1. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$.

   (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

Solution (binary search): Suppose we define a second array $\Delta[1..n]$ by setting $\Delta[i] = A[i] - i$ for all i. For every index i we have

$$\Delta[i] = A[i] - i \le (A[i+1] - 1) - i = A[i+1] - (i+1) = \Delta[i+1],$$

so this new array is sorted in increasing order. Clearly, $A[i] = i$ if and only if $\Delta[i] = 0$. So we can find an index i such that $A[i] = i$ by performing a binary search in the array Δ. But we don't actually need to explicitly compute Δ; instead, whenever the binary search needs to access some value $\Delta[i]$, we can compute $A[i] - i$ on the fly!

Here are two formulations of the resulting algorithm, the first recursive (passing the array *A by reference* with the top-level call FindMatch$(A, 1, n)$) and the second iterative.

```
⟨⟨Return any index i such that lo ≤ i ≤ hi and A[i] = i⟩⟩
FindMatch(A, lo, hi):
  if lo > r
      return None
  mid ← (lo + hi)/2
  if A[mid] = mid                          ⟨⟨Δ[mid] = 0⟩⟩
      return mid
  else if A[mid] < mid                     ⟨⟨Δ[mid] < 0⟩⟩
      return FindMatch(A, mid + 1, hi)
  else                                     ⟨⟨Δ[mid] > 0⟩⟩
      return FindMatch(A, lo, mid − 1)
```

```
FindMatch(A[1..n]):
  lo ← 1
  hi ← n
  while lo ≤ hi
      mid ← (lo + hi)/2
      if A[mid] = mid                      ⟨⟨Δ[mid] = 0⟩⟩
          return mid
      else if A[mid] < mid                 ⟨⟨Δ[mid] < 0⟩⟩
          lo ← mid + 1
      else                                 ⟨⟨Δ[mid] > 0⟩⟩
          hi ← mid − 1
  return None
```

In both formulations, the algorithm ***is*** binary search, so it runs in $O(\log n)$ time. ■

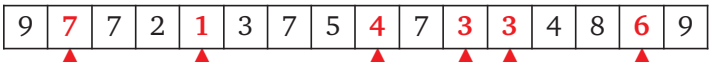
(b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

Solution: The following algorithm solves this problem in $O(1)$ time:

```
FindMatchPos(A[1..n]):
  if A[1] = 1
      return 1
  else
      return None
```

Again, the array $\Delta[1..n]$ defined by setting $\Delta[i] = A[i] - i$ is sorted in increasing order. It follows that if $A[1] > 1$ (that is, $\Delta[1] > 0$), then $A[i] > i$ (that is, $\Delta[i] > 0$) for every index i. $A[1]$ cannot be less than 1. ■

2. Suppose we are given an array $A[1..n]$ such that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a ***local minimum*** if both $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are exactly six local minima in the following array:



Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 9, because $A[9]$ is a local minimum. *[Hint: Any array with the stated boundary conditions* ***must*** *contain at least one local minimum. Why?]*

Solution (binary search): The following algorithm solves this problem in $O(\log n)$ time, Again, we are passing the array A by reference; the top-level function call is LocalMin$(A, 1, n)$.

```
⟨⟨Find a local minimum in A[lo..hi]⟩⟩
⟨⟨Assumes A[lo] > A[lo+1] and A[hi] > A[hi-1]⟩⟩
LocalMin(A, lo, hi):
  if hi - lo < 100
      find minimum of A[lo..hi] by brute force
  mid ← ⌊(hi + lo)/2⌋
  if A[mid] < A[mid + 1]
      return LocalMin(A, lo, mid + 1)
  else
      return LocalMin(A, mid, hi)
```

If $hi - lo < 100$, then a brute-force search finds the minimum of $A[lo..hi]$ in $O(1)$ time. (There's nothing special about 100 here; I think any integer larger than 2 will work. On the other hand, optimizing that constant doesn't improve the $O(\,)$ running time, so why bother?)

Otherwise, if $A[mid] < A[mid+1]$, the subarray $A[lo..mid+1]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray.

Finally, if $A[mid] \geq A[mid+1]$, the subarray $A[mid..hi]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray.

The running time satisfies the recurrence $T(n) \leq T(\lceil n/2 \rceil + 1) + O(1)$. Except for the $+1$ and the ceiling in the recursive argument, which we can ignore, this is the binary search recurrence, whose solution is $T(n) = O(\log n)$.

Alternatively, we can observe that $\lceil n/2 \rceil + 1 < 2n/3$ when $n \geq 100$, and therefore $T(n) \leq T(2n/3) + O(1)$, which implies $T(n) = O(\log_{3/2} n) = O(\log n)$. ■

3. Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

your algorithm should return the integer 9. *[Hint: What can you learn by comparing one element of A with one element of B?]*

Solution (binary search): The following algorithm solves this problem in $O(\log n)$ time:

```
MEDIAN(A[1..n], B[1..n]):
  if n < 10^100
      use brute force
  else
      m ← ⌈n/2⌉
      if A[m] > B[m]
          return MEDIAN(A[1..m], B[n−m+1..n])
      else
          return MEDIAN(A[n−m+1..n], B[1..m])
```

There are three cases to consider.

- If $n < 10^{100}$, then brute force works. (There's obviously nothing special about the constant 10^{100}, but I can't be bothered to optimize it.)
- Suppose $A[m] > B[m]$. (The following analysis is carefully written to handle both even and odd n.)

  $A[m]$ is larger than all $2m-1 \geq n-1$ elements in $A[1..m] \cup B[1..m-1]$, and therefore greater than or equal to the median of $A \cup B$. Thus, all $n-m$ elements in the subarray $A[m+1..n]$ are larger than the median and can be safely discarded.

  Similarly, $B[n-m] \leq B[m]$ is smaller than all $2n-2m+1 \geq n$ elements in $A[m..n] \cup B[m+1..n]$, and therefore less than or equal to the median of $A \cup B$. Thus, all $n-m$ elements in the subarray $B[1..n-m]$ are smaller than the median and can be safely discarded.

  We discard the same number of elements above and below the median, so the median of the remaining elements is the median of the original $A \cup B$.
- The remaining case $A[m] < B[m]$ is symmetric.

The running time satisfies the binary-search recurrence $T(n) = O(1) + T(n/2)$. We conclude that the algorithm runs in ***O(log n) time***. ■

Harder problem to think about later:

4. Now suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ with distinct elements and an integer k. Describe a fast algorithm to find the kth smallest element in the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 7, 17, 19] \qquad k = 6$$

your algorithm should return the integer 7.

Solution: The following algorithm solves this problem in $O(\log \min\{k, m+n-k\}) = O(\log(m+n))$ time:

```
Select(A[1..m], B[1..n], k):
  if k < (m+n)/2
      return LowerMedian(A[1..k], B[1..k])
  else
      return UpperMedian(A[k-n..m], B[k-m..n])
```

This algorithm relies on two subroutines, both minor modifications of the algorithm from problem 3.

- LowerMedian finds the ℓth smallest element of the union of two arrays of some equal length ℓ, using the algorithm from problem 3, with one simple but crucial tweak. If LowerMedian ever tries to access an entry $A[i]$ with $i > m$ or $B[i]$ with $i > n$, it uses the value ∞ instead of instead of raising an array-index exception, crashing with a segmentation fault, or reading garbage. This tweak is necessary when either $k > m$ or $k > n$.

  The kth smallest element of $A \cup B$ must lie in either $A[1..k]$ or $B[1..k]$, and in fact must be the kth smallest element of $A[1..k] \cup B[1..k]$. Our call to LowerMedian finds this element.

- Similarly, UpperMedian finds the ℓth *largest* element of two sorted arrays of some equal length ℓ. If UpperMedian wants an entry $A[i]$ or $B[i]$ with $i \le 0$, it uses the value $-\infty$ instead of instead of crashing. This tweak is necessary when either $k \le n$ or $k \le n$. Otherwise, the algorithm is identical to the algorithm in problem 3 (except for changes in the $O(1)$-time brute force base case).

  The kth smallest element of $A \cup B$ is also the $(m+n-k+1)$th *largest* element of $A \cup B$. Thus, it must be either one of the $m+n-k+1$ largest elements of A or one of the $m+n-k+1$ largest elements of B; these are the subarrays $A[k-n..m]$ and $B[k-m..n]$. Moreover, the $(m+n-k+1)$th largest element of $A \cup B$ is also the $(m+n-k+1)$th largest element of $A[k-n..m] \cup B[k-m..n]$. Our call to UpperMedian finds this element.

Our Select algorithm chooses whichever of these two options requires the smaller subarrays. But in fact, we could use either option, regardless of the value of k, and the algorithm would still be correct and would still run in $O(\log(m+n))$ time. ■