1.
Algorithm description in Plain English:

Recursively split the array A into two halves, A_1 and A_2. After there are no more splits to be made (all subarrays are now length 1), iterate over each single element in each subarray and keep track of their counts (if a certain element is encountered for the first time, initialize its count to 1; otherwise, increment its count by 1). Then, merge the subarrays back together to form the original array. During the merging process, combine counts for each element. If the count of any element at any recursion of the merging process is not greater than the ceiling of (j/2) where j is the length of the original subarray that was split, delete the count of that element. Finally, iterate over all final counts and return True if any count is greater than the ceiling of (n/2). Otherwise, return False.

Proof of Correctness:

Note that, in order for there to be a majority element in an array, the repetitions of some element in its first and second halves must be greater than the length of half the original array. The algorithm is based on this observation. It is easy to verify that the algorithm works correctly in the base case (i.e., when the subarrays each contain 1 element). In cases where the subarrays are larger, the subproblem will be formed by smaller subproblems (which are formed by smaller subproblems…) which have been formed by the base case. If at the end of the merging process it is discovered that some element was repeated more times than the ceiling of (n/2), the algorithm returns True, and otherwise False.

Complexity Analysis:

The algorithm's time complexity can be represented by the master theorem T(n) = T(floor(n/2)) + T(ceiling(n/2)) + n, where a = 2, b = 2, and c = 1. Therefore, since c = $\log_b(a)$, the algorithm's total time complexity is Θ(n*log(n)).

2a. Each merging of the algorithm requires O(n) time since it must iterate over each element, and this occurs (k-1) times to merge each array. However, the first array must be traversed (k-1) times, the second array must be traversed (k-2) times, and so on. Therefore, the total time complexity is $O(nk^2)$.
2b. Connect all k arrays together and call the mergesort algorithm, which utilizes divide and conquer, on the new array. Since the length of the new array is kn, the mergesort algorithm's time complexity will now be O(kn*log(kn)).

3a. a = 2, b = 2, c = 4
$c > \log_2(2) = 1$
$T(n) = \Theta(n^c) = \Theta(n^4)$
3b. a = 1, b = (10/7), c = 1
$c > \log_{(10/7)}(1) = 0$
$T(n) = \Theta(n^c) = \Theta(n)$
3c. a = 16, b = 4, c = 2

$c = \log_4(16) = 2$

$T(n) = \Theta(n^c \log(n)) = \Theta(n^2 \log(n))$

3d. $T(n) = T(n-1) + 2 = T(n-2) + 4 = T(n-3) + 6 = T(n-4) + 8 = \ldots$

$T(n) = T(1) + 2(n-1)$

$T(1)$ is $\Theta(1)$ and $2(n-1)$ is $\Theta(n)$. Therefore $T(n) = \Theta(n)$.

4.

Algorithm description in Plain English:

Initialize an integer variable i as 0. While $A[2^i] \neq \infty$, increment i by 1. Then, perform a binary search using 0 as the left bound and the final 2^i value as the right bound. If the binary search finds the integer x in between the bounds, then return x's position in the array. If the binary search ends and x is never found, then return Fail.

Proof of Correctness:

The incrementation of i is used to find a right bound to perform a binary search. Most likely, it will result in an oversized array. However, this oversized array is guaranteed to contain all of the n sorted integers, as well as the integer x if it exists in the original infinite array. Assuming the implementation of binary search is done correctly, the algorithm will return x's position if it exists in the array or Fail if it does not.

Complexity Analysis:

The incrementation of i occurs in $O(\log(n))$ time. The binary search also occurs in $O(\log(n))$ time. Therefore, the total time complexity is $O(\log(n))$.

5.

Algorithm description in Plain English:

Initialize an array A to contain the sequence of n numbers, and an answer variable to represent the amount of significant inversions. Split the current array in half, with the first half containing the floor of (n/2) and the second half containing the ceiling of (n/2). Iterate over each element in the first half - during each iteration, increment the answer by 1 for each element in the second half that is greater than the current element in the first half, and has a greater index (make sure to use the new indexes based on the new arrays, not the original indexes from the original array). Recursively split and iterate on the first half, and stop after reaching a first half that contains 1 element.

Proof of Correctness

Note that, in order for a significant inversion to exist, one element must be in the first half of a subarray starting at index 0, and the second element must be in the second half with a greater index. This algorithm is based on this observation. It is easy to verify that the algorithm works correctly in the base case (i.e., when the subarrays each contain 1 element). In cases where the subarrays are larger, the subarray will still always start at index 0 (meaning that for any element in the first half, significant inversions can occur for any element in the second half as long as the element in the second half has a greater index). At the end of the recursion when the first half can no longer be split, the algorithm returns the number of significant inversions found.

Complexity Analysis

The algorithm's time complexity can be represented by the master theorem T(n) = T(floor(n/2)) + T(ceiling(n/2)) + n, where a = 2, b = 2, and c = 1. Therefore, since $c = \log_b(a)$, the algorithm's total time complexity is Θ(n*log(n)).