COMP 285
Analysis of Algorithms

Welcome to COMP 285

Lecture 21: Dynamic Programming II

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW7!

Due 11/15 @ 11:59PM ET

HW7!

Walkthrough Pt. 1 (available) + Pt.2 (EoD)

HW8 FYI

Released next week, due last week of class

Quizzes

Quiz 8 (today), Quiz 9 (11/15), Quiz 10 (11/22)

Final Exam

Tuesday 12/06 2:00pm-4:00pm

Quiz!

www.comp285-fall22.ml or Blackboard

Recall where we ended last lecture…

What is Dynamic Programming?

- It is an algorithm design paradigm
  - like divide-and-conquer, greediness, etc. are algorithm design paradigms.
- Usually, it is for solving **optimization problems**
  - E.g., ***shortest***, ***best***, ***maximum/minimum*** option
  - (Fibonacci numbers aren't an optimization problem, but they are a good example of dynamic programming anyway…)
- Similar to greedy, there are two properties to look for…

Properties of Dynamic Programming

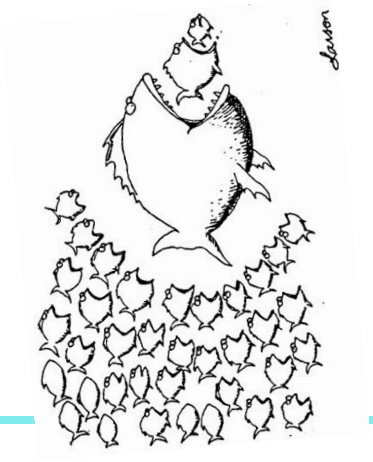
1. **Optimal substructure**
   - Big problems break up into sub-problems
     - **Fibonacci numbers: F(i) for i <= n**
   - The solution to a subproblem can be expressed in terms of solutions to smaller subproblems.
     - **Fibonacci numbers: F(i) = F(i-1) + F(i-2)**

2. **Overlapping subproblems**
   - Subproblems overlap/can be reused
     - **Fibonacci numbers:**
       1. **Both F[i+1] and F[i+2] directly use F[i]**
       2. **Lots of different F[i+x] indirectly use F[i].**
   - This means that we can save time by solving a sub-problem just once and storing the answer.
     - **To be continued…**

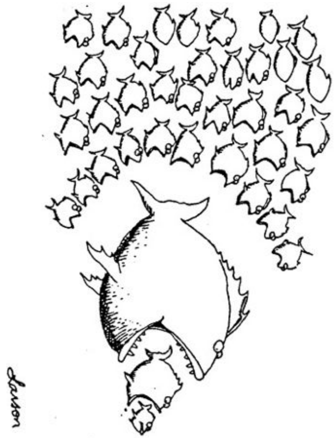
Bottom up approach (what we just saw!)

- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]

```
def fasterFibonacci(n):
    • F = [0, 1, None, None, …, None]:
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```



Top down approach



- Think of it like a recursive algorithm.
- To solve the big problem:
    - Recurse to solve smaller problems
        - Those recurse to solve smaller problems
            - etc…

- define a global list F = [0,1,None, None, …, None]
- **def** Fibonacci(n):
    - **if** F[n] != None:
        - **return** F[n]
    - **else:**
        - F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    - **return** F[n]

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?
   - F[i] = the i-th Fibonacci number
2. Relate subproblems
   - How do subproblems build upon or use other subproblems?
   - F[i] = F[i-1] + F[i-2]. Base case: F[1] = F[2] = 1
3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n
4. Solve original problem
   - Return F[n]

Steps 1 and 2 are often the trickiest / take the most practice.

Example: Longest Increasing Subsequence

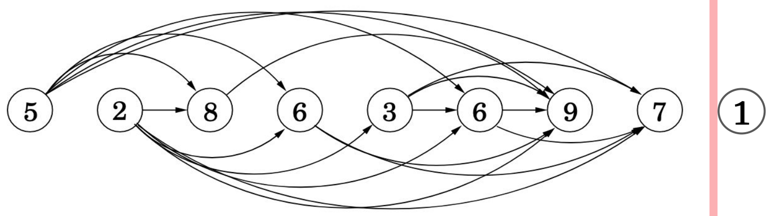
Input: vector of integers vec of size N > 0

Output: length of the longest increasing subsequence within the vector

Note: with a subsequence, we pick numbers within the vector in order (we're allowed skips)

Example: [5, 2, 8, 6, 3, 6, 9, 7, 1] → 4

Example: [6, 1, 8, 2, 3, 1, 9] → 3



Example: Longest Increasing Subsequence

```
algorithm longestIncreasingSubsequence
  Input: vector of integers vec of size N > 0
  Output: length of the longest increasing subsequence of vec

  L = array to store subproblem solutions
  for i = 0, 1, 2, 3, ... N-1:
    maxLength = 1
    for j = 0, 1, 2, ... i:
      if vec[j] < vec[i]
        maxLength = max(maxLength, L[j] + 1)
    L[i] = maxLength

  // find max
  answer = 1
  for each value in L:
     answer = max(value, answer)
  return answer
```

1. L[i] = longest subsequence ending at index i.
2. L[i] = max(L[j] for j in 0...i if vec[i] > vec[j]) + 1
3. Solve i = 0, 1, 2, … n
4. Return max value in table

Big Questions!

- More examples of dynamic programming!

Big Questions!

- More examples of dynamic programming!

Example #1: Longest Common Subsequence

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

Longest Common Subsequence

- Subsequence:
  - BDFH is a **subsequence** of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
  - BDFH is a **common subsequence** of ABCDEFGH and of ABDFGHI
- A **longest common subsequence**…
  - …is a common subsequence that is longest.
  - The **longest common subsequence** of ABCDEFGH and ABDFGHI is ABDFGH.

We sometimes want to find these

- Applications in bioinformatics

- The unix command `diff`
- Merging in version control
  - svn, git, etc...

Quick Overview of Approach

LCS(X = “ace”, Y = “abcde”)   1+LCS(X = “ac”, Y = “abcd”)

LCS(X = “aca”, Y = “abcde”)   max(LCS(X=“ac”, Y=“abcde”),
LCS(X=“aca”, Y = “abcd”))

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?

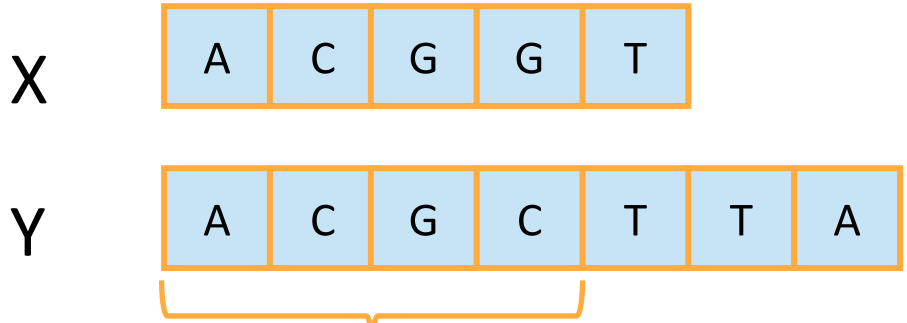
2. Relate subproblems
   - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Step 1: Define recursive subproblem

Prefixes:



Notation: denote this prefix **ACGC** by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let $C[i,j]$ = length_of_LCS(X_i, Y_j)

Examples: C[2,3] = 2
C[4,4] = 3

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we’re solving look like?

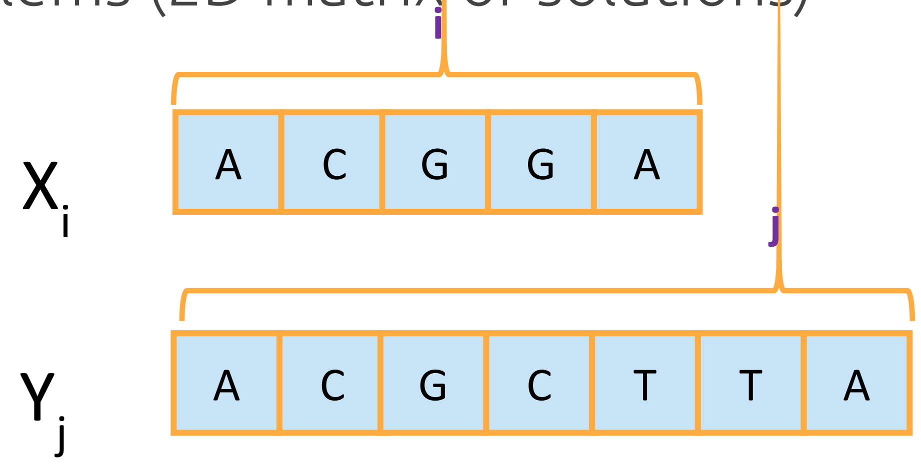
2. Relate subproblems
   - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Step 2: Relate subproblems

- Write C[i,j] in terms of the solutions to smaller sub-problems (2D matrix of solutions)



$C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Two cases

Case 1: X[i] = Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

X_i: A C G G A

Y_j: A C G C T T A

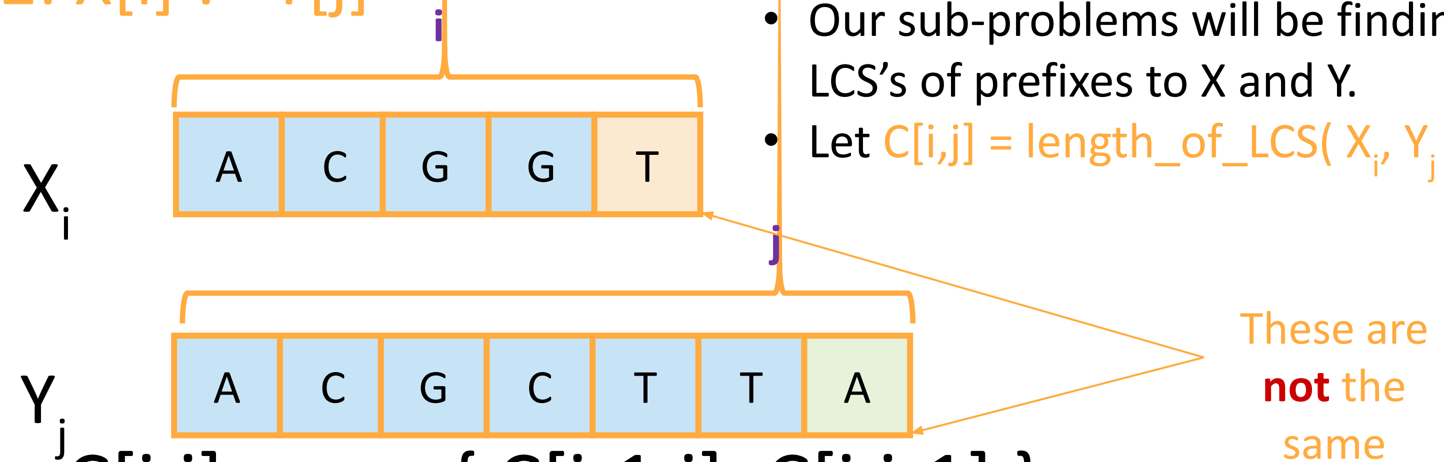
These are the same

- Then C[i,j] = 1 + C[i-1,j-1].
  - because LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1}) followed by A

Two cases

Case 2: X[i] != Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)



- Then C[i,j] = max{ C[i-1,j], C[i,j-1] }.
  - either LCS(X_i,Y_j) = LCS(X_{i-1},Y_j) and T is not involved,
  - or LCS(X_i,Y_j) = LCS(X_i,Y_{j-1}) and A is not involved,

Recursive Formulation of the Optimal Solution

X_0 | |

Y_j | A | C | G | C | T | T | A |

Case 0

- $$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Case 1

X_i | A | C | G | G | A |

Y_j | A | C | G | C | T | T | A |

Case 2

X_i | A | C | G | G | T |

Y_j | A | C | G | C | T | T | A |

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
    - What does an instance of the problem we're solving look like?

2. Relate subproblems
    - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
    - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Longest Common Subsequence Dynamic Programming

- LCS(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 0,…,m, j=0,…n.
  - For i = 1,…,m
    - For j = 1,…,n:
      - If X[i] = Y[j]:
        - C[i,j] = C[i-1,j-1] + 1
      - Else:
        - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Longest Common Subsequence Dynamic Programming

- LCS(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 0,…,m, j=0,…,n.
  - For i = 1,…,m
    - For j = 1,…,n:
      - If X[i] = Y[j]:
        - C[i,j] = C[i-1,j-1] + 1
      - Else:
        - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

Running time: O(nm)

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X

A	C	G	G	A

Y

A	C	T	G

		Y: A	C	T	G
	0	0	0	0	0
X: A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

So the LCM of X and Y has length 3.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1], C[i-1, j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X: A C G G A

Y: A C T G

		Y: A	C	T	G
	0	0	0	0	0
X: A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{C[i,j-1], C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X

A	C	G	G	A

Y

A	C	T	G

Y

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X

- Once we've filled this in, we can work backwards.

That 3 must have come from the 3 above it.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X: | A | C | G | G | A |

Y: | A | C | T | G |

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

This 3 came from that 2 – we found a match!

G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i,j-1], C[i-1,j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X: A C G G A

Y: A C T G

		Y: A	C	T	G
X	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

That 2 may as well have come from this other 2.

G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i,j-1], C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X	A	C	G	G	A
Y	A	C	T	G	

Y

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
X G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X: A C G G A

Y: A C T G

		Y: A	C	T	G
	0	0	0	0	0
X: A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

C G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i,j-1], C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X
A	C	G	G	A

Y
A	C	T	G

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

A	C	G

This is the LCS!

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1]+1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i,j-1], C[i-1,j]\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

LCS Complexity

- Good exercise to write out pseudocode for what we just saw!
  - Or you can find it in lecture notes.
- Takes time $O(mn)$ to fill the table
- Takes time $O(n + m)$ on top of that to recover the LCS
  - We walk up and left in an n-by-m array
  - We can only do that for $n + m$ steps.
- Altogether, we can find LCS(X,Y) in time $O(mn)$.

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

What have we learned?

- We can find LCS(X,Y) in time O(nm), what about space?
  - if |Y|=n, |X|=m
  - O(nm) space as well to store the solutions C[i, j]

- We went through the steps of coming up with a dynamic programming algorithm.
  - We kept a 2-dimensional table, breaking down the problem by decrementing the length of X and Y.

Big Questions!

- More examples of dynamic programming!

Example 2: Knapsack Problem

- We have n items with weights and values:

Item:					
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- And we have a knapsack:
  - it can only carry so much weight

Capacity: 10

Capacity: 10

Item:	🐢	💡	🍉	🌮	🚒
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all items.
  - What's the most valuable way to fill the knapsack?

🌮 🌮 💡 💡

Total weight: 10
Total value: 42

- 0/1 Knapsack:
  - Suppose I have only one copy of each item.
  - What's the most valuable way to fill the knapsack?

💡 🍉 🌮

Total weight: 9
Total value: 35

Some notation

Item:					
Weight:	w_1	w_2	w_3	...	w_n
Value:	v_1	v_2	v_3		v_n

Capacity: W

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?

2. Relate subproblems
   - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Define recursive subproblem

- Sub-problems:
    - Unbounded Knapsack with a smaller knapsack.
    - K[x] = value you can fit in a knapsack of capacity x

First solve the problem for small backpack

Then larger …

Then larger …

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?

2. Relate subproblems
   - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Relate subproblems

item i

- Suppose this is an optimal solution for capacity x:

Say that the optimal solution contains at least one copy of item i.

Weight w_i

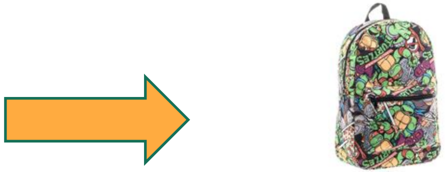
Value v_i

Capacity x

Value V

- Then this is the optimal solution for capacity x - w_i:

Do we agree?



Capacity x – w_i

Value V - v_i

Relate subproblems

- Let K[x] be the optimal value for capacity x.

$K[x] = \max_i \{$  $+$  $\}$

The maximum is over all i so that $w_i \leq x$.

Optimal way to fill the smaller knapsack

The value of item i.

$$K[x] = \max_i \{ K[x - w_i] + v_i \}$$

- (And K[x] = 0 if the maximum is empty).
  - That is, if there are no i so that $w_i \leq x$

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
   - What does an instance of the problem we're solving look like?

2. Relate subproblems
   - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
   - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Let's write a bottom-up approach

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{\, K[x], K[x \ - w_i] + v_i \,\}$
  - **return** K[W]

K[x] = $\max_i$ {  +  }

= $\max_i$ { $K[x - w_i] + v_i$ }

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
    - What does an instance of the problem we're solving look like?

2. Relate subproblems
    - How do subproblems build upon or use other subproblems?

3. Top-down with memoization **or** build table bottom-up with ordering
    - e.g. Build table bottom-up by starting at i=1 then solving 2, 3, 4, … n

4. Solve original problem

Let's write a bottom-up approach

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
  - **return** K[W]

$K[x] = \max_i \{$  $+$  $\}$

$= \max_i \{ K[x - w_i] + v_i \}$

Let's write a bottom-up approach

- UnboundedKnapsack(W, n, weights, values):
    - K[0] = 0
    - ITEMS[0] = Ø
    - **for** x = 1, ..., W:
        - K[x] = 0
        - **for** i = 1, ..., n:
            - **if** $w_i \leq x$:
                - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
                - If K[x] was updated:
                    - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
    - **return** ITEMS[W]

$K[x] = \max_i \{$  $+$  $\}$

$= \max_i \{ K[x - w_i] + v_i \}$

Example

	0	1	2	3	4
K	0				
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:				
Weight:	1	2	3	
Value:	1	4	6	Capacity: 4

Example

ITEMS[1] = ITEMS[0] + 

	0	1	2	3	4
K	0	1			
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[2] = ITEMS[1] + 

	0	1	2	3	4
K	0	1	2		
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[2] = ITEMS[0] + 

	0	1	2	3	4
K	0	1	4		
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[3] = ITEMS[2] + 🐢

	0	1	2	3	4
K	0	1	4	5	
ITEMS		🐢	💡	💡 🐢	

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, ..., W:
    - K[x] = 0
    - **for** i = 1, ..., n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[3] = ITEMS[0] + 

	0	1	2	3	4
K	0	1	4	6	
ITEMS					

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:			
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[4] = ITEMS[3] + 🐢

	0	1	2	3	4
K	0	1	4	6	7
ITEMS		🐢	💡	🍉	🍉 🐢

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{ K[x], K[x - w_i] + v_i \}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

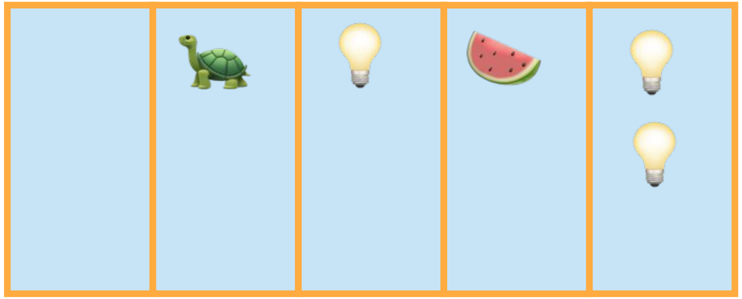
Capacity: 4

Example

ITEMS[4] = ITEMS[2] + 

	0	1	2	3	4
K	0	1	4	6	8

ITEMS



- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = Ø
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{\, K[x], K[x - w_i] + v_i \,\}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	(turtle)	(light bulb)	(watermelon)
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Example

ITEMS[4] = ITEMS[2] + 💡

	0	1	2	3	4
K	0	1	4	6	8
ITEMS		🐢	💡	🍉	💡 💡

- UnboundedKnapsack(W, n, weights, values):
  - K[0] = 0
  - ITEMS[0] = ∅
  - **for** x = 1, …, W:
    - K[x] = 0
    - **for** i = 1, …, n:
      - **if** $w_i \leq x$:
        - $K[x] = \max\{\, K[x], K[x - w_i] + v_i \,\}$
        - If K[x] was updated:
          - ITEMS[x] = ITEMS[x – w_i] ∪ { item i }
  - **return** ITEMS[W]

Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 4

Final solution is K[4]
Max value of 8 using two 💡

Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

npr KQED PUBLIC MEDIA news arts & life music programs shop

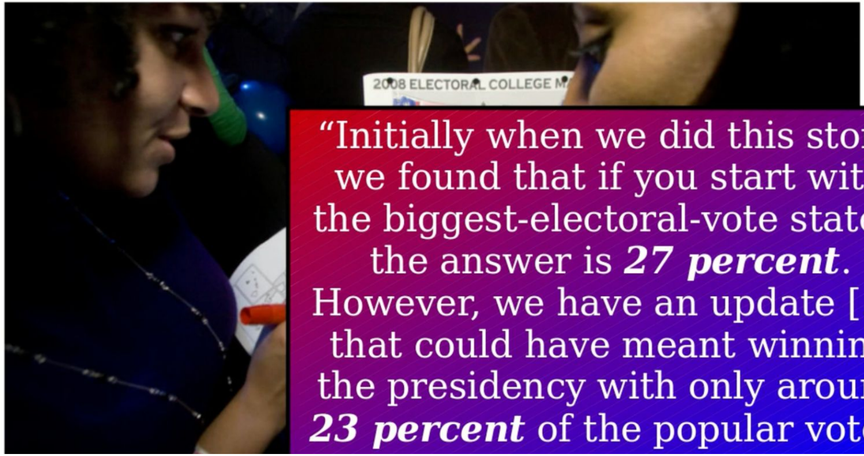
POLITICS

21.1%

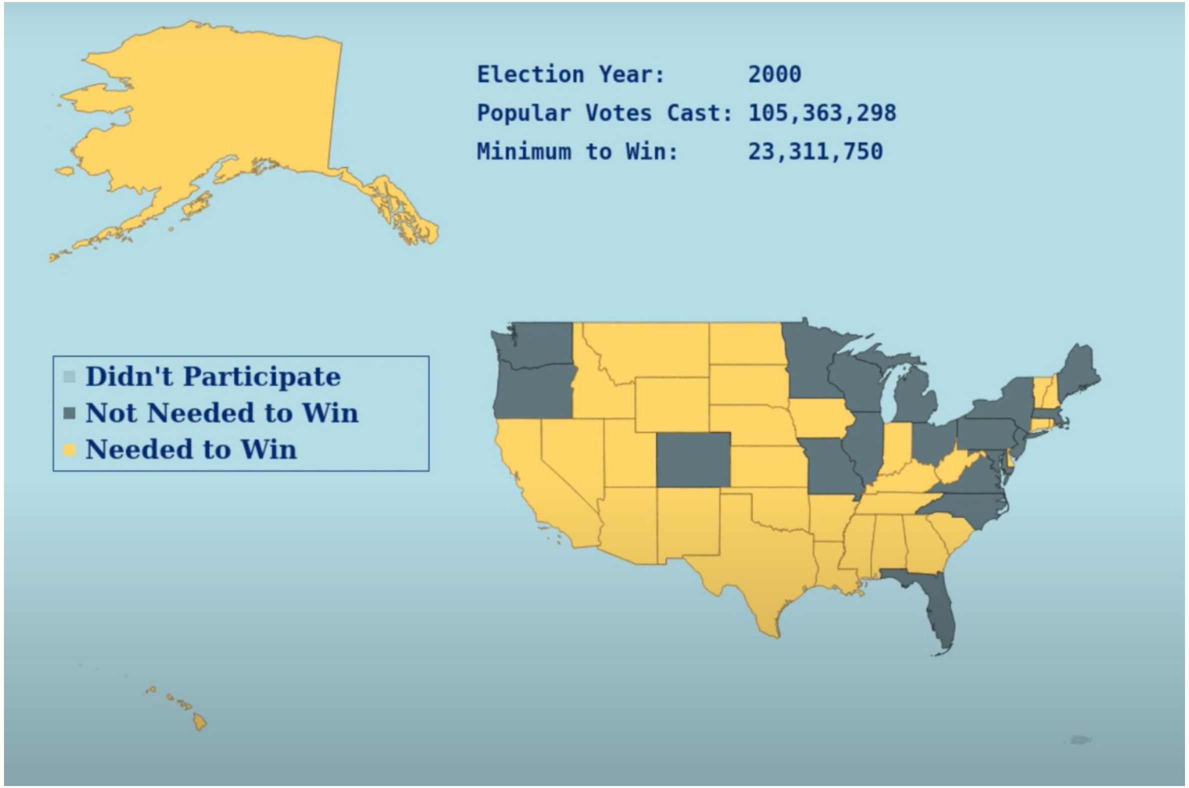
How To Win The Presidency With ~~23~~ Percent Of The Popular Vote

November 2, 2016 · 6:00 AM ET

DANIELLE KURTZLEBEN



"Initially when we did this story, we found that if you start with the biggest-electoral-vote states, the answer is ***27 percent***. However, we have an update […] that could have meant winning the presidency with only around ***23 percent*** of the popular vote."


Election Year: 2000
Popular Votes Cast: 105,363,298
Minimum to Win: 23,311,750
Didn't Participate
Not Needed to Win
Needed to Win

What have we learned?

- We can solve unbounded knapsack in time O(nW).
  - If there are n items and our knapsack has capacity W.
- We again went through the steps to formulate our solution:
  - We kept a one-dimensional table, creating smaller problems by making the knapsack smaller.

COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 21: Dynamic Programming II

Lecturer: Chris Lucas (cflucas@ncat.edu)