Algorithms and Data structures

Lab 1

Gruppdeltagare

Fabian (fabmoe-4@student.ltu.se)
Alex (alxkar-4@student.ltu.se)
Amund (amuknu-3@student.ltu.se)

Date: 2025-11-14

Luleå tekniska universitet

Introduction

This report discusses lab 1 in which we implement a sorting algorithm for a stack. We implement the algorithm using only the 3 main operations pop(), push() and isEmpty(), we are only allowed to use 2 stacks and a constant memory size for variables. We discuss the steps and the runtime of the algorithm for different input sizes. the algorithm for different input sizes.

Algorithm description (step by step)

This is the step by process and how the algorithm works

1. While input stack is not empty
    1. Pop first element from unsorted stack and store in `temp`
    2. While sorted stack is not empty
        1. Pop top element and store in `peeked_var`
        2. Compare:
            - If `peeked_var` is smaller than `temp`
                1. This element belongs above `temp` so it must be moved out of the way
                2. Push `peeked_var` onto the unsorted stack
                3. Continue the inner loop to check next element
            - Else (`peeked_var` is larger or equal to `temp`)
                1. This is correct spot for `temp`, `peeked_var` belongs below it
                2. Push `peeked_var` back onto the sorted stack
                3. Break from inner loop, correct spot i found
    3. Insert element
        - The inner loop is now done and correct spot for `temp` is found
        - Push `temp` onto the sorted stack

Code

```
while not unsortedStack.isEmpty():

      temp = unsortedStack.pop()

      while not sortedStack.isEmpty():
          peeked_var = sortedStack.pop()
          if peeked_var < temp:
              unsortedStack.push(peeked_var)
          else:
              sortedStack.push(peeked_var)
              break

      sortedStack.push(temp)
```

Calculating operations

We define the operations as push, pop, and comparisons in the main algorithm. These calculations are based on the worst case scenario.

The first step is to figure out when the worst-case happens. When understanding our algorithm, we see that the algorithm only looks at one number all the time. This means that if we want to add something to the bottom of the stack, the entire stack needs to move out the way. To maximize the amount of times this happens, we see that we need to give the input as a fully sorted stack. This

means that there will always come bigger and bigger numbers which will mean that we always have to move the stack away.

We can now start to analyze how many operations will happen. To do this we start with the inner loop.

```
while not sortedStack.isEmpty():
          peeked_var = sortedStack.pop()
          if peeked_var < temp:
              unsortedStack.push(peeked_var)
          else:
              sortedStack.push(peeked_var)
              break
```

We start by looking at the if statement. We have

```
if peeked_var < temp:
    unsortedStack.push(peeked_var)
else:
    sortedStack.push(peeked_var)
    break
```

. Since we have a sorted stack as input, the if statement will go through every time until the stack is completely emptied. For the first element the stack is already empty so the else statement will go through. For the next element the if statement will go through exactly once. Then increase by one for every element in the stack until the last one which will go through $n-1$ times. This can be written as an arithmetic sum:

$$\sum_{k=0}^{n-1} k = 0 + 1 + ... + n - 1 = \frac{n(n-1)}{2}$$

This is the amount of times the first part of the if statement is run.

After this, the else statement will go trough for all elements that were previously in the sorted stack so all n-1 elements that were ejected via the if statement. This will be expressed with the same arithmetic sum since all elements will need to be added back.

So the total number of times that the entire inner loop will be run can be expressed by: $\frac{n(n-1)}{2} \cdot 2 = n(n-1) = n^2 - n$. This is also the amount of comparisons in the if-statement. Looking at the operations again we can then see that, `peeked_var = sortedStack.pop()` is run $n^2 - n$ times, and both the inner push lines are each run, $\frac{n(n-1)}{2}$ times.

Now for the outer loop. Since we have n elements in the stack, the outer loop will run for n times. But, we also have all the times that we moved elements back into the unsorted stack so we have to add these times. This means that for both the push and pop method in the outer loop, we will have $n + \frac{n(n-1)}{2} = \frac{n(n+1)}{2}$ times.

So the total number of operations are all these added together $n^2 - n + n^2 - n + n^2 + n + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} = 4n^2 - 2n$

To prove this function, we first used counters in the actual code for all the comparisons, push and pops and got the counter to tell us the same thing as the formula. This formula was also tested further in the worst case graph below where we, from one given input size, could calculate the time the next input size would take.

Graphs

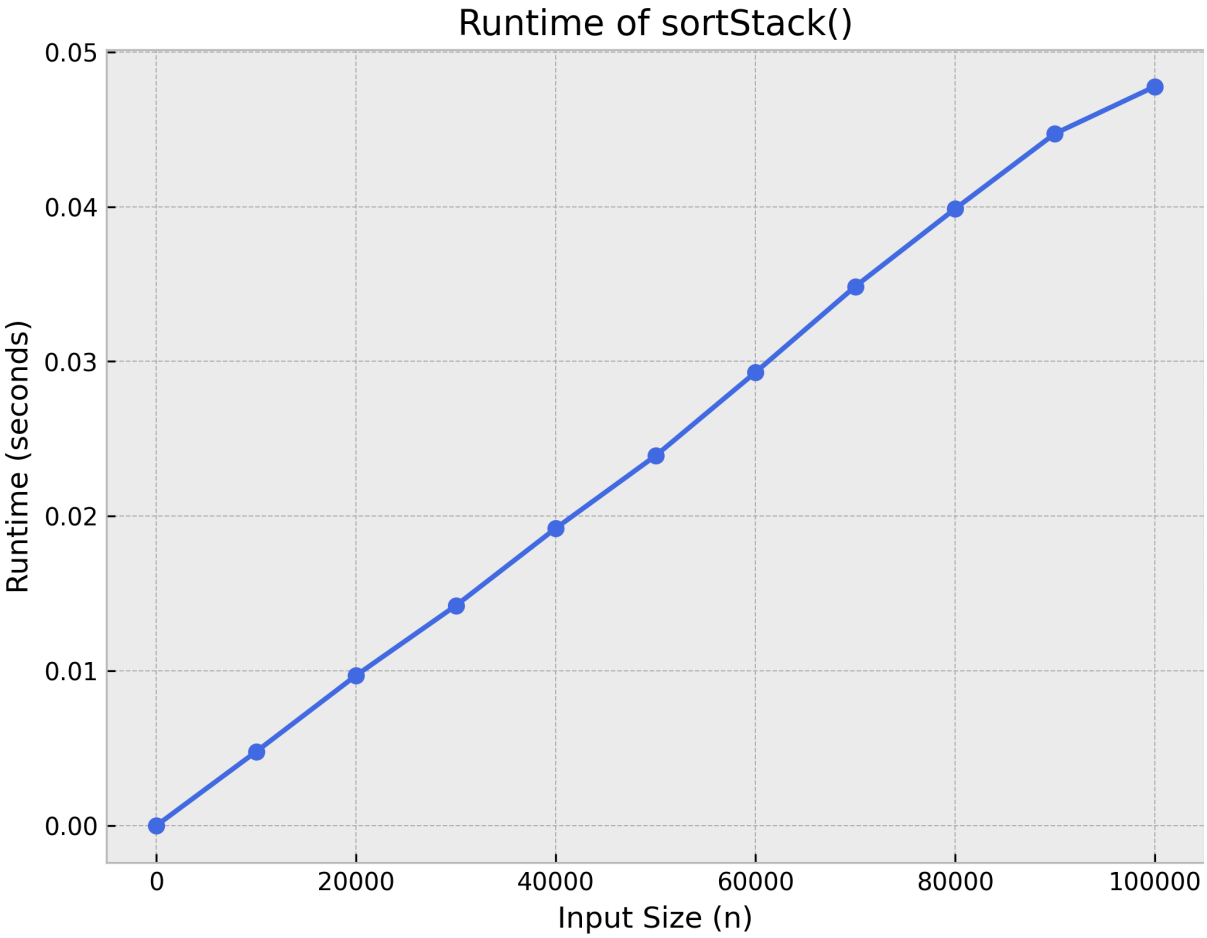

Figure 1: Best case. We see that the time of this is basically linear. This is because we don't have to loop through the elements at all since they are already in the correct spots. The reason why it is not completely linear is because the computer that runs it also has background tasks which will impact on the times. This is visible because the time is so small.

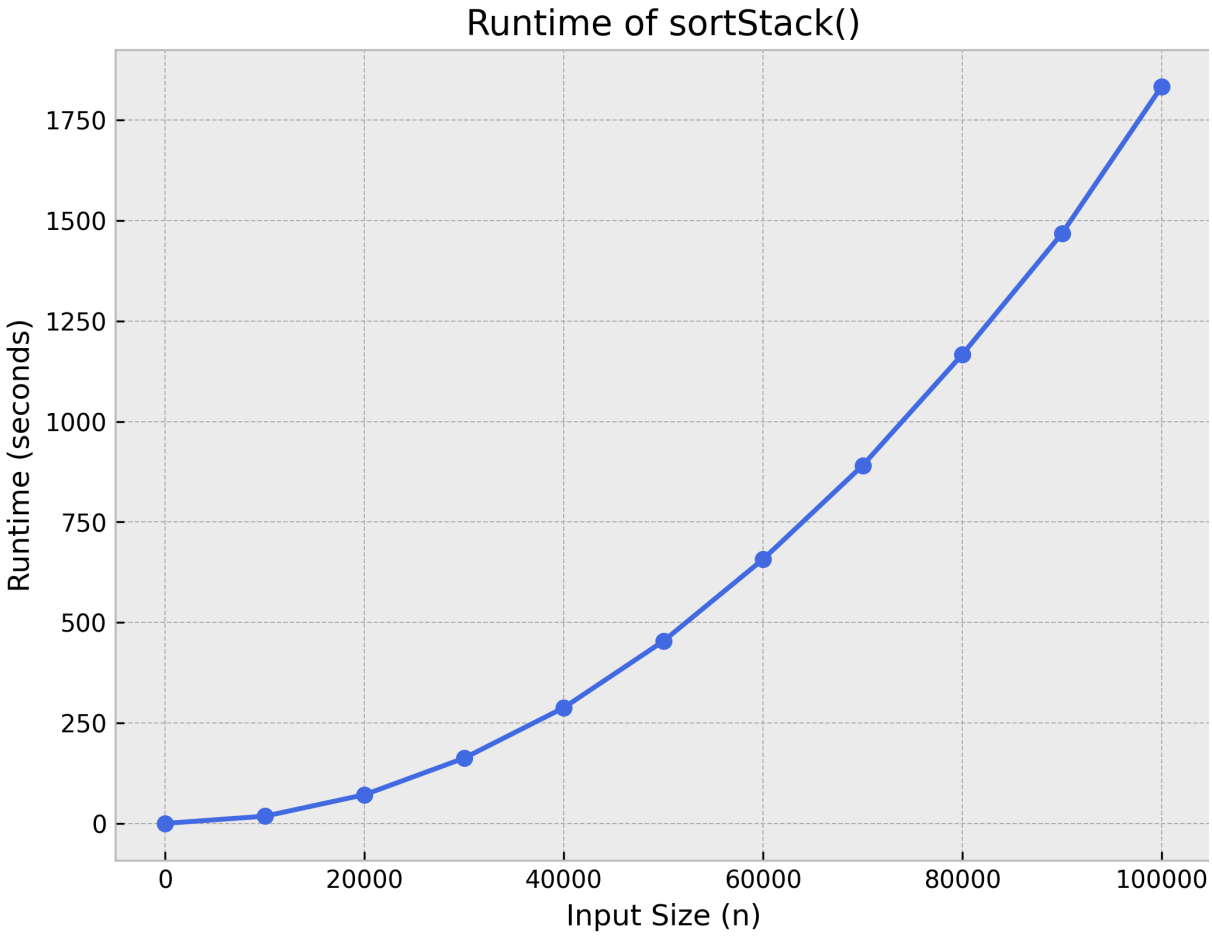

Figure 2: Random. This is a typical exponential graph n^2 wich makes sense considering our worst case is represented by O(n^2)

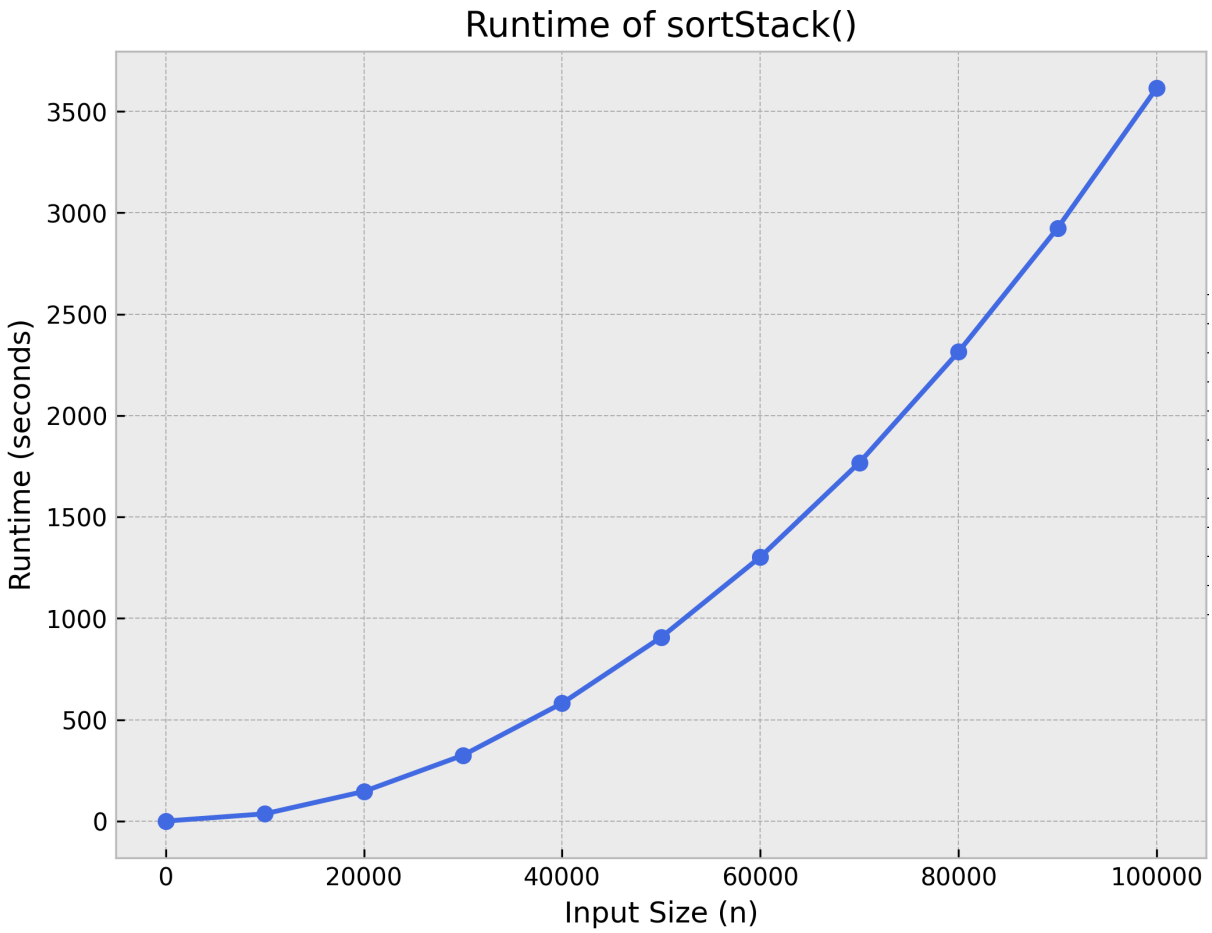

Figure 3: Worst case, when the stack is already sorted we see the time increasing drastically with bigger input sizes, because of exponential n^2 time complexity.