CS 170
Efficient Algorithms and Intractable Problems

Lecture 9
Greedy Algorithms

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Midterm 1 is in less than two weeks!

→ Scope: Everything up to and including 09/26 lecture (Min spanning trees)

→ There will be some midterm review sessions (look for announcements next week)

HW4 released late, yesterday

→ So it's due next Wed evening.

→ But, there aren't additional office hours HW parties on Tuesday/Wed

→ Attend OH and HW parties early!

Announcements

Discussion sheets:

→ We will release the solutions with the discussion sheet

→ But …

  → They are password protected! Get the password from your TA when you attend a discussion!

  → This is to encourage you to go to sessions but also make things available earlier for you!

→ Password is announced for those who can’t go to the session on Thursday evenings!

Last two lectures

Lots of graph algorithms

- BFS, DFS
- Applications of BFS, DFS

Today and Next Lecture: Greedy Algorithms

Algorithms that build up a solution

piece by piece, always choosing the next piece

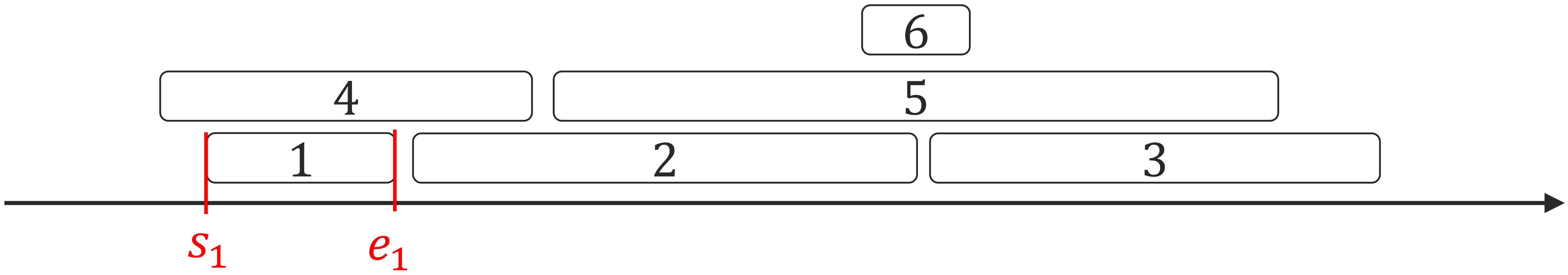
that offers the most obvious and immediate benefit!

Examples of problems where greedy works
- Scheduling
- Satisfiability
- Huffman Coding (next lecture probably)
- Minimum spanning trees (next lecture)

(Interval) Scheduling

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

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



Application example:

- intervals denote activities you are interested in.

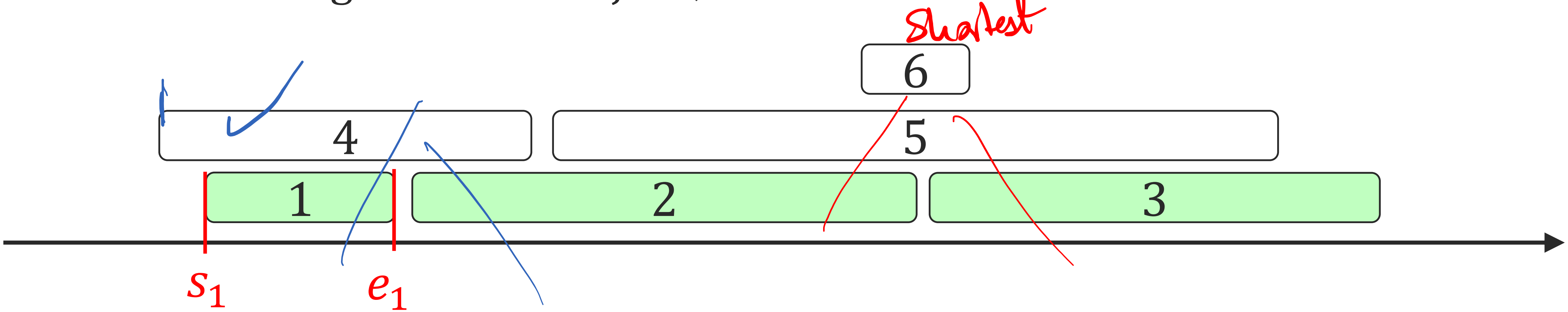
→ classes take, times you can hangout with friends, time for rest, appointments, ...

- You want to do as many activities as possible!

(Interval) Scheduling

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

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

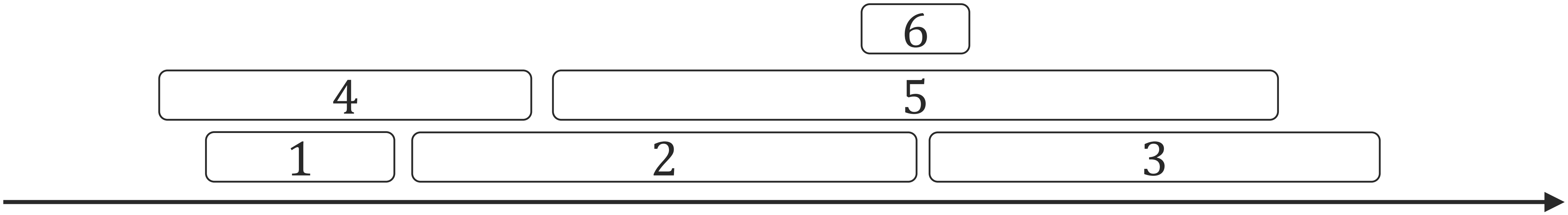


Discuss

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!



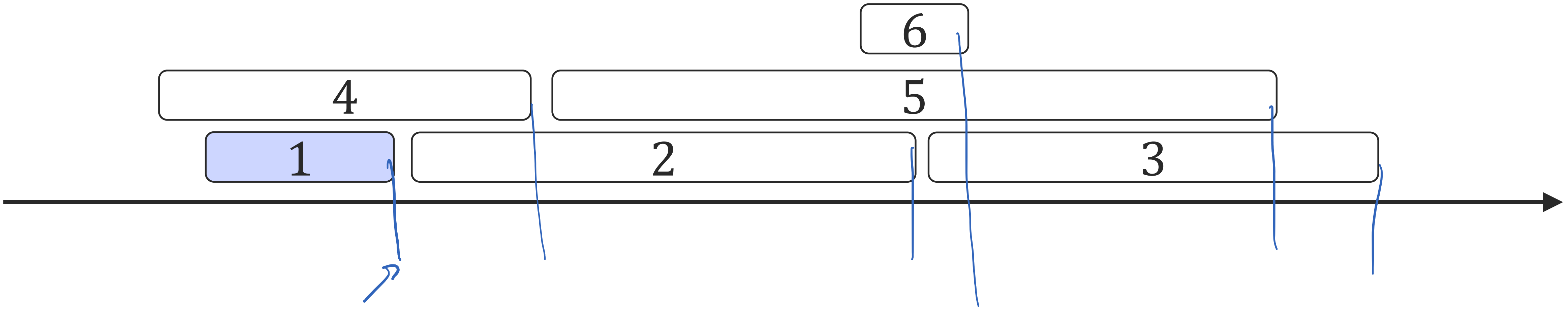
Algorithm:

While the set of intervals is non-empty

Add interval j with the earliest finish time e_i.

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!



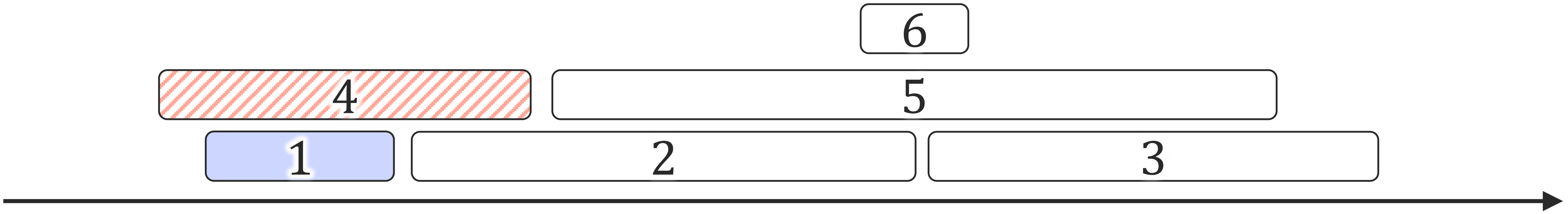
Algorithm:

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!



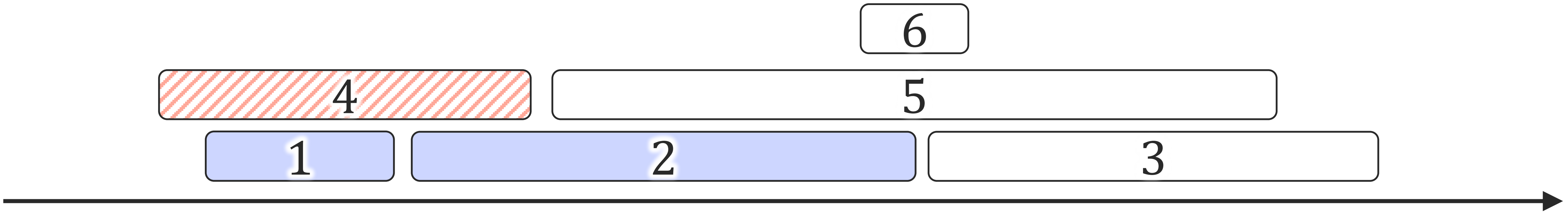
Algorithm:

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!



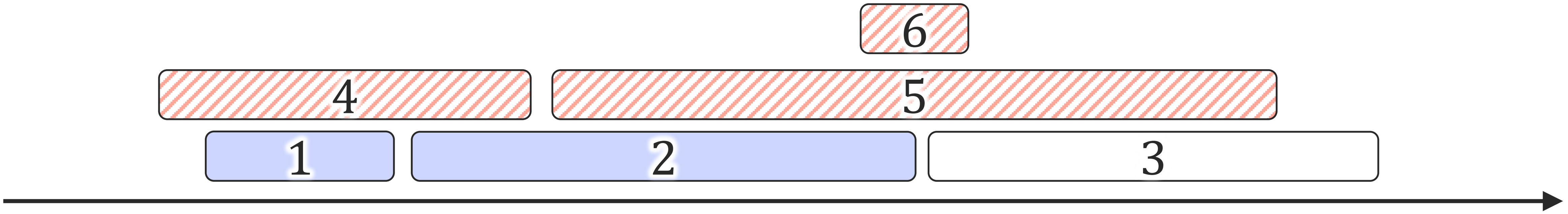
Algorithm:

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!



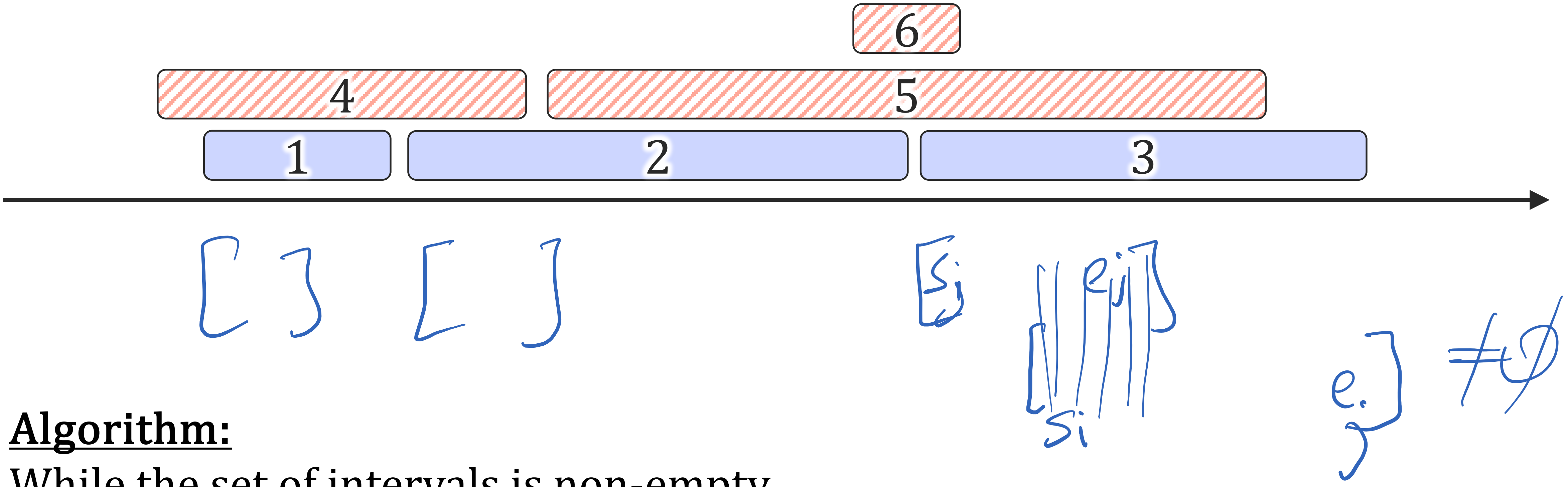
Algorithm:

While the set of intervals is non-empty

Add interval j with the earliest finish time e_i.

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!



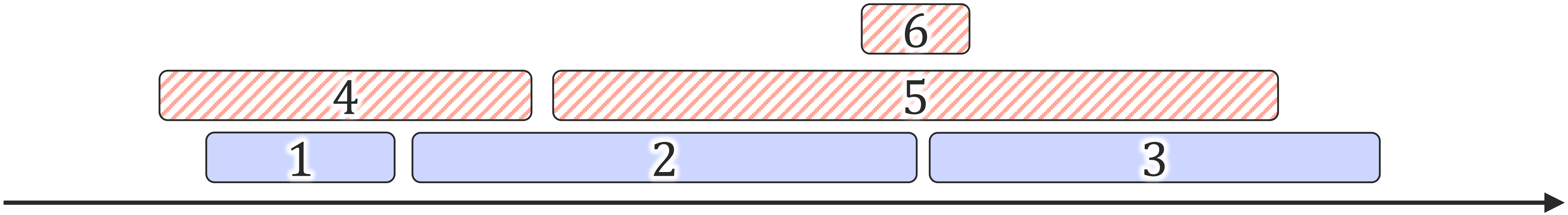
Algorithm:

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.

→ Otherwise $O(n\log(n))$ if we have to sort them by the finish time.

Why does greedy work for interval scheduling?

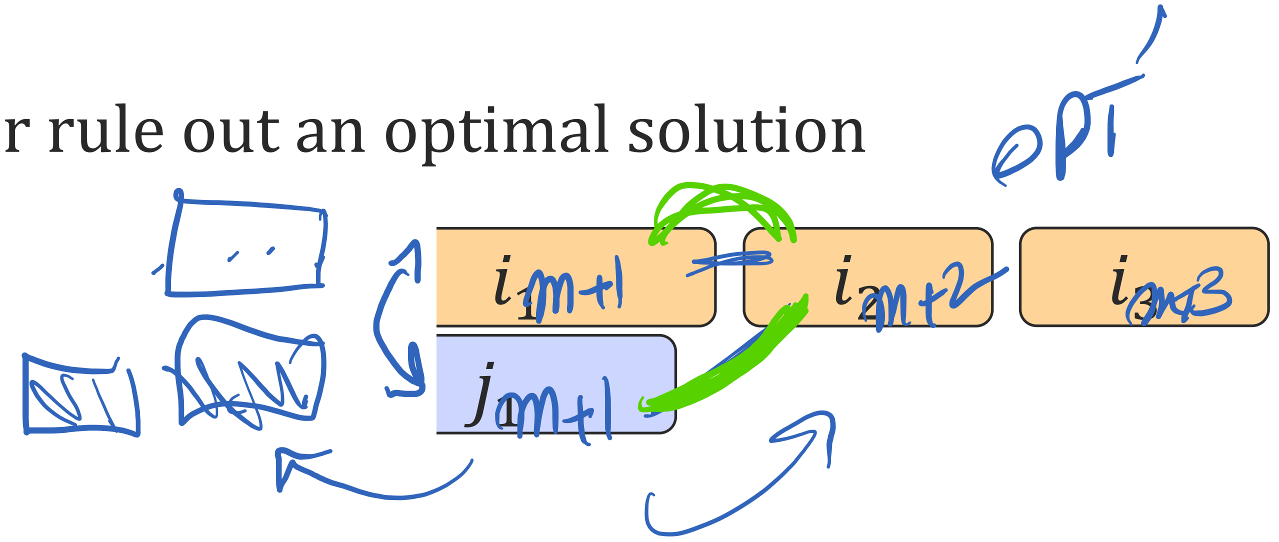
Whenever we make a choice to include an interval in the solution, **we don't rule out an optimal solution.**

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

Intuition for why we never rule out an optimal solution

$$\text{Greedy} = \boxed{j_1}, j_2, j_3, \ldots, j_k$$

$$\text{OPT} = \boxed{i_1}, i_2, i_3, \ldots, i_k, i_{k+1}$$



Swap in j_1 for i_1.

Why does greedy work for interval scheduling?

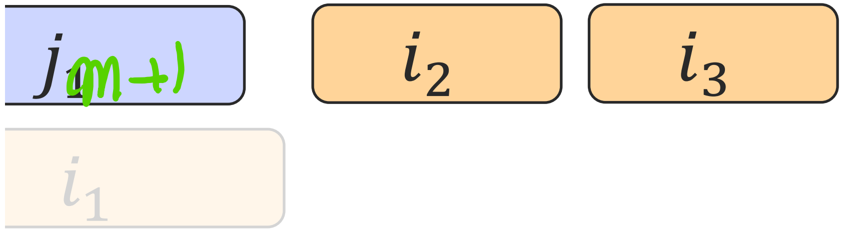
Whenever we make a choice to include an interval in the solution, **we don't rule out an optimal solution.**

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

Intuition for why we never rule out an optimal solution

Greedy $= j_1, j_2, j_3, \ldots, j_k$

OPT $= i_1, i_2, i_3, \ldots, i_k, i_{k+1}$



Swap in j_1 for i_1.

More formal argument: Proof by induction

$e_{j_{m+1}} < e_{i_{m+1}}$

$i_{m+1} = j_{m+1}$

(2) OPT' is the same size as OPT $\Rightarrow$ (1)+(2) OPT' is a valid schedule.

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

OPT = $i_1 \dots \dots i_{k+1}$

end: $e_{i_1} < s_{i_2} < e_{i_2} < s_{i_3}$

$G = j_1 \dots \dots j_k$

Induction on m:

Base Case $\Rightarrow m = 0$ ✓ Any two schedules agree on 0 things.

Induction hypothesis: claim is true for $m \Rightarrow i_1 = j_1 \dots i_m = j_m$.

Inductive step:

Case 1: $i_{m+1} = j_{m+1}$ ✓

Case 2: $i_{m+1} \neq j_{m+1}$: OPT' = OPT except replace i_{m+1} with j_{m+1}

$j_1 \dots j_m j_{m+1} i_{m+2} \dots$

(1) OPT' is not conflicted : 1) j_{m+1} doesn't conflict $j_1 \dots j_m$ (b/c greedy solution)

2) j_{m+1} doesn't conflict with $i_{m+2} \dots$ (b/c j_{m+1} ends earlier than i_{m+1})

* If $j_{m+1} \cap i_{m+2} \neq \emptyset \Rightarrow i_{m+1} \cap i_{m+2} \neq \emptyset$ ✗ b/c OPT is valid schedule.

A Pattern in 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 no worst than OPT.

Conclusion: The greedy algorithm outputs an optimal solution.

3 Min Break

Please close the auditorium’s door

Horn Formula

$\Rightarrow x$

Variables: $x_1, \ldots, x_n \in \{True, False\}$, a literal is x_i or $\overline{x_i}$.

Clauses:

1. "Implication clause" (with no negatived variable)

if False only $\Leftarrow$ $\underbrace{(x_i \wedge x_j \wedge \cdots)}_{\text{True}} \Rightarrow \underbrace{x_k}_{\text{False.}}$ $\longleftrightarrow$ Equivalent to $\overline{x_i} \vee \overline{x_j} \vee \cdots \vee x_k$

2. "Pure negative clauses"

$$(\overline{x_i} \vee \overline{x_j} \vee \cdots)$$

Horn Formula: AND of m Horn clauses

Horn Clause’s Significance

Why care about these clauses?

→Used in computational logic, theorem proving, etc.

→Prolog is based on Horn clauses.

Horn-SAT

$(x_1 \wedge \dots x_k) \Rightarrow x_{k+1}$

Implication

sparse negative.

Input:

A Horn formula (AND of Horn clauses)

Output:

Find an assignment for the variables that makes the Horn formula True, if such and assignment exists.

Greedy Algorithm for Horn-SAT

Horn-Formula Clauses:

$(w \wedge y \wedge z) \Rightarrow \overline{x}$ ✓

$(x \wedge z) \Rightarrow w$ ✓

$\overset{T}{x} \Rightarrow \overset{F}{y}$ ✓

X $\Rightarrow x$ ✓

$(\overset{T}{x} \wedge \overset{T}{y}) \Rightarrow w$ ✓

$(\overline{w} \vee \overline{x} \vee \overline{y})$ $\Rightarrow$ F

F F F

Variable assignments:

x $\cancel{F}$ T

y $\cancel{F}$ T

z F

w $\cancel{F}$ T

True.

For all i, set $x_i = False$

While there exists $\left(\left(x_i \wedge \cdots \wedge x_j\right) \Rightarrow x_k\right) = False$

Set $x_k = True$

If every pure negative clause $\left(\overline{x_i} \vee \cdots \vee \overline{x_j}\right) = True$

Return $(x_1, \ldots, x_n)$

T F

Else

Return "not satisfiable"

Why does Greedy Work for Horn-SAT?

What's the pattern in this case?

We want to establish that when Greedy sets a variable $x_i = True$, it does not ruin a satisfying assignment.

In fact, we will prove

→ The set of variables set to True by the Greedy algorithm, are also set to True in any satisfying assignment.

Proof of Claim

Claim: The variables set to True by Greedy, are also True in the satisfying solution.

Proof: By induction on the iteration of the While loop

<u>Base case</u>: In the 0th iteration of the While loop, nothing is set to True.

<u>Induction hypothesis</u>: The first m variables set to True by Greedy are also True in every satisfying solution.

<u>Inductive step:</u>

- Let x_{m+1} be the $m+1^{st}$ variable set to True by Greedy. True F
- This means there was an unsatisfied implication $(x_i \wedge \cdots \wedge x_j) \Rightarrow x_{m+1}$ before the $m+1$ iteration of the while loop.

→ This only happens if $(x_i \wedge \cdots \wedge x_j) = True$ before the m+1 iteration of Greedy, meaning that $(x_i \wedge \cdots \wedge x_j) = True$ also in the satisfying solution.

- The only way to satisfy this clause in SAT is to also have $x_{m+1} = True$.

Horn-SAT Proof completed

Claim: The greedy solution is correct.

1) If Greedy outputs a solution, then the solution is satisfiable.

This is true because the While loop and If condition check that all clauses are satisfied

2) If the Horn Formula is satisfiable, then Greedy outputs a satisfiable solution.

Assume to the contrary that this is not true. So, Greedy outputs “unsatisfiable” even though a satisfying assignment exists.

$x_i = T$

→In Greedy’s assignment, there is a violated pure clause $\boxed{(\bar{x}_i \vee \cdots \vee \bar{x}_j)} = F \Rightarrow$

→So, every variable in this clause is set to True

→By previous slide, these variables are also set to True in any satisfiable solution and this clause is also violated by the satisfying assignment

→Contradiction!

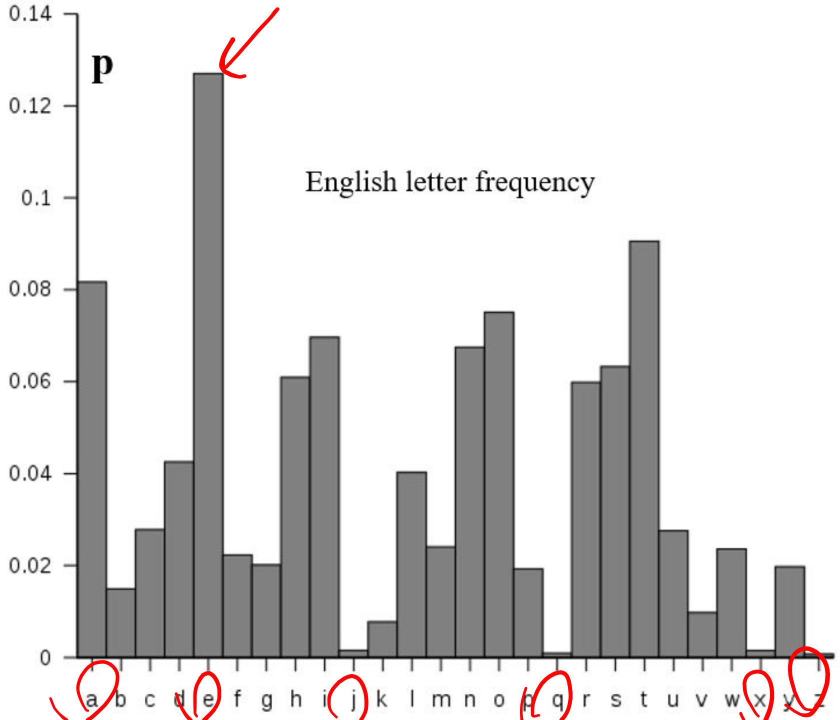
Next up: Codes!

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

text of length N

$\overbrace{00}\,01011$

ABCD

000101

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

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

Code 2 is lossy: 000 $\rightarrow$ AB, BA ?

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	A	00	0	0
0.2	B	01	00	110
0.3	C	10	1	10
0.1	D	11	01	111
Total cost		$2N$	$(0.4 + 0.3)\times N + (0.1 + 0.2) \times 2N = 1.3N$	$0.4\times N + 0.3\times 2N + (0.2 + 0.1)\times 3N = 1.9N$

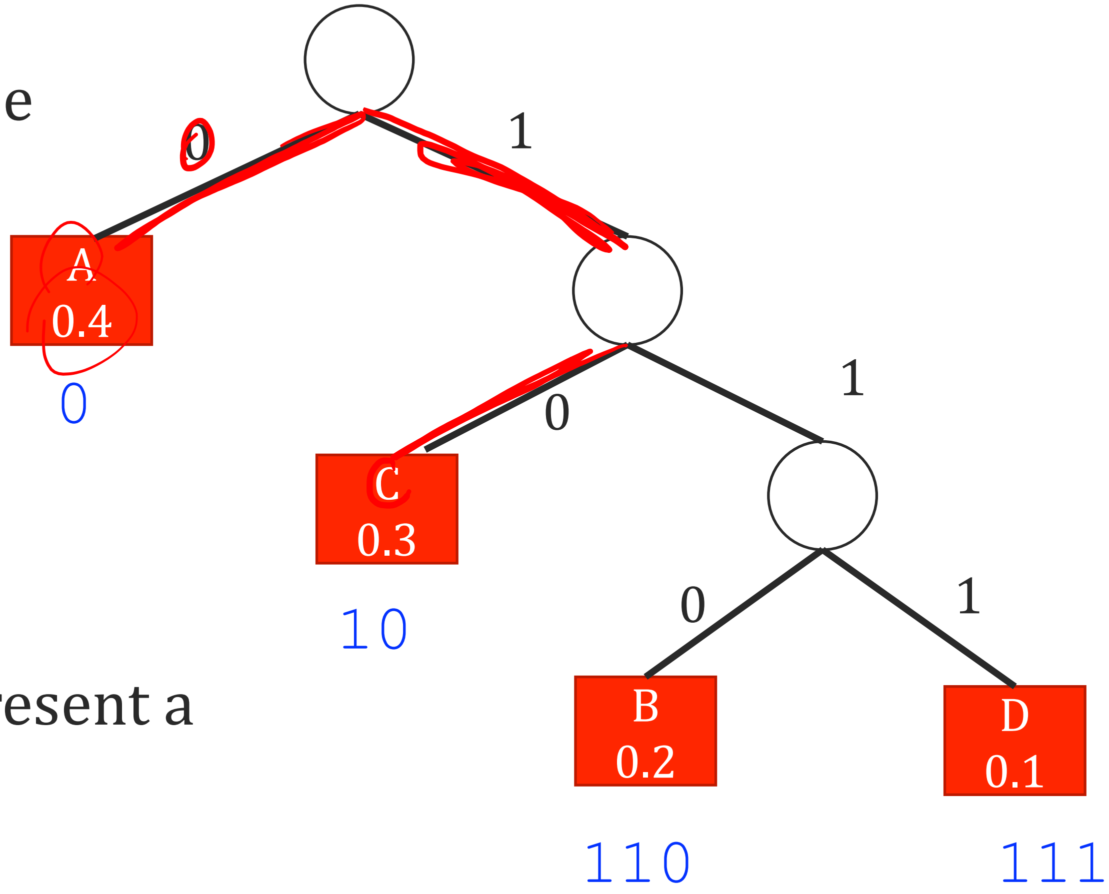
But encoding #2 is lossy: What does 000 represent? AB or BA?
Encoding #3: No code is a prefix of another.
→ There is only one way to interpret any code.

Any Prefix codes and Trees

means "A" has freq. 0.4.

Any prefix-free code can be represented as a binary tree with k 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.

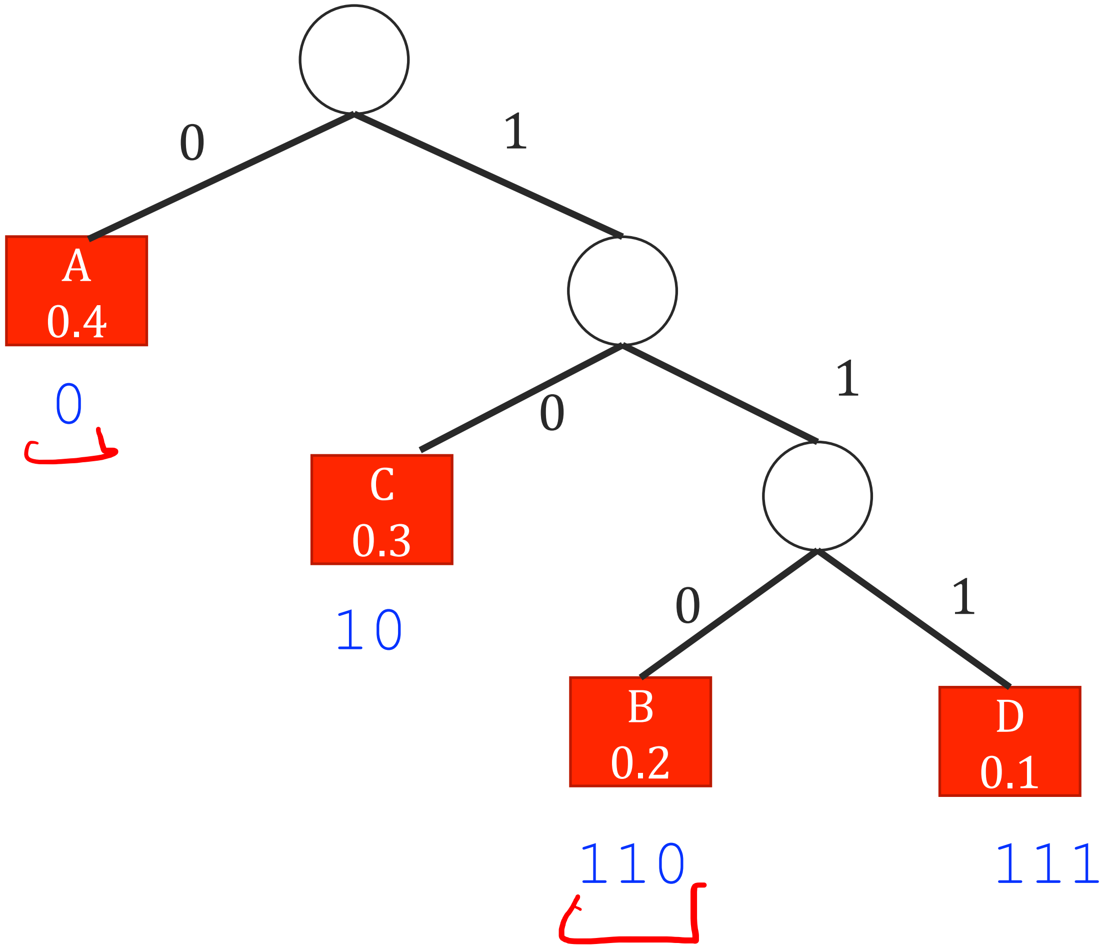
Tree and Code Size

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?

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



Definition: Cost of a prefix-code/tree is

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

Optimal Prefix-free Codes

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

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

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

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