International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2020): 7.803

A Comprehensive Study of Various Sorting Algorithms

Cherukuri Nischay Sai

Bachelor's Student, Computer Science Engineering (CSE) Malla Reddy College of Engineering, Maisamaguda, Hyderabad

Abstract: ***Sorting is a vital data structure activity that facilitates finding, organizing, and locating information. This research paper presents the primary sorting algorithm types, implementation of algorithms, examples, advantages, disadvantages are discussed. Sorting is regarded as a basic process in computer science since it serves as an intermediary step in a variety of activities. Finally, several sorting algorithms are compared with their practical efficiency, and values are recorded in tabular and graphical form. This paper aims to conduct a review of several sorting algorithms and evaluate the various performance variables among them.***

Keywords: Bubble sort, Selection sort, Insertion sort, Merge sort, Quicksort, Heap sort, Counting sort, Bucket sort, Radix sort

1. Introduction

Sorting is also called the ordering of the data that arranges the elements of a list in a particular order, either ascending order or descending order. The output is the reordering of the input or permutation. Sorting algorithm can be considered one of the fundamental tasks in many computer applications under searching in an ordered list takes less time when compared with an unsorted list. we can optimize searching to a high level if the data is in sorted order

A real-life scenario example of sorting is Dictionary where it stores the words in alphabetical order, so searching of any words becomes simple

2. Sorting Algorithms Classification:

Sorting algorithms are classified based on the following parameters

- **By the number of comparisons:**
  In this method, sorting algorithms are categorized based on the number of comparisons. For comparison-based sorting algorithms, the best case is O(nlogn), and the worst case is O(n^2). Comparison-based sorting algorithms need at least O(nlogn) comparisons for most inputs.

- **By memory usage:**
  There are two kinds of memory usage patterns, such as "In-place" sorting in which the algorithm needs constant memory size, i.e., O(1) beyond the items being stored. In contrast, other sorting methods use additional memory space according to the input size called "out-place" sorting. In-place sorting is slower than the sorting algorithms that use extra memory. Sometimes O(logn) additional memory is considered for "in-place.

- **By recursion:**
  Sorting algorithms are classified as recursive or non-recursive. Quicksort is the example for recursive, while insertion and selection sort is an example for non-recursive. Merge sort is an algorithm that uses both.

- **By stability:**
  A stable sorting algorithm maintains the relative order of elements with equal keys (equivalent elements retain their relative positions even after sorting. For instance, a sorting algorithm is stable whenever there are two elements A[0] and A[1] with the same value and with A[0] is shown up before A[1] in the unsorted list, A[0] will also show up before A[1] in the sorted list.

- **By adaptability:**
  In some sorting algorithms, the complexity changes based on the pre-sorted input, i.e., a pre-sorted list of input affects the running time. The algorithms that take this adaptability into account are called adaptive algorithms. For instance, Quicksort is the adaptive algorithm in which time complexity depends upon the initial input sequence. If the input is already sorted, then time complexity will be O(n^2), and if the input is not sorted then time complexity will be O(nlogn).

- **Internal sorting:**
  Internal sorting algorithms are sorting algorithms that sort using main memory or primary memory. This type of algorithm presumes high-speed random access to all memory. Some of the common algorithms that use internal sorting are insertion Sort, Bubble Sort, and Quicksort.

- **External Sorting:**
  Sorting algorithms that utilize external memory or secondary memory during the sorting come under this category. External sorting algorithms are comparatively slower than internal sorting algorithms. Merge sort uses external sorting.

3. Various Sorting Algorithms

Sorting algorithms are divided into two different categories:

Comparison sort:
In this type, the elements are sorted by comparing them with one another
Comparison based algorithms
1) Bubble sort
2) Selection sort
3) Insertion sort
4) Merge sort
5) Quicksort
6) Heapsort

Non- comparison sort:
In non-comparison algorithms, they are not sorted based on comparing with other elements. Instead, each algorithm uses its approach.

Non-comparison based algorithms
1) Counting sort
2) Bucket sort
3) Radix sort

3.1. Bubble sort

Bubble sort is also referred to as comparison sort, and it is a simple and the slowest sorting algorithm. Bubble sort works by iterating the first element to the last, comparing every element in the list with its neighboring elements, and swapping them if they are in undesirable order. This process is repeated until the algorithm has gone through the entire list without swapping any elements, indicating that the list is sorted. When the size of the input n is increased, the algorithm works slower. Hence it is considered the most inefficient sorting algorithm with a large amount of data.

Implementation:
```
void bubbleSort(int array[], int size) {
 for (int pass = 0; pass < pass - 1; ++pass) {
  // loop to compare array elements
    for (int i = 0; i < size - pass - 1; ++i) {
        if (array[i] > array[i + 1]) {
       int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }}
}}
```

Example:

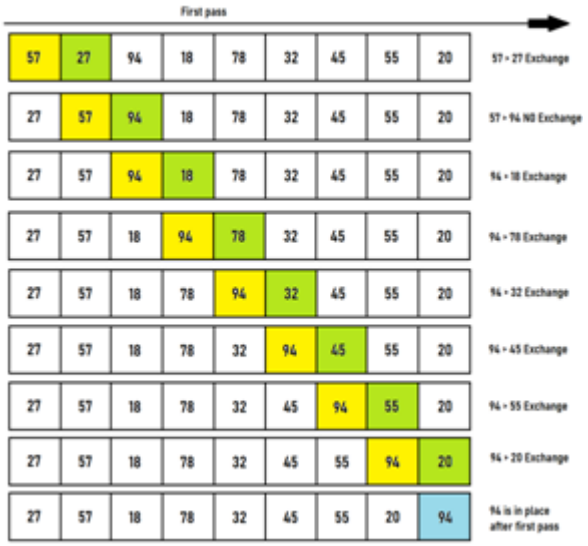


Figure 1: Example of Bubble Sort

3.2. Optimized Bubblesort:

In a regular Binary search, all the comparisons are made even if the list is sorted, and it increases execution time. To solve, we need to introduce a variable swapped. If there occurs changing of elements, then the value of the swap is set to true. Otherwise, it is said to be false.

Implementation:
```
void BubbleSortImproved(int A[], int N){
int pass, i, temp, swapped = 1;
for(pass = N – 1; pass >=0&& swapped; pass--)
swapped = 0;
for(i = 0; i <= pass; i++){
if(A[i] > A[i+1]){
temp = A[i];
A[i] = A[i+1];
A[i+1] = temp;
Swapped = 1;
}}}}
```

Advantages:
- It is easy to understand and simple to write, and takes few lines of code.

Disadvantage:
- It takes more time to execute.
- It is inefficient for the large volume of data.

3.3. Selection sort

Selection sort is a fundamental sorting algorithm, and it is an in-place comparison-based approach that separates the list into two halves, the sorted part on the left end and the unsorted part on the right. Initially, the sorted list is empty but, the unsorted list is the whole list.

The smallest element in the unsorted array is picked and swapped with the leftmost element, and that element becomes

a part of the sorted array. This procedure is continued until all items have been sorted.

Implementation:

```
void selection( int A[], int N){
int I, j, temp, min;
for ( i = 0; i < N-1; i++){
min = i;
for(j = i+1; j < N; j++){
if(A[i] < A[min]){
min = j;
}
temp = A[min];
A[min] = A[i];
A[i] = temp;
}}
```

Example:

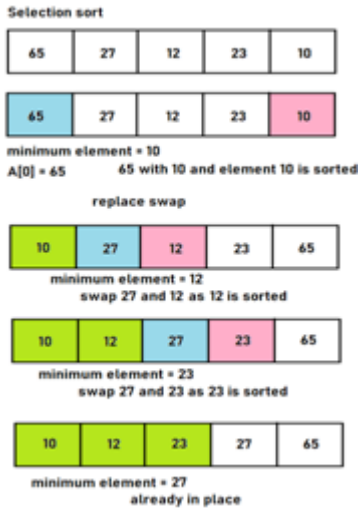


Figure 2: Example of Selection Sort

Advantages

- Selection sort is easy to implement.
- Selection sort does not require additional memory.

Disadvantages:

- It is not efficient for a large amount of data

3.4. Insertion sort

In Insertion sort each iteration removes one element from the input data and inserts it into the proper place in the final sorted list by comparing it to the adjacent elements. This operation continues until the list is sorted in the desired order. Insertion sort is an in-place algorithm and only needs a constant amount of additional memory.

Insertion sort takes $\frac{n^2}{4}$comparisons and $\frac{n^2}{8}$ swaps in the average case, and in the worst case, they are double.

Implementation:

```
void InsertionSort(int A[], int n){
int i, j, v;
for(i=1; i <s= n-1; i++){
v = A[i];
j = i;
while(A[j-1] > v && j >= 1){
A[j] = A[j-1];
j--;
}
A[j] = v;
}}
```

Example:

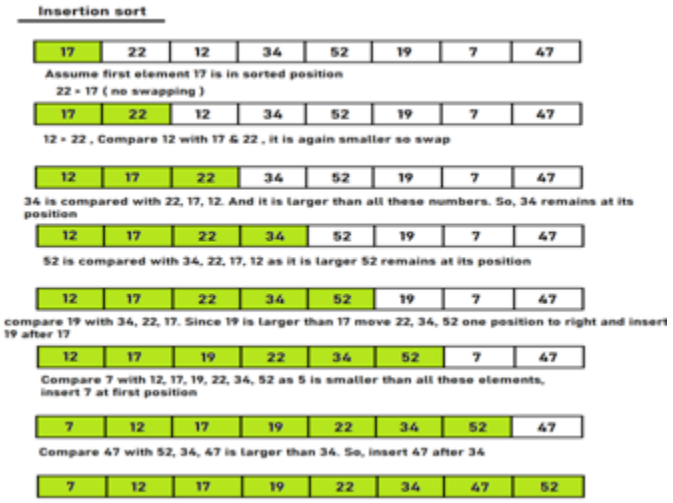


Figure 3: Example of Insertion Sort

Advantages:

- Simple to implement.
- Efficient for small data.
- Maintains relative order of input data if the keys are the same.

Disadvantages:

- Inefficient for a large amount of data.

3.5. Merge sort

Merge sort uses Divide and Conquer strategy to solve a given problem. It works by splitting the unsorted array into several sub-lists in which each sub-list consists of one element. Generally, an array with one element is considered to be sorted. Finally, it merges each sub-array to generate a sorted array. Merge sort is a stable sort, implying it keeps the relative order of elements with the same key. Merge sort may also be implemented non-recursively, although most people prefer the recursive technique since non-recursive is inefficient.

Implementation:

```
void mergeSort(int a[], int start, int end)
{
   int mid;
   if(start<end)
   {
      mid = (start+end)/2;
```

```
    mergeSort(a,start,mid);
    mergeSort(a,mid+1,end);
    merge(a,start,mid,end);
  }}
void merge(int a[], int start, int mid, int end)
{
  int i=start,j=mid+1,k,index = start;
  int temp[10];
  while(i<=mid && j<=end)
  {
    if(a[i]<a[j])
    {
      temp[index] = a[i];
      i = i+1;
    }
    else
    {
      temp[index] = a[j];
      j = j+1;
    }
    index++;
  }
  if(i>mid)
  {
    while(j<=end)
    {
      temp[index] = a[j];
      index++;
      j++;
} }

  else
  {
    while(i<=mid)
    {
      temp[index] = a[i];
      index++;
      i++;
} }

  k = start;
  while(k<index)
  {
    a[k]=temp[k];
    k++;
} }
```

Example:

Implementation of merge sort is performed with 8 elements in the list.

Figure 4: Example of Merge Sort

Advantages:

- Fast for large arrays, unlike insertion, selection, and a bubble sort, it doesn't traverse the whole array many times.

Disadvantages:

- Merge sort is works slow for small arrays.
- The algorithm does the whole working process even if the array is sorted.
- It requires extra space to store sub arrays.

3.6. Quicksort:

Quicksort is a highly efficient sorting algorithm, and it uses a divide and conquer approach. The name "Quicksort" comes from the fact that it can sort the data elements significantly faster than any other sorting algorithm. It works by selecting an element from the unsorted array named pivot and splitting it into two parts called sub-arrays, one with elements greater than the pivot and the other with elements smaller than a pivot. Repeat the algorithm repeatedly for both halves of the array. Quicksort is also called a partition exchange sort.

Implementation:

```
void QuickSort( int A[], int low, int high){
int pivot;
if(high > low){
pivot = Partion(A, low, high);
QuickSort( A, low, pivot – 1);
QuickSort( A, pivot + 1, high);
}}
int Partion(int A, int low, int high){
int left, right, pivot_item = A[low];
left = low;
right = high;
while( left < right){
while( A[left] <= pivot_item)
left++;
while(A[right] > pivot_item)
right--;
if(left< right)
swap(A, left, right);
}
A[low] = A[right];
A[right] = pivot_item;
return right;
```

}

Example: Implementation of Quick sort is shown below.

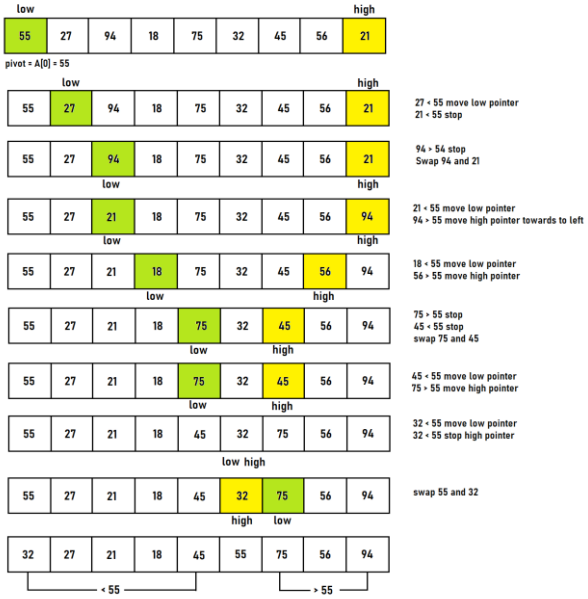


Figure 5: Example of Quick Sort

Randomized Quick sort:

In order to reduce the probability of getting the worst case in quicksort, we can add randomization to the algorithm. We can add randomization by randomly choosing an element in the input data for a pivot. We will use a randomly chosen element from the subarray A[low..high] in the randomized version of Quicksort.

Advantages:

- Quicksort is fast and efficient when dealing with a large amount of data.
- Quicksort is an in-place in which no additional storage is required.

Disadvantages:

- If the list is already sorted, then bubble sort is more efficient than quicksort.
- Quicksort's worst-case performance is the same as average performances of the bubble, insertion, and selection sorts.

3.7 Heap sort

Heap sort is based on the heap data structure and in-place sorting algorithm. Heap is a specialized tree-based data structure that satisfies the heap property binary tree is the most commonly used. Heap sort works by first constructing a heap from the input array, then removing the most significant element from the heap and storing it at the end of the array, i.e., at the n-1 position. When it eliminates the heap's maximum element, it restores the heap property until the heap is empty. As a result, it removes the second largest element from the heap and places it in position n-2, and so on. The algorithm repeats this operation until the array is sorted. Heapsort is not a stable sort because it does not preserve the relative order of elements with equal keys.

Implementation:

```
void heapify(int A[], int size, int i)
{
int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;
  if (left < size && A[left] >A[largest])
largest = left;
if (right < size && A[right] > A[largest])
largest = right;
  if (largest != i)
{
    int temp = A[i];
    A[i]= A[largest];
    A[largest] = temp;
heapify(A, size, largest);
}}
 void heapSort(int A[], int size)
{
int i;
for (i = size / 2 - 1; i >= 0; i--)
heapify(A, size, i);
for (i=size-1; i>=0; i--)
{
int temp = A[0];
    A[0]= A[i];
    A[i] = temp;
heapify(A, i, 0);
}  }
```

Example: Implementation Heap sort is shown below with 10 elements in the array.

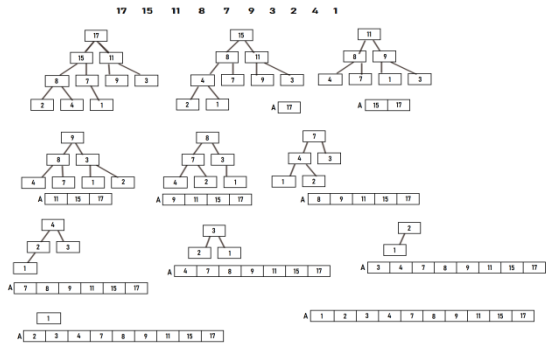


Figure 6: Example of Heap Sort

Advantages:

- Heap sort can be implemented as an in-place sorting algorithm.
- Heap sort is widely used because of its efficiency, and memory utilization is minimal.

Disadvantages:

- Heap sort is not a stable sort.
- In many cases, Quicksort is more efficient than Heap sort.

3.8. Counting sort

Counting sort is an integer sorting algorithm with linear running time complexity. For some integer K, counting sort assumes that each element is an integer in the range 1 to K. It works by counting the number of occurrences of each element in the input, usually called keys, and store this information in another array, we need to modify the array to identify the location of each key value in the output sequence, use the prefix sum on the counts. Counting sort is not a comparison sort, and it preserves the relative order of the elements with equal keys.

In counting sort, A[0 ..n – 1] is the input array with length n, and we need two more arrays: let us assume array B[0 ..n – 1] contains the sorted output and array C[0 ..K – 1] provides temporary storage.
Implementation:

```
void CountingSort(int A[], int n, int B[], int K){
int C[k], i, j;
for(i=0; i<K; i++)
C[i] = 0;
for(j = 0; j < n; j++)
C[A[j]] = C[A[j]] + 1;
for(i=1; i < K; i++)
C [i] = C [i] + C[i-1];
for( j = n-1; j >= 0; j--){
B[C[A[j]]] = A[j];
C[A[j]] = C[A[j]] – 1;
}}
```

Example:

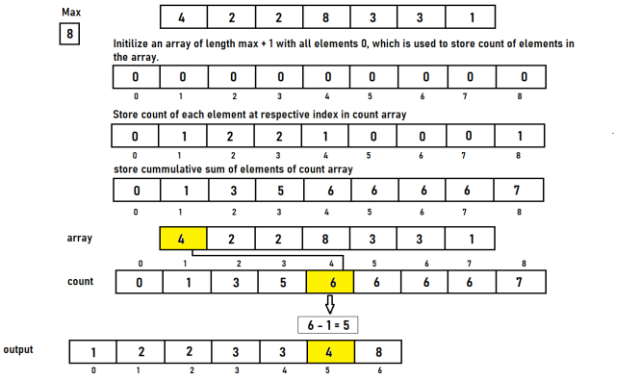


Figure 7: Example of Count Sort

Advantages:

- Count sort has a Linear Time Complexity. Since it is not a comparison-based sorting, it is not lower bounded by O(nlogn).

Disadvantages:

- Counting sort is not suitable for extensive input data.
- Counting sort cannot be used for an array with non-integer elements.

3.9. Bucket sort:

Bucket sort is a sorting algorithm that divides the unsorted array elements into several buckets. Each bucket is sorted independently, either using various sorting algorithms or recursively implement the same technique. Bucket sort finally gathers the elements in the sorted order.

Bucket sort can be understood as a scatter-order-gather approach because elements are first scattered in buckets, ordered within them, and finally gathered into a sorted list.

Implementation:

```
#define BUCKETS 10
void BucketSort( int A[], int array_size) {
int i, j, k;
int buckets[BUCKETS];
for( j = 0; j < BUCKETS; j++)
buckets[j]=0;
for(i=0; i< array_size; i++)
++buckets[A[i]];
for(i=0,j=0;j<BUCKETS; j++)
for(k= buckets[j]; k >0; --k)
A[i++] = j;
}
```

Example:

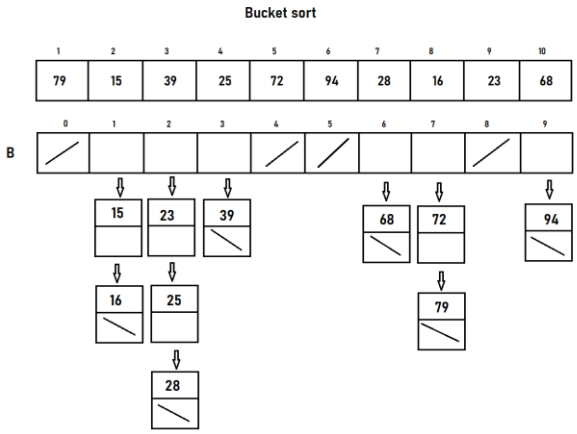


Figure 8: Example of Bucket Sort

Advantages:

- Each bucket may be processed individually using bucket sort. As a result, after sorting the primary array, you'll usually need to sort much smaller arrays as a subsequent step.

Disadvantages:

- The performance of bucket sort depends on the number of buckets selected, which may need some additional performance adjustment compared to other algorithms.
- Bucket sort's efficiency is dependent on the distribution of the input values. Thus it's not worth it if you have closely clustered values.

3.10 Radix sort

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In Radix sort, first sort the elements based on the least significant digit (last digit). These results are again sorted by the next least significant digit(second digit). Repeat for all numbers until we reach the most significant digits. Use some stable sort to sort them by the last digit. Then, stable sort them by the second least significant digit, then by the third, and so on. If we use counting sort as the stable sort, the total time is O(nd) ≈ O(n).

The number of passes required by the Radix sort algorithm equals the number of digits contained in the biggest number in the list of numbers. For example, if the biggest number is a three-digit number, the list is sorted three times.

Implementation:

```
int getMax(int list[], int n) {
   int mx = list[0];
   int i;
   for (i = 1; i < n; i++)
      if (list[i] > mx)
         mx = list[i];
   return mx;
}
void countSort(int list[], int n, int exp) {
   int output[n];
   int i, count[10] = { 0 };
   for (i = 0; i < n; i++)
 count[(list[i] / exp) % 10]++;
  for (i = 1; i < 10; i++)
      count[i] += count[i - 1];
for (i = n - 1; i >= 0; i--) {
      output[count[(list[i] / exp) % 10] - 1] = list[i];
      count[(list[i] / exp) % 10]--;
   }
  for (i = 0; i < n; i++)
      list[i] = output[i];
}
void radixsort(int list[], int n) {
   int m = getMax(list, n);
int exp;
   for (exp = 1; m / exp > 0; exp *= 10)
      countSort(list, n, exp);
}
```

Examples:

Radix sort

Input	1st pass	2nd pass	3rd pass
219	791	219	219
445	372	627	372
627	884	445	445
884	884	445	445
433	445	372	627
791	627	884	884
372	219	791	791

Figure 9: Example of Radix Sort

Advantages:

- Radix algorithm works fast when the array elements have less range of the values that they can possess, which results in short keys.
- Radix Sort is a stable sort since it maintains the relative order of elements with equal values.

Disadvantages:

- Radix Sort is not as flexible as other sorts since it is dependent on numbers or letters. As a result, for every different type of data must be rebuilt.
- Radix sort is not an in-place sorting algorithm, so it requires extra additional space.4

4. Analysis of various Sorting Algorithms:

The comparison of various search algorithms is done based on their time and space complexity is shown below.

Table 1: Analysis of Various Sorting Algorithms

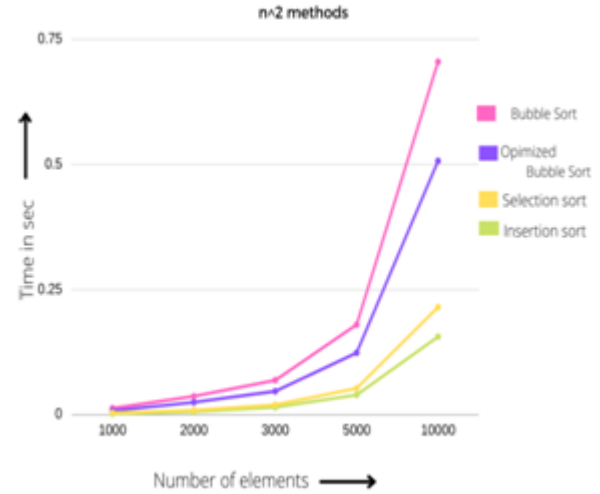
S.NO	Sorting Algorithm	Table Complexity			Space complexity	Stable
		Best case	Average case	Worst case		
1)	Bubble sort	O(n)	O (n^2)	O (n^2)	O(1)	Yes
2)	Selection sort	O (n^2)	O (n^2)	O (n^2)	O(1)	Yes
3)	Insertion sort	O(n)	O (n^2)	O (n^2)	O(1)	Yes
4)	Merge sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Yes
5)	Quick sort	O(nlogn)	O(nlogn)	O (n^2)	O(1)	No
6)	Heap sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	No
7)	Counting sort	O(n+k)	O(n+k)	O(n+k)	O(k)	Yes
8)	Bucket sort	O(n+k)	O(n+k)	O (n^2)	O(n+k)	Yes
9)	Radix sort	O(nk)	O(nk)	O(nk)	O(n+k)	Yes

5. Experimental Analysis

5.1. Case 1: Results of Case 1 is shown in Table 1. It shows running time of O(n^2) sorting algorithms, the size of array varied from 1000 to 10000.

Table 2: Comparison of O(n^2) Sorting Algorithms

Number of elements	Bubble sort	Optimized Bubble sort	Insertion sort	Selection sort
1000	0.013	0.008	0.0021	0.004
2000	0.037	0.025	0.006	0.009
3000	0.069	0.047	0.0150	0.020
5000	0.180	0.124	0.0398	0.053
10000	0.705	0.507	0.156	0.215

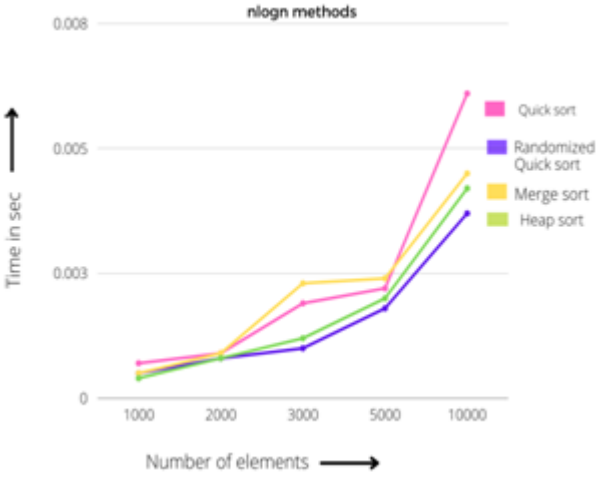


Graph 1: Comparison of O (n^2) Sorting Algorithms

5.2. Case 2: Results of Case 2 is shown in Table 2. It shows running time of O(nlogn) sorting algorithms, the size of array is varied from 1000 to 10000.

Table 3: Comparison of O(nlogn) Sorting Algorithms

Number of elements	Quick sort	Randomized Quick sort	Merge sort	Heap sort
1000	0.0007	0.0005	0.0005	0.0004
2000	0.0009	0.0008	0.0009	0.0008
3000	0.0019	0.0010	0.0023	0.0012
5000	0.0022	0.0018	0.0024	0.0020
10000	0.0061	0.0037	0.0045	0.0042

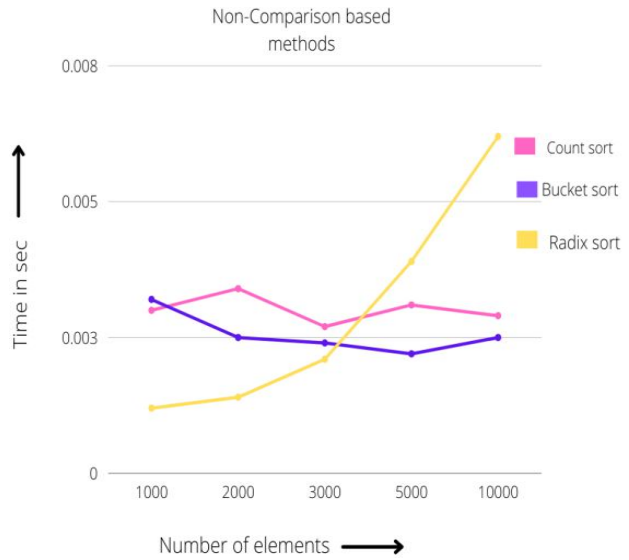


Graph 2: Comparison of O (n^2) Sorting Algorithms

5.3. Case 3: Results of Case 3 is shown in Table 3. It shows running time of Non-Comparison based sorting algorithms, the size of array is varied from 1000 to 10000.

Table 4: Comparison of Non-Comparison Sorting Algorithms

Number of elements	Counting sort	Bucket sort	Radix sort
1000	0.0030	0.0032	0.0012
2000	0.0034	0.0025	0.0014
3000	0.0027	0.0024	0.0021
5000	0.0031	0.0022	0.0039
10000	0.0029	0.0025	0.0062



Graph 3: Comparison of Non-Comparison Sorting Algorithms

6. Conclusion

Sorting is used in a wide range of applications and on a wide range of data. Because it is such a frequent and resource-intensive activity, sorting algorithms and the development of optimum implementations are an important area of computer science. With the evolution of new technology and increased Internet usage, data availability is also grown on the Internet, creating demand for a fast and efficient sorting algorithm.

This paper discusses nine different sorting algorithms, along with their implementation, benefits, drawbacks, and examples, as well as actual performance findings. The experimental results demonstrate that the theoretical performance behavior of each sorting algorithm is related to its actual performance. According to the experiment's findings, the algorithm's behavior will vary depending on the size of the items to be sorted.

In this paper, three different types of performance behavior were investigated, such as O(n^2) class and O(nlogn) and non-comparison class. In large data set, Non-comparison based sorting algorithms outperform the comparison-based sorting algorithms in large data sets, and O(nlogn) class outperforms as O(n^2) class in large data set.

References

[1] Narasimha Karumanchi, ”Data Structures and Algorithms”.
[2] Aditya Bhargava, “Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People”.
[3] Thomas Niemann, “Sorting and Searching Algorithms”.
[4] https://en.wikipedia.org/wiki/Sorting_algorithm
[5] Hammad, J. A Comparative Study between Various Sorting Algorithms, International Journal of Computer

Science and Network Security (IJCSNS), Vol 15, No. 3 (2015).
[6] Robert Sedgewick, “Algorithms in C++ Parts 1-4: Fundamentals, Data Structure, Sorting, Searching”.
[7] Kurt Mehlhorn, “Data Structures and Algorithms 1: Sorting and Searching Kurt Mehlhorn”.
[8] D. R. Babu, R. S. Shankar, V. P. Kumar, C. S. Rao, D. M. Babu and V. C. Sekhar, "Array-indexed sorting algorithm for natural numbers," 2011 IEEE 3rd International Conference on Communication Software and Networks, 2011, pp. 606-609, doi: 10.1109/ICCSN.2011.6014966.
[9] Robert Sedgewick, “Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms. Bundle of Algorithms in Java”, AddisonWesley Professional, Third Edition.
[10] Cormen, T. H., Leiserson, C.E., & Rivest, R.L. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill (2009).
[11] Cormen, T.H., Leiserson, C.E., & Rivest, R.L. Introduction to Algorithms (2nd ed.). Prentice Hall of India private limited, New Delhi-110001 (2001).
[12] Niklaus Wirth, Algorithms + Data Structures = Programs.