Assignment 1

January 31, 2024

1 Assignment 1

Ardijan Rexhaj, Daniel Alvsåker, Ove Oftedal

Source code for jupyterlab: https://github.com/ardijanr/dat600

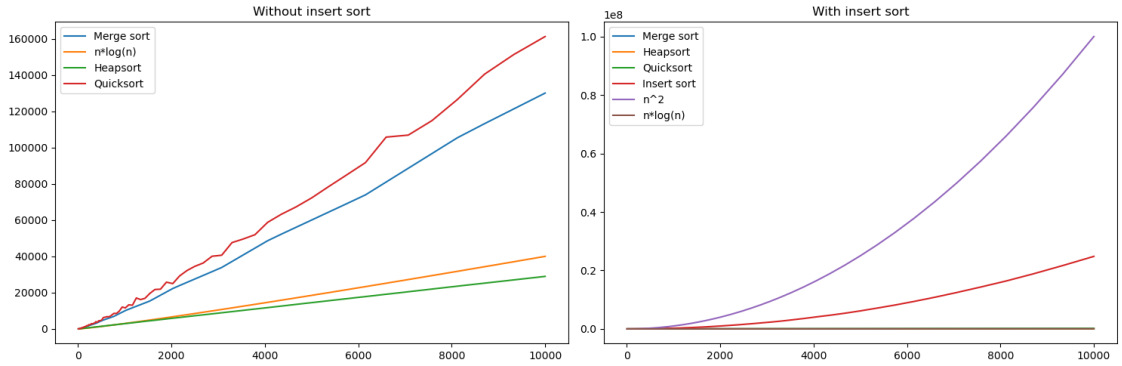
1.1 Task 1

We decided to use the algorithms implemented in Python to compare the expected asymptotic running time versus the actual number of steps taken. We created two graphs to illustrate this is because the n^2 running time makes seeing the difference between the $n \times log(n)$ functions difficult.

In order to compare the different algorithms we had to make sure they were all operating on an identical copy of the unsorted array. This was achieved by hard-coding an array with 10 000 elements which was then sliced into a desired length and then passed to each function.

The results are as expected, although not an exact match the general growth of the functions follows the expected $n \times log(n)$ and n^2 trends. Although our insertion sort was much faster than the worst-case n^2 number of iterations it still grows at an exponential rate.

[1]:
```
%run Assignment_1_1.ipynb
```

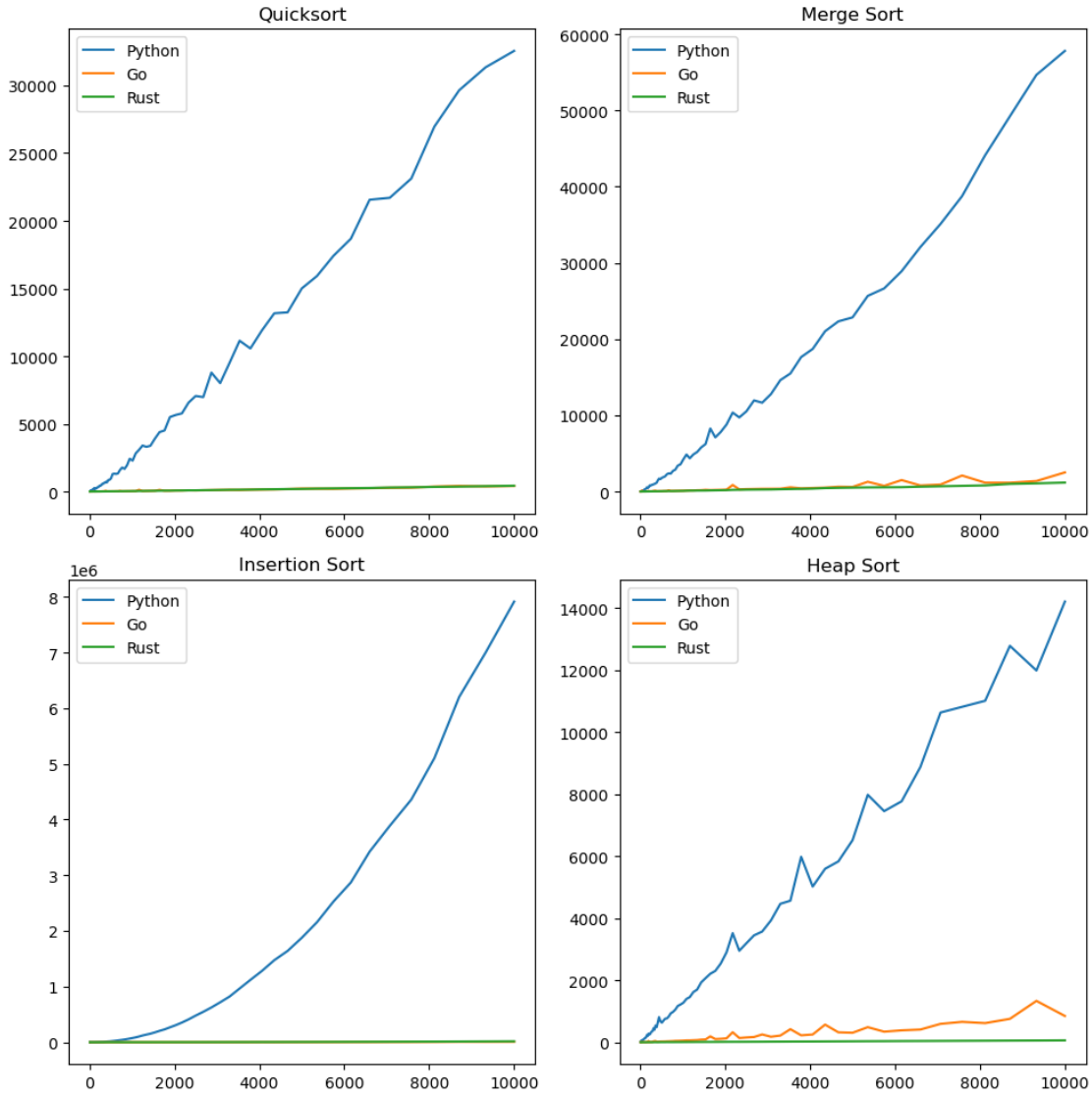


1.2 Task 2

We decided to compare three different programming languages; Python, Go and Rust. These were chosen because they are different types of languages, Python is interpreted, Go is garbage-collected but compiled and Rust is compiled without any garbage collector.

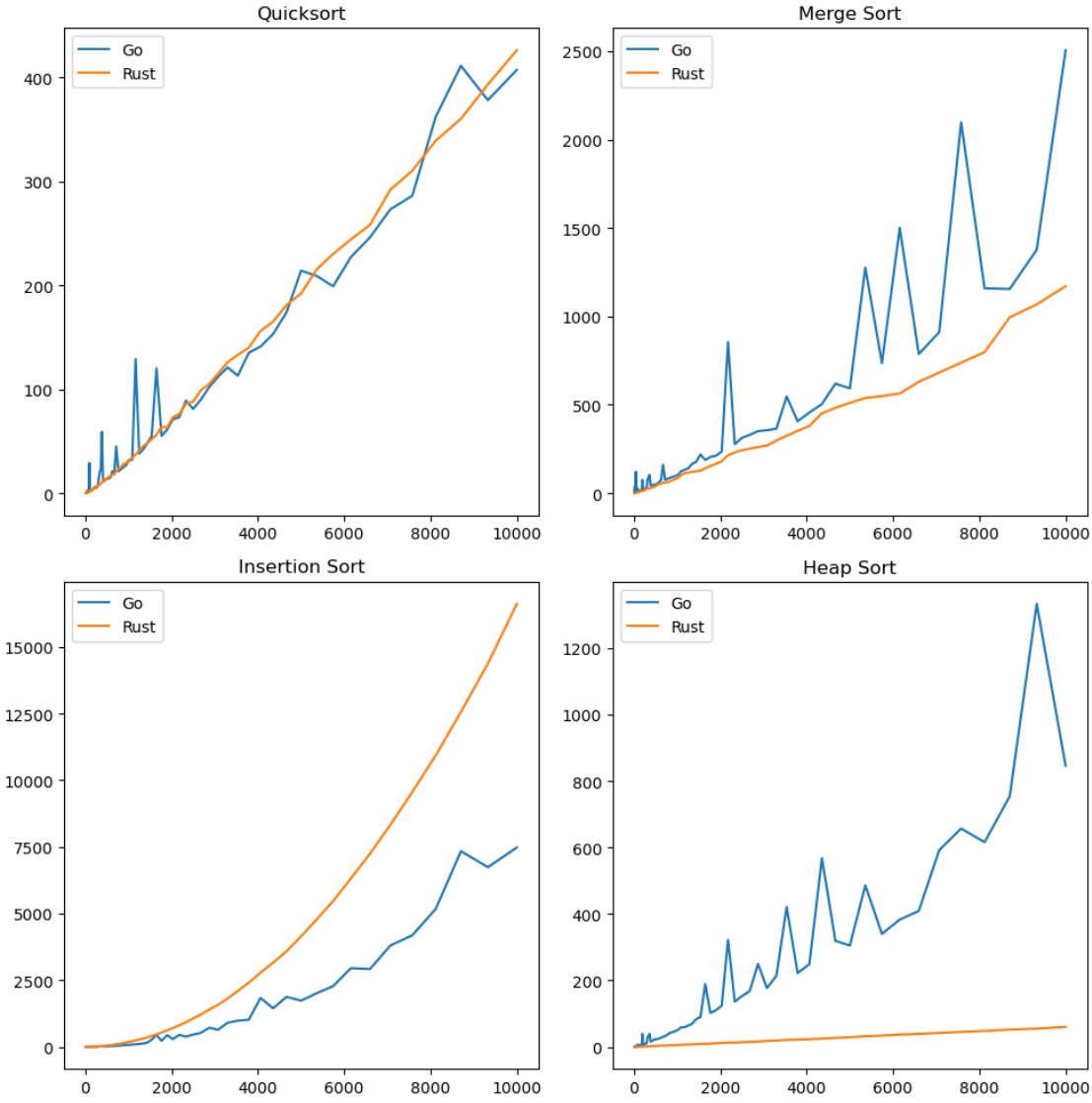
As expected Python is consistently worst in terms of running time and it is difficult to compare to the other two due to the massive gap in performance. Therefore another set of plots is provided without Python to compare Go and Rust.

[2]:
```
%run Assignment_1_2.ipynb
```



Below we compare Rust to Go and we were surprised by how close Go performs to Rust. However it seems the insertion sort implementation in Rust is somehow suboptimal most likely because it was written by two different members of the group and thus does not work identically. However it is also possible to see the heap sort algorithm is much faster in Rust compared to Go and the code is essentially a 1:1 mapping. In addition the spikes observed are most likely the garbage collector suspending the process to deallocate outscoped memory. Sometimes these spikes effectively double the running time of the Go code.

[3]:
```
%run Assignment_1_2_nopy.ipynb
```



1.3 Task 3

1. We this is solved by factoring out n and finding the limit, thus:

$$(n+a)^b = (n(1+\frac{a}{n}))^b$$

$$\lim_{n\to\infty}(n(1+0))^b = n^b$$

2. Here we use the definition from the book, and divide by what we are attempting to proove. If it converges to 0 it means it is true, if it diverges meaning the result is >0 it is false. Thus we can in this case prove that it holds.

$$\frac{n^2}{\frac{log(n)}{n^2}} = \frac{n^2}{log(n)\times n^2}, \lim_{n\to\infty}\frac{1}{log(n)} = 0$$

3. For this case we use the same method as in 2. and we can see that it diverges, and does not hold:

$$\lim_{n\to\infty}\frac{n^2}{n^2} = 1, n^2 \neq O(n^2)$$

1.4 Task 4

1.4.1 Master Theorem

$$T(n) = aT(\frac{n}{b}) + f(n^c)$$

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

which means:

$$a = 3, b = 2, c = 1$$

$$log_b a = log_2(3) \approx 1.585$$

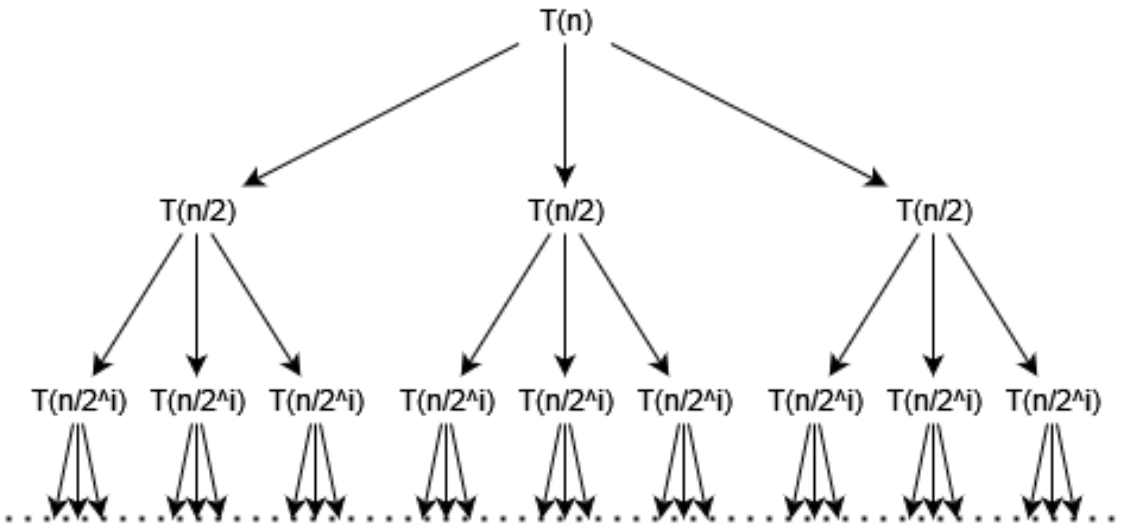
We check case 1 and find that it holds because $c < log_b a$ and $c > 0$.

$$O(n^{log_b a}) = O(n^{log_2(3)} \approx O(n^{1.585})$$

This does not converge thus $n^2 \neq O(n^2)$

1.4.2 Recursive Tree

This is how the tree looks:



Since it is broken into halves each time, the depth will equal to $log_2(n)$. It starts at $i = 0$ where the root node is and therefore can be expressed with:

$$\sum_{i=0}^{log_2(n)} (\frac{3}{2})^i n$$