

Dynamic Programming

CSE 421 23Wi
Lecture 10

Divide And Conquer Summary

Takeaways from D&C:

Recursive thinking can let us solve problems faster!

When solving a recursive problem, state precisely what the recursive call is giving back to you.

Use the values of the recursive call (or at least the fact you've made recursive calls) when designing the combine step

If your "combine" step isn't faster than baseline, your whole algorithm isn't better than the baseline!

How Were We Solving All Those Recurrences?

The techniques from CSE 332 (unrolling and/or recursion trees) still work!

But now that you've done them a bunch in 332, we'll give you a shortcut...

Master Theorem

Given a recurrence of the following form, where $a, b, c,$ and d are constants:

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c \log^k n)$ for $k \geq 0, a \in \mathbb{Z}^+, c \geq 1$

If $\log_b a < c$ then $T(n) \in \Theta(n^c \cdot \log^k n)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log^{k+1} n)$

If $\log_b a > c$ then $T(n) \in \Theta(n^{\log_b a})$

Theorem still holds even if there are ceilings/floors in the $T\left(\frac{n}{b}\right)$ term.

Proof?

The easiest way to understand is actually to make a bunch of recursion trees!

a is the "branching factor" you end up with a^i nodes at level i .

b is the amount you cut down the problem size by.

So at level i you do: $a^i \cdot f\left(\frac{n}{b^i}\right)$ work.

For example, when $c = \log_b(a)$, that simplifies to $\frac{a^i n^c}{b^{\log_b(a) \cdot i}} = \frac{a^i n^c}{a^i} = n^c$

And there will be $\log_b n$ levels, so the work is: $n^c \log n$.

Proof?

When $\log_b a < c$

The total work decreases at each level, so the first level dominates, where you do $O(n^c)$ work.

When $\log_b a > c$

The total work increases at each level, so the last level (which is at $\log_b a$) dominates, where you have $O(a^{\log_b n})$ nodes, each doing $O(1)$ work. Some log tricks will rearrange to the more familiar $O(n^{\log_b a})$.

A stronger version

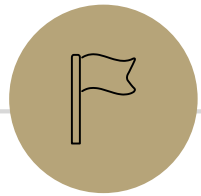
Also what to do when format doesn't quite fit (e.g. k negative).

The version on the last slide should suffice.

From [Wikipedia](#)

$$c_{\text{crit}} = \log_b a = \log(\text{\#subproblems}) / \log(\text{relative subproblem size})$$

Case	Description	Condition on $f(n)$ in relation to c_{crit} , i.e. $\log_b a$	Master Theorem bound
1	Work to split/recombine a problem is dwarfed by subproblems. i.e. the recursion tree is leaf-heavy	When $f(n) = O(n^c)$ where $c < c_{\text{crit}}$ (upper-bounded by a lesser exponent polynomial)	... then $T(n) = \Theta(n^{c_{\text{crit}}})$ (The splitting term does not appear; the recursive tree structure dominates.)
2	Work to split/recombine a problem is comparable to subproblems.	When $f(n) = \Theta(n^{c_{\text{crit}}} \log^k n)$ for a $k \geq 0$ (rangebound by the critical-exponent polynomial, times zero or more optional logs)	... then $T(n) = \Theta(n^{c_{\text{crit}}} \log^{k+1} n)$ (The bound is the splitting term, where the log is augmented by a single power.)
3	Work to split/recombine a problem dominates subproblems. i.e. the recursion tree is root-heavy.	When $f(n) = \Omega(n^c)$ where $c > c_{\text{crit}}$ (lower-bounded by a greater-exponent polynomial)	... this doesn't necessarily yield anything. Furthermore, if $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n (often called the <i>regularity condition</i>) then the total is dominated by the splitting term $f(n)$: $T(n) = \Theta(f(n))$



Dynamic Programming



Dynamic Programming

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

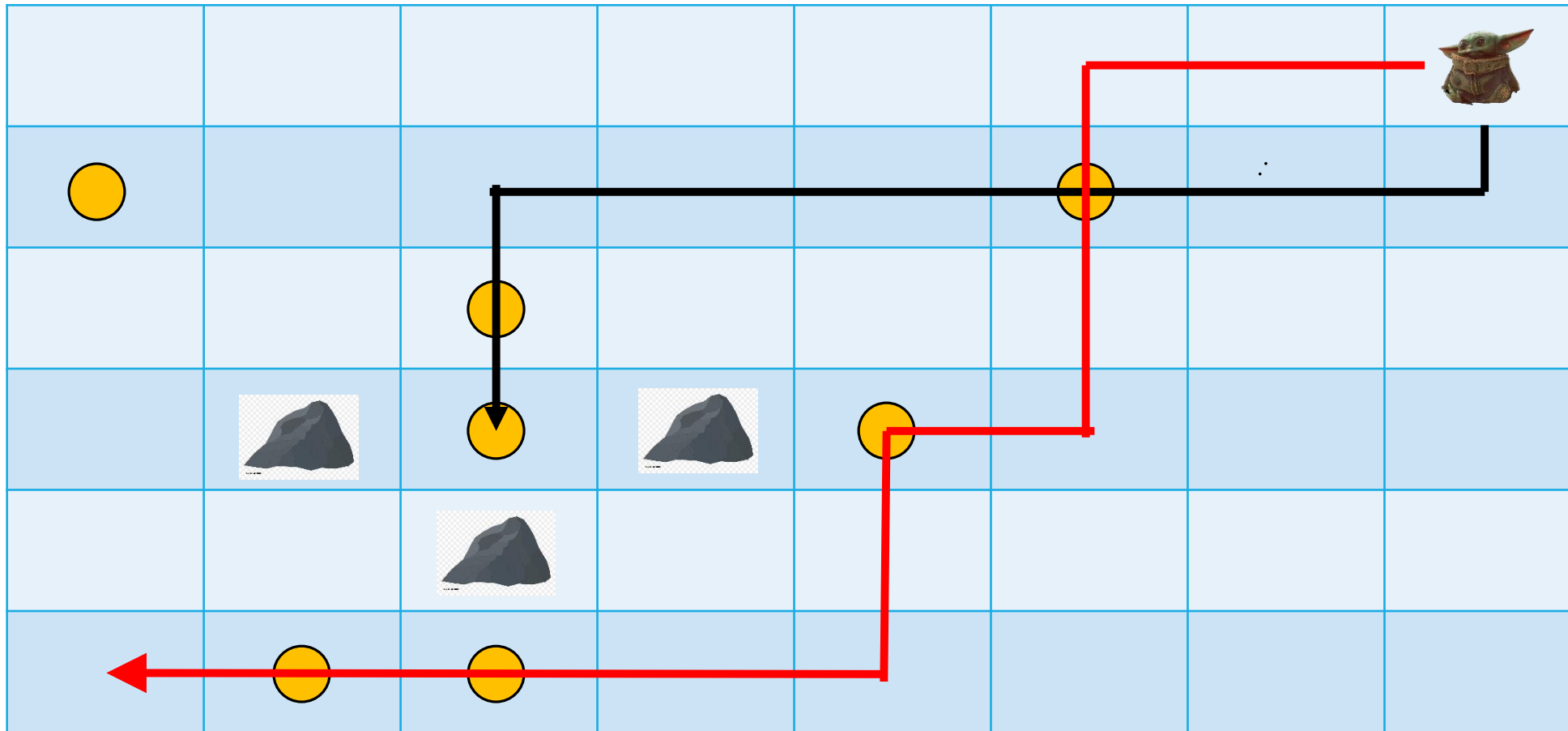
Small changes in the problem usually lead to small changes in the algorithm.

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



Black path: get stuck. Invalid.

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

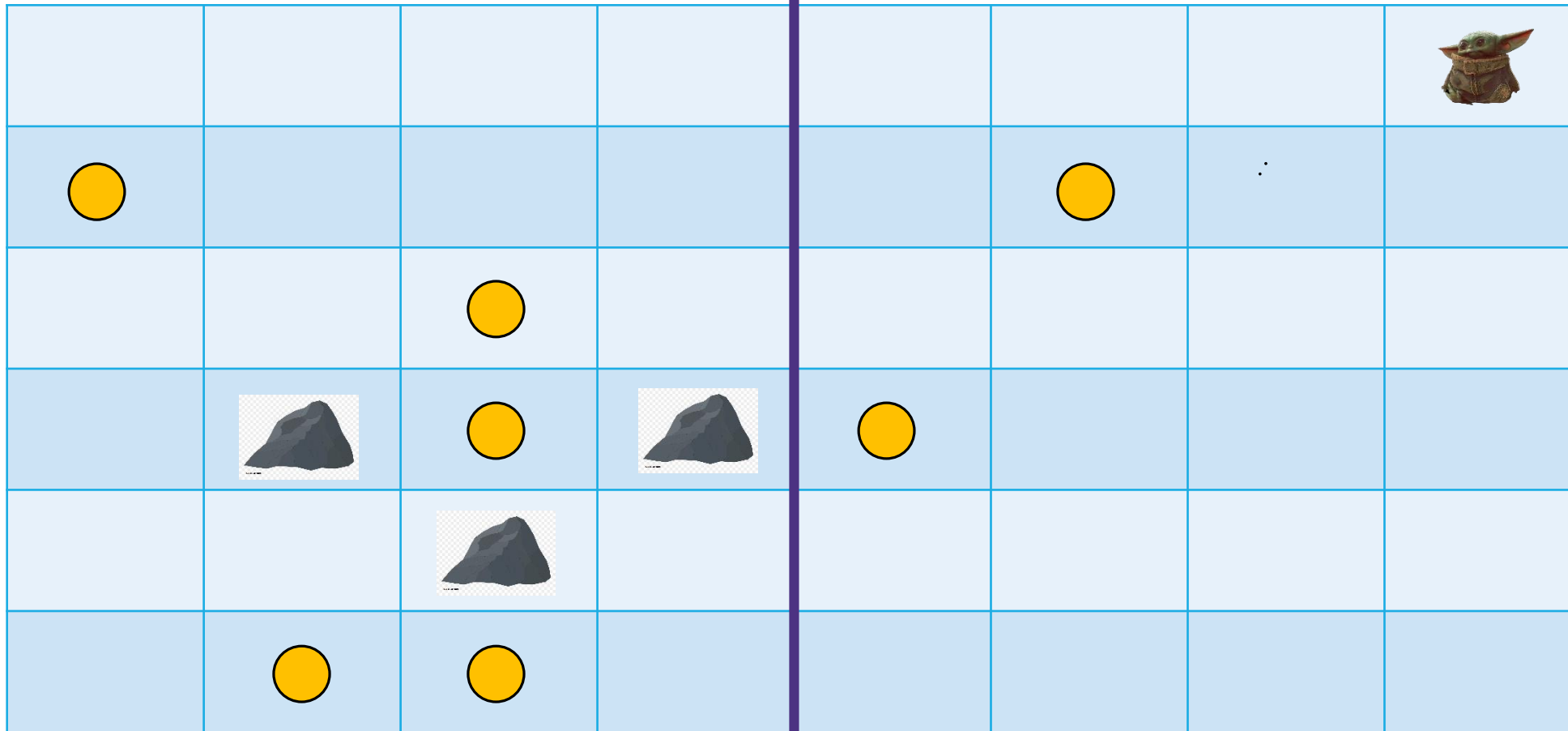
Baby Yoda Searching



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



							
						.	
							
							
							
							

Define the problem

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)

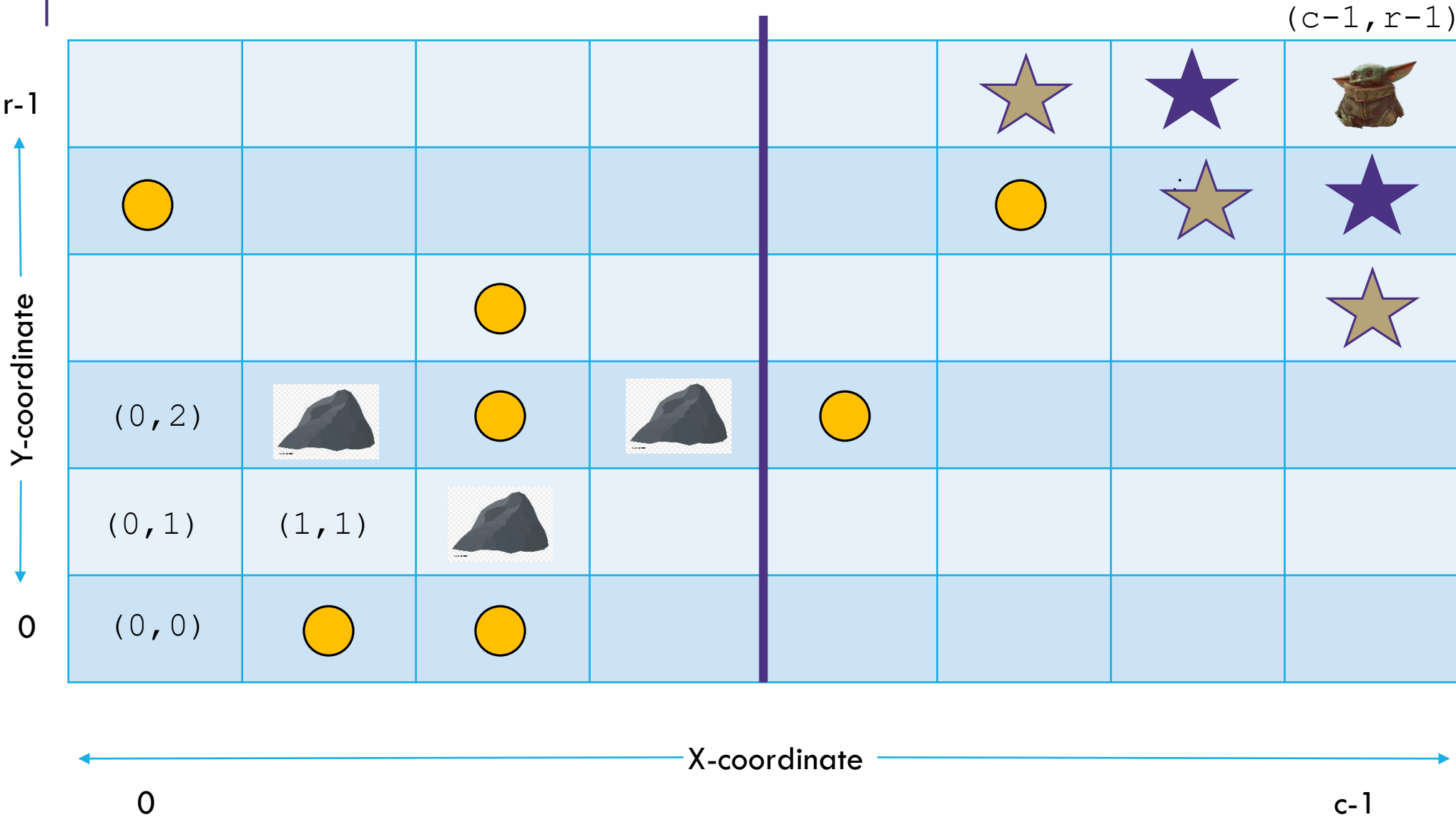
What recursive calls do we need?

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

We have to decide whether to go down or left...

"How could we take one step toward the solution?"

Baby Yoda Searching



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 $-\infty$)

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

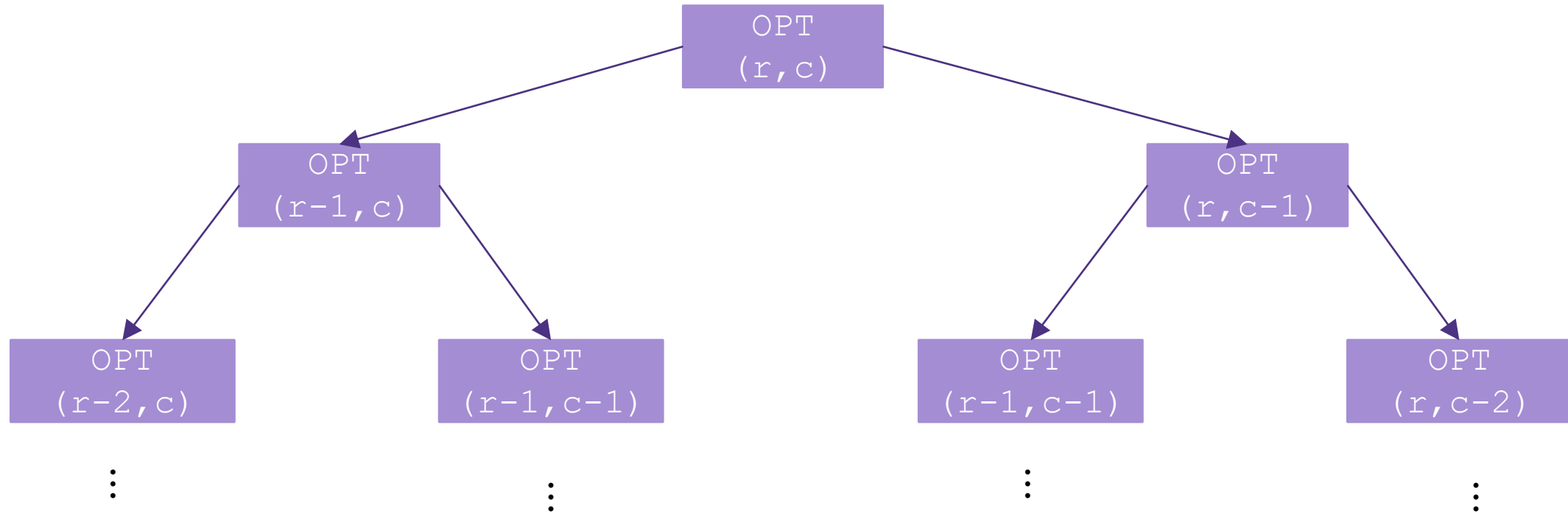
So...how does the code work? What's its running time?

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

Master Theorem doesn't help.

Not even the fancy version on Wikipedia. It's only for "divide and conquer style" recurrences (**dividing** the problem)

Tree Method, Maybe...



When do we hit the base case?

Sometime between $\min(r, c)$ and $r + c$ levels.

Tree Method

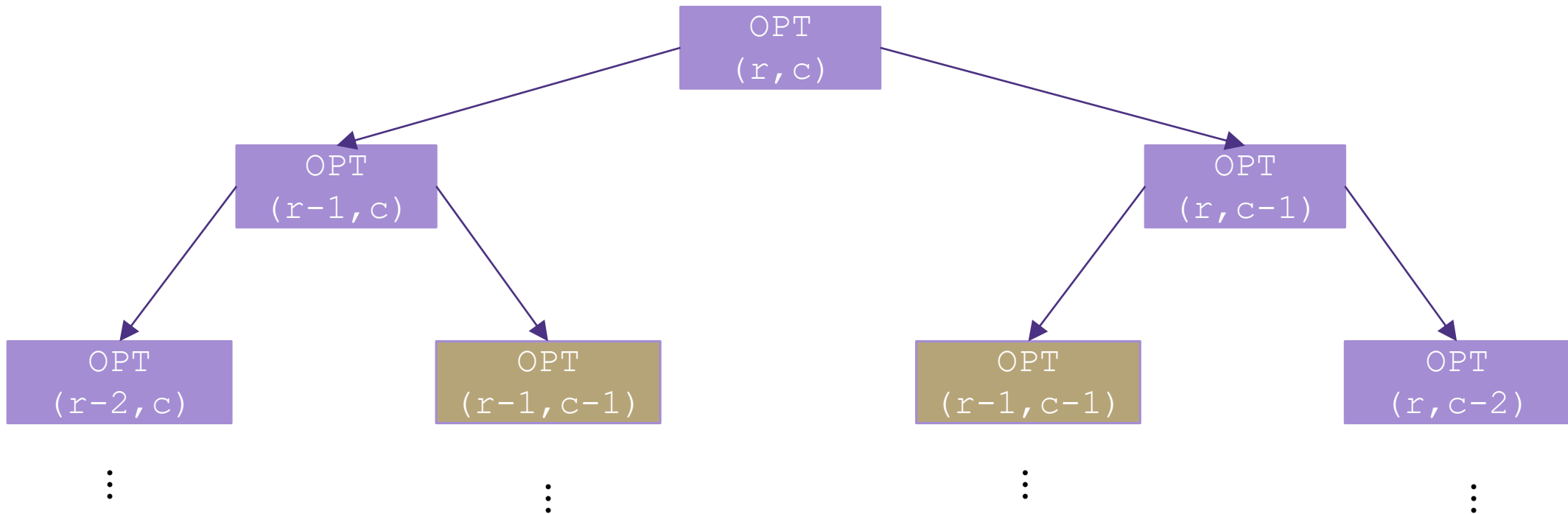
Nodes at level i	2^i	
Work/node	$\Theta(1)$	
Work at level i	$\Theta(2^i)$	
Base Case level	At least $\min(r, c)$	At most $r + c$
Work at base case	$\Omega(2^{\min(r, c)})$	$O(2^{r+c})$
Total work	$\Omega(2^{\min(r, c)})$	$O(2^{r+c})$

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 question at
pollev.com/robbie

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

Not a typo! We're leaving ourselves a memo.

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

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

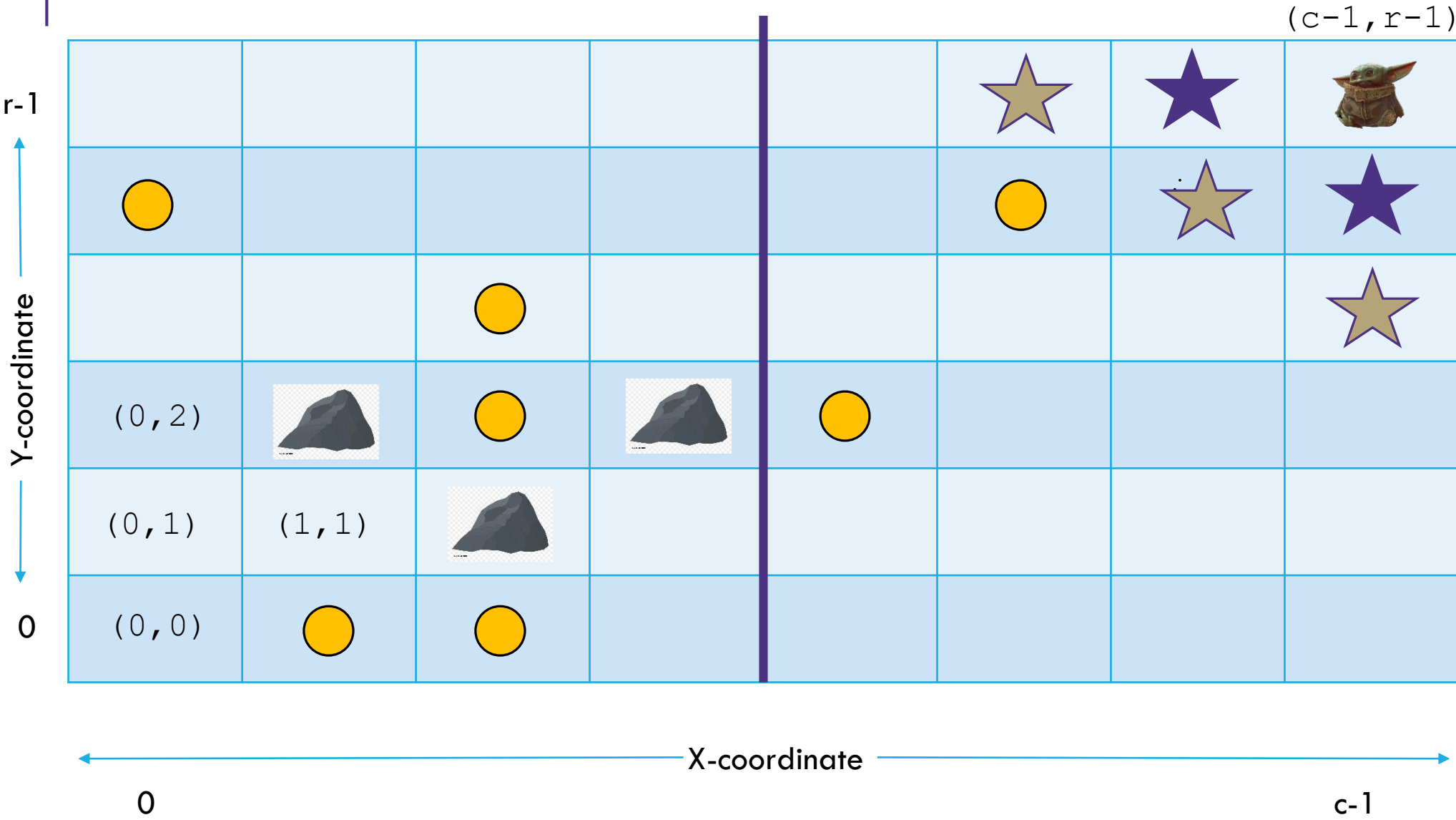
Have some initial value (null?) to mark as uninitialized

If initialized, return that.

Otherwise do the algorithm from the last slide.

How fast? Now $\Theta(rc)$.

Baby Yoda Searching



Going Bottom-up

So how does that recursion work?

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

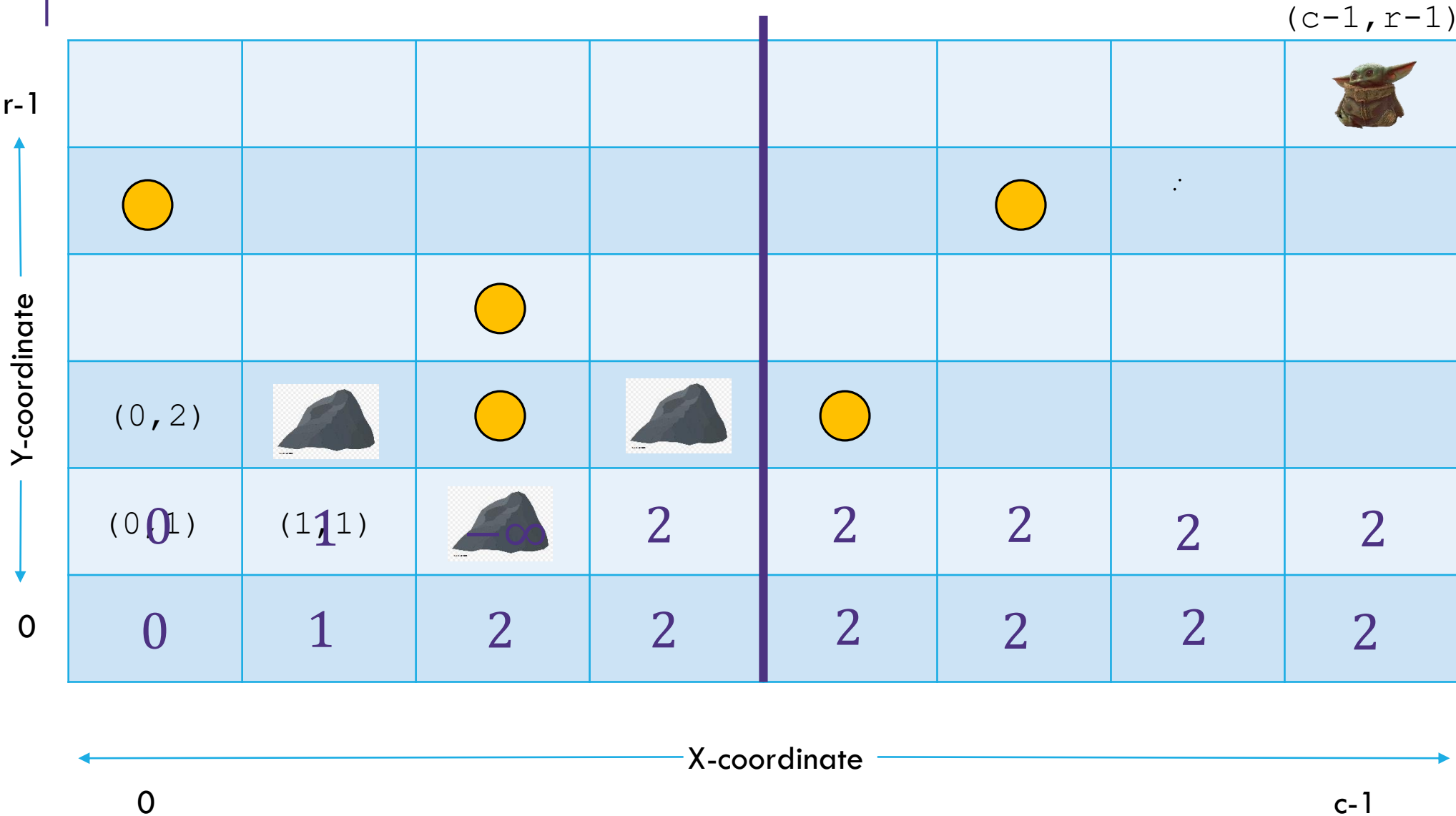
```
OPT[0][0]
```

Why not just start filling in there?

Baby Yoda Searching



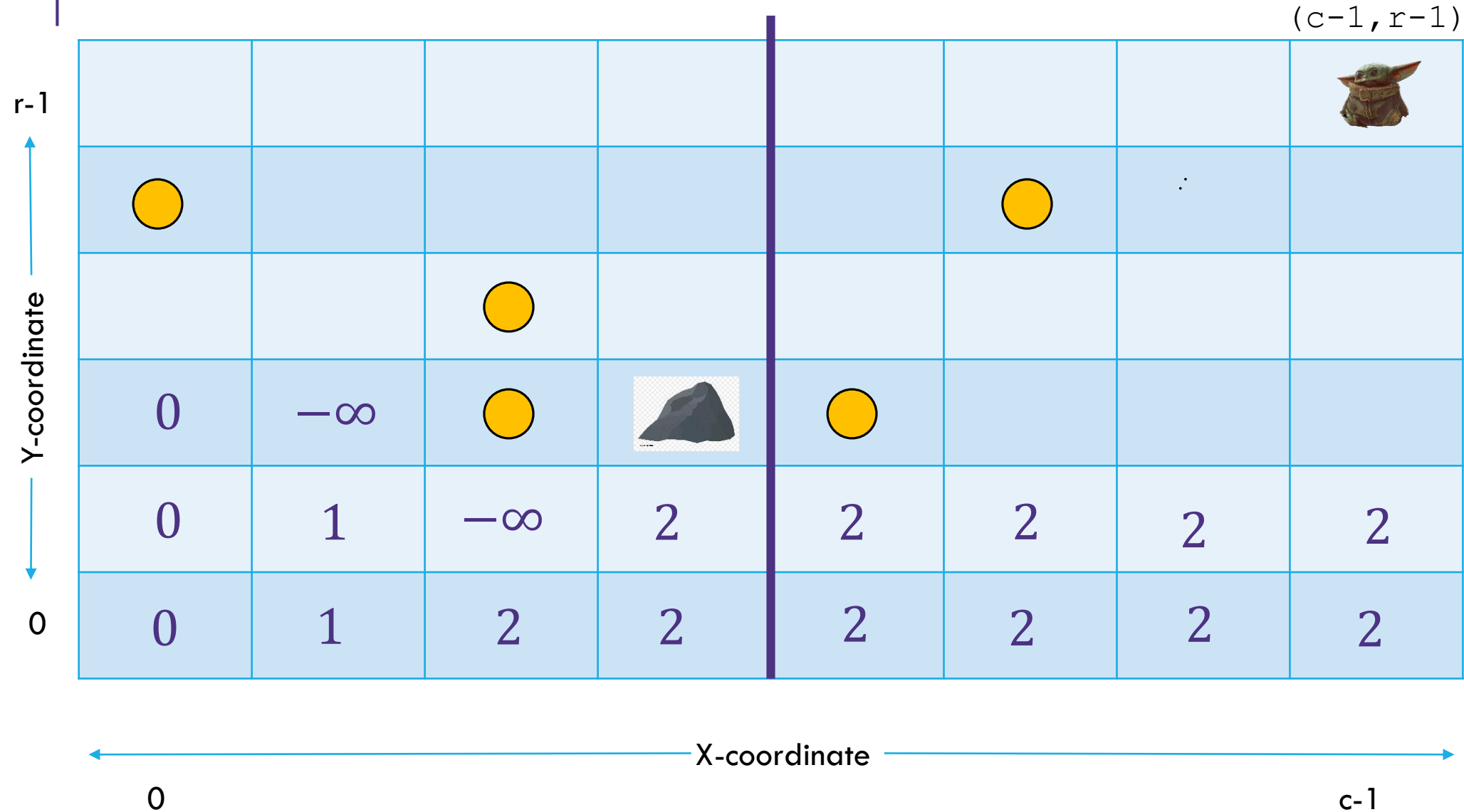
What else can we fill in?



Baby Yoda Searching



What else
can we fill in?



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

Where's the final answer?

In the top right. Where Baby Yoda starts.

← X-coordinate →
0 c-1

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 ?  $-\infty$  : 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 ?  $-\infty$  : eggsSoFar
```

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