Solutions for Homework 2

1. Suppose we want to find the \( k \) smallest numbers in a list of \( n \) numbers, where \( k = \lfloor n / \log n \rfloor \). Design an algorithm that solves this task in worst-case time that is linear in \( n \).

**Solution:** We build a min-priority queue (using a min-heap) out of these elements in \( O(n) \) time. Then we repeat the following step \( k \) times: we extract the minimum from the min-priority queue. This can be done in \( O(\log m) \) time if there are \( m \) numbers left in the priority queue. Thus the total running time is

\[
O \left( \sum_{i=0}^{k-1} \log (n - i) \right) = O \left( \sum_{i=0}^{k-1} \log n \right) = O(k \log n) =
\]

\[
= O \left( \frac{n}{\log n} \log n \right) = O(n).
\]

(20 points)

2. In class we showed using an adversary argument that any algorithm to compute the MAX and the MIN of a set of \( n \) distinct numbers simultaneously using pairwise comparisons must, in the worst-case, use at least \( \lceil 3n/2 \rceil - 2 \) comparisons.

(a) Use the decision tree argument (finding a lower bound on the number of possible responses, which is a lower bound on the number of leaves, and using this as a bound on the height of the tree) to develop a worst-case lower bound on the number of pairwise comparisons.

(b) If this bound is different than \( \lceil 3n/2 \rceil - 2 \), explain how seemingly contradictory bounds can both be correct.

**Solution:**
(a) There are \(n(n-1)\) possible responses, and hence every decision tree must have at least \(n(n-1)\) leaves. This implies that the depth of any tree (corresponding to the worst-case number of comparisons) must be at least

\[
\log(n(n-1)) \geq \log n.
\]

(b) Bounds can be different without contradicting each other. The optimal worst-case number of comparisons, \([3n/2] - 2\), is indeed greater than or equal to both lower bounds. (15 points)

3. You are given a pile of phone bills and a pile of checks sent in to pay the bills. Assume that the phone numbers are on the checks. Assume that the number of bills is roughly the same as the number of checks and let this number be \(n\). Design an algorithm that finds all the bills that were unpaid, i.e. bills for which there is no corresponding check. What is your worst-case time complexity in terms of \(n\)?

**Solution:** Sort both the bills and the checks by phone number; then check through the piles for bills without corresponding checks. This method does \(O(n \log n)\) steps. An alternative solution would be to sort just the bills, then do binary search for each check, marking the bills that are paid. One sequential pass through the bills finds those that are unpaid. Again, the number of steps is in \(O(n \log n)\). (15 points)

4. Show that the second largest element can be found with \(n + \lceil \log n \rceil - 2\) comparisons in the worst case.

**Solution:** We will find MAX first by using the tournament method. Elements are paired off and compared in rounds. In each round after the first one, the winners from the preceding round are paired off and compared. (If at any round the number of keys is odd, then one of them simply waits for the next round.) We can describe this tournament by a tree, each leaf contains an element, and at each subsequent level the parent of each pair contains the winner. The root contains MAX. We have \(n - 1\) comparisons in total.

In the process of finding MAX, every element except MAX loses in one comparison. The second largest element must lose directly to MAX. Since MAX is involved in at most \(\lceil \log n \rceil\) comparisons, the second largest must be one of at most \(\lceil \log n \rceil\) elements. We find the maximum
of these by \( \lceil \log n \rceil - 1 \) comparisons, that’s the second largest element. Thus the total number of comparisons is \( n + \lceil \log n \rceil - 2 \). (15 points)

5. In our linear-time selection algorithm, the inputs are divided into groups of 5. What if you used groups of 3 instead? What if used groups of 7 or larger (odd integers)?

**Solution:** It does not work for groups of 3 because we get the following recursion:

\[
T(n) \leq T \left( \frac{n}{3} \right) + T \left( \frac{2n}{3} + 4 \right) + O(n).
\]

Indeed, if we try similarly to prove \( T(n) \leq cn \) with substitution, it doesn’t work for any constant \( c \):

\[
T(n) \leq \frac{cn}{3} + \frac{2cn}{3} + 4c + an = cn + 4c + an \not\leq cn
\]

However, it works for groups of 7 or larger with a slightly worse constant. Say we use groups of size \( 2k+1 \) with an arbitrary integer \( k \geq 3 \). Then we get the following recursion which has a linear solution for each fixed \( k \):

\[
T(n) \leq T \left( \frac{n}{2k+1} \right) + T \left( \frac{(3k+1)n}{2(2k+1)} + 2(k+1) \right) + O(n).
\]

Indeed, as for groups of 5 we can prove \( T(n) \leq cn \) by substitution if \( c \) is large enough compared to \( k \). We get

\[
T(n) \leq cn - \frac{c(k-1)n}{2(2k+1)} + 2c(k+1) + an \leq cn,
\]

if \( c \) is large enough compared to \( k \) and \( a \). (15 points)

6. Show that \( 2n - 1 \) comparisons are necessary in the worst case to merge two sorted lists containing \( n \) elements each.

**Solution:** When the sorted order perfectly interleaves the two lists, each element in the final order must have been compared to its neighbors, which both come from the other list. So consider the two sorted lists \( a_1 < \ldots < a_n \) and \( b_1 < \ldots < b_n \) such that

\[
a_1 < b_1 < a_2 < b_2 < \ldots < a_n < b_n.
\]
We claim that in order to correctly merge the two lists, we must make the following $2n - 1$ comparisons:

$$(a_1 : b_1), (b_1 : a_2), (a_2 : b_2), \ldots, (a_n : b_n).$$

Indeed, if for example $(b_1 : a_2)$ is not made, then the configuration

$$a_1 < a_2 < b_1 < b_2 < \ldots < a_n < b_n$$

is indistinguishable from (1), since all other results are the same, so our algorithm cannot be correct. (20 points)