

Greedy Algorithms I

Mid-Course Review

We've covered a lot so far!

Techniques for algorithmic analysis

Asymptotics, lower-bounding functions, proofs of correctness, runtime analysis of algorithms

2 algorithmic paradigms: divide and conquer, randomized.

Randomized/graph: karger, for finding minimum cut

Divide and conquer/randomized: quicksort, quickselect

Several problems: sorting, single-source shortest path (Dijkstra's), global minimum cut (karger), hashing, SCC finding (Kosaraju's), topological sorting (by DFS), bipartite finding (by BFS).

A lot of cool stuff ahead!

2 more algorithmic paradigms: greedy algorithm (today's topic) and dynamic programming.

Approximation algorithms, amortized analysis, intractability.

Outline for Today

Greedy algorithms

- Frog Hopping

Greedy graph algorithms

- Minimum Spanning Trees

- Prim's Algorithm

- Kruskal's Algorithm

Frog Hopping

A warmup example

Greedy Algorithms

Greedy algorithms **construct solutions one step at a time**, at each step choosing the **locally best option**.

Advantages: simple to design, often efficient

Disadvantages: difficult to verify correctness or optimality

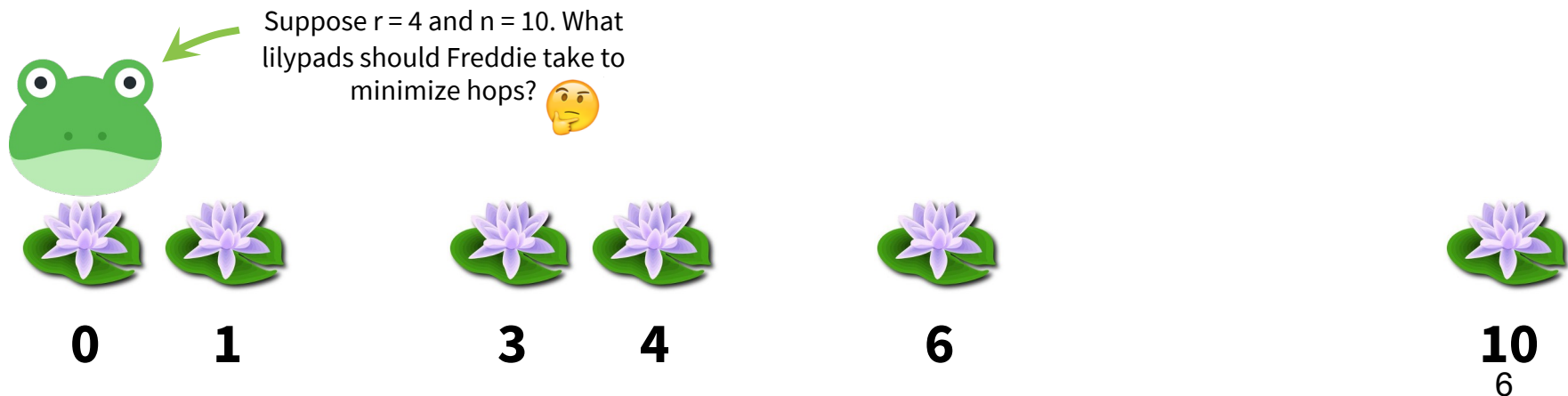
Freddie the Frog

Freddie the Frog starts at position 0 along a river. His goal is to reach position n .

There are lily pads at various positions, including at position 0 and position n .


Freddie can hop at most r units at a time.

Task: Find the path Freddie should take to minimize hops, assuming such a path exists.



Frog Hopping

```
algorithm frog_hopping(lilys, r, n):  
    # lilys = [0, 1, 3, 4, 6, 10] in the previous example  
    H = [0] # contains hops  
    cur_lily = {"index": 0, "position": 0}  
    while cur_lily["position"] < n:  
        next_lily = furthest_reachable_lily(  
            cur_lily, lilys, r  
        ) # finds the furthest lilypad still reachable  
           # from cur_lily  
        H.append(next_lily["position"])  
        cur_lily = next_lily  
    return H
```

 You should be able to implement this function yourself.

Runtime: $O(n)$

Frog Hopping

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

- (1) **Feasibility.** The algorithm finds a **feasible (aka legal) series of hops** (i.e. it doesn't "get stuck" or break any rules).
- (2) **Optimality.** The algorithm finds an **optimal series of hops** (i.e. there isn't a better path available).

Frog Hopping

Lemma 1: frog_hopping always finds a **feasible** series of hops for Freddie.

Proof: We proceed by contradiction.

Suppose it did not. A path might be infeasible for one of three reasons:

Notation
for the first
element in
list H.

 (1) $H.first \neq 0$, (2) $H[k] + r < H[k+1]$ for some k , or (3) $H.last \neq n$.

Frog Hopping

Lemma 1: frog_hopping always finds a **feasible** series of hops for Freddie.

Proof: We proceed by contradiction.

Suppose it did not. A path might be infeasible for one of three reasons:

Notation
for the first
element in
list H.

 (1) $H.first \neq 0$, (2) $H[k] + r < H[k+1]$ for some k , or (3) $H.last \neq n$.

Since the algorithm initializes H to [0], (1) is impossible.

Frog Hopping

Lemma 1: frog_hopping always finds a **feasible** series of hops for Freddie.

Proof: We proceed by contradiction.

Suppose it did not. A path might be infeasible for one of three reasons:

Notation
for the first
element in
list H.

 (1) $H.first \neq 0$, (2) $H[k] + r < H[k+1]$ for some k , or (3) $H.last \neq n$.

Since the algorithm initializes H to $[0]$, (1) is impossible.

By construction of the algorithm, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

Frog Hopping

Lemma 1: frog_hopping always finds a **feasible** series of hops for Freddie.

Proof: We proceed by contradiction.

Suppose it did not. A path might be infeasible for one of three reasons:

Notation
for the first
element in
list H.

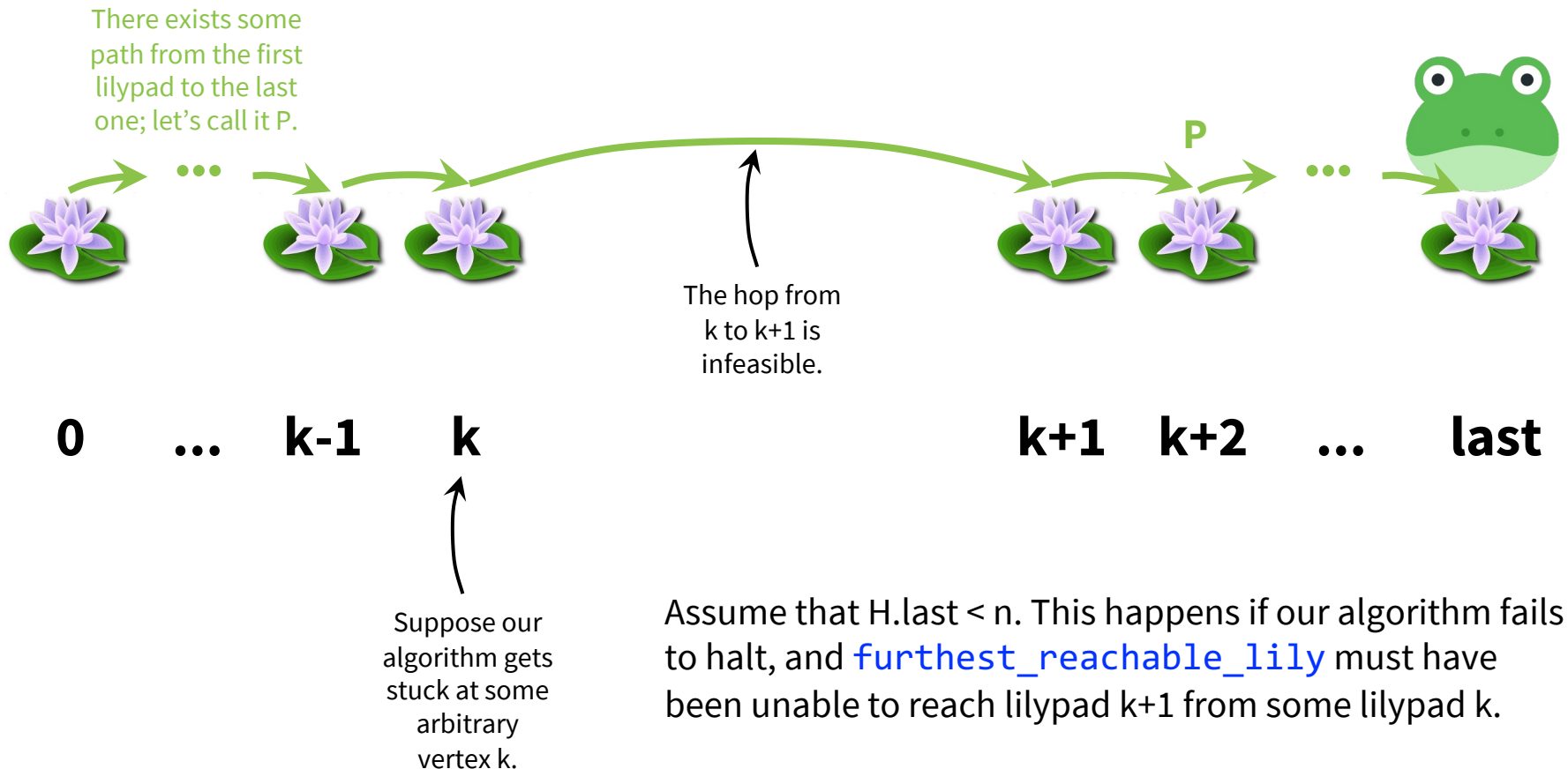
 (1) $H.first \neq 0$, (2) $H[k] + r < H[k+1]$ for some k , or (3) $H.last \neq n$.

Since the algorithm initializes H to $[0]$, (1) is impossible.

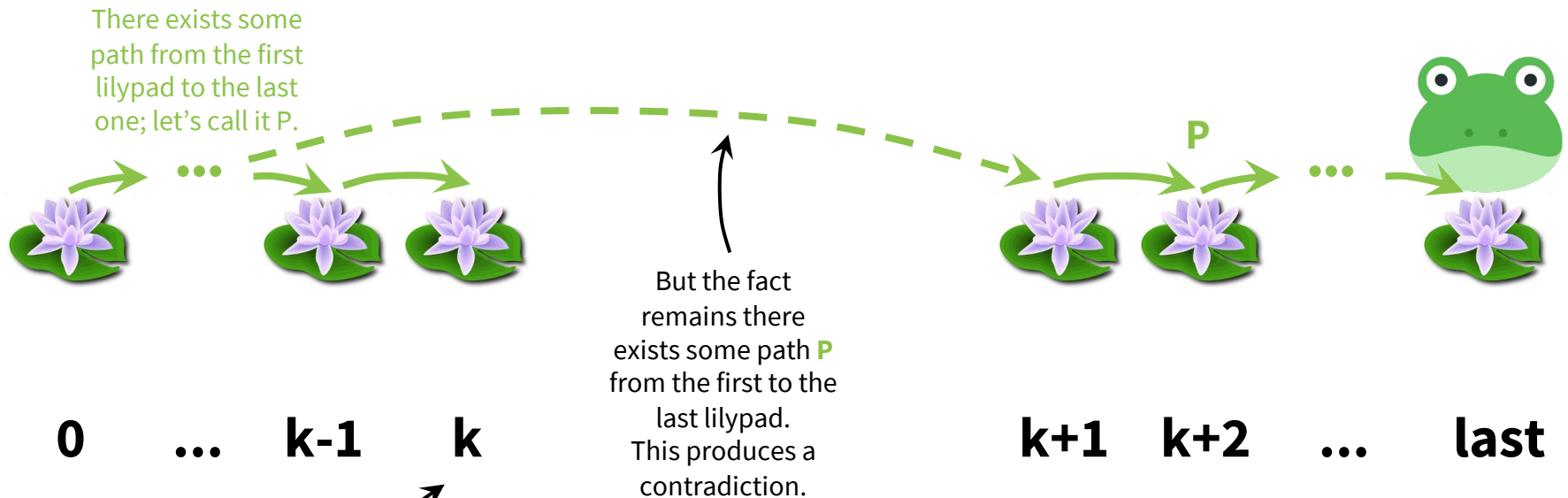
By construction of the algorithm, `next_lily` will always be reachable from the `cur_lily`; therefore (2) is impossible.

To prove that $H.last \neq n$ is impossible, we proceed by contradiction.

Frog Hopping



Frog Hopping



Suppose our algorithm gets stuck at some arbitrary vertex k .

But then for some lily $s < k$, we will have $\text{lily}[s] + r < \text{lily}[k] + r < \text{lily}[k+1]$. This would imply that any Lily pad $[0:k]$ is unreachable to Lily pad $[k+1]$, this contradicts with our knowledge that there **exists** a path from the first to the last Lily pad.

We have reached a contradiction, so our assumption must have been incorrect; therefore, (3) is impossible.

Frog Hopping

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

(1) **Feasibility.** The algorithm finds **a feasible (aka legal) series of hops** (i.e. it doesn't "get stuck" or break any rules).



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



5-Minute Break

Proof of Optimality

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` **always** finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision ...

Let H be the series of hops produced **by our algorithm** and H^* be **an arbitrary** (not necessarily **the only**) **optimal series of hops**. Then $|H|$ and $|H^*|$ denote the number of hops in H and H^* , respectively.

Note that $|H| \geq |H^*|$. Why? 🤔

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` **always** finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision ...

Let H be the series of hops produced **by our algorithm** and H^* be **an arbitrary** (not necessarily **the only**) **optimal series of hops**. Then $|H|$ and $|H^*|$ denote the number of hops in H and H^* , respectively.

Note that $|H| \geq |H^*|$. Why? 🤔 Otherwise, H^* wouldn't be optimal.

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` **always** finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision ...

Let H be the series of hops produced **by our algorithm** and H^* be **an arbitrary** (not necessarily **the only**) **optimal series of hops**. Then $|H|$ and $|H^*|$ denote the number of hops in H and H^* , respectively.

Note that $|H| \geq |H^*|$. Why? 🤔 Otherwise, H^* wouldn't be optimal.

We want to prove that $|H| = |H^*|$. How?

Frog Hopping

Now for the difficult part: How might we prove that `frog_hopping` **always** finds an optimal series of hops?

Let's introduce notation to talk about the algorithm with greater precision ...

Let H be the series of hops produced **by our algorithm** and H^* be **an arbitrary** (not necessarily **the only**) **optimal series of hops**. Then $|H|$ and $|H^*|$ denote the number of hops in H and H^* , respectively.

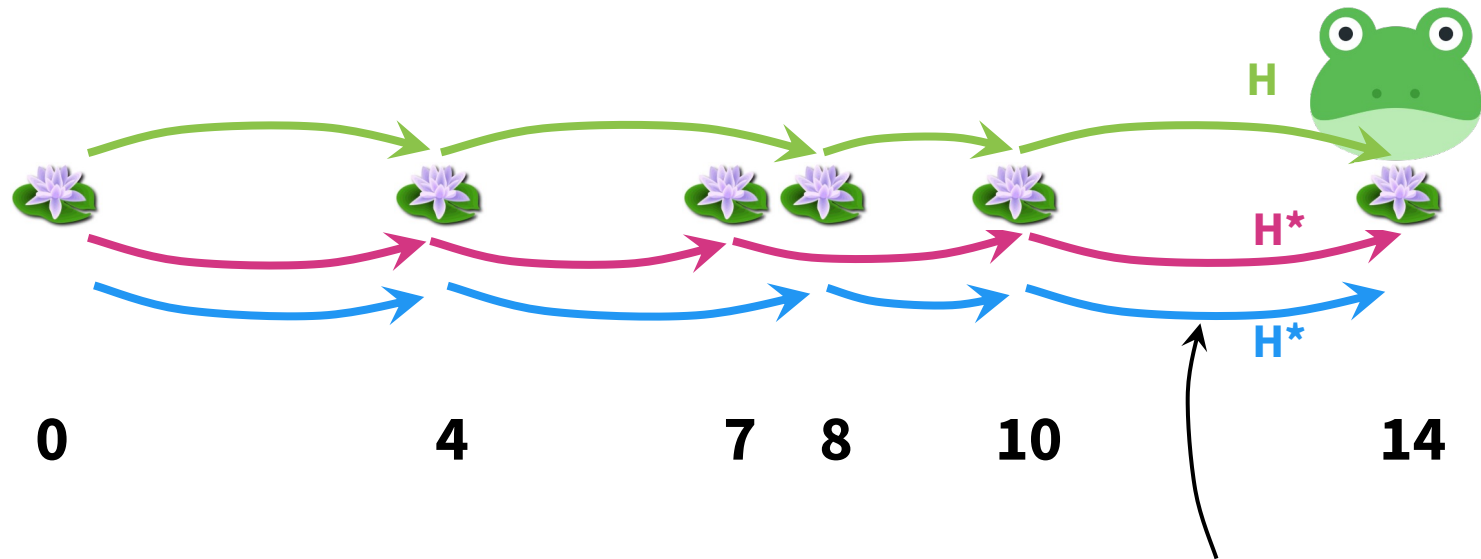
Note that $|H| \geq |H^*|$. Why? 🤔 Otherwise, H^* wouldn't be optimal.

We want to prove that $|H| = |H^*|$. How?

Intuition: Consider an arbitrary optimal series of hops H^* , then show that our greedy algorithm produces a series of hops H no worse than H^* , i.e., $|H| \leq |H^*|$

What Does Arbitrary H^* Mean?

Suppose Freddie's longest hop at each time is $r=4$.



There could be many optimal H^* (this series of lily pads has 2 optimal H^*); this proof relies on an arbitrary choice from among these H^* .

Suppose we choose H^* .

Frog Hopping

Let $p(i, H)$ denote Freddie's position after taking the first i hops from series H .

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

i.e. After taking i hops according to our greedy algorithm, Freddie will be at least as far forward as if it took i jumps according to an optimal solution.

Let's formalize this using induction.

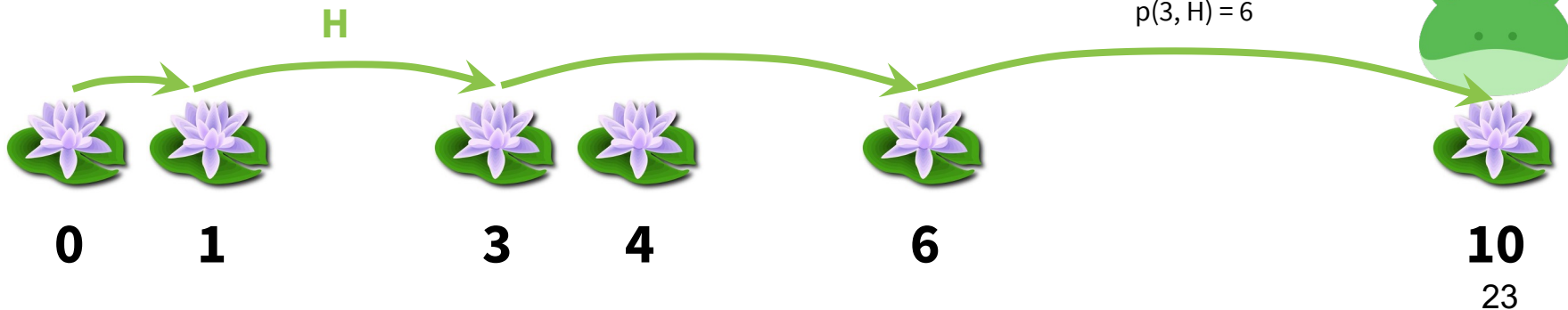
For this arbitrary H :
(unrelated to our alg)

$$p(0, H) = 0$$

$$p(1, H) = 1$$

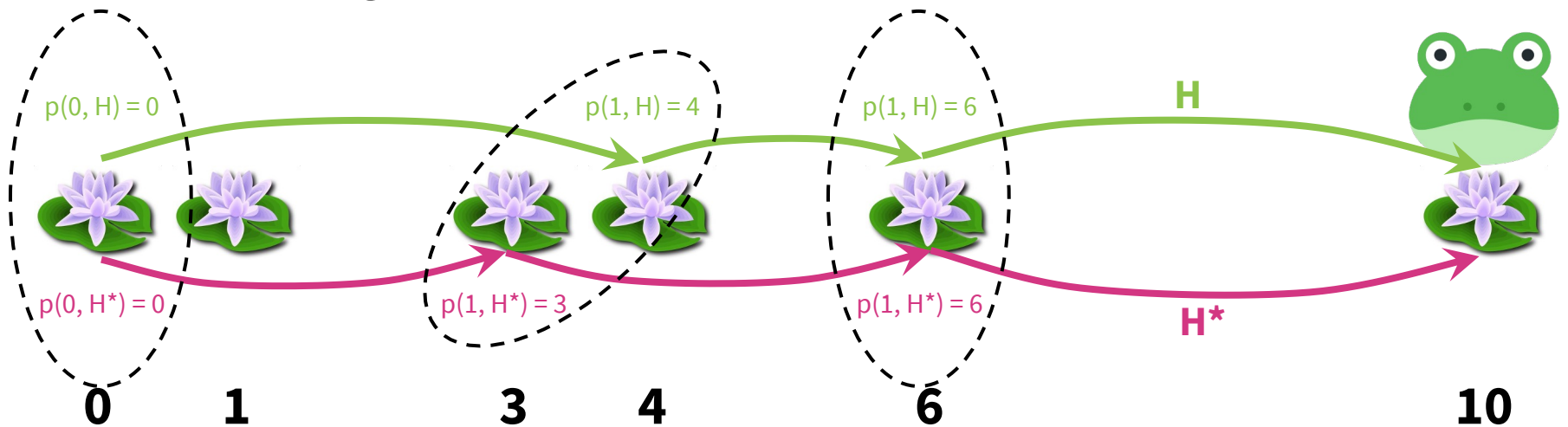
$$p(2, H) = 3$$

$$p(3, H) = 6$$



Frog Hopping

Lemma: For any i in $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from frog_hopping.



Frog Hopping

Lemma 2: For all $0 \leq i \leq |H^*|$, we have $p(i, H) \geq p(i, H^*)$, constructing H from `frog_hopping`.

Proof: We proceed by induction.

As a base case, when $i = 0$, then $p(0, H) = 0 \geq 0 = p(0, H^*)$ since Freddie hasn't moved.

For the inductive step, assume that the claim holds for some $0 \leq i < |H^*|$.

We'll prove the claim holds for $i + 1$ by considering two cases:

Case 1: $p(i, H) \geq p(i+1, H^*)$. Since each hop moves forward, we have $p(i+1, H) \geq p(i, H)$, so we have $p(i+1, H) \geq p(i+1, H^*)$.

Case 2: $p(i, H) < p(i+1, H^*)$. Each hop is of size at most r , so $p(i+1, H^*) \leq p(i, H^*) + r$. By our inductive hypothesis, we know $p(i, H) \geq p(i, H^*)$, so $p(i+1, H^*) \leq p(i, H) + r$; i.e. position $p(i+1, H^*)$ is reachable from position $p(i, H)$. Since the greedy algorithm hops to the furthest lilypad still reachable from position $p(i, H)$, it hops to at least position $p(i+1, H^*)$. Therefore, $p(i+1, H) \geq p(i+1, H^*)$.

So $p(i+1, H) \geq p(i+1, H^*)$, completing the induction. ■

Frog Hopping

Now for the theorem: `frog_hopping` produces an optimal solution for Freddie.

Frog Hopping

Theorem: frog_hopping produces an optimal solution for Freddie.

Proof:

Since H^* is an optimal solution, we know that $|H^*| \leq |H|$. We will prove $|H^*| = |H|$.
Let $k = |H^*|$. By **Lemma 2**, we have $p(k, H) \geq p(k, H^*)$. Since Freddie arrives at position n after k hops along series H^* , we know that $p(k, H) \geq p(k, H^*) = n$.
Because the greedy algorithm never hops past position n , we know $p(k, H) \leq n$.
Since $n \leq p(k, H) \leq n$, then $p(k, H) = n$.

The greedy algorithm arrives at position n after k hops, so $|H| = k$. Importantly, it's impossible to reach position n in fewer than k hops since doing so would contradict the optimality of H^* . Thus, $|H| = k = |H^*|$, so frog_hopping produces an optimal solution. ■

Frog Hopping

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

(1) **Feasibility.** The algorithm finds a feasible (aka legal) series of hops (i.e. it doesn't "get stuck" or break any rules).



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



Greedy Stays Ahead

The style of proof we just wrote is an example of a **greedy stays ahead** proof.

(1) Find **intermediate values** that **evaluate the solution produced by any algorithm**, including the greedy one.

What's our values in frog_hopping? 🤔 **The position after i hops: $p(i, H)$.**

(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 Exchange Argument

There's another style of proof that uses **greedy exchange argument**.

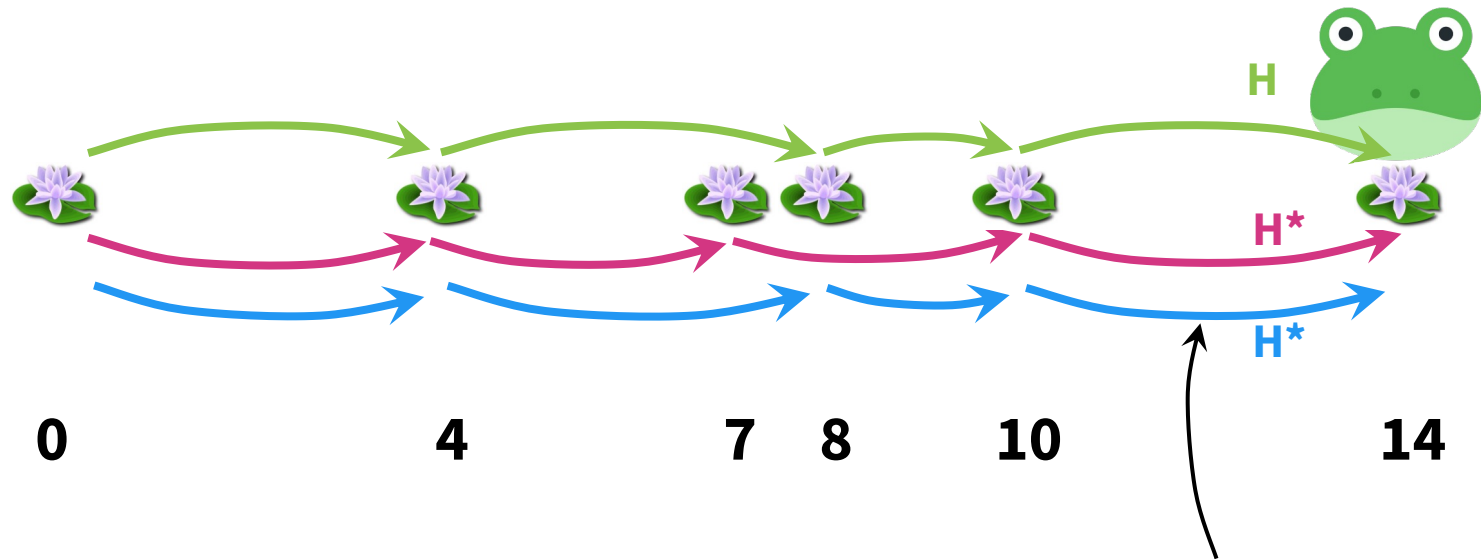
If we **swap an optimal solution out** for the greedy solution, argue that we're **still optimal**.

5-Minute Break

Greedy Exchange Argument Proof

Greedy Exchange Argument

Again, this proof will rely on an arbitrary choice of H^* .



There could be many optimal H^* (this series of lily pads has 2 optimal H^*); this proof relies on an arbitrary choice from among these H^* .

Suppose we choose H^* .

Greedy Exchange Argument

Theorem: frog_hopping produces an optimal solution.

Proof: We proceed by induction.

As a base case, we initialize H to $[\emptyset]$ and all feasible hops H^* must have $H^*[0] = 0$, as a result, $H^*[0] = H[0]$.

For the inductive step, assume that after hop j has been added to H , there exists an optimal feasible series of hops H^* such that $H^*[0..j] = H[0..j]$. We'll prove that after hop $j+1$ has been added to H , there still exists an optimal series of hops H_{new}^* such that $H_{\text{new}}^*[0..j+1] = H[0..j+1]$.

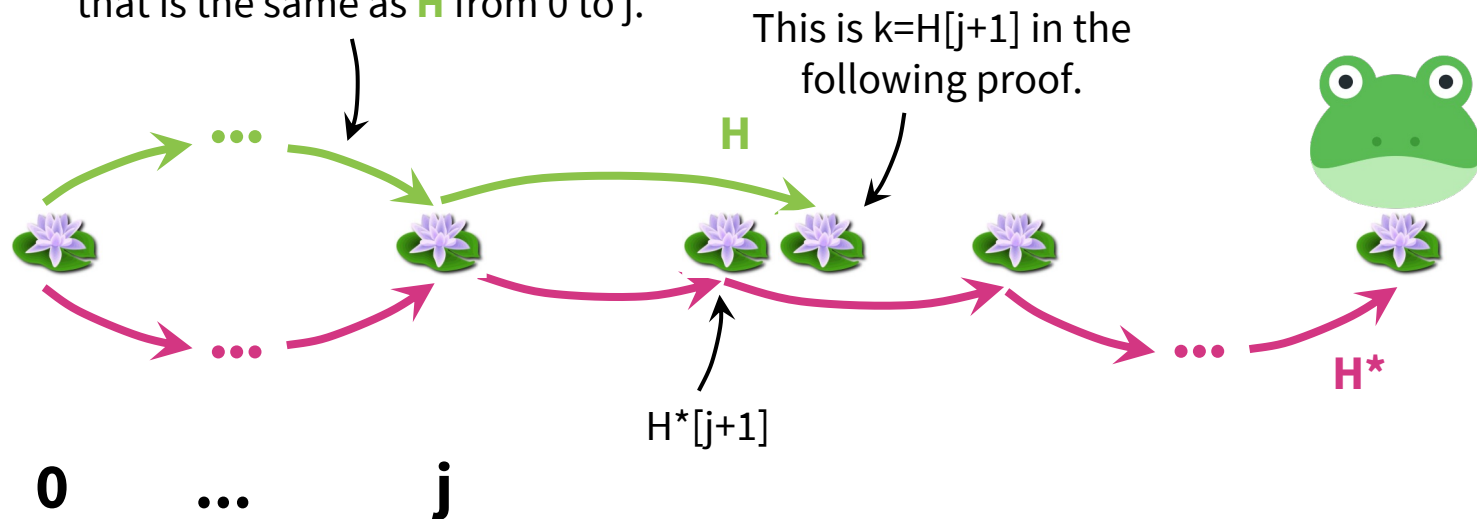
Greedy Exchange Argument

By the inductive hypothesis, H and

H^* are the same from 0 to j .

i.e., for our H , there exists an H^*

that is the same as H from 0 to j .

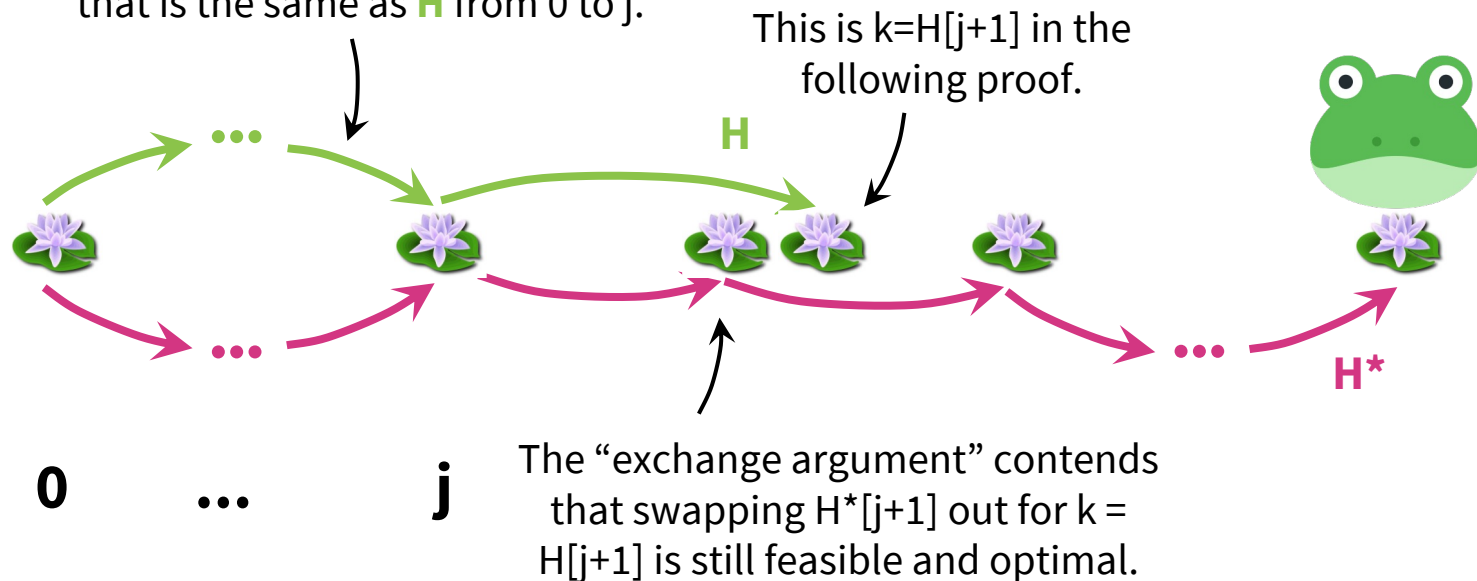


Greedy Exchange Argument

By the inductive hypothesis, H and

H^* are the same from 0 to j .

i.e., for our H , there exists an H^*
that is the same as H from 0 to j .



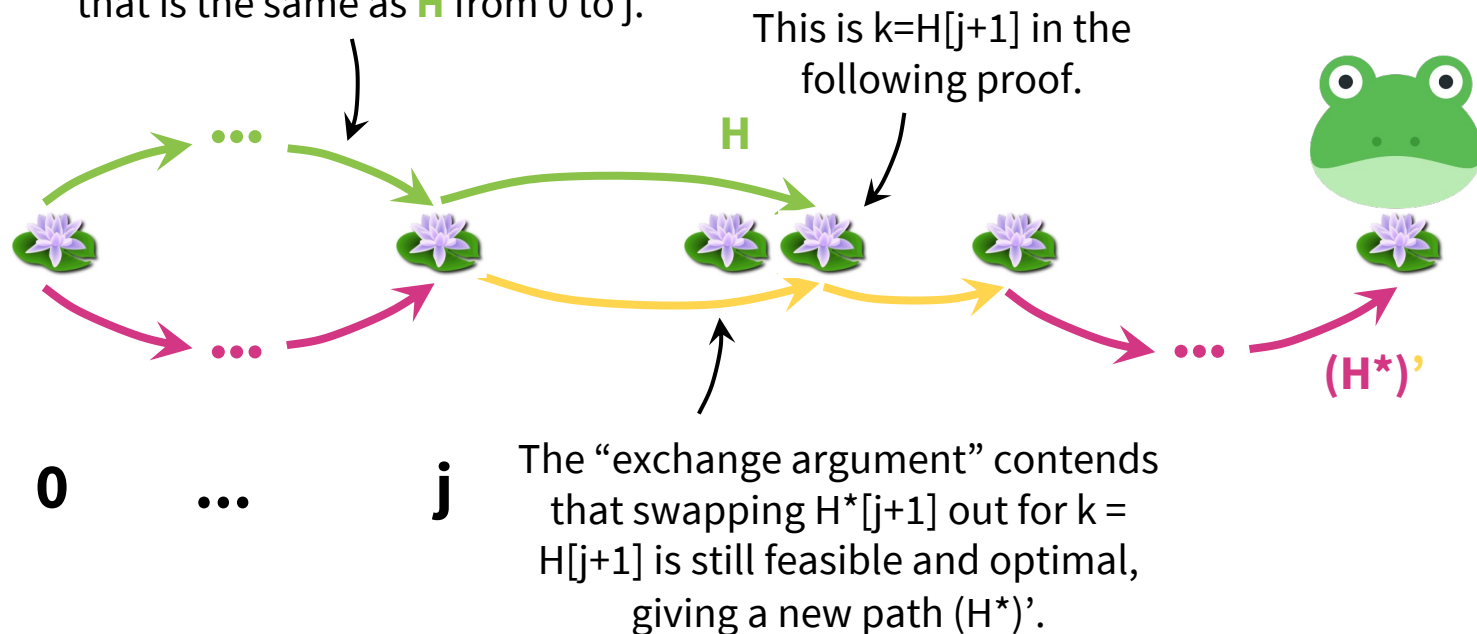
Greedy Exchange Argument

By the inductive hypothesis, H and

H^* are the same from 0 to j .

i.e., for our H , there exists an H^*

that is the same as H from 0 to j .

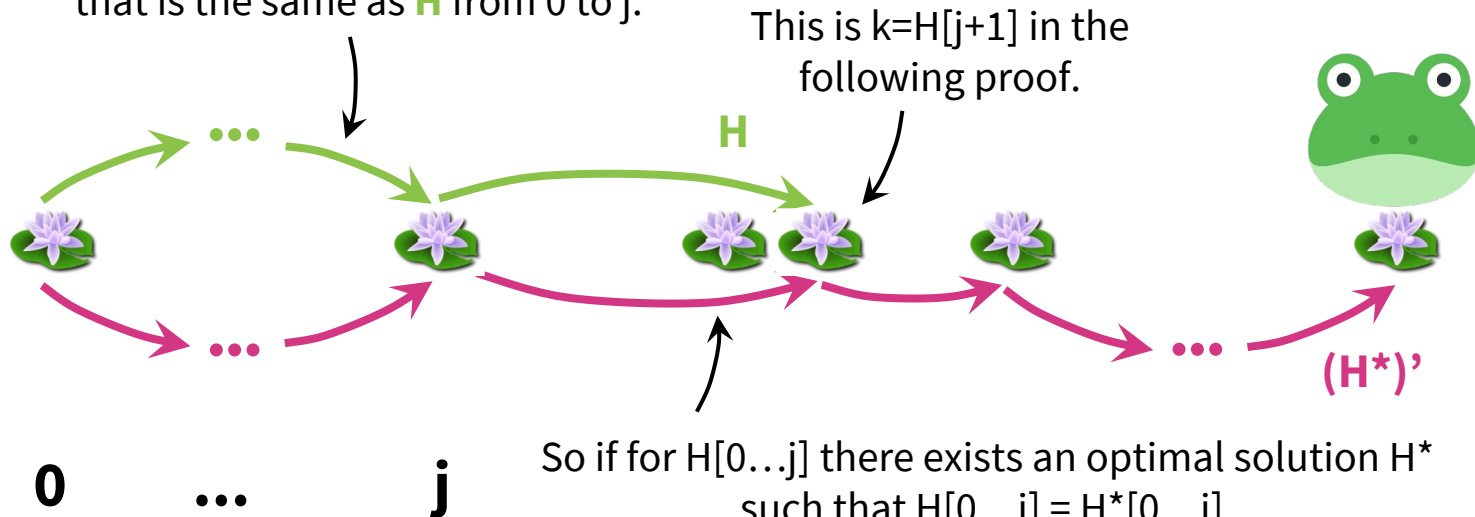


Greedy Exchange Argument

By the inductive hypothesis, H and

H^* are the same from 0 to j .

i.e., for our H , there exists an H^*
that is the same as H from 0 to j .



So if for $H[0 \dots j]$ there exists an optimal solution H^*
such that $H[0 \dots j] = H^*[0 \dots j]$,
then for $H[0 \dots j+1]$ there also exists an optimal
solution $(H^*)'$ such that $H[0 \dots j+1] = (H^*)'[0 \dots j+1]$.

So when the greedy algorithm concludes and the
algorithm returns $H[0 \dots K]$, there exists an optimal
solution H^* such as $H[0 \dots K] = H^*[0 \dots K]$. Thus H is an
optimal solution.

Greedy Exchange Argument

Theorem: frog_hopping produces an optimal solution.

Proof: We proceed by induction.


As a base case, we initialize H to $[0]$ and all feasible hops H^* must have $H^*[0] = 0$, as a result, $H^*[0] = H[0]$.

For the inductive step, assume that after hop j has been added to H , there exists an optimal feasible series of hops H^* such that $H^*[0..j] = H[0..j]$. We'll prove that after hop $j+1$ has been added to H , there still exists an optimal series of hops H_{new}^* such that $H_{\text{new}}^*[0..j+1] = H[0..j+1]$.

Let H^* be an optimal series of hops such that $H^*[0..j] = H[0..j]$. Suppose we add k as $H[j+1]$. Then $k \geq H^*[j+1]$, because (1:) $H^*[j+1] \leq r + H^*[j] = r + H[j]$ and, (2:) by construction, k is the furthest lilypad such that $k \leq r + H[j]$.

Consider $(H^*)'$ obtained from H^* by setting $H^*[j+1] = k$.

This is still feasible because (1:) $(H^*)'[j+1] = k \leq r + (H^*)'[j]$ and (2:) $(H^*)'[j+2] = H^*[j+2] \leq r + H^*[j+1] \leq r + k = r + (H^*)'[j+1]$.

 Since H^* and $(H^*)'$ are the same except at position $j+1$.

This is still optimal since $(H^*)'$ has the same number of hops as H^* .

Activity Selection

Planning Your Life

You have a list of activities $(s_1, e_1), (s_2, e_2), \dots, (s_n, e_n)$ 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 Stays Ahead

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.

Activity Selection

```
algorithm activity_selection(activities):  
    sort activities into ascending order by end time  
    U = set of all activities  
    S = an empty set  
    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: $O(n^2)$

Activity Selection

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

- (1) **Legality.** The algorithm finds a legal 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 **legal schedule**.

Intuition: Use **induction** to show that at each step, the set U only contains activities that **do not conflict with the selected activities in S** .

Activity Selection

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

- (1) **Legality.** The algorithm finds a **legal 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

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 k-th 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).

Activity Selection

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

Let's introduce notation to talk with greater precision about the algorithm ...

Let S be the schedule produced by our algorithm and S^* be **an arbitrary** (not necessarily **the only**) **optimal** schedule. Then $|S|$ and $|S^*|$ denote the number of activities in S and S^* , respectively.

Note that $|S| \leq |S^*|$. Why? 🤔 Because otherwise S^* would not be optimal

We want to prove that $|S| = |S^*|$. How?

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

Activity Selection

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

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

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!

Proving Optimality

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 claim holds for $i + 1$. According to the induction assumption, we have $f(i, S) \leq f(i, S^*)$, which means the i -th activity in S finishes before the i -th activity in S^* finishes. Since the $(i+1)$ -th activity in S^* must start after the i -th activity in S^* ends (otherwise the two activities in S^* will conflict), we have $f(i, S^*) \leq b(i+1, S^*)$, ($b(^*)$ means the beginning time of an activity). Combining the two inequalities, we have $f(i, S) \leq b(i+1, S^*)$, meaning that the $(i+1)$ -th activity in S^* must start after the i -th activity in S ends.

Therefore, the $(i+1)$ -th activity in S^* must remain in U after you select the i -th activity for S . So when you use the greedy algorithm to select the $(i+1)$ -th activity for S , you will select the activity in U with the lowest end time, so we must have $f(i+1, S) \leq f(i+1, S^*)$, completing the induction.

Proving Optimality

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 $|S| \neq |S^*|$, we must have $|S| < |S^*|$.

Let $k = |S|$. By our lemma, we know $f(k, S) \leq f(k, S^*)$, so the k -th activity in S finishes no later than the k -th activity in S^* . Since $|S| < |S^*|$, there is a $(k+1)$ -th 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 k -th activity to S , the $(k+1)$ -th activity from S^* would still belong to U , because it does not conflict with $f(k, S)$.

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. ■



Activity Selection

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

- (1) **Legality.** The algorithm finds a legal 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

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

- (1) **Legality.** The algorithm finds a legal 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). 

Acknowledgement: Part of the materials are adapted from Virginia Williams and David Eng's lectures on algorithms. We appreciate their contributions.