Competitive Programming and Contests

Hands-on 1

LEONARDO DA POZZO

A.A. 2022/2023

1 Introduction

The purpose of this hands-on is to analyze and compare three different solutions to the "Sliding Window Maximum" problem. All solutions were implemented in the Rust programming language and tested on a Thinkpad T480 with an Intel(R) Core(TM) i5-8350U quad core CPU @ 1.70GHz and 8GB of installed RAM. The OS running on the machine is EndeavourOS. In all the following sections, n will refer to the array's size while k will be the window's length.

2 Brute Force Solution

2.1 Description

The brute force approach involves iterating over every window, keeping track of the maximum.

2.2 Complexity

Since there are $n - k$ windows, each of which has size k, the overall complexity is $O(n \cdot k)$.

3 Heap-based Solution

3.1 Description

The idea is to add one element to the heap at every iteration, querying for the max once the window is fully formed. Since non-root removal from a heap is quite complicated, we decide to leave elements in even when they drop out of the window, taking care to check that the queried max at each iteration is indeed still in the window and removing it if this is not the case. Elements are wrapped along with their position before insertion in the heap in order to perform this last check.

3.2 Complexity

Overall, this implementation takes $O(n \cdot log(n))$ time: we perform n insertions, each of which costs $O(log(n))$, and no more than $n - k$ max-extractions, which cost $O(log(n))$ as well. The cost of max-extractions come from the necessity of checking and eventually removing maximal elements from the root of the heap, which is $O(log(n))$ as mentioned.

4 BBST-based Solution

4.1 Description

This solution is almost identical to the heap-based one, but this time we can perform arbitrary removals in $O(log(k))$. Indeed, since we can perform removals, the tree will never exceed the size of $k + 1$, so insertions will be $O(log(k))$ too.

4.2 Complexity

Overall, this approach has a time complexity of $O(n \cdot log(k))$. Note that this is based on the assumption that Rust's BinarySearchTree is the implementation of a balanced binary search tree. Should this not be the case, the worst-case complexity would actually be $O(n \cdot k)$, as we would have no guarantees on the height of the tree itself.

5 Deque-based Solution

5.1 Description

The last solution uses a deque in which, at every iteration, we push at the tail the new element entering the window while popping from the head the one leaving it. When inserting at the tail, we remove every tail element which is smaller than the one being inserted. This removes useless elements which could never be maxima and guarantees that the deque's head will host the current window's maximum at each iteration.

5.2 Complexity

The complexity of this approach is $O(n)$, as every element enters and leaves the queue at most once.

6 Comparisons

6.1 Comparing Brute Force Solutions

The two solutions show comparable behaviours for all sizes of n, so we will limit ourselves to the largest one. As can be noticed from the plots, both solutions take roughly the same time in the debug version, while the non-idiomatic solution seems to work better when compiling in release mode, with or without additional optimizations [1]. Indeed, such optimizations seem to actually lengthen the running time, when compared to the simple release build.

6.2 Comparing Heap, BBST and Deque Solutions

Like in the case of the brute force solutions, the plots for various quantities of n have similar dynamics, so we will consider the one with the highest value of n again. In all cases, the BST solution appears to be the slowest, despite having better asymptotic complexity compared to the heap-based one. Perhaps this could be due to the BST being unbalanced. Another thing we notice about the BST solution is that its performance worsens with the increase in window size, but this is to be expected, since operations on the tree take $O(log(k))$. As far as the heap solution is concerned, it starts very slow for low values of k, but then speeds up to match or even surpass the deque-based solution as k increases. The slow startup could be chalked up to the fact that maximal elements which are out of the window need to be popped, and a small window means more popping is necessary. The deque-based solution consistently claims first place and is the most stable for varying values of k, as removing elements which are out-of-range is a $O(1)$ operation anyway. The additional flags and the release mode both bring performance improvements.

7 Conclusions

Overall, the deque-based solution seems to be the fastest in most cases, only sporadically being surpassed by the heap-based one for very large values of k. The slowest solutions are, as expected, the brute-force ones, with the sole exception of very low values of k, for which they manage to match or even surpass the heap and BST solutions. Increasing the value of n leads to an increase in runtime in all cases, unlike k.

7.1 Edge Cases

Whenever n and k coincide, the window has no chance to slide and the returned array only contains a single value. In this case, the brute force solutions are actually the fastest: they simply scan the whole array once and return the maximum, while the other solutions need to populate the respective data structure with the full array. Another interesting edge case is $k = 1$: with such windows, the returned array of maxima is nothing but a copy of the input array. Like in the $n = k$ case, brute force solutions fair better than the other tree, while the BST-based one appears to be the slowest.

[1] RUSTFLAGS='-C and target-cpu=native'

8 Plots

8.1 Brute Force Only

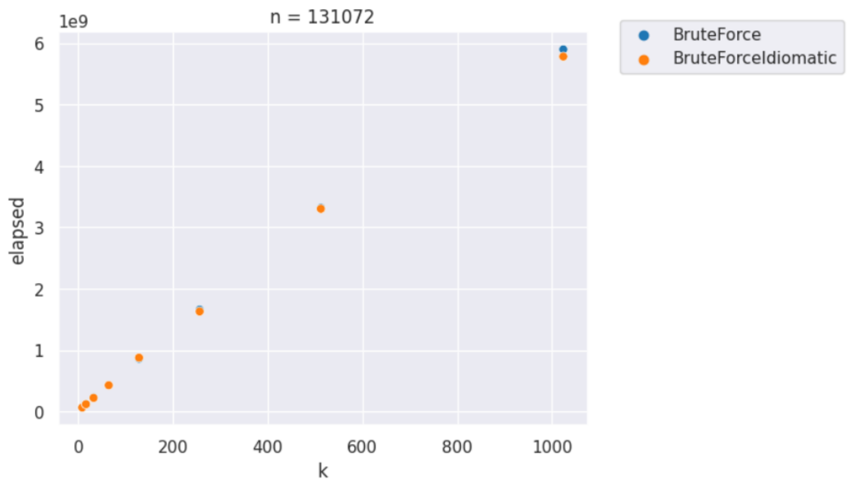

Figure 1: Debug.

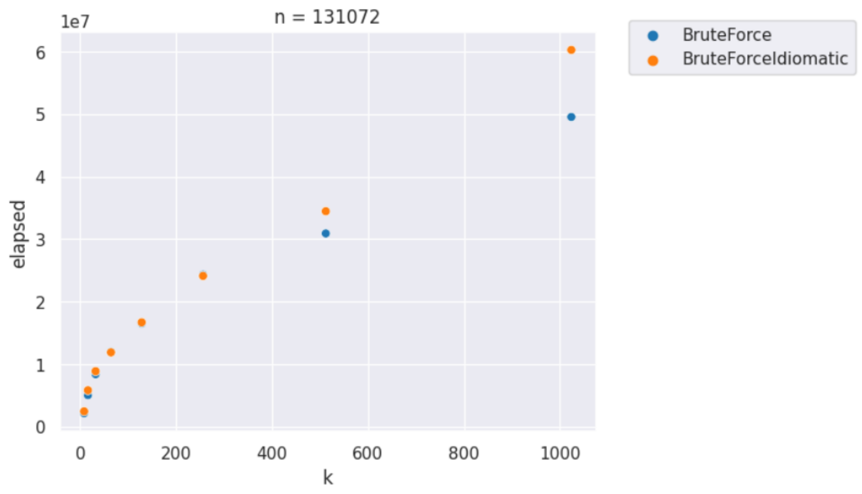

Figure 2: Release.

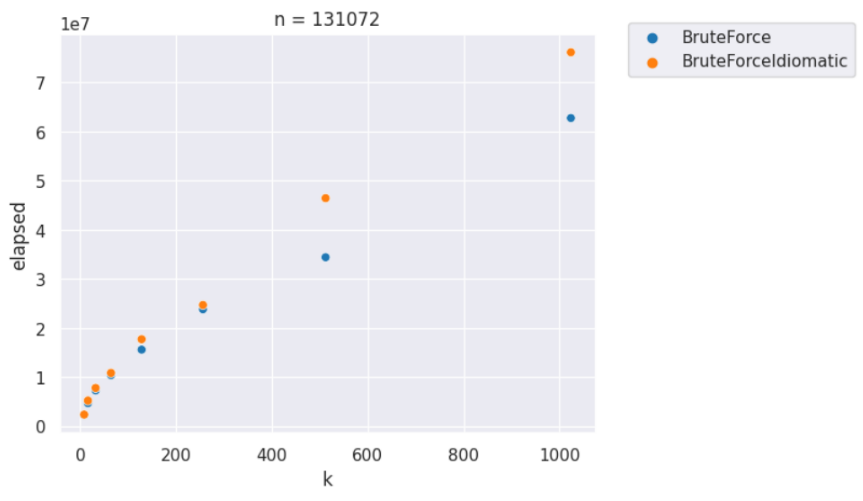

Figure 3: Optimized release.

8.2 Heap, BBST and Deque

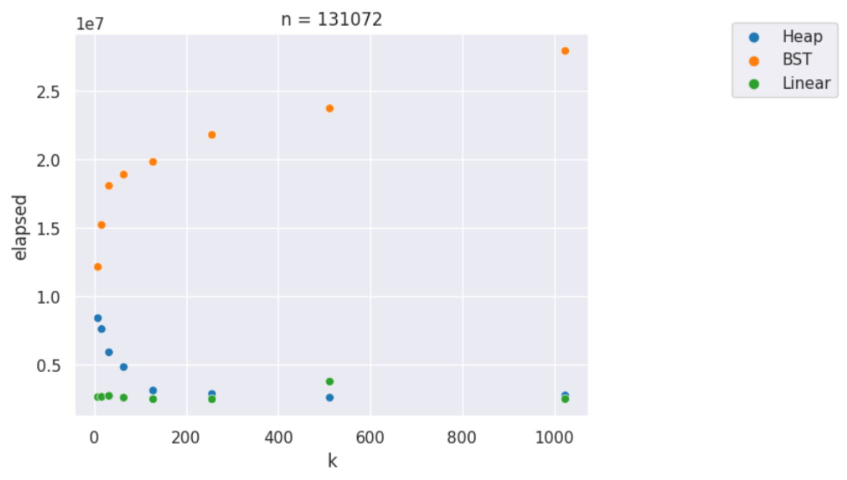

Figure 4: Debug.

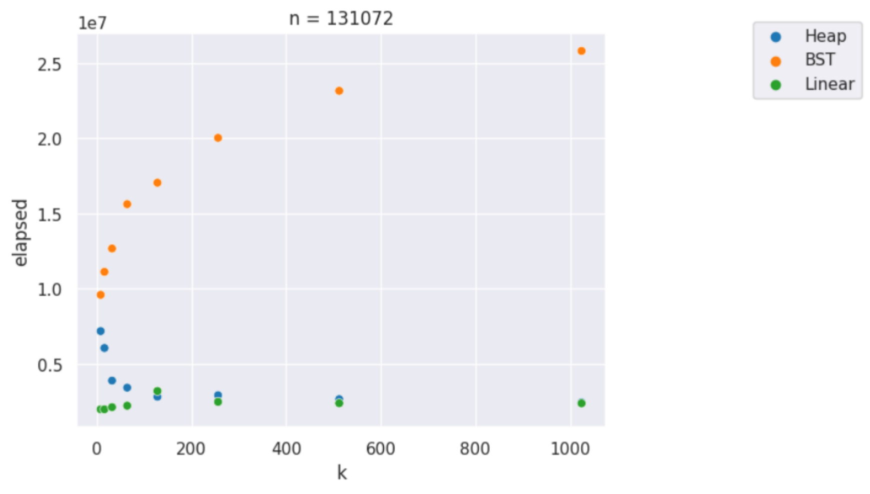

Figure 5: Release.

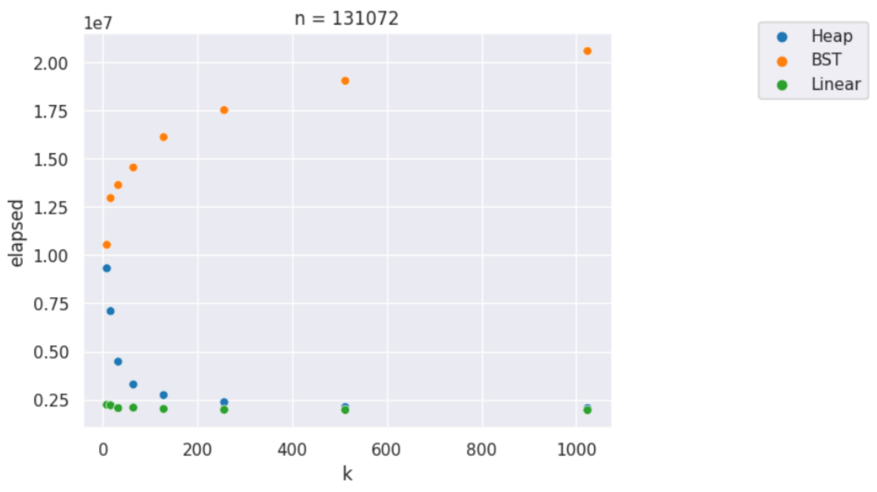

Figure 6: Optimized release.

8.3 All Solutions

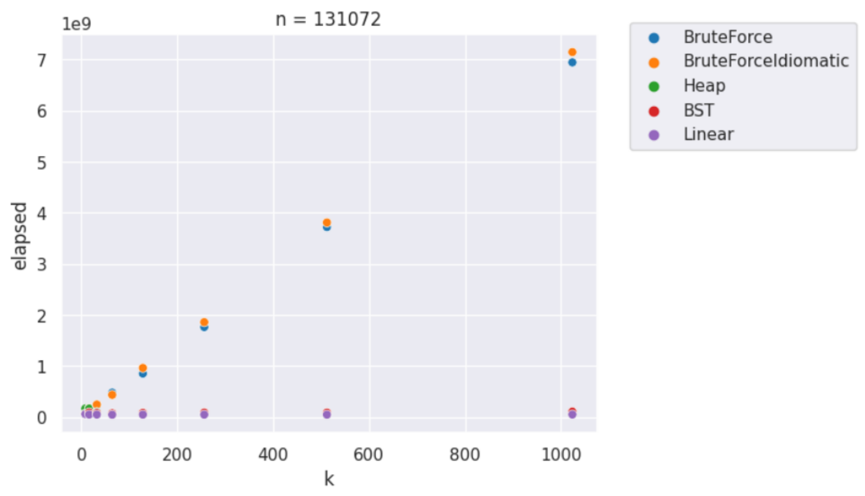

Figure 7: Debug.

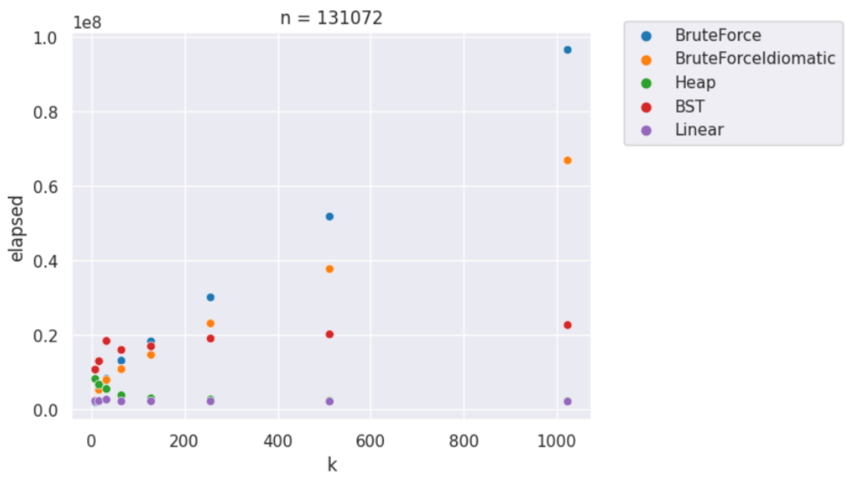

Figure 8: Release.

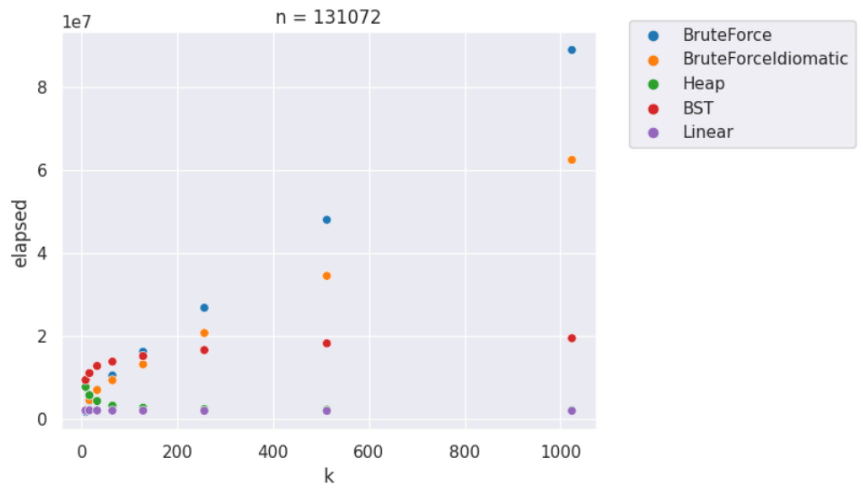

Figure 9: Optimized release.

8.4 Edge Cases

8.4.1 n = k

Notice the rightmost column of dots.

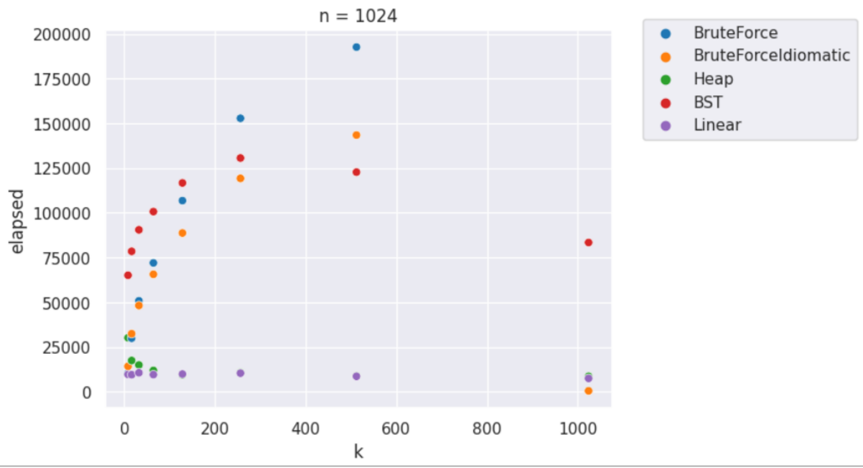

Figure 10: Optimized release.

8.4.2 k = 1

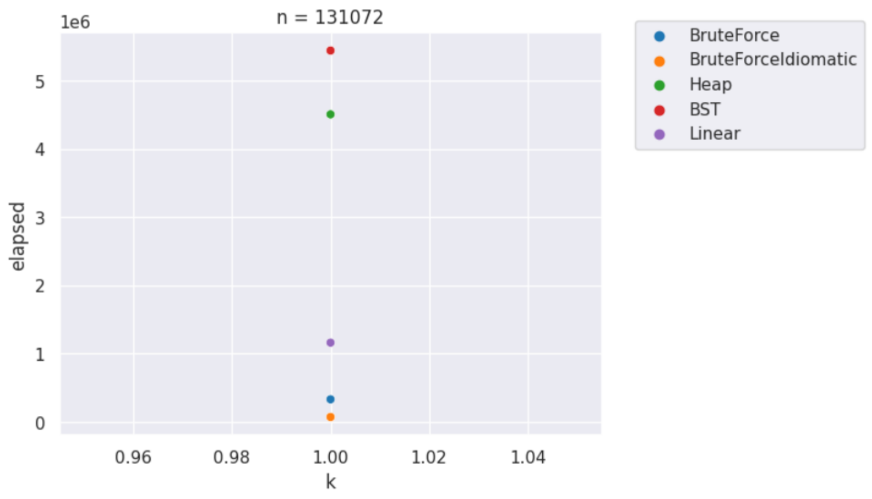

Figure 11: Optimized release.