Dynamic Programming

- called that because Richard Bellman, researcher in 1940's and 1950's, needed a term that would impress the US Secretary of Defense, who did not like "research" or "math"

  "Program" here refers to program (order) of evaluation.

  "Dynamic" means it changes over time (i.e. during execution of the "program" of evaluations.

Example 1:

$$\mathrm{Fib}(0) = \mathrm{Fib}(1) = 1$$

$$\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2), \quad \forall n > 1.$$

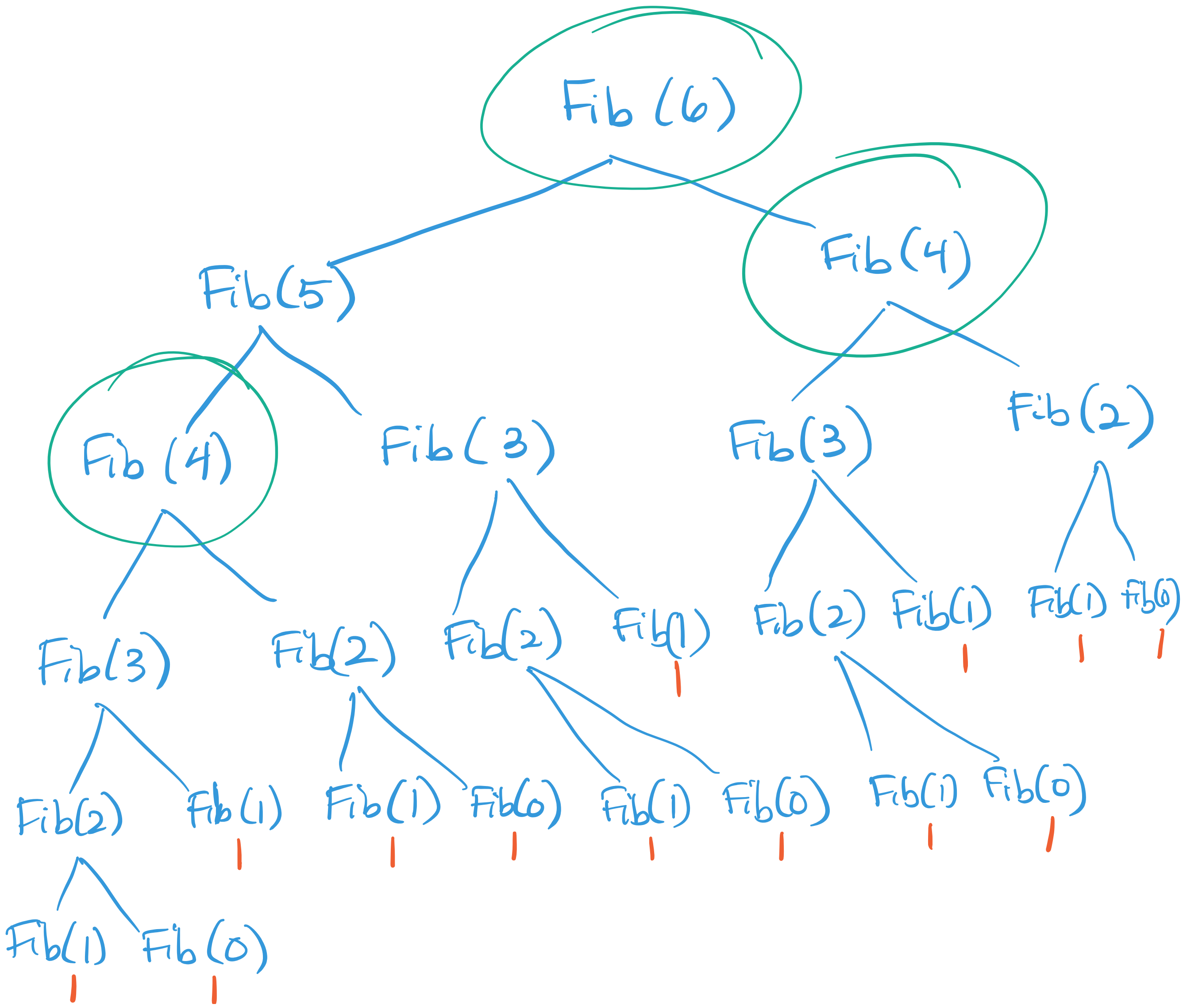
Write a program that computes Fib(n).

1, 1, 2, 3, 5, 8, 13, 21 33 54.

1. Straight forward Recursion

```
Fib (n)
    if n==0 or n==1 return 1
    return Fib(n-1) + Fib(n-2)
```

Evaluation Tree ("calls" tree)



$$T(n) = T(\underset{\nearrow}{n-1}) + T(\underset{\nearrow}{n-2}) + O(1)$$

$$= \underline{2T(n-2)} + T(n-3)$$

$$= \ldots \; 4\,\underline{T(n-4)}$$

$$\ldots \; 8\,T(n-6)$$

$$16\,T(n-8) \; .$$

$$\vdots$$

Tree grows exponentially, as does the running time, as n gets larger.

A better idea...

- Memo-ize the results
- don't recompute, just look up results that have already been computed.

```
F = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | . . . .

Fib2(n)
          0                                       n
    F = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
    F[0] = 1;    F[1] = 1;
    return  fibhelper(F, n)


fibhelper(F, n)

    if  F[n] == 0        /* not yet computed

        F[n] = fibhelper(F, n-1) +
               fibhelper(F, n-2)

    return F[n].
```

/* it is recursive, but always looks to see
if F has already populated F[i] with
an answer — it does not repeat evaluation */

But in this case we know we will need <u>all</u> values of F, F[0] to F[n-1].

In this case, we also know exactly what values we need to compute F[i] (F[i-1] and F[i-2])

So a viable ordering to compute In is:

F[0] ✓, F[1] ✓, F[2] ✓, F[3] ✓, ... F[i] ✓, . . .

```
Fib3(n)
    F = [0, 0, ..., 0]     (indices 0, 1, ..., n)
    F[0]=1 ; F[1]=1
    for i=2 to n
        F[i] = F[i-1] + F[i-2]

    return F[n]
```

Dynamic Programming

... is an algorithmic approach whose correctness is based on a recurrence (such as $F[i] = F[i-1] + F[i-2]$) and where the subproblems have dependencies that are not circular (may be treelike) allowing us an evaluation order to memo-ize in.

DP is smart recursion.

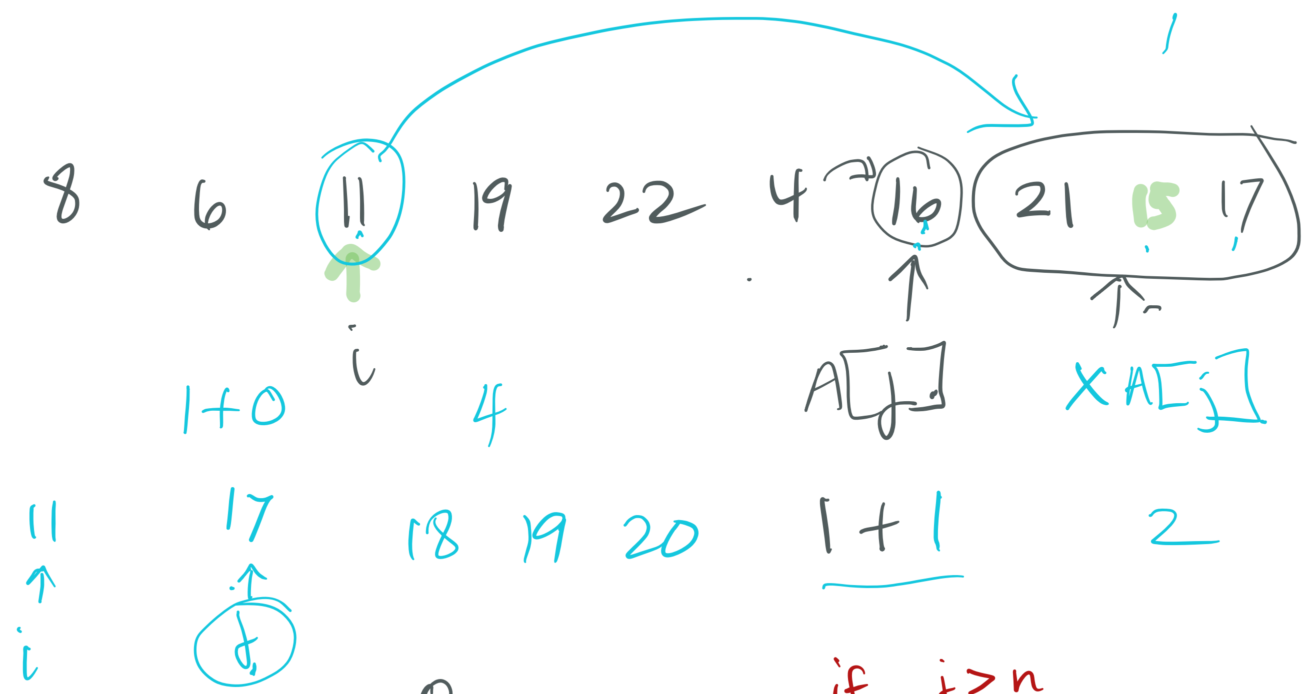
E.g. Longest Increasing Subsequence (LIS)

Input: an array $A[1..n]$ of ints

E.g. $A = [18\ \ \underline{4}\ \ 13\ \ \underline{6}\ \ 2\ \ \underline{9}\ \ 29\ \ \underline{14}]$

Output: max value of K, length of longest subsequence (not nec. contiguous) of ints that appear in increasing order in A.

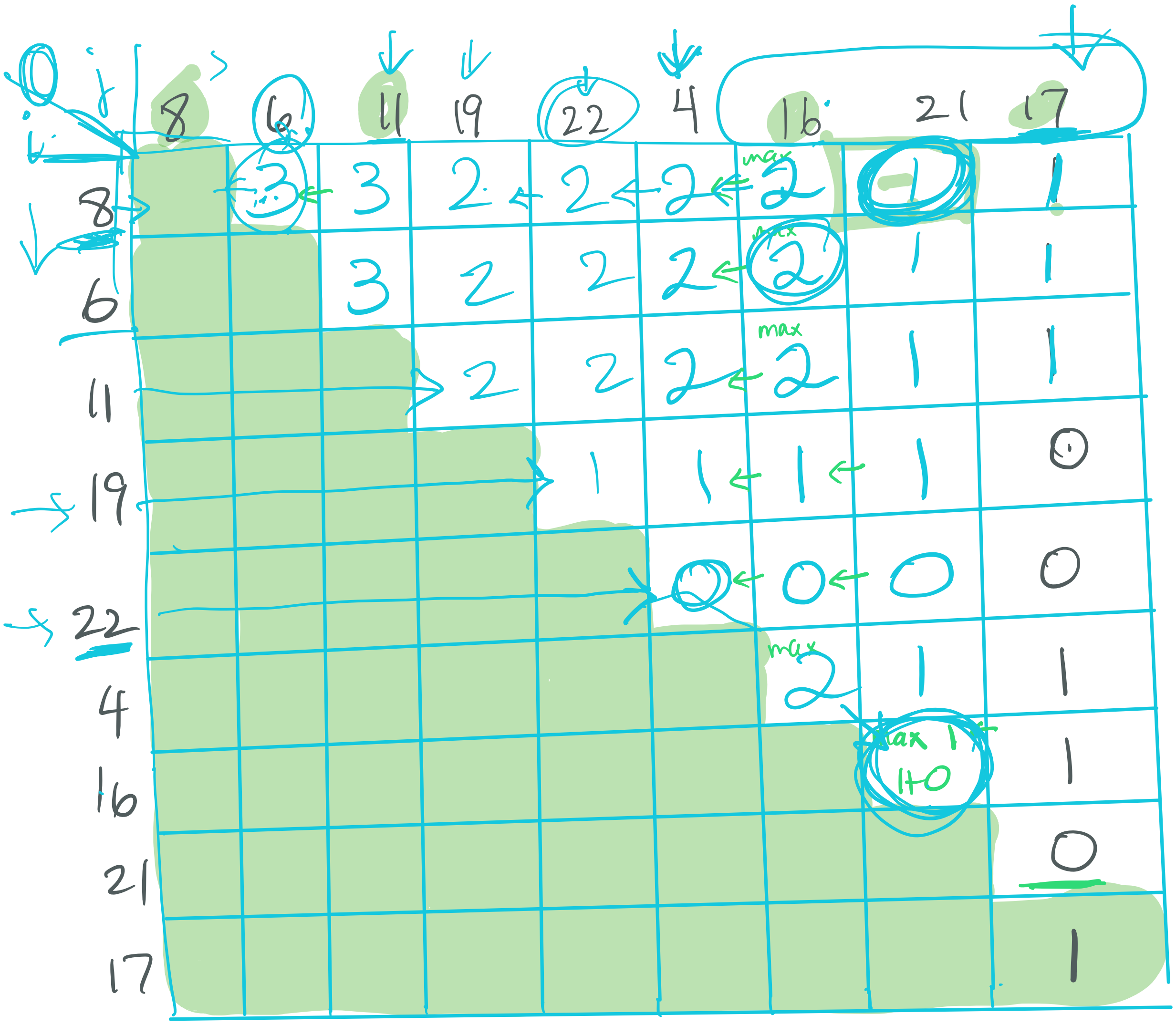
Eg 4   4, 6, 9, 14



$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\begin{cases} LIS(i, j+1) \\ 1 + LIS(j, j+1) \end{cases} & \text{otherwise} \end{cases}$$

A[j] will be in list.

where $LIS(i,j) \overset{\text{def}}{=}$ length of longest increasing Subsequence of $A[j \ldots n]$ where all elements are $\geq A[i]$



	8	6	11	19	22	4	16	21	17
8		3	3	2	2	2	max 2	1	1
6			3	2	2	2	max 2	1	1
11				2	2	2	max 2	1	1
19					1	1	1	1	0
22						0	0	0	0
4							max 2	1	1
16								max 1 1+0	1
21									0
17									1

What order can we fill in the table?

fill in one column at a time.

FastLIS$(A[1..n])$

```
$A[0] = -\infty$
for $i = 0$ to $n$
    $LIS[i, n+1] = 0$
for $j = n$ downto $1$
    for $i = 0$ to $j-1$
        Keep = $1 + LIS[j, j+1]$
        Skip = $LIS[i, j+1]$
        if $A[i] \geq A[j]$
            $LIS[i, j]$ = Skip
        else $LIS[i, j] = \max($Keep, Skip$)$
return $LIS[0, 1]$
```

Longest Common Subsequence

Eg ABBACBC BCBACCB

- in order, but not necessarily contiguous

X \ Y	λ	B	C	B	A	C	C	B
λ	0	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1	1
B	0	1	1	1	1	1	1	2
B	0	1	1	2	2	2	2	2
A	0	1	1	2	3	3	3	3
C	0	1	2	2	3	4	4	4
B	0	1	2	3	3	4	4	5
C	0	1	2	3	3	4	5	5

$$LCS(i,j) = \begin{cases} \max(LCS(i,j-1) \\ \quad LCS(i-1,j)) & \text{if } X[i] \neq Y[j] \\ \max(LCS(i,j-1), \\ \quad LCS(i-1,j), \\ \quad 1 + LCS(i-1,j-1)) & \text{if } X[i] = Y[j] \end{cases}$$