program 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

program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void selectionSort(int arr[], int n) {
   int i, j, minIndex, temp;
   for (i = 0; i < n - 1; i++) {
      minIndex = i;
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex]) {
            minIndex = j;
         }
      }
      // Swap the found minimum element with the first element
      temp = arr[minIndex];
      arr[minIndex] = arr[i];
      arr[i] = temp;
   }
}

int main() {
   int n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &arr[i]);
   }
   clock_t start, end;
   double cpu_time_used;
   start = clock();
   selectionSort(arr, n);
   end = clock();
   cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
   printf("Sorted elements: ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\nTime taken: %f seconds\n", cpu_time_used);
   return 0;
}
```

program 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.

program

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


// Function prototypes

void quickSort(int arr[], int low, int high);

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

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

double time_elapsed(struct timespec start, struct timespec end);


int main() {

    int n, i;

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

    scanf("%d", &n);


    int arr[n];


    // Generate array with random numbers

    srand(time(0));

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

        arr[i] = rand() % 1000; // Generating random numbers between 0 and 999

    }


    // Start measuring time

    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC, &start);
    // Sort the array using Quick Sort
    quickSort(arr, 0, n - 1);

    // End measuring time
    clock_gettime(CLOCK_MONOTONIC, &end);

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

    printf("Time taken by Quick Sort: %lf seconds\n", time_elapsed(start, end));

    return 0;
}
// Quick Sort function
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);
    }
}

// Partition function for Quick Sort
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

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

// Utility function to calculate time elapsed
double time_elapsed(struct timespec start, struct timespec end) {
    double t;
    t = (end.tv_sec - start.tv_sec) * 1e9;
    t = (t + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    return t;
}
```

program 3

Write a program to sort n randomly generated elements using Heapsort method

program

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

// Function to heapify a subtree rooted with node i which is
// an index in arr[]. n is the size of the heap
void heapify(int arr[], int n, int i) {
   int largest = i; // Initialize largest as root
   int l = 2 * i + 1; // left = 2*i + 1
   int r = 2 * i + 2; // right = 2*i + 2

   // If left child is larger than root
   if (l < n && arr[l] > arr[largest])
      largest = l;

   // If right child is larger than largest so far
   if (r < n && arr[r] > arr[largest])
      largest = r;

   // If largest is not root
   if (largest != i) {
      int temp = arr[i];
      arr[i] = arr[largest];
      arr[largest] = temp;
      // 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
      int temp = arr[0];
      arr[0] = arr[i];
      arr[i] = temp;

      // call max heapify on the reduced heap
      heapify(arr, i, 0);
```

```
    }
}

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

    // Dynamically allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));

    // Seed for random number generation
    srand(time(NULL));

    // Generate random numbers and fill the array
    printf("Randomly generated array: ");
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100; // Generate numbers between 0 and 99
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Perform heap sort
    heapSort(arr, n);

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

    // Free dynamically allocated memory
    free(arr);

    return 0;
}
```

program 4

Write a program to implement Strassen's Matrix multiplication

program

```
#include <stdio.h>

#include <stdlib.h>

// Function prototypes

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

void add(int n, int A[][n], int B[][n], int C[][n]);

void subtract(int n, int A[][n], int B[][n], int C[][n]);

void printMatrix(int n, int matrix[][n]);

int main() {

   int n, i, j;

   printf("Enter the size of square matrices: ");

   scanf("%d", &n);


   // Initialize matrices A and B

   int A[n][n], B[n][n], C[n][n];


   // Input matrices A and B

   printf("Enter elements of matrix A:\n");

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

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

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

      }

   }


   printf("Enter elements of matrix B:\n");

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

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

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

      }
```

```
    }

    // Multiply matrices A and B using Strassen's algorithm
    strassen(n, A, B, C);

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

    return 0;
}

// Strassen's Matrix Multiplication 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 i, j;
    int newSize = n / 2;

    // Define submatrices
    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 (i = 0; i < newSize; i++) {
    for (j = 0; j < newSize; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + newSize];
        A21[i][j] = A[i + newSize][j];
        A22[i][j] = A[i + newSize][j + newSize];

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

// 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 submatrices of C
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 to form result matrix C
for (i = 0; i < newSize; i++) {
  for (j = 0; j < newSize; j++) {
    C[i][j] = C11[i][j];
    C[i][j + newSize] = C12[i][j];
    C[i + newSize][j] = C21[i][j];
    C[i + newSize][j + newSize] = C22[i][j];
```

```
        }
    }
}

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

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

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

program 5

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

program

```
#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 perform quicksort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

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

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

    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();

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

    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken to sort: %f seconds\n", time_spent);

    return 0;
}
```

program 6
Write a program to find Minimum Cost Spanning Tree of a given undirected graph using Prims algorithm
program

```
#include <stdio.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 not yet 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);
}

// Driver program to test above functions
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 solution
    primMST(graph);
    return 0;
}
```

program 7

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

program

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

// Merge two subarrays of arr[]
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
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;
```

```
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

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

// Merge sort function
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Find the middle point
        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);
    // Allocate memory for the array
    int *arr = (int *)malloc(n * sizeof(int));

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

    // Record start time
    clock_t start = clock();

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

    // Record end time
    clock_t end = clock();

    // Calculate the time taken
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

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

    // Output the time taken
    printf("Time taken: %f seconds\n", time_taken);

    // Free dynamically allocated memory
    free(arr);

    return 0;
}
```

program 8
Write a program to implement Knapsack problems using Greedy method
program

```
#include <stdio.h>


#define MAX_ITEMS 100


// Structure to represent an item
```

```
struct Item {
   int value;
   int weight;
};

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

// Function to solve the knapsack problem using a greedy method
void knapsackGreedy(int capacity, struct Item items[], int n) {
   // Sort items based on value-to-weight ratio
   qsort(items, n, sizeof(struct Item), compare);

   int currentWeight = 0; // Current weight in the knapsack
   double totalValue = 0.0; // Total value of items selected

   printf("Selected items:\n");

   for (int i = 0; i < n; i++) {
      // If adding the current item doesn't exceed the capacity
      if (currentWeight + items[i].weight <= capacity) {
         // Select the current item
         printf("Item %d - Value: %d, Weight: %d\n", i + 1, items[i].value, items[i].weight);
         currentWeight += items[i].weight;
         totalValue += items[i].value;
      } else {
            // Take a fraction of the current item to fill the knapsack
            int remainingCapacity = capacity - currentWeight;
            totalValue += items[i].value * ((double)remainingCapacity / items[i].weight);
            printf("Item %d - Value: %d, Weight: %d (Fraction: %.2lf)\n", i + 1, items[i].value, items[i].weight,
((double)remainingCapacity / items[i].weight));
            break;
        }
    }

    printf("Total Value: %.2lf\n", totalValue);
}

int main() {
    int capacity, n;

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

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

    struct Item items[MAX_ITEMS];

    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", &items[i].value, &items[i].weight);
    }

    knapsackGreedy(capacity, items, n);
    return 0;
}
```

program 9

Write a program for the Implementation of Kruskal's algorithm to find minimum cost spanning tree

program

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

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

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

// Function prototypes
int find(struct Subset subsets[], int i);
void Union(struct Subset subsets[], int x, int y);
int myComp(const void* a, const void* b);
void KruskalMST(struct Edge* edges, int V, int E);

// Function to find the subset of an element i
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 that performs union of two subsets
void Union(struct Subset subsets[], int x, int y) {
   int xroot = find(subsets, x);
   int yroot = find(subsets, y);

   if (subsets[xroot].rank < subsets[yroot].rank)
      subsets[xroot].parent = yroot;
   else if (subsets[xroot].rank > subsets[yroot].rank)
      subsets[yroot].parent = xroot;
   else {
      subsets[yroot].parent = xroot;
      subsets[xroot].rank++;
   }
}

// Comparator function used by qsort() to sort edges based on weight
int myComp(const void* a, const void* b) {
   struct Edge* a1 = (struct Edge*)a;
   struct Edge* b1 = (struct Edge*)b;
   return a1->weight > b1->weight;
}

// Function to construct MST using Kruskal's algorithm
void KruskalMST(struct Edge* edges, int V, int E) {
   struct Edge result[V];  // To store the resultant MST
   int e = 0;              // Index used to pick the next edge

   qsort(edges, E, sizeof(edges[0]), myComp);

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

   for (int v = 0; v < V; v++) {
      subsets[v].parent = v;
      subsets[v].rank = 0;
   }

   while (e < V - 1 && E > 0) {
      struct Edge next_edge = edges[E - 1];
      E--;

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

      if (x != y) {
          result[e++] = next_edge;
          Union(subsets, x, y);
      }
   }

   printf("Edges of constructed MST:\n");
   for (int i = 0; i < e; i++)
      printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);

   free(subsets);
}

int main() {
   int V = 4;  // Number of vertices in graph
   int E = 5;  // Number of edges in graph
   struct Edge edges[] = { {0, 1, 10}, {0, 2, 6}, {0, 3, 5}, {1, 3, 15}, {2, 3, 4} };

   KruskalMST(edges, V, E);

   return 0;
}
```

program 10

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

program

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_TREE_HT 100


// A Huffman tree node

struct MinHeapNode {

   char data;

   unsigned freq;

   struct MinHeapNode *left, *right;

};


// A Min Heap: Collection of min-heap (or Huffman tree) nodes

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;
```

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

// Function to check if this 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 insert all characters of data[] in min heap.
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 Huffman Codes
void huffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode *root = buildHuffmanTree(data, freq, size);
    int arr[MAX_TREE_HT], top = 0;
    printCodes(root, arr, top);
}

// Best Case Complexity: O(nlogn)
// Worst Case Complexity: O(nlogn)
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;
}
```

program 11

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

program

```
#include <stdio.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];
  int key[V];
  bool mstSet[V];

  for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = false;

  key[0] = 0;
  parent[0] = -1; // First node is always root of MST

  for (int count = 0; count < V - 1; count++) {
    int u = minKey(key, mstSet);
    mstSet[u] = true;

    for (int v = 0; v < V; v++)
      if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
  }

  printMST(parent, graph);
}
```

```
int main() {
   int graph[V][V] = {
      {0, 2, 0, 6, 0},
      {2, 0, 3, 8, 5},
      {0, 3, 0, 0, 7},
      {6, 8, 0, 0, 9},
      {0, 5, 7, 9, 0}
   };

   primMST(graph);

   return 0;
}
```

program 12

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

program

```
#include<stdio.h>

#include<string.h>


// Function to find length of Longest Common Subsequence (LCS)

int lcsLength(char *X, char *Y, int m, int n) {

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

   int i, j;


   // Building the L[m+1][n+1] matrix in bottom-up manner

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

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

         if (i == 0 || j == 0)

            L[i][j] = 0;

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

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

         else

            L[i][j] = (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 m = strlen(X);

   int n = strlen(Y);

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

   return 0;

}
```

program 13

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

program

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

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

// Function to find the vertex with minimum distance value
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[]) {
   printf("Vertex \t Distance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d \t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm
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.
      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 it's not in sptSet, there is an edge from u to v, and total weight of path
from src to v through u is smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // Print the constructed distance array
    printSolution(dist);
}

// Driver program to test above functions
int main() {
    // Graph representation in adjacency matrix form
    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); // Find shortest paths from vertex 0

    return 0;
}
```

program 14

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

program

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 100


// Structure to represent a graph node

struct Node {

   int data;

   struct Node* next;

};


// Structure to represent a graph

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;

};


// Function to create a new graph node

struct Node* createNode(int data) {

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

   newNode->data = data;

   newNode->next = NULL;

   return newNode;

}


// Function to create a graph with a given number of vertices

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 = (int*)malloc(numVertices * sizeof(int));

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

   return graph;
}

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

// Depth-First Search
void DFS(struct Graph* graph, int vertex, int* visited, struct Node** stack) {
   visited[vertex] = 1;
   struct Node* adjList = graph->adjLists[vertex];
   struct Node* temp = adjList;

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

   // Push the current vertex to the stack after all its neighbors are visited
   *stack = createNode(vertex);
   (*stack)->next = adjList;
}

// Function to perform topological sorting
void topologicalSort(struct Graph* graph) {
   int* visited = (int*)malloc(graph->numVertices * sizeof(int));
   struct Node* stack = NULL;

   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);
       }
   }

   // Print the vertices in the stack to get the topological sorting
   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;
}
```

program 15

Write a program to implement Fractional Knapsack problems using Greedy Method

program

```
#include <stdio.h>

struct Item {
   int value;
   int weight;
   float ratio;
};

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

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]);
         }
      }
   }
}

float knapsack(int capacity, struct Item items[], int n) {
   sortItems(items, n);
   float totalValue = 0.0;
   int remainingCapacity = capacity;

   for (int i = 0; i < n; i++) {
      if (items[i].weight <= remainingCapacity) {
         totalValue += items[i].value;
         remainingCapacity -= items[i].weight;
      } else {
         totalValue += (float)items[i].value * remainingCapacity / items[i].weight;
         break;
      }
   }
   return totalValue;
}

int main() {
   int n, capacity;
   printf("Enter the number of items: ");
```

```
    scanf("%d", &n);
    struct Item items[n];
    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

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

    float maxValue = knapsack(capacity, items, n);
    printf("Maximum value in the knapsack: %.2f\n", maxValue);

    return 0;
}
```

program 16

Write Program to implement Traveling Salesman Problem using nearest neighbor algorithm

program

```
#include <stdio.h>

#include <limits.h>


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


int graph[V][V] = {

   {0, 10, 15, 20},

   {10, 0, 35, 25},

   {15, 35, 0, 30},

   {20, 25, 30, 0}

};


int visited[V];


int findNearestNeighbor(int current) {

   int minDistance = INT_MAX;

   int nearestNeighbor = -1;


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

      if (!visited[i] && graph[current][i] < minDistance) {

         minDistance = graph[current][i];

         nearestNeighbor = i;

      }

   }


   return nearestNeighbor;

}
```

```
void travelSalesman() {
   visited[0] = 1; // Starting from the first city

   printf("Path: 1 ");

   for (int i = 1; i < V; i++) {
      int nearestNeighbor = findNearestNeighbor(i - 1);
      visited[nearestNeighbor] = 1;
      printf("-> %d ", nearestNeighbor + 1);
   }

   printf("-> 1\n"); // Return to the starting city to complete the cycle
}

int main() {
   // Initialize visited array
   for (int i = 0; i < V; i++) {
      visited[i] = 0;
   }

   printf("Optimal TSP Path:\n");
   travelSalesman();

   return 0;
}
```

program 17

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

program

```
#include <stdio.h>

#include <limits.h>


// Function to calculate the optimal binary search tree

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

   int cost[n][n];


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

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

   }


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

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

         int j = i + L - 1;

         cost[i][j] = INT_MAX;


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

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

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

                  sum(freq, i, j);


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

               cost[i][j] = c;

            }

         }

      }

   }
```

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

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

int main() {
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys) / sizeof(keys[0]);

    int result = optimalBST(keys, freq, n);

    printf("Cost of Optimal Binary Search Tree is: %d\n", result);

    return 0;
}
```

program 18

Write a program to implement Sum of Subset by Backtracking

program

```
#include <stdio.h>


#define MAX 10


int subset[MAX];

int n, sum;


void displaySubset() {

   printf("Subset: { ");

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

      if (subset[i] == 1) {

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

      }

   }

   printf("}\n");

}


void generateSubset(int k, int currentSum, int remainingSum) {

   subset[k] = 1; // Include current element


   if (currentSum + k + 1 == sum) {

      displaySubset();

   } else if (currentSum + k + 1 < sum) {

      generateSubset(k + 1, currentSum + k + 1, remainingSum - (k + 1));

   }


   subset[k] = 0; // Exclude current element
```

```
    if (currentSum + remainingSum - k >= sum) {
        generateSubset(k + 1, currentSum, remainingSum - k);
    }
}

void subsetSum() {
    int totalSum = (n * (n + 1)) / 2; // Sum of all elements in the set

    if (totalSum % 2 != 0) {
        printf("No subset with given sum exists.\n");
        return;
    }

    sum = totalSum / 2;

    printf("Subsets with sum %d are:\n", sum);
    generateSubset(0, 0, totalSum);
}

int main() {
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    if (n <= 0 || n > MAX) {
        printf("Invalid input for the number of elements.\n");
        return 1;
    }
    subsetSum();

    return 0;
}
```

program 19

Write a program to implement Huffman Code using greedy methods

program

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

// Define the structure for a Huffman Tree Node
struct MinHeapNode {
    char data; // character
    unsigned freq; // frequency of the character
    struct MinHeapNode *left, *right; // left and right child pointers
};

// Structure for a Min Heap
struct MinHeap {
    unsigned size; // current size of heap
    unsigned capacity; // capacity of heap
    struct MinHeapNode **array; // array of min heap node pointers
};

// 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 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 build 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]));
    buildMinHeap(minHeap);
    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
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 encode input data using Huffman Codes
void HuffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
    int arr[100], top = 0;
    printCodes(root, arr, top);
}

// Driver function
int main() {
    char data[] = {'a', 'b', 'c', 'd',
```

program 20

Write a program to solve 4 Queens Problem using Backtracking

program

```
#include <stdio.h>

#include <stdbool.h>


#define N 4


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

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

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

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

        }

        printf("\n");

    }

}


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

    // Check this row on the left side

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

        if (board[row][i])

            return false;

    }


    // Check upper diagonal on the left side

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

        if (board[i][j])

            return false;

    }


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

    return true;
}

bool solveNQueensUtil(int board[N][N], int col) {
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;

            if (solveNQueensUtil(board, col + 1))
                return true;

            board[i][col] = 0; // backtrack
        }
    }

    return false;
}

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

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

    printSolution(board);
    return true;
}

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

program 21

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

program

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 100


// Structure to represent a graph node

struct Node {

   int data;

   struct Node* next;

};


// Structure to represent a graph

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;

};


// Function to create a new graph node

struct Node* createNode(int data) {

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

   newNode->data = data;

   newNode->next = NULL;

   return newNode;

}


// Function to create a graph with a given number of vertices
```

```
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 = (int*)malloc(numVertices * sizeof(int));

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

   return graph;
}

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

   // Add edge from dest to src
   newNode = createNode(src);
   newNode->next = graph->adjLists[dest];
   graph->adjLists[dest] = newNode;
}

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

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL) {
        int connectedVertex = temp->data;
        if (graph->visited[connectedVertex] == 0) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

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

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

    printf("Depth-First Search starting from vertex 0: \n");
    DFS(graph, 0);

    return 0;
}
```

program 22

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

program

```
#include <stdio.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 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[]) {

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

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

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

}


// Function to implement Dijkstra's algorithm for a graph represented using

// adjacency matrix representation
```

```
void dijkstra(int graph[V][V], int src) {
   int dist[V]; // The output array. dist[i] will hold the shortest
             // distance from src to i

   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);
}

// Driver program to test above function
int main() {
    // Example graph representation using adjacency matrix
    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;
}
```

program 23

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

program

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 100


// Structure to represent a graph node

struct Node {

   int data;

   struct Node* next;

};


// Structure to represent a graph

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;

};


// Function to create a new graph node

struct Node* createNode(int data) {

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

   newNode->data = data;

   newNode->next = NULL;

   return newNode;

}


// Function to create a graph with a given number of vertices
```

```
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 = (int*)malloc(numVertices * sizeof(int));

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

   return graph;
}

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

   // Add edge from dest to src
   newNode = createNode(src);
   newNode->next = graph->adjLists[dest];
   graph->adjLists[dest] = newNode;
}

// Breadth-First Search
void BFS(struct Graph* graph, int startVertex) {
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;

    graph->visited[startVertex] = 1;
    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 (graph->visited[adjVertex] == 0) {
                graph->visited[adjVertex] = 1;
                queue[rear++] = adjVertex;
            }
            temp = temp->next;
        }
    }
}
int main() {
    struct Graph* graph = createGraph(5);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);

    printf("Breadth-First Search starting from vertex 0: \n");
    BFS(graph, 0);    return 0;
}
```

program 24

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

program

```
#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;
  }
}

int main() {
  int n, i;
  clock_t start, end;
  double cpu_time_used;

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

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

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

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

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

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

  printf("Time taken to sort: %lf seconds\n", cpu_time_used);

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

program 25

Write a program to find minimum number of multiplications in Matrix Chain Multiplication

program

```
#include<stdio.h>

#include<limits.h>


// Function to find minimum number of multiplications

// needed to multiply a chain of matrices.

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

  /* For simplicity of the program, one extra row and one

    extra column are allocated in m[][].  0th row and 0th

    column of m[][] are not used */

  int m[n][n];


  int i, j, k, L, q;


  /* m[i,j] = Minimum number of scalar multiplications needed

    to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where

    dimension of A[i] is p[i-1] x p[i] */


  // cost is zero when multiplying one matrix.

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

    m[i][i] = 0;


  // L is 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 = cost/scalar multiplications
```

```
                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[] = {1, 2, 3, 4}; // Matrix sizes: 1x2, 2x3, 3x4
    int n = sizeof(arr) / sizeof(arr[0]);

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

    return 0;
}
```

program 26

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

program

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

// Structure to represent a node in the graph
struct Node {
   int data;
   struct Node* next;
};

// Structure to represent the graph
struct Graph {
   int numVertices;
   struct Node** adjLists;
};

// Function to 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;
}

// Function to create a graph with given 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*));

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

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

   // Since the graph is undirected, add an edge from dest to src as well
   newNode = createNode(src);
```

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

// Function to print the graph
void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("Vertex %d: ", v);
        while (temp) {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

// Function to perform Depth First Search (DFS)
void DFS(struct Graph* graph, int vertex, int visited[]) {
    visited[vertex] = 1;
    printf("%d ", vertex);

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

// Function to perform Breadth First Search (BFS)
void BFS(struct Graph* graph, int startVertex) {
    int visited[graph->numVertices];
    for (int i = 0; i < graph->numVertices; i++) {
        visited[i] = 0;
    }

    // Create a queue for BFS
    int queue[graph->numVertices];
    int front = 0, rear = 0;

    visited[startVertex] = 1;
    queue[rear++] = startVertex;

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

        struct Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->data;
            if (!visited[adjVertex]) {
                visited[adjVertex] = 1;
                queue[rear++] = adjVertex;
            }
            temp = temp->next;
        }
    }
}

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

    printf("Graph:\n");
    printGraph(graph);
    printf("\nDFS traversal: ");
    int visitedDFS[graph->numVertices];
    for (int i = 0; i < graph->numVertices; i++) {
        visitedDFS[i] = 0;
    }
    DFS(graph, 0, visitedDFS);
    printf("\nBFS traversal: ");
    BFS(graph, 0);

    return 0;
}
```

program 27

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

program

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

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

// Function to compare items based on their value/weight ratio
int compare(const void *a, const void *b) {
  double ratio1 = (double)(((struct Item *)a)->value) / (((struct Item *)a)->weight);
  double ratio2 = (double)(((struct Item *)b)->value) / (((struct Item *)b)->weight);
  if (ratio1 > ratio2)
    return -1;
  else if (ratio1 < ratio2)
    return 1;
  else
    return 0;
}

// Function to solve 0/1 knapsack using LCBB
void knapsackLCBB(int capacity, struct Item items[], int n) {
  qsort(items, n, sizeof(items[0]), compare); // Sort items based on value/weight ratio

  int curWeight = 0; // Current weight in knapsack
  double curValue = 0.0; // Current value in knapsack
  int finalSet[n]; // Array to store the selected items

  // Initialize finalSet array
  for (int i = 0; i < n; i++)
    finalSet[i] = 0;

  // Loop through items and select them greedily
  for (int i = 0; i < n; i++) {
    if (curWeight + items[i].weight <= capacity) {
      finalSet[i] = 1; // Mark item as selected
      curWeight += items[i].weight;
      curValue += items[i].value;
    } else {
      double remainingWeight = capacity - curWeight;
      curValue += (remainingWeight / items[i].weight) * items[i].value;
      break;
```

```
        }
    }

    // Print the selected items and their total value
    printf("Selected items:\n");
    for (int i = 0; i < n; i++) {
        if (finalSet[i]) {
            printf("Item %d - Weight: %d, Value: %d\n", i + 1, items[i].weight, items[i].value);
        }
    }
    printf("Total value: %.2f\n", curValue);
}

int main() {
    int capacity, n;
    printf("Enter the capacity of knapsack: ");
    scanf("%d", &capacity);
    printf("Enter the number of items: ");
    scanf("%d", &n);
    struct Item items[n];
    printf("Enter the weight and value of each item:\n");
    for (int i = 0; i < n; i++) {
        printf("Item %d: ", i + 1);
        scanf("%d %d", &items[i].weight, &items[i].value);
    }
    knapsackLCBB(capacity, items, n);
    return 0;
}
```

program 28

Write a program to implement Graph Coloring Algorithm

program

```
#include <stdio.h>

#include <stdbool.h>

#define V 4

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

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

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

           return false;


   return true;

 }


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

   if (v == V) {

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

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


       return true;

   }


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

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

           color[v] = c;


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

               return true;


           color[v] = 0;

       }
```

```
    }

    return false;
}

bool graphColoring(int graph[][V]) {
    int color[V];

    for (int i = 0; i < V; i++)
        color[i] = 0;

    return graphColoringUtil(graph, color, 0);
}

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

    if (!graphColoring(graph))
        printf("Solution does not exist.\n");

    return 0;
}
```

program 29

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

program

```
#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];

   // Building the K[][] array
   for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            K[i][w] = 0;
         else if (wt[i-1] <= w)
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
         else
            K[i][w] = K[i-1][w];
      }
   }

   return K[n][W];
}

int main() {
   int val[] = {60, 100, 120}; // values of items
   int wt[] = {10, 20, 30}; // weights of items
   int W = 50; // knapsack capacity
   int n = sizeof(val)/sizeof(val[0]);

   printf("Maximum value that can be obtained: %d", knapsack(W, wt, val, n));
   return 0;
}
```

program 30

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

program

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 100


// Structure to represent a graph node

struct Node {

   int data;

   struct Node* next;

};


// Structure to represent a graph

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;

};


// Function to create a new graph node

struct Node* createNode(int data) {

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

   newNode->data = data;

   newNode->next = NULL;

   return newNode;

}


// Function to create a graph with a given number of vertices

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 = (int*)malloc(numVertices * sizeof(int));

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

   return graph;
}

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

// Function to perform DFS
int DFS(struct Graph* graph, int startVertex, int currentVertex, int numVisited, int start) {
   graph->visited[currentVertex] = 1;

   if (numVisited == graph->numVertices && start == startVertex) {
       return 1; // Found a Hamiltonian cycle
   }
   struct Node* temp = graph->adjLists[currentVertex];
   while (temp != NULL) {
       int adjVertex = temp->data;
       if (!graph->visited[adjVertex]) {
           if (DFS(graph, startVertex, adjVertex, numVisited + 1, start)) {
               return 1;
           }
       }
       temp = temp->next;
   }

   graph->visited[currentVertex] = 0; // Backtrack
   return 0;
}

// Function to check if the graph contains a Hamiltonian cycle
int isHamiltonianCycle(struct Graph* graph) {
   for (int i = 0; i < graph->numVertices; i++) {
       if (DFS(graph, i, i, 1, i)) {
           return 1; // Found a Hamiltonian cycle starting from vertex i
       }
   }
   return 0; // No Hamiltonian cycle found
}

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

   // Add edges to the graph
   addEdge(graph, 0, 1);
   addEdge(graph, 1, 2);
   addEdge(graph, 2, 3);
   addEdge(graph, 3, 4);
   addEdge(graph, 4, 0);

   if (isHamiltonianCycle(graph)) {
       printf("The graph contains a Hamiltonian cycle.\n");
   } else {
       printf("The graph does not contain a Hamiltonian cycle.\n");
   }

   return 0;
}
```

program 31

Write a program to implement solve ‘N’ Queens Problem using Backtracking

program

```
#include <stdio.h>

#include <stdbool.h>


#define N 8


int board[N][N];


void printSolution() {

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

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

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

      }

      printf("\n");

   }

}


bool isSafe(int row, int col) {

   // Check if there is a queen in the same row on the left side

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

      if (board[row][i] == 1) {

         return false;

      }

   }


   // Check if there is a queen in the upper diagonal on the left side

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

      if (board[i][j] == 1) {

         return false;
```

```
        }
    }

    // Check if there is a queen in the lower diagonal on the left side
    for (int i = row, j = col; i < N && j >= 0; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

bool solveNQueensUtil(int col) {
    if (col >= N) {
        return true; // All queens are placed successfully
    }

    for (int i = 0; i < N; i++) {
        if (isSafe(i, col)) {
            board[i][col] = 1;

            if (solveNQueensUtil(col + 1)) {
                return true;
            }

            board[i][col] = 0; // Backtrack if placing a queen at position (i, col) doesn't lead to a solution
        }
    }
    return false; // No safe place found, need to backtrack to previous column
}

bool solveNQueens() {
    if (!solveNQueensUtil(0)) {
        printf("Solution does not exist.\n");
        return false;
    }

    printf("Solution for N-Queens problem:\n");
    printSolution();
    return true;
}

int main() {
    if (N <= 3) {
        printf("No solution exists for N less than or equal to 3.\n");
        return 1;
    }

    solveNQueens();

    return 0;
}
```

program 32

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

program

```
#include <stdio.h>


#define MAX_ITEMS 100

#define MAX_WEIGHT 100


int max(int a, int b) {

   return (a > b) ? a : b;

}


int knapsack(int values[], int weights[], int n, int capacity) {

   int dp[MAX_ITEMS + 1][MAX_WEIGHT + 1];


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

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

         if (i == 0 || w == 0) {

            dp[i][w] = 0;

         } else if (weights[i - 1] <= w) {

            dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);

         } else {

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

         }

      }

   }


   return dp[n][capacity];

}


int main() {
   int values[] = {60, 100, 120};
   int weights[] = {10, 20, 30};
   int n = sizeof(values) / sizeof(values[0]);
   int capacity = 50;

   int result = knapsack(values, weights, n, capacity);

   printf("Maximum value in knapsack = %d\n", result);

   return 0;
}
```

program 33

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

program

```
#include <stdio.h>

#include <stdbool.h>


#define MAX_NODES 10


// Function to perform depth-first search (DFS)

void DFS(int graph[MAX_NODES][MAX_NODES], int node, bool visited[MAX_NODES]) {

  visited[node] = true;


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

    if (graph[node][i] && !visited[i]) {

      DFS(graph, i, visited);

    }

  }

}


// Function to identify live nodes, E nodes, and dead nodes

void analyzeGraph(int graph[MAX_NODES][MAX_NODES], int numNodes) {

  bool visited[MAX_NODES] = {false};

  int liveNodes = 0, eNodes = 0, deadNodes = 0;


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

    if (!visited[i]) {

      DFS(graph, i, visited);

      liveNodes++;

    }

  }
```

```
    for (int i = 0; i < numNodes; i++) {
        visited[i] = false; // Reset visited array
        int outgoingEdges = 0;

        for (int j = 0; j < numNodes; j++) {
            if (graph[i][j]) {
                outgoingEdges++;
            }
        }

        if (outgoingEdges == 0) {
            eNodes++;
        }
    }

    deadNodes = numNodes - liveNodes;

    // Print results
    printf("Live Nodes: %d\n", liveNodes);
    printf("E Nodes: %d\n", eNodes);
    printf("Dead Nodes: %d\n", deadNodes);
}

int main() {
    // Example directed graph represented as an adjacency matrix
    int graph[MAX_NODES][MAX_NODES] = {
        {0, 1, 1, 0, 0},
        {0, 0, 0, 1, 1},
        {0, 0, 0, 0, 0},
        {0, 0, 1, 0, 0},
        {0, 0, 0, 0, 0}
    };

    int numNodes = 5;

    analyzeGraph(graph, numNodes);

    return 0;
}
```

program 34

Write a program to implement for finding Topological sorting and determine the time complexity for the same

program

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

#define MAX_NODES 100

// Structure to represent a graph node
struct Node {
   int value;
   struct Node* next;
};

// Structure to represent a graph
struct Graph {
   int numNodes;
   struct Node* adjacencyList[MAX_NODES];
};

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

// 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->adjacencyList[src];
   graph->adjacencyList[src] = newNode;
}

// Function to perform Depth First Search
void DFS(struct Graph* graph, int node, int visited[], int stack[], int* top) {
   visited[node] = 1;

   struct Node* current = graph->adjacencyList[node];
   while (current != NULL) {
      int neighbor = current->value;
      if (!visited[neighbor]) {
         DFS(graph, neighbor, visited, stack, top);
      }
      current = current->next;
```

```
    }

    stack[++(*top)] = node;
}

// Function to perform Topological Sorting
void topologicalSort(struct Graph* graph) {
    int visited[MAX_NODES] = {0};
    int stack[MAX_NODES];
    int top = -1;

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

    printf("Topological Sorting: ");
    while (top >= 0) {
        printf("%d ", stack[top--]);
    }
}

int main() {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numNodes = 6; // Number of nodes in the graph

    // Initialize adjacency list for each node
    for (int i = 0; i < graph->numNodes; ++i) {
        graph->adjacencyList[i] = NULL;
    }

    // Add edges to the graph
    addEdge(graph, 5, 2);
    addEdge(graph, 5, 0);
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);

    // Perform topological sort
    topologicalSort(graph);

    return 0;
}
```