# Introduction to Dynamic Programming

Pranav Mathur
Based on Daniel Wisdom's Lecture

December 11th, 2020

*"Those who cannot remember the past are condemned to repeat it"* - George Santayana

## 1 Introduction

Dyanamic programming is an extremely powerful technique that can greatly reduce the time complexities of your solutions. It is heavily tested in the Gold and Platinum divisions of USACO and shows up in nearly every other programming contest.

### 1.1 Example - Fibonacci Numbers

To illustrate the power of dynamic programming, let's look at a classic application - finding the $n$th Fibonacci number. Recall that the Fibonacci sequence is a recursive sequence with $F_0 = 0$, $F_1 = 1$, and, for $n \geq 2$, $F_n = F_{n-1} + F_{n-2}$. This definition motivates the following recursive solution.

**Algorithm 1.1** Fibonacci - Naïve

```
function FIB($n$)
    if $n = 0$ or $n = 1$ then
        return $n$
    else
        return FIB($n - 1$) + FIB($n - 2$)
```

This solution runs in $O(2^n)$ time since for each call to the function, we make two more recursive calls. To see how to speed this up, let's look at an example. Say we want to compute $F_{10}$. we call $FIB(10)$, which calls $FIB(9)$ and $FIB(8)$. Then, $FIB(9)$ calls $FIB(8)$ and $FIB(7)$. Note that we call $FIB(8)$ twice, even though the value returned from both calls is the same!

To solve this problem, we maintain an array that keeps track of all the values of all the Fibonacci numbers we have already calculated.

**Algorithm 1.2** Fibonacci - Top-Down Dynamic Programming

```
initialize $dp[0 \ldots n] \leftarrow -1$
function FIB(n)
    if $dp[n] \neq 0$ then
        return $dp[n]$
    if $n = 0$ or $n = 1$ then
        return $n$
    else
        $dp[n] \leftarrow$ FIB($n - 1$) + FIB($n - 2$)
        return $dp[n]$
```

This solution reduces the runtime of the recursive solution from $O(2^n)$ to $O(n)$ since each $F_n$ is only computed once. This is an enormous improvement and illustrates the power of dynamic programming. However, there is still another improvement we can make. Using recursion comes with an overhead cost due to several function calls. We can eliminate this overhead by computing $F_n$ from the "bottom-up" instead of "top-down" (see sections 2.2 and 2.3). This means we start with $F_0$ and $F_1$, then compute $F_2$, then $F_3$, and so on until we get $F_n$.

**Algorithm 1.3** Fibonacci - Bottom-Up Dynamic Programming

```
function FIB($n$)
    initialize $dp[0 \ldots N]$
    $dp[0] \leftarrow 0$
    $dp[1] \leftarrow 1$
    for $i \leftarrow 2 \ldots n$ do
        $dp[i] \leftarrow dp[i-1] + dp[i-2]$
    return $dp[n]$
```

By using dynamic programming, we have reduced a $O(2^n)$ recursive solution to a $O(n)$ solution with no recursive overhead. In general, DP reduces the time complexities of problems from exponential to polynomial.

## 2 How Do I Use Dynamic Programming?

### 2.1 When to Use DP

The Fibonacci numbers problem has some important properties that are critical to our ability to apply a DP solution. First, recall that our call to $FIB(10)$ led to multiple calls of $FIB(8)$ (and $FIB(7)$, $FIB(6)$,...) that all had the same answer. This property is known as *overlapping subproblems.*

Now, remember that in our iterative solution, we used the solutions to $FIB(1 \dots 9)$ to calculate $FIB(10)$. This property is called *optimal substructures.*

Together, these properties form the criteria for applying dyanmic programming to a problem. Here they are again, written in general terms.

- **overlapping subproblems** - solutions to subproblems with the same state variables (see section 4) are used repeatedly.
- **optimal substructures** - solutions to subproblems are part of the solution to the original problem.

There are generally two approaches to coding DP solutions, which are covered in the next two sections.

### 2.2 Top-Down DP

Top-down DP involves implementing a recursive solution and maintaining a table with already-computed values. This method is also called *memoization.*

**Advantages**

- Only computes subproblem solutions that are necessary for solving the original problem.
- Easier to understand conceptually when starting out.

**Disadvantages**

- Overhead due to recursion.
- Code is generally longer.

### 2.3 Bottom-Up DP

Bottom-up DP involves computing subproblems iteratively in a way that all previous values needed to compute the current value have already been computed. To illustrate, in the Fibonacci example, we used bottom-up DP when we computed $FIB(1 \dots N-1)$ before trying to compute $FIB(N)$.

**Advantages**

- Good when many subproblems are revisited, since there is no overhead due to recursion.
- Opportunities for memory optimization, such as sliding-window trick.
- Code is generally shorter and cleaner.

**Disadvantages**

- May compute unecessary subproblems.

Both these approaches have the same time complexity, and which one you use is largely a matter of preferences, but keep in mind the above advantages and disadvantages when deciding between the two.

## 3 Knapsack Problem

The knapsack problem is a canonical DP problem with many varieties and is tested often in the USACO Gold and Platinum Divisions. When reading through the following sections, try to understand the motivations and broader ideas of DP so you can apply them to other problems.

### 3.1 0-1 Knapsack Problem

Suppose you have a knapsack with a maximum weight capacity of $M$ and you want to fill it up with objects. You have $N$ distinct objects, and the $k$th object has a value $v_k$ and a weight $w_k$. What combination of objects should you choose so that you maximize the value of the objects in your knapsack while staying within the weight limit?

Your first impulse may be to try a greedy solution in which you pick objects in order of their value. However, it is quite easy to find an example that breaks this solution (Try $M = 7$, $N = 3$ and the (value, weight) pairs of the objects are $(7, 7)$, $(5, 5)$, and $(3, 2)$). When greedy doesn't work, we try DP.

The first step to solving any DP problem is to decide what variables we will use to define our states. This is generally the most difficult step. To start, let's think about what happens when we add item $N$ to the knapsack. If the knapsack is empty, then the remaining capacity of the knapsack is now $M - w_N$ and the value of the knapsack is $v_N$. Now, since we have used item $N$, we must now fill a weight of $M - w_N$ with the remaining items. This problem is analogous to maximizing the value of a knapsack with capacity $M - w_N$ with objects $1 \ldots N - 1$. Aha! We have discovered optimal substructures.

What about overlapping subproblems? Note that we may get to the problem of filling a knapsack with weight $m < M$ using objects $1 \ldots k$ in multiple ways (try thinking of some examples). Thus the overlapping subproblems condition is satisfied.

Now, lets outline a solution. Since our states are defined by the capacity of the knapsack and the objects we can choose from, let's define $dp[k][m]$ as the maximum value we can obtain in a knapsack with weight capacity $m$ with objects $1 \ldots k$. We define our state in this way so that we can easily transition to subproblems by choosing either to skip object $k$ or to add it. Our recurrence relation is then $dp[k][m] = max(dp[k-1][m], dp[k-1][m - w[k]] + v[k])$. Before we start coding, let's determine our base cases for recursion. Clearly, $dp[k][0] = 0$ for all $k$ and $dp[0][m] = 0$ for all $m$. These cases are where we terminate our top-down solution or begin our bottom-up solution.

**Algorithm 3.1** 0-1 Knapsack Problem Top-Down Solution

```
initialize $dp[1 \ldots N][0 \ldots M] \leftarrow -1$
$dp[1 \ldots N][0] \leftarrow 0$
$dp[0][0 \ldots M] \leftarrow 0$
function KNAPSACK($k$, $m$)
    if $k = 0$ or $m = 0$ then                              ▷ No more objects or knapsack is full
        return 0
    if $dp[k][m] \neq -1$ then                              ▷ Already computed this subproblem
        return $dp[k][m]$
    if $w[k] > m$ then                                      ▷ Can't fit object $k$
        $dp[k][m] = $ KNAPSACK($k - 1, m$)                   ▷ Skip object $k$
    else
        $dp[k][m] = max($KNAPSACK$(k-1, m)$, KNAPSACK$(k-1, m - w[k]) + v[k])$
    return $dp[k][m]$
```

**Algorithm 3.2** 0-1 Knapsack Problem - Bottom-Up Solution

```
initialize $dp[1 \ldots N][0 \ldots M] \leftarrow -1$        ▷ Final answer will be in $dp[N][M]$
$dp[1 \ldots N][0] \leftarrow 0$
$dp[0][0 \ldots M] \leftarrow 0$
for $k$ from $1 \ldots N$ do                 ▷ Compute solution for items $1 \ldots k-1$ before $1 \ldots k$
    for $m$ from $1 \ldots M$ do
        $dp[k][m] = dp[k-1][m]$                             ▷ Applies to all objects
        if $w[k] \leq m$ then                               ▷ Can fit object $k$
            $dp[k][m] = max(dp[k-1][m], dp[k-1][m - w[k]] + v[k])$
```

Note that both solutions have the same time complexity of $O(NM)$, but the top-down solution is somewhat be easier to understand, while the bottom-up solution has shorter code. It's up to you which one to use.

## 3.2 Unbounded Unordered Knapsack Problem

This variant of the knapsack problem is very similar to the 0-1 variant, but instead of having $N$ distinct objects, we have $N$ distinct *types* of objects and an unlimited supply of each type (hence the name "unbounded"). The order in which the objects are placed into the knapsack does not matter.

The solution to this problem is nearly identical to the solution to the 0-1 problem, but with one distinction. Note that since we can add an object of type $k$ as many times we we want, instead of transitioning to state $dp[k-1][m - w[k]]$, we transition to state $dp[k][m - w[k]]$, since object $k$ is still available to us. Thus, our recurrence relation is $dp[k][m] = max(dp[k-1][m], dp[k][m - w[k]] + v[k])$.

I will omit pseudocode for the top-down approach, since it is nearly identical to the 0-1 top-down solution. However, there is a clever memory optimization we can make to the bottom-up approach. Note that as we are iterating through $0 \ldots M$, we only need to access values to the left of the current index in the same row ($dp[k][m - w[k]]$) and the value at the current index in the previous row ($dp[k-1][m]$). We never use the values in $dp[1 \ldots k-1][0 \ldots m-1]$ again. With this observation in mind, I claim that instead of maintaining a $dp[1 \ldots N][0 \ldots M]$, we only need to maintain $dp[0 \ldots M]$ and we can update this array in each iteration. Our recurrence in code will then be $dp[m] = max(dp[m], dp[m - w[k]])$. To see why this is true, note that we have already computed $dp[m]$ for all values of $m$ to the left of the current index for the current $k$. Thus, this range represents $dp[k][0 \ldots m-1]$ in our previous solution. Anything at or after our current index has not been computed, so it represents $dp[k-1][m \ldots M]$ in our previous solution. Thus, all the information we need can be stored in the one-dimensional DP array.

**Algorithm 3.3** Bottom-Up Unbounded Knapsack Solution with Memory Optimization

```
initialize $dp[0 \dots M] \leftarrow 0$                          ▷ Final answer will be in $dp[M]$
for $k$ from $1 \dots N$ do
    for $m$ from $1 \dots M$ do
        if $w[k] \le m$ then
            $dp[m] = max(dp[m], dp[m - w[k]] + v[k])$
```

This example demonstrates how bottom-up approaches can allow for clever optimzations that streamline your code.

### 3.3 Application - Subset Sum

In this problem, you are given the set of positive integers $1, 2, \dots, N$ and are asked to divide it into two sets with equal sum.

First, note that if the sum of the integers from $1 \dots N$ is odd, such a separation is impossible. If the sum is even, each subset will have sum $S = \frac{N(N+1)}{4}$. Thus, our problem is reduced to finding the number of ways to choose a subset of $1, 2, \dots N$ that adds up to $S$.

This problem can be solved using a variant of the 0-1 knapsack problem. It differs from the problem in section 3.1 since instead of finding the maximum possible value of the knapsack, we are asked to find the number of ways to fill the knapsack. Note that we can either choose to ignore a number $k$, leaving us with the numbers $1 \dots k-1$ tog get a sum of $s$, or add a $k$ to our subset, leaving us with the numbers $1 \dots k-1$ to get a sum of $s-k$. Thus, our recurrence relation is $dp[k][s] = dp[k-1][s] + dp[k-1][s-k]$. Again, the code for this problem is nearly identical to the code in section 3.1.

## 4 General Strategy

1. **Identify state variables** - These can be thought of as the arguments passed into the recursive solution. Generally, the more state variables you have, the slower your solution will be. For example, if the problem you are solving is $f(a, b, c)$, with $a$ from $0 \dots A$, $b$ from $0 \dots B$, and $c$ from $0 \dots C$, then the time complexity of a bottom-up solution is $O(ABC)$.
2. **Write recurrence relation** - Yes. "Write" it. On paper. Since DP code is generally very simple and difficult to discern the problem from, working out full DP states and transitions on paper can be *extremely* helpful in making sure you have the correct solution to your specific and anticipating bugs before you code.
3. **Determine base cases** - DP solves recursive problems, so there will be a base case.
4. **Code** - This is generally the fastest part of solving a DP problem. Make sure you code base cases correctly and pay attention to indexing and order of nested loops.

## 5 Practice Problems

The only way to get good at DP problems is to, and I cannot stress this enough, PRACTICE. The following resources offer excellent problems to apply your new DP skills :))

- USACO Guide Introduction to DP - https://usaco.guide/gold/intro-dp
- USACO Guide Knapsack Problem - https://usaco.guide/gold/knapsack
- CSES Dynamic Programming Problems (also linked in USACO Guide) - https://cses.fi/problemset/
- Codeforces DP Problems - https://codeforces.com/problemset?order=BY_RATING_ASCtags=dp