1. (a) Output the minimum of the first $n-k+1$ elements. Finding the minimum of these elements takes $n-k$ comparisons.

   (b) If you said "Any algorithm must examine at least $n-k$ elements because, if it didn't, the ones it doesn't examine could be the $k$ smallest" that's o.k. but it doesn't bound the number of comparisons that the algorithm makes. (One comparison "examines" two elements.)

   To bound the number of comparisons: If an algorithm makes fewer than $n-k$ comparisons then the graph of comparisons (where vertices are elements and $(i,j)$ is an edge if and only if the algorithm compared the $i$th element to the $j$th element) would have more than $k$ connected components. Whatever element the algorithm outputs, we can make all the elements in its component larger than the elements in the other components, without changing the outcome of an comparison made by the algorithm, and thus without changing its output. There are at least $k$ elements in the other components since there are at least $k$ other components. Thus the output element is not one of the $k$ smallest.

2. **Claim:** *Any algorithm that solves the two-consecutive-zeros problem must perform, for some worst case input string, at least $n-1$ operations if $n = 1 \pmod 3$ and at least $n$ operations otherwise.*

   *Proof.* If $n = 0, 1$, the claim is trivially true. If $n = 2$, the adversary answers '0' to the algorithm's first query and thus forces any correct algorithm to query the other bit.

   Assume that the claim is true for strings of length at most $n-1$. Given a string of length $n > 2$, let $i$ be the position of the algorithm's first query.

   If both $i-1 \neq 1 \pmod 3$ and $n-i \neq 1 \pmod 3$ then the adversary answers '1' for position $i$, and considers positions $1 \ldots i-1$ and $i+1 \ldots n$ as two separate subproblems. By induction, any correct algorithm must query every position in both subproblems in the worst case, since $i-1 \neq 1 \pmod 3$ and $n-i \neq 1 \pmod 3$. (Here, we assume that the adversary never reveals two consecutive zeros in either subproblem.) Thus the claim holds in this case.

   If either $i-1 = 1 \pmod 3$ or $n-i = 1 \pmod 3$ then the adversary answers '0' for position $i$ and sets positions $i-1$ and $i+1$ to '1'. This means that if, in the future, the algorithm queries position $i-1$ or $i+1$ the adversary answers '1'. Notice that the algorithm must query both position $i-1$ and $i+1$ (if they are in the range $1 \ldots n$) at some point or it cannot be correct. The adversary treats positions $1 \ldots i-2$ and $i+2 \ldots n$ as two separate subproblems. If $i = 1$ or $i = n$ then there is only one subproblem, its size is equal to $0 \pmod 3$, and by induction the claim holds. Otherwise, either $i-2$ or $n-i-1$ is equal to $0 \pmod 3$ and thus the other is equal to $(i-2)+(n-i-1) = n-3 = n \pmod 3$. If $n \neq 1 \pmod 3$ then the sizes of both subproblems are not equal to $1 \pmod 3$. If $n = 1 \pmod 3$ then only one subproblem has size equal to $1 \pmod 3$. In either case, the claim holds. □

   The above proof suggests an algorithm that queries $n-1$ bits when $n = 1 \pmod 3$:

```
HasConsecutiveZeros($B[1 \dots n]$)
    If $n < 2$ return False.
    If $n = 2$ return ($B[1] = 0$ and $B[2] = 0$).
    If $B[2] = 1$ then return HasConsecutiveZeros($B[3 \dots n]$)
        (Note: the algorithm doesn't need to look at $B[1]$).
    If $B[3] = 0$ then return True.
    return (HasConsecutiveZeros($B[4 \dots n]$) or $B[1] = 0$).
```

3. Consider the (infinite) rooted binary tree where a left branch represents a '0' and a right branch represents a '1'. Now every bit string corresponds to a node of this tree. For example, 0 is the left child of the root and 00 is this left child's left child. The nodes that correspond to the bit strings of a prefix code have the property that no node is the ancestor of another. So these nodes are the leaves of a (finite) binary tree. The tree's depth (or height) is the length of the longest bit string in the prefix code. If a binary tree has $n$ leaves then it has depth at least $\lg n$ (we use this fact for our decision tree lower bounds), and thus the longest bit string has length at least $\lg n$.

4. (a) The idea is to compare the middle elements, with index $m = \lfloor (n+1)/2 \rfloor$, of each array. Suppose $A[m] < B[m]$ (the other case is symmetric). For every element of $A[1 \dots m-1]$, there are at least $n-m+1$ elements in $A$ that are bigger and, since $A[m] < B[m]$, there are at least another $n-m+1$ elements in $B$ that are bigger. That is a total of at least $2n - 2m + 2 \geq n + 1$ elements that are bigger. Since the $n$th smallest (the median) has only $n$ elements bigger than it, all of $A[1 \dots m-1]$ must be smaller than the median. Similarly, all of $B[m+1 \dots n]$ must be larger than the median.

   If $n$ is odd, $A[1 \dots m-1]$ has the same number of elements as $B[m+1 \dots n]$. By removing them both, we remove an equal number of elements smaller than the median and larger than the median. The original median is the median of the remaining elements, so we can recurse. If $n$ is even, $A[1 \dots m-1]$ has one fewer element than $B[m+1 \dots n]$, but in this case $A[m]$ cannot be the median[1]. Thus, we remove $A[1 \dots m]$ and $B[m+1 \dots n]$ and we can still recurse.

   In both cases, we decrease the problem size by about a factor of $1/2$. In the worst case, when $n = 3$, we decrease the problem size by a factor $2/3$. Even assuming this worst case decrease, the number of times we recurse before reaching a base case is $O(\log n)$. Each recursive call involves a constant amount of work, so the running time is $O(\log n)$.

```
Median($A[1 \dots n]$, $B[1 \dots n]$)
    If $n = 1$ then
        If $A[1] < B[1]$ then return $A[1]$ else return $B[1]$.
    If $n = 2$ then
        If $A[2] < B[1]$ then return $A[2]$.
        If $B[2] < A[1]$ then return $B[2]$.
        If $A[1] < B[1]$ then return $B[1]$.
        return $A[1]$.
    $m = \lfloor (n+1)/2 \rfloor$
```

[1]If $n$ is even, $2n - 2m + 2 = n + 2$.

```
    If $n$ is even then $m' = m + 1$ else $m' = m$
    If $A[m] < B[m]$ then return Median($A[m' \dots n]$, $B[1 \dots m]$)
    return Median($A[1 \dots m]$, $B[m' \dots n]$)
```

(b) We can model any comparison-based algorithm as a decision tree. Since any correct algorithm for this problem must be able to output any one of the $2n$ input elements as the median, we know that the number of leaves in the decision tree is at least $2n$. Since the decision tree is a binary tree[2], the depth of the tree is $\Omega(\log n)$. Thus the worst case number of comparisons made by any correct algorithm on inputs of total size $2n$ is $\Omega(\log n)$, which matches (asymptotically) the running time of our algorithm. In other words, our algorithm is asymptotically optimal.

[2]We can choose an input in which all the elements are distinct to avoid the possibility that a comparison results in equality.