Sliding Window Maxima: solutions and performance analysis

Giuseppe Muntoni

Contents

1 Introduction

This report aims to describe four possible solutions to the sliding window maxima problem and compare the performances of those solutions implemented in Rust.

2 Solutions

2.1 Brute force

The first solution is trivial: for each window, the algorithm searches for the maximum reading of all the elements in the window with a "for" loop. This solution has the problem that reads the same elements more than once because the actual window differs from the previous one in only one element. The idea is to use a support data structure that gives us the possibility to search for the maximum inside the window without reading each element of the window.

This solution has complexity $\Theta(nk)$, where n is the size of the array and k is the size of the window.

2.2 Heap

The heap data structure gives us an efficient implementation of priority queues. This solution, in particular, uses a max heap so that we can read the maximum in constant time.

Instead of doing deletion every time, the idea is to insert inside the heap couples $(value, index)$ where the value is the element of the array and the index is the position in the array. At each iteration, the algorithm slides the window and inserts a new element inside the heap. Afterward, read the maximum and, if the maximum is in the actual window, report it and go to the next iteration, otherwise extract the maximum and retry until it finds a maximum inside the window.

This solution does exactly n insertion and at most n deletions of the maximum, which cost $\Theta(log(n))$ both. The reading of the max is $\Theta(1)$. So, the overall cost is $\Theta(nlog(n))$

2.3 Binary search tree

An alternative solution uses the binary search tree. At each iteration, the algorithms insert inside the tree the element that enters the window and removes the element that exits from the window. Afterward, it searches for the maximum and reports it. In this case, the tree is always of size k, where k is the size of the window, so that the complexity is $\Theta(nlog(k))$, because each insertion, deletion, and max operation run in $\Theta(log(k))$ time, if the tree is balanced. The implementation used in Rust, i.e., *binary search tree*, is not guaranteed to be balanced, so in the worst case, the complexity is $\Theta(nk)$, as in the brute force solution.

2.4 Deque

A deque is a list on which we can push and pop both on the front and on the back, in constant time. The algorithm inserts inside the deque couples $(value, index)$, as in the previous solutions. The algorithm pops from the back while the element popped is smaller than the actual one. Then insert the actual element on the back, remove from the front elements that are not anymore inside the window, and report the front that contains the maximum of the actual window. This algorithm does exactly n insertions and at most n deletions that are done in constant time, so the overall complexity is $\Theta(n)$.

3 Performance comparisons

In this section, are analyzed the performances of the Rust implementation of the solutions described above. All comparisons are done with the optimized version of the code by the compiler with the flag *RUSTFLAGS='-C target-cpu=native'*.

3.1 Best case

Below are described the best cases for each solution.

3.1.1 Brute force

For the **brute force** solution the best case happens when $k = 1$, making the algorithm linear. In addition, in this case, and in general with really small k, this solution is the fastest one, because we do everything in place, without spending time allocating and using other data structures. This case requires only $0,04$ milliseconds with $n = 65536$.

3.1.2 Heap

In the **heap** solution the best case happens when the array is increasingly ordered, because, the algorithm never does deletions of the maximum, as it is always inside the window. This case requires, independently from k, approximately $0,8$ milliseconds, with $n = 65536$, which is really slower than the best case of brute force.

3.1.3 BST

For the **bst** solution the best case happens when $k = 1$ because the tree have only one node, so the operations are constant.

3.1.4 Deque

For the **deque** solution the best case happens when the array is decreasingly ordered and k is $O(n)$ because thanks to the decreasing order the algorithm never does pop on the back, and thanks to the size of k the algorithm does a minimum amount of pop from the front. This case requires approximately $0,2$ milliseconds, with $n = 65536$, which is better than the heap one.

3.2 Worst case

Below are described the worst cases for each solution.

3.2.1 Brute force

For the **brute force** solution the worst case happens when $k = O(n)$, giving a quadratic solution. In particular, when $k = n/2$ the completion time is 2 seconds, the worst possible.

3.2.2 Heap

For the **heap** solution the worst case happens when the array is decreasingly ordered because the algorithm does exactly $n - k$ deletions in this case. The completion time is between 1 and 2 milliseconds depending on the size of k, always with $n = 65536$. This is the fastest solution in the worst case.

3.2.3 BST

In the **bst** solution the worst case happens when the array is increasingly ordered or decreasingly ordered because it becomes a linked list. This solution is slower than any other solution, and in particular, slower than the brute force solution, because the array representation is contiguous in memory, and there is spatial locality so that caches can be used. Instead, a tree has elements sparse in memory. This is the slowest solution in the worst case.

3.2.4 Deque

For the **deque**, apart from the best case that is a bit faster than other cases, the completion time does not vary so much, because the number of deletions varies from $n - k$ to n, so $O(n)$. So, there isn't a real worst case.

The table 1 shows a comparison of the completion time for the best and the worst case.

	Brute force	Heap	BST	Deque
Best case	0,04 ms	0,85 ms	2,1 ms	0,23 ms
Worst case	2026 ms	2 ms	3526 ms	N.d

Table 1: Comparison of the completion time in the best and the worst case.

3.3 Mean case with random vectors

The mean case is generated with vectors of random values. Every solution is tested with large n and with various sizes of k, from really small, to near n. There are also tests with n small.

3.3.1 Small arrays

With small arrays, the heap solution in general is faster than the BST one. The brute force solution is fast when k is really small. The deque solution in general is better than every other solution, but with high k, the heap solution is comparable.

The table 2, shows the difference of completion time, in nanoseconds, between the solutions, variating k, and with $n = 1024$.

k	Brute force	Heap	BBST	Deque
4	7125	90750	160083	17708
8	15208	64875	152166	17750
16	29583	51958	178500	23125
32	98083	28541	236041	19208
64	207625	21125	280625	16750
128	528125	10500	233125	16416
256	832375	19958	214833	12000
512	1000708	7000	198291	11041
1024	3500	12208	212166	10916

Table 2: Comparison of the completion time with small arrays variating k.

3.3.2 Variations of the window size

Consider n sufficiently large, in particular $n = 65536$. How does the completion time vary with different sizes of the window, i.e., variating k?

In the **brute force** solution, as we can expect incrementing k, increments the completion time. But when $n - k$, is really small, the number of windows, i.e., $n - k + 1$, is really small and consequently, the maximum searches inside the windows are few. For those reasons, the plot representing the completion time in function of k is the one in figure 1.

In the **heap** solution, the completion time is inversely proportional to k. In fact, when k is small, it is more probable that the algorithm does more deletions because the maximum reported is outside the window. Instead, when k is high, happens the opposite, i.e., is more probable that the max is inside the window.

Regarding the **BST** solution, is directly proportional to k. In fact, with small values of k, the tree is really small, so the operations are really fast. With higher k the algorithm becomes slower because the tree is larger. The maximum completion time happens when both k and $n - k$ are sufficiently large because the algorithm must do operations on a large tree, and also a large number of deletions, i.e., $n - k$. So, with the maximum value of k, i.e., when $n - k$ is near to zero, the performance increases a bit, because the algorithm does almost no deletions, having a small total number of windows.

Regarding the **Deque** solution, the completion time does not vary so much with k, because it not depends on it.

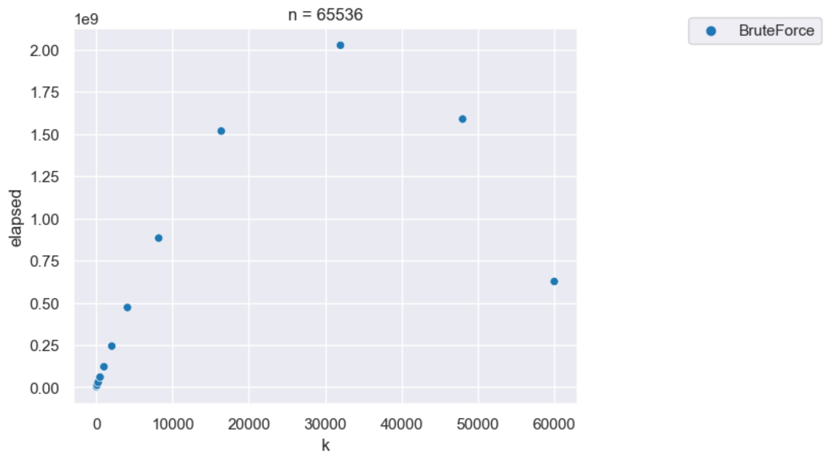

Figure 1: Mean case of the brute force solution, variating k

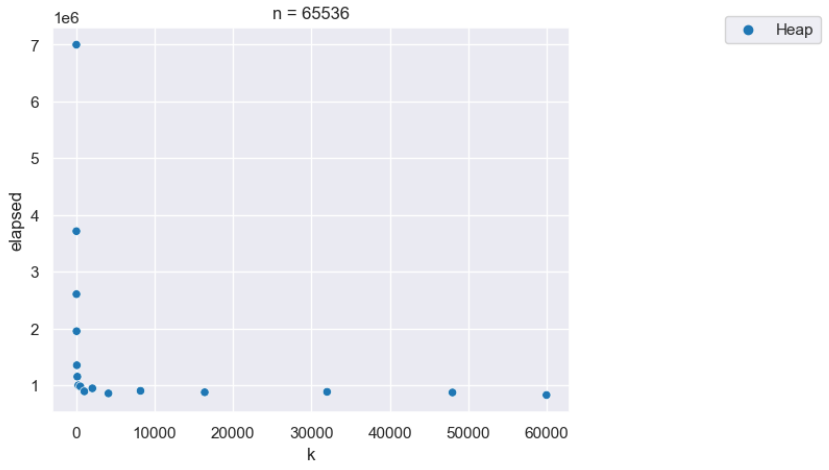

Figure 2: Mean case of the heap solution, variating k

3.4 Is the deque solution always the best one?

In general yes. But there are cases in which other solutions have the same completion time or are better:

- The **brute force** solution with $k = 1$ is the fastest solution between all the solutions discussed in this report. This solution is really fast also when $k = n$.
- The **heap** solution has par performance when n is small and k is high and has comparable performance with n large and k too.

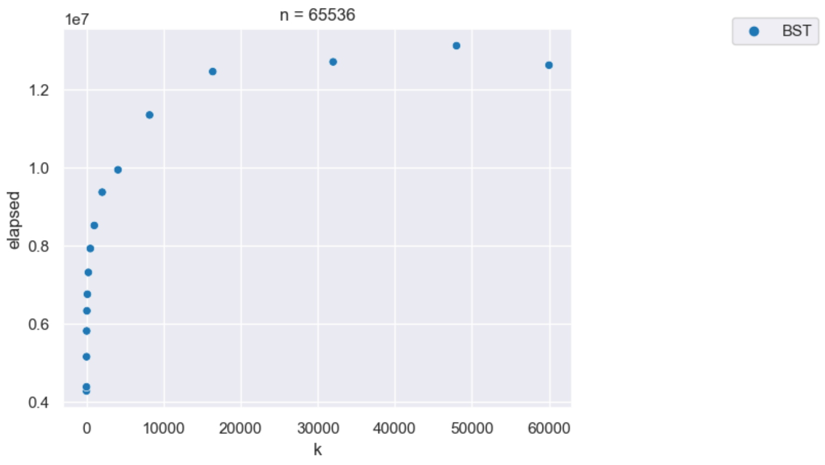

Figure 3: Mean case of the bst solution, variating k

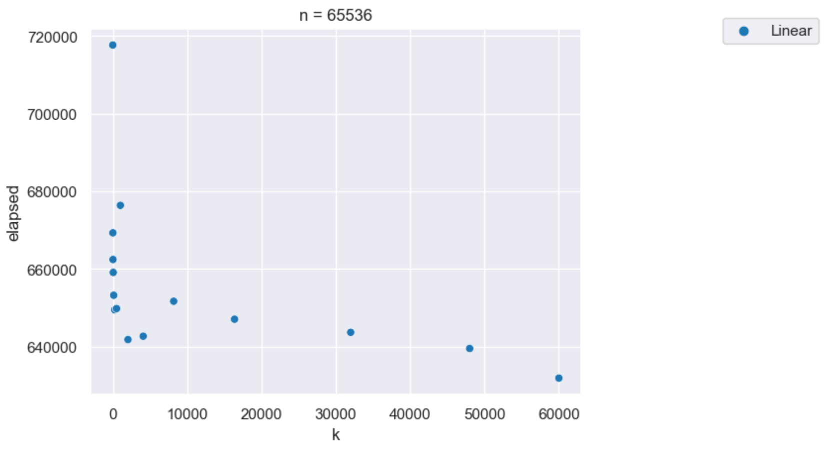

Figure 4: Mean case of the deque solution, variating k

3.5 Heap vs BST Solution

An interesting case is the heap solution that, in general, is really faster than the BST solution, e.g., with $n = 65536$ and $k = 1024$ the heap solution takes $0,8$ milliseconds and the BST takes 8 milliseconds, so $10X$ faster. The heap solution is faster because the heap is stored as an array, so contiguous in memory and can take advantage of the spatial locality and use the cache. Instead, the binary search tree is stored with pointers to sparse memory. The tree is not guaranteed to be balanced, but in the mean case, this is expected.

3.6 Brute force vs brute force idiomatic

The brute force solution is implemented in Rust in two ways: one that uses a standard *for* loop to scroll the array and generate the windows and an idiomatic version that uses a standard function to return all the windows inside an array. The result is that the idiomatic version is 15% faster than the normal one.

3.7 Performance impact with Rust compiler optimizations

The compilation of the Rust implementation is done in three ways:

1. Normal **unoptimized** compilation without any flag
2. **Optimized** version with option *–release*
3. **Optimized** version with flag *RUSTFLAGS='-C target-cpu=native'* that optimize the code using all the features of the CPU

The result is that the completion time of (2) and (3), for every solution is really smaller than the unoptimized version. The (2) and (3) compilations gives comparable performances.