طراحی الگوریتم ها (CE221)

جلسه پانزدهم:
روش های حریصانه

سجاد شیرعلی شهرضا

بهار 1401

دوشنبه، 5 اردیبهشت 1401

Based on Stanford CS161 Slides from Summer 2020 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 16
- نظرسنجی سوم
  - مهلت: ساعت 8 صبح دوشنبه 12 اردیبهشت 1401 (هفته آینده، روز عید فطر!)

روش حریصانه

الگوریتم حریصانه چیست؟

THE GREEDY PARADIGM

Commit to choices one-at-a-time,

never look back,

and hope for the best.

THE GREEDY PARADIGM

Commit to choices one-at-a-time,

never look back,

and hope for the best.

Greedy doesn't always work.

We'll see some non-examples where a tempting greedy approach won't work.
Then, we'll see some examples where a greedy solution exists!

THE GREEDY PARADIGM

DISCLAIMER: It's often surprisingly easy to come up with ideas for greedy algorithms, they're usually pretty easy to write down, and their runtimes are straightforward to analyze! But you'll end up wondering, "how am I supposed to know *when* I can use greedy algorithms?" The answer may not be satisfying: *a lot of the times, greedy algorithms are not correct, and whenever they are correct, it can be difficult to prove its correctness.* This aspect of greedy algorithms is why we've waited until the end of class to discuss this design paradigm!

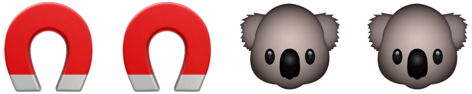
NON-EXAMPLE: GREEDY KNAPSACK?

Can we design a greedy algorithm for Unbounded Knapsack?

UNBOUNDED KNAPSACK

We have infinite copies of all the items.

What's the most valuable way to fill the knapsack?



Total weight: **2 + 2 + 3 + 3 = 10**

Total value: **8 + 8 + 13 + 13 = 42**

Capacity: **10**

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

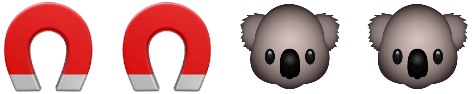
NON-EXAMPLE: GREEDY KNAPSACK?

Can we design a greedy algorithm for Unbounded Knapsack?

UNBOUNDED KNAPSACK

We have infinite copies of all the items.

What's the most valuable way to fill the knapsack?



Total weight: **2 + 2 + 3 + 3 = 10**

Total value: **8 + 8 + 13 + 13 = 42**

Capacity: **10**

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

Greedy approach? Here's an idea: koalas have the best value/weight ratio, so keep using koalas!

Total weight: **3 + 3 + 3 = 9**

Total value: **13 + 13 + 13 = 39**

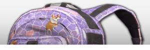
NON-EXAMPLE: GREEDY KNAPSACK?

Can we design a greedy algorithm for Unbounded Knapsack?

UNBOUNDED KNAPSACK

We have inf

What's the mos



Total w

Total va

0

3

13

11

35

This doesn't work! We ended up "regretting" our greedy choices. By the time we put in the third koala, we realized that a magnet would have been better (even though it doesn't immediately seem as valuable at the time) because it would have left enough space for a fourth object that could bump up our overall value!

Greedy approach? Here's an idea: koalas have the best value/weight ratio, so keep using koalas!

Total weight: **3 + 3 + 3 = 9**

Total value: **13 + 13 + 13 = 39**

NON-EXAMPLE: GREEDY KNAPSACK?

Our greedy approach for Unbounded Knapsack doesn't work for all inputs (and we showed it fails via a counterexample)

NON-EXAMPLE: GREEDY KNAPSACK?

Our greedy approach for Unbounded Knapsack doesn't work for all inputs (and we showed it fails via a counterexample)

While we usually don't say "No greedy algorithm can work", you can often get an idea of whether a **nearsighted style of greedy decision making** feels suitable for a problem by going through a few attempts at designing a greedy solution.

NON-EXAMPLE: GREEDY KNAPSACK?

Our greedy approach for Unbounded Knapsack doesn't work for all inputs (and we showed it fails via a counterexample)

While we usually don't say "No greedy algorithm can work", you can often get an idea of whether a **nearsighted style of greedy decision making** feels suitable for a problem by going through a few attempts at designing a greedy solution.

In this Unbounded Knapsack attempt, we saw that making the nearsighted decision of putting in the highest value/weight ratio object that can fit at the time will cause us to have "regret" later down in the road. Making a nearsighted greedy decision feels inappropriate in this problem, since it might be better to give up something earlier on to make room for optimal decisions later. That's why DP made more sense for Unbounded Knapsack: DP tries to optimize its choice by seeing a decision all the way through (via recursive formulations) and *then* picking the most optimal choice

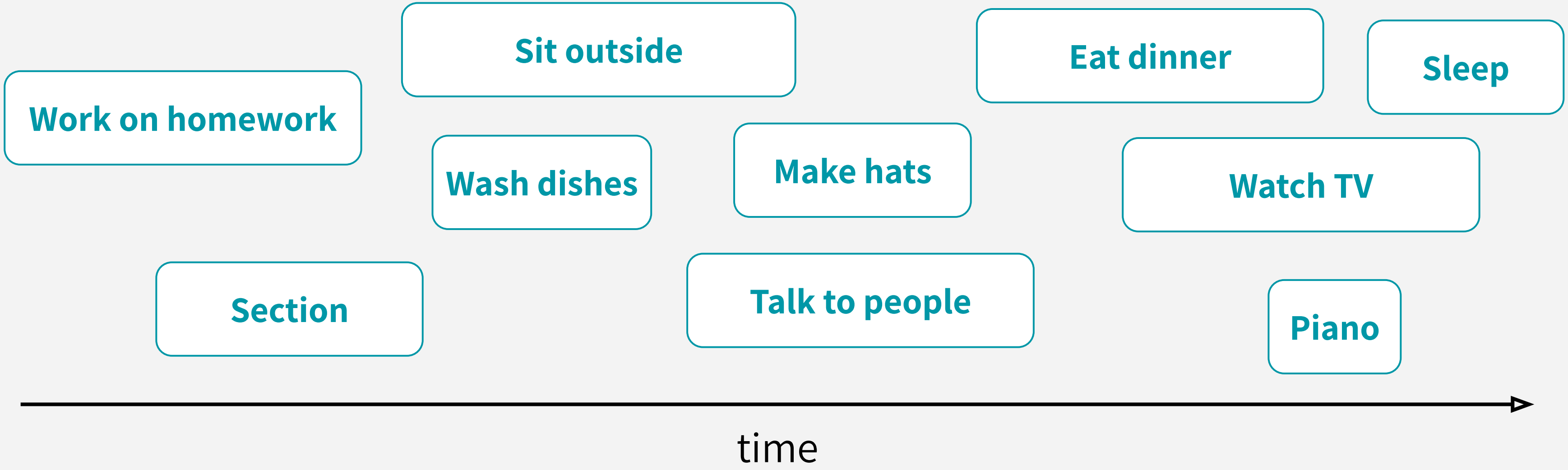
سوال؟

انتخاب فعالیت

یک مثال از جایی که روش حریصانه کار می کند!

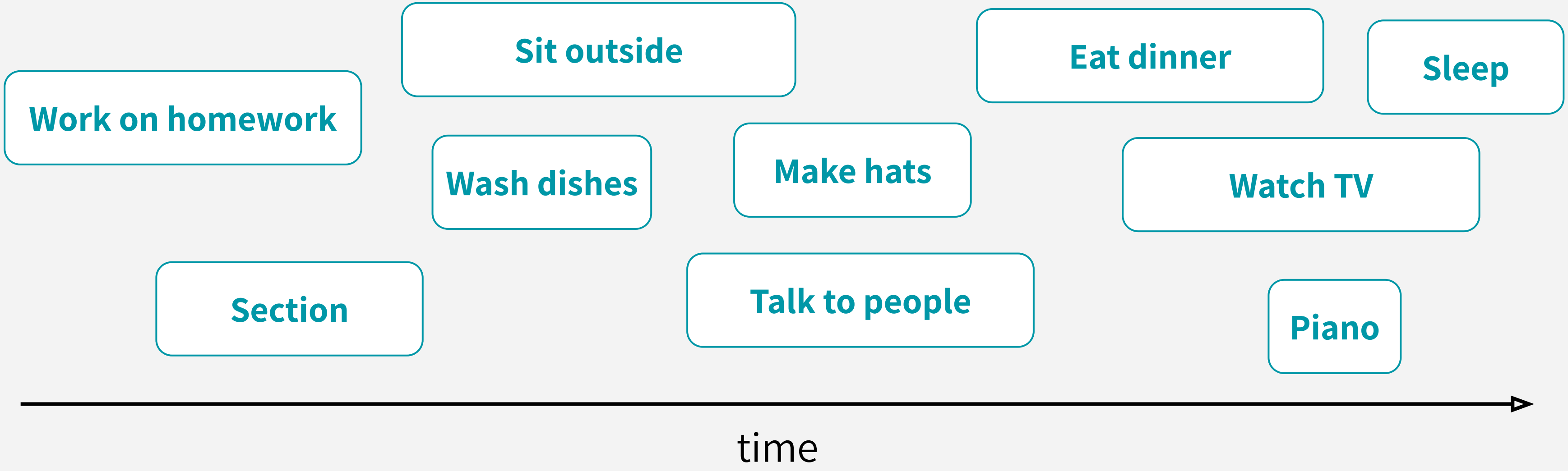
ACTIVITY SELECTION: THE TASK

Input: n activities with start times and finish times
Constraint: All activities are equally important, but you can only do 1 activity at a time!
Output: A way to maximize the number of activities you can do



ACTIVITY SELECTION: THE TASK

In what order should you greedily add activities? Here are 3 ideas:



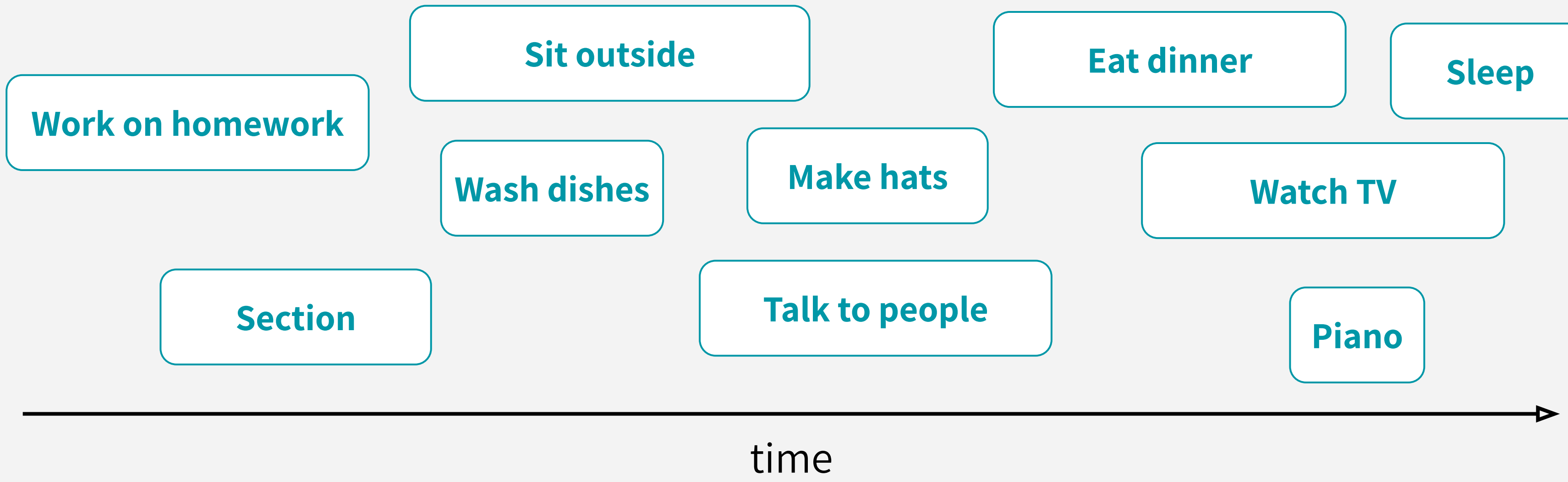
ACTIVITY SELECTION: THE TASK

In what order should you greedily add activities? Here are 3 ideas:

1) Be impulsive: choose activities in ascending order of start times

2) Avoid commitment: choose activities in ascending order of length

3) Finish fast: choose activities in ascending order of end times



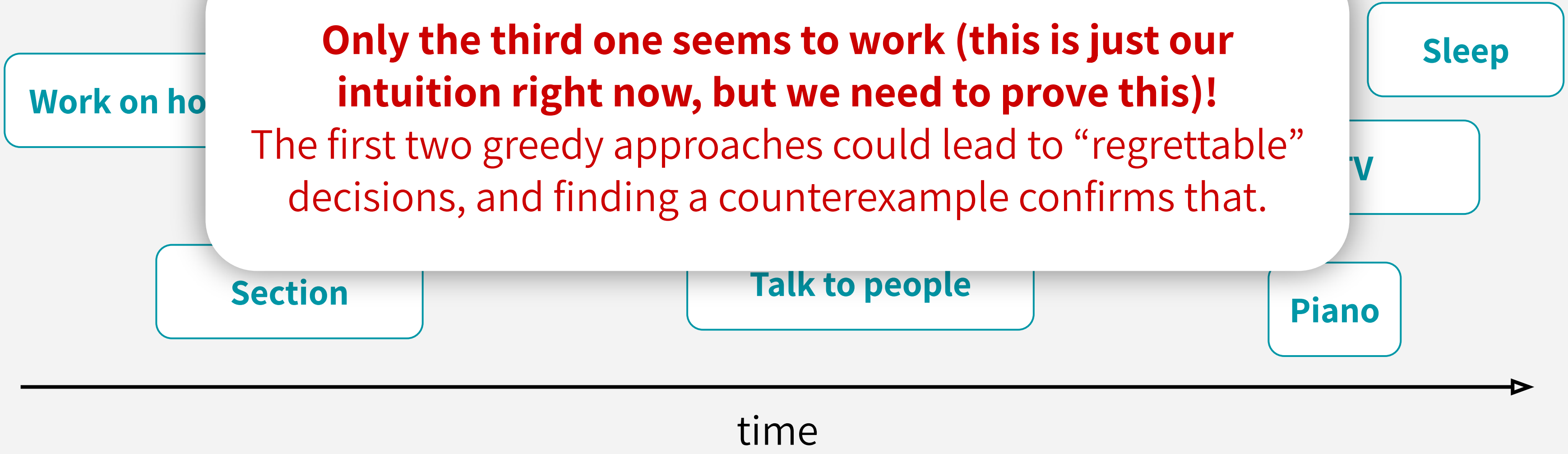
ACTIVITY SELECTION: THE TASK

In what order should you greedily add activities? Here are 3 ideas:

1) Be impulsive: choose activities in ascending order of start times

2) Avoid commitment: choose activities in ascending order of length

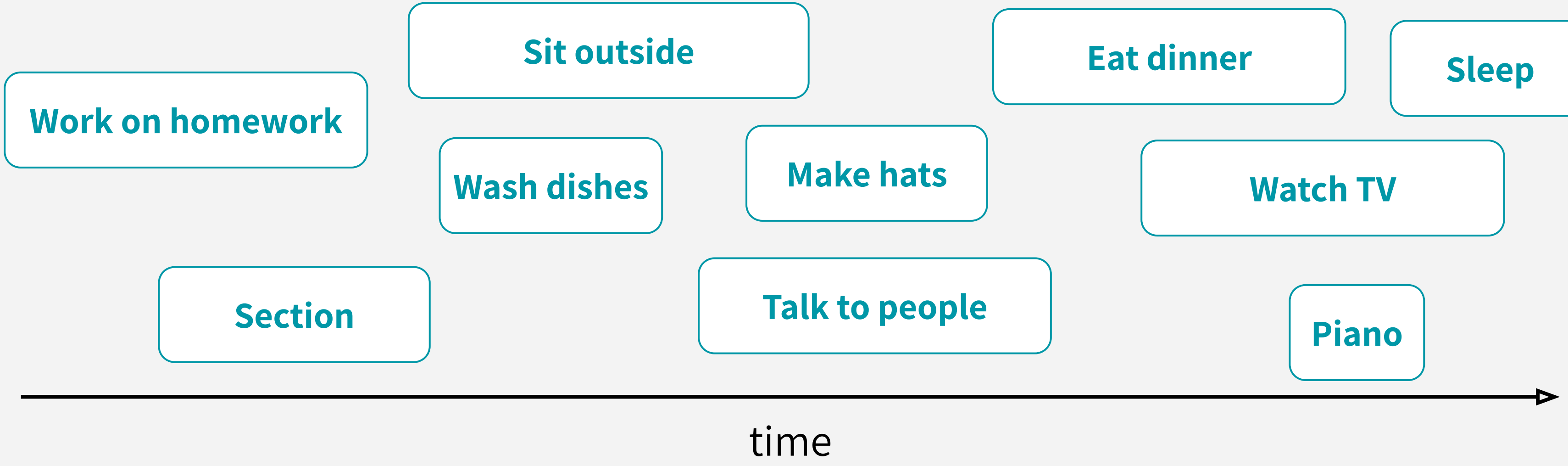
3) Finish fast: choose activities in ascending order of end times



Only the third one seems to work (this is just our intuition right now, but we need to prove this)!
The first two greedy approaches could lead to "regrettable" decisions, and finding a counterexample confirms that.

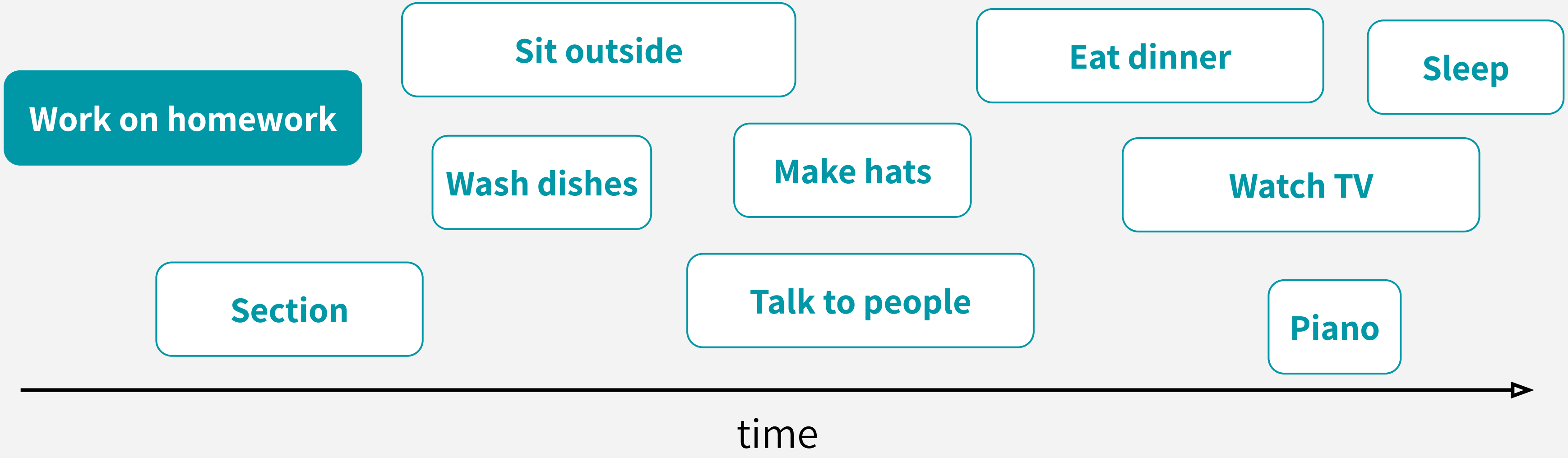
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



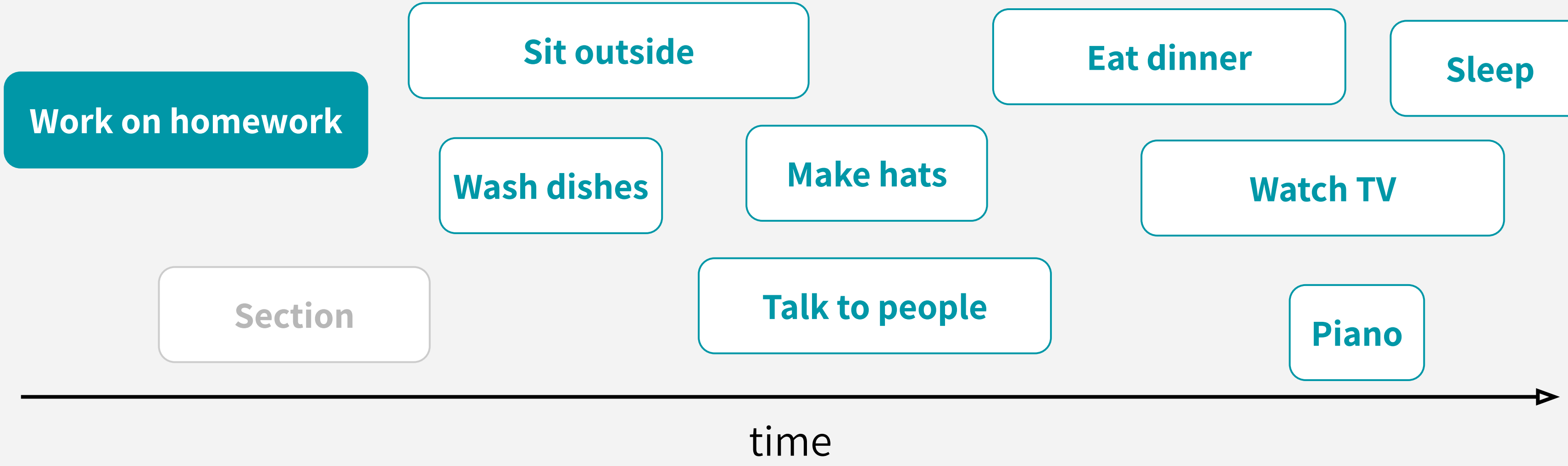
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



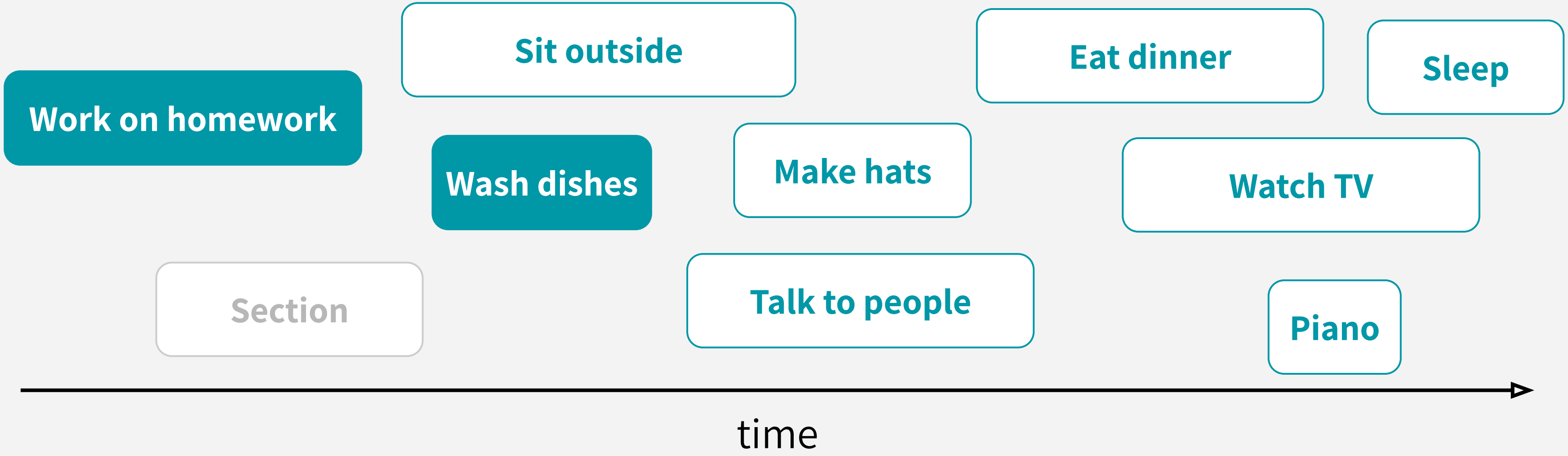
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



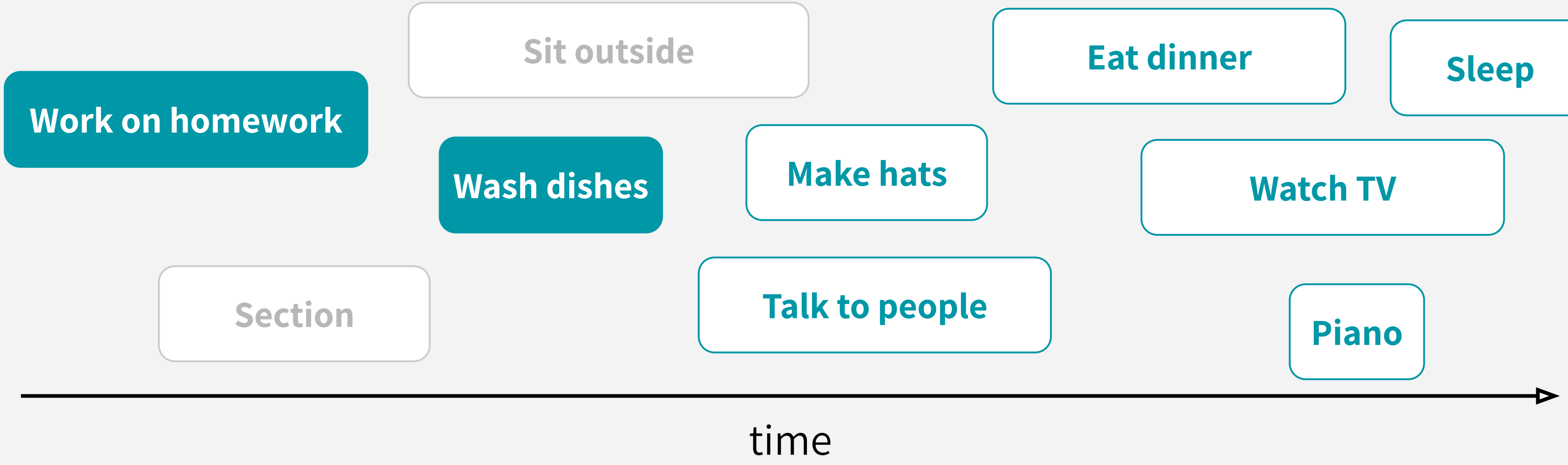
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat

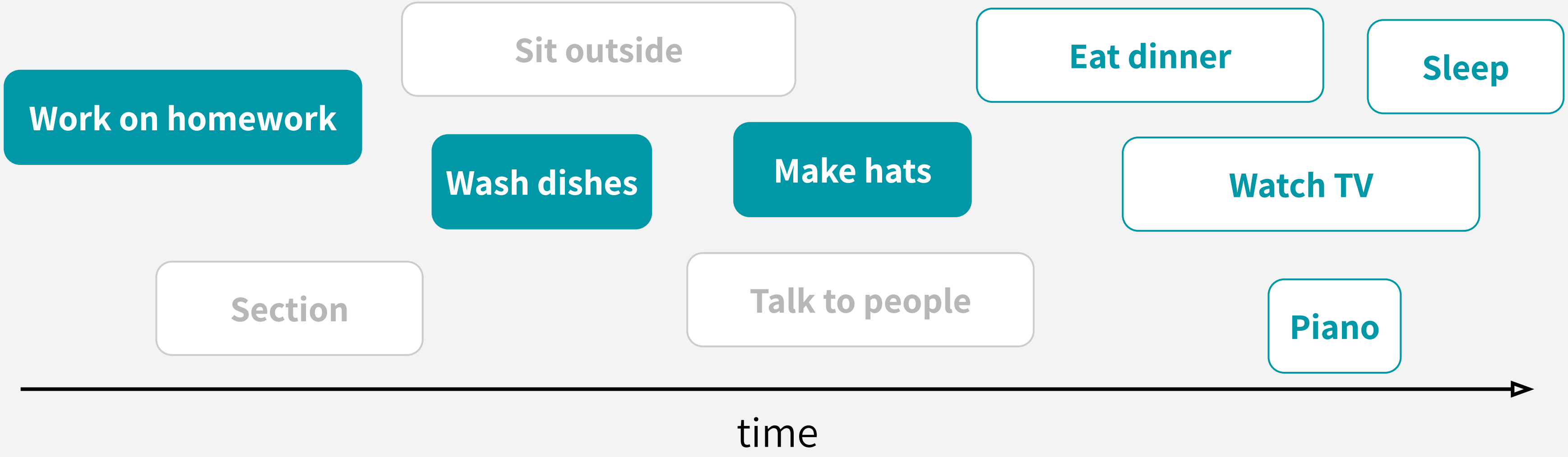


OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat

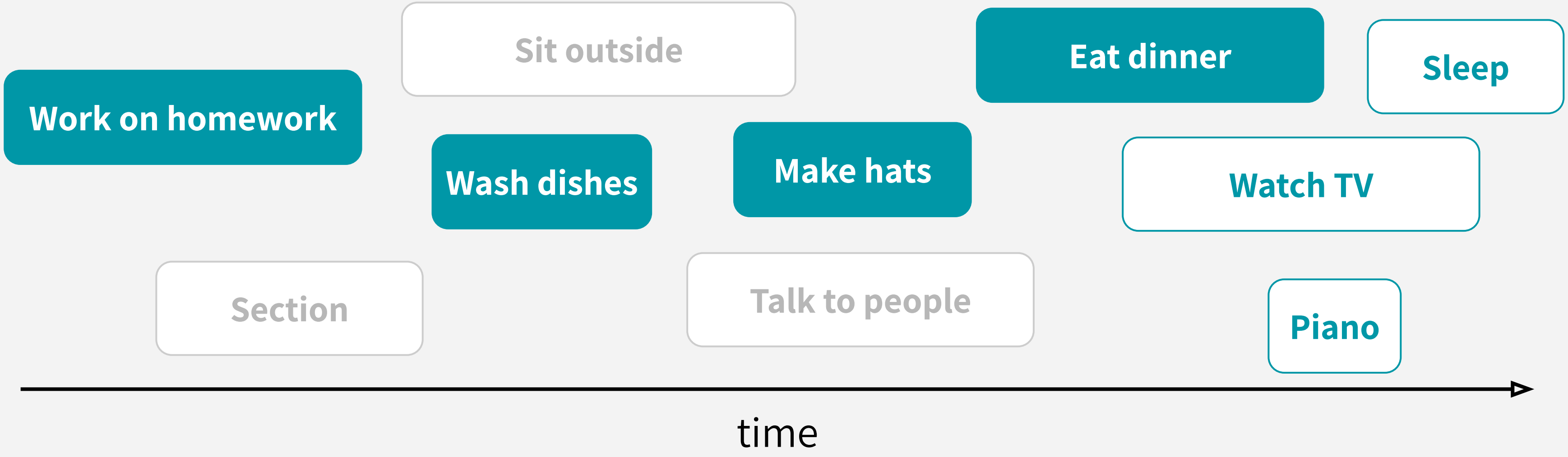
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



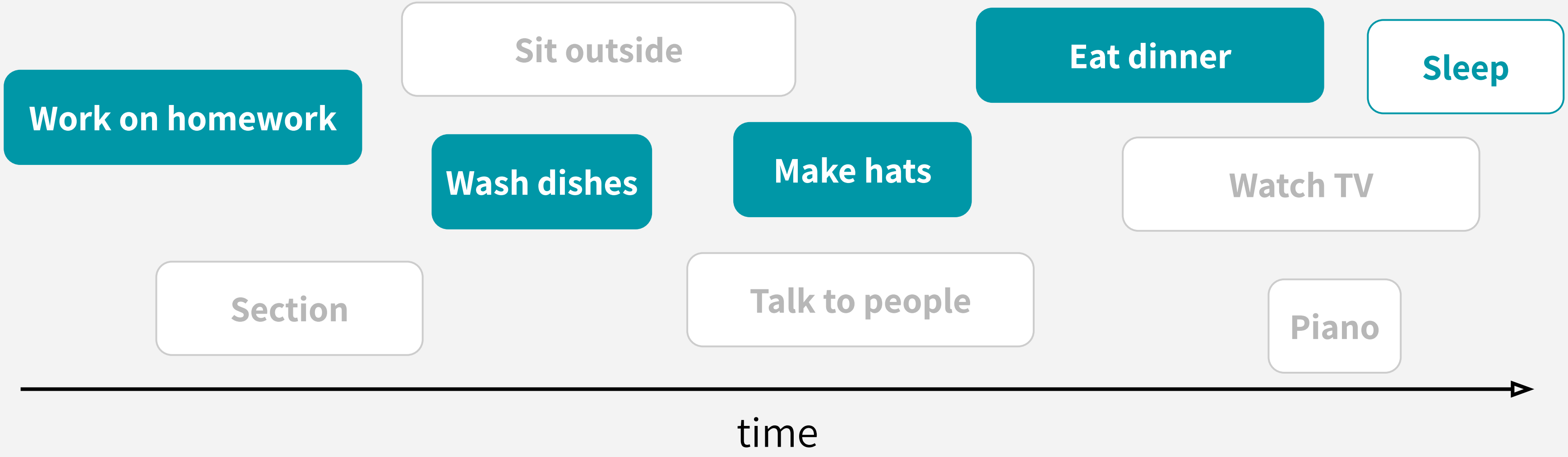
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



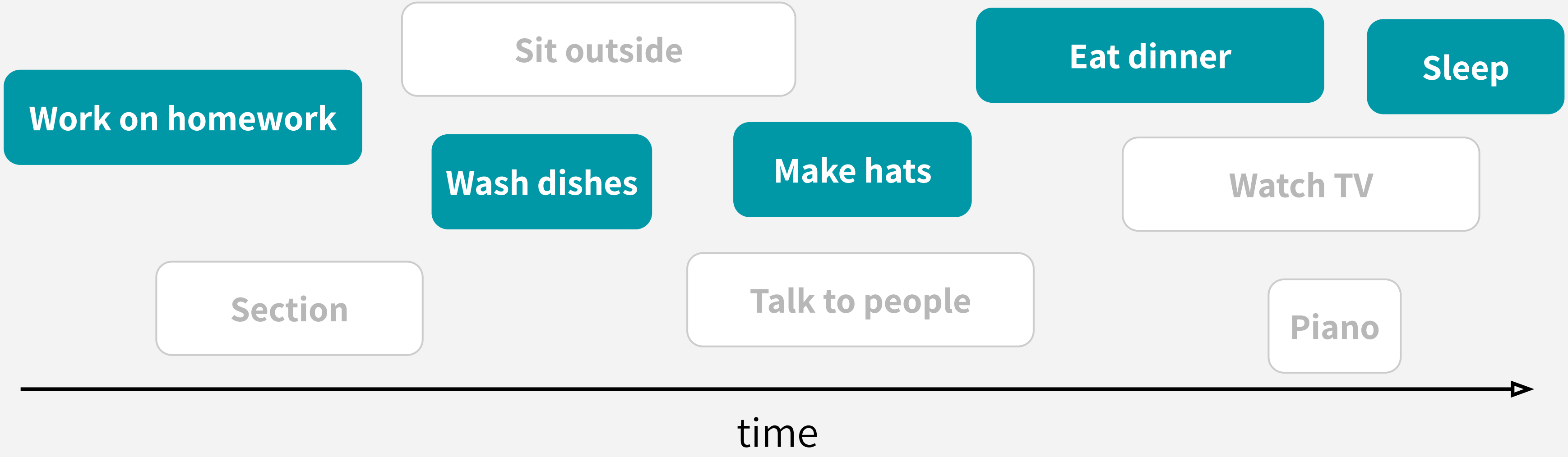
OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



OUR GREEDY ALGORITHM

Pick an available activity with the smallest finish time & repeat



ACTIVITY SELECTION: PSEUDOCODE

```
ACTIVITY_SELECTION(activities A with start and finish times):
    A = MERGESORT_BY_FINISHTIMES(A)
    result = {}
    busy_until = 0
    for a in A:
        if a.start >= busy_until:
            result.add(a)
            busy_until = a.finish
    return result
```

ACTIVITY SELECTION: PSEUDOCODE

```
ACTIVITY_SELECTION(activities A with start and finish times):
    A = MERGESORT_BY_FINISHTIMES(A)
    result = {}
    busy_until = 0
    for a in A:
        if a.start >= busy_until:
            result.add(a)
            busy_until = a.finish
    return result
```

Runtime: O(n log n)

سوال؟

WHY IS IT GREEDY?

What makes our algorithm a **greedy** algorithm?

At each step in the algorithm, we make a choice (pick the available activity with the smallest finish time) and never look back.

WHY IS IT GREEDY?

What makes our algorithm a **greedy** algorithm?

At each step in the algorithm, we make a choice (pick the available activity with the smallest finish time) and never look back.

How do we know that this greedy algorithm is correct?
(Proving correctness is the hard part!)

THE BIG IDEA:
Whenever we make a choice, we don't rule out an optimal solution.

ACTIVITY SELECTION: CORRECTNESS

We want to prove that the algorithm finds an optimal set of activities
(i.e. there isn't a better set available)

Note: there could be other optimal solutions, too! We're just proving that ours is at least as good as any optimal solution.

ACTIVITY SELECTION: CORRECTNESS

We want to prove that the algorithm finds an optimal set of activities (i.e. there isn't a better set available)

Note: there could be other optimal solutions, too! We're just proving that ours is at least as good as any optimal solution.

High-level proof idea:

At every step of the algorithm, the greedy choice we make doesn't rule out an optimal solution. By the end of the algorithm, we've got some solution, so it must be optimal!

In other words, at every step of the algorithm, there is always an optimal solution that *extends* the set of choices we made so far.

We'll perform induction on the # of greedy choices we make!

ACTIVITY SELECTION: CORRECTNESS

INDUCTIVE HYPOTHESIS

After adding the t^{th} activity, there is an optimal solution that extends the current set of activities.

ACTIVITY SELECTION: CORRECTNESS

INDUCTIVE HYPOTHESIS

After adding the t^{th} activity, there is an optimal solution that extends the current set of activities.

BASE CASE

After adding 0 activities, there is an optimal solution extending that (any solution extends 0 activities).

ACTIVITY SELECTION: CORRECTNESS

INDUCTIVE HYPOTHESIS

After adding the t^{th} activity, there is an optimal solution that extends the current set of activities.

BASE CASE

After adding 0 activities, there is an optimal solution extending that (any solution extends 0 activities).

INDUCTIVE STEP *(weak induction)*

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities. **Let's do this step!**

ACTIVITY SELECTION: CORRECTNESS

INDUCTIVE HYPOTHESIS

After adding the t^{th} activity, there is an optimal solution that extends the current set of activities.

BASE CASE

After adding 0 activities, there is an optimal solution extending that (any solution extends 0 activities).

INDUCTIVE STEP *(weak induction)*

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities. **Let's do this step!**

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 set of activities (there is no remaining activity that we could still fit in). So, the current solution is optimal.

INDUCTIVE STEP

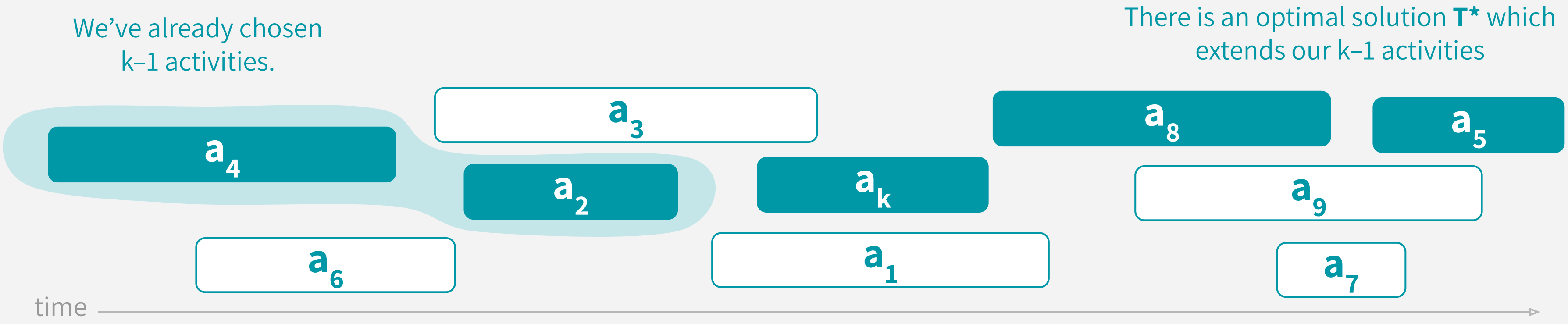
Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities.

Let T* denote the optimal solution that extends our k–1 activities. Our greedy algo's next choice is a_k.

INDUCTIVE STEP

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities.

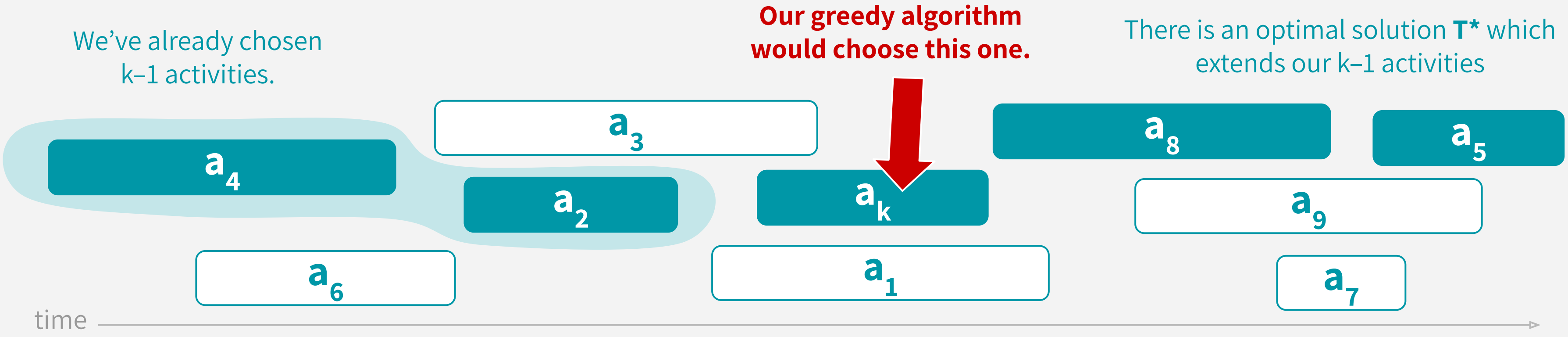
Let T* denote the optimal solution that extends our k–1 activities. Our greedy algo's next choice is a_k.



INDUCTIVE STEP

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities.

Let T* denote the optimal solution that extends our k–1 activities. Our greedy algo's next choice is a_k.



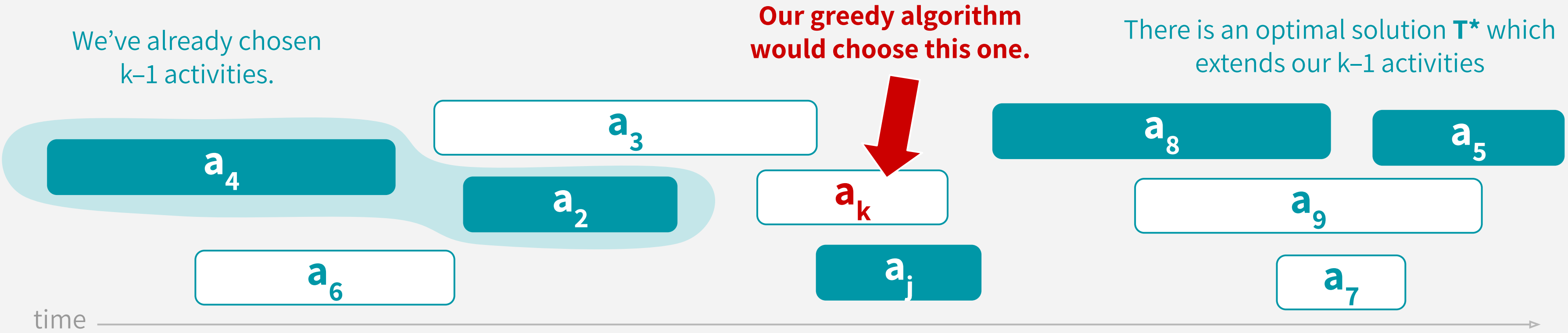
Case 1: Our greedy choice of a_k belongs in T*.

Clearly, T* still extends these k activities, so this case is all good.

INDUCTIVE STEP

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities.

Let T* denote the optimal solution that extends our k–1 activities. Our greedy algo's next choice is a_k.

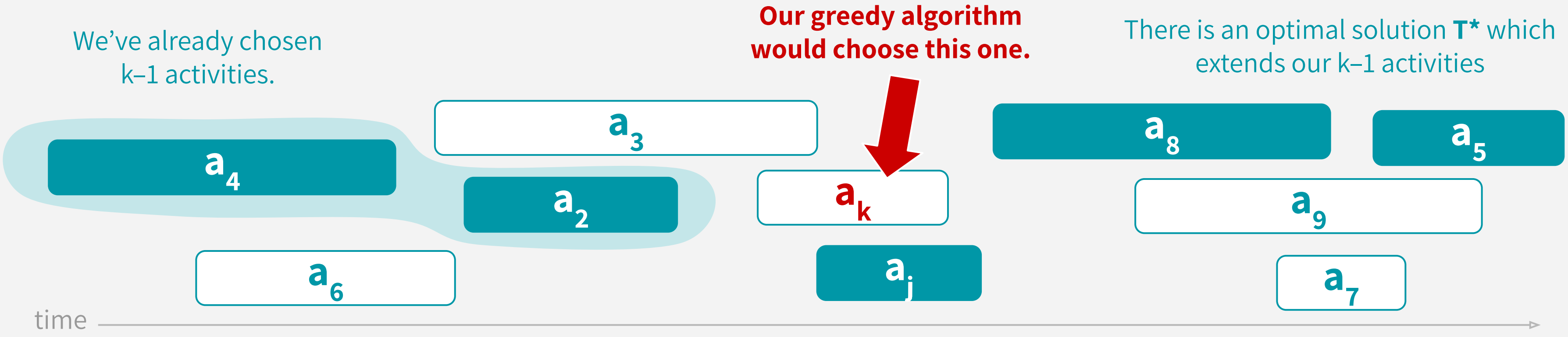


Case 2: Our greedy choice of a_k doesn't belong in T*.

INDUCTIVE STEP

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities.

Let T* denote the optimal solution that extends our k–1 activities. Our greedy algo's next choice is a_k.



Case 2: Our greedy choice of a_k doesn't belong in T*.

Then, let a_j be the activity in **T*** (after the first k–1 activities) w/ the smallest end time.

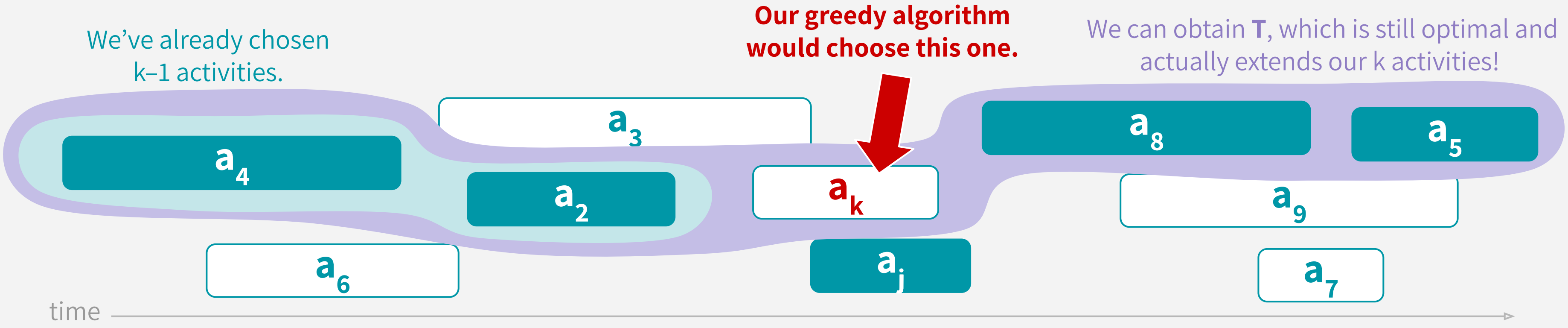
Since a_k was chosen because it has the smallest end time, we know it ends before a_j.

Thus, a_k doesn't conflict with anything in **T*** after a_j. Swapping a_k with a_j would preserve optimality!

INDUCTIVE STEP

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities.

Let T* denote the optimal solution that extends our k–1 activities. Our greedy algo's next choice is a_k.



Case 2: Our greedy choice of a_k doesn't belong in T*.

In other words, the **schedule T obtained by replacing a_j in T* with a_k** would be an optimal one.

Thus, we know that after adding our greedy algorithm's k^{th} activity,

there is still an **optimal solution (T)** that extends our k chosen activities, and this case is done!

ACTIVITY SELECTION: CORRECTNESS

INDUCTIVE HYPOTHESIS

After adding the t^{th} activity, there is an optimal solution that extends the current set of activities.

BASE CASE

After adding 0 activities, there is an optimal solution extending that (any solution extends 0 activities).

INDUCTIVE STEP *(weak induction)*

Suppose we've already chosen (k–1) activities, and there's still an optimal solution that extends these choices. After adding the k^{th} activity, we'll show that there is still an optimal solution that extends these k activities. **We proved the existence of an optimal solution that extends our k activities (in 2 cases).**

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 set of activities (there is no remaining activity that we could still fit in). So, the current solution is optimal.

سوال؟

A STRATEGY FOR GREEDY PROOFS

Prove that after each choice, you're not ruling out success.
(i.e. you're not ruling out 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 show that you won't rule out success after choice t+1 (more strategy details on next slide)
- **CONCLUSION:** If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

A STRATEGY FOR GREEDY PROOFS

The inductive step (If you haven't ruled out success after choice t, then show that you won't rule out success after choice t+1) **will often look like:**

A STRATEGY FOR GREEDY PROOFS

The inductive step (If you haven't ruled out success after choice t, then show that you won't rule out success after choice t+1) **will often look like:**

Suppose we're on track to make some optimal solution T*

(e.g. after we've picked k–1 activities, we're still on track)

A STRATEGY FOR GREEDY PROOFS

The inductive step (If you haven't ruled out success after choice t, then show that you won't rule out success after choice t+1) **will often look like:**

Suppose we're on track to make some optimal solution T*

(e.g. after we've picked k–1 activities, we're still on track)

Suppose that T* disagrees with our next greedy choice

(e.g. T* doesn't involve activity k)

A STRATEGY FOR GREEDY PROOFS

The inductive step (If you haven't ruled out success after choice t, then show that you won't rule out success after choice t+1) **will often look like:**

Suppose we're on track to make some optimal solution T*
(e.g. after we've picked k–1 activities, we're still on track)

Suppose that T* disagrees with our next greedy choice
(e.g. T* doesn't involve activity k)

Manipulate T* in order to make another solution T that's not worse (i.e. also optimal) but now *agrees* with our greedy choice!
(e.g. replace whatever activity T* had picked next with our greedy choice of activity k)

A STRATEGY FOR GREEDY PROOFS

This is not the only way to prove that greedy algorithms are correct!
Unlike previous classes that we discussed divide-and-conquer, where there was a pretty formulaic approach to proving that divide-and-conquer algorithms were correct, there isn’t a one-size-fits-all approach to proving every greedy algorithm. As you can see in books, there are other styles of proofs, and the strategy used varies problem by problem. However, the strategy we use here is a good starting point, so that’s why we’ll focus on this in class!

DP vs. GREEDY

Like Dynamic Programming, Greedy algorithms often work for problems with nice optimal substructure. However, not only are optimal solutions to a problem made up from optimal solutions of sub-problems,

but each problem depends on only one sub-problem!

(there's some "best" decision to be made now, and then we solve a single sub-problem)

DP vs. GREEDY

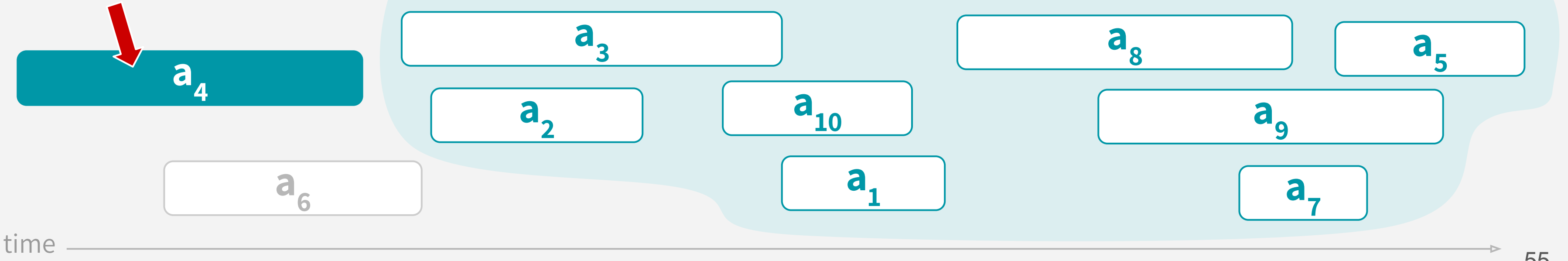
Like Dynamic Programming, Greedy algorithms often work for problems with nice optimal substructure. However, not only are optimal solutions to a problem made up from optimal solutions of sub-problems,

but each problem depends on only one sub-problem!

(there's some "best" decision to be made now, and then we solve a single sub-problem)

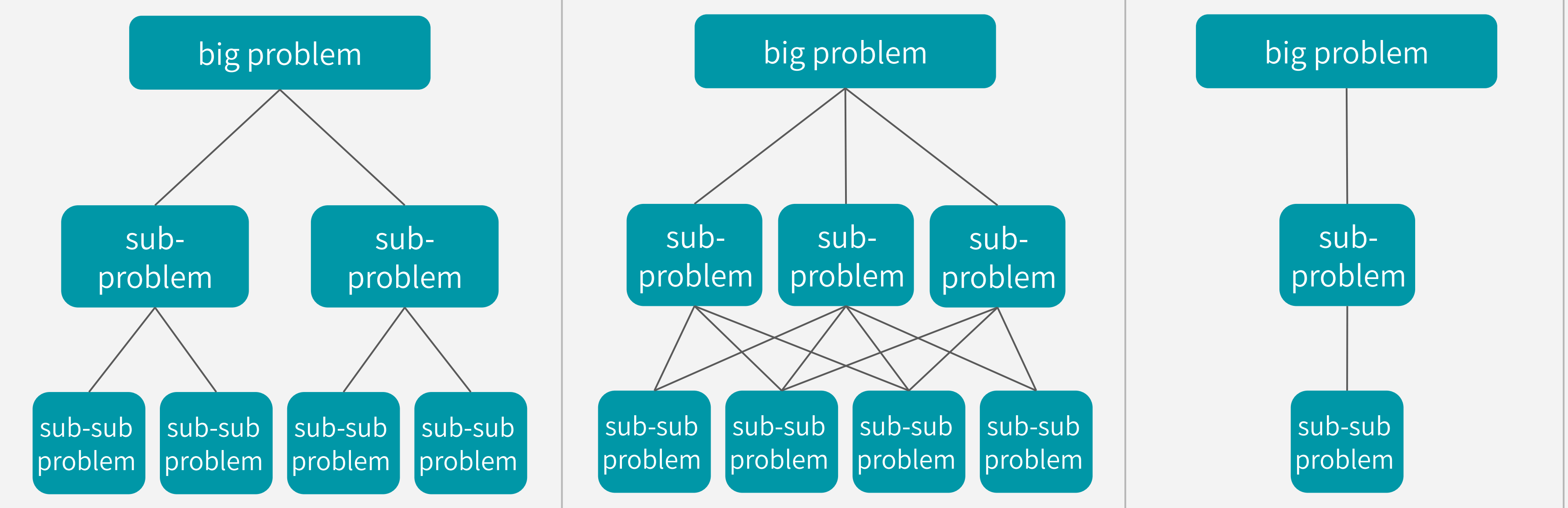
In our greedy activity selection problem, we made a choice...

And then we moved on to solve this subproblem!
(i.e. find the optimal set of activities with this smaller set of activities)



D&C vs. DP vs. GREEDY

DIVIDE-AND-CONQUER	DYNAMIC PROGRAMMING	GREEDY



سوال؟