

Dynamic Programming



EECS3101 E:
Design and Analysis of Algorithms
Summer 2026

CHEN-WEI WANG

Learning Outcomes of this Lecture

This module is designed to help you:

- Recognize when **Dynamic Programming (DP)** is appropriate:
 - **overlapping subproblems**
 - **optimal substructure**
- Define **state**, **recurrence relation**, and **base cases**.
- Differentiate the two main approaches to **DP**:
 - **Top-Down DP**: recursive, with **memoization**
 - **Bottom-Up DP**: iterative, with **tabulation**
- Analyze time and space complexities of **DP** algorithms.

Dynamic Programming (DP): Why

- A **recursive** solution:
 - may be natural/straightforward and elegant to define
 - explores a (large) number of possibilities [runtime **recursion tree**]
 - but (many) branches may **repeatedly** solve the **same subproblems**
⇒ **exponential** running time ∴ unnecessary recomputations
- **DP** works by:
 - solving **each subproblem once**
 - **storing** its answer
 - **reusing** that stored answer later
- **DP** is applicable when:
 - **Subproblems overlap**
 - Same subproblem appears **multiple** times in the **recursive** process.
 - e.g., To solve *fib*(6), we need to solve *fib*(4) twice. Why?
 - **The problem has optimal substructure**
 - Optimal solution to the **original problem** can be built from optimal solutions to smaller **subproblems**.
 - Without **optimal substructure**, solutions to smaller subproblems **may not combine correctly** into a solution to the larger problem.

Dynamic Programming (DP): What

- **DP** is a general algorithm design technique for problems with:
 - **overlapping subproblems**
 - **optimal substructure**
- **DP** is often used to solve problems asking for, e.g.,:
 - **number of ways**
e.g., Number of distinct ways to climb a staircase of height n ?
 - **minimum or maximum value**
e.g., Minimum total cost to climb to the top of a staircase?
 - **feasibility**
e.g., Possible to choose some numbers whose sum is exactly S ?
 - **best value under constraints**
e.g., Max total value of items that fit into a bag of limited capacity?
- Why the name?
 - “**Programming**”: computations planned in a **step-by-step** manner
 - “**Dynamic**”: solution to the current **subproblem** built from solutions to smaller **subproblems** computed earlier

Dynamic Programming (DP): How

- Define the **state** and **subproblem**.
e.g., In solving the **Fibonacci problem**,
the **state** is the index $i \geq 0$, and
the corresponding **subproblem** is to compute $fib(i)$.
- Break the original **problem** into smaller **subproblems**.
- Express the solution using a **recurrence relation**.
- Identify the **base cases**.
- Compute solutions using one of two approaches:
 - **Top-Down**: **recursion** + **memoization**
 - solve the original problem recursively
 - store answers to **subproblems**
 - reuse stored answers to **avoid recomputations**
 - **Bottom-Up**: **iteration** + **tabulation**
 - solve the original problem iteratively
 - fill the (1D or 2D) table between iterations
 - build toward the final answer from the smallest **subproblems** upward

DP Example (1.1): Fibonacci Number

- Can you identify the pattern of a *Fibonacci sequence*?

$$F = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

- The formal *recurrence relation* for computing the i_{th} *Fibonacci number* (denoted as F_i):

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{if } i > 1 \end{cases}$$

- Then, straightforward to transform F_i into executable Java:

```
static int fib(int i) {  
    if(i < 0) { throw new IllegalArgumentException("Negative i"); }  
    if(i == 0) { /* base case */ return 0; }  
    if(i == 1) { /* base case */ return 1; }  
    /* recursive case */  
    return fib(i - 1) + fib(i - 2);  
}
```

DP Example (1.2): Fibonacci Number

A top-down DP approach using *recursion* and *memoization*:

```
1 static int fibTopDown(int i) {
2     if(i < 0) { throw new IllegalArgumentException("Negative i"); }
3     /* memoization: storing the result for each fib(i), i >= 0 */
4     int[] memo = new int[i + 1];
5     /* Initialize base cases. */
6     Arrays.fill(memo, -1); memo[0] = 0; if(i > 0) { memo[1] = 1; }
7     return fibTopDownH(i, memo);
8 }
9 static int fibTopDownH(int i, int[] memo) {
10     if(i == 0) { /* base case */ return memo[0]; }
11     if(i == 1) { /* base case */ return memo[1]; }
12     /* recursive case: reusing previous computation */
13     if(memo[i] != -1) { return memo[i]; }
14     /* recursive case: compute once, store once, reuse later */
15     memo[i] = fibTopDownH(i - 1, memo) + fibTopDownH(i - 2, memo);
16     return memo[i];
17 }
```

DP Example (1.3): Fibonacci Number

A bottom-up DP approach using *iteration* and *tabulation*:

```
1 static int fibBottomUp(int i) {
2     if(i < 0) { throw new IllegalArgumentException("Negative i"); }
3     /* tabulation: tab[i] stores fib(i), i >= 0 */
4     int[] tab = new int[i + 1];
5     /* Initialize base cases. */
6     Arrays.fill(tab, -1); tab[0] = 0; if(i > 0) { tab[1] = 1; }
7     if(i == 0) { /* base case */ return tab[0]; }
8     if(i == 1) { /* base case */ return tab[1]; }
9     /* Build from smaller subproblems upward. */
10    for(int j = 2; j <= i; j++) {
11        tab[j] = tab[j - 1] + tab[j - 2];
12    }
13    return tab[i];
14 }
```

DP Example (2.1): Factorial

- The formal **recurrence relation** for computing the **factorial** of i (denoted as $i!$):

$$i! = \begin{cases} 1 & \text{if } i = 0 \\ i \times (i-1)! & \text{if } i > 0 \end{cases}$$

- Then, straightforward to transform $i!$ into executable Java:

```
static int fac(int i) {  
    if(i < 0) { throw new IllegalArgumentException("Negative i"); }  
    if(i == 0) { /* base case */ return 1; }  
    /* recursive case */  
    return i * fac(i - 1);  
}
```

DP Example (2.2): Factorial

A top-down DP approach using *recursion* and *memoization*:

```
1 static int facTopDown(int i) {
2     if(i < 0) { throw new IllegalArgumentException("Negative i"); }
3     /* memoization: storing the result for each fac(i), i >= 0 */
4     int[] memo = new int[i + 1];
5     /* Initialize base cases. */
6     Arrays.fill(memo, -1); memo[0] = 1;
7     return facTopDownH(i, memo);
8 }
9 static int facTopDownH(int i, int[] memo) {
10    if(i == 0) { /* base case */ return memo[0]; }
11    /* recursive case: reusing previous computation */
12    if(memo[i] != -1) { return memo[i]; }
13    /* recursive case: compute once, store once, reuse later */
14    memo[i] = i * facTopDownH(i - 1, memo);
15    return memo[i];
16 }
```

DP Example (2.3): Factorial

A bottom-up DP approach using *iteration* and *tabulation*:

```
1 static int facBottomUp(int i) {  
2     if(i < 0) { throw new IllegalArgumentException("Negative i"); }  
3     /* tabulation: tab[i] stores fac(i), i >= 0 */  
4     int[] tab = new int[i + 1];  
5     /* Initialize base cases. */  
6     Arrays.fill(tab, -1); tab[0] = 1;  
7     if(i == 0) { /* base case */ return tab[0]; }  
8     /* Build from smaller subproblems upward. */  
9     for(int j = 1; j <= i; j++) {  
10         tab[j] = j * tab[j - 1];  
11     }  
12     return tab[i];  
13 }
```

DP: Fibonacci vs. Factorial

- Fibonacci
 - **optimal substructure**
 - **overlapping subproblems**
 - Recursive solution:
 - **Space** Complexity: $O(n)$ [recursion stack depth]
 - **Time** Complexity: $O(2^n)$ [recursion tree branches heavily]
 - **DP** is useful
 - ⇒ top-down and bottom-up approaches both improve efficiency
 - **Space** Complexity: $O(n)$ [recursion stack depth vs. 1D table size]
 - **Time** Complexity: $O(n)$ [each $fib(i)$ solved once]
- Factorial
 - **optimal substructure**
 - **no** overlapping subproblems
 - Recursive solution:
 - **Space** Complexity: $O(n)$ [recursion stack depth]
 - **Time** Complexity: $O(n)$ [only one recursive branch]
 - **DP** is **not** really needed
 - ⇒ top-down and bottom-up approaches do **not** improve efficiency
 - **Space** Complexity: $O(n)$ [recursion stack depth vs. 1D table size]
 - **Time** Complexity: $O(n)$ [each $fac(i)$ solved once, stored result not reused]

DP Example (3.1a): Minimum Cost Climbing

- **In:** an array *cost*, where *cost*[*i*] denotes the cost of stepping on stair *i*
- **Rules:**
 - Start climbing at stair 0 or stair 1.
 - From stair *i*, one may climb one step to stair *i* + 1, or two steps to stair *i* + 2.
 - The top (of the staircase) refers to the imaginary index *cost.length*.
- **Out:** *minimum total cost to climb to the top or beyond*
 - {5} → 0 [start: stair 1 → top reached, costing nothing]
 - {10, 15} → 10 [pay: stair 0; climb 2 steps to top]
 - {15, 10} → 10 [pay: stair 1; climb 1 step to top]
 - {5, 5, 3} → 5 [pay: stair 1; climb 2 steps to top]
 - {4, 3, 5, 7} → 8 [pay: stair 1; climb 1 step; pay: stair 2; climb 2 steps to top]

DP Example (3.1b): Minimum Cost Climbing

- Given n stairs, the formal **recurrence relation** for computing the **minimum cost to climb to the top or beyond** from stair i (denoted as M_i):

$$M_i = \begin{cases} 0 & \text{if } i \geq n \\ \text{cost}[i] + \min(M_{i+1}, M_{i+2}) & \text{if } 0 \leq i < n \end{cases}$$

- One may start from stair 0 or stair 1 $\Rightarrow$ **Final output:** $\min(M_0, M_1)$
- Then, straightforward to transform M_i into executable Java:

```
static int minCostClimbing(int[] cost) {
    if(cost == null || cost.length == 0) { throw new IllegalArgumentException(...); }
    /* Starting at stair 1 means already being at the top, costing nothing. */
    if(cost.length == 1) { return 0; }
    /* Start at either stair 0 or stair 1. */
    return Math.min(minCostClimbingH(cost, 0), minCostClimbingH(cost, 1));
}

static int minCostClimbingH(int[] cost, int i) {
    /* base case: when the top or beyond is reached, pay nothing */
    if(i >= cost.length) { return 0; }
    /* recursive case: after paying cost[i], choose the cheaper of 1-step or 2-step move */
    return cost[i] + Math.min(minCostClimbingH(cost, i + 1), minCostClimbingH(cost, i + 2));
}
```

DP Example (3.2): Minimum Cost Climbing

A top-down DP approach using *recursion* and *memoization*:

```
1 static int minCostClimbingTopDown(int[] cost) {
2     if(cost == null || cost.length == 0) { throw new IllegalArgumentException(...); }
3     /* Starting at stair 1 means already being at the top, costing nothing. */
4     if(cost.length == 1) { return 0; }
5     /* memo[i]: minimum cost to reach the top or beyond, starting from stair i */
6     int[] memo = new int[cost.length];
7     /* Initialize: memo[i] == -1 means no cost determined and stored yet */
8     Arrays.fill(memo, -1);
9     /* Start at either stair 0 or stair 1. */
10    return Math.min(mccTD_H(cost, 0, memo), mccTD_H(cost, 1, memo));
11 }
12 static int mccTD_H(int[] cost, int i, int[] memo) {
13     /* base case: when the top or beyond is reached, pay nothing */
14     if(i >= cost.length) { return 0; }
15     /* recursive case: reusing previous computation */
16     if(memo[i] != -1) { return memo[i]; }
17     /* recursive case: compute once, store once, reuse later */
18     memo[i] = cost[i] + Math.min(mccTD_H(cost, i + 1, memo), mccTD_H(cost, i + 2, memo));
19     return memo[i];
20 }
```

DP Example (3.3): Minimum Cost Climbing

A bottom-up DP approach using *iteration* and *tabulation*:

```
1 static int minCostClimbingBottomUp(int[] cost) {
2     if(cost == null || cost.length == 0) { throw new IllegalArgumentException(...); }
3     /* Starting at stair 1 means already being at the top, costing nothing. */
4     if(cost.length == 1) { return 0; }
5     /* tab[i]: minimum cost to reach the top or beyond, starting from stair i
6      * The last two indices of 'tab', n and n + 1, denote "top" and "beyond".
7      */
8     int[] tab = new int[cost.length + 2];
9     /* Initialize base cases: starting at or beyond the top costs 0 */
10    Arrays.fill(tab, -1); tab[cost.length] = 0; tab[cost.length + 1] = 0;
11    /* Fill backwards because tab[i] depends on tab[i+1] and tab[i+2]. */
12    for(int i = cost.length - 1; i >= 0; i--) {
13        tab[i] = cost[i] + Math.min(tab[i + 1], tab[i + 2]);
14    }
15    /* Return the cheaper total cost: starting from stair 0 or starting from stair 1. */
16    return Math.min(tab[0], tab[1]);
17 }
```

Beyond this lecture

- Explore the source code on the lectures site:
`ExampleDynamicProgramming.zip`

Index (1)

Learning Outcomes of this Lecture

Dynamic Programming (DP): Why

Dynamic Programming (DP): What

Dynamic Programming (DP): How

DP Example (1.1): Fibonacci Number

DP Example (1.2): Fibonacci Number

DP Example (1.3): Fibonacci Number

DP Example (2.1): Factorial

DP Example (2.2): Factorial

DP Example (2.3): Factorial

DP: Fibonacci vs. Factorial

Index (2)

DP Example (3.1a): Minimum Cost Climbing

DP Example (3.1b): Minimum Cost Climbing

DP Example (3.2): Minimum Cost Climbing

DP Example (3.3): Minimum Cost Climbing

Beyond this lecture