Analysis of Algorithms

BLG 335E

Project 1 Report

SELİN YILMAZ
yilmazsel21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 03.11.2024

1. Implementation

1.1. Sorting Strategies for Large Datasets

Apply ascending search with the algorithms you implemented on the data expressed in the header rows of Tables 1.1 and 1.2. Provide the execution time in the related cells. Make sure to provide the unit.

	tweets	**tweetsSA**	**tweetsSD**	**tweetsNS**
Bubble Sort	15.3065 s	6.34328 s	17.3589 s	16.2309 s
Insertion Sort	4.54946 s	0.000898352 s	9.17144 s	0.235906 s
Merge Sort	0.0346809 s	0.0354697 s	0.0390542 s	0.0321634 s

Table 1.1: Comparison of different sorting algorithms on input data (Same Size, Different Permutations).

	5K	**10K**	**20K**	**30K**	**50K**
Bubble Sort	0.197371 s	0.75615 s	3.17296 s	6.64746 s	18.6841 s
Insertion Sort	0.0697101 s	0.221866 s	0.78702 s	1.7434 s	4.62295 s
Merge Sort	0.00527559 s	0.00871594 s	0.0152564 s	0.0197351 s	0.0376762 s

Table 1.2: Comparison of different sorting algorithms on input data (Different Size).

Discuss your results

In Table 1.1, the performance of each sorting algorithm (bubble sort, insertion sort, and merge sort) can be seen based on different arrangements of data in four datasets:

- **Bubble Sort:** This algorithm shows the strongest response to the initial ordering of data.
  - **tweets:** This is randomly sorted. Bubble sort follows its typical complexity.
  - **tweetsSA:** This is sorted in ascending order by retweet count. Bubble sort performs best. Because it requires minimal operations.
  - **tweetsSD:** This is sorted in descending order. Bubble sort hits its **worst-case scenario** as it needs to make the maximum number of comparisons and swaps.
  - **tweetsNS:** This is nearly sorted in ascending order. Bubble sort also performs relatively well.
- **Insertion Sort:** Insertion sort also benefits from initial order.
  - **tweets:** Insertion sort follows its average complexity.

  - **tweetsSA:** Insertion sort hits its **best-case scenario** as it reqires minimal element shifts.
  - **tweetsSD:** Like bubble sort, insertion sort encounters its worst case.
  - **tweetsNS:** Insertion sort performs efficiently due to the dataset's mostly sorted nature.

- **Merge Sort:** Unlike bubble sort and insertion sort, merge sort performs consistently for all arrangements. It maintains the $O(n \log n)$ complexity across all cases.

In Table 1.2, the performance of each sorting algorithm can be seen on different data sizes.

- **Bubble Sort:** Its runtime increases significantly with data size due to its $O(n^2)$ time complexity. This table shows bubble sort's inefficiency with large datasets, making it impractical for sizable inputs.

- **Insertion Sort:** This also has a $O(n^2)$ time complexity in the average and worst cases but generally performs faster than bubble sort on small to moderately sized datasets. Although insertion sort remains inefficient for large datasets, it can be effective for smaller or nearly inserted inputs.

- **Merge Sort:** With its $O(n \log n)$ complexity, merge sort scales efficiently, showing minimal growth in runtime as data size increases. The consistent performance makes merge sort ideal for handling lager inputs.

1.2. Targeted Searches and Key Metrics

Run a binary search for the index 1773335. Search for the number of tweets with more than 250 favorites on the datasets given in the header row of Table 1.3. Provide the execution time in the related cells. Make sure to provide the unit.

	5K	**10K**	**20K**	**30K**	**50K**
Binary Search	3 μs	3.401 μs	1.5 μs	1.8 μs	2.2 μs
Threshold	32.802 μs	82.307 μs	148.911 μs	142.811 μs	236.819 μs

Table 1.3: Comparison of different metric algorithms on input data (Different Size).

Discuss your results

In Table 1.3, the performance of two algorithms (binary search and threshold) can be seen based on datasets with varying input sizes:

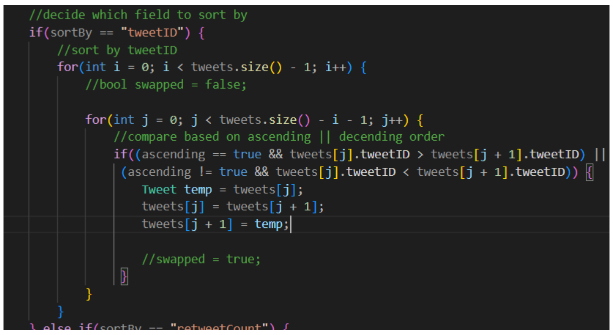
- **Binary Search:** The runtime of binary search remains consistently small for all data sizes. This consistency is expected because of binary search having a time complexity of $O(\log n)$.
- **Threshold Algorithm:** It can be seen that, the runtime of the threshold algorithm increases with data size, from 32.8 μs for 5000 input to 236.8 μs for 50000 input.

Overall, these results indicate that binary search consistently shows high performance for all data sizes, making it well-suited for applications with high search requirements. The threshold algorithm, in contrast, shows increasing runtime with data size.

1.3. Discussion Questions

Discuss the methods you've implemented and the complexity of those methods.

1. **Bubble Sort:** First, there are three *if statements* for *sortBy* parameter: tweetID, retweetCount, favoriteCount. Inside of them looks similar to figure 1.1.



```
//decide which field to sort by
if(sortBy == "tweetID") {
    //sort by tweetID
    for(int i = 0; i < tweets.size() - 1; i++) {
        //bool swapped = false;

        for(int j = 0; j < tweets.size() - i - 1; j++) {
            //compare based on ascending || decending order
            if((ascending == true && tweets[j].tweetID > tweets[j + 1].tweetID) ||
             (ascending != true && tweets[j].tweetID < tweets[j + 1].tweetID)) {
                Tweet temp = tweets[j];
                tweets[j] = tweets[j + 1];
                tweets[j + 1] = temp;

                //swapped = true;
            }
        }
    }
} else if(sortBy == "retweetCount") {
```

Figure 1.1

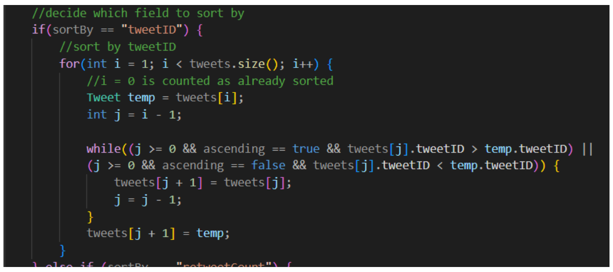
Which follows the logic:

Algorithm 1 Bubble Sort

```
for $i \leftarrow 1$ to $n - 1$ do
    for $j \leftarrow n - i - 1$ do
        if $A[j] > A[j + 1]$ then
            Exchange $A[j]$ and $A[j + 1]$
        end if
    end for
end for
```

The **best case** scenario is dataset is already sorted, bubble sort only needs one pass through the array to verify no swaps are needed. So the complexity is $O(n)$. In **average** and **worst case** scenarios, bubble sort needs to compare and swap adjacent elements. So the complexity is $O(n^2)$.

2. **Insertion Sort:** Also here, there are three *if statements* for *sortBy* parameter. Inside of them looks like figure 1.2.



```
//decide which field to sort by
if(sortBy == "tweetID") {
    //sort by tweetID
    for(int i = 1; i < tweets.size(); i++) {
        //i = 0 is counted as already sorted
        Tweet temp = tweets[i];
        int j = i - 1;

        while((j >= 0 && ascending == true && tweets[j].tweetID > temp.tweetID) ||
        (j >= 0 && ascending == false && tweets[j].tweetID < temp.tweetID)) {
            tweets[j + 1] = tweets[j];
            j = j - 1;
        }
        tweets[j + 1] = temp;
    }
} else if (sortBy == "retweetCount") {
```

Figure 1.2

Which follows the logic:

Algorithm 2 Insertion Sort

```
for $i \leftarrow 1$ to $n-1$ do do
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j >= 0$ and $A[j] > v$ do do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    end while
    $A[j+1] \leftarrow v$
end for
```

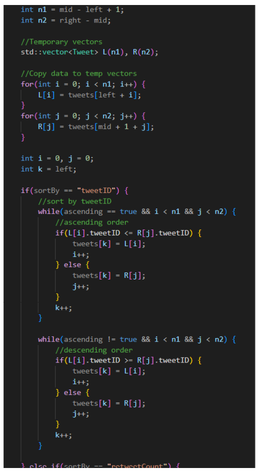
In the **best case** scenario, the array is already sorted so the complexity becomes $O(n)$. In the **average case**, elements are in random order. On average, each element will need to be compared with half of the already sorted elements. So the complexity is $O(n^2)$. The **worst case** scenario occurs when the array is sorted in reverse order. The complexity is $O(n^2)$.

3. **Merge:** Its pseudocode looks like:

Algorithm 3 Merge(A, p, q, r)

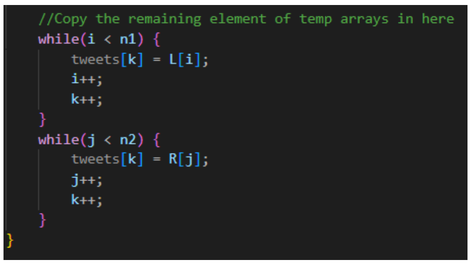
```
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i \leftarrow 1$ to $n_1$ do
    $L[i] \leftarrow A[p + i - 1]$
end for
for $j \leftarrow 1$ to $n_2$ do
    $R[j] \leftarrow A[q + j]$
end for
$i \leftarrow 1, j \leftarrow 1$
for $k \leftarrow p$ to $r$ do
    if $L[i] \leq R[j]$ then
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
    end if
end for
```

The implementation looks like figures 1.3 and 1.4.



```
int n1 = mid - left + 1;
int n2 = right - mid;

//Temporary vectors
std::vector<Tweet> L(n1), R(n2);

//Copy data to temp vectors
for(int i = 0; i < n1; i++) {
    L[i] = tweets[left + i];
}
for(int j = 0; j < n2; j++) {
    R[j] = tweets[mid + 1 + j];
}

int i = 0, j = 0;
int k = left;

if(sortBy == "tweetID") {
    //sort by tweetID
    while(ascending == true && i < n1 && j < n2) {
        //ascending order
        if(L[i].tweetID <= R[j].tweetID) {
            tweets[k] = L[i];
            i++;
        } else {
            tweets[k] = R[j];
            j++;
        }
        k++;
    }

    while(ascending != true && i < n1 && j < n2) {
        //descending order
        if(L[i].tweetID >= R[j].tweetID) {
            tweets[k] = L[i];
            i++;
        } else {
            tweets[k] = R[j];
            j++;
        }
        k++;
    }

} else if(sortBy == "retweetCount") {
```

Figure 1.3



```
    //Copy the remaining element of temp arrays in here
    while(i < n1) {
        tweets[k] = L[i];
        i++;
        k++;
    }
    while(j < n2) {
        tweets[k] = R[j];
        j++;
        k++;
    }
}
```

Figure 1.4

The merge function combines two already sorted subvectors into a single sorted vector. Since each element in the two subvectors is processed exactly once, the time complexity is $O(n)$ for all cases.

4. **Merge Sort:** This function divides the vector in half at each recursive call. So it has relatively smaller time complexity of $O(\log n)$. Its pseudocode (nearly same thing with implementation) looks like:

Algorithm 4 Merge-Sort(A, p, r)

```
if $p < r$ then
    $q \leftarrow [(p + r)/2]$
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
end if
```

What are the limitations of binary search? Under what conditions can it not be applied, and why?

1. First of the limitations is actually something I accidentally observed during the tests. Binary search needs the data to be sorted (in ascending or descending order) else it does not work properly.

2. It works only when elements in the dataset can be compared with each other (e.g integers).

3. It performs best on data structures whose elements has an index (e.g array, vector, list). In data structures like linked list, binary search becomes inefficient.

How does merge sort perform on edge cases, such as an already sorted dataset or a dataset where all tweet counts are the same? Is there any performance improvement or degradation in these cases?

Merge sort has a consistent complexity $O(n \log n)$. For already sorted dataset, merge sort maintains its time complexity. The algorithm divides the dataset into smaller sublists and then merges them back together. So there is neither performance improvement nor degradation.

Similarly, for a dataset where all elements are the same, merge sort also exhibits the same complexity.During the merging phase it will try to perform comparisons to determine the order, but since the all elements are same, there will not be any movement of elements. So there is also neither performance improvements nor degradation.

Were there any notable performance differences when sorting in ascending versus descending order? Why do you think this occurred or didn’t occur?

Bubble sort and insertion sort has a response to the initial order of the datasets which results in a performance difference for some cases: sorting a ascending permutation in ascending order versus descending order takes different times. So sorting in ascending versus descending order can make difference according to the given datasets. However, merge sort’s performance does not get effected by this.