Learn DAA: From B K Sharma

TCS-503: Design and Analysis of Algorithms

Advanced Design and Analysis Techniques: Dynamic Programming

Unit III

- Advanced Design and Analysis Techniques:
  - Dynamic Programming
  - Greedy Algorithms
  - Amortized Analysis
  - Backtracking.

Learn DAA: From B K Sharma

Advanced Algorithm Design Techniques: Optimization Techniques

- Dynamic Programming
- Greedy Algorithms
- Backtracking

Advanced Algorithm Analysis Technique

- Amortized Analysis

Learn DAA: From B K Sharma

Advanced Algorithm Design Techniques: Optimization Techniques

Dynamic Programming

Optimal Substructure Property + Overlapping Substructure Property

Programming means "Tabular Method", not computer programming.

Works for more problems

In between, not as fast as greedy

Greedy Algorithms

Greedy Choice Property + Optimal Substructure Property

Can be applied to few problems

but gives fast algorithms

Backtracking

Can be applied to almost all problems

but gives very slow algorithms

Learn DAA: From B K Sharma

Dynamic Programming

Why Dynamic Programming?

Divide and Conquer technique / Recursion gives us "bad"(poor) performance, because sub-problems repeatedly solved.

When Dynamic Programming?

When the Problem has

Optimal Substructure Property	+	Overlapping Substructure Property

a.k.a. Elements of DP

How Dynamic Programming? What are the steps of DP?

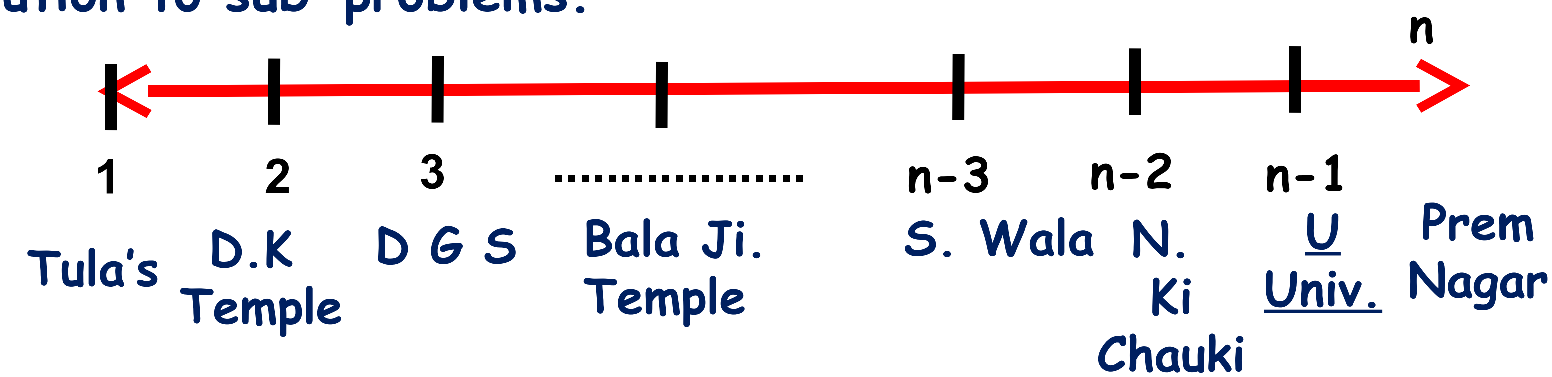
1) Characterize the structure of an optimal solution
2) Recursively define the value of an optimal solution
3) Compute the value of an optimal solution in a bottom-up fashion
4) Construct an optimal solution from computed information

Dynamic Programming

Optimal Substructure Property

a.k.a Principle of Optimality

An optimal solution to a problem contains within it an optimal solution to sub-problems.



Dynamic Programming

Overlapping Substructure Property

If a recursive algorithm revisits the same sub-problems over and over:

$\Rightarrow$ the problem has overlapping sub-problems.

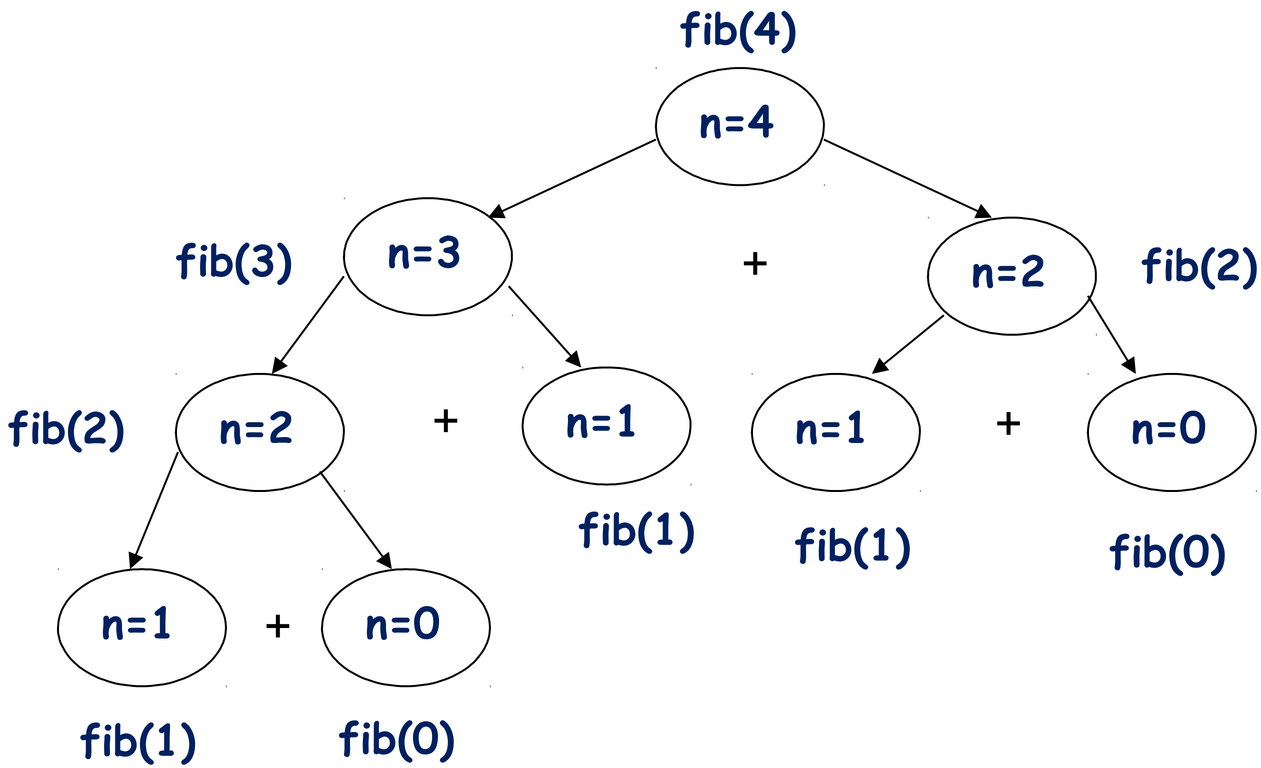
Dynamic Programming is mainly used when solutions of same sub-problems are needed again and again.

In dynamic programming, computed solutions to sub-problems are stored in a table so that these don't have to be recomputed.

So Dynamic Programming is not useful when there are no common (overlapping) sub-problems because there is no point storing the solutions if they are not needed again.

Dynamic Programming

Overlapping Substructure Property



We can see that the function fib(2) is being called 2 times. If we would have stored the value of fib(2), then instead of computing it again, we could have reused the old stored value.

Learn DAA: From B K Sharma

Problems to be solved using Techniques of Dynamic Programming

- Matrix Chain Multiplication
- Longest common Sequence
- Assembly Line Scheduling
- Optimal BST
- Knapsack Problem

Matrix Chain Multiplication

Problem: Given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

In what order should we multiply the matrices?

Parenthesize the product to get the order in which matrices are multiplied.

E.g.: $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$ Or

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

Which one of these orderings should we choose?

The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Learn DAA: From B K Sharma

Matrix Chain Multiplication

Example

$$A_1 \cdot A_2 \cdot A_3$$

A_1: 10 x 100 A_2: 100 x 5 A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2$ = 10 x 100 x 5 = 5,000 (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3)$ = 10 x 5 x 50 = 2,500

   Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3$ = 100 x 5 x 50 = 25,000 (100 x 50)

   $(A_1 \cdot (A_2 \cdot A_3))$ = 10 x 100 x 50 = 50,000

   Total: 75,000 scalar multiplications

The first way is ten times faster than the second !!!

Thus, goal of DP is:

Given a chain of matrices to multiply, determine the fewest number of multiplications necessary to compute the product.

Matrix Chain Multiplication

What is the number of possible parenthesizations?

It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

Exhaustively checking all possible parenthesizations is not efficient!

Matrix Chain Multiplication

Problem Statement

Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$

A_1	A_2	A_i	A_{i+1}	A_n
$p_0 \times p_1$	$p_1 \times p_2$	$p_{i-1} \times p_i$	$p_i \times p_{i+1}$	$p_{n-1} \times p_n$

fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Matrix Chain Multiplication

1. Characterize the structure of an optimal solution

Optimal Substructure Property

The optimal parenthesization of A_{i---j} contains within it the optimal parenthesization of A_{i----k} and $A_{k+1----j}$

Every optimal parenthesization consists of two optimal parenthesizations : one of a prefix A_1, A_2, ….A_k and other of a suffix A_{k+1}. A_{k+2}.,…..A_j

$$A_i \times A_{i+1} \times \ldots \times A_k \times A_{k+1} \times \ldots \times A_j$$

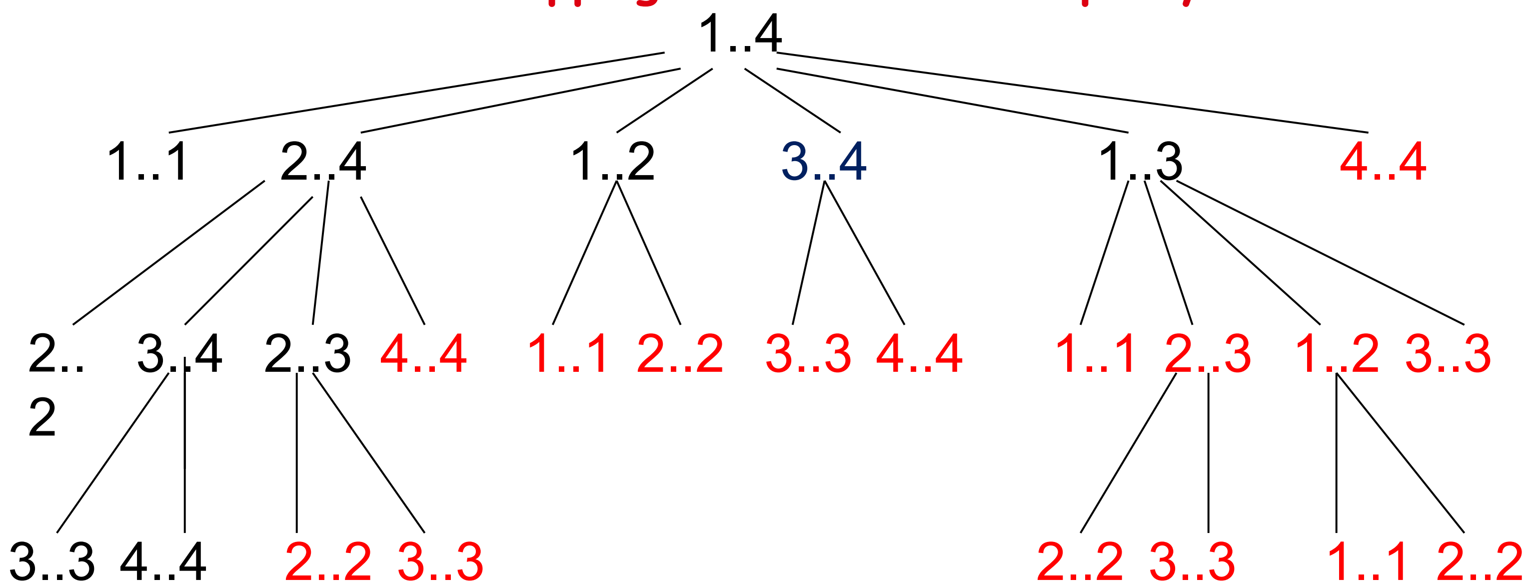
$$A_{i \ldots j}$$

$$= A_{i \ldots k} A_{k+1 \ldots j}$$

Matrix Chain Multiplication

1. Characterize the structure of an optimal solution

Overlapping Substructure Property



The divide-and-conquer recursive algorithm solves the overlapping problems *over and over.*

In contrast, DP solves the same (overlapping) sub-problems only once (at the first time), then store the result in a table, when the same subproblem is encountered later, just look up the table to get the result.

Learn DAA: From B K Sharma

Matrix Chain Multiplication

2) Recursively define the value of an optimal solution

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

If we have to calculate m[1,2], then k=1.

If we have to calculate m[2,3], then k=2.

If we have to calculate m[1,3], then k=1,2.

Matrix Chain Multiplication

3) Compute the value of an optimal solution in a bottom-up fashion

Compute $A_1 \cdot A_2 \cdot A_3$

A_1: 10 x 100 $(p_0 \times p_1)$

A_2: 100 x 5 $(p_1 \times p_2)$

A_3: 5 x 50 $(p_2 \times p_3)$

1. $((A_1 \cdot A_2) \cdot A_3)$
2. $(A_1 \cdot (A_2 \cdot A_3))$

Example: $m[i, j] = \min \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{ m[i, k] + m[k+1, j] + p_{i-1}p_kp_j \} & \text{if } i < j \end{cases}$

Compute $A_1 \cdot A_2 \cdot A_3$

A_1: 10 x 100 $(p_0 \times p_1)$

A_2: 100 x 5 $(p_1 \times p_2)$

A_3: 5 x 50 $(p_2 \times p_3)$

j \ i	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

$m[i, i] = 0$ for $i = 1, 2, 3$

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2$ (A_1A_2)

$= 0 + 0 + 10 * 100 * 5 = 5{,}000$

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3$ (A_2A_3)

$= 0 + 0 + 100 * 5 * 50 = 25{,}000$

$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = 7{,}500 & ((A_1A_2)A_3) \end{cases}$

Matrix Chain Multiplication

If we want to perform Step 4, then in step 3 we must store additional information.

Additional Information in MCM: s[i, j]=k

When we calculate m[1,2], then k=1. Hence, s[1,2]=1

When we calculate m[2,3], then k=2. Hence, s[1,2]=2

When we calculate m[1,3], then k=1,2 but min. is for k=2. Hence, s[1,3]=2

Table s[i, j]=k

j \ i	1	2	3
3	2	2	
2	1		
1			

Learn DAA: From B K Sharma

Matrix Chain Multiplication

4) Construct an optimal solution from computed information

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"i
   else print "("
         PRINT-OPT-PARENS(s, i, s[i, j])
         PRINT-OPT-PARENS(s, s[i, j] + 1, j)
         print ")"
```

	1	2	3	
3	2	2		
2	1			j
1				

i

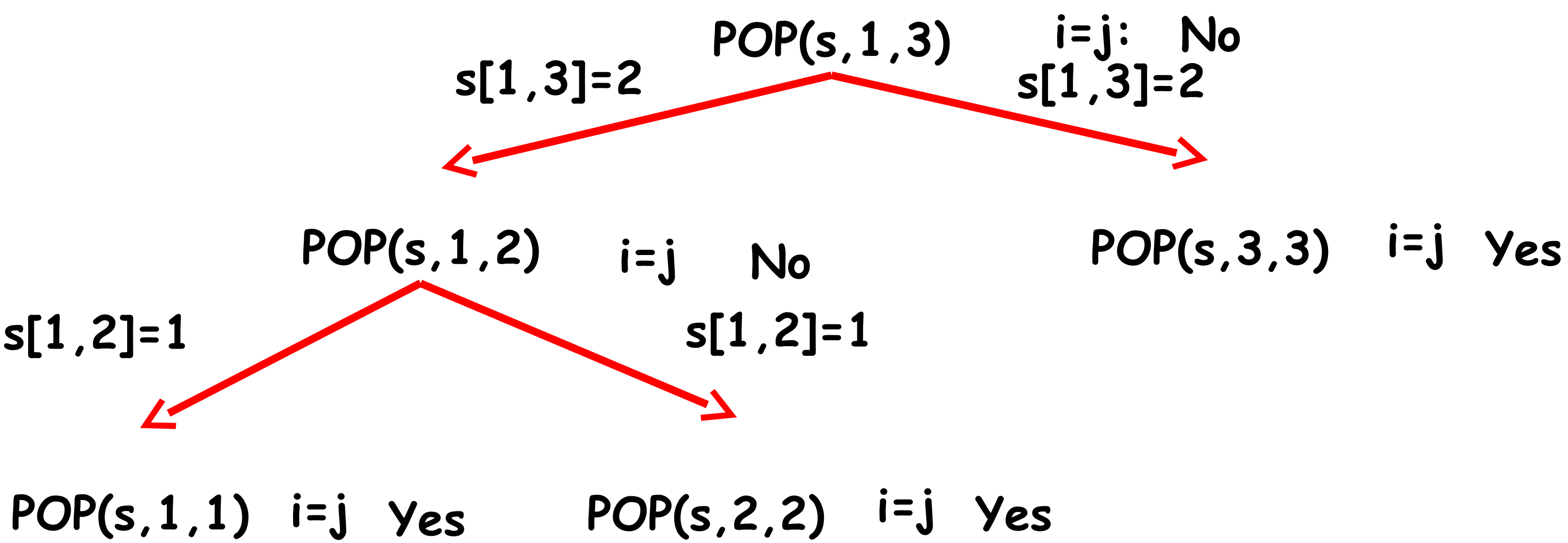
Table s[i, j]=k

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"i
   else print "("
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1, j)
        print ")"
```

j \ i	1	2	3
3	2	2	
2	1		
1			

Table s[i, j]=k

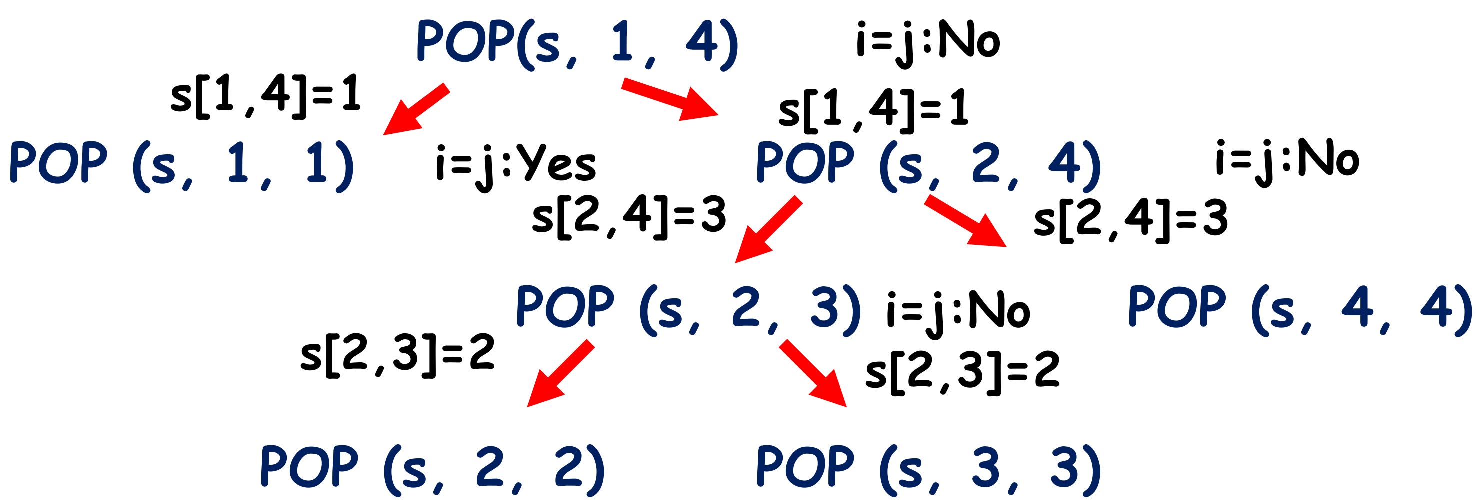
$(\; (\; A_1 \; A_2 \;) \; A_3 \;)$



```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"i
   else print "("
         PRINT-OPT-PARENS(s, i, s[i, j])
         PRINT-OPT-PARENS(s, s[i, j] + 1, j)
         print ")"
```

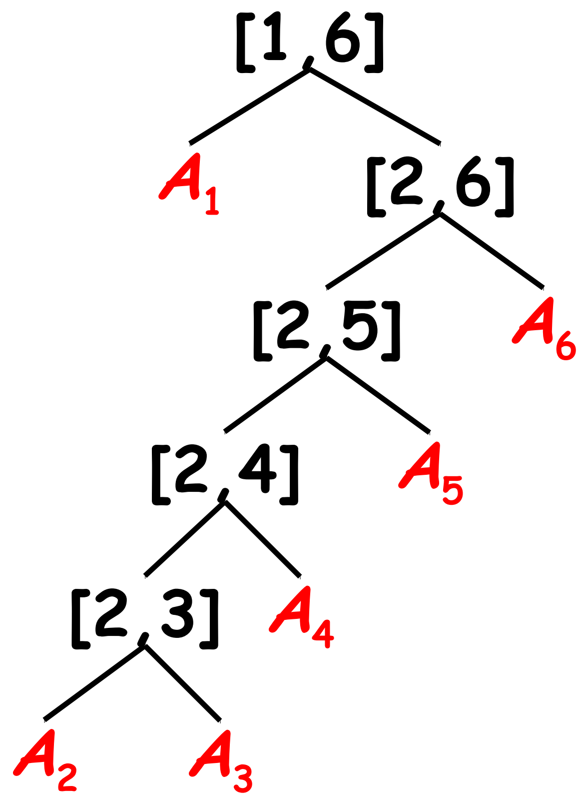
i \ j	1	2	3	4
1	0	1	1	1
2		0	2	3
3			0	3
4				0

(A_1 ((A_2 A_3) A_4))



	1	2	3	4	5	6	
1		1	1	1	1	1	
2			2	3	4	5	
3				3	4	5	
4						4	5
5						5	
6							

$A_1((((A_2A_3)A_4)A_5)\ A_6)$



Comparison between Divide-and-conquer and Dynamic programming

	Divide-and-conquer	Dynamic programming
similarity	solves problems by combining the solutions to sub-problems.	
difference	1) partition the problem into independent sub-problems, 2) solve the sub-problems recursively, 3) combine their solutions to solve the original problem.	the sub-problems are not independent, that is, when sub-problems share sub-sub-problems.
	might do more work than necessary, repeatedly solving the common Sub-sub-problems.	solves every sub-sub-problem just once and then saves its answer in a table, avoiding re-computing the answer every time the Sub-sub-problem is encountered.