Problem Solving Techniques 문제해결

Jinkyu Lee

Dept. of Computer Science and Engineering,
Sungkyunkwan University (SKKU)

Contents

- Chapter 11 – Dynamic Programming
  - What is Dynamic Programming?
    - Binomial coefficient example
    - Rod cutting example
  - Program design example: elevator optimization

What is Dynamic Programming? 〈1〉

- **Many problems call for finding the best solution satisfying certain constraints!**
  - **Method 1: Backtracking**
    - It searches all possible solutions and selects the best one, and hence must return the correct answer
    - But it is only feasible for small problem instances
  - **Method 2: Greedy Algorithm**
    - It focuses on making the best local choice at each decision point!
      - Ex) A shortest path from x to y might be walking out of x, repeatedly following the cheapest edge until reaching y.
    - Without a correctness proof, this method is very likely to fail!
  - **Method 3: Dynamic Programming**
    - It systematically searches all possibilities (guarantee correctness) while storing results to avoid recomputing (provide efficiency)
      - Defined by recursive algorithms that describe the solution to the whole problem by those to smaller problems.

What is Dynamic Programming? <2>

- **Implementing Recursive Algorithms**
  - **Method 1: Divide-and-Conquer**
    - As the general method, it is a Top-Down approach
    - It is not feasible for the problems in which common parts exist in subproblems, such as Fibonacci numbers.

1 1 2 3 5 8 13 21 34 55 …

What is Dynamic Programming? 〈2〉

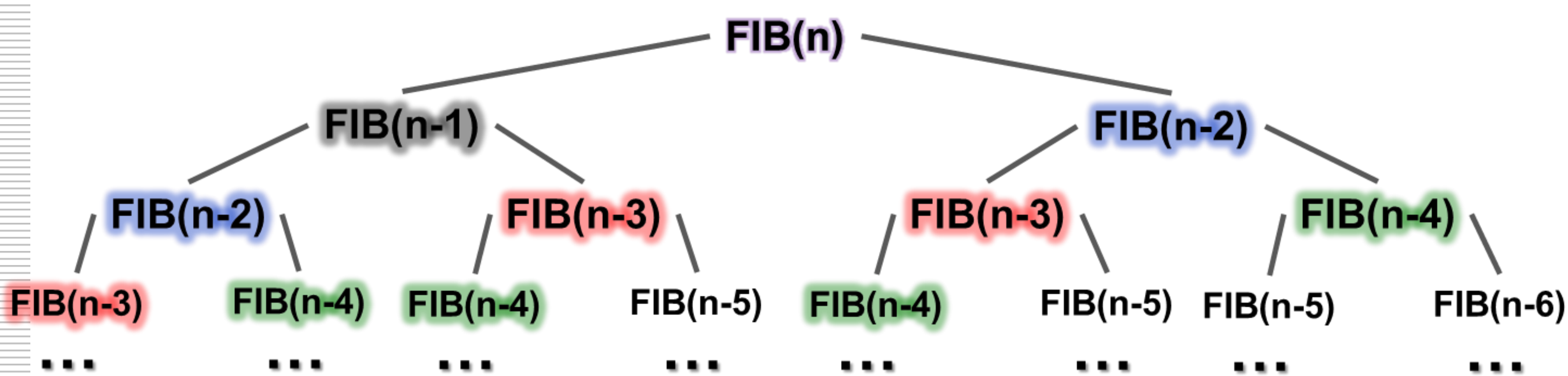
Implementing Recursive Algorithms

Method 1: Divide-and-Conquer

- As the general method, it is a Top-Down approach
- It is not feasible for the problems in which common parts exist in subproblems, such as Fibonacci numbers.
  - Ex) $F(n) = F(n-1) + F(n-2)$

$$= \{F(n-2)+F(n-3)\} + \{F(n-3)+\underline{F(n-4)}\}$$
$$= [\{F(n-3)+\underline{F(n-4)}\}+\{\underline{F(n-4)}+F(n-5)\}]$$
$$+[\{\underline{F(n-4)}+F(n-5)\}+\{F(n-5)+F(n-6)\}] = \ldots$$

Many repetitive computation!

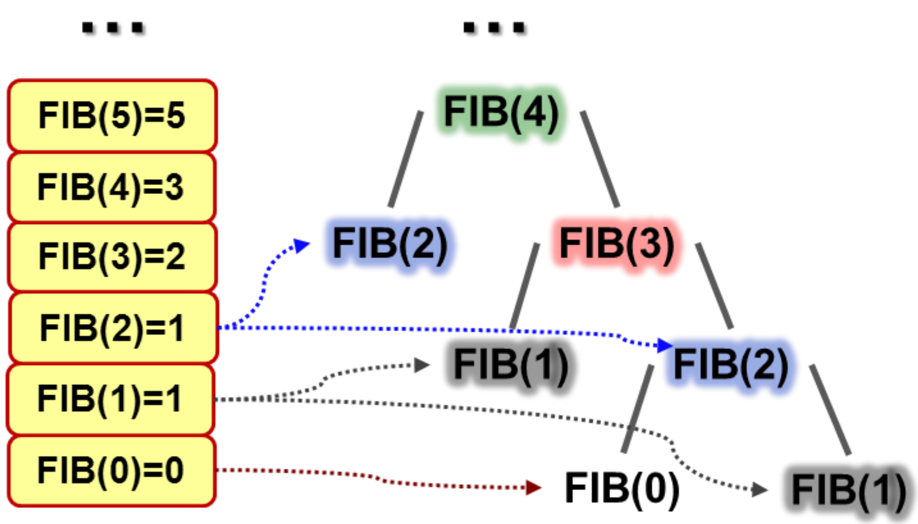


What is Dynamic Programming? 〈3〉

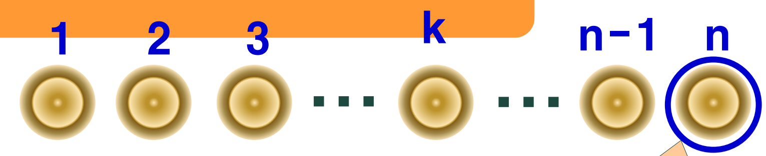
❖ Implementing Recursive Algorithms (cont.)

▪ Method 2: Dynamic Programming

- Unlike the divide-and-conquer, it is a Bottom-Up approach.
- After dividing the entire problem into various levels of subproblems, their solutions are stored at each level. When solving a problem, the necessary information can be reused.
  - Ex) Fibonacci numbers F(1), F(2), … are stored, and they are reused to compute F(n) if necessary.



Binomial Coefficient Example <1>



- **Using the Divide-and-Conquer Method**
  - **Recurrence relation of Binomial Coefficient:**

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Consider whether the nth element belongs to the chosen k elements!

```
int bino(int n, int k) {
    if (k==0 || n==k)
        return 1;
    else
        return bino(n−1, k−1) + bino(n−1, k);
}
```

{bino(n−2, k−2) + bino(n−2, k−1)} + {bino(n−2, k−1) + bino(n−2, k)}

Too many repetitive computation!

- **Thus, huge computational cost is required as n increases!**

Binomial Coefficient Example <2>

Using the Dynamic Programming Method

- The following structure can be discovered from the recurrence relation:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$${}_kC_0 = {}_kC_k = 1$$
$${}_2C_1 = {}_1C_0 + {}_1C_1$$
$${}_3C_1 = {}_2C_0 + {}_2C_1$$
$${}_3C_2 = {}_2C_1 + {}_2C_2$$
$${}_4C_1 = {}_3C_0 + {}_3C_1$$
$${}_4C_2 = {}_3C_1 + {}_3C_2$$

… … … …

	0	1	2	3	4	
0	1					
1	1	1				
2	1	2	1			…
3	1	3	3	1		
4	1	4	6	4	1	
			…			

It does not require a huge cost by avoiding repetitive computation even when n increases!

```
bino(int n, int k) {
    long bc[n][k];
    for( int i=0; i<=n; i++)
        for( int j=0; j<=min(i,k); j++ )
            if( j==0 || j==i ) bc[i][j]=1;
            else    bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
    return bc[n][k];
}
```

Rod cutting example

- Suppose that you have a rod of length n, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you earn. A piece of length i is worth p[i] dollars.

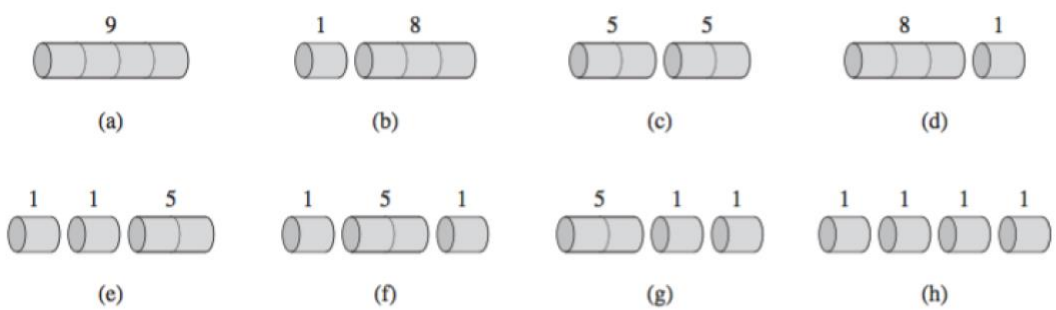
i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

Rod cutting example

- Suppose that you have a rod of length n, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you earn. A piece of length i is worth p[i] dollars.

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

- For example, if you have a rod of length 4, there are eight ways to cut it, and the best strategy is cutting it into two pieces of length 2, which gives you 10 dollars.



https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture12.pdf

Rod cutting example

- Can you solve this problem by backtracking?
- If so, what is the time complexity?

Rod cutting example

- Is it possible to apply the divide-and-conquer approach?

Rod cutting example

- Naïve algorithm

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Jinkyu Lee
Dept. of Computer Science and Engineering

Rod cutting example

- Naïve algorithm

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

- What is the problem?

Rod cutting example

- How can we apply dynamic programming?
  - We want to search all possible cases, but
  - To reduce the time-complexity.

Rod cutting example

- Memoization (top down approach)

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Rod cutting example

- Bottom up approach

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Rod cutting example

- So far, we can find the maximum profit.
- But, what if you want to find the optimal way to split the rod?

Rod cutting example

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

PRINT-CUT-ROD-SOLUTION(p, n)

```
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

i	1	2	3	4	5	6	7	8	9	10
p[i]	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

BOTTOM-UP-CUT-ROD(p, n)

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

Contents

- Program Design Example
  - Elevator Optimization

Design Example: Elevator Optimization <1>

Problem Description

- I work in a very tall building with a very slow elevator.
  It is frustrating for me when people press the buttons for many consecutive floors.
- Thus, we need to write an elevator optimization program
- The riders all enter their intended destinations at the beginning of the trip.
- We limit the elevator to making at most k stops on any given run.
- We assume that the penalty for walking up and down is same.
- Management proposes to break ties among equal-cost solutions by given preference to stopping the elevator at the lowest floor.
- Elevator does not necessary to stop at one of the floors the riders specified.
  - Ex.) If riders specify floors 27 and 29, it can be decided to stop at floor 28.
- The aim is to select the floors to be stopped, so as to minimize the total number of floors people have to walk either up or down.