So far in class, the problems which we have solved with a divide-and-conquer algorithm have had the property that a subproblem we recursively generate will never be generated more than once.

- For example in Merge-Sort($A[1..n]$), we recursively call Merge-Sort($A[1..n/2]$) and Merge-Sort($A[n/2+1..n]$). At no other point in time of the algorithm will we again make one of these calls.

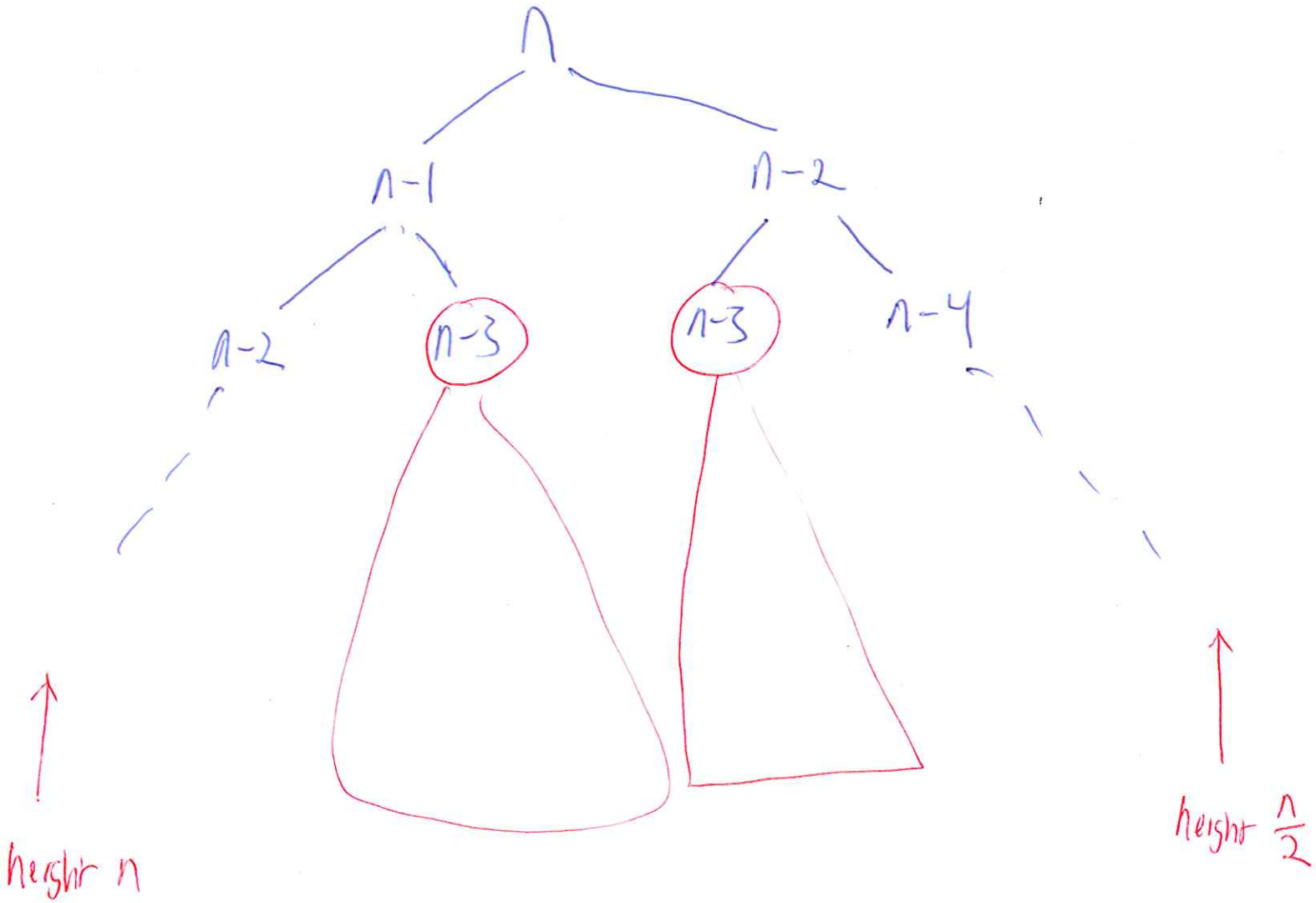
There are some other problems which have natural recursive definitions for which this is not the case.

For example the *Fibonacci Sequence*:

- 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- $\mathrm{F}(1) = 1$, $\mathrm{F}(2) = 1$, $\mathrm{F}(n) = \mathrm{F}(n-1) + \mathrm{F}(n-2)$ for $n \geq 3$.

Recursion tree for computing the nth Fibonacci number $F(n)$:

- $\mathrm{F}(1) = 1$, $\mathrm{F}(2) = 1$, $\mathrm{F}(n) = \mathrm{F}(n-1) + \mathrm{F}(n-2)$ for $n \geq 3$.



$2^{n/2} \leq T(n) \leq 2^n \Rightarrow T(n) = \Theta(2^n)$

The running time of the straight-forward recursive algorithm for computing $\mathrm{F}(n)$ takes $O(2^n)$ time, but we compute the same subproblems multiple times.

Idea: check to see if we have already solved a subproblem. If we have not solved it yet, then do so now and store the value of the subproblem in an array. If we have already solved it, then the value is stored in the array and we can return the value of the subproblem in $O(1)$ time rather than making another recursive call.

This technique is called **memoization** and is a variant of **dynamic programming**.

Memoization algorithm for the Fibonacci sequence. F is an array of size n with each element initialized to null.

```
fibMemoizationRec(n, F){
    if(F[n] == null){
        if(n==1) F[n] = 1
        if(n==2) F[n] = 1
        F[n] = fibMemoizationRec(n-1, F) +
               fibMemoizationRec(n-2, F)
    }
    return F[n]
}
```

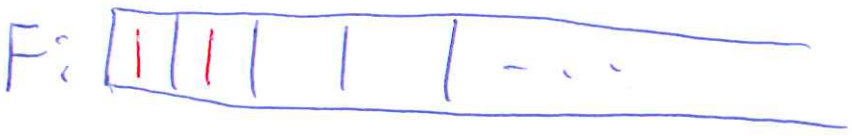
$\Theta(n)$ time + space

Memoization is a "top-down" approach in which we start at the top of the recursion tree as we have been doing and work our way down the tree as is done in a regular recursive algorithm.

An alternative is a "bottom-up" approach in which we first start by computing the smallest subproblems (i.e. the base case) and then compute the larger subproblems based off of the smaller subproblems which we already solved.

This is typically done with an iterative algorithm (using loops instead of recursion). This is the other variant of dynamic programming.

A bottom-up DP algorithm for the Fibonacci sequence.



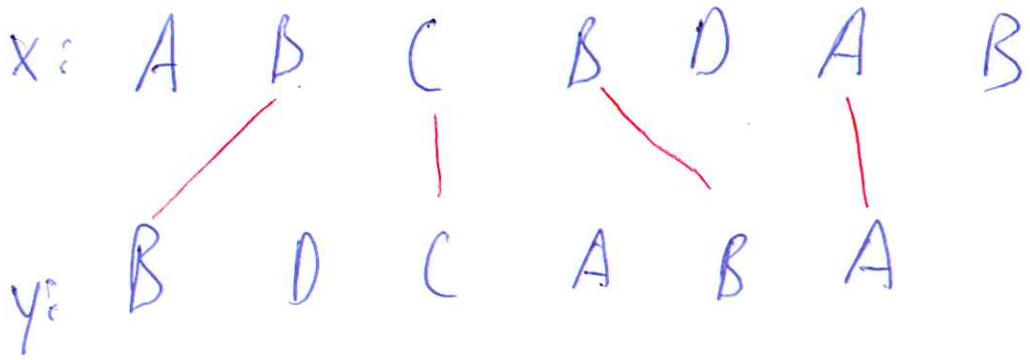

```
fib BottomUpDP(n) {
    F[1] = 1
    F[2] = 1
    for (i = 3 to n)
        F[i] = F[i-1] + F[i-2]
    Return F[n]
}
```

Time = $\Theta(n)$

Space = $\Theta(n)$

Another example: *Longest Common Subsequence* (LCS)

- Given two sequences of characters $x[1..m]$ and $y[1..n]$, find a longest subsequence common to them both (note that a subsequence and a substring are different).



$LCS(x, y) = B\ C\ B\ A$

Consider a brute-force algorithm for computing LCS.

- Check every subsequence of $x[1..m]$ and check to see if it is also a subsequence of $y[1..n]$.

How many subsequences does x contain?

- 2^m (each 0/1 vector of length m determines a distinct subsequence of x).

This algorithm would have exponential running time.

We will show that this problem can be solved by a bottom-up dynamic programming algorithm.

- Our approach will first compute the *length* of a LCS.
- We will then show how to extend the algorithm to find the LCS itself.

We denote the length of a sequence s by $|s|$.

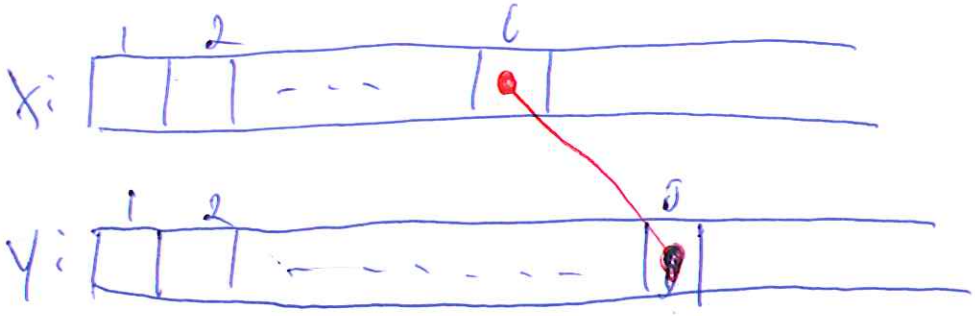
Idea: Consider *prefixes* of x and y.

- Define $c[i, j] = |LCS(x[1..i], y[1..j])|$ (the LCS of the first i characters of x and the first j characters of y).
- We thus have $c[m, n] = |LCS(x, y)|$.

Theorem

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

Proof of case when $x[i] = y[j]$



Let $Z[1 \ldots k] = LCS(x[1 \ldots i], y[1 \ldots j])$, where $c[i,j] = k$. Then, $Z[k] = x[i]$ or else we could extend Z, contradicting $Z = LCS(x[1 \ldots i], y[1 \ldots j])$.

This implies $Z[1 \ldots k-1] = LCS(x[1 \ldots i-1], y[1 \ldots j-1])$.

For the sake of contradiction, suppose w is a longer CS of $x[1..i-1]$ and $y[1..j-1]$. $|w| \geq k$.

But then $w \| x[i]$ is a CS of $x[1..i]$ and $y[1..j]$ with length $\geq k+1$. But this contradicts the assumption that $|LCS(x[1..i], y[1..j])| = k$.

($\|$: concatenation)

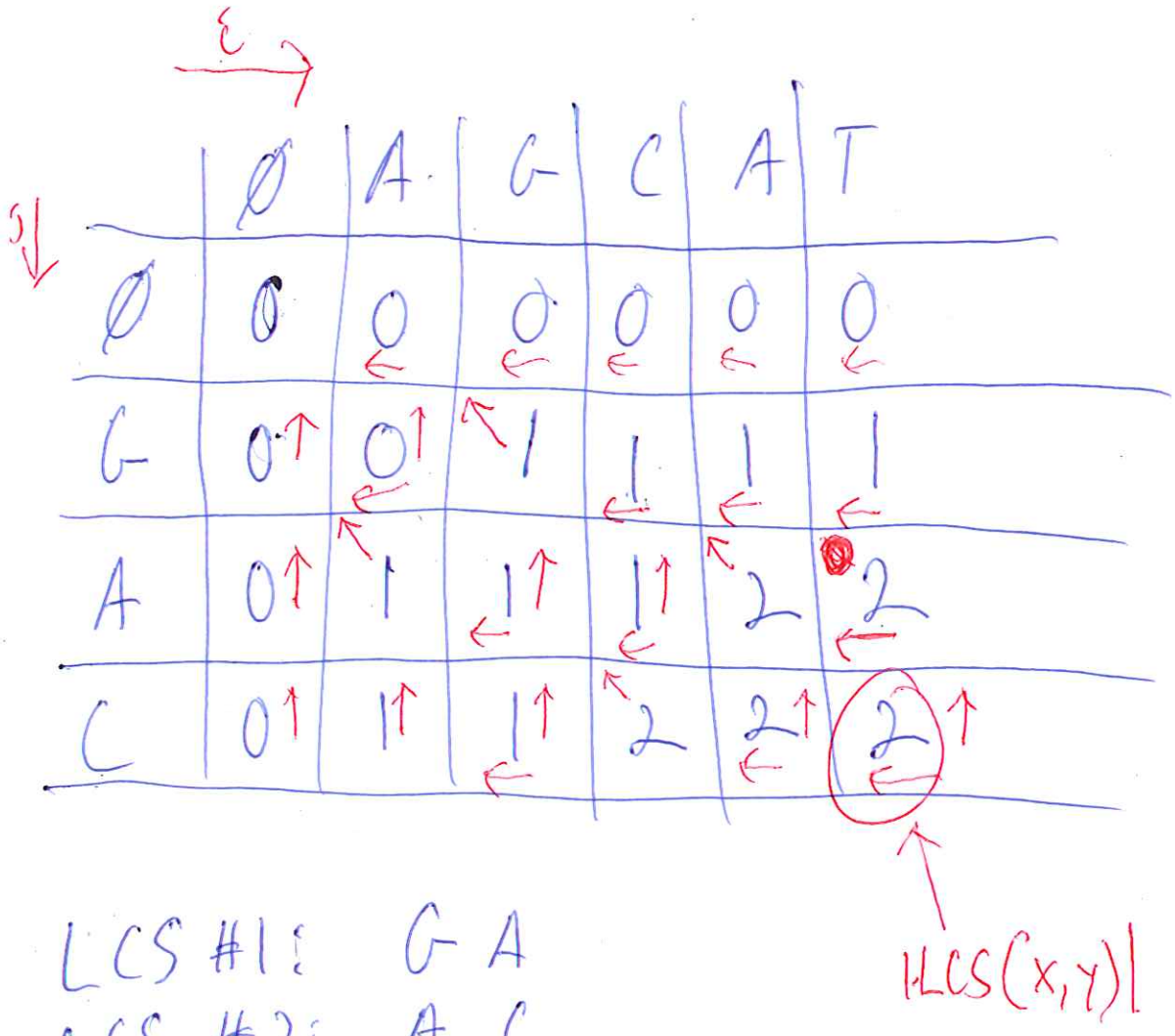
This implies $C[i,j] = C[i-1, j-1] + 1$.

Other cases are proved similarly.

$$C[i,j] = \begin{cases} C[i-1, j-1] + 1 & \text{if } x[i] = y[j] \\ \max\{C[i-1,j], C[i,j-1]\} & \text{o.w.} \end{cases}$$

X: A G C A T

Y: G A C



LCS #1: G A

LCS #2: A C

LCS #3: G C