Impact of Sorting Algorithms on Search Efficiency

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

Bachelor of Technology In

Computer Science and Engineering

School of Engineering and Sciences

Submitted by

Candidate Name

DNV LIKHITHA CHITTINEEDI

AADARSH SENAPATI

PRAVEEN KUMAR KOMMANABOYINA

(AP23110010469)

(AP23110010458)

(AP23110010460)

Under the Guidance of

Ms. Roopa Tirumalasetti

SRM University–AP

Neerukonda, Mangalagiri, Guntur

Andhra Pradesh – 522 240

[November,2024]

Certificate

Date: 20-Nov-24

This is to certify that the work present in this Project entitled "**Impact of Sorting Algorithms on Search Efficiency**" has been carried out by **DNV Likhitha**(AP23110010469) , **Aadarsh**(AP23110010458) , **Praveen**(AP23110010460) under our supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

Supervisor

(Signature)

Prof. / Dr. [Name]

Designation,

Affiliation.

Co-supervisor

(Signature)

Prof. / Dr. [Name]

Designation,

Affiliation.

Acknowledgements

We would like to thank everyone who contributed to this study on sorting and search algorithms. We are grateful to our academic guides and mentors for their valuable insights and support, as well as the programming community for providing helpful resources that enriched our understanding. Special thanks go to open-source contributors for creating tools and libraries that made our coding and testing more efficient. We also appreciate the feedback from colleagues and friends, which helped improve the quality of our work. Their support and guidance have been crucial in completing this project successfully.

Special thanks to all who helped take this project to success.

Table of Contents

Abstract

The performance of search operations, especially in large datasets, is crucial in applications like database indexing and e-commerce platforms. This report analyzes how sorting algorithms, particularly Merge Sort and Quick Sort, influence the efficiency of Binary Search under three scenarios:

(1) elements sorted in ascending order.

(2) elements sorted in descending order.

(3) elements in random order.

The analysis highlights the time taken for sorting, the impact on search speed, and practical implications in real-world applications.

Statement of Contributions

Aadarsh: Led the implementation of Merge Sort and Quick Sort algorithms, conducted experimental tests, and recorded the time complexity results. Aadarsh also worked on creating the tables and summarizing the data for better clarity.

DNV Likhitha: Focused on the theoretical research and writing the introductory and concluding sections of the report. Likhitha also contributed to analysing the performance data of the algorithms and compiling the overall findings.

Praveen: Managed the Binary Search implementation, evaluated search performance across different scenarios, and helped in the discussion and analysis section. Praveen also ensured that the final report was well-structured and cohesive.

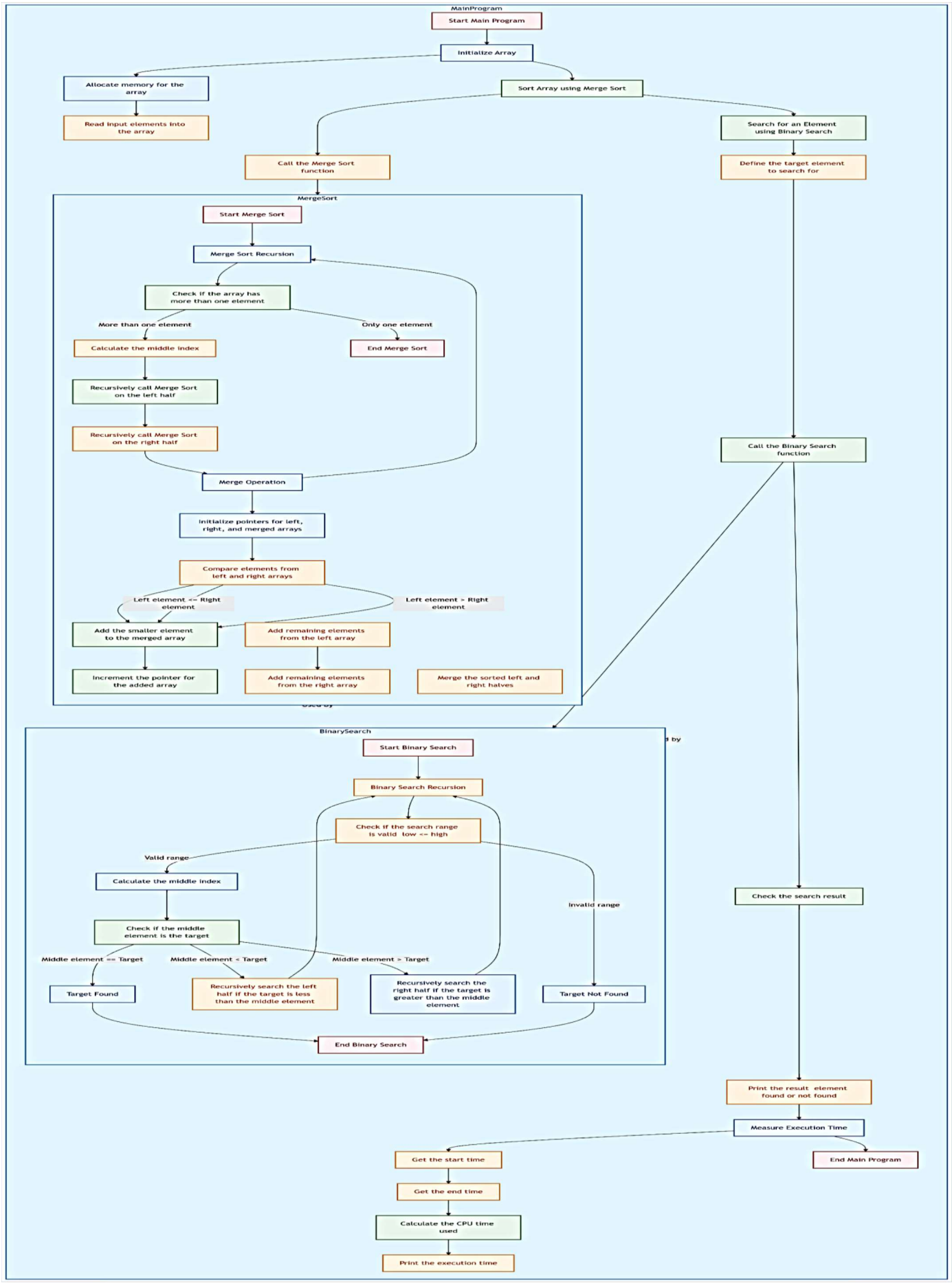
List of Tables

Algorithm	Best Case	Average Case	Worst Case
Merge Sort	O(N logN)	O(N logN)	O(N logN)
Quick Sort	O(N logN)	O(N logN)	$O(N^2)$
Binary search	O(N logN)	O(N logN)	O(N logN)

	Ascending Order	Descending Order	Random Order
Merge sort time	5.705000 s	4.563000 s	3.739000 s
Quick sort time	4.167000 s	3.616000 s	3.982000 s
Merge sort with binary search time	4.650000 s	3.833000 s	3.415000 s
Quick sort with binary search time	5.036000 s	3.559000 s	2.359000 s

List of figures

Flowchart:


MainProgram
Start Main Program
Initialize Array
Allocate memory for the array
Read input elements into the array
Sort Array using Merge Sort
Call the Merge Sort function
Search for an Element using Binary Search
Define the target element to search for
MergeSort
Start Merge Sort
Merge Sort Recursion
Check if the array has more than one element
More than one element
Only one element
Calculate the middle index
End Merge Sort
Recursively call Merge Sort on the left half
Recursively call Merge Sort on the right half
Merge Operation
Initialize pointers for left, right, and merged arrays
Compare elements from left and right arrays
Left element <= Right element
Left element > Right element
Add the smaller element to the merged array
Add remaining elements from the left array
Increment the pointer for the added array
Add remaining elements from the right array
Merge the sorted left and right halves
Call the Binary Search function
BinarySearch
Start Binary Search
Binary Search Recursion
Check if the search range is valid low <= high
Valid range
Invalid range
Calculate the middle index
Check if the middle element is the target
Middle element == Target
Middle element < Target
Middle element > Target
Target Found
Recursively search the left half if the target is less than the middle element
Recursively search the right half if the target is greater than the middle element
Target Not Found
End Binary Search
Check the search result
Print the result element found or not found
Measure Execution Time
End Main Program
Get the start time
Get the end time
Calculate the CPU time used
Print the execution time

List of equation

Master Theorem Formula

$T(n) = a\ T(n/b) + f(n)$

Merge Sort Time Complexity Analysis

1. **Recurrence Relation**: $T(n)=2T(n/2) +O(n)$
2. **Master's Theorem Parameters**:

    - $a = 2$
    - $b = 2$
    - $f(n) = O(n)$
    - $\log_b a = \log_2 2 = 1$

3. **Comparison**:

    - $f(n) = \Theta(n^1)$ and $\log_b a = 1$

4. **Result**: $T(n) = O (n \log n)$

Quick Sort Time Complexity Analysis (Average Case)

1. **Recurrence Relation**: $T(n) = 2T(n/2) +O(n)$
2. **Master's Theorem Parameters**:

    - $a=2$
    - $b=2$
    - $f(n)=O(n)$
    - $\log_b a=\log_2 2=1$

3. **Comparison**:

    - $f(n)=\Theta(n^1)$ and $\log_b a=1$

4. **Result**: $T(n)=O(n \log n)$

1. Introduction

Retrieving data in an efficient manner is important in many applications-one of these is databases and the e-commerce servers where fast access to information can be the difference between success or failure. There's one optimization method simply to sort data before searching; a better alternative is Binary Search that is known for its speed when used against sorted data. This chapter will explore how two of the most prevalent sorting techniques, Merge Sort and Quick Sort, prepare data for Binary Search, so we can begin to understand the role that which sorting plays in possibly optimizing search times.

Merge Sort has its strengths: it is consistent and efficient, while Quick Sort is usually faster, with a possible problem that it can have with specific data patterns. These sorting techniques don't exist in an abstract world; these effects are real and noticeable as they reflect into designing databases, optimizing search functions, and managing system performance. Forging deeper into the impacts of these methods can help make smarter decisions about how to deal with information so that systems smoothly execute, even when dealing with masses of information. The study uncovered such trade-offs between sorting techniques and offered practical insights into just which strategies work best for specific needs.

2. Methodology

We focused on using two well-known sorting algorithms, **Merge Sort** and **Quick Sort**, to organize a dataset of 100 integers. Our aim was to see how sorting affects the speed of searching using **Binary Search**, which is a very fast method but only works well on sorted data. By comparing these algorithms in different situations, we learned more about how sorting helps make searching efficient.

Sorting Algorithms

- **Merge Sort**: This is a reliable and consistent way to sort data, always working efficiently with a time complexity of O(N log N). It splits the data into smaller parts, sorts them, and then merges them back together in order. Merge Sort's performance stays steady no matter how the data is arranged.
- **Quick Sort**: Known for being faster than Merge Sort in most cases, Quick Sort sorts data by picking a pivot and dividing the numbers into smaller groups. On average, it is efficient with O(N log N) complexity, but if the data is arranged poorly, it can slow down to $O(N^2)$. Still, it's often used because it's generally fast.

Binary Search

We used **Binary Search** after sorting the data to measure how efficiently we could find elements. Binary Search is very fast, with a time complexity of O(log N), but it only works if the data is already in order. This means that sorting is a crucial first step for making searches run smoothly.

Scenarios We Tested

1. **Ascending Order**: We sorted the data in ascending order. This setup is ideal for Binary Search, giving us the quickest search times because the data is already perfectly organized.
2. **Descending Order**: We first arranged the data in descending order, and then had to reverse it to make it searchable. This added some extra steps, which slightly affected how fast the sorting was completed.
3. **Random Order**: Here, the data was jumbled without any pattern. Both Merge Sort and Quick Sort had to work harder to organize this data before Binary Search could be used. This situation showed clear differences in how the sorting algorithms performed, as they had to deal with a completely unstructured set of numbers.

3. Discussion

Our project showed us that sorting the data before searching has a huge impact on how fast we can find what we are seeking. Let's break what we found down.

Sorting Makes Search Faster

It was pretty clear pretty fast that sorted data was the thing which made all the difference in terms of speed of search. Binary search is an exceptionally good way to find an item in a list but only works given everything is in the proper order. Thus, not sorting would mean strictly not feasible Binary search. Truly, it really shows how much need for sorting is to make operations involved in the search as fast and efficient as they can be.

Merge Sort vs. Quick Sort

We observed some interesting things when we compared Merge Sort and Quick Sort. Merge Sort is one of those reliable old friends: steady, efficient, no matter how messy or organized data might be to start with. It's slower than other algorithms sometimes, but you can always count on it to always do its job in an efficient manner. Quick Sort, on the other hand, is a bit of a risk-taker. It's often faster, sorts data very quickly, but on rare occasions where it gets very unlucky by the random way the data was arranged (like the numbers were already in some sort of pattern), it really slows down. While Quick Sort is usually the better faster choice, it's also less predictable than Merge Sort.

Some real time scenarios

- **E-commerce Platforms**: Websites like Amazon or eBay need to sort millions of products based on price, popularity, or relevance. When a user searches for a product, having the items already sorted allows the platform to quickly show results, making the shopping experience faster and smoother.
- **Database Management**: In large organizations, databases contain vast amounts of information, such as employee records or customer details. Sorting this data in advance enables faster searching, which is crucial when the system needs to retrieve information quickly for operations or customer queries.

- **Search Engines**: Search engines like Google use complex sorting and searching mechanisms to rank and display web pages. Sorting helps prioritize which pages are shown first, ensuring users get the most relevant information without delay.
- **Financial Applications**: In stock trading platforms, where prices and transactions update rapidly, sorting algorithms help organize this information for quick searches and analysis. This efficiency is vital for making real-time financial decisions.
- **Healthcare Data**: Hospitals use sorted medical records to quickly access patient information. For instance, if a doctor needs to find a specific patient's history among thousands of records, efficient searching enabled by sorting saves valuable time in critical situations.

4. Concluding Remarks

Sorting algorithms have a significant impact on search efficiency. Both Merge Sort and Quick Sort help prepare data for Binary Search, reducing search time but adding an initial overhead due to the sorting step. The choice between Merge Sort and Quick Sort depends on factors like data size and distribution. Merge Sort is reliable and consistent, while Quick Sort tends to be faster but may degrade with poorly arranged data. Efficient sorting followed by Binary Search is essential for applications like online platforms and database systems, where quick data access is crucial. As data grows, optimizing sorting and search operations becomes even more important. In the future, hybrid or improved sorting methods could enhance performance, providing faster retrieval times and better system efficiency for large datasets.

5. Future Work

- **Algorithm Optimization**: Explore hybrid algorithms that adapt to data characteristics, reducing worst-case scenarios.

- **Parallel Processing**: Investigate parallel versions of sorting algorithms for faster execution in multi-core systems.

- **Application-Specific Studies**: Examine sorting and search efficiencies in real-world databases and e-commerce systems.

6. References

- https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/
- https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-quick-sort/
- https://www.w3schools.com/dsa/dsa_algo_binarysearch.php
- https://hasty.dev/blog/sorting/merge-vs-quick

Appendix

A. Sample Code:

- **Merge Sort with Binary search:**

```
#include <stdio.h>
#include<time.h>
#define N 100
void merge(int arr[], int l, int m, int r){
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        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 l, int r){
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
```

```
}
int binarySearch(int arr[], int low, int high, int x){
    if (high >= low) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, low, mid - 1, x);
        return binarySearch(arr, mid + 1, high, x);
    }
    return -1;
}
int main(){
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    int a[N];
    for(int i=0;i<N;i++){
        scanf("%d",&a[i]);
    }
    mergeSort(a, 0, N-1);
    int x = 1576858;
    int result = binarySearch(a, 0, N-1, x);
    if (result == -1) printf("Element is not present in array");
    else printf("Element is present at index %d", result);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\nTime: ");
    printf("%lf",cpu_time_used);
}
```

- **Quick Sort with Binary search:**

```
#include <stdio.h>
#include<time.h>
#define N 100
int split(int a[],int lower, int upper){
int i,p, q,t;
    p = lower +1;
    q = upper ;
    i = a[lower] ;
    while (q>=p){
        while ( a[p] <i)
            p++;

        while (a[q] > i)
            q--;

        if(q>p){
```

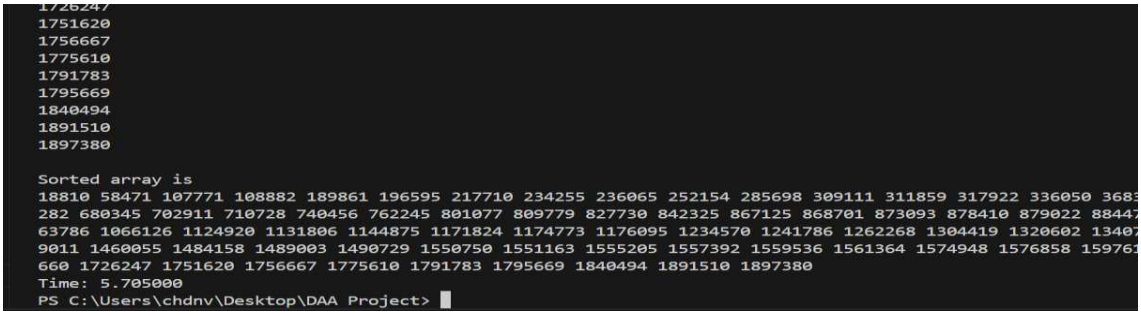
```
            t = a[p];
            a[p] = a[q];
            a[q] = t;
        }
    }
        t = a[lower];
        a[lower] = a[q];
        a[q]=t;
        return q;
}
void quicksort (int a[], int lower, int upper ){
    int i;
    if ( upper > lower ){
        i = split ( a, lower, upper ) ;
        quicksort ( a, lower, i-1);
        quicksort (a, i+ 1, upper) ;
    }
}
int binarySearch(int arr[], int low, int high, int x)
{
    if (high >= low) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, low, mid - 1, x);
        return binarySearch(arr, mid + 1, high, x);
    }
    return -1;
}
int main(){
    clock_t start, end;
    double cpu_time_used;
    start = clock();
    int a[N];
    for(int i=0;i<N;i++){
        scanf("%d",&a[i]);
    }
    quicksort(a,0,N-1);
    int x = 1576858;
    int result = binarySearch(a, 0, N-1, x);
    if (result == -1) printf("Element is not present in array");
    else printf("Element is present at index %d", result);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\nTime: ");
    printf("%lf",cpu_time_used);
}
```

B. Sample Screenshots:

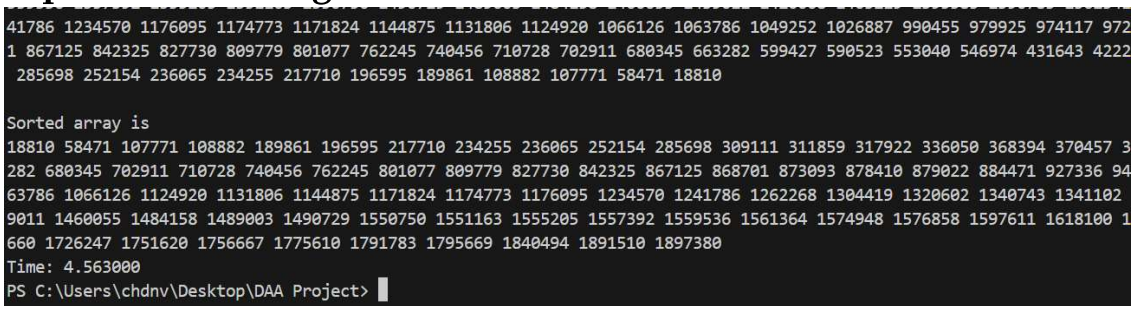
Complexity Comparisons

1) Merge Sort:

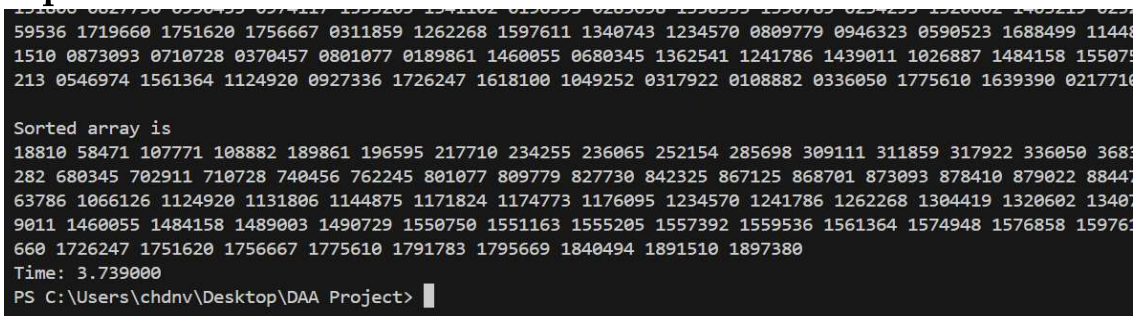
a) Input – ascending order



b) Input – descending order

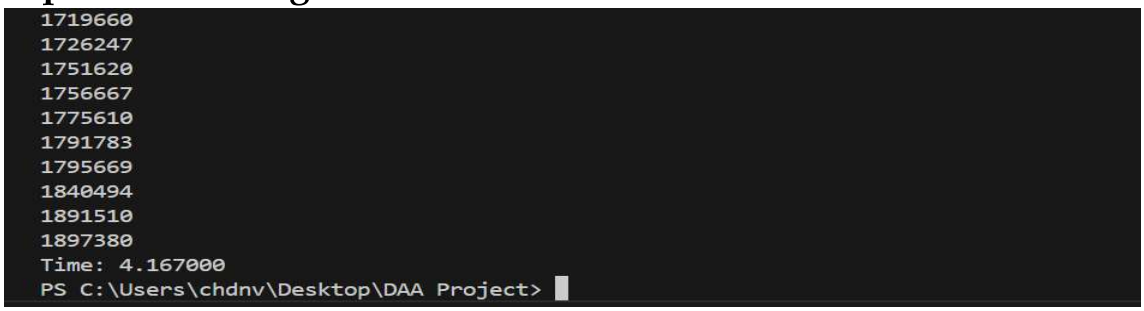


c) Input – random order



2) Quick Sort:

a) Input – ascending order



b) Input – descending order

```
1726247
1751620
1756667
1775610
1791783
1795669
1840494
1891510
1897380
Time: 3.616000
PS C:\Users\chdnv\Desktop\DAA Project>
```

c) Input – random order

```
1791783
1795669
1840494
1891510
1897380
Time: 3.982000
PS C:\Users\chdnv\Desktop\DAA Project>
```

3) Merge sort with binary search

a) Input – ascending order

```
1751620
1756667
1775610
1791783
1795669
1840494
1891510
1897380
Element is present at index 81
Time: 4.659000
PS C:\Users\chdnv\Desktop\DAA Project>
```

b) Input – descending order

```
1751620
1756667
1775610
1791783
1795669
1840494
1891510
1897380
Time: 3.833000
PS C:\Users\chdnv\Desktop\DAA Project> \
```

c) Input – random order

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS

PS C:\Users\chdnv\Desktop\DAA Project> cd "c:\Users\chdnv\Desktop\DAA Project\" ; if ($?) { gcc MergeSort_binary.c

0431643 0879022 1576858 1557392 1840494 0740456 1420068 1489003 0309111 0878410 1174773 0599427 0762245 0368394 086
131806 0827730 0990455 0974117 1555205 1341102 0196595 0285698 1358335 1390785 0234255 1320602 1403219 0252154 1897
59536 1719660 1751620 1756667 0311859 1262268 1597611 1340743 1234570 0809779 0946323 0590523 1688499 1144875 06632
1510 0873093 0710728 0370457 0801077 0189861 1460055 0680345 1362541 1241786 1439011 1026887 1484158 1550750 165391
213 0546974 1561364 1124920 0927336 1726247 1618100 1049252 0317922 0108882 0336050 1775610 1639390 0217710 0842325
Element is present at index 81
Time: 3.415000
PS C:\Users\chdnv\Desktop\DAA Project>
```

4) Quick sort with binary search

a) Input – ascending order

```
1751620
1756667
1775610
1791783
1795669
1840494
1891510
1897380
Element is present at index 81
Time: 5.036000
PS C:\Users\chdnv\Desktop\DAA Project>
```

b) Input – descending order

```
559536 1557392 1555205 1551163 1550750 1490729 1489003 1484158 1460055 1439011 1420068 14032
41786 1234570 1176095 1174773 1171824 1144875 1131806 1124920 1066126 1063786 1049252 102688
1 867125 842325 827730 809779 801077 762245 740456 710728 702911 680345 663282 599427 590523
 285698 252154 236065 234255 217710 196595 189861 108882 107771 58471 18810
Element is present at index 81
Time: 3.559000
PS C:\Users\chdnv\Desktop\DAA Project>
```

c) Input – random order

```
0431643 0879022 1576858 1557392 1840494 0740456 1420068 1489003 0309111 0878410 1174773 0599427
131806 0827730 0990455 0974117 1555205 1341102 0196595 0285698 1358335 1390785 0234255 1320602 1
59536 1719660 1751620 1756667 0311859 1262268 1597611 1340743 1234570 0809779 0946323 0590523 16
1510 0873093 0710728 0370457 0801077 0189861 1460055 0680345 1362541 1241786 1439011 1026887 148
213 0546974 1561364 1124920 0927336 1726247 1618100 1049252 0317922 0108882 0336050 1775610 1639
Element is present at index 81
Time: 2.359000
PS C:\Users\chdnv\Desktop\DAA Project>
```