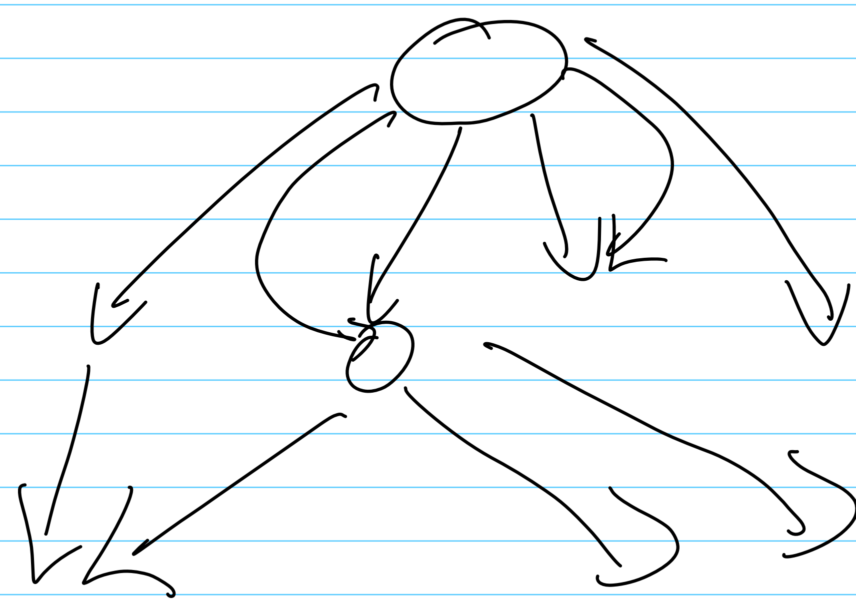
ECS 122A Lecture 6 Jasper Lee

Dynamic Programming

High-level idea, then lots of examples

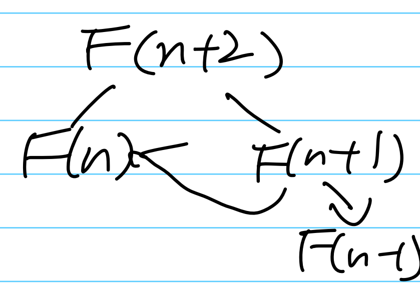
- Recursion: breaks down problem into subproblems
- D+C: no overlap in subproblems
- Sometimes, recursion can generate a tree with many identical subproblems



- DP: Avoid repeating computation!

Silly but informative example: Fibonacci numbers

$F(n+2) = F(n) + F(n+1)$
$F(0) = F(1) = 1$



If implemented recursively, takes $O(F(n))$ time!

Instead:

Init $F \leftarrow$ zeros;

for $i = 2$ to n:
$F(i) = F(i-1) + F(i-2)$

Clearly can improve to use $O(1)$ space & not $O(n)$ space

$O(n)$ time :)

Components of a (standard) DP alg:

- Table entries, including precise meaning (subproblems)
- Recurrence: How are the entries related to each other?
- (Pseudo) code to fill in table (Bottom up vs top down)
- Wrap up: How does the final answer come from table?

Top down Fib:

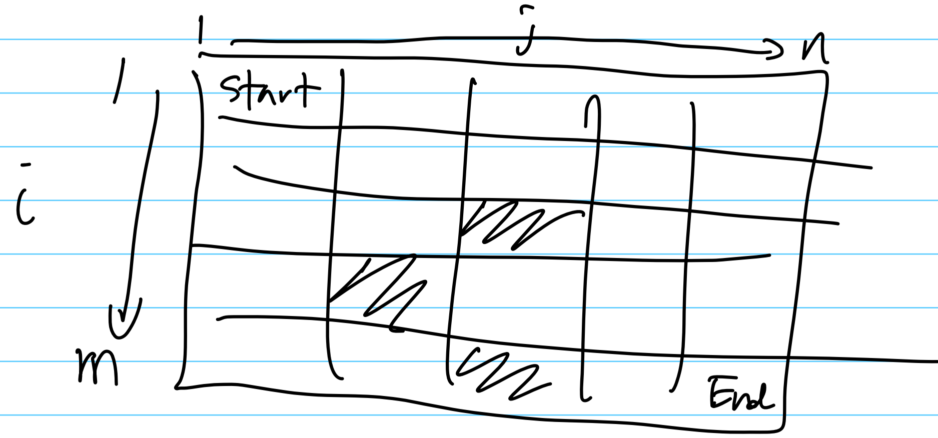
Global var array $F[0..n]$ initialized to -1s

```
function Fib(n):
    if F[n] ≠ -1 : return F[n];
    else if n = 0 : F[0] ← 1 ; return 1
            n = 1 : F[1] ← 1 ; return 1

    else F[n] ← Fib(n-1) + Fib(n-2);
         return F[n];
```

right-down paths

Input:
- Int m, n
- Grid of size m×n, some cells blocked



Count # paths going from (1,1) to (m,n), using only ↓ and ⇒ moves.

- Table entries, what are the possible subproblems?

Let's try T(i, j) structure following grid.
But what subproblem/meaning?
Try directly generalising
$T(i,j) \doteq$ # of ↓, ⇒ paths from (1,1) to (i, j).

T(blocked) = 0

How do they relate?

$T(i+1, j+1) = \overset{\downarrow}{T(i, j+1)} + \overset{\rightarrow}{T(i+1, j)}$

$T(0, 0) = 1 \quad T(\text{out of bounds}) = 0.$

gist of DP correctness

Why is the recurrence correct?

$T(i+1, j+1)$ is # of $\downarrow, \rightarrow$ paths to $(i+1, j+1)$

Any path's last step <u>must</u> be either

$T(i, j+1) \leftarrow$ $\downarrow$: then must have come from $(i, j+1)$

$T(i+1, j)$ $\rightarrow$: $(i+1, j)$

No overlap in considered paths.

So $T(i+1, j+1) = T(i, j+1) + T(i+1, j)$

How to compute?

$T(1,1) = 1$ $\quad$ $\left.\begin{array}{l} T(0, j) \leftarrow 0 \\ T(i, 0) \leftarrow 0 \end{array}\right\}$ for loops,

for $i = 1$ to m
  for $j = 1$ to n
    if (i, j) blocked then $T(i, j) \leftarrow 0$
    else $T(i, j) = T(i-1, j) + T(i, j-1)$

return $T(m, n)$

Clearly $O(mn)$ time :)

Correctness:

Implicitly an induction argument. $\left\{\right.$ Only need to make sure that when we compute $T(i, j)$, we have already computed $T(i-1, j) + T(i, j-1)$.

Already know that recurrence is true.

Maximum Subarray Sum

Setting: Given array $A[1..n]$, compute

$$\max_{i \le j} \text{sum}(A[i..j])$$

Naive alg: $O(n^3)$ time (Try all $(i \le j)$, compute sum)

Want: $O(n)$ time!

If we want $O(n)$ time, how large can the table be? Only $O(n)$ size...

(Each entry takes $\Omega(1)$ time to compute/process)

$T[1..n]$ then? What can $T[i]$ mean?

$T[i] \doteq$ Max subarray sum of $A[1..i]$.

$T[1] = A[1]$

Reasonable generalization of whole problem?

Recurrence? Observation?

Either use $A[i]$ or not.

how? how to do it in silly way?

$T[i-1]$

Slow recurrence:

$$T[i] = \max\Big(T[i-1],\ \underbrace{\begin{array}{l} \text{sum}(A[i]), \\ \text{sum}(A[i-1..i]), \\ \text{sum}(A[i-2..i]), \\ \vdots \\ \text{sum}(A[1..i]) \end{array}}_{\text{New DP}}\Big)$$

True but slow :(Double for loop $O(n^2)$ time

Compute max subarray sum in $A[1..i]$ that has to use $A[i]$.
is itself solvable by DP :)

Obs: Either use $A[i-1]$ or not

might as well use max subarray sum in $A[1..i-1]$ that has to use $A[i-1]$

Just $A[i]$

$$T_2[i] = \max(A[i],\ T_2[i-1] + A[i])$$

alternatively just checking if this is negative.

$O(n)$ time recurrence

Better recurrence for $T[i]$

$$T[i] = \max(T[i-1],\ T_2[i])$$

Also $O(n)$ time now :)