Name -Suchana Hazra

Enrollment No-2022CSB102

Software Engineering First Lab

Tower Of Hanoi :

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class TowerOfHanoiRuntime {
    public static void solveHanoi(int n, char source, char target, char auxiliary) {
        if (n == 1) {
            return;
        }
        solveHanoi(n - 1, source, auxiliary, target);
        solveHanoi(n - 1, auxiliary, target, source);
    }

    public static void main(String[] args) {
        // Define the file name and path (current directory)
        String fileName = "tower_of_hanoi_runtime.csv";
        File file = new File(fileName);

        try (FileWriter writer = new FileWriter(file)) {
            // Write header to the CSV file
            writer.append("Disks,ExecutionTime\n");

            // Generate data for disks (n = 1 to 20)
            for (int n = 1; n <= 20; n++) {
                long startTime = System.nanoTime(); // Start timer
                solveHanoi(n, 'A', 'C', 'B'); // Run Tower of Hanoi
                long endTime = System.nanoTime(); // End timer

                long executionTime = endTime - startTime; // Calculate runtime

                // Write the data into the CSV file
                writer.append(n + "," + executionTime + "\n");
            }

            System.out.println("Data successfully written to " + file.getAbsolutePath(
        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

This code will generate the no of moves required to place the disks recursively. For different no of disks the code will generate the runtime using nanoTime function in java and finally store the data in CSV file.

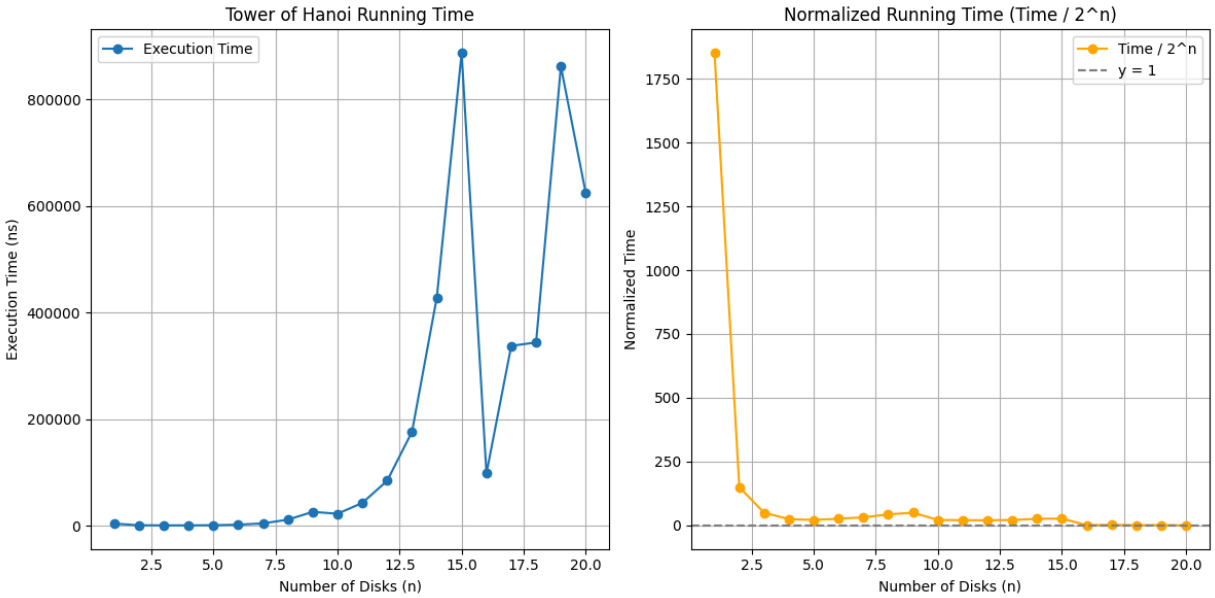
- Tabular Data:

```
Disks,ExecutionTime
1,3700
2,600
3,400
4,400
5,700
6,1700
7,4100
8,11100
9,25800
10,22200
11,42600
12,84200
13,176100
14,428400
15,887500
16,98100
17,337500
18,344000
19,862300
20,625000
```

- Time Complexity Analysis:

Best-case: The best-case occurs when n=1 The function executes a single move O(1)
Average-case: Average-case time complexity is O(2^n) The number of moves doubles with each increment.
Worst-case: The worst-case is the same as the average-case because the recursive process always executes exponential times.

- Plot the Graph:



Dijkstra's Algorithm :

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;

public class DijkstraRuntime {

    // Function to find the vertex with the minimum distance
    static int minDistance(int[] dist, boolean[] sptSet, int V) {
        int min = Integer.MAX_VALUE, minIndex = -1;

        for (int v = 0; v < V; v++) {
            if (!sptSet[v] && dist[v] <= min) {
                min = dist[v];
                minIndex = v;
            }
        }
        return minIndex;
    }

    // Dijkstra's algorithm
    static void dijkstra(int[][] graph, int src, int V) {
        int[] dist = new int[V]; // Distances from source to vertices
        boolean[] sptSet = new boolean[V]; // Shortest path tree set

        // Initialize all distances as INFINITE and sptSet as false
        Arrays.fill(dist, Integer.MAX_VALUE);
        Arrays.fill(sptSet, false);
```

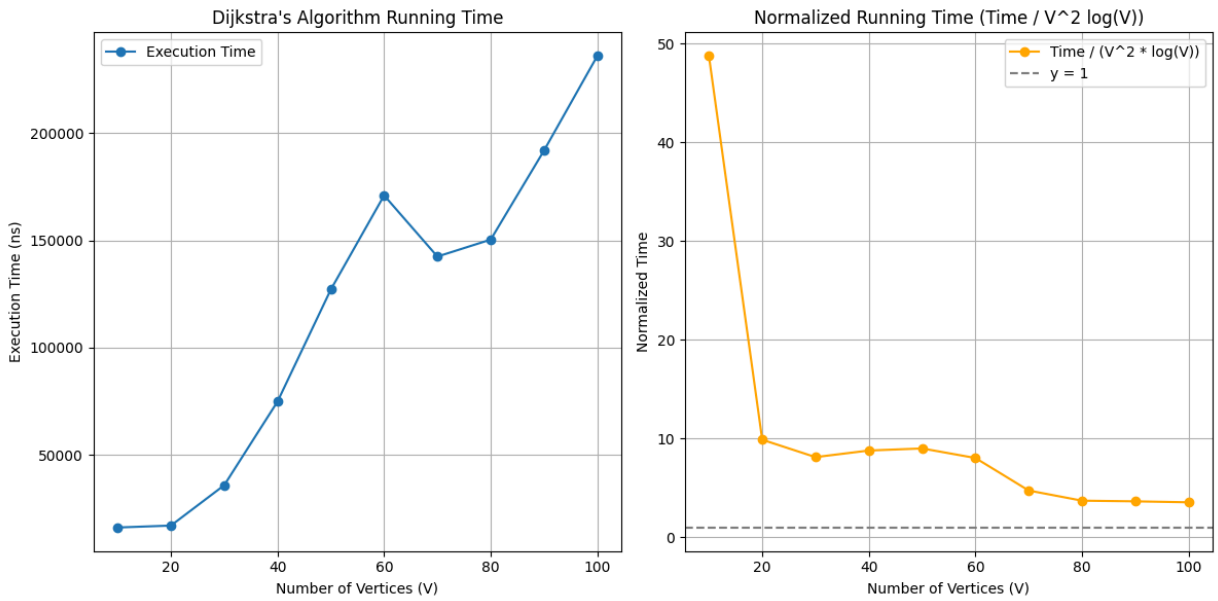
```
        dist[src] = 0; // Distance from source to itself is always 0

        for (int count = 0; count < V - 1; count++) {
            int u = minDistance(dist, sptSet, V); // Pick the minimum distance vertex
            sptSet[u] = true; // Mark the vertex as processed

            for (int v = 0; v < V; v++) {
                if (!sptSet[v] && graph[u][v] != 0 && dist[u] != Integer.MAX_VALUE &&
                    dist[v] = dist[u] + graph[u][v];
                }
            }
        }
    }

    public static void main(String[] args) {
        String fileName = "dijkstra_runtime.csv";

        try (FileWriter writer = new FileWriter(fileName)) {
            // Write header
            writer.append("Vertices,ExecutionTime\n");

            // Generate runtime data for varying graph sizes
            for (int V = 10; V <= 100; V += 10) {
                int[][] graph = new int[V][V];

                // Randomly initialize the adjacency matrix with weights (0 to 10)
                for (int i = 0; i < V; i++) {
                    for (int j = 0; j < V; j++) {
                        graph[i][j] = (i == j) ? 0 : (int) (Math.random() * 10 + 1);
                    }
                }

                long startTime = System.nanoTime(); // Start time
                dijkstra(graph, 0, V); // Run Dijkstra's algorithm
                long endTime = System.nanoTime(); // End time

                long executionTime = endTime - startTime; // Calculate runtime
                writer.append(V + "," + executionTime + "\n"); // Write to CSV
            }

            System.out.println("Data successfully written to " + fileName);

        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

This code represents the graph using adjacency matrix where the cost between each pair of vertex is initialized randomly. This code uses array to find out the minimum distance each time. we can use another data structure min heap or min priority queue to find out min distance which takes less time O(logn) instead of O(n).

- Tabular data:

```
Vertices,ExecutionTime
10,16200
20,17100
30,35900
40,74900
50,127100
60,171000
70,142500
80,150400
90,191900
100,236000
```

- Plot the graph:



- Time Complexity analysis:

Best Case Time Complexity: O((V + E) log V)

> This best-case scenario occurs when using an optimized data structure like a Fibonacci heap for implementing the priority queue. This scenario is typically encountered when the graph is sparse, meaning it has relatively few edges compared to vertices.

Average Case Time Complexity: O((V + E) log V)

> The algorithm efficiently finds shortest paths in graphs with varying densities, finding a balance between the quantity of edges and vertices. In practice, this average complexity is encountered in a wide range of scenarios, making Dijkstra's algorithm a reliable choice for many shortest path problems.

Worst Case Time Complexity: O((V^2) log V)

> The worst-case scenario often arises in fully connected graphs or graphs with many edges between each pair of vertices.

Kruskal's Algorithm:

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;

public class KruskalRuntime {

    // Class for Union-Find (Disjoint Set)
    static class Subset {
        int parent, rank;
    }

    // Find the root of the subset containing a vertex (with path compression)
    static int find(Subset[] subsets, int i) {
        if (subsets[i].parent != i) {
            subsets[i].parent = find(subsets, subsets[i].parent);
        }
        return subsets[i].parent;
    }

    // Union of two subsets by rank
    static void union(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++;
        }
    }

    // Kruskal's algorithm
    static void kruskal(int[][] graph, int V) {
        // Sort all edges by weight
        // In Kruskal's Algorithm, we don't need a complex edge structure.
        // Here, we simply deal with the matrix as the representation of edges.

        // Create a list of edges and sort them based on their weight
        int[] edges = new int[V * (V - 1) / 2]; // To store edge weights
```

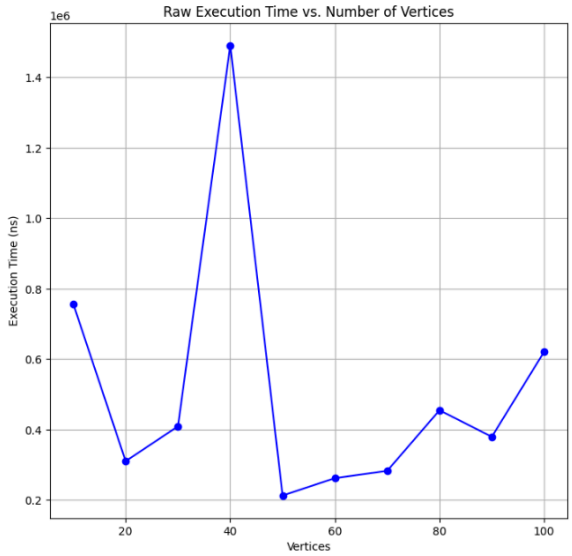
```
        int edgeIndex = 0;
        for (int i = 0; i < V; i++) {
            for (int j = i + 1; j < V; j++) {
                if (graph[i][j] != 0) {
                    edges[edgeIndex++] = graph[i][j]; // Add weight to edge array
                }
            }
        }

        Arrays.sort(edges); // Sort edges by weight

        // Apply union-find algorithm to build MST
        Subset[] subsets = new Subset[V];
        for (int i = 0; i < V; i++) {
            subsets[i] = new Subset();
            subsets[i].parent = i;
            subsets[i].rank = 0;
        }

        int edgeCount = 0;
        for (int i = 0; i < edgeIndex && edgeCount < V - 1; i++) {
            int weight = edges[i];

            // Find the vertices corresponding to the edge weight
            for (int u = 0; u < V; u++) {
                for (int v = u + 1; v < V; v++) {
                    if (graph[u][v] == weight) {
                        int x = find(subsets, u);
                        int y = find(subsets, v);

                        if (x != y) {
                            edgeCount++;
                            union(subsets, x, y);
                        }
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        String fileName = "kruskal_runtime.csv";

        try (FileWriter writer = new FileWriter(fileName)) {
            // Write header
            writer.append("Vertices,Edges,ExecutionTime,NormalizedTime\n");

            // Generate runtime data for varying graph sizes
            for (int V = 10; V <= 100; V += 10) {
                int E = V * (V - 1) / 4; // Randomly assume a sparse graph (~25% edges

                int[][] graph = new int[V][V];
                // Randomly initialize the adjacency matrix with weights (0 to 10)
                for (int i = 0; i < V; i++) {
```

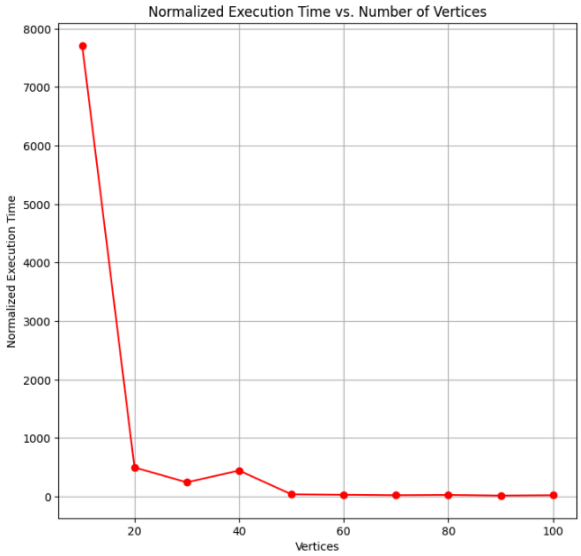
```
                    for (int j = i + 1; j < V; j++) {
                        graph[i][j] = graph[j][i] = (int) (Math.random() * 10 + 1);
                    }
                }

                long startTime = System.nanoTime(); // Start time
                kruskal(graph, V); // Run Kruskal's algorithm
                long endTime = System.nanoTime(); // End time

                long executionTime = endTime - startTime; // Calculate runtime

                // Normalize time by dividing by E * log(E)
                double normalizedFactor = E * Math.log(E) / Math.log(2); // log base 2
                double normalizedTime = executionTime / normalizedFactor;

                // Write data to CSV
                writer.append(V + "," + E + "," + executionTime + "," + normalizedTime
            }

            System.out.println("Data successfully written to " + fileName);

        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

- Tabular data:

```
Vertices,Edges,ExecutionTime,NormalizedTime
10,22,756600,7711.948972864433
20,95,310600,497.64772304350936
30,217,409500,243.13397124858147
40,390,1491000,444.1652386652135
50,612,213100,37.61348452673286
60,885,262400,30.287160338181835
70,1207,283100,22.911366297398384
80,1580,455200,27.113604379516964
90,2002,379800,17.2979279567106
100,2475,621200,22.264274011619776
```

- Plot the graph:





- Time Complexity Analysis:

  Sorting of edges takes O(E * logE) time.

  After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(logV) time.

  So overall complexity is O(E * logE + E * logV) time.

  The value of E can be at most $O(V^2)$, so O(logV) and O(logE) are the same. Therefore, the overall time complexity is O(E * logE) or O(E*logV)

Prims:

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.PriorityQueue;

public class PrimRuntime {

    // Class to represent the graph edge
    static class Edge implements Comparable<Edge> {
        int vertex, weight;

        public Edge(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge other) {
            return this.weight - other.weight;
        }
    }
```

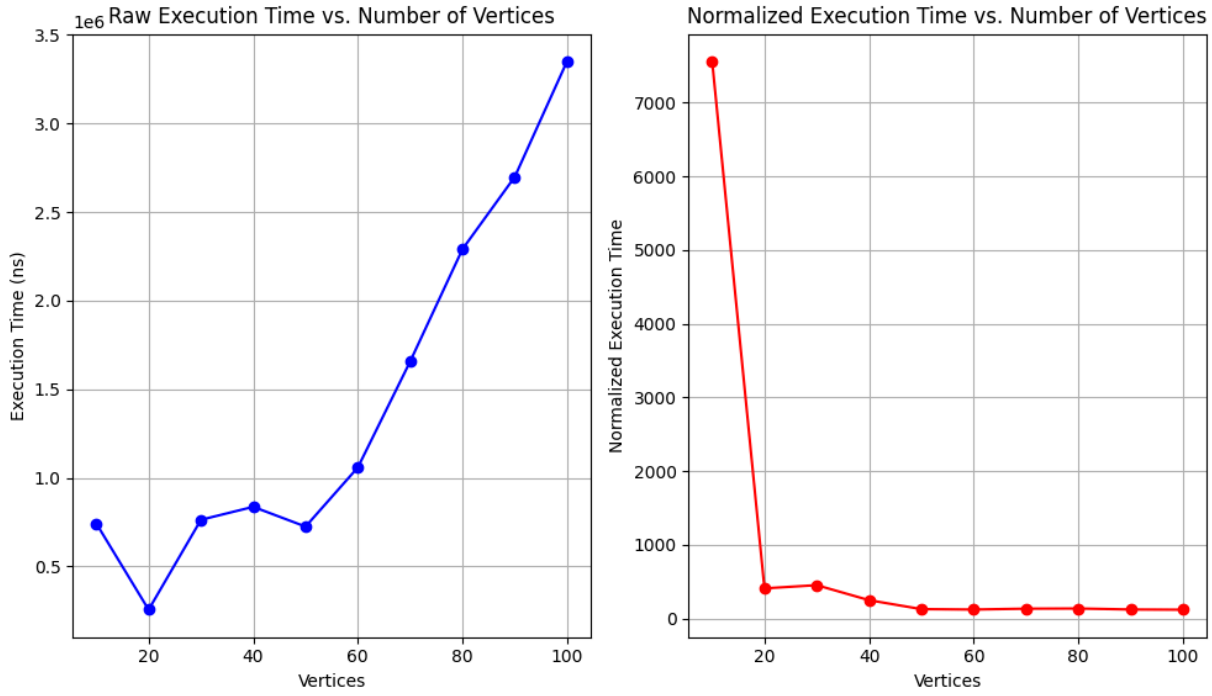
```java
    // Prim's Algorithm
    static void prim(int[][] graph, int V) {
        // Priority queue to pick the minimum weight edge
        PriorityQueue<Edge> pq = new PriorityQueue<>();
        boolean[] inMST = new boolean[V]; // To keep track of vertices already in MST

        // Start from vertex 0
        inMST[0] = true;

        // Add all edges from vertex 0 to priority queue
        for (int i = 0; i < V; i++) {
            if (graph[0][i] != 0) {
                pq.add(new Edge(i, graph[0][i]));
            }
        }

        // MST construction
        while (!pq.isEmpty()) {
            Edge edge = pq.poll();
            int u = edge.vertex;
            int weight = edge.weight;

            if (inMST[u]) continue;

            // Include this edge in MST
            inMST[u] = true;

            // Add adjacent edges to the priority queue
            for (int i = 0; i < V; i++) {
                if (!inMST[i] && graph[u][i] != 0) {
                    pq.add(new Edge(i, graph[u][i]));
                }
            }
        }
    }

    public static void main(String[] args) {
        String fileName = "prim_runtime.csv";

        try (FileWriter writer = new FileWriter(fileName)) {
            // Write header
            writer.append("Vertices,Edges,ExecutionTime,NormalizedTime\n");

            // Generate runtime data for varying graph sizes
            for (int V = 10; V <= 100; V += 10) {
                int E = V * (V - 1) / 4; // Randomly assume a sparse graph (~25% edges

                int[][] graph = new int[V][V];
                // Randomly initialize the adjacency matrix with weights (0 to 10)
                for (int i = 0; i < V; i++) {
                    for (int j = i + 1; j < V; j++) {
                        graph[i][j] = graph[j][i] = (int) (Math.random() * 10 + 1);
                    }
                }
```

```
                long startTime = System.nanoTime(); // Start time
                prim(graph, V); // Run Prim's algorithm
                long endTime = System.nanoTime(); // End time

                long executionTime = endTime - startTime; // Calculate runtime

                // Normalize time by dividing by E * log(E)
                double normalizedFactor = E * Math.log(E) / Math.log(2); // log base 2
                double normalizedTime = executionTime / normalizedFactor;

                // Write data to CSV
                writer.append(V + "," + E + "," + executionTime + "," + normalizedTime
            }

            System.out.println("Data successfully written to " + fileName);

        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

Tabular data:

```
Vertices,Edges,ExecutionTime,NormalizedTime
10,22,742000,7563.13261679277
20,95,255600,409.52594336742106
30,217,762900,452.9594790367346
40,390,836100,249.07213685310867
50,612,724500,127.878787140394
60,885,1058500,122.17591165383183
70,1207,1656500,134.06103239717564
80,1580,2290200,136.4138329305135
90,2002,2698400,122.89818009054207
100,2475,3351000,120.10235385212148
```

Plot the graph:



Time complexity analysis:

> **Adjacency Matrix**: **O(V^2)**
> **Adjacency List + Min-Heap**: **O((V + E) log V)**
> **Fibonacci Heap**: **O(E + V log V)**
> **Space Complexity**: **O(V + E)**

Knapsack Problem:

```java
import java.io.FileWriter;
import java.io.IOException;

public class Knapsack_runtime {

    // Function to solve the 0/1 Knapsack problem using dynamic programming
    static int knapsack(int[] weights, int[] values, int W, int n) {
        int[][] dp = new int[n + 1][W + 1];

        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                if (i == 0 || w == 0) {
                    dp[i][w] = 0;
                } else if (weights[i - 1] <= w) {
                    dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]],
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
```

```
        }
        return dp[n][W];
    }

    public static void main(String[] args) {
        String fileName = "knapsack_runtime.csv";

        try (FileWriter writer = new FileWriter(fileName)) {
            // Write header
            writer.append("Items,Capacity,ExecutionTime,NormalizedTime\n");

            // Generate runtime data for varying numbers of items and capacity
            for (int n = 10; n <= 100; n += 10) {
                for (int W = 50; W <= 500; W += 50) {
                    // Generate random weights and values for items
                    int[] weights = new int[n];
                    int[] values = new int[n];
                    for (int i = 0; i < n; i++) {
                        weights[i] = (int) (Math.random() * 20 + 1);  // Random weight
                        values[i] = (int) (Math.random() * 100 + 1);  // Random value
                    }

                    long startTime = System.nanoTime(); // Start time
                    knapsack(weights, values, W, n);  // Solve Knapsack Problem
                    long endTime = System.nanoTime(); // End time

                    long executionTime = endTime - startTime; // Calculate runtime

                    // Normalize time by dividing by n * W
                    double normalizedFactor = n * W;
                    double normalizedTime = executionTime / normalizedFactor;

                    // Write data to CSV
                    writer.append(n + "," + W + "," + executionTime + "," + normalized
                }
            }

            System.out.println("Data successfully written to " + fileName);

        } catch (IOException e) {
            System.out.println("An error occurred while writing to the file.");
            e.printStackTrace();
        }
    }
}
```

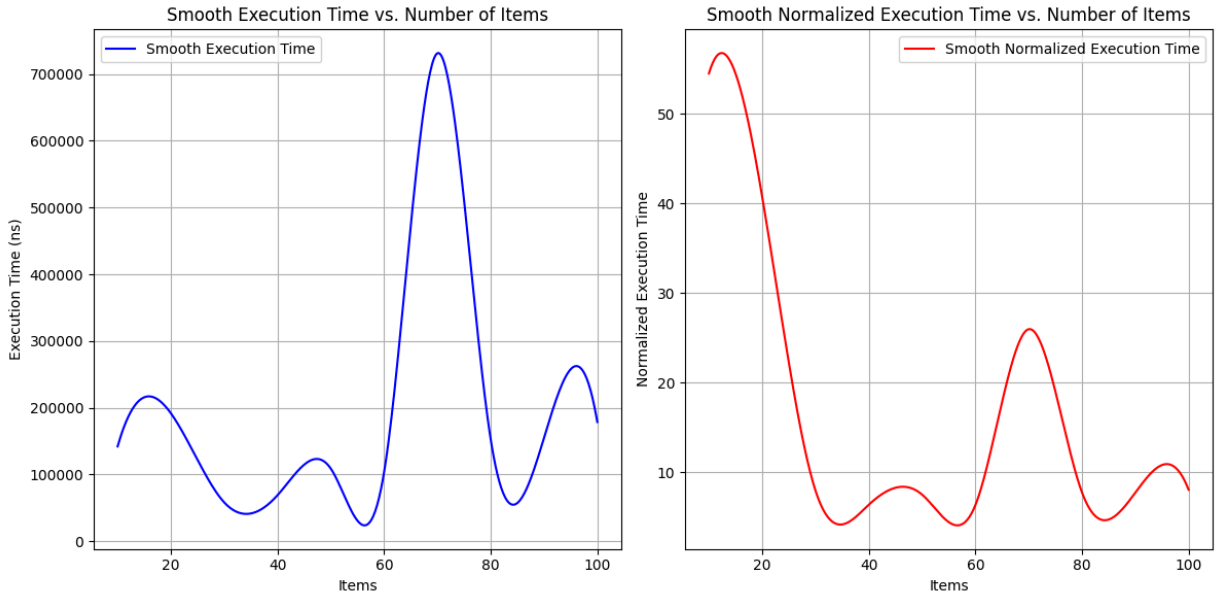
Tabular Data:

```
Items,Capacity,ExecutionTime,NormalizedTime
10,50,38700,77.4
10,100,59100,59.1
10,150,75900,50.6
```

```
10,200,106300,53.15
10,250,131200,52.48
10,300,152900,50.96666666666667
10,350,175000,50.0
10,400,203100,50.775
10,450,227700,50.6
10,500,251600,50.32
20,50,47200,47.2
20,100,97900,48.95
20,150,149900,49.96666666666667
20,200,200600,50.15
20,250,240600,48.12
20,300,299200,49.86666666666667
20,350,344700,49.24285714285714
20,400,384700,48.0875
20,450,61500,6.833333333333333
20,500,90900,9.09
30,50,16600,11.066666666666666
30,100,30800,10.266666666666667
30,150,31700,7.044444444444444
30,200,74900,12.483333333333333
30,250,49900,6.653333333333333
30,300,62900,6.988888888888889
30,350,65200,6.20952380952381
30,400,74700,6.225
30,450,84900,6.288888888888889
30,500,90900,6.06
40,50,13900,6.95
40,100,25700,6.425
40,150,37700,6.283333333333333
40,200,52800,6.6
40,250,61500,6.15
40,300,73900,6.158333333333333
40,350,89100,6.364285714285714
40,400,102400,6.4
40,450,109100,6.061111111111111
40,500,124200,6.21
50,50,16900,6.76
50,100,33000,6.6
50,150,46400,6.1866666666666665
50,200,61200,6.12
50,250,84300,6.744
50,300,91100,6.073333333333333
50,350,176200,10.06857142857143
50,400,229400,11.47
50,450,189300,8.413333333333334
50,500,160900,6.436
60,50,19500,6.5
60,100,37000,6.166666666666667
60,150,60100,6.677777777777778
60,200,74900,6.241666666666666
60,250,92400,6.16
60,300,116300,6.461111111111111
60,350,130600,6.219047619047619
60,400,155200,6.466666666666667
```

```
60,450,163400,6.051851851851852
60,500,187900,6.263333333333334
70,50,22500,6.428571428571429
70,100,43200,6.171428571428572
70,150,66100,6.295238095238095
70,200,85800,6.128571428571429
70,250,111200,6.354285714285714
70,300,234900,11.185714285714285
70,350,262300,10.706122448979592
70,400,271600,9.7
70,450,6033500,191.53968253968253
70,500,180900,5.168571428571428
80,50,33400,8.35
80,100,105200,13.15
80,150,113500,9.458333333333334
80,200,135200,8.45
80,250,134100,6.705
80,300,148300,6.179166666666666
80,350,189900,6.7821428571428575
80,400,161500,5.046875
80,450,203900,5.663888888888889
80,500,303000,7.575
90,50,51200,11.377777777777778
90,100,143000,15.88888888888889
90,150,169200,12.533333333333333
90,200,102100,5.6722222222222225
90,250,104000,4.622222222222222
90,300,116900,4.32962962962963
90,350,181800,5.771428571428571
90,400,205500,5.708333333333333
90,450,227000,5.604938271604938
90,500,247500,5.5
100,50,86300,17.26
100,100,116400,11.64
100,150,127100,8.473333333333333
100,200,150700,7.535
100,250,169400,6.776
100,300,180100,6.003333333333333
100,350,206300,5.894285714285714
100,400,236400,5.91
100,450,244000,5.4222222222222225
100,500,271100,5.422
```

Plot the graph:



Time Complexity analysis:

The time complexity is O(n×W) ,where `n` is the number of items and `W` is the maximum weight of the knapsack.

Floyd-Warshall algorithm:

```
import java.io.FileWriter;
import java.io.IOException;

public class Floyd_warshall_runtime {

    static final int INF = 99999;

    // Floyd-Warshall algorithm
    public static void floydWarshall(int[][] graph, int V) {
        int[][] dist = new int[V][V];
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                dist[i][j] = graph[i][j];
            }
        }

        for (int k = 0; k < V; k++) {
            for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                    if (dist[i][k] + dist[k][j] < dist[i][j]) {
                        dist[i][j] = dist[i][k] + dist[k][j];
                    }
                }
            }
        }
```

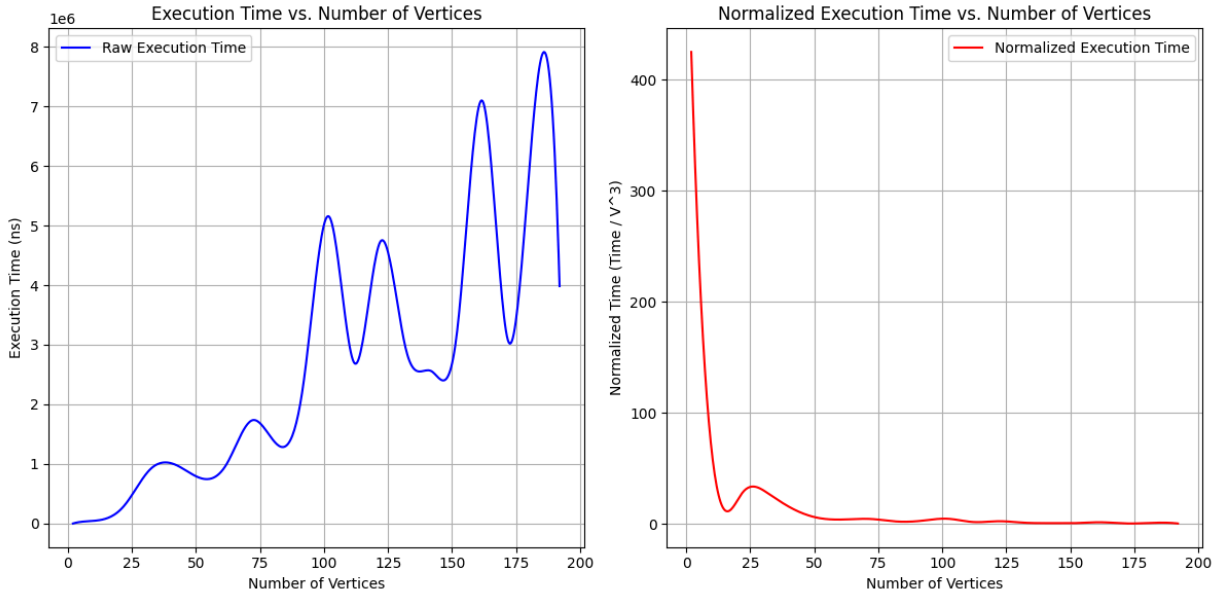
```
    }

    public static void main(String[] args) {
        int[][] graph;
        int V;

        try (FileWriter writer = new FileWriter("floyd_warshall_runtime.csv")) {
            writer.write("Vertices,ExecutionTime(ns)\n");

            for (V = 2; V <= 200; V += 10) { // Varying number of vertices
                graph = new int[V][V];

                // Fill graph with random weights and infinity
                for (int i = 0; i < V; i++) {
                    for (int j = 0; j < V; j++) {
                        if (i == j) {
                            graph[i][j] = 0;
                        } else {
                            graph[i][j] = (int) (Math.random() * 50) + 1; // Random we
                        }
                    }
                }

                // Measure execution time
                long start = System.nanoTime();
                floydWarshall(graph, V);
                long end = System.nanoTime();

                long executionTime = end - start;
                writer.write(V + "," + executionTime + "\n");
                System.out.println("Vertices: " + V + ", ExecutionTime: " + executionT
            }

            System.out.println("Execution times written to floyd_warshall_runtime.csv'
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Tabular Data:

```
Vertices,ExecutionTime(ns)
2,3400
12,57300
22,301600
32,905100
42,989100
52,762000
62,999800
72,1734000
82,1318500
92,2353800
102,5150000
```

```
112,2687100
122,4728200
132,2923700
142,2556300
152,3227200
162,7082400
172,3053900
182,6867700
192,3983100
```

Plot the Graph:



Time Complexity Analysis:
Best-case, Average-case, Worst-case : all are O(V^3) where V is the no of vertices. Regardless of input configuration, Floyd-Warshall evaluates all triplets of vertices, making its time complexity cubic.

Merge Sort Algorithm:

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class mergeSort_runtime {

    // MergeSort implementation
    public static void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;

            // Sort first and second halves
            mergeSort(array, left, mid);
            mergeSort(array, mid + 1, right);
```

```
            // Merge the sorted halves
            merge(array, left, mid, right);
        }
    }

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

        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];

        System.arraycopy(array, left, leftArray, 0, n1);
        System.arraycopy(array, mid + 1, rightArray, 0, n2);

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

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

        while (i < n1) {
            array[k++] = leftArray[i++];
        }

        while (j < n2) {
            array[k++] = rightArray[j++];
        }
    }

    public static void main(String[] args) {
        Random random = new Random();

        try (FileWriter writer = new FileWriter("mergesort_cases_runtime.csv")) {
            writer.write("ArraySize,Case,ExecutionTime(ns)\n");

            for (int size = 1000; size <= 10000; size += 1000) {
                int[] array = new int[size];

                // Best case (already sorted)
                for (int i = 0; i < size; i++) {
                    array[i] = i;
                }
                long start = System.nanoTime();
                mergeSort(array, 0, array.length - 1);
                long end = System.nanoTime();
                writer.write(size + ",BestCase," + (end - start) + "\n");

                // Average case (random)
                for (int i = 0; i < size; i++) {
```

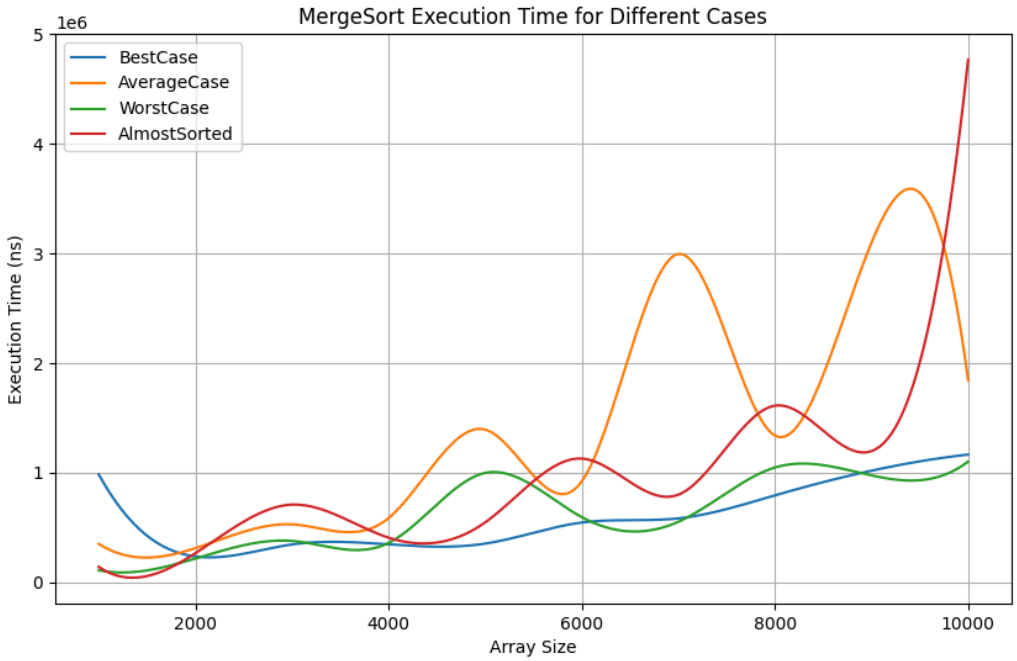
```
                    array[i] = random.nextInt(10000);
                }
                start = System.nanoTime();
                mergeSort(array, 0, array.length - 1);
                end = System.nanoTime();
                writer.write(size + ",AverageCase," + (end - start) + "\n");

                // Worst case (reverse sorted)
                for (int i = 0; i < size; i++) {
                    array[i] = size - i;
                }
                start = System.nanoTime();
                mergeSort(array, 0, array.length - 1);
                end = System.nanoTime();
                writer.write(size + ",WorstCase," + (end - start) + "\n");

                // Almost sorted
                for (int i = 0; i < size; i++) {
                    array[i] = i;
                }
                // Introduce a small random shuffle
                for (int i = 0; i < size / 10; i++) {
                    int idx1 = random.nextInt(size);
                    int idx2 = random.nextInt(size);
                    int temp = array[idx1];
                    array[idx1] = array[idx2];
                    array[idx2] = temp;
                }
                start = System.nanoTime();
                mergeSort(array, 0, array.length - 1);
                end = System.nanoTime();
                writer.write(size + ",AlmostSorted," + (end - start) + "\n");

                System.out.println("ArraySize: " + size + " cases completed.");
            }

            System.out.println("Execution times written to mergesort_cases_runtime.csv
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Tabular Data:

```
ArraySize,Case,ExecutionTime(ns)
1000,BestCase,985900
1000,AverageCase,352100
1000,WorstCase,112000
1000,AlmostSorted,143699
2000,BestCase,237399
2000,AverageCase,312999
```

```
2000,WorstCase,218500
2000,AlmostSorted,266899
3000,BestCase,347099
3000,AverageCase,530200
3000,WorstCase,379300
3000,AlmostSorted,710601
4000,BestCase,347600
4000,AverageCase,587000
4000,WorstCase,360600
4000,AlmostSorted,409300
5000,BestCase,354800
5000,AverageCase,1394400
5000,WorstCase,999400
5000,AlmostSorted,547801
6000,BestCase,547000
6000,AverageCase,921901
6000,WorstCase,596500
6000,AlmostSorted,1130001
7000,BestCase,584800
7000,AverageCase,2995100
7000,WorstCase,553600
7000,AlmostSorted,800900
8000,BestCase,794601
8000,AverageCase,1340199
8000,WorstCase,1048701
8000,AlmostSorted,1612499
9000,BestCase,1019900
9000,AverageCase,3100301
9000,WorstCase,976700
9000,AlmostSorted,1198601
10000,BestCase,1166900
10000,AverageCase,1844800
10000,WorstCase,1102500
10000,AlmostSorted,4767100
```

Plot the Graph:



Time complexity Analysis:

Best Case(Already Sorted): 0(nlogn)
Average Case: Random O(nlogn)
Worst Case: Reverse sorted O(nlogn)

Merge Sort Algorithm:

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public class quicksort_runtime {

    // QuickSort implementation
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(array, low, high);

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

    private static int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = low - 1;
```

```
        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
                // Swap array[i] and array[j]
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        // Swap array[i+1] and pivot (array[high])
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }

    public static void main(String[] args) {
        Random random = new Random();

        try (FileWriter writer = new FileWriter("quicksort_cases_runtime.csv")) {
            writer.write("ArraySize,Case,ExecutionTime(ns)\n");

            for (int size = 1000; size <= 10000; size += 1000) {
                int[] array = new int[size];

                // Best Case (Balanced Split)
                for (int i = 0; i < size; i++) {
                    array[i] = i; // Already sorted array
                }
                long start = System.nanoTime();
                quickSort(array, 0, array.length - 1);
                long end = System.nanoTime();
                writer.write(size + ",BestCase," + (end - start) + "\n");

                // Average Case (Random Array)
                for (int i = 0; i < size; i++) {
                    array[i] = random.nextInt(10000);
                }
                start = System.nanoTime();
                quickSort(array, 0, array.length - 1);
                end = System.nanoTime();
                writer.write(size + ",AverageCase," + (end - start) + "\n");

                // Worst Case (Already Sorted in Reverse Order)
                for (int i = 0; i < size; i++) {
                    array[i] = size - i;
                }
                start = System.nanoTime();
                quickSort(array, 0, array.length - 1);
                end = System.nanoTime();
                writer.write(size + ",WorstCase," + (end - start) + "\n");
```

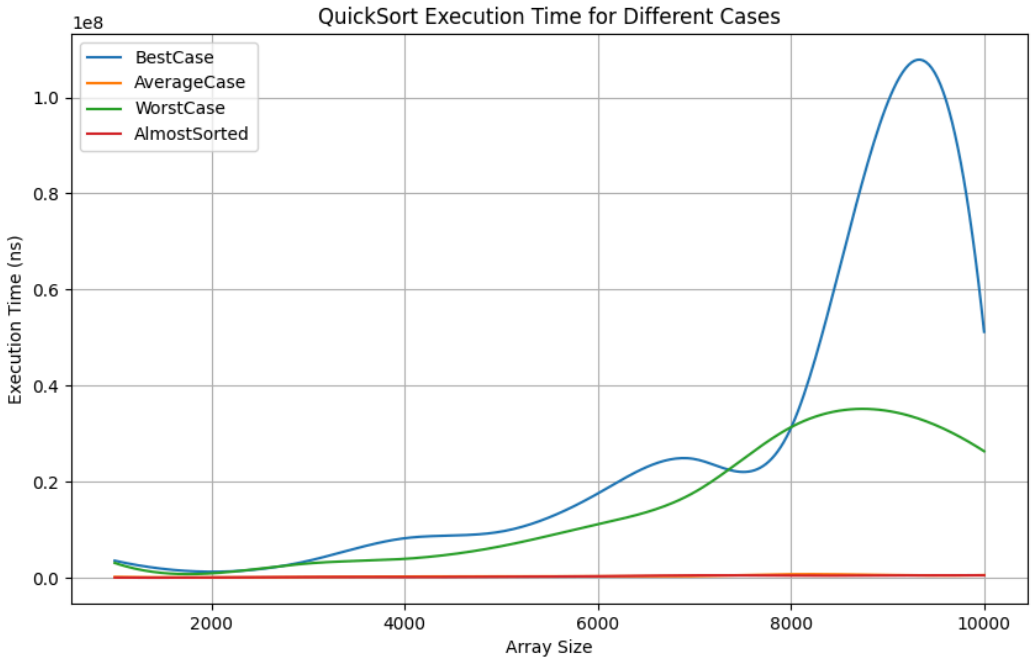
```
                // Almost Sorted Case
                for (int i = 0; i < size; i++) {
                    array[i] = i;
                }
                // Introduce a small random shuffle
                for (int i = 0; i < size / 10; i++) {
                    int idx1 = random.nextInt(size);
                    int idx2 = random.nextInt(size);
                    int temp = array[idx1];
                    array[idx1] = array[idx2];
                    array[idx2] = temp;
                }
                start = System.nanoTime();
                quickSort(array, 0, array.length - 1);
                end = System.nanoTime();
                writer.write(size + ",AlmostSorted," + (end - start) + "\n");

                System.out.println("ArraySize: " + size + " cases completed.");
            }

            System.out.println("Execution times written to quicksort_cases_runtime.csv
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Tabular Data:

```
ArraySize,Case,ExecutionTime(ns)
1000,BestCase,3570500
1000,AverageCase,190200
1000,WorstCase,3106500
1000,AlmostSorted,54700
2000,BestCase,1272199
2000,AverageCase,88401
2000,WorstCase,926401
2000,AlmostSorted,70301
3000,BestCase,3519600
3000,AverageCase,182100
3000,WorstCase,2996800
3000,AlmostSorted,163900
4000,BestCase,8238801
4000,AverageCase,281399
4000,WorstCase,3949300
4000,AlmostSorted,138500
5000,BestCase,9623200
5000,AverageCase,295700
5000,WorstCase,6579000
5000,AlmostSorted,238300
6000,BestCase,17537500
6000,AverageCase,322301
```

```
6000,WorstCase,11152200
6000,AlmostSorted,316999
7000,BestCase,24732800
7000,AverageCase,364700
7000,WorstCase,17796800
7000,AlmostSorted,521800
8000,BestCase,31166399
8000,AverageCase,758700
8000,WorstCase,31346100
8000,AlmostSorted,454501
9000,BestCase,98831000
9000,AverageCase,601699
9000,WorstCase,34750700
9000,AlmostSorted,489400
10000,BestCase,51204199
10000,AverageCase,573901
10000,WorstCase,26347000
10000,AlmostSorted,526200
```

Plot the Graph:



Time Complexity Analysis:
Best Case: Occurs when the pivot splits the array into two equal parts. O(nlogn)
Average Case: Typical case when the pivot splits the array unevenly but not extremely. O(nlogn)
Worst Case:
Occurs when the pivot results inone very small subarray and one large subarray(n-1 and 1) recursively. O(n^2)

N-Queens algorithm:

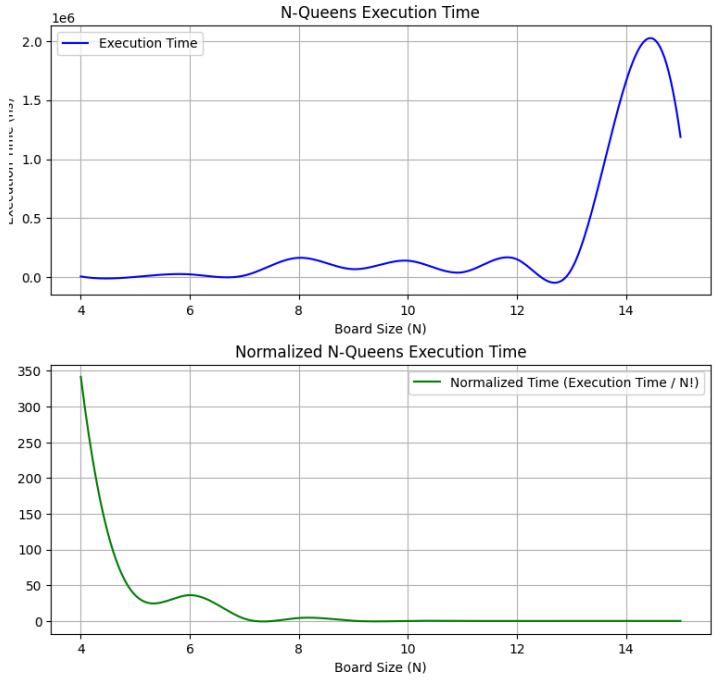
```java
import java.io.FileWriter;
import java.io.IOException;

public class NQueensRuntime {

    // Function to check if a queen can be placed at board[row][col]
    public static boolean isSafe(int[][] board, int row, int col, int n) {
        // Check the column
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 1) {
                return false;
            }
        }

        // Check the upper-left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        // Check the upper-right diagonal
        for (int i = row, j = col; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        return true;
    }

    // Backtracking function to solve the N-Queens problem
    public static boolean solveNQueens(int[][] board, int row, int n) {
        if (row >= n) {
            return true; // All queens are placed successfully
        }

        for (int col = 0; col < n; col++) {
            if (isSafe(board, row, col, n)) {
                board[row][col] = 1; // Place the queen

                if (solveNQueens(board, row + 1, n)) {
                    return true;
                }

                board[row][col] = 0; // Backtrack
            }
        }

        return false; // No solution for this configuration
    }
```

```
    // Main method to analyze runtime
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("nqueens_runtime.csv")) {
            writer.write("BoardSize,ExecutionTime(ns)\n");

            for (int n = 4; n <= 15; n++) {
                int[][] board = new int[n][n];

                long startTime = System.nanoTime();
                solveNQueens(board, 0, n);
                long endTime = System.nanoTime();

                long executionTime = endTime - startTime;
                writer.write(n + "," + executionTime + "\n");

                System.out.println("N = " + n + ", Execution Time: " + executionTime +
            }

            System.out.println("Execution times written to nqueens_runtime.csv");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Tabular Data:

```
BoardSize,ExecutionTime(ns)
4,8200
5,4300
6,26000
7,16100
8,166100
9,69900
10,141400
11,43500
12,153200
13,68700
14,1656000
15,1187400
```

Plot the Graph:



Time Complexity Analysis:

Best Case:

O(N!) Solved quickly for small N due to fewer configurations to explore.

Average Case:

O(N!) As N increases. the search space grows factorially. The average case runtime approaches the worst case.

Worst Case: O(N!)

O(N!) For large N, the algorithm may explore most or all configurations. making it computationally expensive.

Traveling Salesman Problem Algorithm:

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;

public class TravelingSalesmanRuntime {

    static int N; // Number of cities
    static int[][] distance; // Distance matrix
    static int[][] dp; // DP table for memoization

    // Recursive function to calculate minimum cost
    public static int tsp(int mask, int pos) {
        if (mask == (1 << N) - 1) { // All cities visited
            return distance[pos][0]; // Return to starting city
        }
```

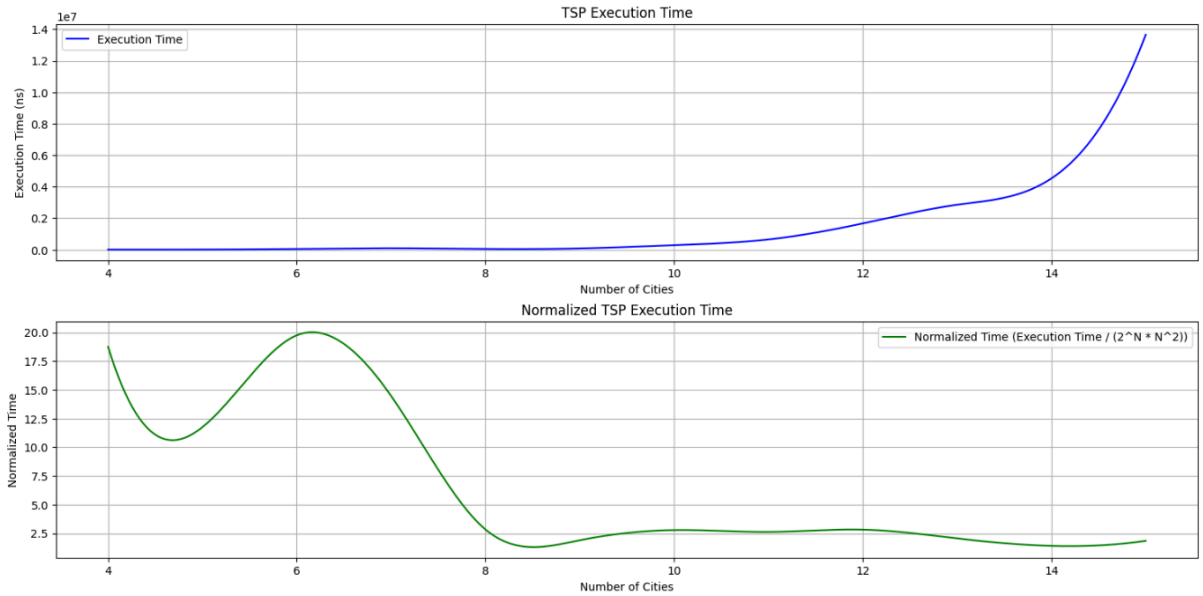
```
        if (dp[mask][pos] != -1) {
            return dp[mask][pos];
        }

        int minCost = Integer.MAX_VALUE;
        for (int city = 0; city < N; city++) {
            if ((mask & (1 << city)) == 0) { // City not visited
                int newCost = distance[pos][city] + tsp(mask | (1 << city), city);
                minCost = Math.min(minCost, newCost);
            }
        }

        dp[mask][pos] = minCost;
        return minCost;
    }

    // Generate a random distance matrix
    public static void generateRandomDistanceMatrix(int n) {
        distance = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                distance[i][j] = distance[j][i] = (int) (Math.random() * 100) + 1;
            }
            distance[i][i] = 0;
        }
    }

    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("tsp_runtime.csv")) {
            writer.write("Cities,ExecutionTime(ns)\n");

            for (int cities = 4; cities <= 15; cities++) {
                N = cities;
                generateRandomDistanceMatrix(N);

                dp = new int[1 << N][N];
                for (int[] row : dp) {
                    Arrays.fill(row, -1);
                }

                long startTime = System.nanoTime();
                tsp(1, 0);
                long endTime = System.nanoTime();

                long executionTime = endTime - startTime;
                writer.write(cities + "," + executionTime + "\n");

                System.out.println("Cities: " + cities + ", Execution Time: " + execut
            }

            System.out.println("Execution times written to tsp_runtime.csv");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

Tabular Data:

```
Cities,ExecutionTime(ns)
4,6600
5,11000
6,136700
7,172000
8,61200
9,193700
10,337900
11,575300
12,679800
13,5298700
14,3485800
15,17215300
```

Plot the Graph:



Time Complexity Analysis:

Brute Force Approach: O(n!) n is the number of cities. Compute All possible combinations of city visits.
Dynamic Programming Approach: O(n*n*2^n) In this case, there are 2 parameters - mask which represents the set of all visited cities so far and curr which represents the current city. Both the parameters have a finite range. The max limit of mask is 2^n and max limit of curr is n. We use a 2D array of size(nx2^n) to store the results for each combination of mask and curr.