Dynamic Programming I

Week 9: *In-Class*

Yong-Hyuk Moon @ SAIL

The Department of Computer Engineering
Sungshin Women's University

Smart Attendance:

*****

Administrative

- **Midterm Exam**
  - Score Distribution
    - ⇢ We appreciate your efforts on the midterm exam.
    - ⇢ The score distribution and overall statistics will be posted soon on LMS to help you gauge your standing in the course.

Quick Recap & Warm-Up Questions

- **Pre-Class Recap**
  - Explaining the strategic difference between D&C and DP
  - Understanding the DP framework using the Fibonacci sequence problem
  - Applying memoization and tabulation to save solutions for subproblems

Can you explain the most fundamental difference in the problem-solving approach between Divide and Conquer (D&C) and Dynamic Programming (DP)?

Storing the solutions to subproblems can reduce time complexity – so why do we need both memoization and tabulation as separate techniques?

Can you distinguish between top-down and bottom-up strategies, and explain how they relate to recursive and iterative implementations?

Today's Topics

- **What's Coming Up in Class?**
  - Examining the role of the call stack in space complexity
  - Identifying the characteristics of problems where dynamic programming can be applied
  - Understanding and applying memoization and tabulation techniques
  - Developing the skill to derive recurrence relations for given problems
    - ⇢ Computing Binomial Coefficients

Call Stack and Space Complexity

Aspect	D&C	DP	DP	D&C	D&C
Strategy	Top-Down, Recursive	Top-Down, Recursive with Memoization	Bottom-Up, Iterative with Tabulation	Top-Down, Recursive, Matrix Exponentiation	Bottom-Up, Iterative, Matrix Exponentiation
Time Complexity	$O(2^n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Space Complexity	$O(n)$ call stack	$O(n)$ call stack + mem list	$O(n)$ no call stack + table	$O(logn)$ call stack	$O(1)$ no call stack + constant storage

Space Complexity: *Call Stack*

- **Call Stack: Precise Space Analysis**
  - A system stack structure that stores information about function calls during program execution.
    - ① When a function is called, information is saved in a stack frame.
    - ② When the function finishes, the corresponding stack frame is popped and removed from the stack.
    - ③ The system manages this process using the "Call Stack."

```python
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

- **Example: Calling fib(5)**
  - In this process, fib(5) waits for fib(4) to complete, fib(4) waits for fib(3), and so on.
  - Function calls accumulate on the stack as they are called but not yet completed.
  - At its maximum depth, the stack grows like this: fib(n) → fib(n-1) → ... → fib(1)
    - ⇢ This shows the stack depth.

```
fib(5)
├── fib(4)
│   ├── fib(3)
│   │   ├── fib(2)
│   │   │   ├── fib(1) ← base
│   │   │   └── fib(0) ← base
│   │   └── fib(1) ← base
│   └── fib(2) ...
└── fib(3) ...
```

Space Complexity: *Recursive Approach*

- **Why is the space complexity of a D&C recursive algorithm O(n)?**
  - The depth of nested function calls determines the amount of call stack space used.
  - Refer to the table below.

Component	Description
Number of Stack Frames	Up to n functions may exist simultaneously on the stack
Space per Frame	Each frame occupies a constant amount of space
Total Space Usage	$O(n)$ stack frames → $O(n)$ space complexity

  - Information stored in the stack frame:

⇢ return value, return address, previous frame pointer, parameters, and local variables

Space Complexity: *Iterative Approach*

- **Why is the space complexity of iterative loops O(1)?**
  - Only a single function is executed, and only looping occurs.
  - No additional stack frames are accumulated.
  - Therefore, the space complexity is O(1).

- **Comparison**
  - Additional topic: memoization vs. tabulation (covered today)

Aspect	D&C	DP	DP	DP
Strategy	Top-Down, Recursive	Top-Down, Recursive with Memoization	Bottom-Up, Iterative with Tabulation	Bottom-Up, Iterative using Constant Space
Time Complexity	$O(2^n)$	$O(n)$	$O(n)$	$O(n)$
Space Complexity	$O(n)$ call stack	$O(n)$ call stack + mem list	$O(n)$ no call stack + table	$O(1)$ no call stack constant space

```python
def fib(n):
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Applying DP in Practice

Dynamic Programming Paradigm

- **What Types of Problems Are Suitable for DP?**
  - To apply dynamic programming, a problem must satisfy both of the following properties:

① Overlapping Subproblem

: The same subproblems appear multiple times.

⇢ In the Fibonacci sequence, recursive calls form a tree where many subproblems repeat.

⇢ In contrast, binary search does not have overlapping subproblems.

② Optimal Substructure

: An optimal solution to the whole problem can be derived from optimal solutions to its subproblems.

⇢ If we know the optimal results for fib(4) and fib(5), we can compute the optimal result for fib(6).

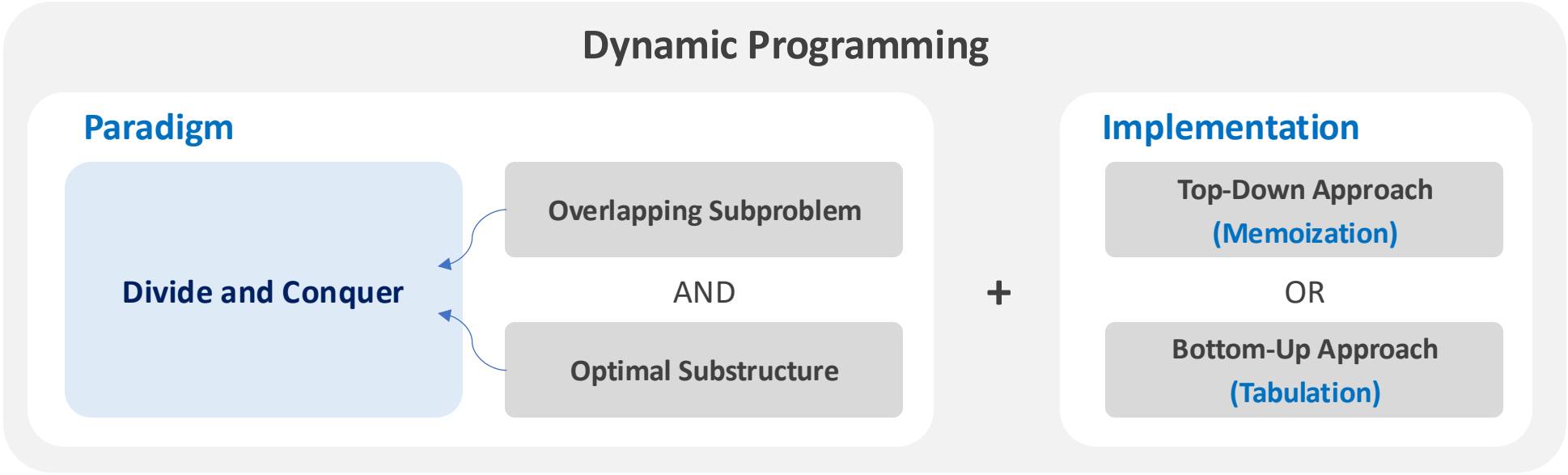
⇢ Many problems exhibit this property including the shortest path problem, the knapsack problem, etc.

  - What about the factorial problem? → ①, ②

Dynamic Programming Paradigm

- **Divide-and-Conquer and Dynamic Programming Are Closely Related**
  - If a problem suitable for divide-and-conquer also satisfies the two DP properties above,
  - Then dynamic programming can be applied (using one of two methods)

    ⇢ i.e., memoization or tabulation



DP Strategies

• DP: Memoization vs. Tabulation

- Tabulation is generally faster because it avoids the overhead of recursive calls.
- It also allows the table to be built outside of the function, making it easier to modularize the code.
- However, if it's hard to predict which subproblems will be solved in advance, memoization is a better choice.

• Implementation Differences

Aspect	Top-down (Recursive)	Bottom-up (Iterative)
Approach	Start from the original problem and break it down	Start from the smallest subproblems and build up
Implementation	Recursive calls (used call stack)	Iterative loops (e.g., for, while to fill table)
Storage Method	Memoization (store results when needed)	Tabulation (fill the table incrementally from the base case)
Example	Computing Binomial Coefficients	Fibonacci Sequence, Longest Common Sequence

DP Strategies

- **Selection Criteria**
  - ✓ When performance or memory is critical, prefer a Bottom-Up approach.
  - ✓ When rapid prototyping or intuitive structure matters more, go with Top-Down.
  - ✓ If both are viable, the choice depends on the problem's nature and constraints.

- **Detailed Comparison**
  - I'll leave the details to you – it's a good opportunity to reinforce your understanding.

Aspect	To—Down (Momoization)	Bottom-Up (Tabulation)
Code Structure	Based on recursion → simpler and more intuitive	Based on loops → iterative logic of more verbose
Number of Computations	May skip some subproblems → efficient if not all are needed	All subproblems must be computed
Function Call Cost	Higher (recursive calls + memo lookups)	Lower (no recursion, sequential access to table)
Storage Usage	Call stack + memo table → harder to manage	Only a table (easier to control and analyze)
Design/Debugging	Easier to write but harder to debug stack behavior	Requires planning, but often more stable
Language Constraints	Not all languages optimize tail recursion (e.g., python)	Works uniformly across languages
Runtime Performance	May be slower due to recursive call overhead	Often faster (no function calls)

DP Strategies: *Tabulation*

- **Design Process: How to Apply DP with Tabulation?**

① Break down the given problem into subproblems:

Derive a recurrence relation that defines how the original problem relates to its subproblems.

② Prepare a table to store the solutions to the subproblems.

③ Based on the problem definition, identify and store the base cases in the table.

(Base cases are the smallest subproblems with known answers.)

④ Use the stored base cases to iteratively compute the solutions to larger subproblems.

- **Next, let's solve the following problems using the tabulation approach.**

Computing Binomial Coefficients

- **Binomial Theorem:**

  A formula that expands the power of a binomial (an algebraic sum of two terms).

  - e.g.,

$$(a+b)^3 = aaa + aab + aba + baa + abb + bab + bba + bbb$$
$$= a^3 + 3a^2 b + 3ab^2 + b^3$$

  - When expanded in its generalized form:

$$(a+b)^n = a^n + {}_nC_1 a^{n-1}b + {}_nC_2 a^{n-2}b^2 + \cdots + {}_nC_k a^{n-k}b^k + \cdots + b^n$$

  - Binomial Coefficients:

$${}_nC_0(=1), {}_nC_1, {}_nC_2, \ldots, {}_nC_k, \ldots {}_nC_n(=1)$$

  ⇢ ${}_nC_k \Leftrightarrow C(n,k)\ or\ \binom{n}{k}$

  ⇢ These coefficients represent the number of unordered combinations of selecting *k* elemetns from a set of *n*.

Computing Binomial Coefficients

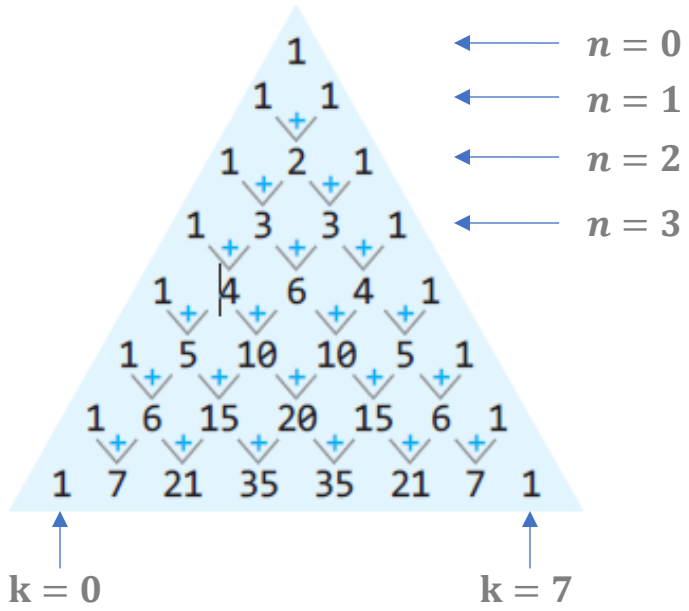
- **Binomial Theorem:**
  - Rewriting the binomial expression:

$$(a+b)^n = \sum_{k=0}^{n} C(n,k)\, a^{n-k}\, b^k$$

  - Binomial coefficients can be computed using a closed-form formula:

$$C(n,k) = \frac{n!}{k!\,(n-k)!}$$

  - But let's try computing them using dynamic programming instead.
    - ⇢ Can we break down the computation of binomial coefficients?
    - ⇢ They can be computed step-by-step using Pascal's Triangle:
      a) Start with a root value of 1.
      b) Each value is the sum of the number above to the left and the number above to the right.
      c) The outermost numbers (far left and right) are always 1.
      d) The k-th element in the n-th level of the triangle is denoted as C(n,k).
    - ⇢ What about $(a+b)^3$?
      - ✓ At level n = 3, $C(3,0) = 1, C(3,1) = 3, C(3,2) = 3, C(3,3) = 1$



Computing Binomial Coefficients

- **Recurrence Relation**
  - **Base Case ($k = 0, k = n$)**
    - ⇢ These are the simplest cases where the answer is already known.
    - ⇢ What kind of cases are these? Cases where there is only one way to choose!
    - ⇢ Specifically, there are two such cases: $C(n, 0) = C(n, n) = 1$
    - ⇢ In other words, choosing none of the items or choosing all of them — both have exactly one possible combination.
  - **General Case ($k \neq 0 \neq n$)**
    - ⇢ Break the problem into two independent cases:
    - ① When the n-th item is excluded: Choose k items from the remaining $n - 1$ items → $C(n-1, k)$
    - ② When the n-th item is included: Choose the remaining $k - 1$ items from the remaining $n-1$ → $C(n-1, k-1)$

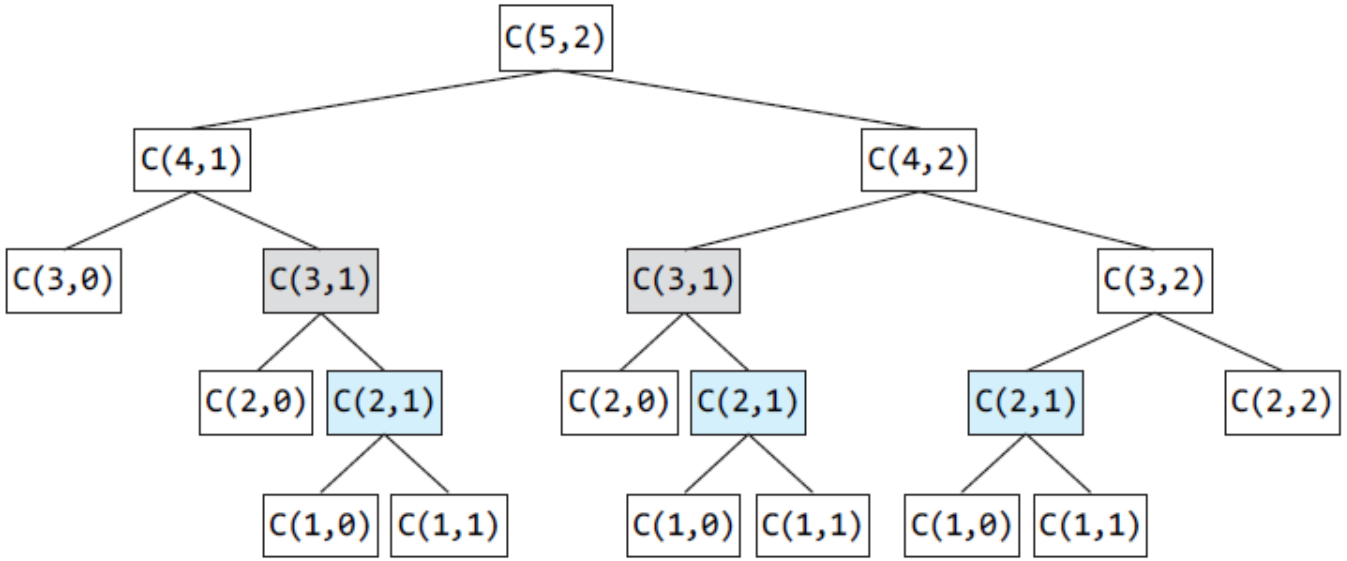
$$c(n,k) = \begin{cases} 1 & k = 0, k = n \\ C(n-1, k-1) + C(n-1, k) & otherwise \end{cases}$$

Computing Binomial Coefficients

- **Applying D&C to solve $C(n, k)$**

```python
def bino_coef_dc(n, k):
    if k == 0 or k == n:
        return 1
    return bino_coef_dc(n-1, k-1) + bino_coef_dc(n-1, k)
```

- Heavy overlap in computations



Computing Binomial Coefficients

- **Applying DP to Solve $C(n, k)$**
  - This problem satisfies the optimal substructure property.
  - In addition, from the D&C tree, we also observe overlapping subproblems.
- **Table Design: C**
  - We need to store values from $C(0,0)$ up to $C(n, k)$ → Table size: $(n+1) \times (k+1)$
    - ⇢ For $k = 0 = n$, all values are 1.
    - ⇢ For $k > n$, the value is undefined. → Skip or mark as invalid
  - The following table shows an example of $C(5,3)$.

	$k=0$	$k=1$	$k=2$	$k=3$
$n=0$	1			
$n=1$	1	1		
$n=2$	1	2	1	
$n=3$	1	3	3	1
$n=4$	1	4	6	4
$n=5$	1	5	10	10

Computing Binomial Coefficients

- **Implementing DP with Tabulation to Solve $C(n, k)$**

```
1  def bino_coef_dp(n, k):
2      C = [[-1 for _ in range(k+1)] for _ in range(n+1)]
3
4      for i in range(n+1):
5          for j in range(min(i, k)+1):
6              if j == 0 or j == i:
7                  C[i][j] = 1
8              else:
9                  C[i][j] = C[i-1][j-1] + C[i-1][j]
10
11     return C[n][k]
```

- Line 2: C = [[-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]] → 2D List
- Time Complexity: ( ), Space Complexity: ( )
- Further considerations: memoization - or any potential improvements?

Weekly Breakdown

• Week 9: Key Takeaways

- Dynamic Programming I
  1. Explaining the strategic difference between D&C and DP
  2. Understanding the DP framework using the Fibonacci sequence problem
  3. Examining the role of the call stack in space complexity
  4. Identifying the characteristics of problems where dynamic programming can be applied
  5. Exploring when to choose memoization vs. tabulation, based on their characteristics
  6. Developing the skill to derive recurrence relations for computing binomial coefficients

• Next Week's Topics

- Dynamic Programming II

Any Final Thoughts?

yhmoon@sungshin.ac.kr

The Department of Computer Engineering
Sungshin Women's University