Sample Solution for Problem Set 4

Data Structures and Algorithms, Fall 2020

November 7, 2020

Contents

1 Problem 1

(a) This statement ignores all other possible algorithms to solve this problem, so it is not the lower bound of the problem.

(b) Since length of each subsequece is k, there are $(k!)^{n/k}$ possible output permutations. To compute the height h of the decision tree, we must have $(k!)^{n/k} \leq 2^h$. Taking logs on both sides, we know that

$$\begin{aligned} h &\geq \frac{n}{k} \times \lg(k!) \\ &\geq \frac{n}{k} \times \left(\frac{k \ln k - k}{\ln 2} \right) \\ &= \frac{n \ln k - n}{\ln 2} \\ &= \Omega(n \lg k). \end{aligned}$$

2 Problem 2

(a) The algorithm will begin by preprocessing exactly as COUNTING-SORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range $[a..b]$, simply compute $C[\min(b,k)] - C[a-1]$ (when $a = 0$, simply use $C[\min(b,k)]$). This takes $O(1)$ times and yields the desired output.

b. Group the integers by the number of digits ($O(n)$), and use radix sorting algorithm to sort in each group ($O(n)$).

3 Problem 3

(a)

$$\begin{aligned}
& \frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \\
\Leftrightarrow \quad & \sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j] \\
\Leftrightarrow \quad & \sum_{j=i+1}^{i+k-1} A[j] + A[i] \leq A[i+k] + \sum_{j=i+1}^{i+k-1} A[j] \\
\Leftrightarrow \quad & A[i] \leq A[i+k]
\end{aligned}$$

(b)

- Algorithm: Split the array into k sub-arrays of size n/k. ($A_i = a_i, a_{i+k}, a_{i+2k}, \ldots; i = 1, 2, \ldots, k$. Then use MergeSort to each sub-arrays.
- Complexity: $O(k\left(\frac{n}{k}\log\frac{n}{k}\right)) = O(n\log\frac{n}{k})$.

(c)

- Algorithm: Split the array into k sorted lists. Construct a min-heap from the heads of each of the k list. Pop a list each time, extract its first element, then push the list to the heap. Each operation extract the current smallest element.
- Complexity:

	Split:	$O(n)$
	Build-heap:	$O(k)$
For each operation,		
	Pop:	$O(\lg k)$
	Extract:	$O(1)$
	Push:	$O(\lg k)$

  In total $O(n + k + n\,(\lg k + 1 + \lg k)) = O(n \lg k)$.

(d)

- Assume we can k-sort a array in $T(k, n)$. According to (c), we can sort it in $O(n \lg k)$. Therefore we can sort an arbitrary array in $T(k, n) + O(n \lg k)$, which needs to satisfy the lower bound of the comparison sort.
- Because k is a constant, $T(k, n) + O(n \lg k) \geq \Omega(n \lg n)$ implys $T(k, n) \geq \Omega(n \lg n)$.

4 Problem 4

Suppose the array is $A[1...n]$. For each element $A[i]$ create an empty stack $S[i]$ initialy. Then do the following procedure to find the smallest element in A. Create array $B[1...n]$ to store the index of elements in $A[i]$. Initially set $B[1...n] = \{1, 2, ..., n\}$:

- For $k = 0$ to $(\log n) - 1$ do
  - $N \leftarrow \frac{n}{2^k}$
  - For $i = 1$ to $\frac{N}{2}$ do
    - $S[B[2i-1]].push(A[2i])$, $S[B[2i]].push(A[2i-1])$.
    - If $A[2i-1] < A[2i]$, then $A[i] = A[2i-1]$, $B[i] = 2i-1$. Else, $A[i] = A[2i]$, $B[i] = 2i$.
- Return the smallest element in $S[B[1]]$ (In $S[B[1]].size - 1$ comparisons).

Intuitively, our algorithm first use $n-1$ comparisons to find the smallest element by pairing up, while storing all the elements that have been compared to $A[i]$ in $S[i]$. Then the smallest element that have been compared to the smallest element is the second smallest element.

Correctness and complexity: The following claim is trivial and helpful to prove the correctness.

Claim 0.1. *$S[B[1]]$ has stored all the element that compared to the smallest element during the algorithm.*

Claim 0.2. *The size of $S[B[1]]$ is $\log n$.*

To prove the second smallest element is in $S[B[1]]$, notice that any other element must be thrown out by another element that smaller than it. But the "another element" can not be the smallest element, which means the element is not the second smallest element in $A[1...n]$. Thus, by finding the smallest element that have compared to the smallest element, we get the second smallest element.

5 Problem 5

5.1 (a)

- Divided into groups of 7:
  - At least $\frac{2n}{7}$ numbers are less than *median of medians*.
  - Recursion: $T(n) \leq T(\frac{5n}{7}) + T(\frac{n}{7}) + O(n)$.
  - Assume that $T(n) \leq cn$, $O(n) \leq c'n$,
  - $T(n) \leq c\frac{5n}{7} + c\frac{n}{7} + c'n = c\frac{6n}{7} + c'n$. Just need $c \geq 7c'$.
  - The method is linear.
- Divided into groups of 3:
  - At least $\frac{n}{3}$ numbers are less than *median of medians*.
  - Recursion: $T(n) \leq T(\frac{2n}{3}) + T(\frac{n}{3}) + O(n)$.
  - Assume that $T(n) \leq cn$, $O(n) \leq c'n$,
  - $T(n) \leq c\frac{2n}{3} + c\frac{n}{3} + c'n = cn + c'n$. No solution.
  - The method is not linear.

5.2 (b)

- Algorithm: Divide and conquer method. Find the midpoint recursively. Call Quantiles($A, 1, n, k$).

Algorithm 1: Quantiles(A, l, r, k)

```
1 mid=(l+r-1)/2;
2 if k > 1 then
3 |   QuickSelection(A, l, r, (r − l + 1)/2);
4 |   Quantiles(A, l, mid, k/2);
5 |   Quantiles(A, mid + 1, r, k/2);
6 end
```

- Correctness:
  - I.H: When $k = 2^m$, $m \geq 1$, after the algorithm, i-th quantile of $A[l : r]$ is at $A[(l-1) + \frac{i(r-l+1)}{k}]$, $1 \leq i \leq k-1$.
  - I.B: When $k = 2^1$, this algorithm is the same as QuickSelection. The only quantile is at $A[r-l+1] = A[(l-1) + \frac{r-l+1}{2}]$.
  - I.S: When $k = 2^{m+1}$,
    - After QuickSelection($A, l, r, (r-l+1)/2$), the only quantile is at $A[\frac{r-l+1}{2}] = A[(l-1) + (r-l+1)\frac{1}{2}]$.
    - After Quantiles($A, l, mid, k/2$), i-th quantile of $A[l : mid]$ is at $A[(l-1)+\frac{i(mid-l+1)}{k/2}] = A[(l-1) + (r-l+1)\frac{i}{k}]$, $1 \leq i \leq \frac{k}{2} - 1$.
    - After Quantiles($A, mid+1, r, k/2$), i-th quantile of $A[mid+1 : r]$ is at $A[mid + \frac{i(r-mid)}{k/2}] = A[(l-1) + (r-l+1)\frac{\frac{k}{2}+i}{k}]$, $1 \leq i \leq \frac{k}{2} - 1$.
    - Combine the three cases, i-th quantile of $A[l : r]$ is at $A[(l-1) + (r-l+1)\frac{i}{k}]$, $1 \leq i \leq k-1$.

- Complexity:
  - Each layer is $O(n)$ for all $(\lg k)$ layers. $T(n) = O(n \lg k)$.

6 Problem 6

TL;DR: Use divide and conquer technique with SELECT oracle to solve a generalized version of this task.

6.1 Generalization

In the following, we will provide an algorithm which receives array a, w and a threshold t as input, and outputs element a_k such that $\sum_{a_i<a_k} w_i < t$ and $\sum_{a_i>a_k} w_i \leq \sum_i w_i - t$.

6.2 Algorithm

Algorithm is attached in the Appendix page.

6.3 Correctness

We would like to show that our algorithm runs correctly for input with threshold $t > 0$ and $t \leq \sum_i w_i$. And without any doubt, we will prove it by induction.

Base case holds trivially. If element array a consists of n elements, the algorithm will use SELECT oracle to select the $(1 + \lfloor \frac{n}{2} \rfloor)$-th smallest element, and divide the whole array into two parts, smaller part (whose element are smaller than this selected element) and larger part as algorithm suggested. If the total weight of the smaller part is greater or equal than t, It can be seen that our weighted median should be less than this selected element. Another part can be seen similarly, and that concludes our proof.

6.4 Time Complexity

$T(n) = T(\frac{n}{2}) + O(n)$ implies $T(n) = O(n)$ in the worst case.

7 Problem 7

7.1 Uniqueness

We may prove it by induction on the number of vertices in the given tree. Indeed, it almost tells the algorithm which may constructs the tree via pre-order and in-order numbers.

Base case trivially holds. Given a tree with n vertices, we may need the following easy but powerful observations to conclude our proof and provide our algorithm.

- pre-order(resp. in-order) numbers in a given subtree form a sequence of continous integers.
- Given pre-order numbers of a subtree which is exactly $L \leq i \leq R$, the pre-order number of the root of this subtree is L.
- Given in-order numbers of a subtree, the root's in-order number x is exactly the size of root's left subtree plus the number of vertices which are not in the subtree but in the left side[1] of the subtree.

We may derive our proof using those observations. Due to last two observations, we may know that the left-child of root has pre-order number 1, and right-child has $x + 1$. And our proof concludes immediatly after applying induction hypothesis and the first observation.

7.2 Algorithm

We may assume that rev_pre and rev_in array are given in the input section where $rev\_pre[pre[i]] = i$ for all $0 \leq i < n$ (resp. in). In fact, this can be precomputed in $O(n)$ easily. To present this given tree, we will provide a $parent$ array, representing the parent of vertice i. Specifically, $parent[i] = -1$ if vertice i is root. Initially, $parent[i] = -1$ for all $0 \leq i < n$. Algorithm is attached in the Appendix section.

7.3 Correctness

Proof of correctness is almost the same with previous proof. We will omit the detailed proof here.

7.4 Time Complexity

$T(n) = T(m) + T(n - m - 1) + O(1)$ implies $T(n) = O(n)$.

[1] To be more precise, u is on the left side of this given subtree if and only if there exists an ancestor v of this given subtree so that u is on the left subtree of v and this given tree lies on the opposite side.

8 Problem 8

(a) Suppose the pre-, post-, and in-order of u is $u.pre$, $u.post$, $u.in$ separately. The following algorithm will return the size of subtree rooted at u:

- If $u.pre == 1$, return $u.post$.
- If $u == (u.parent).leftchild$, return $((u.parent).rightchild).pre - u.pre$.
- Else, return $u.post - ((u.parent).leftchild).post$.

(b) Suppose the BFS order of a tree T with n nodes is $(v_1, v_2, ..., v_n)$. We encode the tree T into a $\{0, 1\}$ string S_T (indexed begin with 1) with length $2n$ by the following procedure:

- If v_i has left-child, then let $S_T[2i - 1] = 1$, otherwise let $S_T[2i - 1] = 0$.
- If v_i has right-child, then let $S_T[2i] = 1$, otherwise let $S_T[2i] = 0$.

To decode a string into a tree, simply construct the tree layer by layer, i.e., see the code for the first i-th layer and construct the (i+1)-th layer, using the information of left and right children in the i-th layer.

Appendix

Algorithm 2: Weighted Median Search

Input: element array a of length n, weight array w of length n, and threshold t
Output: element a_k such that above requirement satisfied

assert $t > 0$ and $t \leq \sum_i w_i$;
if $n == 1$ **then**
  return a_1;
end
$median = SELECT(a, 1 + n/2)$;
$small\_element = [], large\_element = []$;
$small\_weight = [], large\_weight = []$;
$sum\_small\_weight = 0$;
for $i \leftarrow 1$ ***to*** n **do**
  if $a_i < median$ **then**
    $small\_element.append(a_i)$;
    $small\_weight.append(w_i)$;
    $sum\_small\_weight+ = w_i$
  end
  else
    $large\_element.append(a_i)$;
    $large\_weight.append(w_i)$;
  end
end
if $sum\_small\_weight < t$ **then**
  return $WMS(large\_element, large\_weight, t - sum\_small\_weight)$;
end
else
  return $WMS(small\_element, small\_weight, t)$
end

Algorithm 3: Tree Reconstruction(TR)

Input: array pre, in, rev_pre, rev_in;
integers $left = 0$, $right = n - 1$, $shift = 0$(default value)
Output: array $parent$

```
if l == r then
    return;
end
root = rev_pre[l];
sz = in[root] − shift;
if sz > 0 then
    left_child = rev_pre[l + 1];
    parent[left_child] = root;
    TR(pre, in, rev_pre, rev_in, l + 1, l + sz + 1, shift);
end
if l + sz < r then
    right_child = rev_pre[l + sz + 1];
    parent[right_child] = root;
    TR(pre, in, rev_pre, rev_in, l + sz + 1, r);
end
```