Dynamic Programming

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

These notes sketch the content of the 13th lecture. The topics of this lecture are treated in any introductory book on Algorithms. You may refer to Chapter 15 of *Introduction to Algorithms, 3rd Edition*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2009 for a more detailed explanation.

1 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

In contrast, dynamic programming applies when subproblems overlap, that is, when subproblems share subsubproblems.

In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.

A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table. Thus, avoiding the work of recomputing the answer every time it solves each subsubproblem.

2 Fibonacci numbers

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

The goal is to compute F_n.

Consider the following trivial recursive algorithm.

Fibonacci(**n**)
1. if $(n == 0)$ return 0
2. if $(n == 1)$ return 1
3. else return Fibonacci(n-1) + Fibonacci(n-2)

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

The recurrence $T(n) \geq F_n$.

$$T(n) > 2T(n-2) > 2(2T(n-4)) = 2^2T(n-4)$$

$$> 2^2(2T(n-6)) = 2^3T(n-6) > \dots$$

$$> 2^kT(n-2k) > \dots$$

Thus, for n even, we get

$$T(n) > 2^{\frac{n}{2}}T(0) = 2^{\frac{n}{2}}c_0$$

while for n odd, we get

$$T(n) > 2^{\frac{n-1}{2}}T(1) = 2^{\frac{n-1}{2}}c_1$$

Memoization is the trick that allows to reduce the time complexity. Whenver we compute a Fibonacci number, we add it to a dictionary. Everytime we need a Fibonacci number, we first check the dictionary, if the number is not there, compute it.

FibonacciMemo(**n**)

1. if ($n == 0$) return 0
2. if ($n == 1$) return 1
3. if $M[n] \neq NULL$ return M[n]
4. M[n] = FibonacciMemo (n-1) + FibonacciMemo (n-2)
5. return M[n]

This algorithm requires linear time and space.

Actually, there is a more direct bottom-up approach which uses linear time and constant space. How?

This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems.

We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions.

We solve each sub-problem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

In our Fibonacci problem this approach correspond to compute an array F which entry $F[i]$ is the ith Fibonacci number.

We can fill this array from left (smaller subproblems) to right (larger subproblems). Entry $F[i]$ requires only entries $F[i-1]$ and $F[i-2]$.

3 Rod cutting

▶ **Problem 1.** *Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells.*

Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

Given a rod of lenght n and for any $i \in [1, n]$, the price p_i of a rod of lenght i, the goal is that of determining the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

The bottom-up DP solution of the problem is as follows.

Our goal is to fill an array $r[0, n]$, where entry $r[i]$ stores the maximum revenue obtainable with a rod of size i.

Assuming we have already solved all the subproblems of size $j < i$, what's the value of $r[i]$?

Let's list all the possibilities. We do not cut. The revenue is p_i (or, if you prefer, $p_i + r[0]$). We make a cut of lenght 1 and we optimally cut the remaining rod of size $i - 1$. The revenue in this care is $p_1 + r[i - 1]$. We make a cut of lenght 2 and we optimally cut the remaining rod of size $i - 2$. The revenue is $p_2 + r[i - 2]$. And so on.

The value of $r[i]$ is the maximum among all these revenues.
The pseudocode of this algorithm is reported below.

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

The algorithm runs in $\Theta(n^2)$ time.

The algorithm computes only optimal revenue but not a cut that gives such a revenue. The algorithm can be easily nodified to obtain the cutting. How?

Consider the following example.

lenght	0	1	2	3	4	5	6	7	8	9	10
price	0	1	5	8	9	10	17	17	20	24	30
revenue	0	1	5	8	10	13	17	18	22	25	30
cut	0	1	2	3	2	2	6	1	2	3	10

4 Longest Common Subsequence

▶ **Problem 2** (Longest Common Subsequence). *Given two sequences S_1 and S_2, find the length of longest subsequence present in both of them.*

As an example, consider $S_1 =$ `ABCBDAB` and $S_2 =$ `BDCABA`. A longest common subsequence is `BCBA`. There may be other, e.g., `BCAB`.

The subproblems are prefixes of the two sequences.

Consider two prefixes $S_1[1,i]$ and $S_2[1,j]$, our goal is to compute $\mathsf{LCS}(S_1[1,i], S_2[1,j])$.

Assume we already know $\mathsf{LCS}(S_1[1,i-1], S_2[1,j-1])$, $\mathsf{LCS}((S_1[1,i], S_2[1,j-1]))$, and $\mathsf{LCS}(S_1[1,i-1], S_2[1,j])$, i.e., three smaller subproblems.

If $S_1[i] = S_2[j]$, we can extend a longest common subsequence of $S_1[1,i-1]$ and $S_2[1,j-1]$ by adding the character $c = S_1[i]$. For example, `ABCB` and `BDCAB`. The lcs of $S_1[1,i-1] =$ `ABC` and $S_2[1,j-1] =$ `BDCA` has lenght 2 and can be extended with character $c =$ `B`.

We could also use $\mathsf{LCS}(S_1[1,i], S_2[1,j-1])$, and $\mathsf{LCS}(S_1[1,i-1], S_2[1,j])$ ignoring the match on the last character, but these two cannot be longer.

Instead, if $S_1[i] \neq S_2[j]$, we can only consider $\mathsf{LCS}(S_1[1,i], S_2[1,j-1])$, and $\mathsf{LCS}(S_1[1,i-1], S_2[1,j])$. We take the longest.

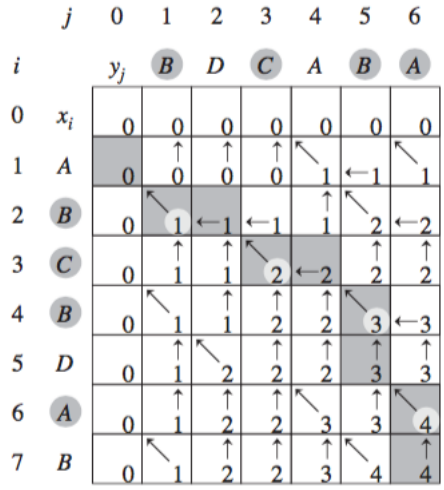
Summarizing,

$$\mathsf{LCS}(S_1[1,i], S_2[1,j]) = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \mathsf{LCS}(S_1[1,i-1], S_2[1,j-1]) + 1 & S_1[i] = S_2[j] \\ \max(\mathsf{LCS}(S_1[1,i], S_2[1,j-1]), \mathsf{LCS}(S_1[1,i-1], S_2[1,j])) & \text{otherwise} \end{cases}$$

Below is the pseudocode of the algorithm implementing the above recurrence. We use a matrix c to store the solution of the subproblems and a matrix b to keep track of the computed solutions.

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Figure below show the execution of the algorithm on $S_1 =$ `ABCBDAB` and $S_2 =$ `BDCABA`.



5 0/1 Knapsack

▶ **Problem 3** (0/1 Knapsack)**.** *We are given n items. Each item i has a value v_i and a weight w_i. We need put a subset of these items in a knapsack of capacity C to get the maximum total value in the knapsack.*

We can use the following solutions

1. if C is small, we can use *Weight Dynamic Programming*. The time complexity is $\Theta(Cn)$).
2. if $V = \sum_i v_i$ is small, we can use *Price Dynamic Programming*. The time complexity is $\Theta(Vn)$).
3. If both are large, we can use *branch and bound*.

The idea of the *Weight DP* algorithm is to fill a $(n+1) \times (C+1)$ matrix K. Let $K[i, A]$ be the max profit for weight at most A using items from 1 up to i. Thus, we have that

$$K[i, A] = \max(K[i-1, A], K[i-1, A - w[i]] + v[i]).$$

The last entry $K[C, n]$ contains the solution.

Here an example. Consider a knapsack of capacity $C = 7$ and the following items.

weight	value
1	1
3	4
4	5
5	7

The matrix K is as follows.

value	weight	0	1	2	3	4	5	6	7
1	1	0	1	1	1	1	1	1	1
4	3	0	1	1	4	5	5	5	5
5	4	0	1	1	4	5	6	6	9
7	5	0	1	1	4	5	7	8	9

The idea of the *Profit DP* algorithm is similar. We use a $(n+1) \times (V+1)$ matrix K. Let $K[V, i]$ be the min weight for profit at least V using items $1, \leq, i$. Thus, we have

$$K[A, i] = \min(K[A, i-1], K[A - p[i], i-1] + w[i]).$$

The solution is $\max\{a : K[A, n] \leq C\}$.

6 IWGBS - 0110SS

▶ **Problem 4** (IWGBS - 0110SS)**.** *Dor is IWGolymp student so he has to count in how many ways he can make N digit numbers that is formed by ones and zeroes. But zeroes can not be next to each other. Help to him in how many different numbers can he make.*

For example, $N = 3$: `101`, `010`, `111`, `110`, `011`

Let $F(N)$ be number we are looking for. Any number with the desired property can be obtained either: 1) by taking any $n-1$ digits number and by appending a 1, or by taking any $n-2$ number and by appending 10. Thus, $F(N) = F(n-1) + F(n-2)$ (Looks familiar?). We also know that $F(1) = 2$ and $F(0) = 1$.