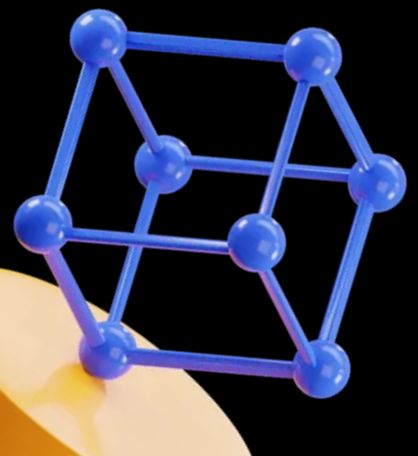
Sohum Sheth, Dean Richardson, Griffin Charlson

HKD MEDIUM SECTION #2

Time Complexity and Big-O

October 14th, 2026



Attendance and Announcements



- 🍕Pizza at the end as always :)

So what is this Big-O stuff, anyway?

Last time, we left off with a solution that vastly improved
its run time by implementing a minor change,
but how do you know how much it changes?

In short, you may think of a lot of algorithms as functions. They have an input and an output.

But unlike in math, **this function now has a factor of how long it takes**.

x = 10

find_square_root(x);



output: 3.16

takes 3 seconds

For example, let’s just say our function does something like this:

```
  6 void function(int[] array){
  5   int n = array.length;
  4   for(int i = 0; i < n; i++){
  3     System.out.println("hi"); //takes 1 second
  2   }
  1 }
7
~
~
```

and my array looks like this:

```
int[] array = {1, 2, 3};
```

how many seconds would my function() take to run?

> 3 · 1 second = 3 seconds

Now, what if I change it to be this:

```
 7 void function(int[] array){
 6   int n = array.length;
 5   for(int i = 0; i < n; i++){
 4     for(int j = 0; j < n; j++){
 3       System.out.println("hi"); //takes 1 second
 2   }}
 1 }
8
~
```

and my array looks like this:

```
int[] array = {10, 9, 8, 7};
```

how many seconds would my function() take to run?

> $4^2 \cdot 1$ second = 16 seconds

Now, obviously print statements don’t take 1 second to run.
But what if you were advertising your code to Google, and
you didn’t have the best computer?

What if there was some way to measure a
function’s performance, even when technology changes?

```
 7 void function(int[] array){
 6   int n = array.length;
 5   for(int i = 0; i < n; i++){
 4     for(int j = 0; j < n; j++){
 3       System.out.println("hi"); //takes 1 second
 2   }}
 1 }
8
~
```

That is where Big-O comes in!

Big-O is a way to measure a function's runtime without using seconds, but instead as **mathematical functions**.

Imagine you didn’t know what size array was going into this function. How many “seconds” would this function take?

```
 7 void function(int[] array){
 6   int n = array.length;
 5   for(int i = 0; i < n; i++){
 4     for(int j = 0; j < n; j++){
 3       System.out.println("hi"); //takes 1 second
 2   }}
 1 }
8
~
```

```
 7 void function(int[] array){
 6   int n = array.length;
 5   for(int i = 0; i < n; i++){
 4     for(int j = 0; j < n; j++){
 3       System.out.println("hi"); //takes 1 second
 2   }}
 1 }
8
~
```

Imagine you didn't know what size array was going into this function.
How many "seconds" would this function take?

N = array length

We have a nested for loop, meaning the print statement is run N * N (or N^2) times.

No matter the value of N, this code takes N^2 seconds

```
7 void function(int[] array){
6   int n = array.length;
5   for(int i = 0; i < n; i++){
4     for(int j = 0; j < n; j++){
3       System.out.println("hi"); //takes 1 second
2   }}
1 }
8
~
```

Imagine you didn't know what size array was going into this function.
How many "seconds" would this function take?

The time complexity is O(N^2)

$O(N^2)$ does not mean that a function takes exactly N^2 seconds, but that the total time is proportional to N^2

$O(N^2)$: Execution time $\propto$ N^2

Some more examples:

Let’s say we have a function that accepts N=100 inputs

O(1): Execution Time is **constant**, regardless of input

O(log N) : Execution time is about **6.64** time units

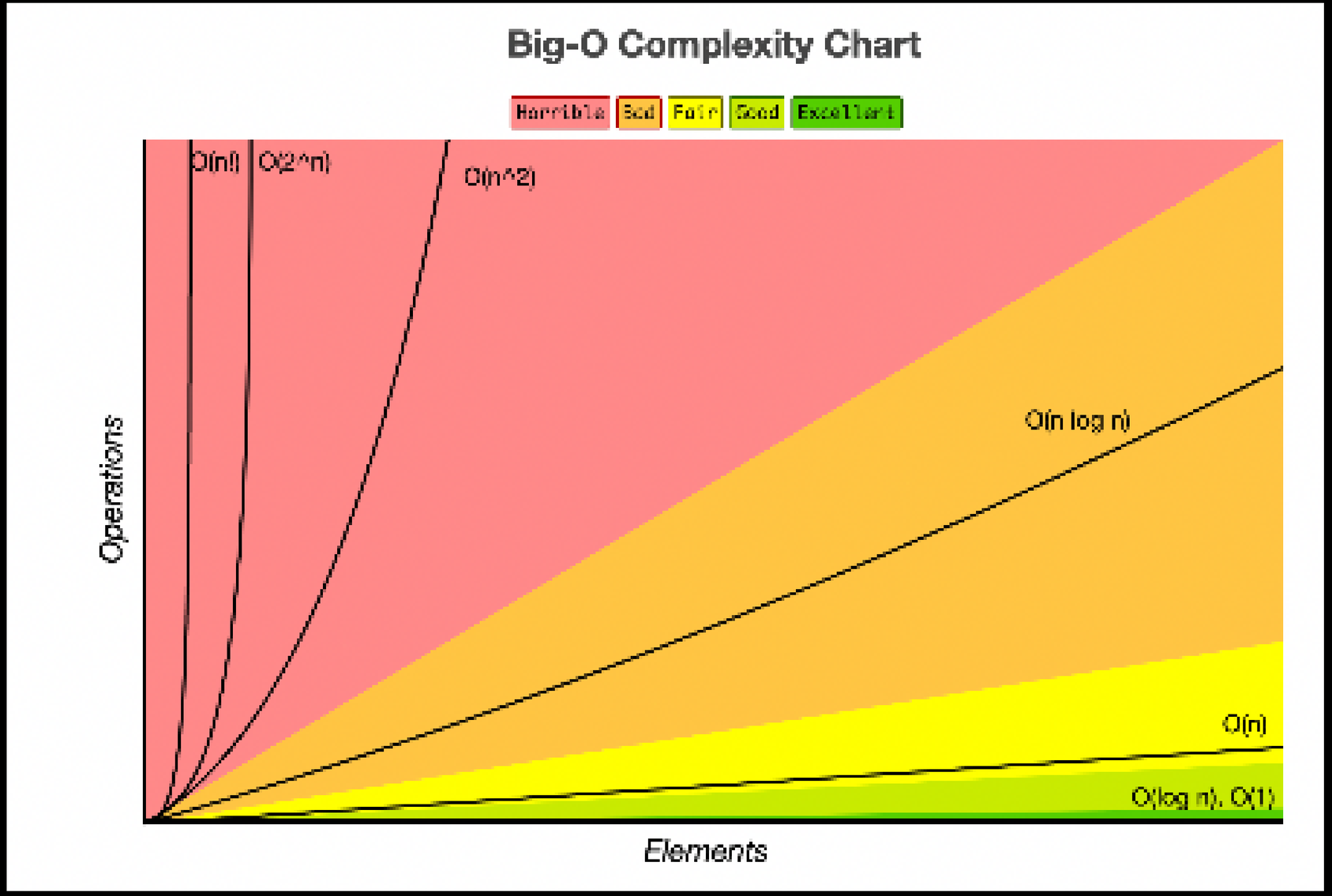
O(N): Execution time is about **100** time units

O(N * log N): Execution time is about **664** time units

$O(N^2)$ Execution time is about **10000** time units

$O(2^N)$: Execution time is about $\mathbf{1.2 \cdot 10^{30}}$ time units

O(N!) Execution time is about $\mathbf{9.33 \cdot 10^{157}}$ time units


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

A Quick Note on Space Complexity

Time is not the only metric that code is reviewed on

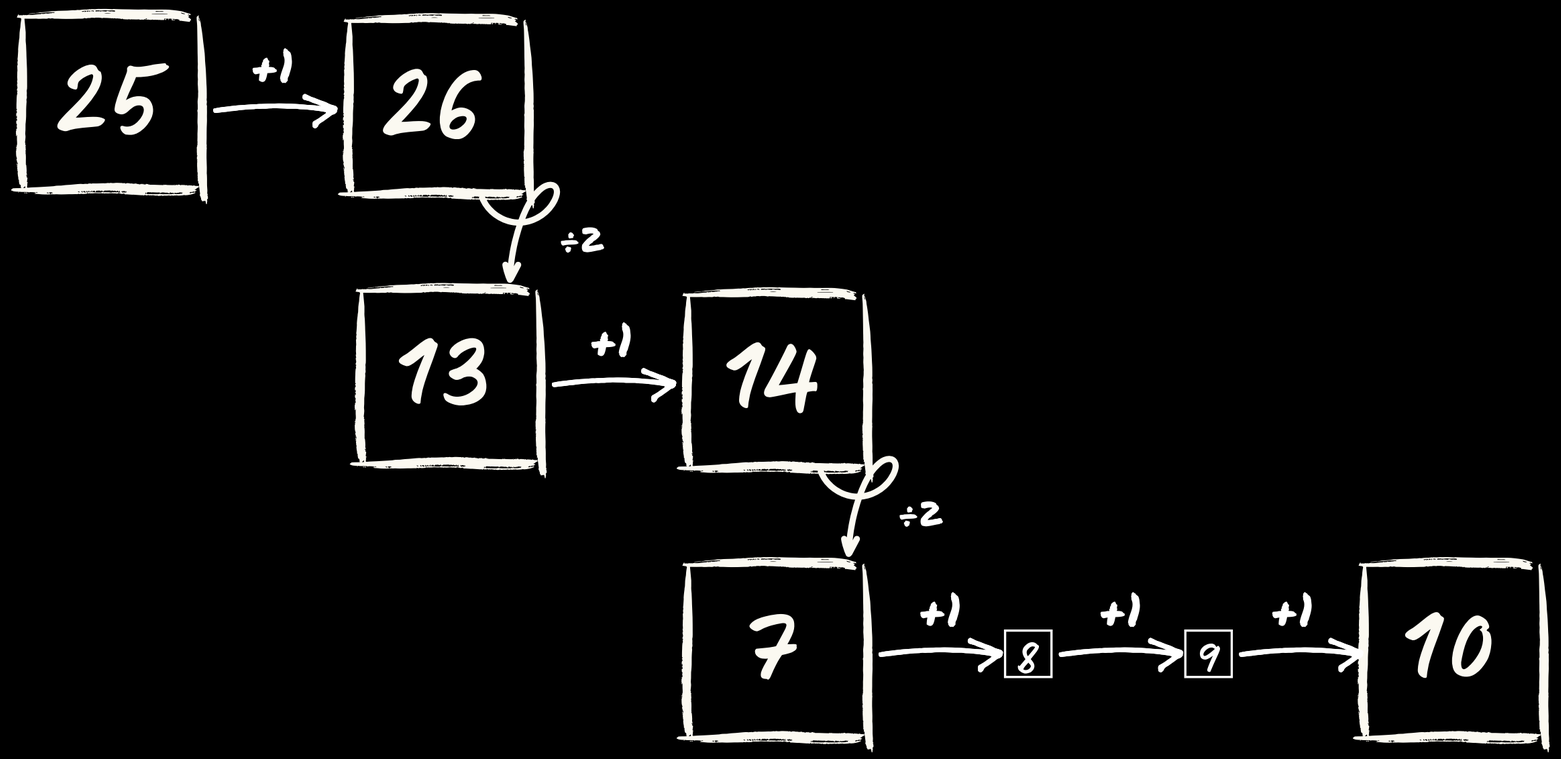
It is also important to analyze how much space in the computer's memory the code uses to store all needed data (such as integers, strings, arrays, 2D arrays, and all other data types and structures)

Space Complexity is also measured using big-O, but instead of being proportional to execution time, it is proportional to memory space

Space complexity is far less important than time complexity today, computer memories have exponentially grown, but CPU execution times are bottlenecked (a CPU can only do one command at a time). It is not important for any problems we will do.

Last problem...

Last week, we had a greedy solution that has two moves.
These moves take different amounts of steps to get to the solution.



Last problem...

Let’s say we start at 7, and we make our way up to 10.

What would the time complexity of this algorithm be?

O(1) = O(yeah)

O(logn) = O(nice)

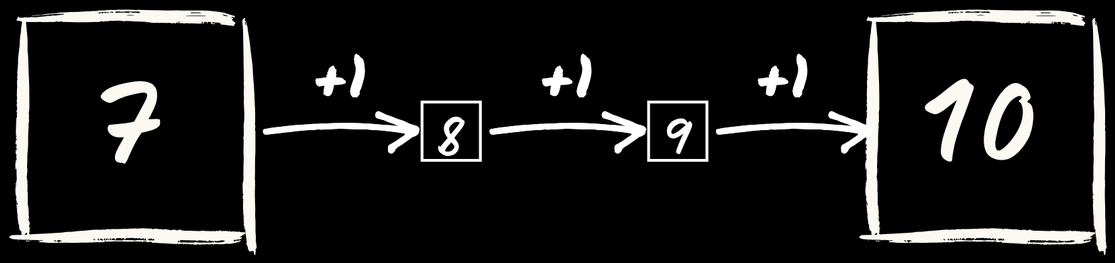
O(n) = O(k)

O(n²) = O(my)

O(2^n) = O(no)

O(n!) = O(mg)

O(n^n) = O(sh*t!)



quantcdev

Last problem...

It would actually be O(n). Like a linear graph,
I only move up only one at a time. It’s not really
changing how much it increments.

O(1) = O(yeah)

O(logn) = O(nice)

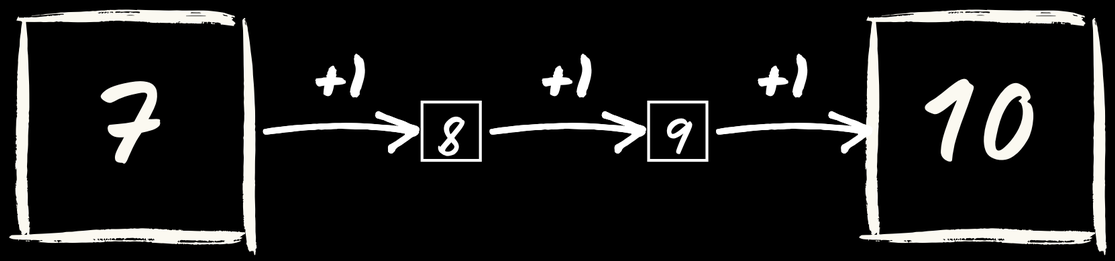
O(n) = O(k)

O(n²) = O(my)

O(2^n) = O(no)

O(n!) = O(mg)

O(n^n) = O(sh*t!)



quantcdev

Last problem...

Now we start at 28 and make our way down to 7. What would this be?

O(1) = O(yeah)

O(logn) = O(nice)

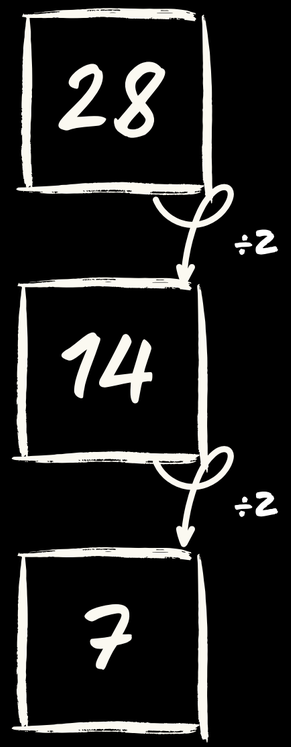
O(n) = O(k)

O(n^2) = O(my)

O(2^n) = O(no)

O(n!) = O(mg)

O(n^n) = O(sh*t!)



quantcdev

Last problem...

It would be O(logn), because the logarithm is asking “how many divisions should I do to get to my solution?”

O(1) = O(yeah)

O(logn) = O(nice)

O(n) = O(k)

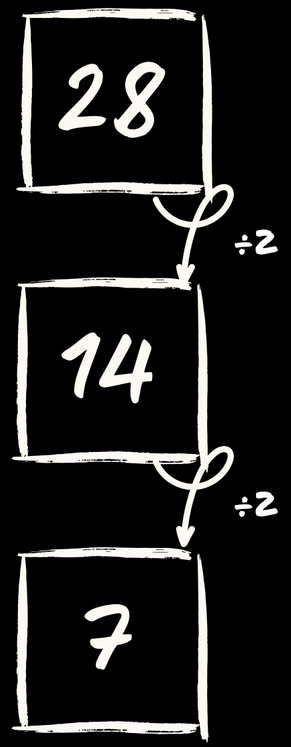
O(n²) = O(my)

O(2^n) = O(no)

O(n!) = O(mg)

O(n^n) = O(sh*t!)

quantic dev



LeetCode #121: Best Time to Buy and Sell Stock

121. Best Time to Buy and Sell Stock

Easy | Topics | Companies

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i^th` day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Test Case

We begin with no stock. We want to buy at one point, and sell at a later point for maximum profit

> You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

> Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Test Case

Goal is to identify the buy-point and the sell-point such that

1. Profit (sell - buy) is maximized
2. Sell-point is later than the buy-point (we must first buy the stock to sell it

> You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

> Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Test Case

Profit = 5 - 7 = -2



Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Test Case

Profit = 3 - 7 = -4



Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Test Case

Profit = 3 - 1 = 2



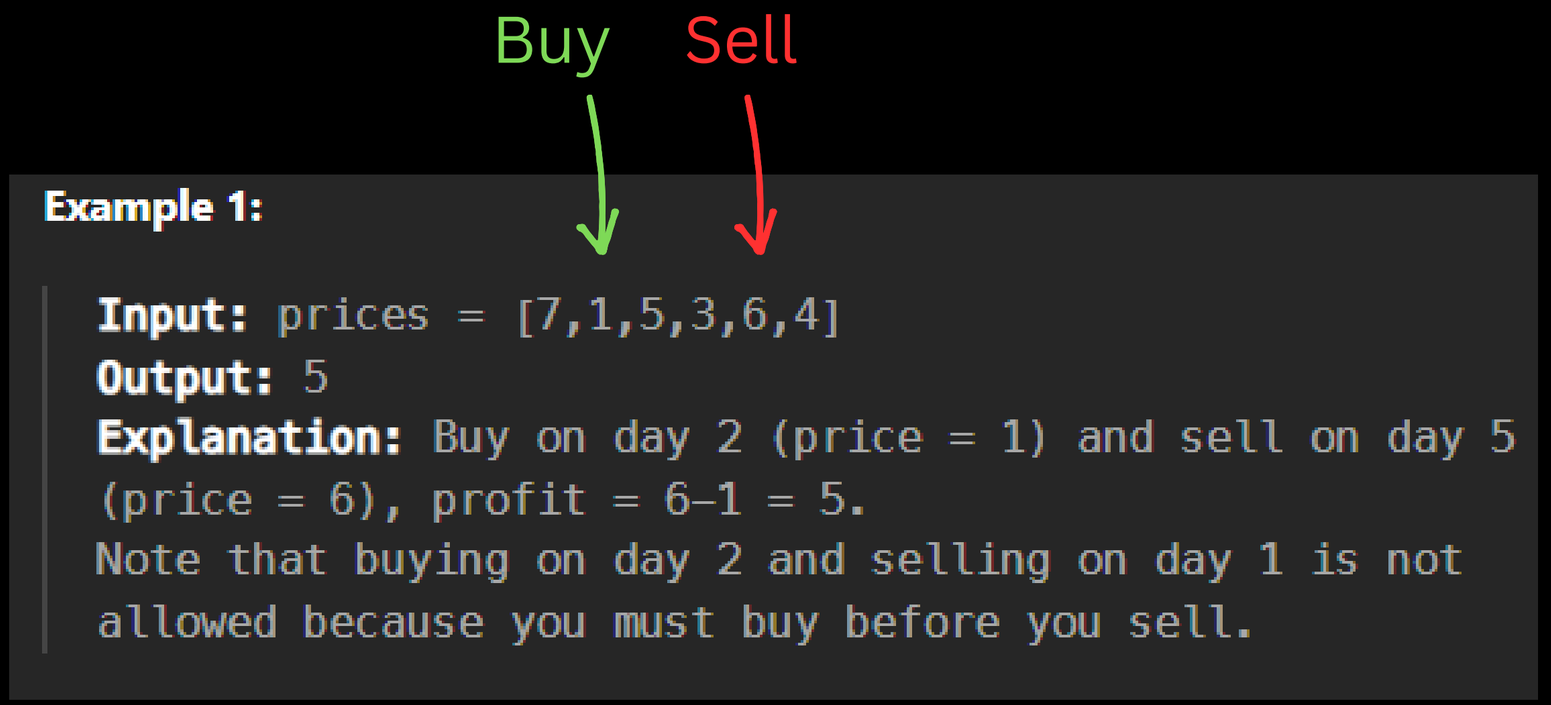
Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Test Case

Profit = 6 - 1 = 5

5 is the maximum possible profit, so we return it



Now you try it!!!

And, do NOT worry about time complexity for now. Just solve the problem using as much time, space, and loops as you need. A “Brute-Force” Solution is fine.

Solution - Brute Force

In any given array, there exist a finite number of buy and sell points

We can simply test them all and keep ofrecord the maximum of all the profit values

Solution - Brute Force

Going back to the test case:

Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

We will first identify all possible buy/sell combinations

Solution - Brute Force

Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

If we buy at

- 7: We can sell at 1,5,3,6,4 (5 pairs)
- 1: We can sell at 5,3,6,4 (4 pairs)
- 5: We can sell at 3,6,4 (3 pairs)
- 3: We can sell at 6,4 (2 pairs)
- 6: We can sell at 4 (1 pair)

Solution - Brute Force

Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

For an input array of N=6, we need to check:

5+4+3+2+1 = 15 buy/sell pairs

Solution - Brute Force Implementation

So, with only 15 combinations, we write code that checks all combinations. This can be done with a nested for loop

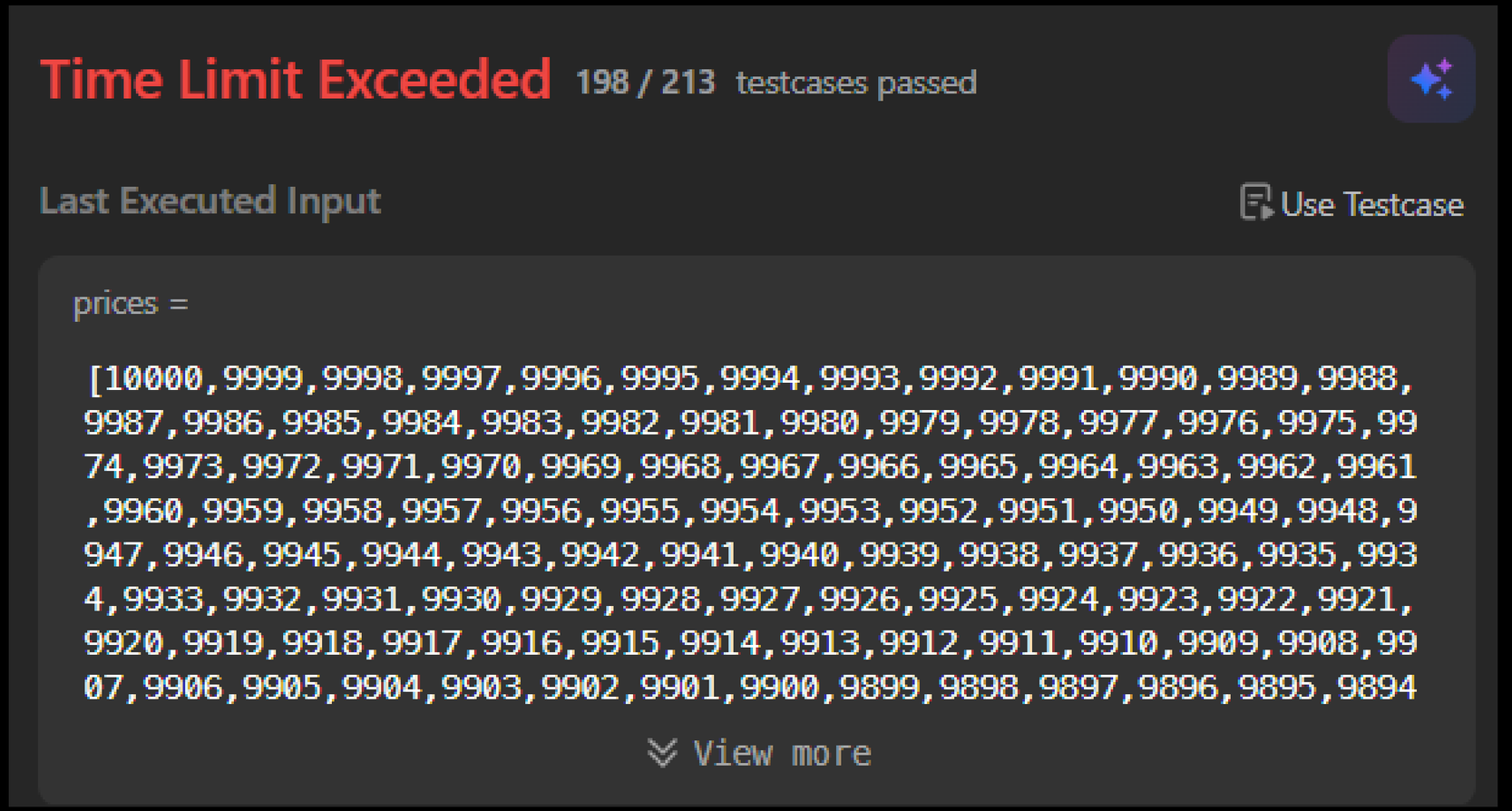
Solution – Python

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        best = 0
        n = len(prices)
        for i in range(n):                  # buy day
            for j in range(i + 1, n):       # sell day (must come after buy)
                profit = prices[j] - prices[i]
                if profit > best:
                    best = profit
        return best
```

Solution - Java

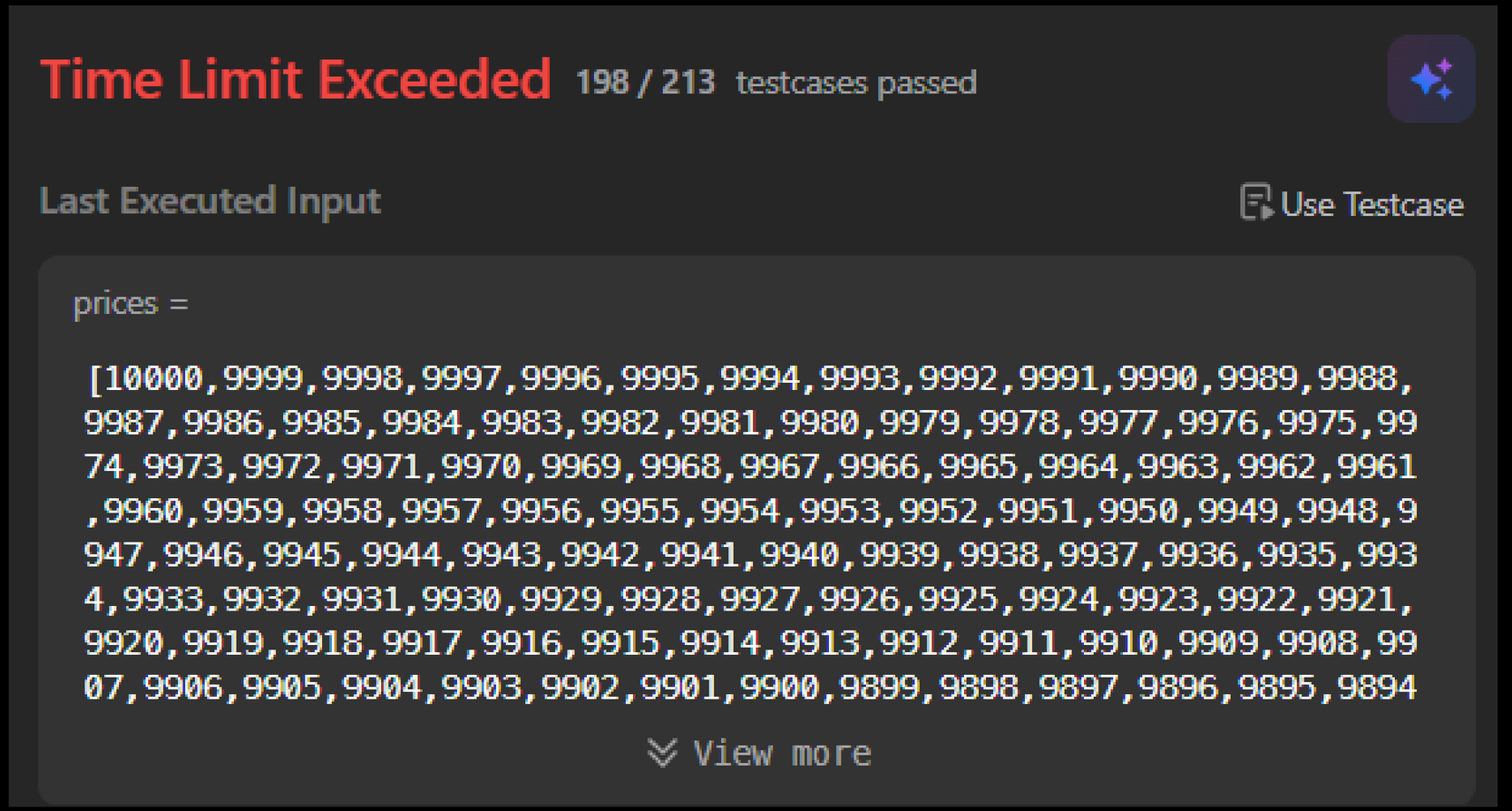
```java
class Solution {
    public int maxProfit(int[] prices) {
        int best = 0;
        int n = prices.length;
        for (int i = 0; i < n; i++) {           // buy day
            for (int j = i + 1; j < n; j++) {   // sell day (must come after buy)
                int profit = prices[j] - prices[i];
                if (profit > best) {
                    best = profit;
                }
            }
        }
        return best;
    }
}
```

Solution - Wait a minute...



Our solution worked perfectly for 198 test cases, but “Time Limit Exceeded” for the remaining 15

Solution - Wait a minute...



This is when we need to analyze our time complexity.

Solution - Time complexity

Remember, we found that:

When N=6, total pairs = 5+4+3+2+1 = 15

This pattern actually holds for all values of N.

When N=10, total pairs = 9+8+7+6+5+4+3+2+1 = 45

Our input only went up by four, but our pairs went up by 3x!

Solution - Time complexity

Let’s extrapolate:

For N inputs, our total pairs = N-1 + N-2 + N-3 + … + 1

This can be mathematically proven as equal to $\frac{n(n-1)}{2}$

Which is also equal to: $\text{Sum} = \frac{n^2 - n}{2}$

For example, when N=6, $(6^2 - 6)/2 = (36-6)/2 = 30/2 = 15$

Solution - Time complexity

$$\text{Sum} = \frac{n^2 - n}{2}$$

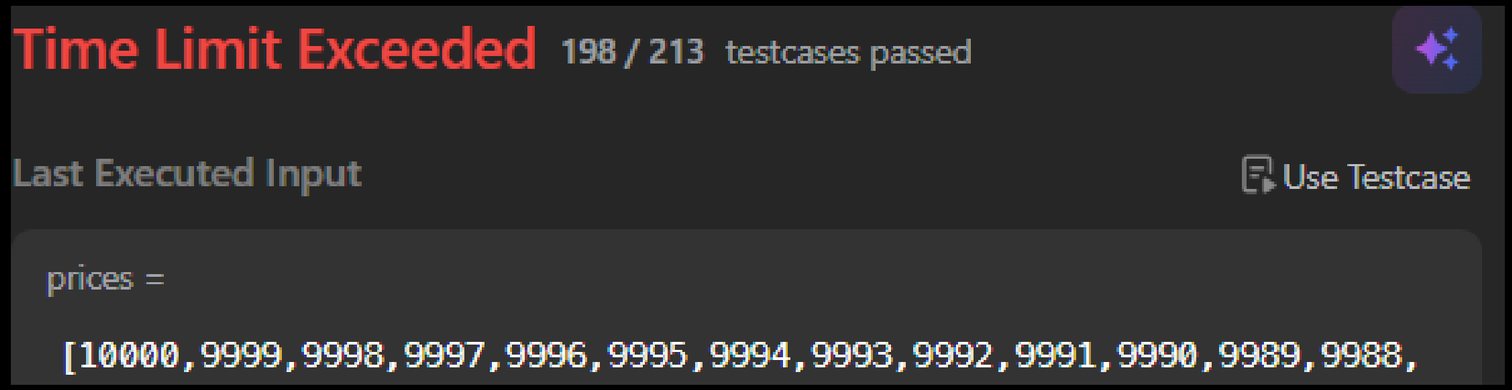
Time complexity is always equal to the fastest growing term in the formula. In this case, the fastest growing term is $n^{2.}$

We ignore the (-n) and the divide by two, because those are far less impactful than n^2

Our solution's time complexity is $O(N^2)$

Solution – Examining our failure

Our solution's time complexity is $O(N^2)$



This input has 10000 inputs.

At $O(N^2)$, we must do about $10000^2 = 100000000$ pairs

Typically, 2000 inputs is the maximum $O(N^2)$ can handle

Problem - Run it back

121. Best Time to Buy and Sell Stock

Easy | Topics | Companies

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i^th` day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

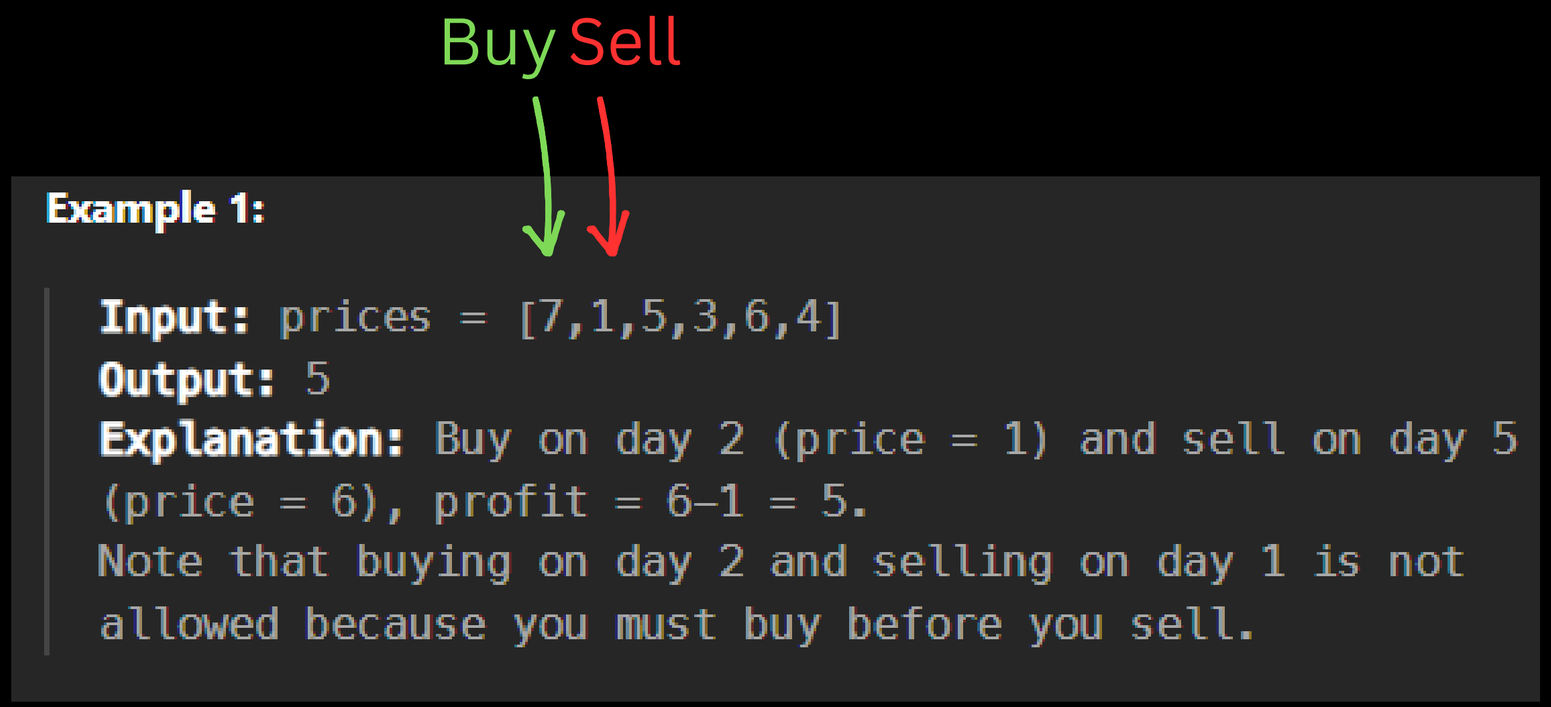
Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Solve the same exact problem, but find a faster solution

Hint: We don't need to check every single pair. Figure out how you can avoid checking unnecessary pairs.

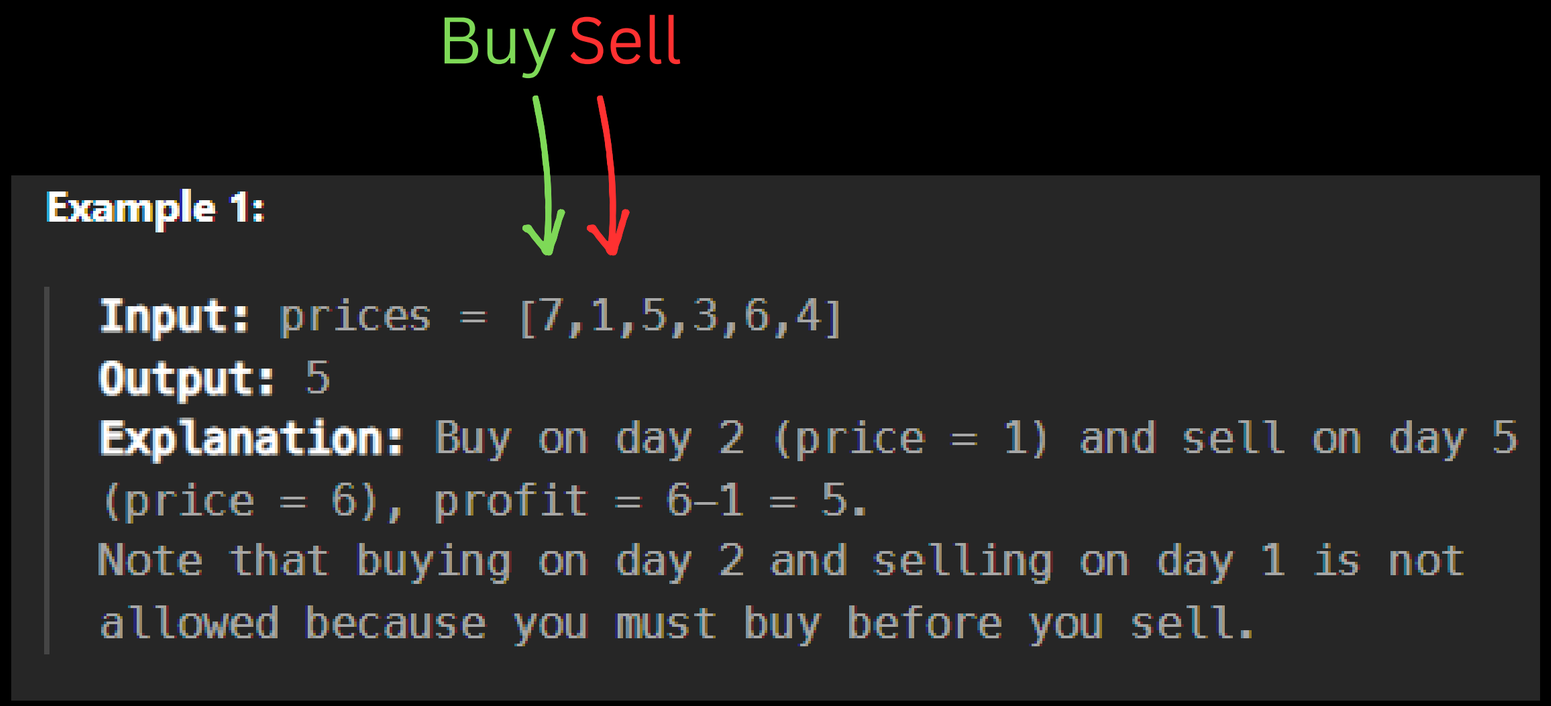
Solution - How can we speed this up?

Here is the test case from before, set to the first possible buy/sell pair.



Solution - How can we speed this up?

Let’s actually remove the buy pointer, and focus only on the sell



Solution - How can we speed this up?

Let’s actually remove the buy pointer, and focus only on the sell

For this sell value, what is the best possible buy value?

(Of course, it is 7, there is only one possible buy value)



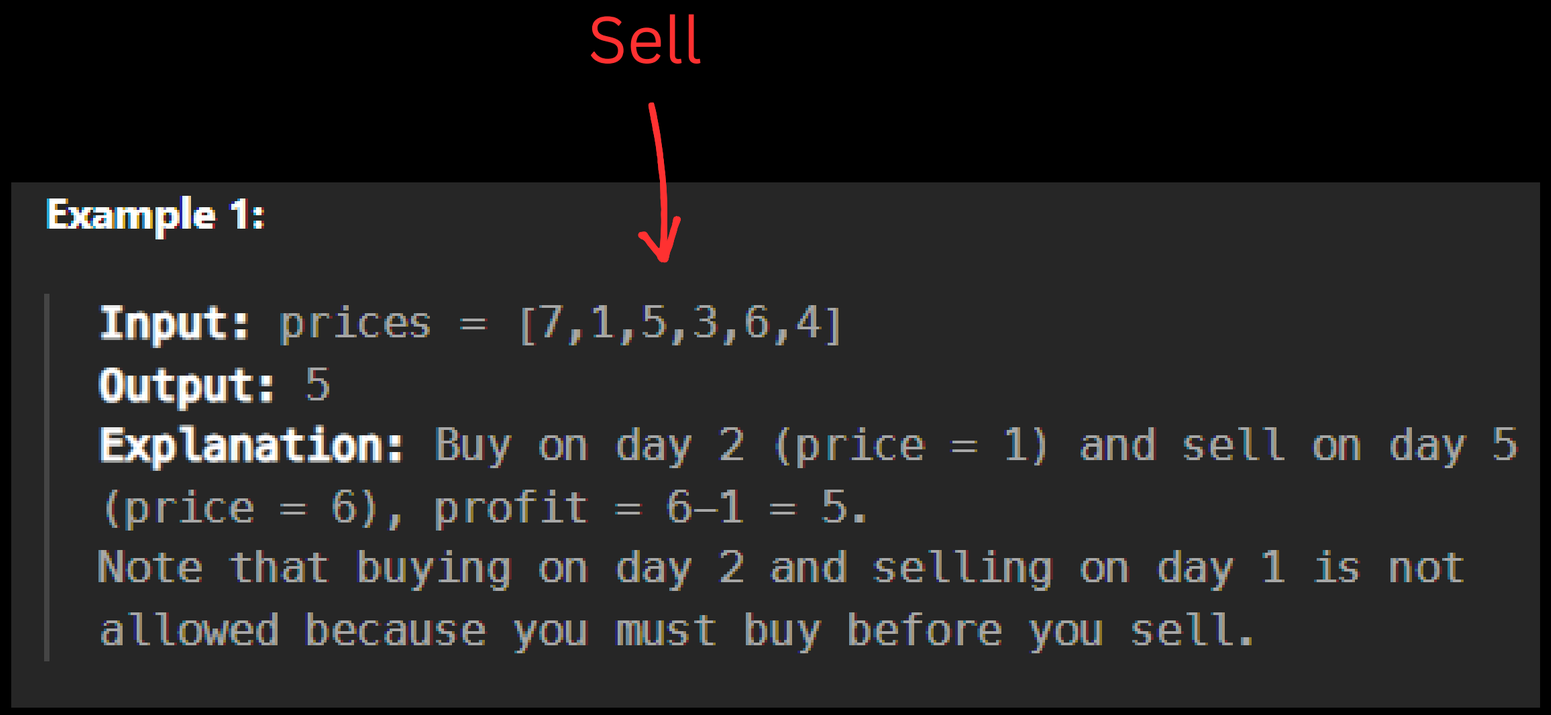
Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Solution - How can we speed this up?

For this sell value, what is the best possible buy value?

It is 1, because 5 - 1 = 4 is better profit than 5 - 7 = -2



Solution - How can we speed this up?

For this sell value, what is the best possible buy value?

It is again 1.



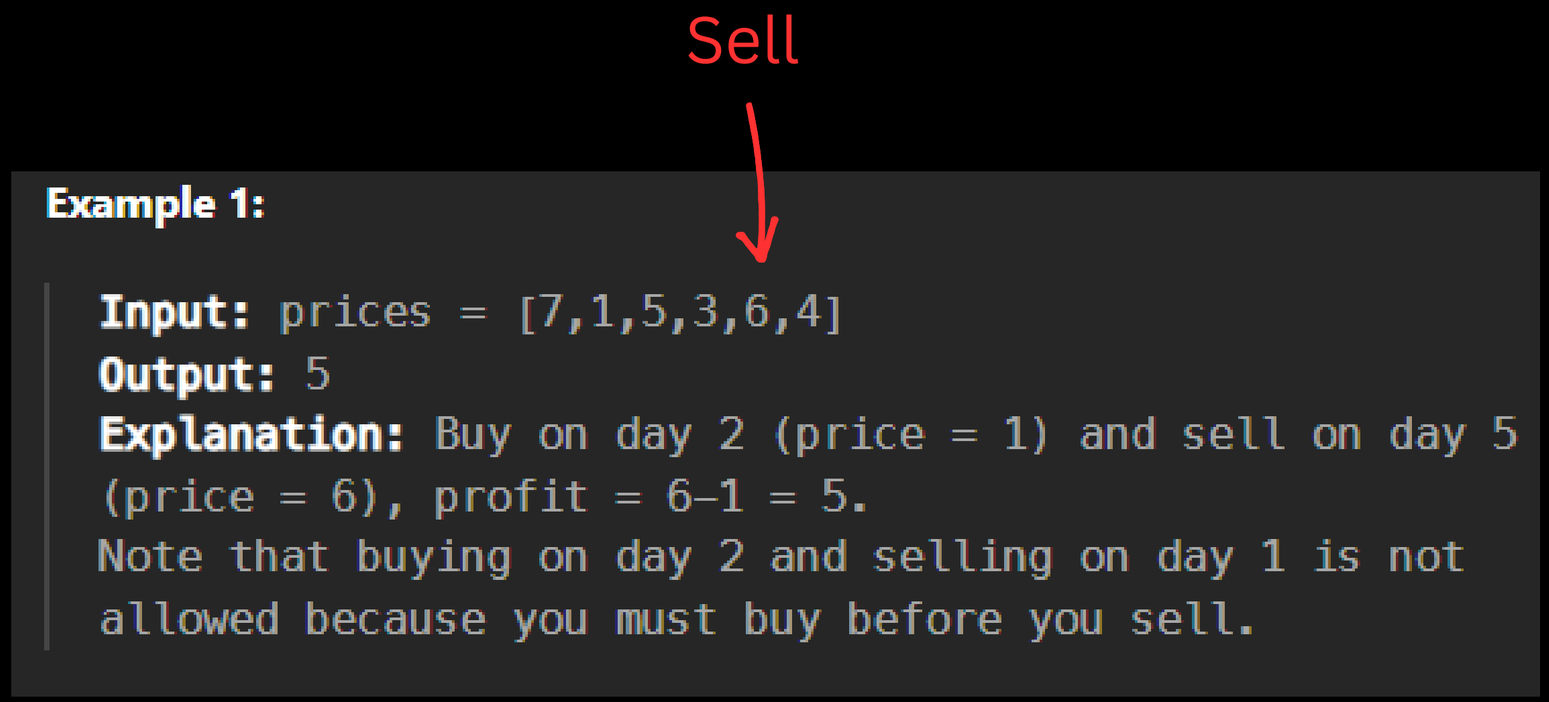
Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Solution - How can we speed this up?

For this sell value, what is the best possible buy value?

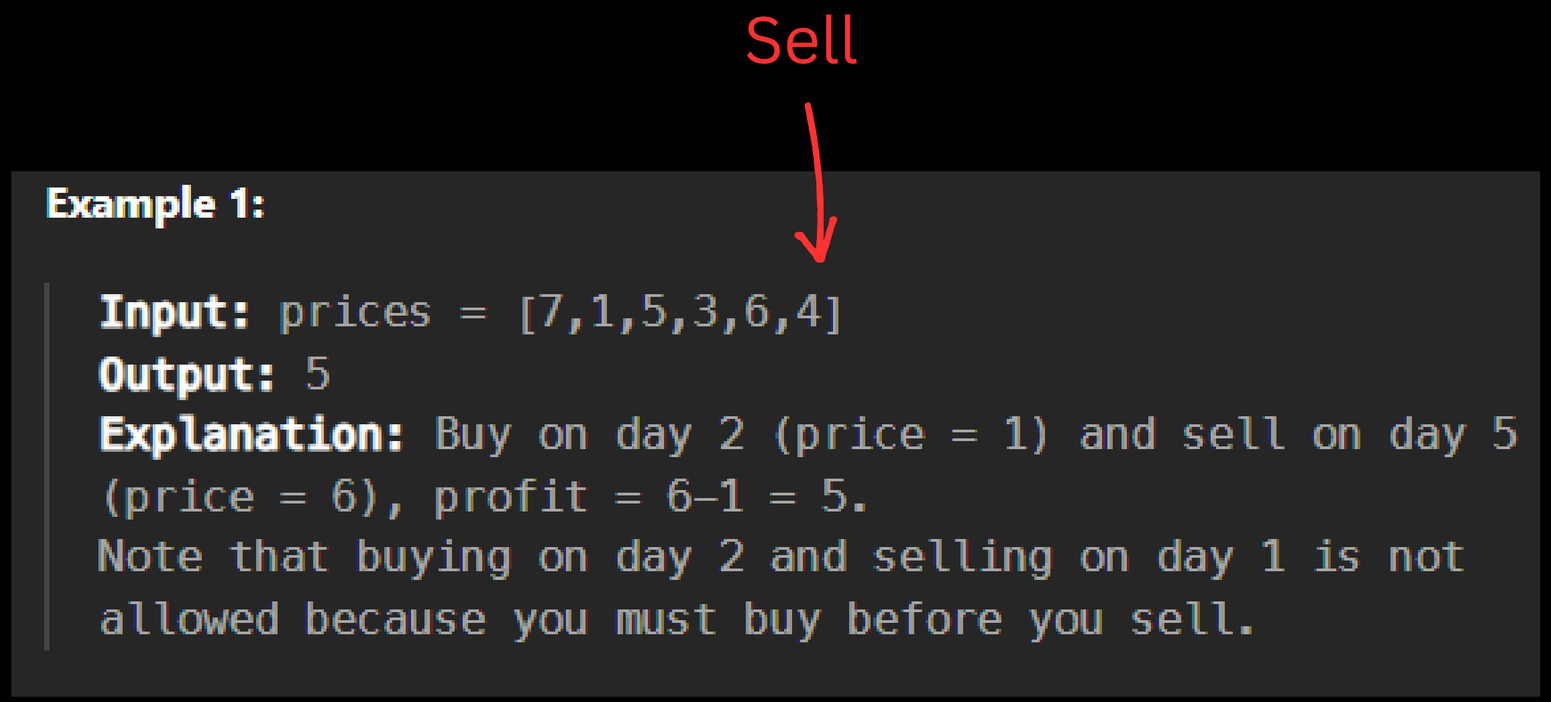
It is again 1.



Solution - How can we speed this up?

For this sell value, what is the best possible buy value?

It is again 1.



Solution - How can we speed this up?

What do we notice?

For each sell value, the best buy value is just the minimum of all values before it!



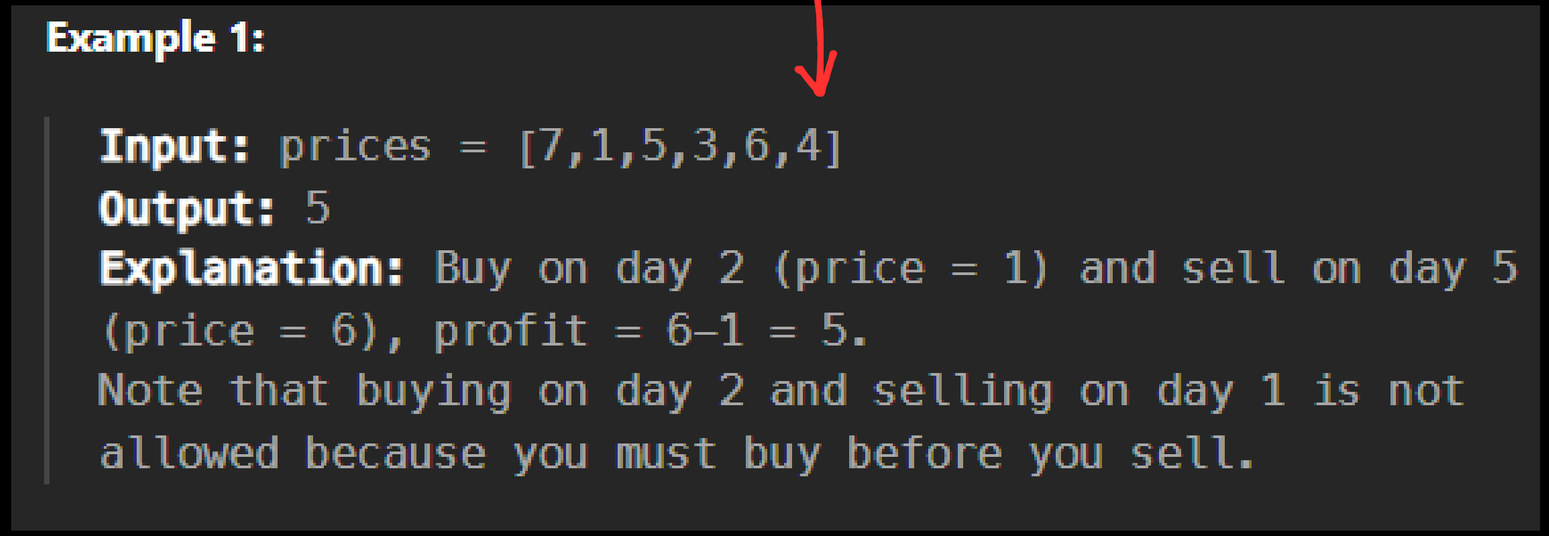
Example 1:

> **Input:** prices = [7,1,5,3,6,4]
> **Output:** 5
> **Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
> Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Solution - How can we speed this up?

This means, instead of testing every possible buy/sell pair, we only need to find the best sell value. Then, profit is simply the sell value - the minimum of all values before it

Sell



Solution - Algorithm

Iterate through the array one time

- Keep track of the
  - Minimum buy value so far
  - Maximum profit so far
- For each value k we encounter
  - If k is less than the current minimum buy, it becomes the new minimum buy
  - If (k - minimum buy) is greater than maximum profit, it becomes the newmaximum profit

Return maximum profit (may be zero)

Solution – Python

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        min_price = float('inf')   # cheapest buy seen so far
        best = 0                   # best profit seen so far
        for price in prices:
            if price < min_price:
                min_price = price              # found a better day to buy
            elif price - min_price > best:
                best = price - min_price       # found a better day to sell
        return best
```

Solution - Java

```java
class Solution {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE;  // cheapest buy seen so far
        int best = 0;                      // best profit seen so far
        for (int price : prices) {
            if (price < minPrice) {
                minPrice = price;          // found a better day to buy
            } else if (price - minPrice > best) {
                best = price - minPrice;   // found a better day to sell
            }
        }
        return best;
    }
}
```

Solution - Time complexity

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        min_price = float('inf')   # cheapest buy seen so far
        best = 0                   # best profit seen so far
        for price in prices:
            if price < min_price:
                min_price = price              # found a better day to buy
            elif price - min_price > best:
                best = price - min_price       # found a better day to sell
        return best
```

Here, we only scan through the array one time. This is because we only check the sell values. There are ~N sell values, so this is O(N)

Time complexity shortcut

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        min_price = float('inf')   # cheapest buy seen so far
        best = 0                   # best profit seen so far
        for price in prices:
            if price < min_price:
                min_price = price              # found a better day to buy
            elif price - min_price > best:
                best = price - min_price       # found a better day to sell
        return best
```

A simpler way to calculate time complexity is to examine the total number of nested loops.

For example, this has one loop: O(N)

Time complexity shortcut

```java
class Solution {
    public int maxProfit(int[] prices) {
        int best = 0;
        int n = prices.length;
        for (int i = 0; i < n; i++) {           // buy day
            for (int j = i + 1; j < n; j++) {   // sell day (must come after buy)
                int profit = prices[j] - prices[i];
                if (profit > best) {
                    best = profit;
                }
            }
        }
        return best;
    }
}
```

Our previous solution has a loop inside a loop, and it is $O(N^2)$

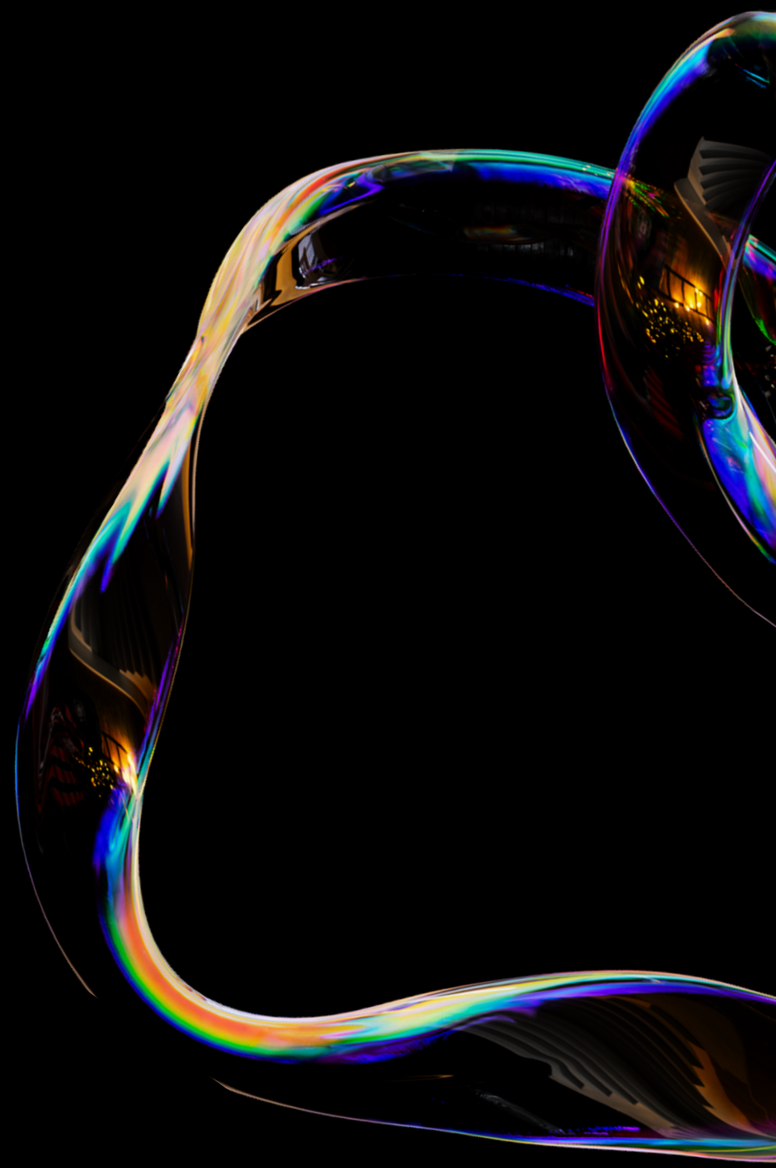
If it had a loop inside a loop inside a loop, it would be $O(N^3)$
(Please note however, this is only true for loops that increment linearly, such as i++. This does not work for i = i*3 for example.)

Hacker Devils

Presented by Sohum Sheth, Dean Richardson and Griffin Charlson

THANK YOU

for your time and attention

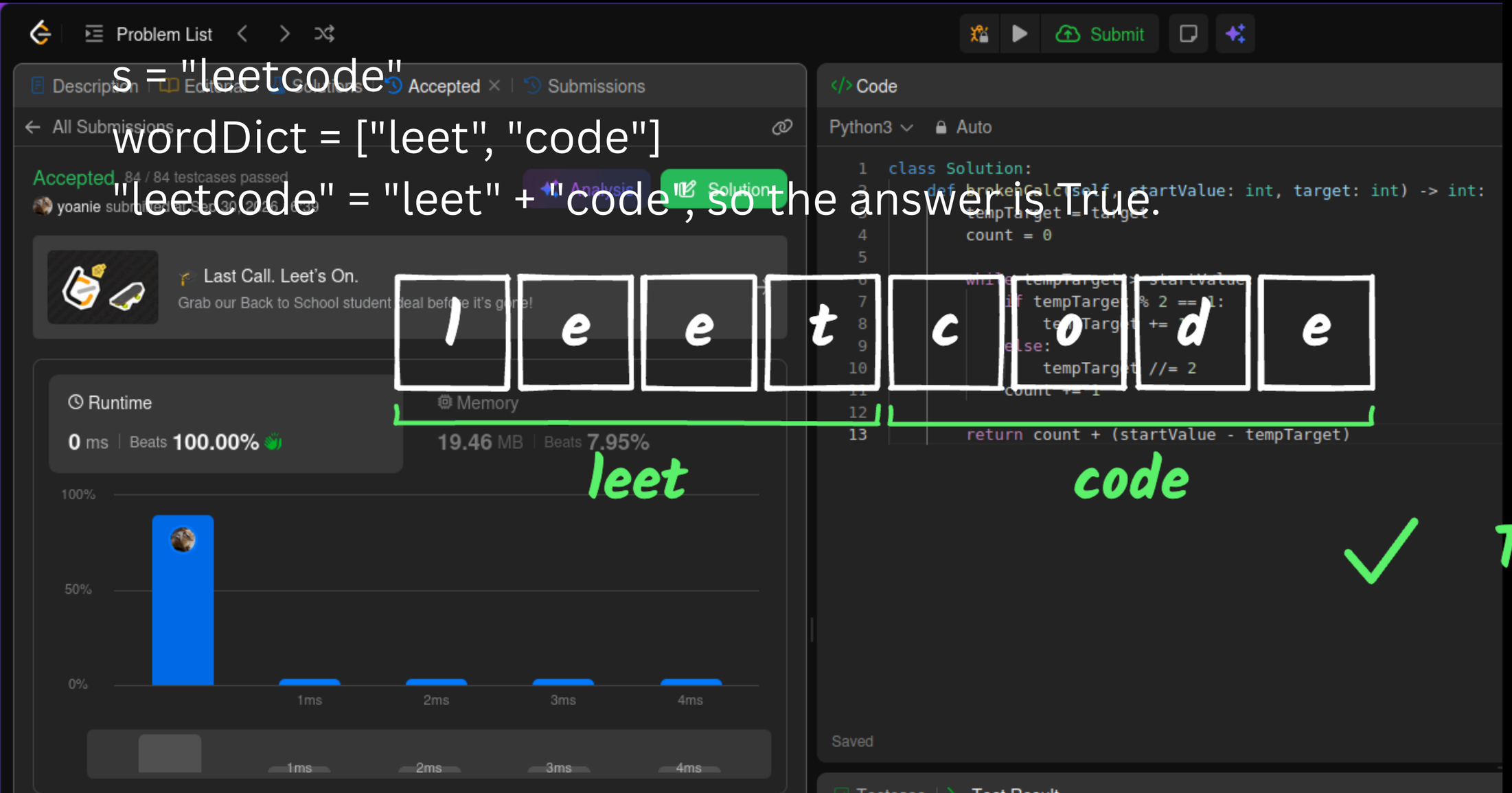


Example Test Case #1

s = "leetcode"

wordDict = ["leet", "code"]

"leetcode" = "leet" + "code", so the answer is True.

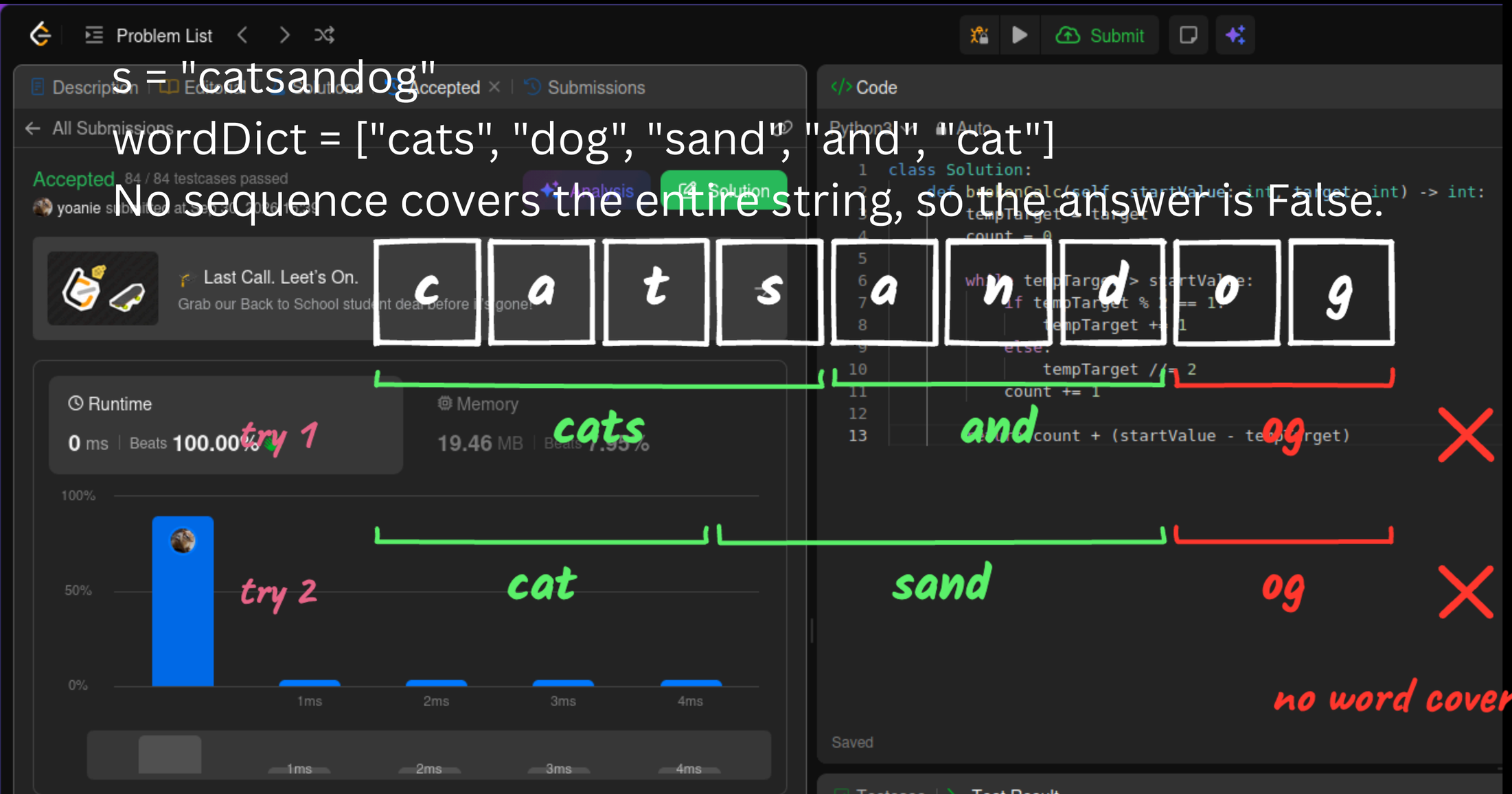


Example Test Case #2

s = "catsandog"

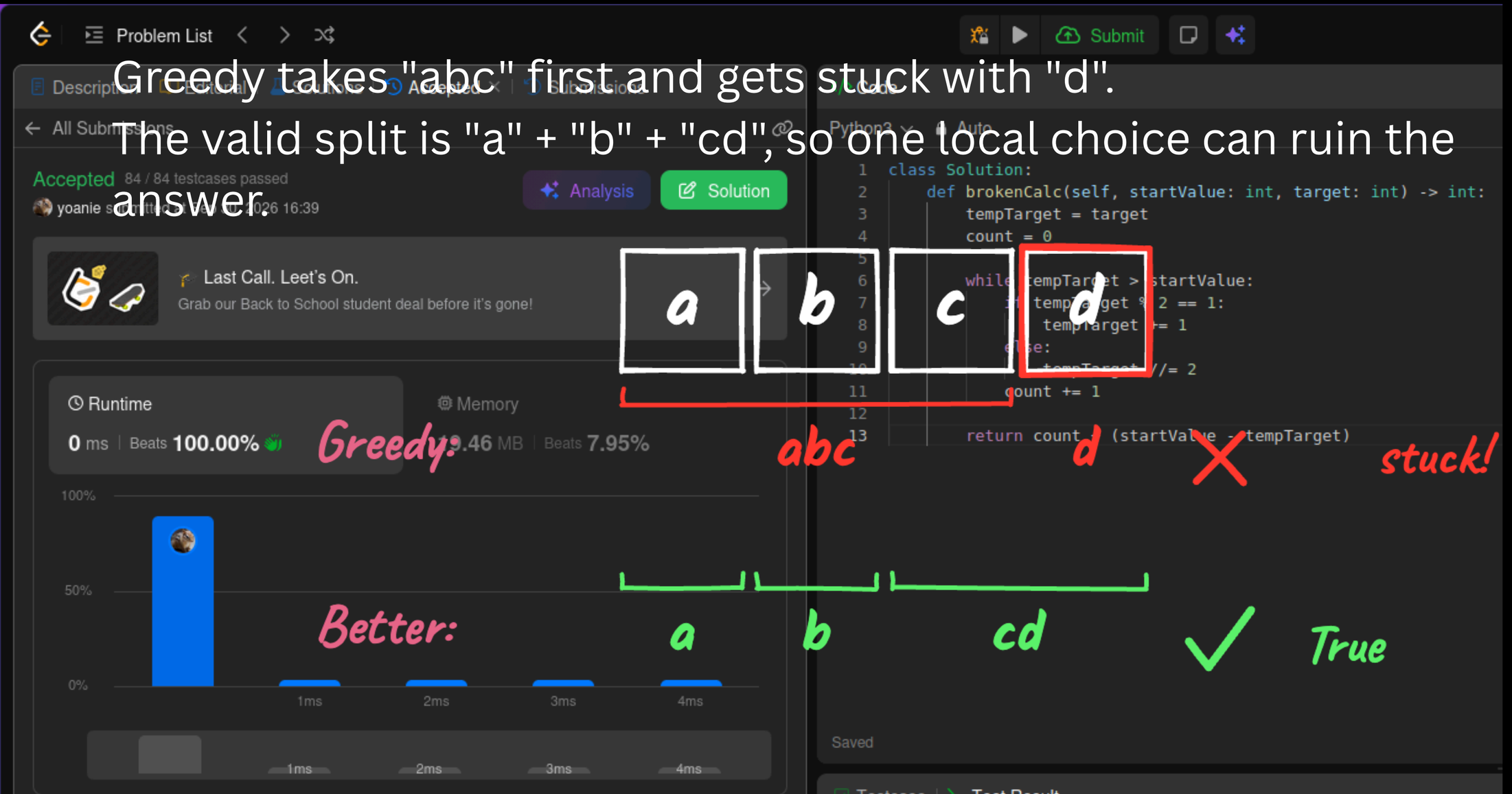
wordDict = ["cats", "dog", "sand", "and", "cat"]

No sequence covers the entire string, so the answer is False.



Can We Be Greedy Here?

Greedy takes "abc" first and gets stuck with "d".
The valid split is "a" + "b" + "cd", so one local choice can ruin the answer.



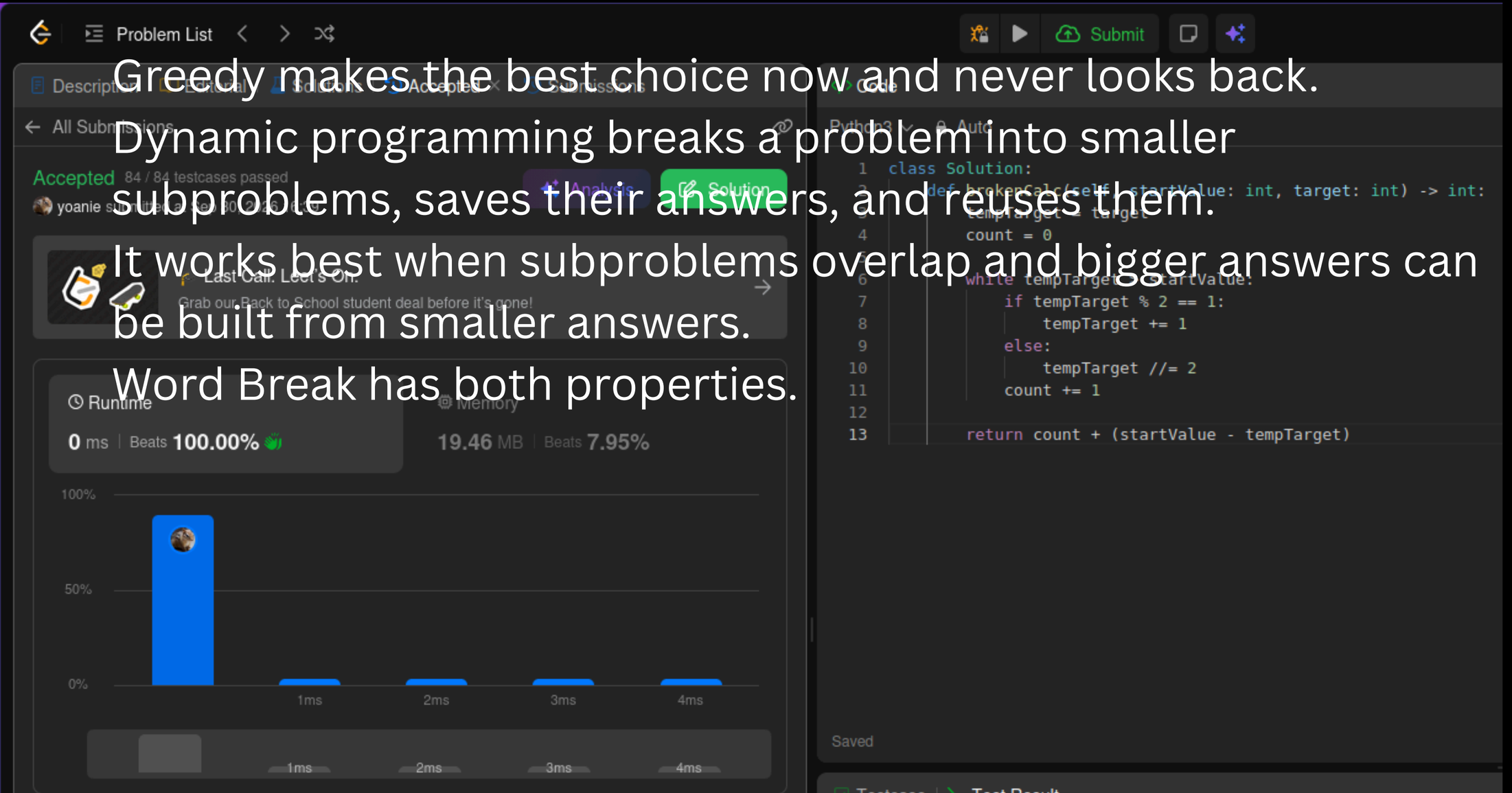
Dynamic Programming

Greedy makes the best choice now and never looks back.

Dynamic programming breaks a problem into smaller subproblems, saves their answers, and reuses them.

It works best when subproblems overlap and bigger answers can be built from smaller answers.

Word Break has both properties.

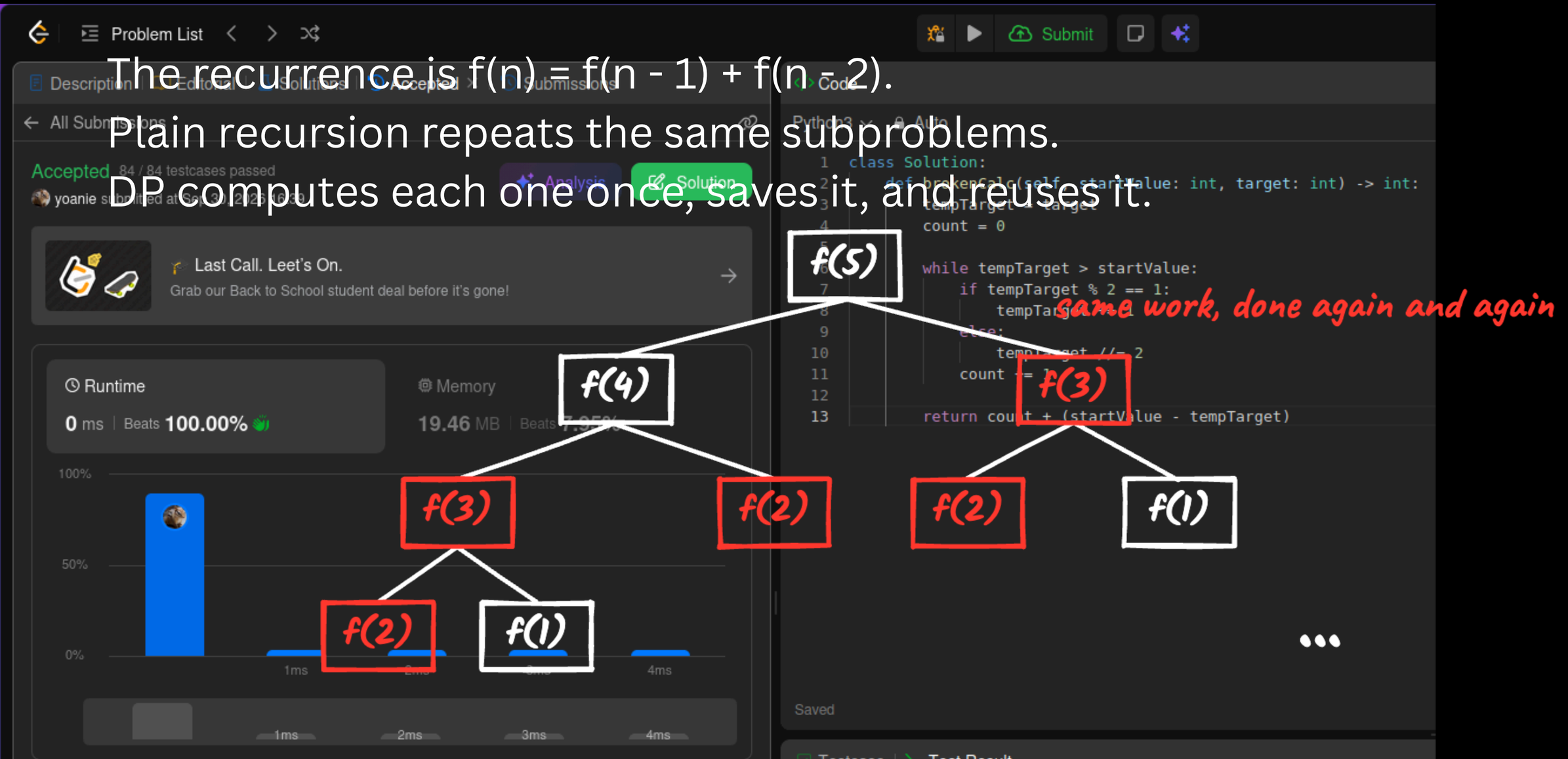


A Familiar Example: Fibonacci

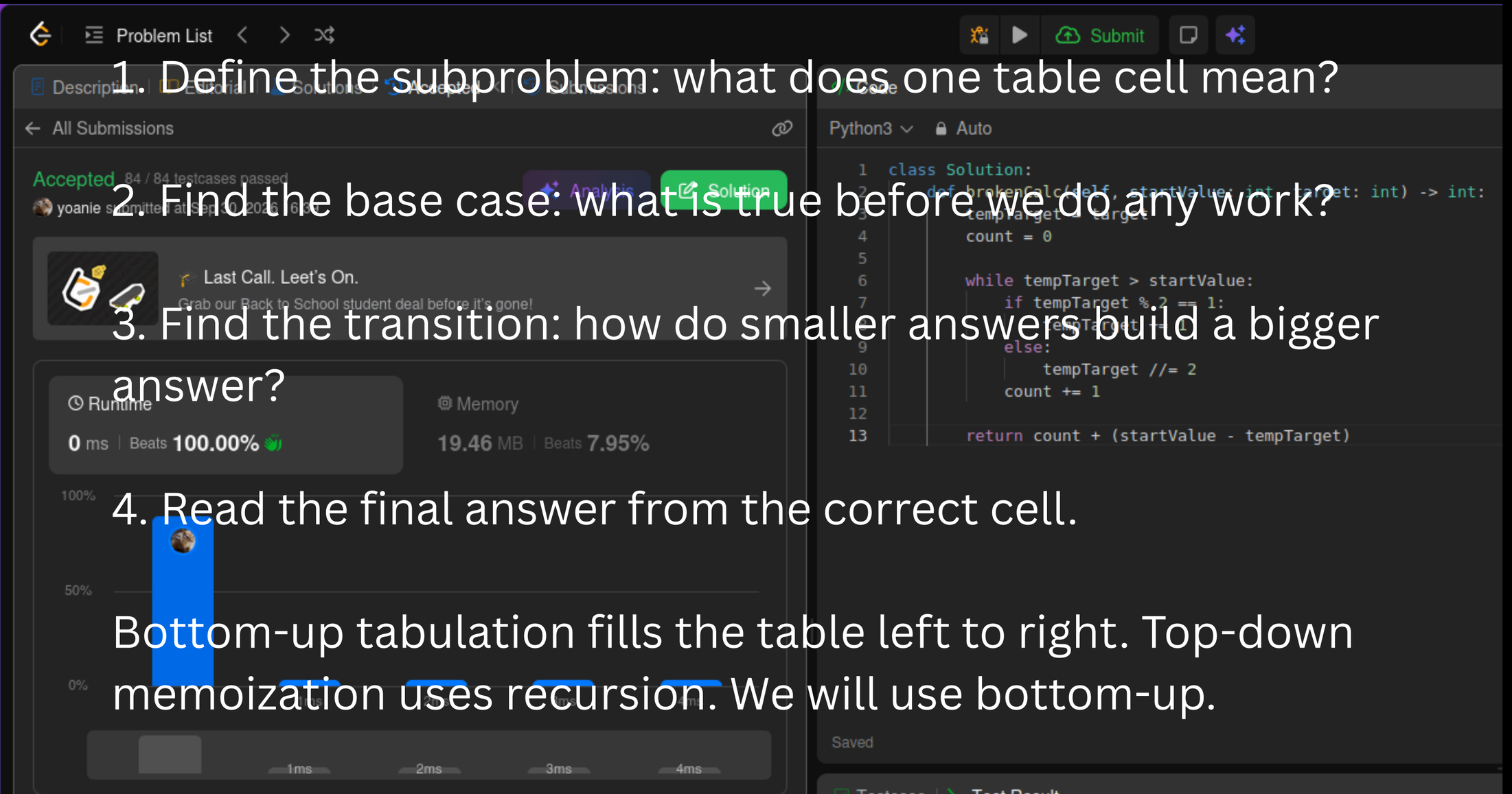
The recurrence is f(n) = f(n - 1) + f(n - 2).

Plain recursion repeats the same subproblems.

DP computes each one once, saves it, and reuses it.



The DP Recipe



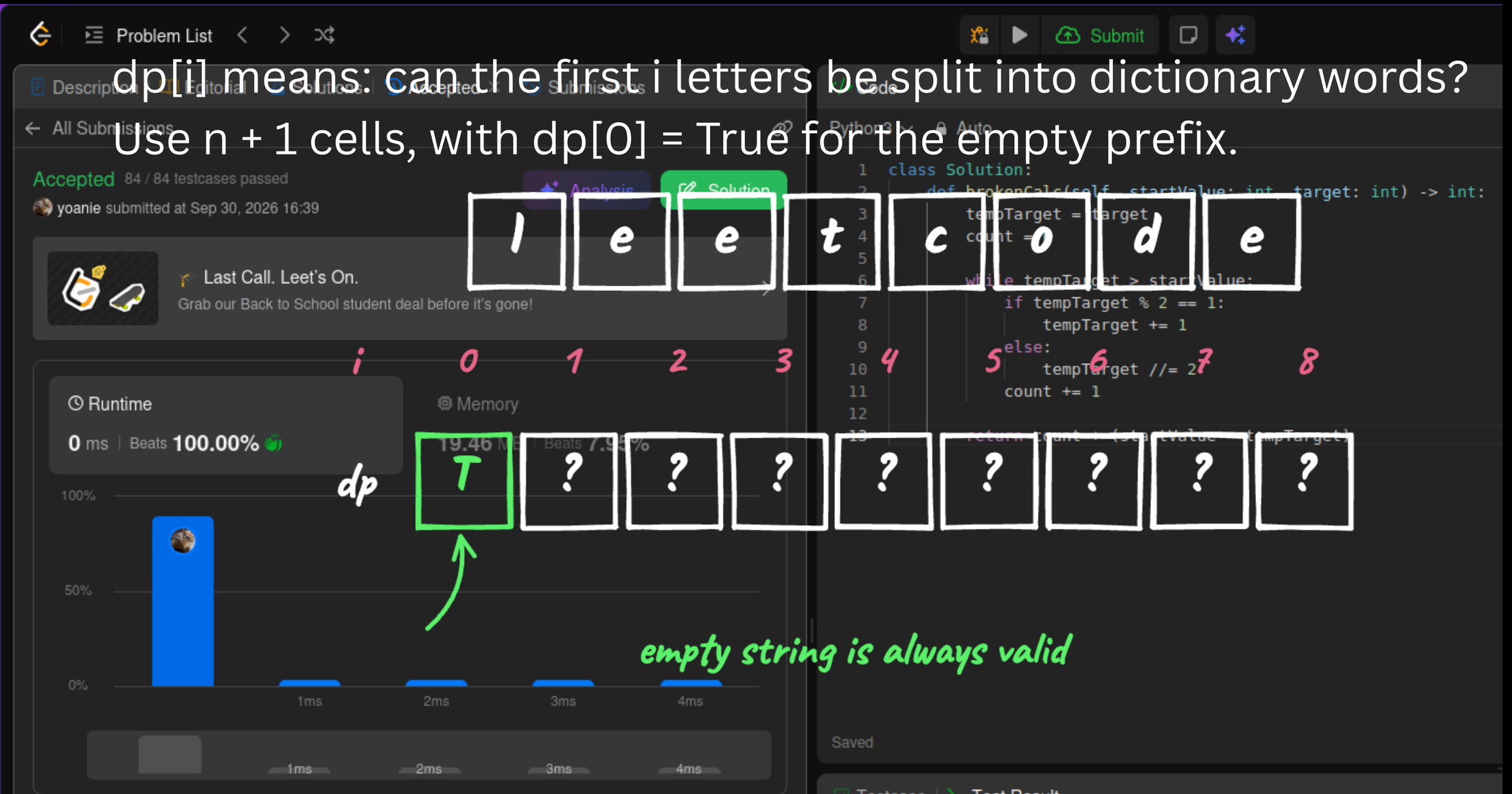
1. Define the subproblem: what does one table cell mean?
2. Find the base case: what is true before we do any work?
3. Find the transition: how do smaller answers build a bigger answer?
4. Read the final answer from the correct cell.

Bottom-up tabulation fills the table left to right. Top-down memoization uses recursion. We will use bottom-up.

Setup: What Does dp[i] Mean?

dp[i] means: can the first i letters be split into dictionary words?

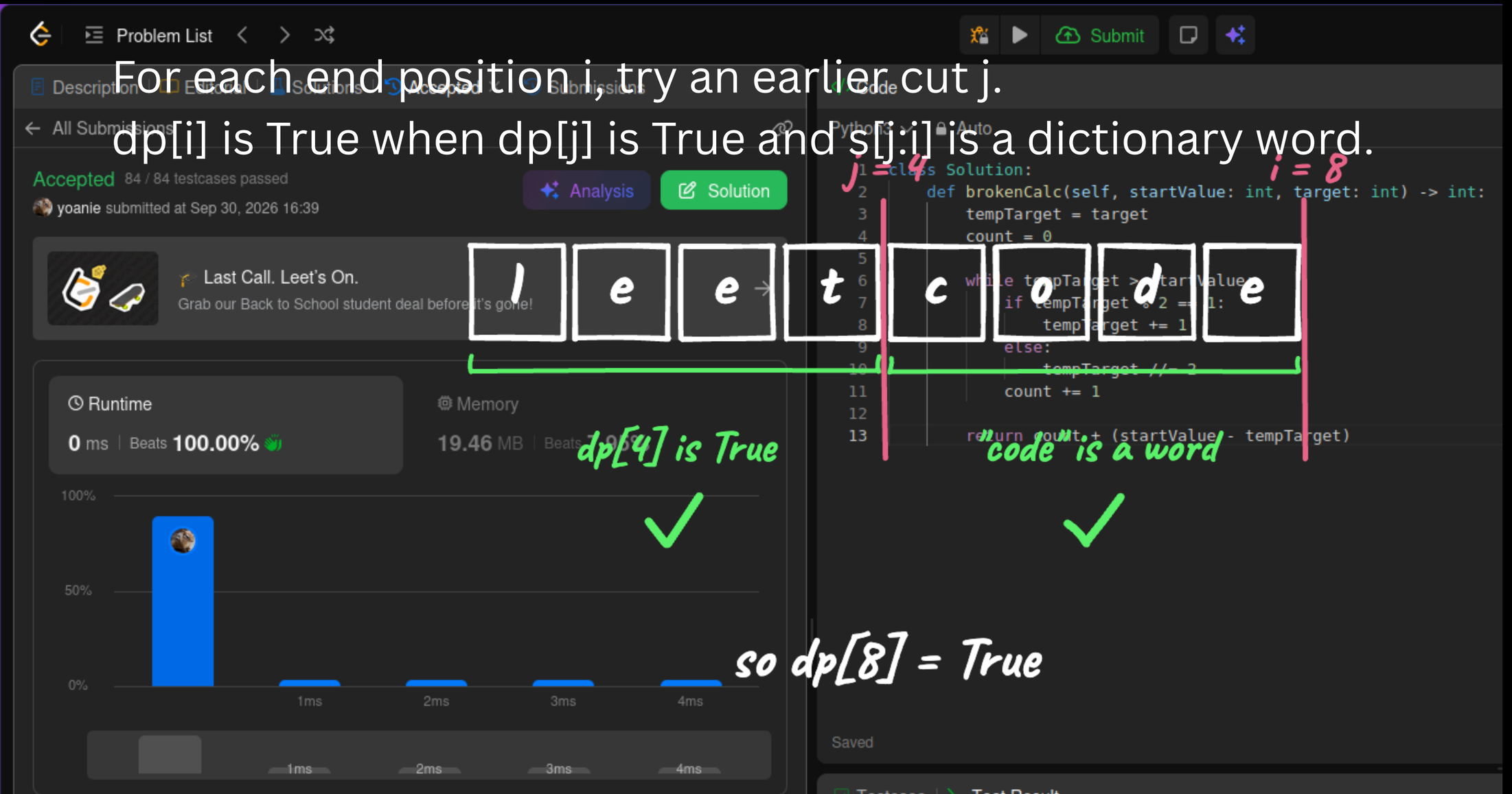
Use n + 1 cells, with dp[0] = True for the empty prefix.



Transition: How dp[i] Becomes True

For each end position i, try an earlier cut j.

dp[i] is True when dp[j] is True and s[j:i] is a dictionary word.

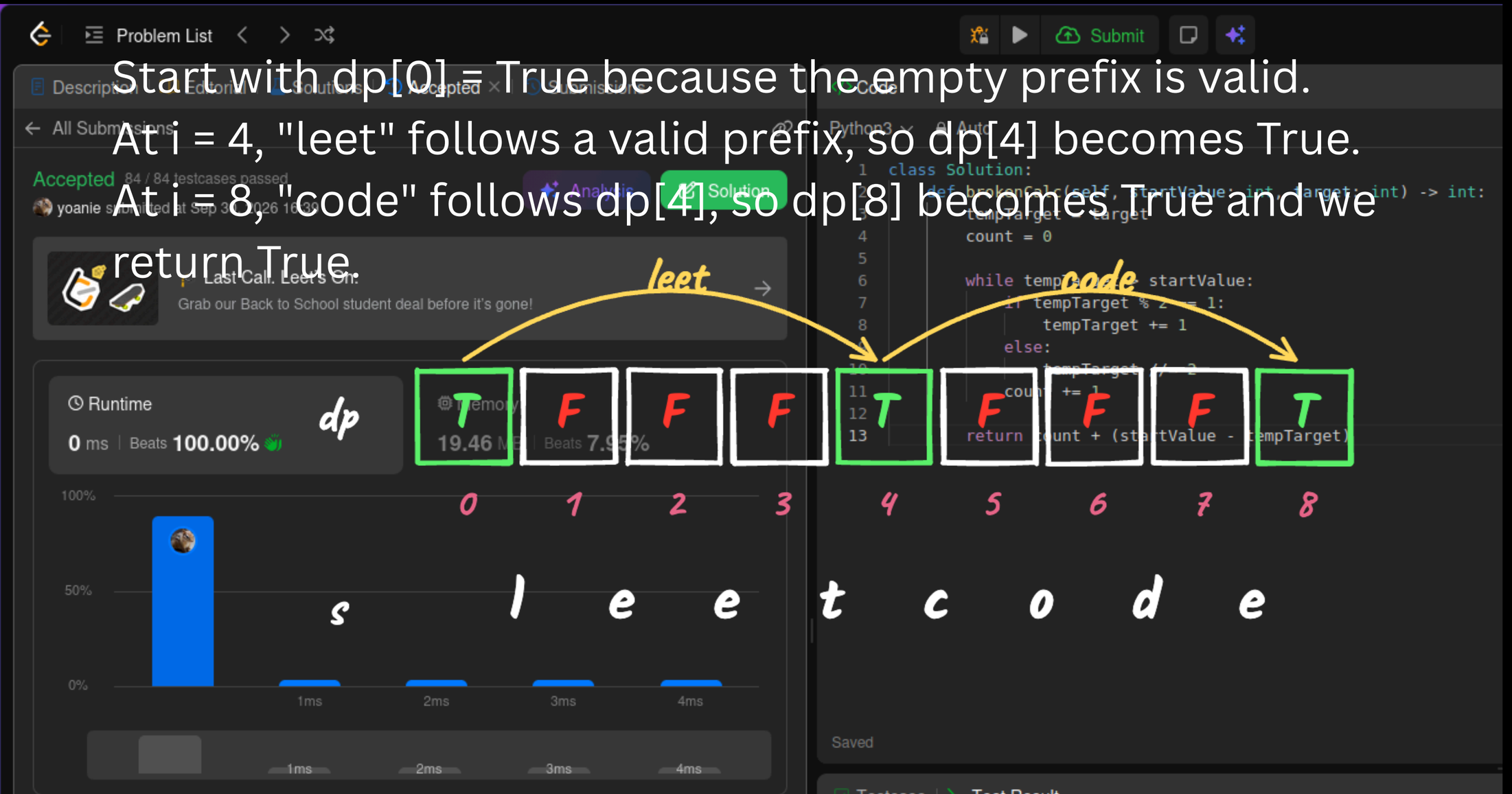


Walkthrough #1: leetcode

Start with dp[0] = True because the empty prefix is valid.

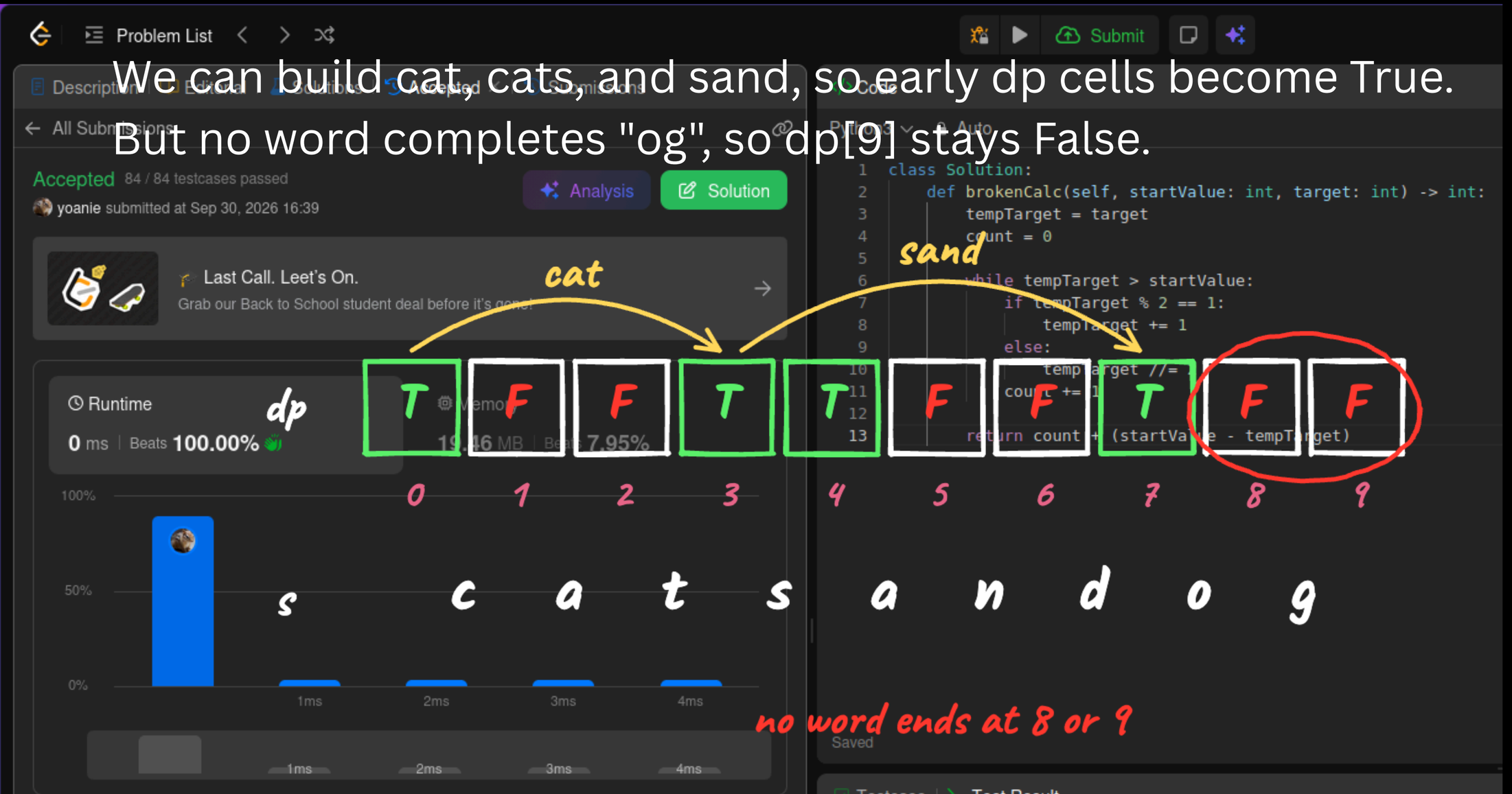
At i = 4, "leet" follows a valid prefix, so dp[4] becomes True.

At i = 8, "code" follows dp[4], so dp[8] becomes True and we return True.



Walkthrough #2: catsandog

We can build cat, cats, and sand, so early dp cells become True.
But no word completes "og", so dp[9] stays False.



We now have our solution (#1)!

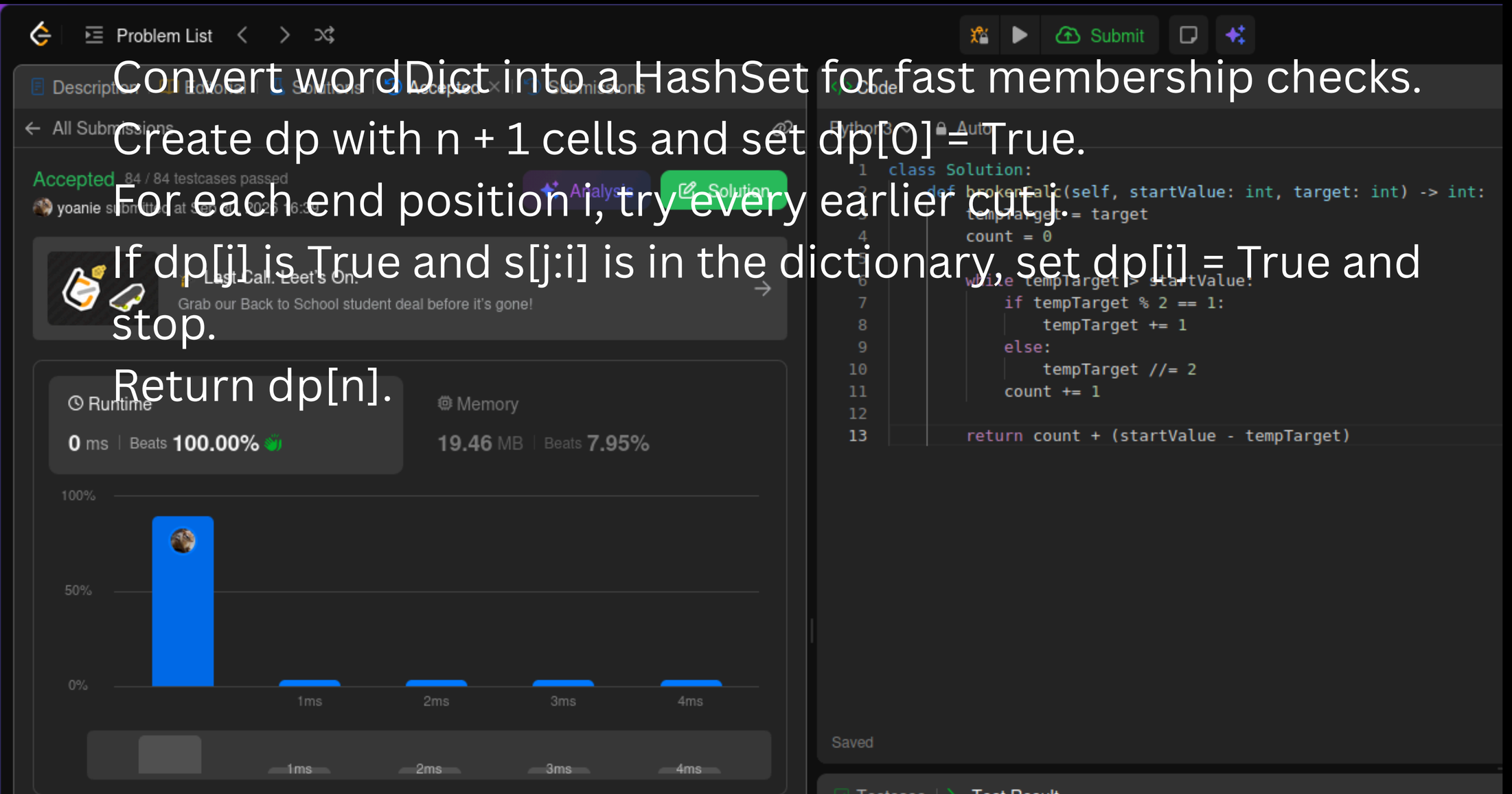
Convert wordDict into a HashSet for fast membership checks.

Create dp with n + 1 cells and set dp[0] = True.

For each end position i, try every earlier cut j.

If dp[j] is True and s[j:i] is in the dictionary, set dp[i] = True and stop.

Return dp[n].



Java Implementation (#1)

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> words = new HashSet<>(wordDict);
        int n = s.length();
        boolean[] dp = new boolean[n + 1];
        dp[0] = true;  // empty string
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && words.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[n];
    }
}
```

time complexity: O(n^2)

space complexity: O(n + m·k)

Python Implementation (#1)

```python
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        words = set(wordDict)
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True  # empty string
        for i in range(1, n + 1):
            for j in range(i):
                if dp[j] and s[j:i] in words:
                    dp[i] = True
                    break
        return dp[n]
```

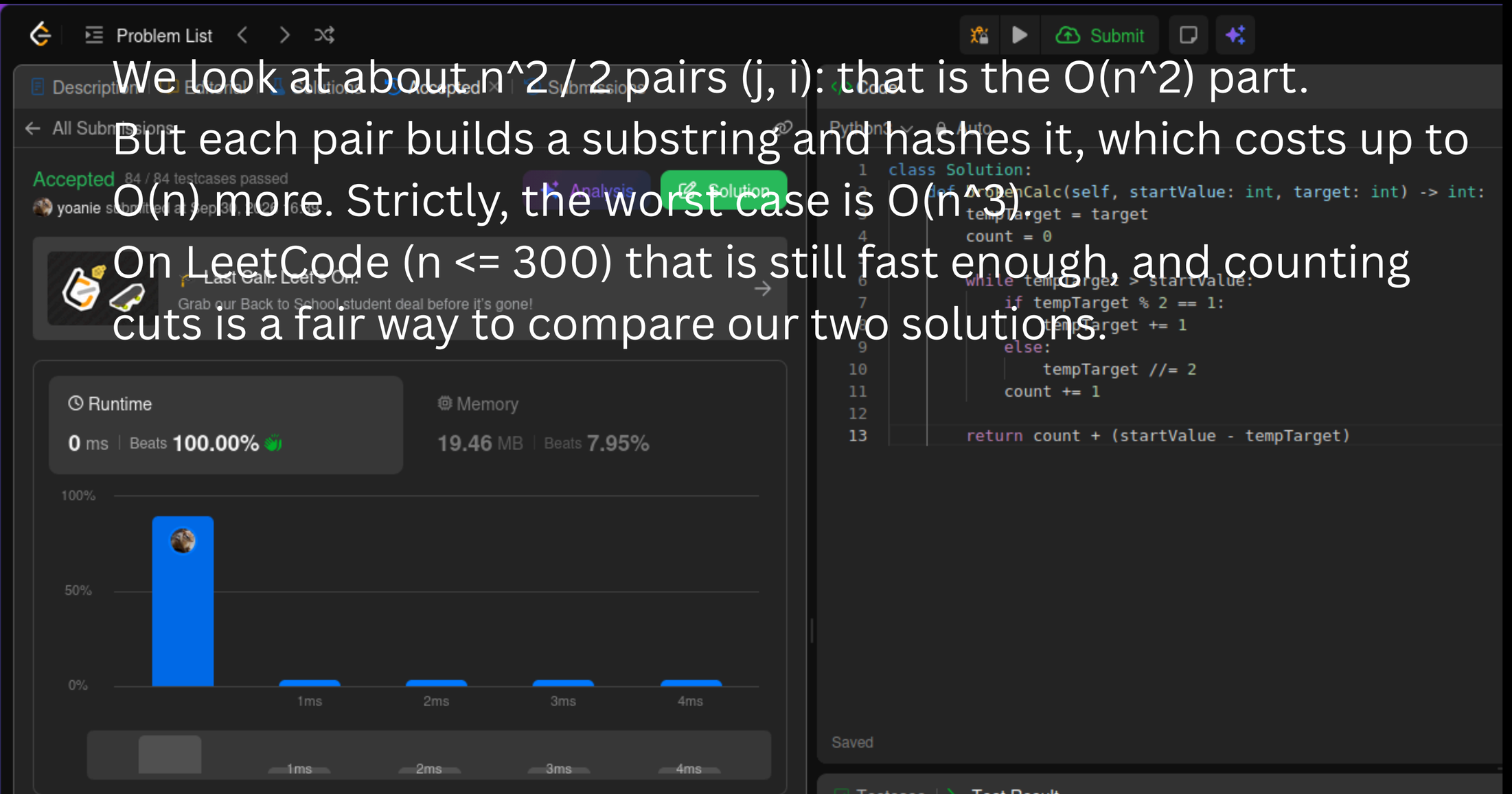
time complexity: O(n^2)
space complexity: O(n + m·k)

Wait, is that really O(n^2)?

We look at about n^2 / 2 pairs (j, i): that is the O(n^2) part.

But each pair builds a substring and hashes it, which costs up to O(n) more. Strictly, the worst case is O(n^3).

On LeetCode (n <= 300) that is still fast enough, and counting cuts is a fair way to compare our two solutions.

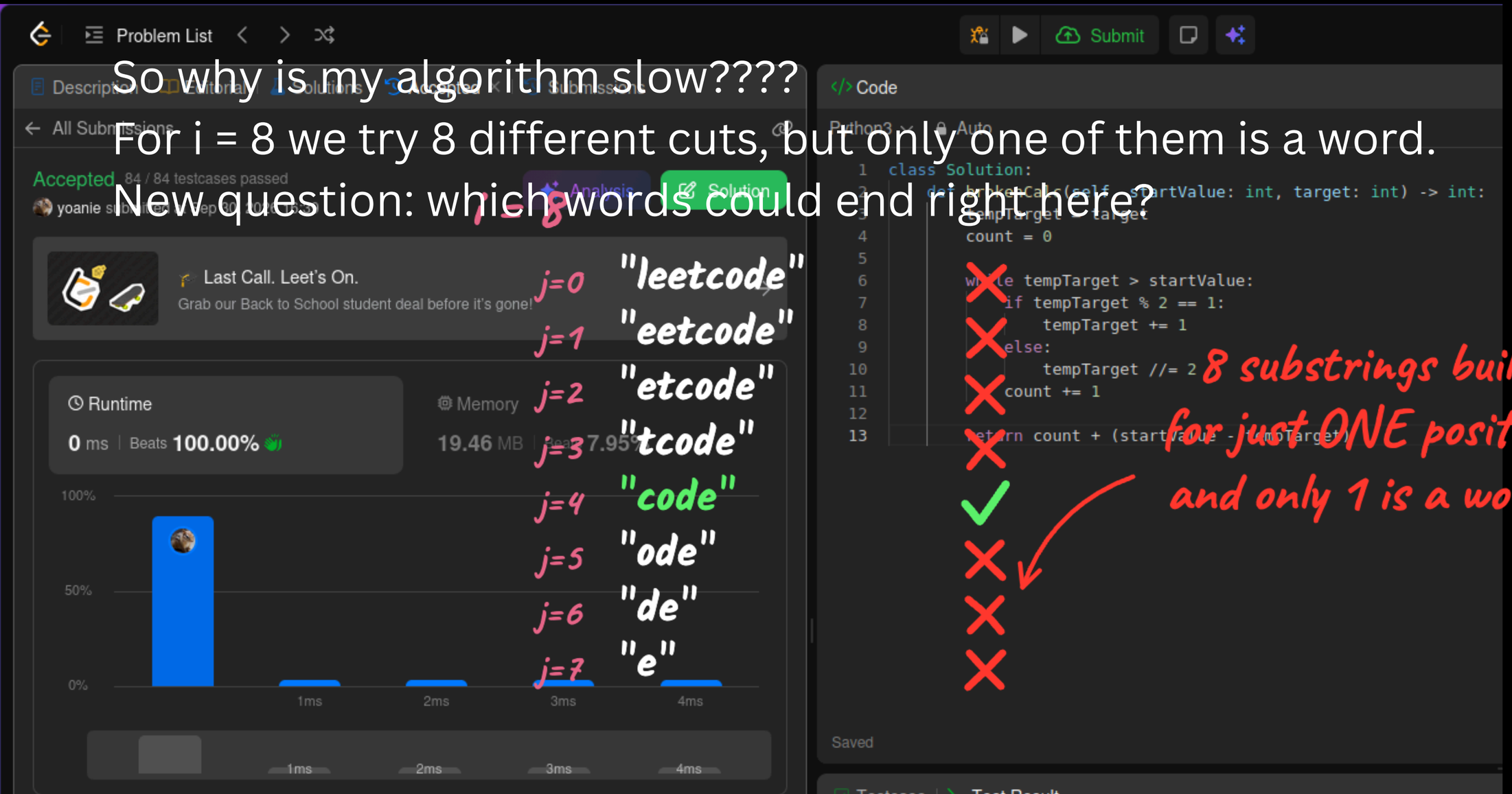


Further Optimization

So why is my algorithm slow????

For i = 8 we try 8 different cuts, but only one of them is a word.

New question: which words could end right here?

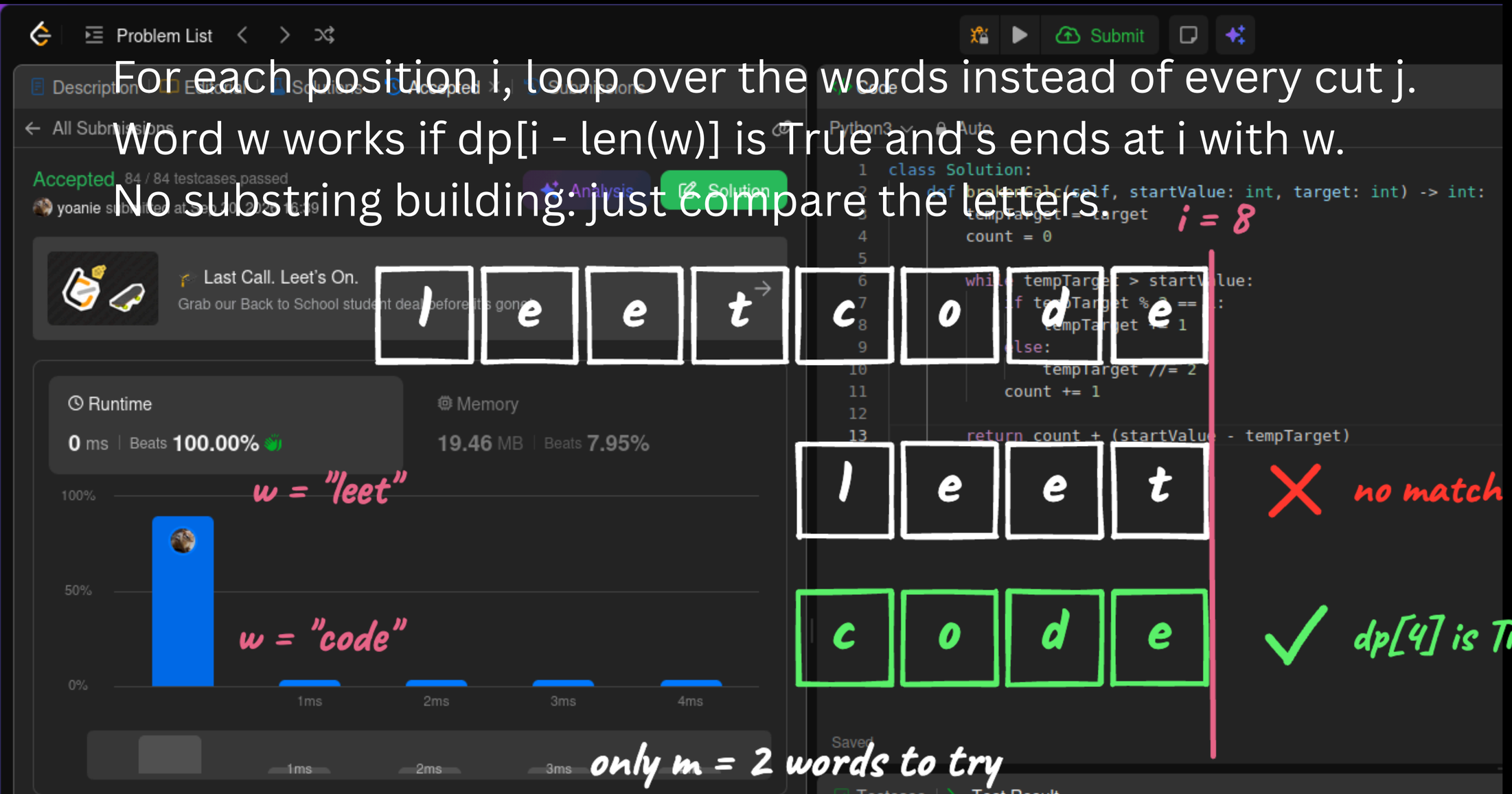


Solution (#2): Flip the Loop

For each position i, loop over the words instead of every cut j.

Word w works if dp[i - len(w)] is True and s ends at i with w.

No substring building: just compare the letters.



We now have our solution (#2)!

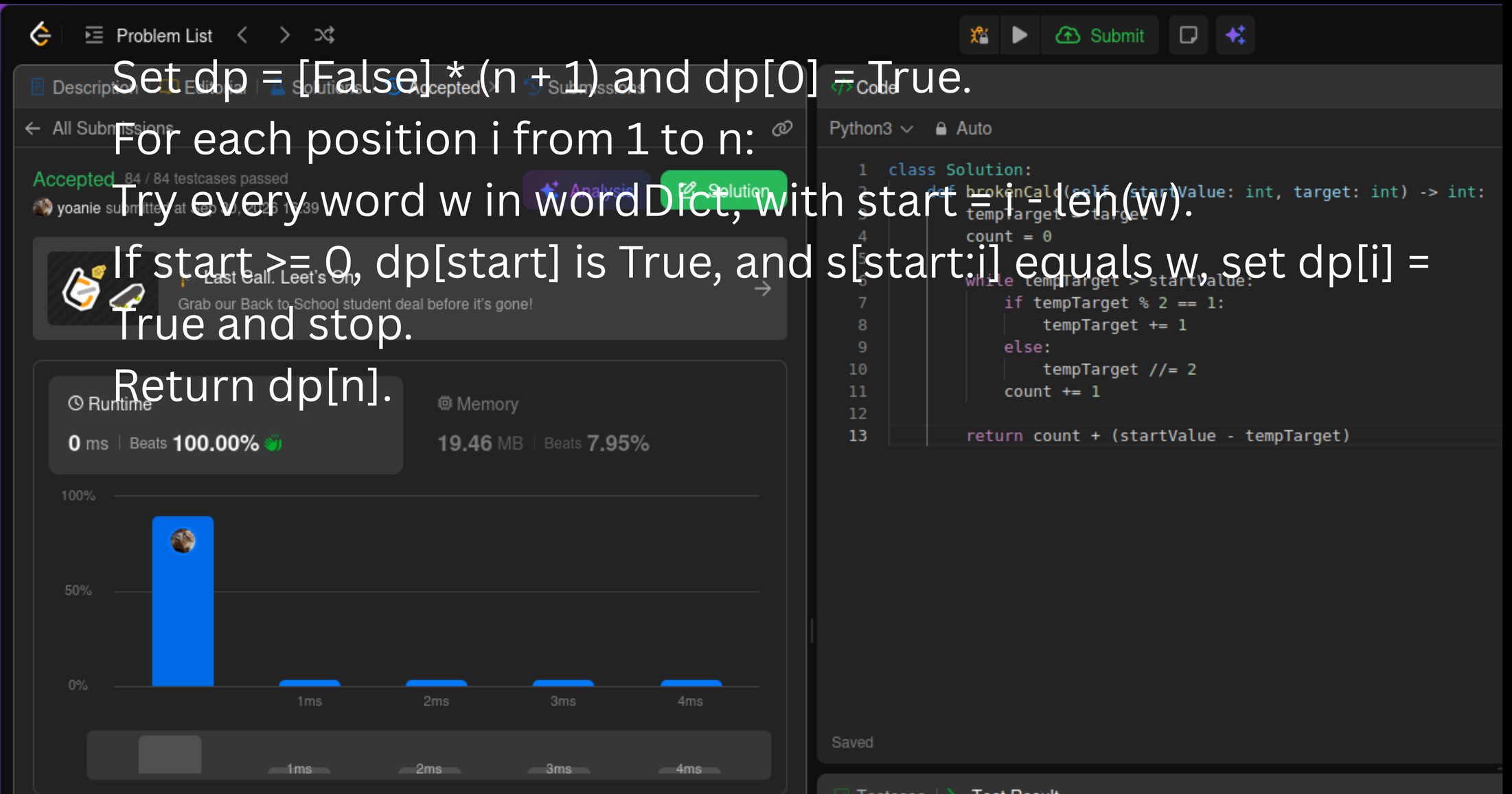
Set dp = [False] * (n + 1) and dp[0] = True.

For each position i from 1 to n:

Try every word w in wordDict, with start = i - len(w).

If start >= 0, dp[start] is True, and s[start:i] equals w, set dp[i] = True and stop.

Return dp[n].



Java Implementation (#2)

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        int n = s.length();
        boolean[] dp = new boolean[n + 1];
        dp[0] = true;  // empty string
        for (int i = 1; i <= n; i++) {
            for (String w : wordDict) {
                int start = i - w.length();
                if (start >= 0 && dp[start] && s.startsWith(w, start)) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[n];
    }
}
```

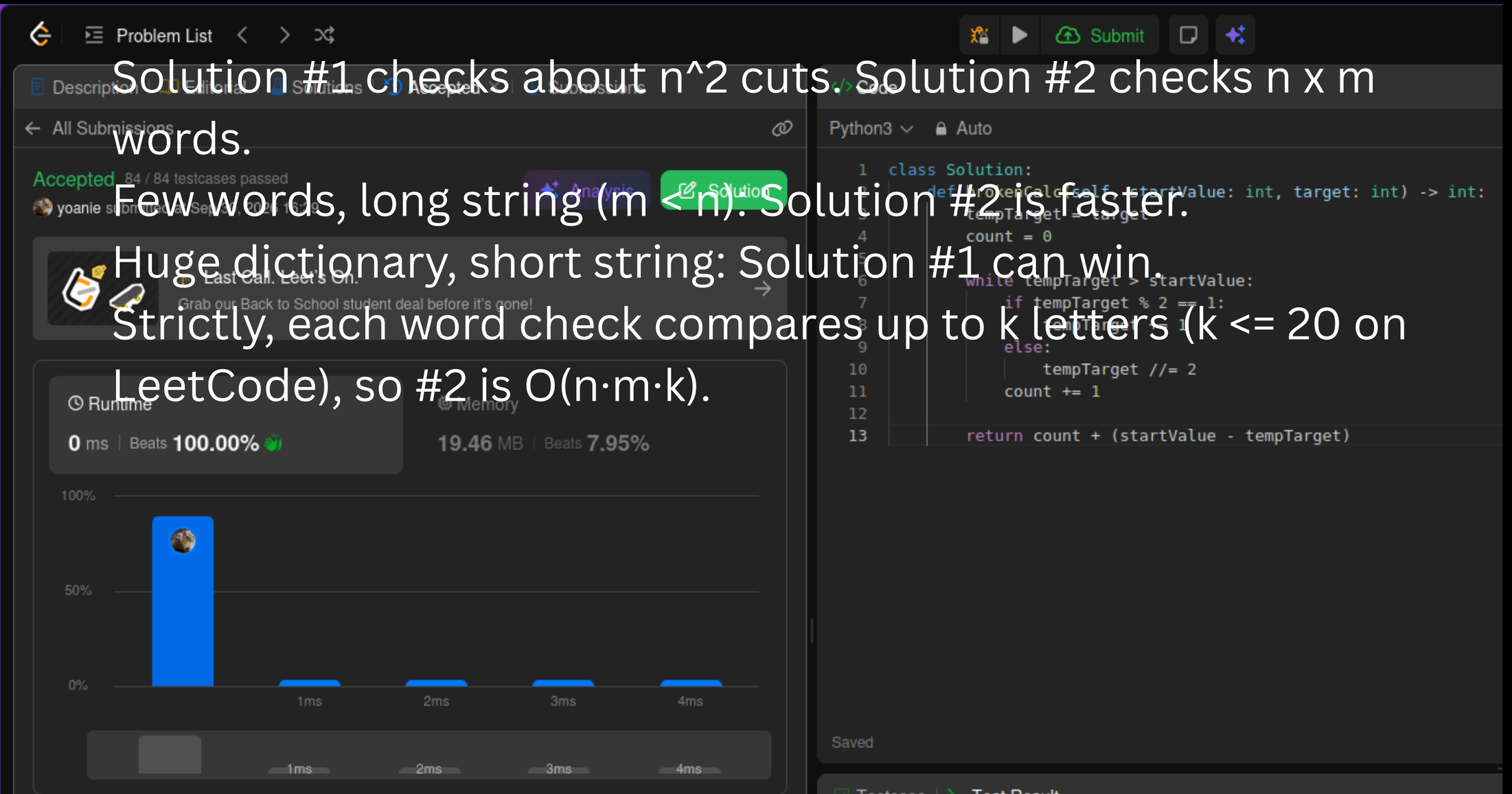
time complexity: O(n·m)
space complexity: O(n)

Python Implementation (#2)

```python
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True  # empty string
        for i in range(1, n + 1):
            for w in wordDict:
                start = i - len(w)
                if start >= 0 and dp[start] and s.startswith(w, start):
                    dp[i] = True
                    break
        return dp[n]
```

time complexity: O(n·m)
space complexity: O(n)

Which Solution Wins?



Solution #1 checks about n^2 cuts. Solution #2 checks n x m words.

Few words, long string (m < n): Solution #2 is faster.

Huge dictionary, short string: Solution #1 can win.

Strictly, each word check compares up to k letters (k <= 20 on LeetCode), so #2 is O(n·m·k).

Hacker Devils

Presented by Sohum Sheth, Dean Richardson and Griffin Charlson

THANK YOU

for your time and attention

