1. Implement the three different approaches we discussed in the class to find "a peak" of a given 1-dimensional array of integers, using your favorite programming language. An element is said to be a peak if and only if it is greater than or equal to its left and right neighbors.
   a. Global maximum, linear search, divide and conquer (**2 pts each = 6 pts**)
   b. Test and provide runtimes of each approach for an array of size 1 million random integers. How does the run time change when array is sorted? (**1 pt**)
   c. When definition of peak is changed to greater than its neighbors, instead of greater than and equal to, what impact does it have on the problem? Are you guaranteed to find a peak ? Provide proper reasoning to support your argument. (**3 pts**)
   d. How can you extend the peak finding problem to 2-dimensional array of integers, when a peak is said to be greater and or equal to all its four (left, right, top, bottom) neighbors. Provide an efficient implementation (**5 pts, extra credit**)

2. Solve the following recurrences and give its upper bound (**10 pts**)
   a. $T(n) = 2T(n/4) + 1$
   b. $T(n) = 2T(n/4) + n^2$
   c. $T(n) = T(n-2) + 2n$
   d. $T(n) = 4T(n/2) + n^2 \log n$
   e. $T(n) = 3T(\sqrt{n}) + \log n$

3. Indicate, for each pair of functions f(n) and g(n) in the table below, whether f(n) is O, Ω and Θ of g(n). Assume that $k \geq 1$, $e > 0$, and $c > 1$ are constants. Your answers should in the form of the table with "yes" or "no" written in each box. (**3 pts**)

f(n)	g(n)	O	Ω	Θ
$\log^k n$	n^e			
n^k	c^n			
2^n	$2^{n/2}$			
$\log(n!)$	$\log(n^n)$			
$\sqrt{n}$	$n^{\sin(n)}$			
$n^{\log c}$	$c^{\log n}$			

4. Prove that running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$. (**2 pts**)

5. Arrange the following in order from smallest to largest. (**2 pts**)

    $O(n^2)$, $O(1)$, $O((\log n)^2)$, $O(n \sqrt{n})$, $O(n \log n)$

6. What is the worst-case complexity for the code below? Show the recurrence relation (**3 pts**)

```
int someFunctionFoo(int n) {
    int count = 0;
    int cur = 1;
    while (cur < n) {
        count++;
        cur = cur * 2;
    }
    return cur;
}
```