

# Question of the day

---

- How did dynamic programming work again?
- How can a dynamic programming problem be solved iteratively, when dynamic programming is so strongly related to recursion?
- What's an algorithm design strategy that's applicable to more problems than divide-and-conquer, and more efficient and much easier to implement than dynamic programming?
  - What's the catch?

# Dynamic programming and greedy algorithms

**William Hendrix**

# Outline

---

- Review
  - Design strategies
  - Memoization
- Longest common subsequence
- Analyzing memoized algorithms
- Iterative dynamic programming

# Algorithm design review

---

- **Brute force**
  - A.k.a., exhaustive search
  - Try every possible solution
  - Applicable to all problem
  - Often unworkably slow
- **Divide-and-conquer**
  - Divide problem, solve recursively, and combine solutions
    - Reduce recursive calls if possible
  - Needs *optimal substructure*
    - Can efficiently compute full solution from recursive calls
  - Inefficient when subproblems overlap
- **Data organization**
  - Use appropriate data structures to improve complexity
  - Complements any other strategy
  - Time/space trade-off

# Dynamic programming review

---

- Algorithm design strategy related to divide-and-conquer
  - Make recursive calls, combine solutions, solve base cases directly
  - Recursive calls repeated => save solutions in lookup table
- Often reduces exponential algorithms to polynomial time
- Can be solved with *memoization* or iteratively
  - Function that stores return values in lookup table
  - Not limited to recursive functions/dynamic programming
- **Map** usually array-based
  - Dimensionality depends on parameters
  - Fibonacci(n): 1D array
  - Binomial(n, k): 2D array
- **Sentinel value**
  - Special value for a problem that's not yet been solved
  - Must not be a solution
  - E.g., -1 for Fibonacci or Binomial

# Memoization summary

- Five step procedure
  - 1. Start with naïve recursive algorithm**
    - Based on recurrence
  - 2. Decide data structure and sentinel value**
  - 3. Add “memoization check” to the beginning of the algorithm**
    - If solution is in data structure, return it
  - 4. Store solution before returning**
  - 5. Write wrapper function to initialize data structure**

$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \end{cases}$$

**Input:**  $n, k$ : binomial coefficient to compute

**Output:**  $\binom{n}{k}$

```
1 Algorithm: DPBinom
2  $binom = \text{Array}(n, k)$ 
3 Initialize  $binom$  to 0
4 return MemoBinom( $n, k$ )
```

```
1 Algorithm: MemoBinom
2 if  $binom[n, k] > 0$  then
3   | return  $binom[n, k]$ 
4 else if  $k = 0$  or  $k = n$  then
5   |  $binom[n, k] = 1$ 
6 else
7   |  $binom[n, k] = \text{MemoBinom}(n-1, k-1)$ 
   |    $+ \text{MemoBinom}(n-1, k)$ 
8 end
9 return  $binom[n, k]$ 
```

# Another DP application

- `diff`
  - Unix utility to compare files for changes
  - Identifies lines to be **added** or **removed**
  - Based on *longest common subsequence* problem
- Longest common subsequence
  - Given two strings,  $a$  and  $b$
  - Compute string  $c$  so that characters of  $c$  appear in both
    - In same order

- **Example**

g	a	r	b	a	g	e
---	---	---	---	---	---	---

g	a	r				e
---	---	---	--	--	--	---

		g	a	r		e	
--	--	---	---	---	--	---	--

b	o	g	a	r	t	e	d
---	---	---	---	---	---	---	---

- **LCS:** gare

# Modelling the problem recursively

- Three options when comparing character-by-character

- Match  $a_i$  with  $b_j$  (if possible)
- Skip letter in  $a$
- Skip letter in  $b$

- Each one reduces length of  $a$  or  $b$  or both

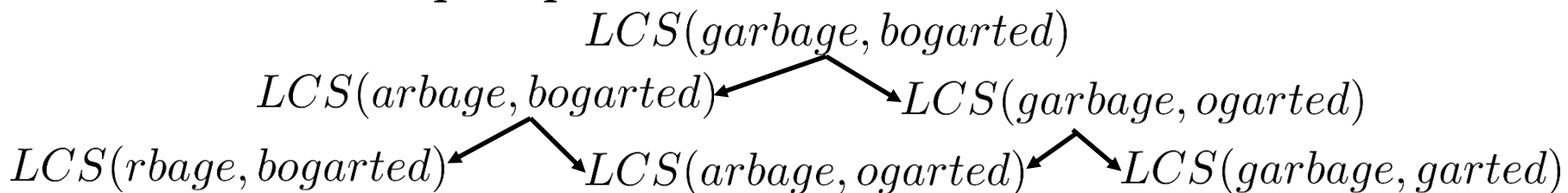
- Solve remainder *recursively*
- $LCS(a', b')$  or  $LCS(a', b)$  or  $LCS(a, b')$

Strings with first character removed

- Formally:  $LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a', b) \\ LCS(a, b') \end{cases}, & \text{otherwise} \end{cases}$
- Recursive function

- Accepts start position in each string

- Does LCS overlap subproblems?





# Formalizing the problem

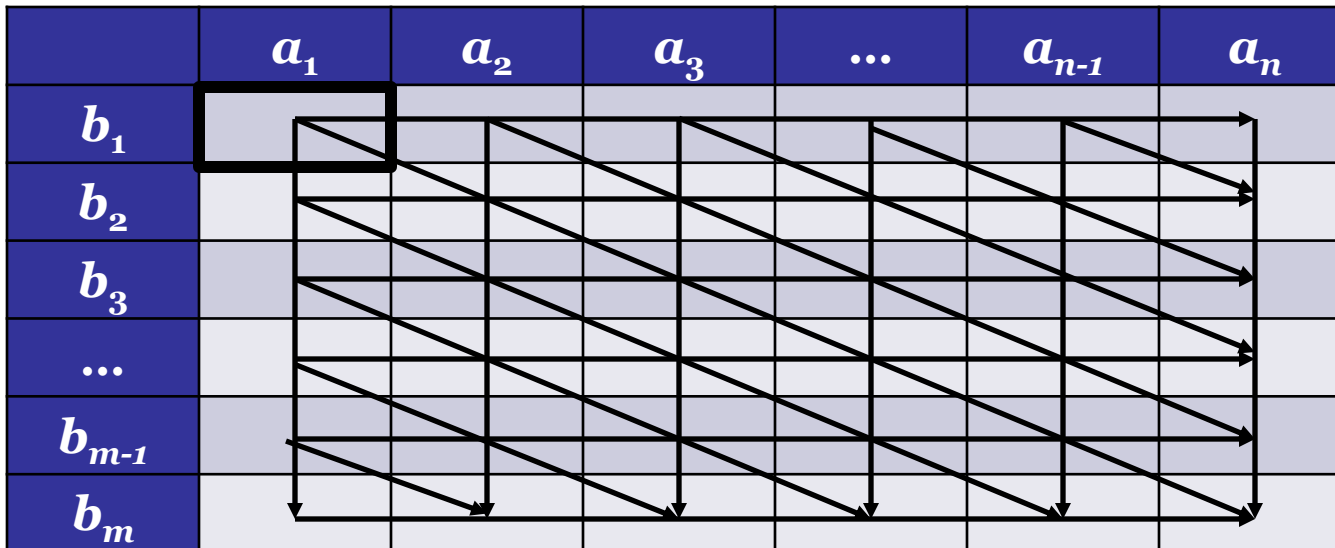
$$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a', b) \\ LCS(a, b') \end{cases} & , \text{ otherwise} \end{cases}$$

The diagram shows a dynamic programming table for the Longest Common Subsequence (LCS) problem. The table has columns labeled  $a_1, a_2, a_3, \dots, a_{n-1}, a_n$  and rows labeled  $b_1, b_2, b_3, \dots, b_{m-1}, b_m$ . The cell at the intersection of  $a_1$  and  $b_1$  is highlighted with a thick black border. From this cell, three arrows point to the adjacent cells: one to the right (to  $a_2, b_1$ ), one down (to  $a_1, b_2$ ), and one diagonally down-right (to  $a_2, b_2$ ). A blue 'X' is drawn in the cell at  $a_3, b_2$ . A blue bracket on the left side of the table groups the rows  $b_1$  through  $b_m$ . A blue bracket on the top side of the table groups the columns  $a_3$  through  $a_n$ .

	$a_1$	$a_2$	$a_3$	$\dots$	$a_{n-1}$	$a_n$
$b_1$						
$b_2$						
$b_3$						
$\dots$						
$b_{m-1}$						
$b_m$						

# Dependency structure

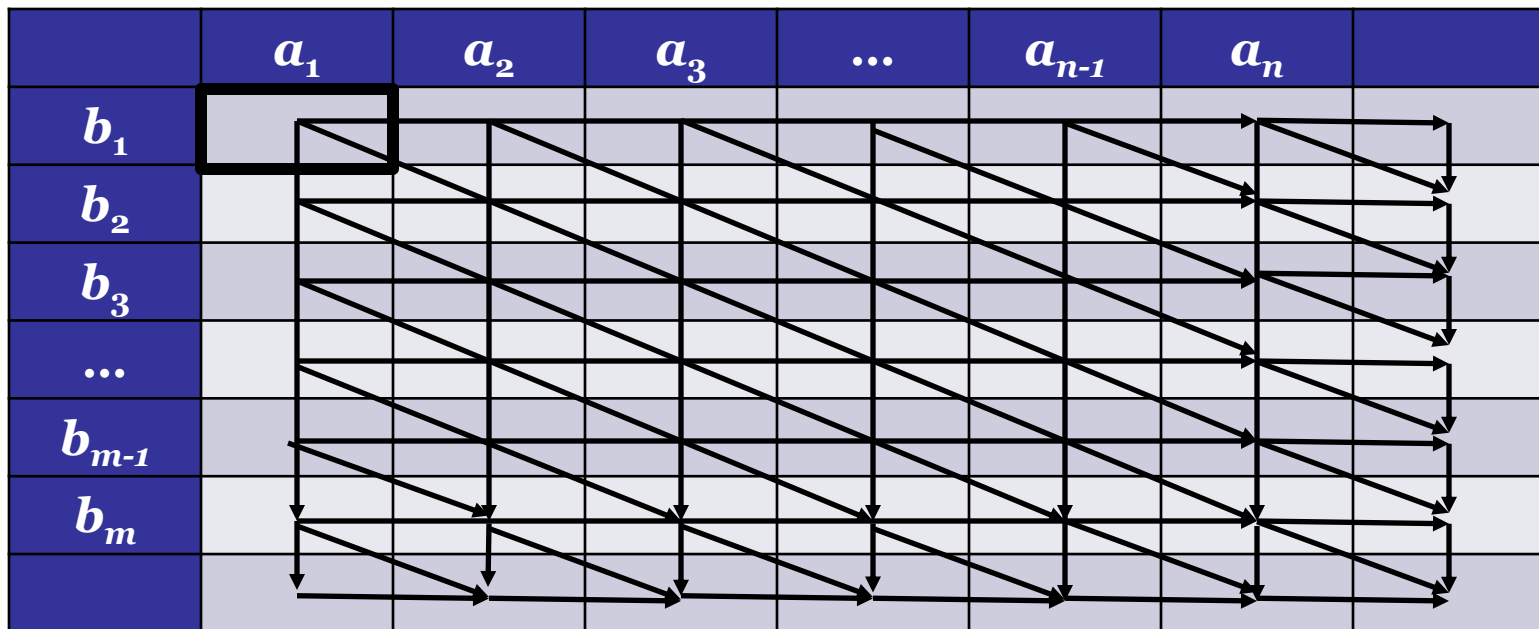
$$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a', b) \\ LCS(a, b') \end{cases} & , \text{ otherwise} \end{cases} \quad \text{Base cases?}$$



# Dependency structure

$$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a', b) \\ LCS(a, b') \end{cases} & , \text{ otherwise} \end{cases}$$

Base cases  
 $LCS(a, \epsilon) = \epsilon$   
 $LCS(\epsilon, b) = \epsilon$



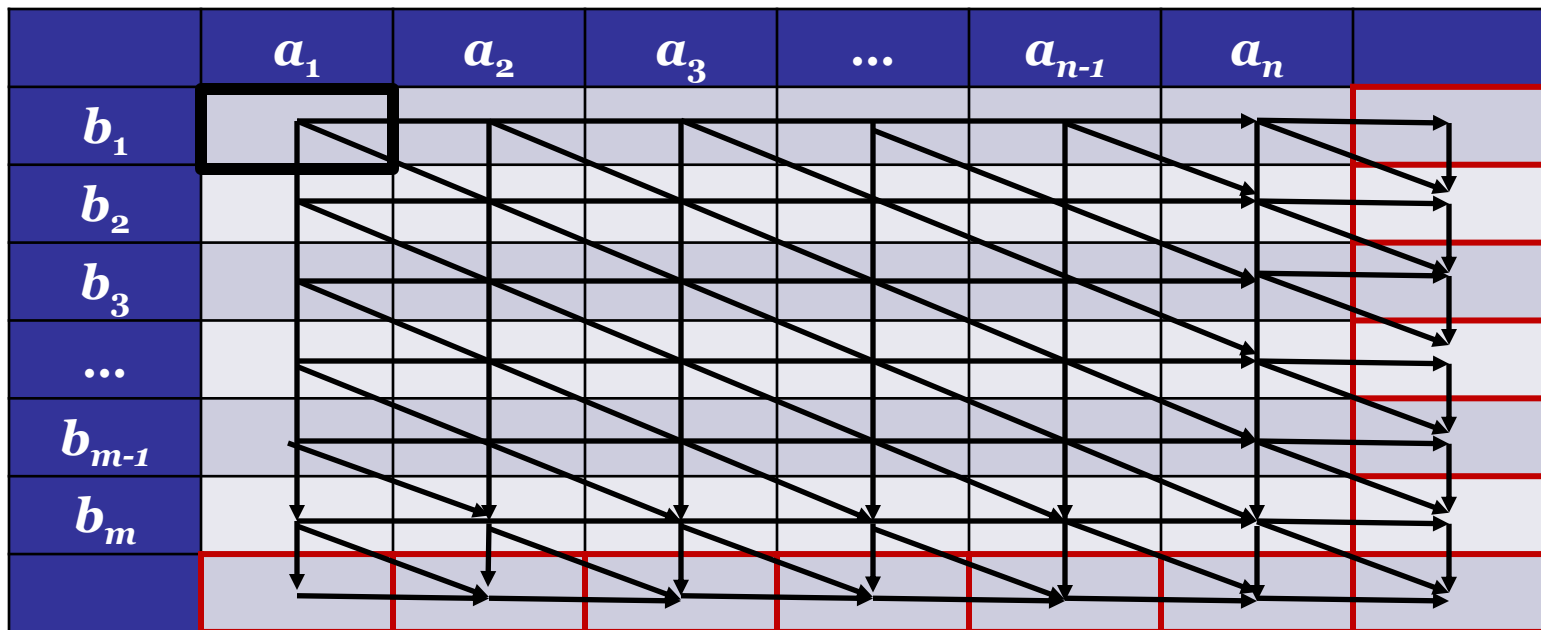
# Dependency structure

$$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a', b) \\ LCS(a, b') \end{cases} & , \text{ otherwise} \end{cases}$$

Base cases

$LCS(a, \epsilon) = \epsilon$

$LCS(\epsilon, b) = \epsilon$



# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	✗			
<i>a</i>				
<i>r</i>				
<i>e</i>				

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	→ → →			ε
<i>a</i>				
<i>r</i>				
<i>e</i>				

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>				← $\epsilon$
<i>a</i>				↓ $\epsilon$
<i>r</i>				
<i>e</i>				

# Tracing example

- LCS(*air*, *care*):


	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>				€
<i>a</i>				← €
<i>r</i>				↓
<i>e</i>				↘ €



# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>				€
<i>a</i>				€
<i>r</i>			<i>r</i>	
<i>e</i>				€



# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>				€
<i>a</i>			<i>r</i>	€
<i>r</i>			<i>r</i> ↑ <i>r</i>	
<i>e</i>				€

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>			<i>r</i>	€
<i>r</i>			<i>r</i>	
<i>e</i>				€

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>			<i>r</i>	€
<i>r</i>			<i>r</i>	
<i>e</i>				€
			€	

The diagram illustrates the alignment of the longest common subsequence (LCS) between the strings "air" and "care". The table shows the characters of "air" (rows) and "care" (columns). Arrows indicate the mapping of characters from "air" to "care":

- 'a' in "air" maps to 'a' in "care".
- 'i' in "air" maps to 'i' in "care".
- 'r' in "air" maps to 'r' in "care".
- 'e' in "air" maps to 'e' in "care".

The empty string '€' is used for non-matching positions. The alignment shows that the LCS is "aire".

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>			<i>r</i>	€
<i>r</i>			<i>r</i>	
<i>e</i>			€	€
			↑ €	

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>			<i>r</i>	€
<i>r</i>			<i>r</i>	
<i>e</i>			€	€
		€	€	

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>			<i>r</i>	€
<i>r</i>			<i>r</i>	
<i>e</i>		€	€	€
		€	€	

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>			<i>r</i>	€
<i>r</i>		<i>r</i> ↑	<i>r</i>	
<i>e</i>		€	€	€
		€	€	



# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>			<i>r</i>	€
<i>a</i>		<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

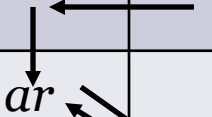
# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>		<i>r</i>	<i>r</i>	€
<i>a</i>		<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>		<i>r</i>	<i>r</i>	€
<i>a</i>		<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i> ↑	<i>r</i>	<i>r</i>	€
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

# Tracing example

- LCS(*air*, *care*):

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

# LCS: memoization

- $$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a', b) \\ LCS(a, b') \end{cases}, & \text{otherwise} \end{cases}$$
- Implement recurrence directly
- Rely on map for previous subproblems
- Write outer function to initialize map
  - Need to choose sentinel value carefully

Base cases

$LCS(a, \epsilon) = \epsilon$

$LCS(\epsilon, b) = \epsilon$

```
1 Algorithm: MemoLCS(a, i, b, j)
2 if lcs[i, j] ≠ NIL then
3   | return lcs[i, j]
4 else if i = n + 1 or j = m + 1 then
5   | lcs[i, j] =  $\epsilon$ 
6 else if a[i] = b[j] then
7   | lcs[i, j] = a[i] + MemoLCS(a, i + 1, b, j + 1)
8 else
9   | o1 = MemoLCS(a, i + 1, b, j)
10  | o2 = MemoLCS(a, i, b, j + 1)
11  | lcs[i, j] = max(o1, o2)
12 end
13 return lcs[i, j]
```

```
Input: a, b: two strings
Input: n, m: length of a and b, respectively
Output: LCS of a and b
1 Algorithm: DPLCS
2 lcs = Array(n, m)
3 Initialize all cells of lcs to NIL
4 return MemoLCS(a, 1, b, 1)
```

# An alternative approach

- Consider the problem of computing the *size* of the LCS
- Solution is very similar:

$$LCSlen(i, j) = \begin{cases} 0, & \text{if } i = n + 1 \text{ or } j = m + 1 \\ 1 + LCSlen(i + 1, j + 1), & \text{if } a_i = b_j \\ \max(LCSlen(i + 1, j), LCSlen(i, j + 1)), & \text{otherwise} \end{cases}$$

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	$\epsilon$
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	$\epsilon$
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		$\epsilon$	$\epsilon$	$\epsilon$
		$\epsilon$	$\epsilon$	$\epsilon$



	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	2	1	1	0
<i>a</i>	2	1	1	0
<i>r</i>		1	1	
<i>e</i>		0	0	0
		0	0	0

+

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	↓	→	↓	
<i>a</i>	↘	→	↓	
<i>r</i>		→	↘	
<i>e</i>				

- Observation:** we can reconstruct LCS string if we also store where max came from
  - Follow "pointers" from start to base case
  - Add characters when we move diagonally
- Extra  $\Theta(n+m)$  processing step (doesn't change complexity)
- Reduces space complexity

# Analyzing memoization

- $LCS(\text{air}, \text{care})$ :

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

$$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max(LCS(a', b), LCS(a, b')), & \text{otherwise} \end{cases}$$

## Two questions

- How many cells are filled in (non-memoized calls)?
- How long does it take to fill in one cell?

**Total time** = # cells \* Cost per cell + Initialization cost

- Could potentially require summation



# Analyzing our solution

- $LCS(\text{air}, \text{care})$ :

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

$$LCS(a, b) = \begin{cases} a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max(LCS(a', b), LCS(a, b')), & \text{otherwise} \end{cases}$$

## Worst-case analysis

- $O(nm)$  cells
- $\Theta(1)$  time per cell
- Total time:  $O(nm)$

# Iterative dynamic programming

---

## Main idea

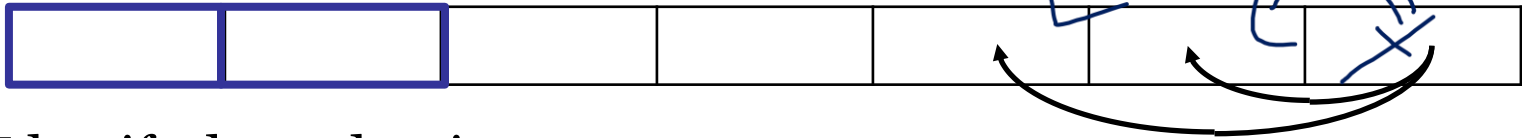
- Recursion not needed if dependent cells already filled in

## Process

- Analyze dependencies to choose appropriate order
- Write for loop(s)
- Inside loop(s): naïve recursive algorithm
  - Replace calls with lookups
  - Replace return with assignment
  - Change variable to loop variable
- Allocate data structure beforehand

# Iterative example

- Fibonacci numbers
  - Each value is sum of two previous values
- **Recurrence:**  $F(n) = F(n - 1) + F(n - 2)$
- One parameter
  - 1D array-based map

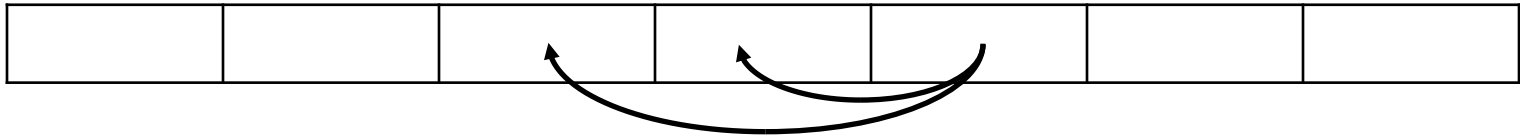


- Identify dependencies
  - Each cell needs the two cells to the left
  - Move in *opposite* direction
- Base cases
  - Must start here
- Need to iterate left-to-right
  - When calculating  $F(n)$ ,  
 $F(n-1)$  and  $F(n-2)$  already solved

```
Input:  $n$ : index of Fibonacci number to compute
Output:  $F_n$ 
1 Algorithm: IterativeFib
2  $fib = \text{Array}(n)$ 
3 for  $i = 1$  to  $n$  do
4   if  $i = 1$  or  $2$  then
5      $fib[i] = 1$ 
6   else
7      $fib[i] = fib[i - 1] + fib[i - 2]$ 
8   end
9 end
10 return  $fib[n]$ 
```

# Optimizing iterative DP

- Sometimes possible to improve space complexity
- **Main idea**
  - Don't store entire data structure
  - Only store what's necessary to calculate future values
- **Example:** Fibonacci numbers



- Each cell only needs previous two values
  - Values not useful after moving three away
- *Optimization:* only store previous two values
  - Constant space complexity

```
Input:  $n$ : index of Fibonacci number to compute
Output:  $F_n$ 
1 Algorithm: BestFib
2 if  $n = 1$  or  $2$  then
3   | return 1
4 end
5  $prev = 1$ 
6  $twoprev = 1$ 
7 for  $i = 3$  to  $n$  do
8   |  $curr = prev + twoprev$ 
9   |  $twoprev = prev$ 
10  |  $prev = curr$ 
11 end
12 return  $curr$ 
```

# Iterative DP example

- Longest common subsequence

$$LCS(a, b) = \begin{cases} \epsilon, & \text{if } a = \epsilon \text{ or } b = \epsilon \\ a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max(LCS(a', b), LCS(a, b')), & \text{otherwise} \end{cases}$$

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>				
<i>a</i>				
<i>r</i>				
<i>e</i>				

- What order should we iterate through the array?

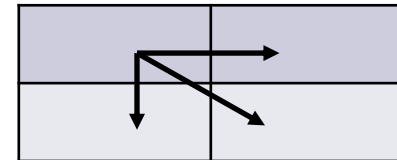
# Iterative DP example

- Longest common subsequence

$$LCS(a, b) = \begin{cases} \epsilon, & \text{if } a = \epsilon \text{ or } b = \epsilon \\ a_1 + LCS(a', b'), & \text{if } a_1 = b_1 \\ \max(LCS(a', b), LCS(a, b')), & \text{otherwise} \end{cases}$$

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		<i>r</i>	<i>r</i>	
		<i>r</i>	<i>r</i>	

Dependency structure:



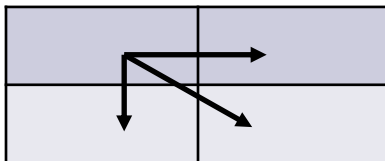
- What order should we iterate through the array?
  - Base cases are on bottom and right side of array
  - Cells to the right, bottom, and diagonal must be filled
  - Bottom-to-top, right-to-left works well
  - Other orders are possible

# Space complexity

- Can also reduce space complexity for LCS
- **Example**

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	<i>r</i>
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	<i>r</i>
<i>r</i>		<i>r</i>	<i>r</i>	<i>r</i>
<i>e</i>		<i>r</i>	<i>r</i>	<i>r</i>
		<i>r</i>	<i>r</i>	<i>r</i>

*Dependency structure:*

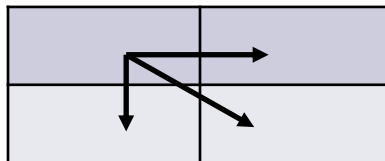


# Space complexity

- Can also reduce space complexity for LCS
- **Example**

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	↖
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	↖
<i>r</i>		<i>r</i>	<i>r</i>	↖
<i>e</i>		↖	↖	↖
		↖	↖	↖

*Dependency structure:*



**Input:** *a*, *b*: strings to calculate LCS for  
**Input:** *n*, *m*: length of *a* and *b*, respectively  
**Output:** LCS of *a* and *b*

```

1 Algorithm: IterativeLCS
2 prev = Array(n + 1)
3 curr = Array(n + 1)
4 Initialize prev with  $\epsilon$ 
5 for i = n down to 1 do
6   curr[m + 1] =  $\epsilon$ 
7   for j = m down to 1 do
8     if a[i] = b[j] then
9       curr[j] = b[j] + prev[j + 1]
10    else
11      curr[j] = max{curr[j + 1], prev[j]}
12    end
13  end
14  prev = curr
15 end
16 return curr[1]
```

Only need previous column



# Dynamic programming analysis

- Applies to recursive problems with overlap
- Generally reduces exponential-time to polynomial
- Increases space requirements
- Can be tricky

- Memoization

- Easier to program
  - May skip some cases

	<i>a</i>	<i>i</i>	<i>r</i>	
<i>c</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>a</i>	<i>ar</i>	<i>r</i>	<i>r</i>	€
<i>r</i>		<i>r</i>	<i>r</i>	
<i>e</i>		€	€	€
		€	€	

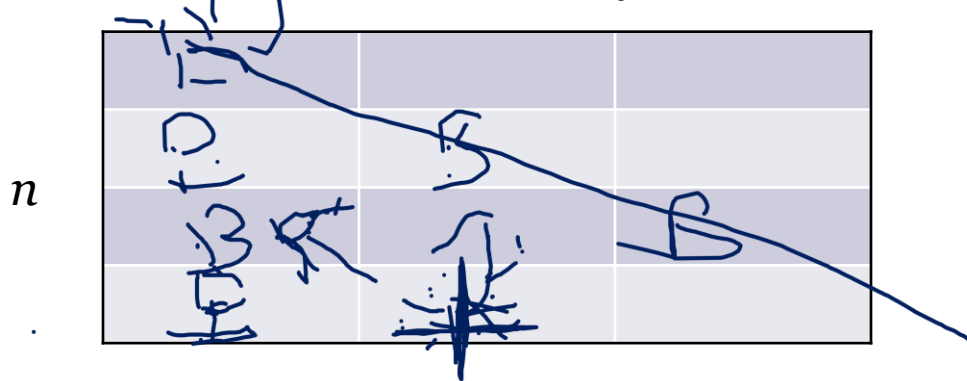
- Iteration

- Often reduces memory requirements
  - Easier to analyze
  - Lower coefficients
    - Unless memoization skips too many cases

# Iterative DP exercise

- Binomial coefficients (“ $n$  choose  $k$ ”)

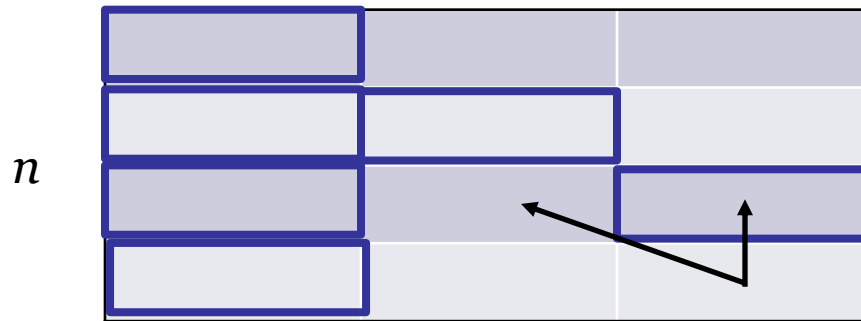
- Recurrence:** 
$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \end{cases}$$



1. Where are base cases?
2. Which direction(s) do the recursive dependencies point?
3. What order should we iterate through array?
4. Describe for loop(s) that iterate in this order.
5. Give pseudocode for an iterative DP algorithm.

# Iterative DP exercise

- Binomial coefficients (“ $n$  choose  $k$ ”)
- Recurrence:** 
$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$

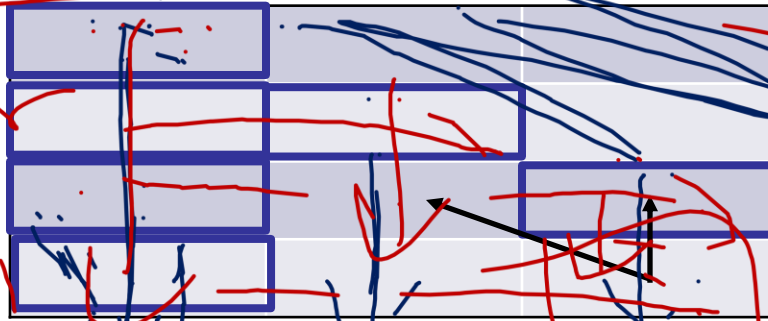


- Where are base cases?  
 **$k = 0$  and  $n = k$**
- Which direction(s) do the recursive dependencies point?  
**Up and up-left**

# Iterative DP exercise

- Binomial coefficients (“ $n$  choose  $k$ ”)

- Recurrence:** 
$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \end{cases}$$

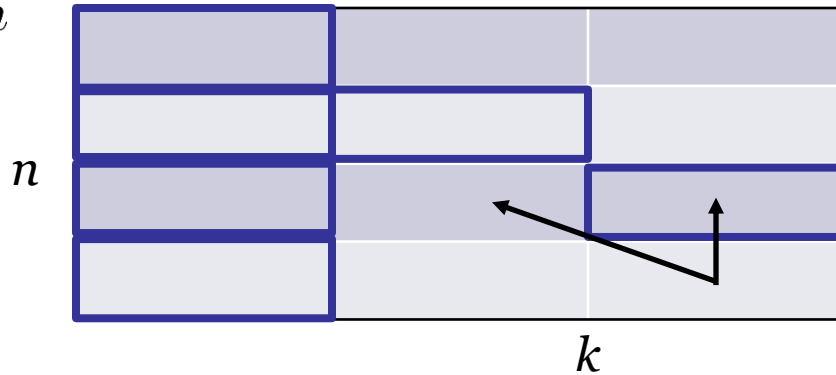


- What order should we iterate through array?
- Describe for loop(s) that iterate in this order.
- Give pseudocode for an iterative DP algorithm.

$dp = \text{Array}(n+1, k+1)$   
 for  $j = 0$  to  $k$   
 for  $i = j$  to  $n$   
 $dp[i][j] =$   
 $dp[i-1][j-1]$   
 $+ dp[i-1][j]$   
 return  $dp[n][k]$

# Iterative DP exercise

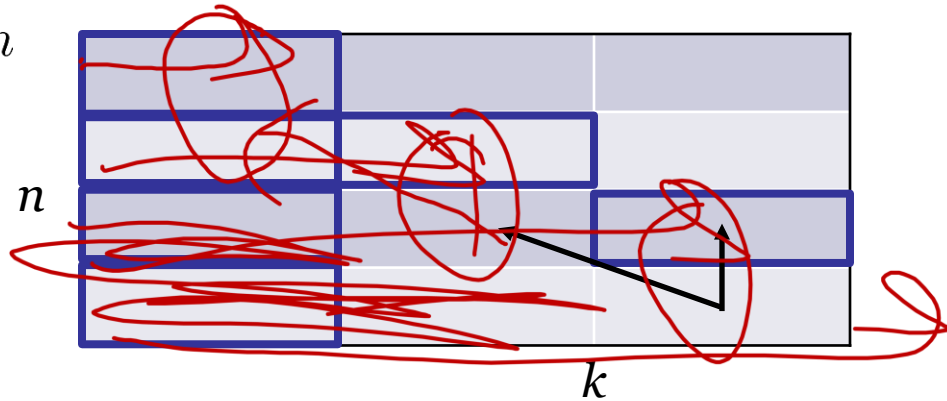
$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \end{cases}$$



3. What order should we iterate through the array?
4. Write for loop(s) that iterate in this order.
5. Give pseudocode for an iterative DP algorithm.

# Iterative DP exercise

$$\binom{n}{k} = \begin{cases} 1, & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \end{cases}$$



3. What order should we iterate through array?

Multiple options

E.g., left-to-right, top-to-bottom

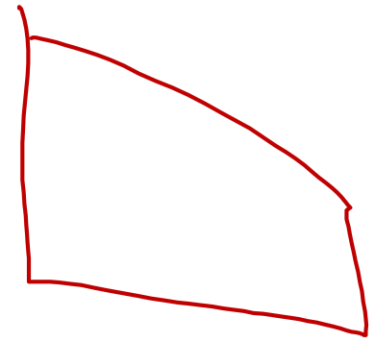
4. Describe for loop(s) that iterate in this order.

for  $i = 0$  to  $n$

for  $j = 0$  to  $\min\{i, k\}$

for  $j = 0$  to  $k$

for  $i = j$  to  $j + n - k$



# Iterative DP exercise

**Input:**  $n$ : number of objects in total

**Input:**  $k$ : number of objects to pick

**Output:** number of ways to select  $k$  objects from a pool of  $n$

1 **Algorithm:** NChooseK

2  $C = \text{Array}(n, k)$

3 **for**  $i = 0$  to  $n$  **do**

4     **for**  $j = 0$  to  $\min\{i, k\}$  **do**

5         **if**  $j = 0$  or  $j = i$  **then**

6              $C[i, j] = 1$

7         **else**

8              $C[i, j] = C[i - 1, j - 1] + C[i - 1, j]$

9         **end**

10     **end**

11 **end**

12 **return**  $C[n, k]$

# Strategy: greedy algorithms

---

- Usually applied to *optimization* problems
  - "Find the best/largest/smallest/etc ..."
- **Strategy outline**
  - Formulate the problem as sequence of decisions
    - E.g., which element to add to/remove from a set
  - Always select the best available option
    - "Best" depends on problem
      - "Local optimum"
  - Continue until solution cannot be improved



# Greedy example

---

- **Problem:** knapsack problem
- **Input:** array of positive integers *data* (size  $n$ ), and target weight  $t$
- **Output:** largest subset of data with sum  $\leq t$
  
- **Example:**  $data = \{2, 3, 8, 7, 7, 6\}$ ,  $t = 21$

# Greedy example

---

- **Problem:** knapsack problem
- **Input:** array of positive integers *data* (size  $n$ ), and target weight  $t$
- **Output:** largest subset of data with sum  $\leq t$
  
- **Example:**  $data = \{2, 3, 8, 7, 7, 6\}$ ,  $t = 21$ 
  - *Solution:*  $\{2, 3, 6, 7\}$  or  $\{2, 3, 6, 8\}$
  
- 1. What is our sequence of decisions?
  - What number do we add next?
- 2. Which option is best?
  - Add the smallest remaining element

# Example greedy algorithm

**Input:** *data*: array of positive integers

**Input:** *n*: size of *data*

**Input:** *t*: target value










**Output:** largest subset of *data* with sum  $\leq t$

```
1 Algorithm: GreedyKnapsack
2 heap = Heapify(data)
3 soln = {}
4 sum = 0
5 next = heap.DeleteMin()
6 while sum + next  $\leq t$  do
7   |   Add next to soln
8   |   sum = sum + next
9   |   next = heap.DeleteMin()
10 end
11 return soln
```

- Alternatively: sort first, then iterate
  - Same answer and time complexity
- Is the answer correct?

# Algorithm design example

- Consider the following problem:
- Problem:** workshop scheduling
  - Input:** the start times and durations for a set of workshops
  - Output:** the largest number of workshops whose times do not overlap
- Example instance:**

	Start	Dur.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	1	7														
B	2	3														
C	3	6														
D	6	3														
E	8	2														
F	9	3														
G	9	5														
H	11	1														
I	13	1														

# GreedySearch

**Input:** *start*: an array of start times for workshops

**Input:** *duration*: an array of durations for workshops

**Input:** *n*: the number of workshops

**Output:** Maximum possible workshops to attend

```
1 Algorithm: GreedyWorkshop
2 best = 0
3 subset = {}
4 Sort start and duration by _____
5 while start and duration not empty do
6   | Add (start[1], duration[1]) to subset
7   | Remove all workshops from start and duration
   |   that overlap this workshop
8   | best = best + 1
9 end
10 return best
```

# GreedySearch variants

---

- Earliest workshop first
- Shortest workshop first

# GreedySearch variants

- Earliest workshop first

- **Counterexample:**



- Shortest workshop first

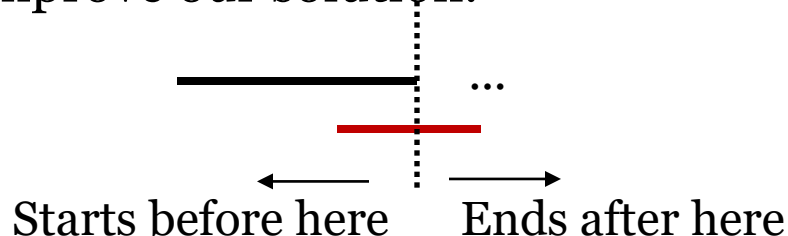
- **Counterexample:**



- Earliest workshop end first

- Guaranteed to be correct

- **Proof idea:** when we select the first workshop to add, every workshop we eliminate must overlap everything else we eliminate, and possibly more workshops. None of these could improve our solution.



# Greedy algorithm exercise

- Prove that the greedy algorithms below do not find the largest **sum**  $\leq t$  in a given array

**Input:** *data*: array of positive integers  
**Input:** *n*: size of *data*  
**Input:** *t*: target value  
**Output:** subset of *data* with largest sum  $\leq t$

```
1 Algorithm: GreedyKnapsack
2 soln = {}
3 repeat
4   | Let m be the smallest element we haven't added yet
5   | Add m to soln if it doesn't push the sum over t
6 until the next smallest element is too big
7 return soln
```

**Input:** *data*: array of positive integers  
**Input:** *n*: size of *data*  
**Input:** *t*: target value  
**Output:** subset of *data* with largest sum  $\leq t$

```
1 Algorithm: GreedyKnapsack
2 soln = {}
3 repeat
4   | Let m be the largest element we haven't tried adding yet
5   | Add m to soln if it doesn't push the sum over t
6 until we can't add anything else
7 return soln
```

2 3 4 5

$t = 10$

greedy: 2 3 4

better: 2 3 5

5 4 3 2

gs: 5 4



# Greedy algorithm exercise

- Prove that the greedy algorithms below do not find the largest **sum**  $\leq t$  in a given array

**Input:** *data*: array of positive integers  
**Input:** *n*: size of *data*  
**Input:** *t*: target value  
**Output:** subset of *data* with largest sum  $\leq t$

```
1 Algorithm: GreedyKnapsack
2 soln = {}
3 repeat
4   | Let m be the smallest element we haven't added yet
5   | Add m to soln if it doesn't push the sum over t
6 until the next smallest element is too big
7 return soln
```

**Counterexample:**  
*data* = {1, 10}, *t* = 10

Optimal sol'n: {10}  
Greedy answer: {1}

**Input:** *data*: array of positive integers  
**Input:** *n*: size of *data*  
**Input:** *t*: target value  
**Output:** subset of *data* with largest sum  $\leq t$

```
1 Algorithm: GreedyKnapsack
2 soln = {}
3 repeat
4   | Let m be the largest element we haven't tried adding yet
5   | Add m to soln if it doesn't push the sum over t
6 until we can't add anything else
7 return soln
```

**Counterexample:**  
*data* = {4, 3, 2}, *t* = 5

Optimal sol'n: {3, 2}  
Greedy answer: {4}

# Coming up

---

- Iterative dynamic programming
- Greedy algorithms
- Graph representations
- Graph traversals
  
- **Recommended readings:** Sections 12.2, 12.5-12.5.2
- *Practice problems:* R-12.8, C-12.2 and C-12.4 (read section 12.4), C-12.6, A-12.1