Introduction to Dynamic Programming

Charles Zhao

December 2, 2016

"Unfortunately, programmer, no one can be told what DP is. You have to try it for yourself."

– Stolen from an old SCT lecture that stole it from an older SCT lecture that adapted it from *The Matrix*

1 Introduction

Dynamic Programming (DP) is a technique for reducing the runtime of certain kinds of problems. A problem that can be solved using DP must satisfy two prerequisites:

1. It must have *optimal substructures*: The solution to the subproblem is part of the solution to the original problem.
2. It must have *overlapping subproblems*: The solutions to subproblems are used repeatedly.

2 Examples

2.1 Fibonacci

If you have created a Fibonacci program, you have most likely already used DP. This problem involves calculating the nth Fibonacci number. Remember that the Fibonacci sequence is defined as $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$. Here is a naive solution:

Algorithm 1 Fibonacci: Naive

```
function FIB(n)
    if n = 0 or n = 1 then
        return n
    return FIB(n − 1) + FIB(n − 2)
```

Since each level of recursion calls itself twice, this results in exponential time complexity. However, notice that certain computations are repeated multiple times. Therefore, we can improve this to linear time at the expense of space by caching computations. This use of caching is called the *top-down* approach, also known as *memoization*, because we start from the overall problem and recursively break it down into subproblems until we reach the base case(s).

Algorithm 2 Fibonacci: Top-Down

```
initialize memo[0...N] ← −1
function FIB(n)
    if n = 0 or n = 1 then
        return n
    if memo[n] > −1 then
        return memo[n]
    memo[n] ← FIB(n − 1) + FIB(n − 2)
    return memo[n]
```

Another approach is the *bottom-up* approach, which involves starting from the smallest subproblems and then gradually combining them into larger subproblems until we reach the solution to the desired problem.

Algorithm 3 Fibonacci: Bottom-Up

```
function FIB(N)
    initialize dp[0...N]
    dp[0] ← 0
    dp[1] ← 1
    for i ← 2...N do
        dp[i] ← dp[i − 1] + dp[i − 2]
    return dp[N]
```

Both of these DP solutions run in linear time and linear space, which is far better than the naive solution's exponential time complexity. In general, DP reduces the time complexity of problems from exponential to polynomial.

2.2 Unbounded Knapsack Problem

Now let's try a slightly harder problem. The knapsack problem is the canonical DP problem and is often present in introductory algorithms contests. Given N types of objects, with the ith type having value `V[i]` and weight `W[i]`, which objects do we select to maximize the value without exceeding the knapsack's weight capacity C? Note that we can take multiple copies of the same type of object, hence the name "unbounded."

The most difficult part of this problem, as with most DP problems, is determining how to divide the problem into overlapping subproblems. Let's define `memo[c]` as the solution to the subproblem with capacity c. Note that adding an object to the knapsack is analogous to reducing the capacity of the knapsack. At each step, we want to add an object such that the sum of the value of the object and the value of the smaller capacity knapsack is maximized. Thus, the knapsack problem exhibits *optimal substructures*, which is one of the characteristics of a DP problem. Specifically, `memo[c] = max(memo[c], memo[c-W[i]] + V[i])`, looping i from 1 to N and with `memo[0] = 0`.

Algorithm 4 Unbounded Knapsack

```
initialize memo[0...N] ← −1
memo[0] ← 0
function KNAPSACK(c)
    if memo[c] > −1 then                          ▷ Already computed this subproblem
        return memo[c]
    for i ← 1...N do
        if c − W[i] > 0 then                      ▷ Check if the object can fit
            memo[c] ← MAX(memo[c], KNAPSACK(c − W[i]) + V[i])
```

2.3 0-1 Knapsack Problem

There are several variations of the knapsack problem. In the 0-1 knapsack problem, we can either take 1 copy of an item or not take it at all. Let's define `memo[i][c]` as the solution to the subproblem considering only the available objects up to the ith object and with a knapsack of capacity c.

Algorithm 5 0-1 Knapsack

```
initialize memo[0...N][0...C] ← −1
memo[0] ← 0
function KNAPSACK(i, c)
    if i = N or c = 0 then                        ▷ No more objects or bag is full
        return 0
    if memo[i][c] > −1 then                       ▷ Already computed this subproblem
        return memo[i][c]
    if W[i] > c then                              ▷ Cannot fit this object
        memo[i][c] ← KNAPSACK(i + 1, c)           ▷ Skip this object
    else
        memo[i][c] ← MAX(KNAPSACK(i + 1, c), V[i] + KNAPSACK(i + 1, c − W[i]))
    return memo[i][c]
```

3 Top-Down vs. Bottom-Up

As we saw with Fibonacci, there are generally two approaches to implementing DP. Both approaches use tables, but the bottom-up DP table is filled differently from the top-down DP memo table. In the top-down approach, the memo table is only filled as needed through recursion. In the bottom-up approach, the table is filled iteratively in an order such that the previous values needed to compute the current value have already been computed. Although both approaches will have the same time complexity and in general which approach you use is a matter of preference, it is worth noting the advantages and disadvantages of each, which may become important at higher level competitions.

As mentioned before, the top-down approach only computes the necessary subproblems. However, there is overhead due to recursion. Therefore, the bottom-up approach is faster if many subproblems are revisited, because there is no overhead due to recursion. However, the bottom-up approach may compute some unnecessary subproblems. Another advantage of the bottom-up approach is that it has opportunities for optimization that the top-down approach does not, such as the sliding window trick.

4 General Strategy[1]

1. **Identify state variables.** These are the variables that define subproblems, i.e., they are the arguments to the recursive function. In general, the more state variables there are, the slower the algorithm. For example, if we have state `dp[i][j][k]` where i ranges from 0 to A, j from 0 to B, and k from 0 to C, then the bottom-up approach must take at least $O(ABC)$ time.

2. **Determine base case(s).** Because DP solves recursive problems, there has to be a base case.

3. **Determine the recurrence relation.** This is the relationship between the overall problem and the subproblems (remember, DP problems must exhibit optimal substructures). Also, these subproblems must overlap.

5 Problems

The best way to get better at DP is to practice!

1. **Max Range Sum.** Given an integer array, determine its maximum range sum, i.e., the maximum range sum query.

2. **Longest Increasing Subsequence.** Given a sequence of numbers, determine its longest increasing subsequence. Note that the subsequence is not necessarily contiguous.

3. **Coin Change.** Given n types of coin denominations with values `V[0]`, `V[1]`, ..., `V[n-1]`, determine the minimum number of coins needed to make change for an amount of money C. Assume `V[0] = 1` so that you will always be able to make change for any amount C.

[1] This section borrows heavily from Ryan Jian's lecture.