VE281

Data Structures and Algorithms

Dynamic Programming

Announcement

- Pre-test for programming project three is available online.

- Participate in the online course evaluation "IDEA".
  - It will close on Dec. 16th.
  - Follow the link in an email sent to your SJTU email account.

Review

- Quick Sort
- Comparison Sort Summary and Time Complexity
  - The worst case time complexity for any comparison sort is $\Omega(n \log n)$.
- Non-Comparison Sort
  - Counting Sort and Bucket Sort
  - Radix Sort

Outline

- Motivation of Dynamic Programming
- Example: Matrix-Chain Multiplication
- Summary

Algorithm Design Methods

- We have already learned two ways to design algorithms:
  - Greedy method.
  - Divide and conquer.

- Some more design methods:
  - Dynamic programming.
  - Backtracking.
  - Branch and bound.

Greedy Method

- Solve problem by making a sequence of decisions.
- Decisions are made one by one in some order.
- Each decision is made using a **greedy** criterion.
- A decision, once made, is usually not changed later.

- Example: Dijkstra's algorithm and Prim's algorithm

Divide and Conquer

- Given a problem to be solved, **split** the problem into several, smaller sub-problems (often recursively).
- Solve each sub-problem independently.
- Combine the solutions to the sub-problems to yield a solution to the original problem.

- Examples: merge sort and quick sort.

Limitation of Divide and Conquer

- Recursively solving sub-problems can result in the same computations being repeated when the subproblems **overlap.**
- For example: computing the **Fibonacci sequence**

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

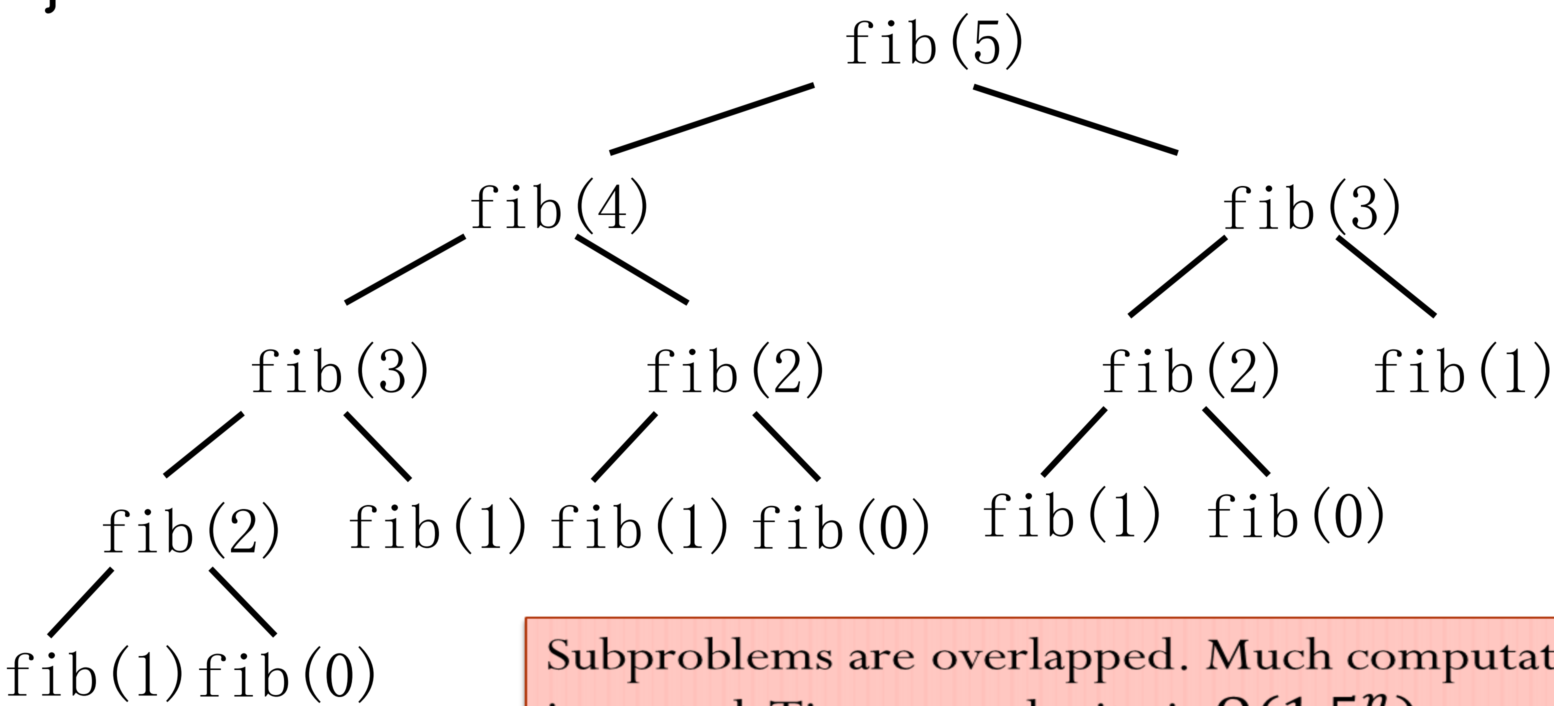
- Divide and conquer approach:

```
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);
}
```



Subproblems are overlapped. Much 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 sub-problem once and store the solution in a table.
    $\Rightarrow$ Trading space for time.
  - If a sub-problem is encountered again, simply look up its solution in the table.
  - Reconstruct the solution to the original problem from the solutions to the sub-problems.
- The more overlap the better, as this reduces the number of sub-problems.
- 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.
  - Constraints: the subgraph must be a spanning tree.
  - Objective function: the sum of all edge weights.

Outline

- Motivation of Dynamic Programming
- Example: 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.

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.
- How many different orders on matrix multiplication?

Matrix-Chain Multiplication

Number of Orders

- Suppose the number of order is $P(n)$ for multiplying n matrices.
- Suppose the last multiplication is $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$.
  - The number of possible order is $P(k)P(n-k)$.
- Thus $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$.
- It can be proved that $P(n) = \Omega(4^n/n^{1.5})$.
- 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_{ij}. The minimal number of scalar multiplication is m_{ij}.
  - We ultimately want to solve Q_{1n}.

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, 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_{ij} into two smaller instances: Q_{ik} and $Q_{(k+1)j}$.

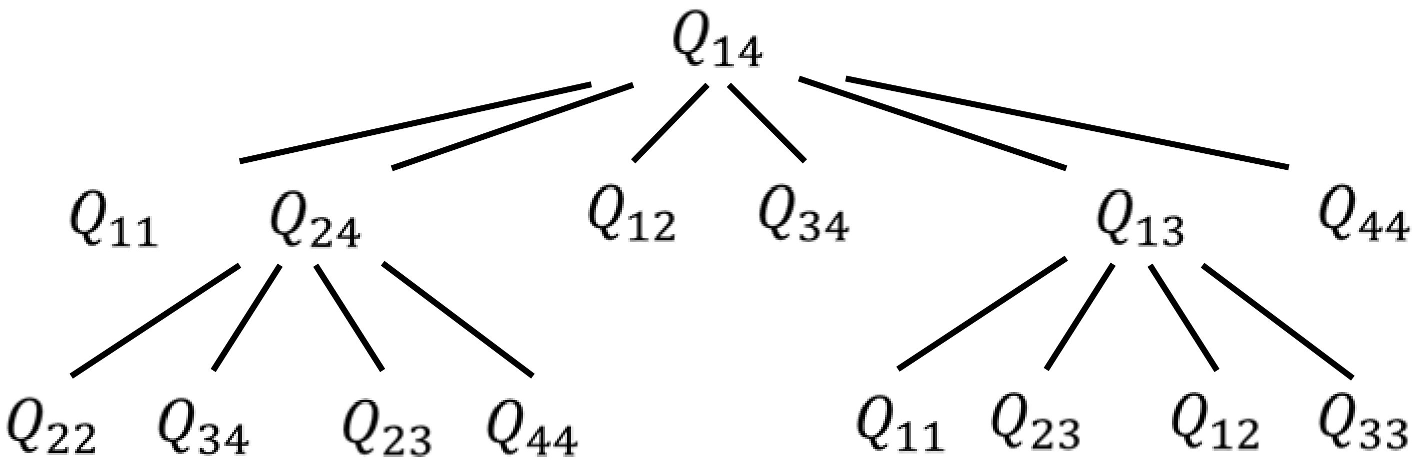
Matrix-Chain Multiplication

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

- However, we don't know k. We need to consider all possible divisions, i.e., all $i \le k \le j-1$.
- Thus, in order to solve Q_{ij}, we need to consider all subproblems Q_{ik} and $Q_{(k+1)j}$, for all $i \le k \le j-1$.
  - $m_{ij} = \min_{i \le k \le j-1}(m_{ik} + m_{(k+1)j} + p_{i-1}p_kp_j)$

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_{ij}, 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:
$$m_{ij} = \min_{i \le k \le j-1} (m_{ik} + m_{(k+1)j} + p_{i-1} p_k p_j)$$
- Initial situation $m_{11} = m_{22} = \cdots = m_{nn} = 0$.
- In the first round, we compute $m_{12}, m_{23}, \dots, m_{(n-1)n}$.
- In the second round, we compute $m_{13}, m_{24}, \dots, m_{(n-2)n}$.
- So on and so forth. In the l-th round, we compute $m_{1(l+1)}, m_{2(l+2)}, \dots, m_{(n-l)n}$.
- Finally, we compute m_{1n}.
- We also record the partition k which gives the minimal m_{ij} in s_{ij}.

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

$T(RightSz)$

- Recursive relation:
  $T(N) = T(LeftSz) + T(RightSz) + \Theta(N)$
- Worst case happens when each time the pivot is the smallest item or the largest item.
  - $T(N) = T(N-1) + T(0) + \Theta(N)$

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

$T(RightSz)$

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

$$= T(N-1) + T(0) + dN$$

$$= T(N-2) + 2T(0) + d(N-1) + dN$$

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

$$= T(0) + NT(0) + d + 2d + \cdots + d(N-1) + dN$$

$$= \min\{20, 100\}$$

Matrix-Chain Multiplication

Example

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

$\boxed{\Theta(N)}$

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

$\boxed{T(LeftSz)}$

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

$$= T(0) + NT(0) + d + 2d + \cdots + d(N-1) + dN$$

$$= \min\{20, 100\}$$

Matrix-Chain Multiplication

Example

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

$\Theta(N)$	1	2	3	4
1	0	100	20	
2	–	0	10	
3	–	–	0	200
4	–	–	–	0

$T(LeftSz)$	1	2	3	4
1	–	1	1	
2	–	–	2	
3	–	–	–	3
4	–	–	–	–

$$= \Theta(N^2)$$

- Recursive relation:
  $T(N) = T(LeftSz) + T(RightSz) + \Theta(N)$
- Best case happens when each time the pivot divides the array into two equal-sized ones.
  - $T(N) = T((N-1)/2) + T((N-1)/2) + \Theta(N)$
  - The recursive relation is similar to that of merge sort.
  - $T(N) = \Theta(N \log N)$
- Average case time complexity of quick sort can be proved to be $\Theta(N \log N)$.

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

$$= \Theta(N^2)$$

- Recursive realation:
  $T(N) = T(LeftSz) + T(RightSz) + \Theta(N)$
- Best case happens when each time the pivot divides the array into two equal-sized ones.
  - $T(N) = T((N-1)/2) + T((N-1)/2) + \Theta(N)$
  - The recursive relation is similar to that of merge sort.
  - $T(N) = \Theta(N \log N)$
- Average case time complexity of quick sort can be proved to be $\Theta(N \log N)$.

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

Matrix-Chain Multiplication

Example

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

$\Theta(N)$

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

$T(LeftSz)$

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

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	Yes	No
Merge Sort	$O(N \log N)$	$O(N \log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	Weakly	No

- sort runs at $O(N^2)$?
  - Insertion sort corrects one reverse-ordered pair at a time.
  - Quick sort moves elements far distances, correcting multiple reverse-ordered pairs at a time.
- Why is quick sort's worst-case $O(N^2)$ while merge sort has no such a problem?
  - The choice of pivot determines size of partitions in quick sort,

Matrix-Chain Multiplication

Example

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

$\Theta(N)$	1	2	3	4
1	0	100	20	220
2	–	0	10	30
3	–	–	0	200
4	–	–	–	0

Optimal Value

$T(LeftSz)$	1	2	3	4
1	–	1	1	3
2	–	–	2	3
3	–	–	–	3
4	–	–	–	–

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Heap Sort	$O(N \log N)$	$O(N \log N)$	Yes	No
Merge Sort	$O(N \log N)$	$O(N \log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	Weakly	No

- How can quick sort run at $O(N \log N)$, while insertion sort runs at $O(N^2)$?
  - Insertion sort corrects one reverse-ordered pair at a time.
  - Quick sort moves elements far distances, correcting multiple reverse-ordered pairs at a time.
- Why is quick sort's worst-case $O(N^2)$ while merge sort has no such a problem?
  - The choice of pivot determines size of partitions in quick sort, whereas merge sort cuts array in half every time.

Matrix-Chain Multiplication

Constructing an Optimal Order

- For comparison sort, is $O(N \log N)$ the best we can do in the worst case?

- Theorem: A sorting algorithm that is based on pairwise comparisons must use $\Omega(N \log N)$ operations to sort in the worst case.

- Proof: Consider the decision tree.

Matrix-Chain Multiplication

Example

- Decision tree is a binary tree.
- The sorting result is at one of the leaves following the results of a sequence of pairwise comparisons.
- The number of pairwise comparisons in the worst case corresponds to the deepest leaf in the decision tree, or the height of the tree.
- The number of leaves in a decision tree for sorting N items is $N!$, i.e., the number of permutations on N items.
- Since a binary tree of height h has **at most** 2^h leaves, the height of the decision tree is **at least** $\lceil \log N! \rceil$.

$$\geq \frac{N}{2}\log(N/2) = \Omega(N \log N)$$

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

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

$$A_2 \times A_3 = A_2 \times A_3$$

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

Matrix-Chain Multiplication

Time Complexity

- **Radix sort** sorts integers by looking at one digit at a time.
- Procedure: Given an array of integers, from the least significant bit (LSB) to the most significant bit (LSB), repeatedly do **stable** bucket sort according to the current bit.

- For sorting base-b numbers, bucket sort needs b buckets.
  - For example, for sorting decimal numbers, bucket sort needs 10 buckets.

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 sub-

Outline

- Motivation of Dynamic Programming
- Example: 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.
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.

Memoization

- In dynamic programming, solutions to subproblems are pre-computed and stored in a table.
  - A **bottom-up** approach.
- An alternative approach is to "**memoize**" during the recursion.
  - A **top-down** approach. Start from the large subproblem.
  - When a subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in a table. Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.

Reference

- **Introduction to Algorithms (3rd Edition)**, by *Thomas Cormen et. al.*, MIT Press (2009)
  - Chapter 15 Dynamic Programming