COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 20: Dynamic Programming I

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW6 Due!

Tonight @ 11:59PM ET

HW6 Due!

Late deadline 11/08 @ 11:59PM ET

HW7 Released by EoD!

Due 11/15 @ 11:59PM ET

Netflix Opportunity!

- Fill out this form (link)
- Check “Meta Classroom”

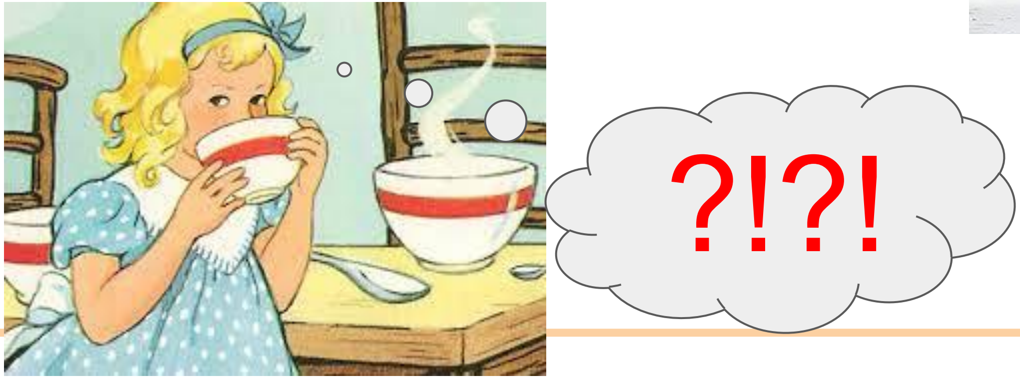
Mock Interview with Meta!

- +1% Extra credit opportunity! (link)
- Nov. 16-18 (limited availability)

Recall where we ended last lecture…

The Greedy Algorithm Process

1. Make choices one at a time
2. Never look back
3. Hope for the best



Activity Selection

COMP 361 Class

Math 121 Class

COMP 285 Office Hours

Sleep

You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

nar

Progr

team

35 Class

Underwater basket weaving class

COMP 160 Class

COMP 285 study group

Swimming lessons

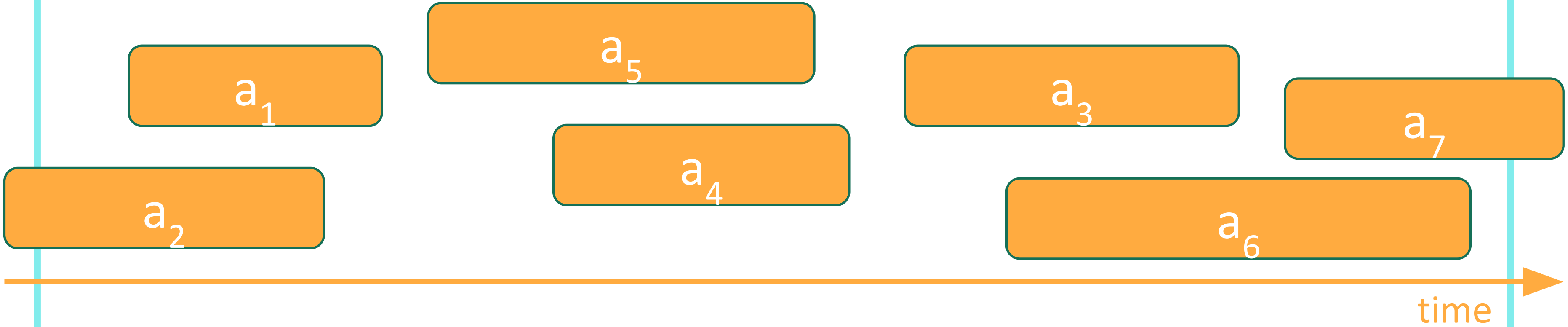
Theory Lunch

Combinatorics Seminar

Social activity

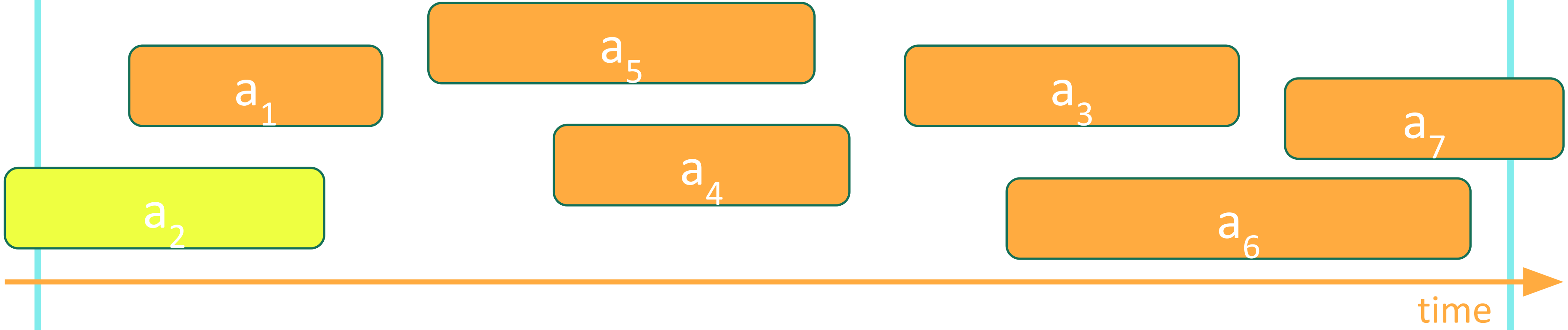
time

Greedy Algorithm



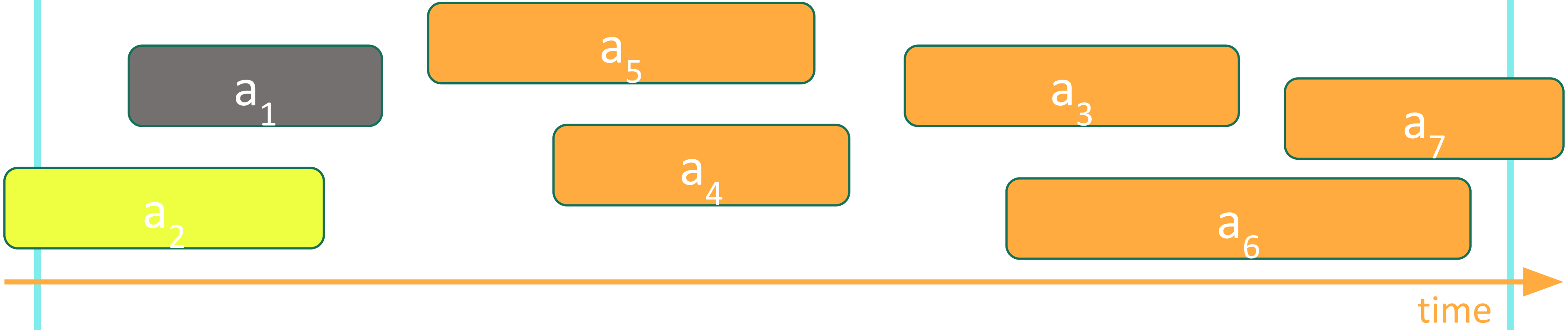
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



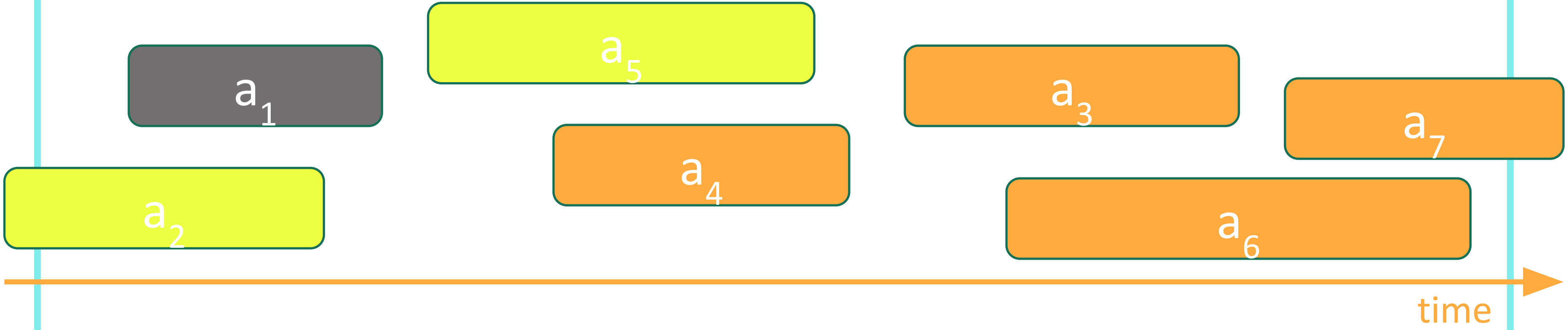
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



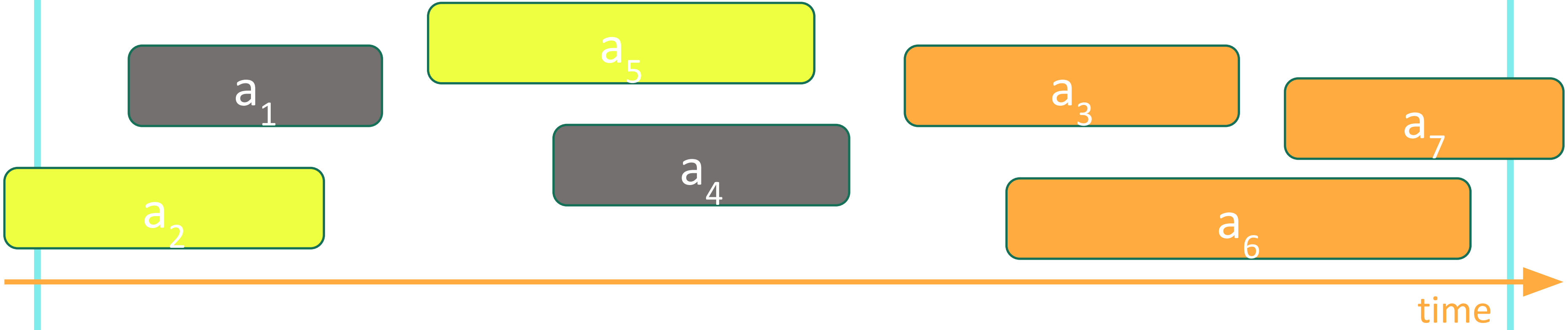
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



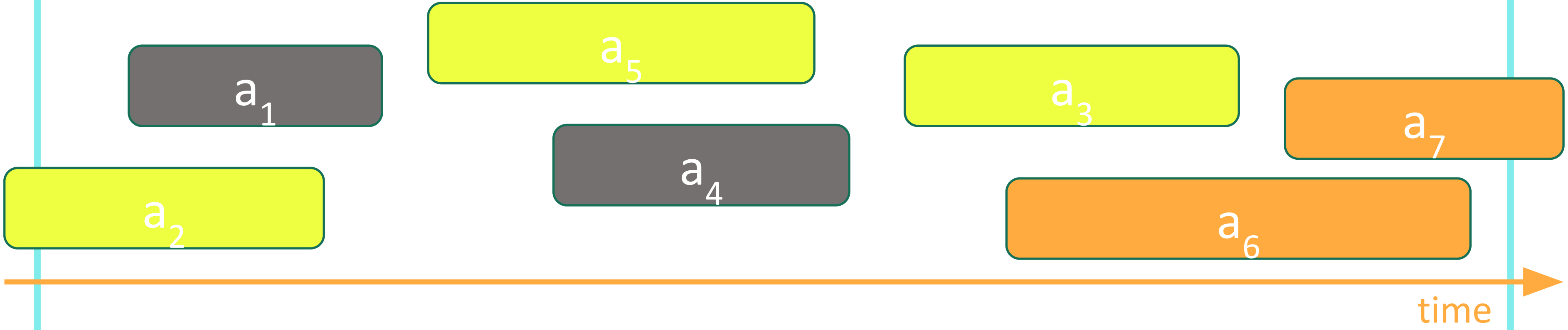
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



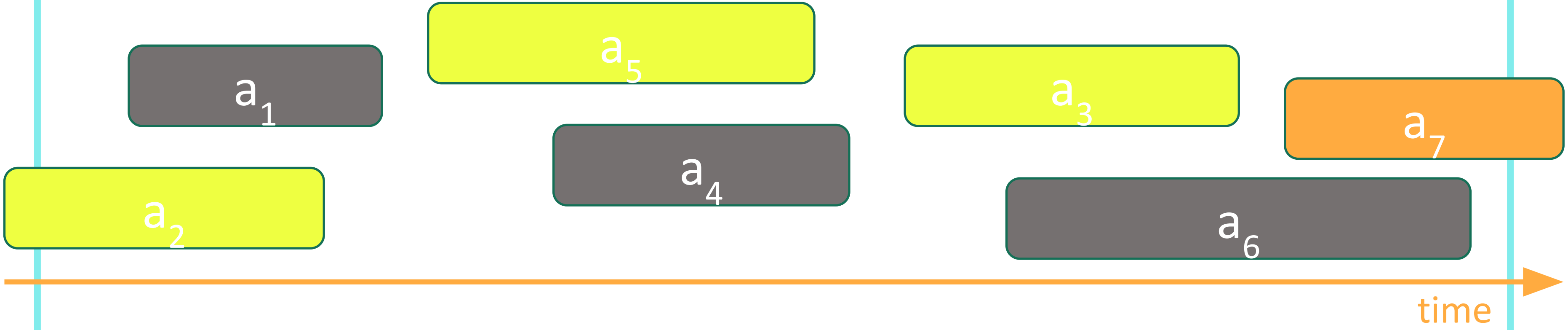
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



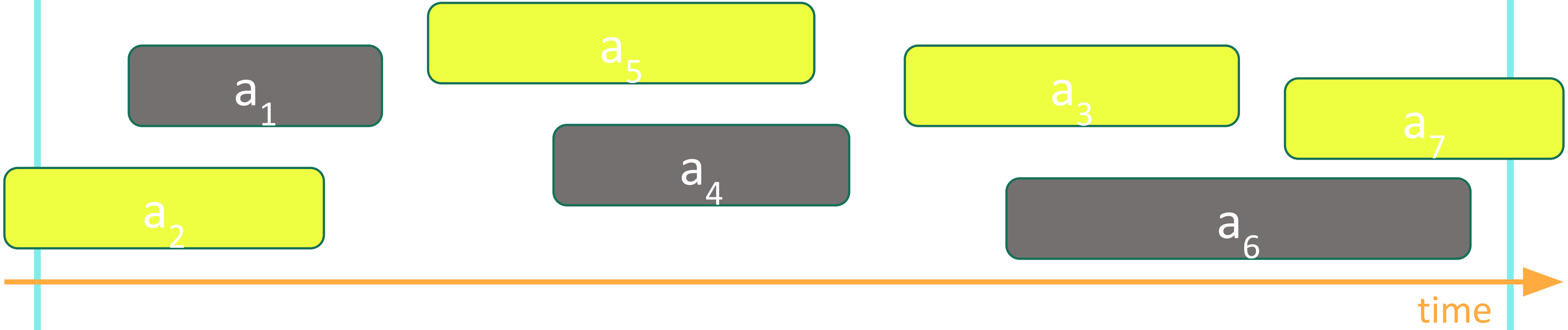
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

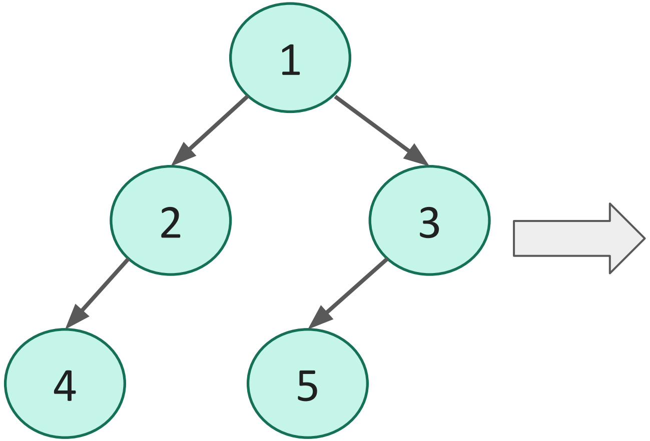
Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

This cannot be greedy!

Problem: Find root-to-leaf path of maximal sum in binary tree.



1+ 3 +5 = 9

When to Use a Greedy Approach?

Two properties need to be satisfied

1. **Optimal Substructure:** the optimal solution for a problem can be solved based on the optimal solutions to subproblems
2. **Greedy Property:** if you make a choice that seems to be best in the moment while solving the remaining sub-problems later, you still reach an optimal solution. You will never have to reconsider your earlier choices.

If #1 isn't satisfied, you can't use a greedy approach.

If #2 isn't satisfied, you'll end up with a sub-optimal solution.

Pros/Cons

- **Pros**:
    - Generally fast, easier to analyze the runtime
    - Can be more intuitive than other algorithmic approaches
    - Non-exhaustive, doesn’t search the whole solution space

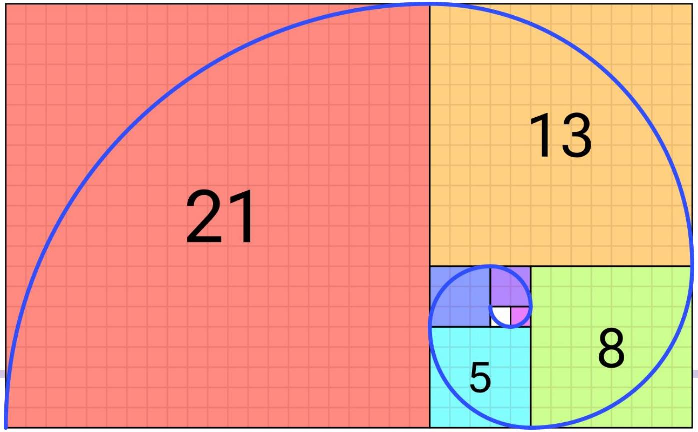
- **Cons**:
    - Difficult to prove correctness
    - Not always applicable
    - Non-exhaustive, doesn’t search the whole solution space
        - Won’t always reach optimal answer depending on the problem

Big Questions!

- What’s an example of dynamic programming?
- What is dynamic programming?
- How to dynamically program?

Fibonacci

- **Input:** which Fibonacci number which we want, n
- **Output:** the nth Fibonacci number
- Fibonacci is defined as follows: $F_n = F_{n-1} + F_{n-2}$ with base cases $F_1 = F_2 = 1$;
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Examples:
  - fib(1) = 1
  - fib(4) = 3
  - fib(10) = 55

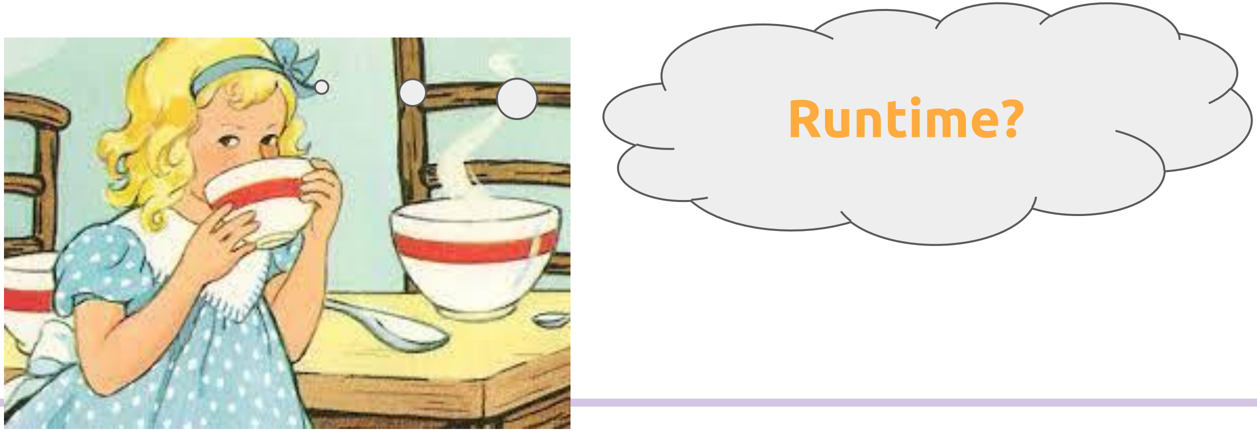


Fibonacci

- **def** Fibonacci(n):
  - **if** n == 0, **return** 0
  - **if** n == 1, **return** 1
  - **return** Fibonacci(n-1) + Fibonacci(n-2)

Fibonacci

- `def Fibonacci(n):`
    - `if n == 0, return 0`
    - `if n == 1, return 1`
    - `return Fibonacci(n-1) + Fibonacci(n-2)`



Fibonacci

- ```
  def Fibonacci(n):
  ```
  - ```
    if n == 0, return 0
    ```
  - ```
    if n == 1, return 1
    ```
  - ```
    return Fibonacci(n-1) + Fibonacci(n-2)
    ```

Recurrence relation: ...

$T(n) = T(n\text{-}1)+T(n\text{-}2)+c$

$T(n) > 2*T(n\text{-}2)+c$ // Approx: T(n-1) ~ T(n-2)

$T(n) > 2*(2*T(n\text{-}4)+c)+c = 4*T(n\text{-}4)$

$T(n) > 4*(2*T(n\text{-}6)+c) = 8*T(n\text{-}6)$

$T(n) > 2^k*T(n\text{-}2k)+(2^k - 1)*c$

Let's find the value of k for which:

$n - 2k = 0$

$k = n/2$

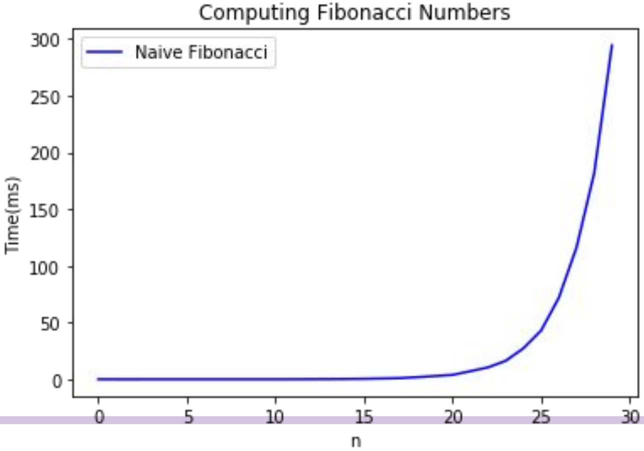
$T(n) > 2^{(n/2)} * T(0) + (2^{(n/2)} - 1)*c$

$> 2^{(n/2)} * (1 + c) - c$

$T(n) \sim 2^{(n/2)}$ or 2^n

Fibonacci

```
def Fibonacci(n):
    if n == 0, return 0
    if n == 1, return 1
    return Fibonacci(n-1) + Fibonacci(n-2)
```



Recurrence relation: ...

$T(n) = T(n-1)+T(n-2)+c$

$T(n) > 2*T(n-2)+c$ // Approx: $T(n-1) \sim T(n-2)$

$T(n) > 2*(2*T(n-4)+c)+c = 4*T(n-4)$

$T(n) > 4*(2*T(n-6)+c) = 8*T(n-6)$

$T(n) > 2^k * T(n - 2k) + (2^k - 1)*c$

Let's find the value of k for which:

$n - 2k = 0$

$k = n/2$

$T(n) > 2^{(n/2)} * T(0) + (2^{(n/2)} - 1)*c$

$> 2^{(n/2)} * (1 + c) - c$

$T(n) \sim 2^{(n/2)}$ or 2^n

Fibonacci

- `def Fibonacci(n):`
  - `if n == 0, return 0`
  - `if n == 1, return 1`

Recurrence relation:

But why...?

$\sim T(n-2)$

$n-4)$

$T(n-6)$

$+ T(n) > 2^k * T(n - 2k) + (2^k - 1)*c$

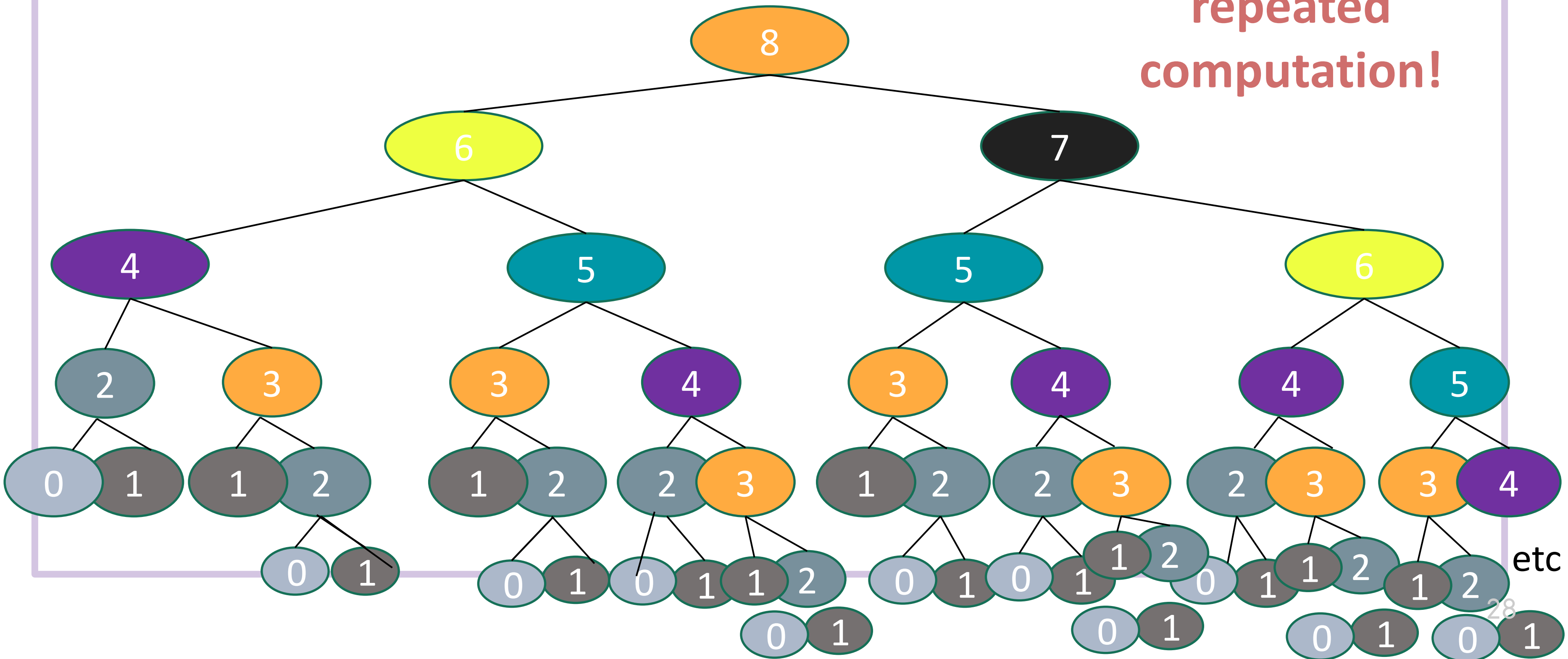
et's find the value of k for which:

$$n - 2k = 0$$
$$k = n/2$$

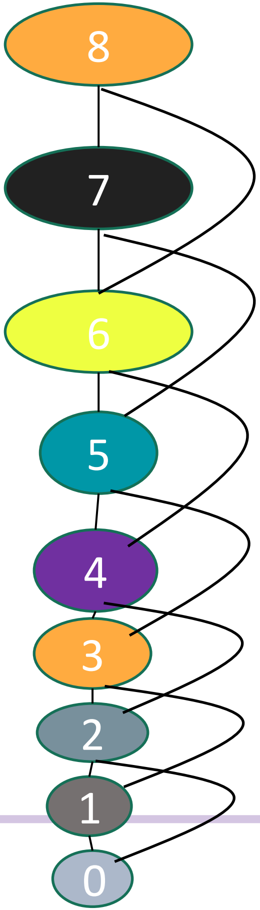
$$(n) > 2^{(n/2)} * T(0) + (2^{(n/2)} - 1)*c$$
$$> 2^{(n/2)} * (1 + c) - c$$
$$T(n) \sim 2^{(n/2)} \text{ or } 2^n$$

What's going on? Consider Fib(8)

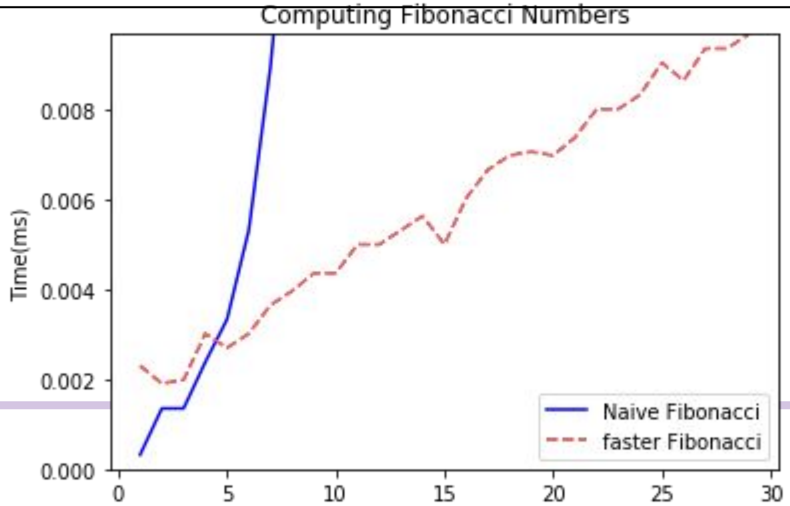
That's a lot of repeated computation!



Maybe this would be better?



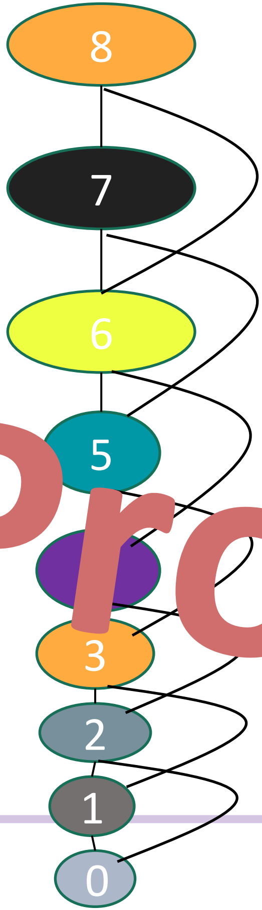

```
def fasterFibonacci(n):
    • F = [0, 1, None, None, …, None ]
        • \\ F has length n + 1
    • for i = 2, …, n:
        • F[i] = F[i-1] + F[i-2]
    • return F[n]
```



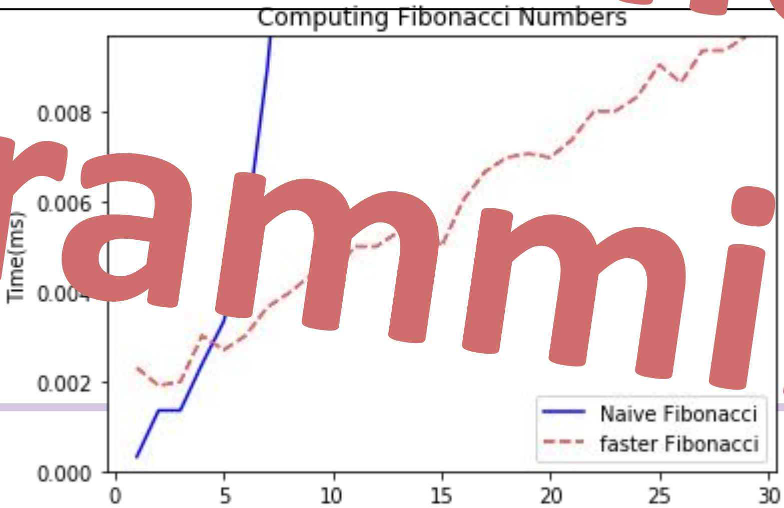
Much better running time!

Aside

Maybe this would be better?




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



Much better running time!

Aside

Dynamic Programming!

Big Questions!

- What's an example of dynamic programming?
- What is dynamic programming?
- How to dynamically program?

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 DP anyway…)
- Similar to greedy, there are two properties to look for…

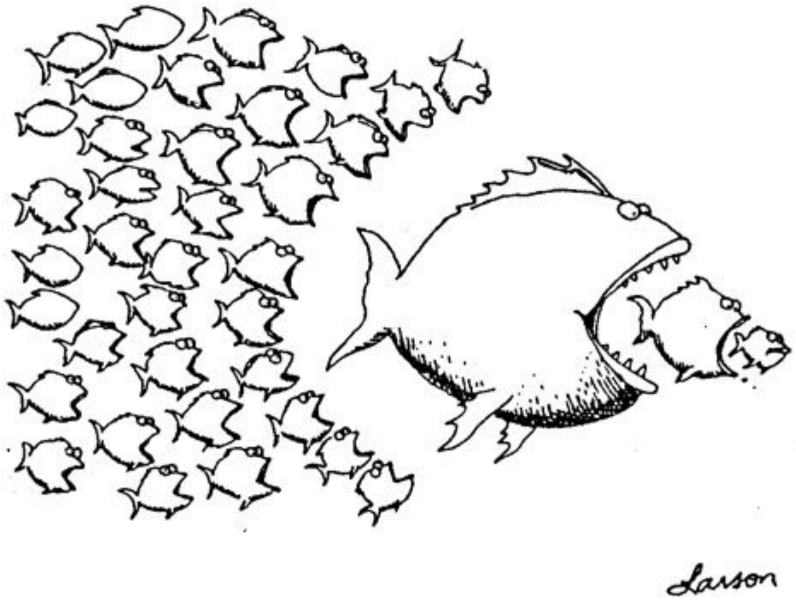
Properties of Dynamic Programming

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…**

Two ways to think about/implement dynamic programming

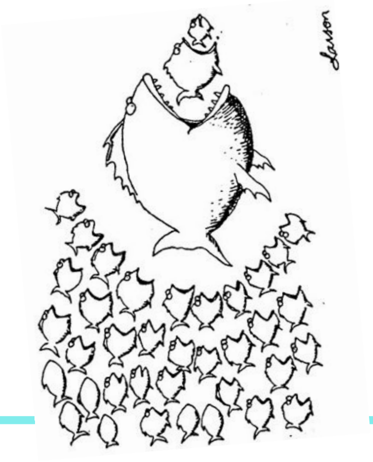
- Top down
- Bottom up



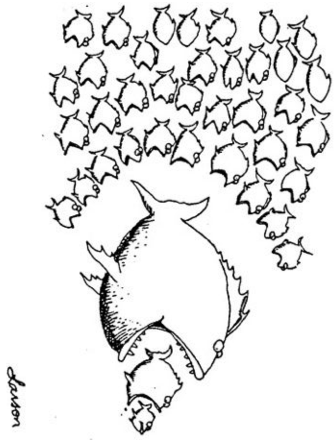
Bottom up approach (what we just saw!)

- 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..

- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "**memoization**"

Example of top-down Fibonacci

- define a global list F = [0,1,None, None, …, None]
- **def** Fibonacci(n):
  - **i**
  - **e**
  - -1) + Fibonacci(n-2)
  - **re**

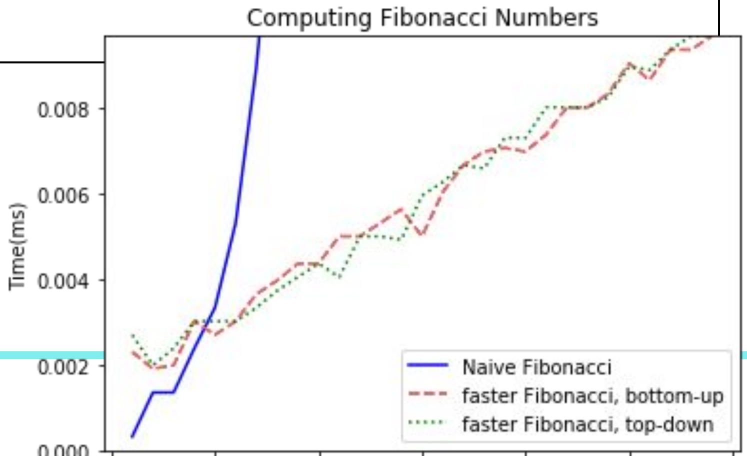
Which approach is faster?

Memo-ization: Keeps track (in F) of the stuff you've already done.

Example of top-down Fibonacci

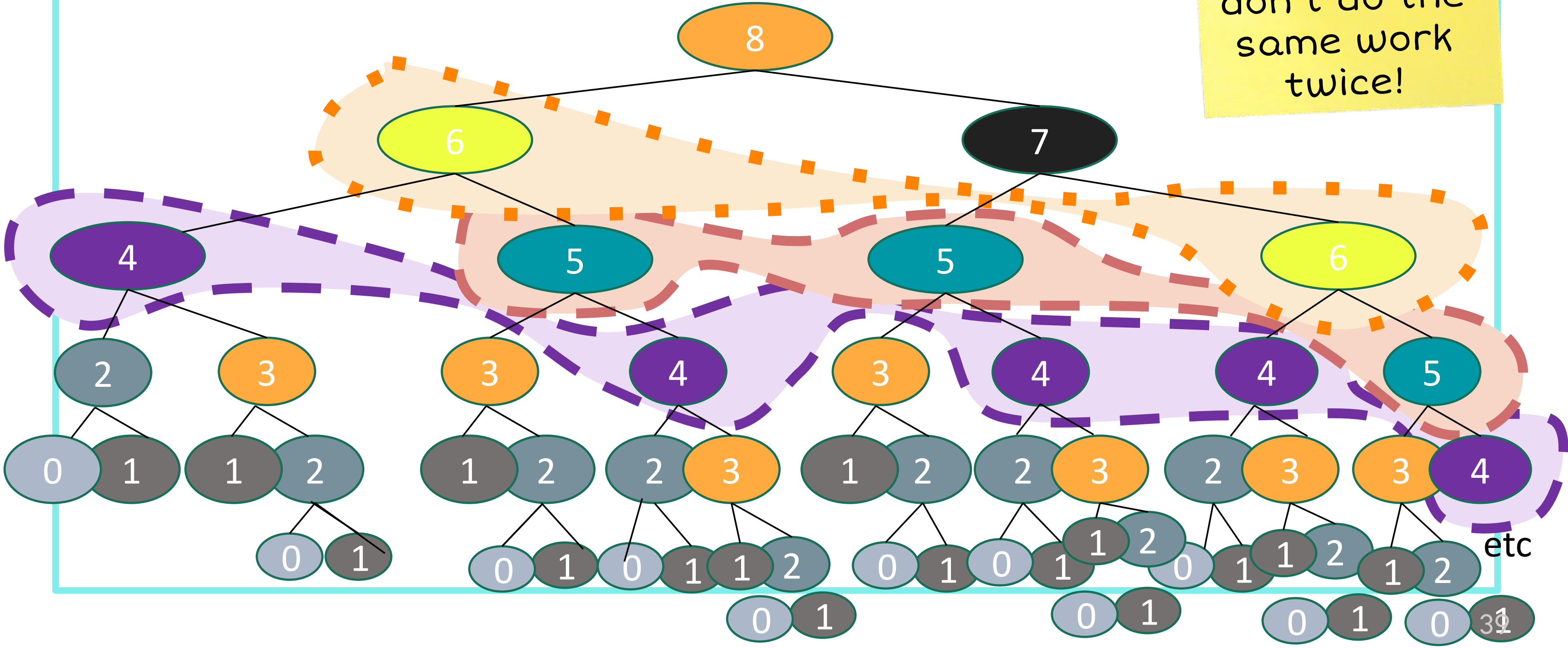
- define a global list F = [0,1,None, None, …, None]
- **def** Fibonacci(n):
    - **i**
    - **e**
    - -1) + Fibonacci(n-2)
    - **re**

Which approach is faster?



Memo-ization: Keeps track (in F) of the stuff you've already done.

Memoization visualization

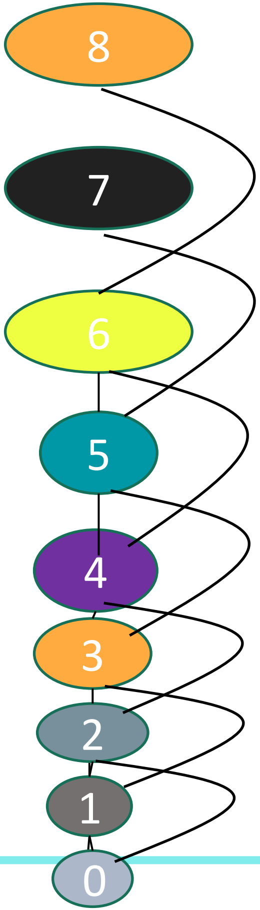


Memoization visualization

Collapse repeated nodes and don't do the same work twice!

But otherwise treat it like the same old recursive algorithm.

- 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]

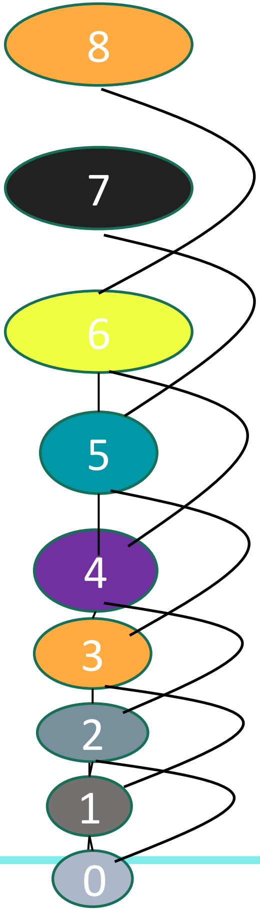


Top-down vs. Bottom Up Comparison

- 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]

def fasterFibonacci(n):

- F = [0, 1, None, None, …, None]
- **for** i = 2, …, n:
    - F[i] = F[i-1] + F[i-2]
- **return** F[n]



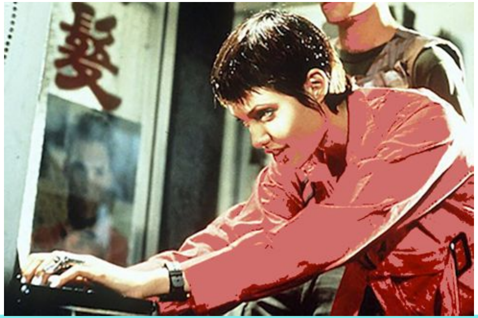
Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Why "dynamic programming"?

- Programming refers to finding the optimal "program."
  - as in, a shortest route is a *plan* aka a *program*.
- Dynamic refers to the fact that it's multi-stage.
- But also it's just a fancy-sounding name.



Manipulating computer code in an action movie?

Why “dynamic programming”?

- Richard Bellman invented the name in the 1950’s.
- At the time, he was working for the RAND Corporation, which was basically working for the Air Force, and government projects needed flashy names to get funded.
- From Bellman’s autobiography:
    - “It’s impossible to use the word, dynamic, in the pejorative sense…I thought dynamic programming was a good name. It was something not even a Congressman could object to.”

Big Questions!

- What's an example of dynamic programming?
- What is dynamic programming?
- How to dynamically program?

How to Create Algorithms with Dynamic Programming

1. Define recursive subproblem
    - 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 $N > 0$

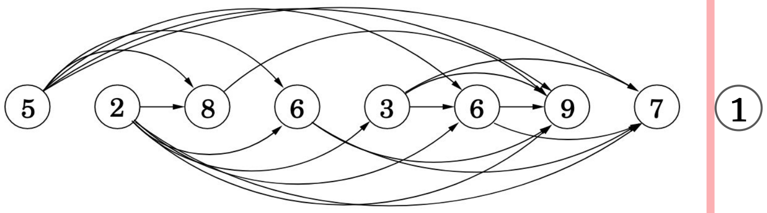
Output: length of the longest increasing subsequence within the vector

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

Example: $[5, 2, 8, 6, 3, 6, 9, 7, 1] \rightarrow 4$

Example: $[6, 1, 8, 2, 3, 1] \rightarrow ???$

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] → ???

Example: Longest Increasing Subsequence

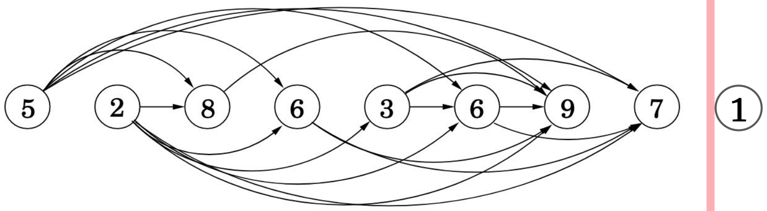
Input: vector of integers vec of size N > 0

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] → ???



1. Define recursive subproblem
2. Relate subproblems (with base-cases)
3. Top-down with memoization or build DP table bottom-up with ordering
4. Solve original problem

Example: Longest Increasing Subsequence

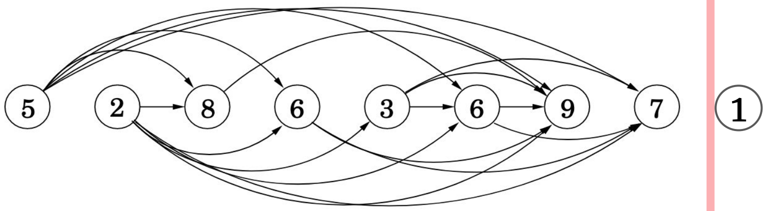
Input: vector of integers vec of size N > 0

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] → ???



1. L[i] = # of vertices on longest path ending at index i.
2. Relate subproblems (with base-cases)
3. Top-down with memoization or build DP table bottom-up with ordering
4. Solve original problem

Example: Longest Increasing Subsequence

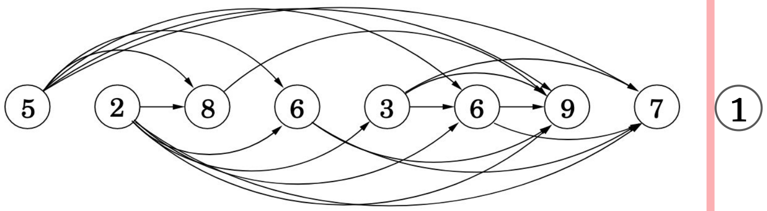
Input: vector of integers vec of size N > 0

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] → ???



1. L[i] = # of vertices on longest path ending at index i.
2. L[i] = 1 + max(L[j] for j in 0…i-1 if vec[i] > vec[j]), or 1 if can't build on anything.
3. Top-down with memoization or build DP table bottom-up with ordering
4. Solve original problem

Example: Longest Increasing Subsequence

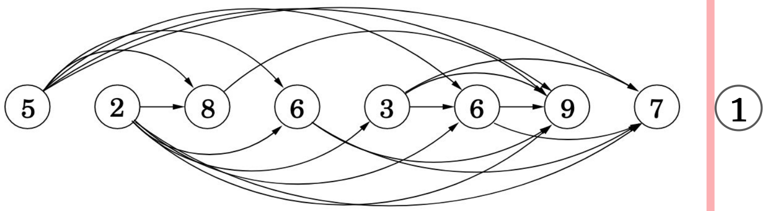
Input: vector of integers vec of size N > 0

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] → ???



1. L[i] = # of vertices on longest path ending at index i.
2. L[i] = 1 + max(L[j] for j in 0...i-1 if vec[i] > vec[j]), or 1 if can't build on anything.
3. Solve i = 0, 1, 2, … n - 1
4. Solve original problem

Example: Longest Increasing Subsequence

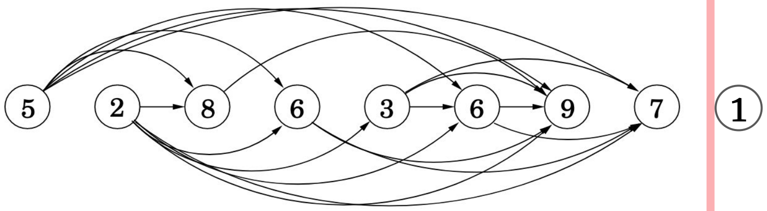
Input: vector of integers vec of size N > 0

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] → ???



1. L[i] = # of vertices on longest path ending at index i.
2. L[i] = 1 + max(L[j] for j in 0...i-1 if vec[i] > vec[j]), or 1 if can't build on anything.
3. Solve i = 0, 1, 2, … n - 1
4. Return max value in table

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, memo[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

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, memo[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

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, memo[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

What have we learned?

Dynamic programming

- Paradigm in algorithm design.
- Uses optimal substructure
- Uses overlapping subproblems
- Can be implemented bottom-up or top-down.
- It's a fancy name for a pretty common-sense idea:

COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 20: Dynamic Programming I

Lecturer: Chris Lucas (cflucas@ncat.edu)