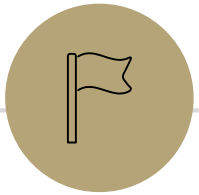
D&C wrap; Dynamic Programming

CSE 417 Winter 24
Lecture 10



Classic Divide & Conquer

Activity

Write a recurrence to describe the running time of this code. What's the big-O?

```
DivConqExponentiation(int a, int b, int n)
  if(n==0) return 1
  if(n==1) return a%b
  int k = divConqExponentiation(a,b,n/2) /*int div*/
  if(n%2==1) return (k*k*a)%b
  return (k*k)%b
```

Go to pollev.com/robbie so I know how long to explain

Running Time?

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + \Theta(1) & \text{if } n \geq 2 \\ \Theta(1) & \text{otherwise} \end{cases}$$

Which is $\Theta(\log n)$ time.

Much faster than $O(n)$.

Fun Fact: RSA encryption (one of the schemes most used for internet security) needs exactly this sort of "raise to a large power, mod b" operation.

Classic Applications

There are a few really famous divide & conquer algorithms where the way to recurse is **very clever**.

The point of the next few examples is **not** to teach you really useful tricks (they often don't generalize to new problems).

It's partially to show you that sometimes being really clever gives you a really big improvement

And partially because these are "standard" in algorithms classes, so you should at least have heard of these algorithms.

Multiplying Faster

Our end goal is to multiply really big numbers.

Really big.

Like around 1000 bits per number.

Why? Another step in the RSA encryption scheme!

Also a favorite problem of theoreticians.

Place Values

Remember the decimal number 12,345 is really:

$1 \cdot 10^4 + 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$

The "normal" representation for numbers is base-10. Each place value is worth 10 times more than the same number would be one spot over.

The choice of 10 is arbitrary. You can use 2 and get binary

101101_2 is $1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Place Values

Imagine, we instead of using base-10, use base $\mathbf{100}$.

We need 100 symbols now. (we need 2 bits for binary, 10 digits for decimal) But other than that, it would make sense.

We could take 12345 and think of it as:

$$1 \cdot 100^2 + 23 \cdot 100^1 + 45 \cdot 100^0$$

If we thought in base-100, we'd have special symbols for 23 and 45…but it does show what's going on.

Notice that since 100 is a power of 10, we've "regrouped" the digits.

Place Values

If we have an n-digit number in base 10. We can think of it as a number in base 10^i by just "grouping together" the digits in groups of i.

We just saw $i = 2$

$i = 3$ 12345 in base 10 is:

$12 \cdot 1000^1 + 345 \cdot 1000^0$ in base 1000.

Why does it matter?

Imagine I asked you to multiply

$$
\begin{array}{r}
12345 \\
\times\ 67210 \\
\hline
00000 \\
12345- \\
24690- - \\
\cdots
\end{array}
$$

Take the ones place, multiply digit-by-digit.
Take the 10's place, multiply digit-by-digit, and shift answer by 10.
Take the 100's place, multiply digit-by-digit, and shift the answer by 100.

Why does it matter?

Now imagine they're in base 1000

$$\begin{array}{r} 12345 \\ \times\ 67210 \\ \hline \end{array}$$

Take the ones place, multiply digit-by-digit.
Take the 1000's place, multiply digit-by-digit, and shift answer by 1000.

Alright…so…

We can multiply in a different way, why should you care?

It’s going to let us divide-and-conquer multiplication.

Classic Application

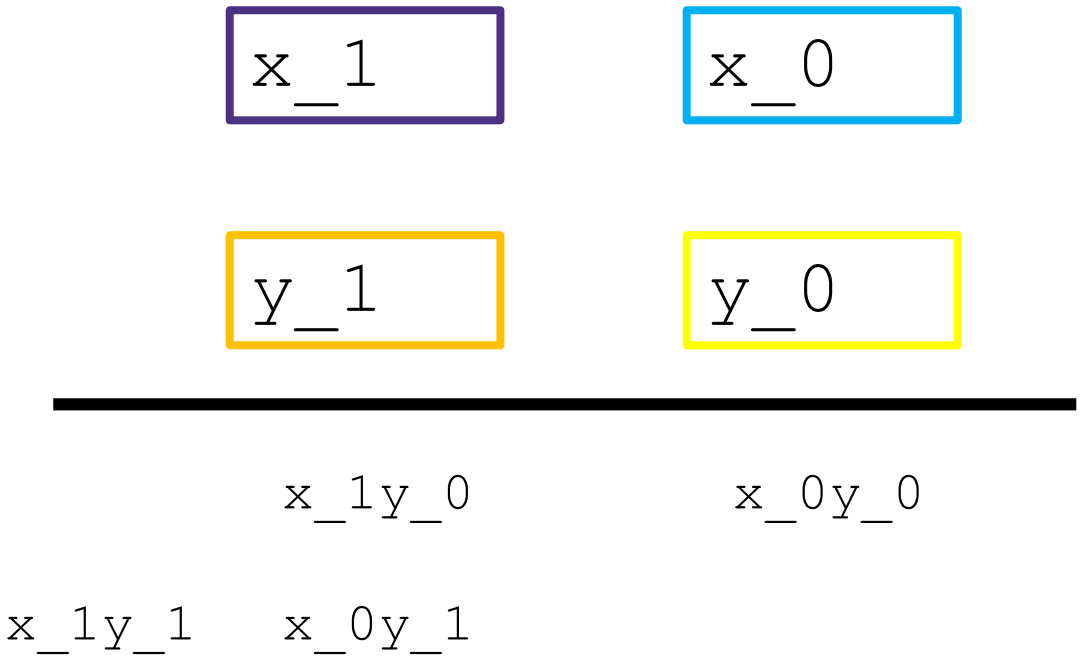
Suppose you need to multiply **really** big numbers.

Much bigger than `ints`

Split the n bit numbers in half

Think of them as written in base $2^{n/2}$

What would the "normal" multiplication algorithm do?



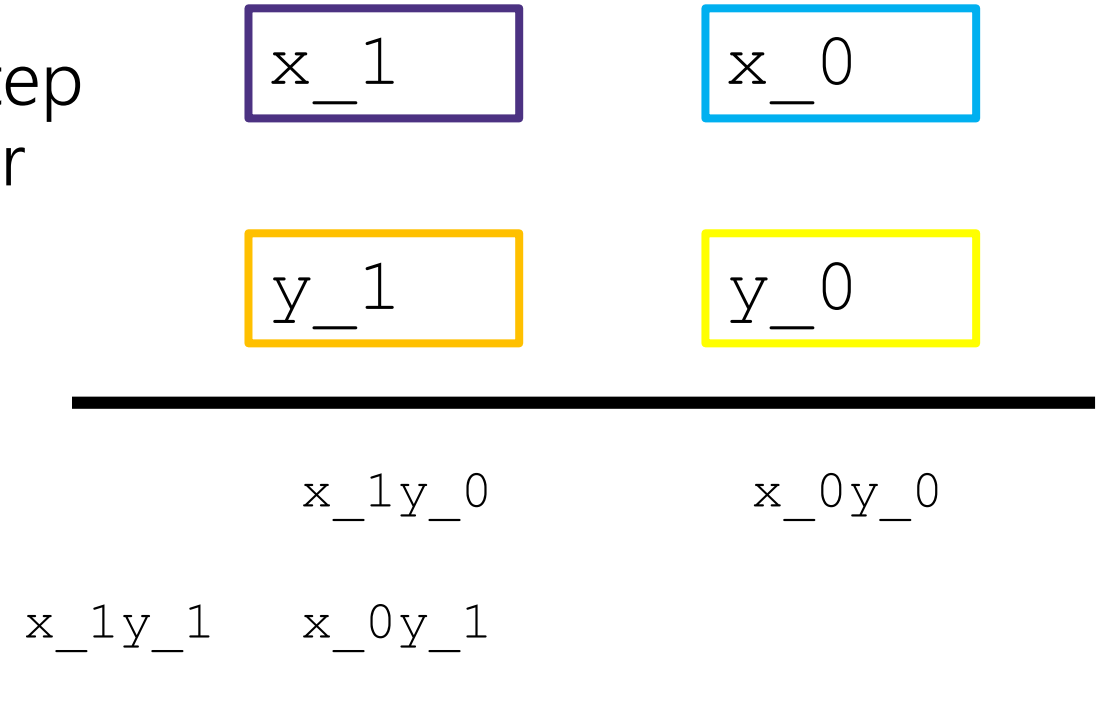
4 multiplications, i.e. 4 recursive calls.

Classic Application

If n bits is too many to multiply in one step (e.g. it's more than one byte, or whatever your processor does in one cycle)

Recurse! Running time?

$$T(n) = \begin{cases} 4T\left(\frac{n}{2}\right) + O(n) \text{ if } n \text{ is large} \\ O(1) \quad \text{if } n \text{ fits in one byte} \end{cases}$$



Why $O(n)$? It takes $O(n)$ time to add up $O(n)$ bit numbers – they have $O(n)$ bytes!
(Why have you never seen this before? We assumed our numbers were ints, where the number of bytes is a constant)

Overall running time is $\Theta(n^2)$

Clever Trick

We need to find x_1y_0 + x_0y_1.

Does that look familiar? It's the middle to terms when you FOIL

Define $\alpha = (x_0 + x_1), \beta = (y_0 + y_1)$

$\alpha \cdot \beta = x_0y_0 + x_1y_0 + x_0y_1 + x_1y_1$

So $\alpha\beta - x_0y_0 - x_1y_1 = x_1y_0 + x_0y_1$

What do we need to find the overall multiplication?

$x_0y_0 + (\alpha\beta - x_0y_0 - x_1y_1) \cdot 2^{\frac{n}{2}} + x_1y_1 \cdot 2^n$

x_0y_0, x_1y_1 and $\alpha\beta$ are enough to calculate the overall answer! Only 3 multiplies of n/2 bits!

Running Time

$$T(n) = \begin{cases} 3T\left(\frac{n}{2}\right) + O(n) \text{ if } n \text{ is large} \\ O(1) \quad \text{if } n \text{ fits in one byte} \end{cases}$$

$\log_2(3) > 1$,

So running time is $O\left(n^{\log_2(3)}\right)$

Or about $O(n^{1.585})$

You can go faster!

In 2019 (!!!) a paper was published describing an algorithm to multiply numbers in $O(n \log n)$ time

Suspected to be impossible to improve, but not proven yet!

Unfortunately, the algorithm isn't practical. It's defined only when the numbers to multiply have more than 2^{4096} digits.

But it really has been a favorite problem of theoreticians for decades through extremely recently.

Strassen's Algorithm

Apply that "save a multiplication" idea to multiplying matrices, and you can also get a speedup.

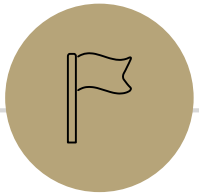
Called Strassen's Algorithm

Instead of $O(n^3)$ time, can get down to $O\left(n^{\log_2 7}\right) \approx O(n^{2.81})$.

Lots of more clever ideas have gotten matrix multiplication even faster **theoretically**.

We even have special notation $O(n^\omega)$ is defined as "the fastest we know how to multiply matrices." It's another favorite problem.

In practice, the constant factors are too high to use the "fastest" algorithms.



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)

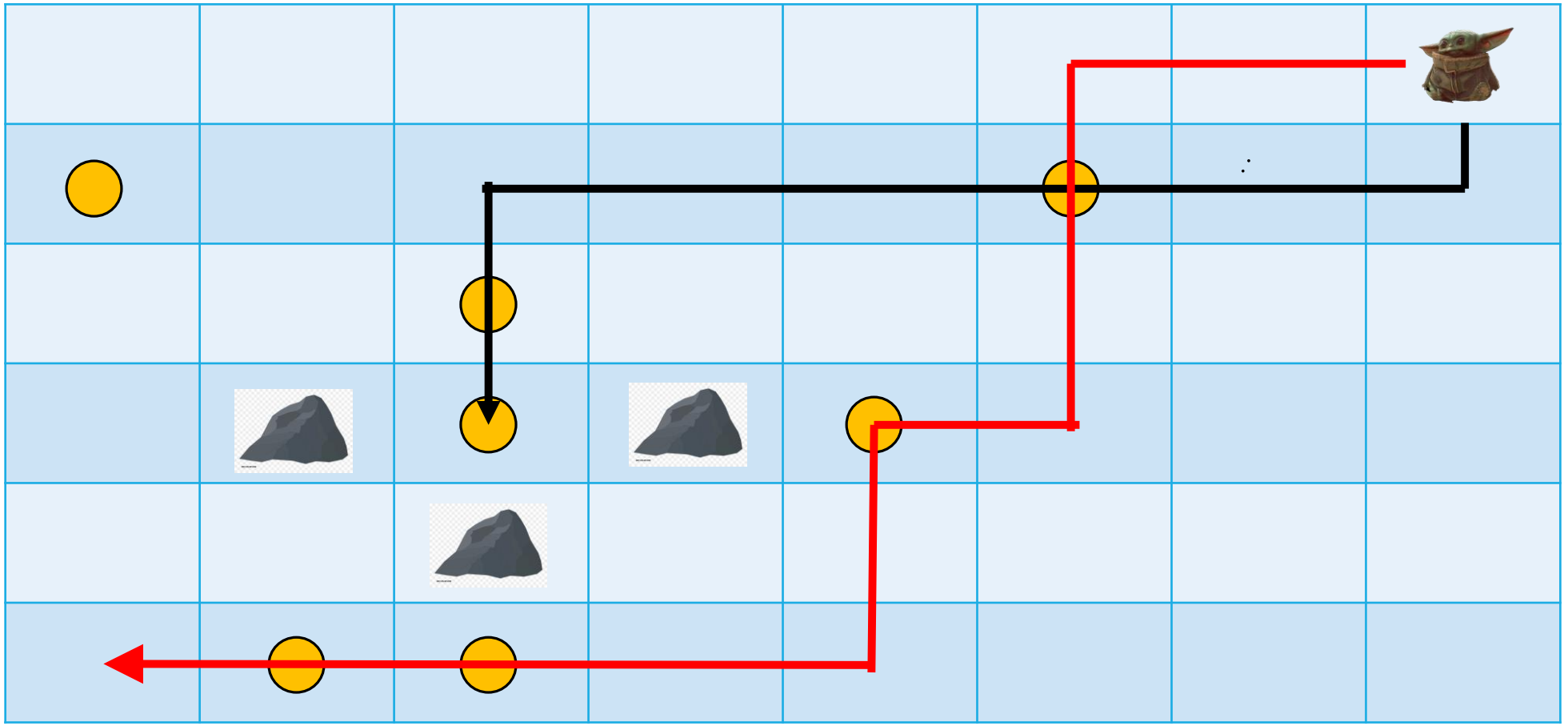
But it is going to make it much easier to do the hard DP problems next week.

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.

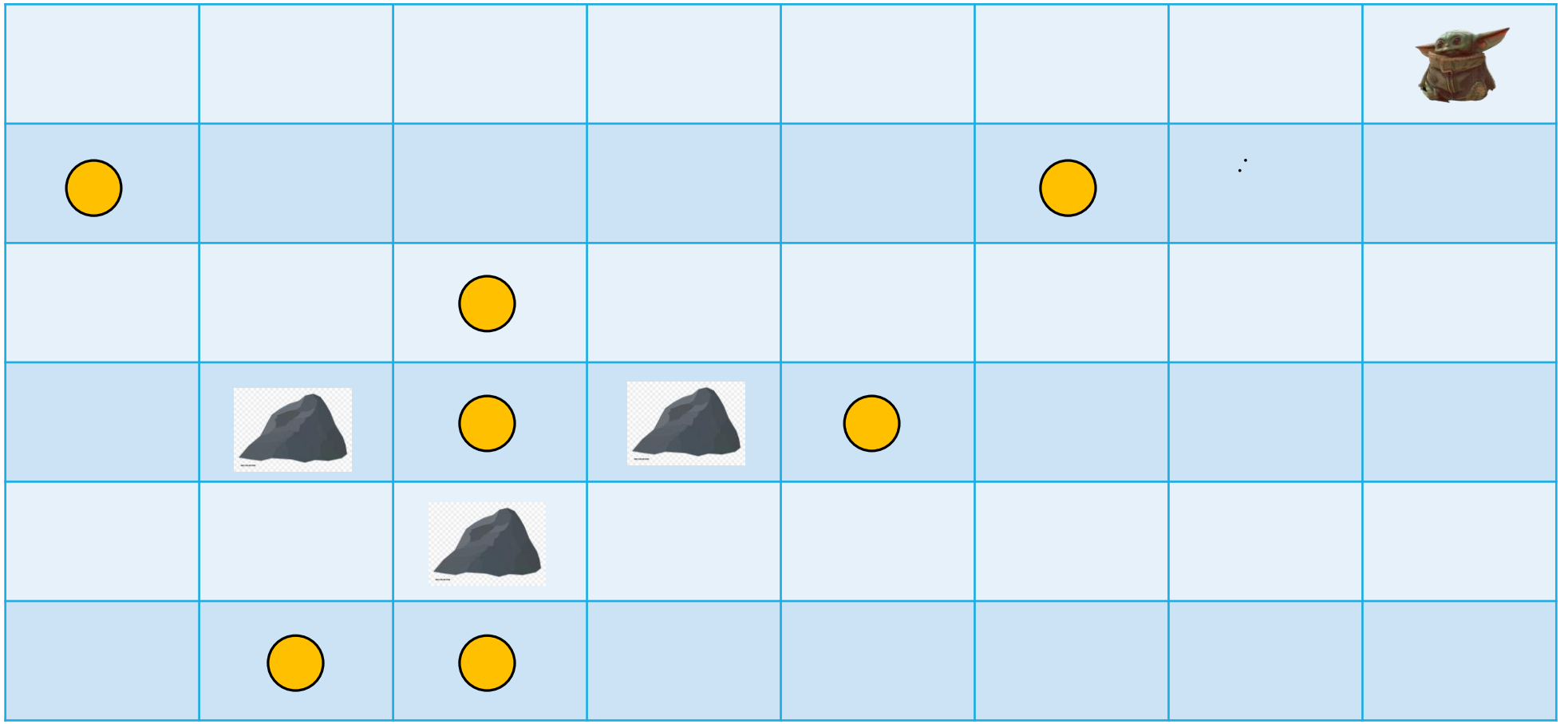
Baby Yoda Searching



Black path: get stuck. Invalid.

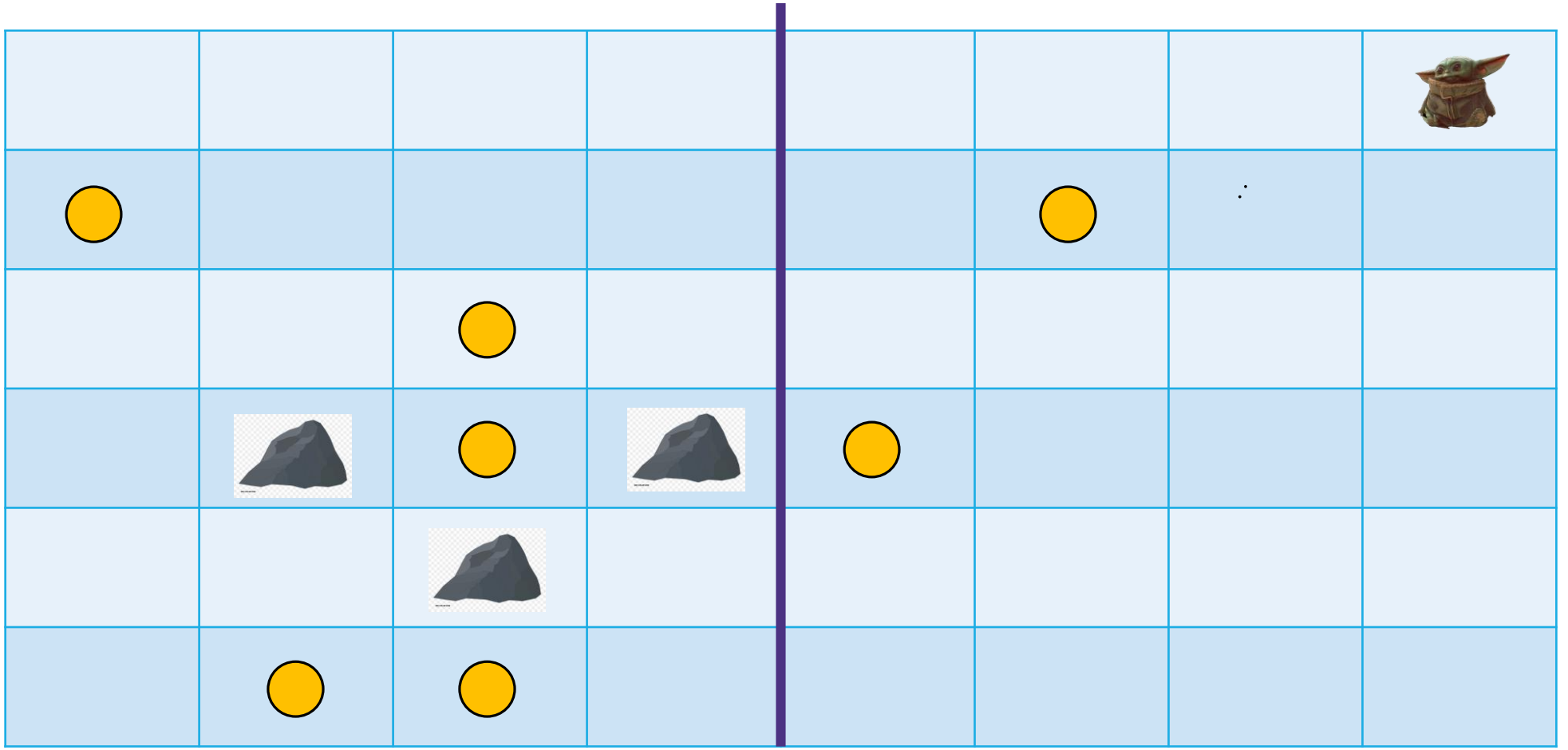
Red path: valid! And optimal (no path collects more than 4 eggs.)

Baby Yoda Searching



Can we represent the problem with a graph?
Yeah, but it doesn't work great (want most eggs, Dijkstra's can only do least eggs)

Baby Yoda Searching

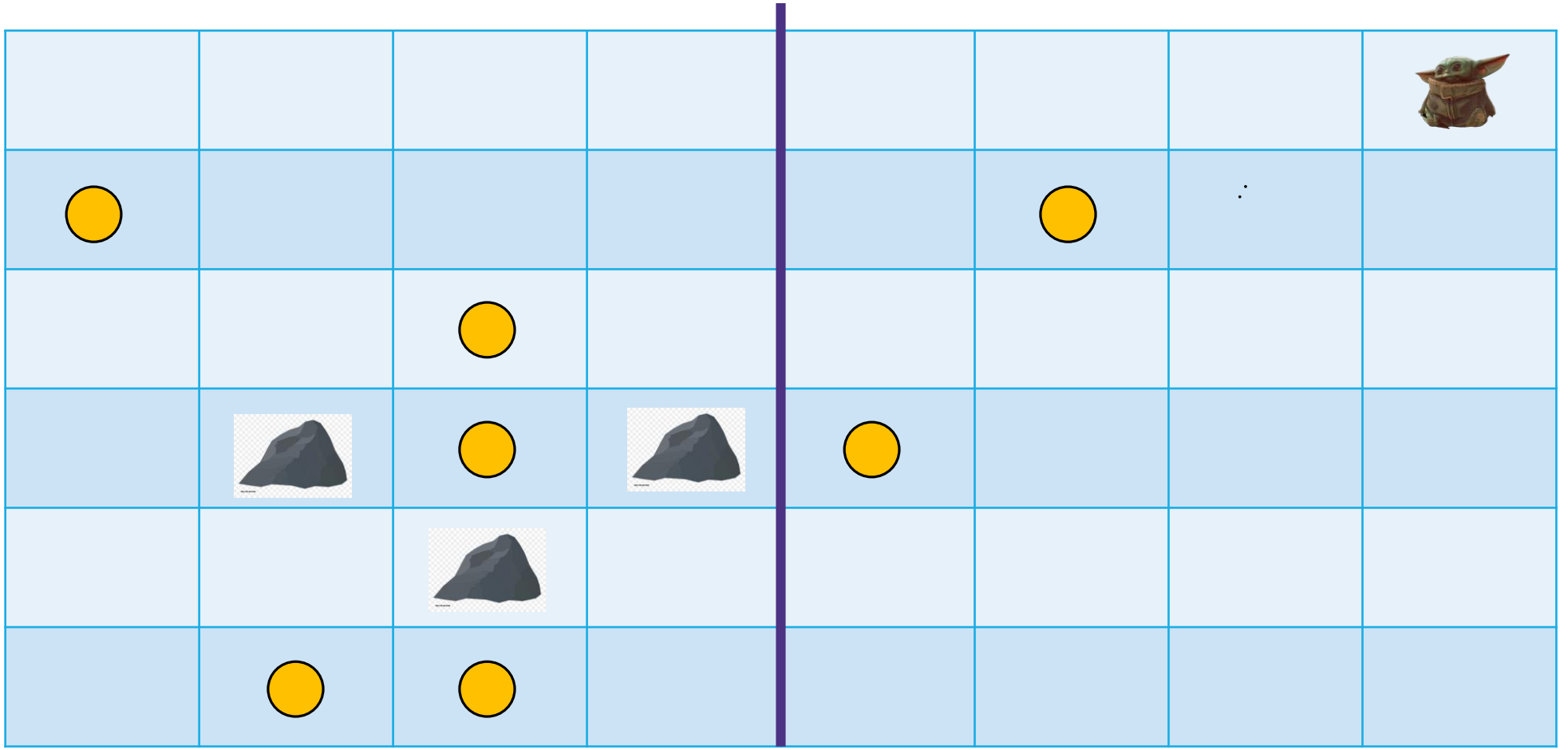


Best left-side path might start at a place inaccessible to end of best right-side path.

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

(0,2)

(0,1)

(1,1)

(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

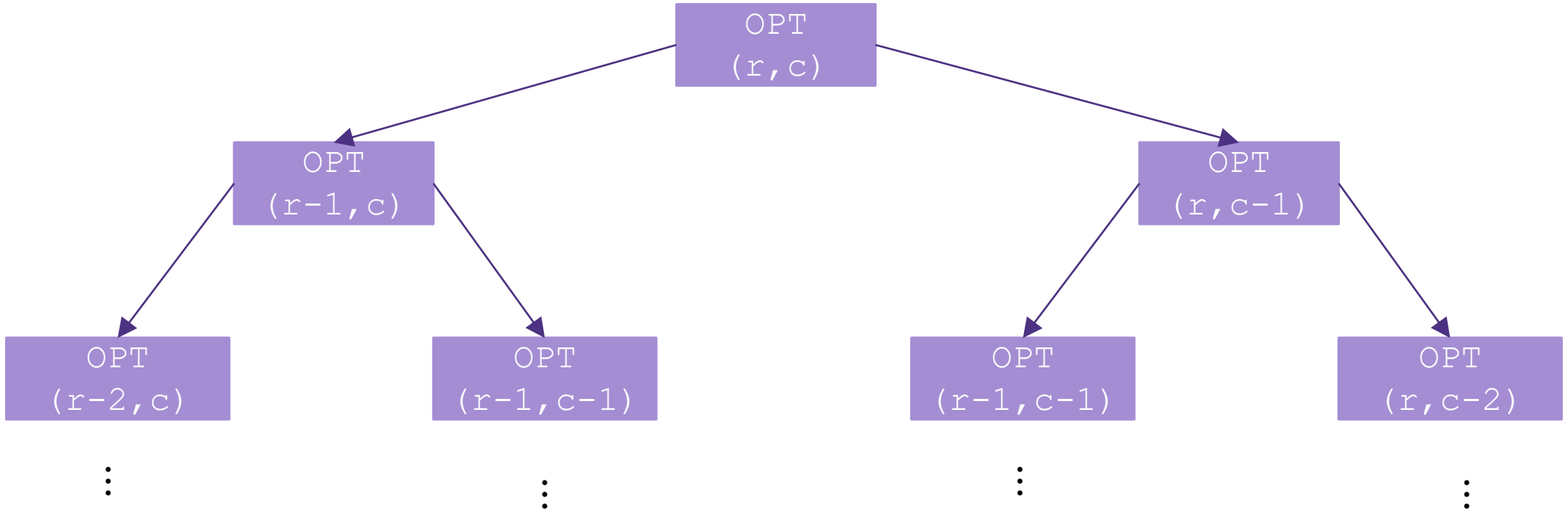
So…how does the code work? What's its running time?

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

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

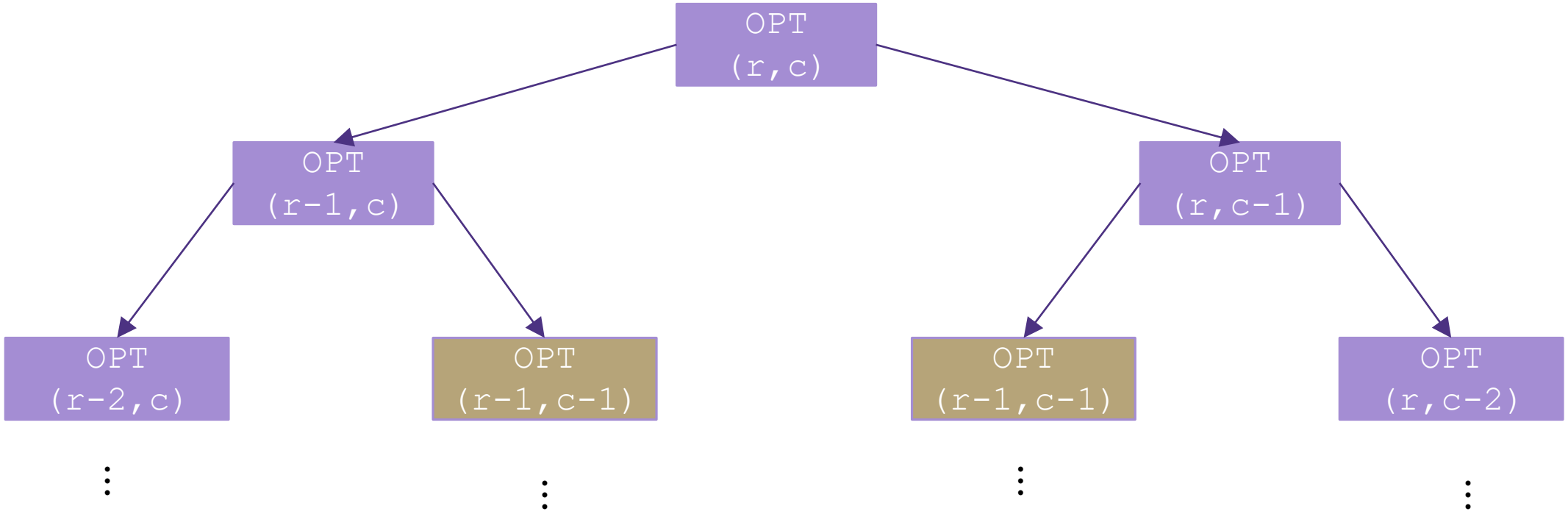
Nodes at level i	2^i	
Work/node	$\Theta(1)$	
Work at level i	$\Theta(2^i)$	
Base Case level	At least $\min(r, c)$	At most $r + c$
Work at base case	$\Omega(2^{\min(r,c)})$	$O(2^{r+c})$
Total work	$\Omega(2^{\min(r,c)})$	$O(2^{r+c})$

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 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

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

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

(0,2)

(0,1)

(1,1)

(0,0)

X-coordinate

0

c-1

Going Bottom-up

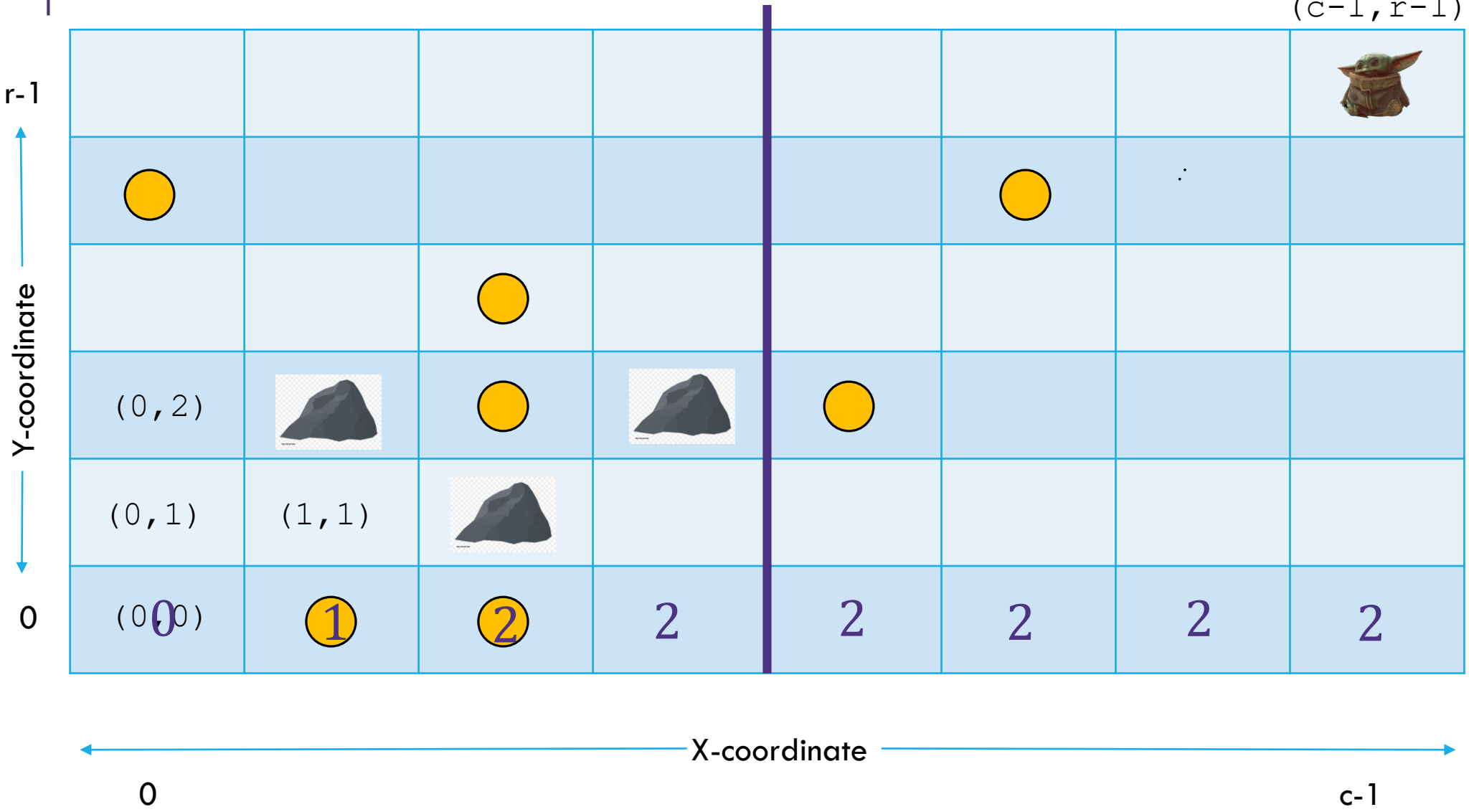
So how does that recursion work?

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

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

Why not just start filling in there?

Baby Yoda Searching



What else can we fill in?

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

(0,2)

(0,1)

(1,1)

0

1

−∞

2

2

2

2

2

0

1

2

2

2

2

2

2

What else can we fill in?

X-coordinate

0

c-1

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

							
							
							
0	−∞						
0	1	−∞	2	2	2	2	2
0			2	2	2	2	2

X-coordinate

0

c-1

What else can we fill in?

Baby Yoda Searching

(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	−∞	−∞	−∞	3	3	3	3
0	1	−∞	2	2	2	2	2
0	1	2	2	2	2	2	2

r-1

Y-coordinate

0

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…