Dynamic Programming

Dynamic Programming

The most **robust** algorithm design paradigm we'll study this quarter.

Small changes in the problem usually lead to small changes in the algorithm.

Also the one you're most likely to be asked about in a tech interview.

Classic DP

This problem is going to **look** silly (and it is)

But it is going to make it much easier to do the hard DP problems next week.

Baby Yoda Searching

Baby Yoda has to get from the upper-right corner to the lower left. His cradle will only let him go left and down.

He can't get past the rocks (too high) – he has to go around them (but still only going left and down)

As usual…he's hungry.

He wants to eat as many frog eggs as possible on the way.

Baby Yoda Searching

Black path: get stuck. Invalid.

Red path: valid! And optimal (no path collects more than 4 eggs.)

Baby Yoda Searching

Might get us stuck between rocks.
Or pass up a series of eggs we can't see.

Can we greedily head to the next accessible egg?

Baby Yoda Searching

Best left-side path might start at a place inaccessible to end of best right-side path.

Could make a subproblem for each start and ending spot?

Can we divide and conquer?

Baby Yoda Searching

Baby Yoda Searching

So what should we do?

Let's try to use recursion.

What should our recursive calls be finding?

What recursive calls do we need?

Let `OPT(i,j)` be the maximum number of eggs we can get on a legal path from `(i,j)` to `(0,0)` (including the egg in `(i,j)` if there is one)

What recursive calls do we need?

Don't try to divide & conquer, think closer to home…

We have to decide whether to go down or left…

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

(0,2)

(0,1) (1,1)

0 (0,0)

X-coordinate

0

c-1

Recursive Baby Yoda

Let `OPT(i,j)` be the maximum number of eggs we can get on a legal path from `(i,j)` to `(0,0)` (including the egg in `(i,j)` if there is one)

Base Case?

At `(0,0)`, nowhere to go, return `eggs[0][0]`

Recursive case?

Find best path to left `OPT(i-1, j)`, and down `OPT(i,j-1)`
Take `max` of those, add in `eggs[i][j]`
Need some error handling (can't go off the edge)
And if we're on rocks, we can't get to the end (`return -∞`)

A Recursive Function

```
FindOPT(int i,int j, bool[][] rocks, bool[][] eggs)
    if(i<0 || j < 0) return -∞
    if(rocks[i][j]) return −∞
    if(i==0 && j==0) return eggs[0][0]
    int left = FindOPT(i-1,j,rocks,eggs)
    int down = FindOPT(i,j-1,rocks,eggs)
    return Max(left,down) + eggs[i][j]
```

Recurrence Form

$$OPT(i,j) = \begin{cases} -\infty & \text{if } rocks(i,j) \text{ is true} \\ -\infty & \text{if } i < 0 \text{ or } j < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \\ \max\{OPT(i-1,j), OPT(i,j-1)\} + eggs(i,j) & \text{otherwise} \end{cases}$$

Recurrences can also be used for outputs of a recursive function (not just their running times!)

This definition is a little more compact than code.

And you could write a recursive function for a recurrence like this.

Analyzing the recursive function

So…how does the code work? What's its running time?

$$T(c,r) = \begin{cases} T(c-1,r) + T(c,r-1) + \Theta(n) & \text{if } r \geq 0 \text{ and } c \geq 0 \\ \Theta(1) & \text{otherwise} \end{cases}$$

Master Theorem says…

Analyzing the recursive function

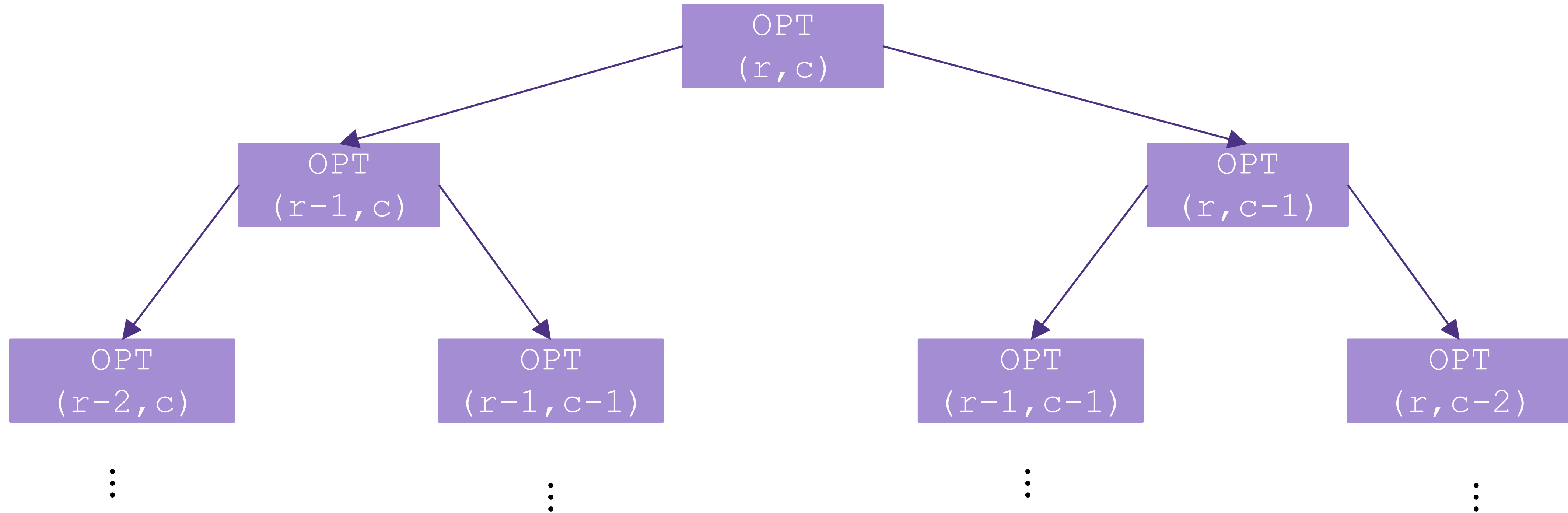
So…how does the code work? What's its running time?

$$T(c,r) = \begin{cases} T(c-1,r) + T(c,r-1) + \Theta(n) & \text{if } r \geq 0 \text{ and } c \geq 0 \\ \Theta(1) & \text{otherwise} \end{cases}$$

Master Theorem doesn't help.

Not even the fancy version on Wikipedia that handled the logs last time.

Tree Method, Maybe…



When do we hit the base case?
Sometime between $\min(r, c)$ and $r + c$ levels.

Tree Method

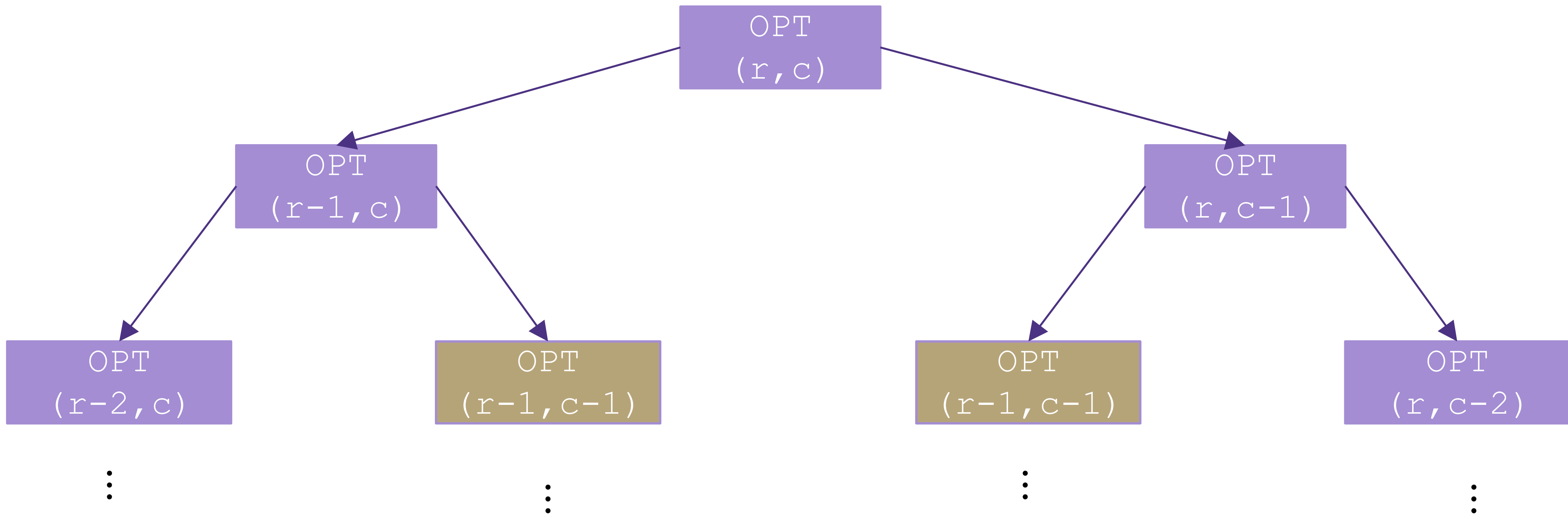
Nodes at level i	2^i	
Work/node	$\Theta(1)$	
Work at level i	$\Theta(2^i)$	
Base Case level	At least $\min(r,c)$	At most $r+c$
Work at base case	$\Omega(2^{\min(r,c)})$	$O(2^{r+c})$
Total work	$\Omega(2^{\min(r,c)})$	$O(2^{r+c})$

Overall work is sum over all levels – each level has twice the work as the last, so the last level is about half the total work.

Tight big-O depends on relationship between r and c…but regardless – it's slow.

Speedup

That's way too slow…but it doesn't have to be.



Activity

Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity

Figure out how to take advantage of the repeated calculation.
What do you think the running time will be of your new algorithm?

```
FindOPT(int i,int j, bool[][] rocks, bool[][] eggs)
    if(i<0 || j < 0) return -∞
    if(rocks[i][j]) return —∞
    if(i==0 && j==0) return eggs[0][0]
    int left = FindOPT(i-1,j,rocks,eggs)
    int down = FindOPT(i,j-1,rocks,eggs)
    return Max(left,down) + eggs[i][j]
```

Speedup

How do we go faster? Don’t recalculate! **memoize**

Once you know $OPT(i, j)$ put it in an array `OPT[i][j]`

Have some initial value (null?) to mark as unitialized

If initialized, return that.

Otherwise do the algorithm from the last slide.

How fast? Now $\Theta(rc)$. A little harder to analyze – ask Robbie after

Baby Yoda Searching

							(c-1,r-1)
(0,2)							
(0,1)	(1,1)						
(0,0)							

Y-coordinate: 0 … r-1

X-coordinate: 0 … c-1

Going Bottom-up

So how does that recursion work?

What's the first entry of the table that we fill?

`OPT[0][0]`

Why not just start filling in there?

Baby Yoda Searching

Y-coordinate								(c-1,r-1)
r-1								
	(0,2)							
	(0,1)	(1,1)						
0	(0,0) 0	1	2	2	2	2	2	2
	0			X-coordinate				c-1

What else can we fill in?

Baby Yoda Searching

(c-1,r-1)

Y-coordinate								
r-1								
	(0,2)							
	(0,1) 0	(1,1) 1	$-\infty$	2	2	2	2	2
0	0	1	2	2	2	2	2	2

X-coordinate

0 c-1

What else can we fill in?

Baby Yoda Searching

What else can we fill in?

(c-1,r-1)

Y-coordinate								
r-1								
	0	$-\infty$						
	0	1	$-\infty$	2	2	2	2	2
0	0	1	2	2	2	2	2	2

X-coordinate

0 … c-1

Baby Yoda Searching

(c-1, r-1)

r-1								
	1	1	1	2	3	4	4	4
	1	1	1	2	3	4	4	4
	0	0	1	2	3	3	3	3
	0	$-\infty$	$-\infty$	$-\infty$	3	3	3	3
	0	1	$-\infty$	2	2	2	2	2
0	0	1	2	2	2	2	2	2

Y-coordinate

X-coordinate

0 c-1

Where's the final answer?

In the top right. Where Baby Yoda starts.

What order?

Fill in a row at a time (left to right)

Going up to the next row once a level is done.

In actual code, probably easier to handle edges first

Avoid the index-out-of-bound exceptions.

Pseudocode

```
int eggsSoFar=0;
Boolean rocksInWay=false
for(int x=0; x<c; x++)
   if(rocks[x][0]) rocksInWay = true
   eggsSoFar+=eggs[x][0]
   OPT[x][0]= rocksInWay ? −∞ : eggsSoFar

eggsSoFar=0
rocksInWay=false
for(int y=0; y<r; y++)
   if(rocks[0][y]) rocksInWay = true
   eggsSoFar+=eggs[0][y]
   OPT[0][y]= rocksInWay ? −∞ : eggsSoFar

for(int y=0;y<r;y++)
  for(int x=0;x<c;x++)
    if(rocks[x][y])
        OPT[x][y]=-∞
    else
        OPT[x][y]=max(OPT[x-1][y], OPT[x][y-1])+eggs[x][y]
```

Why Switch To Iterative?

It does the same thing…

It’s easier to analyze (no need to imagine a recursion tree)

Saves constant factors (recursive version puts a lot on the call stack)

Will let you optimize memory (next week)

Recursive version is often a little more intuitive, though…

Updating the Problem

A new twist on the problem.

Baby Yoda can use the force to knock over rocks.

But he can only do it once (he tires out)

How do you decide which rocks to knock over?

Could run the algorithm once for every set of rocks knocked over.

k rocks -- $\Theta(krc)$. Can we do better?

Updating the Problem

$OPT(i,j,f)$ is the maximum amount of eggs Baby Yoda can collect on a legal path from (i,j) to $(0,0)$ using the force f times to knock over rocks.

For simplicity, assume there are no rocks at the starting location (r-1,c-1)

Here was the old rule without the force – how do we update?

$$OPT(i,j) = \begin{cases} -\infty & \text{if } rocks(i,j) \text{ is true} \\ -\infty & \text{if } i < 0 \text{ or } j < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \\ \max\{OPT(i-1,j), OPT(i,j-1)\} + eggs(i,j) & \text{otherwise} \end{cases}$$

Updating the Problem

$OPT(i,j,f)$ is the maximum amount of eggs Baby Yoda can collect on a legal path from (i,j) to $(0,0)$ using the force f times to knock over rocks.

For simplicity, assume there are no rocks at the starting location (r-1,c-1)

Here was the old rule without the force – how do we update?

$$OPT(i,j,f) = \begin{cases} -\infty & \text{if } i < 0 \text{ or } j < 0 \text{ or } f < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \text{ and } f \geq 0 \\ \max\{OPT(i-1,j,f - rocks(i-1,j)), OPT(i,j-1,f - rocks(i,j-1))\} + eggs(i,j) & \text{otherwise} \end{cases}$$

Casting Boolean as an integer (subtract 1 if you would need to knock over rocks)

Updating the Problem

$OPT(i,j,f)$ is the maximum amount of eggs Baby Yoda can collect on a legal path from (i,j) to $(0,0)$ using the force f times to knock over rocks.

For simplicity, assume there are no rocks at the starting location (r-1,c-1)

Here was the old rule without the force – how do we update?

$$OPT(i,j,f) = \begin{cases} -\infty & \text{if } i < 0 \text{ or } j < 0 \text{ or } f < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \text{ and } f \geq 0 \\ \max\{OPT(i-1,j,f-rocks(i-1,j)), OPT(i,j-1,f-rocks(i,j-1))\} + eggs(i,j) & \text{otherwise} \end{cases}$$

`rocks(i,j)` doesn't guarantee $-\infty$ anymore. Only if you were out of force uses before trying to jump onto that location.

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

X-coordinate

0

c-1

What can we fill in?

a/b

a is for (x,y,0)

b is for (x,y,1)

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?

X-coordinate

0 c-1

What can we fill in?

a/b

a is for (x,y,0)

b is for (x,y,1)

Baby Yoda Searching

(c-1, r-1)

Y \ X	0							c-1
r-1	1/?	1/?	1/?	1/?	3/?	4/?	4/?	4/?
	1/?	1/?	1/?	1/?	3/?	4/?	4/?	4/?
	0/?	0/?	1/?	1/?	3/?	3/?	3/?	3/?
	0/?	1/?	$-\infty$/?	2/?	3/?	3/?	3/?	3/?
	0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?
0	0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?

Y-coordinate

X-coordinate

What can we fill in? Everything with $f = 0$ in the same order as before.

Entries are slightly different – we're handling rocks differently.

Baby Yoda Searching

(c-1, r-1)

r-1	1/?	1/?	1/?	1/?	3/?	4/?	4/?	4/?
	1/?	1/?	1/?	1/?	3/?	4/?	4/?	4/?
	0/?	0/?	1/?	1/?	3/?	3/?	3/?	3/?
	0/?	1/?	$-\infty$/?	2/?	3/?	3/?	3/?	3/?
	0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?
0	0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?

Y-coordinate

X-coordinate

0 c-1

What can we fill in?

Again from left to right, bottom to top, now filling in

Baby Yoda Searching

(c-1, r-1)

r-1	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	0/0	0/0	1/4	1/4	3/4	3/4	3/4	3/4
	0/0	1/1	$-\infty/3$	2/3	3/3	3/3	3/3	3/3
	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2
0	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2

Y-coordinate

X-coordinate

0 c-1

What can we fill in?
Again from left to right, bottom to top, now filling in

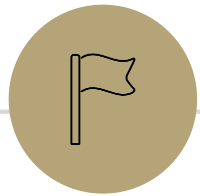
Dynamic Programming Process

1. Define the object you're looking for

2. Write a recurrence to say how to find it

3. Design a memoization structure

4. Write an iterative algorithm



Bells,Whistles, and optimiziation

Baby Yoda Searching

(c-1, r-1)

Y-coordinate								
r-1	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	0/0	0/0	1/4	1/4	3/4	3/4	3/4	3/4
	0/0	1/1	$-\infty/3$	2/3	3/3	3/3	3/3	3/3
	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2
0	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2

X-coordinate

0 ... c-1

What can we fill in?

Again from left to right, bottom to top, now filling in

Which Way to Go

When you're taking the `max` in the recursive case, you can also record which option gave you the max.

That's the way to go.

We'll ask you to do that once in HW4…but for the most part we'll just have you find the number.

Optimizing

Do we need all that memory?

Let’s go back to the simple version (no using the force)

Recurrence Form

$$OPT(i,j) = \begin{cases} -\infty & \text{if } rocks(i,j) \text{ is true} \\ -\infty & \text{if } i < 0 \text{ or } j < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \\ \max\{OPT(i-1,j), OPT(i,j-1)\} + eggs(i,j) & \text{otherwise} \end{cases}$$

What values do we need to keep around?

Baby Yoda Searching

(c-1, r-1)

Y-coordinate								
r-1	1	1	1	2	3	4	4	4
	1	1	1	2	3	4	4	4
	0	0	1	2	3	3	3	3
	0	$-\infty$	$-\infty$	$-\infty$	3	3	3	3
	0	1	$-\infty$	2	2	2	2	2
0	0	1	2	2	2	2	2	2

Need one spot left and one down.

Keep one full row, and a partially full row around. $\Theta(c)$ memory.