Assignment 2: Sample solutions and comments

1. Consider the following algorithm:

Algorithm 1 UnaryMax(S)

```
randomly reorder the elements $x_1, x_2, \ldots, x_n$ of $S$
determine the value of $x_1$ (by binary search) and assign this value to $MAX$
for $i = 2, \ldots, n$ do
    if $x_i > MAX$ then
        determine the value of $x_i$, and assign this value to $MAX$
    end if
end for
return $MAX$.
```

Each time MAX is updated the binary search has a cost of $O(\lg m)$. But, by the analysis of the hiring problem (done in class) the expected number of times that MAX is updated is $O(\lg n)$. Thus, the expected total cost of the algorithm above is $O(n + \lg n \cdot \lg m)$.

2. (a) After thinking about this for a bit it is natural to consider a strategy of the form:

Algorithm 2 α-sampling-strategy

```
for the first $\alpha n$ of the numbers do
    respond “no”, but
    keep track of the largest number $MAX$ seen so far
end for
for the remaining numbers in sequence do
    if $x_i > MAX$ then
        respond “yes” and stop
    end if
end for
```

It turns out that this strategy is successful in choosing the maximum with probability very close to $\alpha \ln \frac{1}{\alpha}$. (This analysis is described in detail in section 5.4.4 of the Cormen et al text.) This probability is maximized (as $1/e \approx 0.3678$) when α is chosen to be $1/e$.

(b) The analysis is particularly simple when we choose $\alpha = 0.5$. In this case, the strategy is certain to succeed if the second smallest number appears in the first half of the inputs and the largest number appears in the second half of the inputs. Since each of these events occurs with probability 1/2, for a randomly chosen input sequence, the probability of them both occurring is at least 1/4. [Note the events are not exactly independent, but the probability that one occurs, given that the other occurs, is slightly larger than the unconditional probability.]

3. In addition to the arrays A and B, we use one additional variable n that denotes the current size of the dynamic set S. Initially, $n = 0$, and the invariant holds trivially. Here is pseudocode for the desired operations:

Algorithm 3 member(i)

```
1: $j \leftarrow A[i]$
2: if $j < 0$ or $j > n - 1$ then
3:   return false
4: else
5:   return $B[j] = i$
6: end if
```

Algorithm 4 insert(i)

```
1: if not(member($i$)) then
2:   increment $n$
3:   $B[n-1] \leftarrow i$
4:   $A[i] \leftarrow n - 1$
5: end if
```

Algorithm 5 delete(i)

```
1: if member($i$) then
2:   $A[B[n-1]] \leftarrow A[i]$
3:   $B[A[i]] \leftarrow B[n-1]$
4:   decrement $n$
5: else
6:   return error
7: end if
```

The correctness of member is immediate from the invariant. (Note that it is essential to check that $0 \leq j \leq n-1$ holds.) For the correctness of insert and delete it suffices to confirm that they preserve the invariant. This is particularly straightforward for insert; for delete it is only slightly more involved (the special case where $i = B[n-1]$ is interesting and should not be overlooked). Note that if we did not care about preserving the invariant $n = |S|$ (i.e. reclaiming the space in B) then we could delete element i by simply setting $A[i]$ to -1.

4. (a) Following the hint, we start by designing an algorithm whose search cost, when the algorithm returns index i, is proportional to $\lg(1+i)$. (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 kth 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 $\text{cost}_{\mathcal{S}}(i) \le 2k$, 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 6 Search($D[1:n]$, x)

```
if $x > D[n]$ then
    return $n$
end if
if $x \le D[1]$ then
    return 1
end if
$i \leftarrow 2$
while $x > D[i]$ do
    $i \leftarrow \min(i * 2, n)$
end while
return BinarySearch for $x$ in $D[i/2 : i]$.
```

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 subarrays 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 $2k = \Theta(\lg(2+\Delta_j))$ 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: for at least some queries with distance $\Delta \in [0, 2^k]$ from their predecessor we need at least $k \geq \log(2 + \Delta)$ comparisons.