

CSE 417 Autumn 2025

# Lecture 18: Greedy algorithms

Glenn Sun

**What is greedy?**

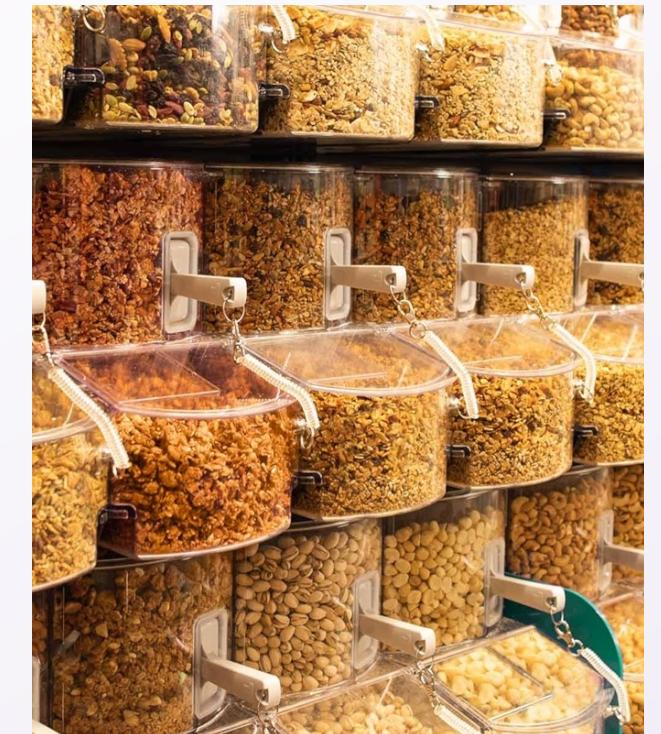
# Whole Foods thief (“fractional knapsack”)

**Input:** At the Whole Foods bulk department:

- For item  $i$ , there are  $w_i$  pounds in the bin at  $\$p_i/\text{lb}$ .
- You can carry  $W$  pounds total.

**Goal:** Steal as much value as possible

**Algorithm:** Take as much as possible of the most expensive bin, until you can't carry more!



# Whole Foods thief

**Algorithm:** Take as much as possible of the most expensive bin, until you can't carry more!

**Why does it work?** Suppose another solution didn't do this and look at the first time our solutions differed, in order of price.

	2 lbs	3 lbs	1 lb	5 lbs
Available:	\$5/lb	\$4/lb	\$3/lb	\$3/lb
We picked:	2 lbs	3 lbs	1 lb	2 lbs
Other solution:	2 lbs	3 lbs	0 lb	3 lbs

# Whole Foods thief

The other solution must have taken less of this item, because we took the maximum possible.

In the other solution, they could've picked this, and then picked less of something else later.

	2 lbs	3 lbs	1 lb	5 lbs
Available:	\$5/lb	\$4/lb	\$3/lb	\$3/lb
We picked:	2 lbs	3 lbs	1 lb	2 lbs
Other solution:	2 lbs	3 lbs	0 lb	3 lbs

# Whole Foods thief

Making this change can only improve the other solution (or stay the same quality), because later items are cheaper (or same price).

Because every other solution can only get better by becoming ours, our solution must be optimal!

	2 lbs	3 lbs	1 lb	5 lbs
Available:	\$5/lb	\$4/lb	\$3/lb	\$3/lb
We picked:	2 lbs	3 lbs	1 lb	2 lbs
Other solution:	2 lbs	3 lbs	1 lb	2 lbs

# Exchange argument

This kind of argument is called an **exchange argument**.

To show that your solution is optimal,

- Consider a different solution. Usually, it helps to consider the **first time that it differs**.
- Show that you can **change the other solution** to be more like yours, while improving or maintaining the quality.
- Conclude that yours is optimal!

# What is greedy?

**Greedy algorithms** solve optimization problems by:

- Making a small choice that is “locally optimal”
- Repeating this on the remaining subproblem

If the locally optimal choice **must occur** in some globally optimal solution, then greedy will always find the global optimum!

- **Today and Friday:** When greedy is optimal
- **Monday:** Greedy algorithms that are not optimal but useful

# **More examples**

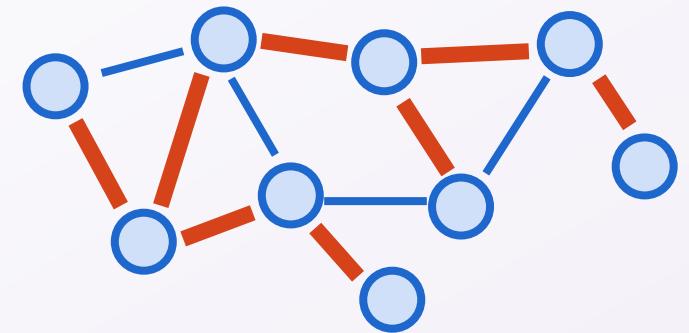
# Kruskal's/Prim's algorithms

Recall the minimum spanning tree problem:

**Input:** A connected, undirected, weighted graph with vertices  $V$  and edges  $E$

**Goal:** Find a spanning tree of minimum total weight

Kruskal's and Prim's algorithms were greedy!



# Kruskal's/Prim's algorithms

Kruskal's algorithm:

1. **repeat  $n - 1$  times**
2. Pick the cheapest edge that connects two components.

Prim's algorithm:

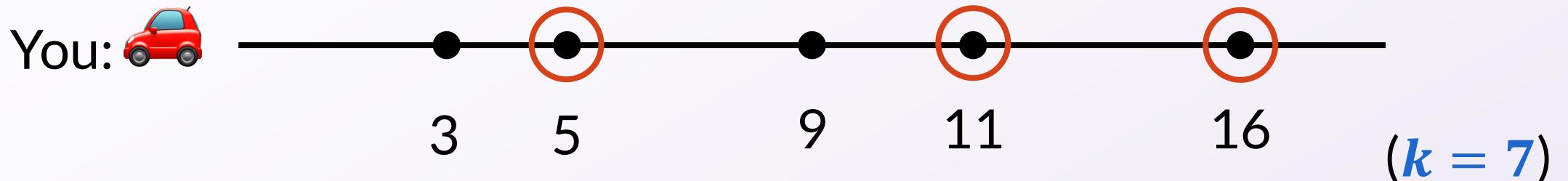
1. **repeat  $n - 1$  times**
2. Pick the cheapest edge that extends the current tree to a new vertex.

# EV charging

**Input:** On a road trip for  $L$  miles, EV chargers are available at mile markers  $A[1], \dots, A[n]$ . Your battery lasts  $k$  miles.

**Goal:** Finish the trip stopping as infrequently as possible.

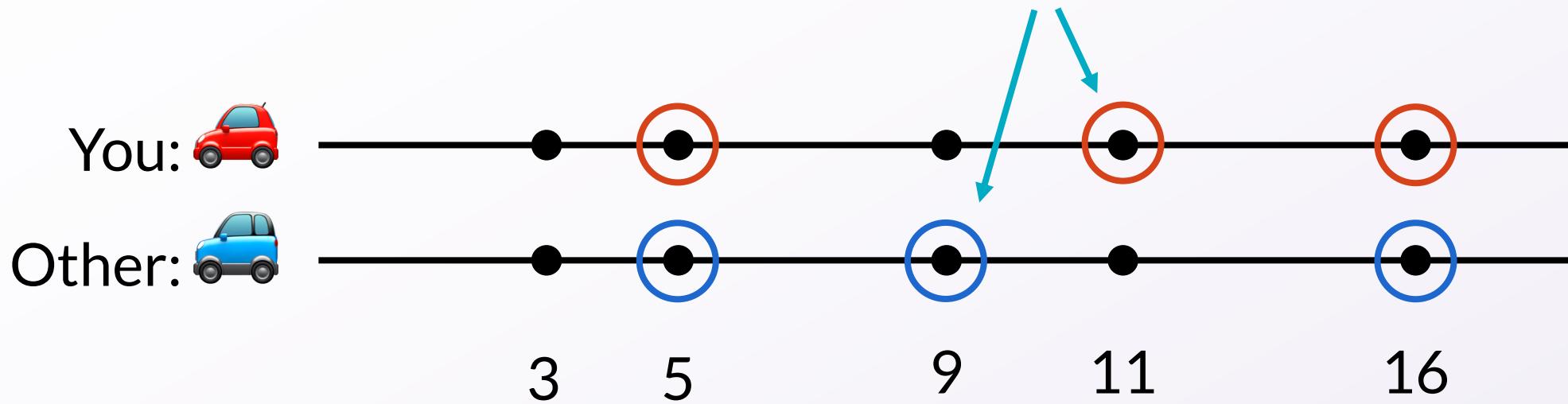
**Algorithm:** Pick the farthest charger within  $k$  miles of your current location.



# EV charging

**Algorithm:** Pick the farthest charger within  $k$  miles of your current location.

Consider a different solution, and **look at the first time** it differs.



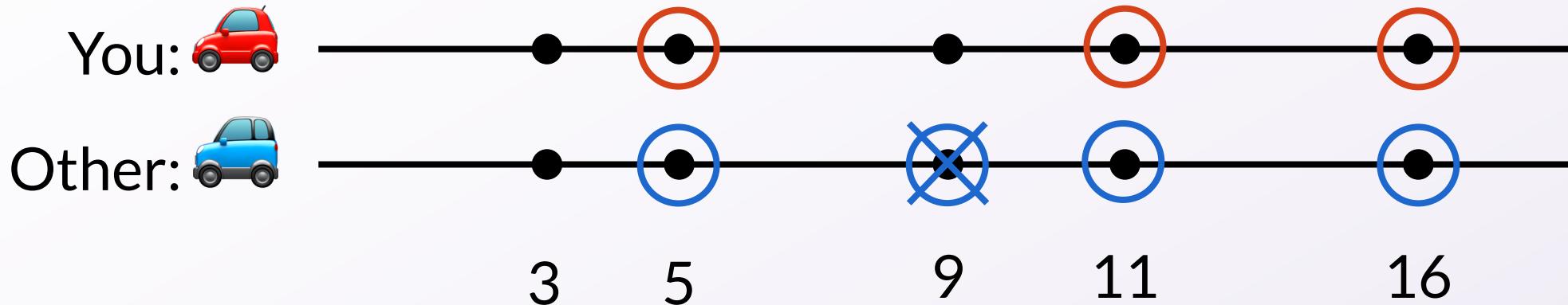
$$(k = 7)$$

# EV charging

The other charger must be closer than the one we picked.

Since the other solution was the same until this point, they could've picked this (i.e. driven to the charger we picked).

This can only improve the other solution. Thus, we were optimal!



# Making change

**Input:** A number  $n$

**Goal:** Make change using the fewest coins, with pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents).

**Algorithm:** Take the largest coin that we still can.

# Running time for making change

**Algorithm:** Take the largest coin that we still can.

Naive implementation:

1. **while** amountLeft > 0 do
2.   **if** amountLeft >= 25, take a quarter and update amountLeft
3.   **else if** amountLeft >= 10, take a dime and update amountLeft
4.   **else if** amountLeft >= 5, take nickel and update amountLeft
5.   **else** take a penny and update amountLeft

On input  $n$ , takes  $O(n)$  time.

# Running time for making change

**Algorithm:** Take the largest coin that we still can.

Better implementation:

1. Take  $\lfloor \text{amountLeft}/25 \rfloor$  quarters and update amountLeft
2. Take  $\lfloor \text{amountLeft}/10 \rfloor$  dimes and update amountLeft
3. Take  $\lfloor \text{amountLeft}/5 \rfloor$  nickels and update amountLeft
4. Take the remaining amount in pennies.

Takes  $O(1)$  time!

# Different exchange argument for change

An optimal solution uses no more than 4 pennies:

- If there are 5 pennies, swap with 1 nickel.

An optimal solution uses no more than 1 nickel:

- If there are 2 nickels, swap with 1 dime.

An optimal solution uses no more than 2 dimes:

- If there are 3 dimes, swap with 1 quarter and 1 nickel.

# Greedy with other coin values

Does the greedy algorithm work with the following coin values?

**Q:** 1 cent, 2 cents, 4 cents, 8 cents, 16 cents?

**A:** Yes! Even easier exchange argument:

**Q:** 1 cent, 10 cents, 11 cents?

**A:** No. For 20 cents, greedy gives  $(11 \text{ cents}) + 9 * (1 \text{ cent})$ , using 10 coins. Optimal is  $2 * (10 \text{ cents})$ , using 2 coins.

# Making change with general coin values

Use a dynamic program! If coin values are  $A[1], \dots, A[k]$ ,

$$\text{numCoins}(n) = \min \left( \begin{array}{c} \text{numCoins}(n - A[1]) \\ \vdots \\ \text{numCoins}(n - A[k]) \end{array} \right) + 1$$

Dynamic program tries every possibility for which coin to take.

Greedy means you already know which coin you want.

DP runs in  $O(nk)$  time, whereas greedy takes just  $O(k)$ !

# Meeting scheduler

**Input:** List of time intervals (booking requests for a meeting room)

**Output:** Maximum number of meetings that can be booked



# Lots of greedy options!

Things we could try:

- Pick the shortest meeting
- Pick the meeting that conflicts with the fewest other meetings
- Pick the meeting that would start first
- Pick the meeting that would end first

(...that doesn't conflict with meetings we've already picked.)

**Q:** Can you eliminate any of these by finding counterexamples?

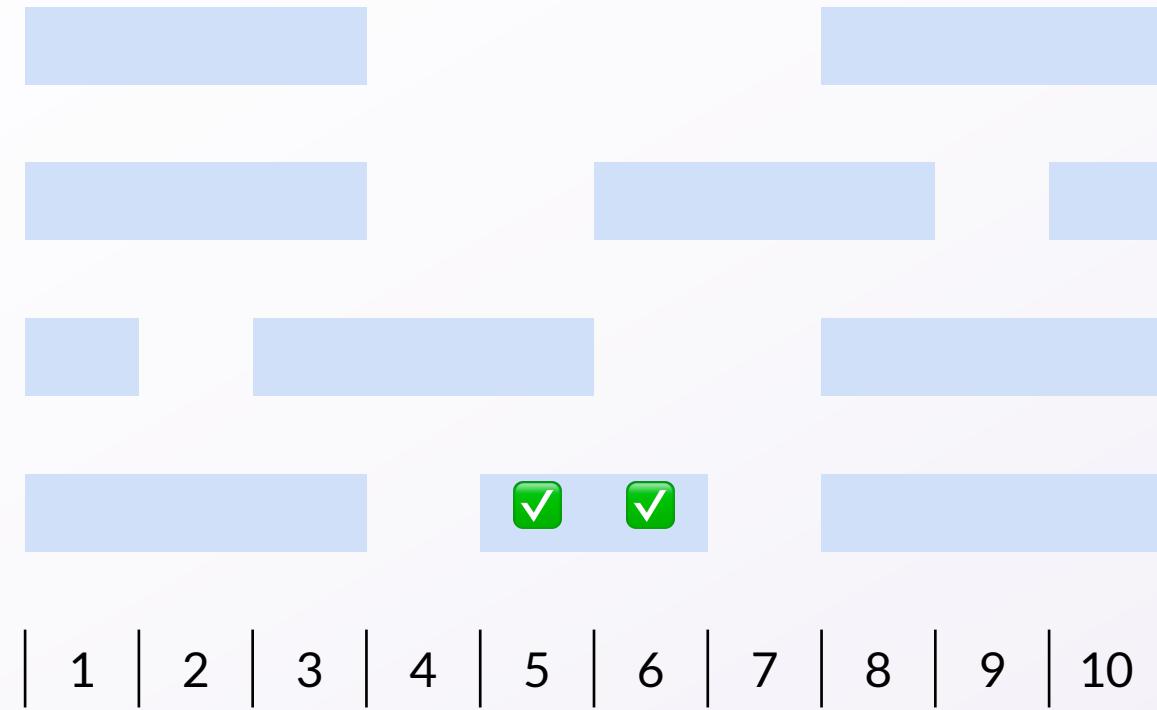
# Counterexamples

- Pick the shortest meeting



# Counterexamples

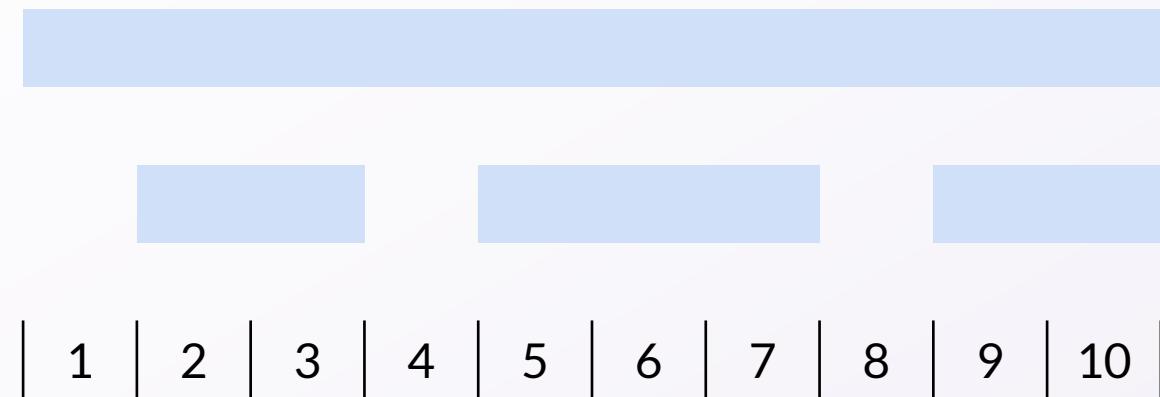
- Pick the meeting that conflicts with the fewest other meetings



( has 2 conflicts, others at least 3. Max is 4 meetings, but 3 with .)

# Counterexamples

- Pick the meeting that would start first



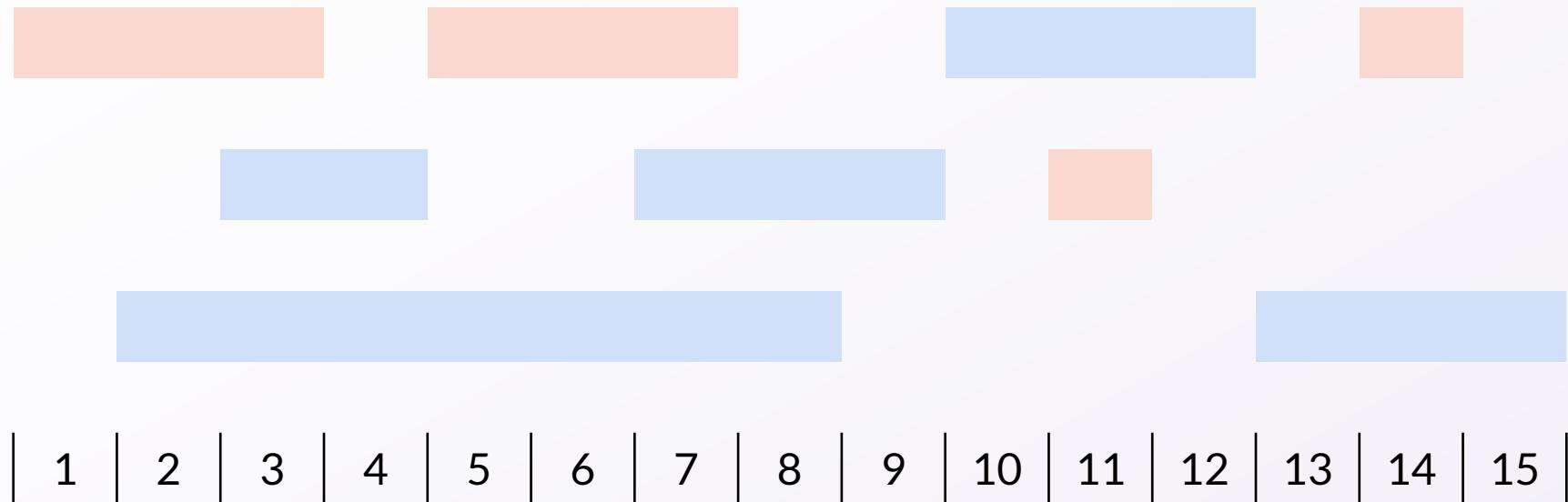
# Counterexamples

- Pick the meeting that would end first

Actually seems to work...

# Our greedy solution

**Algorithm:** Pick the meeting that would end first.



# Exchange argument for meeting scheduler

Suppose another schedule was different and look at the first time our schedules differed.



# Exchange argument for meeting scheduler

In the ★ solution, we could've picked our solution instead:

- Compatible with previous picks because previous are same
- Compatible with future picks because ■ finishes earliest



# Exchange argument for meeting scheduler

In the ★ solution, we could've picked our solution instead:

- Compatible with previous picks because previous are same
- Compatible with future picks because  finishes earliest

Because this process can transform any solution into ours while maintaining (or increasing) the number of meetings, our solution must be optimal!

# Final reminders

I have OH now-12:30pm:

- Meet at front of classroom, we'll walk over together
- CSE (Allen) 214 if you're coming later

Nathan has online OH 12-1pm:

- <https://washington.zoom.us/my/nathanbrunelle>