# *Design and Analysis of Algorithms*

CS3230

Lecture 8

## Dynamic Programming

**Warut Suksompong**

# Fibonacci Number F(n)

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

**Problem:** Given $n, m$, compute $F(n) \; mod \; m$

- Recursive algorithm
- Iterative algorithm

# Two algorithms for Fibonacci (mod m)

## Recursive Algorithm

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

For memoization build a table T[0 1 ... n] storing the values of $F_i$ for $0 \leq i \leq n$.

Before calling RFIB each time, check first whether the value is already stored in T.

If it is already stored, use it.

Else, call RFIB, and once the answer arrives, store it in T.

## Iterative Algorithm

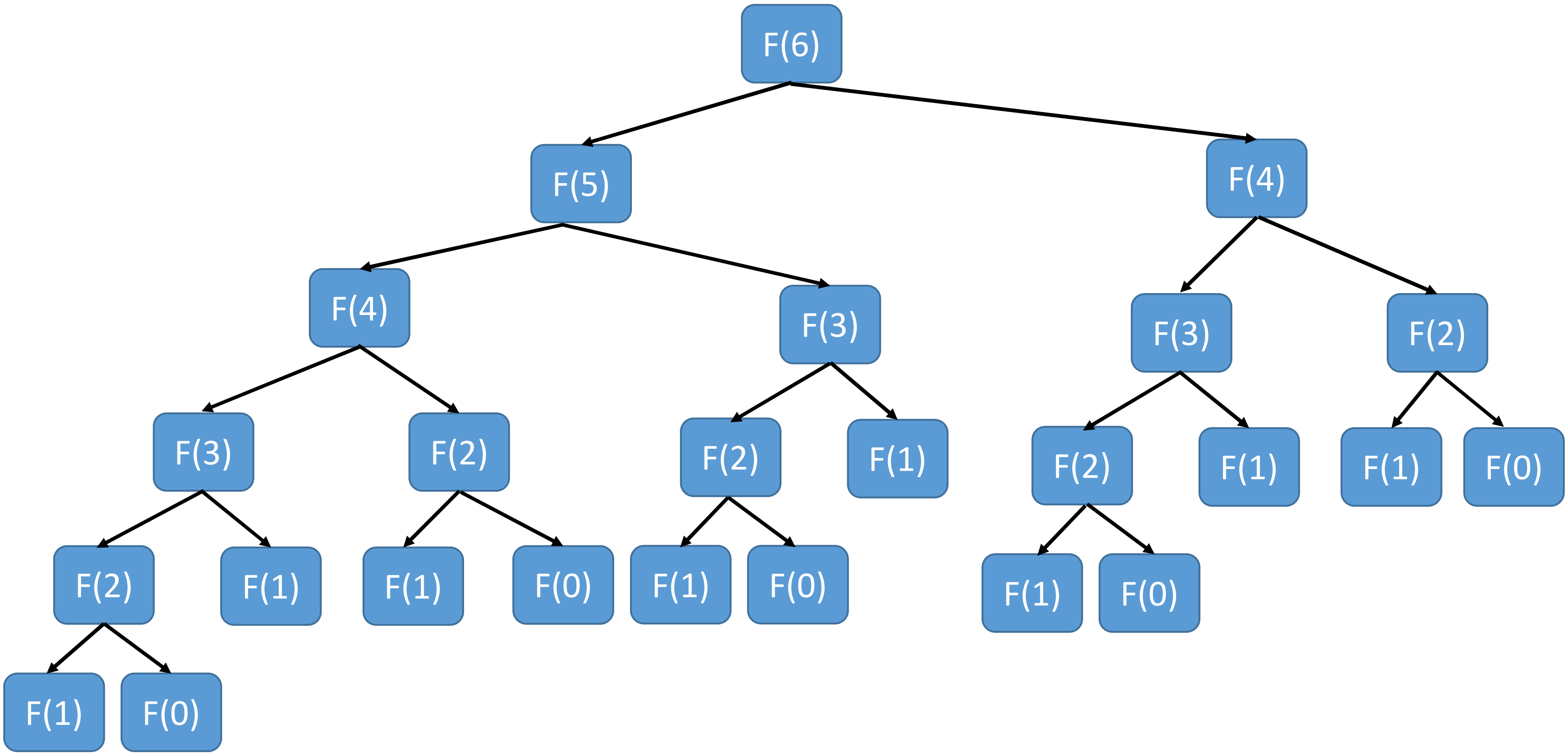
```
IFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
       else {
                a ← 0;   b← 1;
                For(i=2 to n) do
                {
                temp ← b;
                b← (a+b) mod m;
                a← temp; }
        }
        return b;}
```

# Compare two algorithms for F(n) mod m

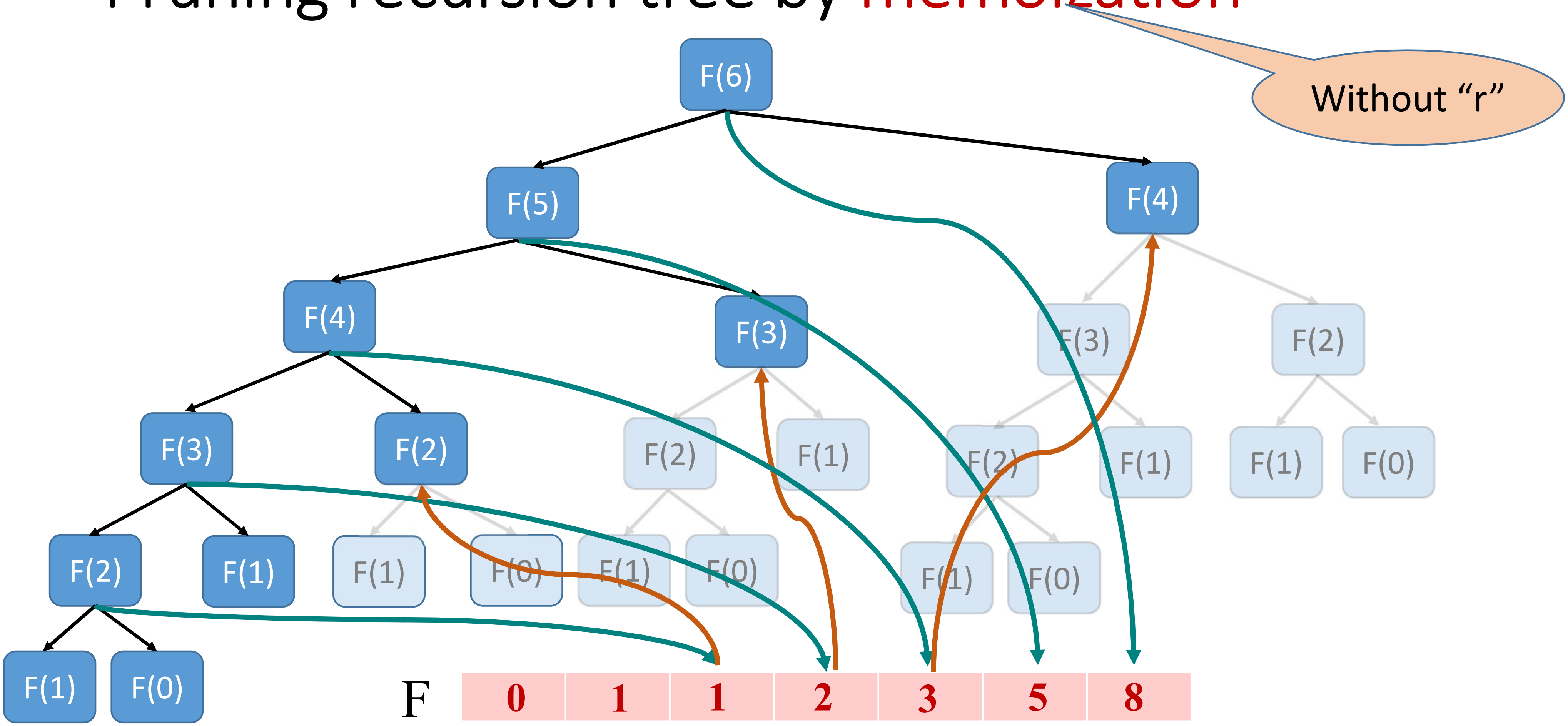
- No. of instructions by recursive algorithm **RFIB**(n,m) is $\geq 2^{(n-2)/2}$ (**exponential in $\boldsymbol{n}$**)
- No. of instructions by iterative algorithm **IFIB**(n,m) is $\approx 5n$ (**linear in $\boldsymbol{n}$**)

Can you see why IFIB() is much faster than RFIB() ?

# Recursion tree for F(n)



# Pruning recursion tree by memoization



# Longest Common Subsequence

Applications in Computational Biology, Text Processing and many more

# What is a subsequence?

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

Can be stored in an array $A[1..n]$, $A[j]$ : $a_j$ $\qquad$ $A[1..k]$ : $a_1, a_2, \ldots, a_k$

**Definition**: $C$ is said to be a **subsequence** of $A$ if we can obtain $C$ by removing zero 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: $1 \leq i_1 < \cdots < i_k \leq n$ s.t.

for all $1 \leq j \leq k$ $\qquad$ $C[j] = A[i_j]$

# Longest Common Subsequence - Definition

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

**Aim** : To compute a (not "the") <u>longest</u> sequence $C$ such that

$C$ is subsequence of $A$ as well as $B$

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

$B$ : $a\ c\ b\ s\ d\ c\ d\ e\ b$

**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 $B[1..m]$

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

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

- Check all the possible subsequences of A to see if it is also a subsequence of B, and then output a 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? $2^n$

  (Each bit-vector of length $n$ determines a distinct subsequence)
- So total time = $O(m2^n)$

Can we do better?

# Finding LCS: Recursive Formulation

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

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

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

**Notation for recursive formulation:**

**LCS**($i$, $j$): Longest common subsequence of $A[1..i]$ and $B[1..j]$

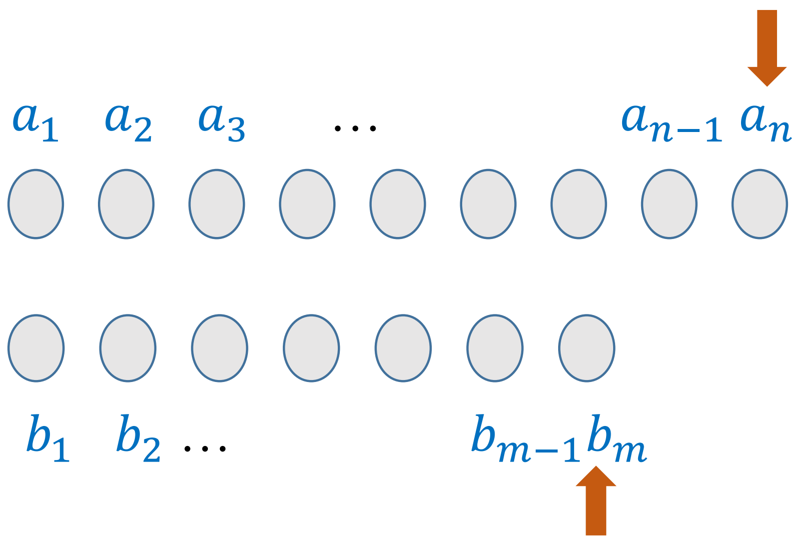
**Aim** : To express **LCS**($i$, $j$) recursively.

**Base Case**:

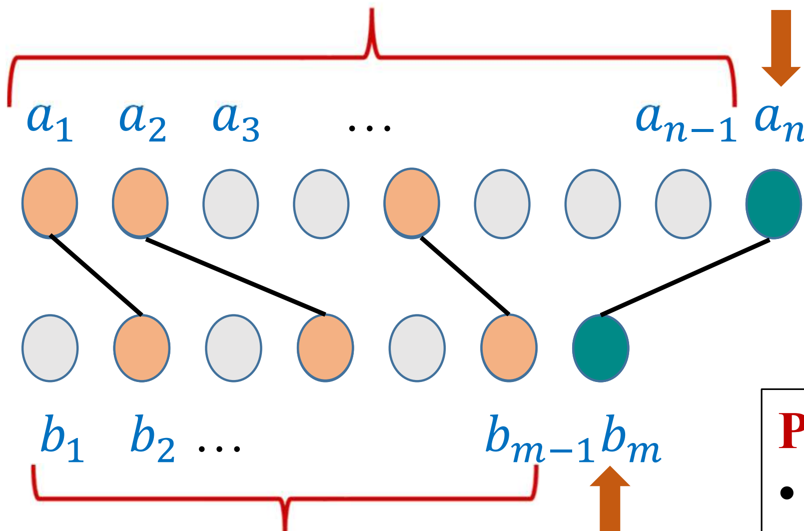
$$\left.\begin{array}{ll} \textbf{LCS}(i, 0) = \emptyset & \text{for all } i \\ \textbf{LCS}(0, j) = \emptyset & \text{for all } j \end{array}\right\}$$

Since one of the sequences is **empty**

# Recursive Formulation of LCS(n,m)



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



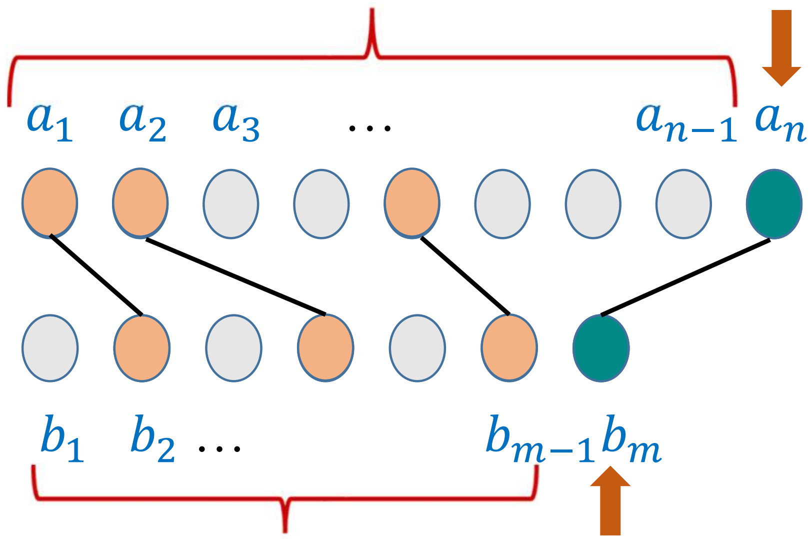

**Intuition**: **LCS**($n$,$m$) should terminate with $a_n$

**Lemma:** If $a_n = b_m$ then
**LCS**($n$,$m$) = **LCS**($n-1$,$m-1$) **::** $a_n$

**Proof Idea:**

- 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 LCS($\boldsymbol{n}$,$\boldsymbol{m}$) look like when $\boldsymbol{a_n = b_m}$?

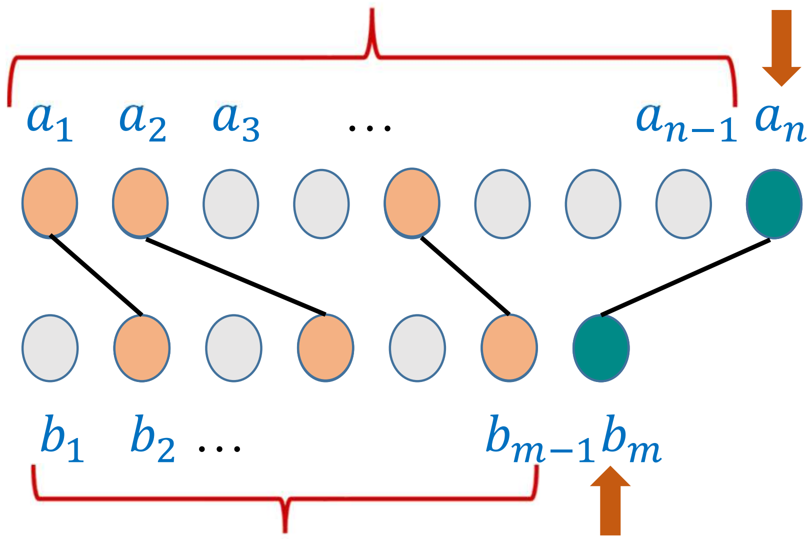



**Proof:** (Proof by contradiction)

- If the last symbol in $S$ =LCS($n$,$m$) is not the same as $a_n (= b_m)$, then that last symbol must be part of $a_1, \cdots, a_{n-1}$ and $b_1, \cdots, b_{m-1}$.

- So, $S$ is actually a subsequence of $a_1, \cdots, a_{n-1}$ and $b_1, \cdots, b_{m-1}$.

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

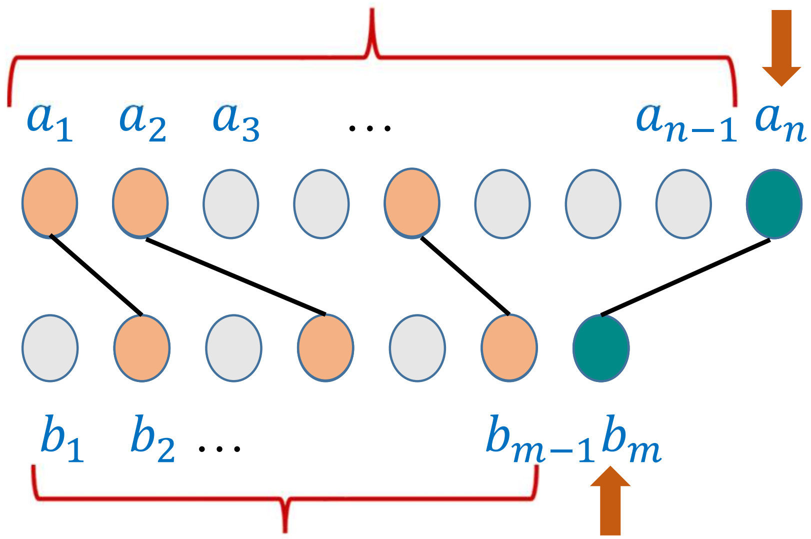



This type of argument is also referred to as **cut-and-paste** argument

**Proof:** (Proof by contradiction)

- Now we can append $a_n$ with $S$ (i.e., $S$ :: $a_n$) and get a subsequence of <u>length one more</u>
- Thus $S$ cannot be the largest subsequence of $a_1, \cdots, a_n$ and $b_1, \cdots, b_m$ (Contradiction)

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

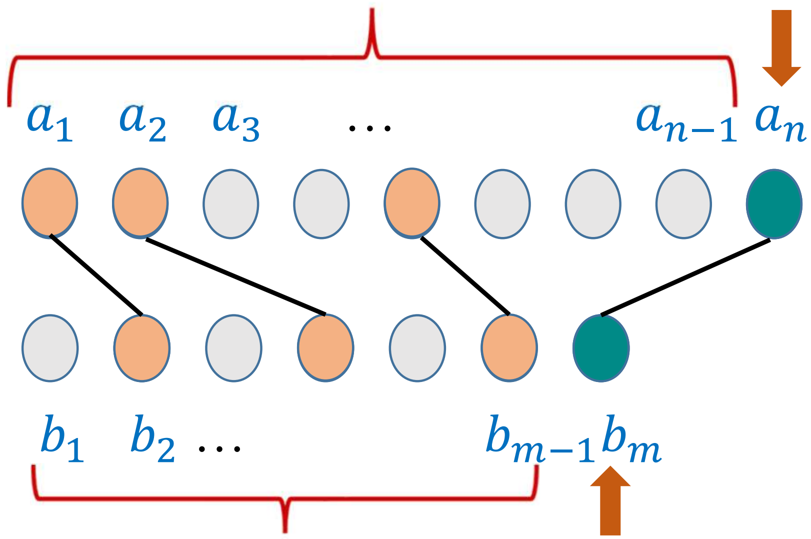



**Proof:** (Proof by contradiction)

- Recall, we need to prove **LCS**($n, m$) = **LCS**($n-1$,$m-1$) :: $a_n$

- So far, we only argued that $a_n$ must the last symbol in **LCS**($n, m$)

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

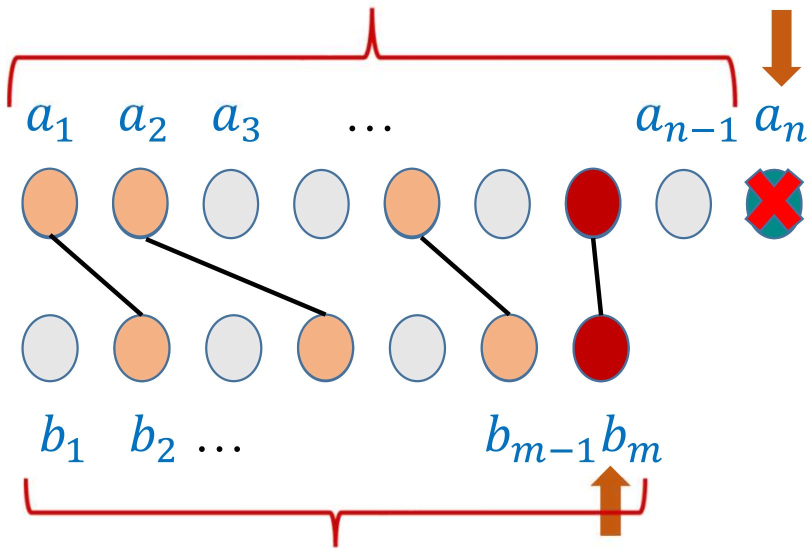



**Proof:** (Proof by contradiction)

- Observe, it is fine to **match** $a_n$ with $b_m$ (since $a_n$ is the last symbol in the **LCS**$(n, m)$)

- So we conclude **LCS**$(n, m)$ = **LCS**$(n-1, m-1)$ :: $a_n$

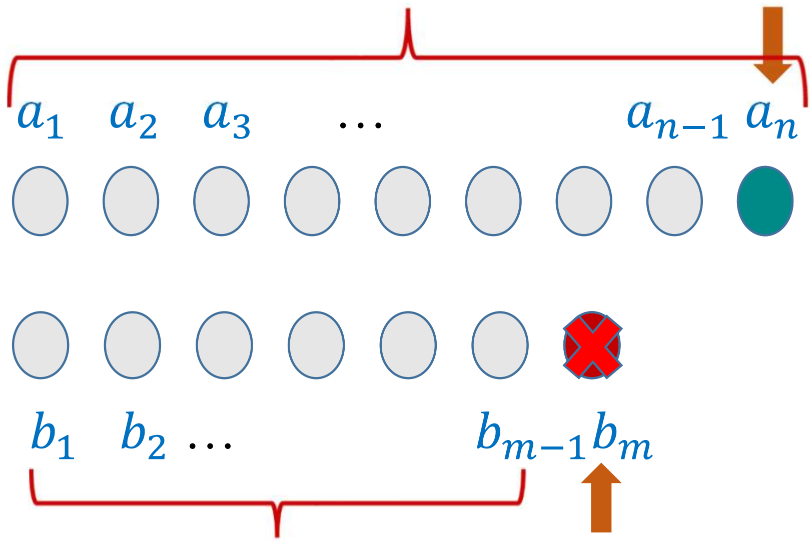
# How does LCS($\boldsymbol{n}$,$\boldsymbol{m}$) look like when $\boldsymbol{a_n \neq b_m}$?




**Intuition**: Either $a_n$ or $b_m$ <u>is not</u> the last symbol of **LCS**($n$,$m$)

**Observation**: If $a_n$ <u>is not</u> the last symbol of **LCS**($n$,$m$) **LCS**($n$,$m$) = **LCS**($n-1$,$m$)

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

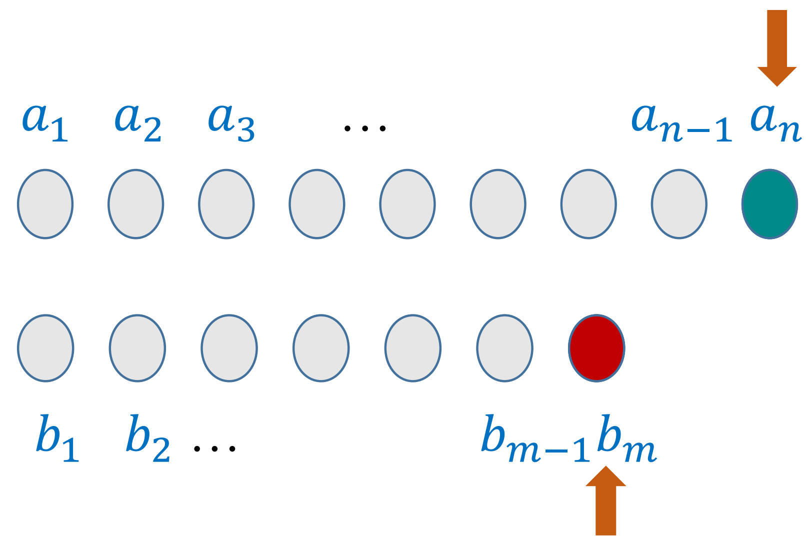



**Intuition**: Either $a_n$ or $b_m$ is not the last symbol of **LCS**($n$,$m$)

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

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

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




**Intuition**: Either $a_n$ or $b_m$ <u>is not</u> the last symbol of **LCS**($n$,$m$)

**Lemma**: If $a_n \neq b_m$ then
**LCS**($n$,$m$) is either **LCS**($n-1$,$m$) or **LCS**($n$,$m-1$)

# Finding LCS: Recursive Formulation

***Optimal substructure***

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

**Base Case**:

**LCS**($i$, $0$) = ∅ for all $i$

**LCS**($0$, $j$) = ∅ for all $j$

**General Case**:

If $a_n = b_m$ then **LCS**($n$,$m$) = **LCS**($n-1$,$m-1$) :: $a_n$

If $a_n \neq b_m$ then **LCS**($n$,$m$) = *bigger* of **LCS**($n$,$m-1$) or **LCS**($n-1$,$m$)

**Simplified Problem**: Find the length of LCS

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

$L(n,m)$ = $0$ if $n$ or $m$ is $0$

# Finding LCS: Recursive Formulation

***Optimal substructure***

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

**Base Case**:

**LCS**($i$, $0$) = ∅ for all $i$

**LCS**($0$, $j$) = ∅ for all $j$

**General Case**:

If $a_n = b_m$ then **LCS**($n$,$m$) = **LCS**($n-1$,$m-1$) :: $a_n$

If $a_n \neq b_m$ then **LCS**($n$,$m$) = <u>*bigger*</u> of **LCS**($n$,$m-1$) or **LCS**($n-1$,$m$)

**Simplified Problem**: Find the length of LCS

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

$L(n,m)$ = $0$ if $n$ or $m$ is $0$

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

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

# Recursive algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$
{ 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**($l_1$, $l_2$);
  }
 }
}

$\boldsymbol{T}(n,m)$ : Worst case running time of $L(n,m)$

$\boldsymbol{T}(n,m) = \boldsymbol{T}(n-1,m) + \boldsymbol{T}(n,m-1)$

A simple exercise from discrete math (not important, you can skip):

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

Exponential !!

But why ?

Let us explore

# Recursive algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$

{ If ($n = 0$ or $m = 0$)

return $0$;

Else

{ If $a_n = b_m$ then

return ($L(n-1,m-1) + 1$);

Else

{ $l_1 \leftarrow L(n-1,m)$ ;

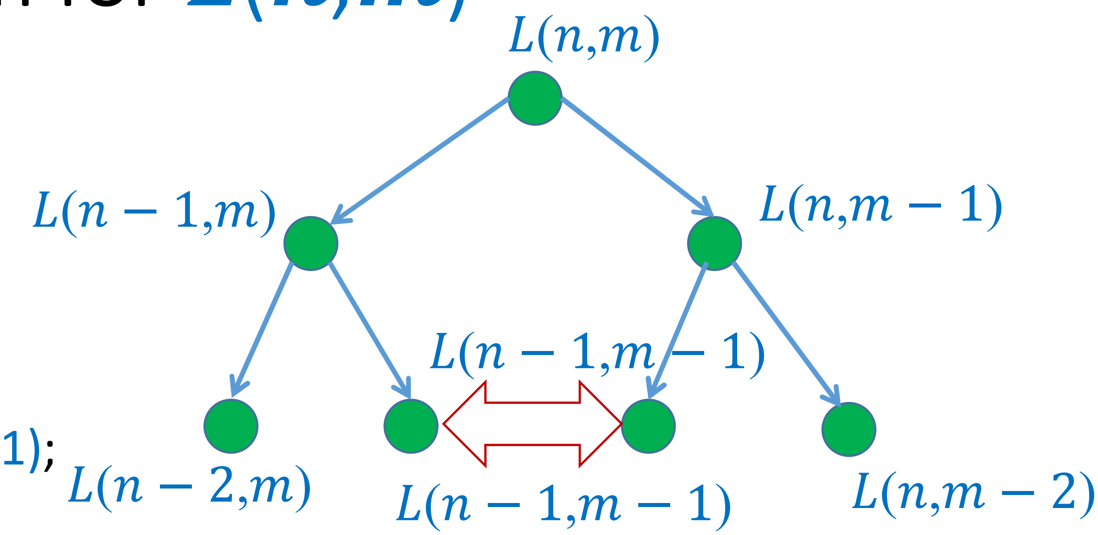
$l_2 \leftarrow L(n,m-1)$ ;

return **Max**($l_1$, $l_2$);

}

}

}



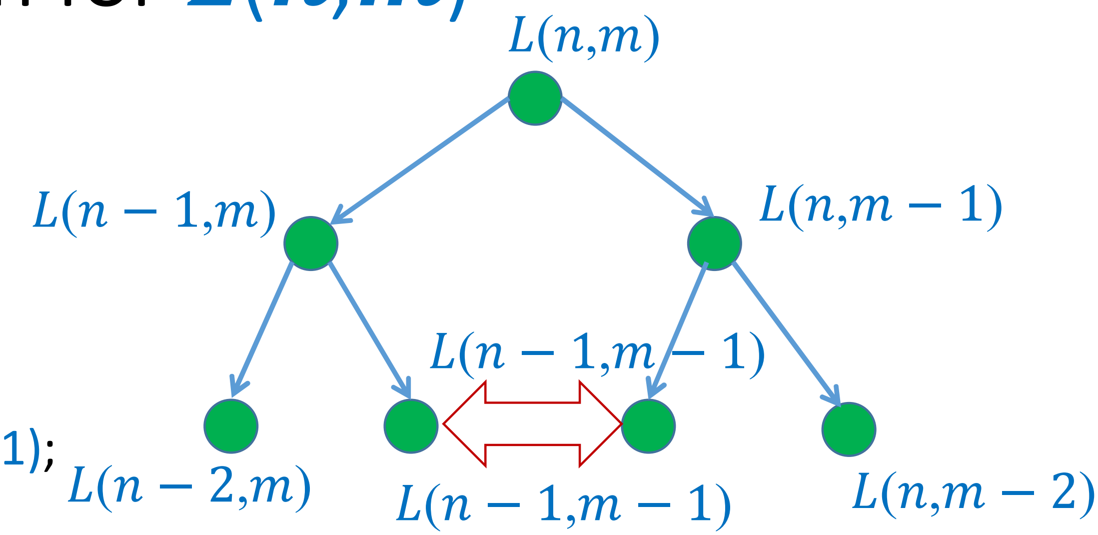

- Solving same sub-problem multiple times !!
- But how many **distinct** sub-problems are there ?
- Only $(n+1) \times (m+1)$

***Overlapping subproblems***

*A recursive solution contains a "small" number of distinct subproblems repeated many times*

# Recursive algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$
{   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 algorithm for Fibonacci number !

# Recursive algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$

{ 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**($l_1$, $l_2$);

  }

 }

}

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $m$ | 0 | | | | | | |
| | 0 | | | | | | |
| | 0 | | | | | | |
| | 0 | | | | | | |
| $\vdots$ | 0 | | | | | | |
| 1 | 0 | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | | … | | | $n$ |

$\boldsymbol{T}$

# Dynamic Programming algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$

{ **For** ($i = 0$ to $n$) $\boldsymbol{T}[i,0] \leftarrow 0$;

**For** ($j = 0$ to $m$) $\boldsymbol{T}[0,j] \leftarrow 0$;

**For** ($j = 1$ to $m$){

**For** ($i = 1$ to $n$){

**If** $a_i = b_j$ then

$\boldsymbol{T}[i,j] \leftarrow \boldsymbol{T}[i-1,j-1] + 1$;

**Else** {

$l_1 \leftarrow \boldsymbol{T}[i-1,j]$ ;

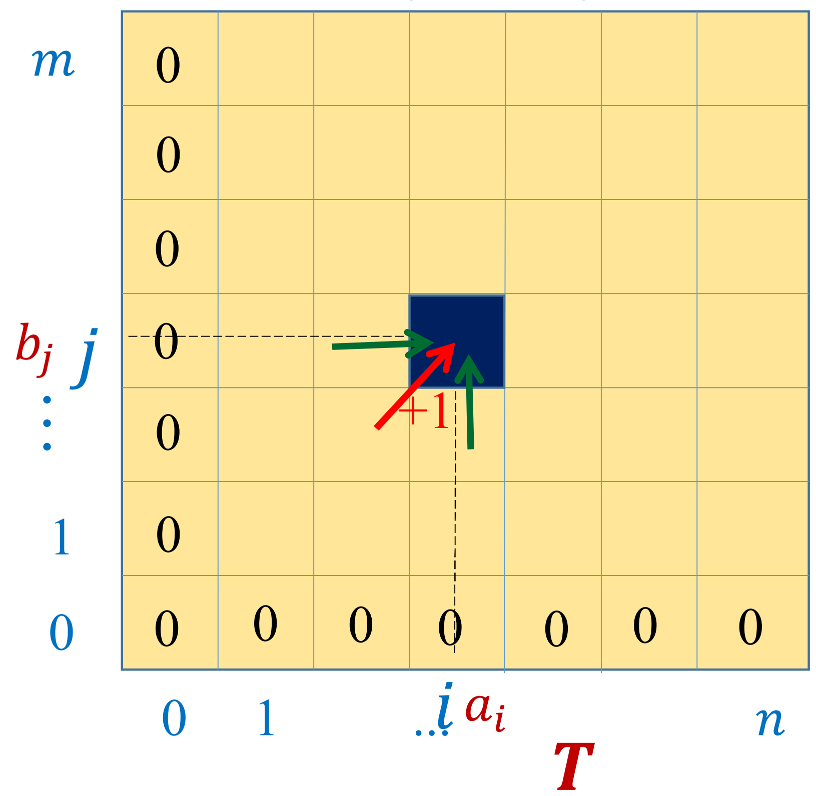
$l_2 \leftarrow \boldsymbol{T}[i,j-1]$ ;

$\boldsymbol{T}[i,j] \leftarrow$ **Max**$(l_1, l_2)$;

} } }

}

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



# Dynamic Programming algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$

{ **For** ($i = 0$ to $n$) $\boldsymbol{T}[i,0] \leftarrow 0$;

**For** ($j = 0$ to $m$) $\boldsymbol{T}[0,j] \leftarrow 0$;

**For** ($j = 1$ to $m$){

**For** ($i = 1$ to $n$){

**If** $a_i = b_j$ then

$\boldsymbol{T}[i,j] \leftarrow \boldsymbol{T}[i-1,j-1] + 1$;

**Else** {

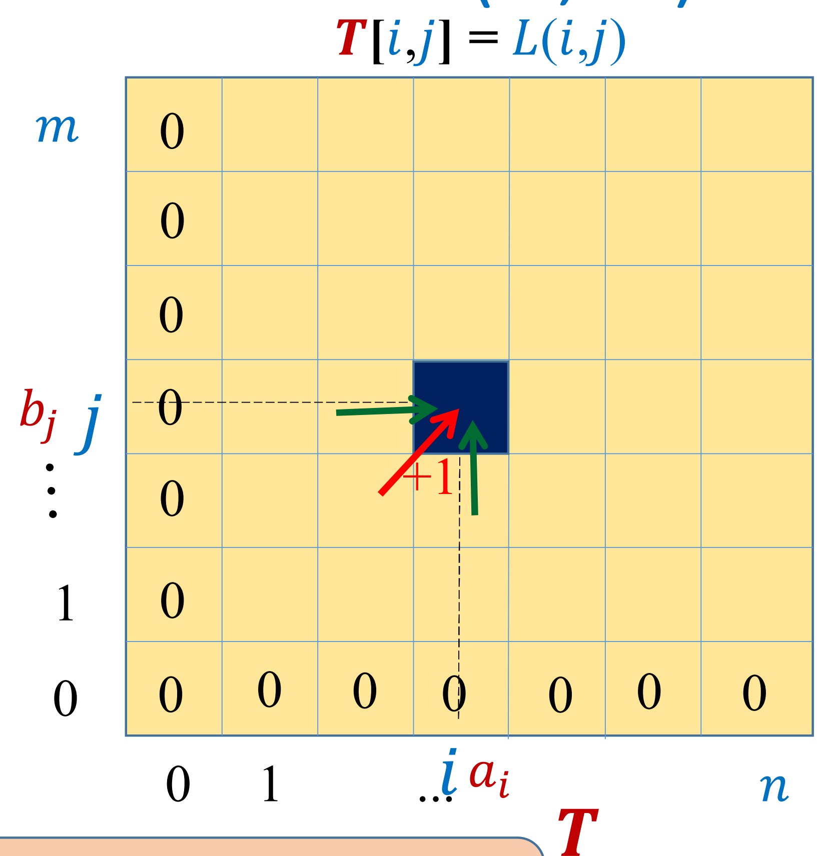
$l_1 \leftarrow \boldsymbol{T}[i-1,j]$ ;

$l_2 \leftarrow \boldsymbol{T}[i,j-1]$ ;

$\boldsymbol{T}[i,j] \leftarrow$ **Max**$(l_1, l_2)$;

} } }

}




Time per table entry = $\boldsymbol{O(1)}$

Total time = $\boldsymbol{O(nm)}$

# Dynamic Programming algorithm for $\boldsymbol{L(n,m)}$

$L(n,m)$

{   **For** ($i = 0$ to $n$)   $\boldsymbol{T}[i,0] \leftarrow 0$;

    **For** ($j = 0$ to $m$)   $\boldsymbol{T}[0,j] \leftarrow 0$;

    **For** ($j = 1$ to $m$){

        **For** ($i = 1$ to $n$){

            **If** $a_i = b_j$ then

                $\boldsymbol{T}[i,j] \leftarrow \boldsymbol{T}[i-1,j-1] + 1$;

            **Else** {

                $l_1 \leftarrow \boldsymbol{T}[i-1,j]$ ;

                $l_2 \leftarrow \boldsymbol{T}[i,j-1]$ ;

                $\boldsymbol{T}[i,j] \leftarrow$ **Max**$(l_1, l_2)$;

            } } }

}

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $a$ | 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 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | | $a$ | $d$ | $c$ | $d$ | $s$ | $a$ |

$\boldsymbol{T}$

# 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** {
$l_1 \leftarrow T[i-1,j]$ ;
$l_2 \leftarrow T[i,j-1]$ ;
$T[i,j] \leftarrow$ **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 $O(\min\{m,n\})$?
  Keep 2 rows (or columns) at a time
- Can you modify the algorithm so that you can output a LCS ?

# Dynamic Programming algorithm paradigm

- Expressing the solution recursively

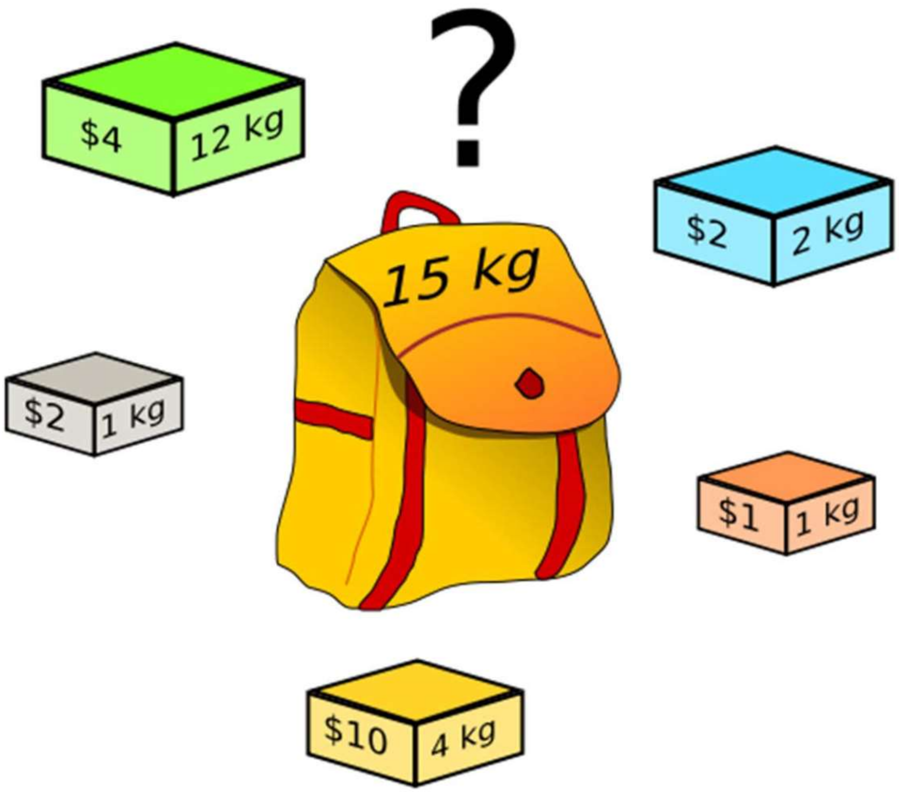
- Overall there are only polynomial number of subproblems

- 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

# Knapsack Problem

# Knapsack Problem




*What is the maximum value you can get?*

$15 (take all items except 12kg item)

# Formal Definition

**KNAPSACK**

**Input:**
$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$, and $W$

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

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

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

# Dynamic Programming

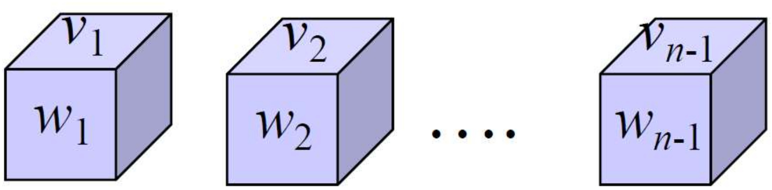
**Problem:** $(w_1, v_1), \ldots, (w_n, v_n), W$

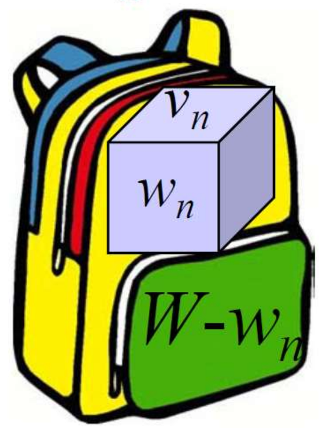

**Is there optimal substructure?**

# Dynamic Programming

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

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

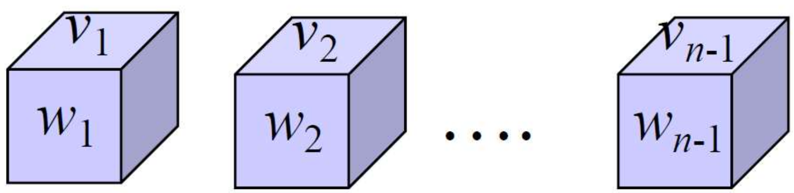


# Dynamic Programming

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

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





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 $\{1,2,\ldots,i\}$
- 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, ..., W:
        m[0, j] ← 0
    for i = 1, ..., n:
        m[i, 0] ← 0
    ⟨Recursive cases⟩

    return m[n, W]
```

# Pseudocode

⟨*Recursive cases*⟩

**for** $i = 1, \ldots, n$:

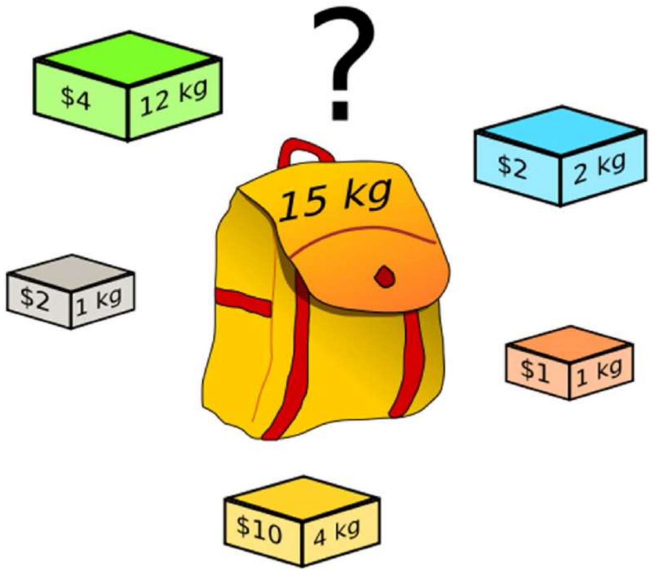
    **for** $j = 0, \ldots, W$:

        **if** $j \geq w[i]$:

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

        **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** | j |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | **1** | 0 | | | | | | | | | | | | | | | | |
| i | **2** | 0 | | | | | | | | | | | | | | | | |
| | **3** | 0 | | | | | | | | | | | | | | | | |
| | **4** | 0 | | | | | | | | | | | | | | | | |
| | **5** | 0 | | | | | | | | | | | | | | | | |

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

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

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

$$W = 15$$

| i \ j | 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** | | | |
| **2** | 0 | | | | | | | | | | | | | | | |
| **3** | 0 | | | | | | | | | | | | | | | |
| **4** | 0 | | | | | | | | | | | | | | | |
| **5** | 0 | | | | | | | | | | | | | | | |

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

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

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

$W = 15$

| i \ j | 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** | | | | | | | | | | |
| **4** | 0 | | | | | | | | | | | | | | | |
| **5** | 0 | | | | | | | | | | | | | | | |

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

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

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

$W = 15$

| i \ j | 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 | | | | | | | | | | | | | | | |

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

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

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

$W = 15$

| | | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | j |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **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 | |
| i | **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 | 13 | 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 | **15** | |

# Pseudocode

representing $W$ takes only $O(\lg W)$ bits

⟨*Recursive cases*⟩

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

NOT a polynomial-time algorithm
but "pseudo polynomial-time" polynomial in the value (not in the size)

**for** $i = 1, \ldots, n$:

**for** $j = 0, \ldots, W$:

**if** $j \geq w[i]$:

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

**else**:

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

# Example: Changing Coins

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 $n =$ 30c 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 $i < j$.

# Example: Changing Coins

**Optimal substructure:** Suppose $M[j] = t$, meaning that

$$j = d_{i_1} + d_{i_2} + \cdots + d_{i_t}$$

for some $i_1, \dots, i_t \in \{1, \dots, k\}$. Then, if $j' = d_{i_1} + d_{i_2} + \cdots + d_{i_{t-1}}$, $M[j'] = t - 1$, because otherwise if $M[j'] < t - 1$, by **cut-and-paste** argument, $M[j] < t$.

$$M[j] = \begin{cases} 1 + \min\limits_{i \in [k]} M[j - d_i], & j > 0 \\ 0, & j = 0 \\ \infty, & j < 0 \end{cases}$$

# Example: Changing Coins

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

# Example: Changing Coins

NUM-COINTS-DP$(n, d)$:

**for** $j = 0, \ldots, n$:

$\quad M[j] \leftarrow \infty$

$M[0] \leftarrow 0$

**for** $j = 1, \ldots, n$:

$\quad$ **for** $i = 1, \ldots, k$:

$\quad\quad$ **if** $(j - d_i \geq 0) \wedge (M[j - d_i] + 1 < M[j])$:

$\quad\quad\quad M[j] \leftarrow M[j - d_i] + 1$

**return** $M[n]$

Running time = $\boldsymbol{O(nk)}$

# 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. Diptarka Chakraborty
  - The slides from Prof. Arnab Bhattacharyya