**Dynamic Programming**

This file includes the learning notes of Youtube video (<https://youtu.be/oBt53YbR9Kk>).

**Section I Motivation of having dynamic programming**

Questions such as calculate the 40th number of the Fibonacci sequence and count the number of different ways to move through a 6x9 grid can all be categorised into the dynamic programming (DP) cluster. There are two main parts in DP, Memoisation (记忆化) and Tabulation (列表化).

To understand why we need DP, we need to first settle the concept of algorithm complexity. In Java, the recursive algorithm for Fibonacci sequence can be given as Fig. 1.

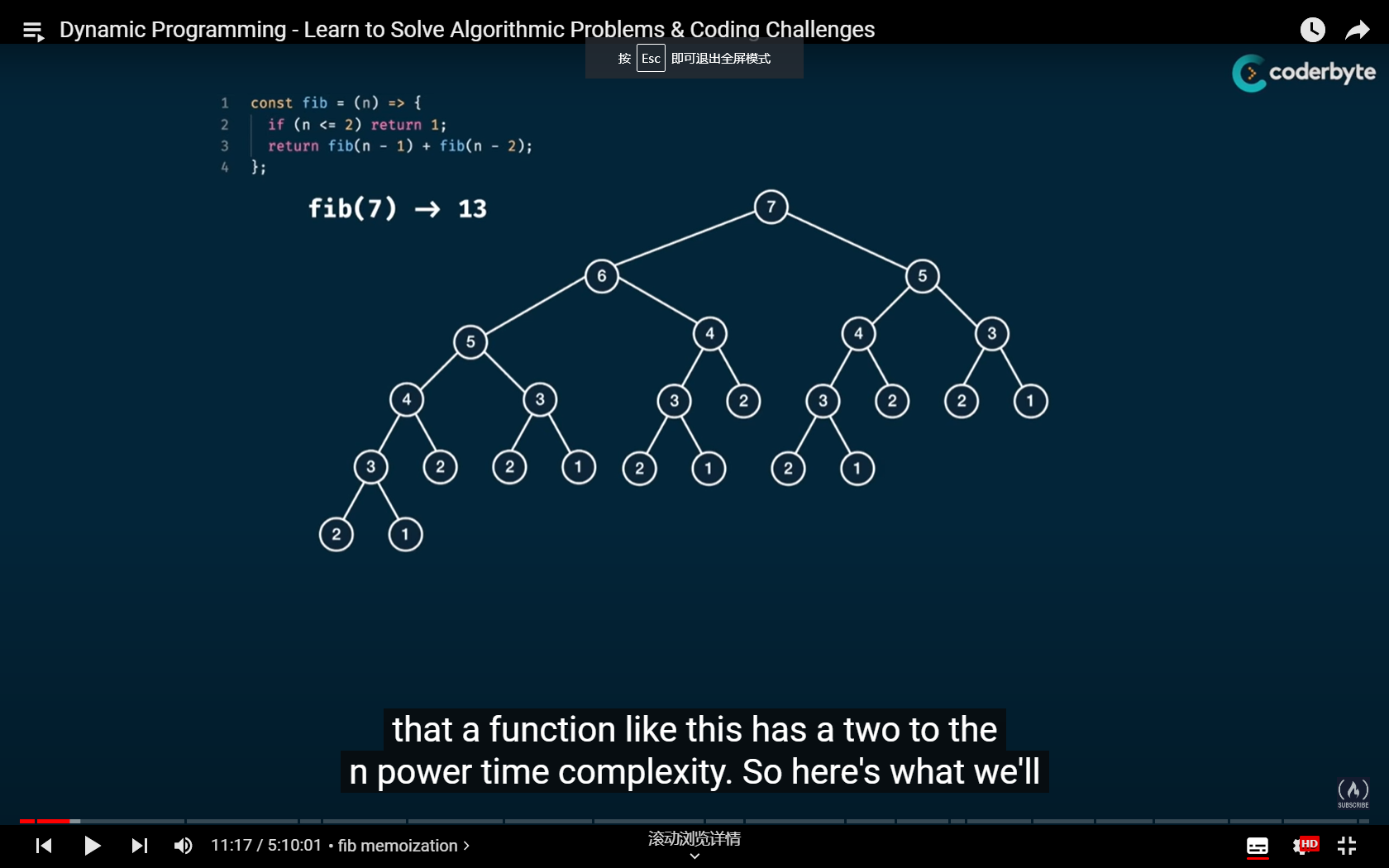


Fig. 1 Illustration recursive algorithm for Fibonacci sequence

However, it seems a bit hard for us to directly consider the complexity of the algorithm. Hence, let’s pick another example. For example, we consider the algorithm in Fig. 2. If we want to obtain the fifth element in the sequence, we need to calculate for 5 times. This indicates that the time complexity of the algorithm is . Similarly, we have that the space complexity is also .

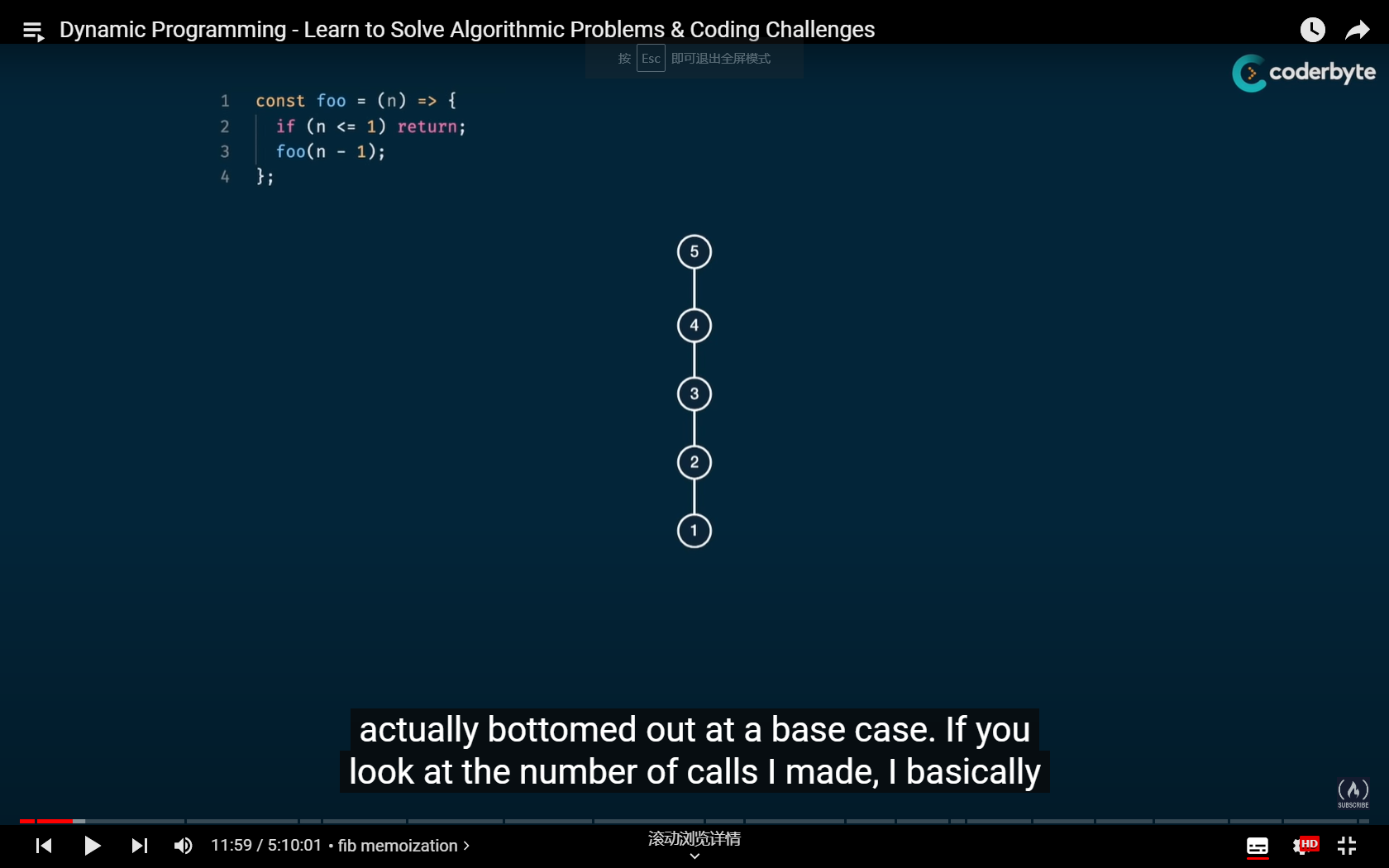


Fig. 2 Illustration of a simple algorithm

If we consider another simple algorithm as shown in Fig. 3, the time complexity is , but in computer science, we treat it the same as . The space complexity is also . If you are careful enough, you may have discovered that the algorithm in Fig. 1 is a combination of the one in Fig. 2 and Fig. 3.

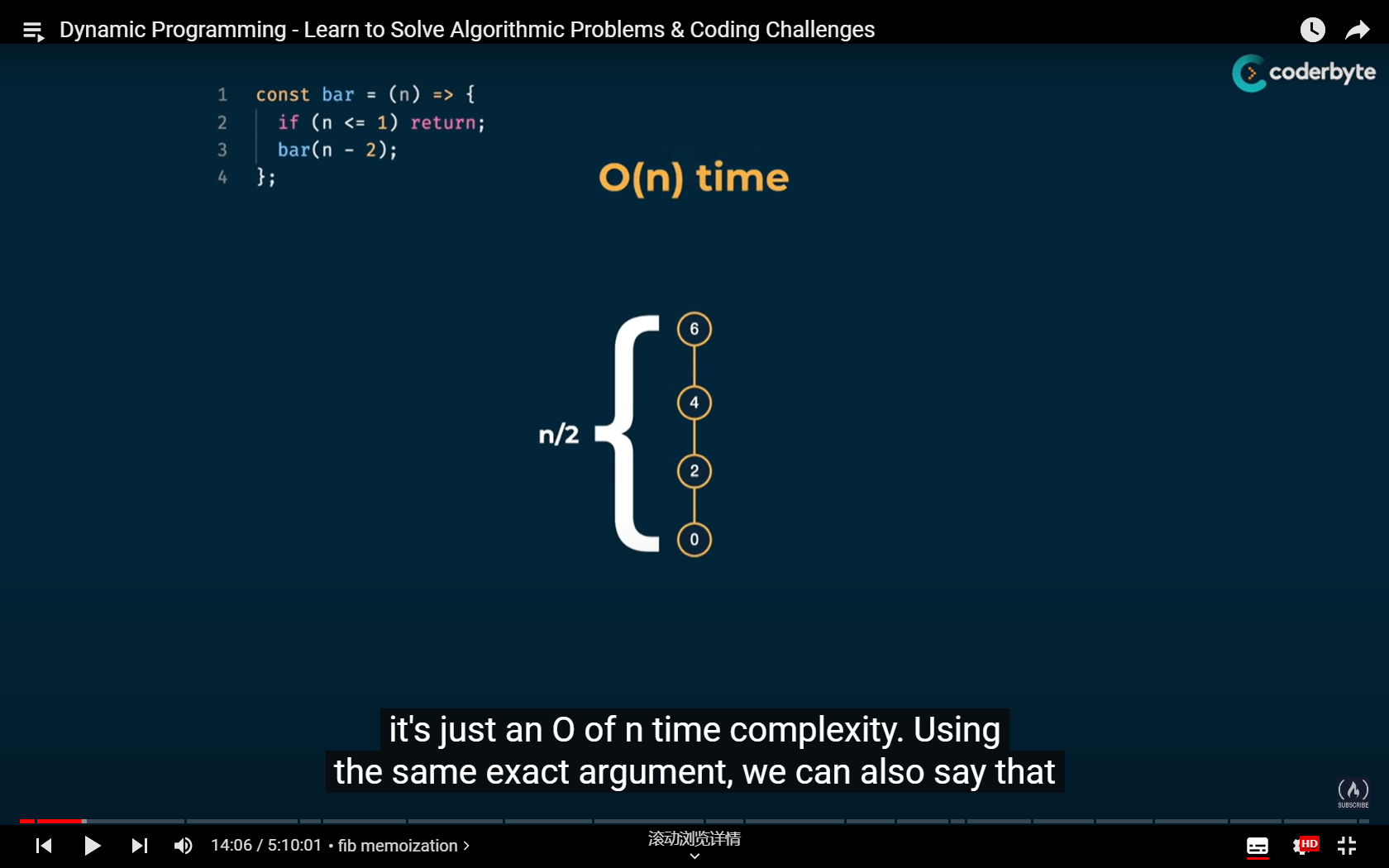


Fig. 3 Another simple algorithm

Then we consider another “Dib” function as illustrated in Fig. 4. Based on previous discussions, we can see that the time complexity of the Fibonacci algorithm is , which is treated as . Hence, if we run the above problem with the simple recursive design, then the complexity of the algorithm is . Meanwhile, the space complexity is only because if we want to calculate the value for , every path we need to calculate contains five element (5-4-3-2-1).

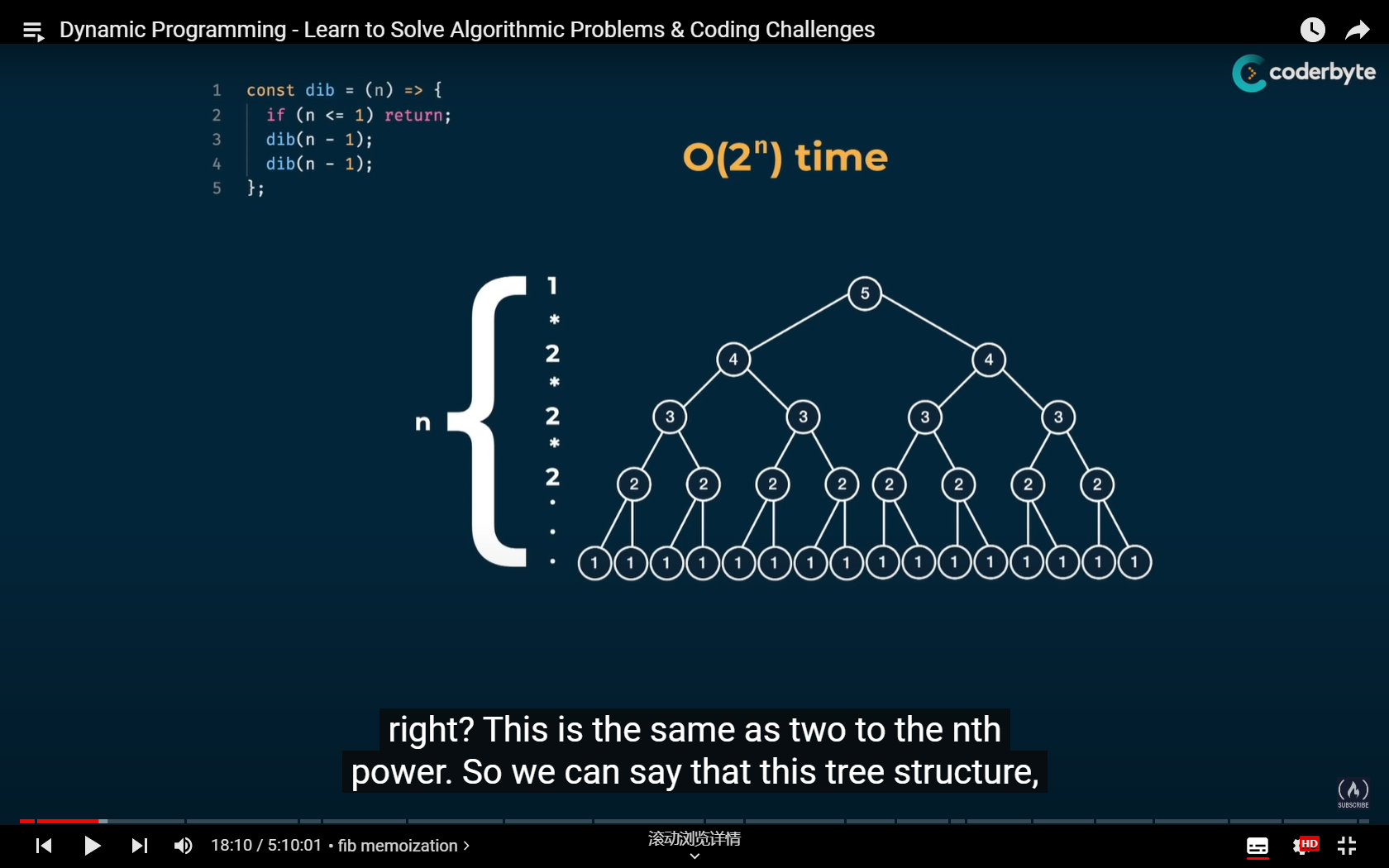


Fig. 4 Complexity of the “Dib” algorithm

Consider another function “Lib” in Fig. 5. Based on our discussion regarding Fig. 4, we know that the time complexity of the algorithm is because the “height” of the algorithm tree is approximately . According to the concept, the time complexity is .

As discussed, the Fibonacci algorithm can be seen as the combination of “Foo” and “Bar”, which could be equivalently expressed as “”. In the same sense, we also have “” and “” and the fact that “” in the sense of time complexity. Therefore, for the Fibonacci algorithm, we have “” in the sense of time complexity. Hence, we have that the time complexity of the recursive Fibonacci algorithm is . And that explains why it takes “forever” for us to calculate the 50th element of the Fibonacci sequence by recursive design.

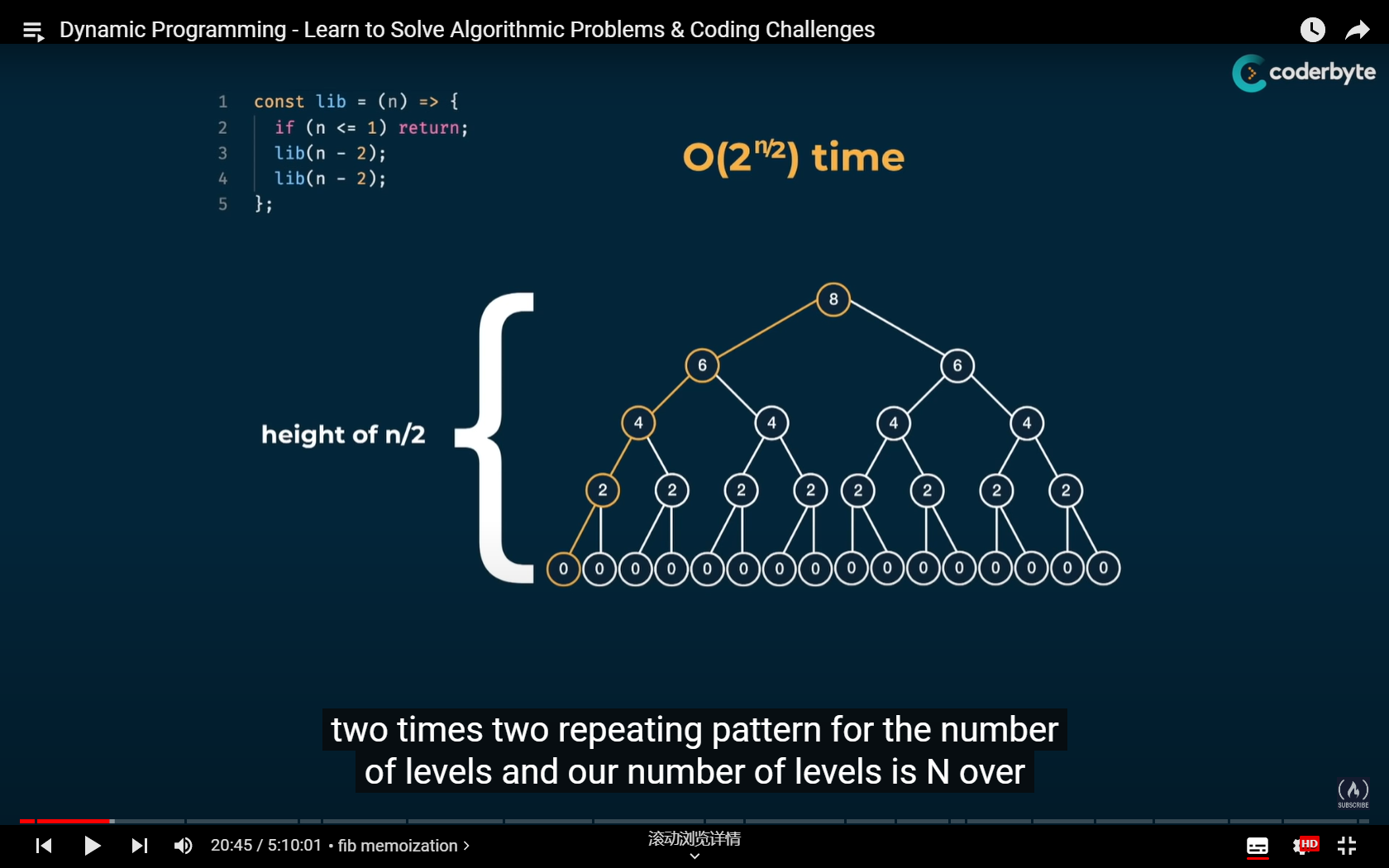


Fig. 5 Complexity of the “Lib” function

Now we have our motivation, to reduce time complexity. To solve this issue, we also need to understand “what’s wrong with the recursive design”. From the algorithm tree in Fig. 6, we can see that a large portion of the calculation is duplicate. Then we have our goal, which is to memorise what we have obtained and trim those unnecessary steps.

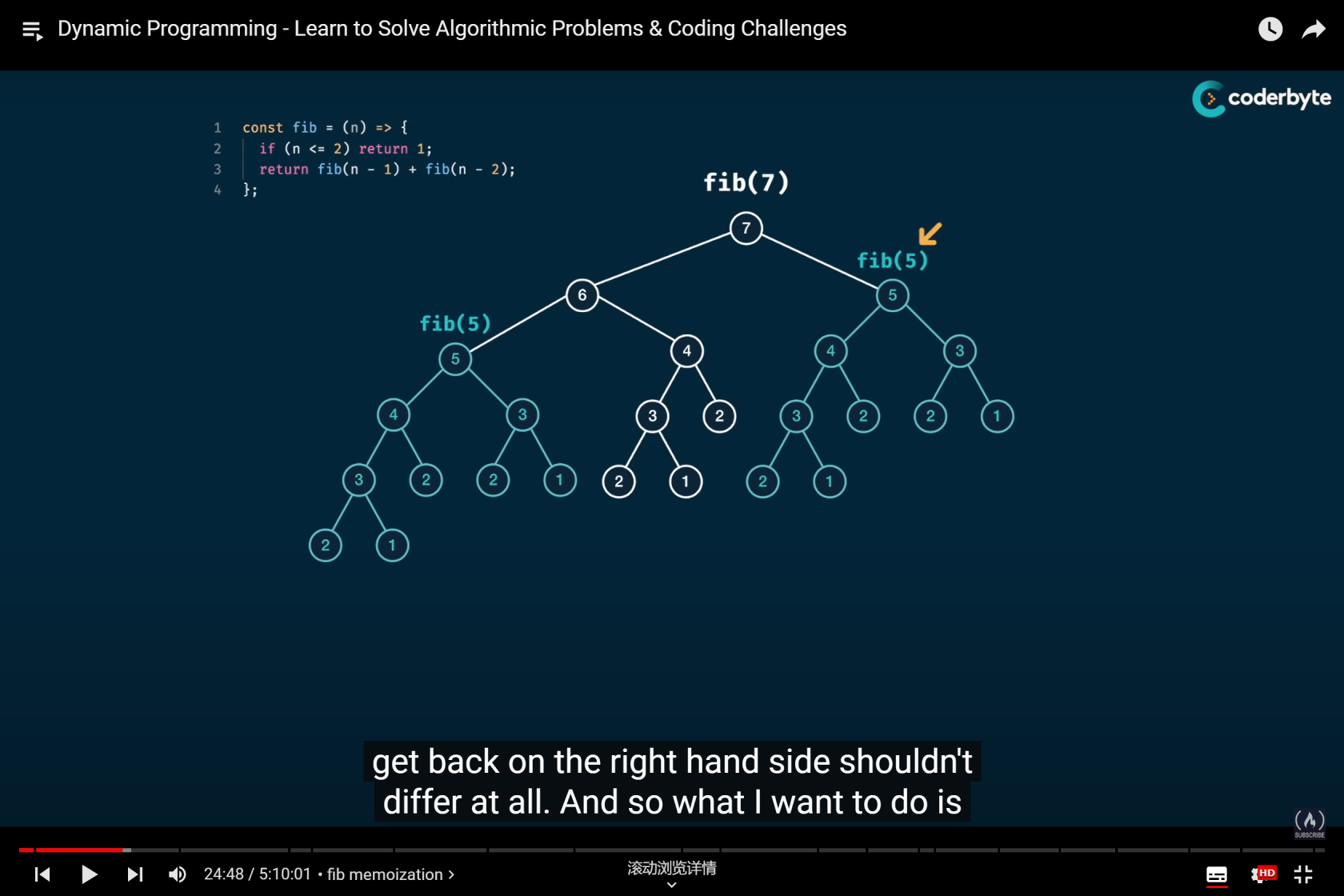


Fig. 6 What’s wrong with the recursive design?

**Section II Memoisation**

Hence, we can introduce one notebook “Memo” into the algorithm so that we can store what we have calculated to avoid duplicate running (see Fig. 7). Now, we can get the values of a Fibonacci sequence in a much faster speed!

Judging by the running speed, we can see that the algorithm design is significantly improved, which means that the algorithm tree of the DP process should be a lot different than the one of the recursive design. In Fig. 8, we present the new algorithm tree of the DP process. You can see that a lot of the unnecessary calculations are omitted, which saves our time. From the figure, we can see that the time complexity and the space complexity of the DP structure are both .

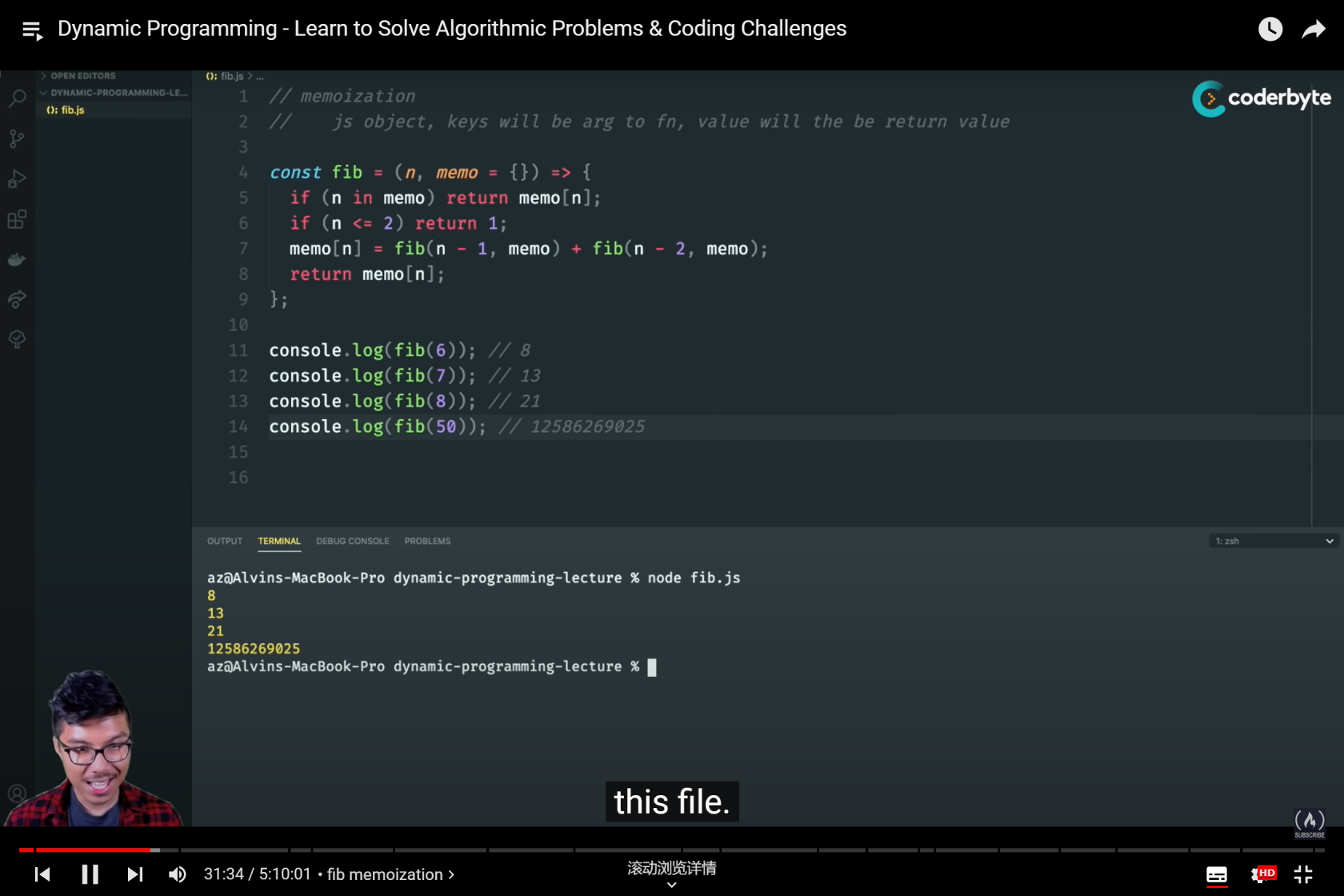


Fig. 8 The updated DP structure

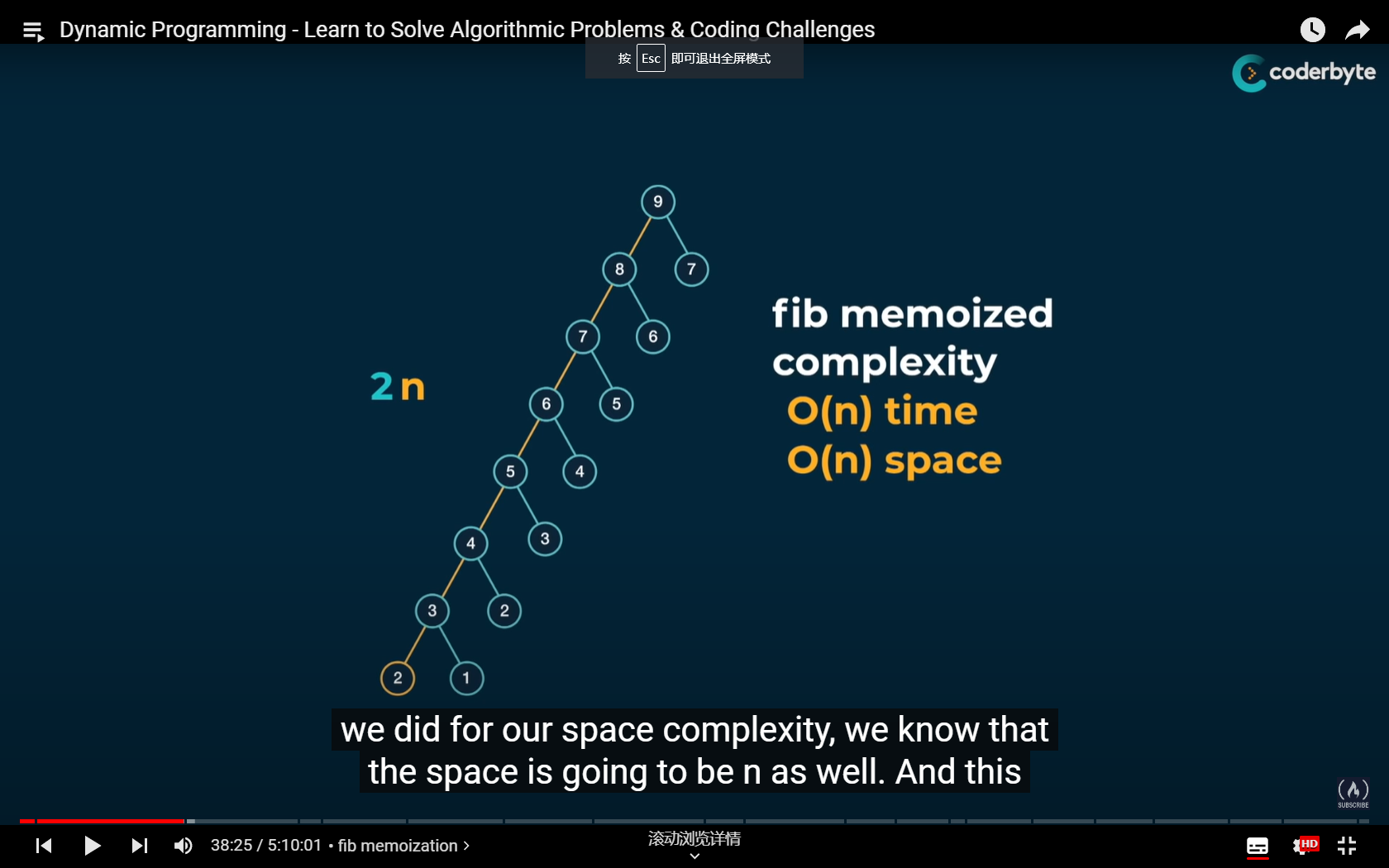


Fig. 9 The new algorithm tree when DP is applied

**Section III Grid traveler problem**

Now we move on to a more complex problem, the grid traveler issue. Suppose you are a traveler on a 2D grid. You begin in the top-left corner and your goal is to travel to the bottom-right corner. You can only move down or right. Then in how many ways can you travel to your destination?

Before going deep into this question, let’s first settle some basic scenarios:

1. When at least one of the row-column dimension is 0, the answer is 0 because we can’t go anywhere.
2. When at least one of the row-column dimension is 1, the answer is 1 because there’s only one way left.

The above two scenarios can be treated as the basic child circumstances that don’t need to be calculated. Similar to the Fibonacci issue, we can see that the time complexity of , where and are row number and column number, respectively. Meanwhile, judging by the height of the tree, we can see that the space complexity is .

Then our main concern is actually the same, to memorise the results we have obtained to avoid duplication. However, the issue is a bit more complex because we have multiple variables that determine the result of one function. Hence, the concept of dictionary is introduced. By creating a dictionary, we can have a one-on-one match between a string key (which includes the variables we have in a string format) and the function value (the number of possible paths from the beginning to the destination).

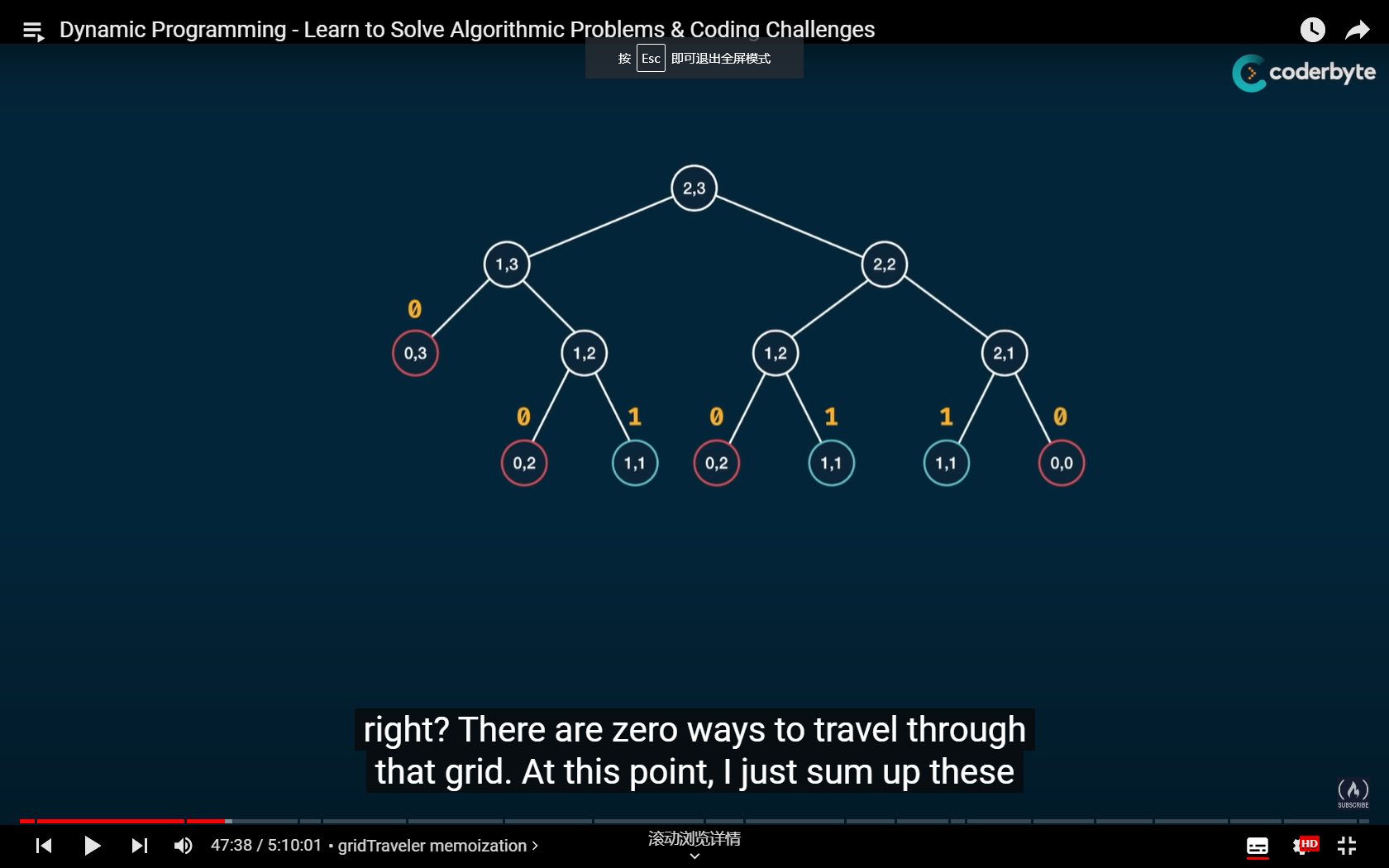


Fig. 10 The algorithm tree when row=2, column=3

By the time we have a set of input (row and column numbers), we can start the irritation by asking if the keys we are looking for already has a corresponding answer. If yes, then we just use the answer directly. If not, then we calculate the current circumstance by adding up the result of two circumstances, when we have one less row, and when we have one less column, which further leads to

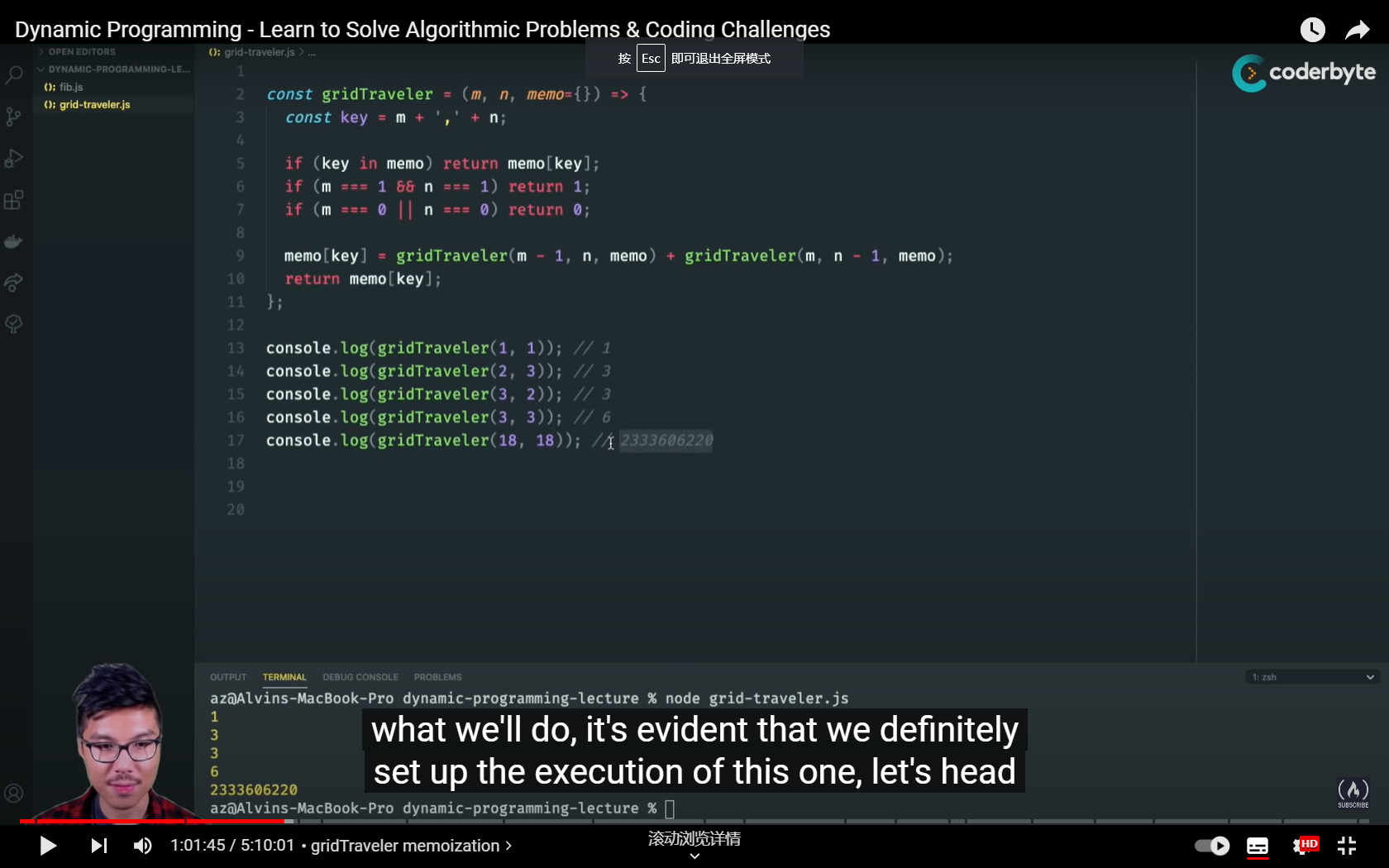


Fig. 11 Algorithm design of the grid traveler issue

In the sense of time complexity, because there will roughly be kinds of situation in total (actually a lot less because we stop calculating when row or column is 1). Hence, we made an improvement from to in time complexity, while the space complexity stays the same.

To sum up, we need to follow a two-step procedure to have a good DP structure.

Step 1: Visualise the problem as a tree and make it work.

Step 2: Introduce a memo object (or a notebook as you may want) to store the answers you have calculated to make your algorithm efficient. Also, always have a basic case that returns value.

**Section IV Can-Sum issue**

Now we consider a new issue, the Can-Sum issue. The question is, suppose we have a target number and a possible selection array , which includes nonnegative independent variables. If we can assume that we can use every independent choice for nonnegative number of times. Then our goal is to develop an algorithm that can help us determine if we can use the combination of arbitrary elements from to obtain the target number .

For example, if we have , then we should obtain a Boolean answer “true” when because we have . However, when because we will run out of choices when we calculate .

Again, we will start with the recursive design, which means that there will be no “memory-based” structure that records the circumstances we have calculated. While we are trying to determine the time complexity, we need to consider the worst-case scenario. As usual, we will use the algorithm tree for illustration. First, we need to know about the height of the tree. Suppose we have element “1” in , then the longest path we have contains nodes in total.

For each node, we will have different choices to check. Hence, each node would split into different nodes in each level. Therefore, the time complexity of the algorithm can be given as . Because the height of the tree is m nodes, then the space complexity is .

**Remark 1**. Regarding the worst-case scenario that we have element “1” in , the longest path should contain nodes because the iteration won’t stop until we hit “0” or something negative. However, if there is a “1” in , then every nonnegative element should result in “True”. Hence, we have that the height of the algorithm tree is at most m nodes high.

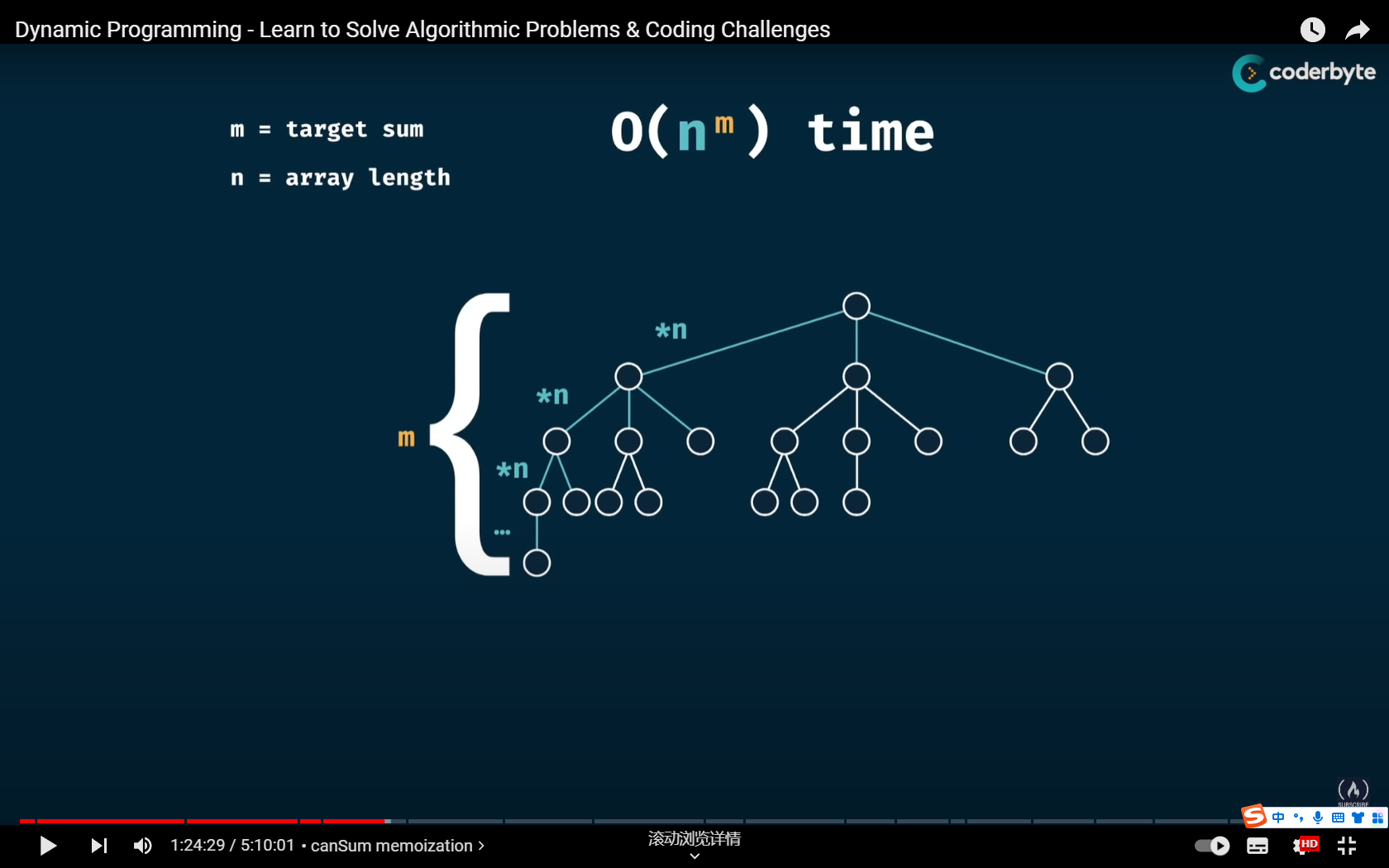


Fig. 12 Recursive design of Can-Sum issue

Just as the video mentioned, the recursive algorithm for case , will take quite a long time to finish. Then it is time for us to include the memory-based structure, our notebook (or I would usually called it the result dictionary).

Then we have the DP-based design in Fig. 13. Regarding the time complexity of the program, because the recursive calculation will not stop until we hit 0, then there should be at most target numbers we need to check. For each possible target number, the result is also determined by its child scenarios (in specific, the results of , ,…,). Hence, there should be about possible scenarios we need to check. Then the time complexity should be . The space complexity should be the same because we only need to store the result of scenarios at one time to ensure we can have the correct result.

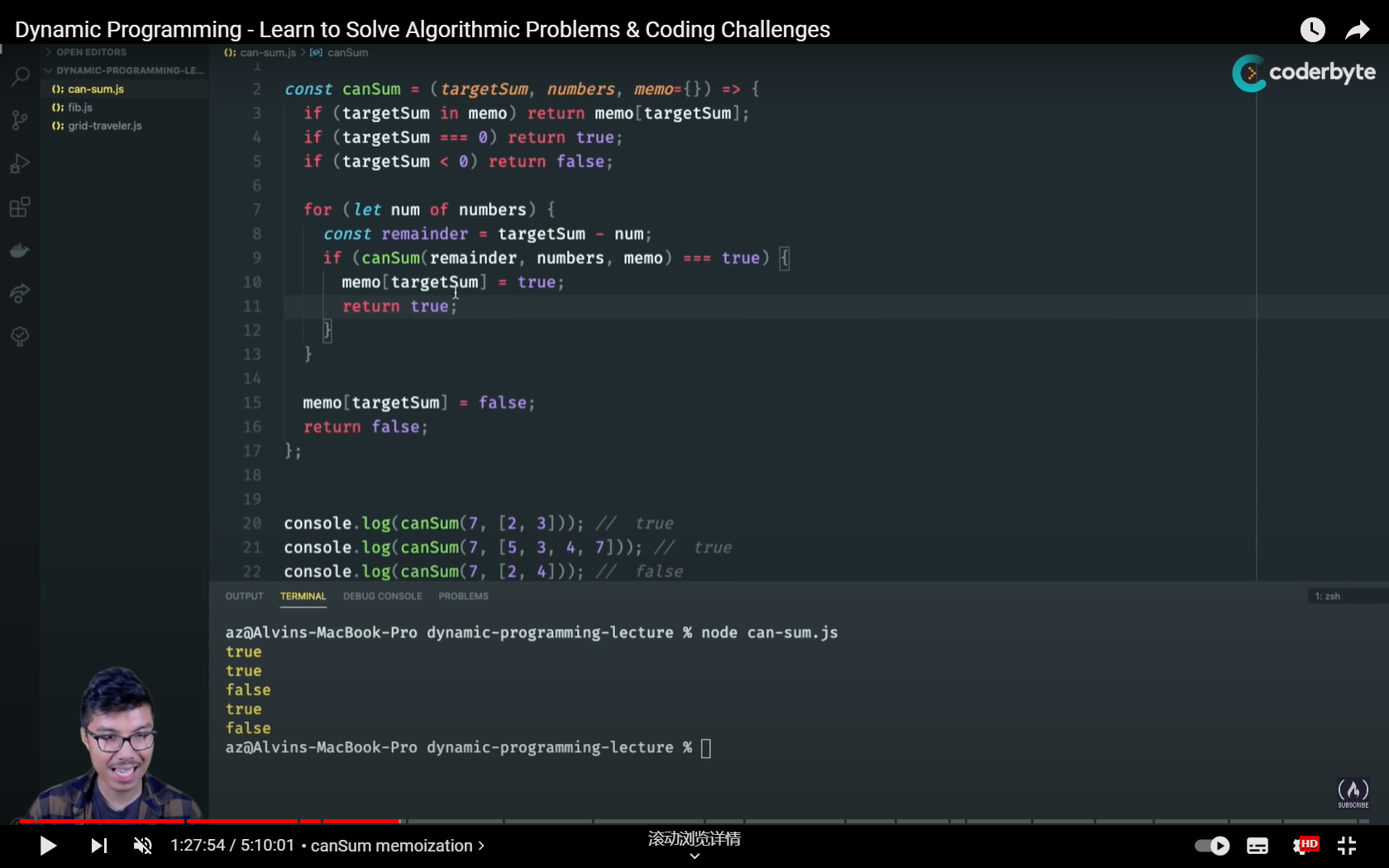


Fig. 13 The DP-based algorithm design

**Section V How-Sum Problem**