ECM5605(5075) S'20 Algorithms

Lecture 05

Dynamic Programming (2): Matrix-Chain Multiplication and Shortest Path Problem

DP vs. Divide-and-Conquer

- Common: solve problems by combining the solutions to subproblems ⇨ "*If you cannot solve a problem, then there is an easier problem you can solve it: find it.*" by George Polya (1987-1985)
- Divide-and-conquer algorithms
  - partition a problem into ***independent*** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem
  - what's the potential problem??
    ⇨ *inefficient to solve overlapping subproblems*
- Dynamic programming (DP)
  - applicable if ***many*** subproblems are ***repeated***
  ⇨ DP solves each overlapping subproblem ***just once***

Key Questions to DP

- What kind of problems cannot be solved by dynamic programming effectively?
  - ⇨ if it does not follow ***optimal substrcture***
  - ⇨ the optimal solution to a problem always contains optimal solutions to all subproblems
  - ⇨ Not "combining optimal solutions to subproblems can guarantee a optimal solution!" (What's different?)
- What kind of problems cannot be solved by dynamic programming efficiently?
  - – if too few ***overlapping*** problems
  - ⇨ a recursive solution only needs to compute a "small" number of distinct subproblems repeated many times

DP(1): Longest Common Subsequence

- What is the problem formulation?
  - Given 2 sequences, $\boldsymbol{X} = \langle x_1,\ldots,x_m \rangle$ and $\boldsymbol{Y} = \langle y_1,\ldots,y_n \rangle$, find a common *in-order* subsequence whose length is maximum
- How to apply DP to the LCS problem?
  1. Define the ***optimal (sub)structure*** and ***recurrence relationship***) and then compute the length of a longest-common subsequence
  2. Backtrack the algorithm to find one solution
- Complexity of the LCS problem
  - Space: O***(mn)*** to construct the length table
  - Time: Computing the table: O***(mn)***
    Constructing an LCS: O***(m+n)***

Other Similar Sequence Problems

- ***Edit distance:***
  - Given 2 sequences, $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$, what is the minimum number of deletions, insertions, and substitutions that you must do to change one to another?
- ***Protein sequence alignment:***
  - Given a score matrix on amino acid pairs, $s(a,b)$ for $a,b \in \{\Lambda\} \cup A$, and 2 amino acid sequences, $X = \langle x_1,\ldots,x_m \rangle \in A^m$ and $Y = \langle y_1,\ldots,y_n \rangle \in A^n$, find the alignment with lowest score…
- Study the textbook by yourself

Edit Distance & Example

- Suppose we have two strings x,y
  - e.g. x = kitten and y = sitting
- And we want to transform x into y.
- We use *edit operations*:
  - insertions
  - deletions
  - substitutions

k i t t e n

s i t t i n g

- A close look
  - 1st step: kitten ⇨ sitten (substitution)
  - 2nd step: sitten ⇨ sittin (substitution)
  - 3rd step: sittin ⇨ sitting (insertion)

Development of a DP (revisited)

- Four steps in developing a DP algorithm
  1. Characterize the ***optimal*** structure
  2. ***Recursively*** define the formula of an optimal solution
  3. Compute the value of an optimal solution either ***bottom-up*** in a ***table*** or *top-down* with memorization (*caching*)
  4. Construct an optimal solution from computed information (this step can be omitted if only the value is required)

Developing DP for Edit Distance

- Given strings $X = \langle x_1,\ldots,x_m \rangle$ and $Y = \langle y_1,\ldots,y_n \rangle$
- Define $d(i,j) = |ED(X[1..i\,], Y[1..j\,])|$ ⇨ Find $d(m,n) = |ED(X,Y)|$
- Then
  1. If $x_i = y_j$, then no cost on changing x_i to y_j ⇨ $d(i,j) = d(i\text{-}1,j\text{-}1)$.
  2. If $x_i \neq y_j$, then $d(i,j)$ from the minimum of
     - insert x_i ⇨ $d(i,j) = d(i\text{-}1,j) + 1$
     - delete y_j ⇨ $d(i,j) = d(i,j\text{-}1) + 1$
     - substitute x_i to y_j ⇨ $d(i,j) = d(i\text{-}1,j\text{-}1) + 1$
- How about $d(i,j)$ as i=0 or j=0 ???

Modify *LCS_v2.cpp*

Binomial Coefficient

- A binomial coefficient is indexed by a pair of integers $n \geq k \geq 0$ and is written

$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{else} \end{cases}$$

- Write a recursive in C/C++ to implement binomial coefficient (NaiveCnk.cpp)
- Let's write this recursive again with memorization (MemCnk.cpp)
- Observe their efficiency

⇨ Now we shall understand *top-down caching*

DP(2): Matrix-Chain Multiplication

- Problem Formulation:
  - Given a chain $<A_1, A_2, \ldots, A_n>$ of n matrices, matrix A_i has dimension $p_{i-1} \times p_i$, parenthesize the product $\boldsymbol{A} = A_1 A_2 \ldots A_n$ to minimize the number of scalar multiplications.
- Example: 3 matrices: A_1:4×2, A_2:2×5, A_3:5×1
  - $(A_1A_2)A_3$: total multiplications = 4×2×5+4×5×1 = 60.
  - $A_1(A_2A_3)$: total multiplications = 2×5×1+4×2×1 = 18.
- So the order of parenthesis for multiplications can make a big difference
  - But how to find the best order??
  - # of combinations of different parenthesizations??

Number of Different Parenthesizations

- **Catalan Number:** For any n,
  # ways to fully parenthesize the product of a chain of n+1 matrices
  = # of binary trees with n nodes.
  = # of permutations generated from 1 2 … n through a stack.
  = # of n pairs of fully matched parentheses.
  = n-th Catalan Number
  = $C(2n,n)/(n+1) = \Omega(4^n/n^{3/2})$
- The Brute-Force algorithm is hopeless!
  – What else can we try?
  ⇨ *Dynamic Programming*

Apply DP to Matrix-Chain Multiplication

- Step 1: **Characterize the optimal structure**

  Let $m[i, j]$ be the minimum number of multiplications to compute matrix $A_{i..j} = A_i A_{i+1} \dots A_j$, $1 \le i \le j \le n$

  – $A_1, A_2, \dots, A_n$ with sizes in $p_0 \times p_1, p_1 \times p_2, \dots, p_{n-1} \times p_n$

  – $m[1, n]$: the cheapest cost to compute $A_{1..n}$

  Suppose the optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}, then the parenthesizations of $A_{i..k}$ and $A_{k+1..j}$ must be optimal

  – Why? How to prove?? (by contradiction!)

Apply DP to Matrix-Chain Multiplication (cont'd)

- Step 2: **Define the recurrence relationship**

$$m[i,j] = \begin{cases} 0 & , i = j \\ \min_{i \le k < j} m[i,k]+m[k+1,j]+p_{i-1}p_k p_j & , i < j \end{cases}$$

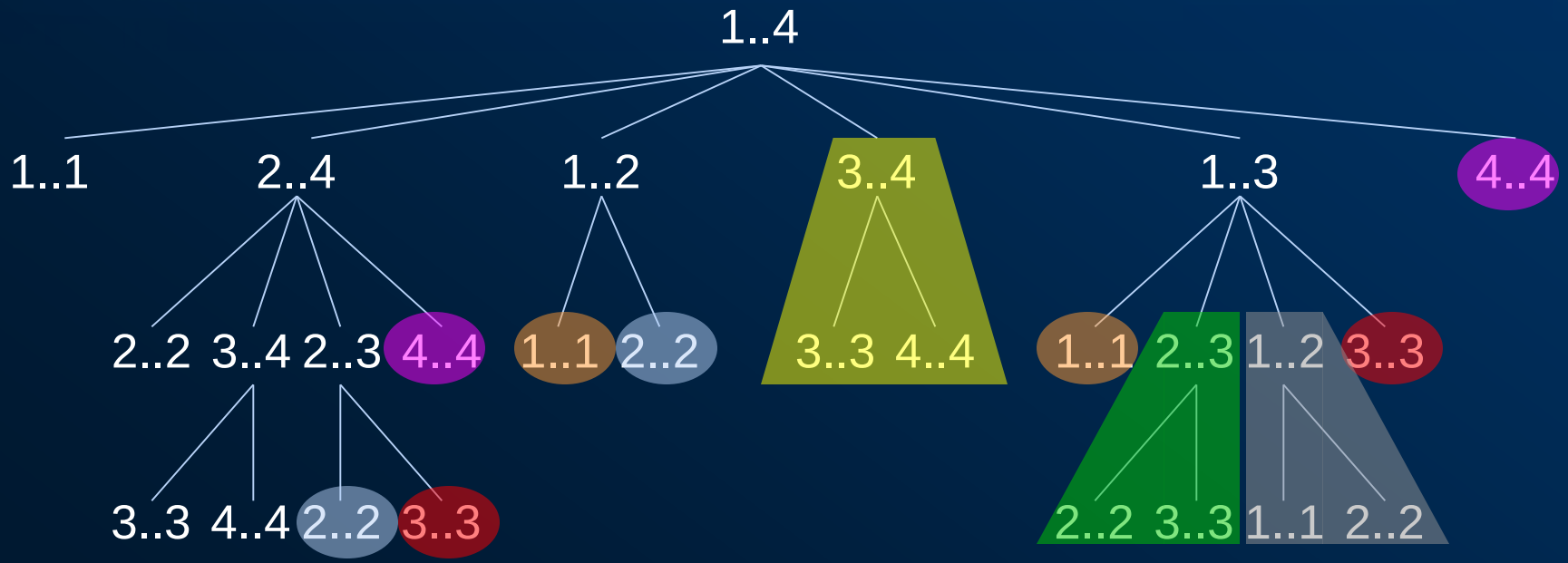
- Step 3: **Compute costs tabularly (top-down)**
  - input: dimension sequence $\boldsymbol{p} = \langle p_0, p_1, \ldots, p_n \rangle$

```
Naive_Matrix_Chain(p,i,j)
1   if i = j then return 0;
2   m[i,j] ← ∞; //or INT_MAX
3   for k ← i to (j-1)
4       q ← N_M_C(p,i,k)+N_M_C(p,k+1,j)+p_{i-1}p_k p_j;
5       if q < m[i,j] then m[i,j] ← q
6   return m[i,j];
```

Example of Matrix-Chain-Order

```
Naive_Matrix_Chain(p,i,j)
1   if i = j then return 0;
2   m[i,j] ← ∞; //or INT_MAX
3   for k ← i to (j-1)
4       q ← NMC(p,i,k)+NMC(p,k+1,j)+p_{i-1}p_k p_j;
5       if q < m[i,j] then m[i,j] ← q
6   return m[i,j];
```

Problem: repeat on computing the same subproblems !!



Example of Matrix-Chain-Order (cont'd)

- Consider an example with sequence of dimensions <5,2,3,4,6,7,8>

$m[i,j] = \min_{i \le k < j} \{ m[i,k]+m[k+1,j]+p_{i-1}p_k p_j \}$

	1	2	3	4	5	6
1	0	30	64	132	226	348
2		0	24	72	156	268
3			0	72	198	366
4				0	168	392
5					0	336
6						0

Ex: m[1,1]+m[2,6]+5*2*8=0+268+80=348,

m[1,2]+m[3,6]+5*3*8=30+366+120=516, …

Memorization for Top-down DP

- Simple idea: maintain a table ***M*** to memorize solutions to the computed subproblems

```
Memorized_Matrix_Chain(p)
1   n ← length[p]-1;
2   for  i ← 1 to n
3       for  j ← 1 to n
4           m[i,j] ← ∞;
5   return Lookup_Chain(p,1,n);
```

```
Lookup_Chain(p, i, j)
1   if m[i,j] < ∞
2       return m[i, j] ;
3   if i = j then m[i,j] ← 0;
4   else
5       for k ← i to (j -1)
6           q ← Lookup_Chain(p,i,k)+Lookup_Chain(p,k+1,j)+p_{i-1}p_k p_j;
7           if q < m[i,j] then m[i,j] ← q;
8   return m[i,j];
```

Apply DP to Matrix-Chain Multiplication (cont'd)

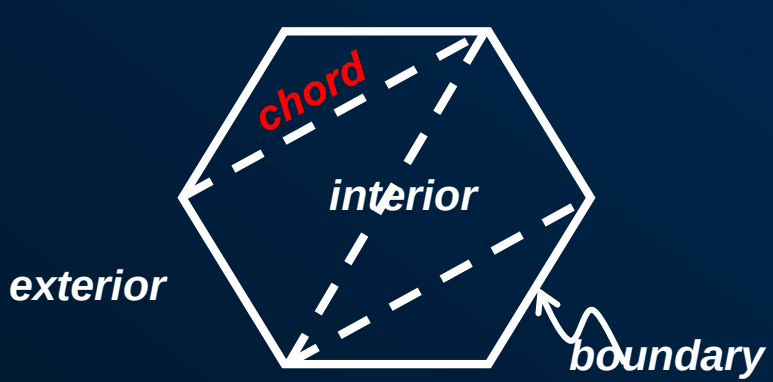
- Step 4: **Construct an Optimal Solution**
  - Optimal matrix $A_{1..n}$ multiplication: $A_{1..s[1,n]} \times A_{s[1,n]+1..n}$.
  - Backtrack the Lookup-Chain table ***M*** to find the current splitting ***k***

- Complexity analysis
  - Space: $O(n^2)$ for the memorization table ***M***
  - Time: total runtime $O(n^3)$. Why??

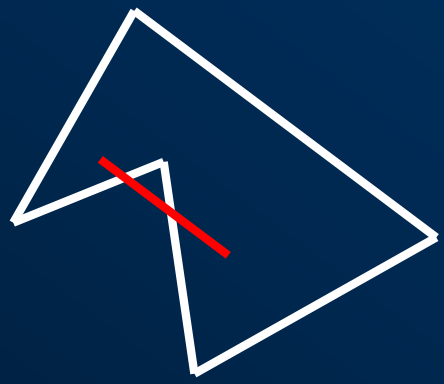
    ∵ each entry in ***M*** takes $O(n)$ time

DP(3): Application on Polygon

- A ***polygon*** is a piecewise-linear, closed curve in the plane
- Def: A polygon P is ***convex*** if, when you connect any two points p, q in the polygon (interior + boundary), the entire line segment $\overline{pq}$ lies in the polygon.



Convex



Not convex

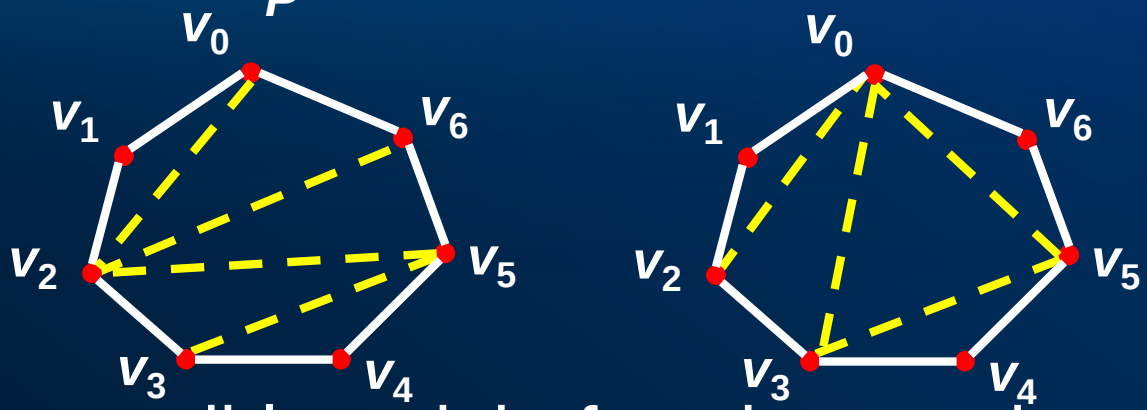
Optimal Polygon Triangulation

- A ***triangulation*** of a polygon $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ is a set of ***non-intersecting diagonals*** that partitions the polygon into triangles.
  - an edge $v_i v_j$ is called a ***diagonal*** if v_i and v_j are not adjacent vertices
- **Optimal Polygon Triangulation Problem:**

  Given a convex polygon $\boldsymbol{P} = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ and a weight function $\boldsymbol{w}$ defined on triangles, find a triangulation that minimizes $\sum_{\forall \Delta}\{\boldsymbol{w}(\Delta)\}$.

Optimal Polygon Triangulation Problem

- Input: a convex polygon $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$
- Output: an optimal triangulation with minimum w_P



- One possible weight function on triangle:

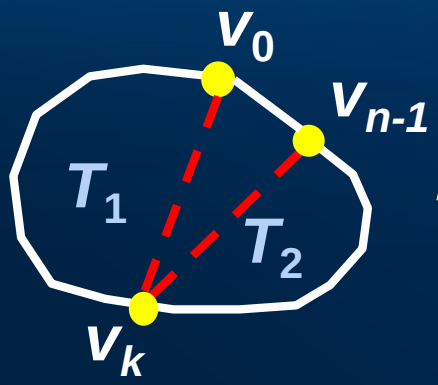
$$w(\Delta v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

where $|v_i v_j|$ is the Euclidean distance

  - total weight = boundary+2×(length of all diagonals)

Apply DP to Optimal Polygon Triangulation

- Matrix-chain multiplication is a special case of the optimal polygonal triangulation problem.
- Complexity: Runs in O(n^3) time and uses O(n^2) space.
- Hint:



$$T = T_1 \cup T_2 \cup \{w(v_0 v_k v_{n-1})\}$$

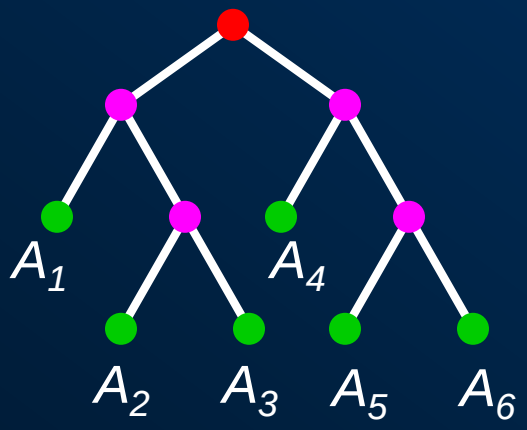
- Step 1:
  - Suppose the optimal polygon triangulation of $P_{i..j}$ connect at v_k, then the polygon triangulation of $P_{i..k}$ and $P_{k+1..j}$ must be optimal
- Step 2:

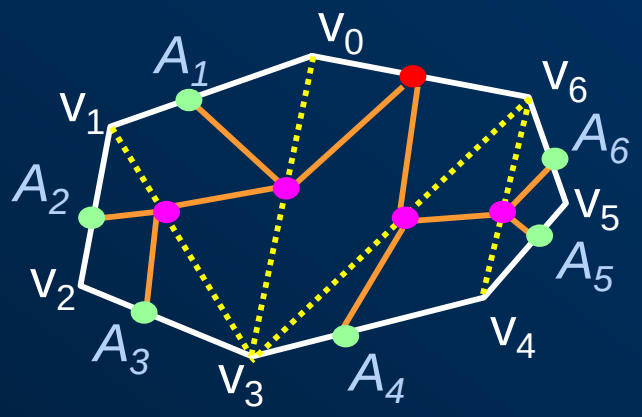
$$t[i,j] = \begin{cases} 0 & , \text{if } i = j \\ \min\limits_{i \le k \le j-1} \{t[i,k] + t[k+1,j] + w(\Delta v_{i-1} v_k v_j)\} & , \text{if } i < j \end{cases}$$

Correspondence between DP Problems

- Between ***full parenthesization***, **full binary tree** (parse tree), and ***triangulation***
- full parenthesization: multiplication pairs
  ⇔ full binary tree: non-leaf nodes
  ⇔ triangulation: mid-points on diagonals

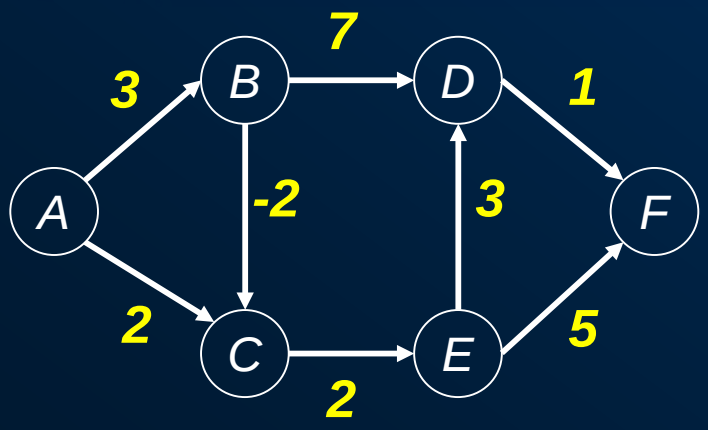
$((A_1(A_2A_3)(A_4(A_5A_6)))$



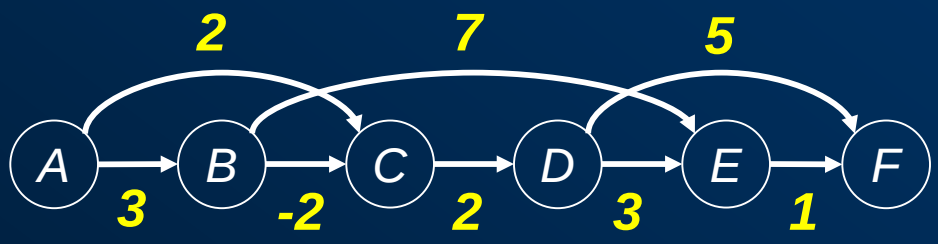


DP(4): Shortest Path Problem

- **(Single Source) Shortest Path**
  - Input: given a weighted graph ***G***=(***V***,***E***), a source $s \in V$, and a sink $t \in V$,
  - Output: a path of minimum weight between s and t.



DAG



linearized DAG

How to check acyclic??

Apply DP to Shortest Path Problem

- Step 1: **Characterize the optimal substructure**

  Any part of the shortest path is also shortest.
- Step 2: **Define the recurrence relationship**

$$d(v) = \min_{(u,v)\in E} (d(u) + w(u,v))$$

  – where $d(v)$ is the shortest distance from s to v and $d(s)=0$
- **Complexity analysis:**
  – Runs in O(|*E*|) time and uses O(|*V*|) space.
  – Why?

3 Types of Overlapping Problems

① input is $x_1,x_2, \ldots,x_n$ *and a subproblem is* $x_1,x_2,\ldots,x_i$

$\Rightarrow$ # of subproblems is **O(*n*)** .

EX: (single source) shortest path problem on DAGs

② input is $x_1,x_2, \ldots,x_n$ *and* $y_1,y_2, \ldots,y_m$, *a subproblem is* $x_1,x_2,\ldots,x_i$ *and* $y_1,y_2,\ldots,y_j$

$\Rightarrow$ # of subproblems is **O(*nm*)** .

EX: longest common subsequence, edit distance, protein sequence alignment

③ input is $x_1,x_2, \ldots,x_n$ *and a subproblem is* $x_i,x_{i+1},\ldots,x_j$

$\Rightarrow$ # of subproblems is **O(n^2)** .

EX: matrix-chain multiplication, convex polygon triangulation

Summary

- How to apply DP to your problem?
  ① Characterize the optimal structure
  ② Define the recurrence formula (relationship)
  ③ Compute costs tabularly (bottom-up/top-down)
  ④ Construct an Optimal Solution
- Matrix-Chain Multiplication in top-down
  – think about bottom up
  – Time complexity? Space complexity?
  – Optimal Polygon Triangulation and its correspondence w/ Matrix-Chain Multiplication
- 3 Types of common overlapping problems
- Next Lecture ⇨ Knapsack problem and Greedy Algorithms