CS 2500: Algorithms

Lecture 23: Dynamic Programming: Introduction

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

November 5, 2024

Problem: Calculating Binomial Coefficient

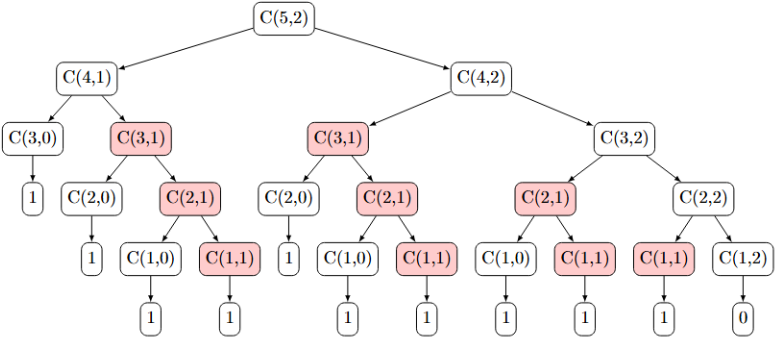

Figure: Recursion tree for calculating the binomial coefficient. We notice that same subproblem is being solved multiple times.

Problem: Calculating Binomial Coefficients

- To calculate the binomial coefficient $C(n, k)$, we can divide the problem into subproblems, calculating $C(n-1, k-1)$ and $C(n-1, k)$, and then combine these to obtain the original solution $C(n, k) = C(n-1, k-1) + C(n-1, k)$.
- This recursive approach often results in overlapping subproblems, as many values of $C(i, j)$ are computed multiple times across different branches of the recursion tree.
- For example, calculating $C(5, 2)$ requires calculating $C(4, 1)$ and $C(4, 2)$, which themselves require calculations for $C(3, 1)$ multiple times, leading to redundant calculations.
- If we ignore this duplication, the algorithm becomes inefficient, with exponential time complexity.

Solution: Dynamic Programming

Dynamic programming (DP): A method for solving complex problems by breaking them down into simpler, smaller subproblems and solving each subproblem only once.

Key Idea: Avoid calculating the same thing twice, usually by keeping a table of known results that fills up as subinstances are solved.

What is Dynamic Programming?

Overlapping Subproblems:

- Dynamic programming (DP) is effective when a problem can be divided into subproblems that are solved multiple times.
- Example: In the binomial coefficient calculation, calculating $C(5,2)$ involves calculating $C(3,1)$ multiple times, as it appears in different branches of the recursion tree.
- DP saves the results of each subproblem in a table (or memoization) to avoid re-computing them, reducing the time complexity from exponential to polynomial.

What is Dynamic Programming?

Optimal Substructure:

- A problem exhibits optimal substructure if an optimal solution to the problem can be constructed from optimal solutions to its subproblems.
- Example: In the binomial coefficient problem, to compute $C(n, k)$, we can use the formula:

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

  which means the optimal solution $C(n, k)$ depends on the optimal solutions to the subproblems $C(n-1, k-1)$ and $C(n-1, k)$.
- DP leverages this property to build solutions from previously computed optimal solutions to subproblems, ensuring each subproblem is only solved once.

Dynamic Programming Approach for Binomial Coefficients

	0	1	2	3	$\cdots$	$k-1$	k
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		
$n-1$	1	$C(n-1,1)$	$C(n-1,2)$	$\cdots$		$C(n-1,k-1)$	1
n	1	$C(n,1)$	$C(n,2)$	$\cdots$	$C(n,k-2)$	$C(n-1,k-1)+C(n-1,k)$	$C(n,k)$

Figure: To calculate binomial coefficients efficiently, we can use a table of intermediate results. This table can be filled line by line, where each entry $C(n,k)$ is the sum of $C(n-1,k-1)$ and $C(n-1,k)$. By storing only the current line, we reduce space complexity to $O(k)$ and time complexity to $\Theta(nk)$

Dynamic Programming Approach for Binomial Coefficients

	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Figure: Suppose we want to calculate $C(5,2)$. Start by filling in the first few rows up to $n = 5$. Use the recursive relation: $C(5,2) = C(4,1) + C(4,2)$. Calculate $C(4,1)$ and $C(4,2)$ similarly, filling in the values step-by-step. From the table, $C(5,2) = 10$.

Making Change Problem

- The goal is to devise an algorithm to make change for a given amount N using the minimum number of coins.
- Previously, we considered a greedy approach to this problem.
- However, the greedy algorithm may fail to find an optimal solution in cases where:
  - Some coin denominations are missing.
  - There is a shortage of certain coins.
- For example, with coins of 1, 4, and 6 units, to make change for 8 units:
  - Greedy algorithm uses one 6-unit coin and two 1-unit coins (total 3 coins).
  - Optimal solution uses two 4-unit coins (total 2 coins).
- Dynamic programming can be used to guarantee the optimal solution.

Dynamic Programming Approach for Making Change

- To solve the making change problem with dynamic programming, we use a table to store intermediate results.
- Define $c[i, j]$ as the minimum number of coins needed to make an amount j using the first i denominations.
- Assume:
  - There are n coin denominations, denoted by $d_1, d_2, \ldots, d_n$.
  - We have an unlimited supply of coins for each denomination.
- The table $c[i, j]$ is filled row by row:
  - Start from 0 up to the target amount N.
  - For each denomination, decide whether to include or exclude it in the current amount calculation.

Dynamic Programming Approach for Making Change: Recurrence Relation

Base Cases:

- **Case $j = 0$:**
  - $c[i, 0] = 0$ for all i.
  - Explanation: No coins are needed to make an amount of 0.
- **Case $i = 1$ (Only the 1-unit coin is available):**
  - If $j < d_1$: We cannot make the amount j using only the 1-unit coin, so $c[1, j] = \infty$.
  - If $j \geq d_1$: We can make the amount j using exactly j coins of the 1-unit denomination, so $c[1, j] = j$.

Dynamic Programming Approach for Making Change: Recurrence Relation

General Cases:

- For $c[i, j]$, we have two choices:
  1. **Exclude** the i-th denomination d_i.
  2. **Include** the i-th denomination d_i.

Dynamic Programming Approach for Making Change: Recurrence Relation

Case 1: Excluding the Current Coin:

- If we exclude the i-th denomination (d_i):
  - The minimum number of coins needed to make amount j is the same as when we only consider the first $i-1$ denominations.
  - Thus, we have:
$$c[i,j] = c[i-1,j]$$
- This option is necessary when $j < d_i$, as we cannot use the i-th coin for amount j.

Dynamic Programming Approach for Making Change: Recurrence Relation

Case 2: Including the Current Coin:

- If we include the i-th denomination (d_i):
  - We use one d_i-coin, so we add 1 to our solution.
  - The remaining amount to be made is $j - d_i$.
  - Therefore, the recurrence relation becomes:

$$c[i, j] = 1 + c[i, j - d_i]$$

- This option is only possible if $j \geq d_i$, meaning the current denomination can contribute to the amount.

Dynamic Programming Approach for Making Change: Recurrence Relation

Combining the Two Cases:

- For each entry $c[i, j]$, we want the minimum number of coins needed to make the amount j.
- Thus, we take the minimum of the two choices (including or excluding d_i):

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j - d_i])$$

- This recurrence ensures that we always choose the option with the fewest coins.

Dynamic Programming Approach for Making Change: Recurrence Relation

Complete Recurrence Relation: The complete recurrence relation is:

$$c[i,j] = \begin{cases} \infty & \text{if } i = 1 \text{ and } j < d_i \\ 1 + c[i, j - d_i] & \text{if } i = 1 \text{ and } j \geq d_i \\ c[i-1, j] & \text{if } j < d_i \\ \min(c[i-1, j], 1 + c[i, j - d_i]) & \text{otherwise} \end{cases}$$

- This formula handles all cases for $c[i,j]$:
  - Base cases and initialization.
  - Minimum coin selection by combining choices.

Dynamic Programming Table Setup

- Define $c[i, j]$: Minimum number of coins needed to make amount j using the first i denominations.
- Table dimensions: n rows (one for each denomination) and $N + 1$ columns (for each amount from 0 to N).
- Initialization:
  - $c[i, 0] = 0$ for all i, since no coins are needed to make amount 0.
  - Set $c[i, j] = \infty$ initially for all other values.

Dynamic Programming Approach for Making Change

Step 1: Filling Row 1 (Using Only 1-Unit Coins)

- For $i = 1$ (using only the 1-unit coin):
  - For each amount j from 1 to 8:

$$c[1, j] = j$$

  - This is because we can only use 1-unit coins, so j coins are needed to make amount j.
- The first row of the table becomes:

$$c[1, j] = [0, 1, 2, 3, 4, 5, 6, 7, 8]$$

Dynamic Programming Approach for Making Change

Step 2: Filling Row 2 (Using Coins of 1 and 4 Units)

- For $i = 2$ (using coins of 1 and 4 units):
  - For $j < 4$: We can only use the 1-unit coin, so $c[2, j] = c[1, j]$.
  - For $j \geq 4$: We have two choices:
    1. Exclude the 4-unit coin: Use $c[1, j]$.
    2. Include the 4-unit coin: Use $1 + c[2, j - 4]$.
  - Take the minimum of these two options.
- The second row of the table becomes:

$$c[2, j] = [0, 1, 2, 3, 1, 2, 3, 2, 3]$$

Dynamic Programming Approach for Making Change

Step 2: Filling Row 2 (Using Coins of 1 and 4 Units)

- For $i = 2$, we consider both the 1-unit and 4-unit coins.
- We fill each column j in row 2 as follows:
  - For $j = 0$: No coins are needed to make 0, so $c[2, 0] = 0$.
  - For $j = 1$: Only the 1-unit coin can be used (since $j < 4$), so $c[2, 1] = c[1, 1] = 1$.
  - For $j = 2$: Only the 1-unit coin can be used (since $j < 4$), so $c[2, 2] = c[1, 2] = 2$.
  - For $j = 3$: Only the 1-unit coin can be used (since $j < 4$), so $c[2, 3] = c[1, 3] = 3$.
  - For $j = 4$: We have two choices:
    1. Exclude the 4-unit coin: Use the value from the previous row, $c[1, 4] = 4$.
    2. Include the 4-unit coin: Use $1 + c[2, 0] = 1$ (one 4-unit coin plus the solution for the remaining amount 0).
  - Take the minimum: $c[2, 4] = \min(4, 1) = 1$.

Step 2: Filling Row 2 (Using Coins of 1 and 4 Units)

- For $i = 2$, we consider both the 1-unit and 4-unit coins.
- We fill each column j in row 2 as follows:
  - For $j = 5$: Two choices:
    1. Exclude the 4-unit coin: $c[1, 5] = 5$.
    2. Include the 4-unit coin: $1 + c[2, 1] = 1 + 1 = 2$.
  - Take the minimum: $c[2, 5] = \min(5, 2) = 2$.
  - For $j = 6$: Two choices:
    1. Exclude the 4-unit coin: $c[1, 6] = 6$.
    2. Include the 4-unit coin: $1 + c[2, 2] = 1 + 2 = 3$.
  - Take the minimum: $c[2, 6] = \min(6, 3) = 3$.
  - For $j = 7$: Two choices:
    1. Exclude the 4-unit coin: $c[1, 7] = 7$.
    2. Include the 4-unit coin: $1 + c[2, 3] = 1 + 3 = 4$.
  - Take the minimum: $c[2, 7] = \min(7, 4) = 4$.
  - For $j = 8$: Two choices:
    1. Exclude the 4-unit coin: $c[1, 8] = 8$.
    2. Include the 4-unit coin: $1 + c[2, 4] = 1 + 1 = 2$.
  - Take the minimum: $c[2, 8] = \min(8, 2) = 2$.

Dynamic Programming Approach for Making Change

Step 3: Filling Row 3 (Using Coins of 1, 4, and 6 Units)

- For $i = 3$ (using coins of 1, 4, and 6 units):
  - For $j < 6$: We can only use the denominations 1 and 4, so $c[3, j] = c[2, j]$.
  - For $j \geq 6$: We have two choices:
    1. Exclude the 6-unit coin: Use $c[2, j]$.
    2. Include the 6-unit coin: Use $1 + c[3, j - 6]$.
  - Take the minimum of these two options.
- The third row of the table becomes:

$$c[3, j] = [0, 1, 2, 3, 1, 2, 1, 2, 2]$$

Dynamic Programming Approach for Making Change

Final Table

	0	1	2	3	4	5	6	7	8
$d_1 = 1$	0	1	2	3	4	5	6	7	8
$d_2 = 4$	0	1	2	3	1	2	3	2	3
$d_3 = 6$	0	1	2	3	1	2	1	2	2

- The final answer is in the cell $c[3, 8] = 2$.
- This indicates that the minimum number of coins needed to make 8 units is 2.

Dynamic Programming Algorithm for Making Change

Algorithm 1 MakeChangeDP

```
Input: Amount N, denominations d = [d_1, d_2, ..., d_n]
Output: Minimum number of coins needed to make change for N
Initialize a table c[i, j] with size n × (N + 1)
for i = 1 to n do
    for j = 0 to N do
        if i = 1 and j < d[i] then
            c[i, j] ← ∞
        else if i = 1 and j ≥ d[i] then
            c[i, j] ← 1 + c[i, j − d[i]]
        else if j < d[i] then
            c[i, j] ← c[i − 1, j]
        else
            c[i, j] ← min(c[i − 1, j], 1 + c[i, j − d[i]])
        end if
    end for
end for
return c[n, N]
```

Analysis of the Algorithm

- The algorithm fills up an $n \times (N+1)$ table, giving it a time complexity of $\Theta(nN)$.
- For each entry $c[i,j]$, the algorithm makes a constant-time decision.
- By storing results in a table, the algorithm avoids redundant calculations.
- This approach ensures that we get the minimum number of coins needed to make any amount up to N.

Principle of Optimality

- The solution to the making change problem obtained by dynamic programming is straightforward.
- However, it is important to understand that it relies on a fundamental concept called the **principle of optimality**.
- The principle of optimality states:

  > *In an optimal sequence of decisions or choices, each subsequence must also be optimal.*

- This principle often appears natural in dynamic programming problems, but it is crucial to the correctness of the solution.

Applying the Principle of Optimality

- In the making change problem, we calculate $c[i, j]$ as:

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j - d_i])$$

- This calculation assumes:
  - If $c[i, j]$ is the optimal way to make change for j units using coins of denominations d_1 to d_i,
  - Then $c[i-1, j]$ and $c[i, j - d_i]$ must also represent optimal solutions for their respective subproblems.
- In other words, each value in the table represents the optimal solution to the subproblem it addresses.

Principle of Optimality

Optimality in Table Values

- The only value in the table that we are truly interested in is $c[n, N]$ (the minimum coins needed for amount N using all denominations).
- However, for $c[n, N]$ to be optimal, each other entry in the table must also represent optimal choices for their subproblems.
- Thus, we rely on the principle of optimality throughout the entire table.

Principle of Optimality

When the Principle of Optimality Does Not Apply

- Although the principle of optimality may appear obvious, it does not apply to every problem.
- When the principle of optimality does not hold:
    - It may not be possible to solve the problem using dynamic programming.
- For example, if a problem concerns the optimal use of limited resources:
    - The optimal solution to an instance might not be achievable by combining optimal solutions of subproblems.

Example: Shortest Route Problem

- Consider the shortest route from Montreal to Toronto via Kingston.
- If the shortest route from Montreal to Toronto passes through Kingston:
  - Then the segment from Montreal to Kingston must be the shortest possible route.
  - Similarly, the segment from Kingston to Toronto must also be the shortest.
- In this case, the principle of optimality applies.

Principle of Optimality

When the Principle Fails in Shortest Route Problems

- Suppose the fastest way from Montreal to Toronto passes through Kingston:
  - It does not necessarily follow that it's best to drive as fast as possible from Montreal to Kingston and then from Kingston to Toronto.
  - For example, if we use too much fuel on the first half, we may need to refill later, losing time overall.
- Here, the sub-trips from Montreal to Kingston and from Kingston to Toronto are **not independent**:
  - They share resources (fuel).
  - An optimal solution for one part may prevent an optimal solution for the other.
- In this case, the principle of optimality does not apply.

Principle of Optimality

Why the Principle Often Applies in Dynamic Programming

- Despite some exceptions, the principle of optimality applies more often than not.
- Restatement of the principle:

  > *The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to* **some** *of its subproblems.*

- The difficulty in applying this principle lies in identifying which subproblems are relevant to the solution.

Example of Relevance in Shortest Route Problems

- In finding the shortest route from Montreal to Toronto, we may not need the shortest route from Montreal to Ottawa if it is not on the path.
- Dynamic programming avoids calculating irrelevant subproblems:
  - It calculates solutions only for subproblems relevant to the main problem.
- This is one of the strengths of dynamic programming.