Before we start: Check out HW3 #5 solution @ course drive/Homework/Solutions

EECS 376 Discussion 4

Sec 27: Th 5:30-6:30 DOW 1017

IA: Eric Khiu

Slide deck available at course drive/Discussion/Slides/Eric Khiu

Agenda

- DP Recap
- 0-1 Knapsack
- Worksheet Problems

DP Recap

Lecture Notes

DP Recipe

- Write recurrence
  - Choose the subject of recurrence
  - Base case(s)
  - Form <u>optimal</u> sub-<u>solution</u> (“up to this point”)
- Size of table (Dimensions? Range of each dimensions?)
- To fill in cell, which other cells do I look at?
- Which cell(s) contain the final answer?
- Reconstructing solution: Follow arrows from final answer to base case

Reconstructing Solution

5. **A pebble game.**

Many two-player strategic games, like the several variants of Nim, can be modeled as follows. There is a directed acyclic graph $G = (V, E)$ presented in topological order: the vertices are labeled as $V = \{1, \ldots, n\}$, and every edge $(u, v) \in E$ has $u < v$. Moreover, every vertex $i < n$ has at least one outgoing edge.

Two players, called maize and blue, take turns in the following game on this graph. They start with a pebble at vertex 1, and maize plays first. On each turn, the acting player must move the pebble from the current vertex u to a new vertex v along some edge $(u, v) \in E$ of the graph, of the acting player's choice. If a player moves the pebble to vertex n (the final one), then that player wins the game.

Give a dynamic-programming algorithm that, given the graph G as input, determines which player can be guaranteed a win by playing perfectly, no matter how the opponent plays.

- Recurrence Relation:

$$W(i) = \begin{cases} \text{false} & \text{if } i = n, \\ \bigvee_{j:(i,j)\in E} \neg W(j) & \text{otherwise.} \end{cases}$$

Reconstructing Solution

- Recurrence Relation:

$$W(i) = \begin{cases} \text{false} & \text{if } i = n, \\ \bigvee_{j:(i,j)\in E} \neg W(j) & \text{otherwise.} \end{cases}$$

- Bottom-up solution:

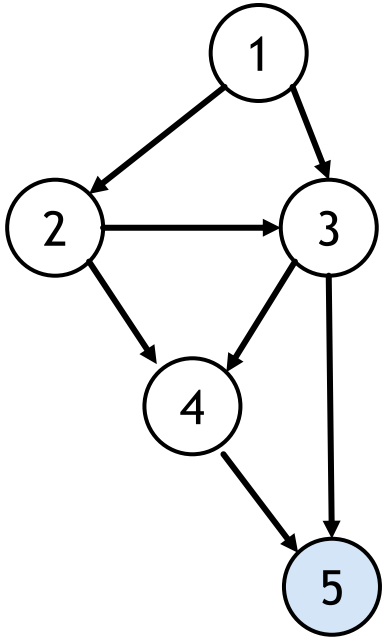
```
allocate table W[1...n] of booleans
W[n] = false
for i = n - 1 down to 1 do
    W[i] ← false
    for all edges (i, j) ∈ E (walk the adjacency list of i) do
        if W[j] = false then W[i] ← true
```

- Suppose the pebble is currently at position 1 and it's your turn
- Task: Reconstructing solution (winning strategy), return NULL if none exist
- DP recipe says "Follow arrows from final answer to base case"
  - Where is the final answer?
  - Where is the base case?

Reconstructing Solution: Demo

```
allocate table W[1...n] of booleans
W[n] = false
for i = n − 1 down to 1 do
    W[i] ← false
    for all edges (i, j) ∈ E (walk the adjacency list of i) do
        if W[j] = false then W[i] ← true
```

i	1	2	3	4	5
W[i]					

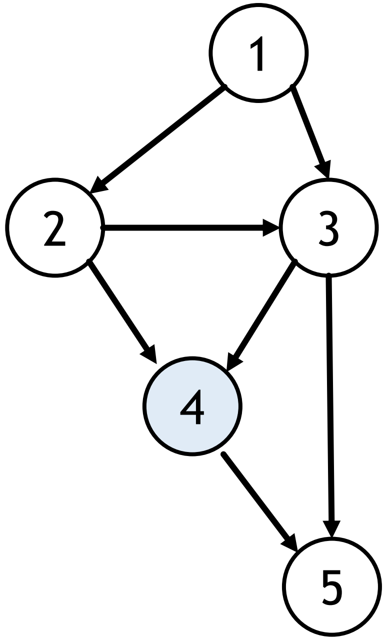


For node 5

- Base case: W[5] = F

Reconstructing Solution: Demo

```
allocate table W[1...n] of booleans
W[n] = false
for i = n − 1 down to 1 do
    W[i] ← false
    for all edges (i, j) ∈ E (walk the adjacency list of i) do
        if W[j] = false then W[i] ← true
```



i	1	2	3	4	5
W[i]					F

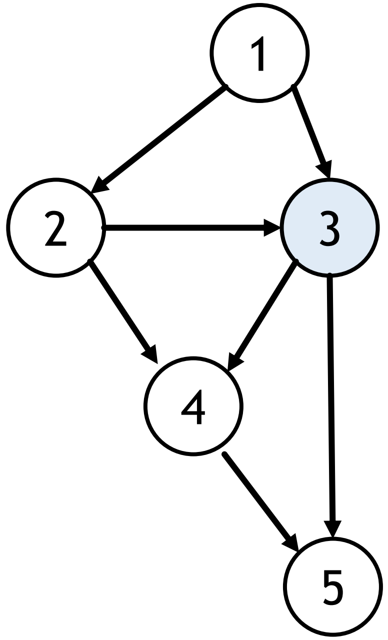
For node 4

- W[5] = F → W[4] = T

Reconstructing Solution: Demo

```
allocate table W[1...n] of booleans
W[n] = false
for i = n − 1 down to 1 do
    W[i] ← false
    for all edges (i, j) ∈ E (walk the adjacency list of i) do
        if W[j] = false then W[i] ← true
```

i	1	2	3	4	5
W[i]				T	F



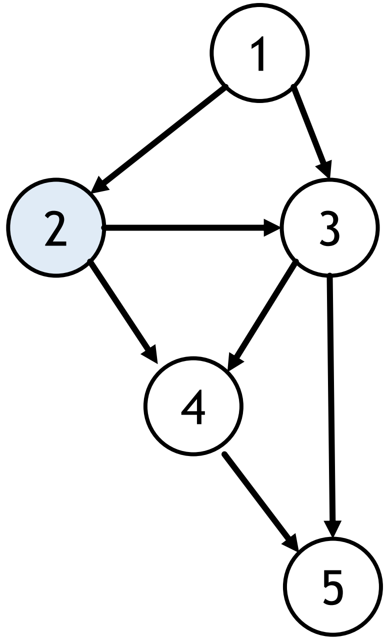
For node 3

- W[4] = T
- W[5] = F → W[3] = T

Reconstructing Solution: Demo

allocate table $W[1 \ldots n]$ of booleans
$W[n] = \text{false}$
for $i = n - 1$ down to 1 **do**
    $W[i] \leftarrow \text{false}$
    for all edges $(i, j) \in E$ (walk the adjacency list of i) **do**
        if $W[j] = \text{false}$ **then** $W[i] \leftarrow \text{true}$

i	1	2	3	4	5
W[i]			T	T	F

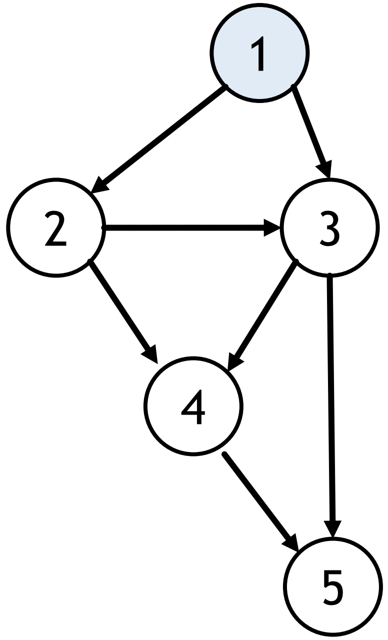


For node 2
- W[3] = T
- W[4] = T
- No adjacent False node → W[2] = F

Reconstructing Solution: Demo

allocate table $W[1 \dots n]$ of booleans
$W[n]$ = false
for $i = n - 1$ down to 1 **do**
  $W[i] \leftarrow$ false
  for all edges $(i, j) \in E$ (walk the adjacency list of i) **do**
    if $W[j]$ = false **then** $W[i] \leftarrow$ true

i	1	2	3	4	5
W[i]		F	T	T	F

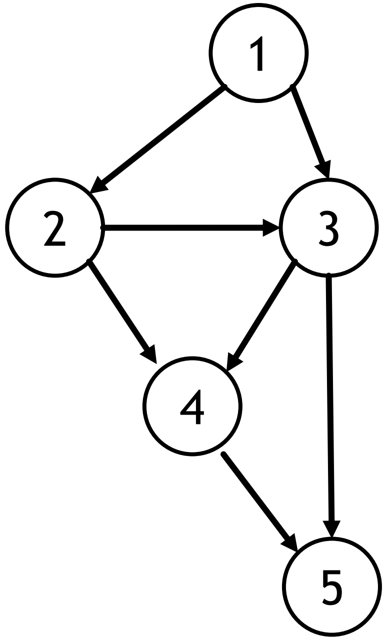


For node 1
- W[3] = T
- W[2] = F → W[1] = T

Reconstructing Solution: Demo

allocate table $W[1 \ldots n]$ of booleans
$W[n]$ = false
for $i = n - 1$ down to 1 **do**
    $W[i] \leftarrow$ false
    for all edges $(i, j) \in E$ (walk the adjacency list of i) **do**
        if $W[j]$ = false **then** $W[i] \leftarrow$ true

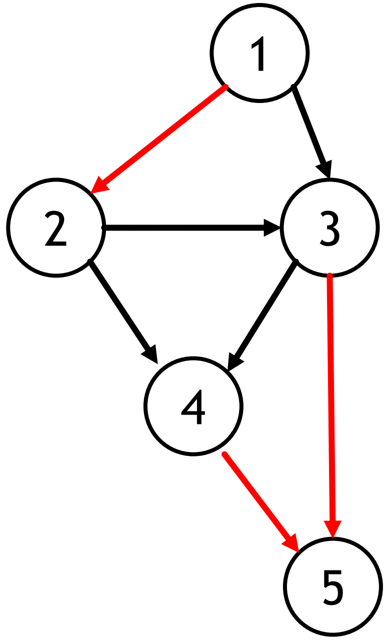
i	1	2	3	4	5
W[i]	T	F	T	T	F



- Suppose the pebble is at node 1 during you turn, where should you move it to?
- Key observation: Whenever W[i] = True, there must be some adjacent node j such that W[j] = False
    - Use that to draw the arrow!

Reconstructing Solution: Demo

allocate table $W[1 \dots n]$ of booleans
$W[n]$ = false
for $i = n - 1$ down to 1 **do**
  $W[i] \leftarrow$ false
  for all edges $(i, j) \in E$ (walk the adjacency list of i) **do**
    if $W[j]$ = false **then** $W[i] \leftarrow$ true



i	1	2	3	4	5
W[i]	T	F	T	T	F
W[j] that yield W[i] = T, if any	2	NULL	5	5	NULL

- Suppose the pebble is at node 1 during you turn, where should you move it to?
- Key observation: Whenever W[i] = True , there must be some adjacent node j such that W[j] = False
  - Use that to draw the arrow!
  - Here, the arrow tells us the next step when the pebble is at node i (with winning strategy)

Reconstructing Solution: Takeaway

- Identify the cell containing final solution, c_s
- Identify the cell containing the base case
- Track all the cells used to fill c_s, cells used to fill those cells, …, all the way to the base case

- **Remark:** In OOP, it might be easier to represent each cell as an object and store the “where I come from” info

0-1 Knapsack

0-1 Knapsack Set-Up

- You have a set of n items, each with weight w_i and value v_i, and you have a knapsack with maximum weight capacity C

- Inputs:
  - n-length array of positive integer weights $W = (w_1, w_2, \ldots, w_n)$
  - n-length array of positive integer values $V = (v_1, v_2, \ldots, v_n)$
  - Capacity of the knapsack $C \in \mathbb{N}$

- Goal: pick a subset of items $S \subseteq \{1,2, \ldots n\}$ that maximizes the value of the knapsack $\sum_{i \in S} v_i$, while staying within the capacity $\sum_{i \in S} w_i \leq C$

0-1 Knapsack Recurrence

- **Step 1: Subject of recurrence**
  - Let $K(i,j)$ be an optimal knapsack solution using only items up to index i, and having capacity only up to j
- **Step 2: Base cases**
  - $i = 0$ (No item) or $j = 0$ (no space): $K(i,j) = 0$
  - $w_i > j$ (Not enough space to consider item i): $K(i,j) = K(i-1,j)$
- **Step 3: Optimal sub-solution**
  - [Sub-solution] For items 1, …, i, how do we reduce the problem?
    - Deal with [Items 1, …, i-1]and [Item i] separately
    - Q: What would happen if we take item i in the knapsack?
      - Capacity reduces by w_i, total value increases by v_i
      - Else: Both capacity and total value remain unchanged
  - [Optimal] *Choose* between whether to include the ith item
    - Maximization problem: Use max
    - Objective function? $K(i,C) = \max\{K(i-1,C-w_i)+v_i, K(i-1,C)\}$

DP Implementations

- Top-down recursion
- Top-down memorization
- Bottom-up (iterative)

- For this class we usually expect the algorithms to be implemented bottom-up

Top Down Recursive Approach

- Implement the recursion as in the recurrence
- **Runtime:** $O(2^n)$
- **Space:** $O(n)$

Input: Integers n, C, arrays W, V. Again, note the 1-based indexing.
Output: The maximum total value of objects the Knapsack can hold

```
function KNAPSACK(n, C, W, V)
    if n = 0 or C = 0 then
        return 0
    if W[n] > C then
        return Knapsack(n − 1, C, W, V)
    return max(Knapsack(n − 1, C − W[n], W, V) + V[n], Knapsack(n − 1, C, W, V))
```

- Advantages
  - Easy to translate from recurrence relation
  - No additional data structures necessary
- Disadvantages
  - A lot of recursive calls- may not be time-efficient
  - Correctness proof is usually harder
    - Not as smooth as bottom up- imagine proving by induction $P(k) \Longrightarrow P(k+1)$, but we can't do that with recursive top-down

Top Down Memoization Approach

- Same as before, but include a memo recording results of previous recursive calls
- **Runtime:** $O(n \cdot C)$
- **Space:** $O(n \cdot C)$

Input: Integers n, C, arrays W, V, and lookup-table DP with values initialized to -1. Again, note the 1-based indexing.
Output: The maximum total value of objects the Knapsack can hold

```
function KNAPSACK(n, C, W, V, DP)
    if n = 0 or C = 0 then
        return 0
    if DP[n − 1][C] = −1 then
        DP[n − 1][C] ←Knapsack(n − 1, C, W, V, DP)
    if W[n] > C then
        return DP[n − 1][C]
    if DP[n − 1][C − W[n]] = −1 then
        DP[n − 1][C − W[n]] ←Knapsack(n − 1, C − W[n], W, V, DP)
    return max(DP[n − 1][C − W[n]] + V[n], DP[n − 1][C])
```

- Advantages:
  - Usually much better time-complexity than top-down recursion
  - While harder than recursion, the logic is often more intuitive than bottom-up
- Disadvantage:
  - May still be slower than bottom up

Botton Up Iterative Approach

- Build the table without recursion
- Iterate over previous results to fill the cells
- Runtime: $O(n \cdot C)$
- Space: $O(n \cdot C)$

Input: Integers n, C, arrays W, V, and memo table DP.
Output: The maximum total value of objects the Knapsack can hold

function KNAPSACK(n, C, W, V)
    $DP[n][C] \leftarrow -1$ ▷ Initialize values in lookup-table to -1
    for $i = 0 : n$ **do**
        $DP[i][0] = 0$
    for $j = 0 : C$ **do**
        $DP[0][j] = 0$
    for $i = 1 : n$ **do**
        for $j = 1 : C$ **do**
            if $W[i] > j$ **then**
                $DP[i][j] = DP[i-1][j]$
            else
                $DP[i][j] = \max(DP[i-1][j-W[i]] + V[i], DP[i-1][j])$
    return $DP[n][C]$

- Advantages:
  - Almost always fastest in practice
  - Easier to extend the algorithm to reconstruct solution
  - [Practical] Don’t have to worry about seg fault
- Disadvantage: Less intuitive

Worksheet Problems

Matrix Multiplication

($M[1,\ldots,n]$): Given a sequence of matrices to multiply together, write a recurrence to determine the minimum number of element-element multiplications required (you may disregard additions).

As an example, suppose the sequence of matrices to multiply is ABC, where A is 2×3, B is 3×4, and C is 4×5.

Computing $(AB)C$ requires $2\cdot 3\cdot 4+2\cdot 4\cdot 5=64$ multiplications. Computing $A(BC)$ requires $3\cdot 4\cdot 5+2\cdot 3\cdot 5=90$ multiplications. Therefore, the minimum number of multiplications required is 64.

The following is a short example of how to perform matrix multiplication:

If $A=\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ and $B=\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$, then $AB=\begin{bmatrix} a_{00}b_{00}+a_{01}b_{10} & a_{00}b_{01}+a_{01}b_{11} \\ a_{10}b_{00}+a_{11}b_{10} & a_{10}b_{01}+a_{11}b_{11} \end{bmatrix}$.

Matrix Multiplication

The following is a short example of how to perform matrix multiplication:

If $A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$ and $B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$, then $AB = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{bmatrix}$.

- **Sanity check:** What is the number of multiplications in AB in terms of
  - A.r = number of rows of matrix A
  - A.c = number of columns of matrix A
  - B.r = number of rows of matrix B
  - B.c = number of columns of matrix B

Ans: A.r · A.c · B.c = A.r · B.r · B.c

Matrix Multiplication Recurrence

- **Step 1: Subject of recurrence**
  - Let $L(i,j)$ = minimum number of multiplications to multiply matrices i through j
- **Step 2: Base cases**
  - $i = j$ (One matrix): $M(i,j) = 0$
- **Step 3: Optimal sub-solution**
  - [Sub-solution] For $M_i M_{i+1} \dots M_j$, how do we reduce the problem?
    - "Partition" by some k: $(M_i \dots M_k)(M_{k+1} \dots M_j) = AB$
    - A is M_i.r × M_k.c; B is M_{k+1}.r × M_j.c = M_k.c × M_j.c
    - Number of multiplication to multiply AB? M_i.r · M_k.c · M_j.c from previous slide
  - [Optimal] *Choose* the best k
    - Minimization problem: Use min across all k's in range $i \le k < j$
    - Objective function? $L(i,j) = \min_{i \le k < j} (L(i,k) + L(k+1,j) + M[i].r \cdot M[k].c \cdot M[j].c).$

Edit Distance

Imagine that you are building a spellchecker for a word processor. When the spellchecker encounters an unknown word, you want it to suggest a word in its dictionary that the user might have meant (perhaps they made a typo). One way to generate this suggestion is to measure how "close" the typed word A is to a particular word B from the dictionary, and suggest the closest of all dictionary words. There are many ways to measure closeness; in this problem we will consider a measure known as the *edit distance*, denoted EDIT-DIST(A, B).

In more detail, given a strings A and B, consider transforming A into B via character *insertions* (i), *deletions* (d), and *substitutions* (s). For example, if A = `ALGORITHM` and B = `ALTRUISTIC`, then one way of transforming A into B is via the following operations:

A	L	G	O	R		I		T	H	M
A	L	T		R	U	I	S	T	I	C
		s	d		i		i		s	s

Write the base case(s) and recurrence for **EDIT-DIST(A, B), the minimal cost of transforming string A to string B**, parameterized by the following three numbers:

- c_i, the cost to *insert* a character,
- c_d, the cost to *delete* a character,
- c_s, the cost to *substitute* a character.

Edit Distance Recurrence

- **Step 1: Subject of recurrence**
  - Suppose we have arrays A[1, …, n] and B[1, …, m]
  - Let $ED(i, j)$ = edit distance from A[1, …, i] to B[1, …, j]
- **Step 2: Base cases**
  - $i = 0$ (A is empty):
    - Insert all j characters from B to A
    - $ED[0, j] = c_i \cdot j$
  - $j = 0$ (B is empty):
    - Delete i characters from A
    - $ED[i, 0] = c_d \cdot i$

Edit Distance Recurrence

- **Step 1: Subject of recurrence**
  - Suppose we have arrays A[1, …, n] and B[1, …, m]
  - Let $ED(i, j)$ = edit distance from A[1, …, i] to B[1, …, j]
- **Step 2: Base cases**
  - $i = 0$ (A is empty): $ED[0, j] = c_i \cdot j$
  - $j = 0$ (B is empty): $ED[i, 0] = c_d \cdot i$
- **Step 3: Optimal sub-solution**
  - [Sub-solution] From A[1, …, i] and B[1, …, j], how do we reduce the problem?
    - Reduce into A[1, …, i-1] and B[1, …, j], OR
    - A[1, …, i] and B[1, …, j-1], OR
    - A[1, …, i-1] and B[1, …, j-1]
    - How to know which one to recurse into?

Edit Distance Recurrence

- **Step 3: Optimal sub-solution**
  - [Sub-solution] From A[1, …, i] and B[1, …, j], how do we reduce the problem?
    - Compare A[i] and B[j]
    - If A[i] = B[j], do nothing and recurse into A[1, …, i-1] and B[1, …, j-1]
    - If A[i] ≠ B[j], what options do we have?
      - [1] Insert B[j] into A[i+1] then recurse into A[1, …, i] and B[1, …, j-1]
      - [2] Delete A[i] then recurse into A[1, …, i-1] and B[1, …, j]
      - [3] Substitute A[i] with B[j] then recurse into A[1, …, i-1] and B[1, …, j-1]
  - [Optimal] *Choose* the best option
    - If A[i] = B[j], we don't have to choose: $\mathbf{ED}(i,j) = \mathbf{ED}(i-1, j-1)$
    - If A[i] ≠ B[j], we have a minimization problem: Use min across options [1], [2], [3]

$$\mathbf{ED}(i,j) = \min \begin{cases} c_i \ + \ \mathbf{ED}(i, j-1) \\ c_d \ + \ \mathbf{ED}(i-1, j) \\ c_s \ + \ \mathbf{ED}(i-1, j-1) \end{cases}$$

Longest Palindromic Subsequence

Given a string S, write a recurrence relation to determine the length of a longest subsequence of S (not necessarily a substring) that is a palindrome. Recall that a palindrome is a string which is the same forwards and backwards (e.g., "racecar").

- Example: LPS("abca") = 3 ("aba" or "aca")
- **Step 1: Subject of recurrence**
  - $LPS[i,j]$ = length of the longest palindromic subsequence of $S[i,\dots,j]$
- **Step 2: Base cases**
  - $i = j$ (string of length 1): $LPS[i,j] = 1$
  - $i > j$ (invalid input): $LPS[i,j] = 0$

Longest Palindromic Subsequence

- Step 3: Optimal sub-solution
  - [Sub-solution] From S[i, …, j], how to reduce the problem?
    - Compare S[i] and S[j]
    - If S[i] = S[j], add 2 and recurse into S[i+1, …, j-1]
    - If S[i] ≠ S[j], what options do we have?
      - [1] Recurse into S[i+1, …, j]
      - [2] Recurse into S[I, …, j-1]
  - [Optimal] *Choose* the best option
    - If A[i] = B[j], we don't have to choose: $2 + LPS[i+1, j-1]$
    - If A[i] ≠ B[j], we have a maximization problem: Use max across options [1] and [2]

$$\max\{LPS[i+1, j], LPS[i, j-1]\}$$

DP Recurrence- Takeaway

- **Step 1: Define the subject of recurrence**
  - Is this a 1D DP problem? 2D? What dimension do we want to do it in?
  - In which/ how many direction(s) do we want to reduce the problem?
- **Step 2: Identify the base case(s)**
  - In what situation(s) we can't reduce the problem further?
  - Is there any special cases?
- **Step 3: Construct (optimal) sub-solution**
  - Sub-solution:
    - [Sub] How to reduce the problem to smaller version of the same problem?
    - [Solution] How to combine the result so that the overall result is correct?
    - If [some condition] is/ isn't satisfied, what options do we have?
  - Optimal: (only for optimization problem)
    - Is it a maximization or minimization problem?
    - What is/ are the variable(s) we're taking max/ min over?
    - What is the objective function to be maximize/ minimize?