1. (a) There are two steps.

    i. Perform a linear scan to find the index into the array that contains the (time, name) pair with the correct name, say $(*,\mathrm{N})$.
    Running time: $O(n)$

    ii. Bubble item $(*,\mathrm{N})$ up to the root by repeatedly interchanging with its parent node, and then delete the root node using the same recipe as for `deleteMin`.
    Running time: $O(\log n)$

    The overall running time is $O(n)$.

   (b) Same as part (a), except modify step i. to search based on the time. The running time is still $O(n)$.

   (c) Apply step ii. from part (a). The running time is now $O(\log n)$.

2.
```
(a) i := 0;
(b) while i < n do
(c)    if A[i]=i then
(d)       i := i+1
(e)    else
(f)       A[A[i]],A[i] := A[i],A[A[i]]
(g)    fi
(h) od
```

At the beginning of each loop iteration $A[0], A[1], \ldots, A[i-1] = 0, 1, \ldots, i-1$. Each iteration of the loop either increments i or increases by at least one the number of array elements that are in correct position. Thus, the number of iterations is between n and $2n$, giving a running time of $\Theta(n)$.

More details (not required for full marks): The only possible modifications made to A are to swap two elements. Thus, $A[0 \ldots n-1]$ remains a permutation of $0 \ldots n-1$ throughout execution.

The first time the loop iterates we have $i = 0$, so the loop invariant

$$A[0], A[1], \ldots, A[i-1] = 0, 1, \ldots, i-1 \tag{1}$$

holds trivially. Induction on the number of times the loop iterates shows that (1) holds throughout execution. On the one hand, if line (d) is executed, then $A[i] = i$ and upon incrementing i invariant (1) still holds. On the other hand, considering invariant (1), line (f) is executed only if $A[i] > i$ and thus $A[A[i]] \neq A[i]$. But then upon completion of line (f) we have $A[A[i]] = A[i]$, thus increasing by at least one the number of array elements in correct position.

3. There are three steps.

   i. Perform an in place heapify so that $A[0 \ldots n-1]$ is a min-heap.
   Running time: $O(n)$.

j43luo

This document is for the exclusive use of j43luo.

ii. Perform k `deleteMin` operations to obtain the k smallest integers in the array. Similar to heapsort, for $i = 1, 2, \ldots, k$, place the ith integer extracted from the heap into position $A[n-i]$ of the array. Now $A[n-1], A[n-2], \ldots, A[n-k]$ are the k smallest integers in increasing order.
Running time: $O(k \log n)$, which is $O(n)$ if $k \in O(n/\log n)$.

iii. Reverse the order of elements in the array by interchanging $A[i]$ and $A[n-i-1]$ for $i = 0, 1, \ldots, \lfloor (n-1)/2 \rfloor$.
Running time: $O(n)$.

4. (a) Each coin is genuine or counterfeit, but the two cases where all coins are genuine or all are counterfeit are excluded; the total number of possible answers is thus $2^n - 2$. Each weighing has exactly 3 possible outcomes, so if an algorithm performs at most k weighings, the number of different answers the algorithm could return is at most 3^k. So we must have $3^k \geq 2^n - 2$; solving for the integer k yields the lower bound of $\lceil \log_3(2^n - 2) \rceil$ weighings.

   (b) Using the algorithm from part (c), we need 3 weighings when $n = 4$.
   Notice that $\lceil \log_3(2^4 - 2) \rceil = \lceil \log_3(14) \rceil = \lceil 2.402173503 \rceil = 3$.

   (c) Perform exactly $n-1$ weighings between the pairs of coins C_1 and C_i for $2 \leq i \leq n$. C_1 is counterfeit if and only if all weighings determine that C_1 weighs at most as much as the any of the other coins, with C_1 weighing less than at least one other coin since there is at least one genuine coin; the weak players are those that weigh the same as C_1 and the genuine coins are those that weigh more than C_1.

   The case where C_1 is a genuine coin is similar.

   This algorithm is $\Theta(n)$. Observe that $\lceil \log_3(2^n - 2) \rceil > \log_3(2^{n-1})$ for $n \geq 2$, and this is equal to $\log_3(2) \cdot (n-1)$, so the lower bound from part (a) is $\Omega(n)$. Therefore is algorithm is asymptotically optimal in the number of contests.

5. (a) Choose i uniformly and randomly from the range $0, \ldots, n-1$ and return $A[i]$. This will be the dominating element with probability at least $p = (\lfloor n/2 \rfloor + 1)/n > 1/2$.

   (b) Use the algorithm from part a) to find an element x which will be the dominating element with probability at least p. Perform a linear scan of the array to assay if x occurs at least $\lfloor n/2 \rfloor + 1$ times; if so then report yes, and if no repeat by calling anew the algorithm from part a).

   The expected-case running time is $O(n)$ because each iteration has cost $O(n)$, and the expected number of iterations is bounded by $\sum_{i=1}^{\infty} i(1-p)^{i-1} p = 1/p < 2$.

   An alternative analysis would be to observe that, with probability at least 1/2, $T(n) \leq cn$ for some constant $c > 0$, and otherwise $T(n) \leq cn + T(n)$. So the expected cost is given by $T(n) \leq cn + \frac{1}{2}T(n)$, which solves to $T(n) \leq 2cn$.

This document is for the exclusive use of j43luo.