



EXPERIMENT - 1

Algorithms Design and Analysis Lab

Aim

To implement following algorithm using array as a data structure and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

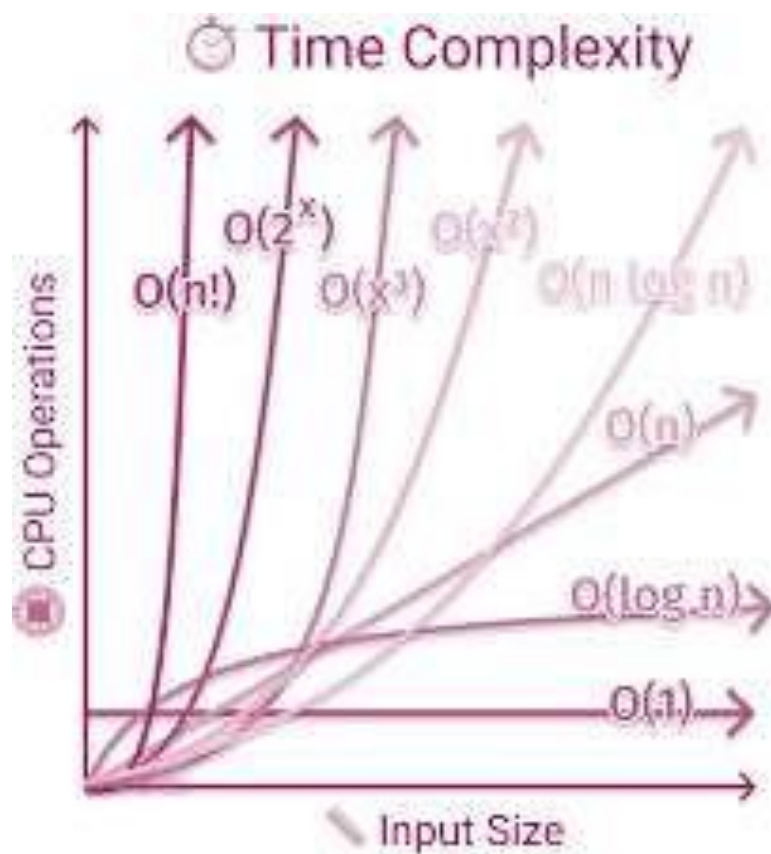
EXPERIMENT – 1

Aim:

To implement following algorithm using array as a data structure and analyse its time complexity.

Theory:

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm.



```
    cout << current_time;
    cout << " seconds has passed since 00:00:00 GMT, Jan 1, 1970";
    return 0;
}
```

Output

```
1635196484 seconds has passed since 00:00:00 GMT, Jan 1, 1970
...Program finished with exit code 0
Press ENTER to exit console.
```

Sorting implementation

Code

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void printArray(int arr[], int size)    {
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()  {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    auto start = chrono::steady_clock::now();
    // unsync the I/O of C and C++.
```

```
ios_base::sync_with_stdio(false);

cout << "Array: ";
printArray(arr, n);
cout << endl;
cout << "Sorted array: ";
printArray(arr, n);
cout << endl;
auto end = chrono::steady_clock::now();

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;

cout << "Time difference is: " << time_taken << setprecision(6);
return 0;
}
```

Output

```
Array: 64 34 25 12 22 11 90
Sorted array: 64 34 25 12 22 11 90
Time difference is: 6.88e-05
...Program finished with exit code 0
Press ENTER to exit console.□
```

Merge Sort:

Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n subsists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge subsists to produce new sorted subsists until there is only 1 subsist remaining. This will be the sorted list.

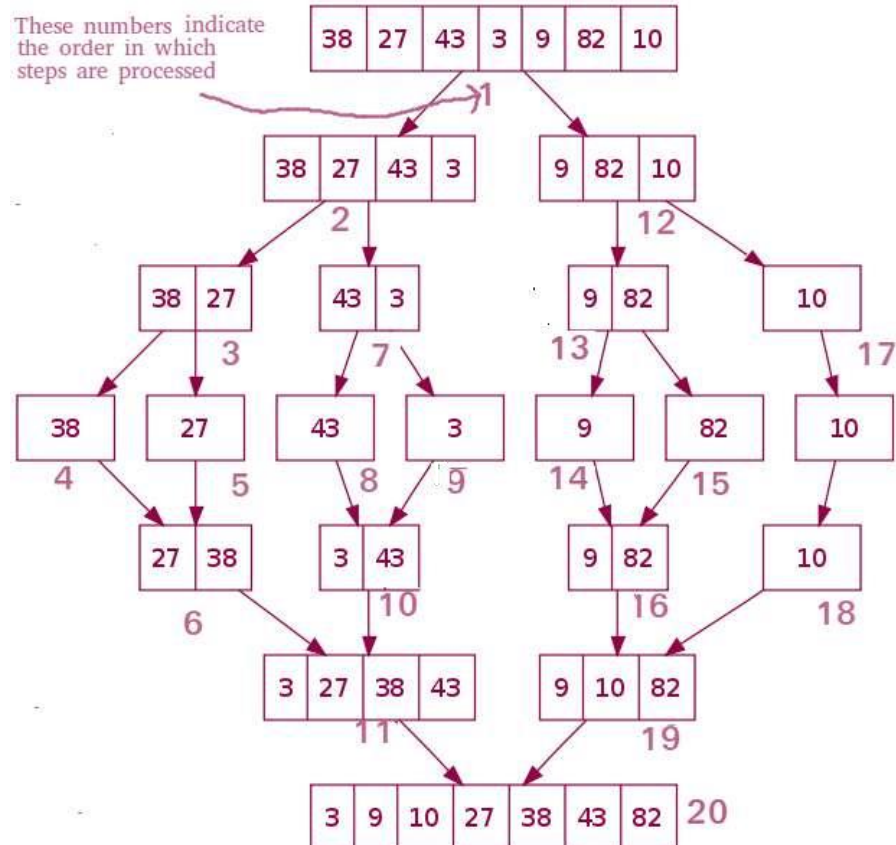
Pseudo code

MergeSort(arr[], l, r)

If $r > l$

1. Find the middle point to divide the array into two halves:
middle $m = l + (r-l)/2$
2. Call mergeSort for first half:
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)

Example:



Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const midIdx, int const right) {
    int temp[right - left + 1]; // sorted Arr

    int start = left, mid = midIdx + 1, tempIdx = 0;

    // sorted merging of left and right arrays
    while (start <= midIdx && mid <= right) {
        if (arr[start] <= arr[mid]) {
            temp[tempIdx] = arr[start];
            tempIdx++;
            start++;
        }
        else {
            temp[tempIdx] = arr[mid];
            tempIdx++;
            mid++;
        }
    }

    // check and add for remaining element in left arr
    while (start <= midIdx) {
        temp[tempIdx] = arr[start];
        tempIdx++;

        start++;
    }

    // check and add for remaining element in right arr
    while (mid <= right) {
        temp[tempIdx] = arr[mid];
        tempIdx++;
        mid++;
    }

    for (int i = left; i <= right; i++) {
        arr[i] = temp[i - left];
    }
}

void mergeSort(int array[], int const begin, int const end) {
```

```
        if (begin >= end) {
            return; // Returns recursively
        }
        auto mid = begin + (end - begin) / 2;
        mergeSort(array, begin, mid);
        mergeSort(array, mid + 1, end);
        merge(array, begin, mid, end);
    }

    void printArr(int arr[], int size) {
        for (auto i = 0; i < size; i++) {
            cout << arr[i] << " ";
        }
        cout << "\n\n" << endl;
    }

    int main() {
        int n = rand() % 100;
        int arr[n];
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 100;
        }
        cout << "Given array: \n";
        printArr(arr, n);

        auto start = chrono::high_resolution_clock::now();
        // unsync the I/O of C and C++.
        ios_base::sync_with_stdio(false);

        mergeSort(arr, 0, n - 1);

        auto end = chrono::high_resolution_clock::now();

        cout << "\nSorted array: \n";
        printArr(arr, n);

        double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
        time_taken *= 1e-9;

        cout << "Time difference is: " << time_taken << setprecision(6);

        return 0;
    }
```

Output:

```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78

Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98

Time difference is: 3.4026e-05

...Program finished with exit code 0
Press ENTER to exit console.
```


Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// merging 2 arrays
void merge(int arr[], int const left, int const mid, int const right) {
    auto const subArr1 = mid - left + 1;
    auto const subArr2 = right - mid;

    auto *leftArr = new int[subArr1],
        *rightArr = new int[subArr2];

    for (auto i = 0; i < subArr1; i++)
        leftArr[i] = arr[left + i];

    for (auto j = 0; j < subArr2; j++)
        rightArr[j] = arr[mid + 1 + j];

    auto indexOfSubArr1 = 0,
        indexOfSubArr2 = 0;
    int indexOfMergedArr = left;

    // sorted merging of left and right arrays
    while (indexOfSubArr1 < subArr1 && indexOfSubArr2 < subArr2) {
        if (leftArr[indexOfSubArr1] <= rightArr[indexOfSubArr2]) {
            arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
            indexOfSubArr1++;
        }
        else {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
        }
        indexOfMergedArr++;
    }

    // check and add for remaining element in left arr
    while (indexOfSubArr1 < subArr1) {
        arr[indexOfMergedArr] = leftArr[indexOfSubArr1];
        indexOfSubArr1++;
        indexOfMergedArr++;
    }
}
```

```
        // check and add for remaining element in left arr
        while (indexOfSubArr2 < subArr2) {
            arr[indexOfMergedArr] = rightArr[indexOfSubArr2];
            indexOfSubArr2++;
            indexOfMergedArr++;
        }
    }

void mergeSort(int array[], int const begin, int const end) {
    if (begin >= end) {
        return; // Returns recursively
    }
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

void printArr(int arr[], int size) {
    for (auto i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n\n" << endl;
}

int main() {
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }
    cout << "Given array: \n";
    printArr(arr, n);

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    mergeSort(arr, 0, n - 1);

    auto end = chrono::high_resolution_clock::now();

    cout << "\nSorted array: \n";
```

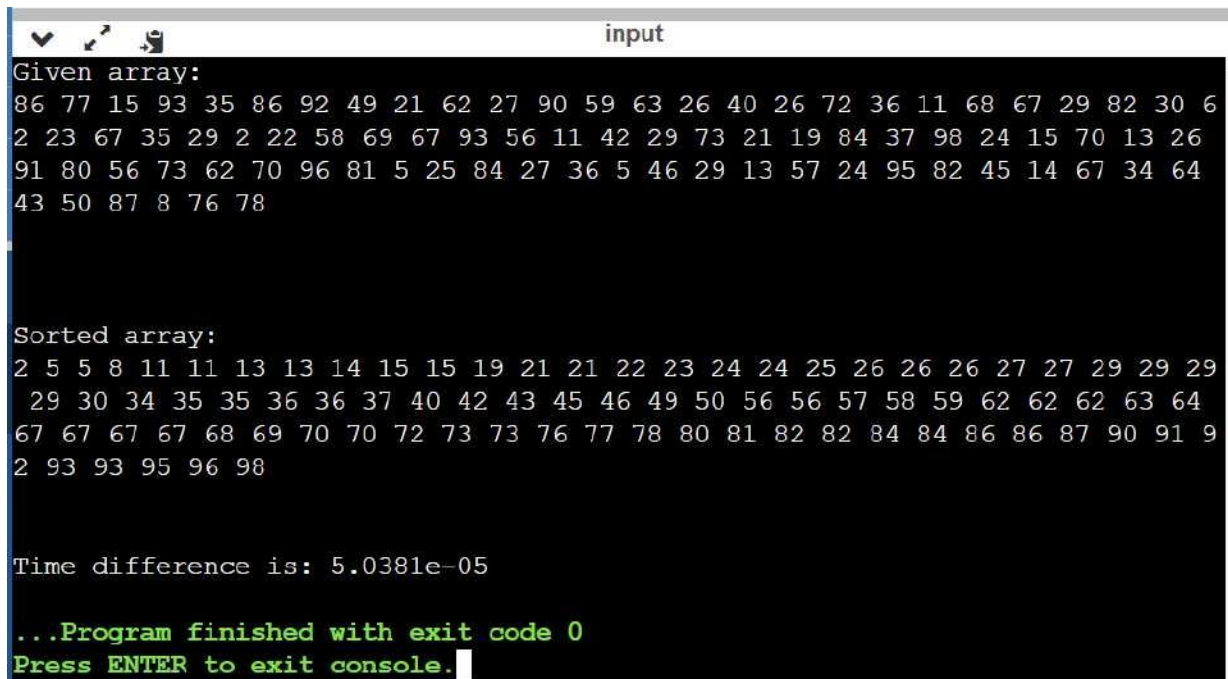
```
    printArr(arr, n);

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:



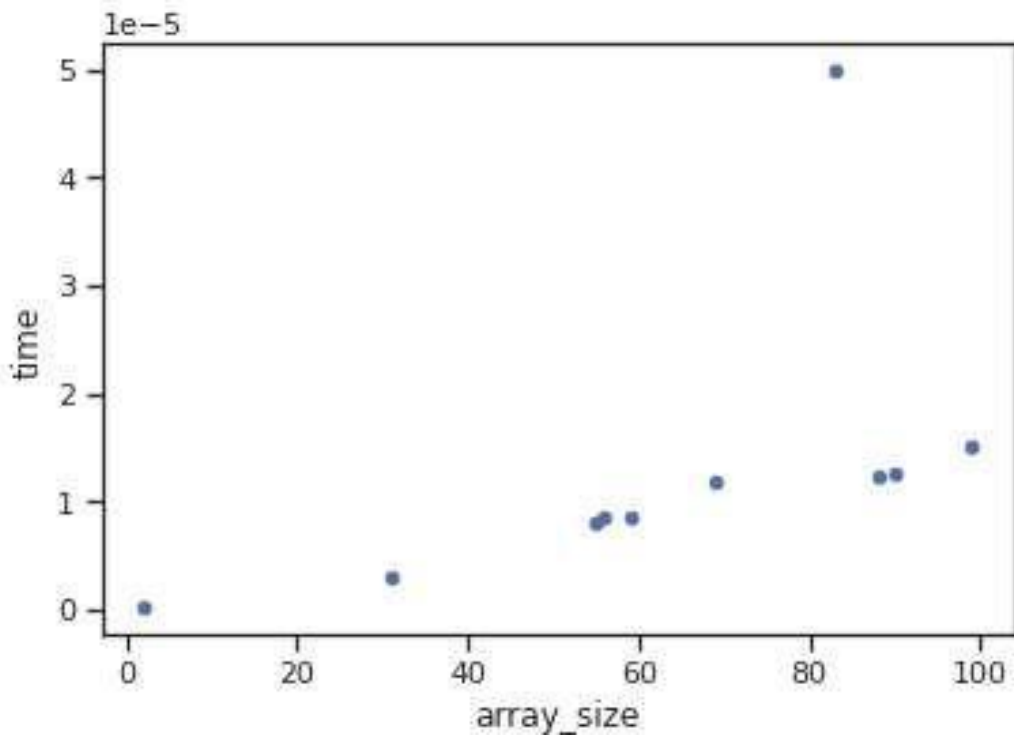
```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78

Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98

Time difference is: 5.0381e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

```
arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]  
time = [5.0004e-05, 1.2248e-05, 8.572e-06, 1.1836e-05, 1.2539e-05, 8.589e-06,  
1.34e-07, 1.5209e-05, 7.996e-06, 3.056e-06]
```



Quick Sort:

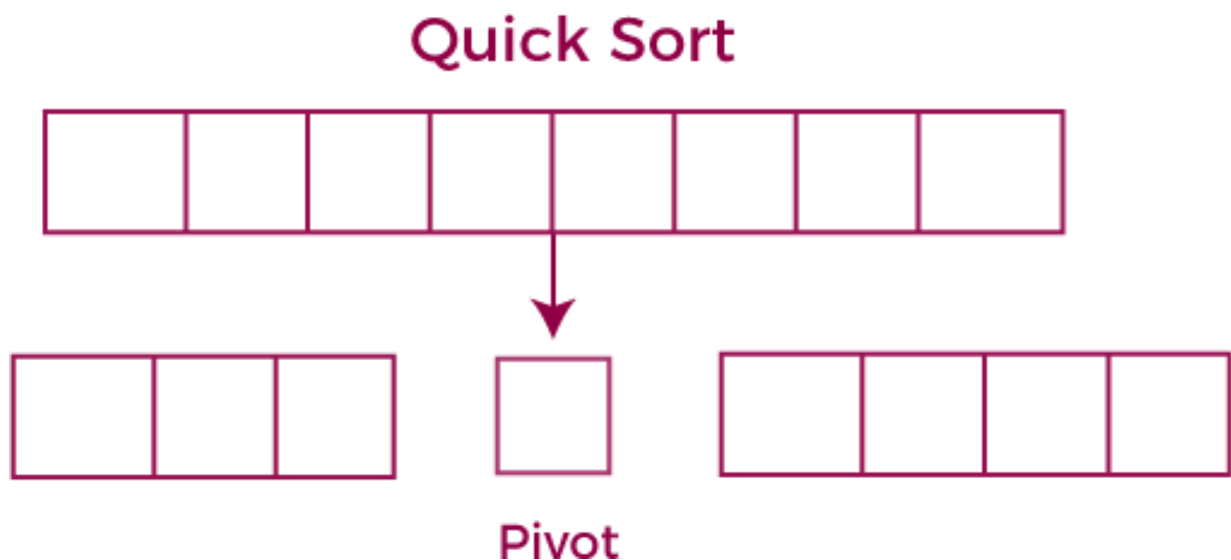
Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes $n \log n$ comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element or the leftmost element of the given array.
- Select median as the pivot element.

Pseudo code

Algorithm:

```
1. QUICKSORT (array A, start, end)
2. {
3.   1 if (start < end)
4.   2 {
5.     3 p = partition(A, start, end)
6.     4 QUICKSORT (A, start, p - 1)
7.     5 QUICKSORT (A, p + 1, end)
8.   6 }
9. }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in place.

```
1. PARTITION (array A, start, end)
2. {
3.   1 pivot ? A[end]
4.   2 i ? start-1
5.   3 for j ? start to end -1 {
6.     4 do if (A[j] < pivot) {
7.       5 then i ? i + 1
8.       6 swap A[i] with A[j]
9.     7 }}
10.      8 swap A[i+1] with A[end]
11.      9 return i+1
12. }
```

Result and Analysis

Time complexity

| Case | Time Complexity |
|--------------|---------------------|
| Best Case | $O(n \cdot \log n)$ |
| Average Case | $O(n \cdot \log n)$ |
| Worst Case | $O(n^2)$ |

- **Best Case Complexity** - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **$O(n \cdot \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **$O(n \cdot \log n)$** .
- **Worst Case Complexity** - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$** .

Space Complexity

| Space Complexity | $O(n \cdot \log n)$ |
|------------------|---------------------|
| Stable | NO |

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>

using namespace std;

// A utility function to swap two elements
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// last element pivot and lower in 1 half and greater in 1 half divide
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);    // Index of smaller element and indicates the right
    position of pivot found so far

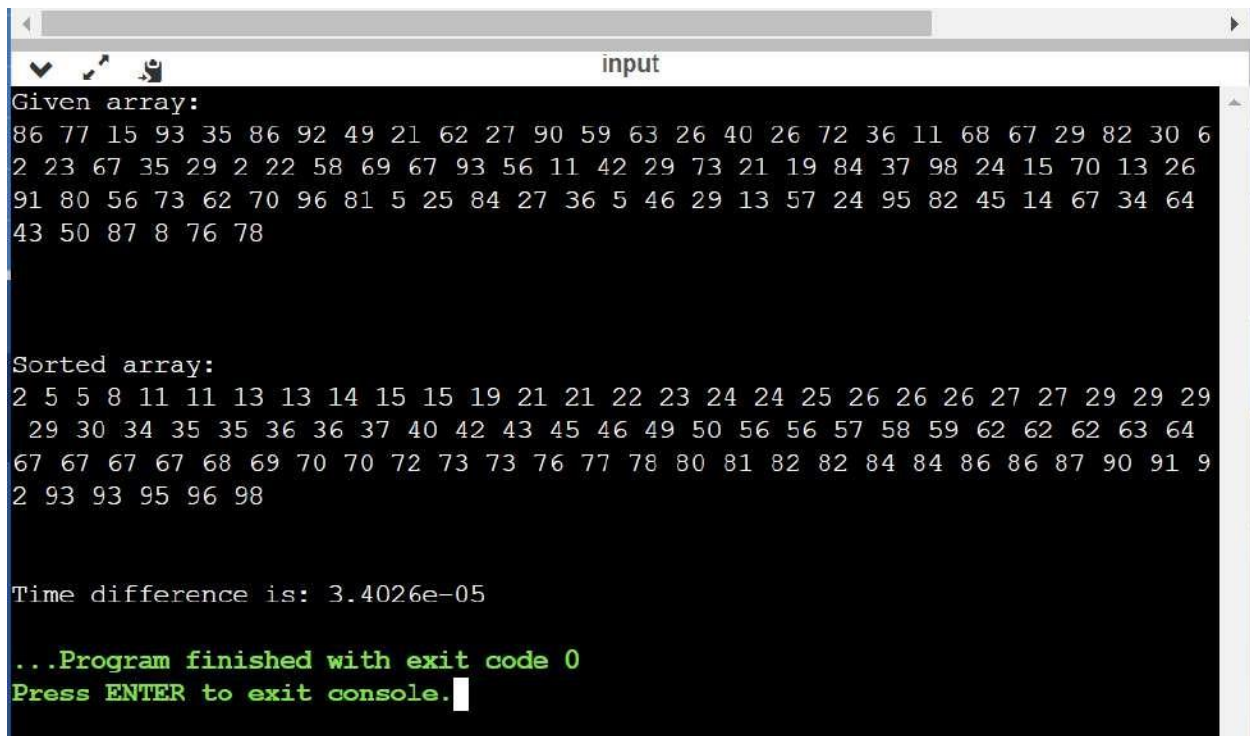
    for (int j = low; j <= high - 1; j++)
    {
        // j == current element
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int p = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, p - 1);
        quickSort(arr, p + 1, high);
    }
}
```



```
    }  
}  
  
void printArray(int arr[], int size)  
{  
    int i;  
    for (i = 0; i < size; i++)  
        cout << arr[i] << " ";  
    cout << "\n\n" << endl;  
}  
  
int main()  
{  
    int n = rand() % 100;  
    int arr[n];  
    for (int i = 0; i < n; i++)  
    {  
        arr[i] = rand() % 100;  
    }  
    cout << "Given array: \n";  
    printArray(arr, n);  
  
    auto start = chrono::high_resolution_clock::now();  
  
    quickSort(arr, 0, n - 1);  
  
    auto end = chrono::high_resolution_clock::now();  
  
    cout << "\nSorted array: \n";  
    printArray(arr, n);  
  
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -  
start).count();  
    time_taken *= 1e-9;  
  
    cout << "Time difference is: " << time_taken << setprecision(6);  
  
    return 0;  
}
```

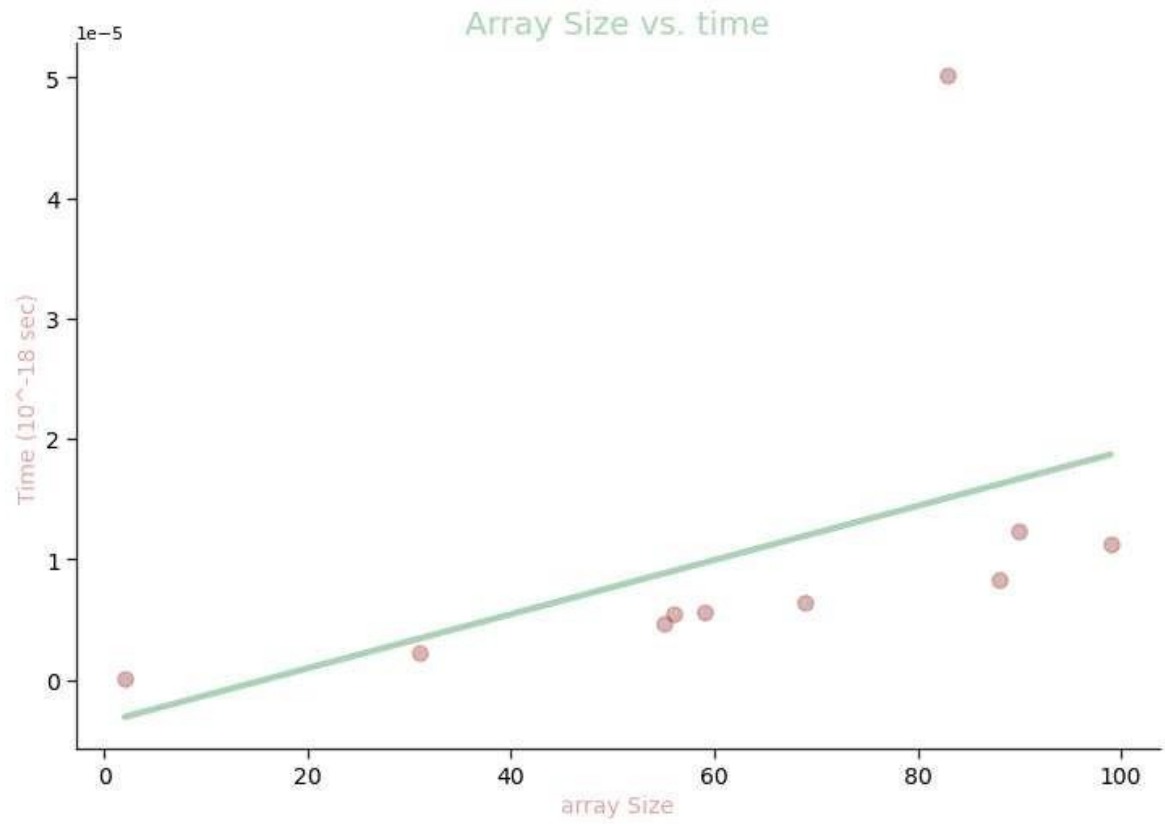
Output:

```
input
Given array:
86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29 82 30 6
2 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70 13 26
91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67 34 64
43 50 87 8 76 78

Sorted array:
2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26 27 27 29 29 29
29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 62 62 62 63 64
67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86 86 87 90 91 9
2 93 93 95 96 98

Time difference is: 3.4026e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Quick Sort

Bubble Sort:

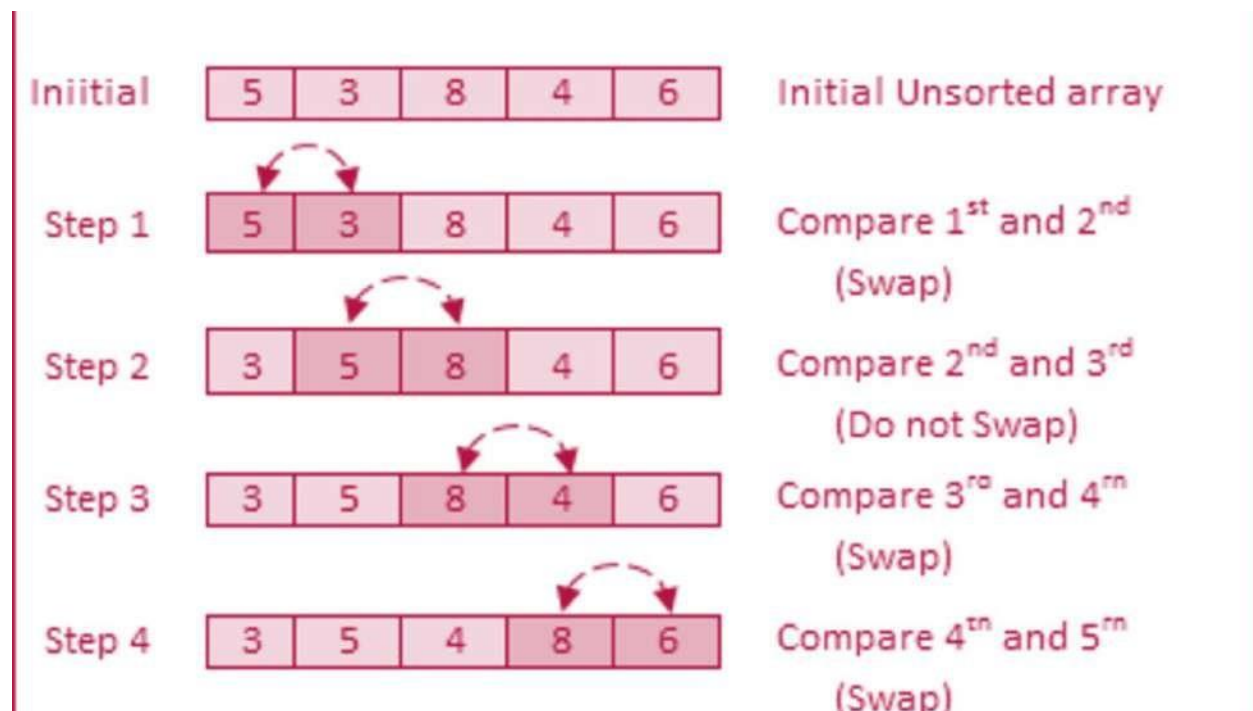
This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared, and the elements are swapped if they are not in order.

Pseudo code

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort (list)
  for all elements of list
    if list[i] > list[i+1]
      swap (list[i], list [i+1])
    end if
  end for
  return list
end BubbleSort
```

Example:



Result and Analysis

Worst and Average Case Time Complexity: $O(n^2)$. Worst case occurs when array is reverse sorted and its time complexity is $O(n^2)$ and Best case occurs when array is already sorted and its time complexity is $O(n)$ (linear time). This sort takes just $O(1)$ extra space.

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void bubbleSort(int arr[], int n) {
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false) { // no swap means array sorted so break out
            break;
        }
    }
}

void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();

    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    bubbleSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78

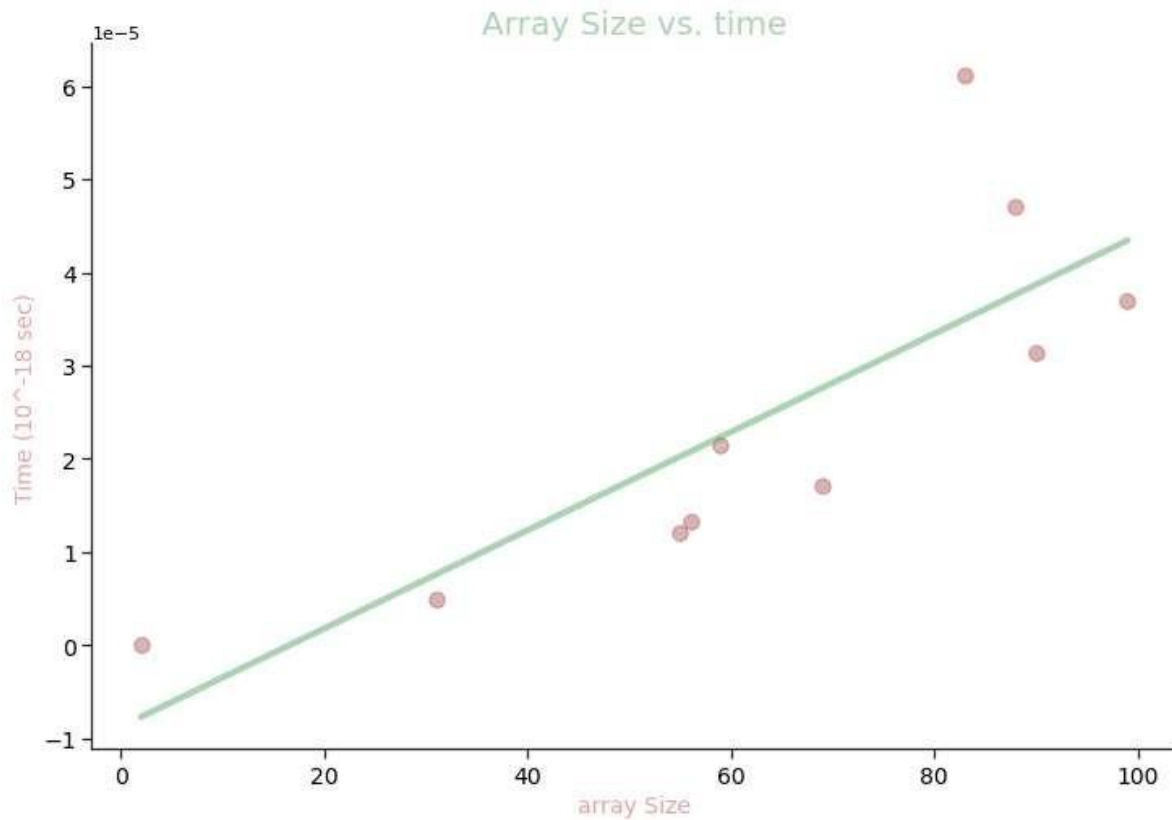
Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98

Time difference is: 5.3534e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [6.1129e-05, 4.7125e-05, 1.3298e-05, 1.7092e-05, 3.147e-05, 2.1538e-05, 8.6e-08, 3.6987e-05, 1.2126e-05, 4.909e-06]



Bubble Sort

Viva Questions

1. How can the best-case efficiency of bubble sort be improved?

Ans.

Some iterations can be skipped if the list is sorted, hence efficiency improves to $O(n)$.

A better version of bubble sort, known as modified bubble sort, includes a flag that is set if an exchange is made after an entire pass over the array. If no exchange is made, then it should be clear that the array is already in order because no two elements need to be switched.

2. Is bubble sorting a stable sort?

Ans.

Yes

sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

3. How much time will bubble sort take if all the elements are same?

Ans.

$O(n)$

Bubble sort has a worst-case and average complexity of $O(n^2)$, where n is the number of items being sorted. Most practical sorting algorithms have substantially better worst-case or average complexity, often $O(n \log n)$.

Bucket Sort:

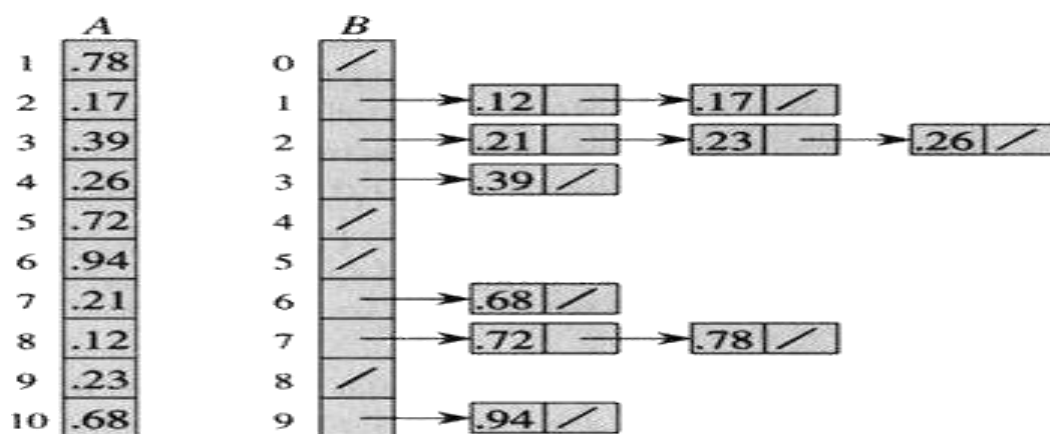
Bucket sort, or **bin sort**, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

It is a distribution sort, a generalization of pigeonhole sort, and is a cousin of radix sort in the most-to-least significant digit flavour. Bucket sort can be implemented with comparisons and therefore can also be considered a comparison sort algorithm.

It works as follows:

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

. This will be the sorted list.



Pseudo code

The code assumes that input is an n -element array A and each element in A satisfies $0 \leq A[i] \leq 1$. We also need an auxiliary array $B[0 \dots n-1]$ for linked-lists (buckets).

BUCKET SORT (A)

1. $n \leftarrow \text{length}[A]$
2. For $i = 1$ to n do
3. Insert $A[i]$ into list $B[A[i]/b]$ where b is the bucket size
4. For $i = 0$ to $n-1$ do
5. Sort list B with Insertion sort
6. Concatenate the lists $B[0], B[1] \dots B[n-1]$ together in order.

Example:



Result and Analysis

In the above Bucket sort algorithm, we observe in the *best case*, the algorithm distributes the elements uniformly between buckets, a few elements are placed on each bucket and sorting the buckets is **$O(1)$** . Rearranging the elements is one more run through the initial list.

$$\begin{aligned}
 T(n) &= [\text{time to insert } n \text{ elements in array } A] + [\text{time to go through auxiliary array } B \\
 &\quad [0 \dots n-1] * (\text{Sort by INSERTION_SORT}) \\
 &= O(n) + (n-1) \cdot O(n) \\
 &= O(n^2)
 \end{aligned}$$

In the *worst case*, the elements are sent all to the same bucket, making the process take **$O(n^2)$** .

Source Code:

```

#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void bucketSort(float arr[], int n) {
    vector<float> bucket[n];

    // put elements in respective buckets
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i]; // Index in bucket
        bucket[bi].push_back(arr[i]);
    }

    for (int i = 0; i < n; i++) // sorting each buckets
        sort(bucket[i].begin(), bucket[i].end());

    int index = 0;
    for (int i = 0; i < n; i++) // Concatenate all buckets into arr[]
        for (int j = 0; j < bucket[i].size(); j++)
            arr[index++] = bucket[i][j];
}

```

```
void printArray(float arr[], int size)    {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);
    int n = rand() % 100;
    float arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = (float(rand())/float((RAND_MAX)));
    }
    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    bucketSort(arr, n); // sort arr

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

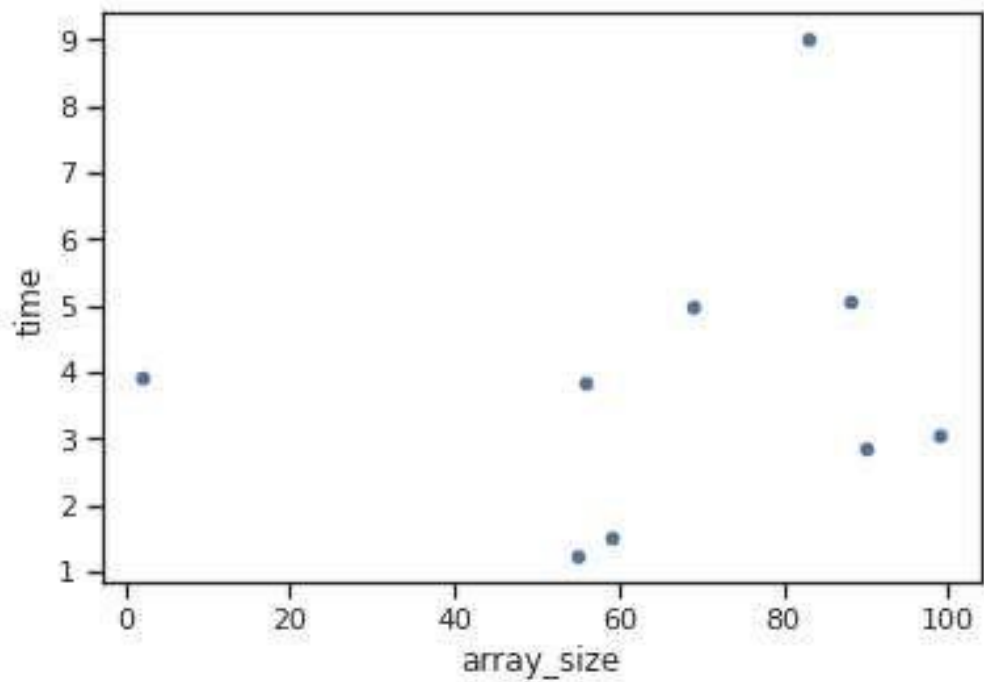
Output:

```
input
Array: 0.394383 0.783099 0.79844 0.911647 0.197551 0.335223 0.76823 0.277775
0.55397 0.477397 0.628871 0.364784 0.513401 0.95223 0.916195 0.635712 0.717
297 0.141603 0.606969 0.0163006 0.242887 0.137232 0.804177 0.156679 0.400944
0.12979 0.108809 0.998924 0.218257 0.512932 0.839112 0.61264 0.296032 0.637
552 0.524287 0.493583 0.972775 0.292517 0.771358 0.526745 0.769914 0.400229
0.891529 0.283315 0.352458 0.807725 0.919026 0.0697553 0.949327 0.525995 0.0
860558 0.192214 0.663227 0.890233 0.348893 0.0641713 0.020023 0.457702 0.063
0958 0.23828 0.970634 0.902208 0.85092 0.266666 0.53976 0.375207 0.760249 0.
512535 0.667724 0.531606 0.0392803 0.437638 0.931835 0.93081 0.720952 0.2842
93 0.738534 0.639979 0.354049 0.687861 0.165974 0.440105 0.880075

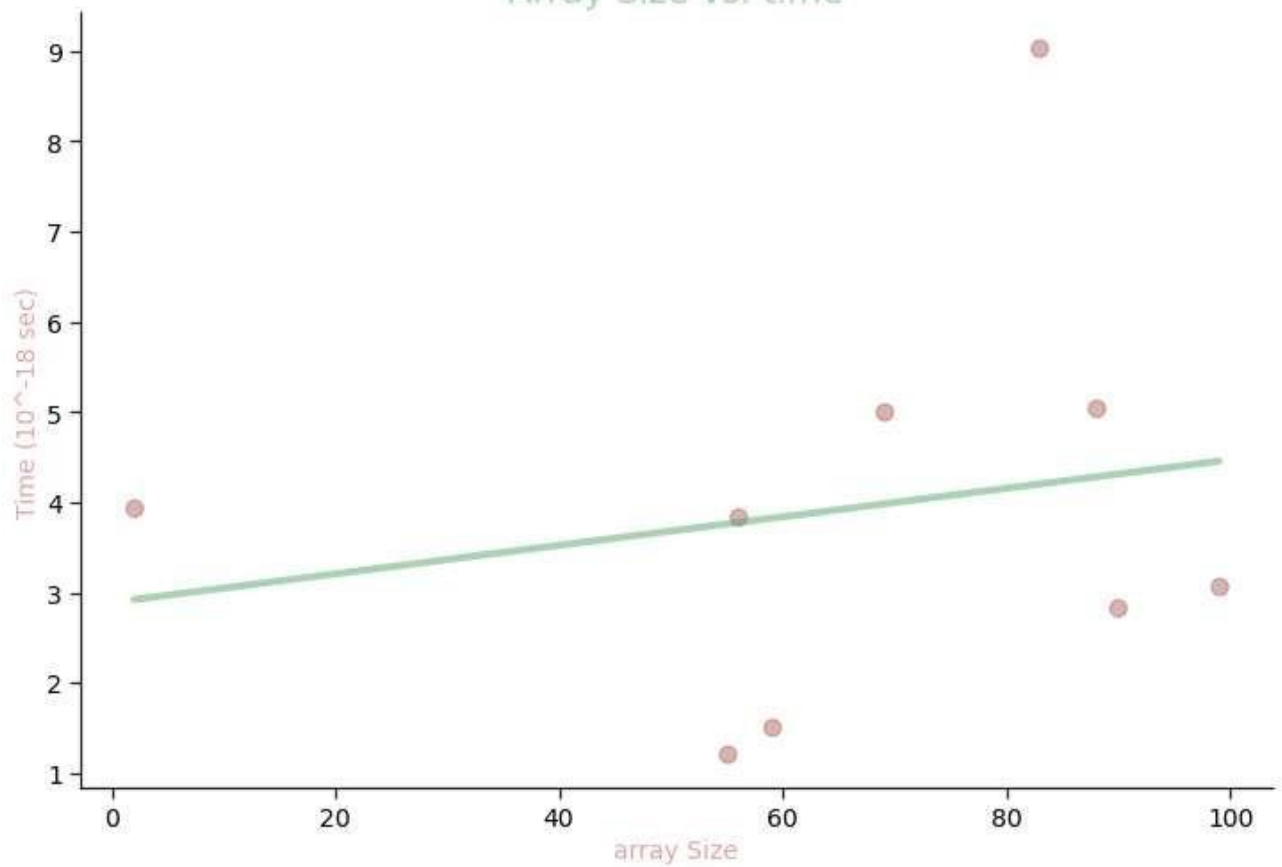
Sorted array: 0.0163006 0.020023 0.0392803 0.0630958 0.0641713 0.0697553 0.0
860558 0.108809 0.12979 0.137232 0.141603 0.156679 0.165974 0.192214 0.19755
1 0.218257 0.23828 0.242887 0.266666 0.277775 0.283315 0.284293 0.292517 0.2
96032 0.335223 0.348893 0.352458 0.354049 0.364784 0.375207 0.394383 0.40022
9 0.400944 0.437638 0.440105 0.457702 0.477397 0.493583 0.512535 0.512932 0.
513401 0.524287 0.525995 0.526745 0.531606 0.53976 0.55397 0.606969 0.61264
0.628871 0.635712 0.637552 0.639979 0.663227 0.667724 0.687861 0.717297 0.72
0952 0.738534 0.760249 0.76823 0.769914 0.771358 0.783099 0.79844 0.804177 0.
807725 0.839112 0.85092 0.880075 0.890233 0.891529 0.902208 0.911647 0.9161
95 0.919026 0.93081 0.931835 0.949327 0.95223 0.970634 0.972775 0.998924

Time difference is: 7.2901e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Array Size vs. time



Bucket Sort

Viva Questions

1. Is the bucket sort in place sort? Why or why not?

Ans.

No, it's not an in-place sorting algorithm. The whole idea is that input sorts themselves as they are moved to the buckets.

2. Is bucket sort a stable sort?

Ans.

Bucket sort is stable, if the underlying sort is also stable, as equal keys are inserted in order to each bucket. Counting sort works by determining how many integers are behind each integer in the input array A. Using this information, the input integer can be directly placed in the output array B.

3. Why bucket sort is good for large size arrays?

Ans.

Yes

The advantage of bucket sort is **that once the elements are distributed into buckets, each bucket can be processed independently of the others.** This means that you often need to sort much smaller arrays as a follow-up step than the original array.

Bucket sort works by distributing the array elements into a number of buckets. So bucket sort is most efficient in the case **when the input is uniformly distributed.**

4. State any disadvantage of using this sort.

Ans.

1. The problem is that if the buckets are distributed incorrectly, you may wind up spending a lot of extra effort for no or very little gain. As a result, bucket

sort works best when the data is more or less evenly distributed, or when there is a smart technique to pick the buckets given a fast set of heuristics based on the input array.

2. Can't apply it to all data types since a suitable bucketing technique is required. Bucket sort's efficiency is dependent on the distribution of the input values, thus it's not worth it if your data are closely grouped. In many situations, you might achieve greater performance by using a specialized sorting algorithm like radix sort, counting sort, or burst sort instead of bucket sort.
3. Bucket sort's performance is determined by the number of buckets used, which may need some additional performance adjustment when compared to other algorithms.

Radix Sort:

Radix Sort is a non-comparative sorting algorithm. It is one of the most efficient and fastest linear sorting algorithms. In radix sort, we first sort the elements based on last digit (least significant digit). Then the result is again sorted by second digit, continue this process for all digits until we reach most significant digit. We use counting sort to sort elements of every digit.

Pseudo code

```

Radix-Sort(A, d)
//It works same as counting sort for d number of passes.
//Each key in A [1...n] is a d-digit integer.
// (Digits are numbered 1 to d from right to left.)
  for j = 1 to d do
    //A[]-- Initial Array to Sort
    intcount[10] = {0};
    //Store the count of "keys" in count[]
    //key- it is number at digit place j
    for i = 0 to n do
      count[key of(A[i]) in pass j]++

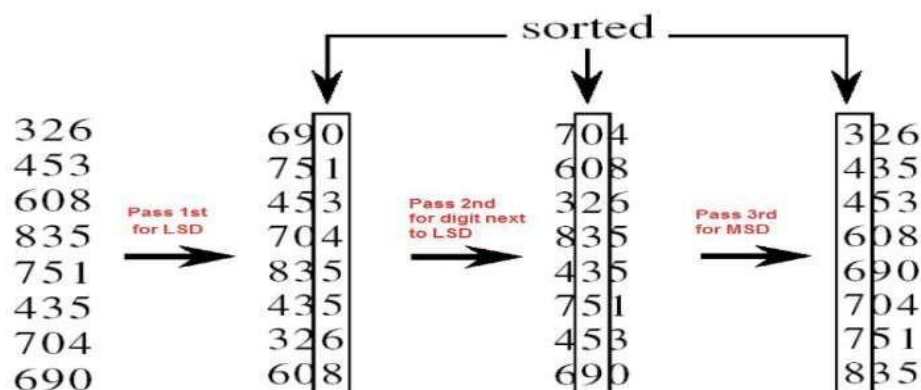
    fork = 1 to 10 do
      count[k] = count[k] + count[k-1]

    //Build the resulting array by checking
    //new position of A[i] from count[k]
    for i = n-1 down to 0 do
      result[ count[key of(A[i])] ] = A[j]
      count[key of(A[i])]--

    //Now main array A[] contains sorted numbers
    //according to current digit place
    for i=0 to n do
      A[i] = result[i]

  end for(j)
end func

```

Example:

In the above example:

For 1st pass: We sort the array on basis of least significant digit (1s place) using counting sort. Notice that 435 is below 835, because 435 occurred below 835 in the original list.

For 2nd pass: We sort the array on basis of next digit (10s place) using counting sort. Notice that here 608 is below 704, because 608 occurred below 704 in the previous list, and similarly for (835, 435) and (751, 453).

For 3rd pass: We sort the array on basis of most significant digit (100s place) using counting sort. Notice that here 435 is below 453, because 435 occurred below 453 in the previous list, and similarly for (608, 690) and (704, 751).

Result and Analysis

In this algorithm running time depends on intermediate sorting algorithm which is counting sort. If the range of digits is from 1 to k , then counting sort time complexity is $O(n+k)$. There are d passes i.e. counting sort is called d time, so total time complexity is $O(nd+nk) = O(nd)$. As $k=O(n)$ and d is constant, so radix sort runs in linear time. It performs the same way for best case as well

| | | | | | | | | |
|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|
| Initial Array | 10 | 21 | 17 | 34 | 44 | 11 | 654 | 123 |
| Sorted based on One's Place | 10 | 21 | 11 | 123 | 34 | 44 | 654 | 17 |
| Sorted based on Ten's Place | 10 | 11 | 17 | 21 | 123 | 34 | 44 | 654 |
| Sorted based on Hundred's Place | 010 | 011 | 017 | 021 | 034 | 044 | 123 | 654 |

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

int getMax(int arr[], int n)    {
    int maxEle = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > maxEle)
            maxEle = arr[i];
    return maxEle;
}

void countSort(int arr[], int n, int exp)    {
    int output[n]; // output array
    int count[10] = { 0 };

    // Store count of occurrences in count[]
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual position of this
    digit in output[]
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[], so that arr[] now contains sorted numbers
    according to current digit
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixsort(int arr[], int n)    {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead of passing digit
    number, exp is passed. exp is 10^i where i is current digit number
```

```
        for (int exp = 1; m / exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }

    void printArray(int arr[], int size)    {
        for (int i = 0; i < size; i++)
            cout << arr[i] << " ";
        cout << endl;
    }

    int main() {
        // int arr[] = {64, 34, 25, 12, 22, 11, 90};
        // int n = sizeof(arr)/sizeof(arr[0]);
        int n = rand() % 100;
        int arr[n];
        for (int i = 0; i < n; i++) {
            arr[i] = rand() % 100;
        }
        cout << "Array: ";
        printArray(arr, n);
        cout << endl;

        auto start = chrono::high_resolution_clock::now();
        // unsync the I/O of C and C++.
        ios_base::sync_with_stdio(false);

        radixsort(arr, n); // sort arr

        auto end = chrono::high_resolution_clock::now();

        cout << "Sorted array: ";
        printArray(arr, n);
        cout << endl;

        double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
        time_taken *= 1e-9;

        cout << "Time difference is: " << time_taken << setprecision(6);
        return 0;
    }
```

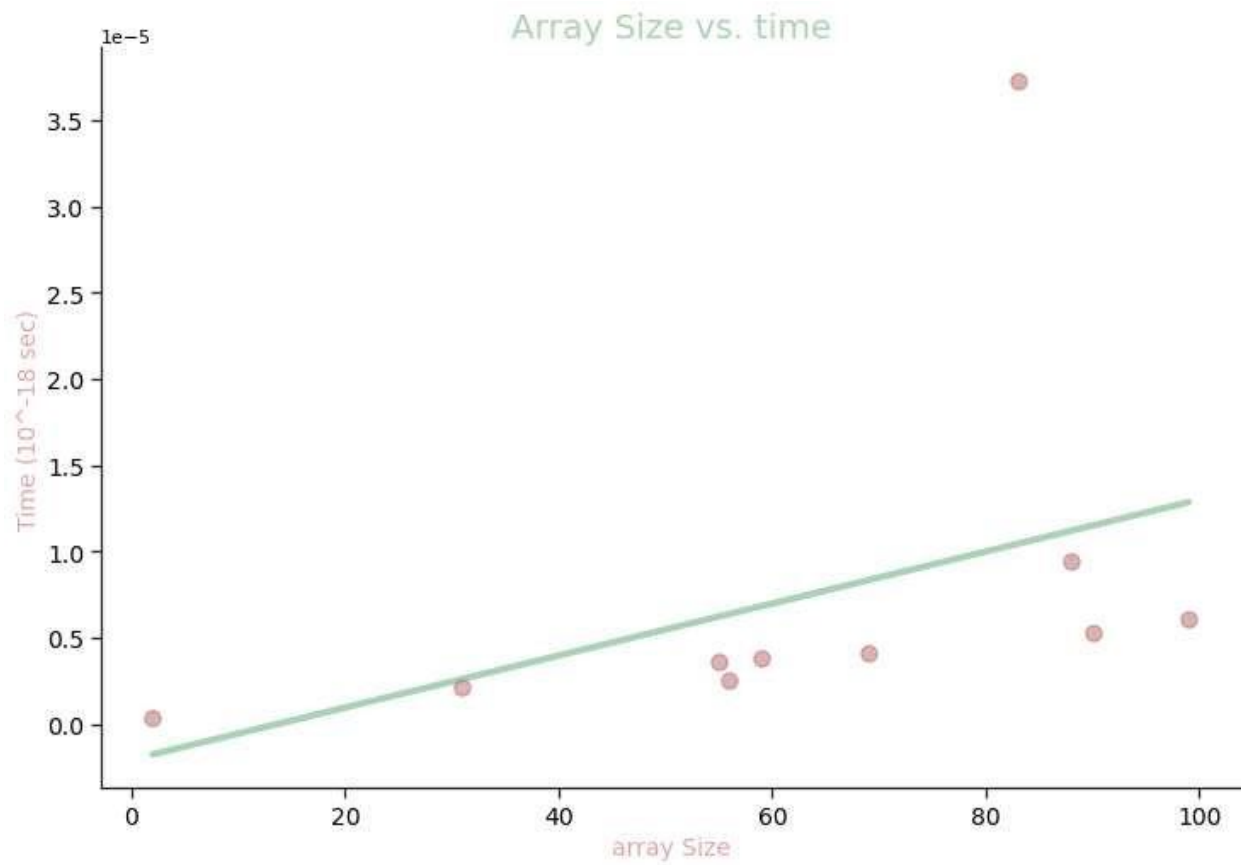
Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98

Time difference is: 3.7338e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Radix Sort

Viva Questions

1. Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

Ans.

If we have $\log_2 n$ bits for every digit, the **running time of Radix appears to be better than Quick Sort** for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively.

Radix sort is slower for (most) real world use cases.

One reason is the complexity of the algorithm:

If items are unique, $k \geq \log(n)$. Even with duplicate items, the set of problems where $k < \log(n)$ is small.

Another is the implementation:

The additional memory requirement (which in it self is a disadvantage), affects cache performance negatively.

2. Is this sort stable?

Ans.

No

MSD sorts are not necessarily stable if the original ordering of duplicate keys must always be maintained. Other than the traversal order, MSD and LSD sorts differ in their handling of variable length input. LSD sorts can group by length, radix sort each group, then concatenate the groups in size order.

3. How much additional space is taken by this sort?

Ans.

$O(k + n)$

Radix sort's **space complexity is bound to the sort it uses to sort each radix**. In best case, that is counting sort. As you can see, counting sort creates multiple arrays, one based on the size of K , and one based on the size of N . B is the output array which is size n .

Shell Sort:

It is a generalized version of insertion sort. It is an in-place comparison sort. It is also known as **diminishing increment sort**; it is one of the oldest sorting algorithms invented by Donald L. Shell (1959.) This algorithm uses insertion sort on the large interval of elements to sort. Then the interval of sorting keeps on decreasing in a sequence until the interval reaches 1. These intervals are known as **gap sequence**.

Increment Sequences:

Shell's original sequence: $N/2, N/4, \dots, 1$ (repeatedly divide by 2);

Hibbard's increments: $1, 3, 7, \dots, 2^k - 1$;

Knuth's increments: $1, 4, 13, \dots, (3^k - 1) / 2$;

Sedgewick's increments: $1, 5, 19, 41, 109, \dots$

Here interval is calculated based on Knuth's formula as –

Knuth's Formula

$h = h * 3 + 1$, where $\rightarrow h$ is interval with initial value 1

Pseudo code

Procedure shellSort ()

A: array of items

/ calculate interval*/*

while interval < A. length /3 do:

*interval = interval * 3 + 1*

end while

while interval > 0 do:

for outer = interval; outer < A. length; outer ++ do:

/ select value to be inserted */*

ValueToInsert = A[outer]

inner = outer;

*/*shift element towards right*/*

while inner > interval -1 && A [inner - interval] >=

valueToInsert do:

```

        A[inner] = A [inner - interval]
        inner = inner - interval
    end while
    /* insert the number at hole position */
    A[inner] = valueToInsert
end for
/* calculate interval*/
interval = (interval -1) /3;
end while
end procedure

```

Example:

| | <i>a</i>₁ | <i>a</i>₂ | <i>a</i>₃ | <i>a</i>₄ | <i>a</i>₅ | <i>a</i>₆ | <i>a</i>₇ | <i>a</i>₈ | <i>a</i>₉ | <i>a</i>₁₀ | <i>a</i>₁₁ | <i>a</i>₁₂ |
|------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|------------------------------|------------------------------|------------------------------|
| Input data | 62 | 83 | 18 | 53 | 07 | 17 | 95 | 86 | 47 | 69 | 25 | 28 |
| After 5-sorting | 17 | 28 | 18 | 47 | 07 | 25 | 83 | 86 | 53 | 69 | 62 | 95 |
| After 3-sorting | 17 | 07 | 18 | 47 | 28 | 25 | 69 | 62 | 53 | 83 | 86 | 95 |
| After 1-sorting | 07 | 17 | 18 | 25 | 28 | 47 | 53 | 62 | 69 | 83 | 86 | 95 |

The first pass, 5-sorting, performs insertion sort on five separate sub arrays (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). For instance, it changes the sub array (a_1, a_6, a_{11}) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on the three sub arrays (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}). The last pass, 1-sorting, is an ordinary insertion sort of the entire array ($a_1 \dots a_{12}$).

Result and Analysis

Since in this algorithm insertion sort is applied in the large interval of elements and then interval reduces in a sequence, therefore the running time of Shell sort is heavily dependent on the gap sequence it uses. So in worst case time complexity is $O(n^2)$ and in Average Case time complexity depends on gap sequence while in Best Case Time complexity is $O(n \log n)$.

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

int shellSort(int arr[], int n) {
    for (int gap = n/2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i += 1) {
            int temp = arr[i];

            // shift gap sorted elements to find position for ith element
            int j = i;
            for (j; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp; // place i th element in correct position

        }
    }
    return 0;
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
```

```
// unsync the I/O of C and C++.
ios_base::sync_with_stdio(false);

shellSort(arr, n); // sort arr

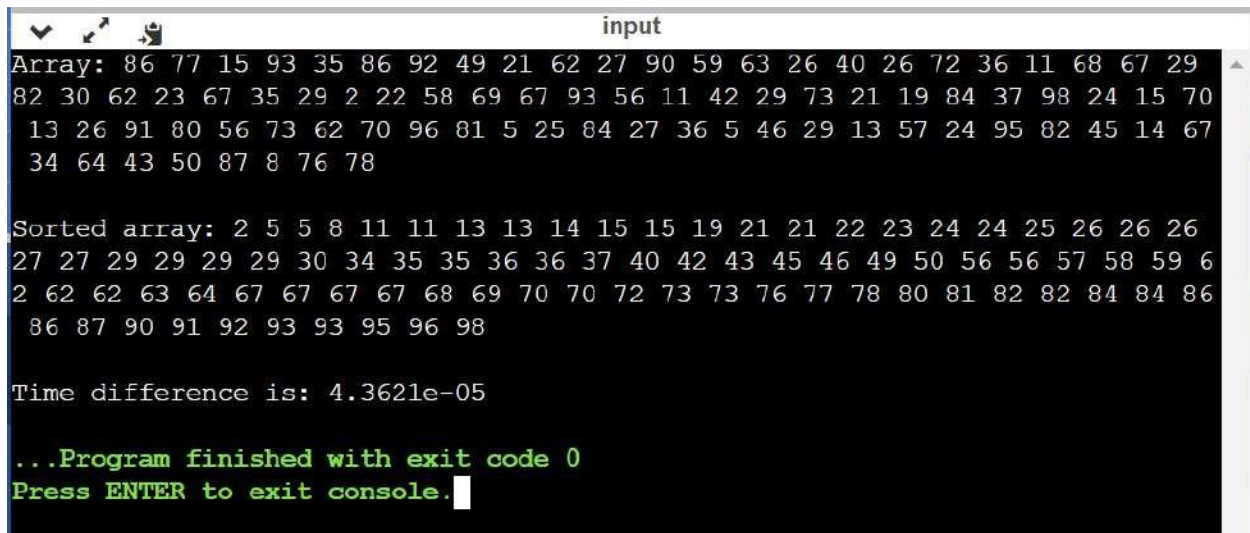
auto end = chrono::high_resolution_clock::now();

cout << "Sorted array: ";
printArray(arr, n);
cout << endl;

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;

cout << "Time difference is: " << time_taken << setprecision(6);
return 0;
}
```

Output:

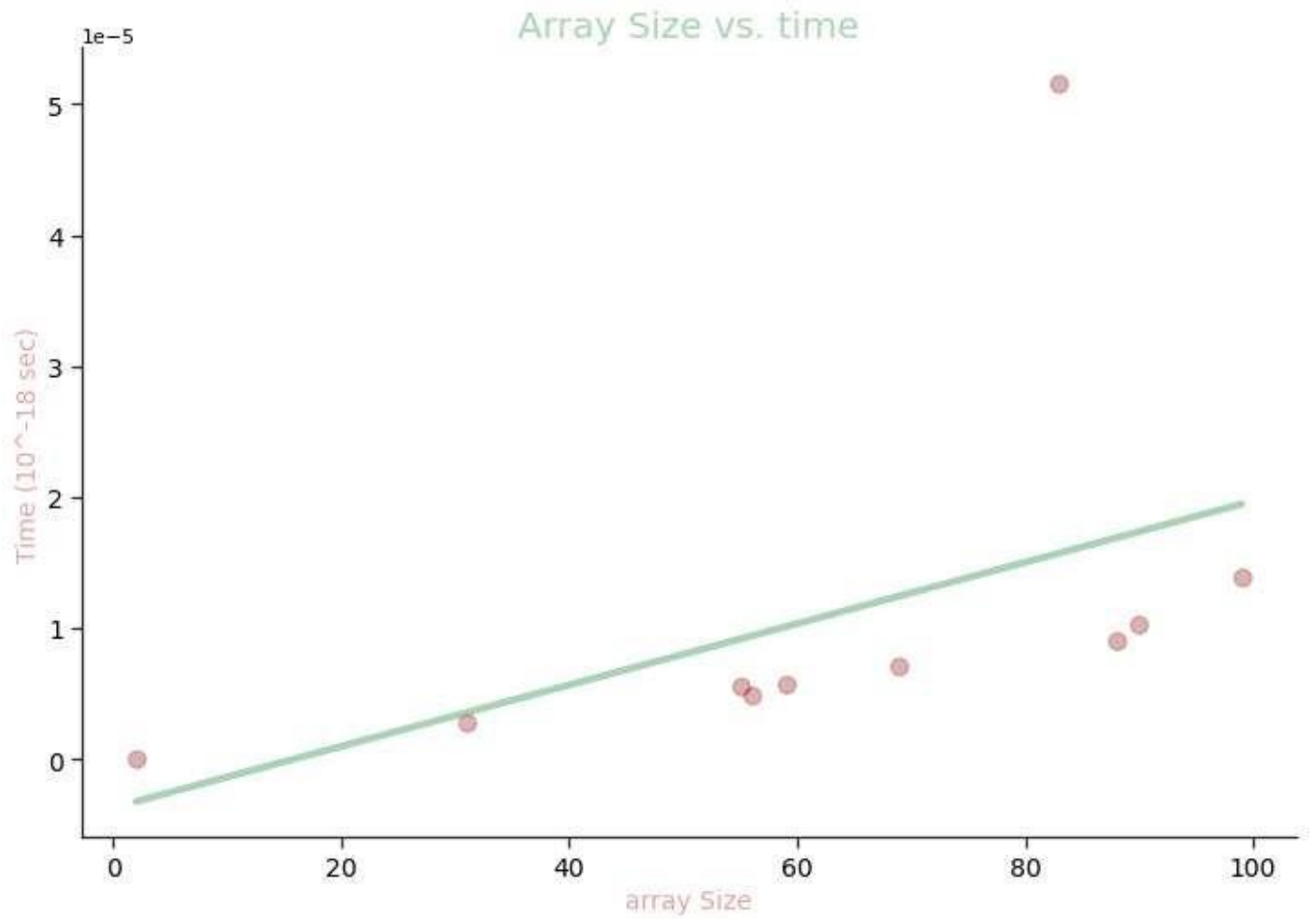


```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98

Time difference is: 4.3621e-05

...Program finished with exit code 0
Press ENTER to exit console.
```



Shell Sort

Viva Questions

1. How can the time complexity of shell sort be improved?

Ans.

Shell Sort improves its time complexity by taking the advantage of the fact that using Insertion Sort on a partially sorted array results in less number of moves.

2. State any application of shell sort?

Ans.

- Shellsort performs more operations and has higher cache miss ratio than quicksort.
- However, since it can be implemented using little code and does not use the call stack, some implementations of the qsort function in the C standard library targeted at embedded systems use it instead of quicksort. Shellsort is, for example, used in the uClibc library. For similar reasons, an implementation of Shellsort is present in the Linux kernel.
- Shellsort can also serve as a sub-algorithm of introspective sort, to sort short subarrays and to prevent a slowdown when the recursion depth exceeds a given limit. This principle is employed, for instance, in the bzip2 compressor.

3. Given the following list of numbers: [5,16,20,12,3,8,9,17,19,7]. What is the content of the list after all swapping is complete for a gap size of 3?

Ans.

[5, 3, 8, 7, 16, 19, 9, 17, 20, 12]

Selection Sort:

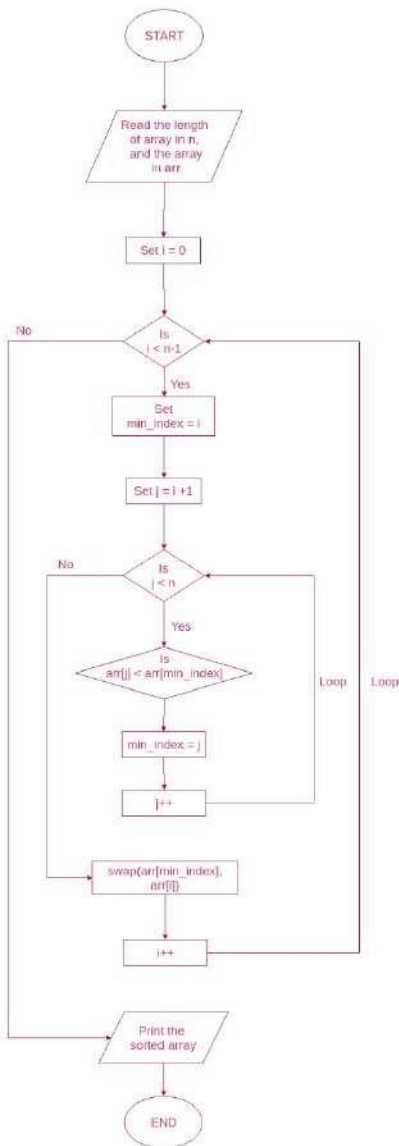
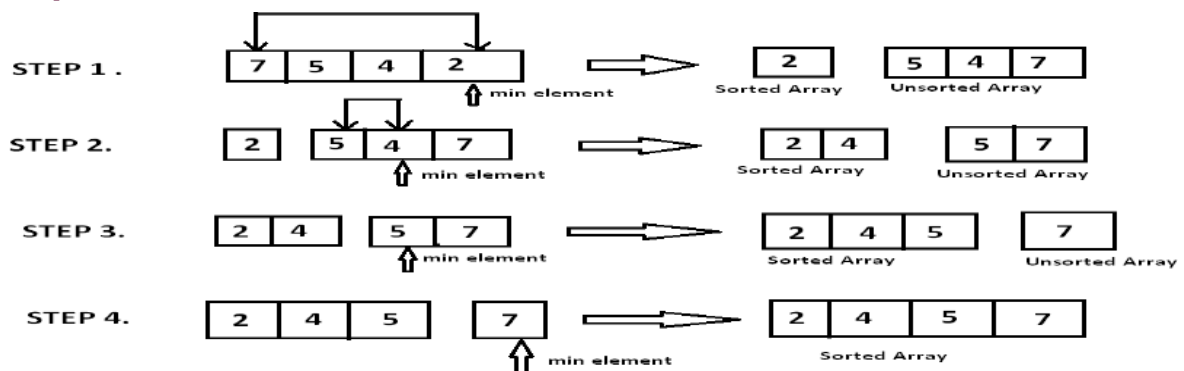
Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

Pseudo code

```
Procedure selection sort
list: array of items
n: size of list
  for i = 1 to n - 1
    /* set current element as minimum*/
    min = i
    /* check the element to be minimum */
    for j = i+1 to n
      if list[j] < list [min] then
min = j;
      end if
    end for
    /* swap the minimum element with the current element*/
    if indexMin != i then
      swap list[min] and list[i]
    end if
  end for
end procedure:
```


Example:Flowchart for Selection Sort

Result and Analysis

To find the minimum element from the array of N elements, $N-1$ comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to $N-1$ and then $N-2$ comparisons are required to find the minimum in the unsorted array. Therefore $(N-1) + (N-2) + \dots + 1 = (N(N-1))/2$ comparisons and N swaps result in the overall complexity of $O(N^2)$

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// keep finding minimum element and place it in beginning
void selectionSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]); // swap min with first
    }
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
```

```
}

int main()
{
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();

    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    selectionSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98

Time difference is: 4.2556e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

Batch Analysis:

Source Code:

```
#include <bits/stdc++.h>

#include <chrono>

using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// keep finding minimum element and place it in beginning
void selectionSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        swap(&arr[min_idx], &arr[i]); // swap min with first
    }
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
```

```
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";
    cout << endl;
}

double sortApp(int n)
{
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    selectionSort(arr, n);
```

```
    auto end = chrono::high_resolution_clock::now();

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}

int main()
{
    double times[10];
    int ns[10];
    for (int x = 0; x < 10; x++)
    {
        int n = rand() % 100;
        ns[x] = n;
        times[x] = sortApp(n);
    }
    cout << "value of n's: " << endl;
    printArray(ns, 10);
    cout << "time for each n: " << endl;
    printArray(times, 10);
}
```

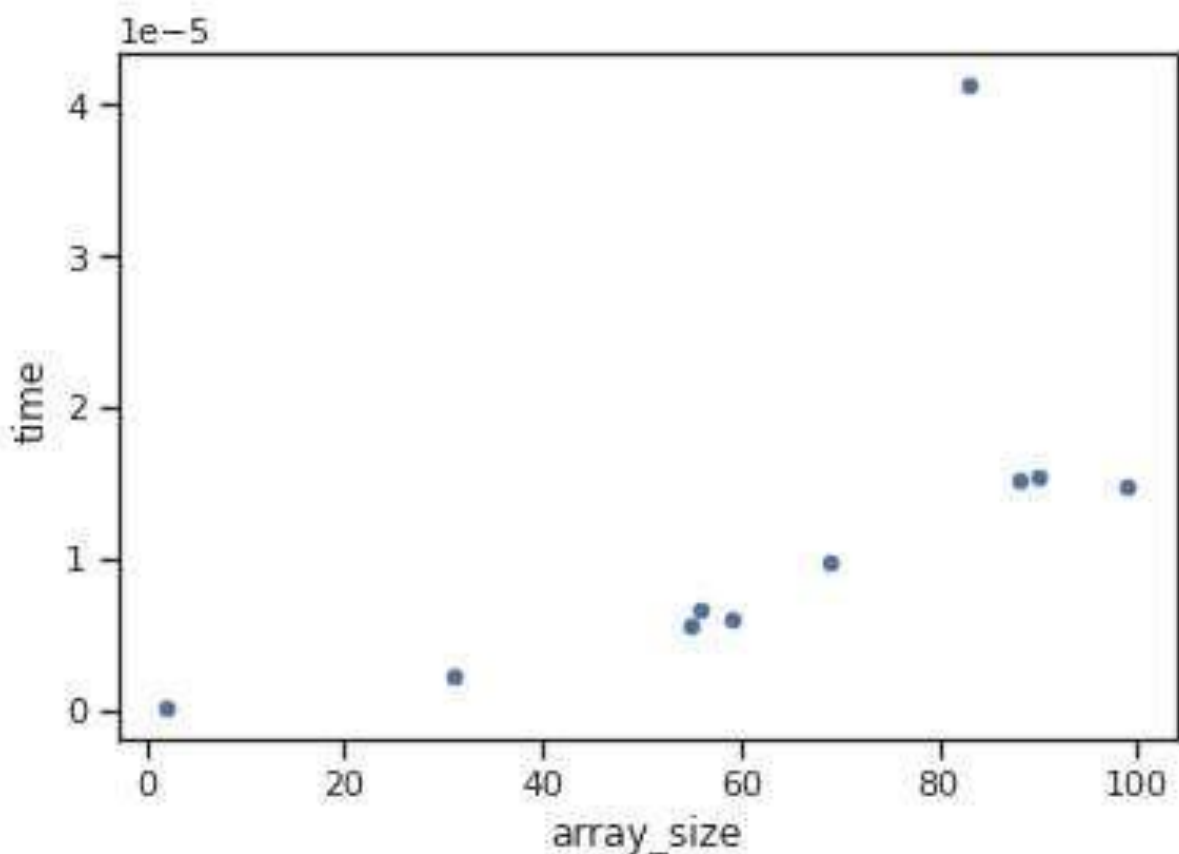
Output:

```
input
value of n's:
83, 88, 56, 69, 90, 59, 2, 99, 55, 31,
time for each n:
4.4366e-05, 1.6036e-05, 6.706e-06, 9.483e-06, 1.5256e-05, 7.488e-06, 8.3e-08,
1.8461e-05, 6.788e-06, 2.525e-06,

...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [4.1399e-05, 1.5271e-05, 6.697e-06, 9.652e-06, 1.537e-05, 6.069e-06, 4.2e-08, 1.4875e-05, 5.673e-06, 2.172e-06]



Viva Questions

1. What is the average case performance of Selection Sort?

Ans.

$\Theta(N^2)$

Average Case Time Complexity is: **$O(N^2)$**

2. For each i from 1 to $n-1$, how many comparisons are there in this sort?

Ans.

Selection sort finds the smallest element and swaps it into the first position. Then, from the remaining $N-1$ elements after the first one (which we know is the smallest), it finds the next smallest and swaps it into the 2nd position. And so on.

To find the smallest element from a group of N elements takes $N-1$ comparisons. Basically, compare item 1 to 2, remember the smallest. Compare item 3 to smallest and remember which ever is smaller, and so on. You end up doing 1 comparison for every element from 2 through N .

We have to find the smallest element $N-1$ times*, but each time we have a smaller list to look through.

For N elements it takes $1+2+3+\dots+N-1 = (N-1)N/2$ comparisons.

3. If all the input elements are identical then how it will affect the time taken by this sort?

Ans.

Selection sort will run in **$O(n^2)$** regardless of whether the array is already sorted or not.

4. What is the output of selection sort after the 2nd iteration given the following sequence of numbers: 16 3 46 9 28 14

Ans.

3 9 46 16 28 14

5. What is straight selection sort?

Ans.

Selection sorting refers to a class of algorithms for sorting a list of items using comparisons. These algorithms select successively smaller or larger items from the list and add them to the output sequence. This is an improvement of the Simple Selection Sort and it is called Straight Selection Sort. Therefore, instead of replacing the selected element by a unique value in the i-th pass (as happens in Simple Selection Sort), the selected element is exchanged with the i-th element.

Heap Sort:

This algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

What is a heap?

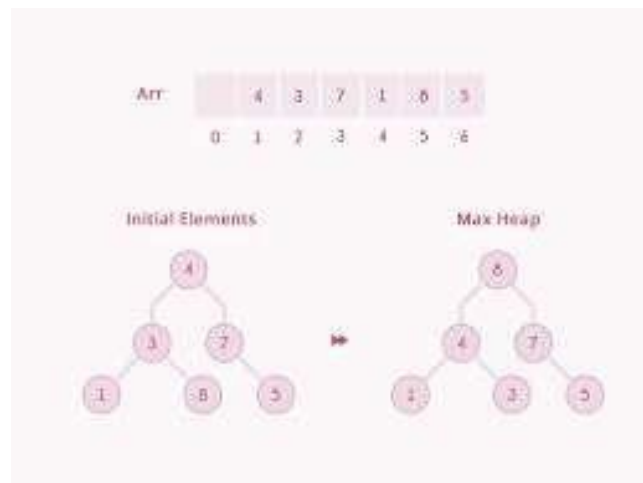
Heap is a special tree-based data structure, which satisfies the following special heap properties:

Shape Property: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

Heap Property: All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.



Pseudo code

```

Heapsort (A) {
    BuildHeap(A)
    for i <- length (A) downto 2 {
        exchange A [1] <-> A[i]
        heapsize <- heapsize -1
        Heapify(A, 1)
    }

    BuildHeap (A) {
        heapsize <- length (A)
        for i <- floor (length/2) downto 1
        Heapify (A, i)
    }

    Heapify (A, i) {
        le <- left (i)
        ri <- right(i)
        if (le<=heapsize) and (A[le]>A[i])

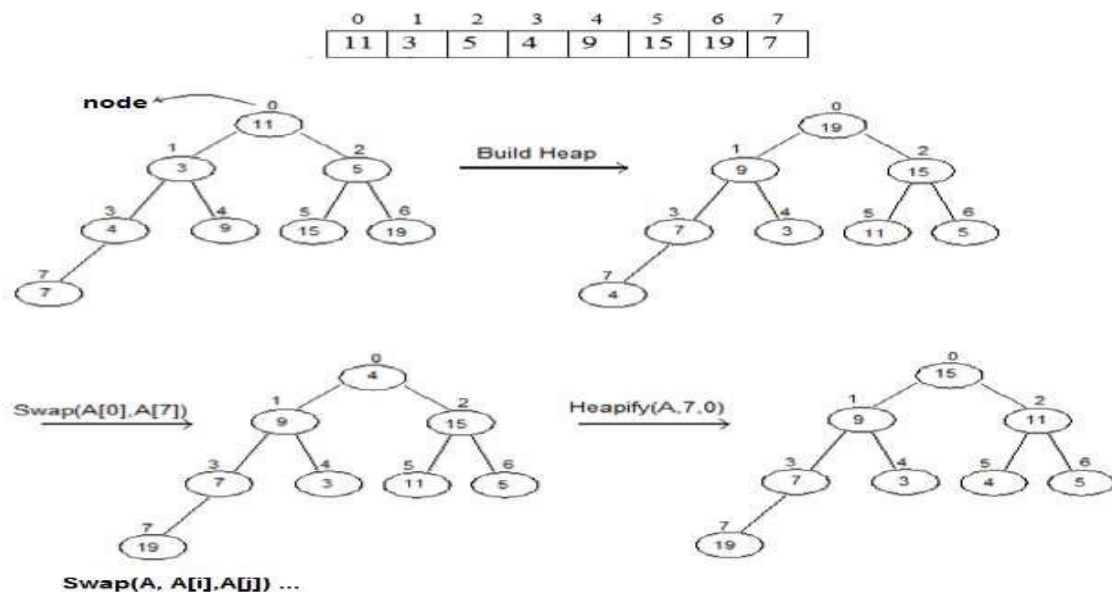
```

```

    largest <- le
else
    largest <- i
    if (ri <= heapsize) and (A[ri] > A[largest])
        largest <- ri
    if (largest != i) {
        exchange A[i] <-> A[largest]
        Heapify(A, largest)
    }
}

```

Example:



Result and Analysis

Heap sort has the best possible worst case running time complexity of $O(n \log n)$. It doesn't need any extra storage and that makes it good for situations where array size is large. Heapify runs in time $O(h)$ and there are at most $n = 2^{h+1} - 1$ nodes in an almost complete binary tree of height h , so Heapify runs in time $O(\log n)$. Build-Heap calls Heapify $n/2$ times, so it takes $O(n \log n) = O(n \ln n)$.

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

void heapify(int arr[], int n, int root)
{
    int largest = root;
    int leftChild = 2 * root + 1;
    int rightChild = 2 * root + 2;

    if (leftChild < n && arr[leftChild] > arr[largest])
        largest = leftChild;

    if (rightChild < n && arr[rightChild] > arr[largest])
        largest = rightChild;

    if (largest != root)
    {
        swap(arr[root], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    // Build heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--)
    {
        // extract elements from heap
        swap(arr[0], arr[i]); // Move current root to end
        heapify(arr, i, 0);   // call max heapify on the reduced heap
    }
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";

    cout << endl;
}
```

```
}

int main()
{
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();

    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    heapSort(arr, n); // sort arr

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);

    return 0;
}
```

Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78

Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98

Time difference is: 5.123e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [4.9828e-05, 1.5332e-05, 8.655e-06, 1.0828e-05, 1.4977e-05, 8.714e-06, 1.48e-07, 1.7703e-05, 8.224e-06, 3.716e-06]

Viva Questions

1. What are the minimum and maximum numbers of elements in a heap of height h ?

Ans.

2^h

A heap of height h is complete up to the level at depth $h-1$ and needs to have at least one node on level h .

Therefore the minimum total number of nodes must be at least $\sum_{i=0}^{h-1} 2^i + 1 = 2^h - 1 + 1 = 2^h$. $\sum_{i=0}^{h-1} 2^i + 1 = 2^h - 1 + 1 = 2^h$, and this tight since a heap with 2^h nodes has height h .

2. Where in a heap might the smallest element reside?

Ans.

Smallest element will be at the last level of the max heap.

3. Is an array that is in reverse sorted order a heap?

Ans.

The correct version of this statement is "An array sorted in ascending order is can be treated **as min- heap**" and its complementary statement is "An array sorted in descending order can be treated as max heap"

4. Does heap sort uses extra space for storage?

Ans.

No, The Heap sort algorithm can be implemented as an in-place sorting algorithm. This means that its memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work.

Insertion Sort:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

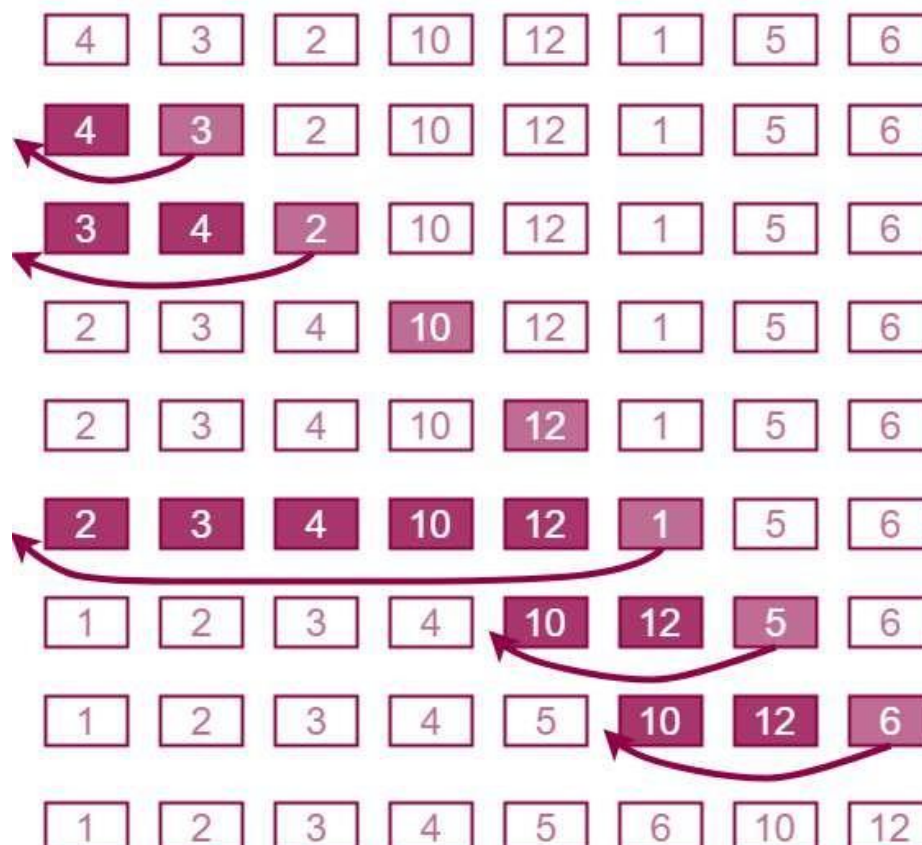
Algorithm

To sort an array of size n in ascending order:

- 1: Iterate from $\text{arr}[1]$ to $\text{arr}[n]$ over the array.
- 2: Compare the current element (key) to its predecessor.
- 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Example:

Insertion Sort Execution Example



Result and Analysis

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Source Code:

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;

// place current element in right order as we move forward
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int key = arr[i];
        int j = i - 1;
        // compare to all predecessors
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int size)
```

```
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    // int arr[] = {64, 34, 25, 12, 22, 11, 90};
    // int n = sizeof(arr)/sizeof(arr[0]);
    int n = rand() % 100;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    cout << "Array: ";
    printArray(arr, n);
    cout << endl;

    auto start = chrono::high_resolution_clock::now();
    // unsync the I/O of C and C++.
    ios_base::sync_with_stdio(false);

    insertionSort(arr, n);

    auto end = chrono::high_resolution_clock::now();

    cout << "Sorted array: ";
    printArray(arr, n);
    cout << endl;

    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    cout << "Time difference is: " << time_taken << setprecision(6);
    return 0;
}
```

Output:

```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78

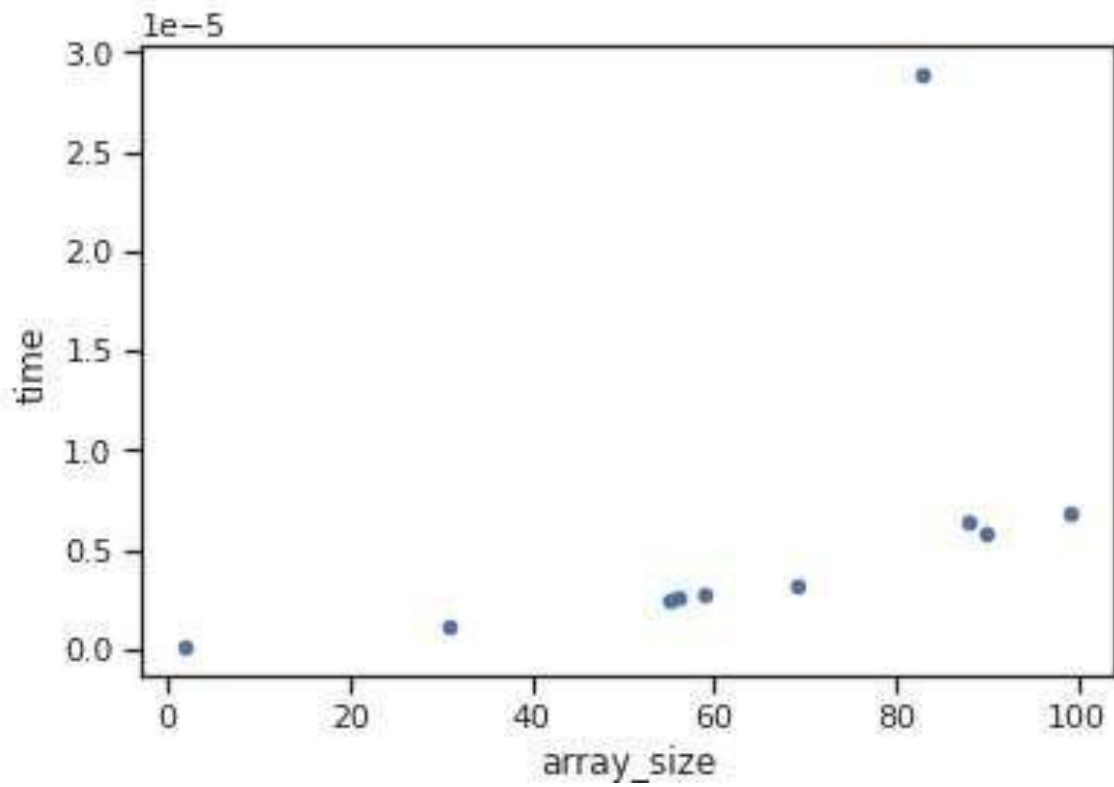
Sorted array: 2 5 5 8 11 11 13 13 14 15 15 19 21 21 22 23 24 24 25 26 26 26
27 27 29 29 29 29 30 34 35 35 36 36 37 40 42 43 45 46 49 50 56 56 57 58 59 6
2 62 62 63 64 67 67 67 67 68 69 70 70 72 73 73 76 77 78 80 81 82 82 84 84 86
86 87 90 91 92 93 93 95 96 98

Time difference is: 5.155e-05

...Program finished with exit code 0
Press ENTER to exit console.
```

arraySize = [83, 88, 56, 69, 90, 59, 2, 99, 55, 31]

time = [2.8917e-05, 6.425e-06, 2.608e-06, 3.153e-06, 5.76e-06, 2.767e-06, 5.5e-08, 6.81e-06, 2.451e-06, 1.186e-06]



Viva Questions

1. Why is time complexity of insertion sort?

Ans.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n .

Wherein for an unsorted array, it takes for an element to compare with all the other elements which mean every n element compared with all other n elements. Thus, making it for $n \times n$, i.e., n^2 comparisons. One can also take a look at other sorting algorithms such as *Merge sort*, *Quick Sort*, *Selection Sort*, etc. and understand their complexities.

Worst Case Time Complexity [Big-O]: $O(n^2)$

Best Case Time Complexity [Big-omega]: $O(n)$

Average Time Complexity [Big-theta]: $O(n^2)$

2. Is it possible to do insertion sort in place?

Ans.

Yes, It is. Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

3. Is insertion sort the worst?

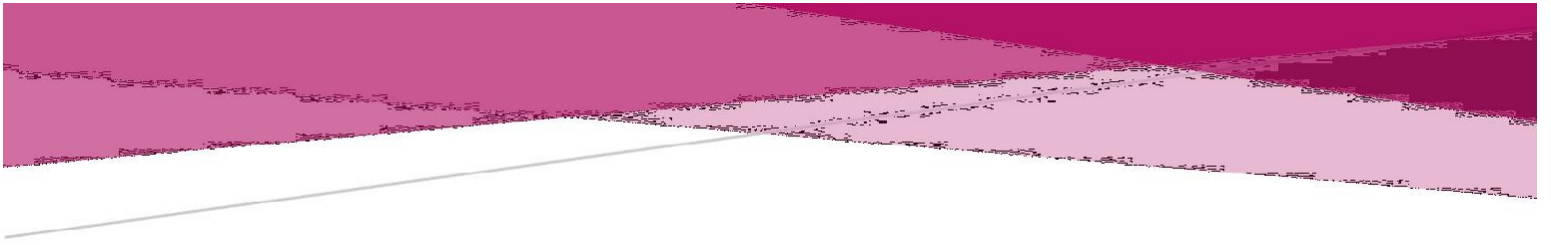
Ans.

Insertion sort has a fast best-case running time and is a good sorting algorithm to use if the input list is already mostly sorted. For larger or more unordered lists, an algorithm with a **faster worst** and average-case running time, such as mergesort, would be a better choice. Now iterate through this array just one time. The time

4. What is Binary Insertion Sort?

Ans.

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes $O(i)$ (at i th iteration) in worst case. We can reduce it to $O(\log i)$ by using binary search. The algorithm, as a whole, still has a running worst case running time of $O(n^2)$ because of the series of swaps required for each insertion.



EXPERIMENT - 2

Algorithms Design and Analysis Lab

Aim

To implement Linear search and Binary search and analyse its time complexity.

Syeda Reeha Quasar

14114802719

4C7

EXPERIMENT – 2

Aim:

To implement Linear search and Binary search and analyse its time complexity.

Theory:

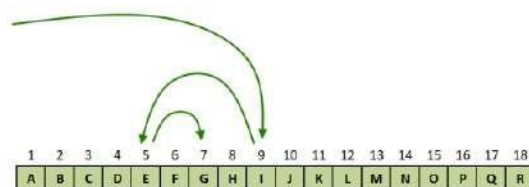
The search operation can be done in the following two ways:

Linear search:

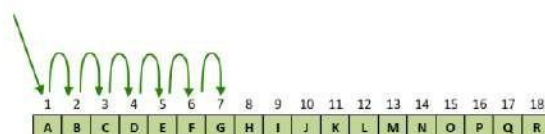
It's a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Binary search:

This search algorithm works on the principle of divide and conquers. For this algorithm, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.



Binary Search - Find 'G' in sorted list A-R



Linear Search - Find 'G' in sorted list A-R

Pseudo code for linear search:

```
Procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

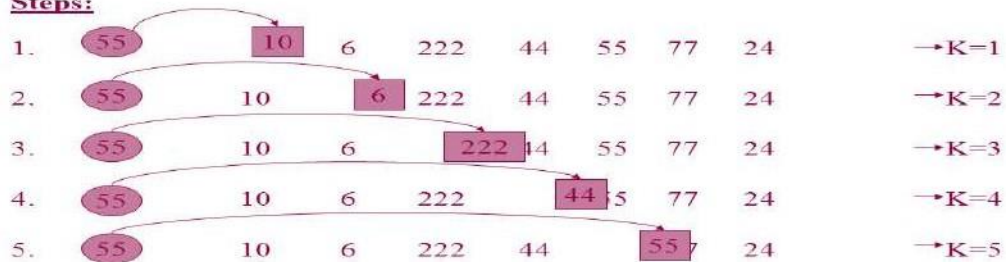
Pseudo code for Binary search:

```
BinarySearch (A [0...N-1], value)
{
  low = 0
  high = N - 1
  while (low <= high) {
    // invariants: value > A[i] for all i < low
    value < A[i] for all i > high
    mid = (low + high) / 2
    if (A[mid] > value)
      high = mid - 1
    else if (A[mid] < value)
      low = mid + 1
    else
      return mid
  }
  return not_found // value would be inserted at index "low"
}
```

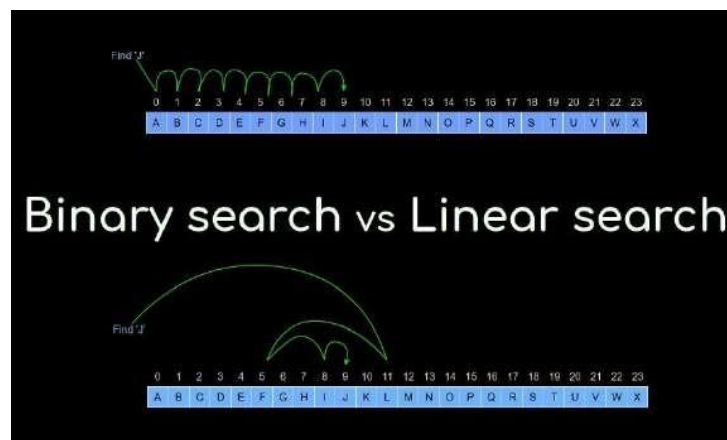
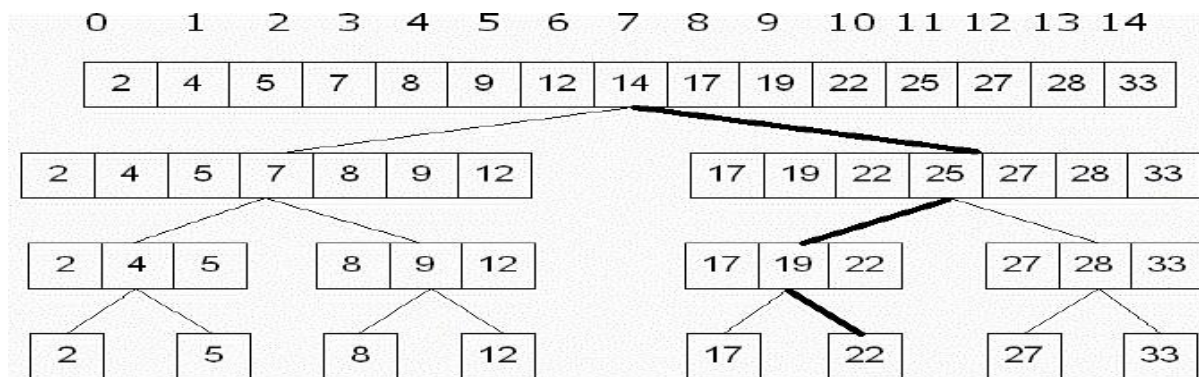
Sample Example:**Example of Linear Search**

List : 10, 6, 222, 44, 55, 77, 24

Key: 55

Steps:**So, Item 55 is in position 5.****Example of Binary Search:**

A sorted array is taken as input. The mid value is found out recursively.



Result and Analysis

Input data needs to be sorted in Binary Search and not in Linear Search and it does the sequential access whereas Binary search access data randomly. So, time complexity of linear search is $O(n)$ while Binary search has time complexity $O(\log n)$ as search is done to either half of the given list.

| Base of comparison | Linear search | Binary search |
|--------------------------|----------------------|-------------------------------|
| Time complexity | $O(N)$ | $O(\log_2 N)$ |
| Best case time | $O(1)$ first element | $O(1)$ center element |
| Prerequisite of an array | No prerequisite | Array must be sorted in order |
| Input data | No need to be sorted | Need to be sorted |
| Access | Sequential | random |

| LINEAR SEARCH | BINARY SEARCH |
|--|--|
| An algorithm to find an element in a list by sequentially checking the elements of the list until finding the matching element | An algorithm that finds the position of a target value within a sorted array |
| Also called sequential search | Also called half-interval search and logarithmic search |
| Time complexity is $O(N)$ | Time complexity is $O(\log_2 N)$ |
| Best case is to find the element in the first position | Best case is to find the element in the middle position |
| It is not required to sort the array before searching the element | It is necessary to sort the array before searching the element |
| Less efficient | More efficient |
| Less complex | More complex |

Linear search:

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

int linearSearch(int arr[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{
    // int arr[] = { 2, 3, 4, 10, 40 };
    // int n = sizeof(arr) / sizeof(arr[0]);

    int n = rand() % 100;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }

    cout << "Array: ";
    printArray(arr, n);

    int x;

    cout << "Enter element you want to search in the array: ";
```

```
cin >> x;

auto start = chrono::high_resolution_clock::now();
// unsync the I/O of C and C++.
ios_base::sync_with_stdio(false);

int result = linearSearch(arr, n, x);

auto end = chrono::high_resolution_clock::now();

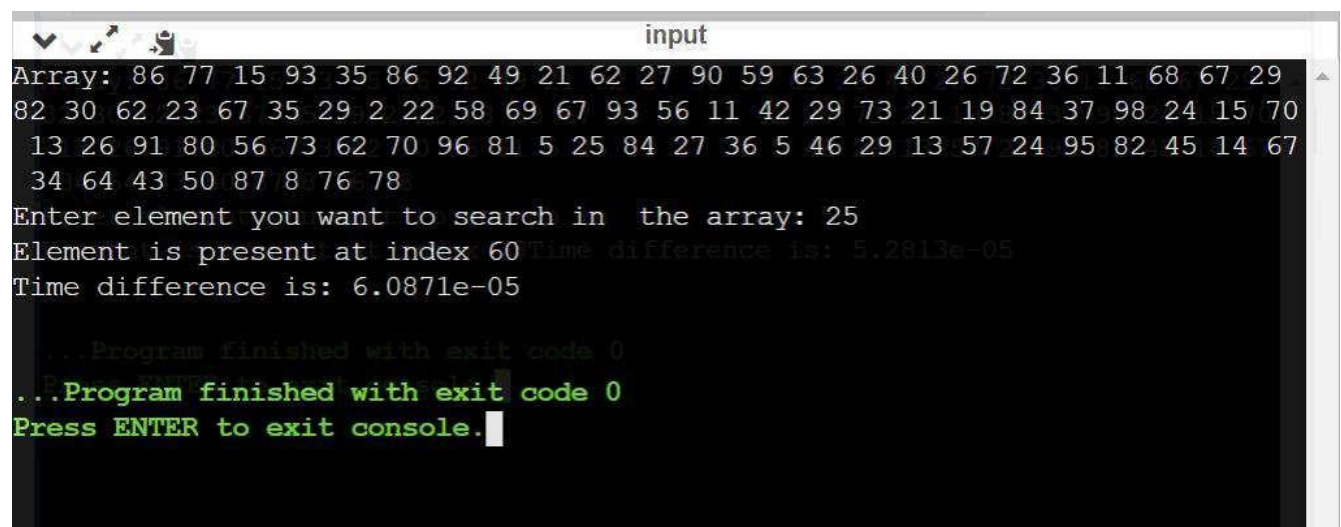
(result == -1)
    ? cout << "Element is not present in array"
    : cout << "Element is present at index " << result;

double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
time_taken *= 1e-9;

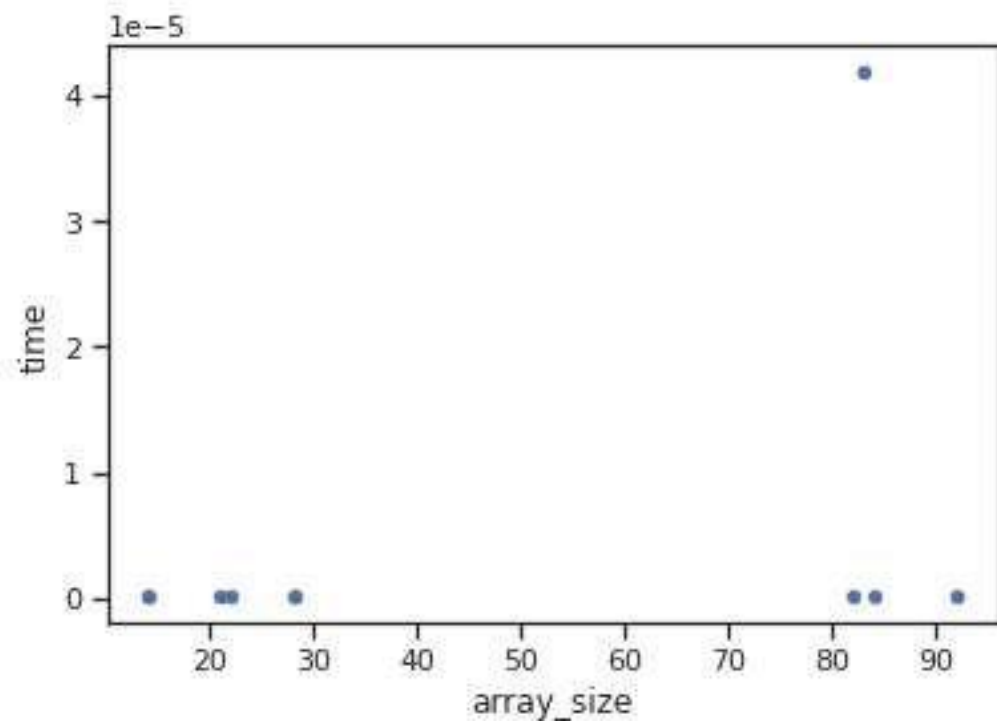
cout << "Time difference is: " << time_taken << setprecision(6) << endl;

return 0;
}
```

Output:



```
input
Array: 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 11 68 67 29
82 30 62 23 67 35 29 2 22 58 69 67 93 56 11 42 29 73 21 19 84 37 98 24 15 70
13 26 91 80 56 73 62 70 96 81 5 25 84 27 36 5 46 29 13 57 24 95 82 45 14 67
34 64 43 50 87 8 76 78
Enter element you want to search in the array: 25
Element is present at index 60 Time difference is: 5.2813e-05
Time difference is: 6.0871e-05
...Program finished with exit code 0
...Program finished with exit code 0
Press ENTER to exit console.
```



Source Code:

```
#include <bits/stdc++.h>

using namespace std;

int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l) / 2;

        if (arr[mid] == x)
            return mid;

        else if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}

int binarySearchIter(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
```

```
        return m;

    if (arr[m] < x)
        l = m + 1;

    else
        r = m - 1;
}
return -1;
}

void printArray(double arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << ", ";

    cout << endl;
}

void printArray(float arr[], int size)
{
    for (int i = 0; i < size; i++)
```

```
        cout << arr[i] << ", ";

    cout << endl;
}

double search(int n)
{
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 100;
    }
    sort(arr, arr + n);

    int x = rand() % 100;

    auto start = chrono::high_resolution_clock::now();
    ios_base::sync_with_stdio(false);

    binarySearch(arr, 0, n, x);

    auto end = chrono::high_resolution_clock::now();

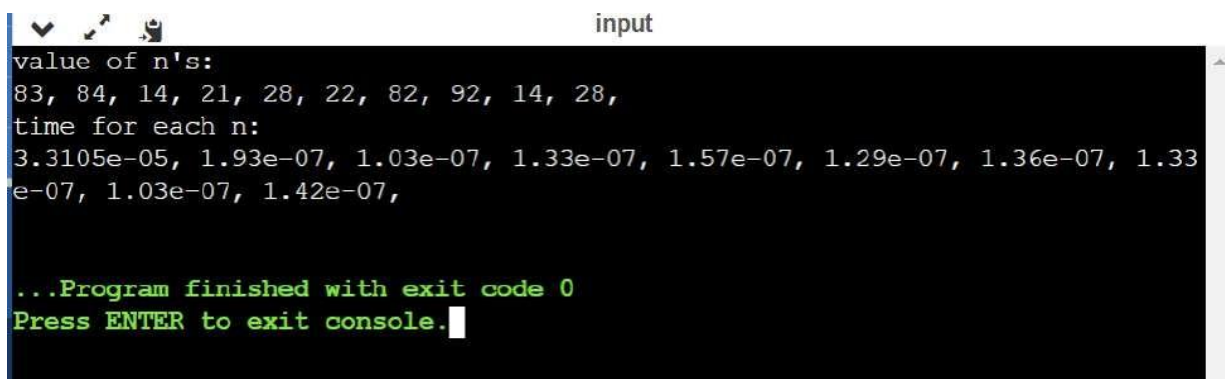
    double time_taken = chrono::duration_cast<chrono::nanoseconds>(end -
start).count();
    time_taken *= 1e-9;

    return time_taken;
}

int main()
```

```
{  
    double times[10];  
    int ns[10];  
  
    for (int x = 0; x < 10; x++)  
    {  
        int n = rand() % 100;  
        ns[x] = n;  
        times[x] = search(n);  
    }  
  
    cout << "value of n's: " << endl;  
    printArray(ns, 10);  
  
    cout << "time for each n: " << endl;  
    printArray(times, 10);  
}
```

Output:

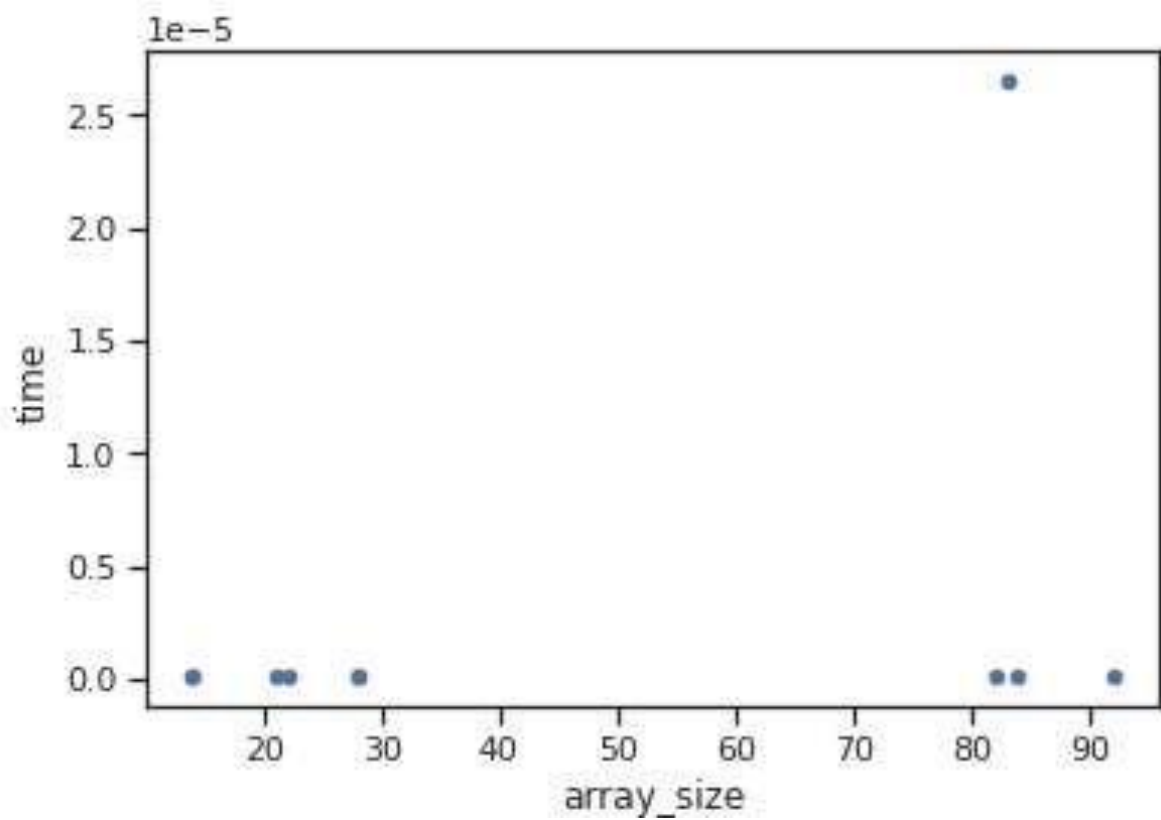


The screenshot shows a console window titled "input" with a dark background. The output of the program is displayed in white text. It shows the values of the array 'ns' and the corresponding search times for each element. The search times are in scientific notation. At the bottom, a green message indicates the program finished with exit code 0, and a prompt asks to press ENTER to exit the console.

```
value of n's:  
83, 84, 14, 21, 28, 22, 82, 92, 14, 28,  
time for each n:  
3.3105e-05, 1.93e-07, 1.03e-07, 1.33e-07, 1.57e-07, 1.29e-07, 1.36e-07, 1.33  
e-07, 1.03e-07, 1.42e-07,  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

```
arraySize = [83, 84, 14, 21, 28, 22, 82, 92, 14, 28]
```

```
time = [2.6536e-05, 1.6e-07, 7.9e-08, 9.3e-08, 1.31e-07, 8.7e-08, 1e-07,  
1.15e-07, 8.9e-08, 1.06e-07]
```



Viva Questions

1. The sequential search, also known as _____?

Ans.

Linear Search

One of the most straightforward and elementary searches is the sequential search, also known as **a linear search**.

2. What is the primary requirement for implementation of binary search in an array?

Ans.

The one pre-requisite of binary search is that an array should be in sorted order, whereas the linear search works on both sorted and unsorted array. The binary search algorithm is based on the divide and conquer technique, which means that it will divide the array recursively.

3. Is binary search applicable on array and linked list both?

Ans.

Yes, Binary search is possible on the linked list if the list is ordered and you know the count of elements in list. But While sorting the list, you can access a single element at a time through a pointer to that node i.e. either a previous node or next node.

4. What is the principle of working of Binary search?

Ans.

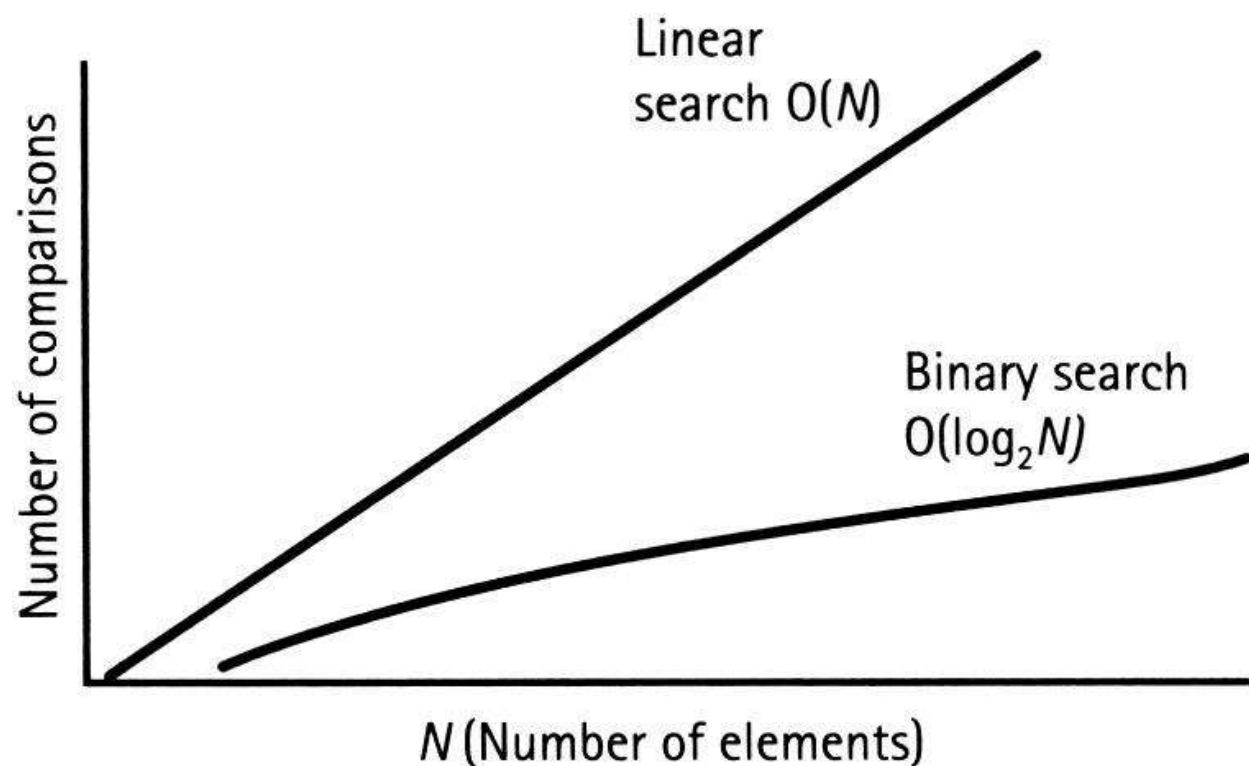
Divide and Conquer

Binary search is a fast search algorithm with run-time complexity of $O(\log n)$. This search algorithm works on the principle of **divide and conquer**. For this algorithm to work properly, the data collection should be in the sorted form.

5. Which searching algorithm is efficient one?

Ans.

Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.



6. Under what circumstances, binary search cannot be applied to a list of elements?

Ans.

In case the list of elements is not sorted, there's no way to use binary search because the median value of the list can be anywhere and when the list is split into two parts, the element that you were searching for could be cut off.

The main problem that binary search takes **$O(n)$ time in Linked List** due to fact that in linked list we are not able to do indexing which led traversing of each element in Linked list take $O(n)$ time. In this paper a method is implemented through which binary search can be done with time complexity of $O(\log_2 n)$.