Homework #2
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2013

Solutions

February 20, 2014

1 Problem 1 (20 points)

1.1 15 points

Given a set of numbers $x_1, x_2, \ldots, x_n$, μ_∞ is the value of μ that minimizes the quantity $\max_i |x_i - \mu|$. That is

$$\mu_\infty = \min_\mu \max_{1 \le i \le n} |x_i - \mu|.$$

Give an algorithm that computes μ_∞ for a set of n numbers in $O(n)$ time.

Solution: let us first note that the minimization problem is solved by

$$\mu_\infty = \frac{x_m + x_M}{2},$$

where $x_m = \min_{1 \le i \le n} x_i$ and $x_< = \max_{1 \le i \le n} x_i$. To see that this is the case, consider $\mu = (x_m + x_M)/2 + \delta$, where δ is some real number. Then we can rewrite our problem as

$$\mu_\infty = \min_\delta \max_{1 \le i \le n} |x_i - (\frac{x_m + x_M}{2} + \delta)|.$$

Using the triangle inequality, we have

$$|x_i - (\frac{x_m + x_M}{2} + \delta)| \le |x_i - \frac{x_m + x_M}{2}| + |\delta|,$$

and since $|x_i - \frac{x_m+x_M}{2}|$ is maximized when $x_i = x_m$ or $x_i = x_M$, this reduces to solving

$$\min_\delta |x_m - \frac{x_m + x_M}{2}| + |\delta| = |x_m - \frac{x_m + x_M}{2}| + \min_\delta |\delta|,$$

which is indeed minimized when $\delta = 0$, so $\mu_\infty = \frac{x_m+x_M}{2}$

It remains to devise an algorithm to calculate $(x_m + x_M)/2$ for a given set of numbers $x_1, x_2, \ldots, x_n$. This can be done in linear time by traversing the list of numbers once, keeping track of the smallest and largest numbers seen so far. When we reach the end of the list, we simply add these two numbers and divide by two. This algorithm runs in $O(n)$ time, since it requires observing each of the n numbers in the list and comparing each to 2 numbers (the minimum and maximum numbers seen so far).

1.2 5 points

In class, we discussed the closest-pair problem for points in $\mathbb{R}^2$. What about when our points are in $\mathbb{R}$? That is, given a list of reals numbers $x_1, x_2, \ldots, x_n$, find the value

$$\min_{i \neq j} |x_i - x_j|.$$

Given an algorithm that computes this value in $O(n \log n)$ time. Prove the correctness of your algorithm and prove that its runtime is indeed $O(n \log n)$.

Solution: it suffices to sort the numbers using merge sort, which requires $O(n \log n)$ time, and then to proceed by simply looking at each adjacent pair of numbers in the list and tracking the smallest seen so far. This part of the algorithm can be done in linear time. Thus, the algorithm requires $O(n \log n)$ time in total.

To see the correctness of this algorithm, let us first introduce a bit of notation. Let $x_{(1)}, x_{(2)}, \ldots, x_{(n)}$ denote our input after it has been ordered. Thus, $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$. Let (x_a, x_b) be the closest pair in the input, with distance $|x_a - x_b|$. Suppose without loss of generality that $x_a \leq x_b$. We need to show that the linear scan step of the proposed algorithm, in which we observe value $|x_{(i)} - x_{(i+1)}|$ for $i = 1, 2, \ldots, n-1$, does indeed observe this value. But this follows immediately from the fact that for all $i \neq j$

$$|x_a - x_b| \leq |x_i - x_j|.$$

2 Problem 2 (20 points)

Given a sequence of distinct integers $a_1, a_2, \ldots, a_n$ sorted in ascending order (so that $a_1 < a_2 < \cdots < a_n$), devise an algorithm that returns True if there exists

an element in the sequence such that $a_i = i$, and returns false if no such element exists. Prove the correctness of your algorithm and give an analysis of its runtime. Your algorithm must run in $O(\log n)$ time for full credit.

Solution: Let us first note the following facts:

1. We must have $a_n - a_1 \geq n - 1$, since $a_1, a_2, \ldots, a_n$ are all distinct and increasing.

2. If $i < j$, $a_i < i$ and $a_j < j$, then for any integer k for which $i \leq k \leq j$, we must have $a_k < k$. To see this fact, suppose that there existed some k, $i \leq k \leq j$, for which $k \leq a_k$. By assumption, $a_i < a_k < a_j$. If indeed we have $k \leq a_k < a_j < j$, then consider a_{k+1}. We must have $a_{k+1} \geq k + 1$, since $a_{k+1} > a_k \geq k$. Similarly, we must have $a_{k+2} \geq k+2$, $a_{k+3} \geq k+3$, and so on, up to $a_j \geq j$, a contradiction.

3. If $i < j$, $a_i > i$ and $a_j > j$, then for any integer k for which $i \leq k \leq j$, we must have $k < a_k$. This fact follows from reasoning similar to the previous statement, except with inequalities reversed.

With these facts in hand, we can show that the following algorithm will suffice.

```
1: ℓ ← 1
2: u ← n
3: while ℓ ≤ u do
4:    if (a_ℓ = ℓ or a_u = u) then
5:       return true
6:    else if (a_ℓ < ℓ and a_u < u) or (a_ℓ > ℓ and a_u > u ) then
7:       return false
8:    else
9:       i ← ⌊n/2⌋
10:      if a_i = i then
11:         return true
12:      else if (a_i < i and a_u < u) or (a_i > i and a_u > u) then
13:         u ← i − 1
14:      else
15:         ℓ ← i + 1
16:      end if
17:   end if
18: end while
19: return false
```

Proof of correctness: Suppose that no index i exists for which $a_i = i$. Then the conditional test for $a_i = i$ in line 10 of the above algorithm will never evaluate to true, nor will the conditional test for $a_\ell = \ell$ or $a_u = u$ in line 4. Since these are the only ways that the algorithm can return true, we need only show that the algorithm terminates when no such index exists. This follows from the fact that at every step of the algorithm, if $a_i \neq i$, then either u is decreased or ℓ is increased. Since this happens at every step, we must eventually reach a point where either ℓ is increased or u is decreased such that we have $\ell > u$, causing the while-loop to terminate and the algorithm to return false. Note that the algorithm may return false long before this, if it is indeed the case that every element of the sequence has $a_i > i$ or every element has $a_i < i$, but in the case where this is does not hold, the termination of the while-loop guarantees that the algorithm terminates eventually.

In the event that some index k exists for which $a_k = k$, we need to show that either line 4 or line 10 does indeed get evaluated to true. We will do this via a proof by contradiction. Toward this end, suppose that our algorithm terminates by returning false. This can occur either because we performed enough iterations of the while-loop for it to terminate, as argued above, or because line 6 evaluated to true. If we have $a_k = k$ for some index k, then $\ell \leq k \leq u$ precludes line 6 from evaluating to true, since if line 6 evaluates to true, there cannot be an index between ℓ and u for which $a_k = k$, as proved in our facts above. Thus, for line 6 to evaluate to true we must have either $k < \ell$ or $u < k$. But due to the initial values of ℓ and u, there must have existed at least one iteration of the while-loop at which $\ell \leq k \leq u$. At the final such iteration, either $\ell \leq i \leq k \leq u$ or $\ell \leq k \leq i \leq u$. In the former case, for ℓ and u to be updated so that $k \notin [\ell, u]$, it must have been that either $a_i < i$ and $a_u < u$ or that $a_i > i$ and $a_u > u$. In either of these cases, this is a contradiction of the fact that $i \leq k \leq u$, since the fact that $a_k = k$ and $i \leq k \leq u$ precludes $a_i - i$ and $a_u = u$ having the same sign. In the latter case, for ℓ and u to be updated so that $k \notin [\ell, u]$, it must have been that (i) it was not the case that $a_i < i$ and $a_u < u$ (ii) it was not the case that $a_i > i$ and $a_u > u$. (iii) it was not the case that $a_u - u$ and $a_\ell - \ell$ had the same sign (iv) $a_\ell \neq \ell$ and $a_u \neq u$ and (v) $a_i \neq i$. Thus, $a_i - i$ and $a_u - u$ have different signs and are both non-zero. Further, $a_\ell - \ell$ is non-zero. By the pigeonhole prinicple, two of the three quantities $a_\ell - \ell$, $a_i - i$ and $a_u - u$ must have the same sign. Further, since $\ell \leq k \leq i$, $a_\ell - \ell$ and $a_i - i$ must have different signs (else it would not be possible that $a_k = k$). Thus, $a_\ell - \ell$ and $a_u - u$ must have the same sign, contradicting the fact that $\ell \leq k \leq u$.

In the event where we performed enough iterations of the while-loop for it to terminate, the same reasoning as above shows a contradiction, since each iteration of the while-loop that does not result in a return statement updates either ℓ or u.

Runtime analysis: the algorithm terminates in $O(\log n)$ time. To see this, not that in the worst case, we iterate through the while-loop until $u < \ell$. At every

such iteration, we update either u or ℓ so that the difference between u and ℓ is approximately halved. Initially, $u - \ell = n - 1$, so we can perform this update at most $\log(n-1)$ times before we are forced to set $u < \ell$ and terminate the loop. Thus, our algorithm has an $O(\log n)$ worst-case runtime.

2.1 5 points

An array of numbers A is *almost sorted* if for every $1 \leq i \leq \sqrt{n} \leq j$, we have $A[i] \leq A[j]$ and for every $\sqrt{n} \leq j \leq k \leq n$ we have $A[j] \leq A[k]$. Give an algorithm that takes as input an almost sorted array A and sorts A in $o(n)$ time.

Solution: It suffices to only sort the first $\sqrt{n}$ entries of A, which can be done in $O(\sqrt{n}\log\sqrt{n})$ time using mergesort. This runtime is $o(n)$. The correctness of the algorithm follows from the fact that by assumption A is sorted except for its first $\sqrt{n}$ elements, so sorting the first $\sqrt{n}$ elements suffices to make A fully sorted.

3 Optional Problems

Solve the following problems from CLRS: $3.1, 3.5, 4.1, 7.2$