There are 60 regular and 15 extra credit points available on this problem set.

1. (20 pts) *Solve the following recurrence relations using any of the following methods: unrolling, tail recursion, recurrence tree (include tree diagram), or expansion. Each each case, show your work.*

   (a) $T(n) = T(n-1) + 2^n$ *if* $n > 1$, *and* $T(1) = 2$

$$\begin{aligned} T(n) &= T(n-1) + 2^n \\ &= T(n-2) + 2^{n-1} + 2^n \\ &= T(n-3) + 2^{n-2} + 2^{n-1} + 2^n \\ &\vdots \\ &= T(1) + 2^2 + \ldots + 2^n \\ &= 2(2^n - 1) \end{aligned}$$

   (b) $T(n) = T(\sqrt{n}) + 1$ *if* $n > 2$, *and* $T(n) = 0$ *otherwise*

$$\begin{aligned} T(n) &= T(\sqrt{(n)}) + 1 \\ &= T(n^{\frac{1}{4}}) + 2 \\ &= T(n^{\frac{1}{8}}) + 3 \\ &\vdots \\ &= T(n^{\frac{1}{2^j}}) + j \end{aligned}$$

   We seek to find value of j when $n^{\frac{1}{2^j}} \leq 2$. We obtain $\frac{1}{2^j}\log(n) \leq 1$ and therefore $j \geq \log_2(\log_2(n))$. Therefore, $T(n) = T(2) + \log_2(\log_2(n)) = \Theta(\log_2(\log_2(n)))$.

2. (10 pts) *Consider the following function:*

```
def foo(n) {
     if (n > 1) {
        print( ''hello'' )
        foo(n/4)
        foo(n/4)
}}
```

In terms of the input n, determine how many times is "hello" printed. Write down a recurrence and solve using the Master method.

Let $H(n)$ count the number of times "hello" is printed:

$$H(n) = 2H(n/4) + 1 \qquad n > 1 \ ,$$

and $H(n) = 0$ otherwise. Using the Master method with $\epsilon = \log_b a = \log_4 2 = 0.5$, we obtain $H(n) = \Theta(\sqrt{n})$ from case 1.

3. (30 pts total) Professor McGonagall asks you to help her with some arrays that are *spiked*. A *spiked* array has the property that the subarray $A[1, \ldots, i]$ has the property that $A[j] < A[j+1]$ for $1 \leq j < i$, and the subarray $A[i, \ldots, n]$ has the property that $A[j] > A[j+1]$ for $i \leq j < n$. Using her wand, McGonagall writes the following *spiked* array on the board: $A = [7, 8, 10, 15, 16, 23, 19, 17, 4, 1]$, as an example.

   (a) *Write a recursive algorithm that takes asymptotically sub-linear time to find the maximum element of A.*

   A variation on binary search will work. As in binary search, we can solve the problem recursively by ensuring that the precondition on A also holds for the smaller instance that we select for recursion. The precondition is that A is a spiked array. Note that this implies the base case: an array of length $n = 3$, because no smaller array can satisfy the conditions of being spiked. And, like binary search, to reduce the current problem to one of half the size, we examine the mid-point of A and make a decision about whether the target lays in the left-half or the right-half of A. The decision here is based on the slope of the function thereif the slope is positive, then the midpoint must be on the left-side of the spike and the spike must lay in the right-half of the problem, while if the slope is negative, the reverse is true. To make the boundary conditions work, we have to control for the possibility that the midpoint is the spike, and thus we recurse on the left-or right-half plus one element from the other half. Here is pseudocode for this recursive algorithm:

```
def findSpike(A, s, t):
    mid = floor((s + t) / 2)  // split A in half
    if t - s + 1 == 3 {return A[mid]} // base case
    if A[mid] < A[mid + 1] {
        return findSpike(A, mid, t) // spike is in right-half of A
    } else {
        return findSpike(A, s, mid + 1) // spike is in left-half of A
    }
```

(b) *Prove that your algorithm is correct. (Hint: prove that your algorithms correctness follows from the correctness of another correct algorithm we already know.)*

The correctness of `findSpike` follows a very similar logic to that of binary search. Recall that there are at least two ways to prove the correctness of a binary search algorithm: we can either use strong induction, or we can use an invariant on the size of the array in which we know the target does not appear. We can use strong induction here pretty simply.

First, `findSpike` is correct for the smallest possible spiked array (the base case), in which A has three elements. Proof: By definition, the spike must be in $A[2]$, and in this case, `findSpike` computes `mid=2` and the first conditional is true, hence the algorithm returns $A[2]$ correctly.

Now, assuming `findSpike` is correct for all spiked arrays of any size $3 \leq i \leq n$, `findSpike` is correct for a spiked array of size $n+1$. Proof: Because the array is spiked, if $A[mid] < A[mid+1]$ then the spike must be at some $A[i]$ for $i > mid$, i.e., $A[mid]$ must be to the left of the spike. In this case, we recurse on the subarray $A[mid..t]$, which is also a spiked array (if the spike is at $A[mid+1]$, including $A[mid]$ in the subarray guarantees that $A[mid..t]$ is also a spiked array), and has size strictly smaller than $n+1$: $t-mid+1 = t-\lfloor(s+t)/2\rfloor+1 < t-s+1 = n+1$. Since we've assumed `findSpike` works on all ranges smaller than $n+1$, this case is correct. The logic for when $A[mid] > A[mid+1]$ is similar, except that now A[mid+1] must be to the right of the spike, and including it in the subarray we recurse on guarantees it is also a spikeed array. (Note that the equality case of

$A[mid] == A[mid+1]$ is excluded by the definition of the spikeed array.) Hence, the algorithm is correct.

(c) *Now consider the multi-spiked generalization, in which the array contains k spikes, i.e., it contains k subarrays, each of which is itself a spiked array. Let $k = 2$ and prove that your algorithm can fail on such an input.*

Proof by counter example: $A = [1, 4, 3, 2, 1, 5, 3]$. In the first call, `findSpike` chooses mid = 4 and $A[4] = 2$. Now, $A[4] > A[5]$, so the algorithm recurses on the subarray $[1, 4, 3, 2, 1]$ and eventually returns $A[2] = 4$ as the largest value. But, the largest value is $A[6] = 5$. In fact, any double-spiked array A in which the initial midpoint is located within the subarray corresponding to the lower of the two spikes suffices as a counter example, because `findSpike` only climbs to the top of the nearest spike, and hence is only guaranteed to find a local optimum.

(d) *Suppose that $k = 2$ and we can guarantee that neither spike is closer than $n/4$ positions to the middle of the array, and that the joining point of the two singly-spiked subarrays lays in the middle half of the array. Now write an algorithm that returns the maximum element of A in sublinear time. Prove that your algorithm is correct, give a recurrence relation for its running time, and solve for its asymptotic behavior.*

Because the two spikes cannot be within $n/4$ positions of the middle of A, we know that the subarray $-1xA[n/4+1, 3n/4-1]$ is itself a single spiked array. That is, its an array with a single minimum in it. Thus, we can define a function `findValley` that is identical to `findSpike` except that we change the $>$ comparison to a $<$ comparison, and we change return $A[mid]$ to return mid, so that instead of returning the value, we return its index. Correctness follows from the correctness of `findSpike`.

To solve the problem described here, we simply run `findValley` on the subarray $A[n/4+1, 3n/4-1]$ to obtain the index k that is the "joining point" between the two single-spiked subarrays of A. Then, we return the spike value in either of the two single-spiked subarrays: `max( findSpike(A,1,k) , findSpike(A,k,n) )`.

The recurrence relation for this algorithm is $T(n) = 3G(n/2) + c_1$, where $G(n) = G(n/2) + c_2$ is the recurrence relation for `findSpike`. We have already shown that $G(n) = \mathcal{O}(\log n)$, and hence it follows that $T(n) = \mathcal{O}(\log n)$ as well.

4. (15 pts extra credit) *Asymptotic relations like O, Ω, and Θ represent relationships between functions, and these relationships are transitive. That is, if some $f(n) =$*

$\Omega(g(n))$, *and* $g(n) = \Omega(h(n))$, *then it is also true that* $f(n) = \Omega(h(n))$. *This means that we can sort functions by their asymptotic growth.*

Sort the following functions *by order of asymptotic growth such that the final arrangement of functions* $g_1, g_2, \ldots, g_{12}$ *satisfies the ordering constraint* $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{11} = \Omega(g_{12})$.

n	n^2	$(\sqrt{2})^{\lg n}$	$2^{\lg^* n}$	$n!$	$(\lg n)!$	$\left(\frac{3}{2}\right)^n$	$n^{1/\lg n}$	$n \lg n$	$\lg(n!)$	e^n	1

Give the final sorted list and identify which pair(s) functions $f(n), g(n)$, *if any, are in the same equivalence class, i.e.,* $f(n) = \Theta(g(n))$.

A few of these comparisons require some careful thought.

For instance, $(\lg n)! = \omega(n^2)$ by taking logarithms of both sides and then applying Stirling's approximation to the left-hand side.

The trickiest is probably $2^{\lg^* n}$ (the iterated logarithm, pronounced "log star"); the identity $\lg^* n = 1 + \lg^*(\lg n)$ can be helpful, but here it is better to notice that because the iterated logarithm grows *extremely* slowly, it must be $\omega(1)$ but $o((1/2)\lg n)$ (Do you see why?).

For other comparisons, these basic identities are useful:

$$a^{\log_b c} = c^{\log_b a}$$
$$\left(a^b\right)^c = a^{b \cdot c}$$
$$\left(\sqrt[k]{k}\right)^{\log_k n} = \sqrt[k]{n}$$

Applying these ideas to clean up an initial rough sort, yields a final sorted list of

$n!$	e^n	$\left(\frac{3}{2}\right)^n$	$(\lg n)!$	n^2	$n \lg n \quad \lg(n!)$	n	$(\sqrt{2})^{\lg n}$	$2^{\lg^* n}$	$n^{1/\lg n} \quad 1$

with the following equivalence classes: $\{n \lg n, \lg(n!)\}$ and $\left\{n^{1/\lg n}, 1\right\}$.