Greedy Algorithms II

Summer 2018 • Lecture 08/07

A Few Notes

Homework 5

- Due 8/10 at 5 p.m. on Gradescope.

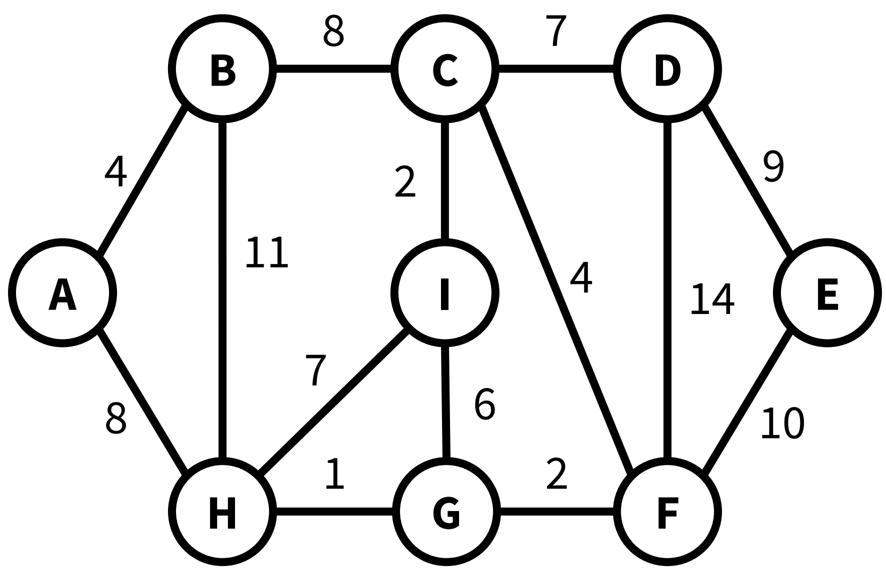
Outline for Today

- Greedy algorithms
  - Greedy graph algorithms
    - Kruskal's Algorithm
  - Activity Selection

Kruskal’s Algorithm

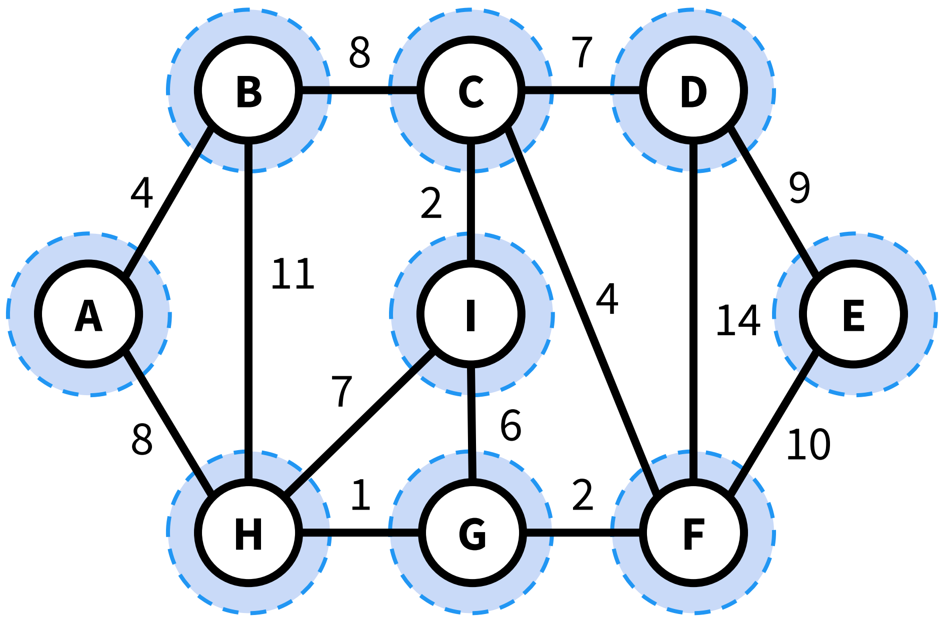
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



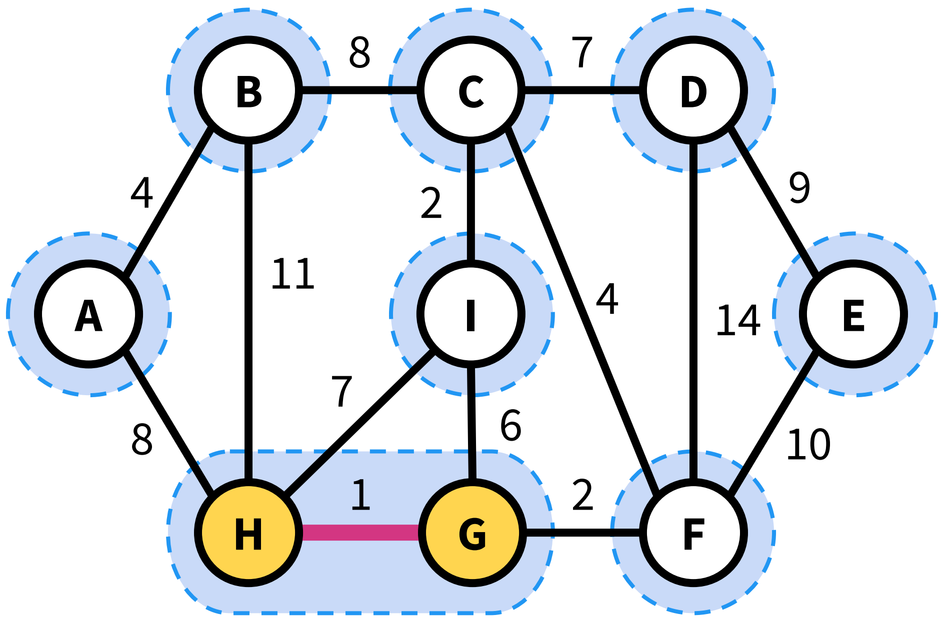
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



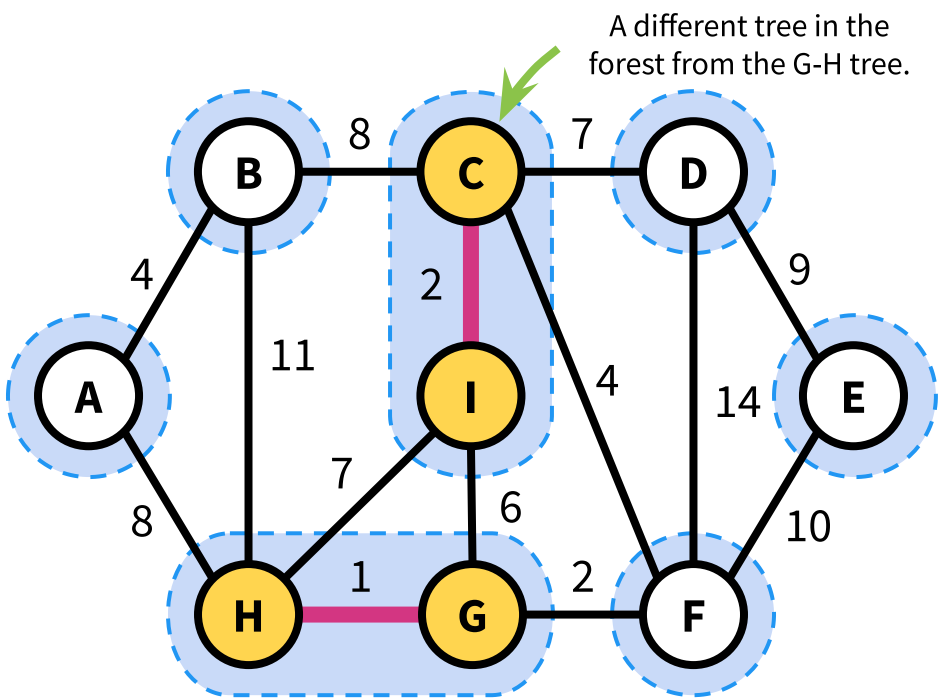
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



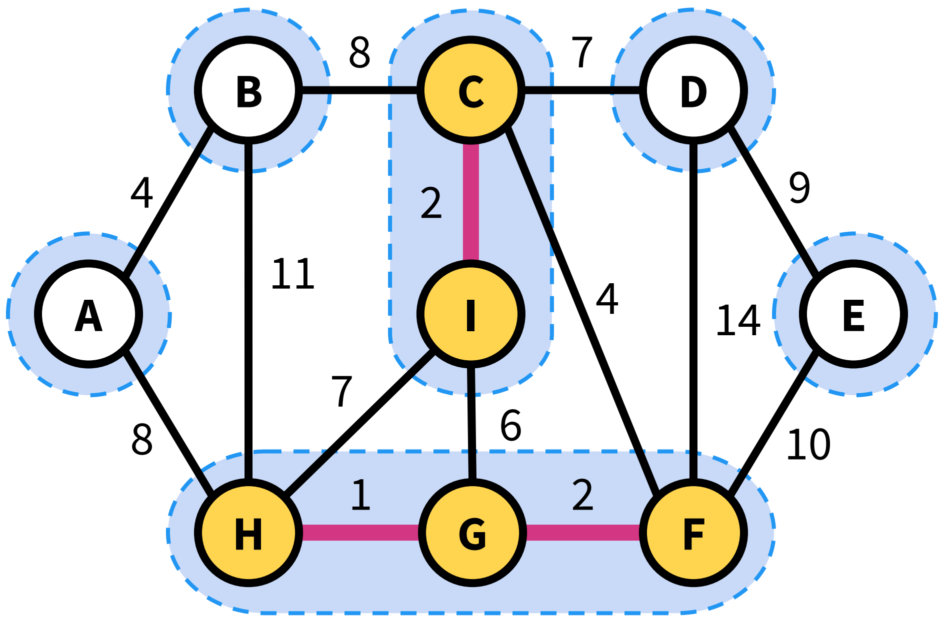
Kruskal’s Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



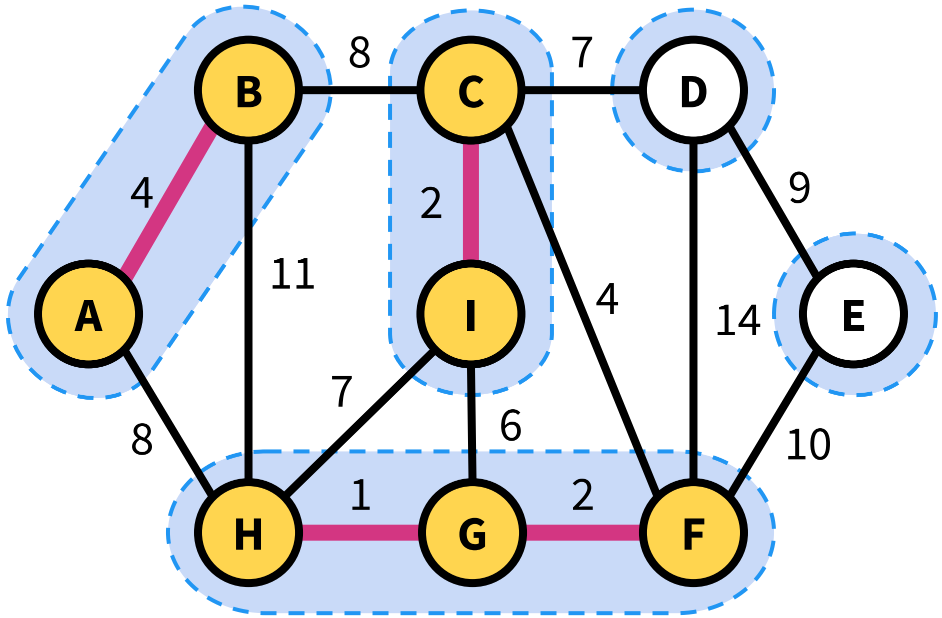
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



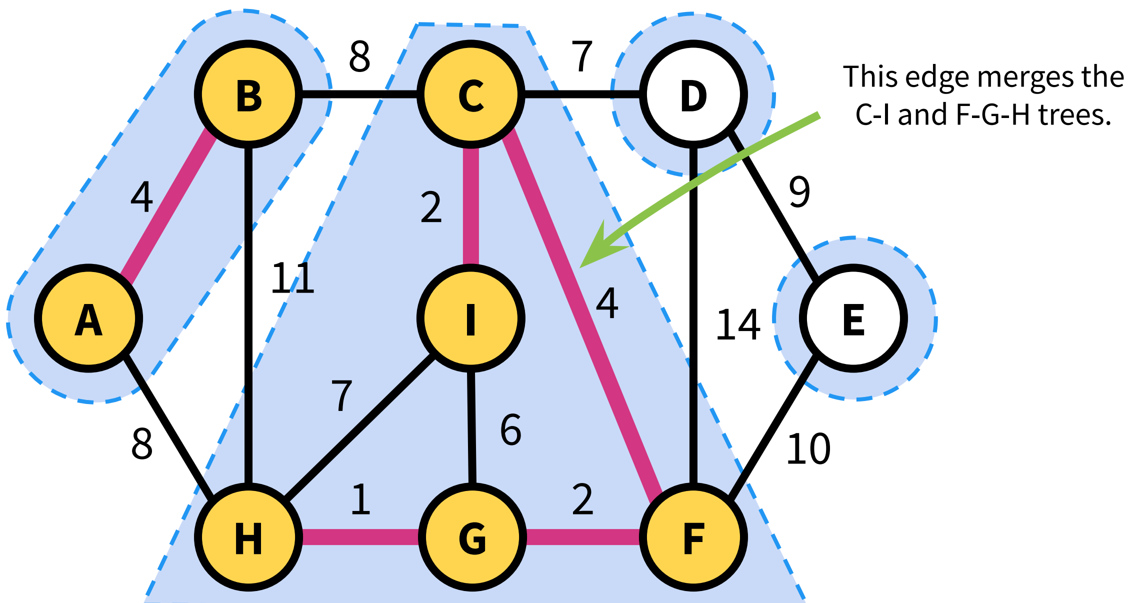
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



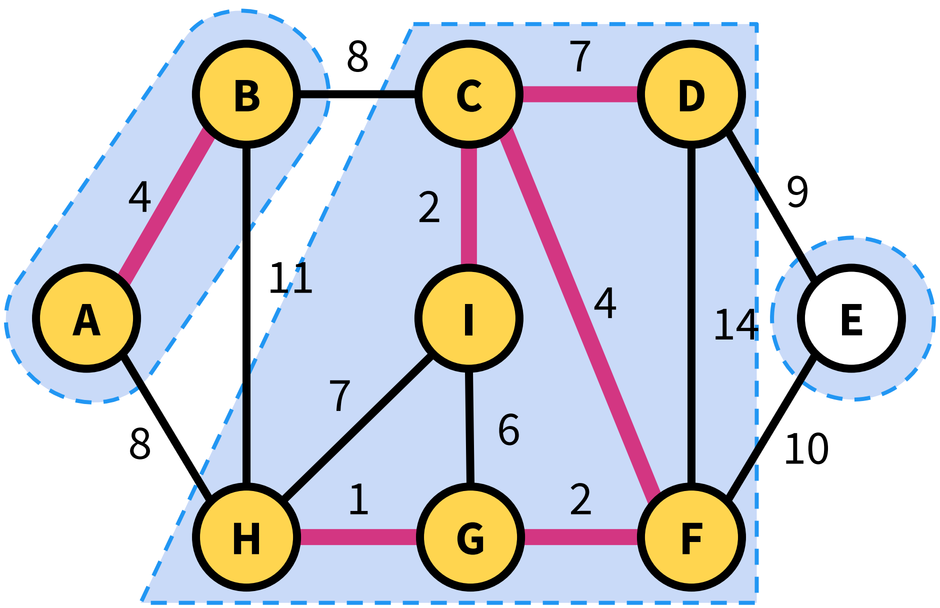
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



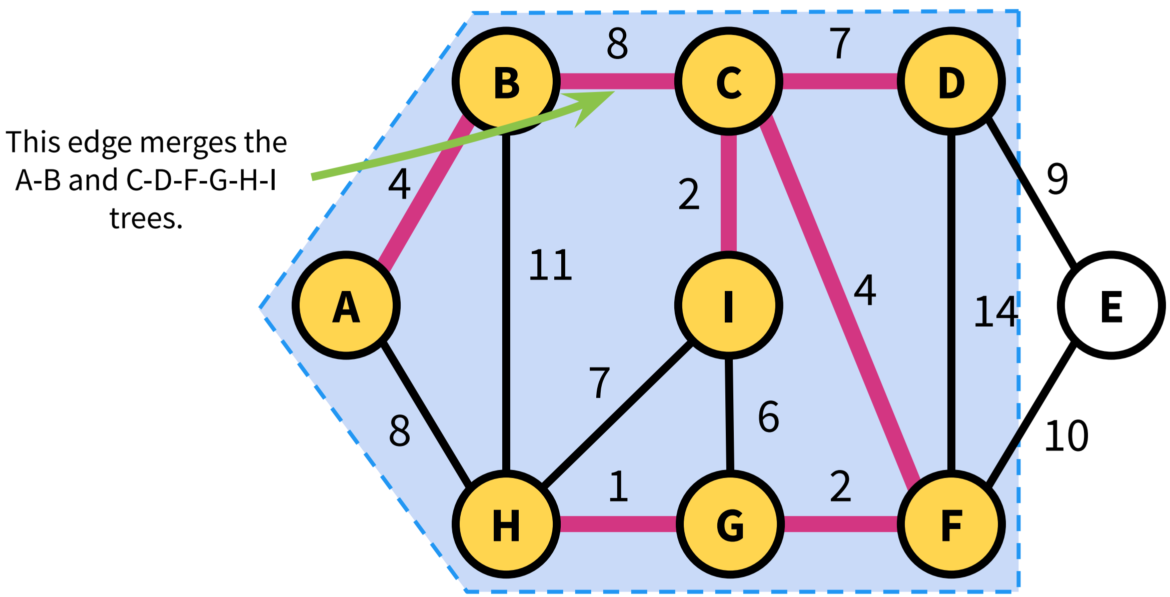
Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



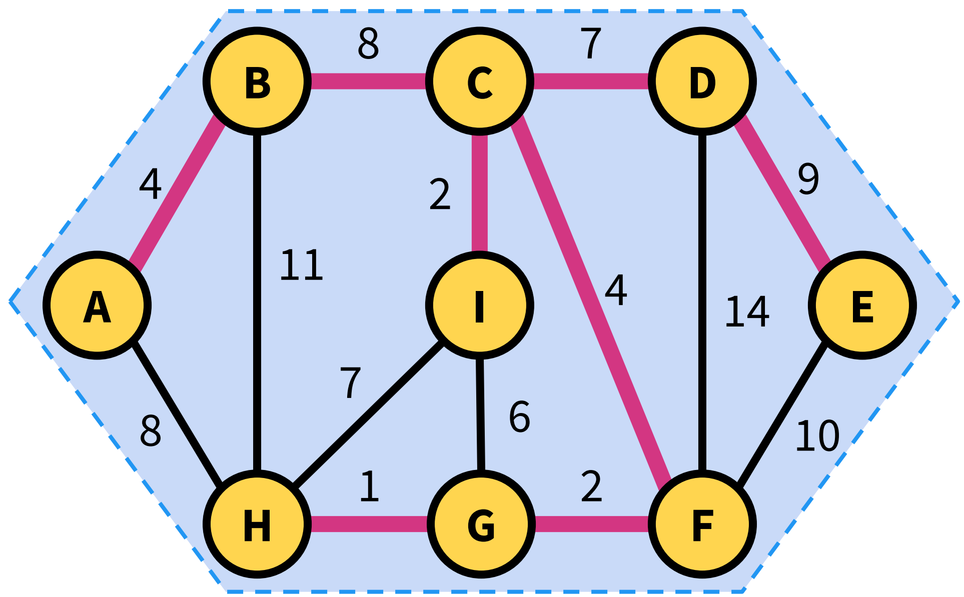
Kruskal’s Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

Main idea: Maintain a forest of trees of visited vertices by greedily adding the cheapest edge.



Kruskal's Algorithm

`kruskal` uses union-find data structure, which supports …

- `make_set(u)`: create a set {u} in `O(1)`
- `find(u)`: returns the set containing u in `O(1)`
- `union(u,v)`: merges the sets containing u and v in `O(1)`

Technically, these operations all run in amortized-time `α(|V|)`; $\alpha(n) \le 4$, provided n < # of atoms in the universe. We will discuss amortized analysis in greater detail later this quarter.

Kruskal’s Algorithm

```
def kruskal(G):
  E_sorted = sort the edges in E by non-decreasing weight
  MST = {}
  for v in V:
    make_set(v)  # put each vertex in its own tree
  for (u, v) in E_sorted:
    if find(u) != find(v):  # u and v in different trees
      MST.add((u, v))
      union(u, v)  # merge u’s tree with v’s tree
  return MST
```

Runtime:

$O(|E|\log(|V|))$

Using comparison-based sort. Note $|E|\log(|E|) = O(|E|\log(|V|^2)) = O(|E| \bullet 2\log(|V|) = O(|E|\log(|V|))$.

$O(|E|)$

Using radix sort

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Kruskal's Algorithm

Recall our lemma:

> Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.
>
> **Lemma:** There exists an MST containing **A** $\cup$ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

`kruskal` finds an edge (u, v) that merges two trees T_1 and T_2. Consider the cut $\{T_1, V - T_1\}$; `MST` respects this cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(u, v)}.

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

Recall, we proved our lemma with an exchange argument!

`kruskal` finds an edge (u, v) that merges two trees T_1 and T_2. Consider the cut $\{T_1, V - T_1\}$; `MST` respects this cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(u, v)}.

Kruskal's Algorithm

Recall our lemma:

Consider a cut that respects a set of edges **A**, such that there's an MST **T** containing **A**, and a light edge (u, v) not in **T**.

Lemma: There exists an MST containing **A** ∪ {(u, v)}.

Theorem: `kruskal` returns a minimum spanning tree.

Proof:

At the start of the first iteration of the while loop, there exists a minimum spanning tree with the edges in `MST`. This trivially holds since we initialize `MST` to the empty set.

`kruskal` finds an edge (u, v) that merges two trees T_1 and T_2. Consider the cut $\{T_1, V - T_1\}$; `MST` respects this cut. By our lemma, there exists a minimum spanning tree containing `MST` ∪ {(u, v)}.

Recall, we proved our lemma with an exchange argument!

After adding the the $(n-1)^{st}$ edge, we have a spanning tree; therefore, `MST` contains a minimum spanning tree. ■

Prim's and Kruskal's

	Description	Runtime	Use-cases
Prim's	Grows a tree	`O(|E|log(|V|))` **with red-black tree** `O(|E|+|V|log(|V|))` **with Fibonacci heap**	Better on dense graphs
Kruskal's	Grows a forest	`O(|E|log(|V|))` **with union-find** `O(|E|)` **with union-find and radix sort**	Better on sparse graphs and if the edge weights can be radix sorted.

Beyond Prim's and Kruskal's

Karger-Klein-Tarjan (1995): Las Vegas randomized algorithm

$O(|E|)$ expected, $O(\min\{|E|\log(|V|), |V|^2\})$ worst-case

Chazelle (2000): $O(|E|\alpha(|V|))$ deterministic algorithm

Inverse Ackermann function

Activity Selection

Planning Your Life

You have a list of activities $(s_1, e_1), (s_2, e_2), \ldots, (s_n, e_n)$ denoted by their start and end times.

All activities are equally attractive to you, and you want to maximize the number of activities you do.

Task: Choose the largest number of non-overlapping activities possible.

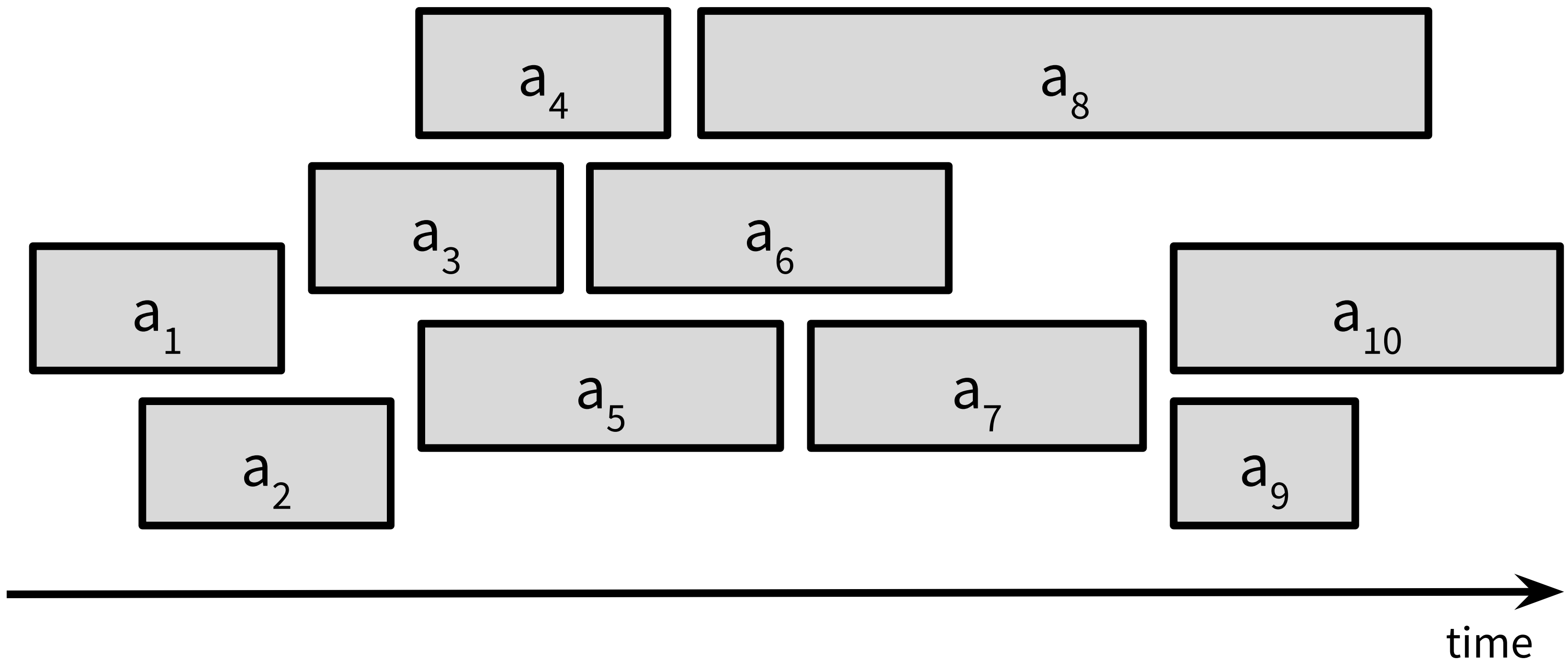
Greedy Choices

What are a few ways of picking activities greedily? 🤔

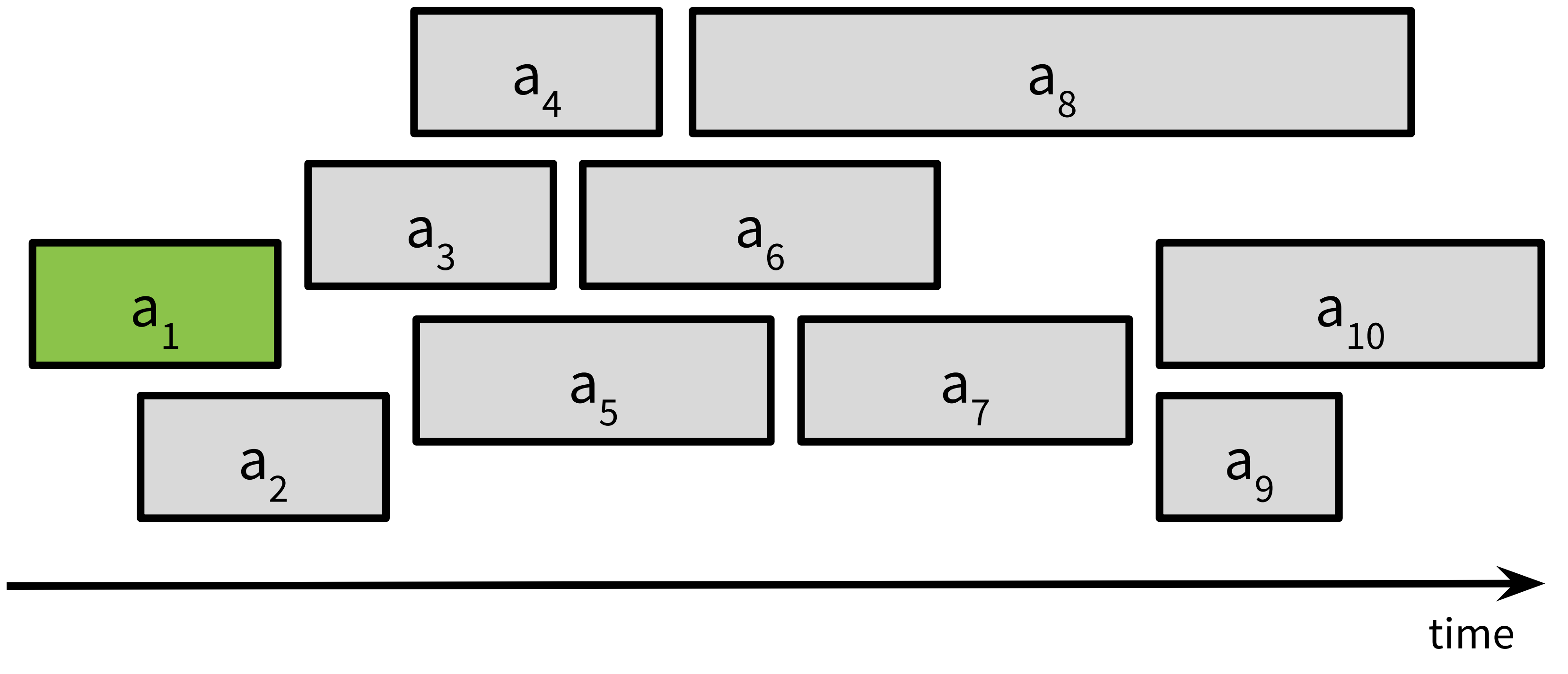
- **Be impulsive:** choose activities in ascending order of start times.
- **Avoid commitment:** choose activities in ascending order of length.
- **Finish fast:** choose activities in ascending order of end times.

Only the third one seems to work.

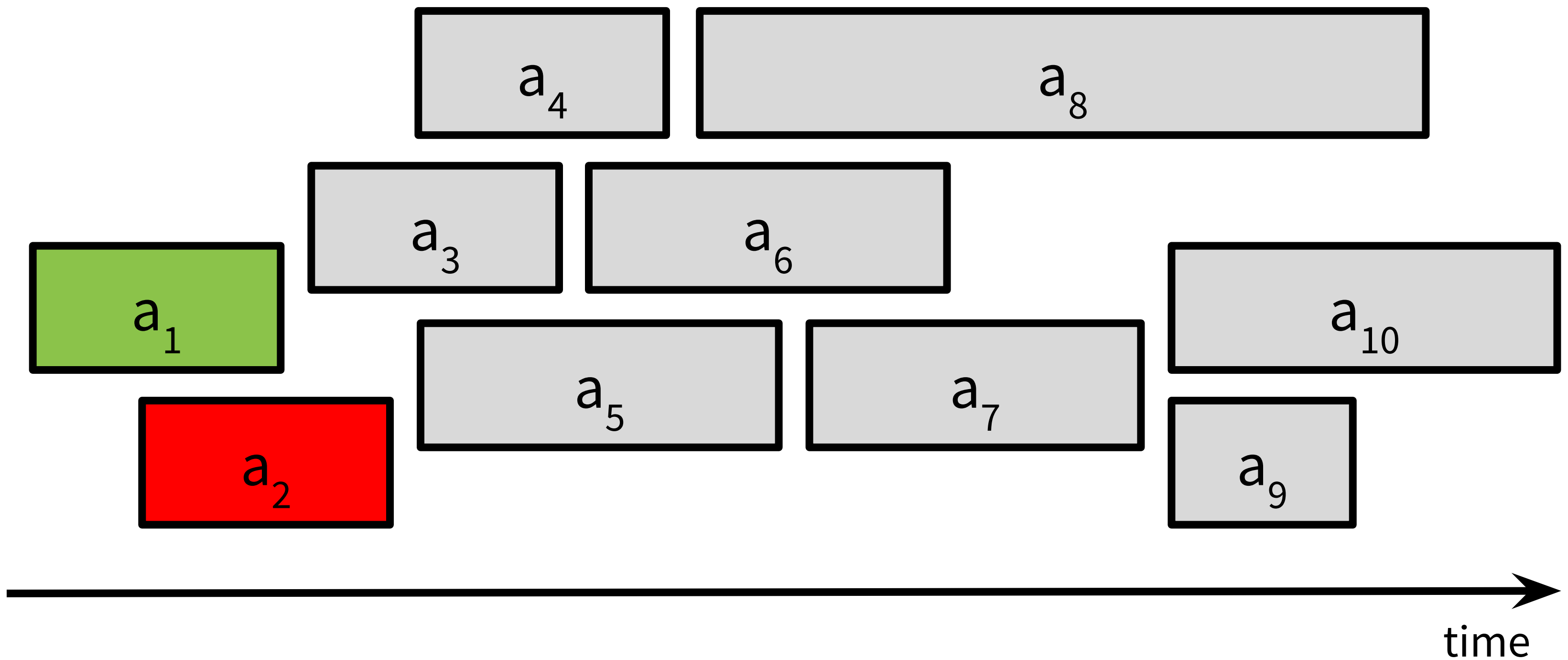
Greedy Choices



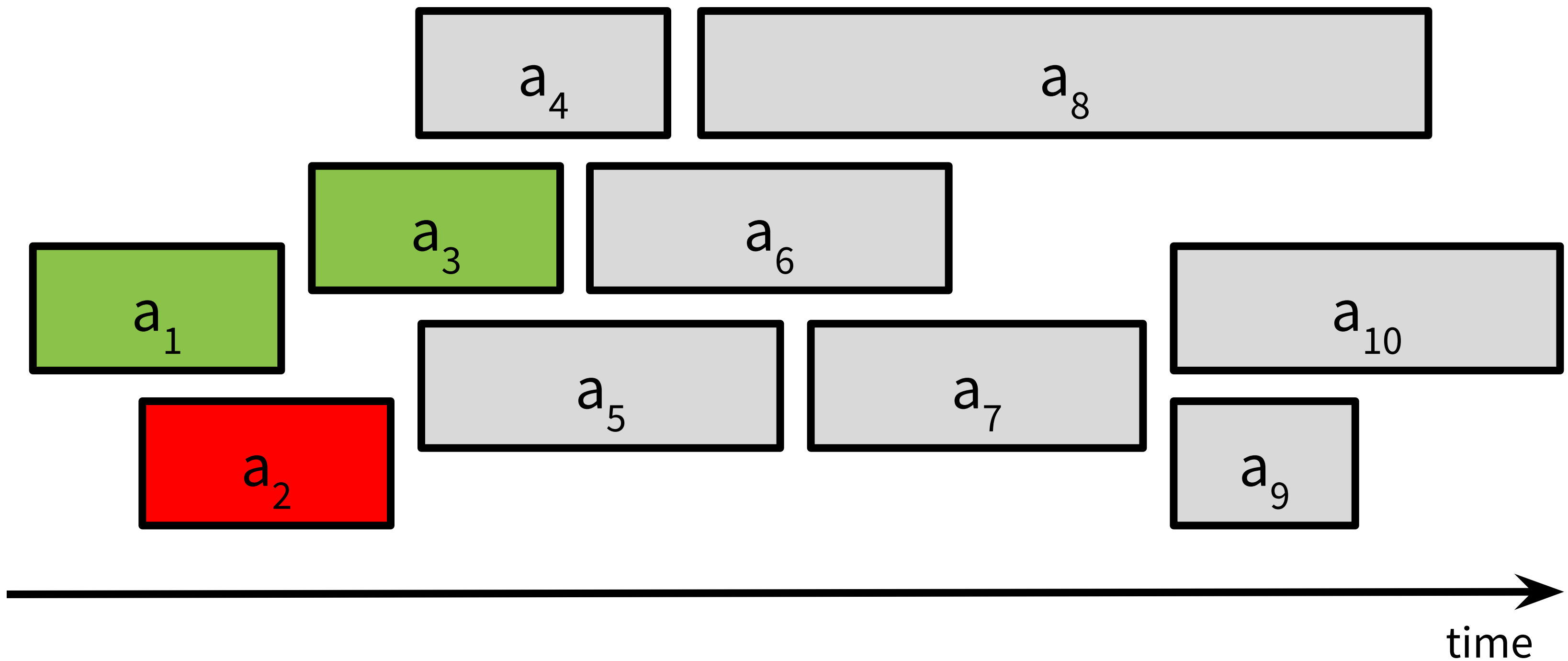
Greedy Choices



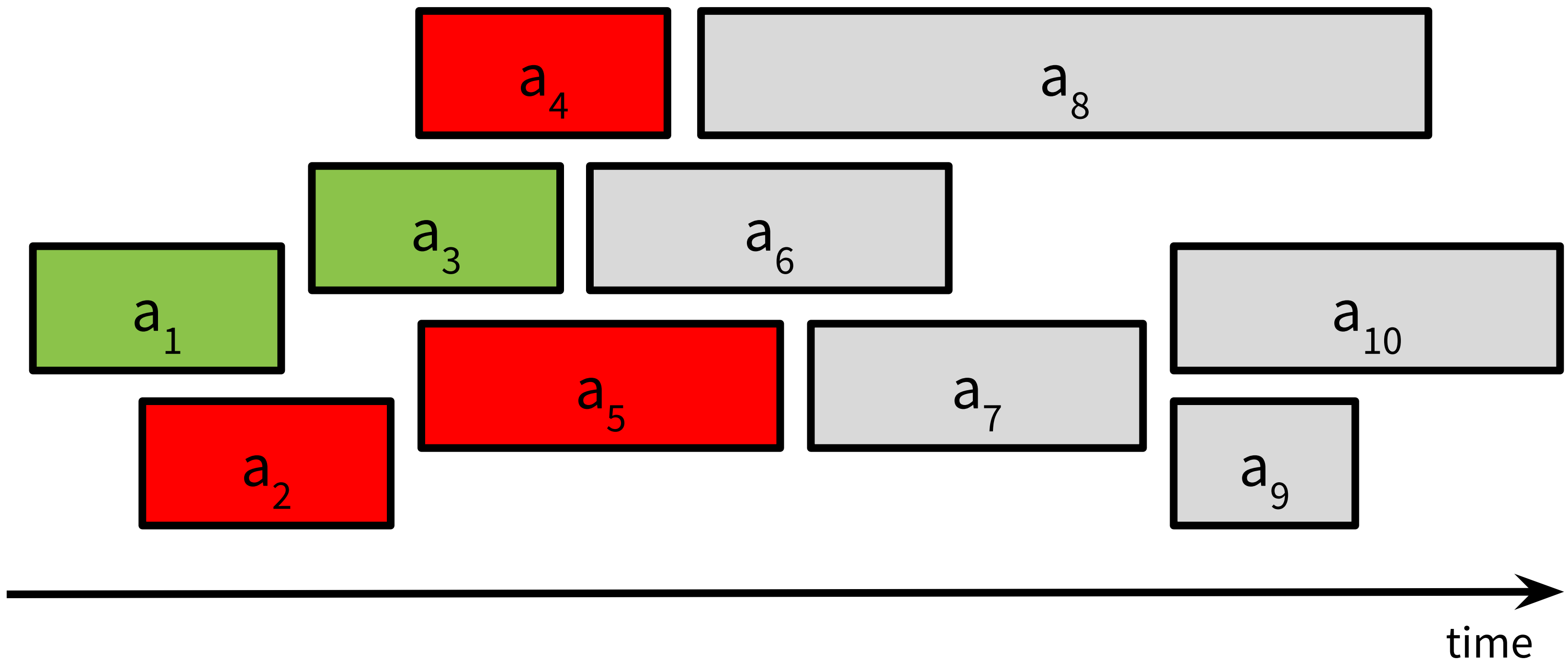
Greedy Choices



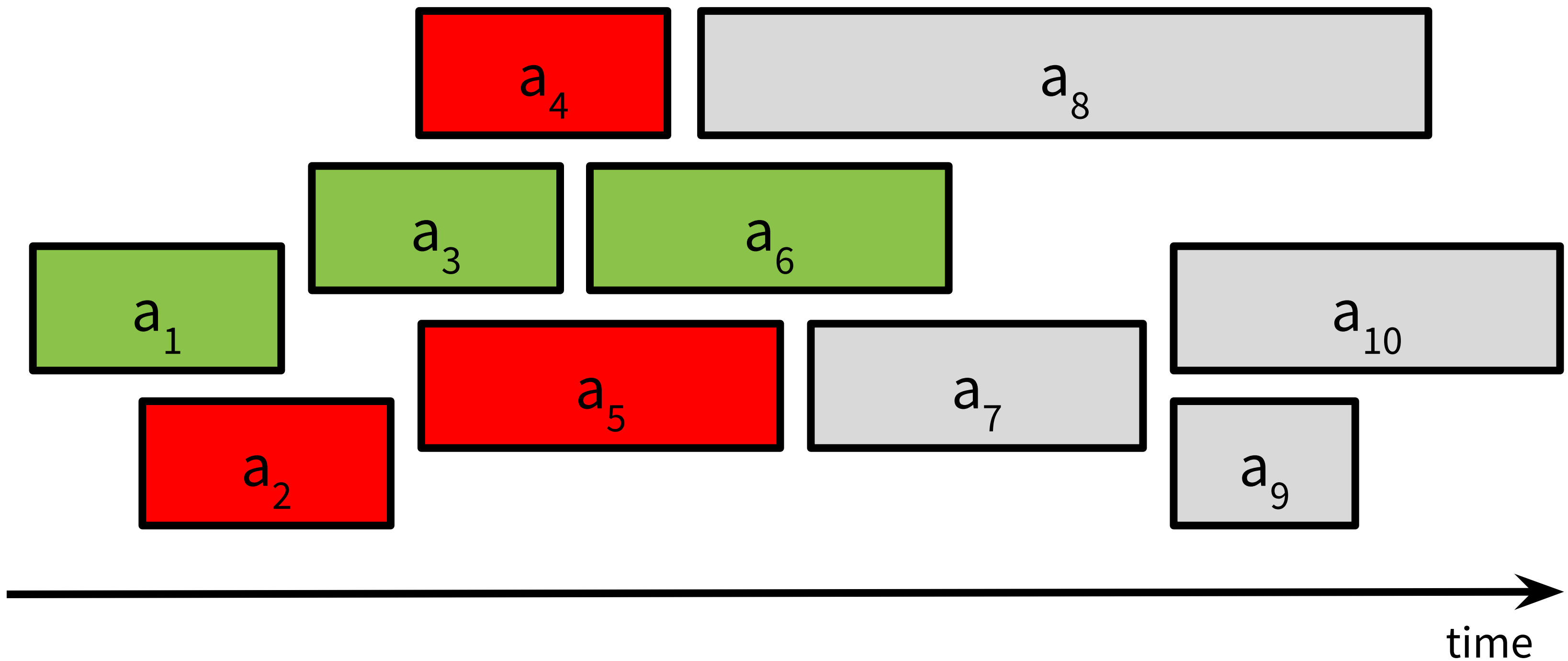
Greedy Choices



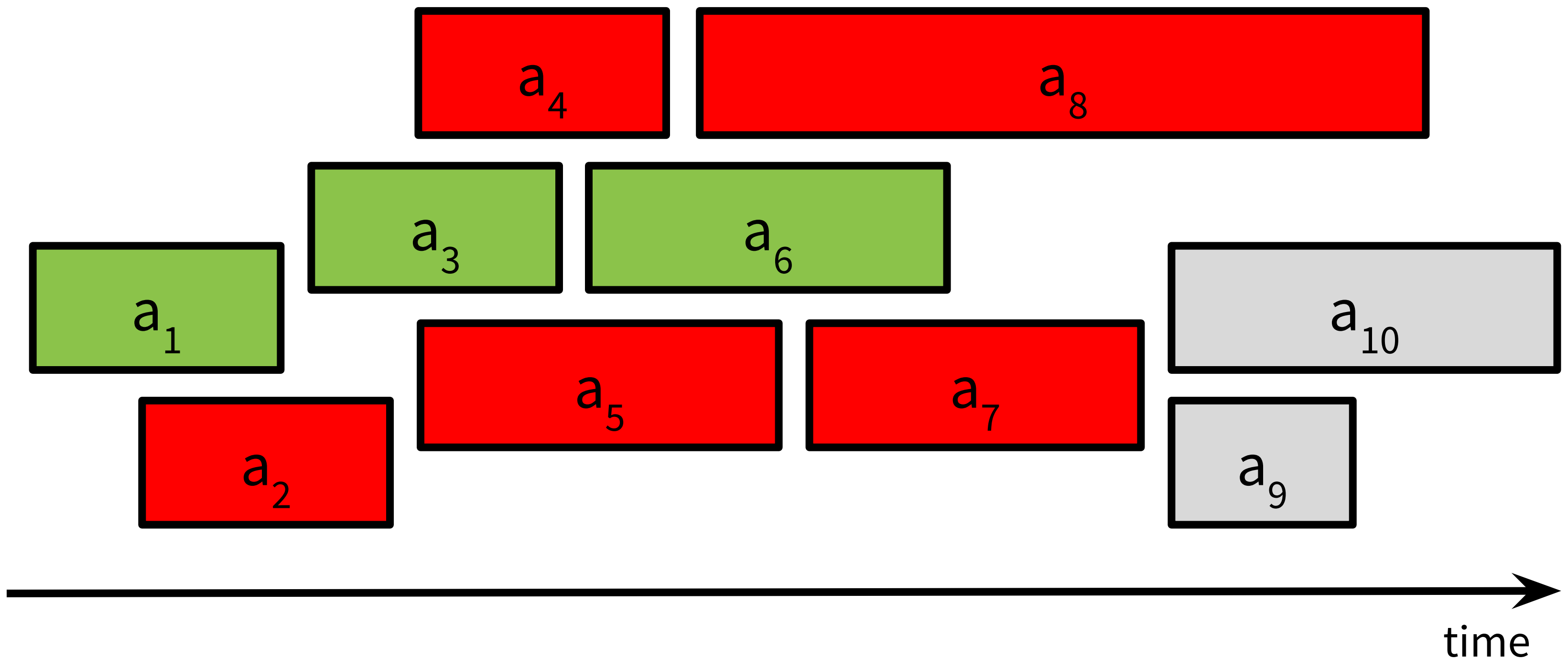
Greedy Choices



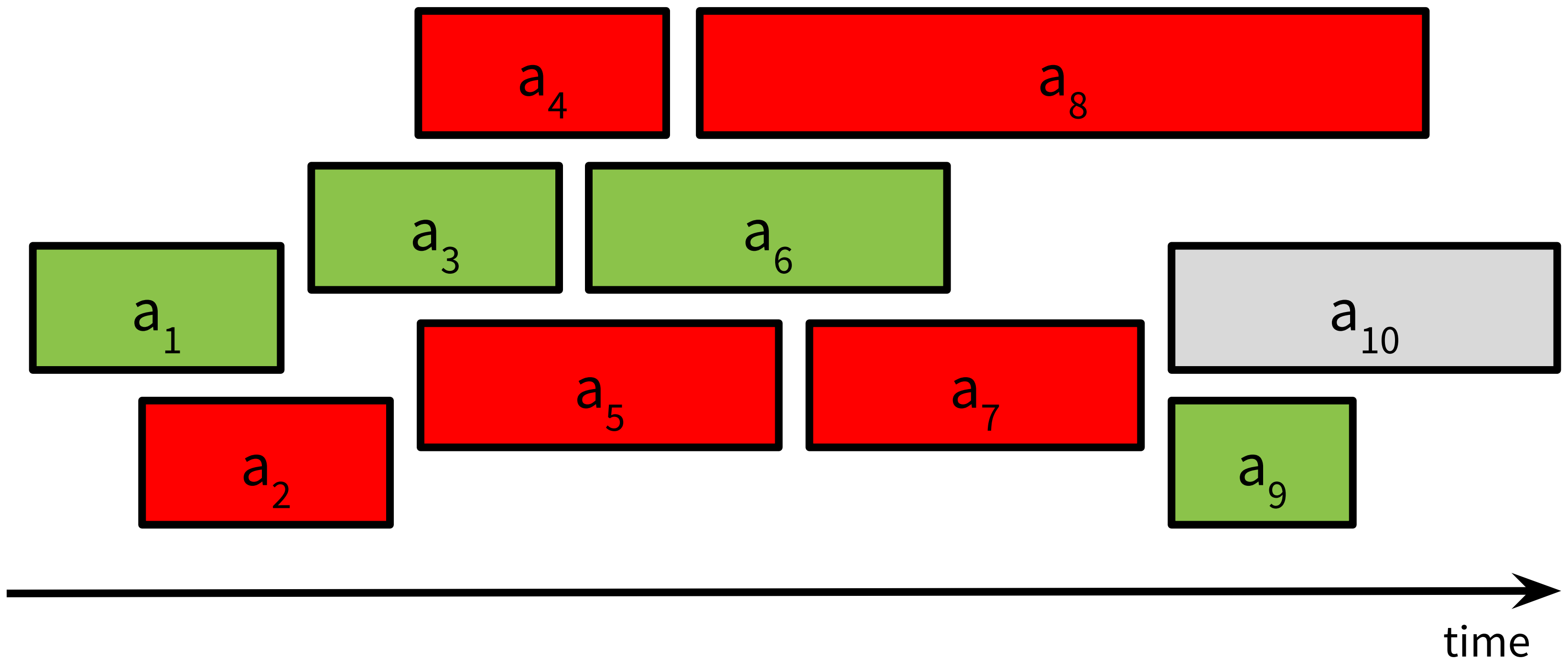
Greedy Choices



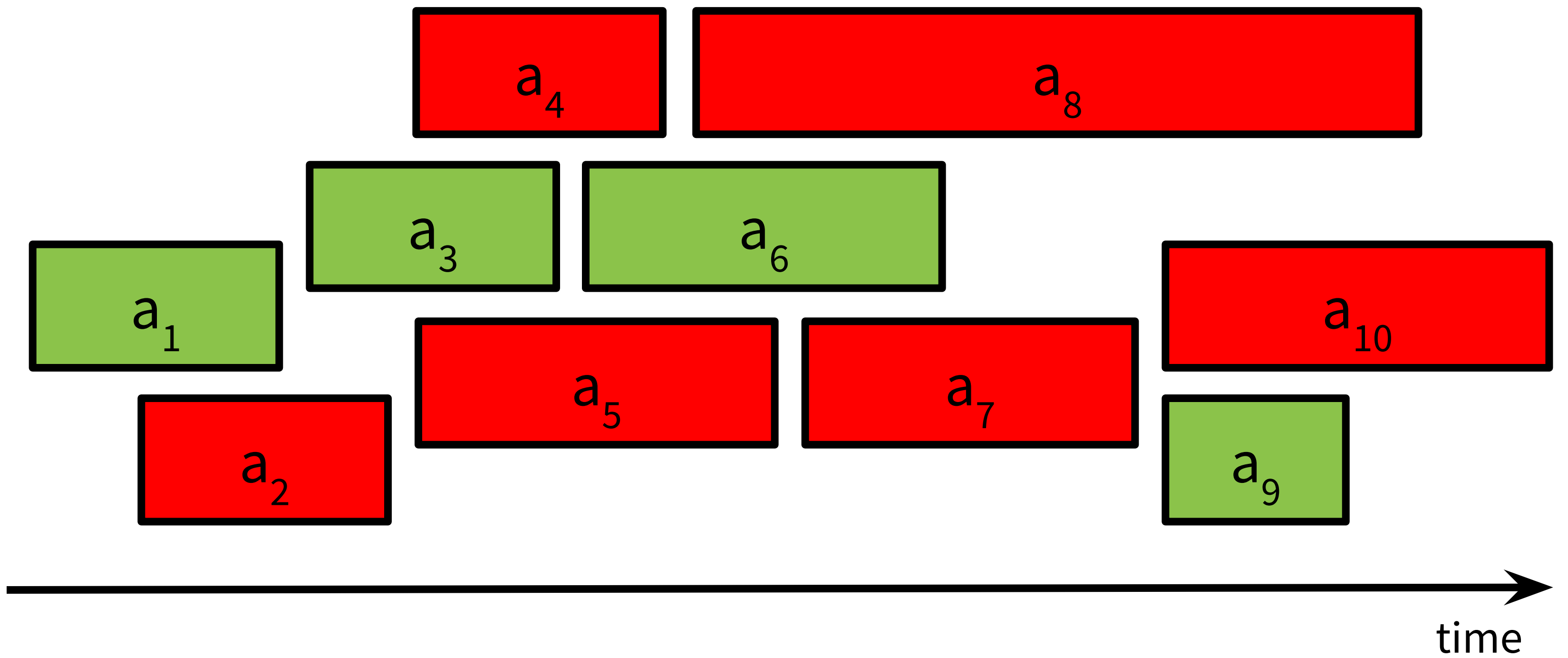
Greedy Choices



Greedy Choices



Greedy Choices



Activity Selection

```
def activity_selection(activities):
  sort activities into ascending order by end time
  S = {}
  U = set of activities
  while U not empty:
    choose any activity with the earliest finishing time
    add that activity to S
    remove other activities that overlap with it from U
  return S
```

Runtime: O(n log(n))

Activity Selection

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible schedule of activities (i.e. it doesn't "schedule conflicting activities").

(2) **Optimality.** The algorithm finds an optimal schedule of activities (i.e. there isn't a better schedule available).

Activity Selection

Lemma: The schedule produced by `activity_selection` is a feasible schedule.

Intuition: Use induction to show that at each step, the set U only contains activities that don't conflict with activities selected from S.

Activity Selection

We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible schedule of activities (i.e. it doesn't "schedule conflicting activities"). 👌

(2) **Optimality.** The algorithm finds an optimal schedule of activities (i.e. there isn't a better schedule available). 🤔

Greedy Stays Ahead

To prove that the schedule S produced by the algorithm is optimal, we will use another “greedy stays ahead” argument.

(1) Find intermediate values that evaluate the solution produced by any algorithm, including the greedy one. **Here, the end_time of the kth activity chosen.**

(2) Show the greedy algorithm produces values at least as good as any solution’s (using induction).

(3) Prove that since the greedy algorithm produces values at least as good as any solution’s, it must be optimal (using direct proof or proof by contradiction).

Greedy Stays Ahead

How might we prove that `activity_selection` finds an optimal schedule of activities?

Intuition: Consider an arbitrary optimal schedule S*, then show that our greedy algorithm produces a schedule S no worse than S*.

Greedy Stays Ahead

Let f(i, S) denote the time that the ith activity finishes in schedule S.

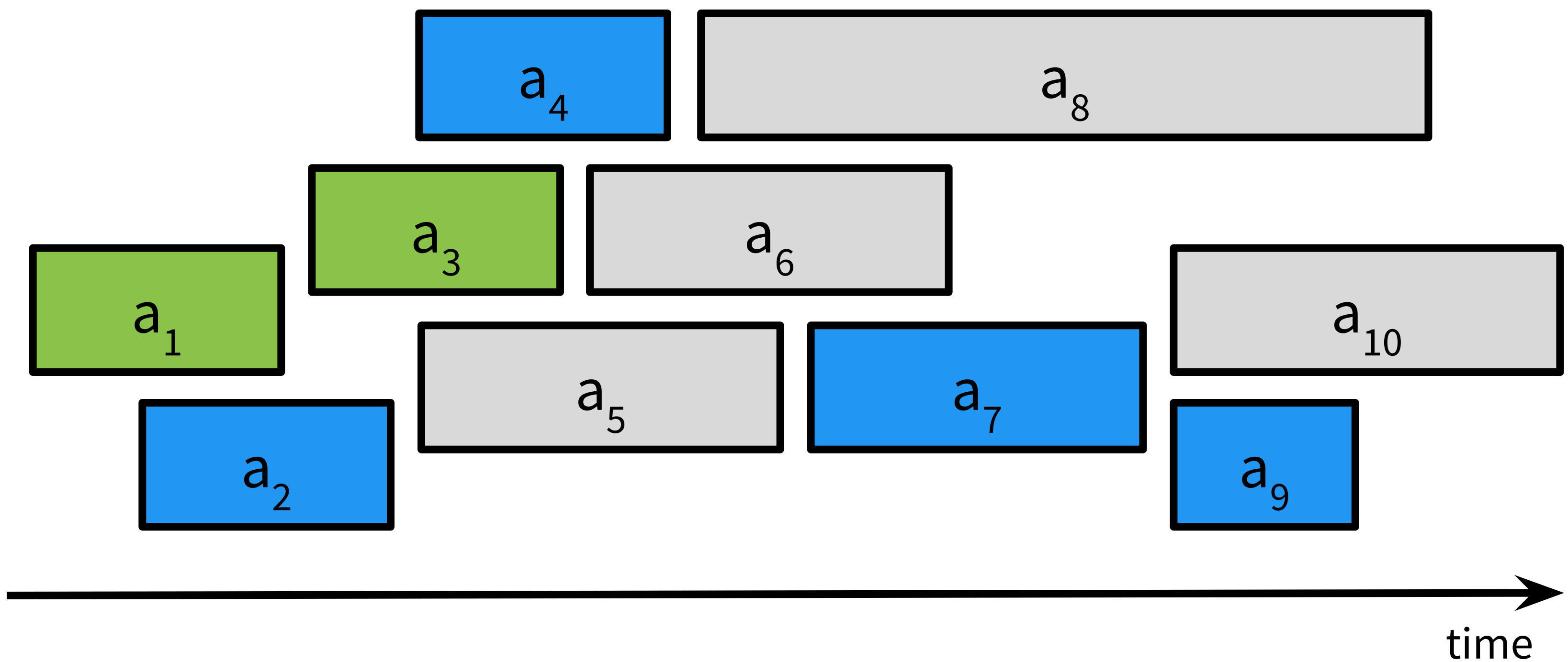
Lemma: For any $1 \le i \le |S|$, we have $f(i, S) \le f(i, S^*)$.

i.e. After scheduling i activities according to the greedy algorithm, you will be at most as late as if you scheduled i activities according to an optimal solution.

Let's formalize this using induction!

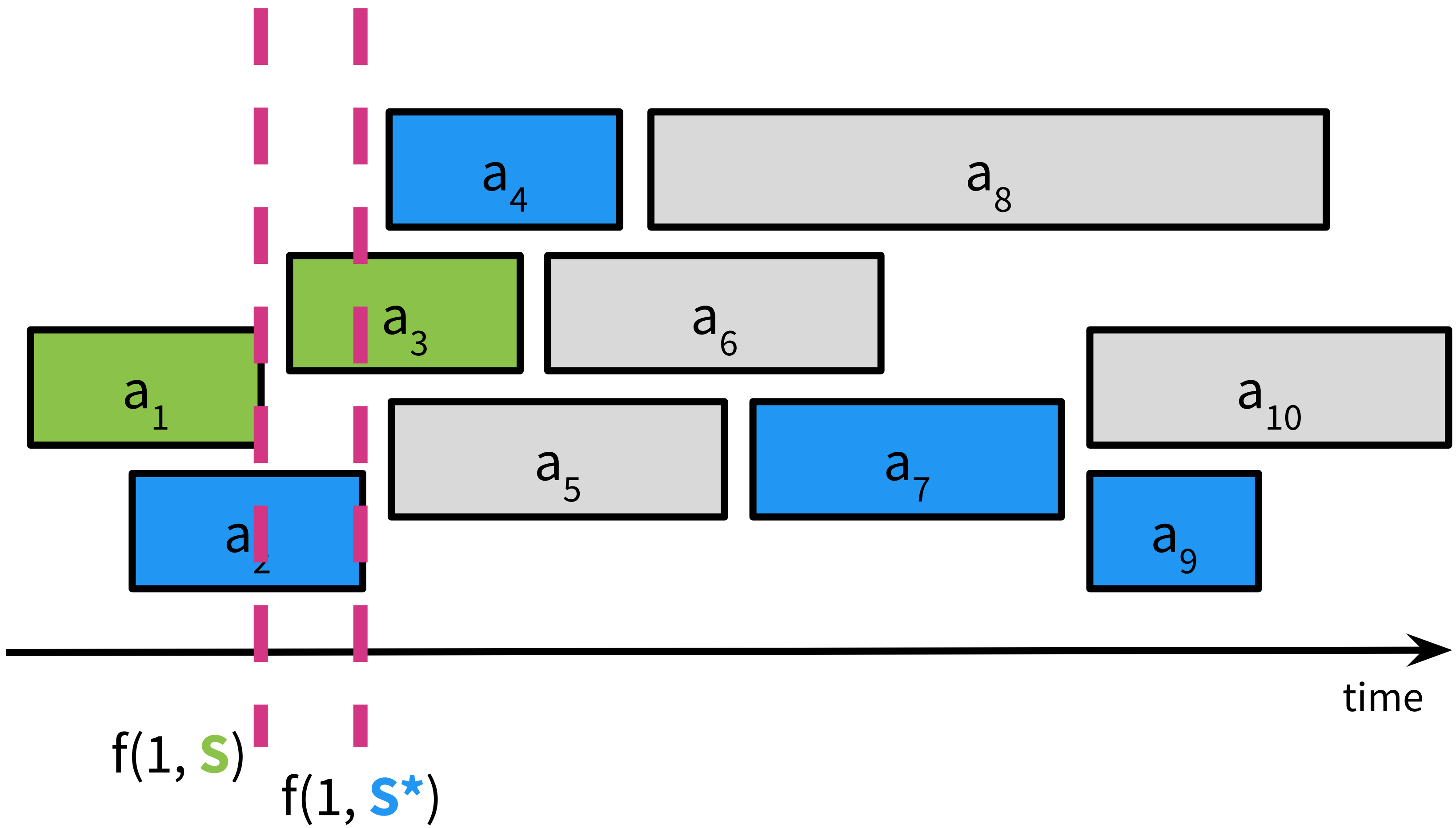
Greedy Stays Ahead

For any $1 \leq i \leq |S|$, we have $f(i, S) \leq f(i, S^*)$.



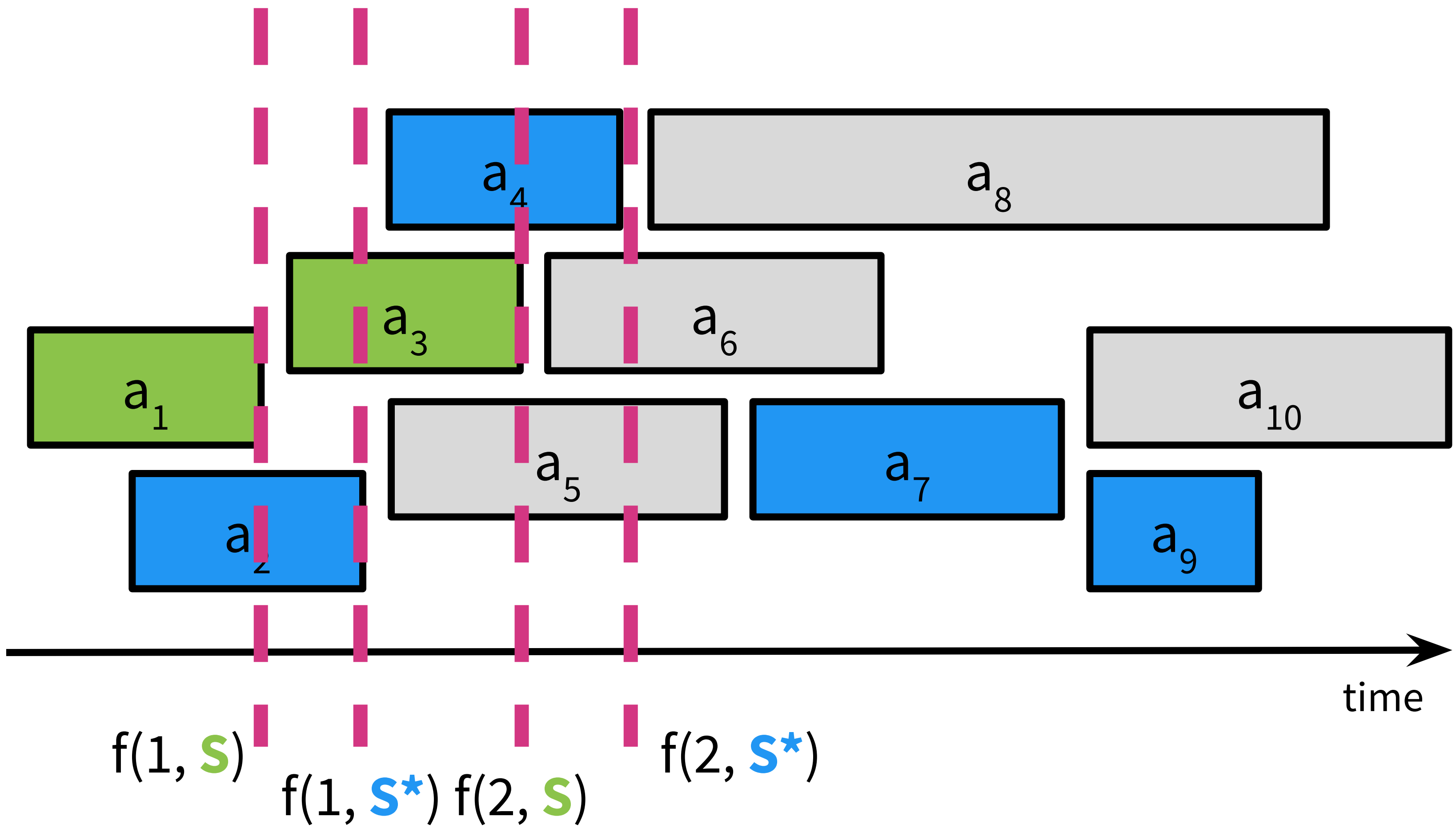
Greedy Stays Ahead

For any $1 \leq i \leq |S|$, we have $f(i, S) \leq f(i, S^*)$.



Greedy Stays Ahead

For any $1 \leq i \leq |S|$, we have $f(i, S) \leq f(i, S^*)$.



Greedy Stays Ahead

Lemma: For all $1 \le i \le |S|$, we have $f(i, S) \le f(i, S^*)$.

Proof: We proceed by induction.

As a base case, the first activity the greedy algorithm selects must be an activity that ends no later than any other activity, so $f(1, S) \le f(1, S^*)$.

Greedy Stays Ahead

Lemma: For all $1 \leq i \leq |S|$, we have $f(i, S) \leq f(i, S^*)$.

Proof: We proceed by induction.

As a base case, the first activity the greedy algorithm selects must be an activity that ends no later than any other activity, so $f(1, S) \leq f(1, S^*)$.

For the inductive step, assume that the claim holds for some $1 \leq i < |S|$. We will prove the claims holds for $i + 1$. Since $f(i, S) \leq f(i, S^*)$, the ith activity in S finishes before the ith activity in S^*.

Greedy Stays Ahead

Lemma: For all $1 \le i \le |S|$, we have $f(i, S) \le f(i, S^*)$.

Proof: We proceed by induction.

As a base case, the first activity the greedy algorithm selects must be an activity that ends no later than any other activity, so $f(1, S) \le f(1, S^*)$.

For the inductive step, assume that the claim holds for some $1 \le i < |S|$. We will prove the claims holds for $i + 1$. Since $f(i, S) \le f(i, S^*)$, the ith activity in S finishes before the ith activity in S*. Since the (i+1)st activity in S* must start after the ith activity in S* ends, the (i+1)st activity in S* must start after the ith activity in S ends.

Greedy Stays Ahead

Lemma: For all $1 \leq i \leq |S|$, we have $f(i, S) \leq f(i, S^*)$.

Proof: We proceed by induction.

As a base case, the first activity the greedy algorithm selects must be an activity that ends no later than any other activity, so $f(1, S) \leq f(1, S^*)$.

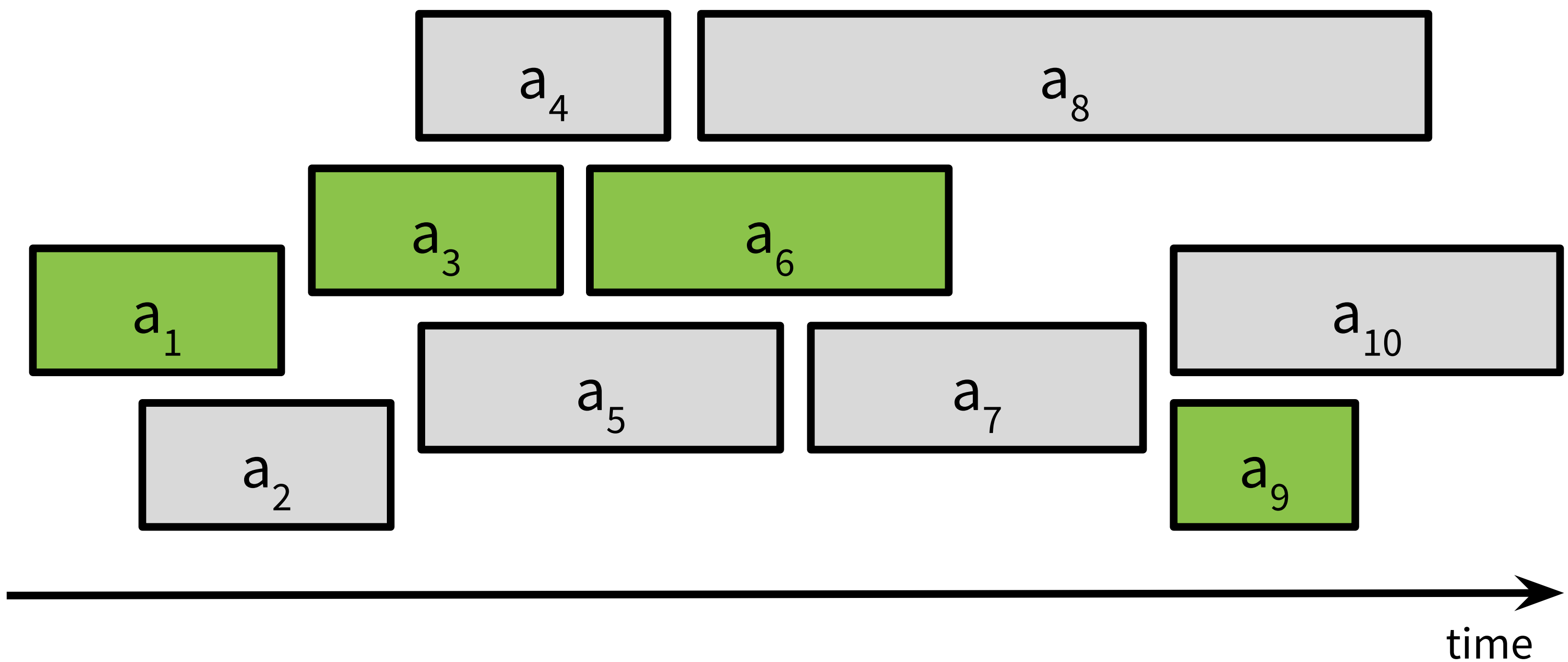
For the inductive step, assume that the claim holds for some $1 \leq i < |S|$. We will prove the claims holds for $i + 1$. Since $f(i, S) \leq f(i, S^*)$, the ith activity in S finishes before the ith activity in S*. Since the (i+1)st activity in S* must start after the ith activity in S* ends, the (i+1)st activity in S* must start after the ith activity in S ends.

Therefore, the (i+1)st activity in S* must be in U when the greedy algorithm selects the activity in U with the lowest end time, we have $f(i+1, S) \leq f(i+1, S^*)$, completing the induction.

Greedy Stays Ahead

Bringing it home: By contradiction, suppose there was an S* with more activities than our solution **S**.

Since for all $1 \leq i \leq |S|$, we have $f(i, S) \leq f(i, S^*)$ it must be the case than the $|S|+1^{st}$ activity has a start time after the end time of the last activity in S. **Impossible!**



Activity Selection

```
def activity_selection(activities):
  sort activities into ascending order by end time
  S = {}
  U = set of activities
  while U not empty:
    choose any activity with the earliest finishing time
    add that activity to S
    remove other activities that overlap with it from U
  return S
```

Runtime: O(n log(n))

Greedy Stays Ahead

Theorem: `activity_selection` produces an optimal solution.

Proof: Since S^* is optimal, we have $|S| \leq |S^*|$. We will prove $|S| = |S^*|$.

We proceed by contradiction. Suppose that $|S| < |S^*|$. Let $k = |S|$. By our lemma, we know $f(k, S) \leq f(k, S^*)$, so the kth activity in S finishes no later than the kth activity in S^*.

Greedy Stays Ahead

Theorem: `activity_selection` produces an optimal solution.

Proof: Since S^* is optimal, we have $|S| \leq |S^*|$. We will prove $|S| = |S^*|$.

We proceed by contradiction. Suppose that $|S| < |S^*|$. Let $k = |S|$. By our lemma, we know $f(k, S) \leq f(k, S^*)$, so the kth activity in S finishes no later than the kth activity in S^*. Since $|S| < |S^*|$, there is a (k+1)st activity in S^*, and its start time must be after $f(k, S^*)$ and therefore after $f(k, S)$.

Greedy Stays Ahead

Theorem: `activity_selection` produces an optimal solution.

Proof: Since S^* is optimal, we have $|S| \leq |S^*|$. We will prove $|S| = |S^*|$.

We proceed by contradiction. Suppose that $|S| < |S^*|$. Let $k = |S|$. By our lemma, we know $f(k, S) \leq f(k, S^*)$, so the kth activity in S finishes no later than the kth activity in S^*. Since $|S| < |S^*|$, there is a $(k+1)$st activity in S^*, and its start time must be after $f(k, S^*)$ and therefore after $f(k, S)$. Thus after the greedy algorithm added its kth activity to S, the $(k+1)$st activity from S^* would still belong to U. But the greedy algorithm ended after k activities, so U must have been empty.

Greedy Stays Ahead

Theorem: `activity_selection` produces an optimal solution.

Proof: Since S^* is optimal, we have $|S| \leq |S^*|$. We will prove $|S| = |S^*|$.

We proceed by contradiction. Suppose that $|S| < |S^*|$. Let $k = |S|$. By our lemma, we know $f(k, S) \leq f(k, S^*)$, so the kth activity in S finishes no later than the kth activity in S^*. Since $|S| < |S^*|$, there is a (k+1)st activity in S^*, and its start time must be after $f(k, S^*)$ and therefore after $f(k, S)$. Thus after the greedy algorithm added its kth activity to S, the (k+1)st activity from S^* would still belong to U. But the greedy algorithm ended after k activities, so U must have been empty.

We have reached a contradiction, so our assumption was wrong and $|S^*| = |S|$, so the greedy algorithm produces an optimal solution. ■

Greedy Stays Ahead

Theorem: `activity_selection` produces an optimal solution.

Proof: Since S^* is optimal, we have $|S| \le |S^*|$. We will prove $|S| = |S^*|$.

We proceed by contradiction. Suppose that $|S| < |S^*|$. Let $k = |S|$. By our lemma, we know $f(k, S) \le f(k, S^*)$, so the kth activity in S finishes no later than the kth activity in S^*. Since $|S| < |S^*|$, there is a (k+1)st activity in S^*, and its start time must be after $f(k, S^*)$ and therefore after $f(k, S)$. Thus after the greedy algorithm added its kth activity to S, the (k+1)st activity from S^* would still belong to U. But the greedy algorithm ended after k activities, so U must have been empty.

We have reached a contradiction, so our assumption was wrong and $|S^*| = |S|$, so the greedy algorithm produces an optimal solution. ■

In Frog Hopping, we proved this step using a direct proof. Here, we use a proof by contradiction. You should be able to structure the direct proof here too.

Activity Selection

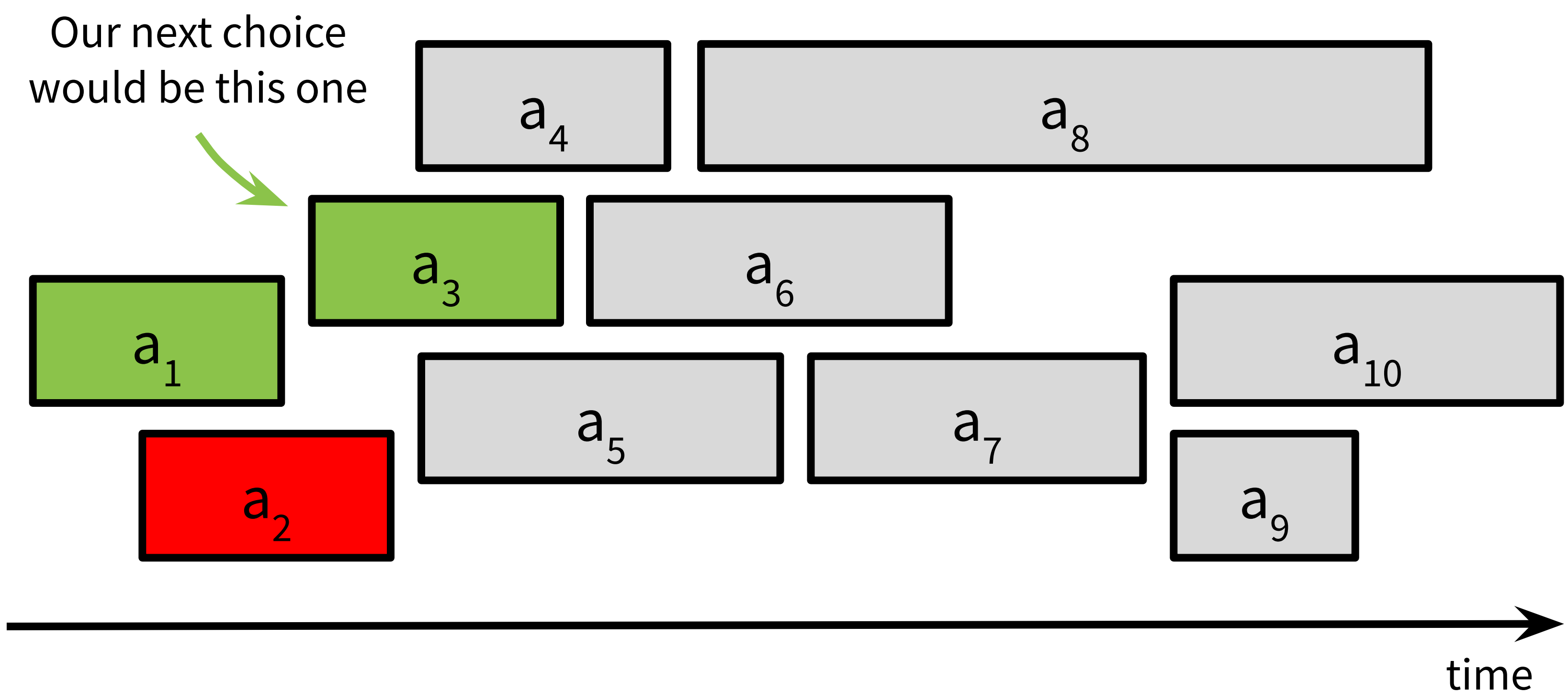
We need to prove two properties about the algorithm to guarantee correctness.

(1) **Feasibility.** The algorithm finds a feasible schedule of activities (i.e. it doesn't "schedule conflicting activities"). 👌

(2) **Optimality.** The algorithm finds an optimal schedule of activities (i.e. there isn't a better schedule available). 👌

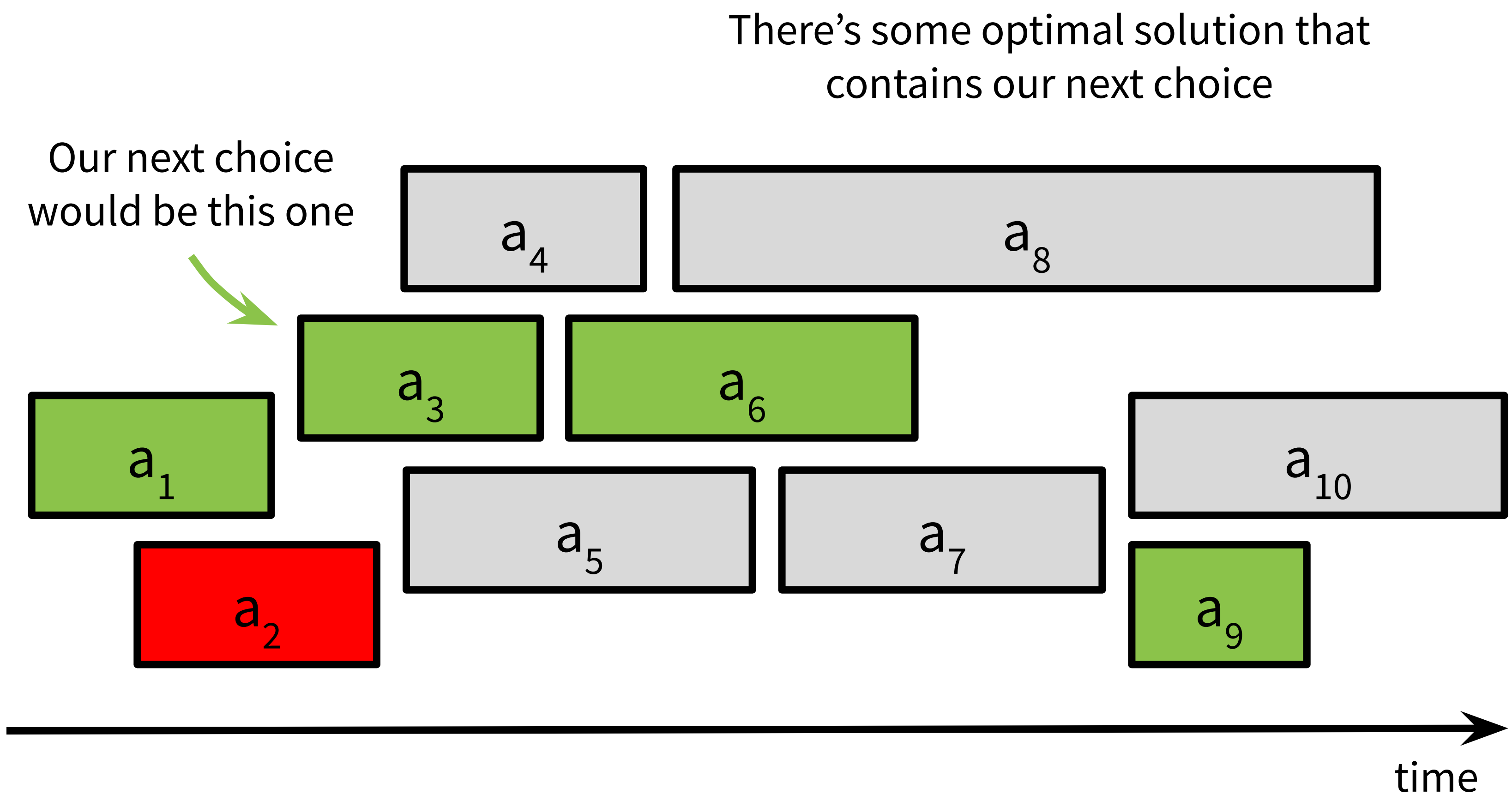
Greedy Exchange

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



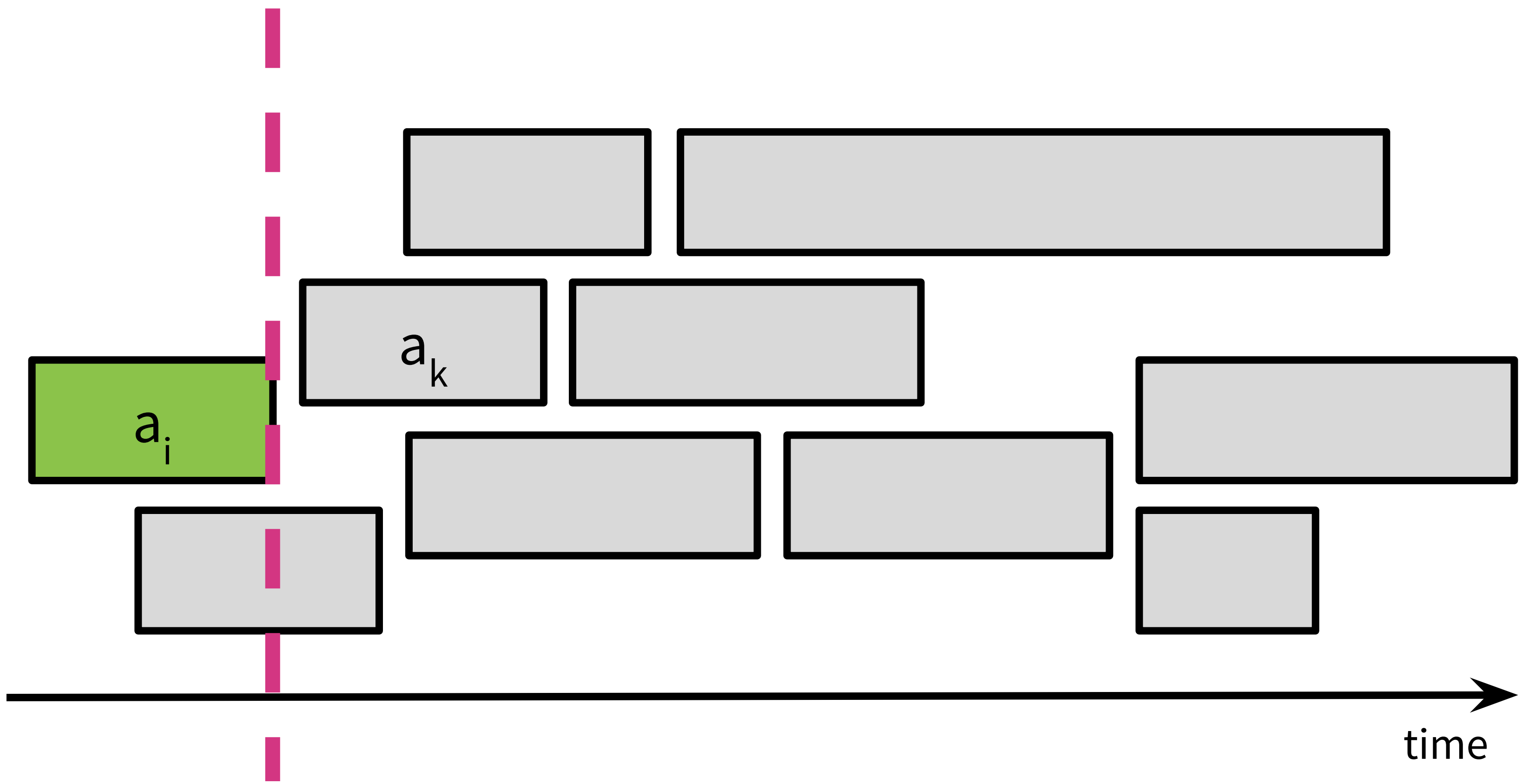
Greedy Exchange

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



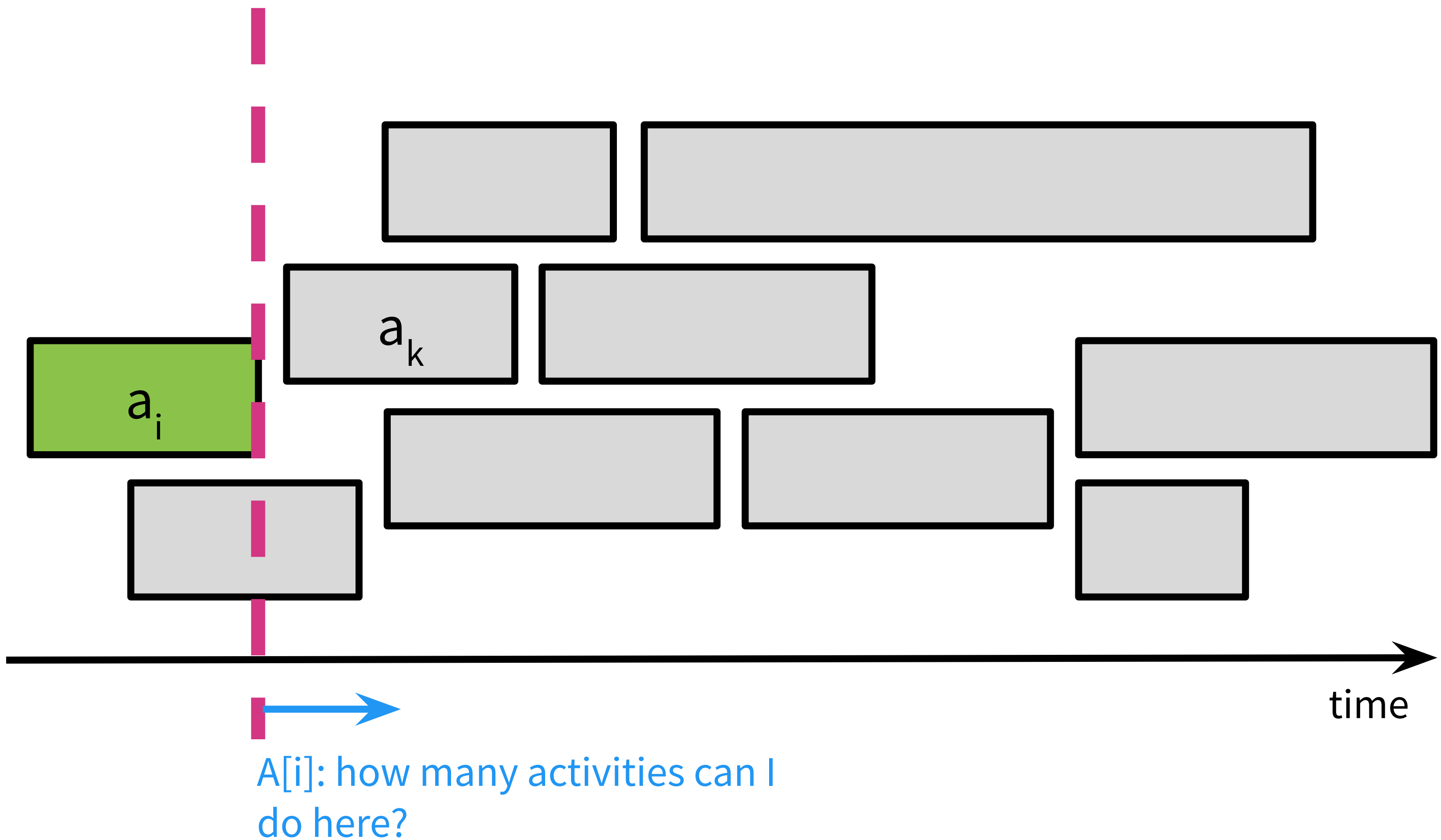
Greedy Exchange

Subproblem(i): Let A[i] be the number of activities you can do after activity i finishes.



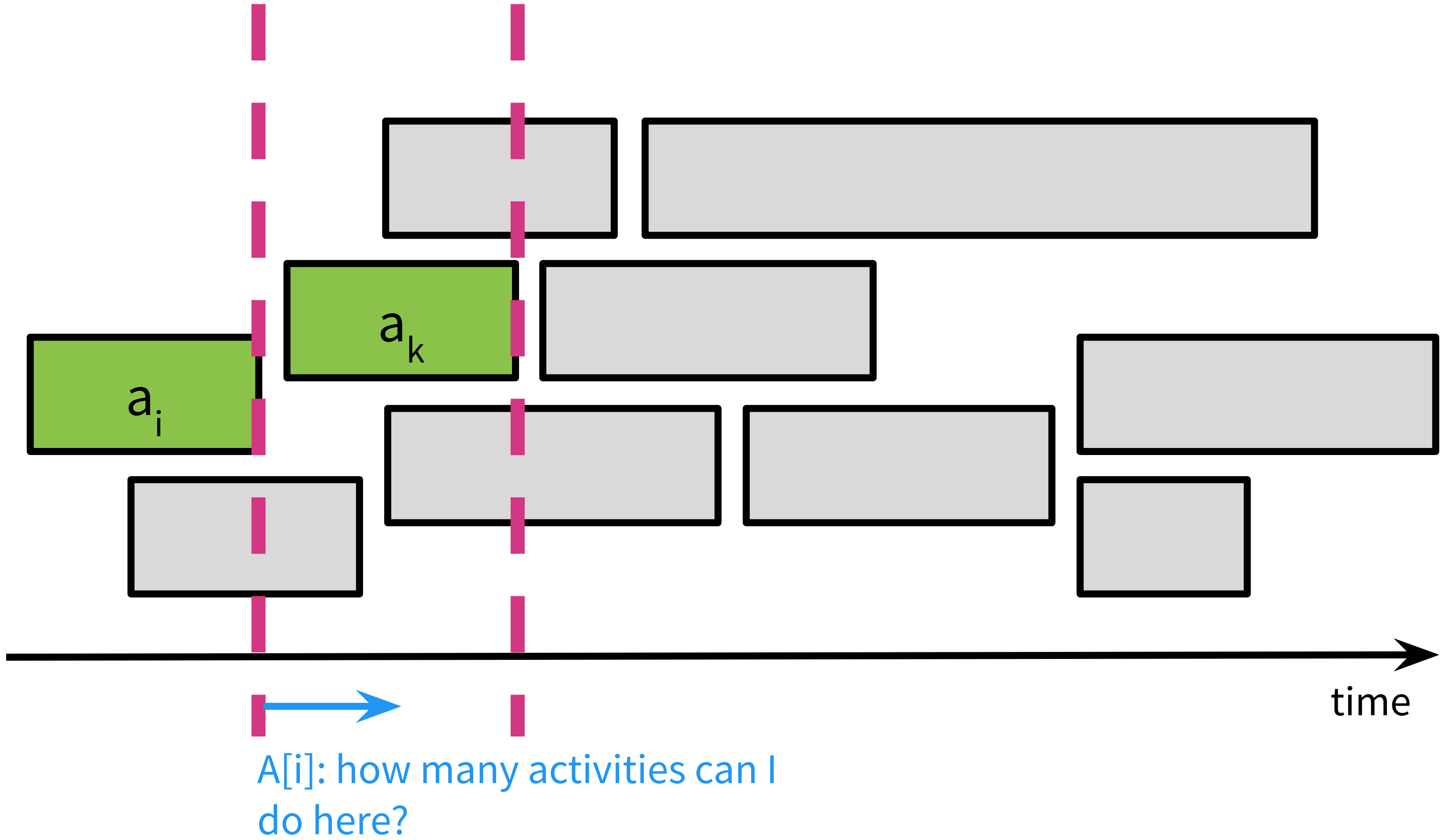
Greedy Exchange

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



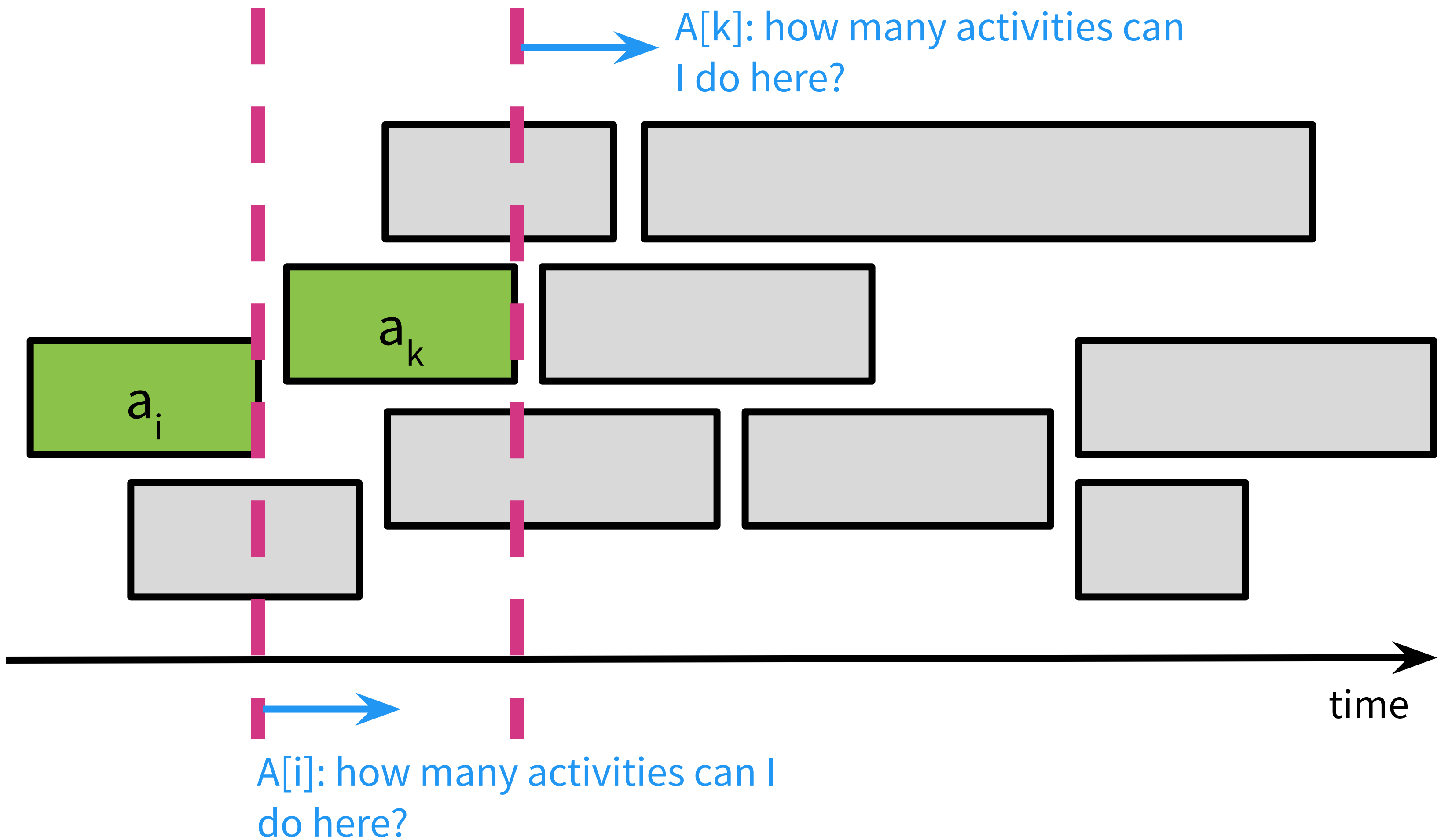
Greedy Exchange

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



Greedy Exchange

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



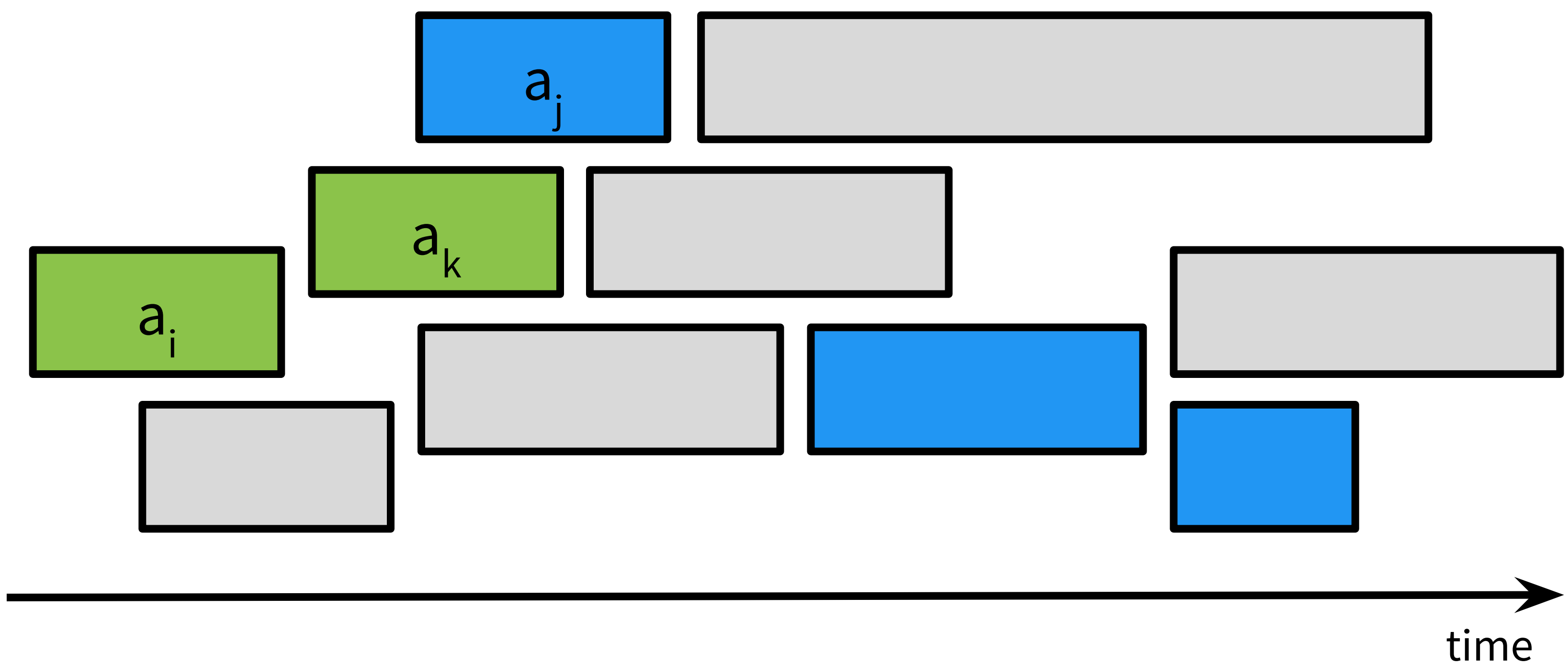
Greedy Exchange

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

- First, A[i] ≥ A[k] + 1 since we have a solution with A[k] + 1 activities.
- Suppose toward contradiction that A[i] > A[k] + 1 i.e. there's some better solution to Subproblem(i) that doesn't use a_k.

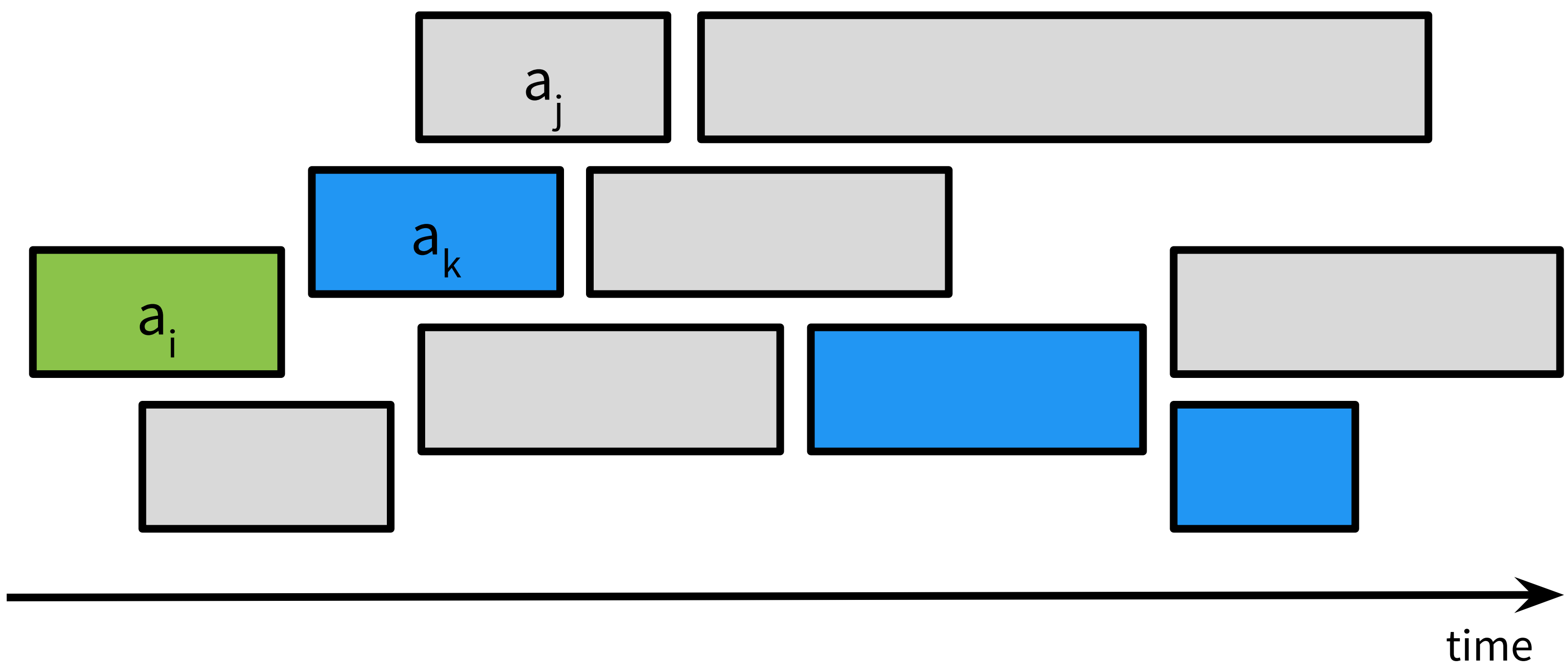
Greedy Exchange

Suppose toward contradiction that A[i] > A[k] + 1 i.e. there's some better solution to Subproblem(i) that doesn't use a_k. Let a_j be the activity that ends first in that better solution.



Greedy Exchange

Suppose toward contradiction that A[i] > A[k] + 1 i.e. there's some better solution to Subproblem(i) that doesn't use a_k. Let a_j be the activity that ends first in that better solution. Exchange a_k for a_j in that better solution.



Greedy Exchange

Suppose toward contradiction that A[i] > A[k] + 1 i.e. there's some better solution to Subproblem(i) that doesn't use a_k. Let a_j be the activity that ends first in that better solution. Exchange a_k for a_j in that better solution. Now you have a solution of the same size but it Includes a_k so it must have size ≤ A[k]+ 1 (contradiction!).