# Experiment No: 1

**Experiment Name**: Implementation and Analysis of Linear Search Using recursive function.

**Objective:** Implement and analyse the recursive implementation of linear search to understand its recursive nature and assess its time and space complexity.

**Program:**

#include <stdio.h>

int recursiveLinearSearch(int arr[], int target, int index, int size) {

if (index == size) {

return -1;

}

if (arr[index] == target) {

return index;

}

return recursiveLinearSearch(arr, target, index + 1, size);

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

int targetElement;

printf("Enter the target element to search: ");

scanf("%d", &targetElement);

int result = recursiveLinearSearch(myArray, targetElement, 0, arraySize);

if (result != -1) {

printf("Element %d found at index %d.\n", targetElement, result);

} else {

printf("Element %d not found in the array.\n", targetElement);

}

return 0;

}

# Experiment No: 1

**Experiment Name**: Implementation and Analysis of Binary Search Using recursive function

**Objective:** Implement and analyse the recursive implementation of Binary search to understand its recursive nature and assess its time and space complexity.

**Program:**

#include <stdio.h>

int recursiveBinarySearch(int arr[], int target, int low, int high) {

if (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == target) {

return mid;

}

if (arr[mid] > target) {

return recursiveBinarySearch(arr, target, low, mid - 1);

} else {

return recursiveBinarySearch(arr, target, mid + 1, high);

}

}

return -1;

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d sorted elements for the array:\n", arraySize);

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

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

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

}

int targetElement;

printf("Enter the target element to search: ");

scanf("%d", &targetElement);

int result = recursiveBinarySearch(myArray, targetElement, 0, arraySize - 1);

if (result != -1) {

printf("Element %d found at index %d.\n", targetElement, result);

} else {

printf("Element %d not found in the array.\n", targetElement);

}

return 0;

}

# Experiment No: 2

**Experiment Name**: Implementation and Analysis of Insertion Sort .

**Objective:** To implement the Insertion Sort algorithm and analyze its efficiency in sorting data, evaluating its time complexity and practical performance.

**Program:**

#include <stdio.h>

#include <stdlib.h>

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

int i, key, j;

for (i = 1; i < size; 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 arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

insertionSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 2

**Experiment Name**: Implementation and Analysis of Bubble Sort.

**Objective:** Objective: To implement and analyse the Bubble Sort algorithm's efficiency and performance in sorting a given dataset.

**Program:**

#include <stdio.h>

#include <stdlib.h>

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

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

for (int j = 0; j < size - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

bubbleSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 2

**Experiment Name**: Implementation and Analysis of Selection Sort.

**Objective:** Objective: To implement and analyse the Selection Sort algorithm's efficiency and performance in sorting a given dataset.

**Program:**

#include <stdio.h>

#include <stdlib.h>

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

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

int minIndex = i;

for (int j = i + 1; j < size; j++) {

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

minIndex = j;

}

}

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

selectionSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 3

**Experiment Name**: Implementation and Analysis of Merge Sort.

**Objective:** To implement and analyse the Merge Sort algorithm for efficient sorting of data, assessing its time and space complexity.

**Program:**

#include <stdio.h>

#include <stdlib.h>

void merge(int arr[], int left, int middle, int right) {

int i, j, k;

int n1 = middle - left + 1;

int n2 = right - middle;

int L[n1], R[n2];

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

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

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

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

i = 0;

j = 0;

k = left;

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 left, int right) {

if (left < right) {

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

mergeSort(arr, left, middle);

mergeSort(arr, middle + 1, right);

merge(arr, left, middle, right);

}

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

mergeSort(myArray, 0, arraySize - 1);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 3

**Experiment Name**: Implementation and Analysis of Quick Sort.

**Objective:** To implement and analyse the Quick Sort algorithm for efficient sorting of data, assessing its time and space complexity.

**Program:**

#include <stdio.h>

#include <stdlib.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 arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

quickSort(myArray, 0, arraySize - 1);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 4

**Experiment Name**: Implementation and Analysis of Heap Sort.

**Objective:** To implement and analyse the Heap Sort algorithm for efficient sorting of data, assessing its time and space complexity.

**Program:**

#include <stdio.h>

#include <stdlib.h>

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

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

largest = left;

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

largest = right;

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapify(arr, size, largest);

}

}

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

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

heapify(arr, size, i);

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

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

heapSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 4

**Experiment Name**: Implementation and Analysis of Counting Sort.

**Objective:** To implement and analyse the Counting Sort algorithm for efficient sorting of data, assessing its time and space complexity.

**Program:**

#include <stdio.h>

#include <stdlib.h>

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

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

int\* count = (int\*)malloc((max + 1) \* sizeof(int));

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

count[i] = 0;

}

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

count[arr[i]]++;

}

int k = 0;

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

while (count[i] > 0) {

arr[k++] = i;

count[i]--;

}

}

free(count);

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

countingSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 5

**Experiment Name**: Implementation and Analysis of Radix Sort.

**Objective:** To implement and analyse the Radix Sort algorithm for efficient sorting of data, assessing its time and space complexity.

**Program:**

#include <stdio.h>

#include <stdlib.h>

int getMax(int arr[], int size) {

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

void countingSort(int arr[], int size, int place) {

const int max = 10;

int output[size];

int count[max];

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

count[i] = 0;

}

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

count[(arr[i] / place) % 10]++;

}

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

count[i] += count[i - 1];

}

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

output[count[(arr[i] / place) % 10] - 1] = arr[i];

count[(arr[i] / place) % 10]--;

}

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

arr[i] = output[i];

}

}

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

int max = getMax(arr, size);

for (int place = 1; max / place > 0; place \*= 10) {

countingSort(arr, size, place);

}

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

radixSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 5

**Experiment Name**: Implementation and Analysis of Shell Sort.

**Objective:** To implement and analyse the Shell Sort algorithm for efficient sorting of data, assessing its time and space complexity.

**Program:**

#include <stdio.h>

#include <stdlib.h>

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

for (int gap = size / 2; gap > 0; gap /= 2) {

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

int temp = arr[i];

int j;

for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {

arr[j] = arr[j - gap];

}

arr[j] = temp;

}

}

}

int main() {

int arraySize;

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

scanf("%d", &arraySize);

int myArray[arraySize];

printf("Enter %d elements for the array:\n", arraySize);

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

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

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

}

shellSort(myArray, arraySize);

printf("Sorted array: ");

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

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

}

printf("\n");

return 0;

}

# Experiment No: 6

**Experiment Name**: Implementation of Activity Selection Problem

**Objective:** To successfully implement the Activity Selection Problem algorithm and gain practical insights into its application in scheduling activities.

**Program:**

#include <stdio.h>

void greedy\_activity\_selector(int start\_time[], int finish\_time[], int n) {

    int arr[n];

    int count=1;

    int j=0;

    int i;

    arr[0] = 1;

    printf("Selected Activity : \n");

    for (i = 1; i < n; i++) {

        if (start\_time[i] >= finish\_time[j]) {

            arr[count] = i+1;

            count++;

            j = i;

        }

    }

    for(int i=0;i<count;i++){

        printf("-------");

    }

    printf("\n");

    for(int i=0 ; i<count ; i++){

        printf("  P%d  |", arr[i]);

    }

    printf("\n");

    for(int i=0;i<count;i++){

        printf("-------");

    }

    printf("\n");

}

int main() {

    int start\_times[] = {1, 3, 0, 5, 8, 5,11,15};

    int finish\_times[] = {2, 4, 6, 7, 9, 9,14,18};

    int n = sizeof(start\_times) / sizeof(start\_times[0]);

    greedy\_activity\_selector(start\_times, finish\_times, n);

    return 0;

}

# Experiment No: 6

**Experiment Name**: Implementation of Knapsack Problem using Greedy Solution

**Objective:** To implement the Greedy Solution for the Knapsack Problem and explore its effectiveness in achieving optimal solutions for resource allocation, without the need for exhaustive search techniques.

**Program:**

#include <stdio.h>

#include <stdlib.h>

struct Item {

    int value;

    int weight;

};

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

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

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

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

}

double greedy\_knapsack(struct Item items[], int n, int capacity) {

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

    double max\_value = 0.0;

    int current\_weight = 0;

    printf("Selecting Items :\n");

    for (int i = 0; i < n; i++) {

        if (current\_weight + items[i].weight <= capacity) {

            current\_weight += items[i].weight;

            max\_value += items[i].value;

            printf("Added item { value: %4d , weight : %4d } to bag\n",items[i].value,items[i].weight);

        } else {

            double remaining\_capacity = capacity - current\_weight;

            max\_value += (remaining\_capacity / items[i].weight) \* items[i].value;

            break;

        }

    }

    return max\_value;

}

int main() {

    struct Item items[] = {{60, 10}, {100, 20}, {120, 30},{50,10},{200,10}};

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

    int capacity = 50;

    double max\_value = greedy\_knapsack(items, n, capacity);

    printf("Maximum value: %.2lf\n", max\_value);

    return 0;

}

# Experiment No: 7

**Experiment Name**: Implement and Analysis of the 0/1 Knapsack problem using Dynamic Programming method

**Objective:** Implement and analyse 0/1 Knapsack with Dynamic Programming, evaluating efficiency in optimal resource allocation under constraints.

**Program:**

#include <stdio.h>

struct Item

{

    int value;

    int weight;

};

int max(int a, int b)

{

    return (a > b) ? a : b;

}

int knapsackDP(int capacity, const struct Item items[], int n)

{

    int dp[n + 1][capacity + 1];

    int isSelected[n + 1][capacity + 1];

    for (int i = 0; i <= n; i++)

    {

        for (int w = 0; w <= capacity; w++)

        {

            if (i == 0 || w == 0)

            {

                dp[i][w] = 0;

                isSelected[i][w] = 0;

            }

            else if (items[i - 1].weight <= w)

            {

                int include = items[i - 1].value + dp[i - 1][w - items[i - 1].weight];

                int exclude = dp[i - 1][w];

                if (include > exclude)

                {

                    dp[i][w] = include;

                    isSelected[i][w] = 1;

                }

                else

                {

                    dp[i][w] = exclude;

                    isSelected[i][w] = 0;

                }

            }

            else

            {

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

                isSelected[i][w] = 0;

            }

        }

    }

    printf("Selected items: \n");

    int i = n, w = capacity;

    while (i > 0 && w > 0)

    {

        if (isSelected[i][w])

        {

            printf("Item { value: %d , weight: %d} \n", items[i - 1].value, items[i - 1].weight);

            w -= items[i - 1].weight;

        }

        i--;

    }

    printf("\n");

    return dp[n][capacity];

}

int main()

{

    struct Item items[] = {{60, 10}, {100, 20}, {120, 30}, {50, 10}, {200, 10}};

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

    int capacity = 50;

    int max\_value = knapsackDP(capacity, items, n);

    printf("Maximum value: %d\n", max\_value);

    return 0;

}

# Experiment No: 7

**Experiment Name**: Implementation and Analysis of LCS

**Objective:** Implement and analyse LCS algorithm, examining its effectiveness in finding the longest common subsequence between two sequences.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int lcsLength(const char \*X, const char \*Y) {

    int m = strlen(X);

    int n = strlen(Y);

    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {

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

            if (i == 0 || j == 0)

                dp[i][j] = 0;

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

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

            else

                dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ? dp[i - 1][j] : dp[i][j - 1];

        }

    }

    return dp[m][n];

}

void printLCS(const char \*X, const char \*Y, int m, int n) {

    int dp[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {

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

            if (i == 0 || j == 0)

                dp[i][j] = 0;

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

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

            else

                dp[i][j] = (dp[i - 1][j] > dp[i][j - 1]) ? dp[i - 1][j] : dp[i][j - 1];

        }

    }

    int index = dp[m][n];

    char lcs[index + 1];

    lcs[index] = '\0';

    int i = m, j = n;

    while (i > 0 && j > 0) {

        if (X[i - 1] == Y[j - 1]) {

            lcs[index - 1] = X[i - 1];

            i--;

            j--;

            index--;

        } else if (dp[i - 1][j] > dp[i][j - 1]) {

            i--;

        } else {

            j--;

        }

    }

    printf("Longest Common Subsequence: %s\n", lcs);

}

int main() {

    const char \*X = "ABCBDAB";

    const char \*Y = "BDCAB";

    int m = strlen(X);

    int n = strlen(Y);

    int length = lcsLength(X, Y);

    printf("Length of LCS: %d\n", length);

    printLCS(X, Y, m, n);

    return 0;

}

# Experiment No: 8

**Experiment Name**: Implementation of Kruskal’s algorithm to find MST.

**Objective:** Implement Kruskal's algorithm for finding the Minimum Spanning Tree (MST), evaluating its effectiveness in connecting nodes with minimal total edge weights.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define V 10

#define E 20

struct Edge {

    int src, dest, weight;

};

struct Subset {

    int parent, rank;

};

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

    return ((struct Edge \*)a)->weight - ((struct Edge \*)b)->weight;

}

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

    if (subsets[i].parent != i)

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

    return subsets[i].parent;

}

void Union(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++;

    }

}

void kruskalMST(struct Edge edges[], int n, int e) {

    struct Edge result[n];

    int i = 0, j = 0;

    qsort(edges, e, sizeof(edges[0]), compareEdges);

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

    for (int v = 0; v < n; v++) {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

    while (i < n - 1 && j < e) {

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

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

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

        if (x != y) {

            result[i++] = next\_edge;

            Union(subsets, x, y);

        }

    }

    printf("Edge   Weight\n");

    for (int i = 0; i < n - 1; i++)

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

    free(subsets);

}

int main() {

    int n, e;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &n);

    printf("Enter the number of edges in the graph: ");

    scanf("%d", &e);

    struct Edge edges[E];

    printf("Enter the edges (source destination weight) of the graph:\n");

    for (int i = 0; i < e; i++)

        scanf("%d %d %d", &edges[i].src, &edges[i].dest, &edges[i].weight);

    kruskalMST(edges, n, e);

    return 0;

}

# Experiment No: 8

**Experiment Name**: Implementation of Prim’s algorithm to find MST.

**Objective:** Implement Prim's algorithm for finding the Minimum Spanning Tree (MST), assessing its ability to connect nodes with minimal total edge weights.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define V 10

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

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < n; v++) {

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

            min = key[v];

            min\_index = v;

        }

    }

    return min\_index;

}

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

    printf("Edge   Weight\n");

    for (int i = 1; i < n; i++)

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

}

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

    int parent[V]; // Array to store the constructed MST

    int key[V];    // Key values used to pick the minimum weight edge

    bool mstSet[V]; // To represent set of vertices included in MST

    for (int i = 0; i < n; i++) {

        key[i] = INT\_MAX;

        mstSet[i] = false;

    }

    key[0] = 0;

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

        int u = minKey(key, mstSet, n);

        mstSet[u] = true;

        for (int v = 0; v < n; v++) {

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

                parent[v] = u;

                key[v] = graph[u][v];

            }

        }

    }

    printMST(parent, graph, n);

}

int main() {

    int n;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &n);

    int graph[V][V];

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

    for (int i = 0; i < n; i++)

        for (int j = 0; j < n; j++)

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

    primMST(graph, n);

    return 0;

}

# Experiment No: 9

**Experiment Name**: Implementation of Warshal’s Algorithm for All Pair Shortest Path

**Objective:** Implement Warshall's Algorithm for All-Pair Shortest Path, analyzing its efficiency in finding the shortest paths between all pairs of vertices in a graph.

**Program:**

#include <stdio.h>

#define V 4

#define INF 99999

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

void floydWarshall(int dist[][V])

{

    int i, j, k;

    for (k = 0; k < V; k++) {

        for (i = 0; i < V; i++) {

            for (j = 0; j < V; j++) {

                if (dist[i][k] + dist[k][j] < dist[i][j])

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

            }

        }

    }

    printSolution(dist);

}

void printSolution(int dist[][V])

{

    printf(

        "The following matrix shows the shortest distances"

        " between every pair of vertices \n");

    for (int i = 0; i < V; i++) {

        for (int j = 0; j < V; j++) {

            if (dist[i][j] == INF)

                printf("%7s", "INF");

            else

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

        }

        printf("\n");

    }

}

int main()

{

    int graph[V][V] = { { 0, 5, INF, 10 },

                        { INF, 0, 3, INF },

                        { INF, INF, 0, 1 },

                        { INF, INF, INF, 0 } };

    floydWarshall(graph);

    return 0;

}

# Experiment No: 9

**Experiment Name**: Implementation of Dijkstra Algorithm for Single Source Shortest Path.

**Objective:** Implement Dijkstra's algorithm for Single Source Shortest Path, evaluating its effectiveness in finding the shortest paths from a source node to all other nodes in a graph.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define V 10

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

    int min = INT\_MAX, min\_index;

    for (int v = 0; v < n; v++) {

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

            min = dist[v];

            min\_index = v;

        }

    }

    return min\_index;

}

void printSolution(int dist[], int n, int src) {

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

    for (int i = 0; i < n; i++)

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

}

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

    int dist[V];

    bool sptSet[V];

    for (int i = 0; i < n; i++) {

        dist[i] = INT\_MAX;

        sptSet[i] = false;

    }

    dist[src] = 0;

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

        int u = minDistance(dist, sptSet, n);

        sptSet[u] = true;

        for (int v = 0; v < n; 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, n, src);

}

int main() {

    int n;

    printf("Enter the number of vertices in the graph: ");

    scanf("%d", &n);

    int graph[V][V];

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

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

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

        }

    }

    int src;

    printf("Enter the source vertex: ");

    scanf("%d", &src);

    dijkstra(graph, src, n);

    return 0;

}

# Experiment No: 10

**Experiment Name**: Implementation of N Queen Problem using Backtracking

**Objective:** Implement the N Queen Problem using Backtracking, exploring its ability to find all safe placements for N queens on a chessboard without mutual attacks.

**Program:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

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

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++)

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

        printf("\n");

    }

    printf("\n");

}

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

    int i, j;

    for (i = 0; i < col; i++)

        if (board[row][i])

            return false;

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

        if (board[i][j])

            return false;

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

        if (board[i][j])

            return false;

    return true;

}

bool solveNQueensUtil(int \*\*board, int col, int N) {

    if (col >= N) {

        printSolution(board, N);

        return true;

    }

    bool res = false;

    for (int i = 0; i < N; i++) {

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

            board[i][col] = 1;

            res = solveNQueensUtil(board, col + 1, N) || res;

            board[i][col] = 0;

        }

    }

    return res;

}

void solveNQueens(int N) {

    int \*\*board = (int \*\*)malloc(N \* sizeof(int \*));

    for (int i = 0; i < N; i++) {

        board[i] = (int \*)malloc(N \* sizeof(int));

        for (int j = 0; j < N; j++)

            board[i][j] = 0;

    }

    if (!solveNQueensUtil(board, 0, N)) {

        printf("Solution does not exist");

    }

    for (int i = 0; i < N; i++) {

        free(board[i]);

    }

    free(board);

}

int main() {

    int N;

    printf("Enter the value of N for N-Queens: ");

    scanf("%d", &N);

    solveNQueens(N);

    return 0;

}

# Experiment No: 10

**Experiment Name**: Implementation of Sum of Subset problem using Back Tracking

**Objective:** Implement the Sum of Subset problem using Backtracking, examining its capability to find subsets with a specific sum within a given set of numbers.

**Program:**

#include <stdio.h>

#include <stdbool.h>

bool isSubsetSum(int set[], int n, int sum) {

    if (sum == 0)

        return true;

    if (n == 0)

        return false;

    if (set[n - 1] > sum)

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

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

}

int main() {

    int n, sum;

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

    scanf("%d", &n);

    int set[n];

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

    for (int i = 0; i < n; i++) {

        scanf("%d", &set[i]);

    }

    printf("Enter the target sum: ");

    scanf("%d", &sum);

    if (isSubsetSum(set, n, sum)) {

        printf("Found a subset with the given sum\n");

    } else {

        printf("No subset with the given sum\n");

    }

    return 0;

}