Dynamic Programming Recognition and Implementation:
Approaching Algorithms through Memoization or Tabulation

Scott A. Smith
9/26/2020

When first introduced to dynamic programming, the overall steps to fulfill it for a simple one-variable problem seemed straightforward. But once presented with my first multi-variable problem, brain seizure commenced. I don't like brain seizures. I don't like not understanding the logic behind the code I'm being called upon to write. So as a still new student to programming, and dynamic programming in particular, I took it upon myself to find a few sources to help me unfreeze the brain. This short article is the result of that research.

A concise definition of ***dynamic programming*** (DP hereafter) is:

> **Breaking down an optimization problem** into simpler sub-problems, and **storing the solution to each sub-problem** so that each sub-problem is only solved once.[1]

An "optimization problem" is one in which a *maximum* or *minimum* is sought, or some *probability* is sought, under a certain *set of conditions*. For a problem to be solved dynamically, it must "satisfy the overlapping subproblems property,"[2] which as the definition noted, means being able to break down into "simpler sub-problems" of the *same form* as the original problem. The form the problem takes is the *substructure*, and DP techniques work best when they "satisfy the optimal substructure property"[3] (i.e. each has the same form of the equation for solving, and so lends itself to being solved via *recursion*, as the same form of the problem recurs over and over). If you solve the simpler versions, you can solve the main problem.

The sub-problems are "simpler" in the sense that they require less input to compute the solution; the *simplest* version is that right after the *base case(s)* for the problem. The base cases do *not* have the form of the original problem but are the *trivial* or *base value* from which the formulaic solutions build.

Because the same form of the problem recurs during the solution, DP can help solve it. Revisiting the definition above, another key part of DP is "*storing* the solution to each sub-problem." Also often referred to as *caching*, this storage step is what *speeds up* the finding of the main solution. The speed up is quite literal. In Big-O terms of time complexity, many such problems are exponential, $O(2^n)$ or worse. Space complexity is also a serious issue with the potentially massive number of recursions done. Once a simpler solution is found, rather than having to *resolve that solution again for each* higher level of the complex problem, the prior solution's cached answer is referenced. So DP "trades memory space for time efficiency,"[4] but even that fixed memory space is less than the space needed for the extra recursion calls.

---

[1] Alaina Kafkes, "Dymystifying Dynamic Programming," 2017, accessed Sept. 26, 2020, https://dev.to/alainakafkes/demystifying-dynamic-programming; bold in original.

[2] Nitish Kumar, GeeksforGeeks, "How to solve a Dynamic Programming Problem?" last updated Sept. 4, 2020, accessed Sept. 26, 2020, https://www.geeksforgeeks.org/solve-dynamic-programming-problem/

[3] Kumar.

[4] Fabian Robaina, "A Systematic Approach to Dynamic Programming," Aug 9, 2019, accessed Sept 24, 2020, https://medium.com/better-programming/a-systematic-approach-to-dynamic-programming-54902b6b0071.

The Two Main Methods of DP

There are two primary ways to do this dynamic storing of simpler solutions:

1. **Memoization**: a ***top-down***[5] approach that ***uses recursion***. It is top-down because the *first* call for solving the problem is the one to the *final* problem itself. In the famous Fibonacci sequence used for many introductions on the topic of DP, the *initial* solution sought within the algorithm is a call for the *final* solution, so Fib(N), where N is the final Fibonacci number desired, is the first step done in the program. But to solve that, the algorithm must recursively call what the solution to Fib(N-1) and Fib(N-2) is, etc., ***down*** to the base cases of Fib(1) = 1 and Fib(0) = 0.[6] The *storing* of the answers (often in what is called the ***memo***, hence *memoization*) comes *during the return up* the recursion chain, after the base cases are reached. The algorithm started at the top N, came down to the bases 0 & 1, and now return to the top, storing needed answers for each other step along the way. In the end, memoization can bring the exponential time complexity down to polynomial time, perhaps as low as quadratic $O(n^2)$, while also reducing the overall space complexity as well with far less recursion calls (depending on the problem and memoized implementation, down to O(n) or lower).
2. **Tabulation**: a ***bottom-up*** approach, does *not* use recursion, but ***iteration***. The *first* call for solving the problem is to the first *simple* problem of the *same form* as the main problem. So to solve for Fib(N), the base cases (of 0 and 1) are already given, and the first call is to the simple solving of Fib(2) using the known Fib(1) and Fib(0), then working ***up*** to the solution sought, which is Fib(N). The *storing* of the answers (often in a variable called the ***table***, hence *tabulation*, but also commonly the ***cache***) comes immediately in the process with each step of the iteration. Through tabulation, time complexity can be reduced O(n)—one iteration through each needed set of parameters for the solution. Space complexity may potentially be reduced to O(1).

The Storage Mechanics

So how does the storing work, especially in more complex cases where there is not just a single sized, integer-based variable (like with Fibonacci) related to the solution? Such one size, one number variable cases lend nicely to either an array index or a unique object key. But really, the number and size of variables contribute no matter what for the storage of solutions. For every variable (V) needed, you either:

- Make it a **V-dimensional array**; e.g. 3 variables would be an array of three dimension (*three indexes*) to store the unique solution to that: array[v1][v2][v3]
- Make it a **V-distinguished key**: e.g. 3 variables would build a *key string* off all three, maybe something like "v1Name-v2Name-v3Name"

For any variable that itself has size (like another array), then more dimensions related to that size may be needed, but the process is still the same. Which implementation for storing is

[5] Robaina; the parallel "bottom-up" terminology for tabulation also was talked about in this article.

[6] Fibonacci numbers are found by adding together the prior two numbers of the sequence, starting with 0 and 1. So 0 + 1 = 2 then 1 + 2 = 3 then 2 + 3 = 5 then 3 + 5 = 8, etc.

better will depend on your preferences and programming needs, since how arrays and objects use actual computer memory are distinctly different (at least in some programming languages).

The Steps to Solve

So, what are practical steps to doing DP? That can be the challenging part, especially with a multivariable set of conditions that affect the solution. Here is what I've gleaned from the sources cited herein about a potential "best" practice procedure:

1. **Define the Problem**: through (a) stating *concisely in words* and/or (b) *diagramming out visually*, convey what you are trying to solve for, paying special attention to the ***state*** of the problem. The state is "the set of parameters that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space."[7] Defining this is important, because it is the changing of the parameters that ***transitions*** from one state to the next and is what determines the *relationship* between each state. Knowing the states and transitions allows one to...
2. **Formulate the Problem**: the ***mathematical expression*** of the problem, showing the parameters relationships to one another. Between steps 1 and 2, you should already know the size and number of variables you need to transition to each state, as well as know what the base case(s) are for the problem. After this step, one is ready to...
3. **Encode in the Program**: write out the code version for ***repetitively solving*** the math problem from step 2. It is often suggested to start with the basic, naive recursion solution (without any DP aspect), and then once that is solved...
4. **Refactor with DP**: using either memoization or tabulation, whichever method suits the need; often strategies say that after the naive recursion of step 3, it is easiest to move to memoization (since that is still recursive), and then refactor a second time from memoization to tabulation.

An observant person might note that the acronym formed by those steps is **DFER**, as in *defer* solving the main problem until you have solutions to the smaller ones. Yes, that is my own acronym: **Define, Formulate, Encode, Refactor**.

Conclusion

I am just starting my journey in actually putting into practice DP. But having this foundational understanding now of the reasoning and process will, I hope, give me a head start on future success at it. Perhaps it will also help someone else at the same point in their journey.

[7] Kumar.