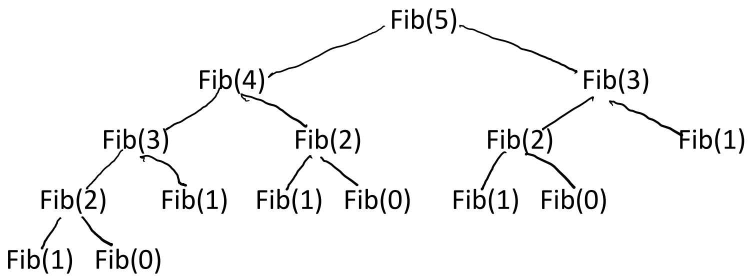

height n

how many calls?

Referential Transparency (same input -> same output) means we can memoize!!

```
Algorithm isValidAvl(T)
        for each p in T.positions() do
                setHeight(p, -1)    // initialize the attribute in the tree to special value
        height(T, T.root())
        res <- isValidAvlHelper(T, T.root())
1       return res
```

```
Algorithm isValidAvlHelper(T, p)
n       if T.isExternal(p) then
n               return true
n       lRes <- isValidAvlHelper(T, T.leftChild(p))
n       rRes <- isValidAvlHelper(T, T.rightChild(p))
n       lh <- height(T, T.leftChild(p))
n       rh <- height(T, T.rightChild(p))
n       pRes <- (abs(lh-rh) <= 1)
n       return lRes /\ rRes /\ pRes
```

```
Algorithm height(T, p)  // memoized version of height (simplifies code of isValidAvlHelper)
        if T.isExternal(p)) then
                return 0
        h <- getHeight(p)
        if h < 0 then  // has the height already been computed for internal node
                lh <- height(T, T.leftChild(p))
                rh <- height(T, T.rightChild(p))
                h <- max(lh, rh) + 1
                setHeight(p, h)
        return h
```

abc

_

a, b, c
ab, ac, bc
abc
2^3

cababc

abdbc

longest common subsequence is based on aligning the two strings so the maximum number of characters match

c a b a b c .
. a b d . c b   LCS length = 3

c a b a . b c
. a b d c b .   LCS length = 3

c a b a b c
. a b d c b   LCS length = 2

$$L_{i,j} = MAX \{ L_{i-1,\ j-1} + S(a_i,b_j),\ \ L_{i,\ j-1} + 0,\ \ L_{i-1,\ j} + 0 \ \}$$

Longest Common Sequence

		0	1	2	3	4	5	6	7	8	9	10	11
		___	G	A	A	T	T	C	A	G	T	T	A
0	___	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	1	1	1	1	1	1	1	1	1	1	1
2	G	0	1	1	1	1	1	1	1	2	2	2	2
3	A	0	1	2	2	2	2	2	2	2	2	2	3
4	T	0	1	2	2	3	3	3	3	3	3	3	3
5	C	0	1	2	2	3	3	4	4	4	4	4	4
6	G	0	1	2	2	3	3	4	4	5	5	5	5
7	A	0	1	2	3	3	3	4	5	5	5	5	6

A T T G A C T T A A . G
A . . G . C . T . A G G

G . A A T T C A G T T A
G G A . T . C . G . . A

$$L_{i,j} = \text{MAX} \{ L_{i-1,\, j-1} + S(a_i,b_j),\ \ L_{i,\, j-1},\ \ L_{i-1,\, j} \}$$

Non-memoized version:

```
Algorithm LCS(S1, S2):
   Input: Strings S1 and S2
   Output: Length of the LCS of S1 and S2
         m <- S1.size()
         n <- S2.size()
         return LCShelper(S1, S2, m, n)
Algorithm LCShelper(S1, S2, m, n):
   Input: Strings S1 and S2 with at least m and n elements, respectively
   Output: Length of the LCS of S1[1..m] and S2[1..n]
         if n = 0   then
              return 0
         else  if m = 0   then
              return 0
         else  if S1[ m ] = S2[ n ]   then
              return LCShelper (S1, S2, m -1, n -1) + 1
         else
              return MAX ( LCShelper (S1, S2, m, n -1), LCShelper (S1, S2, m -1, n) )
```

Memoized version:

```
Algorithm LCS(S1, S2):
   Input: Strings S1 and S2
   Output: Length of the LCS of S1 and S2
         m <- S1.size()
         n <- S2.size()
         L <- new array[m+1, n+1]  // create and initialize a table
         for i <- 0 to m do
               for j <- 0 to n do
                    L[i,j] <-  -1
         return LCShelper(S1, S2, m, n, L)

Algorithm LCShelper(S1, S2, m, n, L):
   Input: Strings S1 and S2 with at least m and n elements, respectively
   Output: Length of the LCS of S1[1..m] and S2[1..n]
     if L[m,n] < 0  then  // has the LCS already been computed
         if n = 0   then
              L[m,n] <- 0
         else  if m = 0   then
              L[m,n] <- 0
         else  if S1[ m ] = S2[ n ]   then
              L[m,n] <- LCShelper (S1, S2, m -1, n -1) + 1
         else
              L[m,n] <- MAX ( LCShelper (S1, S2, m, n -1), LCShelper (S1, S2, m -1, n) )
      return L[m,n]
```