# Assignment 3: Sample solutions (and comments)

1. (a) Consider the following algorithm:

**Algorithm 1** Max&Min($\{x_1, x_2, \ldots x_n\}$)

1: **for** $i = 1, \ldots, \lfloor n/2 \rfloor$ **do**
2: compare $x_i$ and $x_{i+1}$, putting the larger into the set MAXcontenders and the smaller into the set MINcontenders
3: **if** $n$ is odd **then**
put $x_n$ into both MAXcontenders and MINcontenders
4: **end if**
5: **end for**
6: determine MAX, the largest of MAXcontenders using |MAXcontenders| $- 1$ comparisons
7: determine MIN, the smallest of MINcontenders using |MINcontenders| $- 1$ comparisons
8: **return** MAX and MIN.

The analysis (separating the cases where $n$ is odd or even) is straightforward.

(b) Despite the hint, several people seemed confused about how to structure the induction argument. The basis of the induction is the case where $f(A, B, C) = 0$. Since both $|B|$ and $|C$ remain greater than zero after the first comparison, we can reach a state where $f(A, B, C) = 0$, only when $|A| = n = 1$, or $|A| = 0$ and $|B| = |C| = 1$. In either case, there remains just one candidate for each of max and min, so no more comparisons are needed (which establishes the basis of the induction).

For the induction step, which most people seeemed more comfortable with, we observe that for all possibilities of the next comparison (I will not go through all of them here), following the rules of the adversary ensures that $f(A', B', C')$, the value of $f$ on the updated sets, is reduced by at most 1. (For example, if the next comparison is of type A:A, then $|A'| = |A| - 2$, $|B'| = |B| + 1$ and $|C'| = |C| + 1$, yielding $f(A', B', C') = f(A, B, C) - 1$.) So, by the induction hypothesis, we have $f(A, B, C) - 1$ more comparisons remaining.

(c) The lower bound follows by observing that at the start of the algorithm $|A| = n$ and $|B| = |C| = 0$.

2. (a)

   (b) Consider the following algorithm:

**Algorithm 2** UnaryMax($S$)

```
1: determine the value of x_1 (by binary search) and assign this value to MAX
2: for i = 2, ..., n do
3:    if x_i > MAX then
         determine the value of x_i, and assign this value to MAX
4:    end if
5: end for
6: return MAX.
```

Each time $MAX$ is updated the binary search uses $O(\lg m)$ unary predicate evaluations. Since there are at most $n-1$ such updates, in the worst case, the total number of unary predicate evaluations is $O(n \lg m)$.

(c) A simple adversary can force $\Omega(n + \lg |\mathcal{U}|)$ comparisons as follows: The adversary maintains the intersection of the uncertainty intervals associated with elements in $S$ (initially $[0, m-1]$). Each time a question of the form "$x_i \geq c_j$?" is asked the adversary responds in a way that reduces the size of this intersection by at most one half. After $(\lg m) - 1$ comparisons the this intersection has been reduced to something that has size at least 2. Suppose that the resulting intersection is $[c_{min}, c_{max}]$. At this point the adversary will respond: (i) "no" to all queries of the form "$x_i \geq c$?", where $c > c_{max}$, (ii) "yes" to all queries of the form "$x_i \geq c$"?, where $c \leq c_{max} - 1$, and (iii) "no" to the first $n-2$ queries of the form "$x_i \geq c_{max}$?". At this point the adversary is free to make any one of the (at least two) elements that has not been compared with $c_{max}$ take the value $c_{max}$, and all the rest take the value $c_{max} - 1$. Thus at least one more comparison must be made, ensuring a total of at least $\lg m + n - 2$ comparisons.

3. (a) Following the hint, we start by designing an algorithm whose search cost, when the algorithm returns index $i$, is proportional to $1 + \lg i$. (This essentially treats the special case when $\ell(j-1) = 1$.) With this in mind, we observe that if the positions of $D$ are partitioned into $\Theta(\lg n)$ blocks, where the $k$-th block consists of indices $i \in (2^{k-1}, 2^k]$, then the search cost when the algorithm returns an index in the $k$th block should be proportional to $k$. A reasonable hybrid strategy that achieves this goal is to (i) use linear (sequential) search to find the block that contains the query, then (ii) use binary search within the block to locate the correct answer.

   This search strategy, summarized in pseudo-code below, achieves $\text{cost}_{\mathcal{S}}(i) \leq 2k$, for $i \in (2^{k-1}, 2^k]$ ($k$ steps to locate the correct block and $k$ more to do the binary search). In other words, $\text{cost}_{\mathcal{S}}(i) = \Theta(1 + \lg i)$.

**Algorithm 3** Search($D$, $x$)

```
1: n ← length(D)
2: if x > D[n] then
3:    return n
4: end if
5: if x ≤ D[1] then
6:    return 1
7: end if
8: i ← 2
9: while x > D[i] do
10:   i ← min(i * 2, n)
11: end while
12: return BinarySearch D between i/2 and i.
```

Now to treat the more general case (where $\ell(j-1)$ is not necessarily 1) we (conceptually) form blocks, of increasing powers of two in size, to both the left (lower indices) and right (higher indices) of $D[\ell(j-1)]$. One comparison (with $D[\ell(j-1)]$) determines which of these two sub-arrays contains the desired answer. Thereafter we proceed as in the pseudo-code above (with increasing or decreasing indices, as appropriate). If the result $\ell(j)$ satisfies $|\ell(j) - \ell(j-1)| \in (2^{k-1}, 2^k]$, then it is discovered in at most $2k = \Theta(\lg \Delta_j)$ additional comparisons.

(b) It is easy to see, by induction on $d$, that any binary tree has at most $2^d$ *nodes* at depth $d$; thus it certainly has at most $2^d$ *leaves* at this depth.

It follows from this that a binary tree has no more than $\sum_{d \le k} 2^d = 2^{k+1} - 1$ leaves at depth less than or equal to $k$. (Of course, we can give a tighter bound than this, but this is all we need for this question.) Thus, any algorithm that correctly answers queries leading to locations in the range $[\ell(j-1) - 2^k,\ \ell(j-1) + 2^k]$ must use more than $k$ comparisons for at least some of these queries. What this says is that there is a limit to how much one can exploit locality of reference in the worst case: if the next query has distance $\Delta \in [0, 2^k]$ from its predecessor then we need at least $\log \Delta$ comparisons for at least some such queries.

4. The data structure developed in this question corresponds to the static version of a more general structure presented in the paper:
Bose, P., Douieb, K., Dujmovic, V., Howat, J. and Morin, P., "Fast local searches and updates in bounded universes", Proceeding of CCCG 2010, Winnipeg, MB, August 2010.

(a) When the query is an element of $S$ then it the lowest marked node on the path to the root. Its predecessor and successor values are stored at the element itself, so they can be retrieved in $O(1)$ time.

(b) Let $a$ = ancestor$(x, h)$, $b$ = left-neighbour(ancestor$(x, h)$) and $c$ = right-neighbour(ancestor$(x, h)$). We know, by the absence of marks, that the tree rooted at $a$, $b$, and $c$ all have no elements of $S$ at their leaves. Thus, no matter where $x$ lies in the tree rooted at $a$, it must have at least $2^h$ unmarked leaves to both its left and right. (Here I have assumed that $u$ is a power of 2, so the bottom level of the tree is full.)

(c) Suppose that $a$ is unmarked but $c$ is marked. Then, since no leaves in the tree rooted at $a$ belong to $S$, the successor of $x$ must be the leftmost marked leaf in the subtree rooted at $c$. This can be found in $O(1)$ time by following the min-pointer stored at $c$. Furthermore if the marked leaves (elements of $S$) are doubly linked, we can find the predecessor as well, once we have the successor. (Of course, if $b$ is marked instead of $c$, a symmetric argument applies).

(d) This is *exactly* what is done in $x$-fast tries, as described in class: we do binary search on the levels between 0 and $h$ to find the lowest marked ancestor. From this we can find the successor (or predecessor) by following the associated min (or max) pointer. The predecessor (or successor) can then be found as in part (b).

(e) We start by performing a *doubling search* (as in Q 1(a) above) along the path from $x$ to the root, i.e. we look at ancestor$(x, h)$ for $h = 2^i, i = 0, 1, \ldots$. We stop when either (i) ancestor$(x, h)$ is unmarked but at least one of left-neighbour(ancestor$(x, h)$) or right-neighbour(ancestor$(x, h)$) is marked, or (ii) ancestor$(x, h)$ is marked. In either case this takes $\lg h$ steps which is $O(\lg \lg \Delta)$ by part (b). In case (i), we can find the predecessor and successor of $x$ in $O(1)$ additional time, by part (c), and in case (ii), we can find the predecessor and successor of $x$ in $O(\lg h) = O(\lg \lg \Delta)$ additional time, by part (d).