



# **SORTING ALGORITHMS: A Comparative Study**

**MEMBERS:**

19BCI0148(ROHIT ANIL KUMAR)

19BCI0157(BHUVAN SIREESH)

19BCE0197(AMIT KRISHNA A)

19BCI0125(MADHAV H NAIR)

Report Submitted for the final review of  
DATA STRUCTURES AND ALGORITHMS(CSE2003)

Professor: DR. ILANTHENRAL KANDASAMY

# **INTRODUCTION**

Efficiency can be defined as complete and optimum usage of all resources available on the computer. Efficiency is not the same as speed since it only takes into factor time, while neglecting all other factors. Speed along with memory utilization would account for good efficiency. Other resources include functionality, power consumption, user friendliness and effort in computer development. A perfect ratio of what we want to what it consumes in terms of computer resources would lead to maximum efficiency.

In computer science, an algorithm that puts elements of a list into a certain order is known as sorting algorithm. The orders that are mainly used are numerical order and lexicographical order. For making use of other algorithms (like search and merge algorithms) sorted lists are required to work correctly; it is also often useful for conforming to well-established patterns or rules of data and for producing such output which is easy to read and understand. There are two conditions enlisted that output must satisfy. These conditions are:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order)
2. The output is a permutation or reordering of the input.

# Literature Review Summary Table

Authors and Year of Reference	Title of the Study	Concept/Theoret ical model	Methodolo gy used/ Implement ation	Dataset details/ Analysis	Relevant Finding	Limitations/ Future Research/ Gaps identified
	Sorting algorithms					

## **Objective**

In this paper we are going to investigate the various types of sorting algorithms and analyse their efficiency in computer systems. When writing a computer program, the primary concern must be efficiency.

An algorithm is a set of predefined instructions for a computer to perform a task. Apart from computer science, algorithms are also used in various other fields of science like mathematics, physics, statistics and life sciences. But in the world of computer science, algorithms are used for activities such as calculations, summarizing, sorting, classifying and retrieval.

Our task in this paper is to look at two particular types of data structures and compare how they could be sorted by using 5 common sorting algorithms. We shall thereafter analyse the performance and contrast between them and arrive at a conclusion.

## **Innovation**

## **Implementation**

### **1.Bubble Sort:**

Bubble Sort is a simple algorithm that starts at the beginning of the given data and compares the first two elements. If the first element is greater than the second, then the items are swapped. This process is continued for adjacent pairs till the end of the data set. It then repeats the cycle until no more swaps occur.

Bubble Sort has an average and worst case performance of  $O(n^2)$ , so it is rarely used to sort large, unordered data sets. This causes larger values at the end of the list while smaller bubbles sink towards the beginning of the list. Bubble sort can be used to sort a small number of items (where inefficiency is not a high penalty).

Bubble Sort may also be efficiently used on a list that is already sorted except for a very small number of elements. For example, if only one element is not in order, bubble sort will take only at most  $3n$  time.

### **Advantages:**

1. Simplicity and ease of implementation.
2. Auxiliary space used is  $O(1)$ .

### **Disadvantages:**

1. Very inefficient.

General Complexity is  $O(n^2)$ . Best case complexity is  $O(n)$

### **Code:**

```
void bubble_sort(long int *arr, long int start, long int end){
    long int i, j, temp, n=end+1;
    for(i=0;i<n-1;i++){
        for(j=0;j<n-i-1;j++){
            if(arr[j]>arr[j+1]){
                temp = arr[j] ;
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

## 2. Insertion sort:

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort is a variant of insertion sort that is more efficient for larger lists.

### Advantages:

1. Auxiliary space used is  $O(1)$ .

### Disadvantages:

1. General Complexity is  $O(n^2)$
2. Best Case is  $O(n)$  when the list is already sorted.

### Code:

```
void insertion_sort(long int *arr, long int start, long int end){
    long int i, j, temp, n=end+1;
    for(i=0;i<n;i++){
        temp = arr[i];
        j = i;

        while(j>0 && temp<arr[j-1]){
            arr[j] = arr[j-1];
            j--;
        }
        arr[j] = temp;
    }
}
```

## 3. Merge Sort:

Merge sort starts by dividing up all the elements into  $n$  sub-lists. Afterwards, each sub-list is compared to the next one and they merge into new sorted sub-list containing two elements. Next step is to compare these two sub-lists with one next to it so it will create a new sub-list of four sorted elements. The process is repeated so on

for eight, sixteen and so on until there is only one remaining list. This is the finalized set of sorted elements.

**Advantages:**

1. Marginally faster than the heap sort for larger sets.
2. Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

**Disadvantages:**

1. At least twice the memory requirements of the other sorts because it is recursive. This is the BIGGEST cause for concern as its space complexity is very high. It requires about a  $\Theta(n)$  auxiliary space for its working.
2. Function overhead calls ( $2n-1$ ) are much more than those for quick sort ( $n$ ). This causes it to take more time marginally to sort the input data.

**Code:**

```

void merge(long int* arr, long int start, long int mid, long
    long int i, j, k, L[100000], R[10000], m, n;

    m = mid-start+1;
    n = end-mid;
    for(i=start;i<=mid;i++){
        L[i-start] = arr[i];
    }
    for(i=mid+1;i<=end;i++){
        R[i-mid-1] = arr[i];
    }
    i=j=0;
    k=start;
    while(i<m && j<n){
        if(L[i]<=R[j]){
            arr[k] = L[i];
            i++;
        }
        else if(L[i]>R[j]){
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while(i<m){
        arr[k] = L[i];
        i++;
        k++;
    }
    while(i<n){
        arr[k] = R[j];
        i++;
        k++;
    }
}

void merge_sort(long int *arr, long int start, long int end){
    long int mid;
    if(start>=end){
        return;
    }

    mid = start + (end-start)/2;
    merge_sort(arr, start, mid);
    merge_sort(arr, mid+1, end);
    merge(arr, start, mid, end);
}

```

---



## 5. Quick Sort:

Quick Sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. All elements smaller than the pivots are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sub-lists are then recursively sorted. Efficient implementations of quick sort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest  $O(\log n)$  space usage, quick sort is one of the most popular sorting algorithms and is available in many standard programming libraries. The most complex issue in quick sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower  $O(n^2)$  performance, if at each step the median is chosen as the pivot then the algorithm works in  $O(n \log n)$ . Finding the median however, is an  $O(n)$  operation on unsorted lists and therefore exacts its own penalty with sorting.

### **Advantages:**

- One advantage of parallel quick sort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sub list is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.
- All comparisons are being done with a single pivot value, which can be stored in a register.
- The list is being traversed sequentially, which produces very good locality of reference and cache behaviour for arrays.

### **Disadvantages:**

- Auxiliary space used in the average case for implementing recursive function calls is  $O(\log n)$  and hence proves to be a bit space costly, especially when it comes to large data sets.
- Its worst case has a time complexity of  $O(n^2)$  which can prove very fatal for large data sets

## Code:

```
void swap(long int *a, long int *b){
    long int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int partition(long int *arr, long int start, long int end){
    long int i=start+1, j=end, pivot=arr[start];

    while(i<j){
        while(arr[i]<=pivot)
            i++;
        while(arr[j]>pivot)
            j--;
        if(i<j){
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[start], &arr[j]);
    return j;
}

void quick_sort(long int *arr, long int start, long int end){
    long int part;

    if(start>=end){
        return;
    }
    part = partition(arr, start, end);
    quick_sort(arr, start, part-1);
    quick_sort(arr, part+1, end);
}
```

## Selection Sort:

This 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 sorted sub array
2. Unsorted Sub array

This algorithm has a time complexity of  $O(n^2)$  as there are two nested loops

### **Advantages:**

The good thing about selection sort is it never makes more than  $O(n)$  swaps and can be useful when memory write is a costly operation.

### **Disadvantages:**

This algorithm has poor efficiency when dealing with a huge list of items

### **Code:**

```
void selection_sort(long int *arr, long int start, long int end){
    long int i, j, temp, n=end+1;
    long int smallest;
    for(i=0;i<n-1;i++){
        smallest=i;
        for(j=i+1;j<n;j++){
            if(arr[j]<arr[smallest]){
                smallest = j;
            }
        }
        temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
    }
}
```

### **Methodology:**

The runtimes were compared in various languages including C++, Java and Python and it was found that C++ had the least runtime despite following the same algorithms and hence C++ has been used as the standard language for comparison in the project.

The *time.h* module of C++ was used to compute the execution time of each algorithm. The dataset for the experiment was obtained by using *random.h* to get an array of random numbers which was then fed to individual algorithms.

### **Dataset used:**

The array initially used was small in size containing small integer values. Later on, to compare the time complexities bigger arrays

were taken to be sorted. The data for such big arrays were randomly generated in C++ using *random.h* module.

## Tools used:

1. Dev C++ IDE
2. Codeblocks IDE

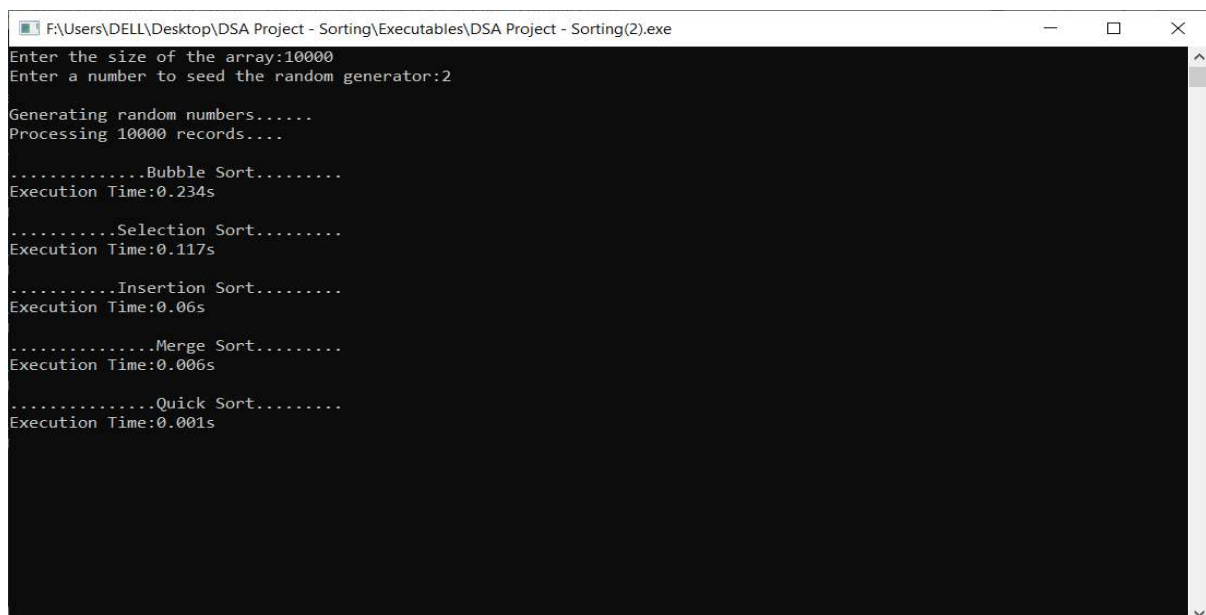
## Results:

The execution time for the different algorithms was obtained and tabulated for two different cases as follows:

### For an array of random numbers

Records	Bubble	Select	Insert	Merge	Quick
1000	0.0005	0.0015	0.0005	0.0005	0
10000	0.383	0.2025	0.104	0.0105	0.0005
20000	1.5365	0.7805	0.3765	0.0225	0.01
30000	3.601	1.6755	0.7995	0.0425	0.001
40000	6.66	3.1735	2.0285	0.0545	0.004
50000	10.718	5.0085	2.7195	0.086	0.0155
60000	15.097	7.196	3.672	0.0855	0.008

## Screenshot:



```
F:\Users\DELL\Desktop\DSA Project - Sorting\Executables\DSA Project - Sorting(2).exe
Enter the size of the array:10000
Enter a number to seed the random generator:2

Generating random numbers.....
Processing 10000 records....

.....Bubble Sort.....
Execution Time:0.234s

.....Selection Sort.....
Execution Time:0.117s

.....Insertion Sort.....
Execution Time:0.06s

.....Merge Sort.....
Execution Time:0.006s

.....Quick Sort.....
Execution Time:0.001s
```

## For an Already Sorted Array

Records	Bubble	Select	Insert	Merge	Quick
1000	0	0	0	0.008	0
10000	0.149	0.1615	0	0.0145	0.1235
20000	0.6205	0.656	0.0005	0.025	0.428
30000	1.3515	1.558	0	0.039	0.2325
40000	2.41	2.7685	0	0.055	0.2765
50000	3.798	4.2635	0.0005	0.0655	0.4395
60000	5.4485	6.175	0	0.079	0.693
70000	7.639	8.071	0	0.09	0.284

## Screenshot:

```
F:\Users\DELL\Desktop\DSA Project - Sorting\Executables\DSA Project - Sorting(4).exe
Enter the size of the array:10000
Enter a number to seed the random generator:3

Generating random numbers.....

First sorting the array using bubble sort
Now Processing 10000 records....

.....Bubble Sort.....
Execution Time:0.108s

.....Selection Sort.....
Execution Time:0.12s

.....Insertion Sort.....
Execution Time:0s

.....Merge Sort.....
Execution Time:0.005s

.....Quick Sort.....
Execution Time:0.086s
```

### **Conclusions:**

<b><u>Algorithms</u></b>	<b><u>Conclusion</u></b>
Bubble	Very slow with many but are easy to understand and implement
Selection	Poor efficiency when dealing with huge lists but has decent speed when working with a small dataset.
Insertion	Decent speed and works well when most of the elements are already sorted
Merge	Fast and efficient in all cases but is complex to implement and also consumes a lot of space.
Quick	Slow when most elements are already sorted. Efficient and fast on random datasets.

---

### **References:**