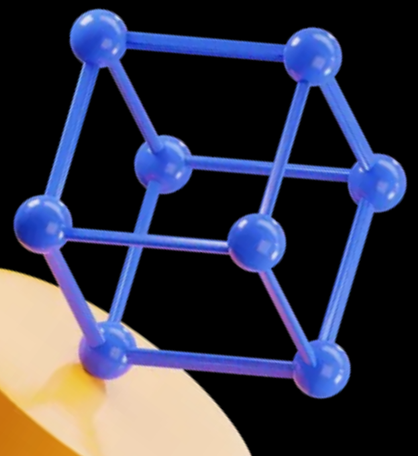
Sohum Sheth, Dean Richardson, Griffin Charlson

HKD MEDIUM SECTION #1

Greedy Algorithms

September 30th, 2026



Attendance and Announcements



- 🍕Pizza at the end as always :)

Medium Section Overview

For each meeting, we will have one high-level concept that we will try to explain

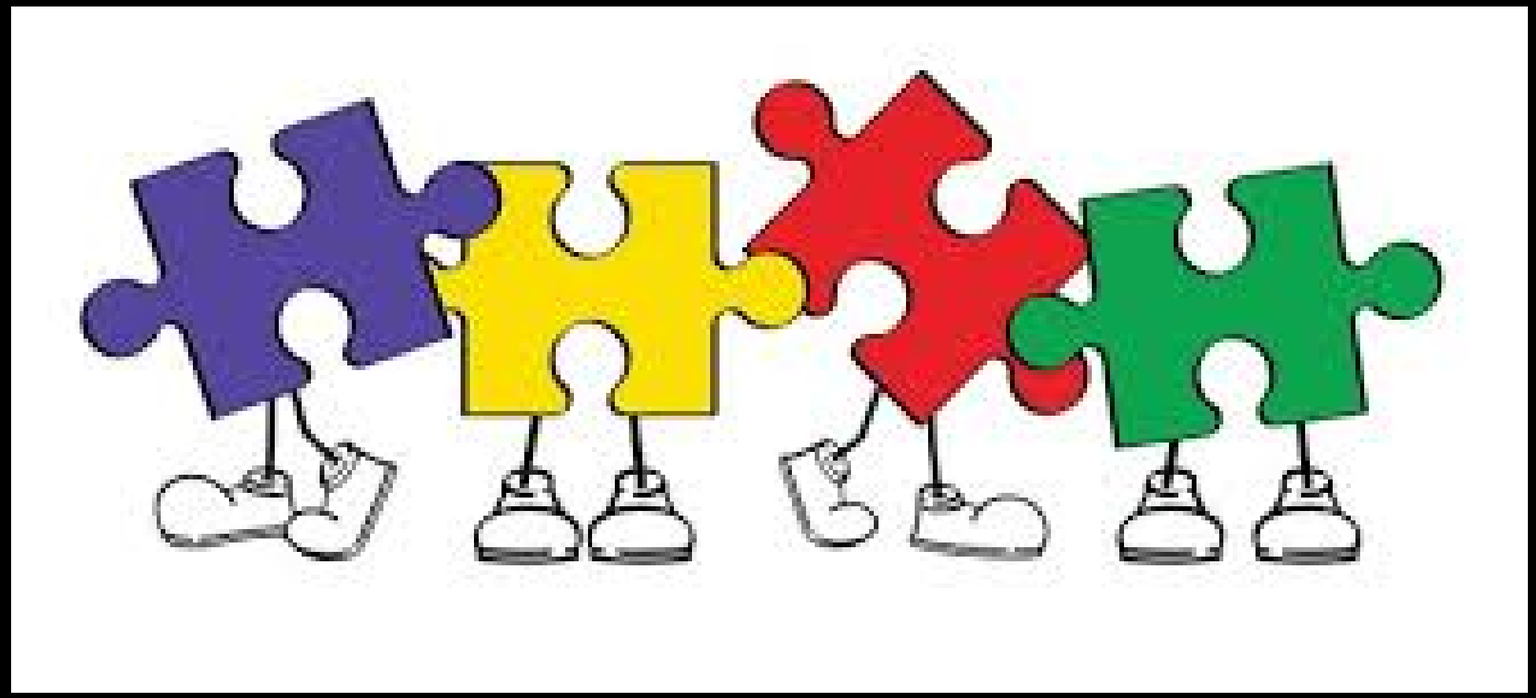
We will then go into two practice problems (one warmup, one tougher) that apply this concept

By the end, you should have an idea of both what the concept is, and how to use it

Today’s Meeting is on...

Greedy Algorithm Introduction

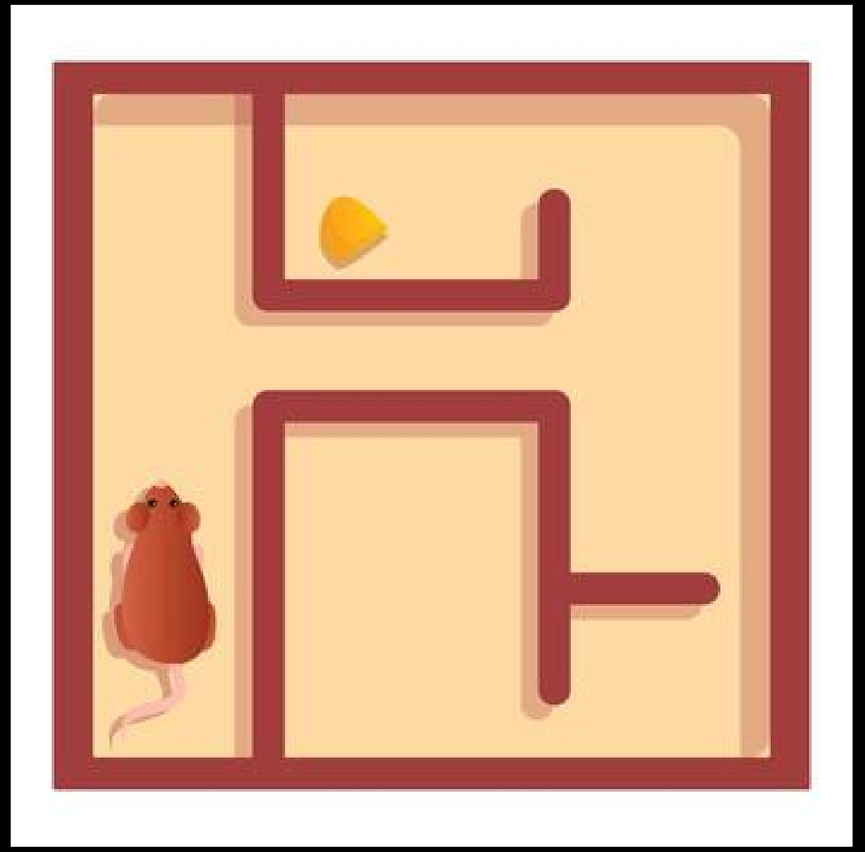
When we have a large complex problem, we can split it up into several small decisions



Greedy Algorithm Introduction

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

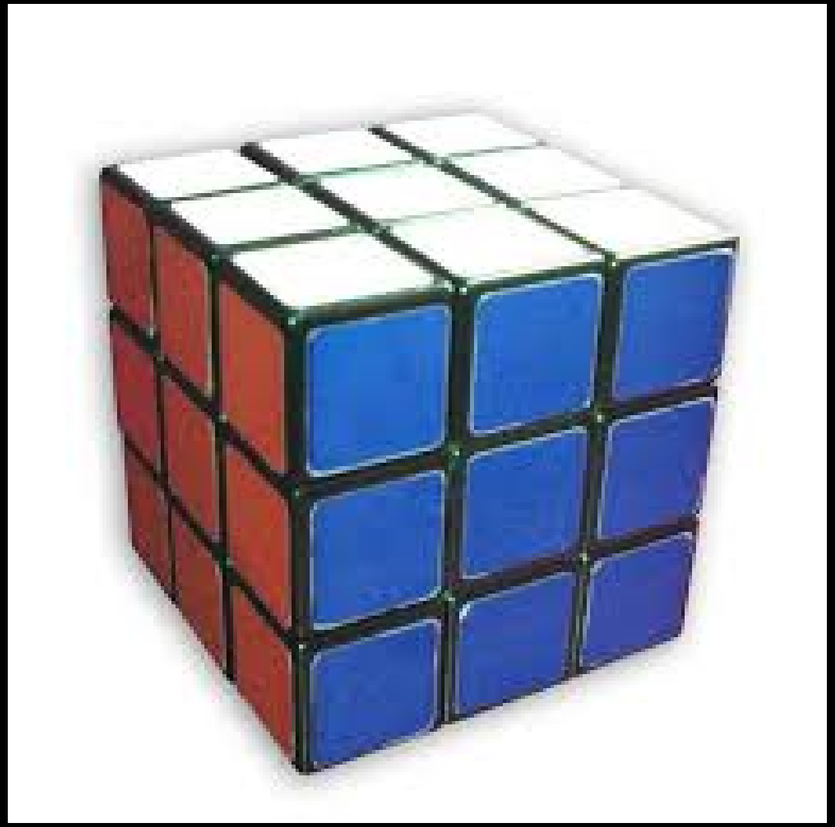
-Goes Forward
-Goes Left
-Goes Right
-Goes Backward



Greedy Algorithm Introduction

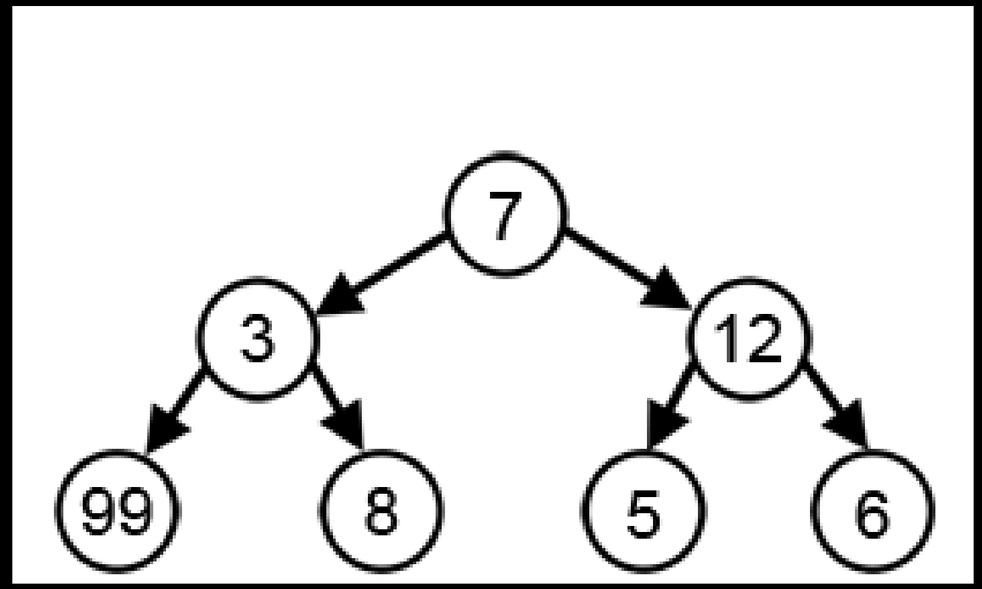
Solving a Rubik's Cube can be split into a long string of several different turns where for each one we can:

Take either of the six sides and spin them either clockwise or counterclockwise (12 options per decision)



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

Lastly, 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

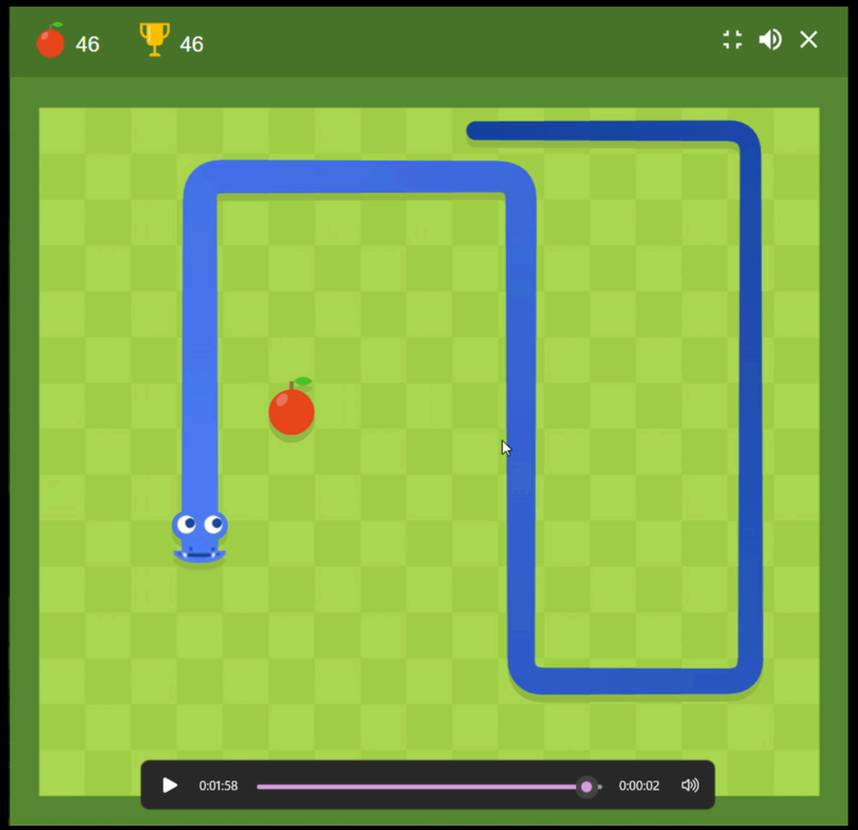
Greedy Algorithm

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

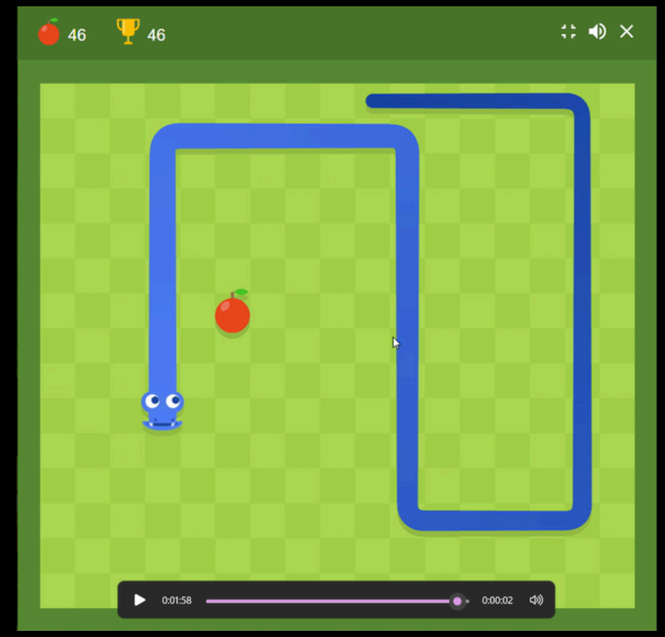




Revisiting Snake

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?



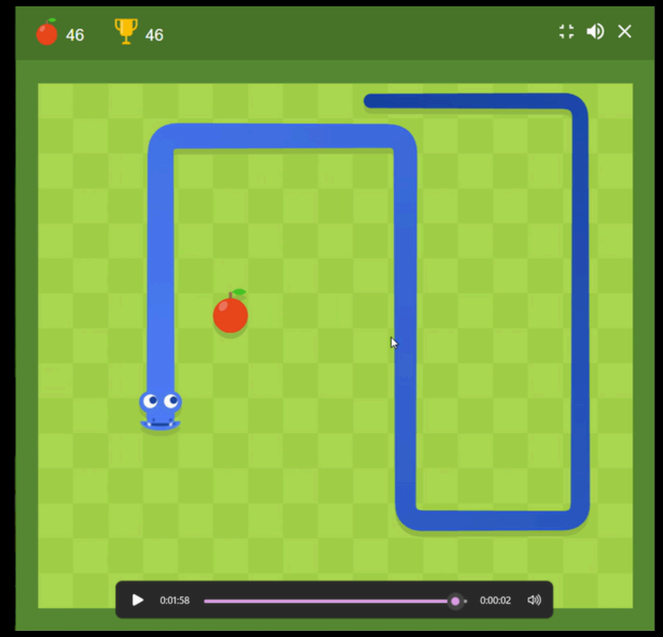


Revisiting Snake

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



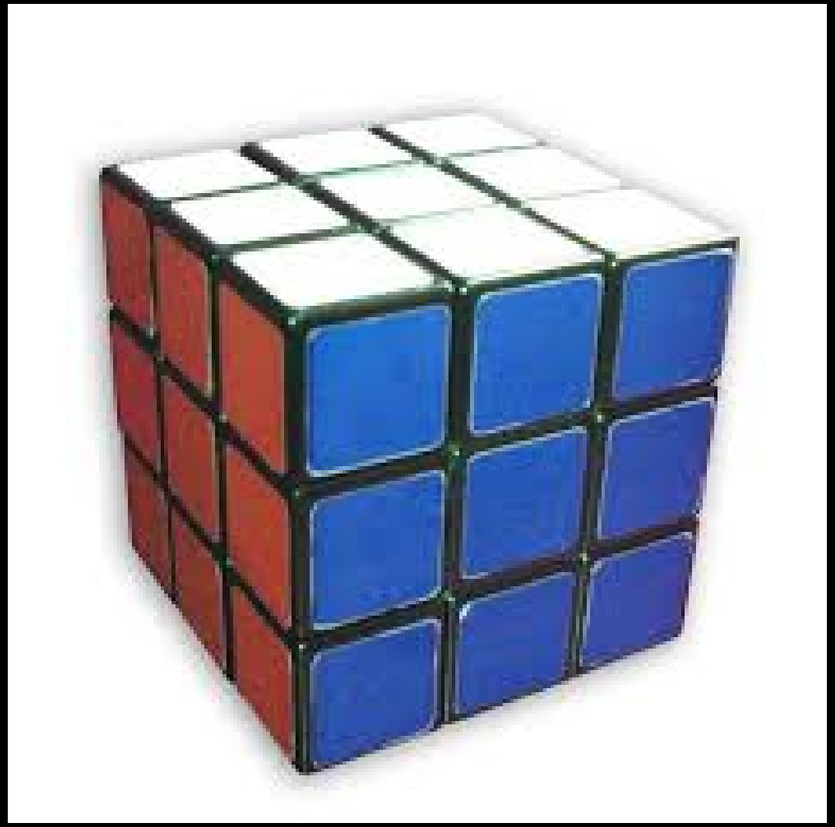


Revisitng the Rubik's Cube

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.

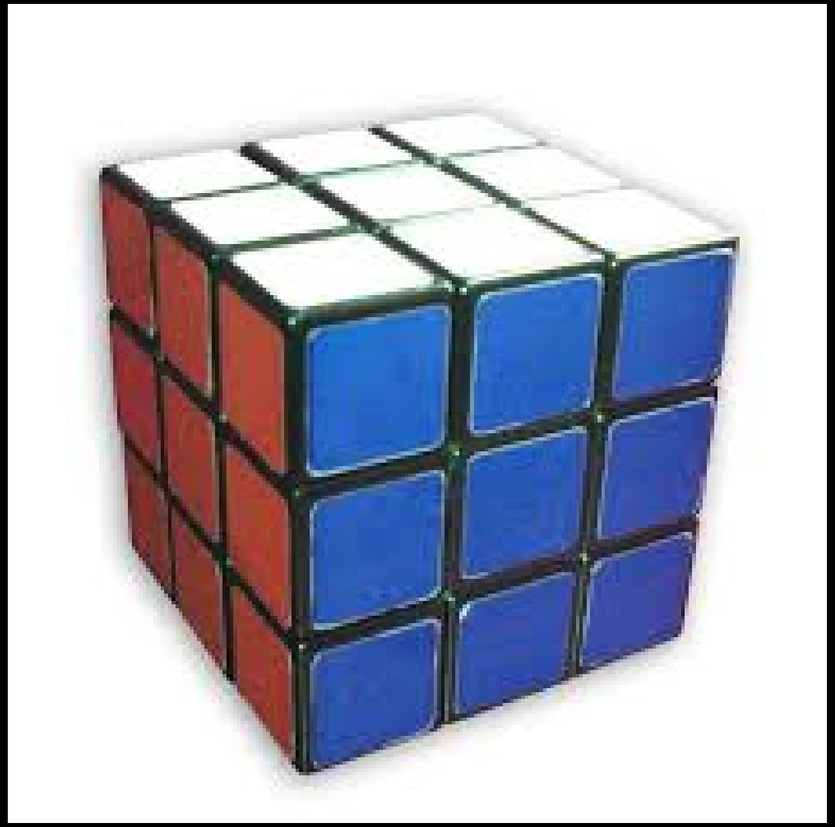
This means we can treat each decision as its own completely isolated subproblem



Revisitng the Rubik's Cube

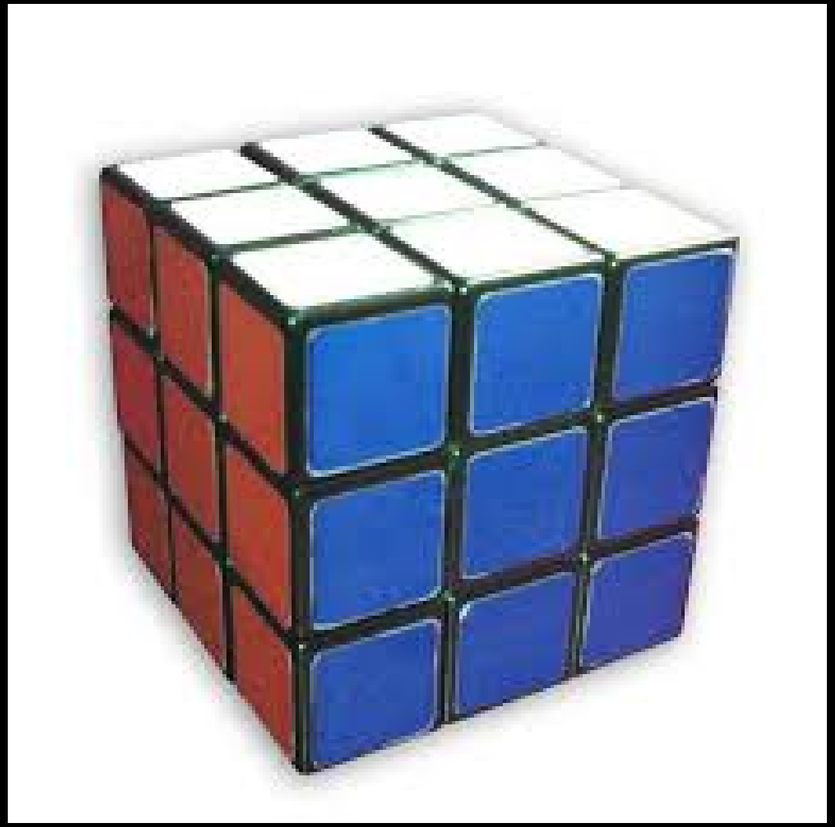
Remember, a Greedy Algorithm will always perform the best option available to it for each subproblem

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



Revisitng the Rubik's Cube

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

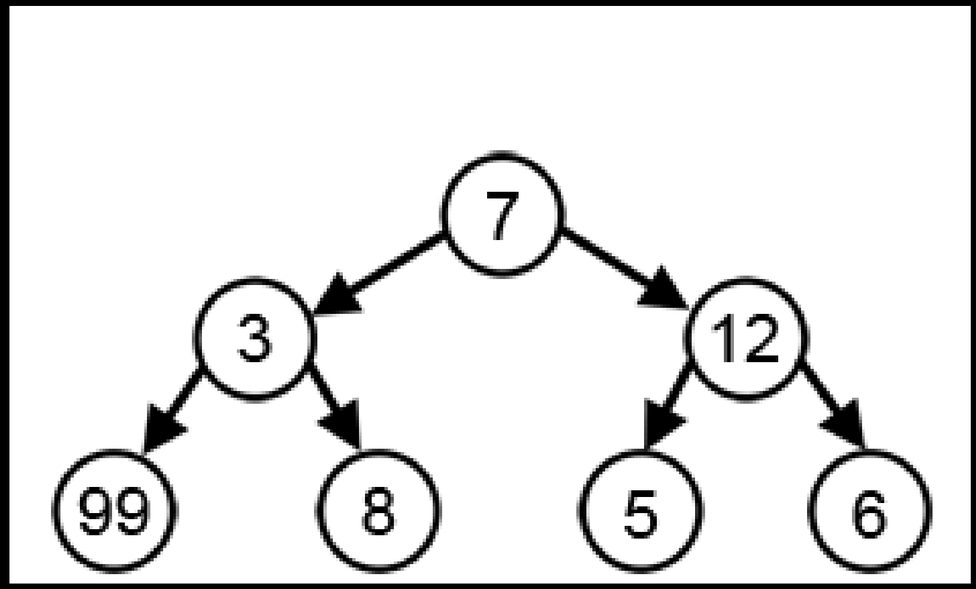


Revisiting the Tree

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!

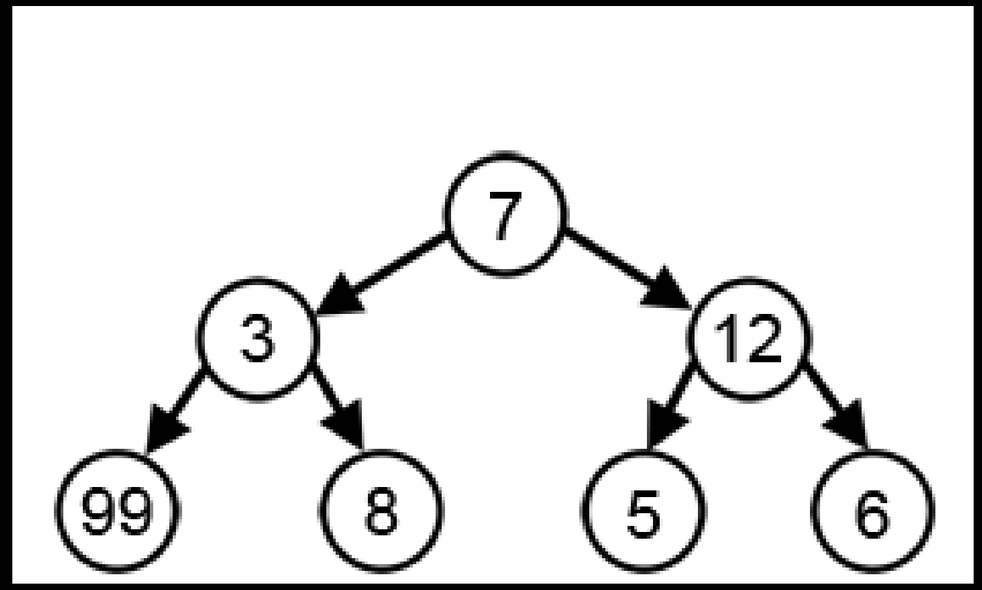


Revisiting the Tree

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

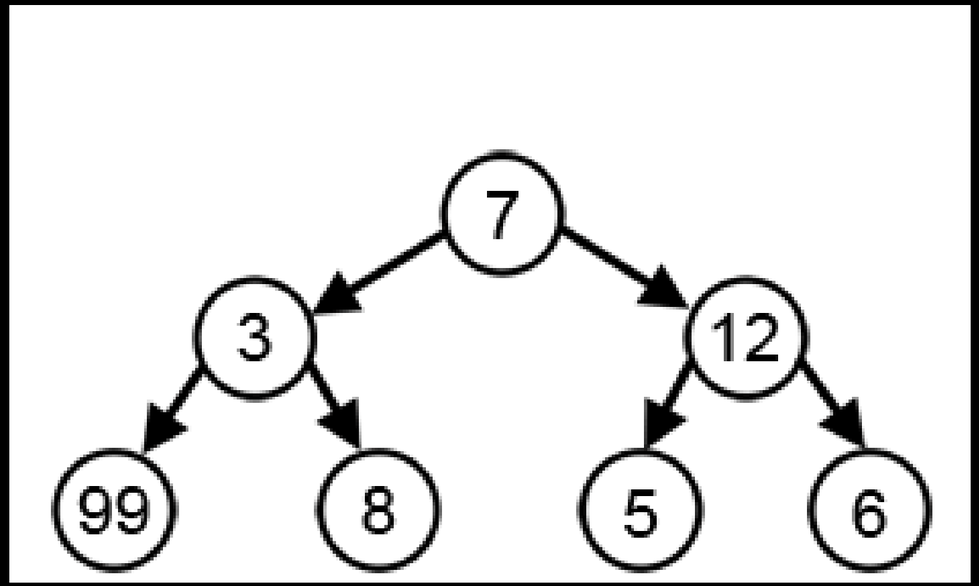
Start at 7:
Choose 12 → Choose 6

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 define 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.*

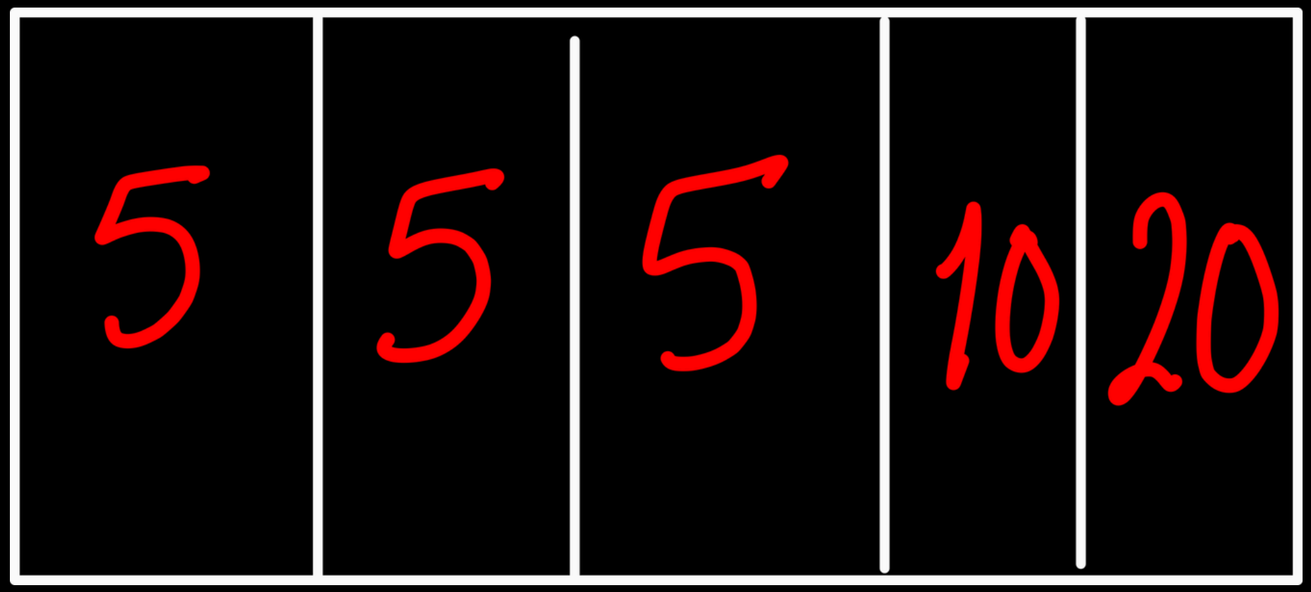
What’s required?

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.

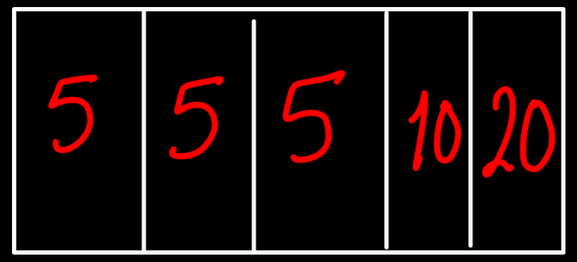
Example Test Cases:

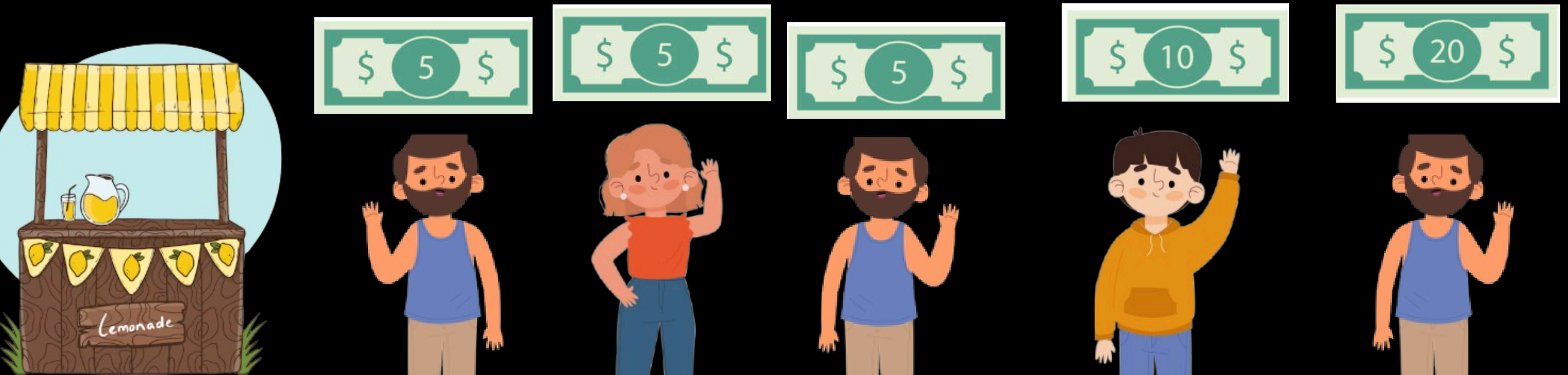
Let’s go over the example test cases, to make it clearer what we’re required to do.



Example Test Cases:

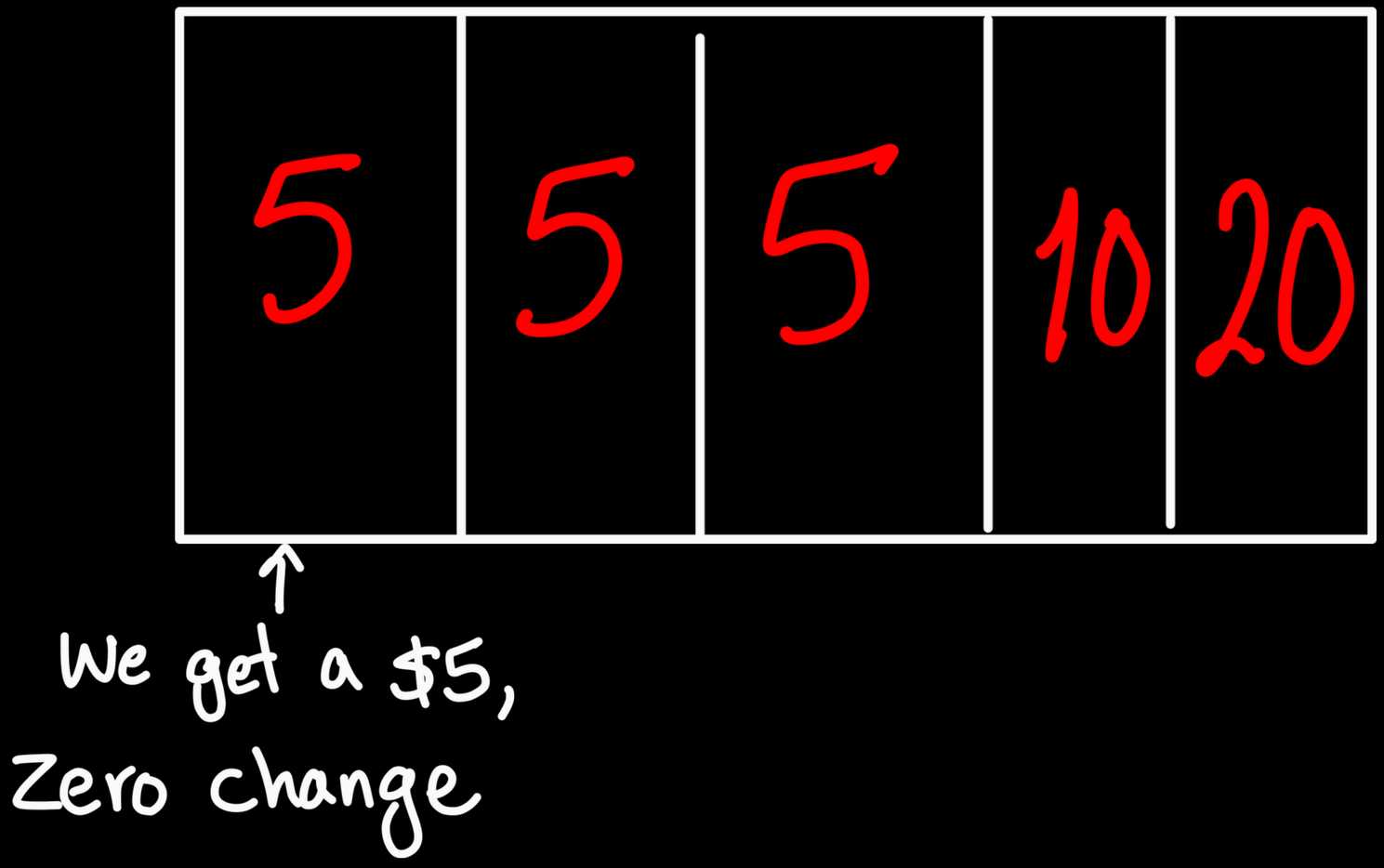
Imagine a line of people at the lemonade stand with only one 5,10 or 20 bill and they all want only 1 lemonade which costs $5



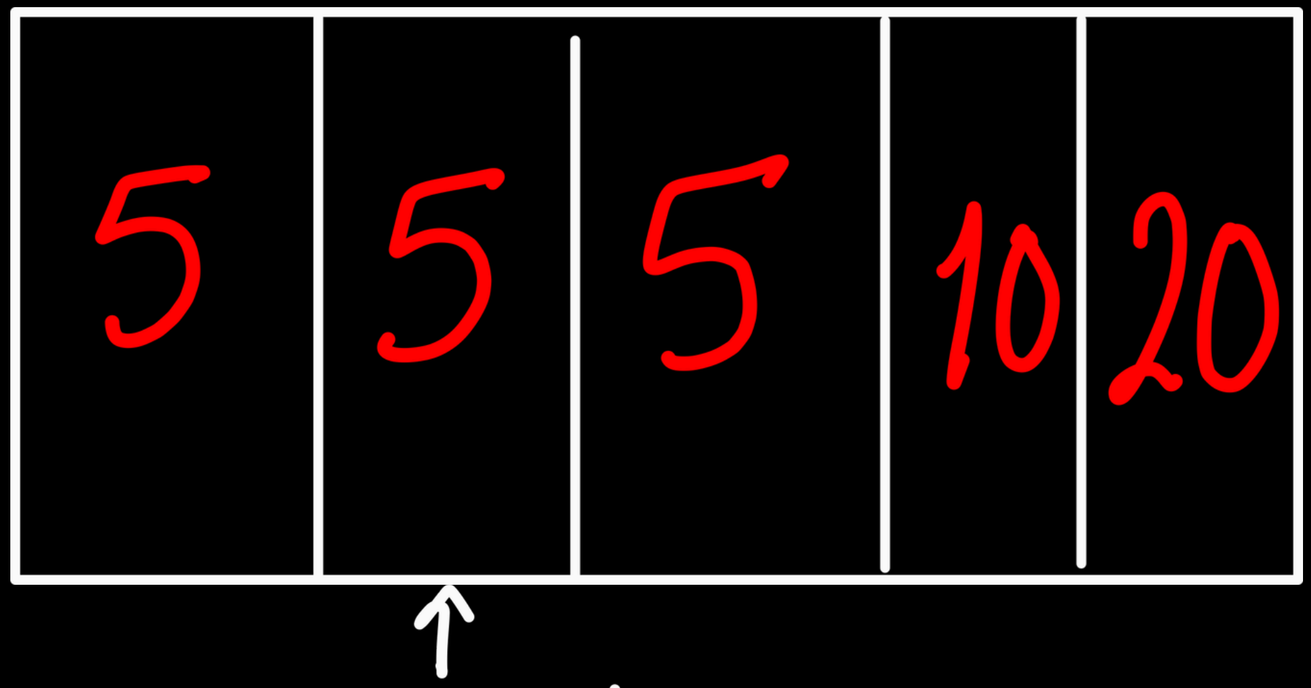


Example Test Cases:

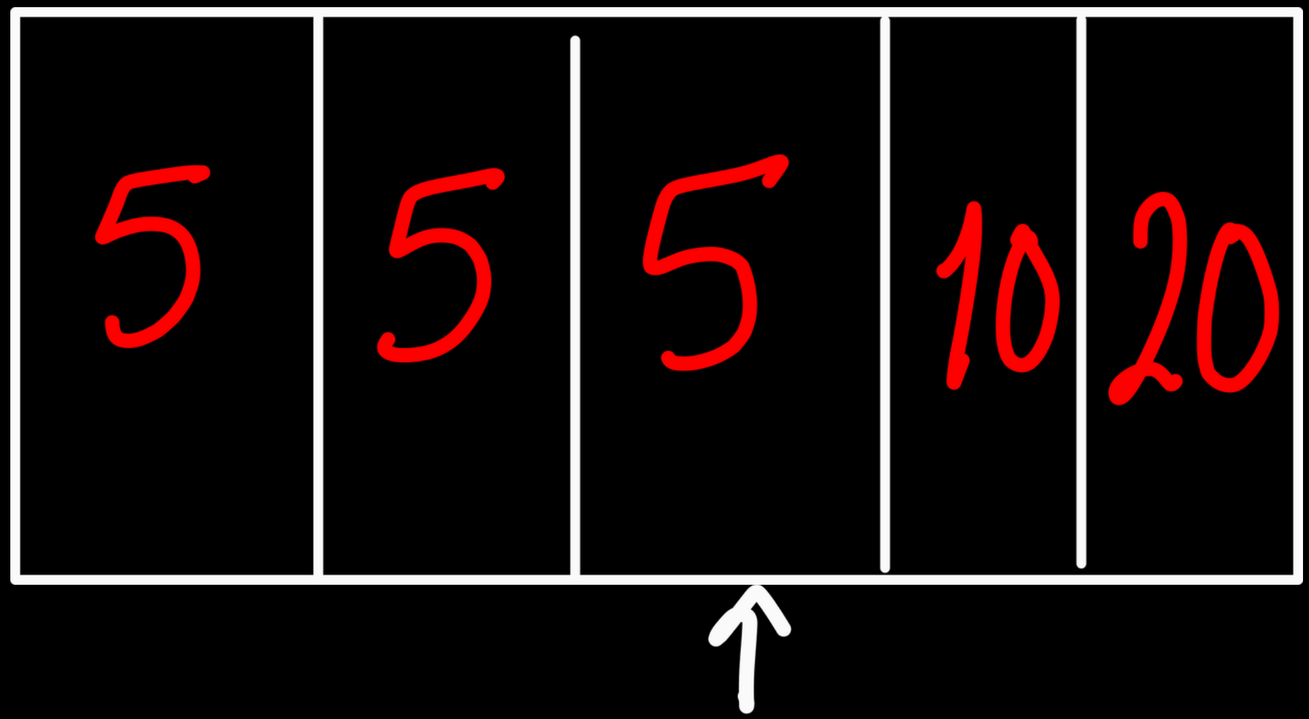
Let’s go over the example test cases, to make it clearer what we’re required to do.



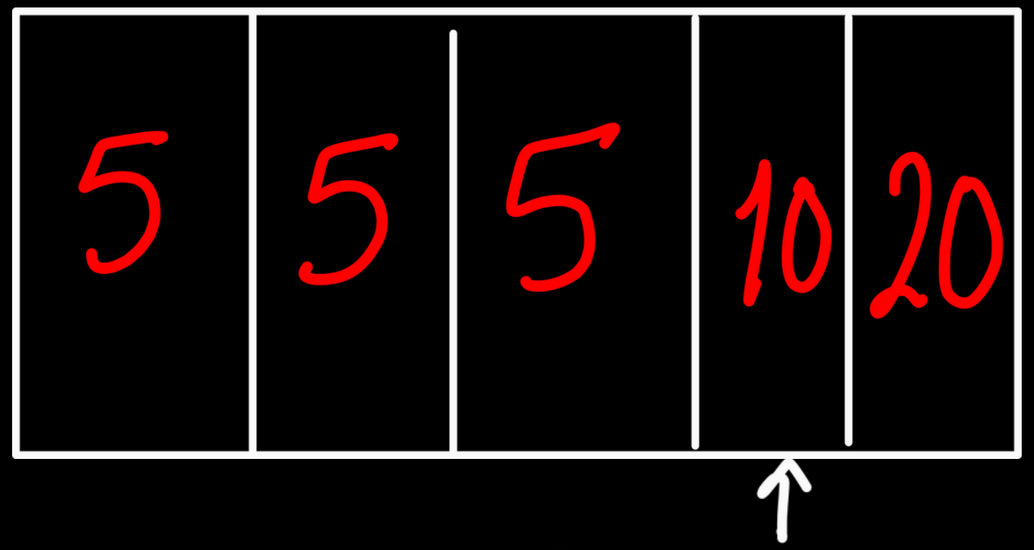
Example Test Cases:



Example Test Cases:

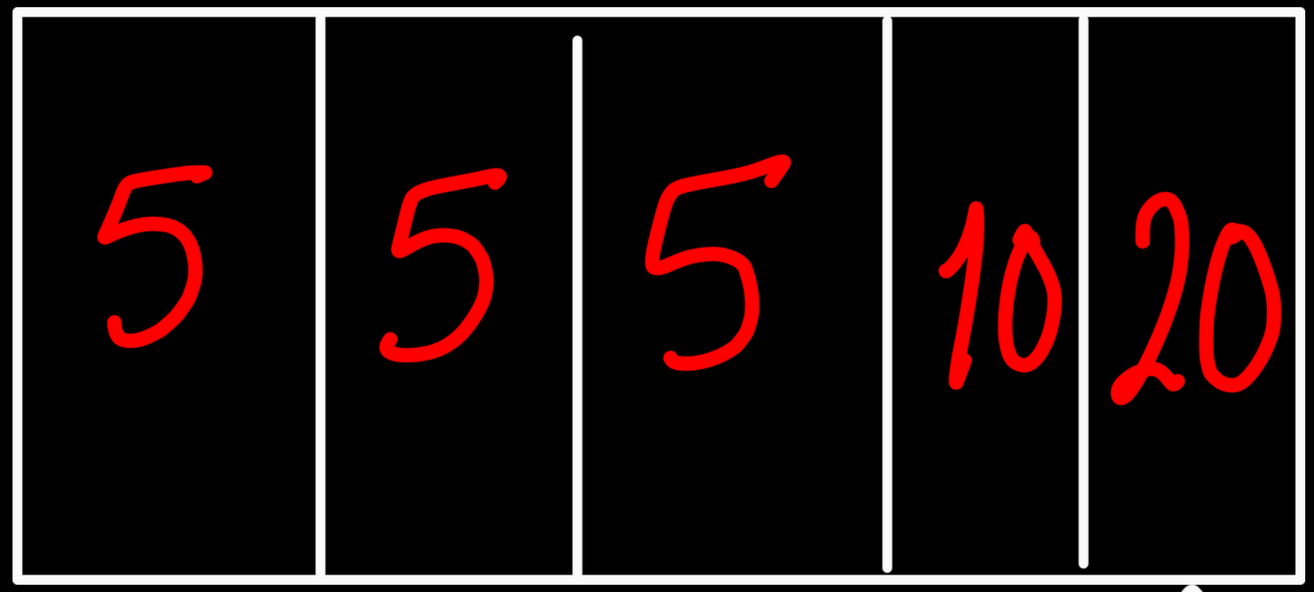


Example Test Cases:



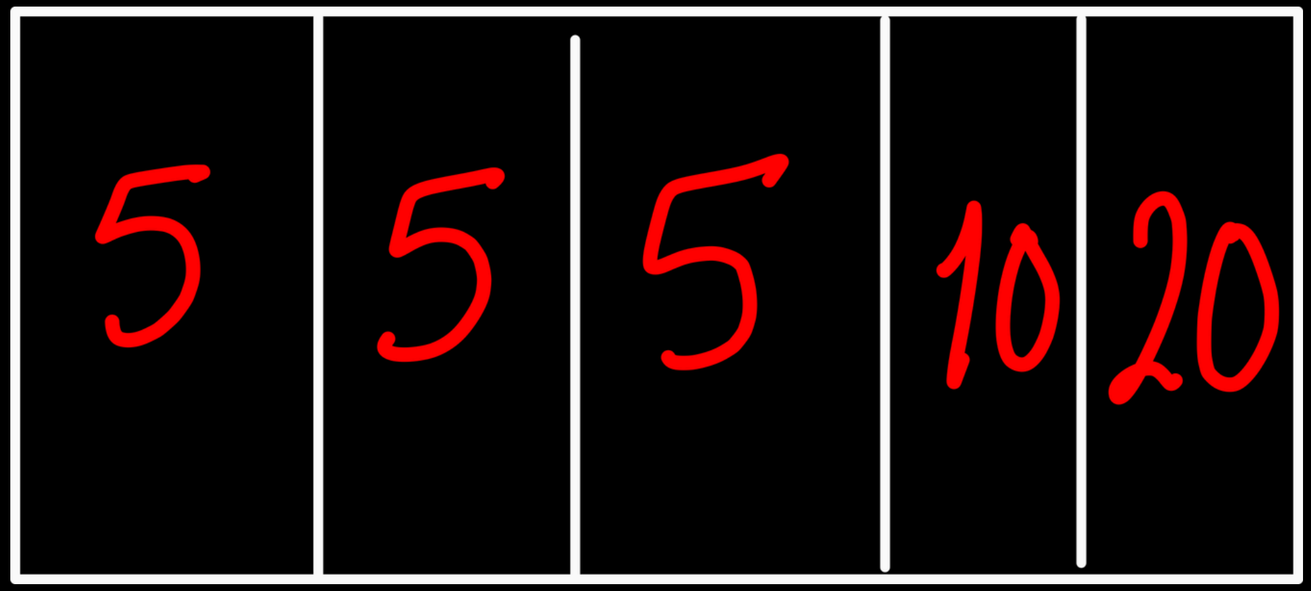
Got a $10 bill.
Take out one of the three fives we have.
We now have $10 and two fives.

Example Test Cases:



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

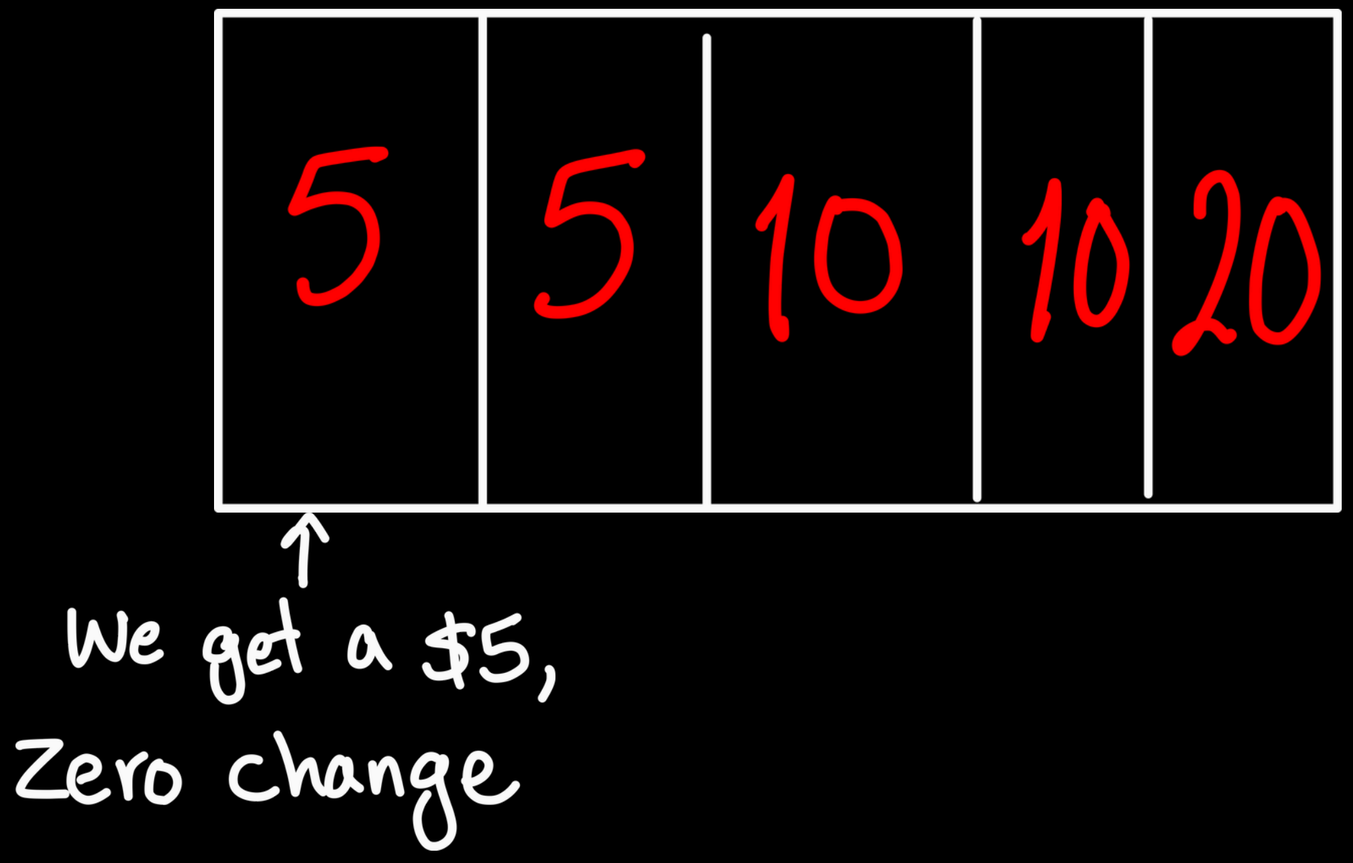
Example Test Cases:



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.

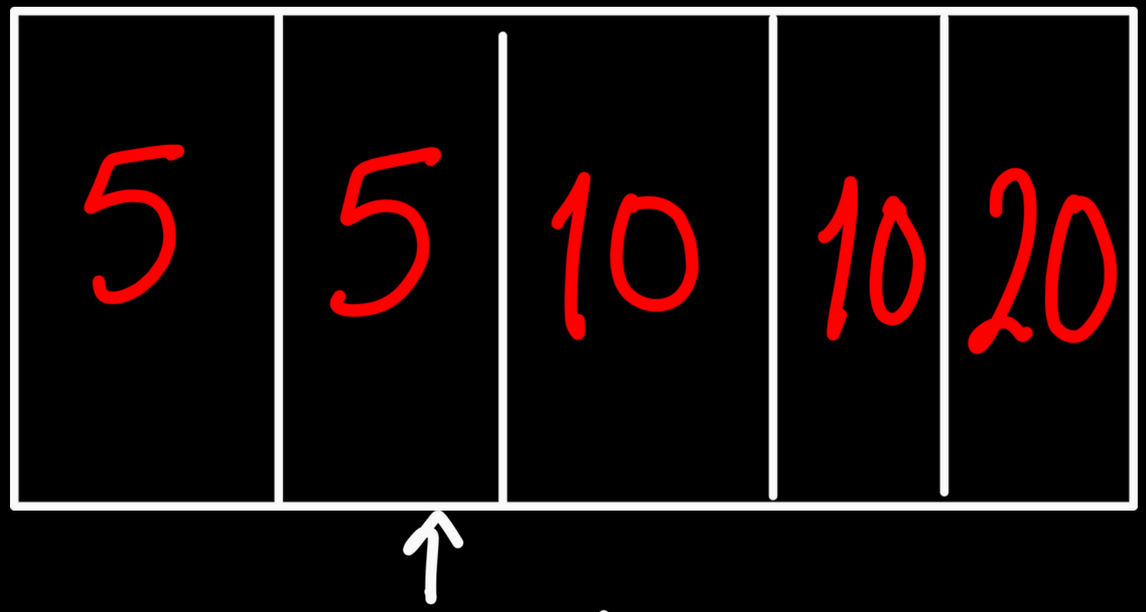
Example Test Cases:

Second Test Case:



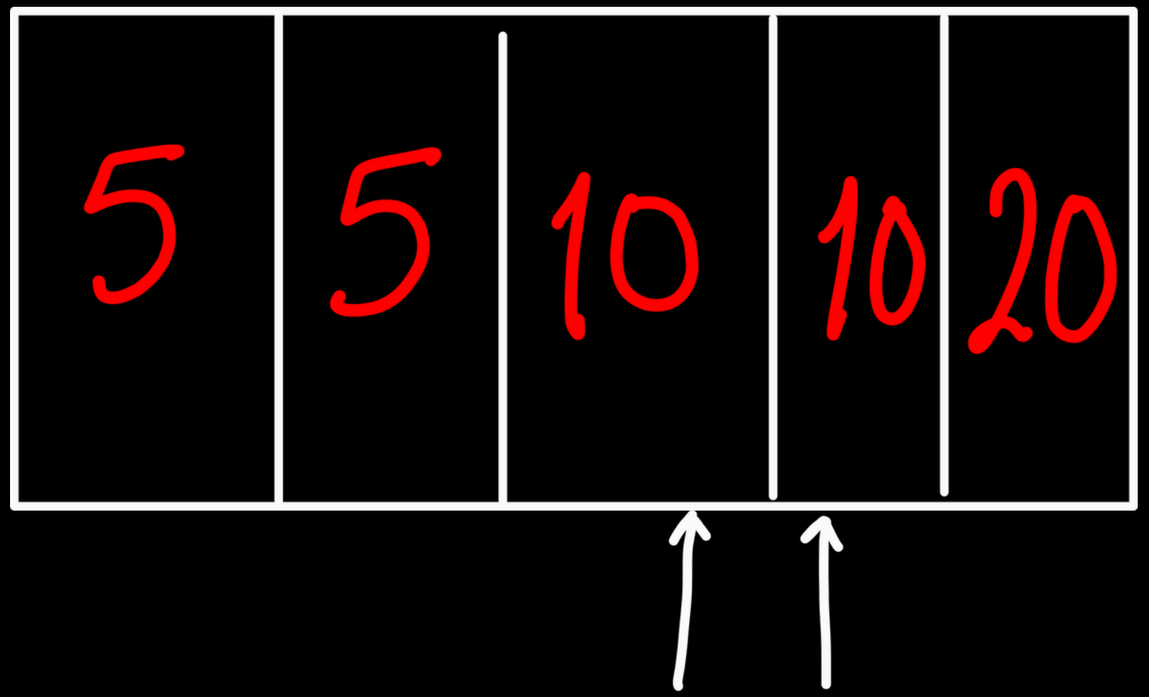
Example Test Cases:

Second Test Case:



Example Test Cases:

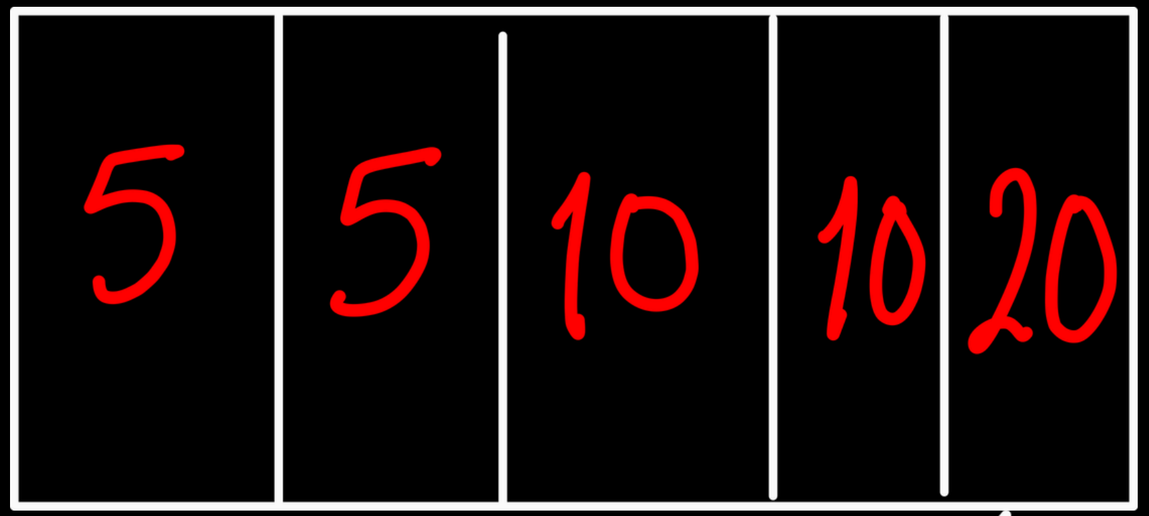
Second Test Case:



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

Example Test Cases:

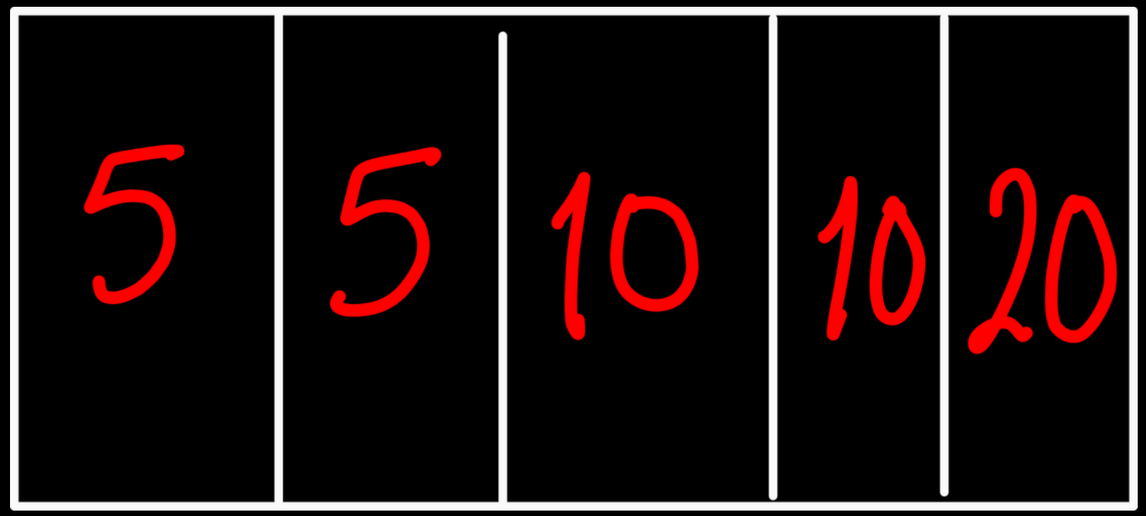
Second Test Case:



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

Example Test Cases:

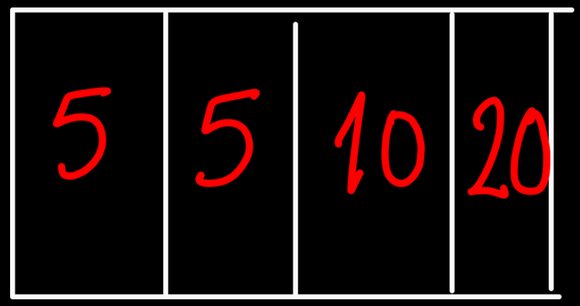
Second Test Case:



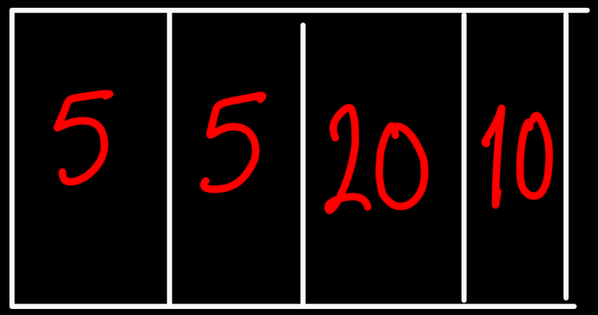
This testcase will, therefore, return False.

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.

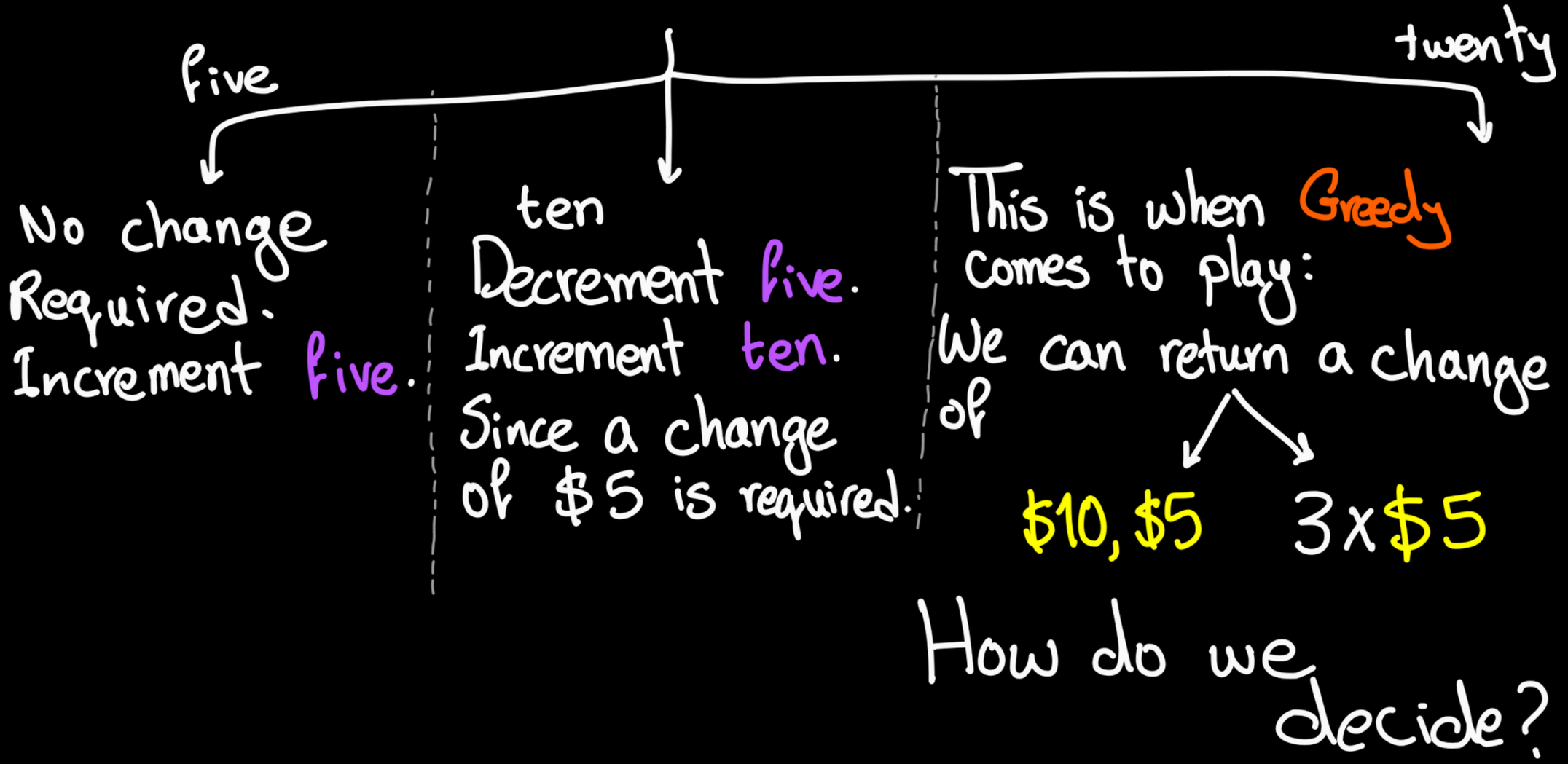
There’re only 3 integers we can encounter as per the constraints:

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.

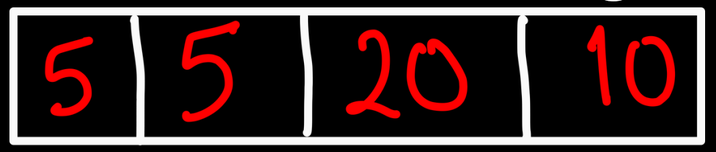
Therefore, to be greedy, it would make more sense to check if we have a $10 bill first,
and if we don’t, we could take out 3 $5 bills.

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

```java
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;
```

time complexity: O(n)
space complexity: O(1)

The Problem

991. Broken Calculator

Medium | Topics | Companies

There is a broken calculator that has the integer `startValue` on its display initially. In one operation, you can:

- multiply the number on display by `2`, or
- subtract `1` from the number on display.

Given two integers `startValue` and `target`, return *the minimum number of operations needed to display* `target` *on the calculator.*

Example 1:

```
Input: startValue = 2, target = 3
Output: 2
Explanation: Use double operation and then decrement operation {2 -> 4
-> 3}.
```

Example 2:

```
Input: startValue = 5, target = 8
Output: 2
Explanation: Use decrement and then double {5 -> 4 -> 8}.
```

Example 3:

```
Input: startValue = 3, target = 10
Output: 3
Explanation: Use double, decrement and double {3 -> 6 -> 5 -> 10}.
```

Constraints:

- `1 <= startValue, target <= 10^9`

https://leetcode.com/problems/broken-calculator/

Test Cases

Example 2:

```
Input: startValue = 5, target = 8
```

ARSENAL:

SUBTRACT BY 1

MULTIPLY BY 2

Test Cases

Example 2:

```
Input: startValue = 5, target = 8
```

5
→ 4 (Subtract by one)
→ 8 (Multiply by two)

```
Output: 2
```

ARSENAL:

SUBTRACT BY 1

MULTIPLY BY 2

Test Cases

Example 3:

```
Input: startValue = 3, target = 10
```

3
→ 6 (Multiply by two)
→ 5 (Subtract by one)
→ 10 (Multiply by two)

```
Output: 3
```

ARSENAL:

SUBTRACT BY 1

MULTIPLY BY 2

Now you try it!!!

Solution

Remember, we are looking to write
a Greedy Solution, meaning however we set
it up, it should take the optimal path.

(Multiplying at every step is NOT necessarily optimal!)

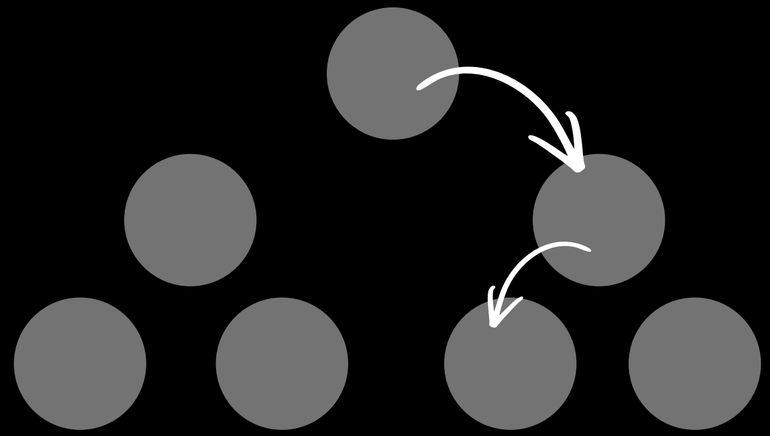
Solution

```
int startValue = 5;
int target = 9;
```

5 → 10 (Multiply by two) → 9 (subtract by one)
Multiplication first IS optimal

```
int startValue = 5;
int target = 8;
```

5 → 4 (Subtract by one) → 8 (Multiply by two)
Multiplication first is NOT optimal



Remember the tree from earlier? You would need to do additional checking, which isn't in spirit of a Greedy Algorithm.

Solution

We cannot do a greedy solution in the current problem,
because we cannot determine whether or not
multiplication or subtraction is the best move

However, we can change the problem to
make it greedy:
by starting from the end!

Solution

We cannot do a greedy solution in the current problem,
because we cannot determine whether or not
multiplication or subtraction is the best move

However, we can change the problem to
make it greedy:
by starting from the end!

For this to happen,
OUR ARSENAL NEEDS TO CHANGE:

SUBTRACT BY 1
MULTIPLY BY 2



ADD 1
DIVIDE BY 2

Solution

```
int startValue = 5;
int target = 9;
```

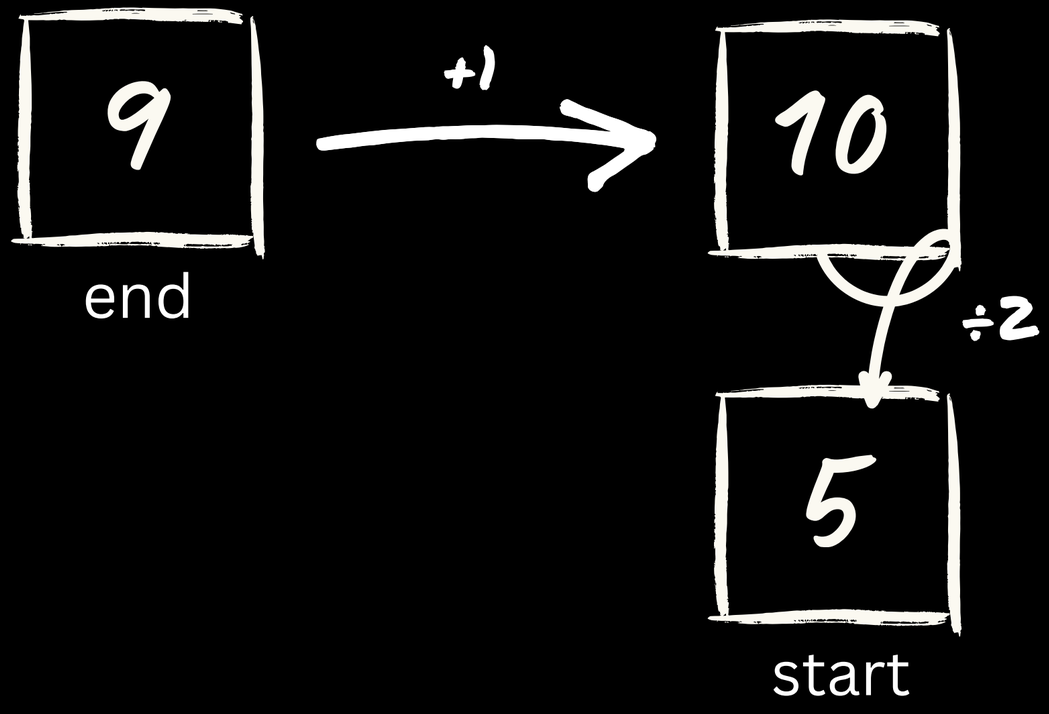
Starting from the end, we can see that simply can't perform 9 ÷ 2.
So, we are forced into adding by one

Next, you divide by two, and you arrive at 5.

ARSENAL:

ADD 1

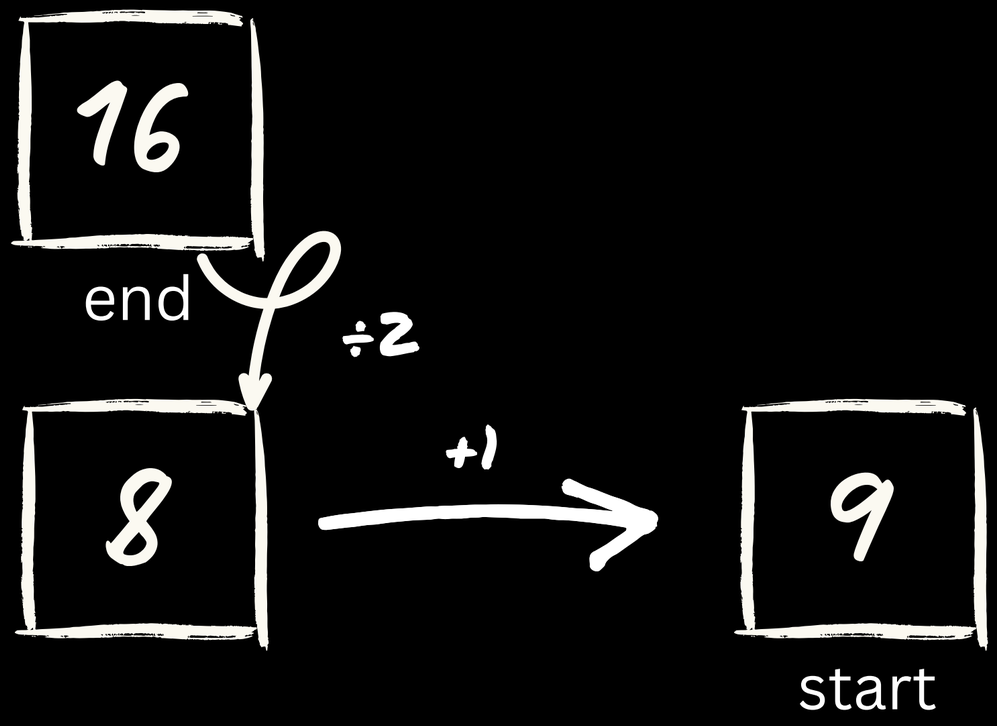
DIVIDE BY 2



Solution

```
int startValue = 5;
int target = 9;
```

In this case, since 16 is even,
we perform 16 ÷ 2 immediately.
Why?



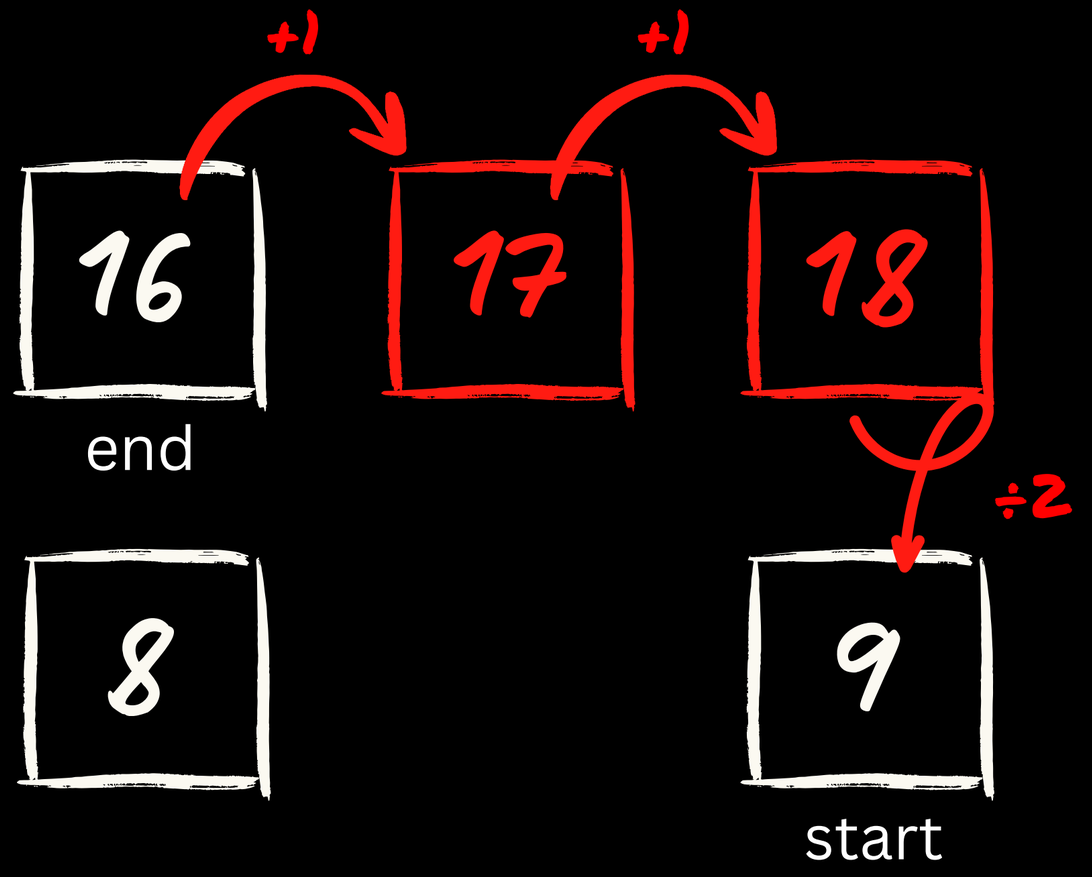
ARSENAL:

ADD 1

DIVIDE BY 2

Solution

```
int startValue = 5;
int target = 9;
```



It's because if the end is even, first dividing by 2 effectively makes our subsequent adds two times as effective!

(Let's be greedy!!!)

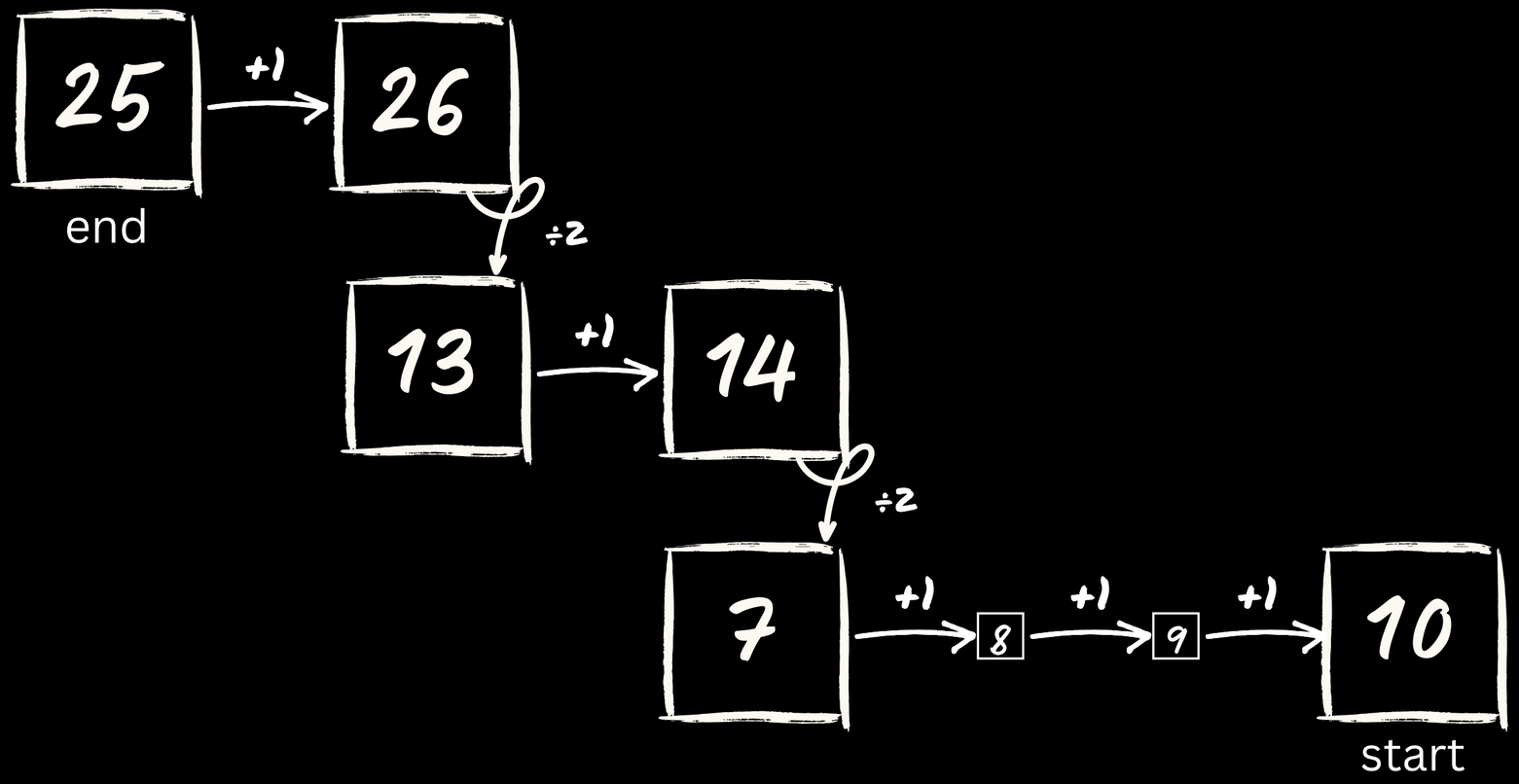
ARSENAL:

ADD 1

DIVIDE BY 2

Solution

```
int startValue = 10;
int target = 25;
```



ARSENAL:

ADD 1

DIVIDE BY 2

We now have our solution (#1)!

- If we have the option to divide, we ALWAYS should
- Otherwise, we should add by one (which converts an odd to an even number, which can be divided)

Java Implementation..?

Code

Java Auto

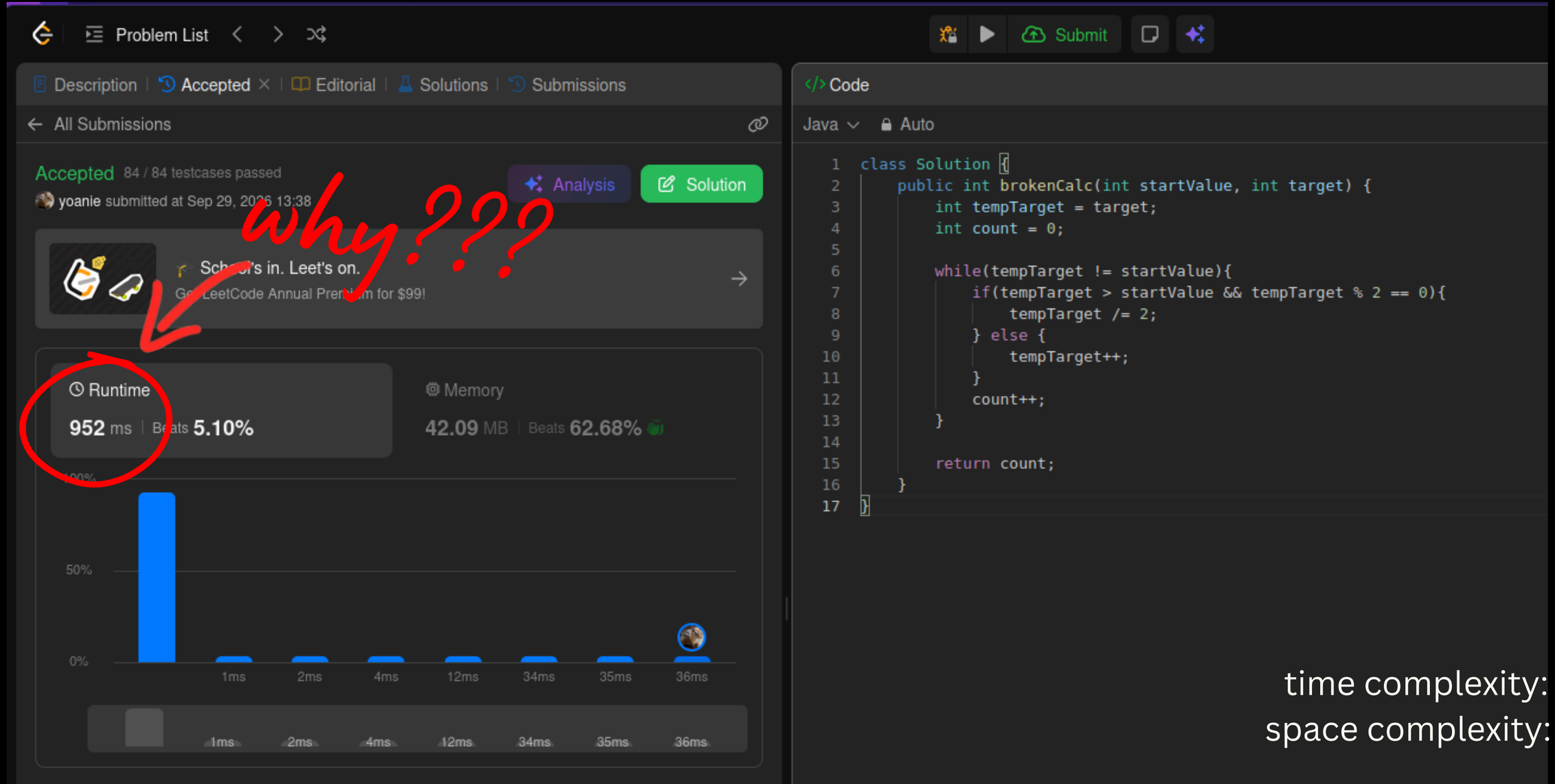
```java
class Solution {
    public int brokenCalc(int startValue, int target) {
        int tempTarget = target;
        int count = 0;

        while(tempTarget != startValue){
            if(tempTarget > startValue && tempTarget % 2 == 0){
                tempTarget /= 2;
            } else {
                tempTarget++;
            }
            count++;
        }

        return count;
    }
}
```

time complexity: O(n)
space complexity: O(1)

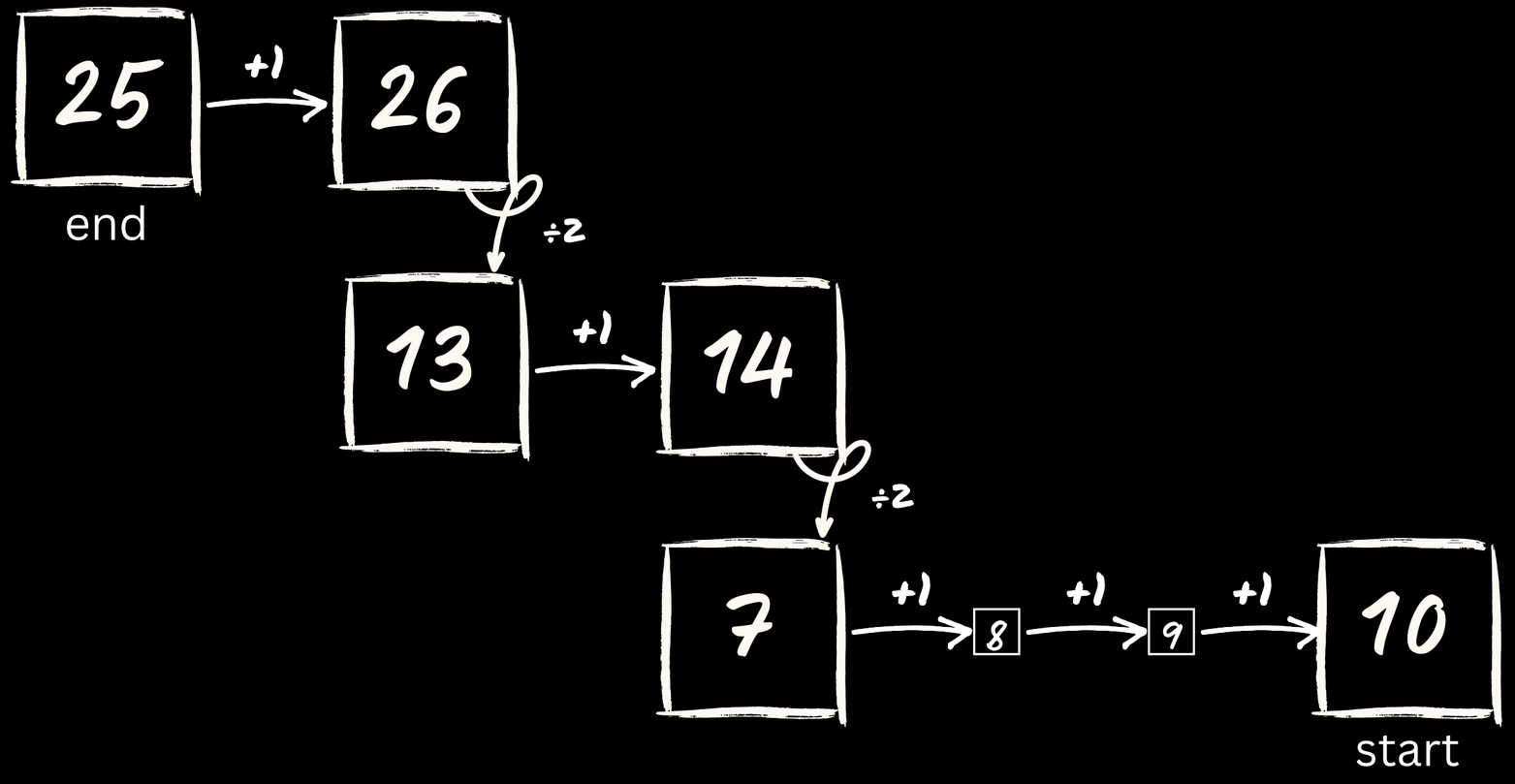
Java Implementation..?

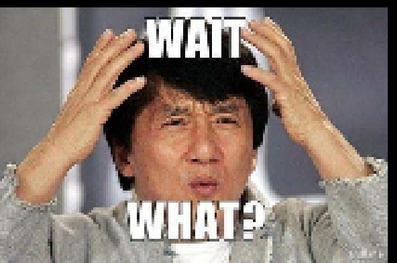


time complexity: O(n)
space complexity: O(1)

Further Optimization

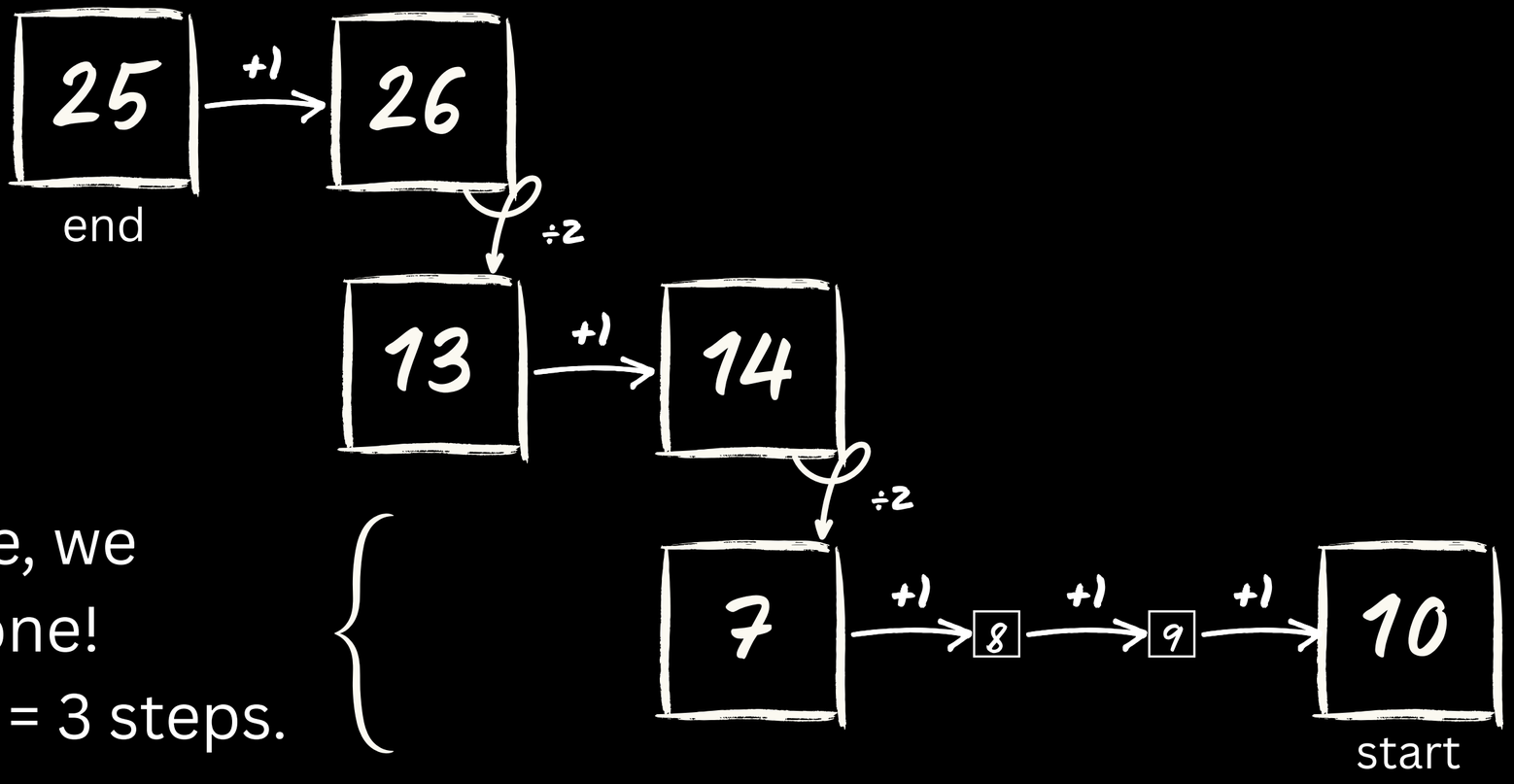
So why is my algorithm so slow????
Let’s take a look at the last example we did:





Further Optimization

So why is my algorithm so slow????
Let’s take a look at the last example we did:



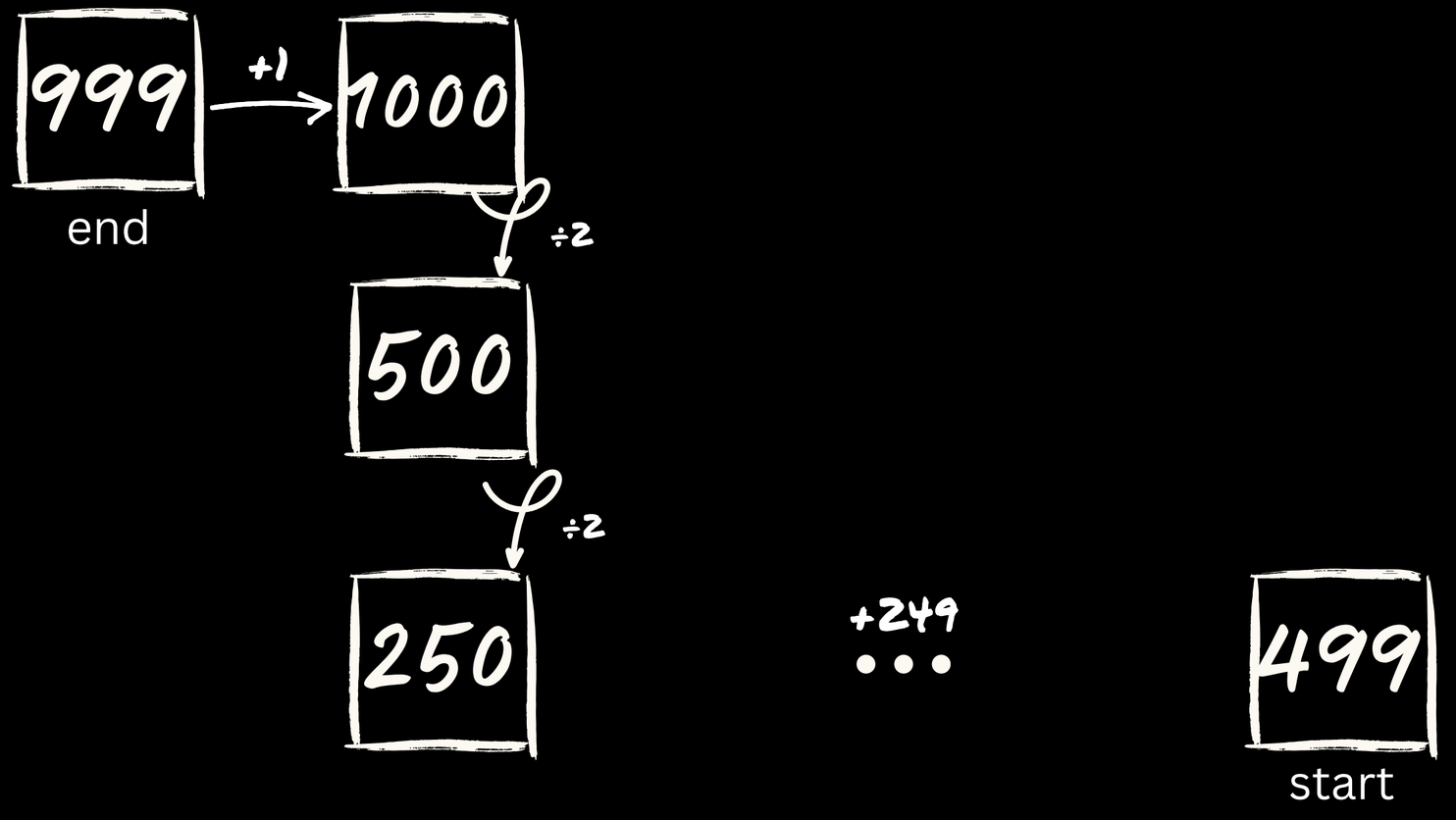
Once we get here, we
are basically done!
This just takes (10 - 7) = 3 steps.

Why does this save so much?

```
int startValue = 499;
int target = 999;
```

Simply put, adding like this takes a lot of time, while all you need to do to get to the bottom “layer” is to keep dividing by 2.

more on this in the next meeting!



We now have our solution (#2)!

While our target is greater than starting number:

- If we have the option to divide, we ALWAYS should
- Otherwise, we should add by one (which converts an odd to an even number, which can be divided)

Once target < starting number,

- return operation count + difference between target and starting number

Java Implementation

Problem List

Submit

Description | Editorial | Solutions | Accepted | Submissions

All Submissions

Accepted 84 / 84 testcases passed

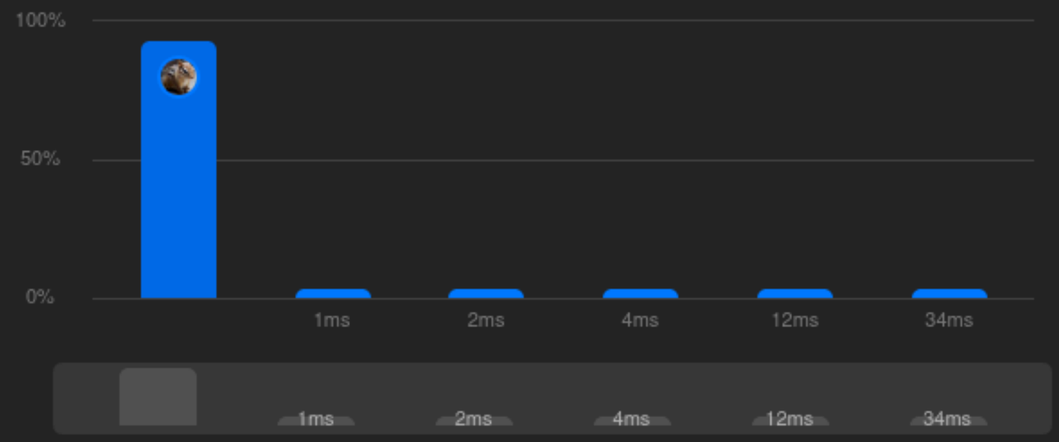
yoanie submitted at Sep 30, 2026 16:40

Analysis Solution

Last Call. Leet's On.
Grab our Back to School student deal before it's gone!

Runtime
0 ms | Beats 100.00%

Memory
42.07 MB | Beats 63.00%



Code

Java | Auto

```java
class Solution {
    public int brokenCalc(int startValue, int target) {
        int tempTarget = target;
        int count = 0;

        while(tempTarget > startValue){
            if(tempTarget % 2 == 1){
                tempTarget++;
            } else {
                tempTarget /= 2;
            }
            count++;
        }

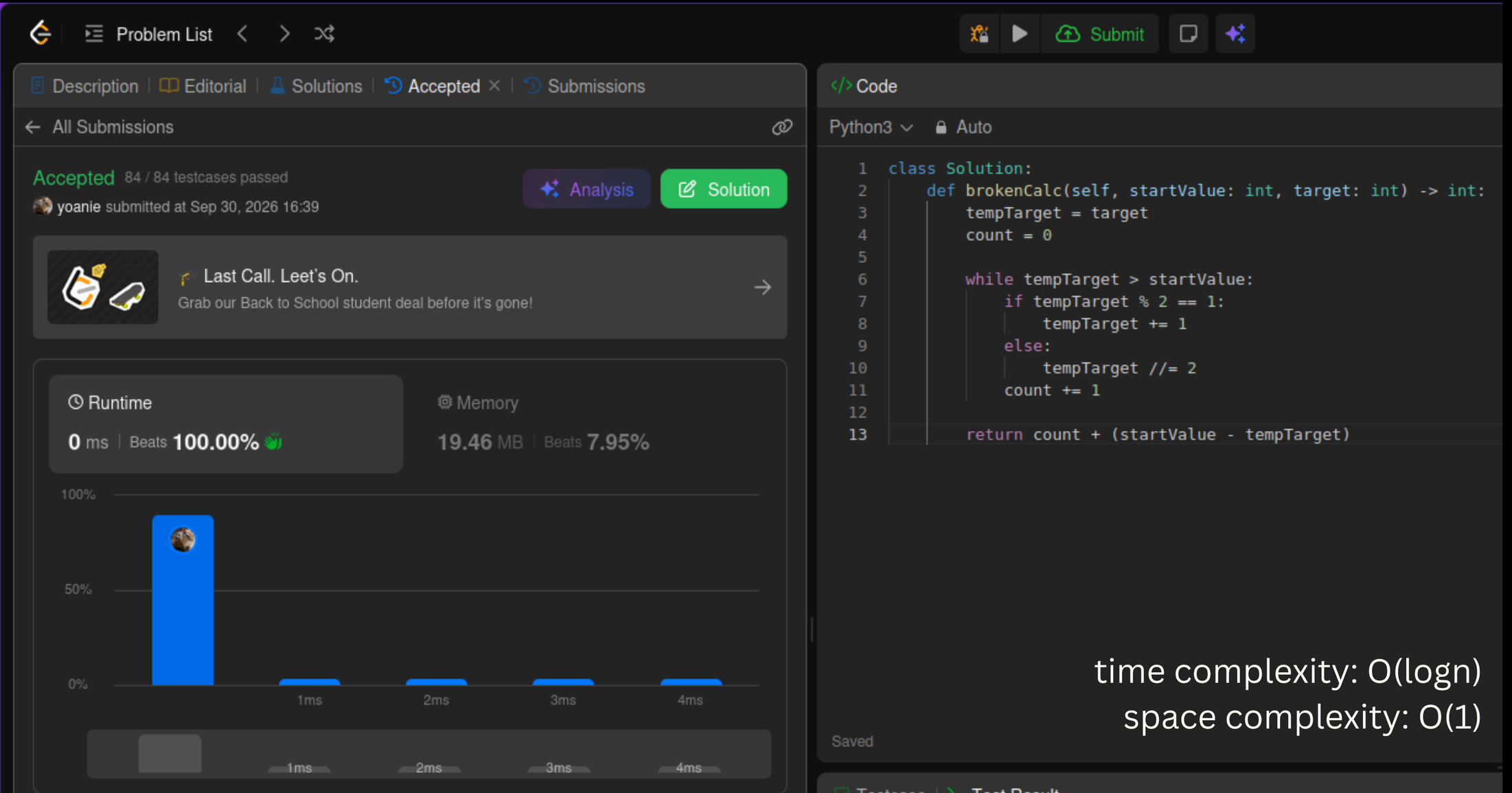
        return count + (startValue - tempTarget);
    }
}
```

Saved

time complexity: O(logn)
space complexity: O(1)



Python Implementation



time complexity: O(logn)
space complexity: O(1)

Hacker Devils

Presented by Sohum Sheth, Dean Richardson and Griffin Charlson

THANK YOU

for your time and attention

