RUSTAMJI INSTITUTE OF TECHNOLOGY

BSF ACADEMY, TEKANPUR

Practical File for

CS402 (Analysis Design of Algorithm)

(Session: Jan-Jun 2025)

Submitted by

Shani Bharadwaj (0902CS231105)
B.Tech. 4th Semester (2023-2027 batch)

Department of Computer Science & Engineering

Subject Teacher	Submitted to
Dr. Jagdish Makhijani	Mr. Yashwant Pathak

TABLE OF CONTENTS

Section-D (Dynamic Programming)

S. No.	Practical Description	Page Nos.	COs
1	Write a program for 0/1 Knapsack Problem using dynamic programming.	39-40	CO-3
2	Write a program for Multistage Graph Problem using dynamic programming.	41	CO-3
3	Write a program for Reliability Design Problem (simplified version).	42	CO-3
4	Write a program for Floyd-Warshall All-Pairs Shortest Path Algorithm.	43-44	CO-3

Section-E (Backtracking)

S. No.	Practical Description	Page Nos.	COs
1	Write a program for the N-Queens Problem.	45-46	CO-4
2	Write a program for the Hamiltonian Cycle Problem.	47-48	CO-4
3	Write a program for the Graph Coloring Problem using backtracking.	49-50	CO-4
4	Write a program to generate permutations of a string using backtracking.	51-52	CO-4

Section-F (Branch and Bound)

S. No.	Practical Description	Page Nos.	COs
1	Write a program for Traveling Salesman Problem (TSP) using Branch and Bound.	53-54	CO-4
2	Write a program to solve the 0/1 Knapsack Problem using Branch and Bound.	55-56	CO-4

Section-G (Trees and Graphs)

S. No.	Practical Description	Page Nos.	COs
1	Write a program for Inorder, Preorder, and Postorder Traversals of a Binary Tree.	57-59	CO-5
2	Write a program for Depth-First Search (DFS) traversal OR Breadth-First Search (BFS) traversal.	60-61	CO-5

Experiment No.: 1

Program Description: Write a program for Iterative Binary Search.

Solution:

```
  #include <stdio.h>

 // Function for iterative binary search int
binarySearch(int arr[], int n, int key) {
   int left = 0, right = n - 1;

   while (left <= right) {
      int mid = (left + right) / 2;

      if (arr[mid] == key)           return mid;
// Found        else if (key < arr[mid])
right = mid - 1; // Search in left half
      else
         left = mid + 1;  // Search in right half
   }

   return -1; // Not found
}

int main() {
   int n, key, i;

   // Input size and elements
   printf("Enter number of elements (sorted): ");
scanf("%d", &n);

   int arr[n];
   printf("Enter %d sorted elements:\n", n);
for (i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }

   // Input key to search      printf("Enter
the number to search: ");     scanf("%d",
&key);

   // Perform binary search     int result
= binarySearch(arr, n, key); // Output
result
```

```
   if (result == -1)
       printf("Element not found in the array.\n");
else
       printf("Element found at index: %d\n", result);

   return 0;
}
```

Output

```
Enter number of elements (sorted): 6
Enter 6 sorted elements
1 3 5 7 9 11
Enter the number to search: 7
Element found at index: 3
```

Experiment No.: 2

Program Description: Write a program for Recursive Binary Search.

Solution:
```
#include <stdio.h>

// Recursive binary search function int
binarySearch(int arr[], int left, int right, int key) {
   if (left <= right) {        int
mid = (left + right) / 2;

       if (arr[mid] == key)            return
mid;  // Found the element
       else if (key < arr[mid])
           return binarySearch(arr, left, mid - 1, key); // Search left
else
           return binarySearch(arr, mid + 1, right, key); // Search right
   }

   return -1; // Element not found
}

int main() {
   int n, key, i;

   // Input array size and elements    printf("Enter
number of elements (sorted): ");      scanf("%d",
&n);

   int arr[n];
   printf("Enter %d sorted elements:\n", n);
for (i = 0; i < n; i++) {
       scanf("%d", &arr[i]);
   }

   // Input key to search      printf("Enter
the number to search: ");     scanf("%d",
&key);

   // Call recursive binary search     int result
= binarySearch(arr, 0, n - 1, key);

   // Output result
   if (result == -1)
```

```
        printf("Element not found in the array.\n");
else
        printf("Element found at index: %d\n", result);

    return 0;
}
```

Output:

```
Enter number of elements (sorted): 5 Enter
5 sorted elements:
2 4 6 8 10
Enter the number to search: 6
Element found at index: 2
```

Experiment No.: 3

Program Description: Write a program for Merge Sort.

Solution:

```
 #include <stdio.h>

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
int i, j, k;
   int n1 = mid - left + 1; // size of left subarray
   int n2 = right - mid;    // size of right subarray

   int L[n1], R[n2]; // Temporary arrays

   // Copy data to temp arrays
for (i = 0; i < n1; i++)        L[i]
= arr[left + i];     for (j = 0; j <
n2; j++)        R[j] = arr[mid + 1
+ j];

   // Merge the temp arrays back into arr[]
i = 0; // Initial index of first subarray     j =
0; // Initial index of second subarray
   k = left; // Initial index of merged array

   while (i < n1 && j < n2) {
if (L[i] <= R[j])
arr[k++] = L[i++];        else
        arr[k++] = R[j++];
   }

   // Copy remaining elements of L[]
   while (i < n1)
      arr[k++] = L[i++];

   // Copy remaining elements of R[]
   while (j < n2)
      arr[k++] = R[j++];
}

// Merge sort function
void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = (left + right) / 2;
 // Sort first and second halves
mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge sorted halves
        merge(arr, left, mid, right);
    }
}

int main() {
    int n, i;

    // Input
    printf("Enter number of elements: ");
scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Perform merge sort
    mergeSort(arr, 0, n - 1);

    // Output
    printf("Sorted array using Merge Sort:\n");
for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter number of elements: 6
Enter 6 elements:
5 2 8 1 3 7
Sorted array using Merge Sort:
1 2 3 5 7 8
```

Experiment No.: 4

Program Description: Write a program for Quick Sort.

Solution:

```
#include <stdio.h>

// Function to swap two numbers
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // pivot element
    int i = low - 1;       // index of smaller element

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

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

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Main function
```

```
int main() {
    int n;
// Input size
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Input elements
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Quick Sort call
    quickSort(arr, 0, n - 1);

    // Output sorted array
    printf("Sorted array using Quick Sort:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter number of elements: 5
Enter 5 elements:
10 3 8 2 6
Sorted array using Quick Sort:
2 3 6 8 10
```

Experiment No.: 5

Program Description :Write a program for Heap Sort using Max Heap.
Solution:

```
#include <stdio.h>

// Function to swap two numbers
void swap(int *a, int *b) {      int
temp = *a;     *a = *b;
   *b = temp;
}

// Max-Heapify function void maxHeapify(int
arr[], int n, int i) {     int largest = i;          //
Initialize largest as root     int left = 2 * i + 1;
// Left child index
   int right = 2 * i + 2;    // Right child index

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

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

   // If largest is not root
if (largest != i) {
      swap(&arr[i], &arr[largest]);
      maxHeapify(arr, n, largest); // Recursively heapify the affected subtree
   }
}

// Function to build a Max Heap void
buildMaxHeap(int arr[], int n) {
   // Start from the last non-leaf node and heapify each node
   for (int i = n / 2 - 1; i >= 0; i--) {
      maxHeapify(arr, n, i);
   }
}

// Heap Sort function void
heapSort(int arr[], int n) {
   buildMaxHeap(arr, n);

   // One by one extract elements from heap
```

```
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // Call maxHeapify on the reduced heap
        maxHeapify(arr, i, 0);
    }
}

int main() {
    int n, i;

    //     Input     size     and     elements
printf("Enter  number  of  elements:  ");
scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
for (i = 0; i < n; i++) {        scanf("%d",
&arr[i]);
    }

    // Perform Heap Sort
    heapSort(arr, n);

    // Output sorted array        printf("Sorted
array using Heap Sort:\n");     for (i = 0; i < n;
i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter number of elements: 6 Enter
6 elements:
20 5 15 22 9 30
Sorted array using Heap Sort:
5 9 15 20 22 30
```

Experiment No.: 6

Program Description :Write a program to build a Min Heap and perform delete operation.
Solution:

```
#include <stdio.h>



#define MAX 100



// Function to swap two numbers
void swap(int *a, int *b) {       int
temp = *a;     *a = *b;
   *b = temp;
}

// Heapify function for Min Heap void
minHeapify(int heap[], int n, int i) {
int smallest = i;     int left = 2 * i + 1;  //
left child index
   int right = 2 * i + 2; // right child index

   if (left < n && heap[left] < heap[smallest])
smallest = left;

   if (right < n && heap[right] < heap[smallest])
smallest = right;

   if (smallest != i) {
      swap(&heap[i], &heap[smallest]);
      minHeapify(heap, n, smallest);
   }
}

// Function to build Min Heap void
buildMinHeap(int heap[], int n) {
   for (int i = n / 2 - 1; i >= 0; i--) {
      minHeapify(heap, n, i);
   }
}
```

```
// Function to delete root (minimum element)
int deleteMin(int heap[], int *n) {
    if (*n == 0) {
        return -1;
    }

    int min = heap[0];              // Root element
    heap[0] = heap[*n - 1];         // Replace root with last element
    (*n)--;                         // Reduce heap size
    minHeapify(heap, *n, 0);        // Heapify root

    return min;
}

int main() {
    int heap[MAX], n, i;

    // Input
    printf("Enter number of elements: ");
scanf("%d", &n);

    printf("Enter %d elements:\n", n);
for (i = 0; i < n; i++) {
        scanf("%d", &heap[i]);
    }

    // Build Min Heap
    buildMinHeap(heap, n);

    printf("Min Heap built:\n");
for (i = 0; i < n; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");

    // Delete root
    int deleted = deleteMin(heap, &n);
    printf("Deleted root element: %d\n", deleted);

    // Display heap after deletion
    printf("Heap after deletion:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter number of elements: 6
Enter 6 elements:
10 20 15 30 40 5
Min Heap built:
5 20 10 30 40 15
Deleted root element: 5
Heap after deletion:
10 20 15 30 4
```

Experiment No.: 7

Program Description: Compare the execution time of different sorting algorithms on the same input set.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to copy array
void copyArray(int source[], int dest[], int n) {
   for (int i = 0; i < n; i++)
dest[i] = source[i];
}

// ---------- Bubble Sort ----------
void bubbleSort(int arr[], int n) {
for (int i = 0; i < n-1; i++)
   for (int j = 0; j < n-i-1; j++)
if (arr[j] > arr[j+1]) {
int temp = arr[j];
arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
}

// ---------- Insertion Sort ---------- void
insertionSort(int arr[], int n) {
   for (int I = 1; I < n; i++) {
int key = arr[i], j = I – 1;
while (j >= 0 && arr[j] > key)
         arr[j + 1] = arr[j--];
      arr[j + 1] = key;
   }
}

//  ----------  Merge  Sort  ----------  void
merge(int arr[], int l, int m, int r) {    int
I, j, k;
```

```
   int n1 = m – l + 1, 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)        arr[k++] =
(L[i] <= R[j]) ? L[i++] : R[j++];     while (I <
n1) arr[k++] = L[i++];
   while (j < n2) arr[k++] = R[j++];
}
void mergeSort(int arr[], int l, int r) {
   if (l < r) {        int m = (l +
r) / 2;        mergeSort(arr, l,
m);        mergeSort(arr,
m+1, r);
       merge(arr, l, m, r);
   }
}

// ---------- Quick Sort ---------- int
partition(int arr[], int low, int high) {
int pivot = arr[high], I = low – 1, temp;
for (int j = low; j < high; j++) {        if
(arr[j] < pivot) {
           i++;
           temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
       }
   }
   temp = arr[i+1]; arr[i+1] = arr[high]; arr[high] = temp;
   return i+1;
}
void ijkstra(int arr[], int low, int high) {     if
(low < high) {       int pi = partition(arr, low,
high);        ijkstra(arr, low, pi – 1);
       ijkstra(arr, pi + 1, high);
   }
}

// ---------- Main Function ---------- int
main() {
   int n, l;
   printf(“Enter number of elements: “);
scanf(“%d”, &n);

   int        original[n],        temp[n];
printf(“Enter  %d  elements:\n”,  n);
for (I = 0; I < n; i++) {
       scanf(“%d”, &original[i]);
   }
   clock_t start, end;
   double time_taken;

   // Bubble Sort
copyArray(original, temp, n);
   start = clock();
bubbleSort(temp, n);
   end = clock();
   time_taken = ((double)(end – start)) / CLOCKS_PER_SEC;
printf(“Bubble Sort Time: %.6f seconds\n”, time_taken);

   // Insertion Sort
copyArray(original, temp, n);     start
= clock();     insertionSort(temp, n);
   end = clock();
   time_taken = ((double)(end – start)) / CLOCKS_PER_SEC;
   printf(“Insertion Sort Time: %.6f seconds\n”, time_taken);

   // Merge Sort
copyArray(original, temp, n);
start = clock();
mergeSort(temp, 0, n – 1);
end = clock();
   time_taken = ((double)(end – start)) / CLOCKS_PER_SEC;
   printf(“Merge Sort Time: %.6f seconds\n”, time_taken);

   // Quick Sort
copyArray(original, temp, n);
start = clock();     ijkstra(temp,
0, n – 1);     end = clock();
   time_taken = ((double)(end – start)) / CLOCKS_PER_SEC;
   printf(“Quick Sort Time: %.6f seconds\n”, time_taken);

   return 0;
}
```

Experiment No.: 8

Output:

```
Enter number of elements: 6 Enter
6 elements:
9 3 7 1 5 2

Bubble Sort Time: 0.000010 seconds
Insertion Sort Time: 0.000008 seconds
Merge Sort Time: 0.000006 seconds
Quick Sort Time: 0.000005 seconds
```

Experiment No.: 8

Program Description: (Divide and Conquer algorithm) Write a program for Strassen’s Matrix Multiplication.

Solution:

```
#include <stdio.h>

int main() {
   int A[2][2], B[2][2], C[2][2];
   int M1, M2, M3, M4, M5, M6, M7;

   // Input matrix A
   printf(“Enter elements of 2x2 matrix A:\n”);
for (int I = 0; I < 2; i++)        for (int j = 0; j < 2;
j++)
         scanf(“%d”, &A[i][j]);

   // Input matrix B
   printf(“Enter elements of 2x2 matrix B:\n”);
for (int I = 0; I < 2; i++)        for (int j = 0; j < 2;
j++)
         scanf(“%d”, &B[i][j]);

   // Strassen’s 7 formula steps
   M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
   M2 = (A[1][0] + A[1][1]) * B[0][0];
   M3 = A[0][0] * (B[0][1] – B[1][1]);
   M4 = A[1][1] * (B[1][0] – B[0][0]);
   M5 = (A[0][0] + A[0][1]) * B[1][1];
   M6 = (A[1][0] – A[0][0]) * (B[0][0] + B[0][1]);
   M7 = (A[0][1] – A[1][1]) * (B[1][0] + B[1][1]);

   // Calculate result matrix C
   C[0][0] = M1 + M4 – M5 + M7;
   C[0][1] = M3 + M5;
   C[1][0] = M2 + M4;
   C[1][1] = M1 – M2 + M3 + M6;

   // Display result
   printf(“Result of Strassen’s Matrix Multiplication:\n”);
for (int I = 0; I < 2; i++) {        for (int j = 0; j < 2; j++)
printf(“%d “, C[i][j]);        printf(“\n”);
   }
```

```
    return 0;
}
```

Output:

```
Enter elements of 2x2 matrix A:
1 2
3 4
Enter elements of 2x2 matrix B:
5 6
7 8

Result of Strassen's Matrix Multiplication:
19 22
43 50
```

Experiment No.: 9

Program Description: Write a program to find the Maximum and Minimum using Divide and Conquer.

Solution:

```
#include <stdio.h>

// Structure to hold both max and min
struct Result {
int max;
   int min;
};

// Function to find max and min using divide and conquer
struct Result findMaxMin(int arr[], int low, int high) {
struct Result res, left, right;

   // If only one element
if (low == high) {
res.max = arr[low];
res.min = arr[low];
      return res;
   }

   // If two elements     if
(high == low + 1) {        if
(arr[low] > arr[high]) {
res.max = arr[low];
res.min = arr[high];
      } else {
         res.max = arr[high];
         res.min = arr[low];
      }
      return res;
   }

   // If more than two elements
   int mid = (low + high) / 2;

   left = findMaxMin(arr, low, mid);
   right = findMaxMin(arr, mid + 1, high);

   // Compare left and right results     res.max = (left.max
> right.max) ? left.max : right.max;     res.min = (left.min
< right.min) ? left.min : right.min;    return res;
}
```

```
int main() {
   int n, I;
   printf(“Enter number of elements in the array: “);
scanf(“%d”, &n);

   int arr[n];
   printf(“Enter %d elements:\n”, n);
for (I = 0; I < n; i++) {
      scanf(“%d”, &arr[i]);
   }

   struct Result result = findMaxMin(arr, 0, n – 1);

   printf(“Maximum element: %d\n”, result.max);
   printf(“Minimum element: %d\n”, result.min);

   return 0;
}
```

Output:

```
Enter number of elements in the array: 6 Enter
6 elements:
12 45 3 67 23 9
Maximum element: 67
Minimum element: 3
```

Experiment No.: 10

Program Description: Write a program to count the number of inversions in an array using Divide and Conquer.

Solution:

```
#include <stdio.h>

// Function to merge two halves and count inversions
int mergeAndCount(int arr[], int temp[], int left, int mid, int right) {
int I = left;      // Left subarray     int j = mid + 1;   // Right subarray
int k = left;      // Merged array     int inv_count = 0;

   while (I <= mid && j <= right) {
if (arr[i] <= arr[j]) {
           temp[k++] = arr[i++];
       } else {
           temp[k++] = arr[j++];
           inv_count += (mid – I + 1);  // Count inversions
       }
   }

   // Copy remaining elements
   while (I <= mid) {
      temp[k++] = arr[i++];
   }
   while (j <= right) {
      temp[k++] = arr[j++];
   }

   // Copy back to original array
for  (I  =  left;  I  <=  right;  i++)  {
arr[i] = temp[i];
   }

   return inv_count;
}

// Recursive function to count inversions
int countInversions(int arr[], int temp[], int left, int right) {
int mid, inv_count = 0;


  if (left < right) {
      mid = (left + right) / 2;
```

```
        inv_count += countInversions(arr, temp, left, mid);
        inv_count += countInversions(arr, temp, mid + 1, right);

        inv_count += mergeAndCount(arr, temp, left, mid, right);
    }

    return inv_count;
}

int main() {
    int n, I;
    printf(“Enter number of elements: “);
scanf(“%d”, &n);

    int arr[n], temp[n];    printf(“Enter
%d elements:\n”, n);    for (I = 0; I <
n; i++) {
        scanf(“%d”, &arr[i]);
    }

    int result = countInversions(arr, temp, 0, n – 1);
    printf(“Number of inversions in the array: %d\n”, result);

    return 0;
}
```

Output:

```
Enter number of elements: 5 Enter
5 elements:
2 4 1 3 5
Number of inversions in the array: 3
```

<u>Experiment No.: 11</u>

Program Description: (Greedy Algorithms) Write a program for Optimal Merge Pattern.

Solution:

```
#include <stdio.h>

#define MAX 100

// Function to sort the array in ascending order
void  sort(int  arr[],  int  n)  {
int I, j, temp;     for (I = 0; I <
n-1; i++) {         for (j = 0; j <
n-i-1; j++) {           if (arr[j] >
arr[j+1]) {
            // Swap arr[j] and arr[j+1]
            temp = arr[j];
arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}

// Function to calculate Optimal Merge Pattern cost
int optimalMerge(int files[], int n) {
   int totalCost = 0;

   while (n > 1) {
      sort(files, n); // Always keep array sorted

      // Pick two smallest files
int merged = files[0] + files[1];
totalCost += merged;

      // Replace the first two with their merged size
files[0] = merged;        for (int I = 1; I < n – 1; i++) {
         files[i] = files[I + 1];
      }
      n--; // Reduce the number of files
   }

   return totalCost;
}
int main() {
   int n, I;     int
files[MAX];
```

```
   printf(“Enter number of files: “);
scanf(“%d”, &n);

   printf(“Enter sizes of %d files:\n”, n);
   for (I = 0; I < n; i++) {
      scanf(“%d”, &files[i]);
   }

   int cost = optimalMerge(files, n);
   printf(“Minimum total cost of merging: %d\n”, cost);

   return 0;
}
```

Output:

```
Enter number of files: 4 Enter
sizes of 4 files:
20 30 10 5
Minimum total cost of merging: 115
```

Experiment No.: 12

Program Description: Write a program for Huffman Coding Algorithm.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// A node in the Huffman tree
struct Node {     char data;
int freq;
   struct Node *left, *right;
};

// Create a new node
struct Node* createNode(char data, int freq) {
   struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
temp->data = data;      temp->freq = freq;      temp->left = temp-
>right = NULL;     return temp;
}

// Swap function for sorting void swap(struct
Node** a, struct Node** b) {      struct Node*
temp = *a;
   *a = *b;
   *b = temp;
}

// Simple sorting by frequency (Selection Sort)
void sort(struct Node* arr[], int n) {      int I, j,
min;     for (I = 0; I < n-1; i++) {        min = I;
       for (j = i+1; j < n; j++) {            if
(arr[j]->freq < arr[min]->freq)
min = j;
       }
       swap(&arr[i], &arr[min]);
   }
}
// Build Huffman Tree
struct Node* buildHuffmanTree(char data[], int freq[], int size) {
struct Node* nodes[100];  // Max 100 nodes
   for (int I = 0; I < size; i++)
       nodes[i] = createNode(data[i], freq[i]);

   int n = size;

   while (n > 1) {
       sort(nodes, n); // Sort nodes by frequency

       // Take two smallest nodes
struct Node* left = nodes[0];
       struct Node* right = nodes[1];

       // Merge them
       struct Node* merged = createNode('-', left->freq + right->freq);
merged->left = left;        merged->right = right;

       // Replace two nodes with the merged one
       nodes[0]     =     merged;
for  (int  I  =  1;  I  <  n-1;  i++)
nodes[i] = nodes[i+1];
       n--;
   }

   return nodes[0]; // Root of Huffman Tree
}

// Print Huffman codes
void printCodes(struct Node* root, char code[], int top) {
if (root->left) {        code[top] = '0';
       printCodes(root->left, code, top + 1);
   }
   if (root->right) {
code[top] = '1';
       printCodes(root->right, code, top + 1);
   }

   // If it's a leaf node      if (!root->left
&& !root->right) {       code[top] = '\0';
// End the string
       printf("Character: %c, Huffman Code: %s\n", root->data, code);
   }
}
int main() {
int n;
   printf("Enter number of characters: ");
scanf("%d", &n);

   char data[n];
   int freq[n];
   printf("Enter characters:\n");
   for (int I = 0; I < n; i++) {
       scanf(" %c", &data[i]); // space before %c to avoid newline issues
   }

   printf("Enter frequencies:\n");
for (int I = 0; I < n; i++) {
       scanf("%d", &freq[i]);
   }

   struct Node* root = buildHuffmanTree(data, freq, n);

   char code[100];
printf("\nHuffman Codes:\n");
   printCodes(root, code, 0);

   return 0;
}
```

Output:

```
Enter number of characters: 4 Enter
characters:
A B C D
Enter frequencies:
5 9 12 13

Huffman Codes:
Character: A, Huffman Code: 1100
Character: B, Huffman Code: 1101
Character: C, Huffman Code: 10
Character: D, Huffman Code: 0
```

<u>Experiment No.: 13</u>

Program Description: Write a program for Kruskal's Algorithm for MST.

Solution:

```
#include <stdio.h>
```

```
// Structure to represent an edge
struct Edge {     int src,
dest, weight;
};

// Structure to represent a subset for union-find
struct Subset {     int parent;
    int rank;
};

// Function to find the parent of a node (with 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 do union of two sets void
unionSets(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 sort edges by weight (simple Bubble Sort)
void sortEdges(struct Edge edges[], int E) {     for (int I =
0; I < E – 1; i++) {         for (int j = 0; j < E – I – 1; j++) {
            if (edges[j].weight > edges[j + 1].weight) {


  struct Edge temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }
```

```
}

// Kruskal’s algorithm
void kruskalMST(struct Edge edges[], int V, int E) {
struct Subset subsets[V];
   struct Edge result[V – 1]; // Store MST edges
int e = 0; // Count edges in MST
   int I = 0; // Index for sorted edges

   // Create subsets with single elements
   for (int v = 0; v < V; ++v) {
subsets[v].parent = v;        subsets[v].rank
= 0;
   }

   sortEdges(edges, E); // Sort edges by weight

   while (e < V – 1 && I < E) {        struct
Edge nextEdge = edges[i++];        int x
= find(subsets, nextEdge.src);
      int y = find(subsets, nextEdge.dest);

      // If including this edge doesn’t cause a cycle
if (x != y) {
         result[e++] = nextEdge;
         unionSets(subsets, x, y);
      }
   }

   // Print the result
   printf(“\nEdges in Minimum Spanning Tree:\n”);
   int totalWeight = 0;
for (I = 0; I < e; ++i) {
      printf(“%d -- %d == %d\n”, result[i].src, result[i].dest, result[i].weight);
totalWeight += result[i].weight;
   }
   printf(“Total weight of MST: %d\n”, totalWeight);
}

int main() {

 int V, E;
```

```
    printf(“Enter number of vertices: “);
scanf(“%d”, &V);

    printf(“Enter number of edges: “);
scanf(“%d”, &E);

    struct Edge edges[E];

    printf(“Enter each edge as: source destination weight\n”);
for (int I = 0; I < E; i++) {         printf(“Edge %d: “, I + 1);
        scanf(“%d %d %d”, &edges[i].src, &edges[i].dest, &edges[i].weight);
    }

    kruskalMST(edges, V, E);

    return 0;
}
```

Output:

```
Enter number of vertices: 4
Enter number of edges: 5
Enter each edge as: source destination weight
Edge 1: 0 1 10
Edge 2: 0 2 6
Edge 3: 0 3 5
Edge 4: 1 3 15
Edge 5: 2 3 4

Edges in Minimum Spanning Tree:
2 – 3 == 4
0 – 3 == 5
0 – 1 == 10
Total weight of MST: 19
```

Experiment No.: 14

Program Description: Write a program for Prim’s Algorithm for MST.

Solution:

```
#include <stdio.h>
```

```
#define MAX 100
#define INF 9999  // A large number to represent infinity

int main() {     int
cost[MAX][MAX];     int
visited[MAX];     int n, I, j,
no_of_edges = 0;
   int min, a = 0, b = 0;

   printf("Enter number of vertices: ");
scanf("%d", &n);

   printf("Enter the cost adjacency matrix (enter 9999 if no edge):\n");
for (I = 0; I < n; i++) {         for (j = 0; j < n; j++) {
         scanf("%d", &cost[i][j]);
      }
      visited[i] = 0; // Mark all vertices as unvisited initially
   }

   visited[0] = 1; // Start from first vertex

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

   while (no_of_edges < n – 1) {
      min = INF;

      for  (I  =  0;  I  <  n;  i++)  {
if (visited[i]) {               for (j =
0; j < n; j++) {
               if (!visited[j] && cost[i][j] < min) {
                  min = cost[i][j];
                  a = I;
b = j;
               }
            }
         }
      }

      printf("Edge %d: (%d -> %d) cost = %d\n", no_of_edges + 1, a, b, min);
visited[b] = 1;
      no_of_edges++;
   }
```

```
    return 0;
}
```

Output:

```
Enter number of vertices: 4
Enter the cost adjacency matrix (enter 9999 if no edge):
9999 2    9999 6
2    9999 3    8
9999 3    9999 1
6    8    1    9999
```

<u>Experiment No.: 15</u>

Program Description: Write a program for Job Sequencing with deadline.

Solution:

```
#include <stdio.h>
#include <string.h>
#define MAX 100
```

```
// Job structure
struct Job {
char id[10];
int deadline;
int profit;
};

void swap(struct Job *a, struct Job *b) {
struct Job temp = *a;
   *a = *b;
   *b = temp;
}

// Sort jobs by profit (descending) void
sortJobs(struct Job jobs[], int n) {
   for (int I = 0; I < n-1; i++)        for (int j
= 0; j < n-i-1; j++)           if (jobs[j].profit
< jobs[j+1].profit)
            swap(&jobs[j], &jobs[j+1]);
}

int min(int a, int b) {
   return (a < b) ? a : b;
}

int main() {
int n;
   struct Job jobs[MAX];
   int maxDeadline = 0;

   printf("Enter number of jobs: ");
scanf("%d", &n);

   // Input jobs     for (int I
= 0; I < n; i++) {
      printf("Enter Job ID, Deadline, Profit for Job %d: ", I + 1);
scanf("%s %d %d", jobs[i].id, &jobs[i].deadline, &jobs[i].profit);      if
(jobs[i].deadline > maxDeadline)                       maxDeadline =
jobs[i].deadline;
   }

   sortJobs(jobs, n);
```

```
    char schedule[MAX][10];  // To store job IDs at each time slot
int slot[MAX];
    for (int I = 0; I <= maxDeadline; i++) {
slot[i] = 0;
        strcpy(schedule[i], "");
    }

    int totalProfit = 0;

    for (int I = 0; I < n; i++) {
        for (int j = min(maxDeadline, jobs[i].deadline); j > 0; j--) {
if (slot[j] == 0) {
                slot[j] = 1;
                strcpy(schedule[j], jobs[i].id);
totalProfit += jobs[i].profit;
                break;
            }
        }
    }

    printf("\nScheduled Jobs: ");
    for (int I = 1; I <= maxDeadline; i++) {
        if (slot[i])
            printf("%s ", schedule[i]);
    }

    printf("\nTotal Profit: %d\n", totalProfit);

    return 0;
}
```

Output:

```
Enter number of jobs: 5
Enter Job ID, Deadline, Profit for Job 1: J1 2 60
Enter Job ID, Deadline, Profit for Job 2: J2 1 100
Enter Job ID, Deadline, Profit for Job 3: J3 3 20
Enter Job ID, Deadline, Profit for Job 4: J4 2 40
Enter Job ID, Deadline, Profit for Job 5: J5 1 20

Scheduled Jobs: J2 J1 J3
Total Profit: 180
```

Experiment No.: 16

Program Description: Write a program for Fractional Knapsack Problem using greedy approach.

Solution:

```
#include <stdio.h>

#define MAX 100
```

```
// Structure to hold item details
struct Item {     int weight;     int
profit;     float ratio;
};

// Swap function
void swap(struct Item *a, struct Item *b) {
struct Item temp = *a;
   *a = *b;
   *b = temp;
}

// Sort items by profit/weight ratio in descending order void
sortItems(struct Item items[], int n) {
   for (int I = 0; I < n-1; i++) {        for (int j =
0; j < n-i-1; j++) {             if (items[j].ratio <
items[j+1].ratio) {
            swap(&items[j], &items[j+1]);
         }
      }
   }
}

int main() {     int
n, capacity;
   struct Item items[MAX];

   printf("Enter number of items: ");
scanf("%d", &n);

   printf("Enter knapsack capacity: ");
scanf("%d", &capacity);

   // Input profit and weight

  for (int I = 0; I < n; i++) {
      printf("Enter profit and weight of item %d: ", I + 1);
scanf("%d     %d",     &items[i].profit,     &items[i].weight);
items[i].ratio = (float)items[i].profit / items[i].weight;
   }

   // Sort by ratio
   sortItems(items, n);
   float totalProfit = 0.0;
   int remaining = capacity;

   printf("\nItems taken into the knapsack:\n");

   for (int I = 0; I < n; i++) {
      if (items[i].weight <= remaining) {
         //       Take       full       item
totalProfit      +=      items[i].profit;
remaining -= items[i].weight;
         printf("Item %d: 100%% (Profit: %d)\n", I + 1, items[i].profit);
      } else {
         // Take fraction
         float fraction = (float)remaining / items[i].weight;
totalProfit += items[i].profit * fraction;
         printf("Item %d: %.2f%% (Profit: %.2f)\n", I + 1, fraction * 100, items[i].profit * fraction);
break;
      }
   }

   printf("\nTotal profit: %.2f\n", totalProfit);

   return 0;
}
```

Output:

```
Enter number of items: 3
Enter knapsack capacity: 50
Enter profit and weight of item 1: 60 10
Enter profit and weight of item 2: 100 20
Enter profit and weight of item 3: 120 30
Items taken into the knapsack:
Item 1: 100% (Profit: 60)
Item 2: 100% (Profit: 100)
Item 3: 66.67% (Profit: 80.00)

Total profit: 240.00
```

Experiment No.: 17

Program Description: Write a program for Dijkstra's Single Source Shortest Path Algorithm.

Solution:

```
#include <stdio.h>

#define MAX 100
#define INF 9999
```

```
// Function to find the vertex with the minimum distance
int minDistance(int dist[], int visited[], int n) {
int min = INF, min_index = -1;     for (int v = 0;
v < n; v++) {        if (!visited[v] && dist[v] <
min) {
           min = dist[v];
           min_index = v;
        }
    }
    return min_index;
}

// Dijkstra's algorithm function void ijkstra(int
graph[MAX][MAX], int n, int source) {     int
dist[MAX];      // Shortest distance from source     int
visited[MAX];   // Visited array

    // Initialize distances
for (int I = 0; I < n; i++) {
dist[i] = INF;
        visited[i] = 0;
    }

    dist[source] = 0; // Distance to source is 0

    // Find shortest path for all vertices     for
(int count = 0; count < n – 1; count++) {
int u = minDistance(dist, visited, n);
        visited[u] = 1;

        for (int v = 0; v < n; v++) {
            if (!visited[v] && graph[u][v] && dist[u] + graph[u][v] < dist[v]) {
dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    // Output shortest distances
    printf("\nShortest distances from source vertex %d:\n", source);
for (int I = 0; I < n; i++) {
        printf("To vertex %d: %d\n", I, dist[i]);
    }
}
```

```
int main() {
   int graph[MAX][MAX], n, I, j, source;

   //    Input    number    of    vertices
printf(“Enter  number  of  vertices:  “);
scanf(“%d”, &n);

   // Input graph as adjacency matrix
   printf(“Enter the adjacency matrix (0 if no edge):\n”);
for (I = 0; I < n; i++) {        for (j = 0; j < n; j++) {
scanf(“%d”, &graph[i][j]);            if (graph[i][j] == 0 && I
!= j)
            graph[i][j] = INF; // No edge, set to INF
      }
   }

   // Input source vertex
   printf(“Enter the source vertex (0 to %d): “, n – 1);
scanf(“%d”, &source);

   // Call Dijkstra
ijkstra(graph, n, source);

   return 0;
}
```

Output:

```
Enter number of vertices: 4
Enter the adjacency matrix (0 if no edge):
0 5 0 7
5 0 3 0
0 3 0 1
7 0 1 0
Enter the source vertex (0 to 3): 0

Shortest distances from source vertex 0:
To vertex 0: 0
To vertex 1: 5
To vertex 2: 8 To
vertex 3:
```

Experiment No.: 18

Program Description: (Dynamic Programming) Write a program for 0/1 Knapsack Problem using dynamic programming.

Solution:

```
#include <stdio.h>

int max(int a, int b) {
   return (a > b) ? a : b;
}

int main() {
   int n, capacity, I, w;
   printf("Enter number of items: ");
scanf("%d", &n);

   int weight[n], profit[n];
   printf("Enter profits and weights of items:\n");
for (I = 0; I < n; i++) {
      printf("Item %d profit and weight: ", I + 1);
scanf("%d%d", &profit[i], &weight[i]);
   }

   printf("Enter knapsack capacity: ");
scanf("%d", &capacity);

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

   // Build table     for (I
= 0; I <= n; i++) {
      for (w = 0; w <= capacity; w++) {
         if (I == 0 || w == 0)
dp[i][w] = 0;
         else if (weight[I – 1] <= w)
            dp[i][w] = max(profit[I – 1] + dp[I – 1][w – weight[I – 1]], dp[I – 1][w]);
else
            dp[i][w] = dp[I – 1][w];
      }
   }

   printf("Maximum profit: %d\n", dp[n][capacity]);
return 0;
}
```

Output:

```
Enter number of items: 3 Enter
profits and weights of items:
```

```
Item 1 profit and weight: 60 10
Item 2 profit and weight: 100 20
Item 3 profit and weight: 120 30
Enter knapsack capacity: 50
Maximum profit: 220
```

Experiment No.: 19

Program Description :Write a program for Multistage Graph Problem using dynamic programming.
Solution:

```
#include <stdio.h>
#define INF 9999

int main() {     int
n;
   printf("Enter number of vertices in the multistage graph: ");
  scanf(“%d”, &n);

   int cost[n][n], dist[n], path[n];

   printf(“Enter the cost adjacency matrix (use 9999 if no edge):\n”);
for (int I = 0; I < n; i++)         for (int j = 0; j < n; j++)
            scanf(“%d”, &cost[i][j]);

   dist[n – 1] = 0;  // Distance to destination is 0

   for (int I = n – 2; I >= 0; i--) {
dist[i] = INF;
        for (int j = I + 1; j < n; j++) {
            if  (cost[i][j]  !=  INF  &&  cost[i][j]  +  dist[j]  <  dist[i])  {
dist[i] = cost[i][j] + dist[j];                path[i] = j;
            }
        }
   }

   printf(“Minimum cost from source to destination: %d\n”, dist[0]);     return 0;
}
```

Output:

```
Enter number of vertices in the multistage graph: 4 Enter the
cost adjacency matrix:
9999 1 2 9999
9999 9999 3 6
9999 9999 9999 1
9999 9999 9999 9999
Minimum cost from source to destination: 4
```

<u>**Experiment No.: 20**</u>

Program Description: Write a program for Reliability Design Problem (simplified version).

Solution:

```
#include <stdio.h>

int main() {     int
n;
    float rel[100], total = 1.0;
    printf("Enter number of components in series: ");     scanf("%d",
&n);

    printf("Enter reliability of each component (0 to 1):\n");    for
(int I = 0; I < n; i++) {         scanf("%f", &rel[i]);
        total *= rel[i];
    }

    printf("Overall system reliability: %.4f\n", total);     return 0;
}
```

Output:

```
Enter number of components in series: 3 Enter
reliability of each component (0 to 1):
0.9 0.95 0.8
Overall system reliability: 0.6840
```

Experiment No.: 21

Program Description: Write a program for Floyd-Warshall All-Pairs Shortest Path Algorithm.

Solution:

```
#include <stdio.h>
#define INF 9999

int main() {
    int n, i, j, k;
    printf("Enter number of vertices: ");     scanf("%d",
&n);

    int graph[n][n];
    printf("Enter adjacency matrix (use 9999 if no edge):\n");
for (i = 0; i < n; i++)         for (j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    // Floyd-Warshall algorithm     for (k = 0; k < n;
k++) {          for (i = 0; i < n; i++) {             for (j = 0; j <
n; j++) {                   if (graph[i][k] + graph[k][j] <
graph[i][j])                     graph[i][j] = graph[i][k] +
graph[k][j];
            }
        }
    }

    printf("All-Pairs Shortest Path Matrix:\n");
    for (i = 0; i < n; i++) {         for
(j = 0; j < n; j++) {             if
(graph[i][j] == INF)
printf("INF ");
            else
                printf("%3d ", graph[i][j]);
        }
        printf("\n");
    }

    return 0;
)
```

Output:

```
Enter number of vertices: 3 Enter adjacency matrix:
0 5 9999
9999 0 2
3 9999 0
All-Pairs Shortest Path Matrix:
  0   5   7
  5   0   2
  3   8   0
```

<u>Experiment No.: 22</u>

Program Description: (Backtracking) Write a program for the N-Queens Problem.

Solution:

```
#include <stdio.h>
#include <math.h>

int board[20], count = 0;

int isSafe(int row, int col) {     for (int i = 1; i <
row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
 return 0;
    }
    return 1;
}

void printBoard(int n) {
    printf("\nSolution %d:\n", ++count);     for (int i = 1; i
<= n; i++) {         for (int j = 1; j <= n; j++) {             if
(board[i] == j)                 printf("Q ");             else
                printf(". ");
        }
        printf("\n");
    }
}

void nQueens(int row, int n) {     for (int col = 1; col
<= n; col++) {
        if (isSafe(row, col)) {             board[row]
= col;             if (row == n)
printBoard(n);
            else
                nQueens(row + 1, n);
        }
    }
}
```

```
int main() {
int n;
printf("Enter number of queens: ");
scanf("%d", &n);
nQueens(1, n);
if (count == 0)
 printf("No solution exists.\n");
return 0;
}
```

Output:

```
Enter number of queens: 4

Solution 1:
. Q . .
. . . Q Q . . .
. . Q .

Solution 2:
. . Q .
Q . . .
. . . Q . Q . .
```

Experiment No.: 23

Program Description: Write a program for the Hamiltonian Cycle Problem.
Solution:

```
#include <stdio.h>
#define MAX 10

int path[MAX], visited[MAX];

int isSafe(int v, int pos, int graph[MAX][MAX], int n) {
if (graph[path[pos - 1]][v] == 0)
        return 0;     for (int i =
0; i < pos; i++)         if
(path[i] == v)           return
0;
    return 1;
}

int hamiltonianCycle(int graph[MAX][MAX], int pos, int n) {
if (pos == n) {
        return graph[path[pos - 1]][path[0]] == 1;
    }

    for (int v = 1; v < n; v++) {       if
(isSafe(v,    pos,    graph,    n))    {
path[pos] = v;
            if (hamiltonianCycle(graph, pos + 1, n))
return 1;
            path[pos] = -1;
        }
    }
    return 0;
}

int main() {     int n, graph[MAX][MAX];
printf("Enter   number   of   vertices:");
scanf("%d", &n);

    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
```

```
   for (int i = 0; i < n; i++) {
path[i] = -1;
        visited[i] = 0;
    }

    path[0] = 0;

    if (hamiltonianCycle(graph, 1, n)) {
printf("Hamiltonian Cycle exists:\n");
for (int i = 0; i < n; i++)
printf("%d ", path[i]);
printf("%d\n", path[0]);
    }
else {
        printf("No Hamiltonian Cycle exists.\n");
    }

    return 0;
}
```

Output:

```
Enter number of vertices: 4 Enter
adjacency matrix:
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0
Hamiltonian Cycle exists:
0 1 2 3 0
```

Experiment No.: 24

Program Description:Write a program for the Graph Coloring Problem using backtracking.
Solution:

```
#include <stdio.h>
#define MAX 10

int graph[MAX][MAX], color[MAX];

int isSafe(int v, int c, int n) {     for
(int i = 0; i < n; i++)         if
(graph[v][i] && color[i] == c)
            return 0;
    return 1;
}

int graphColoring(int v, int n, int m) {
if (v == n)
        return 1;

    for (int c = 1; c <= m; c++) {
if (isSafe(v, c, n)) {
color[v] = c;
            if (graphColoring(v + 1, n, m))
                return 1;
            color[v] = 0;
        }
    }
    return 0;
}

int main() {
int n, m;
    printf("Enter number of vertices: ");
scanf("%d", &n);
    printf("Enter number of colors: ");
scanf("%d", &m);

    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
```

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

    for (int i = 0; i < n; i++)
color[i] = 0;
    if (graphColoring(0, n, m)) {
printf("Colors assigned to vertices:\n");
        for (int i = 0; i < n; i++)
            printf("Vertex %d --> Color %d\n", i, color[i]);
    } else {
        printf("No solution exists with %d colors.\n", m);
    }

    return 0;
}
```

Output:

```
Enter number of vertices: 4
Enter  number  of  colors:  3
Enter adjacency matrix:
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0
Colors assigned to vertices:
Vertex 0 --> Color 1
Vertex 1 --> Color 2
Vertex 2 --> Color 3
Vertex 3 --> Color 2
```

Experiment No.: 25

Program Description: Write a program to generate permutations of a string using backtracking.
Solution:

```
#include <stdio.h>
#include <string.h>

void swap(char *x, char *y) {
char temp = *x;
   *x = *y;
   *y = temp;
}

void permute(char str[], int l, int r) {
if (l == r)
      printf("%s\n", str);
   else {       for (int i = l; i
<= r; i++) {
swap(&str[l], &str[i]);
permute(str, l + 1, r);
         swap(&str[l], &str[i]); // backtrack
      }
   }
}

int main() {    char
str[100];    printf("Enter a
string: ");    scanf("%s",
str);

   int n = strlen(str);
printf("Permutations are:\n");
   permute(str, 0, n - 1);

   return 0;
}
```

Output:

```
Enter a string: abc
Permutations are:
```

```
abc acb bac bca
cba cab
```

Experiment No.: 26

Program Description: (Branch and Bound) Write a program for Traveling Salesman Problem (TSP) using Branch and Bound.

Solution:

```
#include <stdio.h>
#define INF 9999
#define MAX 10

int tsp(int graph[MAX][MAX], int visited[], int n, int pos, int count, int cost,
int start, int *minCost) {    if (count == n && graph[pos][start]) {       int
totalCost = cost + graph[pos][start];       if (totalCost < *minCost)
*minCost = totalCost;
        return *minCost;
    }

    for (int i =
0; i < n; i++) {
if (!visited[i]
&&
graph[pos][i])
{
visited[i] = 1;
            tsp(graph, visited, n, i, count + 1, cost + graph[pos][i],
start, minCost);          visited[i] = 0;
        }
    }
    return *minCost;
}

int main() {
    int
graph[MAX][
MAX],        n;
printf("Enter
number      of
cities:        ");
scanf("%d",
&n);

    printf("Enter cost
matrix (%d x %d):\n", n,
n);     for (int i = 0; i < n;
i++)         for (int j = 0; j <
n; j++)             scanf("%d",
&graph[i][j]);

    int visited[MAX] = {0},
minCost = INF;     visited[0]
= 1;

    int result = tsp(graph, visited, n, 0, 1, 0, 0,
&minCost);
printf("Minimum cost of visiting all cities: %d\n",
result);

    return 0;
}
```

Output:

```
Enter number of cities: 4
Enter cost matrix:
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
Minimum cost of visiting all cities: 80
```

Experiment No.: 27

Program Description: Write a program to solve the 0/1 Knapsack Problem using Branch and Bound.

Solution:

```
#include <stdio.h>
#include <stdlib.h>

struct Item {
int weight, profit;
 float ratio;
};
// Sorting items by profit/weight ratio
void sortItems(struct Item arr[], int n) {
      struct Item temp;
      for (int i = 0; i < n-1; i++)
          for (int j = 0; j < n-i-1; j++)
              if (arr[j].ratio < arr[j+1].ratio) {
                  temp = arr[j];
                   arr[j] = arr[j+1];
                   arr[j+1] = temp;
             }
}

// Bound calculation
float bound(int level, int weight, int profit, int capacity, struct Item items[], int n){
float result = profit;
      int w = weight;

    for (int i = level; i < n && w < capacity; i++) {        if (w + items[i].weight <=
capacity) {            w += items[i].weight;            result += items[i].profit;
        } else {
            int rem = capacity - w;            result += rem * items[i].ratio;            break;
        }
    }
    return result;
}
// Branch and Bound
void knapsack(int level, int weight, int profit, float *maxProfit, int capacity, struct
Item items[], int n) {     if (weight > capacity)        return;
    if (profit > *maxProfit)
```

```
        *maxProfit = profit;

    if (bound(level, weight, profit, capacity, items, n) < *maxProfit)        return;

    if (level < n) {
        knapsack(level + 1, weight + items[level].weight, profit + items[level].profit,
maxProfit, capacity, items, n);
        knapsack(level + 1, weight, profit, maxProfit, capacity, items, n);
    }
}

int main() {     int n, capacity;
    printf("Enter number of items: ");     scanf("%d", &n);

    struct Item items[n];

    printf("Enter profit and weight of each item:\n");     for (int i = 0; i < n; i++) {
        scanf("%d%d", &items[i].profit, &items[i].weight);        items[i].ratio =
(float)items[i].profit / items[i].weight;
    }

    printf("Enter knapsack capacity: ");     scanf("%d", &capacity);

    sortItems(items, n);
    float maxProfit = 0;

    knapsack(0, 0, 0, &maxProfit, capacity, items, n);

    printf("Maximum profit using Branch and Bound: %.2f\n", maxProfit);

    return 0;
}
```

Output:

```
Enter number of items: 4
Enter profit and weight of each item:
40 2
30 5
50 10
10 5
Enter knapsack capacity: 16
Maximum profit using Branch and Bound: 90.00
```

Experiment No.: 28

Program Description: (Trees and Graphs) Write a program for Inorder, Preorder, and Postorder Traversals of a Binary Tree.

Solution:

```
#include <stdio.h>
#include <stdlib.h>

// Structure
for a binary
tree node
struct Node
{    int data;
   struct Node *left, *right;
};

// Create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));    newNode->data = data;
   newNode->left =
newNode->right = NULL;
return newNode;
}

// Inorder
Traversal
void
inorder(str
uct Node*
root) {
   if (root !=
NULL) {
inorder(root
->left);
printf("%d ",
root->data);
      inorder(root->right);
   }
}
```

```
// Preorder
Traversal
void
preorder(str
uct Node*
root) {
   if (root !=
NULL) {
printf("%d ",
root->data);
preorder(roo
t->left);

preorder(roo
t->right);
   }
}
 // Postorder
Traversal
void
postorder(str
uct Node*
root) {
   if (root !=
NULL) {
postorder(root-
>left);
postorder(root-
>right);
printf("%d ",
root->data);     }
}

int main() {
   int n, val;
   struct Node *root = NULL, *queue[100];
   int front = 0, rear = 0;

   printf("Enter number of nodes to insert in
binary tree: ");     scanf("%d", &n);

   if (n
== 0)
{
```

```
printf
("Emp
ty
tree.\
n");
        return 0;
    }


printf("Enter
root      node
value:        ");
scanf("%d",
&val);     root
=
createNode(
val);
    queue[rear++] = root;

for  (int  i  =  1;  i  <  n;
i++) {
 struct  Node*  temp
=queue[front++];
int leftVal, rightVal;

        printf("Enter left child of %d (-1 for no node): ",
temp->data);
scanf("%d", &leftVal);
        if (leftVal != -1) {
            temp->left = createNode(leftVal);
            queue[rear++] = temp->left;
        }

        printf("Enter right child of %d (-1 for no node): ",
temp->data);
scanf("%d", &rightVal);
        if (rightVal != -1) {
            temp->right = createNode(rightVal);
            queue[rear++] = temp->right;
        }
    }

    printf("\nInorder traversal: ");
   inorder(root);
```

```
    printf("\nPreorder traversal: ");
   preorder(root);

    printf("\nPostorder traversal: ");
   postorder(root);

    printf("\n");
    return 0; }
```

Output:

```
Enter number of nodes to insert in binary tree: 5
Enter root node value: 1
Enter left child of 1 (-1 for no node): 2
Enter right child of 1 (-1 for no node): 3
Enter left child of 2 (-1 for no node): 4
Enter right child of 2 (-1 for no node): 5
Enter left child of 3 (-1 for no node): -1
Enter right child of 3 (-1 for no node): -1
Enter left child of 4 (-1 for no node): -1
Enter right child of 4 (-1 for no node): -1
Enter left child of 5 (-1 for no node): -1
Enter right child of 5 (-1 for no node): -1

Inorder traversal: 4 2 5 1 3
Preorder traversal: 1 2 4 5 3
Postorder traversal: 4 5 2 3 1
```

Experiment No.: 29

Program Description: Write a program for Depth-First Search (DFS) traversal

Solution:

```
#include <stdio.h>

#define MAX 100

int graph[MAX][MAX], visited[MAX];

void DFS(int v, int n) {     visited[v] = 1;     printf("%d ", v);

    for (int i = 0; i < n; i++) {
        if (graph[v][i] == 1 && !visited[i]) {
            DFS(i, n);        }
    }
}

int main() {
    int n, edges, u, v, start;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter number of edges: ");
    scanf("%d", &edges);

    // Initialize adjacency matrix
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
            graph[i][j] = 0;

    // Input edges
for (int i = 0; i < edges; i++) {
    printf("Enter edge (u v): ");
    scanf("%d%d", &u, &v);
     graph[u][v] = 1;
     graph[v][u] = 1;
// For undirected graph
    }
```

```
       printf("Enter starting vertex for DFS: ");
       scanf("%d", &start);

printf("DFStraversal: ");
DFS(start, n);
 printf("\n");

return 0;
}
```

Output:

```
Enter number of vertices: 5
Enter number of edges: 4
Enter edge (u v): 0 1
Enter edge (u v): 0 2
Enter edge (u v): 1 3
Enter edge (u v): 1 4
Enter starting vertex for DFS: 0
DFS traversal: 0 1 3 4 2
```