Q1 Suppose you have an unsorted array $A[1..n]$ of elements, and this array cannot be sorted. For example this can be array of JPEG images that you can only compare for equality/inequality. An element of this array is called dominant if it appears in the array more than half the time. For example in $[a, b, a, a]$ the dominant element is a, but arrays $[a, b, a, c]$ and $[a, a, b, c]$ have no dominant element. You want to find a dominant element in array A. Straightforward approach of checking if $A[i]$ is a dominant for all $i = 1, \ldots, n$ will have run-time in $\Theta(n^2)$ which is too slow. Consider the following approach to finding a dominant element: recursively find the dominant element y in the first half of the array and the dominant element z in the second half of the array, and combine results of recursive calls into the answer to the problem.

(a) show that if x is a dominant element in A then it has to be a dominant element in the first half of the array or the second half of the array (or both). [5 points]

We define a list A with an arbitrary length n such that $n \geq 1$ containing elements $w_1, w_2, \ldots, w_n$ where each w_k for $1 \leq k \leq n$ is a JPEG image. Since each w_k can be compared for uniqueness we define a function

$$freq(u, A) = \{ \text{`number of times } u \text{ appears in } A\text{'} \}$$

We see that if $freq(u, A) > \frac{|A|}{2}$ then u is the dominant element in A.

When $n \geq 2$ we can split A into two halves y and z where the length of y is defined as $|y| = \lfloor \frac{n}{2} \rfloor$ and the length of z as $|z| = \lceil \frac{n}{2} \rceil$. This says that y will contain the elements $w_1, w2, \ldots, w_{\lfloor \frac{n}{2} \rfloor}$ and z will contain the elements $w_{\lfloor \frac{n}{2} \rfloor + 1}, w_{\lfloor \frac{n}{2} \rfloor + 2}, \ldots w_n$.//

We claim: Given an arbitrary $x \in A$ if $freq(x, A) > \frac{|A|}{2}$ then one of the following conditions must be true:

1. $freq(x, y) > \frac{|y|}{2}$
2. $freq(x, z) > \frac{|z|}{2}$
3. $freq(x, y) > \frac{|y|}{2}$ and $freq(x, z) > \frac{|z|}{2}$

Proof:

Suppose we have a list A with dominant element $x \in A$ such that $freq(x, A) > \frac{|A|}{2}$. We divide A into two parts y and z where $|y| = \lfloor \frac{|A|}{2} \rfloor$ and $|z| = \lceil \frac{|A|}{2} \rceil$ and note that $|y| + |z| = |A|$. We assume x, while being the dominant element in A, is dominant in neither y nor z. Meaning that despite being the dominant element in A none of the three conditions in our claim are satisfied.

With this assumption it follows that $freq(x, A) > \frac{|A|}{2}$ but $freq(x, y) < \frac{|y|}{2}$ and $freq(x, z) < \frac{|z|}{2}$. This means that:

$$freq(x, y) + freq(x, z) < \frac{|y|}{2} + \frac{|z|}{2}$$

However, we know that $|A| = |y| + |z|$, and furthermore, it follows from our definition that $freq(x, y) + freq(x, z) = freq(x, A)$ so we simplify the above as,

$$freq(x, A) < \frac{|A|}{2}$$

This contradicts our assumption since we know $freq(x, A) > \frac{|A|}{2}$ must be true as x is the dominant element in A by definition. Therefore, as we have reached a contradiction when we assumed that none of our three defined conditions would hold it follows that if x is the dominant element of A then one of the following conditions must be true:

1. $freq(x, y) > \frac{|y|}{2}$ "x is the dominant element in the first half of A"
2. $freq(x, z) > \frac{|z|}{2}$ "x is the dominant element in the second half of A"
3. $freq(x, y) > \frac{|y|}{2}$ and $freq(x, z) > \frac{|z|}{2}$ "x is the dominant element in both halves of A"

(b) Using observation of part (a) give a divide-and-conquer algorithm to find a dominant element, that runs in time $O(n \log n)$. Detailed pseudocode is required. Be sure to argue correctness and analyze the run time. If given array has no dominant element, return $FAIL$. [5 points]

```
function Freq(x, A)
    count ← 0
    for img ∈ A do                                        ▷ O(n)
        if img = x then
            count ← count + 1
        end if
    end for
end function
```

In the above, $Freq(x, A)$ takes a JPEG image x and a list of JPEG images A then steps through the list one by one comparing x to each image and returning the number of times x appears in A. This function will have a worst case running time of $O(n)$

```
function Dominant(A)
    len ← length(A)
    if len = 1 then
        return A[1]
    end if
    mid ← ⌊len/2⌋
    y ← dominant(A[1 ... mid])
    z ← dominant(A[mid + 1 ... len])                      ▷ O(lg n)
    if y = z or z = None then
        return y
    else if y = None then
        return z
    end if
```

$y_{freq} \leftarrow freq(y, A)$ $\triangleright$ $O(n)$
$z_{freq} \leftarrow freq(z, A)$ $\triangleright$ $O(n)$
if $y_{freq} > mid$ **then**
    return y
else if $z_{freq} > mid$ **then**
    return z
else
    return $None$
end if
end function

In the above $Dominant(A)$ divides the list of JPEG images A into two relatively equal parts by finding the midpoint and recursively calling itself on each part of the list. This will continue until a base case is reached where the resulting list has one element. At this point as a 1 element list the only element will comprise more than 50% of the sub-list and is thus the dominant element in that list. As the list A is being split into two equal parts at each call of *Dominant* the total depth of the recursive tree will be $\lg n$.

At each call to dominant the $Freq$ function above is called. Since $Freq$ is called with half of the list at each recursion we know from Zeno's paradox that this will result in a total of n steps at each call to *Doiminant*. Knowing this we can solve the recurrence relation where,

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

We can apply the master theorem to the above (or solve the recurrence through substitution) and we will find:

$$T(n) = O(n \lg n)$$

Thus completing the complexity requirement. We need one more function to fulfill the algorithms specification. The above says the algorithm must return $FAIL$ when there is no dominant element in the list. Currently our solution will return $None$ as is required for unneeded branches of the recurrence. To make sure we return the required $FAIL$ we can wrap $Dominant(A)$ in a simple helper function, shown below.

function GetDominant(A)
    $dom \leftarrow dominant(A)$ $\triangleright$ $O(n \lg n)$
    if $dom = None$ **then**
        return $FAIL$
    else
        return dom
    end if
end function

While the above function does not add to the complexity of the algorithm it is important to meet the specifications laid out above.

Q2 An array A of n distinct integers $A_1, A_2, \ldots, A_n$ is known to have the following property: Elements follow in descending order up to a certain index p where $1 < p < n$ and then follow in ascending order:

$$A_1 > A_2 > \cdots > A_{p-1} > A_p < A_{p+1} < \cdots < A_n \tag{1}$$

Give an efficient algorithm (analogue of non-recursive binary search) to find the index of smallest value in this array (i.e., to find p). Note: the worst case running time of your algorithm must be in o(n), so simple scanning from left to right is not going to work. Detailed pseudocode is required. [5 points]

Require: $A = A_1 > A_2 > \cdots > A_{p-1} > A_p < A_{p+1} < \cdots < A_n$
Ensure: $FindMin(A) = A_p$

```
function FindMin(A)
    len ← length(A)
    if len = 1 then
        return A[1]
    end if
    mid ← ⌊len/2⌋
    midNext ← mid + 1
    if A[mid] > A[midNext] then
        return FindMin(A[midNext ... len])
    else
        return FindMin(A[1 ... mid])
    end if
end function
```

In the above algorithm, $FindMin$ finds the mid point of the list A, then the list is reduced by half depending on if the next element of the after the midpoint is greater than it. The reduced list is then recursed on. At each step the work done inside $FindMin$ is completed in a constant $O(1)$ number of steps so we have the recurrence relation of,

$$T(n) = 2T(\frac{n}{2}) + O(1)$$

Solving this with the master theorem or by substitution yields a complexity of $O(\lg n)$ which satisfies the constraint of a complexity of $o(n)$.

Q3 A singly linked list contains $n-1$ strings that are binary representations of numbers from the set $\{0, 1, \ldots, n-1\}$ where n is an exact power of 2. However, the string corresponding to one of the numbers is missing. For example, if $n = 4$, the list will contain any three strings from 00, 01, 10 and 11. Note that the strings in the list may not appear in any specific order. Also note that the length of each string is $\lg n$, hence the time to compare two strings in $O(\lg n)$. Write an algorithm that generates the missing string in $O(n)$. [5 points]

Require: L is a singly linked list as defined above.
Ensure: $FindMissing(L)$ = a binary string not found in L with $\lg n$ digits

function FINDMISSING(L)
  $digits \leftarrow$ length($curr.data$)
  $num[] \leftarrow$ zeros($digits$)
  $missingNum \leftarrow buildMissing(L, digits, num)$ ▷ $O(n)$
  $missingString \leftarrow$ "X"^{digits}
  for $i \leftarrow 1$ to $digits$ **do** ▷ $O(\lg n)$
    $missingString[i] \leftarrow$ string($missingNum[i]$)
  end for
  return $missingString$
end function

The above function is a wrapper which calls the recursive function $BuildMissing$ then converts the decimal array representation of the missing binary number to a string in $O(\lg n)$ time. As we will see below $BuildMissing$ has a complexity of $O(n)$ so when calling $FindMissing(L)$ The total running time will be $O(n + \lg n)$, as n grows, the rate of growth will eclipse the $\lg n$ portion of the formula and the total running time will be $O(n)$ satisfying the requirements for the question.

Require: L a Linked list, bit: the current bit to check, num: missing number being built
Ensure: $bit > 0$

function BUILDMISSING($L, bit, num[]$)
  if $bit = 0$ **then**
    return num
  end if
  $bitSet \leftarrow$ new$LinkedList$
  $bitSetLen \leftarrow 0$
  $bitUnset \leftarrow$ new$LinkedList$
  $bitUnsetLen \leftarrow 0$
  $curr \leftarrow L.head$
  while $curr \neq None$ **do**
    if $curr.data[bit] =$ "1" **then**
      $bitSet.push(curr.data)$
      $bitSetLen \leftarrow bitSetLen + 1$
    else
      $bitUnset.push(curr.data)$

```
            bitUnsetLen ← bitUnsetLen + 1
        end if
        curr ← curr.next
    end while
    if bitSetLen < bitUnsetLen then
        num[bit] ← 1
        return BuildMissing(bitUnset, bit − 1, num)
    else
        num[bit] ← 0
        return BuildMissing(bitSet, bit − 1, num)
    end if
end function
```

The above algorithm steps through a linked list L and counts the number of ones and zeros at position *bit*. Since the missing number will cause either the count of ones or zeros to be unbalanced we append the binary numbers with either a 1 or a 0 at position *bit* to separate lists and discard the list which is longer. This has the effect of reducing the search space by half at each call to $BuildMissing$.

On top of discarding the longer list, thereby reducing our search space, we also set the *bit* position of the missing number to either a one or a zero based on which list was shorter when we built them above. We pass the shorter list as A and the missing number we are building as *num* then decrease the *bit* position for the next recursion. Once the *bit* position reaches 0 we have reached our base-case and know we have finished building *num*, our missing number, and can return it to our wrapper function.

Each call to $BuildMissing$ requires $\frac{n}{2^k}$ steps. The total steps can be described by,

$$\left(\sum_{k=1}^{\infty} \frac{1}{2^k}\right) n = (1)n = n$$

This is similar to the $BuildMaxHeap$ algorithm described in lecture notes.

The correctness of this algorithm is maintained despite halving the search space at each iteration. This is clear since the missing number will be confined to the list of strings which have one less bit set or unset. If we did not discard the unneeded portion of the list it would be impossible to achieve a complexity lower than $O(n \lg n)$, However, by using a divide and conquer approach we can solve this problem in linear time with a complexity equal to $O(n)$ using the above algorithm.