DAT600 – Assignment 1

Andreas Pedersen and Marius Hauge Håland

January 29, 2025

GitHub Repository

1 Task 1: Counting the steps

In this task we implement the four sorting algorithms; insertion sort, merge sort, heap sort, and quick sort. Then we vary the input size and record the steps taken for each algorithm. Plots for this are shown below.

Figure 1 shows the worst case for insertion sort, which is $O(n^2)$. Figure 2 shows random cases for insertion sort.

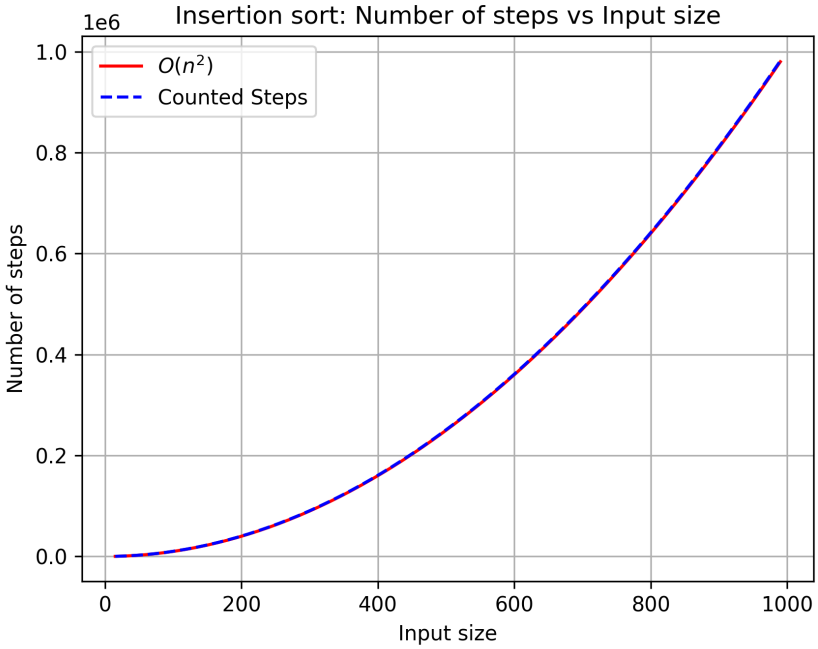

Figure 1: Insertion sort with **worst** case array. Shows that it is $O(n^2)$.

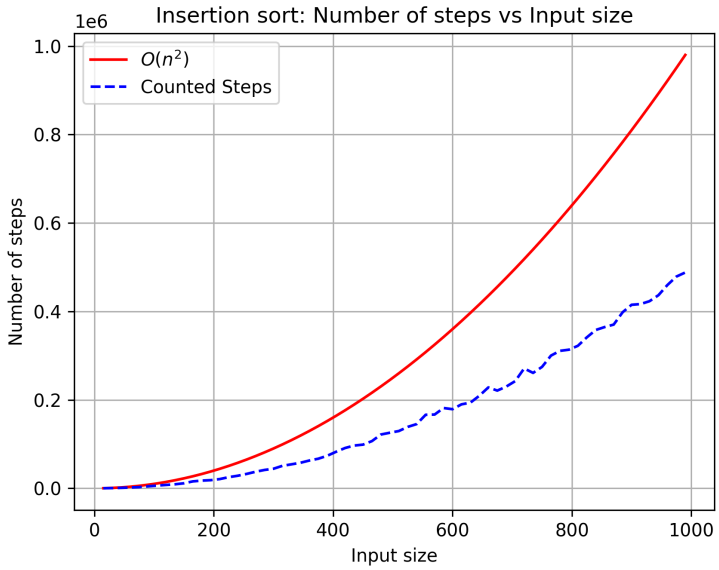

Figure 2: Insertion sort with a **random** array. Showcases a more realistic case, where it is not always the worst case.

Figure 3 shows merge sort with random cases.

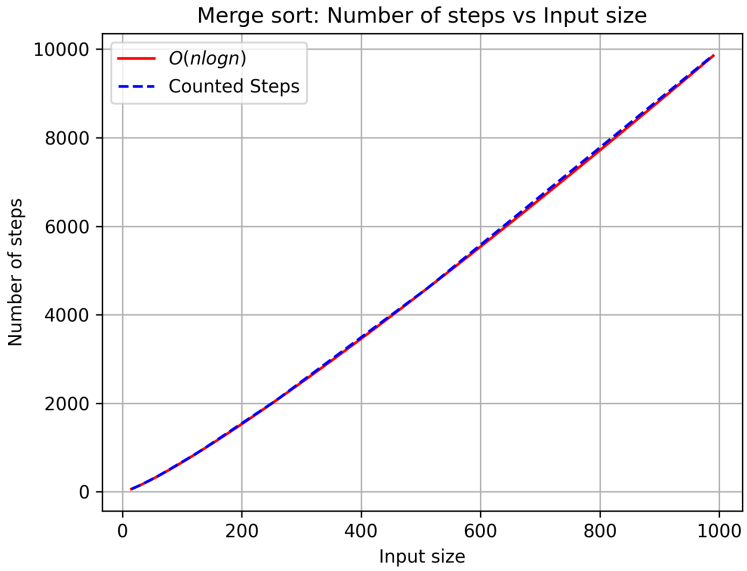

Figure 3: Merge sort with a **random** array, very close to O($nlog(n)$).

Figure 4 shows heap sort with random cases. As seen, the counted steps function is above the upper bound (O($nlog(n)$). This observation suggests that while heap sort generally performs within the expected time complexity bounds, practical factors such as the number of comparisons and swaps during heapification may introduce additional overhead.

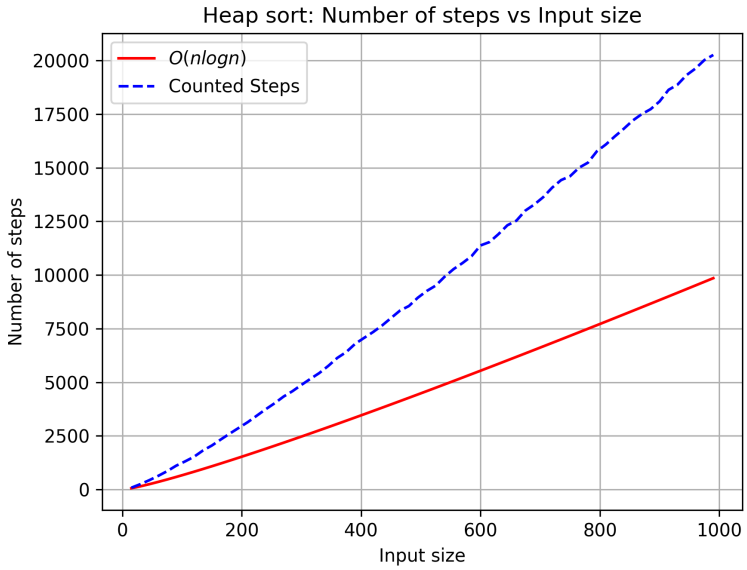

Figure 4: Heap sort with **random** cases.

Figure 5 shows the quick-sort algorithm with worst-case time complexity.

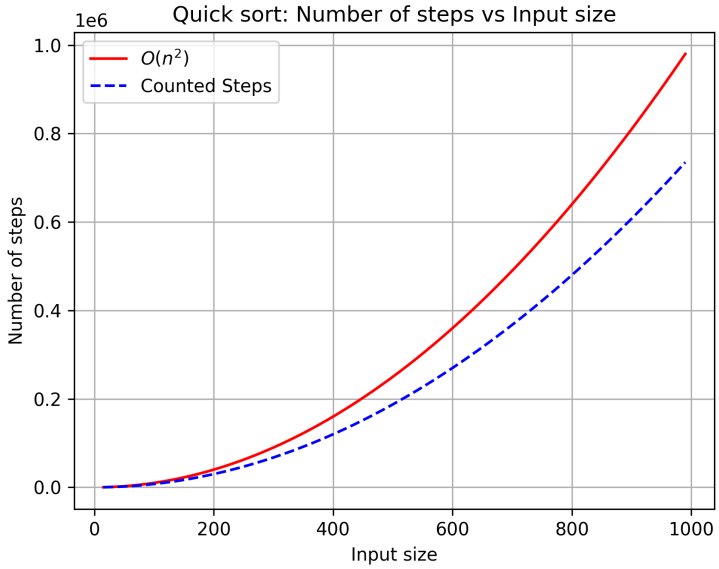

Figure 5: Quick sort with **worst** cases.

Figure 6 shows a comparison of all the sorting algorithms implemented with the worst-case time complexity.

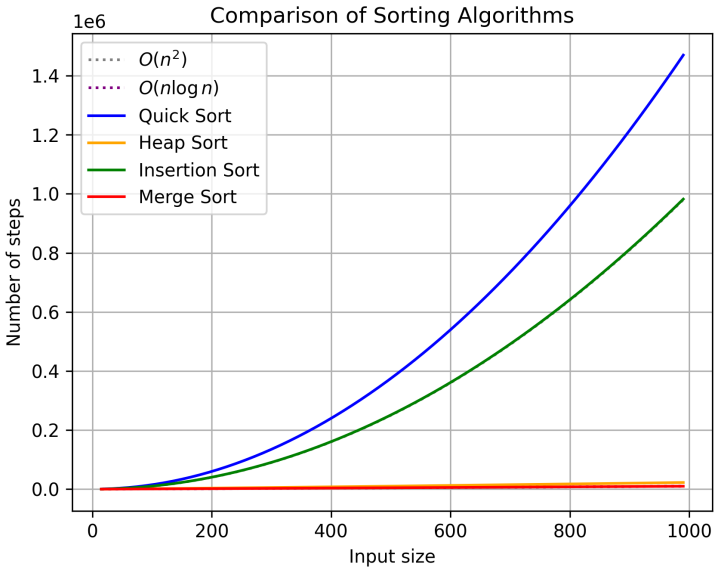

Figure 6: Comparison plot of all four sorting algorithms.

2 Task 2: Compare true execution time

We chose to implement the insertion sort algorithm in two different languages in this task. This was done in **Python** and **Go** (Golang).

Array Size	Go (ms)	Python (ms)	Percentage Difference (%)
10	0.000459	0.006198	1250.3
100	0.00625	0.178098	2749.5
1000	0.480667	20.088672	4079.3

Table 1: Runtime comparison of insertion sort algorithm in Go and Python with Percentage Difference

The programming language Go is significantly faster in sorting arrays using the insertion sort algorithm. For an array with n = 1000, Go is about 4000% quicker.

2.1 Code for Insertion Sort

2.1.1 Arrays used

```
# Sorted arrays, but reversed
arr10 = [i for i in range(10, 0, -1)]
arr100 = [i for i in range(100, 0, -1)]
arr1000 = [i for i in range(1000, 0, -1)]
array_list = [arr10, arr100, arr1000]
```

Listing 1: Arrays used in both implementations.

2.1.2 Python Code

```python
def insertion_sort(arr):
    n = len(arr)

    if n <= 1:
        return

    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return

# Store times taken
times = []

for arr in array_list:
    start_time = time.time()
    insertion_sort(arr)
    end_time = time.time()
    elapsed_time_ms = (end_time - start_time) * 1000 # Convert to milliseconds
    times.append(elapsed_time_ms)
```

Listing 2: Insertion sort in Python with varying the input size and measuring the execution time.

2.1.3 Go Code

```go
func insertionSort(arr []int) {
    n := len(arr)

    if n <= 1 {
        return
    }

    for i := 1; i < n; i++ {
        key := arr[i]
        j := i - 1
        for j >= 0 && key < arr[j] {
            arr[j+1] = arr[j]
            j--
        }
        arr[j+1] = key
    }
    return
}

// Store times taken
times := []float64{}

for _, arr := range arrays {
    timeStart := time.Now()
    insertionSort(arr)
    timeEnd := time.Now()
    times = append(times, float64(timeEnd.Sub(timeStart).Nanoseconds())/1e6)
}
```

Listing 3: Insertion sort in Go with varying the input size and measuring the execution time.

3 Task 3: Basic proofs

3.1 Show that for any real constants a and b, where $b > 0, (n + a)^b = \Theta(n^b)$

According to the definition of Big-Theta notation, for $f(n) = (n + a)^b$ and $g(n) = n^b$, we say $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0.$$

Step 1: Proving the Upper Bound
To determine the upper bound, we start with the inequality:

$$(n+a)^b \leq c_2 n^b.$$

Rewriting this to isolate c_2, we have:

$$\left(\frac{n+a}{n}\right)^b \leq c_2.$$

Further simplification yields:

$$\left(1+\frac{a}{n}\right)^b \leq c_2.$$

For sufficiently large n, the term $\left(1+\frac{a}{n}\right)^b$ approaches 1. Therefore, there exists a constant $1 \leq c_2$. This establishes the upper bound.

Step 2: Proving the Lower Bound
Similarly, observe that:

$$c_1 n^b \leq (n+a)^b.$$

Using the same simplification as for the upper bound we get:

$$c_1 \leq \left(1+\frac{a}{n}\right)^b.$$

For a sufficiently large n, the term $\left(1+\frac{a}{n}\right)^b$ approaches 1. Therefore, there exists a constant $c_1 \leq 1$. This establishes the lower bound.

Step 3: Conclusion
Combining these inequalities, we obtain:

$$c_1 n^b \leq (n+a)^b \leq c_2 n^b$$

for all $n \geq n_0$, where n_0 is large enough so that the inequalities hold. Therefore, we have shown that:

$$(n+a)^b = \Theta(n^b)$$

This completes the proof.

3.2 Show that $\frac{n^2}{\log n} = o(n^2)$

By definition, $f(n) = o(g(n))$ if for every constant $c > 0$, there exists a positive integer N such that for all $n > N$,

$$f(n) < c \cdot g(n).$$

Let $f(n) = \frac{n^2}{\log n}$ and $g(n) = n^2$. We need to show that

$$\frac{n^2}{\log n} < c \cdot n^2 \quad \text{for all} \quad n > N.$$

Dividing by n^2,

$$\frac{1}{\log n} < c.$$

Since $\log n \to \infty$ as $n \to \infty$, there exists a sufficiently large N such that for all $n > N$,

$$\frac{1}{\log n} < c.$$

Thus, the inequality holds. We conclude that

$$\frac{n^2}{\log n} = o(n^2).$$

3.3 Show that $n^2 \neq o(n^2)$

Assume, for the sake of contradiction, that $n^2 = o(n^2)$. By definition, this implies that for every constant $c > 0$, there exists an integer $N > 0$ such that for all $n > N$,

$$n^2 < c \cdot n^2.$$

Dividing both sides by n^2 (valid for $n \neq 0$),

$$1 < c \quad \text{for all } n > N.$$

This is a contradiction because choosing $c = 1$ makes the inequality false. Thus, our assumption is incorrect, and we conclude

$$n^2 \neq o(n^2).$$

4 Task 4: Divide and Conquer Analysis

4.1 Time Complexity of Recurrence $T(n) = 3T(n/2) + \Theta(n)$

We apply the Master Theorem for recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a = 3$, $b = 2$, and $f(n) = \Theta(n)$.

First, calculate $\log_b a$:

$$\log_2 3 \approx 1.585$$

Now, we compare $f(n) = \Theta(n)$ with $n^{\log_2 3}$. Since $n = O(n^{\log_2 3 - \epsilon})$ for some $\epsilon > 0$, we are in **Case 1** of the Master Theorem. Thus, the time complexity is:

$$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.585})$$

At each level of the recursion tree, the work done is $\Theta(n)$. The number of subproblems grows exponentially by a factor of 3 at each level, so at level k, there are 3^k subproblems, each of size $n/2^k$. The total work at each level remains $\Theta(n)$, but the number of levels is $\log_2 n$.

Thus, summing the work at all levels of the recursion tree:

$$\text{Total work} = \Theta(n) \times \log_2 3 = \Theta(n^{1.585})$$

This confirms that the time complexity grows faster than $\Theta(n \log n)$ due to the exponential growth in the number of subproblems.