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:

$$Fib(0) = Fib(1) = 1$$

$$Fib(n) = Fib(n-1) + Fib(n-2), \forall n > 1.$$

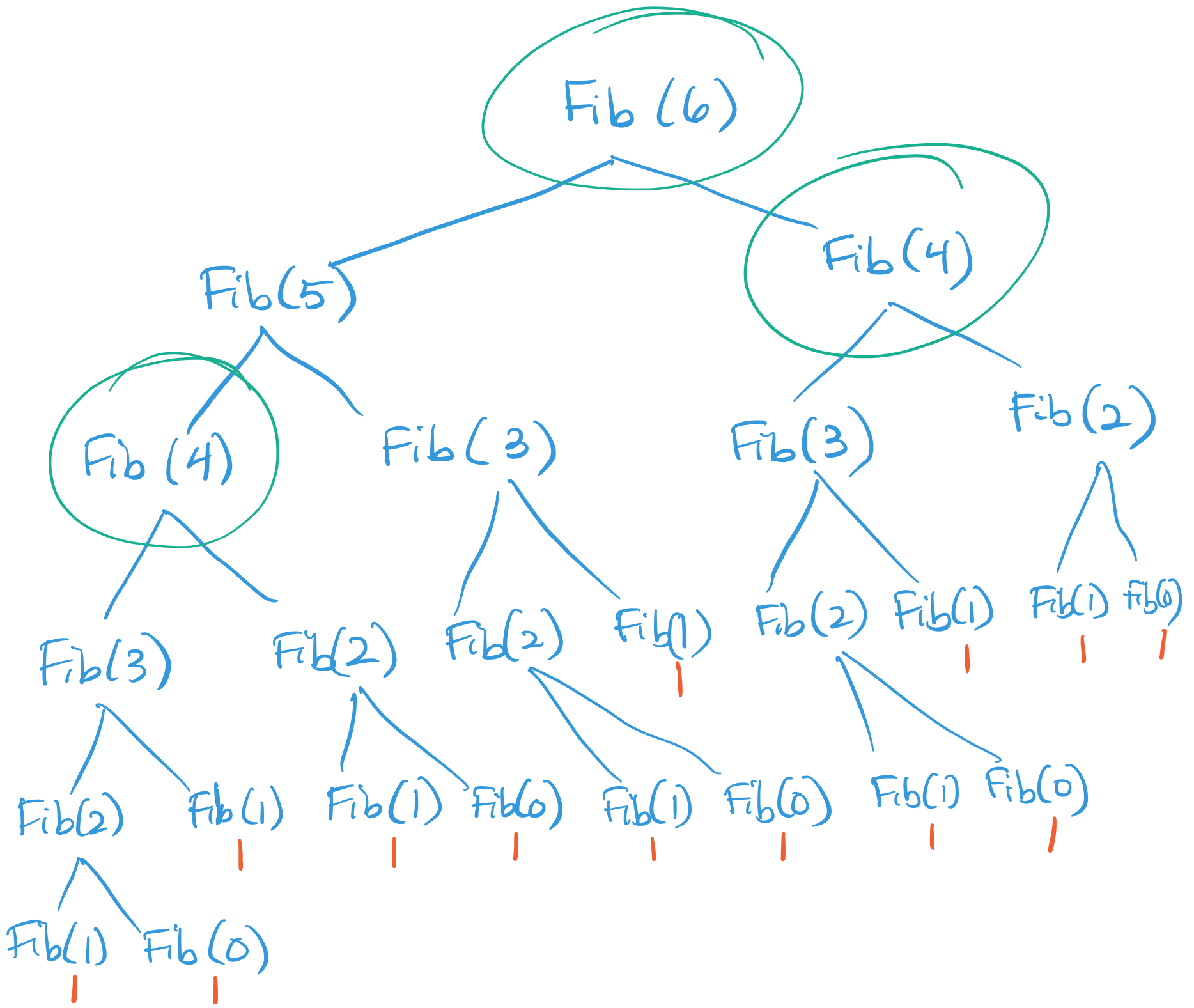
Write a program that computes Fib(n).

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)



$$T(n) = T(n-1) + T(n-2) + O(1)$$

$$= 2T(n-2) + T(n-3)$$

$$= \cdots \quad 4T(n-4)$$

$$\cdots \quad 8T(n-6)$$

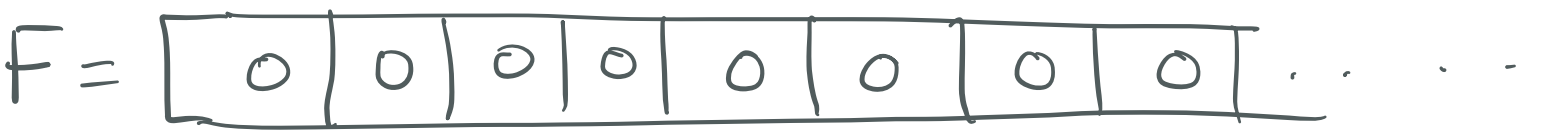
$$16\,T(n-8)$$

$$\vdots$$

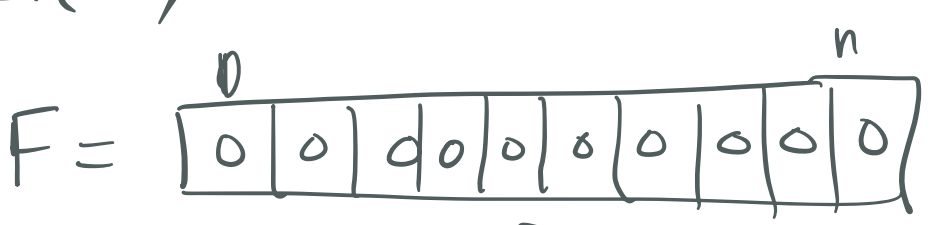
Tree grows exponentially, as does the running time, as n gets larger.

A better idea...

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



Fib2(n)



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 all 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]   (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 tree like) 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. $A = [\ 18\ \ 4\ \ 13\ \ 6\ \ 2\ \ 9\ \ 29\ \ 14\ ]$

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

8 6 11 19 22 4 16 21 15 17

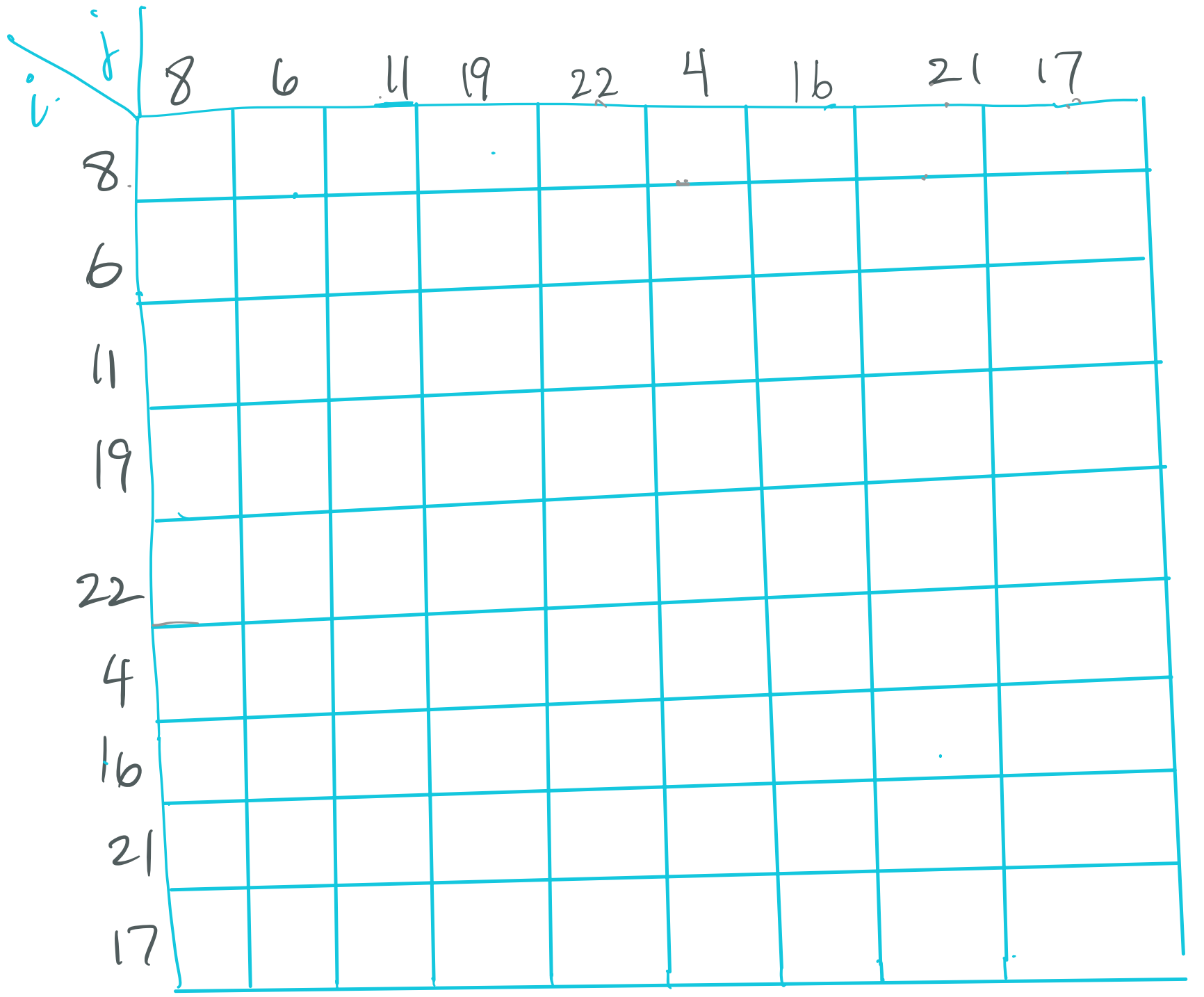
9

LIS[i, 9] = length of LIS from A[9] on, all values are ≥ A[i]

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

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

↑ A[j] will be in list.



What order can we fill in the table?

fill in one column at a time.

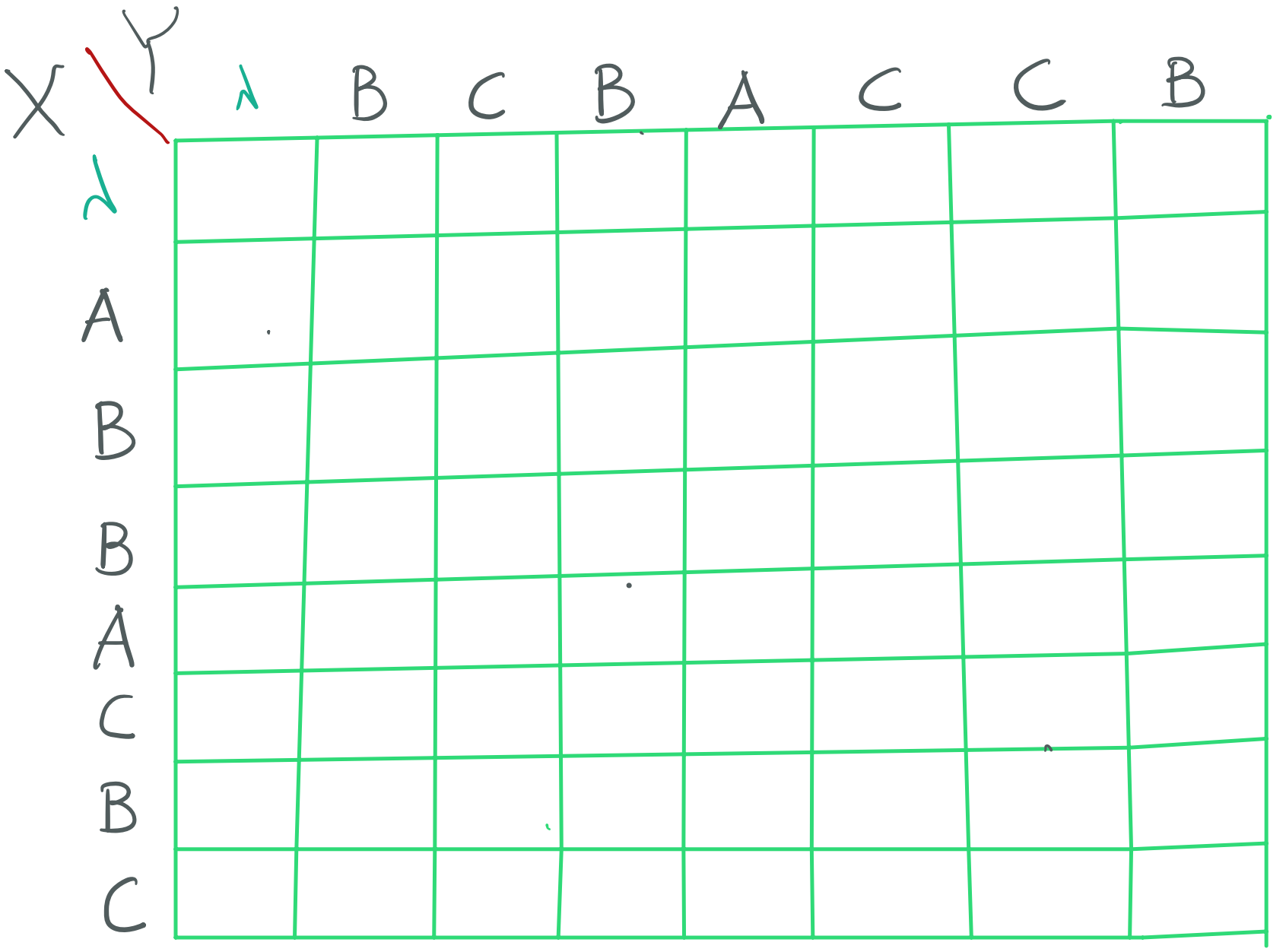
Fast LIS (A[1..n])

```
A[0] = -∞
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] ≥ 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



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