Slip 1

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

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

void selectionSort(int arr[], int n) {
   int i, j, min_idx;
   for (i = 0; i < n - 1; i++) {
      min_idx = i;
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[min_idx]) {
            min_idx = j;
         }
      }
      // Swap the found minimum element with the first element
      int temp = arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
   }
}

double measureTime(int arr[], int n) {
   clock_t start, end;
   double time_taken;

   start = clock();
   selectionSort(arr, n);
   end = clock();

   time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
   return time_taken;
}

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

   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed.\n");
      return 1;
   }

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

   double time_taken = measureTime(arr, n);

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

   printf("Time taken: %f seconds\n", time_taken);

   free(arr);
   return 0;
}
```

Q.2) Write a program to sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from a file or can be generated using the random number generator.

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

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

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    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;
}

// Function to implement Quick sort algorithm
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);
    }
}

// Function to generate random numbers and fill the array
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000; // generating random numbers between 0 and 999
    }
}

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

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Generating random array
    generateRandomArray(arr, n);

    // Sorting and measuring time
    clock_t start, end;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

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

    printf("Time taken: %f seconds\n", time_taken);

    free(arr);
    return 0;
    }
```

Slip 2

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

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

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

// Function to heapify a subtree rooted with node i which is an index in arr[]
void heapify(int arr[], int n, int i) {
  int largest = i;    // Initialize largest as root
  int left = 2 * i + 1;   // left = 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], &arr[largest]);
    // Recursively heapify the affected sub-tree
    heapify(arr, n, largest);
  }
}

// main function to do heap sort
void heapSort(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
    swap(&arr[0], &arr[i]);
    // call max heapify on the reduced heap
    heapify(arr, i, 0);
  }
}

// Function to generate random numbers and fill the array
void generateRandomArray(int arr[], int n) {
   for (int i = 0; i < n; i++) {
      arr[i] = rand() % 1000; // generating random numbers between 0 and 999
   }
}

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

   int *arr = (int *)malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed.\n");
      return 1;
   }

   // Generating random array
   generateRandomArray(arr, n);

   // Sorting using heap sort
   clock_t start, end;
   start = clock();
   heapSort(arr, n);
   end = clock();

   double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

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

   printf("Time taken: %f seconds\n", time_taken);

   free(arr);
   return 0;
}
```

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

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

// Function to add two matrices
void add(int n, int A[][n], int B[][n], int C[][n]) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
}

// Function to subtract two matrices
void subtract(int n, int A[][n], int B[][n], int C[][n]) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         C[i][j] = A[i][j] - B[i][j];
      }
   }
}

// Function to multiply two matrices using Strassen's algorithm
void strassen(int n, int A[][n], int B[][n], int C[][n]) {
   if (n == 1) {
      C[0][0] = A[0][0] * B[0][0];
      return;
   }

   int newSize = n / 2;
   int A11[newSize][newSize], A12[newSize][newSize], A21[newSize][newSize], A22[newSize][newSize];
   int B11[newSize][newSize], B12[newSize][newSize], B21[newSize][newSize], B22[newSize][newSize];
   int C11[newSize][newSize], C12[newSize][newSize], C21[newSize][newSize], C22[newSize][newSize];
   int M1[newSize][newSize], M2[newSize][newSize], M3[newSize][newSize], M4[newSize][newSize],
M5[newSize][newSize], M6[newSize][newSize], M7[newSize][newSize];
   int temp1[newSize][newSize], temp2[newSize][newSize];

   // Divide matrices A and B into submatrices
   for (int i = 0; i < newSize; i++) {
      for (int j = 0; j < newSize; j++) {
         A11[i][j] = A[i][j];
         A12[i][j] = A[i][j + newSize];
         A21[i][j] = A[i + newSize][j];
         A22[i][j] = A[i + newSize][j + newSize];

         B11[i][j] = B[i][j];
         B12[i][j] = B[i][j + newSize];
         B21[i][j] = B[i + newSize][j];
         B22[i][j] = B[i + newSize][j + newSize];
```

```
        }
    }

    // Calculate M1 to M7
    add(newSize, A11, A22, temp1);
    add(newSize, B11, B22, temp2);
    strassen(newSize, temp1, temp2, M1);

    add(newSize, A21, A22, temp1);
    strassen(newSize, temp1, B11, M2);

    subtract(newSize, B12, B22, temp1);
    strassen(newSize, A11, temp1, M3);

    subtract(newSize, B21, B11, temp1);
    strassen(newSize, A22, temp1, M4);

    add(newSize, A11, A12, temp1);
    strassen(newSize, temp1, B22, M5);

    subtract(newSize, A21, A11, temp1);
    add(newSize, B11, B12, temp2);
    strassen(newSize, temp1, temp2, M6);

    subtract(newSize, A12, A22, temp1);
    add(newSize, B21, B22, temp2);
    strassen(newSize, temp1, temp2, M7);

    // Calculate C11, C12, C21, C22
    add(newSize, M1, M4, temp1);
    subtract(newSize, temp1, M5, temp2);
    add(newSize, temp2, M7, C11);

    add(newSize, M3, M5, C12);

    add(newSize, M2, M4, C21);

    add(newSize, M1, M3, temp1);
    subtract(newSize, temp1, M2, temp2);
    add(newSize, temp2, M6, C22);

    // Combine submatrices into result matrix C
    for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
            C[i][j] = C11[i][j];
            C[i][j + newSize] = C12[i][j];
            C[i + newSize][j] = C21[i][j];
            C[i + newSize][j + newSize] = C22[i][j];
        }
    }
}

// Function to print a matrix
void printMatrix(int n, int matrix[][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n;
    printf("Enter the size of the matrices (power of 2): ");
    scanf("%d", &n);

    // Creating matrices A and B
    int A[n][n], B[n][n];
    printf("Enter the elements of matrix A:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &A[i][j]);
        }
    }
    printf("Enter the elements of matrix B:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &B[i][j]);
        }
    }

    // Resultant matrix C
    int C[n][n];

    // Perform Strassen's matrix multiplication
    strassen(n, A, B, C);

    // Print the resultant matrix C
    printf("Resultant matrix C:\n");
    printMatrix(n, C);

    return 0;
}
```

Slip 3

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

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

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

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = low - 1;
  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;
}

// Function to implement Quick sort algorithm
void quickSort(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

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

  int *arr = (int *)malloc(n * sizeof(int));
  if (arr == NULL) {
    printf("Memory allocation failed.\n");
    return 1;
  }
```

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

    // Sorting and measuring time
    clock_t start, end;
    start = clock();
    quickSort(arr, 0, n - 1);
    end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

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

    printf("Time taken: %f seconds\n", time_taken);

    free(arr);
    return 0;
}
```

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

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

#define V 5 // Number of vertices in the graph

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; 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]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
```

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

void primMST(int graph[V][V]) {
    int parent[V]; // Array to store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }

    // Always include first  vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices
        // of the picked vertex. Consider only those vertices which are
        // not yet included in MST
        for (int v = 0; v < V; v++) {
            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // print the constructed MST
    printMST(parent, graph);
}

int main() {
    // Example graph represented as an 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}};

    // Print the Minimum Spanning Tree
    primMST(graph);

    return 0;
}
```

Slip 4

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

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

// Function to merge two subarrays of arr[]
void merge(int arr[], int l, int m, int r) {
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   // Create temporary arrays
   int L[n1], R[n2];

   // Copy data to temporary arrays L[] and R[]
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   // Merge the temporary arrays back into arr[l..r]
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }

   // Copy the remaining elements of L[], if any
   while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }

   // Copy the remaining elements of R[], if any
   while (j < n2) {
      arr[k] = R[j];
      j++;
```

```
        k++;
    }
}

// Merge sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for large l and r
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

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

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

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

    // Sorting and measuring time
    clock_t start, end;
    start = clock();
    mergeSort(arr, 0, n - 1);
    end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    printf("Time taken: %f seconds\n", time_taken);

    free(arr);
    return 0;
}
```

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

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

// Structure for items
struct Item {
    int value;
    int weight;
};

// Function to compare two items by their value-to-weight ratio
int compare(const void *a, const void *b) {
    double ratioA = ((double)((struct Item *)a)->value) / ((struct Item *)a)->weight;
    double ratioB = ((double)((struct Item *)b)->value) / ((struct Item *)b)->weight;
    if (ratioA > ratioB) return -1;
    if (ratioA < ratioB) return 1;
    return 0;
}

// Greedy function to solve Knapsack problem
void knapsackGreedy(int W, struct Item arr[], int n) {
    // Sort items by value-to-weight ratio
    qsort(arr, n, sizeof(struct Item), compare);

    int currentWeight = 0; // Current weight in knapsack
    double totalValue = 0.0; // Total value in knapsack

    // Loop through sorted items and select greedily
    for (int i = 0; i < n; i++) {
        // If adding the current item doesn't exceed the weight limit, add it
        if (currentWeight + arr[i].weight <= W) {
            currentWeight += arr[i].weight;
            totalValue += arr[i].value;
        } else {
            // If adding the current item exceeds the weight limit, add a fraction of it
            int remainingWeight = W - currentWeight;
            totalValue += arr[i].value * ((double)remainingWeight / arr[i].weight);
            break;
        }
    }

    printf("Total value in knapsack: %lf\n", totalValue);
}

int main() {
    int n, W; // Number of items and the knapsack capacity
    printf("Enter the number of items: ");
    scanf("%d", &n);
    printf("Enter the knapsack capacity: ");
    scanf("%d", &W);

    struct Item *arr = (struct Item *)malloc(n * sizeof(struct Item));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the value and weight of each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d %d", &arr[i].value, &arr[i].weight);
    }

    // Solve the Knapsack problem using Greedy approach
    knapsackGreedy(W, arr, n);

    free(arr);
    return 0;
}
```

Slip 5

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

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

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

// Structure to represent a graph
struct Graph {
   int V, E;
   struct Edge *edge;
};

// Function to create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
   struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
   graph->V = V;
   graph->E = E;
   graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));
   return graph;
}

// Function to find the subset of an element i
int find(int parent[], int i) {
   if (parent[i] == -1)
      return i;
   return find(parent, parent[i]);
}

// Function to perform union of two subsets
void Union(int parent[], int x, int y) {
   int xset = find(parent, x);
   int yset = find(parent, y);
   parent[xset] = yset;
}

// Function to implement Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
   int V = graph->V;
   struct Edge result[V];
   int e = 0;
   int i = 0;

   // Step 1: Sort all the edges in non-decreasing order of their weight
   qsort(graph->edge, graph->E, sizeof(graph->edge[0]), [](const void* a, const void* b) {
        struct Edge* a1 = (struct Edge*)a;
        struct Edge* b1 = (struct Edge*)b;
        return a1->weight > b1->weight;
    });

    // Allocate memory for creating V subsets
    int *parent = (int*) malloc(V * sizeof(int));

    // Initialize all subsets as single element sets
    for (int v = 0; v < V; ++v)
        parent[v] = -1;

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E) {
        // Step 2: Pick the smallest edge. And increment the index for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(parent, next_edge.src);
        int y = find(parent, next_edge.dest);

        // If including this edge does't cause cycle, include it
        if (x != y) {
            result[e++] = next_edge;
            Union(parent, x, y);
        }
        // Else discard the next_edge
    }

    // Print the minimum spanning tree
    printf("Following are the edges in the minimum spanning tree:\n");
    int minimumCost = 0;
    for (i = 0; i < e; ++i) {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
        minimumCost += result[i].weight;
    }
    printf("Minimum Cost Spanning Tree: %d\n", minimumCost);
}

int main() {
    int V, E; // Number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);

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

    printf("Enter the source, destination and weight of each edge:\n");
    for (int i = 0; i < E; ++i) {
        scanf("%d %d %d", &graph->edge[i].src, &graph->edge[i].dest, &graph->edge[i].weight);
    }

    // Call Kruskal's algorithm
    KruskalMST(graph);

    return 0;
}
```

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

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

// Define maximum number of characters in the input
#define MAX_CHAR 26

// Define structure for a Huffman tree node
struct MinHeapNode {
    char data;
    unsigned freq;
    struct MinHeapNode *left, *right;
};

// Define structure for a Min Heap
struct MinHeap {
    unsigned size;
    unsigned capacity;
    struct MinHeapNode **array;
};

// Function to create a new Min Heap Node
struct MinHeapNode* newNode(char data, unsigned freq) {
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// Function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
MinHeapNode*));
```

```
    return minHeap;
}

// Function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// Function to heapify at given index
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

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

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

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// Function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap) {
    return (minHeap->size == 1);
}

// Function to extract minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// Function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode) {
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
```

```
  }
  minHeap->array[i] = minHeapNode;
}

// Function to build the Huffman Tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
  struct MinHeapNode *left, *right, *top;
  struct MinHeap* minHeap = createMinHeap(size);
  for (int i = 0; i < size; ++i)
    insertMinHeap(minHeap, newNode(data[i], freq[i]));
  while (!isSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);
    top = newNode('$', left->freq + right->freq);
    top->left = left;
    top->right = right;
    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}

// Function to print Huffman codes from the root of Huffman Tree
void printCodes(struct MinHeapNode* root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printCodes(root->right, arr, top + 1);
  }
  if (!(root->left) && !(root->right)) {
    printf("%c: ", root->data);
    for (int i = 0; i < top; ++i)
      printf("%d", arr[i]);
    printf("\n");
  }
}

// Function to build Huffman Tree and print codes
void HuffmanCodes(char data[], int freq[], int size) {
  struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
  int arr[MAX_CHAR], top = 0;
  printCodes(root, arr, top);
}

int main() {
  char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
  int freq[] = { 5, 9, 12, 13, 16, 45 };
```

```
    int size = sizeof(data) / sizeof(data[0]);
    printf("Huffman Codes:\n");
    HuffmanCodes(data, freq, size);
    return 0;
}
```

Slip 6

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

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

// Number of vertices in the graph
#define V 5

// Function to find the vertex with minimum key value, from the set of vertices not yet included in MST
int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++) {
      if (mstSet[v] == 0 && key[v] < min) {
         min = key[v];
         min_index = v;
      }
   }

   return min_index;
}

// Function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V]) {
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++)
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency matrix representation
void primMST(int graph[V][V]) {
   int parent[V]; // Array to store constructed MST
   int key[V];    // Key values used to pick minimum weight edge in cut
   int mstSet[V]; // To represent set of vertices included in MST

   // Initialize all keys as INFINITE
   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = 0;
   }

   // Always include first  vertex in MST.
   key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
   parent[0] = -1; // First node is always root of MST

   // The MST will have V vertices
   for (int count = 0; count < V - 1; count++) {
      // Pick the minimum key vertex from the set of vertices
      // not yet included in MST
      int u = minKey(key, mstSet);

      // Add the picked vertex to the MST Set
      mstSet[u] = 1;

      // Update key value and parent index of the adjacent vertices
      // of the picked vertex. Consider only those vertices which are
      // not yet included in MST
      for (int v = 0; v < V; v++) {
         // graph[u][v] is non zero only for adjacent vertices of m
         // mstSet[v] is false for vertices not yet included in MST
         // Update the key only if graph[u][v] is smaller than key[v]
         if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
             parent[v] = u;
             key[v] = graph[u][v];
         }
      }
   }

   // Print the constructed MST
   printMST(parent, graph);
}

int main() {
   /* Let us create the following graph
         2      3
      (0)---(1)---(2)
         |     / \    |
      6  |   8/   \5  |7
         |   /     \  |
      (3)----------(4)
            9
*/
   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}};

   // Print the Minimum Spanning Tree
   primMST(graph);

   return 0;
}
```

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

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

// Function to find the length of the Longest Common Subsequence of two strings
int lcsLength(char *X, char *Y) {
   int m = strlen(X);
   int n = strlen(Y);

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

   // Build L[m+1][n+1] in bottom-up fashion
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0 || j == 0)
            L[i][j] = 0;
         else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;
         else
            L[i][j] = (L[i - 1][j] > L[i][j - 1]) ? L[i - 1][j] : L[i][j - 1];
      }
   }

   return L[m][n];
}

int main() {
   char X[] = "AGGTAB";
   char Y[] = "GXTXAYB";

   int length = lcsLength(X, Y);
   printf("Length of Longest Common Subsequence: %d\n", length);

   return 0;
}
```

Slip 7

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

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

// Number of vertices in the graph
#define V 9

// Function to find the vertex with minimum distance value, from the set of vertices not yet included in
the shortest path tree
int minDistance(int dist[], int sptSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (sptSet[v] == 0 && dist[v] <= min)
         min = dist[v], min_index = v;

   return min_index;
}

// Function to print the constructed distance array
void printSolution(int dist[], int n) {
   printf("Vertex   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
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

   int 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 sptSet[] as false
   for (int i = 0; i < V; i++)
      dist[i] = INT_MAX, sptSet[i] = 0;

   // 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] = 1;

        // 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, V);
}

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}};

    dijkstra(graph, 0);

    return 0;
}
```

Q.3) Write a program for finding Topological sorting for Directed Acyclic Graph (DAG)

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

#define MAX_VERTICES 100

struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjLists;
    int* visited;
};

// Create a new Node
struct Node* createNode(int v) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Create a Graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

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

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

// DFS function
```

```
void DFS(struct Graph* graph, int vertex, int visited[], struct Node** stack, int* top) {
    visited[vertex] = 1;

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

    while (temp) {
        int adjVertex = temp->vertex;
        if (!visited[adjVertex]) {
            DFS(graph, adjVertex, visited, stack, top);
        }
        temp = temp->next;
    }

    (*top)++;
    stack[*top] = createNode(vertex);
}

// Topological Sorting
void topologicalSort(struct Graph* graph) {
    int visited[MAX_VERTICES];
    struct Node* stack[MAX_VERTICES];
    int top = -1;

    for (int i = 0; i < graph->numVertices; i++) {
        visited[i] = 0;
    }

    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            DFS(graph, i, visited, stack, &top);
        }
    }

    printf("\nTopological sorting: ");
    while (top >= 0) {
        printf("%d ", stack[top--]->vertex);
    }
}

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 5, 2);
    addEdge(graph, 5, 0);
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);

    printf("Graph:\n");
    printGraph(graph);

    topologicalSort(graph);

    return 0;
}
```

Slip 8

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

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

// Structure to represent items
struct Item {
  int value;
  int weight;
};

// Function to compare two items based on their value-to-weight ratio
int compare(const void *a, const void *b) {
  double ratioA = ((struct Item *)a)->value / (double)((struct Item *)a)->weight;
  double ratioB = ((struct Item *)b)->value / (double)((struct Item *)b)->weight;
  return (ratioB > ratioA) - (ratioB < ratioA);
}

// Function to solve fractional knapsack problem
double fractionalKnapsack(int W, struct Item arr[], int n) {
  // Sort items based on their value-to-weight ratio
  qsort(arr, n, sizeof(arr[0]), compare);

  int currentWeight = 0; // Current weight in knapsack
  double totalValue = 0.0; // Total value in knapsack

  // Loop through sorted items and select greedily
  for (int i = 0; i < n; i++) {
    // If adding the current item doesn't exceed the weight limit, add it fully
    if (currentWeight + arr[i].weight <= W) {
      currentWeight += arr[i].weight;
      totalValue += arr[i].value;
    } else {
      // If adding the current item exceeds the weight limit, add a fraction of it
      int remainingWeight = W - currentWeight;
      totalValue += arr[i].value * ((double)remainingWeight / arr[i].weight);
      break;
    }
  }

  return totalValue;
}

int main() {
  int n, W; // Number of items and the knapsack capacity
  printf("Enter the number of items: ");
  scanf("%d", &n);
  printf("Enter the knapsack capacity: ");
```

```
    scanf("%d", &W);

    struct Item *arr = (struct Item *)malloc(n * sizeof(struct Item));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the value and weight of each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d %d", &arr[i].value, &arr[i].weight);
    }

    // Solve the Fractional Knapsack problem using Greedy approach
    double totalValue = fractionalKnapsack(W, arr, n);
    printf("Total value in knapsack: %.2f\n", totalValue);

    free(arr);
    return 0;
}
```

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

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

#define V 4 // Number of vertices

// Function to find the vertex with minimum distance and is not visited
int minDistance(int dist[], int visited[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (visited[v] == 0 && dist[v] < min)
            min = dist[v], min_index = v;

    return min_index;
}

// Function to print the path
void printPath(int parent[], int graph[V][V]) {
    printf("Path: ");
    int j;
    for (int i = 1; i < V; i++) {
        j = parent[i];
        printf("%d -> ", j);
    }
    printf("0\n");
}

// Function to implement the nearest neighbor algorithm for Traveling Salesman Problem
void nearestNeighbor(int graph[V][V]) {
    int visited[V]; // To keep track of visited vertices
    int dist[V];    // To store the distance from the current vertex to its nearest neighbor
    int parent[V];  // To store the parent of each vertex in the tour

    // Initialize visited[], dist[] and parent[] arrays
    for (int i = 0; i < V; i++) {
        visited[i] = 0;
        dist[i] = INT_MAX;
        parent[i] = -1;
    }

    // Mark the starting vertex as visited
    visited[0] = 1;

    // Start from the first vertex
    int current = 0;
    int count = 0;
    while (count < V - 1) {
        // Find the nearest neighbor of the current vertex
        int nearest = minDistance(graph[current], visited);

        // Add the nearest neighbor to the tour
        parent[nearest] = current;
        visited[nearest] = 1;
        current = nearest;
        count++;
    }

    // Print the tour
    printf("Nearest Neighbor Tour: ");
    printPath(parent, graph);
}

int main() {
    int graph[V][V] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    // Solve the TSP using the nearest neighbor algorithm
    nearestNeighbor(graph);
```

```
   return 0;
}
```

Slip 8

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

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

// Function to calculate the sum of probabilities from i to j
float sumOfProbabilities(float probabilities[], int i, int j) {
   float sum = 0;
   for (int k = i; k <= j; k++) {
      sum += probabilities[k];
   }
   return sum;
}

// Function to construct optimal binary search tree using dynamic programming
float optimalBST(float probabilities[], int n) {
   float cost[n + 1][n + 1];

   // Initialize cost matrix
   for (int i = 0; i <= n; i++) {
      for (int j = 0; j <= n; j++) {
         cost[i][j] = 0;
      }
   }

   // Initialize diagonal with probabilities
   for (int i = 0; i < n; i++) {
      cost[i][i] = probabilities[i];
   }

   // Calculate minimum cost
   for (int L = 2; L <= n; L++) {
      for (int i = 0; i <= n - L + 1; i++) {
         int j = i + L - 1;
         cost[i][j] = INT_MAX;

         // Try making all keys in the range [i, j] as root
         for (int r = i; r <= j; r++) {
            float c = ((r > i) ? cost[i][r - 1] : 0) +
                  ((r < j) ? cost[r + 1][j] : 0) +
                  sumOfProbabilities(probabilities, i, j);
            if (c < cost[i][j]) {
               cost[i][j] = c;
            }
         }
      }
   }
```

```
    return cost[0][n - 1];
}

int main() {
    float probabilities[] = {0.2, 0.15, 0.1, 0.2, 0.3};
    int n = sizeof(probabilities) / sizeof(probabilities[0]);

    float min_cost = optimalBST(probabilities, n);
    printf("Minimum cost of optimal binary search tree: %.2f\n", min_cost);

    return 0;
}
```

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

```
#include <stdio.h>

#define MAX 10

int total_nodes; // Total nodes in the search tree

// Function to print the subset
void printSubset(int subset[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", subset[i]);
    }
    printf("\n");
}

// Function to generate all subsets of given set
void subsetSum(int s[], int t[], int s_size, int t_size, int sum, int ite, int const target_sum) {
    total_nodes++;

    if (target_sum == sum) {
        // Found subset
        printSubset(t, t_size);

        // Exclude previously added item and consider next candidate
        subsetSum(s, t, s_size, t_size - 1, sum - s[ite], ite + 1, target_sum);
        return;
    } else {
        // Generate nodes along the breadth
        for (int i = ite; i < s_size; i++) {
            t[t_size] = s[i];

            // Consider next level node (along depth)
            subsetSum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
        }
```

```
   }
}

// Function to generate all subsets that sum to the given target
void generateSubsets(int s[], int size, int target_sum) {
   int t[MAX]; // Temporary array to store subset
   subsetSum(s, t, size, 0, 0, 0, target_sum);
}

int main() {
   int weights[] = {10, 7, 5, 18, 12, 20, 15};
   int size = sizeof(weights) / sizeof(weights[0]);

   int target_sum = 35;

   printf("Subsets with sum %d:\n", target_sum);
   generateSubsets(weights, size, target_sum);

   return 0;
}
```

Slip 10

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

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

// Structure for a Huffman tree node
struct MinHeapNode {
   char data;
   unsigned freq;
   struct MinHeapNode *left, *right;
};

// Structure for a min heap
struct MinHeap {
   unsigned size;
   unsigned capacity;
   struct MinHeapNode **array;
};

// Function to create a new MinHeapNode
struct MinHeapNode* newNode(char data, unsigned freq) {
   struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
   temp->left = temp->right = NULL;
   temp->data = data;
   temp->freq = freq;
   return temp;
}

// Function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity) {
   struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
   minHeap->size = 0;
   minHeap->capacity = capacity;
   minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
MinHeapNode*));
   return minHeap;
}

// Function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
   struct MinHeapNode* t = *a;
   *a = *b;
   *b = t;
}

// Function to heapify at given index
void minHeapify(struct MinHeap* minHeap, int idx) {
   int smallest = idx;
   int left = 2 * idx + 1;
   int right = 2 * idx + 2;

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

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

   if (smallest != idx) {
       swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
       minHeapify(minHeap, smallest);
   }
}

// Function to check if size of heap is 1
int isSizeOne(struct MinHeap* minHeap) {
   return (minHeap->size == 1);
}

// Function to extract the minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
   struct MinHeapNode* temp = minHeap->array[0];
   minHeap->array[0] = minHeap->array[minHeap->size - 1];
   --minHeap->size;
   minHeapify(minHeap, 0);
   return temp;
}

// Function to insert a new node to min heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode) {
   ++minHeap->size;
   int i = minHeap->size - 1;
   while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
       minHeap->array[i] = minHeap->array[(i - 1) / 2];
       i = (i - 1) / 2;
   }
   minHeap->array[i] = minHeapNode;
}

// Function to build a min heap
void buildMinHeap(struct MinHeap* minHeap) {
   int n = minHeap->size - 1;
   int i;
   for (i = (n - 1) / 2; i >= 0; --i)
       minHeapify(minHeap, i);
}
```

```
// Function to print an array of size n
void printArr(int arr[], int n) {
   for (int i = 0; i < n; ++i)
      printf("%d", arr[i]);
   printf("\n");
}

// Function to check if node is leaf
int isLeaf(struct MinHeapNode* root) {
   return !(root->left) && !(root->right);
}

// Function to create a min heap of capacity equal to size and inserts all character of data[]
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
   struct MinHeap* minHeap = createMinHeap(size);
   for (int i = 0; i < size; ++i)
      minHeap->array[i] = newNode(data[i], freq[i]);
   minHeap->size = size;
   buildMinHeap(minHeap);
   return minHeap;
}

// Function to build Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
   struct MinHeapNode *left, *right, *top;
   struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

   while (!isSizeOne(minHeap)) {
      left = extractMin(minHeap);
      right = extractMin(minHeap);
      top = newNode('$', left->freq + right->freq);
      top->left = left;
      top->right = right;
      insertMinHeap(minHeap, top);
   }
   return extractMin(minHeap);
}

// Function to print Huffman codes from the root of Huffman Tree
void printCodes(struct MinHeapNode* root, int arr[], int top) {
   if (root->left) {
      arr[top] = 0;
      printCodes(root->left, arr, top + 1);
   }

   if (root->right) {
      arr[top] = 1;
      printCodes(root->right, arr, top + 1);
   }
```

```
    if (isLeaf(root)) {
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

// Function to build Huffman Tree and print codes by traversing the tree
void HuffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
    int arr[MAX], top = 0;
    printCodes(root, arr, top);
}

int main() {
    char data[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(data) / sizeof(data[0]);
    HuffmanCodes(data, freq, size);
    return 0;
}
```

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

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

#define N 4

// Function to print the solution matrix
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}

// Function to check if a queen can be placed at board[row][col]
bool isSafe(int board[N][N], int row, int col) {
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// Recursive function to solve N Queens problem
bool solveNQUtil(int board[N][N], int col) {
    // Base case: If all queens are placed then return true
    if (col >= N)
        return true;

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < N; i++) {
        // Check if the queen can be placed on board[i][col]
        if (isSafe(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col] doesn't lead to a solution then backtrack
            board[i][col] = 0;
        }
    }

    // If the queen cannot be placed in any row in this column, then return false
    return false;
}

// Function to solve the N Queens problem
void solveNQ() {
    int board[N][N] = { {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0} };

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return;
    }
    printSolution(board);
}

int main() {
    solveNQ();
    return 0;
}
```

Slip 11

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

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

#define MAX_VERTICES 100

struct Node {
   int vertex;
   struct Node* next;
};

struct Graph {
   int numVertices;
   struct Node** adjLists;
   bool* visited;
};

// Function to create a new node
struct Node* createNode(int v) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

// Function to create a graph with a given number of vertices
struct Graph* createGraph(int vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numVertices = vertices;

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

   for (int i = 0; i < vertices; i++) {
      graph->adjLists[i] = NULL;
      graph->visited[i] = false;
   }

   return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjLists[src];
```

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

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Depth First Search (DFS) function
void DFS(struct Graph* graph, int vertex) {
    struct Node* adjList = graph->adjLists[vertex];
    struct Node* temp = adjList;

    graph->visited[vertex] = true;
    printf("Visited vertex: %d\n", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->vertex;
        if (!graph->visited[connectedVertex]) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

int main() {
    struct Graph* graph = createGraph(5);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    printf("DFS traversal starting from vertex 0:\n");
    DFS(graph, 0);

    return 0;
}
```

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

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

#define V 6 // Number of vertices in the graph
```

```
// Function to find the vertex with minimum distance value
int minDistance(int dist[], bool sptSet[]) {
   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;
}

// Function to print the constructed distance array
void printSolution(int dist[], int n) {
   printf("Vertex   Distance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d \t\t %d\n", i, dist[i]);
}

// Function to implement Dijkstra's algorithm for a given graph and source vertex
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 the shortest path tree

   // Initialize all distances as INFINITE and sptSet[] 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 it 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, V);
}

int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {{0, 4, 0, 0, 0, 0},
                       {4, 0, 8, 0, 0, 0},
                       {0, 8, 0, 7, 0, 4},
                       {0, 0, 7, 0, 9, 14},
                       {0, 0, 0, 9, 0, 10},
                       {0, 0, 4, 14, 10, 0}};

    // Run Dijkstra's algorithm with source vertex 0
    dijkstra(graph, 0);

    return 0;
}
```

Slip 12

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

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

#define MAX_VERTICES 100

struct Node {
   int vertex;
   struct Node* next;
};

struct Graph {
   int numVertices;
   struct Node** adjLists;
   bool* visited;
};

// Function to create a new node
struct Node* createNode(int v) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

// Function to create a graph with a given number of vertices
struct Graph* createGraph(int vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numVertices = vertices;

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

   for (int i = 0; i < vertices; i++) {
      graph->adjLists[i] = NULL;
      graph->visited[i] = false;
   }

   return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjLists[src];
```

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

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to perform Breadth First Search (BFS) traversal
void BFS(struct Graph* graph, int startVertex) {
    // Initialize a queue for BFS traversal
    int queue[MAX_VERTICES];
    int front = 0, rear = -1;

    // Mark the current vertex as visited and enqueue it
    graph->visited[startVertex] = true;
    queue[++rear] = startVertex;

    while (front <= rear) {
        // Dequeue a vertex from the queue and print it
        int currentVertex = queue[front++];
        printf("Visited vertex: %d\n", currentVertex);

        // Get all adjacent vertices of the dequeued vertex currentVertex
        // If an adjacent vertex has not been visited, mark it as visited and enqueue it
        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!graph->visited[adjVertex]) {
                graph->visited[adjVertex] = true;
                queue[++rear] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

int main() {
    struct Graph* graph = createGraph(6);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    addEdge(graph, 4, 5);

    printf("BFS traversal starting from vertex 0:\n");
    BFS(graph, 0);

    return 0;
}
```

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

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

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

// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;

    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Measure the time taken to execute selection sort
    clock_t start, end;
    double cpu_time_used;

    start = clock(); // Record the starting time

    selectionSort(arr, n);

    end = clock(); // Record the ending time
    // Calculate the time taken
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\nTime taken to sort: %f seconds\n", cpu_time_used);

    return 0;
}
```

Slip 13

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

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

// Function to find the minimum number of multiplications needed for matrix chain multiplication
int matrix_chain_order(int p[], int n) {
    int m[n][n];
    int i, j, k, L, q;

    // Initialize cost matrix with 0
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is the chain length
    for (L = 2; L < n; L++) {
        for (i = 1; i < n - L + 1; i++) {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++) {
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    return m[1][n - 1];
}

int main() {
    int arr[] = {10, 30, 5, 60}; // Matrix dimensions: 10x30, 30x5, 5x60
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Minimum number of multiplications is %d\n", matrix_chain_order(arr, n));
    return 0;
}
```

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

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

// Utility function to find sum of probabilities in a given range
float sum(float freq[], int i, int j) {
    float s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];
    return s;
}

// A utility function to get the optimal cost of a BST rooted with keys[i] through keys[j]
float optimal_cost(float freq[], int i, int j) {
    // If only one element is there in the tree, return its frequency
    if (j < i)
        return 0;
    if (j == i)
        return freq[i];

    // Get sum of frequencies in the current range
    float fsum = sum(freq, i, j);

    float minCost = INT_MAX;

    // One by one consider all elements as root and recursively find the cost of the BST,
    // compare the cost with minCost and update minCost accordingly
    for (int r = i; r <= j; ++r) {
        float cost = optimal_cost(freq, i, r - 1) +
                  optimal_cost(freq, r + 1, j);
        if (cost < minCost)
           minCost = cost;
    }

    // Return the sum of frequencies of elements from i to j plus the minimum cost
    return minCost + fsum;
}

// Function to calculate the optimal cost of a binary search tree
float optimal_bst(float keys[], float freq[], int n) {
    // Call the recursive helper function to find the optimal cost of the BST
    return optimal_cost(freq, 0, n - 1);
}

int main() {
    float keys[] = {10, 12, 20};
    float freq[] = {34, 8, 50};
    int n = sizeof(keys) / sizeof(keys[0]);
    printf("Cost of Optimal BST is %.2f\n", optimal_bst(keys, freq, n));
    return 0;
}
```

Slip 14

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

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

// Function to perform Insertion Sort
void insertion_sort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;

      // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current
position
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}

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

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

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

   clock_t start_time = clock(); // Start measuring time

   insertion_sort(arr, n); // Perform Insertion Sort

   clock_t end_time = clock(); // End measuring time

   printf("Sorted list in ascending order:\n");
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");

   // Calculate the time taken to sort the elements
   double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
```

```
    printf("Time taken to sort %d elements: %f seconds\n", n, time_taken);

    free(arr); // Free dynamically allocated memory

    return 0;
}
```

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

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

#define MAX_VERTICES 100

struct Node {
    int data;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjLists;
    bool* visited;
};

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

// Create a graph
struct Graph* createGraph(int vertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

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

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = false;
    }

    return graph;
}
```

```
// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjLists[src];
   graph->adjLists[src] = newNode;

   newNode = createNode(src);
   newNode->next = graph->adjLists[dest];
   graph->adjLists[dest] = newNode;
}

// DFS
void DFS(struct Graph* graph, int vertex) {
   struct Node* adjList = graph->adjLists[vertex];
   struct Node* temp = adjList;

   graph->visited[vertex] = true;
   printf("Visited %d \n", vertex);

   while (temp != NULL) {
      int connectedVertex = temp->data;
      if (!graph->visited[connectedVertex]) {
         DFS(graph, connectedVertex);
      }
      temp = temp->next;
   }
}

// BFS
void BFS(struct Graph* graph, int startVertex) {
   struct Node* adjList = graph->adjLists[startVertex];
   struct Node* temp = adjList;

   int* queue = (int*)malloc(graph->numVertices * sizeof(int));
   int front = 0, rear = -1;
   bool* visited = (bool*)malloc(graph->numVertices * sizeof(bool));

   for (int i = 0; i < graph->numVertices; i++)
      visited[i] = false;

   visited[startVertex] = true;
   queue[++rear] = startVertex;

   while (front <= rear) {
      int currentVertex = queue[front++];
      printf("Visited %d \n", currentVertex);

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

```
        while (temp) {
            int adjVertex = temp->data;

            if (!visited[adjVertex]) {
                visited[adjVertex] = true;
                queue[++rear] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

// Driver code
int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 5);
    addEdge(graph, 4, 5);

    printf("Depth First Search (DFS):\n");
    DFS(graph, 0);
    printf("\n");

    printf("Breadth First Search (BFS):\n");
    BFS(graph, 0);
    printf("\n");

    return 0;
}
```

Slip 15

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

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

#define MAX 100

struct Item {
   int weight;
   int value;
};

struct Node {
   int level;
   int profit;
   int weight;
   float bound;
};

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

// Function to compare items based on their value per unit weight
int compare(const void* a, const void* b) {
   double r1 = (double)((struct Item*)a)->value / ((struct Item*)a)->weight;
   double r2 = (double)((struct Item*)b)->value / ((struct Item*)b)->weight;
   if (r1 < r2)
      return 1;
   else if (r1 > r2)
      return -1;
   return 0;
}

// Function to calculate bound of a node
float bound(struct Node u, int n, int W, struct Item arr[]) {
   if (u.weight >= W)
      return 0;
   float profit_bound = u.profit;
   int j = u.level + 1;
   int total_weight = u.weight;
   while ((j < n) && (total_weight + arr[j].weight <= W)) {
      total_weight += arr[j].weight;
      profit_bound += arr[j].value;
      j++;
   }
   if (j < n)
```

```
        profit_bound += (W - total_weight) * ((double)arr[j].value / arr[j].weight);
    return profit_bound;
}

// Function to solve 0/1 Knapsack problem using LCBB algorithm
int knapsack(int W, struct Item arr[], int n) {
    qsort(arr, n, sizeof(arr[0]), compare);

    struct Node u, v;
    u.level = -1;
    u.profit = u.weight = 0;
    float maxProfit = 0;
    u.bound = bound(u, n, W, arr);

    int maxProfitIdx = 0;

    int* queue = (int*)malloc(MAX * sizeof(int));
    int front = -1, rear = -1;
    queue[++rear] = 0; // Initialize queue with the index of root node

    while (front != rear) {
        front++;
        u = v = queue[front];

        if (u.level == -1)
            v.level = 0;

        if (u.level == n - 1)
            continue;

        v.level = u.level + 1;
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;

        if (v.weight <= W && v.profit > maxProfit) {
            maxProfit = v.profit;
            maxProfitIdx = v.level;
        }

        v.bound = bound(v, n, W, arr);

        if (v.bound > maxProfit)
            queue[++rear] = v.level;

        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            queue[++rear] = v.level;
    }

    free(queue);
    return maxProfit;
}

int main() {
    int W, n;
    printf("Enter the capacity of knapsack: ");
    scanf("%d", &W);
    printf("Enter the number of items: ");
    scanf("%d", &n);
    struct Item arr[n];
    printf("Enter weight and value of each item:\n");
    for (int i = 0; i < n; i++)
        scanf("%d %d", &arr[i].weight, &arr[i].value);
    int maxProfit = knapsack(W, arr, n);
   printf("Maximum profit: %d\n", maxProfit);
    return 0;
}
```

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

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

#define V 4 // Number of vertices in the graph

// Function to check if the current color assignment is safe for vertex v
bool isSafe(int v, bool graph[V][V], int color[], int c) {
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && c == color[i])
            return false;
    }
    return true;
}

// Function to recursively assign colors to vertices using backtracking
bool graphColoringUtil(bool graph[V][V], int m, int color[], int v) {
    if (v == V)
        return true;

    for (int c = 1; c <= m; c++) {
        if (isSafe(v, graph, color, c)) {
            color[v] = c;

            if (graphColoringUtil(graph, m, color, v + 1))
                return true;
            color[v] = 0;
        }
    }

    return false;
}

// Function to perform graph coloring using greedy approach
void graphColoring(bool graph[V][V], int m) {
    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    if (!graphColoringUtil(graph, m, color, 0)) {
        printf("Graph cannot be colored with %d colors\n", m);
        return;
    }

    printf("Graph can be colored with %d colors:\n", m);
    for (int i = 0; i < V; i++)
        printf("Vertex %d: Color %d\n", i, color[i]);
}

int main() {
    bool graph[V][V] = {{0, 1, 1, 1},
                        {1, 0, 1, 0},
                        {1, 1, 0, 1},
                        {1, 0, 1, 0}};

    int m = 3; // Number of colors

    graphColoring(graph, m);

    return 0;
}
```

Slip 16

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

```
#include <stdio.h>

// Function to find the maximum of two integers
int max(int a, int b) {
   return (a > b) ? a : b;
}

// Function to solve 0/1 knapsack problem using dynamic programming
int knapsack(int W, int wt[], int val[], int n) {
   int i, w;
   int K[n + 1][W + 1];

   // Build table K[][] in bottom-up manner
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            K[i][w] = 0;
         else if (wt[i - 1] <= w)
            K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
         else
            K[i][w] = K[i - 1][w];
      }
   }

   // The maximum value that can be obtained with weight W
   return K[n][W];
}

int main() {
   int val[] = {60, 100, 120};
   int wt[] = {10, 20, 30};
   int W = 50;
   int n = sizeof(val) / sizeof(val[0]);
   printf("Maximum value that can be obtained: %d\n", knapsack(W, wt, val, n));
   return 0;
}
```

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

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

#define V 5 // Number of vertices

bool isSafe(int v, bool graph[V][V], int path[], int pos) {
    // Check if this vertex is an adjacent vertex of the previously added vertex
    if (!graph[path[pos - 1]][v])
        return false;

    // Check if the vertex has already been included
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

bool hamCycleUtil(bool graph[V][V], int path[], int pos) {
    // If all vertices are included in Hamiltonian Cycle
    if (pos == V) {
        // And if there is an edge from the last included vertex to the first vertex
        if (graph[path[pos - 1]][path[0]])
            return true;
        else
            return false;
    }

    // Try different vertices as the next candidate in Hamiltonian Cycle
    for (int v = 1; v < V; v++) {
        if (isSafe(v, graph, path, pos)) {
            path[pos] = v;
            if (hamCycleUtil(graph, path, pos + 1) == true)
                return true;
            // Remove vertex if it doesn't lead to a solution
            path[pos] = -1;
        }
    }

    return false;
}

bool hamCycle(bool graph[V][V]) {
    int path[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    // Add the first vertex to the path
    path[0] = 0;

    return hamCycleUtil(graph, path, 1);
}

int main() {
    bool graph1[V][V] = {{0, 1, 0, 1, 0},
                    {1, 0, 1, 1, 1},
                    {0, 1, 0, 0, 1},
                    {1, 1, 0, 0, 1},
                    {0, 1, 1, 1, 0}};

    if (hamCycle(graph1))
        printf("Graph has a Hamiltonian cycle\n");
    else
        printf("Graph does not have a Hamiltonian cycle\n");

    bool graph2[V][V] = {{0, 1, 0, 1, 0},
                    {1, 0, 1, 1, 1},
                    {0, 1, 0, 0, 1},
                    {1, 1, 0, 0, 0},
                    {0, 1, 1, 0, 0}};

    if (hamCycle(graph2))
        printf("Graph has a Hamiltonian cycle\n");
    else
        printf("Graph does not have a Hamiltonian cycle\n");

    return 0;
}
```

Slip 17

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

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

#define N 4 // Number of Queens

void printSolution(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         printf(" %d ", board[i][j]);
      printf("\n");
   }
}

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

   // Check this row on left side
   for (i = 0; i < col; i++)
      if (board[row][i])
         return false;

   // Check upper diagonal on left side
   for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
      if (board[i][j])
         return false;

   // Check lower diagonal on left side
   for (i = row, j = col; j >= 0 && i < N; i++, j--)
      if (board[i][j])
         return false;

   return true;
}

bool solveNQUtil(int board[N][N], int col) {
   // Base case: If all queens are placed then return true
   if (col >= N)
      return true;

   // Consider this column and try placing this queen in all rows one by one
   for (int i = 0; i < N; i++) {
      // Check if the queen can be placed on board[i][col]
      if (isSafe(board, i, col)) {
         // Place this queen in board[i][col]
         board[i][col] = 1;
```

```
            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col] doesn't lead to a solution, then remove the queen from
board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in this column col then return false
    return false;
}

bool solveNQ() {
    int board[N][N] = {{0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0}};

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

int main() {
    solveNQ();
    return 0;
}
```

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

```
#include <stdio.h>

// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve 0/1 knapsack problem using dynamic programming
int knapsack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom-up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }

    // The maximum value that can be obtained with weight W
    return K[n][W];
}

int main() {
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    printf("Maximum value that can be obtained: %d\n", knapsack(W, wt, val, n));
    return 0;
}
```

Slip 18

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

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

#define V 4 // Number of vertices

// Function to check if the current color assignment is safe for vertex v
int isSafe(int v, int graph[V][V], int color[], int c) {
   for (int i = 0; i < V; i++) {
      if (graph[v][i] && c == color[i])
         return 0;
   }
   return 1;
}

// Function to assign colors to vertices
void graphColoring(int graph[V][V], int m) {
   int color[V];
   for (int i = 0; i < V; i++)
      color[i] = 0;

   // Assign the first color to the first vertex
   color[0] = 1;

   // Assign colors to remaining V-1 vertices
   for (int v = 1; v < V; v++) {
      // Initialize the color of vertex v as unavailable
      color[v] = 0;

      // Process all adjacent vertices of v and flag their colors as unavailable
      for (int c = 1; c <= m; c++) {
         if (isSafe(v, graph, color, c)) {
            color[v] = c;
            break;
         }
      }
   }

   // Print the assigned colors
   printf("Vertex   Color\n");
   for (int i = 0; i < V; i++)
      printf("  %d       %d\n", i, color[i]);
}

int main() {
   // Adjacency matrix representation of the graph
   int graph[V][V] = {{0, 1, 1, 1},
                   {1, 0, 1, 0},
                   {1, 1, 0, 1},
                   {1, 0, 1, 0}};

   int m = 3; // Number of colors

   graphColoring(graph, m);

   return 0;
}
```

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

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

#define V 6 // Number of vertices

// Function to perform depth-first search (DFS) traversal
void DFS(int graph[V][V], int v, bool visited[]) {
    visited[v] = true;
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && !visited[i])
            DFS(graph, i, visited);
    }
}

// Function to find live, E-node, and dead nodes in the graph
void findNodes(int graph[V][V]) {
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Perform DFS traversal to find reachable nodes
    DFS(graph, 0, visited);

    printf("Live nodes: ");
    for (int i = 0; i < V; i++) {
        if (visited[i])
            printf("%d ", i);
    }
    printf("\n");

    printf("E-nodes: ");
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            bool eNode = true;
            for (int j = 0; j < V; j++) {
                if (graph[j][i] && visited[j]) {
```

```
                eNode = false;
                break;
            }
        }
        if (eNode)
            printf("%d ", i);
    }
  }
  printf("\n");

  printf("Dead nodes: ");
  for (int i = 0; i < V; i++) {
    if (!visited[i]) {
        bool deadNode = true;
        for (int j = 0; j < V; j++) {
            if (graph[i][j] && visited[j]) {
                deadNode = false;
                break;
            }
        }
        if (deadNode)
            printf("%d ", i);
    }
  }
  printf("\n");
}

int main() {
  int graph[V][V] = {
    {0, 1, 0, 1, 0, 0},
    {0, 0, 1, 0, 0, 0},
    {0, 0, 0, 1, 1, 0},
    {0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0}
  };

  findNodes(graph);

  return 0;
}
```

Slip 19

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

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

#define V 5 // Number of vertices

bool isSafe(int v, bool graph[V][V], int path[], int pos) {
   // Check if this vertex is an adjacent vertex of the previously added vertex
   if (!graph[path[pos - 1]][v])
      return false;

   // Check if the vertex has already been included
   for (int i = 0; i < pos; i++)
      if (path[i] == v)
         return false;

   return true;
}

bool hamCycleUtil(bool graph[V][V], int path[], int pos) {
   // Base case: If all vertices are included in Hamiltonian Cycle
   if (pos == V) {
      // And if there is an edge from the last included vertex to the first vertex
      if (graph[path[pos - 1]][path[0]])
         return true;
      else
         return false;
   }

   // Try different vertices as the next candidate in Hamiltonian Cycle
   for (int v = 1; v < V; v++) {
      if (isSafe(v, graph, path, pos)) {
         path[pos] = v;
         if (hamCycleUtil(graph, path, pos + 1))
            return true;
         // Remove vertex if it doesn't lead to a solution
         path[pos] = -1;
      }
   }

   return false;
}

bool hamCycle(bool graph[V][V]) {
   int path[V];
   for (int i = 0; i < V; i++)
      path[i] = -1;
```

```
    // Add the first vertex to the path
    path[0] = 0;

    return hamCycleUtil(graph, path, 1);
}

int main() {
    // Adjacency matrix representation of the graph
    bool graph[V][V] = {{0, 1, 0, 1, 0},
                        {1, 0, 1, 1, 1},
                        {0, 1, 0, 0, 1},
                        {1, 1, 0, 0, 1},
                        {0, 1, 1, 1, 0}};

    if (hamCycle(graph))
        printf("Graph has a Hamiltonian cycle\n");
    else
        printf("Graph does not have a Hamiltonian cycle\n");

    return 0;
}
```

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

```
#include <stdio.h>

#define N 4

void printBoard(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
}

// Function to check if a queen can be placed at board[row][col]
int isSafe(int board[N][N], int row, int col) {
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return 0;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
```

```
            return 0;

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return 0;

    return 1;
}

// Recursive function to solve N Queens problem
int solveNQUtil(int board[N][N], int col) {
    // If all queens are placed then return true
    if (col >= N)
        return 1;

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < N; i++) {
        // Check if the queen can be placed on board[i][col]
        if (isSafe(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return 1;

            // If placing queen in board[i][col] doesn't lead to a solution, then remove the queen from
board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    return 0;
}

// Function to solve N Queens problem and print board configuration
void solveNQ() {
    int board[N][N] = {{0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0}};

    if (solveNQUtil(board, 0) == 0) {
        printf("Solution does not exist");
        return;
    }

    printBoard(board);
```

```
}

int main() {
   solveNQ();
   return 0;
}
```

Slip 20

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

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

#define MAX_VERTICES 100

struct Node {
   int data;
   struct Node* next;
};

struct Graph {
   int numVertices;
   struct Node** adjLists;
   bool* visited;
};

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}

struct Graph* createGraph(int numVertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numVertices = numVertices;
   graph->adjLists = (struct Node**)malloc(numVertices * sizeof(struct Node*));
   graph->visited = (bool*)malloc(numVertices * sizeof(bool));

   for (int i = 0; i < numVertices; i++) {
      graph->adjLists[i] = NULL;
      graph->visited[i] = false;
   }

   return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
   struct Node* newNode = createNode(dest);
   newNode->next = graph->adjLists[src];
   graph->adjLists[src] = newNode;
}

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

   struct Node* temp = graph->adjLists[v];
   while (temp != NULL) {
      int adjVertex = temp->data;
      if (!visited[adjVertex])
          topologicalSortUtil(graph, adjVertex, visited, stack);
      temp = temp->next;
   }

   // Push current vertex to stack which stores result
   struct Node* newNode = createNode(v);
   newNode->next = *stack;
   *stack = newNode;
}

void topologicalSort(struct Graph* graph) {
   struct Node* stack = NULL;

   bool* visited = (bool*)malloc(graph->numVertices * sizeof(bool));
   for (int i = 0; i < graph->numVertices; i++)
      visited[i] = false;

   for (int i = 0; i < graph->numVertices; i++) {
      if (!visited[i])
          topologicalSortUtil(graph, i, visited, &stack);
   }

   // Print contents of the stack
   printf("Topological Sorting: ");
   while (stack != NULL) {
      printf("%d ", stack->data);
      stack = stack->next;
   }
}

int main() {
   struct Graph* graph = createGraph(6);
   addEdge(graph, 5, 2);
   addEdge(graph, 5, 0);
   addEdge(graph, 4, 0);
   addEdge(graph, 4, 1);
   addEdge(graph, 2, 3);
   addEdge(graph, 3, 1);

   topologicalSort(graph);

   return 0;
}
```

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

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

#define N 4 // Number of queens

void printSolution(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         printf(" %d ", board[i][j]);
      printf("\n");
   }
}

// Function to check if a queen can be placed at board[row][col]
bool isSafe(int board[N][N], int row, int col) {
   int i, j;

   // Check this row on left side
   for (i = 0; i < col; i++)
      if (board[row][i])
         return false;

   // Check upper diagonal on left side
   for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
      if (board[i][j])
         return false;

   // Check lower diagonal on left side
   for (i = row, j = col; j >= 0 && i < N; i++, j--)
      if (board[i][j])
         return false;

   return true;
}

// Recursive function to solve N Queens problem
bool solveNQUtil(int board[N][N], int col) {
   // Base case: If all queens are placed then return true
   if (col >= N)
      return true;

   // Consider this column and try placing this queen in all rows one by one
   for (int i = 0; i < N; i++) {
      // Check if the queen can be placed on board[i][col]
      if (isSafe(board, i, col)) {
         // Place this queen in board[i][col]
```

```
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col] doesn't lead to a solution, then remove the queen from
board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in this column col then return false
    return false;
}

// Function to solve N Queens problem and print board configuration
bool solveNQ() {
    int board[N][N] = {{0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0},
                       {0, 0, 0, 0}};

    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

int main() {
    solveNQ();
    return 0;
}
```