Hands-on 1: Sliding Window Maximum problem

Eduardo Venturini

Algorithms

Here is a brief description of the implemented algorithms.

Priority Queue based solution

In this solution, we keep a heap containing all the values of the sliding window at a certain time and eventually some additional values before the sliding window, while we iterate through the vector. At each iteration we push the new value at the right of the window in the heap, then we check the maximal value in the heap: if it's a value inside the window, we keep it and go on with the next step; otherwise, we pop the value out of the heap and we repeat this check until we have a value inside the window. To check if a value is inside the window, it is sufficient to attach the index of the element to the element before pushing it inside the heap.

The time complexity of this algorithm is $\Theta(n \log(n))$, indeed the heap can grow up to n elements, so each push/pop operation costs up to $\Theta(\log(n))$, and we repeat these operations $\Theta(n)$ times.

Balanced Binary Search Tree based solution

This solution is similar to the previous one, but a little simpler: instead of using a heap, we use a BBST (capable of handling multiple occurrences of the same element). Furthermore, at each step, in addition to pushing the new element at the right of the window in the BBST, we also pop the element at the left of the window out of the BBST: this way we don't need to check if the maximal element of the BBST is inside the window and we can proceed directly to the next step.

The time complexity of this algorithm is $\Theta(n \log(k))$: the BBST size is always k, so each push/pop operation costs $\Theta(\log(k))$, and we execute these operations $\Theta(n)$ times.

Deque based solution

The main idea behind this solution is the following: given the new element at the right of the window, if it is greater than another element already in the window, we can just ignore the old element when computing the max value of this window and of every next window (sliding the window to the right).

To implement this idea, we keep a deque of some elements in the window, in the same order as in the original vector. At each step, we pop the leftmost element of the deque if it is outside the window, then we pop the rightmost elements of the deque less than the new element at the right of the window and, finally, we push the new element at the right of the deque. At this point, the maximal element in the window is just the leftmost element in the deque.

The time complexity of this algorithm is $\Theta(n)$, indeed we push and pop all the elements of the vector in the deque once.

Results and conclusions

We are going to analyze the performances of the different algorithms and the different compiler optimizations.

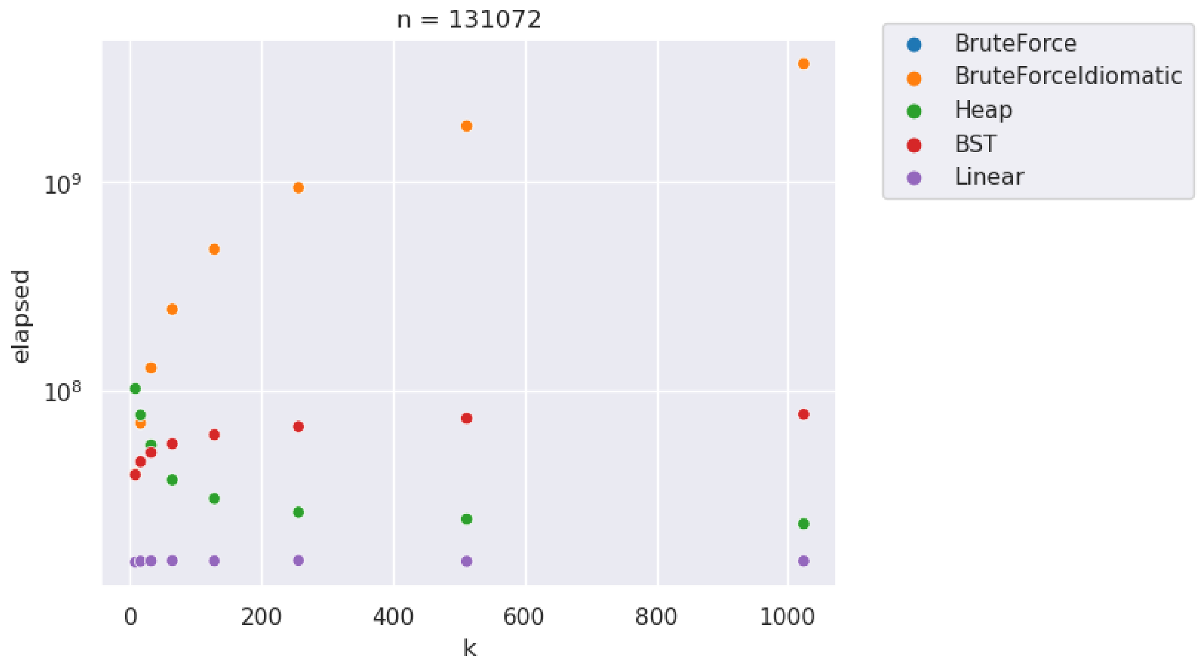
Methods

The conclusions drawn in this section are based on the program compiled with minimal optimizations.

Comparing the `BruteForce` and `BruteForceIdiomatic` solutions, we notice that writing the code in a more idiomatic way does not affect the time complexity.

Comparing the `Heap` and `BST` solutions, we notice that a `BinaryHeap` is faster than a `BinarySearchTree`: in fact, when $\log(k)$ and $\log(n)$ are comparable, the `Heap` solution is 2-3 times faster than the `BST` one. Only for small values of k and big values of n the `BST` solution is faster than the other one. We also notice that the `Heap` solution, whose time complexity does not depend on k, is much faster for big values of k: this is probably because, in the general case, lesser values need to be popped, although in the worst-case-scenario all values need to be popped.

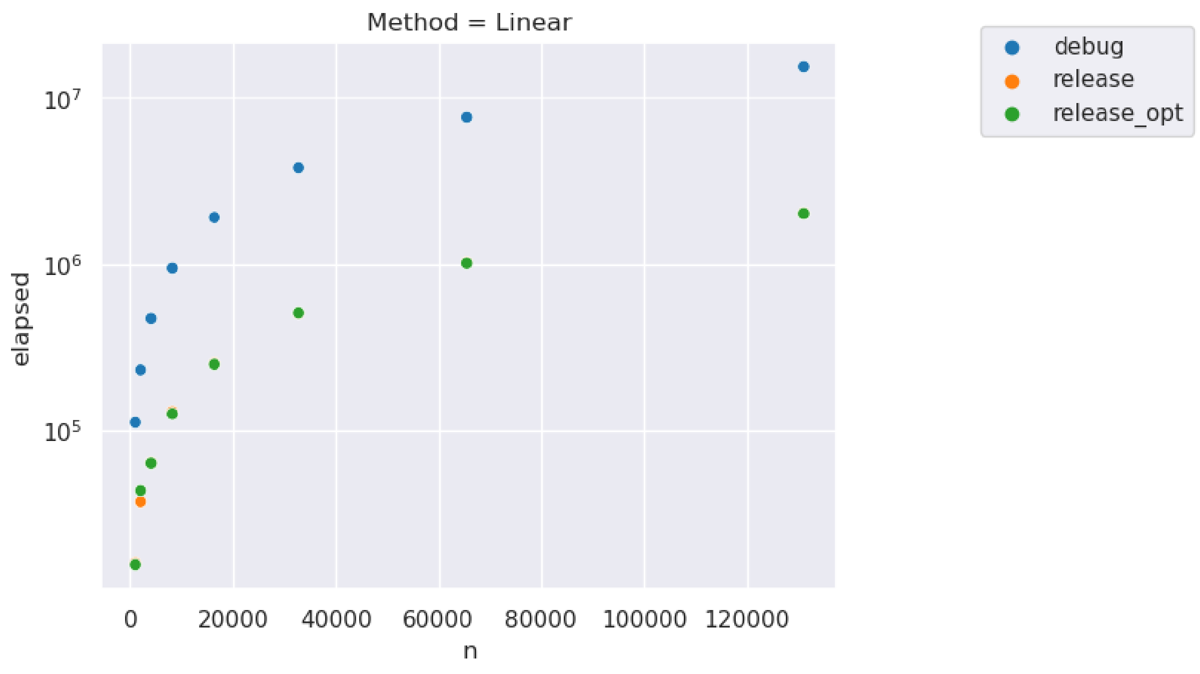
In general, the brute-force solutions are far slower than all the other ones, and the linear solution is always the best one.



Compiler optimizations

We notice that using the flags `RUSTFLAGS='-C target-cpu=native'` for the compilation in release mode does not affect the performance of the program.

On the other hand, compiling the program in release mode instead of debug mode improves the performance by a factor of ~10. This improvement is more evident in the brute-force solutions and less in the solutions using more complicated data structures, probably due to the easier vectorization of the operations.


Method = Linear
elapsed
10^7
10^6
10^5
0
20000
40000
60000
80000
100000
120000
n
debug
release
release_opt