7/25 Lecture Agenda

- Announcements
- Part 5-1: Bellman-Ford
- 10 minute break!
- Part 5-2: Intro to Dynamic Programming

Announcements: Remaining Assignments

Remaining assignments shifted back to avoid overlap (due dates updated on site). *(These are firm release dates)*

- Pre-HW5 out this Wednesday evening (due Weds. next week) – to give a bit more time for remaining midterm grading on Mon
- HW5 out this Friday evening (due Fri. next week)
- Pre-HW6 out Weds. next week (due Weds. of last week)
- HW6 out Fri. next week (due Thu. of last week, but can still use late days until Sat. as usual)
  - this one will be less time-consuming and will designed to be quick to grade (e.g., give straightforward or numerical answers, no proofs), submission on Gradescope

Announcements: The Final

- Friday, August 12, 3:30-6:30 PM
- Unlike the midterm, our final exam is in **200-002** (Lane History Corner, the southeast corner of the Main Quad)
- The final will cover all six units, but the Special Topics lecture has been removed from scope and is now totally optional.
- HW4 has a question where you can share feedback on the midterm and suggestions for the final

Announcements: Grades and Friday's deadline

- Midterm grades will be released on Wednesday (mostly graded, just waiting for remote ones)
- HW2 grading will be complete by Thursday, and then I'll release some overall context to give you a better idea of where you stand
- The Summer Quarter deadline for withdrawal or grading basis change is this Friday at 5 PM

7/25 Lecture Agenda

- Announcements
- Part 5-1: Bellman-Ford
- 10 minute break!
- Part 5-2: Intro to Dynamic Programming

WORLD 5-1

Bellman-Ford: Beyond Dijkstra

Divide and Conquer
Sorting & Randomization
Data Structures
Graph Search
Dynamic Programming
Greed & Flow

Special Topics

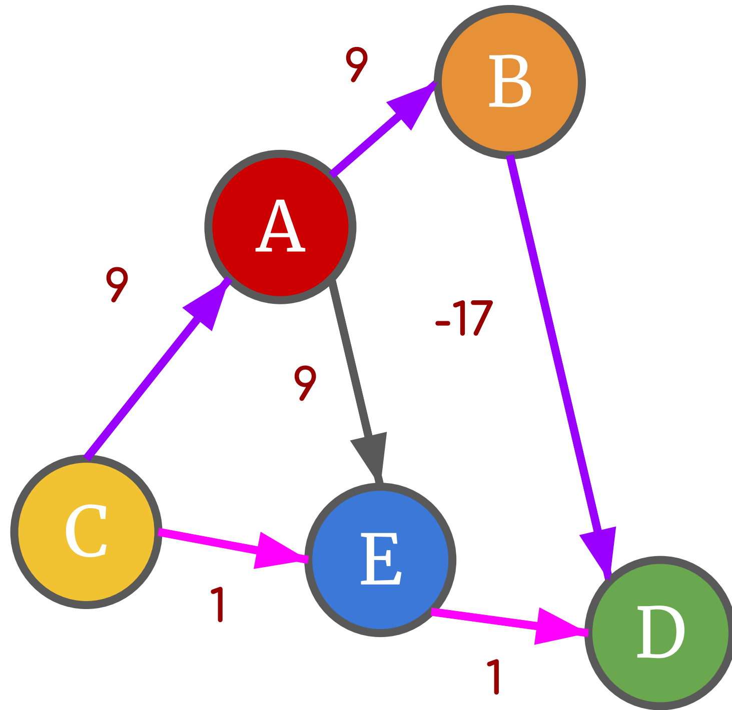
Flashback to

12 DAYS AGO

which, because we're at Stanford, already seems like

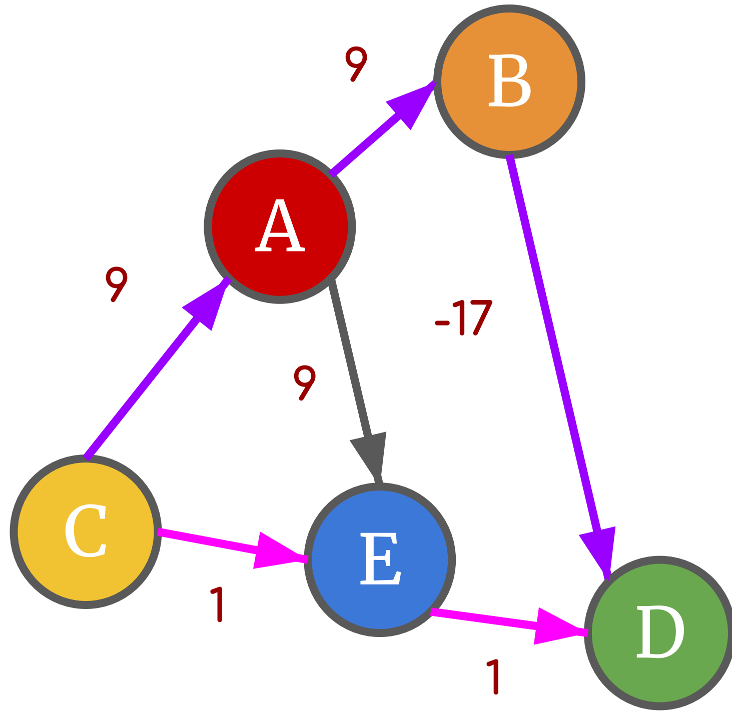
12 WEEKS AGO

What's wrong with negative edge weights?



For one thing, we may stop too early, with an answer that is too big!

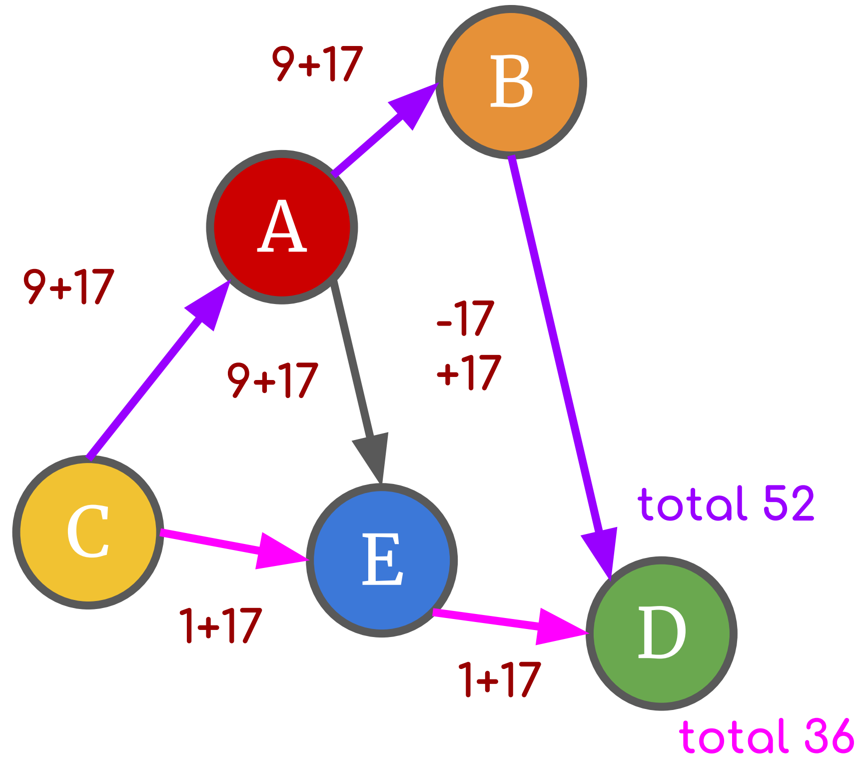
(Here we get 2 for C to D, but the answer is 1)



wait, weren't those edges undirected last time we saw this?

Yes – but a negative undirected edge makes the solution arbitrarily small (just go back and forth), so we're focusing on directed graphs now.

What's wrong with negative edge weights?

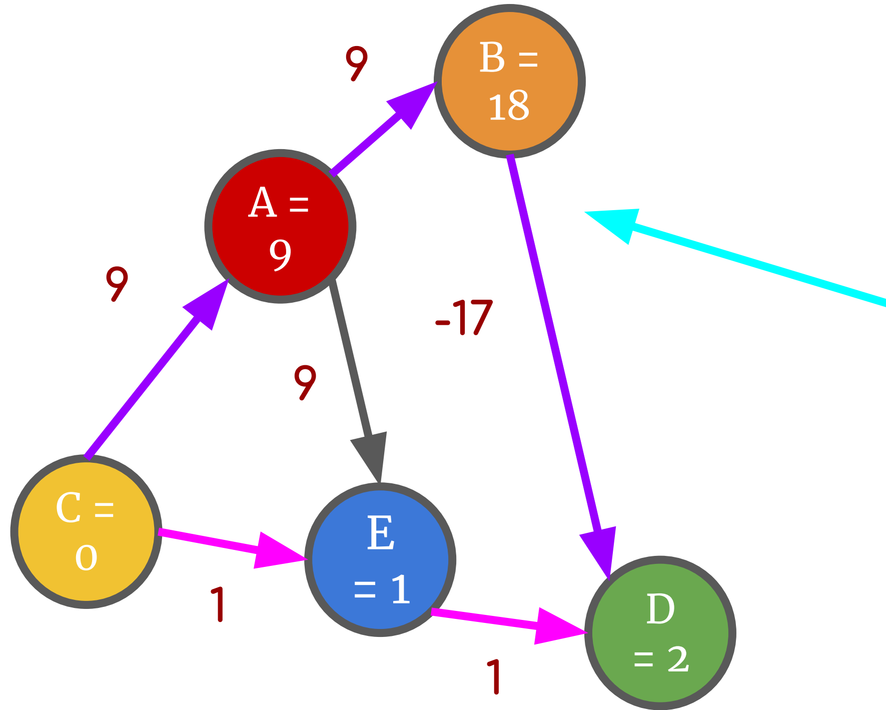


A possible fix:

- Add the most negative weight to every edge.
- Run the algorithm.
- Subtract off the added weights at the end.

Does this work? **No! We still end up picking the bad path because it has fewer steps.**

We could have asked: Why not just check again?



Dijkstra's goes through the entire graph, right? So when we're inspecting this edge...

why not just re-check our estimate for D, even though we called it "sure"?

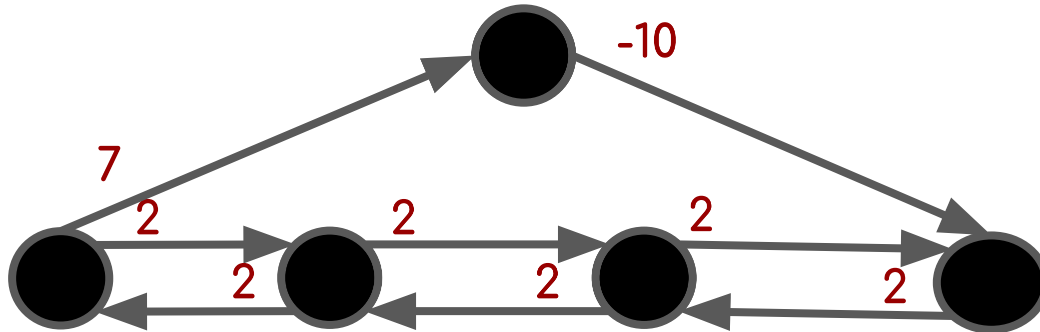
So why *not* just check again?

- OK, so sometimes an edge makes us check and update a node again.

- Wouldn't the running time still be bounded by the number of edges, like before?

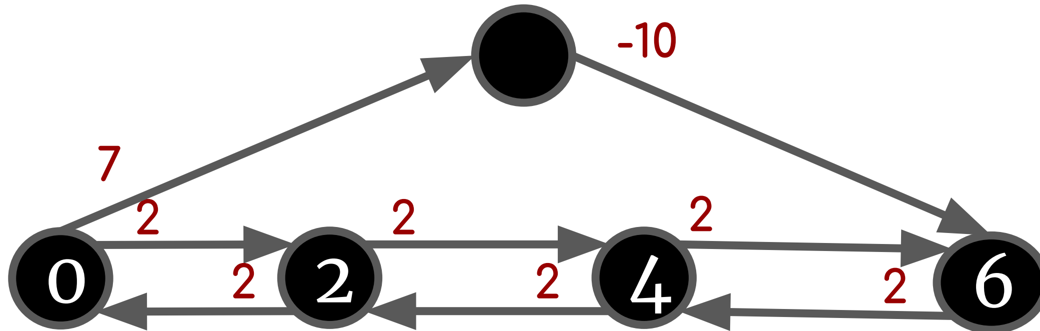
It's even worse than that!

- We might need to fix almost every estimate in the graph!



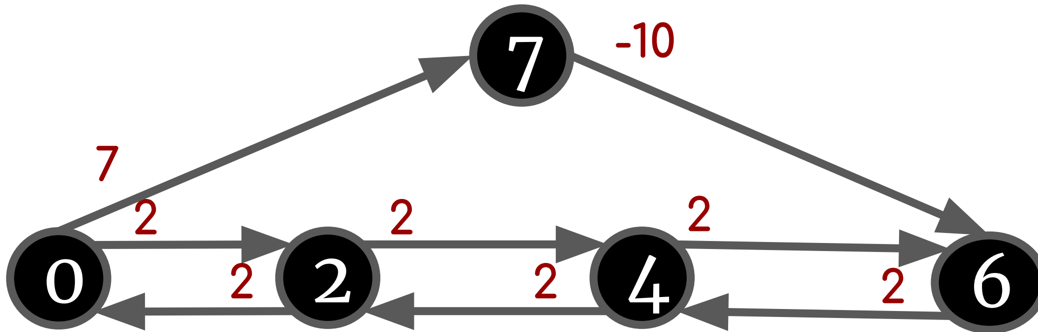
It's even worse than that!

- We might need to fix almost every estimate in the graph!



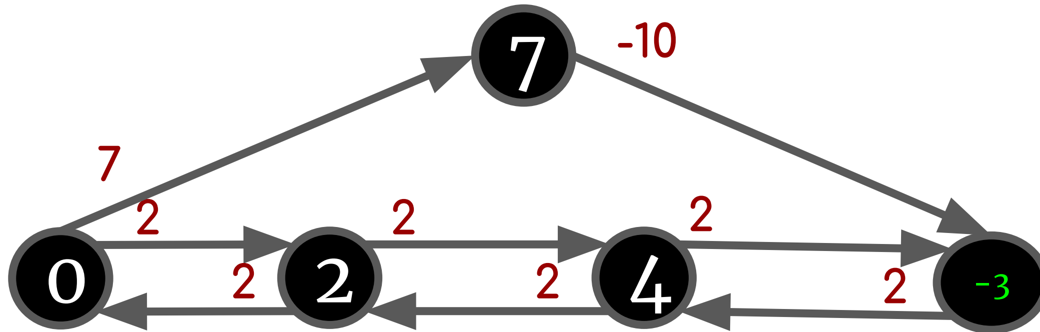
It's even worse than that!

- We might need to fix almost every estimate in the graph!



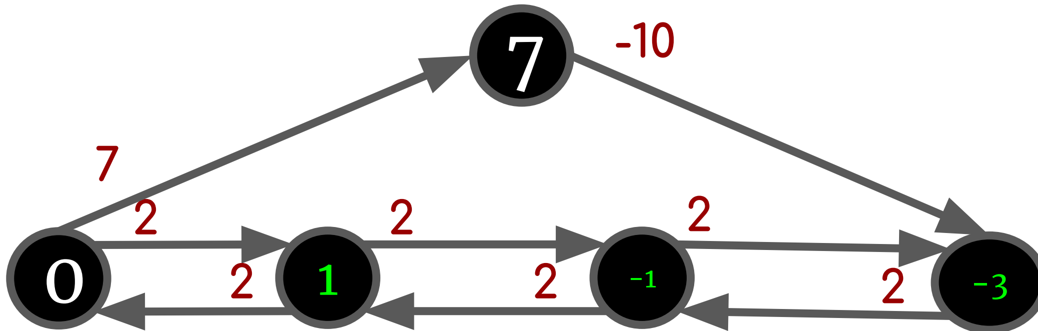
It's even worse than that!

- We might need to fix almost every estimate in the graph!



It's even worse than that!

- We might need to fix almost every estimate in the graph!



Negative edges break everything...

~~...but we can fix it by shifting the edge costs?~~

~~...or by just checking the estimates again?~~

OK I bet they're not even important anyway!

Negative edges break everything...

~~...but we can fix it by shifting the edge costs?~~

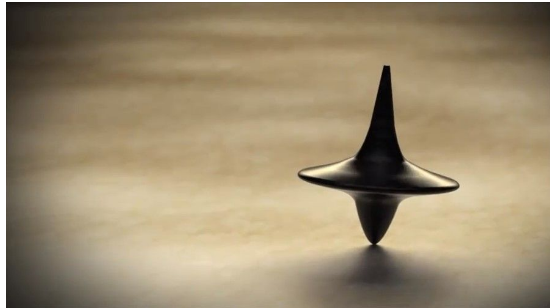
~~...or by just checking the estimates again?~~

~~OK I bet they're not even important anyway!~~

- *Profits and costs for performing certain actions!*
- *Elevation! An example from StackExchange: electric cars use energy on uphills, regenerate on downhills*

OK, so what do we do?

- We need to handle negative edges correctly and realize when we're in a negative cycle.



If the top is still spinning, we're stuck in an infinite negative loop!

- We can't just look at every possible path/cycle! There could be something like $O(m^n)$ of those.

The key observation

- In each step of Dijkstra's, we only "process" edges originating from the current node we're looking at (the one with the lowest estimate).
    - "Process" = if using the edge improves the estimate for the edge's destination node, we update that estimate.

The key observation

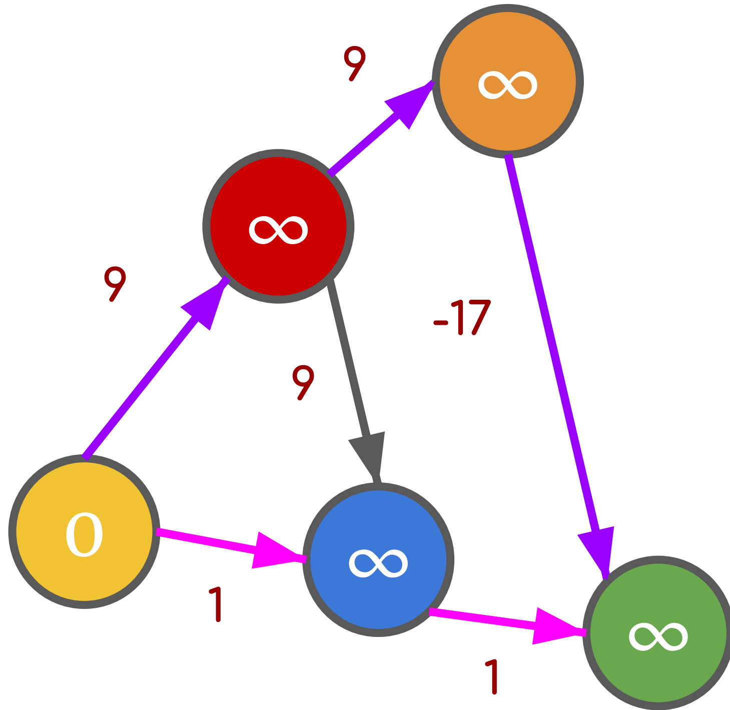
- In each step of Dijkstra's, we only "process" edges originating from the current node we're looking at (the one with the lowest estimate).
    - "Process" = if using the edge improves the estimate for the edge's destination node, we update that estimate.
- What if we just process *all* the edges at each step?

How would this help with negative edges?

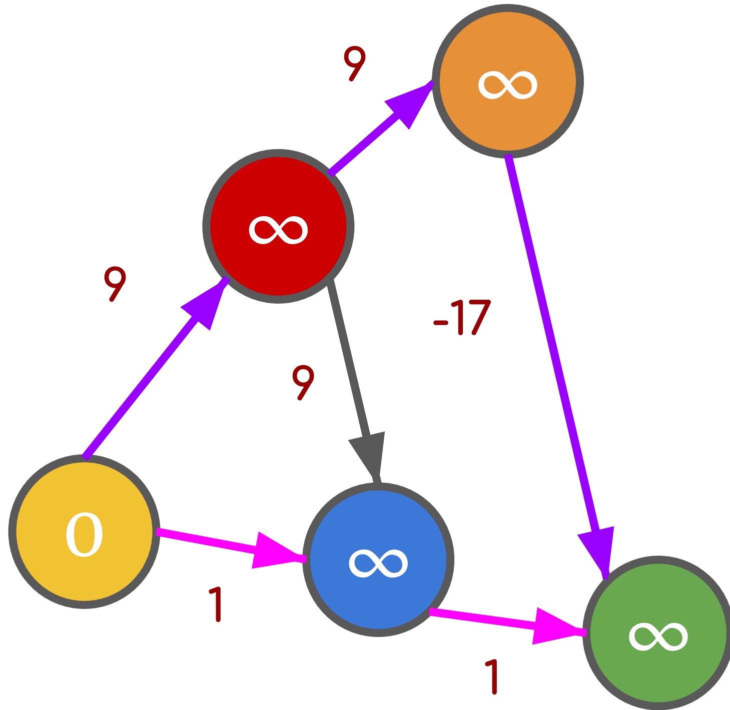
- Intuitively, there are no more nasty surprises where we prematurely call a node "done", but there's a better path (with a negative edge) that we missed.

- If there are no negative cycles, the longest possible path is n-1 steps (edges) long. So if we process all edges n-1 times, we can't miss the best path.

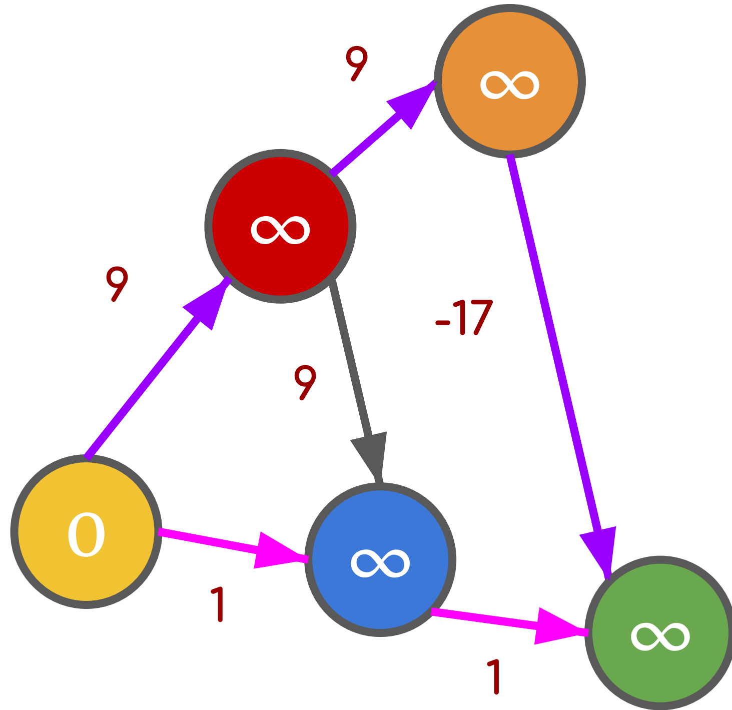
Step 0



Step 1...



Wait a minute



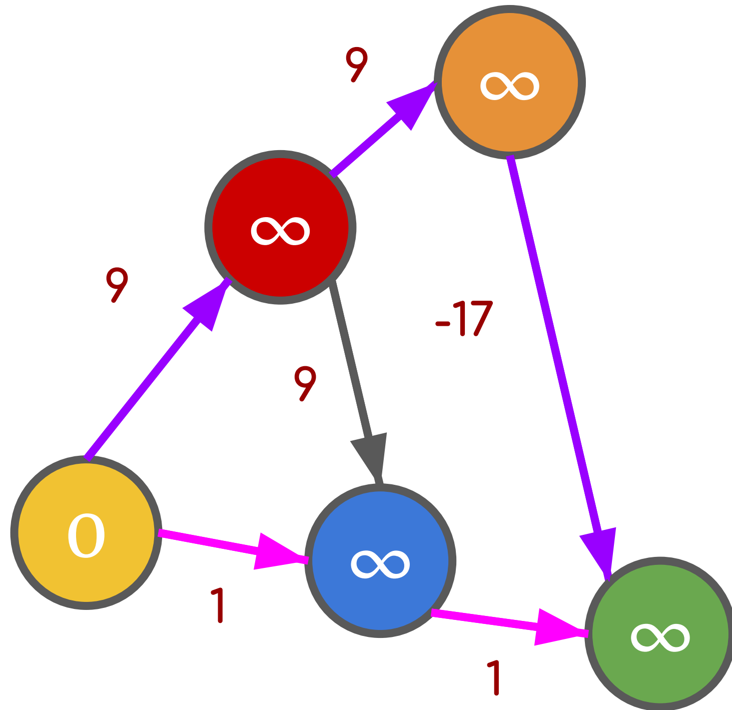
What order should we visit the edges in?

*Aren't there better and worse ones? e.g., what if we happen to pick a sequence of edges that **is** the best path to some node?*

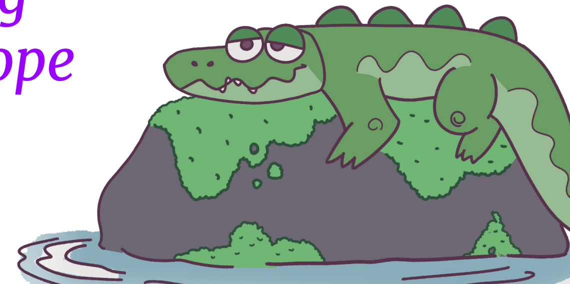
Wait a minute

What order should we visit the edges in?

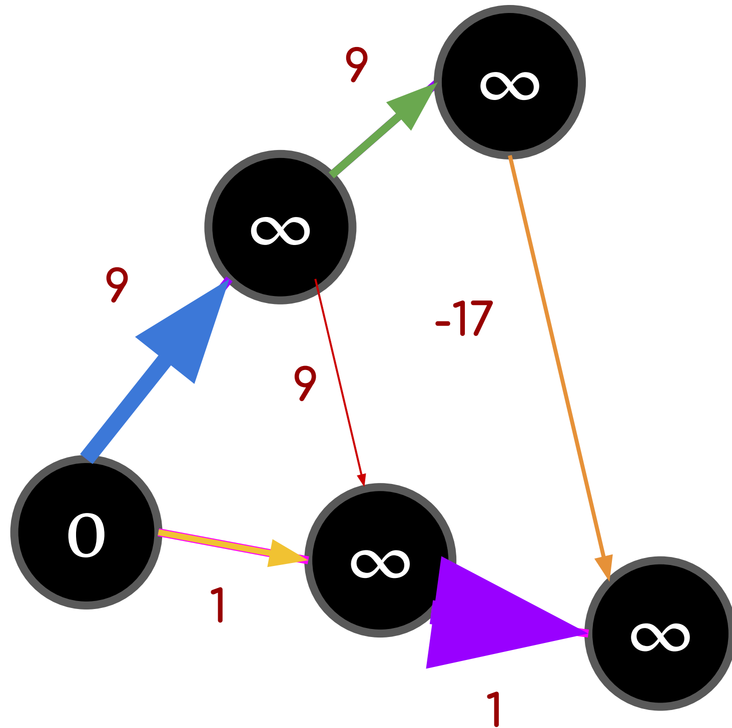
*Aren't there better and worse ones? e.g., what if we happen to pick a sequence of edges that **is the best path to some node?***



Just do something arbitrary! And hope it'll all work out.



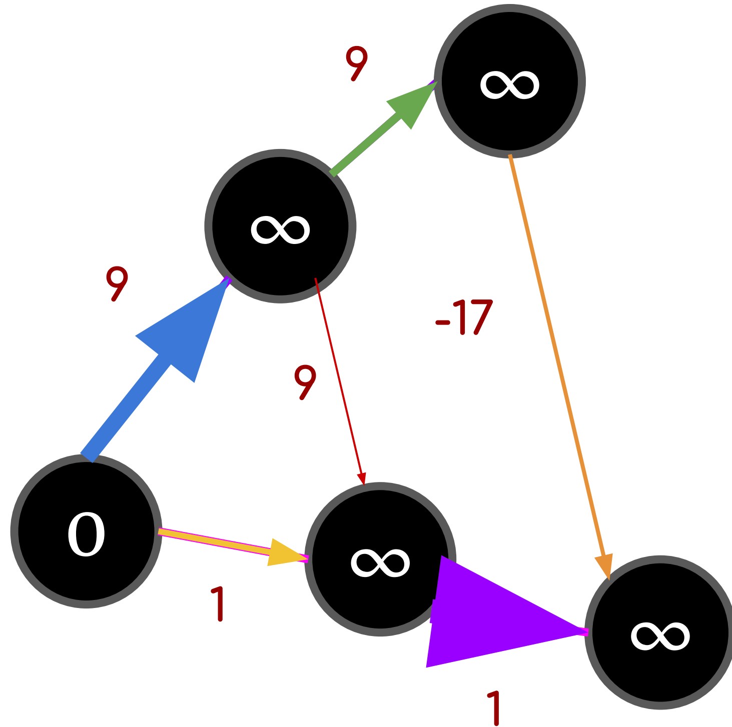
Step 1



Now edges are colorful instead of nodes, to indicate an order (which I picked at random!)

(If you can't distinguish all the colors, the order is also from thinnest to thickest.)

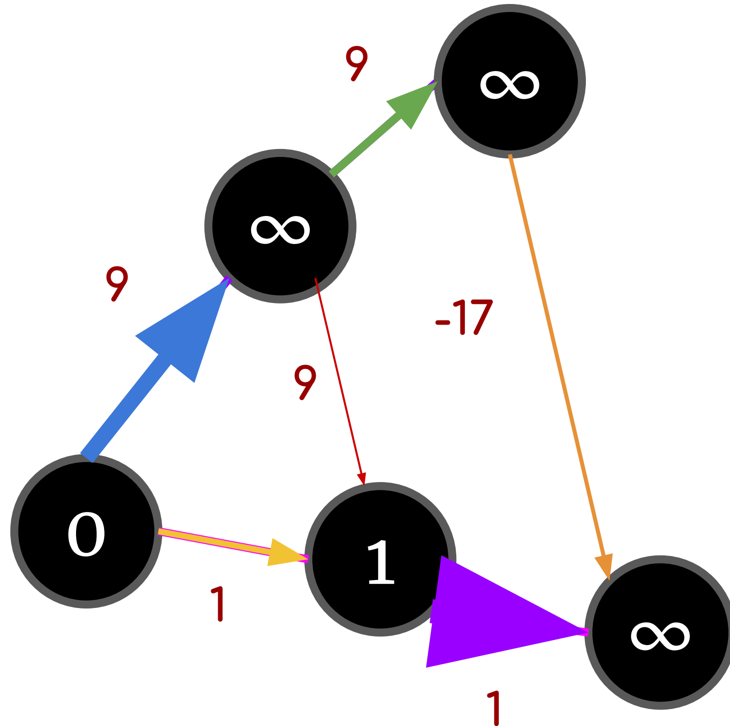
Step 1



Red: source is infinity, ignore

Orange: source is infinity, ignore

Step 1

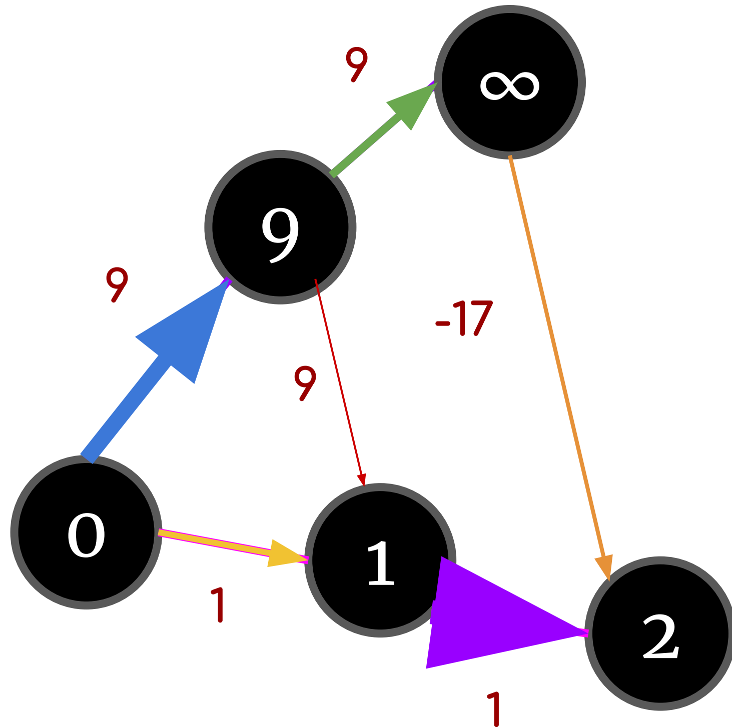


Red: source is infinity, ignore

Orange: source is infinity, ignore

Yellow: following edge reduces destination estimate from ∞ to 1.

Step 1

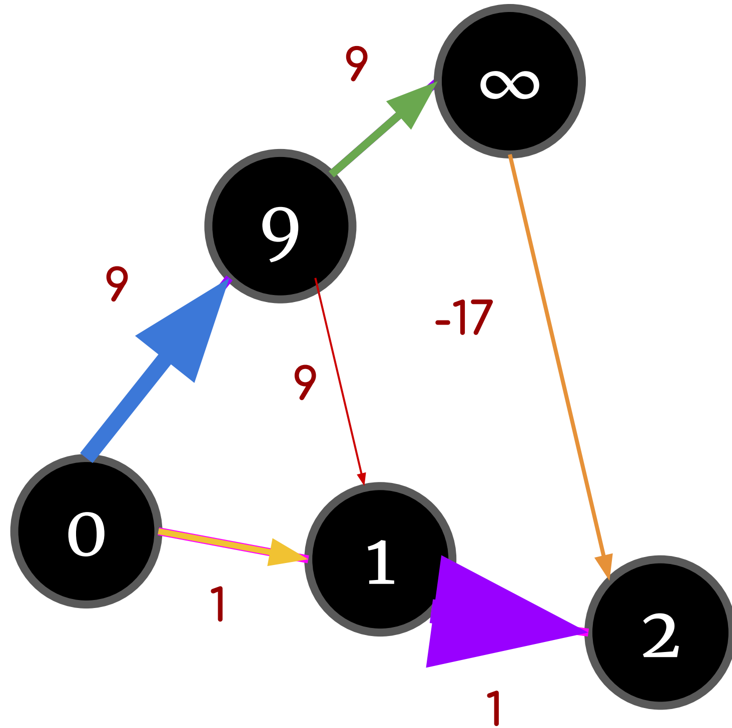


Green: source is infinity, ignore

Blue: following edge reduces destination estimate from ∞ to 9.

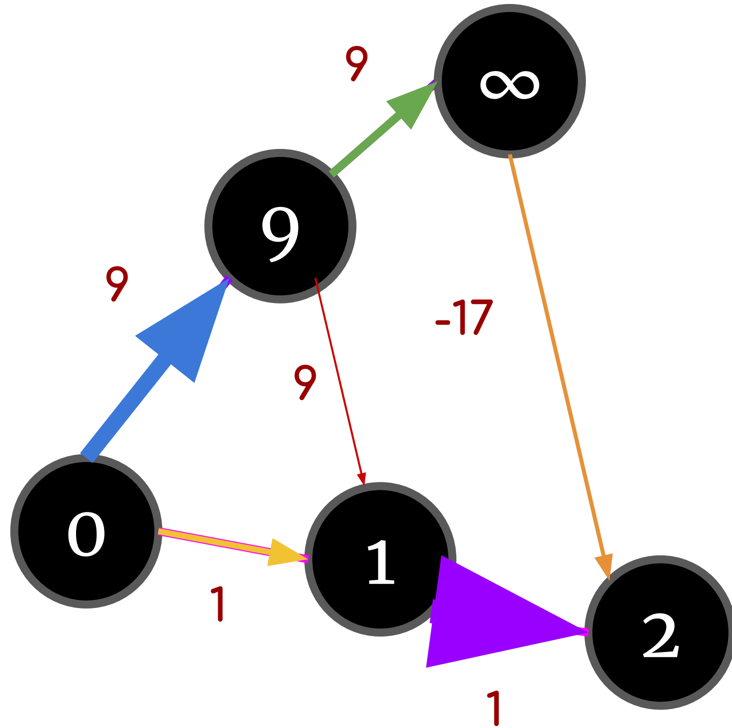
Purple: following edge reduces destination estimate from ∞ to 2.

Step 1



But 2 was the bad answer for that node! Didn't we just cause the same problem?!?!?

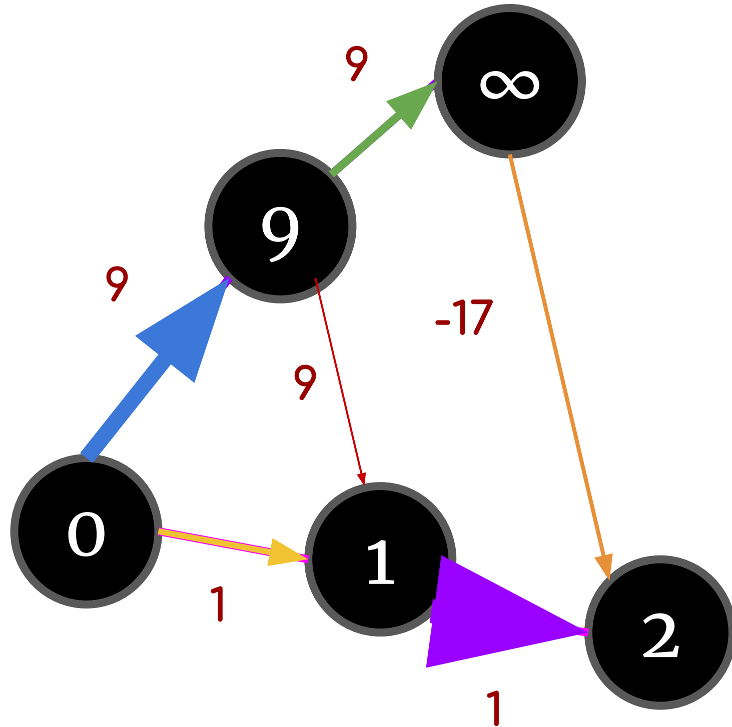
Step 1



But 2 was the bad answer for that node! Didn't we just cause the same problem?!?!?

Bellman-Ford isn't done yet! It's just getting started...

Step 2

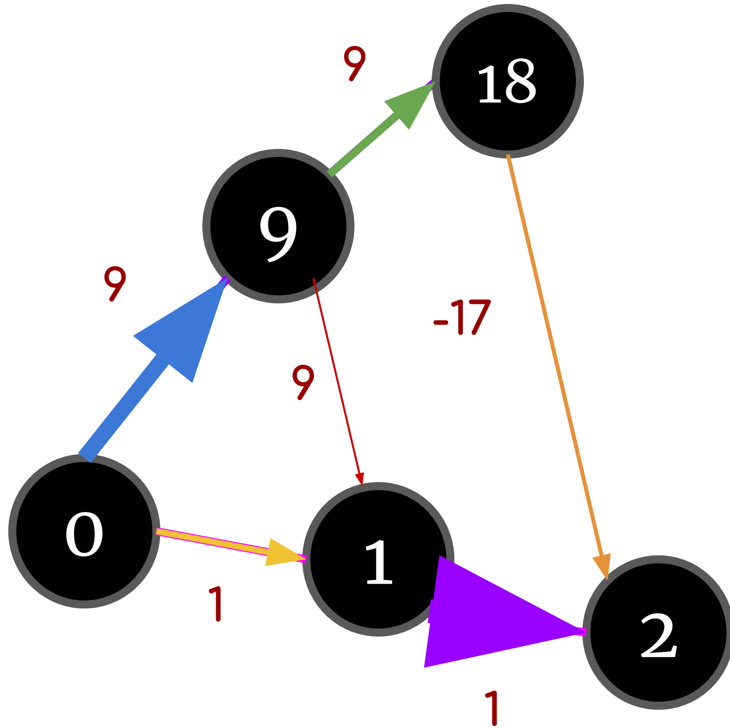


Red: following edge gives estimate of 18, not better than 1, ignore

Orange: source is infinity, ignore

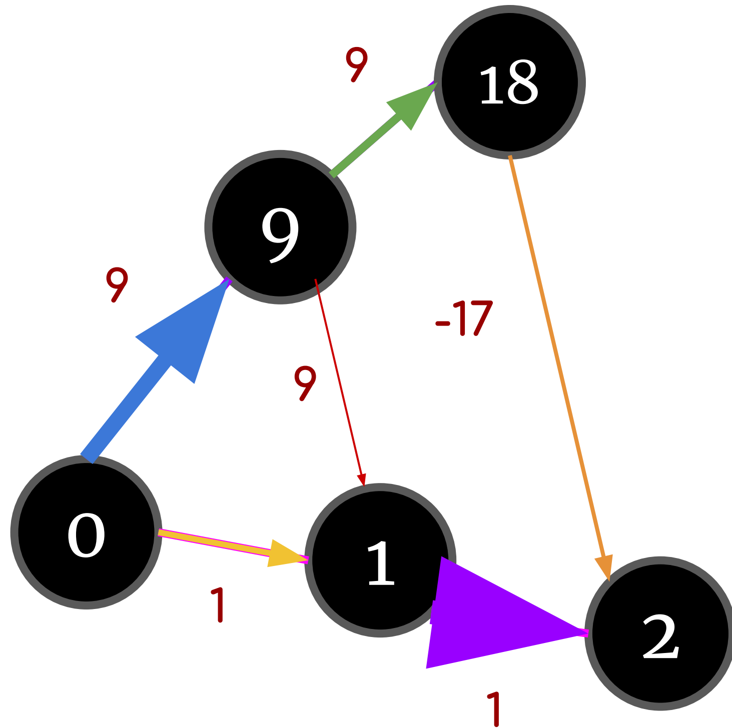
Yellow: following edge gives estimate of 1, not better than 1, ignore

Step 2



This is just checking the same stuff over and over!

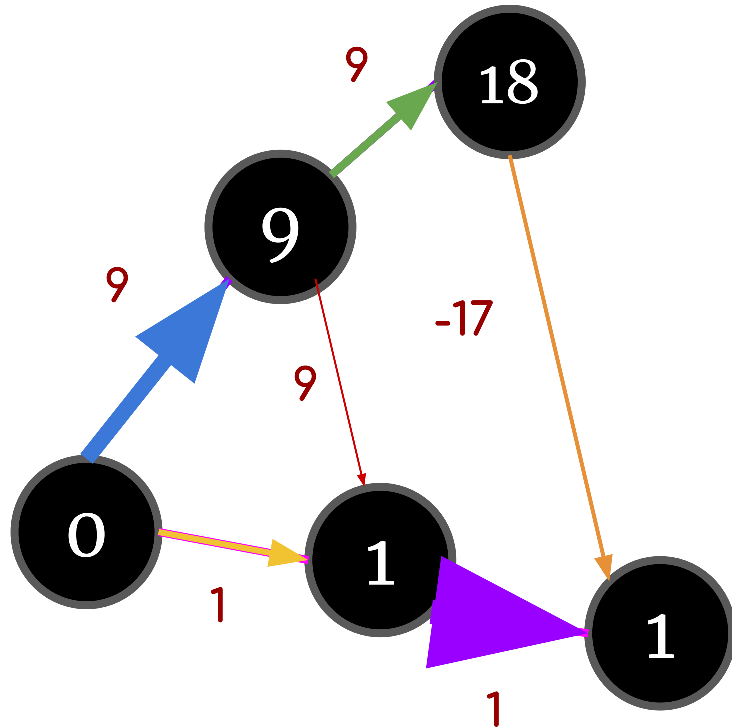
Step 2



This is just checking the same stuff over and over!

True, but so is guard duty. And that's the whole point. We have to be **EVER VIGILANT** for changes that would need to be propagated forward to other nodes!

Step 3

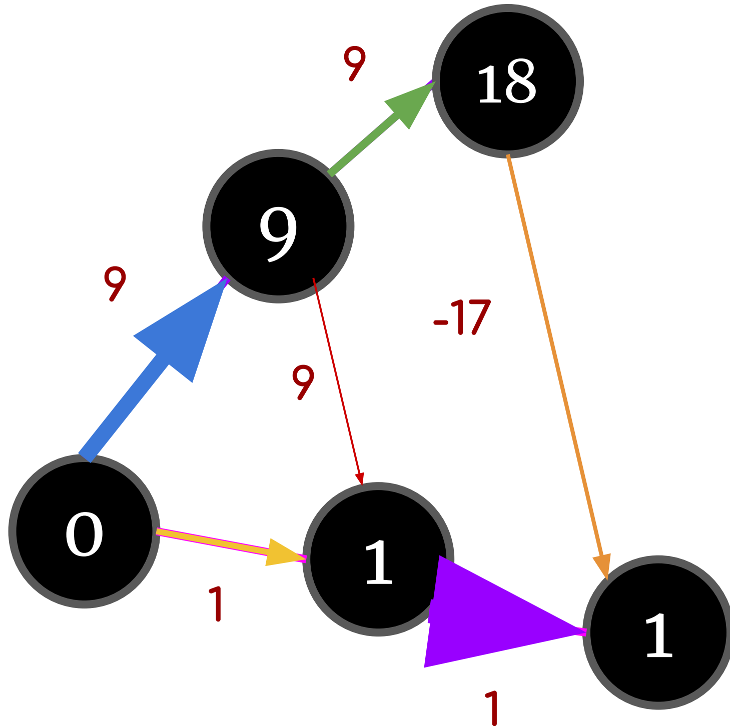


Red: following edge gives estimate of 18, not better than 1, ignore

Orange: following edge reduces estimate from 2 to 1

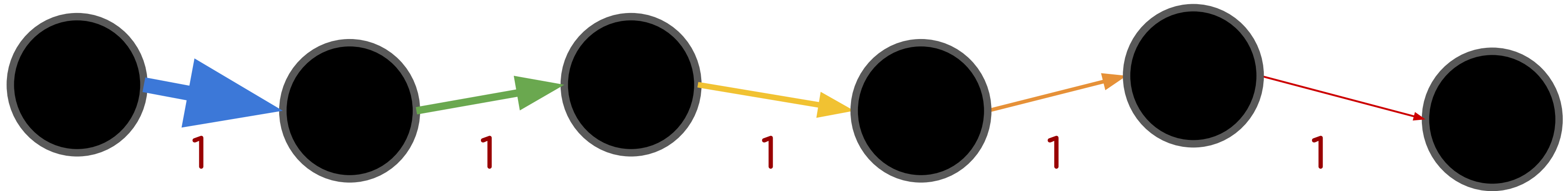
YAY!!!!!!!

Uh... the rest of Step 3?

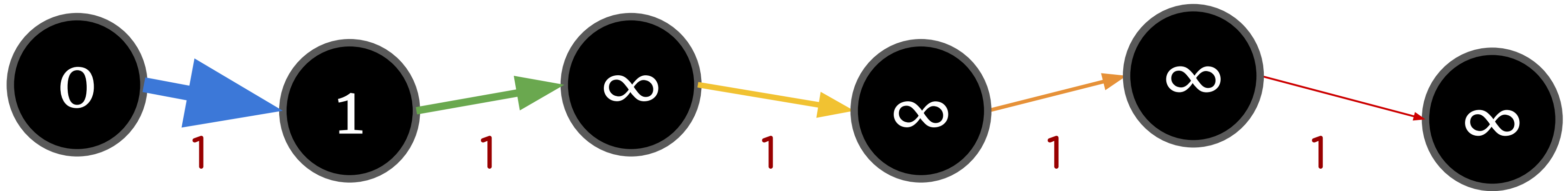


Bellman-Ford doesn't know it won't ever improve an estimate again. It goes through all the rest of Step 3, then Step 4.

We might need all n-1 rounds…



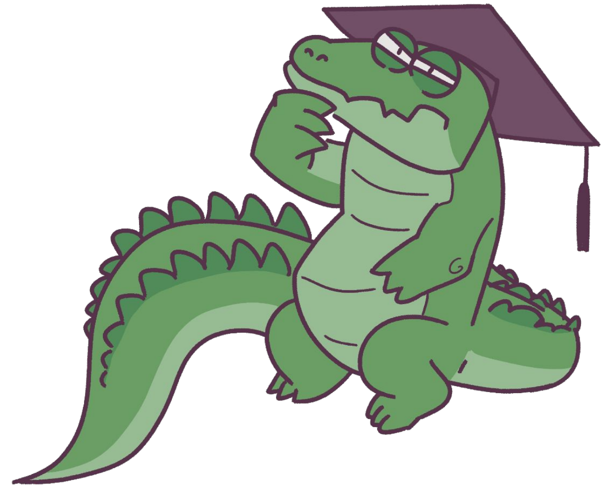
We might need all n-1 rounds...



Here, in each round, we only make one step of progress, because we chose a bad order in which to look at the edges...

1. Could we safely stop if we see a whole round with no changes?

2. After n-1 steps, are we guaranteed to see a round with no changes?



1. Could we safely stop if we see a whole round with no changes? Yes!

2. After n-1 steps, are we guaranteed to see a round with no changes?

1. Could we safely stop if we see a whole round with no changes? Yes!

Then the next round is necessarily exactly the same, i.e., also doesn't change.

2. After n-1 steps, are we guaranteed to see a round with no changes?

1. Could we safely stop if we see a whole round with no changes? Yes!

Then the next round is necessarily exactly the same, i.e., also doesn't change.

2. After n-1 steps, are we guaranteed to see a round with no changes? Maybe...

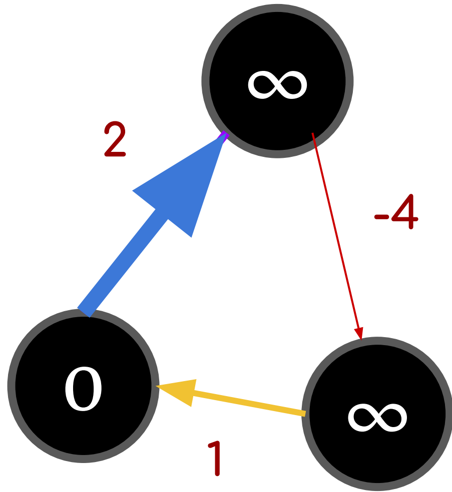
1. Could we safely stop if we see a whole round with no changes? Yes!

Then the next round is necessarily exactly the same, i.e., also doesn't change.

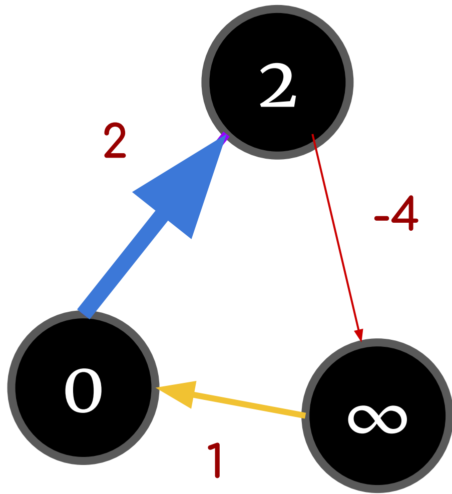
2. After n-1 steps, are we guaranteed to see a round with no changes? Maybe...

The longest path is n−1 steps, so there is no more propagation... unless...

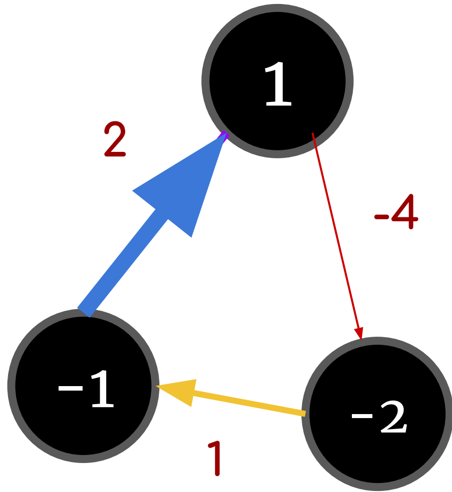
Start



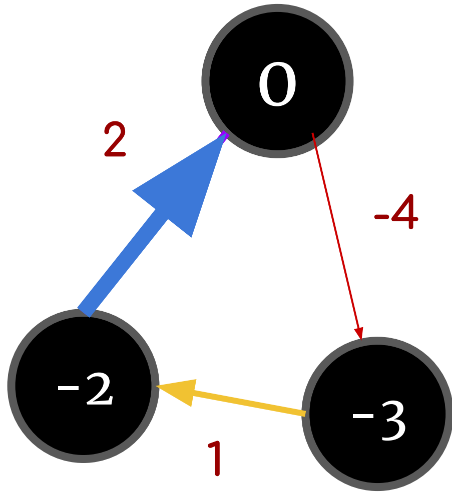
After Round 1



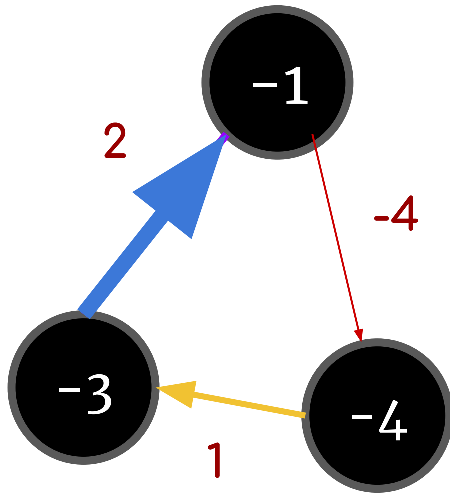
After Round 2



After Round 3



After Round 4...



How do we deal with this? How can we detect a negative cycle without running arbitrarily many rounds?

Our Strategy

- Do $n-1$ rounds of processing all edges.
- Then do one extra round. If anything changes, we have a negative cycle. Otherwise, we're fine.

Finding actual paths / negative cycles

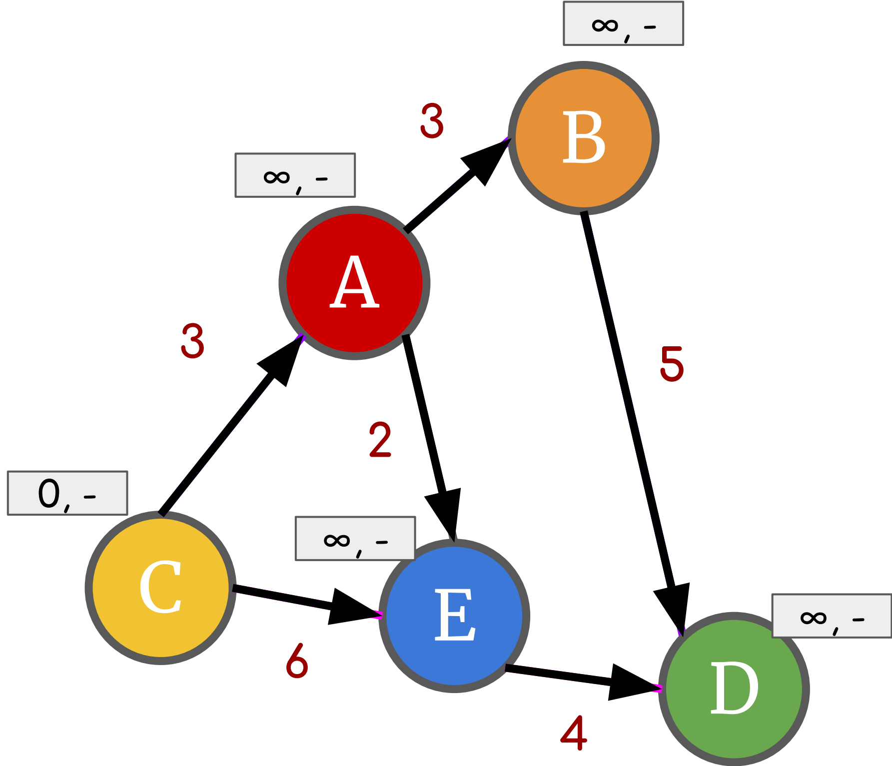
- Have each node store an extra piece of information: its *predecessor*.
  - The starting node, of course, has no predecessor.

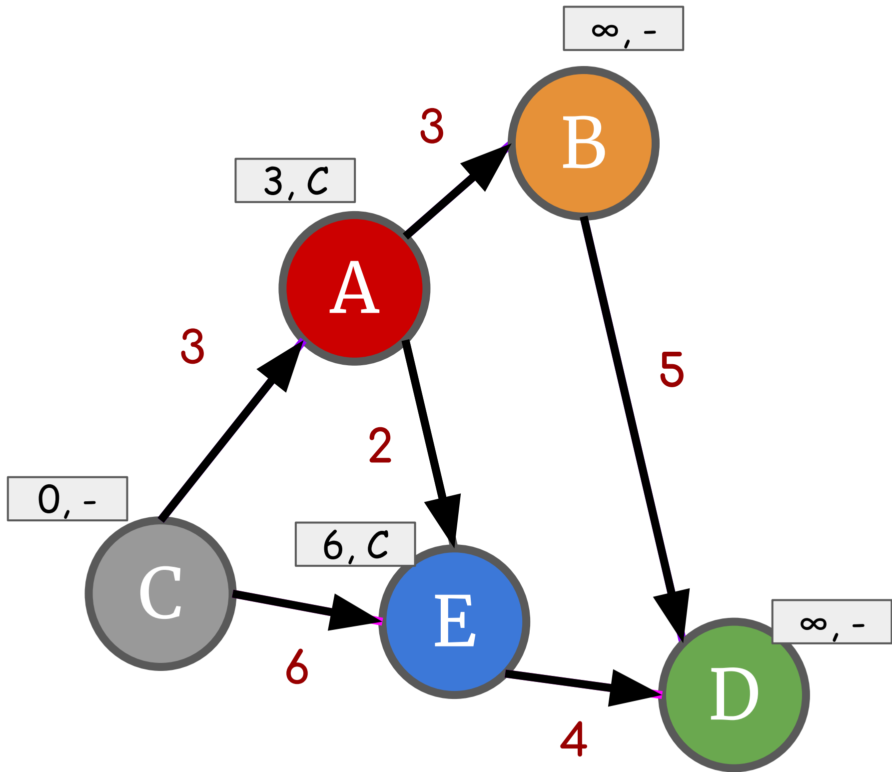
- When updating a node's estimate, we also update its predecessor (the node we just used).

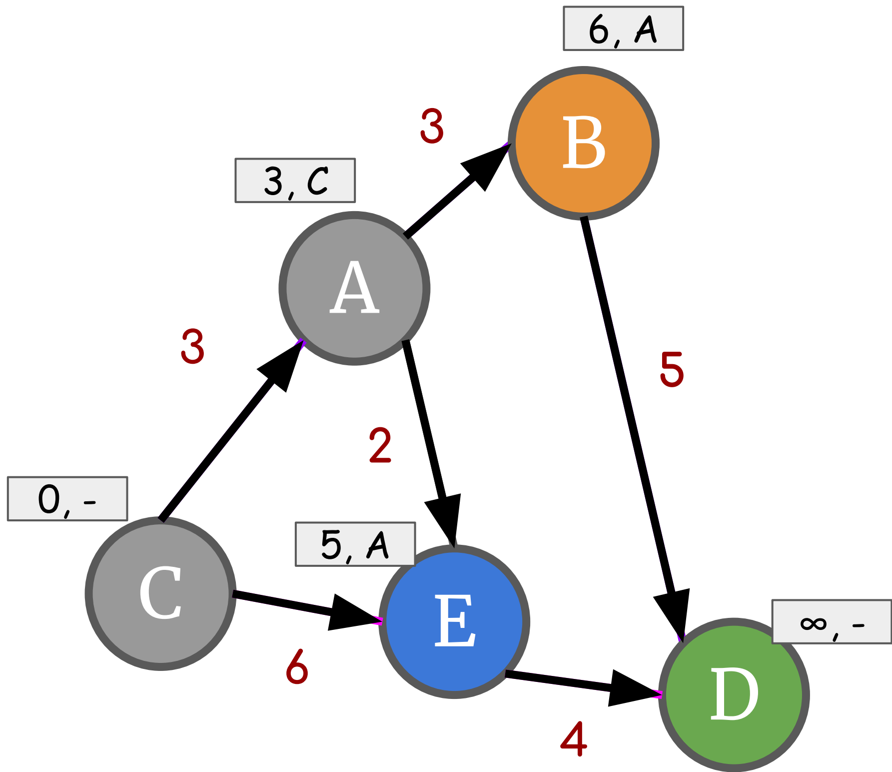
- To reconstruct a path, start at the end and follow the predecessor chain backwards.

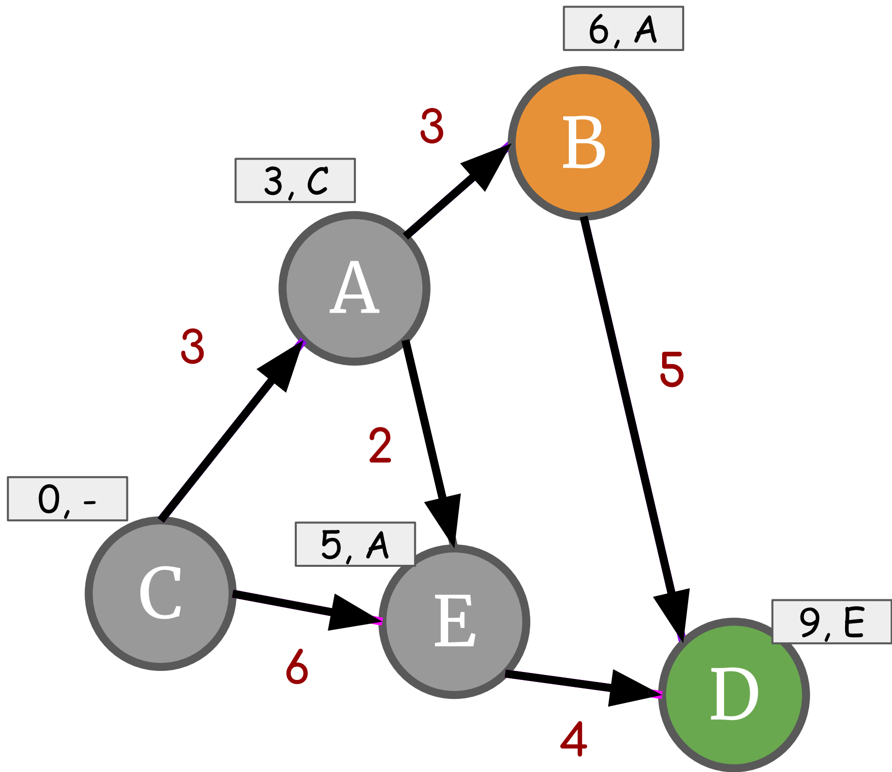
- Can do this with Dijkstra's as well!

(We'll just do Dijkstra's here, rather than Bellman-Ford, to illustrate the idea.)

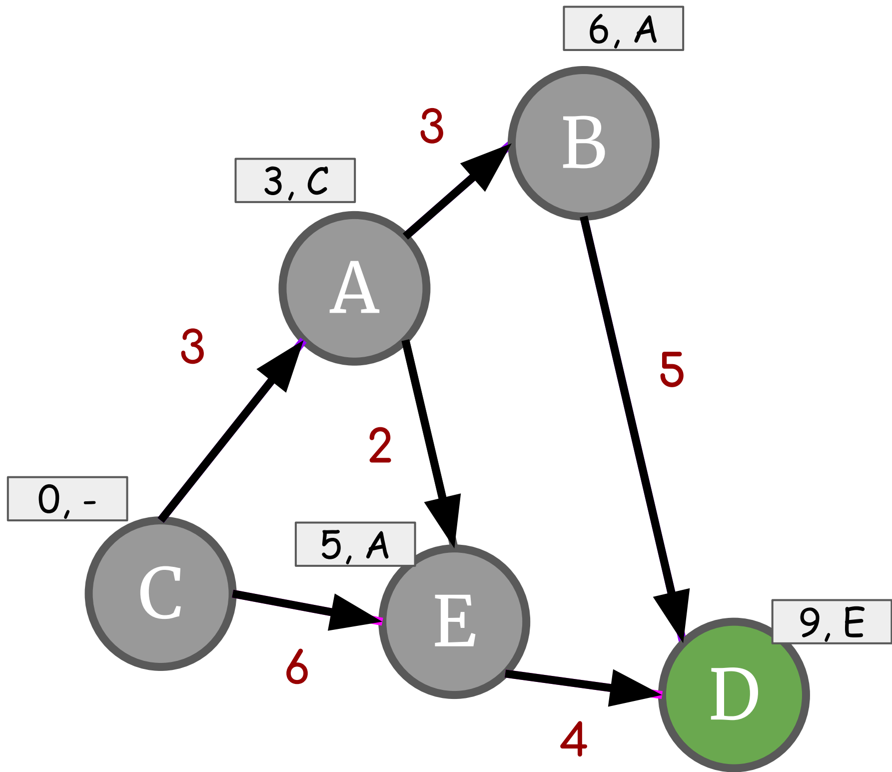


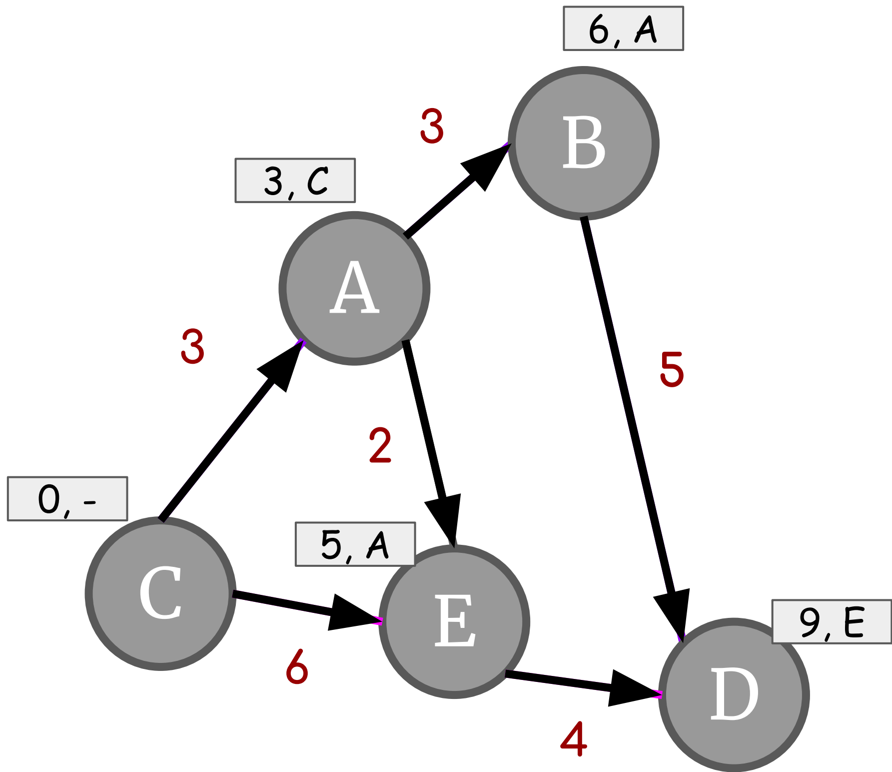

∞, -
B
3
3, C
A
3
5
2
0, -
6, C
C
E
∞, -
D
6
4

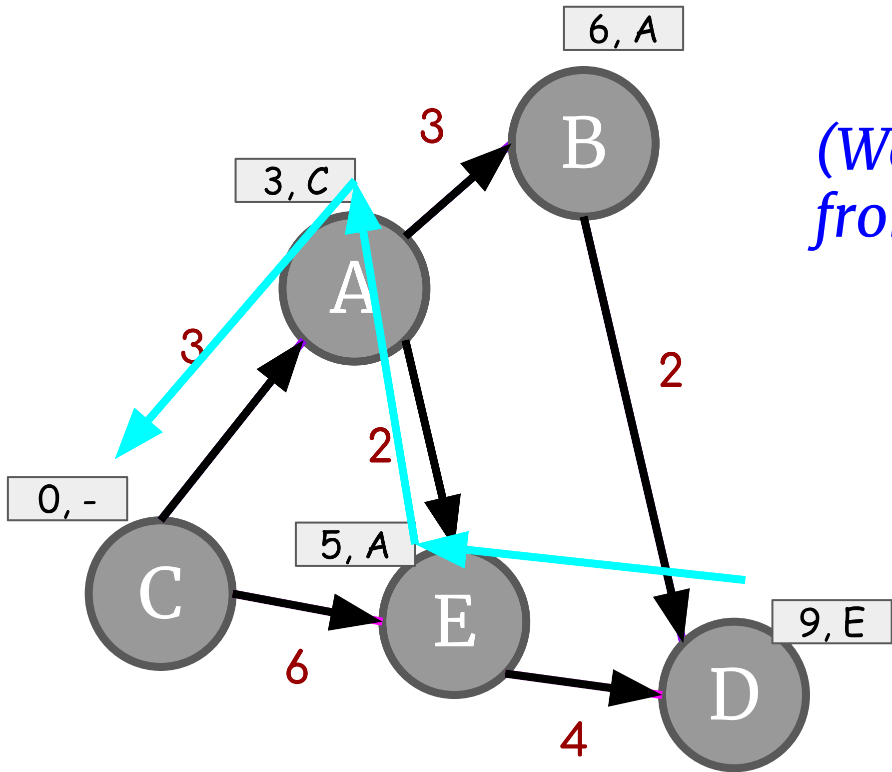

6, A
3
B
3, C
A
3
5
2
0, -
5, A
C
E
∞, -
6
D
4



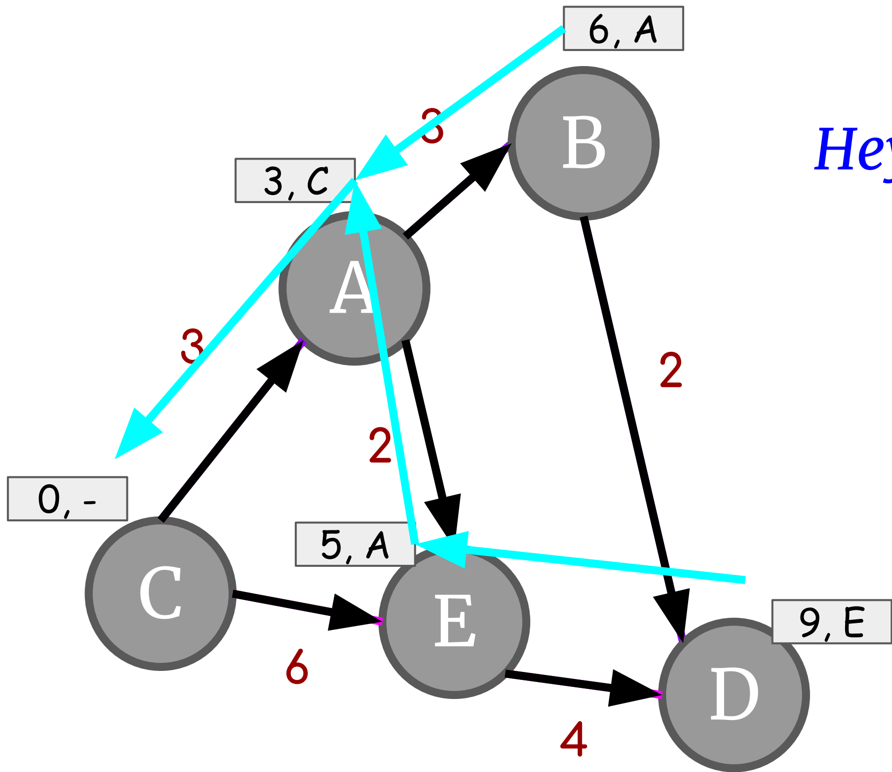
6, A
B
3, C
3
A
3
5
2
0, -
5, A
C
E
9, E
6
D
4


6, A
B
3
3, C
A
3
5
2
0, -
5, A
C
E
9, E
6
D
4


6, A
3
B
3, C
A
3
5
2
0, -
5, A
C
E
9, E
6
D
4



(We work backwards to find the path from C to D.)



Hey, we built a tree!

How fast is it?

- We do n rounds of looking at m edges each.
- Each check of an edge (and update of the destination's estimate and predecessor) takes O(1) time.
- Therefore: $O(nm)$ overall.
  - For a dense graph with $m = O(n^2)$, could be $O(n^3)$.
- Implementation: Easy! We no longer need a super-complicated heap, because we no longer try to pick the node with the smallest estimate.

Does it work?

- We may deal with some part of the proof on HW5.
- The idea: show that after k rounds, each vertex v's estimate is no greater than the cost of any source-to-v path that uses at most k edges.
    - This gets around the issue the Dijkstra's proof would have with negative edges.
- Also need to show that negative cycles get detected, but this is easier – just show that only they keep changing estimates even on an n-th round.

Bellman-Ford in practice

- Dijkstra's needs to know every vertex's estimate at all times, but notice that Bellman-Ford operates more locally (there is no heap of estimates).

- This can make it good for situations with changing values (though the changes may take a while to propagate).

Rip

Routing Information Protocol.

This module implements distance vector routing as specified in RFC 2453 (RIPv2) and RFC 2080 (RIPng). The routing protocol uses the Bellman-Ford algorithm to compute the optimal routes. Each router periodically sends its routes to the neighboring routers. This update message contains the destination (address, netmask) and metric of each route. When a router receives an update message, it adds the cost of the incoming interface to the cost of the received route and if it is smaller than the metric of its current route then it updates the current route.

7/25 Lecture Agenda

- Announcements
- Part 5-1: Bellman-Ford
- 10 minute break!
- Part 5-2: Intro to Dynamic Programming

7/25 Lecture Agenda

- Announcements
- Part 5-1: Bellman-Ford
- 10 minute break!
- Part 5-2: Intro to Dynamic Programming

WORLD 5-2

What Even Is Dynamic Programming?

Divide and Conquer
Sorting & Randomization
Data Structures
Graph Search
Dynamic Programming
Greed & Flow

Special Topics

What did we just do?

- Bellman-Ford let us avoid explicitly checking all possible paths through the graph.

- We found a way to divide the process into rounds.

- We maintained estimates of the form "this is the best we can do so far on this round".

- This is the essence of **dynamic programming**: find some substructure and exploit it to avoid doing redundant work.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

"Dynamic programming" is a basically meaningless name

but the story is funny

Indy loves DP questions

In interviews, DP is kind of a screen for "have you taken an algorithms class?"

It appears to be a sufficiently difficult idea to create sufficient "spread" among candidates

It's also fairly easy to *write* DP problems.

Is DP actually useful beyond interviews?

- **Yes**, in that it is used in a number of fundamental algorithms that make computing and the Internet work. (compilers! networking! string searching! etc. etc.)

- **Maybe?**, in that you can probably have a stellar career in industry or research (or whatever!) without ever implementing your own DP.
    - But it's still worth knowing it when you see it, and understanding the overall idea.

You already know some DP!

- **Bellman-Ford**, as mentioned.

- **Dijkstra's** is arguably also DP and I will fight any algorithms person who insists it's not.
  - We will never have an exam question like "is this DP?"

- You may have seen the idea of memoization (that is not a typo for "memorization"!)
  - *memoized!*
  - Fib(10) = Fib(9) + Fib(8) = Fib(8) + Fib(7) + Fib(8)...
  - Instead of computing Fib(8) over and over, store the result in a table the first time you compute it. Then just look up that result every time.

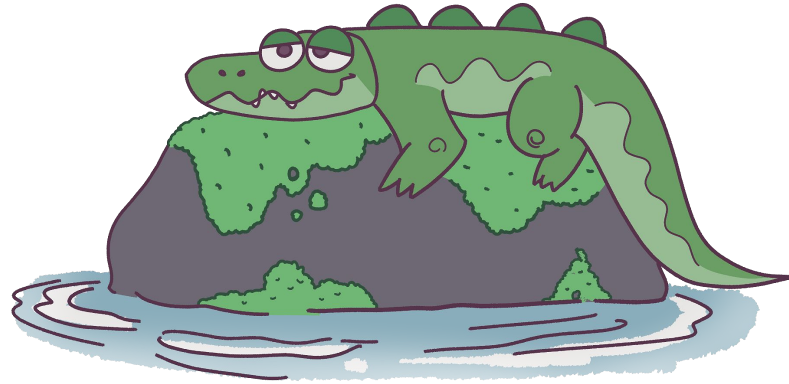
Top-down vs. bottom-up

- **Top-down:** Start from the original problem, e.g., Fib(5). Remember the solutions to subproblems (memoization).

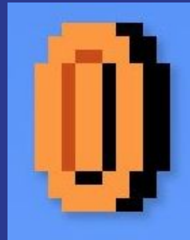
- **Bottom-up:** Start from the base case(s), e.g., Fib(0) and Fib(1). Use these to compute Fib(2), then use this info to compute Fib(3), etc.

- Bottom-up is often a better choice since it avoids the overhead of lookups and storing a bunch of intermediate data. But it's also often harder to write.

This is all very abstract! Let's understand DP by example.

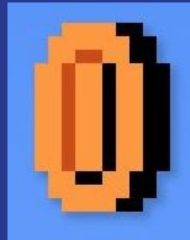


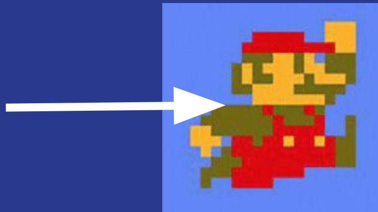
Mario's extremely basic adventure
(probably like 50 bucks on Switch)



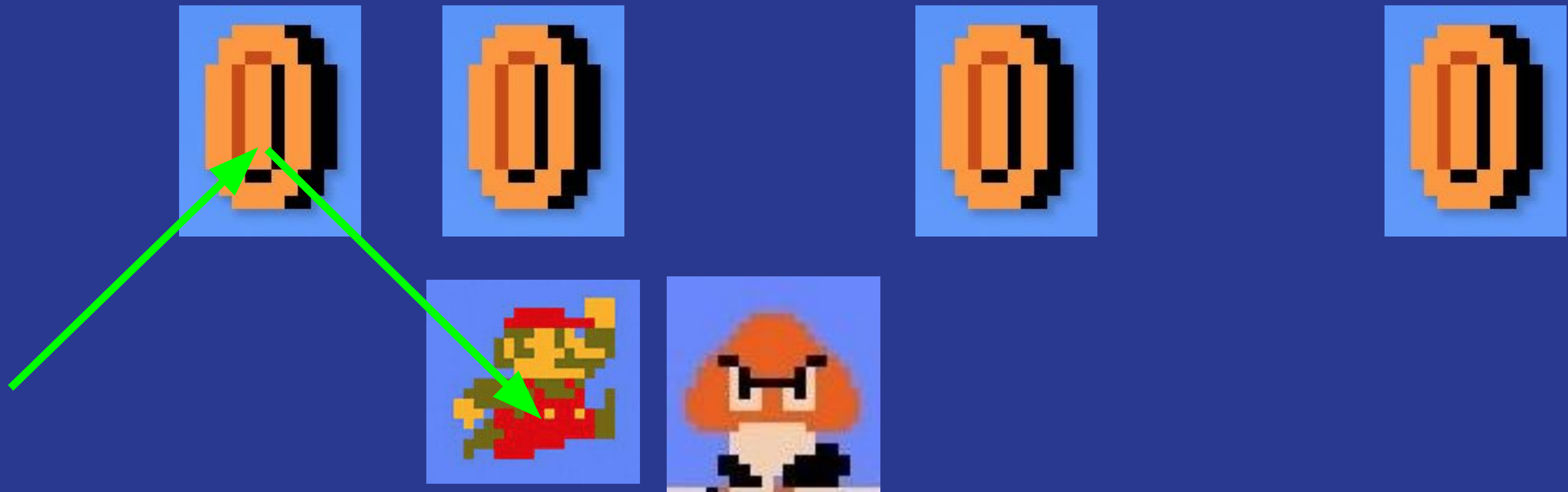
In this game, Mario has two kinds of move

Option 1: Go forward one step

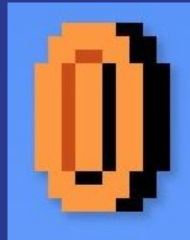




Option 2: Jump

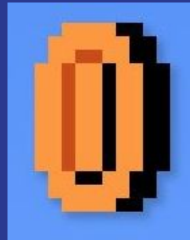


coins are good
you want as many as possible
because Mario's life is empty



enemies do not move

(they've been doing this for 35+ years, the excitement isn't there anymore)



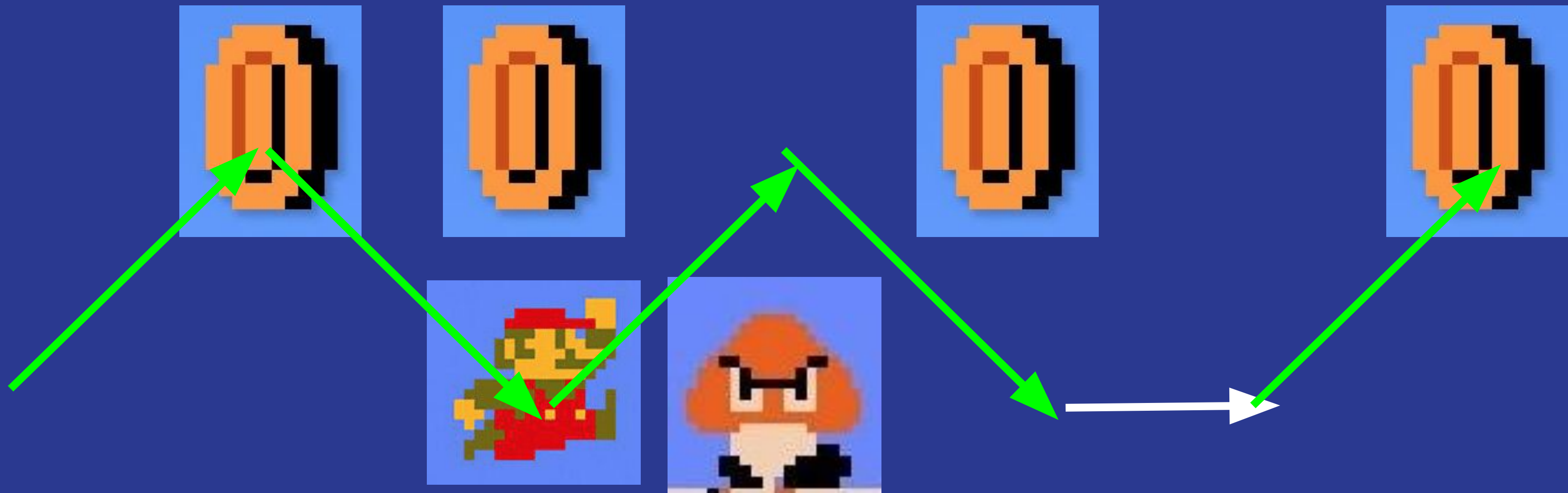
not OK to walk into enemies

how did you get hit, it was just standing there

OK to land on enemies
because Mario is an asshole

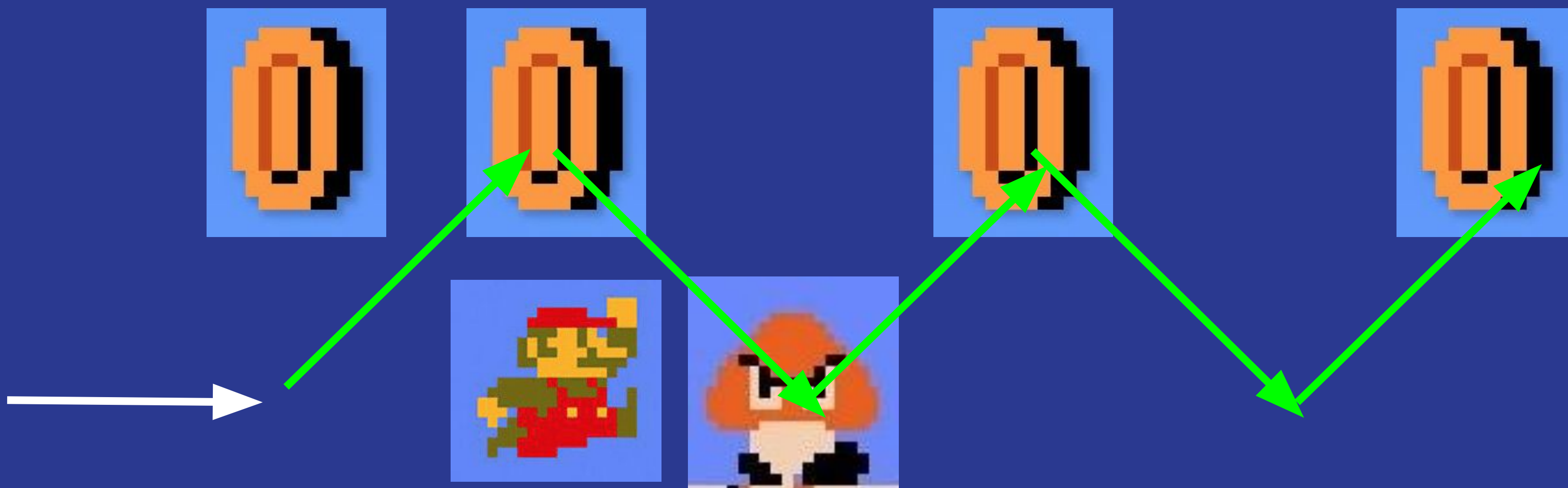
Greedy strategies aren't always optimal

2 coins



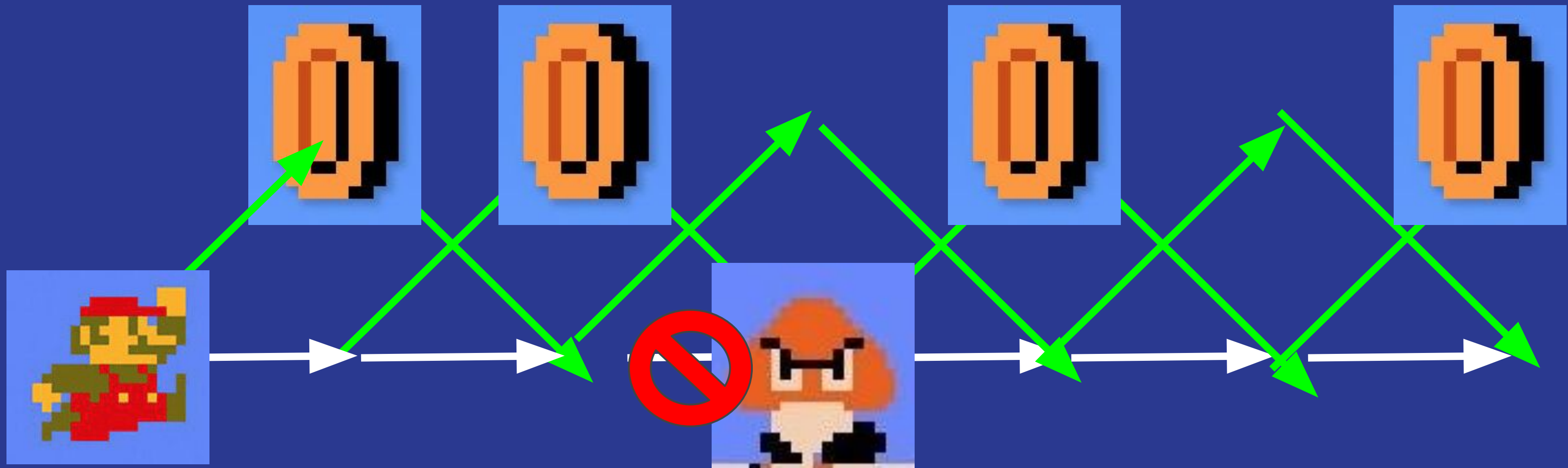
What we should have done

3 coins



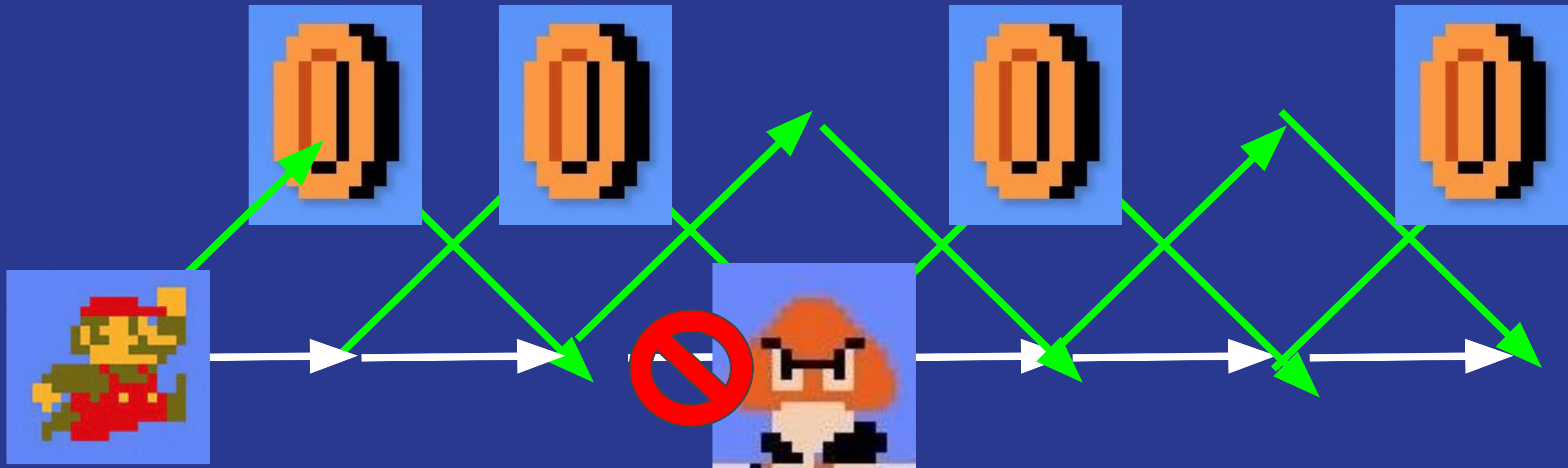
Why not just try every path?

Exponential number…



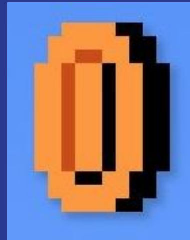
Why not just try every path?

Exponential number… <u>so any solution that explicitly considers them all is exponential</u>



Solving via DP

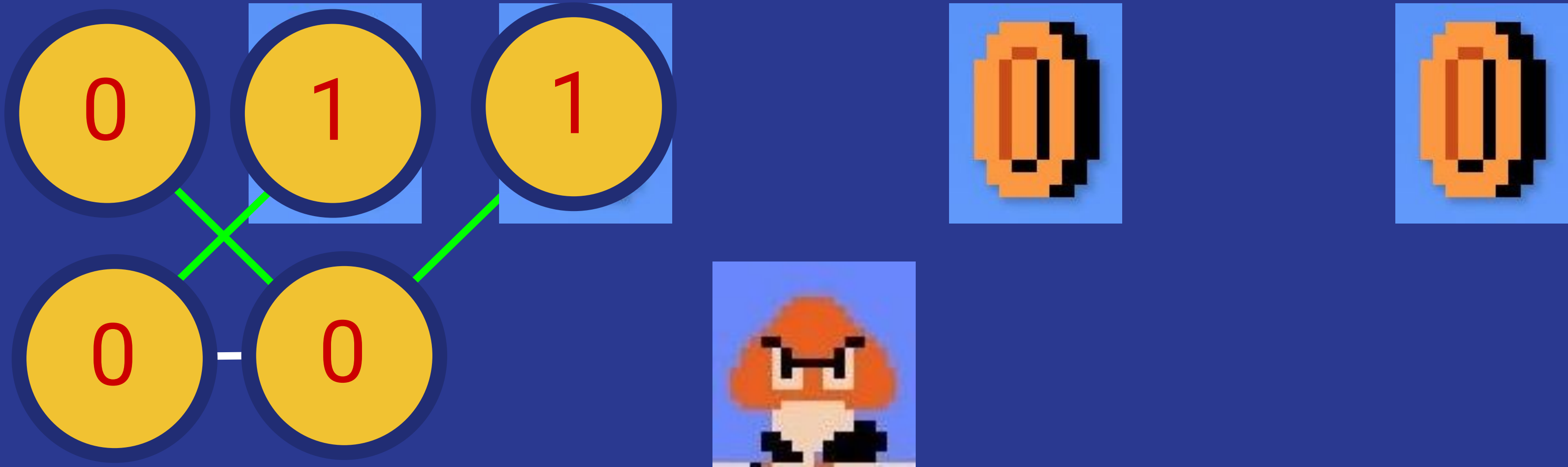
what? we can't get here...
but you'll see why we
need it



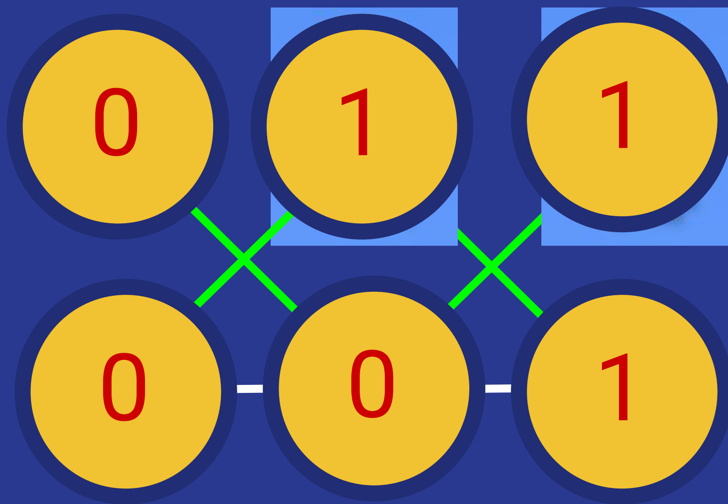
top row cell:
value from downleft,
plus 1 if coin

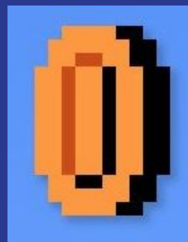
bottom row cell:
max of:
1. value from upleft
2. value from left if no enemy here

top row cell:
value from downleft,
plus 1 if coin



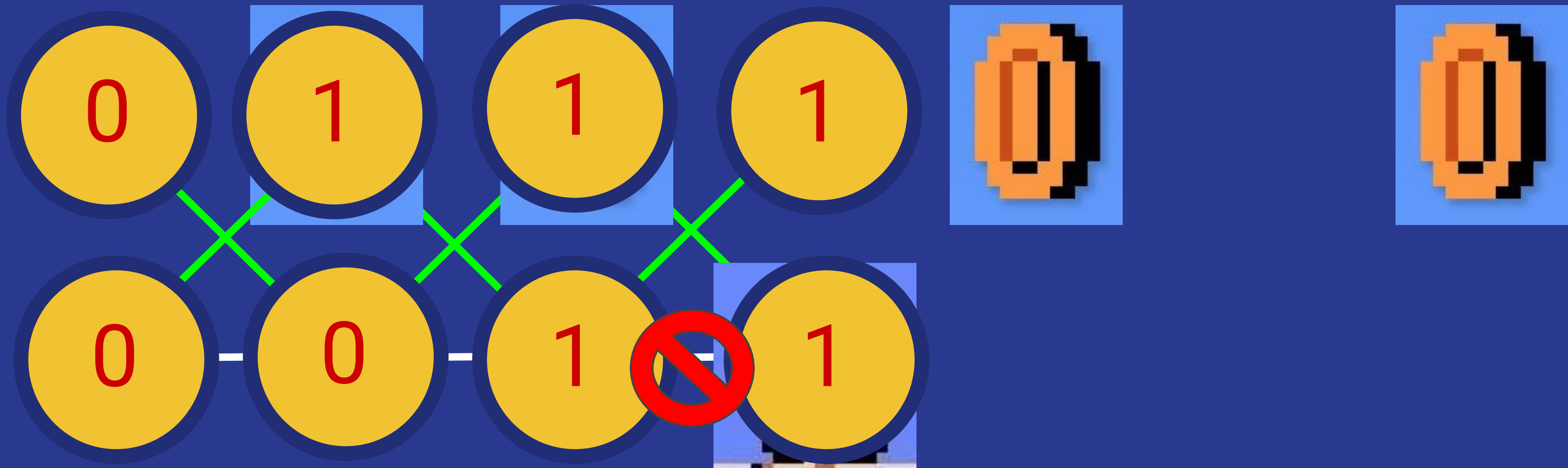
bottom row cell:
max of:
1. value from upleft
2. value from left if no enemy here



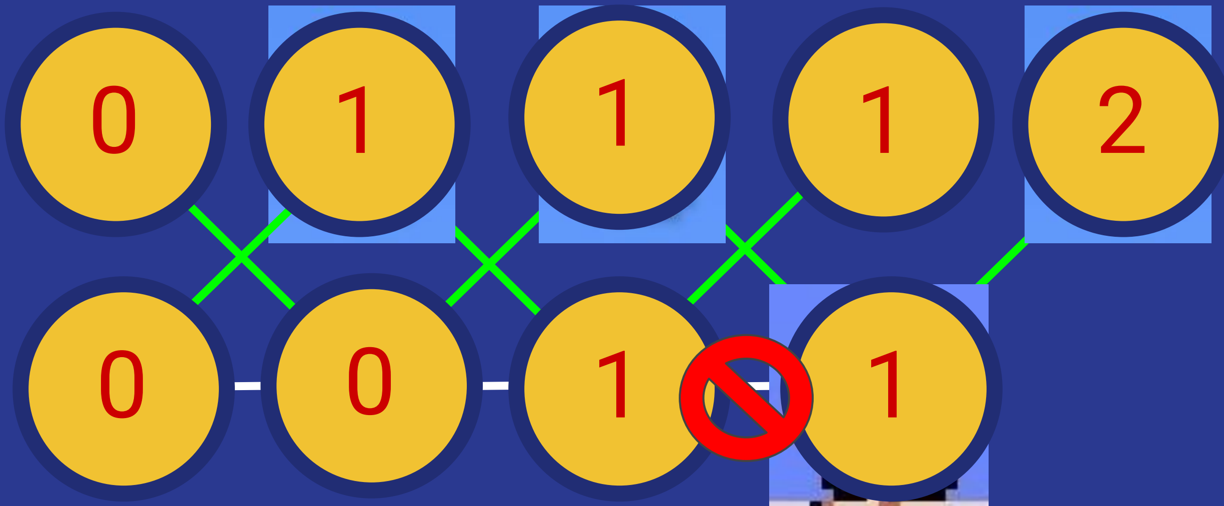


top row cell:
value from downleft,
plus 1 if coin

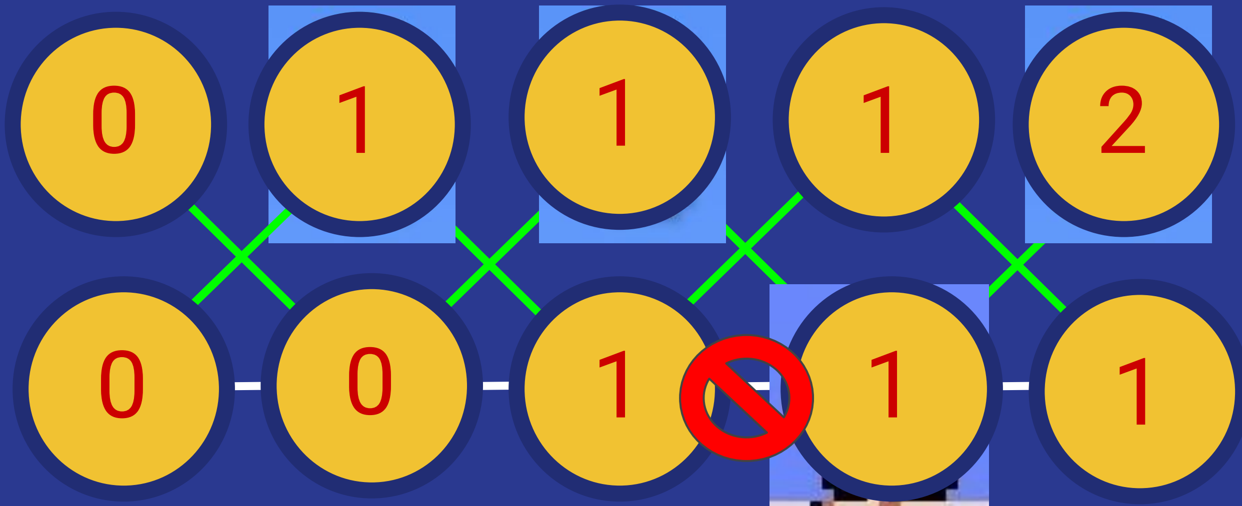
bottom row cell:
max of:
1. value from upleft
2. value from left if no enemy here



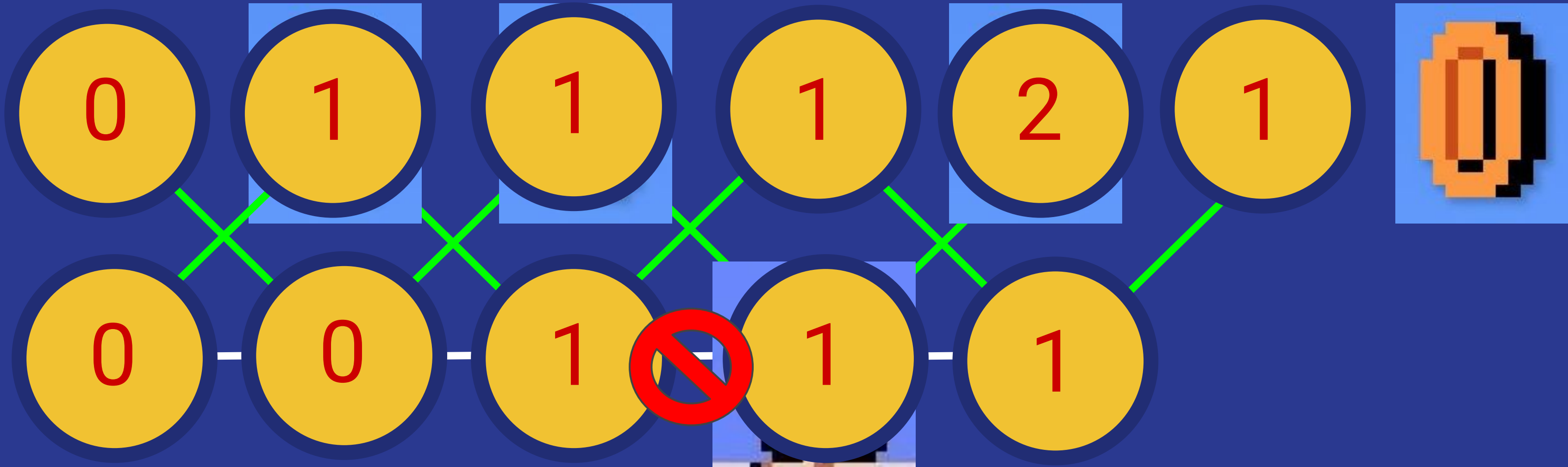
top row cell:
value from downleft,
plus 1 if coin



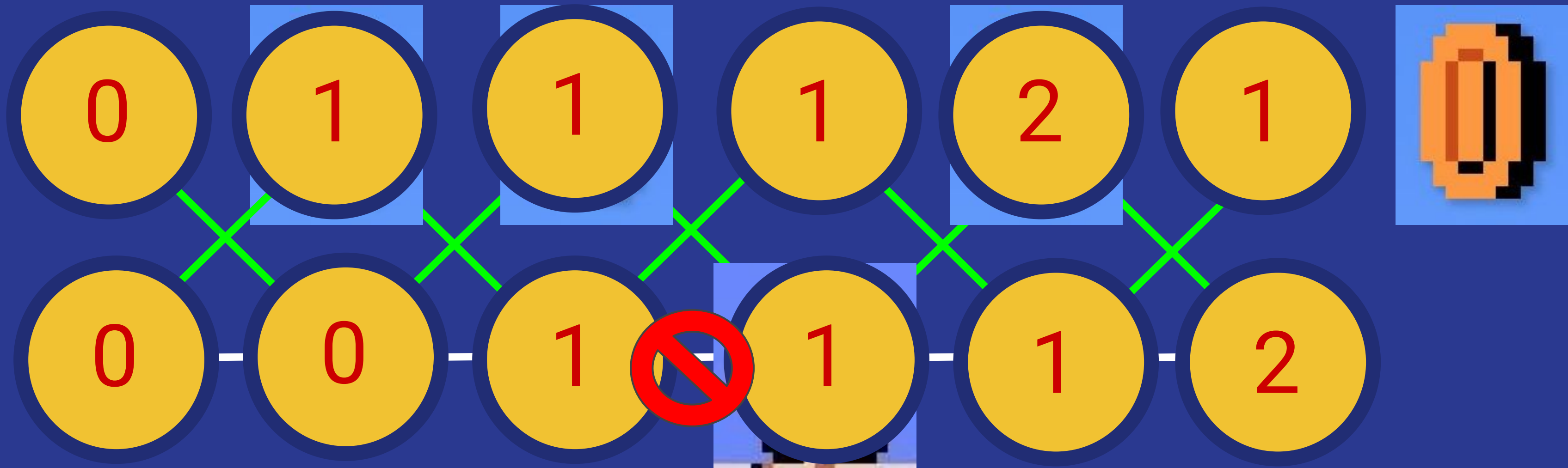
bottom row cell:
max of:
1. value from upleft
2. value from left if no enemy here



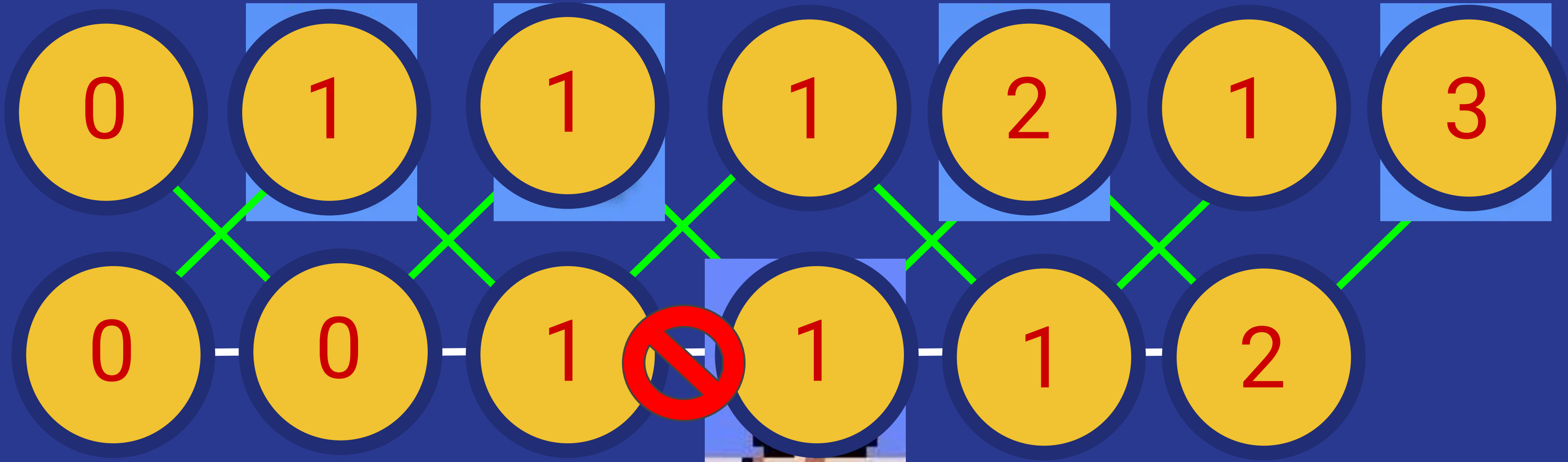
top row cell:
value from downleft,
plus 1 if coin



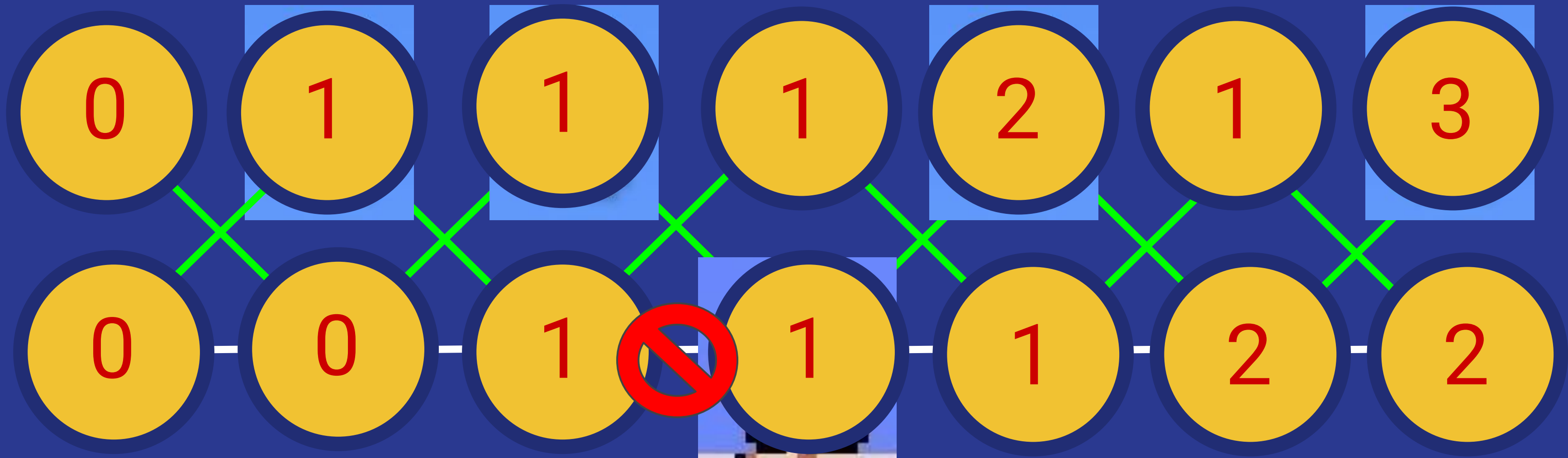
bottom row cell:
max of:
1. value from upleft
2. value from left if no enemy here



top row cell:
value from downleft,
plus 1 if coin

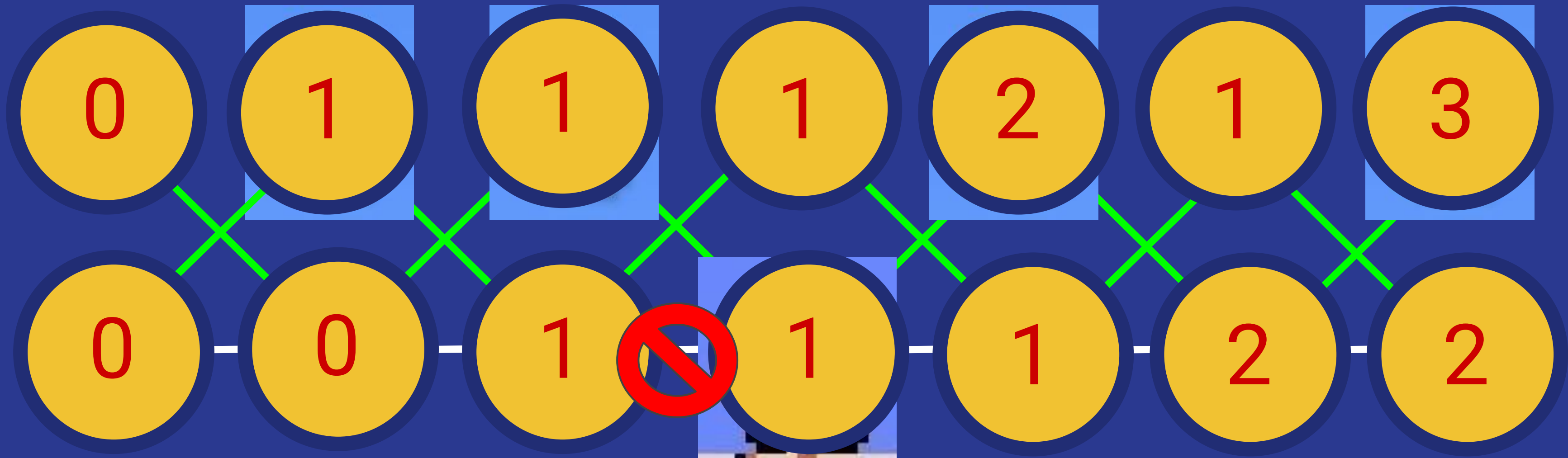


bottom row cell:
max of:
1. value from upleft
2. value from left if no enemy here



Wait a minute...

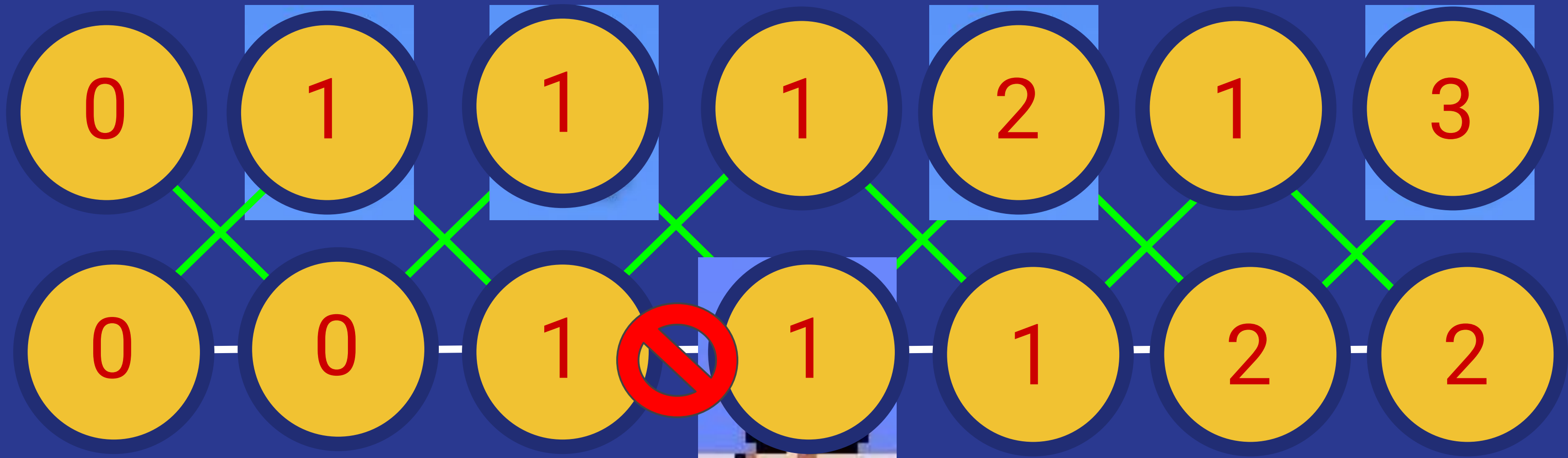
Isn't this just the "exponential" slide again?



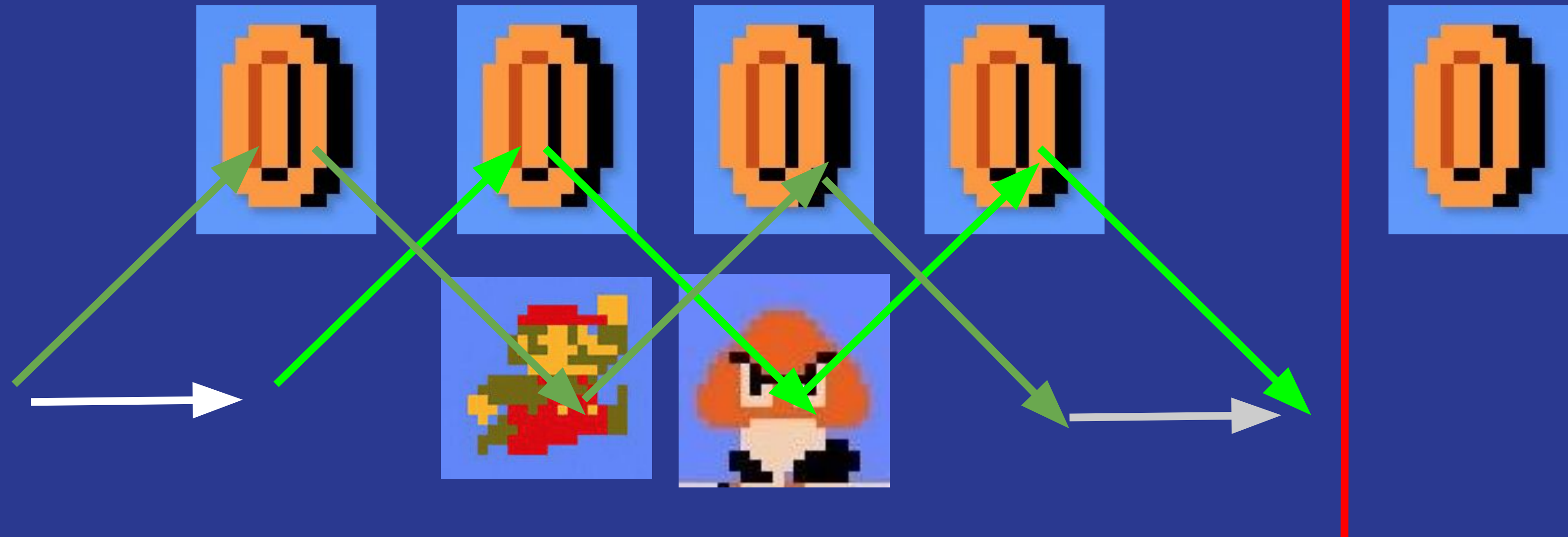
Wait a minute...

Isn't this just the "exponential" slide again?

No! We took <u>linear time</u>.



once we get this far, the strategy from then on doesn't depend on how we got there



Code!

```
def solve(length, coins, enemies):
    dp = [[0 for _ in range(2)] for __ in range(length)]
    for i in range(1, length):
        # in second index, 1 = top row, 0 = bottom row
        dp[i][1] = dp[i-1][0] + (1 if coins[i] else 0)
        dp[i][0] = max(-1 if enemies[i] else dp[i-1][0], dp[i-1][1])
    print(dp)
    print(max(dp[length-1][0], dp[length-1][1]))

solve(7, [False, True, True, False, True, False, True],
      [False, False, False, True, False, False, False])

# (base) Ians-MacBook-Air:Desktop iantullis$ python mario.py
### [[0, 0], [0, 1], [1, 1], [1, 1], [1, 2], [2, 1], [2, 3]]
# 3
```

More space-efficient code!

```
def solve(length, coins, enemies):
    prv = [0, 0]
    nxt = [0, 0]
    for i in range(1, length):
        # in second index, 1 = top row, 0 = bottom row
        nxt[1] = prv[0] + (1 if coins[i] else 0)
        nxt[0] = max(-1 if enemies[i] else prv[0], prv[1])
        prv = nxt
    print(max(nxt[0], nxt[1]))

solve(7, [False, True, True, False, True, False, True],
      [False, False, False, True, False, False, False])

# (base) Ians-MacBook-Air:Desktop iantullis$ python mario.py
# 3
```

Even more space-efficient code (thx Manas!)

```
def solve(length, coins, enemies):
    curr = 0         # the column we are in
    nxt = None       # the column to the right of that
    nxtnxt = None    # the column to the right of THAT
    for i in range(length-1):
        if curr is not None:
            # walk
            if not enemies[i+1]:
                nxt = curr if nxt is None else max(nxt, curr)
            # or jump
            newscore = curr + (1 if coins[i+1] else 0)
            nxtnxt = newscore if nxtnxt is None else max(nxtnxt, newscore)
        # shift to next column
        curr = nxt
        nxt = nxtnxt
        nxtnxt = 0
    print(max(curr, nxt))

solve(7, [False, True, True, False, True, False, True],
      [False, False, False, True, False, False, False]) # answer 3

solve(4, [True, True, True, True],
      [False, False, False, False]) # answer 2

solve(3, [False, True, False],
      [False, False, False]) # answer 1

solve(3, [False, False, True],
      [False, True, False]) # answer 0
```

This eliminates the need for a 2D array – and now only uses 3 values – but is a little harder to understand.

Key points

- That was a **bottom-up** approach. We started at the beginning of the "stage" rather than working backwards from the end.
    - Top-down would have been: best at position n = max(best at position n-1 if we didn't jump, best at position n-1 if we did jump), etc.

- We were able to divide the problem into "rounds" in a natural way (each round = one step forward to the right) such that each round depended only on the previous round's results.

- We only cared about: "what's the best we can do *up to this point*?"

What if there's a more clever approach?

- In this problem, there quite possibly is. But hey, this DP was already linear-time. We're not going to do asymptotically better.

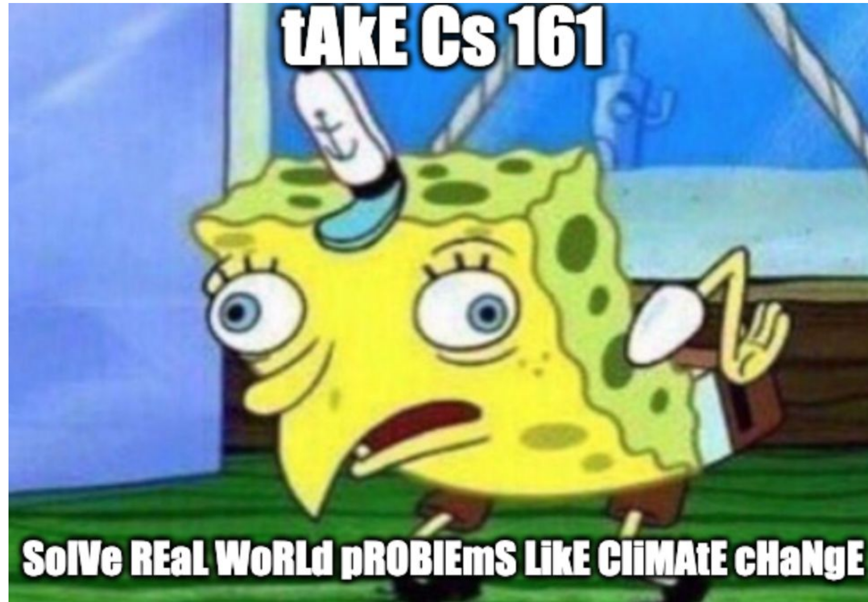
- In more complicated problems, it is way harder to spot a simple rule / "greedy" approach.
  - and much easier to spot a tantalizing but incorrect greedy approach...

Another problem (if time): frog friends!

- Fizz and Buzz are two frog friends who live on a very long linear chain of lilypads.

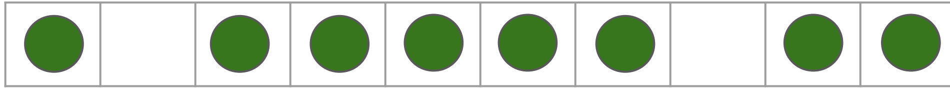
Another problem (if time): frog friends!

- Fizz and Buzz are two frog friends who live on a very long linear chain of lilypads.

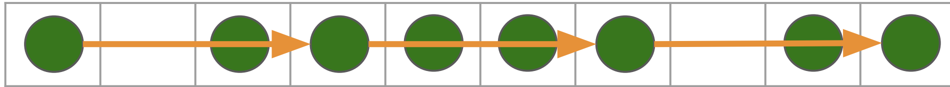


Another problem (if time): frog friends!

- Fizz and Buzz are two frog friends who live on a very long linear chain of lilypads.
- The Internet ruined the frog emoji a while ago, so just imagine cute, wholesome frogs.
- Fizz can put Buzz on her back and jump **exactly 3** lilypads to the right, but this requires 2 flies' worth of energy.
- Buzz can put Fizz on his back and jump **exactly 2** lilypads to the right, but this requires 1 fly worth of energy.
- Goal: reach a particular lilypad using as little energy as possible.

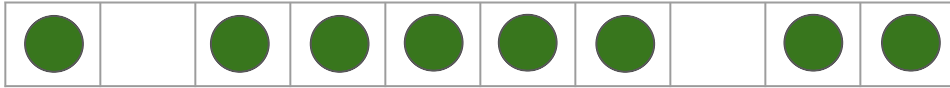


The obvious greedy strategy: Jump as far as possible! Go, Fizz, go!

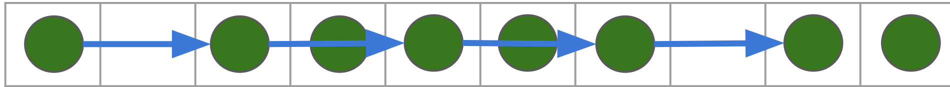


total cost 6

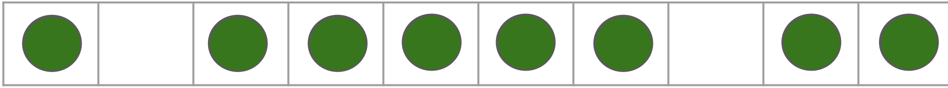
But this actually turns out not to be optimal!



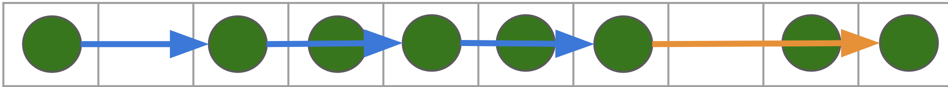
OK, let's use as little energy as possible! Go, Buzz, go!



can't reach end. RIP



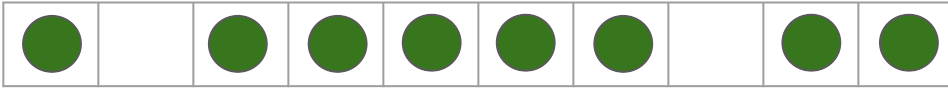
An optimal solution uses both.



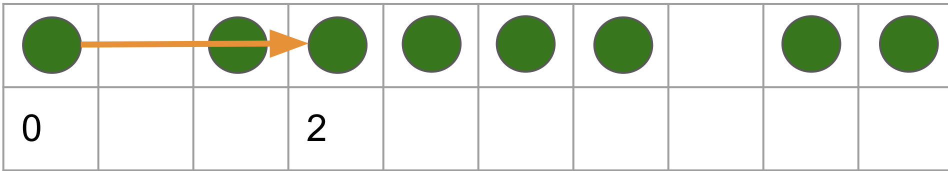
total cost 5

DP: ask – what is the *least* we can have spent to get this far?

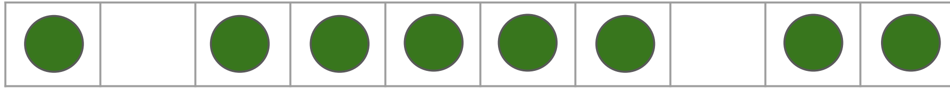
0									



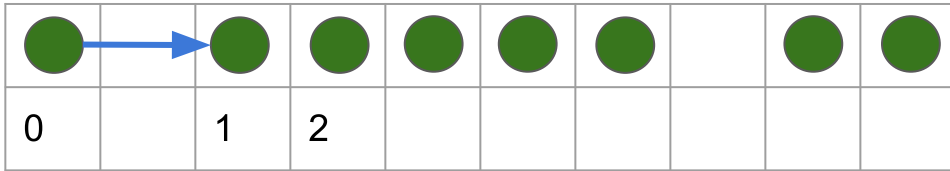
DP: ask – what is the *least* we can have spent to get this far?



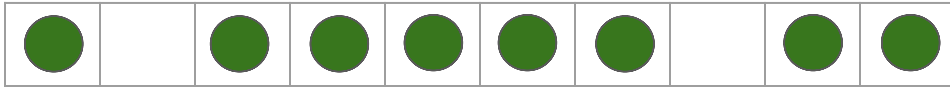
We can either use Fizz, who costs 2...



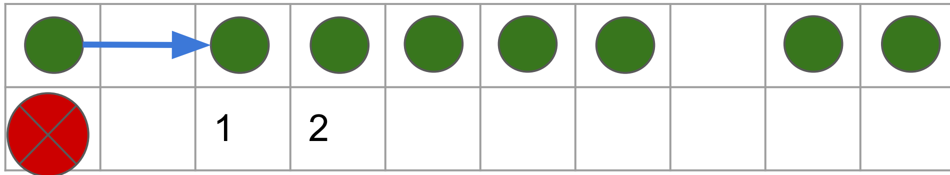
DP: ask – what is the *least* we can have spent to get this far?



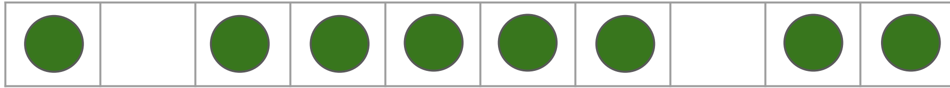
We can either use Fizz, who costs 2...
...or Buzz, who costs 1.



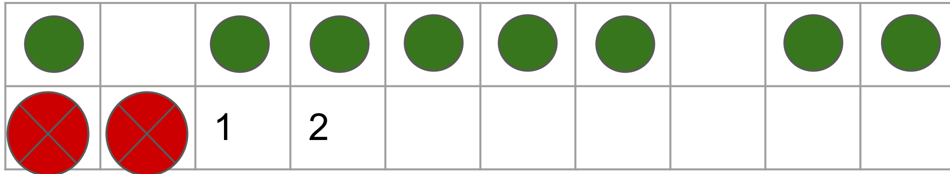
DP: ask – what is the *least* we can have spent to get this far?



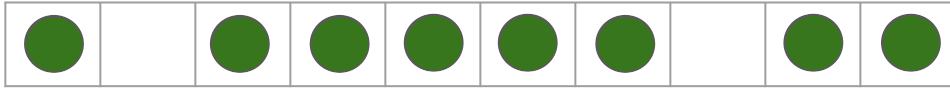
We can either use Fizz, who costs 2...
...or Buzz, who costs 1.
That's all our options from the first position, so we are done with that (because there is no moving backward...)



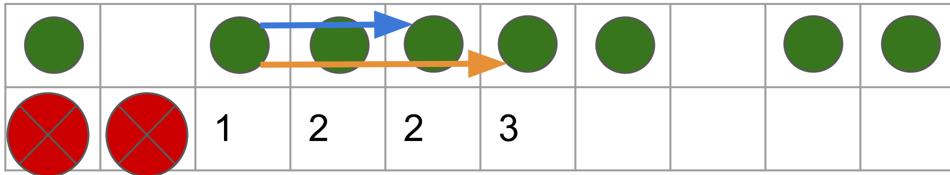
DP: ask – what is the *least* we can have spent to get this far?



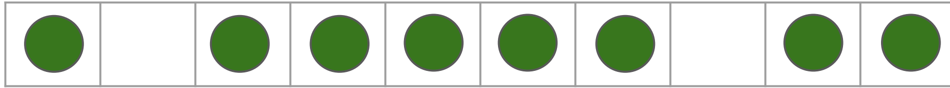
There is no lilypad at 1, so we move on.



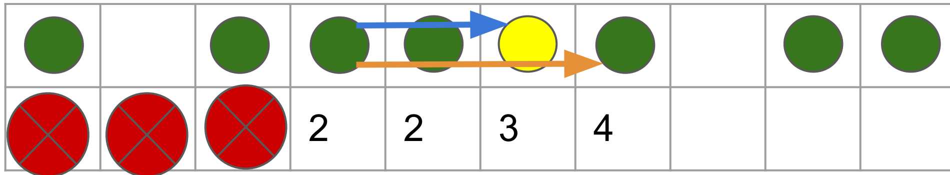
DP: ask – what is the *least* we can have spent to get this far?



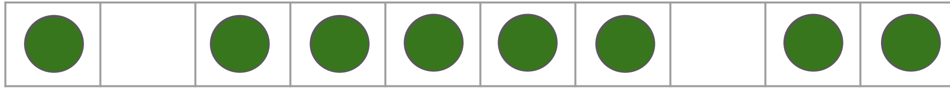
We can either use Fizz, who costs 2...
...or Buzz, who costs 1.



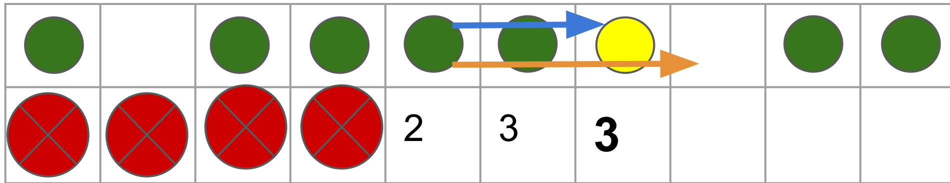
DP: ask – what is the *least* we can have spent to get this far?



Here, if we use Buzz, we can get to the highlighted lilypad with cost 3.
But we already had a way to do that in 3.

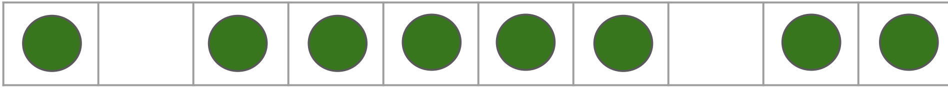


DP: ask – what is the *least* we can have spent to get this far?

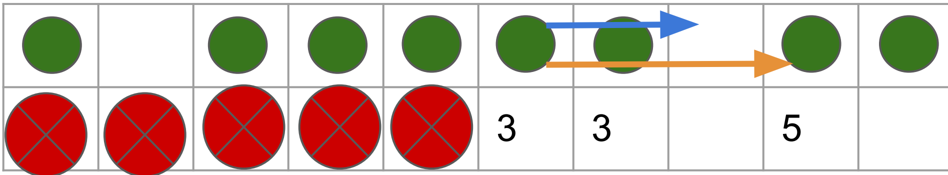


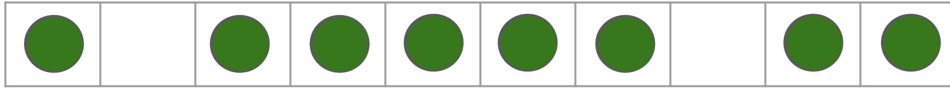
Here, if we use Buzz, we can get to the highlighted lilypad with cost 3.
This is better than our previous estimate of 4, so we replace it.

(Also, we can't use Fizz from here.)

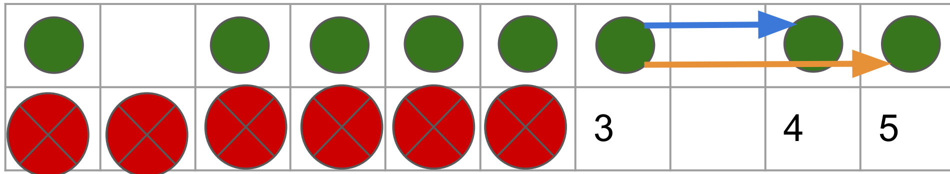


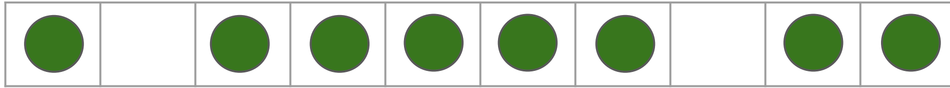
DP: ask – what is the *least* we can have spent to get this far?



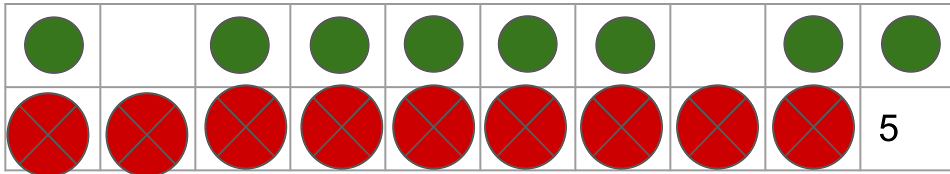


DP: ask – what is the *least* we can have spent to get this far?



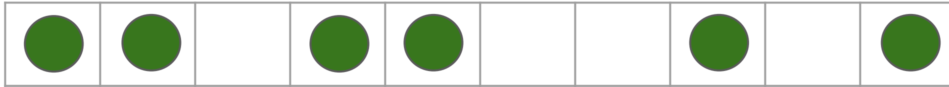


DP: ask – what is the *least* we can have spent to get this far?



So we can do better than 6.

Extension: what if the frogs can jump either direction?



Is DP the right call here? How would you solve it with DP?

What, if anything that we've learned, might be a better fit?