**Introduction to Dynamic Programming**

Let’s go over the Dynamic Programming pattern, its real-world applications, and some problems we can solve with it.

**About the pattern**

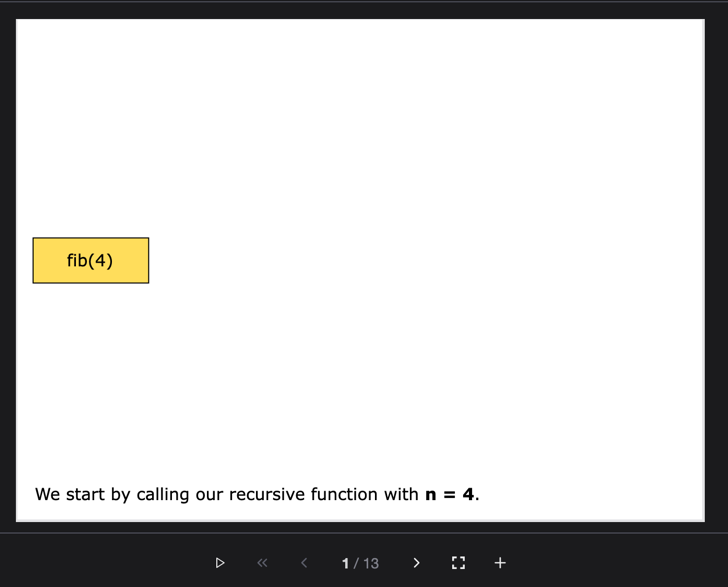
Many computational problems are solved by recursively applying a divide-and-conquer approach. In some of these problems, we see an **optimal substructure**, i.e., the solution to a smaller problem helps us solve the bigger one.

Let’s consider the following problem as an example: Is the string “rotator” a palindrome? We can start by observing that the first and the last characters match, so the string *might* be a palindrome. We shave off these two characters and try to answer the same question for the smaller string “otato”. The subproblems we encounter are: “rotator”, “otato”, “tat”, and “a”. For any subproblem, if the answer is *no*, we know that the overall string is not a palindrome. If the answer is *yes* for all of the subproblems, then we know that the overall string is indeed a palindrome.

While each subproblem in this recursive solution is distinct, there are many problems whose recursive solution involves solving some subproblems over and over again (overlappingsubproblems). An example is the recursive computation of the *nth* Fibonacci number. Let’s review the definition of the Fibonacci series:

A black background with white text

AI-generated content may be incorrect.

Here’s the call tree for the naive recursive solution to this problem, with n=4n=4. A screenshot of a computer screen

AI-generated content may be incorrect.A screenshot of a computer screen

AI-generated content may be incorrect.A screenshot of a computer screen

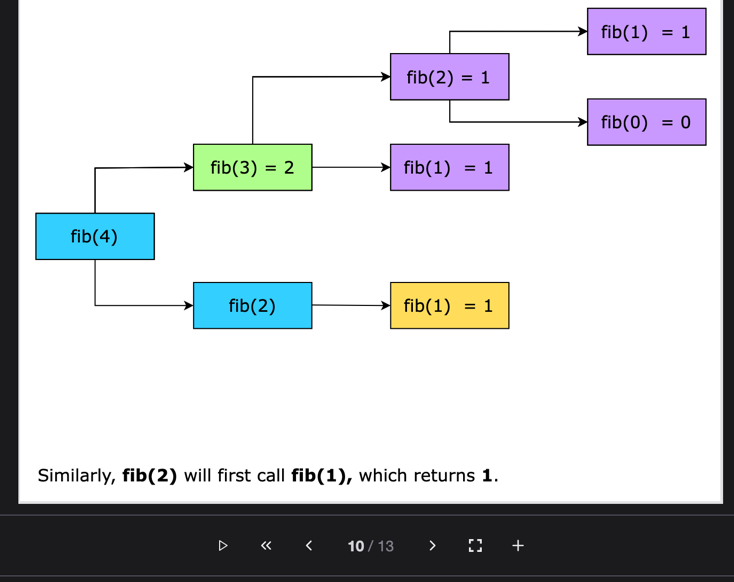
AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

We can see above that the subproblem fib(2)fib(2) is evaluated twice, fib(1)fib(1) is evaluated thrice, and fib(0)fib(0) is evaluated twice. These are repeated or overlapping subproblems. As every subproblem is reevaluated every time it appears, the naive recursive implementation of this solution will have exponential time complexity.

An optimization of such a recursive solution would be to store and reuse solutions to subproblems, reducing the time complexity to polynomial time. Such an approach is called dynamic programming (DP).

We’ve discussed that we need to save the computations, but how can we save and use them? We use the following two approaches that primarily save our computations by reusing previous calculations of subproblems:

* **Top-down approach:** It is a recursive approach that stores the results of redudant function calls to avoid repeating calculations for the same subproblems.
* **Bottom-up approach:** It is an iterative strategy that systematically fills a table with results of subproblems to solve larger problems efficiently.

**Top-down approach**

The top-down approach is also known as memoization. It is usually implemented as an enhancement of the naive recursive solution. It uses recursion to break down larger subproblems into smaller ones. The smallest one is solved and the result is stored in a lookup table for use in computing larger subproblems.

To take advantage of the stored results of subproblems, in every call, the top-down recursive function first checks if a solution to a subproblem already exists. If it does, the result is fetched from the lookup table instead of making a recursive call to compute it. Otherwise, the recursive call is made. As we can see in the illustration below, using the top-down approach, we are able to avoid recomputing the subproblems fib(2)fib(2), fib(1)fib(1), and fib(0)fib(0).

Compared to the naive recursive solution, the top-down approach takes up additional space in memory because it stores intermediate results in a lookup table.

**Bottom-up approach**

The bottom-up approach is also known as tabulation. In this approach, the smallest problem is solved first, the results saved, and then larger subproblems are computed based on the evaluated results. In contrast to the top-down approach, which uses recursion to first break down a larger problem into smaller subproblems, the bottom-up approach starts by solving the smallest subproblem, and then iterates progressively through larger subproblems to reach the overall solution.

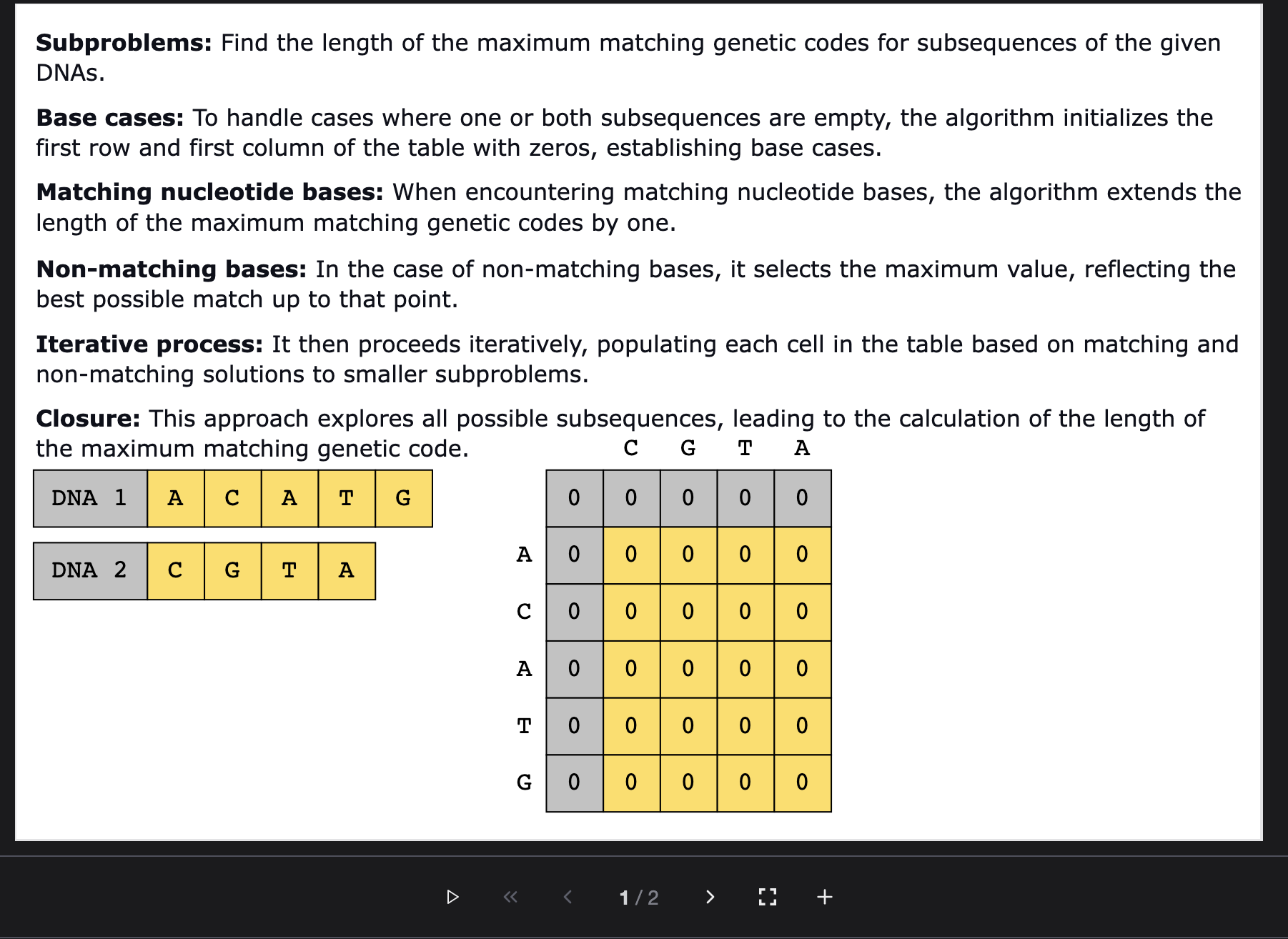
We start by initializing a lookup table and setting up the values of the base cases. For every subsequent, larger subproblem, we simply fetch the results of the required preceding smaller subproblems and use them to get the solution to the current subproblem.

For example, to compute the Fibonacci series, we first set up the two base cases, fib(0)fib(0) and fib(1)fib(1), and then proceed to calculate the larger subproblems:

A black background with white text

AI-generated content may be incorrect.

The following examples illustrate some problems that can be solved with this approach:

**Maximum matching genetic code:** Given two DNA structures having nucleotide bases, i.e., Adenine (A), Guanine (G), Cytosine (C), and Thymine (T), find the length of maximum matching genetic codes.

A screenshot of a computer

AI-generated content may be incorrect.