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.


   (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

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

   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 $\Delta$. But we don’t actually need to explicitly compute $\Delta$; instead, whenever the binary search needs to access some value $\Delta[i]$, we can just compute $A[i] - i$ on the fly!

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

   ![Algorithm](image1.png)

   ![Algorithm](image2.png)

   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. ■

For example, there are exactly six local minima in the following array:

\[
\begin{array}{cccccccccccc}
9 & 7 & 7 & 2 & 1 & 3 & 7 & 5 & 4 & 7 & 3 & 3 & 4 & 8 & 6 & 9
\end{array}
\]

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:

```python
def LocalMin(A[1..n]):
    if n < 100
        find the smallest element in A by brute force
    m ← ⌊n/2⌋
    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; I think any integer larger than 2 will work. On the other hand, optimizing that constant doesn’t actually make the algorithm faster, so why bother?)

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 $A[n/2] \geq A[n/2 + 1]$, 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 $T(n) = O(\log_{8/3} n) = O(\log n)$. ■
3. Suppose you are given two sorted arrays $A[1..n]$ and $\Delta[1..n]$ 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

$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20]$ \hspace{1em} $\Delta[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$

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:

```python
def Median(A[1..n], B[1..n]):
    if n < 1000
        use brute force
    else if A[n/2] > B[n/2]
        return Median(A[1..n/2], B[n/2+1..n])
    else
        return Median(A[n/2+1..n], B[1..n/2])
```

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 $A[n/2] > B[n/2]$. 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 $A \cup B$.

- The remaining case $A[n/2] < B[n/2]$ 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 $B[1..n]$ 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

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

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:

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

Here, Median is the algorithm from problem 3 with one minor tweak. If Median wants an entry in either $A$ or $B$ that is outside the bounds of the original input arrays, it uses the value $-\infty$ if the index is too low, or $\infty$ if the index is too high, instead of raising an array-index error or a segmentation fault, or reading garbage. (This tweak is necessary when either $m \leq k < n$ or $n \leq k < m$.)