Experiment-2

Experiment Name: Sort a given set of elements using Merge sort algorithm and find the time complexity for different values of n.

Description:

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void merge(int arr[], int l, int m, int r) {

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

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

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;
```

```
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
```

```
        printf("%d ", arr[i]);
    printf("\n");
}
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    srand(time(0));
    for (int i = 0; i < n; i++) {
        arr[i] = rand();
    }
    // printf("Original array:\n");
    // printArray(arr, n);
    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();
    // printf("Sorted array:\n");
    // printArray(arr, n);
    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("MergeSort execution time (seconds): %.10f s\n", time_taken);
    free(arr);
```

```
    return 0;
}
```

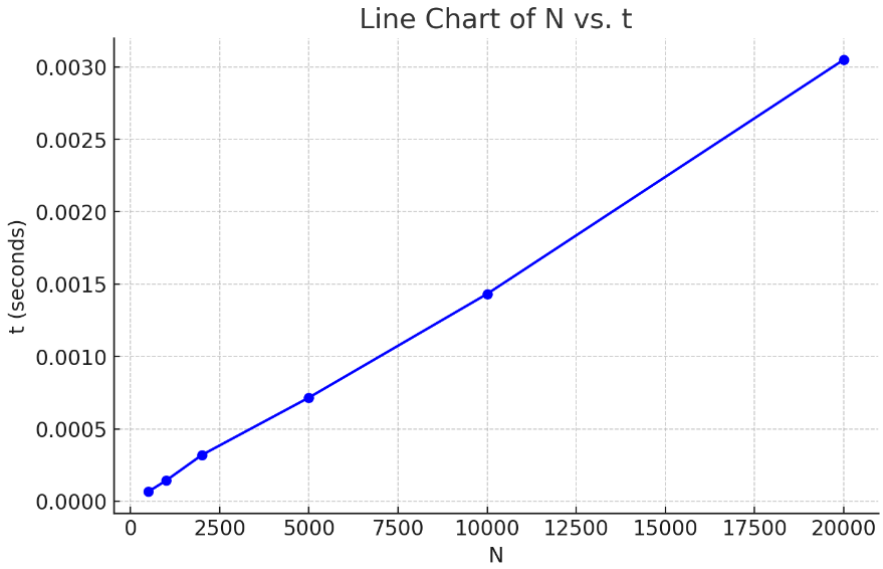
Output Screenshot:

```
/tmp/PsGzL1Y4xf.o
Enter the number of elements: 20000
MergeSort execution time (seconds): 0.0030490000 s


=== Code Execution Successful ===
```

Input (n) Vs Time Taken (t) Table:

N	t
500	0.0000670000
1000	0.0001440000
2000	0.0003200000
5000	0.0007160000
10000	0.0014320000
20000	0.0030490000

GRAPH:



Time Complexity:

	Best Case	Worst Case	Average Case
Merge Sort	O (n Log(n))	O (n Log(n))	O (n Log(n))

Learning Outcomes:

Experiment-1

Experiment Name: Sort a given set of elements using insertion sort algorithm and find the time complexity for different values of n.

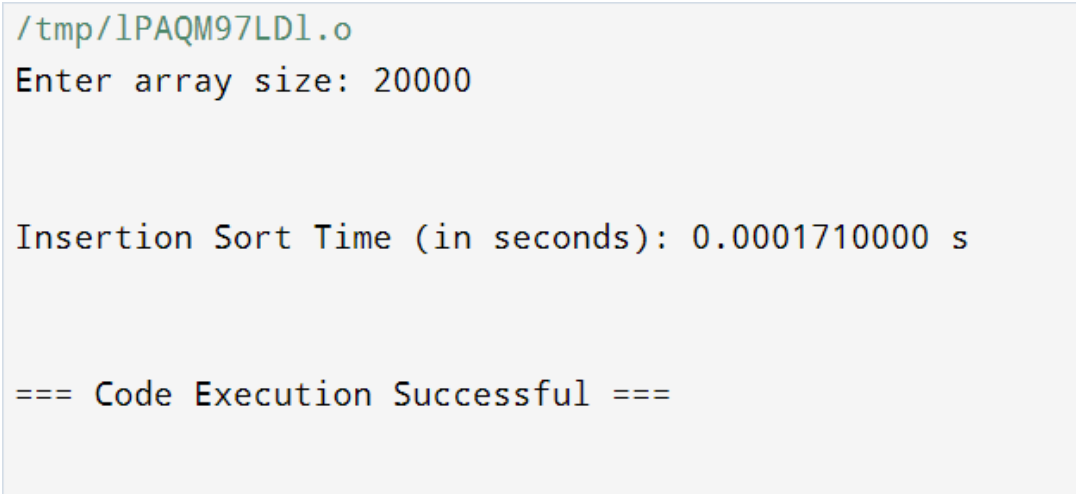
Description:

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
int main() {
    int n;
    printf("Enter array size: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int));
    srand(time(NULL));
    // printf("Unsorted Array:\n");
    // for (int i = 0; i < n; ++i) {
    //     arr[i] = rand();
```

```
    // printf("%d ", arr[i]);
  // }
  printf("\n");
  clock_t start_time = clock();
  insertionSort(arr, n);
  clock_t end_time = clock();
  // printf("Sorted Array:\n");
  // for (int i = 0; i < n; ++i) {
    // printf("%d ", arr[i]);
  // }
  printf("\n");
  double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;
  printf("Insertion Sort Time (in seconds): %.10f s\n", time_taken);
  free(arr);
  return 0;
}
```

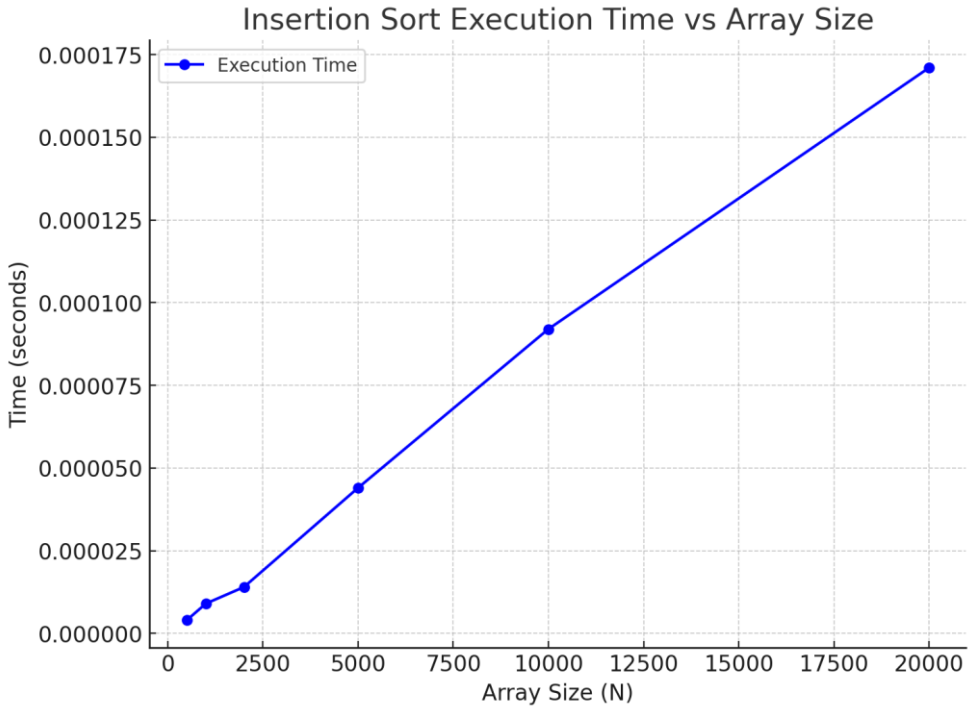
Output Screenshot:



Input (n) Vs Time Taken (t) Table:

N	t
500	0.0000040000
1000	0.0000090000
2000	0.0000140000
5000	0.0000440000
10000	0.0000920000
20000	0.00017100000

Graph:



Time Complexity:

	Best Case	Worst Case	Average Case
Insertion Sort	O (n)	O (n log n)	O (n^2)

Learning Outcomes:

Experiment: 4

Experiment Name: Write a program to implement Knapsack problem using greedy approach.

Description:

Algorithm:

Code:

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

// Function to perform the fractional knapsack algorithm
void fractional_knapsack(int n, float weight[], float value[], float capacity) {
    float ratio[n], total_value = 0, used_capacity = 0;
    int i, j;

    // Calculate the ratio of value to weight for each item
    for (i = 0; i < n; i++) {
        ratio[i] = value[i] / weight[i];
    }

    // Sort items based on the value-to-weight ratio (in descending order)
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
                float temp = ratio[i];
                ratio[i] = ratio[j];
                ratio[j] = temp;

                temp = weight[i];
                weight[i] = weight[j];
                weight[j] = temp;

                temp = value[i];
                value[i] = value[j];
                value[j] = temp;
            }
        }
    }

    // Fill the knapsack
    for (i = 0; i < n; i++) {
        if (used_capacity + weight[i] <= capacity) {
            used_capacity += weight[i];
            total_value += value[i];
        } else {
            float remaining_capacity = capacity - used_capacity;
            total_value += value[i] * (remaining_capacity / weight[i]);
            break;
        }
    }

    printf("Total value in the knapsack: %.2f\n", total_value);
}

int main() {
    int n;
```

```
    float capacity;
    clock_t start_time, end_time;

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

    // Allocate arrays for weight and value
    float weight[n], value[n];

    // Seed the random number generator
    srand(time(0));

    // Generate random weights and values
    printf("Items generated:\n");
    for (int i = 0; i < n; i++) {
        weight[i] = (rand() % 100) + 1; // Random weight between 1 and 100
        value[i] = (rand() % 500) + 1;  // Random value between 1 and 500
        printf("Item %d - Weight: %.2f, Value: %.2f\n", i+1, weight[i], value[i]);
    }

    // Input the maximum capacity of the knapsack
    printf("Enter the maximum capacity of the knapsack: ");
    scanf("%f", &capacity);

    // Start timing
    start_time = clock();

    // Call the fractional knapsack function
    fractional_knapsack(n, weight, value, capacity);

    // End timing
    end_time = clock();

    // Calculate the total computation time in seconds
    double total_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Total computation time: %.6f seconds\n", total_time);

    return 0;
}
```

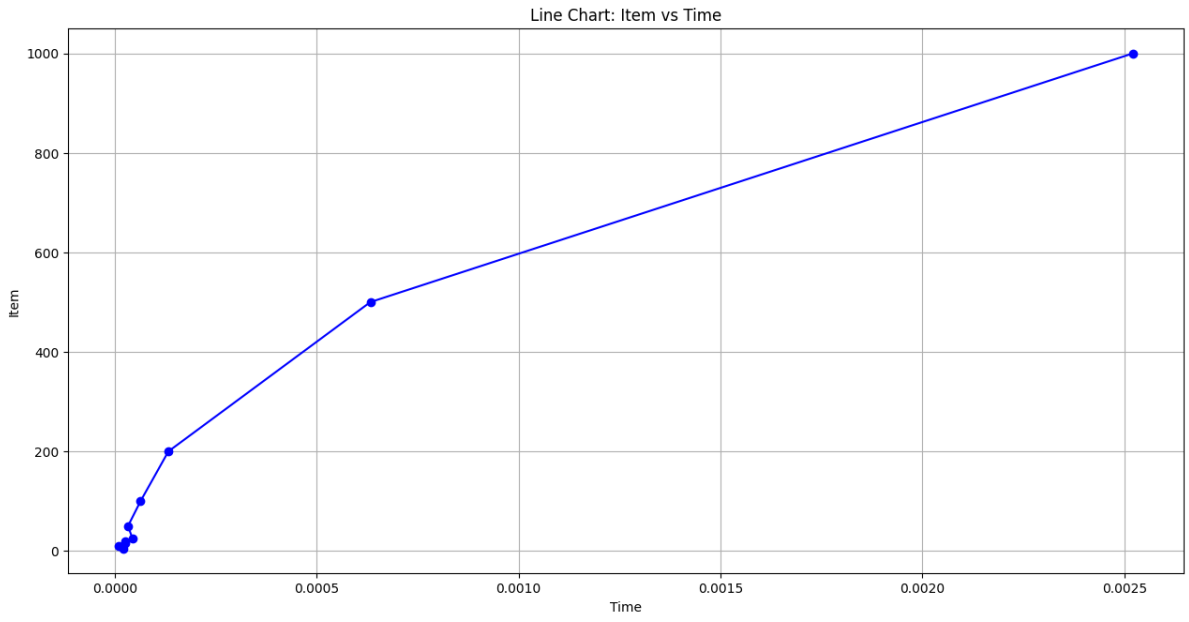
Output:

```
Enter the number of items: 10
Items generated:
Item 1 - Weight: 43.00, Value: 477.00
Item 2 - Weight: 13.00, Value: 228.00
Item 3 - Weight: 26.00, Value: 28.00
Item 4 - Weight: 14.00, Value: 263.00
Item 5 - Weight: 65.00, Value: 466.00
Item 6 - Weight: 14.00, Value: 58.00
Item 7 - Weight: 34.00, Value: 77.00
Item 8 - Weight: 56.00, Value: 467.00
Item 9 - Weight: 3.00, Value: 472.00
Item 10 - Weight: 88.00, Value: 140.00
Enter the maximum capacity of the knapsack: 18
Total value in the knapsack: 752.54
Total computation time: 0.000010 seconds
```

Number of items VS Time graph:

Item	Time
5	0.000021
10	0.00001
15	0.000025
20	0.000027
25	0.000044
50	0.000033
100	0.000064
200	0.000133
500	0.000633
1000	0.002522

Graph



Time Complexity:

Algorithm	Best	Worst	Average
Knapsack	O(n)	$O(n^2)$	$O(n^2)$

Learning Outcome:

Experiment 9

AIM: Write a C program to perform All Pair Shortest Path Algorithm using Floyd Warshall Algorithm.

Description:

Algorithm:

Program:

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

#define MAX 20
#define INF 99999

// Function to implement the Floyd-Warshall algorithm
void floydWarshall(int graph[MAX][MAX], int n) {
    // Create a distance matrix initialized to the original graph
    int dist[MAX][MAX];

    // Initialize the distance matrix with graph values
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Update the distance matrix using the Floyd-Warshall algorithm
    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] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    // Print the calculated shortest distances
    printf("Vertex\t\tShortest Path Distance\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][j] == INF) {
                printf("From %d to %d: INF\n", i, j);
            } else {
                printf("From %d to %d: %d\n", i, j, dist[i][j]);
            }
        }
        printf("\n");
    }
```

```
}

int main() {
    int n;
    int graph[MAX][MAX];

    // Seed the random number generator
    srand(time(0));

    // Input the number of nodes
    printf("Enter the number of nodes (max %d): ", MAX);
    scanf("%d", &n);

    // Generate a random graph with weights between 1 and 10
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                graph[i][j] = rand() % 10 + 1; // Random weights from 1 to 10
            } else {
                graph[i][j] = 0; // Distance to self is 0
            }
        }
    }

    // Replace random weights with INF (to simulate disconnected graph) with some probability
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (graph[i][j] != 0 && rand() % 10 < 3) { // 30% chance to make the edge INF
                graph[i][j] = INF;
            }
        }
    }

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

    // Run Floyd-Warshall's algorithm
    floydWarshall(graph, n);

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

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

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

    return 0;
}
```

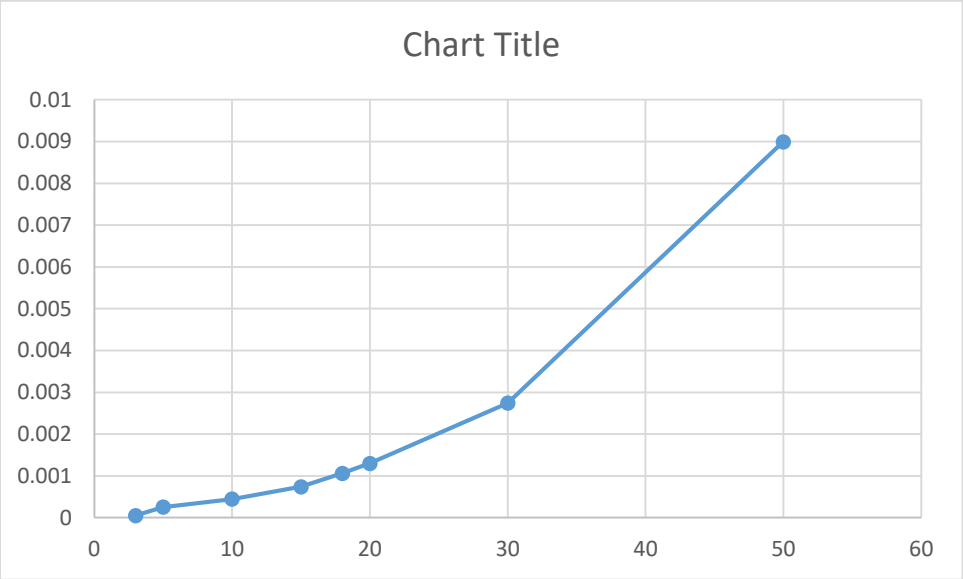
Output:

```
Enter the number of nodes (max 20): 5
Vertex          Shortest Path Distance
From 0 to 0: 0
From 0 to 1: 1
From 0 to 2: 14
From 0 to 3: 5
From 0 to 4: 4

From 1 to 0: 1
From 1 to 1: 0
From 1 to 2: 13
From 1 to 3: 4
From 1 to 4: 5

From 2 to 0: 4
From 2 to 1: 5
From 2 to 2: 0
From 2 to 3: 9
From 2 to 4: 8

From 3 to 0: 10
From 3 to 1: 9
From 3 to 2: 9
From 3 to 3: 0
From 3 to 4: 14

From 4 to 0: 6
From 4 to 1: 5
From 4 to 2: 10
From 4 to 3: 9
From 4 to 4: 0

Time taken for calculation: 0.000253 seconds
vikram@DESKTOP-S5P315A:~/dsa$
```

Plot between number of nodes VS time.

nodes	time
3	**0.00005**
5	0.000253
10	0.000445
15	0.000739
18	0.001063
20	0.001297
30	0.002742
50	0.008993

Graph:



Time Complexity

Algorithm	Best Case	Average Case	Worst Case
Floyd's algorithm	O(V3)	O(V3)	O(V3)

Learning Outcome: