Dynamic Programming (Continued)

Matrix-chain-order $O(n^3)$ dynamic prog algorithm

Pseudo-code that we wrote yesterday has bottom-up structure

Alternative way of coding (top-down). Recursive Program

structure is similar to recursive describing the optimal solution.

We need to remember solutions to subproblems that we have already solved

Memoization (word 'memo' means here writing something for future reference)

Pseudo code

```
matrix-chain-order-memo(p)
    n = p.length - 1
    m[1...n, 1...n] is new two dimensional array

    for i = 1 to n do
        for j = 1 to n do
            m[i, j] = ∞

    matrix-chain-order-memo-aux(p, m, 1, n)
```

```
matrix-chain-order-memo-aux(P, m, i, j)
  if (m[i, j] < ∞)
        return m[i, j]   // — ①

  if  i == j
        return 0         // — ②

  temp = ∞
  for l = i to j-1 do
      a = matrix-chain-order-memo-aux(P, m, i, l)
      b = matrix-chain-order-memo-aux(P, m, l+1, j)
      c = a + b + P[i]·P[l+1]·P[j+1]
      if (c < temp)
           temp = c

  m[i, j] = temp
  return m[i, j]
```

Time complexity analysis is not obvious.

We are interested in estimating Time complexity of matrix-chain.....-memo-aux(P, m, 1, n)

All the recursive calls arising in the computation of above call are divided into two kinds

(i) Those calls which are returned either at ① or ②

(ii) Those which execute the for loop.

- Each call is executed as call of (ii) kind at most once.

$\Longrightarrow$ There are $O(n^2)$ calls of (ii) kind

- All calls are made by calls of 2^{nd} kind only

Each call of 2^{nd} kind can make at most $O(n)$ calls.

$\Rightarrow$ Total no. of calls is $O(n^3)$

Total time required, summed over all recursive calls, is estimated as follows.

Calls of kind (i) $\longrightarrow$ $O(1)$ time — total $O(n^3)$

Calls of kind (ii) $\longrightarrow$ $O(n)$ time // excluding time to execute recursive calls, because time in all calls is being summed separately

$\longrightarrow$ $O(n^2) \cdot O(n) = O(n^3)$

$\Longrightarrow$ The algorithm matrix-chain-order-memo-aux $(p, m, 1, n)$ works in $O(n^3)$ time.

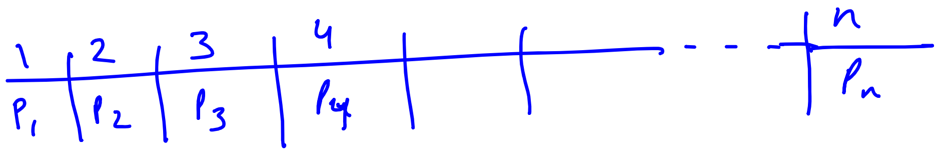
Another Example of dynamic Programming

Rod cutting Problem

We are given a rod of integer length (n)

We need to sell this rod to get some revenue

The rod can be sold as a whole or it can be cut into pieces of integer length and pieces can be sold



← i length of the rod

← p_i Price of rod of length i.

(possibly)
Cut the rod and sell pieces to maximize our revenue.

Concrete data example

$i \rightarrow$	1	2	3	4	5
$p_i \rightarrow$	2	1	7	3	6

Pieces	Revenue
1 + 1 + 1 + 1 + 1	10
2 + 1 + 1 + 1	7
2 + 2 + 1	4
3 + 1 + 1	11
3 + 2	8
4 + 1	5
5	6

No. of possible partitions of this rod is exponential (in n)

$R[i] \equiv$ the maximum revenue that can be earned from rod of length i

$$R[0] = 0$$

$$R[n] = \max_{1 \le l \le n} \{ p_l + R[n-l] \}$$

$R[0] = 0$

$$R[n] = \max_{1 \le \ell \le n} \{ P_\ell + R[n-\ell] \}$$

for $n > 0$

$$R[n] \geq \max_{\ell} \underbrace{\{ P_\ell + R[n-\ell] \}}$$

This gives a way of realizing this revenue (cut into size ℓ, and repeat the procedure with rod of length $n-\ell$)

We need to show

$$R[n] \leq \max_{\ell} \{ P_\ell + R[n-\ell] \} \quad \text{—— } (1)$$

Consider any cutting which gives revenue $R[n]$ place cuts on the rod from left to right.

Consider the leftmost cut, it will be at some i, $1 \le i \le n$

$$R[n] = P_i + \text{the revenue realized from rod of length } n-i$$

$$\leq P_i + R[n-i]$$

$\exists i$

$$R[n] \leq P_i + R[n-i]$$

$$\boxed{\Longrightarrow \quad \text{Equation } (1).}$$

There are $n-$ subproblems $R[0], R[1], \ldots, R[n-1]$ for solving $R[n]$

To compute $R[i]$, we need to look-up i subproblems. $O(n)$ many subproblems

$\Longrightarrow$ $O(n^2)$ Dynamic Programming Algorithm.

Exercise: Consider writing bottom-up and top-down dynamic programming algorithms and pseudocodes for this problem which compute the maximum revenue as well as the cuts that realize this revenue.