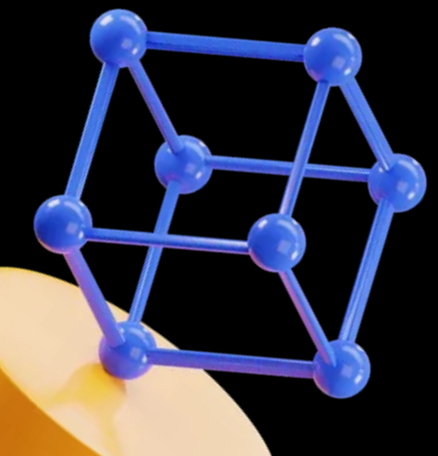
Sohum Sheth, Dean Richardson, Ahmed Shaheen

HKD MEDIUM SECTION #2

Greedy Algorithms

February 11th, 2026



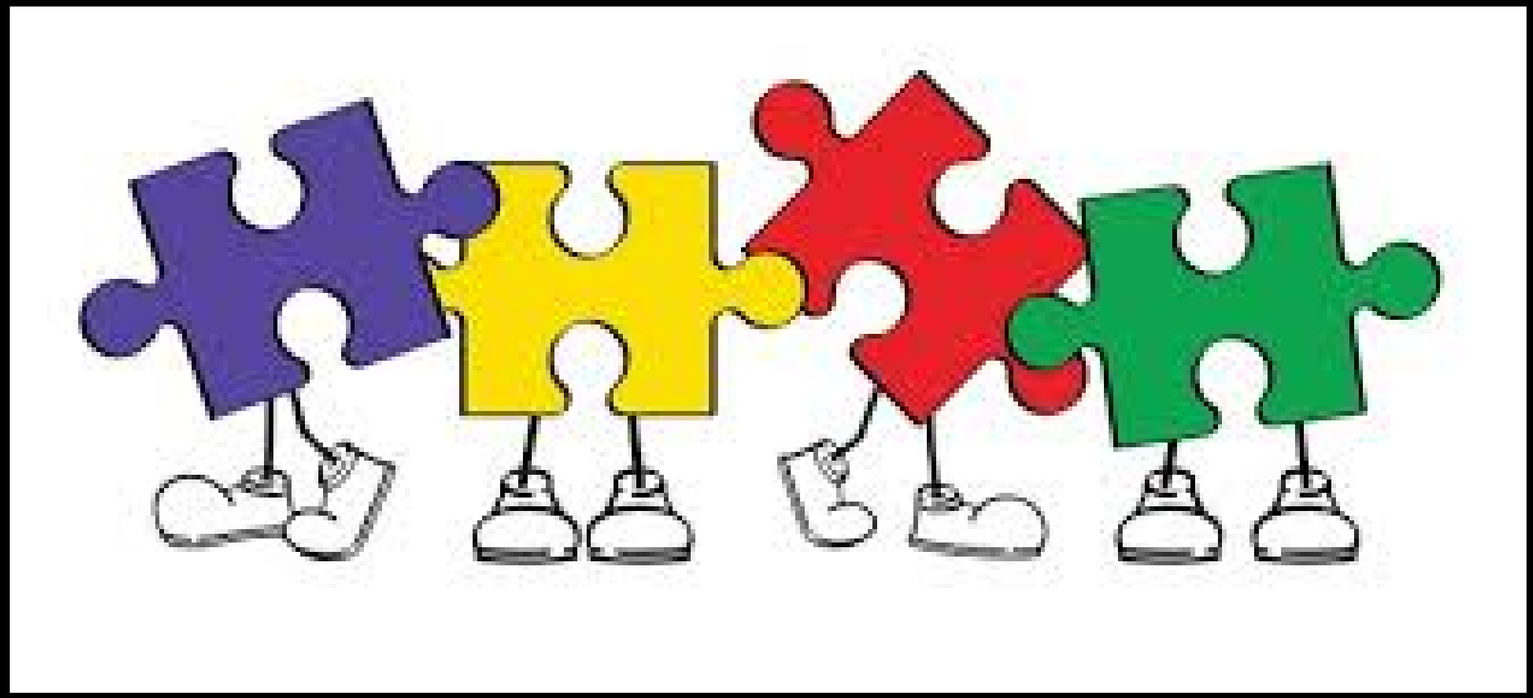
Attendance and Announcements



- 🍕Pizza at the end as always :)

Greedy Algorithm Introduction

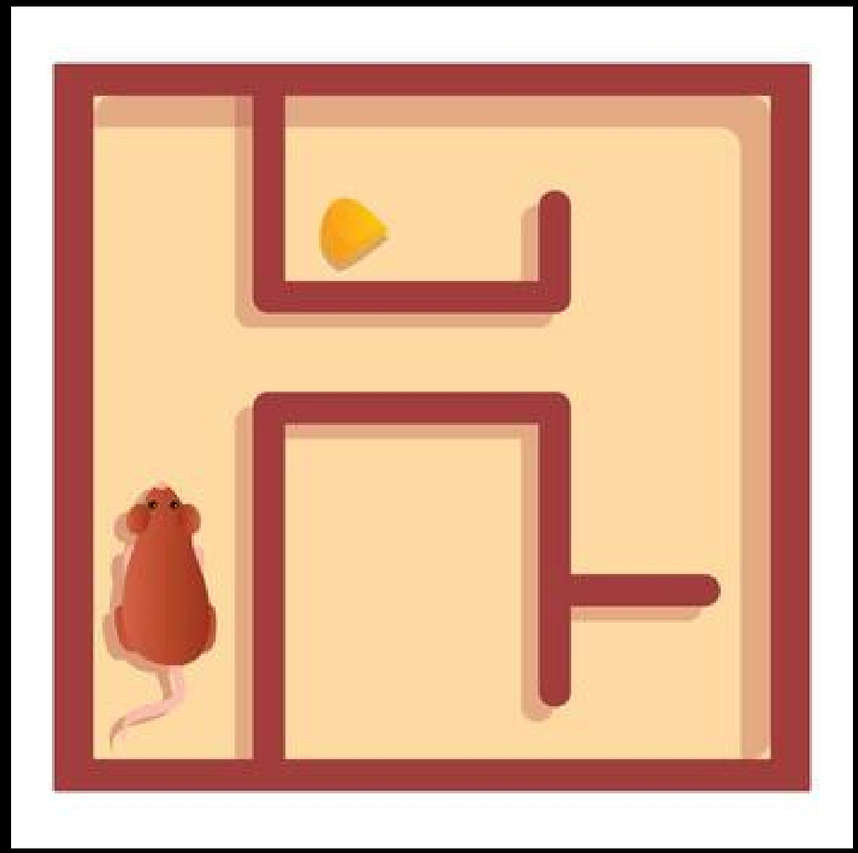
When we have a large complex problem, we can split it up into several small decisions



Greedy Algorithm Introduction

A rat solving a large maze can be split into a long string of decisions where the rat either:

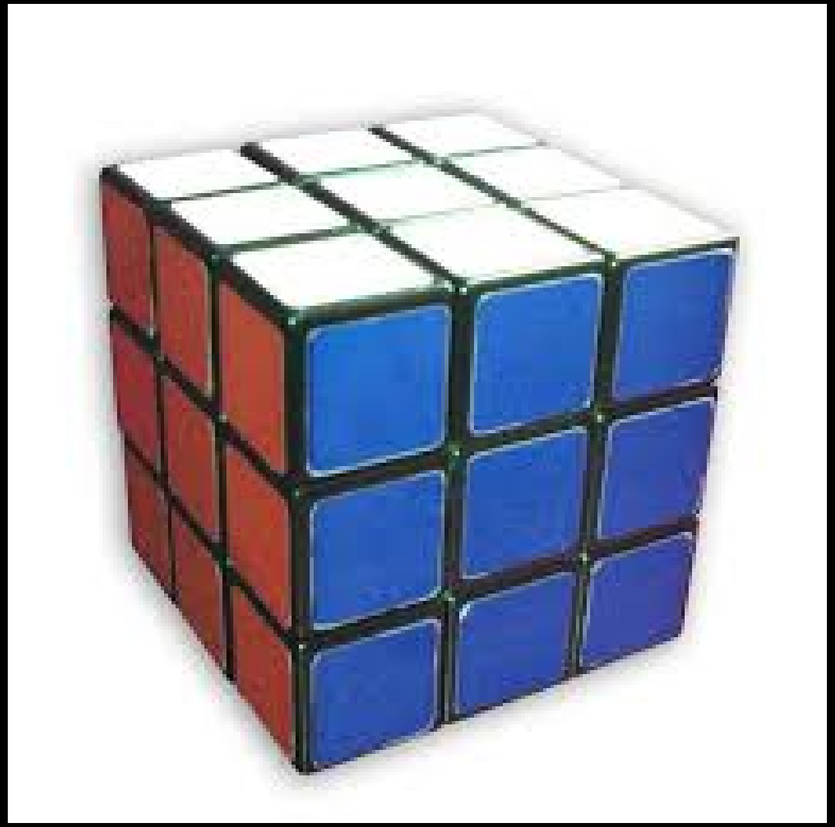
-Goes Forward
-Goes Left
-Goes Right
-Goes Backward



Greedy Algorithm Introduction

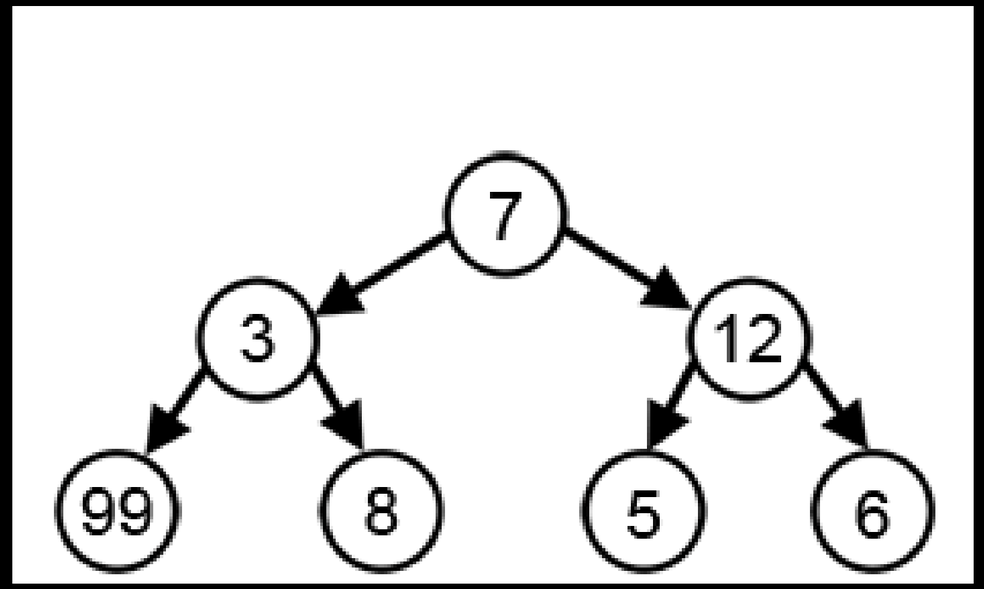
Solving a Rubik's Cube can be split into a long string of several different turns where for each one we can:

Take either of the six sides and spin them either clockwise or counterclockwise (12 decisions)



Greedy Algorithm Introduction

We start on top of a tree and want to move down, collecting the greatest sum we can



Starting from 7, at each state we can:

-Choose the node to the left
-Choose the node to the right

Greedy Algorithm Introduction

In a game of snake, we want to collect all the apples. This can be broke down into a long string of decisions where we either:

-Continue forward
-Turn right
-Turn left



Greedy Algorithm Introduction

We can see how large, complex problems can be split up into small isolated decisions!

A common approach to solve these problems is a Greedy Algorithm

BUT, only in certain cases

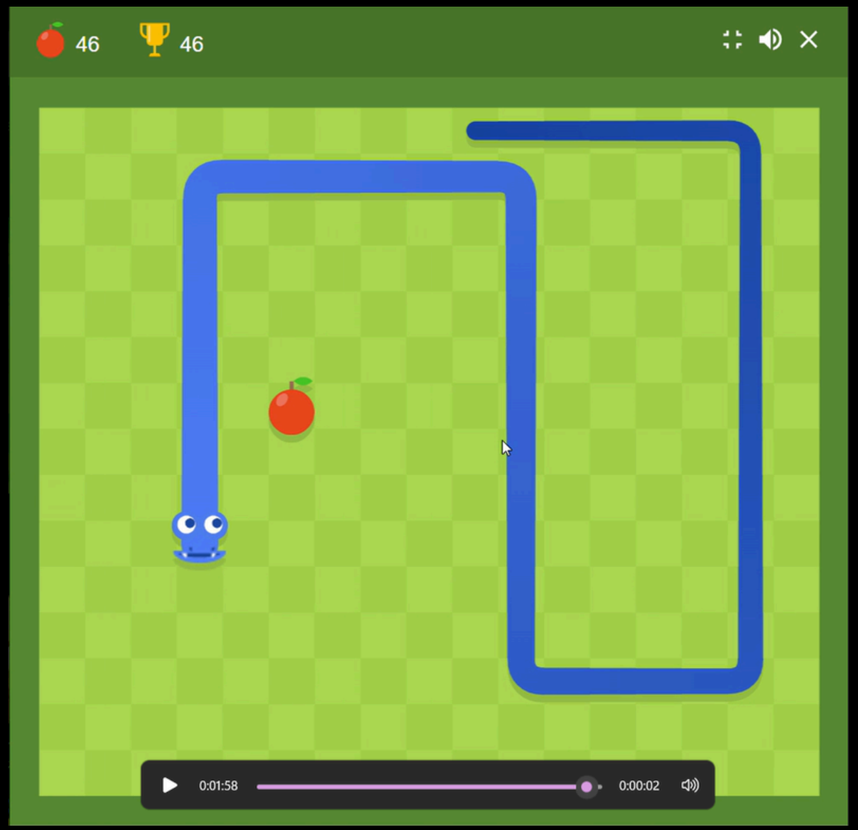
Greedy Algorithm

A Greedy Algorithm is a type of algorithm that at every single step will make the best decision available to it.

Lets go back through our examples and see if it would work or fail!

Revisiting Snake

Notice that the two snakes are in exactly the same position and have exactly the same options available

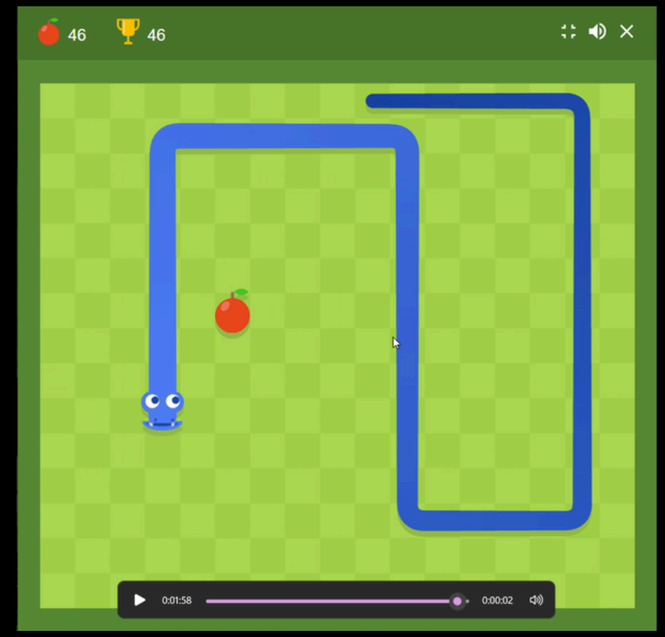




Revisiting Snake

The core requirement for Greedy Algorithms to work is being able to divide a large problem (getting all the apples) into smaller isolated problems (deciding which direction to turn at each step)

Can we truly break Snake up into isolated subproblems?



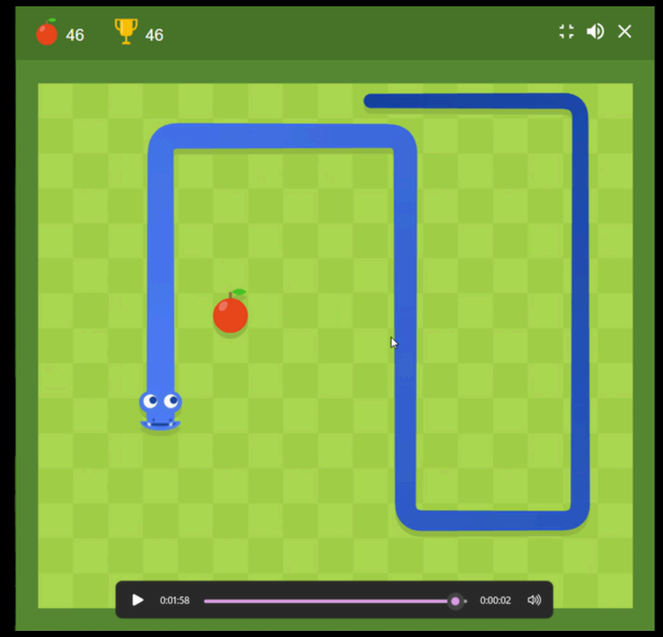


Revisiting Snake

No: The previous decisions we made impact our current decision (because of the tail of the snake)

If every previous decision is relevant to our current decision, we can NOT truly split Snake up into isolated subproblems, because every decision impacts the next.

Snake is NOT Path Independent



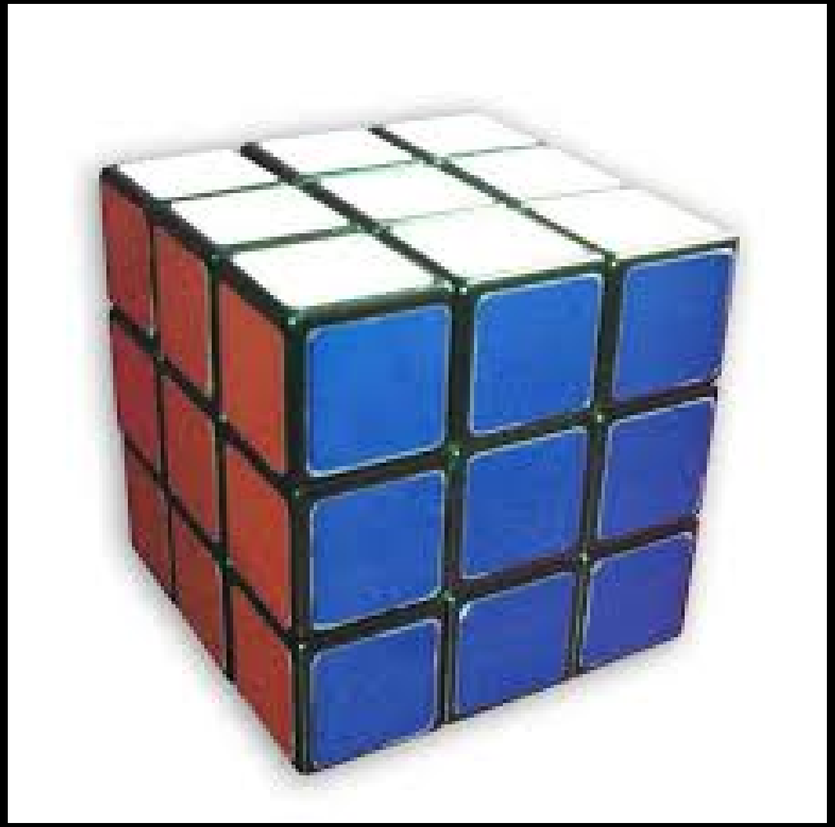


Revisitng the Rubik's Cube

Is a Rubik's Cube path independent or not?

It is! For any orientation of the Rubik's cube, it does not matter what combination of turns it took us to get there.

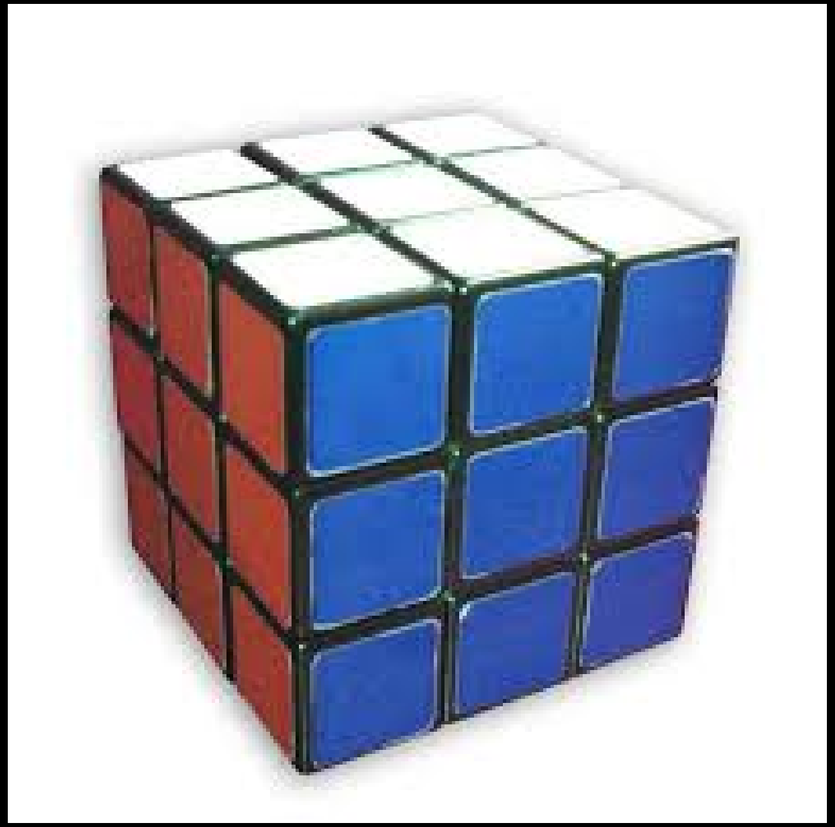
This means we can treat each decision as its own completely isolated subproblem



Revisitng the Rubik's Cube

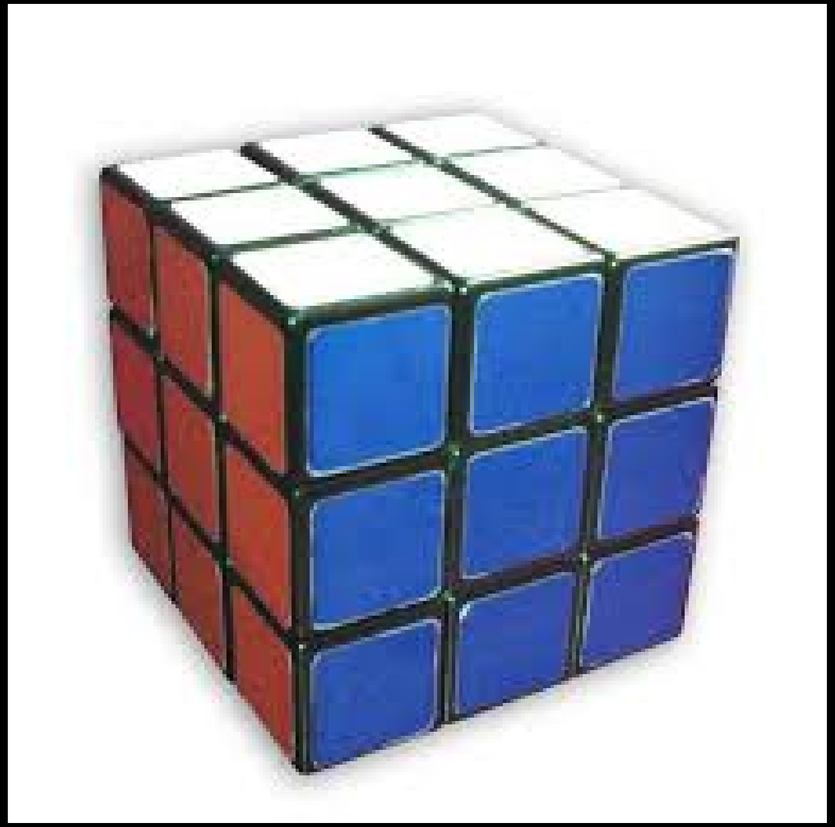
Remember, a Greedy Algorithm will always perform the best option available to it for each subproblem

but... What is the best option in a Rubik's Cube?



Revisitng the Rubik's Cube

Because there is no reliably way to measure how good or bad any decision is when making it, we cannot use a Greedy Algorithm

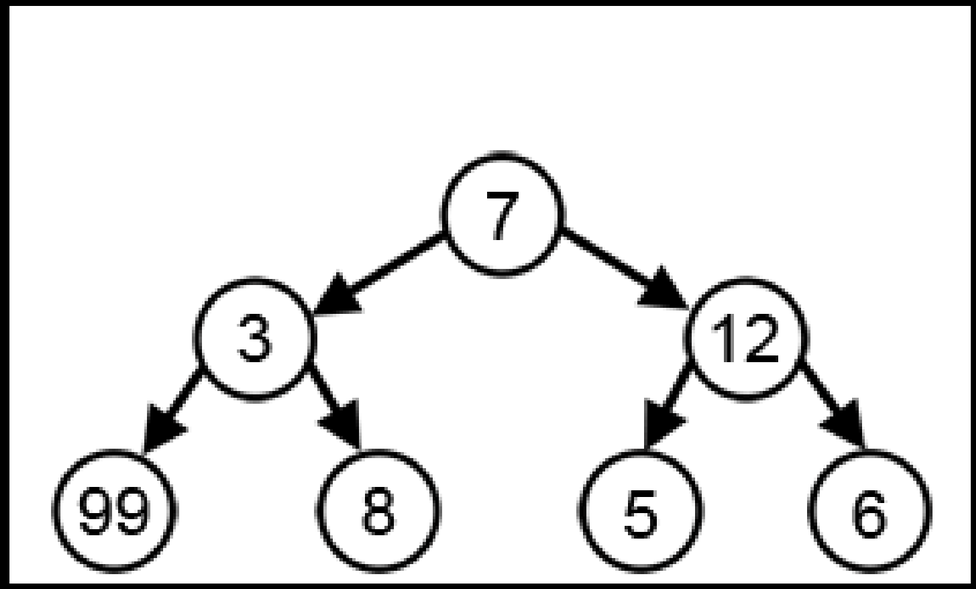


Revisiting the Tree

Is this path independent?

Can we measure how good each choice we make is?

Yes! This means we can split this problem up into several isolated subproblems, and solve each of those as best as we can!

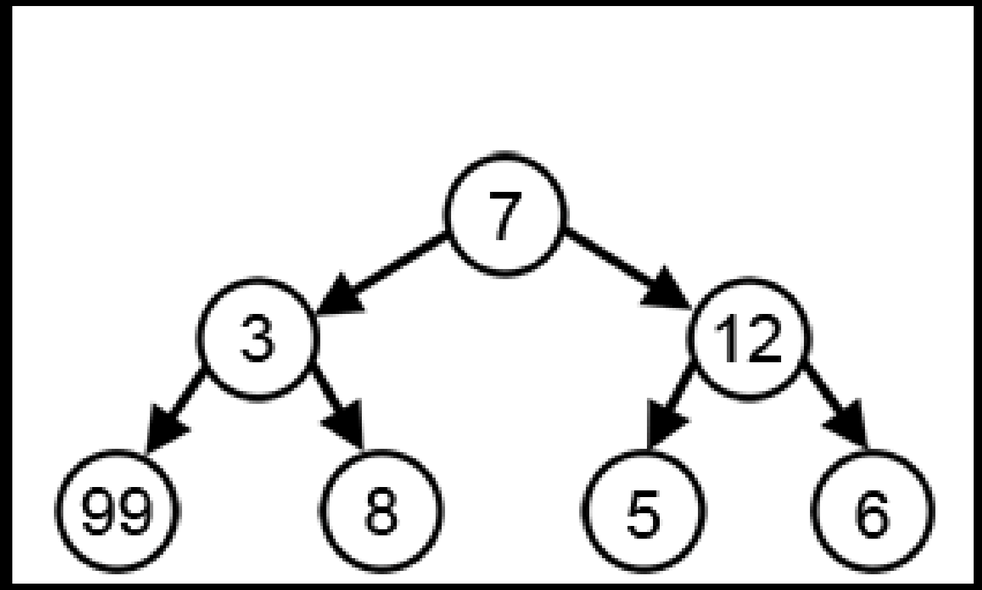


Revisiting the Tree

However, does solving each subproblem actually solve the entire puzzle?

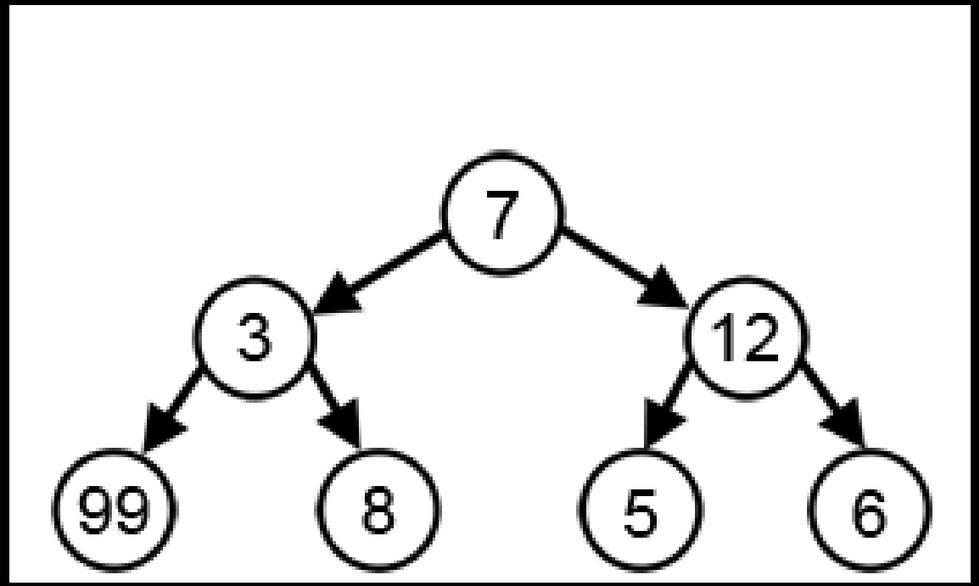
Start at 7:
Choose 12 → Choose 6

We made the best possible choice at each step, but still didn’t solve the overall puzzle optimally



Revisiting the Tree

We can NOT reach the best solution to the puzzle by making the best decision for each subproblem (optimal decisions locally do NOT gurantee optimal decisions globally)



Greedy doesn’t work

When Greedy?

1. The problem MUST be path-independent, meaning the previous decisions we made to get to our current decision bear no impact on our current decision

2. We must be able to quantify how good or bad a decision is when making it (Remember, Greedy is defined as always making the best decision for each subproblem)

3. The solution to the problem must be reachable by solving every subproblem

The Warmup

https://leetcode.com/problems/lemonade-change/

860. Lemonade Change

Easy | Topics | Companies

At a lemonade stand, each lemonade costs `$5`. Customers are standing in a queue to buy from you and order one at a time (in the order specified by bills). Each customer will only buy one lemonade and pay with either a `$5`, `$10`, or `$20` bill. You must provide the correct change to each customer so that the net transaction is that the customer pays `$5`.

Note that you do not have any change in hand at first.

Given an integer array `bills` where `bills[i]` is the bill the `i^th` customer pays, return `true` *if you can provide every customer with the correct change, or* `false` *otherwise.*

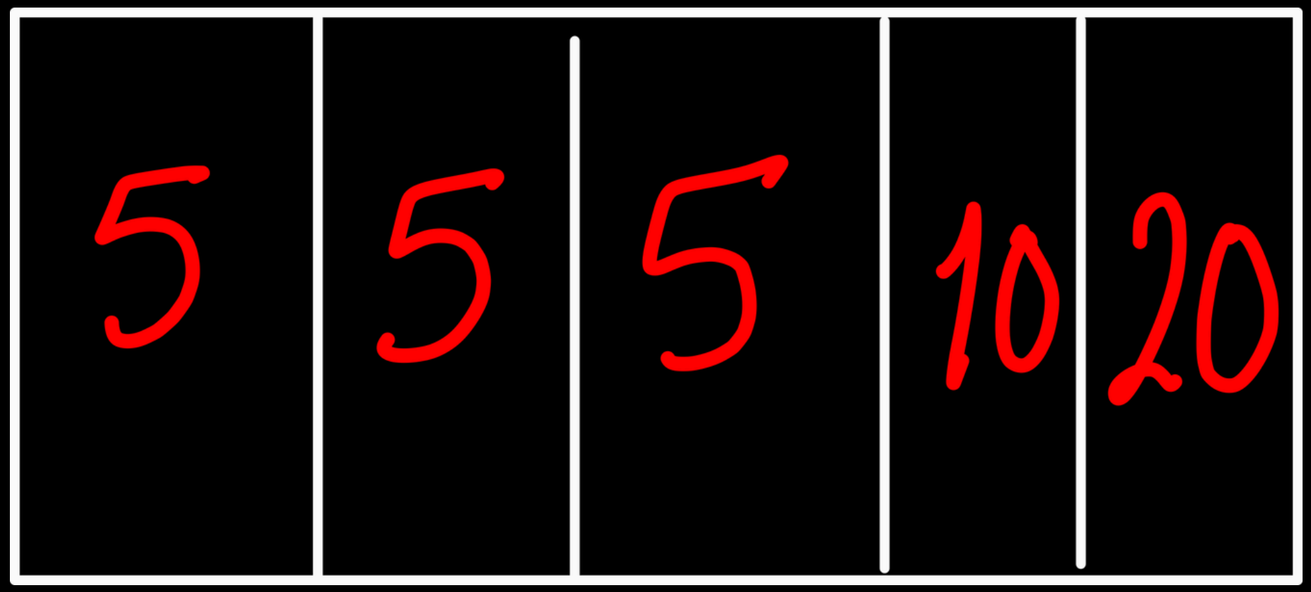
What’s required?

INPUT: We’re given an array of the integers: 5, 10, 20.

OUTPUT REQUIRED:
We will determine
, assuming the integers are dollar bills, the price is $5, and that we’re starting from zero,
 if we could return all the change required.

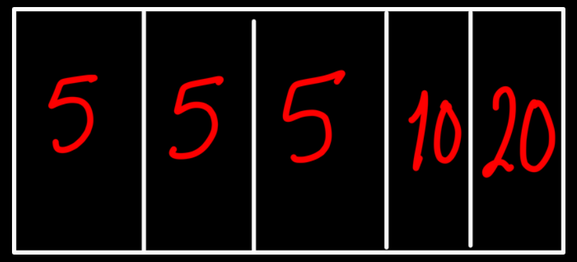
Example Test Cases:

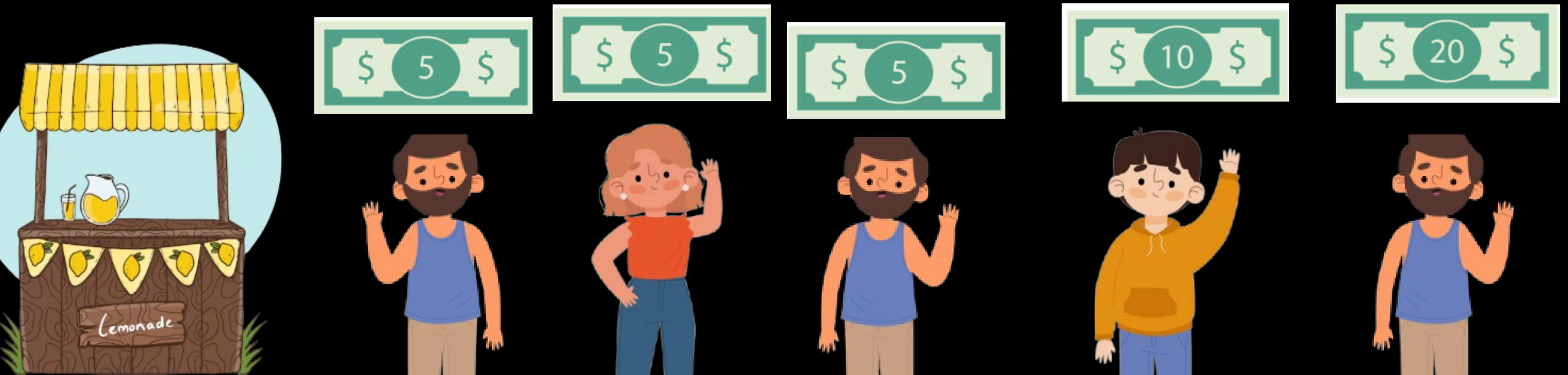
Let’s go over the example test cases, to make it clearer what we’re required to do.



Example Test Cases:

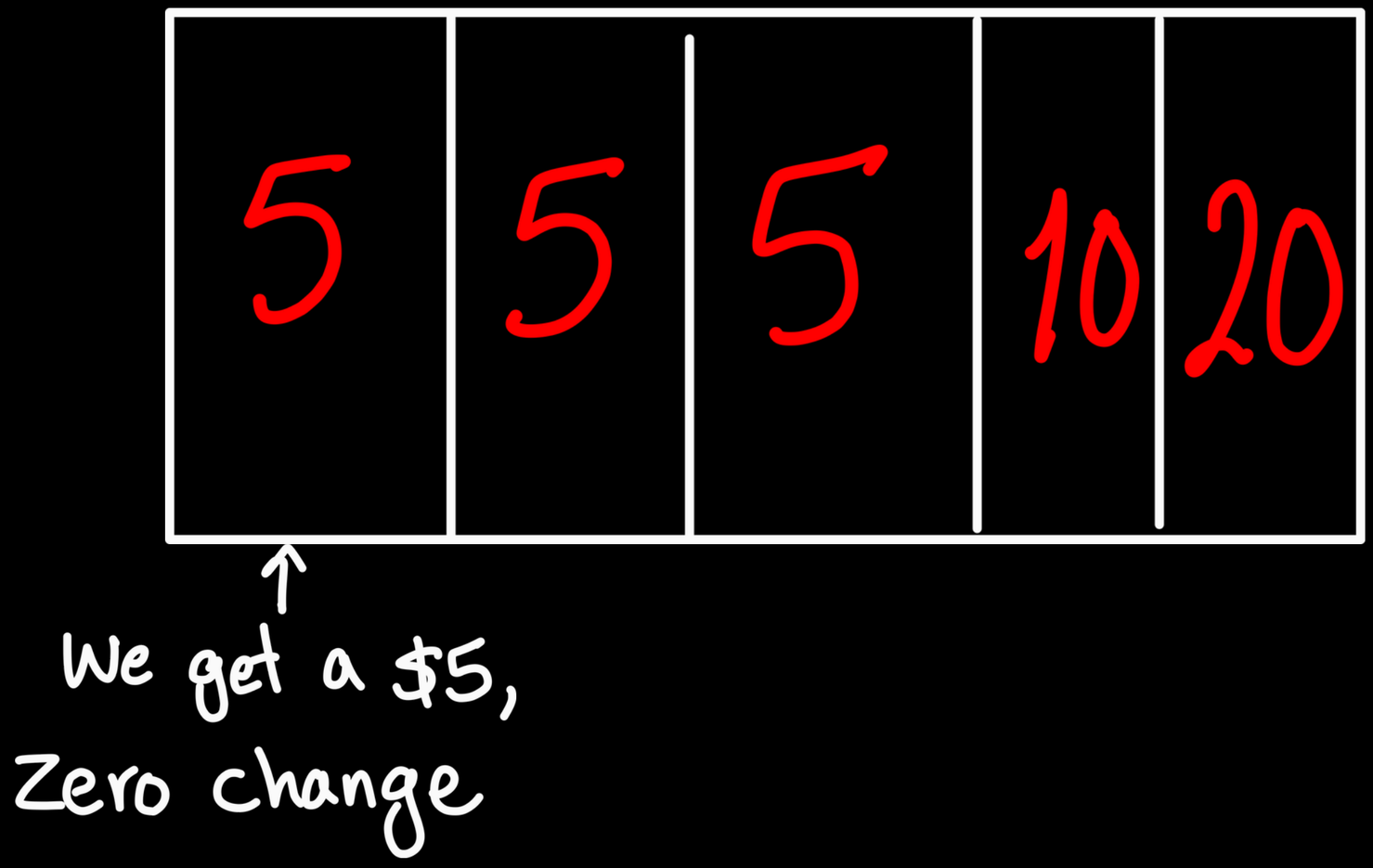
Imagine a line of people at the lemonade stand with only one 5,10 or 20 bill and they all want only 1 lemonade which costs $5



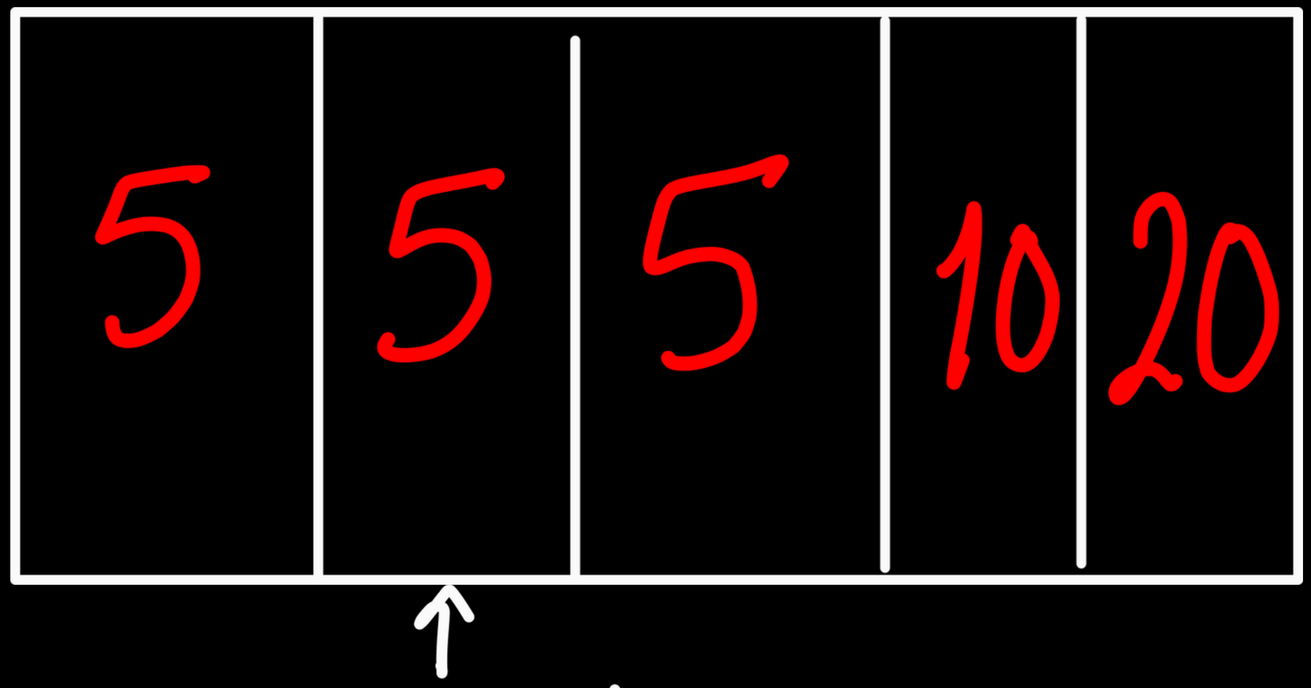


Example Test Cases:

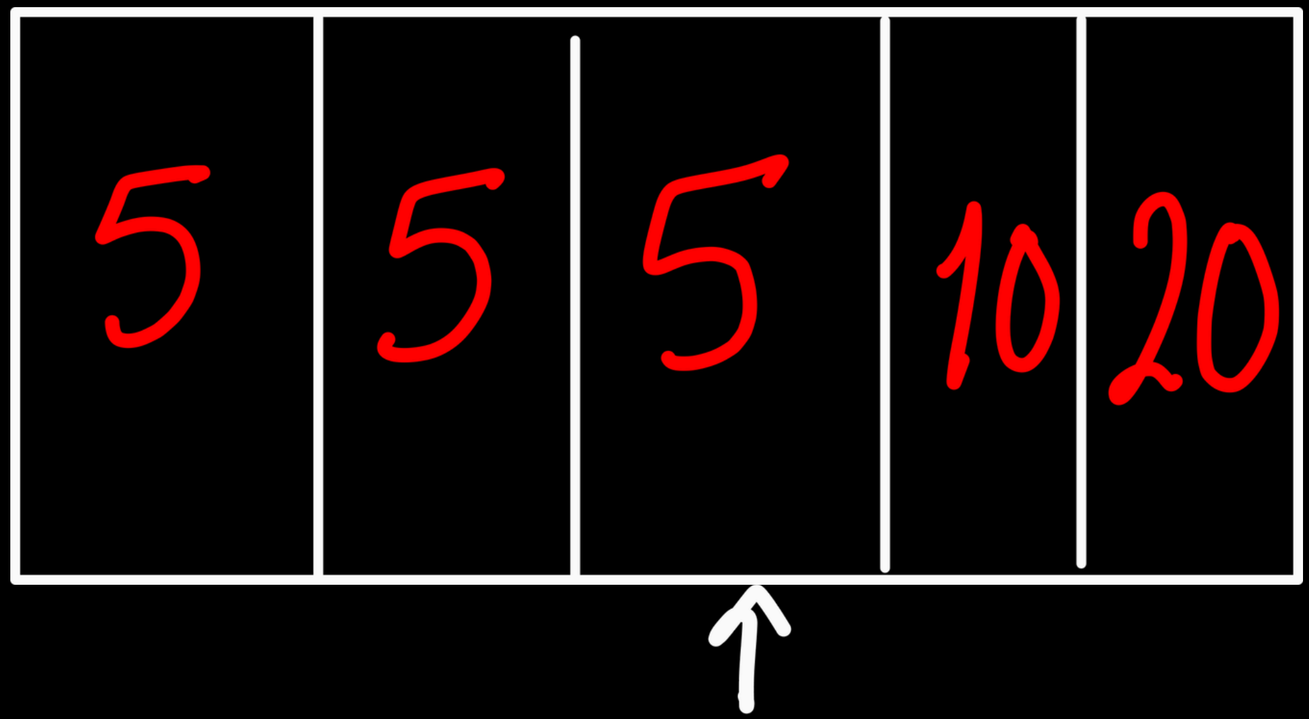
Let’s go over the example test cases, to make it clearer what we’re required to do.



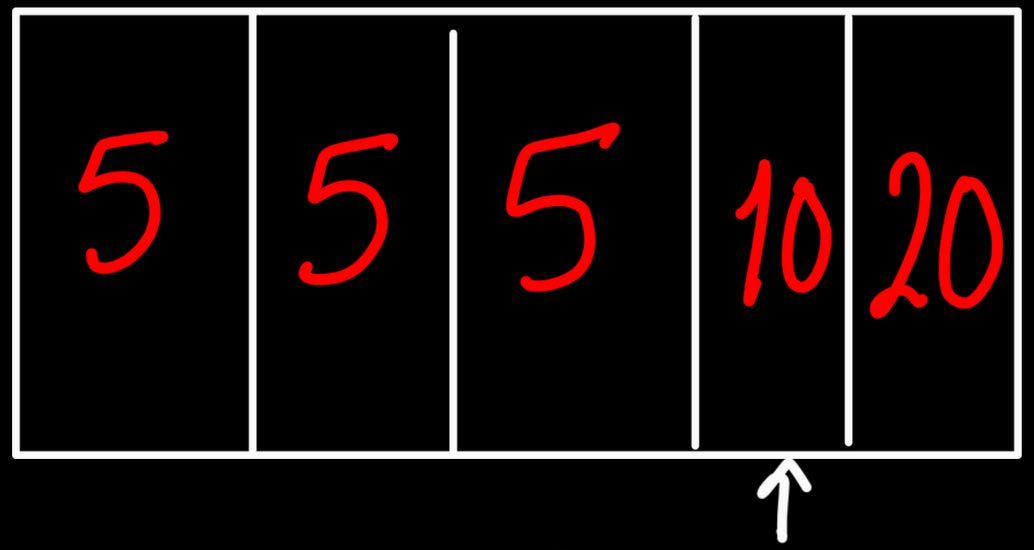
Example Test Cases:



Example Test Cases:

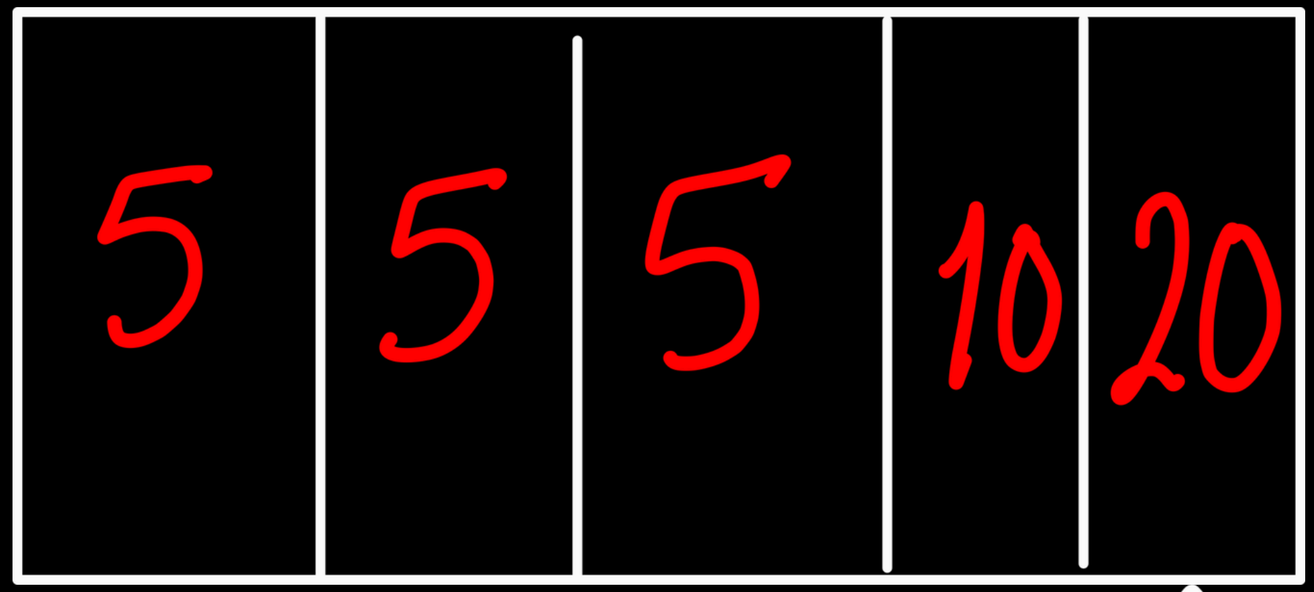


Example Test Cases:



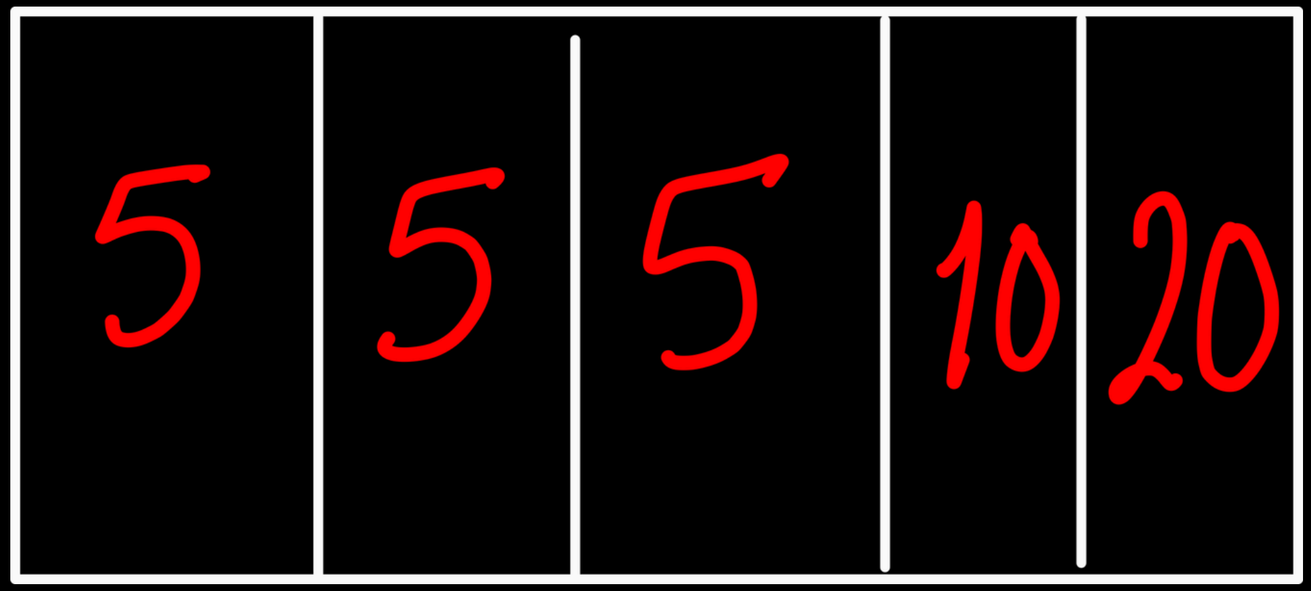
Got a $10 bill.
Take out one of the three fives we have.
We now have $10 and two fives.

Example Test Cases:



Take a five and a ten.
We now only have a five
and a twenty.

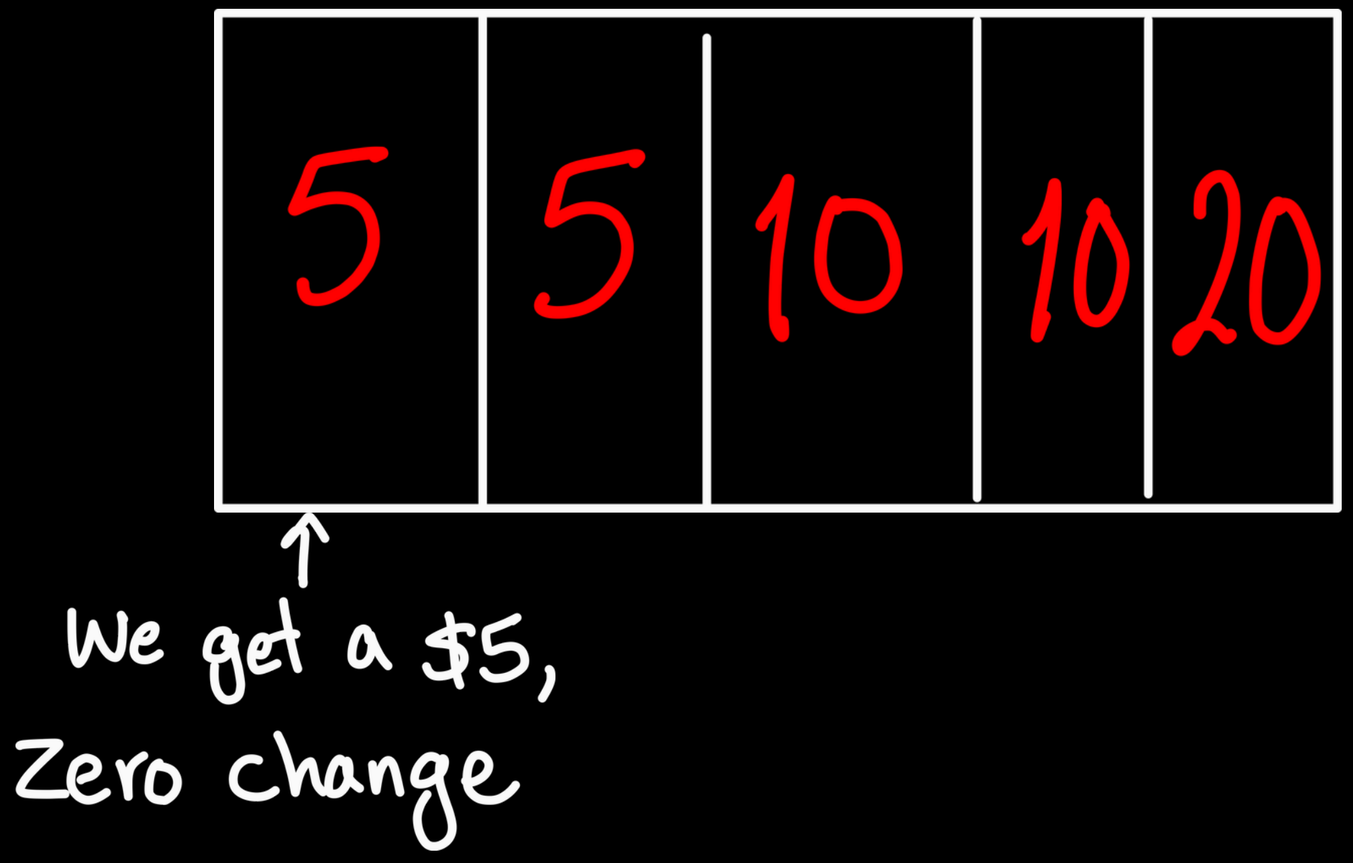
Example Test Cases:



We end up with 1 $20 bill and 1 $5 bill.
We were able to return all the change required, so this will RETURN True.

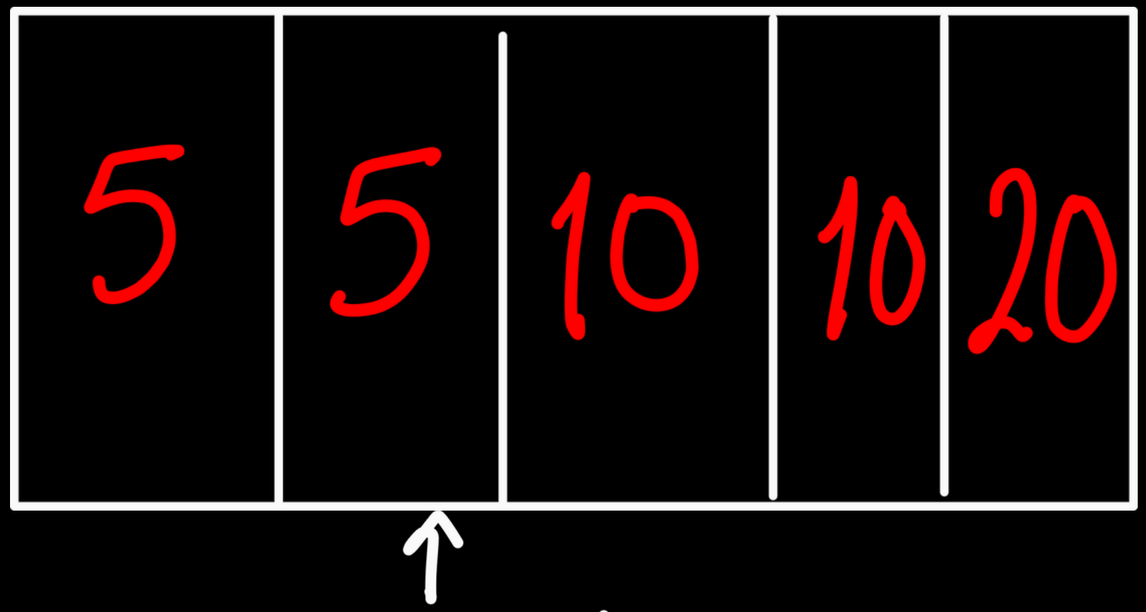
Example Test Cases:

Second Test Case:



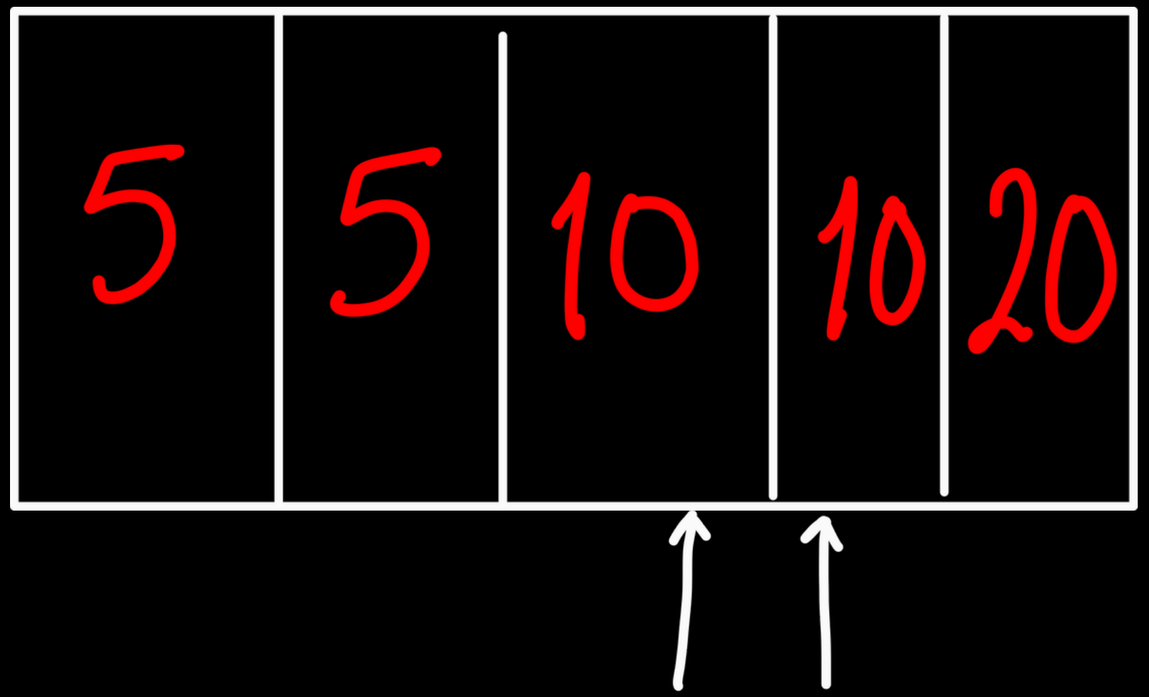
Example Test Cases:

Second Test Case:



Example Test Cases:

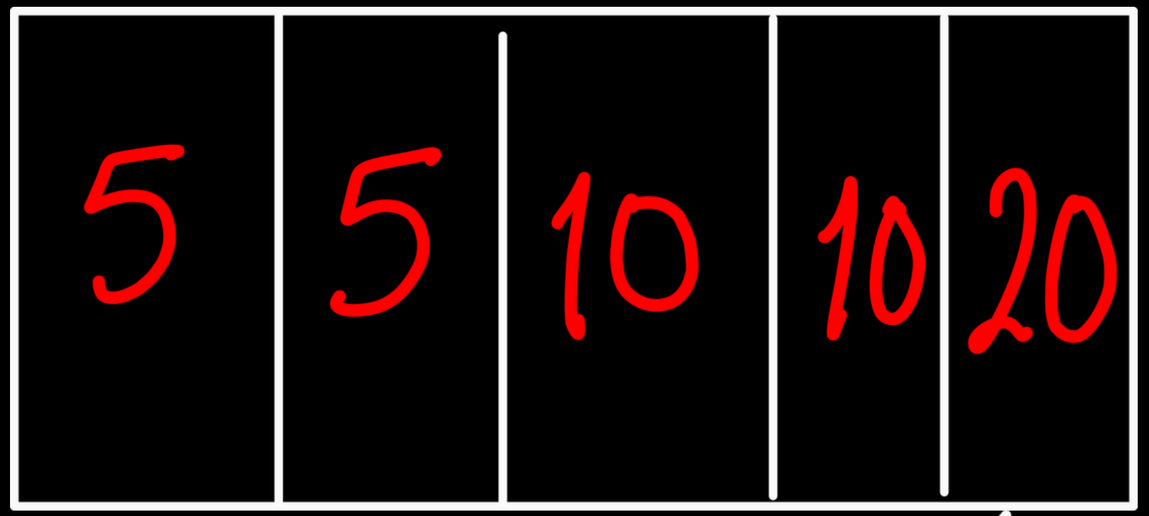
Second Test Case:



Two $10 bills,
Take out $5 for each one.
No fives left. Only two tens.

Example Test Cases:

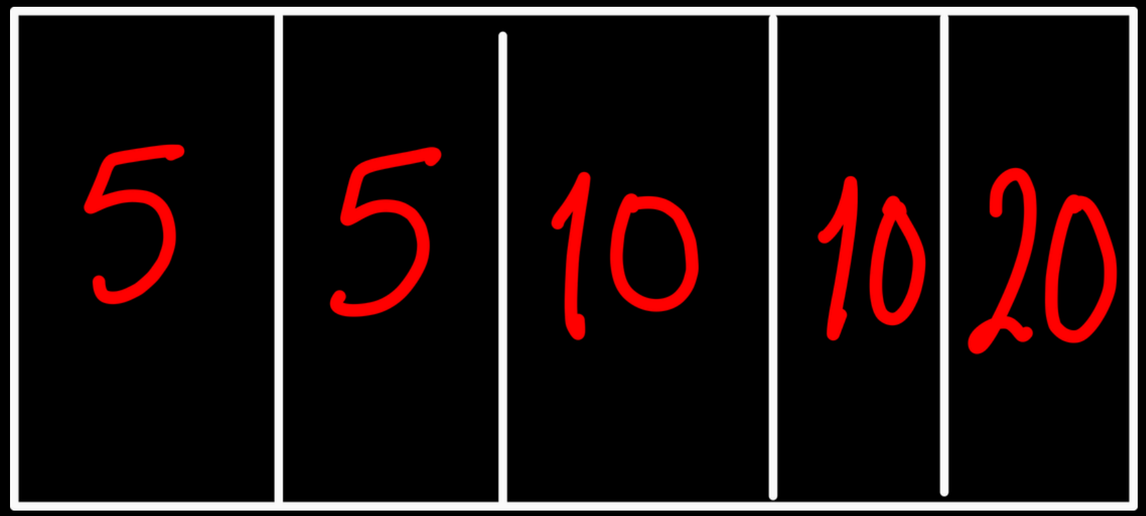
Second Test Case:



A Twenty requires a change of ten + five, or three five's. Neither is available

Example Test Cases:

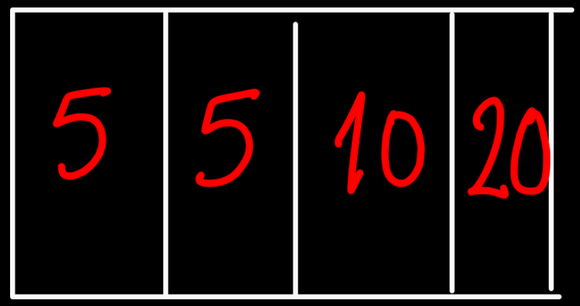
Second Test Case:



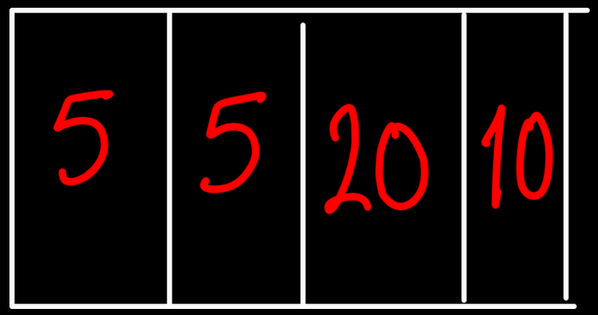
This testcase will, therefore, return False.

Helpful insights before you start

Order Matters:



Returns True.



Returns False.

Iterate through the whole array.
Apply the greedy concepts you just learned.

GO FOR IT!

Solution: Setup

Let’s go through the array step by step, and see if we have enough change for each bill.

But we’ll need to track how many bills we already have.
Luckily, we don’t need to track $20 bills since they can’t be used for change.

We’ll initialize two variables for fives and tens:

```
int five = 0;
int ten = 0;
```

Solution: Loop

Next, let’s iterate over the array and, for each element, check if we have enough change for it.

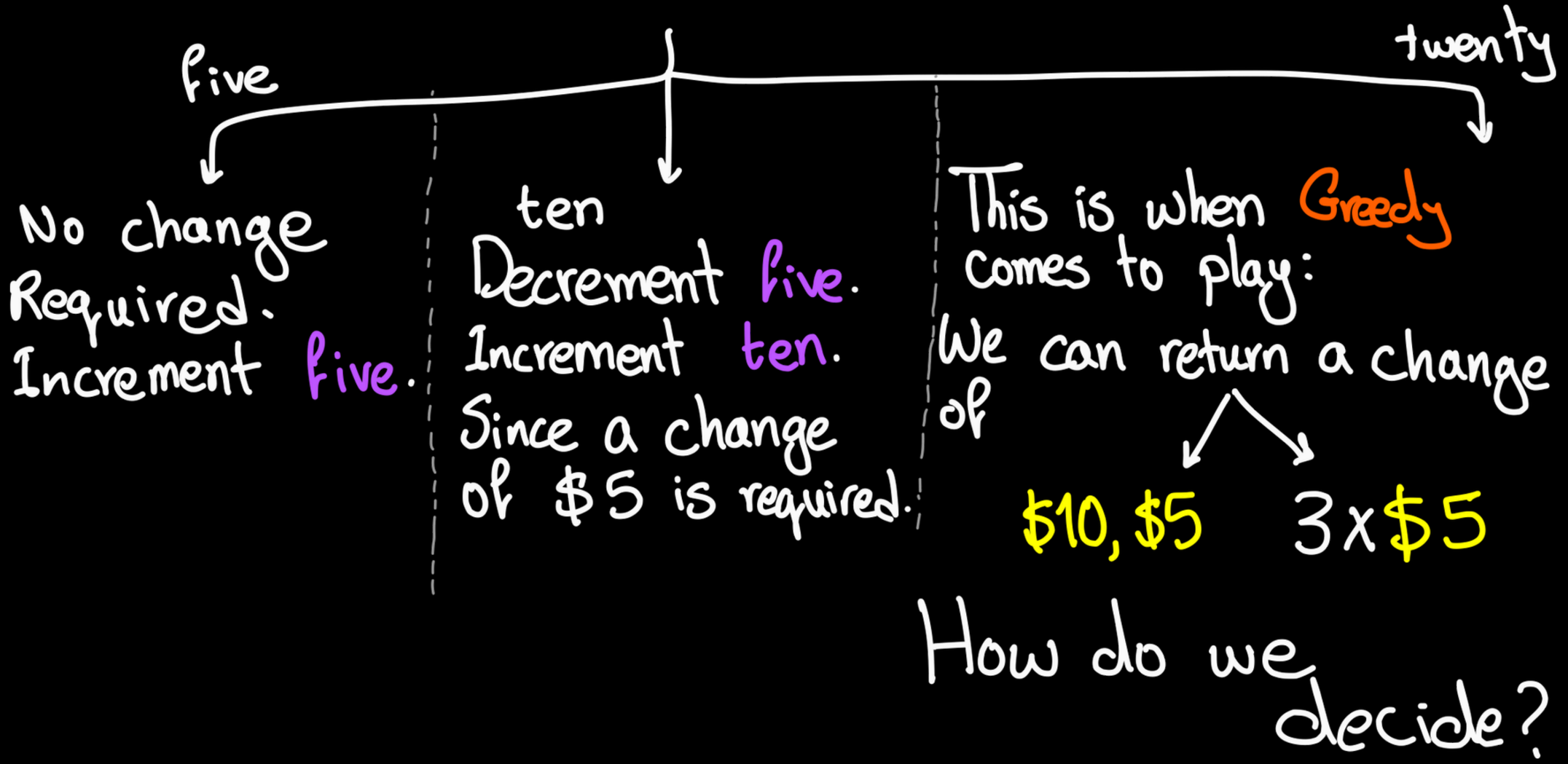
There’re only 3 integers we can encounter as per the constraints:

Constraints:

- `1 <= bills.length <= 10^5`
- `bills[i]` is either `5`, `10`, or `20`.

Solution: Loop

For each element:



Solution: Greedy

A $10 bill can only be used to return change for a $20 bill.
While a $5 bill can return change for both the $10 bill and the $20 bill.

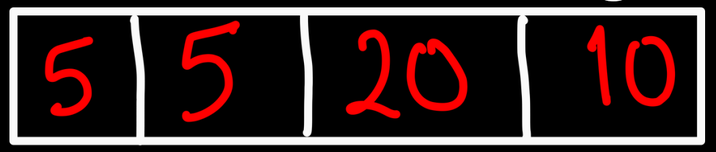
Therefore, to be greedy, it would make more sense to check if we have a $10 bill first,
and if we don’t, we could take out 3 $5 bills.

Solution: Loop Continued

We could simply check after the loop is finished if either of the counters is negative.

That means we needed more bills than we could have and couldn't return the change, so we'd return False.

However, in some test cases, we could return false before we go over the whole array, like this one:



So, we could check **inside the loop** if we have negative counters, and return false if that's the case, to be more efficient

Solution: Finale

```
class Solution {
    public boolean lemonadeChange(int[] bills) {
        int five = 0;
        int ten = 0;
        for (int i = 0; i < bills.length; i++){
            switch(bills[i]){
                case 5:
                    five ++;
                    break;
                case 10:
                    five --;
                    ten ++;
                    break;
                case 20:
                    five --;
                    if(ten > 0){
                        ten --;
                    }else{
                        five -=2;
                    }
                    break;
            }
            if (five < 0 || ten < 0){
                return false;
            }
        }
        return true;
```

The Problem

https://leetcode.com/problems/gas-station/

134. Gas Station

Medium | Topics | Companies

There are `n` gas stations along a circular route, where the amount of gas at the `i^th` station is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from the `i^th` station to its next `(i + 1)^th` station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays `gas` and `cost`, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* `-1`. If there exists a solution, it is **guaranteed** to be **unique**.

The Problem—

A Solved Test Case

Example 1:

> **Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]
> **Output:** 3
> **Explanation:**
> Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
> Travel to station 4. Your tank = 4 - 1 + 5 = 8
> Travel to station 0. Your tank = 8 - 2 + 1 = 7
> Travel to station 1. Your tank = 7 - 3 + 2 = 6
> Travel to station 2. Your tank = 6 - 4 + 3 = 5
> Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
> Therefore, return 3 as the starting index.

Let’s walk through an example!

It would help a lot to visualize the problem, so let’s do just that.

The Problem—

A Solved Test Case



Firstly, the Problem states that all the gas stations are on a circular path.

Let’s make a circular diagram for our problem.

Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
```

gas = [1,2,3,4,5]; <-

The Problem—

A Solved Test Case



The Problem says that there is gas[i] gallons of gas at each station with index i.

What does this mean?

We can represent this visually by putting gas[i] number of gallons at each station according to our testcase.

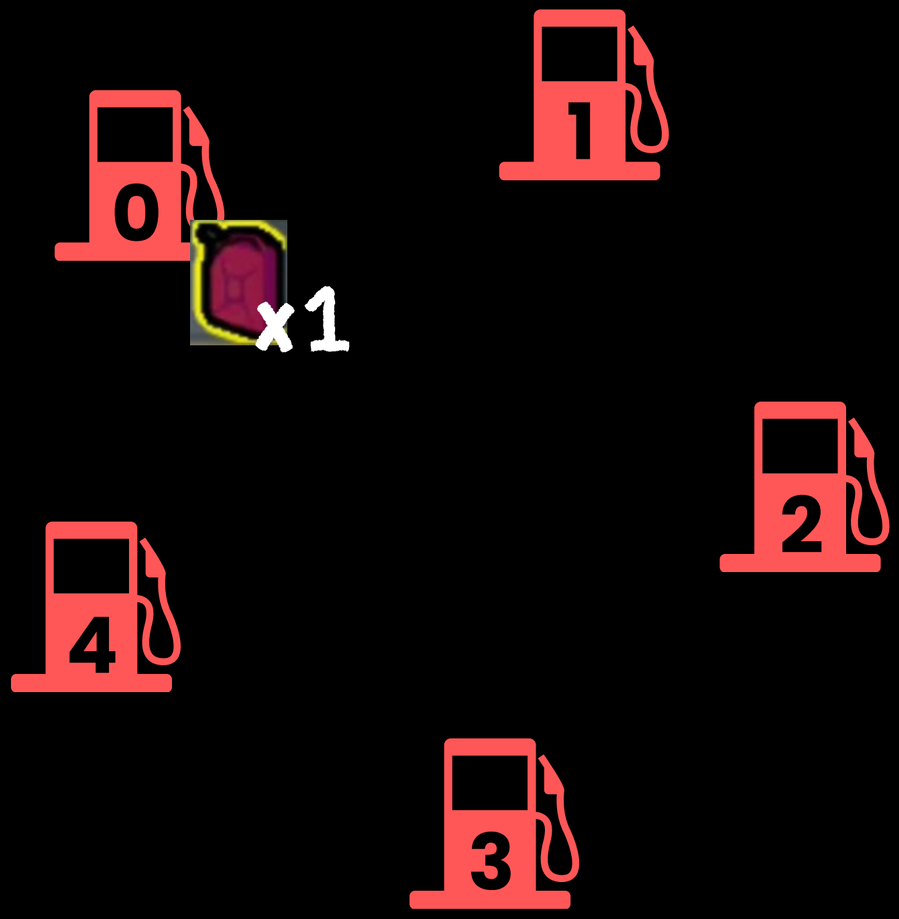
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

gas = [1,2,3,4,5]; <-

The Problem—

A Solved Test Case



Gas station 0 has 1 gallons of gas,
because gas[0] = 1.
Let’s put it in our visual.

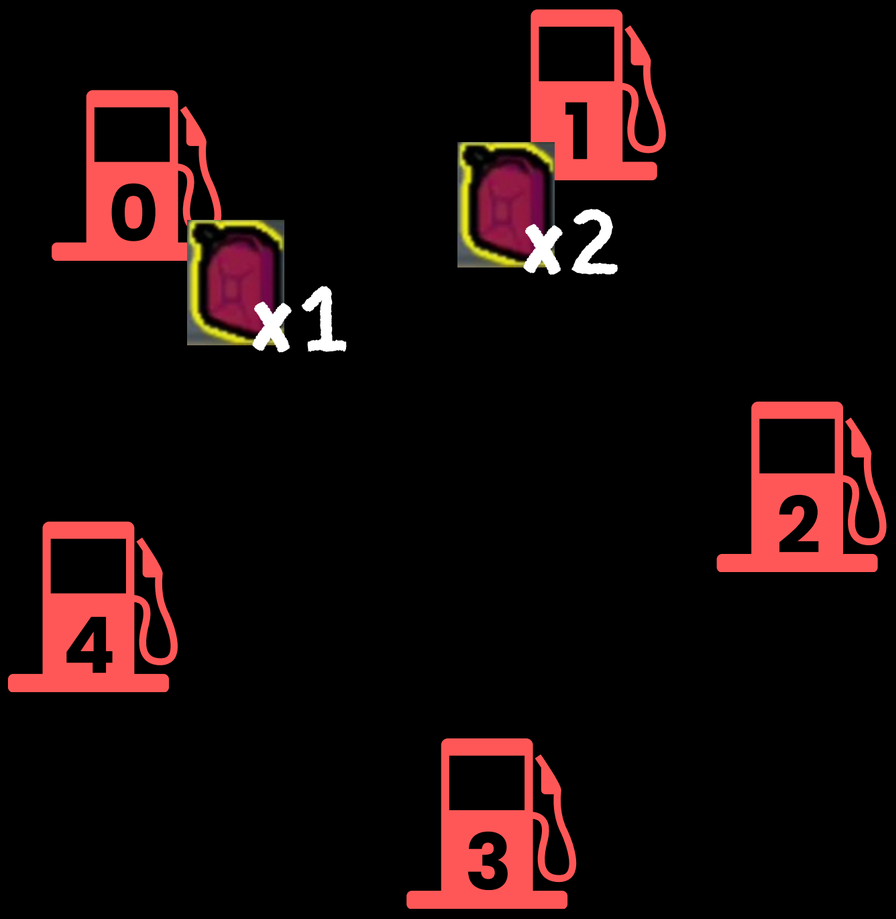
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

gas = [1,2,3,4,5]; <-

The Problem—

A Solved Test Case

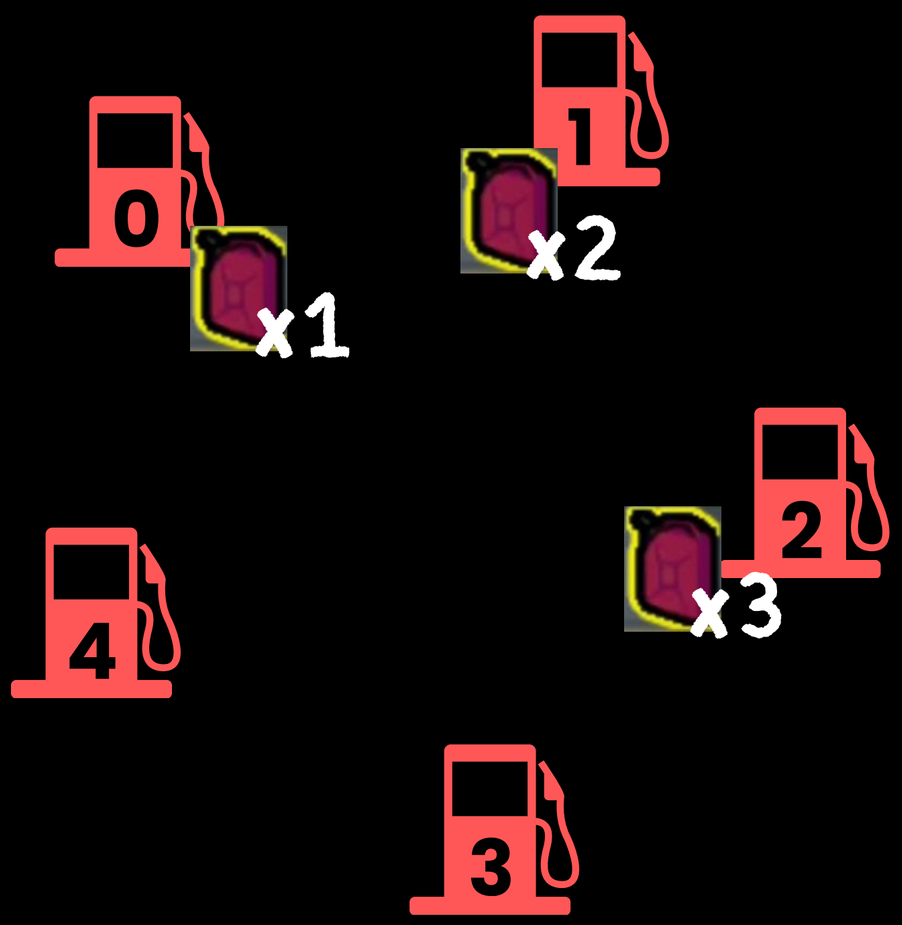


Gas station 1 has 2 gallons of gas,
because gas[1] = 2.
Let’s put it in our visual.

Example 1:

Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit

gas = [1,2,3,4,5]; <-



The Problem—

A Solved Test Case

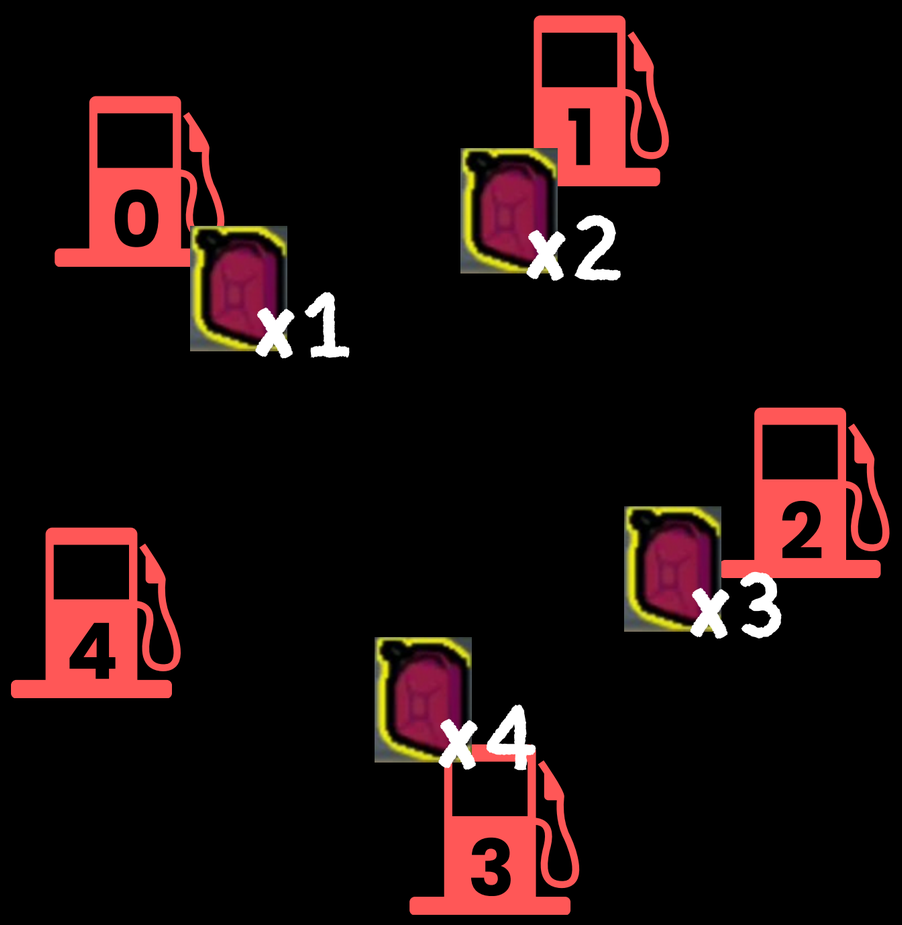
Gas station 2 has 3 gallons of gas,
because gas[2] = 3.

Let’s put it in our visual.

Example 1:

> **Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]
> **Output:** 3
> **Explanation:**
> Start at station 3 (index 3) and fill up with 4 unit

gas = [1,2,3,4,5]; <-



The Problem—

A Solved Test Case

Gas station 3 has 4 gallons of gas,
because gas[3] = 4.

Let’s put it in our visual.

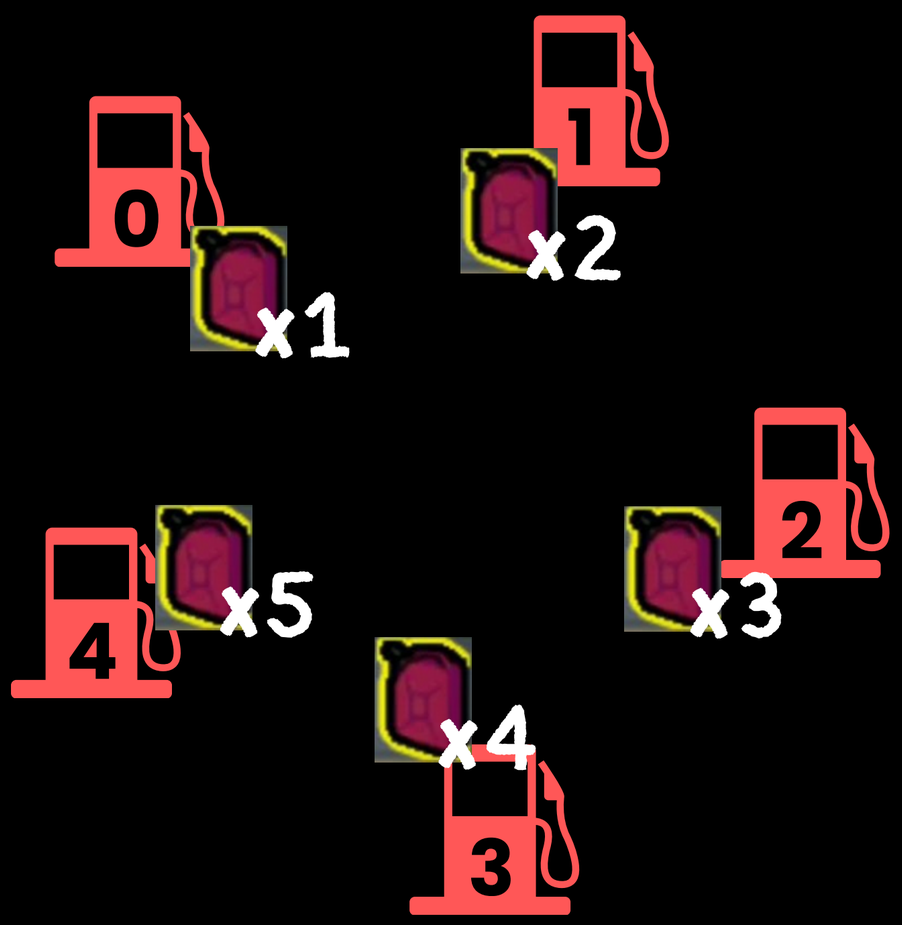
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

gas = [1,2,3,4,5]; <-

The Problem—

A Solved Test Case

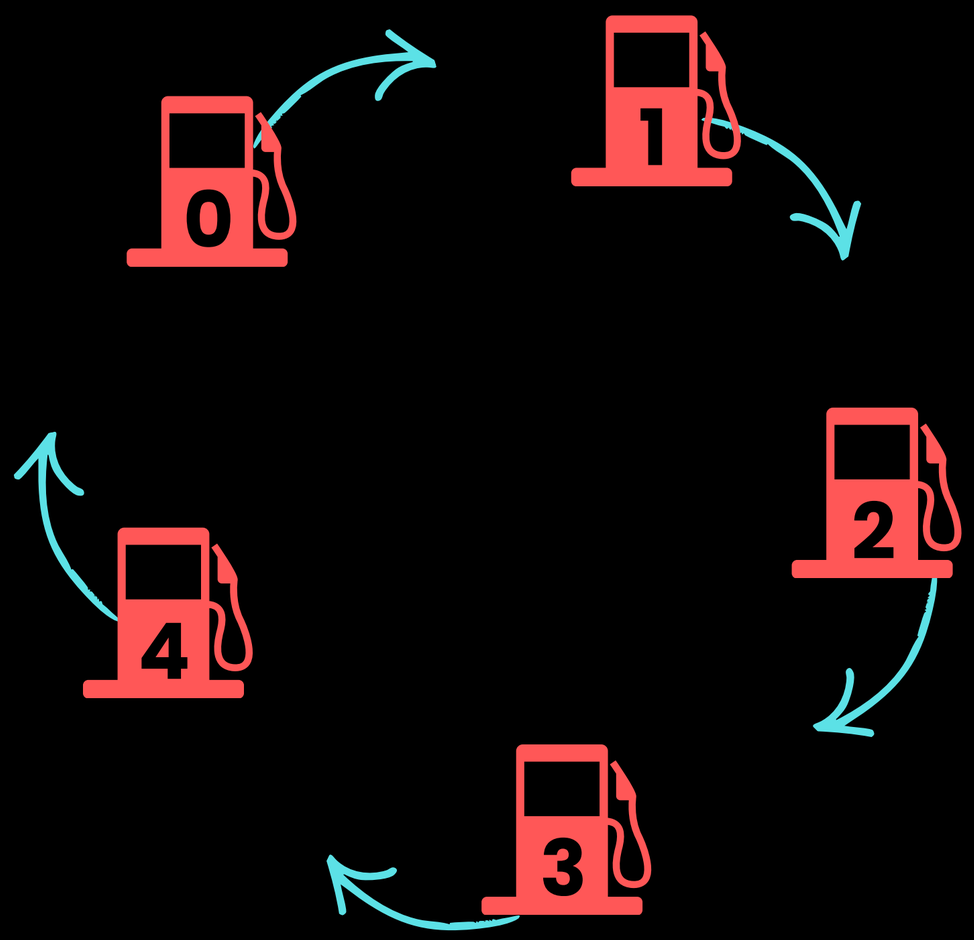


And we're done!

Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

```
gas  = [1,2,3,4,5];
cost = [3,4,5,1,2]; <-
```



The Problem—

A Solved Test Case

Now we have the cost!

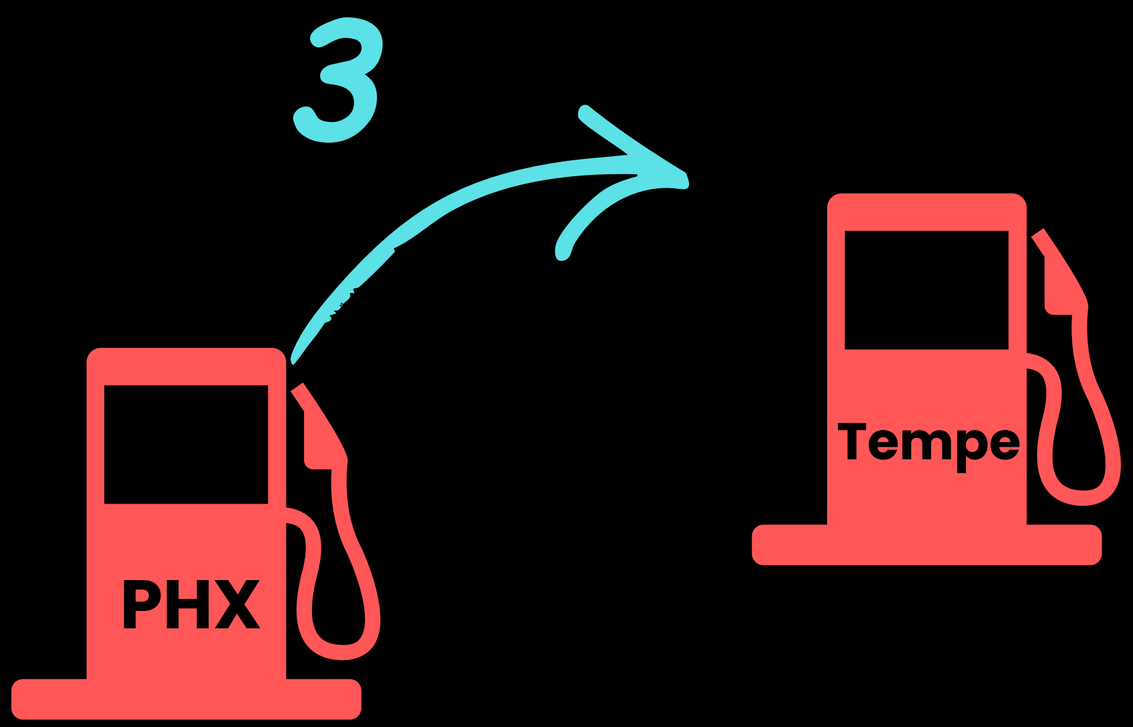
The problem words this a bit confusingly. They say that the value at cost[i] represents the gas needed to travel from location i to location i+1.

Example 1:

> **Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]
> **Output:** 3
> **Explanation:**
> Start at station 3 (index 3) and fill up with 4 unit

The Problem—

A Solved Test Case



What does that even mean??

Let’s just say it takes 3 gallons of gas to get from Phoenix to Tempe.

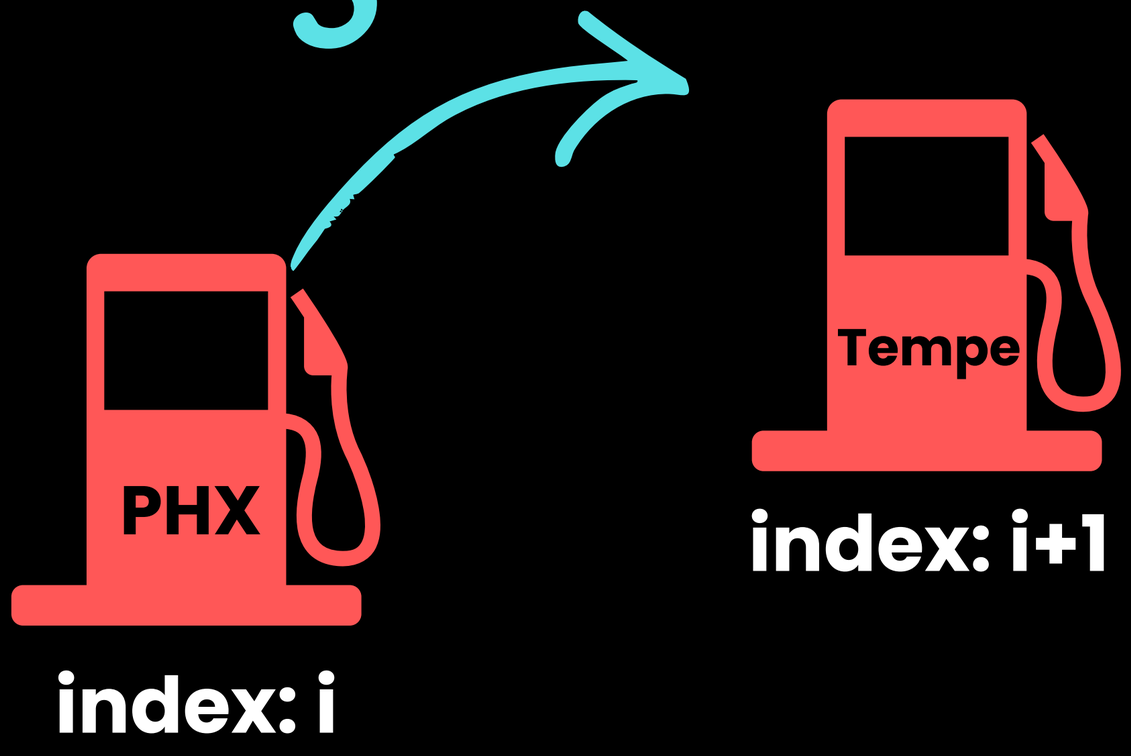
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

The Problem—

A Solved Test Case

```
cost[i] = 3;
```



If Pheonix were index i and Tempe were index i+1, then according to the Problem the gas needed would be stored at cost[i].

Example 1:

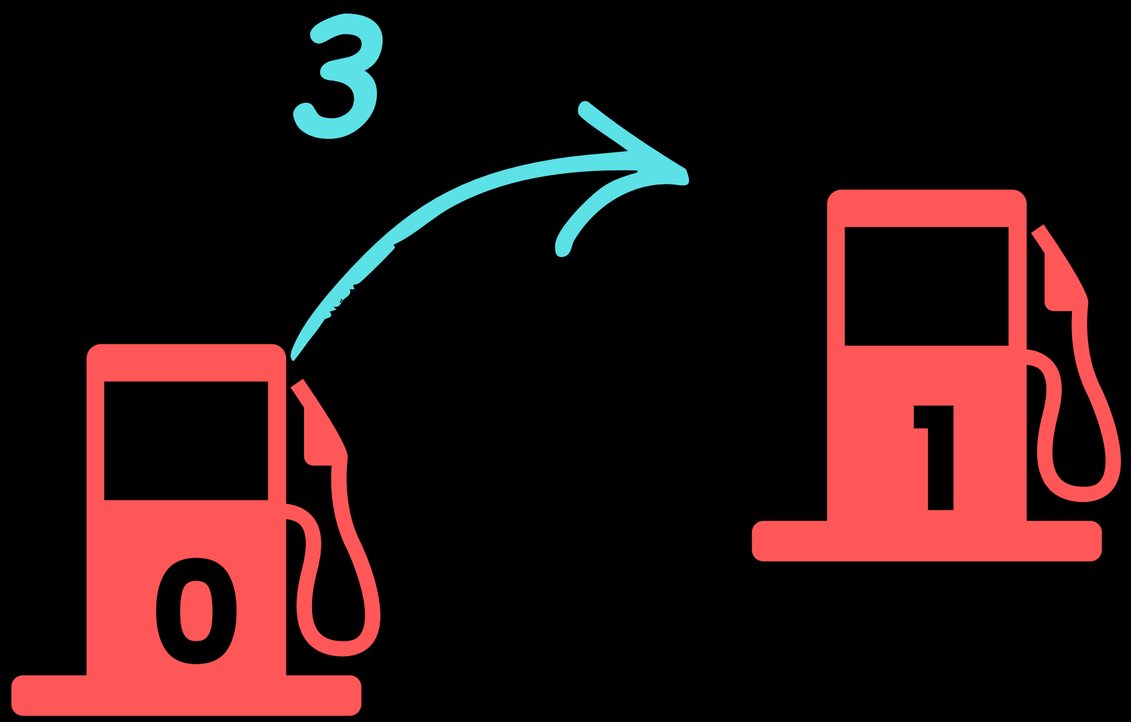
```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

The Problem—

A Solved Test Case

```
cost = [3, 4, ...];

cost[0] = 3;
```



To make it more generalized, here is how it would look from indexes 0 to 1.

Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

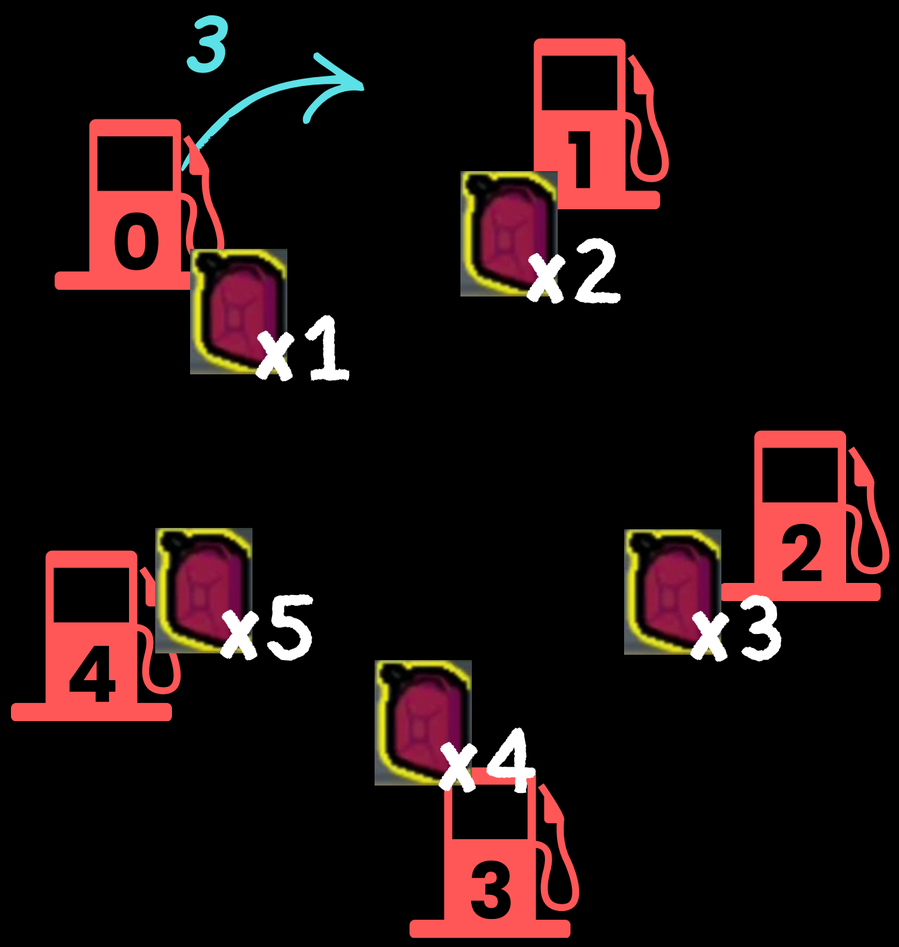
```
cost = [3,4,5,1,2];
```

The Problem—

A Solved Test Case

Looking at the big picture,
this is what our diagram looks like.

```
cost[0] = 3;
```



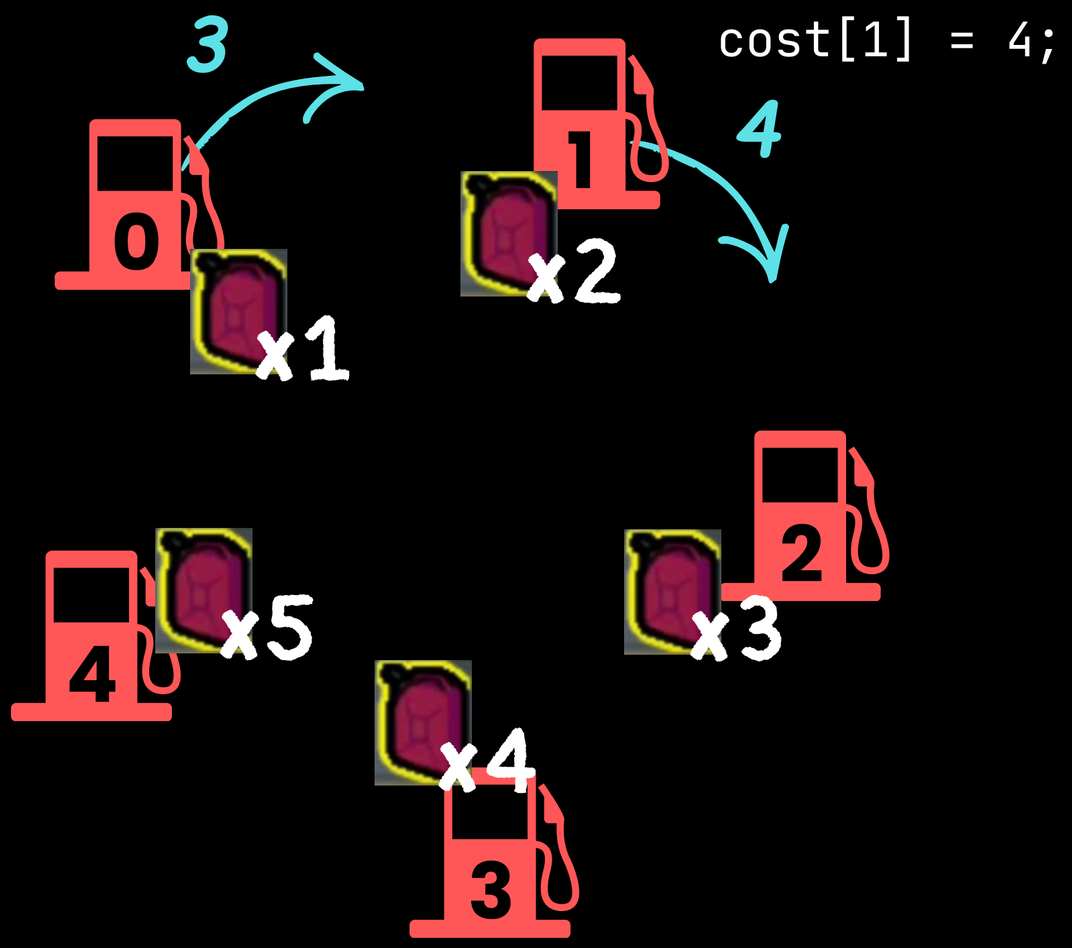
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

The Problem—

A Solved Test Case

```
cost = [3,4,5,1,2];
```



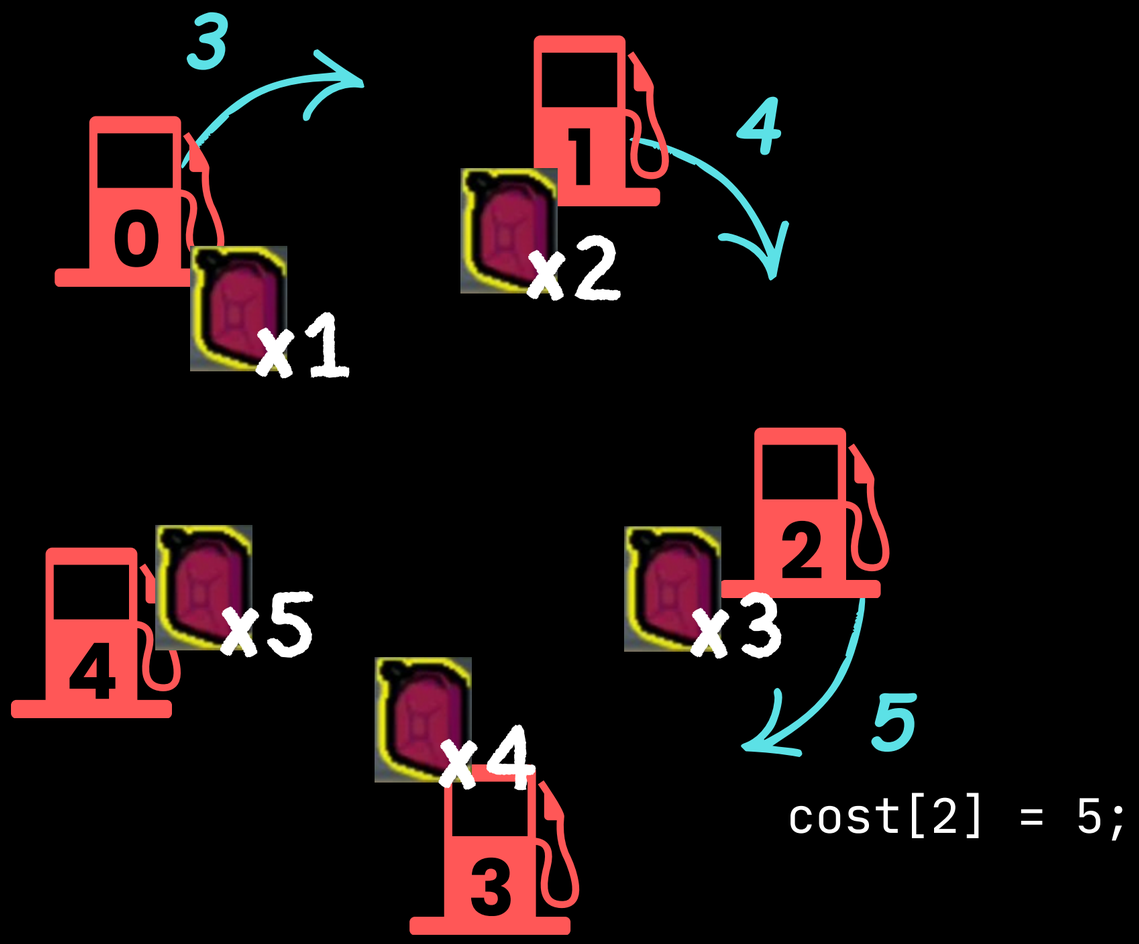
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

```
cost = [3,4,5,1,2];
```

The Problem—

A Solved Test Case



cost[2] = 5;

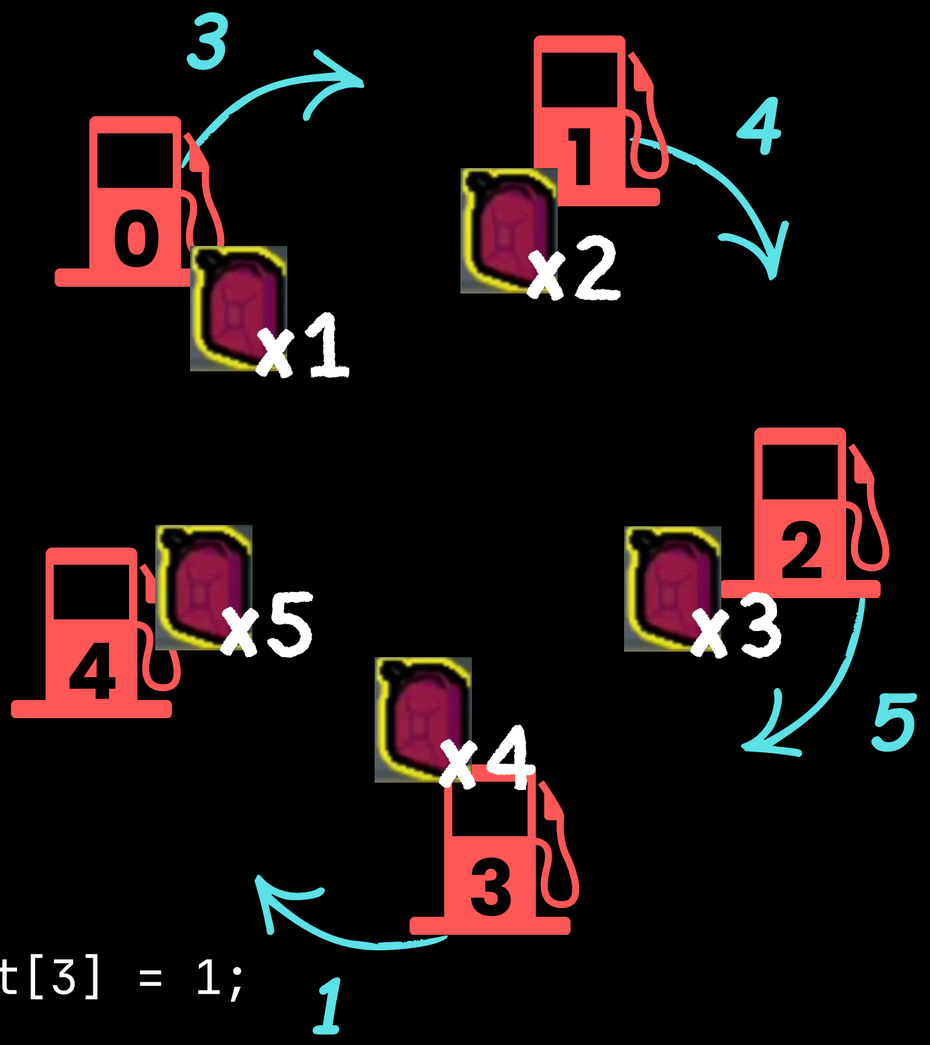
Example 1:

> **Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]
> **Output:** 3
> **Explanation:**
> Start at station 3 (index 3) and fill up with 4 unit

The Problem—

A Solved Test Case

```
cost = [3,4,5,1,2];
```




```
cost[3] = 1;
```

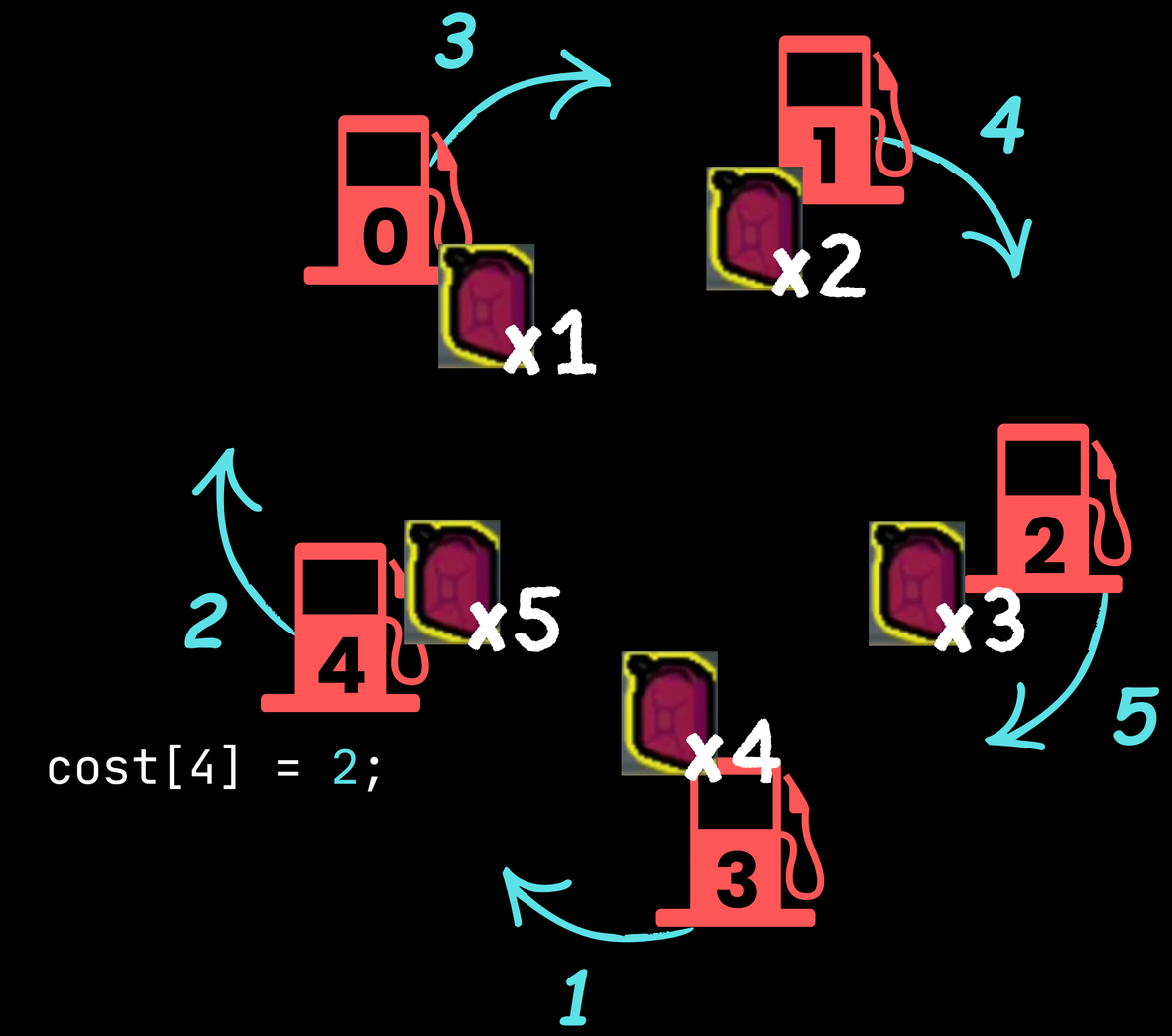
Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

```
gas  = [1,2,3,4,5];
cost = [3,4,5,1,2];
```

The Problem—

A Solved Test Case



Done!

And from here, you can see that the solution is starting at index 3.

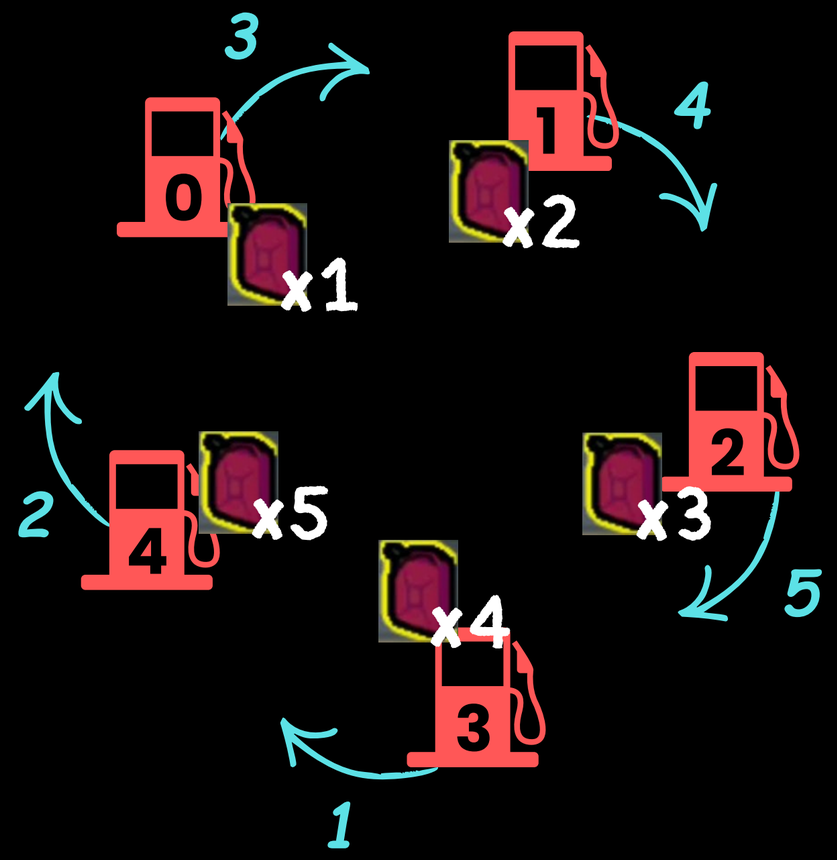
From index three, you gain 3 gallons, then gain 3 again, then lose 2, lose 3 and lose 2. **Since there is only one unique solution**, you return 3.

Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

Now you try it!!!

```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```



The Problem—

Solution

Now we have everything.
But it looks like we can simplify something!

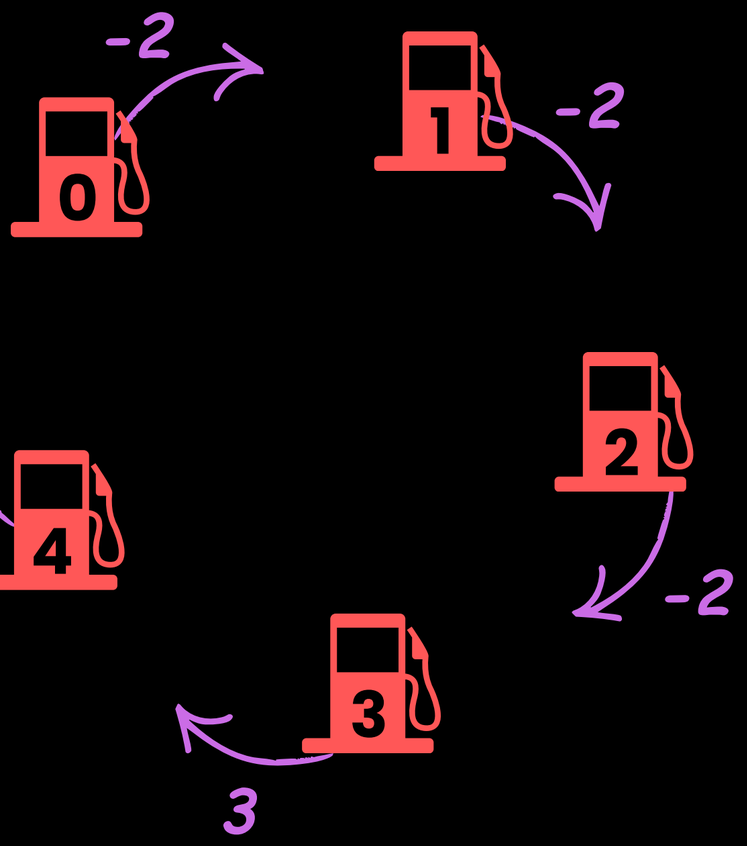
We can combine gas and cost into one array,
the difference of the two: delta.

Why does this work? Because all that matters
is the difference of gas gained and used
for this problem.

Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit
```

```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```



Now we have everything.
But it looks like we can simplify something!

We can combine gas and cost into one array,
the difference of the two: delta.

Why does this work? Because all that matters
is the difference of gas gained and used
for this problem.

Example 1:

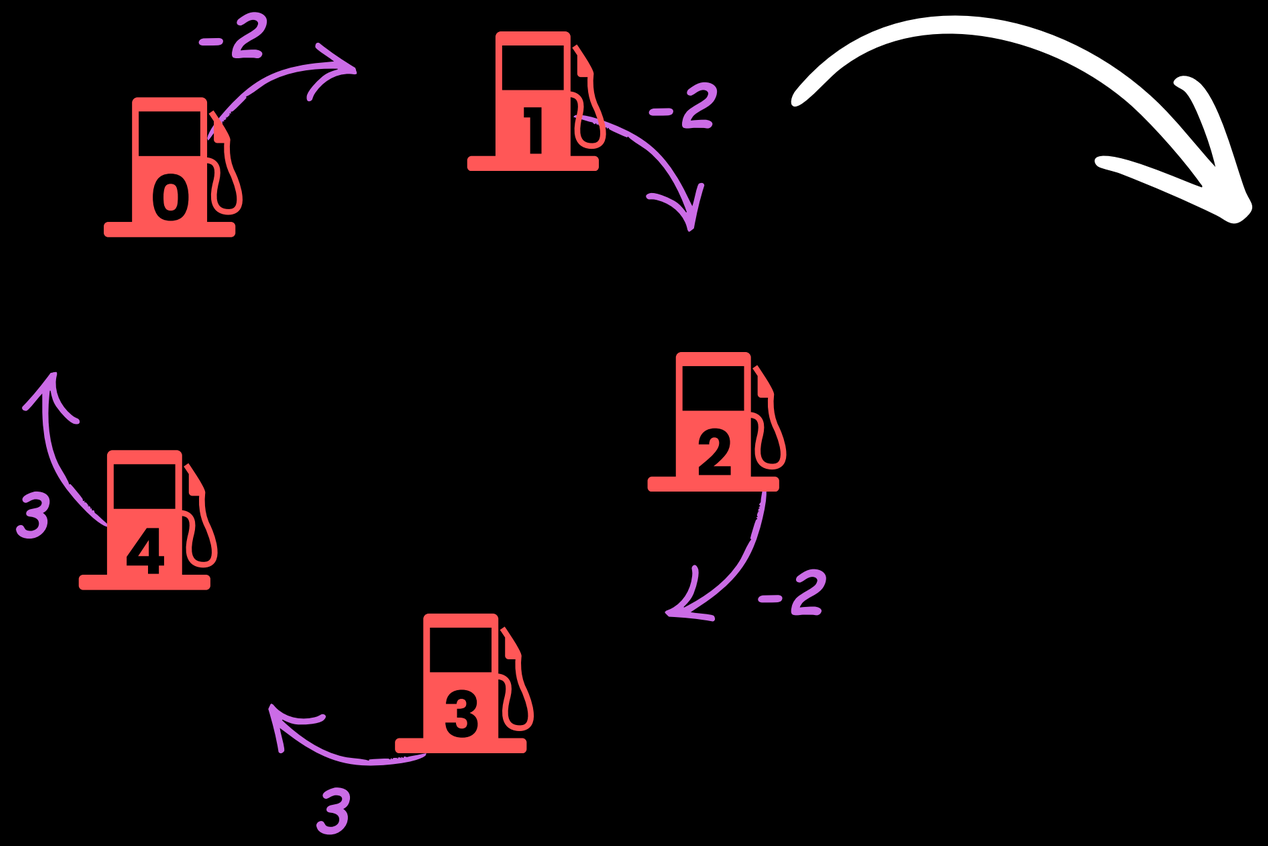
> **Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]
> **Output:** 3
> **Explanation:**
> Start at station 3 (index 3) and fill up with 4 unit

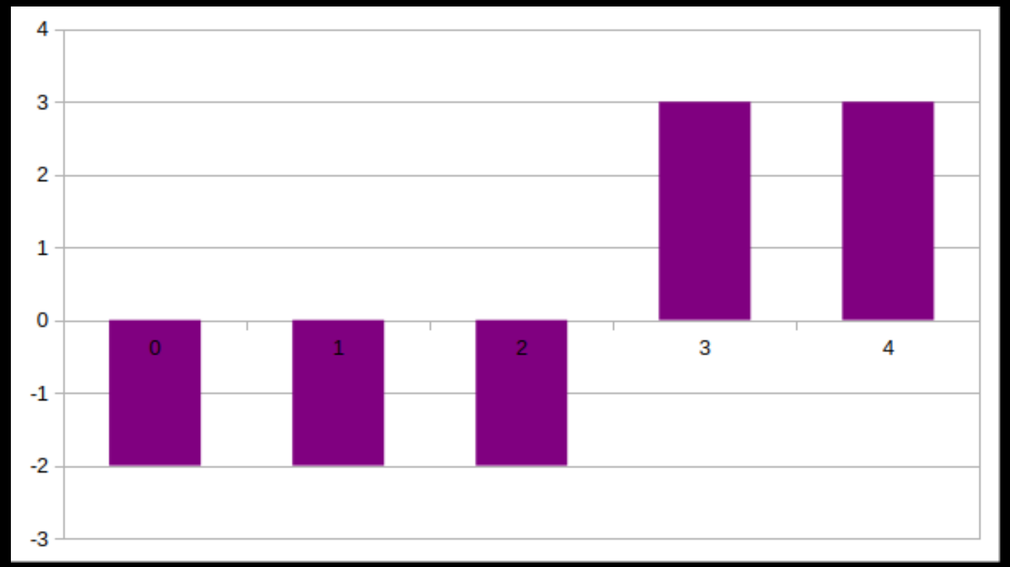
```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```

The Problem—

Solution

In fact, we can convert this visual to something else now

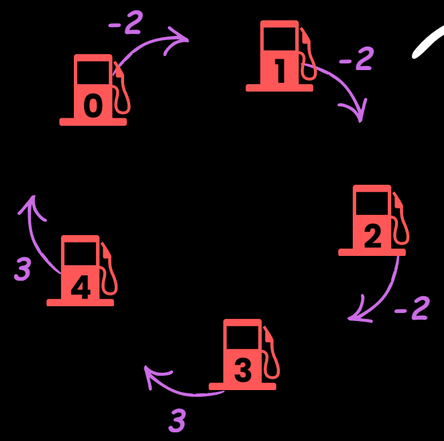


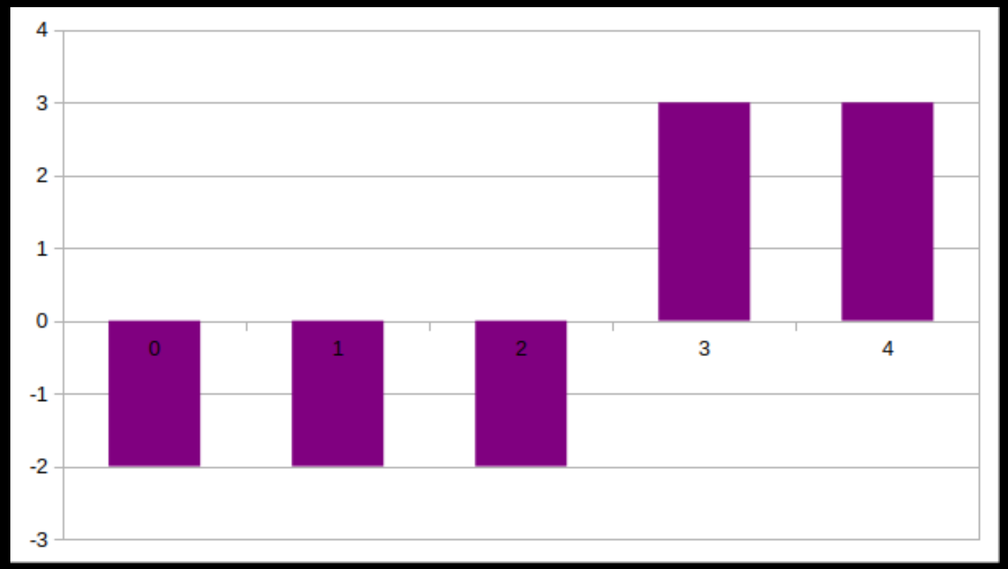


```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```

Now, it is a lot easier to see all of the deltas in a 2D space!

From this we can hint at the solution.

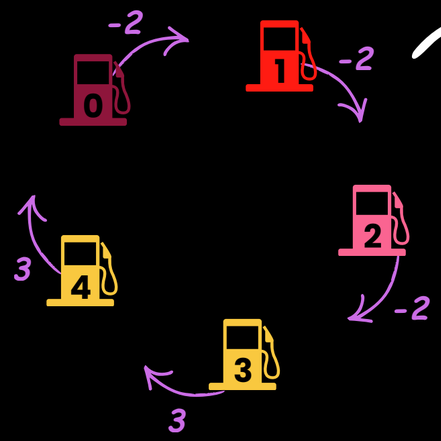




```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```

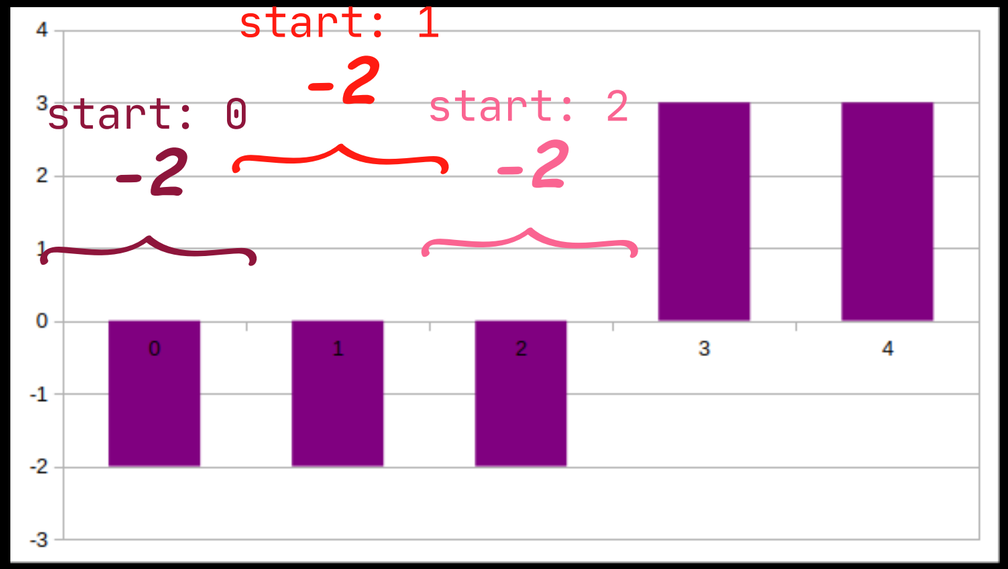
We can group these into “sections”

I’m keeping track of the start position and the sum of the deltas in each section.



start: 3

6

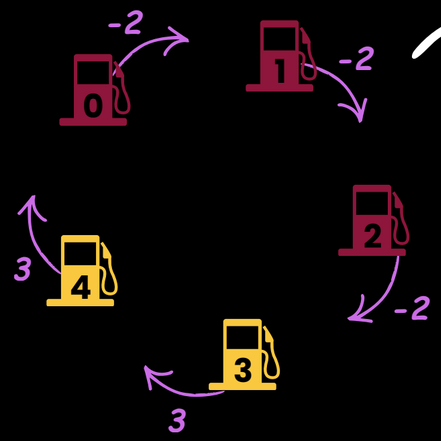


I'll get into how I choose these "sections" in a bit!

```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```

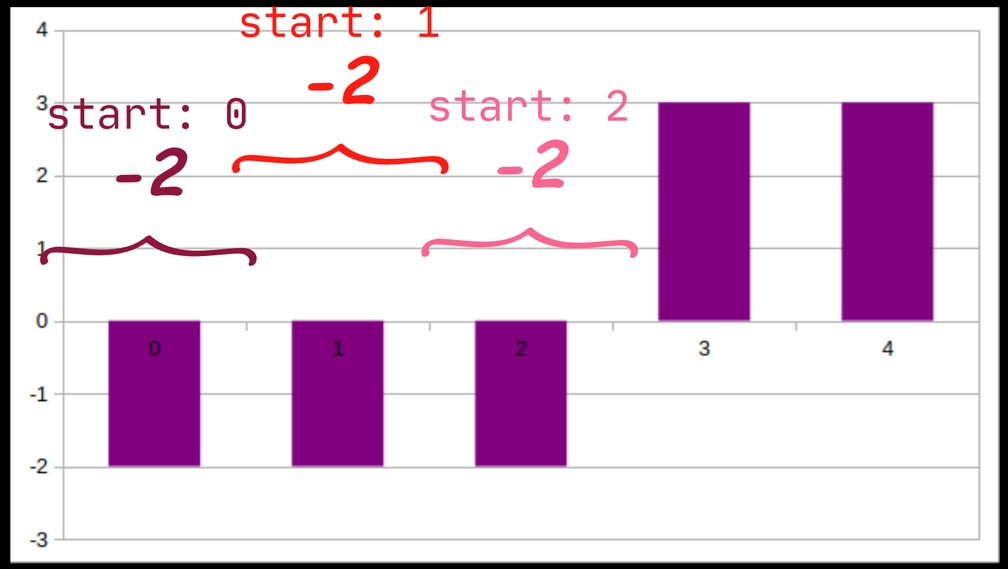
Firstly, I want to point out that the sum of all of these sections is >= 0.

What does this mean? This means we have a **valid solution!**



start: 3

6

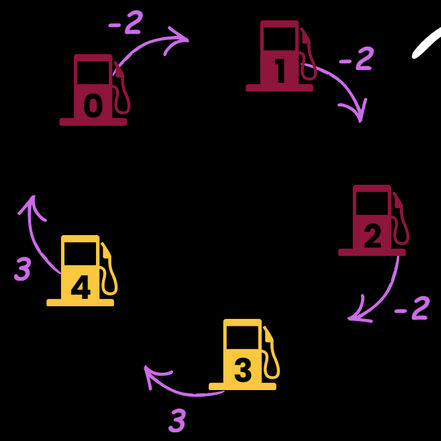


I'll get into how I choose these "sections" in a bit!

```
gas   = [1, 2, 3, 4, 5];
cost  = [3, 4, 5, 1, 2];
delta=[-2,-2,-2, 3, 3];
```

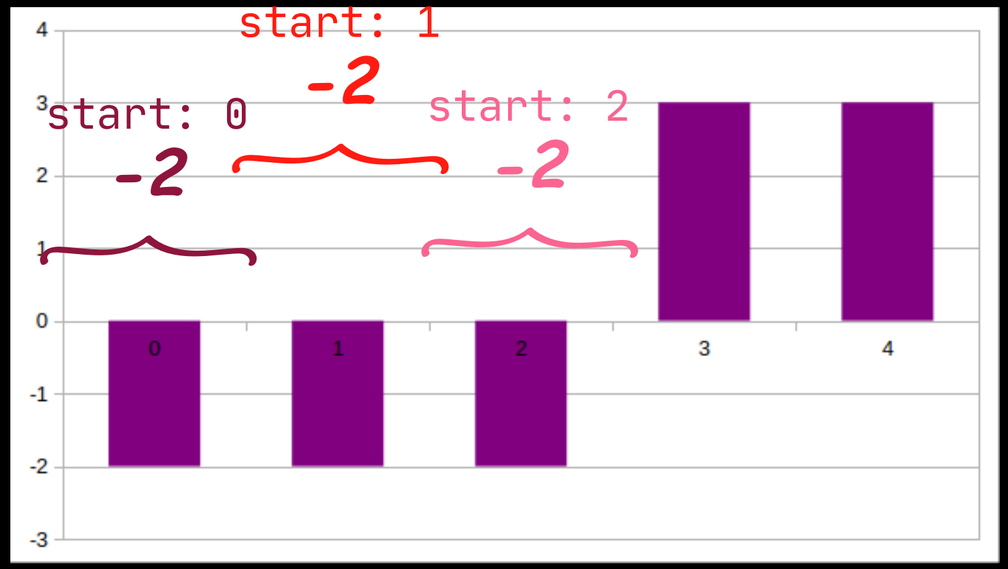
Here, we can also clearly see that we have only one option, which is the **Gold** section.

Since our **Gold** section has a start position of 3, **we return 3**.



start: 3

6



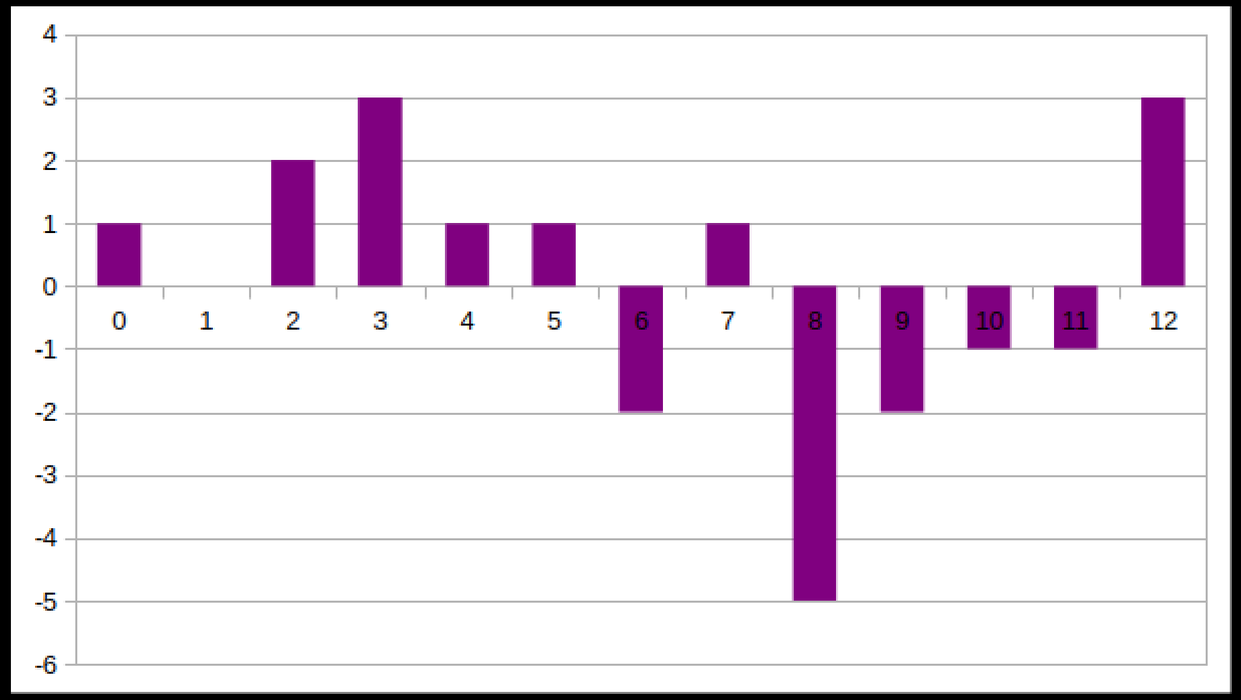
I'll get into how I choose these "sections" in a bit!

```
gas   = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost  = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [0, 1, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

Let’s look at another testcase.

I’ve already computed the delta array and made the bar graph.

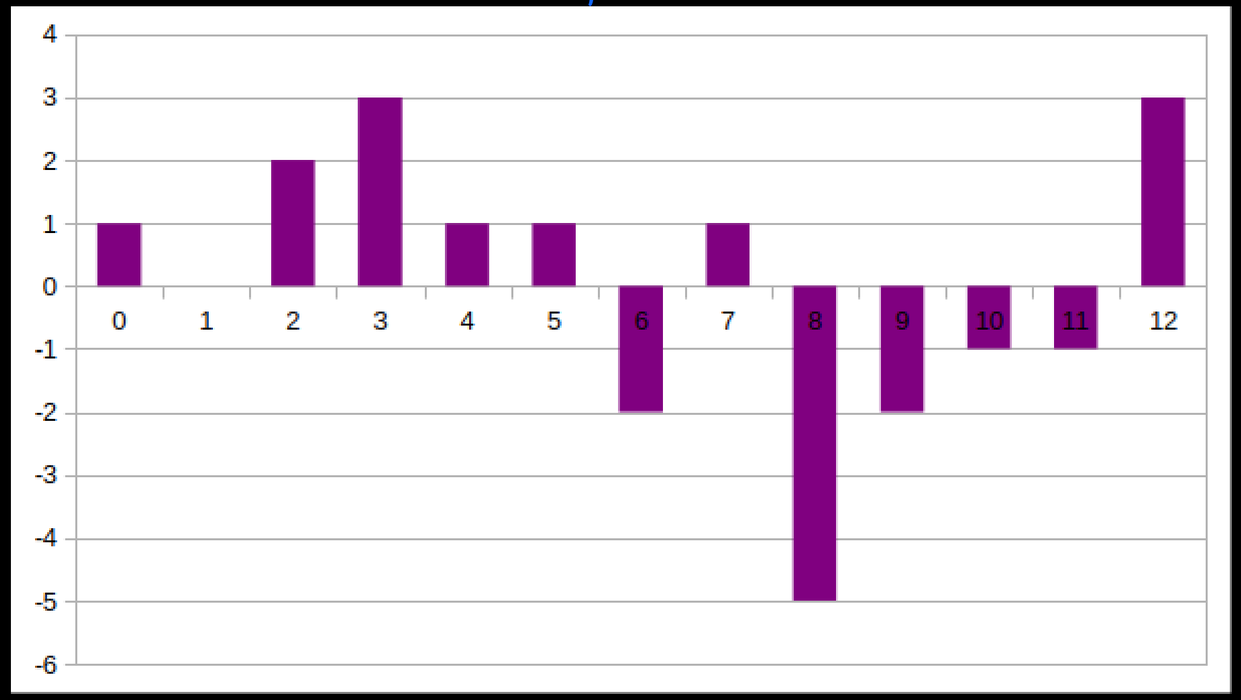
But let’s make the sections!



```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

8



But let's make the sections!

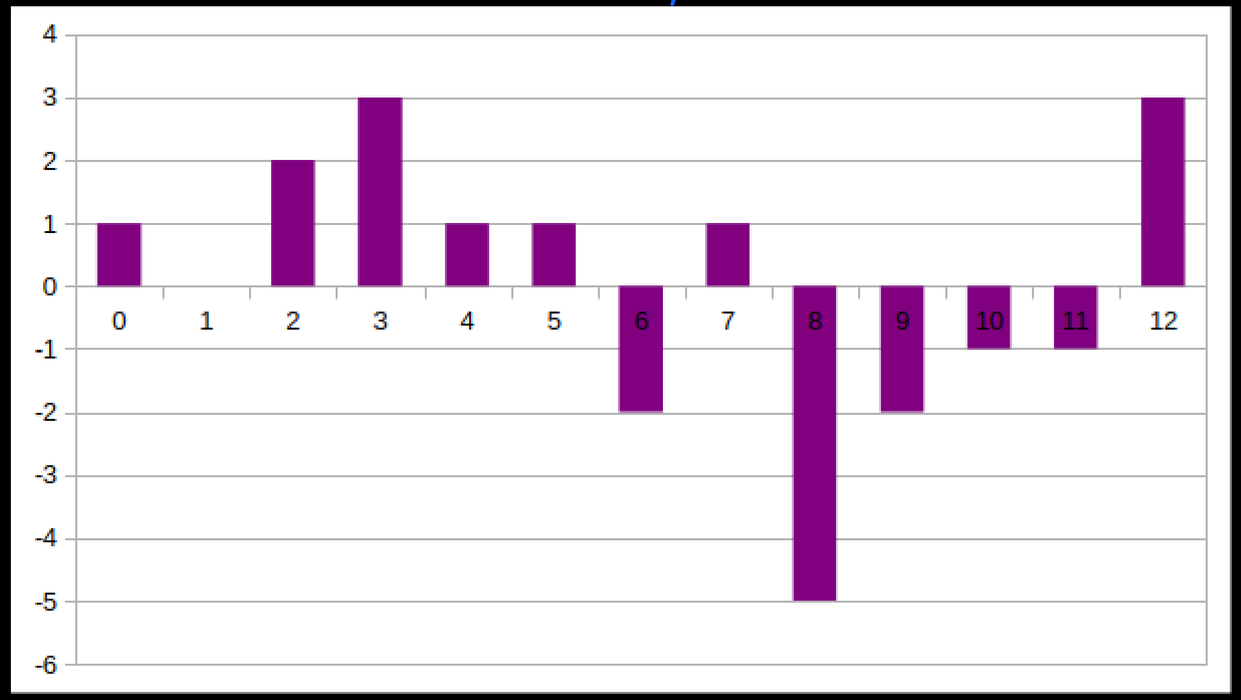
The sections are all calculated from index 0, and is the sum of all of the indexes **from left to right** until the sum is negative.

Here, the Blue section is currently 8 because (1+0+2+3+1+1) = 8.

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

6



But let's make the sections!

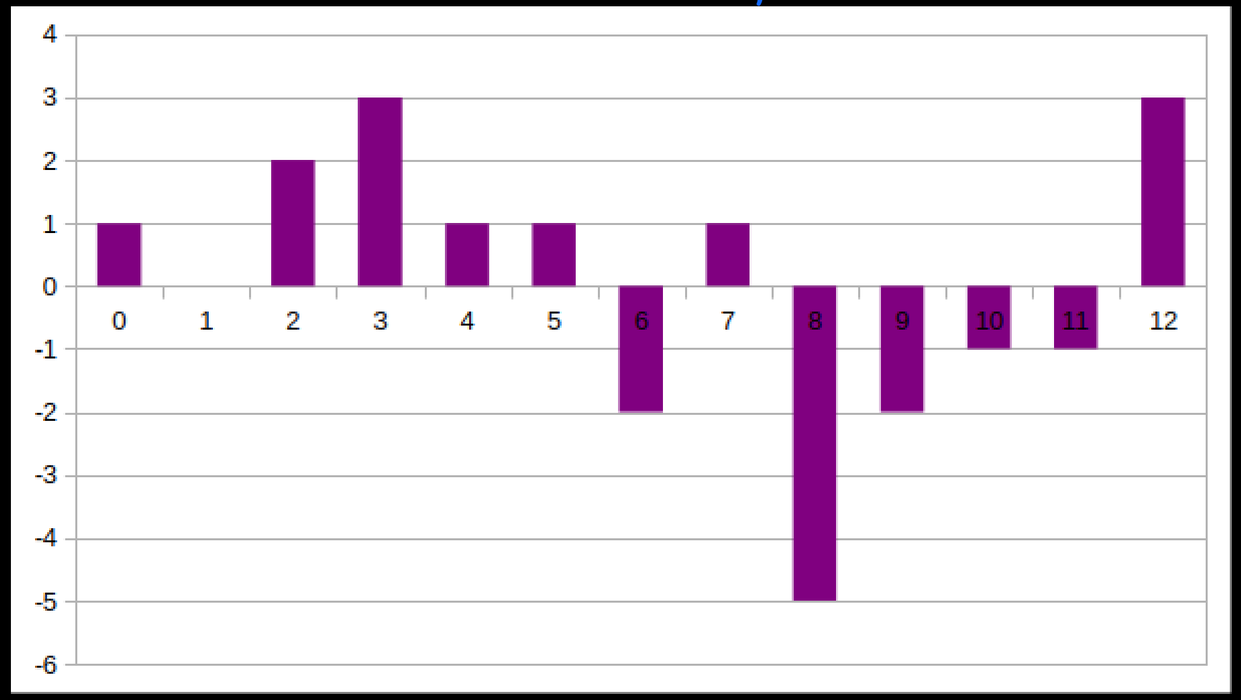
The sections are all calculated from index 0, and is the sum of all of the indexes **from left to right** until the sum is negative.

The next is -2, so the Blue section is currently 6 because (1+0+2+3+1+1-2) = 6.

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

7



But let's make the sections!

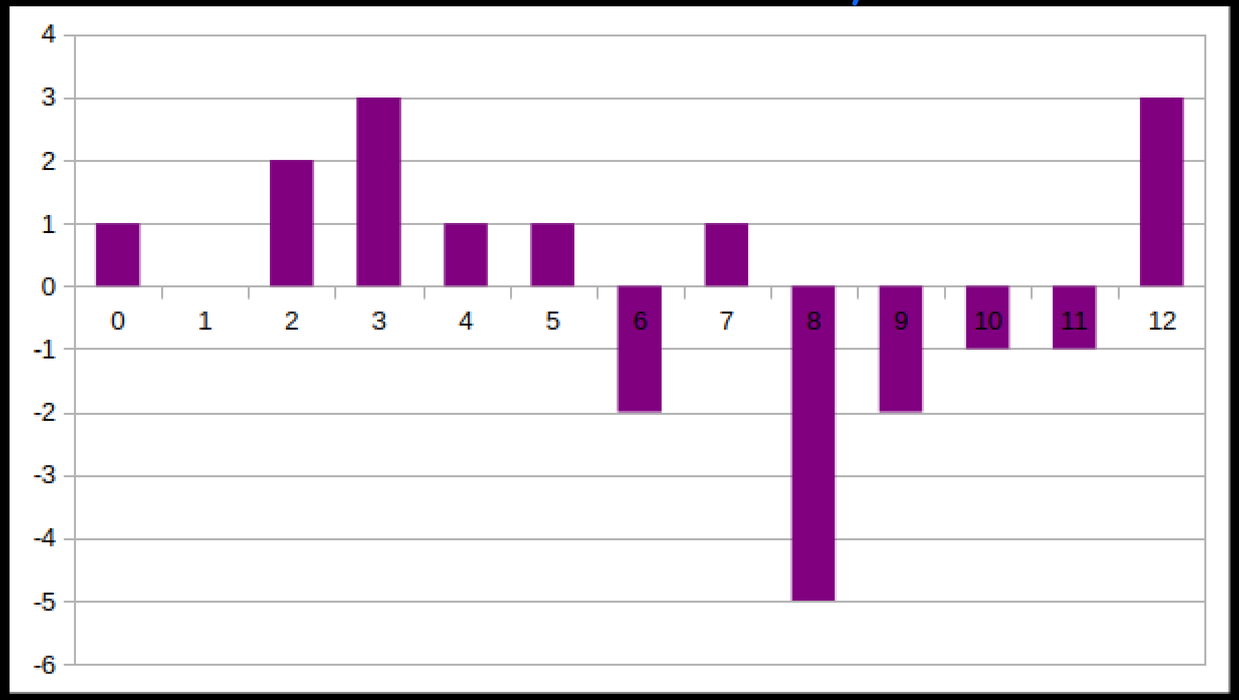
The sections are all calculated from index 0, and is the sum of all of the indexes **from left to right** until the sum is negative.

The next is 1, so the Blue section is currently 7 because (0+1+2+3+1+1-2+1) = 7.

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

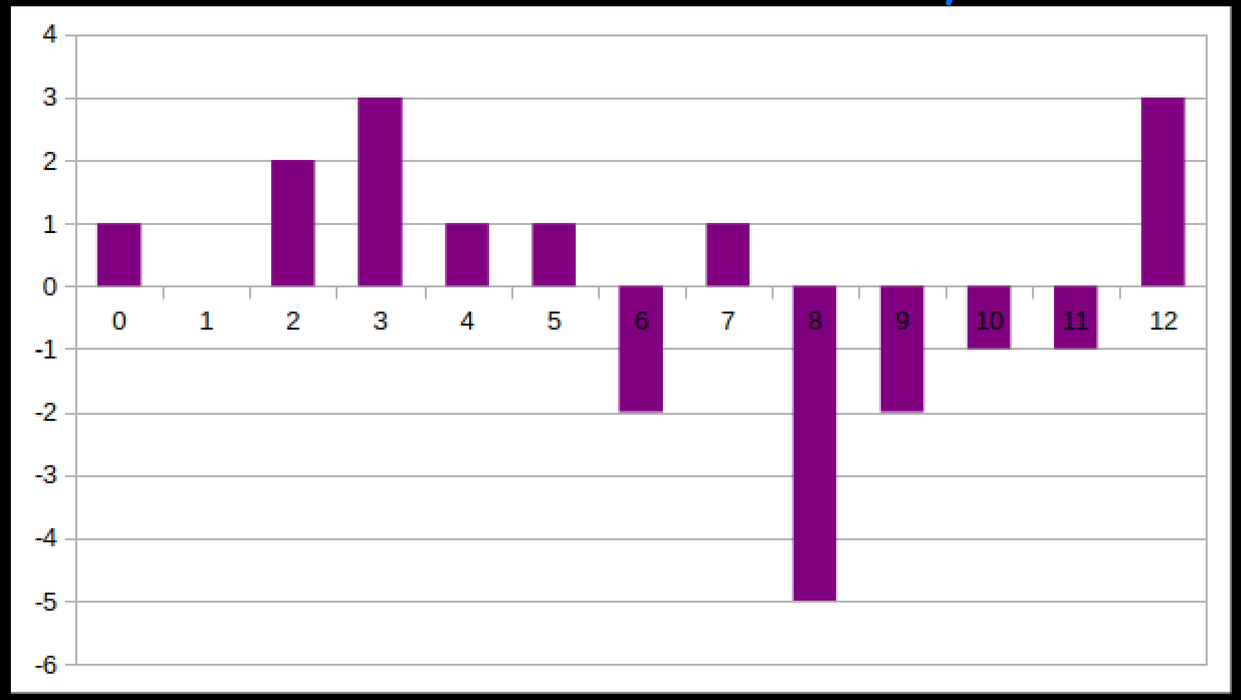
2



```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

0

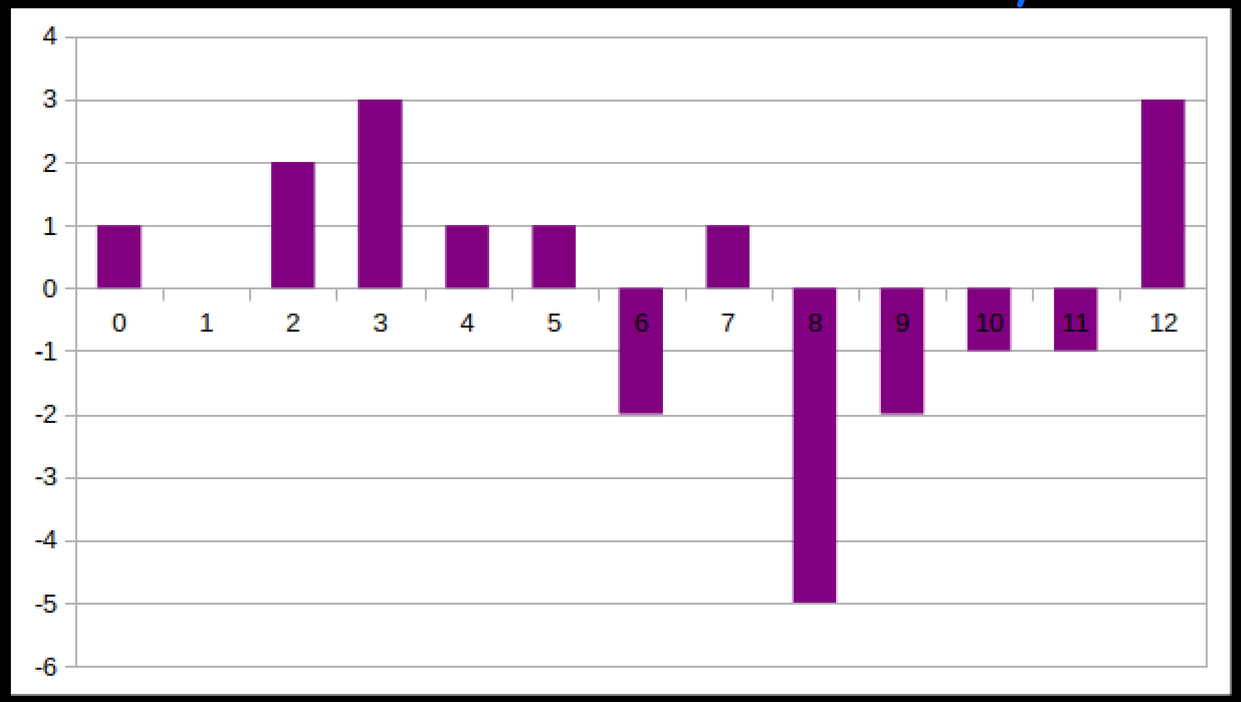


Even though the Blue section is currently 0, it is not negative, so we don’t stop (The car is running on fumes)

```
gas   = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost  = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

-1



Now, the Blue section is negative!

What does that represent? That means if we started our car at gas station 0, **then we wouldn't have enough gas to make the loop.**

Notice how we can ignore every starting position inside the Blue section, because the sum is always going to be less than (or same) if you started at index: 0.

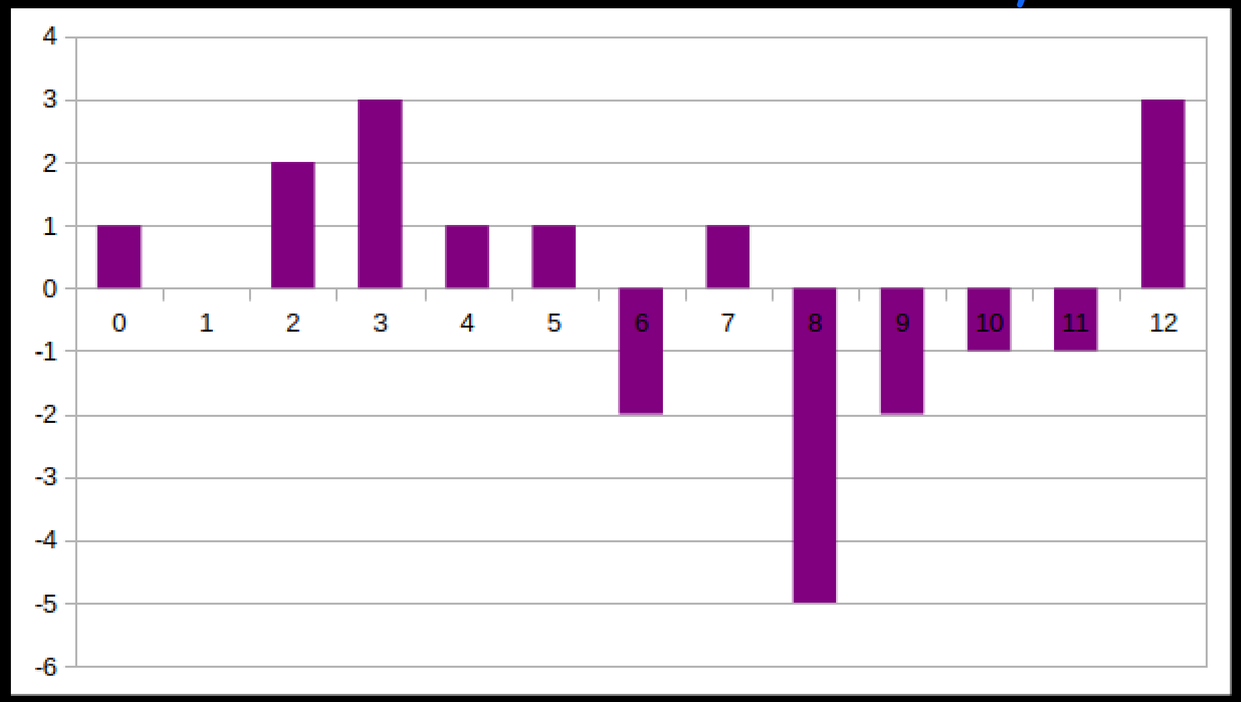
```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start: 0

-1

start:11

-1



Let’s continue!

The Green section reached -1, so index 11 can’t be the index we start with.
Let’s begin another one.

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start:11

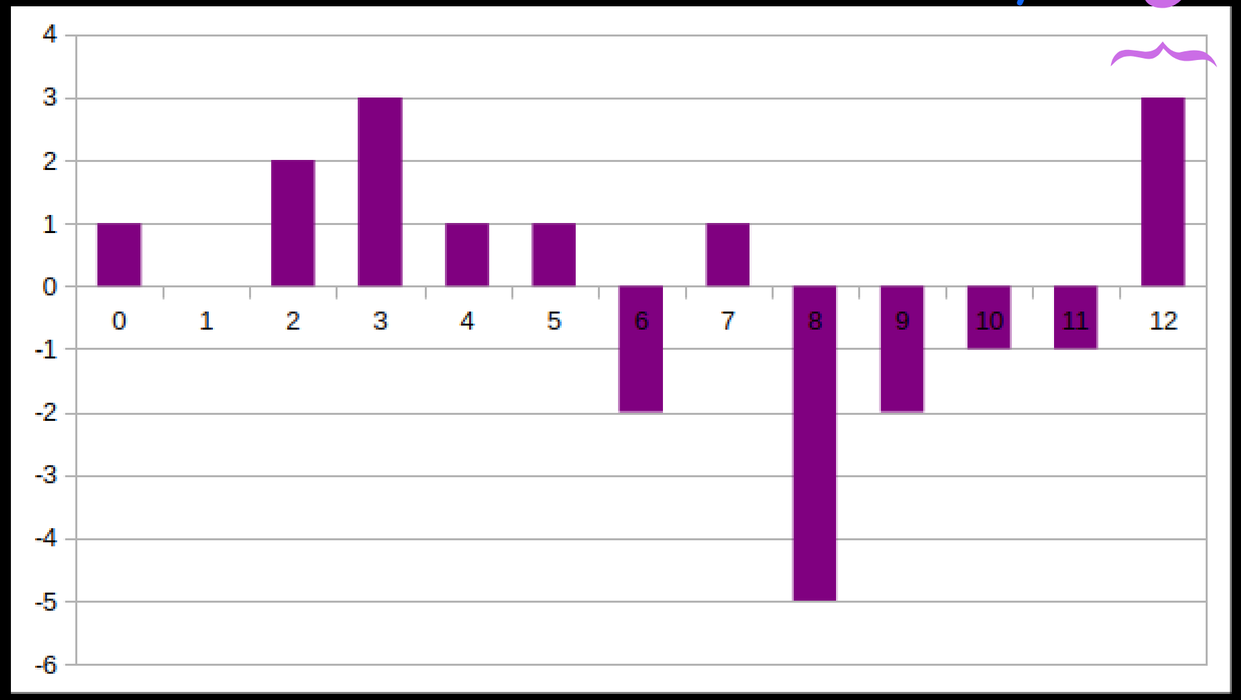
-1

start: 0

-1

start:12

3



Let’s continue!

The Purple section is the only one left, with three.

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start:11

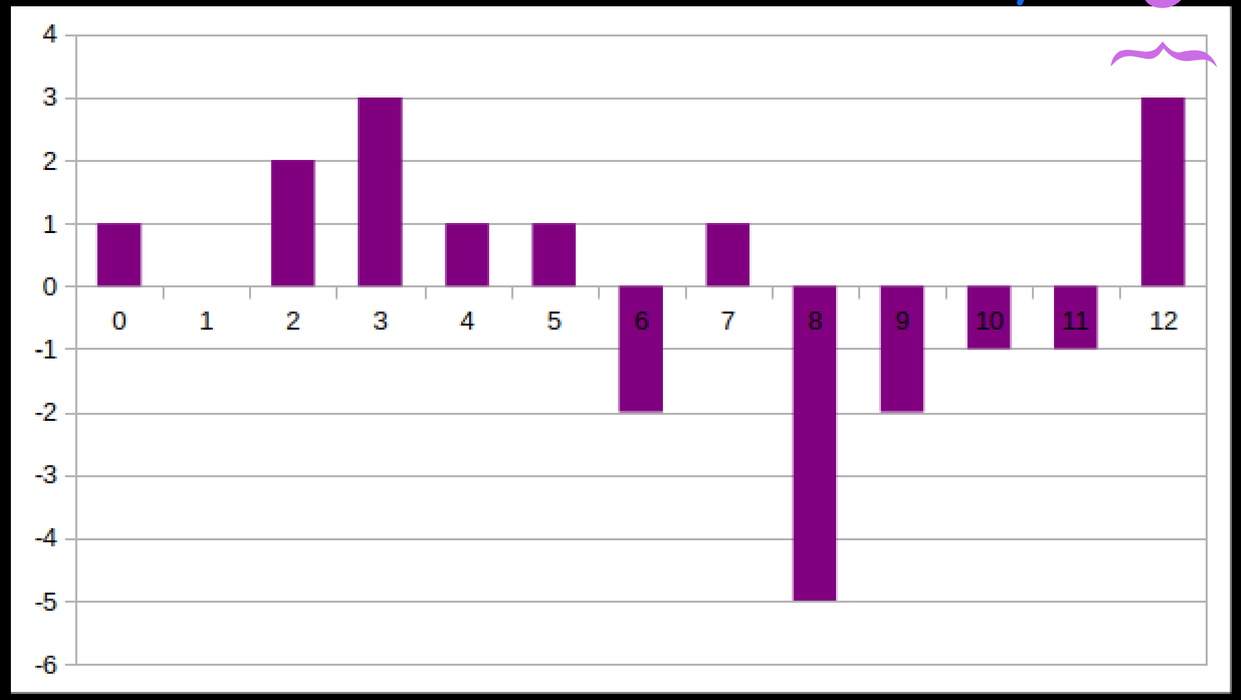
-1

start: 0

-1

start:12

3



What do we notice here?

Well, we have a valid solution for sure, since 3-1-1 = 1 and 1 is greater than 0.

But also look, the only solution is the Purple section, with start = 12. **Thus, we return 12.**

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start:11

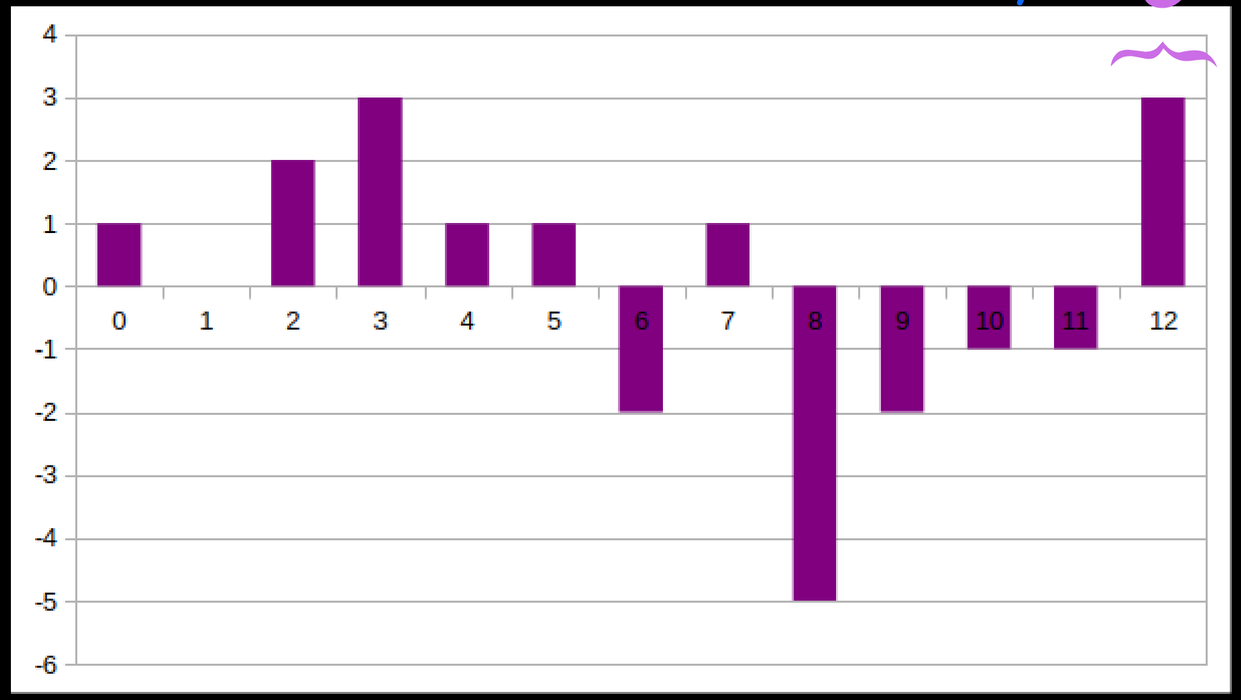
-1

start: 0

-1

start:12

3



How do we implement this?
Using a single for loop! And, we can see that this is actually an O(n) greedy solution, taking what it thinks is best at every turn.

```
gas  = [1, 2, 3, 4, 3, 2, 4, 1, 5, 3, 2, 4];
cost = [1, 1, 1, 3, 2, 4, 3, 6, 7, 4, 3, 1];
delta= [1, 0, 2, 1, 1,-1, 1,-5,-2,-1,-1, 3];
```

start:11

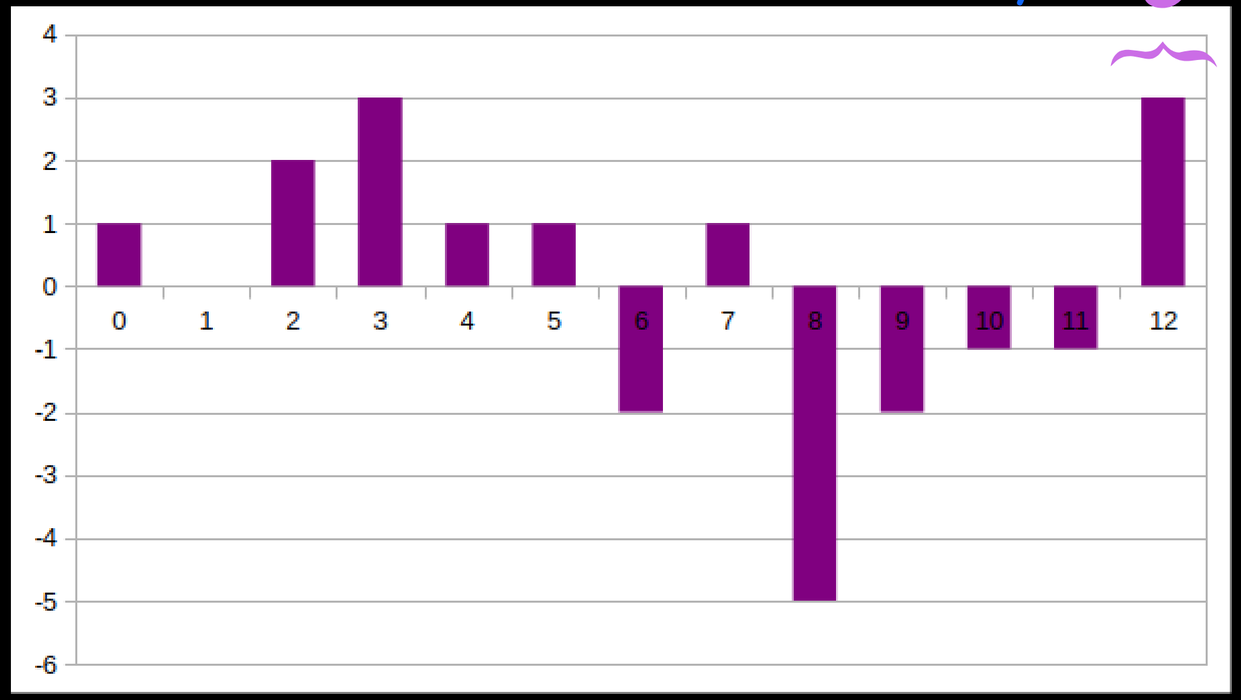
-1

start: 0

-1

start:12

3



Q&A: Don’t you need to keep track of more starting positions?

Not so fast! Because our problem states that there is **only one unique solution**, there are a lot of liberties we can take, that being we don’t keep track of more variables.

Java's O(n) Solution

Code

Java Auto

```java
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int start = 0;
        int currentGas = 0;
        int deltaGas = 0;

        for(int i = 0; i < gas.length; i++){
            currentGas += gas[i] - cost[i];
            deltaGas += gas[i] - cost[i];

            // if we run out of gas, this start point (and all points before it)
            // are invalid starting locations. move start to the next station.
            // (since we are optimistic about total gas being enough tomake it)
            if(currentGas < 0){
                start = i+1;
                currentGas = 0;
            }
        }

        // if the total gas is less than total cost, it's just impossible
        if(deltaGas < 0){
            return -1;
        }
        return start;
    }
}
```

Python's O(n) Solution

Code

Python3 Auto

```python
class Solution:
    def canCompleteCircuit(self, gas: list[int], cost: list[int]) -> int:
        start = 0
        current_gas = 0
        total_gas = 0

        for i in range(len(gas)):
            diff = gas[i] - cost[i]
            current_gas += diff
            total_gas += diff

            # if we run out of gas, this start point (and all points before it)
            # are invalid starting locations. move start to the next station.
            # (since we are optimistic about total gas being enough tomake it)
            if current_gas < 0:
                start = i + 1
                current_gas = 0

        # if the total gas is less than total cost, it's just impossible
        return start if total_gas >= 0 else -1

```

Brief syntax lesson!

We should all be familiar with the idea of sorting an array

c++

```cpp
int main() {
    std::vector<int> nums = {42, 7, 13, 1, 99};

    // Sorts from the beginning iterator to the end iterator
    std::sort(nums.begin(), nums.end());
```

Java

```java
public class Main {
    public static void main(String[] args) {
        int[] nums = {42, 7, 13, 1, 99};

        // Modifies the array in-place
        Arrays.sort(nums);
```

Python

```python
nums = [42, 7, 13, 1, 99]

# Option 1: In-place (modifies nums)
nums.sort()
```

Brief syntax lesson!

We can even do it in reverse (decreasing)

c++

```cpp
int main() {
    std::vector<int> nums = {42, 7, 13, 1, 99};

    // Sort using the built-in "greater" comparator
    std::sort(nums.begin(), nums.end(), std::greater<int>());
```

Java

```java
public class Main {
    public static void main(String[] args) {
        // Must use Integer[] (objects) instead of int[] (primitives)
        Integer[] nums = {42, 7, 13, 1, 99};

        Arrays.sort(nums, Collections.reverseOrder());
```

Python

```python
nums = [42, 7, 13, 1, 99]

# Simply set reverse to True
nums.sort(reverse=True)
```

Brief syntax lesson!

But... these only work for integers!

What if we want to sort:

- Arrays, by their third elements
- Sets, by whether they contain the element 7
- Linked Lists by their size
- Arrays of Arrays by the sum of all elements in each array of arrays

Unfortunately, our programming languages do not provide easy ways to do this :(

Brief syntax lesson!

Fortunately, we can create custom sorting instructions!

These are called custom comparators, they are functions that we provide to the programming language that tell it how to sort

These functions tell us for any two inputs (a and b), which one do we put first?

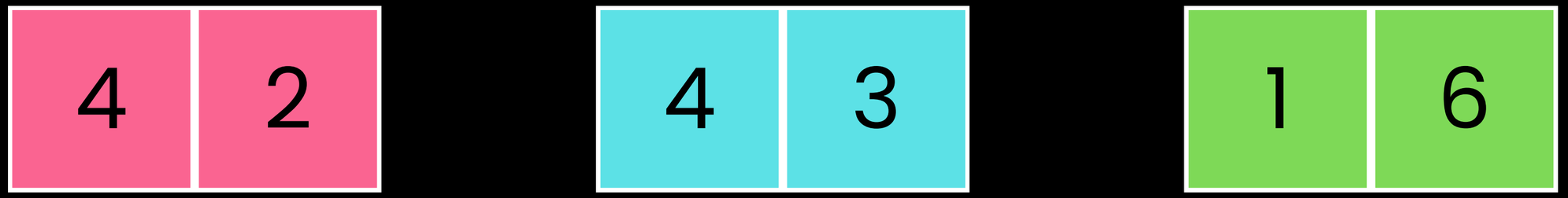
Brief syntax lesson!

They work slightly different in each language, so we'll go one-by-one

But first! Let's get an example problem

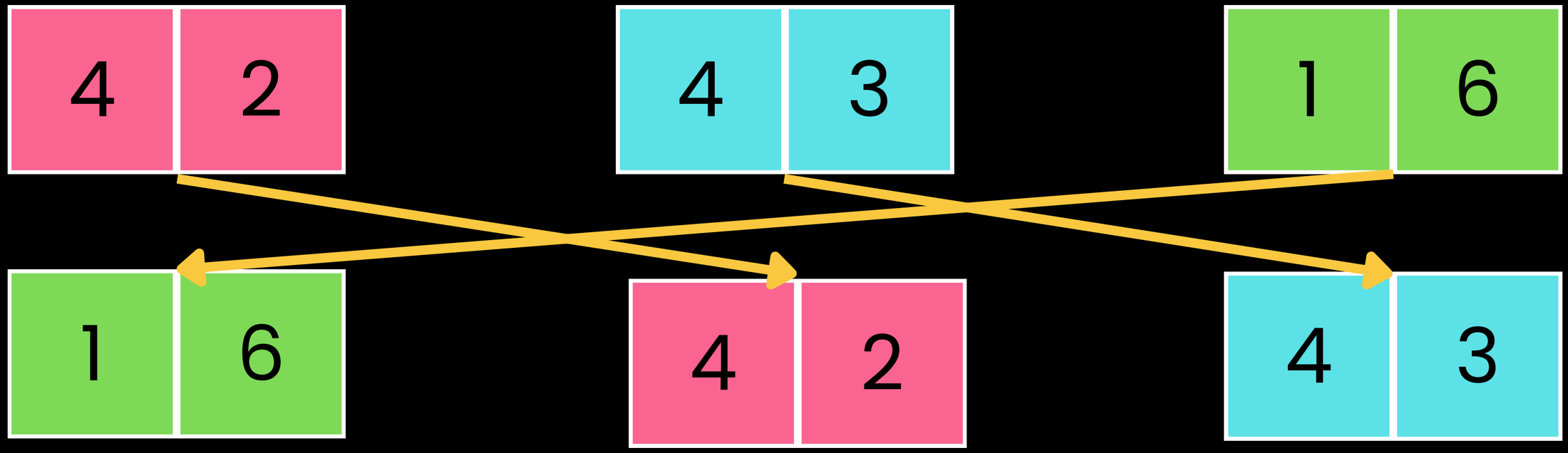
Brief syntax lesson!

Let’s say want to sort a list of lists, where each internal list has size 2



Specifically, let’s say we want to sort them first by increasing first elements, and (incase of a tie), then by increasing second elements

Brief syntax lesson!



Let’s say we want to sort them first by increasing first elements, and (incase of a tie), then by increasing second elements

Java

```
public static int compareItems(int[] a, int[] b) {
```

Function Description

- Inputs
  - a/b: Arrays of size 2
- Outputs
  - Output -1: a comes before b
  - Output 0: a and b are equivalent
  - Output 1: b comes before a

Java

Function Description

- Inputs
  - a/b: Arrays of size 2
- Outputs
  - Output -1: a comes before b
  - Output 0: a and b are equivalent
  - Output 1: b comes before a

```java
public static int compareItems(int[] a, int[] b) {
    if (a[0] != b[0]) {
        // Primary: First element increasing
        if (a[0] < b[0]) {
            return -1; // a comes before b
        } else {
            return 1;  // a comes after b
        }
    }

    // Secondary: Second element increasing (Tie-breaker)
    if (a[1] != b[1]) {
        if (a[1] < b[1]) {
            return -1;
        } else {
            return 1;
        }
    }
```

Python

```
def compare_items(a, b):
```

Function Description

- Inputs
  - a/b: Arrays of size 2
- Outputs
  - Output -1: a comes before b
  - Output 0: a and b are equivalent
  - Output 1: b comes before a

Python

Function Description

- Inputs
  - a/b: Arrays of size 2
- Outputs
  - Output -1: a comes before b
  - Output 0: a and b are equivalent
  - Output 1: b comes before a

```python
def compare_items(a, b):
    # Primary: First element increasing
    if a[0] != b[0]:
        return -1 if a[0] < b[0] else 1

    # Secondary: Second element increasing
    if a[1] != b[1]:
        return -1 if a[1] < b[1] else 1

    return 0 # Elements are exactly equal
```

C++

```cpp
bool compareItems(const std::vector<int>& a, const std::vector<int>& b) {
```

Function Description

- Inputs
  - a/b: Lists of size 2
- Outputs
  - Output True: a comes before b
  - Output False: b comes before a OR a and b are equal

C++

Function Description

- Inputs
  - a/b: Lists of size 2
- Outputs
  - Output True: a comes before b
  - Output False: b comes before a OR a and b are equal

```cpp
bool compareItems(const std::vector<int>& a, const std::vector<int>& b) {
    if (a[0] != b[0]) {
        return a[0] < b[0]; // Primary: Increasing
    }
    return a[1] < b[1];     // Secondary (Tie-breaker): Increasing
}
```

How to use these functions?

Plug them into the sorting function as an argument!

Java

```java
public static void main(String[] args) {
    int[][] data = {{4, 2}, {4, 3}, {1, 6}};

    Arrays.sort(data, Main::compareItems);

    for (int[] row : data) {
        System.out.print(Arrays.toString(row) + " ");
    }
}
```

```
[1, 6] [4, 2] [4, 3]
```

Remember, compareItems is our comparator function

Python

```
from functools import cmp_to_key

data = [[4, 2], [4, 3], [1, 6]]

data.sort(key=cmp_to_key(compare_items))

print(data)
```

```
[1, 6] [4, 2] [4, 3]
```

Remember, compareItems is our comparator function

C++

```cpp
int main() {
    std::vector<std::vector<int>> data = {{4, 2}, {4, 3}, {1, 6}};

    std::sort(data.begin(), data.end(), compareItems);

    for (const auto& row : data) {
        std::cout << "[" << row[0] << ", " << row[1] << "] ";
    }
    return 0;
}
```

```
[1, 6] [4, 2] [4, 3]
```

Remember, compareItems is our comparator function

Now, using both Greedy and Sorting

435. Non-overlapping Intervals

Medium | Topics | Companies

Given an array of intervals `intervals` where `intervals[i] = [start_i, end_i]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

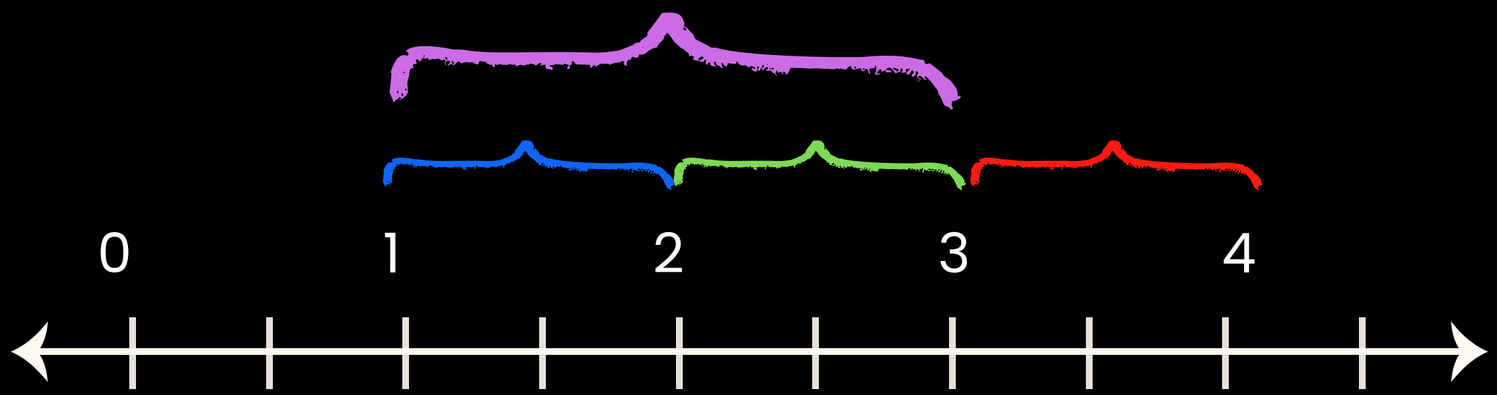
Note that intervals which only touch at a point are **non-overlapping**. For example, `[1, 2]` and `[2, 3]` are non-overlapping.

Sample Case

First, lets plot all of our intervals on a number line

Example 1:

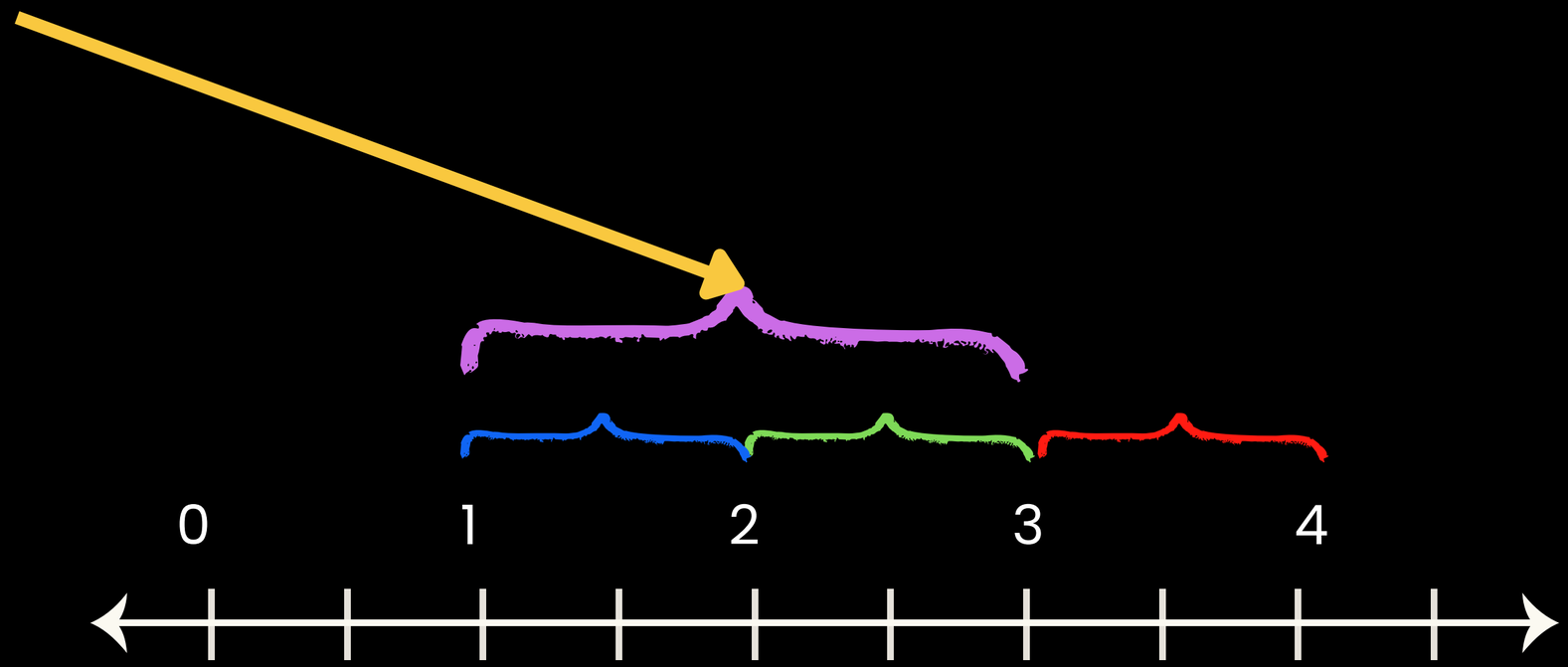
Input: intervals = [[1,2] [2,3] [3,4] [1,3]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.



Sample Case

How many intervals do we need to remove to make these all non-overlapping?

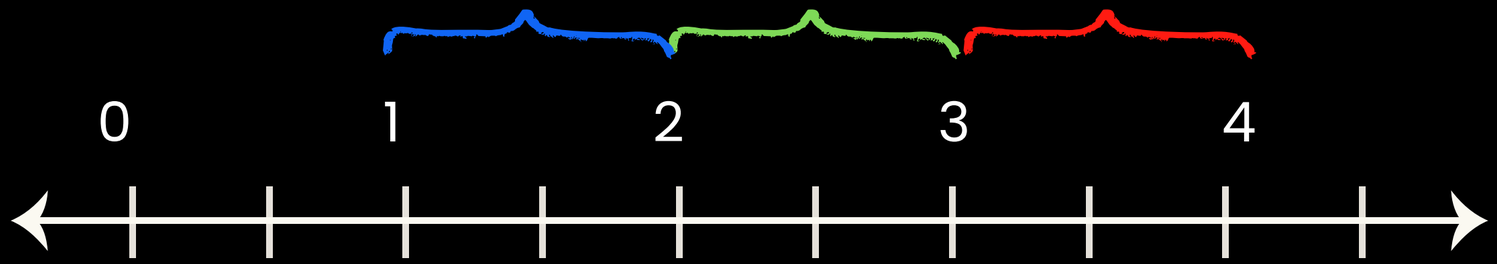
Just one!



Sample Case

After removing one interval, we have no overlap

Output: 1



Good Luck!

Hint:

You will need to sort these intervals in a specific order that lets you iterate through them in an optimal order

We encourage you to think about and test different sorting options!

Solution

Some intuition:

Figuring out the number of intervals we must remove is equal to finding the maximum number of intervals that ARE NOT overlapping, and subtracting that from the total number of intervals

#intervalsToRemove = #totalIntervals - #maximum-nonoverlapping-intervals

Solution

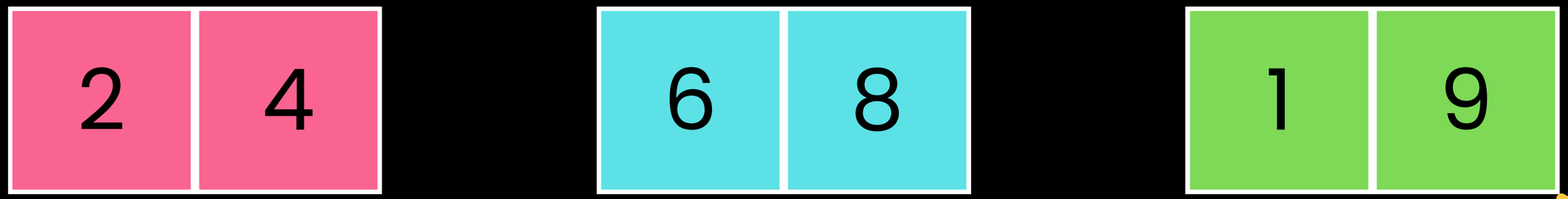
Then, we need to find the maximum number of intervals we can fit with no overlap!

As such, we need to sort our intervals in a way that orders them such that we can fit as many as possible

Sort by starting time?

If we want to cram as many intervals as possible, it makes sense to start with the intervals that start first, so what happens if we sort arrays by starting time?

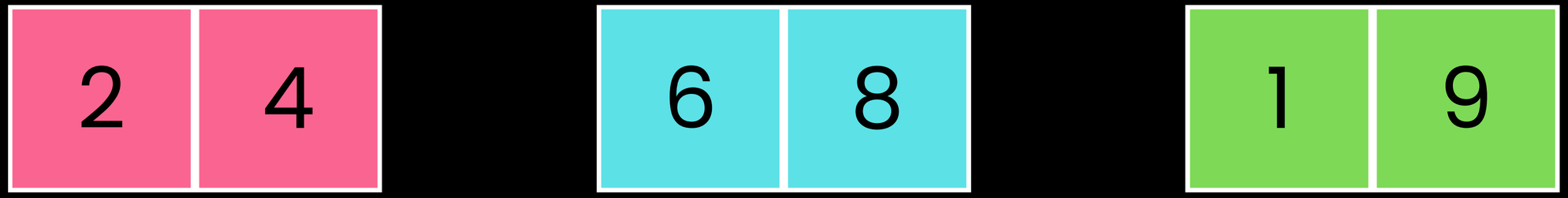
Sort by starting time?



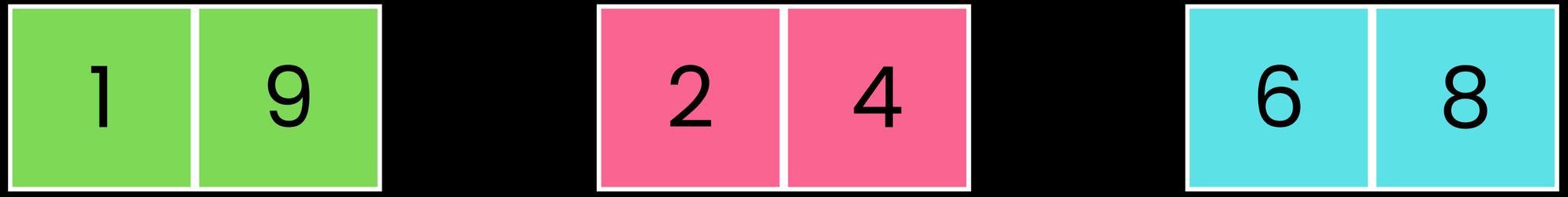
How many intervals do we need to remove here?

One!

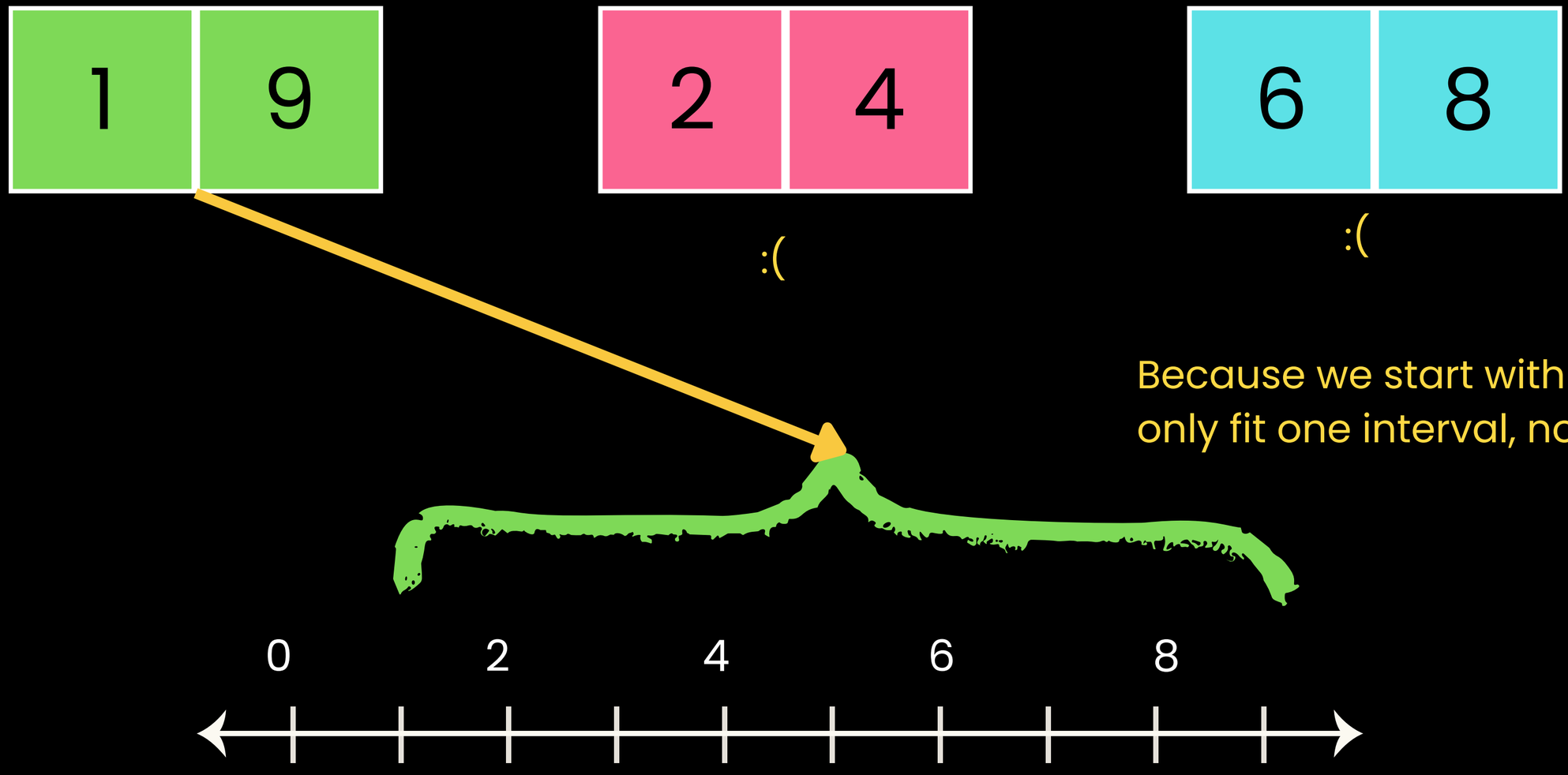
Sort by starting time?



But... If we sort by starting time, this happens:



Sort by starting time?



Alternate method?

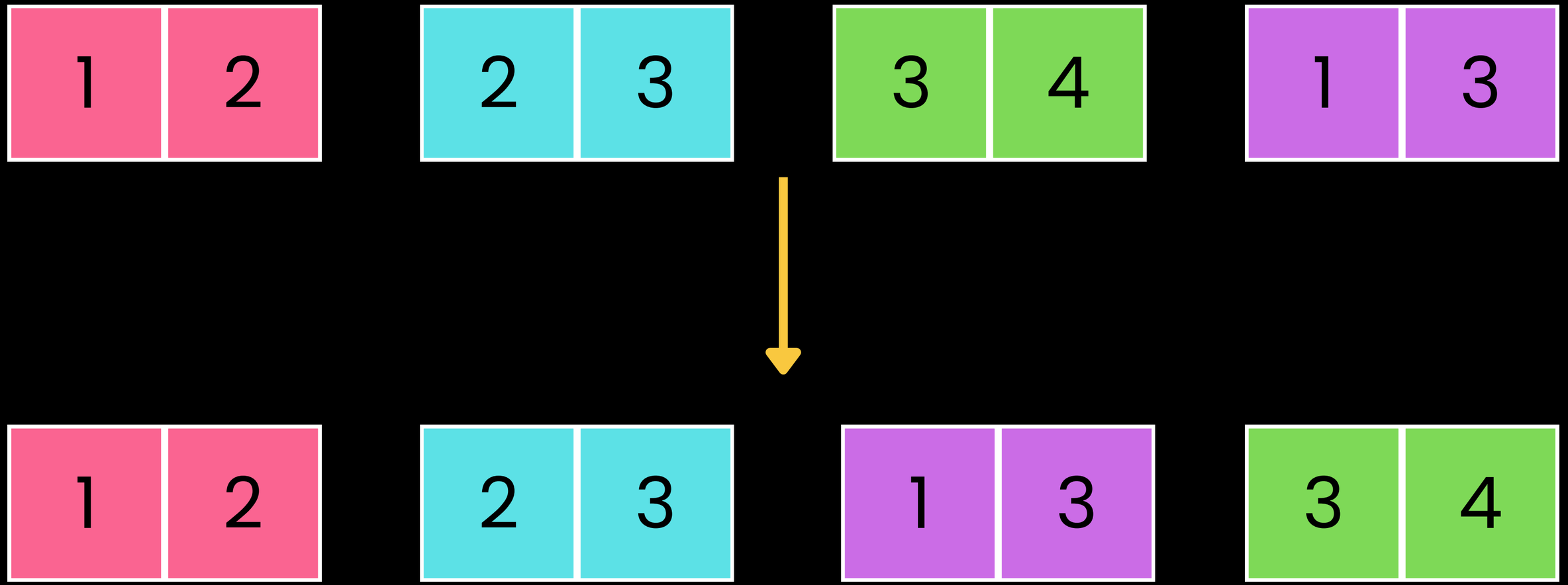
We saw that sorting by starting time doesn’t work, why?

Just because an interval starts early, it doesn’t mean that it ends early

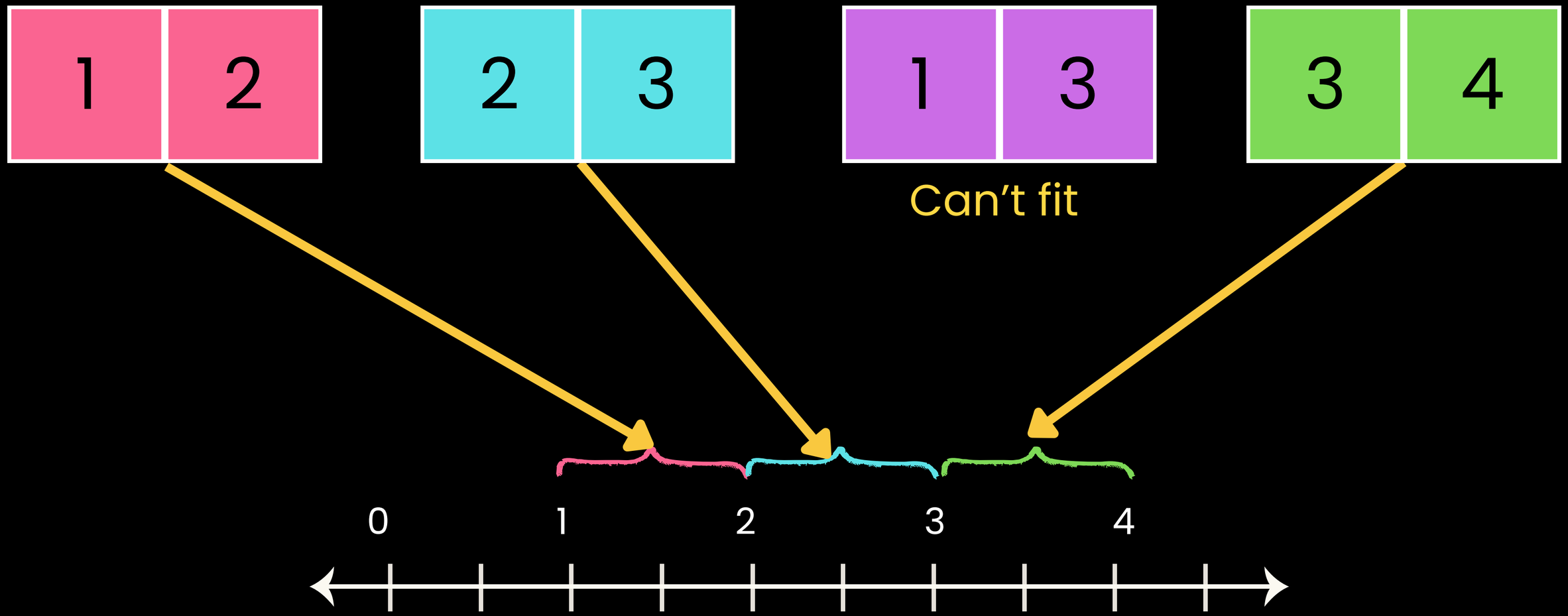
Any other ideas?

Sort by Ending Time

If we want intervals that end early, let's just sort that way!



Sort by Ending Time



Solution

Sorting by ending time lets us fit the maximum number of intervals that can fit!

Then, our solution becomes:

1. Sort intervals by end time
2. Iterate over the intervals, and track how many we can fit
    a. To test if an interval fits, we check if it's start time is later than the end time of the last interval that we fit
3. Subtract the total intervals by how many we could fit

Java

Hacker Devils

Presented by Sohum Sheth, Dean Richardson and Ahmed Shaheen

THANK YOU

for your time and attention

