SC1007
Data Structures and Algorithms

Dynamic Programming

Dr Liu Siyuan (syliu@ntu.edu.sg)

N4-02C-117a

Office Hour: Mon & Wed 4-5pm

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 looks like:

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

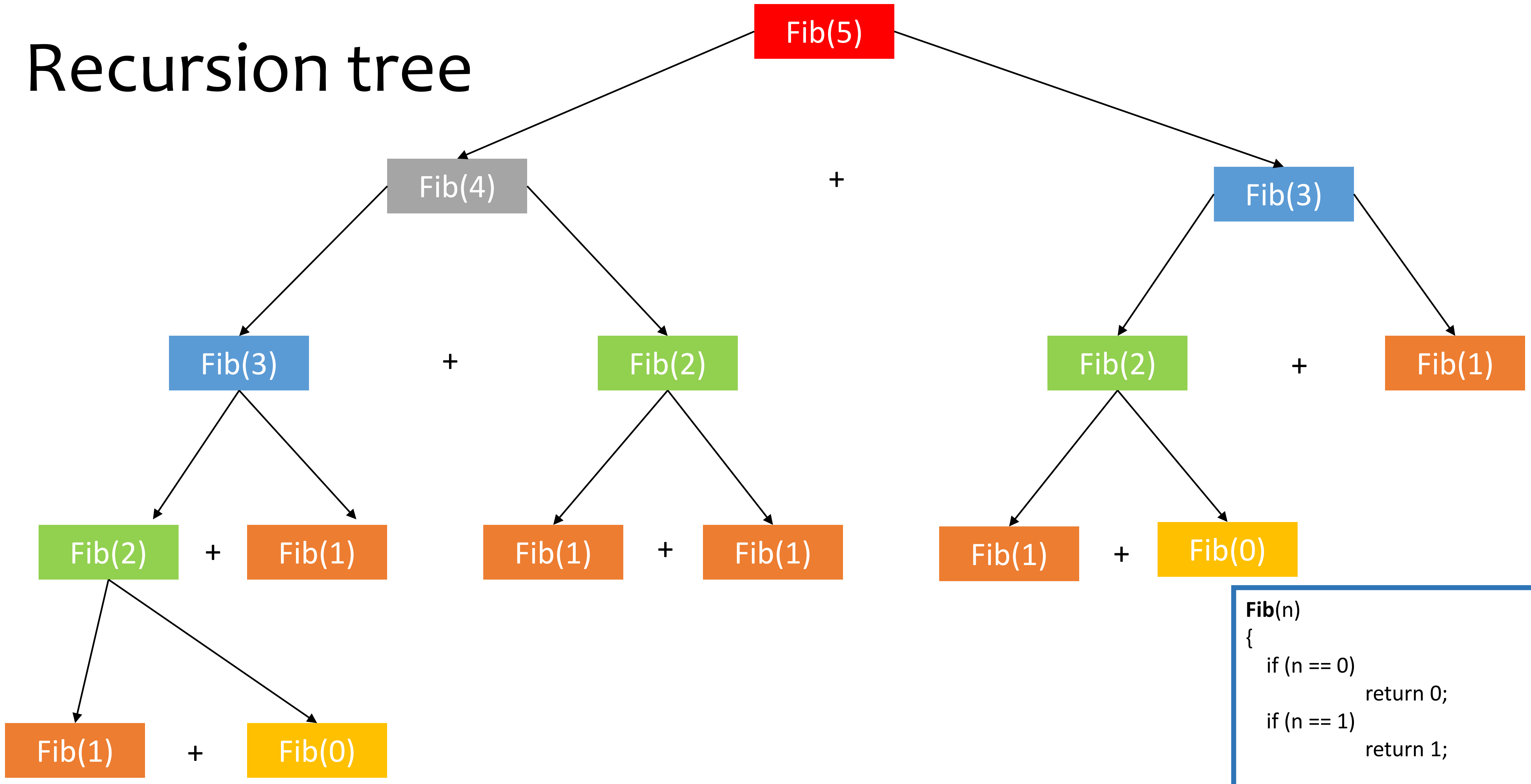
Fibonacci Sequence

```
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




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

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

Fibonacci Sequence

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

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

- It has a serious issue!
  - Many subproblems overlap: a lot of re-computations.
  - The complexity is $O(2^n)$

Dynamic Programming

by Richard Ernest Bellman in 1953

What is Dynamic Programming (DP)?

- It is not a programming language like C
  - The term “Programming” refers to a tabular method (filling tables)
    - It is applied to optimization problems
      - Other “programming” methods in mathematical optimization are
        - Linear Programming
        - Integer Programming
        - Convex Programming
        - Semidefinite Programming
      - not related to coding
- Applied from system control to economics

What is Dynamic Programming (DP)?

Dynamic Programming = Recursion + Memoization

- Recursion: problem can be solved recursively
- Memoization: Store optimal solutions to sub-problems in table (or memory or cache) => If the sub-problems are independent, DP is not useful!
- Optimal substructure
  - Combination of optimal solutions to its sub-problems
- Overlapping sub-problems
  - Having the same sub-problems

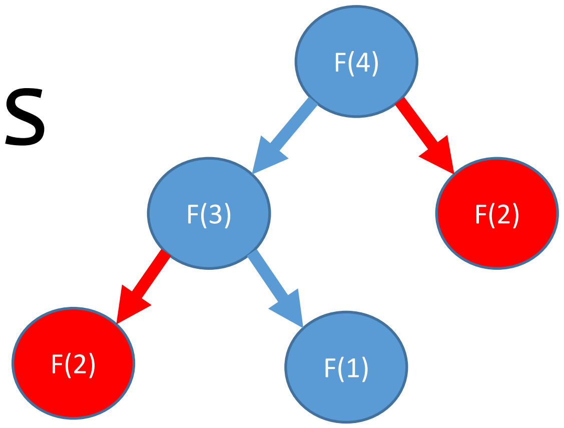
The term "memoization" was coined by Donald Michie in the 1960s, and it is derived from the Latin word "memorandum," which means "to be remembered." Michie used the term to distinguish the technique of caching function results from the more general concept of caching in computer science.

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

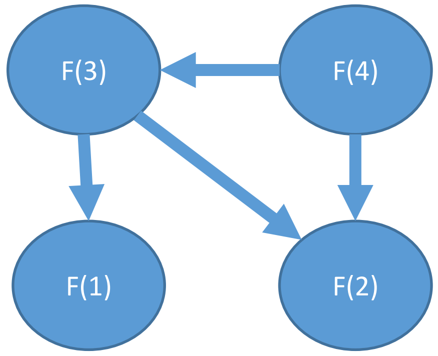
- What is the difference between them?
  - DP can be applied when the sub-problems are not independent
    - Every sub-problem is solved once and is saved in a table
  - The problem usually can have multiple optimal solutions
    - DP may just return one of them

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

Fib(6)->Fib(5)->Fib(4)->Fib(3)->Fib(2)->Fib(1), Fib(0)

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

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

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];
}
```

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

Return to Fib(2)

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

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];
}
```

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

Return to Fib(3)

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

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];
}
```

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

Return to Fib(4)

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

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];
}
```

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

Return to Fib(5)

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

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];
}
```

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

Return to Fib(6)

0	1	2	3	4	5	6
0	1	1	2	3	5	8

Complexity: Θ(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

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

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

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

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];
}
```

i=4

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

i=5

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

i=6

0	1	2	3	4	5	6
0	1	1	2	3	5	8

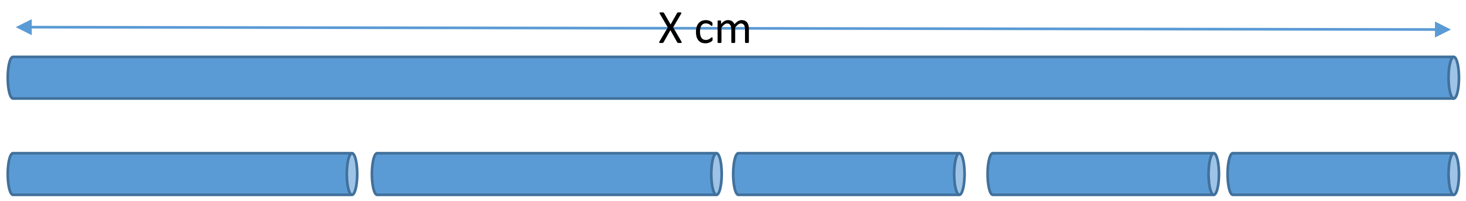
Complexity: Θ(n)

Examples of DP

- String algorithms like longest common subsequence, longest increasing subsequence, longest common substring etc.
- Graph algorithms like Floyd’s algorithm
- Chain matrix multiplication
- Rod Cutting
- 0/1 Knapsack
- Travelling salesman problem
- Subset Sum
- Useful resource: https://algorithm-visualizer.org/

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

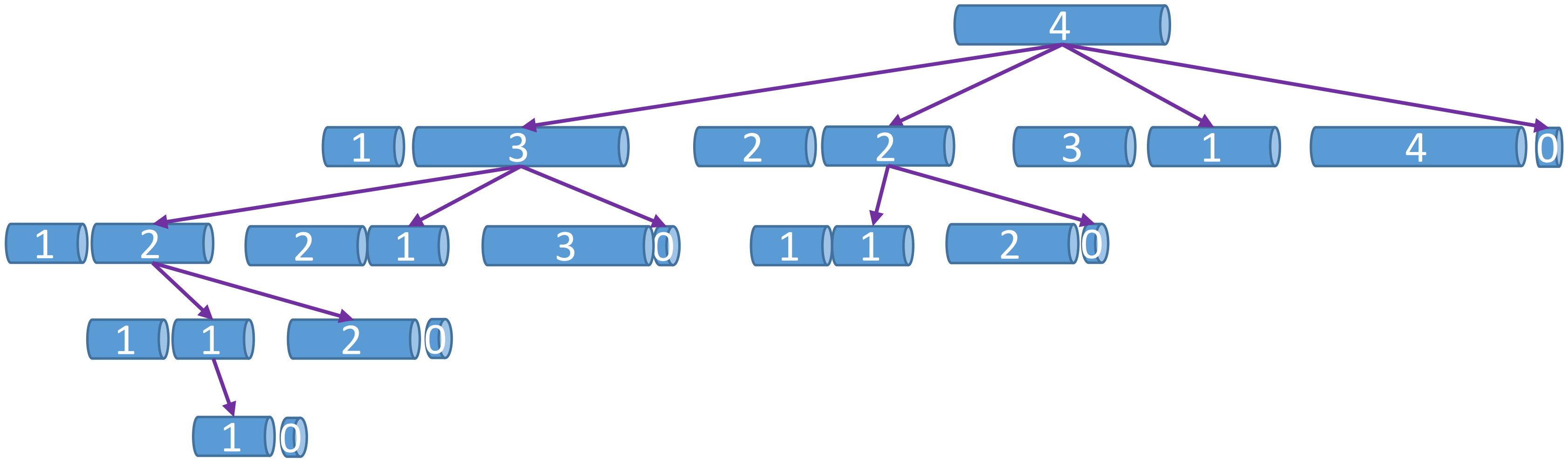
Length cm	1	2	3	4	5	6	7	8	9
Price $	1	5	8	9	10	17	17	20	24

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 Algorithm

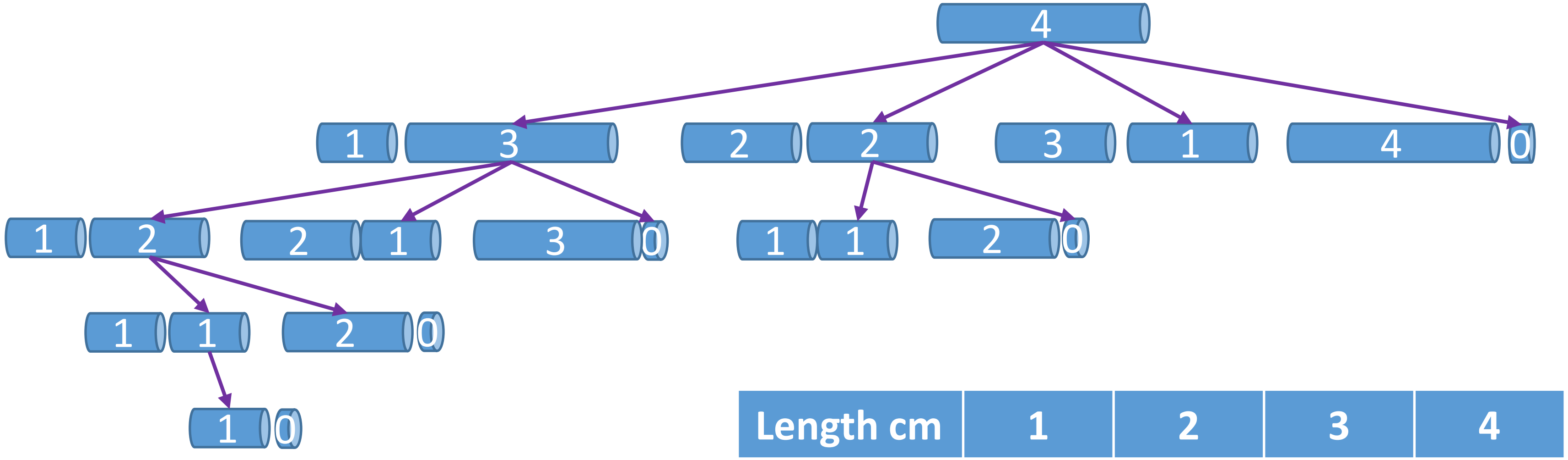



```
Cut-Rod (p,n)
begin
      if n==0
            return 0
      q ← -∞
      for i = 1 to n do
            q ← max (q, p[i] + Cut-Rod(p, n-i))
      return q
end
```

Length cm	1	2	3	4
Price $	1	5	8	9

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

Top-down DP Approach



Length cm	1	2	3	4
Price $	1	5	8	9

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

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

```
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[0, …, 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
```

r	0	1	2	3	4
	0	0	0	0	0

Length cm	1	2	3	4
Price $	1	5	8	9

Bottom-up DP Approach

```
DP-Cut-Rod (p,n)
begin
        r[0, …, 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
```

r	0	1	2	3	4
j=1, i=1	0	0 1	0	0	0

Length cm	1	2	3	4
Price $	1	5	8	9

Bottom-up DP Approach

```
DP-Cut-Rod (p,n)
begin
        r[0, …, 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
```

r	0	1	2	3	4
j=2, i=1	0	1	2	0	0

Length cm	1	2	3	4
Price $	1	5	8	9

Bottom-up DP Approach

```
DP-Cut-Rod (p,n)
begin
        r[0, …, 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
```

r	0	1	2	3	4
j=2, i=2	0	1	5	0	0

Length cm	1	2	3	4
Price $	1	5	8	9

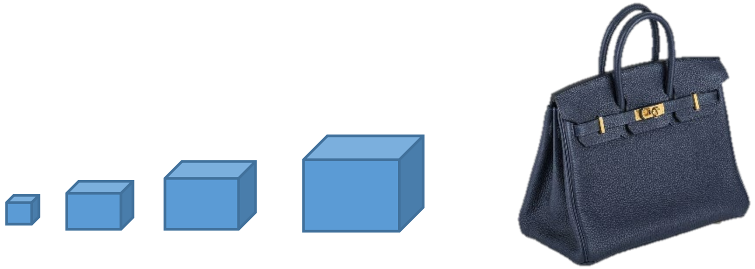
Bottom-up DP Approach

```
DP-Cut-Rod (p,n)
begin
        r[0, …, 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 weight 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 fit in the knapsack

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

Subject to

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

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

0/1 Knapsack

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

Subject to

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

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

- 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?

The recursive implementation

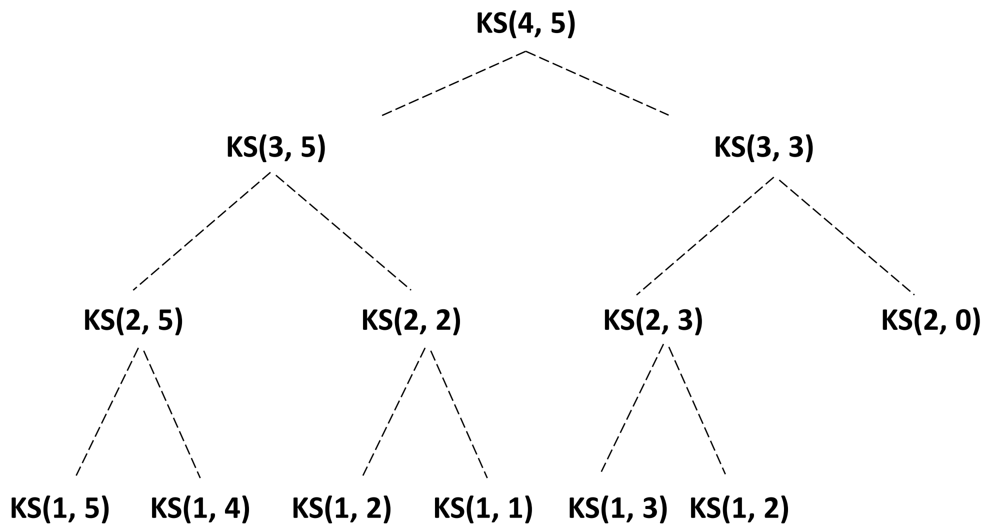
```
KS(i, j){
if (j<=0) return 0
if (i == 1)
        if  (s_i<=j)
             return v_i
        else return 0
else
        if (j - s_i<0)
          return KS(i - 1, j)
        else
          return
```

$$\max\{\underbrace{KS(i-1,j)}_{\text{ith item is unused}},\ \underbrace{KS(i-1,j-s_i)+v_i}_{\text{ith item is used}}\}$$

```
}
```

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

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



The recursive implementation

KS(i, j){

if (j<=0) return 0

if (i == 1)

    if (s_i<=j)

        return v_i

    else return 0

else

    if $(j - s_i$<0)

      return $KS(i-1, j)$

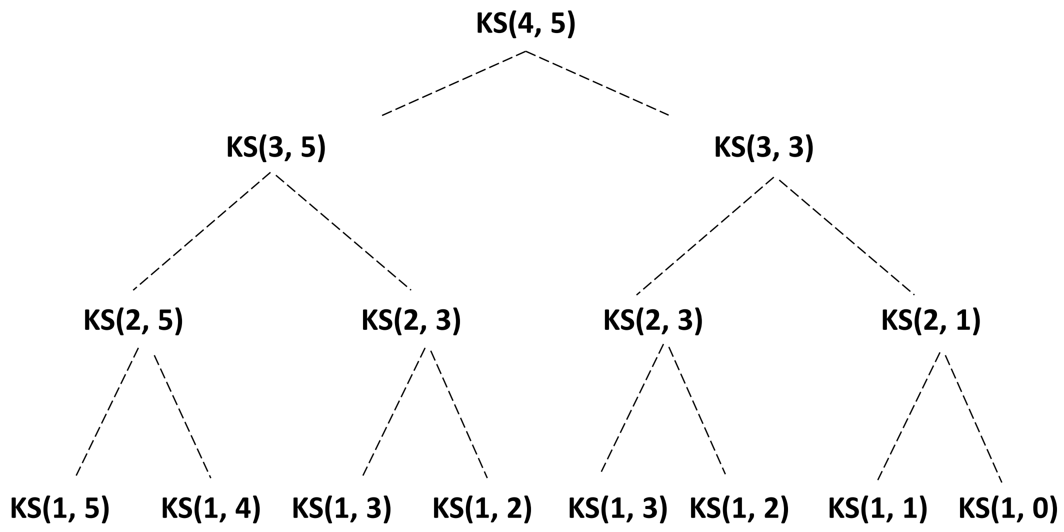
    else

      return $\max\{\underbrace{KS(i-1,j)}_{\text{ith item is unused}},\ \underbrace{KS(i-1, j-s_i) + v_i}_{\text{ith item is used}}\}$

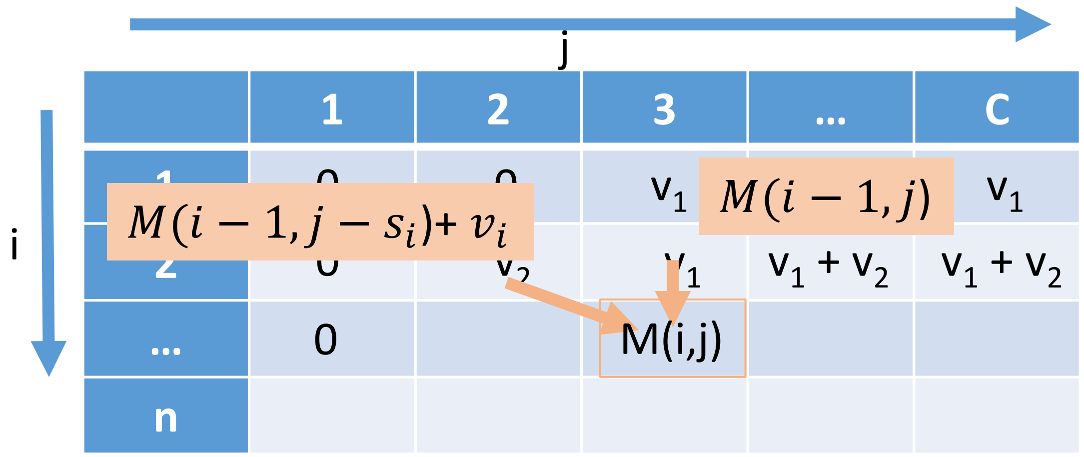
}

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

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



```
DPKS(n, C, m ){
    for j from 1 to C do
        if s[1]<j  m[1, j] = v[1]
    end for
    for i from 2 to n do
        for j from 1 to C do
                if j >= s[i]
                        m[i, j] = max(m[i-1, j], m[i-1, j-s[i]] + v[i])
                else m[i, j] = m[i-1, j]
        end if
      end for
    end for
}
```



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

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

i = 1, ... n

j = 1, ... C

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 $\Theta(nC)$

j

i

	1	2	3	…	C
1	0	0	v_1		v_1
2	0	v_2	v_1	$v_1 + v_2$	$v_1 + v_2$
…	0		M(i,j)		
n					

$M(i-1,j-s_i)+v_i$

$M(i-1,j)$

Using DP to solve 0/1 Knapsack

```
DPKS(i, j, m){
if m[i][j]>=0 return m[i][j]
if (j<=0) return 0
if (i == 1)
        if  (s_i<=j)
                m[i][j]=v_i
                return v_i
        else
                m[i][j]=0
                 return 0
else
        if (j - s_i<0)
                m[i][j] = DPKS(i-1,j,m)
                return m[i][j]
        else
                m[i][j]= max{DPKS(i-1,j), DPKS(i-1,j-s_i)+v_i}
                return m[i][j]
}
```

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

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
2		$12			$22
3					$32
4					

Summary

- Dynamic Programming

  Dynamic Programming = Recursion + Memoization

  - Recursion: problem can be solved recursively
  - Memoization: Store optimal solutions to sub-problems in table (or memory or cache) => If the sub-problems are independent, DP is not useful!
- Examples
  - Fibonacci sequence
  - Rod Cutting Problem
  - 0/1 Knapsack Problem

Problems for you to think about

- Longest palindromic substring: Given a string *s*, return the longest palindromic substring in *s*, e.g.., *s* = “babad”, “bab” and “aba” are the answers.

- Jump game: You are given an integer array *nums*. You are initially positioned at the array’s first index, and each element in the array represents your maximum jump length at that position. Return true if you can reach the last index, or false otherwise.

- Climbing stairs: You are climbing a staircase. It takes *n* steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?