Analysis of Algorithms I: Dynamic Programming

Xi Chen

Columbia University

Compared to Greedy Algorithms, Dynamic Programming (DP) is a more sophisticated scheme to attack optimization problems. We start with the following example: Rod cutting.

Input: A rod of length n inches and a sequence of prices $p_1, \ldots, p_n \geq 0$. Output: Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces: the value of a piece of length i is p_i. See an example on page 361: A rod of length $n = 4$ with 8 possible ways of cutting it. Seems to be a very difficulty problem: the number of feasible solutions is exponential in n. How can we compute the maximum revenue r_n efficiently?

The problem has the following Optimal Substructure:
An optimal solution to the problem contains within it optimal solutions to subproblems. Usually after the first choice, finding an optimal solution reduces to solving multiple subproblems. Here assume that the first piece has length k, where $1 \leq k \leq n$:

1. If $k = 1$, then the maximum revenue is $p_1 + r_{n-1}$;
2. If $k = 2$, then the maximum revenue is $p_2 + r_{n-2}$;
3. $\cdots$
4. If $k = n - 1$, then the maximum revenue is $p_{n-1} + r_1$;
5. If $k = n$, then the maximum revenue is $p_n + r_0$

where we set $r_0 = 0$ for convenience.

While at this moment, it is by no means clear which choice of k is the best (this is the crucial difference between DP and Greedy), we obtain the following useful recursive formula:

$$r_n = \max\left(p_1 + r_{n-1}, \ldots, p_{n-1} + r_1, p_n + r_0\right)$$

This gives us the following naive recursive algorithm:

1. If $n = 0$, return 0
2. Recursively compute $r_1, \ldots, r_{n-1}$ (with $r_0 = 0$)
3. Return $r_n = \max\left(p_1 + r_{n-1}, \ldots, p_{n-1} + r_1, p_n + r_0\right)$

However, if we denote its running time by $T(n)$, then

$$T(n) = O(n) + \sum_{i=1}^{n-1} T(i)$$

which solves to be exponential in n.

The reason why this recursive algorithm is so slow is due to its huge recursion tree. See Figure 15.3 on page 364. A quick observation reveals the following idea to speed it up: The first recursive call is made to compute r_3. During its execution, we made a recursive call to compute r_2. Once the algorithm gets r_3, it makes another (and totally unnecessary) recursive call to compute r_2 again! If we have stored the value of r_2, then there is no need to make this recursive call for r_2 at all.

Inspired by this observation, DP-Rod-Cutting(n):

```
1  Create $r[1 \dots n]$ (to store the max revenues $r_1, \dots, r_n$)
2  Set $r[1] = p_1$ (initialization)
3  For $i = 2$ to $n$ do
4      $r[i] = \max\left(p_1 + r[i-1], \dots, p_{i-1} + r[1], p_i\right)$
5  Return $r[n]$
```

The worst-case running time of this algorithm is $\Theta(n^2)$: there are $n-1$ for-loops and each has $\Theta(i)$ running time. The correctness can be proved by induction: Before the ith for-loop, we have $r[j] = r_j$ (max revenue from a rod of length j) for all $j : 1 \le j < i$.

Recovering an optimal solution: The DP algorithm computes the value of an optimal solution. But can we compute an optimal solution, here a way of cutting up a rod of length n to achieve the maximum revenue r_n, efficiently? It turns out that a DP algorithm can usually be easily modified to not only compute the optimal value but also construct an optimal solution efficiently as well.

For this purpose, we come back to the main recursive formula:

$$r_n = \max\left(p_1 + r_{n-1}, \ldots, p_{n-1} + r_1, p_n + r_0\right)$$

with $r_0 = 0$. Let $k \in [n]$ denote an index that maximizes $p_k + r_{n-k}$ (so that $r_n = p_k + r_{n-k}$), then to achieve the maximum revenue of r_n, we first cut the rod into two pieces: the first one has length k and the second one has length $n - k$. Then if we continue to cut the second rod optimally, the total revenue will be $p_k + r_{n-k} = r_n$ and we get an optimal solution. We call k an (instead of "the" because it may not be unique) optimal size of the first piece.

We modify DP-Rod-Cutting(n) as follows:

```
1 Create $r[1 \dots n]$ (store the max revenues $r_1, \dots, r_n$)
2 Create $k[1 \dots n]$ (store the optimal sizes of the first piece)
3 Set $r[1] = p_1$ and $k[1] = 1$
4 For $i = 2$ to $n$
5     $r[i] = \max\left(p_1 + r[i-1], \dots, p_{i-1} + r[1], p_i\right)$
6     $k[i] =$ any $k \in [i]$ such that $r[i] = p_k + r[i-k]$
7 While $n > 0$ (print out an optimal cut for a length-$n$ rod)
8     print $k[n]$ (optimal size of the first piece)
9     set $n$ to be $n - k[n]$
```

To summarize, problems for which DP works usually share the following critical properties:

1. Optimal substructure: After the first choice, finding an optimal solution reduces to solving multiple subproblems.
2. Overlapping subproblems: The total number of subproblems needed to solve is "small". (In Rod-cutting, we have n subproblems to solve in total: computing $r_1, \ldots, r_n$.) Do not solve the same subproblem repeatedly (avoid the naive recursive implementation). Solve each subproblem once and store the result in an array (or table as in the next example).

When developing a DP algorithm, we follow three steps:

1. Understand the optimization problem. Use its optimal substructure to derive a recursive formula for the optimal value in terms of optimal values of smaller subproblems.
2. Take advantage of the overlapping subproblems property. Use the recursive formula to compute the optimal values of all subproblems, usually in a bottom-up fashion. (For example, in Rod-cutting we compute from $r_1, r_2, \ldots$ to r_n.)
3. Maintain additional information (the optimal choice made in each subproblem) to construct an optimal solution efficiently. For example, in Rod-cutting we need to maintain the optimal sizes of the first piece, for rods of length $1, 2, \ldots, n$.

Longest Common Subsequence (LCS): Given two sequences

$$X = (x_1, \ldots, x_m) \quad \text{and} \quad Y = (y_1, \ldots, y_n)$$

determine the length of the longest common subsequence of X and Y, denoted by $\text{LCS}(X, Y)$. We use DP to give an efficient algorithm for $\text{LCS}(X, Y)$. Later by maintaining additional information, we will modify the algorithm so that it can output an LCS of X and Y efficiently.

Here a subsequence of a given sequence X is a sequence that can be derived from X by deleting zero or more characters (without changing the order of the remaining characters). For example, if

$$X = (A, B, C, B, D, A, B) \quad \text{and} \quad Y = (B, D, C, A, B, A)$$

then (B, C, A) is a common subsequence of both X and Y. Again, the LCS problem seems to be difficult because a sequence X of length m has 2^m possible subsequences. If we enumerate all subsequences of X and find the longest one that also appears in Y, the running time will be clearly exponential.

It turns out that the LCS problem has the following optimal substructure. Here we use LCS(X, Y) to denote the length of the longest common subsequence of X and Y:

Theorem

Let $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ be two sequences with $m, n \geq 1$. Denote $X' = (x_1, \ldots, x_{m-1})$ and $Y' = (y_1, \ldots, y_{n-1})$. Then we have the following two cases:

1. *If $x_m = y_n$, $LCS(X, Y) = 1 + LCS(X', Y')$.*
2. *If $x_m \neq y_n$, $LCS(X, Y) = \max\left(LCS(X, Y'), LCS(X', Y)\right)$.*

Assume $x_m = y_n$, and let $Z = (z_1, \ldots, z_k)$ denote an LCS of X and Y, so $\text{LCS}(X, Y) = k$. First we show that $z_k = x_m = y_n$. If $z_k \neq x_m$, we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the assumption that Z is longest. Thus, we get $z_k = x_m = y_n$. Let $Z' = (z_1, \ldots, z_{k-1})$ then it is easy to check that Z' is a common subsequence of X' and Y'. It suffices to show that Z' is an LCS of X' and Y'. This can be proved by contradiction. If X' and Y' have a common subsequence Z'' of length $> k - 1$, then appending $x_m = y_n$ to Z'' produces a common subsequence of X and Y, with length $> k$, contradicting with the assumption that Z is an LCS of X and Y.

Now consider the case when $x_m \neq y_n$. One direction is easy:

$$\mathrm{LCS}(X, Y) \geq \max\left(\mathrm{LCS}(X, Y'), \mathrm{LCS}(X', Y)\right)$$

To prove the other direction, we let $Z = (z_1, \ldots, z_k)$ denote an LCS of X and Y and consider the following two cases: If $z_k \neq x_m$, then Z must be a common subsequence of X' and Y and thus, $\mathrm{LCS}(X, Y) = k \leq \mathrm{LCS}(X', Y)$. If $z_k \neq y_n$, similarly we can show that $\mathrm{LCS}(X, Y) \leq \mathrm{LCS}(X, Y')$. Combining the two cases we get

$$\mathrm{LCS}(X, Y) \leq \max\left(\mathrm{LCS}(X, Y'), \mathrm{LCS}(X', Y)\right)$$

The theorem is now proved.

This gives us the following naive recursive algorithm $\mathrm{LCS}(X, Y)$:

```
1  If $m = 0$ or $n = 0$, return 0
2  If $x_m = y_n$ then
3      return $1 + \mathrm{LCS}(X', Y')$ (one recursive call)
4  else
5      return $\max\left(\mathrm{LCS}(X, Y'), \mathrm{LCS}(X', Y)\right)$ (two recursive calls)
```

Unfortunately this naive implementation ends up with exponential running time in the worst case, because it makes multiple recursive calls to solve the same subproblem again and again.

Actually, each of the recursive calls in the naive recursive algorithm is made to compute the following number:

$$\mathrm{LCS}(X_i, Y_j), \quad \text{for some } i, j : 0 \le i, j \le n$$

where we use $X_i = (x_1, \ldots, x_i)$ to denote the ith prefix of X and $Y_j = (y_1, \ldots, y_j)$ to denote jth prefix of Y. From now on we will use $c[i, j]$ to denote $\mathrm{LCS}(X_i, Y_j)$ for convenience.

Using the theorem proved earlier, we immediately get the following useful recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Recall our goal here is to compute $\text{LCS}(X, Y) = c[m, n]$.

To compute $c[m, n]$, we set up a table $c[0 \dots m, 0 \dots n]$ to store $c[i, j]$, $i, j : 0 \le i \le m$ and $0 \le j \le n$. By definition, we have

$$c[0, j] = c[i, 0] = 0, \quad \text{for all } i, j$$

so we can set these entries of c, its row 0 and column 0, to be 0. The trick is then to compute the rest of the entries of c in the row-major order: fill in row 1 from $c[1, 1]$ to $c[1, n]$; fill in row 2 from $c[2, 1]$ to $c[2, n]$; … fill in row m from $c[m, 1]$ to $c[m, n]$. In this order, whenever we need to fill in an entry $c[i, j]$, all three entries $c[i - 1, j - 1], c[i, j - 1]$ and $c[i - 1, j]$ must have already computed so we can invoke the recursive formula to compute $c[i, j]$ very efficiently in $O(1)$ time.

```
1  Create two tables $c[0 \ldots m, 0 \ldots n]$ and $b[1 \ldots m, 1 \ldots n]$
2  Set $c[i, 0]$ and $c[0, j]$ to be 0 for all $i, j$
3  for $i = 1$ to $m$ (follow the row-major order)
4      for $j = 1$ to $n$
5          if $x_i = y_j$
6              set $c[i, j] = c[i-1, j-1] + 1$ and $b[i, j] = \nwarrow$
7          else if $c[i-1, j] \geq c[i, j-1]$
8              set $c[i, j] = c[i-1, j]$ and $b[i, j] = \uparrow$
9          else
10             set $c[i, j] = c[i, j-1]$ and $b[i, j] = \leftarrow$
```

Here we also maintain a table $b[1 \dots m, 1 \dots n]$ to help construct a longest common subsequence efficiently. By the end of the execution, the entries of the b table has the following meaning:

1. Case $b[i,j] = \uparrow$: To get an LCS of X_i and Y_j, we only need to construct an LCS of X_{i-1} and Y_j;
2. Case $b[i,j] = \leftarrow$: To get an LCS of X_i and Y_j, we only need to construct an LCS of X_i and Y_{j-1};
3. Case $b[i,j] = \nwarrow$: To get an LCS of X_i and Y_j, we only need to get an LCS of X_{i-1} and Y_{j-1} and append $x_i = y_j$ to it.

Using the table b, we can easily construct an LCS of X and Y. Start from the $[m, n]$th entry of b, its lower-right corner, and follow the arrows in b. Every time we arrive at an entry $[i, j]$ with an

$$b[i, j] = \nwarrow$$

print $x_i = y_j$. At the end, reverse the string and we get an LCS.