Homework #3 Solutions

CS 344: Design and Analysis of Computer Algorithms (Fall 2022)

Problem 1

Given an array A, we define a *subsequence* of A to be any sequence $A[i_1], A[i_2], ..., A[i_k]$ where $i_1 < i_2 < ... < i_k$. Note that the indices do not have to be right next to each other; we could have something like $i_1 = 3, i_2 = 7$.

Now, we say that a subsequence $A[i_1], A[i_2], ..., A[i_k]$ is *serial* if $A[i_1] = A[i_2] - 1 = A[i_3] - 2 = A[i_4] - 3 = ... = A[i_k] - k + 1$. Note that any one element on its own is a serial subsequence of length 1, so every array A always contains a serial subsequence of length at least 1. Now consider the following problem

- INPUT: unsorted array A of length n. You can assume all numbers in A are distinct.
- OUTPUT: the length of the longest serial subsequence.

Write pseudocode for an algorithm that solves the longest serial subsequence problem in time $O(n)$.

The idea is as follows: for each $A[i]$, we want to know the longest serial subsequence of $A[0..i-1]$ that ends with value $A[i]-1$. Given such a subsequence, we can produce the longest serial subsequence of $A[0..i]$ that ends in $A[i]$.

To implement our algorithm, we need not actually store the subsequence. Indeed, every serial subsequence is uniquely determined by its largest element and its length. Using the above update strategy, we store the length of the longest serial subsequence that ends at each value in the input array.

```
LongestSerialSubsequenceLength(A[0..n − 1]):
  ℓ ← 0
  D ← BuildDictionary()
  for i ← 0 to n − 1
      v ← D.search(A[i] − 1)
      if v = Nil
          v ← 0
      D.add(A[i], v + 1)
      ℓ ← max(l, v + 1)
  return l
```

Correctness. To show the algorithm is correct, we can argue that it maintains the following loop invariant: at the end of the ith iteration, the dictionary maps each element $x \in A[0..i]$ to the length of the longest serial subsequence ending in x. This invariant can be proved by induction which seals the argument.

Time complexity. Every operation in the loop takes $O(1)$ expected time, so the entire algorithm runs in $O(n)$ time in expectation.

Problem 2

Consider the following problem, where we are given an array A with some duplicate elements, and want to find the number of distinct elements in each interval of size k.

- Input: a positive integer k, and an unsorted A with n numbers, some of them repeating.
- Output: an array B of length $n - k + 1$, where $B[i]$ should contain the number of *distinct* elements in the interval $A[i], A[i+1], ..., A[i+k-1]$.

Write pseudocode for an algorithm for this problem with expected running time $O(n)$. Note that your expected running time should be $O(n)$ even if k is large. A run-time of $O(nk)$ is too slow, and will receive very little credit.

Our algorithm follows a "sliding window" approach. At each step, the algorithm considers an interval of size k, determines the number of distinct elements within the window, and then shifts the window by one. As with most sliding window algorithms, the key is to determine how to deduce the value for the next window without recomputing things from scratch. In this problem, we use a dictionary to track the frequency of each element within the current window.

```
DistinctElements(A[0..n-1], k):
  D ← BuildDictionary()
  for i ← 0 to k − 1
      f ← D.search(A[i])
      if f = Nil
          f ← 0
      D.update(A[i], f + 1)

  initialize B[0..n − k]
  for i ← 0 to n − k − 1
      B[i] ← D.size()
      D.update(A[i], D.search(A[i]) − 1)
      if D.search(A[i]) = 0
          D.delete(A[i])
      f ← D.search(A[i + k])
      if f = Nil
          f ← 0
      D.update(A[i + k], f + 1)
  B[n − k] ← D.size()
  return B
```

Correctness. The loop invariant maintained by this algorithm (of the second loop) is that at the end of the ith iteration, D maps every element x to its frequency in $A[i..i+k-1]$, and elements not in $A[i..i+k-1]$ are not keys in D. In particular, the number of keys in D is the number of distinct elements in the ith window.

Time complexity. Every operation in the main loop takes $O(1)$ expected time, so the entire algorithm runs in $O(n)$ time in expectation.

Problem 3

Let A be some array of n integers (possibly negative), with no duplicated elements, and recall that Rank(x) = k if x is the kth *smallest* element of A. Now, let us define a number x in A to be *special* if Rank(x) = -x. (Note the minus sign on the right hand sign.) For example, if $A = 1, -2, 4, 7, -5$, then -2 is special because it is the second smallest number, so Rank$(-2) = 2$, so we have Rank$(-2) = -(-2)$.

Consider the following problem:

- Input: unsorted array A of length n
- Output: return a special number x in A, or return "no solution" if none exists.

(1) Give pseudocode for a $O(n \log(n))$ algorithm for the above problem.

Since the elements of A are distinct, an element's rank is completely determined by its position in A after sorting. Indeed, if an element has rank k in A, then it will be in index $k-1$ after sorting A in increasing order. This observation suggests the following easy $O(n \log n)$ algorithm.

```
FindSpecial(A[0..n − 1]):
    sort A in increasing order
    for r ← 0 to n − 1
        if A[r] = −(r + 1)
            return A[r]
    return Nil
```

Correctness. Correctness follows from the simple observation above the algorithm description.

Time complexity. The bottleneck of the algorithm is sorting, which takes $O(n \log n)$ time.

(2) Give pseudocode for a recurrence $O(n)$ algorithm for the above problem. Give a brief justification for why the algorithm is correct. Make sure to also state the recurrence formula of the algorithm.

We use the observation of part (1), but now use the high-level approach of median recursion: we explicitly select the element of rank $n/2$, test if it is special, and then recurse on one of the left- and right-halves.

We must be a little bit careful as rank with respect to just the right half is different than rank with respect to the entire array. To handle this minor challenge, we introduce an additional parameter *offset* which tells us how to adjust the rank in the subarray to recover the rank in the entire array. More precisely, we say that x is ***offset-special in B*** if $-x = \text{rank}_B(x) + \textit{offset}$, and the algorithm FindSpecial($B[0..n-1]$, *offset*) below finds an *offset*-special element of B.

```
FindSpecial(B[0..n − 1], offset):
  k ← ⌊n/2⌋ + 1
  m ← Select(B, k)
  if −m = k + offset
      return m
  if n ≤ 1
      return Nil
  L, R ← Partition(B, m)
  if m ≥ 0 or −m < k + offset
      return FindSpecial(L, offset)
  else
      return FindSpecial(R, k + offset)
```

To find a special number in A, we can simply call FindSpecial$(A, 0)$ since being 0-special is the same as being special.

Correctness. We must show that FindSpecial$(B[0..n-1], \mathit{offset})$ finds an x such that $-x = \text{rank}_B(x) + \mathit{offset}$.

There are three cases to consider:

(i) $-m = k + \mathit{offset}$. In this case, m is *offset*-special since k is the rank of m.

(ii) $m \geq 0$ or $-m < k + \mathit{offset}$. If m is non-negative, no element greater than m can be *offset*-special as only negative numbers are candidates. Suppose then that m is negative.

Since any element $x > m$ has $\text{rank}_B(x) > \text{rank}_B(m)$, it follows that $-x < -m < k + \mathit{offset} = \text{rank}_B(m) + \mathit{offset} < \text{rank}_B(x) + \mathit{offset}$, and thus all $x > m$ can be ruled out as candidates. Any *offset*-special element must then be in L. Because the rank of an element in L with respect to L is the same as the element's rank with respect to B, being *offset*-special in L means being *offset*-special in B.

(iii) $m < 0$ and $-m > k + \mathit{offset}$. Similar to the previous case, if $x < m$, then $-x > -m > k + \mathit{offset} = \text{rank}_B(m) + \mathit{offset} > \text{rank}_B(x) + \mathit{offset}$. Thus any *offset*-special element must be in R. Unlike before, however, rank with respect to R is different than rank with respect to B. Fortunately, this is not a big problem: being $(k + \mathit{offset})$-special in R means being *offset*-special in B. To see this, note that the difference between B and R is that B contains k more elements, all of which are smaller than all of the elements of R.

Time complexity. The recurrence for FindSpecial satisfies $T(n) = T(n/2) + O(n)$. We have analyzed this recurrence before, and it solves to $T(n) = O(n)$.

Problem 4 (Extra Credit)

Say that you have 1024 toys $t_1, ..., t_{1024}$. You can assume that all toys have different weights. The only measurement tool you have available is a scale: you put two toys on the scale and it shows you which is lighter, which is heavier. We call this a comparison. Your goal for this problem is to find simultaneously both the lightest toy AND the second-lightest toy.

Show that using at most 1050 comparisons you can simultaneously find both the lightest and second-lightest toy.

We find the two lightest toys in two phases. In the first phase, we will find the lightest toy. Using information gained from the first phase, we find the second-lightest toy in the second phase.

(Phase 1) Finding the lightest toy. To find the lightest toy, we can face them off in a single-elimination tournament where each match is played by two toys and the outcome of a match is determined by a call to CompareWithScale; the winner of a match is what CompareWithScale identifies as the lightest among the two toys. The winner of the finals is then the lightest toy.

Here is how we can conduct a round of the tournament on 2^k toys. (i) Partition the 2^k toys into 2^{k-1} groups of 2. (ii) Call CompareWithScale on each of the 2^{k-1} groups. (iii) Advance the 2^{k-1} winners into the next round.

Information gained from finding the lightest toy. Once the tournament has been played out, let us think about where the second-lightest toy could be in the bracket. The second-lightest toy could not have won the finals (because the lightest toy wins the finals), so the second-lightest toy must have been eliminated in some match. No toy except for the lightest toy can win the second-lightest toy in a match, so the second-lightest toy must have played a match against the lightest toy and been eliminated!

(Phase 2) Finding the second-lightest toy. We have thus found a small pool of candidate toys among which one of them must be the second-lightest toy: the toys which have lost a match against the lightest toy. We can find them by looking at the tournament bracket of the first phase. The number of such candidates is $\log_2 1024 = 10$ since any toy – and so, the lightest toy – plays one match per round and there are $\log_2 1024$ rounds in the tournament (because only half the competitors of a round move on to the next round). We can thus hold another single-elimination tournament among these 10 candidates who had lost a match against the lightest toy. The winner of the finals of this second tournament is then the second-lightest toy.

To summarize, we can find the two lightest toys in two phases. In the first phase we run a single-elimination tournament among all the toys wherefrom we identify the winner of the finals with the lightest toy. In the second phase we run a single-elimination tournament among the 10 toys who lost matches against the lightest toy in the first phase; the winner of the finals of this second tournament is the second-lightest toy.

Complexity. Our cost model is the number of calls to CompareWithScale. Let us first count the number of calls in the first tournament used to determine the lightest toy.

In round 1, there are $1024/2 = 512$ calls to CompareWithScale.

More generally, in round ℓ, there are $1024/2^\ell$ calls to CompareWithScale.

Since there are $\log_2 1024 = 10$ rounds, there are

$$\sum_{\ell=1}^{10} 1024/2^\ell = 1023$$

calls to CompareWithScale made to determine the bracket for the first tournament.

The lightest toy would have played 10 matches and so there are 10 toys in the second tournament to determine the second-lightest toy. A similar argument shows the second tournament would make 9 calls to CompareWithScale.

The total number of calls to CompareWithScale is thus $1023 + 9 = 1032$.

Pseudocode. The pseudocode below provides an alternative way to understand the solution. For simplicity of exposition (base case handling), we assume that $n = 2^{2^m}$ for some integer m. FindTwoLightest is what we call to find the two lightest toys. It calls FindLightestWithHistory, which is a simulation of a tournament that returns a winner and the list of all toys who played a match against the winner but came in second-place in those matches. Remember that the cost model here is the number of calls to CompareWithScale; running time is irrelevant.

```
FindTwoLightest(T[0..n − 1]):
    s_1, H ← FindLightestWithHistory(T)
    //We don't need the history for the second call, but may as well reuse the implementation.
    s_2, H ← FindLightestWithHistory(H)
    return s_1, s_2
```

```
FindLightestWithHistory(T[0..n − 1]):
    if n = 2
        //s : small, ℓ : large.
        s, ℓ ← CompareWithScale(T[0], T[1])
        //The history comprises of ℓ.
        return s, {ℓ}
    //The toys which advance to the next round.
    S ← {}
    for i ← 0 to n − 1 where i increments by 2
        //s : small, ℓ : large.
        s, ℓ ← CompareWithScale(T[i], T[i + 1])
        //s advances to the next round.
        S ← S ∪ {s}
    s, H ← FindLightestWithHistory(S)
    //ℓ_s : the toy which lost to s in this round. Add it to history.
    return s, H ∪ {ℓ_s}
```