CSE100: Design and Analysis of Algorithms
Lecture 23 – More Dynamic Programming (wrap up) and Greedy Algorithms

Apr 19th 2022

Knapsack and Greedy Algorithms!

Capacity: 10

Item:	turtle	light bulb	watermelon	taco	fire truck
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

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

Total weight: 10
Total value: 42

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

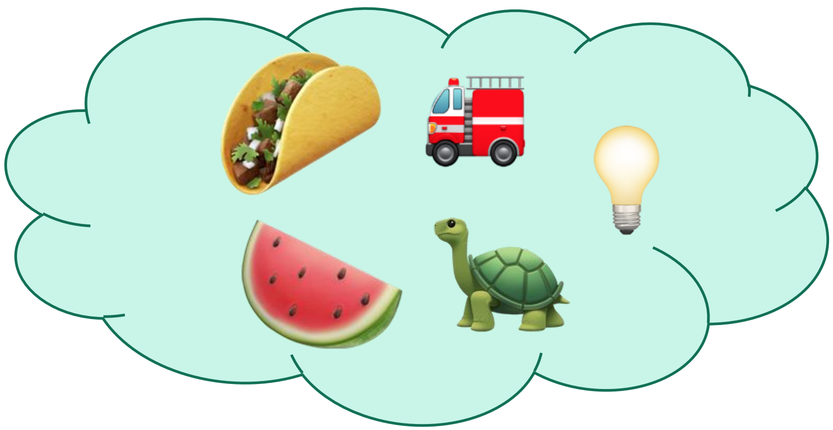
Total weight: 9
Total value: 35

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Our sub-problems:

- Indexed by x and j



First j items

Capacity x

K[x,j] = optimal solution for a knapsack of size x using only the first j items.

Recursive relationship

- Let K[x,j] be the optimal value for:
  - capacity x,
  - with j items.

$$K[x,j] = \max\{ K[x, j-1] , K[x - w_j, j-1] + v_j \}$$

Case 1: optimal solution for j items doesn't use item j

Case 2: optimal solution for j items does use item j

- (And K[x,0] = 0 and K[0,j] = 0).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

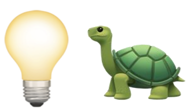
Bottom-up DP algorithm

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1] Case 1
      - **if** $w_j \leq x$:
        - Case 2
        - K[x,j] = max{ K[x,j], K[$x - w_j$, j-1] + v_j }
  - **return** K[W,n]

Running time O(nW)

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
 j=1	0			
 j=2	0			
 j=3	0			

current entry

relevant previous entry

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

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	0		
j=2	0			
j=3	0			

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
🐢 j=1	0	1 🐢		
💡 🐢 j=2	0			
🍉 💡 🐢 j=3	0			

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1		
j=2	0	1		
j=3	0			

current entry

relevant previous entry

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

Capacity: 3

Example

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
🐢 j=1	0	1 🐢		
💡 🐢 j=2	0	1 🐢		
🍉 💡 🐢 j=3	0	1 🐢		

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], $K[x - w_j, j-1] + v_j$ }
  - **return** K[W,n]

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], $K[x - w_j, j-1] + v_j$ }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1 (turtle)	0	1 (turtle)	0	
j=2 (lightbulb, turtle)	0	1 (turtle)		
j=3 (watermelon, lightbulb, turtle)	0	1 (turtle)		

current entry

relevant previous entry

Item:	turtle	lightbulb	watermelon
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], $K[x - w_j, j-1] + v_j$ }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1		
j=3	0	1		

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], $K[x - w_j, j\text{-}1] + v_j$ }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1	1	
j=3	0	1		

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], $K[x - w_j, j-1] + v_j$ }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1	4	
j=3	0	1		

current entry

relevant previous entry

Item:	turtle	lightbulb	watermelon
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	
j=2	0	1	4	
j=3	0	1	4	

current entry

relevant previous entry

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

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	0
j=2	0	1	4	
j=3	0	1	4	

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	
j=3	0	1	4	

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[$x - w_j$, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	1
j=3	0	1	4	

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	5
j=3	0	1	4	

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

	x=0	x=1	x=2	x=3
j=0	0	0	0	0
j=1	0	1	1	1
j=2	0	1	4	5
j=3	0	1	4	5

current entry

relevant previous entry

Item:

Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], K[x – w_j, j-1] + v_j }
  - **return** K[W,n]

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	5 🐢 💡
🍉 💡 🐢	j=3	0	1 🐢	4 💡	6 🍉

current entry

relevant previous entry

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

Capacity: 3

Example

- Zero-One-Knapsack(W, n, w, v):
  - K[x,0] = 0 for all x = 0,…,W
  - K[0,i] = 0 for all i = 0,…,n
  - **for** x = 1,…,W:
    - **for** j = 1,…,n:
      - K[x,j] = K[x, j-1]
      - **if** $w_j \leq x$:
        - K[x,j] = max{ K[x,j], $K[x - w_j, j-1] + v_j$ }
  - **return** K[W,n]

		x=0	x=1	x=2	x=3
	j=0	0	0	0	0
🐢	j=1	0	1 🐢	1 🐢	1 🐢
💡 🐢	j=2	0	1 🐢	4 💡	5 🐢 💡
🍉 💡 🐢	j=3	0	1 🐢	4 💡	6 🍉

So the optimal solution is to put one watermelon in your knapsack!

current entry

relevant previous entry

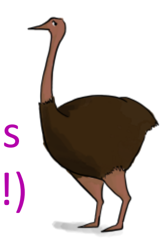
Item:	🐢	💡	🍉
Weight:	1	2	3
Value:	1	4	6

Capacity: 3

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

You do this one!
(We did it on the slide in the previous example, just not in the pseudocode!)



What have we learned?

- We can solve 0/1 knapsack in time O(nW).
  - If there are n items and our knapsack has capacity W.

- We again went through the steps to create DP solution:
  - We kept a two-dimensional table, creating smaller problems by restricting the set of allowable items.

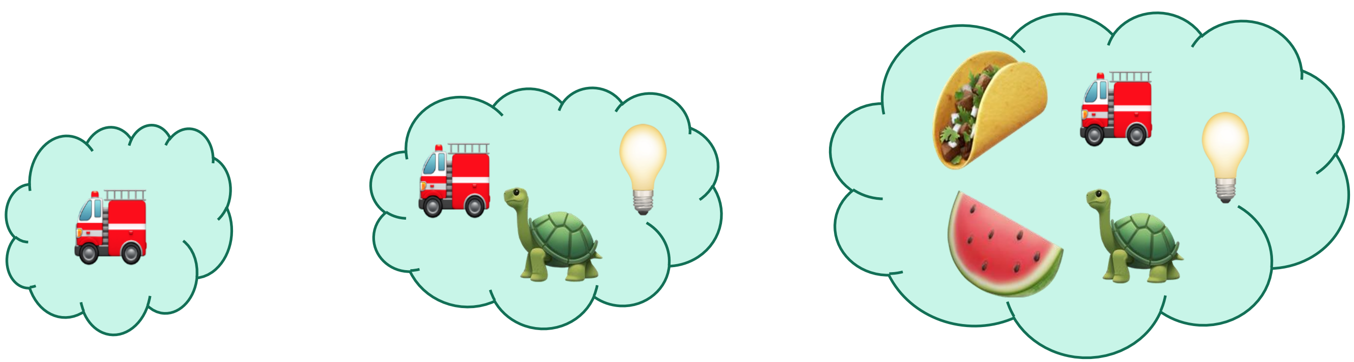
Question

- How did we know which substructure to use in which variant of knapsack?

Answer in retrospect:

This one made sense for unbounded knapsack because it doesn't have any memory of what items have been used.

vs.



In 0/1 knapsack, we can only use each item once, so it makes sense to leave out one item at a time.

Operational Answer: try some stuff, see what works!

Recap

- Saw examples of how to come up with dynamic programming algorithms.
  - Longest Common Subsequence
  - Knapsack two ways
  - (If time) maximal independent set in trees.
- There is a **recipe** for dynamic programming algorithms.

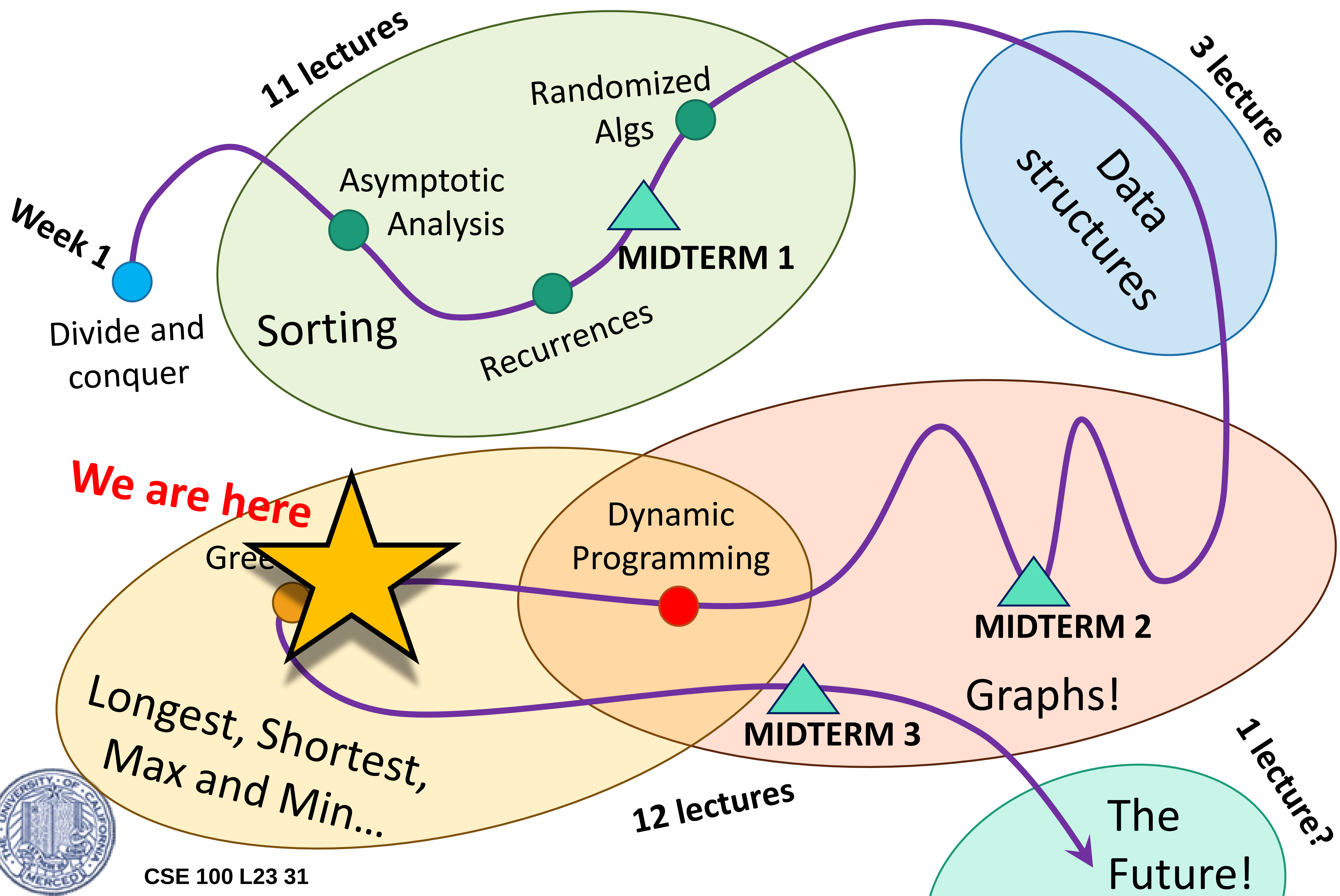
Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Recap

- Saw examples of how to come up with dynamic programming algorithms.
  - Longest Common Subsequence
  - Knapsack two ways
  - (If time) maximal independent set in trees.
- There is a **recipe** for dynamic programming algorithms.
- Sometimes coming up with the right substructure takes some creativity
  - You'll get lots of practice on Discussion Chapter 15! ☺

Roadmap



11 lectures
Randomized Algs
Asymptotic Analysis
MIDTERM 1
Week 1
Divide and conquer
Sorting
Recurrences
3 lecture
Data structures
We are here
Gree
Dynamic Programming
MIDTERM 2
Graphs!
MIDTERM 3
Longest, Shortest, Max and Min…
12 lectures
1 lecture?
The Future!

This week

- Greedy algorithms!

Gordon Geckko in Wall Street (1987)

Greedy algorithms

- Make choices one-at-a-time.
- Never look back.
- Hope for the best.

Today

- One example of a greedy algorithm that **does not work**:
  - Knapsack again
- Three examples of greedy algorithms that **do work**:
  - Activity Selection
  - Job Scheduling
  - Huffman Coding

Non-example

- Unbounded Knapsack.

Capacity: 10

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

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

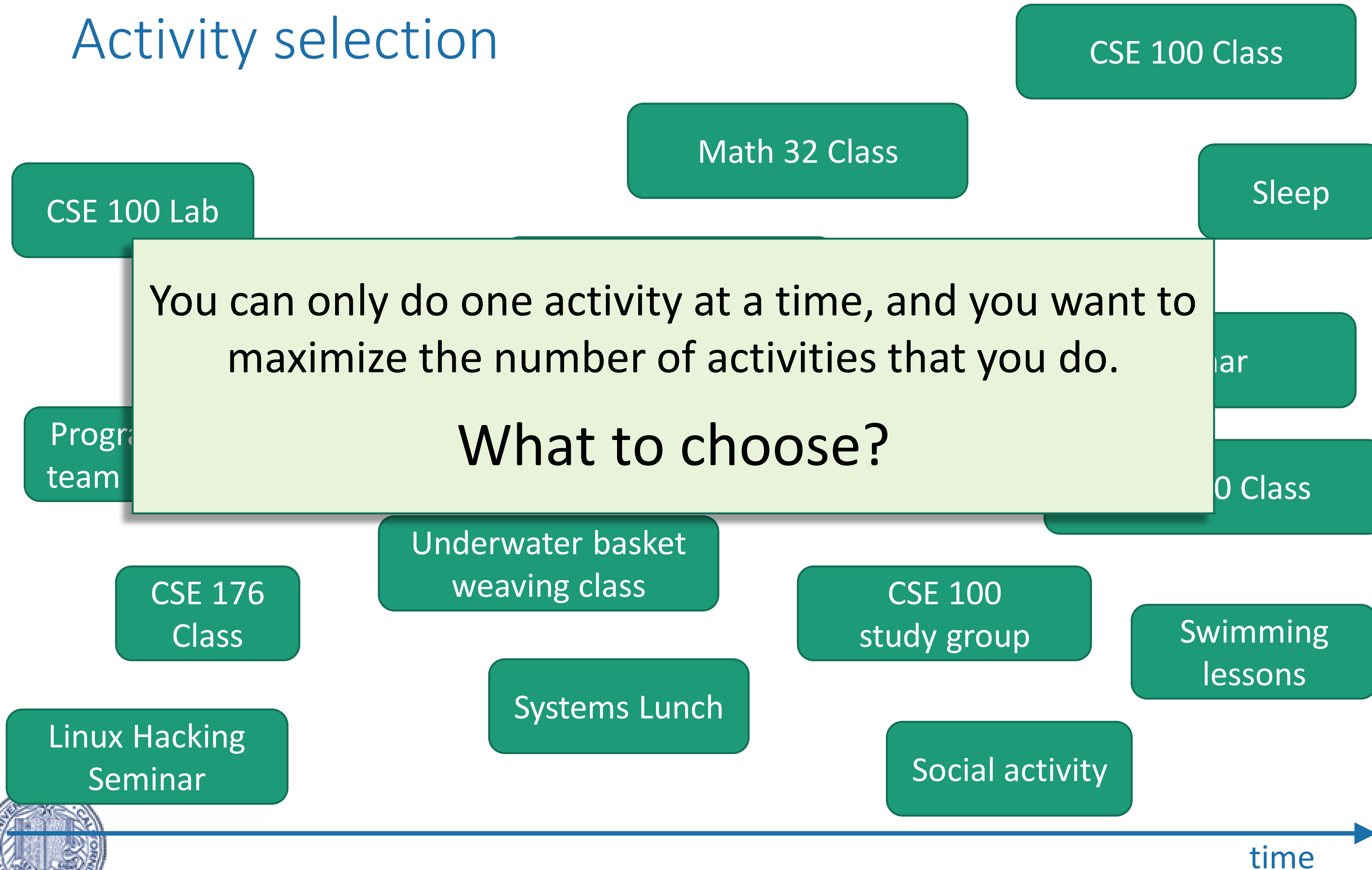
Total weight: 10
Total value: 42

- "Greedy" algorithm for unbounded knapsack:
  - Tacos have the best Value/Weight ratio!
  - Keep grabbing tacos!

Total weight: 9
Total value: 39

Example where greedy works

Activity selection

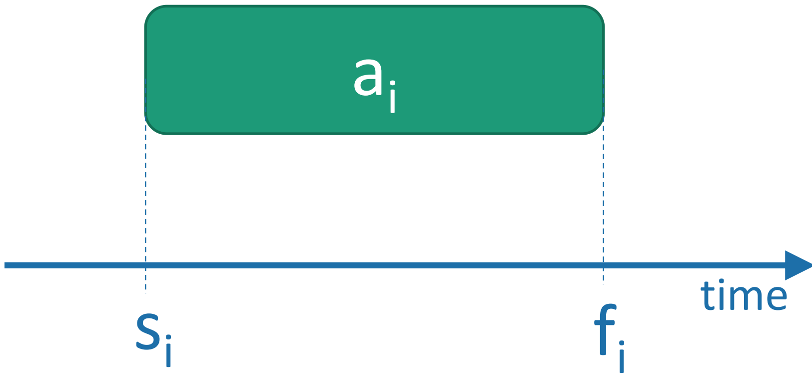


You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

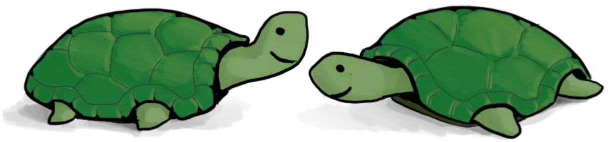
Activity selection

- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$



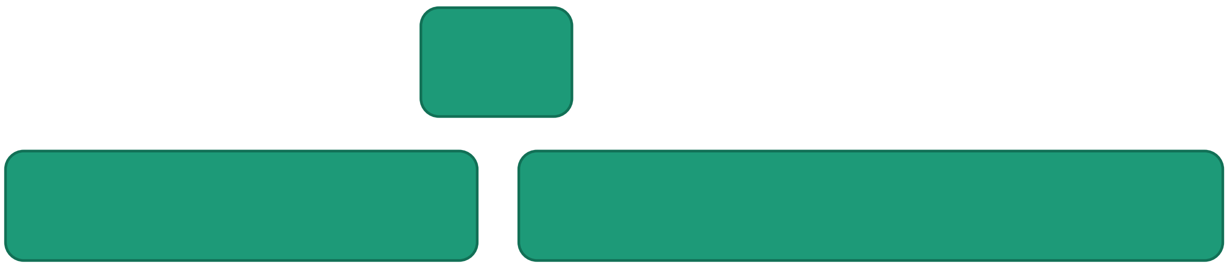
- Output:
  - A way to maximize the number of activities you can do today.

In what order should you greedily add activities?



Think-pair-share!

Shortest job first?



Anakin the adversarial aardvark

Fewest conflicts first?

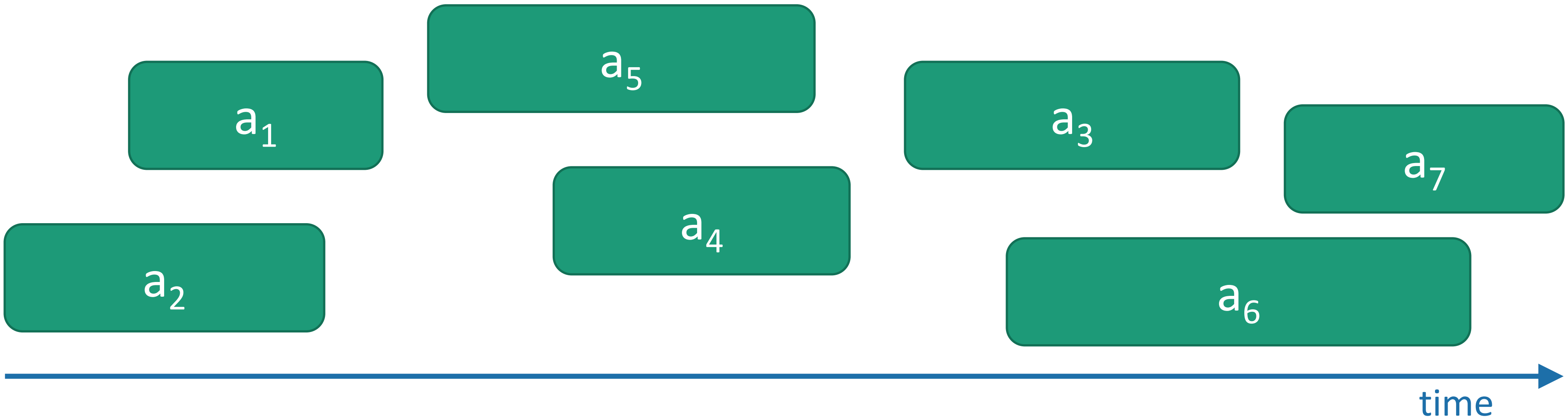
Anakin the adversarial aardvark

Earliest ending time first?

- This will do it!

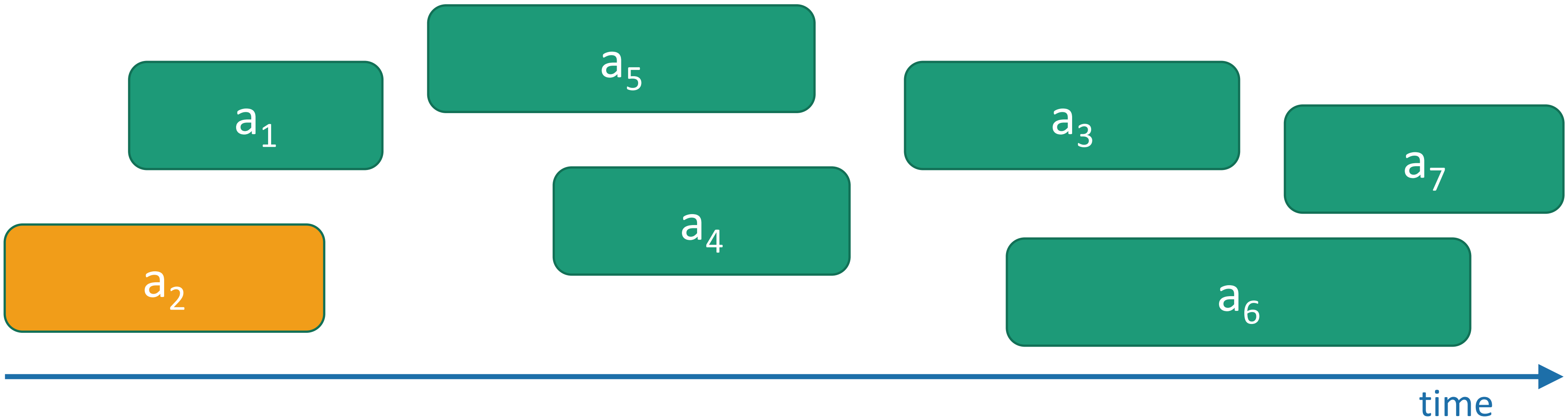
Anakin the adversarial aardvark

Greedy Algorithm



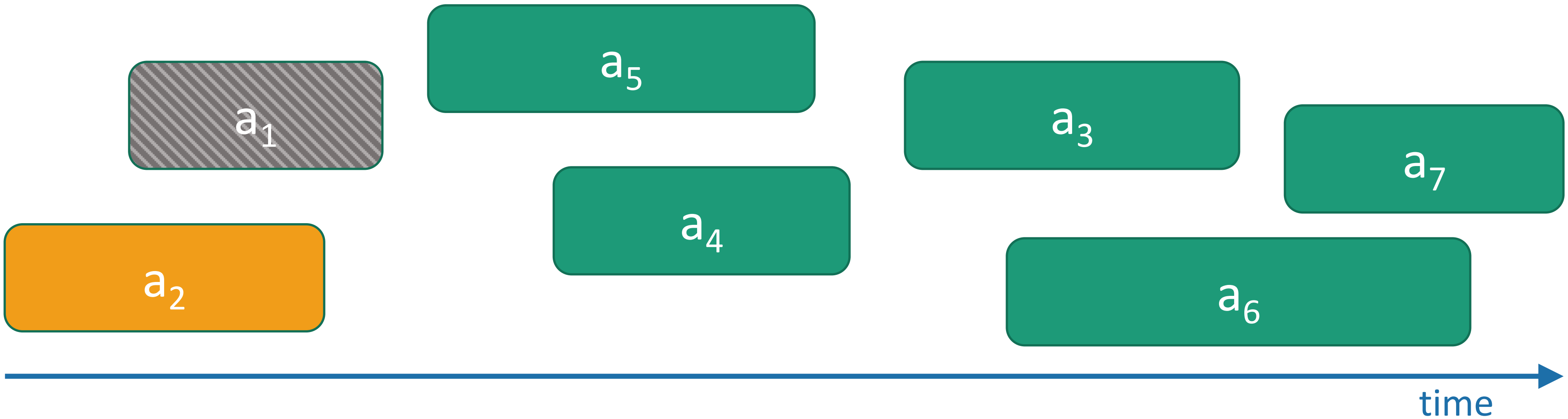
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



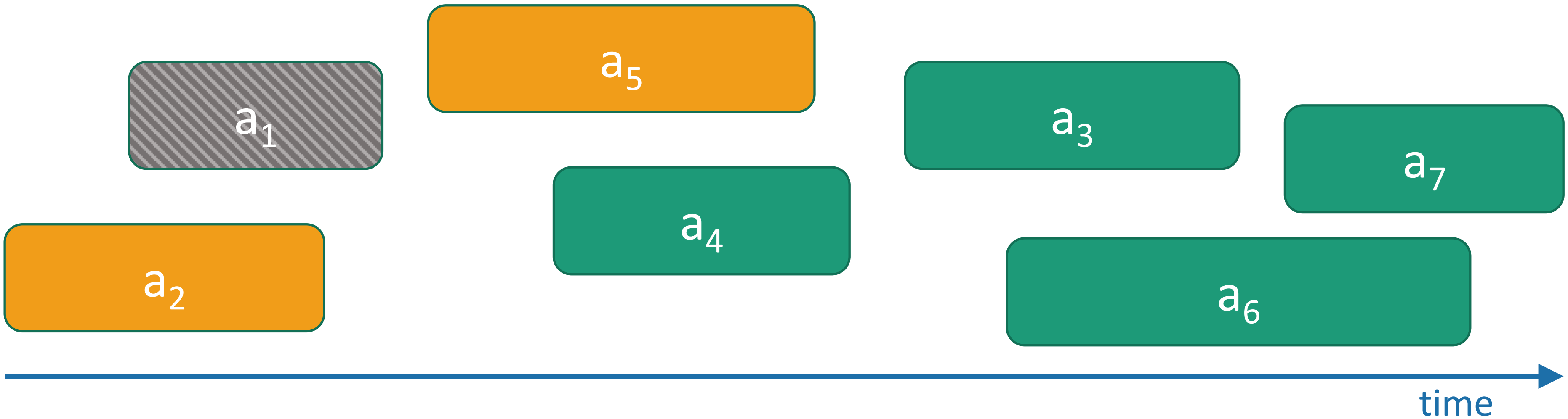
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



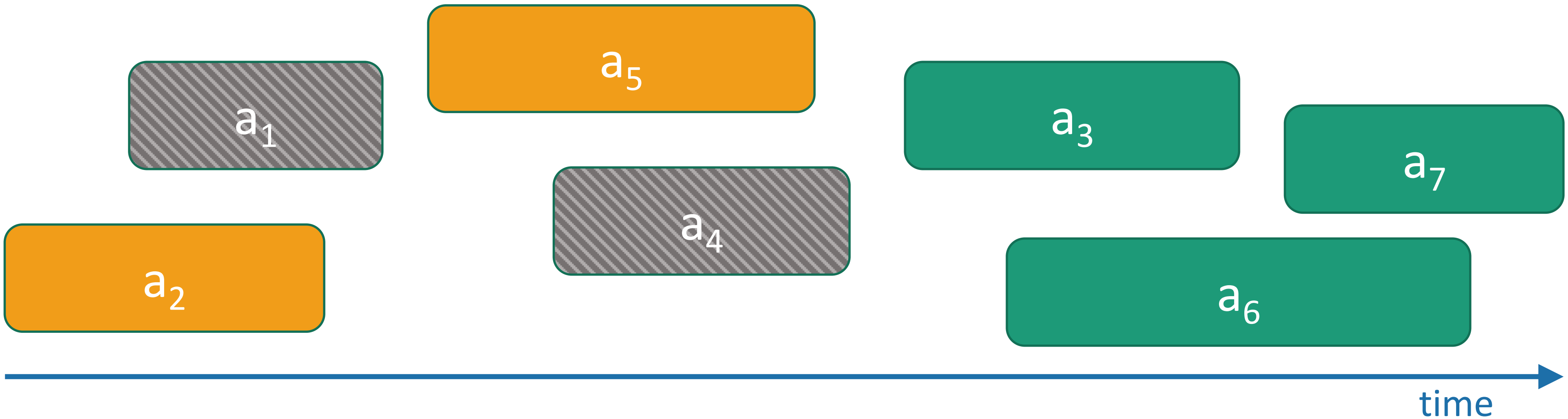
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



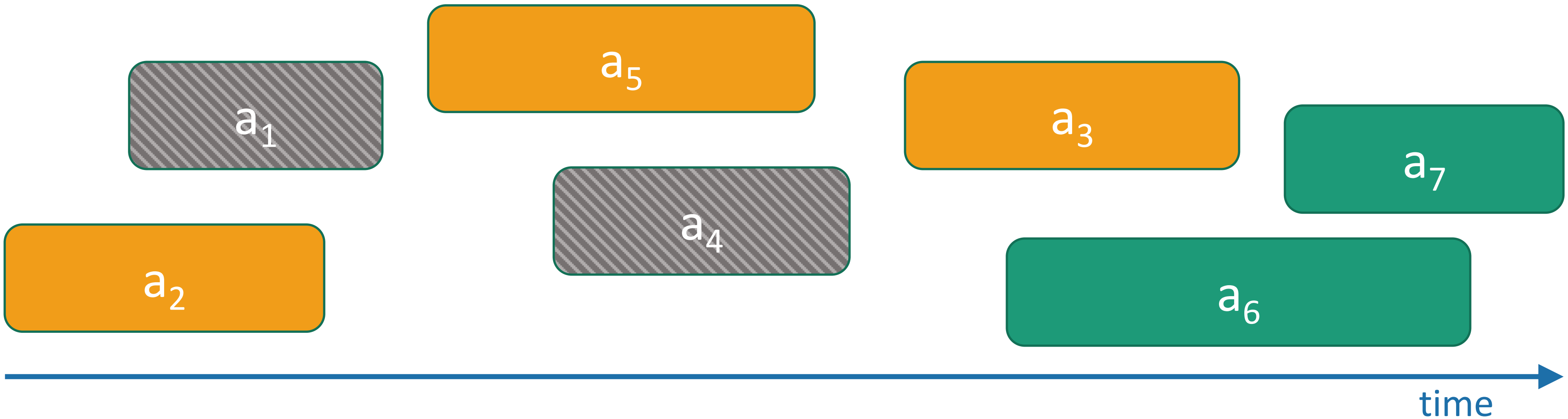
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



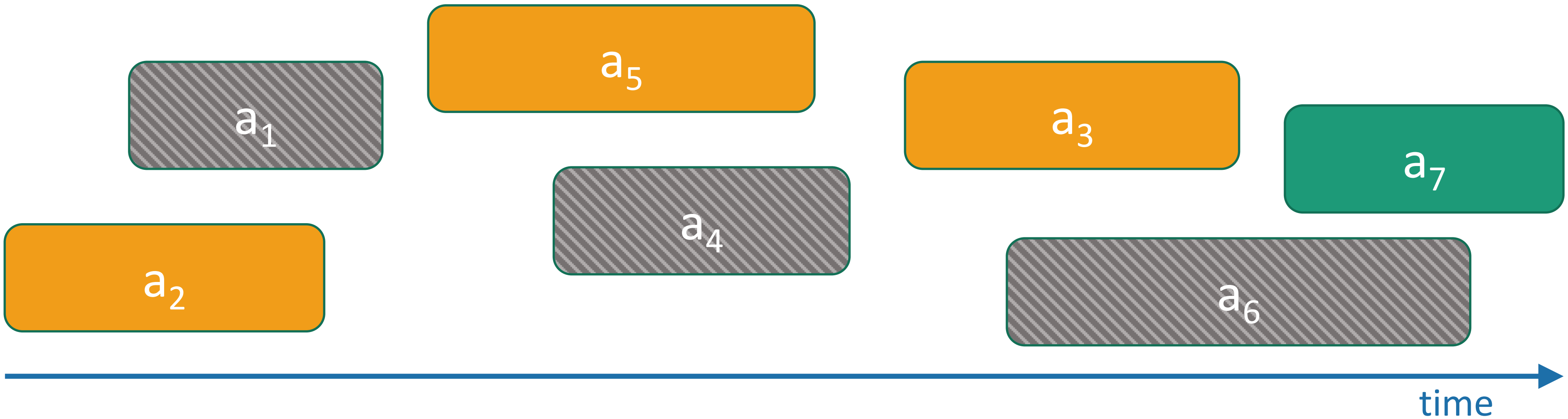
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



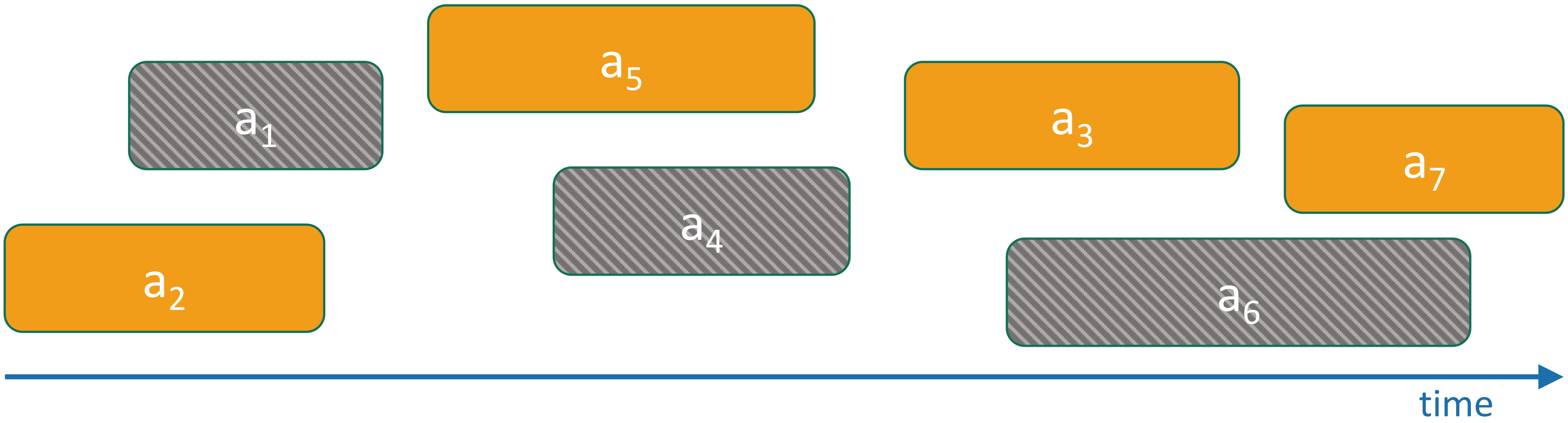
- Pick activity you can add with the smallest finish time.
- Repeat.

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.

At least it's fast

- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise O(nlog(n)) if you have to sort them first.

What makes it greedy?

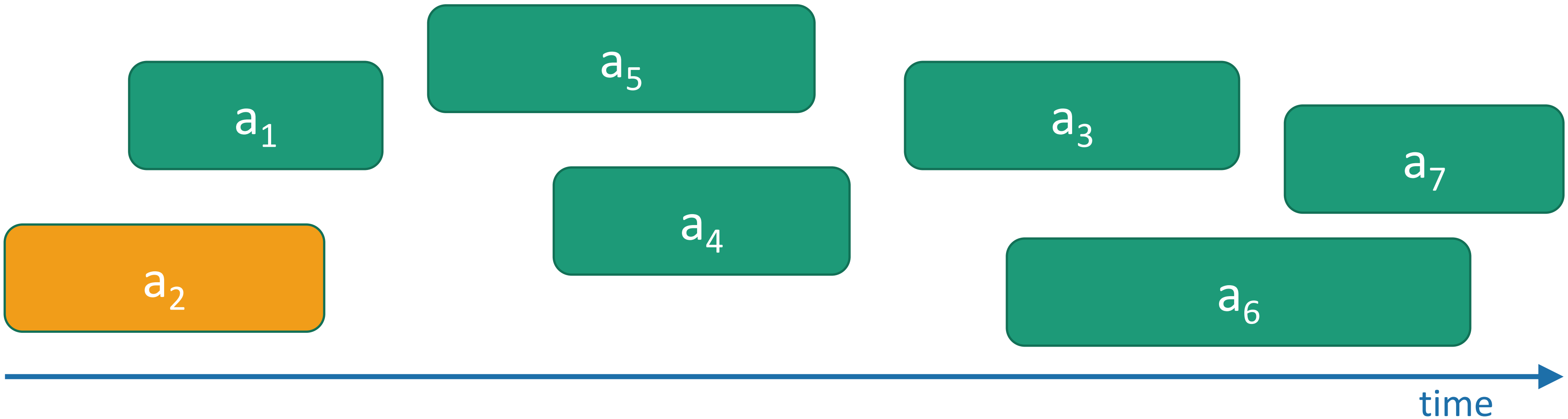
- At each step in the algorithm, make a choice.
  - Hey, I can increase my activity set by one,
  - And leave lots of room for future choices,
  - Let's do that and hope for the best!!!

- **Hope** that at the end of the day, this results in a globally optimal solution.

Three Questions

1. Does this greedy algorithm for activity selection work?
   - Yes. (We will see why in a moment…)
2. In general, when are greedy algorithms a good idea?
   - When the problem exhibits especially nice optimal substructure.
3. The "greedy" approach is often the first you'd think of…
   - Why are we getting to it now, in Week 14?
     - Proving that greedy algorithms work is often not so easy…

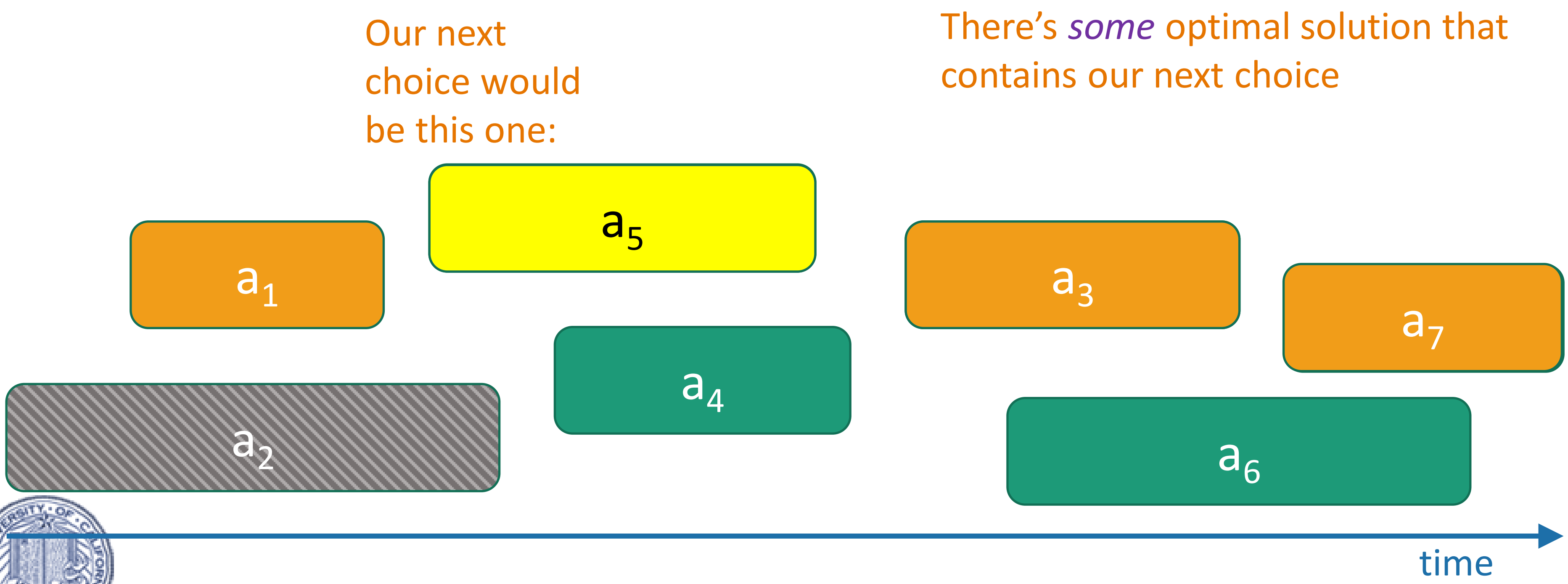
Back to Activity Selection



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

Why does it work?

- Whenever we make a choice, **we don't rule out an optimal solution.**



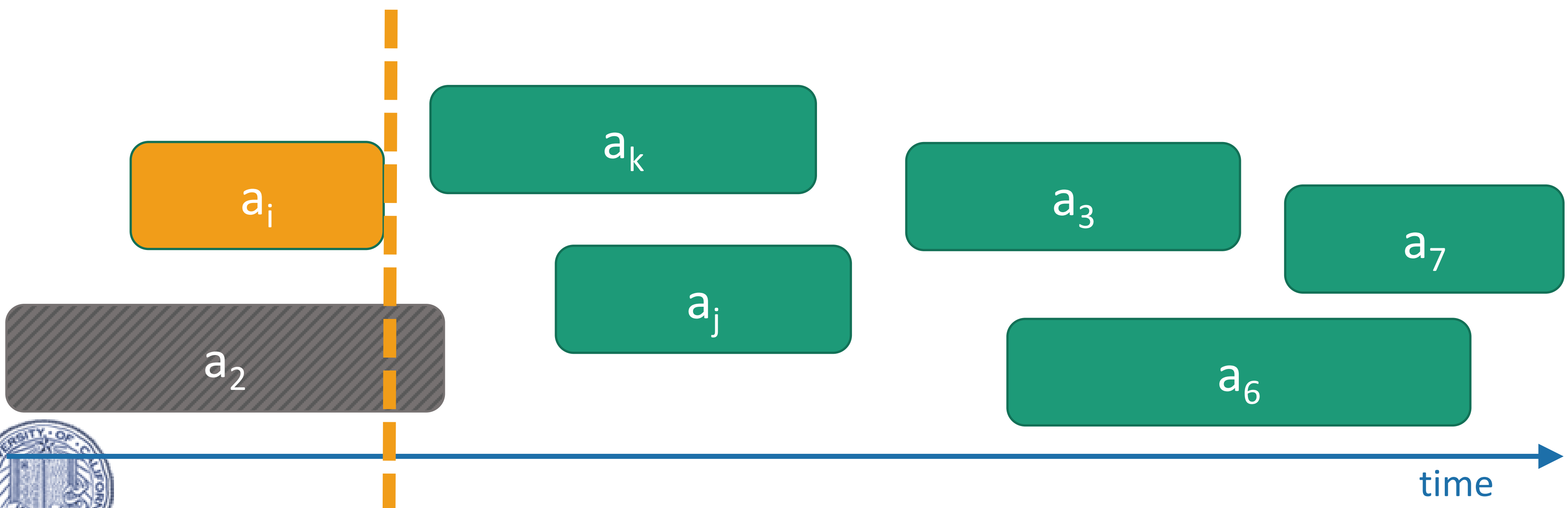
Assuming we can prove that

- **We never rule out an optimal solution**
- At the end of the algorithm, we've got some solution.
- So it must be optimal.

Lucky the Lackadaisical Lemur

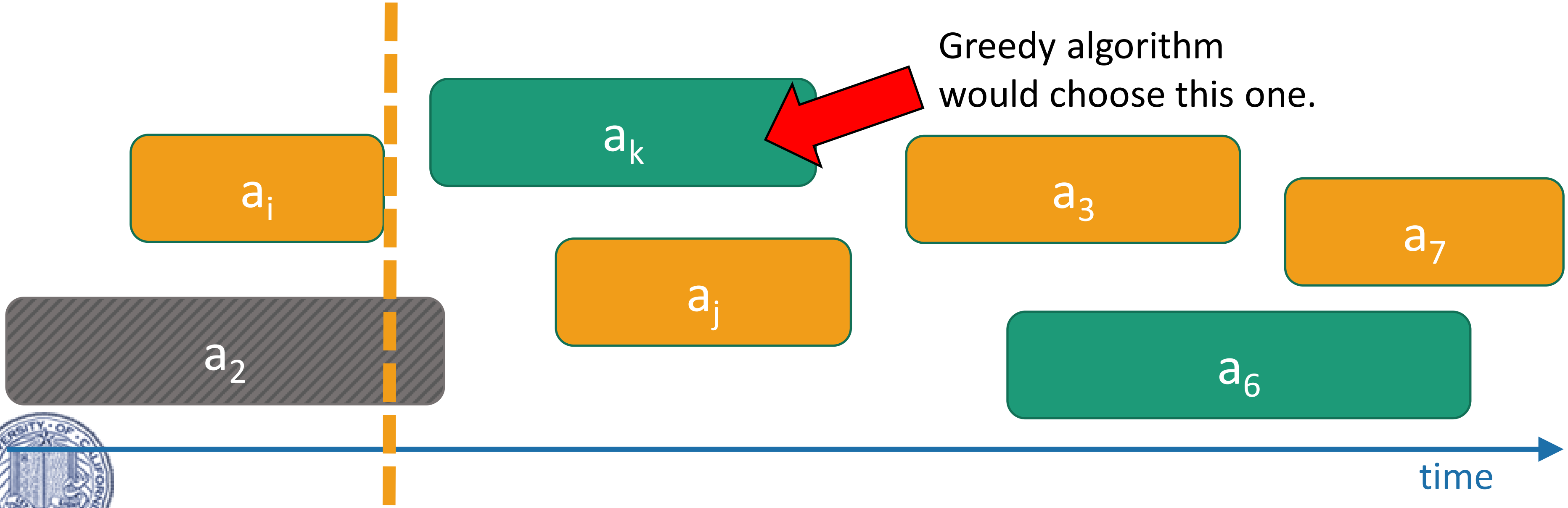
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.



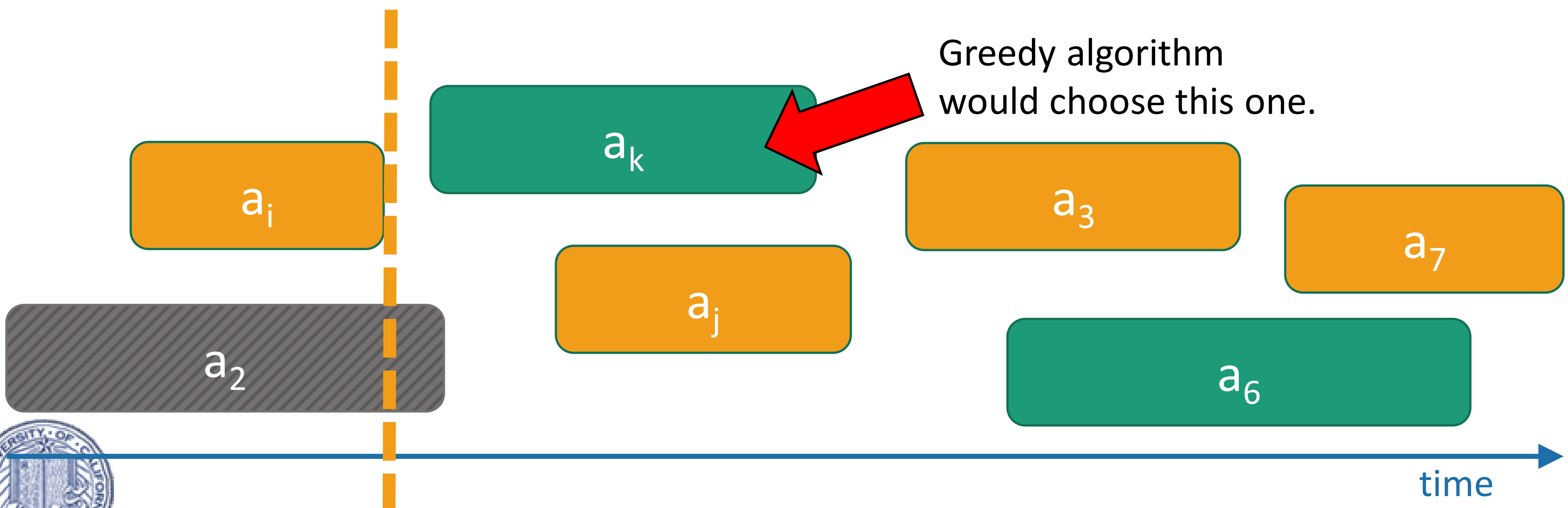
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is in T*, we're still on track.



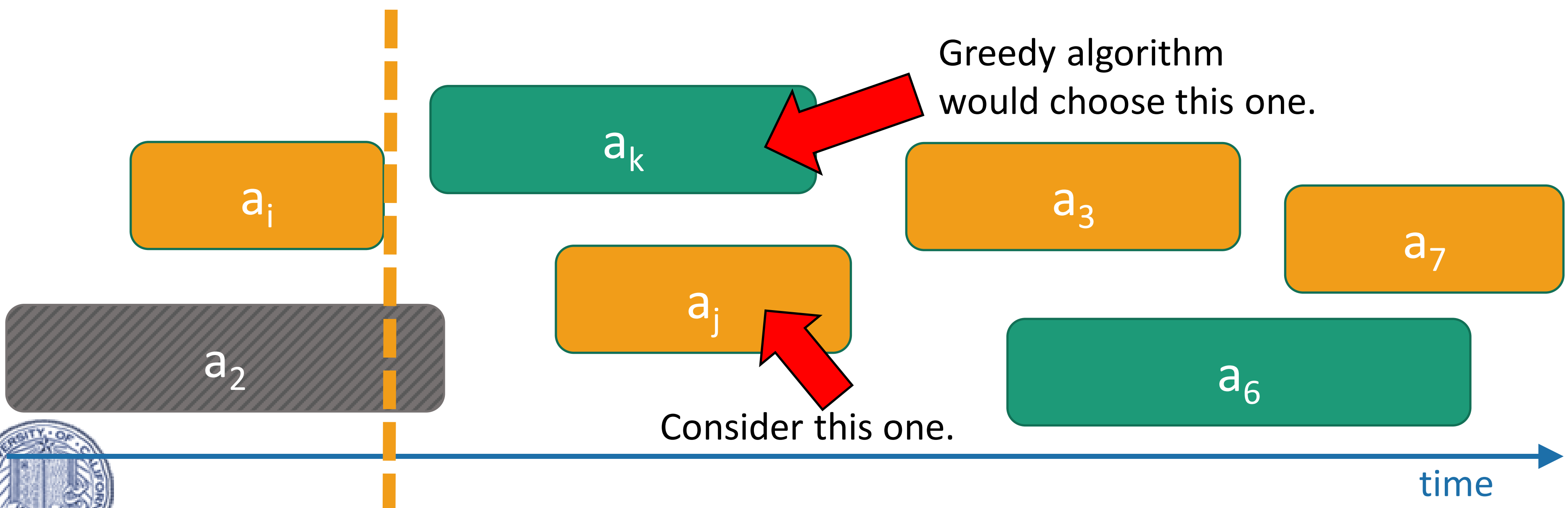
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is **not** in T*...



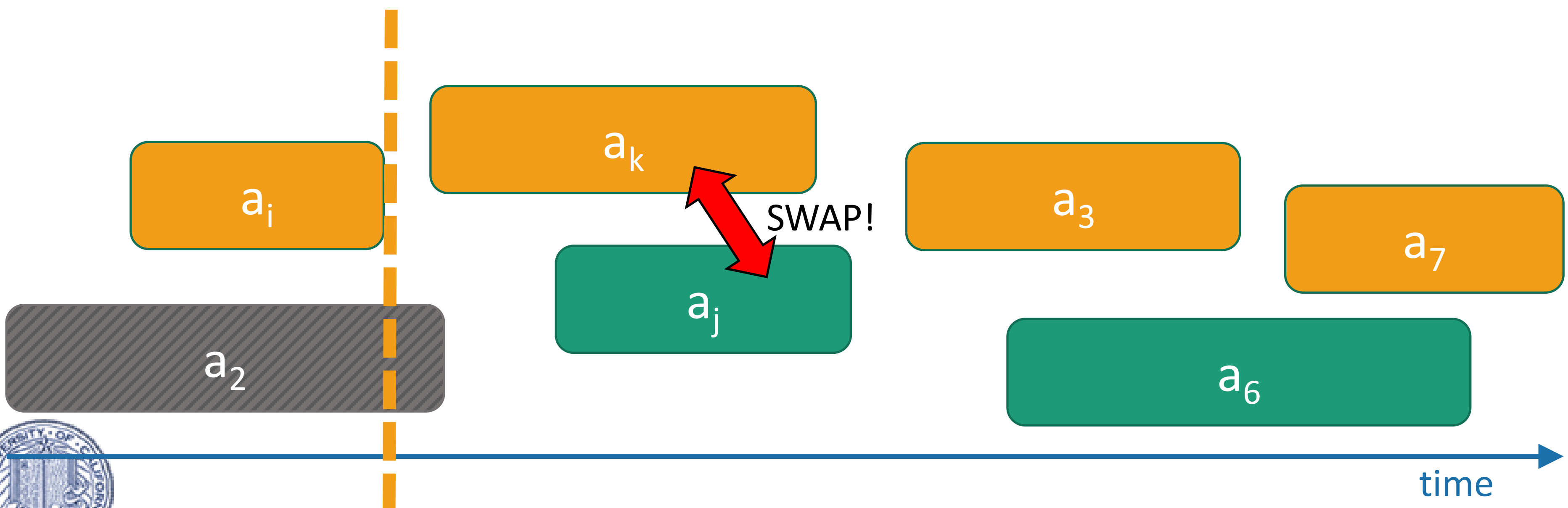
We never rule out an optimal solution ctd.

- If a_k is **not** in T*…
- Let a_j be the activity in T* with the smallest end time.
- Now consider schedule T you get by swapping a_j for a_k



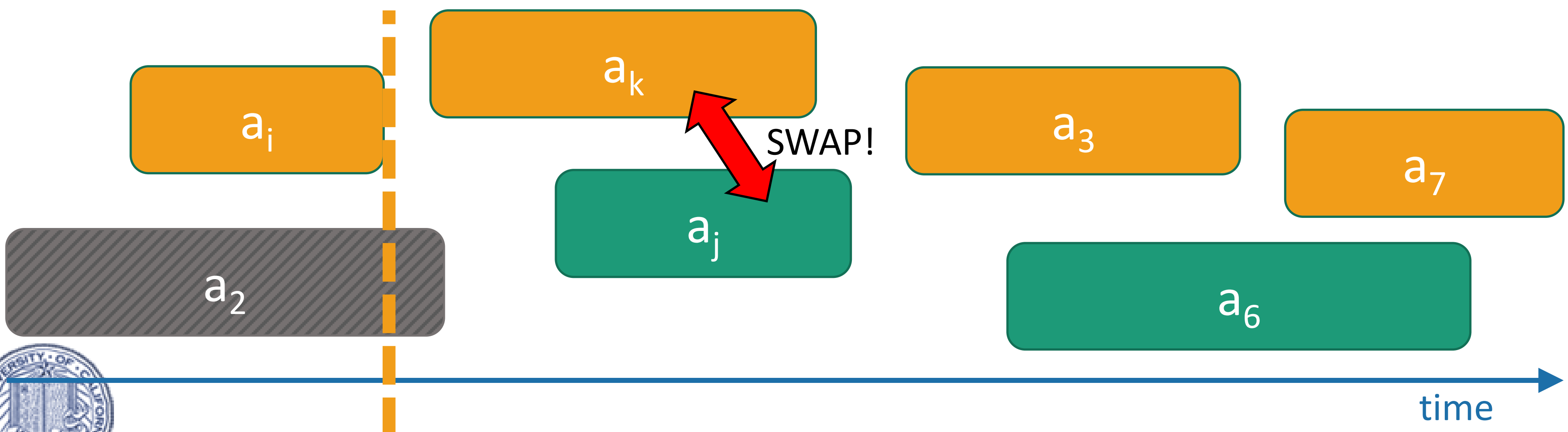
We never rule out an optimal solution ctd.

- If a_k is **not** in T*…
- Let a_j be the activity in T* with the smallest end time.
- Now consider schedule T you get by swapping a_j for a_k



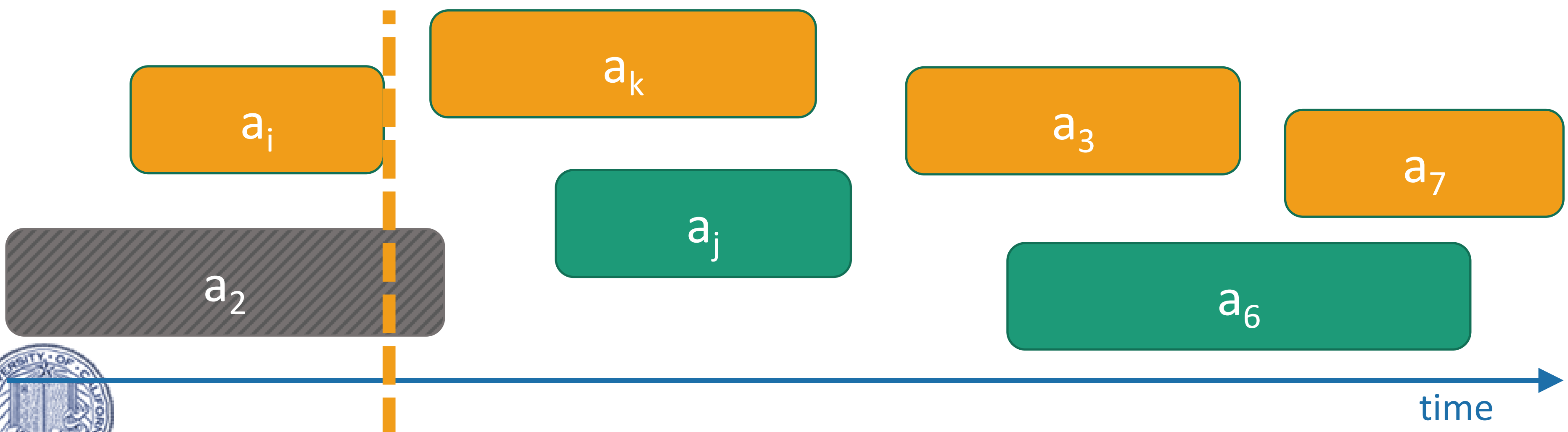
We never rule out an optimal solution
ctd.

- This schedule T is still allowed.
  - Since a_k has the smallest ending time, it ends before a_j.
  - Thus, a_k doesn't conflict with anything chosen after a_j.
- And, T is still optimal.
  - It has the same number of activities as T*.



We never rule out an optimal solution
ctd.

- We've just shown:
  - If there was an optimal solution that extends the choices we made so far…
  - …then there is an optimal schedule that also contains our next greedy choice a_k.



So the algorithm is correct

- We never rule out an optimal solution
- At the end of the algorithm, we've got some solution.
- So it must be optimal.

Lucky the Lackadaisical Lemur

So the algorithm is correct

Plucky the Pedantic Penguin

- Inductive Hypothesis:
  - After adding the t'th thing, there is an optimal solution that extends the current solution.
- Base case:
  - After adding zero activities, there is an optimal solution extending that.
- Inductive step:
  - **We just did that!**
- Conclusion:
  - After adding the last activity, there is an optimal solution that extends the current solution.
  - The current solution is the only solution that extends the current solution.
  - So the current solution is optimal.

Three Questions

1. Does this greedy algorithm for activity selection work?
   - Yes.
2. In general, when are greedy algorithms a good idea?
   - When the problem exhibits especially nice optimal substructure.
3. The "greedy" approach is often the first you'd think of…
   - Why are we getting to it now, in Week 12?
     - Proving that greedy algorithms work is often not so easy…

Common strategy
for greedy algorithms

- Make a **series of choices**.
- Show that, at each step, our choice **won't rule out an optimal solution** at the end of the day.
- After we've made all our choices, we haven't ruled out an optimal solution, **so we must have found one.**

Common strategy (formally)
for greedy algorithms

"Success" here means "finding an optimal solution."

- Inductive Hypothesis:
    - After greedy choice t, you haven't ruled out success.
- Base case:
    - Success is possible before you make any choices.
- Inductive step:
    - If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.
- Conclusion:
    - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

Common strategy

for showing we don’t rule out success

- Suppose that you’re on track to make an optimal solution T*.
  - Eg, after you’ve picked activity i, you’re still on track.
- Suppose that T* *disagrees* with your next greedy choice.
  - Eg, it *doesn’t* involve activity k.
- Manipulate T* in order to make a solution T that’s not worse but that *agrees* with your greedy choice.
  - Eg, swap whatever activity T* did pick next with activity k.

Three Questions

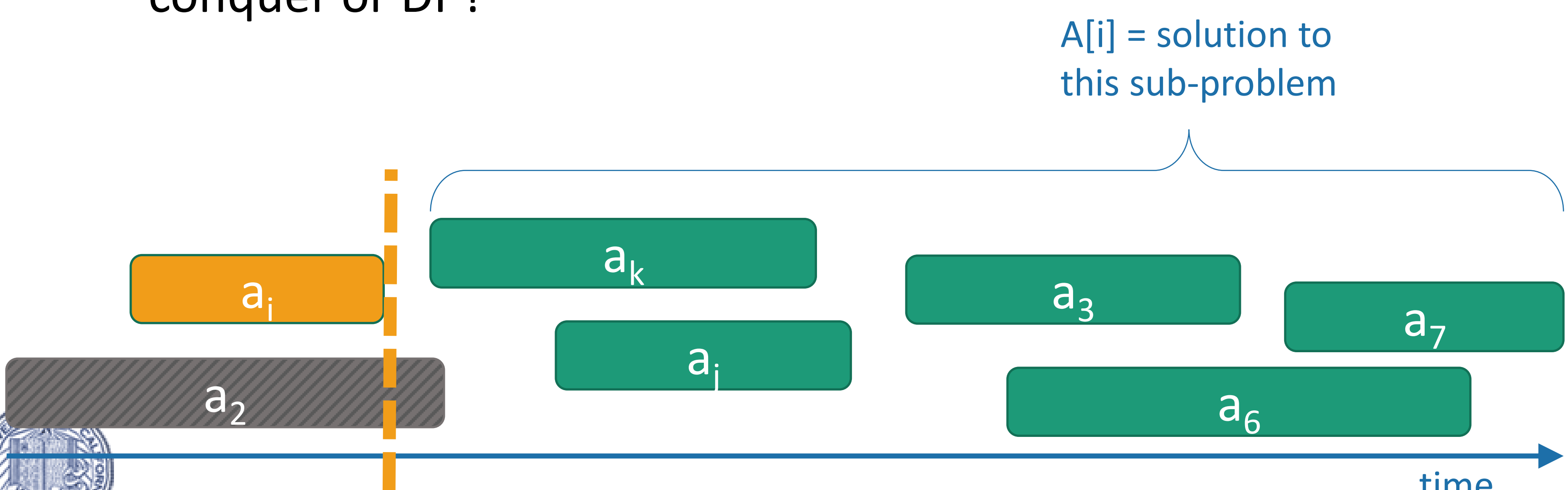
1. Does this greedy algorithm for activity selection work?
   - Yes.
2. In general, when are greedy algorithms a good idea?
   - When the problem exhibits especially nice optimal substructure.
3. The "greedy" approach is often the first you'd think of…
   - Why are we getting to it now, in Week 12?
     - Proving that greedy algorithms work is often not so easy…

Optimal sub-structure
in greedy algorithms

- Our greedy activity selection algorithm exploited a natural sub-problem structure:

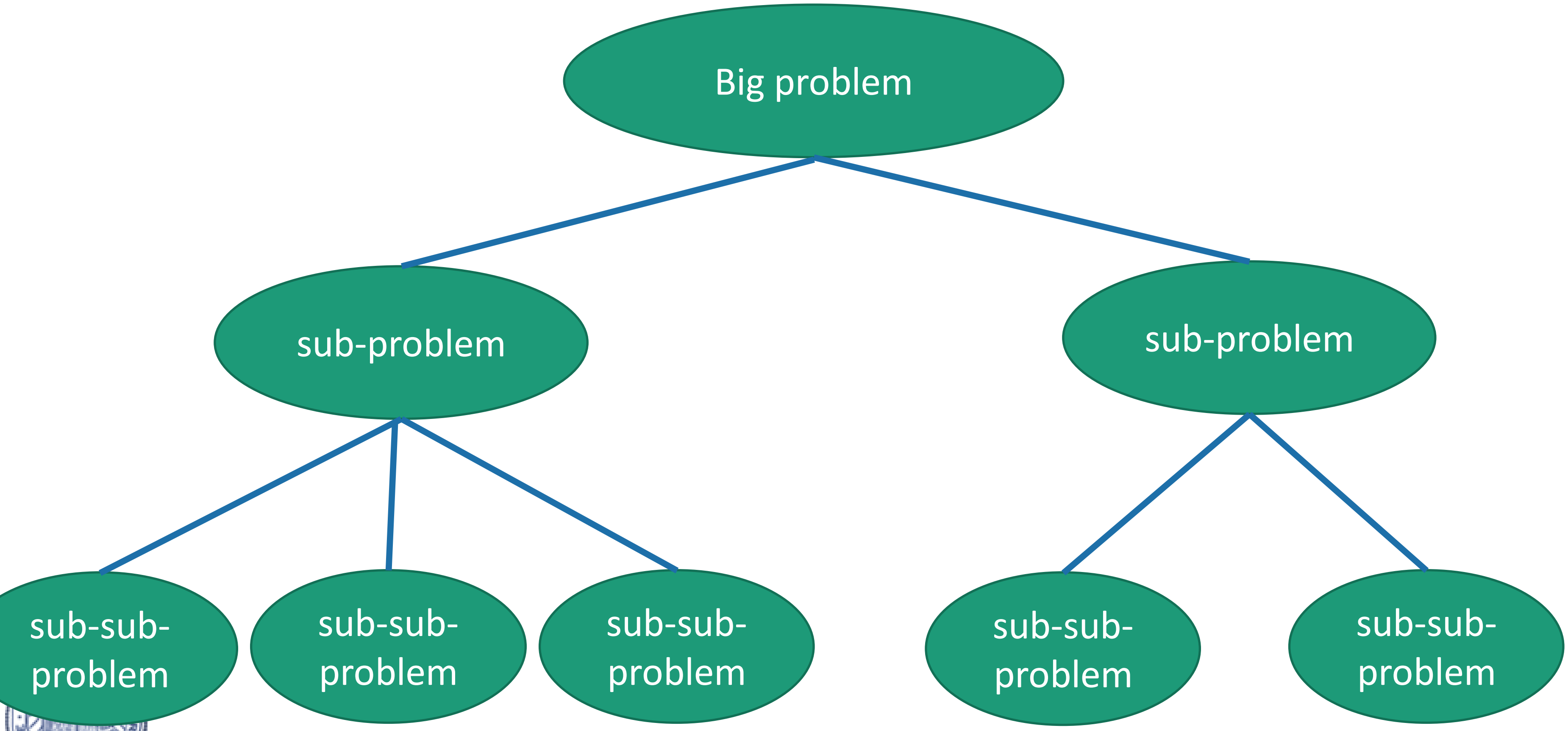
  A[i] = number of activities you can do after the end of activity i

- How does this substructure relate to that of divide-and-conquer or DP?



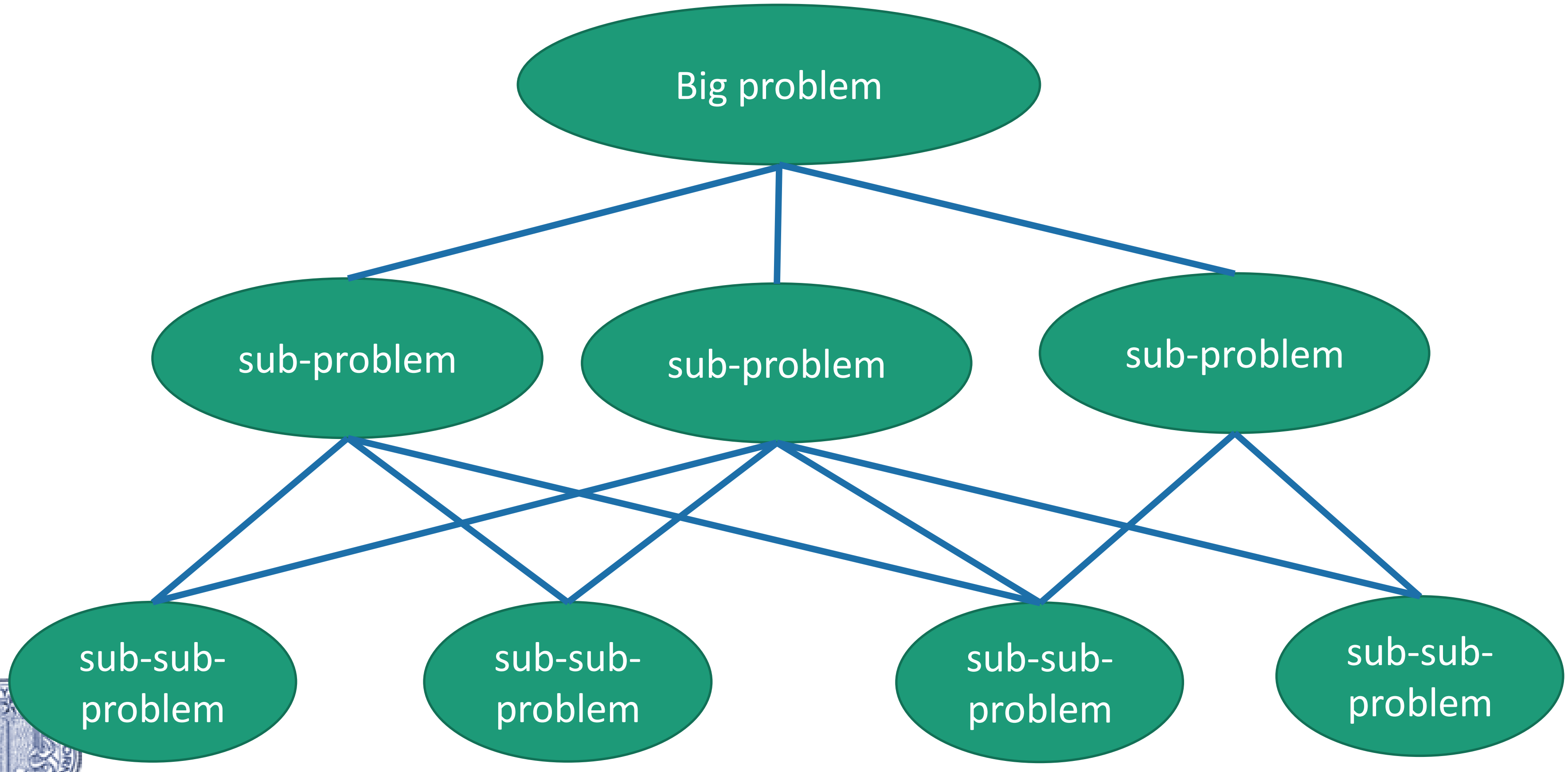
Sub-problem graph view

- Divide-and-conquer:



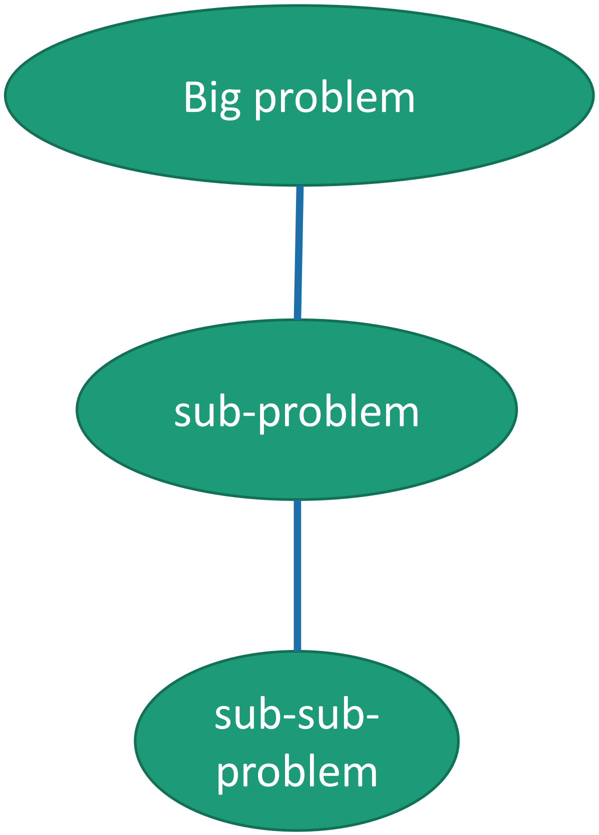
Sub-problem graph view

- Dynamic Programming:



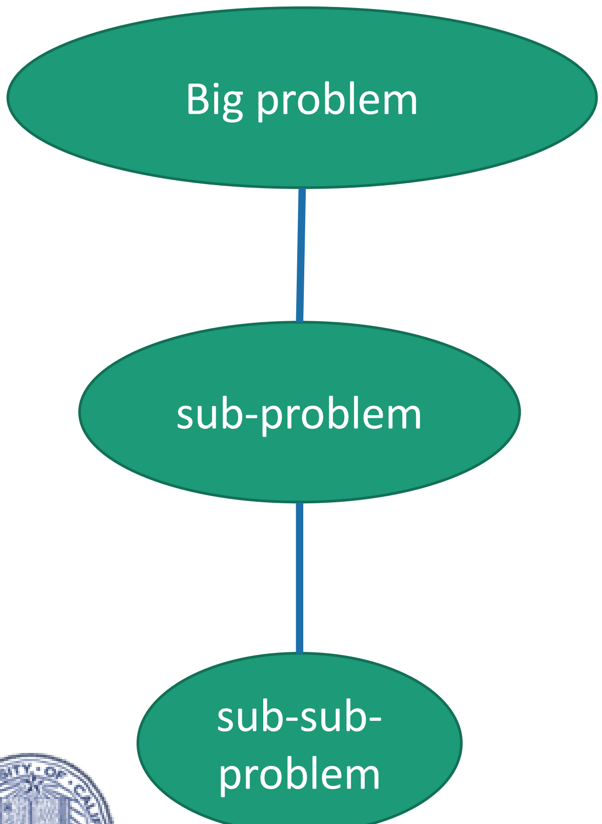
Sub-problem graph view

- Greedy algorithms:



Sub-problem graph view

- Greedy algorithms:



- Not only is there **optimal sub-structure:**
  - optimal solutions to a problem are made up from optimal solutions of sub-problems
- but each problem **depends on only one sub-problem**.

Write a DP version of activity selection (where you fill in a table)! [See next slides for one way to do it]

Ollie the Over-achieving Ostrich

More detail: Greedy vs. DP

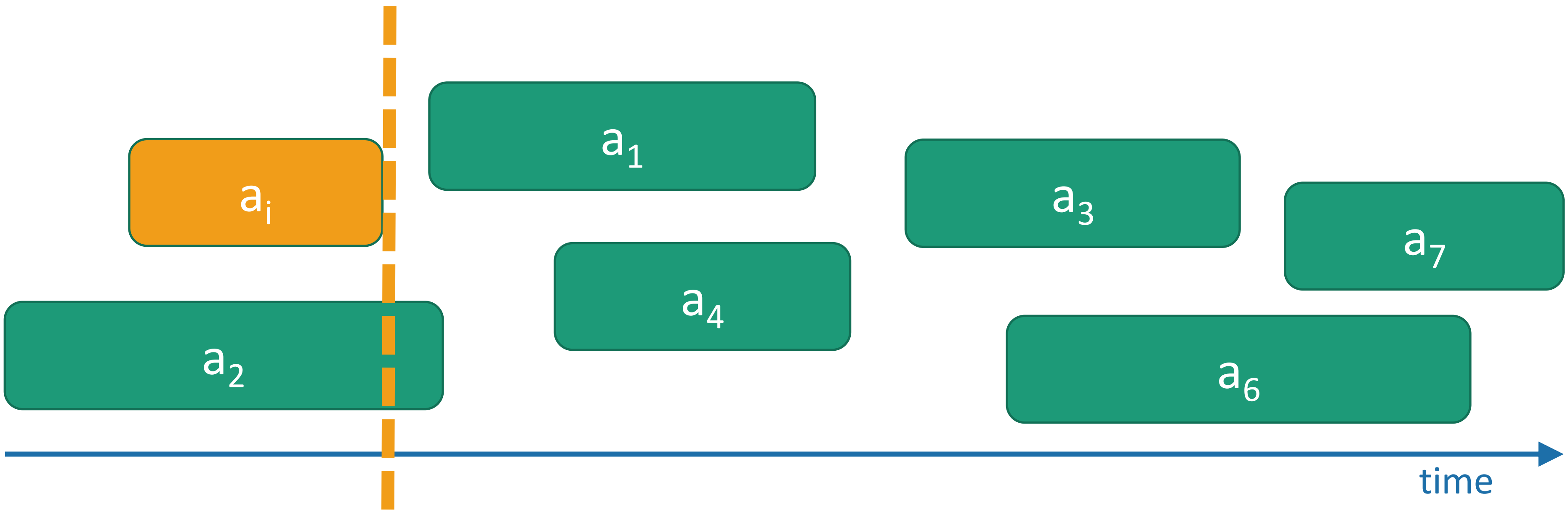
- Just for pedagogy!
- (This isn't the best way to think about activity selection).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Optimal substructure

- Subproblem i:
  - A[i] = number of activities you can do after Activity i finishes.

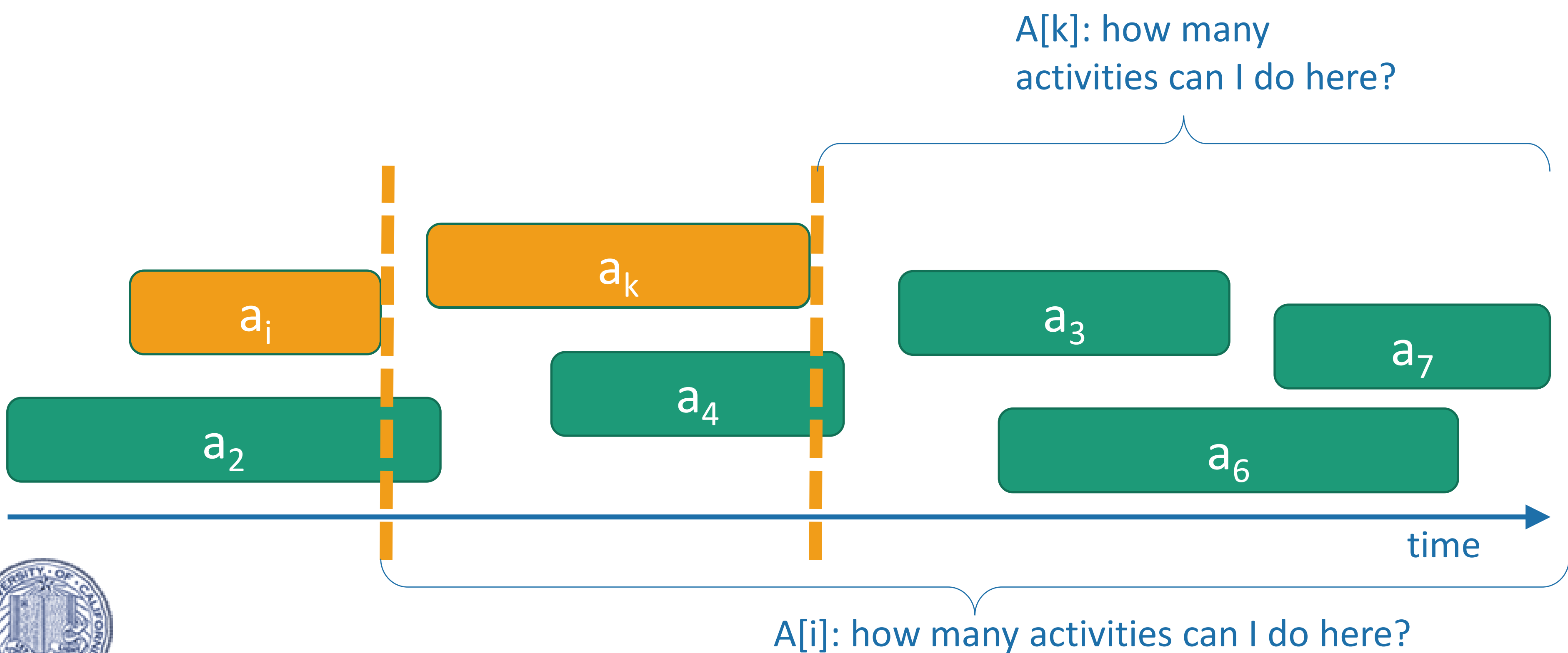


Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

We did that already

- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then A[i] = A[k] + 1.



Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Top-down DP

- Initialize a global array A to [None,…,None]
- Make a “dummy” activity that ends at time -1.
- **def** findNumActivities(i):
    - If A[i] != None:
        - **Return** A[i]
    - Let Activity k be the activity I can fit in my schedule after Activity i with the smallest finish time.
    - **If** there is no such activity k, set A[i] = 0
    - **Else,** A[i] = findNumActivities(k) + 1
    - **Return** A[i]
- **Return** findNumActivities(0)

This is a terrible way to write this!
The only thing that matters here is that the highlighted lines are our recursive relationship.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

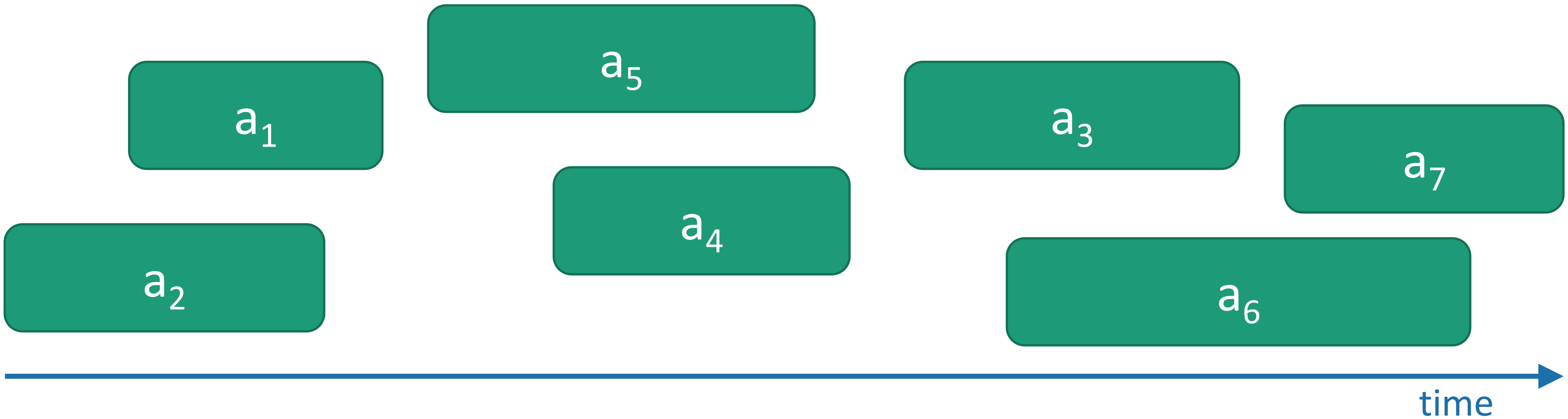
Top-down DP

- Initialize a global array A to [None,…,None]
- Initialize a global array Next to [None, …, None]
- Make a "dummy" activity that ends at time -1.
- **def** findNumActivities(i):
  - If A[i] != None:
    - **Return** A[i]
  - Let Activity k be the activity I can fit in my schedule after Activity i with the smallest finish time.
  - **If** there is no such activity k, set A[i] = 0
  - **Else,** A[i] = findNumActivities(k) + 1 and Next[i] = k
  - **Return** A[i]
- findNumActivities(0)
- Step through "Next" array to get schedule.

This is a terrible way to write this!

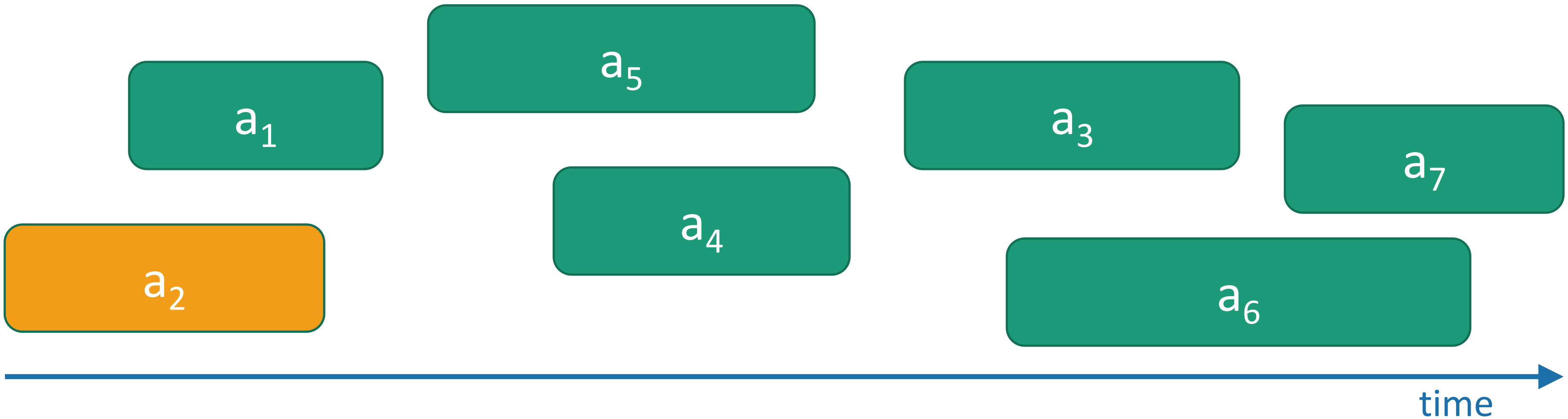
The only thing that matters here is that the highlighted lines are our recursive relationship.

Let's step through it.



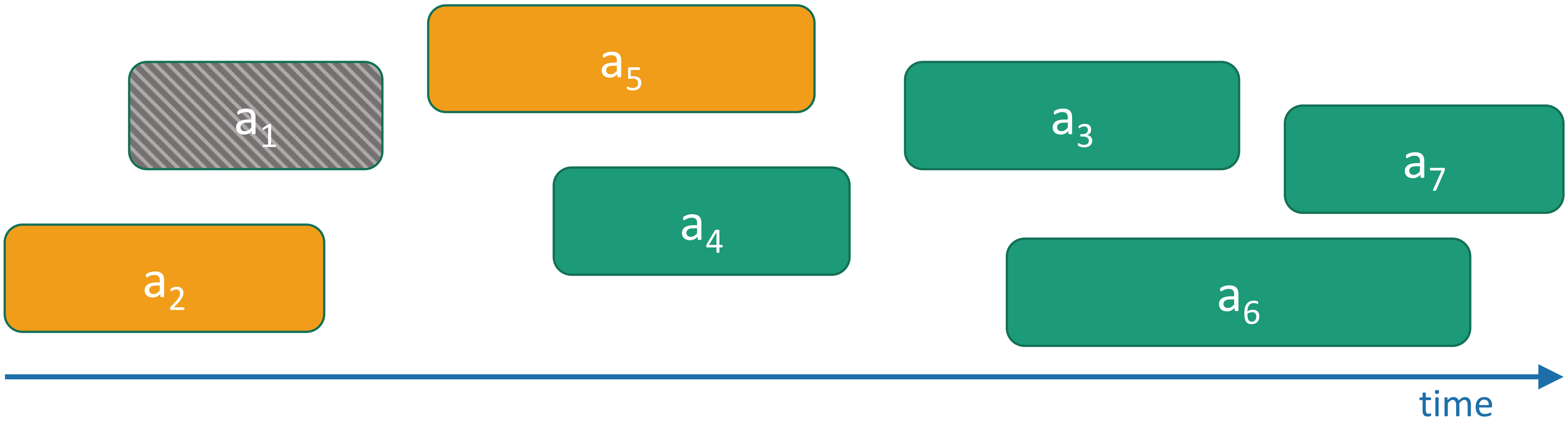
- Start with the activity with the smallest finish time.

Let's step through it



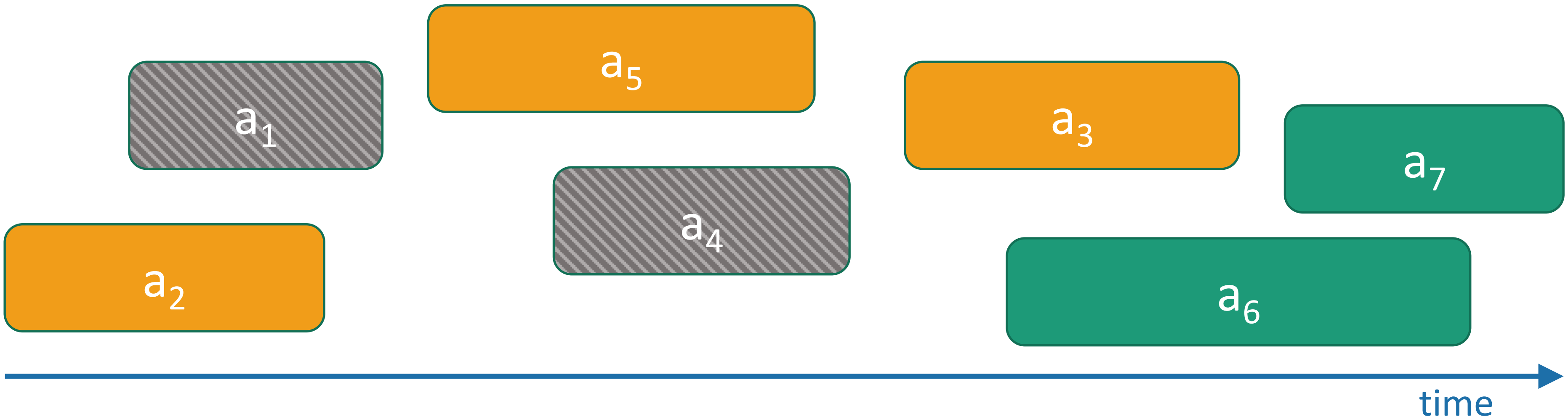
- Now find the next activity still do-able with the smallest finish time, and recurse after that.

Let’s step through it



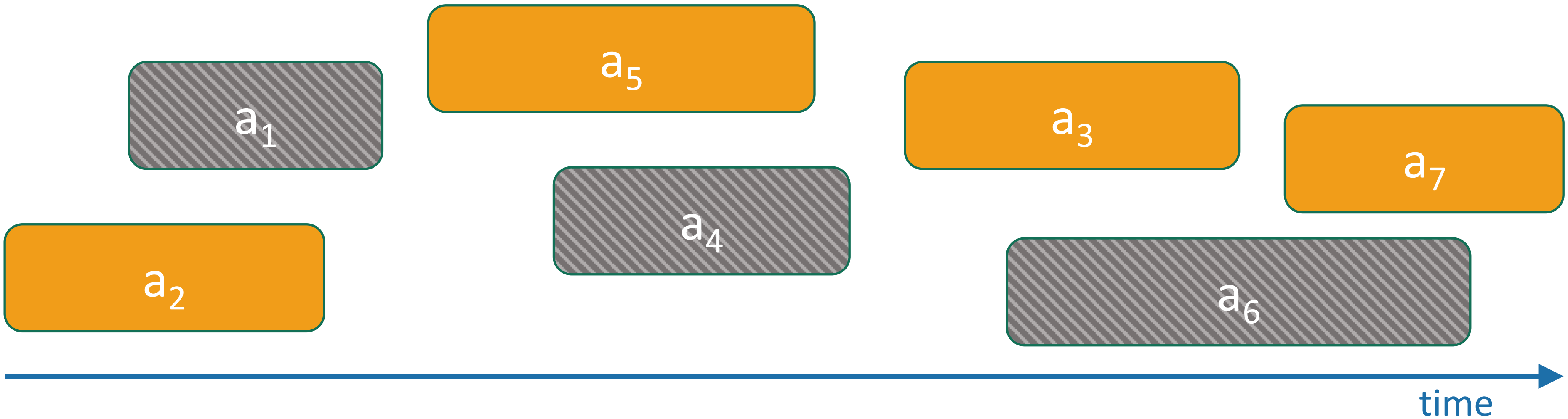
- Now find the next activity still do-able with the smallest finish time, and recurse after that.

Let's step through it



- Now find the next activity still do-able with the smallest finish time, and recurse after that.

Let’s step through it



- Ta-da!

It’s exactly the same* as the greedy solution!

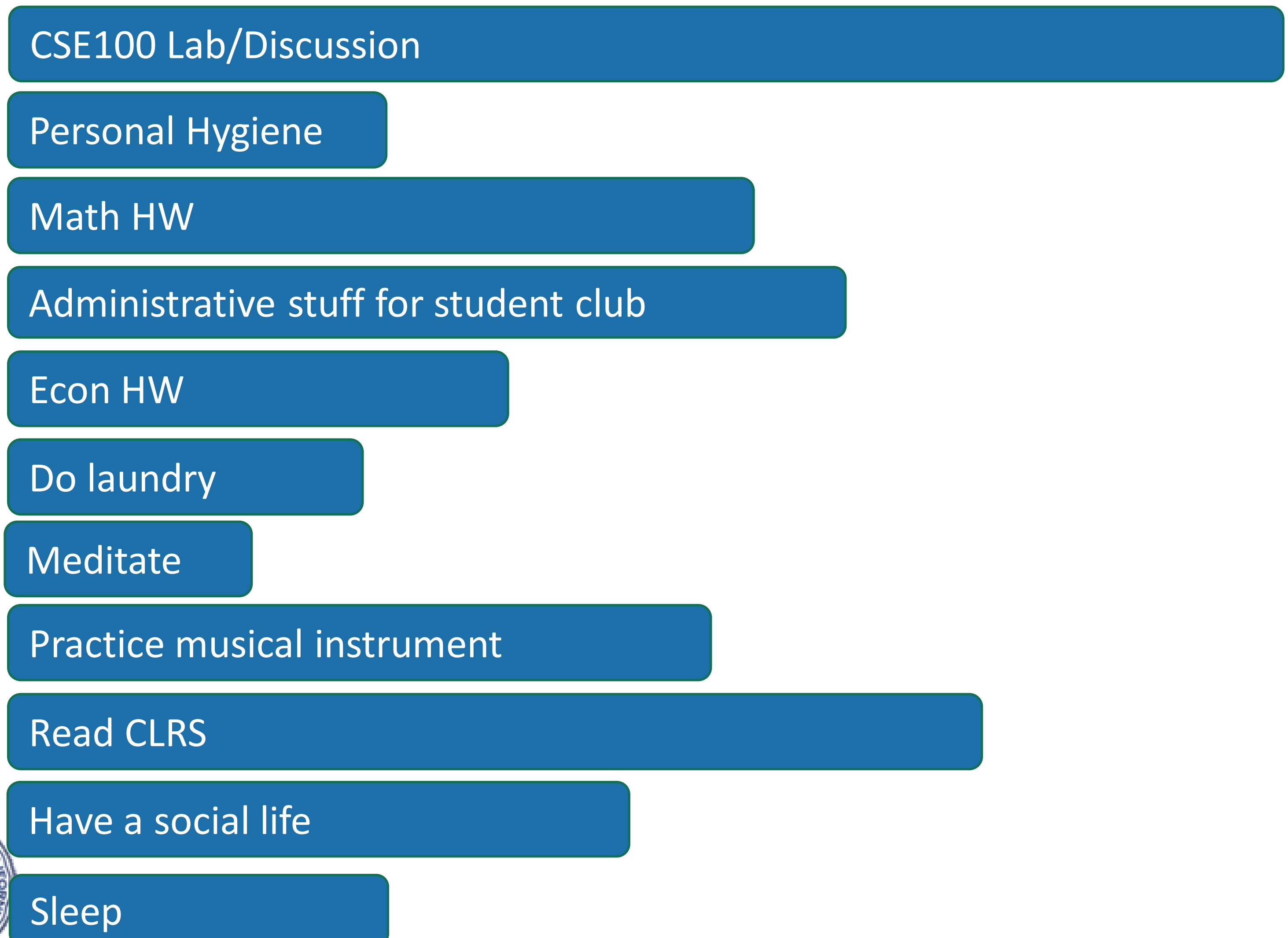
*if you implement the top-down DP solution appropriately.

Three Questions

1. Does this greedy algorithm for activity selection work?
    - Yes.
2. In general, when are greedy algorithms a good idea?
    - When they exhibit especially nice optimal substructure.
3. The "greedy" approach is often the first you'd think of…
    - Why are we getting to it now, in Week 10?
        - Proving that greedy algorithms work is often not so easy.

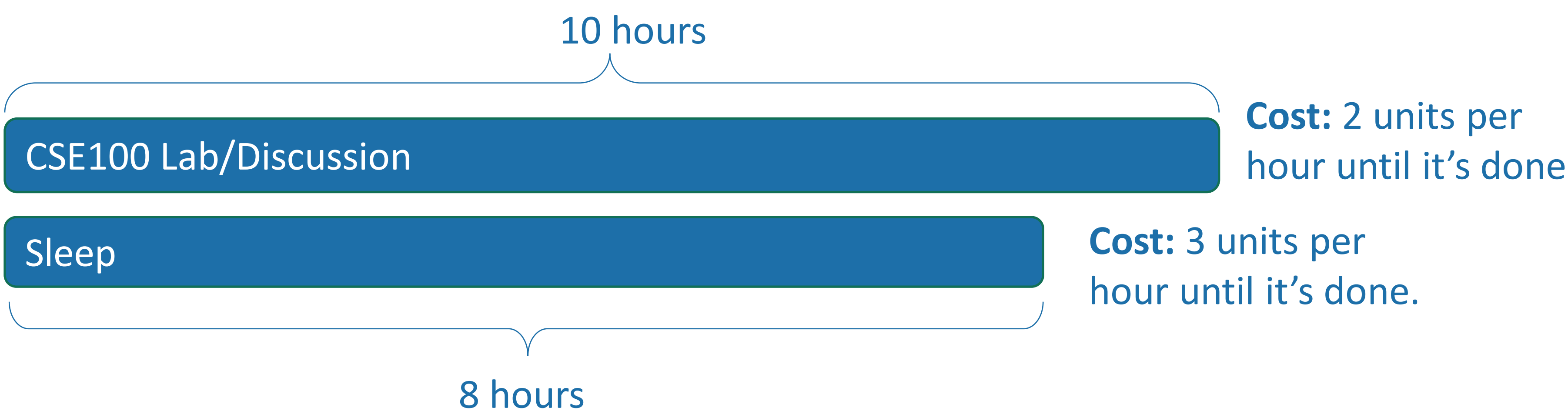
Let’s see a few more examples

Another example:
Scheduling



Scheduling

- n tasks
- Task i takes t_i hours
- For every hour that passes until task i is done, pay c_i



- CSE100 Lab, then Sleep: costs $\mathbf{10 \cdot 2 + (10 + 8) \cdot 3 = 74}$ **units**
- Sleep, then CSE100 Lab: costs $\mathbf{8 \cdot 3 + (10 + 8) \cdot 2 = 60}$ **units**

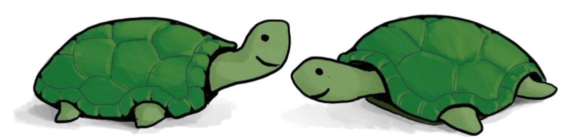
Optimal substructure

- This problem breaks up nicely into sub-problems:

Suppose this is the optimal schedule:

Then this must be the optimal schedule on just jobs B,C,D.

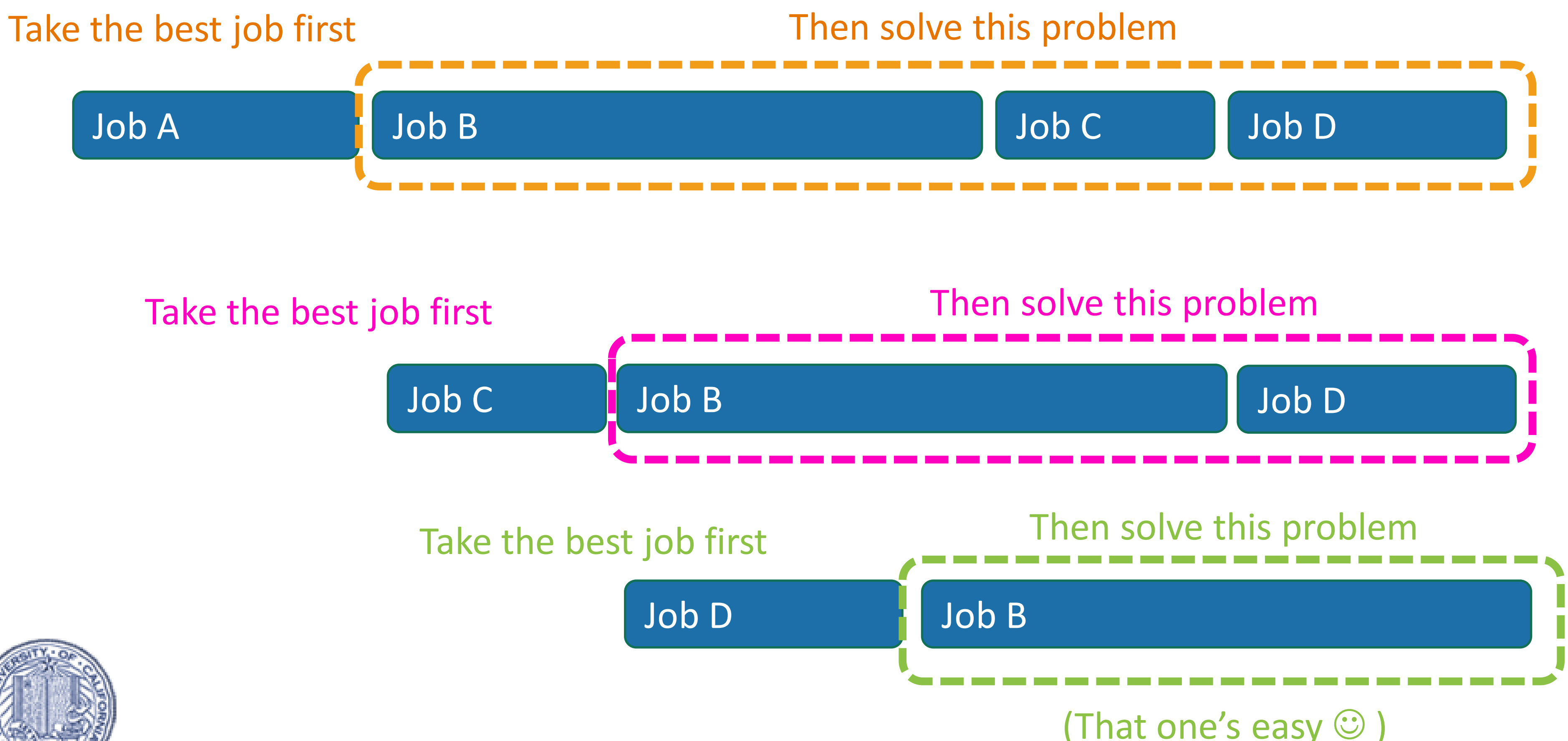
Why?



Think-pair-share

Optimal substructure

- Seems amenable to a greedy algorithm:

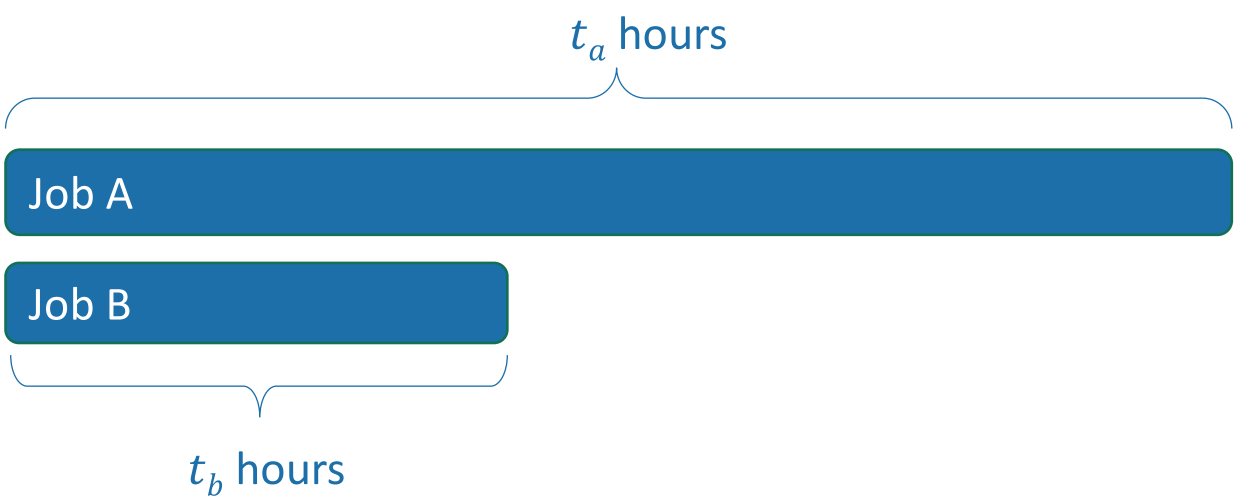


What does “best” mean?

AB is better than **BA** when:

$$t_a c_a + (t_a + t_b) c_b \leq t_b c_b + (t_a + t_b) c_a$$
$$t_a c_a + t_a c_b + t_b c_b \leq t_b c_b + t_a c_a + t_b c_a$$
$$t_a c_b \leq t_b c_a$$
$$\frac{c_b}{t_b} \leq \frac{c_a}{t_a}$$

- Of these two jobs, which should we do first?



Cost: $\boldsymbol{c_a}$ units per hour until it’s done.

Cost: $\boldsymbol{c_b}$ units per hour until it’s done.

What matters is the ratio:

$$\frac{\textbf{cost of delay}}{\textbf{time it takes}}$$

“Best” means biggest ratio.

- Cost(**A then B**) = $t_a c_a + (t_a + t_b) c_b$
- Cost(**B then A**) = $t_b c_b + (t_a + tb)ca$

Idea for greedy algorithm

- Choose the job with the biggest $\frac{\text{cost of delay}}{\text{time it takes}}$ ratio.

Lemma

This greedy choice doesn't rule out success

- Suppose you have already chosen some jobs, and haven't yet ruled out success:

Already chosen E

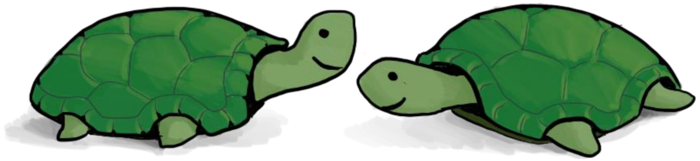
There's some way to order A, B,C, D that's optimal…

Job E | Job C | Job A | Job B | Job D

Say greedy chooses job B

- Then if you choose the next job to be the one left that maximizes the ratio **cost/time**, you still won't rule out success.
- **Proof sketch:**
  - Say Job B maximizes this ratio, but it's not the next job in the opt. soln.

How can we manipulate the optimal solution above to make an optimal solution where B is the next job we choose?



Lemma

This greedy choice doesn't rule out success

- Suppose you have already chosen some jobs, and haven't yet ruled out success:

Already chosen E

There's some way to order A, B,C, D that's optimal…

Job E | Job C | Job A | Job B | Job D

Say greedy chooses job B

- Then if you choose the next job to be the one left that maximizes the ratio **cost/time**, you still won't rule out success.
- **Proof sketch:**
  - Say Job B maximizes this ratio, but it's not the next job in the opt. soln.
  - Switch A and B! Nothing else will change, and we just showed that the cost of the solution won't increase.

Job E | Job C | Job B | Job A | Job D

  - Repeat until B is first.

Job E | Job B | Job C | Job A | Job D

  - Now this is an optimal schedule where B is first.

Back to our framework for proving correctness of greedy algorithms

- Inductive Hypothesis:
  - After greedy choice t, you haven’t ruled out success.
- Base case:
  - Success is possible before you make any choices.
- Inductive step:
  - If you haven’t ruled out success after choice t, then you won’t rule out success after choice t+1.
- Conclusion:
  - If you reach the end of the algorithm and haven’t ruled out success then you must have succeeded.

Just did the inductive step!

Fill in the details!

Greedy Scheduling Solution

- **scheduleJobs**(JOBS):
  - Sort JOBS in decreasing order by the ratio:
    - $r_i = \frac{c_i}{t_i} = \frac{\text{cost of delaying job i}}{\text{time job i takes to complete}}$
  - **Return** JOBS

Running time: O(nlog(n))

Now you can go about your schedule peacefully, in the optimal way.

Formally, use induction!

SLIDE SKIPPED IN CLASS

- **Inductive hypothesis**:
  - There is an optimal ordering so that the first t jobs are **sorted_JOBS**[:t].
- **Base case**:
  - When t=0, this reads: "There is an optimal ordering so that the first 0 jobs are []"
  - That's true.
- **Inductive Step:**
  - Boils down to: there is an optimal ordering on sorted_JOBS[t:] so that sorted_JOBS[t] is first.
  - This follows from the Lemma.
- **Conclusion:**
  - When t=n, this reads: "There is an optimal ordering so that the first n jobs are **sorted_JOBS**."
  - aka, what we returned is an optimal ordering.

What have we learned?

- A greedy algorithm works for scheduling

- This followed the same outline as the previous example:
  - Identify **optimal substructure:**

  - Find a way to make choices that **won't rule out an optimal solution.**
    - largest cost/time ratios first.