Slip 1

Q1) Write a program to sort a list of n numbers in ascending order using selection sort and determine the time required to sort the elements.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

int i, j, minIndex, temp;

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

minIndex = i;

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

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

int main() {

int n, i;

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

scanf("%d", &n);

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

if (arr == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

printf("Enter %d integers: ", n);

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

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

}

clock\_t start\_time, end\_time;

double time\_used;

start\_time = clock();

selectionSort(arr, n);

end\_time = clock();

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

printf("Sorted array in ascending order:\n");

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

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

}

printf("\n")

printf("Time taken for sorting: %lf seconds\n", time\_used);

free(arr);

return 0;

}

Q2) Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from a file or can be generated using the random number generator.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; 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 pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

FILE \*fp;

fp = fopen("numbers.txt", "w");

srand(time(NULL));

int n\_values[] = {100, 500, 1000, 5000};

int n\_values\_count = sizeof(n\_values) / sizeof(n\_values[0]);

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

int n = n\_values[k];

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

if (arr == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

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

arr[i] = rand() % 10000;

fprintf(fp, "%d\n", arr[i]);

}

clock\_t start\_time, end\_time;

double time\_used;

start\_time = clock();

quickSort(arr, 0, n - 1);

end\_time = clock();

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

printf("Time taken to sort %d elements: %lf seconds\n", n, time\_used);

free(arr);

}

fclose(fp);

return 0;

}

Slip 2

Q.1) Write a program to sort n randomly generated elements using Heapsort method.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

}

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

largest = right;

}

if (largest != i) {

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

heapify(arr, n, largest);

}

}

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

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

heapify(arr, n, i);

}

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

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

heapify(arr, i, 0);

}

}

int main() {

srand(time(NULL));

int n\_values[] = {100, 500, 1000, 5000};

int n\_values\_count = sizeof(n\_values) / sizeof(n\_values[0]);

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

int n = n\_values[k];

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

if (arr == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

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

arr[i] = rand() % 10000;

}

clock\_t start\_time, end\_time;

double time\_used;

start\_time = clock();

heapSort(arr, n);

end\_time = clock();

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

printf("Time taken to sort %d elements using Heapsort: %lf seconds\n", n, time\_used);

free(arr);

}

return 0;

}

Q.2) Write a program to implement Strassen’s Matrix multiplication

Sol

#include <stdio.h>

#include <stdlib.h>

void strassen(int n, int A[][n], int B[][n], int C[][n]) {

if (n == 1) {

C[0][0] = A[0][0] \* B[0][0];

return;

}

int i, j;

int newSize = n / 2;

int A11[newSize][newSize], A12[newSize][newSize], A21[newSize][newSize], A22[newSize][newSize];

int B11[newSize][newSize], B12[newSize][newSize], B21[newSize][newSize], B22[newSize][newSize];

int C11[newSize][newSize], C12[newSize][newSize], C21[newSize][newSize], C22[newSize][newSize];

int M1[newSize][newSize], M2[newSize][newSize], M3[newSize][newSize], M4[newSize][newSize], M5[newSize][newSize], M6[newSize][newSize], M7[newSize][newSize];

int temp1[newSize][newSize], temp2[newSize][newSize];

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

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

A11[i][j] = A[i][j];

A12[i][j] = A[i][j + newSize];

A21[i][j] = A[i + newSize][j];

A22[i][j] = A[i + newSize][j + newSize];

B11[i][j] = B[i][j];

B12[i][j] = B[i][j + newSize];

B21[i][j] = B[i + newSize][j];

B22[i][j] = B[i + newSize][j + newSize];

}

}

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

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

temp1[i][j] = A11[i][j] + A22[i][j];

temp2[i][j] = B11[i][j] + B22[i][j];

}

}

strassen(newSize, temp1, temp2, M1);

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

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

temp1[i][j] = A21[i][j] + A22[i][j];

}

}

strassen(newSize, temp1, B11, M2);

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

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

temp1[i][j] = B12[i][j] - B22[i][j];

}

}

strassen(newSize, A11, temp1, M3);

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

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

temp1[i][j] = B21[i][j] - B11[i][j];

}

}

strassen(newSize, A22, temp1, M4);

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

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

temp1[i][j] = A11[i][j] + A12[i][j];

}

}

strassen(newSize, temp1, B22, M5);

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

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

temp1[i][j] = A21[i][j] - A11[i][j];

temp2[i][j] = B11[i][j] + B12[i][j];

}

}

strassen(newSize, temp1, temp2, M6);

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

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

temp1[i][j] = A12[i][j] - A22[i][j];

temp2[i][j] = B21[i][j] + B22[i][j];

}

}

strassen(newSize, temp1, temp2, M7);

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

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

C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];

}

}

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

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

C12[i][j] = M3[i][j] + M5[i][j];

}

}

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

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

C21[i][j] = M2[i][j] + M4[i][j];

}

}

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

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

C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];

}

}

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

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

C[i][j] = C11[i][j];

C[i][j + newSize] = C12[i][j];

C[i + newSize][j] = C21[i][j];

C[i + newSize][j + newSize] = C22[i][j];

}

}

}

void printMatrix(int n, int mat[][n]) {

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

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

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

}

printf("\n");

}

}

int main() {

int n = 2;

int A[2][2] = {{1, 2}, {3, 4}};

int B[2][2] = {{5, 6}, {7, 8}};

int C[2][2];

strassen(n, A, B, C);

printf("Matrix A:\n");

printMatrix(n, A);

printf("\nMatrix B:\n");

printMatrix(n, B);

printf("\nResultant Matrix C:\n");

printMatrix(n, C);

return 0;

}

Slip3

Q.1) Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements

Sol

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; 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 pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int n;

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

scanf("%d", &n);

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

if (arr == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

printf("Enter %d integers: ", n);

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

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

}

clock\_t start\_time, end\_time;

double time\_used;

start\_time = clock();

quickSort(arr, 0, n - 1);

end\_time = clock();

time\_used = ((double)(end\_time - start\_time))

printf("Sorted array in ascending order:\n");

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

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

}

printf("\n");

printf("Time taken for sorting: %lf seconds\n", time\_used);

free(arr);

return 0;

}

Q.2) Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prims algorithm

Sol

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 5

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

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == 0 && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

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

printf("Edge Weight\n");

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

printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

}

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

int parent[V];

int key[V];

int mstSet[V];

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, mstSet);

mstSet[u] = 1;

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

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

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

}

}

}

printMST(parent, graph);

}

int main() {

int graph[V][V] = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}

};

primMST(graph);

return 0;

}

Slip4

Q.1) Write a program to implement a Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements

Sol

#include <stdio.h>

#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);

}

}

int main() {

int n;

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

scanf("%d", &n);

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

if (arr == NULL) {

printf("Memory allocation failed.\n");

return 1;

}

printf("Enter %d integers: ", n);

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

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

}

clock\_t start\_time, end\_time;

double time\_used;

start\_time = clock();

mergeSort(arr, 0, n - 1);

end\_time = clock();

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

printf("Sorted array in ascending order:\n");

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

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

}

printf("\n");

printf("Time taken for sorting: %lf seconds\n", time\_used);

free(arr);

return 0;

}

Q.2) Write a program to implement Knapsack problems using Greedy method

Sol

#include <stdio.h>

#include <stdlib.h>

struct Item {

int value, weight;

};

int compare(const void \*a, const void \*b) {

double ratio1 = (double)(((struct Item \*)a)->value) / (((struct Item \*)a)->weight);

double ratio2 = (double)(((struct Item \*)b)->value) / (((struct Item \*)b)->weight);

if (ratio1 < ratio2)

return 1;

else if (ratio1 > ratio2)

return -1;

return 0;

}

void knapsackGreedy(struct Item items[], int n, int capacity) {

qsort(items, n, sizeof(struct Item), compare);

int currentWeight = 0;

double finalValue = 0.0;

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

if (currentWeight + items[i].weight <= capacity) {

currentWeight += items[i].weight;

finalValue += items[i].value;

} else {

int remainingWeight = capacity - currentWeight;

finalValue += items[i].value \* ((double)remainingWeight / items[i].weight);

break;

}

}

printf("Maximum value in Knapsack = %.2lf\n", finalValue);

}

int main() {

int n, capacity;

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

scanf("%d", &n);

struct Item \*items = (struct Item \*)malloc(n \* sizeof(struct Item));

printf("Enter the value and weight of each item:\n");

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

scanf("%d %d", &items[i].value, &items[i].weight);

}

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

scanf("%d", &capacity);

knapsackGreedy(items, n, capacity);

free(items);

return 0;

}

Slip5

Q.1) Write a program for the Implementation of Kruskal’s algorithm to find minimum cost spanning tree.

Sol

#include <stdio.h>

#include <stdlib.h>

#define V 6

#define E 9

struct Edge {

int src, dest, weight;

};

struct Graph {

int V, E;

struct Edge \*edge;

};

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

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->V = V;

graph->E = E;

graph->edge = (struct Edge\*)malloc(E \* sizeof(struct Edge));

return graph;

}

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) {

int xset = find(parent, x);

int yset = find(parent, y);

parent[xset] = yset;

}

int myComp(const void\* a, const void\* b) {

struct Edge\* a1 = (struct Edge\*)a;

struct Edge\* b1 = (struct Edge\*)b;

return a1->weight > b1->weight;

}

void KruskalMST(struct Graph\* graph) {

int \*parent = (int\*)malloc(V \* sizeof(int));

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

parent[i] = -1;

struct Edge result[V];

int e = 0;

int i = 0;

qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

while (e < V - 1 && i < graph->E) {

struct Edge next\_edge = graph->edge[i++];

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

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

if (x != y) {

result[e++] = next\_edge;

Union(parent, x, y);

}

}

printf("Edges of Minimum Spanning Tree:\n");

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

printf("%d - %d : %d\n", result[i].src, result[i].dest, result[i].weight);

}

int main() {

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

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

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

graph->edge[0].weight = 4;

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 = 2;

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

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

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

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

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

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

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

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

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

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

graph->edge[6].dest = 4;

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

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

graph->edge[7].dest = 5;

graph->edge[7].weight = 2;

graph->edge[8].src = 2;

graph->edge[8].dest = 5;

graph->edge[8].weight = 3;

KruskalMST(graph);

return 0;

}

Q.2) Write a program to implement Huffman Code using greedy methods and also calculate the best case and worst-case complexity.

Sol

#include <stdio.h>

#include <stdlib.h>

#define MAX\_TREE\_HT 100

struct MinHeapNode {

char data;

unsigned freq;

struct MinHeapNode \*left, \*right;

};

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHeapNode \*\*array;

};

struct MinHeapNode\* newNode(char data, unsigned freq) {

struct MinHeapNode\* temp = (struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

struct MinHeap\* createMinHeap(unsigned capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*\*)malloc(minHeap->capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b) {

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

void minHeapify(struct MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

int isSizeOne(struct MinHeap\* minHeap) {

return (minHeap->size == 1);

}

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap) {

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

void buildMinHeap(struct MinHeap\* minHeap) {

int n = minHeap->size - 1;

int i;

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

minHeapify(minHeap, i);

}

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

int i;

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

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

printf("\n");

}

int isLeaf(struct MinHeapNode\* root) {

return !(root->left) && !(root->right);

}

struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size) {

struct MinHeap\* minHeap = createMinHeap(size);

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

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size) {

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

struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

void printCodes(struct MinHeapNode\* root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

if (isLeaf(root)) {

printf("%c: ", root->data);

printArr(arr, top);

}

}

void HuffmanCodes(char data[], int freq[], int size) {

struct MinHeapNode\* root = buildHuffmanTree(data, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

int main() {

char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};

int freq[] = {5, 9, 12, 13, 16, 45};

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

printf("Huffman Codes:\n");

HuffmanCodes(arr, freq, size);

return 0;

}

Slip 6

Q-1) Write a program for the Implementation of Prim’s algorithm to find minimum cost spanning tree.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 5

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

int min = INT\_MAX, min\_index;

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

if (mstSet[v] == 0 && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

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

printf("Edge Weight\n");

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

printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

}

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

int parent[V];

int key[V];

int mstSet[V];

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

key[i] = INT\_MAX;

mstSet[i] = 0;

}

key[0] = 0;

parent[0] = -1;

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

int u = minKey(key, mstSet);

mstSet[u] = 1;

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

if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {

parent[v] = u;

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

}

}

}

printMST(parent, graph);

}

int main() {

int graph[V][V] = {

{0, 2, 0, 6, 0},

{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},

{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}

};

primMST(graph);

return 0;

}

Q.2) Write a Program to find only length of Longest Common Subsequence.

Sol

#include <stdio.h>

#include <string.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

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

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

int i, j;

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

for (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);

printf("Length of Longest Common Subsequence is %d\n", lcs\_length(X, Y, m, n));

return 0;

}

Slip7

Q-1) Write a program for the Implementation of Dijkstra’s algorithm to find shortest path to other vertices

Sol

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

#define V 9

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

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int dist[]) {

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

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

int dist[V];

int sptSet[V];

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

dist[i] = INT\_MAX, sptSet[i] = 0;

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = 1;

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] = {

{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}

};

dijkstra(graph, 0);

return 0;

}

Q.2) Write a program for finding Topological sorting for Directed Acyclic Graph (DAG)

Sol

#include<stdio.h>

#include<stdlib.h>

#define V 6

struct Node {

int data;

struct Node\* next;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

void addEdge(struct Node\* adj[], int src, int dest) {

struct Node\* newNode = createNode(dest);

newNode->next = adj[src];

adj[src] = newNode;

}

void printGraph(struct Node\* adj[]) {

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

struct Node\* temp = adj[i];

printf("Vertex %d: ", i);

while (temp != NULL) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("NULL\n");

}

}

void topologicalSortUtil(int v, struct Node\* adj[], int visited[], int stack[]) {

visited[v] = 1;

struct Node\* temp = adj[v];

while (temp != NULL) {

if (!visited[temp->data]) {

topologicalSortUtil(temp->data, adj, visited, stack);

}

temp = temp->next;

}

stack[--top] = v;

}

void topologicalSort(struct Node\* adj[]) {

int visited[V];

int stack[V];

int top = V;

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

visited[i] = 0;

}

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

if (!visited[i]) {

topologicalSortUtil(i, adj, visited, stack);

}

}

printf("Topological Sort: ");

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

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

}

printf("\n");

}

int main() {

struct Node\* adj[V];

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

adj[i] = NULL;

}

addEdge(adj, 5, 2);

addEdge(adj, 5, 0);

addEdge(adj, 4, 0);

addEdge(adj, 4, 1);

addEdge(adj, 2, 3);

addEdge(adj, 3, 1);

printf("Graph:\n");

printGraph(adj);

topologicalSort(adj);

return 0;

}

Slip 8

Q.1) Write a program to implement Fractional Knapsack problems using Greedy Method

Sol

#include<stdio.h>

#include<stdlib.h>

struct Item {

int value, weight;

};

int compare(const void \*a, const void \*b) {

double ratio1 = (double)(((struct Item \*)a)->value) / (((struct Item \*)a)->weight);

double ratio2 = (double)(((struct Item \*)b)->value) / (((struct Item \*)b)->weight);

if (ratio1 < ratio2)

return 1;

else if (ratio1 > ratio2)

return -1;

return 0;

}

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

qsort(arr, n, sizeof(arr[0]), compare);

int curWeight = 0;

double finalValue = 0.0;

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

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

curWeight += arr[i].weight;

finalValue += arr[i].value;

} else {

int remain = W - curWeight;

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

break;

}

}

return finalValue;

}

int main() {

int W = 50;

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

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

printf("Maximum value in Knapsack = %.2lf\n", fractionalKnapsack(W, arr, n));

return 0;

}

Q.2) Write Program to implement Traveling Salesman Problem using nearest neighbor algorithm

Sol

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 4

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

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

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

printf("Edge Weight\n");

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

printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

void nearestNeighbor(int graph[V][V], int start) {

int parent[V];

int dist[V];

int visited[V];

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

dist[i] = INT\_MAX;

visited[i] = 0;

}

dist[start] = 0;

parent[start] = -1;

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

int u = minDistance(dist, visited);

visited[u] = 1;

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

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

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

parent[v] = u;

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

}

}

printSolution(parent, graph);

}

int main() {

int graph[V][V] = {{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}};

nearestNeighbor(graph, 0);

return 0;

}

Slip9

Q.1) Write a program to implement optimal binary search tree and also calculate the best-case complexity.

Sol

#include <stdio.h>

#include <limits.h>

int sum(int freq[], int i, int j) {

int s = 0;

for (int k = i; k <= j; k++) {

s += freq[k];

}

return s;

}

int optimalSearchTree(int keys[], int freq[], int n) {

int cost[n][n];

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

cost[i][i] = freq[i];

}

for (int L = 2; L <= n; L++) {

for (int i = 0; i <= n - L + 1; i++) {

int j = i + L - 1;

cost[i][j] = INT\_MAX;

for (int r = i; r <= j; r++) {

int c = ((r > i) ? cost[i][r - 1] : 0) +

((r < j) ? cost[r + 1][j] : 0) +

sum(freq, i, j);

if (c < cost[i][j])

cost[i][j] = c;

}

}

}

return cost[0][n - 1];

}

int main() {

int keys[] = {10, 12, 20};

int freq[] = {34, 8, 50};

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

printf("Cost of Optimal BST is: %d\n", optimalSearchTree(keys, freq, n));

return 0;

}

Q.2) Write a program to implement Sum of Subset by Backtracking

Sol

#include <stdio.h>

#define MAX 10

int total\_nodes;

int solution[MAX];

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

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

if (solution[i] == 1) {

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

}

}

printf("\n");

}

void subsetSum(int arr[], int size, int sum, int currSum, int index, int limit) {

if (currSum == sum) {

printSubset(arr, size);

if (index + 1 < size && currSum - arr[index] + arr[index + 1] <= sum) {

subsetSum(arr, size, sum, currSum - arr[index], index + 1, limit);

}

return;

} else {

if (index < size && currSum + arr[index] <= sum) {

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

solution[i] = 1;

if (currSum + arr[i] <= sum) {

subsetSum(arr, size, sum, currSum + arr[i], i + 1, limit - 1);

}

solution[i] = 0;

}

}

}

}

void generateSubsets(int arr[], int size, int sum) {

int total = 0;

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

total += arr[i];

}

if (arr[0] <= sum && total >= sum) {

subsetSum(arr, size, sum, 0, 0, size);

}

}

int main() {

int arr[] = {10, 7, 5, 18, 12, 20, 15};

int sum = 35;

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

printf("Subsets with sum %d are:\n", sum);

generateSubsets(arr, size, sum);

return 0;

}

Slip 10

Q.1)Write a program to implement Huffman Code using greedy methods

Sol

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_TREE\_HT 100

struct MinHeapNode {

char data;

unsigned freq;

struct MinHeapNode \*left, \*right;

};

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHeapNode \*\*array;

};

struct MinHeapNode\* newNode(char data, unsigned freq) {

struct MinHeapNode\* temp = (struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

struct MinHeap\* createMinHeap(unsigned capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*\*)malloc(minHeap->capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b) {

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

void minHeapify(struct MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

int isSizeOne(struct MinHeap\* minHeap) {

return (minHeap->size == 1);

}

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap) {

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

void buildMinHeap(struct MinHeap\* minHeap) {

int n = minHeap->size - 1;

int i;

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

minHeapify(minHeap, i);

}

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

int i;

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

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

printf("\n");

}

int isLeaf(struct MinHeapNode\* root) {

return !(root->left) && !(root->right);

}

struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size) {

struct MinHeap\* minHeap = createMinHeap(size);

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

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size) {

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

struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

void printCodes(struct MinHeapNode\* root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

if (isLeaf(root)) {

printf("%c: ", root->data);

printArr(arr, top);

}

}

void HuffmanCodes(char data[], int freq[], int size) {

struct MinHeapNode\* root = buildHuffmanTree(data, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

int main() {

char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};

int freq[] = {5, 9, 12, 13, 16, 45};

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

printf("Huffman Codes:\n");

HuffmanCodes(arr, freq, size);

return 0;

}

Q-2) Write a program to solve 4 Queens Problem using Backtracking

Sol

#include <stdio.h>

#include <stdbool.h>

#define N 4

void printSolution(int board[N][N]) {

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

for (int j = 0; j < N; j++)

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

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

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[N][N], int col) {

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ() {

int board[N][N] = {{0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0}};

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

printf("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

int main() {

solveNQ();

return 0;

}

Slip 11

Q.1) Write a programs to implement DFS (Depth First Search) and determine the time complexity for the same.

Sol

#include <stdio.h>

#include <stdlib.h>

#define MAX\_NODES 100

struct Node {

int data;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

int\* visited;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct Node\*\*)malloc(vertices \* sizeof(struct Node\*));

graph->visited = (int\*)malloc(vertices \* sizeof(int));

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

graph->adjLists[i] = NULL;

graph->visited[i] = 0;

}

return graph;

}

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

struct Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

void DFS(struct Graph\* graph, int vertex) {

struct Node\* adjList = graph->adjLists[vertex];

struct Node\* temp = adjList;

graph->visited[vertex] = 1;

printf("Visited %d \n", vertex);

while (temp != NULL) {

int connectedVertex = temp->data;

if (graph->visited[connectedVertex] == 0) {

DFS(graph, connectedVertex);

}

temp = temp->next;

}

}

void printGraph(struct Graph\* graph) {

for (int v = 0; v < graph->numVertices; v++) {

struct Node\* temp = graph->adjLists[v];

printf("Adjacency list of vertex %d\n ", v);

while (temp) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("\n");

}

}

int main() {

struct Graph\* graph = createGraph(4);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

printf("Graph:\n");

printGraph(graph);

printf("\nDFS traversal:\n");

DFS(graph, 0);

return 0;

}

Q.2) Write a program to find shortest paths from a given vertex in a weighted connected graph, to other vertices using Dijkstra’s algorithm.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define V 9

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

int min = INT\_MAX, min\_index;

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

if (sptSet[v] == 0 && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int dist[]) {

printf("Vertex Distance from Source\n");

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

printf("%d \t\t %d\n", i, dist[i]);

}

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

int dist[V];

int sptSet[V];

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

dist[i] = INT\_MAX;

sptSet[i] = 0;

}

dist[src] = 0;

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

int u = minDistance(dist, sptSet);

sptSet[u] = 1;

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] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

{0, 0, 7, 0, 9, 14, 0, 0, 0},

{0, 0, 0, 9, 0, 10, 0, 0, 0},

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}};

dijkstra(graph, 0);

return 0;

}

Slip 12

Q.1) Write a program to implement BFS (Breadth First Search) and determine the time complexity for the same.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_NODES 100

struct Node {

int data;

struct Node\* next;

};

struct Queue {

struct Node \*front, \*rear;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

bool\* visited;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct Node\*\*)malloc(vertices \* sizeof(struct Node\*));

graph->visited = (bool\*)malloc(vertices \* sizeof(bool));

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

graph->adjLists[i] = NULL;

graph->visited[i] = false;

}

return graph;

}

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

struct Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

void enqueue(struct Queue\* queue, int data) {

struct Node\* newNode = createNode(data);

if (queue->rear == NULL) {

queue->front = queue->rear = newNode;

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

}

int dequeue(struct Queue\* queue) {

if (queue->front == NULL)

return -1;

struct Node\* temp = queue->front;

int data = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL)

queue->rear = NULL;

free(temp);

return data;

}

bool isEmpty(struct Queue\* queue) {

return queue->front == NULL;

}

void BFS(struct Graph\* graph, int startVertex) {

struct Queue\* queue = createQueue();

graph->visited[startVertex] = true;

enqueue(queue, startVertex);

while (!isEmpty(queue)) {

int currentVertex = dequeue(queue);

printf("Visited %d \n", currentVertex);

struct Node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->data;

if (!graph->visited[adjVertex]) {

graph->visited[adjVertex] = true;

enqueue(queue, adjVertex);

}

temp = temp->next;

}

}

}

void printGraph(struct Graph\* graph) {

for (int v = 0; v < graph->numVertices; v++) {

struct Node\* temp = graph->adjLists[v];

printf("Adjacency list of vertex %d\n ", v);

while (temp) {

printf("%d -> ", temp->data);

temp = temp->next;

}

printf("\n");

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

addEdge(graph, 3, 5);

addEdge(graph, 4, 5);

printf("Graph:\n");

printGraph(graph);

printf("\nBFS traversal:\n");

BFS(graph, 0);

return 0;

}

Q.2) Write a program to sort a given set of elements using the Selection sort method and determine the time required to sort the elements.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

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

int i, j, min\_idx;

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

min\_idx = i;

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

if (arr[j] < arr[min\_idx])

min\_idx = j;

}

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

int main() {

int n;

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

scanf("%d", &n);

int arr[n];

printf("Enter the elements: ");

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

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

}

clock\_t start, end;

double cpu\_time\_used;

start = clock();

selectionSort(arr, n);

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: %f seconds\n", cpu\_time\_used);

return 0;

}

Slip13

Q.1) Write a program to find minimum number of multiplications in Matrix Chain Multiplication.

Sol

#include <stdio.h>

#include <limits.h>

#define MAX\_SIZE 100

int MatrixChainOrder(int p[], int n) {

int m[n][n];

int i, j, k, L, q;

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

m[i][i] = 0;

for (L = 2; L < n; L++) {

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

j = i + L - 1;

m[i][j] = INT\_MAX;

for (k = i; k <= j - 1; k++) {

q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j];

if (q < m[i][j])

m[i][j] = q;

}

}

}

return m[1][n - 1];

}

int main() {

int arr[] = {10, 20, 30, 40, 30};

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

printf("Minimum number of multiplications is %d", MatrixChainOrder(arr, n));

return 0;

}

Q.2) Write a program to implement an optimal binary search tree and also calculate the best case and worst-case complexity.

Sol

#include <stdio.h>

#include <limits.h>

#define MAX\_SIZE 100

int optimalBST(int keys[], int freq[], int n) {

int cost[n][n];

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

cost[i][i] = freq[i];

}

for (int L = 2; L <= n; L++) {

for (int i = 0; i <= n - L + 1; i++) {

int j = i + L - 1;

cost[i][j] = INT\_MAX;

for (int r = i; r <= j; r++) {

int c = ((r > i) ? cost[i][r - 1] : 0) +

((r < j) ? cost[r + 1][j] : 0) +

freq[i] + freq[j];

if (c < cost[i][j])

cost[i][j] = c;

}

}

}

return cost[0][n - 1];

}

int main() {

int keys[] = {10, 12, 20};

int freq[] = {34, 8, 50};

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

printf("Cost of Optimal BST is: %d\n", optimalBST(keys, freq, n));

return 0;

}

Slip 14

Q.1) Write a program to sort a list of n numbers in ascending order using Insertion sort and determine the time required to sort the elements.

Sol

#include <stdio.h>

#include <time.h>

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 n;

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

scanf("%d", &n);

int arr[n];

printf("Enter the elements: ");

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

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

}

clock\_t start, end;

double cpu\_time\_used;

start = clock();

insertionSort(arr, n);

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: %f seconds\n", cpu\_time\_used);

return 0;

}

Q.2) Write a program to implement DFS and BFS. Compare the time complexity

Sol

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_NODES 100

struct Node {

int data;

struct Node\* next;

};

struct Stack {

struct Node\* top;

};

struct Queue {

struct Node \*front, \*rear;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

bool\* visited;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct Stack\* createStack() {

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->top = NULL;

return stack;

}

void push(struct Stack\* stack, int data) {

struct Node\* newNode = createNode(data);

newNode->next = stack->top;

stack->top = newNode;

}

int pop(struct Stack\* stack) {

if (stack->top == NULL)

return -1;

struct Node\* temp = stack->top;

int data = temp->data;

stack->top = stack->top->next;

free(temp);

return data;

}

bool isStackEmpty(struct Stack\* stack) {

return stack->top == NULL;

}

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

void enqueue(struct Queue\* queue, int data) {

struct Node\* newNode = createNode(data);

if (queue->rear == NULL) {

queue->front = queue->rear = newNode;

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

}

int dequeue(struct Queue\* queue) {

if (queue->front == NULL)

return -1;

struct Node\* temp = queue->front;

int data = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL)

queue->rear = NULL;

free(temp);

return data;

}

bool isQueueEmpty(struct Queue\* queue) {

return queue->front == NULL;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct Node\*\*)malloc(vertices \* sizeof(struct Node\*));

graph->visited = (bool\*)malloc(vertices \* sizeof(bool));

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

graph->adjLists[i] = NULL;

graph->visited[i] = false;

}

return graph;

}

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

struct Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

void DFS(struct Graph\* graph, int vertex) {

struct Stack\* stack = createStack();

push(stack, vertex);

graph->visited[vertex] = true;

while (!isStackEmpty(stack)) {

int currentVertex = pop(stack);

printf("Visited %d \n", currentVertex);

struct Node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->data;

if (!graph->visited[adjVertex]) {

push(stack, adjVertex);

graph->visited[adjVertex] = true;

}

temp = temp->next;

}

}

}

void BFS(struct Graph\* graph, int startVertex) {

struct Queue\* queue = createQueue();

graph->visited[startVertex] = true;

enqueue(queue, startVertex);

while (!isQueueEmpty(queue)) {

int currentVertex = dequeue(queue);

printf("Visited %d \n", currentVertex);

struct Node\* temp = graph->adjLists[currentVertex];

while (temp) {

int adjVertex = temp->data;

if (!graph->visited[adjVertex]) {

graph->visited[adjVertex] = true;

enqueue(queue, adjVertex);

}

temp = temp->next;

}

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 4);

addEdge(graph, 3, 4);

addEdge(graph, 3, 5);

addEdge(graph, 4, 5);

printf("DFS traversal:\n");

clock\_t start\_dfs = clock();

DFS(graph, 0);

clock\_t end\_dfs = clock();

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

printf("Time taken for DFS: %f seconds\n", cpu\_time\_used\_dfs);

for (int i = 0; i < graph->numVertices; i++)

graph->visited[i] = false;

printf("\nBFS traversal:\n");

clock\_t start\_bfs = clock();

BFS(graph, 0);

clock\_t end\_bfs = clock();

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

printf("Time taken for BFS: %f seconds\n", cpu\_time\_used\_bfs);

return 0;

}

Slip 15

Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using LCBB (Least Cost Branch and Bound).

Sol

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Item {

int value;

int weight;

float ratio;

};

int compare(const void\* a, const void\* b) {

float ratio\_a = ((struct Item\*)a)->ratio;

float ratio\_b = ((struct Item\*)b)->ratio;

return ratio\_b > ratio\_a ? 1 : -1;

}

int knapsack(int W, struct Item arr[], int n) {

qsort(arr, n, sizeof(arr[0]), compare);

int totalValue = 0, totalWeight = 0;

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

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

totalWeight += arr[i].weight;

totalValue += arr[i].value;

} else {

int remainingWeight = W - totalWeight;

totalValue += (arr[i].value \* remainingWeight) / arr[i].weight;

break;

}

}

return totalValue;

}

int main() {

int n, W;

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

scanf("%d", &n);

struct Item items[n];

printf("Enter the weight and value of each item:\n");

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

printf("Item %d: ", i + 1);

scanf("%d %d", &items[i].weight, &items[i].value);

items[i].ratio = (float)items[i].value / items[i].weight;

}

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

scanf("%d", &W);

int maxValue = knapsack(W, items, n);

printf("Maximum value in the knapsack: %d\n", maxValue);

return 0;

}

Q.2) Write a program to implement Graph Coloring Algorithm

Sol

#include <stdio.h>

#include <stdbool.h>

#define V 4

bool isSafe(int v, bool graph[V][V], int color[], int c) {

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

if (graph[v][i] && c == color[i])

return false;

return true;

}

bool graphColoringUtil(bool graph[V][V], int m, int color[], int v) {

if (v == V)

return true;

for (int c = 1; c <= m; c++) {

if (isSafe(v, graph, color, c)) {

color[v] = c;

if (graphColoringUtil(graph, m, color, v + 1))

return true;

color[v] = 0;

}

}

return false;

}

bool graphColoring(bool graph[V][V], int m) {

int color[V];

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

color[i] = 0;

if (!graphColoringUtil(graph, m, color, 0)) {

printf("Solution does not exist");

return false;

}

printf("Solution exists: \n");

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

printf("Vertex %d ---> Color %d\n", i + 1, color[i]);

return true;

}

int main() {

bool graph[V][V] = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3; // Number of colors

graphColoring(graph, m);

return 0;

}

Slip 16

Q.1) Write a program to implement to find out solution for 0/1 knapsack problem using dynamic programming.

Sol

#include <stdio.h>

#define max(a, b) ((a > b) ? a : b)

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

int i, w;

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

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

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

int W = 50;

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

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

return 0;

}

Q.2) Write a program to determine if a given graph is a Hamiltonian cycle or not.

Sol

#include <stdio.h>

#include <stdbool.h>

#define V 5 // Number of vertices

bool isSafe(int v, bool graph[V][V], int path[], int pos) {

if (!graph[path[pos - 1]][v])

return false;

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

if (path[i] == v)

return false;

return true;

}

bool hamCycleUtil(bool graph[V][V], int path[], int pos) {

if (pos == V) {

if (graph[path[pos - 1]][path[0]])

return true;

else

return false;

}

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

if (isSafe(v, graph, path, pos)) {

path[pos] = v;

if (hamCycleUtil(graph, path, pos + 1))

return true;

path[pos] = -1;

}

}

return false;

}

bool hamCycle(bool graph[V][V]) {

int path[V];

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

path[i] = -1;

path[0] = 0;

if (!hamCycleUtil(graph, path, 1)) {

printf("No Hamiltonian cycle exists");

return false;

}

printf("Hamiltonian cycle exists: \n");

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

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

printf("%d\n", path[0]);

return true;

}

int main() {

bool graph[V][V] = {

{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0}

};

hamCycle(graph);

return 0;

}

Slip 17

Q.1) Write a program to implement solve ‘N’ Queens Problem using Backtracking.

Sol

#include <stdio.h>

#include <stdbool.h>

#define N 8

void printSolution(int board[N][N]) {

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

for (int j = 0; j < N; j++)

printf("%2d ", board[i][j]);

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

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[N][N], int col) {

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ() {

int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},

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

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

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

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

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

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

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

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

printf("Solution does not exist");

return false;

}

printf("Solution exists: \n");

printSolution(board);

return true;

}

int main() {

solveNQ();

return 0;

}

Q.2) Write a program to find out solution for 0/1 knapsack problem.

Sol

#include <stdio.h>

#define max(a, b) ((a > b) ? a : b)

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

int i, w;

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

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

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

int W = 50;

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

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

return 0;

}

Slip 18

Q.1) Write a program to implement Graph Coloring Algorithm.

Sol

#include <stdio.h>

#include <stdbool.h>

#define V 4 // Number of vertices

void printSolution(int color[]) {

printf("Vertex\tColor\n");

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

printf("%d\t%d\n", i, color[i]);

}

bool isSafe(int v, bool graph[V][V], int color[], int c) {

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

if (graph[v][i] && c == color[i])

return false;

return true;

}

bool graphColoringUtil(bool graph[V][V], int m, int color[], int v) {

if (v == V)

return true;

for (int c = 1; c <= m; c++) {

if (isSafe(v, graph, color, c)) {

color[v] = c;

if (graphColoringUtil(graph, m, color, v + 1))

return true;

color[v] = 0;

}

}

return false;

}

bool graphColoring(bool graph[V][V], int m) {

int color[V];

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

color[i] = 0;

if (!graphColoringUtil(graph, m, color, 0)) {

printf("Solution does not exist");

return false;

}

printf("Solution exists: \n");

printSolution(color);

return true;

}

int main() {

bool graph[V][V] = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3; // Number of colors

graphColoring(graph, m);

return 0;

}

Q.2) Write a program to find out live node, E node and dead node from a given graph.

Sol

#include <stdio.h>

#include <stdbool.h>

#define V 4

void findNodes(bool graph[V][V]) {

bool visited[V];

int liveCount = 0, eNodeCount = 0, deadCount = 0;

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

visited[i] = false;

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

if (!visited[i]) {

bool live = false, eNode = true;

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

if (graph[i][j]) {

live = true;

if (i != j)

eNode = false;

visited[j] = true;

}

}

if (live)

liveCount++;

if (eNode)

eNodeCount++;

if (!live)

deadCount++;

}

}

printf("Live Nodes: %d\n", liveCount);

printf("E-Nodes: %d\n", eNodeCount);

printf("Dead Nodes: %d\n", deadCount);

}

int main() {

bool graph[V][V] = {

{0, 1, 1, 0},

{1, 0, 0, 1},

{1, 0, 0, 1},

{0, 1, 1, 0}

};

findNodes(graph);

return 0;

}

Slip 19

Q.1) Write a program to determine if a given graph is a Hamiltonian cycle or Not

Sol

#include <stdio.h>

#include <stdbool.h>

#define V 5 // Number of vertices

bool isSafe(int v, bool graph[V][V], int path[], int pos) {

if (!graph[path[pos - 1]][v])

return false;

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

if (path[i] == v)

return false;

return true;

}

bool hamCycleUtil(bool graph[V][V], int path[], int pos) {

if (pos == V) {

if (graph[path[pos - 1]][path[0]])

return true;

else

return false;

}

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

if (isSafe(v, graph, path, pos)) {

path[pos] = v;

if (hamCycleUtil(graph, path, pos + 1))

return true;

path[pos] = -1;

}

}

return false;

}

bool hamCycle(bool graph[V][V]) {

int path[V];

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

path[i] = -1;

path[0] = 0;

if (!hamCycleUtil(graph, path, 1)) {

printf("No Hamiltonian cycle exists");

return false;

}

printf("Hamiltonian cycle exists: \n");

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

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

printf("%d\n", path[0]);

return true;

}

int main() {

bool graph[V][V] = {

{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0}

};

hamCycle(graph);

return 0;

}

Q.2) Write a program to show board configuration of 4 queens’ problem.

Sol

#include <stdio.h>

#define N 4

void printSolution(int board[N][N]) {

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

for (int j = 0; j < N; j++)

printf("%c ", board[i][j] ? 'Q' : '.');

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

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[N][N], int col) {

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

void solveNQ() {

int board[N][N] = { {0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0},

{0, 0, 0, 0} };

if (!solveNQUtil(board, 0))

printf("Solution does not exist");

else

printSolution(board);

}

int main() {

solveNQ();

return 0;

}

Slip 20

Q.1) Write a program to implement for finding Topological sorting and determine the time complexity for the same.

Sol

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_VERTICES 100

struct Node {

int data;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

bool\* visited;

int\* indegree;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct Node\*\*)malloc(vertices \* sizeof(struct Node\*));

graph->visited = (bool\*)malloc(vertices \* sizeof(bool));

graph->indegree = (int\*)malloc(vertices \* sizeof(int));

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

graph->adjLists[i] = NULL;

graph->visited[i] = false;

graph->indegree[i] = 0;

}

return graph;

}

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

struct Node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

graph->indegree[dest]++;

}

void topologicalSortUtil(struct Graph\* graph, int v, bool visited[], struct Node\*\* stack) {

visited[v] = true;

struct Node\* temp = graph->adjLists[v];

while (temp) {

int adjVertex = temp->data;

if (!visited[adjVertex])

topologicalSortUtil(graph, adjVertex, visited, stack);

temp = temp->next;

}

struct Node\* newNode = createNode(v);

newNode->next = \*stack;

\*stack = newNode;

}

void topologicalSort(struct Graph\* graph) {

bool visited[MAX\_VERTICES];

for (int i = 0; i < graph->numVertices; i++)

visited[i] = false;

struct Node\* stack = NULL;

for (int i = 0; i < graph->numVertices; i++) {

if (!visited[i])

topologicalSortUtil(graph, i, visited, &stack);

}

printf("Topological Sorting: ");

while (stack != NULL) {

printf("%d ", stack->data);

stack = stack->next;

}

}

int main() {

struct Graph\* graph = createGraph(6);

addEdge(graph, 5, 0);

addEdge(graph, 5, 2);

addEdge(graph, 4, 0);

addEdge(graph, 4, 1);

addEdge(graph, 2, 3);

addEdge(graph, 3, 1);

printf("Topological Sorting: ");

topologicalSort(graph);

return 0;

}

Q.2) Write a program to solve N Queens Problem using Backtracking.

Sol

#include <stdio.h>

#include <stdbool.h>

#define N 8

void printSolution(int board[N][N]) {

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

for (int j = 0; j < N; j++)

printf("%c ", board[i][j] ? 'Q' : '.');

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

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[N][N], int col) {

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ() {

int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},

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

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

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

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

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

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

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

if (!solveNQUtil(board, 0)) {

printf("Solution does not exist\n");

return false;

}

printf("Solution exists:\n");

printSolution(board);

return true;

}

int main() {

solveNQ();

return 0;

}