Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster.


1.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:
Suppose we define a second array $B[1..n]$ by setting $B[i] = A[i] - i$ for all $i$. For every index $i$ we have


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

Here are two formulations of the resulting algorithm, first recursive (keeping the array $A$ as a global variable), and second iterative.

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

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

In both formulations, the algorithm is binary search, so it runs in $O(\log n)$ time.
1.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:

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


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:

```
9 7 7 2 1 3 7 5 4 7 3 3 4 8 6 9
```

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: With the given boundary conditions, any array must contain at least one local minimum. Why?)

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

```python
LocalMin(A[1..n]):
    if n < 100
        find the smallest element in A by brute force
        $m \leftarrow \lfloor n/2 \rfloor$
        if $A[m] < A[m + 1]$
            return LocalMin(A[1..m + 1])
        else
            return LocalMin(A[m..n])
```

If $n$ is less than 100, then a brute-force search runs in $O(1)$ time. There’s nothing special about 100 here; any other constant will do.

Otherwise, if $A[n/2] < A[n/2 + 1]$, the subarray $A[1..n/2 + 1]$ satisfies the precise boundary conditions of the original problem, so the recursion fairy will find local minimum inside that subarray.

Finally, if $A[n/2] > A[n/2 + 1]$, the subarray $A[n/2..n]$ 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 \( n \)th smallest element) of the union \( A \cup B \). For example, given the input

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

your algorithm should return the integer 9. (Hint: What can you learn by comparing one element of \( A \) with one element of \( B \)?)

Solution:

The following algorithm solves this problem in \( O(\log n) \) time:

\[
\text{Median}(A[1..n], B[1..n]) :
\]

\[
\begin{align*}
\text{if } n < 10^{100} & \quad \text{use brute force} \\
\text{else if } A[n/2] > B[n/2] & \quad \text{return } \text{Median}(A[1..n/2], B[n/2+1..n]) \\
\text{else} & \quad \text{return } \text{Median}(A[n/2+1..n], B[1..n/2])
\end{align*}
\]

Suppose \( A[n/2] > B[n/2] \). Then \( A[n/2+1] \) is larger than all \( n \) elements in \( A[1..n/2] \cup B[1..n/2] \), and therefore larger than the median of \( A \cup B \), so we can discard the upper half of \( A \). Similarly, \( B[n/2-1] \) is smaller than all \( n+1 \) elements of \( A[n/2..n] \cup B[n/2+1..n] \), and therefore smaller than the median of \( A \cup B \), so we can discard the lower half of \( B \). Because we discard the same number of elements from each array, the median of the remaining subarrays is the median of the original \( A \cup B \).

To think about later:

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

\[
A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \quad B[1..5] = [2, 4, 5, 8, 17, 19] \quad k = 6
\]

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:

\[
\text{Select}(A[1..m], B[1..n], k) :
\]

\[
\begin{align*}
\text{if } k < (m + n)/2 & \quad \text{return } \text{Median}(A[1..k], B[1..k]) \\
\text{else} & \quad \text{return } \text{Median}(A[k - n..m], B[k - m..n])
\end{align*}
\]

Here, \( \text{Median} \) is the algorithm from problem 3 with one minor tweak. If \( \text{Median} \) wants an entry in either \( A \) or \( B \) that is outside the bounds of the original arrays, it uses the value \(-\infty\) if the index is too low, or \( \infty \) if the index is too high, instead of creating a core dump.