Practice Report

Author: Andrii Voznesenskyi,
Course Instructor: Mikhail Tamashuk
Date: 24.08.2023

I. Introduction

SORTING algorithms have always been a cornerstone in computer science and play a pivotal role in many applications. Efficient sorting is crucial for optimizing downstream operations on data. In this report, we delve into a comparative analysis of three fundamental sorting algorithms: QuickSort, BubbleSort, and Merge Sort.

I-A. Objective

Our primary aim is to discern the efficiency of these algorithms relative to each other. Specifically, we intend to pinpoint the array length at which both QuickSort and Merge Sort begin to outperform BubbleSort consistently. It is noteworthy to mention that the implementation of these algorithms was done from scratch, ensuring a first-principles approach to this analysis.

I-B. Methodology

Our methodology for evaluating the performance of these algorithms hinges on the following procedures:

1. **Array Construction:** Lets take into the consideration the next definition $g : 2^{\mathbf{R}} \to \mathbf{R},\ g(A) = |A|$, is the size of the $A \in 2^{\mathbf{R}}$ and the $2^{\mathbf{R}}$ is the set of all the subsets of R. We harness three distinct types of arrays for the testing:
   - *Sorted Array:* An array A is said to be in ascending order if for every pair of indices i and j such that $0 \le i < j < |A|$, we have $A[i] \le A[j]$. Formally, this can be represented as:
   $$\forall i, j : 0 \le i < j < \text{length}(A) \Rightarrow A[i] \le A[j]$$
   - *Sorted Backward Array:* An array A is in descending order if for every pair of indices i and j such that $0 \le i < j < |A|$, we have $A[i] \ge A[j]$. Formally:
   $$\forall i, j : 0 \le i < j < |A| \Rightarrow A[i] \ge A[j]$$
   - *Random Array:* An array A where the position of elements doesn't follow the constraints of ascending or descending order. The sequence of elements is unpredictable.
2. Let the array length $|A|$ of the any array A described earlier be denoted by n. We start with $n = 2$ and increment n for subsequent tests. This progression can be captured by:
$$n \leftarrow n + 1$$
3. For each array type and length n, let $T_i : \mathbf{N} \to \mathbf{R},\ i \in \{QuickSort,\ BubbleSort,\ MergeSort\}$ be the function of the execution time, more precisely $T_{\text{QuickSort}}(n)$, $T_{\text{BubbleSort}}(n)$, and $T_{\text{MergeSort}}(n)$ represent the execution time of QuickSort, BubbleSort, and MergeSort, respectively.
4. Our objective is to find the smallest length n^* such that:
$$T_{\text{QuickSort}}(n^*) < T_{\text{BubbleSort}}(n^*)$$
and
$$T_{\text{MergeSort}}(n^*) < T_{\text{BubbleSort}}(n^*)$$
5. For reliability, let each sorting test for a specific n be performed k times, where k is a sufficiently large number. The average time for each sort on that n is then given by:
$$\bar{T}_{\text{QuickSort}}(n) = \frac{1}{k} \sum_{i=1}^{k} T_{\text{QuickSort},j}(n)$$
$$\bar{T}_{\text{BubbleSort}}(n) = \frac{1}{k} \sum_{i=1}^{k} T_{\text{BubbleSort},j}(n)$$
$$\bar{T}_{\text{MergeSort}}(n) = \frac{1}{k} \sum_{i=1}^{k} T_{\text{MergeSort},j}(n)$$
Where $T_{\text{QuickSort},j}(n)$, $T_{\text{BubbleSort},j}(n)$, and $T_{\text{MergeSort},j}(n)$ are the individual execution times for the j^{th} test.
6. After determining n^*, we extend our analysis to $n^* + m$ for some chosen $m > 0$. The objective is to analyze the growth behavior of the sorting algorithms. This could be described as analyzing the function behaviors of $\bar{T}_{\text{QuickSort}}(n)$, $\bar{T}_{\text{BubbleSort}}(n)$, and $\bar{T}_{\text{MergeSort}}(n)$ for $n > n^*$.

With the methodology firmly set, the following sections will unfurl the results and delve into a comprehensive analysis.

II. The approach of the Sorting Algorithms

In the realm of computer science, sorting algorithms are fundamental constructs that order elements in a specific sequence or manner. Each sorting algorithm has a unique design and approach that offers advantages in certain scenarios over others. In this section section we discuss the logic behind three classic sorting algorithms in this research: QuickSort, BubbleSort, and MergeSort, highlighting their complexities and the specific cases where they are most effective.

This article has a chance to see the world be the author's degree in the 2021-2025.

II-A. *QuickSort*

QuickSort is a divide-and-conquer sorting algorithm that selects an element, referred to as the ”pivot”, and partitions the array around the pivot. The algorithm then recursively applies the same logic to the sub-arrays. The implementation of the algorithm is represented in Listing 1,

Complexity: - Average Time Complexity: $O(n \log n)$ - Worst-Case Time Complexity: $O(n^2)$ - Space Complexity: $O(\log n)$

Effective Usage: QuickSort is usually faster in practice than other $O(n \log n)$ algorithms like MergeSort or HeapSort. It works especially well when the pivots chosen are median or near-median values. However, its worst-case time complexity can be a limitation with certain datasets or if the pivot selection strategy isn’t effective.

```
/**
 * Sorts an array using the QuickSort algorithm.
 *
 * @param A: The array to be sorted.
 * @param start: The starting index of the portion of A to be sorted.
 * @param end: The ending index of the portion of A to be sorted.
 */
Function QuickSort(A, start, end)
    if start < end
        pivotIndex ← Partition(A, start, end)
        QuickSort(A, start, pivotIndex - 1)
        QuickSort(A, pivotIndex + 1, end)
    end if

/**
 * Partitions an array around a pivot, such that elements less than the pivot are
 * on the left, and elements greater than the pivot are on the right.
 *
 * @param A: The array to be partitioned.
 * @param start: The starting index for the partition.
 * @param end: The ending index for the partition.
 * @return: Index of the pivot after partitioning.
 */
Function Partition(A, start, end)
    pivot ←  A[end]
    pivotIndex ←  start
    for i from start to end - 1
        if A[i] ≤  pivot
            swap A[i] and A[pivotIndex]
            pivotIndex ← pivotIndex + 1
        end if
    end for
    swap A[pivotIndex] and A[end]
    return pivotIndex
```

Listing 1: QuickSort and its Partition function

II-B. *Average Case Analysis*

The main value of the final complexity depends on the **Partition** function implementation. When QuickSort partitions the array evenly, the recurrence relation is given by:

$$T_{\text{QuickSort}}(n) = 2T_{\text{QuickSort}}\left(\frac{n}{2}\right) + n$$

Using the Master Theorem, this recurrence yields a time complexity of:

$$T_{\text{QuickSort}}(n) = O(n \log n)$$

II-C. *Worst Case Analysis*

For the worst-case scenario, if the pivot is always the smallest or largest element, the partitioning will be highly unbalanced. The recurrence relation becomes:

$$T_{\text{QuickSort}}(n) = T_{\text{QuickSort}}(n-1) + n$$

Expanding this recurrence:

$$\begin{aligned} T_{\text{QuickSort}}(n) &= n + T_{\text{QuickSort}}(n-1) \\ &= n + (n-1) + T(n-2) \\ &= n + (n-1) + (n-2) + \cdots + 2 + 1 \\ &= \sum_{i=1}^{n} i \\ &= \frac{n(n+1)}{2} \\ &= \frac{n^2}{2} + \frac{n}{2} \end{aligned}$$

Thus, in the worst-case, the time complexity is:

$$T_{\text{QuickSort}}(n) = O(n^2)$$

II-D. *BubbleSort*

BubbleSort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats for each item in the list until the list is sorted.

Complexity: - Average Time Complexity: $O(n^2)$ - Worst-Case Time Complexity: $O(n^2)$ - Space Complexity: $O(1)$

Effective Usage: Due to its simplicity, BubbleSort can be effective for smaller lists or lists that are mostly sorted. However, for larger datasets, its quadratic time complexity makes it less efficient compared to more advanced sorting algorithms. The implementation of the algorithm is represented in Listing 2,

```
/**
 * Sorts an array using the BubbleSort algorithm.
 *
 * @param A: The array to be sorted.
 */
Function BubbleSort(A)
    n ←  length(A)
    for i from 0 to n-1
        for j from 0 to n-i-1
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
            end if
        end for
    end for
```

Listing 2: BubbleSort Algorithm

II-E. *Worst Case Analysis of BubbleSort*

In the worst-case scenario for BubbleSort, the list is in reverse order. During each iteration of the inner loop, the largest element (or the next largest) is ”bubbled up”to its correct position. For an array of size n:

The inner loop will run for:

$$(n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{n(n-1)}{2}$$

This yields a worst-case time complexity of:

$$T_{\text{BubbleSort}}(n) = O(n^2)$$

II-F. *Average Case Analysis of BubbleSort*

For the average case, assuming random input, half of the elements will be out of order on average. This means that the inner loop will still perform approximately half of its maximum number of comparisons, which results in the quadratic time complexity.

Thus, the average case time complexity is also:

$$T_{\text{BubbleSort}}(n) = O(n^2)$$

II-G. *MergeSort*

MergeSort is another divide-and-conquer algorithm that works by recursively splitting the array in half until each sub-array consists of a single element and then merging those sub-arrays in a manner that results in a sorted array.

Complexity: - Average Time Complexity: $O(n \log n)$ - Worst-Case Time Complexity: $O(n \log n)$ - Space Complexity: $O(n)$

Effective Usage: MergeSort is a stable sort and guarantees $O(n \log n)$ time complexity in the worst case. This makes it a reliable choice for many applications. It is especially useful when data is stored in linked lists because of its linear space complexity, or when stability in sorting is a requirement. The implementation of the algorithm is represented in Listing 3,

```
/**
 * Sorts an array using the MergeSort algorithm.
 *
 * @param A: The array to be sorted.
 * @param start: The starting index of the portion of A to be sorted.
 * @param end: The ending index of the portion of A to be sorted.
 */
Function MergeSort(A, start, end)
    if start < end
        mid ← (start + end) / 2
        MergeSort(A, start, mid)
        MergeSort(A, mid + 1, end)
        Merge(A, start, mid, end)
    end if

/**
 * Merges two sorted sub-arrays into a single sorted array.
 *
 * @param A: The main array containing the two sub-arrays.
 * @param start: The starting index of the first sub-array.
 * @param mid: The ending index of the first sub-array and starting index of the second - 1.
 * @param end: The ending index of the second sub-array.
 */
Function Merge(A, start, mid, end)
    n1 ← mid - start + 1
    n2 ← end - mid
    let L[0..n1] and R[0..n2] be new arrays
    for i = 0 to n1
        L[i] ← A[start + i]
    for j ← 0 to n2
        R[j] ← A[mid + j + 1]

    (i, j, k) ← (0, 0, start)
    while i < n1 and j < n2
        if L[i] ≤ R[j]
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← j + 1
        end if
        k ← k + 1
    end while

    while i < n1
        A[k] ← L[i]
        i ← i + 1
        k ← k + 1
    end while
    while j < n2
        A[k] ← R[j]
        j ← j + 1
        k ← k + 1
    end while
```

Listing 3: MergeSort and its Merge function

II-H. *Complexity Analysis of MergeSort*

II-H1. Merge Operation: The complexity of the merge operation is linear, $O(n)$, because every element is processed once during the merging phase. The main value of the final complexity depends on the **Merge** function implementation.

II-H2. Recursive Splitting: The sorting operation of MergeSort works by recursively dividing the array into two halves and then merging them. Let $T(n)$ represent the time complexity of sorting an array of size n. The recurrence relation for MergeSort is given by:

$$T_{\text{MergeSort}}(n) = 2T_{\text{MergeSort}}\left(\frac{n}{2}\right) + O(n)$$

Where:

- $2T_{\text{MergeSort}}\left(\frac{n}{2}\right)$ represents the time to sort the two halves.
- $O(n)$ represents the time to merge the two halves.

This recurrence can be solved using the Master Theorem or the tree method. For MergeSort, it resolves to:

$$T_{\text{MergeSort}}(n) = O(n \log n)$$

Both for the average and worst-case scenarios.

III. Experiment's results

The whole table with the experiment results, slightly separated and occasionally pointed, can be found at the following GitHub Link. In the Table **??** and Table **??** the results of the the performance efficiency are represented respectfully for the testa with the arrays of the randomly refined elements and the array sorted in the backward manner indicating the worst case.

IV. Proof of Performance Superiority

IV-A. *Random Case Analysis*

Given the data for the random case in Table 1, we observed that for $n = 342$:

$$\bar{T}_{\text{QuickSort}}(342) < \bar{T}_{\text{BubbleSort}}(342)$$

Tabla I: Performance Analysis for Random Case

Array Length	QuickSort Time	BubbleSort Time	Merge Sort Time
2	0.040	0.020	0.020
102	0	0.020	0.060
202	0.010	0.070	0.100
302	0.030	0.160	0.130
402	0.020	0.440	0.310
502	0.030	0.410	0.190
602	0	0.580	0.240
702	0.080	0.730	0.270
802	0.050	0.950	0.330
902	0.070	1.170	0.400

Tabla II: Performance Analysis for Worst Case

Array Length	QuickSort Time	BubbleSort Time	Merge Sort Time
2	0	0	0
102	0.010	0.030	0.030
202	0.020	0.050	0.110
302	0.130	0.120	0.070
402	0.180	0.230	0.160
502	0.280	0.370	0.200
602	0.420	0.530	0.240
702	0.540	0.730	0.250
802	0.840	1.060	0.410
902	1.060	1.340	0.440

and

$$\bar{T}_{\text{MergeSort}}(342) < \bar{T}_{\text{BubbleSort}}(342)$$

From the definition of our function $\bar{T}$ and the data, it can be seen that these inequalities hold true. Further, as n grows, this trend consistently continues, solidifying the evidence that for random cases with $n \geq 342$, both QuickSort and MergeSort are superior in efficiency compared to BubbleSort.

IV-B. *Worst Case Analysis*

Referring to the worst-case data in Table II, it is evident that for $n = 472$:

$$\bar{T}_{\text{QuickSort}}(472) < \bar{T}_{\text{BubbleSort}}(472)$$

and

$$\bar{T}_{\text{MergeSort}}(472) < \bar{T}_{\text{BubbleSort}}(472)$$

Again, leaning on our function $\bar{T}$ and the given data, these inequalities hold true. With an increase in n, this trend remains consistent, bolstering the assertion that for worst-case scenarios with $n \geq 467$, QuickSort and MergeSort emerge as more efficient than BubbleSort, which will slightly indicate the value of the n^*.

Conclusion: From our detailed analysis, it's conclusively evident, under the testing conditions and given the performance metrics, that QuickSort and MergeSort consistently outperform BubbleSort beyond certain array lengths in both random and worst-case scenarios.

V. Conclusion

Sorting algorithms, by virtue of their foundational importance in computer science, have been a subject of continuous exploration. This study aimed to discern the relative efficiencies of three prevalent sorting algorithms: BubbleSort, QuickSort, and MergeSort.

From our rigorous evaluations, we observed pivotal values of n, specifically n^*, where the performance characteristics of these algorithms diverge significantly. Mathematically, for the random case, we determined that at $n = 342$:

$$\bar{T}_{\text{QuickSort}}(342) < \bar{T}_{\text{BubbleSort}}(342)$$

and

$$\bar{T}_{\text{MergeSort}}(342) < \bar{T}_{\text{BubbleSort}}(342)$$

For the worst-case scenario, these disparities were discernible at $n = 472$:

$$\bar{T}_{\text{QuickSort}}(472) < \bar{T}_{\text{BubbleSort}}(472)$$

and

$$\bar{T}_{\text{MergeSort}}(472) < \bar{T}_{\text{BubbleSort}}(472)$$

These mathematical conclusions, drawn from the experimental results, underline BubbleSort's inefficiencies when confronted with larger array sizes. Both QuickSort and MergeSort, with complexities in the ballpark of $O(n \log n)$, showcased their resilience and efficiency, outperforming the $O(n^2)$ complexity of BubbleSort beyond the established thresholds of n^*.

In essence, our findings highlight the essentiality of choosing the appropriate sorting algorithm tailored to the specific nature and size of the dataset in question. For larger datasets, the robustness and efficiency of QuickSort and MergeSort are incontrovertible.

Referencias

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo - Techniczne, 2004.