// Connected using DFS

#include <stdio.h>

#include <stdlib.h>

int n,visited[100],adj[100][100];

void dfs(int u){

int v;

visited[u]=1;

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

if(adj[u][v] && !visited[v]){

printf("%d->%d\n",u,v);

dfs(v);

}

}

}

int main()

{

int i,j,count=0;

printf("Enter the no. of vertices: ");

scanf("%d",&n);

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

visited[i]=0;

}

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

adj[i][j]=0;

}

printf("\nEnter the elements in array: \n");

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

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

scanf("%d",&adj[i][j]);

}

}

dfs(0);

printf("\n");

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

if(visited[i]){

count++;

}

}

if(count==n){

printf("\nConnected");

}

else{

printf("Not connected");

}

return 0;

}

// Floyd’s Algorithm

#include <stdio.h>

#include <stdlib.h>

#define INF 99999

void print(int distance[100][100],int n){

printf("Shortest distance between every pair of vertices: \n");

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

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

if(distance[i][j]==INF){

printf("%7s","INF");

}

else{

printf("%7d",distance[i][j]);

}

}

printf("\n");

}

}

void Floyd(int graph[100][100],int n){

int distance[100][100];

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

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

distance[i][j]=graph[i][j];

}

}

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

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

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

if(distance[i][j] > distance[i][k]+distance[k][j]){

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

}

}

}

}

print(distance,n);

}

int main()

{

int n;

printf("Enter the no. of vertices: ");

scanf("%d",&n);

int graph[100][100];

printf("\nEnter the adjacency matrix:\n ");

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

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

scanf("%d", &graph[i][j]);

}

}

Floyd(graph,n);

return 0;

}

#include <stdio.h>//Merge sort

#include <stdlib.h>

#include <time.h>

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

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

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

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

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

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

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 fillArray(int arr[], int n) {

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

arr[i] = rand() % 100;

}

}

int main() {

int n;

printf("Enter the number of elements: ");

scanf("%d", &n);

int\* arr = (int\*)malloc(n \* sizeof(int));

srand(time(0));

fillArray(arr, n);

printf("Unsorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

clock\_t start, end;

double cpu\_time\_used;

start = clock();

mergeSort(arr, 0, n - 1);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

printf("Time taken to sort %d elements: %f seconds\n", n, cpu\_time\_used);

free(arr);

return 0;

}

//#merge sort

Knapsack Algo

#include <stdio.h>

// Function to find maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to solve the 0/1 knapsack problem using dynamic programming

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

int i, w;

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

// Build table K[][] in bottom-up manner

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

for (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 n, i;

printf("Enter the number of items: ");

scanf("%d", &n);

int val[n], wt[n];

// Input the values of items3

printf("Enter the values of the items:\n");

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

scanf("%d", &val[i]);

}

// Input the weights of items

printf("Enter the weights of the items:\n");

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

scanf("%d", &wt[i]);

}

int W;

printf("Enter the capacity of knapsack: ");

scanf("%d", &W);

printf("Maximum value that can be put in a knapsack of capacity %d: %d\n", W, knapsack(W, wt, val, n));

return 0;

}

Quick sort

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to swap two elements in the array

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to partition the array using the last element as pivot

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // pivot

int i = (low - 1); // Index of smaller element

for (int j = low; j <= high - 1; j++) {

// If current element is smaller than the pivot

if (arr[j] < pivot) {

i++; // increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

// Function to implement Quick Sort

void quickSort(int arr[], int low, int high) {

if (low < high) {

// pi is partitioning index, arr[p] is now at right place

int pi = partition(arr, low, high);

// Separately sort elements before partition and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print an array

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

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

printf("%d ", arr[i]);

printf("\n");

}

int main() {

int n;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

printf("Original array:\n");

printArray(arr, n);

// Measure the time taken by quick sort

clock\_t start, end;

double cpu\_time\_used;

start = clock();

// Perform quick sort

quickSort(arr, 0, n - 1);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted array:\n");

printArray(arr, n);

printf("Time taken to sort array: %f seconds\n", cpu\_time\_used);

return 0;

}

Heapsort

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Function to swap two elements in the array

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted with node i which is an index in arr[]

void heapify(int arr[], int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If left child is larger than root

if (left < n && arr[left] > arr[largest])

largest = left;

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest])

largest = right;

// If largest is not root

if (largest != i) {

swap(&arr[i], &arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

// Function to implement Heap Sort

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

// Build max heap

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

// Extract elements from heap one by one

for (int i = n - 1; i > 0; i--) {

// Move current root to end

swap(&arr[0], &arr[i]);

// Call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Function to print an array

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

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

printf("%d ", arr[i]);

printf("\n");

}

int main() {

int n;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

printf("Original array:\n");

printArray(arr, n);

// Measure the time taken by heap sort

clock\_t start, end;

double cpu\_time\_used;

start = clock();

// Perform heap sort

heapSort(arr, n);

end = clock();

cpu\_time\_used = ((double) (end - start)) / CLOCKS\_PER\_SEC;

printf("Sorted array:\n");

printArray(arr, n);

printf("Time taken to sort array: %f seconds\n", cpu\_time\_used);

return 0;

}

Connected Using BFS

#include<stdio.h>

#include<stdlib.h>

int visited[100], queue[100], front = -1, rear = -1, n, i, j;

int adj[100][100];

void bfs(int v){

while(front<=rear){

v=queue[front++];

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

if(adj[v][i] && !visited[i]){

queue[rear++]=i;

visited[i]=1;

}

}

}

}

int main(){

int v;

printf("Enter the no. of vertices: ");

scanf("%d",&n);

printf("\nEnter the matrix: \n");

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

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

scanf("%d",&adj[i][j]);

}

}

printf("\nEnter the starting vertex: ");

scanf("%d",&v);

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

queue[i]=0;

visited[i]=0;

}

bfs(v);

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

if(visited[i]){

printf("->%d",i);

}

else{

printf("Not possible");

}

}

}

Topo using DFS

#include <stdio.h>

#include <stdlib.h>

int s[100],res[100];

int j;

void adjacency(int a[100][100],int n){

printf("Enter the elements: \n");

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

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

scanf("%d",&a[i][j]);

}

}

}

void dfs(int u,int n,int a[100][100]){

s[u]=1;

int v;

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

if(a[u][v]&& !s[v]){

dfs(v,n,a);

}

}

res[j--]=u;

}

void topological(int n, int a[100][100]){

int i,u;

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

s[i]=0;

}

j=n-1;

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

if(!s[u]){56

dfs(u,n,a);

}

}

}

int main()

{

int a[100][100],n,i,j;

printf("Enter the no. of verƟces: ");

scanf("%d",&n);

adjacency(a,n);

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

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

printf("%d\t",a[i][j]);

}

printf("\n");

}

printf("Topological Order: \n");

topological(n,a);

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

printf("->%d",res[i]);

}

return 0;

}

Topo Using Source Removal

#include <stdio.h>

int main(){

int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;

printf("Enter the no of vertices:\n");

scanf("%d",&n);

printf("Enter the adjacency matrix:\n");

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

printf("Enter row %d\n",i+1);

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

scanf("%d",&a[i][j]);

}

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

indeg[i]=0;

flag[i]=0;

}

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

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

indeg[i]=indeg[i]+a[j][i];

printf("\nThe topological order is:");

while(count<n){

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

if((indeg[k]==0) && (flag[k]==0)){

printf("%d ",(k+1));

flag [k]=1;

}

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

if(a[i][k]==1)

indeg[k]--;

}

}

count++;

}

return 0;

}

//source removal