Dynamic Programming (DP) 11.20.

See Jeff Erickson's Algorithms Ch 3

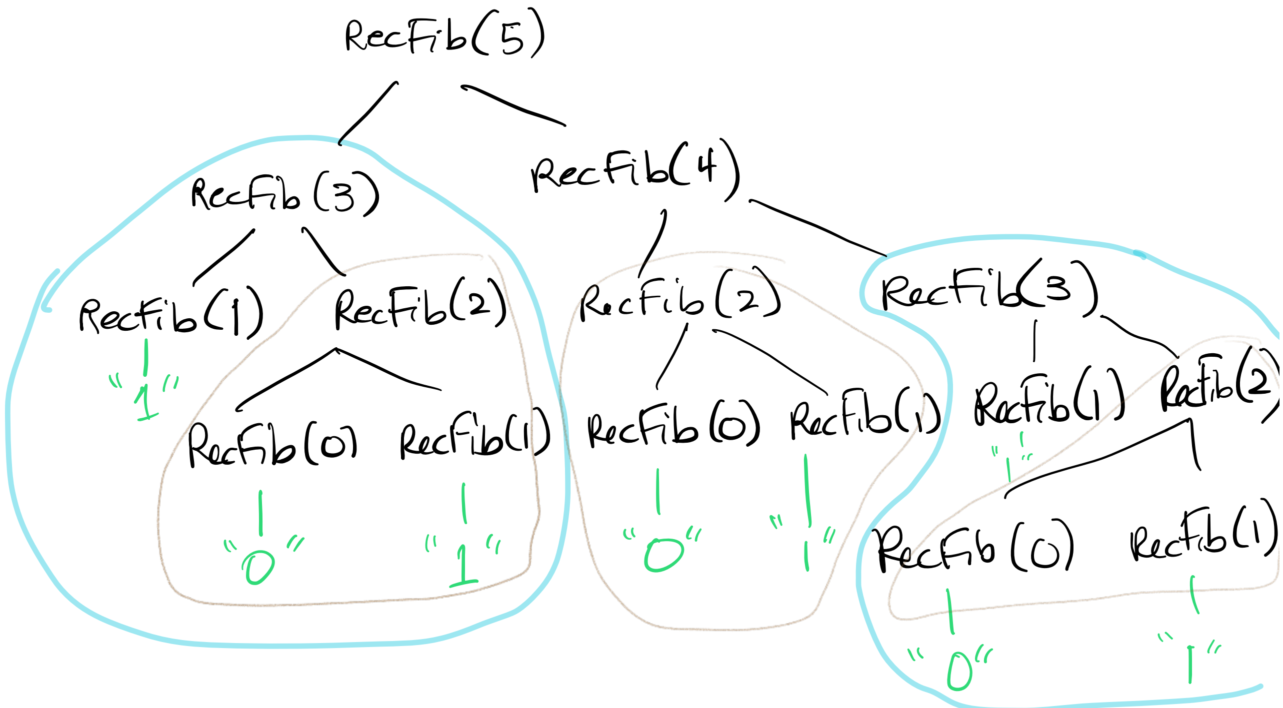
Recall (from 260 intro to DP) that some recursive algorithms may repeatedly compute the same value, to the detriment of the running time:

```
RecFib(n)              // the naive method
    if  n = 0
        return 0
    else if  n = 1
        return 1
    else
        return RecFib(n-1) + RecFib(n-2)
```

Running time of RecFib is $O(F_n)$

(can show $T(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ T(n-1) + T(n-2) + 1 & \text{otherwise,} \end{cases}$

where $T(n)$ = # recursive calls to RecFib on input n.



"Dynamic Programming" = memo-ize data you might use again (don't recompute)

```
MemFib(n)
    if  n=0
         return 0
    else if  n=1
         return 1
    else
         if F[n] is undefined
              F[n] = MemFib(n-1) + MemFib(n-2)
         return F[n]
```

The above is "top down", a good and necessary direction when you don't know what values of F[i] we will need. But for Fibonacci numbers, we will need all of them ... F[2..n]. So the following also works:

```
IterFib(n)
    F[0] = 0;  F[1] = 1
    for i = 2 to n do
        F[i] = F[i-1] + F[i-2]
    return F[n].
```

IterFib uses $O(n)$ additions and stores $O(n)$ integers.

We have seen one example of DP already... the min Contig Sum problem.
We were able to solve that without $O(n)$ storage... just two integers! Can you do the same with IterFib?

DP is not about filling in tables... it is about smart recursion

↑ this can be implemented as iteration.

How to come up with DP solutions

1. Formulate problem recursively
   - specification – describe coherently
   - solution – clear recursive formula
     – smaller instances of exactly same problem

2. Build solutions from bottom up.
   - identify subproblems
   - choose memoizing structure
   - identify dependencies
   - find good evaluation order
   - analyze space & running time.
   - write down algorithm.

Eg. Longest Increasing Subsequence LIS (3.6)

A =

$-\infty$	9	4	18	6	11	13	17	16	19	80	7	21
0	1	2	...									n

Exhaustive search is exponential time.

all sequences that start with 9

all sequences that start with 9, 18

⋮

all sequences that start with 4

all ...

If we know stuff about the LIS that starts at 11, that can help us ascertain solutions starting at 9 and at 4.

$$LIS[i] = 1 + \max_{j>i,\ A[i]<A[j]} \left(LIS[j] \right)$$

$-\infty$ 9 4 18 6 11 13 17 16 19 80 7 21

The recursive formula suggests how to compute it.

int LongestIncreasingSubsequence($A[1..n]$)

// find integer that is length of longest strictly

// increasing subsequence of $A[1..n]$

// (not necessarily contiguous

```
A[0] = -∞
for i = n downto 0
    LIS[i] = 1          // LIS[i] is length of longest subseq
    for j = i+1 to n       // that starts at (includes) A[i]
        if A[j] > A[i] and LIS[j] > LIS[i]
            LIS[i] =

return
```

Running time =

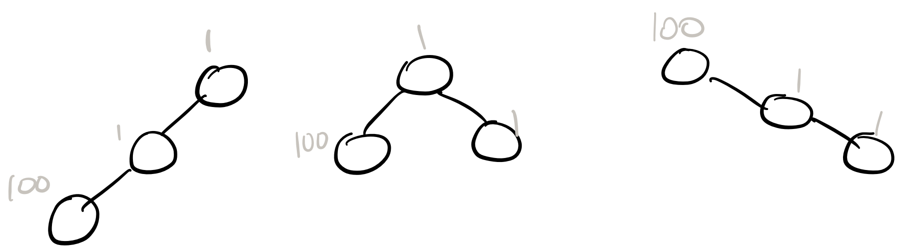
Space =

Optimal BST

index	1	2	3	4	5	6	7	8	9
key:	1	3	8	15	19	22	28	35	44
freq:	60	40	3	19	17	35	19	22	6

We want a binary search tree whose shape is optimal for the frequencies given - ie, makes fewest comparisons in 60 searches for key 1, 40 searches for key 2, etc.

Eg freq = [100, 1, 1]



tree costs under given frequency assumptions

Optimal BST

index	1	2	3	4	5	6	7	8	9	
key:	1	3	8	15	19	22	28	35	44	= A
freq:	60	40	3	19	17	35	19	22	6	

What is the optimal tree cost if A[5] is root?

- useful to know OptCost[1, 4] and OptCost[6, 9]
- for OptCost[6, 9] useful to know OptCost[6, 6] and OptCost[8, 9], . . .

$$\text{OptCost}[i, K] = \begin{cases} 0 & \text{if } i > K \\ \sum_{j=i}^{K} \end{cases}$$

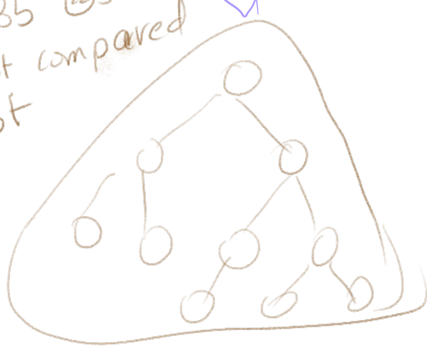
OptCost[4, 6]

19 15's

17 19's

35 22's

all get compared to root



First, let's compute $F[i,k] \overset{\text{defn}}{=} \sum_{j=i}^{k} f[j]$ using DP:

$$F[i,k] = \begin{cases} f[i] & \text{if} \\ F[\quad] + & \text{otherwise} \end{cases}$$

recurrence formula

```
ComputeF(f[1..n])
    for i = 1 to n
        F[i, i-1] = 0
        for k = i to n
            F[i,k] = F[i, k-1] + f[k]
```

i

$F[i]$:

0	0	0	0	0	19	36	71	90	112	118

$O(n-i)$

$f[i]$ 19 17 35 19 22 6

$$\text{OptCost}[i,k] = \begin{cases} 0 & \text{if } i > k \\ F[i,k] + \min\limits_{i \le r \le k} \begin{cases} \text{OptCost}[i, r-1] \\ + \text{OptCost}[r+1, k] \end{cases} \end{cases}$$

```
ComputeOptCost(i, k)                          // assumes OptCost[i, j<k]
    OptCost[i,k] = ∞                          // and OptCost[j>i, k] have
    for r=1 to k                              // been computed already
        tmp = OptCost[i, r-1] + OptCost[r+1, k]   // = smaller ranges.
        if OptCost[i,k] > temp
            OptCost[i,k] = temp
    OptCost[i,k] = OptCost[i,k] + F[i,k]
```

```
OptimalBST(f[1..n])                           // Note: smaller ranges
    ComputeF(f[1..n])                         // are computed first.
    for i=1 to n+1
        OptCost[i, i-1] = 0
    for d=0 to n-1
        for i=1 to n-d
            ComputeOptCost(i, i+d)
    return OptCost[1..n]
```