COMP - 285
Advanced Algorithms

Welcome to COMP 285

Lecture 19: Greedy Algorithms

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW6 Due Thursday!

#include <limits>

HW6 Due Thursday!

11/03 @ 11:59PM ET

Mock Interview with Meta!

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

Quiz!

www.comp285-fall22.ml or Blackboard

Recall where we ended last lecture…

Re re we

e d last e...

But first!

Our Course Map!

Building Blocks of Algorithms

- Recursion
- Big-O (time/space complexity)
- Data structures (stacks, queues, maps, sets, etc.)
- General problem solving methodology

Sorting Algorithms

- How to arrange and “sort” data in data structures
- Mergesort, InsertionSort, QuickSort
- Master Theorem, linear sorting approaches

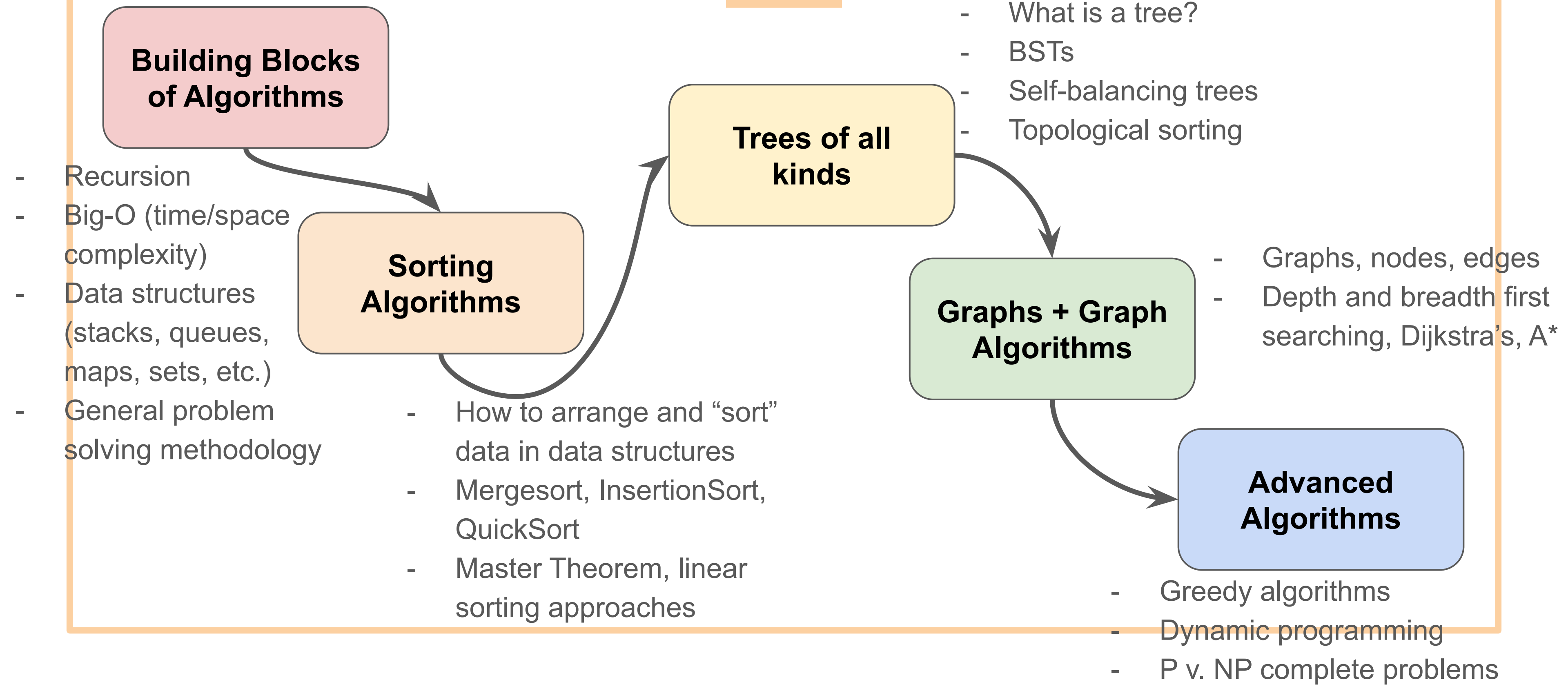
Trees of all kinds

- What is a tree?
- BSTs
- Self-balancing trees
- Topological sorting

Graphs + Graph Algorithms

- Graphs, nodes, edges
- Depth and breadth first searching, Dijkstra’s, A*

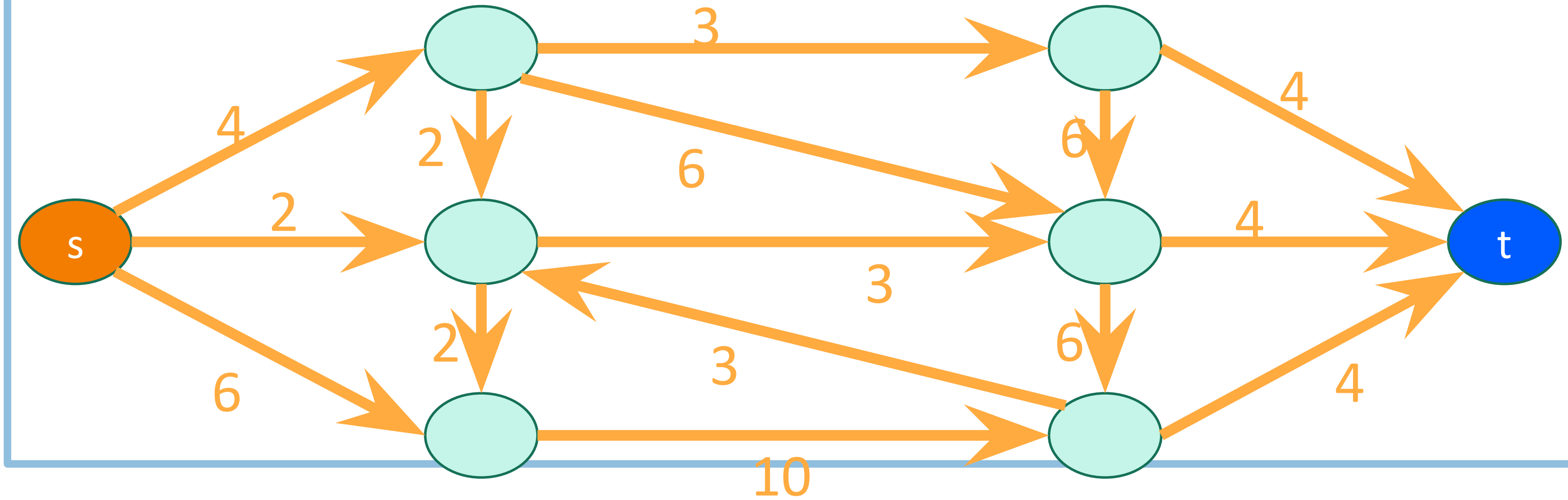
Our Course Map!



Recall where we ended last lecture…

A More General Problem Statement

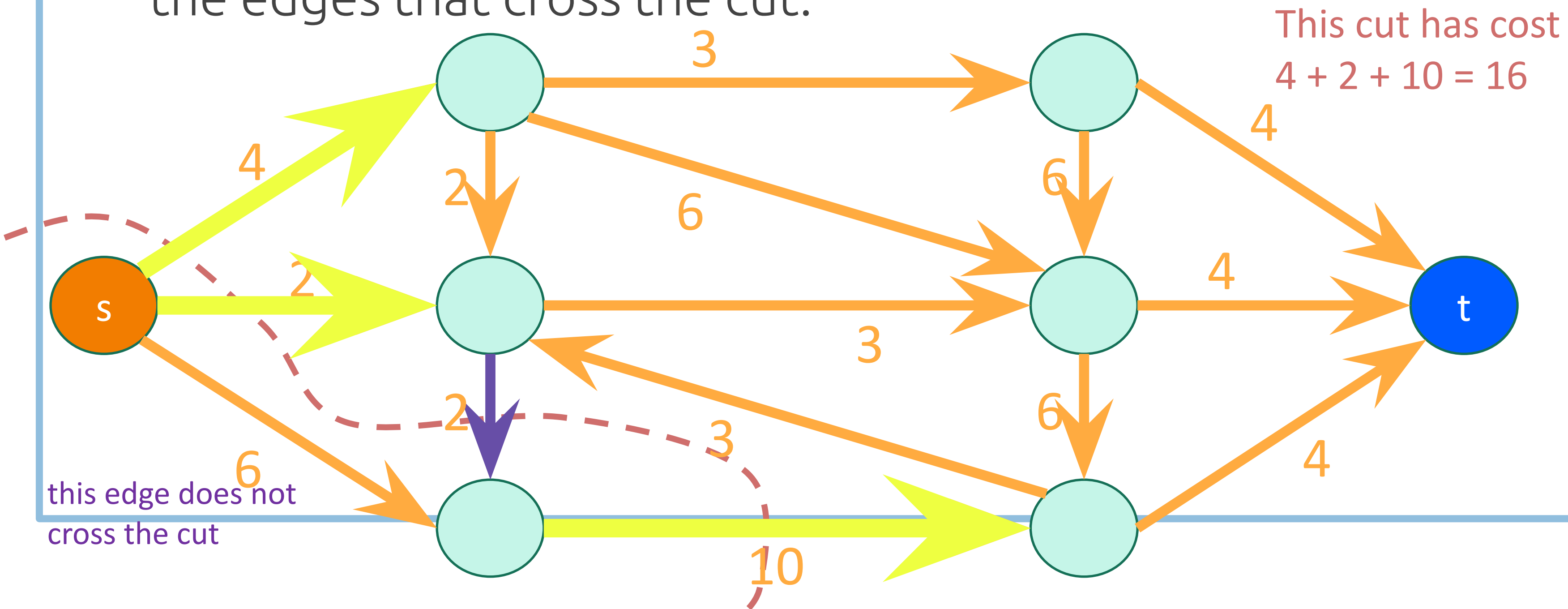
- Graphs are directed and edges have "capacities" (weights)
- We have a special "source" vertex s and "sink" vertex t.
  - s has only outgoing edges*
  - t has only incoming edges*



*simplifying assumptions, but everything can be generalized to arbitrary directed graphs

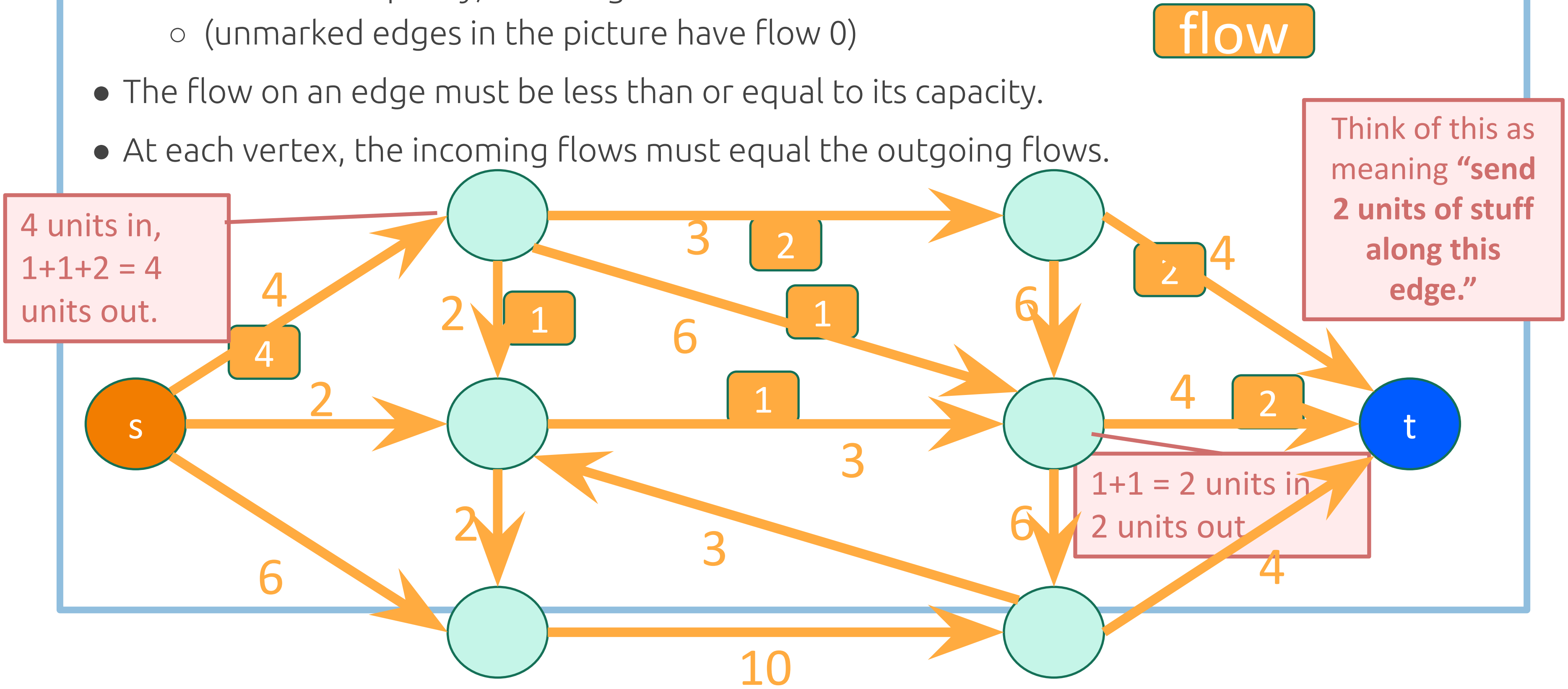
An s-t cut is a cut which separates s from t

- An edge **crosses the cut** if it goes from s's side to t's side.
- The **cost** (or capacity) of a cut is the sum of the capacities of the edges that cross the cut.



Flows

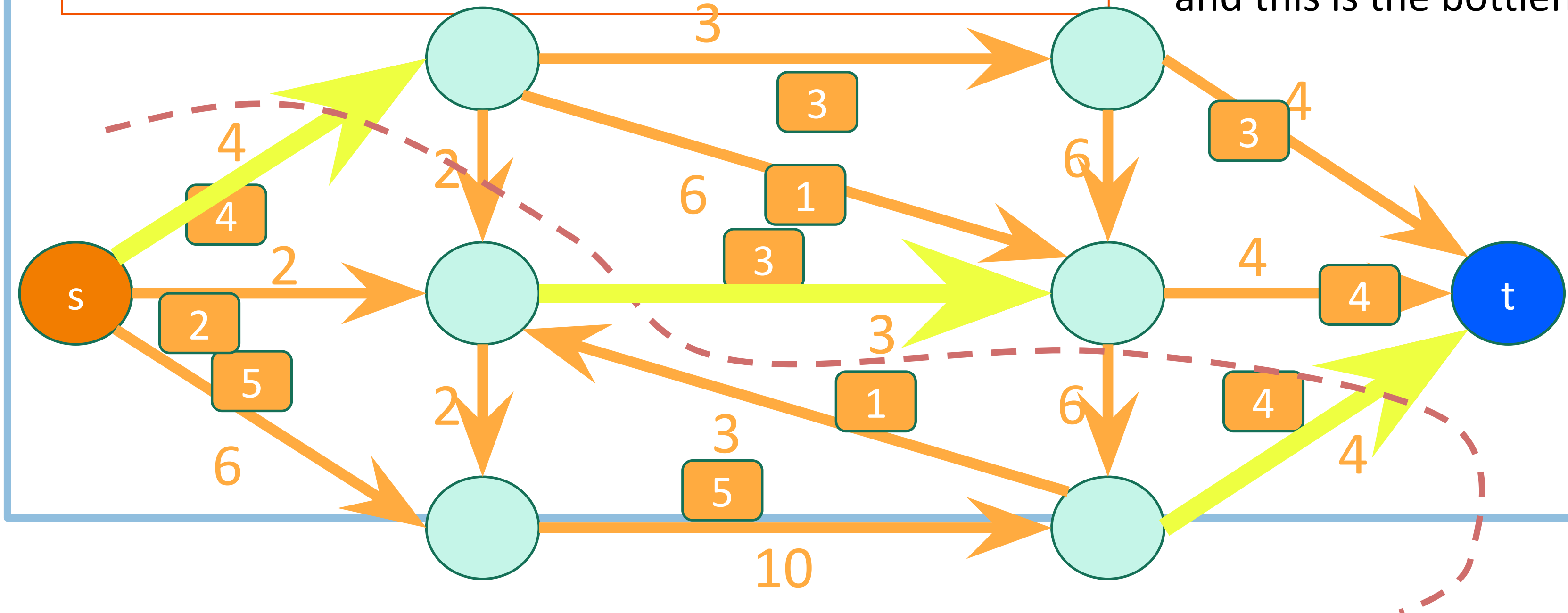
- In addition to a capacity, each edge has a flow
  - (unmarked edges in the picture have flow 0)
- The flow on an edge must be less than or equal to its capacity.
- At each vertex, the incoming flows must equal the outgoing flows.



Theorem: Max-flow min-cut theorem

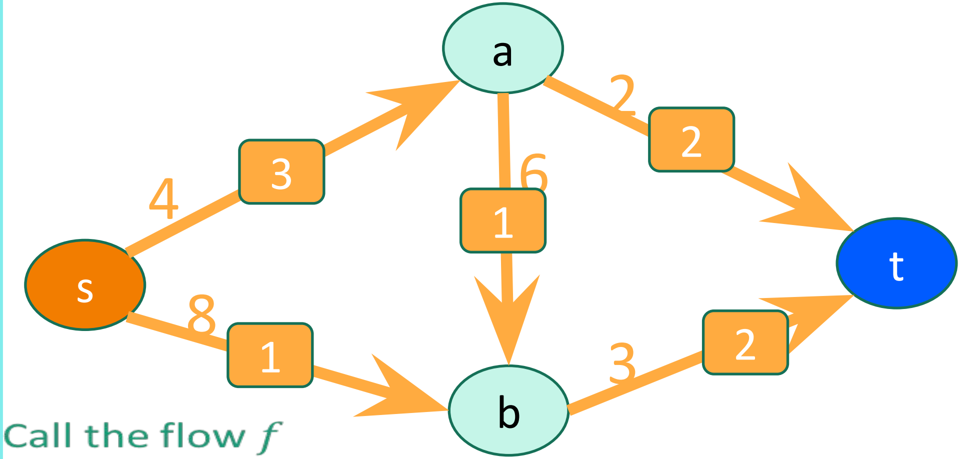
The value of a max flow from s to t
is equal to
the cost of a min s-t cut.

Intuition: in a max flow, the min cut better fill up, and this is the bottleneck.



Residual networks

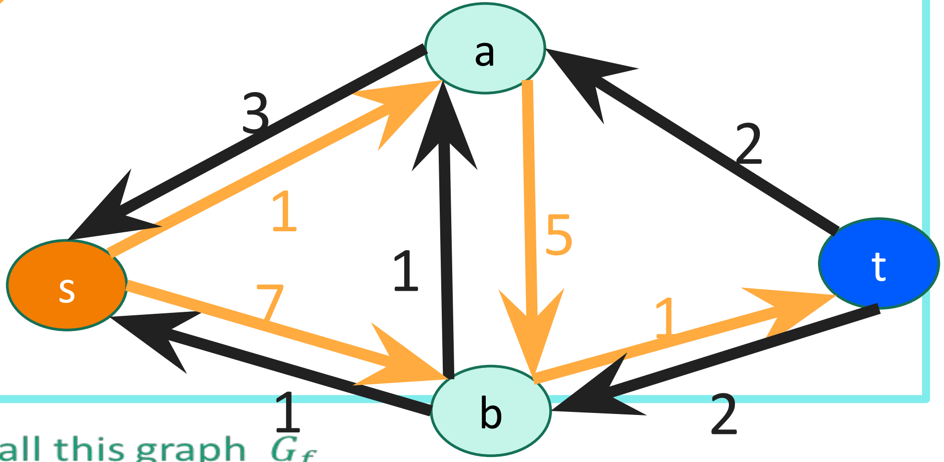
Say we have a flow...



Call the flow f
Call the graph G

Forward edges are the amount that's left.

Backwards edges are the amount that's been used.



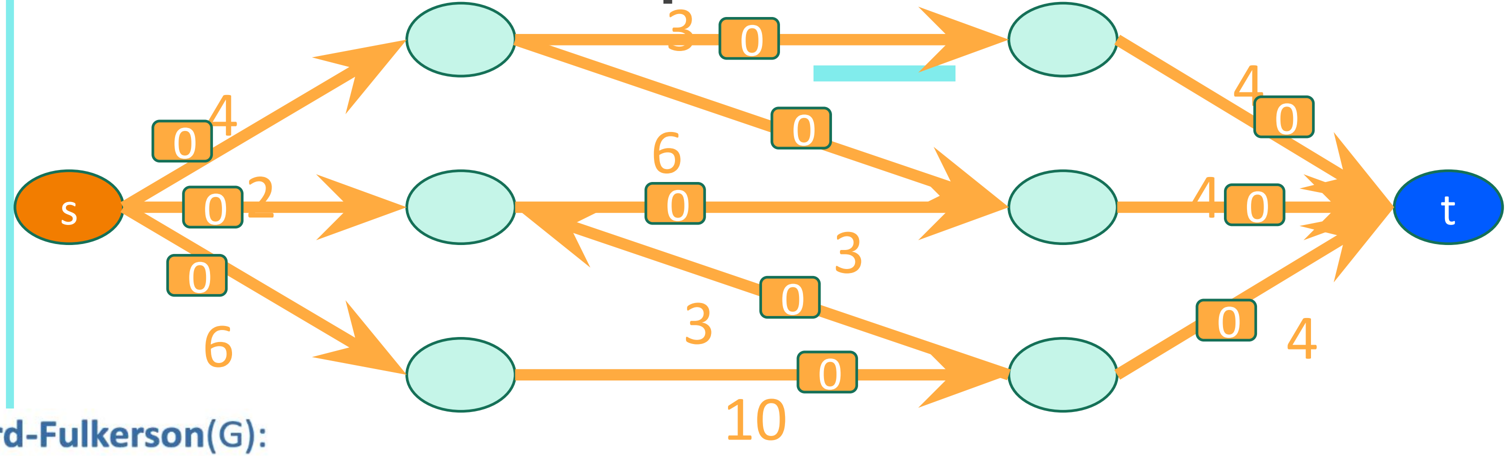
Call this graph G_f

Create a new **residual network** from this flow:

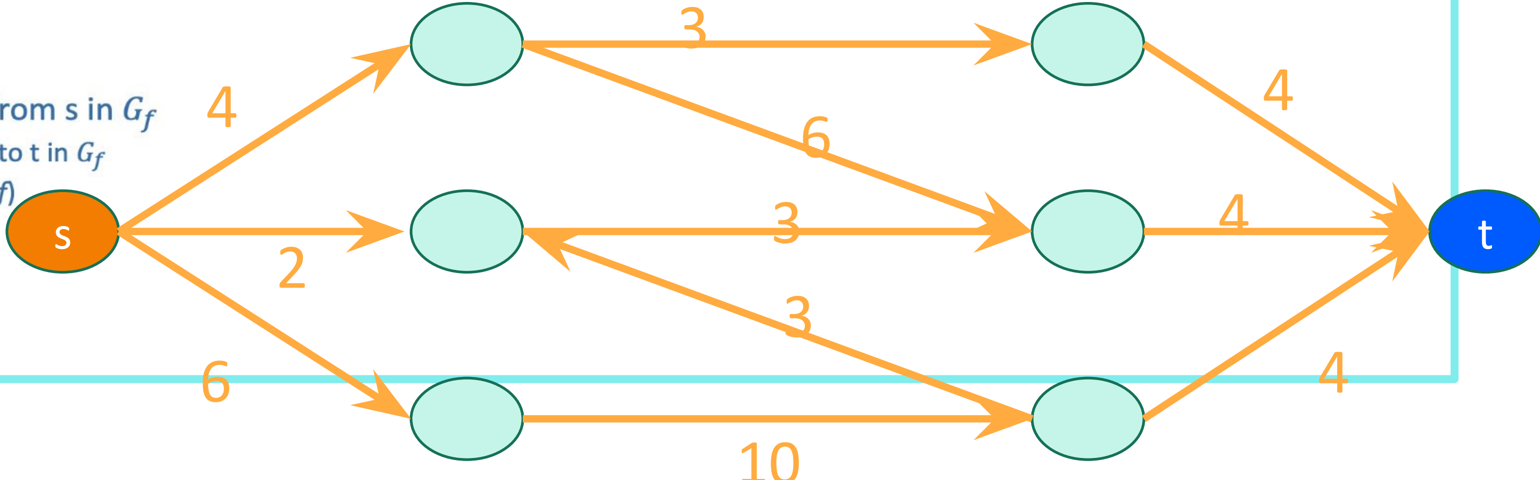
Ford-Fulkerson Algorithm

- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f // e.g., use DFS or BFS
    - $f \leftarrow$ **increaseFlow**(P, f)
    - update G_f
  - **return** f

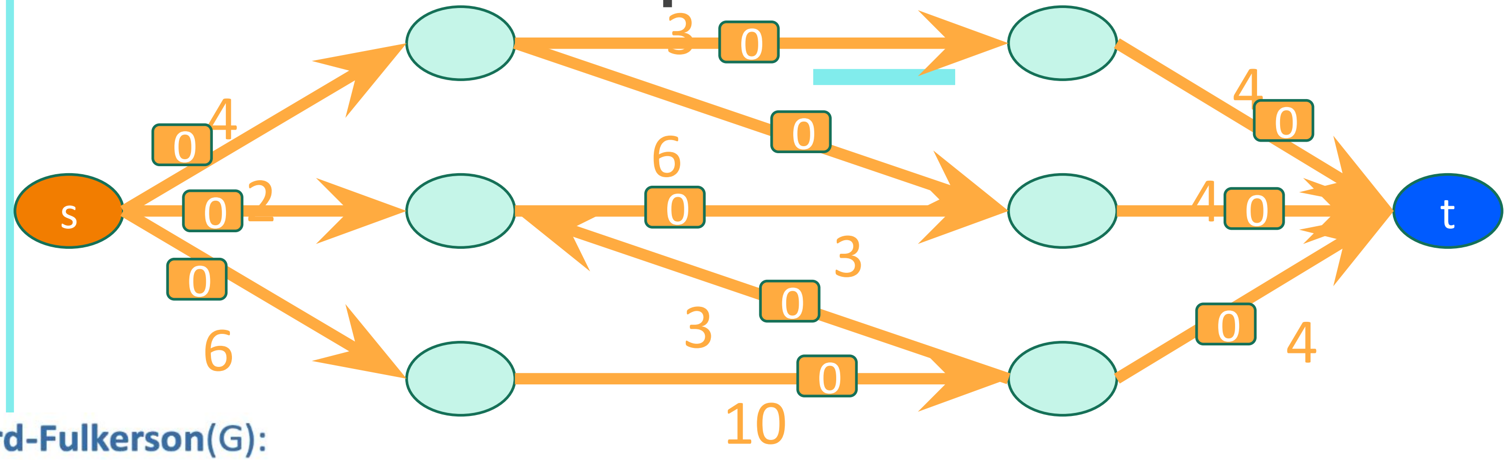
Example of Ford-Fulkerson



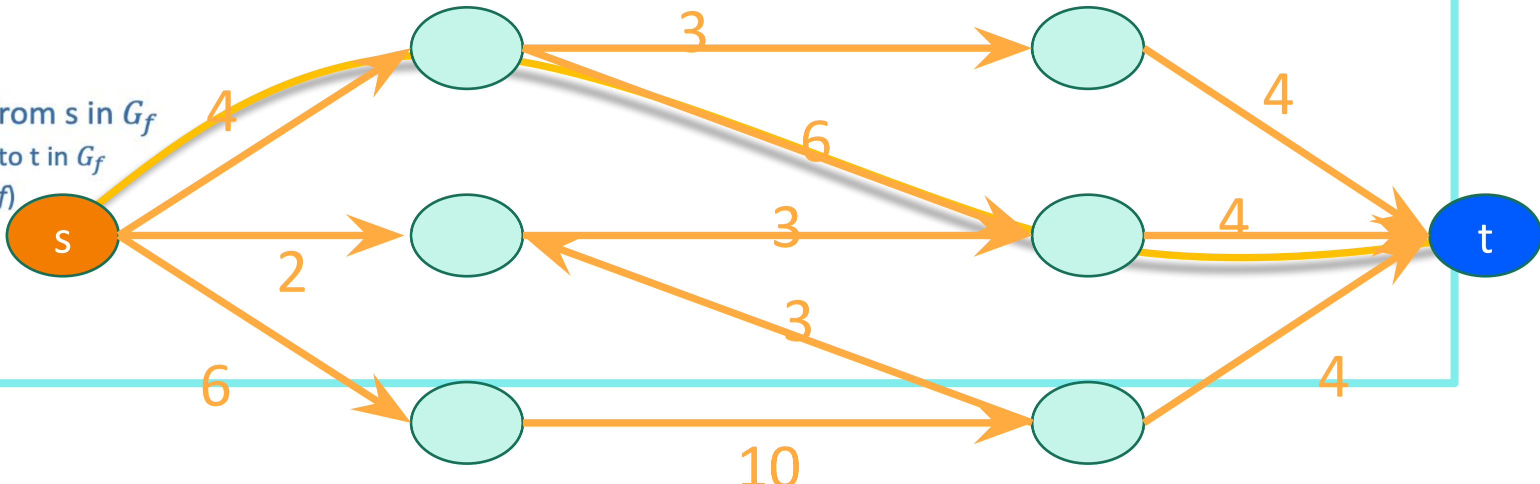
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



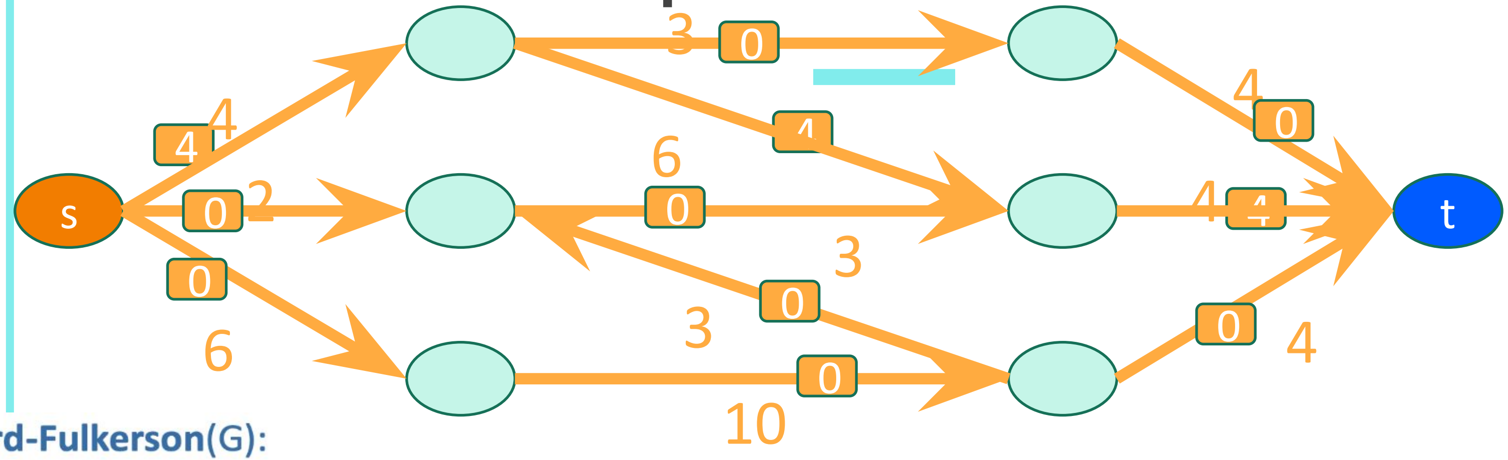
Example of Ford-Fulkerson



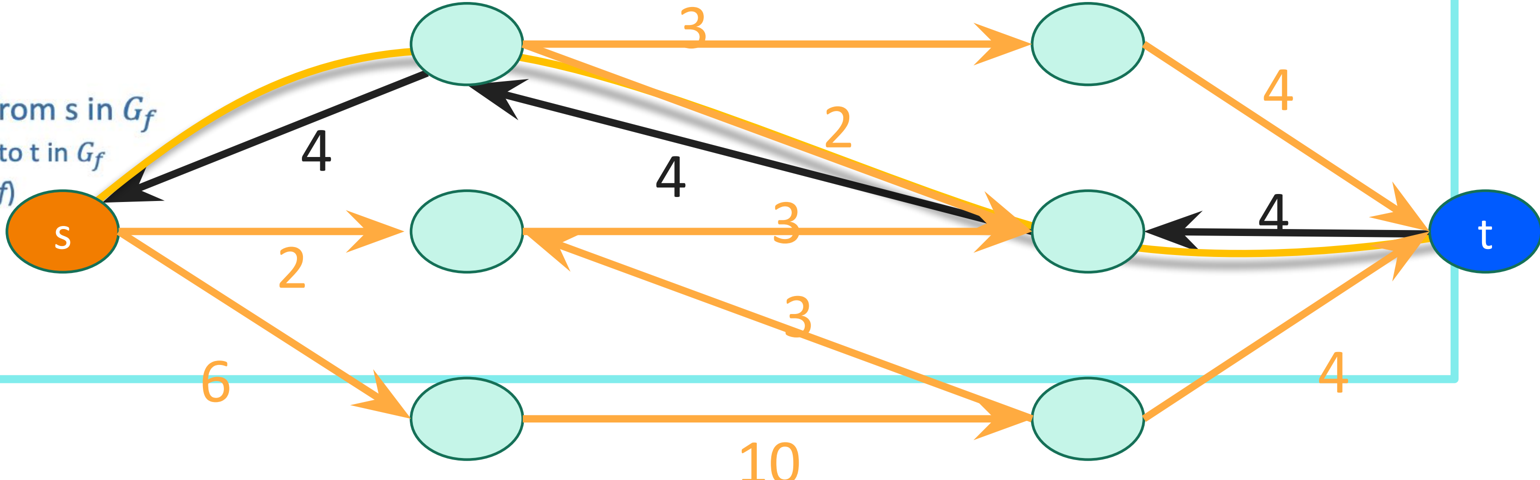
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



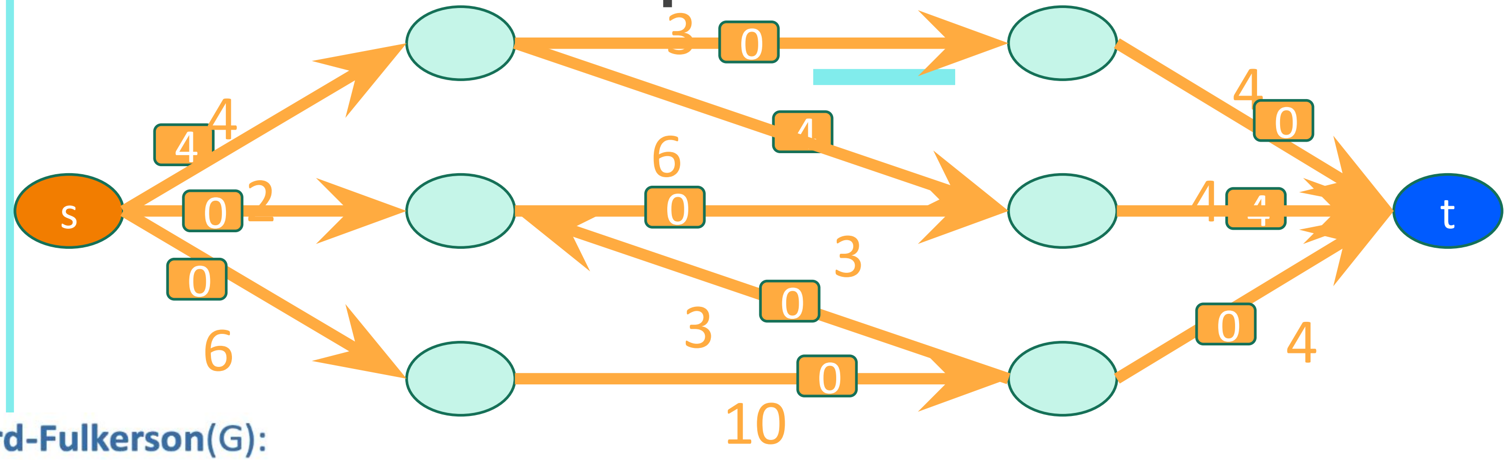
Example of Ford-Fulkerson



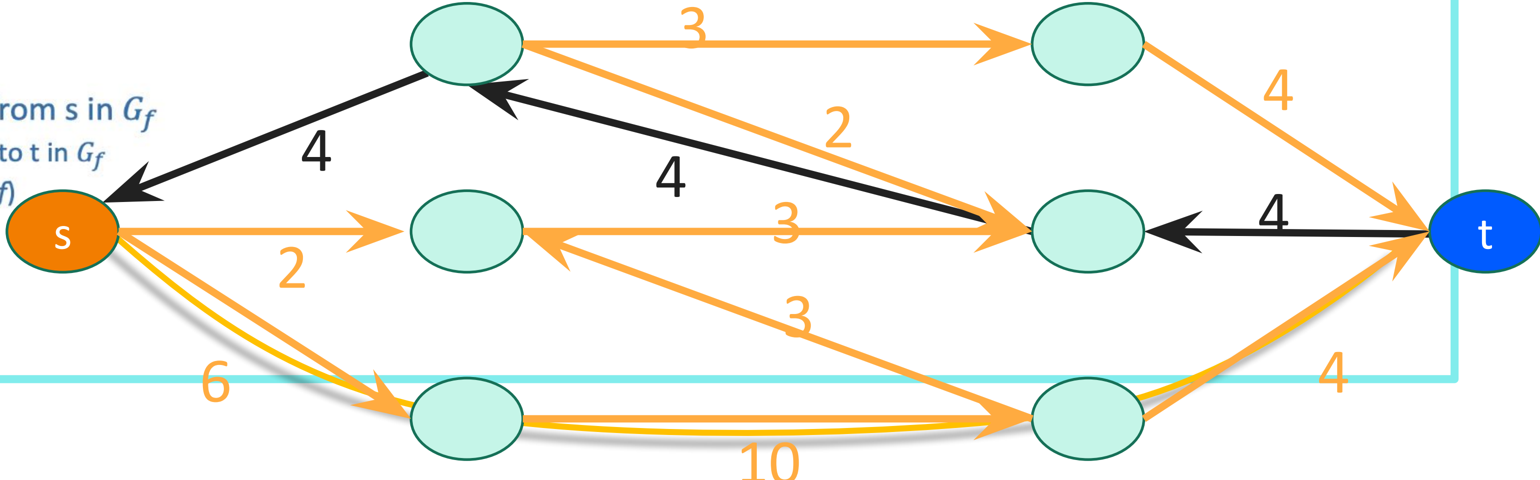
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



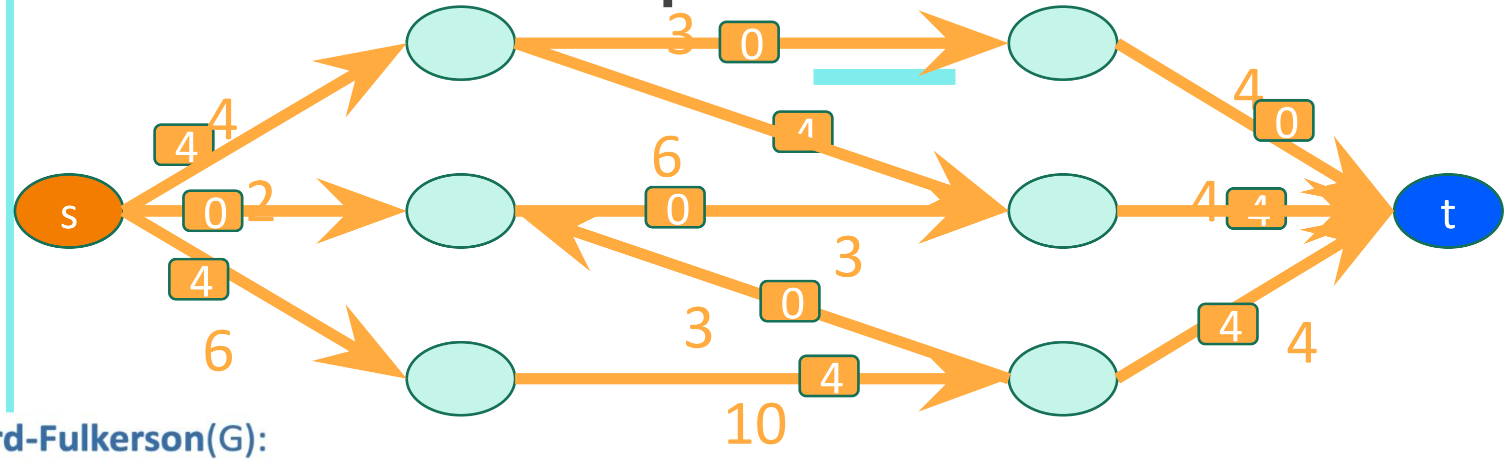
Example of Ford-Fulkerson



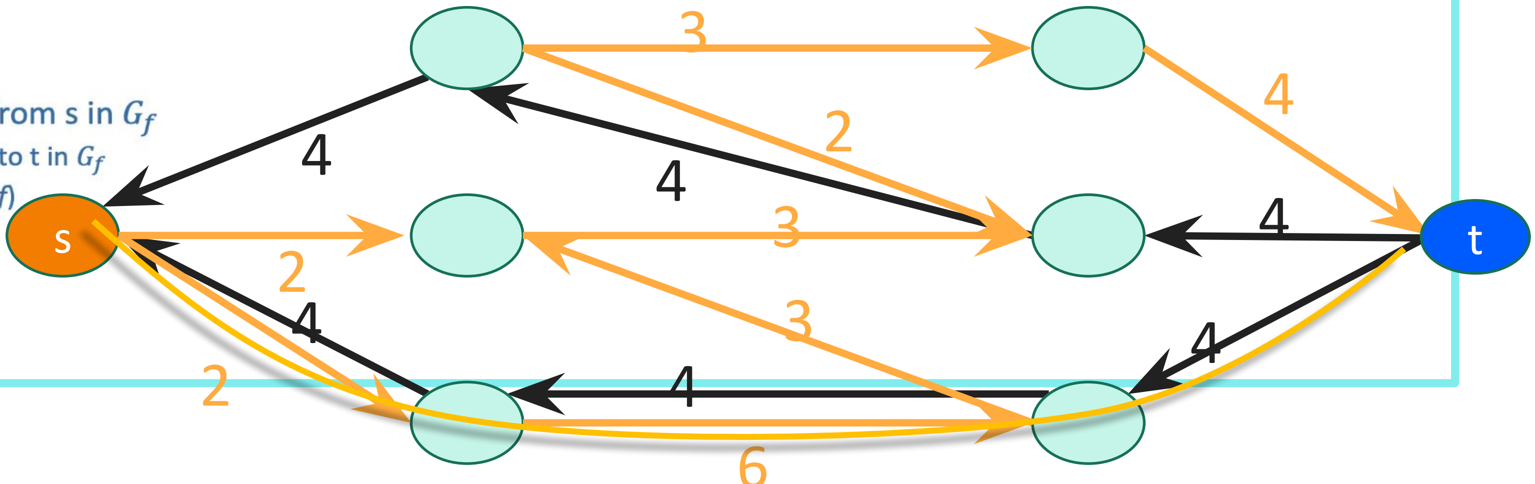
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



Example of Ford-Fulkerson

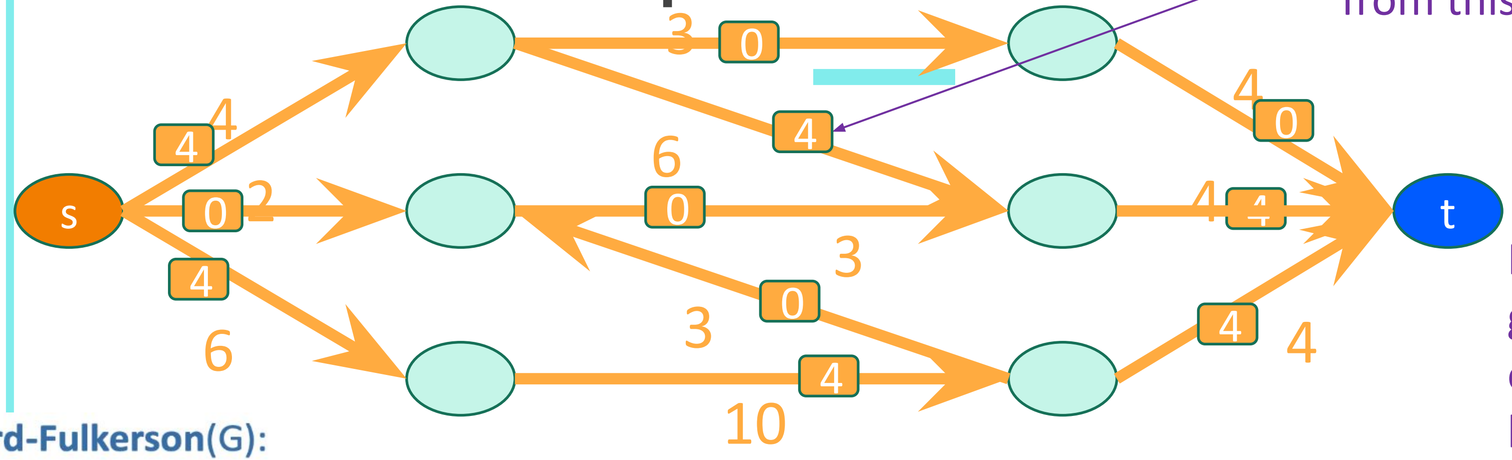


- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



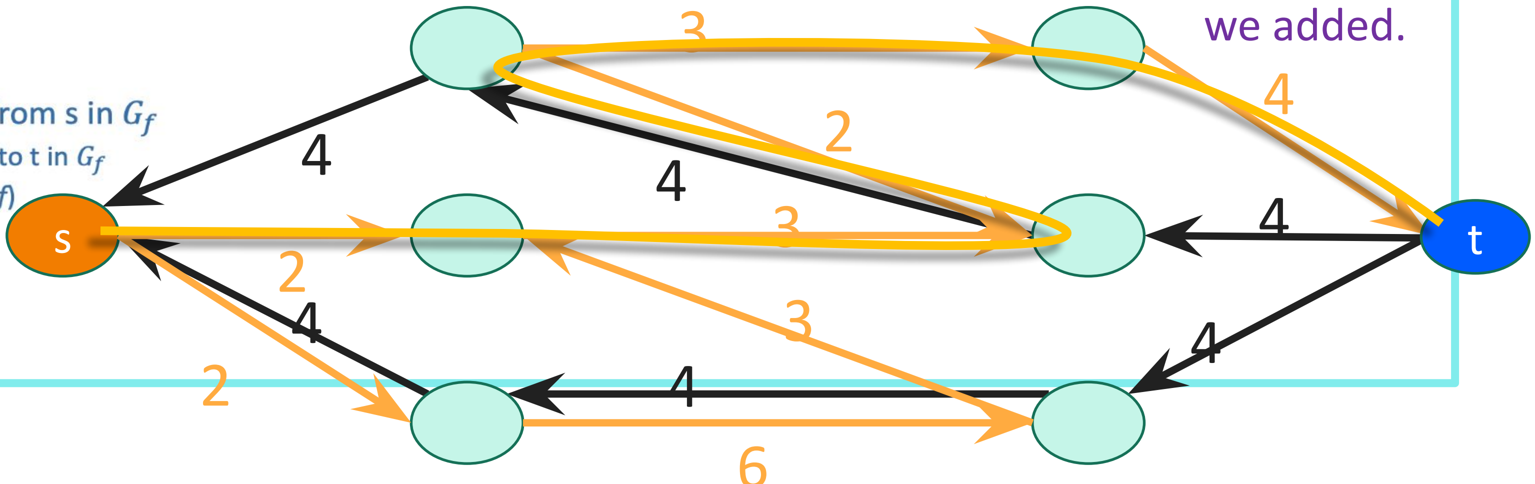
Example of Ford-Fulkerson

We will **remove** flow from this edge.



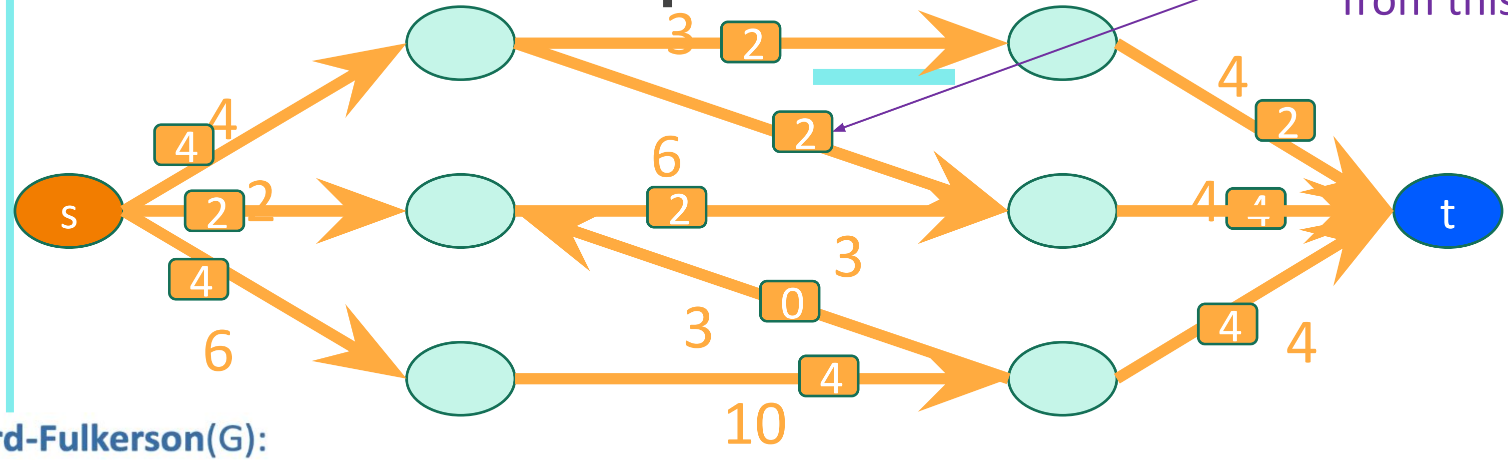
Notice that we're going back along one of the backwards edges we added.

- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



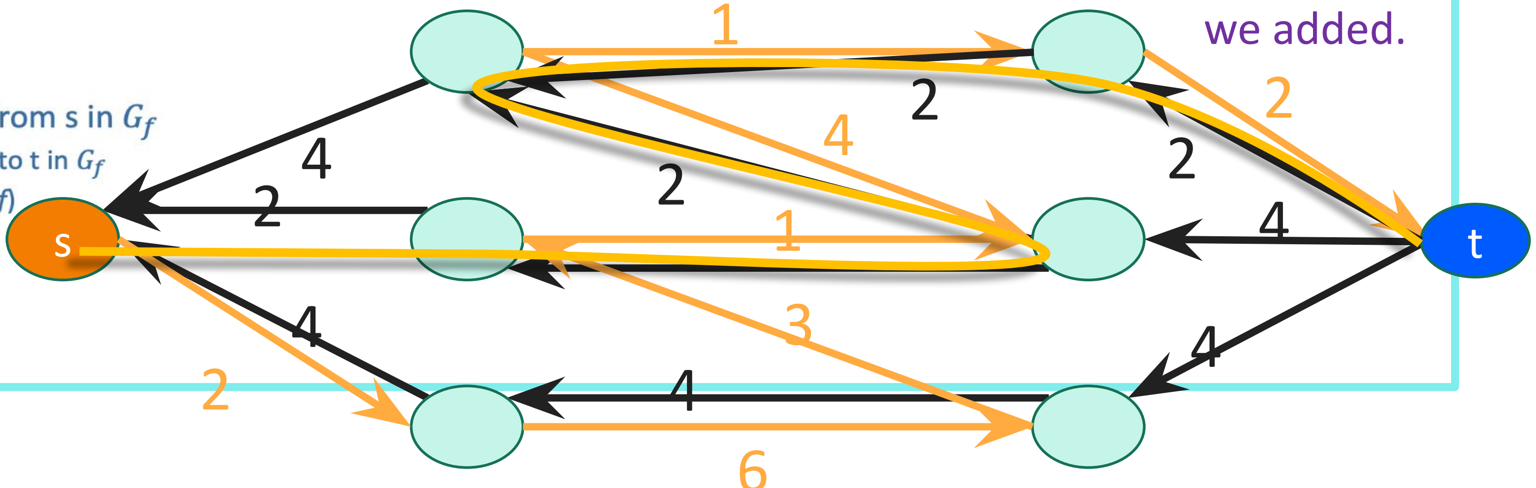
Example of Ford-Fulkerson

We will **remove** flow from this edge.



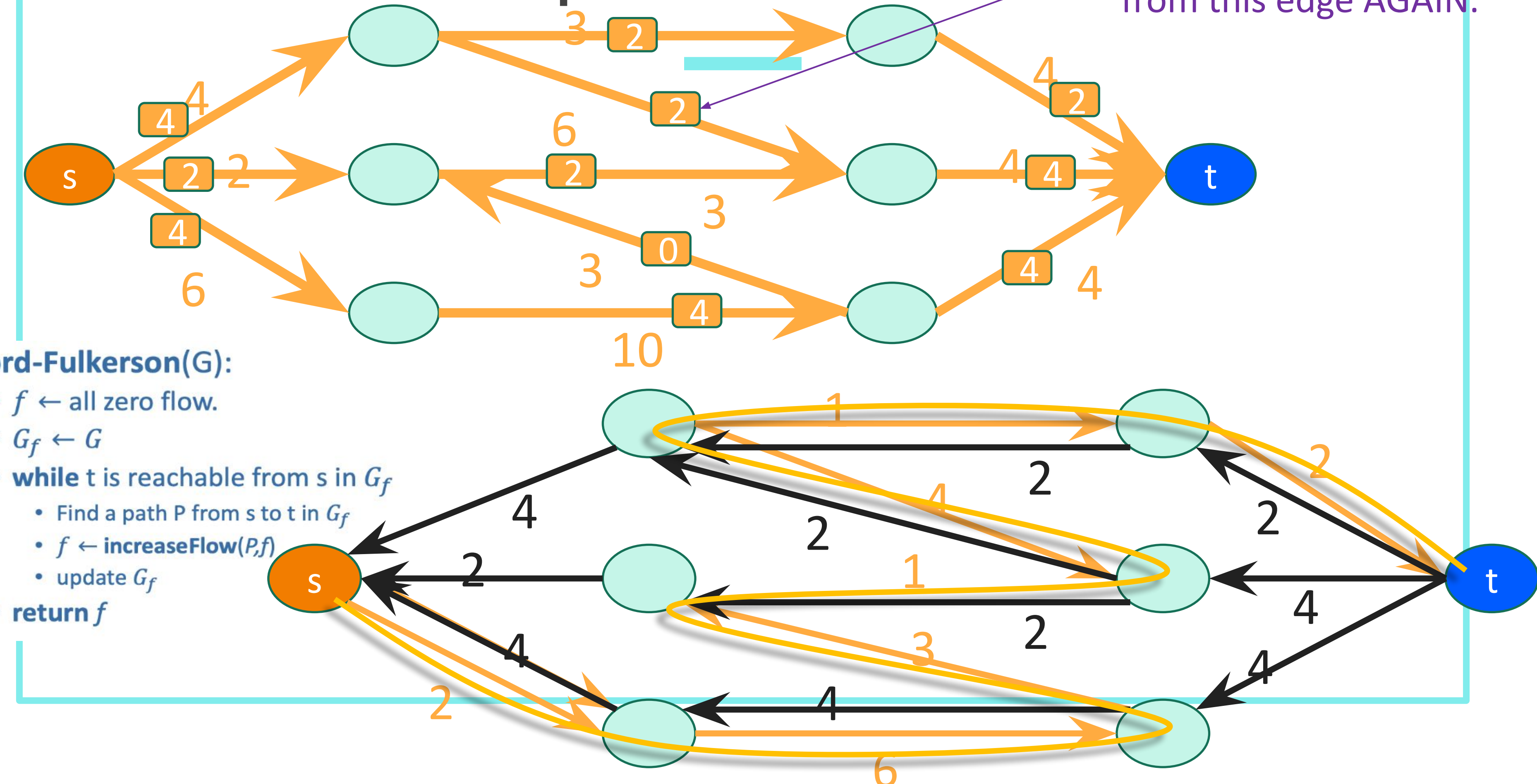
Notice that we're going back along one of the backwards edges we added.

- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



Example of Ford-Fulkerson

We will remove flow from this edge AGAIN.

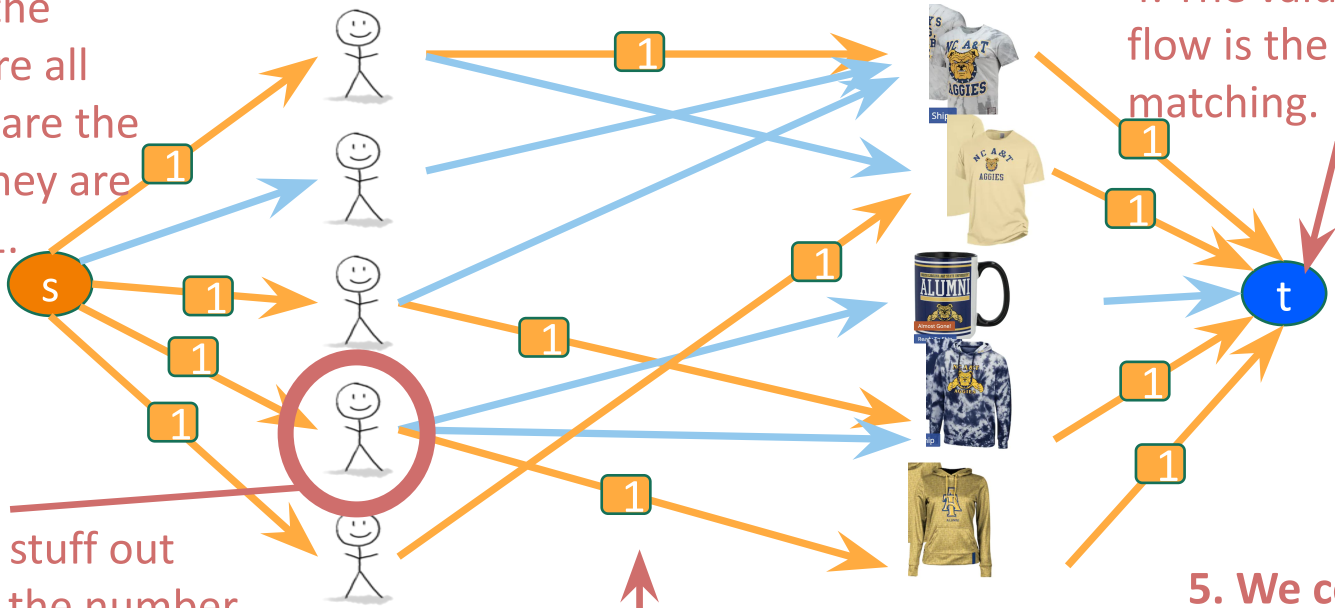


- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f

Solution via Max Flow: Why does it work?

All edges have capacity 1.

1. Because the capacities are all integers, so are the flows – so they are either 0 or 1.



4. The value of the flow is the size of the matching.

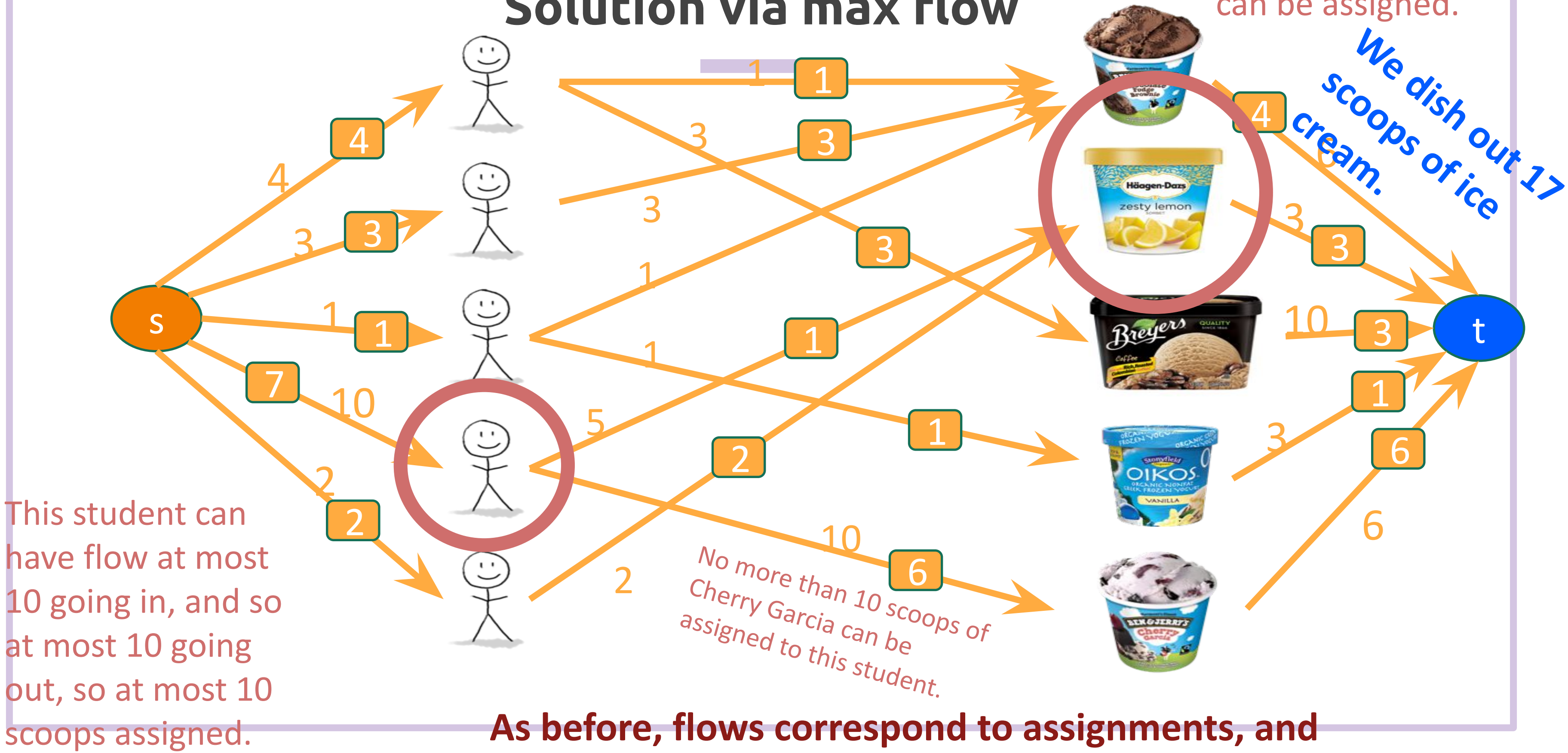
Value of this flow is 4.

2. Stuff in = stuff out means that the number of items assigned to each student 0 or 1. (And vice versa).

3. Thus, the edges with flow on them form a matching. (And, any matching gives a flow).

5. We conclude that the max flow corresponds to a max matching.

Solution via max flow



As before, flows correspond to assignments, and max flows correspond to max assignments.

Big Questions!

- What are greedy algorithms?
- What's an example of a greedy algorithm?
-
- When to use a greedy approach?

Big Questions!

- What are greedy algorithms?
- What's an example of a greedy algorithm?
-
- When to use a greedy approach?

Overview: What does “greedy” mean?

- Always makes the choice that seems to be the best at that moment (locally-optimal choice) in the hope that this will lead to a good solution overall (globally-optimal solution).
- In other words, we don't really plan, and we don't really look back. We just assume picking the next "obvious" best thing will lead to an optimal result. The key is often in knowing how to pick the next "obvious" best thing.
- This approach only works for some problems, so there is a skill in recognizing when a greedy approach will work.
- It often involves explicitly sorting, or using a data structure with sort-like behavior (e.g. priority queue, max/min heap)
- Examples that we’ve seen:
  - Dijkstra’s: pick next univisted node that’s closest based on dist.
  - Prim’s: pick next cheapest unvisited node to visit.
  - Kruskal’s: pick next cheapest edge that doesn’t cause a cycle.

The Greedy Process

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

?!?!

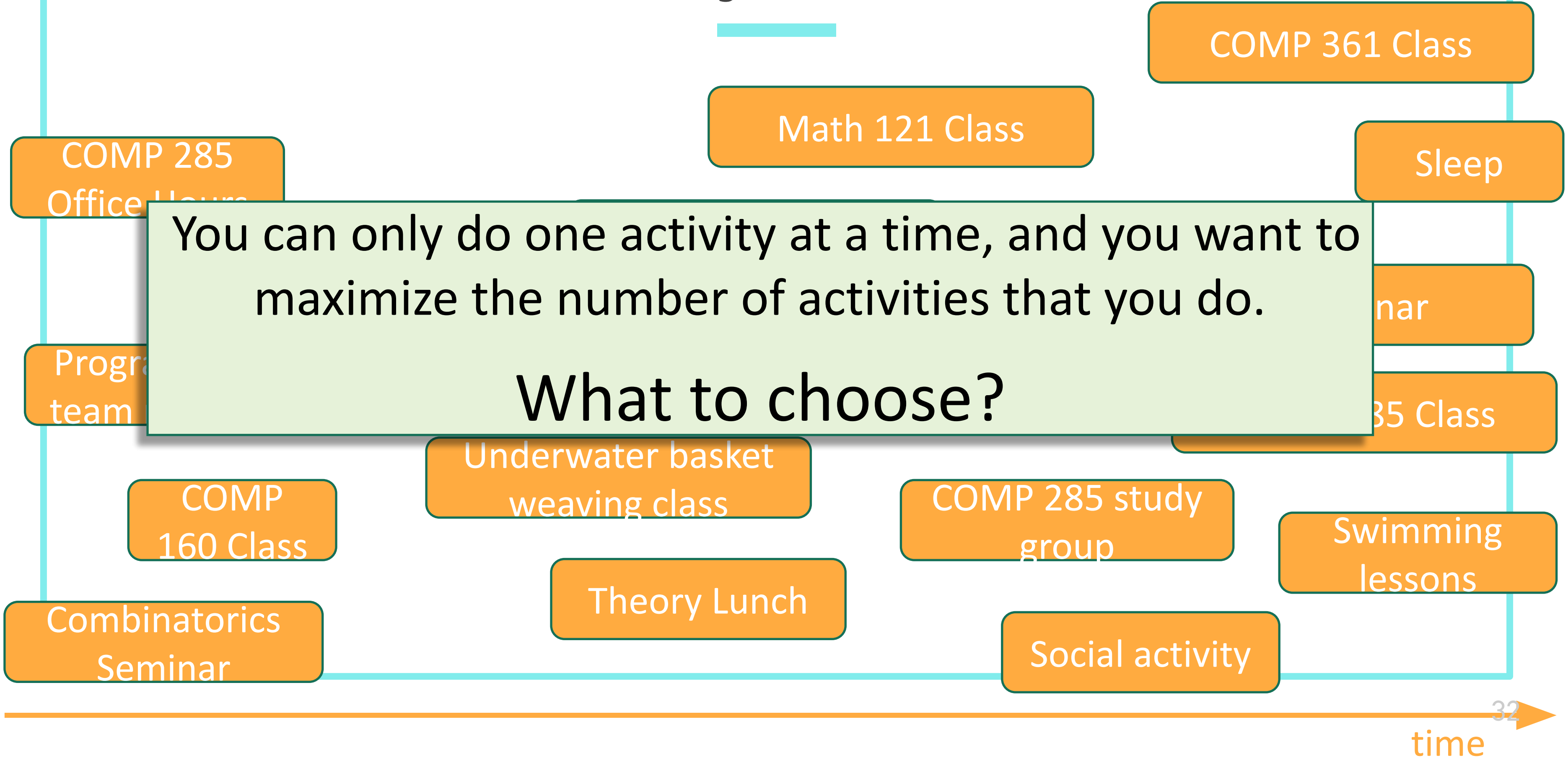
Big Questions!

- What are greedy algorithms?
- What's an example of a greedy algorithm?
-
- When to use a greedy approach?

Activity Selection

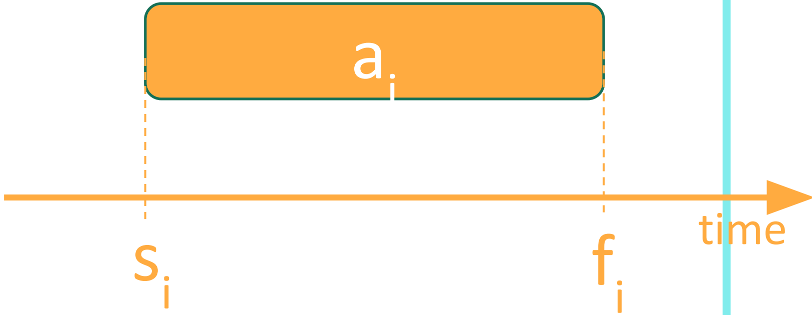
You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?



Activity Selection

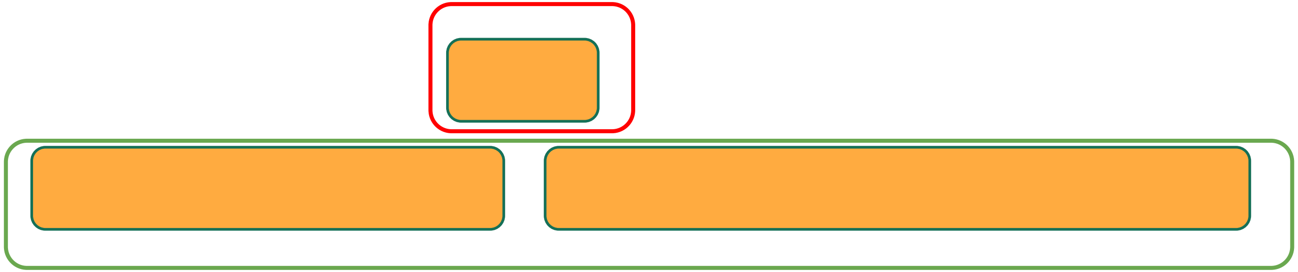
- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$



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

In what order should you greedily add activities?

What about shortest activity time?



We should select 2… but we'd pick 1.

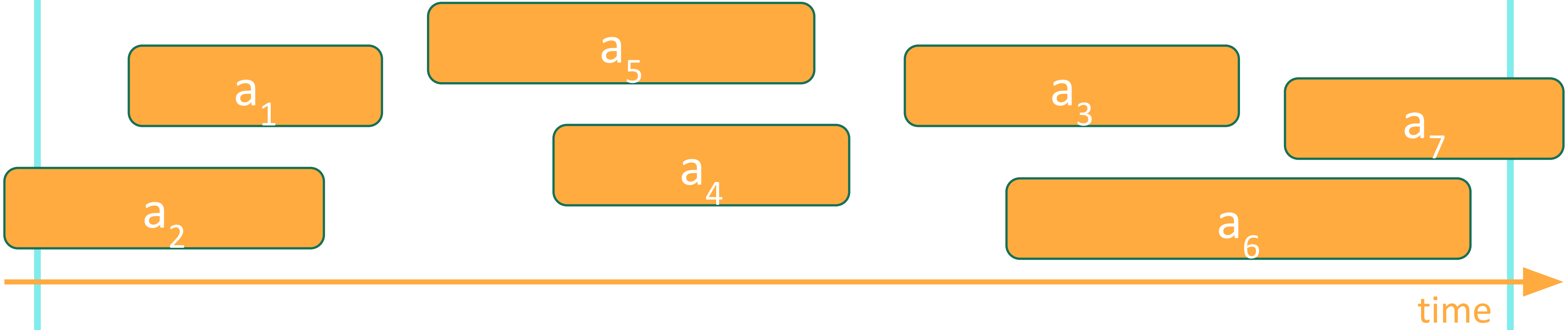
What about fewest conflicts?

????

What about sorting by ending time?

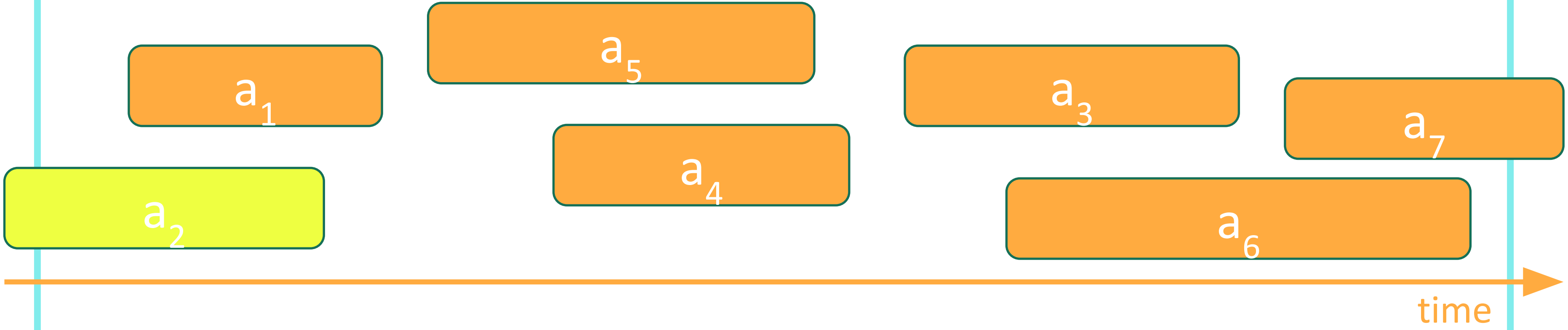
!!!!!

Greedy Algorithm



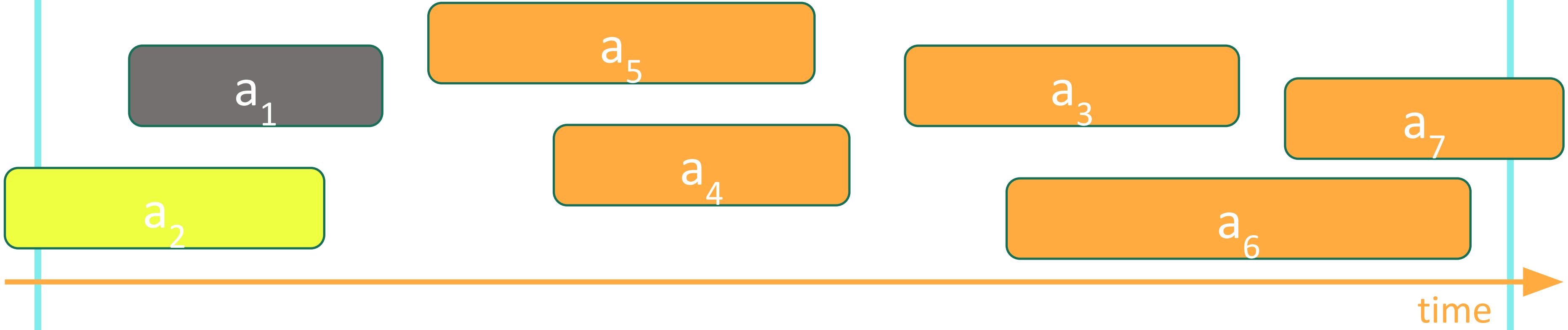
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



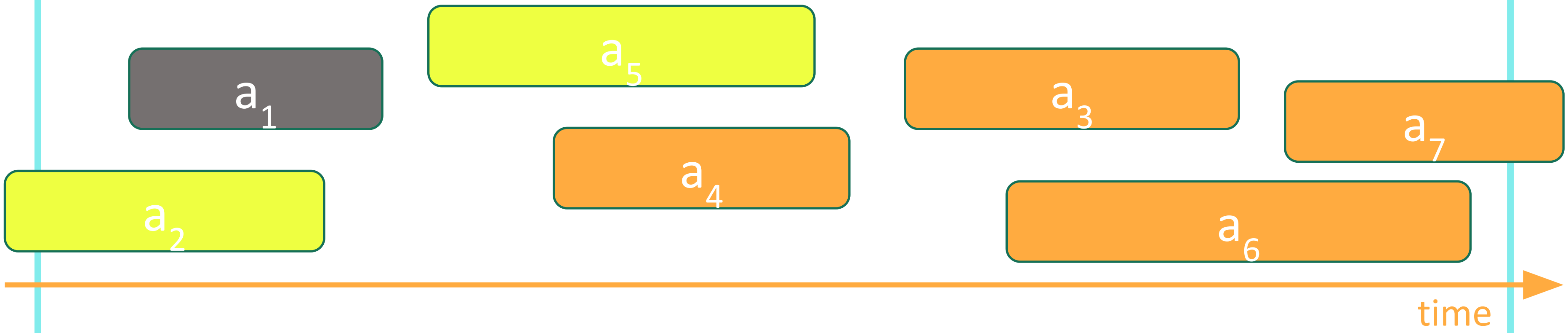
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



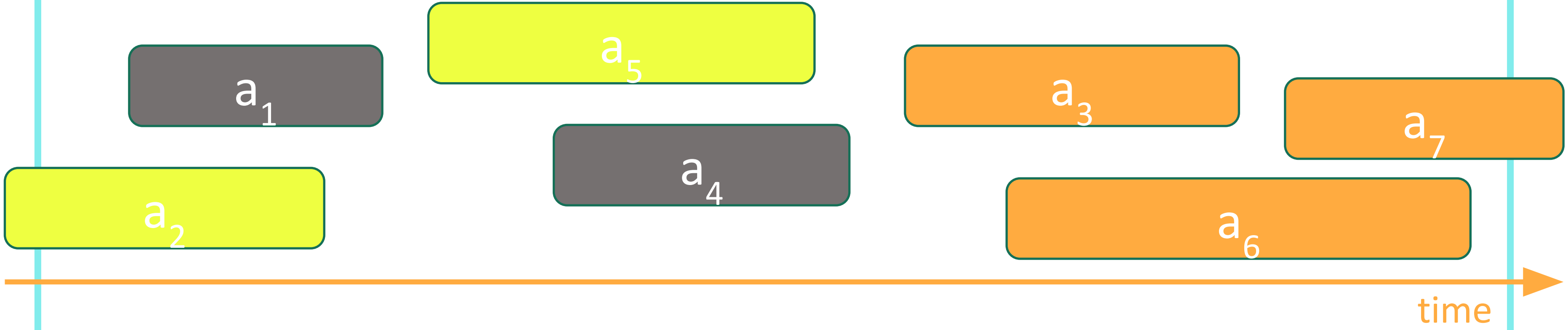
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



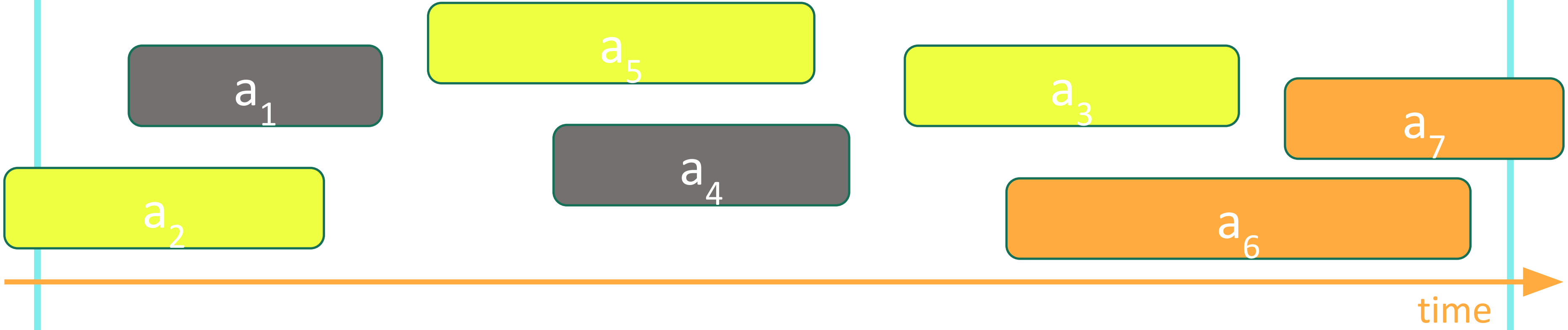
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



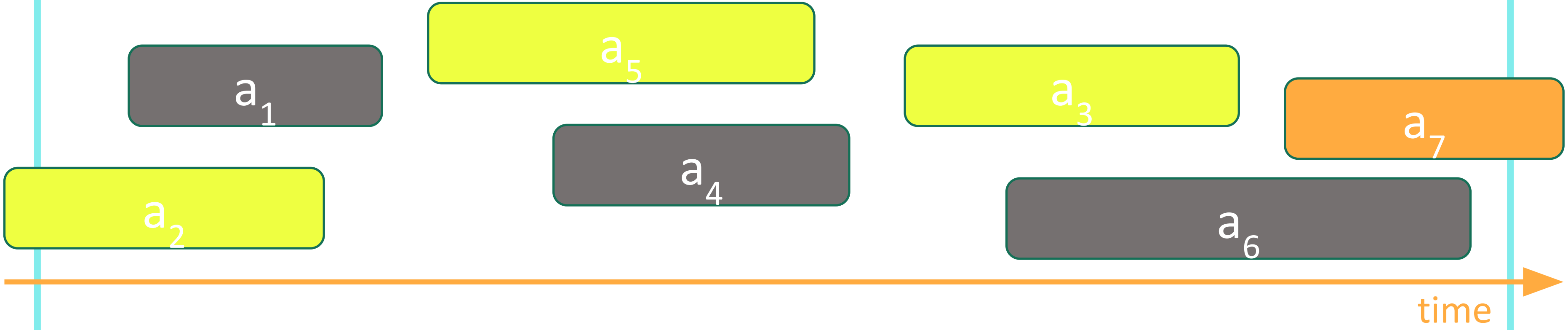
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



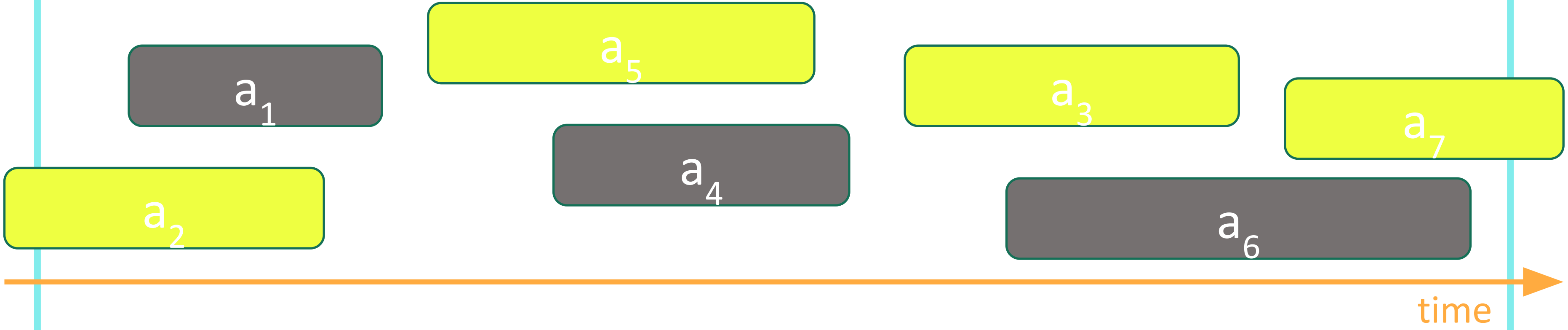
- Pick activity you can add with the smallest finish time.
- Repeat.

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.

At least it's fast

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

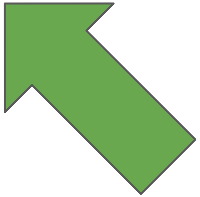
What makes it greedy?

- At each step in the algorithm, we 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.

Big Questions!

- What are greedy algorithms?
- What's an example of a greedy algorithm?
-
- When to use a greedy approach?



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.

Making Change

- Sometimes, the smallest difference in a problem can mean it can or cannot be solved using a greedy approach.
- Problem 1: A vending machine stocks pennies (1c), nickels (5c), and quarters (25c). What is the fewest number of coins that must be dispensed to return exactly N cents to the customer?
- Problem 2: A vending machine stocks pennies (1c), dimes (10c), and quarters (25c). What is the fewest number of coins that must be dispensed to return exactly N cents to the customer?

Making Change Problem #1

Problem 1: A vending machine stocks pennies (1c), nickels (5c), and quarters (25c). What is the fewest number of coins that must be dispensed to return exactly N cents to the customer?

Say N = 31, what is our optimal # of coins?

    3

What is our greedy approach here?

    Use the biggest coin as much as you can, then move on

Let’s code it!!!

Making Change Problem #2

Problem 2: A vending machine stocks pennies (1c), dimes (10c), and quarters (25c). What is the fewest number of coins that must be dispensed to return exactly N cents to the customer?

What value of N would make our greedy approach fail?

Why does this not work while our previous one did?

In the previous example, all coins are divisible by each other, so it’s always best to use less coins to achieve the same value.

Takeaways

- Greedy algorithms pick the next "obvious" thing locally to build up our solution
- For some problems, Greedy approaches provide the global optimal solution.
- We'll often be working through our data in a sorted way.
- To show a Greedy approach <u>will not</u> work, come up with a counterexample.
- To show when it <u>will</u> work requires making a formal proof not covered in this class, but you can still build confidence with intuition and examples.

COMP - 285

Advanced Algorithms

Welcome to COMP 285

Lecture 19: Greedy Algorithms

Lecturer: Chris Lucas (cflucas@ncat.edu)