योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;

Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

BTECH Programme: CSE-B

Course Title: Design and Analysis of Algorithm Lab

Course Code: (CIC-359)

Submitted By:-

Name: Rudra Sharma

Enrollment No:10417702722

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;

Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

VISION OF INSTITUTE

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

MISSION OF INSTITUTE

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building capabilities to enable them to make significant contributions to the world.

योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS

Grade A++ Accredited Institution by NAAC

NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;

Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE

An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

INDEX

S.No	EXP.	Date	Marks			Remark	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			

PROGRAM 1

Aim: To implement following algorithm using array as a data structure and analyse its time complexity.

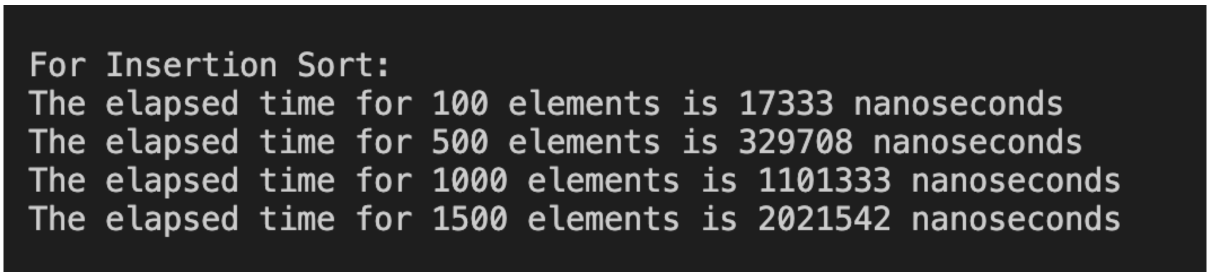
a) Insertion Sort

Theory:

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>

using namespace std;
using namespace std::chrono;

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        //move one position right if greater than key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout<<"For Insertion Sort:"<<endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];

        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        insertionSort(arr, size);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;

        delete[] arr;
    }

    return 0;
}
```

Output:



```
For Insertion Sort:
The elapsed time for 100 elements is 17333 nanoseconds
The elapsed time for 500 elements is 329708 nanoseconds
The elapsed time for 1000 elements is 1101333 nanoseconds
The elapsed time for 1500 elements is 2021542 nanoseconds
```

Graph:

Learning Outcomes:

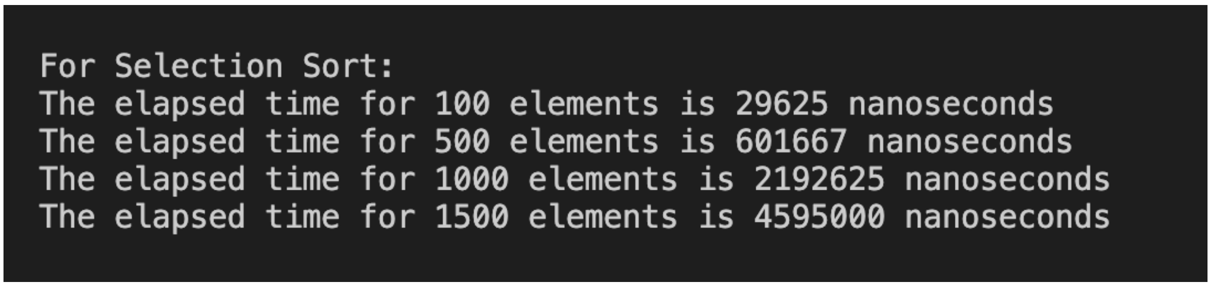
b) Selection Sort

Theory:

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>

using namespace std;
using namespace std::chrono;

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        // Find the minimum element in the unsorted part of the array
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element of unsorted part
        if (minIndex != i) {
            swap(arr[i], arr[minIndex]);
        }
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout<<"\n\nFor Selection Sort:"<<endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        selectionSort(arr, size);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;
        delete[] arr; // Deallocate the array
    }
    cout<<endl<<endl;
    return 0;
}
```

Output:



```
For Selection Sort:
The elapsed time for 100 elements is 29625 nanoseconds
The elapsed time for 500 elements is 601667 nanoseconds
The elapsed time for 1000 elements is 2192625 nanoseconds
The elapsed time for 1500 elements is 4595000 nanoseconds
```

Graph:

Learning Outcomes:

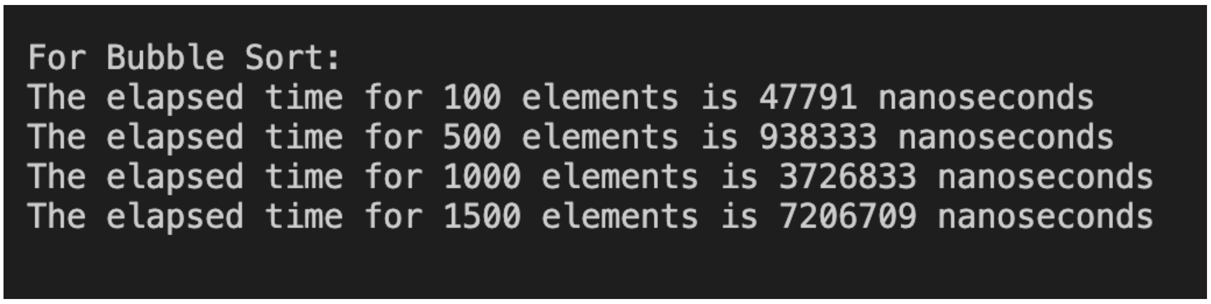
c) Bubble Sort

Theory:

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>

using namespace std;
using namespace std::chrono;

void bubbleSort(int arr[], int n) {
   int flag=0;
   for (int i = 0; i < n - 1; ++i) {
      flag=0;
      for (int j = 0; j < n - i - 1; ++j) {
         if (arr[j] > arr[j + 1]) {
            // Swap if the element found is greater than the next element
            swap(arr[j], arr[j + 1]);
            flag=1;
         }
      }
      // If no two elements were swapped by inner loop, then the array is sorted
      if (flag!=1) break;
   }
}

void generateRandomArray(int arr[], int size) {
   for (int i = 0; i < size; ++i) {
      arr[i] = rand();
   }
}

int main() {
   srand(static_cast<unsigned int>(time(0)));
   int sizes[] = {100, 500, 1000, 1500};

   cout<<"\n\nFor Bubble Sort:"<<endl;

   for (int i = 0; i < 4; ++i) {
      int size = sizes[i];
      int* arr = new int[size];
      generateRandomArray(arr, size);
      auto start = high_resolution_clock::now();
      bubbleSort(arr, size);
      auto end = high_resolution_clock::now();

      auto time_spent = duration_cast<nanoseconds>(end - start).count();

      cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;

      delete[] arr; // Deallocate the array
   }
   cout<<endl<<endl;
   return 0;
}
```

Output:



```
For Bubble Sort:
The elapsed time for 100 elements is 47791 nanoseconds
The elapsed time for 500 elements is 938333 nanoseconds
The elapsed time for 1000 elements is 3726833 nanoseconds
The elapsed time for 1500 elements is 7206709 nanoseconds
```

Graph:

Learning Outcomes:

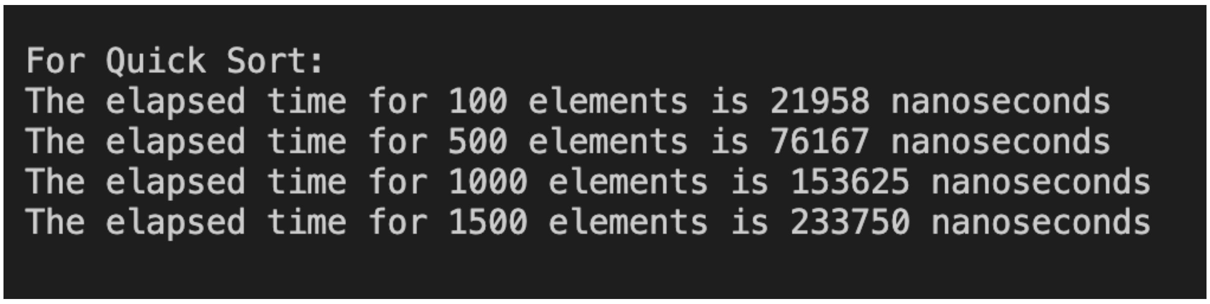
d) Quick Sort

Theory:

Program:

```
#include <iostream>
#include <cstdlib>
#include <chrono>

using namespace std;
using namespace std::chrono;

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;
}

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);
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout<<"\n\nFor Quick Sort:"<<endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];

        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        quickSort(arr, 0, size - 1);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;

        delete[] arr;
    }
    cout<<endl<<endl;
    return 0;
}
```

Output:



```
For Quick Sort:
The elapsed time for 100 elements is 21958 nanoseconds
The elapsed time for 500 elements is 76167 nanoseconds
The elapsed time for 1000 elements is 153625 nanoseconds
The elapsed time for 1500 elements is 233750 nanoseconds
```

Graph:

Learning Outcomes:

e) Merge Sort

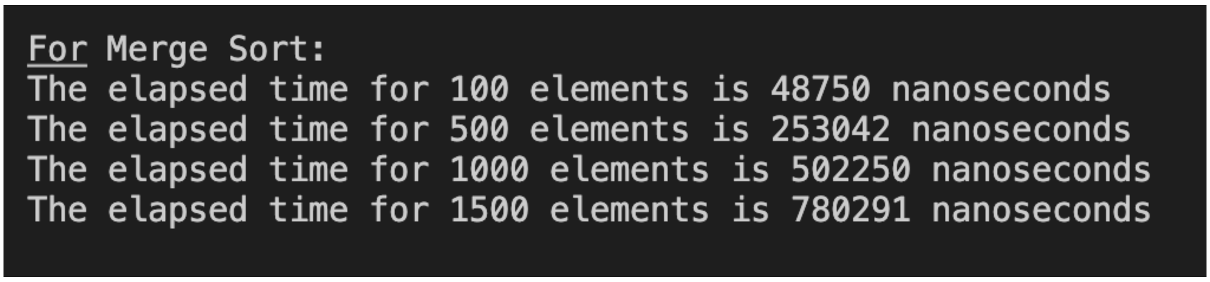
Theory:

Program:

```
#include <iostream>
#include <cstdlib>
#include <chrono>

using namespace std;
using namespace std::chrono;

void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1;
   int n2 = right - mid;

   int* L = new int[n1];
   int* R = new int[n2];

   for (int i = 0; i < n1; ++i)
      L[i] = arr[left + i];
   for (int j = 0; j < n2; ++j)
      R[j] = arr[mid + 1 + j];

   int i = 0;
   int j = 0;
   int k = left;

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

   while (i < n1) {
      arr[k] = L[i];
      ++i;
      ++k;
   }

   while (j < n2) {
      arr[k] = R[j];
      ++j;
      ++k;
   }

   delete[] L;
   delete[] R;
}

void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;
```

```
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Merge Sort:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];

        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        mergeSort(arr, 0, size - 1);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

Output:



```
For Merge Sort:
The elapsed time for 100 elements is 48750 nanoseconds
The elapsed time for 500 elements is 253042 nanoseconds
The elapsed time for 1000 elements is 502250 nanoseconds
The elapsed time for 1500 elements is 780291 nanoseconds
```

Graph:

Learning Outcomes:

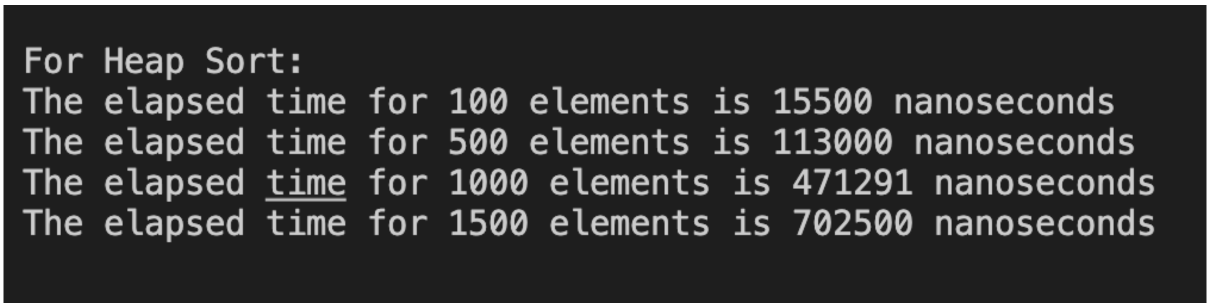
f) Heap Sort

Theory:

Program:

```
#include <iostream>
#include <cstdlib>
#include <chrono>

using namespace std;
using namespace std::chrono;

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

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

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

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; --i)
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; --i) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Heap Sort:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];
        generateRandomArray(arr, size);

        auto start = high_resolution_clock::now();
        heapSort(arr, size);
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

Output:



```
For Heap Sort:
The elapsed time for 100 elements is 15500 nanoseconds
The elapsed time for 500 elements is 113000 nanoseconds
The elapsed time for 1000 elements is 471291 nanoseconds
The elapsed time for 1500 elements is 702500 nanoseconds
```

Graph:

Learning Outcomes:

PROGRAM 2

Aim: To implement linear search and binary search and analyse its time complexity.

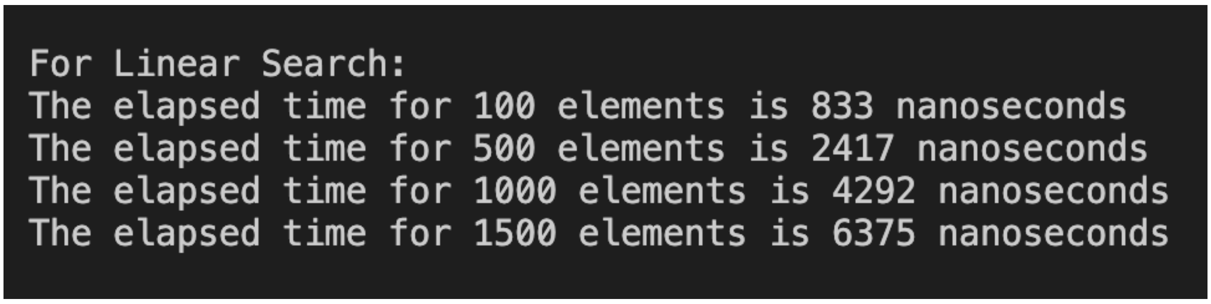
Linear Search

Theory:

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;

int linearSearch(int arr[], int n, int key)
{
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == key)
        {
            return i;
        }
    }
    return -1;
}

void generateRandomArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] = rand();
    }
}

int main() {
    srand(static_cast<unsigned int>(time(0)));
    int sizes[] = {100, 500, 1000, 1500};

    cout << "\n\nFor Linear Search:" << endl;

    for (int i = 0; i < 4; ++i) {
        int size = sizes[i];
        int* arr = new int[size];

        generateRandomArray(arr, size);
        auto start = high_resolution_clock::now();
        linearSearch(arr, size, rand());
        auto end = high_resolution_clock::now();

        auto time_spent = duration_cast<nanoseconds>(end - start).count();

        cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;

        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

Output:



```
For Linear Search:
The elapsed time for 100 elements is 833 nanoseconds
The elapsed time for 500 elements is 2417 nanoseconds
The elapsed time for 1000 elements is 4292 nanoseconds
The elapsed time for 1500 elements is 6375 nanoseconds
```

Graph:

Binary Search

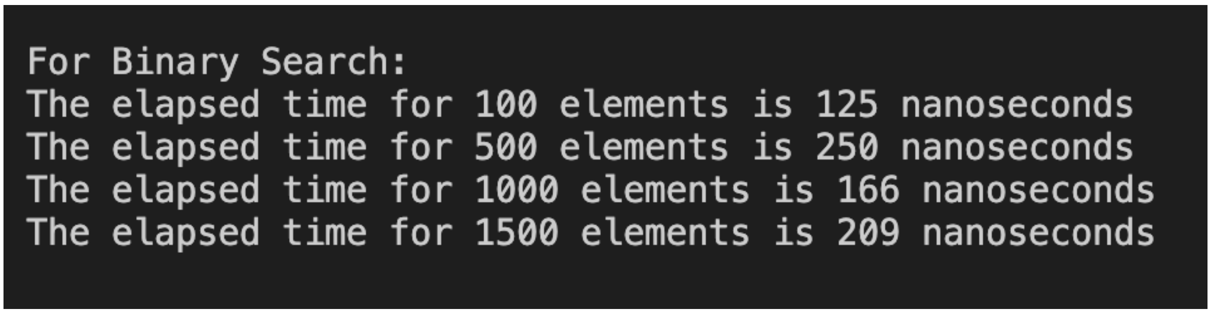
Theory

Program:

```
#include <iostream>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;

int binarySearch(int arr[], int n, int key)
{
   int low = 0;
   int high = n - 1;
   while (low <= high)
   {
      int mid = (low + high) / 2;
      if (arr[mid] == key)
      {
         return mid;
      }
      else if (arr[mid] < key)
      {
         low = mid + 1;
      }
      else
      {
         high = mid - 1;
      }
   }
   return -1;
}

void generateRandomArray(int arr[], int size) {
   for (int i = 0; i < size; ++i) {
      arr[i] = rand();
   }
}

int main() {
   srand(static_cast<unsigned int>(time(0)));
   int sizes[] = {100, 500, 1000, 1500};

   cout << "\n\nFor Binary Search:" << endl;

   for (int i = 0; i < 4; ++i) {
      int size = sizes[i];
      int* arr = new int[size];
      generateRandomArray(arr, size);

      auto start = high_resolution_clock::now();
      binarySearch(arr, size, rand());
      auto end = high_resolution_clock::now();

      auto time_spent = duration_cast<nanoseconds>(end - start).count();

      cout << "The elapsed time for " << size << " elements is " << time_spent << " nanoseconds" << endl;
```

```
        delete[] arr;
    }

    cout<<endl<<endl;
    return 0;
}
```

Output:



```
For Binary Search:
The elapsed time for 100 elements is 125 nanoseconds
The elapsed time for 500 elements is 250 nanoseconds
The elapsed time for 1000 elements is 166 nanoseconds
The elapsed time for 1500 elements is 209 nanoseconds
```

Graph:

Learning Outcomes:

PROGRAM 3

Aim: To design, implement, and test a Huffman coding algorithm in C, which efficiently compresses a given set of characters based on their frequencies, and to analyze the resulting Huffman codes.

Theory:

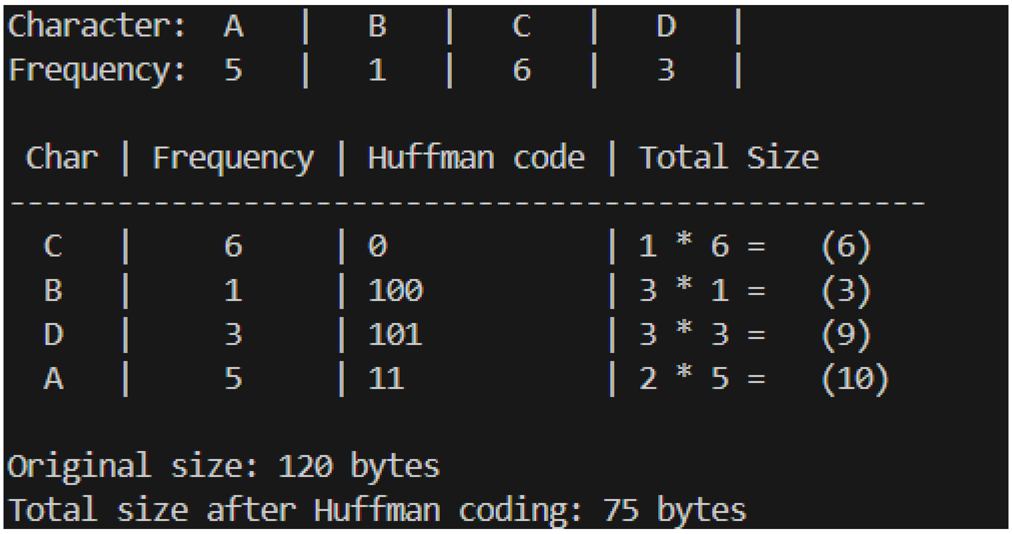
Program:

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

#define MAX_TREE_HT 50

struct MinHNode
{
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
};

struct MinHeap
{
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};

void printArray(int arr[], int n);

struct MinHNode *newNode(char item, unsigned freq)
{
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
  temp->item = item;
  temp->freq = freq;

  return temp;
}


struct MinHeap *createMinH(unsigned capacity)
{
  struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode *));
  return minHeap;
}

void swapMinHNode(struct MinHNode **a, struct MinHNode **b)
{
  struct MinHNode *t = *a;
  *a = *b;
  *b = t;
}

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)
  {
    swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}

int checkSizeOne(struct MinHeap *minHeap)
{
  return (minHeap->size == 1);
}

struct MinHNode *extractMin(struct MinHeap *minHeap)
{
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];

  --minHeap->size;
  minHeapify(minHeap, 0);

  return temp;
}

void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *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;
}

void buildMinHeap(struct MinHeap *minHeap)
{
  int n = minHeap->size - 1;
  int i;

  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}
```

```
int isLeaf(struct MinHNode *root){
  return !(root->left) && !(root->right);
}

struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size){
  struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);

  minHeap->size = size;
  buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size){
  struct MinHNode *left, *right, *top;
  struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(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);
}

void printHCodes(struct MinHNode *root, int arr[], int top){
  if (root->left)
  {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right)
  {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
    printf("  %c   |     %d     | ", root->item, root->freq);
    printArray(arr, top);
    int totalSize = top * root->freq;                          // calculate total size
    printf("%d * %d =   (%d)  \n", top, root->freq, totalSize); // print total size
  }
}

void HuffmanCodes(char item[], int freq[], int size){
  struct MinHNode *root = buildHuffmanTree(item, freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}

void printArray(int arr[], int n){
  int i;
  for (i = 0; i < n; ++i)
    printf("%d", arr[i]);
  for (i = n; i < 6; ++i)
    printf(" ");
  printf("       | ");
}

int main(){
  char arr[] = {'A', 'B', 'C', 'D',};
  int freq[] = {5, 1, 6, 3};

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

  printf("Character:");
  for (int i = 0; i < size; i++)
  {
    printf("  %c   | ", arr[i]);
  }
  printf("\n");
  printf("Frequency:");
  for (int i = 0; i < size; i++) {
    printf("  %d   | ", freq[i]);
  }
  printf("\n\n");

  printf(" Char | Frequency | Huffman code | Total Size");
  printf("\n---------------------------------------------------\n");
  HuffmanCodes(arr, freq, size);
  int originalSize = 0;
  for (int i = 0; i < size; i++)  {
    originalSize += freq[i] * 8; // assuming 8 bytes per character
  }
  int totalSizeAfterHuffman = 0;
  for (int i = 0; i < size; i++)
  {
    totalSizeAfterHuffman += freq[i] * (sizeof(int) + sizeof(char)); // assuming 1 byte per character and 4 bytes per int
  }
  printf("\nOriginal size: %d bytes", originalSize);
  printf("\nTotal size after Huffman coding: %d bytes\n", totalSizeAfterHuffman);
}
```

Output:



```
Character:  A   |   B   |   C   |   D   |
Frequency:  5   |   1   |   6   |   3   |

 Char | Frequency | Huffman code | Total Size
---------------------------------------------------
  C   |     6     | 0            | 1 * 6 =   (6)
  B   |     1     | 100          | 3 * 1 =   (3)
  D   |     3     | 101          | 3 * 3 =   (9)
  A   |     5     | 11           | 2 * 5 =   (10)

Original size: 120 bytes
Total size after Huffman coding: 75 bytes
```

Learning Outcomes:

PROGRAM 4

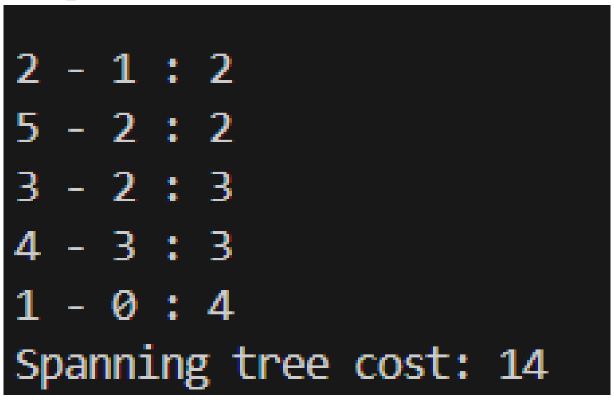
Aim: To implement Kruskal's algorithm in C to find the minimum spanning tree of a given graph, and to demonstrate the algorithm's ability to efficiently find the shortest connections between nodes in a network.

Theory:

Program:

```
#include <stdio.h>
#define MAX 30

typedef struct edge {
  int u, v, w;
} edge;

typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

// Applying Krushkal Algo
void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;

  for (i = 1; i < n; i++)
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }

  sort();

  for (i = 0; i < n; i++)
    belongs[i] = i;

  spanlist.n = 0;

  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n = spanlist.n + 1;
      applyUnion(belongs, cno1, cno2);
    }
  }
}

int find(int belongs[], int vertexno) {
  return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
  int i;

  for (i = 0; i < n; i++)
    if (belongs[i] == c2)
      belongs[i] = c1;
}

// Sorting algo
void sort() {
  int i, j;
  edge temp;

  for (i = 1; i < elist.n; i++)
    for (j = 0; j < elist.n - 1; j++)
      if (elist.data[j].w > elist.data[j + 1].w) {
        temp = elist.data[j];
        elist.data[j] = elist.data[j + 1];
        elist.data[j + 1] = temp;
      }
}

// Printing the result
void print() {
  int i, cost = 0;

  for (i = 0; i < spanlist.n; i++) {
    printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
    cost = cost + spanlist.data[i].w;
  }

  printf("\nSpanning tree cost: %d", cost);
}

int main() {
  int i, j, total_cost;

  n = 6;

  Graph[0][0] = 0;
  Graph[0][1] = 4;
  Graph[0][2] = 4;
  Graph[0][3] = 0;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;
  Graph[1][0] = 4;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 0;
  Graph[1][6] = 0;

  Graph[2][0] = 4;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 3;
  Graph[2][4] = 4;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 0;
  Graph[3][1] = 0;
  Graph[3][2] = 3;
  Graph[3][3] = 0;
  Graph[3][4] = 3;
  Graph[3][5] = 0;
  Graph[3][6] = 0;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 4;
  Graph[4][3] = 3;
  Graph[4][4] = 0;
  Graph[4][5] = 0;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 0;
  Graph[5][2] = 2;
  Graph[5][3] = 0;
  Graph[5][4] = 3;
  Graph[5][5] = 0;
  Graph[5][6] = 0;

  kruskalAlgo();
  print();
}
```

Output:



```
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14
```

Learning Outcomes:

PROGRAM 5

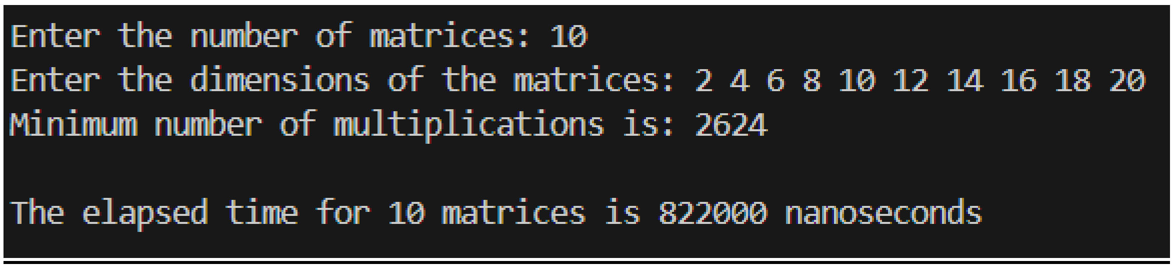
Aim: To implement Matrix Multiplication and analyse its time complexity.

Theory:

Program:

```
#include <iostream>
#include <climits>
#include<chrono>

using namespace std;
using namespace std::chrono;

int MatrixChainMultiplication(int dimensions[], int n) {

    int dp[n][n];

    // cost is zero when multiplying one matrix
    for (int i = 1; i < n; i++)
        dp[i][i] = 0;

    // L is the chain length
    for (int L = 2; L < n; L++) {
        for (int i = 1; i < n - L + 1; i++) {
            int j = i + L - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                // q = cost/scalar multiplications
                int q = dp[i][k] + dp[k + 1][j] + dimensions[i - 1] * dimensions[k] * dimensions[j];
                if (q < dp[i][j])
                    dp[i][j] = q;
            }
        }
    }

    // The minimum cost is found at dp[1][n-1]
    return dp[1][n - 1];
}

int main() {
    int n;
    cout<<"Enter the number of matrices: ";
    cin>>n;
    cout<<"Enter the dimensions of the matrices: ";
    int dimensions[n];
    for (int i = 0; i < n; i++)
        cin>>dimensions[i];

    auto start = high_resolution_clock::now();
    cout << "Minimum number of multiplications is: "
         << MatrixChainMultiplication(dimensions, n) << endl;
    auto end = high_resolution_clock::now();

    auto time_spent = duration_cast<nanoseconds>(end - start).count();
    cout << "\nThe elapsed time for " << n << " matrices is " << time_spent << " nanoseconds" << endl;

    cout<<endl;
    return 0;
}
```

Output:



```
Enter the number of matrices: 10
Enter the dimensions of the matrices: 2 4 6 8 10 12 14 16 18 20
Minimum number of multiplications is: 2624

The elapsed time for 10 matrices is 822000 nanoseconds
```

Learning Outcomes:

PROGRAM 6

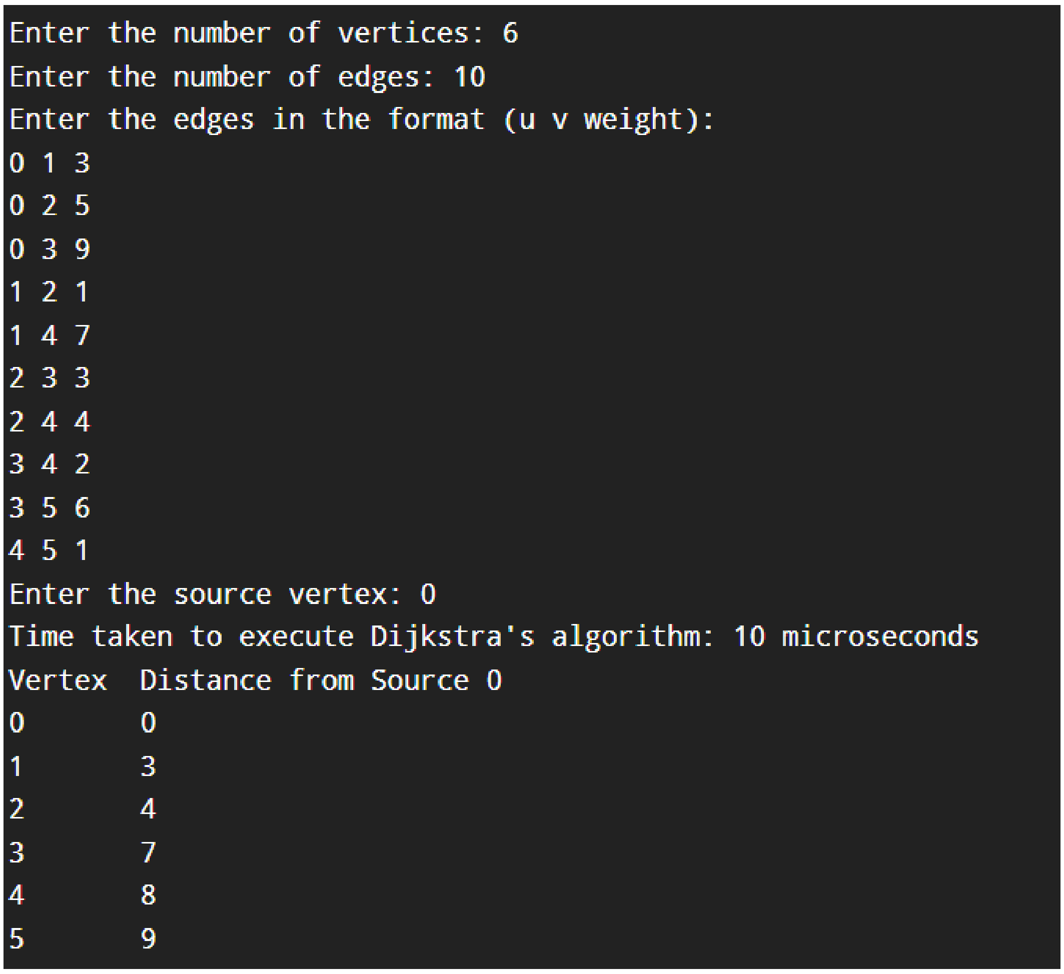
Aim: To implement Dijkstra algorithm and analyse its time complexity.

Theory:

Program:

```
#include <iostream>

#include <vector>

#include <queue>

#include <utility>

#include <climits>

#include <chrono>


using namespace std;

using namespace chrono;


typedef pair<int, int> pii;

void addEdge(vector<vector<pii>>& graph, int u, int v, int weight) {

    graph[u].push_back({weight, v});

    graph[v].push_back({weight, u});

}

vector<int> dijkstra(const vector<vector<pii>>& graph, int source) {

    int V = graph.size();

    vector<int> dist(V, INT_MAX);

    priority_queue<pii, vector<pii>, greater<pii>> pq;

    dist[source] = 0;

    pq.push({0, source});

    while (!pq.empty()) {

        int u = pq.top().second;

        int uDist = pq.top().first;

        pq.pop();

        if (uDist > dist[u]) continue;

        for (const auto& edge : graph[u]) {

            int weight = edge.first;

            int v = edge.second;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}

int main() {
    int V, E;
    cout << "\nEnter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;
    vector<vector<pii>> graph(V);
    cout << "Enter the edges in the format (u v weight):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v, weight;
        cin >> u >> v >> weight;
        addEdge(graph, u, v, weight);
    }
    int source;
    cout << "Enter the source vertex: ";
    cin >> source;
    auto start = high_resolution_clock::now();
    vector<int> distances = dijkstra(graph, source);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Time taken to execute Dijkstra's algorithm: " << duration << " microseconds" << endl;
    cout << "Vertex\tDistance from Source " << source << endl;
    for (int i = 0; i < V; ++i) {
        cout << i << "\t\t" << distances[i] << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of vertices: 6
Enter the number of edges: 10
Enter the edges in the format (u v weight):
0 1 3
0 2 5
0 3 9
1 2 1
1 4 7
2 3 3
2 4 4
3 4 2
3 5 6
4 5 1
Enter the source vertex: 0
Time taken to execute Dijkstra's algorithm: 10 microseconds
Vertex  Distance from Source 0
0       0
1       3
2       4
3       7
4       8
5       9
```

Learning Outcomes:

PROGRAM 7

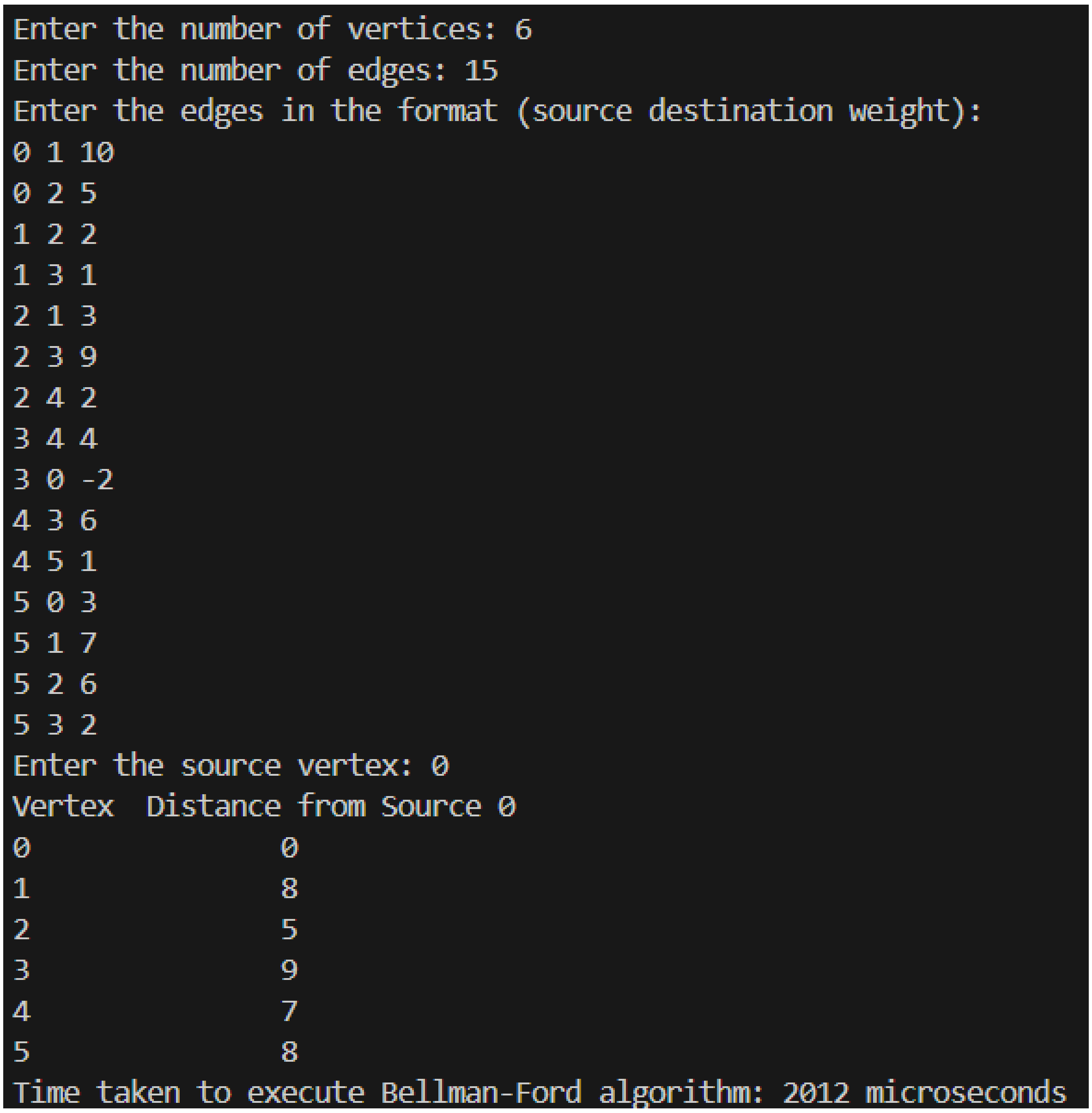
Aim: To implement Bellman Ford algorithm and analyse its time complexity

Theory:

Program:

```
#include <iostream>

#include <vector>

#include <limits.h>

#include <chrono>


using namespace std;

using namespace chrono;


struct Edge {

    int src, dest, weight;

};

class BellmanFord {

    int vertices;

    vector<Edge> edges;

public:

    BellmanFord(int vertices) {

        this->vertices = vertices;

    }

    void addEdge(int src, int dest, int weight) {

        edges.push_back({src, dest, weight});

    }

    void bellmanFord(int source) {

        vector<int> distance(vertices, INT_MAX);

        distance[source] = 0;

        // Step 1: Relax all edges V-1 times

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

            for (const auto& edge : edges) {

                int u = edge.src;

                int v = edge.dest;

                int weight = edge.weight;

                if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {

                    distance[v] = distance[u] + weight;

                }
            }
        }
        // Step 2: Check for negative-weight cycles
        for (const auto& edge : edges) {
            int u = edge.src;
            int v = edge.dest;
            int weight = edge.weight;
            if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
                cout << "Graph contains a negative-weight cycle" << endl;
                return;
            }
        }
        // Print the distances from source
        printDistances(distance, source);
    }
private:
    void printDistances(const vector<int>& distance, int source) {
        cout << "Vertex\tDistance from Source " << source << endl;
        for (int i = 0; i < vertices; ++i) {
            if (distance[i] == INT_MAX)
                cout << i << "\t\tInfinity" << endl;
            else
                cout << i << "\t\t" << distance[i] << endl;
        }
    }
};
int main() {
    int vertices, edges;
    cout << "\nEnter the number of vertices: ";
    cin >> vertices;
    BellmanFord graph(vertices);
    cout << "Enter the number of edges: ";
    cin >> edges;
    cout << "Enter the edges in the format (source destination weight):" << endl;
    for (int i = 0; i < edges; ++i) {
        int src, dest, weight;
        cin >> src >> dest >> weight;
        graph.addEdge(src, dest, weight);
    }
    int source;
    cout << "Enter the source vertex: ";
    cin >> source;
    // Start measuring time
    auto start = high_resolution_clock::now();
    graph.bellmanFord(source);
    auto end = high_resolution_clock::now();
    // Calculate and print elapsed time
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Time taken to execute Bellman-Ford algorithm: " << duration << " microseconds" << endl;
    return 0;
}
```

Output:


```
Enter the number of vertices: 6
Enter the number of edges: 15
Enter the edges in the format (source destination weight):
0 1 10
0 2 5
1 2 2
1 3 1
2 1 3
2 3 9
2 4 2
3 4 4
3 0 -2
4 3 6
4 5 1
5 0 3
5 1 7
5 2 6
5 3 2
Enter the source vertex: 0
Vertex  Distance from Source 0
0               0
1               8
2               5
3               9
4               7
5               8
Time taken to execute Bellman-Ford algorithm: 2012 microseconds
```

Learning Outcomes:

PROGRAM 8

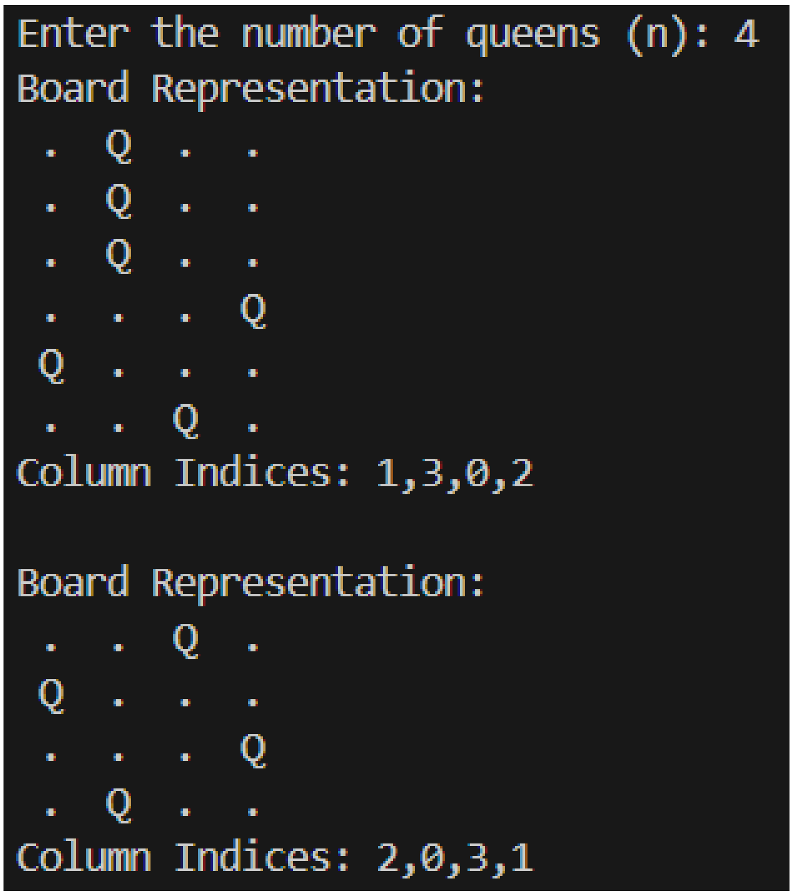
Aim: To implement n-Queen problem using backtracking.

Theory:

Program:

```
#include <iostream>

#include <vector>

#include <sstream>


using namespace std;


class NQueens {

private:

    int n;

    vector<vector<int>> board;

public:

    NQueens(int n) : n(n), board(n, vector<int>(n, 0)) {}

    void printBoard() {

        cout << "Board Representation:" << endl;

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

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

                if (board[i][j]) {

                    cout << " Q "; // Mark the queen's position

                } else {

                    cout << " . ";

                }

            }

            cout << endl;

        }

        cout << "Column Indices: ";

        vector<int> solution; // To store the column positions of queens

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

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

                if (board[i][j]) {

                    solution.push_back(j); // Store the column index of the queen

                }

            }

        }
        for (size_t i = 0; i < solution.size(); ++i) {
            cout << solution[i];
            if (i < solution.size() - 1) {
                cout << ",";
            }
        }
        cout << endl << endl;
    }
    // Function to check if placing a queen is safe
    bool isSafe(int row, int col) {
        // Check the column
        for (int i = 0; i < row; i++) {
            if (board[i][col]) return false;
        }
        // Check the upper left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j]) return false;
        }
        // Check the upper right diagonal
        for (int i = row, j = col; i >= 0 && j < n; i--, j++) {
            if (board[i][j]) return false;
        }
        return true;
    }
    // Backtracking function to solve the n-Queens problem
    bool solveNQueensUtil(int row) {
        if (row == n) {
            printBoard();
            return true;
        }
        bool foundSolution = false; // To track if we found a solution
        for (int col = 0; col < n; col++) {
            if (isSafe(row, col)) {
                board[row][col] = 1;
                foundSolution = solveNQueensUtil(row + 1) || foundSolution;
                board[row][col] = 0;
            }
        }
        return foundSolution;
    }
    void solveNQueens() {
        if (!solveNQueensUtil(0)) {
            cout << "No solution exists for n = " << n << endl;
        }
    }
};

int main() {
    int n;
    cout << "\nEnter the number of queens (n): ";
    cin >> n;
    NQueens nQueens(n);
    nQueens.solveNQueens();
    return 0;
}
```

Output:



```
Enter the number of queens (n): 4
Board Representation:
 .  Q  .  .
 .  Q  .  .
 .  Q  .  .
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .
Column Indices: 1,3,0,2

Board Representation:
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .
Column Indices: 2,0,3,1
```

Learning Outcomes:

PROGRAM 9

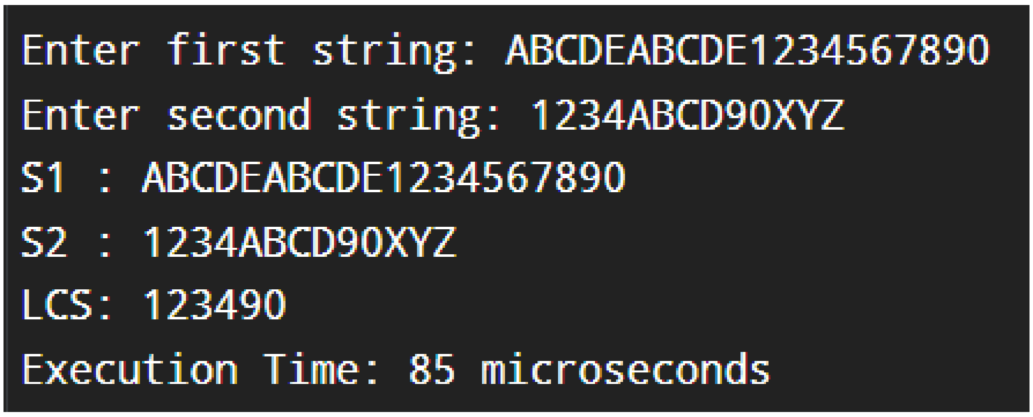
Aim: To implement Longest Common Subsequence problem and analyse its time complexity.

Theory:

Program:

```
#include <iostream>

#include <vector>

#include <chrono>

#include <string>


using namespace std;

using namespace chrono;


void lcsAlgo(const string &S1, const string &S2, int m, int n) {

    vector<vector<int>> LCS_table(m + 1, vector<int>(n + 1));

    // Building the matrix in bottom-up way

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

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

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

                LCS_table[i][j] = 0;

            else if (S1[i - 1] == S2[j - 1])

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

            else

                LCS_table[i][j] = max(LCS_table[i - 1][j], LCS_table[i][j - 1]);

        }

    }

    int index = LCS_table[m][n];

    string lcsString(index, '\0'); // Preallocate string with size index

    int i = m, j = n;

    while (i > 0 && j > 0) {

        if (S1[i - 1] == S2[j - 1]) {

            lcsString[index - 1] = S1[i - 1];

            i--;

            j--;

            index--;

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

            i--;

        } else {
            j--;
        }
    }
    // Printing the subsequences
    cout << "S1 : " << S1 << "\nS2 : " << S2 << "\nLCS: " << lcsString << "\n";
}

int main() {
    string S1, S2;
    // Taking input from user
    cout << "\nEnter first string: ";
    cin >> S1;
    cout << "Enter second string: ";
    cin >> S2;
    int m = S1.length();
    int n = S2.length();
    auto start = high_resolution_clock::now();
    lcsAlgo(S1, S2, m, n);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Execution Time: " << duration << " microseconds" << endl;
    return 0;
}
```

Output:



```
Enter first string: ABCDEABCDE1234567890
Enter second string: 1234ABCD90XYZ
S1 : ABCDEABCDE1234567890
S2 : 1234ABCD90XYZ
LCS: 123490
Execution Time: 85 microseconds
```

Learning Outcomes:

PROGRAM 10

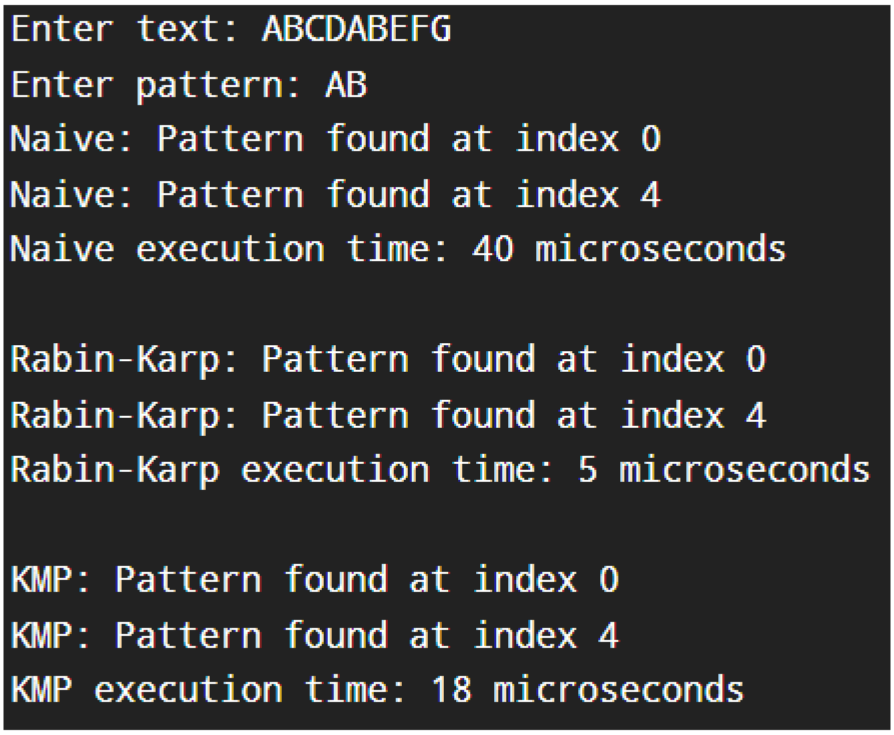
Aim: To implement Naive String-Matching algorithm, Rabin Karp algorithm and knuth Morris Pratt algorithm and analyse its time complexity

Theory:

Program:

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>
using namespace std;
using namespace chrono;
// Naive String Matching Algorithm
void naiveStringMatch(const string& text, const string& pattern) {
    int m = pattern.length();
    int n = text.length();
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (text[i + j] != pattern[j]) {
                break;
            }
        }
        if (j == m) {
            cout << "Naive: Pattern found at index " << i << endl;
        }
    }
}
// Rabin-Karp Algorithm
void rabinKarp(const string& text, const string& pattern) {
    int m = pattern.length();
    int n = text.length();
    const int d = 256; // Number of characters in the input alphabet
    const int q = 101; // A prime number for hashing
    int h = 1;
    int p = 0; // Hash value for pattern
    int t = 0; // Hash value for text
    // Calculate the value of h = d^(m-1) % q
    for (int i = 0; i < m - 1; i++) {
        h = (h * d) % q;
    }
    // Calculate initial hash values for pattern and first window of text
    for (int i = 0; i < m; i++) {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }
    for (int i = 0; i <= n - m; i++) {
        if (p == t) {
            // Check for characters one by one
            int j;
            for (j = 0; j < m; j++) {
                if (text[i + j] != pattern[j]) {
                    break;
                }
            }
            if (j == m) {
                cout << "Rabin-Karp: Pattern found at index " << i << endl;
            }
        }
        // Calculate hash value for next window
        if (i < n - m) {
            t = (d * (t - text[i] * h) + text[i + m]) % q;
            if (t < 0) {
                t += q;
            }
        }
    }
}
// Knuth-Morris-Pratt (KMP) Algorithm
void computeLPSArray(const string& pattern, vector<int>& lps) {
    int len = 0; // Length of previous longest prefix suffix
    lps[0] = 0; // LPS[0] is always 0
    int i = 1;
    int m = pattern.length();
    while (i < m) {
        if (pattern[i] == pattern[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
void kmp(const string& text, const string& pattern) {
    int n = text.length();
    int m = pattern.length();
    vector<int> lps(m);
    computeLPSArray(pattern, lps);
    int i = 0; // Index for text
    int j = 0; // Index for pattern
    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }
        if (j == m) {
            cout << "KMP: Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {
    string text, pattern;
    cout << "\nEnter text: ";
    cin >> text;
    cout << "Enter pattern: ";
    cin >> pattern;

    // Naive String Matching
    auto start = high_resolution_clock::now();
    naiveStringMatch(text, pattern);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Naive execution time: " << duration << " microseconds" << endl << endl;

    // Rabin-Karp Algorithm
    start = high_resolution_clock::now();
    rabinKarp(text, pattern);
    end = high_resolution_clock::now();
    duration = duration_cast<microseconds>(end - start).count();
    cout << "Rabin-Karp execution time: " << duration << " microseconds" << endl << endl;

    // KMP Algorithm
    start = high_resolution_clock::now();
    kmp(text, pattern);
    end = high_resolution_clock::now();
    duration = duration_cast<microseconds>(end - start).count();
    cout << "KMP execution time: " << duration << " microseconds" << endl << endl;
    return 0;
}
```

Output:



```
Enter text: ABCDABEFG
Enter pattern: AB
Naive: Pattern found at index 0
Naive: Pattern found at index 4
Naive execution time: 40 microseconds

Rabin-Karp: Pattern found at index 0
Rabin-Karp: Pattern found at index 4
Rabin-Karp execution time: 5 microseconds

KMP: Pattern found at index 0
KMP: Pattern found at index 4
KMP execution time: 18 microseconds
```

Learning Outcomes: