ALG Problem Set 3

Problem 1

(a) Maintain a set of candidate numbers C. Initially $C = \{1, 2, \dots, n\}$. For each question, we evenly divide C into two subsets C_1 and C_2 (if $|C|$ is odd, let $|C_1| = |C_2| + 1$). Ask the question "Is the number in C_1?". If the answer is "Yes", let $C = C_1$, otherwise let $C = C_2$. Repeat this process until $|C| = 1$. The remaining number in C is the answer. Since each question halves the size of C, we need at most $\lceil \log_2 n \rceil$ questions.

Note: This strategy is optimal because each question can be viewed as a binary partition of the current candidate set, and to minimize the worst-case number of questions, we should always split the set as evenly as possible.

(b) If n is unknown, it is hard to determine the optimal strategy. However, we can use an exponential search strategy to find an upper bound for the target integer and then apply the binary search strategy. We can ask questions in the following manner:

- For the i-th question ($i = 1, 2, ...$), ask "Is the number in $[2^{i-1}, 2^i - 1]$?" If "Yes", we can let $C \leftarrow [2^{i-1}, 2^i - 1]$ and proceed to binary search within this range. If "No", continue to the next i.

If the chosen integer is x, we will find the range $[2^{k-1}, 2^k - 1]$ such that $2^{k-1} \leq x < 2^k$ after k questions, where $k = \lfloor \log_2 x \rfloor + 1$. Then we can perform binary search within this range, which will take at most $\lceil \log_2(2^k - 2^{k-1}) \rceil = k - 1$ questions. Therefore, the total number of questions asked is at most $k + (k - 1) = 2k - 1 = 2\lfloor \log_2 x \rfloor + 1$.

Problem 2

(a) (b) The answer to (b) is yes. We can find the smallest integer in $[1..m]$ that is not present in A within $O(\log n)$ time using a binary search approach. The algorithm is as follows:

- Initialize two pointers: $low = 1$ and $high = n + 1$. By the pigeonhole principle, the smallest missing integer must be in the range $[1..n + 1]$.
- While $low < high$:
  - Let $mid = \lfloor (low + high)/2 \rfloor$.
  - If $a[mid] = mid$, then $a[1..mid] = [1..mid]$. The smallest missing integer must be in the right half. Set $low = mid + 1$.
  - Otherwise, the smallest missing integer is in the left half. Set $high = mid$
- The smallest missing integer is the value of low after the loop ends.

This is a standard binary search algorithm, which runs in $O(\log n)$ time.

Problem 3

The algorithm FindKth(A, B, k) takes two sorted arrays A, B, an integer $1 \leq k \leq |A| + |B|$, and returns the k-th smallest element in the array $A + B$. The runnning time is $O(\log k) = O(\log(|A| + |B|))$. For our question, we can call FindKth$(A, B, (|A| + |B|)/2)$ to get the median of $A + B$ in $O(\log(|A| + |B|))$ time.

// The initial call is FindKth$(A, B, n, m, (n+m)/2)$

FindKth(A, B, n, m, k)

```
1:  if k = 1 then
2:      return min(A[1], B[1])
3:  mid1 = ⌊k/2⌋
4:  mid2 = k − mid1
5:  a_mid = mid1 ≤ n ? A[mid1] : ∞
6:  b_mid = mid2 ≤ m ? B[mid2] : ∞
7:  if a_mid ≤ b_mid then
8:      return FindKth(A[mid1+1:][1], B, n-mid1, m, k-mid1)
9:  else
10:     return FindKth(A, B[mid2+1:], n, m-mid2, k-mid2)
```

[1]We just pass the starting index of the subarray instead of creating a new one.

For the correctness, we first prove when all elements in A and B are distinct. Assume the k-th smallest element is x:

- If $A[mid1] < B[mid2]$, then x must $> A[mid1]$. Otherwise, if $x \leq A[mid1]$, then there are at most $mid1$ elements in A and at most $mid2-1$ elements in B that are not greater than x, which means there are at most $mid1 + (mid2 - 1) = k - 1$ elements not greater than x. This contradicts the definition of x. Similarly, if $A[mid1] > B[mid2]$, then $x > B[mid2]$.
- Note that if we remove a number $y < x$, then x is the $(k-1)$-th smallest element in the new array. If $A[1..mid1]$ all less than x, we can remove them all. Similarly, if $B[1..mid2]$ all less than x, we can remove them all (corresponding to line 8 and 10 respectively).

What if there are duplicates? Let $A'[i] = (A[i], i)$ and $B'[i] = (B[i], n+i)$ be the augmented arrays. We can apply the same algorithm on A' and B'. Since A' and B' are distinct, we can find the k-th smallest element (x, j) in $A' + B'$. However, $A'[i] < B'[j] \iff A[i] \leq B[j]$. So the algorithm can be applied directly on A and B without any modification.

The running time $T(k)$ satisfies the recurrence: $T(k) = T(\lceil k/2 \rceil) + O(1)$

So $T(k) = O(\log k)$.

Problem 4

(a) Given player i, query the other $n-1$ players. Player i is citizen if and only if at least $\lfloor n/2 \rfloor$ of them say player i is citizen. The initial call is IsCitizen$([1, 2, \ldots, n], i)$.

IsCitizen$(A[1..n], i)$

```
1:  count = 0
2:  for j = 1 to n do
3:      if A[j] ≠ i then
4:          if player A[j] says player i is citizen then
5:              count = count + 1
6:  if count ≥ ⌊n/2⌋ then
7:      return true
8:  else
9:      return false
```

Since #citizens > #werewolves, #citizens $\geq \lfloor n/2 \rfloor + 1$. If player i is a citizen, there are at least $\lfloor n/2 \rfloor$ citizens among the other $n-1$ players, so at least $\lfloor n/2 \rfloor$ players will tell the truth and say player i is a citizen. If player i is a werewolf, there are at most $n - \lfloor n/2 \rfloor - 2 = \lceil n/2 \rceil - 2$ werewolves among the other $n-1$ players, so at most $\lceil n/2 \rceil - 2$ players will lie and say player i is a citizen. Since $\lfloor n/2 \rfloor \geq \lceil n/2 \rceil - 1 > \lceil n/2 \rceil - 2$, the algorithm is correct.

(b) We can develop an algorithm that guarranteed to return a citizen when the number of citizens is more than the number of werewolves.

FindCitizen($A[l..r]$)

```
1: if l = r then
2:     return A[l]
3: mid = ⌊(l + r)/2⌋
4: x = FindCitizen(A[l..mid])
5: if IsCitizen(A[l..r], x) then
6:     return x
7: else
8:     return FindCitizen(A[mid + 1..r])
```

The initial call is FindCitizen($[1, 2, \dots, n]$). Correctness: we induce on $r - l$.

Base case: If $l = r$, l is the only player. If there are more citizens than werewolves, $A[l]$ must be a citizen.

Induction step: Assume the algorithm works for all $r - l < k$. Now consider $r - l = k$. We divide it into two halves. Assume FindCitizen($A[l..mid]$) returns a x. If x is a citizen, the algorithm returns x and is correct. If x is a werewolf, then by the induction hypothesis, there left half has no citizens more than werewolves. So the right half must have more citizens than werewolves. The algorithm returns FindCitizen($A[mid + 1..r]$). By the induction hypothesis, it returns a citizen. The algorithm is correct.

Analysis for the running time $T(k), k = r - l + 1$: in the worst case, we call FindCitizen twice on size $k/2$ and call IsCitizen once. So $T(k) \leq 2T(k/2) + O(k)$. $T(n) = O(n \log n)$.

(c) **[bonus]** If the number of players is odd, we pick an player x and check its identity. If x is a citizen, we done. Otherwise, we remove x and the number of players is even. If the number of players is even, we pair up the players. For each pair, we query them about each other. We keep one person from the pairs saying each other are citizens and remove the others. Repeat until only one player remains or find a citizen.

// Initial call: FindCitizenLinear($[1, 2, \dots, n]$)

FindCitizenLinear($A[1..k]$)

```
1: if k is odd then
2:     x = A[k]
3:     if IsCitizen(A, x) then
4:         return x
5:     else
6:         A = A[1..k − 1]
7: for i = 1 to k/2 do
8:     Query A[2i − 1] and A[2i] about each other
9:     if both say the other is citizen then
10:        B.append(A[2i − 1])
11: return FindCitizenLinear(B)
```

Correctness: When $k = 1$, the algorithm always return $A[1]$ at line 4. When $k > 1$, assume the algorithm is correct for all $k' < k$. If k is odd:

- If $A[k]$ is a citizen, the algorithm returns x and is correct.
- If $A[k]$ is a werewolf, the number of citizens in $A[1..k-1]$ is more than the number of werewolves. By the induction hypothesis, the algorithm is correct.

If k is even. The table below shows all possible outcomes of a pair of players:

Player 1	Player 2	Both say citizens?	Keep
Citizen	Citizen	Always	Citizen
Citizen	Werewolf	Never	None
Werewolf	Citizen	Never	None
Werewolf	Werewolf	Maybe	Werewolf

Assume all the players are divided into 3 groups: k_1 pairs of two citizens, k_2 pairs of two werewolves, k_3 pairs of one citizen and one werewolf.

We have $2k_1 + k_3 > 2k_2 + k_3$. So $k_1 > k_2$. The number of citizens in B is k_1 and the number of werewolves is $\leq k_2$. There are more citizens than werewolves in B. By induction, the algorithm is correct.

The running time $T(k)$: we call FindCitizenLinear once on size at most $k/2$, query $k/2$ pairs of players and the IsCitizen(k) takes O(k) time. So $T(k) \leq T(k/2) + O(k)$. $T(n) = O(n)$.

Problem 5

(a) Assume the root is at level 1. The k-th ($2 \leq k \leq \lfloor n/2 \rfloor$) largest element might at any level from $\boxed{2}$ to $\boxed{\min\{k, \lfloor \lg n \rfloor + 1\}}$ inclusive.

(b) For min-heap, we can use Min-Heapify(A, i) and Heap-Decrease-Key(A, i, key) to implement HeapUpdate(A, i, val).

HeapUpdate(A, i, val)

```
1: if A[i] < val then
2:     A[i] = val
3:     Min-Heapify(A, i)
4: else if A[i] > val then
5:     Heap-Decrease-Key(A, i, val)
```

Heap-Decrease-Key(A, i, key)

```
1: if key > A[i] then
2:     error "new key is larger than current key"
3: A[i] = key
4: while i > 1 and A[Parent(i)] > A[i] do
5:     swap A[i] and A[Parent(i)]
6:     i = Parent(i)
```

Min-Heapify(A, i)

```
1: l = Left(i)
2: r = Right(i)
3: if l ≤ A.heap-size and A[l] < A[i] then
4:     smallest = l
5: else
6:     smallest = i
7: if r ≤ A.heap-size and A[r] < A[smallest] then
8:     smallest = r
9: if smallest ≠ i then
10:    swap A[i] and A[smallest]
11:    Min-Heapify(A, smallest)
```

If $A[i] < val$, we increase $A[i]$ to val and call Min-Heapify to maintain the min-heap property in the subtree rooted at i. For each node j, if

- j is not in the subtree rooted at i. $A[j]$ and $A[\text{Parent}(j)]$ are not changed, so $A[j] \geq A[\text{Parent}(j)]$ still holds.

- j is in the subtree rooted at i but $j \neq i$. PARENT(j) is also in the subtree. After MIN-HEAPIFY, $A[j] \geq A[\text{PARENT}(j)]$.
- $j = i$. Assume the elements in the subtree rooted at i before HEAPUPDATE are S. $A[i]$ is changed from $\min(S)$ to $\min\{S - \{\min(S)\} + \{val\}\}$, so it is not decreased. $A[i] \geq A[\text{PARENT}(i)]$ holds.

If $A[i] > val$, just call HEAP-DECREASE-KEY to maintain the min-heap property.

The MIN-HEAPIFY and HEAP-DECREASE-KEY have $O(\log n)$ runtime. So does HEAPUPDATE.