Sample Solution for Problem Set 5

Data Structures and Algorithms, Fall 2021

October 28, 2021

Contents

1 Problem 1

1.1 Algorithm

Algorithm 1 K-SORT

function K-SORT(A, l, r, k)
  if $k = 1$ **then**
    return
  end if
  $blockSize = (r - l + 1)/k$
  $pos = \lfloor k/2 \rfloor \times blockSize$
  RANDOMIZED-SELECT(A, l, r, pos)
  K-SORT$(A, l, l + pos - 1, \lfloor k/2 \rfloor)$
  K-SORT$(A, l + pos, r, \lceil k/2 \rceil)$
end function

1.2 Correctness

Proof.
Basis When $k = 1$, trivial
I.H. Assume when $k < k_0$, K-SORT works correctly.
Ind. Step When $k = k_0$, after RANDOMIZED-SELECT(A, l, r, pos), any number in $A[l, l + pos - 1]$ is smaller than any number in $A[l + pos, r]$. And the length of left/right part is a multiple of $blockSize$. According to the induction hypothesis, two subproblems for $A[l, l + pos - 1]$ and $A[l + pos, r]$ can be solved correctly. So $A[l, \cdots, r]$ can be $k - sort$ correctly. □

1.3 Time Complexity

$$T(n, k) = 2T(n/2, k/2) + O(n)$$

and $T(*, 1) = O(1)$. The depth of the recursion tree is $\Theta(\lg k)$, each level costs $O(n)$. So $T(n, k) = O(n \lg k)$

1.4 Lower Bound

The number of all possible answers is equal to multinomial coefficient

$$\binom{n}{\frac{n}{k}, \frac{n}{k}, \cdots, \frac{n}{k}} = \frac{n!}{\left(\frac{n}{k}!\right)^k}$$

So the height of decision tree satisfies

$$2^h \geq \frac{n!}{\left(\frac{n}{k}!\right)^k}$$

Then

$$\begin{aligned} h &\geq \lg \frac{n!}{\left(\frac{n}{k}!\right)^k} \\ &= \lg n! - k \lg \frac{n}{k}! \\ &= \Omega(n \lg n - k\frac{n}{k} \lg \frac{n}{k}) \\ &= \Omega(n \lg k) \end{aligned}$$

2 Problem 2

(a)

$$Pr(\text{two counterfeit coins on the same side}) = \frac{\binom{n-2}{n/2-2} + \binom{n-2}{n/2}}{\binom{n}{n/2}} = \frac{n-2}{2n-2}.$$

(b)

Note that if two counterfeit coins are put on the different side, we can apply standard binary search technique to find both counterfeit coins in $\log n/2$ times of comparison. Hence, our strategy simple works as follows:

- Randomly split n coins into two even parts, and repeat this procedure until two counterfeit coins are not on the same side (which meas the balance tilts towards one side).

Let X be the random variable indicating the number of attempts to put two counterfeit coins into different side. Note that X follows geometry distribution with parameter $p = \frac{n}{2n-2}$, which means

$$\mathbb{E}[X] = \frac{2n-2}{n}.$$

Therefore, the expected number of comparisons is $2\log n/2 + \frac{2n-2}{n}$.

Remark

In binary search part, the number of comparisons can be further optimized to $2\lceil \log_3 n/2 \rceil$. Furthermore, there exist some deterministic algorithms which can **guarantee** that two coins can be put in different sides in $\log n$ rounds[1]. Suppose $n = 2^k$ and balls are numbered from 0 to $n-1$, the algorithm goes as follows:

- In i-th round, put j-th ball to the left side of balance if the $i-1$-th bit of j is 0, and right side otherwise.

[1]Note that our randomized approach will **not** guarantee this property in any finite rounds!

3 Problem 3

(a)

Algorithm

Use $Quicksort$ to sort array S and W based on the value of S. Iterate through array W to calculate the sum of array W. Iterate through array S and check if $S[i]$ is the magical-mean.

Algorithm 2 Finding Magical Mean

```
function FindingMagicalMean(S,W)
    Sort array S and W based on the value of S;
    tot = 0;
    for i = 1 to n do
        tot = tot + W[i];
    end for
    cur = 0;
    for i = 1 to n do
        if cur ≤ tot/2 and tot − cur − W[i] ≥ tot/2 then
            ans = S[i];
            break;
        end if
        cur = cur + W[i];
    end for
    return ans;
end function
```

Analysis

We have $T(n) = O(n \log n) + O(n) = O(n \log n)$.

(b)

Algorithm

Here is a recursive algorithm for finding magical mean. To calculate the magical mean of a set of items represented by array S and W, call $FindingMagicalMean(S, W, 1, n, 0, 0)$ initially.

What does $FindingMagicalMean$ function do? First, use the $O(n)$ time select algorithm to select the median m of current range $[l, r]$ of S. Partition array S together with W using m as a pivot. Check if m is the magical mean. If m is the magical mean, return m. Otherwise, recursively apply finding function to the heavier sub-range.

Analysis

We have $T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n)$.

Algorithm 3 Finding Magical mean

function FINDINGMAGICALMEAN(S, W, l, r, lsum, rsum)
  $m = FindMedian(S, l, r)$;
  $q = Partition(S, W, l, r, m)$;
  $wl = 0$;
  for $i = l$ **to** $q - 1$ **do**
    $wl = wl + W[i]$;
  end for
  $wr = 0$;
  for $i = q + 1$ **to** r **do**
    $wr = wr + W[i]$;
  end for
  $tot = wl + wr + lsum + rsum + W[q]$;
  if $wl + lsum \leq tot/2$ and $wr + rsum \leq tot/2$ **then**
    $ans = m$;
  else if $wl + lsum > tot/2$ **then**
    $ans = FindingMagicalMean(S, W, l, q - 1, lsum, rsum + wr + W[q])$;
  else
    $ans = FindingMagicalMean(S, W, q + 1, r, lsum + wl + W[q], rsum)$;
  end if
  return ans;

4 Problem 4

4.1 (a)

1 Create three lists l_1, l_2, l_3.

2 Compare $a[i]$ $(i \in \{2, 3, \cdots, n\})$ with $a[1]$.

- If $a[i] < a[1]$, append $a[i]$ to l_1.
- If $a[i] = a[1]$, append $a[i]$ to l_2.
- If $a[i] > a[1]$, append $a[i]$ to l_3.

3 Finally append $a[1]$ to l_2.

4 Let $l =$ concatenate(l_1, l_2, l_3), l is what we want.

4.2 (b)

Algorithm 4 STRING-SORT

function STRING-SORT(strList, i)
  Create 27 buckets $B_\emptyset, B_a, B_b, \cdots, B_z$
  for s **in** $strList$ **do**
    Assign s to $B_{s[i]}$
  end for
  for $i =$ 'a' **to** 'z' **do**
    STRING-SORT$(B_i, i + 1)$
  end for
  return concat$(B_\emptyset, B_a, B_b, \cdots, B_z)$

5 Problem 5

(a)

Indeed, we can prove the property for arbitrary tree.

- There exists a centroid in any tree T.

Suppose our claim fails to hold on tree $T = (V, E)$. Let s_u be the size of the largest (connected) component which does not contain $u \in V$. Choose $u^\star \in V$ that achieves minimum value of s_u over all vertices. By our assumption, $s_{u^\star} > \frac{n}{2}$. Therefore, there exists a component C with size $s_{u^\star}$ that does not contain $u^\star$. Let $v \in C$ such that $(u^\star, v) \in E$. Obviously, $s_v < s_{u^\star}$, which leads to the desired contradiction,

(b)

A standard approach is to pre-compute the size of subtree rooted at $u \in V$ for all vertices, and then check the validity of each vertices. These implementations are standard, and we will omit here.

6 Problem 6

(a)

We can solve this problem by Divide and Conquer.

6.1 Algorithm

Algorithm 5 Find K_{th} Smallest Element (FKSE)

Require: $A[1...n], B[1...m], k.$
Ensure: The k_{th} smallest elements in $A \cup B$

if A is empty. **then**
    return $B[k]$
end if
if B is empty. **then**
    return $A[k]$
end if
if $k = 1$ **then**
    return $min(A[1], B[1])$
end if
if $n > m$ **then**
    return FKSE($B[1...m], A[1...n], k$)
end if
$k_0 \leftarrow min(k/2, n)$
$k_1 \leftarrow k - k_0$
if $A[k_0]$ = $B[k_1]$ **then**
    return $A[k]$
end if
if $A[k_0] < B[k_1]$ **then**
    return FKSE($A[k_0 + 1...n], B[1...m], k - k_0$)
else
    return FKSE($A[1...n], B[k_1 + 1...m], k - k_1$)
end if=0

(b)

This is a classic problem, and has classic algorithm : Morris Traverse.

6.2 Algorithm

Algorithm 6 Morris Traverse

Require: $root.$

$cur \leftarrow root$
while $cur \neq null$ **do**
  if $cur.left \neq null$ **then**
    $pre \leftarrow cur.left$
    while $pre.right \neq null$ and $pre.right \neq cur$ **do**
      $pre \leftarrow pre.right$
    end while
    if $pre.right = null$ **then**
      $pre.right \leftarrow cur$
      $cur \leftarrow cur.left$
    else
      $Visit(cur)$
      $pre.right \leftarrow null$
      $cur \leftarrow cur.right$
    end if
  else
    $Visit(cur)$
    $cur \leftarrow cur.right$
  end if
end while