SC1007
Data Structures and Algorithms

Week 11: Dynamic Programming

College of Engineering

School of Computer Science and Engineering

Instructor: Luu Anh Tuan

Email: anhtuan.luu@ntu.edu.sg

Office: #N4-02b-66

Fibonacci sequence

- Let's consider the calculation of **Fibonacci** numbers:

$$F(n) = F(n-1) + F(n-2)$$

with seed values $F(0) = 0$, $F(1) = 1$

- The sequence look like:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci sequence: recursive algorithm

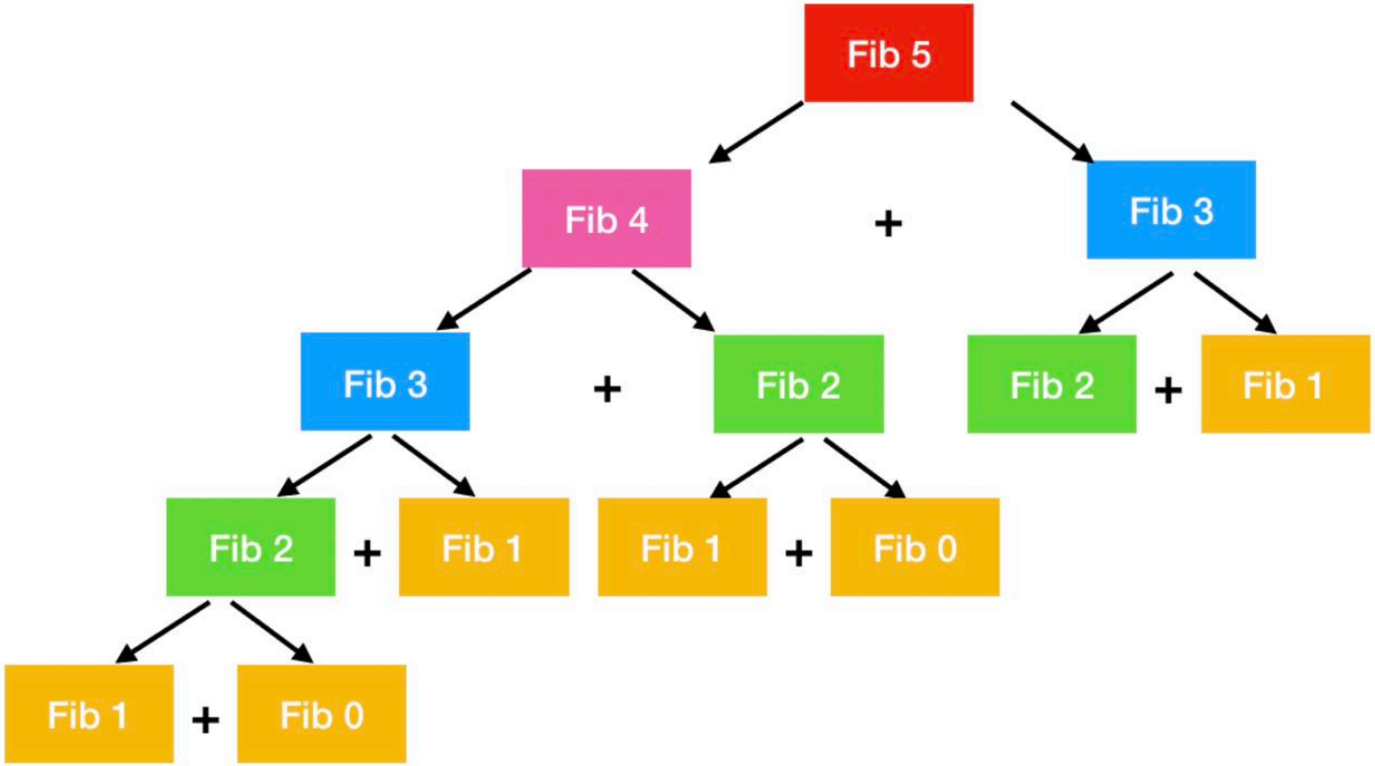
```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

  return Fib(n-1) + Fib(n-2);
}
```

It has a serious issue!

Recursion tree



What’s the problem?

Many subproblems are overlapping: a lot of recomputation

Complexity: $O(2^n)$

What is Dynamic Programming (DP)?

$\boxed{\text{Dynamic Programming = Recursion + Memoization}}$

- Recursion: problem can be solved recursively
- Memoization: Store optimal solutions to sub-problems in table (or memory or cache)

What is Dynamic Programming (DP)?

- It is similar to divide-and-conquer strategy
  - Breaking the big problem into sub-problems
  - Solve the sub-problems recursively
  - Combining the solutions to the sub-problems

- What is the difference between them?
  - DP can be applied when the sub-problems are not independent
    - Every sub-problem is solved once and is saved in a table
  - The problem usually can have multiple optimal solutions
    - DP may just return one of them
- If the sub-problems are independent, DP is not useful!

Dynamic Programming Approaches

- Top-down approach
  - Recursively using the solution to its sub-problems
  - Memoize the solutions to the sub-problems and reuse them later

- Bottom-up approach
  - Figure out the order of calculation
  - Solve the sub-problems to build up solutions to larger problem

Fibonacci: Top-down approach

```
Fib(n)
{
   if (n == 0)
            M[0] = 0; return 0;
   if (n == 1)
            M[1] = 1; return 1;

   if (M[n-1] == -1)                //F(n-1) was not calculated
            M[n-1] = Fib(n-1)       //calculate F(n-1) and store in M

   if (M[n-2] == -1)                //F(n-2) was not calculated
            M[n-2] = Fib(n-2)       //calculate F(n-2) and store in M

  M[n] = M[n-1] + M[n-2]
  return M[n];
}
```

Store an array M

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

Complexity: O(n)

Fibonacci: Bottom-up approach

```
Fib(n)
{
   M[0] = 0;
   M[1] = 1;

   int i = 0;
   for (i = 2; i<=n; i++)
        M[i] = M[i-1] + M[i-2];
   return M[n];
}
```

Store an array M

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

Complexity: O(n)

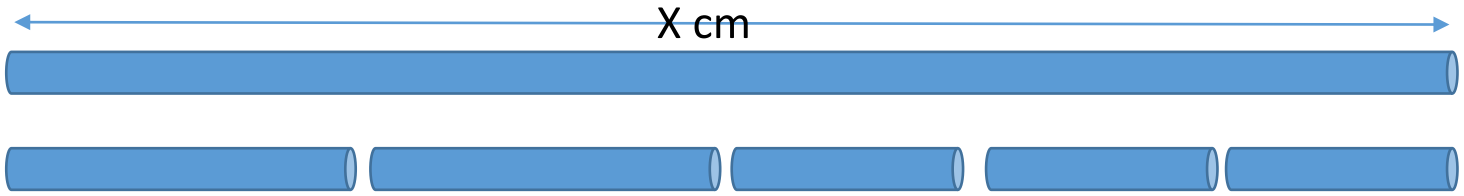
Other examples of DP

- String algorithms like longest common subsequence, longest increasing subsequence, longest common substring etc.
- Graph algorithms like Bellman-Ford algorithm, Floyd’s algorithm
- Chain matrix multiplication

- Rod Cutting
- 0/1 Knapsack
- Travelling salesman problem
- Subset Sum

Rod Cutting Problem

Given a rod of a certain length and price of rod of different lengths, determine the maximum revenue obtainable by cutting up the rod at different lengths based on the prices.



Length cm	1	2	3	4	5	6	7	8	9
Price $	1	5	8	9	10	17	17	20	24

Rod Cutting Problem

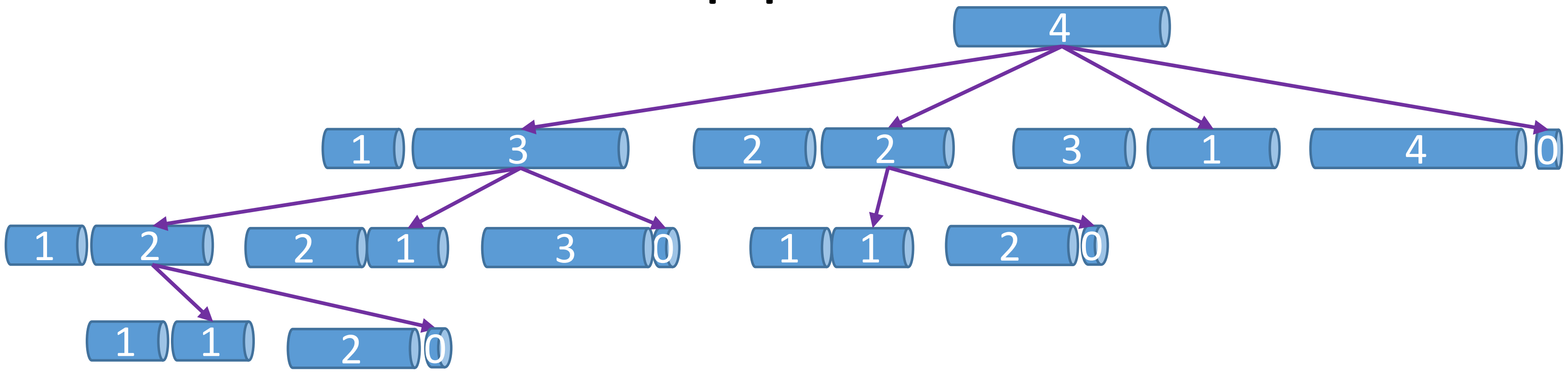
Length cm	1	2	3	4	5	6	7	8	9
Price $	1	5	8	9	10	17	17	20	24

If a rod of length 4,

Length of each piece	Total Revenue
4	9
1 + 3	1+8 = 9
1 + 1 + 2	1+1+5 =7
1 + 1 + 1 + 1	1+1+1+1=4
2 + 2	**5+5 =10**

From all possible solutions, the maximum revenue is 10 by cutting the rod into two pieces of length 2 each.

Naïve Top-down Recursive Approach




```
Cut-Rod (p,n)
begin
    if n==0
        return 0
    q ← -∞
    for i = 1 to n do
        q ← max (q, p[i] + Cut-Rod(p, n-i))
    return q
end
```

The recursive calls will repeatedly find the revenue for a rod of the same length. Its time complexity is $\Theta(2^n)$

Top-down Memoized Approach

- The result of each sub-problem is stored and reused

```
Cut-Rod (p,n)
begin
        r[1,…,n] ← {0}
        return Mem-Cut-Rod-Aux(p,n,r)
end
```

```
Mem-Cut-Rod-Aux (p,n,r)
begin
        if n==0
                return 0
        if(r[n]>0)
                return r[n]
        else
            q ← -∞
            for i = 1 to n do
                q ← max (q, p[i] + Mem-Cut-Rod-Aux(p, n-i, r))
            r[n] ← q
        return q
end
```

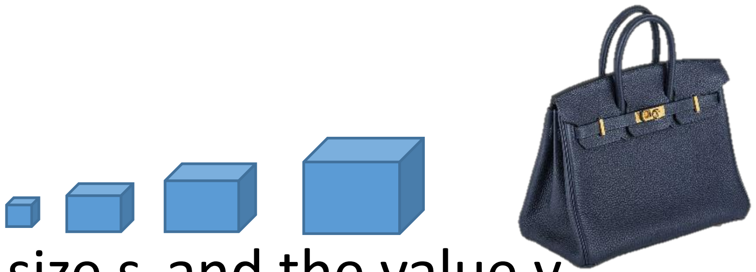
Bottom-up DP Approach

```
DP-Cut-Rod (p,n)
begin
        r[1, …, n] ← {0}
        for j = 1 to n do
            for i = 1 to j do
                 r[j] ← max (r[j], p[i] + r[j-i])
        return r[n]
end
```

- The bottom-up and top-down versions has the same asymptotic running time, $\Theta(n^2)$

Length cm	1	2	3	4	5	6	7	8	9
Price $	1	5	8	9	10	17	17	20	24
Max Rev $	1	5	8	10	13	17	18	22	25

0/1 Knapsack



- Given *n* items, where the i^{th} item has the size s_i and the value v_i
- Put these items into a knapsack of capacity *C*

- *Optimization problem: Find the largest total value of the items that fits in the knapsack*

$$\max_x \sum_{i=1}^{n} v_i x_i$$

Subject to

$$\sum_{i=1}^{n} s_i x_i \leq C$$

$$x_i \in \{0,1\} \qquad\qquad i = 1, 2, \ldots, n$$

0/1 Knapsack

$$\max_x \sum_{i=1}^{n} v_i x_i$$

Subject to

$$\sum_{i=1}^{n} s_i x_i \leq C$$

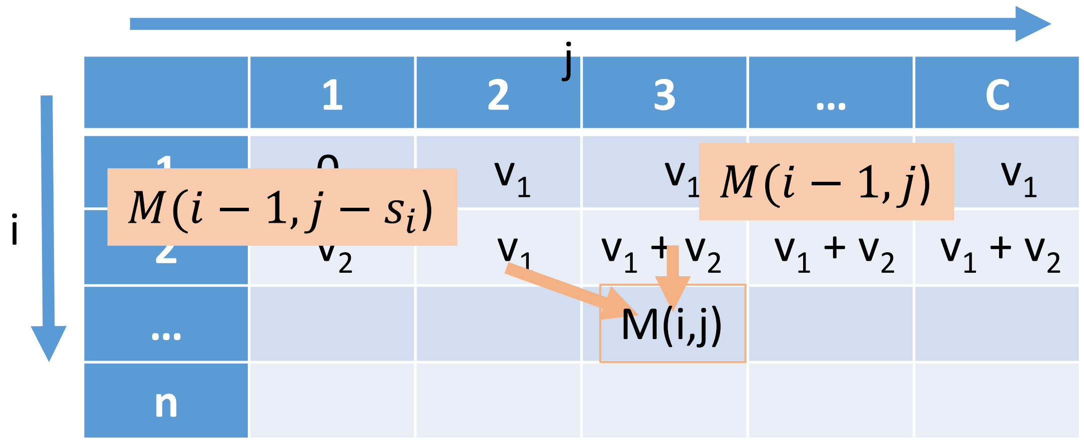
$$x_i \in \{0,1\} \qquad i = 1, 2, \dots, n$$

- Brute-force algorithm
- The i^{th} item is either included (1) or excluded (0)
- The time complexity of the algorithm is $\Theta(2^n)$

Item 1	Item 2	Item 3	Value
0	0	0	0
0	0	1	V3
0	1	0	V2
0	1	1	V2+V3
1	0	0	V1
1	0	1	V1+V3
1	1	0	V1+V2
1	1	1	V1+V2+V3

Can you see that some sub-problems are overlapping?

Using DP to solve 0/1 Knapsack



- The recursive formula
  - $M(i,j) = \max\{\underbrace{M(i-1,j)}_{\text{ith item is unused}},\ \underbrace{M(i-1,j-s_i)+v_i}_{\text{ith item is used}}\}$
    - i = 1, … n
    - j = 1, … C

Item	Weight	Value
1	2kg	$12
2	1kg	$10
3	3kg	$20
4	2kg	$15

The capacity of knapsack is 5kg. (C = 5)

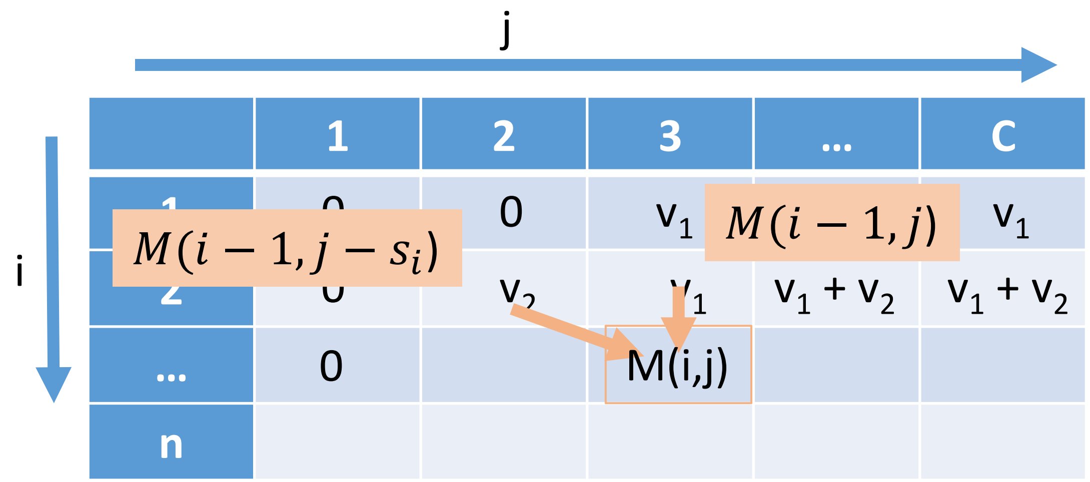
Capacity

Item

i \ j	1	2	3	4	5
1	$0	$12	$12	$12	$12
2	$10	$12	$22	$22	$22
3	$10	$12	$22	$30	$32
4	$10	$15	$25	$30	**$37**

Using DP to solve 0/1 Knapsack

- The recursive formula
  - $M(i,j) = \max\{\underbrace{M(i-1,j)}_{\text{ith item is unused}},\ \underbrace{M(i-1,j-s_i)+v_i}_{\text{ith item is used}}\}$
    - i = 1, … n
    - j = 1, … C
- Create a n-by-C matrix, M
- All the possible sizes from 1 to C



- Bottom up approach
- Time Complexity is $\Theta(nC)$

Summary

- Dynamic Programming
  - Rod Cutting Problem
  - 0/1 Knapsack Problem


Q&A