01:198:344 - Homework III

Kev Sharma - kks107, Section 08

February 19, 2021

1\.

1: **procedure** nSMALLEST(n Arrays)
2: $heap \leftarrow$ new MinHeap ▷ $\mathcal{O}(1)$
3: **for** $i = 1 \ldots, n$ **do**
4: ▷ *Populates the heap. Does $\mathcal{O}(log(n))$ work per iteration.* ◁
5: $pair \leftarrow (A_i[0], (A_i, 0))$ ▷ $\mathcal{O}(1)$
6: $heap$.add($pair$) ▷ $\mathcal{O}(log(n))$
7: $result \leftarrow$ new Array of size n
8: $indexResult \leftarrow 0$
9: **while** $indexResult < n - 1$ **do**
10: ▷ *Find n-1 smallest elements among all arrays. Populates the first n-1 elements in result. Gets smallest item from heap and its metadata.* ◁
11: $pair \leftarrow heap$.find-min() ▷ $\mathcal{O}(1)$
12: $result[indexResult] \leftarrow pair$.key
13: $indexResult \leftarrow indexResult + 1$
14: ▷ *Replace smallest element. For the element that we remove, insert its successor from that same array into heap.* ◁
15: $heap$.delete-min() ▷ $\mathcal{O}(log(n))$
16: $arrayOfMin \leftarrow pair$.value.array
17: $successor \leftarrow pair$.value.index+1
18: $pair \leftarrow (arrayOfMin[successor], (arrayOfMin, successor))$
19: $heap$.add($pair$) ▷ $\mathcal{O}(log(n))$
20: ▷ *Handles case where all n smallest elements come from the same array. Prevents out of bounds indexing for that array.* ◁
21: $pair \leftarrow heap$.find-min() ▷ $\mathcal{O}(log(n))$
22: $result[indexResult] \leftarrow pair$.key ▷ *$\mathcal{O}(1)$, populates nth element in result.*
23: 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 while 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 $2 * \mathcal{O}(nlog(n) + \mathcal{O}(log(n)) = \mathcal{O}(nlog(n))$.

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:         ▷ A[i] has a direct predecessor iff the dictionary contains the key A[i] − 1 ◁
7:         if value! = NIL then
8:             ▷ If the current element b 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. ◁
9:             D.add(A[i], value + 1)                  ▷ Dictionary.add = O(1)
10:            longest ← max(longest, value + 1)
11:        else
12:            D.add(A[i], 1)                          ▷ Dictionary.add = O(1)
13:    ▷ After loop terminates, longest contains the value of the length of the longest
       serial sub sequence in A. ◁
14:    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 $c * n = \mathcal{O}(n)$

3.

```
procedure NumberDistinctOver-k-Intervals(A, k)
    n ← A.length
    B ← empty array of size n − k + 1                              ▷ Output array
    indexB ← 0                                                    ▷ Used to populate B
    D ← new Dictionary
    distinct ← 0
    ▷ 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.  ◁
    for i = 0 . . . , n − 1 do                                     ▷ n iterations
        ▷ Only enter this if block after the completion of k iterations. At the end
          of the first interval, distinct will be appropriately initialized to contain
          how many different numbers were in the first interval.  ◁
        if i ≥ k then
            ▷ At the start of each new interval, append distinct to B to keep track
              of previous interval's num distinct elements.  ◁
            B[indexB] ← distinct
            indexB ← indexB + 1
            ▷ 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.  ◁
            value ← D.search(A[i − k])                             ▷ O(1)
            if value == 1 then
                D.remove(A[i − k])                                 ▷ O(1)
                distinct ← distinct − 1
            else
                D.update(A[i − k], value − 1)                      ▷ O(1)
        ▷ 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.  ◁
        if D.search(A[i])! = NIL then
            D.update(A[i], D.search(A[i]) + 1)                     ▷ O(1)
        else
            distinct ← distinct + 1
            D.add(A[i], 1)                                         ▷ O(1)
    ▷ We initialize B[indexB] for each interval at the beginning of the next inter-
      val; hence when the for loop terminates we still need to initialize the last
      element of B to contain the number of distinct elements in the last interval. ◁
    B[indexB] ← distinct
    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.

4\. ▷ *Part 1 sorts the array and trivially iterates from the end of the array to the front, comparing each element to n+1-(index of that element) to determine if it is special. Here I assume index convention of 1 to n rather than 0 to n-1.* ◁

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)
    if A.length == 0 **then**
        return "no solution"
    return FINDSPECIAL(A,0)

procedure FINDSPECIAL(A, offset)
    $n \leftarrow A.length$
    if $n == 1$ **then** ▷ *Base Case -* $\mathcal{O}(1)$
        if $A[1] == (n$+offset$)$ **then**
            return $A[1]$
        else
            return "no solution"

    $median \leftarrow Select(A, \lfloor(n/2)\rfloor)$ ▷ $\mathcal{O}(n)$ *(discussed in class)*
    $A_{lesser}, A_{greater} \leftarrow Partition(A, median)$ ▷ $\mathcal{O}(n)$ *(discussed in class)*
    ▷ *The median of an array is the $\lfloor(n/2)\rfloor$th order statistic. Therefore $k = \lfloor(n/2)\rfloor$. Accordingly, we have the formula below:* ◁
    $medianInverseRank \leftarrow n + 1 - \lfloor(n/2)\rfloor$+offset
    if $median == medianInverseRank$ **then**
        return $median$
        ▷ *We either have a solution or should recursively determine the answer from either A_{lesser} or $A_{greater}$ (but not both).* ◁
    else if $median > medianInverseRank$ **then**
        return FINDSPECIAL(A_{lesser}, $medianInverseRank$)
    else
        return FINDSPECIAL($A_{greater}$, offset)
    ▷ *Recall that* PARTITION*(A, median) splits A into two arrays where A_{lesser} denotes the elements less than the median and $A_{greater}$ denotes the elements greater than the median in A. Hence we split A into two sub-problems of half the size (definition of median) . Our recurrence relation for this function is ∴ $T(n) = T(n/2) + \mathcal{O}(n) = \mathcal{O}(n)$ as seen in class.* ◁

Justification for 4.2

- Base Case: For an array of size 1, there is only one possible special element. If that sole element is not the solution, there can no longer be one.
- Note that inverseRank when contrasted to indexing, goes from n to 1.
- After we partition the array into elements less than and greater than the median, we can halve our problem into two parts if the median is not the solution.
- Say the median exceeds medianInverseRank, then all elements in $A_{greater}$ exceed medianInverseRank as well. But since indexing of inverseRank happens in descending order, all elements x in $A_{greater}$ exceed their respective inverseRank(x), by virtue of exceeding medianInverseRank. Thus no element in $A_{greater}$ can be the solution.
- If instead medianInverseRank exceeds median, then all elements in A_{lesser} are less than medianInverseRank. Likewise, no element in x A_{lesser} can be the solution because x is less than medianInverseRank but medianInverseRank is less than any inverseRank(x) (descending order). Thus no element from A_{lesser} can be the solution.
- In this way we can divide the problem into a sub-problem of half the size (partitioned around the median).
- A discussion on offset took place in class, so to be brief: recursively calling A_{lesser} resets the inverseRank indexing to be sizeOf(A_{lesser}) to 1. We would like to preserve it to be sizeOf(A) to 1 to ensure proper comparison, so we pass in an offset + inverseRank(median).

5.

1: **procedure** EXTRACREDIT(A)
2: $n \leftarrow A.length$
3: $total \leftarrow 0$
4: $D \leftarrow$ new $Dictionary$ of key,value pair type $(integer, integer)$
5: **for** $x = 0, \ldots, n-1$ **do** ▷ n *iterations*
6: $total \leftarrow total + A[x]$
7: **if** $total == 100$ **then**
8: return $(0, x)$
9: $value \leftarrow D.search(total - 100)$ ▷ $Dictionary.search = \mathcal{O}(1)$
10: **if** $value\ != NIL$ **then**
11: return $(value + 1, x)$
12: $D.add(total, x)$ ▷ $Dictionary.add = \mathcal{O}(1)$
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.