Lec17: Greedy Algorithms I

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee
School of Computer Science and Engineering
Kyungpook National University **(KNU)**

Last time

- Dynamic programming
  - An algorithm design paradigm.
  - Basic idea:
    - Identify optimal sub-structure
    - Take advantage of overlapping sub-problems
    - Keep track of the solutions to sub-problems in a table as you build to the final solution.

- And examples!
  - LCS
  - 0/1 knapsack

Course Overview

- Algorithmic Analysis
- Divide and Conquer
- Randomized Algorithms
- Tree Algorithms
- Graph Algorithms
- Dynamic Programming
- Greedy Algorithms
- NP Completeness

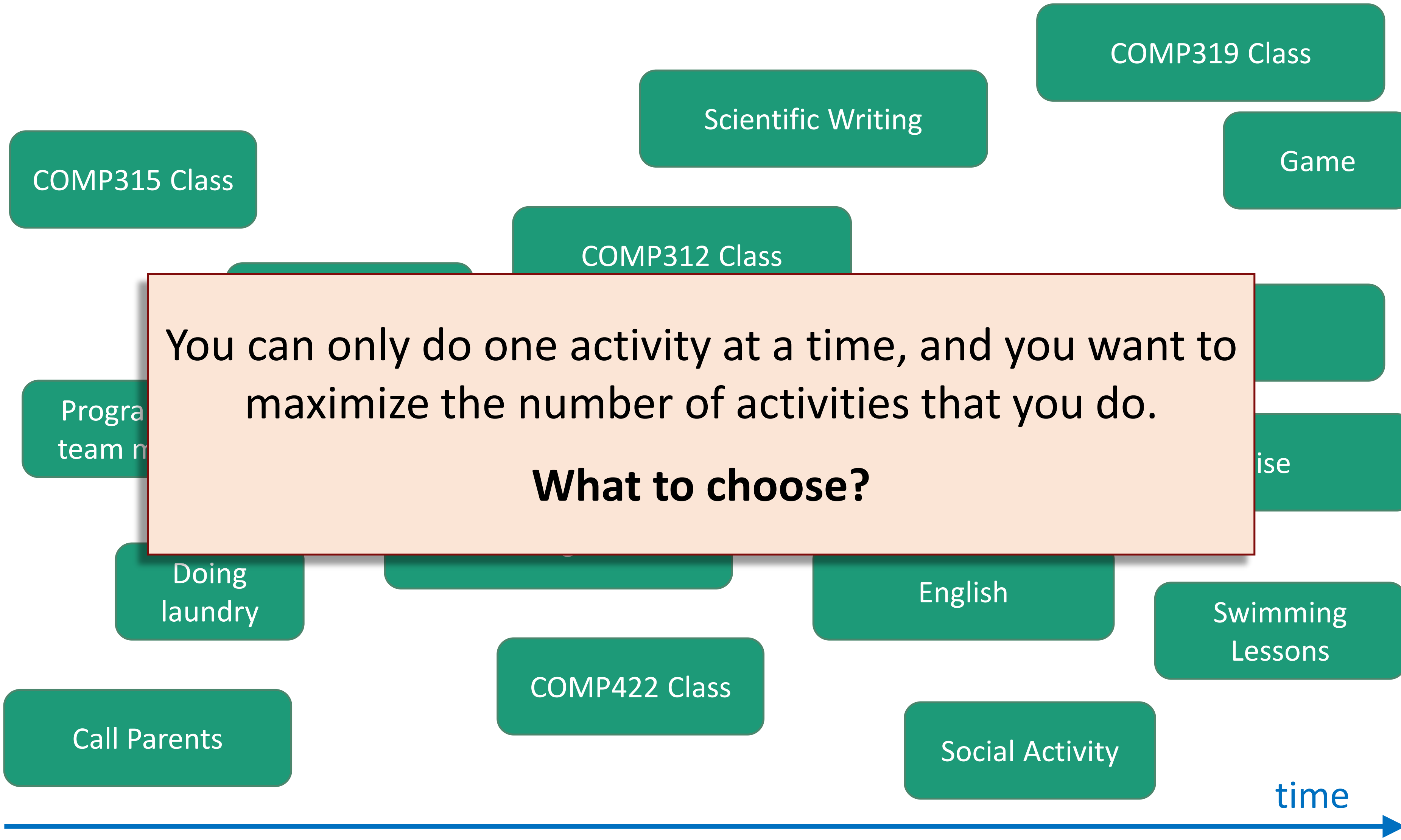
Today

- **Greedy algorithms**
  - Make choices one-at-a-time.
  - Never look back.
  - Hope for the best.

  - Advantages: simple to design, often efficient
  - Disadvantages: difficult to verify correctness or optimality

- **Examples** of it works well
  - Activity Selection
  - Huffman Coding

Example

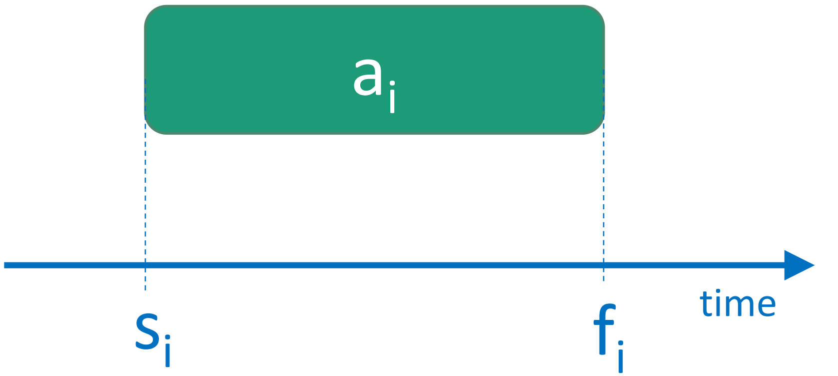


You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

Activity Selection

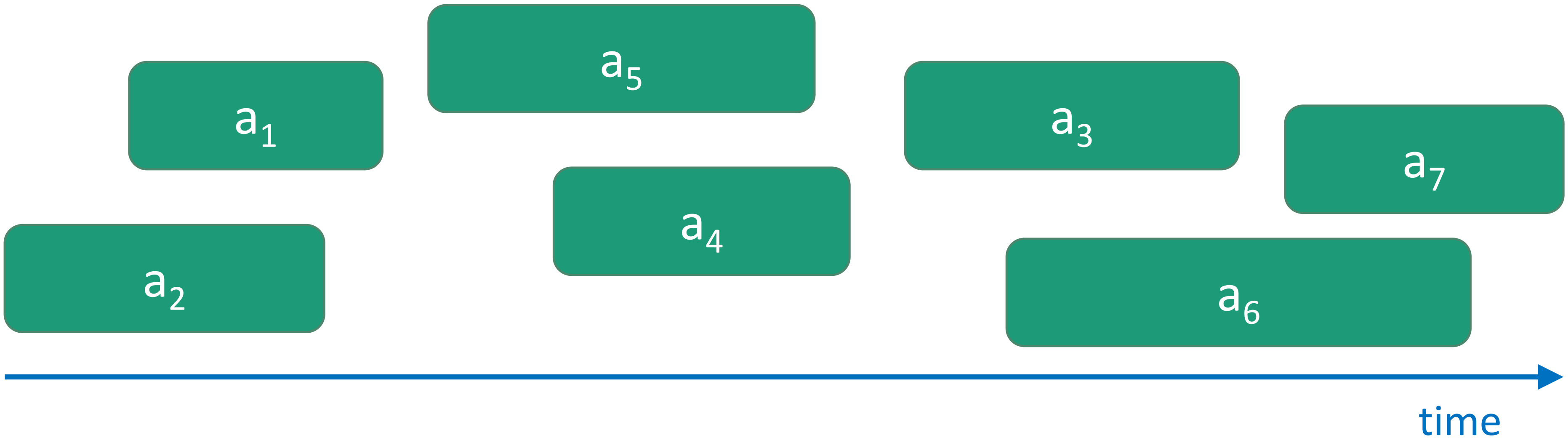
- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$



- Output:
  - A way to maximize the number of activities you can do today.

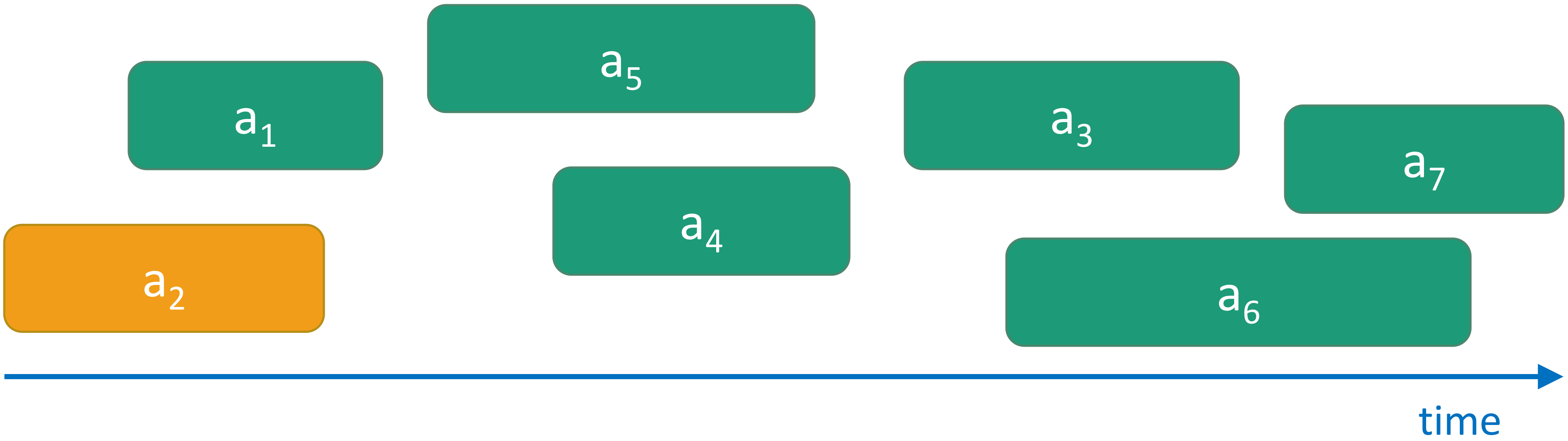
- In what order should you greedily add activities? There are many options:
  - Shortest job first?
  - Fewest conflicts first?
  - Earliest ending time first? ← We will use this!

Greedy Algorithm



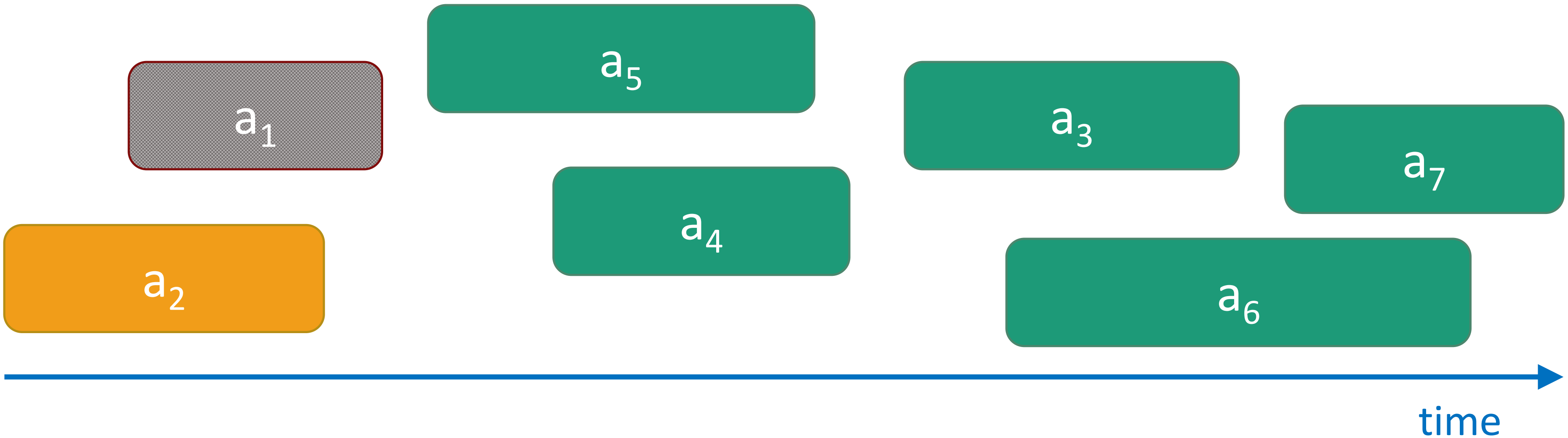
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



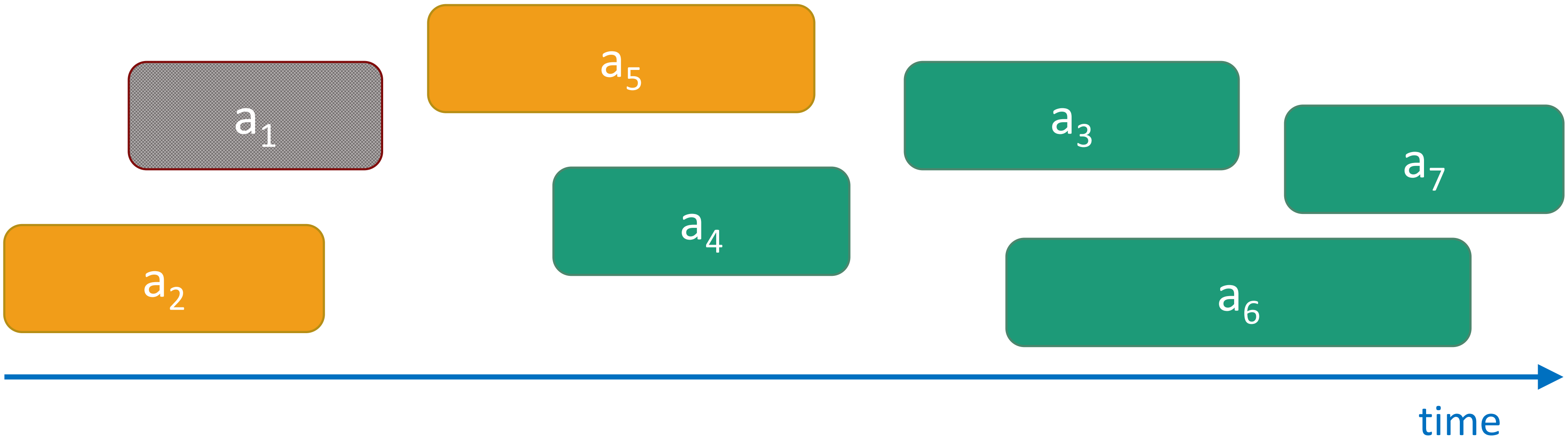
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



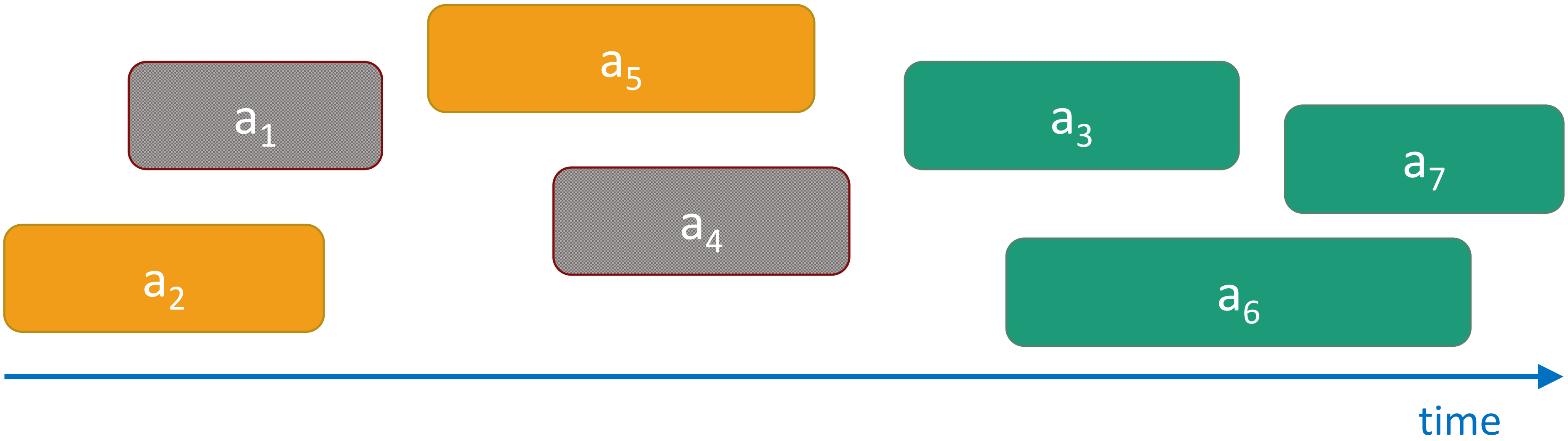
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



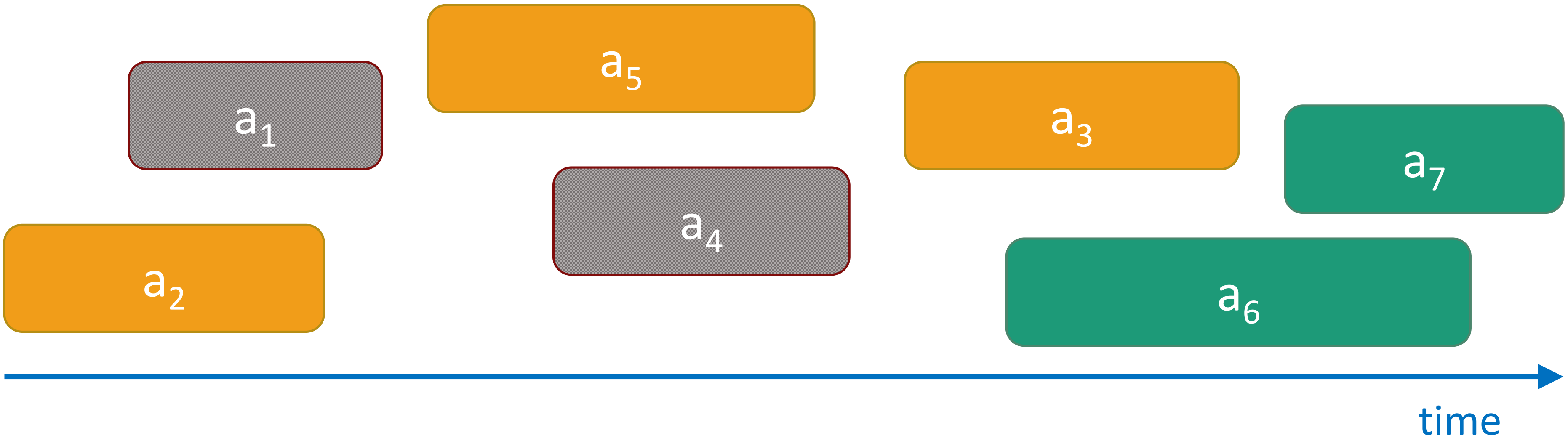
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



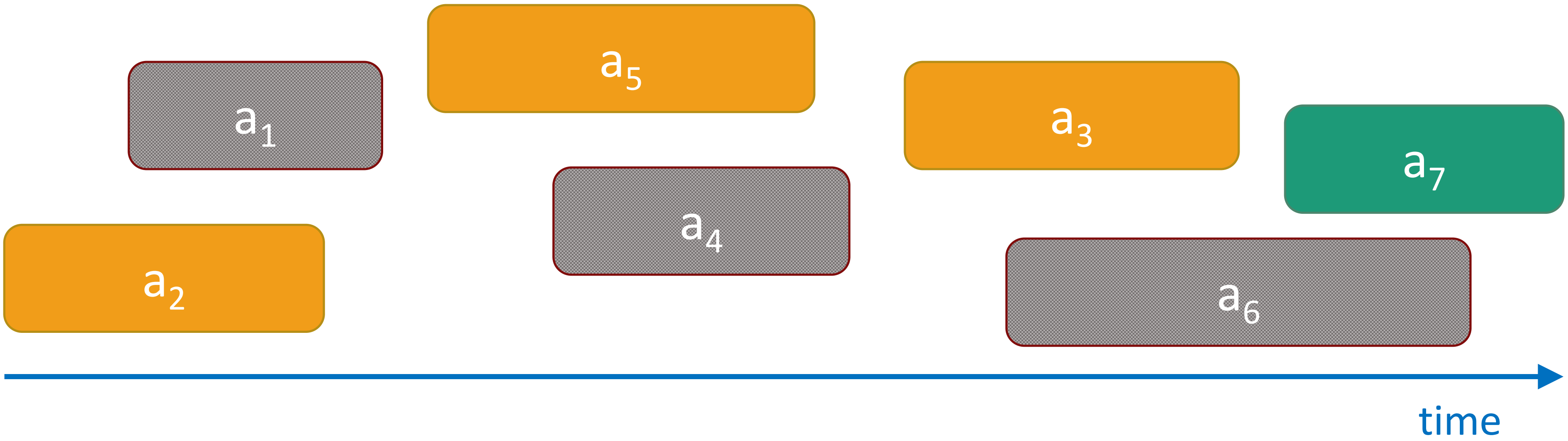
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



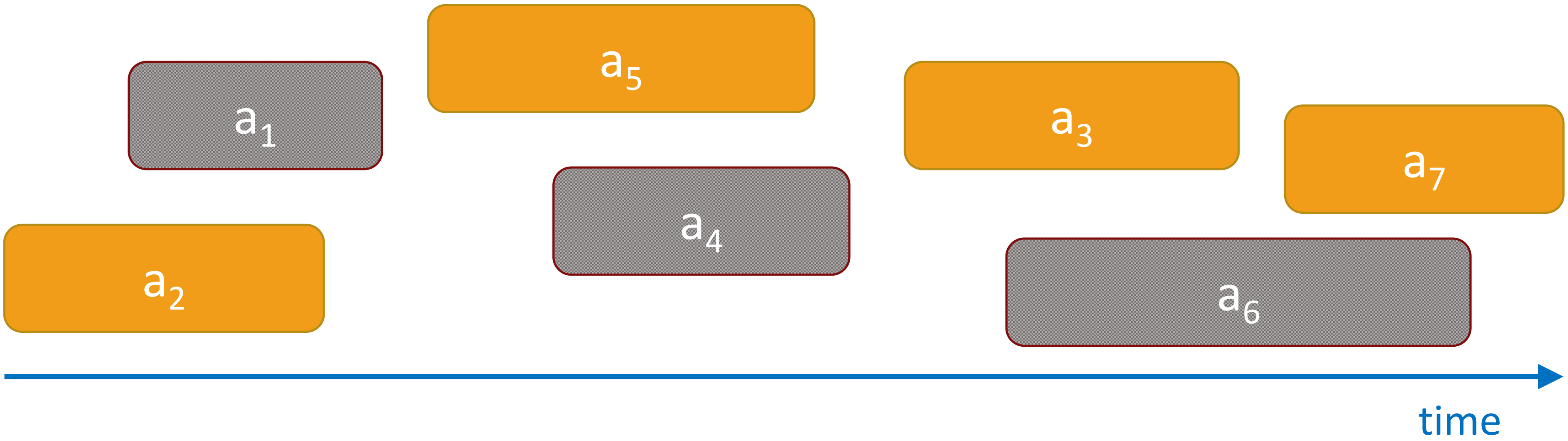
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

At least it's fast

- Running time:
  - $O(n)$ if the activities are already sorted by finish time.
  - Otherwise, $O(n\log(n))$ if you have to sort them first.

What makes it greedy?

- At each step in the algorithm, make a choice.
  - Hey, I can increase my activity set by one,
  - And leave lots of room for future choices,
  - Let’s do that and hope for the best!!!

- **Hope** that at the end of the day, this results in a globally optimal solution.

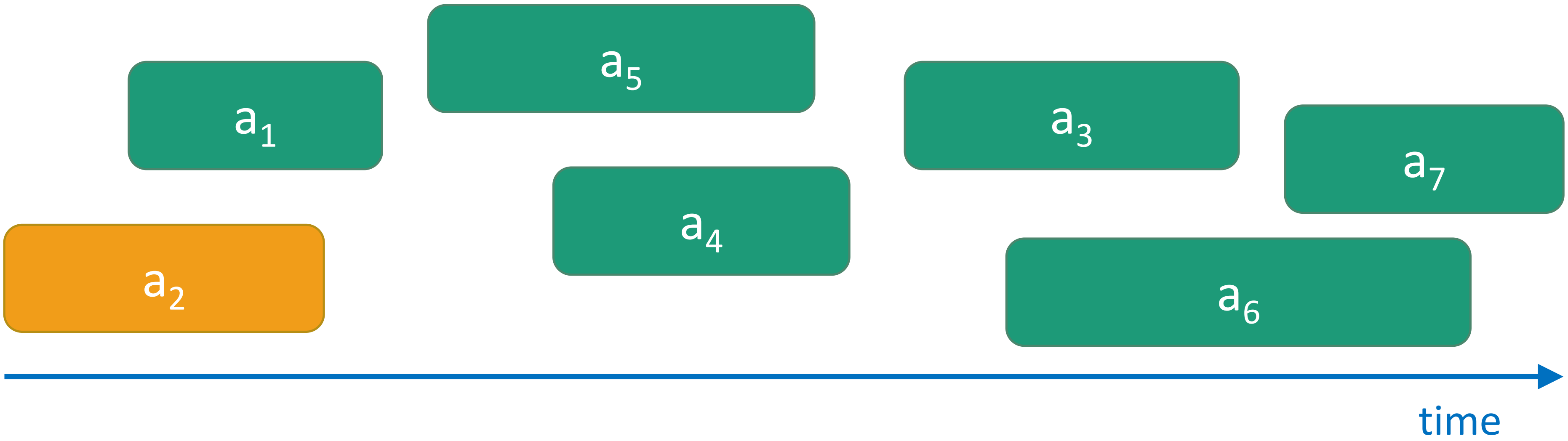
Three Questions

1. Does this greedy algorithm for activity selection work?

2. In general, when are greedy algorithms a good idea?

3. The **"greedy"** approach is often the first you'd think of… Why are we getting to it now, in Week 12?

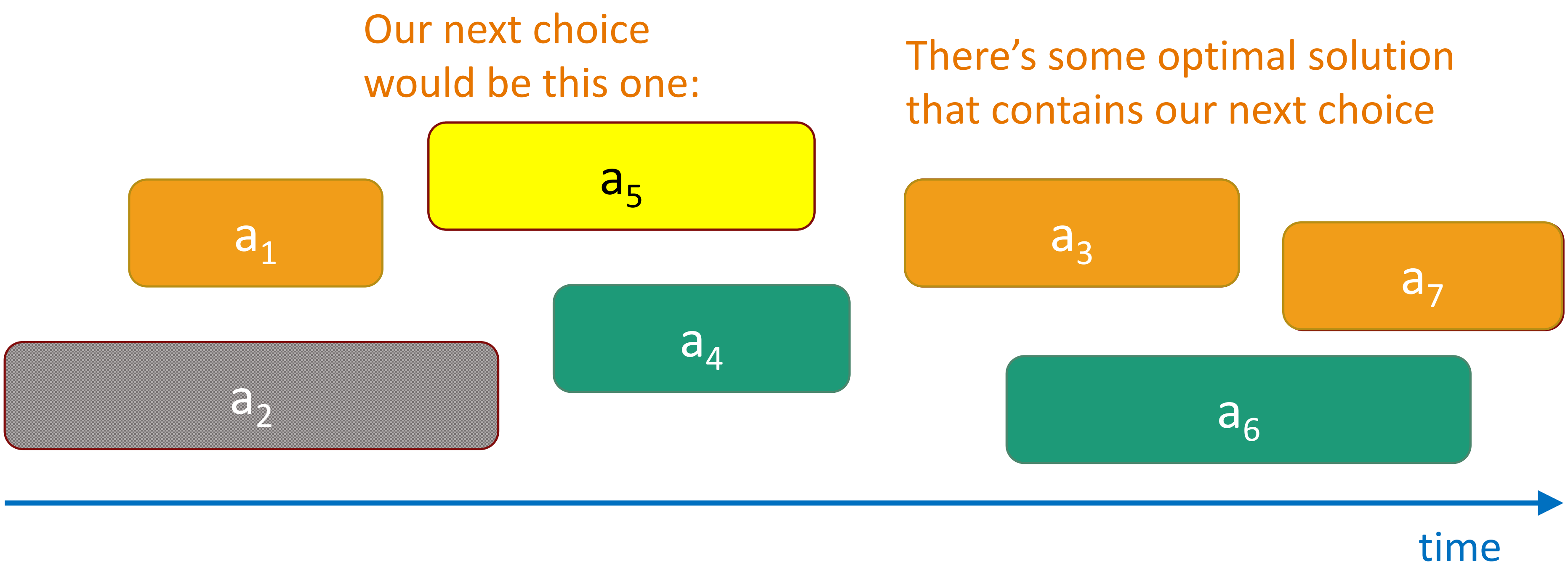
Back to Activity Selection



- Pick activity you can add with the smallest finish time.
- Repeat.

Why does it work?

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

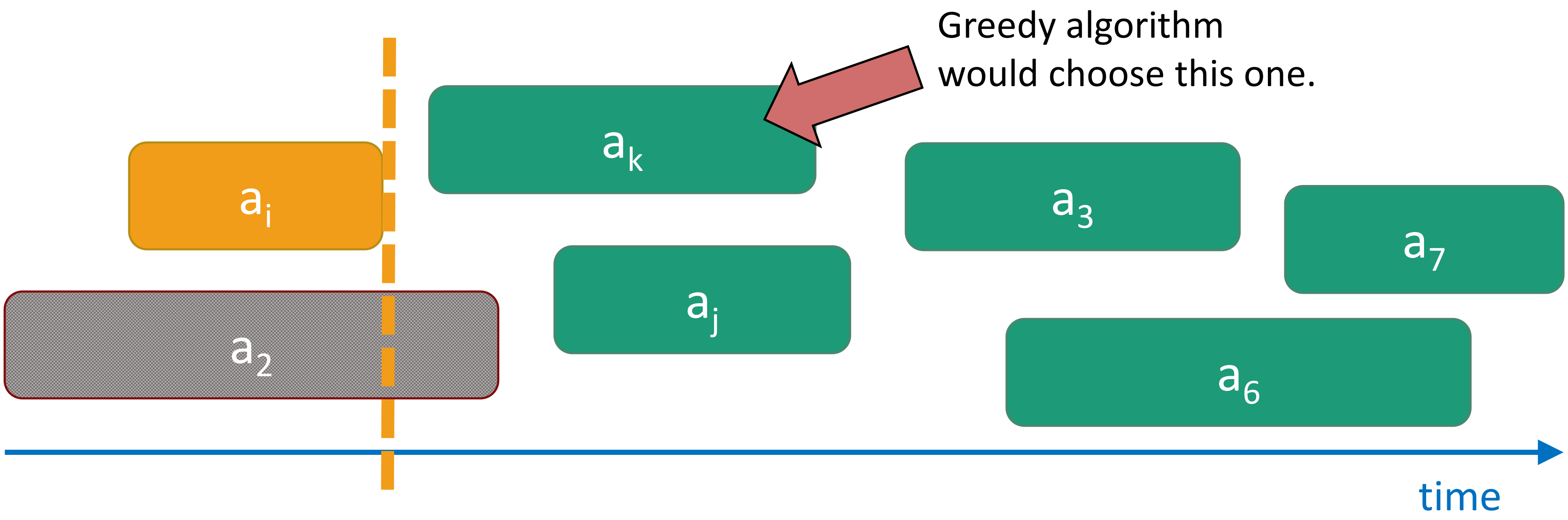


Assuming we can prove that

- **We never rule out an optimal solution.**
- At the end of the algorithm, we’ve got some solution.
- So it must be optimal.

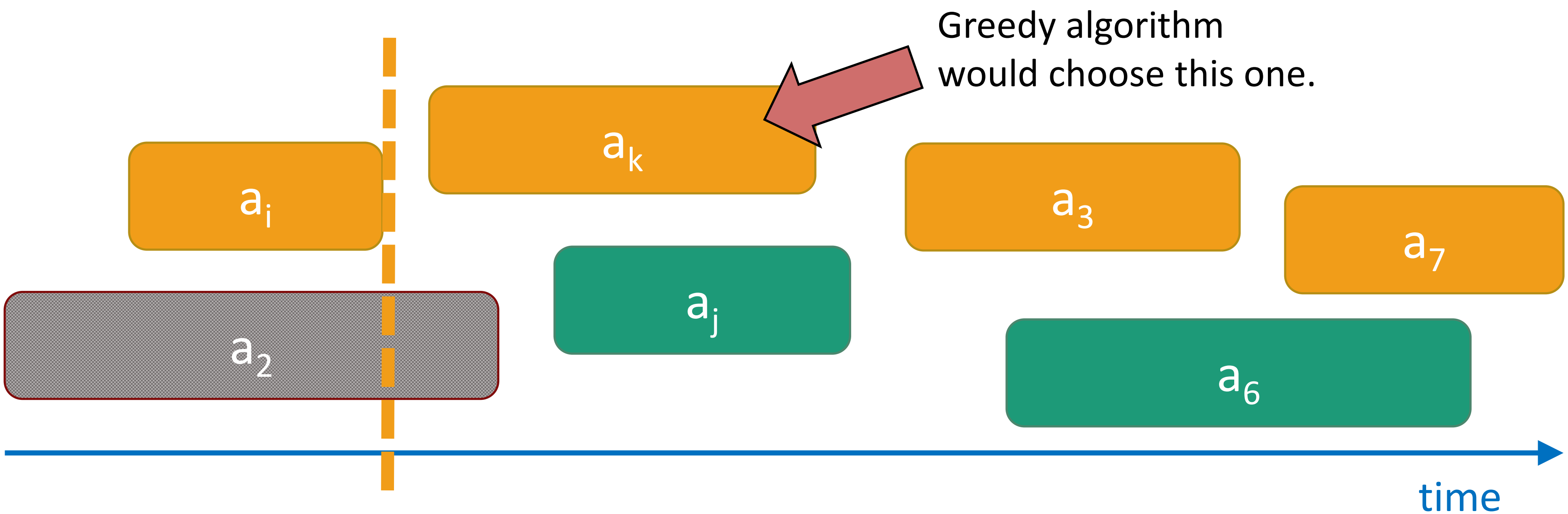
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.



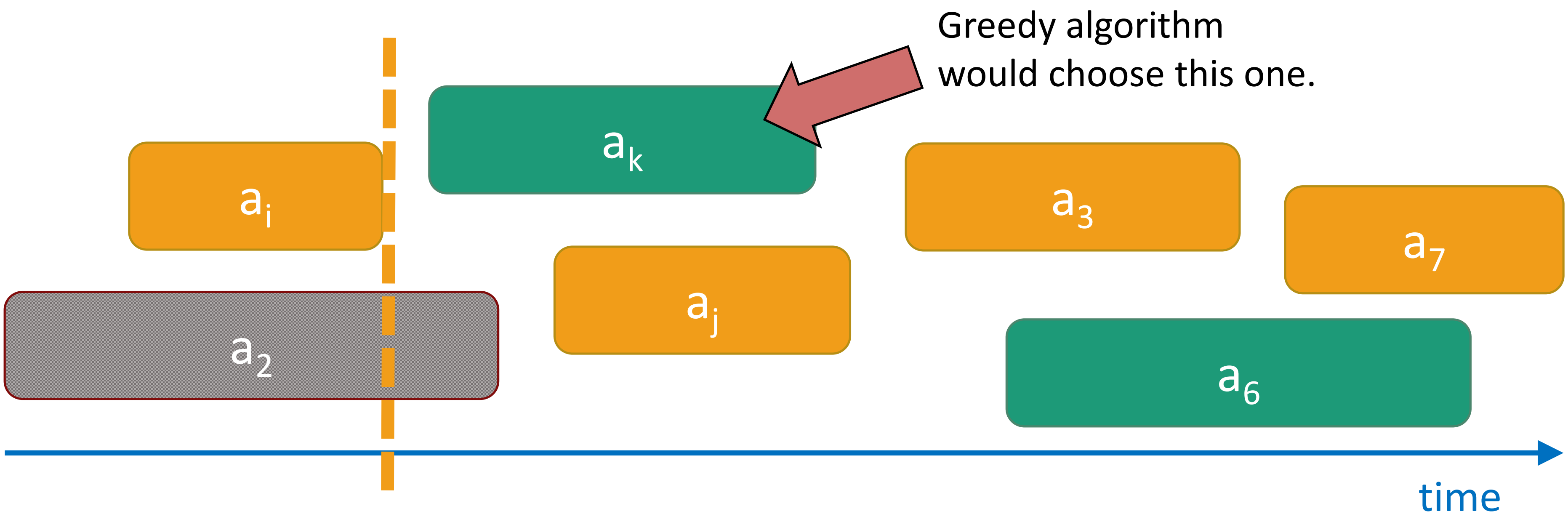
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is in T*, we're still on track.



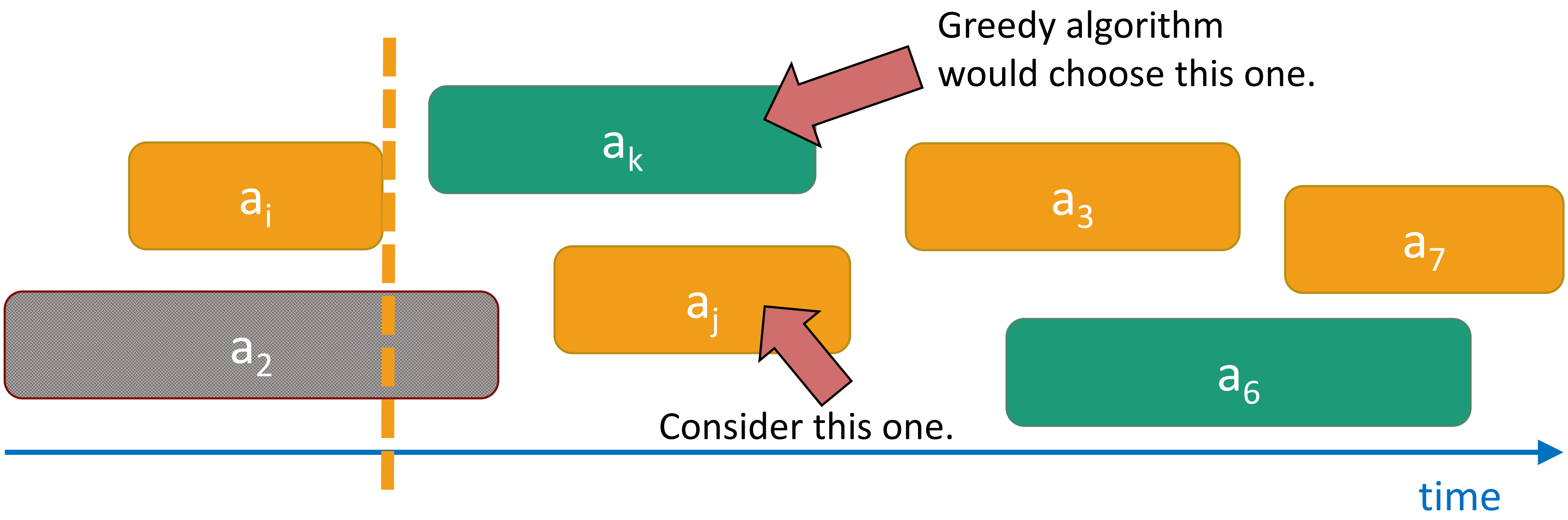
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is **not** in T*…



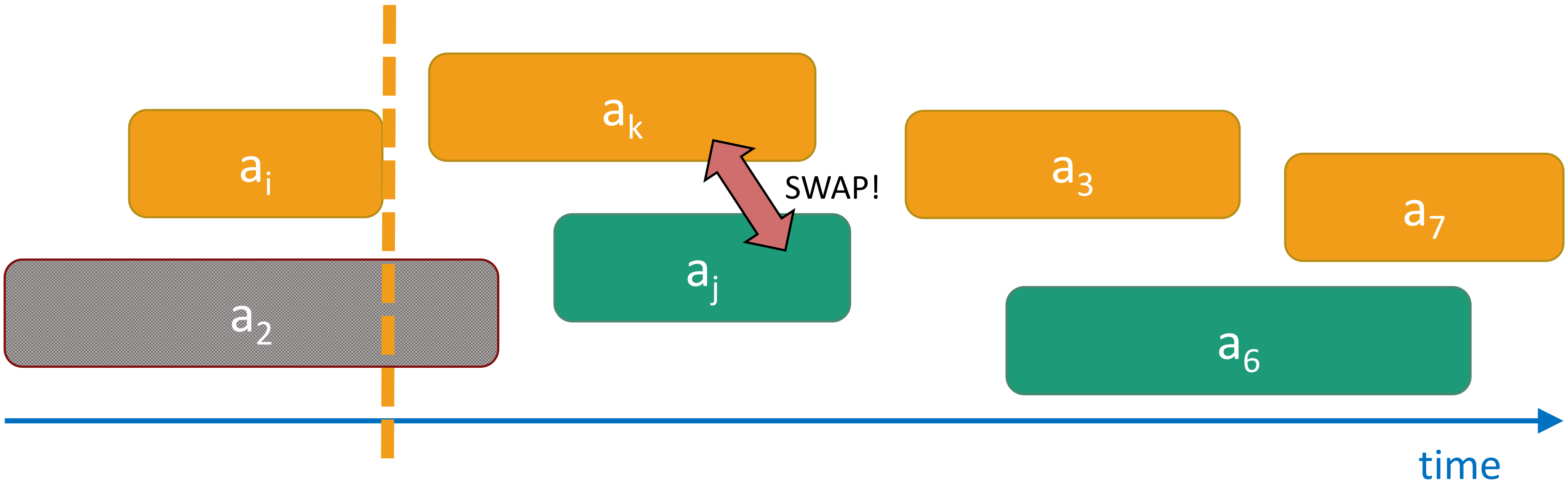
We never rule out an optimal solution

- If a_k is **not** in T*…
- Let a_j be the activity in T* (after a_i ends) with the smallest end time.



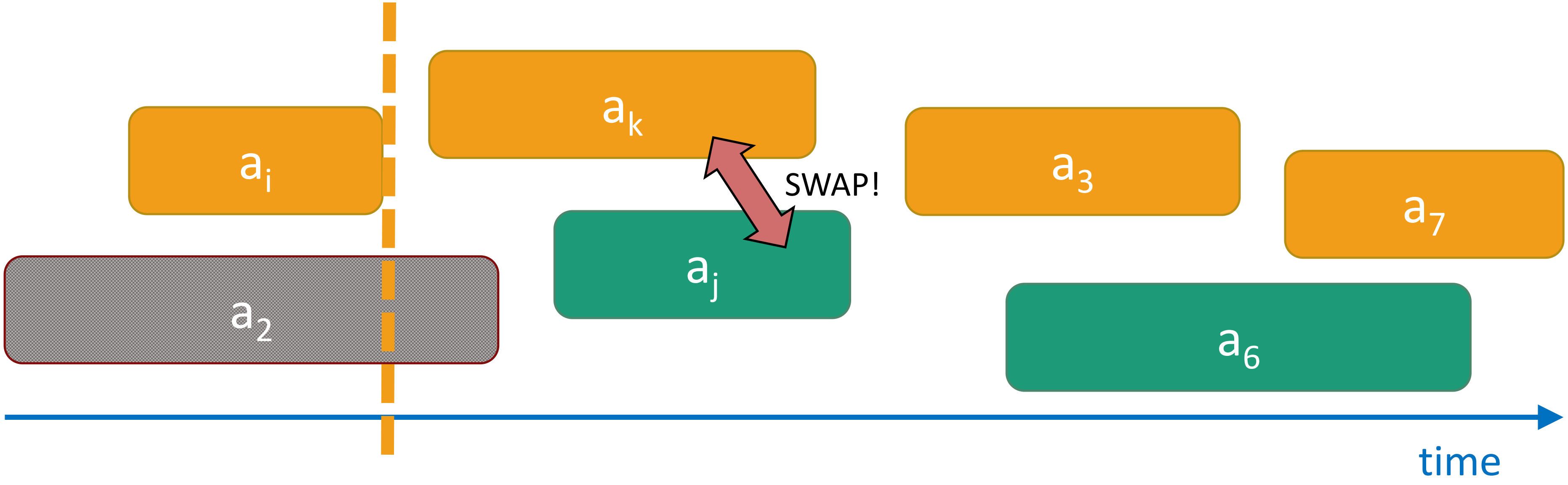
We never rule out an optimal solution

- If a_k is **not** in T*…
- Let a_j be the activity in T* (after a_i ends) with the smallest end time.
- Now consider schedule T you get by swapping a_j for a_k



We never rule out an optimal solution

- This schedule T is still allowed.
  - Since a_k has the smallest ending time, it ends before a_j.
  - Thus, a_k doesn't conflict with anything chosen after a_j.
- And T is still optimal.
  - It has the same number of activities as T*.



So the algorithm is correct

- **Inductive Hypothesis:**
  - After adding the t-th thing, there is an optimal solution that extends the current solution.
- **Base case:**
  - After adding zero activities, there is an optimal solution extending that.
- **Inductive step:**
  - **We just did that!**
- **Conclusion:**
  - After adding the last activity, there is an optimal solution that extends the current solution.
  - The current solution is the only solution that extends the current solution.
  - So the current solution is optimal.

Three Questions

1. Does this greedy algorithm for activity selection work?
   - Yes. (We will see why in a moment…)

2. In general, when are greedy algorithms a good idea?

3. The **“greedy”** approach is often the first you’d think of… Why are we getting to it now, in week 12?
   - Proving that greedy algorithms work is often not so easy…

One Common Strategy

- Make a series of choices.
- Show that, at each step, our choice won’t rule out an optimal solution at the end of the day.
- After we’ve made all our choices, we haven’t ruled out an optimal solution, so we must have found one.

- Note on “Common Strategy”
  - This common strategy is not the only way to prove that greedy algorithms are correct.
  - I’m emphasizing it in lecture because it often works, and it gives you a framework to get started.

Formally

- **Inductive Hypothesis:**
  - After greedy choice t, you haven't ruled out success.

"Success" here means "finding an optimal solution."

- **Base case:**
  - Success is possible before you make any choices.
- **Inductive step:**
  - If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.
- **Conclusion:**
  - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

For showing we don’t rule out success

- Suppose that you’re on track to make an optimal solution T*.
  - E.g., after you’ve picked activity i, you’re still on track.
- Suppose that T* <u>disagrees</u> with your next greedy choice. (showing by contradiction.)
  - E.g., it <u>doesn’t</u> involve activity k.
- Manipulate T* in order to make a solution T that’s not worse but that agrees with your greedy choice.
  - E.g., swap whatever activity T* did pick next with activity k.

Three Questions

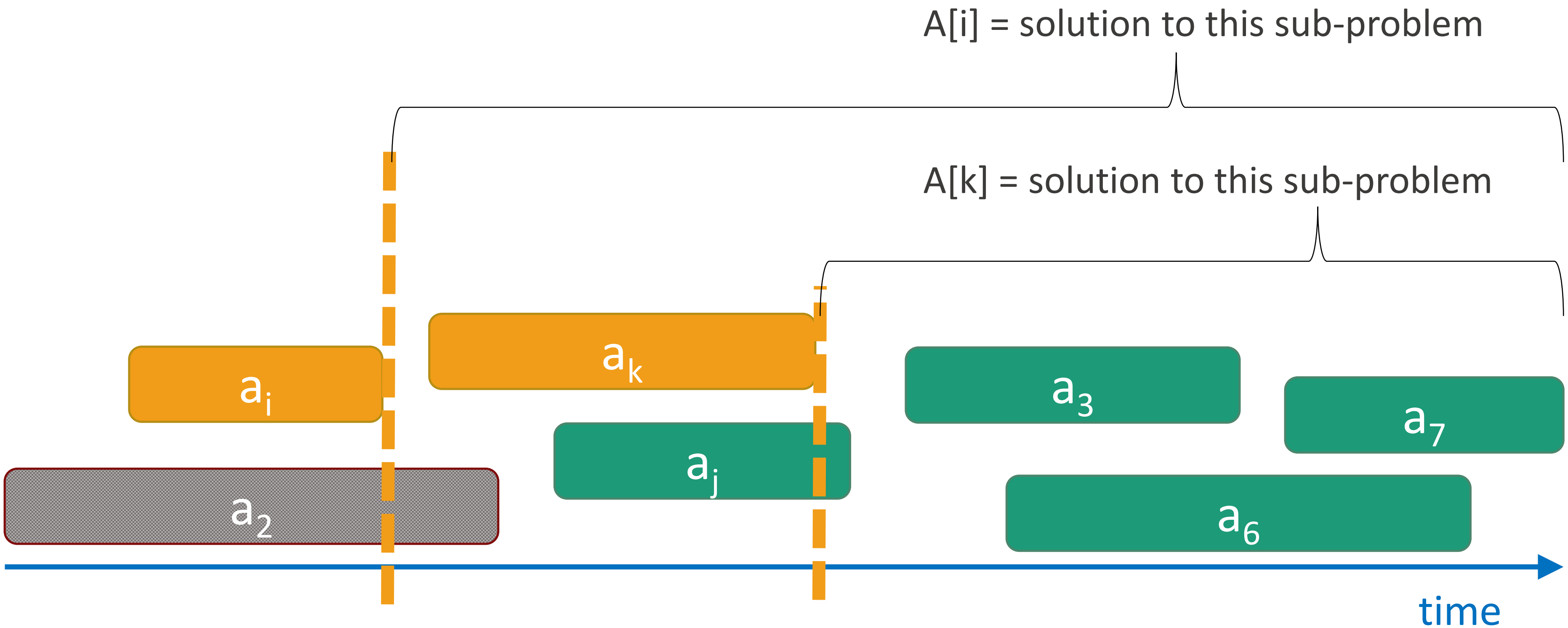
1. Does this greedy algorithm for activity selection work?
   - Yes. (We will see why in a moment…)

2. In general, when are greedy algorithms a good idea?
   - When the problem exhibits especially nice optimal substructure.

3. The **"greedy"** approach is often the first you'd think of… Why are we getting to it now, in Week 12?
   - Proving that greedy algorithms work is often not so easy…

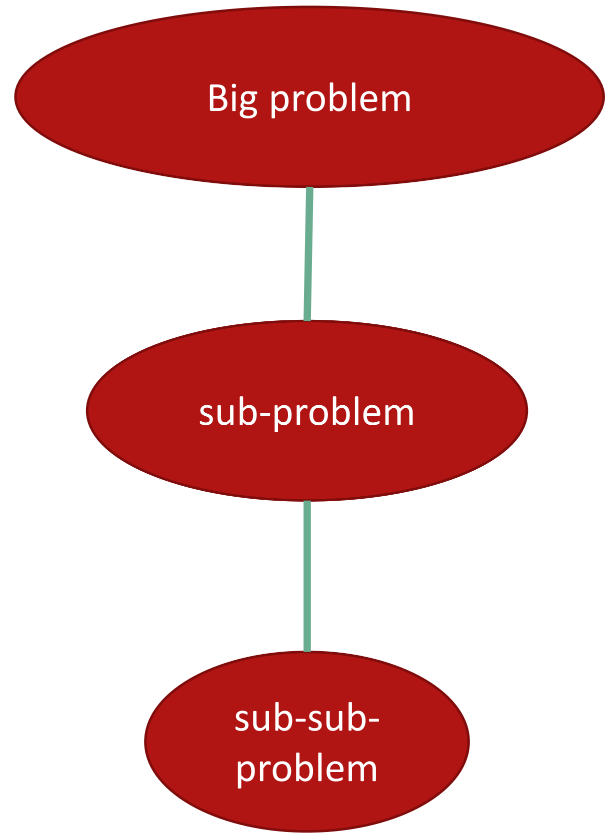
Optimal sub-structure

- Our greedy activity selection algorithm exploited a natural sub-problem structure:
  - A[i] = number of activities you can do after the end of activity i
  - Then A[i] = A[k] + 1.



Sub-problem graph view

- **Greedy algorithms:**
- Not only is there optimal sub-structure:
  - Optimal solutions to a problem are made up from optimal solutions of sub-problems
- but each problem depends on only one sub-problem.



*Check a DP version of activity selection in CLRS 16.

Three Questions

1. Does this greedy algorithm for activity selection work?
   - Yes. (We will see why in a moment…)
2. In general, when are greedy algorithms a good idea?
   - When the problem exhibits especially nice optimal substructure.
3. The **"greedy"** approach is often the first you'd think of… Why are we getting to it now, in Week 12?
   - Proving that greedy algorithms work is often not so easy…

Let’s see a few more examples

- Huffman Coding (today)
- Minimum Spanning Tree (next time)

Huffman Coding

Huffman Coding

- everyday english sentence
  - 01100101 01110110 01100101 01110010 01111001 01100100 01100001 01111001 00100000 01100101 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101
- qwertyui_opasdfg+hjklzxcv
  - 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Huffman Coding

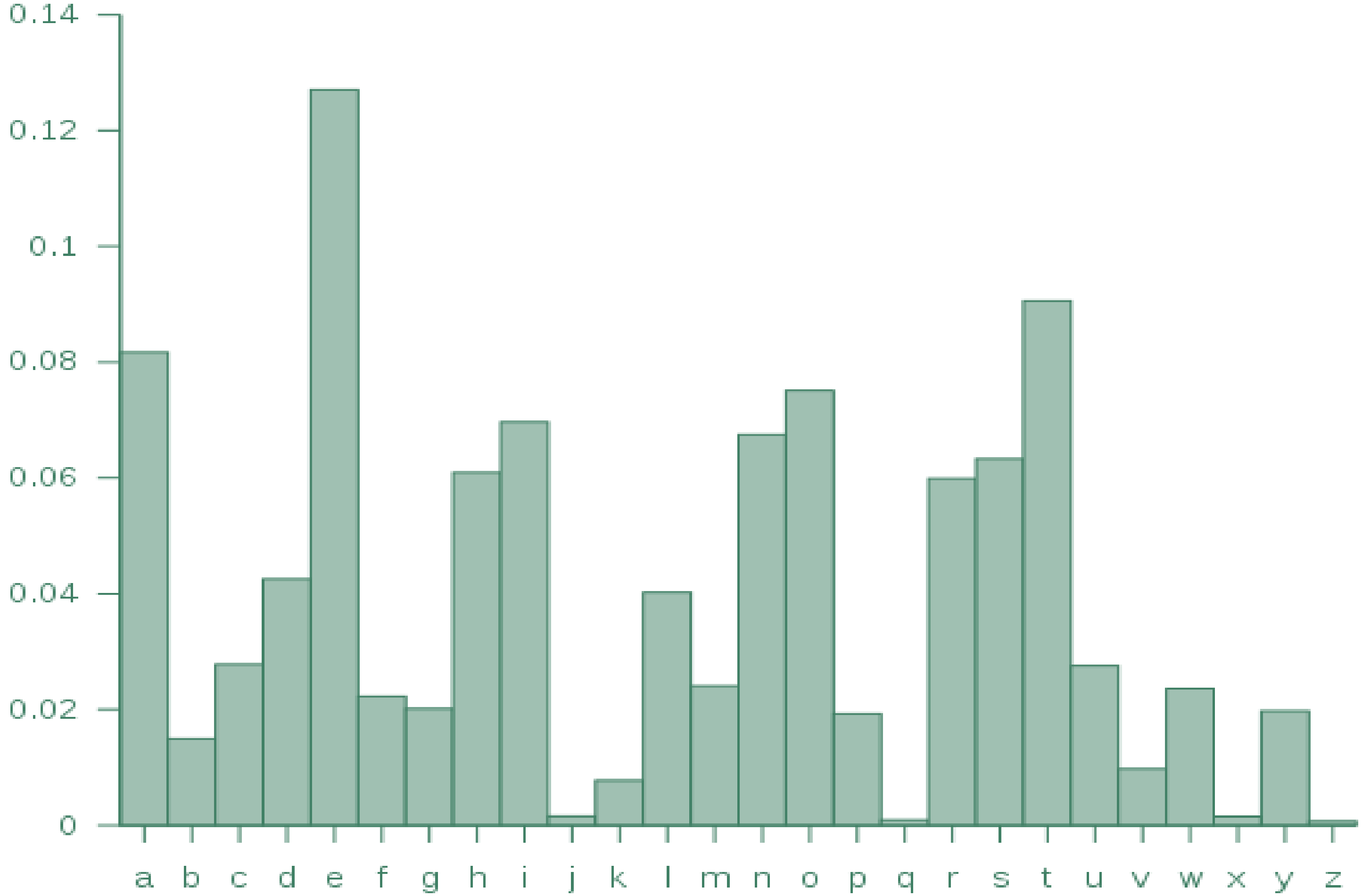
- **e**v**e**ryday **e**nglish s**e**nt**e**nc**e**
  - **01100101** 01110110 **01100101** 01110010 01111001 01100100 01100001 01111001 00100000 **01100101** 01101110 01100111 01101100 01101001 01110011 01101000 00100000 01110011 **01100101** 01101110 01110100 **01100101** 01101110 01100011 **01100101**

ASCII is pretty wasteful for English sentences. If **e** shows up so often, we should have a shorter way of representing it!

- qwertyui_opasdfg+hjklzxcv
  - 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

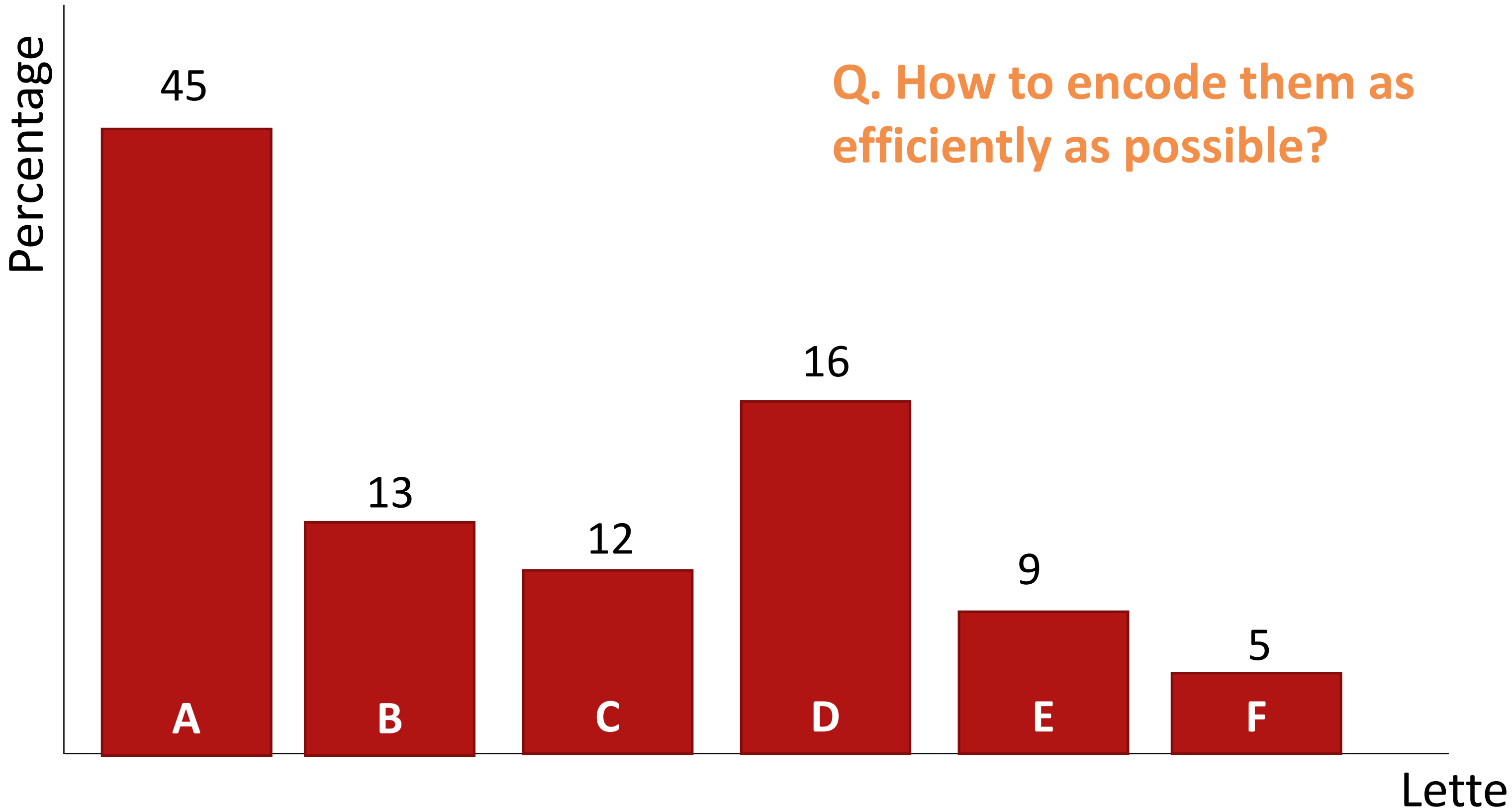
Suppose that

- We have some distribution on characters.



Suppose that

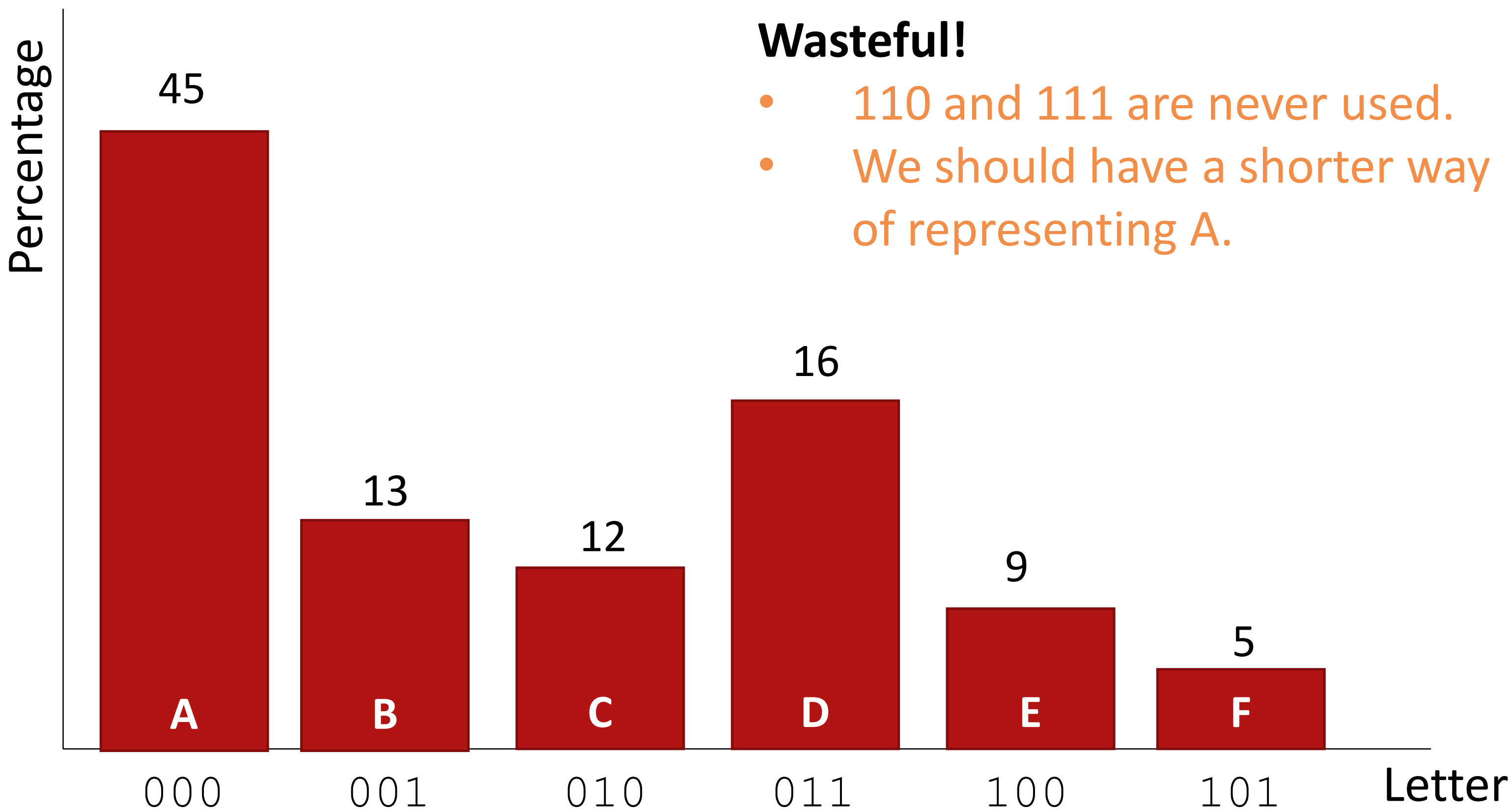
- We have some distribution on characters.
  - Suppose we have 6 characters.



Q. How to encode them as efficiently as possible?

Try 0 (like ASCII)

- Every letter is assigned a binary string of three bits.

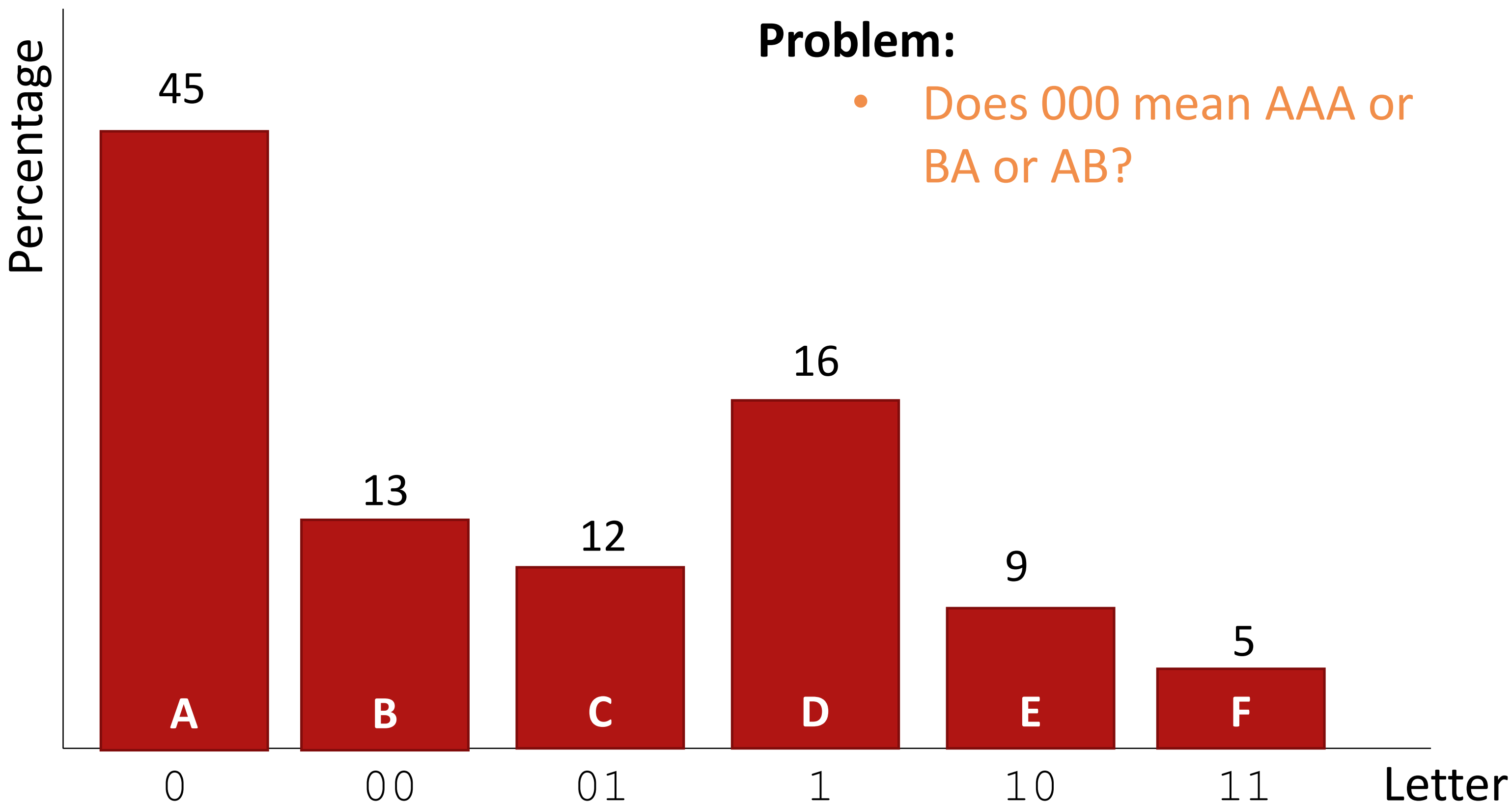


Wasteful!

- 110 and 111 are never used.
- We should have a shorter way of representing A.

Try 1

- Every letter is assigned a binary string of one or two bits.
- More frequent letters get shorter strings.

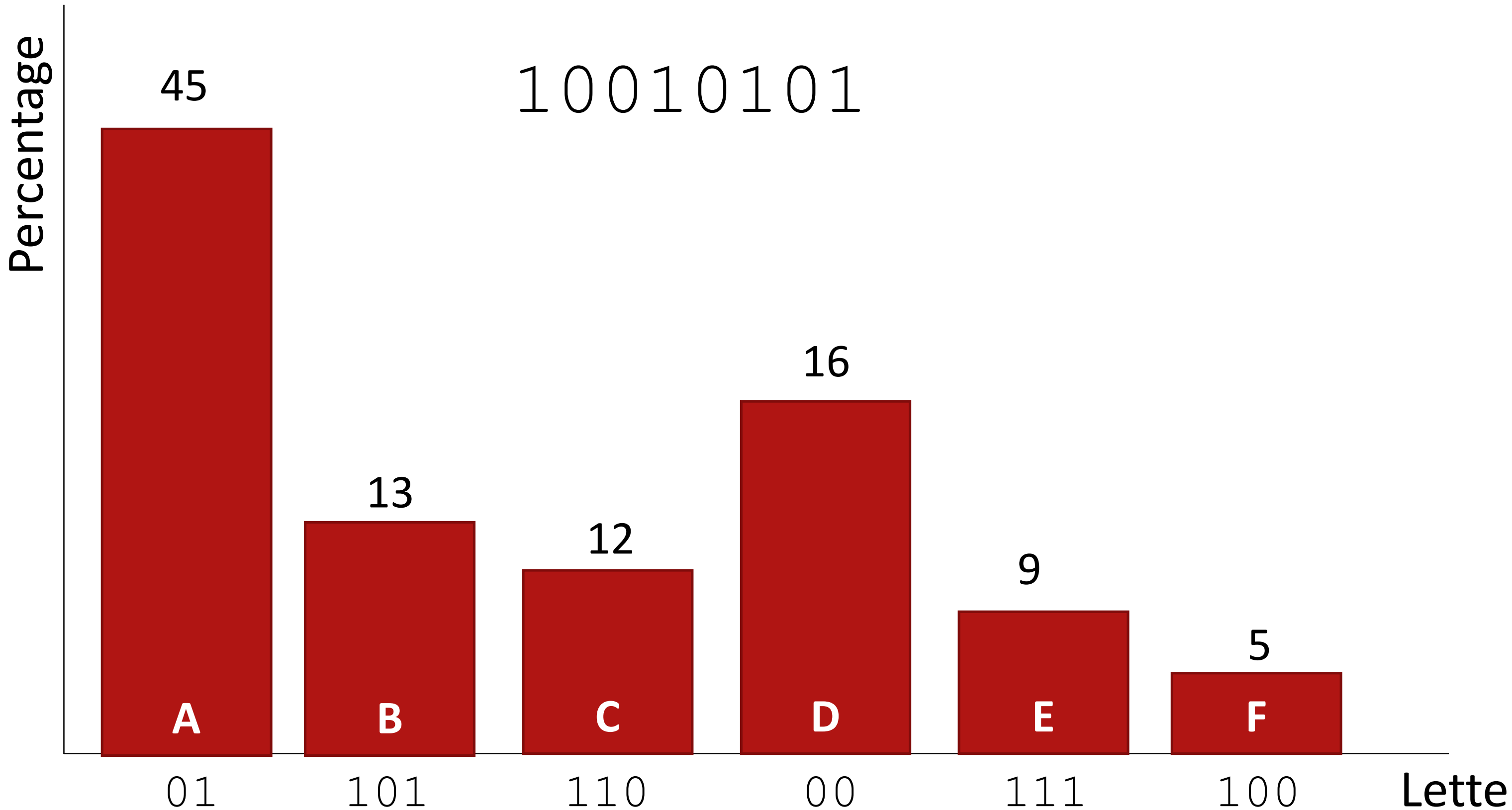


Problem:

- Does 000 mean AAA or BA or AB?

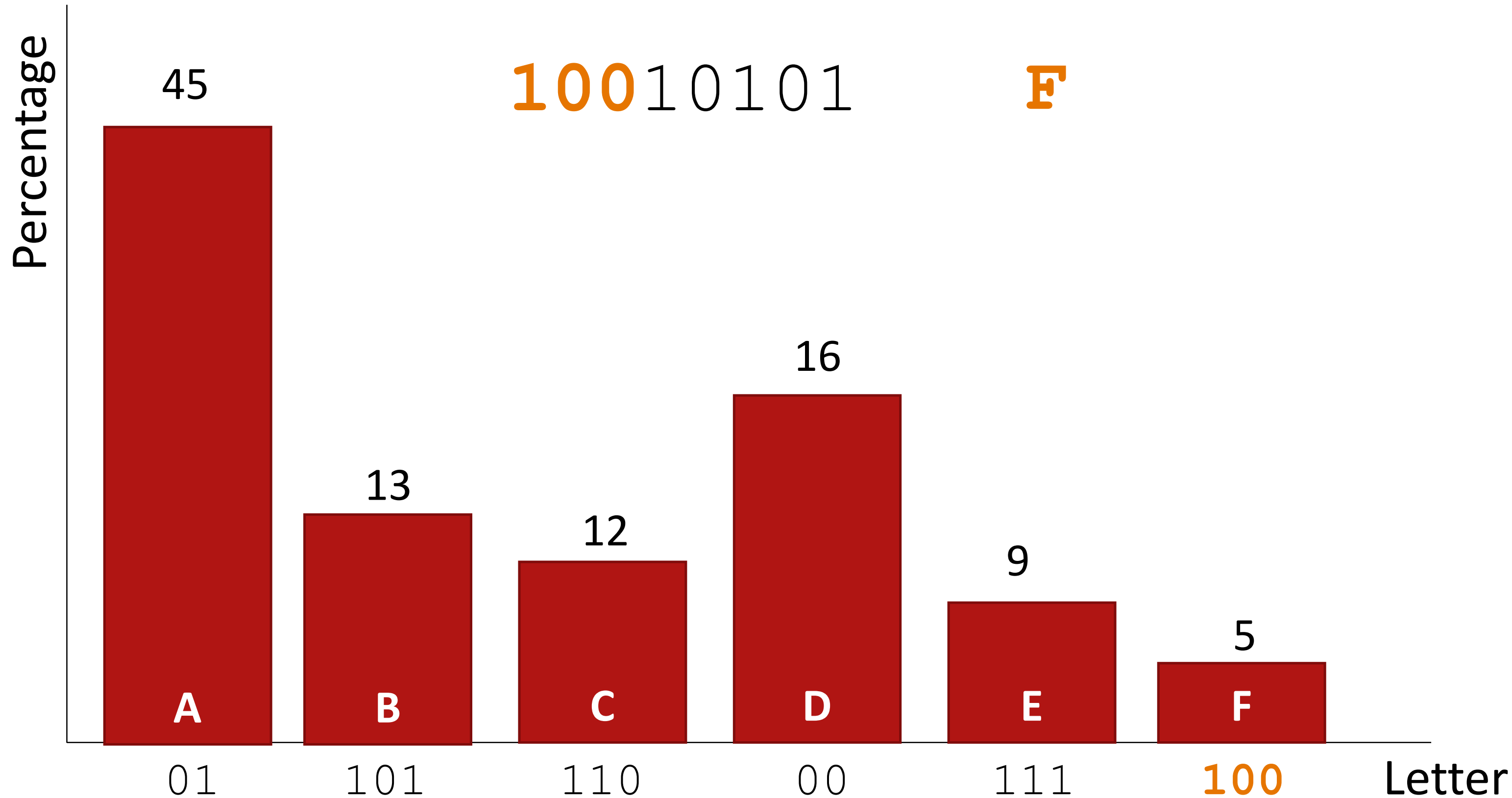
Try 2: Prefix-free Coding

- Every letter is assigned a binary string.
- More frequent letters get shorter strings.
- No encoded string is a prefix of any other.



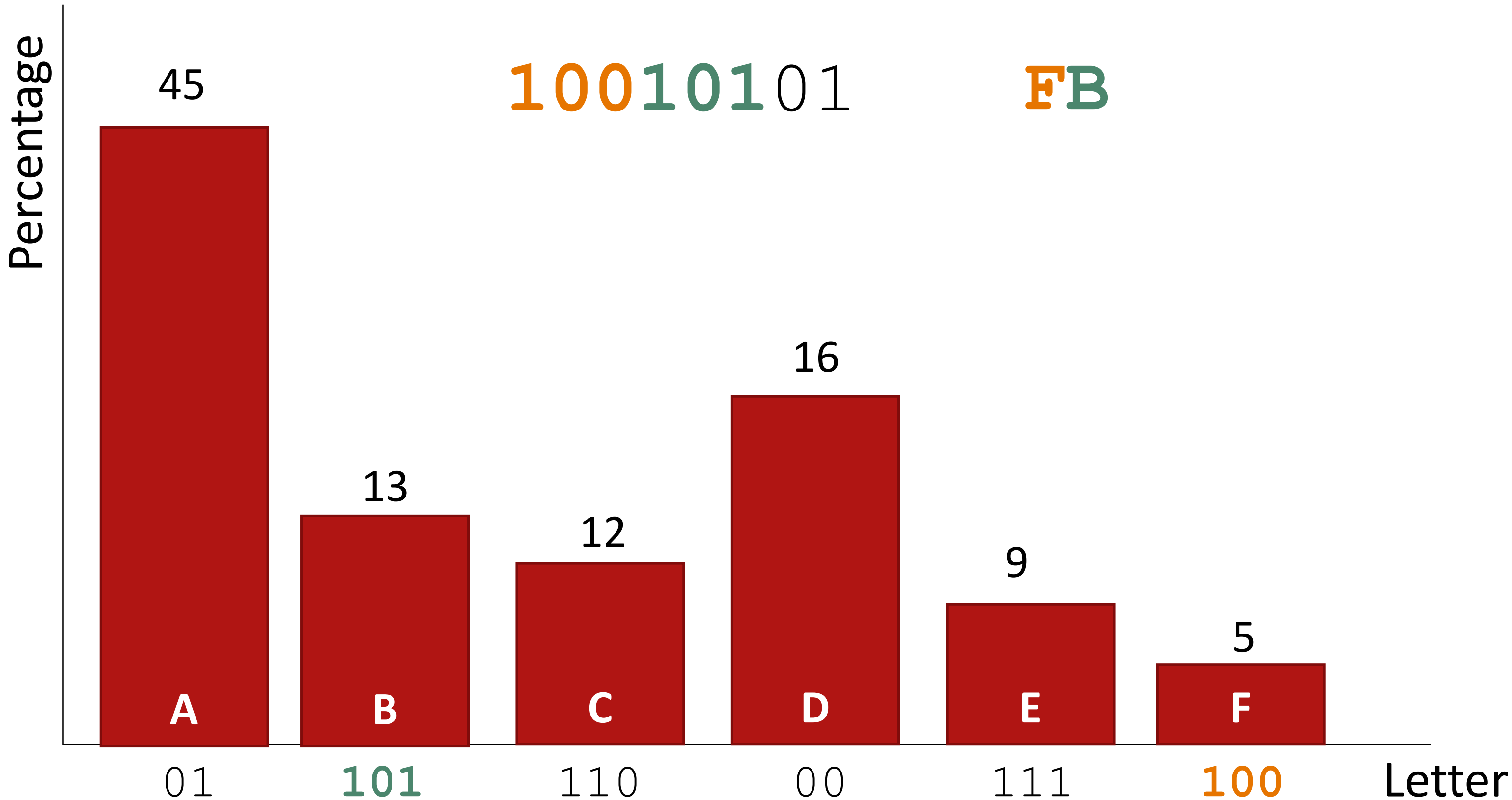
Try 2: Prefix-free Coding

- Every letter is assigned a binary string.
- More frequent letters get shorter strings.
- No encoded string is a prefix of any other.



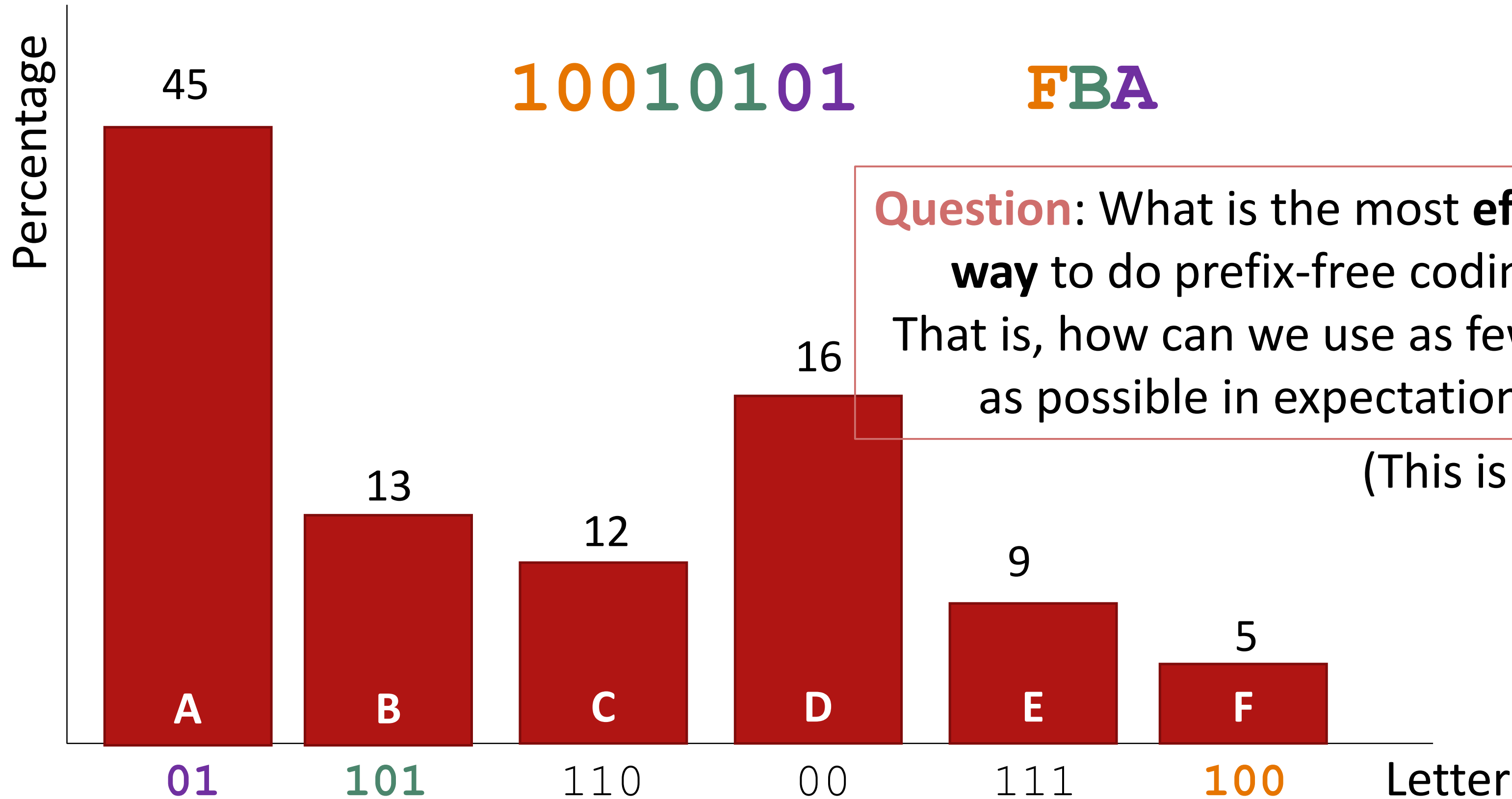
Try 2: Prefix-free Coding

- Every letter is assigned a binary string.
- More frequent letters get shorter strings.
- No encoded string is a prefix of any other.



Try 2: Prefix-free Coding

- Every letter is assigned a binary string.
- More frequent letters get shorter strings.
- No encoded string is a prefix of any other.

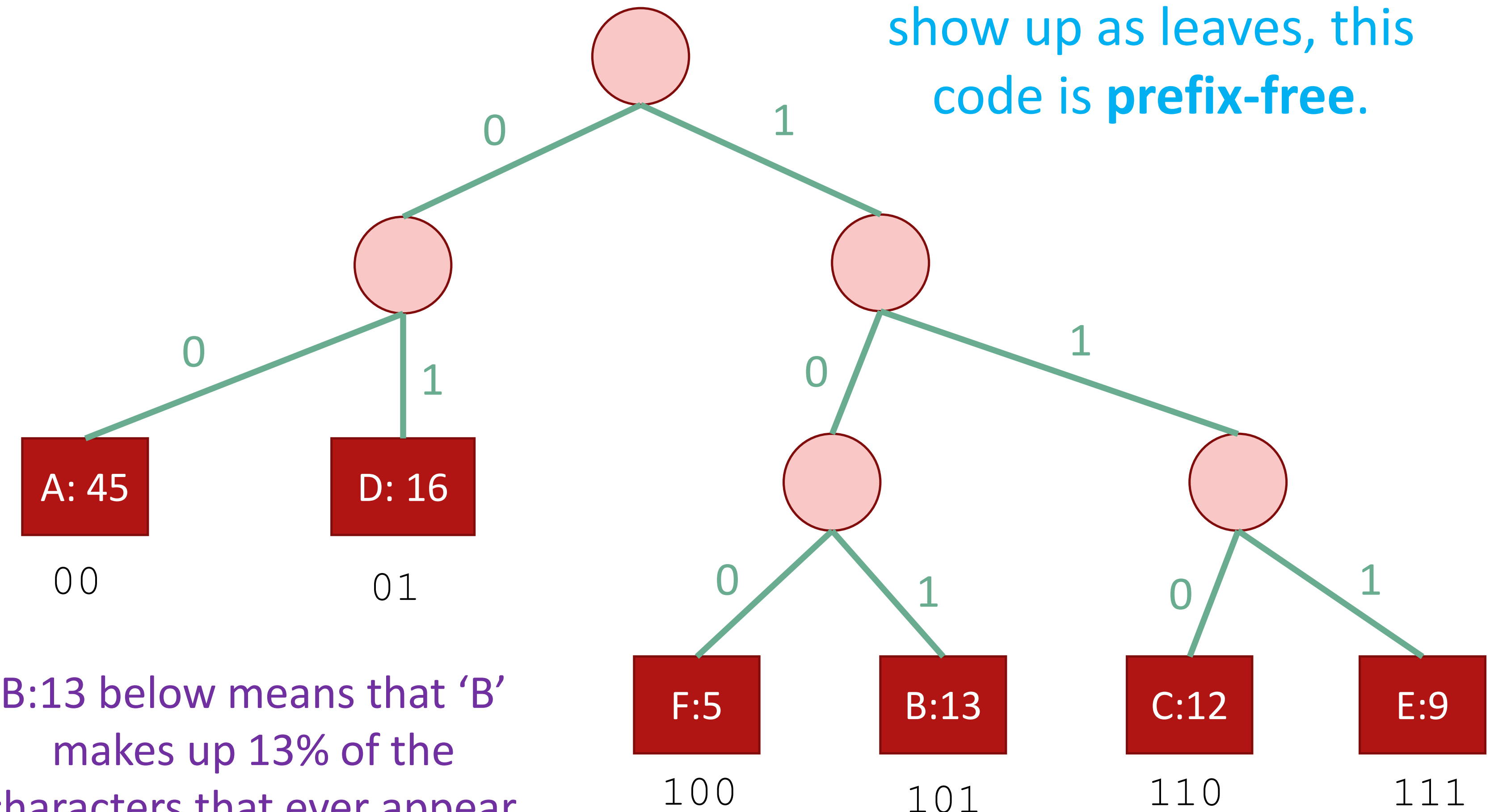


Question: What is the most **efficient way** to do prefix-free coding? That is, how can we use as few bits as possible in expectation?

(This is not it).

A prefix-free code is a tree

As long as all the letters show up as leaves, this code is **prefix-free**.



B:13 below means that 'B' makes up 13% of the characters that ever appear.

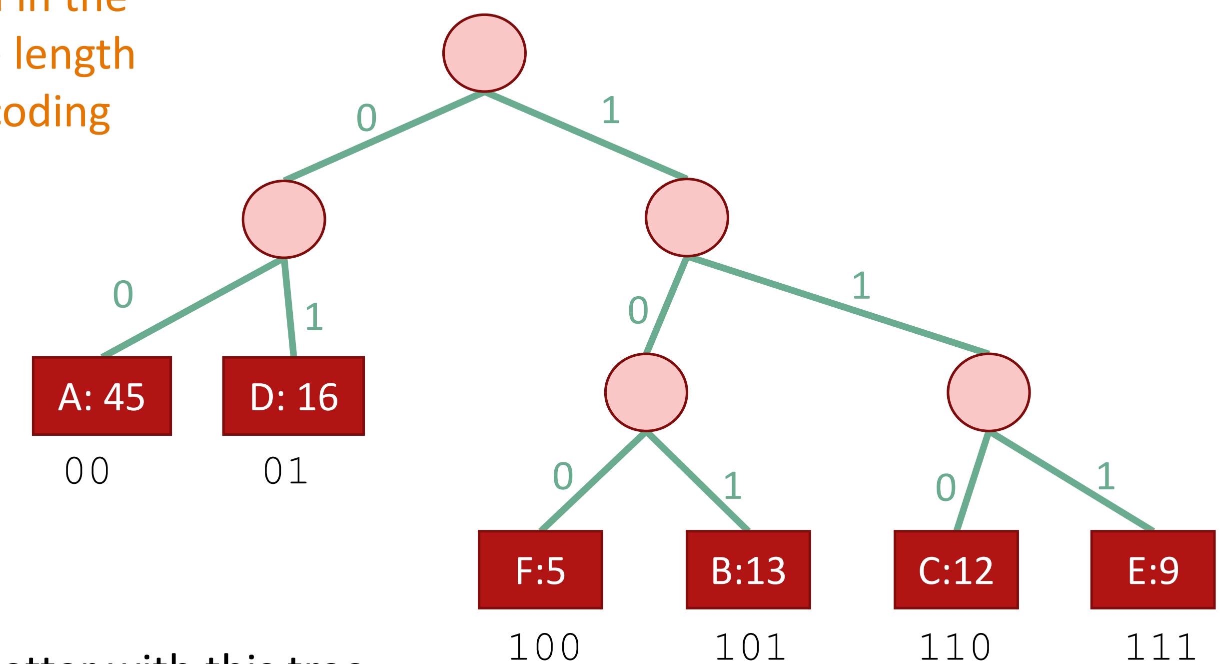
How good is a tree?

- Imagine choosing a letter at random from the language.
  - Not uniformly random, but according to our histogram.
- The cost of a tree is the expected length of the encoding of a random letter.

The depth in the tree is the length of the encoding

Cost = $\sum_{leaves\ x} \mathrm{P(x)} \cdot \mathrm{depth}(x)$

P(x) is the probability of letter x



Expected cost of encoding a letter with this tree:

$$2(0.45 + 0.16) + 3(\ 0.05 + 0.13 + 0.12 + 0.09) = 2.39$$

Goal

- Given a distribution P on letters, find the lowest-cost tree, where

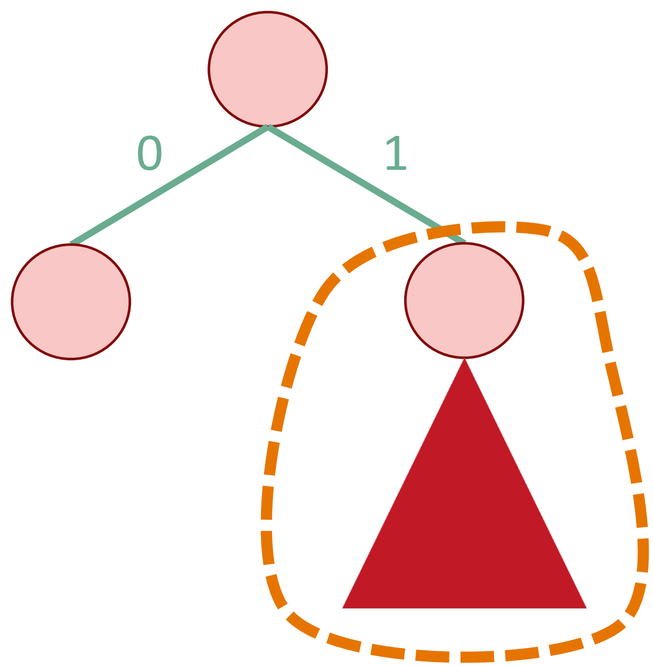
$$\mathrm{cost(tree)} = \sum_{\mathrm{leaves}\ x} \mathrm{P(x)} \cdot \mathrm{depth}(x)$$

P(x) is the probability of letter x

The depth in the tree is the length of the encoding

Greedy algorithm

- **Greedy goal:** less frequent letters should be further down the tree.
- **Approach:** Greedily build sub-trees from the bottom up.



What's a **safe choice** to make for these lower sub-trees?

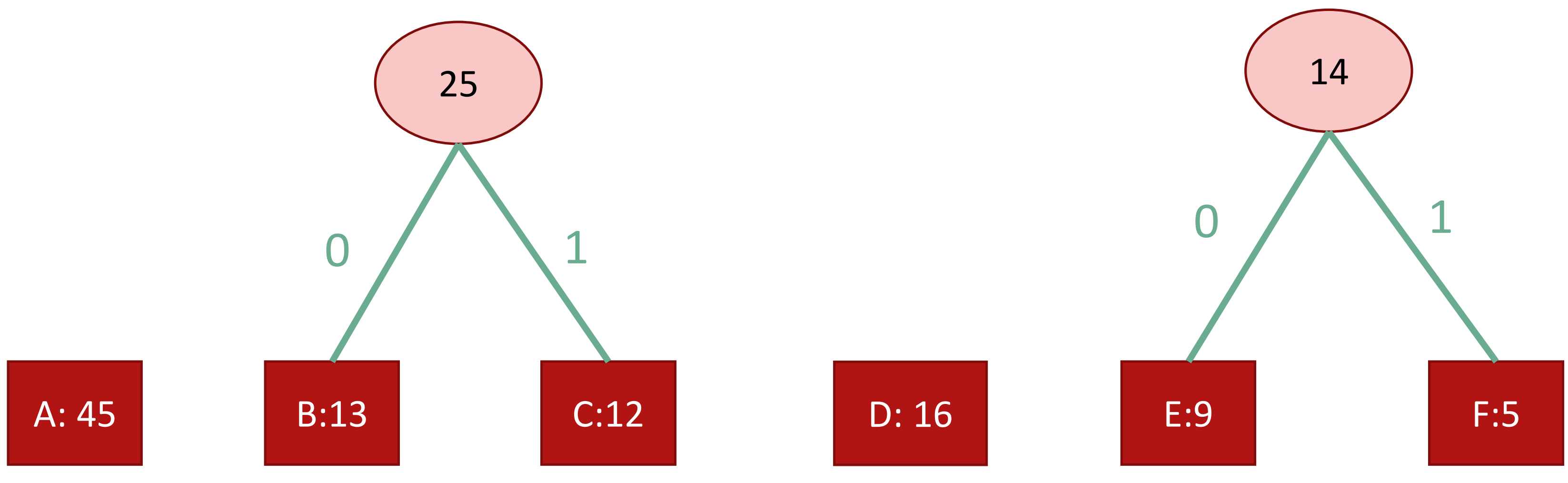
Infrequent elements! We want them as low down as possible.

Solution

- Greedily build subtrees, starting with the infrequent letters

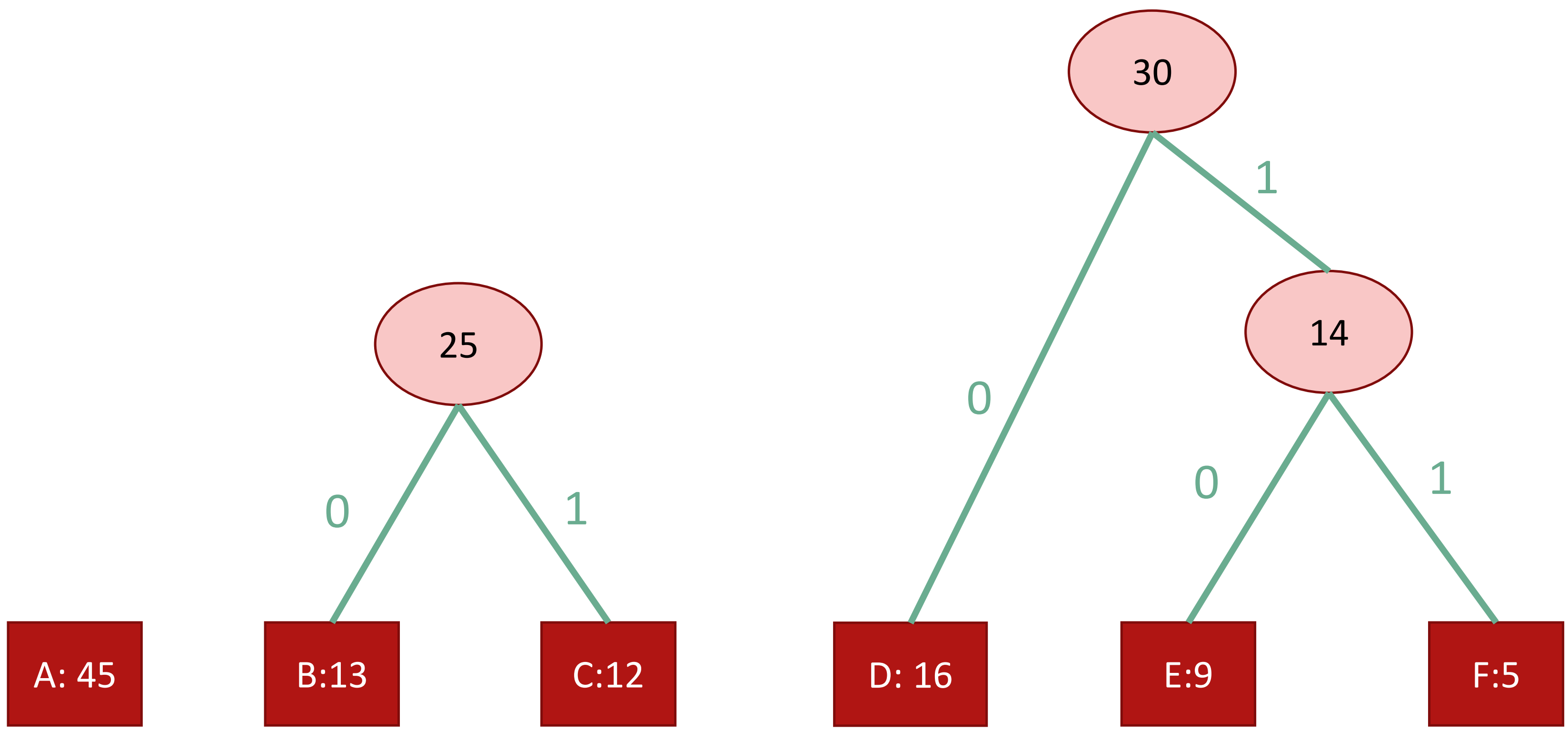
Solution

- Greedily build subtrees, starting with the infrequent letters



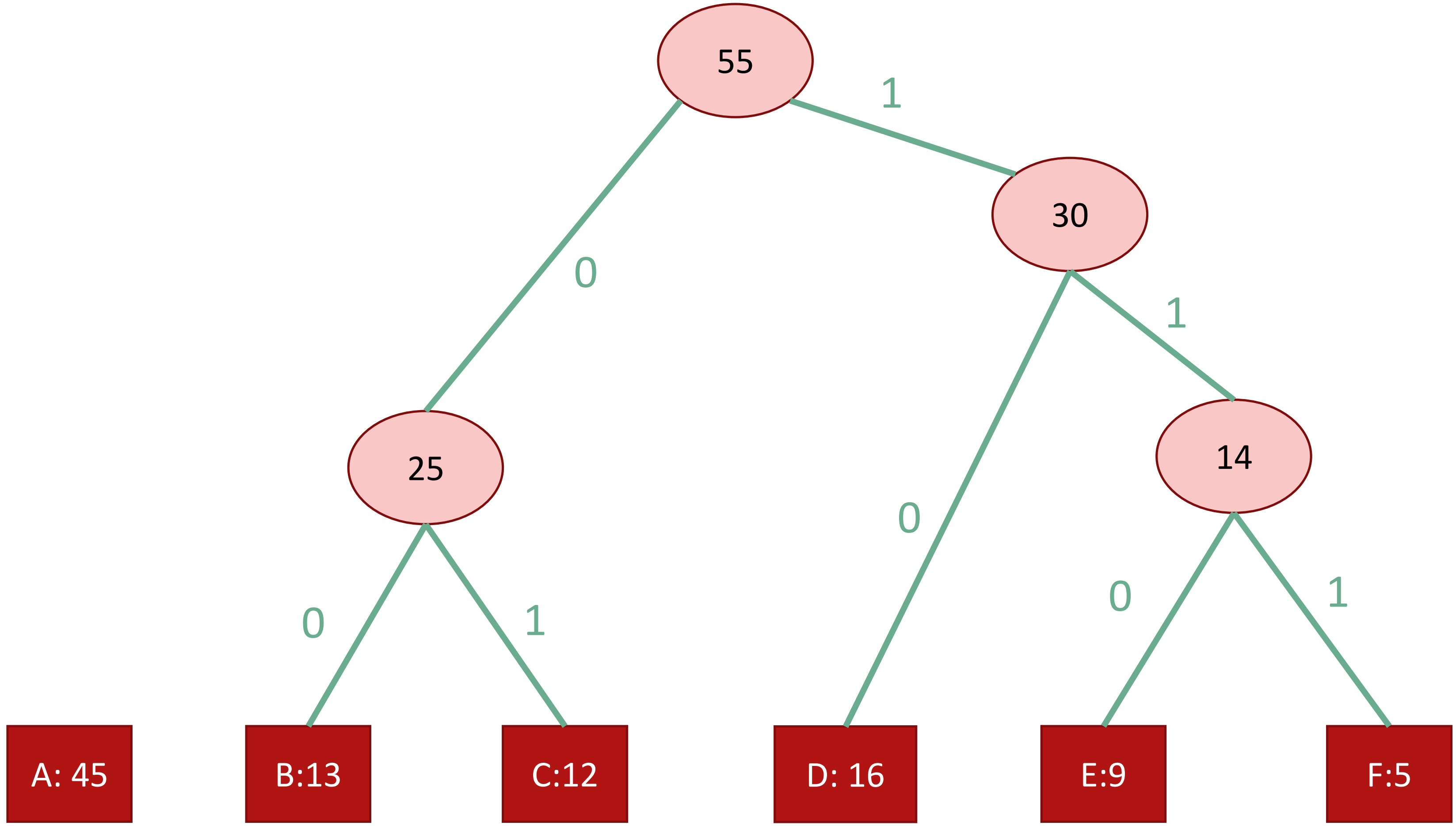
Solution

- Greedily build subtrees, starting with the infrequent letters



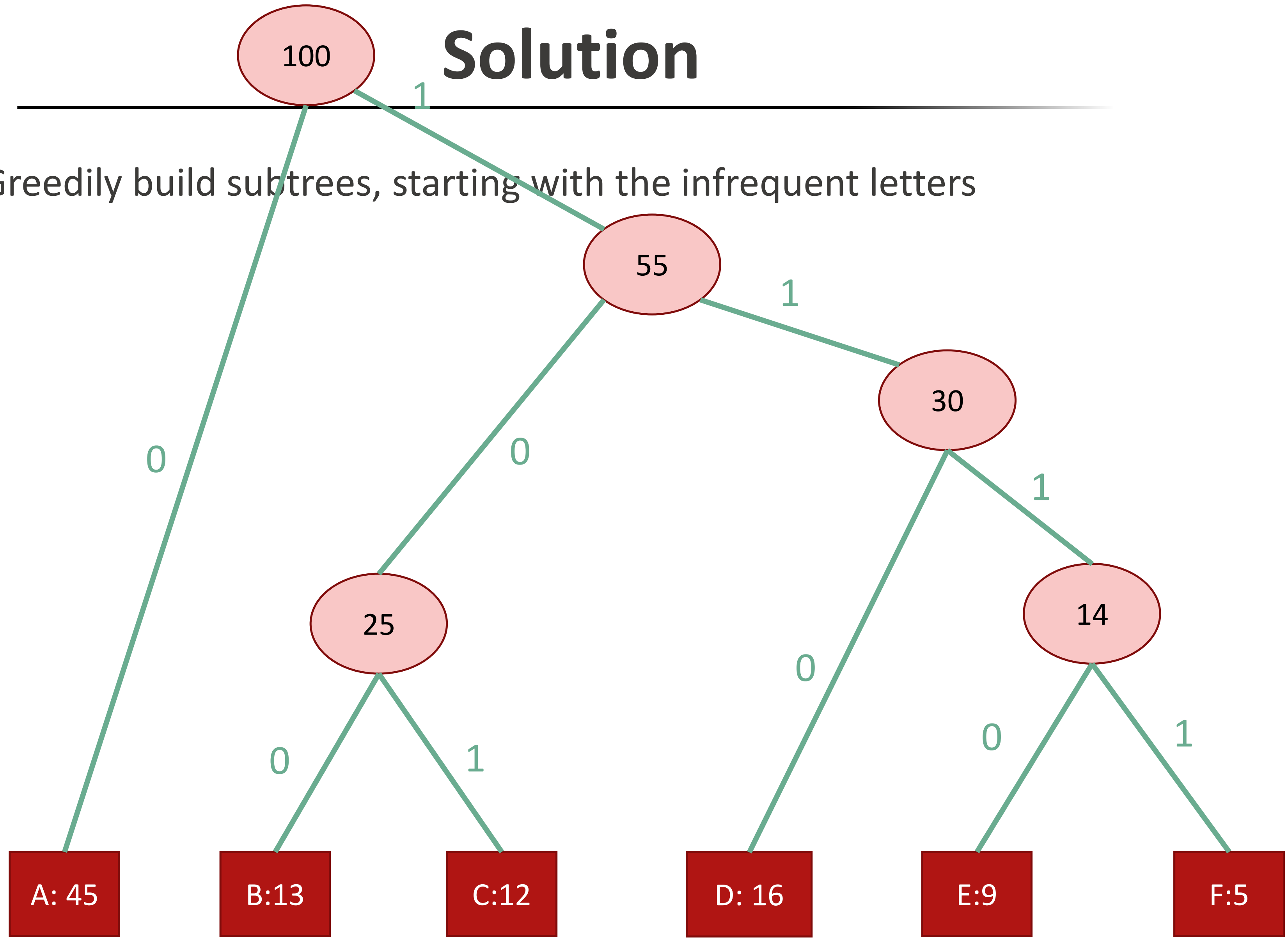
Solution

- Greedily build subtrees, starting with the infrequent letters



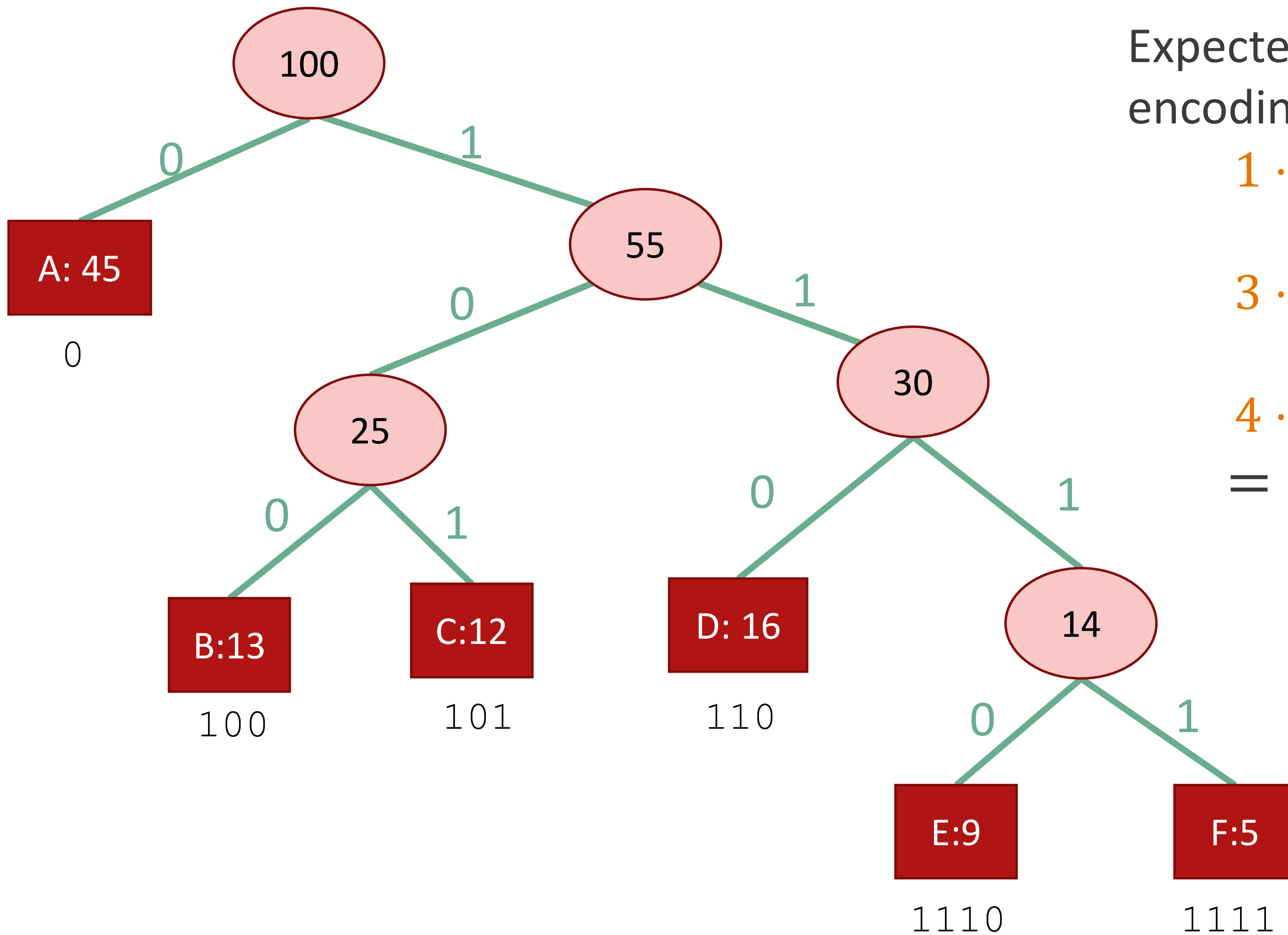
Solution

- Greedily build subtrees, starting with the infrequent letters



Solution

- Greedily build subtrees, starting with the infrequent letters

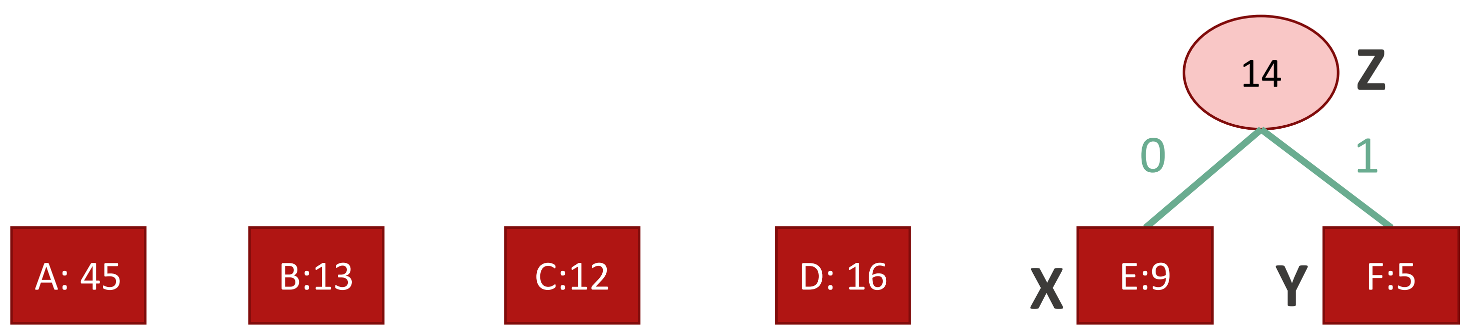


Expected cost of encoding a letter:

$$1 \cdot 0.45 + 3 \cdot 0.41 + 4 \cdot 0.14 = 2.24$$

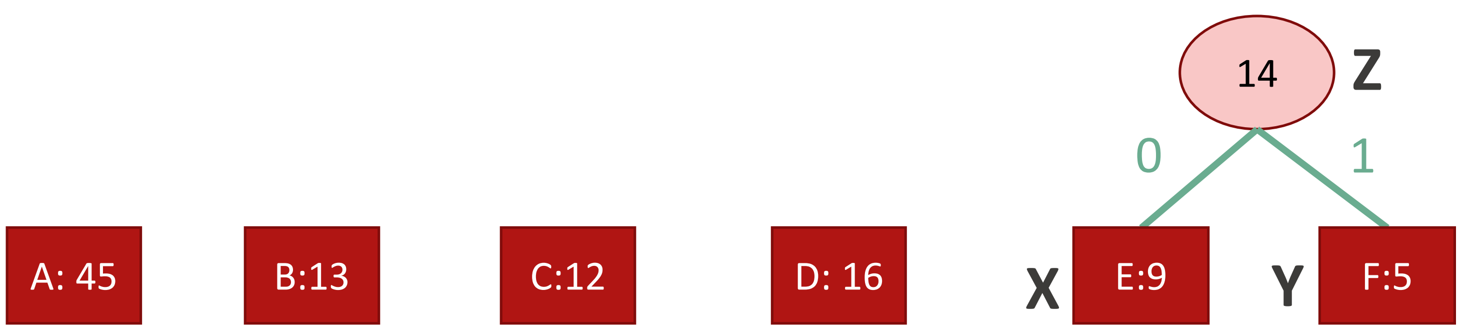
What exactly was the algorithm?

- Create a node like D: 16 for each letter/frequency
- Let CURRENT be the list of all these nodes.
- **while** len(CURRENT) > 1:
  - X and Y ← the nodes in CURRENT with the smallest keys.
  - Create a new node Z with Z.key = X.key + Y.key
  - Set Z.left = X, Z.right = Y
  - Add Z to CURRENT and remove X and Y
- **return** CURRENT[0]



This is called Huffman Coding

- Create a node like D: 16 for each letter/frequency
- Let CURRENT be the list of all these nodes.
- **while** len(CURRENT) > 1:
  - X and Y ← the nodes in CURRENT with the smallest keys.
  - Create a new node Z with Z.key = X.key + Y.key
  - Set Z.left = X, Z.right = Y
  - Add Z to CURRENT and remove X and Y
- **return** CURRENT[0]



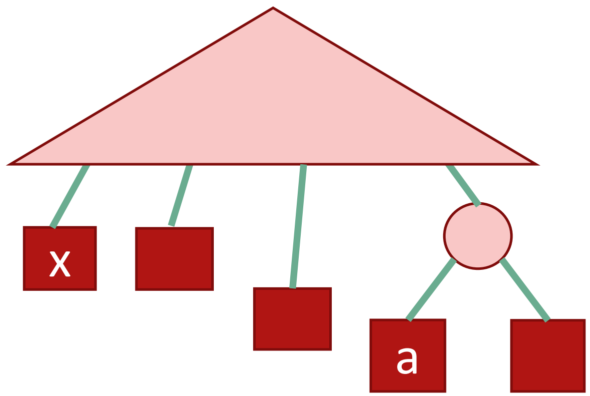
Does it work?

- We will **sketch** a proof here.
- Same strategy:
  - Show that at each step, the choices we are making **won't rule out** an optimal solution.

- We will use this:
  - Lemma 1: Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

Lemma 1

- **Lemma 1:** If x and y are the two least-frequent letters, there is an optimal tree where x and y are siblings.
- **Proof Idea.** Say that an optimal tree looks like this:

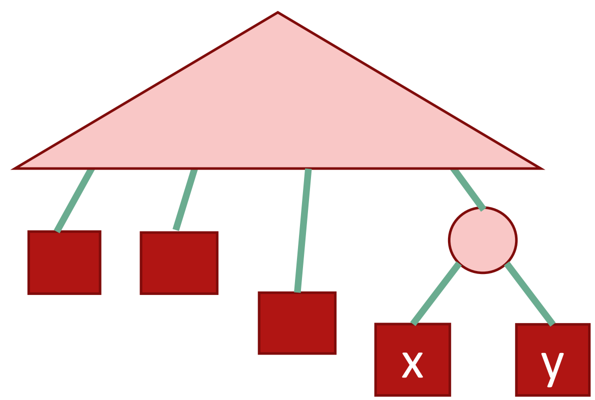


Lowest-level sibling nodes: at least one of them is neither x nor y

  - What happens to the cost if we swap **x** for **a**?
    - The cost can't increase; **a** was more frequent than **x**, and we just made **a**'s encoding shorter and **x**'s longer.

Lemma 1

- **Lemma 1:** If x and y are the two least-frequent letters, there is an optimal tree where x and y are siblings.
- **Proof Idea.** Say that an optimal tree looks like this:



Lowest-level sibling nodes: at least one of them is neither x nor y

  - What happens to the cost if we swap **x** for **a**?
    - The cost can't increase; **a** was more frequent than **x**, and we just made **a**'s encoding shorter and **x**'s longer.
  - Repeat this logic until we get an optimal tree with **x** and **y** as siblings.
    - The cost never increased so this tree is still optimal.

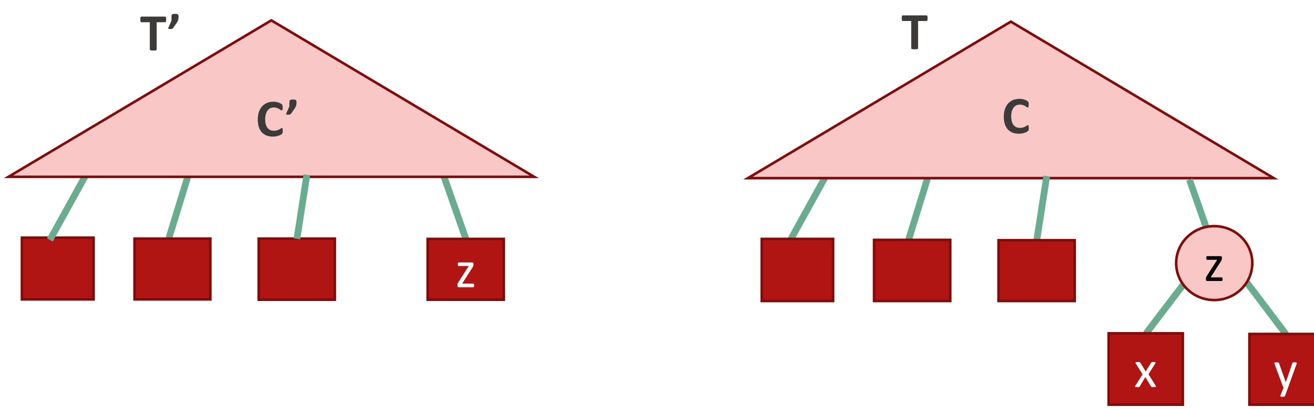
The whole argument

- Let C be a given alphabet with frequency c.freq defined for each character $c \in C$.
- Let x and y be two characters in C with minimum frequency.
- Let C' be the alphabet C with the characters x and y removed and a new character z added.

z.freq = x.freq + y.freq.

- **Lemma 2:** We suppose that T' is an optimal for the C'.

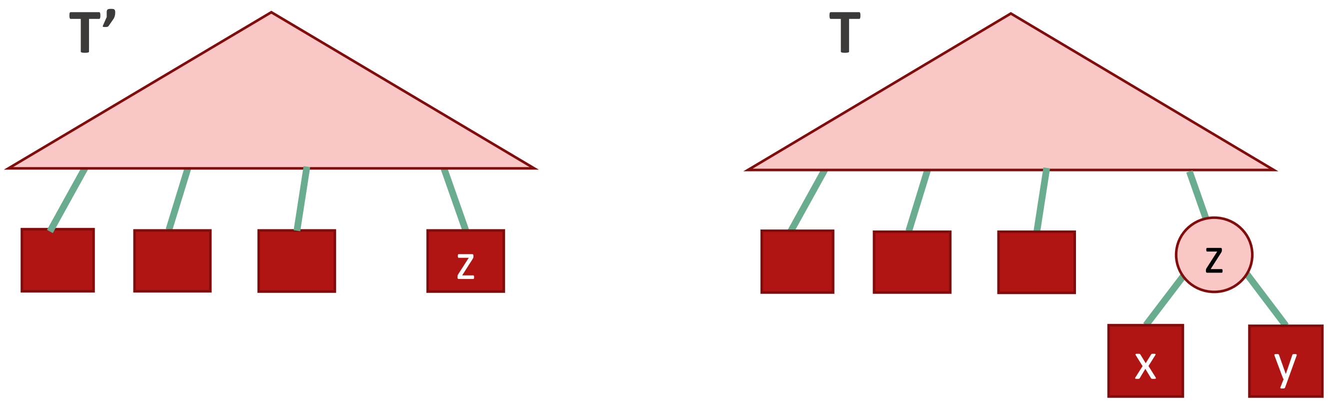
T is an optimal for the C'.



We want to prove T is an optimal for the C.

- Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, is an optimal.

The whole argument

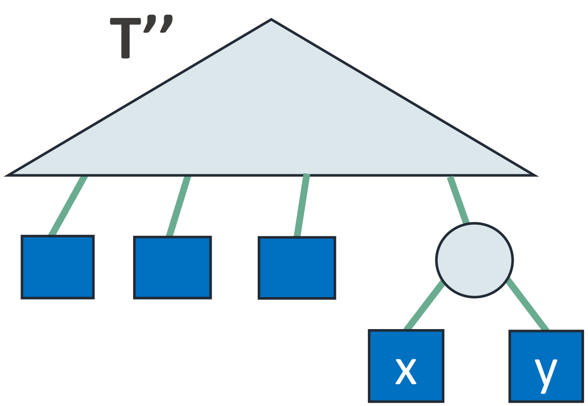


- Let's express the **Cost** relation first.
  - Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$. We have,
  - $x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1)$

    $= z.freq \cdot d_{T'}(z) + (x.freq + y.freq)$
- From which we conclude that:
  - Cost(T) = Cost(T') + x.freq + y.freq.
  - Or, equivalently,
  - Cost(T') = Cost(T) - x.freq - y.freq.

The whole argument

- Now, I claim T is not an optimal tree. (the way of contradiction.)
- Then, there exists an optimal tree T’’ such that Cost(T’’) < Cost(T).
- T’’ has x and y as siblings. (by Lemma 1)
- It looks like:



By prev. slide

Then,
Cost(T’’) = Cost(T’’- {x, y}) + x.freq + y.freq.

Cost(T’’- {x, y}) = Cost(T’’) - Cost(x) - Cost(y)

T’’ is an optimal, T is not.

< Cost(T) - Cost(x) - Cost(y)

= Cost(T’)

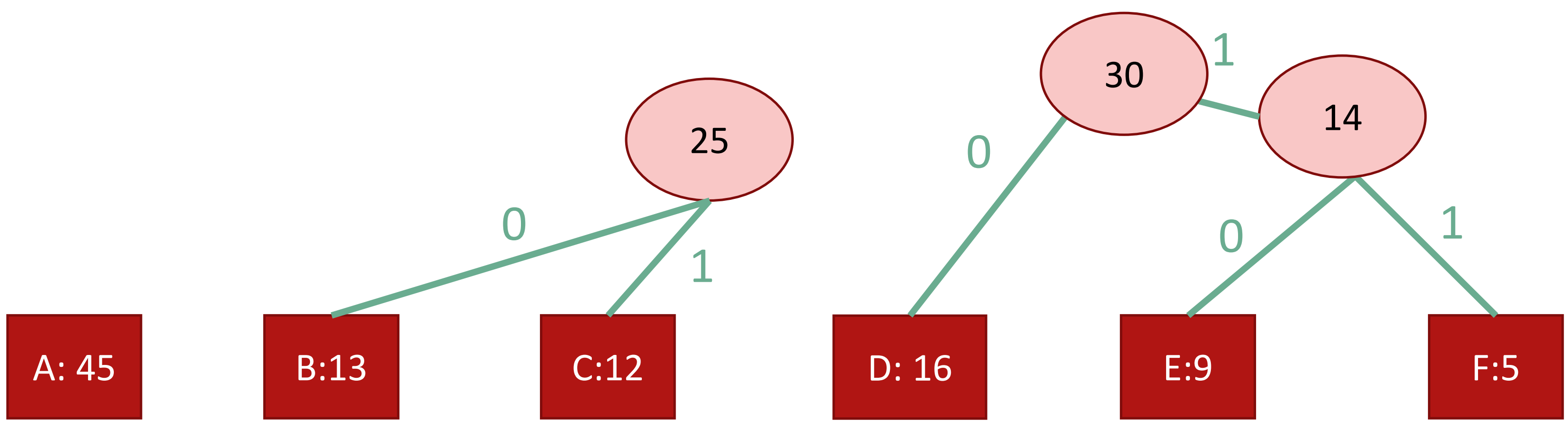
Cost(T’’- {x, y}) < Cost(T’)

CONTRADICTION!!

Yielding a contradiction to the assumption that T’ represents an optimal prefix code for C’.

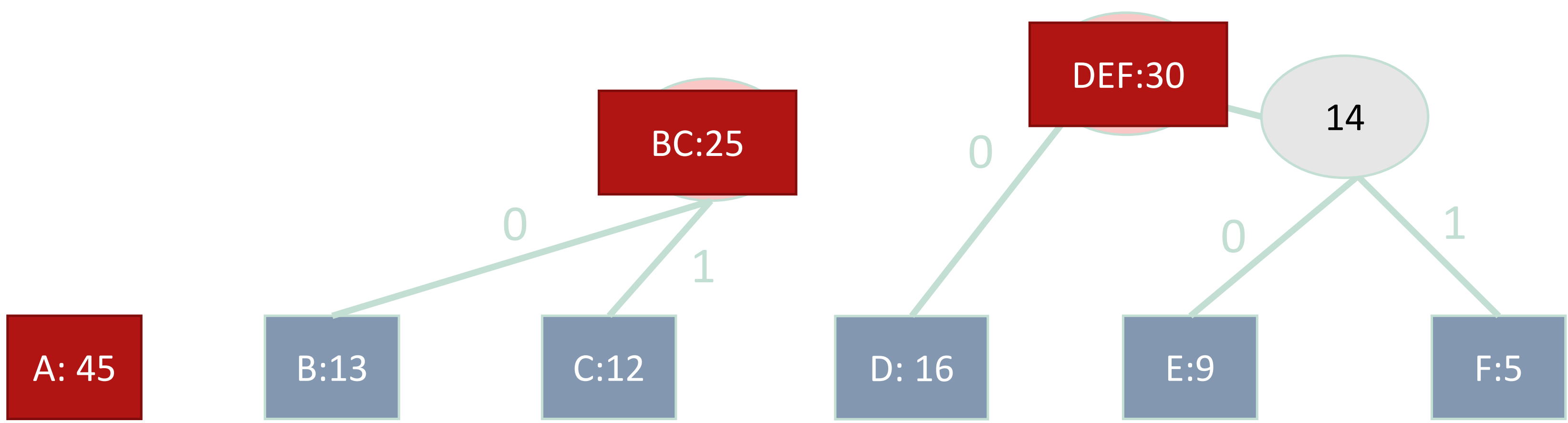
Huffman Coding Works idea

- Huffman Coding continuously grouping leaves.
- What about once we start grouping stuff?
  - We treat the “groups” as leaves in a new alphabet.



Huffman Coding Works idea

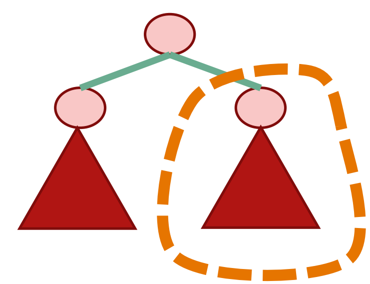
- Huffman Coding continuously grouping leaves..
- What about once we start grouping stuff?
  - We treat the "groups" as leaves in a new alphabet.
- Then we can use the lemma from before.
  - We never rule out optimality once we start grouping stuff.



What have we learned?

- ASCII isn't an optimal way* to encode English, since the distribution on letters isn't uniform. *If all we care about is number of bits.
- **Huffman Coding** is an optimal way!
- To come up with an optimal scheme for any language efficiently, we can use a greedy algorithm.

- To come up with a greedy algorithm:
  - Identify optimal substructure
  - Find a way to make choices that won't rule out an optimal solution.
    - Create subtrees out of the smallest two current subtrees.



Recap

- **Greedy algorithms!**
- Often easy to write down.
  - But may be hard to justify.
- The natural greedy algorithm may not always be correct.
- A problem is a good candidate for a greedy algorithm if:
  - it has optimal substructure
  - that optimal substructure is **REALLY NICE**
    - solutions depend on just one other sub-problem.

Any Question?