A)

We will solve this problem using a divide and conquer algorithm.

First consider the first n/2 elements of the array. Determine if there is a majority element in this set. Then consider the second n/2 elements, and determine if there is a majority element in this set. Either there are majority elements in both sets, one set, or zero sets. If there is no majority element to be found in either set, we can return that there is no majority element in the whole input. If at least one side has a majority element e, then we can iteratively scan to see if e appears in the other side enough times to be considered a majority element for the whole input. If both sides return the same majority element, then return true.

Recurrence relation:

T(n) $\leq 2T(n/2) + O(n)$

when $n > 2$, and

T(2) $\leq O(2)$

Proof: Base case: n = 2. In this case, if both elements are the same, the algorithm returns yes, otherwise it returns no. For an input of size $2^P$, the algorithm is correct, by induction. For an input of size $2^P + 1$, the algorithm divides the input into two sets of size $2^P$, and as we have said, correctly determines whether each set has a majority element. If both sets return yes, and the two elements are the same, the algorithm returns yes. Otherwise, if only one set returns yes, the algorithm checks to see if there are enough of the majority element in the other set to constitute a majority element for the whole input. Lastly, if both sets return no, the algorithm returns no. Thus we have proven that this algorithm works.

B) Linear Time Algorithm

Go through the array and arbitrarily pair each element with another element in the array. If they are different, both elements get discarded. If they are the same, then only one gets thrown away. After going through the array once, there are at most n/2 elements left, because in the worst case scenario, all the elements are the same and so half of them get thrown out. With this implementation, either one of three things is true:
The first would be that all the elements are the same, in which case the result is an array with n/2 of the same elements, so it is clear that this is the majority element.
The second is that none of the elements are the same, in which case the result is an array with 0 elements, so it is clear that there is no majority element.
The last case is that there is potentially a majority element, but it is not obvious. In this case, we know that whatever element is the majority element in our smaller array is the majority element of the whole input.
Because we go through each input in the array, this algorithm takes O(n) time.