

Analysis and Design of Algorithms



CS3230
C23S30

Week 8
Dynamic Programming

Diptarka Chakraborty
Ken Sung

Recall: Divide and Conquer Paradigm

- **Divide** the input instance into 2 or more parts (of equal sizes).
- Solve the same problem for each smaller instance **recursively**.
- **Combine** their solutions to get the solution of the original instance.

It is usually quite straight forward to see if divide and conquer can be applied to solve a problem.

Fibonacci Number $F(n)$

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n > 1$

Problem: Given n compute $F(n)$

- Recursive algorithm
- Iterative algorithm

Analyze recursive algorithm for $F(n)$

```
RFIB(n)
{
    if n=0
        return 0;
    else if n=1
        return 1;
    else
        return((RFIB(n-1) + RFIB(n-2)));
}
```

Let $T(n)$ be the number of instructions executed by $RFIB(n)$

$$T(0) = 1$$

$$T(1) = 2$$

$$T(n) = 4 + T(n-1) + T(n-2) \text{ for } n > 1$$

Simple Exercise:
Show $F(n) > 2^{(n-2)/2}$
(may be use induction)

Observation: $T(n) > F(n)$

Analyze iterative (array-filling) algorithm for $F(n)$

```
IFIB(n) {  
    F[0] <- 0;  
    F[1] <- 1;  
    For(i=2 to n) do  
        F[i] <- F[i-1] + F[i-2];  
    return F[n];  
}
```

2 instructions

$n - 1$ iterations

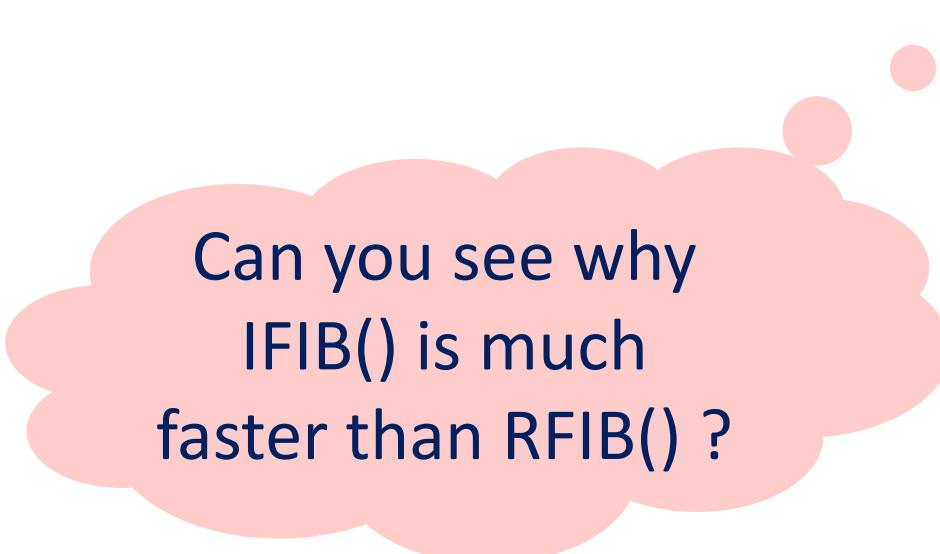
1 instruction

1 instruction

$$\text{No. of instructions} = 2 + (n - 1) \times 1 + 1 = n + 2$$

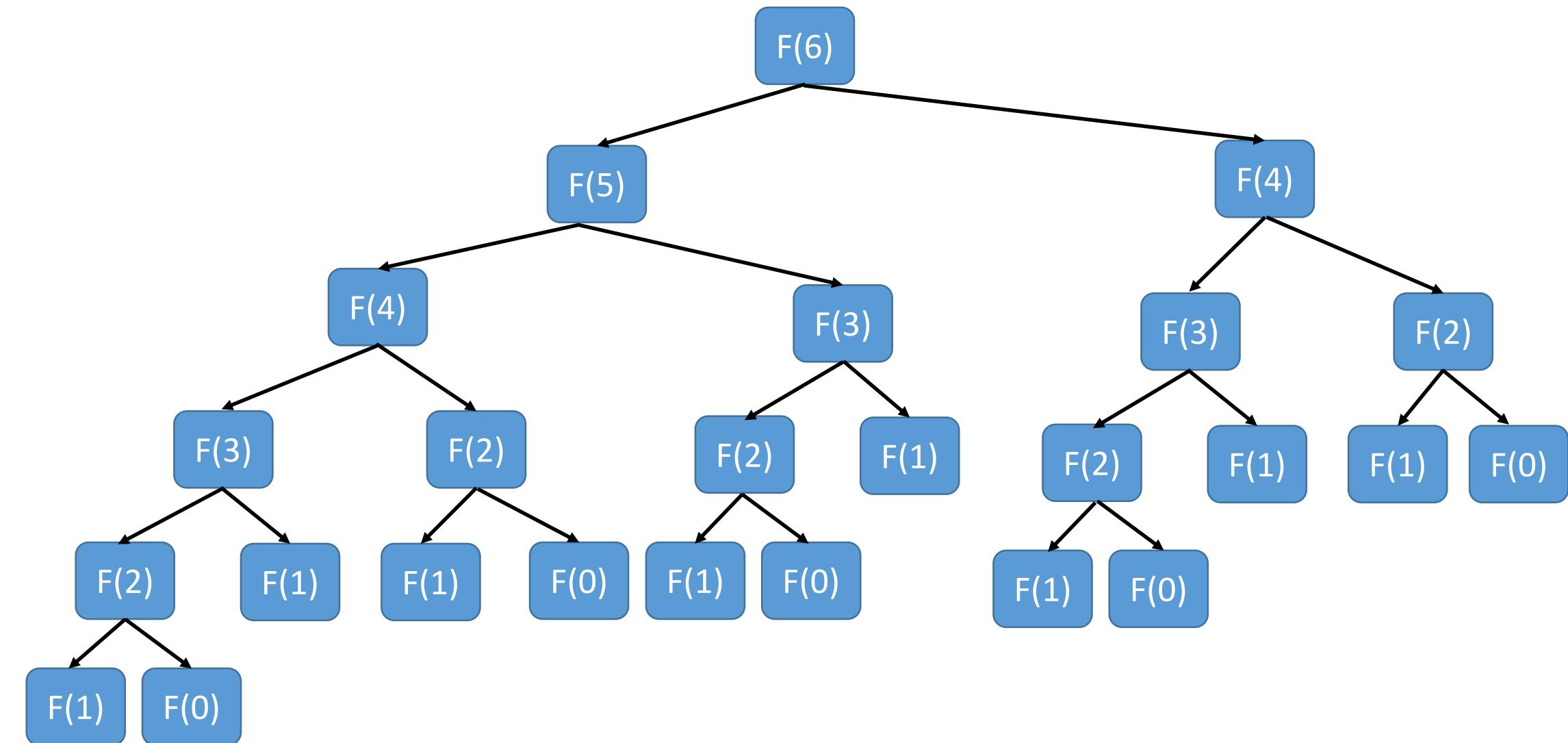
Compare two algorithms for $F(n)$

- No. of instructions by recursive algorithm $\text{RFIB}(n)$ is $> 2^{(n-2)/2}$ (exponential in n)
- No. of instructions by iterative algorithm $\text{IFIB}(n)$ is $= n + 2$ (linear in n)

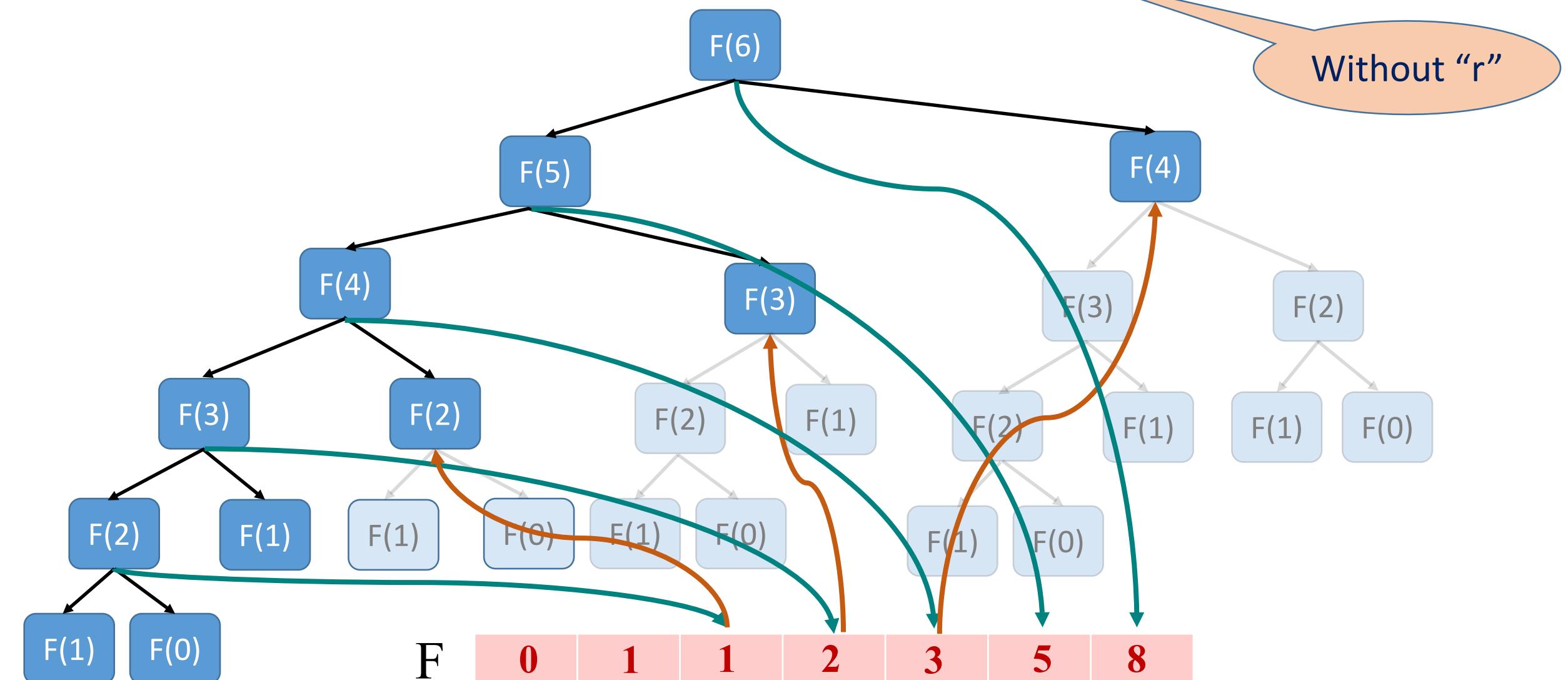


Can you see why
 $\text{IFIB}()$ is much
faster than $\text{RFIB}()$?

Recursion tree for $F(n)$



Pruning recursion tree by memoization for $F(n)$



Longest Common Subsequence

Applications in Computational Biology, Text Processing and many more

What is a subsequence?

Sequence $A : a_1, a_2, \dots, a_n$

Can be stored in an array $A[1..n]$, $A[j] : a_j$

$A[1..k] : a_1, a_2, \dots, a_k$

Definition: C is said to be a subsequence of A if we can obtain C by removing 0 or more elements from A .

Example: $A : a, b, a, d, e, b, c, a$

$C : a, a, e, b, a$

A more formal definition:

C is a subsequence of A if there exists k integers: $1 \leq i_1 < \dots < i_k \leq n$ s.t.
for all $1 \leq j \leq k$ $C[j] = A[i_j]$

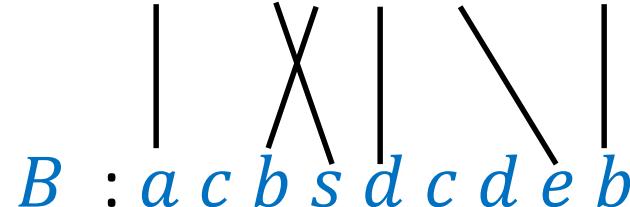
Longest Common Subsequence - Problem Definition

Given : Two sequences $A[1..n]$ and $B[1..m]$,

Aim : To compute a (not “the”) longest sequence C s.t.

C is subsequence of A as well as B

Example: $A : a \ a \ s \ b \ d \ e \ s \ c \ b \ d$



Answer: $a \ s \ d \ e \ b$

Question : How to compute a LCS of A and B efficiently.

Finding LCS: Trivial Brute-Force Solution

Given : two sequences $A[1..n]$ and $B[1..m]$

$$A : a_1, a_2, \dots, a_n$$

$$B : b_1, b_2, \dots, b_m$$

- Check all the possible subsequences of A to see if it is also a subsequence of B , and then output the longest one.

Analysis:

- Checking whether a particular subsequence of A is a subsequence of B takes $O(m)$ time.
- How many possible subsequences of A are there?
(Each bit-vector of length n determines a distinct subsequence)
- So total time = $O(m2^n)$

2^n

Can we do better?

Finding LCS: Recursive Formulation

Given : two sequences $A[1..n]$ and $B[1..m]$

$$A : a_1, a_2, \dots, a_n$$

$$B : b_1, b_2, \dots, b_m$$

Notation for recursive formulation:

$\text{LCS}(i, j)$: Longest common subsequence of $A[1..i]$ and $B[1..j]$

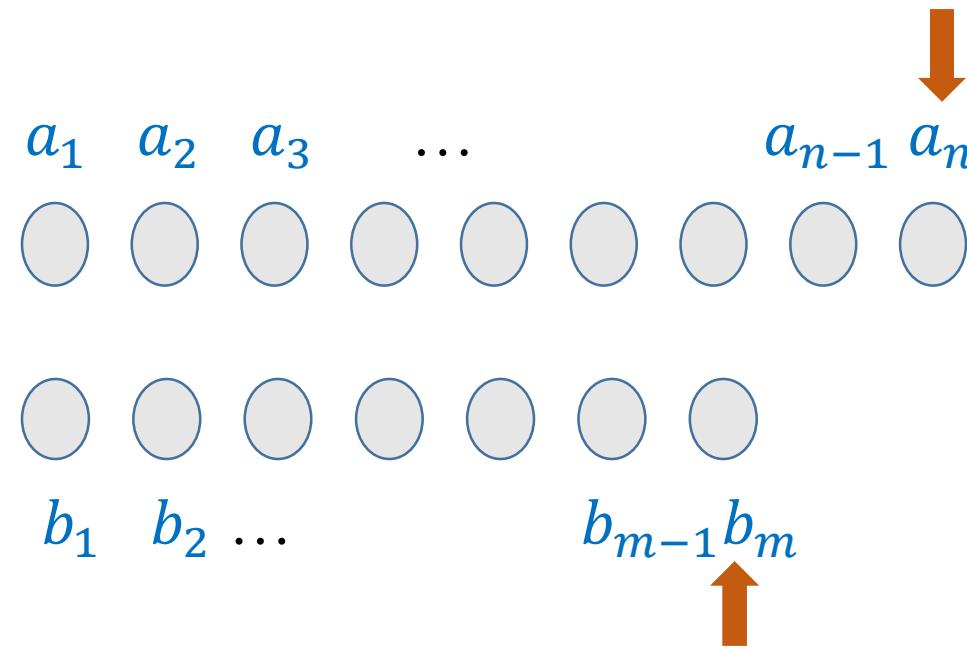
Aim : To express $\text{LCS}(i, j)$ recursively.

Base Case:

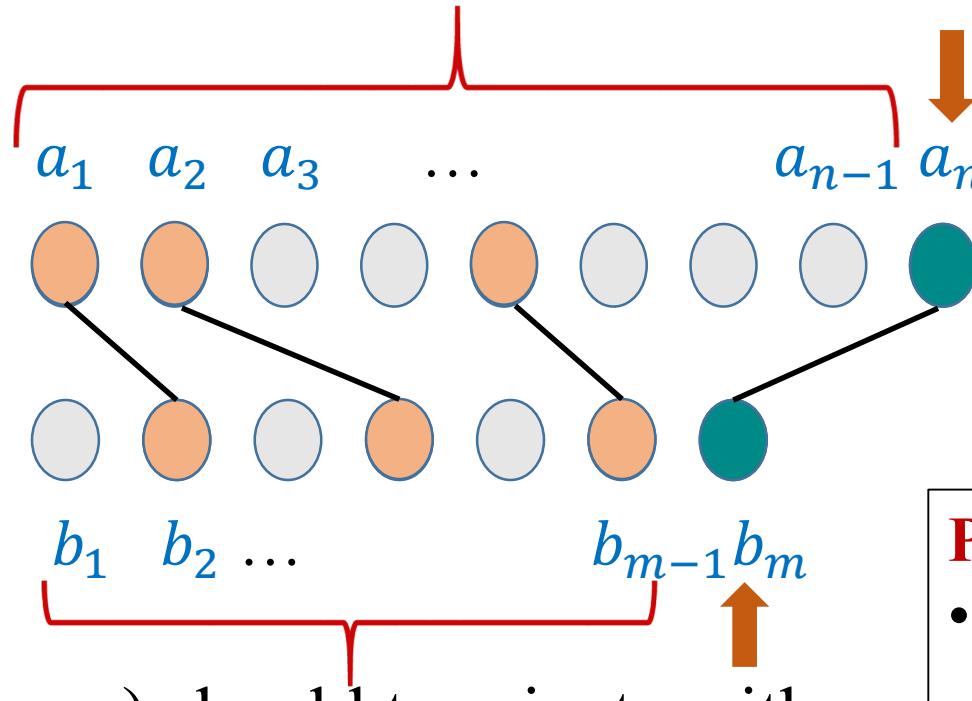
$$\left. \begin{array}{ll} \text{LCS}(i, 0) = \emptyset & \text{for all } i \\ \text{LCS}(0, j) = \emptyset & \text{for all } j \end{array} \right\}$$

Since one of the sequences is **empty**

Recursive Formulation of LCS(n, m)



How does $\text{LCS}(n,m)$ look like when $a_n = b_m$?



Intuition: $\text{LCS}(n,m)$ should terminate with a_n

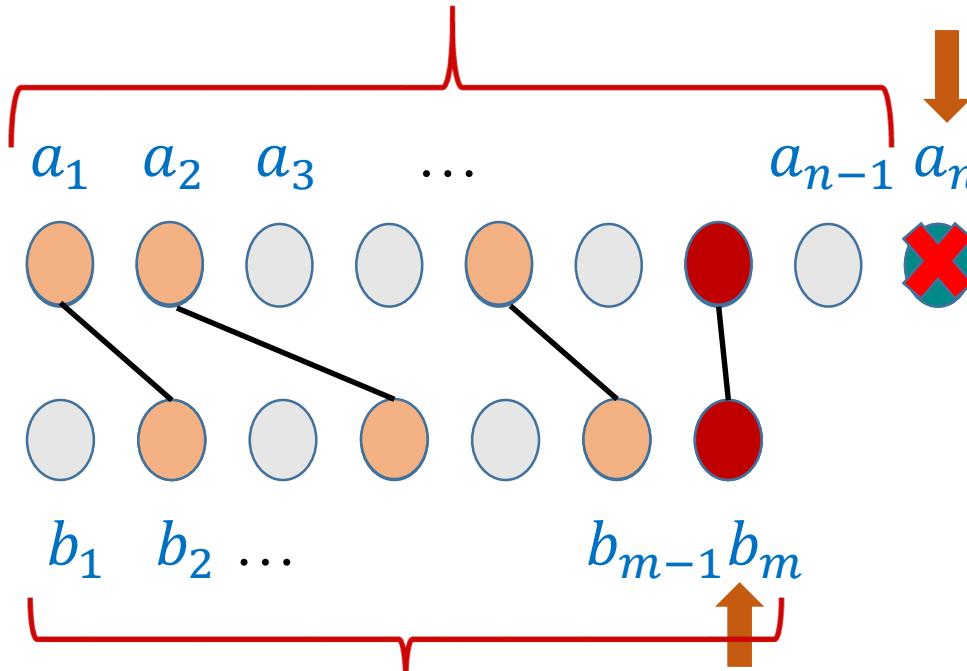
Lemma: If $a_n = b_m$ then

$$\text{LCS}(n,m) = \text{LCS}(n-1, m-1) :: a_n$$

Proof Idea:

- $\text{LCS}(n,m)$ must terminate with the symbol same as a_n ; otherwise we could extend the **solution** by concatenating a_n
- Observe, it is fine to match a_n with b_m

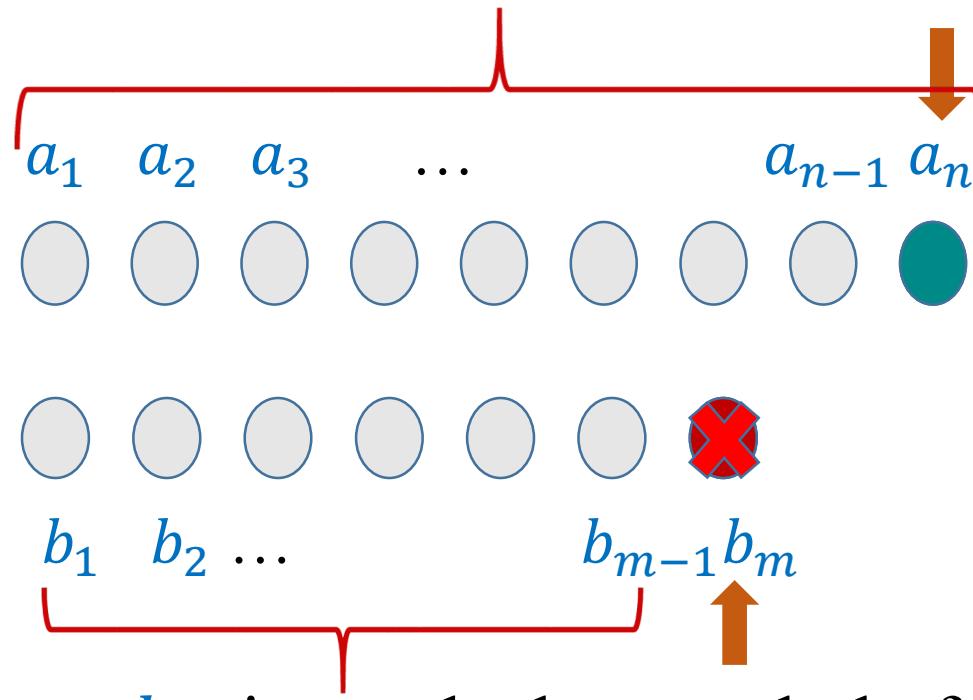
How does $\text{LCS}(n,m)$ look like when $a_n \neq b_m$?



Intuition: Either a_n or b_m is not the last symbol of $\text{LCS}(n,m)$

Observation: If a_n is not the last symbol of $\text{LCS}(n,m)$ $\text{LCS}(n,m) = \text{LCS}(n - 1,m)$

How does $\text{LCS}(n,m)$ look like when $a_n \neq b_m$?

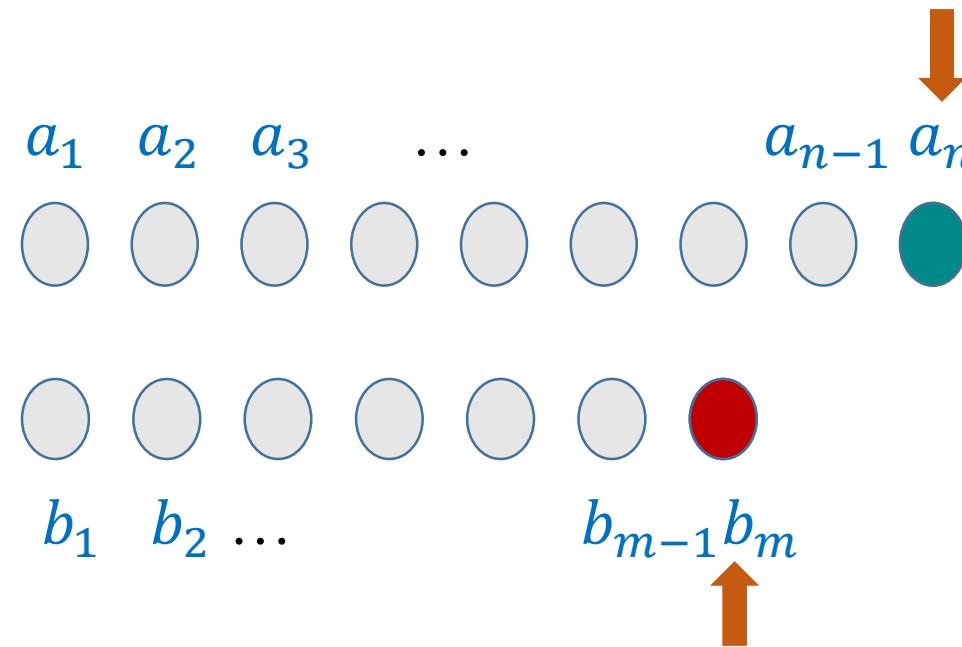


Intuition: Either a_n or b_m is not the last symbol of $\text{LCS}(n,m)$

Observation: If a_n is not the last symbol of $\text{LCS}(n,m)$ $\text{LCS}(n,m) = \text{LCS}(n-1,m)$

Observation: If b_m is not the last symbol of $\text{LCS}(n,m)$ $\text{LCS}(n,m) = \text{LCS}(n,m-1)$

How does $\text{LCS}(n,m)$ look like when $a_n \neq b_m$?



Intuition: Either a_n or b_m is not the last symbol of $\text{LCS}(n,m)$

Lemma: If $a_n \neq b_m$ then

$\text{LCS}(n,m)$ is either $\text{LCS}(n-1, m)$ or $\text{LCS}(n, m-1)$

Finding LCS: Recursive Formulation

Base Case:

$$\text{LCS}(i, 0) = \emptyset \quad \text{for all } i$$

$$\text{LCS}(0, j) = \emptyset \quad \text{for all } j$$

General Case:

$$\text{If } a_n = b_m \text{ then } \text{LCS}(n, m) = \text{LCS}(n - 1, m - 1) :: a_n$$

$$\text{If } a_n \neq b_m \text{ then } \text{LCS}(n, m) = \underline{\text{bigger}} \text{ of } \text{LCS}(n, m - 1) \text{ or } \text{LCS}(n - 1, m)$$

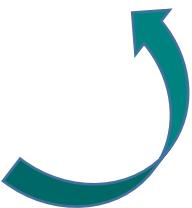
Simplified Problem: Find the length of LCS

Let $L(n, m)$: Length of LCS of $A[1..n]$ and $B[1..m]$

$$L(n, m) = 0 \quad \text{if } n \text{ or } m \text{ is 0}$$

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems



Question 1

Can you represent $L(n,m)$ in terms of subproblems?



Finding LCS: Recursive Formulation

Base Case:

$$\text{LCS}(i, 0) = \emptyset \quad \text{for all } i$$

$$\text{LCS}(0, j) = \emptyset \quad \text{for all } j$$

General Case:

$$\text{If } a_n = b_m \text{ then } \text{LCS}(n, m) = \text{LCS}(n - 1, m - 1) :: a_n$$

$$\text{If } a_n \neq b_m \text{ then } \text{LCS}(n, m) = \underline{\text{bigger}} \text{ of } \text{LCS}(n, m - 1) \text{ or } \text{LCS}(n - 1, m)$$

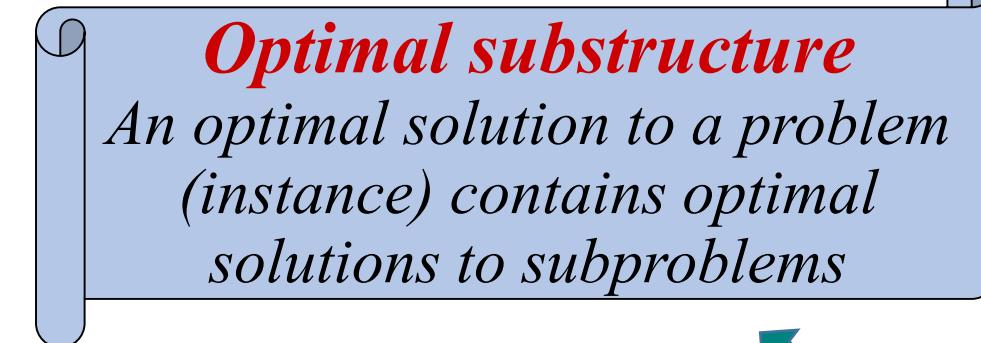
Simplified Problem: Find the length of LCS

Let $L(n, m)$: Length of LCS of $A[1..n]$ and $B[1..m]$

$$L(n, m) = 0 \quad \text{if } n \text{ or } m \text{ is 0}$$

$$\text{If } a_n = b_m \text{ then } L(n, m) = L(n - 1, m - 1) + 1$$

$$\text{If } a_n \neq b_m \text{ then } L(n, m) = \text{Max}(L(n, m - 1), L(n - 1, m))$$



Recursive algorithm for $L(n,m)$

```
 $L(n,m)$ 
{   If ( $n = 0$  or  $m = 0$ )
    return 0;
Else
{   If  $a_n = b_m$  then
    return ( $L(n - 1,m - 1) + 1$ );
Else
{    $l_1 \leftarrow L(n - 1,m)$  ;
     $l_2 \leftarrow L(n,m - 1)$  ;
    return Max( $l_1, l_2$ );
}
}
}
```

$T(n,m)$: Worst case running time of $L(n,m)$

$$T(n,m) = T(n - 1,m) + T(n,m - 1)$$

A simple exercise from discrete math:

$$T(n,m) \geq \binom{n+m}{n} > 2^n$$

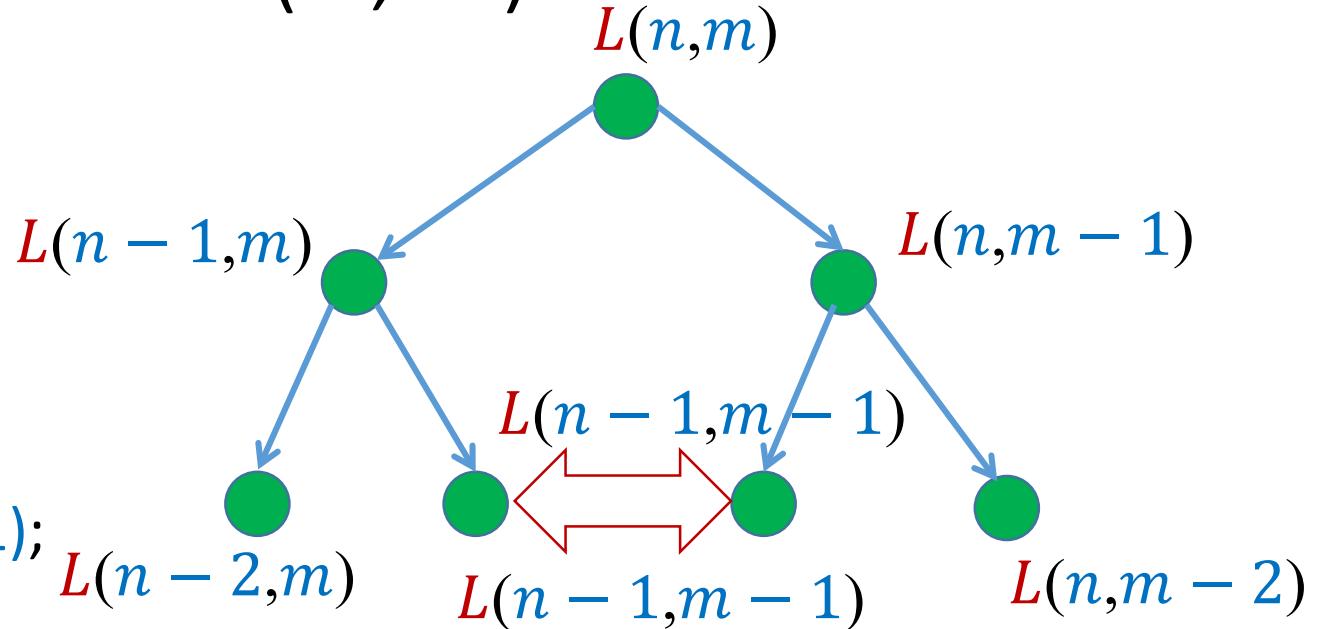
Exponential !!



But why ?
Let us explore

Recursive algorithm for $L(n,m)$

```
 $L(n,m)$ 
{   If ( $n = 0$  or  $m = 0$ )
    return 0;
Else
{   If  $a_n = b_m$  then
    return ( $L(n - 1,m - 1) + 1$ );
Else
{    $l_1 \leftarrow L(n - 1,m)$  ;
     $l_2 \leftarrow L(n,m - 1)$  ;
    return Max( $l_1, l_2$ );
}
}
```



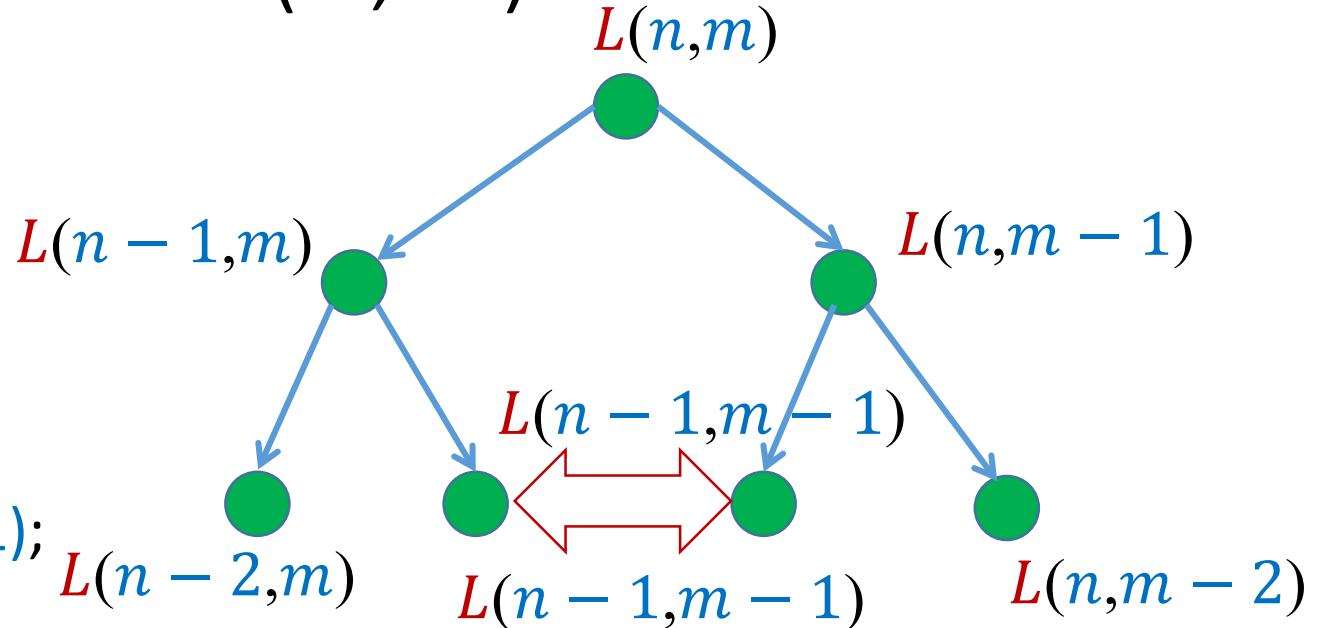
- Solving same sub-problem multiple times !!
- But how many **distinct** sub-problems are there ?
- Only $(n + 1) \times (m + 1)$

Overlapping subproblems

A recursive solution contains a “small” number of distinct subproblems repeated many times

Recursive algorithm for $L(n,m)$

```
 $L(n,m)$ 
{   If ( $n = 0$  or  $m = 0$ )
    return 0;
Else
{   If  $a_n = b_m$  then
    return ( $L(n - 1,m - 1) + 1$ );
Else
{    $l_1 \leftarrow L(n - 1,m)$  ;
     $l_2 \leftarrow L(n,m - 1)$  ;
    return Max( $l_1, l_2$ );
}
}
}
```



- Solving same sub-problem multiple times !!
- But how many sub-problems are there ?
- Only $(n + 1) \times (m + 1)$
- Can we compute them efficiently ?
- Get inspiration from algo for Fibonacci number !

Recursive algorithm for $L(n,m)$

```

 $L(n,m)$ 
{   If ( $n = 0$  or  $m = 0$ )
    return 0;
Else
{   If  $a_n = b_m$  then
    return ( $L(n - 1, m - 1) + 1$ );
Else
{        $l_1 \leftarrow L(n - 1, m)$  ;
         $l_2 \leftarrow L(n, m - 1)$  ;
        return Max( $l_1, l_2$ );
}
}

```

$$\textcolor{red}{T}[i,j] = L(i,j)$$

m	0	0	0	0	0	0
⋮	0	0	0	0	0	0
1	0	0	0	0	0	0
0	0	0	0	0	0	0

$0 \quad 1 \quad \dots \quad n$

$\textcolor{red}{T}$

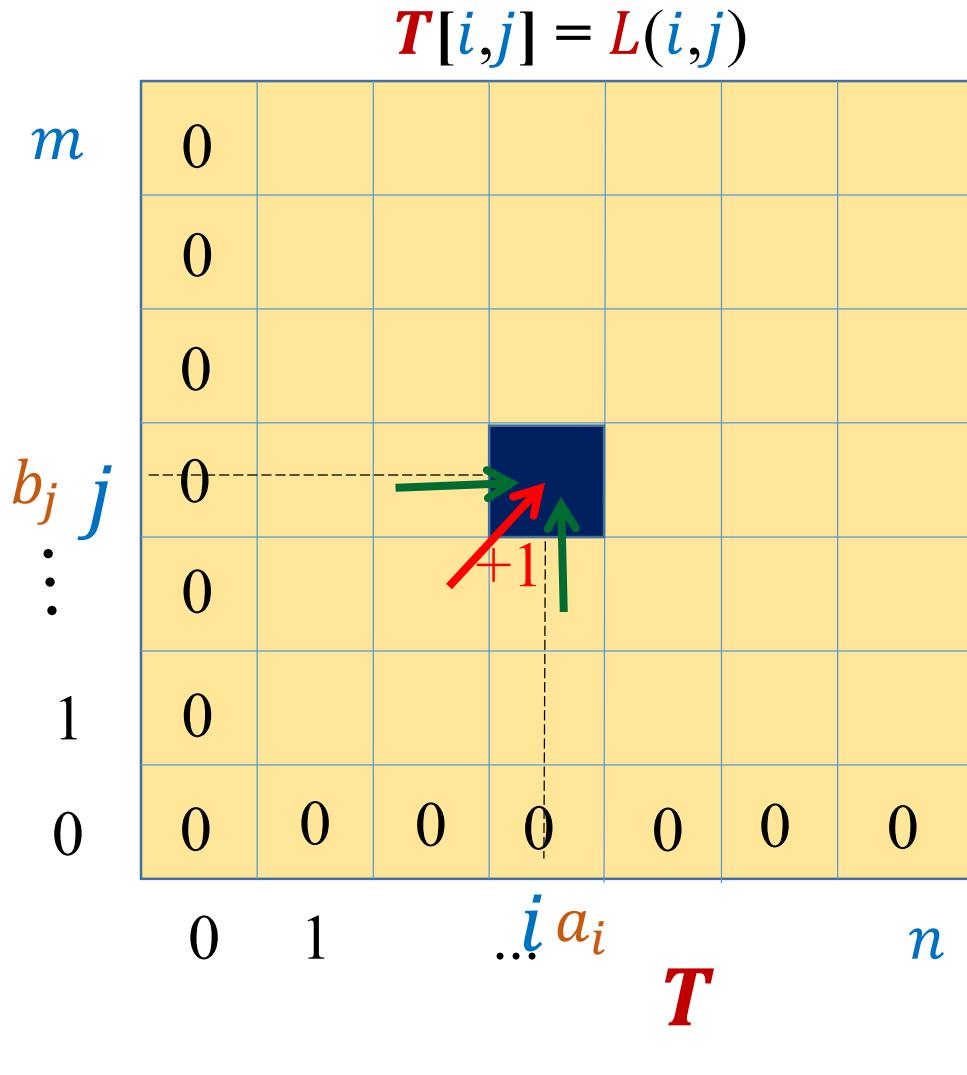
Dynamic Programming algorithm for $L(n,m)$

$L(n,m)$

```

{   For ( $i = 0$  to  $n$ )    $T[i,0] \leftarrow 0$ ;
    For ( $j = 0$  to  $m$ )    $T[0,j] \leftarrow 0$ ;
    For ( $j = 1$  to  $m$ ){
        For ( $i = 1$  to  $n$ ){
            If  $a_i = b_j$  then
                 $T[i,j] \leftarrow T[i - 1,j - 1] + 1$ ;
            Else {
                 $l_1 \leftarrow T[i - 1,j]$ ;
                 $l_2 \leftarrow T[i,j - 1]$ ;
                 $T[i,j] \leftarrow \text{Max}(l_1, l_2)$ ;
            }
        }
    }
}

```



Dynamic Programming algorithm for $L(n,m)$

$$L(n,m)$$

```

{   For (i = 0 to n)  T[i,0]  $\leftarrow$  0;
    For (j = 0 to m)  T[0,j]  $\leftarrow$  0;
    For (j = 1 to m){
        For (i = 1 to n){
            If ai = bj then
                T[i,j]  $\leftarrow$  T[i - 1,j -
            Else {
                l1  $\leftarrow$  T[i - 1,j] ;
                l2  $\leftarrow$  T[i,j - 1] ;
                T[i,j]  $\leftarrow$  Max(l1, l2);
            }
        }
    }
}

```

$$\textcolor{red}{T}[i,j] = \textcolor{red}{L}(\textcolor{blue}{i},j)$$

A diagram showing a 7x7 grid representing a sparse vector. The grid has 0s in most cells and 1s in four specific cells: (3,3), (3,4), (4,3), and (4,4). A red arrow labeled '+1' points from the origin to the cell at (3,3). A green arrow points right along the row 3, and another green arrow points up along column 3.

Time per table entry = $O(1)$
Total time = $O(nm)$

Dynamic Programming algorithm for $L(n,m)$

$L(n,m)$

```

{  For ( $i = 0$  to  $n$ )   $T[i,0] \leftarrow 0$ ;
  For ( $j = 0$  to  $m$ )   $T[0,j] \leftarrow 0$ ;
  For ( $j = 1$  to  $m$ ){
    For ( $i = 1$  to  $n$ ){
      If  $a_i = b_j$  then
         $T[i,j] \leftarrow T[i - 1,j - 1] + 1$ ;
      Else {
         $l_1 \leftarrow T[i - 1,j]$  ;
         $l_2 \leftarrow T[i,j - 1]$  ;
         $T[i,j] \leftarrow \text{Max}(l_1, l_2)$ ;
      }
    }
  }
}
  
```

							$T[i,j] = L(i,j)$
a	0	1	2	2	3	3	4
d	0	1	2	2	3	3	3
c	0	1	1	2	2	2	3
a	0	1	1	1	2	2	3
d	0	0	1	1	2	2	2
d	0	0	1	1	1	1	1
a	0	0	0	0	0	0	0

T

Dynamic Programming algorithm for $L(n,m)$

```
 $L(n,m)$ 
{  For ( $i = 0$  to  $n$ )   $T[i,0] \leftarrow 0$ ;
   For ( $j = 0$  to  $m$ )   $T[0,j] \leftarrow 0$ ;
   For ( $j = 1$  to  $m$ ){
      For ( $i = 1$  to  $n$ ){
         If  $a_i = b_j$  then
             $T[i,j] \leftarrow T[i - 1,j - 1] + 1$ ;
         Else {
             $l_1 \leftarrow T[i - 1,j]$  ;
             $l_2 \leftarrow T[i,j - 1]$  ;
             $T[i,j] \leftarrow \text{Max}(l_1, l_2)$ ;
         }
      }
   }
}
```

Note, you need to store the table T .
So space requirement is $O(mn)$.

Exercise:

- How can you reduce the space requirement to $O(\min\{m, n\})$?
- Can you modify the algorithm so that you can output a LCS ?

Dynamic Programming algorithm paradigm

- Expressing the solution recursively
- Overall there are only polynomial number of subproblems
- But there is a huge overlap among the subproblems. So the recursive algorithm takes exponential time (solving same subproblem multiple times)
- So we compute the recursive solution iteratively in a bottom-up fashion (like in case of Fibonacci numbers). This avoids wastage of computation and leads to an efficient implementation

Question 2

Who is the Master of Algorithms pictured below?



- John Hopcroft
- Andrew Chi-Chih Yao
- Richard Bellman
- Ron Rivest

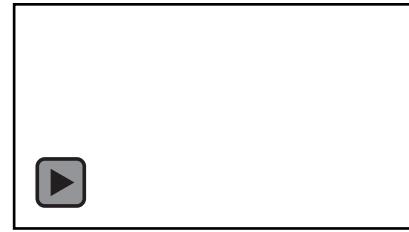


Question 2: Solution

Richard E. Bellman

Bellman started the systematic study of dynamic programming, providing the area with a solid mathematical basis. His contributions are foundational to control theory and economics.

Question 3



A palindrome is a string that reads the same forwards and backwards. Examples: *racecar*, *civic*, *a*.

Use the LCS algorithm to give an $O(n^2)$ -time algorithm to find the longest palindrome subsequence of an input string.

Example: if input is *character*, output should be *carac*.

Question 3: Solution

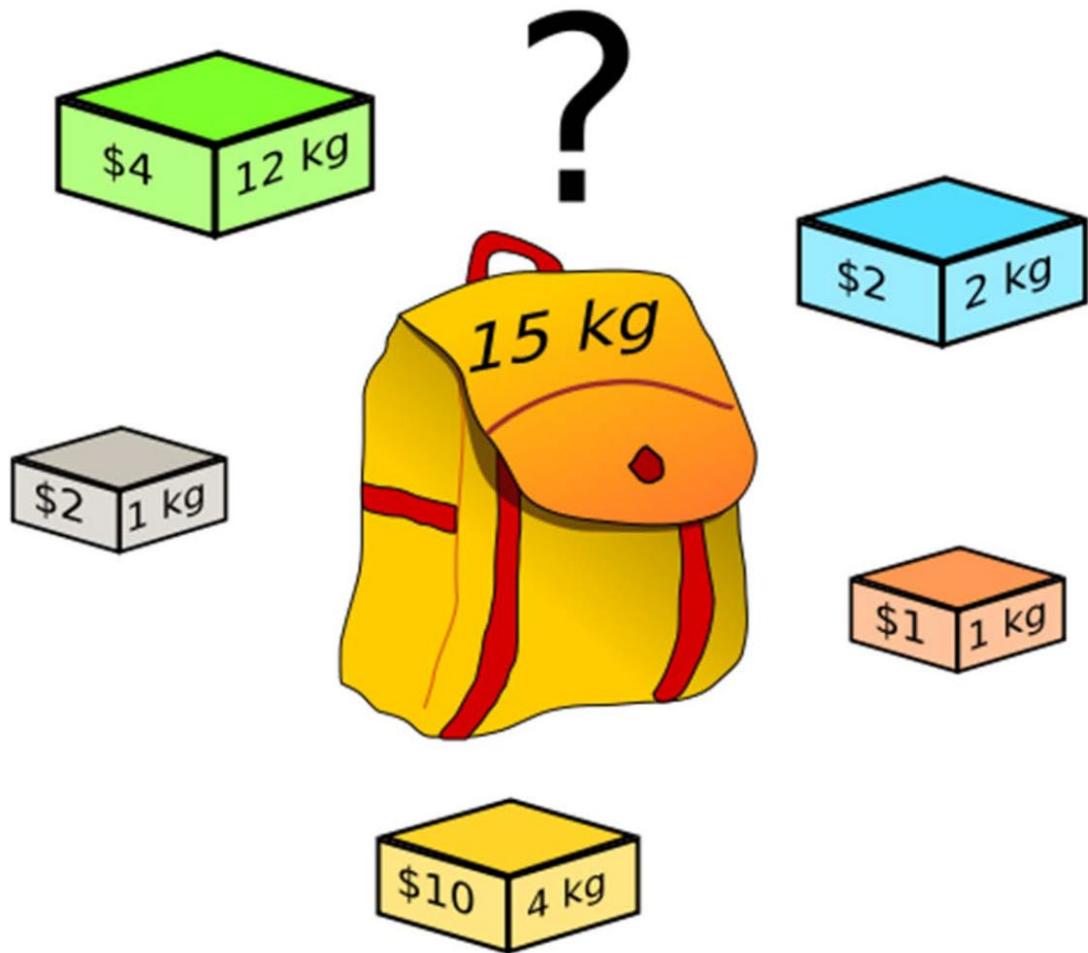
Run the LCS algorithm on the input string and its reverse!

More direct DP approach can also be formulated.

Exercise: Solve the problem this way.

Knapsack Problem

Knapsack Problem



*What is the
maximum
value you
can get?*

Formal Definition

KNAPSACK

Input:

$(w_1, v_1), (w_2, v_2), \dots, (w_n, v_n)$, and W

Output: A subset $S \subseteq \{1, 2, \dots, n\}$ that maximizes

$\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$

2^n many subsets,
so naïve
algorithm is too
costly!

Dynamic Programming

Problem: $(w_1, v_1), \dots, (w_n, v_n), W$

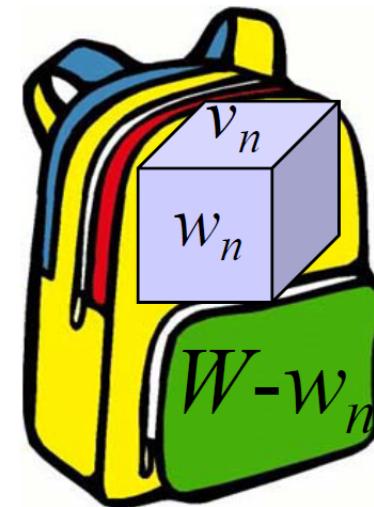
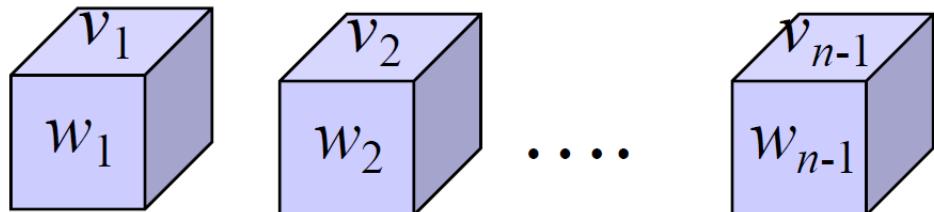


Is there optimal substructure?

Dynamic Programming

Case 1: Item n (the last one) is taken

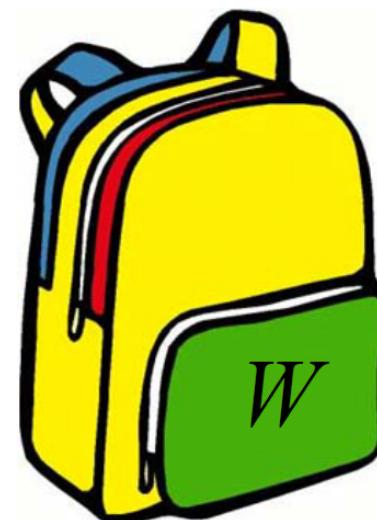
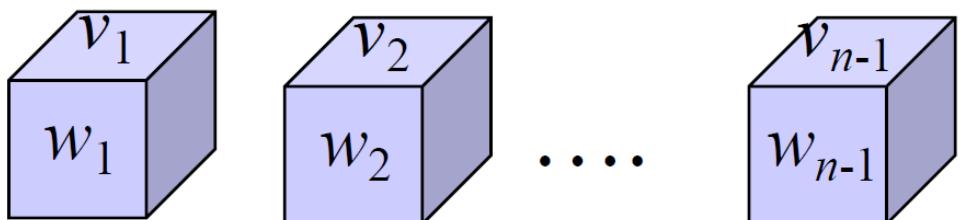
Have optimal solution to subproblem
defined by $(w_1, v_1), \dots, (w_{n-1}, v_{n-1}), W-w_n$



Dynamic Programming

Case 2: Item n (the last one) is **not** taken

Have optimal solution to subproblem
defined by $(w_1, v_1), \dots, (w_{n-1}, v_{n-1}), W$



Otherwise, by *cut and paste*
argument, we can get a better solution

Recursive Solution

Let $m[i, j]$ be the maximum value that can be obtained using:

- a subset of items in $\{1, 2, \dots, i\}$
- with total weight no more than j

$$m[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i - 1, j - w_i] + v_i, m[i - 1, j]\}, & \text{if } w_i \leq j \\ m[i - 1, j], & \text{otherwise} \end{cases}$$

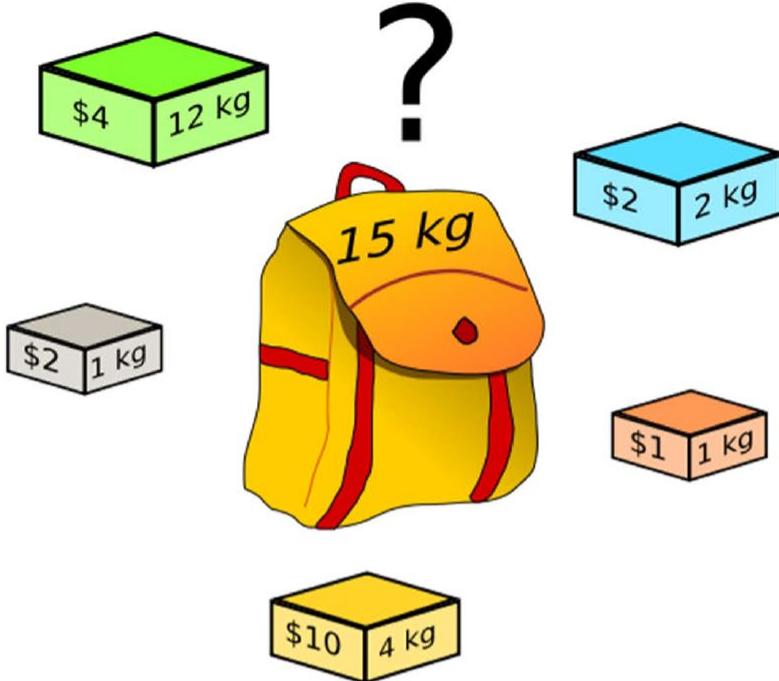
Pseudocode

```
KNAPSACK( $v, w, W$ ):  
    for  $j = 0, \dots, W$ :  
         $m[0, j] \leftarrow 0$   
    for  $i = 1, \dots, n$ :  
         $m[i, 0] \leftarrow 0$   
    (Recursive cases)  
  
return  $m[n, W]$ 
```

Pseudocode

⟨Recursive cases⟩

```
for  $i = 1, \dots, n$ :
    for  $j = 0, \dots, W$ :
        if  $j \geq w[i]$ :
             $m[i, j] \leftarrow \max(m[i - 1, j - w[i]] + v[i], m[i - 1, j])$ 
        else:
             $m[i, j] \leftarrow m[i - 1, j]$ 
```



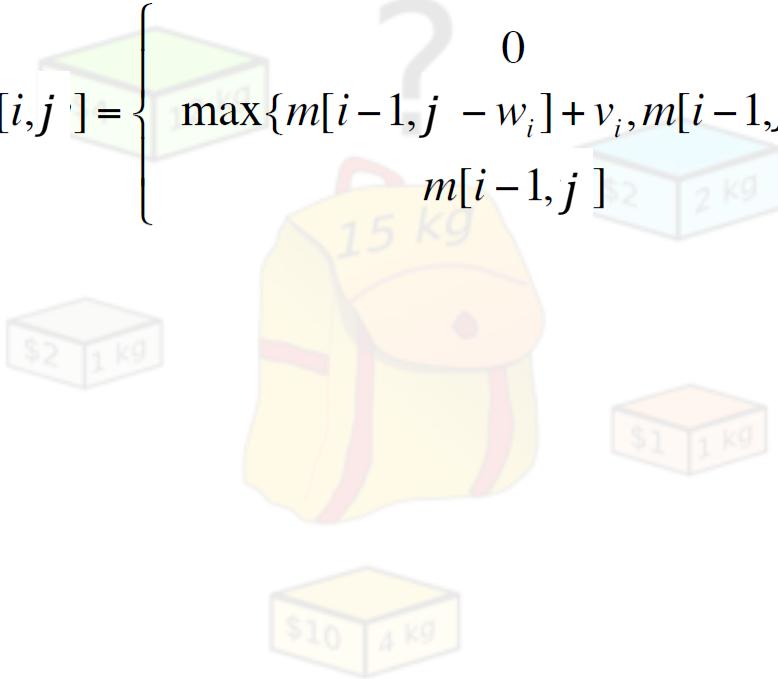
$$v[1..5] = \{4, 2, 10, 1, 2\}$$

$$w[1..5] = \{12, 1, 4, 1, 2\}$$

$$W = 15$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0															
2	0															
3	0															
4	0															
5	0															

$$m[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j - w_i] + v_i, m[i-1, j]\} & \text{if } w_i \leq j \\ m[i-1, j] & \text{otherwise} \end{cases}$$



$$v[1..5] = \{4, 2, 10, 1, 2\}$$

$$w[1..5] = \{12, 1, 4, 1, 2\}$$

$$W = 15$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	4		
2	0															
3	0															
4	0															
5	0															

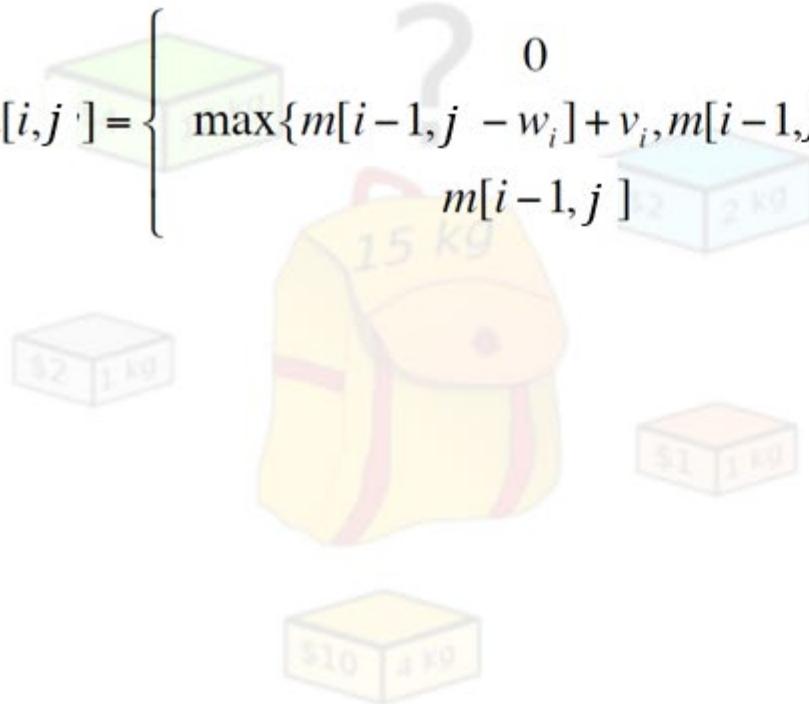
$$m[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j - w_i] + v_i, m[i-1, j]\} & \text{if } w_i \leq j \\ m[i-1, j] & \text{otherwise} \end{cases}$$

$$v[1..5] = \{4, 2, 10, 1, 2\}$$

$$w[1..5] = \{12, 1, 4, 1, 2\}$$

$$W = 15$$

$$m[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j - w_i] + v_i, m[i-1, j]\} & \text{if } w_i \leq j \\ m[i-1, j] & \text{otherwise} \end{cases}$$



$$v[1..5] = \{4, 2, 10, 1, 2\}$$

$$w[1..5] = \{12, 1, 4, 1, 2\}$$

$$W = 15$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	6
3	0	2	2	2	10	12	12	12	12	12	12	12	12	12	12	12
4	0	2	3	3	10	12										
5	0															

12

$$m[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{m[i-1, j - w_i] + v_i, m[i-1, j]\} & \text{if } w_i \leq j \\ m[i-1, j] & \text{otherwise} \end{cases}$$



$$v[1..5] = \{4, 2, 10, 1, 2\}$$

$$w[1..5] = \{12, 1, 4, 1, 2\}$$

$$W = 15$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	
2	0	2	2	2	2	2	2	2	2	2	2	2	4	6	6	
3	0	2	2	2	10	12	12	12	12	12	12	12	12	12	12	
4	0	2	3	3	10	12	13	13	13	13	13	13	13	13	13	
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	

Pseudocode

⟨Recursive cases⟩

for $i = 1, \dots, n$:

for $j = 0, \dots, W$:

if $j \geq w[i]$:

$$m[i, j] \leftarrow \max(m[i - 1, j - w[i]] + v[i], m[i - 1, j])$$

else:

$$m[i, j] \leftarrow m[i - 1, j]$$

Time per table entry = $\mathcal{O}(1)$
Total time = $\mathcal{O}(nW)$

Question 4



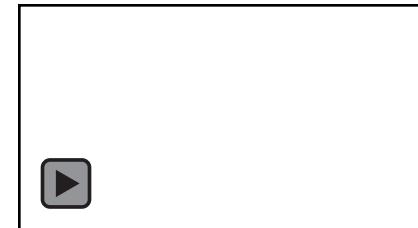
We have n cents and need to get change in terms of denominations d_1, d_2, \dots, d_k . Goal is to use the fewest total number of coins.

Example: If denominations are 25c, 10c, and 1c, then solution for $n = 30$ c should be 10c+10c+10c.

Let $M[j]$ be the fewest number of coins needed to change j cents. Write a recursive formula for $M[j]$ in terms of $M[i]$ with $i < j$.

Question 5

Using the above, derive a DP algorithm to compute the minimum number of coins of denomination d_1, \dots, d_k needed to change n cents.



Dynamic Programming algorithm paradigm

- Expressing the solution recursively
- Overall there are only polynomial number of subproblems
- But there is a huge overlap among the subproblems. So the recursive algorithm takes exponential time (solving same subproblem multiple times)
- So we compute the recursive solution iteratively in a bottom-up fashion (like in case of Fibonacci numbers). This avoids wastage of computation and leads to an efficient implementation

Acknowledgement

- The slides are modified from
 - The slides from Prof. Surender Baswana
 - The slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
 - The slides from Prof. Arnab Bhattacharya and Prof. Wing-Kin Sung