Lecture 5

Topological sort and greedy algorithms

Chinmay Nirkhe | CSE 421 Spring 2025

Administrativa

- Problem set 1 is due tonight! Problem set 2 is uploaded.
- I have second office hours immediately after this lecture in CSE2 353.
- Many questions on EdStem about how much detail to include.
    - https://courses.cs.washington.edu/courses/cse421/25sp/psets.html
    - We have written a whole document about this
- Example solutions from section are also a good start

Directed acyclic graphs

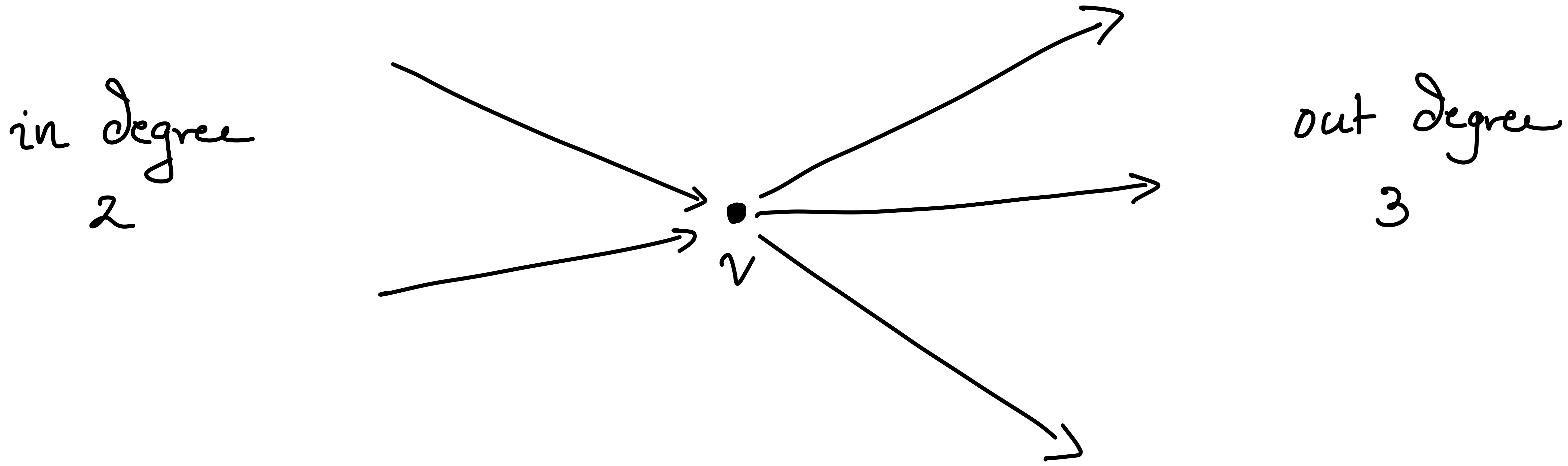
- A directed graph G is *acyclic* iff it has no directed cycles
- Also referred to as a DAG
- If we shrink every strongly connected component to a vertex, this converts the directed graph into a DAG

Topological sorting of graphs

- **Input:** a directed acyclic graph DAG $G = (V, E)$
- **Output:** An injective numbering $N : V \hookrightarrow \{1,\ldots,n\}$ such that edges only go from lower numbered to higher numbered vertices.

  i.e. for $u \to v$, we must have $N(u) < N(v)$.
- **Applications**
  - Vertices represents tasks and edges represent prerequisites
  - Topological sorts gives a sequential ordering for how to solve the system
- For general graphs, generate DAG by shrinking SCCs and then process SCCs in the order given by topological sort. Cannot number the vertices within a SCC.

In-degree and out-degree



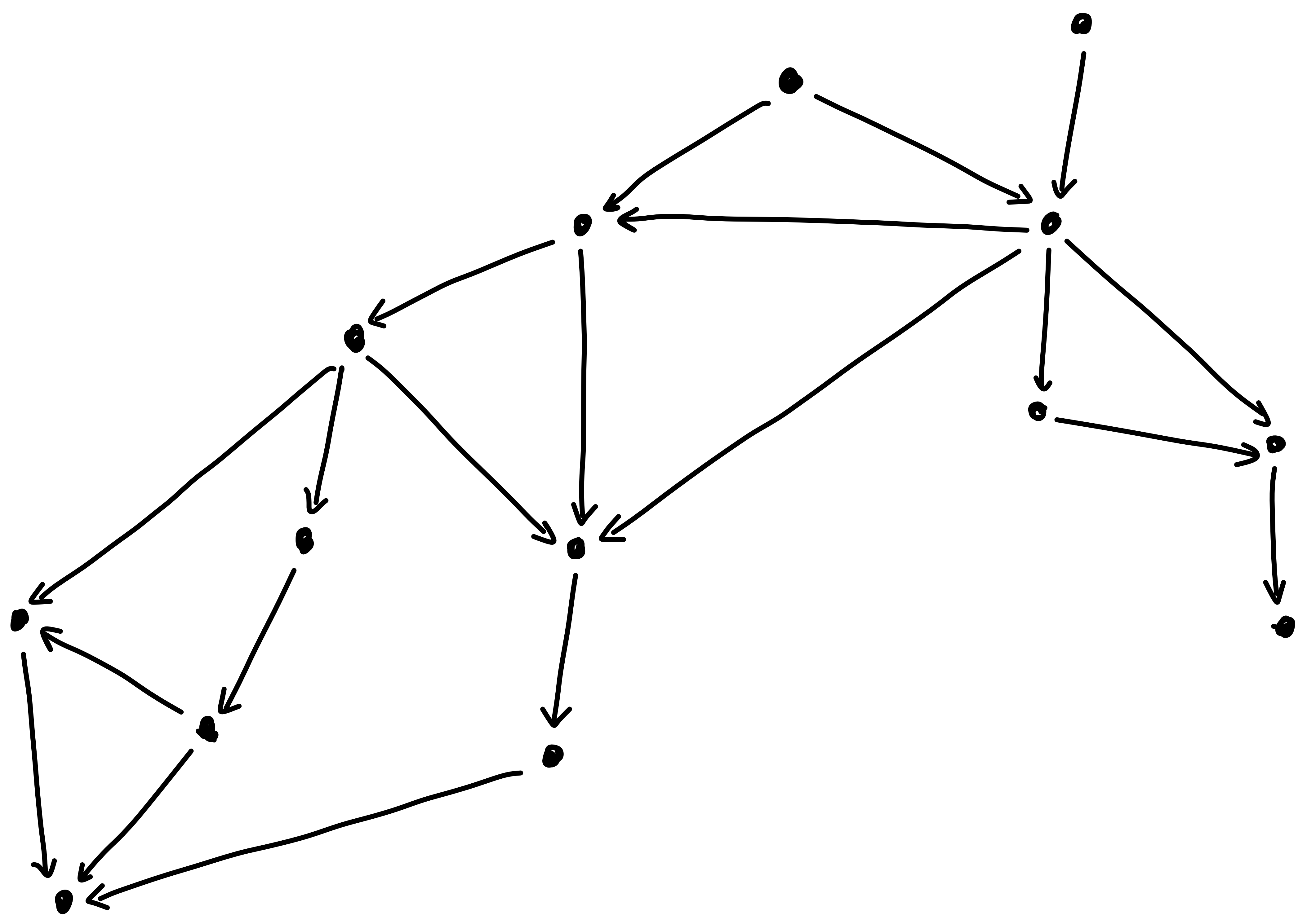
In-degree zero vertices

- **Claim:** Every DAG has at least one vertex of in-degree 0.
- **Proof:**
  - Assume every vertex has in-degree ≥ 1.
  - Starting with any vertex v pick an in-edge $u \to v$ and go in reverse to u. Repeat.
  - Since there are only n vertices, eventually a vertex will be repeated. This means there is a cycle, a contradiction.

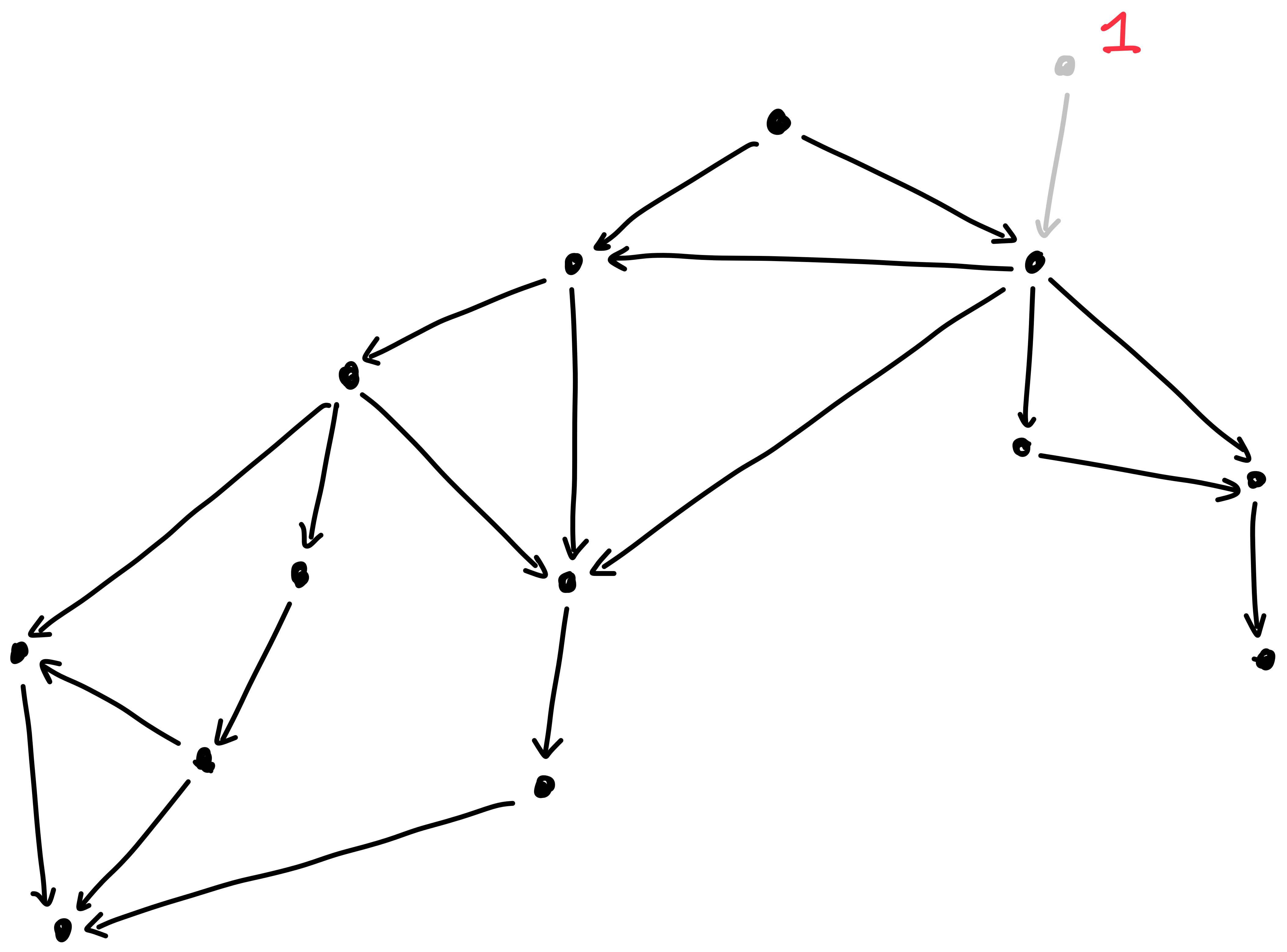
Algorithm for topological sort

- Any vertex v_1 of in-degree 0 can be numbered as 1
- **Topological sorting algorithm:**
    - If we remove v_1 and assign $N(v_1) = 1$, then the rest is still a DAG
    - Then, there is a new vertex v_2 of in-degree 0
    - Repeat, until all vertices are exhausted

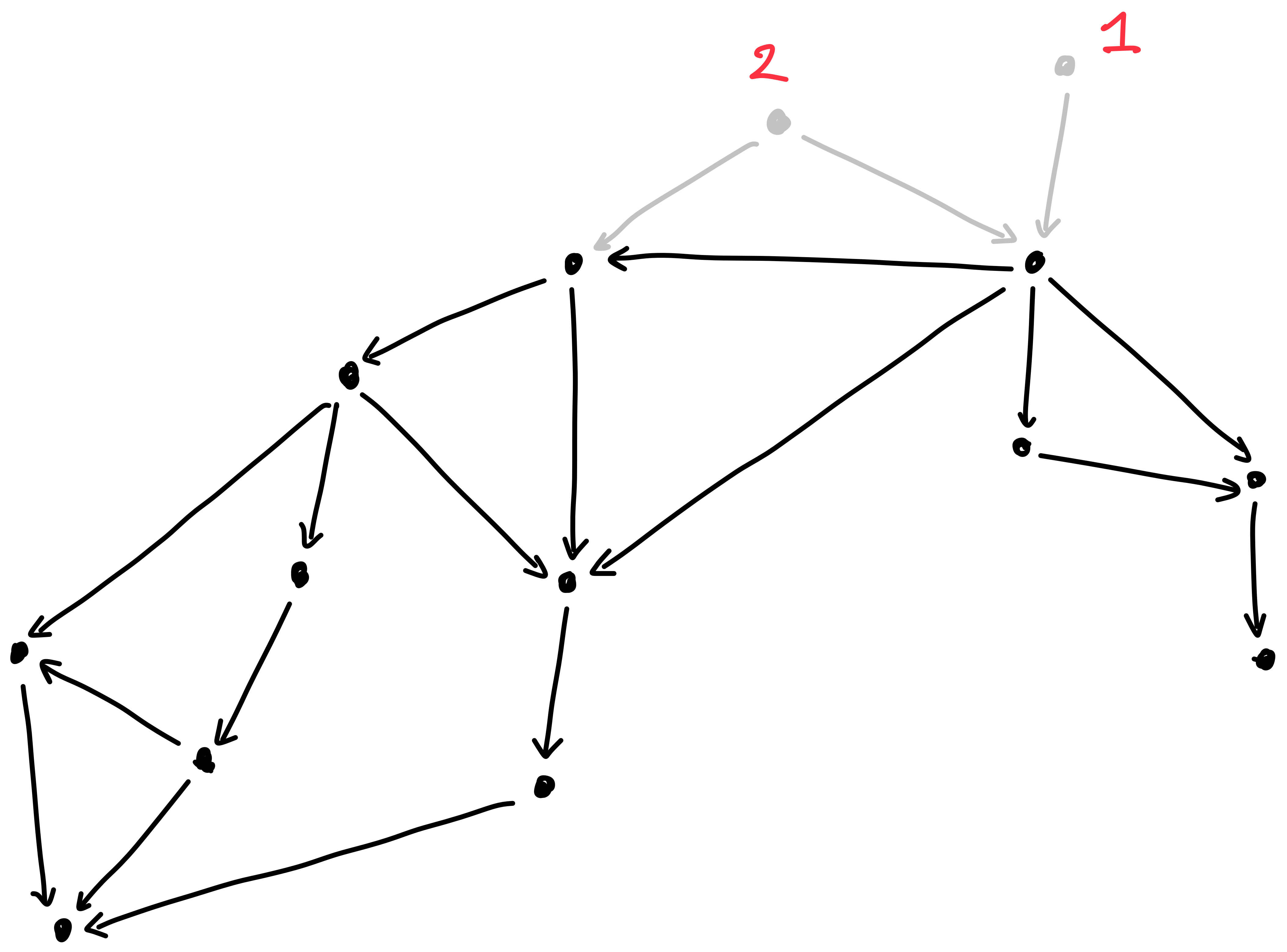
Implementing topological sort



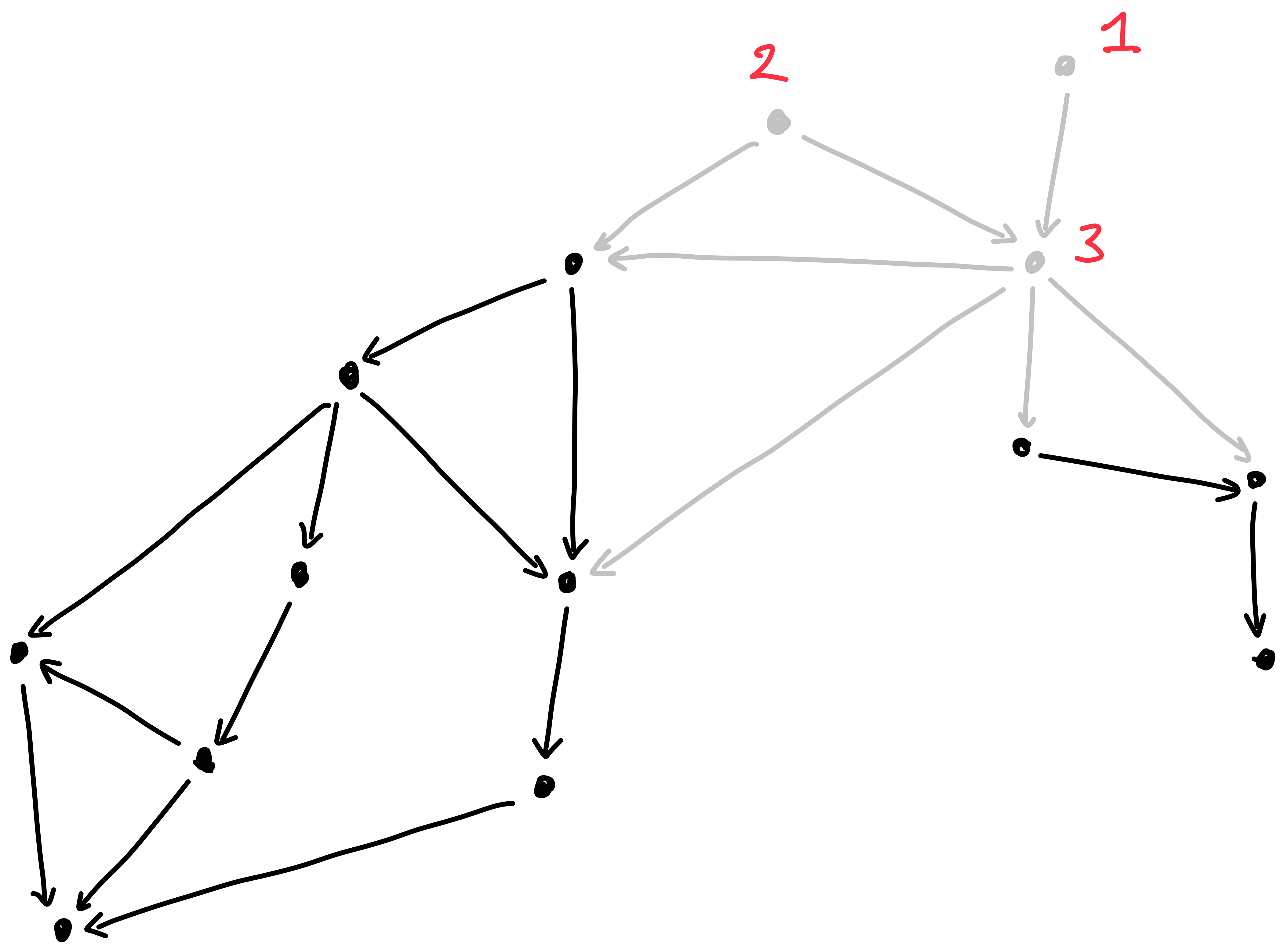
Implementing topological sort



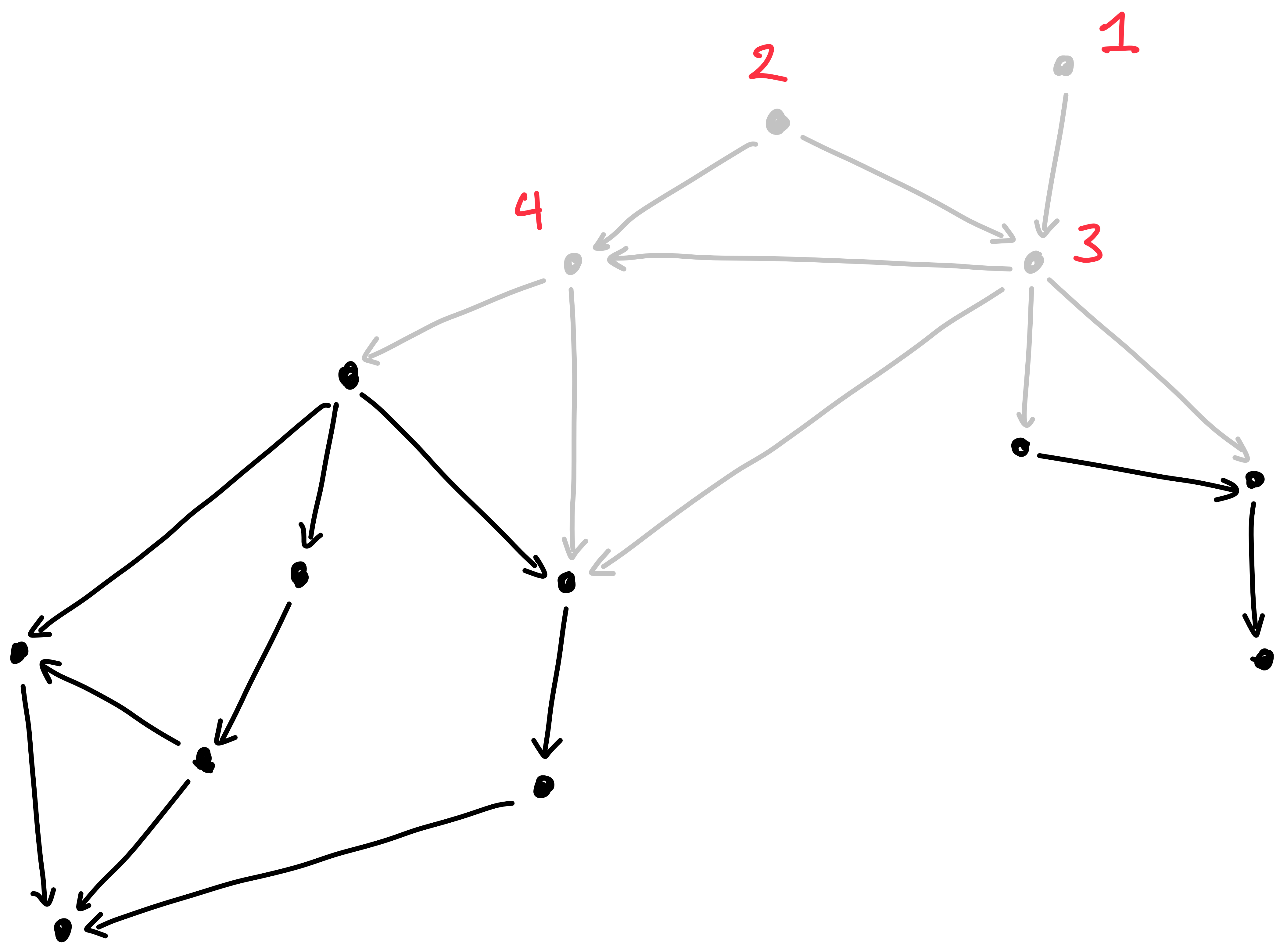
Implementing topological sort



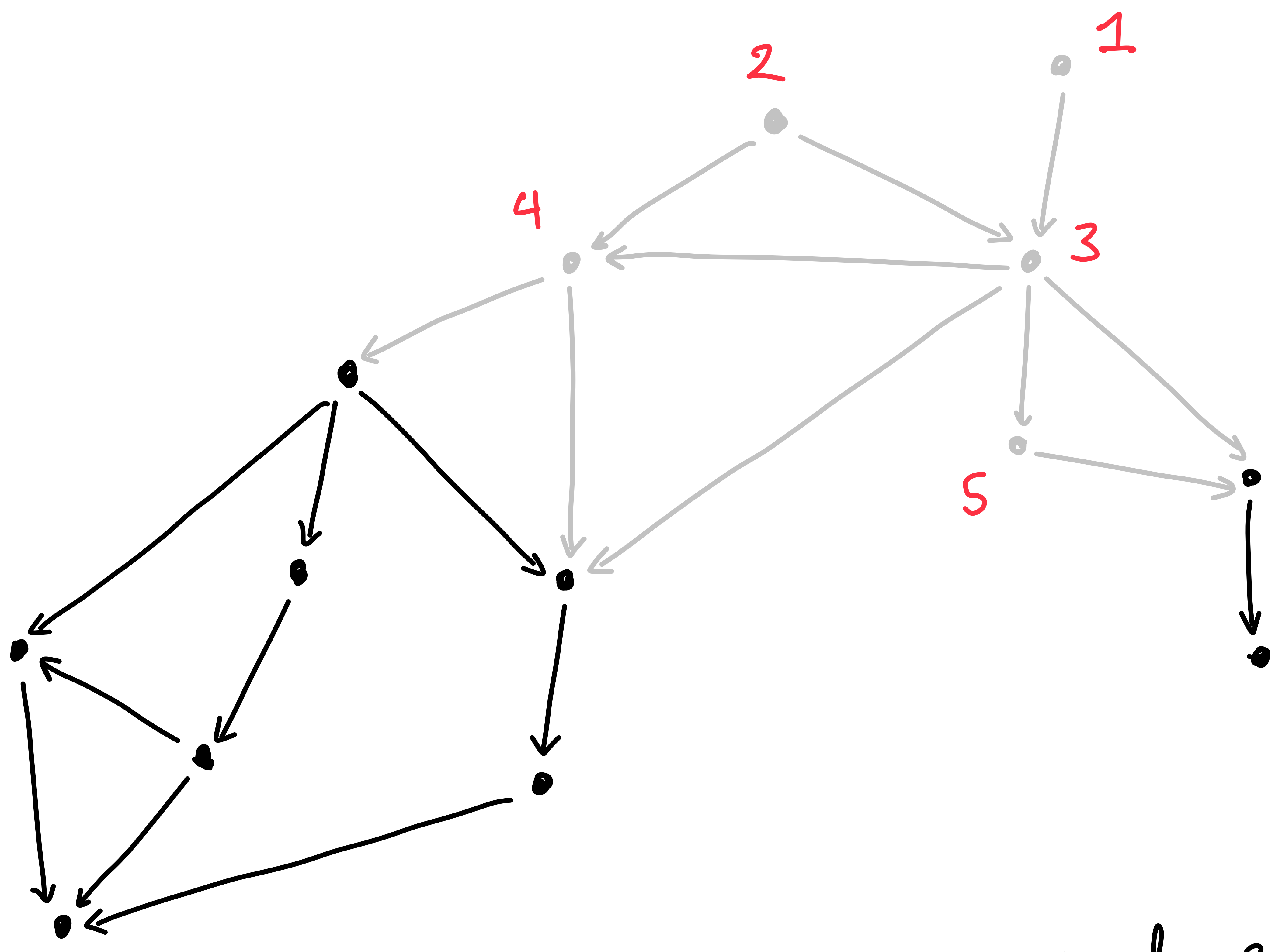
Implementing topological sort



Implementing topological sort



Implementing topological sort



and so on...

Implementing topological sort

- Issue is finding the next vertex that has in-degree 0. Can be algorithmically slow.
- Observe that when we remove the vertex v_j, the in-degree of only the out-neighbors of v_j will decrease.

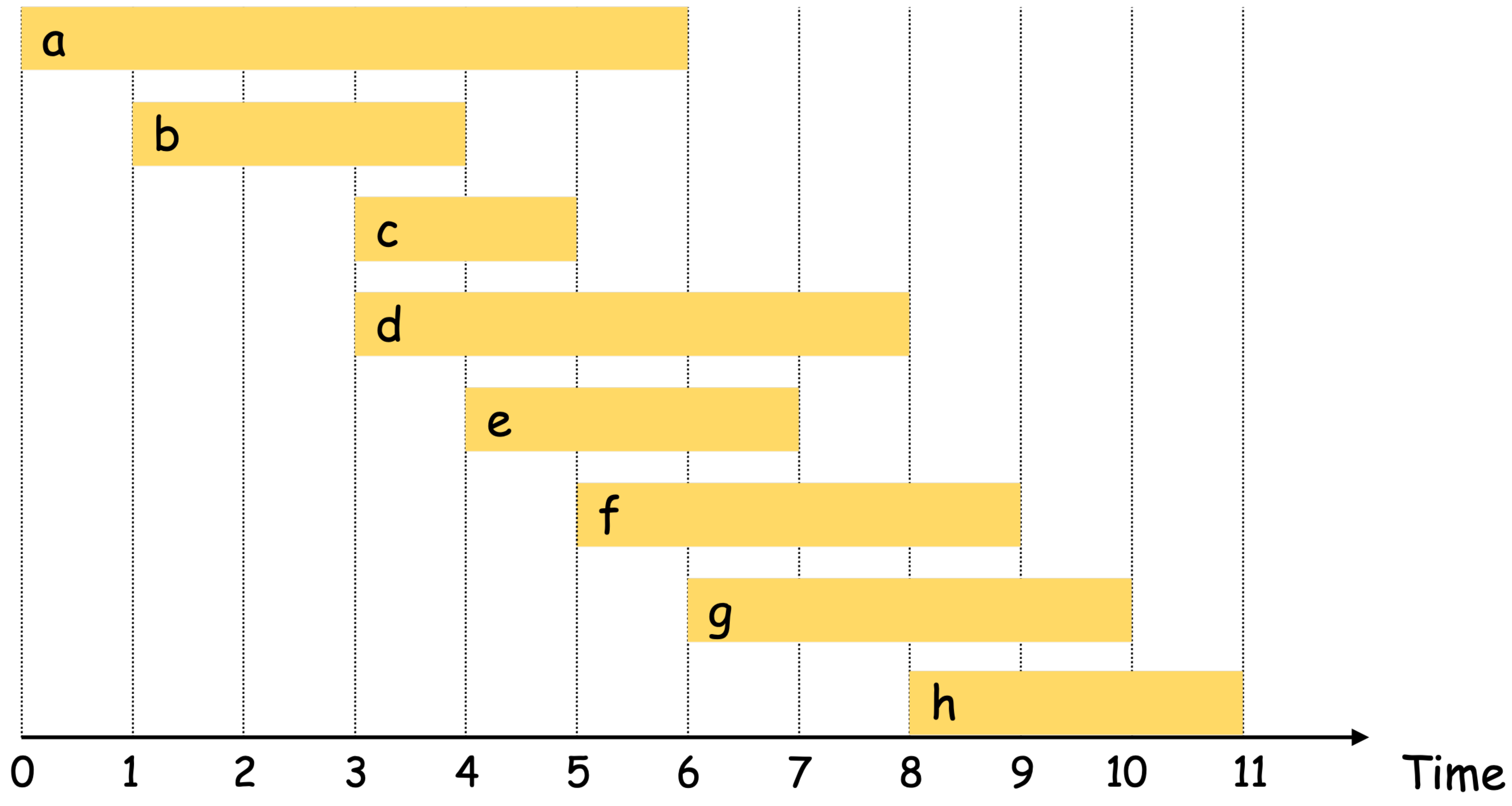
Implementing topological sort

- **Algorithm:**
  - Iterate through all vertices and set $d(v) =$ in-degree of each vertex. Initialize queue Q with vertices such that $d(v) = 0$. Set $j \leftarrow 1$.
  - While Q is non-empty, pop vertex u off queue
    - Set $N(u) \leftarrow j$. $j \leftarrow j + 1$.
    - Decrease $d(v) \leftarrow d(v) - 1$ for every nbhr. v s.t. $u \rightarrow v$. If $d(v) = 0$, add v to Q.
  - **Runtime:** Each edge is visited only once. So $O(n + m)$ time.

Previously in CSE 421…

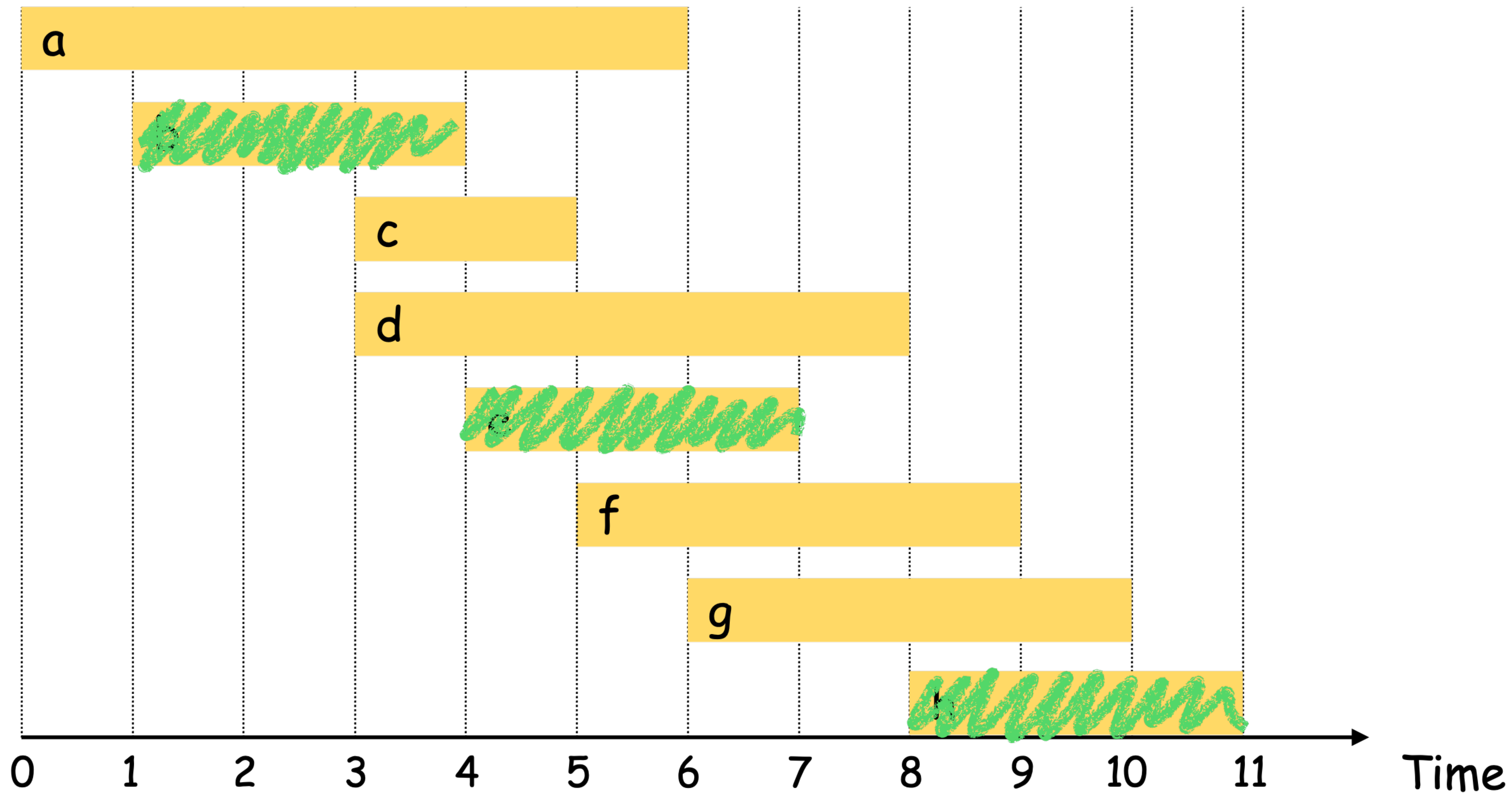
Interval scheduling

- **Input:** start and end times (s_i, t_i) for $i = 1,\ldots,n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs



Greedy algorithms for interval scheduling

- **Algorithm:** Select the job with earliest ending t_i of jobs not selected.
- **Example:**



Today

The principle of greedy algorithms

- Solving the *optimization problem* will require making many decisions (such as whether to include or not a job in the schedule)
- In a greedy algorithm, we make each decision locally without looking as to how it will effect future decisions
- Not every greedy criteria for making decisions works
    - It’s not obvious which criteria will work
    - We will focus on methods for proving that greedy algorithms do work

Greedy algorithms for interval scheduling

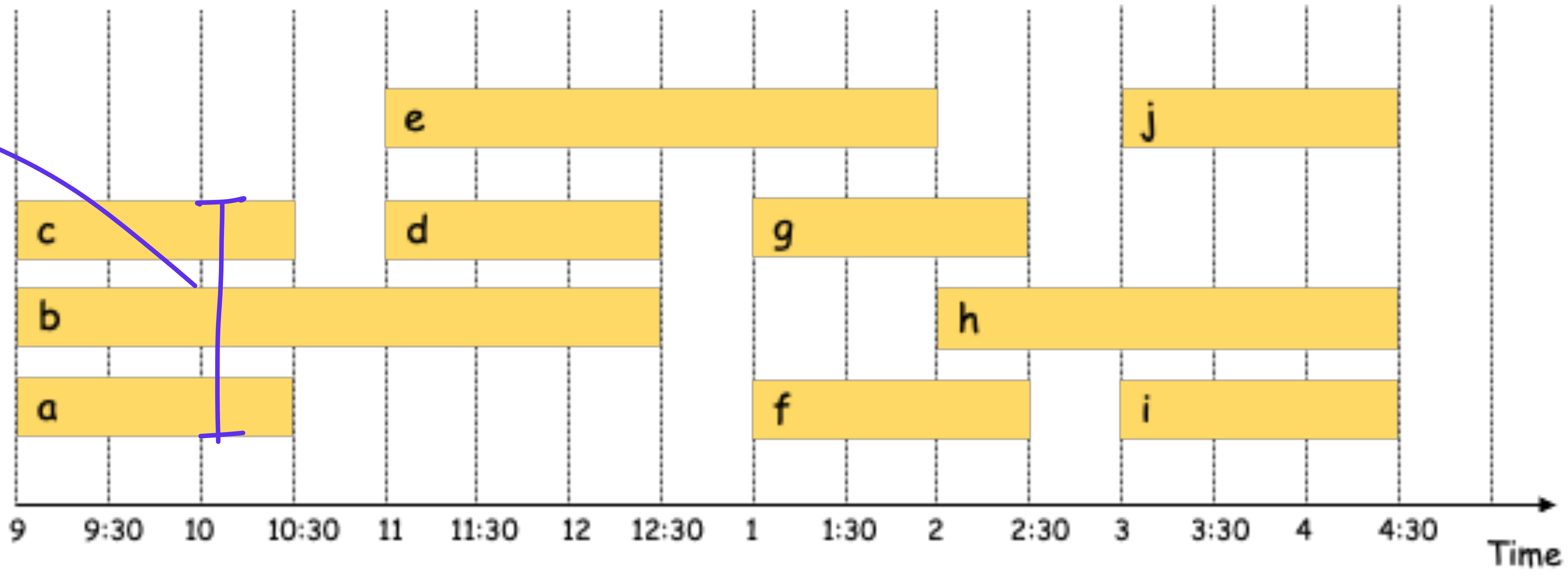
- **Input:** start and end times (s_i, t_i) for $i = 1, \ldots, n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs
- **Algorithm:** Select the job with earliest ending t_i of jobs not selected.
  - **Details:** Sort the jobs by earliest end time t_i. Keep track of current end time of selected jobs T. Add new job (s_i, t_i) if $s_i \geq T$ and update $T \leftarrow t_i$.
  - **Runtime:** Sorting + linear time to create list of jobs. $O(n \log n) + O(n) = O(n \log n)$.

Scheduling all intervals

- **Input:** (s_i, t_i) for $i = 1, \ldots, n$ for n "jobs" each using 1 room.
- **Output:** A scheduling of **all** jobs to rooms using the *minimum number* of rooms so that no two use the same room at the same time.

depth = 3

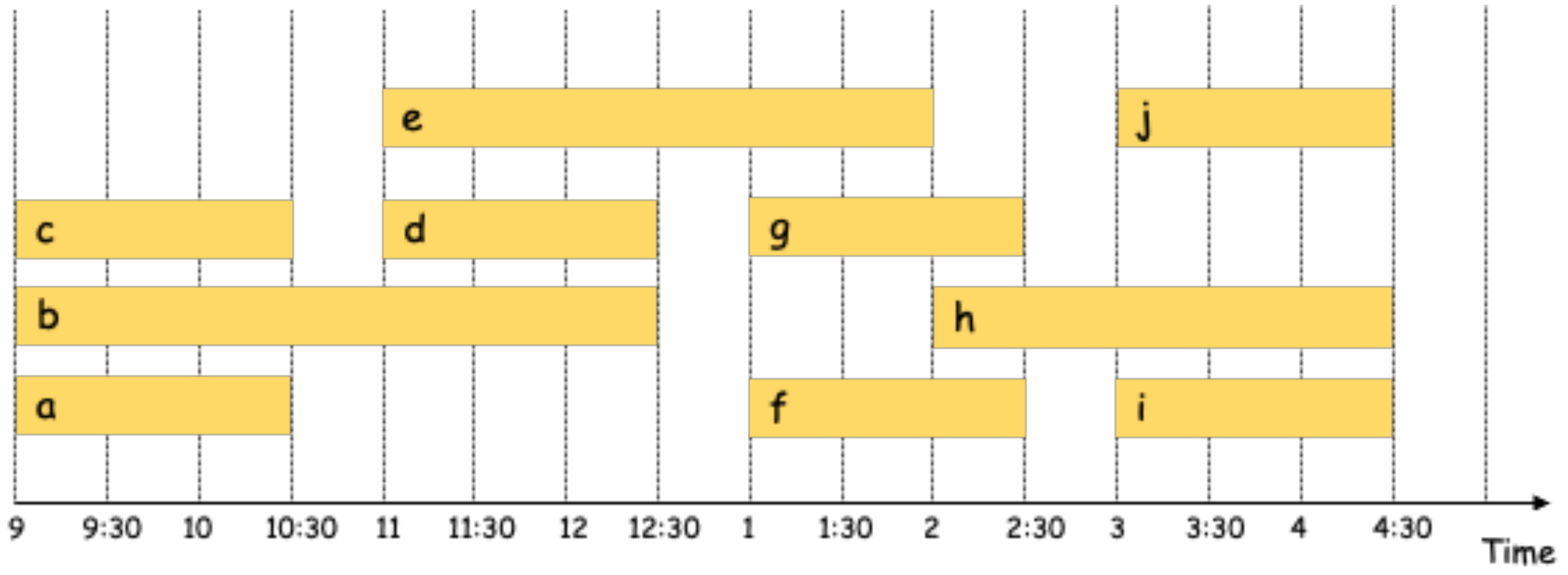
An easy minimum on the number of rooms required.



Scheduling all intervals

A greedy algorithm

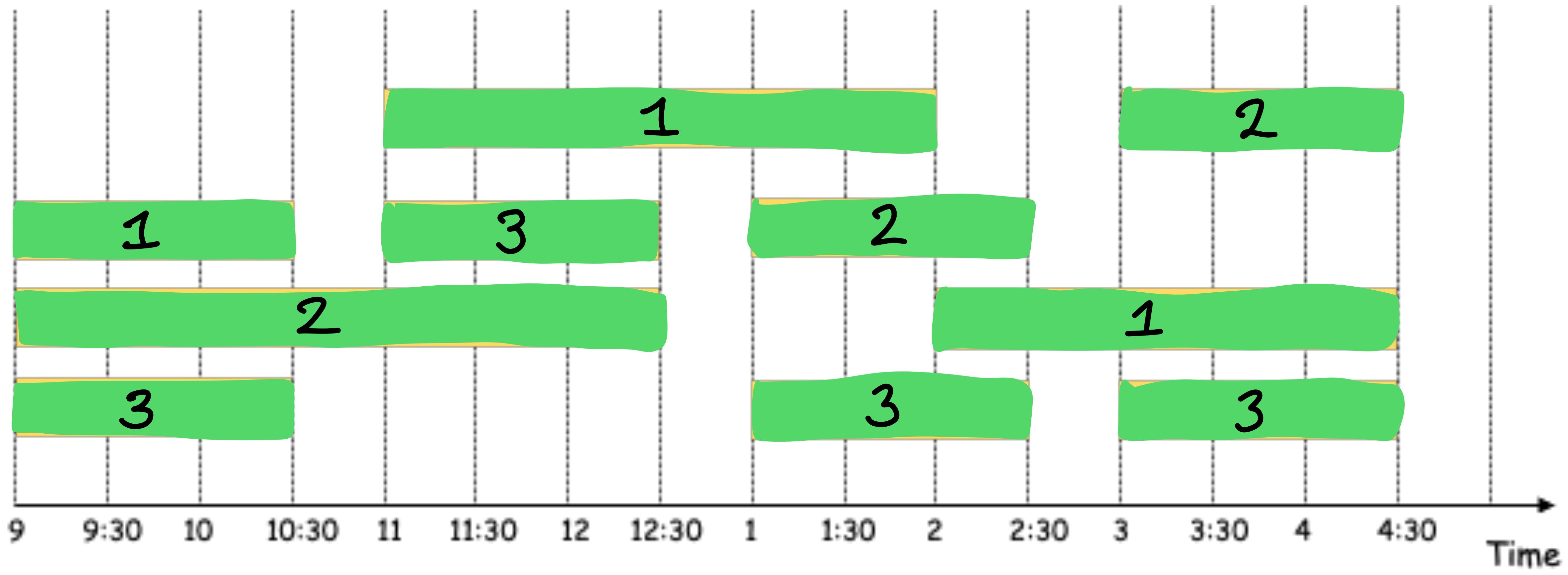
- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
    - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order.
    - Initialize an n sized array $\text{last}(j)$ as zeroes and an n sized array $r(j)$.
    - For $i \leftarrow 1$ to n
        - Find the first j such that $s_i \geq \text{last}(j)$.
        - Then set $\text{last}(j) \leftarrow t_i$ and set $r(i) \leftarrow j$.
    - Return assignment function r.



Scheduling all intervals

A greedy algorithm

- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order.
  - Initialize an n sized array $\text{last}(j)$ as zeroes and an n sized array $r(j)$.
  - For $i \leftarrow 1$ to n
    - Find the first j such that $s_i \geq \text{last}(j)$.
    - Then set $\text{last}(j) \leftarrow t_i$ and set $r(i) \leftarrow j$.
  - Return assignment function r.



Scheduling all intervals

Proof of correctness

- **Theorem:** The greedy algorithm is optimal.
- **Proof:**
  - Consider when a new room j is "allocated" for the **first** time. Let job i be the reason.
  - Then, $s_i \geq \mathrm{last}(j')$ for all $j' < j$.
  - Since $\mathrm{last}(j')$ denotes when that room will free up, the i-th job is incompatible with the jobs currently in the other $j - 1$ rooms.
  - Since we sort requests by start time, those jobs all started before s_i and haven't ended yet.
  - So there are j incompatible requests, requiring at least j rooms.

Scheduling all intervals

Runtime

- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order. ← $O(n \log n)$ time
  - Initialize an n sized array $\text{last}(j)$ as zeroes and an n sized array $r(j)$.
  - For $i \leftarrow 1$ to n
    - Find the first j such that $s_i \geq \text{last}(j)$.
    - Then set $\text{last}(j) \leftarrow t_i$ and set $r(i) \leftarrow j$.
  - Return assignment function r. ← $O(1)$

Loop runs $O(n)$ times.

could be slow. $O(n)$ each time.

Total: $O(n^2)$ due to loop.

Scheduling all intervals

Runtime

- **Greedy strategy:** Increment chronologically and open a new room if all rooms are currently full.
- **Algorithm:**
  - Sort requests by start time $s_1 \leq s_2 \leq \ldots \leq s_n$ in increasing order. Still $O(n \log n)$.
  - Initialize a **priority queue** Q, $k \leftarrow 0$ and an n sized array $r(j)$. Better data structure
  - For $i \leftarrow 1$ to n
    - Set $j \leftarrow \text{findmin}(Q)$. Now only $O(\log k)$ time.
    - If $s_i \geq \text{last}(j)$, schedule job i in room j: $\text{setkey}(j, Q) \leftarrow t_i$ and $r(i) = j$.
    - Else, allocate a new room $k \leftarrow k + 1$ and $\text{setkey}(k, Q) \leftarrow t_i$ and $r(i) = k$.

    Also $O(\log k)$ time.
  - Return assignment function r.

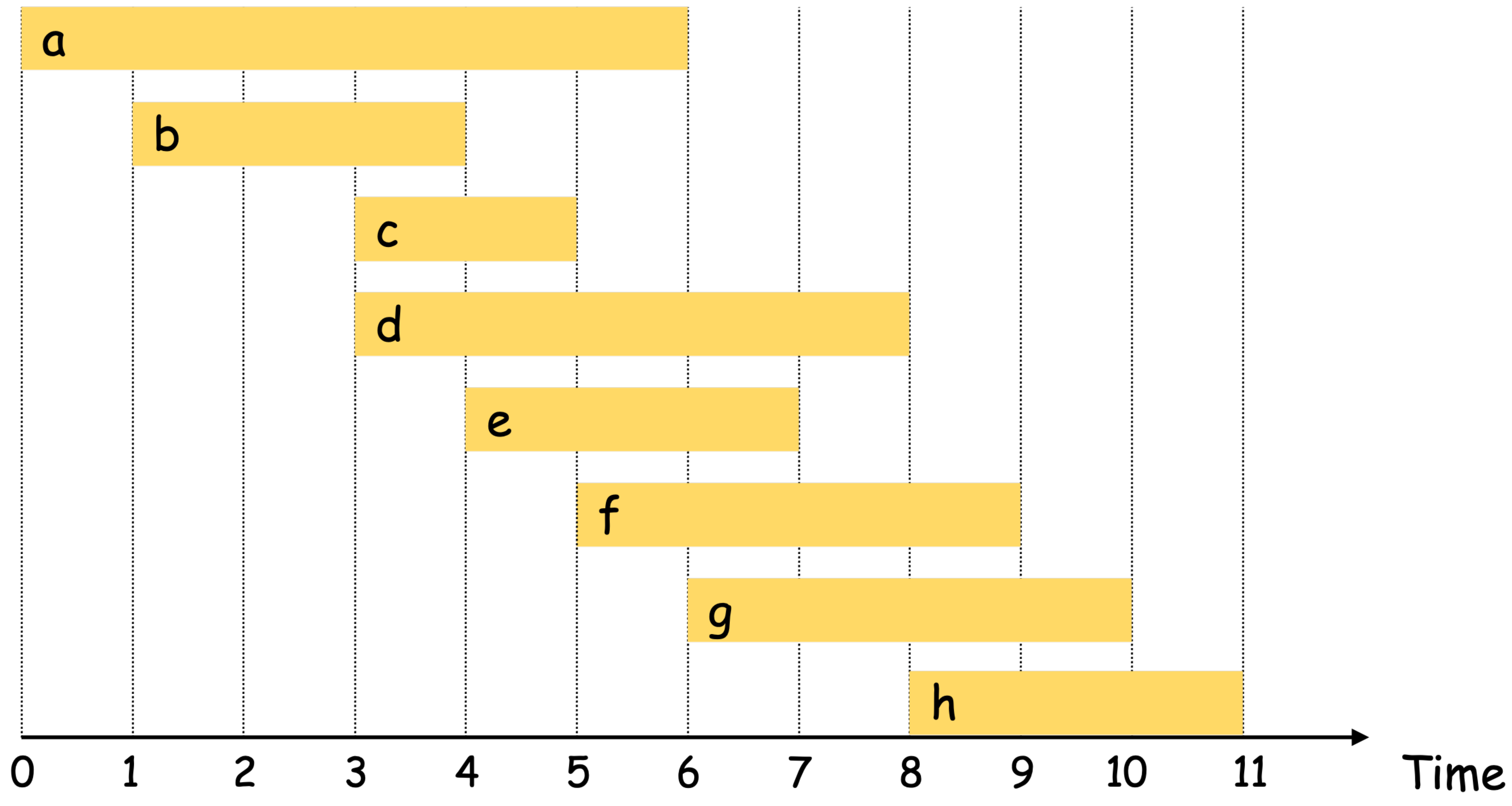
Total runtime: $O(n \log k)$ due to better data structure.

Greedy algorithm general strategies

- **Greedy algorithm stays ahead:** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithms
- **Structural:** Discover a structure-based argument asserting that the greedy solution is at least as good as every possible solution.
- **Exchange argument:** We can gradually transform any solution into the one found by the greedy algorithm with each transform only improving or maintaining the value of the current solution.

Interval scheduling

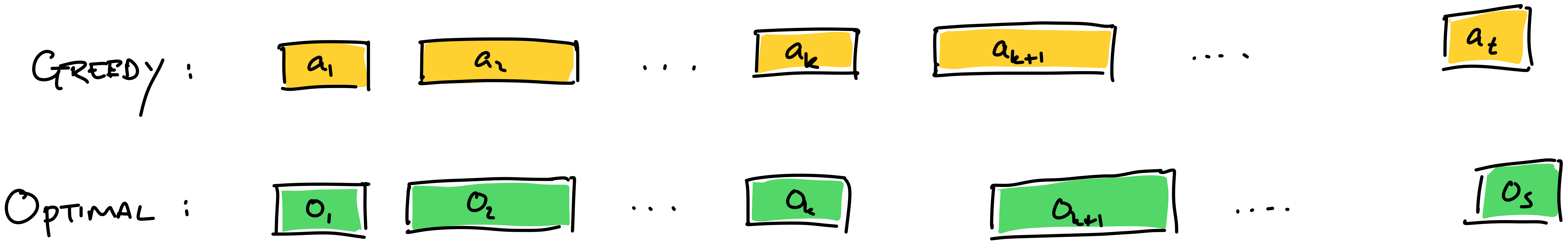
- **Input:** start and end times (s_i, t_i) for $i = 1,\ldots,n$ for n "jobs"
- **Output:** A maximal set of mutually compatible jobs



Greedy algorithm analysis

Contradiction argument edition

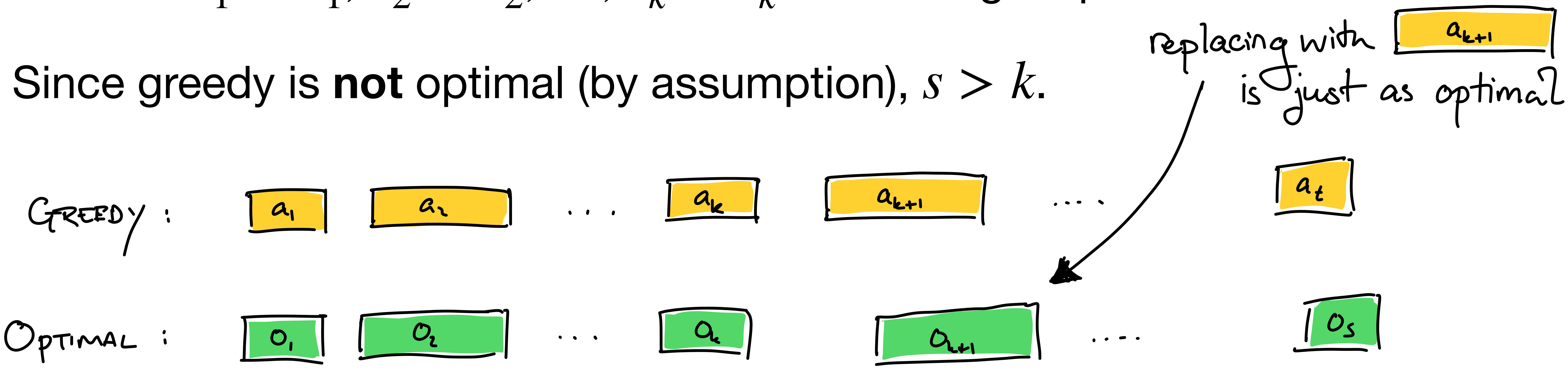
- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.



Greedy algorithm analysis

Contradiction argument edition

- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.

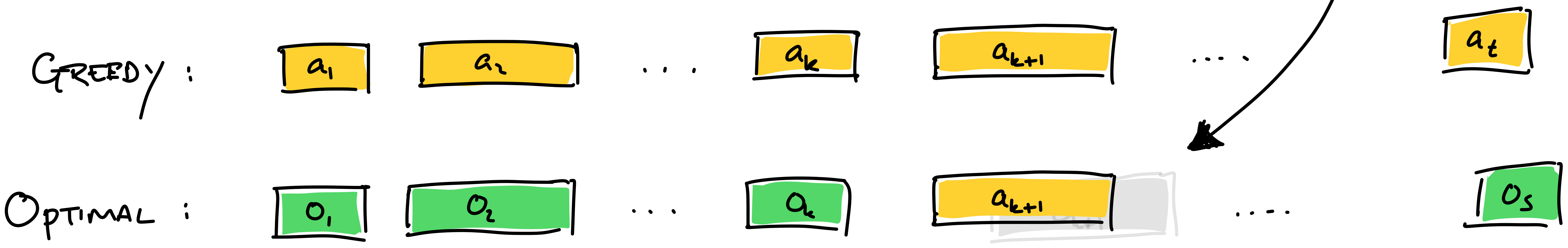


Greedy algorithm analysis

Contradiction argument edition

- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.

replacing with $\boxed{a_{k+1}}$ is just as optimal



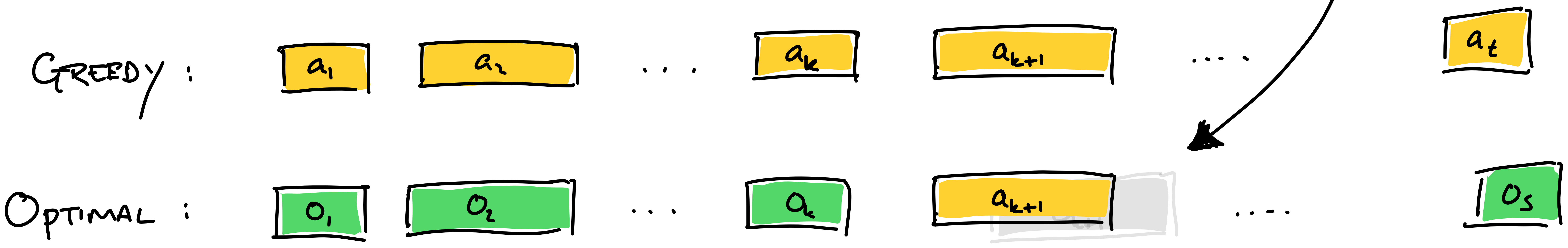
Greedy algorithm analysis

Contradiction argument edition

now we can induct.

- Let $a_1, a_2, \ldots, a_t$ denote the jobs selected by the greedy algorithm.
- Let $o_1, o_2, \ldots, o_s$ denote the jobs selected in an *optimal solution*.
- Assume $a_1 = o_1, a_2 = o_2, \ldots, a_k = o_k$ for the largest possible k.
- Since greedy is **not** optimal (by assumption), $s > k$.

replacing with a_{k+1} is just as optimal

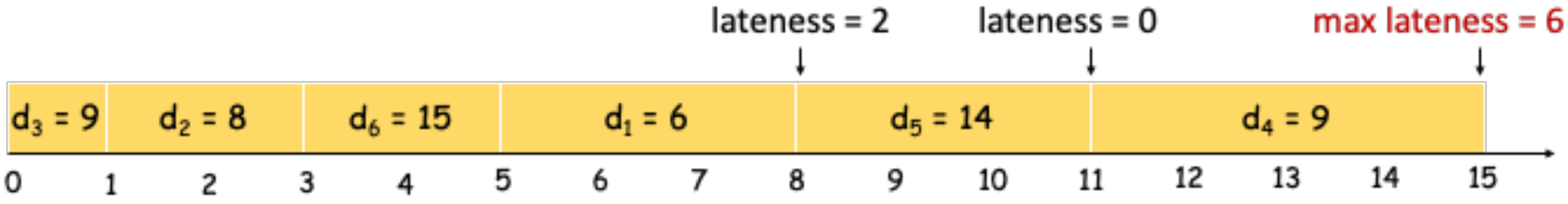


Minimizing lateness

- A new scheduling problem. There is a single resource but instead of start and finish times, each job i has
  - A time requirement τ_i which must be scheduled contiguously
  - A target deadline d_i by which the request is ideally finished
- Minimum start time is 0.
- Each item is graded a lateness: $\ell_i := \min\{0, t_i - d_i\}$ where t_i is it's end time
- Total lateness is defined as the **max** lateness: $L = \max_{i=1,\ldots,n} \ell_i$. crucially different than $L = \sum_{i=1..n} \ell_i$.
- **Goal:** Find a scheduling that *minimizes* the *maximum lateness* L.

Example minimizing lateness problem

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Finding the right greedy strategy

- Greedy template suggests finding a strategy and seeing if there are any glaring counterexamples.
  - **Shortest processing time.** Sort the jobs according to τ_i and select in order.
  - **Earliest deadline first.** Sort according to d_i and select in order.
  - **Smallest slack.** Sort according to slack, $d_i - \tau_i$, and select in order.

Counterexamples

Shortest processing time

- Sort the jobs according to τ_i and select in order.

	1	2
τ_i	1	10
d_i	100	10

Job 1 is selected due to shorter duration.

But then Job 2 incurs a lateness of 1.

Opposite order has 0 lateness.

Counterexamples

Smallest slack

- Sort according to slack, $d_i - \tau_i$, and select in order.

	1	2
τ_i	1	10
d_i	2	10

Job 2 has smaller slack.

Causes a lateness of $11 - 2 = 9$

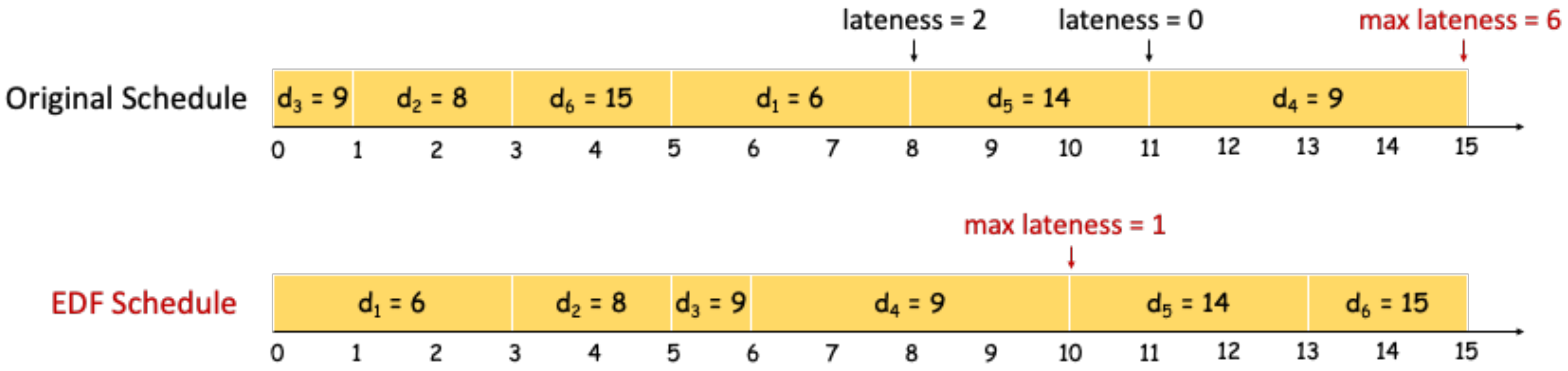
Other order has lateness of 1.

Earliest deadline first (EDF)

- **Algorithm:**
  - Sort deadlines in increasing order $d_1 \leq d_2 \leq \ldots \leq d_n$.
  - Set $T \leftarrow 0$.
  - For $i \leftarrow 1$ to n
    - $(s_i, t_i) \leftarrow (T, T + \tau_i)$
    - $T \leftarrow T + \tau_i$.

Example EDF schedule

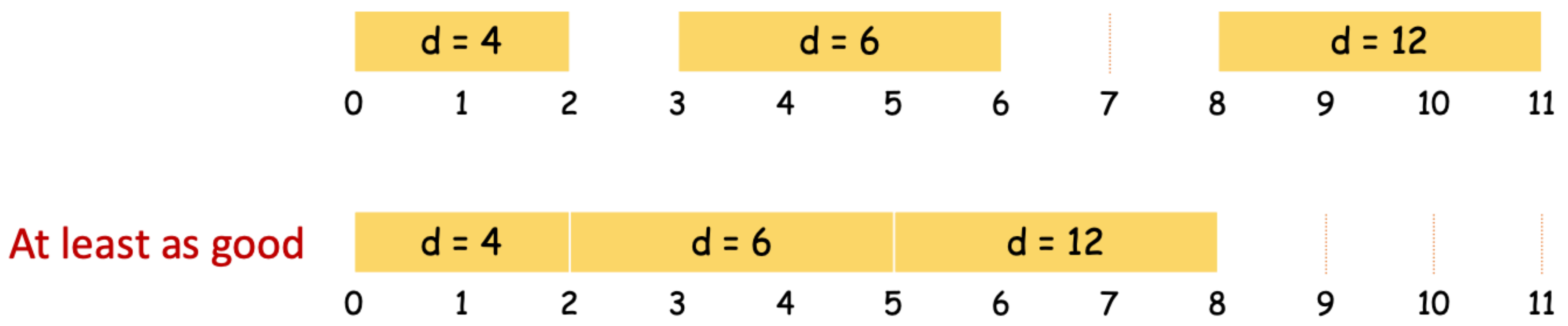
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Exchange argument for optimality

- If for any solution there exists a modification that modifies solution but its value is at least as good as the original, then wlog optimal solution has modification

- Consider a solution with “gaps” between jobs



- Then a “gapless” solution by shifting every job earlier is just as good

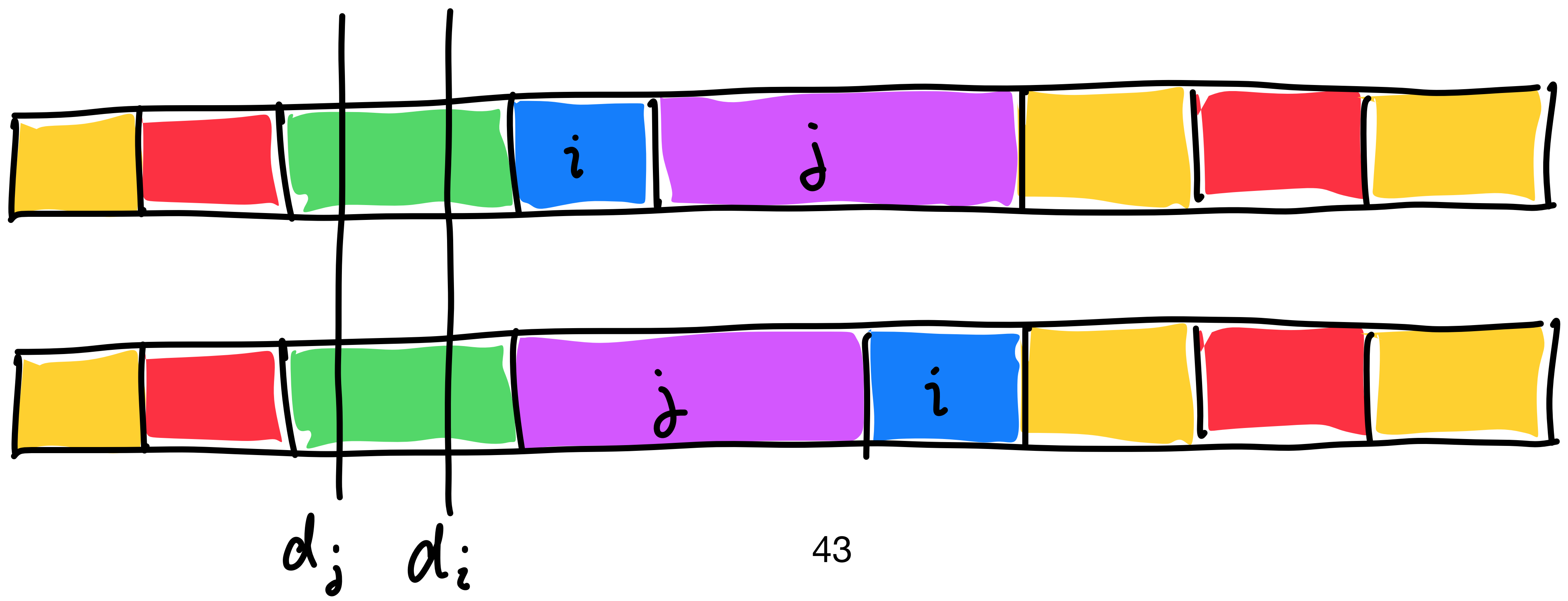
    - Proof: The new t_i' for every job is at most t_i. And ℓ_i is monotonic in t_i. So, the new loss L' is at most L.

The EDF Schedule

- By construction, the EDF schedule is gapless
- This doesn't alone prove optimality
- **Property of EDF:** No inversions in EDF schedule.
  - An inversion is if job i is before job j but $d_i > d_j$.
  - An inversion is adjacent if it occurs between adjacent jobs.
- **Exchange principle:** If a schedule has an adjacent inversion, flipping the adjacent inversion yields a schedule of shorter lateness.

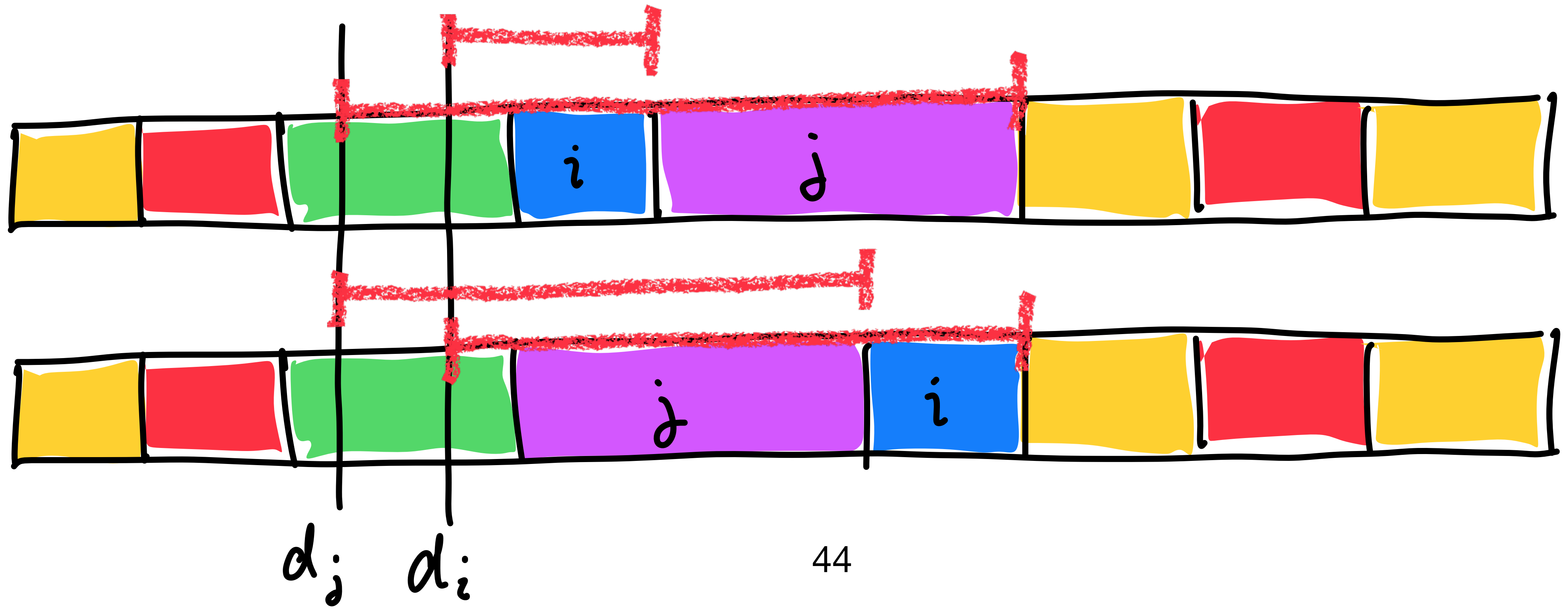
The EDF Schedule

- **Property of EDF:** No inversions in EDF schedule.
  - An inversion is if job i is before job j but $d_i > d_j$.
  - An inversion is adjacent if it occurs between adjacent jobs.
- **Exchange principle:** If a schedule has an adjacent inversion, flipping the adjacent inversion yields a schedule of shorter lateness.
- **Proof:**



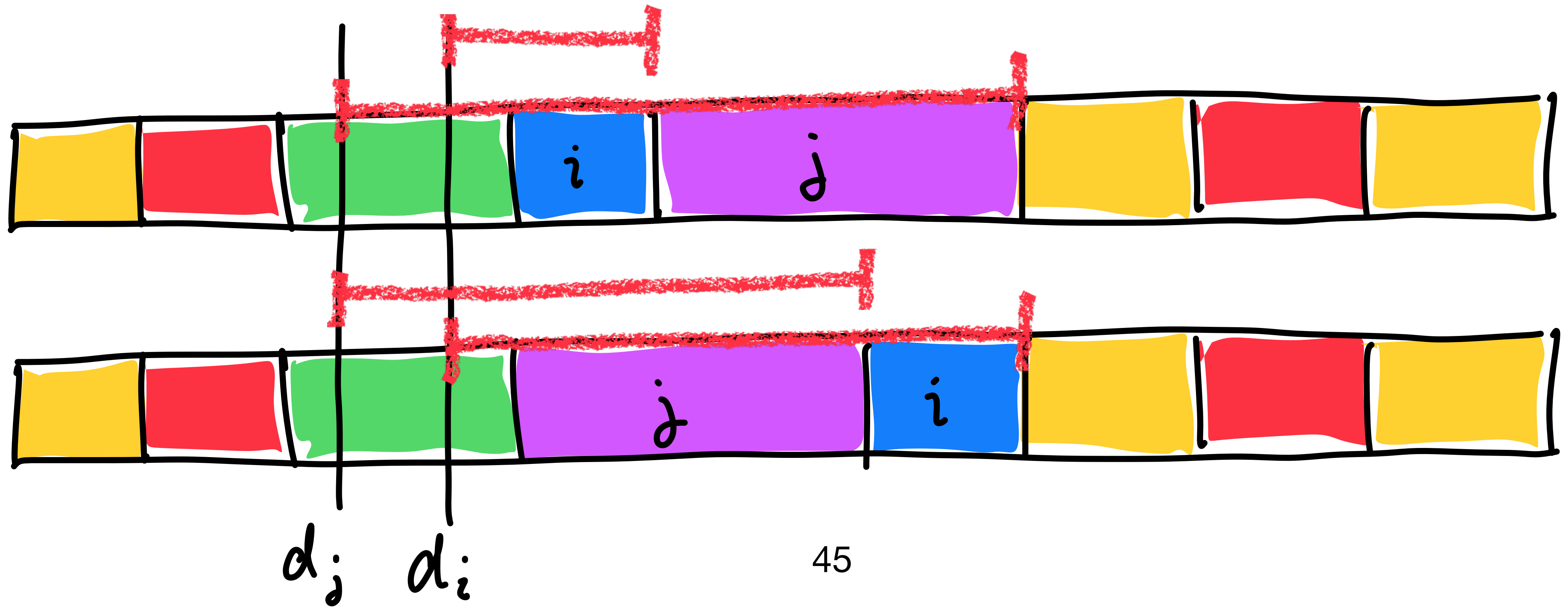
The EDF Schedule

- **Property of EDF:** No inversions in EDF schedule.
  - An inversion is if job i is before job j but $d_i > d_j$.
  - An inversion is adjacent if it occurs between adjacent jobs.
- **Exchange principle:** If a schedule has an adjacent inversion, flipping the adjacent inversion yields a schedule of shorter lateness.
- **Proof:**



The EDF Schedule

- **Property of EDF:** No inversions in EDF schedule.
    - An inversion is if job i is before job j but $d_i > d_j$.
    - An inversion is adjacent if it occurs between adjacent jobs.
- **Exchange principle:** If a schedule has an adjacent inversion, flipping the adjacent inversion yields a schedule of shorter lateness.
- **Proof:**



Notice: max lateness decreases by fixing inversion.

Sum of lateness may increase.

Inversion removal

- If (i, j) is an inversion for $i < j$ but (i, j') is not an inversion for $i < j' < j$, then $(j - 1, j)$ is an *adjacent inversion*
- By induction, if an inversion exists, then an adjacent inversion exists
- Exchange principle lets us clean up all the adjacent inversions
- “Gapless” and “inversion” exchange principles yield a gapless schedule with no inversions
- This is precisely, the earliest deadline first (EDF) schedule up to events of equal deadline. All such schedules have same lateness. Thus, it is optimal