More Dynamic Programming

CSE 421 Spring 26
Lecture 12

Announcements (1)

The midterm exam is less than two weeks away. (Mon. May 4)

Remember it's **in the evening**. 80 minutes, 6-7:20 PM.

We'll be in ARC 147 → will be visible after lecture

Form on the webpage to:

1. Tell us whether you want a left- or right-handed desk

2. Request a conflict, if needed

Request a conflict (work/family responsibilities or something else immovable)

Don't use from if you're sick the night of; email Robbie to request one before the exam starts if you're sick.

Lecture Mon May 4 will be Office Hours style

Announcements (2)

We'll have a reference sheet for you for the midterm (including, e.g., a list of some algorithms from 332 that you've been able to reference)

You'll also be able to bring your own 8.5x11 inch piece of paper with handwritten notes.

Practice materials, topics list, contents of the provided reference, etc. all on the webpage.

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.

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.

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 $\Theta(rc)$.

Why? There are that many spots, each is calculated at most once and looked up at most twice.

Going Bottom-up

So how does that recursion work?

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

`OPT[0][0]`

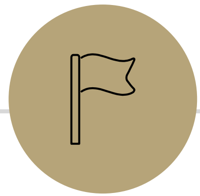
Why not just start filling in there?

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]
```



Robustness

Updating the Problem (new statement)

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 (ans)

$OPT(i,j,f)$ 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 to an int, here. (subtract 1 if you would need to knock over rocks)

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

Filling in (1)

(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)

Filling in (2)

`(c-1,r-1)`

Y-coordinate								
r-1	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
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)

Filling in (3)

(c-1, r-1)

Y \ X	0							c-1
r-1	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	?/?	?/?	?/?	?/?	?/?	?/?	?/?	?/?
	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?
Again from left to right, bottom to top, now filling in

Filling in (4)

	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.

Filling in (5)

f=0 f=1

(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	−∞/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 (Baby Yoda)

1. Define the object you're looking for

$OPT(i, j, f)$ is the maximum number of eggs that can be collected on a path from i, j to $0,0$ using the force to move rocks at most f times

2. Write a recurrence to say how to find it

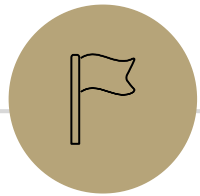
(see prior slides)

3. Design a memoization structure

A 3D array: r x c x $(f + 1)$

4. Write an iterative algorithm

We'll omit for this version---it's a loop over f, with loop over j inside, with a loop over i inside.



Bells,Whistles, and optimiziation

Which Direction?

(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

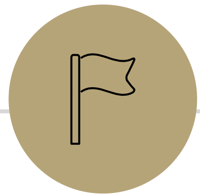
So should Baby Yoda go left or down?

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. It will also tell you whether to use the force or not (was the parameter in that recursive call the same as your value or one less?)

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



Memory Optimization

Optimizing Memory

Do we need all that memory?

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

What Recursive Calls do we need?

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

What values do we need to keep around?

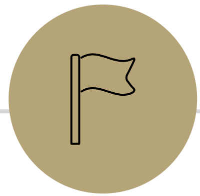
What Calls do we need (visually)

(c-1, r-1)

Y-coordinate	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	$-\infty$	$-\infty$	$-\infty$	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.



More Problems

Maximum Contiguous Subarray Sum

We saw an $O(n \log n)$ divide and conquer algorithm.

Can we do better with DP?

Given: Array $A[]$

Output: i, j such that $A[i] + A[i+1] + \cdots + A[j]$ is maximized.

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

Maximum Contiguous Subarray Sum (define opt)

We saw an $O(n \log n)$ divide and conquer algorithm.

Can we do better with DP?

Given: Array $A[]$

Output: i, j such that $A[i] + A[i+1] + \cdots + A[j]$ is maximized.

For today: just output the value $A[i] + A[i+1] + \cdots + A[j]$.

Define `OPT(i)`.

OPT(i): sum of max sum subarray of A[0]... A[i]

Trying to Recurse

0	1	2	3	4	5	6	7
5	-6	3	4	-5	2	2	4

$OPT(3)$ would give i =2, j = 3

$OPT(4)$ would give $i = 2, j = 3$ too

$OPT(7)$ would give $i = 2, j = 7$ – we need to suddenly backfill with a bunch of elements that weren't optimal…

How do we make a decision on index 7? What information do we need?

What do we need for recursion?

If index i IS going to be included

We need the best subarray **that includes index $\boldsymbol{i-1}$**

If we include anything to the left, we'll definitely include index $i - 1$ (because of the contiguous requirement)

If index i isn't included

We need the best subarray up to $i - 1$, regardless of whether $i - 1$ is included.

Two Values

Pollev.com/robbie

Need two recursive values:

$INCLUDE(i)$: sum of the maximum sum subarray among elements from 0 to i **that includes index $\boldsymbol{i}$** in the sum

$OPT(i)$: sum of the maximum sum subarray among elements 0 to i (that might or might not include i)

How can you calculate these values? Try to write recurrence(s), then think about memoization and running time.

Recurrences

$$INCLUDE(i) = \begin{cases} \max\{A[i], A[i] + INLCUDE(i-1)\} & \text{if } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$OPT(i) = \begin{cases} \max\{INCLUDE(i), OPT(i-1)\} & \text{if } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

If we include i, the subarray must be either just i or also include $i - 1$.

Overall, we might or might not include i. If we don't include i, we only have access to elements $i - 1$ and before. If we do, we want $INCLUDE(i)$ by definition.

Example (1)

A	0	1	2	3	4	5	6	7
	5	-6	3	4	-5	2	2	4

$OPT(i)$	0	1	2	3	4	5	6	7
	5							

$INCLUDE(i)$	0	1	2	3	4	5	6	7
	5							

Example (2)

A	0	1	2	3	4	5	6	7
	5	-6	3	4	-5	2	2	4

$OPT(i)$	0	1	2	3	4	5	6	7
	5	5						

$INCLUDE(i)$	0	1	2	3	4	5	6	7
	5	-1						

Example (3)

A	0	1	2	3	4	5	6	7
	5	-6	3	4	-5	2	2	4

$OPT(i)$	0	1	2	3	4	5	6	7
	5	5	5	7	7	7	7	10

$INCLUDE(i)$	0	1	2	3	4	5	6	7
	5	-1	3	7	2	4	6	10

Pseudocode (subarray sum)

```
int maxSubarraySum(int[] A)
  int n=A.length
  int[] OPT = new int[n]
  int[] Inc = new int[n]
  inc[0]=A[0]; OPT[0] = max{A[0],0}
  for(int i=0;i<n;i++)
    inc[i]=max{A[i], A[i]+inc[i-1]}
    OPT[i]=max{inc[i], opt[i-1]}
  endFor
return OPT[n-1]
```

Recursive Thinking In General (top-down)

As before, the hardest part is designing the recurrence.

It sometimes helps to think from multiple different angles.

Top-down: What's the first step to take?

Baby Yoda will first go left or down. Use recursion to find out which of left or down is better.

Should I include this element or not? Use recursion to see whether it should be included (then you must include i-1 if you have anything else) or not (just OPT(i-1)

Recursive Thinking In General (bottom-up)

Bottom-Up: What information could a recursive call give me that would help?

How does a path through most of the map help Baby Yoda?

Well we just need to know the values one left and one down.

The answers to which subarrays would let us solve the problem?

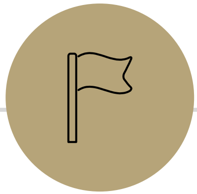
Well, if we know the best option to our left, we could insert one more element or not…but if we insert one more element, we need the best option to our left including the second-to-last element.

Recursive Thinking In General (optimal substructure)

Some people refer to the "Optimal Substructure Property"

From the optimum (most eggs, greatest sum) for a slightly smaller problem (Baby Yoda starting closer to the end, slightly smaller array), we need to be able to build up the optimum for the full problem.

This property is true of INCLUDE, but not (directly) of OPT---that's why we added INCLUDE.



Our Next Problem

Longest Increasing Subsequence

Longest Increasing Subsequence

0	1	2	3	4	5	6	7
5	-6	3	6	-5	2	8	10

Longest set of (not necessarily consecutive) elements that are increasing

5 is optimal for the array above

(indices $1,2,3,6,7$; elements $-6,3,6,8,10$)

Convince yourself that a greedy algorithm doesn't work!

For simplicity – assume all array elements are distinct.