Platinum US Open Review

Justin Zhang

6 April 2018

1 Platinum 1 - Out of Sorts

1.1 Problem

Let's define a "partition point" to occur between indices i and $i+1$ if the maximum of $A[...i]$ is less than or equal to the minimum of $A[i+1...]$. Given an integer array of length N ($1 \leq N \leq 10^5$), determine the value of `work_counter` if the `quickish_sort` is run on the array:

```
bubble_sort_pass(A) {
    for i=0 to length(A)-2
        if A[i] > a[i+1]:
            swap(A[i], A[i+1])
}
quickish_sort(A) {
    if length(A) = 1, return
    do {
        work_counter += length(A)
        bubble_sort_pass(A)
    } while (no partition points exist in A)
    divide A at all partition points and recursively quickish_sort each piece
}
```

1.2 Sample Case

```
7   (N=7)
20 2 3 4 9 8 7
```

We start with `20 2 3 4 9 8 7`. After one `bubble_sort_pass`, we get `2 / 3 / 4 / 9 8 7 / 20` (`/` is a partition point), taking 7 units of work.

From here, each of the length one subarrays take 0 work since we return if length is one. Thus, we recurse only on the `9 8 7` portion to get `8 7 / 9`, which takes 3 units of work, then recurse again on `8 7`, taking 2 more units of work.

Thus the answer is 12.

1.3 Solution

First, notice that running `bubble_sort_pass` on many consecutive subarrays is equivalent to running the function on the entire array. This is because the elements to the left of a partition point are always less than those on the right, so the bubble sort pass won't move any elements from the left of a partition point to the right. However, the only issue is that this counts subarrays of length one as part of `work_counter`, whereas our base case specifically makes them of 0 work.

Thus, it in order to find `work_counter`, we need to be able to find the number of bubble sort passes it takes (on the whole array) before a given element is its own subarray – that is, before both sides of the element have partition points. This will, clearly, be the maximum of (the number of passes it takes to establish the left partition point) and (the number of passes it takes to establish the right partition point). After this many passes, the element no longer counts toward `work_counter`. As such, we can sum this value for each element to get our final answer.

How can we find this for each element?

Let's consider what happens when we do a single `bubble_sort_pass`. If an element is at an index greater than where it should be (in the sorted subarray), it gets moved backward one index. Remember, a partition point is established between $i-1$ and i when the smallest i elements are the first i entries of the subarray. Thus, the number of bubble sort passes it takes to establish the partition point will be $j-i+1$, where j is the maximum index of any of the smallest i elements in the subarray (we must move the element from j to below the partition point between i and $i+1$).

We can calculate the number of passes it takes for each partition point to be established easily: we sort the array by values, tagging each element by its original index. Then, we pass through the array, and for each index i, the maximum index of all elements less than or equal to i gives us the j as referenced in the previous paragraph.

We can do another pass to find the maximum of adjacent partition points, to determine when each element will stop contributing to `work_counter`.

Note: remember to account for the corner case of when a given element is already partitioned. Because it's a do-while loop, it contributes 1 to `work_counter`, not 0.

2 Platinum 2 - Train Tracking

2.1 Problem

Given an integer array of length N ($1 \leq N \leq 10^6$), you're to find the sliding window minimums of length K ($1 \leq K \leq N$) – that is, the minimum from $0...K-1$, $1...K$, and so on until $N-K+1...N$.

However, there is a catch. You're only allowed to read the array (sequentially) twice, and you can only use a length 5500 integer array as memory (between every element read). Additionally, you can only read and write to this array $25 * 10^6$ times. You may output the output to any range at any time, as long as you output all of them by the end of the second pass. These restrictions are imposed via an API.

2.2 Sample Case

```
10 3    (N=10, K=3)
5       -> 5, minimum among 5, 7, 9
7       -> 2
9       -> 0
2       -> 0
0       -> 0
```

```
1       -> 1
7       -> 3
4       -> 3
3
6
```

2.3 Solution

In order to solve this problem, we must clearly use information from the first pass in our second pass. This information must be of $O(\sqrt{N})$ memory (specifically, $\leq 5.5\sqrt{N}$). This notably eliminates the $O(N)$ time and space sliding window minimum algorithm we talked about earlier this year.

For convenience, let's define a function $f(i)$ that is the (minimum) index of the minimum element from the range $i...i+K-1$. Notice that this is an increasing function.

On our first pass, let's calculate $f(0), f(\sqrt{N}), f(2\sqrt{N})...f(N)$ (excluding some at the end depending on K). We can do this in $O(N)$ time and $O(\sqrt{N})$ memory, by storing the location of the minimum element in each $\sqrt{N}$ block of the array in a monotonically increasing deque. Specifically, as we iterate forward through the first pass, we maintain the index of the minimum element in the $\sqrt{N}$ block we're in, by using two spots in memory per block. When we reach the end of a $\sqrt{N}$ block, we pop the top of our deque while the top element is greater than our current minimum, and then push the current minimum on the top of a deque. At the same time, whenever we reach an index i such that i is the end of a range that begins in a multiple of $\sqrt{N}$ (in other words, $\sqrt{N}$ divides $i-K+1$), we pop from the back of the deque until we reach a minimum that falls in the range (i.e. the back of the deque has an element whose index is between $i-K+1...i$). The index of the element at the back of the deque will then be $f(\frac{i-K+1}{\sqrt{N}})$. If we repeat this throughout the first pass, we calculate the desired $f(0), f(\sqrt{N})...f(N)$.

How can we use this information for the second pass?

Notice that for each $\sqrt{N}$ bucket (say, from 0 to $\sqrt{N}$), f is increasing and between $f(0)$ and $f(\sqrt{N})$. Let's figure out how we can calculate all values of f from $0...\sqrt{N}$. We can then repeat this for each bucket to get our final answer.

We can take the minimum of the intersect of the ranges from $0...K$ and $\sqrt{N}...\sqrt{N}+K$. Because this intersect is in every interval that begins from $0...\sqrt{N}$, the minimum value in this range is all we need – we don't need to iterate over it more than once. That is to say, all values $f(0), f(1), f(2)...f(\sqrt{N})$ will be an index such that the index's value is less than or equal to the minimum value in the intersect interval. As an implementation detail, we further do intersect of $f(0)...f(\sqrt{N})$ on the aforementioned interval because we don't want to iterate further than $f(\sqrt{N})$ – no f for values of $0...\sqrt{N}$ can be greater than $f(\sqrt{N})$, so we would start to iterate into answers for the next $\sqrt{N}$ bucket. Let's call the intersect of the three ranges the *intersect interval*.

Our *region of consideration* is $0...\sqrt{N}+K$, since this interval contains all the elements we would have to consider for the values $f(0), f(1), f(2)...f(\sqrt{N})$. After accounting for our intersect interval, our remaining region of consideration (which may consist region of two intervals, to the left of the intersect interval and to the right) is of length $O(\sqrt{N})$. This is because the length of the intersect interval is $K-\sqrt{N}$, and our region of consideration before subtracting the intersect interval is of length $\sqrt{N}+K$, so we can subtract these two to get $2\sqrt{N}$.

Note that we're assuming that $K > \sqrt{N}$. If $K \leq \sqrt{N}$, we don't have an intersect interval, and the following still applies – we don't need an intersect interval since we the region of consideration is already $O(\sqrt{N})$.

Now, we completely ignore the intersect interval part of the region of consideration. Because we're left with segment(s) that add up to $O(\sqrt{N})$, we can do a sliding window minimum – just with taking the minimum of the intersect interval as well each time. This gives us the minimum of each of the segments that begin from $0...\sqrt{N}$ in $\sqrt{N}$ memory and $\sqrt{N}$ time. When we reach $f(\sqrt{N})$, we repeat this for the next $\sqrt{N}$ bucket. This gives us a total runtime for pass two of $O(N)$.

3 Platinum 3 - Disruption

3.1 Problem

Given an unweighted tree with N nodes ($1 \leq N \leq 5*10^4$), and M separate, weighted "replacement" edges ($1 \leq M \leq 5 * 10^4$), determine, for each edge e of the tree, the minimum single edge m of the M additional edges such that all the nodes are connected if we add m to the tree and remove e (or output `-1` if no such m exists).

3.2 Sample Case

```
6 3     (N=6, M=3)
1 2     -> 7
1 3     -> 7
4 1     -> 8
4 5     -> 5
6 5     -> 5
2 3 7
3 6 8
6 4 5
```

3.3 Solution

The problem can be rephrased as follows: for each node n, we want to figure out the minimal edge m that connects the two parts of the tree that result after the edge from n to the parent of n is removed.

Firstly, for each m, instead of storing it directly, we tag the nodes that m connects, noting the weight of m for each of these two nodes. For instance, in the sample case, we would associate nodes 2 and 3 with an edge of weight 7, 3 and 6 with 8, and 6 and 4 with 5. Then, the problem reduces to the following: for each subtree, find the minimum edge associated with any of the subtree's nodes such that we it's not also associated with another node in the subtree (if it's also associated with another node, clearly this edge connects two nodes of the same component, thus rendering it ineligible).

How do we do this? Turns out we can do it (mostly) naively. We first do a DFS. Getting to the leaves, we can push edges associated with them to priority queues, each leaf having a separate priority queue. As we go up, we must merge the priority queues of the children. We do this by merging smaller queues to larger ones. However, whenever we try to add the same edge to a priority queue, we instead remove it. This is done to reflect that the subtree now completely contains the edge. Following this process, we can query the priority queue, once per node (besides the root) as we're coming up from the DFS, to get our final answers.

The runtime analysis of the problem is, however, quite tricky. At first glance, the entire process seems like it'd take $O(N\,M \log M)$. However, this is not the case. Notice that the merges are not necessarily occurring once per node. In fact, we can separate the merging process with the querying process to get a more accurate runtime analysis.

The merging process turns out to be $O(M \log^2 M)$. To see why this is the case, consider the number of times each of the M additional edges are inserted into a priority queue. Every time we merge two priority queues, because we're merging a smaller one to a larger one, our new priority queue is now at least double the smaller one's size. Because the maximal size of any priority queue is $O(M)$, we thus perform at most $O(\log M)$ inserts for each m throughout the whole merging process. In other words, each time we insert an edge m in the merging process, we double the size of the priority queue that m is in, and can only do this $O(\log M)$ times before we get to a size of $O(M)$. Thus, we perform $O(M \log M)$ total inserts, and have a runtime complexity of $O(M \log^2 M)$.

The querying process, during which we query the merged priority queue for each node coming upward from the DFS, is a straightforward $O(N \log M)$.

Thus, our total runtime complexity for this solution is $O(N \log M + M \log^2 M)$, which runs in time.