1. (a) false

   counter example:

   Let $f(n) = 3n$ and $g(n) = n$. Then $f(n/3) - g(n) = 0 \neq \Omega(n)$

   (b) true

   $\exists c', \exists n_0' s.t. \forall n > n_0', f(n) < c'g(n)$

   $\exists c'' > 0, \exists n_0'' s.t. \forall n > n_0'', h(n) > c''g(n)$

   Rearranging, $g(n) < \frac{h(n)}{c''}$.

   Therefore, $\exists c = \frac{c'}{c''}, \exists n_0''' = max(n_0', n_0'') \quad s.t. \ \forall n > n_o''', f(n) < c'(g(n)) < \frac{c'}{c''}h(n)$.

   (c) true

   $\log_3(n) = \frac{\log_5(n)}{\log_5(3)} = \Theta(\log_5(n))$, because $\log_5(3)$ is a constant.

   (d) false

   $n^2|\sin n| \neq \Omega(n^2)$:

   $\forall c, \forall n_0 s.t. \exists n > n_0, n^2|\sin n| < cn^2$

   Take $n > n_0$, such that n is multiple of π. Then $n^2|\sin n| = 0 < cn^2$.

   (e) true

   For $n > 4$, $(\log(n))^{(n/10)} \geq 2^{(n/10)}$

   $2^{(n/10)} = \omega(n^3)$

   Therefore, $n^2 + (\log(n))^{(n/10)} = \omega(n^3)$

2. (a) $n^{\log_b(a)} = n^{\log_3(9)} = n^2$
$n^{(1/2)} \log n \leq c(n^{2-\epsilon})$ for $c = 1$ and $\epsilon = .5$
Therefore, $T(n) = \Theta(n^2)$ by case 1 of the master theorem.

(b) $n^{\log_b(a)} = n^{\log_4(2)} = n^{(1/2)}$
$n \geq c(n^{.5+\epsilon})$ for $c = 1$ and $\epsilon = .25$
$2f(\frac{n}{4}) = 2(\frac{n}{4}) = \frac{n}{2} \leq cf(n) = cn$ for $c = .75$ and all $n \geq 0$
Therefore, $T(n) = \Theta(n)$ by case 3 of the master theorem.

3. Algorithm

```
FindLargest(array A of size n, l, r):
        if (n == 1):
                return A[0]:
        if (n == 2):
                return max(A[0],A[1]):
        mid = ⌊(l+r)/2⌋
        if (A[mid] >A[mid+1]):
                return A[mid]
        else if (A[l] > A[mid]): //left half of array is unsorted half
                return FindLargest(A, l, mid-1)
        else: //right half of array is unsorted half
                return FindLargest(A, mid+1, r)
```

To solve the problem posed by the question, call FindLargest(A, 0, len(A)-1).

Running Time:

This algorithm is a modification of binary search, as we are reducing the size of the problem by half each step of the recursion and spend $O(1)$, thus it has running time:

$$T(n) = T(n/2) + O(1) = O(\log n) \tag{1}$$

Correctness:

The base case n=1 is trivial. In the base case n=2, the largest value in the array is clearly max(A[0],A[1]).

Induction Hypothesis: Our algorithm returns correct results for n = k

Induction step: Let n = k + 1. Our input is the same as the input in the n=k case, with some (k+1)th element that is larger than all other elements inserted at the correct point in the array (such that the array is still a circularly shifted sorted array). If the (k+1)th (largest) element is in the $mid = \lfloor \frac{l+r}{2} \rfloor$ position, then A[mid] >A[mid+1] and the (k+1)th (largest) element is returned. If the (k+1)th element is in the first half of the array, then the second half of the array is sorted and the first half of the array is unsorted. Therefore, A[l] > A[mid] and our algorithm recurses on the left half of the array. The left half of the array is a circularly shifted sorted array of size less than (k-1), so we get the correct answer by our induction hypothesis. If the (k+1)th element is in the second half of the array, then the second half of the array is unsorted and the first half of the array is sorted. Therefore, A[l] $\leq$ A[mid] and our algorithm recurses on the right half of the array. The right half of the array is a circularly shifted sorted array of size less than (k-1), so we get the correct answer by our induction hypothesis.

4. **Algorithm**:

```
FindMode(array A of size n):
        Perform radix sort base n on A
        i = 0
        count = 1
        maxCount = 0
        mostCommonElement = null
        while (i < n):
                if (i < n-1) and (A[i] == A[i+1])
                        count = count + 1
                else:
                        if (count > maxCount):
                                maxCount = count
                                mostCommonElement = A[i]
                        count = 1
                i = i + 1
```

Running time:

Besides the radix sort, the algorithm clearly takes $O(n)$ time, because we do constant work for each i from 0 to $n-1$. As we know from class, the running time of the radix sort is

$$O(d(n+k)) \tag{2}$$

Because we are using base n representation, largest value of each digit is n:

$$k = n \tag{3}$$

and number of digits d is:

$$d = \log_n(n^2) \tag{4}$$

Therefore:

$$O(d(n+k)) = O(n) \tag{5}$$

Correctness:

The correctness of radix sort was proved in class/in the book. Proof of the rest of the algorithm is trivial if we use loop invariant: $mostCommonElement$ is the most common element seen so far, except may be the last inspected one, and *count* contains number of the last inspected element so far.

Initialization:

mostCommonElement = null and $count = 0$ as we didn't see anybody so far. Correct.

Maintenance:

if current inspected element is the same as last one, then we just update the *count*, and *mostCommonElement* keeps it property automatically.

If current element differs from the last inspected, then we need to check if last seen has larger count then current *mostCommonElement*, if so we update *mostCommonElement*. Afterwards we reboot the counter to zero and increment it.

Termination:

Automatically we get *mostCommonElement* in the full array.