Sliding Window Maximum

Competitive Programming and Contests

Silvio Martinico

October 16, 2022

Abstract

In this report we analyze and compare different solutions for the Sliding Window Maximum problem (finding, in an array of length n, the maximum for each subarray of a fixed length k), implemented in `Rust`. We have two brute force implementations (time complexity $\Theta(nk)$) and we compare them with three theoretically better solutions: the first one relies on a heap (priority queue) and its worst-case asymptotic time complexity is $O(n \log n)$; the second one exploits a BST and has a worst-case asymptotic time complexity of $O(nk)$; finally, we present a linear time ($\Theta(n)$) solution based on a deque.

Contents

1 Heap

Let A be the array we are working with and $n = |A|$. The idea of the heap implementation is the following: we start loading the first window (the first k elements of A) in a max-heap; more precisely, we insert in the heap the pairs $\langle A[i], i\rangle$.

The advantage of the heap is that we can find the maximum element in constant time (it's the root). However, removing the maximum element requires $\Theta(\log n)$ time. So, at each iteration, after we have inserted the new element in the heap (in $\Theta(\log n)$ time), instead of removing the element which leaves the window, we keep it and, whenever we take the maximum element, we check if it is still in the window (thanks to the index we stored): if it is, we just take it, otherwise we remove it and we repeat the operation, until we find in the root an element which is in the window.

1.1 Asymptotic time complexity analysis

At each iteration (for a total of $n - k + 1$ iterations) we have to insert one element in $\Theta(\log n)$ time. So the time complexity is at least $\Theta(n \log n)$. Moreover, we may do some deletions. But, how many? If we think about it iteration-wise, we risk to over-count them. So, it's sufficient to just notice that, the total number of deletions can not be greater than n because each element of A could be removed just one time. So, the worst-case asymptotic time complexity is $O(n \log n)$.

2 Binary Search Tree

The binary search tree implementation is the simplest one. We just load the tree with the initial window and then, at each iteration, we remove the element which leaves the window, insert the new element and take the maximum of the updated tree. Each of these three operations has a time complexity of $O(k)$ (because we keep the size of the tree equal to k).

The worst-case asymptotic time complexity is therefore $O(kn)$.

3 Deque

The deque approach relies on a double ended queue where we insert in the tail and remove from the head of the queue. As in the heap case, we actually insert in the deque the pairs $\langle A[i], i\rangle$.

At each iteration, we remove the head elements which are not in the window anymore (thanks to the indexes we are storing in the queue) and also every tail element which is smaller than the new element that is entering the window. At this point, we insert the new element entering the window in the tail of the queue and, if we already loaded the first window into the deque, we can take the head element as the maximum of the current window.

The correctness of this algorithm was shown in class and relies on these two facts:

- The elements in the deque are sorted in decreasing order
- At each iteration, the deque contains all and only the rightLeaders of the current window

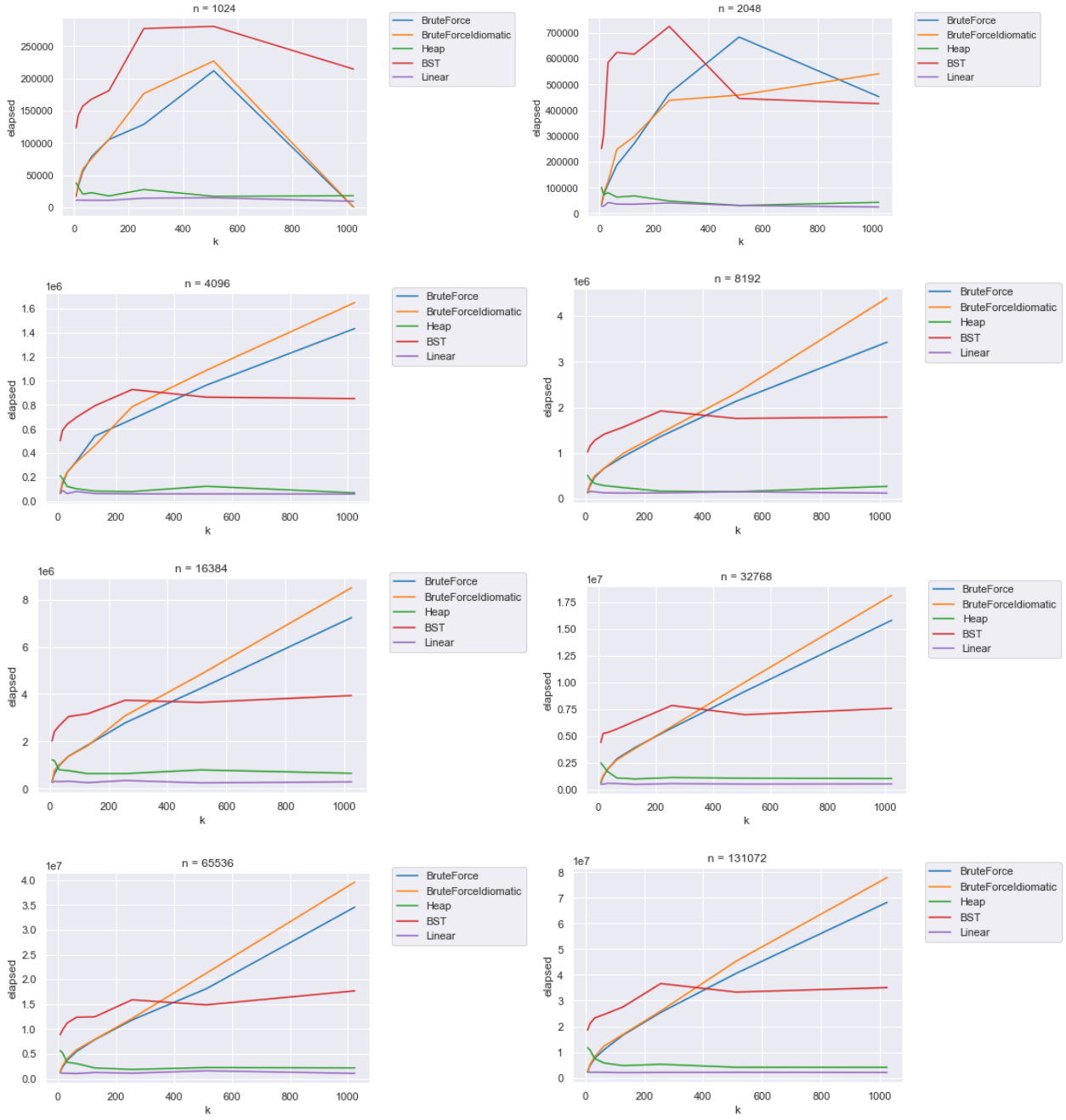
3.1 Asymptotic time complexity analysis

Since the cost of *pop* and *push* operations is constant and we do only one *push* at each iteration, the only thing that could afflict negatively the cost of this algorithm is the number of *pop* operations at each iteration, which is at most n. However, this cost is amortized in the other operations. In

fact, the maximum number of pop operations in the whole algorithm is limited by the number of the elements of A (when an element leaves the deque, it never comes back). So, the total cost is clearly $\Theta(n)$ (it is not a worst-case complexity since we have to do at least n operations to load every element in the deque).

4 Experiments

We tested the five algorithms on various instances. In each figure, we fixed n and plotted the elapsed time (in nanoseconds) with respect to the size of the window k.
Every algorithm was ran with the command `cargo run --release`.



4.1 Considerations on the results

As we can see from the plots, the deque algorithm always outperforms every other algorithm. The only case where it is not the best algorithm is when, for $n = 1024$, we have $k = n$. In this case. the two brute force algorithms becomes just algorithms which scan the array and find the maximum element, so only in this case they are linear and they don't waste time in creating other data structures.

At the "second place", we have the heap algorithm. In fact, the analysis we have done is just a worst-case analysis; in practice, it works very well because it's not mandatory for it to delete elements, so it could delete few elements at each iterations or, even, does not delete any element.

The BST algorithm seems to have not good performances. In fact, for small values of n and k, its performances are very bad, and, specially, it works worse than the two brute-force algorithms.
For the reasons we explained above, in practice, the heap algorithm is better than the BST one and, for big values of k, $\log n$ is also smaller than k, so obviously the heap algorithm works better. When n and k increase, the BST algorithm outperforms the two brute-force algorithms, because, for bigger sizes, the asymptotic complexity becomes more reliable. Moreover, unlike the two brute force algorithms, the complexity $O(kn)$ is a worst-case complexity, and, as we can see, it is not reached in practice. This worst-case complexity is given by the fact that the maximum height of the tree is k, but, this happens only if the tree is very unbalanced; in practice, we expect that the tree is not totally unbalanced (but neither it is perfectly balanced). The fact that, for small values of n and k, the BST algorithm is the worst one, is probably due to the implementation details of the `BinarySearchTree` in `Rust` and to the hidden constants in the asymptotic time complexity.

Another thing we want to observe is that, for big values of k, the BruteForce algorithm works slightly better than the BruteForceIdiomatic.