01:198:344 - Homework III

Kev Sharma - kks107, Section 08

February 19, 2021

1.

```
1:  procedure nSMALLEST(n Arrays)
2:      heap ← new MinHeap                                              ▷ O(1)
3:      for i = 1 ..., n do
4:          ▷ Populates the heap. Does O(log(n)) work per iteration.   ◁
5:          pair ← (A_i[0], (A_i, 0))                                   ▷ O(1)
6:          heap.add(pair)                                              ▷ O(log(n))
7:      result ← new Array of size n
8:      indexResult ← 0
9:      repeat
10:         ▷ Find n-1 smallest elements among all arrays. Populates the first n-1
              elements in result. Explanation found in correctness analysis.   ◁
11:         ▷ Get smallest item from heap and its metadata. Populate result.   ◁
12:         pair ← heap.find-min()                                      ▷ O(1)
13:         result[indexResult] ← pair.key
14:         indexResult ← indexResult + 1
15:         ▷ Replace smallest element. For the element that we remove, insert its
              successor from that same array into heap.                 ◁
16:         heap.delete-min()                                           ▷ O(log(n))
17:         arrayOfMin ← pair.value.array
18:         successor ← pair.value.index+1
19:         pair ← (arrayOfMin[successor], (arrayOfMin, successor))
20:         heap.add(pair)                                              ▷ O(log(n))
21:     until indexResult! = n − 1
22:     pair ← heap.find-min()                                          ▷ O(log(n))
23:     result[indexResult] ← pair.key        ▷ O(1), populates nth element in result.
24:     return result
```

Run-time Analysis:

- Each heap operation is commented to include its run-time as discussed in class.
- The first for loop does $c * log(n)$ work per iteration. Hence we have $n * c * log(n)$ work which equals $\mathcal{O}(nlog(n))$ work. The repeat loop also does $c * log(n)$ work per iteration for $n - 1$ iterations. Hence we have $(n - 1) * c * log(n)$ work for the second loop $= \mathcal{O}(nlog(n))$. Line 22 takes $\mathcal{O}(log(n))$ time.
- In total, our run-time is $2 * \mathcal{O}(nlog(n) + \mathcal{O}(log(n)) = \mathcal{O}(nlog(n))$.

Correctness Analysis:

- We populate *heap* with the smallest elements from each array passed to us. Note that these elements will each occur at index zero of their respective arrays.
- Likewise, each (key,value) *pair* inserted into the heap will contain the smallest element of an array as the key and metadata on where that key comes from in the value.
- The value is a two-tuple of the form (array from where key originates, index of key in said array). *heap* compares pair.key for adding and removing. *heap* only ever maintains a maximum of n elements after the for loop on line 3 terminates (one item in *heap* per array passed).
- The repeat-until loop uses the populated *heap* from the for loop to get the minimum element from all arrays (min of *heap*) and then replaces it in the heap so that there are always n items in the heap where each item represents the smallest element from each array that has not already been added to *result*.
- We refresh the heap with a successor element using the metadata that *pair* stores about the minimum element on line 12. If *pair* is (3, $(A_5, 2)$), then we know the smallest item not in *result* already is the value 3 from the fifth array at index 2 in that array (indexing starts with 0).
- We populate the last element in *result* outside the repeat loop because it could be the case that some array x contains the n minimum elements desired. Hence if we iterated n times in the repeat-loop in this case, line 19 would access the $nth + 1$ element in x. This could cause a stack overflow error.
- The repeat loop terminates after we initialize the first $n - 2$ entries in B. Line 23 then initializes the last (index $n - 1$) entry in B.

2.

```
1: procedure LongestSerialSubsequence(A)
2:     longest ← 1
3:     D ← new Dictionary
4:     for i = 0 . . . , n − 1 do
5:         value ← D.search(A[i] − 1)          ▷ Dictionary.search = O(1)
6:         if value! = NIL then
7:             D.add(A[i], value + 1)          ▷ Dictionary.add = O(1)
8:             longest ← max(longest, value + 1)
9:         else
10:            D.add(A[i], 1)                  ▷ Dictionary.add = O(1)
11:    return longest
```

Run-time Analysis:

- Lines 5 and 7 or 10, where we perform operations on dictionary D each take $\mathcal{O}(1)$ time.
- The remaining statements inside the for loop take constant time as well.
- There are n iterations of the for loop [0,n-1] and we perform a constant amount of work on each iteration.
- Hence the running time is equal to $c * n = \mathcal{O}(n)$

Correctness Analysis:

- Fact 1: Since A contains distinct numbers, an element ($A[i]$) that has not been iterated over by i will not be in the dictionary D. We add $A[i]$ as a key to the dictionary on every iteration, either on line 7 or on line 10 but not on both.
- Because we know that Fact 1 is true, we can conclude on the current i, $A[i]$ has a direct predecessor only if the dictionary contains the key $A[i] - 1$ (line 6).
- Since we cannot do multiple checks on iteration i checking whether $A[i] - 2$, $A[i] - 3$, $A[i] - 4$ (and so on) exist in the Dictionary, we need to let the key referenced by $A[i] - 1$ tell us whether it had found its first predecessor in A.
- Thus our (key, value) pair is of the form (element, length of serial sub-sequence in A ending with that element).
- Hence if an element b is found that is a direct successor to a (where b appears after a in A), then we know that key b's value can be set to key a's value incremented by one. This is possible because now we've extended the length of the sub-sequence ending with a to end with b, and so b's value is a's $value + 1$. If a did not exist in A, then b's value can be set to 1. This logic is found in lines 5 through 10.
- *longest* keeps track of the greatest value we have ever assigned to any key in dictionary D. Consequently, after the for loop terminates, longest will contain the length of the longest serial sub-sequence in A.

3.

```
1: procedure NumberDistinctOver-k-Intervals(A, k)
2:     n ← A.length
3:     B ← empty array of size n − k + 1                              ▷ Output array
4:     indexB ← 0                                                      ▷ Used to populate B
5:     D ← new Dictionary
6:     distinct ← 0
7:     ▷ distinct will be used as a running tally of the number of distinct elements
         over the current interval, distinct won't get reset after an interval ends. ◁
8:     for i = 0 . . . , n − 1 do                                      ▷ n iterations
9:         if i ≥ k then
10:            ▷ At the start of each new interval, append distinct to B to keep track
                 of previous interval's num distinct elements. ◁
11:            B[indexB] ← distinct
12:            indexB ← indexB + 1
13:            ▷ At the start of each interval, remove the previous interval's first el-
                 ement from that dict if it only appeared once otherwise reduce it's
                 value by one. ◁
14:            value ← D.search(A[i − k])                              ▷ O(1)
15:            if value == 1 then
16:                D.remove(A[i − k])                                  ▷ O(1)
17:                distinct ← distinct − 1
18:            else
19:                D.update(A[i − k], value − 1)                       ▷ O(1)
20:        ▷ For the current i, increment distinct iff the key A[i] doesn't exist in dict.
             This ensures we count only the distinct elements in that interval once.
             If A[i] does exist, increment it's value in D to reflect that it appears that
             many times in the current interval. ◁
21:        if D.search(A[i])! = NIL then
22:            D.update(A[i], D.search(A[i]) + 1)                      ▷ O(1)
23:        else
24:            distinct ← distinct + 1
25:            D.add(A[i], 1)                                          ▷ O(1)
26:    ▷ The last interval terminates together with the for loop, so we cannot reach
         line 11 to update B[indexB] for the last interval. So updating it manually for
         the last interval is necessary, otherwise the last element of B (representing
         the number of distinct elements in the last interval of A of size k) remains
         unpopulated. ◁
27:    B[indexB] ← distinct
28:    return B
```

Run-time Analysis: All dictionary calls are labeled in the for loop. Observe that we do a constant amount of work per iteration. Since there are n iterations, we do $c * n$ amounts of work. $\therefore$ our algorithm runs in $\mathcal{O}(n)$ time.

Correctness Analysis:

- Technically since the prompt doesn't ask for justification, you may skip this correctness analysis. However, I've included it to explain the algorithm (though the comments in the pseudocode should suffice).
- This algorithm works by using a dictionary to dictate what happens to *distinct*.
- Line 2 is used to iterate over all elements in the for loop.
- B is the output array. The size of B is equal to the number of k sized intervals found in A. Note that $k \geq 1$. Each entry in B stores the number of distinct elements in that respective interval. For example, the first entry in B (indexed at 0) stores the number of distinct elements in the first interval. Line 4, $indexB$, is used to populate B accordingly.
- Because the first interval ends after k iterations are complete (and because indexing starts at 0), $B[0]$ should be populated as *distinct* at the start of the next iteration, namely when the value of i becomes k (since $|0 \ldots k-1| =$ k).
- From then on, each value i takes on in the for loop will mark the end of another iteration. This can be viewed as a sliding window. If the first iteration is from $i = 0 \ldots k-1$, then the next iteration is from $i = 1 \ldots k$. Thus, we must populate B at each iteration after k iterations are complete.
- Having explained the use of line 9, let us detour to line 21-25 before coming back to the body of the if statement on line 9.
- For the first iteration, line 21 will evaluate to false, and so $distinct \leftarrow distinct + 1$ is justified $A[i]$ was not a key in D (trivial since D was empty at start). For any following iterations, however, line 21 may evaluate to true. If it does, we know that we have already counted that $A[i]$ in our tally *distinct*. To avoid a recount, we simply update $A[i]$'s value in D (see line 20).
- If, on any iteration, $A[i]$ is not a key in D, then we know that the element $A[i]$ is distinct in that interval and can fall through to lines 24 and 25.
- Returning back to our if block on line 9, shifting our figurative window of size k to the right by one element, means we must append to B the number of distinct elements the window previously encapsulated. This is done on lines 11-12.
- Since our window has moved to include a new element, but *distinct* may still be affected by the first element of the previous interval, we must alter *distinct* accordingly.
- To alter *distinct*, we only wish to change the impact the first element of the previous interval had on the tally. The remaining $k-1$ elements' contribution should remain the same because we will later (lines 21-25) contribute the current iteration's impact on the tally (if it isn't a duplicate).
- Lines 15 through 19 contain the logic to alter *distinct* depending on whether the first element of the previous interval had a duplicate in that interval or not.

- In the input example of [3,2,7,3,...] where $k = 4$, the first element of the interval did contain a duplicate. Hence D would contain the (key,value) pair (3,2). We cannot remove the key entirely, since the remaining $k - 1$ elements [3,**2,7,3**] contains a three. Because this is the case, *distinct* should remain unchanged one element in the remaining $k - 1$ elements forming the first $k - 1$ of the second interval contains a three.
- However, if the first element of the previous interval did not have any repeats, i.e $D.search(A[i - k])$ returned 1, then we can take one away from distinct as the new interval does not contain that element. We can also remove that element as a key from D.

```
During a call to our procedure,
Our dictionary D, looks as follows for every iteration:

Beginning of iteration i=0:     {}
Beginning of iteration i=1:     {3=1}
Beginning of iteration i=2:     {2=1, 3=1}
Beginning of iteration i=3:     {2=1, 3=1, 7=1}
Beginning of iteration i=4:     {2=1, 3=2, 7=1}
Beginning of iteration i=5:     {2=1, 3=1, 5=1, 7=1}
Beginning of iteration i=6:     {3=2, 5=1, 7=1}
Beginning of iteration i=7:     {3=2, 5=2}
Beginning of iteration i=8:     {3=1, 5=2, 7=1}
Loop terminated, dictionary:    {2=1, 3=1, 5=1, 7=1}

Contents of B after call: [3, 4, 3, 2, 3, 4]
```

-
- The algorithm, when ran on prompt's input example (with $k = 4$, gives the above output. D is on the right hand side and shows all key,value pairs.
- Note that lines 21 through 25 need to be accounted for even after the last iteration since they have computed *distinct* for the last interval. So when our loop terminates and we cannot reach the if block on line 9, we need to manually initialize the last element of B (equivalently, the number of distinct elements in the last interval of A of size k). We do this on line 27.

4. **procedure** SPECIALPART1(A)
    $sort(A)$ ▷ $\mathcal{O}(nlog(n))$
    $i \leftarrow n$
    while $i \geq 1$ **do**
        if $A[i] == (n + 1 - i)$ **then**
            return $A[i]$
        $i \leftarrow i - 1$
    return "no solution"
▷ *Part 1 has a run-time of $\mathcal{O}(nlog(n))$ to sort and $\mathcal{O}(n)$ to iterate over sorted A. So this first procedure has a run-time of $\mathcal{O}(nlog(n))$* ◁

procedure SPECIALPART2(A)

5.
```
1: procedure extraCredit(A)
2:     n ← A.length
3:     total ← 0
4:     D ← new Dictionary(int, int)
5:     for x = 0, ..., n − 1 do                          ▷ n iterations
6:         total ← total + A[x]
7:         if total == 100 then
8:             return (0, x)
9:         value ← D.search(total − 100)                 ▷ Dictionary.search = O(1)
10:        if value != NIL then
11:            return (value + 1, x)
12:        D.add(total, x)                               ▷ Dictionary.add = O(1)
13:    return “no solution”
```

Run-time Analysis:

- It takes $\mathcal{O}(1)$ amount of time to create a, search for, and add to a Dictionary. Hence lines 4,9, and 12 all take constant time.
- Because we do a $\mathcal{O}(1)$ work per iteration and there are exactly n iterations, we do a total of $n * \mathcal{O}(1)$ of work.
- $\therefore$ the running time of this algorithm comes out to be $\mathcal{O}(n)$.

Correctness Analysis:

- Note that a justification is not required by the prompt.
- The dictionary D stores the running total computed at each index (see line 12). It stores them as a (key,value) pair in the form of (total, index).
- On every iteration, it checks whether the running total has hit 100. If this is the case, then we have a solution (since running total spans from index zero to x) and can return $(0, x)$ where x is guaranteed to be ≥ 0.
- Otherwise line 9-10 checks whether any previous total computed out to be (total - 100). If such a previous total was computed, it would've been stored (line 12).
- previousTotal = total - 100
- 100 = total - previousTotal
- If the above equation can be satisfied with a previousTotal that exists in D, then we can say that elements from index $D.search(previousTotal) + 1$ onwards up to current value of x sum to 100. $\therefore$ we are able to return $(value + 1, x)$.
- If the conditions on line 7 and 10 never evaluate to true, we know there can't be a solution. And so we can safely fall through to line 13.