Lecture 3

Recursion: *Recursion* in computer science solving a problem by solving simpler instantiations of the same problem. As a classic example, consider the *Fibonacci* sequence $1, 1, 3, 5, 8, 13, \ldots$. This sequence is defined by the 0th and 1st Fibonacci numbers both being 1, and subsequent Fibonacci numbers being the sum of the previous two.

That is, if F_i represents the ith Fibonacci number,

$$F_i = \begin{cases} 1 & \text{if } i = 0 \text{ or } i = 1 \\ F_{i-1} + F_{i-2} & \text{otherwise} \end{cases}$$

Now here is an example of using recursion to calculate the nth Fibonacci number. Note that the `fibonacci` function calls itself on smaller, i.e. simpler, inputs.

```
def fibonacci(n):
    if n<2:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Let's give another example. Suppose our friend Bob has a number between 1 and 100 and isn't telling us what it is. However, he's implemented a `bool` function `isGreaterThan`(n) for us which returns `True` if his number is greater than n and `False` otherwise. Now, let's implement a function `searchForBobsNumber`().

```
def searchForBobsNumber():
    x = 1
    while isGreaterThan(x):
        x += 1
    return x
```

We could also implement this function using a recursive helper function.

```
# We assume the answer is in the range [a,b]
def searchForBobsNumberHelper(a, b):
    if isGreaterThan(a):
        return searchForBobsNumberHelper(a+1, b)
    else:
        return a

def searchForBobsNumber():
    return searchForBobsNumber(1, 100)
```

Binary Search: If Bob's number is 100, potentially the `while` loop in the above implementation would run for 100 steps before finding Bob's number. The implementation would be even slower if Bob's number could be in the range from 1 to one billion, and unbearably slow if in the range from 1 to one trillion. One way of remedying this is using a technique known as *binary search*. Suppose Bob's number is between 1 and m and we first ask whether his number is greater than $\lfloor m/2 \rfloor$. Then, no matter what his number is, we will eliminate roughly half of the possibilities, and we can recursively search in the range that is left. That is, consider the following faster version of `searchForBobsNumberHelper(a,b)` (remember that `int` division in Python rounds down to the nearest `int`):

```
# returns True if n equals Bob's number, and False otherwise
def isEqualTo(n):
    return isGreaterThan(n-1) and not isGreaterThan(n)

# We assume the answer is in the range [a,b]
def searchForBobsNumberHelper(a, b):
    mid = (a+b)/2
    if isEqualTo(mid):
        return mid
    elif isGreaterThan(mid):
        return searchForBobsNumberHelper(mid+1, b)
    else:
        return searchForBobsNumberHelper(a, mid)
```

Suppose Bob's number can be in the range $[1, m]$, and for simplicity, let's assume m is a power of 2 (if it isn't, we'll just pretend his number can be in the range $[1, q]$ where q is the smallest power of 2 greater than or equal to m). Since we cut down the number of possibilities in half each time, and we will get the answer right away if there is only one possibility left, the number of times we actually need to ask Bob if his number is greater is the smallest number k such that $m/2^k = 1$, which gives $k = \log_2 m$.

Compare this with the `while` loop implementation in the previous section, which could take m steps. This is a big difference for large m; note that the log base 2 of one trillion is only about 40. Many of you are probably familiar with the term "Gigahertz" corresponding to the speed of the CPU in your computer. This, roughly, corresponds to the number of instructions your computer can execute in one second, with 1GHz meaning one billion instructions per second (this isn't *quite* what it means, but it's a good enough approximation for this discussion). Thus, the `while` loop implementation for m being one trillion on a 1GHz machine would take at least 1000 seconds, which is a bit over 16 minutes. In fact, it would take even more time since some instructions take longer, and some steps which are just one line in Python actually correspond to multiple instructions when you translate to machine language. Meanwhile, the binary search implementation would execute almost instantaneously. From now on in the class, this will be our focus when studying *algorithms*: finding ways of implementing functions such that they run quickly. Note that the `while` loop and binary search implementations in this section both give the right answer, but the latter is vastly superior when it comes to running time.