Unit 6.1 Dynamic Programming

Algorithms

EE3980

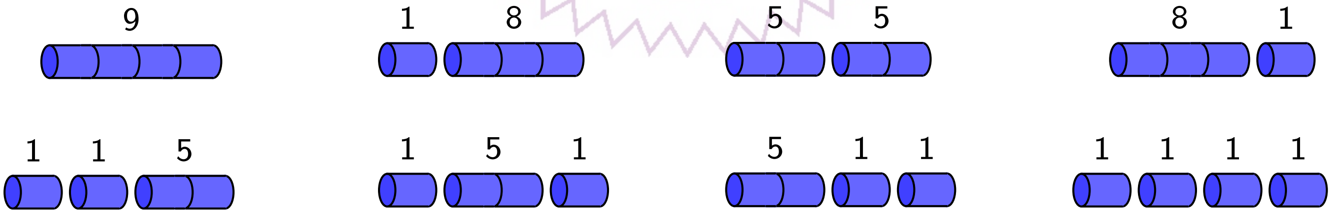
Apr. 23, 2018

Rod Cutting Problem

- Rod cutting problem
  Given a rod of n inches and a price table, p_i, $i = 1, \ldots, n$, determine the maximum revenue r_n obtainable to cutting the rod and selling the pieces.
- Example of the price table for rods.

Length, inches	1	2	3	4	5	6	7	8	9	10
Price, Dollars	1	5	8	9	10	17	17	20	24	30

- Example of cutting a rod of length of 4 inches.
  - Eight different ways of cutting.
  - Maximum revenue is 10.



Rod Cutting Problem, Formulation

- Given a rod of length n inches, there are totally 2^{n-1} ways of cutting.
- In brute-force approach, the maximum revenue of all these cutting is the optimal solution.
- Using recursive function, we can formulate the solution as

$$r_n = \max\{p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\}, \tag{6.1.1}$$

  where r_k is the maximum revenue of cutting the rod of length k, and p_k is the price of length k rod.
- This is a recursive formula and it evaluates all possible rod-cutting solutions and finds the maximum revenue.

Rod Cutting Problem, Recursive Algorithm

Rod_R 6.1.1. Recursive Rod-cutting

```
Algorithm rod_R(p, n)
// Find the maximum revenue for cutting rod of length n. p[1 : n] is the price table.
{
    if (n = 0) return 0;
    max := p[n]; // no cut.
    for i := 1 to n − 1 do { // check all possible cutting using recursion.
        if (p[i]+ rod_R(p, n − i) > max) then max = p[i]+ rod_R(p, n − i);
    }
    return max;
}
```

- Example of Rod_R$(p, 4)$ unrolling

Rod_R$(p, 4) \Rightarrow$	$p[1]$+Rod_R$(p, 3)$	$p[2]$+Rod_R$(p, 2)$	$p[3]$+Rod_R$(p, 1)$	$p[4]$
Rod_R$(p, 3) \Rightarrow$	$p[1]$+Rod_R$(p, 2)$	$p[2]$+Rod_R$(p, 1)$	$p[3]$	
Rod_R$(p, 2) \Rightarrow$	$p[1]$+Rod_R$(p, 1)$	$p[2]$		
Rod_R$(p, 1) \Rightarrow$	$p[1]$			

- As it is, rod_R(p, n) may be called many times for i, $1 <= i < n$.
- This inefficiency can be improved using dynamic programming method.

Rod Cutting Problem, Top-Down Dynamic Programming

- The efficiency of the recursive rod-cutting algorithm can be improved significantly using a revenue array, $r[0:n]$.
- Before calling this rod_TD(p,n,r) function, the revenue array should be initialized as

$$r[i] = \begin{cases} 0, & \text{if } i = 0, \\ -\infty, & \text{otherwise.} \end{cases}$$

Rod_TD 6.1.2. Rod-cutting top-down dynamic programming

```
Algorithm rod_TD(p, n, r)
// Find the maximum revenue for cutting rod of length n.
{
    if (r[n] > 0) return r[n] ; // if prior evaluation is done, return value.
    max := p[n] ; // no cut.
    for i := 1 to n − 1 do { // check all possible cutting using recursion.
        if (p[i]+ rod_TD(p, n − i, r) > max) then
            max := p[i]+ rod_TD(p, n − i, r) ;
    }
    r[n] := max; // record max revenue in r array.
    return max;
}
```


Rod Cutting Problem, Bottom-Up Dynamic Programming

- For the top-down dynamic function, in addition to the proper initialization of the revenue, $r[0:n]$, table, the function should be called as rod_TD(p,n,r);
- A corresponding bottom-up dynamic programming algorithm is as the following.

Rod_BU 6.1.3. Rod-cutting bottom-up dynamic programming

```
Algorithm rod_BU(p, n, r)
// Find the maximum revenue for cutting rod of length n.
{
    r[0] := 0 ;
    for i := 1 to n do {
        max := −∞ ;
        for j := 1 to i do {
            if (p[j] + r[i − j] > max) then max := p[j] + r[i − j] ;
        }
        r[i] := max;
    }
    return r[n] ;
}
```

Rod Cutting Problem, Complexities

- For the `rod_BU`(p, n, r) algorithm, **for** loop on lines 5-11 executes n times.
- The inner **for** loop on lines 7-9 executes $\dfrac{n(n+1)}{2}$ times overall.
- Thus the computational complexity is $\Theta(n^2)$.
- The space complexity is $\Theta(n)$ due to the $r[0:n]$ and $p[1:n]$ arrays.
- For the `rod_TD`(p, n, r) algorithm, both time and space complexities are the same of the `rod_BU`(p, n, r) algorithm asymptotically.
- In both `rod_BU`(p, n, r) and `rod_TD`(p, n, r) algorithms, the maximum revenue array, $r[1:n]$, is found. But, not the actual cutting solution. By adding a solution table, $s[1:n]$, the following algorithm finds the cutting solution as well.

Rod Cutting Problem, Maximum Revenue and Cutting

Rod_R 6.1.4. Rod-cutting with solution

```
1 Algorithm rod_SBU(p, n, r, s)
2 // Find the maximum revenue for cutting rod of length n.
3 {
4       r[0] := 0;
5       for i := 1 to n do {
6           max := −∞;
7           for j := 1 to i do {
8               if (p[j] + r[i − j] > max) then {
9                   max := p[j] + r[i − j];
10                  s[i] := j;
11              }
12          }
13          r[i] := max;
14      }
15      return r[n];
16 }
```

Rod Cutting Problem, Maximum Revenue and Cutting

- Once the cutting solution is found by the `rod_SBU`(p, n, r, s) algorithm, the following algorithm can be used to print out the cutting solution.

Rod_PS 6.1.5. Rod-cutting printing solutions

```
Algorithm rod_PS(n, s)
// Printing the cutting solution store in the solution table, s[1 : n].
{
    while (n > 0) do {
        write s[n];
        n := n − s[n];
    }
}
```


Rod Cutting Problem, Solution Example

- The algorithm `rod_SBU`(p, n, r, s) has the same complexities as the `rod_BU`(p, n, r) algorithm.
  - Time complexity: $\Theta(n^2)$,
  - Space complexity: $\Theta(n)$.
- Solution example:
  Assuming $n = 10$, the following table lists the price table p, maximum revenue table r, solution table s, and the cutting solutions for various rod lengths, $1 \le i \le 10$.

i	1	2	3	4	5	6	7	8	9	10
$p[i]$	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
Cuts:	1	2	3	2	2	6	1	2	3	10
				2	3		6	6	6	

Matrix Multiplication

- Given two matrices, A and B, each of dimensions $p \times q$ and $q \times r$, respectively, i.e., $A[1:p, 1:q]$ and $B[1:q, 1:r]$. The product $C = A \times B$ has the dimension of $p \times r$, $C[1:p, 1:r]$, and it can be found by

$$C[i,j] = \sum_{k=1}^{q} A[i,k] \cdot B[k,j], \qquad 1 \le i \le p, 1 \le j \le r. \tag{6.1.2}$$

  There are $p \times r$ elements in C and each takes q multiplications. Thus, the total number of multiplications to form the resultant matrix is $p \cdot q \cdot r$.
- Given thee matrices $A_1[1:10, 1:100]$, $A_2[1:100, 1:5]$, and $A_3[1:5, 1:50]$, the product of these three matrices, $B = A_1 \cdot A_2 \cdot A_3$, can be formed in two different ways.

$$B = (A_1 \cdot A_2) \cdot A_3 \tag{6.1.3}$$
$$= A_1 \cdot (A_2 \cdot A_3) \tag{6.1.4}$$

  Though the resulting matrix is identical, the number of operations to get matrix B is different.

Matrix-Chain Multiplication Problem

- Using Eq. (6.1.3),

$A_{12} = A_1[1:10, 1:100] \cdot A_2[1:100, 1:5]$	$10 \times 100 \times 5 = 5000$ multiplications
$B = A_{12}[1:10, 1:5] \cdot A_3[1:5, 1:50]$	$10 \times 5 \times 50 = 2500$ multiplications
Total	7500 multiplications

- Using Eq. (6.1.4),

$A_{23} = A_2[1:100, 1:5] \cdot A_3[1:5, 1:50]$	$100 \times 5 \times 50 = 25000$ multiplications
$B = A_1[1:10, 1:100] \cdot A_{23}[1:100, 1:50]$	$10 \times 100 \times 50 = 50000$ multiplications
Total	75000 multiplications

- The order of multiplications can make significant difference in computing the resulting product.
- The matrix-chain multiplication problem is to find the sequence of matrix multiplications for a given matrix chain, $A_1 \cdot A_2 \cdots A_n$, each with dimensions $p_{i-1} \times p_i$, such that the number of scalar multiplications is minimum.

Matrix-Chain Multiplication Problem, Analysis

- Given a chain of matrices, $A_1, A_2, \ldots, A_n$, the number of possible sequences, $P(n)$, can be shown to be

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases} \tag{6.1.5}$$

  - It is shown that $P(n) \geq 2^{n-1}$. Thus, $P(n)$ is $\Theta(2^n)$.
  - Brute force approach is very inefficient.
- Let the dimensions of the matrices A_i, $1 \leq i \leq n$, be $p_{i-1} \times p_i$.
  - These dimensions can be stored in the array $p[0:n]$.
- Let the minimum number of scalar products of performing matrix-chain, $A_i \cdot A_{i+1} \cdots A_{j-1} \cdot A_j$ be $m(i,j)$, then

$$m(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \displaystyle\min_{i \leq k < j} \{m(i,k) + m(k+1,j)\} + p_{i-1} \cdot p_k \cdot p_j & \text{if } i < j. \end{cases} \tag{6.1.6}$$

  - This is to try all groupings, $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, and find the minimum recursively.

Matrix-Chain Multiplication Problem, Recursive Algorithm

- Eq. (6.1.6) can be translated into a recursive algorithm as the following.

Algorithm 6.1.6. Recursive matrix-chain multiplication.

```
Algorithm MCM_R(i, j, n, p)
// To find the minimum scalar multiplication for a matrix chain.
{
    if (i = j) return 0;
    u := ∞;
    for k := i to j − 1 do {
        v := MCM_R(i, k, n, p)+ MCM_R(k + 1, j, n, p) + p[i − 1] ∗ p[k] ∗ p[j];
        if (v < u) u := v;
    }
    return u;
}
```

- Again, this recursive algorithm is inefficient due to repeated evaluation of the MCM_R function with the same arguments.
- Using the top-down dynamic programming technique, this inefficiency can be avoided by saving the value into an array, in this case, it needs to be a two-dimensional matrix, $m[i,j]$.

Matrix-Chain Multiplication, Top-Down Approach

- The top-down dynamic programming approach to solve the matrix-chain multiplication problem is shown below.

Algorithm 6.1.7. Top-down matrix-chain multiplication.

```
Algorithm MCM_TD(i, j, n, p, m)
// To find the minimum scalar multiplication for a matrix chain.
{
    if (m[i, j] ≥ 0) return m[i, j];
    u := ∞;
    for k := i to j − 1 do {
        v := MCM_TD(i, k, n, p)+ MCM_TD(k + 1, j, n, p) + p[i − 1]· p[k]· p[j];
        if (v < u) u := v;
    }
    m[i, j] := u;
    return m[i, j];
}
```

- Before MCM_TD$(1, n, n, p, m)$ is called from the main function, initialization of $m[i][i] = 0$, $1 \le i \le n$, should be performed.
- Also note that only the upper triangular matrix of $m[1 : n, 1 : n]$ is used.

Matrix-Chain Multiplication, Bottom-Up Approach

- The corresponding bottom-up dynamic programming algorithm is as following.

Algorithm 6.1.8. Bottom-up matrix-chain multiplication.

```
Algorithm MCM_BU(i, j, n, p, m, s)
// To find the minimum scalar multiplication for a matrix chain.
{
    for i := 1 to n do m[i, i] := 0;
    for l := 2 to n do { // l is the chain length.
        for i := 1 to n − l + 1 do { // all possible i
            j := i + l − 1; // j − i = l − 1.
            u := ∞;
            for k := i to j − 1 do { // all possible groupings.
                v := m[i, k] + m[k + 1, j] + p[i − 1]· p[k]· p[j];
                if (v < u) {
                    u := v; s[i, j] := k; // record for solution
                }
            }
        }
    }
}
```

Matrix-Chain Multiplication, Print Solution

- In this bottom-up dynamic programming algorithm, again, the solution is recorded in the $s[1:n, 1:n]$ matrix.
- To print out the multiplication sequence after calling MCM_BU algorithm, the following algorithm should be called to print out the solution.

Algorithm 6.1.9. Matrix-chain multiplication print solution.

```
Algorithm MCM_PS(i, j, s)
// To print the matrix multiplication sequence.
{
    if (i = j) write ("A" i);
    else {
        write ("(") ;
        MCM_PS(i, s[i, j], s); // (A_i ··· A_k)
        MCM_PS(s[i, j] + 1, j, s); // (A_{k+1} ··· A_j)
        write (")") ;
    }
}
```


Matrix-Chain Multiplication, Example

- A chain of 6 matrices and their dimensions are shown below.

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

- The optimal solution is

$$(A_1(A_2A_3))((A_4A_5)A_6)$$

  with 15125 scalar multiplications.
- The m and s tables are also shown below.

m table

0	15750	7875	9375	11875	15125
	0	2625	4375	7125	10500
		0	750	2500	5375
			0	1000	3500
				0	5000
					0

s table

-	1	1	3	3	3
	-	2	3	3	3
		-	3	3	3
			-	4	5
				-	5
					-

Matrix-Chain Multiplication, Complexities

- The bottom-up matrix-chain multiplication algorithm (6.1.8) has three nested loops, each executed at most n times.
  - Total time complexity is $\mathcal{O}(n^3)$.
  - The space complexity is $\Theta(n^2)$ due to m and s tables.
- The top-down algorithm (6.1.7) has essentially the same complexities.
  - Time complexity: $\mathcal{O}(n^3)$
  - Space complexity: $\Theta(n^2)$
- Note that the m and s tables need only the upper triangular matrix only, but the space complexity is still $\Theta(n^2)$.
- For the recursive algorithm (6.1.6), however, the time complexity is $\mathcal{O}(2^n)$. It's space complexity is $\mathcal{O}(n)$.

Dynamic Programming

- For the rod-cutting problem, the solution is found by solving Eq. (6.1.1), which is repeated below.

$$r_n = \max\{p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\}.$$

  Time complexity is $\mathcal{O}(n^2)$.
- For the matrix-chain multiplication problem, the solution is found by solving Eq. (6.1.6).

$$m(i,j) = \min_{i \le k \le j} \{m(i,k) + m(k+1,j)\} + p_{i-1} \cdot p_k \cdot p_j.$$

  This requires $\mathcal{O}(n^3)$ time complexity.
- To apply dynamic programming method, the problem can be formulated to the overall optimal solution is constructed using the optimal solutions of its subproblems.
  - The problem should be divided into subproblems.
  - The optimal solutions for the subproblems need to be found.
  - Overall optimal solution is then constructed from those solutions.
- Recursive algorithm can usually developed from the equation.
  - Using table to record solutions of subproblems improves the efficiency greatly.
  - Bottom-up approach, without recursion, usually improve the efficiency further.

Longest Common Subsequence Problem

- Practical problem: Given two strands of DNA, such as

$$S_1 = \mathtt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$
$$S_2 = \mathtt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

find the longest strand S_3 such that S_3 is a subsequence of both S_1 and S_2.

Definition 6.1.10. Subsequence

Given a sequence $X = \langle x_1, x_2, \cdots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \cdots, z_k \rangle$ is a subsequence of X if there is a strictly increasing sequence $\langle i_1, i_2, \cdots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, $x_{i_j} = z_j$.

- Example: Given $X = \langle A, B, C, B, D, A, B \rangle$, $Z = \langle B, C, D, B \rangle$ is a subsequence of X.

Definition 6.1.11. Common subsequence

Given two sequences X and Y, sequence Z is a common subsequence of X abd Y if Z is a subsequence of both X and Y.

- Example: Given $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, then $Z = \langle B, C, B, A \rangle$ is a common subsequence of X and Y.

Longest Common Subsequence – Properties

- Given a sequence $X_m = \langle x_1, x_2, \ldots, x_m \rangle$, then there are 2^m subsequence for X_m.
- Brute-force approach to find a longest common subsequence (LCS) would be impractical for reasonable size sequences.

Theorem 6.1.12.

Given two sequences, $X_m = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y_n = \langle y_1, y_2, \ldots, y_n \rangle$, if $Z_k = \langle z_1, z_2, \ldots, z_k \rangle$ is any LCS of X and Y, then

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $x_m \neq z_k$ implies Z is an LCS of X_{m-1} and Y_n.
3. If $x_m \neq y_n$, then $y_n \neq z_k$ implies Z is an LCS of X_m and Y_{n-1}.

- Proof please see textbook [Cormen], p. 392.

Longest Common Subsequence – Properties, II

- Let $c[i,j]$ be the length of an LCS of the sequences X_i and Y_j, then we have

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max\{c[i,j-1], c[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{6.1.7}$$

- Based on this equation, recursive algorithm can be derived to solve the LCS problem.
  - However, due to exponential number of subsequences the recursive algorithm is very inefficient to solve reasonable size problems.
- A bottom-up dynamic programming algorithm is shown next which is rather efficient.
  - Inputs are two sequences: $X_m = \langle x_1, x_2, \ldots, x_m \rangle$, $Y_n = \langle y_1, y_2, \ldots, y_n \rangle$.
  - Two tables are built by the algorithm.

    $c[0:m, 0:n]$: record the length of the LCS for X_i and Y_j at $c[i,j]$.

    $b[1:m, 1:n]$: record the solution sequence of the LCS for X_i and Y_j at $b[i,j]$.

Longest Common Subsequence – Algorithm

Algorithm 6.1.13. Longest Common Subsequence

```
Algorithm LCS(X, Y)
// To find a LCS of X = ⟨x1, ..., xm⟩ and Y = ⟨y1, ..., yn⟩.
{
    for i := 1 to m do c[i, 0] := 0;
    for j := 0 to n do c[0, j] := 0;
    for i := 1 to m do {
        for j := 1 to n do {
            if (xi = yj) then {
                c[i, j] := c[i − 1, j − 1] + 1;
                b[i, j] := " ↖ ";
            }
            else if (c[i − 1, j] ≥ c[i, j − 1]) then {
                c[i, j] := c[i − 1, j];
                b[i, j] := " ↑ ";
            }
            else {
                c[i, j] := c[i, j − 1];
                b[i, j] := " ← ";
            }
        }
    }
}
```

Longest Common Subsequence – Print Solution

- After the LCS(X, Y) algorithm is called, tables $b[1:m, 1:n]$ and $c[0:m, 0:n]$ are built.
- The length of the LCS is in $c[m, n]$.
- And the following recursive algorithm can print out the LCS using X and table $b[1:m, 1:n]$.
- It should be invoked by LCS_PS(b, X, m, n).

Algorithm 6.1.14. Print Longest Common Subsequence

```
Algorithm LCS_PS(b, X, i, j)
// Use X_m and b[1 : m, 1 : n] to print the LCS found recursively.
{
      if (i = 0 or j = 0) return ;
      if (b[i, j] = " ↖ " ) then {
            LCS_PS(b, X, i − 1, j − 1) ;
            write ( " x_i " ) ;
      }
      else if (b[i, j] = " ↑ " ) then LCS_PS(b, X, i − 1, j) ;
      else LCS_PS(b, X, i, j − 1) ;
}
```


Longest Common Subsequence – Example

- Given two sequences

$$X_7 = \langle A, B, C, B, D, A, B \rangle, \; Y_6 = \langle B, D, C, A, B, A \rangle.$$

After LCS(X, Y) call, we have the following tables.

Table $c[0:7, 0:6]$

	j	0	1	2	3	4	5	6
i		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

Table $b[1:7, 1:6]$

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

- The length of the LCS found is $c[7, 6] = 4$.
- And the LCS is $\langle B, C, B, A \rangle$.

Longest Common Subsequence – Complexity

- The bottom-up dynamic algorithm to solve LCS problem, Algorithm (6.1.13), is dominated by the double loops, lines 6-7.
- Thus, the time complexity is $\Theta(mn)$.
- The LCS solution printing algorithm (6.1.14) traces the $b[1:m, 1:n]$ table for the lower-right corner to the upper-left corner.
  - Thus, the time complexity is $\mathcal{O}(m+n)$.
- The overall space complexity is $\Theta(mn)$ due to those two tables, $c[0:m, 0:n]$ and $b[1:m, 1:n]$.
- It is possible to print out the LCS solution using table $c[0:m, 0:n]$ alone, thus save memory space requirement.
  - Starting from $c[m][n]$, each step it requires to compare x_m vs. y_n and $c[m-1][n]$ vs. $c[m][n-1]$.
- Note that in Algorithm (6.1.13), in constructing $c[i]$ row it needs only the previous row $c[i-1]$.
  - Thus, if only the length of LCS is required, table $b[1:m, 1:n]$ needs not be built. The space complexity can be reduced to $\mathcal{O}(m)$.

Summary

- Rod-cutting problem
- Matrix-chain multiplication problem
- Dynamic programming
- Longest common subsequence problem