Lab 00: A gentle introduction for a “good” report.

Andrés Oswaldo Calderón Romero, PhD

July 30, 2025

1 Introduction

In this report, two algorithms for sorting arrays of integers, as studied in class, are implemented and analyzed. These algorithms are Insertion-sort and Merge-sort. Once both algorithms were implemented, they were subjected to an empirical analysis to compare their response times and verify the asymptotic analysis illustrated in class. Finally, some observations found during the development of this report are discussed.

2 Algorithm Implementation

For the implementation of the algorithms, the pseudocodes studied in class were followed, and the coding was carried out in the Python programming language. Below, we present the pseudocodes and implementations for each algorithm.

2.1 Insertion-sort

The pseudocode on which the implementation was based can be seen in Algorithm 1, and the source code of the implementation is presented in Figure 1.

Algorithm 1 Insertion-sort pseudocode.

```
1: INSERTION-SORT($A, n$)        ▷$A[1 \dots n]$
2:     for $j \leftarrow 2$ to $n$ do
3:         $key \leftarrow A[j]$
4:         $i \leftarrow j - 1$
5:         while $i > 0$ and $A[i] > key$ do
6:             $A[i+1] \leftarrow A[i]$
7:             $i \leftarrow i - 1$
8:         $A[i+1] \leftarrow key$
```

```python
#!/usr/bin/env python3

def insertion_sort(array):
    for step in range(1, len(array)):
        key = array[step]
        j = step - 1
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key

def main():
    array = [8, 2, 4, 9, 3, 6]
    insertion_sort(array)
    print(array)

if __name__ == "__main__":
    main()

```

Figure 1: Implementation of Insertion-sort in Python 3.

2.2 Merge-sort

Similarly, the pseudocode of Algorithm 2 was followed for the implementation shown in Figure 2.

Algorithm 2 Merge-sort pseudocode.

```
1: MERGE-SORT($A, n$)        ▷ $A[1 \dots n]$
2:     if $n = 1$, done.
3:     Recursively sort $A[1 \dots \lceil \frac{n}{2} \rceil]$ and $A[\lceil \frac{n}{2} \rceil + 1 \dots n]$
4:     MERGE the 2 sorted list.
```

```python
#!/usr/bin/env python3

def merge_sort(array):
    if len(array) > 1: # recurse until length of array is 1...
        r = len(array)//2 # split the array in a half...
        L = array[:r]
        R = array[r:]
        merge_sort(L) # recurse in the left hand side (LHS)...
        merge_sort(R) # recurse in the right hand side (RHS)...
        i = 0
        j = 0
        k = 0
        # the MERGE function...
        while i < len(L) and j < len(R): # iterate until we reach the end of one array...
            if L[i] < R[j]:
                array[k] = L[i] # if LHS is lower than RHS, let's pick that value...
                i += 1
            else:
                array[k] = R[j] # otherwise, let's pick the RHS value...
                j += 1
            k += 1 # keep track of the size of the sorted array...
        while i < len(L): # if L still have values, let's move them to the sorted array...
            array[k] = L[i]
            i += 1
            k += 1
        while j < len(R): # if R still have values, let's move them to the sorted array...
            array[k] = R[j]
            j += 1
            k += 1

def main():
    array = [8, 2, 4, 9, 3, 6]
    merge_sort(array)
    print(array)

if __name__ == "__main__":
    main()

```

Figure 2: Implementation of Merge-sort in Python 3.

3 Experiments

To empirically analyze the performance of the implementations 1 and 2, the following subroutine was implemented to generate an array of a specified number of integers. The size of the array is passed to the subroutine as the parameter n. Figure 3 shows the implementation of this subroutine.
Along with the implementations of both algorithms and the random array generator, a script was programmed to run a series of 5 tests that recorded the execution time of each implementation under different array sizes. The analysis focused on arrays ranging in size from 1000 to 20000 integers. The results were analyzed using the R programming language, where we aggregate the data and compute the average execution time for each case. Finally, a graph was generated to visually compare the performance of the two algorithms. Figure 4 illustrates the findings.

```
#!/usr/bin/env python3

import random

# traverse a range of length n and for each n call a get a random integer between 0 and n...
def generate_random_array(n):
    return [random.randint(0, n) for _ in range(n)]

def main():
    n = random.randint(30, 100) # test with a value of n between 30 and 100...
    random_array = generate_random_array(n)
    print(random_array)

if __name__ == "__main__":
    main()

```

Figure 3: Implementation of a random array generator in Python 3.

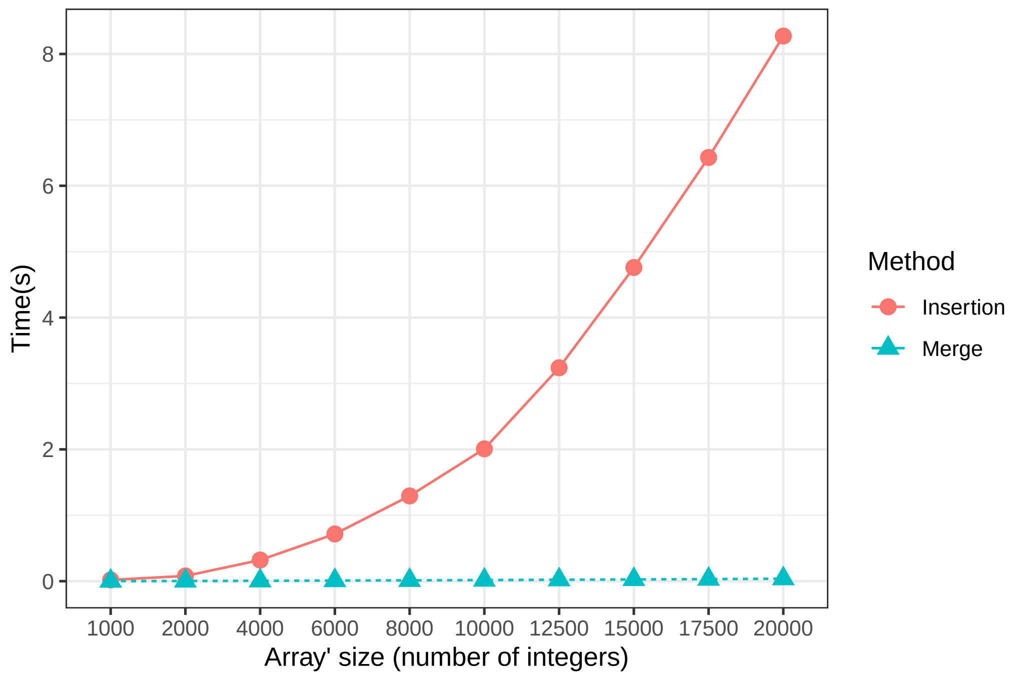

Figure 4: Performance of the analyzed algorithms under various array sizes.

4 Conclusion

It is clear that the implementation of Merge-sort has a much better performance than that of Insertion-sort. The response times of Insertion-sort clearly exhibit its quadratic performance, as seen in class $(O(n^2))$. However, the behavior of Merge-sort, under the analyzed array sizes, gives the illusion of constant performance $(O(1))$ instead of logarithmic $(O(n \log n))$. It seems that the implementation would need to be tested with much larger array sizes to observe this expected trend.