Problem 1 (Find the Rotation Point) *You have a rotated array, which means that it is "sorted" but the starting point is actually in the middle of the array. For example, the following array is rotated:*

$$[10, 14, 43, 61, 63, 87, 2, 6, 8, 9].$$

We would say this is rotated 6 spots, because the smallest entry is entry $A[6]$ and it increases from there. The rotation point would be $A[6]$.

(a) Design an $O(\log n)$ algorithm to find the index of the rotation point. Prove that it is correct, and that it runs in that time.

(b) (Bonus) Design an algorithm that, given some integer x, determines whether x is in the array. It should have the same runtime.

Solution:

(a) **Reasoning:** We'll follow the template of Binary Search here: we want to narrow our search down to half the array, so we'll want to figure out which half of the array the rotation point is on. To do so, we notice that, if the rotation point is **not** in a half, then that half-array must only be increasing. On the other hand, if it is in a half, then the array must decrease at some point. So, we'll just look at the left half of the array. If the left side of that half is smaller than the right side, the rotation point can not be in that half. On the other hand, if the left side of the half is larger than the right side, we must have rotated somewhere in that half.

Pseudocode

Algorithm 1 Rotation Point Searching

```
1:  procedure FindRotation(A)
2:      if length(A)=1 then
3:          return 0
4:      end if
5:      midpoint ← length(A)//2
6:      if A[midpoint] > A[midpoint + 1] then
7:          return midpoint+1
8:      else if A[midpoint] > A[0] then          ▷ A is increasing between 0 and midpoint
9:          return FindRotation(A[midpoint:n]) + midpoint
10:     else
11:         return FindRotation(A[0:midpoint])
12:     end if
13: end procedure
```

Proof of Correctness We must show that this algorithm returns the correct rotation point index. We will proceed by induction on the length of A.

Base Case: When A has length 1, then the rotation point is the only point, and the algorithm returns 0.

Inductive Hypothesis: When A is a rotated array with length less than or equal to n, then the algorithm returns the correct rotation point.

Inductive Step: Take an array A of length $n + 1$. There are three cases: the rotation point is at index midpoint+1, the rotation point is before that index, or the rotation point is after that index.

If it is at midpoint+1, then A[midpoint] $>$ A[midpoint], so according to lines 6-7 this algorithm returns the correct value.

If it is before that index, then A[midpoint] $<$ A[0] because A must be increasing from the midpoint through to 0. Also, A[midpoint] $<$ A[midpoint]. Therefore, the algorithm returns FindRotation(A[0:midpoint]). By the inductive hypothesis, this recursive call is on a rotated array of size $n//2$, so it returns the correct index.

If it is after that index, then A[midpoint] $>$ A[0] because A must be increasing from 0 through to the midpoint. Also, A[midpoint] $<$ A[midpoint]. Therefore, the algorithm returns FindRotation(A[midpoint:n]) + midpoint. By the inductive hypothesis, this recursive call is on a rotated array of size $n//2$, so it returns the correct index in the array A[midpoint:n]. Any index i in that array is equal to the index midpoint+i in the original array, so the algorithm returns the correct value.

Proof of Runtime 1 (Direct): Note that, at each recursive call, the algorithm takes as input an array of half size. Therefore, because there are n entries in the array, this recursive call can only happen at most $\log n$ times. Within each recursive call there is a constant amount of work (just comparisons), so the final runtime is $O(\log n)$.

Proof of Runtime 2 (Master Theorem): Each call of this function recursively calls at most one subproblem. Thus, the runtime follows the recurrence

$$T(n) = T(n/2) + O(1).$$

By Master Theorem, we have $a = 1, b = 2, d = 0$, giving us a runtime $T(n) = O(\log n)$.

(b) The main idea here is that the most reliable source of information is the region without the rotation point, so we should determine which region that is and then use it.

Algorithm 2 Finding x in a Rotated Array

```
1:  procedure FindinRotatedArray(A, x)
2:      if length(A)=1 then
3:          return 0
4:      end if
5:      midpoint ← length(A)//2
6:      if A[midpoint] = x then
7:          return midpoint
8:      else if A[midpoint] > A[0] then          ▷ Rotation point is on the right, so left is increasing.
9:          if A[0] ≤ x ≤ A[midpoint] then       ▷ Check if x is on left side.
10:             return FindinRotatedArray(A[0:midpoint])
11:         else
12:             return FindinRotatedArray(A[midpoint:n]) + midpoint
13:         end if
14:     else                                     ▷ Rotation point is on the left.
15:         if A[midpoint] ≤ x ≤ A[n] then       ▷ Check if x is on right side.
16:             return FindinRotatedArray(A[midpoint:n]) + midpoint
17:         else
18:             return FindinRotatedArray(A[0:midpoint])
19:         end if
20:     end if
21: end procedure
```

We note that, in the end, this is still a constant number of comparisons that lead to a single recursive call, so the runtime is the same.

PROBLEM 2 (FIND THE FIXED POINT) *A fixed point in an array of integers A is an index i such that $A[i] = i$.*

(a) Array A has length n, and is sorted in increasing order ($A[0] < A[1] < A[2] < \ldots < A[n]$). Design a divide and conquer algorithm that either finds a fixed point or determines that one doesn't exist.

(b) Array B is unsorted. One thing you might think to do is simply sort the array and run the algorithm above. Is that a good idea? Why or why not?

(c) (Bonus) Array C is sorted in increasing order, and only contains positive integers. Design an algorithm that either finds a fixed point or determines that one doesn't exist. This algorithm should be asymptotically faster than your algorithm in part (a).

Solution:

(a) Here, the thing to notice is that, if $A[i] > i$, it is impossible for any index greater than i to have a fixed point. Similarly, if $A[i] < i$, it is impossible for any index less than i to have a fixed point. (Make sure you understand why!) This insight helps us narrow down our array by half at each recursive step.

Algorithm 3 Fixed Point Searching

```
1:  procedure FindFixedPoint(A)
2:      midpoint ← length(A)//2
3:      if A[midpoint] = midpoint then
4:          return midpoint
5:      else if A[midpoint] > midpoint then ▷ Then everything past midpoint can't be a fixed point either.
6:          return FindFixedPoint(A[0:midpoint])
7:      else
8:          return FindFixedPoint(A[midpoint:n]) + midpoint
9:      end if
10: end procedure
```

(b) Sorting the array is going to take at least $O(n \log n)$ time, while just iterating through the array only takes $O(n)$ time. We're better off just brute-forcing here, rather than sorting.

(c) Note that the smallest possible value of $A[0]$ is 1, so the smallest possible value of $A[1] = 2$, and so on. In general, $A[i] \geq i + 1$. Thus, it is impossible for there to be a fixed point, so the algorithm can just return False.

PROBLEM 3 (BACK TO DAY 1!) *As mentioned, two of the problems from Day 1 are divide-and-conquer problems. Go back to the slides from Day 1, and try to identify them.*

(a) One of the problems can be solved using the approach we discussed today. Figure out which one it is, and write an algorithm to solve it. Prove that it is correct, and analyze its runtime.

(b) (Bonus) Another problem on the list can also be solved using divide-and-conquer, but it's a bit more tricky. We'll discuss that problem Thursday. Take a guess on which problem it is! Why do you think it's that problem?

Solution:

(a) The problem we are interested in is **Peak-Finding**. If a day is at the end of the array and is larger than the one number it's adjacent to, we'll still call that a peak.

The main thing to notice is that, if any entry $A[i] < A[i+1]$, then there must be a peak on the right side of this entry. After all, any point where it stops increasing is a peak, and if it never stops increasing then the end of the array is a peak.

Algorithm 4 Peak Finding

```
1:  procedure FindPeak(A)
2:      if length(A)=1 then
3:          return 0
4:      end if
5:      midpoint ← length(A)//2
6:      if A[midpoint] < A[midpoint + 1] then
7:          return FindPeak(A[midpoint:n]) + midpoint
8:      else
9:          return FindPeak(A[0:midpoint])
10:     end if
11: end procedure
```

Note: We've done something a bit crazy here by not even checking whether the point is actually a peak, and recursing all the way down until we find an array of length 1. A more natural implementation would check at each recursive step whether midpoint is a peak, but we will show below that this works fine.

Proof of Correctness: We need to show that, for any array A, this algorithm returns the index of a peak. We will do so by inducting on the length of A.

Base Case: When A is length 1, the entry is vacuously a peak, so returning 0 is correct.

Inductive Hypothesis: For any array of length less than or equal to n, the algorithm returns the index of a peak.

Inductive Step: Assume that A[midpoint] < A[midpoint + 1]. Then the right half of the array must have a peak: either the array's values increase the whole way through, in which case the final value is a peak, or the array's values stop increasing at some point after midpoint+1, in which case that point before the decrease is a peak. Therefore, when we call FindPeak(A[midpoint:n]), we are calling it on an array of length $n//2$ that must contain a peak, so by the IH this returns the index of the peak in this subarray. That peak has index midpoint+FindPeak(A[midpoint:n]) in the original array.

Otherwise, A[midpoint] > A[midpoint + 1]. By a symmetric argument, the left side of the array has a peak, so by the IH, FindPeak(A[0:midpoint]) will return its index.

Runtime: Note that we are doing a constant amount of computation and then recursing on half the array, so our runtime follows the recurrence $T(n) = T(n/2) + O(1)$. By Master Theorem, this is $T(n) = O(\log n)$.

(b) It's **Closest Shops**!

PROBLEM 4 (FIND THE MAJORITY) *You're helping count votes for an election. You only care if a candidate has more than half of the votes – otherwise everyone will have to vote again anyway. This is the algorithm you're going to run:*

Algorithm 5 Majority-Finder – only returns if a candidate got the majority.

```
1: tracker ← an empty array                          ▷ Stores vote counts as (candidate, count)
2: for each vote v do
3:     if v's candidate is not in the tracker then
4:         Add (candidate, 0) to tracker.
5:     end if
6:     Add 1 to the candidate's count.
7: end for
8: if some count is greater than n/2 then
9:     return that candidate
10: end if
11: return None
```

(a) This algorithm takes $O(n^2)$ time to run. Why?

(b) Design a divide-and-conquer algorithm that works in $O(n \log n)$ time.

(c) (Bonus) Describe how, using a different data structure for tracker, we can bring this runtime down to $O(n)$.

*(d) (Bonus) The algorithm above has what is called **space complexity** of $O(n)$ – it uses up to $O(n)$ space to solve the problem. After all, tracker can get up to size n. Write an algorithm that solves the problem in $O(n)$ time and $O(1)$ **space**.*

Solution:

(a) Checking whether v's candidate is in the tracker can take up to $O(n)$ time. This can happen up to n times, once for each vote, so this is $O(n^2)$ in total.

(b) The key insight here is to note that a candidate that has a majority must also have a majority in at least one of the halves. After all, if it's not a majority in either half, then it can't add up to more than half in the whole array. (Pseudocode on next page)

(c) If tracker was instead a large enough hashmap, then we could simply do a direct iteration over the votes, counting through the hashmap. This would take $O(n)$ time, assuming placing into the hashmap takes $O(1)$ time.

(d) Left as an exercise. Hint: if you only have $O(1)$ space, you really can't store much. What's the first thing you'd think to store, and is that alone enough?

Algorithm 6 Majority-Finder – only returns if a candidate got the majority.

```
1:  procedure FindMajority(A)
2:      if length(A)=1 then
3:          return A[0]
4:      end if
5:      midpoint = length(A)//2
6:      c1 ← FindMajority(A[0:midpoint])
7:      c2 ← FindMajority(A[midpoint:n])
8:      if c1 = c2 then
9:          return c1
10:     end if
11:     c1counter, c2counter = 0, 0
12:     for each vote v do
13:         Increment c1counter or c2counter if v is c1 or c2, respectively
14:     end for
15:     if c1counter > midpoint then
16:         return c1
17:     else if c2counter > midpoint then
18:         return c2
19:     else
20:         return None
21:     end if
22: end procedure
```