1: Easy Medians

(a)

**Preprocessing:** Let $M$ denote the median we seek. We first check if $n$ is a power of 2. If so, do nothing. If not, pad the start of array $A$ with $-\infty$ and the end of array $B$ with $\infty$ until both $A$ and $B$ have length equal to the lowest power of 2 that is at least $n$. Let $n'$ be the new length of the arrays. Note that fixing $n'$ to be a power of 2 ensures that the length of the array at each recursive step is a power of 2. Also note that since we added the same number of elements to the start and end of $A$ and $B$, the median does not change.

**Recursive Algorithm:** For convenience, assume 1-indexing.

If the length of $A$ or $B$ is 1, compare and return the smaller of $A[1]$ and $B[1]$.

Else compare $x = A[\frac{n'}{2}]$ to $y = B[\frac{n'}{2}]$:

- If $x < y$, recurse with $A' = A[1, \frac{n'}{2}]$ and $B' = B[1, \frac{n'}{2}]$.
- If $x > y$, recurse with $A' = A[1, \frac{n'}{2}]$ and $B' = B[\frac{n'}{2}+1, n']$.

**Correctness:** We will prove the correctness of the $x < y$ case, the $x > y$ case is symmetric. There are $\frac{n'}{2}$ elements greater than $x$ in $A$, and $\frac{n'}{2}$ elements greater than $y$ (so also greater than $x$) in $B$. So counting $y$, there are at least $\frac{n'}{2} + \frac{n'}{2} + 1 = n' + 1$ elements greater than $x$. By definition, $M$ has $n'$ elements greater than it and $n' - 1$ elements smaller than it, so $x < M$. Similarly, we can show that at least $n' - 1$ elements are smaller than $y$, so $x < M \leq y$. Therefore, no element in $A[\frac{n'}{2}+1, n']$ or $B = B[1, \frac{n'}{2}]$ can be the median. Moreover, since we take away $\frac{n'}{2}$ elements both from above and below the median, the median stays the same for the recursive step. Also note that $A'$ and $B'$ have the same length and stay sorted, so they are valid inputs for the recursive step.

**Runtime:** We begin with an input of size $2n'$, the sum of the lengths of $A$ and $B$. Note that the number of comparisons done is equivalent to the number of recursive steps this algorithm takes, and that at each step the total size of the two arrays is halved (this is exact because we started with an array with length equal to a power of 2). It follows that the algorithm performs $\log_2(2n') = \lceil \log_2(2n) \rceil$ many comparisons.

(b)

We prove that $\lceil \log_2(2n) \rceil$ is also a lower bound on the number of comparisons performed. Observe that any of the $2n$ elements from arrays $A$ and $B$ could be the median. WLOG, element $i$ in $A$ (with $n - i$ elements greater than it in $A$) is the median when there are $i$ elements greater than it in $B$. Also, for each of the $2n$ outputs, there exists some input under which it is the only correct answer (for each input the median is unique). Therefore from the information theoretic model of sorting shown in lecture, we require at least $\lceil \log_2(2n) \rceil$ many comparisons to determine the median.
2: A Variant of Quicksort

Approach 1:
We denote a function call to be successful if it does not enter the case where $L < \frac{n}{3}$ or $L > \frac{2n}{3}$. This will occur if we pick $p$ that has rank between $\frac{n}{3}$ and $\frac{2n}{3}$. This happens with probability $\frac{1}{3}$. During a successful trial, it holds that $|LESS| < \frac{2n}{3}$ and $|GREATER| < \frac{2n}{3}$, so the size of the array in the recursive call is at most $\frac{2n}{3}$ the size of the original array.

Let the expected run-time of the algorithm on an array of size $n$ be $T(n)$. We then have $T(n) \leq \frac{2}{3}T(n) + \frac{1}{3}T(\frac{2n}{3}) + O(n)$.
This implies that $T(n) \leq T(\frac{2n}{3}) + O(n)$.
Solving this recurrence yields $T(n) = O(n)$.

Approach 2:
Upon randomly selecting a pivot, we may end up in one of the three following cases:

1. $p$ is the $k$th smallest element, in which case we are done.
2. $p < \frac{n}{3}$ or $p > \frac{2n}{3}$, in which case we retry pivot selection. This is the BAD case because we cannot guarantee a constant reduction in the number of elements we consider with each round.
3. $\frac{n}{3} \leq p \leq \frac{2n}{3}$. This is the GOOD case because we can guarantee the number of elements being considered reduces by a constant fraction of $\frac{2}{3}$ with each round.

Since this is a randomized algorithm, we can reason about the probabilities of each of the above cases. Case 2 has probability $\frac{2}{3}$, since there are $\frac{2n}{3}$ possible values of $p$ that would result in the BAD case. Case 3 has probability $\frac{1}{3}$, since there are $\frac{n}{3}$ possible values of $p$ that would result in the GOOD case.

Note that this algorithm mimics a geometric distribution, modelling the probability of the first success as a function of the number of trials. The expected value of this type of distribution is $\frac{1}{Pr}$, where $Pr$ is the probability of success. Here, a ”success” would be ending up in the GOOD case implying that the expected number of trials till success is $\frac{1}{\frac{1}{3}} = 3$.

In other words, we can expect to run into the BAD case 3 times before hitting the GOOD case.

This leads to the following recurrence: $T(n) \leq 3(n-1) + T(\frac{2n}{3})$, where we repeat the work of the BAD case 3 times before hitting the GOOD case (therefore reducing the problem size by $\frac{2}{3}$). This recurrence is root-dominated and thus solves to $O(n)$. 


(a)

We will use the adversary method to prove this bound. For all queries \( \{u, v\} \), the adversary answers yes if it does not create a cycle and no otherwise. Suppose the querier outputs an answer in fewer than \( \binom{n}{2} \) queries - we will prove that the adversary can reveal the remainder of the graph such that the answer is wrong.

If the querier says that the graph has a cycle, then reveal a graph where all unqueried \( \{u, v\} \) are not edges. In this case, the querier will be wrong because no edge that was revealed creates a cycle.

Otherwise, if the querier says that the graph is acyclic, consider any unqueried \( \{u, v\} \). Consider any other vertex \( w \) (we are implicitly assuming here that the graph has at least 3 vertices). We can show that the revealed graph either already contains a path from \( u \) to \( w \) that does not use \( \{u, v\} \), or that we can create such a path. The same claim holds true for a path from \( v \) to \( w \) as well. Since there are paths from \( u \) to \( w \) and \( w \) to \( v \) not using \( \{u, v\} \), \( u \) and \( v \) are already connected without \( \{u, v\} \) in the graph. Then, if we reveal \( \{u, v\} \) to be an edge, we will have two different paths from \( u \) to \( v \). This implies that the graph contains a cycle, and that the querier is wrong.

We will now show that the revealed graph either already contains a path from \( u \) to \( w \) that does not use \( \{u, v\} \), or that we can create such a path. If the revealed graph includes \( \{u, w\} \) as an edge or \( \{u, w\} \) is unqueried (in which case we can reveal \( \{u, v\} \) to be an edge), then we are done. Otherwise, \( \{u, w\} \) was queried and found not to contain an edge. The adversary only answers no to a potential edge if it would create a cycle - this means that adding \( \{u, w\} \) to the graph would have created a cycle, which means that \( u \) and \( w \) are already connected in the graph, which implies that there is a path from \( u \) to \( w \) not using \( \{u, v\} \).

The same argument also shows that such a path from \( w \) to \( v \) either exists or can be created, completing the proof.

(b)

For this proof, recall that a bipartite graph contains no odd cycles.

We label the vertices from 1 to \( n \). Now fix a star graph with 1 as the center (this is a graph that contains an edge from 1 to every other vertex, but no other edges). The adversary answers yes when queried about any pair of vertices that forms an edge in this star graph and no to all other queries. Suppose the querier makes fewer than \( \binom{n}{2} - (n - 1) \) queries - then we show that the adversary can reveal the remainder of the graph in a way that the querier’s answer is wrong.

If the querier says that the graph is not bipartite, the adversary can output the star graph. This is bipartite so the querier is wrong.

Suppose the querier says that the graph is bipartite. Then there exists at least one pair \( (j, k) \in \{2,...,n\} \) such that \( (j, k) \) is unqueried. Reveal \( \{j, k\} \) to be an edge. Then the graph has an odd cycle \((1, j, k \text{ form a triangle})\), so it is not bipartite and the querier is wrong again.
View the permutation as a collection of cycles, with edges oriented clockwise. A move \((x, y, z)\) rotates the elements \(A_x, A_y,\) and \(A_z\). There are 4 types of moves:

1. The move merges three cycles into a single cycle. For example, three cycles of size 3 can be made into a single cycle of size 9. This occurs when each element is in a different cycle.

2. The move changes two cycles into two cycles. For example, two cycles of size 2 can be made into a cycle of size 1 and a cycle of size 3. This occurs when two of the elements are in the same cycle, and the third is in a different cycle.

3. The move divides a single cycle into three cycles. For example, a cycle of size 5 can be made into two cycles of size 1 and a cycle of size 3. This occurs when all elements are in the same cycle, and \(x, y, z\) appear in clockwise order.

4. The move changes a single cycle into a single cycle. This occurs when all elements are in the same cycle, and \(x, y, z\) appear in counterclockwise order.

The fourth move doesn’t change the cycle structure of the permutation, so we can ignore it.

Let \(X\) be the number of cycles, and \(E\) be the number of even cycles. Our goal is to use operations to get \(N\) cycles of size 1, which has the property that there are no even cycles. It is easy to check that each move cannot change the parity of the number of even cycles. Thus if \(E\) is odd, there exists no solution. If \(E\) is even, there always exists a solution, and this will be shown by construction.

Suppose that \(E\) is even. Now we present two important claims about the moves, which are easy to prove:

- Only the first two types of moves can decrease the number of even cycles (and can decrease only by exactly 2), but do not increase the number of cycles.
- Only the third move increases the number of cycles (and can increase only by exactly 2), but cannot change the number of even cycles.

Thus to arrive at \(N\) cycles and no even cycles, we need at least \(\frac{E}{2}\) moves of type one or two, and at least \(\frac{(N-X)}{2}\) moves of type three. Note that since \(E\) is even, so is \(N - X\) (show this by casing on the parity of \(X\)). This yields a lower bound of \(\frac{E}{2} + \frac{(N-X)}{2}\) moves.

A possible solution is to first use \(\frac{E}{2}\) operations of type two such that each pair of even cycles is changed to a pair of odd cycles. Now we have \(X\) odd cycles, and can apply \((\frac{N-X}{2})\) moves of type three to obtain \(N\) cycles, each of which must now have size 1. This relies on the fact that moves of type three, when applied to an odd cycle, creates three odd cycles.

For implementation purposes, the simplest move of type three is where \(x, y, z\) are in clockwise order and adjacent. This changes a cycle of length \(L\) into two cycles of length 1 and a cycle of length \(L - 2\).

This solution is optimal because it achieves the lower bound of \(\frac{E}{2} + \frac{(N-X)}{2}\) moves.