CS5050 Advanced Algorithms

Spring Semester, 2018

Homework Solution 3

Haitao Wang

1. We can use binary search to find the peak entry p in $O(\log n)$ time, as follows. First, we look at the value $A[\frac{n}{2}]$. From this value alone, we cannot tell whether p lies before or after $\frac{n}{2}$, since we need to know whether entry $\frac{n}{2}$ is sitting on an "up-slope" or on a "down-slope". So we also look at the values $A[\frac{n}{2}-1]$ and $A[\frac{n}{2}+1]$. Because the elements of A are distinct, there are now three possibilities.

   - If $A[\frac{n}{2}-1] < A[\frac{n}{2}] < A[\frac{n}{2}+1]$, then entry $\frac{n}{2}$ must come strictly before p, and so we can continue recursively on entries $\frac{n}{2}+1$ through n.
   - If $A[\frac{n}{2}-1] > A[\frac{n}{2}] > A[\frac{n}{2}+1]$, then entry $\frac{n}{2}$ must come strictly after p, and so we can continue recursively on entries 1 through $\frac{n}{2}-1$.
   - If $A[\frac{n}{2}] > A[\frac{n}{2}+1]$ and $A[\frac{n}{2}] > A[\frac{n}{2}-1]$, then we are done: the peak entry p is in fact equal to $\frac{n}{2}$. So we return $\frac{n}{2}$ as the answer.

   In each of these cases, we read at most three entries of A and prune a sub-array of at least half the size of A. We then apply the same algorithm recursively on the remaining sub-array. Hence, the running time can be described by the following recurrence: $T(n) = T(n/2) + O(1)$. Solving the recurrence gives us $T(n) = O(\log n)$.

   Note that the base case happens when the size of the subarray of $A[i \dots j]$ has less than three elements. If the subarray $A[i \dots j]$ has one element, i.e., $i = j$, then return i as the answer. If $A[i \dots j]$ has two elements, then we compare $A[i]$ and $A[j]$. If $A[i] < A[j]$, then return j; otherwise return i.

   The pseudocode is given in Algorithm 1.

2. If the numbers are divided into groups of seven, the algorithm still runs in $O(n)$ time. We prove it below.

   By the similar analysis as in class, we can obtain a new recurrence: $T(n) = T(n/7) + T(5n/7) + n$ for the running time of the algorithm. We can use the substitution method to prove $T(n) = O(n)$, as follows.

   Guess: We guess $T(n) = O(n)$. In other words, we want to prove there exit constants c and n_0, such that $T(n) \leq c \cdot n$ for all $n \geq n_0$.

Algorithm 1: BinarySearchPeakEntry(A, i, j)

Input: A subarray $A[i, j]$, and initially, $i = 1$ and $j = n$

Output: The index of the peak entry

```
if i = j then
    return i;
end
if j = i + 1 then
    if A[i] < A[j] then
        return j;
    else
        return i;
    end
end
k ← ⌊(i + j)/2⌋;
if A[k − 1] < A[k] < A[k + 1] then
    return BinarySearchPeakEntry(A, k + 1, n);
end
if A[k − 1] > A[k] > A[k + 1] then
    return BinarySearchPeakEntry(A, 1, k − 1);
end
if A[k − 1] < A[k] and A[k] > A[k + 1] then
    return k;
end
```

Verification: We assume the above is true for $T(n/7)$ and $T(5n/7)$, i.e., $T(n/7) \le c \cdot \frac{n}{7}$ and $T(5n/7) \le c \cdot \frac{5n}{7}$. Then, we can obtain the following:

$$T(n) \le c \cdot \frac{n}{7} + c \cdot \frac{5n}{7} + n = c \cdot \frac{6n}{7} + n$$

Our goal is to find c and n_0 such that $T(n) \le cn$ holds for all $n \ge n_0$. Now that $T(n) \le c \cdot \frac{6n}{7} + n$, to prove $T(n) \le cn$, it is sufficient to prove $c \cdot \frac{6n}{7} + n \le cn$, or equivalently to prove $n \le \frac{c}{7} \cdot n$. If we let $c = 7$ and $n_0 = 1$, then clearly $n \le \frac{c}{7} \cdot n$ holds for all $n \ge n_0$.

We conclude that our guess that $T(n) = O(n)$ is correct.

3. Suppose p is the well whose y-coordinate is the $\lceil n/2 \rceil$-th largest among the y-coordinates of all n wells. Then, the optimal location for the main pipeline has the y-coordinate equal to the y-coordinate of p, and in other words, the main pipeline should pass through p.

   More specifically, if n is an odd number, then the main pipeline should pass through p. If n is an even number, and suppose p' is the well with the $(\lceil n/2 \rceil + 1)$-th largest y-coordinate, then any location between p and p' is an optimal location for the main pipeline.

   The reason is the following. Suppose L is the pipeline determined by the above rule. The key observation is that if we move L horizontally upwards or downwards, the total sum of the lengths of the spur pipelines is always monotonically non-decreasing. This implies that L is located at an optimal location.

According to the discussion above, to find an optimal location for the main pipeline, we only need to find the median of the y-coordinates of all n wells. We can use the SELECTION algorithm to find the median in $O(n)$ time.

4. (a) We first sort all elements of A. Then, by scanning the sorted list once, we can find the k_i-th smallest number in A for all $i = 1, 2, \ldots, m$. The running time is dominated by the sorting step, which takes $O(n \log n)$ time.

   (b) For each $1 \leq i \leq m$, we use the linear-time selection algorithm to find the k_i-th smallest number of A. The total time is thus $O(nm)$.

   (c) We use the linear-time selection algorithm and the divide-and-conquer technique.

   For each $1 \leq i \leq m$, let a_i denote the k_i-th smallest number of A. Our goal is to find $a_1, a_2, \ldots, a_m$.

   We first find $a_{\frac{m}{2}}$ in linear time by using the selection algorithm. Let A_1 be the set of all elements of A that are smaller than $a_{\frac{m}{2}}$ and A_2 be the set of all elements of A that are larger than $a_{\frac{m}{2}}$. After $a_{\frac{m}{2}}$ is computed, we can compute A_1 and A_2 in linear time by comparing each element of A with $a_{\frac{m}{2}}$. Then, the observation is that $a_1, a_2, \ldots, a_{\frac{m}{2}-1}$ are all in A_1 and $a_{\frac{m}{2}+1}, a_{\frac{m}{2}+2}, \ldots, a_m$ are all in A_2.

   Based on the observation, we continue to find $a_1, a_2, \ldots, a_{\frac{m}{2}-1}$ in A_1 *recursively*, and find $a_{\frac{m}{2}+1}, a_{\frac{m}{2}+2}, \ldots, a_m$ in A_2 *recursively*. Note that for each $1 \leq i \leq \frac{m}{2} - 1$, a_i is still the k_i-th smallest number in A_1, but for each $\frac{m}{2} - 1 \leq i \leq m$, a_i is actually the $[k_i - (|A_1| + 1)]$-th smallest number in A_2, where $|A_1|$ is the size of the set A_1.

   For the running time, the algorithm has $O(\log m)$ "levels" of recursive steps and each level takes $O(n)$ time in total. Hence, the total time of the algorithm is $O(n \log m)$.

   If we use the recurrence to describe the running time, it is the following, where t is the size of the first subset A_1 (and thus the size of A_2 is $n - t - 1$, but we use $n - t$ in the recurrence for simplicity).

$$T(m, n) = \begin{cases} T(\frac{m}{2}, t) + T(\frac{m}{2}, n - t) + O(n), & \text{if } m \geq 2, \\ O(n) & \text{if } m \leq 1. \end{cases}$$

   The easiest way to solve the recurrence is by recursion tree: the total time of each level of the recursion tree is $O(n)$ and the recursion tree has $O(\log m)$ levels.