Fibonacci numbers

Last lecture we saw several examples of recursion. In some of these, recursion did not have any advantage over iteration, in terms of the number of basic operations carried out. For example, although `countdown` and `factorial` could be written recursively, there was nothing gained by doing so. Indeed, given that recursion involves extra bookkeeping[1] one could argue that recursion should be avoided for those examples.

We next look at an example which a recursive solution performs far *more* operations than an iterative solution (and hence the recursive solution should be avoided!) The example is the Fibonnacci sequence: 1,1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = F(n-1) + F(n-2),$$

where $F(1) = 1, F(2) = 1$.

The recursive algorithm is as follows:

```
Algorithm:  Fib(n)    //   assume n > 0
//  Input:  the index of the Fibonacci number to be computed
//  Output:  F(n)
//
   if (n == 1) || (n == 2)
      return 1
   else
      return Fib(n-1) + Fib(n-2)
```

The trouble with this algorithm is that you end up calling the Fib on the same parameters many times. For example, suppose you are asked to compute $F(249)$. Fib(249) calls Fib(248) and Fib(247), Fib(248) calls Fib(247) and Fib(246). Notice immediately that Fib(247) is computed more than once. Similar redunancies occur each step of the way until you reach Fib(1) and F(2).

A much more efficient algorithm would be:

```
Algorithm:  Fib(n)
   if (n == 1) || (n == 2)
      return 1
   else{
      fibtmp[1] = 1
      fibtmp[2] = 1
      for i = 3 to n
         fibtmp[i] = fibtmp[i-1] + fibtmp[n-2]
      return fibtmp[n]
   }
```

ASIDE: This latter method is example of an technique called *dynamic programming*. Dynamic programming is an alternative to recursion. Like recursion, it solves a problem by reducing it to a set of simpler problems and then combines the solutions. The key to dynamic programming, though, is that it keeps track of the smaller problem solutions (*e.g. storing the results in an array* so that it needs to solve each of them only once.

[1] see discussion of stack frames a.k.a. the Java method stack at the beginning of lecture 10

Binary search in a sorted array

Let's next consider a few more examples for which recursion *is* an appropriate solution technique, and now provides a significantly *faster* method than would an iterative technique. (We saw an example of such a recursive method last class – the `power` method.)

Suppose we have an array of elements which are already sorted, say an array of numbers which are sorted from largest to smallest. Now we would like to search for a particular value and return the location of that value. If that value is not found, then it should return the index -1.

One way to do this would be to iteratively (non-recursively) check each of the values in the array from an index `i = 0 to N-1`, *e.g.* using say a `while` loop. After each item is checked, we would increment an index. Notice that this method takes anywhere from 0 to `N` comparisons and increments of `i`, depending on where the desired value is. Later we will refer to this iterative method as "linear search".

Let's now consider a recursive algorithm:

```
Algorithm:  Binary Search

//   inputs:
//       -  an array a[0,.. N-1] of numbers in decreasing order
//       -  a "search key"  val,  i.e. the number we're searching for
//       -  indices i_low, i_high  where i_low <= i_high

//   output:
//         index i,  such that a[i] == val (if such an index exists)
//        -1,   if val  is not an element in the array

binarySearch(a,  val,  i_low, i_high){

   if (i_low == i_high){
      if ( a[ i_low ] == val )
        return i_low;     // which is equal to i_high
      else
        return -1;
      }
   else {
      mid = (i_low + i_high)/2;     //  i_low <=  mid  <  i_high
      if ( val <= a[mid] )
         return binarySearch( a, val,  i_low, mid );
      else
         return binarySearch( a, val,  mid+1, i_high );
   }
}
```

How many times is the recursion called? Consider an input array of size $N = 2^M$, i.e. we are assuming the array size is a power of 2. What happens each time the recursion is called? For the recursion to be called, the condition in the `if` statement must be false, i.e. `(i_low == i_high) == false`. Then there are a few basic operations performed – a test for the condition just mentioned,

an addition, a division, and an assignment, then another comparision. Let's say the number of these basic operations is c.

How many times is the recursion called? Each time the recursion is called, the number of elements in the array that need to be examined is cut in half. The first recursion call examines 2^{M-1} elements, and the second examines 2^{M-2} elements, etc. In order for `binarySearch` to return a value, we need `i_low == i_high`, and this happens after `M` $= \log_2$ `N` recursive call.

Since there are c operations within each call, and there are $\log_2$`N` calls, it follows that the total number of operations is about $c_1 \log_2$ `N`. I say "about" because we need to account for the operations that are performed the one time that the `i_low == i_high` condition is met.

Note that $\log_2$ `N` is very small. When `N = 1000`, $\log_2$ `N` ≈ 10, and when `N = 1,000,000`, $\log_2$ `N` ≈ 20. You should be able to appreciate that for large `N`, the binary search method (recursive) will be much much faster than the "linear search" method where we iteratively scan the array until we find the item we want (or conclude that it is not there). That is, the binary search method takes some small constant time $\log_2$ `N` operations, whereas the "linear search" method takes (on average) some constant times `N` operations.

Merge-sort

The binary search method assumes the array is already sorted. Let's next return to the problem of how we should sort the array in the first place.

I spent the last 15 minutes of the lecture introducing the "MergeSort" algorithm. This sorting algorithm is much faster than the insertion sort algorithm, which we discussed in an earlier lecture. (I will argue why next week.) The MergeSort is explained well in Chapter 11.1 of the GT textbook, so I won't repeat the discussion here. *You are responsible for reading pp. 488-494 of the textbook.* I will return to the remainder of this section in the coming lectures, when I analyse the performance of "MergeSort".