CIS 606 Analysis of Algorithms

Dynamic Programming

RATIONALE

- Dynamic Programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.
- A dynamic-programming algorithm solves each subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subproblem, while the divide-and-conquer solves the same subproblem multiple times.

OBJECTIVES

- Understand the structure of dynamic programming.
- Understand why dynamic programming solves subproblem once

PRIOR KNOWLEDGE

- **Divide-and-conquer**
- **Recursion tree**
- **The structure of dynamic programming: define subproblems and decide the dependency.**

FIBONACCI NUMBER

- Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8,…
- The first two fibonacci numbers are f(0) = 0 and f(1)=1, respectively.
- The fibonacci number f(i) is computing the i-th fibonacci number defined as f(i) = f(i-1) + f(i-2)

f(0)	f(1)	f(2)	f(3)	f(i)	f(5)	f(6)	f(7)	f(8)	f(9)
0	1	1	2	3	5	8	13	21	34

FIBONACCI NUMBER—DIVIDE-AND-CONQUER

- Input: an integer n>0
- Output: the fibonacci number f(n)
- We know: f(n) = f(n-1) + f(n-2) for n>1; f(0)=0, f(1)=1.
  - Divide-and-Conquer: f(n)
  - Divide:
    - Subproblem 1: computing f(n-1)
    - Subproblem 2: computing f(n-2)
  - Conquer:
    - x = f(n-1)
    - y = f(n-2)
  - Combine: compute f(n) =x+y return x+y

f(n-1) and f(n-2) are overlapping subproblems is because: f(n-1)=f(n-2)+f(n-3) indicates that computing f(n-1) requires solving f(n-2) ahead.

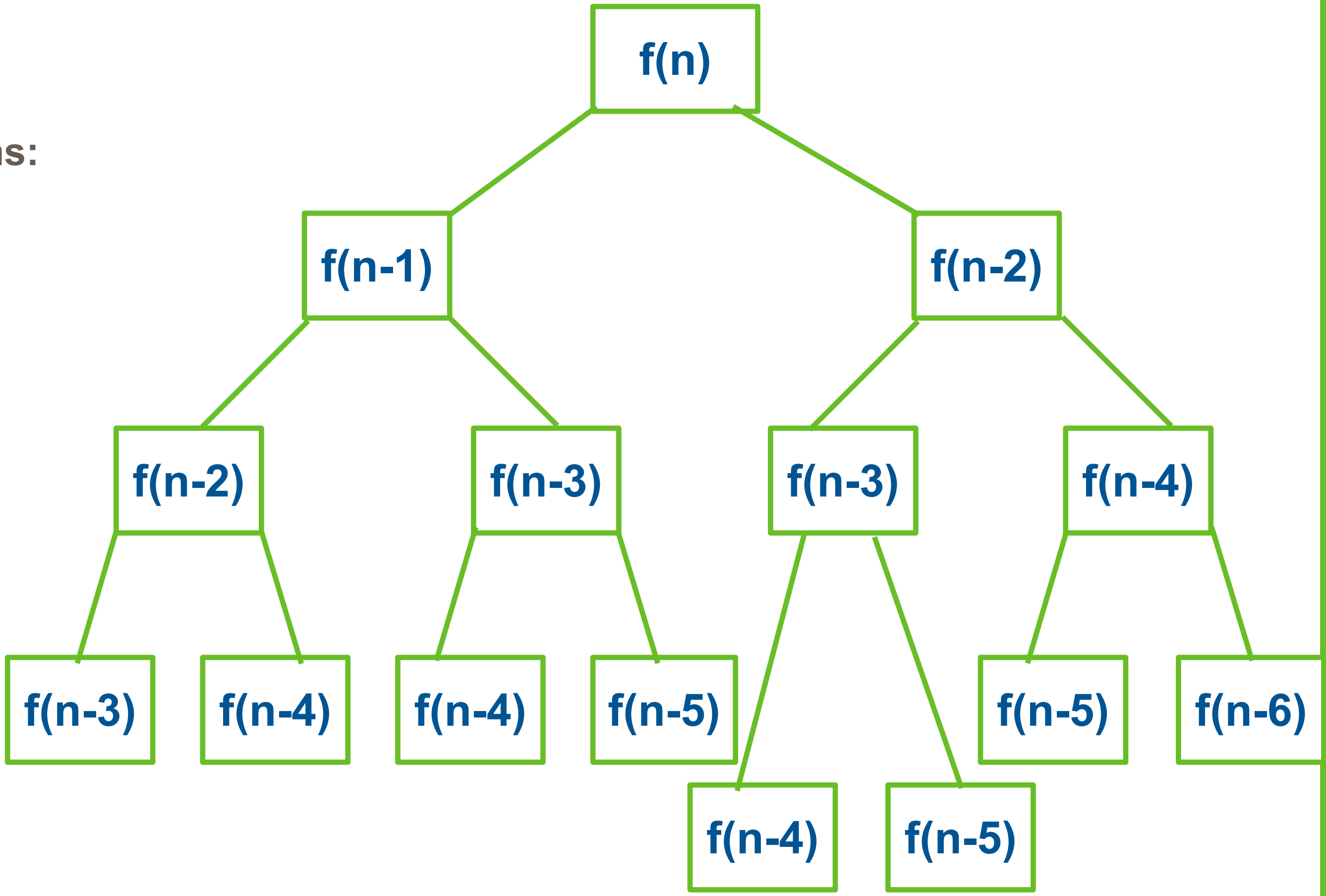
FIBONACCI NUMBER—DIVIDE-AND-CONQUER

```
f(i-1) is the subproblem to compute the i-1-th F number
f(i-2) is the subproblem to compute the i-2-th F number

f(i): is to compute the i-th F number. Input is I
    f(i) = f(i-1)+f(i-2)

f(i)
{
    Base case: if i = 1 return 0
                      if I = 2 return 1
    Divide: divide i into i-1 and i-2
    conquer: recursively solve f(i-1) and f(i-2)
                 x=f(i-1)
                 y=f(i-2)
    Combining step:
             Return x+y
}
```

FIBONACCI NUMBER(CONT)

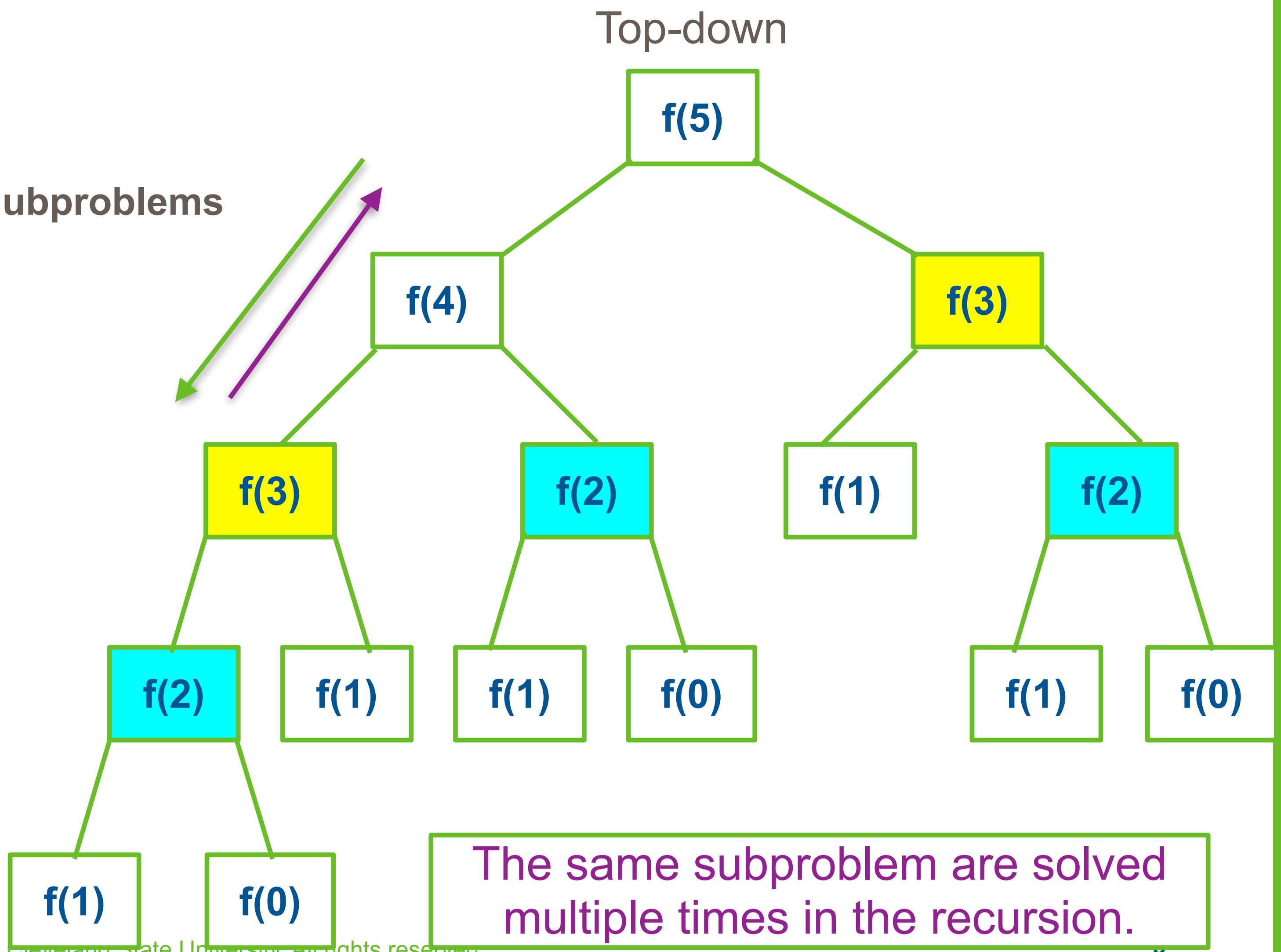
The same problem f(n-4) is solved four times so far

- Input: an integer n>0
- Output: the fibonacci number f(n)
- Divide-and-Conquer for overlapping subproblems:
- main()
- {
-     Int n=1000;
-     std::cout<<f(n)<<endl;
- }
    - f(n)
    - {
    -     If n =0 return 0;
    -     If n=1 return 1;
    -     x = f(n-1)
    -     y = f(n-2)
    -     return x+y
    - }



A PROBLEM WITH DIVIDE-AND-CONQUER

- **Input: an integer n>0**
- **Output: the fibonacci number f(n)**
- **Divide-and-Conquer for overlapping subproblems**
    - **f(n)**
    - **{**
    - **if n=0 return 0**
    - **if n=1 return 1**
    - **x = f(n-1)**
    - **y = f(n-2)**
    - **return x+y**
    - **}**
    - $O((\frac{\sqrt{5}+1}{2})^n)$



WHY NOT SOLVING SMALL SUBPROBLEMS AHEAD —— DYNAMIC PROGRAMMING

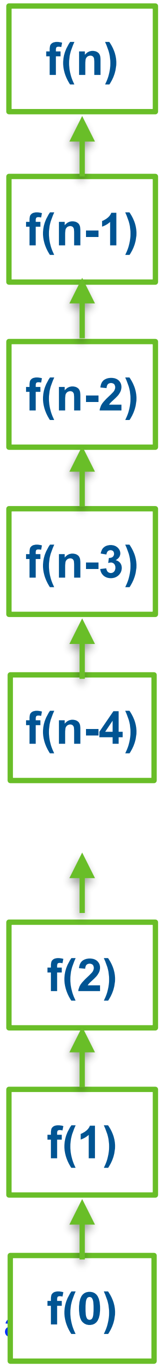
Solve subproblems in a bottom-up order and store solutions of subproblems in a table to solve the original problem.

- Input: an integer n>0
- Output: the fibonacci number f(n)
- Subproblems:f(0), f(1), f(2), …,f(i), 0<=i<=n
- Dependency relations:
  - How do you compute f(n) if f(0), f(1), …, f(n-1) are known?
  - f(n) = f(n-1) + f(n-2)
- Table:

f(0)	f(1)	f(2)	f(3)	f(4)	…	f(n-2)	f(n-1)	f(n)	…
0	1	1	2	3	…	13	21	34	…

DYNAMIC PROGRAMMING

- Input: an integer n>0
- Output: the fibonacci number f(n)
    - f(n)
    - {
    -    Create an array A[0…n] to store sub-solutions
    -   f(0), f(1), f(2), …, f(n);
    -    A[0]=0; A[1]=1; //**base case**
    -    for i = 2 to n // f(2), f(3), f(4), f(5), f(6), … f(n-1), f(n) in order
    -        A[i] = A[i-1]+A[i-2]   //solve subproblems in order based
    -   on the dependency
    -    return F(n)
    - }. O(n)



Dependency:

f(i) = f(i-1) + f(i-2)

Bottom-up

Assume solution for f(0), f(1), f(2), f(3), …, f(n-1) are known.

They are in the table A[0,…, n]   A[i] is the solution for f(i)

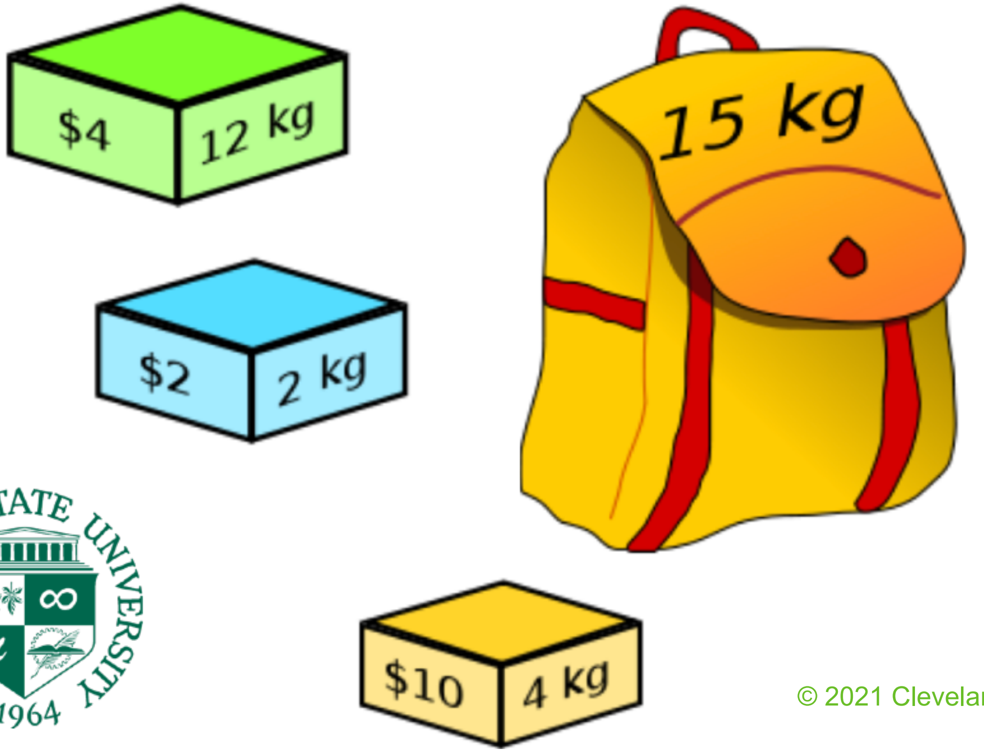
f(n) = f(n-1) + f(n-2)

WHEN I SOLVE f(i) solutions for f(0), f(1), f(2) ,f(3) …f(i-1) are known and they are in A[0…i-1] a fixed.

KNAPSACK PROBLEM

Input: a knapsack of size M and

n items where each item a_i has a weight s_i and a value p_i.



Goal: Pack the knapsack to maximize the total value but the total weight of packed items is no more than M.

KNAPSACK PROBLEM

Each item is accepted or rejected

Knapsack Size 30

{1, 2, 3, 4, 5, 7}

Maximize packed-item values
but total size <=30

Items	Weight	Value
1	1	1
2	4	4
3	4	3
4	2	4
5	3	5
6	12	7
7	15	9

The total size is 29 but the total value is 26.

KNAPSACK PROBLEM

Brute-force Approach:
Try all combinations and find the one with the maximum value and with the total size no more than 30.

Knapsack Capacity 30

Items	Weight	Value
1	1	1
2	4	4
3	4	3
4	2	4
5	3	5
6	12	7
7	15	9

It works but the time complexity is $O(2^n)$

Combination:
{7, 6, 5} value = 21 size = 30, given by the greedy algorithm

KNAPSACK PROBLEM

Greedy Algorithm:
Always pack items with largest value first

Knapsack Size 30

Items	Weight	Value
1	1	1
2	4	4
3	4	3
4	2	4
5	3	5
6	12	7
7	15	9

The total size is 30 but the total value is 21

KNAPSACK PROBLEM

Subproblem: Knapsack(A[1…i], j) —— Pack the bag with size j < M by the first i items 1, 2, 3, …, i:
Original problem: Pack the bag of size M by n items 1, 2, 3, …, n —— Knapsack(A[1…n], M).

Items	Weight	Value
1	1	1
2	4	4
3	4	3
4	2	4
5	3	5
6	8	7
7	15	9

Knapsack Capacity 30

input: A[1…n] to represent n items: A[i] is for item i   capacity = M

A[i] is a pair<value, weight/size>

(1) Define the problem: Knapsack(A[1…i], int j) given i items and a bag of size capacity j.

(2) How many subproblems we need to solve. M[0…n][0…M].

(3) M[i][j]. ———- correspond subproblem Knapsack(A[1…i], j)

Knapsack(A[1…i], int j) // I have i items and capacity is j.

Pack the bag with capacity j by i items 1, 2, 3, …, I

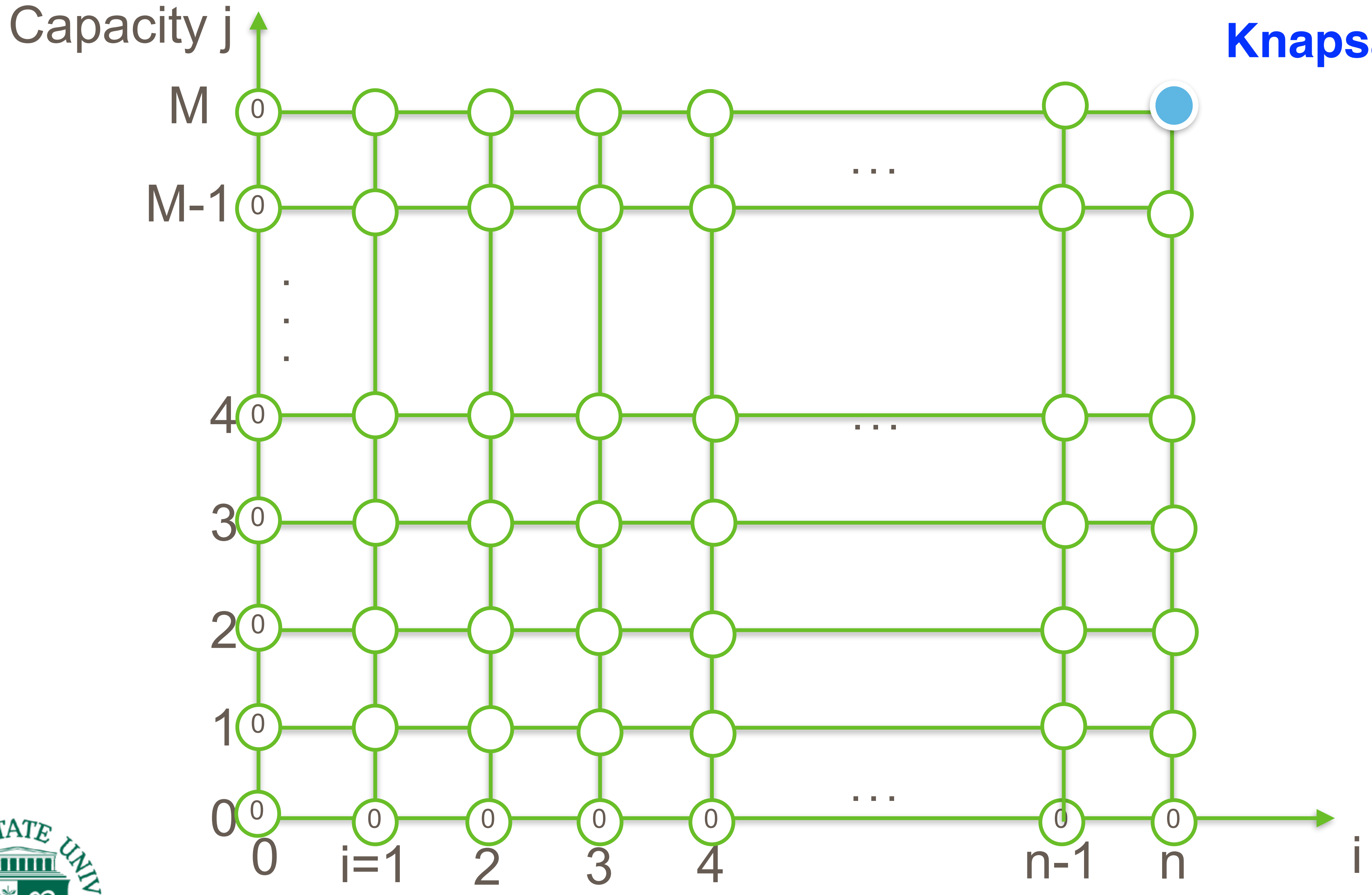
(4) compute the dependency for Knapsack(A[1…i], int j)

(4) M[i][j] = 0 if i = 0 or j = 0 the base case solutions// make a record of solutions for knapsack base;

(4) solve Knapsack(A[1…i], j) with i > 0 and j > 0.

Dependency: Knapsack(A[1…i], j) =max (Knapsack(A[1…i-1], j), Knapsack(A[1…i], j-1))

Knapsack(A[1…i], j)



HOW MANY PROBLEMS TO BE SOLVED? (n+1)(M+1)

HOW MANY SUBPROBLEMS ARE THERE?

Subproblems:

Knapsack(Item[1…i], j): The maximum value of packing size j <= M by the first i items[1, 2, …, i]

	j=0	j=1	…	j=M-1	j=M
i=0	Knapsack (item[], 0)	Knapsack (item[], 1)	…	Knapsack (item[], M-1)	Knapsack (item[], M)
i=1	Knapsack (item[1], 0)	Knapsack (item[1], 1)	…	Knapsack (item[1], M-1)	Knapsack (item[1], M)
i=2	Knapsack (item[1…2], 0)	Knapsack (item[1…2], 1)	…	Knapsack (item[1…2], M-1)	Knapsack (item[1…2], M)
⋮	⋮	⋮	⋮	⋮	⋮
i=n	Knapsack (item[1…n], 0)	Knapsack (item[1…n], 1)	…	Knapsack (item[1…n], M-1)	Knapsack (item[1…n], M)

DEPENDENCY

If Item[i]'s weight is too large to pack, i.e., Weight(Item[i]) > j

==== then we pack the bag of size j by selecting from items[1…i-1]

Knapsack(Item[1…i], j) = Knapsack(Item[1…i-1], j)

DEPENDENCY(CONT)

If Item[i]'s weight is not so large, i.e., Weight(Item[i]) <= j
==== then we have the following two cases for Knapsack(Item[1…i], j) solution and the larger one is the optimal solution

Case 1: Item[i] is not selected for Knapsack(Item[1…i], j)
==== packing bag of size j by selecting from items[1…i-1]

Knapsack(Item[1…i], j) = Knapsack(Item[1…i-1], j)

Case 2: Item[i] is selected for Knapsack(Item[1…i], j)
==== item(i) + pack bag of capacity [j - size(i)] by selecting from items[1…i-1]

Knapsack(Item[1…i], j) = value(i) + Knapsack(Item[1…i-1], j-weight(i))

Dependency: Knapsack(Item[1…i], j) =
max{Knapsack(Item[1…i-1], j), Knapsack(Item[1…i-1], j-weight(i)) + value(i)}

KNAPSACK PROBLEM EXAMPLE

Items(value, size):
1: (5, **10**)
2: (4, 4)
3: (6, 3)
4: (3, 5)

Dependency: Knapsack(i=2, j=4)
max{Knapsack(i=1, 4) = 0, 4+ Knapsack(1, 0)}

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	j=9	j=10
i=0	0	0	0	0	0	0	0	0	0	0	0
i=1	0	0	0	0	0	0	0	0	0	0	5
i=2	0	0	0	0	4	4	4	4	4	4	5
i=3	0	0	0	6	6						10
i=4	0										

DYNAMIC PROGRAMMING

Subproblem:

Knapsack(i=0 to n, 0)=0 K[i][0]= 0: Pack size j <= M from the first i items [1, 2, …, i]

Dependency: Knapsack(Item[1…i], j) =
max{Knapsack(Item[1…i-1], j), value(i) + Knapsack(Item[1…i-1], j-size(i))}

```
Knapsack(Item[1…n], M){
    Create a (n+1) × (M+1) matrix M[0…n][0…M].  //O(Mn)
    for j = 0 to M
        M[0][j] = 0      //base case: pack bag of size j with 0 items.
    for i = 0 to n
        M[i][0] = 0      //base case: pack bag of size 0 with i items.
    for i = 1 to n
        for j = 1 to M
            if Item[i].size <= j
                M[i][j] = max(M[i-1][j], Item[i].value + M[i][j-Item[i].size])
            else    M[i][j] = M[i-1][j]
    return M[n][K]
```

Maximum Subarray

- Input: an array A of n integers >0
- Output: a contiguous subarray of A which has the maximum sum over all contiguous subarrays. MaxSub(A[1…n])
- A = {-1, 0, 2, 1, -4, 5}
- 1. MaxSub(A[1…i=0 to n]): compute the maximum contiguous subarray of array A[1…i].
- 2. I changes from 1 to n: We have O(n) problems to solve
- 3. Base case: MaxSub(A[1..0]) = 0   MaxSub(A[1..1]) = A[1]
- 4. General case: MaxSub(A[1…i]) find the maximum contiguous subarray of A[1…i]
- Case 1: the maximum contiguous subarray of A[1…i] is not ending with A[i], I.e., A[i] is not in the maximum contiguous subarray A[2,…, i-2] of A[1…i]
- drop A[i] from A[1…i].   A[1…i-1] : Is A[2,…, i-2] the maximum c subarray of A[1…i-1] or not
    - MaxSub(A[1…i]) = MaxSub(A[1…i-1])
-

- Case 2: the maximum contiguous subarray of A[1…i] is ending with A[i], I.e., A[i] is not in the maximum subarray A[3,…i-1]+A[i] of A[1…i]
    - MaxSub(A[1…i]) = A[i] + sum of maximum subarray ending with A[i-1] = maximum subarray ending with A[i]
    - i from 1 to n
    - MAXSUB(A[1…i]) = max(MaxSub(A[1…i-1]), maximum subarray ending with A[i]
    -)

```
MaxSub(A[1…n])// O(n)
{
    If n = 1 return A[1]
    M is an array of size n.
    M[i] denote the maximum subarray sum of A[1…i]
    MS[1…n] to the maximum subarray ending with A[i]
    MS[i] denote the maximum subarray sum ending with A[i] of A[1…i]

   For i =1: n
        MS[i] = max{MS[i-1] +A[i] , A[i]}

    For i =1: n
        M[i] = max{MS[i], M[i-1]}

    Return M[n]
}
```

LONGEST COMMON SEQUENCE

Sequences

X is a sequence <x_1, x_2, x_3, …, x_n> of elements over a finite set S.

- e.g., X = programming
- e.g., X = 382429793
- e.g., X = #net@

Subsequences

A sequence Z = <z_1, z_2, z_3, …, z_k> over S is called a **subsequence** of X= <x_1, x_2, x_3, …, x_n> iff Z can be obtained from X by **deleting** elements.

- e.g., X = programming Z = gram   Z = pgm   Z = oam
- e.g., X = 382429793 Z = 3   Z = 49   Z = 2299
- e.g., X = #net@ Z = #@   Z = #net   Z = e@

Common subsequence

X and Y are two sequences over a set S;

Z is a common subsequence of X and Y iff Z is a subsequence of X and also a subsequence of Y.

- e.g., X = algorithms Y= arithmetic Z = ath
- e.g., X = 382429793 Y = 3254346   Z = 324
- e.g., X = #net@ Y = @edu   Z = @

The Longest Common Subsequence Problem

- Input: two sequences X and Y over a set S
- Goal: find the longest common subsequence Z* of X and Y
  - E.g., X = algorithms Y= arithmetic
  - Common subsequences:
  - a, t, m, ar, ai, am, at, ari, ait, art, atm, arit, arith, arithm, …
  - The longest common subsequence of X and Y is “arithm”.

Straightforward Method

- Let X be a sequence of length m and Y be a sequence of length n.

```
for every subsequence z of X        //O(2^m) subsequences
{
    Check whether z is a subsequence of Y;   // O(n) matching
}
Return the longest common subsequence found.
```

Time complexity: $O(n2^m)$

Prefix

- Let X be a sequence $<x_1, x_2, x_3, \ldots, x_n>$
- Prefix: $X_i = <x_1, x_2, x_3, \ldots, x_i>$
  - E.g., X = algorithms
  - Prefix: X_1 = a; X_2 = al; X_3 = alg; X_4 = algo; X_5 = algor;
  - X_6 = algori; X_7 = algorit; X_8 = algorith; X_9 = algorithm;
  - X_{10} = algorithms;

Dynamic Programming

- **Let $X = <x_1, x_2, x_3, \ldots, x_m>$ and $Y = <y_1, y_2, y_3, \ldots, y_n>$ be sequences.**
- **Prefix: $X_i = <x_1, x_2, x_3, \ldots, x_i>$ $\qquad$ $Y_j = <y_1, y_2, y_3, \ldots, y_j>$**
- **L[i][j] is the length of the longest common subsequences of X_i and Y_i**
- ***Subproblem: compute L[i][j]***

E.g., X = algorithms Y= arithmetic

X_5 = algor Y_2= ar L[5][2] = 2

X_6 = algori Y_3= ari x_6=y_3=i L[6][3] = L[5][2] +1 =3

X_5= algor Y_6= arithm X_6= algori Y_6= arithme

L[5][6] = 2 L[6][6] = 3

X_6= algori Y_7= arithme $x_6 \neq y_7$ L[6][7] = L[6][6]

Dynamic Programming

- **Let $X = \langle x_1, x_2, x_3, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, y_3, \ldots, y_n \rangle$ be sequences.**
- **Prefix: $X_i = \langle x_1, x_2, x_3, \ldots, x_i \rangle$ $\qquad$ $Y_j = \langle y_1, y_2, y_3, \ldots, y_j \rangle$**
- **L[i][j] is the length of the longest common subsequences of X_i and Y_i**
- **L[i][j] =**

$$L[i][j] = \begin{cases} 0 & \text{if } i=j=0 \\ L[i-1][j-1] + 1 & \text{if } i, j>0 \;\&\; x_i = y_j \\ \max\{L[i][j-1], L[i-1][j]\} & \text{if } i, j>0 \;\&\; x_i \neq y_j \end{cases}$$

L[i][j] depends on solutions of smaller problems

Problem: compute the length of LCS of X_i and Y_j

Dynamic Programming — Longest Common Subsequence

L[i][j]	X_0 Ø	X_1 B	X_2 A	X_3 C	X_4 B	X_5 A	X_6 D
Y_0 Ø							
Y_1 A							
Y_2 B							
Y_3. A							
Y_4 Z							
Y_5 D							
Y_6 C							

L[i][j]		X_0	X_1	X_2	X_3	X_4	X_5	X_6
		Ø	B	A	C	B	A	D
Y_0	Ø	0	0	0	0	0	0	0
Y_1	A	0	0	1	1	1	1	1
Y_2	B	0	1	1	1	2	2	2
Y_3.	A	0	1	2	2	2	3	3
Y_4	Z	0	1	2	2	2	3	3
Y_5	D	0	1	2	2	2	3	4
Y_6	C	0	1	2	3	3	3	4

```
Algorithm LCS (X,Y):
Input: Strings X and Y with m and n elements, respectively
Output:  L[i][j] for i=0, 1, 2, …, m and j = 0, 1, 2, …, n
{
    Create a 2D matrix L[m+1][n+1];
    for i = 0 to m
        L[0][i] = 0;     // set the first row zero since the LCS of Xi and Y0 for any i is 0
    for j = 0 to n
        L[j][0] = 0;     // set the first column zero since the LCS of X0 and Yj for any j is 0

    for i = 1 to m
        for j = 1 to n
            if xi == yj
                L[i][j] = L[i-1][j-1] +1;
            else
                L[i][j] = max{L[i-1][j], L[i][j-1]};

    return L[m][n];
}
```

L[i][j]	X_0 Ø	X_1 M	X_2 Z	X_3 J	X_4 A	X_5 W	X_6 X
Y_0 Ø							
Y_1 X							
Y_2 M							
$Y_{3.}$ J							
Y_4 Y							
Y_5 A							
Y_6 U							