Dynamic Programming

Dynamic programming (DP) is used heavily in optimization problems (finding the maximum and the minimum of something). Applications range from financial models and operation research to biology and basic algorithm research. So the good news is understanding DP is profitable. However, the bad news is that DP is not an algorithm or a data structure that you can memorize. It is a powerful algorithmic design technique.

Optimal Sub-structure

DP takes the advantage of the optimal sub-structure of a problem. A problem has an optimal sub-structure if the optimum answer to the problem contains optimum answer to smaller sub-problems. As an example, the shortest path from $u$ to $v$ is composed of some edge $(w, v)$ and the shortest path from $u$ to $w$ (a smaller problem to solve).

Dynamic programming (like the divide-and-conquer method) solves problems by combining the solutions to subproblems. Dynamic programming applies when the subproblems re-occur - that is, when subproblems share subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Main four steps for solving a Dynamic program

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, (typically in a bottom-up fashion see next discussion).
4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Review: Memoized Recursive and Bottom-up DP

There are two main ways to approach dynamic programming. Dynamic programming can be framed as a memoized recursive algorithm or as a bottom-up iterative algorithm.
Memoized Recursive

To turn a normal recursive algorithm into a memoized recursive algorithm, we add a memo dictionary to store outputs for each input combination (subproblem) we’ve solved so far. To solve a subproblem that’s not in the memo, we just run our original recursive algorithm and then store the result in memo.

Consider a function to calculate the \( n \)th Fibonacci number. The Fibonacci sequence is defined by \( F_1 = 1 \), \( F_2 = 1 \), and \( F_n = F_{n-1} + F_{n-2} \). The sequence looks like 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

The following is a recursive implementation of calculating the \( n \)th Fibonacci number:

```python
1 def fib(n):
2     if n <= 2:
3         return 1
4     else:
5         return fib(n-1) + fib(n-2)
```

Unfortunately, the above implementation is very slow. The runtime of the algorithm can be described using the recurrence \( T(n) = T(n-1) + T(n-2) \). Its very easy to give a argument that this algorithm has to be at least exponential. We will show this by arguing a lower bound on the recurrence:

\[
T(n) = T(n-1) + T(n-2) + O(1) \geq 2T(n-2) + O(1) \geq 2^n
\]

The inefficiency comes from the fact that we end up redoing our calculations for some \( \text{fib}(n) \) many, many times. We can improve our algorithm by adding memoization: keeping track of the values of \( n \) for which we already computed \( \text{fib}(n) \).

```python
1 memo = {}
2 def fib(n):
3     # Check if we’ve already solve the subproblem
4     if n in memo:
5         return memo[n]
6     # Normal recursive algorithm
7     if n <= 2:
8         f = 1
9     else:
10         f = fib(n-1) + fib(n-2)
11     # Remember our solution for later
12     memo[n] = f
13     return f
```

Analysis  Rather than attempting the complex analysis how the runtimes add up recursively, we instead separate out the problem into: (1) how many subproblems we need to solve and (2) how long each subproblem takes to solve assuming constant lookup of its subproblems. This makes
sense, since we are just reordering the pieces of our recursive runtime to make them easier to analyze. Our overall runtime thus becomes:

\[ T = \text{(number of subproblems)} \times \text{(time per subproblem)} \]

In our Fibonacci example, we thus get \( O(n) \) subproblems and \( O(1) \) to solve each subproblem when we ignore the recursive costs, giving us \( O(n) \) total. For each \( n \), we only have to compute \( F_n \) the first time we call \( \text{fib}(n) \), and if we ever call \( \text{fib}(n) \) in the future, we can pull the result from our memoized dictionary.

**Bottom-Up**

Often times it makes sense to avoid the recursive overhead involved in a recursive algorithm and re-order how we solve the subproblems. Rather than letting the recursive algorithm solve the subproblems on an on-demand basis, we can choose to build up our subproblems from the base cases up, in such a way that every time we solve a subproblem any subproblems it refers to are already solved. Finding such an ordering is equivalent to topologically sorting the DAG defined by the dependencies of each subproblem.

Our Fibonacci example becomes:

```python
def fib_bottom_up(n):
    fib = {}
    for k in range(n+1):
        # Our original recursive solution.
        # Notice that our ''recursive'' calls are now lookups.
        if k <= 2:
            f = 1
        else:
            f = fib[k-1] + fib[k-2]
        fib[k] = f
    return fib[n]
```

**Analysis**  Our analysis is exactly the same as the memoized recursion, though in this case it’s much clearer from where the runtime arises.

**Trade-offs**

There are advantages and disadvantages to both types of DP.

**Advantages to Recursive Memoized:**

- Often much easier to understand
- Don’t have to determine an ordering, which might be hard to do manually in some cases
Advantages to Bottom-Up:

- Doesn’t have the overhead of recursion (we also avoid the issue of exceeding the maximum recursion depth, due to extremely deep recursions)
- Often easier to analyze runtime

Coin Puzzle

We have a grid of squares and on some selected squares, there are coins (at most one coin per square). You start at square \((0, 0)\) at the bottom left to your goal is to go to \((n, m)\) on the top right only moving one step up or to the right each time. Your goal is to maximize the number of coins you encounter. (The number of columns is \(n + 1\) and the number of rows is \(m + 1\).) See Figure 1 for an example setup and path.

**Goal:** Maximize number of coins picked by choosing an appropriate path.

![Figure 1: Coin Puzzle.](http://demonstrations.wolfram.com/PickingUpCoinsInAGrid/)

[Sidenote: There are an exponential number of possible paths!]

Algorithm using DP

Let \(c_{ij} = 1\) if there is a coin at \((i, j)\) else \(0\).
Let’s define \(S(i, j)\) to be largest number of coins you can encounter on a path up to and including \((i, j)\).

**Goal:** Maximize \(S(n, m)\).

The insight is that you can only reach \((i, j)\) in only 2 different ways: from the left \((i-1, j)\) square or from the bottom \((i, j-1)\) square.

Therefore, we can write:
\[
S(i, j) = \max(S(i - 1, j), S(i, j - 1)) + c_{ij}, \text{ if } i \geq 1, j \geq 1
\]
\[
S(0, 0) = c_{00}
\]
\[
S(i, 0) = S(i - 1, 0) + c_{i0}, \text{ if } i \geq 1
\]
\[
S(0, j) = S(0, j - 1) + c_{0j}, \text{ if } j \geq 1
\]

**Analysis** The total number of sub-problems here is \(O(mn)\). Each sub-problem takes \(O(1)\) time to compute, so the total runtime of this algorithm is \(O(mn)\).

**Finding the Optimal Path** Note that \(S(n, m)\) only stores the optimal number of coins that can be encountered on a path to \((n, m)\). It does not actually tell us what path to take. If we wanted to figure out the path that corresponds with the optimal solution, we can create an additional table to store the optimal paths up to each position. Let \(P(i, j)\) be the predecessor of \((i, j)\) on the optimal path that ends at \((i, j)\). We can fill in \(P(i, j)\) while we are computing \(S(i, j)\).

\[
P(i, j) = \text{ either } (i - 1, j) \text{ or } (i, j - 1), \text{ whichever had a higher } S, \text{ if } i \geq 1, j \geq 1
\]
\[
P(0, 0) = \text{ None}
\]
\[
P(i, 0) = (i - 1, 0), \text{ if } i \geq 1
\]
\[
P(0, j) = (0, j - 1), \text{ if } j \geq 1
\]

We can then use \(P\) to reconstruct the optimal path by starting at \((n, m)\), our final position. We can then find the predecessor to \((n, m)\) by looking at \(P(n, m)\). We can then find the predecessor of the predecessor of \((n, m)\), and then find the predecessor of that, and so on, until we eventually reach the starting position. This will give us the backwards version of the optimal path.

The following is example code to compute the optimal path from \((0, 0)\) to \((n, m)\). It computes \(S\) and \(P\) bottom-up, starting with the base cases.

```python
def coin_puzzle(c, n, m):
    # initialize S and P to be (n+1)x(m+1) arrays
    S = [
    P = [
    for i in range(n + 1):
        S.append([0] * (m + 1))
        P.append([None] * (m + 1))
    # dynamic programming computation
    for i in range(n + 1):
        for j in range(m + 1):
            if i >= 1 and j >= 1:
                if S[i - 1][j] > S[i][j - 1]:
                    S[i][j] = S[i - 1][j] + c[i][j]
                    P[i][j] = (i - 1, j)
                else:
                    S[i][j] = S[i][j - 1] + c[i][j]
                    P[i][j] = (i, j - 1)
            elif i == 0 and j == 0:
                S[i][j] = c[0][0]
                P[i][j] = None
            elif i >= 1: # j = 0
                S[i][j] = c[i][0]
                P[i][j] = None
```
Rod cutting

Recall the rod cutting problem from lecture. You manufacture steel rods of some fixed length and sell them in segments of integer length. Because of the laws of supply and demand, the price per length is not constant. Given a set of prices (i.e. an array $p[i]$ indicating the price for an integer cut of length $i$) for each possible length segment, how should you cut a given rod of length $n$ into integer length segments to maximize revenue? Assume that no cost or loss incurred with each cut.

A useful thing to do in Dynamic Program (and in recursion) is usually to clearly state what a subproblem means or define in words what the recursion means. Define $r(j)$ to be the optimal way to cut a rod of integer length $j$. Since we do not know where is the optimal place to cut the rod we consider every location $i \in \{0, \cdots, j\}$ (essentially doing exhaustive search) to do the cut. Then we are left with two rods of length $i$ and $j - i$ to cut optimally. If we consider every location of the cut and take the best one, we are essentially guaranteed to get the best cut for the rod. Thus we get the following recursion:

$$
r(j) = \max\{p[n], r(1) + r(j - 1), r(2) + r(j - 2), \ldots, r(j - 1) + r(1)\}
$$

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, we view a decomposition as consisting of a first piece of length $i$ cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. We may view every decomposition of a length-$n$ rod in this way: as a first piece followed by some decomposition of the remainder. When doing so, we can couch the solution with no cuts at all as saying that the first piece has size $i = n$ and revenue $p(j)$ and that the remainder has size 0 with corresponding revenue $r(0) = 0$. We thus obtain the following simpler version of equation:

$$
r(j) = \max_{1 \leq i \leq j}\{p(i) + r(j - i)\}
$$

In this formulation, an optimal solution embodies the solution to only one related subproblem—the remainder—rather than two.
Constructing optimal solution to Rod cutting problem

Recall the bottom up solution shown in lecture:

```python
1 def r_bottom_up(n,p):
2     r = [0] * (n+1)
3     for j in range(1,n+1):
4         current_optimal_r_j = -1
5         for i in range(1,j):
6             current_optimal_r_j = max(current_optimal_r_j, p[i] + r[j-i])
7         r[j] = current_optimal_r_j
8     return r[n]
```

To extend this and make sure that we track the optimal way of cutting a rod, we just need to keep track of what cuts were responsible for the maximum in \( r(j) = \max_{1 \leq i \leq j} \{p(i) + r(j-i)\} \). i.e. just keep track of what \( i \) resulted in the optimal cut for the current rod in consideration of length \( j \). For this particular problem we will define \( s[j] \) to hold the optimal size \( i \) of the first piece to cut off when solving a subproblem of size \( j \). The resulting code is as following:

```python
1 def r_bottom_up_extended(n,p):
2     r = [0] * (n+1)
3     s = [0] * (n+1)
4     for j in range(1,n+1):
5         current_optimal_r_j = -1
6         for i in range(1,j+1):
7             if current_optimal_r_j < p[i] + r[j-i]:
8                 current_optimal_r_j = p[i] + r[j-i]
9                 s[j] = i
10     r[j] = current_optimal_r_j
11     return r[n]
```

This procedure is similar to \( r\_bottom\_up \), except that it creates the array \( s \) in line 1, and it updates \( s[j] \) in line 8 to hold the optimal size \( i \) of the first piece to cut off when solving a subproblem of size \( j \).

The following procedure takes a price table \( p \) and a rod size \( n \), and it calls \( r\_bottom\_up\_extended \) to compute the array \( s[1, ..., n] \) of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length \( n \):

```python
1 def r_bottom_up_extended(n,p):
2     (r,s) = r_bottom_up_extended(n,p)
3     j = n
4     while j>0:
5         print s[j]
6         i = s[j]
7         j = j - s[j]
```

Essentially what the code does is looks at the optimal location \( i \) to cut the current rod of length \( j \) to produce the optimal size \( i \) of the first piece when solving a subproblem of size \( j \). At the beginning \( j = n \) is the whole rod, so the algorithm prints the correct location to cut the full rod and the optimal size \( i \) of the first piece to cut off when solving a subproblem of size \( n \) (basically you view this as the base case of our induction). Then because our recursion was set
up as \( r(j) = \max_{1 \leq i \leq j} \{ p(i) + r(j - i) \} \), then at the next step we only need to worry about the remaining rod on the right of length \( j - i = j - s[j] \). Therefore we want to get the the optimal size \( i \) of the first piece to cut off when solving the remaining length \( j - s[j] \). We continue in this fashion until we get to the end of the rod and there is not more rod to be cut.

As a last note, section 15.1 of CLRS has a detailed discussion of the rod cutting problem.

**Practice DP problem: Crazy 8’s**

In the game Crazy 8’s, we are given an input of a sequence of cards \( C[0], \ldots, C[n - 1] \), e.g., 7♣, 7♥, K♣, K♠, 8♥. We want to find the longest subsequence of cards where consecutive cards must have the same value, same suit, or have one of the two cards be an eight. The longest such subsequence in the example is 7♣, K♣, K♠, 8♥.

To solve this, if the cards are stored in array \( C \), we will to keep an auxiliary score array \( S \) where \( S[i] \) represents the length of the longest subsequence ending with card \( C[i] \).

We start with \( S[0] = 1 \) since the longest subsequence ending with the first card is that card itself and has a length of 1. We iteratively calculate the next score \( S[i] \) by scanning all previous scores and set \( S[i] \) to be \( S[k] + 1 \) where \( S[k] \) represents the length of the longest subsequence that card \( C[i] \) can be appended to.

**Analysis** For an input of \( n \) cards, there are \( O(n) \) subproblems: \( S[0], \ldots, S[n - 1] \). Solving each subproblem requires iterating over all previous subproblems, for an \( O(n) \) time per subproblem. Thus in total, our runtime is \( O(n^2) \).

**More Practice DP problem**

The web page here [http://people.cs.clemson.edu/~bcdean/dp_practice/](http://people.cs.clemson.edu/~bcdean/dp_practice/) is an amazing resource for getting practice with dynamic programming. On there, you’ll find a number of dynamic programming problems as well as videos demonstrating solutions for each one.

**Closing remarks of DP**

Since DP considers all possibilities (exhaustively) one common way to think about DP is a smart exhaustive search. Smart because of memoization or bottom up solution and exhaustive because it does consider all possible solutions. Without the memoization or bottom up solution, many DPs are in fact, exponential algorithms, so be careful to not implement them without the optimization tricks we showed you!