# Stack sorting algorithm

Leo Man
Jesper Persson
Gustav Hammarström
Department of Engineering Sciences and Mathematics


December 13, 2024

# 1 Introduction

In this assignment, we have created a stack using a linked list. Then using our custom designed stack given that there are integers stored in the stack, we created an algorithm that sorts these integers in a descending order with the allowed operations according to the assignment.

# 2 Designing the stack

There is really nothing special about the stack we desgined. The only thing worth mentioning is what happens when a pop method is used on an empty stack. We decided to design the pop method so that it returns the value **None** when executed on an empty stack. Although not necessary it would make creating the algorithm a bit less verbose then if we were to throw an error.

# 3 How the algorithm works

The best way to explain how the algorithm works is with a deck of cards, that we want to sort in ascending order.

We have two piles, Pile A is the initial unsorted deck of cards, and pile B which is initially empty where we will place our sorted cards. And then we have our two hands, our right hand will be used to pick up cards from pile A, and our left hand to pick up cards from Pile B.

Now we are ready to preform the algorithm.

1. While Pile A is not empty preform the following steps.
2. Pick up the top card from pile A and store it in your right hand.
3. Pick up the top card from pile B and store it in your left hand.
4. If your left hand is still empty put down the card in your right hand in pile B and return to step 1
5. Now compare the cards in your left and right hand. If the card in your left hand is smaller than the right hand card, put the card in your left hand on top of pile A and return to step 3.
6. Put down the card in your left hand on top of pile B and then the card in your right hand on top of pile B
7. Return to step 1

# 4 The Worst Case

Ironically, the worst-case scenario for this algorithm occurs when the stack is already sorted. Which may seem counterintuitive.

Consider a stack of 10 cards sorted in ascending order, and that the first 9 cards have been correctly sorted into pile B. To correctly place the last card into the sorted pile (Pile B), the algorithm must first move each card in pile B back to pile A, before correctly placing the last card. And then it has to individually move back each card from pile A back to pile B.

In contrast, the best-case scenario occurs when the stack is sorted in descending order. Because all the algorithm has to do is move the top card from pile A to pile B.

# 5 The exact number of operations

| Line Number | Operation | Cost | Times Executed |
|---|---|---|---|
| 1 | `while not inputStack.isEmpty()` | $c$ | $n$ |
| 2 | `  rightHand = inputStack.pop()` | $c$ | $n-1$ |
| 3 | `  leftHand = tempStack.pop()` | $c$ | $n-1$ |
| 4 | `  while leftHand < rightHand` | $c$ | $\sum_{i=1}^{n}(t_i)$ |
| 5 | `    leftHand = tempStack.pop()` | $c$ | $\sum_{i=1}^{n}(t_i-1)$ |
| 6 | `  if leftHand is not None` | $c$ | $n-1$ |
| 7 | `    tempStack.push(leftHand)` | $c$ | $n-1$ |
| 8 | `  tempStack.push(rightHand)` | $c$ | $n-1$ |

The rows that are ran the most are the the inner loop comparison. And after that it is the logic inside the nested loop. The lines that are ran the least the lines part of the outer loop.

Assuming that the cost is constant for each operation in this algorithm in combination with how many times each operation is ran gives us this expression

$$T(n) = cn + 5c(n-1) + c\sum_{i=1}^{n}(t_i) + c\sum_{i=1}^{n}(t_i - 1) \tag{1}$$

In the worst case, $t_i = i - 1$, so we substitute and simplify as follows:

$$\sum_{i=1}^{n} t_i = \frac{n^2 - n}{2}, \quad \sum_{i=1}^{n}(t_i - 1) = \frac{n^2 - 3n}{2}$$

Substituting these into $T(n)$:

$$\begin{aligned}
T(n) &= cn + 5c(n-1) + c \cdot \frac{n^2 - n}{2} + c \cdot \frac{n^2 - 3n}{2} \\
&= cn + (5cn - 5c) + c \cdot \frac{n^2 - n}{2} + c \cdot \frac{n^2 - 3n}{2} \\
&= 6cn - 5c + c \cdot \frac{n^2 - n}{2} + c \cdot \frac{n^2 - 3n}{2} \\
&= 6cn - 5c + c(n^2 - 2n) \\
&= 6cn - 5c + cn^2 - 2cn \\
&= cn^2 + 4cn - 5c.
\end{aligned}$$

The worst-case time complexity is:

$$T(n) = cn^2 + 4cn - 5c \tag{2}$$

## 5.1 Number of operations

After the algorithm was developed and modified such that the number of operations was calculated. The operations grew like this

As also can be seen in the graph

| Size | Operations |
|---|---|
| 1 | 6 |
| 10 | 600 |
| 100 | 60000 |
| 1000 | 6000000 |
| 10000 | 600000000 |

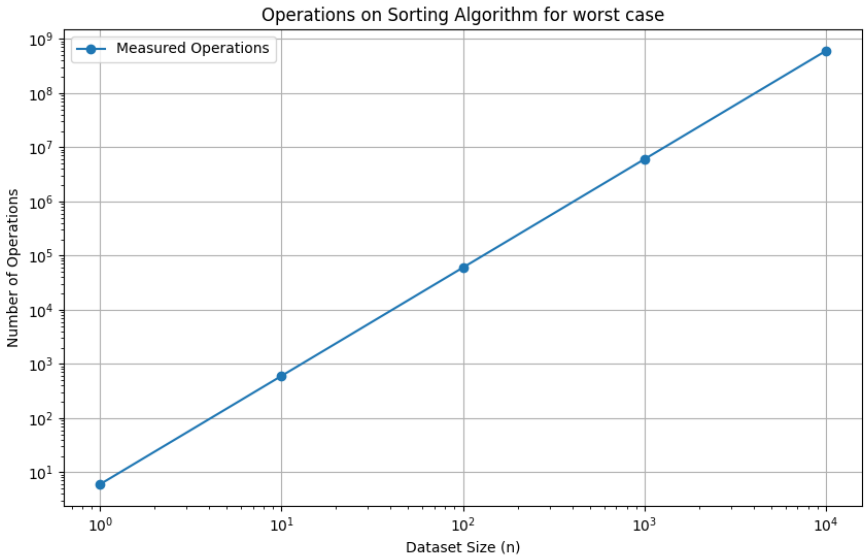


Figure 1: Number of operations on the worst case

# 6 The time complexity

To find the time complexity we ran the algorithm on the worst case scenario for different sizes ranging from 100 all the way up too 100 000. And the result we got was

| Size | Run Time [s] |
|---|---|
| 100 | 0.0041 |
| 1000 | 0.4073 |
| 10000 | 41.5257 |
| 100000 | 4279.9013 |

and as we can see the run time got quite big, quite quickly. And from this table we quickly could draw the conclusion that the time complexity of this was quadratic. When the size of the data was 100 000 it took roughly 71 minutes and if we where to run it on 1000 000 data points it would take just shy of 5 days. In contrast running 1000 000 data points on the best case just only took 8 seconds.