Adam Ten Hoeve
adam.tenhoeve@colorado.edu

---

1. *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*

SOLUTION:
T(n) = T(n-1) + 2^n
T(n-1) = T(n-2) + 2^{n-1}
T(n-2) = T(n-3) + 2^{n-2}
...
T(1) = 2

Plug back in
T(n-2) = (2 + ... + 2^{n-3}) + 2^{n-2}
T(n-1) = (2 + ... + 2^{n-3} + 2^{n-2}) + 2^{n-1}
T(n) = (2 + ... + 2^{n-3} + 2^{n-2} + 2^{n-1}) + 2^n

Notice that when each is plugged back into the function recursively, it forms a summation.

$$T(n) = \sum_{x=1}^{n} 2^x$$

This summation can be converted to the equation:

$$T(n) = 2(\frac{1-2^n}{1-2}) = 2^{n+1} - 2$$

Using this equation, we can see that T(n) = $\Theta(2^n)$.

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

SOLUTION:
T(n) = T($n^{1/2}$) + 1
T($n^{1/2}$) = T($n^{1/4}$) + 1
T($n^{1/4}$) = T($n^{1/8}$) + 1
...
T(n) = 0 when n <= 2

T($n^{1/2}$) = (T($n^{1/8}$) + 1) + 1 = T($n^{1/8}$) + 2
T(n) = (T($n^{1/8}$) + 2) + 1 = T($n^{1/8}$) + 3

$$T(n) = T(n^{\frac{1}{2^x}}) + x$$

In order to be valid, $n^{\frac{1}{2^x}}$ must be less than or equal to 2. So we can solve that equation in terms of j to find the maximum number of iterations depending on the size of n.

$$n^{\frac{1}{2^x}} \leq 2$$

$$\frac{1}{2^x} * log(n) \leq log(2)$$

$$\frac{log(n)}{log(2)} \leq 2^x$$

$$log_2(n) \leq 2^x$$

$$log(log_2(n)) \leq x * log(2)$$

$$log_2(log_2(n)) \leq x$$

x represents the number of iterations for any n. The number of iterations in the function is also the number of operations it performs, so the order of the function is $T(n) = \Theta(log_2(log_2(n))$.

2. *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.

SOLUTION:
T(n) = 2T($n/4$) + 1
a = 2 because each recursive iteration splits into 2 more functions.
b = 4 because each call is 1/4 the size of input n.
c = 0 because the only initial condition is the if statement.
$log_4 2 = \frac{1}{2} > c$ so the leaves dominate the cost of the function.
So T(n) = $\Theta(n^{log_b a})$
Therefor T(n) = $\Theta(n^{log_4 2}) = \Theta(\sqrt{n})$
This means that "hello" is printed approximately $\sqrt{n}$ times by the end of the function.

3. *Professor McGonagall asks you to help her with some arrays that are spiked. A spiked array has the property that the subarray A[1..i] has the property that A[j] < A[j + 1] for 1 <= j < i, and the subarray A[i..n] has the property that A[j] > A[j + 1] for i 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.*

SOLUTION:

```
findSpike(A, low, high)
{
    mid = (low + high) / 2
    if (A[mid] > A[mid + 1] and A[mid] > A[mid - 1])
        return mid
    else if (A[mid] > A[mid + 1])
        findSpike(A, low, mid - 1)
    else
        findSpike(A, mid + 1, high)
}
```

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

SOLUTION:
Base Case: Assume A is spiked. A is an array of 3 elements where if **A[low]** < **A[mid]** > **A[high]**, then A[mid] is the spike of the array.

Inductive Hypothesis: Assume the algorithm is true for any array sized n such that $3 < n \leq k$.

Case 1: If A[mid] < A[mid + 1], then the spike exists to the right of the mid position, so the algorithm increases the position of mid by moving low to mid+1.

Case 2: If A[mid] < A[mid + 1], the the spike exists to the left of the mid position so the algorithm moves the position of mid further to the left by reassigning high to mid-1.

Case 3: If A[mid] > A[mid + 1] and A[mid] > A[mid - 1], then the spike is at the mid position. The algorithm returns mid, which is the intended behavior.

   (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*

SOLUTION: A is an array of size 5 with a spike at A[1] and A[3]. When the algorithm begins, low = 0 and high = 4 and mid = 2. The algorithm will fail the first if statement, meaning A[2] is not a spike, but will succeed the next else if statement. The function will then recursively call the function as findSpike(A, 0, 1). Mid = 1. The function will then complete the first if statement and return 1 as the spike location. However, the algorithm will not find the spike at A[3] so the algorithm will fail if there is more than one spike.

(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 singlyspiked 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.*

SOLUTION:

```
    findSpike(A, low, high)
{
    mid = (low + high) / 2
    if (A[mid] > A[mid + 1] and A[mid] > A[mid - 1])
        return A[mid]
    else if (A[mid] > A[mid + 1])
        findSpike(A, low, mid - 1)
    else
        findSpike(A, mid + 1, high)
}

twoSpike(A, low, high)
{
    x = floor((low + high) / 2)
    m = max(findSpike(A,low,x), findSpike(A, x+1, high))
    return m
}
```

Proving that the function is correct:

Base Case: Assume A has 2 spikes. A is an array with 5 elements with spikes located at A[1] and A[3]. The Array must be at least 5 elements long in order to have 2 spikes. The algorithm begins with twoSpike(A, low, high) where low = 0 and high = 4. x = $\frac{0+4}{2}$ = 2. The algorithm will then call the findSpike function from 0 to 2 and 3 to 4. The first subarray from 0 to 2 will have mid = 1, which is the location of the first spike, and will return the value of A[1]. The second subarray from 3 to 4 will have mid = 3, which is the location of the spike, and will return the value of A[3]. The max function will then compare the values from A[1] and A[3] and stores the maximum into m. m is then returned from the function, which is correct behavior.

Inductive Hypothesis: Assume that twoSpikes() is correct for an array of length n with k = 2 spikes, one that is located in the first n/4 of the array, and the second is in the last 3n/4 of the array. We will now show that the algorithm is still correct for an array of length n+1.

Case: By adding one to the length of the array, the relative positions of the spikes within the arrays do not change. There is still one within A[0 ... n/4] and one within A[3n/4 ... n]. The algorithm sends the first half and last half of A to the findSpike() function, which has already been proven to be correct in *part b* when passed a subarray with one spike. The only difference from *part b* is that findSpike() now returns the value at A[spike] rather than the index position. Because we know the relative positions of the spikes, we know only one has been passed to both calls of findSpike(). The values of the spikes are returned to twoSpike() and the maximum of the two is returned, which is intended behavior.

T(n) = 2G(n) + c
G(n) = $\Theta(log(n))$
T(n) = 2log(n) + c
T(n) = $\Theta(log(n))$

The twoSpike() function splits the array into two recursive calls that have a different runtime G(n). There is also some constant cost because of the max function. G(n) is similar to binary search so it will have an order of T(n) = $\Theta(log(n))$. If we plug this back into the T(n) function, we can see that T(n) is also order $\Theta(log(n))$.

4. *Asymptotic relations like O, Ω, and Θ represent relationships between functions, and these relationships are transitive. That is, if some f(n) = Ω(g(n)), and g(n) = Ω(h(n)), then it is also true that f(n) = Ω(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 g1, g2, . . . , g12 satisfies the ordering constraint g1 = Ω(g2), g2 = Ω(g3), . . . , g11 = Ω(g12).*

n	n^2	$(\sqrt{2})^{log(n)}$	$(2)^{log*n}$	$n!$	$(log(n))!$	$(\frac{3}{2})^n$	$(n^{1/log(n)})$	$nlog(n)$	$log(n!)$	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) = Θ(g(n)).

SOLUTION:

1	$n^{1/log(n)}$	2^{log*n}	$(\sqrt{2})^{log(n)}$	$log(n)!$	n	$log(n!)$	$nlog(n)$	n^2	$(\frac{3}{2})^n$	e^n	$n!$

Functions that grow at the same rate:
Both f(n) = 1 and f(n) = $n^{1/log(n)}$ are constant.