Solution to Q1: Practicing Recurrence Relations

(a) For $T(n) \leq 3T(n/2) + O(n \log^3 n)$, we have:
   - $a = 3$ and $b = 2$; thus, $n^{\log_b a} = n^{\log_2 3}$.
   - $f(n) = O(n \log^3 n)$.

Hence, by case 1 of the Master theorem, $T(n) = O(n^{\log_2 3})$.

(b) For $T(n) \leq 4T(n/2) + O(n^2)$, we have:
   - $a = 4$ and $b = 2$; thus, $n^{\log_b a} = n^{\log_2 4} = n^2$.
   - $f(n) = O(n^2)$.

Hence, by case 2 of the Master theorem, $T(n) = O(n^2 \log n)$.

(c) For $T(n) \leq 2T(n/2) + O(n \log^2 n)$, we have:
   - $a = 2$ and $b = 2$; thus, $n^{\log_b a} = n^{\log_2 2} = n$.
   - $f(n) = O(n \log^2 n)$.

Hence, again by case 2 of the Master theorem, $T(n) = O(n \log^3 n)$.

(d) For $T(n) \leq 2T(n/4) + O(n^{0.5001})$, we have:
   - $a = 2$ and $b = 4$; thus, $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$.
   - $f(n) = O(n^{0.5001})$.

Hence, by case 3 of the Master theorem, $T(n) = O(n^{0.5001})$.

Solution to Q2: Circularly Shifted Sorted Array

Algorithm:
Function LargestInShifted($A[1 \ldots n]$)

- Else:
  - Set $m \leftarrow \lfloor n/2 \rfloor$
If $A[n] \geq A[m+1]$ then return $\text{LargestInShifted}(A[1 \ldots m])$.
Else, return $\text{LargestInShifted}(A[(m+1) \ldots (n-1)])$.

Then, the desired solution is $\text{LargestInShifted}(A[1 \ldots n])$.

**Correctness:** The proof is by induction on $n$. Let $i = k+1$ be the index of the maximum element of $A$. The following cases are exhaustive:

- **Assume $1 \leq i \leq m$.** Then $A[m+1] \leq A[n] < A[1]$, so the algorithm outputs $\text{LargestInShifted}(A[1 \ldots m])$. By induction, this equals the maximum of $(A[1], \ldots, A[m])$, which is $A[i]$.
- **Assume $m+1 \leq i < n$.** Then $A[n] < A[1] < A[m+1]$, so the algorithm outputs $\text{LargestInShifted}(A[(m+1) \ldots (n-1)])$. By induction, this equals the maximum of $(A[m+1], \ldots, A[n-1])$, which is $A[i]$.

**Running Time:** If $T(n)$ is the running time of the algorithm, then we have $T(n) = T(n/2) + O(1)$ with $T(1) = O(1)$. Using the master theorem, this gives $T(n) = O(\log n)$.

**Solution to Q3: Majority Element**

Here’s a solution with runtime $O(n \log n)$. We recursively solve the following more general problem: output the majority element of $A$ if such an element exists, and output ⊥ otherwise.

If $A$ has a single element then the solution is to output that element. Otherwise, start by dividing $A$ into two halves, $L$ and $R$, such that $|L| = |R|$ or $|L| = |R| + 1$. (Here, $|L|$ denotes the number of elements in $L$.)

If $A$ has a majority element $x$, then $x$ is a majority element of either $L$ or $R$. We prove the contrapositive of this statement as follows. If $x$ is not a majority element of $L$ or $R$, then $x$ occurs at most $|L|/2$ times in $L$ and at most $|R|/2$ times in $R$. Therefore $x$ occurs at most $|L|/2 + |R|/2 = |A|/2$ times in $A$, so $x$ is not a majority element of $A$.

Recursively determine whether $L$ has a majority element, and what that element is if it exists. Do the same for $R$. This gives a set $S$ of the majority elements of $L$ and $R$, where $S$ has cardinality 0, 1 or 2. If $A$ has a majority element, then that element is in $S$. So for each $x$ in $S$, count the number of occurrences of $x$ in $A$, and if this number is greater than $n/2$ then output $x$. If this procedure does not output any element, then return ⊥.

The runtime is described by the recurrence $T(n) \leq 2T(n/2) + O(n)$, so the master theorem gives $T(n) = O(n \log n)$.
A more sophisticated $O(n)$ time algorithm is possible. (See https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm). This cannot be achieved by computing the median in $O(n)$ time, because the question only allows equality checks, and not comparisons.

**Solution to Q4: Monotonic Function Evaluation**

Let $k = 1$, and while $f(k) > 0$, double $k$. In at most $\lceil \log n \rceil$ iterations, this will terminate as we will have $k \geq n$. Let $k^*$ be the value at which it terminates. Then, we know that $k^*/2 < n \leq k^*$. We can binary-search $n$ in this range, as described in more detail below.

The running time for finding $k^*$ is $O(\log n)$, and the running time for the subsequent binary search is also $O(\log n)$. Hence, the overall time is $O(\log n)$.

Function FindFirstNonPositive($A[1 \ldots r]$)

- If $r = 1$ then return $A[1]$.
- Else:
  - Set $m \leftarrow \lfloor r/2 \rfloor$
  - If $A[m] \leq 0$ then return FindFirstNonPositive($A[1 \ldots m]$).
  - Else, return FindFirstNonPositive($A[(m + 1) \ldots r]$).