Algorithms Tutorial Solutions

15.5 Recursive versus Iterative Algorithms

a) Foo(n) computes the nth Fibonacci number.

b) $O(n)$. We have a for-loop which does a constant amount of work $O(n)$ times; everything else in the program just adds an additional constant amount of work.

c)
```
RecursiveFoo(n: non-negative integer)
    if n=0 or n=1
        return n
    else
        return RecursiveFoo(n-1) + RecursiveFoo(n-2)
```

This algorithm just follows the (recursive) definition of Fibonacci exactly - to compute the nth Fibonacci number, it just computes and then adds together the $(n-1)$th and $(n-2)$th.

d) We've established Foo runs in linear time; meanwhile RecursiveFoo is exponential time with respect to n. We can write a recurrence for RecursiveFoo's runtime: $T(0) = T(1) = c$, $T(n) = T(n-1) + T(n-2) + d$. Computing the closed form for that recurrence is outside the scope of this class, but it's definitely exponential - one way to see that is to first bound it below by a similar recurrence where $T(n) = 2T(n-2) + d$ instead.

15.3 Mystery Code II

a) crunch computes how many nonnegative numbers are in the array.

b) $T(1) = d$
$T(n) = 2T(\frac{n}{2}) + c$

c) Answer: $\Theta(n)$.

Justification using unrolling:

- $T(n) = 2T(\frac{n}{2}) + c$
- $T(n) = 2[2T(\frac{n}{2^2}) + c] + c = 2^2T(\frac{n}{2^2}) + 2c + c$
- $T(n) = 2^2[2T(\frac{n}{2^3}) + c] + 2c + c = 2^3T(\frac{n}{2^3}) + 2^2c + 2c + c$

Based on the above, we predict the general form is that for any k,

$$T(n) = 2^kT(\frac{n}{2^k}) + \sum_{i=0}^{k-1} 2^i c = 2^kT(\frac{n}{2^k}) + c(2^k - 1)$$

.

When we choose k such that $2^k = n$, this becomes $nT(1) + c(n-1) = dn + cn - c$, which is $\Theta(n)$.

Alternate somewhat handwavy justification using recursion trees:

The 'extra work' term is constant, so we just have to count the number of nodes in the tree. And for a full complete k-ary tree, the number of nodes is proportional to the number of leaves; we can ignore the proportionality constant so we only need to count the number of leaves. The height of the tree is $\log(n)$ and the branching factor is 2, so there are n leaves.

15.4 Mystery Code III

a)
```
FindPeak(-1,3,6,7,0):
- skip several false ifs
- set k=3
- skip line 8's if
- line 10: since 6<7, we return FindPeak(7,0)+3

FindPeak(7,0):
- line 3: since 7>0, we return 1

Thus the original call returns 1+3=4
```

And the peak is indeed at position 4 (starting from that 7, the array strictly decreases in both directions until its ends)

b) **3**. If n were 1, we would have returned on line 1. If n were 2, we would return on either line 4 or line 6 (because the first item is either greater than or less than the second/last). However on an input array with 3 elements whose peak is in the center, like $[5, 6, 4]$, we can reach line 7. *(Note that to argue that 3 is the smallest, we had to argue* both *that 3 works* and *that no smaller number works.)*

c) $T(1) = T(2) = c$
$T(n) = T(n/2) + d$

d) $\Theta(\log(n))$. We find this by unrolling: $T(n) = T(n/2)+d = T(n/2^2)+2d = T(n/2^3)+3d = \cdots = T(n/2^k) + kd = T(n/2^{\log(n)}) + \log(n)d = c + \log(n)d$

15.2 Mystery Code I

a) maxthree computes the largest sum of 3 numbers in the list. (Equivalently, it computes the sum of the largest 3 numbers.) *(Note: this is a spectacularly inefficient way to compute this result. You could easily do it in linear time, but as we'll see below this method is at least factorial-time.)*

b) $T(3) = c$
$T(n) = nT(n-1) + dn$
The for loop runs n times, and each time it does $T(n-1)+d$ work: one recursive call, and then various constant-time operations (incrementing loop variable, removing nth element, etc). *(There is also some constant-time work done outside the loop, but don't write e.g. $dn+f$ as your extra work term - non-dominant terms don't make a difference to the big-O analysis so it'll just make things more complicated without changing the final result.)*

c) $\frac{n!}{3!}$. (The last level of the recursion tree is when the input size equals 3, so the number of leaves is $n \cdot (n-1) \cdot (n-2) \cdots 5 \cdot 4 = \frac{n!}{3!}$)

d) There are $\Theta(n!)$ leaves, so even if we ignore the rest of the tree, $2^n \ll n!$ so the algorithm definitely takes more than $O(2^n)$ time.