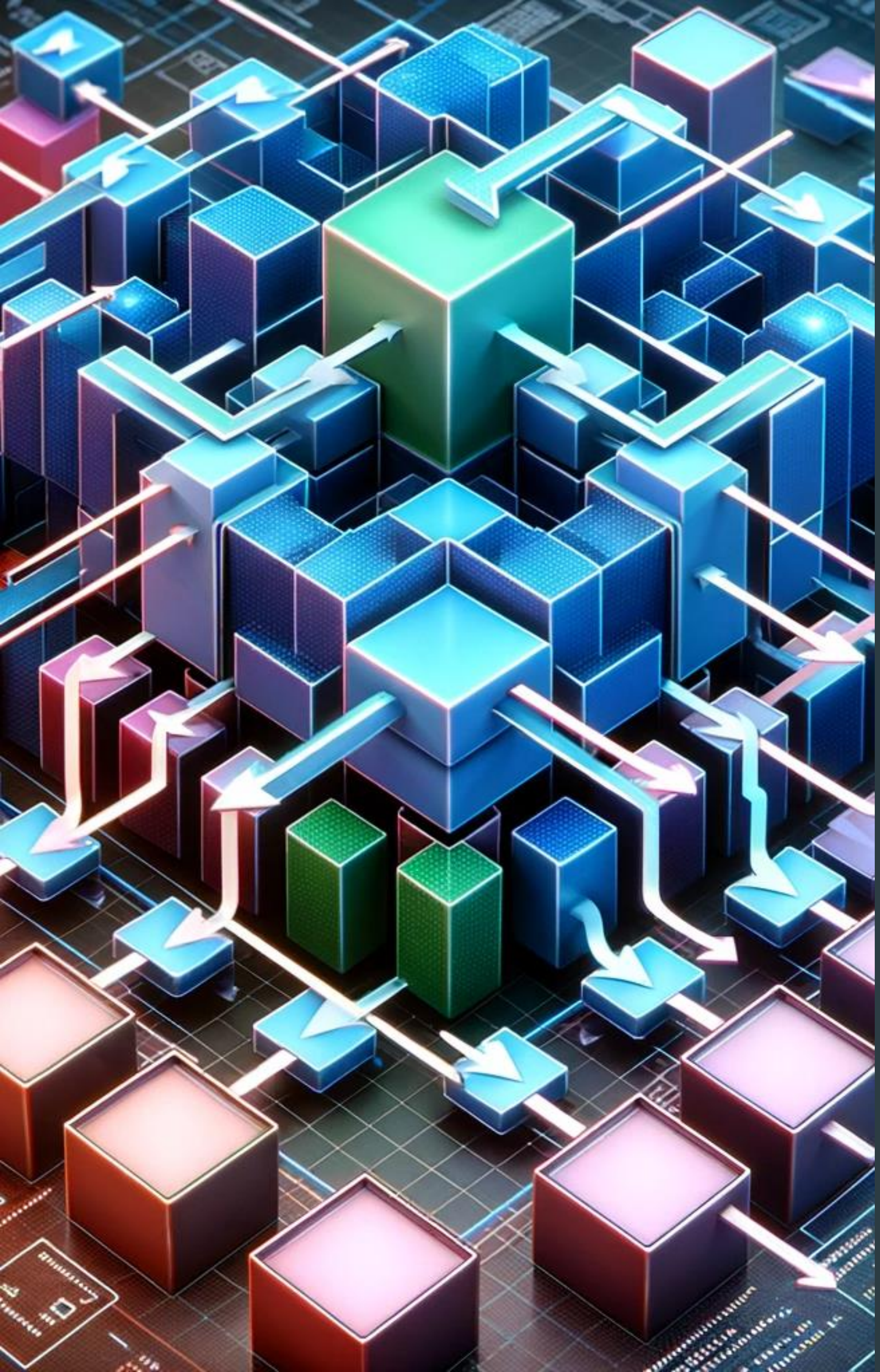


ECE2810J
Data Structures and Algorithms

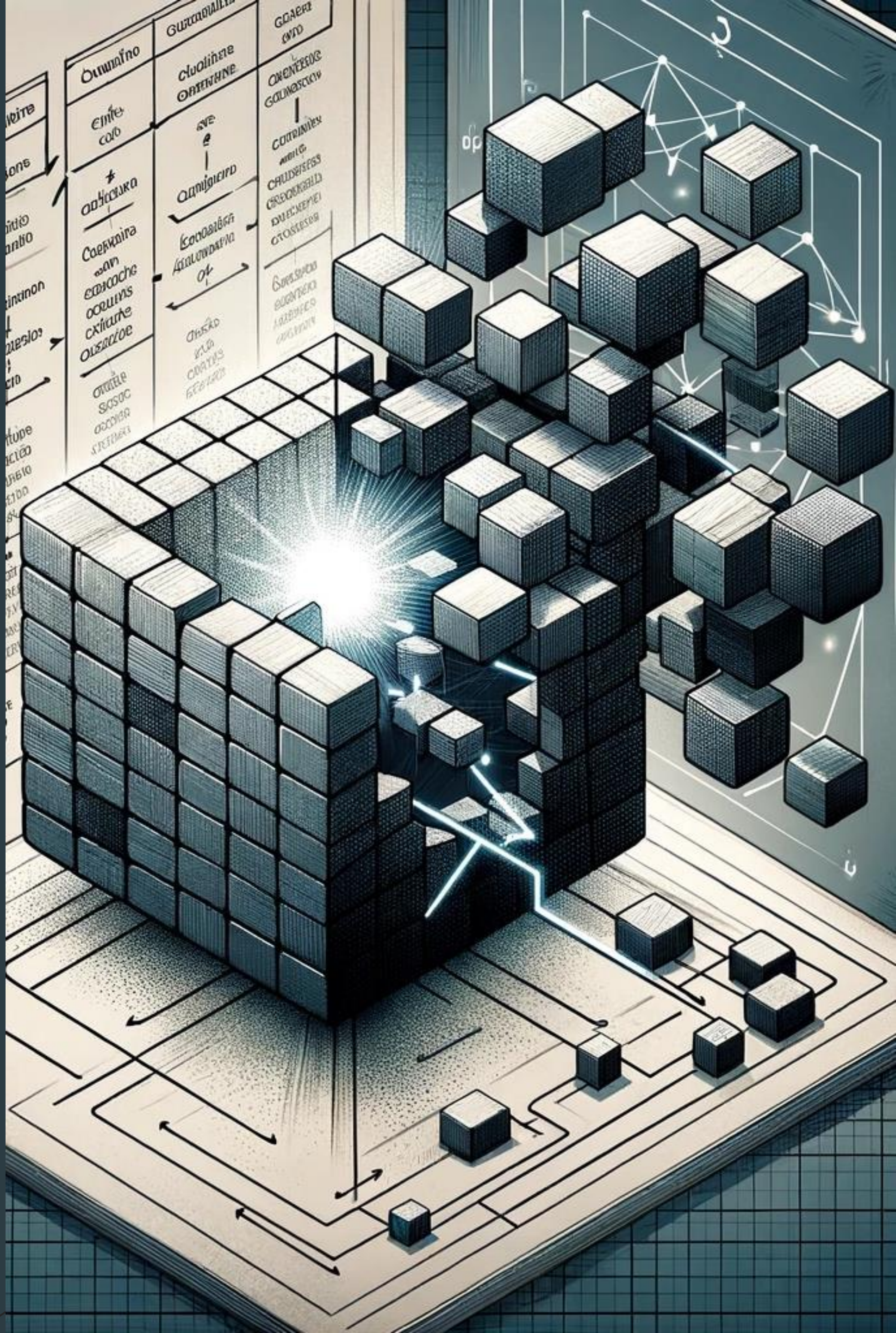
Dynamic Programming

Learning Objectives:

- Understand the basic idea of dynamic programming
- Know under what situation dynamic programming could be applied

Outline

- Dynamic Programming
  - Example 1: Fibonacci sequence
  - Example 2: 0/1 Knapsack problem
  - Exercise
  - Example 3: Matrix-Chain Multiplication
  - Summary



Limitation of Divide and Conquer

- Recursively solving subproblems can result in the same computations being repeated when the subproblems overlap.

- For example: computing the Fibonacci sequence

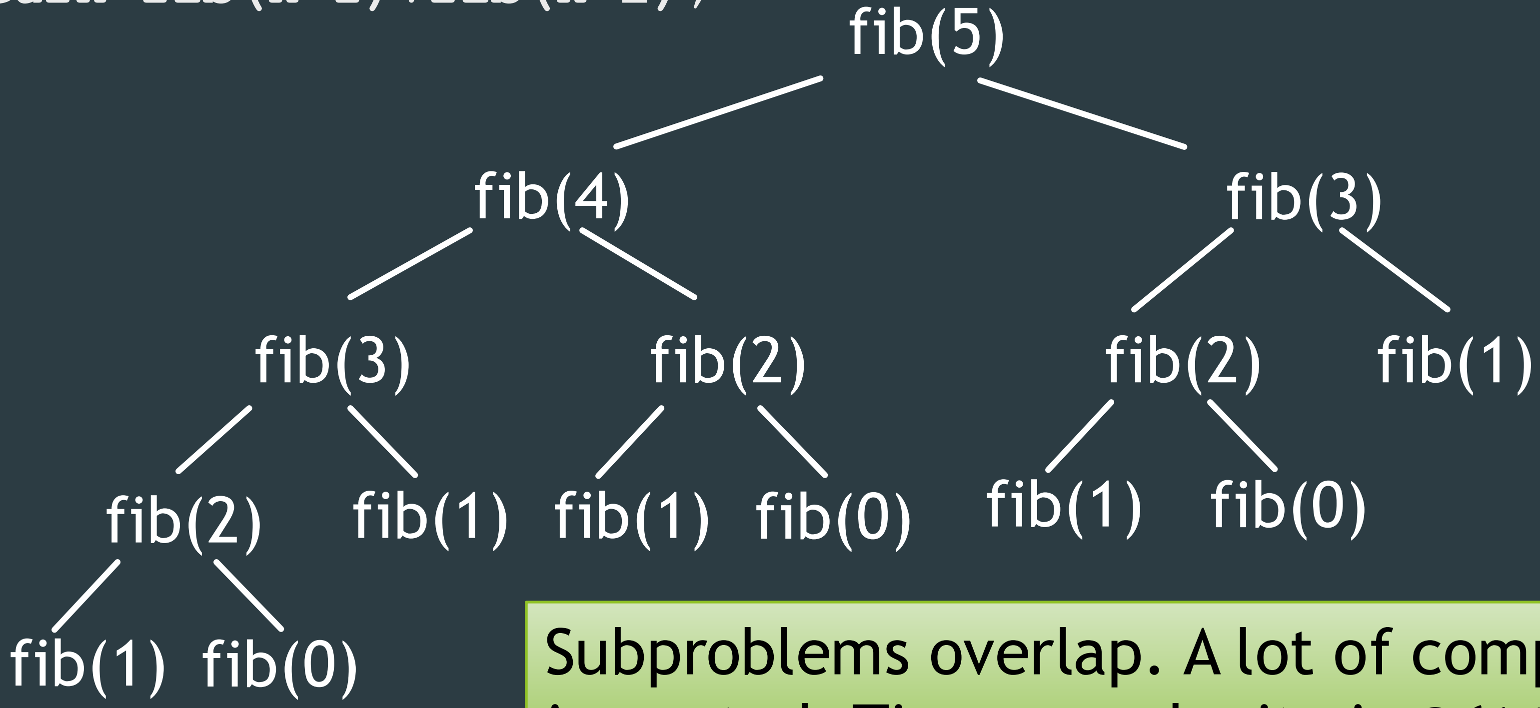
$$f_0 = 0;\ \ f_1 = 1;\ \ f_n = f_{n-1} + f_{n-2}, n \geq 2$$

- Divide and conquer approach:

```
int fib(int n) {
  if(n <= 1) return n;
  return fib(n-1)+fib(n-2);
}
```

Fibonacci Sequence: Divide and Conquer Solution

```
int fib(int n) {
  if(n <= 1) return n;
  return fib(n-1)+fib(n-2);
}
```



fib(5)

fib(4) fib(3)

fib(3) fib(2) fib(2) fib(1)

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

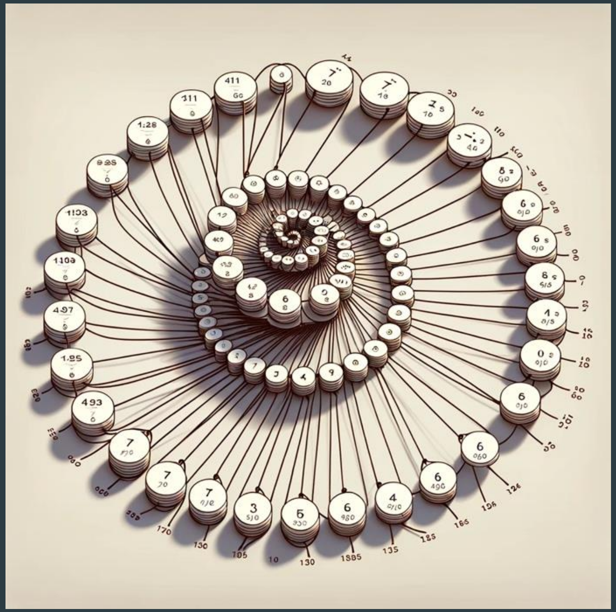
fib(1) fib(0)

Subproblems overlap. A lot of computation is wasted. Time complexity is $\Omega(1.5^n)$.

Fibonacci Sequence Iterative Solution

- We can also compute the Fibonacci sequence in iterative way:

```
int fib(int n) {
  f[0] = 0; f[1] = 1;
  for(i = 2 to n)
    f[i] = f[i-1]+f[i-2];
  return f[n];
}
```



- Time complexity is $\Theta(n)$.

Dynamic Programming

- Used when a problem can be divided into subproblems that overlap.
  - Solve each subproblem once and store the solution in a table.
  - If a subproblem is encountered again, simply look up its solution in the table.
  - Reconstruct the solution to the original problem from the solutions to the subproblems.
- The more overlap the better, as this reduces the number of subproblems.
- Dynamic programming can be applied to solve optimization problem.

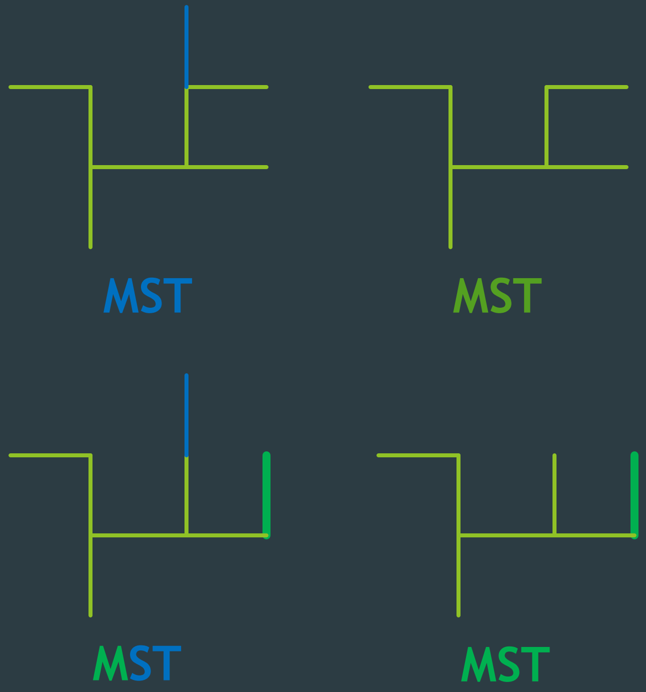
Optimization Problem

- Many problems we encounter are optimization problems:
  - A problem in which some function (called the **objective function**) is to be optimized (usually minimized or maximized) subject to some **constraints**.

- The solutions that satisfy the constraints are called **feasible solutions**.
- The number of feasible solutions is typically very large.
- We obtain the optimal solution by searching the feasible solution space.

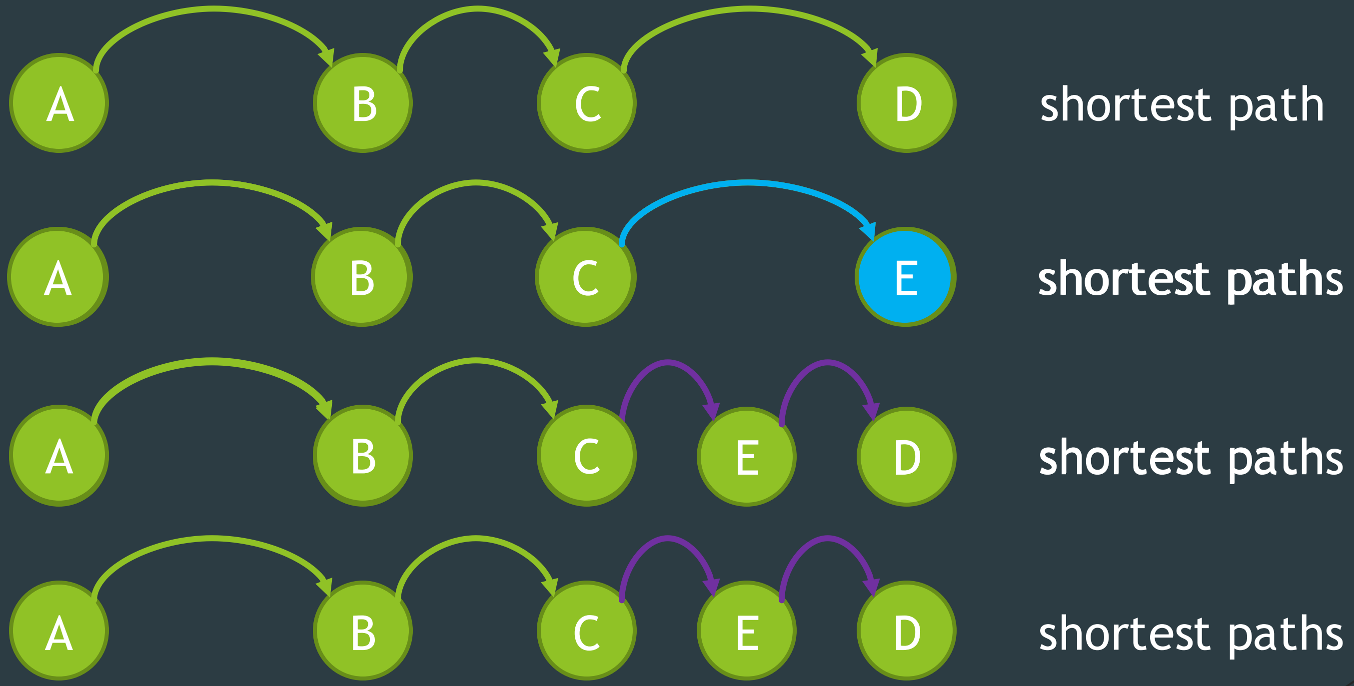
Optimization Problem Example

- Minimum spanning tree.
  - Objective function: the sum of all edge weights.
  - Constraints: a subgraph of a MST must also be MSTs.



Optimization Problem Example

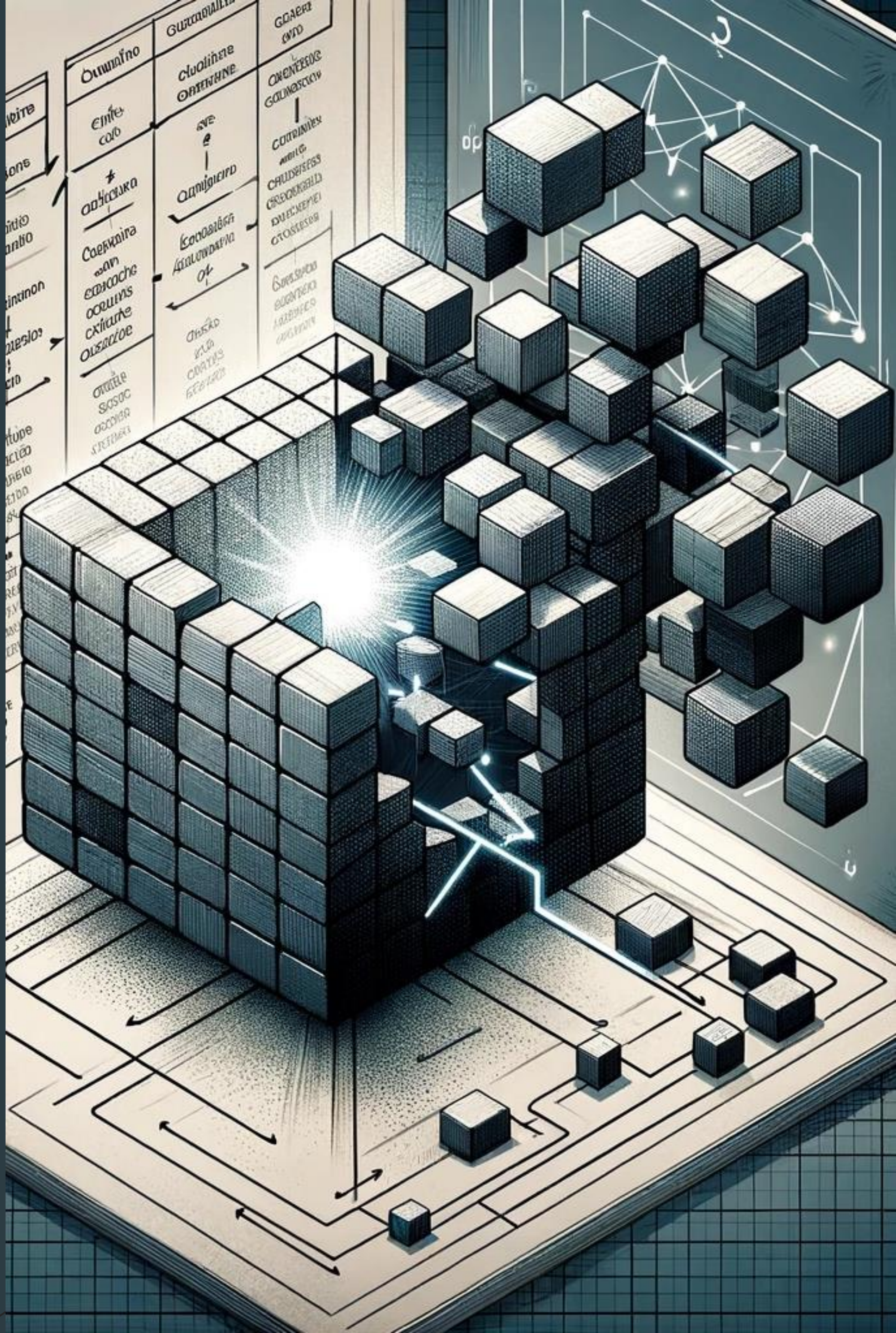
- Shortest path.
  - Objective function: the sum of all edge weights.
  - Constraints: a subgraph of a shortest path must also be shortest paths.



Takeaway: Dynamic Programming is often linked with Induction!
Book-keep partial results to avoid redundant computation!

Outline

- Dynamic Programming
  - Example 1: Fibonacci sequence
  - Example 2: 0/1 Knapsack problem
  - Exercise
  - Example 3: Matrix-Chain Multiplication
  - Summary



0/1 Knapsack problem

0/1 Knapsack problem

Item 1
Weight = 3kg
Value = $2

Item 2
Weight = 1kg
Value = $2

Item 5
Weight = 3kg
Value = $2

Max Capacity: 7kg

Item 3
Weight = 3kg
Value = $4

Item 4
Weight = 4kg
Value = $5

0/1 Knapsack problem

Weights →

Items Included ↓

		0	1	2	3	4	5	6	7
Empty	0								
Item 1 Weight = 3kg Value = $2	1								
Item 2 Weight = 1kg Value = $2	2				?				
Item 3 Weight = 3kg Value = $4	3								
Item 4 Weight = 4kg Value = $5	4								
Item 5 Weight = 2kg Value = $3	5								

Note: Row 2 means empty, item 1, item 2 can be included
Row 3 means empty, item 1, item 2 and item 3 can be included

0/1 Knapsack problem

Weights →

Items Included ↓

Empty
Item 1 Weight = 3kg Value = $2
Item 2 Weight = 1kg Value = $2
Item 3 Weight = 3kg Value = $4
Item 4 Weight = 4kg Value = $5
Item 5 Weight = 2kg Value = $3

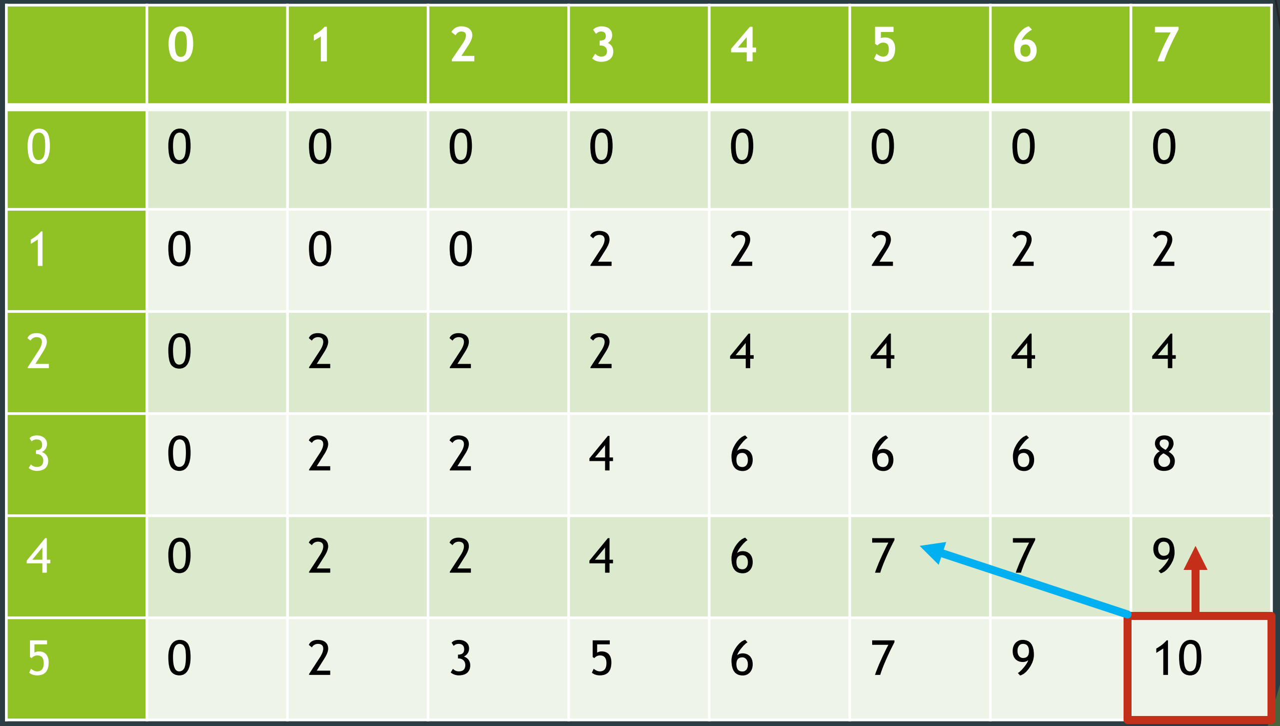
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2
2	0	2	2	2	4	4	4	4
3	0	2	2	4	6	6	6	8
4	0	2	2	4	6	7	7	9
5	0	2	3	5	6	7	9	10

Note: Row 2 means empty, item 1, item 2 can be included
Row 3 means empty, item 1, item 2 and item 3 can be included

0/1 Knapsack problem

Weights

Items Included



		0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0	0
Item 1 Weight = 3kg Value = $2	1	0	0	0	2	2	2	2	2
Item 2 Weight = 1kg Value = $2	2	0	2	2	2	4	4	4	4
Item 3 Weight = 3kg Value = $4	3	0	2	2	4	6	6	6	8
Item 4 Weight = 4kg Value = $5	4	0	2	2	4	6	7	7	9
Item 5 Weight = 2kg Value = $3	5	0	2	3	5	6	7	9	10

Backtrace solution:

(1) Compare with the row above

(2) Subtract the weight of current item

0/1 Knapsack problem

Weights

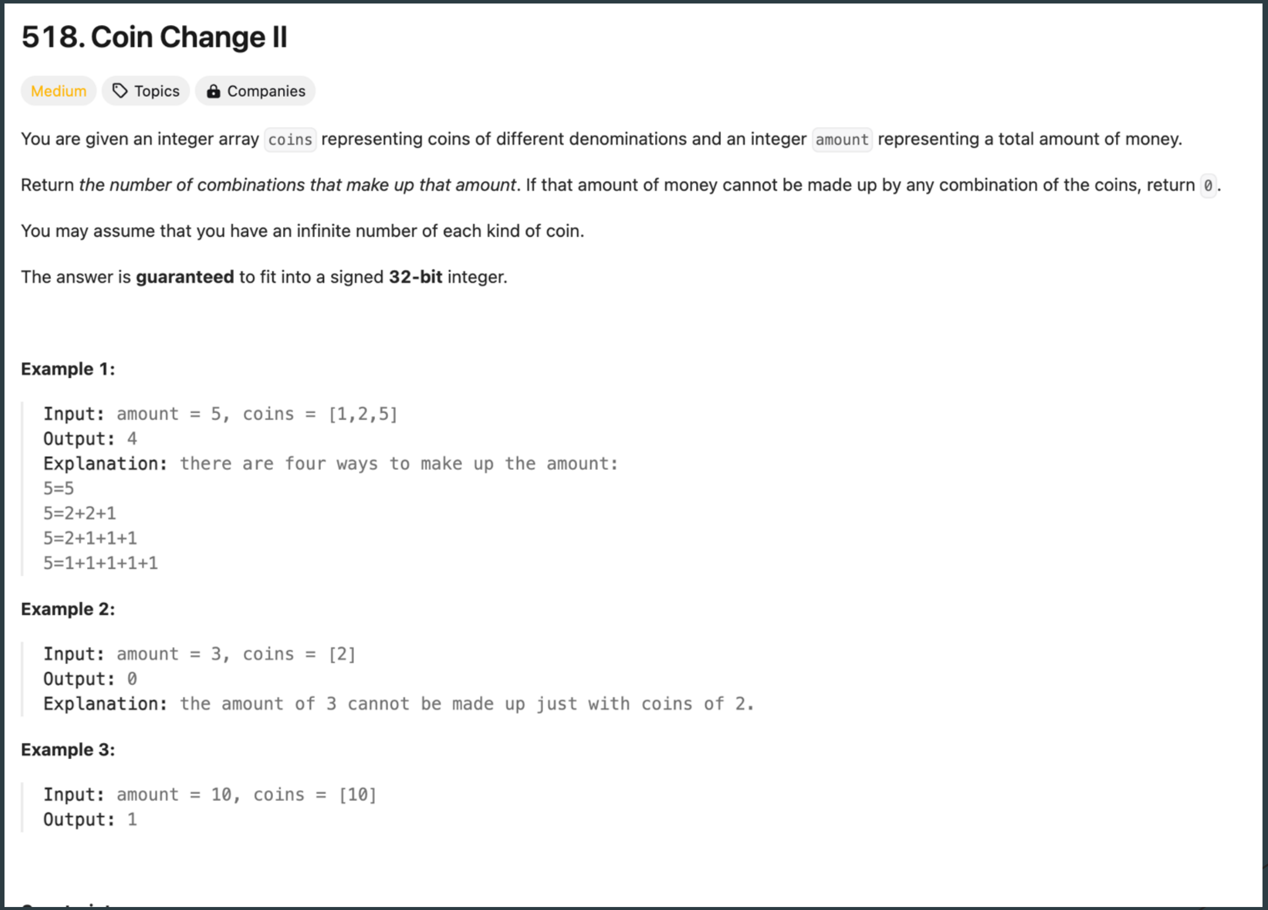
Items Included

		0	1	2	3	4	5	6	7
Empty	0	0	0	0	0	0	0	0	0
Item 1 Weight = 3kg Value = $2	1	0	0	0	2	2	2	2	2
Item 2 Weight = 1kg Value = $2	2	0	2	2	2	4	4	4	4
Item 3 Weight = 3kg Value = $4	3	0	2	2	4	6	6	6	8
Item 4 Weight = 4kg Value = $5	4	0	2	2	4	6	7	7	9
Item 5 Weight = 2kg Value = $3	5	0	2	3	5	6	7	9	10

Final Item pick: Item 2, Item 4, and item 5

Exercise 1

LeetCode: Problem 518



518. Coin Change II

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the number of combinations that make up that amount*. If that amount of money cannot be made up by any combination of the coins, return `0`.

You may assume that you have an infinite number of each kind of coin.

The answer is **guaranteed** to fit into a signed **32-bit** integer.

Example 1:

```
Input: amount = 5, coins = [1,2,5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

Example 2:

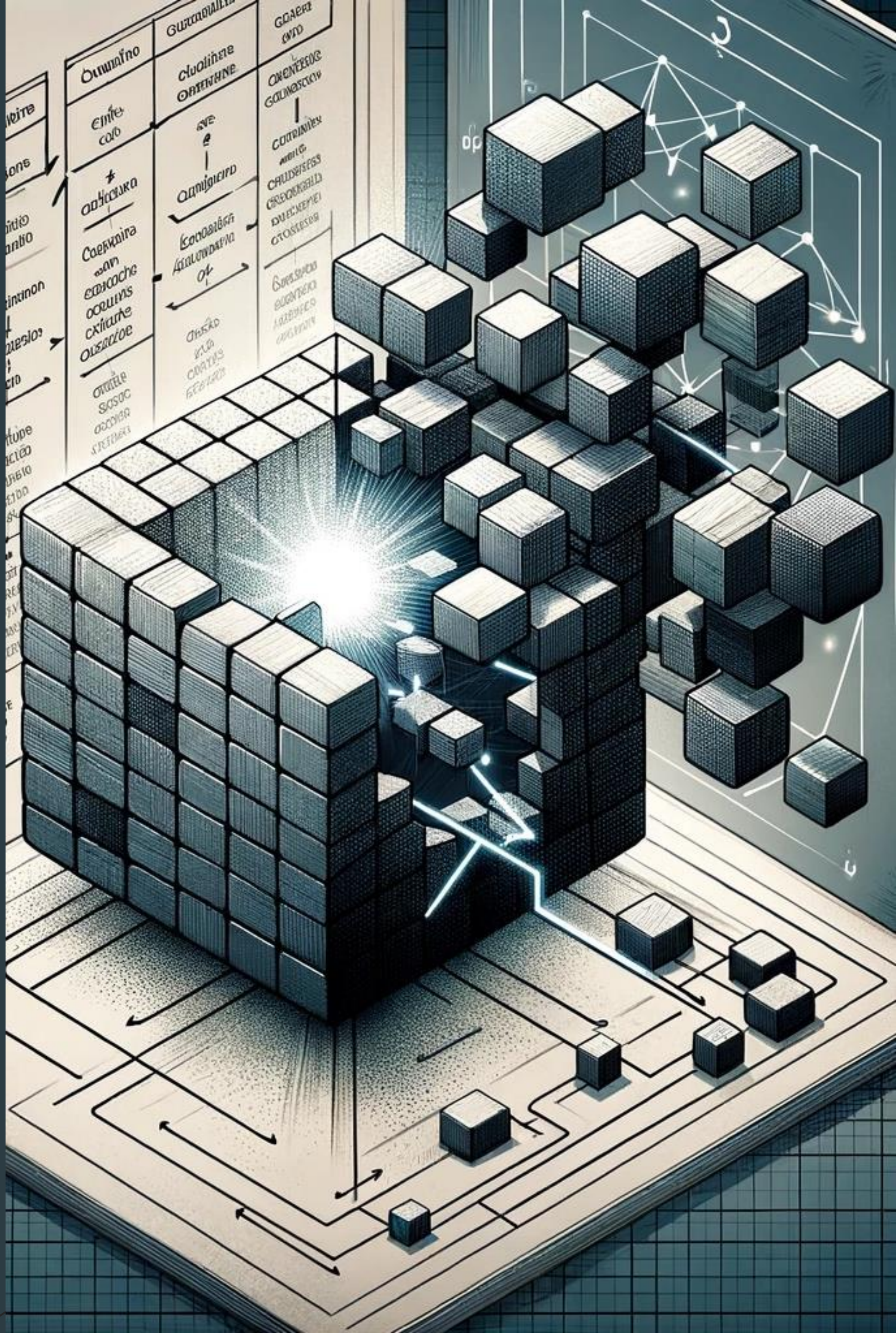
```
Input: amount = 3, coins = [2]
Output: 0
Explanation: the amount of 3 cannot be made up just with coins of 2.
```

Example 3:

```
Input: amount = 10, coins = [10]
Output: 1
```

Outline

- Dynamic Programming
  - Example 1: Fibonacci sequence
  - Example 2: 0/1 Knapsack problem
  - Exercise
  - Example 3: Matrix-Chain Multiplication
  - Summary



Matrix-Chain Multiplication

- What is the cost of multiplying two matrices A and B?
  - Suppose A is a $p \times q$ matrix and B is a $q \times r$ matrix.
  - Since the time to compute $C = AB$ is dominated by the number of **scalar multiplications**, we use the number of scalar multiplications as the complexity measure.

- $C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}$.
  - We need q scalar multiplications to calculate C_{ij}.
  - C is of size $p \times r$.
- The number of scalar multiplications is pqr. (IMPORTANT!)

Matrix-Chain Multiplication

- Now how would you compute the multiplication of three matrices $A \times B \times C$?
    - Suppose A is of size 100×1, B is of size 1×100, and C is of size 100×1.

- If we multiply as $(A \times B) \times C$, the number of scalar multiplications is 20000.
- If we multiply as $A \times (B \times C)$, the number of scalar multiplications is 200.

Matrix-Chain Multiplication

- If we want to multiply a chain of matrices $A_1 \times A_2 \times \cdots \times A_n$, where A_i is of size $p_{i-1} \times p_i$, what is the best order of multiplication to minimize the number of scalar multiplications?

- This is an optimization problem.
- It can be proved that number of different orders on n matrices is $\Omega(4^n/n^{1.5})$.
- Instead of enumerating all of the orders, can we do better to solve the optimization problem?

Matrix-Chain Multiplication

- For simplicity, define the problem of finding the optimal order to multiply $A_i \times A_{i+1} \times \cdots \times A_j$ as $Q_{i\sim j}$. The minimal number of scalar multiplications is $m_{i\sim j}$.
  - We ultimately want to solve $Q_{1\sim n}$.

Matrix-Chain Multiplication

- Suppose in the optimal order for $A_i \times \cdots \times A_j$, the last multiplication is $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$.

- Then the order of computing $A_i \times \cdots \times A_k$ in the optimal order of computing $A_i \times \cdots \times A_j$ must be an optimal order to compute $A_i \times \cdots \times A_k$.
  - Why?
    - If not, then we copy and paste the better order ➔ we have a better order for computing $A_i \times \cdots \times A_j$!
    - Similar conclusion for computing $A_{k+1} \times \cdots \times A_j$.

- If we know k, we can divide the problem $Q_{i\sim j}$ into two smaller instances: $Q_{i\sim k}$ and $Q_{(k+1)\sim j}$.

Matrix-Chain Multiplication

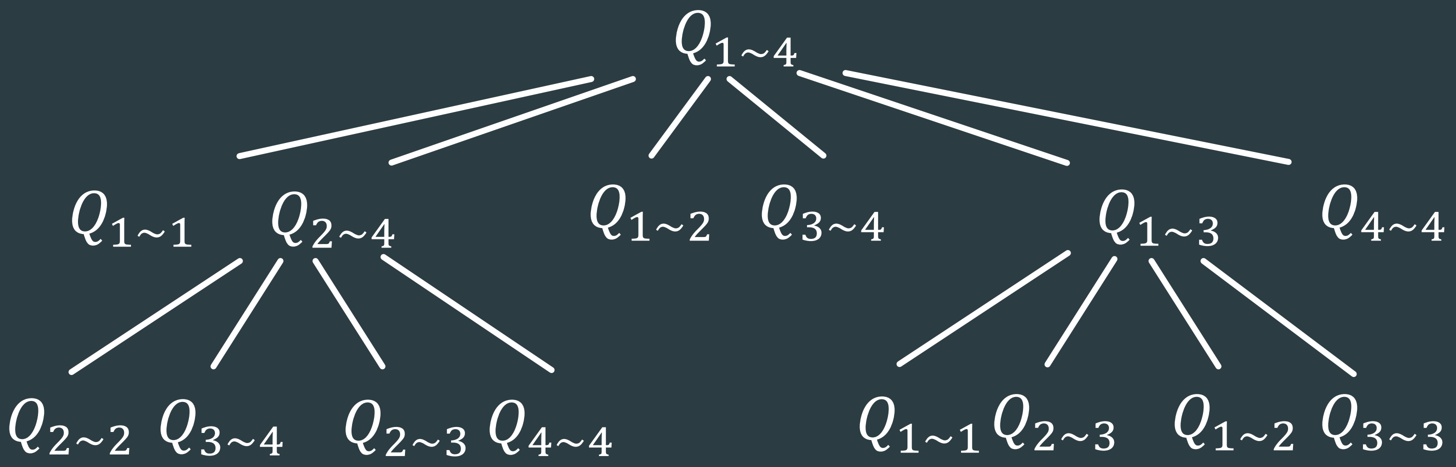
- Assume we have known the minimum number of scalar multiplications for $Q_{i\sim k}$ and $Q_{(k+1)\sim j}$ as $m_{i\sim k}$ and $m_{(k+1)\sim j}$.
  - Then $m_{i\sim j} = m_{i\sim k} + m_{(k+1)\sim j} + p_{i-1}p_k p_j$.

- However, we don't know k! We need to consider all possible divisions, i.e., all $i \leq k \leq j-1$.

- Thus, in order to solve $Q_{i\sim j}$, we need to consider all subproblems $Q_{i\sim k}$ and $Q_{(k+1)\sim j}$, for all $i \leq k \leq j-1$.
  - $m_{i\sim j} = \min_{i\leq k\leq j-1}(m_{i\sim k} + m_{(k+1)\sim j} + p_{i-1}p_k p_j)$

Matrix-Chain Multiplication

- In summary, we can divide the problem into subproblems of the same form.



Many subproblems are overlapped.

Matrix-Chain Multiplication

- The straightforward recursive algorithm has exponential time complexity.
  - However, it will encounter each subproblem many times in different branches of the tree.

- The total number of different subproblems is not exponential.
  - They are $Q_{i\sim j}$, for $1 \leq i \leq j \leq n$.
  - The total number is $n(n+1)/2$.
- Instead, we use a tabular, bottom-up approach.

Matrix-Chain Multiplication
Bottom-up Approach

- Apply the recursive relation:

$$m_{i\sim j} = \min_{i \le k \le j-1} \left(m_{i\sim k} + m_{(k+1)\sim j} + p_{i-1}p_k p_j\right)$$

- Initial situation $m_{1\sim 1} = m_{2\sim 2} = \cdots = m_{n\sim n} = 0$.
- In the first round, we compute $m_{1\sim 2}, m_{2\sim 3}, \dots, m_{(n-1)\sim n}$.
- In the second round, we compute $m_{1\sim 3}, m_{2\sim 4}, \dots, m_{(n-2)\sim n}$.
- So on and so forth. In the l-th round, we compute $m_{1\sim (l+1)}, m_{2\sim (l+2)}, \dots, m_{(n-l)\sim n}$.
- Finally, we compute $m_{1\sim n}$.
- To obtain the multiplication order, we also record the partition k which gives the minimal $m_{i\sim j}$ as $s_{i\sim j}$.

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

	m_{ij}		j		
		1	2	3	4
	1	0			
i	2	–	0		
	3	–	–	0	
	4	–	–	–	0

	k_{ij}		j		
		1	2	3	4
	1	–			
i	2	–	–		
	3	–	–	–	
	4	–	–	–	–

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

m_{ij}

j

i	1	2	3	4
1	0			
2	–	0		
3	–	–	0	
4	–	–	–	0

k_{ij}

j

i	1	2	3	4
1	–			
2	–	–		
3	–	–	–	
4	–	–	–	–

$$m_{i\sim(i+1)} = m_{i\sim i} + m_{(i+1)\sim(i+1)} + p_{i-1}p_ip_{i+1}$$

$$= p_{i-1}p_ip_{i+1}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

j

i

m_{ij}	1	2	3	4
1	0	100		
2	–	0	10	
3	–	–	0	200
4	–	–	–	0

j

i

k_{ij}	1	2	3	4
1	–	1		
2	–	–	2	
3	–	–	–	3
4	–	–	–	–

$$\begin{aligned} m_{i\sim(i+1)} &= m_{i\sim i} + m_{(i+1)\sim(i+1)} + p_{i-1}p_ip_{i+1} \\ &= p_{i-1}p_ip_{i+1} \end{aligned}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

	m_{ij}		j		
		1	2	3	4
	1	0	100		
i	2	–	0	10	
	3	–	–	0	200
	4	–	–	–	0

	k_{ij}		j		
		1	2	3	4
	1	–	1		
i	2	–	–	2	
	3	–	–	–	3
	4	–	–	–	–

$$m_{i\sim(i+2)} = \min\{m_{i\sim i} + m_{(i+1)\sim(i+2)} + p_{i-1}p_ip_{i+2},$$
$$m_{i\sim(i+1)} + m_{(i+2)\sim(i+2)} + p_{i-1}p_{i+1}p_{i+2}\}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

j

i

m_{ij}	1	2	3	4
1	0	100		
2	–	0	10	
3	–	–	0	200
4	–	–	–	0

j

i

k_{ij}	1	2	3	4
1	–	1		
2	–	–	2	
3	–	–	–	3
4	–	–	–	–

$$m_{1\sim3} = \min\{m_{1\sim1} + m_{2\sim3} + p_0p_1p_3,$$

$$m_{1\sim2} + m_{3\sim3} + p_0p_2p_3\} = \min\{20, 200\}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

m_{ij}

j

i	1	2	3	4
1	0	100	20	
2	–	0	10	
3	–	–	0	200
4	–	–	–	0

k_{ij}

j

i	1	2	3	4
1	–	1	1	
2	–	–	2	
3	–	–	–	3
4	–	–	–	–

$$m_{1\sim3} = \min\{m_{1\sim1} + m_{2\sim3} + p_0p_1p_3,$$
$$m_{1\sim2} + m_{3\sim3} + p_0p_2p_3\} = \min\{20, 200\}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

m_{ij} j

i	1	2	3	4
1	0	100	20	
2	–	0	10	
3	–	–	0	200
4	–	–	–	0

k_{ij} j

i	1	2	3	4
1	–	1	1	
2	–	–	2	
3	–	–	–	3
4	–	–	–	–

$$m_{2\sim4} = \min\{m_{2\sim2} + m_{3\sim4} + p_1p_2p_4,\\ m_{2\sim3} + m_{4\sim4} + p_1p_3p_4\} = \min\{400, 30\}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

j

	m_{ij}	1	2	3	4
	1	0	100	20	
i	2	–	0	10	30
	3	–	–	0	200
	4	–	–	–	0

j

	k_{ij}	1	2	3	4
	1	–	1	1	
i	2	–	–	2	3
	3	–	–	–	3
	4	–	–	–	–

$$m_{2\sim4} = \min\{m_{2\sim2} + m_{3\sim4} + p_1p_2p_4,\ m_{2\sim3} + m_{4\sim4} + p_1p_3p_4\} = \min\{400, 30\}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

m_{ij} j

i

	1	2	3	4
1	0	100	20	
2	–	0	10	30
3	–	–	0	200
4	–	–	–	0

k_{ij} j

i

	1	2	3	4
1	–	1	1	
2	–	–	2	3
3	–	–	–	3
4	–	–	–	–

$$m_{i\sim(i+3)} = \min_{i \le k \le i+2} \left(m_{i\sim k} + m_{(k+1)\sim(i+3)} + p_{i-1}p_k p_{(i+3)}\right)$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

j

	m_{ij}	1	2	3	4
	1	0	100	20	
i	2	–	0	10	30
	3	–	–	0	200
	4	–	–	–	0

j

	k_{ij}	1	2	3	4
	1	–	1	1	
i	2	–	–	2	3
	3	–	–	–	3
	4	–	–	–	–

$$m_{1\sim 4} = \min_{1 \le k \le 3} \left(m_{1\sim k} + m_{(k+1)\sim 4} + p_0 p_k p_4 \right)$$
$$= \min\{230, 2300, 220\}$$

Matrix-Chain Multiplication Example

- $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
- $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

j

	m_{ij}	1	2	3	4
	1	0	100	20	220
i	2	–	0	10	30
	3	–	–	0	200
	4	–	–	–	0

Optimal Value

j

	k_{ij}	1	2	3	4
	1	–	1	1	3
i	2	–	–	2	3
	3	–	–	–	3
	4	–	–	–	–

$$m_{1\sim4} = \min_{1\le k\le3}\left(m_{1\sim k} + m_{(k+1)\sim4} + p_0p_kp_4\right)$$

$$= \min\{230, 2300, 220\}$$

Matrix-Chain Multiplication
Constructing an Optimal Order

- We can construct an optimal order based on the records s_{ij}.

```
Print_Order(s, i, j) {
  if(i == j) cout << "A_i";
  else {
    cout << "(";
    Print_Order(s, i, s_ij);
    cout << "*";
    Print_Order(s, s_ij+1, j);
    cout << ")";
  }
}
```

- Initial call is `Print_Order(s, 1, n);`

Matrix-Chain Multiplication Example

- Construct an optimal order
  - $n = 4,\ A_1 \times A_2 \times A_3 \times A_4.$
  - $p_0 = 10, p_1 = 1, p_2 = 10, p_3 = 1, p_4 = 20.$

j

	k_{ij}	1	2	3	4
	1	–	1	1	3
i	2	–	–	2	3
	3	–	–	–	3
	4	–	–	–	–

$k_{14} = 3$ ➡ $A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times A_2 \times A_3) \times A_4$

$k_{13} = 1$ ➡ $A_1 \times A_2 \times A_3 = A_1 \times (A_2 \times A_3)$

$k_{23} = 2$ ➡ $A_2 \times A_3 = A_2 \times A_3$

$$A_1 \times A_2 \times A_3 \times A_4 = (A_1 \times (A_2 \times A_3)) \times A_4$$

Matrix-Chain Multiplication
Time Complexity

- Get the minimum number of scalar multiplications:
  - We need to obtain all m_{ij} and s_{ij}, for $1 \leq i \leq j \leq n$.
    - $O(n^2)$ records
    - Each $m_{i\sim j}$ is the minimum of $O(n)$ terms.
  - Total time complexity is $O(n^3)$.

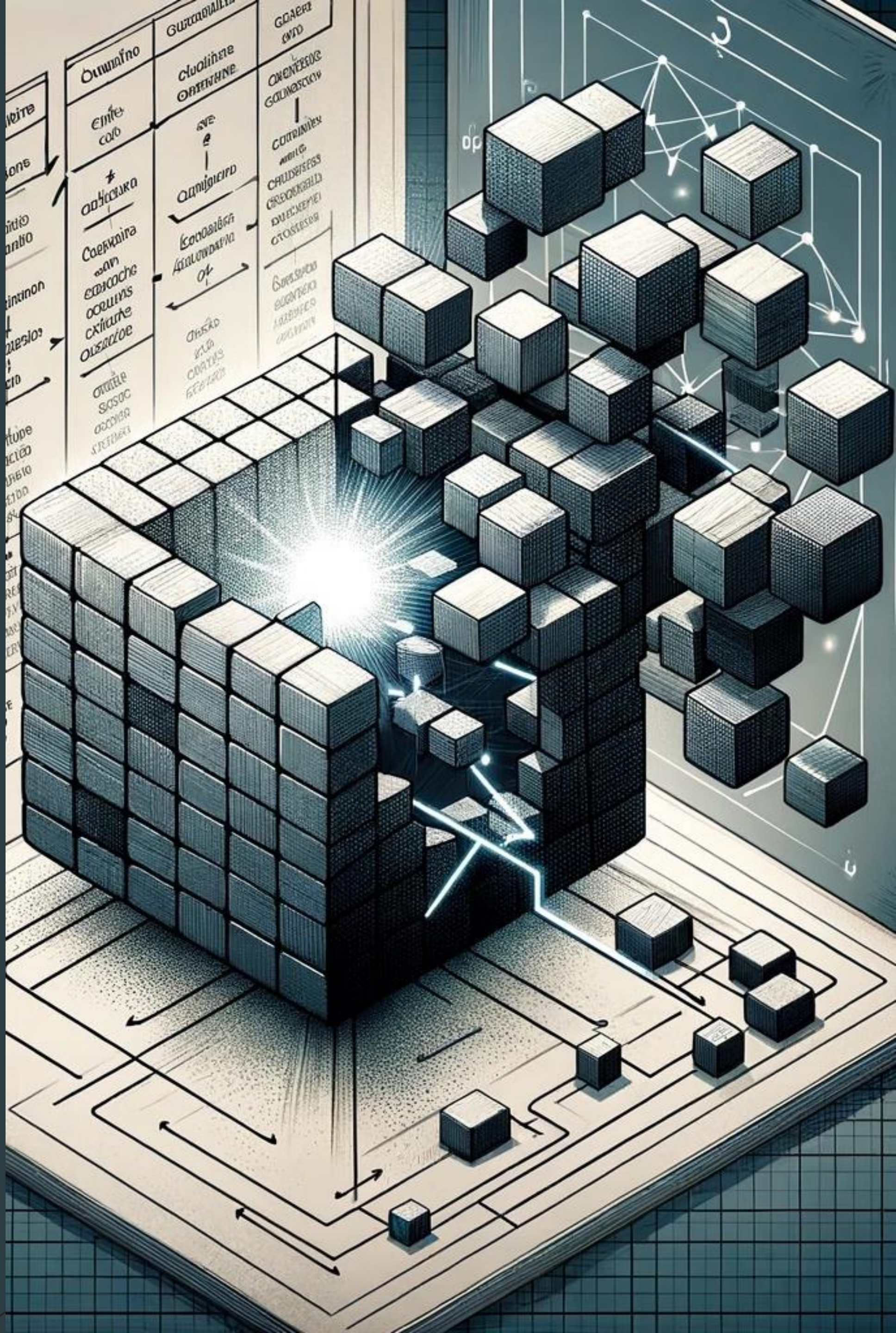
- Obtain the optimal order:
  - $O(n)$

Matrix-Chain Multiplication Summary

- Matrix-chain multiplication is an optimization problem.
- The solution is based on dynamic programming.
  - The original problem can be divided into same subproblems that **overlap**.
  - Each subproblem is solved once and stored in a table.
  - If a subproblem is encountered again, simply look up its solution in the table.
  - Reconstruct the solution to the original problem from the solutions to the subproblems.

Outline

- Dynamic Programming
  - Example 1: Fibonacci sequence
  - Example 2: 0/1 Knapsack problem
  - Exercise
  - Example 3: Matrix-Chain Multiplication
  - Summary



Dynamic Programming for Optimization

- There are two key ingredients that an optimization problem must have in order for dynamic programming to apply:
    - Optimal substructure;
    - Overlapping subproblems.

Optimal Substructure

- An optimal solution to the problem contains within it optimal solutions to subproblems.
  - In matrix-chain multiplication, the optimal order on calculating $A_i \times \cdots \times A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problem of ordering $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$.

- You can show optimal substructure property by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction.

Overlapping Subproblems

- A recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems.
  - E.g., subproblems of matrix-chain multiplication overlap.
  - In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion.
- Dynamic-programming algorithms take advantage of overlapping subproblems by
  - solving each subproblem once …
  - … and then storing the solution in a table where it can be looked up when needed.

Designing a Dynamic-Programming Algorithm

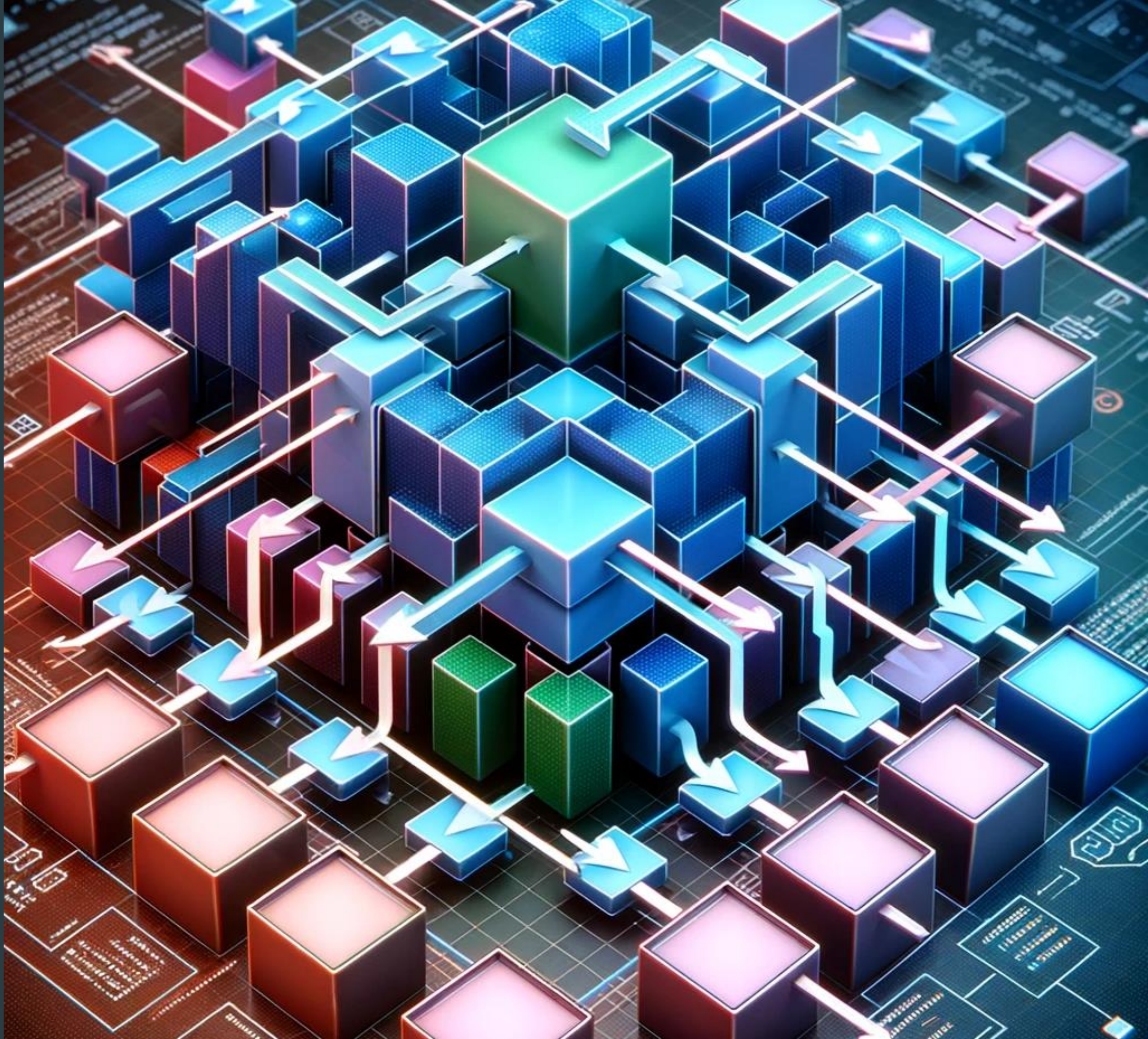
1. Characterize the structure of an optimal solution.

   Usually, we need to define a general problem.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Dynamic Programming

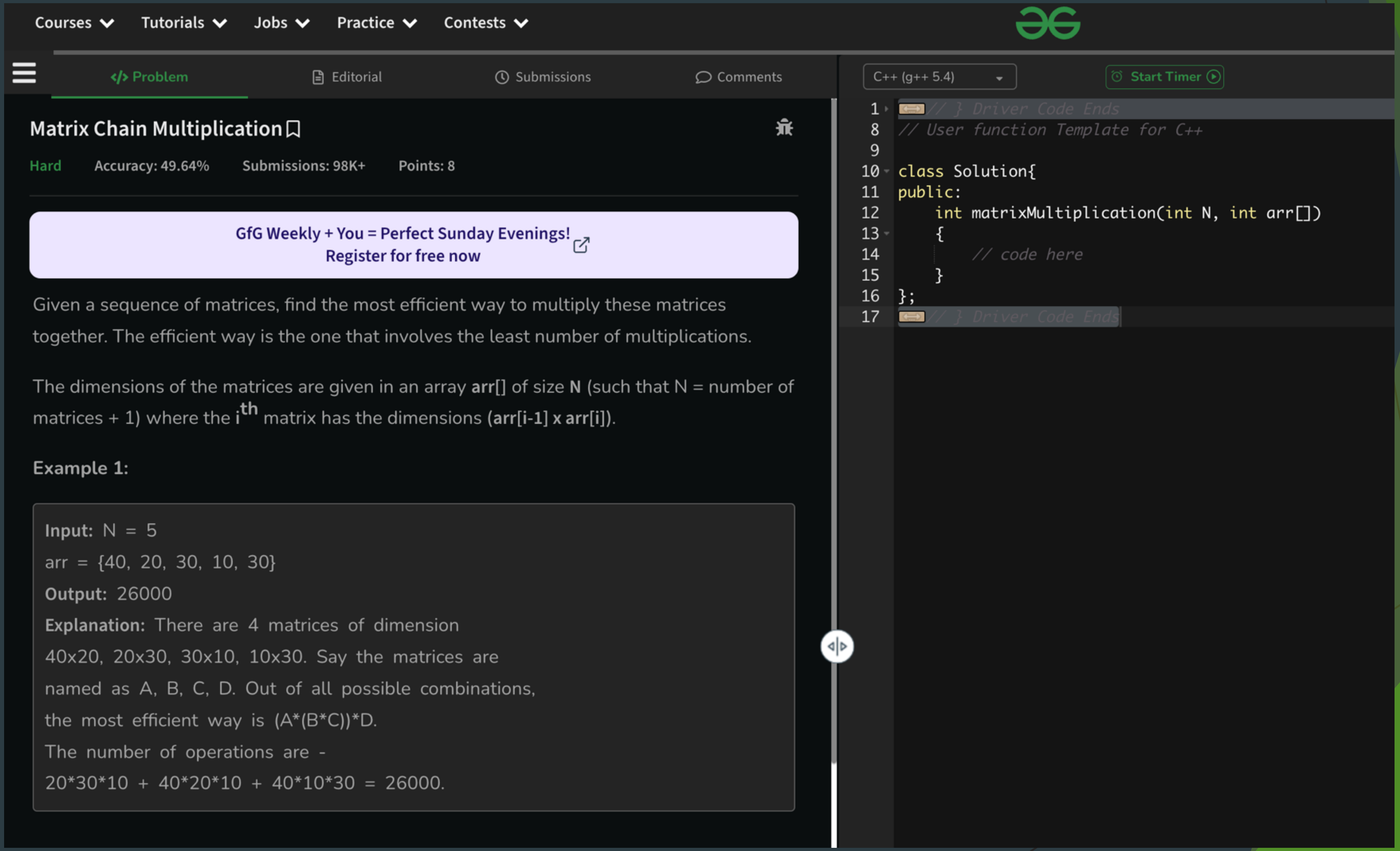
Learning Objectives:

- Understand the basic idea of dynamic programming
- Know under what situation dynamic programming could be applied



Exercise 2

Geeks for Geeks: exercise link



Matrix Chain Multiplication

Hard Accuracy: 49.64% Submissions: 98K+ Points: 8

GfG Weekly + You = Perfect Sunday Evenings!
Register for free now

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrices are given in an array **arr[]** of size **N** (such that N = number of matrices + 1) where the i^{th} matrix has the dimensions **(arr[i-1] x arr[i])**.

Example 1:

```
Input: N = 5
arr = {40, 20, 30, 10, 30}
Output: 26000
Explanation: There are 4 matrices of dimension
40x20, 20x30, 30x10, 10x30. Say the matrices are
named as A, B, C, D. Out of all possible combinations,
the most efficient way is (A*(B*C))*D.
The number of operations are -
20*30*10 + 40*20*10 + 40*10*30 = 26000.
```

```
// } Driver Code Ends
// User function Template for C++

class Solution{
public:
    int matrixMultiplication(int N, int arr[])
    {
        // code here
    }
};
// } Driver Code Ends
```