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

B. Tech Programme: AI-ML (A)

(5th Semester)

Course Title: Design and Analysis of Algorithms Lab

Course Code: AIML- 353

Submitted To:

Dr. Sandhya Tarwani

Submitted By:

Name: Kunsh Sabharwal

Enrolment No: 01117711623

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

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.

KUNSH SABHARWAL

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.	Experiment	Date	Marks			Remarks	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			
1.								
2.								
3.								
4.								
5.								

EXPERIMENT 1

Problem statement: Sort a given set of elements using insertion sort algorithm and find the time complexity.

Theory:

Algorithm:

Source Code:

InsertionSort.cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(vector<int> &arr)
{
    int n = arr.size();
    for (int j = 1; j <= arr.size() - 1; j++)
    {
        int key = arr[j];
        int i = j - 1;
        while (i >= 0 && arr[i] > key)
        {
            arr[i + 1] = arr[i];
            i = i - 1;
        }
        arr[i + 1] = key;
    }
}

int main()
{
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements: " << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    insertionSort(arr);
    cout << "The sorted array is: " << endl;
    for (int k = 0; k < arr.size(); k++)
    {
        cout << arr[k] << endl;
    }
    return 0;
}
```

Output:

Time Complexity:

	Best	Worst	Average
Insertion Sort			

Learning Outcome:

EXPERIMENT 2

Problem statement: Sort a given set of elements using selection and bubble sort algorithms and find the time complexity.

Theory:

(a) Selection Sort -

(b) Bubble Sort –

Algorithm:

(a) Selection Sort –

(b) Bubble Sort

Source Code:

SelectionAndBubbleSort.cpp

SelectionAndBubbleSort.cpp > main()

```cpp
#include <iostream>
#include <vector>
using namespace std;

void selectionSort(vector<int> &arr)
{
    for (int i = 0; i <= arr.size() - 2; i++)
    {
        int min = i;
        for (int j = i; j <= arr.size() - 1; j++)
        {
            if (arr[j] < arr[min])
                min = j;
        }
        swap(arr[i], arr[min]);
    }
}

void bubbleSort(vector<int> &arr)
{
    for (int i = arr.size() - 1; i >= 1; i--)
    {
        bool swapped = false;
        for (int j = 0; j <= i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swapped = true;
                swap(arr[j], arr[j + 1]);
            }
        }
        if (swapped == false)
            break;
    }
}
```

```cpp
int main()
{
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements: " << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int choice;
    cout << "Enter the sorting algorithm you want to use: " << endl;
    cout << "1. Selection Sort" << endl;
    cout << "2. Bubble Sort" << endl;
    cin >> choice;
    if (choice == 1)
        selectionSort(arr);
    else
        bubbleSort(arr);
    cout << "The sorted array is: " << endl;
    for (int k = 0; k < arr.size(); k++)
    {
        cout << arr[k] << endl;
    }
    return 0;
}

```

Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS D:\Kunsh DAA Lab> cd "d:\Kunsh DAA Lab\" ; if ($?) { g++ SelectionAndBubbleSort.cpp -o SelectionAndBubbleSort } ; if ($?) { .\SelectionAndBubbleSort }
Enter the size of the array: 5
Enter the elements:
5
4
3
2
1
Enter the sorting algorithm you want to use:
1. Selection Sort
2. Bubble Sort
1
The sorted array is:
1
2
3
4
5
PS D:\Kunsh DAA Lab>
```

Time Complexity:

	Best	Worst	Average
Selection Sort			
Bubble Sort			

Learning Outcome:

EXPERIMENT 3

Problem statement: Apply linear search and binary search algorithms to find an element in a given set of data and analyse their time complexity.

Theory:

Algorithm: (a) Linear Search

(b) Binary Search

Source Code:

Linear_and_Binary_Search.cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(vector<int> &arr)
{
    int n = arr.size();
    for (int j = 1; j <= arr.size() - 1; j++)
    {
        int key = arr[j];
        int i = j - 1;
        while (i >= 0 && arr[i] > key)
        {
            arr[i + 1] = arr[i];
            i = i - 1;
        }
        arr[i + 1] = key;
    }
}

int linear_search(vector<int> &arr, int key)
{
    for (int i = 0; i < arr.size(); i++)
    {
        if (arr[i] == key)
        {
            return i;
        }
    }
    return -1;
}
```

Linear_and_Binary_Search.cpp

Linear_and_Binary_Search.cpp > main()

```cpp
int binary_search(vector<int> &arr, int key)
{
    int low = 0;
    int high = arr.size() - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
        {
            return mid;
        }
        else if (arr[mid] > key)
        {
            high = mid - 1;
        }
        else
        {
            low = mid + 1;
        }
    }
    return -1;
}
```

Linear_and_Binary_Search.cpp

Linear_and_Binary_Search.cpp > main()

```cpp
int main()
{
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements: " << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int key;
    cout << "Enter the element you want to search in the entered array: ";
    cin >> key;
    int choice;
    cout << "Enter the searching algorithm you want to use: " << endl;
    cout << "1. Linear Search" << endl;
    cout << "2. Binary Search" << endl;
    cin >> choice;
    int ans;
    if (choice == 1)
        ans = linear_search(arr, key);
    else
    {
        insertionSort(arr);
        ans = binary_search(arr, key);
    }
    if (ans == -1)
    {
        cout << "Element not found in the entered array!" << endl;
    }
    else
    {
        cout << "Element found at " << ans << " index in the entered array." << endl;
    }
    return 0;
}
```

Outputs:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Kunsh DAA Lab> cd "d:\Kunsh DAA Lab\" ; if ($?) { g++ Linear_and_Binary_Search.cpp -o Linear_and_Binary_Search } ; if ($?) { .\Linear_and_Binary_Search }
Enter the size of the array: 5
Enter the elements:
5
4
3
2
1
Enter the element you want to search in the entered array: 3
Enter the searching algorithm you want to use:
1. Linear Search
2. Binary Search
1
Element found at 2 index in the entered array.
PS D:\Kunsh DAA Lab>
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Kunsh DAA Lab> cd "d:\Kunsh DAA Lab\" ; if ($?) { g++ Linear_and_Binary_Search.cpp -o Linear_and_Binary_Search } ; if ($?) { .\Linear_and_Binary_Search }
Enter the size of the array: 5
Enter the elements:
5
4
3
2
1
Enter the element you want to search in the entered array: 3
Enter the searching algorithm you want to use:
1. Linear Search
2. Binary Search
2
Element found at 2 index in the entered array.
PS D:\Kunsh DAA Lab>
```

Time Complexity:

	Best	Worst	Average
Linear Search			
Binary Search			

Learning Outcome:

EXPERIMENT 4

Problem statement: Sort a given set of elements using merge sort algorithm and find the time complexity for different values of n.

Theory:

KUNSH SABHARWAL

Source Code:

MergeSort.cpp

MergeSort.cpp > recursiveMergeSort(int *, int, int)

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

void mergeSections(int *array, int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *leftArr = new int[n1];
    int *rightArr = new int[n2];

    for (int i = 0; i < n1; i++)
        leftArr[i] = array[left + i];

    for (int j = 0; j < n2; j++)
        rightArr[j] = array[mid + 1 + j];

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

    while (i < n1 && j < n2)
    {
        if (leftArr[i] <= rightArr[j])
            array[k++] = leftArr[i++];
        else
            array[k++] = rightArr[j++];
    }

    while (i < n1)
        array[k++] = leftArr[i++];
    while (j < n2)
        array[k++] = rightArr[j++];

    delete[] leftArr;
    delete[] rightArr;
}
```

MergeSort.cpp

MergeSort.cpp > mergeSections(int *, int, int, int)

```cpp
void recursiveMergeSort(int *array, int left, int right)
{
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;
    recursiveMergeSort(array, left, mid);
    recursiveMergeSort(array, mid + 1, right);
    mergeSections(array, left, mid, right);
}

int main()
{
    srand(static_cast<unsigned>(time(nullptr)));

    int sizes[] = {100, 1000, 10000, 50000, 80000, 100000};
    int totalSizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int s = 0; s < totalSizes; s++)
    {
        int n = sizes[s];
        cout << "Sorting size: " << n << " using Merge Sort..." << endl;

        int *arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = rand();

        auto start = high_resolution_clock::now();
        recursiveMergeSort(arr, 0, n - 1);
        auto end = high_resolution_clock::now();

        duration<double> elapsed = end - start;
        cout << "Time taken: " << elapsed.count() << " seconds\n"
             << endl;

        delete[] arr;
    }

    return 0;
}
```

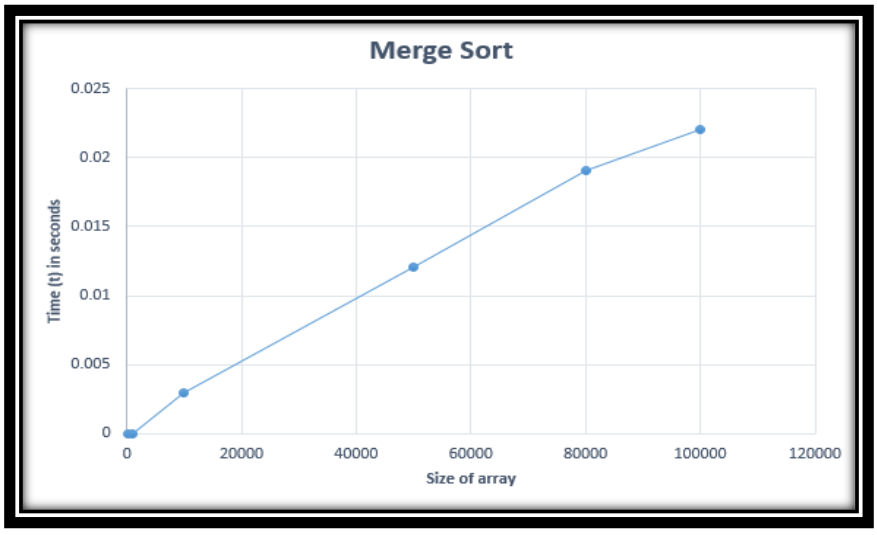
Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Kunsh DAA Lab> cd "d:\Kunsh DAA Lab\" ; if ($?) { g++ MergeSort.cpp -o MergeSort } ; if ($?) { .\MergeSort }
Sorting size: 100 using Merge Sort...
Time taken: 0 seconds

Sorting size: 1000 using Merge Sort...
Time taken: 0 seconds

Sorting size: 10000 using Merge Sort...
Time taken: 0.002084 seconds

Sorting size: 50000 using Merge Sort...
Time taken: 0.010973 seconds

Sorting size: 80000 using Merge Sort...
Time taken: 0.018085 seconds

Sorting size: 100000 using Merge Sort...
Time taken: 0.022001 seconds

PS D:\Kunsh DAA Lab>
```

Input (n) vs Time (t) Table:

N	Time (t) in secs
100	0
1000	0
10000	0.002999
50000	0.012087
80000	0.019073
100000	0.022076

Graph:



Time Complexity:

	Best	Worst	Average
Merge Sort			

Learning Outcome:

EXPERIMENT 5

Problem statement: Sort a given set of elements using quick sort algorithm and find the time complexity for different values of n.

Theory:

Algorithm:

Source Code:

QuickSort.cpp

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

void swapValues(int *array, int idx1, int idx2)
{
    int temp = array[idx1];
    array[idx1] = array[idx2];
    array[idx2] = temp;
}

int partitionHelper(int *array, int low, int pivot, int high)
{
    int left = low, boundary = low;
    while (left < high)
    {
        if (array[left] < array[pivot])
        {
            swapValues(array, left, boundary);
            boundary++;
        }
        left++;
    }
    swapValues(array, boundary, pivot);
    return boundary;
}

void quickSorter(int *array, int low, int high)
{
    if (low >= high)
        return;

    int pivot = high;
    int newPivot = partitionHelper(array, low, pivot, high);

    quickSorter(array, low, newPivot - 1);
    quickSorter(array, newPivot + 1, high);
}
```

QuickSort.cpp

```cpp
int main()
{
    srand(static_cast<unsigned>(time(nullptr)));

    int inputSizes[] = {10000, 50000, 80000, 100000, 150000, 180000};
    int totalCases = sizeof(inputSizes) / sizeof(inputSizes[0]);

    for (int t = 0; t < totalCases; t++)
    {
        int n = inputSizes[t];
        cout << "Sorting size: " << n << " using Quick Sort..." << endl;

        int *arr = new int[n];
        for (int i = 0; i < n; i++)
            arr[i] = rand();

        auto start = high_resolution_clock::now();
        quickSorter(arr, 0, n - 1);
        auto end = high_resolution_clock::now();

        duration<double> elapsed = end - start;
        cout << "Time taken: " << elapsed.count() << " seconds\n"
             << endl;

        delete[] arr;
    }

    return 0;
}
```

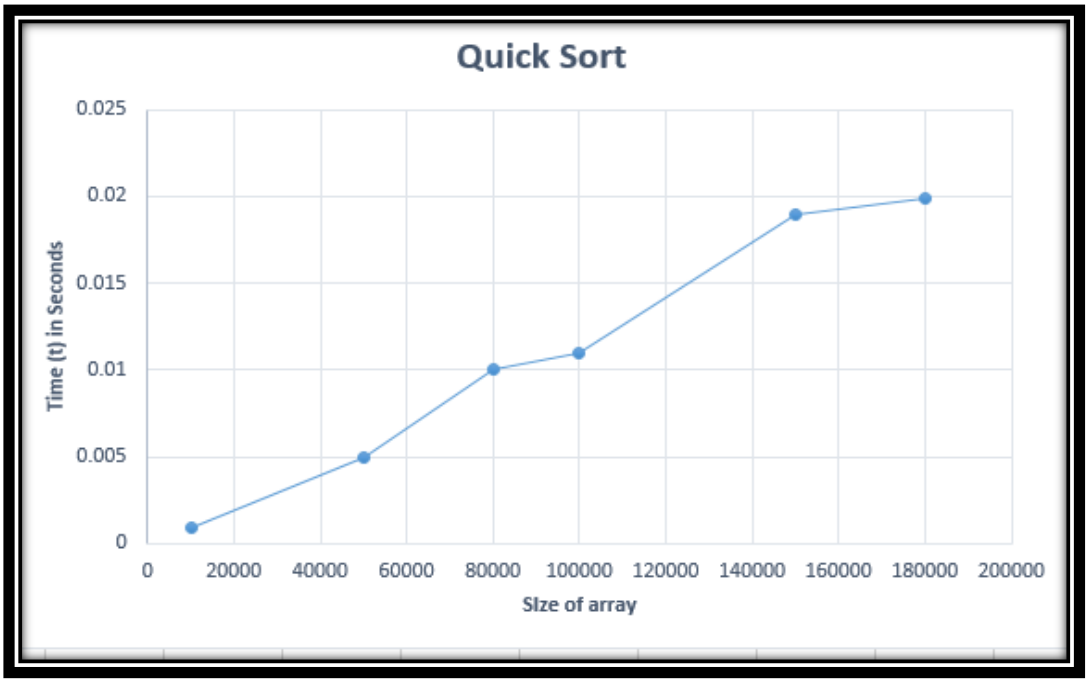
Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Kunsh DAA Lab> cd "d:\Kunsh DAA Lab\" ; if ($?) { g++ QuickSort.cpp -o QuickSort } ; if ($?) { .\QuickSort }
Sorting size: 10000 using Quick Sort...
Time taken: 0.000922 seconds

Sorting size: 50000 using Quick Sort...
Time taken: 0.005 seconds

Sorting size: 80000 using Quick Sort...
Time taken: 0.010016 seconds

Sorting size: 100000 using Quick Sort...
Time taken: 0.010996 seconds

Sorting size: 150000 using Quick Sort...
Time taken: 0.019 seconds

Sorting size: 180000 using Quick Sort...
Time taken: 0.019926 seconds

PS D:\Kunsh DAA Lab>
```

Input (n) vs Time (t) Table:

N	Time (t) in secs
100	0.000922
1000	0.005
10000	0.010016
50000	0.010996
80000	0.019
100000	0.019926

Graph:



Time Complexity:

	Best	Worst	Average
Quick Sort			

Learning Outcome:

EXPERIMENT 6

Problem statement: Write a program to implement Fractional Knapsack Problem using Greedy Method.

Theory:

Algorithm:

Source Code:

FractionalKnapsackProblem.cpp

FractionalKnapsackProblem.cpp > main()

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <algorithm>
using namespace std;
using namespace chrono;

struct Item
{
    double value, weight;
};

// Comparator to sort by value/weight ratio
bool cmp(const Item &a, const Item &b)
{
    return (a.value / a.weight) > (b.value / b.weight);
}

double fractionalKnapsack(Item items[], int n, double capacity)
{
    sort(items, items + n, cmp);

    double totalValue = 0;
    for (int i = 0; i < n && capacity > 0; i++)
    {
        if (items[i].weight <= capacity)
        {
            capacity -= items[i].weight;
            totalValue += items[i].value;
        }
        else
        {
            totalValue += (items[i].value / items[i].weight) * capacity;
            capacity = 0;
        }
    }
    return totalValue;
}
```

FractionalKnapsackProblem.cpp

FractionalKnapsackProblem.cpp > fractionalKnapsack(Item [], int, double)

```cpp
int main()
{
    srand(time(nullptr));

    int sizes[] = {1000, 5000, 10000, 50000, 80000, 100000};
    double capacity = 10000;

    for (int n : sizes)
    {
        Item *items = new Item[n];
        for (int i = 0; i < n; i++)
        {
            items[i].weight = (rand() % 100) + 1;
            items[i].value = (rand() % 500) + 1;
        }

        auto start = high_resolution_clock::now();
        double maxProfit = fractionalKnapsack(items, n, capacity);
        auto end = high_resolution_clock::now();

        double timeTaken = duration<double>(end - start).count();

        cout << "Number of items: " << n << "\n";
        cout << "Highest Profit: " << maxProfit << "\n";
        cout << "Time Taken: " << timeTaken << " seconds\n";
        cout << "------------------------------\n";

        delete[] items;
    }
    return 0;
}
```

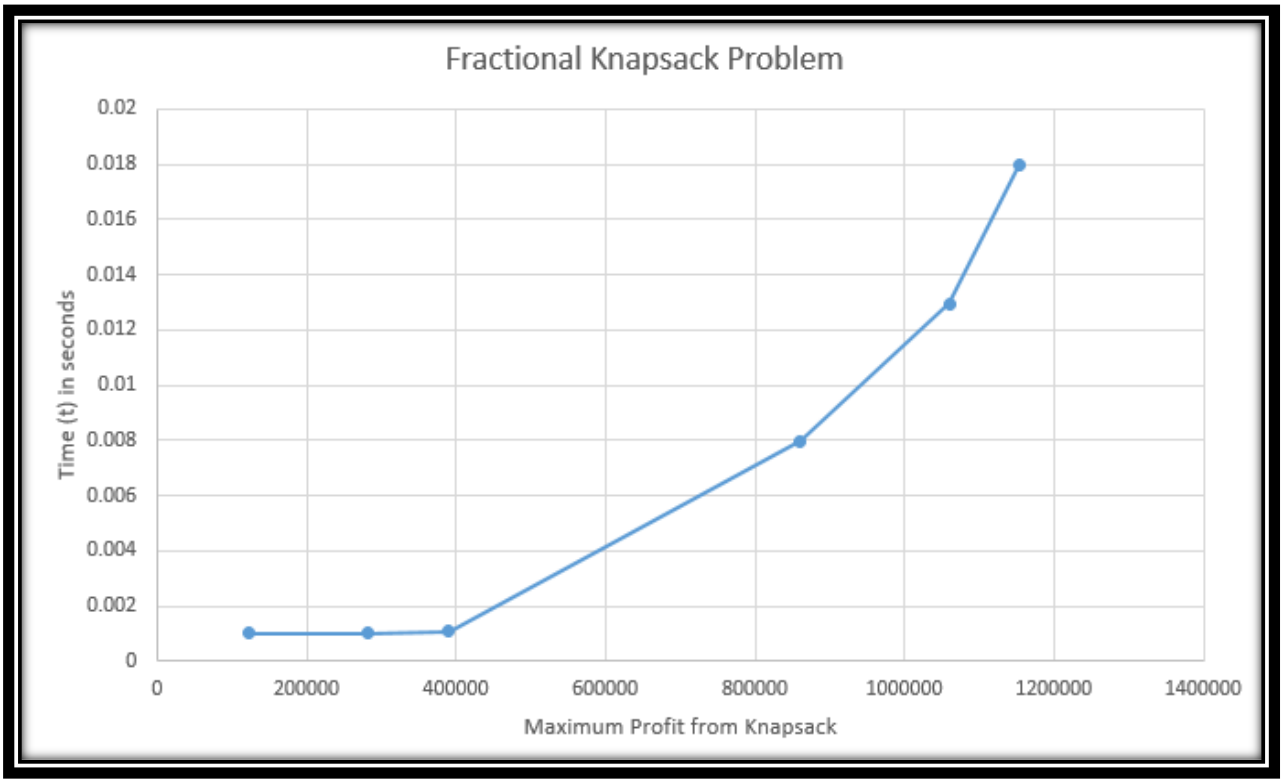
Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS D:\Kunsh DAA Lab> cd "d:\Kunsh DAA Lab\" ; if ($?) { g++ FractionalKnapsackProblem.cpp -o FractionalKnapsackProblem } ; if ($?) { .\FractionalKnapsackProblem }
Number of items: 1000
Highest Profit: 123275
Time Taken: 0.000996 seconds
------------------------------
Number of items: 5000
Highest Profit: 280744
Time Taken: 0.000997 seconds
------------------------------
Number of items: 10000
Highest Profit: 390485
Time Taken: 0.001079 seconds
------------------------------
Number of items: 50000
Highest Profit: 859904
Time Taken: 0.007974 seconds
------------------------------
Number of items: 80000
Highest Profit: 1.05955e+006
Time Taken: 0.012957 seconds
------------------------------
Number of items: 100000
Highest Profit: 1.15275e+006
Time Taken: 0.017954 seconds
------------------------------
PS D:\Kunsh DAA Lab>
```

Input (n) vs Time (t) Table:

Maximum Profit from Knapsack	Time (t) in seconds
123275	0.000996
280744	0.000997
390485	0.001079
859904	0.007974
1.06E+06	0.012957
1.15E+06	0.017954

Graph:



Time Complexity:

	Best	Worst	Average
Fractional Knapsack			

Learning Outcome:

EXPERIMENT 7

Problem statement: Write a program to find minimum spanning tree using Prim's Algorithm.

Theory:

Algorithm:

Source Code:

PrimsAlgorithm.cpp

PrimsAlgorithm.cpp > main()

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <ctime>
#include <chrono>
#include <limits>
using namespace std;
using namespace chrono;

struct Edge
{
    int node;
    double cost;
};

double primMST(const vector<vector<Edge>> &graph, int vertices)
{
    vector<bool> visited(vertices, false);
    vector<double> minCost(vertices, numeric_limits<double>::max());
    priority_queue<pair<double, int>, vector<pair<double, int>>, greater<>> pq;

    minCost[0] = 0;
    pq.push({0, 0});
    double totalCost = 0;

    while (!pq.empty())
    {
        int current = pq.top().second;
        double cost = pq.top().first;
        pq.pop();

        if (visited[current])
            continue;

        visited[current] = true;
        totalCost += cost;
```

PrimsAlgorithm.cpp

PrimsAlgorithm.cpp > main()

```
16  double primMST(const vector<vector<Edge>> &graph, int vertices)
26      while (!pq.empty())
38          for (auto &edge : graph[current])
39          {
40              int nextNode = edge.node;
41              double nextCost = edge.cost;
42
43              if (!visited[nextNode] && nextCost < minCost[nextNode])
44              {
45                  minCost[nextNode] = nextCost;
46                  pq.push({nextCost, nextNode});
47              }
48          }
49      }
50
51      return totalCost;
52  }
53
54  vector<vector<Edge>> createGraph(int vertices, int edges)
55  {
56      vector<vector<Edge>> graph(vertices);
57
58      for (int i = 1; i < vertices; i++)
59      {
60          int connectTo = rand() % i;
61          double cost = (rand() % 100) + 1;
62          graph[i].push_back({connectTo, cost});
63          graph[connectTo].push_back({i, cost});
64      }
65
66      int addedEdges = vertices - 1;
67      while (addedEdges < edges)
68      {
69          int u = rand() % vertices;
70          int v = rand() % vertices;
```

PrimsAlgorithm.cpp

PrimsAlgorithm.cpp > main()

```
54  vector<vector<Edge>> createGraph(int vertices, int edges)
67      while (addedEdges < edges)
72          if (u != v)
73          {
74              bool edgeExists = false;
75              for (auto &edge : graph[u])
76              {
77                  if (edge.node == v)
78                  {
79                      edgeExists = true;
80                      break;
81                  }
82              }
83              if (!edgeExists)
84              {
85                  double cost = (rand() % 100) + 1;
86                  graph[u].push_back({v, cost});
87                  graph[v].push_back({u, cost});
88                  addedEdges++;
89              }
90          }
91      }
92
93      return graph;
94  }
```

PrimsAlgorithm.cpp

PrimsAlgorithm.cpp > createGraph(int, int)

```cpp
int main()
{
    srand(time(nullptr));
    int vertexCounts[] = {100, 500, 1000, 5000, 10000};
    double edgeMultiplier = 3;

    for (int V : vertexCounts)
    {
        int E = V * edgeMultiplier;
        auto graph = createGraph(V, E);

        auto start = high_resolution_clock::now();
        double mstWeight = primMST(graph, V);
        auto end = high_resolution_clock::now();

        double elapsed = duration<double>(end - start).count();

        cout << "Vertices: " << V << "\n";
        cout << "Edges: " << E << "\n";
        cout << "MST Weight: " << mstWeight << "\n";
        cout << "Time: " << elapsed << " seconds\n";
        cout << "------------------------------\n";
    }

    return 0;
}
```

Output:

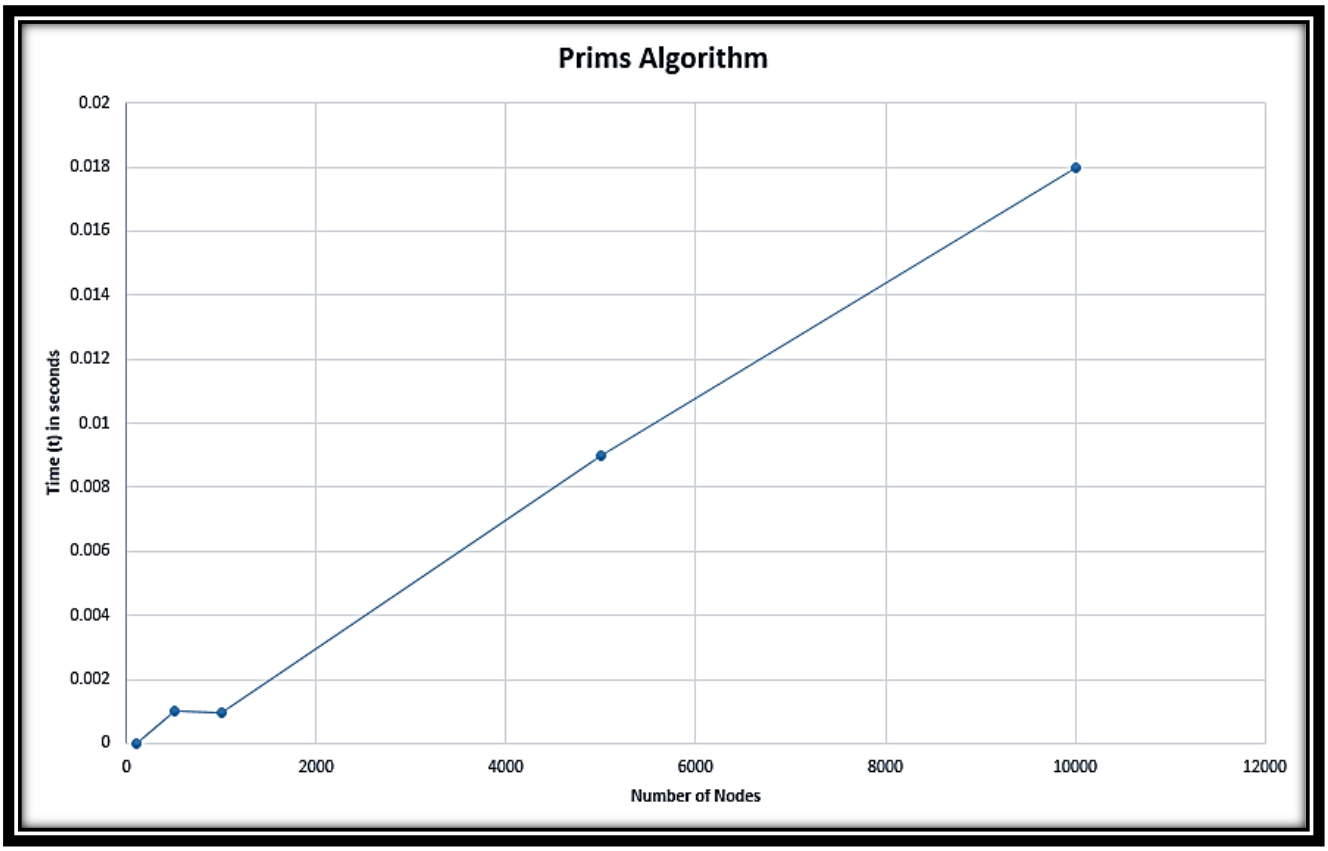
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER

```
d:\Kunsh DAA Lab>cd "d:\Kunsh DAA Lab\" && g++ PrimsAlgorithm.cpp -o PrimsAlgorithm && "d:\Kunsh DAA Lab\"PrimsAlgorithm
Vertices: 100
Edges: 300
MST Weight: 2141
Time: 0 seconds
------------------------------
Vertices: 500
Edges: 1500
MST Weight: 10606
Time: 0.001 seconds
------------------------------
Vertices: 1000
Edges: 3000
MST Weight: 20937
Time: 0.000984 seconds
------------------------------
Vertices: 5000
Edges: 15000
MST Weight: 100332
Time: 0.008969 seconds
------------------------------
Vertices: 10000
Edges: 30000
MST Weight: 201950
Time: 0.018003 seconds
------------------------------
```

Input (n) vs Time (t) Table:

Number of Nodes	Time (t) in seconds
100	0
500	0.001
1000	0.000984
5000	0.008969
10000	0.018003

Graph:



Time Complexity:

	Best	Worst	Average
Prims Algorithm			

Learning Outcome:

EXPERIMENT 8

Problem statement: Write a program to find minimum spanning tree using Kruskal's Algorithm.

Theory:

Algorithm:

Source Code:

KruskalsAlgorithm.cpp

KruskalsAlgorithm.cpp > DisjointSet > unionSets(int, int)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <chrono>
#include <limits>
using namespace std;
using namespace chrono;

struct Edge
{
    int u, v;
    double cost;

    bool operator<(const Edge &other) const
    {
        return cost < other.cost;
    }
};

class DisjointSet
{
public:
    DisjointSet(int n)
    {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i)
            parent[i] = i;
    }

    int find(int u)
    {
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }
```

KruskalsAlgorithm.cpp

KruskalsAlgorithm.cpp > DisjointSet > DisjointSet(int)

```
21  class DisjointSet
39      void unionSets(int u, int v)
40      {
41          int rootU = find(u);
42          int rootV = find(v);
43
44          if (rootU != rootV)
45          {
46              if (rank[rootU] > rank[rootV])
47                  parent[rootV] = rootU;
48              else if (rank[rootU] < rank[rootV])
49                  parent[rootU] = rootV;
50              else
51              {
52                  parent[rootV] = rootU;
53                  rank[rootU]++;
54              }
55          }
56      }
57
58  private:
59      vector<int> parent, rank;
60  };
61
62  double kruskalMST(const vector<Edge> &edges, int vertices)
63  {
64      DisjointSet ds(vertices);
65      double totalCost = 0;
66
67      vector<Edge> sortedEdges = edges;
68      sort(sortedEdges.begin(), sortedEdges.end());
69
70      for (const auto &edge : sortedEdges)
71      {
72          int u = edge.u, v = edge.v;
73          double cost = edge.cost;
```

KruskalsAlgorithm.cpp

KruskalsAlgorithm.cpp > DisjointSet > DisjointSet(int)

```
62  double kruskalMST(const vector<Edge> &edges, int vertices)
70      for (const auto &edge : sortedEdges)
75          if (ds.find(u) != ds.find(v))
76          {
77              ds.unionSets(u, v);
78              totalCost += cost;
79          }
80      }
81
82      return totalCost;
83  }
84
85  vector<Edge> createGraph(int vertices, int edges)
86  {
87      vector<Edge> graph;
88
89      for (int i = 1; i < vertices; i++)
90      {
91          int connectTo = rand() % i;
92          double cost = (rand() % 100) + 1;
93          graph.push_back({i, connectTo, cost});
94          graph.push_back({connectTo, i, cost});
95      }
96
97      int addedEdges = vertices - 1;
98      while (addedEdges < edges)
99      {
100         int u = rand() % vertices;
101         int v = rand() % vertices;
102
103         if (u != v)
104         {
105             bool edgeExists = false;
106             for (auto &edge : graph)
107             {
108                 if ((edge.u == u && edge.v == v) || (edge.u == v && edge.v == u))
109                 {
110                     edgeExists = true;
111                     break;
```

KruskalsAlgorithm.cpp

KruskalsAlgorithm.cpp > DisjointSet > DisjointSet(int)

```
85   vector<Edge> createGraph(int vertices, int edges)
98       while (addedEdges < edges)
103          if (u != v)
106              for (auto &edge : graph)
108                  if ((edge.u == u && edge.v == v) || (edge.u == v && edge.v == u))
112                  }
113              }
114              if (!edgeExists)
115              {
116                  double cost = (rand() % 100) + 1;
117                  graph.push_back({u, v, cost});
118                  graph.push_back({v, u, cost});
119                  addedEdges++;
120              }
121          }
122      }
123
124      return graph;
125  }
126
127  int main()
128  {
129      srand(time(nullptr));
130      int vertexCounts[] = {100, 500, 1000, 5000, 10000};
131      double edgeMultiplier = 3;
132
133      for (int V : vertexCounts)
134      {
135          int E = V * edgeMultiplier;
136          auto graph = createGraph(V, E);
137
138          auto start = high_resolution_clock::now();
139          double mstWeight = kruskalMST(graph, V);
140          auto end = high_resolution_clock::now();
141
142          double elapsed = duration<double>(end - start).count();
```

KruskalsAlgorithm.cpp

KruskalsAlgorithm.cpp > main()

```
127  int main()
133      for (int V : vertexCounts)
142          double elapsed = duration<double>(end - start).count();
143
144          cout << "Vertices: " << V << "\n";
145          cout << "Edges: " << E << "\n";
146          cout << "MST Weight: " << mstWeight << "\n";
147          cout << "Time: " << elapsed << " seconds\n";
148          cout << "-----------------------------\n";
149      }
150
151      return 0;
152  }
```

Output:

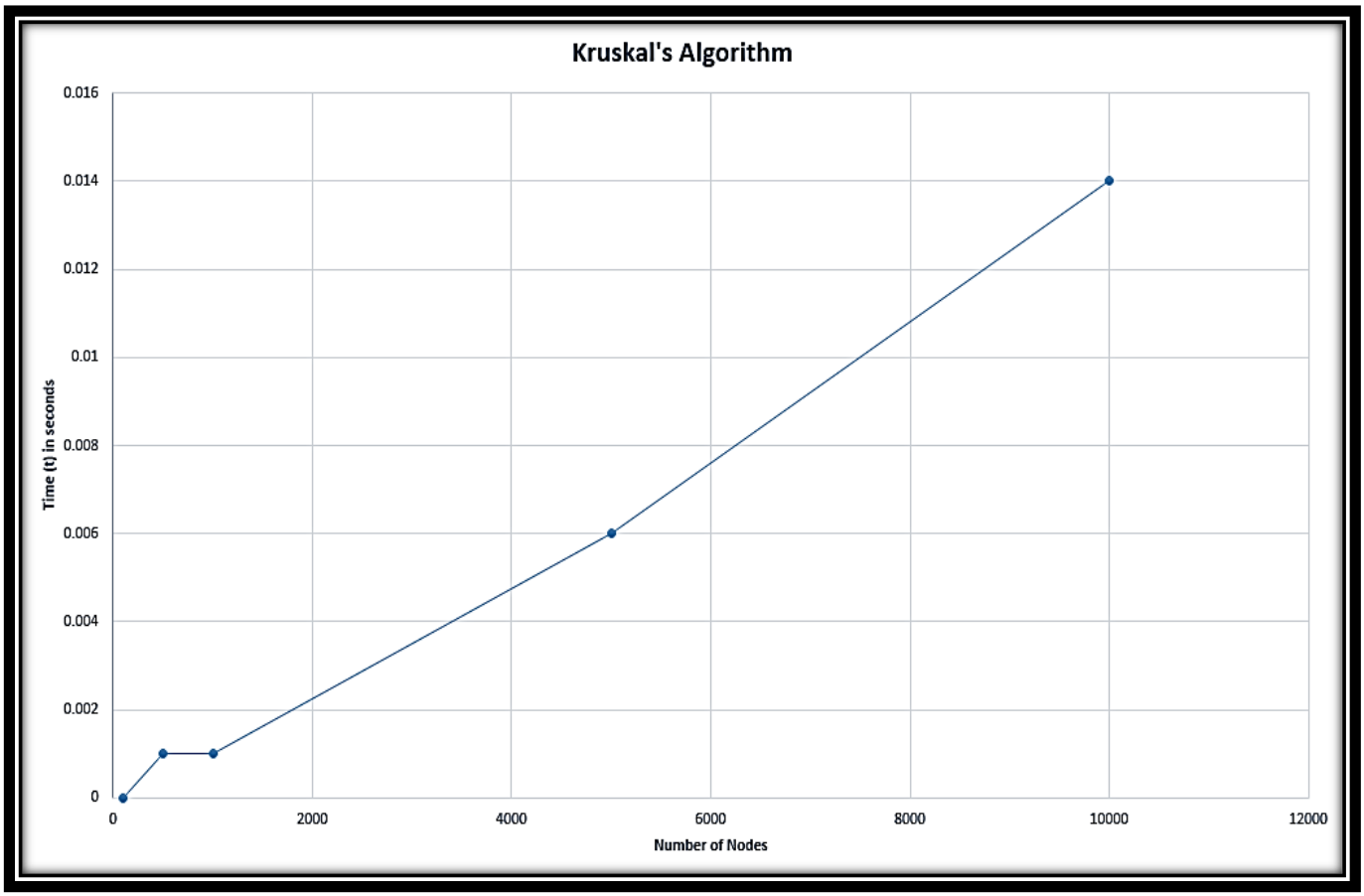
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 2

```
d:\Kunsh DAA Lab>cd "d:\Kunsh DAA Lab\" && g++ KruskalsAlgorithm.cpp -o KruskalsAlgorithm && "d:\Kunsh DAA Lab\"KruskalsAlgorithm
Vertices: 100
Edges: 300
MST Weight: 1671
Time: 0 seconds
-----------------------------
Vertices: 500
Edges: 1500
MST Weight: 10218
Time: 0.001 seconds
-----------------------------
Vertices: 1000
Edges: 3000
MST Weight: 20437
Time: 0.000999 seconds
-----------------------------
Vertices: 5000
Edges: 15000
MST Weight: 103117
Time: 0.006012 seconds
-----------------------------
Vertices: 10000
Edges: 30000
MST Weight: 210141
Time: 0.014 seconds
-----------------------------

d:\Kunsh DAA Lab>
```

Input (n) vs Time (t) Table:

Number of Nodes	Time (t) in seconds
100	0
500	0.001
1000	0.000999
5000	0.006012
10000	0.014

Graph:



Time Complexity:

	Best	Worst	Average
Kruskal's Algorithm			

Learning Outcome:

EXPERIMENT 9

Problem statement: Write a program to perform Single Source Shortest Path problem using Dijkstra's Algorithm.

Theory:

Algorithm:

Source Code:

DijkstraAlgorithm.cpp

DijkstraAlgorithm.cpp > createAdjList(int, int)

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <ctime>
#include <chrono>
#include <limits>
using namespace std;
using namespace chrono;

struct Edge
{
    int v;
    double cost;
};

// Create an adjacency list for an undirected graph
vector<vector<Edge>> createAdjList(int vertices, int edges)
{
    vector<vector<Edge>> adj(vertices);

    // First create a connected tree (like in Kruskal)
    for (int i = 1; i < vertices; i++)
    {
        int connectTo = rand() % i;
        double cost = (rand() % 100) + 1;
        adj[i].push_back({connectTo, cost});
        adj[connectTo].push_back({i, cost});
    }

    int addedEdges = vertices - 1;
    while (addedEdges < edges)
    {
        int u = rand() % vertices;
        int v = rand() % vertices;

        if (u != v)
        {
            bool exists = false;
            for (const auto &e : adj[u])
            {
                if (e.v == v)
                {
                    exists = true;
```

DijkstraAlgorithm.cpp

DijkstraAlgorithm.cpp > createAdjList(int, int)

```
17  vector<vector<Edge>> createAdjList(int vertices, int edges)
31      while (addedEdges < edges)
36          if (u != v)
39              for (const auto &e : adj[u])
41                  if (e.v == v)
44                      break;
45                  }
46              }
47              if (!exists)
48              {
49                  double cost = (rand() % 100) + 1;
50                  adj[u].push_back({v, cost});
51                  adj[v].push_back({u, cost});
52                  addedEdges++;
53              }
54          }
55      }
56
57      return adj;
58  }
59
60  // Dijkstra's algorithm from source to all nodes
61  vector<double> dijkstra(const vector<vector<Edge>> &adj, int source)
62  {
63      int n = adj.size();
64      vector<double> dist(n, numeric_limits<double>::infinity());
65      priority_queue<pair<double, int>, vector<pair<double, int>>, greater<pair<double, int>>> pq;
66
67      dist[source] = 0;
68      pq.push(make_pair(0.0, source));
69
70      while (!pq.empty())
71      {
72          pair<double, int> top = pq.top();
73          pq.pop();
74
75          double d = top.first;
76          int u = top.second;
77
78          if (d > dist[u])
79              continue;
80
81          for (const auto &edge : adj[u])
```

DijkstraAlgorithm.cpp

DijkstraAlgorithm.cpp > createAdjList(int, int)

```
61  vector<double> dijkstra(const vector<vector<Edge>> &adj, int source)
70      while (!pq.empty())
81          for (const auto &edge : adj[u])
82          {
83              int v = edge.v;
84              double cost = edge.cost;
85
86              if (dist[u] + cost < dist[v])
87              {
88                  dist[v] = dist[u] + cost;
89                  pq.push(make_pair(dist[v], v));
90              }
91          }
92      }
93
94      return dist;
95  }
96
97  int main()
98  {
99      srand(time(nullptr));
100     int vertexCounts[] = {500, 1000, 5000, 10000, 50000};
101     double edgeMultiplier = 3;
102
103     for (int V : vertexCounts)
104     {
105         int E = V * edgeMultiplier;
106         auto adj = createAdjList(V, E);
```

DijkstraAlgorithm.cpp

DijkstraAlgorithm.cpp > createAdjList(int, int)

```
97   int main()
102  
103      for (int V : vertexCounts)
104      {
105          int E = V * edgeMultiplier;
106          auto adj = createAdjList(V, E);
107  
108          auto start = high_resolution_clock::now();
109          vector<double> distances = dijkstra(adj, 0); // From source node 0
110          auto end = high_resolution_clock::now();
111  
112          // Total distance from source to all reachable nodes
113          double totalDistance = 0;
114          for (double d : distances)
115          {
116              if (d < numeric_limits<double>::infinity())
117                  totalDistance += d;
118          }
119  
120          double elapsed = duration<double>(end - start).count();
121  
122          cout << "Vertices: " << V << "\n";
123          cout << "Edges: " << E << "\n";
124          cout << "Total Distance from Source (0): " << totalDistance << "\n";
125          cout << "Time: " << elapsed << " seconds\n";
126          cout << "----------------------------\n";
127      }
128  
129      return 0;
130  }
131
```

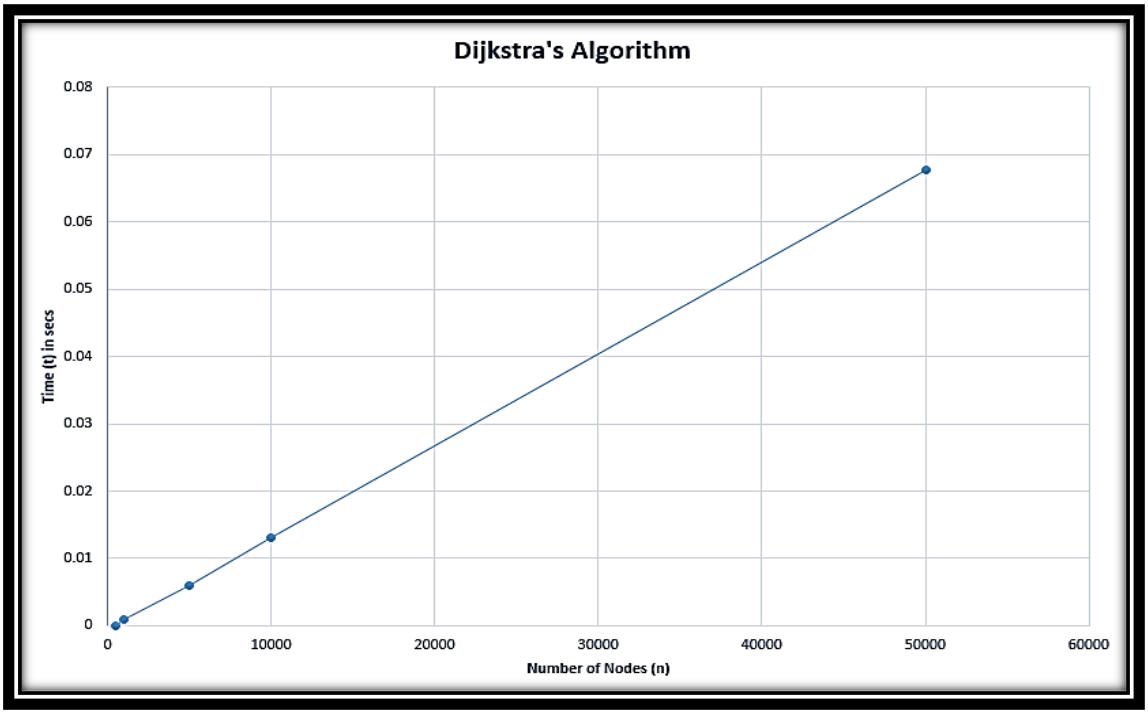
Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER 1

d:\Kunsh DAA Lab>cd "d:\Kunsh DAA Lab\" && g++ DijkstraAlgorithm.cpp -o DijkstraAlgorithm && "d:\Kunsh DAA Lab\"DijkstraAlgorithm
Vertices: 500
Edges: 1500
Total Distance from Source (0): 50259
Time: 0 seconds
------------------------------
Vertices: 1000
Edges: 3000
Total Distance from Source (0): 116638
Time: 0.000996 seconds
------------------------------
Vertices: 5000
Edges: 15000
Total Distance from Source (0): 626977
Time: 0.005989 seconds
------------------------------
Vertices: 10000
Edges: 30000
Total Distance from Source (0): 1.19357e+006
Time: 0.013027 seconds
------------------------------
Vertices: 50000
Edges: 150000
Total Distance from Source (0): 6.85657e+006
Time: 0.067733 seconds
------------------------------

d:\Kunsh DAA Lab>
```

Input (n) vs Time (t) Table:

Number of Nodes	Time (t) in seconds
500	0
1000	0.000996
5000	0.005989
10000	0.013027
50000	0.067733

Graph:



Time Complexity:

	Best	Worst	Average
Dijkstra's Algorithm			

Learning Outcome:

EXPERIMENT 10

Problem statement: Write a program to perform All Pair Shortest Path Algorithm using Floyd Warshall Algorithm.

Theory:

Algorithm:

Source Code:

FloydWarshall.cpp

FloydWarshall.cpp > main()

```cpp
#include <iostream>
#include <vector>
#include <ctime>
#include <chrono>
#include <limits>
using namespace std;
using namespace chrono;

struct Edge
{
    int v;
    double cost;
};

vector<vector<Edge>> createAdjList(int vertices, int edges)
{
    vector<vector<Edge>> adj(vertices);
    for (int i = 1; i < vertices; i++)
    {
        int connectTo = rand() % i;
        double cost = (rand() % 100) + 1;
        adj[i].push_back({connectTo, cost});
        adj[connectTo].push_back({i, cost});
    }
```

FloydWarshall.cpp

FloydWarshall.cpp > main()

```
15  vector<vector<Edge>> createAdjList(int vertices, int edges)
18      for (int i = 1; i < vertices; i++)
25      int addedEdges = vertices - 1;
26      while (addedEdges < edges)
27      {
28          int u = rand() % vertices, v = rand() % vertices;
29          if (u != v)
30          {
31              bool exists = false;
32              for (auto &e : adj[u])
33                  if (e.v == v)
34                  {
35                      exists = true;
36                      break;
37                  }
38              if (!exists)
39              {
40                  double cost = (rand() % 100) + 1;
41                  adj[u].push_back({v, cost});
42                  adj[v].push_back({u, cost});
43                  addedEdges++;
44              }
45          }
46      }
47      return adj;
48  }
49
50  vector<vector<double>> createAdjMatrix(const vector<vector<Edge>> &adj)
51  {
52      int n = adj.size();
53      vector<vector<double>> dist(n, vector<double>(n, numeric_limits<double>::infinity()));
54      for (int i = 0; i < n; ++i)
55      {
56          dist[i][i] = 0;
57          for (auto &e : adj[i])
58              dist[i][e.v] = e.cost;
59      }
60      return dist;
61  }
```

FloydWarshall.cpp

FloydWarshall.cpp > main()

```cpp
void floydWarshall(vector<vector<double>> &dist)
{
    int n = dist.size();
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
}

int main()
{
    srand(time(nullptr));
    int vertexCounts[] = {100, 200, 300, 400, 500};
    double edgeMultiplier = 3;

    for (int V : vertexCounts)
    {
        int E = V * edgeMultiplier;
        auto adj = createAdjList(V, E);
        auto dist = createAdjMatrix(adj);

        auto start = high_resolution_clock::now();
        floydWarshall(dist);
        auto end = high_resolution_clock::now();

        double totalDistance = 0;
        int count = 0;
        for (int i = 0; i < V; ++i)
            for (int j = 0; j < V; ++j)
                if (i != j && dist[i][j] < numeric_limits<double>::infinity())
                {
                    totalDistance += dist[i][j];
                    count++;
                }

        double elapsed = duration<double>(end - start).count();
        cout << "Vertices: " << V << "\nEdges: " << E
            << "\nTotal Distance: " << totalDistance
            << "\nAverage Distance: " << (count ? totalDistance / count : 0)
            << "\nTime: " << elapsed << " seconds\n----------------------------\n";
    }
}
```

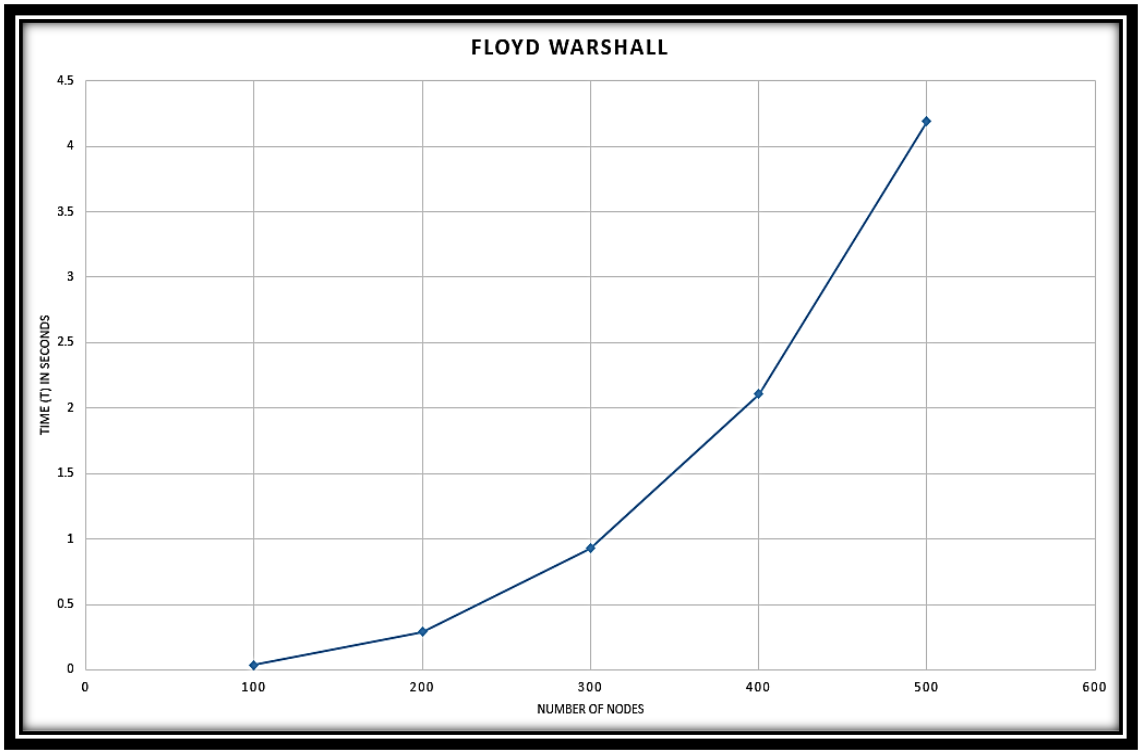
Output:

```
d:\Kunsh DAA Lab>cd "d:\Kunsh DAA Lab\" && g++ FloydWarshall.cpp -o FloydWarshall && "d:\Kunsh DAA Lab\"FloydWarshall
Vertices: 100
Edges: 300
Total Distance: 1.01053e+006
Average Distance: 102.074
Time: 0.035929 seconds
----------------------------
Vertices: 200
Edges: 600
Total Distance: 3.6426e+006
Average Distance: 91.5227
Time: 0.288924 seconds
----------------------------
Vertices: 300
Edges: 900
Total Distance: 9.34059e+006
Average Distance: 104.131
Time: 0.929151 seconds
----------------------------
Vertices: 400
Edges: 1200
Total Distance: 1.76478e+007
Average Distance: 110.575
Time: 2.10865 seconds
----------------------------
Vertices: 500
Edges: 1500
Total Distance: 2.90263e+007
Average Distance: 116.338
Time: 4.18993 seconds
----------------------------

d:\Kunsh DAA Lab>
```

Input (n) vs Time (t) Table:

Number of Nodes	Time (t) in seconds
100	0.035929
200	0.288924
300	0.929151
400	2.10865
500	4.18993

Graph:



Time Complexity:

	Best	Worst	Average
Floyd Warshall's Algorithm			

Learning Outcome:

EXPERIMENT 11

Problem statement: Write a program to implement N-Queens's Problem using Backtracking Approach.

Theory:

Algorithm:

Source Code:

N_Queens.cpp

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <unordered_set>
#include <ctime>
using namespace std;

unordered_set<int> cols;
unordered_set<int> posDiag; // (r + c)
unordered_set<int> negDiag; // (r - c)

void backtrack(vector<vector<string>> &result, int n, vector<string> &board, int row)
{
    if (row == n)
    {
        result.push_back(board);
        return;
    }

    for (int col = 0; col < n; col++)
    {
        if (cols.count(col) || posDiag.count(row + col) || negDiag.count(row - col))
            continue;

        cols.insert(col);
        posDiag.insert(row + col);
        negDiag.insert(row - col);
        board[row][col] = 'Q';

        backtrack(result, n, board, row + 1);

        cols.erase(col);
        posDiag.erase(row + col);
        negDiag.erase(row - col);
        board[row][col] = '.';
    }
}
```

N_Queens.cpp

```cpp
vector<vector<string>> solveNQueens(int n)
{
    vector<vector<string>> result;
    vector<string> board(n, string(n, '.'));
    cols.clear();
    posDiag.clear();
    negDiag.clear();
    backtrack(result, n, board, 0);
    return result;
}

int main()
{
    vector<int> sizes = {4, 8, 12};

    for (int n : sizes)
    {
        clock_t start = clock();
        vector<vector<string>> solutions = solveNQueens(n);
        clock_t end = clock();

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

        cout << "N = " << n << "\n";
        cout << "Total Solutions: " << solutions.size() << "\n";
        cout << "Time Taken: " << time_taken << " sec\n";

        if (!solutions.empty())
        {
            cout << "Example Solution:\n";
            for (const string &row : solutions[0])
                cout << row << "\n";
        }
        cout << "----------------------------\n";
    }

    return 0;
}
```

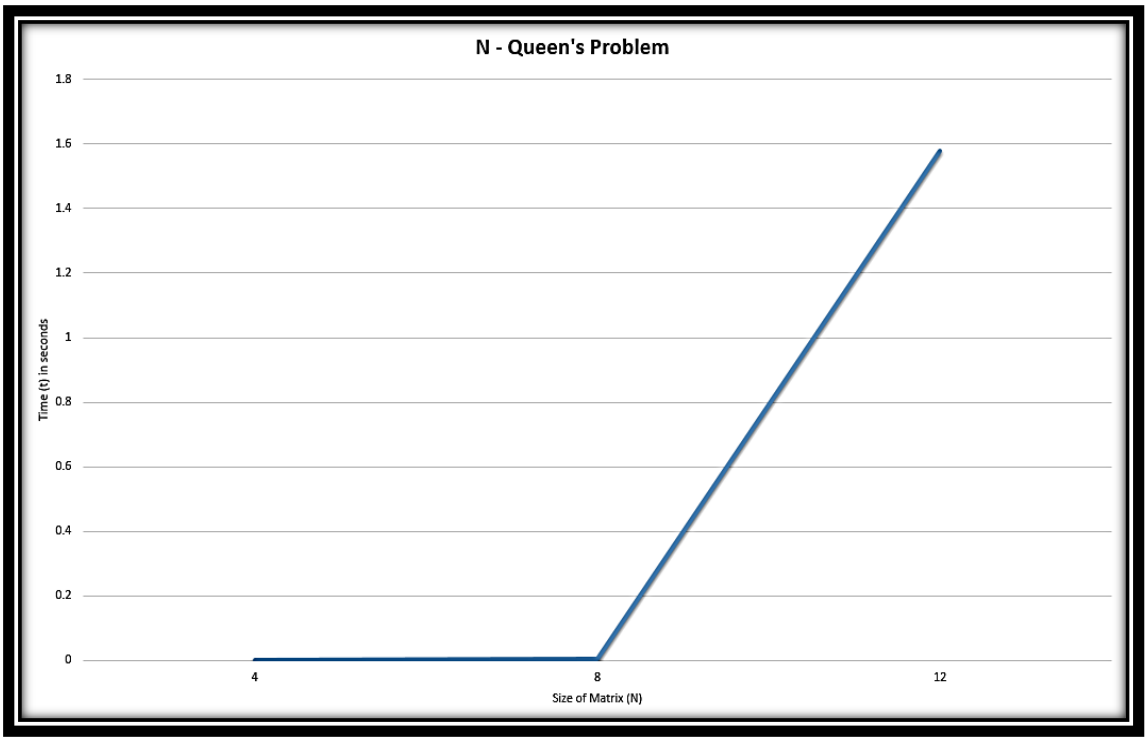
Output:

```
d:\Kunsh DAA Lab>cd "d:\Kunsh DAA Lab\" && g++ N_Queens.cpp -o N_Queens && "d:\Kunsh DAA Lab\"N_Queens
N = 4
Total Solutions: 2
Time Taken: 0 sec
Example Solution:
.Q..
...Q
Q...
..Q.
------------------------------
N = 8
Total Solutions: 92
Time Taken: 0.004 sec
Example Solution:
Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....
------------------------------
N = 12
Total Solutions: 14200
Time Taken: 1.577 sec
Example Solution:
Q...........
..Q.........
....Q.......
.......Q....
.........Q..
...........Q
.....Q......
..........Q.
.Q..........
......Q.....
........Q...
...Q........
------------------------------
```

Input (n) vs Time (t) Table:

Number of Nodes	Time (t) in seconds
4	0
8	0.004
12	1.577

Graph:



Time Complexity:

	Best	Worst	Average
N-Queen's Problem			

Learning Outcome: