CSE 6331 HOMEWORK 4

BRENDAN WHITAKER

1. *Write a recursive, divide-and-conquer algorithm Power(a, n) that computes the number a^n, where a, n are positive integers. Analyze your algorithm. Your algorithm must work in $o(n)$ time.*

Algorithm 1: Power(a, n)

Data: a, n both positive integers.
Result: a^n a positive integer.
1 **begin**
2   **if** $n = 0$ **then**
3   **return** 1;
4   **else if** $n = 1$ **then**
5   **return** a;
6   **end**
7     $rem \longleftarrow n/2 \mod 2$;
8     $n \longleftarrow \lfloor n/2 \rfloor$;
9   **if** $rem = 0$ **then**
10   **return** Power$(a \cdot a, n)$;
11   **else**
12   **return** $a\cdot$Power$(a \cdot a, n)$;
13   **end**
14 **end**

Note that lines 2-9 take constant time, so our running time is given by:

$$T(n) = c + T(n/2). \tag{1}$$

And so by the Master Theorem, since $c \approx n^{log_2(1)} = c'$, we know $T(n) \in \Theta(clogn)$. And since:

$$\lim_{n \to \infty} \frac{clogn}{n} = 0,$$

we know $T(n) \in o(n)$.

Date: SP18.

2. *Rewrite your algorithm Power(a, n) as a non-recursive (iterative) one. (The running time must still be $o(n)$.)*

Algorithm 2: Power(a, n)

Data: a, n both positive integers.
Result: a^n a positive integer.

```
1  begin
2      b ⟵ a;
3      while n > 1 do
4          rem ⟵ n/2  mod 2;
5          n ⟵ ⌊n/2⌋;
6          if rem = 0 then
7              b ⟵ b^2;
8          else
9              b ⟵ b^3;
10         end
11     end
12     if n = 0 then
13         return 1;
14     else if n = 1 then
15         return b;
16     end
17 end
```

Note lines 2,12-16 take constant time, and lines 3-11 take $log_2(n)$ time. So the total running time $T(n) \in \Theta(logn)$, and by the same argument used above, we know $T(n) \in o(n)$.

3. *Consider the closest-pair algorithm. Suppose we do not sort $A[i..j]$ by y-coordinate in Closest-Pair$(A[i..j], (p, q), ptr)$, but instead we sort the whole set of n points (i.e., $A[1..n]$) by y-coordinate into a linked list in the beginning of the algorithm, immediately after sorting them by x. (The procedure Closest-Between-Two-Sets remains intact, but the linked list pointed to by ptr now contains the entire set of n points.) Does the modified algorithm work correctly? Justify your answer. (If YES, explain why; if NO, give a counterexample.)*

Yes, the algorithm is still correct **(wrong)**. Note that when the Closest-Pair algorithm calls Closest-Pair-Between-Two-Sets, we already have *ptr* set to the point in $A[i..j]$ with the least y-coordinate, since our base case statments in Closest-Pair make sure that *ptr* is set to the one with least y-coord of the one/two points being considered, and the Merge function sets *ptr* to the least of the two $ptr1, ptr2$ values being considered. So k is initialized to *ptr*. Now in the while loop, we check if k is between L_1, L_2, and since only points in $A[i, j]$ will be in this range **(this is wrong because although the array $A[1..n]$ is sorted by x coordinate, we can still points from outside $A[i..j]$ in the interval from L_1 to L_2. You should draw up this counterexample.** (it's centered about m and has width $\leq \delta$), we know we will only be considering k in $A[i..j]$. If k is outside of $A[i..j]$ then still in the while loop, we assign $k \longleftarrow Link[k]$, and this point may be outside $A[i..j]$ now, since we are dealing with the entire $A[1..n]$ sorted by y in the lined list now. But again, when we check if k is between L_1, L_2 we again exclude all points outside of $A[i..j]$ so the algorithm runs correctly, since past this point it is unchanged from before.

4. *Whether the above algorithm is correct or not, what is its time complexity?*

Note since we are sorting the entire set $A[1..n]$ of points before we begin Closest-Pair, we have to add the time for mergesort with a linked list to our old running time for $T(n)$, the closest pair algorithm. So our new algorithm takes $T'(n)$ time where $T'(n) \in \Theta(nlogn + T(n))$, but since $T(n) \in \Theta(nlogn)$ we know that $T'(n) \in \Theta(nlogn)$.

5. *Let $A[1..n], B[1..n]$ be two arrays of integers, each sorted in nondecreasing order. Write a divide-and-conquer algorithm that finds the n-th smallest of the $2n$ combined elements. Your algorithm must run in $O(logn)$ time. You may assume that all the $2n$ elements are distinct. (Write your algorithm in pseudo-code and explain it in plain English.) Note: The input consists of two arrays of the same size. When you divide the problem, make sure that the two (sub)arrays of each subproblem are of equal size.*

Algorithm 3: n-smallest(i_a, j_a, i_b, j_b)

Data: Two sorted integer arrays:$A[i_a..j_a], B[i_b..j_b]$.
Result: Then n-th smallest element of the total $2n$ elements of both arrays.

1 **begin**
  `/*` The n-th smallest element is in $A[i_a..j_a] \cup B[i_b..j_b]$. `*/`
  `/*` You should always keep $i_a - j_a = i_b - j_b$. `*/`
2   **global array** $A[1..n], B[1..n]$;
3   **if** $i_a = j_a$ *and* $i_b = j_b$ **then**
4   **return** $min(A[i_a], B[i_b])$;
    `/*` return $A[i_a]$ or $B[i_b]$ `*/`
5   **else**
6     $m_a \longleftarrow i_a + \lfloor (j_a - i_a)/2 \rfloor$;
    `/*` $i_a \leq m_a \leq j_a$ `*/`
7     $m_b \longleftarrow i_b + \lfloor (j_b - i_b)/2 \rfloor$;
    `/*` $i_b \leq m_b \leq j_b$ `*/`
8   **if** $A[m_a] < B[m_b]$ **then**
9   **return** *n-smallest*$(i_a + \lfloor (j_a - i_a + 1)/2 \rfloor, m_a + \lfloor (j_a - i_a + 1)/2 \rfloor, i_b, m_b)$;
      `/*` recursively call n-smallest `*/`
10   **else**
11   **return** *n-smallest*$(i_a, m_a, i_b + \lfloor (j_b - i_b + 1)/2 \rfloor, m_b + \lfloor (j_b - i_b + 1)/2 \rfloor)$;
      `/*` recursively call n-smallest `*/`
12   **end**
13   **end**
14 **end**

When $n = 1$, since we are finding the n-smallest element, we are just finding the smallest element, so we take the minimum of the two 1-element arrays on line 4. We initialize m_a to be the midpoint of array A, taking the floor if the array has an even number of entries, and do the same for m_b. If $A[m_a] < A[m_b]$, this means the midpoint of A is smaller than the midpoint of B. So in this case, when n is odd, m_a is at most the n-smallest element in the two arrays, and when n is even it is at most the $n-1$-smallest element. So when n is odd, we have $j_a - i_a + 1 = n$, so $i_a + \lfloor (j_a - i_a + 1)/2 \rfloor = i_a + \lfloor n/2 \rfloor$, which is exactly the index of the center element in A, which is m_a. So since m_a is at most the n-smallest element in this case, we know that the n-smallest element is still in $A[m_a..j_a]$, which is the new array being passed to the recursive call. Similarly, the elements eliminated by passing $B[i_b, m_b]$ cannot contain the n-smallest element since $B[m_b] > A[m_a]$, and by a similar argument, passing $A[i_a, m_a], B[i_b + \lfloor (j_b - i_b + 1)/2 \rfloor, m_b + \lfloor (j_b - i_b + 1)/2 \rfloor)]$ to the recursive call when $A[m_a] \geq B[m_b]$ also preserves the n-smallest element.