Dynamic Programming – Overview

- Not a specific algorithm, but a technique (like Divide-and-Conquer and Greedy algorithms)
- Developed back in the day (1950s) when "*programming*" meant "*tabular method*" (like linear programming)
- Used for optimization problems
  - Find a solution with the optimal value
  - Minimization or maximization

Dynamic Programming

Four-step (two-phase) method:

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 in a bottom-up fashion
4. Construct an optimal solution from computed information

The rod cutting problem

Problem statement:

- Input:
  1) a rod of length n
  2) an array of prices p_i for a rod of length i for $i = 1, \ldots, n$.

- Output:
  1) the maximum revenue r_n obtainable for rods whose length sum to n
  2) optimal cut, *if necessary.*

In short,

How to cut a rod into pieces in order to maximize the revenue you can get?

Example

rod length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30
r_i	1	5	8	10	13	17	18	22	25	30
s_i	1	2	3	2	2	6	1	2	3	10

- r_i: maximum revenue of a rod of length i
- s_i: optimal size of the first piece to cut

The rod cutting problem

A brute-force solution:

cut up a rod of length n *in* 2^{n-1} *different ways*

Cost: $\Theta(2^{n-1})$

The rod cutting problem

Dynamic Programming – Phase I:

- Since every optimal solution r_n has a leftmost cut with length i, the optimal revenue r_n is given by

$$
\begin{aligned}
r_n &= \max\{p_1 + r_{n-1},\ p_2 + r_{n-2},\ \ldots,\ p_{n-1} + r_1,\ p_n + r_0\} \\
&= \max_{1 \le i \le n} \{p_i + r_{n-i}\} && (1) \\
&= p_{i_*} + r_{n-i_*} && (2)
\end{aligned}
$$

where

$$
\begin{aligned}
i_* &= \text{the index attains the maximum} \\
&= \text{the length of the leftmost cut}
\end{aligned}
$$

The rod cutting problem

Dynamic Programming – Phase II:

- How to compute r_n by the expression (1) ?
  1. Recursive solution:
     - top-down, no memoization
     - Cost:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = \Theta(2^n)$$

  2. Iterative solution
     - bottom-up, memoization (Pseudocode – see next page)
     - Cost:

$$T(n) = \Theta(n^2)$$

The rod cutting problem

```
cut-rod(p,n)
// an iterative (bottom-up) procedure for finding ``r'' and
// the optimal size of the first piece to cut off ``s''
Let r[0...n] and s[0...n] be new arrays
r[0] = 0
for j = 1 to n
    // find q = max{p[i]+r[j-i]} for 1 <= i <= j
    q = -infty
    for i = 1 to j
        if q < p[i] + r[j-i]
           q = p[i] + r[j-i]
           s[j] = i
        end if
    end for
    r[j] = q
end for
return r and s
```

The rod cutting problem

Example

rod length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30
r_i	1	5	8	10	13	17	18	22	25	30
s_i	1	2	3	2	2	6	1	2	3	10

- r_i: maximum revenue of a rod of length i
- s_i: optimal size of the first piece to cut
  Note: $s_i = i_*$ in expression (2).

Longest Common Subsequence (LCS)

Problem:

Input: *Sequences*

$$\begin{aligned} X_m &= \langle x_1, x_2, x_3, \ldots, x_m \rangle \\ Y_n &= \langle y_1, y_2, \ldots, y_n \rangle \end{aligned}$$

Output: *longest common subsequence (LCS)* *of* X_m *and* Y_n

LCS

A brute-force solution:

- For every subsequence of X_m, check if it is a subsequence of Y_n.
- Running time: $\Theta(n \cdot 2^m)$
- Intractable!

LCS

DP – step 1: *characterize the structure of an optimal solution*

Let $Z_k = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of

$$X_m = \langle x_1, x_2, \ldots, x_m \rangle \quad \text{and} \quad Y_n = \langle y_1, \ldots, y_n \rangle$$

Then

- Case 1. If $x_m = y_n$, then

  (a) $z_k = x_m = y_n$

  (b) $Z_{k-1} = \langle z_1, z_2, \ldots, z_{k-1} \rangle = \mathsf{LCS}(X_{m-1}, Y_{n-1})$

- Case 2. If $x_m \neq y_n$, then

  (a) $z_k \neq x_m \Longrightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$

  (b) $z_k \neq y_n \Longrightarrow Z_k = \mathsf{LCS}(X_m, Y_{n-1})$

In words, the optimal solution to the (whole) problem contains within it the otpimal solutions to subproblems = **the optimal substructure property**

LCS

DP – step 2: *recursively define the value of an optimal solution*

- Define
$$c[i,j] = \text{length of LCS}(X_i, Y_j)$$
for $i = 0, 1, \ldots, m$ and $j = 0, 1, \ldots, n$
- $c[m,n] = \text{length of LCS}(X_m, Y_n)$
- By the optimal structure property
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \text{ (initials)} \\ c[i-1,j-1]+1 & \text{if } x[i] = y[j] \quad \text{(Case 1)} \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } x[i] \neq y[j] \quad \text{(Case 2)} \end{cases}$$
- Meanwhile, create $b[i,j]$ to record the optimal subproblem solution chosen when computing $c[i,j]$

LCS

DP – step 3: *compute $c[i, j]$ (and $b[i, j]$) in a bottom-up approach*

- Compute $c[i, j]$ and $b[i, j]$ in a bottom-up approach.
  - $c[i, j]$ is the length of LCS(X_i, Y_j)
  - $b[i, j]$ shows how to construct the corresponding LCS(X_i, Y_j)
- Cost:
  - Running time: $\Theta(mn)$
  - Space: $\Theta(mn)$

LCS

```
LCS-length(X,Y)
set c[i,0] = 0 and c[0,j] = 0
for i = 1 to m // Row-major order to compute c and b arrays
    for j = 1 to n
        if X(i) = Y(j)
           c[i,j] = c[i-1,j-1] + 1
           b[i,j] = 'Diag'      // go to up diagonal
        elseif c[i-1,j] >= c[i,j-1]
           c[i,j] = c[i-1,j]
           b[i,j] = 'Up'      // go up
        else
           c[i,j] = c[i,j-1]
           b[i,j] = 'Left'      // go left
        endif
    endfor
endfor
return c and b
```

DP – step 4: *construct an optimal solution from computed information*

Example: $X_6 = \langle A, B, C, B, D, A, B \rangle$ and $Y_6 = \langle B, D, C, A, B, A \rangle$

$c[\cdot,\cdot] + b[\cdot,\cdot]:$

i \ j		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

(1) Length of LCS $=$ `c[7,6]` $= 4$
(2) By the b-table ("$\uparrow, \leftarrow, \nwarrow$"), the LCS is $B\,C\,B\,A$