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

Lecture 9

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

Algorithm: MemoBinom
if $binom[n, k] > 0$ **then**
    return $binom[n, k]$
else if $k = 0$ or $k = n$ **then**
    $binom[n, k] = 1$
else
    $binom[n, k] = \text{MemoBinom}(n-1, k-1) + \text{MemoBinom}(n-1, k)$
end
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?

$$LCS(garbage, bogarted)$$

$$LCS(arbage, bogarted) \qquad LCS(garbage, ogarted)$$

$$LCS(rbage, bogarted) \quad LCS(arbage, ogarted) \quad LCS(arbage, ogarted) \quad LCS(garbage, garted)$$

Formalizing the problem

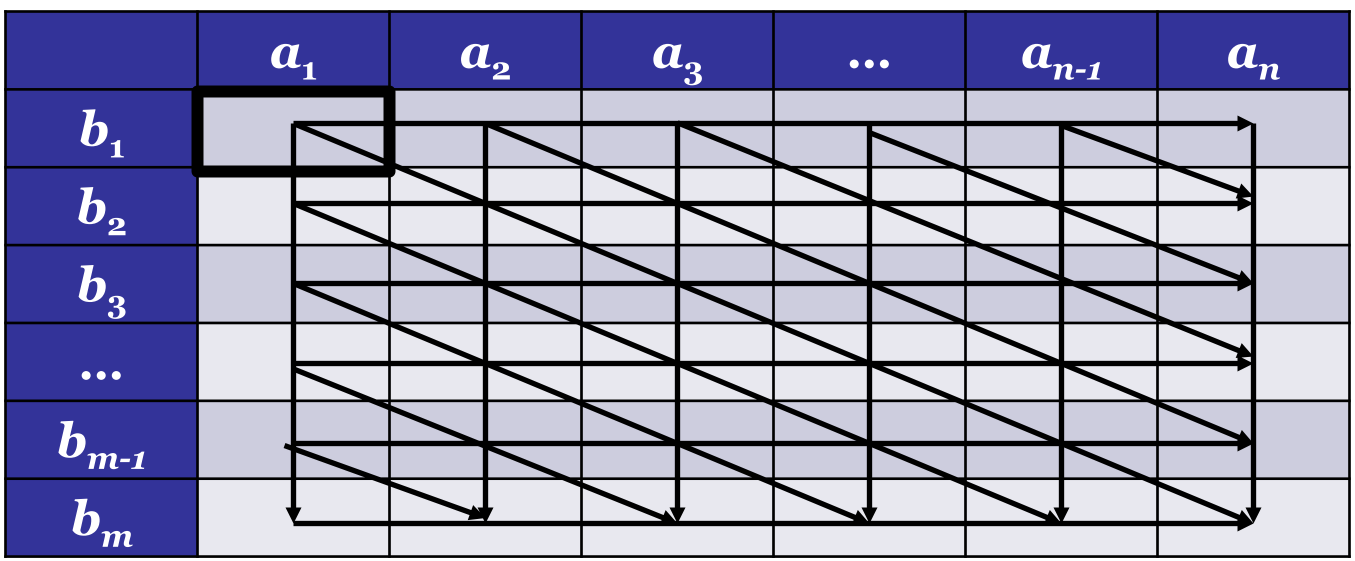
$$LCS(a,b) = \begin{cases} a_1 + LCS(a',b'), & \text{if } a_1 = b_1 \\ \max \begin{cases} LCS(a',b) \\ LCS(a,b') \end{cases}, & \text{otherwise} \end{cases}$$

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

Dependency structure

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

Base cases?



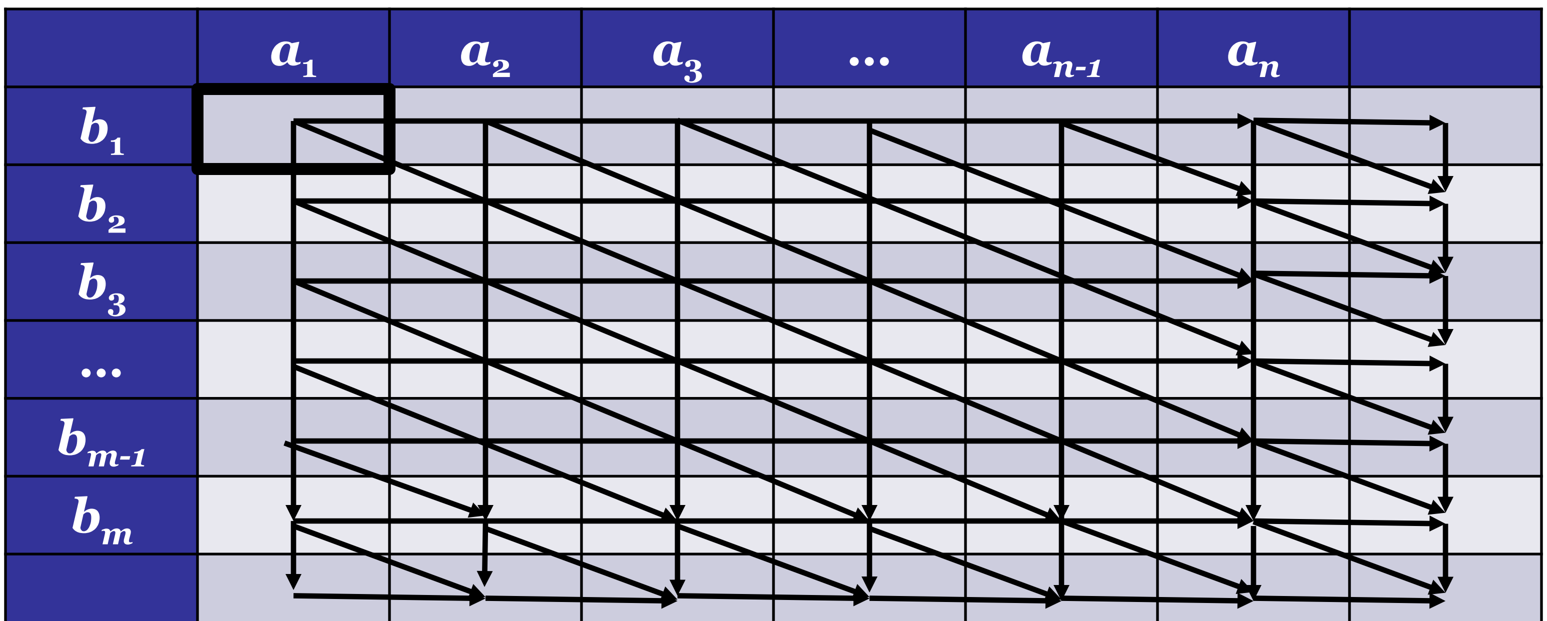
Dependency structure

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

Base cases

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

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



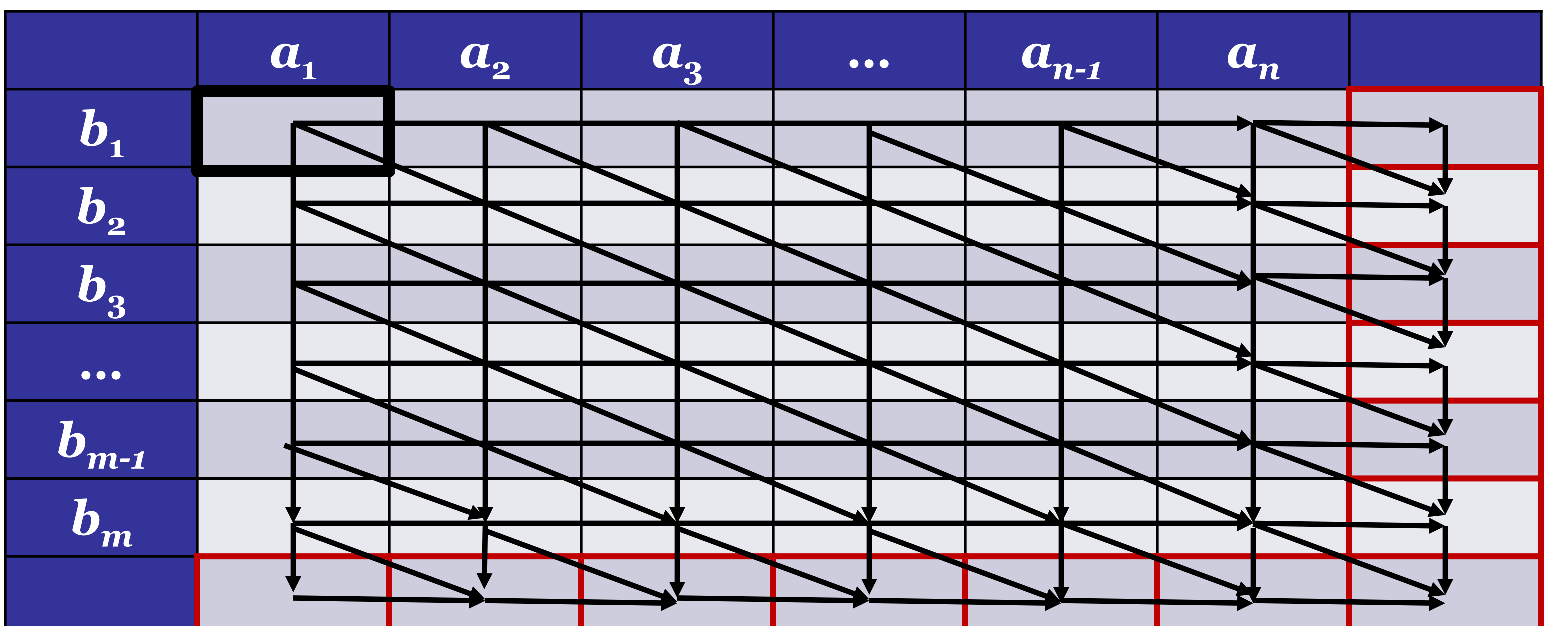
Dependency structure

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

<u>Base cases</u>

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

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



Tracing example

- LCS(air, care):

	a	*i*	*r*	
c				
a				
r				
e				

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c				ϵ
a				
r				
e				

Tracing example

- LCS(air, care):

	a	***i***	***r***	
c				ϵ
a				ϵ
r				
e				

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c				ϵ
a				ϵ
r				
e				ϵ

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c				ϵ
a				ϵ
r			*r*	
e				ϵ

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c				ϵ
a			*r*	ϵ
r			*r*	
e				ϵ

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c			*r*	ϵ
a			*r*	ϵ
r			*r*	
e				ϵ

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c			r	ϵ
a			r	ϵ
r			r	
e				ϵ
			ϵ	

Tracing example

- LCS(air, care):

	a	***i***	***r***	
c			***r***	ϵ
a			***r***	ϵ
r			***r***	
e			ϵ	ϵ
			ϵ	

Tracing example

- LCS(air, care):

	a	***i***	***r***	
c			r	ϵ
a			r	ϵ
r			r	
e			ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c			*r*	ϵ
a			*r*	ϵ
r			*r*	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c			r	ϵ
a			r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c			r	ϵ
a		r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c		r	r	ϵ
a		r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	***i***	***r***	
c		r	r	ϵ
a	ar	r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	*i*	*r*	
c	ar	r	r	ϵ
a	ar	r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Tracing example

- LCS(air, care):

	a	i	r	
c	ar	r	r	ϵ
a	ar	r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

LCS: memoization

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

<u>Base cases</u>

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

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

- Implement recurrence directly
- Rely on map for previous subproblems
- Write outer function to initialize map
  - Need to choose sentinel value carefully

```
Algorithm: MemoLCS(a, i, b, j)
if lcs[i, j] ≠ NIL then
|   return lcs[i, j]
else if i = n + 1 or j = m + 1 then
|   lcs[i, j] = ε
else if a[i] = b[j] then
|   lcs[i, j] = a[i] + MemoLCS(a, i + 1, b, j + 1)
else
|   o1 = MemoLCS(a, i + 1, b, j)
|   o2 = MemoLCS(a, i, b, j + 1)
|   lcs[i, j] = max(o1, o2)
end
return lcs[i, j]
```

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

An alternative approach

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

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

	a	***i***	***r***	
c	*ar*	*r*	*r*	ε
a	*ar*	*r*	*r*	ε
r		*r*	*r*	
e		ε	ε	ε
		ε	ε	ε

→

	a	***i***	***r***	
c	2	1	1	0
a	2	1	1	0
r		1	1	
e		0	0	0
		0	0	0

+

	a	***i***	***r***	
c	↓	→	↓	
a	↘	→	↓	
r		→	↘	
e				

- **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 Θ(*n*+*m*) processing step (doesn't change complexity)
- Reduces space complexity

Analyzing memoization

- LCS(air, care):

	a	**i**	**r**	
c	ar	r	r	ϵ
a	ar	r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

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

<u>Two questions</u>

- 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(air, care):

	a	***i***	***r***	
c	ar	r	r	ϵ
a	ar	r	r	ϵ
r		r	r	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

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

<u>Worst-case analysis</u>

- *O(nm)* cells
- *Θ(1)* time per cell
- Total time: *O(nm)*

Iterative dynamic programming

Main idea

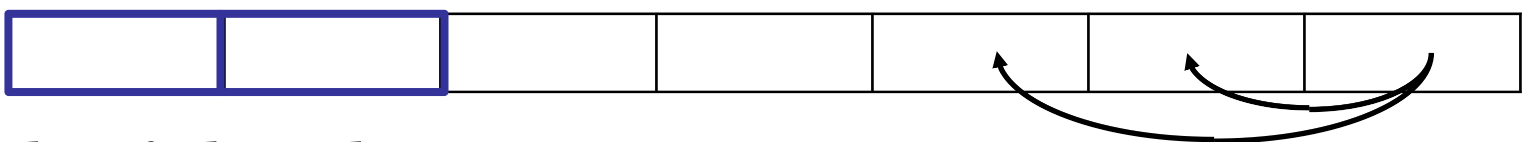
- Recursion not needed if dependent cells already filled in

<u>Process</u>

- 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



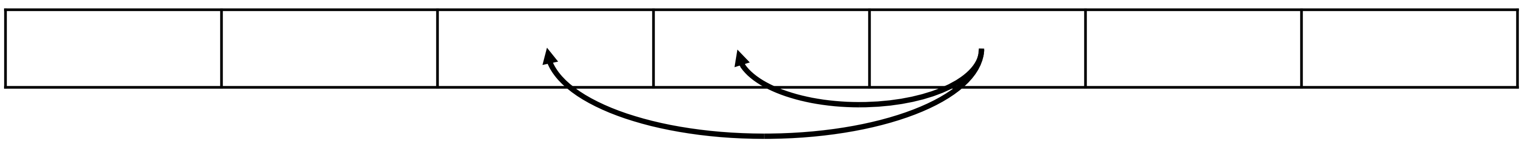
- Identify dependencies
  - Each cells needs the two cells to the left
  - Move in *opposite* direction
- Base cases
  - Must start here
- Need to iterate left-to-right
  - When calculating *F*(*n*), *F*(*n*-1) and *F*(*n*-2) already solved

Input: n: index of Fibonacci number to compute
Output: F_n
Algorithm: IterativeFib

$fib = \text{Array}(n)$
for $i = 1$ to n **do**
    if $i = 1$ or 2 **then**
        $fib[i] = 1$
    else
        $fib[i] = fib[i-1] + fib[i-2]$
    end
end
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

Algorithm: BestFib

if $n = 1$ or 2 **then**
    return 1
end
$prev = 1$
$twoprev = 1$
for $i = 3$ to n **do**
    $curr = prev + twoprev$
    $twoprev = prev$
    $prev = curr$
end
return $curr$

Iterative DP example

- Longest common subsequence

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

	a	*i*	*r*	
c				
a				
r				
e				

- What order should we iterate through the array?

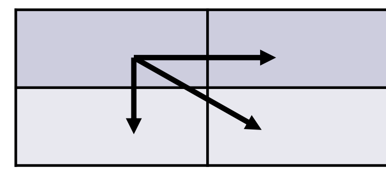
Iterative DP example

- Longest common subsequence

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

	a	*i*	*r*	
c	*ar*	*r*	*r*	ϵ
a	*ar*	*r*	*r*	ϵ
r		*r*	*r*	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Dependency structure:



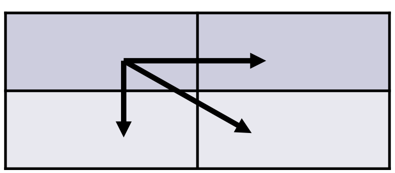
- What order should we iterate through the array?
  - Base cases are on bottom and right side of array
  - Cells to the right, bottom, and diagonal must be filled
  - Bottom-to-top, right-to-left works well
  - Other orders are possible

Space complexity

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

	a	*i*	*r*	
c	*ar*	*r*	*r*	ε
a	*ar*	*r*	*r*	ε
r		*r*	*r*	
e		ε	ε	ε
		ε	ε	

Dependency structure:

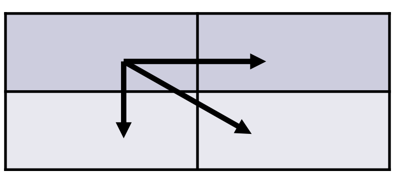


Space complexity

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

	a	*i*	*r*	
c	*ar*	*r*	*r*	ϵ
a	*ar*	*r*	*r*	ϵ
r		*r*	*r*	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

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
Algorithm: IterativeLCS
prev = Array(n + 1)
curr = Array(n + 1)
Initialize prev with ϵ
for i = n down to 1 do
    curr[m + 1] = ϵ
    for j = m down to 1 do
        if a[i] = b[j] then
            curr[j] = b[j] + prev[j + 1]
        else
            curr[j] = max{curr[j + 1], prev[j]}
        end
    end
    prev = curr
end
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

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

	a	**i**	**r**	
c	*ar*	*r*	*r*	ϵ
a	*ar*	*r*	*r*	ϵ
r		*r*	*r*	
e		ϵ	ϵ	ϵ
		ϵ	ϵ	

Iterative DP exercise

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

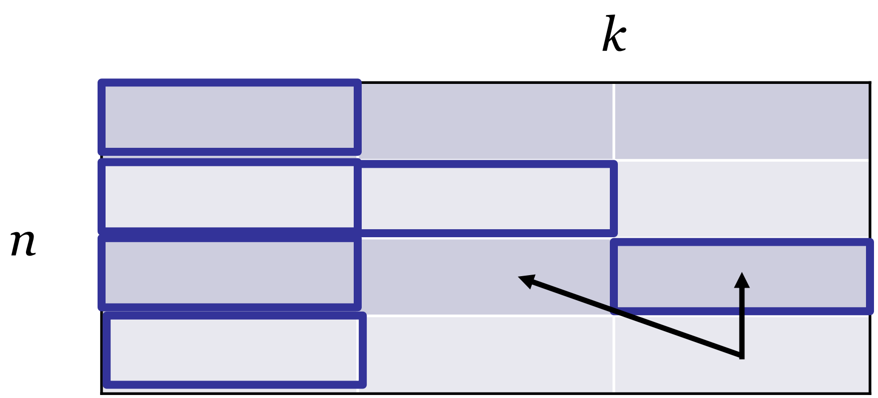
k

n

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

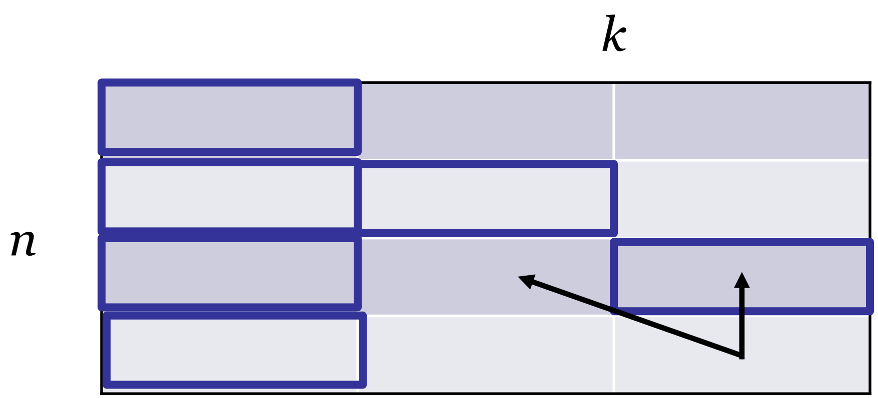
- Binomial coefficients ("*n* choose *k*")
- **Recurrence:** $\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$



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

Iterative DP exercise

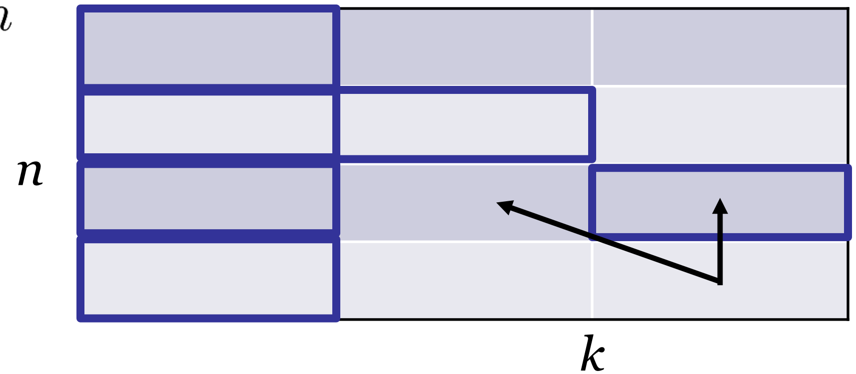
- Binomial coefficients ("n choose k")
- **Recurrence:** $\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$



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

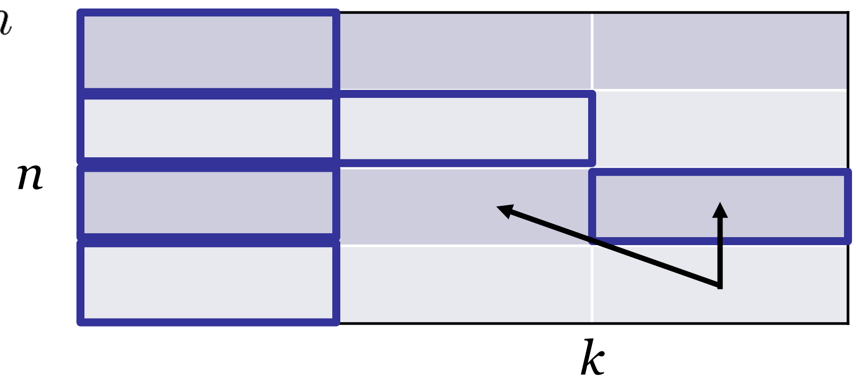
$$\binom{n}{k} = \begin{cases} 1, \text{ if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$



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

```
Algorithm: NChooseK
C = Array(n, k)
for i = 0 to n do
    for j = 0 to min{i, k} do
        if j = 0 or j = i then
            C[i, j] = 1
        else
            C[i, j] = C[i − 1, j − 1] + C[i − 1, j]
        end
    end
end
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 $\leq t$

```
Algorithm: GreedyKnapsack
heap = Heapify(data)
soln = {}
sum = 0
next = heap.DeleteMin()
while sum + next ≤ t do
    Add next to soln
    sum = sum + next
    next = heap.DeleteMin()
end
return soln
```

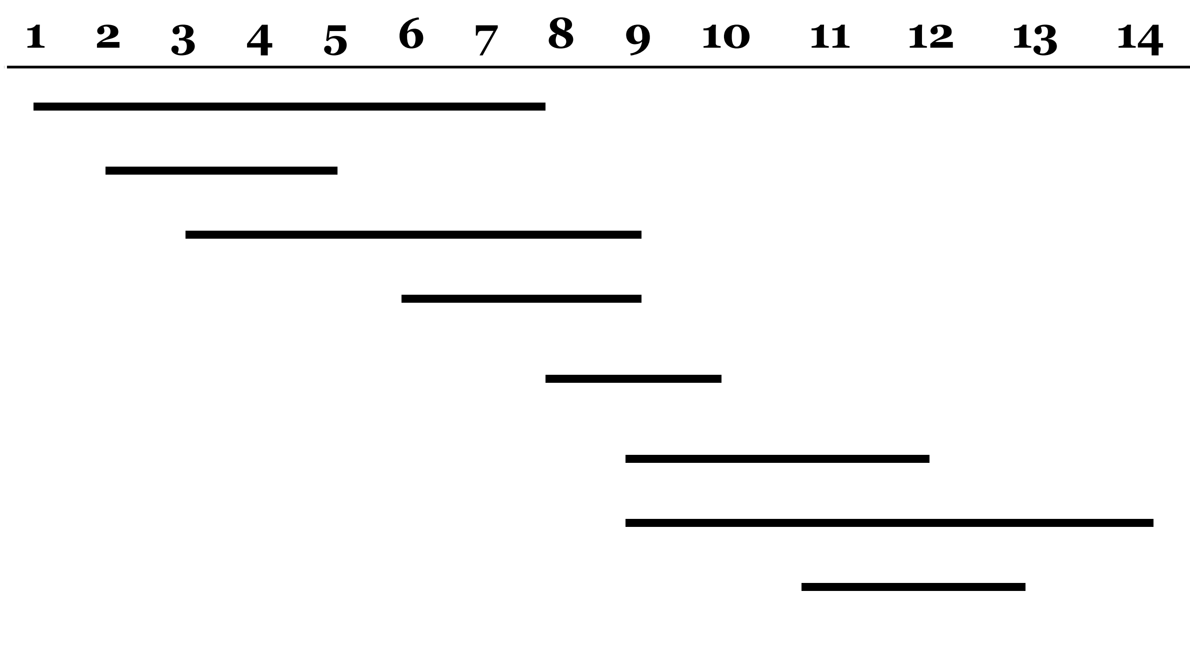
- Alternatively: sort first, then iterate
  - Same answer and time complexity
- Is the answer correct?

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.
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

Algorithm: GreedyWorkshop

$best = 0$
$subset = \{\}$
Sort *start* and *duration* by ________________
while *start* and *duration* not empty **do**
    Add $(start[1], duration[1])$ to *subset*
    Remove all workshops from *start* and *duration* that overlap this workshop
    $best = best + 1$
end
return *best*

GreedySearch variants

- Earliest workshop first

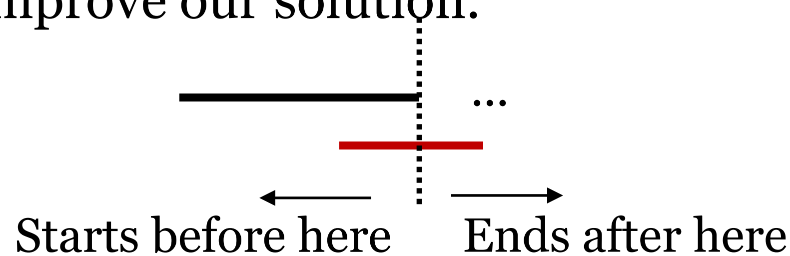
- Shortest workshop first

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 $\leq t$
Algorithm: GreedyKnapsack
$soln = \{\}$
repeat
    Let m be the smallest element we haven't added yet
    Add m to *soln* if it doesn't push the sum over t
until the next smallest element is too big
return *soln*

Input: *data*: array of positive integers
Input: n: size of *data*
Input: t: target value
Output: subset of *data* with largest sum $\leq t$
Algorithm: GreedyKnapsack
$soln = \{\}$
repeat
    Let m be the largest element we haven't tried adding yet
    Add m to *soln* if it doesn't push the sum over t
until we can't add anything else
return *soln*

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 $\leq t$
Algorithm: GreedyKnapsack
$soln = \{\}$
repeat
  Let m be the smallest element we haven't added yet
  Add m to *soln* if it doesn't push the sum over t
until the next smallest element is too big
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 $\leq t$
Algorithm: GreedyKnapsack
$soln = \{\}$
repeat
  Let m be the largest element we haven't tried adding yet
  Add m to *soln* if it doesn't push the sum over t
until we can't add anything else
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