;

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

int y = points[i].y;

if ((y < ymin) || (ymin == y && points[i].x < points[min].x)) {

ymin = points[i].y;

min = i;

}

}

swap(points[0], points[min]);

p0 = points[0];

qsort(&points[1], n - 1, sizeof(Point), compare);

int m = 1;

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

while (i < n - 1 && orientation(p0, points[i], points[i + 1]) == 0) {

i++;

}

points[m] = points[i];

m++;

}

if (m < 3) {

return;

}

stack<Point> S;

S.push(points[0]);

S.push(points[1]);

S.push(points[2]);

for (int i = 3; i < m; i++) {

while (orientation(nextToTop(S), S.top(), points[i]) != 2) {

S.pop();

}

S.push(points[i]);

}

while (!S.empty()) {

Point p = S.top();

cout << "(" << p.x << ", " << p.y << ")" << endl;

S.pop();

}

}

int main() {

int n;

cout << "Enter the number of points: ";

cin >> n;

Point points[n];

cout << "Enter the points (in the form of x and y coordinates):" << endl;

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

cin >> points[i].x >> points[i].y;

}

convexHull(points, n);

return 0;

}

GRAHAMS SCAN ALGO:

#include <iostream>

#include <cmath>

using namespace std;

struct Point {

int x, y;

};

Point p0;

Point nextToTop(Point S[], int &top) {

Point p = S[top];

top--;

Point res = S[top];

S[top] = p;

return res;

}

int distSq(Point p1, Point p2) {

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

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

}

int orientation(Point p, Point q, Point r) {

int val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

if (val == 0) {

return 0;

}

return (val > 0) ? 1 : 2;

}

int compare(const void \*vp1, const void \*vp2) {

Point \*p1 = (Point \*)vp1;

Point \*p2 = (Point \*)vp2;

int o = orientation(p0, \*p1, \*p2);

if (o == 0) {

return (distSq(p0, \*p2) >= distSq(p0, \*p1)) ? -1 : 1;

}

return (o == 2) ? -1 : 1;

}

void grahamScan(Point points[], int n) {

int ymin = points[0].y;

int min = 0;

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

int y = points[i].y;

if ((y < ymin) || (ymin == y && points[i].x < points[min].x)) {

ymin = points[i].y;

min = i;

}

}

swap(points[0], points[min]);

p0 = points[0];

qsort(&points[1], n - 1, sizeof(Point), compare);

int m = 1;

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

while (i < n - 1 && orientation(p0, points[i], points[i + 1]) == 0) {

i++;

}

points[m] = points[i];

m++;

}

if (m < 3) {

return;

}

Point S[m];

int top = 1;

S[0] = points[0];

S[1] = points[1];

S[2] = points[2];

for (int i = 3; i < m; i++) {

while (orientation(nextToTop(S, top), S[top], points[i]) != 2) {

top--;

}

top++;

S[top] = points[i];

}

while (top >= 0) {

Point p = S[top];

cout << "(" << p.x << ", " << p.y << ")" << endl;

top--;

}

}

int main() {

int n;

cout << "Enter the number of points: ";

cin >> n;

Point points[n];

cout << "Enter the points (in the form of x and y coordinates):" << endl;

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

cin >> points[i].x >> points[i].y;

}

grahamScan(points, n);

return 0;

}

JARVIS MARCH:

#include <iostream>

#include <cmath>

using namespace std;

struct Point {

int x, y;

};

int orientation(Point p, Point q, Point r) {

int val = (q.y - p.y) \* (r.x - q.x) -

(q.x - p.x) \* (r.y - q.y);

if (val == 0) {

return 0;

}

return (val > 0) ? 1 : 2;

}

void convexHull(Point points[], int n) {

if (n < 3) {

return;

}

int l = 0;

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

if (points[i].x < points[l].x) {

l = i;

}

}

int p = l, q;

do {

q = (p + 1) % n;

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

if (orientation(points[p], points[i], points[q]) == 2) {

q = i;

}

}

cout << "(" << points[p].x << ", " << points[p].y << ")" << endl;

p = q;

} while (p != l);

}

int main() {

int n;

cout << "Enter the number of points: ";

cin >> n;

Point points[n];

cout << "Enter the points (in the form of x and y coordinates):" << endl;

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

cin >> points[i].x >> points[i].y;

}

convexHull(points, n);

return 0;

}