

SESSION 2023-24

DAA

(Design and Analysis of Algorithms)

Lab File

**COURSE:- BCA**

**ROLL NO :- 41221139**

**SUBMITTED BY :- SUBMITTED TO:-**

**Sachin Rajbhar Dr. Kavita Khanna**

|  |  |  |  |
| --- | --- | --- | --- |
| I N D E X | | | |
| S.NO | **P R A C T I C A L S** | **DATE** | **SIGN.** |
| 1 | Write a program to perform operation count for a given pseudo code. |  |  |
| 2 | WAP for Recursive Binary and Linear search. |  |  |
| 3 | WAP to perform Bubble sort for any given list of numbers. |  |  |
| 4 | WAP to perform Insertion sort for any given list of numbers. |  |  |
| 5 | WAP to perform Selection Sort for the given list of integer values. |  |  |
| 6 | WAP to perform Quick Sort for the given list of integer values. |  |  |
| 7 | WAP to perform Heap Sort for the given list of integer values. |  |  |
| 8 | Write a program to find Maximum and Minimum of the given set of integer values. |  |  |
| 9 | Write a Program to perform Merge Sort on the given two lists of integer values. |  |  |
| 10 | Write a Program to perform Binary Search for a given set of integer values recursively and non-recursively. |  |  |
| 11 | Write a program to find solution for knapsack problem using greedy method. |  |  |
| 12 | Write a program to find minimum cost spanning tree using Prim’s Algorithm. |  |  |
| 13 | Write a program to find minimum cost spanning tree using Kruskal’s Algorithm. |  |  |
| 14 | Write a program to perform Single source shortest path problem for a given graph. |  |  |
| 15 | Write a program to find solution for job sequencing with deadlines problem. |  |  |
|  |  |  |  |
|  |  |  |  |
|  | \*Write a program to implement N Queen Problem using Backtracking.ss (G1) |  |  |
|  | \*Write a program to perform Travelling Salesman Problem (G2) |  |  |
|  |  |  |  |

**PRACTICAL 1**

Q: Write a program to perform operation count for a given pseudo code.

**CODE:**

#include <stdio.h>

#include <string.h>

int count\_operations(char pseudo\_code[]) {

    int comparison\_count = 0;

    int arithmetic\_count = 0;

    // Loop through each character in the pseudo code

    for (int i = 0; i < strlen(pseudo\_code); i++) {

        // Count the number of comparisons

        if ((pseudo\_code[i] == '=' && pseudo\_code[i+1] == '=') || pseudo\_code[i] == '<' || pseudo\_code[i] == '>') {

            comparison\_count++;

        }

        // Count the number of arithmetic operations

        if (pseudo\_code[i] == '+' || pseudo\_code[i] == '-' || pseudo\_code[i] == '\*' || pseudo\_code[i] == '/') {

            arithmetic\_count++;

        }

    }

    // Print the operation counts

    printf("Comparisons: %d\n", comparison\_count);

    printf("Arithmetic: %d\n", arithmetic\_count);

    return 0;

}

int main() {

    char pseudo\_code[] = "if (x == y) {\n\t z = x + y\n\t print(z)\n}";

    count\_operations(pseudo\_code);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 2**

Q: WAP for Recursive Binary and Linear search.

**CODE:**

#include <stdio.h>

// Recursive binary search function

int binary\_search(int arr[], int left, int right, int target) {

    // Check base case

    if (right >= left) {

        int mid = left + (right - left) / 2;

        // If the element is present at the middle itself

        if (arr[mid] == target)

            return mid;

        // If element is smaller than mid, then it can only be present in left subarray

        if (arr[mid] > target)

            return binary\_search(arr, left, mid - 1, target);

        // Else the element can only be present in right subarray

        return binary\_search(arr, mid + 1, right, target);

    }

    // Element is not present in array

    return -1;

}

// Recursive linear search function

int linear\_search(int arr[], int left, int right, int target) {

    // Check base case

    if (left > right)

        return -1;

    // If element is present at the current index

    if (arr[left] == target)

        return left;

    // Recursively search the rest of the array

    return linear\_search(arr, left + 1, right, target);

}

int main() {

    int arr[] = {2, 5, 7, 9, 12, 16, 18, 22, 24, 26};

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

    int target = 18;

    // Perform binary search

    int binary\_index = binary\_search(arr, 0, n - 1, target);

    if (binary\_index == -1) {

        printf("Element not present in array.\n");

    } else {

        printf("Element found at index %d (using binary search).\n", binary\_index);

    }

    // Perform linear search

    int linear\_index = linear\_search(arr, 0, n - 1, target);

    if (linear\_index == -1) {

        printf("Element not present in array.\n");

    } else {

        printf("Element found at index %d (using linear search).\n", linear\_index);

    }

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 3**

Q: WAP to perform Bubble sort for any given list of numbers.

**CODE:**

#include <stdio.h>

void printArray(int\* A, int n){

    for(int i = 0; i < n; i++)

    {

        printf("%d ", A[i]);

    }

    printf("\n");

}

void bubbleSort(int\* A, int n){

    int isSorted = 0;

    for(int i = 0; i<n-1; ++i) // (n-1) Passes

    {

        printf("Working on pass number %d\n", i+1);

        isSorted = 1;

        for(int j=0; j<n-1-i; j++) // Comparisons in each pa

ss

        {

            if(A[j]>A[j+1])

            {

                A[j+1] = A[j+1]+A[j];

                A[j] = A[j+1]-A[j];

                A[j+1] = A[j+1]-A[j];

                isSorted = 0;

            }

        }

        if(isSorted){

            return;

        }

        printArray(A, n); // printing every pass

    }

}

int main(){

    int A[] = {12, 54, 65, 7, 23, 9};

    // int A[] = {1, 2, 3, 4, 5,6};

    int n = sizeof(A)/sizeof(int);

    printf("Original Array:- ");

    printArray(A, n); // Printing before sorting

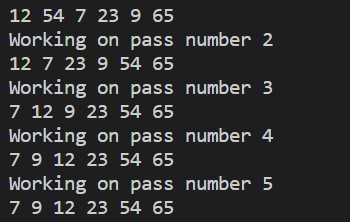
    bubbleSort(A, n);

    printArray(A, n); // Printing after sorting

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 4**

Q: WAP to perform Insertion sort for any given list of numbers

**CODE:**

#include <stdio.h>

void printArray(int\* A, int n){

    for(int i = 0; i < n; i++)

    {

        printf("%d ", A[i]);

    }

    printf("\n");

}

void insertionSort(int \*A, int n){

    int key, j;

    for(int i = 1; i<n; ++i)

    {

        key = A[i]; //All key insert one by one

        j = i-1;

        // Loop for each pass

        while(j>=0 && A[j] > key){ // > for ascending

            A[j+1] = A[j];

            j--;

        }

        A[j+1] = key;

        printArray(A, n); // printing every pass

    }

}

int main(){

    //  -1  0    1    2    3   4    5

    //      12,  54,  65,  7,  23,  9

    int A[] = {12, 54, 65, 7, 23, 9};

    int n = sizeof(A)/sizeof(int);

    printf("Original Array:- ");

    printArray(A, n);

    insertionSort(A, n);

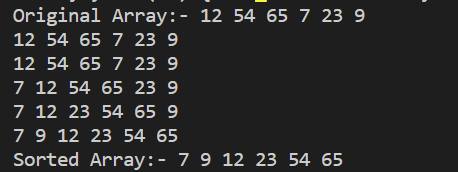
    printf("Sorted Array:- ");

    printArray(A, n);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 5**

Q: WAP to perform Selection Sort for the given list of integer values.

**CODE:**

#include <stdio.h>

void printArray(int\* A, int n){

    for(int i = 0; i < n; i++)

    {

        printf("%d ", A[i]);

    }

    printf("\n");

}

void selectionSort(int \*A, int n){

    int indexOfMin, temp;

    printf("Running Selection Sort...\n");

    for(int i=0; i<(n-1); i++)

    {

        indexOfMin = i;

        for(int j=i+1; j<n; j++)

        {

            if(A[j] < A[indexOfMin])

            {

                indexOfMin = j;

            }

        }

        // Swap A[i] & A[indexOfMin]

        temp = A[i];

        A[i] = A[indexOfMin];

        A[indexOfMin] = temp;

    }

}

int main(){

    // 0  1  3  4   5

    // 3, 5, 2, 13, 2

    int A[] = {3, 5, 2, 13, 2};

    int n = sizeof(A)/sizeof(int);

    printArray(A, n);

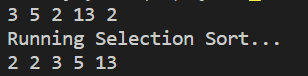
    selectionSort(A, n);

    printArray(A, n);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 6**

Q: WAP to perform Quick Sort for the given list of integer values

**CODE:**

#include <stdio.h>

void printArray(int \*A, int n)

{

    for (int i = 0; i < n; i++)

    {

        printf("%d ", A[i]);

    }

    printf("\n");

}

int partition(int A[], int low, int high) // Partition and swap

{

    int pivot = A[low];

    int i = low + 1;

    int j = high;

    int temp;

    do

    {

        while (A[i] <= pivot)

        {

            i++;

        }

        while (A[j] > pivot)

        {

            j--;

        }

        if (i < j)

        {

            temp = A[i];

            A[i] = A[j];

            A[j] = temp;

        }

    } while (i < j);

    // Swap A[low] and A[j]

    temp = A[low];

    A[low] = A[j];

    A[j] = temp;

    return j;

}

void quickSort(int A[], int low, int high)

{

    int partitionIndex; // Index of pivot after partition

    int pass=1;

    if (low < high)

    {

        partitionIndex = partition(A, low, high);

        printArray(A,9);

        quickSort(A, low, partitionIndex - 1);  // sort left subarray first n

        quickSort(A, partitionIndex + 1, high); // sort right subarray

    }

}

int main()

{

    int A[] = {3, 5, 2, 13, 12, 3, 2, 13, 45};

    // 3, 5, 2, 13, 12, 3, 2, 13, 45

    // 3, 2i, 2, 13, 12, 3, 5j, 13, 45

    // 3, 2, 2, 13i, 12, 3j, 5, 13, 45

    // 3, 2, 2, 3j, 12i, 13, 5, 13, 45 --> firs call to partition returns 3

    int n = sizeof(A) / sizeof(int);

    printf("Before Quick Sort: ");

    printArray(A, n);

    quickSort(A, 0, n - 1);

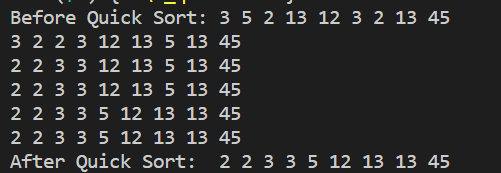
    printf("After Quick Sort:  ");

    printArray(A, n);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 7**

Q: WAP to perform Heap Sort for the given list of integer values.

**CODE:**

#include <stdio.h>

// 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 = 2\*i + 1

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

    // 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] and arr[largest]

        int temp = arr[i];

        arr[i] = arr[largest];

        arr[largest] = temp;

        // Recursively heapify the affected sub-tree

        heapify(arr, n, largest);

    }

}

// Main function to perform heap sort

void heap\_sort(int arr[], int n) {

    // Build heap (rearrange array)

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

        heapify(arr, n, i);

    // One by one extract an element from heap

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

        // Move current root to end

        int temp = arr[0];

        arr[0] = arr[i];

        arr[i] = temp;

        // call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

// Function to print an array

void print\_array(int arr[], int n) {

    for (int i = 0; i < n; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

int main() {

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

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

    printf("Original array:\n");

    print\_array(arr, n);

    heap\_sort(arr, n);

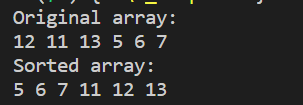
    printf("Sorted array:\n");

    print\_array(arr, n);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 8**

Q: Write a program to find Maximum and Minimum of the given set of integer values.

**CODE:**

#include <stdio.h>

#include <conio.h>

int main()

{

    int a[1000],i,n,min,max;

    printf("Enter the size of the array : ");

    scanf("%d",&n);

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

    for(i=0; i<n; i++)

    {

        scanf("%d",&a[i]);

    }

    // INITIALLY CHOOSE 1st element

    min=max=a[0];

    for(i=1; i<n; i++)

    {

        if(min>a[i])

            min=a[i];

        if(max<a[i])

            max=a[i];

    }

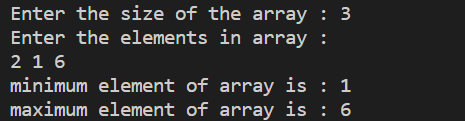
    printf("minimum element of array is : %d",min);

    printf("\nmaximum element of array is : %d",max);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 9**

Q: Write a Program to perform Merge Sort on the given two lists of integer values.

**CODE:**

#include <stdio.h>

void printArray(int \*A, int n)

{

    for (int i = 0; i < n; i++)

    {

        printf("%d ", A[i]);

    }

    printf("\n");

}

void mergeSort(int A[], int B[], int C[], int m, int n)

{

    int i,j,k;

    i=j=k=0;

    while(i<m && i<n){

        if(A[i]<B[j]){

            C[k] = A[i];

            i++;k++;

        }

        else{

          C[k] = B[j];

          j++;k++;

        }

    }

    while(i<m){       // Copy all remaining elements from A to C

        C[k] = A[i];

        k++; i++;

    }

    while(j<n){       // """                         from B to C

        C[k] = A[j];

        k++; j++;

    }

}

void merge(int A[], int mid, int low, int high)

{

    int i, j, k, B[100];

    i=low;

    j=mid+1;

    k=low;

    while(i<=mid && j<=high){

        if(A[i]<A[j]){

            B[k] = A[i];

            i++;

            k++;

        }

        else{

           B[k] = A[j];

           j++;

           k++;

        }

    }

    while(i<=mid){       // Copy all remaining elements from A to C

        B[k] = A[i];

        k++;

        i++;

    }

    while(j<=high){       // """                         from B to C

        B[k] = A[j];

        k++;

        j++;

    }

    for(i=low; i<=high; i++)

    {

        A[i] = B[i];

    }

}

void mergeSortRec(int A[], int low, int high)

{

    int mid;

    if(low<high){

        mid = (low+high)/2;

        mergeSortRec(A,low,mid);

        mergeSortRec(A,mid+1,high);

        merge(A,mid,low,high);

    }

}

int main()

{

    // Merging two arrays

    int A[] = {7, 8, 11};

    int B[] = {1, 2, 3};

    int C[10];

    int m = sizeof(A) / sizeof(int);

    int n = sizeof(B) / sizeof(int);

    printf("Before Merge Sort:-\n");

    printf("A[]: ");

    printArray(A, m);

    printf("B[]: ");

    printArray(B, n);

    mergeSort(A, B, C, m, n);

    printf("After Merge Sort:- ");

    printArray(C,m+n);

    printf("\n\n");

    // Merge Sort then merge

    int X[] = {3, 5, 2, 13, 12, 3, 2, 13, 45};

    int size = sizeof(X) / sizeof(int);

    int low = 0;

    int high = size-1;

    printf("Before Merge Sort in Single Array: ");

    printArray(X, size);

    mergeSortRec(X, low, high);

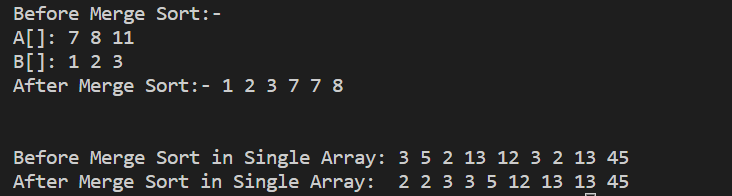
    printf("After Merge Sort in Single Array:  ");

    printArray(X, size);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 10**

Q: Write a Program to perform Binary Search for a given set of integer values recursively and non-recursively.

**CODE:**

#include <stdio.h>

// Function to perform binary search recursively

int binary\_search\_rec(int arr[], int left, int right, int x) {

    if (left <= right) {

        int mid = left + (right - left) / 2;

        // If the element is present at the middle itself

        if (arr[mid] == x)

            return mid;

        // If element is smaller than mid, then it can only be present in left subarray

        if (arr[mid] > x)

            return binary\_search\_rec(arr, left, mid - 1, x);

        // Else the element can only be present in right subarray

        return binary\_search\_rec(arr, mid + 1, right, x);

    }

    // Element is not present in array

    return -1;

}

// Function to perform binary search non-recursively

int binary\_search\_nonrec(int arr[], int n, int x) {

    int left = 0, right = n - 1;

    while (left <= right) {

        int mid = left + (right - left) / 2;

        // If the element is present at the middle itself

        if (arr[mid] == x)

            return mid;

        // If element is smaller than mid, then it can only be present in left subarray

        if (arr[mid] > x)

            right = mid - 1;

        // Else the element can only be present in right subarray

        else

            left = mid + 1;

    }

    // Element is not present in array

    return -1;

}

// Function to print an array

void print\_array(int arr[], int n) {

    for (int i = 0; i < n; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

int main() {

    int arr[] = {2, 3, 4, 10, 40};

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

    int x = 10;

    printf("Original array:\n");

    print\_array(arr, n);

    // Perform binary search recursively

    int result\_rec = binary\_search\_rec(arr, 0, n - 1, x);

    if (result\_rec == -1)

        printf("%d not found recursively in the array\n", x);

    else

        printf("%d found recursively at index %d\n", x, result\_rec);

    // Perform binary search non-recursively

    int result\_nonrec = binary\_search\_nonrec(arr, n, x);

    if (result\_nonrec == -1)

        printf("%d not found non-recursively in the array\n", x);

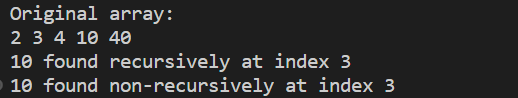
    else

        printf("%d found non-recursively at index %d\n", x, result\_nonrec);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 11**

Q: Write a program to find solution for knapsack problem using greedy method.

**CODE:**

#include<stdio.h>

int main()

{

     float weight[50],profit[50],ratio[50],Totalvalue,temp,capacity,amount;

     int n,i,j;

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

     scanf("%d",&n);

    for (i = 0; i < n; i++)

    {

        printf("Enter Weight and profit for item[%d] :\n",i);

        scanf("%f %f", &weight[i], &profit[i]);

    }

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

    scanf("%f",&capacity);

    for(i=0;i<n;i++)

        ratio[i]=profit[i]/weight[i];

    for (i = 0; i < n; i++)

      for (j = i + 1; j < n; j++)

         if (ratio[i] < ratio[j])

        {

            temp = ratio[j];

            ratio[j] = ratio[i];

            ratio[i] = temp;

            temp = weight[j];

            weight[j] = weight[i];

            weight[i] = temp;

            temp = profit[j];

            profit[j] = profit[i];

            profit[i] = temp;

         }

     printf("Knapsack problems using Greedy Algorithm:\n");

     for (i = 0; i < n; i++)

     {

      if (capacity>0&&weight[i] <= capacity){

          Totalvalue = Totalvalue + profit[i];

          capacity = capacity - weight[i];

      }

       else{

           break;

       }

     }

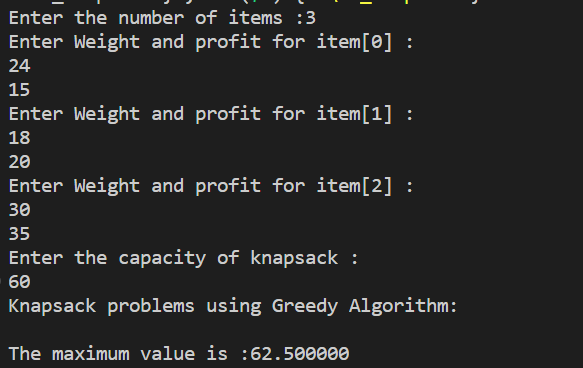
    Totalvalue = Totalvalue + (ratio[i]\*capacity);

    printf("\nThe maximum value is :%f\n",Totalvalue);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 12**

Q: Write a program to find minimum cost spanning tree using Prim’s Algorithm

**CODE:**

#include <stdio.h>

// Number of vertices in the graph

#define V 5

// Utility function to find the vertex with minimum distance value

int min\_distance(int dist[], int visited[]) {

    int min = \_\_INT\_MAX\_\_, min\_index;

    for (int i = 0; i < V; i++) {

        if (visited[i] == 0 && dist[i] < min) {

            min = dist[i];

            min\_index = i;

        }

    }

    return min\_index;

}

// Function to print the minimum spanning tree

void print\_mst(int parent[], int graph[V][V]) {

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

    for (int i = 1; i < V; i++)

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

}

// Function to find the minimum spanning tree using Prim's algorithm

void prim\_mst(int graph[V][V]) {

    int parent[V];

    int dist[V];

    int visited[V];

    for (int i = 0; i < V; i++) {

        dist[i] = \_\_INT\_MAX\_\_;

        visited[i] = 0;

    }

    dist[0] = 0;

    parent[0] = -1;

    for (int i = 0; i < V - 1; i++) {

        int u = min\_distance(dist, visited);

        visited[u] = 1;

        for (int v = 0; v < V; v++) {

            if (graph[u][v] && visited[v] == 0 && graph[u][v] < dist[v]) {

                parent[v] = u;

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

            }

        }

    }

    print\_mst(parent, graph);

}

int main() {

    // Graph represented as adjacency matrix

    int graph[V][V] = {{0, 2, 0, 6, 0},

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

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

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

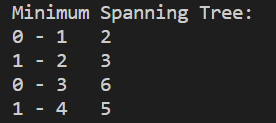
                       {0, 5, 7, 9, 0}};

    prim\_mst(graph);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 13**

Q: Write a program to find minimum cost spanning tree using Kruskal’s Algorithm.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

// Structure to represent an edge in the graph

struct Edge {

    int src, dest, weight;

};

// Structure to represent a subset for union-find

struct Subset {

    int parent;

    int rank;

};

// Comparator function to use in sorting the edges by weight

int comparator(const void\* p1, const void\* p2) {

    struct Edge\* edge1 = (struct Edge\*)p1;

    struct Edge\* edge2 = (struct Edge\*)p2;

    return edge1->weight - edge2->weight;

}

// Function to find the parent of a subset (uses path compression)

int find(struct Subset subsets[], int i) {

    if (subsets[i].parent != i)

        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;

}

// Function to perform union of two subsets (uses union by rank)

void unionSet(struct Subset subsets[], int x, int y) {

    int xroot = find(subsets, x);

    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)

        subsets[xroot].parent = yroot;

    else if (subsets[xroot].rank > subsets[yroot].rank)

        subsets[yroot].parent = xroot;

    else {

        subsets[yroot].parent = xroot;

        subsets[xroot].rank++;

    }

}

// Function to find the minimum cost spanning tree using Kruskal's Algorithm

void kruskalMST(int V, int E, struct Edge edges[]) {

    struct Edge result[V]; // Stores the resultant MST

    int e = 0; // Index variable for result[]

    int i = 0; // Index variable for sorted edges array

    // Sort all the edges in non-decreasing order of their weight

    qsort(edges, E, sizeof(struct Edge), comparator);

    // Allocate memory for creating V subsets

    struct Subset\* subsets = (struct Subset\*)malloc(V \* sizeof(struct Subset));

    // Create V subsets with single elements

    for (int v = 0; v < V; v++) {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

    // Keep adding edges to the MST until V-1 edges are added

    while (e < V - 1 && i < E) {

        struct Edge next\_edge = edges[i++];

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

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

        // If including this edge doesn't form a cycle, add it to the MST

        if (x != y) {

            result[e++] = next\_edge;

            unionSet(subsets, x, y);

        }

    }

    // Print the edges of the MST

    printf("Minimum Cost Spanning Tree:\n");

    int minCost = 0;

    for (i = 0; i < e; i++) {

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

        minCost += result[i].weight;

    }

    printf("Minimum Cost: %d\n", minCost);

}

// Driver code

int main() {

    int V = 4; // Number of vertices

    int E = 5; // Number of edges

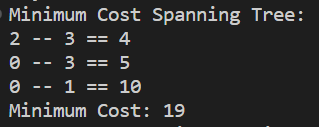
    struct Edge edges[] = { {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4} };

    kruskalMST(V, E, edges);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 14**

Q: Write a program to perform Single source shortest path problem for a given graph.

**CODE:**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum  distance value, from the set of vertices not yet included in shortest path tree

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

    // Initialize min value

    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;}

// A utility function to print the constructed distance array

void printSolution(int dist[]){

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

    for (int i = 0; i < V; i++)

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

}

// Function that implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation

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

    int dist[V]; // The output array.  dist[i] will hold the shortest distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest path tree or shortest distance from src to i is

    // finalized Initialize all distances as INFINITE and stpSet[] as false

    for (int i = 0; i < V; i++)

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

    // Distance of source vertex from itself is always 0

    dist[src] = 0;

    // Find shortest path for all vertices

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

        // Pick the minimum distance vertex from the set of

        // vertices not yet processed. u is always equal to

        // src in the first iteration.

        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed

        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the

        // picked vertex.

        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from u to v, and total weight of path from src to  v through u is smaller than current value of dist[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];    }

    // print the constructed distance array

    printSolution(dist);}

// driver's code

int main(){

    /\* Let us create the example graph discussed above \*/

    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 } };

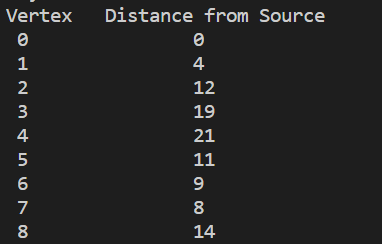
    // Function call

    dijkstra(graph, 0);

    return 0;

}

**OUTPUT:**

****

**PRACTICAL 15**

Q: Write a program to find solution for job sequencing with deadlines problem.

**CODE:**

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

// A structure to represent a job

typedef struct Job {

    char id; // Job Id

    int dead; // Deadline of job

    int profit; // Profit if job is over before or on

                // deadline

} Job;

// This function is used for sorting all jobs according to profit

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

    Job\* temp1 = (Job\*)a;

    Job\* temp2 = (Job\*)b;

    return (temp2->profit - temp1->profit);

}

// Find minimum between two numbers.

int min(int num1, int num2){

    return (num1 > num2) ? num2 : num1;

}

// Returns maximum profit from jobs

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

    // Sort all jobs according to decreasing order of profit

    qsort(arr, n, sizeof(Job), compare);

    int result[n]; // To store result (Sequence of jobs)

    bool slot[n]; // To keep track of free time slots

    // Initialize all slots to be free

    for (int i = 0; i < n; i++)

        slot[i] = false;

    // Iterate through all given jobs

    for (int i = 0; i < n; i++) {

        // Find a free slot for this job (Note that we start

        // from the last possible slot)

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

            // Free slot found

            if (slot[j] == false) {

                result[j] = i; // Add this job to result

                slot[j] = true; // Make this slot occupied

                break;}   }   }

    // Print the result

    for (int i = 0; i < n; i++)

        if (slot[i])

            printf("%c ", arr[result[i]].id);}

// Driver's code

int main(){

    Job arr[] = { { 'a', 2, 100 },

                  { 'b', 1, 19 },

                  { 'c', 2, 27 },

                  { 'd', 1, 25 },

                  { 'e', 3, 15 } };

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

    printf( "Following is maximum profit sequence of jobs \n");

    // Function call

    printJobScheduling(arr, n);

    return 0;

}

**OUTPUT:**

****

Project

**Aim:** Write a program to perform Travelling Salesman Problem

The Traveling Salesman Problem (TSP) is a classic optimization problem in computer science and operations research. It is a combinatorial problem that seeks to find the shortest possible route that a salesman can take to visit a set of cities and return to the starting city, visiting each city exactly once.

The problem can be stated as follows: Given a list of cities and the distances between each pair of cities, the objective is to find the shortest possible route that visits each city exactly once and returns to the starting city.

Here are the key characteristics of the TSP:

1. **Complete Graph:** The TSP is typically formulated as a complete graph, where each city represents a vertex, and the distances between cities represent the weights of the edges connecting the vertices.
2. **Symmetric Distances:** In the symmetric TSP, the distance from city A to city B is the same as the distance from city B to city A. In the asymmetric TSP, this condition does not hold, and the distances between cities may vary in both directions.
3. **Optimization Criteria:** The goal of the TSP is to minimize the total distance traveled or the total cost of the route. Other variants may aim to minimize time, maximize profit, or satisfy additional constraints.
4. **NP-Hard Problem:** The TSP is classified as an NP-hard problem, meaning that there is no known polynomial-time algorithm that can solve all instances of the problem optimally. As the number of cities increases, the problem becomes increasingly computationally intensive.

Given the complexity of the TSP, several algorithms and techniques have been developed to solve or approximate solutions for the problem. Some common approaches include:

* **Brute Force:** Enumerating all possible permutations of the cities and calculating the total distance for each permutation to find the optimal solution. This approach is only feasible for small problem sizes due to the exponential growth in computation time.
* **Dynamic Programming (DP):** Using a bottom-up approach to solve subproblems and build up to the optimal solution. The DP approach is efficient for solving small to moderate-sized TSP instances.
* **Approximation Algorithms:** Heuristic algorithms that provide

near-optimal solutions in polynomial time. Examples include the Nearest Neighbor algorithm, Christofides algorithm, and Lin-Kernighan algorithm.

* **Integer Linear Programming (ILP)**: Formulating the TSP as an optimization problem and using ILP techniques to find the optimal solution. This approach is effective for small to medium-sized instances but becomes computationally expensive for larger problems.
* **Metaheuristic Algorithms:** Evolutionary algorithms, simulated annealing, ant colony optimization, and genetic algorithms are examples of metaheuristic algorithms that provide good approximate solutions for large TSP instances.

The TSP has practical applications in various domains, including logistics, transportation planning, circuit board manufacturing, DNA sequencing, and network routing. Solving the TSP efficiently has been an active area of research, and numerous algorithms and techniques continue to be developed to tackle this challenging problem.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX\_CITIES 10

int distanceMatrix[MAX\_CITIES][MAX\_CITIES];

int numCities;

int bestPath[MAX\_CITIES];

int bestCost = INT\_MAX;

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

{

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

void permute(int \*path, int start, int end)

{

    if (start == end)

    {

        int currentCost = 0;

        for (int i = 0; i < numCities - 1; i++)

        {

            currentCost += distanceMatrix[path[i]][path[i + 1]];

        }

        currentCost += distanceMatrix[path[numCities - 1]][path[0]]; // return to starting city

        if (currentCost < bestCost)

        {

            bestCost = currentCost;

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

            {

                bestPath[i] = path[i];

            }

        }

    }

    else

    {

        for (int i = start; i <= end; i++)

        {

            swap(&path[start], &path[i]);

            permute(path, start + 1, end);

            swap(&path[start], &path[i]); // backtrack

        }

    }

}

void solveTSP()

{

    int path[MAX\_CITIES];

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

    {

        path[i] = i;

    }

    permute(path, 0, numCities - 1);

}

int main()

{

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

    scanf("%d", &numCities);

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

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

    {

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

        {

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

        }

    }

    solveTSP();

    printf("Best Path: ");

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

    {

        printf("%d ", bestPath[i]);

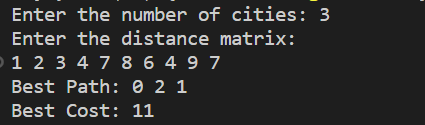
    }

    printf("\nBest Cost: %d\n", bestCost);

    return 0;

}

**OUTPUT:**

****