Experimental Verification and Analysis of Sorting Algorithms

Noor Emam 900222081
Ismail ElDahshan 900221719
Amr Eid 900222213
Ismaiel Sabet 900221277
CSCE 1101
Dr. Amr Goneid

May 22, 2023

Abstract

Advancements in computer science have greatly enhanced the ease and efficiency of people's lives, with sorting algorithms playing a significant role. Sorting algorithms in the developing world have become very crucial, which has resulted in Computer Scientists and Mathematicians constantly developing new algorithms that help achieve both power and efficiency. Sorting algorithms are crucial to computer science since they help search, sort, and organize data, particularly in large databases. Studying these algorithms will help us understand each algorithm's benefits, drawbacks, applications and even deduce what is the most efficient algorithm out of all of them.

Keywords: Sorting algorithms, Selection sort, Insertion Sort, Merge Sort, Quick sort, Models, Complexities.

1 Introduction

Several sorting algorithms have been created over the years to deal with different types of data. Each type of dataset requires a special sorting algorithm for it. For our purposes, efficient sorting algorithms are needed to save space and time complexities. Each sorting algorithm has its own benefits and limitations. In this experiment, 4 types of sorting algorithms will be tested, which are selection sort, merge sort, quick sort, and insertion sort. Merge sort and Quick sort are both types of divide-and-conquer algorithm. Basically, the divide-and-conquer algorithm, from its name, divides the problem into several, smaller pieces. What merge and quick sort do is that they split the main array into several, smaller pieces by recursion, and then they compare the array elements to one another to sort the smaller array back on its way the main array. Next, the insertion sort's array compares an array element to its previous array element. If the subsequent array element has a smaller value than the previous array element, then a swap function is called. The process is repeated until the subsequent array element swaps with every prior element. The process is then repeated until the array comparisons are finished. The selection sort algorithm works by repeatedly selecting the smallest element from an unsorted portion of the list and placing it in its correct position in the sorted portion. In this project report, the comparison of the four sorting algorithms will be tackled and their performance under different array sizes will be analyzed.

2 Problem Definition

Sorting algorithms are required to be analyzed very carefully to determine their efficiency. Since it can sometimes be too complex to trace it using stan-

dard methods, a plan has been devised to create a code that determines the number of comparisons that each sorting algorithm takes. Based on the number of comparisons, a decision will be taken on which sorting algorithm is better than the other. The simplicity of each algorithm will be evaluated, and this is part of the evaluation of the sorting algorithms to find out where each algorithm can be best applied.

3 Methodology

This project aims to research how each sorting algorithm would act under different array sizes. We introduce the size N of type integer under varying sizes, which are 1000, 2000, 3000, 5000, 7000, and 10000. Through the implementation of the code at part IV, we find out the number of comparisons done by each sorting algorithm under different array sizes. We find that the number of comparisons done by each array to be the most essential thing because this will demonstrate the sorting algorithm's number of operations carried out by the process. For instance, the more the number of operations, the less efficient the sorting algorithm is. We use two important helper functions, which are the RPA and swap functions. Furthermore, we introduce the variable "count" of type integer into each sorting algorithm to test out the number of comparisons being done by each sorting algorithm. We will implement these sorting algorithms into classes, each of the divide and conquer sorting algorithms, has its own class and we will put the helper function as a separate class. We create an object for the helper function class to be used as a private data member in the sorting classes. Also, we will make each divide and conquer sorting algorithm functions as public. Through this way, we will help store the helper functions as private data members that can be accessed ONLY through the sorting functions.

4 Specifications of Algorithms Used

Sorting is one of the most essential things used in computer science. As previously stated, it almost has been used in every sub-discipline of computer science. We will go over every sorting algorithm to figure out its time complexities, and in which application is the sorting algorithm best at being utilized. We will also analyze the average case scenario for each of the four sorting algorithms.

5 Selection Sort

Selection sort is a comparison-based algorithm that works by finding the smallest element in the array and then swapping it with the element in its correct position. It keeps on repeating the process until the array has been fully sorted [2].

5.1 Time Complexity

• The worst, the average, and the best time complexities are the same, which is O(N2) or the big O[2].

5.2 Advantages

• It is a stable algorithm
• Efficient for small-data input[2]

5.3 Disadvantages

• Inefficient for large data input
• Even if the array was partially sorted, it would go to compare everything, making it an ineffective algorithm[2].

6 Insertion Sort

The insertion sort algorithm is a simple sorting algorithm which get the second element in the array then compares with the first element, if they the second element is less than the first, they swap with each other. Then each of the elements will increment

while also repeating the same process until the array has been fully sorted[3].

6.1 Time Complexity

- Worst case: O(N2)
- Average case: O(N2)
- Best case: O(N) [3]

6.2 Advantages

• Like the selection sort, the insertion sort is efficient for small input values.
• Very efficient if the array is already almost sorted.
• It is a stable sorting algorithm [3]

6.3 Disadvantages

• It becomes very inefficient at large input data.
• If the array is sorted backwards, then the insertion sort algorithm becomes very inefficient [3].

7 Quick Sort

Quicksort is a divide-and-conquer algorithm that works by dividing an array into two sub-arrays. This process is a recursive process. The algorithm starts by providing a pivot to a variable, and this pivot is decided by a function called partition. This function makes all the elements that are greater than the pivot on the right side, and all the elements that are less than the pivot on the left side. Then the recursive function Quicksort is called, and the process is repeated. The recursive function ends when the low index is higher than the high index. The Quicksort algorithm uses the pivot as the first element of an array, while the rQuicksort algorithm uses the pivot as a random element.

7.1 Time Complexity

• It depends on the location of the pivot.
• Worst case: O(n2) $\rightarrow$ pivot is chosen at a bad place
• Average case: O(n * logn)
• Best case: O(n * logn) [4]

7.2 Advantages

• Efficient on large data input.
• In-place algorithm, which means it doesn't require additional memory space.
• Much better than the insertion sort and selection sort algorithms[3].

7.3 Disadvantages

• Not so efficient in small data input.
• The algorithm may not be stable because some array elements may be lost due to their similarity[4].

8 Merge Sort

Merge sort algorithm is another divide-and-conquer algorithm that works by dividing the array into two parts. The process is repeated until the size of the subarray becomes equal to one. Then, two subarrays are compared to each other, and they are merged together in a sorted format. The process repeats until the recursive call has ended.

8.1 Time Complexity

• The merge sort algorithm best case, worst case, and average case are all the same. The time complexity is O(n * log n) [4]

8.2 Advantages

• It works greatly for large data input.
• Adaptable to any input data[5].

8.3 Disadvantages

• It sometimes requires addition memory space due to sub-divided arrays.
• Not very efficient in small data input.
• It will still have the same time complexity, even if it is partially or full sorted[5].

9 Data Specifications

In this project report, we try to analyze the number of comparisons done by each sorting algorithm. We verify and analyze 4 sorting algorithms, which are Merge sort, Quicksort, Insertion Sort, and Selection Sort algorithms. Our objective is to find out which algorithm performs the best under different sizes and to verify the number of comparisons being made by each algorithm. The data will contain random generated numbers from 1000, 2000, 3000, 5000, 7000 and 10000, depending on the sizes. Moreover, we will use the random permutation array (RPA for short) to generate random numbers that in an order we do not know to avoid bias. The random permutation array will work from indices 1 to N+1 so that the program can execute successfully. Furthermore, we will include ctime and stdlib.h libraries to ensure that the array is randomized for each run. This will make our experiment more efficient and more reliable to be analyzed. The experimental results will be reported on a tabular and graphical format showing the model and experimental results. This will help researchers reading this report to visualize the result of the number of comparisons of each algorithm.

10 Experimental Results

10.1 Selection Sort

10.1.1 Tabular Results

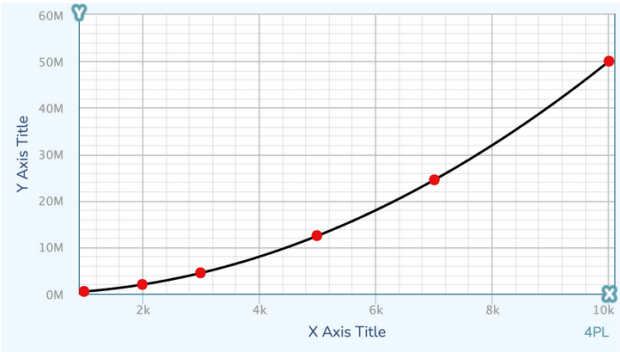
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	499500	1999000	4498500	12497500	24496500	49995000
T(N) Model	499500	1999000	4498500	12497500	24496500	49995000

Tabular results of Selection Sort

10.1.2 Model Equation:

Tmodel(N) = N(N-1)/2

10.1.3 Graphical Representation:



Graphic representation of Selection Sort

Here, the x-axis represents the size of the array, and the Y-axis represents the number of comparisons done by the sorting algorithm.

10.2 Insertion Sort

10.2.1 Tablular results

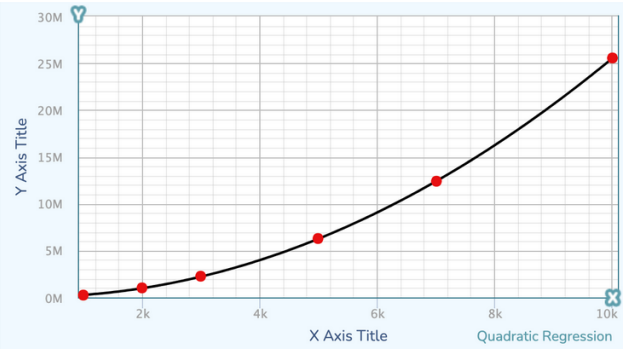
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	251473	1019425	2265121	6288313	12404111	25555527
T(N) Model	278256	999032	2241487	6291432	12428094	25545676

Tabular results of Insertion Sort

10.2.2 Model Equation

- Model = 79159.12 - 61.74223*N + 0.2608394*N*N

10.2.3 Graphic Representation



Graphical representation of Insertion Sort

10.3 Merge Sort

10.3.1 Tabular Results

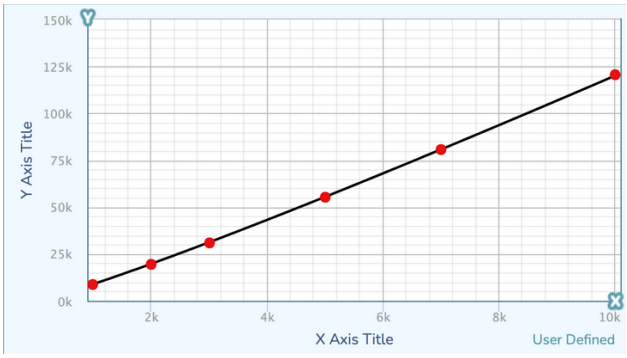
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	8720	19461	30913	55251	80666	120558
T(N) Model	9009	19826	31326	55541	80830	120123

Tabular results of Merge Sort

10.3.2 Model Equation

- Model = 0.9040187*N * log2(N)

10.3.3 Graphical Representation



Graphical representation of Merge Sort

10.4 Quick Sort: Pivot 1st Element

10.4.1 Tabular Results

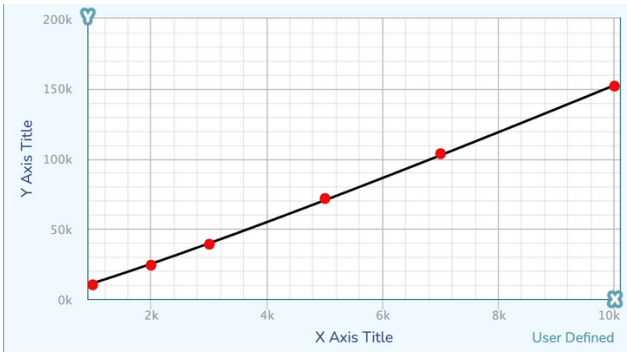
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	10094	24091	39034	71637	103601	151789
T(N) Model	11437	25170	39769	70511	102616	152500

Tabular results of Quick Sort when pivot is the first element in the array

10.4.2 Model Equation

- Model = 1.147683*N * log2(N)

10.4.3 Graphical Representation



Graphical representation Quick Sort when pivot is the first element in the array

10.5 Quick Sort: Pivot Random Element

10.5.1 Tabular Results

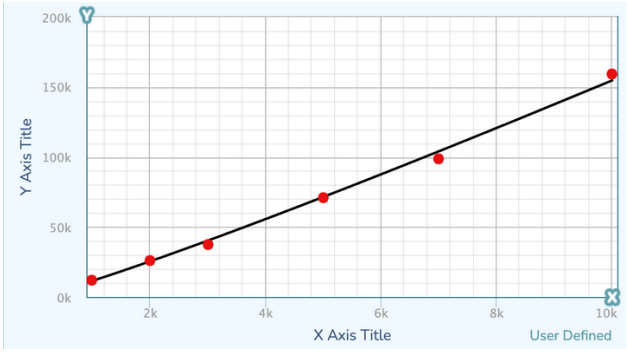
N	1000	2000	3000	5000	7000	10000
T(N) Experiment	11983	26015	37468	70951	98712	159289
T(N) Model	11604	25538	40351	71543	104118	154732

Tabular results of Quick Sort when pivot is a random element in the array

10.5.2 Model Equation

- Model = 1.164476*N * log2(N)

10.5.3 Graphical Representation



Graphical representation Quick Sort when pivot is a random element in the array

11 Analysis and Critique

As we can see, for the insertion and selection sort algorithms, they both have quadratic graphs; whereas for the Merge and quick sort algorithms, the graph represents more of N log (n), which is a slightly linear graph.
Based on experimental results, different sorting algorithms displayed different numbers of comparisons and efficiency overall. It was anticipated that Mergesort would show off as the most efficient sorting algorithm of all. Our anticipation is then verified by the experimental data, showing that the merge sort algorithm was the most effective algorithm in limiting the number of comparisons from N=1000 to N=10000. Furthermore, we will rank the quicksort algorithm as the second most efficient algorithm according to the experimental data. Then, insertion sort as the third most effective algorithm, and the selection sort as the least effective sorting algorithm. This would agree with our average case scenarios. However, please note that we haven't tested the space complexity, since merge sort algorithm may take more space than any other algorithm. What our experimental data tends to focus on was time complexity, ignoring stability, space complexity, and many others. Our experimental data showed that merge sort had the best time complexity of all the sorting algorithms, but not the stability or space complexity. So, we can't really state whether Merge sort was the most "effective" algorithm of all of these sorting algorithms. For example, Merge sort may have a better time complexity than quick sort, theoretically speaking, quicksort also has a better space complexity than merge sort. So, we can't really state who's the best sorting algorithm. Furthermore, some other limitations of our experiment are that we couldn't measure the amount of time that takes for each sorting algorithm to execute successfully. However, from our experimental data, we can conclude that the Merge sort algorithm has the best efficiency in limiting the number of comparisons, thus making it the most efficient sorting algorithm of them all. Additionally, some of research is needed to know what happens to the number of comparisons if we had a partially sorted array. In general, the divide-and-conquer algorithms have proven to be the most effective algorithms in limiting the number of comparisons between all of the sorting algorithms.

12 Conclusion

From the results of our comparative study, they have shown that the sorting algorithms may act a bit different under different sizes. We used tabular and graphical results to help visualize the performance of each sorting algorithm. This paper also discussed some of our limitations to our experimental results. Additionally, there are many characteristics that distinguish between the usage of several sorting algorithms. Due to our team's time and hardware constraints, we couldn't dive deep into this problem, and we decided that we could conclude this experiment by saying that the merge sort has proven to be the most efficient algorithm in limiting the number of comparisons overall. Also, our team believes that our experimental data is very reliable because we have repeated the run several times, and the almost the same number of comparisons are made by each sorting algorithm under different sizes. Furthermore, we believe that researchers shouldn't pick on any sorting algorithm just because it has the least number of comparisons. Our research has a lot of constraints, and it cannot be depended on as a source of information for preferring any sorting algorithm over the other. Further research is required to understand what research algorithm is the best for space complexities, data structures, applications, and for other big sizes.

13 Acknowledgements

We would love to take some time and thank Prof. Amr Goneid for his constant enlightenment and support. He is an extremely professional man, and his stories and life experiences move us. We are glad to call him an idol. We would like to offer thanks to Eng. Mohammed Hany, who guided us step-by-step in both the paper and the project.

14 References

[1] "Sorting algorithms," GeeksforGeeks, https://www.geeksforgeeks.org/sorting-algorithms/ (accessed May 22, 2023).

[2] "Selection sort – data structure and algorithm tutorials," GeeksforGeeks, https://www.geeksforgeeks.org/selection-sort/ (accessed May 22, 2023).

[3] "Insertion sort - data structure and algorithm tutorials," GeeksforGeeks, https://www.geeksforgeeks.org/insertion-sort/ (accessed May 22, 2023).

[4] "Merge sort algorithm," GeeksforGeeks, https://www.geeksforgeeks.org/merge-sort/ (accessed May 22, 2023).

[5] "Quicksort," GeeksforGeeks, https://www.geeksforgeeks.org/quick-sort/ (accessed May 22, 2023).

15 APPENDIX A

15.1 Helper Functions Code

```
class helperFunctions{
public:
   void Swap(int &x,int &y)
   {
       int Temp=x;
       x=y;
       y=Temp;
   }
   void Print(int *a,int n)
   {
       for(int i=0;i<n;i++)
       {
           cout<<a[i]<<endl;
       }
   }
   void RPA(int *a,int n)
   {
       for(int i=0;i<n;i++)
       {
           a[i]=i;
       }
       srand(time(0));
       for(int i=2;i<n;i++)
       {
           int m=  rand() % i + 1;
           Swap(a[i],a[m]);
       }
   }
};
```

In the figure above, it is a snippet for the class that contains the helper functions. The helper function class contains the swap and the RPA functions as public data member. Then we can use this class to create an object to be utilized in the upcoming sorting classes.

15.2 Selection Sort Code

```
int SelectionSort(int arr[],int n)
{
   int count=0;
   helperFunctions H;
   for(int i=1;i<n-1;i++)
   {
       int x=i;
       for(int j=i+1;j<n;j++)
       {
           if(arr[j]<arr[x])
           {
               x=j;
           }
           count++;
       }
       H.Swap(arr[i],arr[x]);
   }
   return count;
}
```

Selection Sort Code

15.3 Insertion Sort Code

```
void insertion(int *a, int size, int& count){
   int i, j, v;
   for (i = 1; i < size; i++)
   {
       v = a[i];
       j = i ;
       while (j > 0 && (count++,a[j-1] > v))
       {
           a[j] = a[j-1];
           j--;
       }
       a[j] = v;
   }
}
```

Insertion Sort Code

15.4 Merge Sort Code

```
class MergeSort{
public:
   void Merge(int *Arr,int left, int mid, int right,int &Counter)
   {
       int i, j, k;int n1 = mid - left + 1;int n2 = right - mid;
       int L[n1], R[n2];
       for (i = 0; i < n1; i++)
           L[i] = Arr[left + i];
       for (j = 0; j < n2; j++)
           R[j] = Arr[mid + 1 + j];
       i = 0;j = 0;k = left;
       while (i < n1 && j < n2) {
           Counter++;
           if (L[i] <= R[j]) {
               Arr[k] = L[i];
               i++;
           }
           else {
               Arr[k] = R[j];
               j++;
           }
           k++;
       }
       while (i < n1) {
           Arr[k] = L[i];i++;k++;
       }
       while (j < n2) {
           Arr[k] = R[j];j++;k++;
       }
   }
   void mergeSort(int *Arr,int left,int right,int &Counter)
   {
       if (left < right)
       {
           int mid = left + (right - left) / 2;
           mergeSort(Arr,left, mid,Counter);
           mergeSort(Arr,mid + 1, right,Counter);
           Merge(Arr,left, mid, right,Counter);
       }
   }
};
```

Implementation of the Merge Sort class including the following member functions, Merge and mergeSort

15.5 Quick Sort Code

```
class QuickSort{
private:
   helperFunctions H;
   int partition(int arr[], int low, int high,int &Counter)
   {
       int pivot = arr[low];
       int k = high;
       for (int i = high; i > low; i--) {
           if (arr[i] > pivot)
           {
               H.Swap(arr[i], arr[k--]);
           }
           Counter++;
       }
       H.Swap(arr[low], arr[k]);
       return k;
   }
public:
   void quickSort(int arr[], int low, int high,int &Counter)
   {
       if (low < high) {
           int i = partition(arr, low, high,Counter);
           quickSort(arr, low, i - 1,Counter);
           quickSort(arr, i + 1, high,Counter);
       }
   }
   void rQuickSort(int *arr,int low,int high,int &Counter)
   {
       if(low<high)
       {
           if((high-low)>5)
           {
               int k=rand()%(high-low+1)+low;
               H.Swap(arr[low],arr[k]);
           }
           int i = partition(arr, low, high,Counter);
           rQuickSort(arr, low, i-1,Counter);
           rQuickSort(arr, i+1, high,Counter);
       }
   }
};
```

Implementation of the Quick Sort class including the following member functions, partition which is responsible for the pivot index and quickSort which sorts the array when the pivot is the first element, rQuickSort which sorts the array when the pivot is a random element

15.6 Load Save Functions

```
void Save(string Txt,int n)
{
    helperFunctions H;
    ofstream MyFile;
    MyFile.open(Txt);
    int *Arr=new int[n];
    H.RPA(Arr,n);
    for(int i=0;i<n;i++)
    {
        MyFile<<Arr[i]<<endl;
    }
    MyFile.close();
    delete [] Arr;
}
void Load(int *Arr,string Txt,int n)
{
    ifstream MyFile;
    MyFile.open(Txt);
    for(int i=0;i<n;i++)
    {
        MyFile>>Arr[i];
    }
    MyFile.close();
}
```

Save function is used to save an array to a text file, while load function loads an array from a text file. We have used those functions so that we have unified arrays for all experiments

15.7 Sample Results Generation Function (Selection Sort)

```
void DisplayExpSelection()
{
  cout<<"---------**SELECTION SORT**---------"<<endl;
  helperFunctions H;
  int N_arr[] = {1001, 2001, 3001, 5001, 7001, 10001};
  cout << "Array size\tT-exp(N)" << endl;
  for (int i = 0; i < 6; i++)
  {
     int count_exp=0;  int N = N_arr[i];
     int *Arr=new int[N];
     if(N==1001)
     {
        H.Load(Arr,"1000Arr.txt",1001);
     }
     else if(N==2001)
     {
        H.Load(Arr,"2000Arr.txt",2001);
     }
     else if(N==3001)
     {
        H.Load(Arr,"3000Arr.txt",3001);
     }
     else if(N==5001)
     {
        H.Load(Arr,"5000Arr.txt",5001);
     }
     else if(N==7001)
     {
        H.Load(Arr,"7000Arr.txt",7001);
     }
     else if(N==10001)
     {
        H.Load(Arr,"10000Arr.txt",10001);
     }
     count_exp=SelectionSort(Arr,N);
     cout << N << "\t\t" << count_exp << endl;
     delete [] Arr;
  }
```

User-defined function to generate the results automatically

15.8 Sample Main Function

```
int main() {
    DisplayExpSelection();
    DisplayExpInsertion();
    DisplayExpMerge();
    DisplayExpQuick();
    DisplayExpRQuick();
}
```

Sample calling for the main function