

CS 170

Efficient Algorithms and Intractable Problems

Lecture 8 Greedy Algorithms

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Quizzes and Midterms:

- In-class Quiz on next lecture (09/24)
- Midterm on Oct 1.

Quiz's scope:

- **30 minutes long** and will consist of **2 questions**
- Scope is HW1-3 and Disc 1-3.

Today and Next 2 Lecture: Greedy Algorithms

Algorithms that build up a solution

piece by piece, always choosing the next piece

that offers the most obvious and immediate benefit!

Examples of problems where greedy works

- Scheduling

- Huffman Coding

- Satisfiability (future lectures)

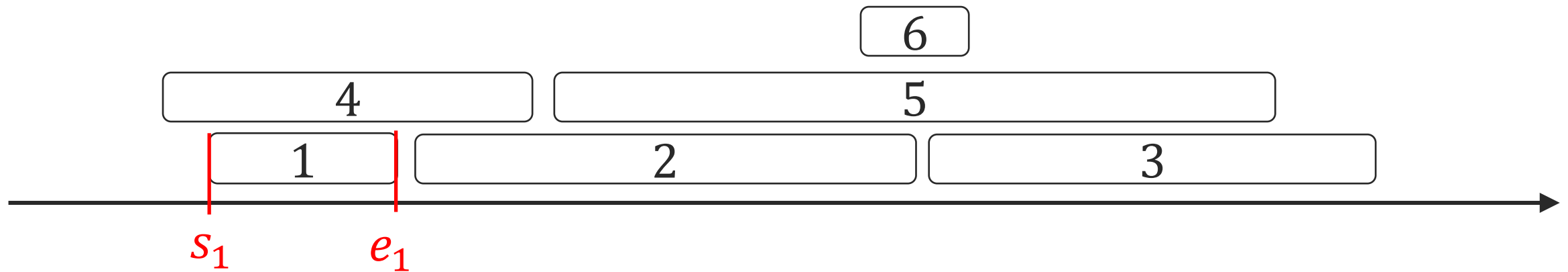
- Minimum spanning trees (future lectures)



(Interval) Scheduling

Input: collection of n jobs specified by their time intervals $[s_1, e_1], \dots, [s_n, e_n]$.

Goal: Find the largest subset of jobs, that have no time conflicts.



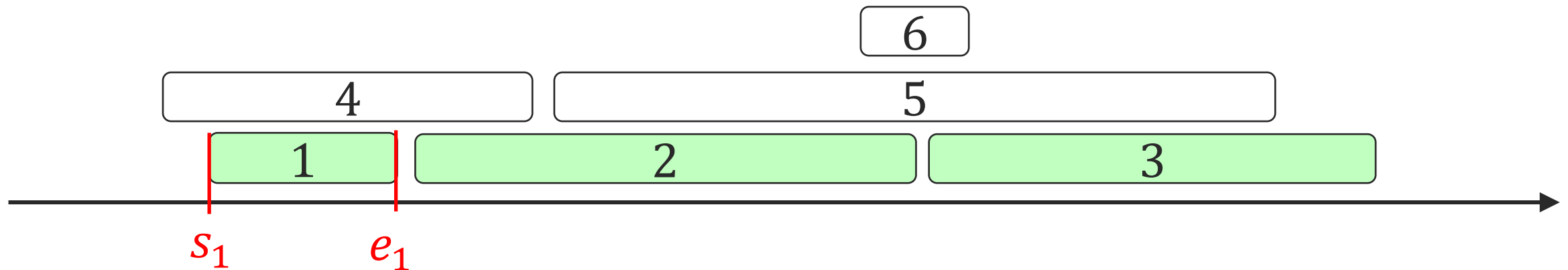
Application example:

- intervals denote activities you are interested in.
→ classes take, times you can hangout with friends, time for rest, appointments, ...
- You want to do as many activities as possible!

(Interval) Scheduling

Input: collection of n jobs specified by their time intervals $[s_1, e_1], \dots, [s_n, e_n]$.

Goal: Find the largest subset of jobs, that have no time conflicts.

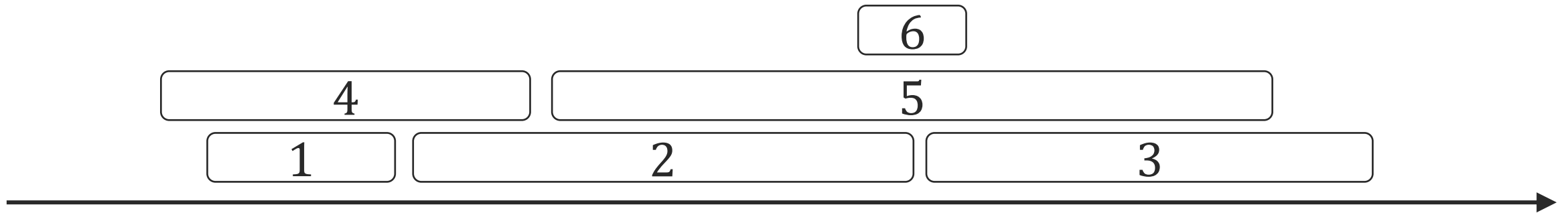


Discuss

Let's pick greedily! Which interval should we pick next?

- Shortest job?
- Earliest start time?
- Earliest end time?

Pick the earliest finish time, and repeat!



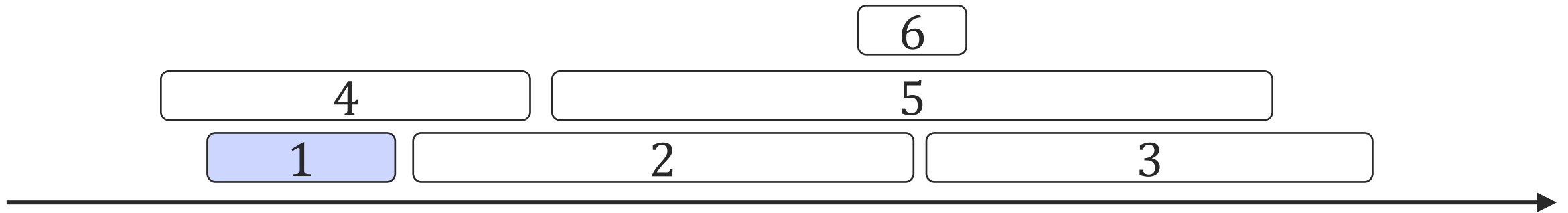
Algorithm:

While the set of intervals is non-empty

 Add interval j with the earliest finish time e_j .

 Remove any conflicted interval i from the set, i.e., $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$

Pick the earliest finish time, and repeat!



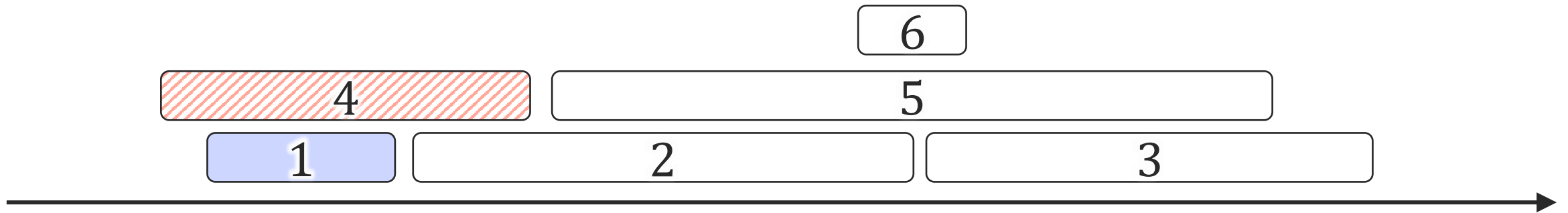
Algorithm:

While the set of intervals is non-empty

 Add interval j with the earliest finish time e_j .

 Remove any conflicted interval i from the set, i.e., $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$

Pick the earliest finish time, and repeat!



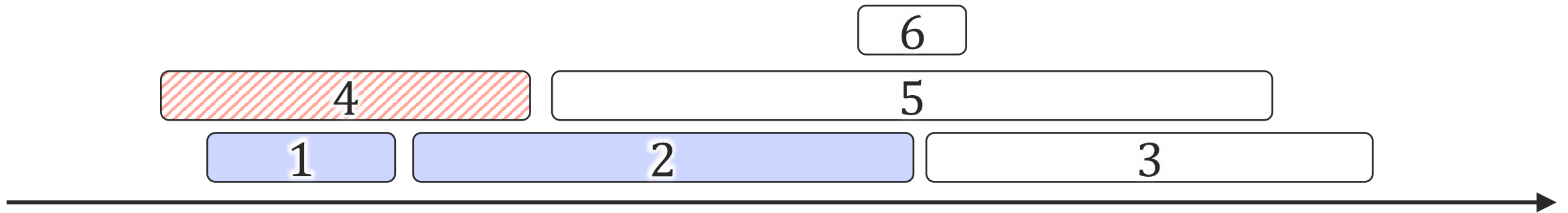
Algorithm:

While the set of intervals is non-empty

 Add interval j with the earliest finish time e_j .

 Remove any conflicted interval i from the set, i.e., $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$

Pick the earliest finish time, and repeat!



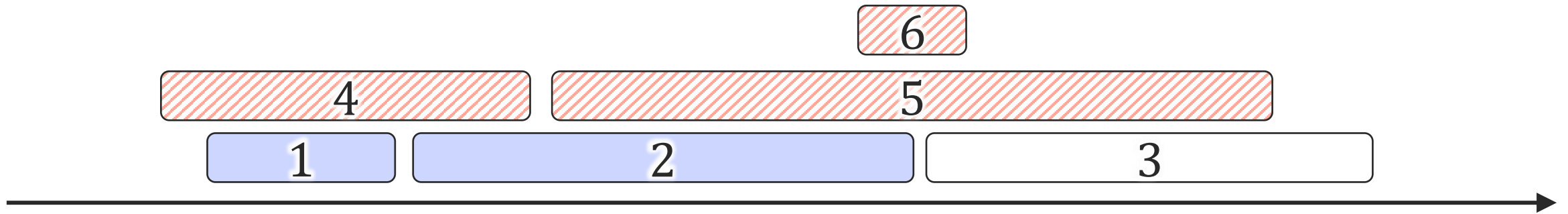
Algorithm:

While the set of intervals is non-empty

 Add interval j with the earliest finish time e_j .

 Remove any conflicted interval i from the set, i.e., $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$

Pick the earliest finish time, and repeat!



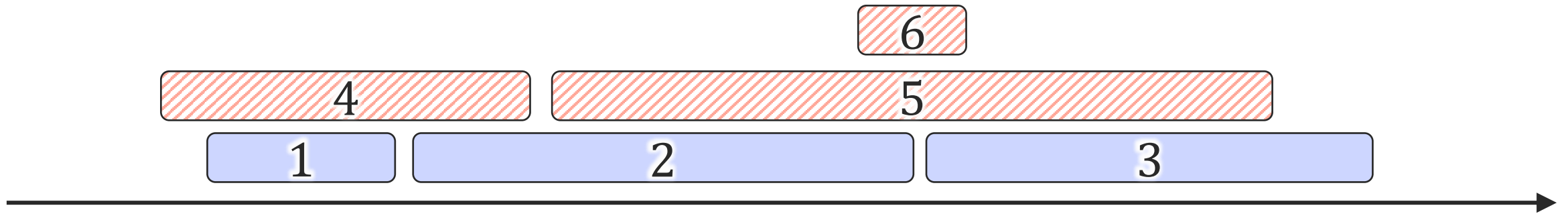
Algorithm:

While the set of intervals is non-empty

 Add interval j with the earliest finish time e_j .

 Remove any conflicted interval i from the set, i.e., $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$

Pick the earliest finish time, and repeat!



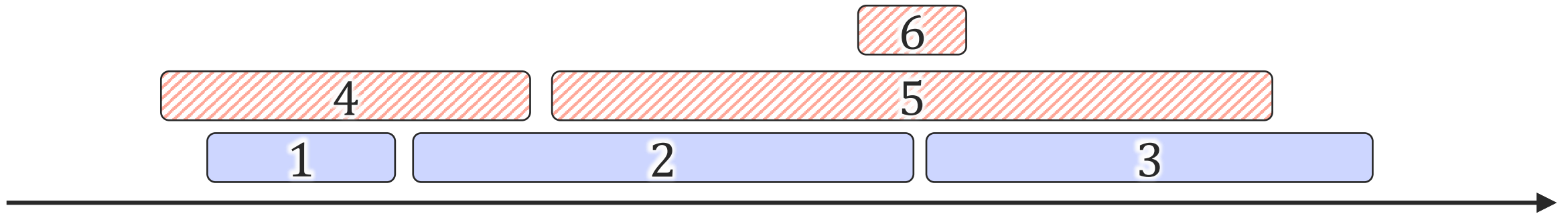
Algorithm:

While the set of intervals is non-empty

 Add interval j with the earliest finish time e_j .

 Remove any conflicted interval i from the set, i.e., $[s_j, e_j] \cap [s_i, e_i] \neq \emptyset$

Pick the earliest finish time, and repeat!



Why is this greedy algorithm correct? We'll see in a minute.

What's the runtime of this algorithm?

- $O(n)$ if the intervals are already sorted by finish time.
→ How? Remember job j that was added last. Next time when looking at candidate job j' , only add it if $s_{j'} > e_j$.
- Otherwise $O(n \log(n))$ if we have to sort them by the finish time.

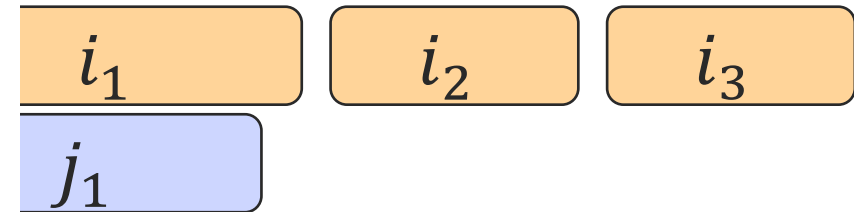
Why does greedy work for interval scheduling?

Whenever we make a choice to include an interval in the solution, **we don't rule out an optimal solution.**

→ So, intuitively after we are done, we have an optimal solution.

Intuition for why we never rule out an optimal solution

OPT = $i_1, i_2, i_3, \dots, i_k, i_{k+1}$
Greedy = $j_1, j_2, j_3, \dots, j_k$



In OPT: Swap in j_1 for i_1 .

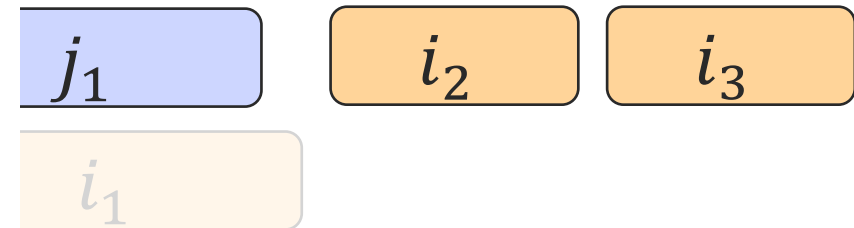
Why does greedy work for interval scheduling?

Whenever we make a choice to include an interval in the solution, **we don't rule out an optimal solution.**

→ So, intuitively after we are done, we have an optimal solution.

Intuition for why we never rule out an optimal solution

OPT = $i_1, i_2, i_3, \dots, i_k, i_{k+1}$
Greedy = $j_1, j_2, j_3, \dots, j_k$



In OPT: Swap in j_1 for i_1 .

More formal argument: Proof by induction

Claim: For any $m \leq k$, there is an optimal schedule **OPT** that agrees with greedy's solution G , on the first m intervals.

Recipe for Greedy Algorithm and Analyses

Greedy makes a series of choices. We show that no choice rules out the optimal solution. How?

Inductive Hypothesis:

- The first m choices of greedy match the first m steps of *some* optimal solution.
- Or, after greedy makes m choices, achieving optimal solution is still a possibility.

Base case: → At the beginning, achieving optimal is still possible!

Inductive step: **Use problem-specific structure**

If the first m choices match, we can change OPT's $m + 1^{st}$ choice to that of greedy's, and still have a valid solution that's no worse than OPT.

Conclusion: The greedy algorithm outputs an optimal solution.

Little Break + Attendance

Lecture 8

Activate Location Services

Log into iClicker:

<https://join.iclicker.com/ASAN>

Check-in window will close in 3
minutes (end of break)



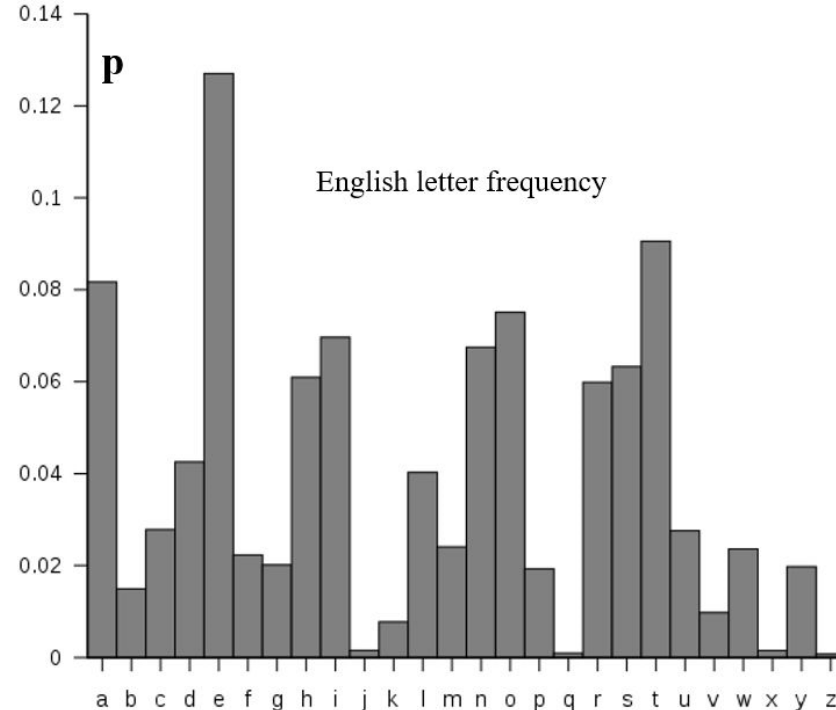
Scan for iClicker

Data Compression and Encoding

Common encodings of English characters use a fixed length of code per character.

If the goal is to save space, can we encode the alphabet better?

- If we know which letters are more common
- Use shorter codes for very common characters (like e, a, s, t).



Example of encodings

Assume we just have 4 letters, A, B, C, D with associated frequencies.

Freq.	Letter	Encoding #1	Encoding #2	Encoding #3
0.4	<i>A</i>			
0.2	<i>B</i>			
0.3	<i>C</i>			
0.1	<i>D</i>			
Total cost				

Example of encodings

Assume we just have 4 letters, A, B, C, D with associated frequencies.

Freq.	Letter	Encoding #1	Encoding #2	Encoding #3
0.4	<i>A</i>	00	0	0
0.2	<i>B</i>	01	00	110
0.3	<i>C</i>	10	1	10
0.1	<i>D</i>	11	01	111
Total cost		$2N$	$(0.4 + 0.3) \times N + (0.1 \times 2 + 0.2 \times 2) \times N$ $= 1.3N$	$0.4 \times N + 0.3 \times 2N + (0.1 \times 3 + 0.2 \times 3) \times N$ $= 1.9N$

But encoding #2 is lossy: What does 000 represent? AB or BA?

Encoding #3: No code is a prefix of another.

→ There is only one way to interpret any code.

Any Prefix codes and Trees

A

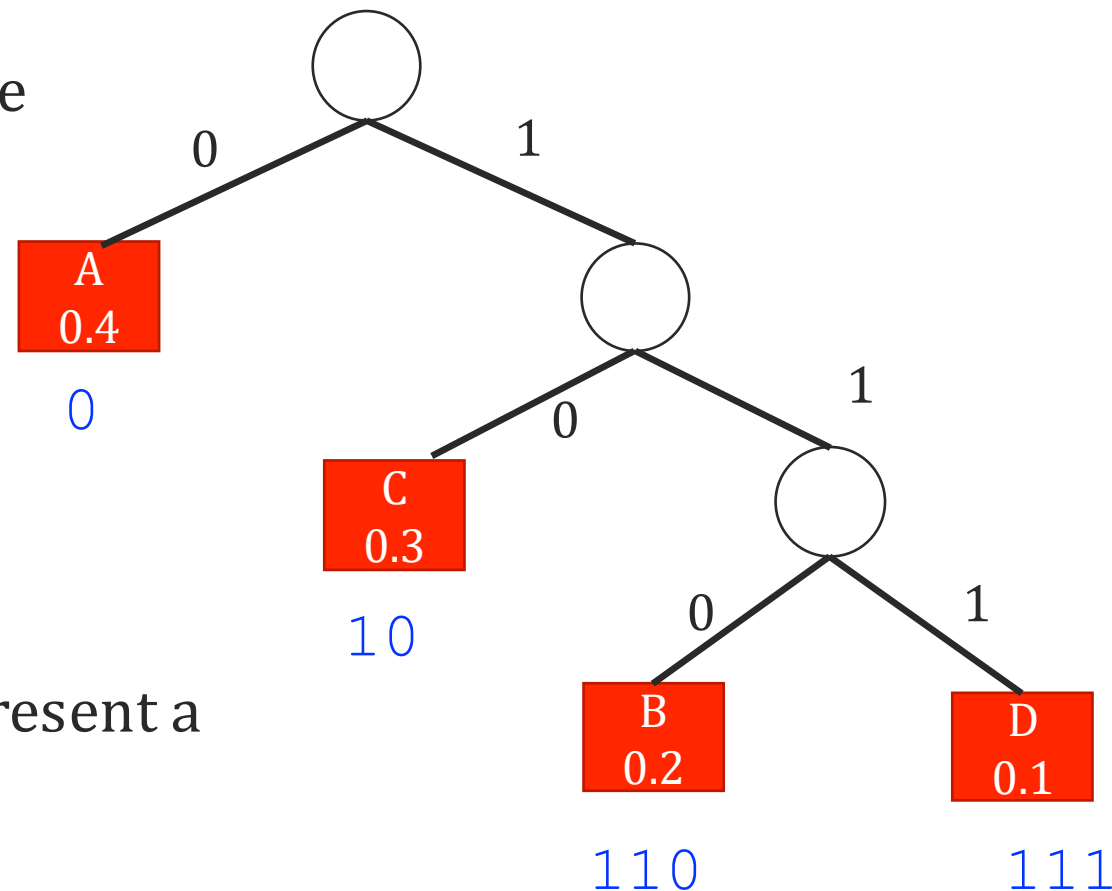
0.4

means "A" has freq. 0.4.

Prefix free code: No code **x** is a prefix of another code **z**.

Any **prefix-free code** on n letters can be represented as a binary tree with n **leaves**.

- **Leaves** indicate the coded letter
- The **code** is the "address" of a letter in the tree



Any tree with the letters at the leaves, also represent a prefix-free code.

Tree and Code Size

A
0.4

means "A" has freq. 0.4.

Imagine we are encoding a length N text:

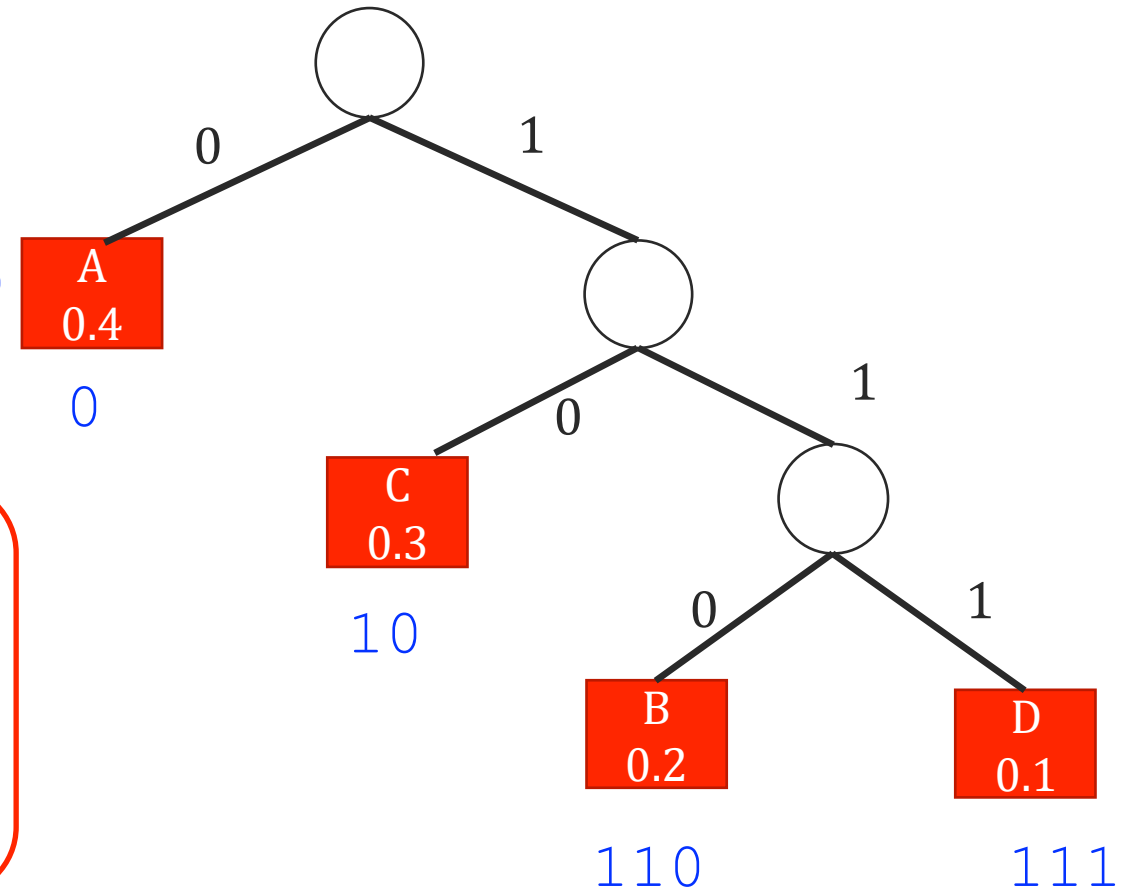
→ that is written in n letters with frequencies $f_1, f_2, \dots, f_n$.

How long is the encoded message?

$$\text{length of encoding} = \sum_{i=1}^n N \cdot f_i \cdot \text{len}(\text{encoding } i)$$

Definition: Cost of a prefix-code/tree is

$$\text{Cost}(\text{tree}) = \sum_{i=1}^n f_i \cdot \text{depth}(\text{leaf } i)$$



Optimal Prefix-free Codes

Input: n symbols with frequencies $f_1, \dots, f_n$

Output: A tree (prefix-free code) encoding.

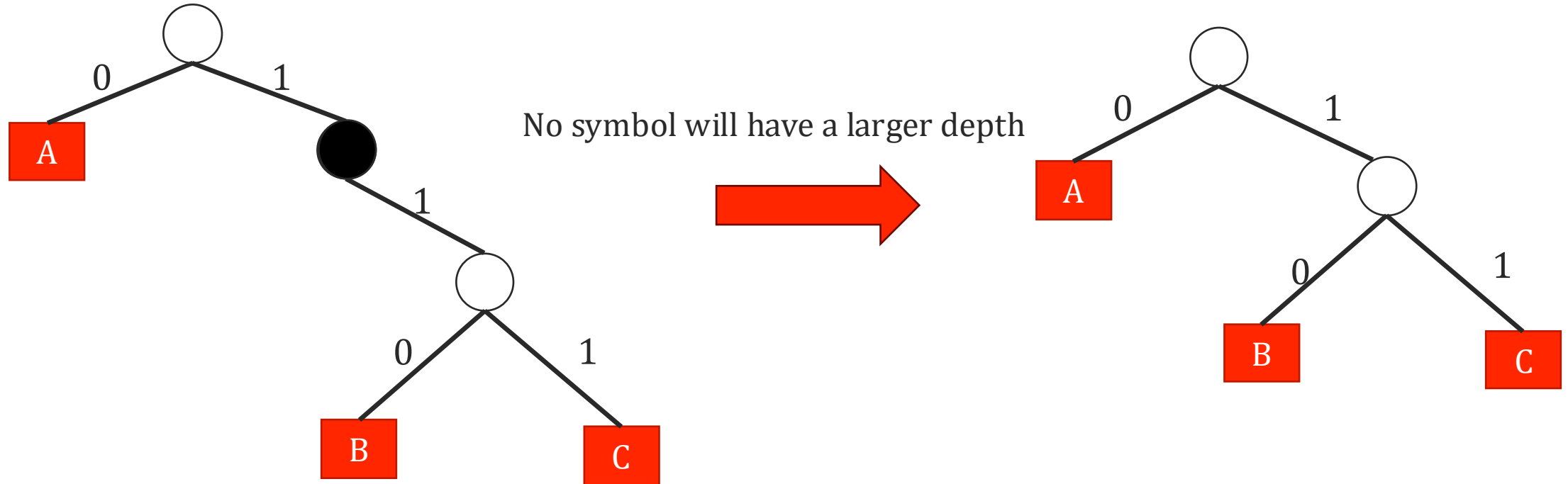
Goal: We want to output the tree/code with the smallest cost

$$\text{Cost}(\text{tree}) = \sum_{i=1}^n f_i \cdot \text{depth}(\text{leaf } i)$$

What do optimal subtrees look like?

Discuss

Even without looking at the frequencies, could this tree be optimal?



Claim: There is a “full binary tree” that is an optimal coding.

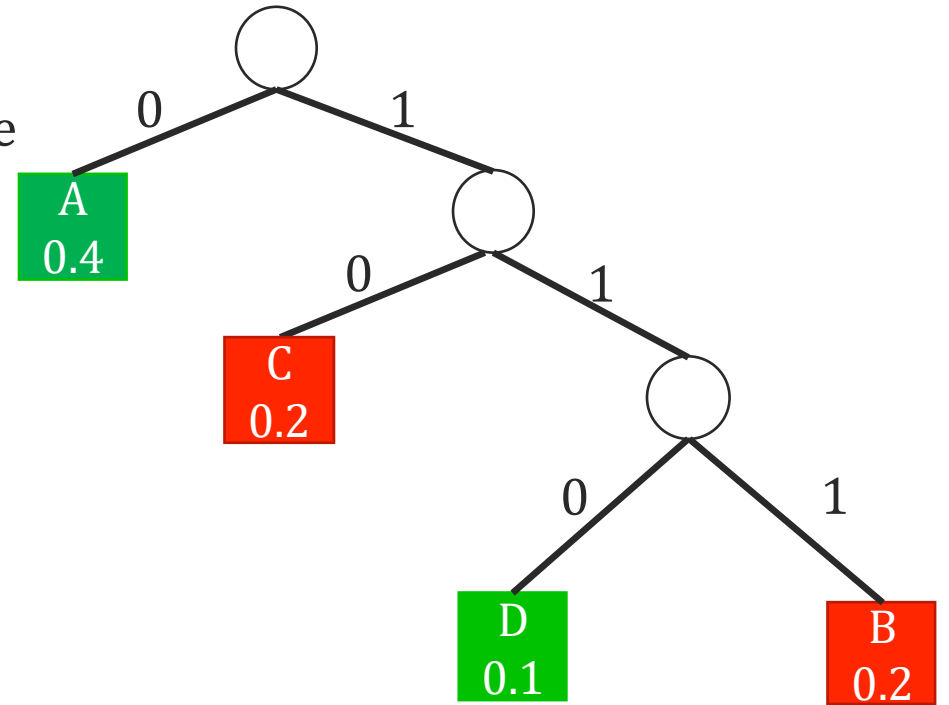
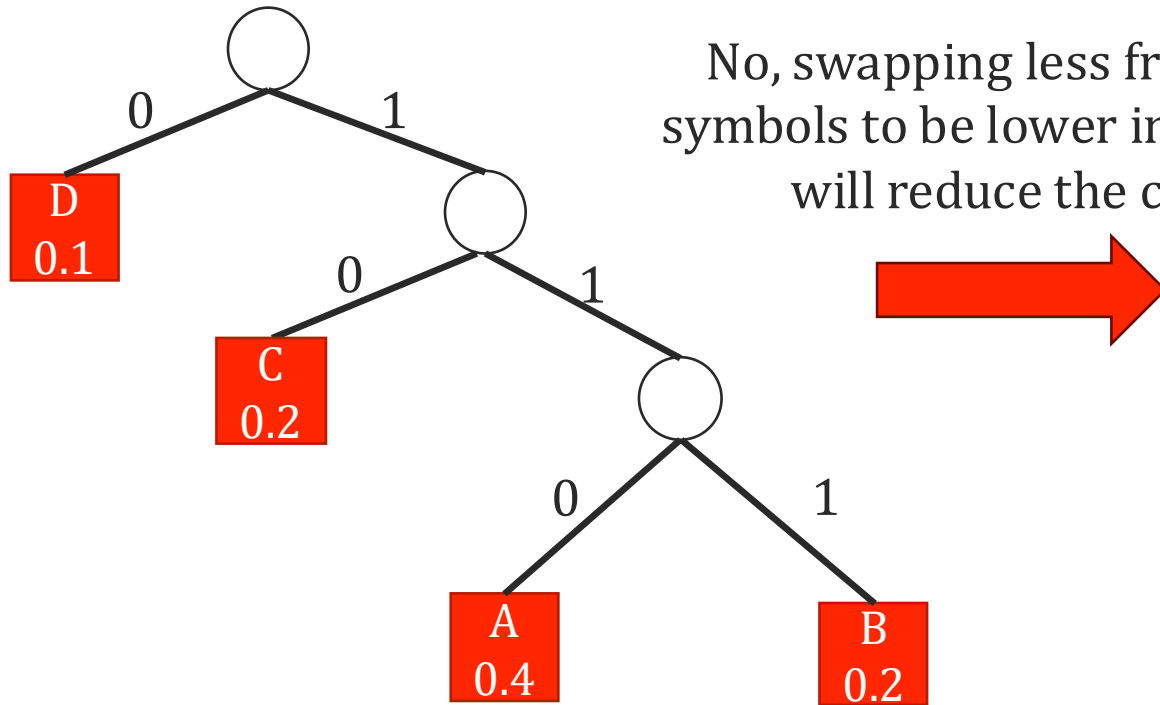
Proof: we just argued above!

Means that every non-leaf node has two children.

What do optimal subtrees look like?

Discuss

Is the following an optimal coding?



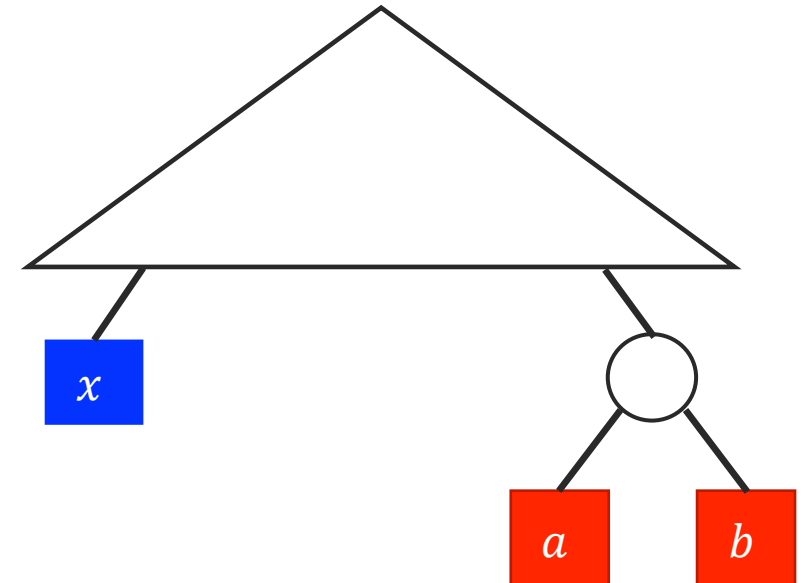
What do optimal subtrees look like?

Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols a, b be the deepest pair of siblings.
- A lowest sibling pair exists because we have a full binary tree.
- At least one of a, b is neither x or y . Let's say $x \neq a$.

What happens if we swap x and a ?



What do optimal subtrees look like?

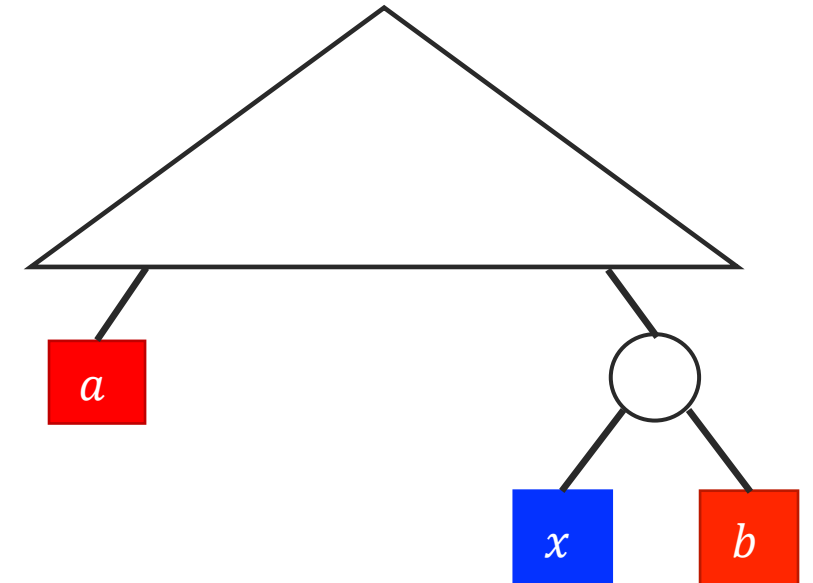
Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols a, b be the deepest pair of siblings.
- A lowest sibling pair exists because we have a full binary tree.
- At least one of a, b is neither x or y . Let's say $x \neq a$.

What happens if we swap x and a ?

→ **The cost of tree can't increase**, because $f_a \geq f_x$ and we just switch the length of a 's code and x 's code.



What do optimal subtrees look like?

Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

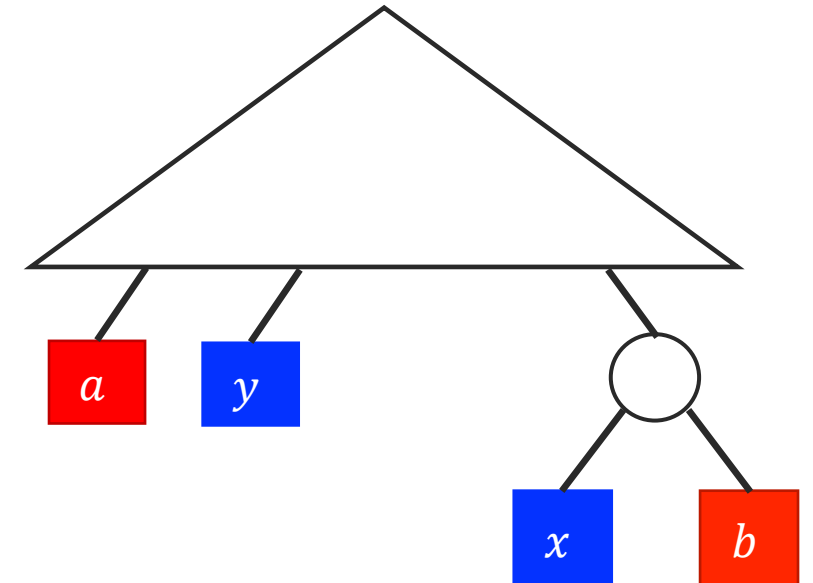
Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

- Let symbols a, b be the deepest pair of siblings.
- A lowest sibling pair exists because we have a full binary tree.
- At least one of a, b is neither x or y . Let's say $x \neq a$.

What happens if we swap x and a ?

→ **The cost of tree can't increase**, because $f_a \geq f_x$ and we just switch the length of a 's code and x 's code.

Repeat this swap and logic if $y \neq b$ either.



What do optimal subtrees look like?

Claim: There is an optimal tree where the two lowest freq. symbols are sibling leaves.

Proof: By contradiction. Let x, y be symbols with lowest frequencies and assume they aren't siblings.

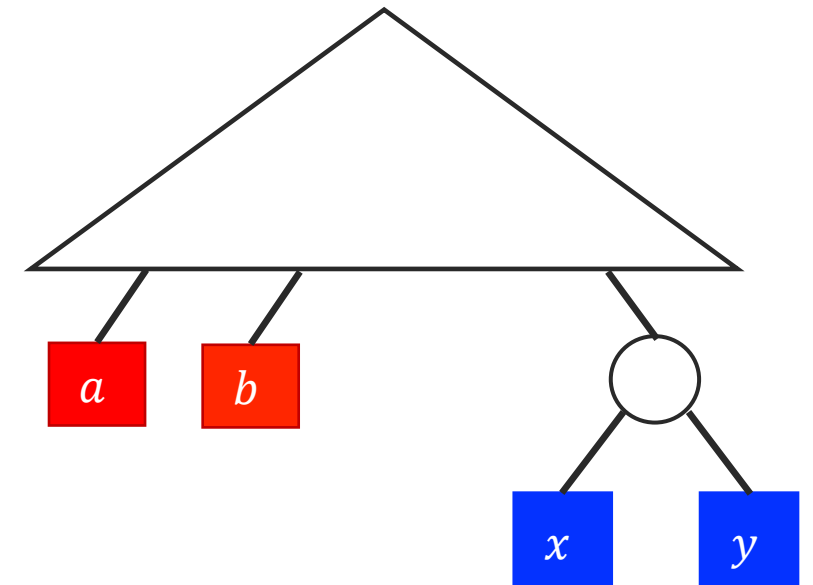
- Let symbols a, b be the deepest pair of siblings.
- A lowest sibling pair exists because we have a full binary tree.
- At least one of a, b is neither x or y . Let's say $x \neq a$.

What happens if we swap x and a ?

→ **The cost of tree can't increase**, because $f_a \geq f_x$ and we just switch the length of a 's code and x 's code.

Repeat this swap and logic if $y \neq b$ either.

We found a cheaper tree, where x, y are siblings!



What do optimal subtrees look like?

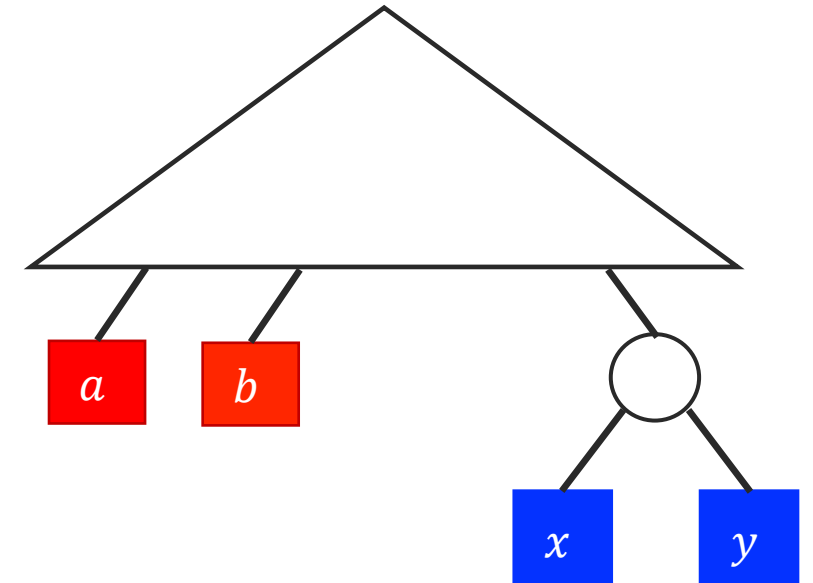
Formally: Swapping x which is at shorter depth d , with a which is at larger depth D , gives:

What happens if we swap x and a ?

→ **The cost of tree can't increase**, because $f_a \geq f_x$ and we just switch the length of a 's code and x 's code.

Repeat this swap and logic if $y \neq b$ either.

We found a cheaper tree, where x, y are siblings!



Greedy algorithm

Idea: Since the lowest frequency letters are sibling leaves in some optimal tree, we will greedily build subtrees from the lowest frequency letters.

This is called **Huffman** Coding.

Node a object with

$a.freq = f_a$

$a.left = \text{left child}$

$a.right = \text{right child}$

Huffman-code($f_1, \dots, f_n$)

For all $a = 1, \dots, n$,

create node a with $a.freq = f_a$ and no children

Insert the node in a **priority queue** Q use key f_a

While $\text{len}(Q) > 1$

x and $y \leftarrow$ the nodes in Q with **lowest keys**

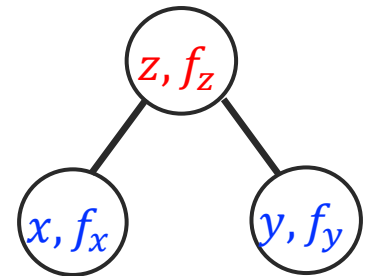
create a node z , with $z.freq = x.freq + y.freq$

Let $z.left = x$ and $z.right = y$.

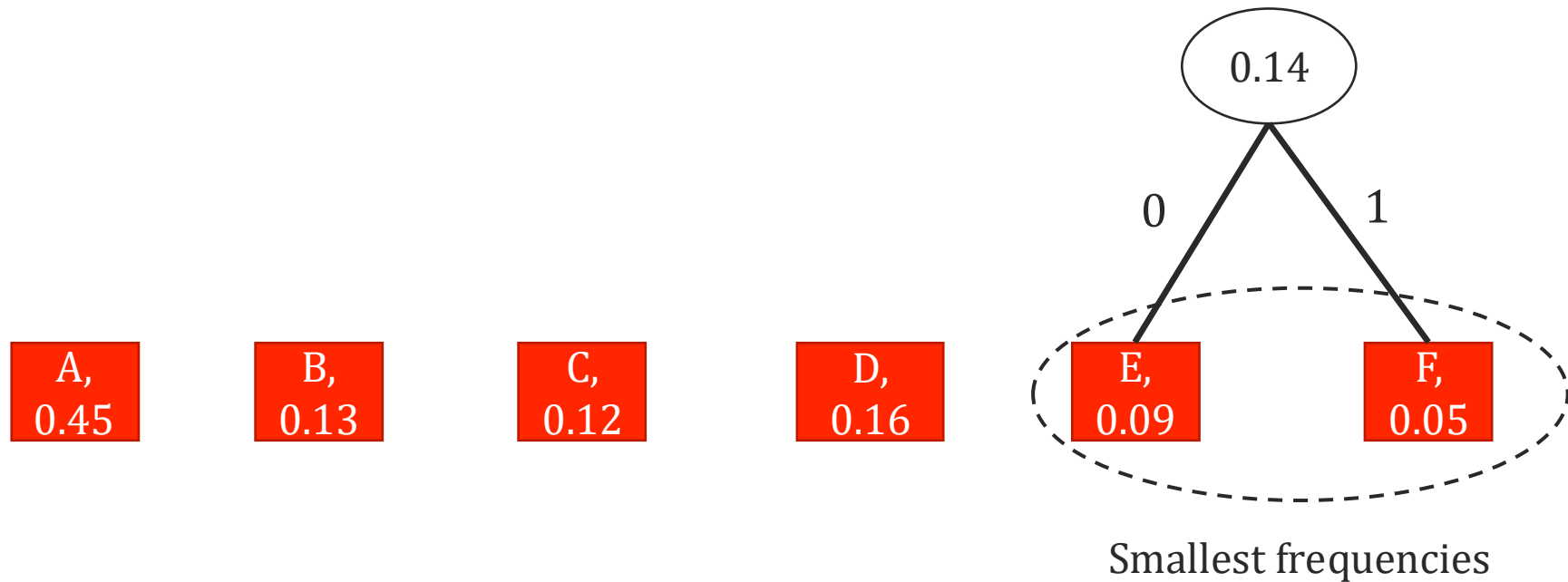
Insert z with key f_z into Q and remove x, y .

Return the only node left in Q .

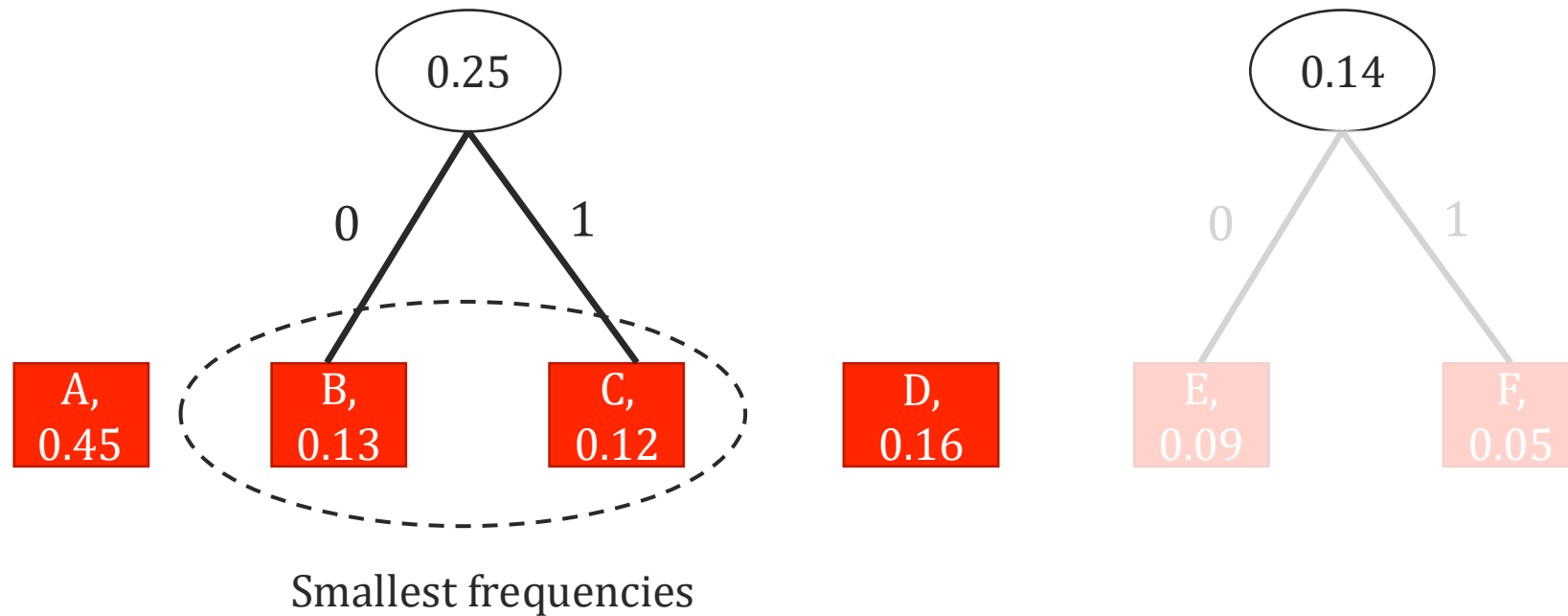
$$\textcircled{a, f_a} \equiv \boxed{a, f_a}$$



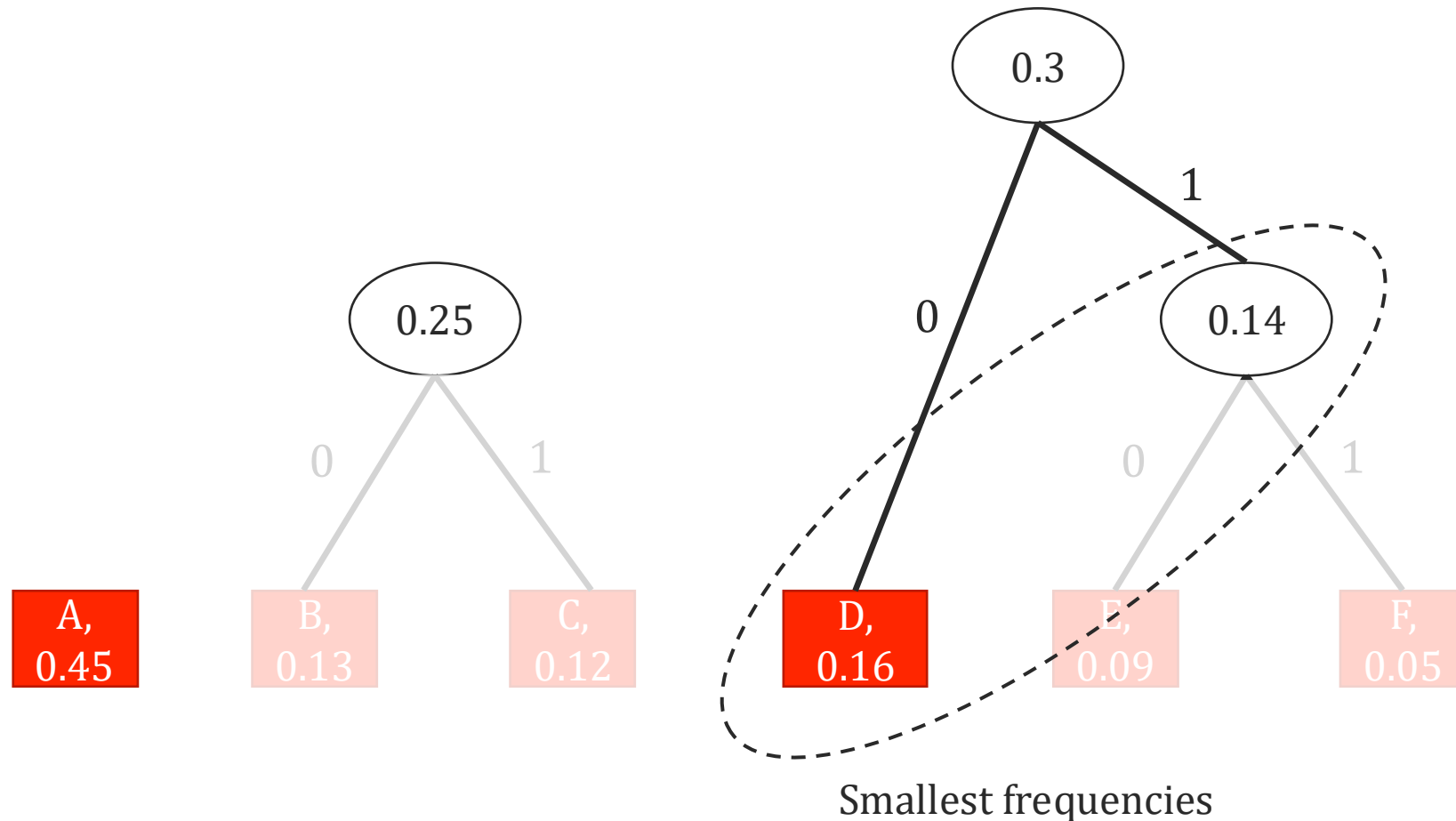
Example of Huffman Code



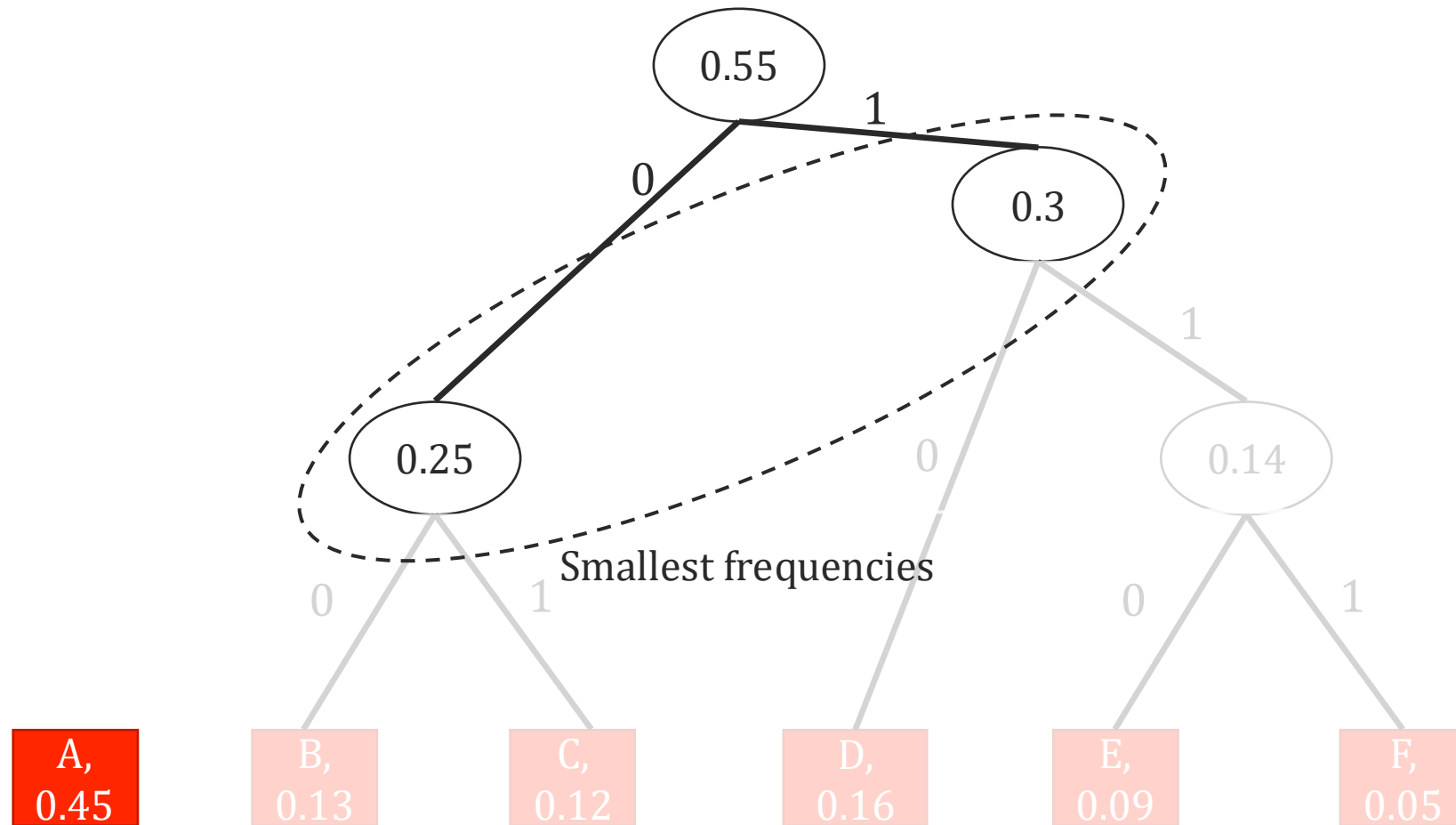
Example of Huffman Code



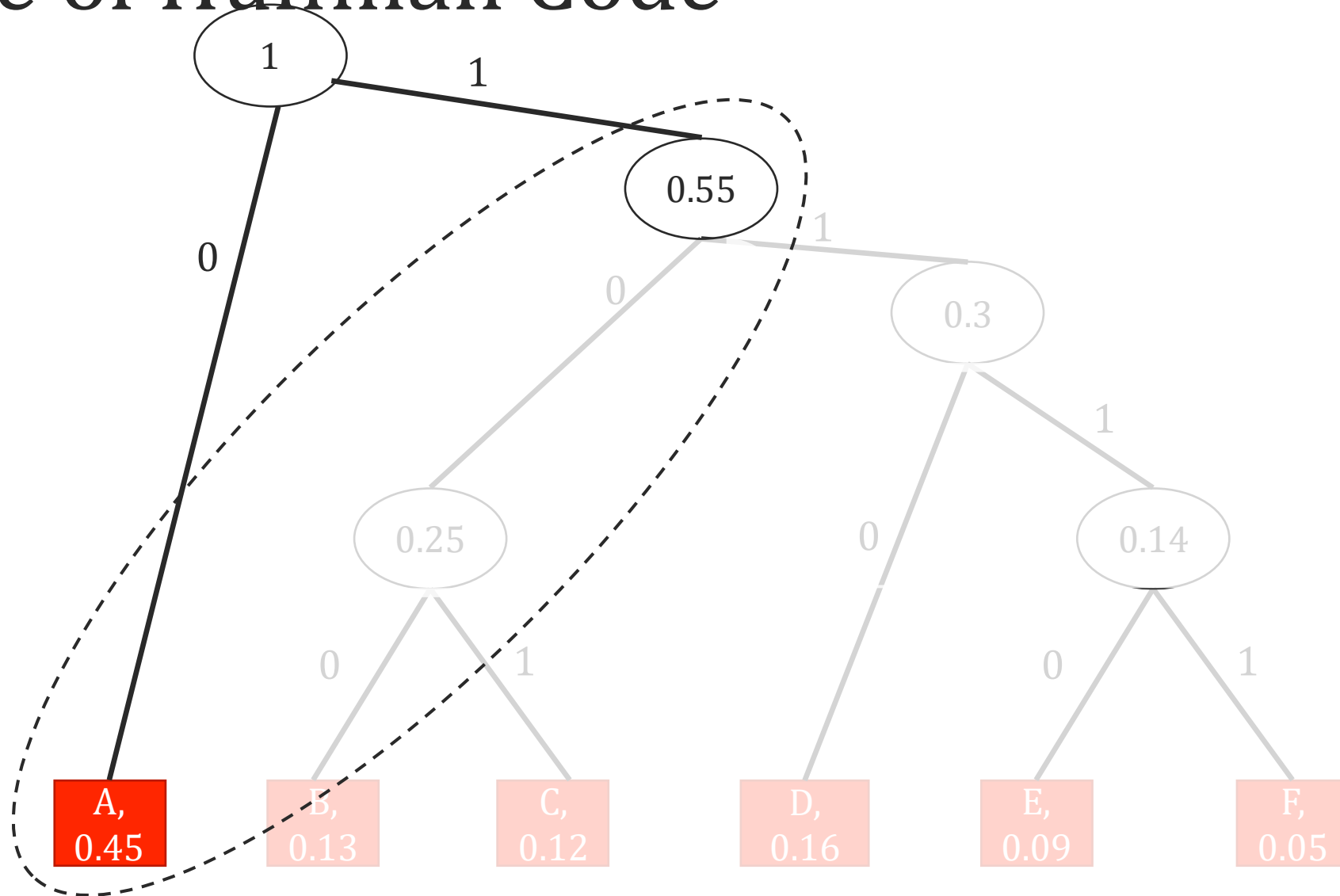
Example of Huffman Code



Example of Huffman Code

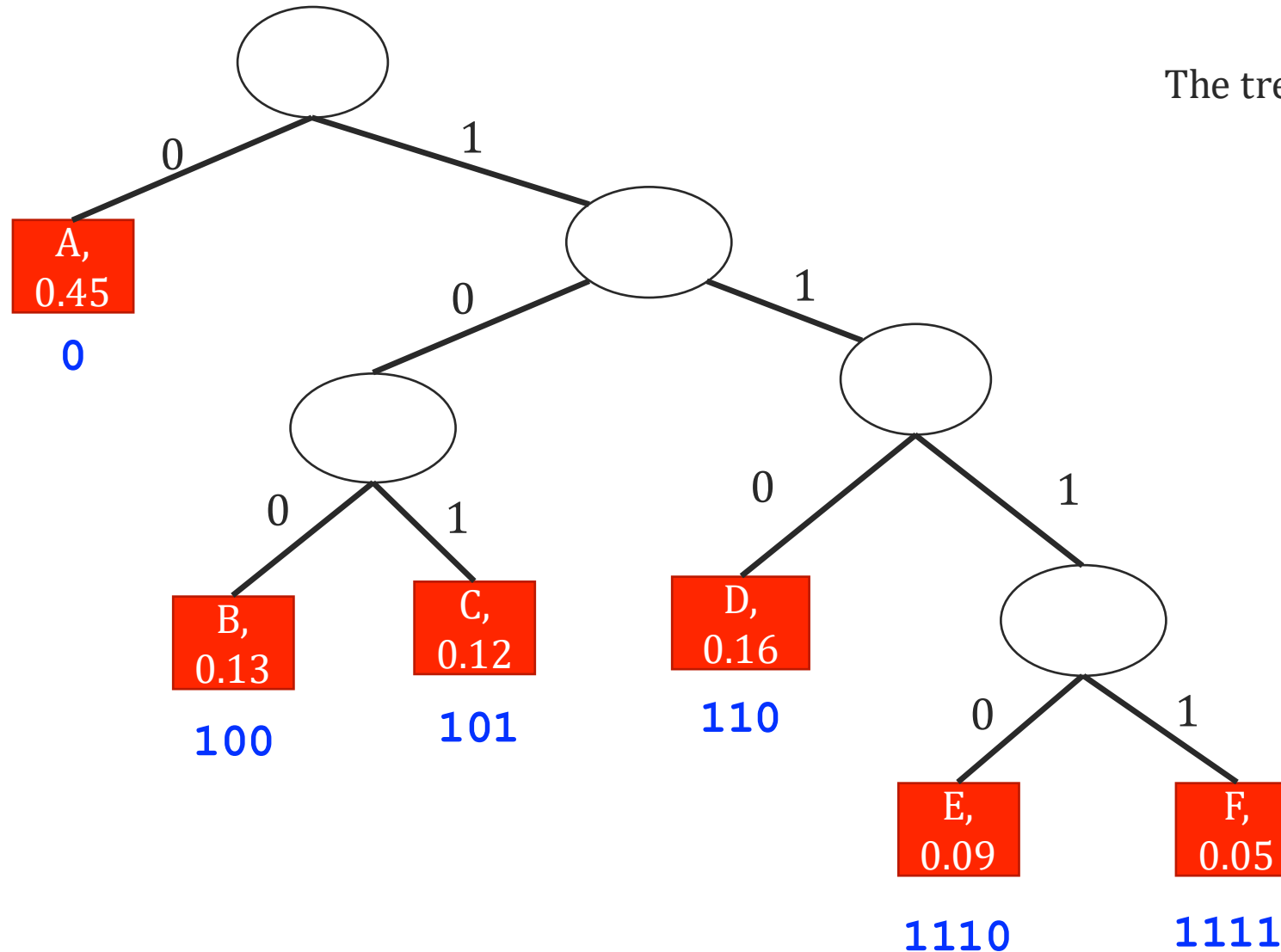


Example of Huffman Code



Smallest frequencies

The corresponding code



The tree cost:

$$\begin{aligned} & 1 \cdot 0.45 \\ & + \\ & 3 \cdot (0.13 + 0.12 + 0.16) \\ & + \\ & 4 \cdot (0.09 + 0.05) \\ & = 2.24 \end{aligned}$$

Runtime of Huffman Coding

Priority queue operation (Lec. 7): Binary heap takes $O(\log(n))$ to Insert and DeleteMin.

Huffman-code($f_1, \dots, f_n$)

n Inserts = $O(n \log(n))$ $\longrightarrow$ For all $a = 1, \dots, n$,

create node a with $a.\text{freq} = f_a$ and no children

Insert the node in a priority queue Q use key f_a

While $\text{len}(Q) > 1$

x and $y \leftarrow$ the nodes in Q with lowest keys $\longleftarrow$ 2 DeleteMin

create a node z , with $z.\text{freq} = x.\text{freq} + y.\text{freq}$

Let $z.\text{left} = x$ and $z.\text{right} = y$.

Insert z with key f_z into Q and remove x, y . $\longleftarrow$ 1 Insert

Return the only node left in Q .

n iterations, total of
 $O(n \log(n))$

Total runtime of Huffman coding: $O(n \log(n))$

Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

Recall we use induction to show that greedy choices don't rule out optimality.

We use induction on the number of letters n .

Base case: $n = 2$. The optimal code is to assign one letter to 0 and the other 1. Huffman does the same.

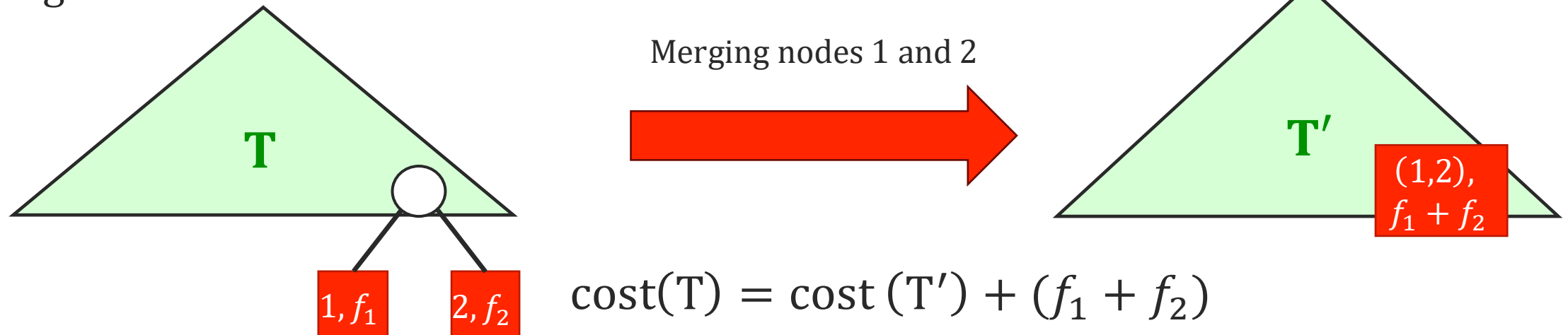
Induction Hypothesis: For $n - 1$ letters, Huffman coding is an optimal pre-fix tree.

Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

Induction step: Let T below be the optimal prefix-free tree for frequencies $f_1, \dots, f_n$ and WLOG $f_1 \leq f_2 \leq \dots \leq f_n$.

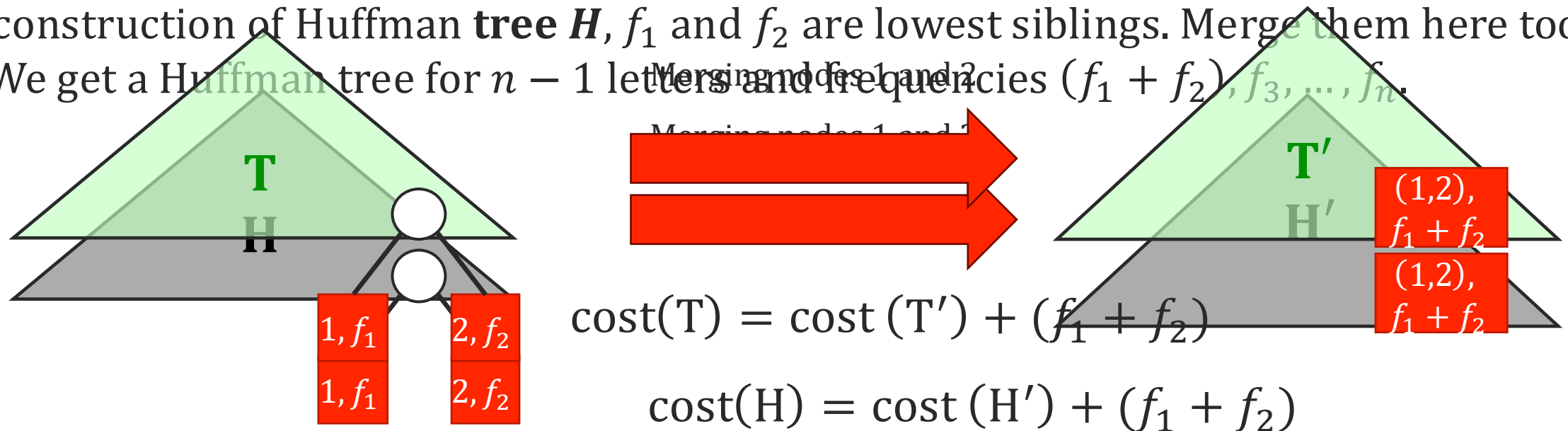
- WLOG, assume that the two lowest frequency nodes are siblings.
→ Because, we proved earlier that this claim is true!
- Merge the two nodes



Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

By construction of Huffman **tree** H , f_1 and f_2 are lowest siblings. Merge them here too.
 → We get a Huffman tree for $n - 1$ letters and frequencies $(f_1 + f_2), f_3, \dots, f_n$.



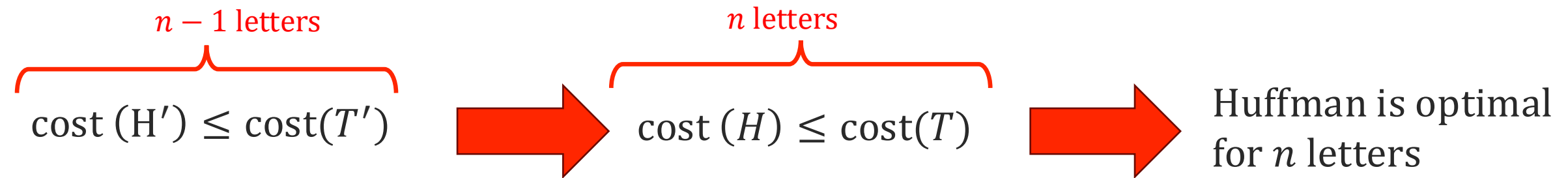
Optimality of Huffman Coding

Claim: Huffman coding is an optimal prefix-free tree.

We showed that for **tree T** that is **optimal for n** letters, $\text{Cost}(T) = \text{cost}(T') + (f_1 + f_2)$.

And for Huffman coding tree H for n letters, $\text{Cost}(H) = \text{cost}(H') + (f_1 + f_2)$.

Putting everything together.



By induction hypothesis,
Huffman coding for $n - 1$
letters is optimal

Wrap up

Greedy Algorithms are simple to design

A bit harder to analyze perhaps!

- Induction is our friend.
- Find a nice substructure of the problem

Next time

- Satisfiability
- Quiz 1