Complete Solution to Majority Element Class problem:

(Das Gupta) An array A[1 . . . n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is A[i] > A[j]?". However you can answer questions of the form: "is A[i] = A[j]?" in constant time.

a) Classical divide and conquer: split into two subsets, A_1 and A_2, …, and show T(n) is O(n log n).

If A has a majority element v, v must also be a majority element of A_1 or A_2 or both. The equivalent contra-positive restatement is immediate: (If v is <= half in each, it is <= half in the total.) If both parts have the same majority element, it is automatically the majority element for A. If one of the parts has a majority element, count the number of repetitions of that element in both parts (in O(n) time) to see if it is a majority element. If both parts have a majority, you may need to do this count for each of the two candidates, still O(n). This splitting can be done recursively. The base case is when n = 1. A recurrence relation is T(n) = 2T(n/2) + O(n), so T(n) is O(n log n) by the Master Theorem.

b) Find an O(n) solution using the reduction technique for pairs indicated in the first bullet below. There are added complications coming from an odd count.

The invariant needed for the solution is:

Proposition P: Given a list L of n > 0 items, and a tiebreaker item, T, which may be None. A majority element is one that is a majority in L or is T if exactly half of L matches T. If there is a majority for (L, T) then there is the same majority for (L1, T1) where

- L1 is formed by arbitrarily partitioning L into pairs, and including an element from each pair where the two elements are the same.
- T1 is the odd unpaired element of L if n is odd, and T1 is T is n is even.

The tiebreaker is not a part of the original problem (when T = None), but it can be a part of sub-problems. The tiebreaker can be thought to get a positive contribution to the vote, but less than one full vote. This is enough to break a tie, but not enough to overturn a majority with at least one extra vote.

Proof: Assume there is a majority element M. Let m be the number of copies of M and u be the count of everything else, and e be the excess, m - u. Then having a direct majority element means e > 0. For the recursion you need to allow something more like the Senate with a tiebreaker vote T, so the more general requirements would be
(e > 0) or (e == 0 and M == T).
Throwing out unmatched pairs will decrease u at least as much as m, and hence not decrease e, so it will not invalidate the majority element. Consider two cases:
1. If n is odd, the tiebreaker is irrelevant, since there will never be a tie. Since unmatched pairs are also tossed, it is enough to consider the L1 pairs and the odd extra

element from L. The odd extra element does serve as a tiebreaker with the L1 pairs, since each pair gets essentially 2 votes, and the odd element only makes a difference if the vote exactly matches in L1, so the same majority will be found for (L1, T1).
2. If n is even, e must be even. Then (e >= 0 and M == T) or (e >= 2). At the next level the numbers are halved, leaving (e >= 0 and M == T) or (e >=1), still implying M is the majority element.

The algorithm:
Note that the base case is when n is 0. At that point the majority element is the tiebreaker (which may be None).

The implication in proposition P indicates that at each level there is at most one candidate for majority element, delivered from the base case or the next lower case down. The algorithm only needs to confirm that the one candidate from the recursive call is actually in the majority, or else there is no majority element.

The initial call to the function is with tiebreaker=None, so any majority comes from an actual majority in the list.

The non-recursive work is O(n) in traversing the list to make pairs before the recursive step and then O(n) again after the recursion to count proposed majority elements. This direct work is O(n), so the recurrence relation is $T(n) = T(n/2) + O(n)$, resulting in $T(n) = O(n)$, by applying the Master Theorem.

```
def majority(data, tiebreaker =None):   // complete code, with testing, in majority.py
   '''Return the majority element of sequence data, OR
   tiebreaker   -  if  exactly half of the elements in data are tiebreaker , OR
   None otherwise.
   '''
   n = len(data)
   if n == 0:
      return tiebreaker
   pairs = [] # empty list
   if n % 2 == 1:
      tiebreaker  = data[-1] # last element in Python sequence
   for i in range(0, n-1, 2): # for(i = 0; i < n-1; i+=2)
      if data[i] == data[i+1]:
         pairs.append(data[i])
   major = majority(pairs, tiebreaker )
   if major is None:
      return None
   nMajor = data.count(major) # handy method does the obvious
   if 2*nMajor > n or (2*nMajor == n and major == tiebreaker ):
      return major
   return None # candidate did not pan out
```