**CS469 Data Structures and Algorithms**

**HOS09 Dynamic Programming**

05/24/2023 Updated by Christopher Sharp

02/29/2024 Reviewed by Anh Nguyen

09/29/2024 Reviewed by Shahid Khan

School of Technology and Computing (STC) @ City University of Seattle (CityU)

**Before You Start**

* The document’s examples are written in Python. Please finish the Python tutorial in the Module00 folder before you start the assignment.
* Some steps are not explained in the tutorial**.** If you are not sure what to do:
  1. Consult the resources listed below.
  2. If you cannot solve the problem after a few tries, ask TA for help.

**Learning Outcomes**

Students will be able to:

* Understand Dynamic Programming
* Implement Dynamic Programming

**Resources**

* Bhargava, Y, A. (2016). Grokking algorithms GitHub repository. Retrieved from: <https://github.com/egonSchiele/grokking_algorithms>
* Python Tutor. <https://pythontutor.com/visualize.html>
* *Optimal substructure*: <https://en.wikipedia.org/wiki/Optimal_substructure>
* *Overlapping subproblems*: <https://en.wikipedia.org/wiki/Overlapping_subproblems>
* Different with Recursive and Inductive: <https://math.stackexchange.com/a/228879>
* Tabulation vs Memoization: <https://www.geeksforgeeks.org/tabulation-vs-memoization/>
* The application of dynamic programming in production planning <https://aip.scitation.org/doi/pdf/10.1063/1.4982520>

# What is Dynamic Programming

Dynamic programming (DP) is both a mathematical optimization method and a computer programming method, used in mathematics, management science, computer science, economics, and bioinformatics to solve complex original problems by decomposing the original problem into relatively simple and easy-to-solve sub-problems. If you are confused by the name "dynamic programming", you can understand it as "multistage decision processes."

DP is often suitable for problems with the properties of *optimal substructure*, *non-aftereffect*, and *overlapping subproblems*:

* A given problem has *Optimal Substructure Property* if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
* *Non-aftereffect property*: Once the state of a certain stage is determined, it is not affected by the state of the future decision-making, that is, the process after a state will not affect the previous state, but only related to the current state.
* Subproblems are basically the smaller versions of an original problem. Any problem is said to have *overlapping subproblems* if calculating its solution involves solving the **same** subproblem multiple times.

DP is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using DP. The idea is to simply store the results of subproblems, so that we do not have to recompute them when needed later. Since the results of all sub-problems are recorded during the calculation process, the time consumed by the DP method is often far less than that of the naive solution method.

DP has two ways to solve the problem, bottom-up and top-down. The top-down approach is **memorized recursion** (memoization), and the bottom-up approach is **inductive** (tabulation).

The problem solved by DP has an obvious feature. Once the result of a sub-problem is solved, the subsequent calculation process will not modify it (i.e., non-aftereffect), and the process of solving the problem forms a directed acyclic graph. DP only solves each sub-problem once, and makes it have the feature of pruning through the way of tabulation or memoization, thereby reducing the amount of calculation.

**Question:** What are the commonalities and differences between DP, and greedy algorithms, recursion, divide-and-conquer?

**Please put your answer here:**

# Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ......

It follows this rule: the current value is the sum of the previous two values.

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation:

F0 = 0, F1 = 1,

Fn = Fn-1 + Fn-2 (n ∈ N, and n > 1)

Now given a number n, print nth Fibonacci Number.

From the above equation, we can clearly deduce that a problem of size "n" has been reduced to subproblems of size "n-1" and "n-2". Hence, we can say that Fibonacci numbers have the optimal substructure property.

Consider evaluating Fib(5). As shown in the breakdown of steps shown in the Figure 1, we can see that:

* Fib(5) is calculated by taking sum of Fib(4) and Fib(3)
* Fib(4) is calculated by taking sum of Fib(3) and Fib(2)
* So on.

Clearly, we can see that the Fib(3), Fib(2), Fib(1) and Fib(0) have been repeatedly evaluated. These are nothing but overlapping subproblems.

Diagram

Description automatically generated

**Figure 1:** Breakdown of steps for calculating a Fibonacci number 5.

This problem has two properties of using the DP method, so let us implement the DP algorithm in Python.

In the repository folder, create a file called **fibonacci.py**.

## Recursive approach

Before using the DP algorithm, let us start with a naive solution to this problem as a control group. Obviously, the problem can be solved by recursion.

Type the following code:

Text

Description automatically generated

If you run test case #1, you can get output 8 very quickly. However, once test case #2 is run, you cannot get the output even if you wait for a few days (use Control-C to force it to stop).

The time complexity of this approach is O(2n), and the space complexity is O(n). If you carefully observe the tree structure of Figure 1, you can find that this approach will traverse all the nodes, and the overlapping subproblems are solved repeatedly.

If you are interested in the specific derivation process of computational complexity, you can see here <https://syedtousifahmed.medium.com/fibonacci-iterative-vs-recursive-5182d7783055>.

## Top-Down Approach of DP

Let us get inspiration from the recursive approach, we need to find a way to avoid sub-problems from being repeatedly calculated.

Whenever we solve a smaller subproblem, we remember (cache) its result so that we don’t solve it repeatedly if it’s called many times. Instead of solving repeatedly, we can just return the cached result. This process of remembering the solutions of already solved subproblems is called Memoization.

Comment out the recursive approach test case #2, and then continue to type the following code in the fibonacci.py file:

Text

Description automatically generated

Now, you can get Fib(100) = 354224848179261915075 very quickly.

Since we use an array of size n to remember the results of the subproblems. Also, consider that the stack memory used by recursion is n. So the space complexity of this approach is O(n + n) = O(n).

The time complexity of this method is O(n). Let us look at Fib(n), when Fib(n-1) is called, it makes a call to Fib(n-2). So, when the call comes back to the original call from Fib(n), Fib(n-2) would already be calculated. Hence the call to Fib(n-2) will be O(1).

Hence,

Thanks to DP, we have successfully reduced an exponential problem to a linear problem.

## Bottom-Up Approach of DP

Now, let's change our thinking. If the calculation starts with Fib(0) and Fib(1), we can know Fib(2), then we can know Fib(3), and so on, we can know Fib(n). This is a kind of inductive, that is, the main thinking of the bottom-up approach of DP.

This approach is different from the top-down approach of DP. Consider the same problem with top-down. In order to calculate Fib(n), you first need to calculate Fib(n-1) and Fib(n-2), and then in order to get Fib(n-1), you need to calculate Fib(n-2) and Fib(n-3), and so on, and terminate when you reach Fib(1) or Fib(0).

Continue to type the following code in the fibonacci.py file:

Text

Description automatically generated

Obviously, the time complexity of this approach is O(n). The space complexity is O(n) because an array of length n is also used to store the results of the subproblems.

**Challenge #1 (optional):** Can you further reduce the space complexity to O(1)?

**Challenge #2 (optional): Maximum path sum in a triangle**

We have given numbers in form of triangle:

7

3 8

8 1 0

3 7 4 4

4 5 2 6 5

Write a program that calculates the highest sum of numbers passed on a route that starts at the top and ends somewhere on the base. Each step can go either diagonally down to the left or diagonally down to the right.

Open the **triangle.py** file, then replace all the question marks (?) in it with useful code to get the correct output.

**Tips:**

* The code uses the bottom-up approach of DP.
* When you are at the second last level node, you know whether to go left or right to get the largest number.
* As an example, in the triangle shown above, if you are currently at the first number 3 in the penultimate level, you know that 5 to the right can get the largest number 7. If you are at the second number 7 on the same level, you know that 5 to the left will get the largest number 12. The first number 8 on the third-to-last level, you know that 12 (=7+5) to the right can get the largest number 20.

**Push Your Work to GitHub**

**Write down all your answers in this document and save this document as a PDF file in the module folder.**

Open a terminal on visual studio code and make sure you’re in the repository folder. (i.e: hos01\_courseName\_GitHubUserName)

**Type the following command to upload your work**:

>>> git add .

>>> git commit -m "Submission for HOS09 - YourName"

>>> git push origin master