

Dynamic Programming

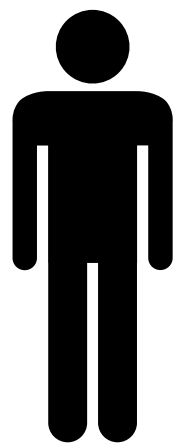
Georg Hasebe

DP in the lecture



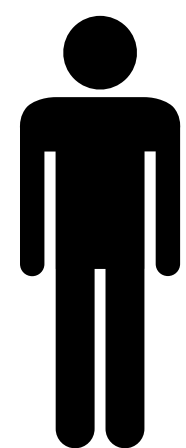
Best

Better

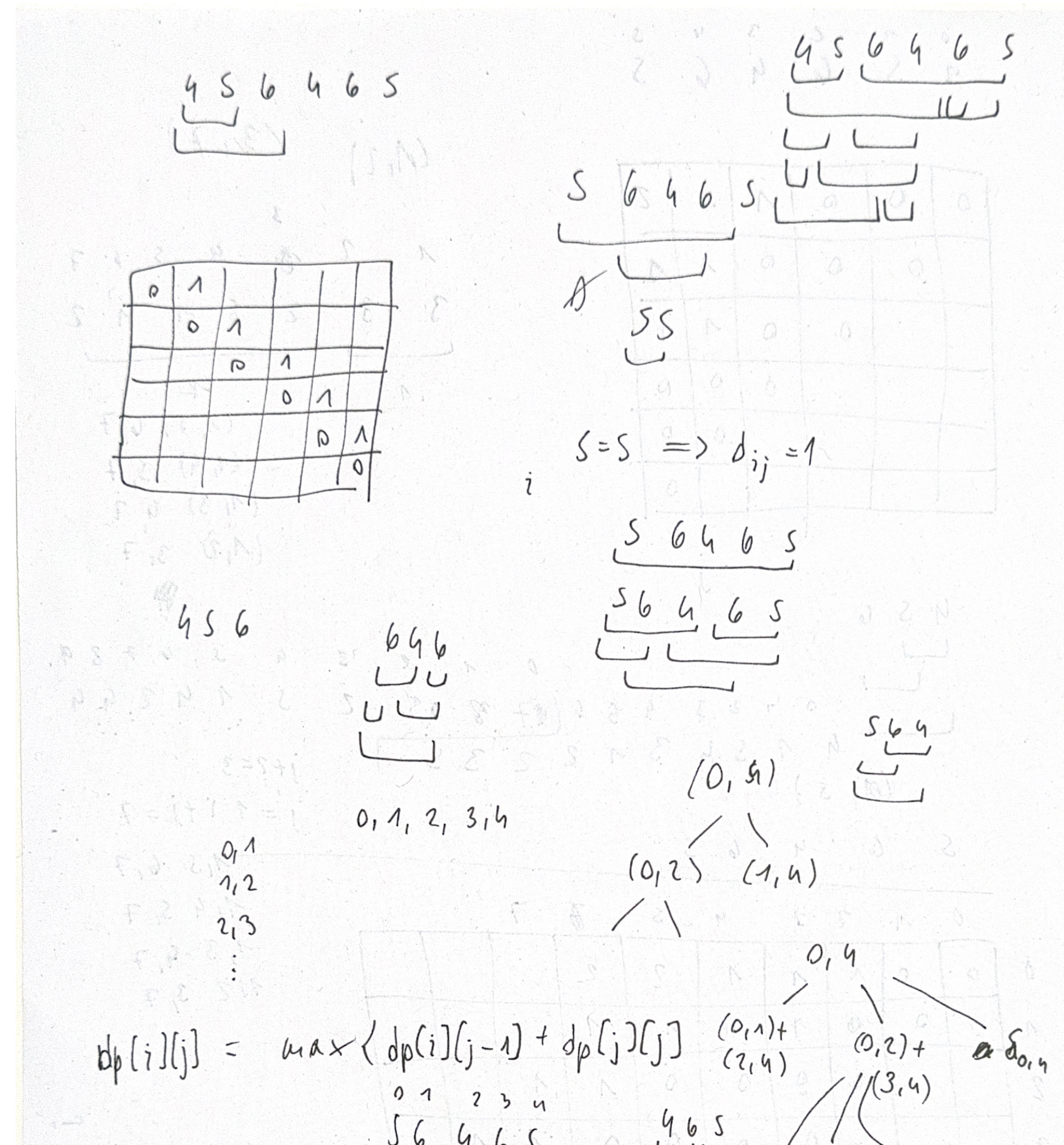


Naive

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.**

Longest Palindromic Subsequence (LPS)

Problem Description

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 $A = \text{"ETZHEEHU"}$, “HEEH” is the longest palindromic subsequence of A , and its length is 4.

Remarks

- This exercise was part of the HS20 Summer CodeExpert Exam
- $O(n^2)$ would give 16/16 points and $O(2^n)$ would give 6/16 points.
- The input array A is indexed from 0, and all characters are capitals.

Do I understand the problem?

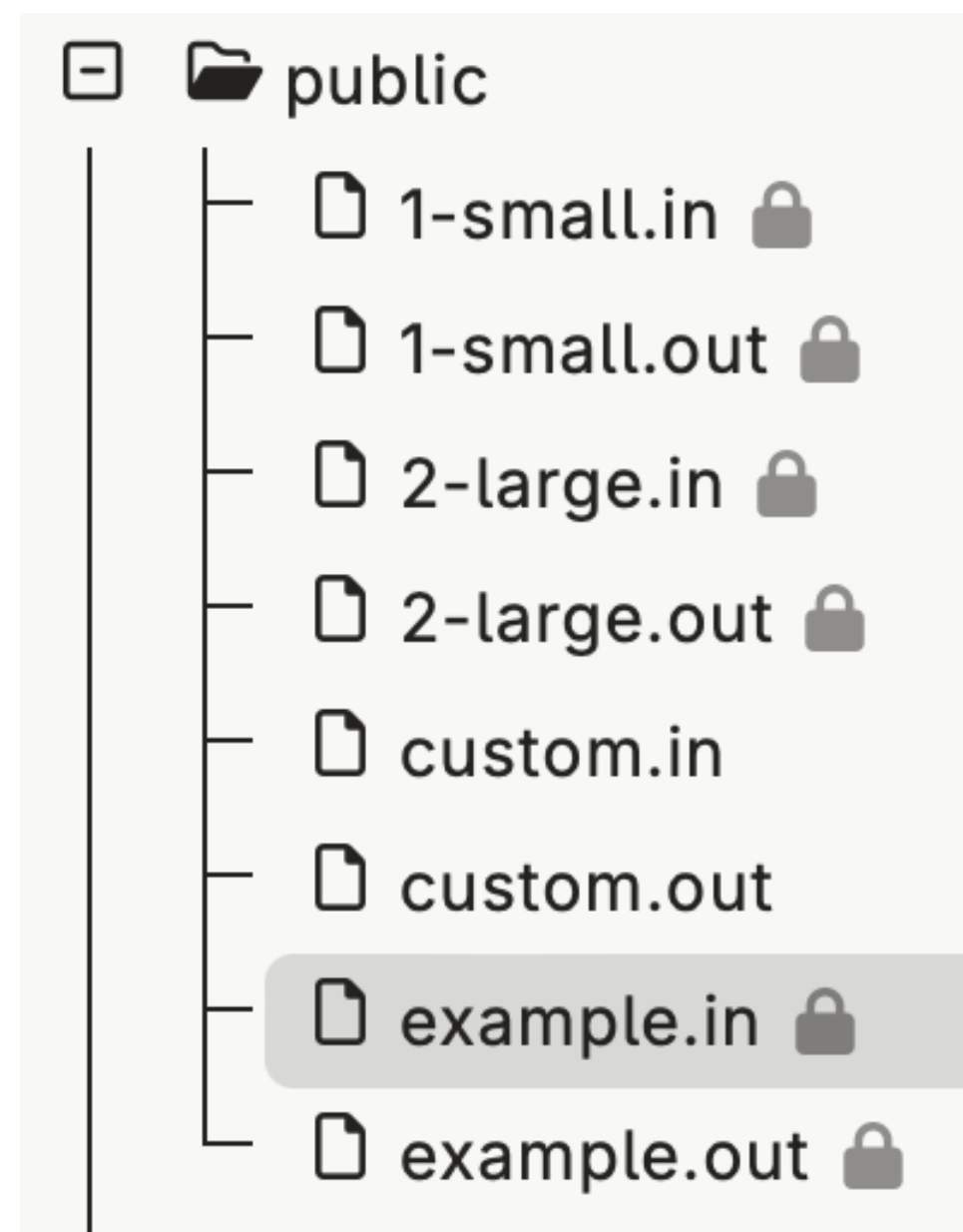
A common mistake is confusing substrings and subsequences

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

The example $A = \text{"ETZHEEHU"}$ 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

Other examples



1	5
2	8
3	"IUIUCHZH"
4	8
5	"RTRREEUR"
6	8
7	"ETZHEEHU"
8	8
9	"URUAATTA"
10	8
11	"CUZURUII"

1	3
2	4
3	4
4	4
5	3

After trying the first two examples and seeing that the second example "RTRREEUR" yields answer 4, I notice that the answer must be "RRRR".

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

Problem Solving

- Often times when solving problems, we rely on solutions of similar problems
- To get better at solving DP problems, we want to practice using different kinds of problems

Similarities?

Exercise 7.1 *1-3 subset sums (1 point).*

Let $A[1, \dots, n]$ be an array containing n positive integers, and let $b \in \mathbb{N}$. We want to know if there exists a subset $I \subseteq \{1, 2, \dots, n\}$, together with multipliers $c_i \in \{1, 3\}$, $i \in I$ such that:

$$b = \sum_{i \in I} c_i \cdot A[i].$$

If this is possible, we say b is a 1-3 subset sum of A . For example, if $A = [16, 4, 2, 7, 11, 1]$ and $b = 61$, we could write $b = 3 \cdot 16 + 4 + 3 \cdot 2 + 3 \cdot 1$.

Exercise 7.4 *String counting (1 point).*

Given a binary string $S \in \{0, 1\}^n$ of length n , let $f(S)$ be the number of times “11” occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{“}\underline{11}10\underline{11}\text{”}) = 3$ because the string contains three 1 that are followed by another 1 (underlined). Given n and k , the goal is to count the number of binary strings S of length n with $f(S) = k$.

Similarities

$$A = [16, 4, 2, 7, 11, 1]$$

Similarities

$A = [16, 4, 2, 7, 11, 1]$

$S = \text{"01011010011"}$

Similarities

- We consider elements of the array or characters of the string one by one from start to finish
- Other examples: Subset Sum, Minimum Editing Distance, Knapsack...

CONCLUSION



we try the same approach on LPS

S = “E T Z H E E H U”

Using this conclusion, I will start familiarizing myself with the problem by considering a concrete example.

The goal is to get a feeling for how the subproblems relate to each other, i.e. how a shorter portion of the string can help me construct a solution for a larger portion of the string.

First Approach

$S = \mathbf{ETHZEBEHU}$

LPS of **E**?

Trivially 1.

LPS of **ET**?

Still 1.

LPS of **ETH**?

Still 1.

...

$S = \mathbf{ETHZEBEHU}$

LPS of **ETHZE**?

For the first time we observe a change in the LPS. Since E and E are equivalent, we have a subsequence EE of length 2. Taking any other character that lies in between EE, we get a palindrome of length 3, e.g. ETE. Thus we have LPS is 3.

What did we learn from this?

Suspicion: If first and last character match, we increase the LPS.

**In order to try and verify our
suspicion we try other examples**

Trying multiple examples is generally a good idea when solving a problem.

Every example has its (sometimes unique) properties that can potentially limit our thinking.

Suspicion: If first and last character match, we increase the LPS.

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

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

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

$$S = \mathbf{AXB} + \mathbf{B}$$

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

Third letter also relevant?

Something feels off.

Something feels off. But what?

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

Consider $S = \mathbf{ETHZEBEHU}$ 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!

Something feels off. But what?

Students often start to solve a DP problem by attempting to build a DP table bottom up.

Possible Reasons:

1. Iterative approach might feel more intuitive than recursive approach.

2.

1. *Dimensions of the DP table*: What are the dimensions of the *DP* table?

2. *Subproblems*: What is the meaning of each entry?

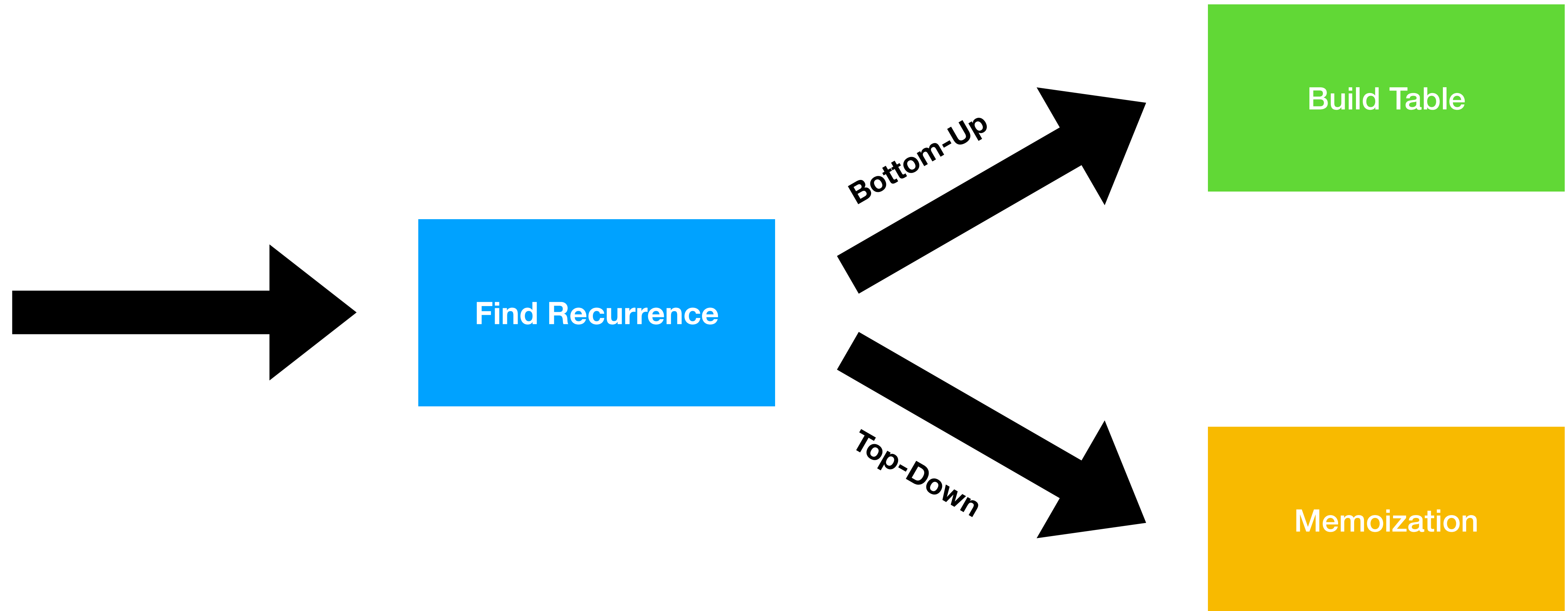
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.

Is this a correct approach?



In my opinion: **NO!**

How else?



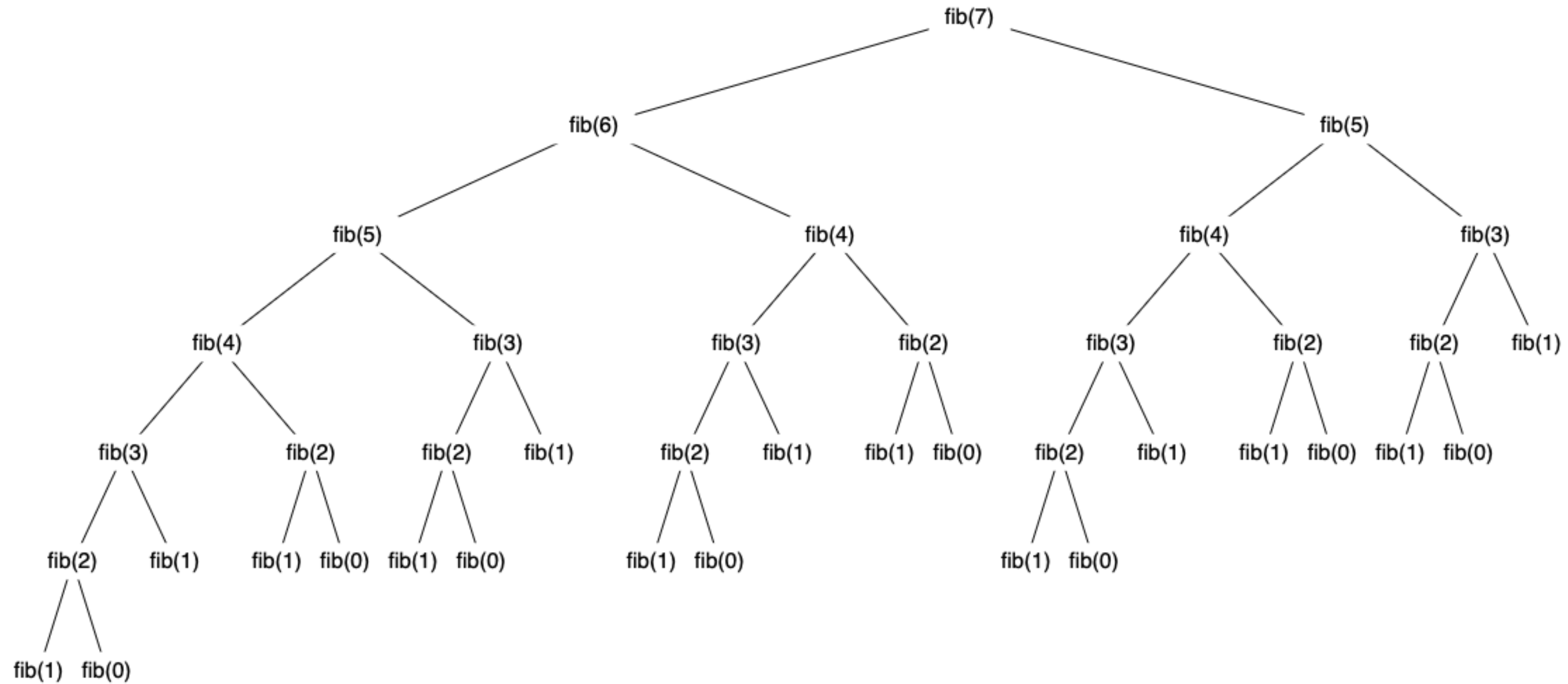
Instead of finding/designing the recurrence by attempting to build a DP table — or trying to do both at the same time — first find/design the recurrence and only then start building a table (or using memoization).

Second Approach

Recall the definition of DP:

DP refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a **recursive manner.**

Recall Fibonacci



$S = \mathbf{ETHZE}$

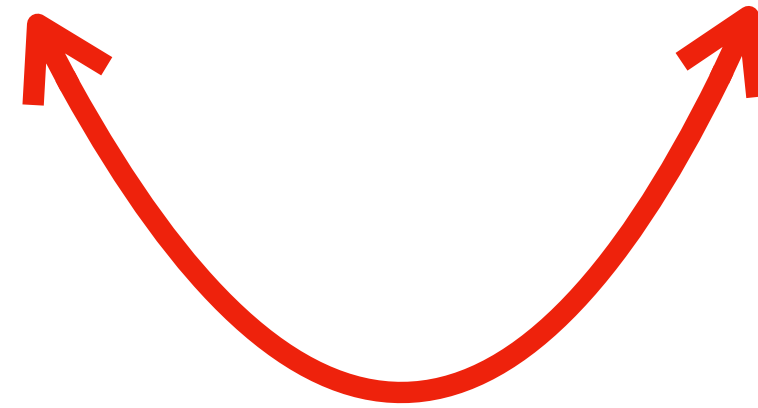
LPS of **ETHZE**?

As we have discussed earlier, since the first and last character match, there was an increase in our LPS.

But increase by how much exactly?

Let's try to visualize the situation.

ETHZE




MATCH!

The matching subsequence, namely **EE**, is of length 2.




ETHZE → E E

Suspicion: If we take the LPS of the blue box, we can simply add 2 to it and we end up with the LPS of ETHZE.

But what if the first and last character do not match?


ETHZX → **E**  **X**

There are only 3 options here:

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

Suspicion: Maybe it's 1.

Suspicion: Case 1.


ETHZX \rightarrow **E**  **X**

1. LPS is LPS of **E** 

Like earlier we try some examples.

ABBB \rightarrow **A**  **B**, but LPS of **ABBB** is LPS of  **B**. Contradiction.


Suspicion: Case 2.

ETHZX \rightarrow **E**  **X**

2. LPS is LPS of  **X**

BBBA \rightarrow **B**  **A**, but LPS of **BBBA** is LPS of **B**  . Contradiction.

Suspicion: Case 3.

ETHZX \rightarrow 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 of our suspicions

$$ETHZ_ \rightarrow E \text{ [redacted] } _$$

- If the first and last character match, we suspect that the LPS will be the LPS of [redacted] + 2.
- If they don't match, then we suspect the LPS will be the greater LPS out of LPS of E [redacted] and [redacted] _.

Let's try to formalize our suspicions.

Formalizing our suspicions

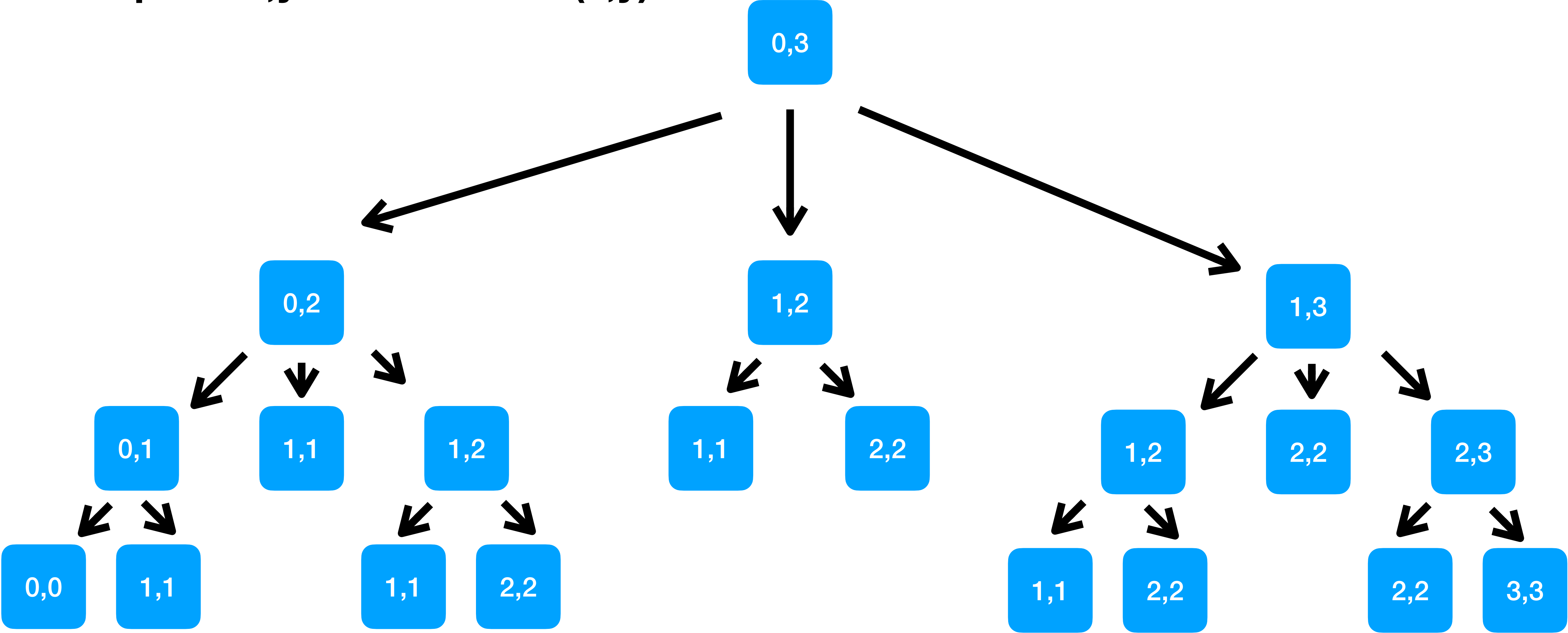
Let $\text{LPS}(i, j) :=$ "LPS of the substring from i to j ."

For example, if $S = \mathbf{ABC}$, then $\text{LPS}(1, 2)$ is LPS of \mathbf{BC} . (We use zero-based indexing)

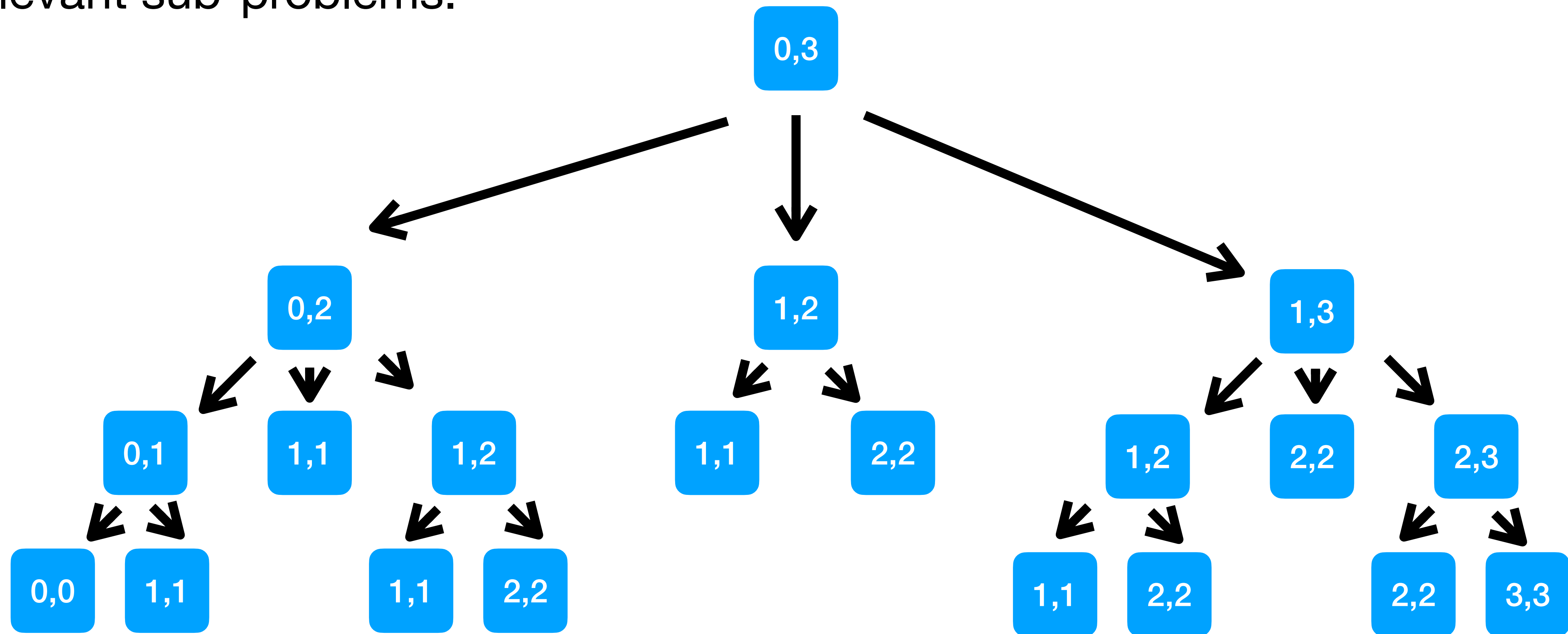
Using our suspicions from earlier, we get:

$$\text{LPS}(i, j) = \begin{cases} \text{LPS}(i + 1, j - 1) + 2 & \text{if character at } i \text{ and } j \text{ match} \\ \max\{\text{LPS}(i, j - 1), \text{LPS}(i + 1, j)\} & \text{else} \end{cases}$$

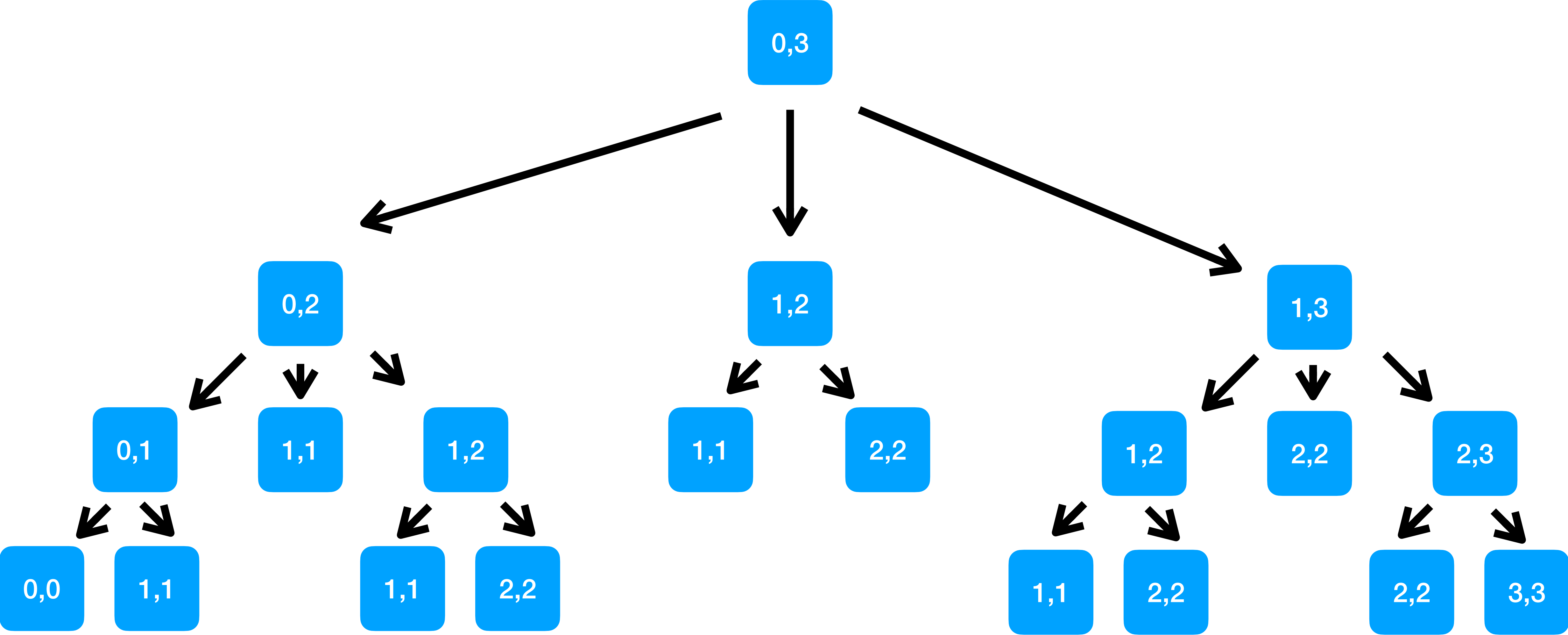
Consider an arbitrary example string $S[0\dots3]$ of length 4. The nodes with caption x,y stand for $LPS(x,y)$.



Notice how the leaves are the base cases. If we build the table starting from the leaves and working our way up to the root $0,3$ we end up calculating $LPS(0,3)$, which is the answer to our problem. By starting at the leaves and working our way up this way, we get access to the answers of all the relevant sub-problems.



Now how exactly do we build a table from this? Notice the labels of each node.



By treating the labels x,y as matrix entries, where x is the row and y is the column, 2D table.



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



Notice how the diagonal holds the base cases and we build an upper triangular matrix starting from the diagonal. The lower triangular matrix is simply left empty.

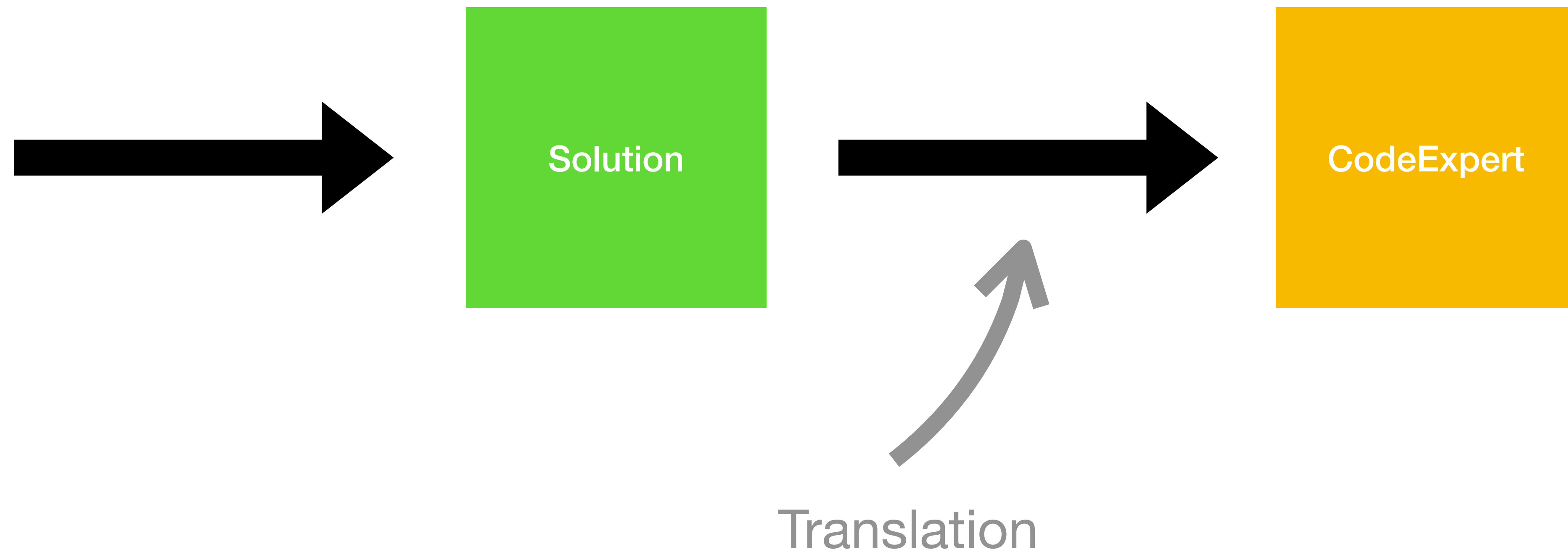
0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

6 ASPECTS

(left as an exercise to the reader)

CodeExpert Part

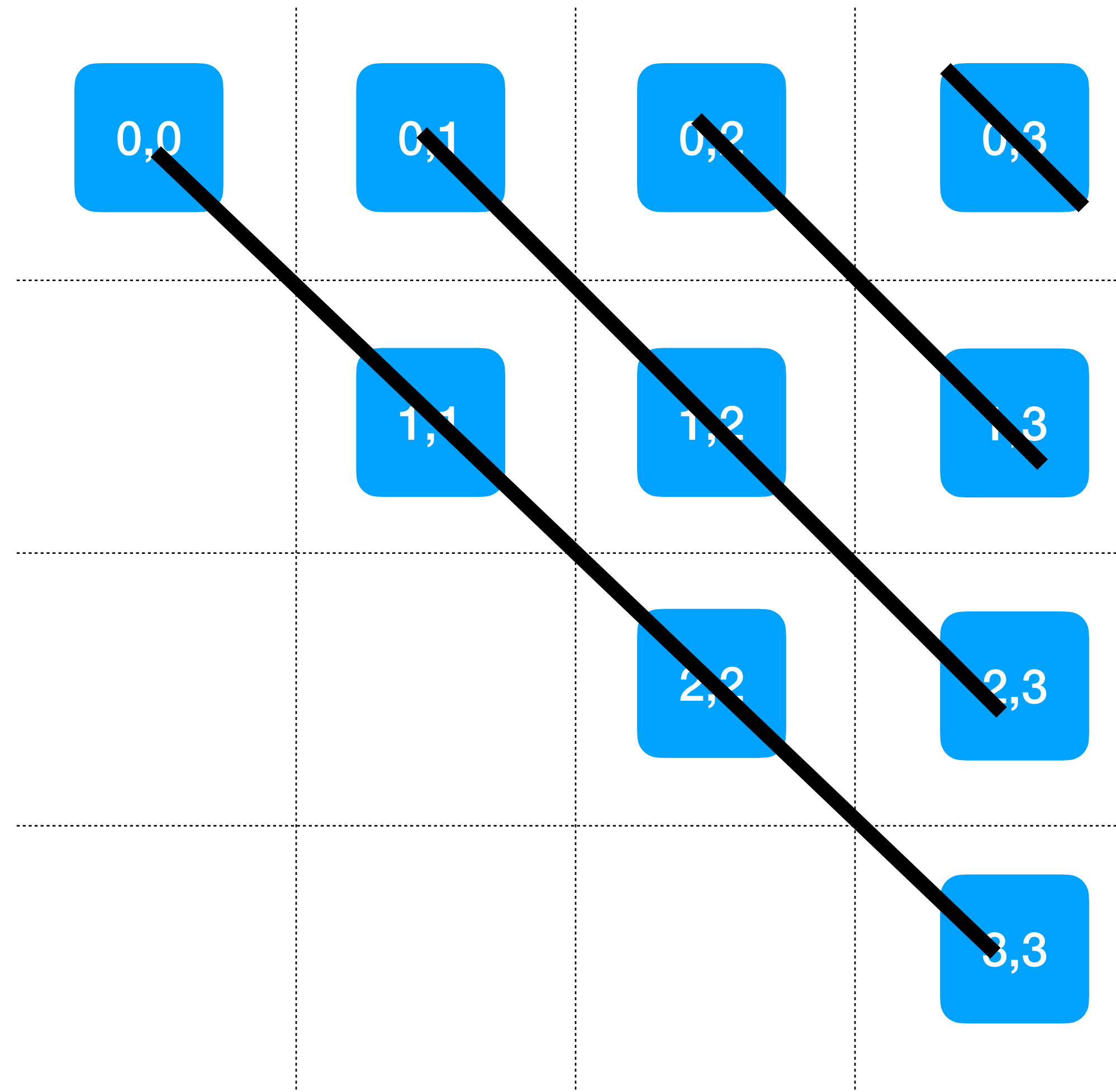
First solve, then code:



But how do we implement LPS?

**How do we implement the upper triangular
matrix DP table?**

How do we implement the upper triangular matrix DP table?



We move diagonal by diagonal!
(Think of the tree structure)

How do we implement the upper triangular matrix DP table?

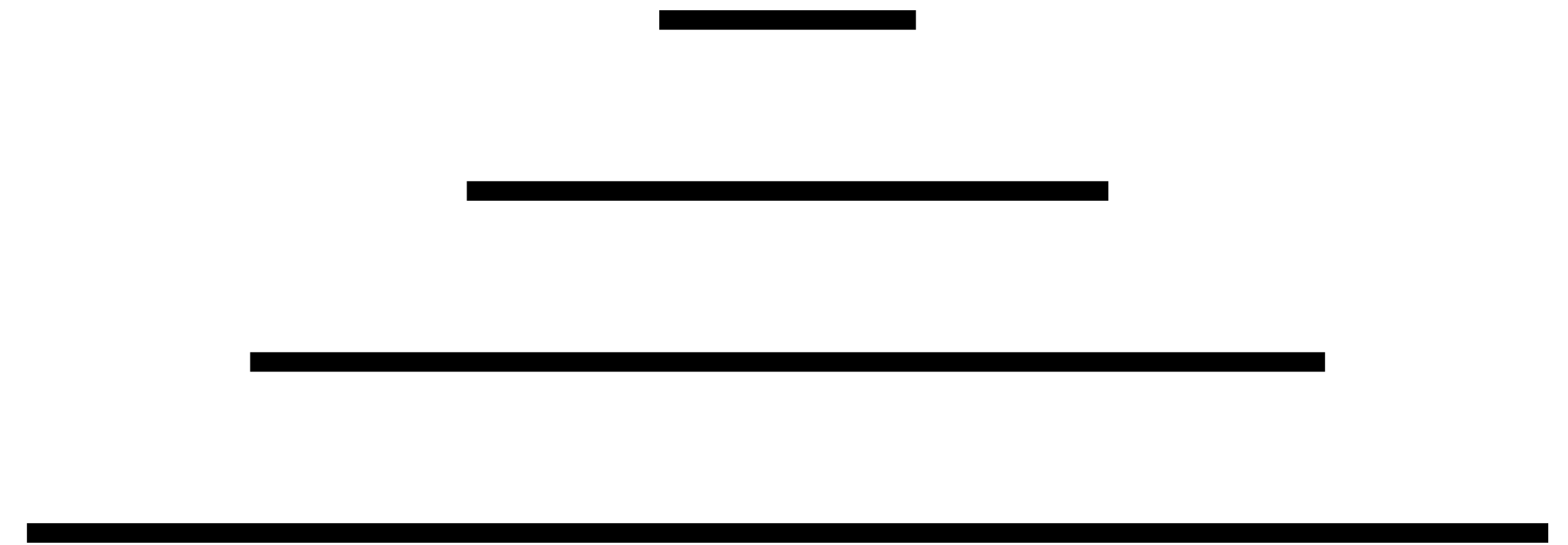
0,0	0,1	0,2	0,3
	1,1	1,2	1,3
		2,2	2,3
			3,3

4

3

2

1



What changes when we go up in diagonals? (with respect to the string we are considering)

Length of the substring!

How do we implement the upper triangular matrix DP table?



How do the indices i, j behave?

Notice i always starts at 0 and gets incremented. On the other hand, j starts off as i but gets increasingly further away from i .

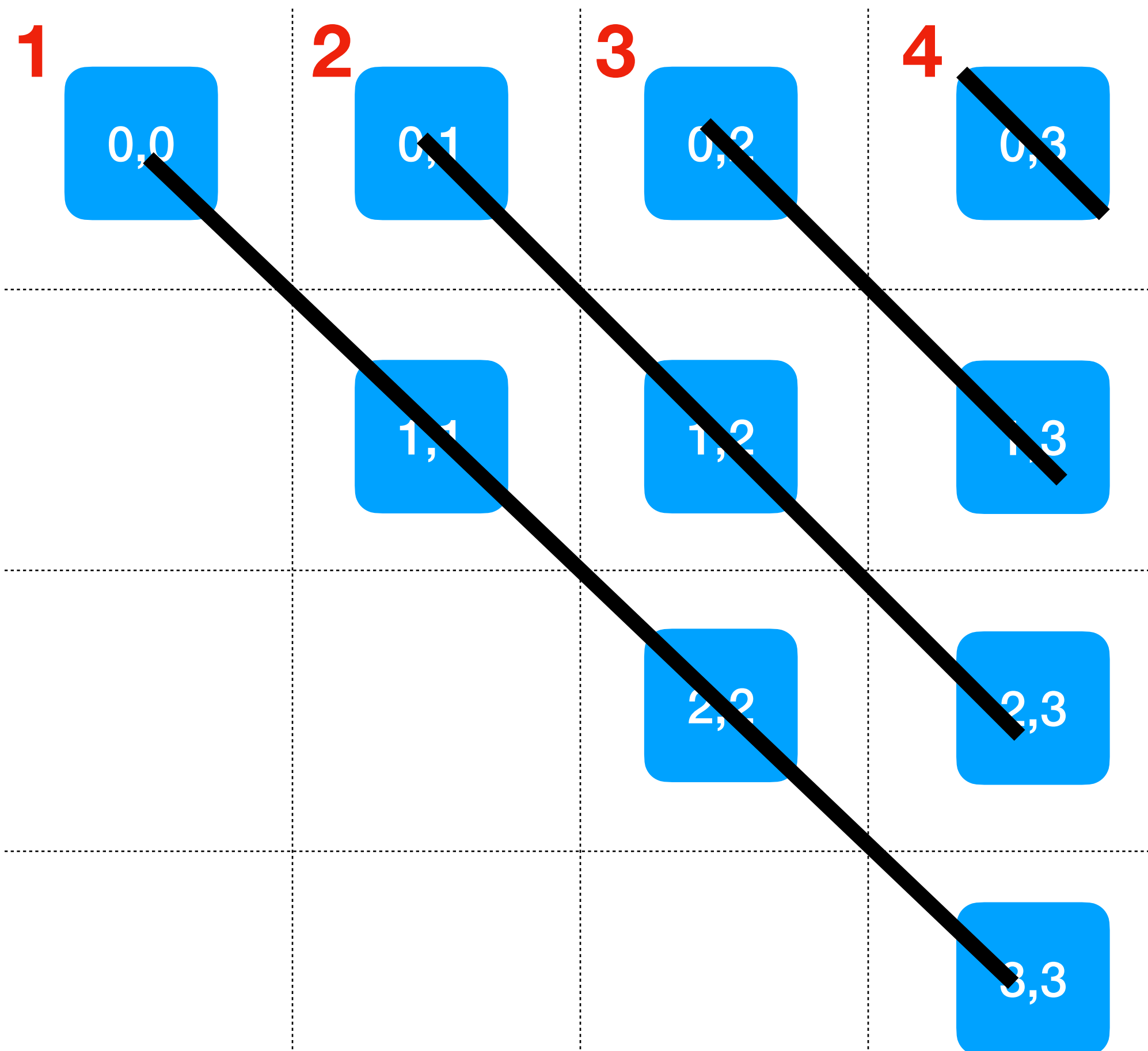
How does that relate to earlier?

i, j are roughly length of substring of the respective diagonal apart.

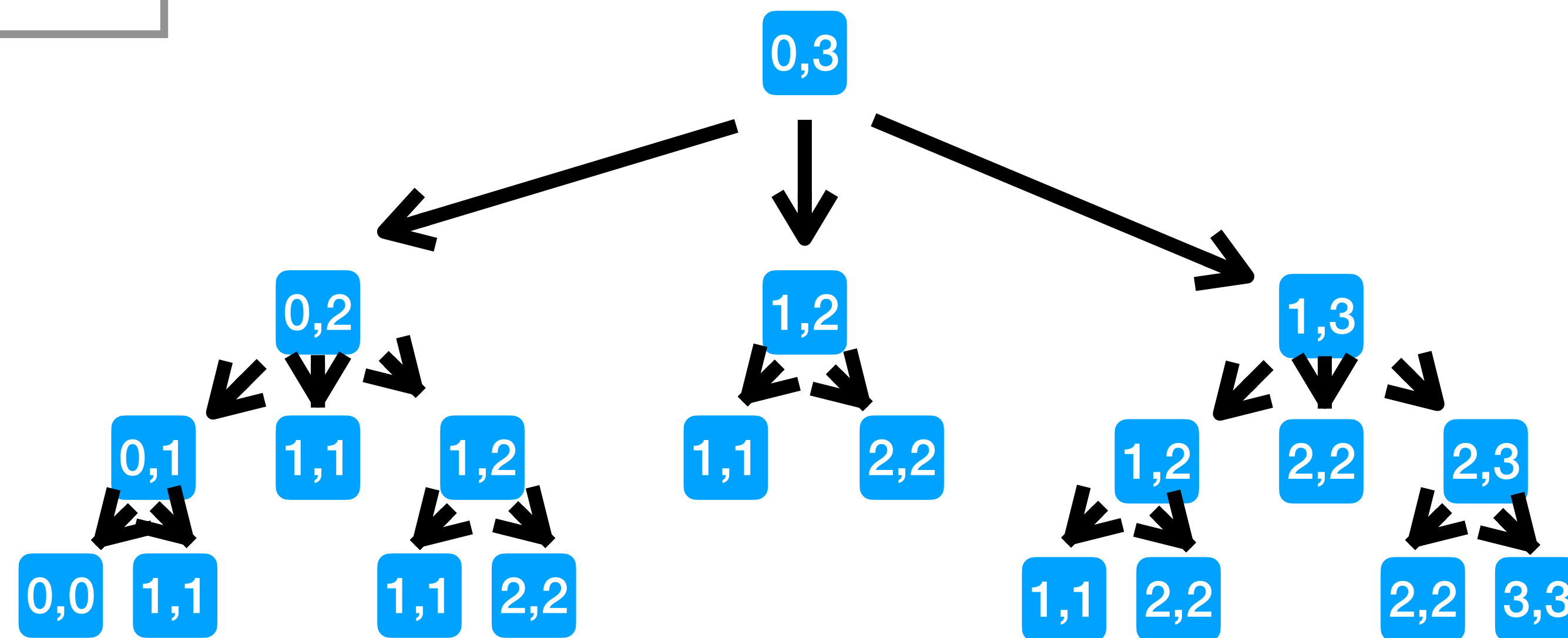
Let's do some coding

i, j are roughly length of substring of the respective diagonal apart.

Length of the substring!



Reference



$$\text{LPS}(i, j) = \text{LPS}(i + 1, j - 1) + 2, \text{ if}$$

letters at i and j match.

$$\text{LPS}(i, j) = \max\{\text{LPS}(i, j - 1), \text{LPS}(i + 1, j)\}$$

if letters at i and j don't match.

Testing solution..

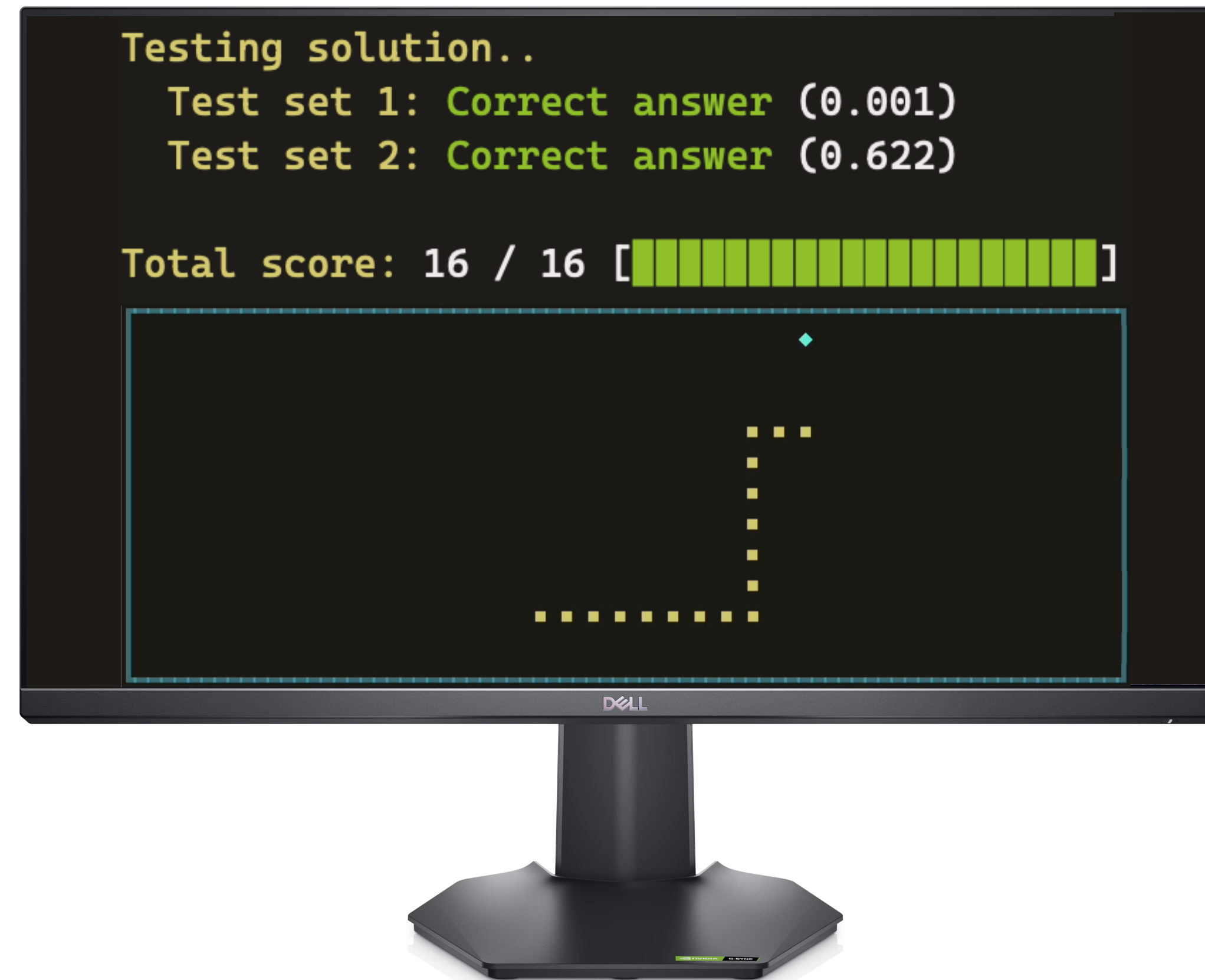
Test set 1: Correct answer (0.001)

Test set 2: Correct answer (0.622)

Total score: 16 / 16 [



POV: CodeExpert Exam



Summary

- We learned how to solve LPS
- We learned how to approach/solve DP problems
- We learned how to implement LPS

**Slides and code will be
available on my website.**