Dynamic Programming

Algorithms and Data Structures

2024/12/09 — Georg Hasebe

Outline

1. Motivation Part
2. Theory Part
3. Implementation Part
4. CodeExpert Part

Motivation Part

DP in the lecture

🎉

Best

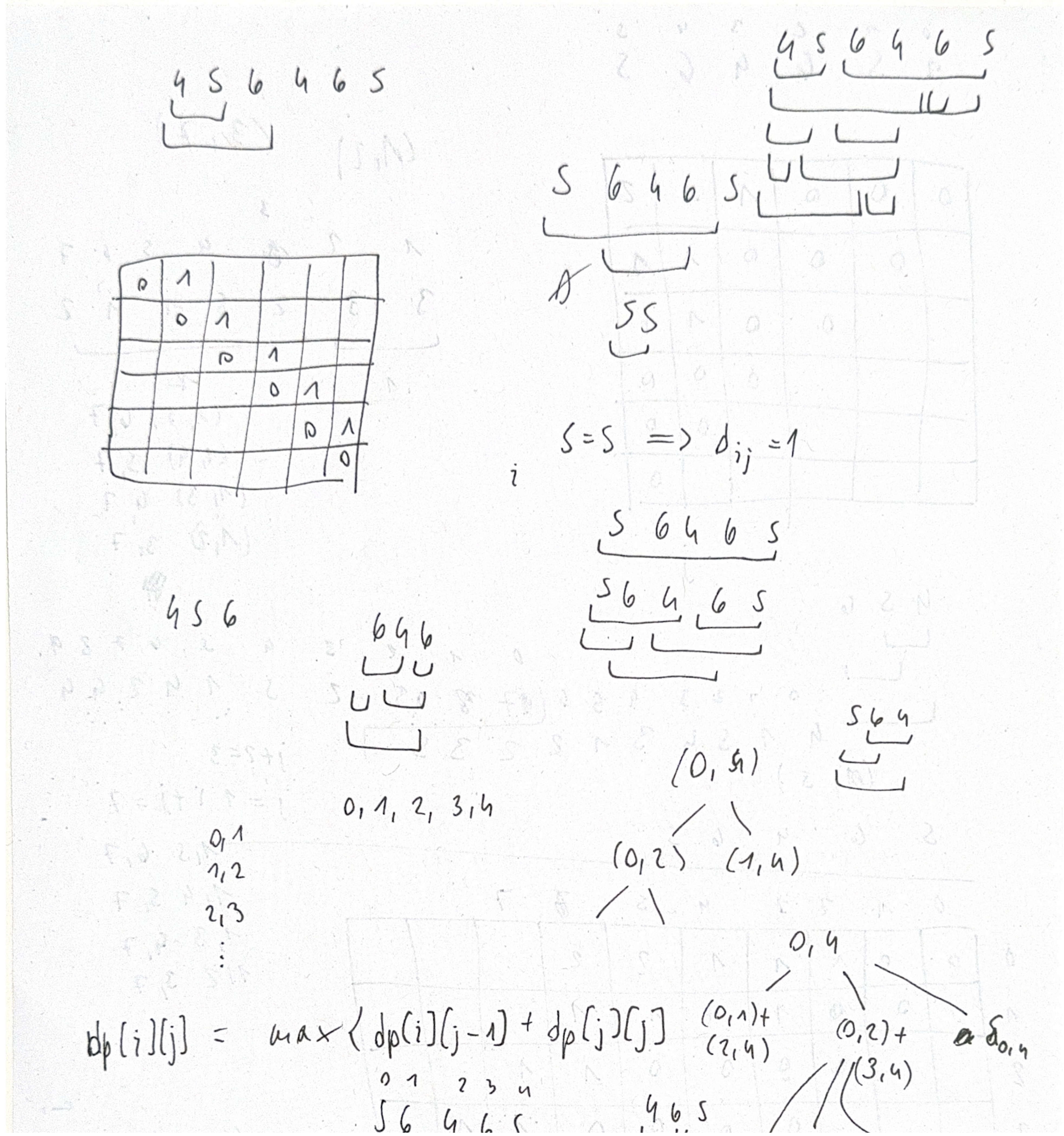
Better

🧍

Naive

DP in real life (at least for me)

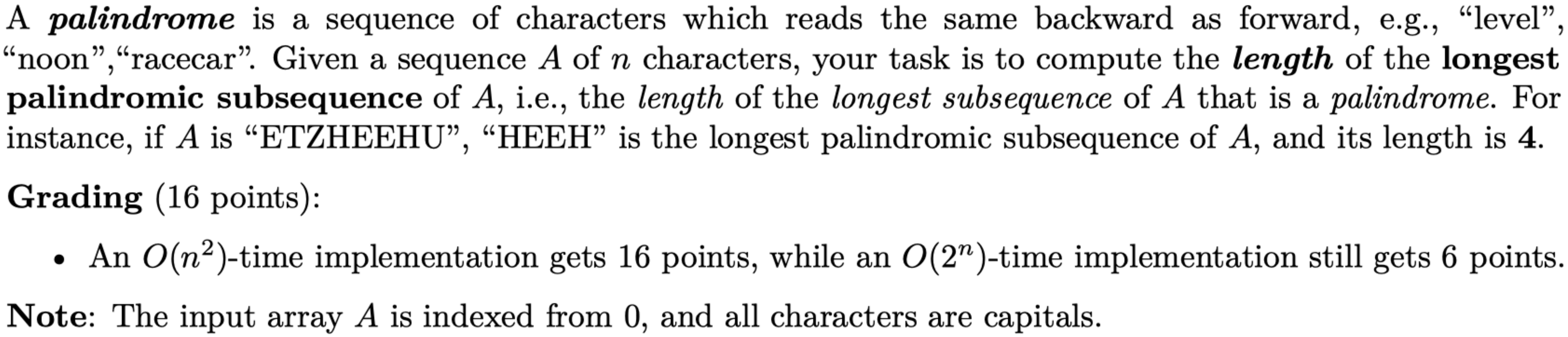
My notes from the CodeExpert exam



Today I want to simulate such a real life situation, **including mistakes** and giving tips along the way.

Theory Part

Longest Palindromic Subsequence (LPS)



A ***palindrome*** is a sequence of characters which reads the same backward as forward, e.g., “level”, “noon”,“racecar”. Given a sequence A of n characters, your task is to compute the ***length*** of the **longest palindromic subsequence** of A, i.e., the *length* of the *longest subsequence* of A that is a *palindrome*. For instance, if A is “ETZHEEEHU”, “HEEH” is the longest palindromic subsequence of A, and its length is **4**.

Grading (16 points):

- An $O(n^2)$-time implementation gets 16 points, while an $O(2^n)$-time implementation still gets 6 points.

Note: The input array A is indexed from 0, and all characters are capitals.

Remark.

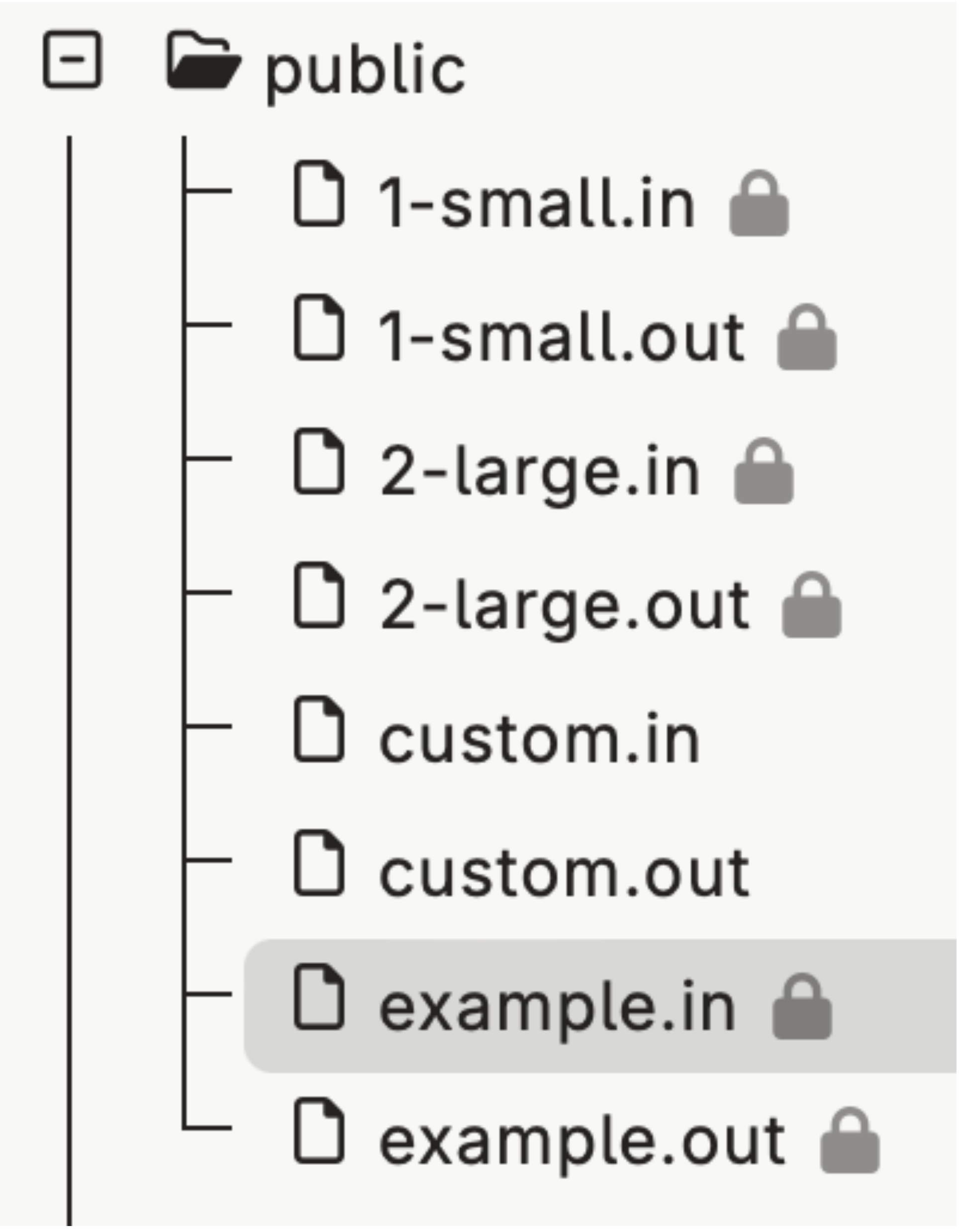
A common mistake is confusing substrings and subsequences.

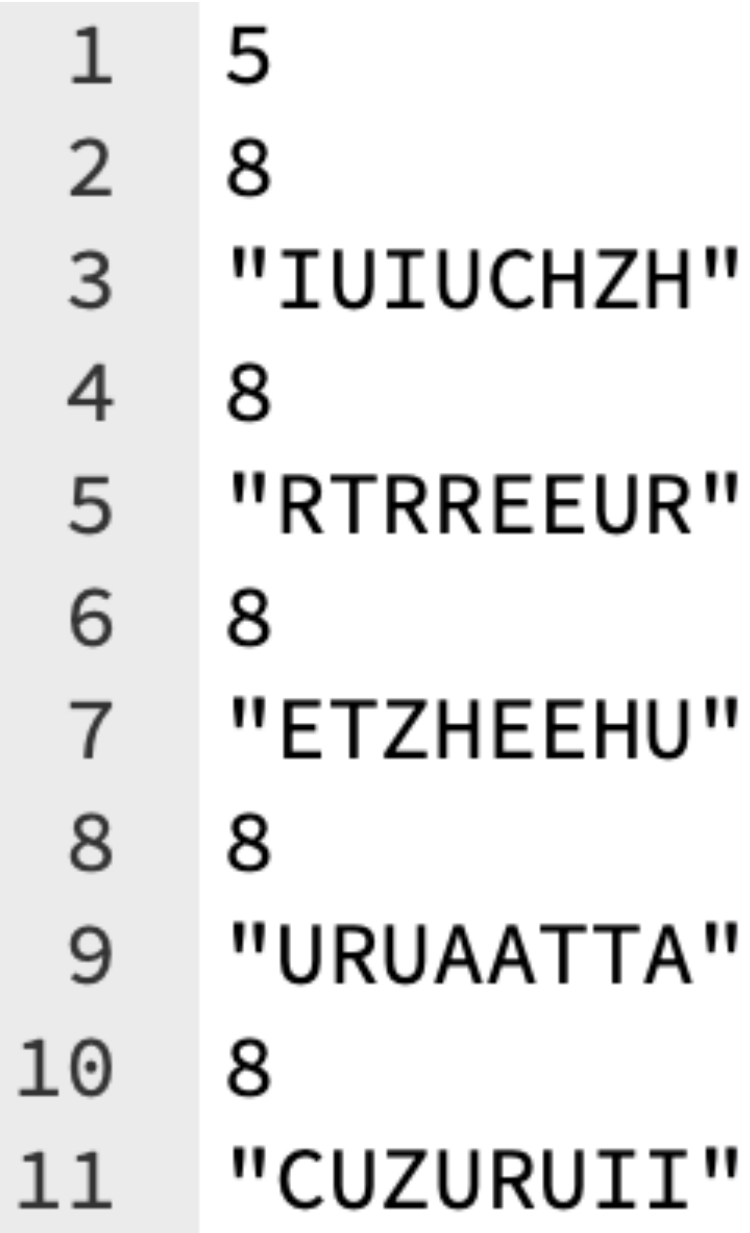
For example, “XYZ” is a subsequence of “XAYBZ” but certainly not a substring.

The example $S =$ *"ETZHEEHU" from the problem description doesn’t show this at all, since the solution they offer is "HEEH" which could be both substring and subsequence!*

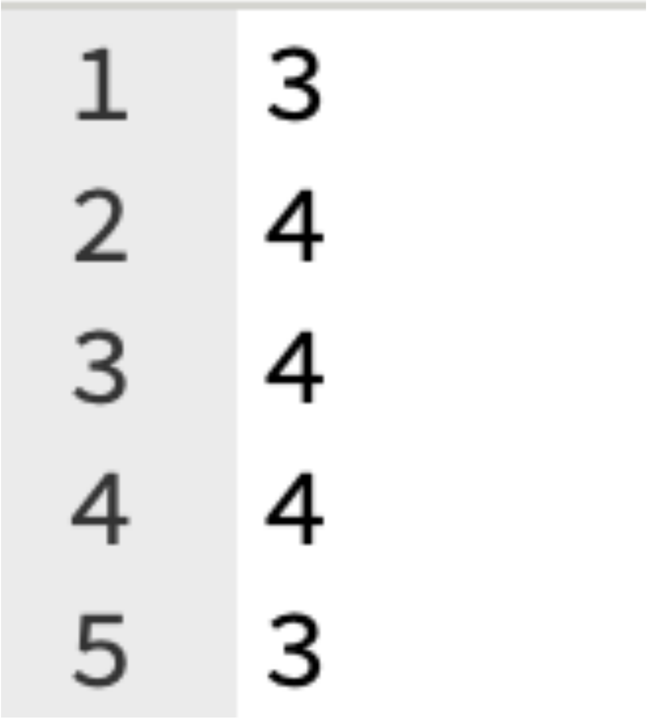
Looking at other examples really helps!

Looking at other examples really helps





```
5
8
"IUIUCHZH"
8
"RTRREEUR"
8
"ETZHEEHU"
8
"URUAATTA"
8
"CUZURUII"
```



```
3
4
4
4
3
```

Even the first example doesn’t help us!

But the second example “RTRREEUR” yields answer 4, so the answer must be “RRRR”.

Now I'm sure what they want from me.

Only after I am **confident** I understand
the problem do I start solving it.

Remark.

When solving problems, we often build on the solutions of similar problems.

To improve at solving dynamic programming (DP) problems, it's important to practice a wide variety of problems, including 1D, 2D, and even 3D DP scenarios.

This diverse practice helps us develop the flexibility to quickly experiment with different approaches during exams, increasing our chances of finding a solution within the time constraints.

A first attempt to solve LPS

Given some input string S, we consider it character by character, from start to finish.

$$S = \text{“}\boxed{\text{E}}\ \text{T Z H E E H U}\text{”}$$

Remark.

I will begin by exploring the problem using a concrete example.

The aim is to understand the structure of the subproblems and how they connect, specifically how solving a smaller portion of the string can contribute to solving a larger portion.

Along the way, not every observation will be useful — or even correct — but each step helps us gather valuable insights.

$S =$ **ETHZEBEHU**

LPS of **E**?

Trivially 1.

LPS of **ET**?

Still 1.

LPS of **ETH**?

Still 1.

$\dots$

$S =$ **ETHZEBEHU**

LPS of **ETHZE**?

For the first time we observe a change in the LPS:

*Since in **ETHZE**, the first **E** and last **E** match, we have a subsequence **EE** of length 2. So the new answer is ~~2~~*

*Taking any other character that lies in between **EE**, we get a palindrome of length 3, e.g. **ETE** or **EHE** or **EZE**.*

What did we learn from this?

***Hypothesis**: If the first and last characters match, the LPS increases.*

Remark.

***Hypothesis**: If the first and last characters match, the LPS increases.*

This hypothesis is a valuable insight, but it remains a hypothesis until we verify that it holds for any input string.

How do we confirm it? By stress-testing the hypothesis with additional examples to see if it consistently works.

Hypothesis: If the first and last characters match, the LPS increases.

$S = \textbf{ABB} + \textbf{B}$

LPS of **ABB** is 2, but appending **B** leads to LPS 3.

It seems, the second letter is also relevant for our LPS.

$S = \textbf{AXBB} + \textbf{B}$

LPS of **AXBB** is 2, but appending **B** leads to LPS 3.

Third letter also relevant? And fourth, and fifth…?

Something feels off.

Remark.

Something feels off.

While our hypothesis is correct to some extent, it seems we need to account for every letter at every position, and it's not immediately clear how to do that.

So, what's the underlying issue? Fundamentally, we're missing some key information.

But why is that?

We’re missing some key information. But why is that?

By (linearly) appending the letter **B** as seen with the examples $S =$ **ABB** $+$ **B** we only consider **B** as the last letter of the subsequence!

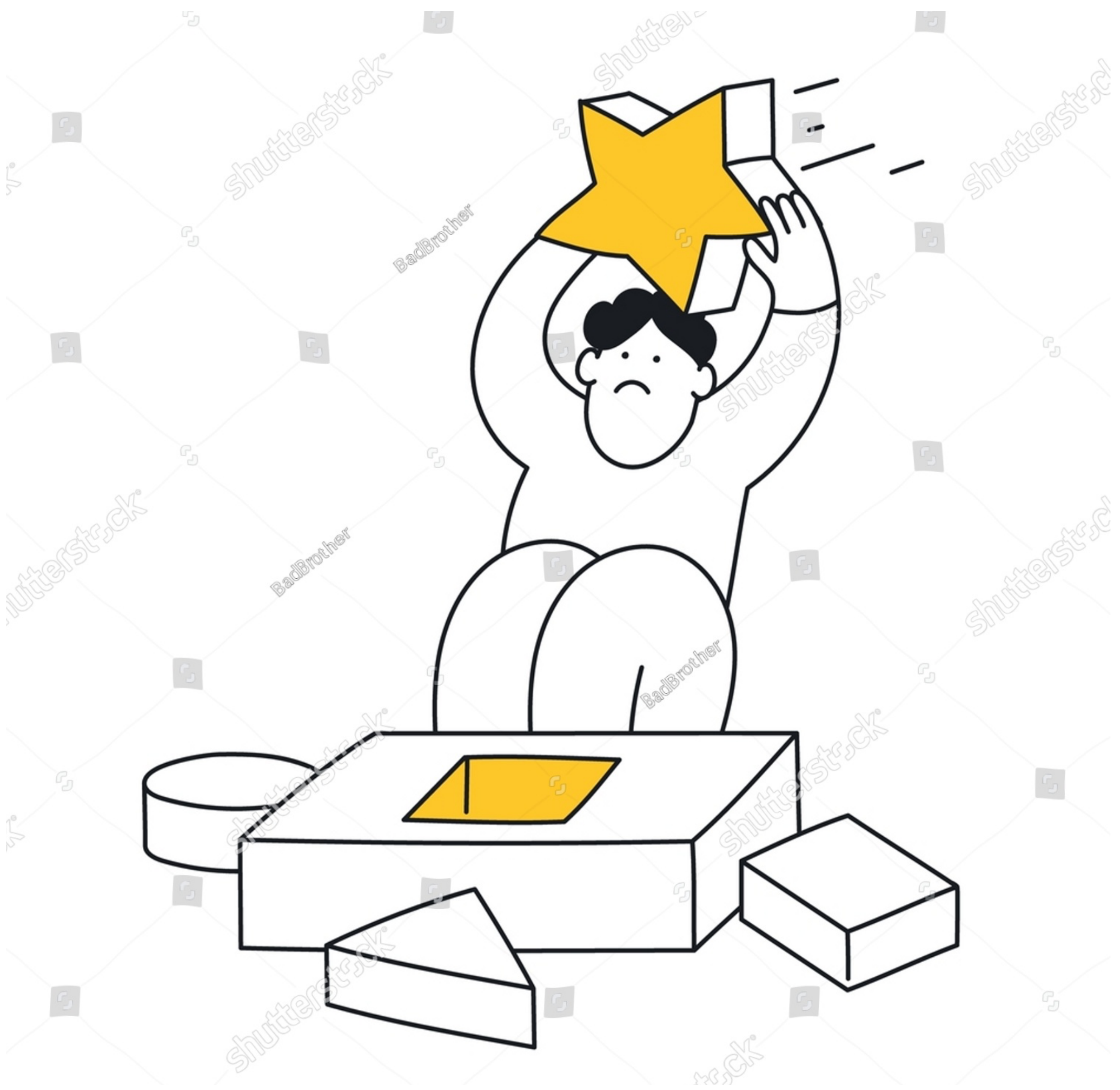
Consider $S =$ **ETHZEBEHU** again. Take **Z** for example. **Z** can be at the end of a subsequence (**ETHZ**), at the start (**ZEBE**) or even the middle (**THZEB**)!

So only considering Z at the end position won’t work!

Remark.

In my experience, students often approach a DP problem by immediately trying to construct a DP table bottom-up.

While this method can work in some cases, it's not always the best approach. In many situations, this strategy feels like forcing the problem into a predefined framework, which may differ ***significantly*** *from the structure of the actual solution.*



dis u?

Remark.

In my opinion, a better starting point for solving a DP problem is to first deeply understand the problem: identify the subproblems, how they depend on each other, and how they collectively build up to the overall solution.

By approaching the problem this way, you are more likely to design a DP strategy that aligns naturally with its structure — whether it's bottom-up, top-down with memoization, or a combination of both.

DP isn't about mechanically filling in a table; it's about uncovering and leveraging the recurrence relationships that form the foundation of the solution.

A second attempt to solve LPS

We abandoned the first approach where we considered each character one by one. What now?

What did we learn from our first approach?

*Recall our **hypothesis** that was correct to some extent: If the first and last characters match, the LPS increases.*

In other words, we don't just care about individual characters, we care about multiple characters, particularly those at the start and end. Let's use this insight as a new starting point.

$S =$ **ETHZE**

LPS of **ETHZE**?

*Since the first and last character match (**hypothesis**), there was an increase in our LPS.*

But increase by how much exactly?

Let’s try to analyze the situation.

In S = **ETHZE**, the matching subsequence, namely **EE**, forms something like a frame of a potential palindrome, and what’s inside of this frame matters.

ETHZE → E □ E

Hypothesis 2*: The LPS is the sum of the LPS of the gray box, and some other number, that depends on if the first and last character match.*

Remark.

***Hypothesis 2**: The LPS is the sum of the LPS of the gray box, and some other number, that depends on if the first and last character match.*

As before, we try to confirm our hypothesis by stress-testing it with additional examples to see if it consistently works.

Hypothesis 2: The LPS is the sum of the LPS of the gray box, and some other number, that depends on if the first and last character match.

ETHZX → E ▭ X

How many cases do we have here?

1. LPS is LPS of **E** ▭
2. LPS is LPS of ▭ **X**
3. LPS is LPS of ▭

Let’s go over these cases and their relevance.

Always case 1?

ETHZX → E X

1. LPS is LPS of **E**

Like earlier we try some examples.

ABBB → **A** **B**, but LPS of **ABBB** is LPS of **B**. Contradiction.

Always case 2?

ETHZX → E ▭ X

2. LPS is LPS of ▭ **X**

BBBA → **B** ▭ **A**, but LPS of **BBBA** is LPS of **B** ▭. Contradiction.

Always case 3?

ETHZX → E ▭ X

3. LPS is LPS of ▭.

Both the examples of case 1 and 2 are counterexample to case 3.

In retrospect, it didn’t make much sense to include this case, because all the characters of the blue box are contained in both case 1 and 2. We omit case 3.

Summary

$$S = c_1 c_2 \ldots c_{n-1} c_n \rightarrow c_1 \text{▨} c_n$$

If $c_1 = c_n$, we know that the LPS will be the LPS of ▨ + 2.

If $c_1 \neq c_n$, then we showed that we must consider the LPS of both:

- $c_1 \text{▨}$
- $\text{▨} c_n$

$$\text{LPS}(c_i \text{▨} c_j) = \begin{cases} \text{LPS}(\text{▨}) + 2 & \text{if } c_i = c_j, \\ \max\{\text{LPS}(c_i \text{▨}), \text{LPS}(\text{▨} c_j)\} & \text{otherwise .} \end{cases}$$

Formalizing our Suspicions

Let LPS(i, j) := "LPS of the substring from i to j." and denote with c_i the i-th character of S.

For example, if $S =$ **ABC**, then LPS$(1,2)$ is LPS of **BC** and $c_0 =$ **A** (we use zero-based indexing).

$$\text{LPS}(i,j) = \begin{cases} \text{LPS}(i+1, j-1) + 2 & \text{if } c_i = c_j, \\ \max\{\text{LPS}(i, j-1), \text{LPS}(i+1, j)\} & \text{otherwise}\,. \end{cases}$$

Remark.

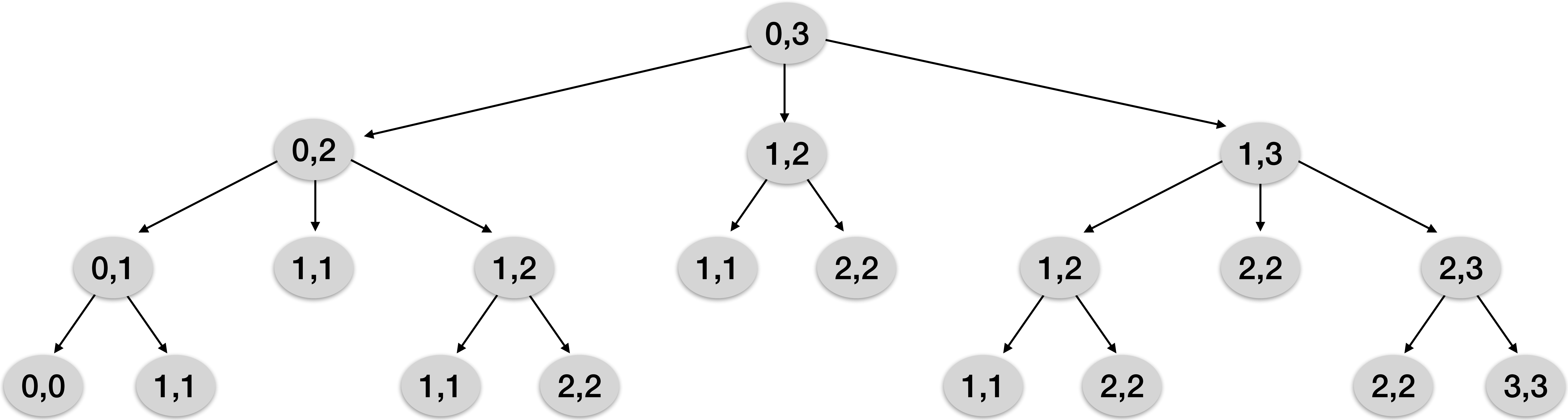
At this point, we would naturally stress-test our recurrence to ensure its correctness before diving into the implementation or filling in all six aspects required in a theory exam. However, I'll take the liberty of skipping this step here, so you can assume the recurrence is correct.

What's next? Now that we've defined suitable subproblems and established the recurrence, we've essentially solved the DP problem. The rest is just about filling in the details.

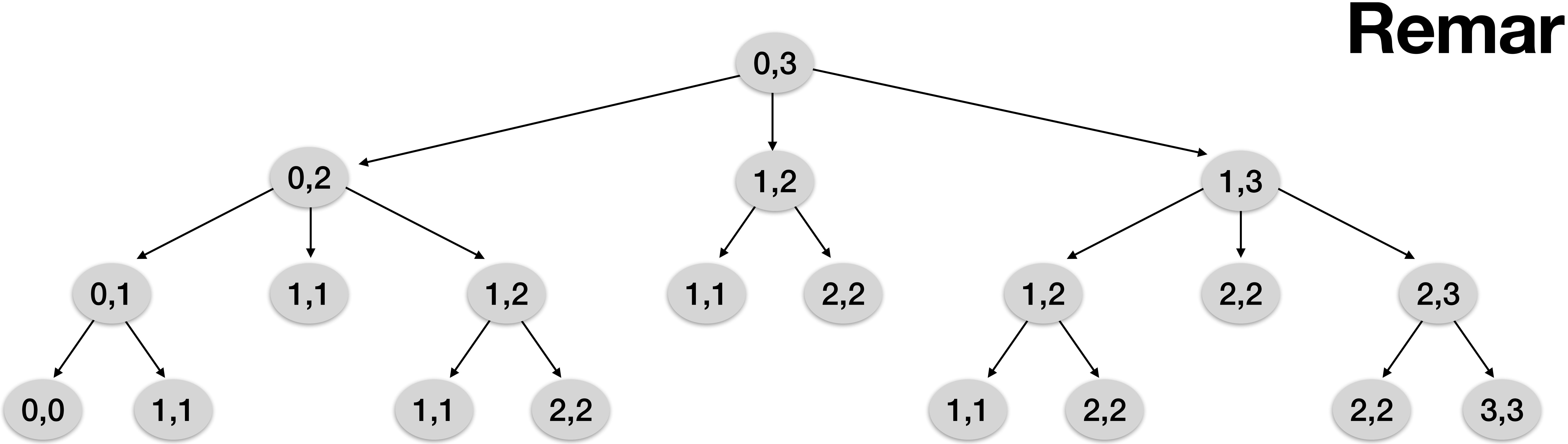
Still not convinced? Let's walk through an example together.

$$\text{LPS}(i,j) = \begin{cases} \text{LPS}(i+1, j-1) + 2 & \text{if } c_i = c_j, \\ \max\{\text{LPS}(i, j-1), \text{LPS}(i+1, j)\} & \text{otherwise}. \end{cases}$$

Let’s visualize the recursion on some string $S[0,\ldots,3]$ of length 4.



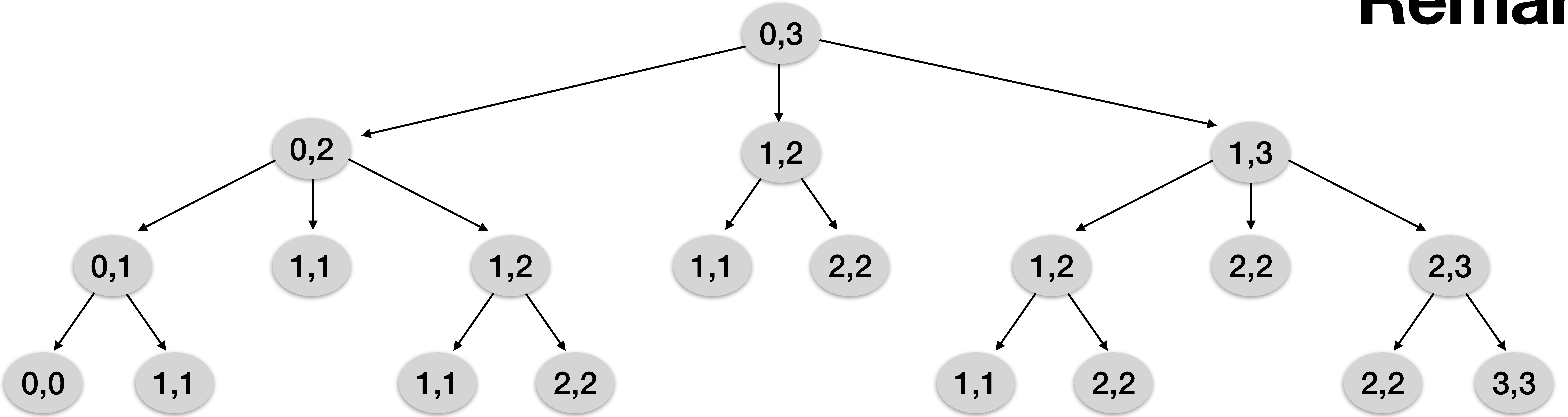
Remark.



The nodes in this graph represent the subproblems. A parent subproblem is connected to its child subproblems whenever the solution of the parent depends on the solutions of the children. The root node is the final result.

Notice that the leaves of the graph correspond to the base cases — those subproblems that do not depend on any others for their solution.

Remark.



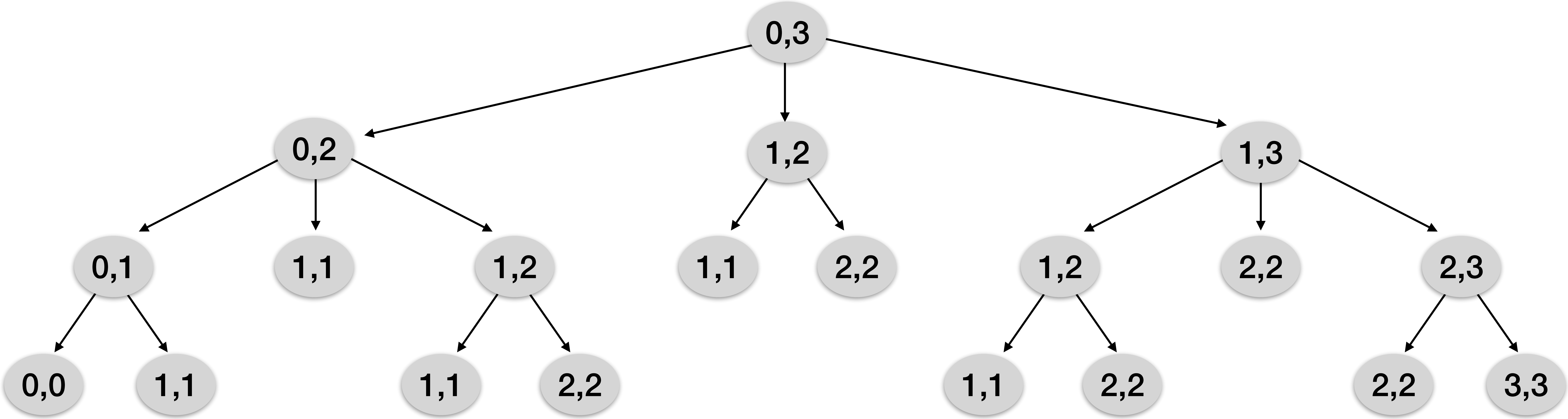
In other words, we now have all the key components needed to fully and formally solve the DP problem.

At this point, we can decide whether to solve it top-down using memoization or bottom-up using tabulation, depending on which approach suits the problem or our preferences better.

Programming Part

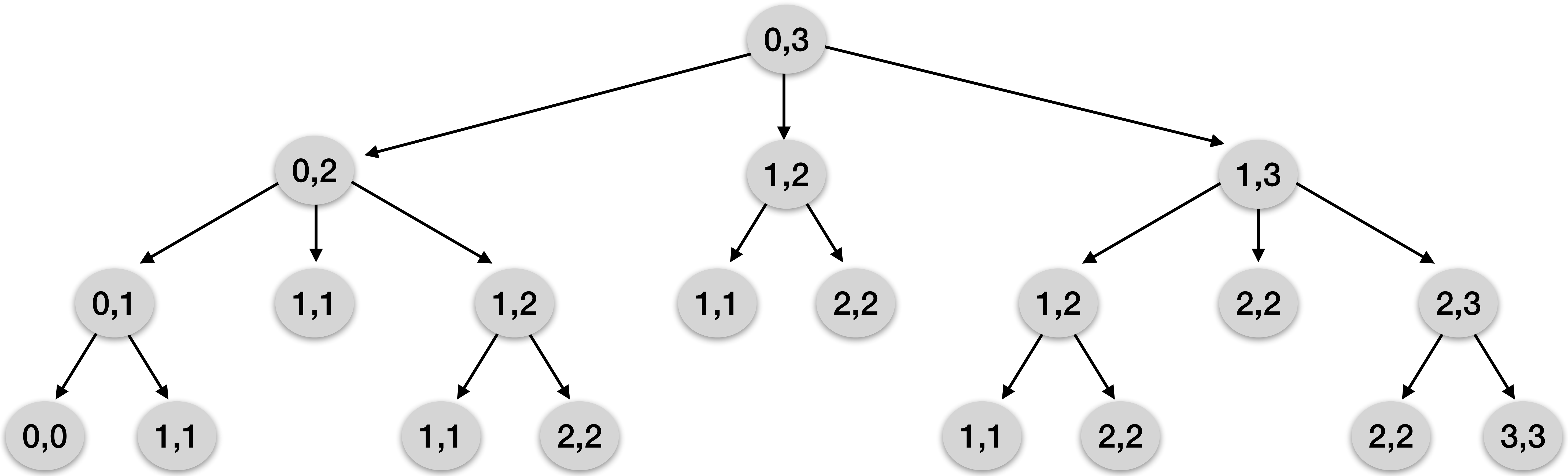
Implementing a solution

Suppose we decide to implement the solution using a bottom-up tabulation approach. Let's revisit the visualization of the subproblems from earlier. Is there a natural way to transform this dependency structure into a table?



Construction of a DP table

0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3



0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

One reason I chose this specific problem is that it's likely unlike anything you've encountered before. Now, let's take a closer look at the table and compare it with the earlier visualization. Notice that:

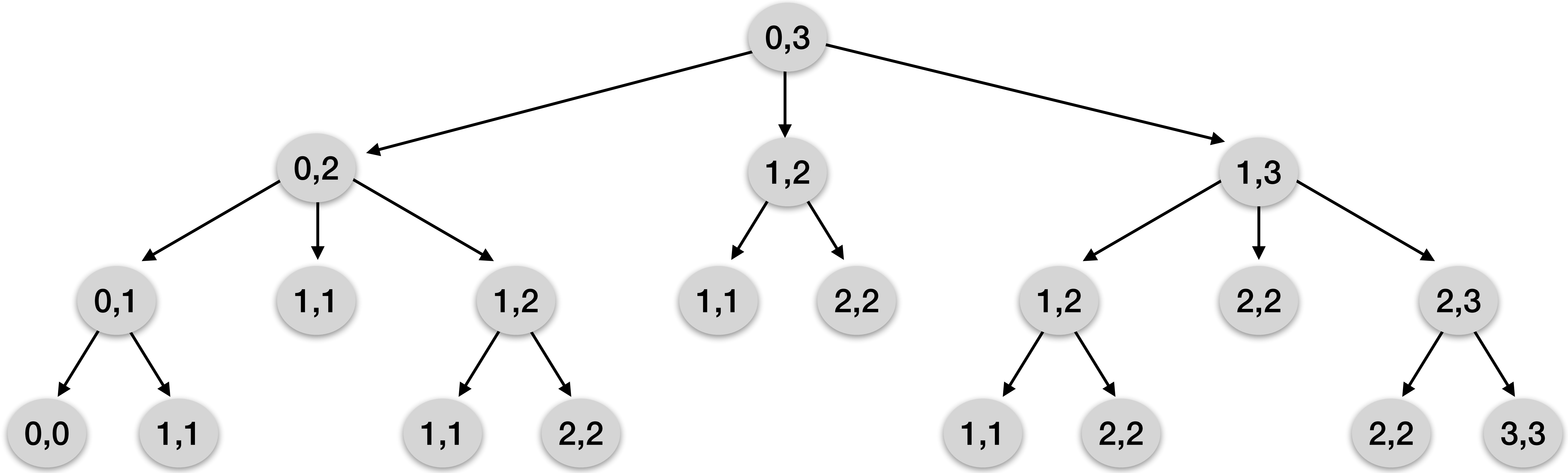
1. The lower left part of the table is simply left empty.

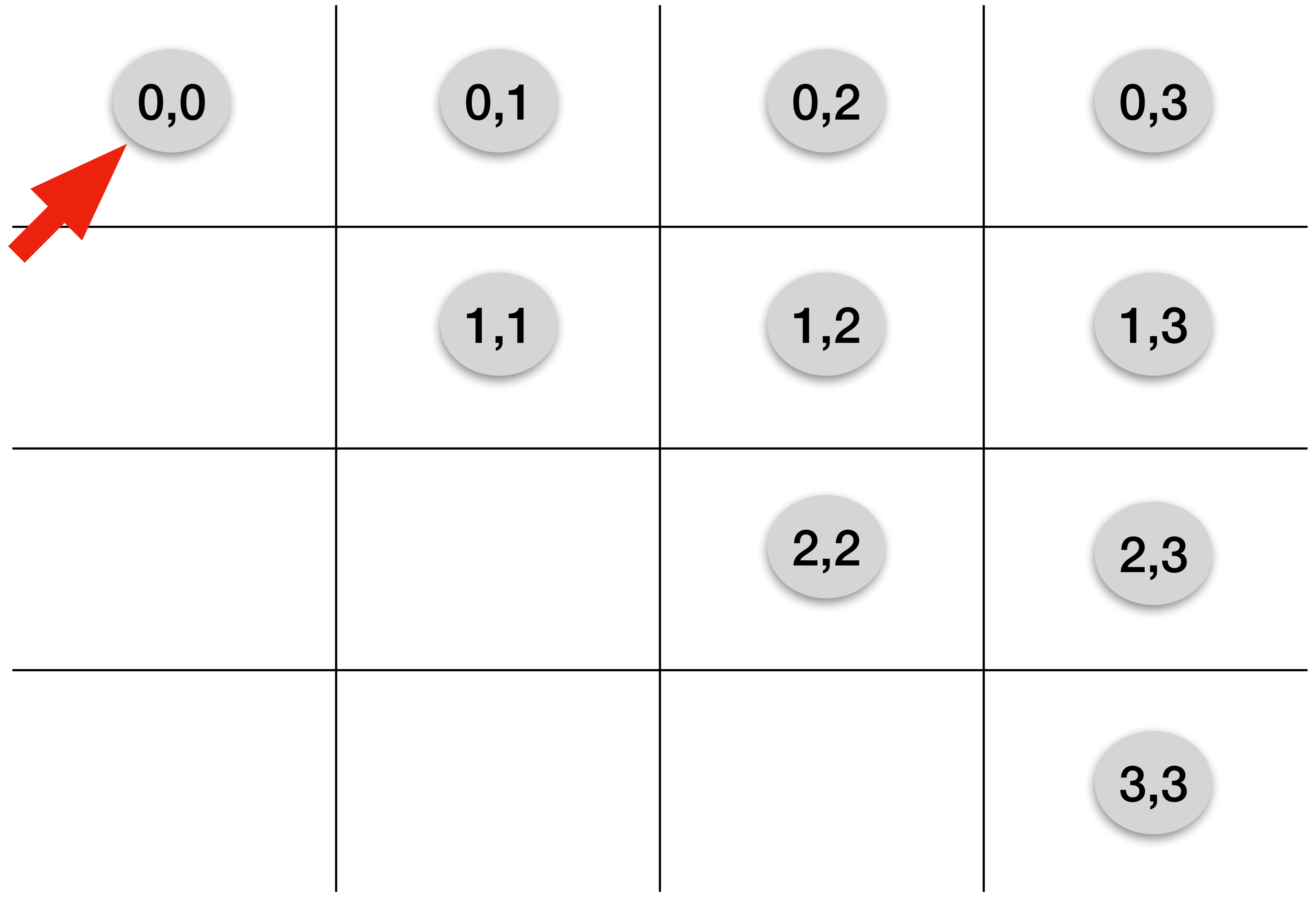
2. The base cases are stored in the diagonal.

3. Each diagonal depends on the lower diagonal.

1. The lower left part of the table is simply left empty.
2. The base cases are stored in the diagonal.
3. Each diagonal depends on the diagonal below it.

How would a bottom-up “move” through such a table?





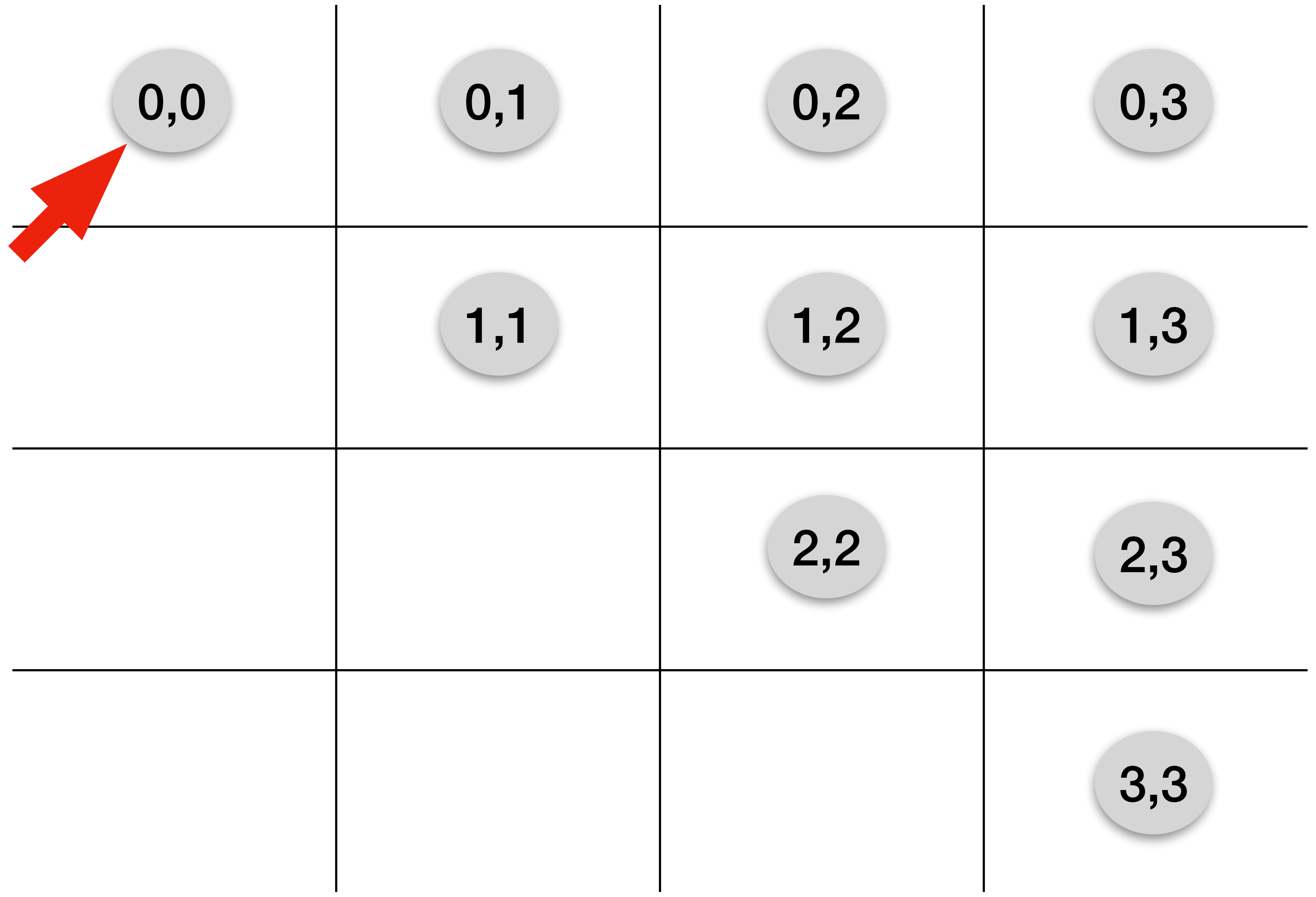
0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

Remark.

This diagonal movement can be tricky to implement, especially if you’re encountering it for the first time. Before jumping into the implementation, let’s take a step back and gather more information.

Understanding the structure of the table, the order of filling it, and how the indices relate to our subproblems will make the implementation process much smoother. Let’s break it down.

DP table implementation



0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

Moving down the diagonal simply increases the row and column by one:

$$(i,j) \to (i+1, j+1).$$

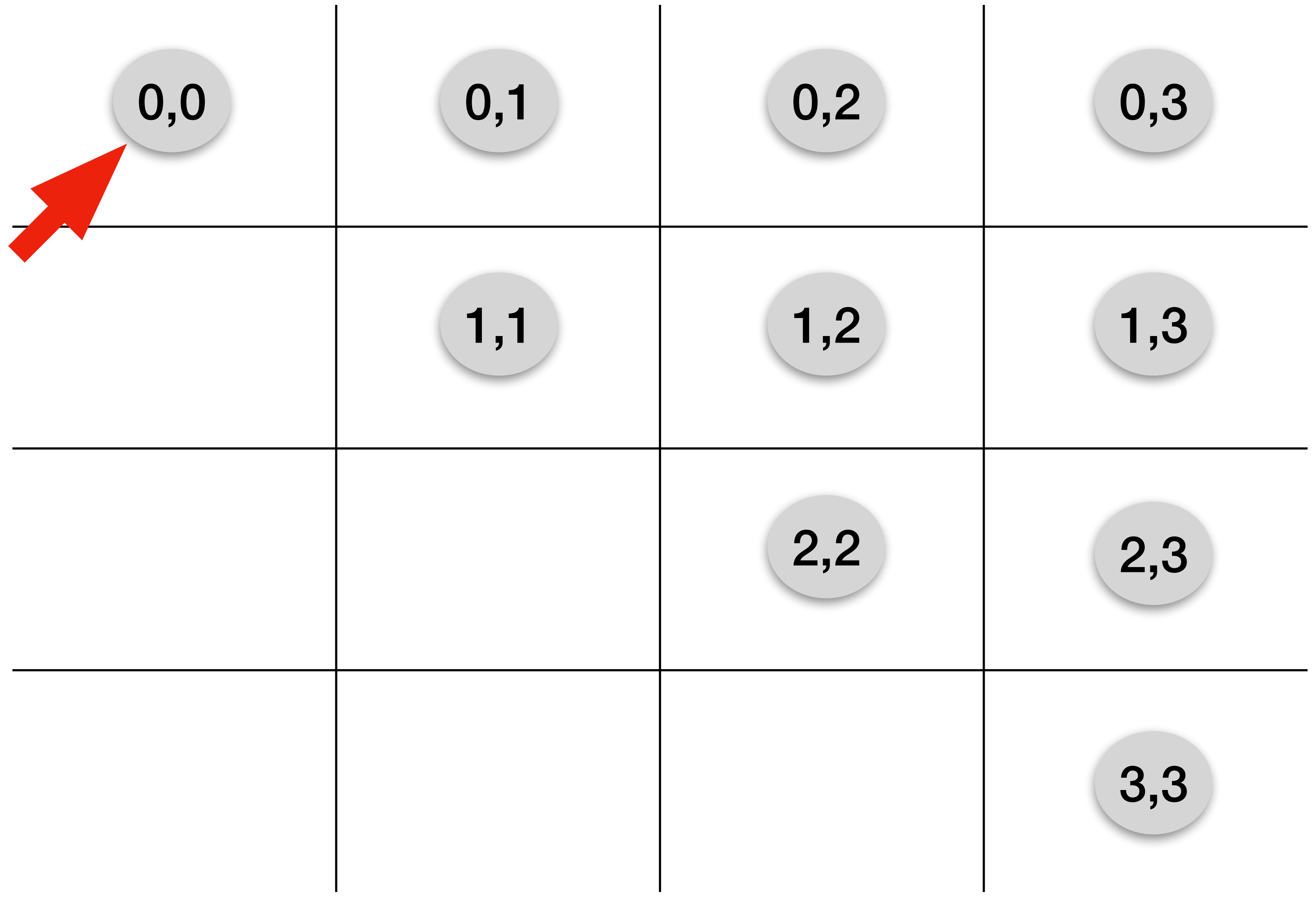
What happens in the second diagonal?

We introduce some *initial offset* $L = 1$:
$(i, L+j) \to (i+1, L+j+1)$.

What are the values for L?

$$L \in \{0,\ldots,3\} = \{0,\ldots,n-1\}.$$

DP table implementation



0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

What are the values for i?

0-th diagonal: 0 to 3.

1-st diagonal: 0 to 2.

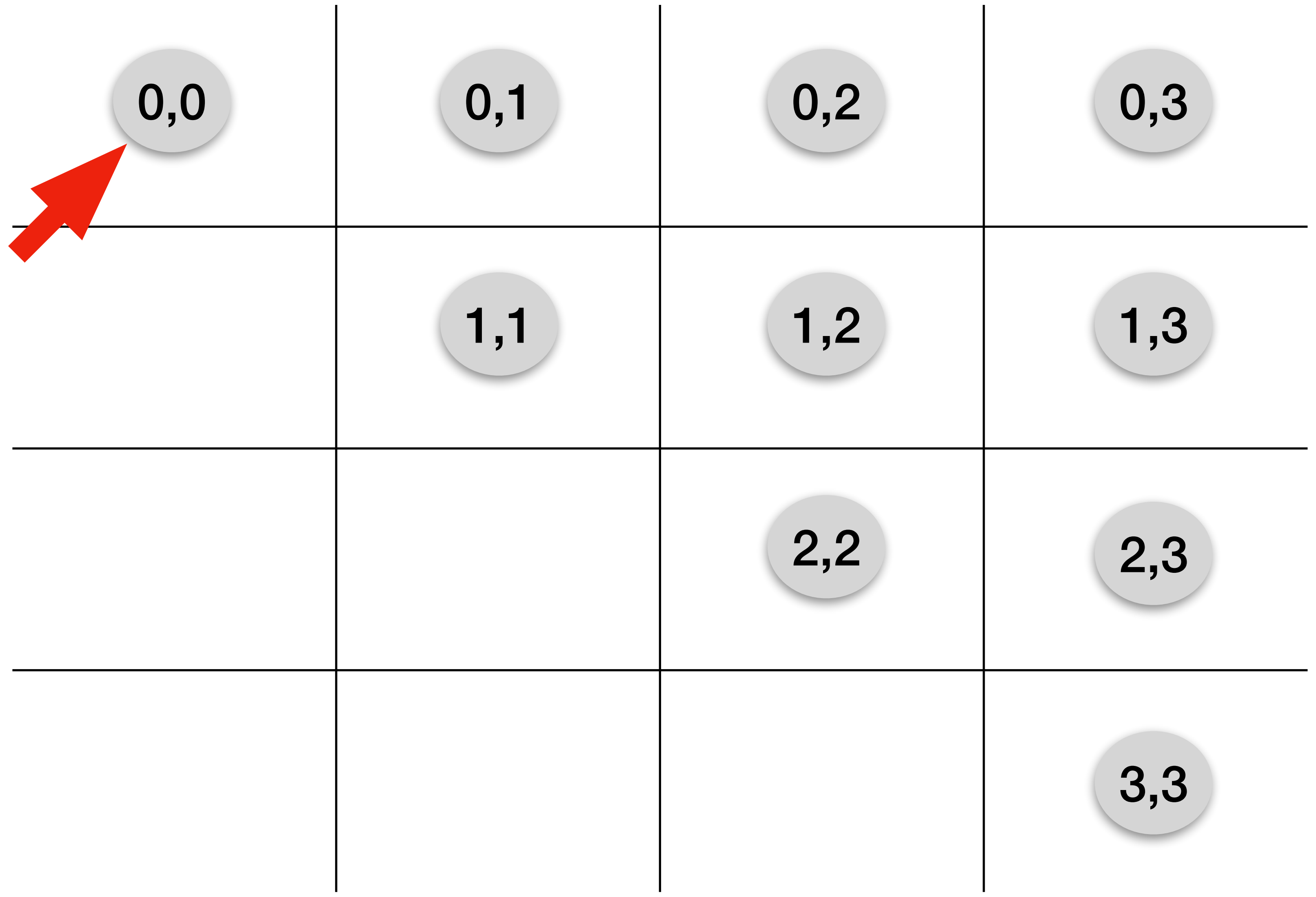
…

$(n-1)$-th diagonal: 0 to 0.

In other words:

L-th diagonal: 0 to $(n-1)-L$.

DP table implementation



0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

What are the values for j?

0-th diagonal: 0 to 3.

1-st diagonal: 1 to 3.

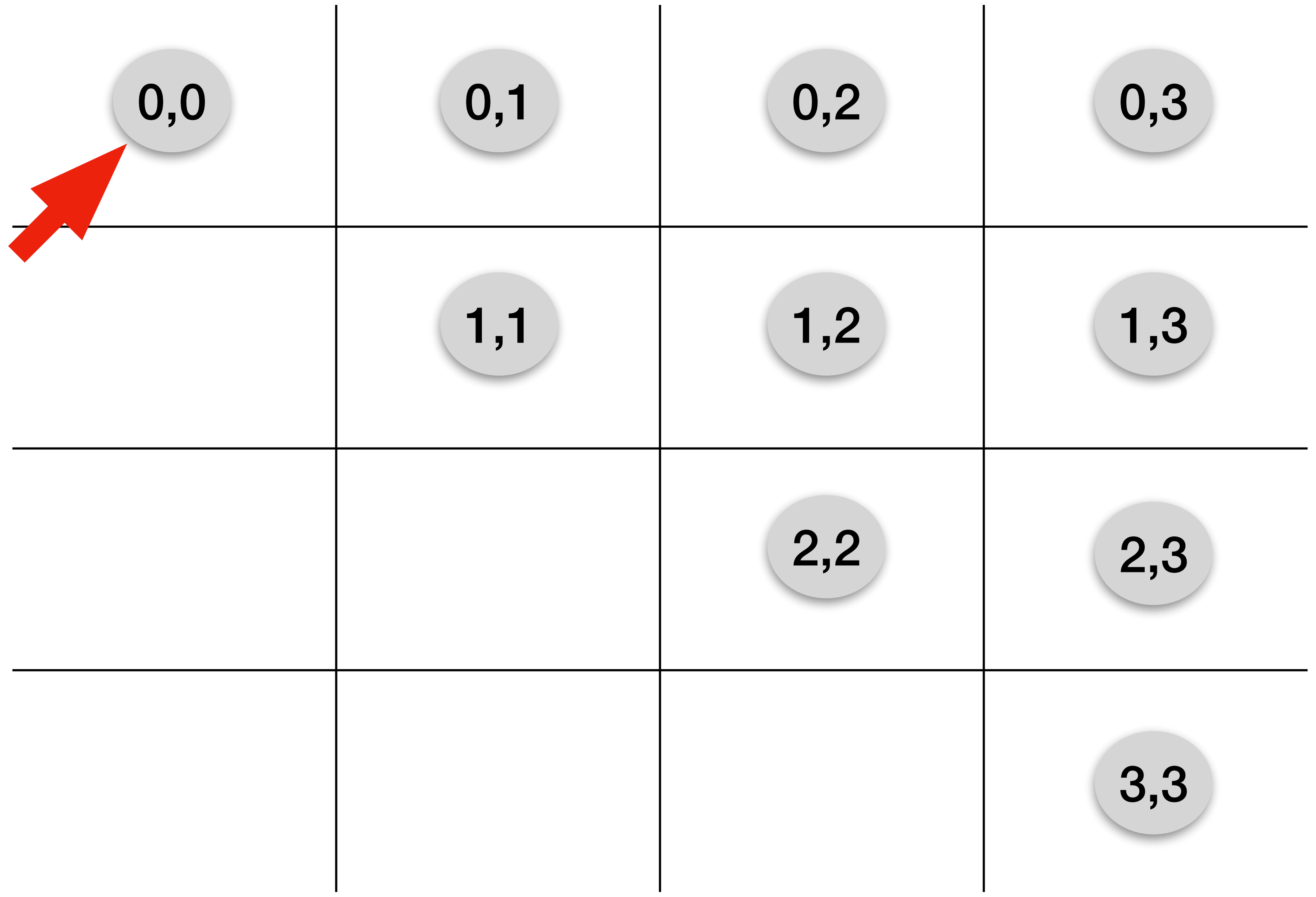
…

$(n-1)$-th diagonal: 3 to 3.

In other words:

L-th diagonal: $0 + L$ to $(n-1)$.

DP table implementation



0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

Summary for the L-th diagonal:

Movement:

$(i, L + j) \rightarrow (i + 1, L + j + 1)$.

Values for i:

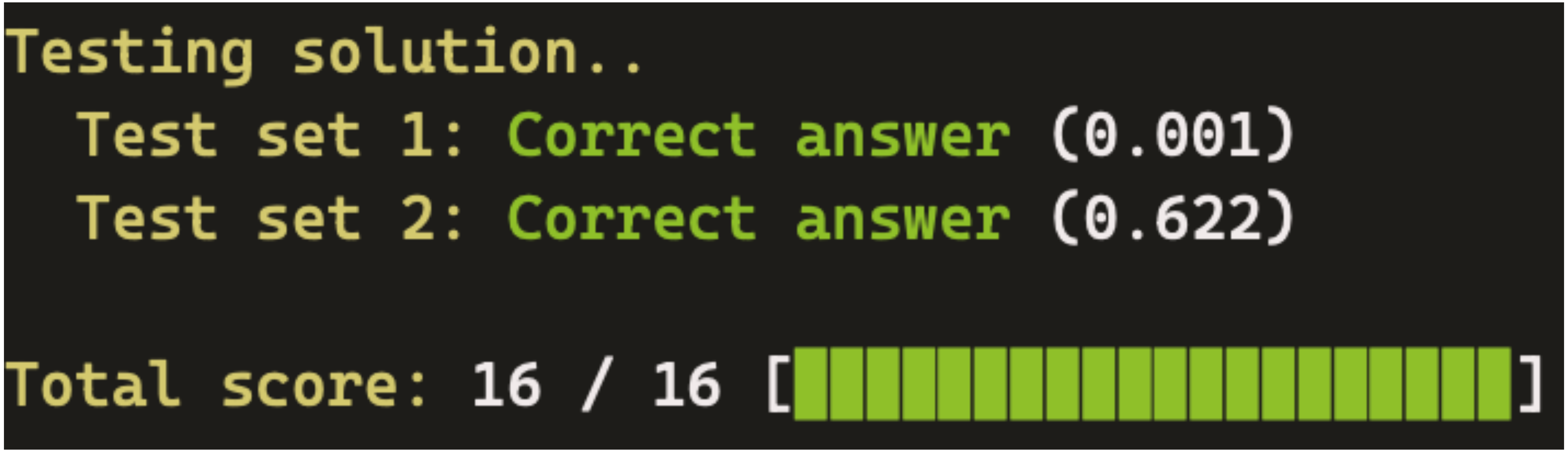
0 to $(n - 1) - L$.

Values for j:

$0 + L$ to $(n - 1)$.

where $L \in \{0, \ldots, n - 1\}$.

CodeExpert Part

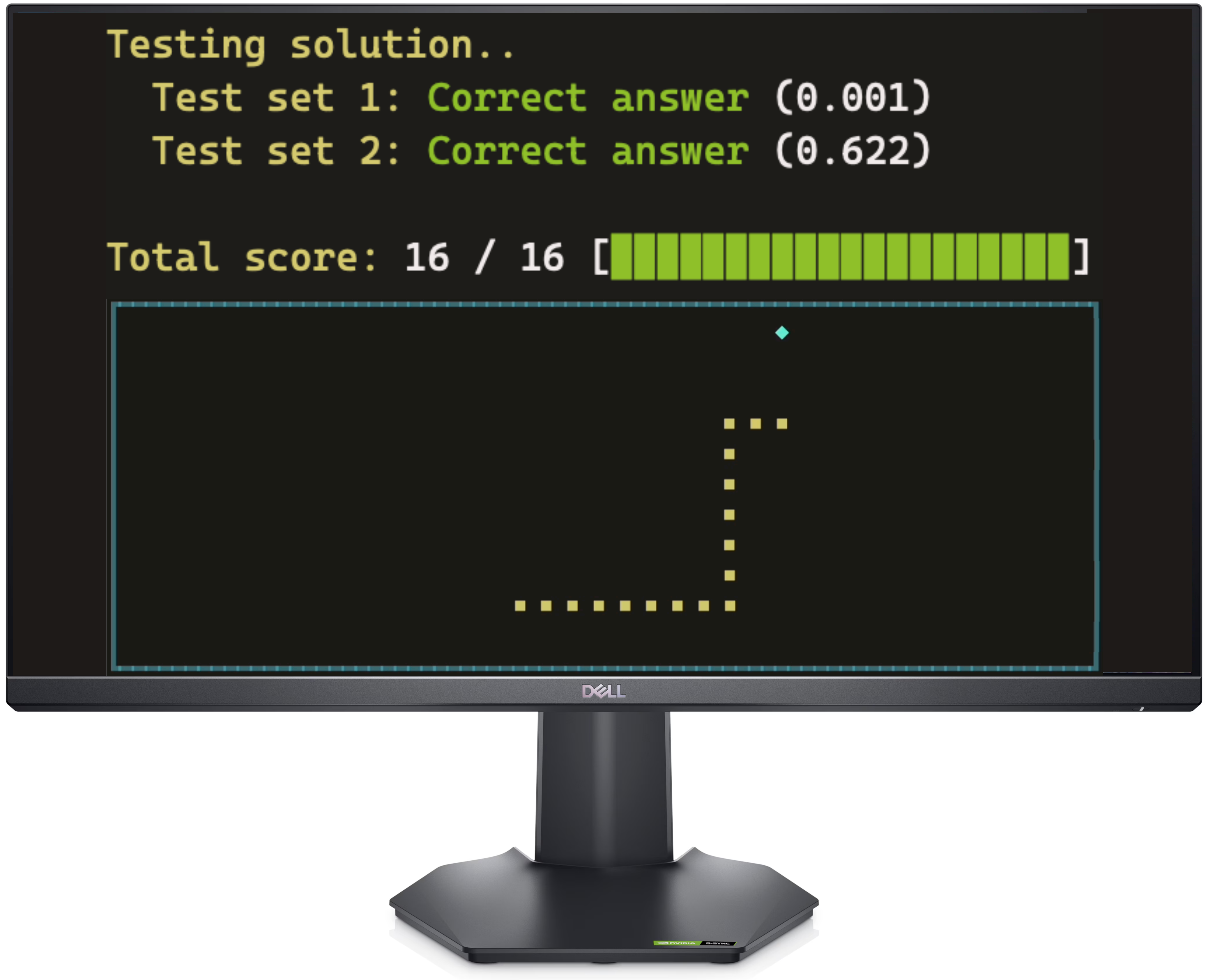


```
Testing solution..
  Test set 1: Correct answer (0.001)
  Test set 2: Correct answer (0.622)

Total score: 16 / 16 [████████████████████]
```

💯💯✅🗣🗣🔝🧠🦾🎓🎓

POV: CodeExpert Exam



```
Testing solution..
  Test set 1: Correct answer (0.001)
  Test set 2: Correct answer (0.622)

Total score: 16 / 16 [████████████████████]
```

(*) Also available as a PDF on my website.

Dynamic Programming Summary

Dynamic Programming (DP) is a problem-solving approach that involves breaking a problem into smaller, overlapping subproblems and building up solutions systematically. To effectively solve a DP problem, follow these steps:

- **Understand the Problem**: Begin by deeply understanding the problem. Work through several concrete examples to get a sense of the subproblems, their dependencies, and how they combine to form the overall solution.

- **Don’t Jump to Bottom-Up**: Avoid starting directly with a bottom-up tabulation approach. Instead, focus on grasping the problem’s structure before choosing an implementation strategy.

- **Define Subproblems and Recurrence**: Identify the subproblems and formulate the recurrence relationships that connect them. These are the core of your solution and must be clear before proceeding.

- **Stress-Test Your Recurrence**: Validate your recurrence by stress-testing it with different examples to ensure its correctness for all input scenarios.

- **Don’t Force a Structure**: Avoid trying to fit the problem into a framework or schema that isn’t suitable. Let the problem naturally guide the structure of your DP approach.

- **Choose an Implementation**: Depending on the problem and recurrence, decide whether to solve it top-down with memoization or bottom-up with tabulation. If using tabulation, think carefully about how to translate subproblem dependencies into a table and the order of filling it.

- **Practice Regularly**: Solving DP problems requires practice. Work through a variety of problems (1D, 2D, and 3D DP) to gain flexibility and confidence. This practice will prepare you to experiment and adapt during exams.

- **Stay Flexible**: If one approach isn’t working, don’t get stuck. Be open to revisiting the problem and exploring alternative strategies.

By following these principles, you’ll develop a strong foundation in dynamic programming and build the skills needed to tackle even the most challenging problems efficiently.