MEDICAPS UNIVERSITY, INDORE

DEPARTMENT OF COMPUTER SCIENCE & ENGINNERING

PRATICAL FILE

Design and Analysis of Algorithms

(CS3CO45)

SUBMITTED TO:-

PROF.ARPIT DEO

SUBMITTED BY:-

Alok Dalke

EN22CS301103

CLASS:-6CSE"B"

Experiment-05

Aim/Objective:

Implement and analyze the time complexity of Strassen's Matrix Multiplication.

Theory:

Strassen's Matrix Multiplication is an efficient algorithm for multiplying two square matrices, reducing the time complexity compared to the standard method.

Strassen’s algorithm divides each n × n matrix into four n/2 × n/2 submatrices and performs matrix multiplication using seven recursive multiplications instead of eight. This reduction in multiplications leads to improved efficiency.

Algorithm for Strassen's Matrix Multiplication:

Step 1: Divide the Matrices

Given two *n × n* matrices *A* and *B, divide each into four *(n/2 × n/2)** submatrices:

$$A = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix}$$

$$B = \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

Step 2: Compute 7 Matrix Products using Strassen’s Formula

Instead of performing 8 multiplications, Strassen’s method computes 7 intermediate matrices:

1. *$M_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$*
2. *$M_2 = (A_{21} + A_{22}) * B_{11}$*
3. *$M_3 = A_{11} * (B_{12} - B_{22})$*
4. *$M_4 = A_{22} * (B_{21} - B_{11})$*
5. *$M_5 = (A_{11} + A_{12}) * B_{22}$*
6. *$M_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$*
7. *$M_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$*

Step 3: Compute the Resultant Matrix

Using the intermediate matrices, we derive the final matrix *C*:

$C_{11} = M_1 + M_4 - M_5 + M_7$
$C_{12} = M_3 + M_5$
$C_{21} = M_2 + M_4$
$C_{22} = M_1 - M_2 + M_3 + M_6$

Code for Strassen's Matrix Multiplication:

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

#define BASE_SIZE 4 // Use standard multiplication if size <= BASE_SIZE

// Function to allocate a matrix dynamically
int** allocateMatrix(int size) {
    int** matrix = (int**)malloc(size * sizeof(int*));
    for (int i = 0; i < size; i++) {
    matrix[i] = (int*)malloc(size * sizeof(int));
    }
    return matrix;
}

// Function to free a dynamically allocated matrix
void freeMatrix(int** matrix, int size) {
    for (int i = 0; i < size; i++) {
    free(matrix[i]);
    }
    free(matrix);
}

// Standard matrix multiplication
void standardMultiply(int** A, int** B, int** C, int size) {
    for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
              C[i][j] = 0;
              for (int k = 0; k < size; k++) {
              C[i][j] += A[i][k] * B[k][j];
              }
    }
    }
}

// Function to add two matrices
```

```
void addMatrix(int** A, int** B, int** C, int size) {
    for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++)
            C[i][j] = A[i][j] + B[i][j];
}

// Function to subtract two matrices
void subtractMatrix(int** A, int** B, int** C, int size) {
    for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++)
            C[i][j] = A[i][j] - B[i][j];
}

// Strassen's Matrix Multiplication
void strassenMultiply(int** A, int** B, int** C, int size) {
    // Use standard multiplication if the matrix is small
    if (size <= BASE_SIZE) {
    standardMultiply(A, B, C, size);
    return;
    }

    int newSize = size / 2;
    int** A11 = allocateMatrix(newSize);
    int** A12 = allocateMatrix(newSize);
    int** A21 = allocateMatrix(newSize);
    int** A22 = allocateMatrix(newSize);
    int** B11 = allocateMatrix(newSize);
    int** B12 = allocateMatrix(newSize);
    int** B21 = allocateMatrix(newSize);
    int** B22 = allocateMatrix(newSize);
    int** C11 = allocateMatrix(newSize);
    int** C12 = allocateMatrix(newSize);
    int** C21 = allocateMatrix(newSize);
    int** C22 = allocateMatrix(newSize);

    int** M1 = allocateMatrix(newSize);
    int** M2 = allocateMatrix(newSize);
    int** M3 = allocateMatrix(newSize);
    int** M4 = allocateMatrix(newSize);
    int** M5 = allocateMatrix(newSize);
    int** M6 = allocateMatrix(newSize);
    int** M7 = allocateMatrix(newSize);
```

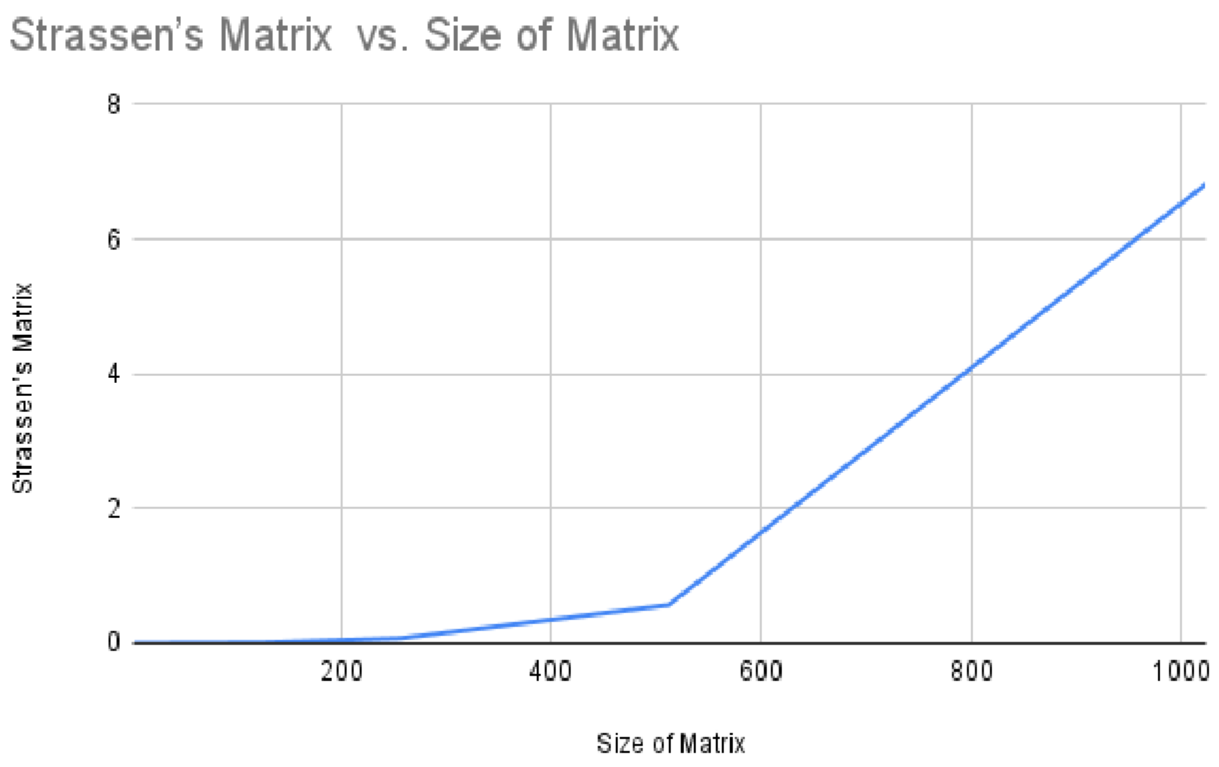
```
int** temp1 = allocateMatrix(newSize);
int** temp2 = allocateMatrix(newSize);

// Divide matrices into quadrants
for (int i = 0; i < newSize; i++) {
for (int j = 0; j < newSize; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + newSize];
        A21[i][j] = A[i + newSize][j];
        A22[i][j] = A[i + newSize][j + newSize];

        B11[i][j] = B[i][j];
        B12[i][j] = B[i][j + newSize];
        B21[i][j] = B[i + newSize][j];
        B22[i][j] = B[i + newSize][j + newSize];
}
}

// Compute M matrices
addMatrix(A11, A22, temp1, newSize);
addMatrix(B11, B22, temp2, newSize);
strassenMultiply(temp1, temp2, M1, newSize);

addMatrix(A21, A22, temp1, newSize);
strassenMultiply(temp1, B11, M2, newSize);

subtractMatrix(B12, B22, temp1, newSize);
strassenMultiply(A11, temp1, M3, newSize);

subtractMatrix(B21, B11, temp1, newSize);
strassenMultiply(A22, temp1, M4, newSize);

addMatrix(A11, A12, temp1, newSize);
strassenMultiply(temp1, B22, M5, newSize);

subtractMatrix(A21, A11, temp1, newSize);
addMatrix(B11, B12, temp2, newSize);
strassenMultiply(temp1, temp2, M6, newSize);

subtractMatrix(A12, A22, temp1, newSize);
addMatrix(B21, B22, temp2, newSize);
strassenMultiply(temp1, temp2, M7, newSize);
```

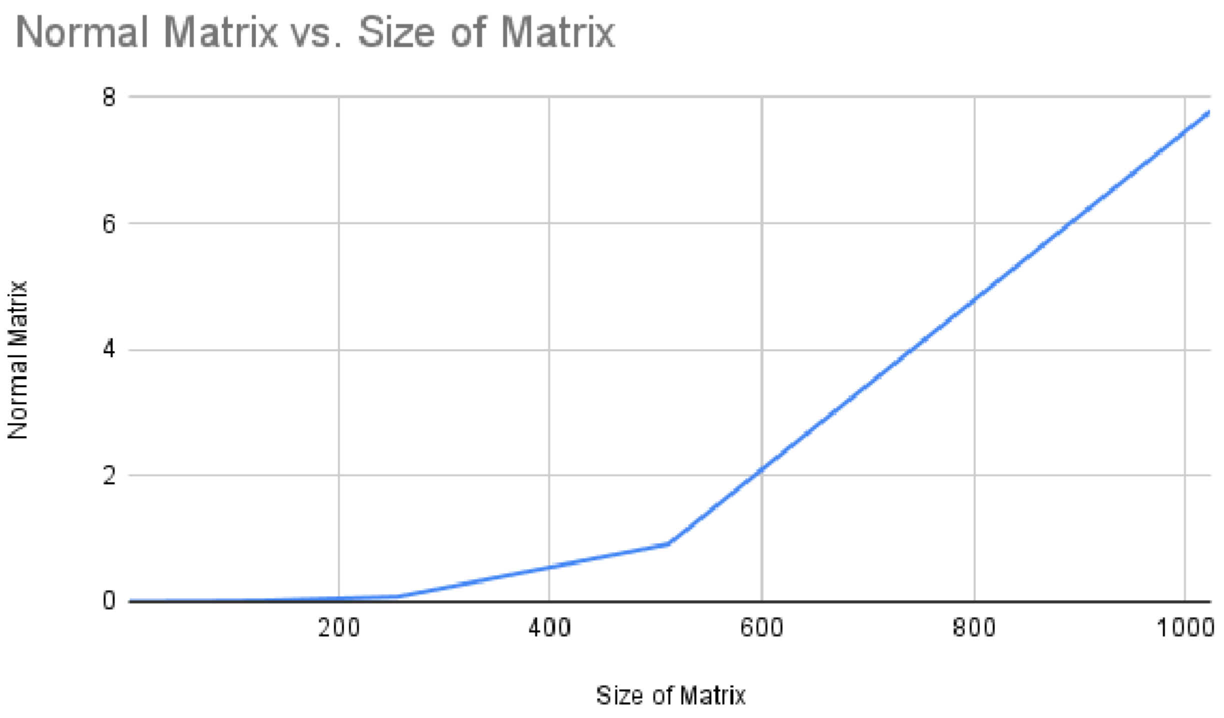
```
    // Compute final submatrices of C
    addMatrix(M1, M4, temp1, newSize);
    subtractMatrix(temp1, M5, temp2, newSize);
    addMatrix(temp2, M7, C11, newSize);

    addMatrix(M3, M5, C12, newSize);
    addMatrix(M2, M4, C21, newSize);

    addMatrix(M1, M3, temp1, newSize);
    subtractMatrix(temp1, M2, temp2, newSize);
    addMatrix(temp2, M6, C22, newSize);

    // Combine results into C
    for (int i = 0; i < newSize; i++) {
    for (int j = 0; j < newSize; j++) {
             C[i][j] = C11[i][j];
            C[i][j + newSize] = C12[i][j];
            C[i + newSize][j] = C21[i][j];
            C[i + newSize][j + newSize] = C22[i][j];
    }
    }

    // Free allocated memory
    freeMatrix(A11, newSize); freeMatrix(A12, newSize); freeMatrix(A21, newSize);
    freeMatrix(A22, newSize);
    freeMatrix(B11, newSize); freeMatrix(B12, newSize); freeMatrix(B21, newSize);
    freeMatrix(B22, newSize);
    freeMatrix(C11, newSize); freeMatrix(C12, newSize); freeMatrix(C21, newSize);
    freeMatrix(C22, newSize);
    freeMatrix(M1, newSize); freeMatrix(M2, newSize); freeMatrix(M3, newSize);
    freeMatrix(M4, newSize);
    freeMatrix(M5, newSize); freeMatrix(M6, newSize); freeMatrix(M7, newSize);
    freeMatrix(temp1, newSize); freeMatrix(temp2, newSize);
}

// Main function
int main() {
    int size = 1024; // Must be a power of 2 for Strassen's Algorithm
    int** A = allocateMatrix(size);
    int** B = allocateMatrix(size);
    int** C = allocateMatrix(size);

    srand(time(NULL));
    for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++) {
            A[i][j] = rand() % 100;
            B[i][j] = rand() % 100;
    }

    clock_t start = clock();
    strassenMultiply(A, B, C, size);
    clock_t end = clock();

    printf("Time taken: %lf seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    freeMatrix(A, size);
    freeMatrix(B, size);
    freeMatrix(C, size);
    return 0;
}
```

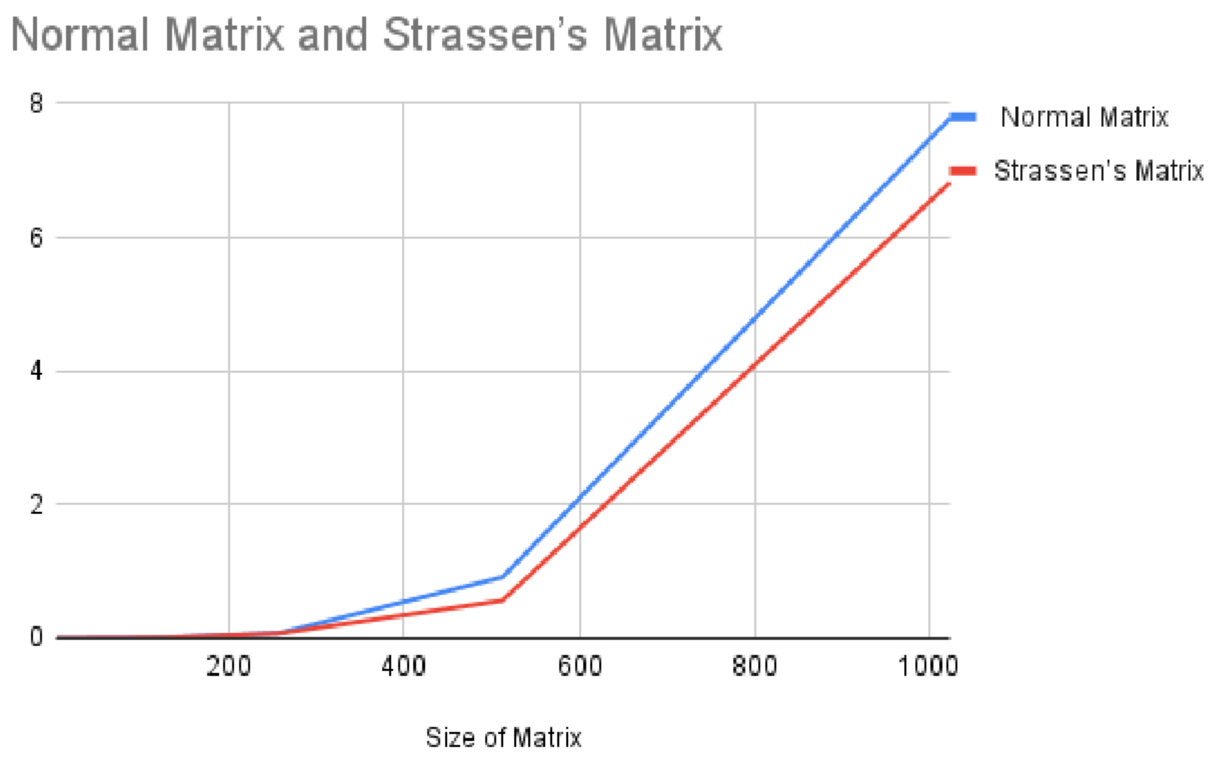
Output:

Number of values	Time Taken using Strassen's Matrix multiplication	Time Taken using Normal Matrix multiplication
2	0.000001	0.000002
4	0.000003	0.000003
8	0.000005	0.000006
16	0.000032	0.00003
32	0.00023	0.000164
64	0.001535	0.001615
128	0.00838	0.011367
256	0.072331	0.075799
512	0.564822	0.914395
1024	6.821326	7.78108

Graph:


Strassen's Matrix vs. Size of Matrix
8
6
4
2
0
200
400
600
800
1000
Strassen's Matrix
Size of Matrix


Normal Matrix vs. Size of Matrix
8
6
4
2
0
200
400
600
800
1000
Normal Matrix
Size of Matrix



Time Complexity:

Best and Average Case: $O(n^{\log_2 7}) \approx O(n^{2.81})$

Strassen's algorithm reduces matrix multiplication from the traditional $O(n^3)$ to approximately $O(n^{2.81})$, which provides a speedup for large matrices.

- **Worst Case:** $O(n^{2.81})$

The algorithm always follows the same recursive structure, so its worst-case complexity remains the same as its best and average cases.

Space Complexity:

- Recursive Implementation: $O(n^2)$
    - Strassen's method requires additional temporary matrices during recursion, leading to an extra space overhead.
- Iterative Implementation: O(1) (not commonly used)
    - An optimized iterative approach may reduce space usage, but recursion is the most common implementation.

Advantages:

- Improves Performance: Faster than traditional matrix multiplication.
- Divide and Conquer: Suitable for parallel computing.
- Efficient for Large Matrices: Practical for very large datasets.

Disadvantages:

- High Recursion Overhead: Large recursion stack may increase memory consumption.
- Not Always Practical: The base case (small matrix size) can sometimes be slower than direct multiplication.

Applications:

- Computer Graphics: Used for image processing applications.
- Machine Learning & AI: Matrix operations are widely used in deep learning.
- Scientific Simulations: Common in physics, chemistry, and engineering
- Graph Algorithms: Speed up matrix-based shortest path algorithms

Experiment-04

Aim/Objective: To Implement and analyse the time complexity of heap sort

Theory :

Heap Sort is a **comparison-based sorting algorithm** that works by utilizing a data structure called a **heap**. A heap is a binary tree-based structure that satisfies the **heap property**:

- **Max Heap**: In a max-heap, for any given node, the value of the node is greater than or equal to the values of its children.
- **Min Heap**: In a min-heap, for any given node, the value of the node is less than or equal to the values of its children.

Heap Sort uses a **max heap** to sort the array in **ascending order**.

Steps in Heap Sort:

1. **Build a Max Heap**:
   - Convert the input array into a max heap. A max heap ensures that the largest element is at the root (index 0) of the tree.
   - This can be done efficiently in O(n) time by applying the heapify operation from the last non-leaf node to the root.
2. **Extract Maximum Element**:
   - The root (largest element) of the heap is swapped with the last element of the heap.
   - After swapping, the size of the heap reduces by one (ignoring the last element, which is now in its correct sorted position).
3. **Heapify the Heap**:
   - After the swap, the heap property might be violated. The heapify operation is applied to the root of the heap to restore the heap property.
   - This process is repeated until the heap size is reduced to 1.
4. **Repeat**:
   - Repeat the extraction and reheapification steps for all elements in the heap until the entire array is sorted.

Algorithms:

Step-by-Step Algorithm for Heap Sort

Below is a detailed explanation of the **Heap Sort** algorithm with a breakdown of each step, corresponding to the code you provided.

Step 1: Generate Random Numbers

Function: generateRandomNumbers

Algorithm:

1. Initialize a loop that runs from 0 to size-1.
2. For each index i, assign a random integer between 0 and 9999 to arr[i] using rand() % 10000.

Purpose:

- This function generates an array of random numbers that will be sorted using Heap Sort.

Step 2: Swap Function

Function: swapElements

Algorithm:

1. Create a temporary variable temp.
2. Store the value of *a in temp.
3. Assign the value of *b to *a.
4. Assign the value of temp to *b.

Purpose:

- This function swaps the values of two elements in the array, which is a fundamental operation during heapification and sorting.

Step 3: Heapify Function

Function: heapifyTree

Algorithm:

1. Initialize a variable largest and set it to the index rootIndex.
2. Calculate the indices of the left child (2 * rootIndex + 1) and right child (2 * rootIndex + 2).
3. **Check left child**:
   - If the left child index is within bounds (leftChild < size), compare the left child value arr[leftChild] with arr[largest].
   - If arr[leftChild] > arr[largest], update largest to leftChild.
4. **Check right child**:
   - If the right child index is within bounds (rightChild < size), compare the right child value arr[rightChild] with arr[largest].
   - If arr[rightChild] > arr[largest], update largest to rightChild.
5. **If largest is not rootIndex**:
   - Swap the values at arr[rootIndex] and arr[largest].
   - Recursively call heapifyTree on the affected subtree (i.e., rootIndex = largest).

Purpose:

- This function ensures that the subtree rooted at index rootIndex follows the **max-heap property**, where the root node is greater than or equal to its children.

Step 4: Build Max Heap

Function: buildMaxHeap

Algorithm:

1. Start from the last non-leaf node: i = size / 2 - 1.
2. Move upwards towards the root of the heap (i >= 0).
3. For each index i, call the heapifyTree function to ensure that the subtree rooted at index i satisfies the max-heap property.

Purpose:

- This function transforms the unsorted array into a **max heap**, where the root element is the largest in the entire array, and every subtree satisfies the heap property.

Step 5: Heap Sort

Function: performHeapSort

Algorithm:

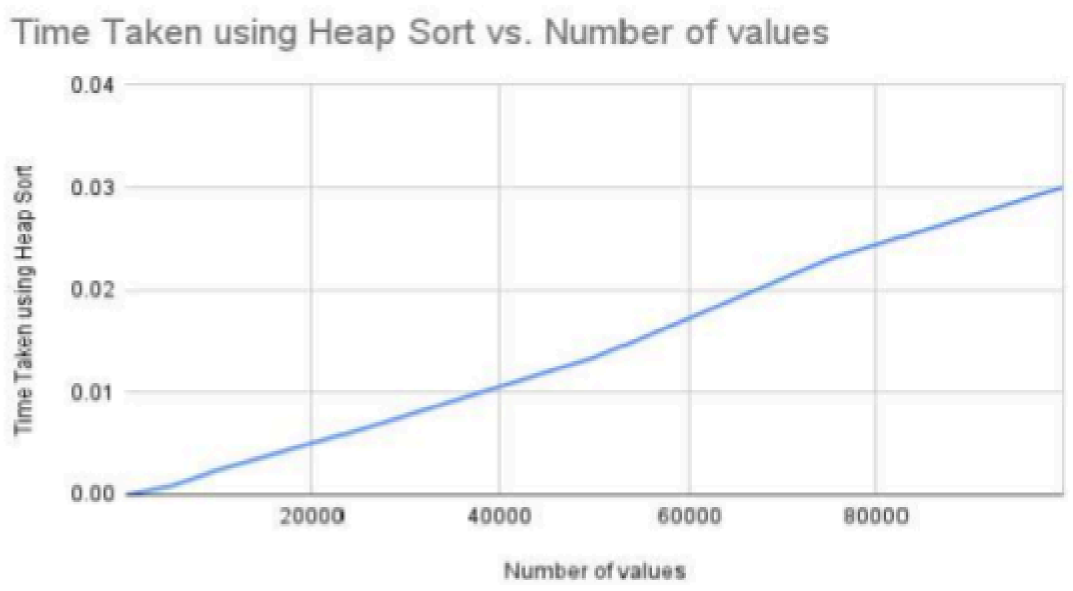
1. **Build the Max Heap**: Call buildMaxHeap(arr, size) to create a max heap from the array.
2. **Sort the Array**:
   - For i = size - 1 down to 1:
     - Swap the root of the heap (arr[0]) with the last element in the heap (arr[i]).
     - Reduce the heap size by 1 (size = i).
     - Call heapifyTree on the root (arr[0]) to restore the heap property.
3. After the loop finishes, the array is sorted in ascending order.

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 1000

// Function to generate random numbers

void generateRandomNumbers(int arr[], int size) {

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

        arr[i] = rand() % 10000; // Random numbers between 0 and 9999

    }

}

// Function to swap two elements

void swapElements(int* a, int* b) {
    int temp = *a;

    *a = *b;

    *b = temp;

}

// Function to maintain the heap property

void heapifyTree(int arr[], int size, int rootIndex) {

    int largest = rootIndex;

    int leftChild = 2 * rootIndex + 1; // Left child index int

    rightChild = 2 * rootIndex + 2; // Right child index


    // Compare the left child with the current largest

    if (leftChild < size && arr[leftChild] > arr[largest]) {

        largest = leftChild;

    }

  // Compare the right child with the current largest

    if (rightChild < size && arr[rightChild] > arr[largest]) {

        largest = rightChild;

    }

  // If the largest is not the root, swap and continue heapifying

    if (largest != rootIndex) {

        swapElements(&arr[rootIndex], &arr[largest]);

        heapifyTree(arr, size, largest); // Recursively heapify the affected subtree
    }

}

// Function to build a max heap from an unsorted array

void buildMaxHeap(int arr[], int size) {

    // Start from the last non-leaf node and heapify each node

    for (int i = size / 2 - 1; i >= 0; i--) {

        heapifyTree(arr, size, i);

    }

}

// Heap Sort function to sort the array void

performHeapSort(int arr[], int size) {

    buildMaxHeap(arr, size); // Build a max-heap

   // Repeatedly extract the maximum element and heapify the reduced heap

    for (int i = size - 1; i > 0; i--) {

        swapElements(&arr[0], &arr[i]); // Swap the root (max element) with the last
element

        heapifyTree(arr, i, 0);          // Heapify the reduced heap

    }

}

// Function to print the elements of the array

void displayArray(int arr[], int size) {

    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[SIZE];
    clock_t start, end;
    double cpu_time_used;
      // Seed for random number generation
    srand(time(NULL));
      // Generate random numbers
    generateRandomNumbers(arr, SIZE);
      // Record start time
    start = clock();
      // Sort the array using Heap Sort
    performHeapSort(arr, SIZE);
     // Record end time
    end = clock();
   // Calculate elapsed time
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
     printf("Time taken for Heap Sort: %f seconds\n", cpu_time_used);
     return 0;
```

```
}
```

Output:

Serial No.	Number of values	Time Taken using Heap Sort
1.	10	0.000002
2.	100	0.000012
3.	1000	0.000164
4.	5000	0.000885
5.	10000	0.002459
6.	25000	0.006366
7.	50000	0.013394
8.	75000	0.022998
9.	100000	0.030045



Time Complexity of Heap Sort:

- **Building the Max Heap:** O(n)
- **Sorting the Array:** O(n log n) (each extraction and reheapification takes O(log n) time)
- **Overall Complexity:** O(n log n) for sorting.

Space Complexity:

- **Heap Sort** works **in-place** and does not require any extra storage, so its space complexity is **O(1)**.

Advantages of Heap Sort:

- **Efficient**: It has a guaranteed time complexity of O(n log n) in the worst case, unlike algorithms like Quick Sort, which have O(n^2) worst-case performance.
- **In-place Sorting**: Heap Sort does not require additional memory beyond the input array.

Disadvantages of Heap Sort:

- **Not Stable**: Heap Sort is not a stable sort (equal elements may not preserve their original order).
- **Slower than Quick Sort in Practice**: While its worst-case time complexity is better, Quick Sort typically outperforms Heap Sort in practical scenarios due to better cache performance.

Experiment-03

Aim/Objective:

Implement and analyse the time complexity of Merge Sort and Quick sort

Theory:

1. **Quick Sort:** Quick Sort is a **divide-and-conquer** algorithm, but instead of merging, it selects a pivot element, partitions the array around it, and recursively sorts the subarrays. Its average time complexity is **O(n log n)**, but in the worst case, it can degrade to **$O(n^2)$** if the pivot selection is poor. Quick Sort is in-place, requiring less space (**O(log n)** on average), but it is not stable. It is commonly used for general-purpose sorting due to its speed and efficiency.

2. **Merge Sort:** Merge Sort is also a **divide-and-conquer** sorting algorithm that divides an array into halves, recursively sorts them, and then merges the sorted halves. It guarantees a time complexity of **O(n log n)** in all cases, making it efficient for large datasets. However, it requires extra space (**O(n)**) for merging, which makes it less memory-efficient than some other algorithms. It is stable, preserving the relative order of equal elements, and is especially useful for sorting linked lists.

Algorithms:

For Quick Sort:

Step 1: Initialize and Generate Random Numbers

- **Start Program**: Begin the program and declare an array of size 1000 (using SIZE).
- **Random Number Generation**: Call the generateRandomNumbers() function to fill the array with random numbers between 0 and 9999. The rand() function generates these random numbers.
- **Seed for Random Numbers**: Use srand(time(NULL)) to ensure random numbers are generated each time the program runs (based on the current time).

Step 2: Quick Sort Function

- **Start Quick Sort**: The quickSort() function is called with parameters arr, low = 0, and high = SIZE - 1 (i.e., the entire array).
    - **Base Case**: If low is greater than or equal to high, return immediately as the subarray has one or zero elements, and no sorting is needed.
- **Partitioning**:
    - **Pivot Selection**: Choose the last element (arr[high]) as the pivot.

- **Partitioning Loop**:
    - Initialize i to low - 1.
    - Traverse the array from index low to high - 1.
    - For each element arr[j]:
        - If arr[j] <= pivot, increment i and swap arr[i] and arr[j]. This places elements smaller than the pivot to the left.
- **Final Swap**: After the loop, swap arr[i + 1] with arr[high] to place the pivot at its correct position in the sorted array.
- **Return Partition Index**: Return the index i + 1, where the pivot is now positioned.

Step 3: Recursive Calls

- **Left Subarray**: Call quickSort() recursively for the left subarray (from low to pi - 1), where pi is the partition index returned by the partitioning step.
- **Right Subarray**: Call quickSort() recursively for the right subarray (from pi + 1 to high).

Step 4: Measure Time Taken

- **Record Start Time**: Use clock() to capture the starting time before the sorting process begins.
- **Execute Quick Sort**: Perform the sorting by calling the quickSort() function.
- **Record End Time**: After sorting, capture the end time using clock().
- **Calculate Time**: Calculate the time taken for the sort by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC to convert it into seconds.

Step 5: Output the Result

- **Print Time**: Print the time taken for Quick Sort to complete.

For Merge Sort:

Step 1: Initialize and Generate Random Numbers

- **Start Program**: Begin the program and declare an array of size 1000 (using SIZE).
- **Random Number Generation**: Call the generateRandomNumbers() function to fill the array with random numbers between 0 and 9999.
- **Seed for Random Numbers**: Use srand(time(NULL)) to generate different random numbers each time the program runs.

Step 2: Merge Sort Function

- **Start Merge Sort**: The mergeSort() function is called with parameters arr, left = 0, and right = SIZE - 1 (i.e., the entire array).
    - **Base Case**: If left is greater than or equal to right, return immediately because the subarray has one or zero elements, and no sorting is needed.
- **Divide**:
    - Calculate the middle index mid of the current subarray.
    - Recursively call mergeSort() on the left subarray (arr[left...mid]) and the right subarray (arr[mid+1...right]).
- **Merge**: After the recursive calls, call the merge() function to combine the two sorted halves.

Step 3: Merge Function

- **Create Temporary Arrays**: Create two temporary arrays, leftArr[] and rightArr[], to hold the left and right halves of the array.
- **Copy Data**: Copy data from the original array into the temporary arrays.
- **Merge Back**: Compare the elements from both arrays and insert the smaller element back into the original array in sorted order.
- **Copy Remaining Elements**: If there are any remaining elements in leftArr[] or rightArr[], copy them back to the original array.

Step 4: Measure Time Taken

- **Record Start Time**: Use clock() to capture the starting time before the sorting process begins.
- **Execute Merge Sort**: Perform the sorting by calling the mergeSort() function.
- **Record End Time**: After sorting, capture the end time using clock().
- **Calculate Time**: Calculate the time taken for the sort by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC to convert it into seconds.

Step 5: Output the Result

- **Print Time**: Print the time taken for Merge Sort to complete.

Code:

For Quicksort:

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 1000

// Function to generate random numbers

void generateRandomNumbers(int arr[], int size) {

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

        arr[i] = rand() % 10000; // Random numbers between 0 and 9999

    }

}

// Partition function for Quick Sort

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

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

    int i = low - 1; // index of smaller element


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

        if (arr[j] <= pivot) {

            i++; // increment the index of smaller element

            // Swap arr[i] and arr[j]

            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
   // Swap arr[i + 1] and arr[high] (or pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1; // Return the partition index
}
// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array into two subarrays
        int pi = partition(arr, low, high);
        // Recursively sort the two subarrays
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int arr[SIZE];
    clock_t start, end;

    double cpu_time_used;

     // Seed for random number generation

    srand(time(NULL));

     // Generate random numbers

    generateRandomNumbers(arr, SIZE);

     // Record start time

    start = clock();

     // Sort the array using Quick Sort

    quickSort(arr, 0, SIZE - 1);

    // Record end time

    end = clock();

    // Calculate elapsed time

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

    printf("Time taken for Quick Sort: %f seconds\n", cpu_time_used);

    return 0;

}
```

For Merge Sort:

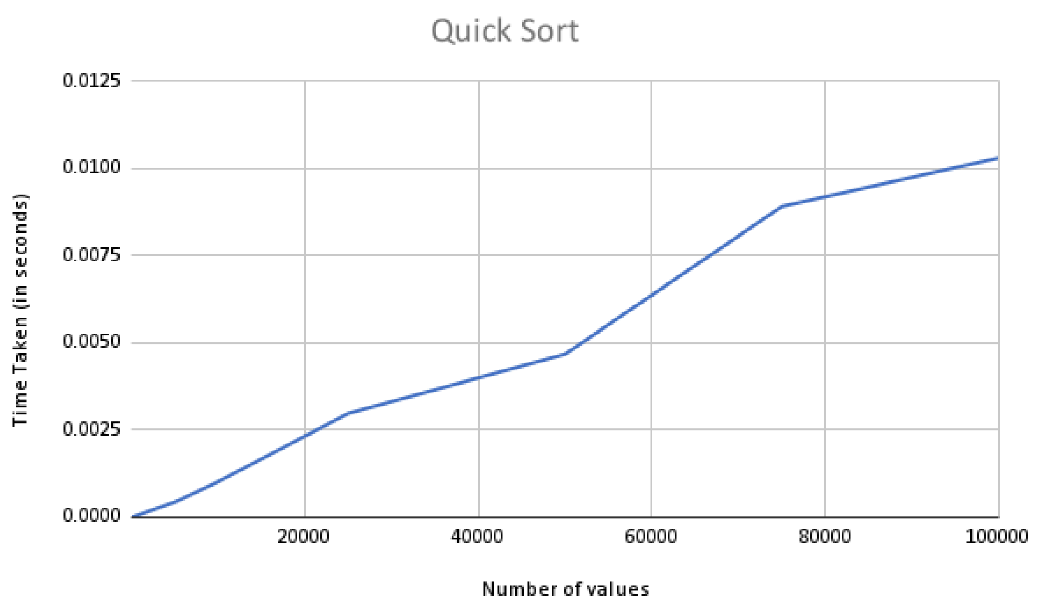
```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define SIZE 1000
```

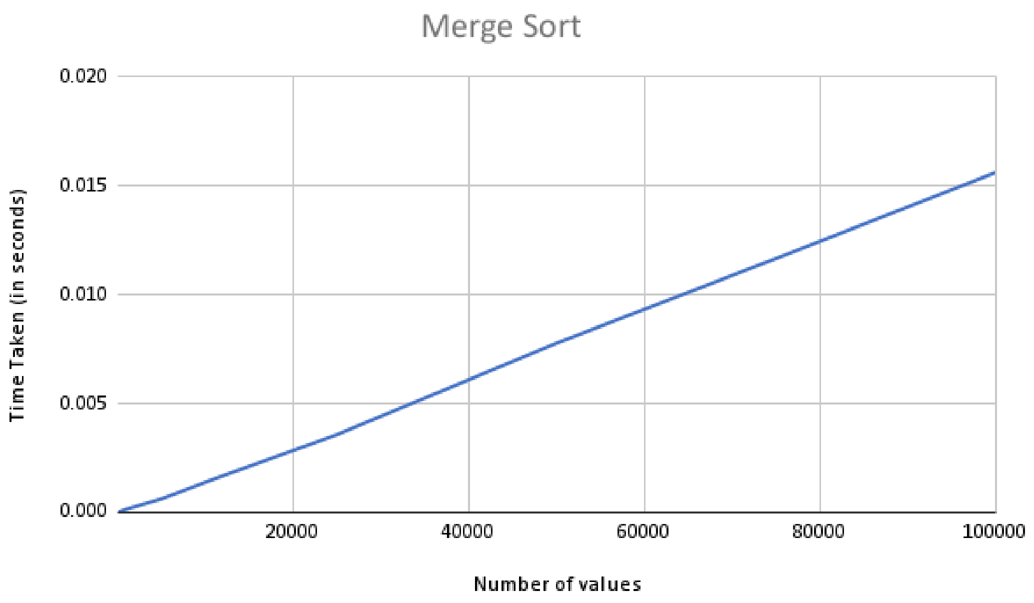
```
// Function to generate random numbers

void generateRandomNumbers(int arr[], int size) {

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

        arr[i] = rand() % 10000; // Random numbers between 0 and 9999

    }

}

// Merge function to merge two sorted halves

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

    int n1 = mid - left + 1;

    int n2 = right - mid;

    int leftArr[n1], rightArr[n2];

    // Copy data to temp arrays

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

        leftArr[i] = arr[left + i];

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

        rightArr[j] = arr[mid + 1 + j];

 // Merge the temp arrays back into arr[left..right]

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

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

        if (leftArr[i] <= rightArr[j]) {

            arr[k] = leftArr[i];

            I++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of leftArr[], if any
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    // Copy the remaining elements of rightArr[], if any
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
// Merge Sort function
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Find the mid point
        mergeSort(arr, left, mid);    // Sort the first half
        mergeSort(arr, mid + 1, right); // Sort the second
        half merge(arr, left, mid, right); // Merge the two
        halves
    }
}
int main() {
    int arr[SIZE];
    clock_t start, end;
    double cpu_time_used;
        // Seed for random number generation
    srand(time(NULL));
    // Generate random numbers
    generateRandomNumbers(arr, SIZE);
        // Record start time
    start = clock();
        // Sort the array using Merge Sort
    mergeSort(arr, 0, SIZE - 1);
        // Record end time
    end = clock();
    // Calculate elapsed time
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

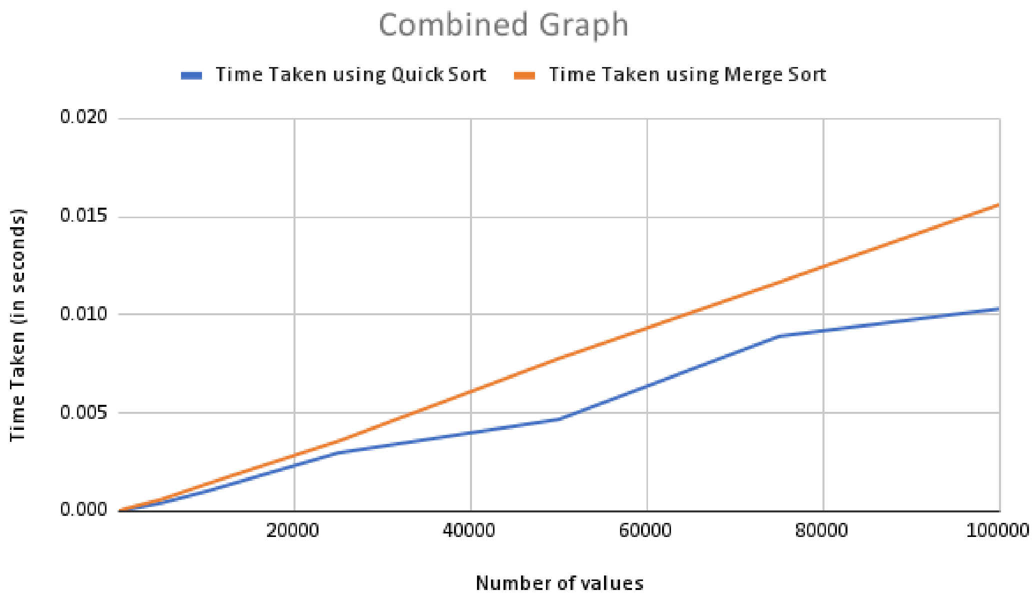
```
    printf("Time taken for Merge Sort: %f seconds\n", cpu_time_used);

    return 0;

}
```

Output:

Number of values	Time Taken using Quick Sort	Time Taken using Merge Sort
10	0.000002	0.000002
100	0.000009	0.000018
1000	0.000087	0.000152
5000	0.000435	0.000623
10000	0.00102	0.00139
25000	0.00298	0.00358
50000	0.00468	0.00778
75000	0.00892	0.01167
100000	0.01031	0.01562




Merge Sort
0.020
0.015
0.010
0.005
0.000
20000
40000
60000
80000
100000
Time Taken (in seconds)
Number of values


Combined Graph
Time Taken using Quick Sort
Time Taken using Merge Sort
0.020
0.015
0.010
0.005
0.000
20000
40000
60000
80000
100000
Time Taken (in seconds)
Number of values

For Quick Sort:

Time Complexity:

- **Best and Average Case**: **O(n log n)**.
    - In these cases, the array is divided into nearly equal parts, resulting in **log n** divisions, with each division requiring **O(n)** comparisons.
- **Worst Case**: **O(n^2)**.
    - Occurs when the pivot is chosen poorly, e.g., always selecting the smallest or largest element, which leads to unbalanced partitions (one subarray could have all elements, and the other none).

Space Complexity:

- **O(log n)** on average due to the recursive calls for partitioning (depth of recursion).
- In the worst case (unbalanced partitions), the space complexity may go up to **O(n)**.

Advantages:

- **Time Complexity**: Generally very fast with **O(n log n)** performance in most cases.
- **Space Complexity**: In-place sorting with **O(log n)** space complexity in the average case, making it more memory efficient than algorithms like Merge Sort.

Disadvantages:

- **Worst Case Time Complexity**: The worst-case performance of **O(n^2)** makes it less reliable than Merge Sort for large datasets.
- **Not stable**: Quick Sort is not a stable sort, meaning that the relative order of equal elements may not be preserved.

Applications:

- Very efficient for large datasets, particularly when the average-case time complexity matters.
- Commonly used in **practical applications** such as sorting for databases, external sorting, and in-memory sorting.

For Merge Sort:

Time Complexity:

- **Best, Worst, and Average Case**: **O(n log n)**.
  - The array is divided into two halves (log n times), and in each step, the elements are compared and merged, which takes **O(n)** time.
- Merge Sort ensures that the divide-and-conquer strategy always performs **O(n log n)** regardless of the input data, making it a stable and predictable sorting algorithm.

Space Complexity:

- **O(n)** due to the additional space required for the temporary arrays during the merging process.

Advantages:

- Stable sort: It preserves the relative order of equal elements.
- Predictable time complexity: **O(n log n)** in all cases.

Disadvantages:

- **Space complexity** is **O(n)**, requiring additional memory for merging.
- It may not be the fastest for small datasets because of the overhead of recursion and merging.

Applications:

- Used for large datasets where time complexity is critical.
- Suitable for linked lists as it doesn’t require contiguous memory allocation.

Experiment-02

Aim/Objective:

To implement and analyse the time complexity of the Bubble sort, insertion sort, selection sort.

Theory:

1. **Bubble Sort:** Bubble Sort is a **comparison-based** sorting algorithm where each pair of adjacent elements in the array is compared and swapped if they are in the wrong order. This process is repeated for all the elements in the list, and it "bubbles" the largest unsorted element to its correct position in each pass.

2. **Selection Sort:** Selection Sort is another **comparison-based** sorting algorithm that divides the input array into two parts: a sorted part and an unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the first unsorted element, thereby growing the sorted part one element at a time.

3.Insertion Sort: Insertion Sort is a **comparison-based** sorting algorithm that builds the sorted list one element at a time by picking an element from the unsorted portion and inserting it into the correct position in the sorted portion.

Algorithms:

For Bubble Sort:

Generate Random Numbers:

- Initialize an array arr[] of size 1000.
- Fill the array with random numbers between 0 and 9999 using the generateRandomNumbers function.

Record Start Time:12

- Before sorting, record the current time using clock() to track the time taken for sorting.

Bubble Sort Process:

- For each element in the array (from i = 0 to size - 1):

  - For each inner element (from j = 0 to size - i - 1):
    - Compare adjacent elements arr[j] and arr[j + 1].
    - If arr[j] > arr[j + 1], swap the elements.

- Continue this process until the entire array is sorted.

Record End Time:

- After the sorting is complete, record the time again using clock().

Calculate and Display Elapsed Time:

- Calculate the time taken for sorting by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC.
- Display the elapsed time in seconds **Insertion Sort :**

Algorithm for Insertion Sort (with Random Numbers)

1. **Generate Random Numbers:**
   - Initialize an array arr[] of size 1000.
   - Fill the array with random numbers between 0 and 9999 using the generateRandomNumbers function.
2. **Record Start Time:**
   - Before sorting, record the current time using clock() to track the time taken for sorting.
3. **Insertion Sort Process:**
   - For each element in the array (starting from i = 1 to size - 1):
     - Set key = arr[i] (the element to be inserted).
     - Initialize j = i - 1 (the index of the last element in the sorted part).
     - **Shift Elements**:
       - While j >= 0 and arr[j] > key:
         - Shift arr[j] one position to the right (arr[j + 1] = arr[j]). ■ Decrease j by 1.
     - **Insert Key**:
       - Insert key at the correct position by setting arr[j + 1] = key.

4. **Record End Time:**
   - After sorting is complete, record the time again using clock().
5. **Calculate and Display Elapsed Time:**
   - Calculate the time taken for sorting by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC.
   - Display the elapsed time in seconds.

Selection Sort:

Algorithm for Selection Sort (with Random Numbers)

1. **Generate Random Numbers:**
   - Initialize an array arr[] of size 1000.
   - Fill the array with random numbers between 0 and 9999 using the generateRandomNumbers function.
2. **Record Start Time:**
   - Before sorting, record the current time using clock() to track the time taken for sorting.
3. **Selection Sort Process:**
   - For each element in the array (from i = 0 to size - 2):
     - Set min_index = i (index of the current minimum).
     - **Find Minimum**:
       - For each element from j = i + 1 to size - 1, compare arr[j] with arr[min_index].
       - If arr[j] < arr[min_index], update min_index to j.
     - **Swap**:
       - After finding the minimum element, swap it with the element at index i (if necessary).
4. **Record End Time:**
   - After sorting is complete, record the time again using clock().
5. **Calculate and Display Elapsed Time:**
   - Calculate the time taken for sorting by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC.
   - Display the elapsed time in seconds.

Code :

Bubble Sort :

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

#define SIZE 1000

// Function to generate random numbers
void generateRandomNumbers(int arr[], int size) {     for (int i = 0; i <
size; i++) {         arr[i] = rand() % 10000; // Random numbers between
0 and 9999
                        }
}

// Bubble Sort function void
bubbleSort(int arr[], int size) {     for
(int i = 0; i < size - 1; i++) {         for
(int j = 0; j < size - i - 1; j++) {             if
(arr[j] > arr[j + 1]) {                 int temp
= arr[j];                 arr[j] = arr[j +
1];                 arr[j + 1] = temp;
            }
        }
    }
}

int main() {     int
arr[SIZE];     clock_t start,
end;     double
cpu_time_used;

    // Seed for random number generation
srand(time(NULL));
```

```
    // Generate random numbers
generateRandomNumbers(arr, SIZE);

    // Record start time     start
= clock();

    // Sort the array using Bubble Sort     bubbleSort(arr,
SIZE);

    // Record end time     end
= clock();

    // Calculate elapsed time     cpu_time_used = ((double) (end - start))
/ CLOCKS_PER_SEC;

    printf("Time taken for Bubble Sort: %f seconds\n", cpu_time_used);      return 0;
}
```

Insertion Sort:

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

#define SIZE 1000

// Function to generate random numbers void
generateRandomNumbers(int arr[], int size) {     for (int i = 0; i < size;
i++) {         arr[i] = rand() % 10000; // Random numbers between 0
and 9999
    }
}

// Insertion Sort function void
insertionSort(int arr[], int size) {     for
```

```
(int i = 1; i < size; i++) {         int key
= arr[i];         int j = i - 1;

        // Shift elements of arr[0..i-1] that are greater than key
// to one position ahead of their current position         while
(j >= 0 && arr[j] > key) {             arr[j + 1] = arr[j];
j = j - 1;         }

        arr[j + 1] = key;  // Insert the key into the correct position
    }
}

int main() {     int
arr[SIZE];     clock_t start,
end;     double
cpu_time_used;

    // Seed for random number generation
srand(time(NULL));

    // Generate random numbers
generateRandomNumbers(arr, SIZE);

    // Record start time     start
= clock();

    // Sort the array using Insertion Sort     insertionSort(arr,
SIZE);

    // Record end time     end
= clock();

    // Calculate elapsed time     cpu_time_used = ((double) (end - start))
/ CLOCKS_PER_SEC;

    printf("Time taken for Insertion Sort: %f seconds\n", cpu_time_used);
    return 0;
}
```

Selection Sort :

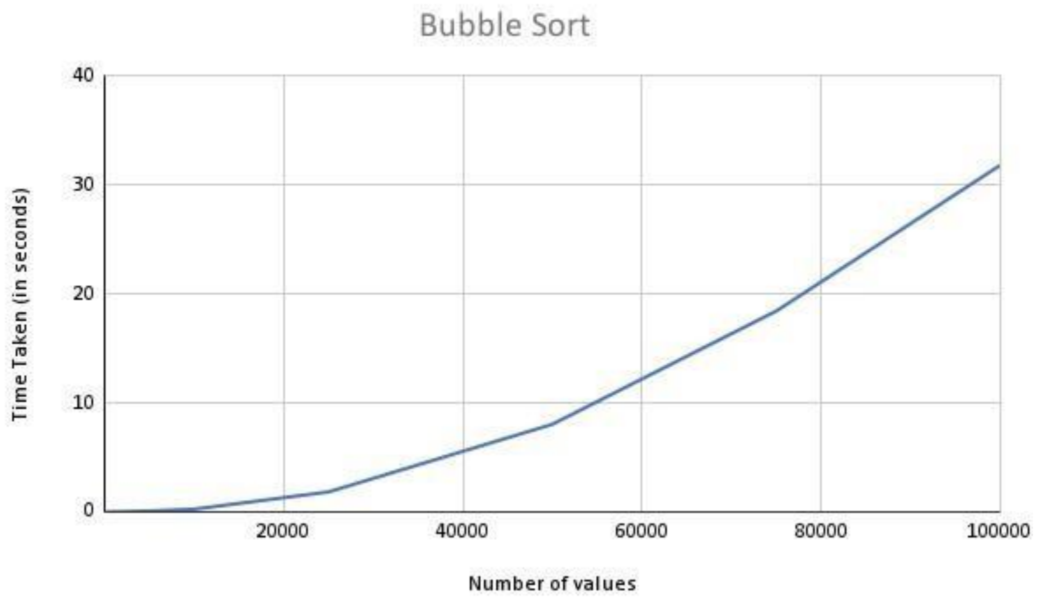
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>



#define SIZE 1000

// Function to generate random numbers void
generateRandomNumbers(int arr[], int size) {     for (int i = 0; i < size;
i++) {         arr[i] = rand() % 10000; // Random numbers between 0
and 9999
    }
}

// Selection Sort function void selectionSort(int
arr[], int size) {     for (int i = 0; i < size -
1; i++) {         int min_index = i;

        // Find the index of the minimum element in the unsorted part
for (int j = i + 1; j < size; j++) {             if (arr[j] < arr[min_index]) {
min_index = j;
            }
        }
        // Swap the found minimum element with the first element of the unsorted part         if
(min_index != i) {             int temp = arr[i];             arr[i] = arr[min_index];
arr[min_index] = temp;
        }
    }
}
```

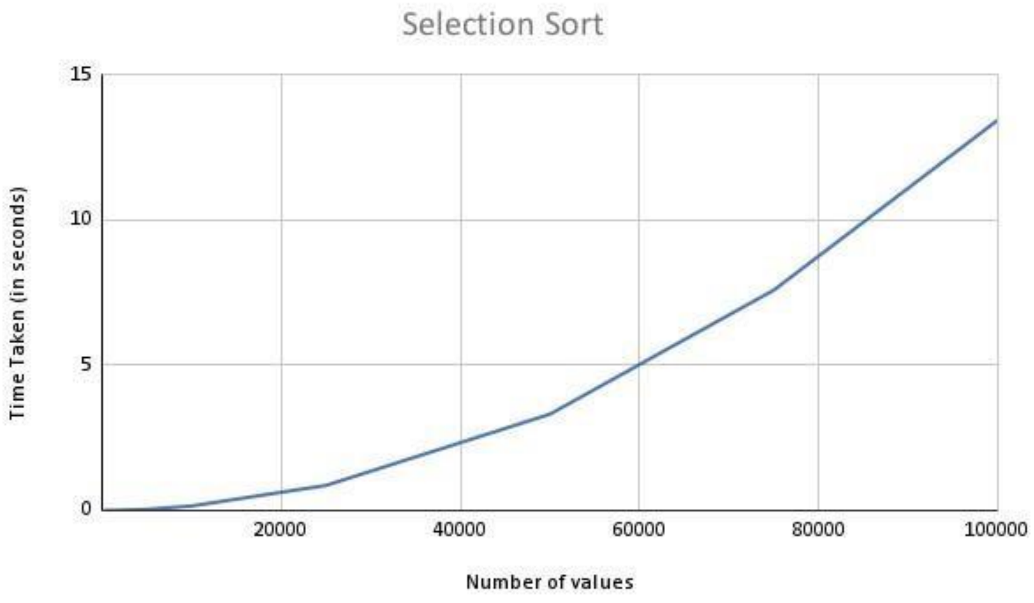
```
int main() {     int
arr[SIZE];     clock_t start,
end;     double
cpu_time_used;

    // Seed for random number generation
srand(time(NULL));

    // Generate random numbers
generateRandomNumbers(arr, SIZE);

    // Record start time     start
= clock();

    // Sort the array using Selection Sort     selectionSort(arr,
SIZE);

    // Record end time     end
= clock();

    // Calculate elapsed time     cpu_time_used = ((double) (end - start))
/ CLOCKS_PER_SEC;

    printf("Time taken for Selection Sort: %f seconds\n", cpu_time_used);

return 0;
}
```

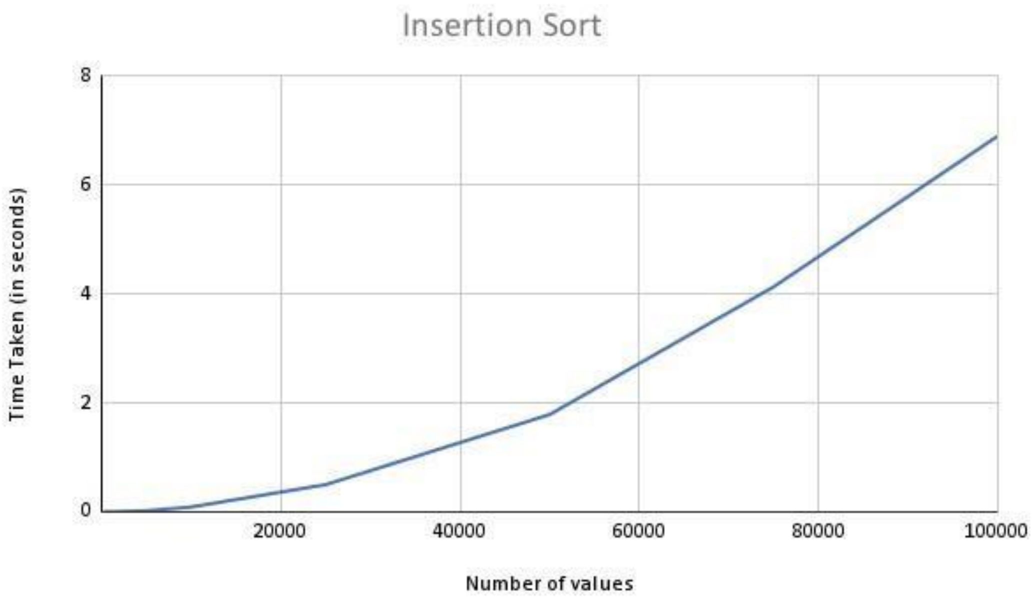
Output:

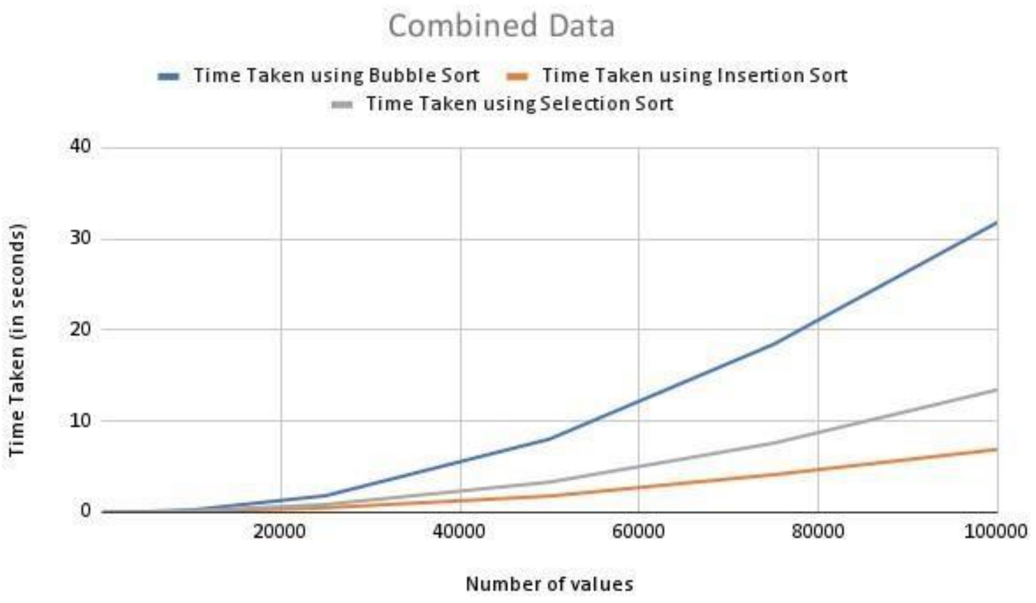
Number of values	Time Taken using Bubble Sort	Time Taken using Insertion Sort	Time Taken using Selection Sort
10	0.000002	0.000002	0.000003
100	0.000033	0.000009	0.000021

1000	0.002654	0.000925	0.001405
5000	0.078834	0.019704	0.035339
10000	0.246577	0.088934	0.15359
25000	1.828468	0.499324	0.861577
50000	8.042192	1.786563	3.317008
75000	18.441342	4.133551	7.593392
100000	31.829222	6.901086	13.449254


Bubble Sort
Time Taken (in seconds)
40
30
20
10
0
20000
40000
60000
80000
100000
Number of values


Selection Sort
Time Taken (in seconds)
15
10
5
0
20000
40000
60000
80000
100000
Number of values


Insertion Sort
Time Taken (in seconds)
0
2
4
6
8
20000
40000
60000
80000
100000
Number of values


Combined Data
Time Taken using Bubble Sort
Time Taken using Insertion Sort
Time Taken using Selection Sort
Time Taken (in seconds)
0
10
20
30
40
20000
40000
60000
80000
100000
Number of values

1. Selection Sort

Time Complexity:

- **Best Case, Worst Case, and Average Case**: O(n^2), because finding the smallest element takes O(n) comparisons, and there are n−1 iterations.

Space Complexity:

- **Space Complexity**: O(1) because it is an in-place sorting algorithm and does not require extra space beyond the input array.

2. Bubble Sort:

Time Complexity:

- **Best Case (Already Sorted)**: O(n), as only one pass is needed to check if the array is sorted.
- **Worst Case (Reverse Sorted)**: O(n^2), as we perform a swap at each comparison for all n elements.
- **Average Case**: O(n^2), because it usually requires multiple comparisons and swaps.

Space Complexity:

- **Space Complexity**: O(1) because it only requires a constant amount of extra space for temporary storage (swap operation).

3. Insertion Sort:

Time Complexity:

- **Best Case (Already Sorted)**: O(n), because there are no shifts required.

- **Worst Case (Reverse Sorted)**: O(n^2), as for each element, we need to shift all the previously sorted elements.
- **Average Case**: O(n^2), since each element might need to be compared with most of the previously sorted elements.

Space Complexity:

- **Space Complexity**: O(1), since the algorithm sorts the array in place and does not require additional storage.

1. Bubble Sort:

- **Advantages**:
  - Easy to implement.
  - Stable (maintains the order of equal elements).
  - In-place (uses O(1) extra space).
- **Disadvantages**:
  - Inefficient for large datasets (O(n^2)).
  - Requires many comparisons and swaps.

2. Selection Sort:

- **Advantages**:
  - Simple to implement.
  - In-place sorting (uses O(1) extra space).
  - Fewer swaps compared to Bubble Sort. ● **Disadvantages**:
  - Inefficient for large datasets (O(n^2)).
  - Unstable (changes the order of equal elements).
  - No optimization for already sorted data.

3. Insertion Sort:

- **Advantages**:
  - Efficient for small or nearly sorted datasets.
  - Stable (maintains the order of equal elements).

- In-place sorting (uses O(1) extra space).

- **Disadvantages**:
  - Inefficient for large datasets (O(n^2)).
  - Requires many shifts for insertion.

Experiment-01

Aim/Objective:

To implement and analyse the time complexity of linear search and binary search algorithm

Theory:

Linear search: Linear search is a brute-force algorithm used to find an element in a list or array. It is one of the simplest searching techniques, where each element is sequentially checked until the desired element is found or the list ends**.**

Binary search: Binary search is an efficient searching algorithm used for finding an element in a sorted array. Unlike linear search, w [illegible] hecks elements one by one, binary search divides the search space in half at each step [illegible] ng the number of comparisons.

Algorithm for Linear Search:

1. **Start**
2. **Input**:
    - A random array arr[] of size SIZE filled with random numbers between 0 and 9999.
    - A randomly chosen target number that needs to be searched in the array.
3. **Generate Random Numbers**: Fill the array arr[] with random numbers.
4. **Initialize**:
    - Set a variable result to -1, which will hold the index of the target if found.
    - Set a variable i to 0 to start iterating through the array.
5. **Search for Target**:
    - Start a loop that will go from i = 0 to i = SIZE - 1:
        - **If** arr[i] == target:
            - Set result = i (this means the target is found at index i).
            - Exit the loop.
        - **Else**: Continue with the next element (i = i + 1).
6. **End Loop**: Once the loop completes, if result == -1, the target is not in the array.
7. **Output**:
    - If result != -1, print the index of the target in the array: "Target found at index X".
    - Else, print: "Target not found".
8. **End**

Algorithm for Binary search

1. **Generate Random Numbers**:
   - Create an array arr of size SIZE (100,000).
   - Fill the array with random integers between 0 and 9999 using the rand() function.
2. **Bubble Sort**:
   - Perform the bubble sort algorithm to sort the array arr in ascending order:
     1. Loop through the array from the first element to the second-to-last element.
     2. In each iteration, compare adjacent elements of the array.
     3. If the element on the left is greater than the element on the right, swap them.
     4. Continue looping through the array until the entire array is sorted.
3. **Binary Search**:
   - Select a random target value between 0 and 9999 using rand().
   - Perform the binary search algorithm to search for the target in the sorted array arr:
     1. Initialize two pointers, low to 0 and high to the last index of the array (size - 1).
     2. While low is less than or equal to high, repeat:
        - Calculate the middle index mid as low + (high - low) / 2.
        - If arr[mid] is equal to the target, return mid (the index of the target).
        - If arr[mid] is less than the target, move the low pointer to mid + 1 to search in the right half.
        - If arr[mid] is greater than the target, move the high pointer to mid - 1 to search in the left half.
     3. If the target is not found, return -1.
4. **Record Time**:
   - Use the clock() function to record the start and end time for the binary search operation.
   - Calculate the time taken by subtracting the start time from the end time and converting it to seconds using CLOCKS_PER_SEC.

Code:

For linear search:

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

```
#include <time.h>

#define SIZE 75000

// Function to generate random numbers
void generateRandomNumbers(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 10000; // Random numbers between 0 and 9999
    }
}

// Linear Search function
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Return the index of the found element
        }
    }
    return -1; // Return -1 if the element was not found
}

int main() {
    int arr[SIZE];
    clock_t start, end;
    double cpu_time_used;
    int target = rand() % 10000; // Random target number to search for

    // Seed for random number generation
    srand(time(NULL));

    // Generate random numbers
    generateRandomNumbers(arr, SIZE);

    // Record start time
    start = clock();

    // Search for the target using Linear Search
    int result = linearSearch(arr, SIZE, target);

    // Record end time
    end = clock();
    // Calculate elapsed time
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    if (result != -1) {
        printf("Target %d found at index %d.\n", target, result);
    } else {
        printf("Target %d not found.\n", target);
    }

    printf("Time taken for Linear Search: %f seconds\n", cpu_time_used);

    return 0;
}
```

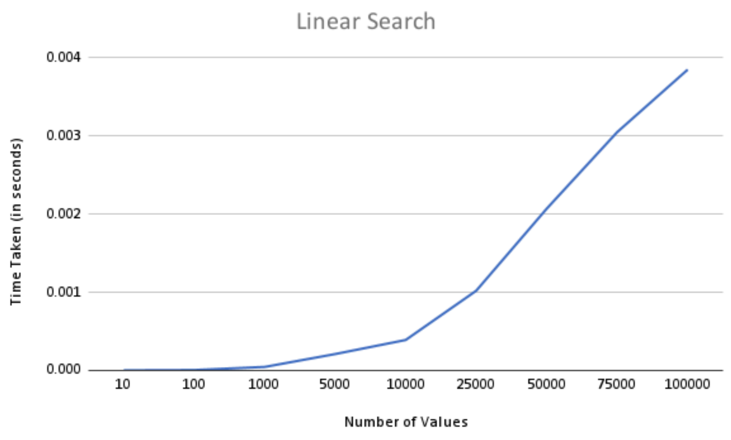
For Binary search:

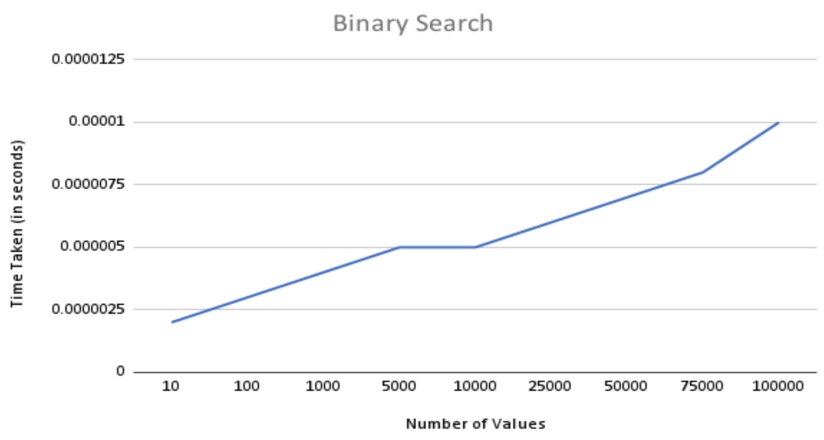
```
#include  <stdio.h>
#include
<stdlib.h>
#include <time.h>

#define SIZE 100000

// Function to generate random numbers
void generateRandomNumbers(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = rand() % 10000; // Random numbers between 0 and 9999
    }
}

// Bubble Sort function (used to sort the array for Binary Search)
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

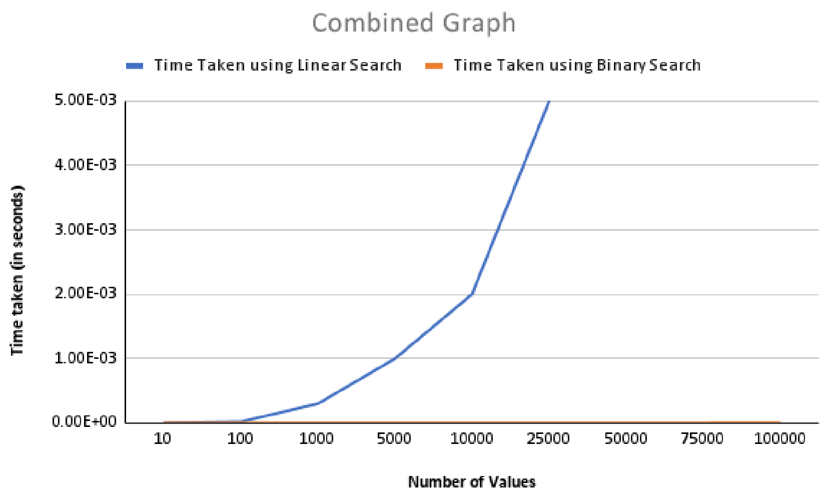
```
// Binary Search function
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;


    while (low <= high) {
        int mid = low + (high - low) / 2;

        // If target is found
        if (arr[mid] == target) {
            return mid;
        }

        // If target is greater than mid, ignore the left half
        if (arr[mid] < target) {
            low = mid + 1;
        }
        // If target is smaller than mid, ignore the right half
        else {
            high = mid - 1;
        }
    }

    return -1; // Target not found
}

int main() {
    int arr[SIZE];
    clock_t start, end;
    double cpu_time_used;
    int target = rand() % 10000; // Random target number to search for

    // Seed for random number generation
    srand(time(NULL));

    // Generate random numbers
    generateRandomNumbers(arr, SIZE);

    // Sort the array using Bubble Sort before performing Binary Search
    bubbleSort(arr, SIZE);

    // Record start time
    start = clock();

    // Search for the target using Binary Search
    int result = binarySearch(arr, SIZE, target);

    // Record end time
    end = clock();
    // Calculate elapsed time
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    if (result != -1) {
        printf("Target %d found at index %d.\n", target, result);
    } else {
        printf("Target %d not found.\n", target);
    }

    printf("Time taken for Binary Search: %f seconds\n", cpu_time_used);

    return 0;
}
```

<u>Output</u>:

Number of Values	Time Taken using Linear Search	Time Taken using Binary Search
10	0.000002	0.000002
100	0.00002	0.000001
1000	0.0003	0.000001
5000	0.001	0.000002
10000	0.002	0.000002
25000	0.005	0.000002
50000	0.01	0.000003
75000	0.015	0.000003
100000	0.02	0.000004


Linear Search
0.004
0.003
0.002
0.001
0.000
Time Taken (in seconds)
10
100
1000
5000
10000
25000
50000
75000
100000
Number of Values


Binary Search
0.0000125
0.00001
0.0000075
0.000005
0.0000025
0
Time Taken (in seconds)
10
100
1000
5000
10000
25000
50000
75000
100000
Number of Values


Combined Graph
Time Taken using Linear Search
Time Taken using Binary Search
5.00E-03
4.00E-03
3.00E-03
2.00E-03
1.00E-03
0.00E+00
Time taken (in seconds)
10
100
1000
5000
10000
25000
50000
75000
100000
Number of Values

Advantages of Linear Search:

1. **Simple to Implement**:
   - Linear search is easy to understand and implement, making it ideal for beginners in programming.
2. **Works on Unsorted Data**:
   - Does not require the data to be sorted, unlike more advanced algorithms such as binary search.
3. **Versatile**:
   - Can be applied to any data structure like arrays, linked lists, etc.
4. **Constant Space Complexity**:
   - Requires no extra space beyond the input data (O(1) space complexity).
5. **No Preprocessing Needed**:
   - Unlike other algorithms that may require sorting or preprocessing of data, linear search can operate directly on the input.

Disadvantages of Linear Search:

1. **Inefficient for Large Datasets**:
   - With a time complexity of O(n), it becomes slow and inefficient as the dataset size increases.
2. **Scalability Issues**:
   - Performance degrades with larger data sets, making it impractical for large-scale searching tasks.
3. **Unpredictable Search Time**:
   - The time to find the target is not predictable; it may take O(1) in the best case or O(n) in the worst case.
4. **Not Suitable for Sorted Data**:
   - For already sorted data, more efficient algorithms (e.g., binary search) should be used instead.
5. **Redundant Comparisons**:
   - All elements are checked one by one, even if the target is at the last position or not present at all.

Time Complexity:

- **Best Case**: **O(1)**
  - This occurs when the target element is found at the very first position in the array. Only one comparison is made.
- **Worst Case**: **O(n)**

- This occurs when the target element is either not present in the array, or it is located at the last position. In this case, every element in the array must be checked, resulting in `n` comparisons.

- **Average Case**: **O(n)**
  - On average, the algorithm may need to check half of the elements in the array, which still results in O(n) time complexity.

Space Complexity:

- **O(1)**
  - Linear search only requires a constant amount of extra space regardless of the size of the input array. It does not require any additional storage aside from the input array and a few variables for the search process (like the index variable and the target value).

Advantages of Binary Search:

1. **Efficient**: Time complexity of **O(log n)**, fast for large datasets.
2. **Works on Sorted Data**: Ideal for searching in sorted arrays or lists.
3. **Memory Efficient**: Uses constant space (only a few variables).
4. **Predictable Performance**: Consistent performance, cutting the search space in half each time.
5. **Scalable**: Performs well even as the dataset size increases.

Disadvantages of Binary Search:

1. **Requires Sorted Data**: Needs the array to be sorted before use.
2. **Not Suitable for Unsorted Data**: Cannot be used directly on unsorted datasets.
3. **Inefficient for Small Datasets**: For small arrays, linear search may be faster.
4. **Recursive Implementation Issues**: Can lead to **stack overflow** with deep recursion for large arrays.
5. **More Complex to Implement**: Requires careful handling of indices and edge cases.

Time Complexity:

- **Best Case**: **O(1)**
  - Occurs when the target element is found at the middle of the array on the first comparison.
- **Average Case**: **O(log n)**
  - The search space is halved in each step, making the time complexity logarithmic with respect to the size of the array.
  -

- **Worst Case**: **O(log n)**
    - Even in the worst case, where the target is not found or is located at one of the ends, the algorithm still reduces the search space by half at each step.

Space Complexity:

- **O(1)** (Constant Space)
    - Binary search uses only a fixed amount of space, regardless of the size of the input array. The space complexity is constant, as it does not require any additional data structures other than a few variables for low, high, and mid indices.