fit@hcmus

PROGRAMMING TECHNIQUES

Recursion & Dynamic Programming

Nguyễn Hải Minh - 04/2024

Outline

- Dynamic Programming Definition
- Walk through examples:
  - Example 1: Fibonacci number
  - Example 2: The Knapsack problem
- Properties of Dynamic Programming
- Application
- More Reading

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems (minimize/maximize)
- DP can be used when the solution to a problem may be viewed as the result of a sequence of decisions
- DP reduces computation by
  - Solving subproblems in a **bottom-up** fashion.
  - **Storing** solution to a subproblem the first time it is solved.
  - **Looking up** the solution when subproblem is encountered again.
- *Key: determine structure of optimal solutions*

Dynamic Programming History

- Bellman. Pioneered the systematic study of dynamic programming in the 1950s.
- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Steps in Dynamic Programming

1. Define the problem and identify its optimal structure.
   - Optimal structure: the optimal solution to the problem can be obtained by combining the optimal solutions to its subproblems.
2. Formulate a recursive solution (top-down).
3. Compute the value of an optimal solution in a bottom-up fashion.
   - Memorize the recursive solutions by storing the results of previous computation in a table.
   - Convert the recursive solution to an iterative one.

→ *We will study these steps through some examples*

EXAMPLE 1: FIBONACCI NUMBERS

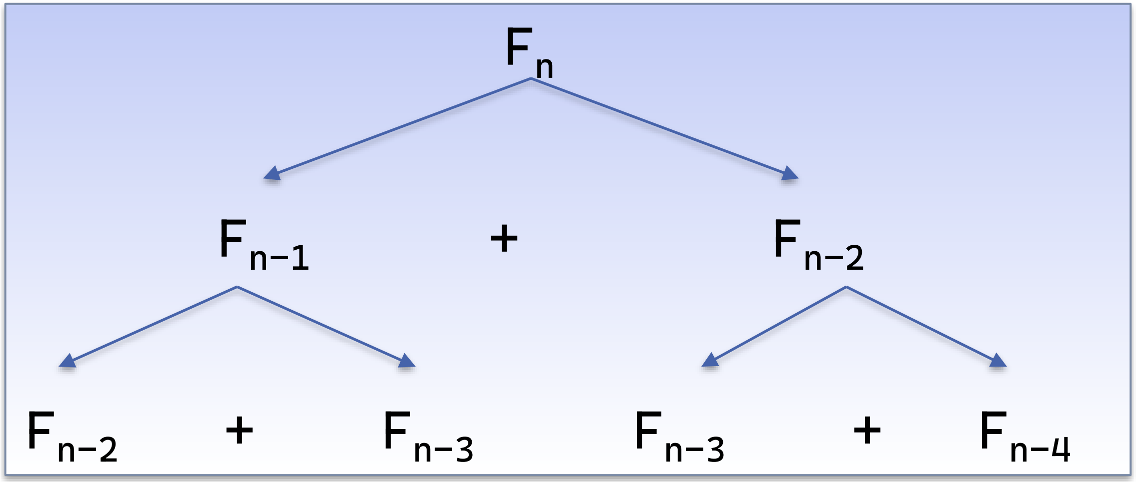
Fibonacci Numbers

- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

$$F_i = i \qquad \text{if } i \le 1$$
$$F_i = F_{i-1} + F_{i-2} \qquad \text{if } i > 1$$

Solved by a recursive program:

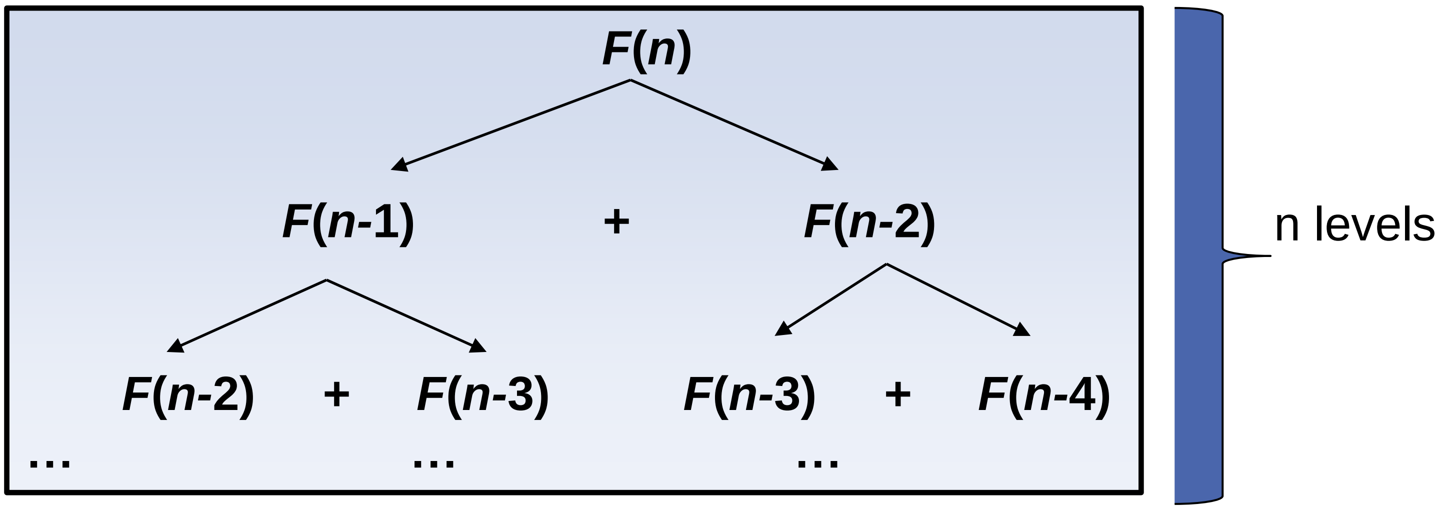
```
int Fib(int n)
{
  if (n <= 1)
    return n;
  else
    return Fib(n – 1)
         + Fib(n – 2);
}
```



→ *This is a **top-down** approach*

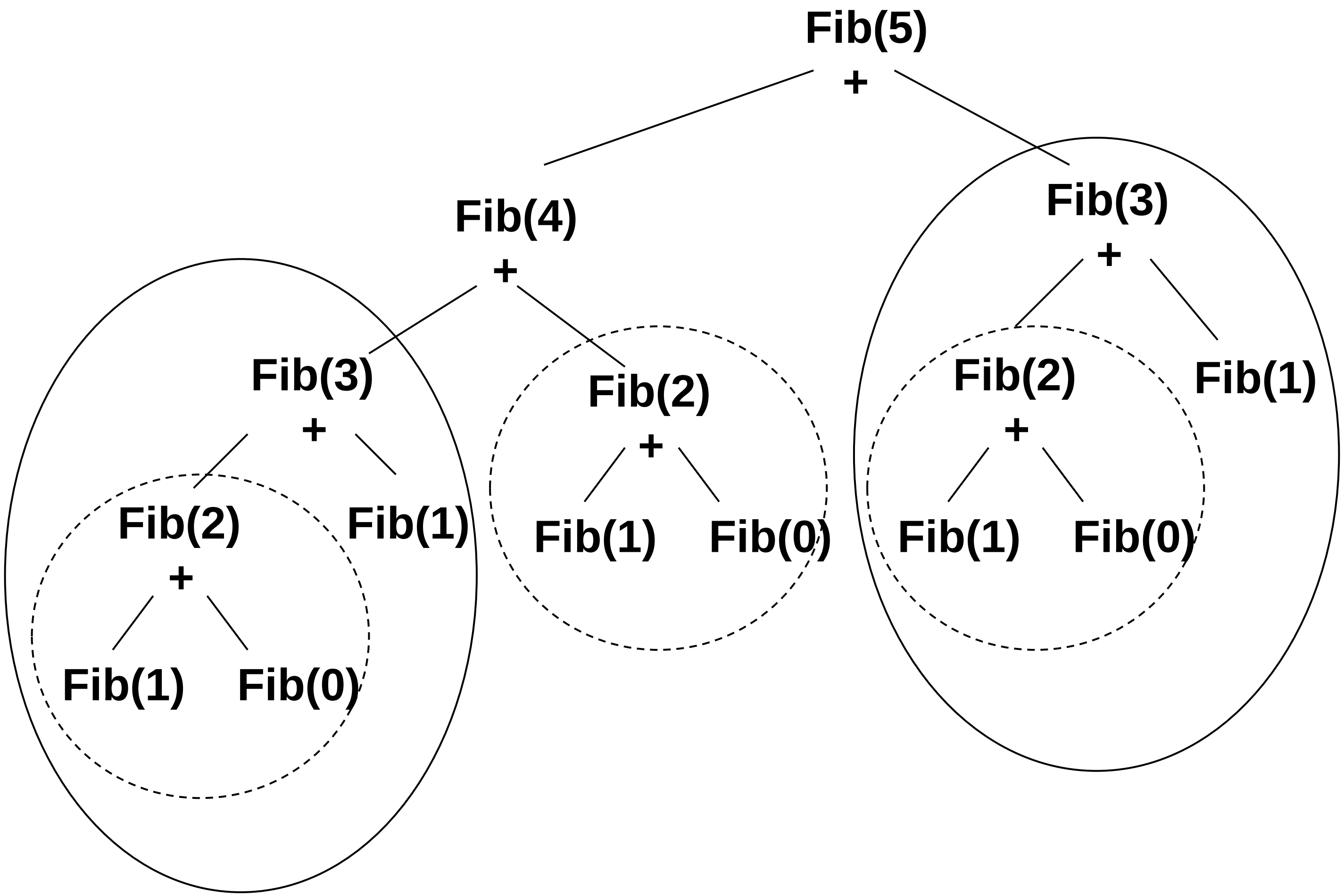
Fibonacci Numbers

- Why is the top-down so inefficient?
  - Recomputes many sub-problems.
    - How many times is `F(n-5)` computed?



- We can enhance this problem by storing solution to the sub-problem.

Fibonacci Numbers



Fibonacci Numbers

```
#include <iostream>
int* memo = new int[n+1];
int Fib (int n)
{
   if (n <= 1)
      return 1;
   if(memo[n] != 0)
      return memo[n];
   int result = Fib(n - 1) + Fib(n - 2);
   memo[n] = result;
   return result;
}
```

Each F_i is calculated only once

But it is still inefficient because *Resursive calls* are called multiple times for an n

Fibonacci Numbers

- Using a ***bottom-up*** approach can solve this problem!
  - F(0) = 0
  - F(1) = 1
  - F(2) = 1+0 = 1
  - …
  - F(n-2) =
  - F(n-1) =
  - F(n) = F(n-1) + F(n-2)

0	1	1	…	F(n−2)	F(n−1)	F(n)

Fibonacci Numbers – DP

```cpp
#include <iostream>
int Fib_DP(int n)
{
  /* Declare an array to store Fibonacci numbers. */
  int *f = new int[n+1];//1 extra to handle case, n
  int i;
  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;
  for (i = 2; i <= n; i++)
     f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

Fibonacci Numbers – DP

```
#include <iostream>
int Fib_DP(int n)
{
  int a = 0, b = 1, c, i;
  if(n == 0)
    return 0;
  for (i = 2; i <= n; i++)
  {
     c = a + b;
     a = b;
     b = c;
  }
  return b;
}
```

Space Optimization

EXAMPLE 2: THE KNAPSACK PROBLEM

The Knapsack problem

- Problem statement:
  - A thief is robbing a museum and he only has a single knapsack to carry all the items he steals.
  - The knapsack has a capacity for the amount of weight it can hold. Each item in the museum has a weight and a value associated with it.

The Knapsack problem - Variation

- 0/1 Knapsack problem
  - Each item is chosen at most once.
  - Decision variable for each item is a binary value (0 or 1)
- Multiple-choice Knapsack problem
  - Each item can be put to the knapsack multiple times.
  - Decision variable for each item is an integer value.
- Bounded Knapsack problem
  - Same with multiple-choice but each item has the max number of times it can be chosen.
- Knapsack problem with fractional items
- Knapsack problem with multiple constraint
- …

0/1 Knapsack problem – Example

- Knapsack's capacity: 10kg
- 5 items can be chosen:
  - Item 1: \$6 (2 kg)
  - Item 2: \$10 (2 kg)
  - Item 3: \$12 (3 kg)
  - Item 4: \$16 (4kg)
  - Item 5: \$20 (5kg)

- Optimal function: $f(n, W)$ $(n = 5, W = 10)$

0/1 Knapsack problem – Example

- Optimal structure: to find $f(n, W)$:
  1. **Case 1:** including the *n*th item
     → find $f(n-1, W-w_n)+x_n$ with x_n is the value of item *n*th
  2. **Case 2:** not include the *n*th item
     → find $f(n-1, W)$
- Hence, optimal f is calculated by:

$$\boldsymbol{f(n, W) = \max\left(f(n-1, W-w_n)+x_n, f(n-1, W)\right)}$$

→ *This can be solved using **recursion** which is a **top-down** strategy.*

The Knapsack problem – Recursive

```
//Returns the maximum value that can be put in a knapsack of
//capacity W
int KnapSack(int n, int wt[], int val[], int W) {
  if (n == 0 || W == 0) // Base Case
     return 0;
 // If weight of the nth item is more than Knapsack capacity W,
 //then this item cannot be included in the optimal solution
  if (wt[n - 1] > W)
     return KnapSack(n - 1, wt, val, W);

  // else: Return the maximum of two cases:
  // (1) nth item included    // (2) not included
  return max( val[n-1] + KnapSack(n-1, wt, val, W-wt[n-1]) ,
                                   KnapSack(n-1, wt, val, W) );
}
```

$f(n-1, W-w_n) + x_n$

$f(n-1, W)$

0/1 Knapsack problem – DP

- We can also use a ***bottom-up*** approach and memorize the solutions to subproblems to a table → *Dynamic Programming*
  - Row: items
  - Column: remaining weight capacity of the knapsack
  - We fill the table using the following recurrence relation:

$$\boldsymbol{f(W,i) = \max\big(f(i-1, W-w_i) + x_i, f(i-1, W)\big)}$$

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

0/1 Knapsack problem – DP

- Use only item 1:

→ $f(1,1) = 0, f(1,2) = 6, f(1,3) = 6, \ldots, f(1,10) = 6$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10										
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1 & 2:

→ $f(2,2) = \max\left(f(1,0) + 10, f(1,2)\right) = 10, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10								
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1 & 2:

→ $f(2,3) = \max\left(f(1,1) + 10, f(1,3)\right) = 10, \ldots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10							
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1 & 2:

→ $f(2,4) = \max\big(f(1,2) + 10, f(1,4)\big) = 16, \dots$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16						
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1 & 2:

→ $f(2, i) = \max\big(f(1, W-2) + 10, f(1, W)\big)$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12										
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1, 2, 3:

→ $f(3,3) = \max\big(f(2,0) + 12, f(2,3)\big) = 12$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12							
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1, 2, 3:

→ $f(3,4) = \max\big(f(2,1) + 12, f(2,4)\big) = 16$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16						
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1, 2, 3:

→ $f(3,5) = \max\left(f(2,2) + 12, f(2,5)\right) = 22$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22					
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1, 2, 3:

→ $f(3, i) = \max\left(f(2, W-3) + 12, f(2, W)\right)$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16										
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1, 2, 3, 4:

→ $f(4, i) = \max\left(f(3, W - 4) + 16, f(3, W)\right)$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20										

0/1 Knapsack problem – DP

- Use item 1, 2, 3, 4, 5:

→ $f(5,10) = \max\left(f(4,5) + 20, f(4,10)\right) = 42$

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20	0	10	12	16	22	26	30	32	38	42

0/1 Knapsack problem – DP

- Solution:
  - item 5 + item 3 + item 2 → $42 – 10kg

			0|1	2	3	4	5	6	7	8	9	10
1	2kg	$6	0	6	6	6	6	6	6	6	6	6
2	2kg	$10	0	10	10	16	16	16	16	16	16	16
3	3kg	$12	0	10	12	16	22	22	28	28	28	28
4	4kg	$16	0	10	12	16	22	26	28	32	38	38
5	5kg	$20	0	10	12	16	22	26	30	32	38	42

The Knapsack problem – DP

```
int KnapSack(int n, int wt[], int val[], int W)
{
  int i, w;
  //Create a table K to store solutions of subproblems
  int** K = new int*[n + 1];
  for(i = 0; i <= n; i++)
    K[i] = new int[W + 1];
  for (i = 0; i <= n; i++) //Build table K[][] in bottom up manner
    for (w = 0; w <= W; w++) {
      if (i==0 || w==0)
          K[i][w] = 0;
      else if (wt[i-1] <= w)
          K[i][w] = K[i-1][w];
      else
          K[i][w] = max(K[i-1][w-wt[i-1]] + val[i-1], K[i-1][w] );
    }
  return K[n][W];
}
```

$f(i-1, W-w_i) + x_i$

$f(i-1, W)$

Multi-choice Knapsack problem

- Knapsack's capacity: 10kg
- 3 items can be chosen:
  - Item 1: \$5 (3 kg)
  - Item 2: \$7 (4 kg)
  - Item 3: \$8 (5 kg)

→ 1 item can be picked many times

- Optimal function: $f(n, W)$ (n=3, w=10)

Multi-choice Knapsack problem

- Optimal $f(i, w)$ is calculated by:

$$\boldsymbol{f(i, W) = \max\left(f(i-1, W-kw_i) + kx_i, f(i-1, W)\right)}$$

k item *i* + optimum combination of weight *w* – *kw*$_i$

NO Item *i* + optimum combination items 1 to *i* – 1

- k is the number of times item i appears in the knapsack. $k = 1,2, \ldots$ so that $kw_i \leq W$

Multi-choice Knapsack problem

- Use only item 1:

→ $f(1,3) = 5, f(1,6) = 10, f(1,9) = 15$

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7											
3	5kg	$8											

$1 \times x_1$ $2 \times x_1$ $3 \times x_1$

Multi-choice Knapsack problem

- Use only item 1 & 2:

→ $f(2, W) = \max\big(f(1, W - 4k) + 7k, f(1, W)\big)$

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	\$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	\$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	\$8											

Multi-choice Knapsack problem

- Use item 1, 2, and 3:

→ $f(3, W) = \max\big(f(2, W - 5k) + 8k, f(2, W)\big)$

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	$8	0	0	0	5	7	8	10	12	14	15	17

Multi-choice Knapsack problem

- Solution:
  - item 2 + 2 x item 1 → \$17 – 10kg

			0	1	2	3	4	5	6	7	8	9	10
1	3kg	\$5	0	0	0	5	5	5	10	10	10	15	15
2	4kg	\$7	0	0	0	5	7	7	10	12	14	15	17
3	5kg	\$8	0	0	0	5	7	8	10	12	14	15	17

Multi-choice Knapsack problem

- As an exercise, rewrite the Knapsack function to solve multi-choice Knapsack problem:
  - Using Recursion (top-down)
  - Using Dynamic Programming (bottom-up)

Properties of Dynamic Programming

- There are 2 main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

1. **Overlapping Subproblems**
   → solutions of same subproblems are needed again and again
2. **Optimal Structures**
   → optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

Dynamic Programming Applications

- Area
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

More Reading

- The best way to get a feel for this is through some more examples.
  1. Longest Common Subsequence
  2. Longest Increasing Subsequence
  3. Matrix Chain Multiplication
  4. Partition problem
  5. Rod Cutting
  6. Coin change problem
  7. Word Break Problem
  8. …