Finding the nth number in the Fibonacci sequence is an issue which lends itself to a solution that uses dynamic programming. This post will briefly explain the Fibonacci sequence, a recursive solution, and then an improvement on that solution which uses dynamic programming.

**The Fibonacci Sequence**

**Definition**:  the next number = sum of previous two numbers.

**The sequence**:  [1, 1, 2, 3, 5, 8, ... , ((n-1) + (n-2))]

**The Recursive Solution**

fibonacci( int i ) {  
  
  if ( i = 1 or i = 2)  
  
    out = 1  
  
  else  
  
    out = fibonacci( i - 1 ) + fibonacci( i - 2 )

  return out  
  
}  
This will work, but the growth rate is quite large at O(2^n).  If i is 6, then we recursively call for 5 and 4 in the first run. Calling 5 will call 4 and 3. Each call to 4 will call 3 and 2, etc... So many of the calls made to fibonacci are repeated. For example, when we call fibonacci(5), it will call fibonacci(4) which will have already been called in the first run by fibonacci(6). The further down we go, the more repeat calls we make.

Enter Dynamic Programming with memoization to improve this function to a level that is actually usable for large inputs.

**The Dynamic Programming Solution**

Note that this solution requires an outside array. This could be implemented with some other helper function.

int[] arr = new int[ i + 1 ]

fibonacci( int i, int[] arr) {

  if ( arr[ i ] is not NULL )

    return arr[ i ]

  if ( i = 1 or i = 2 )

    out = 1

  else

    out = fibonacci( i - 1, arr ) + fibonacci( i - 2, arr )

  arr[ i ] = out

  return out

}

So this solution saves each unique call to arr which saves a considerable amount of repeat calls from happening. The only time fibonacci(i, arr) is called recursively is if fibonacci has not been calculated for i in any of the previous recursive calls.

Example run:

int[] A= new int[7]

f(6, A):  A[6] = out = f(5, A) + f(4, A)

- f(5, A): A[5] = out = f(4, A) + f(3, A)

- - f(4, A):  A[4] = out = f(3, A) + 1 <------- (calls to 2 or 1 = 1)

- - - f(3, A):  A[3] = out = 1 + 1 = 2

- - f(3, A): out = A[3] <----------- this is defined and so no recursive call needed

- f(4, A):  out = A[4] which is already defined.. .no recursive call

I of course did not show calls for 1 or 2, but I think we can agree as friends that A[2] and A[1] are set the first time those are called and then 1 is returned thereafter. The run time for the much improved algorithm is O(n) as opposed to O(2^n)! If you write out a binary tree of function calls for the two algorithms, then the run time difference becomes apparent quickly.