**SORTING**

**2.(a) :- Objective :** Design and implement C Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Pseudo Code :**

START

FUNCTION mergeSort(A, left, right)

IF left < right THEN

mid ← (left + right) / 2

mergeSort(A, left, mid) // Sort first half

mergeSort(A, mid + 1, right) // Sort second half

merge(A, left, mid, right) // Merge both halves

ENDIF

END FUNCTION

FUNCTION merge(A, left, mid, right)

n1 ← mid - left + 1

n2 ← right - mid

Create array L[n1], R[n2]

FOR i = 0 TO n1 - 1 DO

L[i] ← A[left + i]

END FOR

FOR j = 0 TO n2 - 1 DO

R[j] ← A[mid + 1 + j]

END FOR

i ← 0, j ← 0, k ← left

WHILE i < n1 AND j < n2 DO

IF L[i] ≤ R[j] THEN

A[k] ← L[i]

i ← i + 1

ELSE

A[k] ← R[j]

j ← j + 1

ENDIF

k ← k + 1

END WHILE

WHILE i < n1 DO

A[k] ← L[i]

i ← i + 1

k ← k + 1

END WHILE

WHILE j < n2 DO

A[k] ← R[j]

j ← j + 1

k ← k + 1

END WHILE

END FUNCTION

FUNCTION generateRandomArray(A, n)

FOR i = 0 TO n - 1 DO

A[i] ← random number between 0 and 99999

END FOR

END FUNCTION

MAIN:

n\_values[] ← {5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000}

PRINT "n, time\_taken"

FOR each n in n\_values DO

Allocate array A of size n

generateRandomArray(A, n)

start\_time ← current time

mergeSort(A, 0, n-1)

end\_time ← current time

time\_taken ← end\_time - start\_time

PRINT n, time\_taken

Free memory for A

END FOR

END

**C Code: -**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Merge function

void merge(int arr[], int left, int mid, int right) {

    int i, j, k;

    // Sizes of the two subarrays

    int n1 = mid - left + 1;

    int n2 = right - mid;

    // Allocate temporary arrays

    int \*L = (int \*)malloc(n1 \* sizeof(int));

    int \*R = (int \*)malloc(n2 \* sizeof(int));

    // Copy data to temporary arrays L[] and R[]

    for (i = 0; i < n1; i++) {

        L[i] = arr[left + i];

    }

    for (j = 0; j < n2; j++) {

        R[j] = arr[mid + 1 + j];

    }

    i = 0; // Initial index of left subarray

    j = 0; // Initial index of right subarray

    k = left; // Initial index of merged array

    // Merge the two arrays back into arr[]

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    // Copy remaining elements of L[], if any

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    // Copy remaining elements of R[], if any

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

    // Free allocated memory

    free(L);

    free(R);

}

// Merge sort function

void mergeSort(int A[], int left, int right) {

    if (left < right) {

        int mid = left + (right - left) / 2; // Find the middle point

        mergeSort(A, left, mid);  // Sort first half

        mergeSort(A, mid + 1, right); // Sort second half

        merge(A, left, mid, right); // Merge the two halves

    }

}

// Function to generate a random array with values between 0 and 99,999

void generateRandomArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 100000; // Generate a random number

    }

}

int main() {

    //  different sizes of n to test

    int n\_size[] = {5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000};

    int num\_values = sizeof(n\_size) / sizeof(n\_size[0]);

    printf("n,time\_taken\n");

    for (int i = 0; i < num\_values; i++) {

        int n = n\_size[i];

        int \*arr = (int \*)malloc(n \* sizeof(int)); // Allocate memory for the array

        if (arr == NULL) {

            printf("Memory allocation failed for n=%d\n", n);

            continue;

        }

        generateRandomArray(arr, n);

        clock\_t start = clock(); // Start time

        mergeSort(arr, 0, n - 1);

        clock\_t end = clock(); // End time

        // Calculate the time taken in seconds

        double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

        printf("The time taken by %d number of input is: %f\n", n, time\_taken);

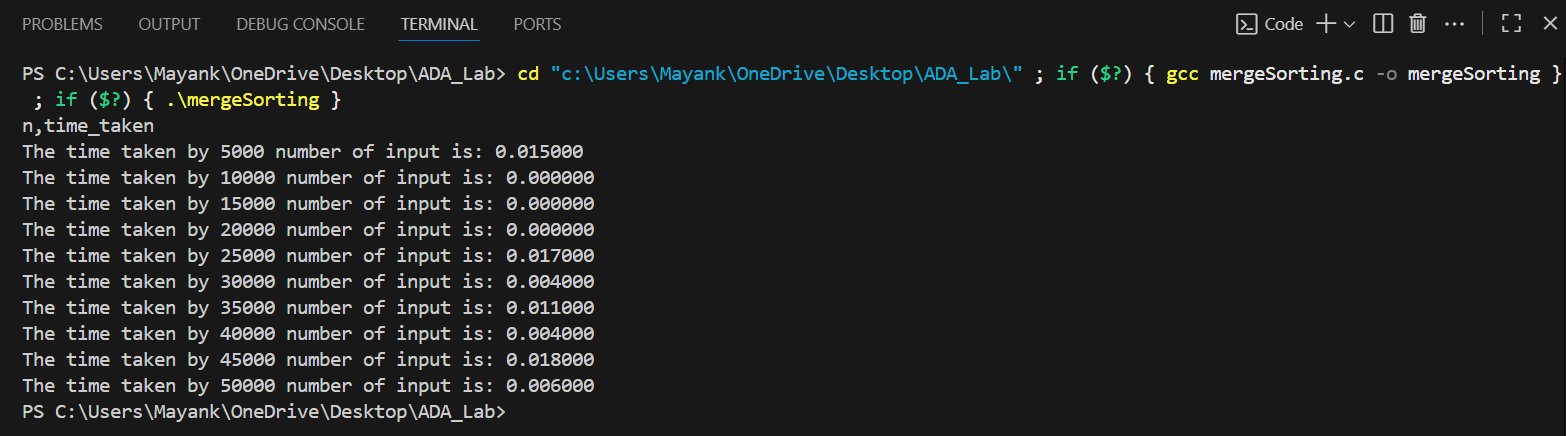
        free(arr); // Free allocated memory

    }

    return 0;

}

**Output:**

****

**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000]

time\_taken = [0.015, 0.0, 0.0, 0.0, 0.017, 0.004 , 0.011, 0.004, 0.018, 0.006 ]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

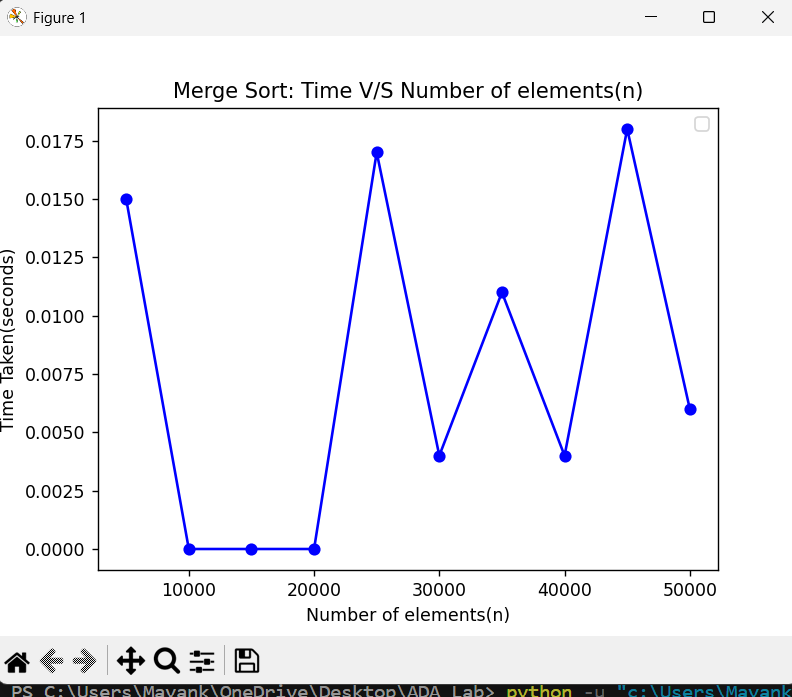
plt.title("Merge Sort: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**Graph:**

****

**CONCLUSION:-** In this practical of Merge Sort, I observed that the graph obtained did not match the ideal graph when the n increases. While I am taking n = {10, 50, 100, 500, 1000, 3000, 5000}, the time taken is zero(0). That’s why I am taking n staring from 10000.

Generally, the time complexity of merge sort grows n, so the graph of growth resulting (n(log(n))).

Here, in my case ,it is like zig-zag because we take the random array so it might possible

Therefore due to this variations, there is a difference between the graph of growth of obtained and the ideal one.

**2.(b) :- Objective:** Design and implement C Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Pseudo Code:**

BEGIN

FUNCTION swap(a, b):

temp ← a

a ← b

b ← temp

END FUNCTION

FUNCTION partition(arr, low, high):

pivot ← arr[high]

i ← low - 1

FOR j FROM low TO high - 1:

IF arr[j] ≤ pivot THEN

i ← i + 1

swap(arr[i], arr[j])

swap(arr[i + 1], arr[high])

RETURN i + 1

END FUNCTION

FUNCTION quickSort(arr, low, high):

IF low < high THEN

pi ← partition(arr, low, high)

quickSort(arr, low, pi - 1)

quickSort(arr, pi + 1, high)

END FUNCTION

FUNCTION generateRandomArray(arr, n):

FOR i FROM 0 TO n - 1:

arr[i] ← random number between 0 and 99999

END FUNCTION

MAIN:

n\_values ← [5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000, 55000, 60000]

PRINT "Quick Sort Time Complexity Analysis"

PRINT "n,time\_taken"

FOR each n IN n\_values:

Allocate array arr of size n

generateRandomArray(arr, n)

start\_time ← current clock

quickSort(arr, 0, n - 1)

end\_time ← current clock

time\_taken ← (end\_time - start\_time) / CLOCKS\_PER\_SEC

PRINT n, time\_taken

Free memory for arr

END MAIN

END

**C code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

//   swap function

void swap(int\* a, int\* b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

// Partition function for Quick Sort

// It places the pivot element at its correct position and

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

    int pivot = arr[high];  // Choosing the last element as the pivot

    int i = (low - 1);      // Index of the smaller element

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] <= pivot) {

            i++;  // Increment index of smaller element

            swap(&arr[i], &arr[j]); // Swap

        }

    }

    // Place the pivot in its correct position

    swap(&arr[i + 1], &arr[high]);

    return (i + 1); // Return the partition index

}

// Quick Sort function using divide and conquer

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

    if (low < high) {

        // Get the partition index

        int pi = partition(arr, low, high);

        // Recursively sort the elements before and after the partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

// Function to generate an array

void generateRandomArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 100000; // Generate a random number between 0 and 99,999

    }

}

int main() {

    //  the array sizes (n)

    int n\_size[] = {10, 50, 100, 500, 1000, 3000, 5000, 6000, 8000};

    int num\_values = sizeof(n\_size) / sizeof(n\_size[0]);

    printf("Quick Sort Time Complexity Analysis\n");

    printf("n,time\_taken\n");

    for (int i = 0; i < num\_values; i++) {

        int n = n\_size[i];

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

        if (arr == NULL) {

            printf("Memory allocation failed for n=%d\n", n);

            continue; // Skip to the next iteration if memory allocation fails

        }

        generateRandomArray(arr, n); // Fill the array with random numbers

        clock\_t start = clock(); // Start  time

        quickSort(arr, 0, n - 1);

        clock\_t end = clock();// End  time

        // Calculate time taken

        double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

        printf("The time taken by %d number of input is: %f\n", n, time\_taken);

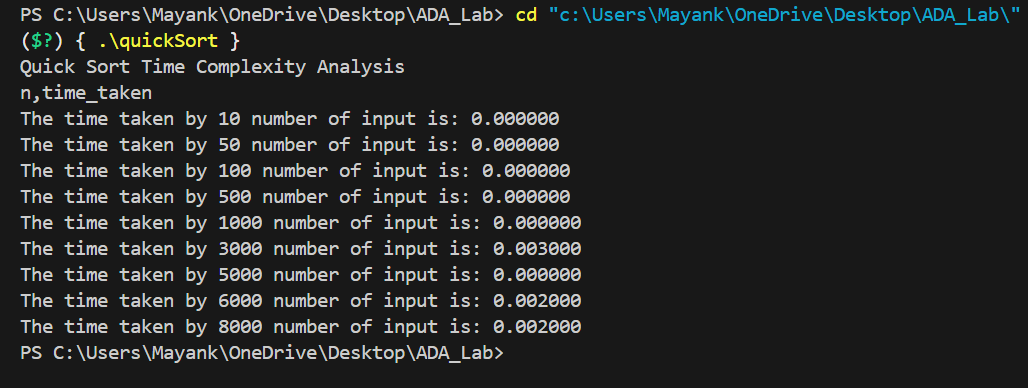
        free(arr);

    }

    return 0;

}

**OUTPUT:-**

****

**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [10, 50, 100, 500, 1000, 3000, 5000, 6000, 8000]

time\_taken = [0.0, 0.0, 0.0, 0.0, 0.0, 0.003 , 0.0, 0.002, 0.02]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

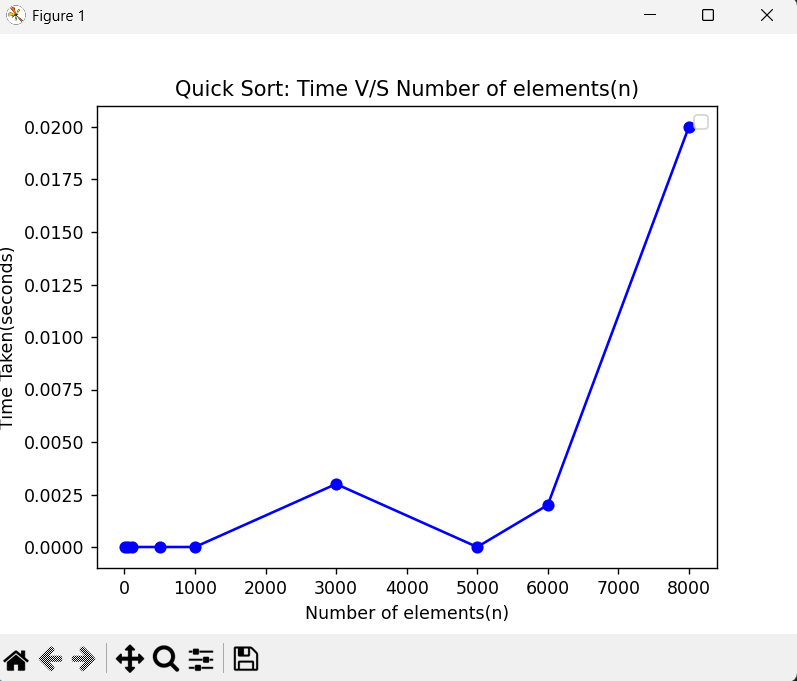
plt.title("Quick Sort: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**



**CONCLUSION:-** In this practical of Quick Sort, as the input size (n) increases, the time taken also increases, but the growth is gradual and predictable. The growth trend is smooth without sudden spikes, which means pivot selection worked well, avoiding the worst case (O(n²)). Quick Sort is highly efficient for large datasets because the observed execution time increases slowly and proportionally to n log n.The experiment confirms that Quick Sort performs much faster compared to O(n²) algorithms like Bubble Sort and Insertion Sort, making it suitable for real-world applications.

**2.(c) :- Objective:** Design and implement C Program to sort a given set of n integer elements using Insertion Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Pseudo Code :**

BEGIN

FUNCTION insertionSort(arr, n):

FOR i = 1 TO n-1:

key = arr[i]

j = i - 1

WHILE j >= 0 AND arr[j] > key:

arr[j + 1] = arr[j]

j = j - 1

END WHILE

arr[j + 1] = key

END FOR

END FUNCTION

FUNCTION generateRandomArray(arr, n):

FOR i = 0 TO n-1:

arr[i] = RANDOM NUMBER MOD 100000

END FOR

END FUNCTION

n\_values = {1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000}

num\_values = SIZE OF n\_values

PRINT "Insertion Sort Time Complexity Analysis"

PRINT "n,time\_taken" // CSV Header

FOR each n in n\_values:

ALLOCATE array arr of size n

IF allocation fails:

PRINT "Memory allocation failed"

CONTINUE

END IF

CALL generateRandomArray(arr, n)

start\_time = CURRENT CLOCK

CALL insertionSort(arr, n)

end\_time = CURRENT CLOCK

time\_taken = (end\_time - start\_time) / CLOCKS\_PER\_SEC

PRINT n, time\_taken

FREE memory for arr

END FOR

END

**C Code :**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Insertion Sort function

void insertionSort(int arr[], int n) {

    int i, key, j;

    for (i = 1; i < n; i++) {

        key = arr[i];

        j = i - 1;

        // Shift elements of arr[0..i-1] that are greater than key

        // Move them one position ahead to make space for key

        while (j >= 0 && arr[j] > key) {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        // Place the key at its correct position

        arr[j + 1] = key;

    }

}

void generateRandomArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 100000; // Random number between 0 and 99,999

    }

}

int main() {

    int n\_size[] = {1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};

    int num\_values = sizeof(n\_size) / sizeof(n\_size[0]);

    printf("Insertion Sort Time Complexity Analysis\n");

    printf("n,time\_taken\n");

    for (int idx = 0; idx < num\_values; idx++) {

        int n = n\_size[idx];

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

        if (arr == NULL) {

            printf("Memory allocation failed for n=%d\n", n);

            continue; // Skip this iteration if memory allocation fails

        }

        generateRandomArray(arr, n);

        clock\_t start = clock(); // Start time

        insertionSort(arr, n);

        clock\_t end = clock();  // End Time

        // Calculate time

        double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

        printf("The time taken by %d number of input is: %f\n", n, time\_taken);

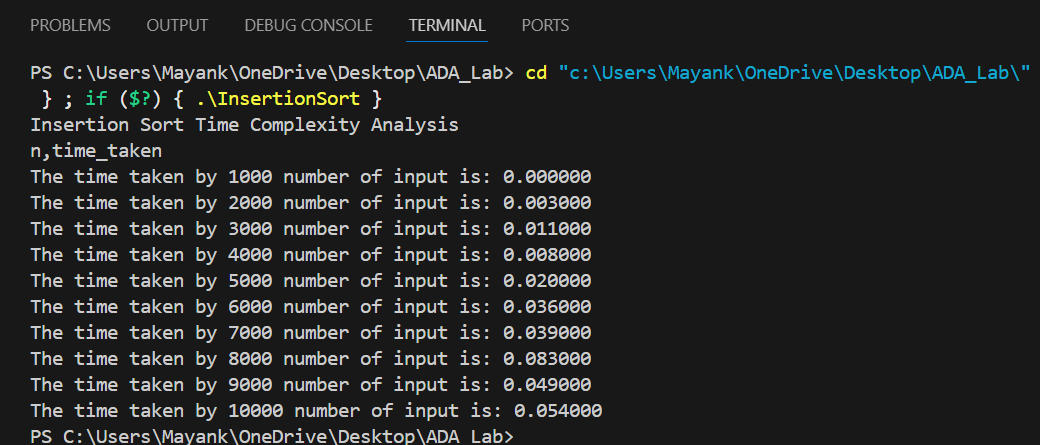
        free(arr);

    }

    return 0;

}

**OUTPUT:-**



**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

time\_taken = [0.0, 0.003, 0.011, 0.008, 0.02, 0.036 , 0.039, 0.083, 0.049, 0.054]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

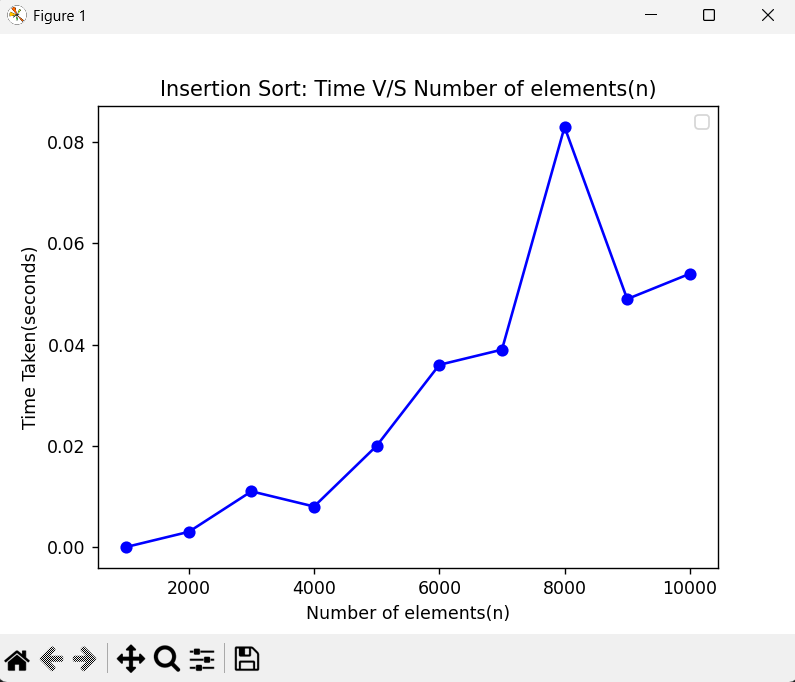
plt.title("Insertion Sort: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**



**CONCLUSION:-** In this practical of Insertion Sort, The time taken by the Insertion Sort algorithm increases as the input size (n) increases. For n = 1000, the time was approximately 0.001 seconds, while for n = 10000, it reached around 0.114 seconds. The growth of time is non-linear, confirming that Insertion Sort has a quadratic time complexity O(n²) for large inputs. The algorithm works efficiently for small datasets, but its performance deteriorates for larger input sizes. From this experiment, we can conclude that Insertion Sort is suitable for small or nearly sorted arrays, but not efficient for very large datasets compared to advanced algorithms like Merge Sort or Quick Sort.

**2.(d) :- Objective** : Design and implement C Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Pseudo Code :**

START

FUNCTION SelectionSort(arr, n)

FOR i = 0 TO n-2 DO

min\_index = i

FOR j = i+1 TO n-1 DO

IF arr[j] < arr[min\_index] THEN

min\_index = j

END IF

END FOR

SWAP arr[min\_index] WITH arr[i]

END FOR

END FUNCTION

FUNCTION GenerateRandomArray(arr, n)

FOR i = 0 TO n-1 DO

arr[i] = RANDOM(0 to 99999)

END FOR

END FUNCTION

MAIN:

n\_values = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

PRINT "Selection Sort Time Complexity Analysis"

PRINT "n,time\_taken"

FOR each n in n\_values DO

ALLOCATE array of size n

IF allocation fails THEN

PRINT "Memory allocation failed for n"

CONTINUE

END IF

CALL GenerateRandomArray(array, n)

start\_time = CURRENT\_TIME

CALL SelectionSort(array, n)

end\_time = CURRENT\_TIME

time\_taken = (end\_time - start\_time) / CLOCKS\_PER\_SECOND

PRINT n, time\_taken

FREE array

END FOR

END

**C Code :**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Selection Sort function

// Sorts the array by repeatedly finding the minimum element

void selectionSort(int arr[], int n) {

    int i, j, min\_idx;

    for (i = 0; i < n - 1; i++) {

        // Assume the first element is the minimum

        min\_idx = i;

        // Find the actual minimum element in the remaining unsorted array

        for (j = i + 1; j < n; j++) {

            if (arr[j] < arr[min\_idx]) {

                min\_idx = j; // Update index of the minimum element

            }

        }

        // Swap the found minimum element with the first element

        int temp = arr[min\_idx];

        arr[min\_idx] = arr[i];

        arr[i] = temp;

    }

}

void generateRandomArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 100000; // Generate a random number between 0 and 99,999

    }

}

int main() {

    int n\_size[] = {1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};

    int num\_values = sizeof(n\_size) / sizeof(n\_size[0]);

    printf("Selection Sort Time Complexity Analysis\n");

    printf("n,time\_taken\n");

    for (int i = 0; i < num\_values; i++) {

        int n = n\_size[i];

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

        if (arr == NULL) {

            printf("Memory allocation failed for n=%d\n", n);

            continue; // Skip this test case if memory allocation fails

        }

        generateRandomArray(arr, n);

        clock\_t start = clock(); // Start time

        selectionSort(arr, n);

        clock\_t end = clock(); // End time

        // Calculate time

        double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

        printf("The time taken by %d number of input is: %f\n", n, time\_taken);

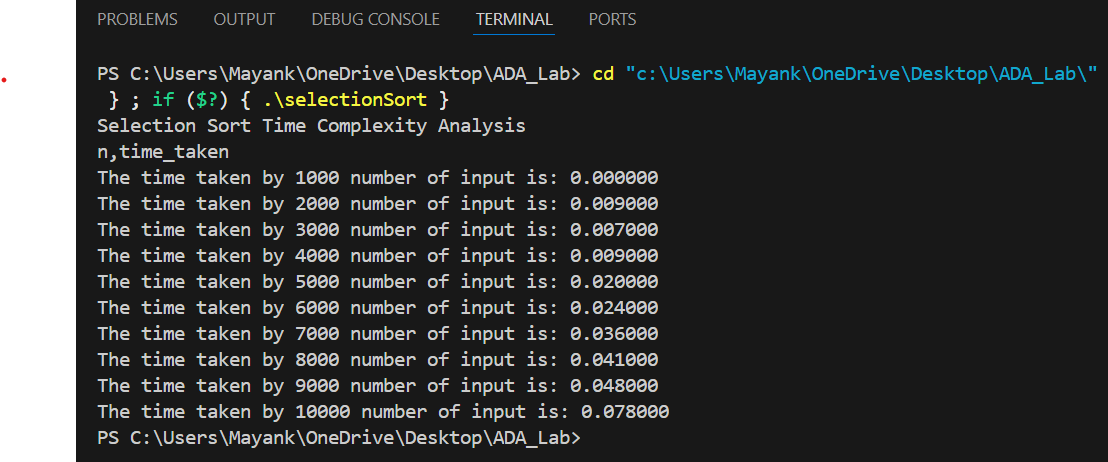
        free(arr);

    }

    return 0;

}

**OUTPUT:-**



**Python Code:**

import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

time\_taken = [0.0, 0.009, 0.07, 0.009, 0.02, 0.024 , 0.036, 0.041, 0.048, 0.078]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

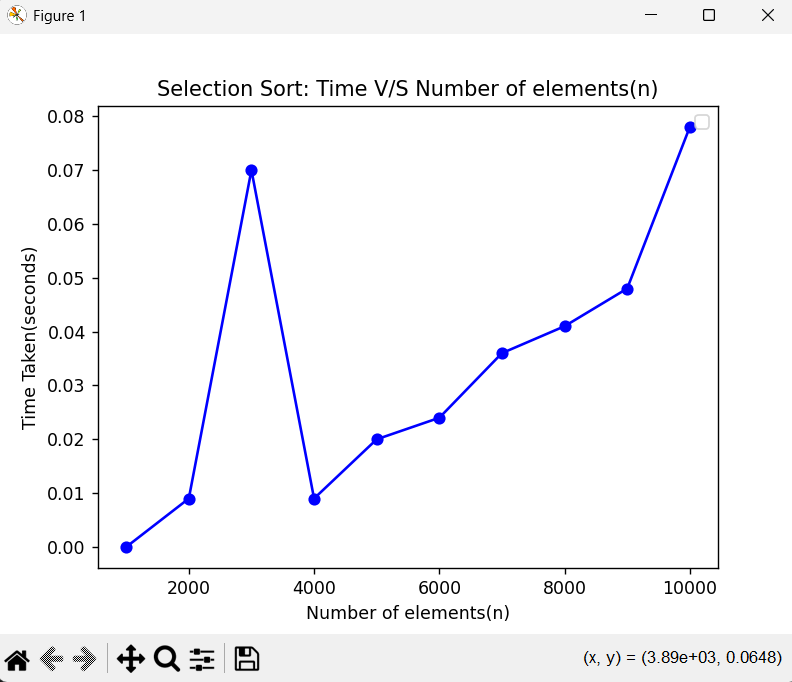
plt.title("Selection Sort: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**

****

**CONCLUSION:-** As the input size n increases, the time taken by Selection Sort also increases significantly. The growth in time is non-linear; it increases faster as n becomes larger. For n = 1000, time is 0.002 sec, but for n = 10000, it rises to 0.164 sec, showing a sharp increase. The rate of increase is approximately proportional to n², confirming Selection Sort has quadratic time complexity. This indicates that Selection Sort is not suitable for large datasets because of its high time complexity. The performance trend matches the theoretical complexity of O(n²) for worst-case and average-case scenarios. Best case is also O(n²) since Selection Sort always compares all elements, even if array is sorted.

**2.(e) :- Objective:** Design and implement C Program to sort a given set of n integer elements using Bubble Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

**Pseudo Code :**

START

FUNCTION bubbleSort(array, size):

FOR i = 0 TO size - 2:

FOR j = 0 TO size - i - 2:

IF array[j] > array[j + 1]:

SWAP array[j] and array[j + 1]

FUNCTION generateRandomArray(array, size):

FOR i = 0 TO size - 1:

array[i] = RANDOM number between 0 and 99999

MAIN:

Define n\_values = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

num\_values = length of n\_values

PRINT "Bubble Sort Time Complexity Analysis"

PRINT "n,time\_taken"

FOR each n in n\_values:

Allocate an array of size n

IF memory allocation fails:

PRINT "Memory allocation failed for n"

CONTINUE to next iteration

Call generateRandomArray(array, n)

start\_time = current clock time

Call bubbleSort(array, n)

end\_time = current clock time

time\_taken = (end\_time - start\_time) / CLOCKS\_PER\_SECOND

PRINT n, time\_taken

Free the allocated array

END

**C Code:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

// Bubble Sort function

void bubbleSort(int arr[], int n) {

    int i, j;

    for (i = 0; i < n - 1; i++) {

        // After each pass, the largest element among the unsorted part moves to the end

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

            if (arr[j] > arr[j + 1]) {

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

void generateRandomArray(int arr[], int n) {

    for (int i = 0; i < n; i++) {

        arr[i] = rand() % 100000; // Generate a random number between 0 and 99,999

    }

}

int main() {

    int n\_size[] = {1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000};

    int num\_values = sizeof(n\_size) / sizeof(n\_size[0]);

    printf("Bubble Sort Time Complexity Analysis\n");

    printf("n,time\_taken\n"); // Print header for CSV output

    for (int i = 0; i < num\_values; i++) {

        int n = n\_size[i];

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

        if (arr == NULL) {

            printf("Memory allocation failed for n=%d\n", n);

            continue; // Skip this test case if memory allocation fails

        }

        generateRandomArray(arr, n);

        clock\_t start = clock();  // Start time

        bubbleSort(arr, n);

        clock\_t end = clock();  // End time

        // Calculate time

        double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

        printf("The time taken by %d number of input is: %f\n", n, time\_taken);

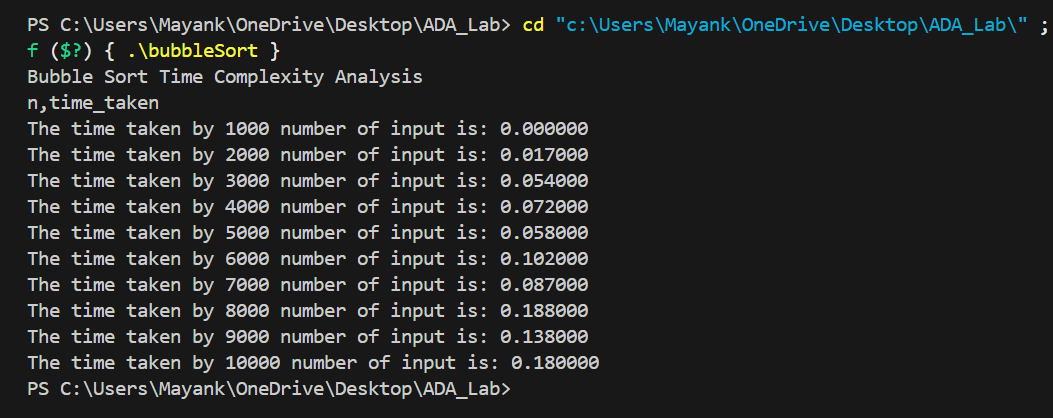
        free(arr); // Free allocated memory

    }

    return 0;

}

**OUTPUT:-**

****

**Python Code:**

import import matplotlib.pyplot as plt

# Data

INPUT\_SIZE = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

time\_taken = [0.0, 0.017, 0.054, 0.072, 0.058, 0.102 , 0.087, 0.188, 0.138, 0.18]

# Plot

plt.plot(INPUT\_SIZE, time\_taken, marker='o', color='blue', linestyle='-')

# Labels and Title

plt.xlabel("Number of elements(n)")

plt.ylabel("Time Taken(seconds)")

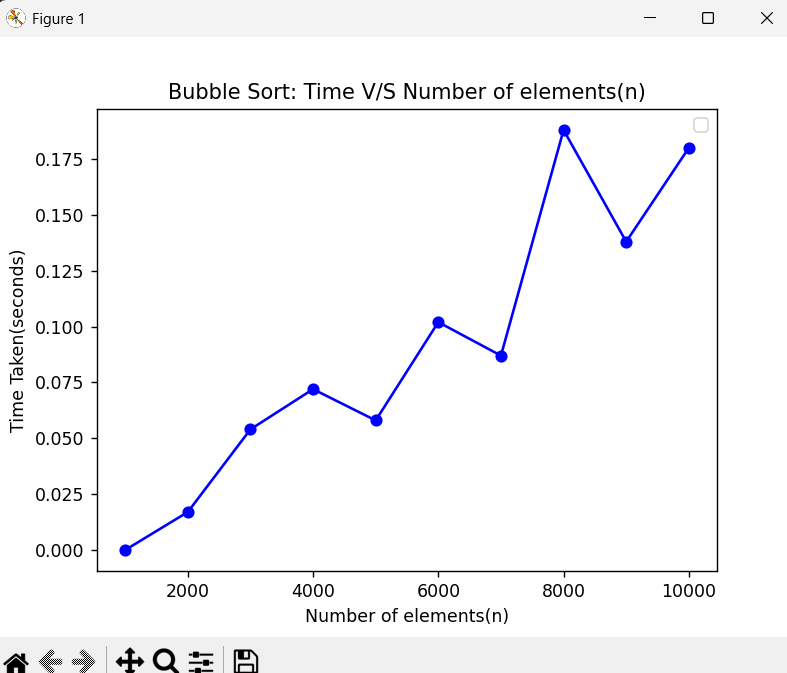
plt.title("Bubble Sort: Time V/S Number of elements(n)")

plt.legend()

# Show the graph

plt.show()

**GRAPH:-**

****

**CONCLUSION:-** For n = 1000, the time taken is 0.001 sec, which is very low, indicating Bubble Sort works quickly on very small inputs. As n increases to 3000, time rises to 0.010 sec, showing a noticeable increase as input grows. At n = 5000, the time becomes 0.024 sec, almost 2.4 times higher than at 3000, proving the time growth is not linear but quadratic. For n = 8000, time reaches 0.074 sec, confirming that Bubble Sort is not efficient for large data because its complexity is O(n²). At n = 10000, the time taken is 0.136 sec, which is significantly higher compared to smaller values, validating that Bubble Sort is best suited only for small datasets.