Hands-on 1 - Competitive Programming

Chiara Molinari

August 2023

1 The problem

You are given an array of integers nums (size n), there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.
Return the max sliding window.

2 Description of the considered algorithms

2.1 Brute Force solution

We iterate over windows of size k. For each window, the maximum is calculated.
We consider the solution written in an idiomatic way, i.e. using the 'windows' method applied on a vector type that gives us iterators over windows of a defined size.
Alternatively, we can range over the possible windows manually.
Complexity: we scan the vector once and, at each step, the max calculation requires $O(k)$ operations, so the total time complexity is $O(nk)$.

2.2 Heap solution

We consider a heap that stores couples (value, index), sorted by 'value'.
We loop through the vector and at each step (let's say i) we consider the max value and its index (they are stored in the head of the heap).
While index $\leq i-k$, it means that its value is out of the window we are considering, so the couple is removed from the heap.
Then a new value is added to the heap.
If the window is complete (i.e. $i \geq k-1$) we add the max value to a 'result' vector.
Complexity: there are n (value, index) couples, giving a heap of maximum size n. Also, each of these couples will be added and removed from the heap at most once. These two operations will cost $O(\log(n))$ time, giving a total time complexity of $O(n\log(n))$.

2.3 Binary search tree solution

This solution is similar to the previous one.
This time we use a BST to store couples (value, count), where count represents the count of appearances of the value in the current window.
We loop through the vector and at each step we take out from the BST the value that just become too old to be in the new current window.
This is done by possibly removing an element from the BST (if count was 1) or by decreasing its count.
We insert the new value or update its count.
As before, if the window is complete, we add the max value (head of the BST) to a 'result' vector.
Complexity: since the BST only stores values appearing in the current window, its max size is k. There are n

steps and at each step we will be removing/updating the BST $O(1)$ times, with operations costing $O(\log(k))$ time. So the total time complexity will be of $O(n \log(k))$.

2.4 Linear solution

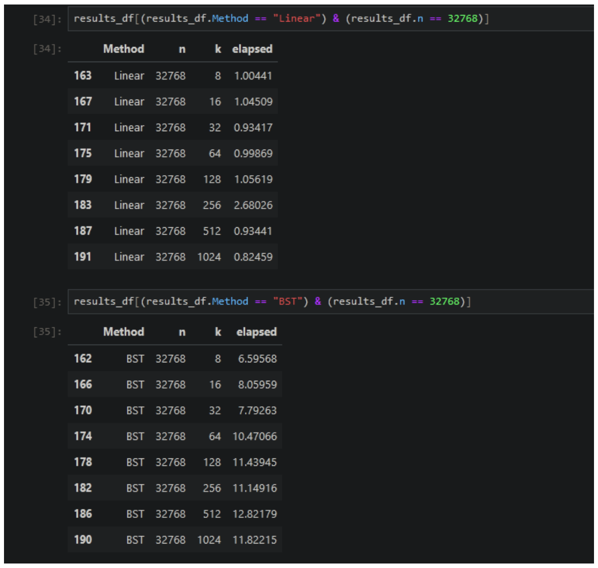
This time the data structure will be a vector called 'queue', that stores some indexes of the vector 'nums' in increasing order.
As before we scan the original vector nums. We have that the values of nums for the indexes in 'queue' will be in decreasing order.
At each step, we remove the indexes out of the current window from 'queue'. Since they are stored in increasing order, this means removing the head of 'queue', until the index is big enough.
Then we consider the newly explored value nums[i]. We remove from 'queue' all indexes j such that $nums[j] \leq nums[i]$. Because of the previously mentioned decreasing order, this means removing the end of 'queue' until it is necessary.
Lastly, we insert it in the queue. Note that these operations preserve the increasing order of the indexes and the decreasing order of the corresponding values.
As before, if the window is complete, we add the max value (head of 'queue') to a 'result' vector.
Complexity: over all the steps, each index will be inserted and removed from queue at most once. These operations require $O(1)$ time, giving a total time complexitiy of $O(n)$.

3 Optimization of the code

In the building process, the debug and release modes are compared, resulting in significantly quicker results if the latter is used. Instead, other flags did not affect the computational time. For this reason, from now on we only consider the case where the building is done with "cargo build –release".

4 Timing of all solutions

Here we compare the execution timings for all the described methods. The timings are in $10^7 ms$.
As the timings for the idiomatic and non-idiomatic brute force algorithm are really close, from now on we only consider one of them (specifically, the idiomatic one).
Here we have the execution time for $n = 32768$, the biggest value considered in the experiment.



```
[34]: results_df[(results_df.Method == "Linear") & (results_df.n == 32768)]
```

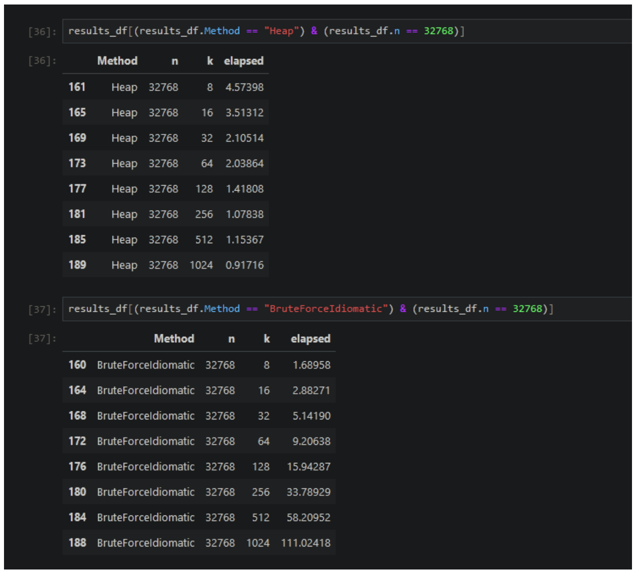
[34]:

	Method	n	k	elapsed
163	Linear	32768	8	1.00441
167	Linear	32768	16	1.04509
171	Linear	32768	32	0.93417
175	Linear	32768	64	0.99869
179	Linear	32768	128	1.05619
183	Linear	32768	256	2.68026
187	Linear	32768	512	0.93441
191	Linear	32768	1024	0.82459

```
[35]: results_df[(results_df.Method == "BST") & (results_df.n == 32768)]
```

[35]:

	Method	n	k	elapsed
162	BST	32768	8	6.59568
166	BST	32768	16	8.05959
170	BST	32768	32	7.79263
174	BST	32768	64	10.47066
178	BST	32768	128	11.43945
182	BST	32768	256	11.14916
186	BST	32768	512	12.82179
190	BST	32768	1024	11.82215



```
[36]: results_df[(results_df.Method == "Heap") & (results_df.n == 32768)]
```

[36]:

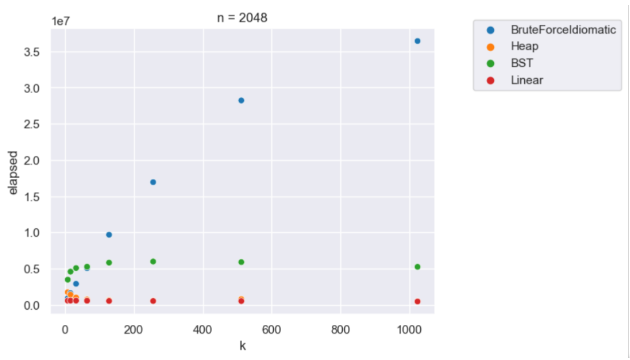
	Method	n	k	elapsed
161	Heap	32768	8	4.57398
165	Heap	32768	16	3.51312
169	Heap	32768	32	2.10514
173	Heap	32768	64	2.03864
177	Heap	32768	128	1.41808
181	Heap	32768	256	1.07838
185	Heap	32768	512	1.15367
189	Heap	32768	1024	0.91716

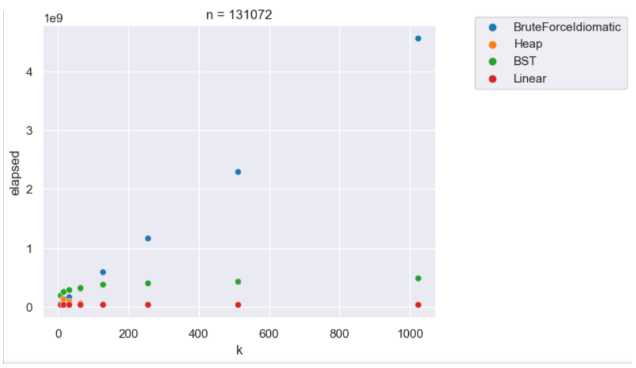
```
[37]: results_df[(results_df.Method == "BruteForceIdiomatic") & (results_df.n == 32768)]
```

[37]:

	Method	n	k	elapsed
160	BruteForceIdiomatic	32768	8	1.68958
164	BruteForceIdiomatic	32768	16	2.88271
168	BruteForceIdiomatic	32768	32	5.14190
172	BruteForceIdiomatic	32768	64	9.20638
176	BruteForceIdiomatic	32768	128	15.94287
180	BruteForceIdiomatic	32768	256	33.78929
184	BruteForceIdiomatic	32768	512	58.20952
188	BruteForceIdiomatic	32768	1024	111.02418

It is already evident that the brute force solution indeed shows a growing computational time as k increases, while the linear solution timing doesn't depend on k as expected.
We also note that the linear and brute force solutions seem to perform better on small values of k.
This indeed happens for different values of n (but when n is small the brute force solution is a bit quicker).
This could be because of the simpler data structures required in these algorithms.
The following graphs represent the timing for fixed n. The two following chosen n are the smallest and the largest considered values in the experiment.

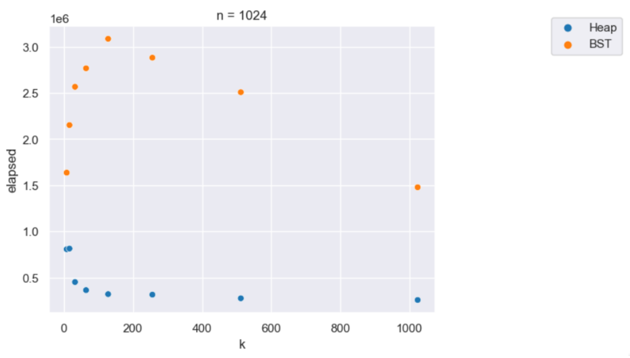


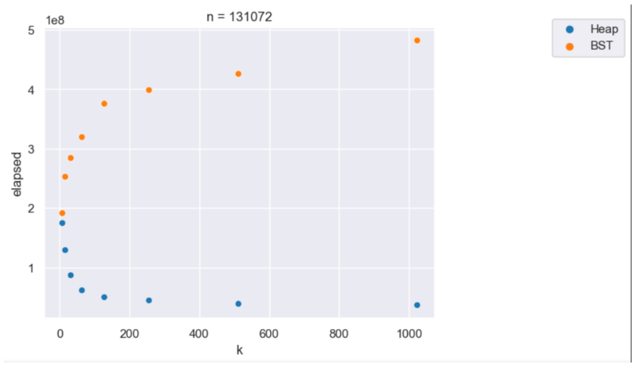


While the second is just a visual representation of the previous table, the first is showing that the curves for a much smaller n are similar on a large scale.

5 Comparison: heap and BST solutions

Now we want to compare more precisely the timing for the heap and BST solution and study their dependence from k.
In the following plots we only keep these two solutions.





In the heap solution, we see that for sufficiently large values of k, there is no more dependence on it k, as expected.
But for small values of k, the time decreases as k increases, meaning that infrequent removals of values from the heap make the solution significantly quicker.
Instead, in the BST solution the dependence on k looks like a log function as expected. Such behavior only appears when n is sufficiently large.
We can also see that for any fixed n, k, the BST solution takes more time than the heap one.
This is probably because the BST is a more complicated data structure to handle, making the heap preferable for values of k and n that are not too big.