Lecture 14

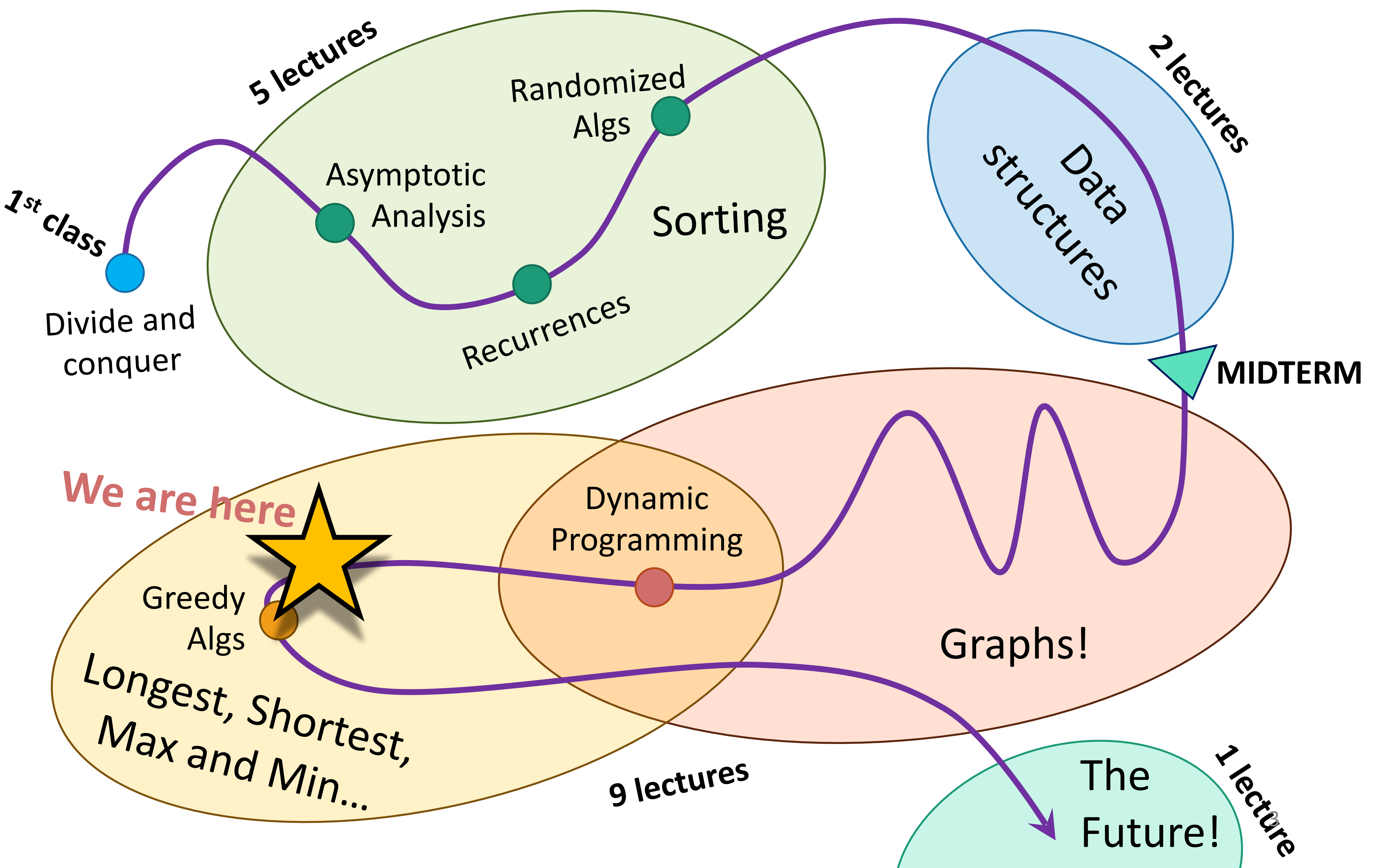
Greedy algorithms!

Announcements

- Homework 6 due today
- Homework 7 out later today
- New EthiCS videos out (Part I – 2 and Part I – 3)

More detailed schedule on the website!

Roadmap



This week

- Greedy algorithms!

Greedy algorithms

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

Today

- One example of a greedy algorithm that **does not work**:
  - Knapsack again
- Three examples of greedy algorithms that **do work**:
  - Activity Selection
  - Job Scheduling
  - Huffman Coding (if time)

You saw these on your pre-lecture exercise!

Non-example

- Unbounded Knapsack.

Capacity: 10

Item:					
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all items.
  - What's the most valuable way to fill the knapsack?

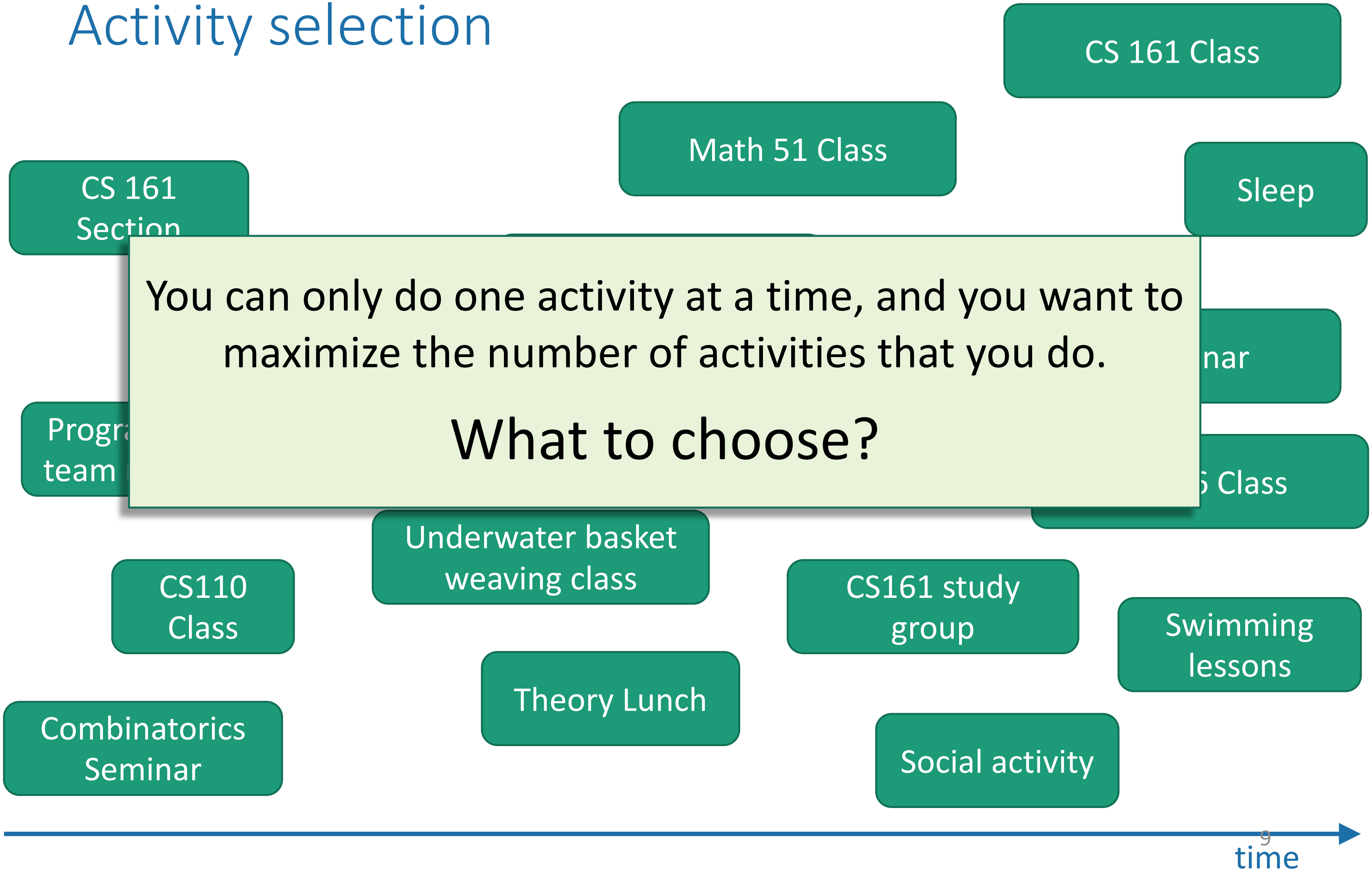
Total weight: 10
Total value: 42

- "Greedy" algorithm for unbounded knapsack:
  - Tacos have the best Value/Weight ratio!
  - Keep grabbing tacos!

Total weight: 9
Total value: 39

Example where greedy works

Activity selection

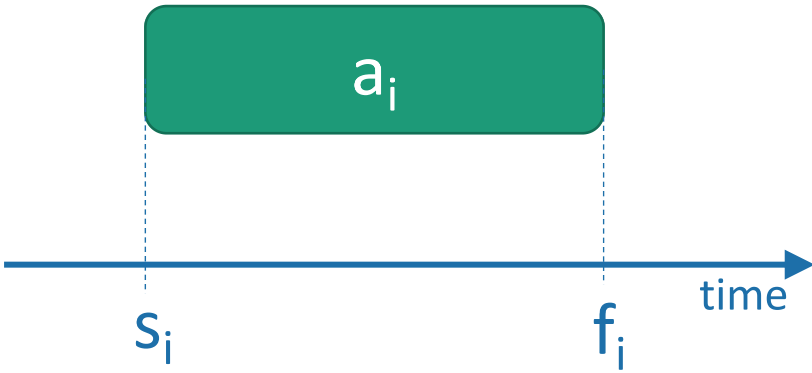


You can only do one activity at a time, and you want to maximize the number of activities that you do.

What to choose?

Activity selection

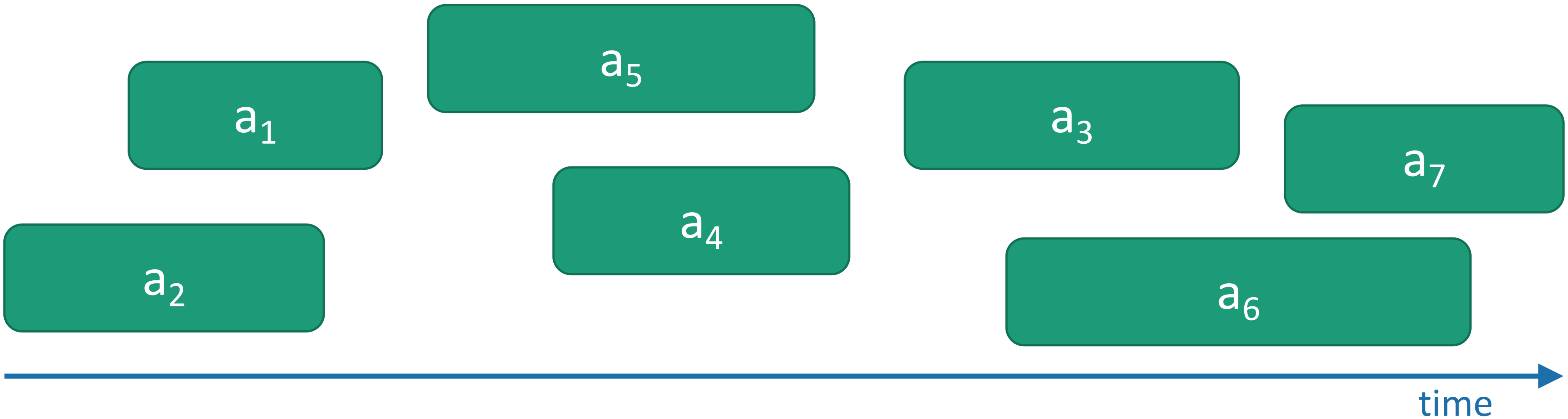
- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$



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

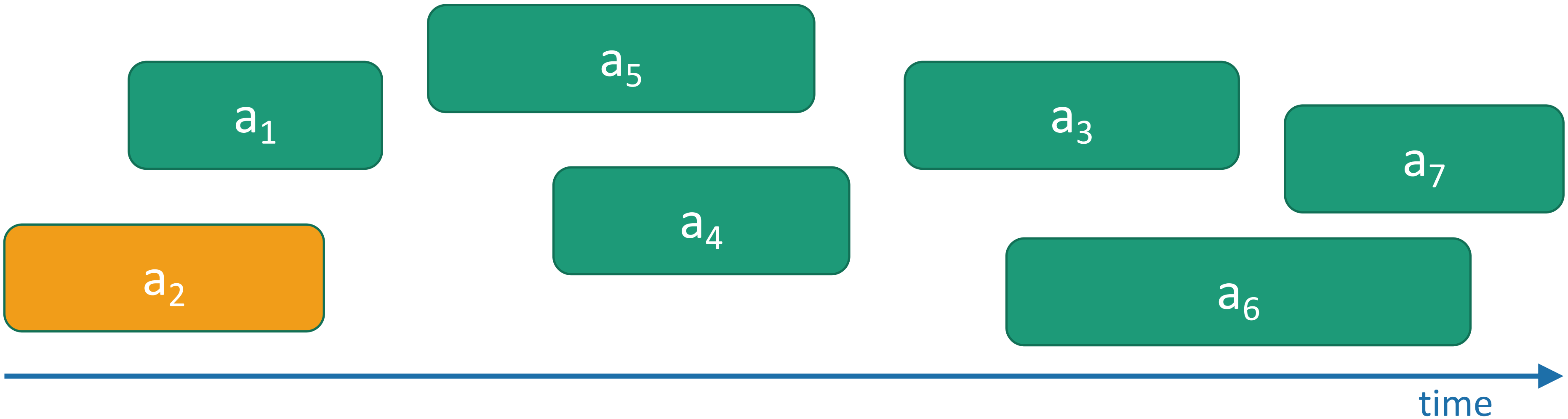
In what order should you greedily add activities?

Greedy Algorithm



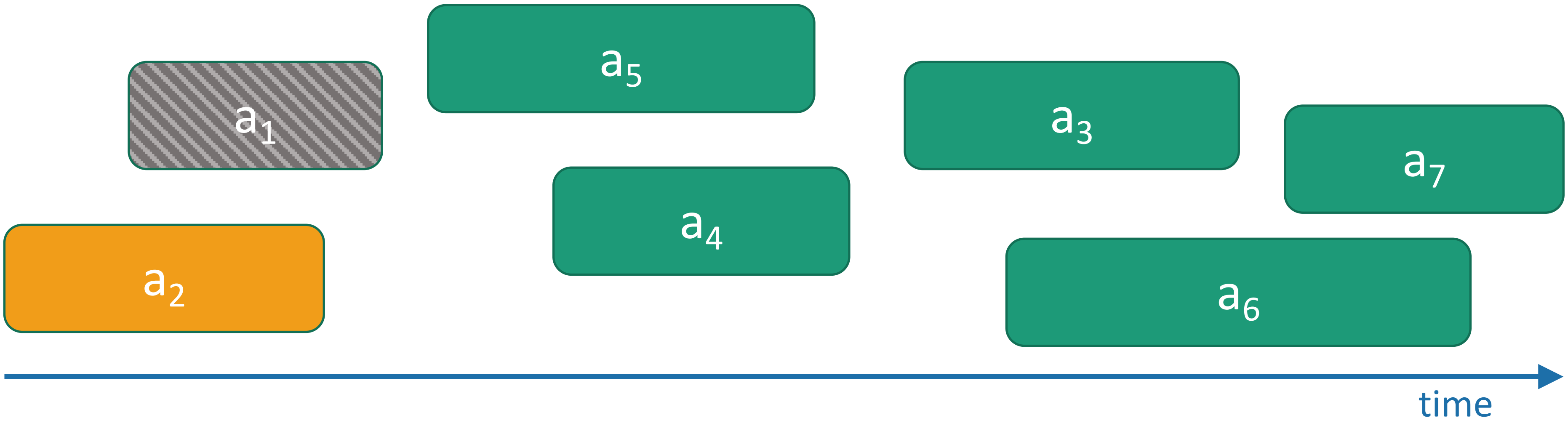
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



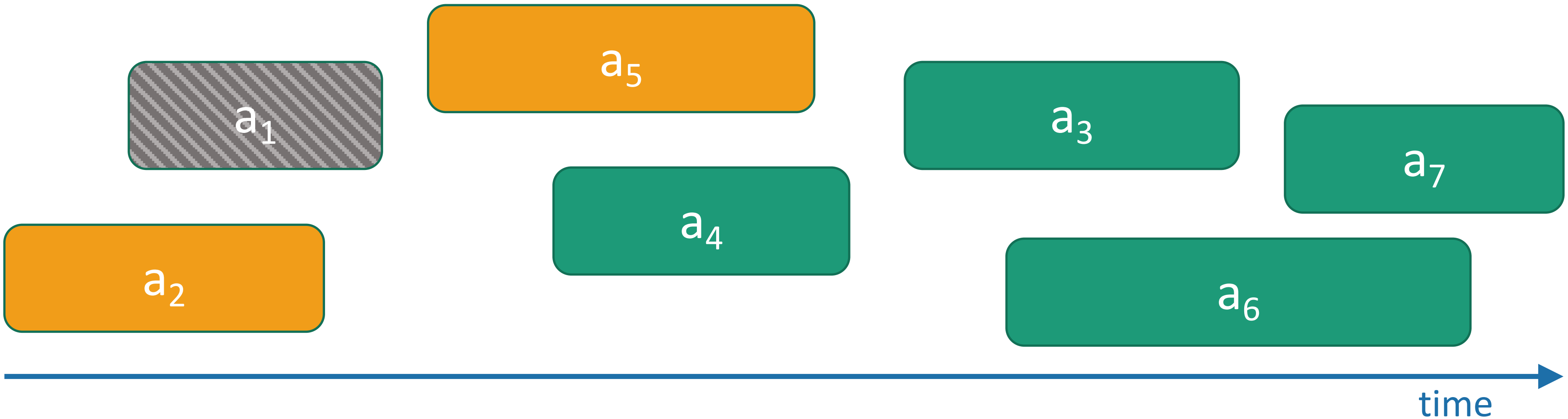
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



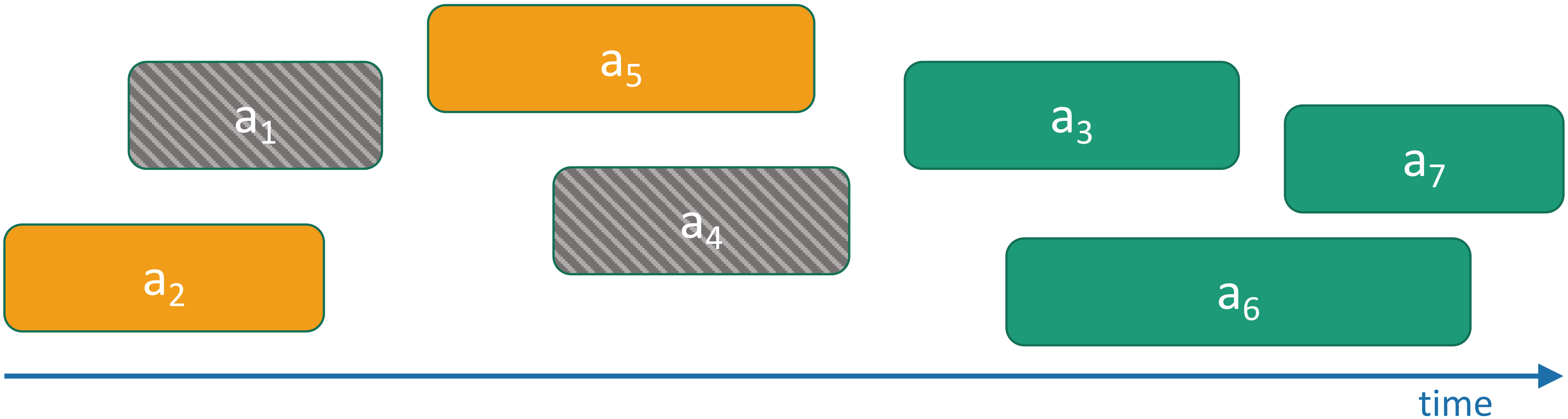
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



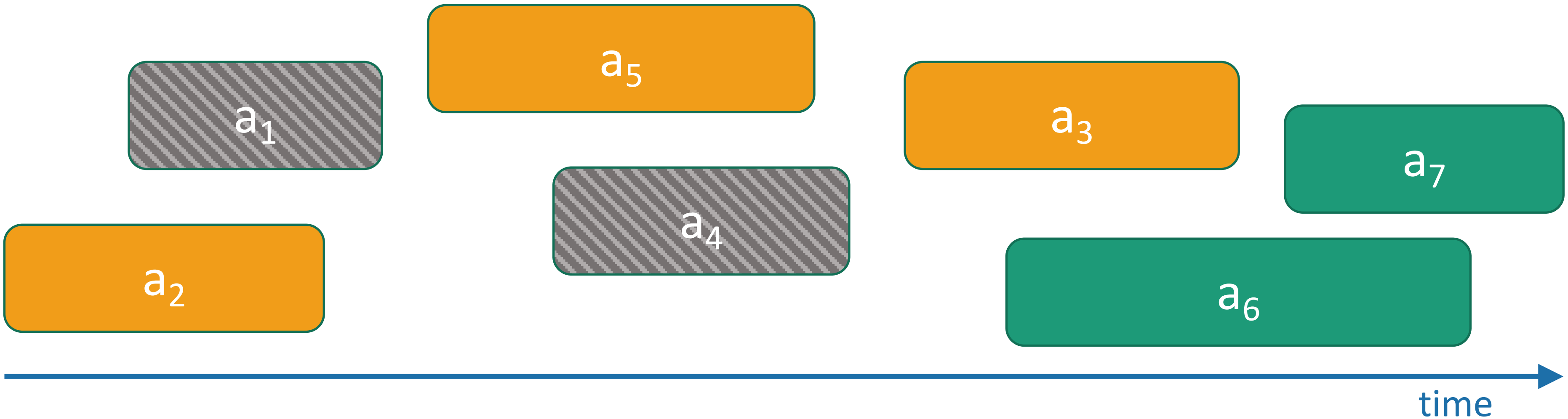
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



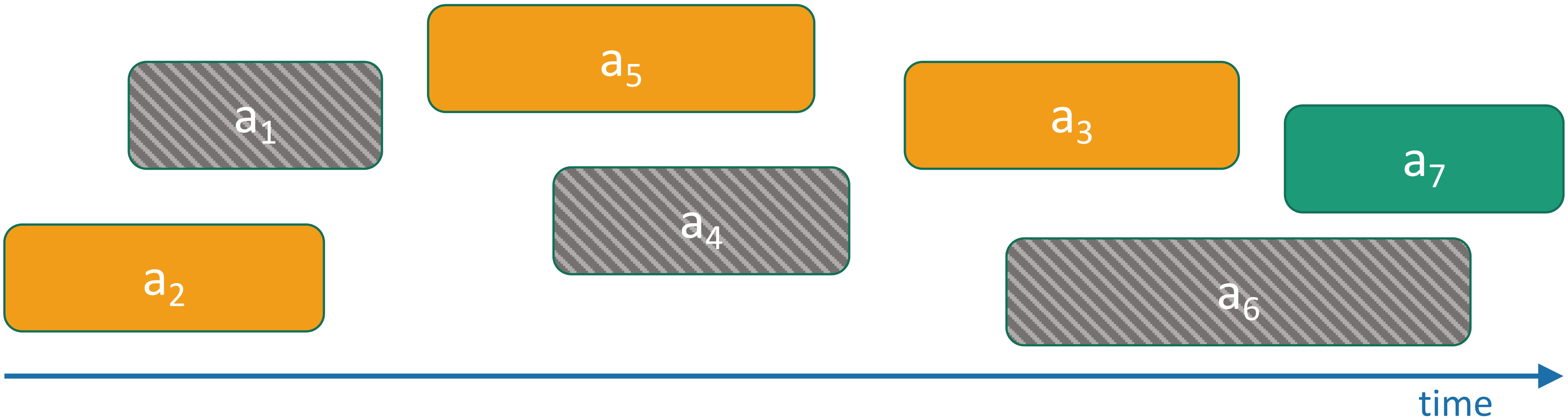
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



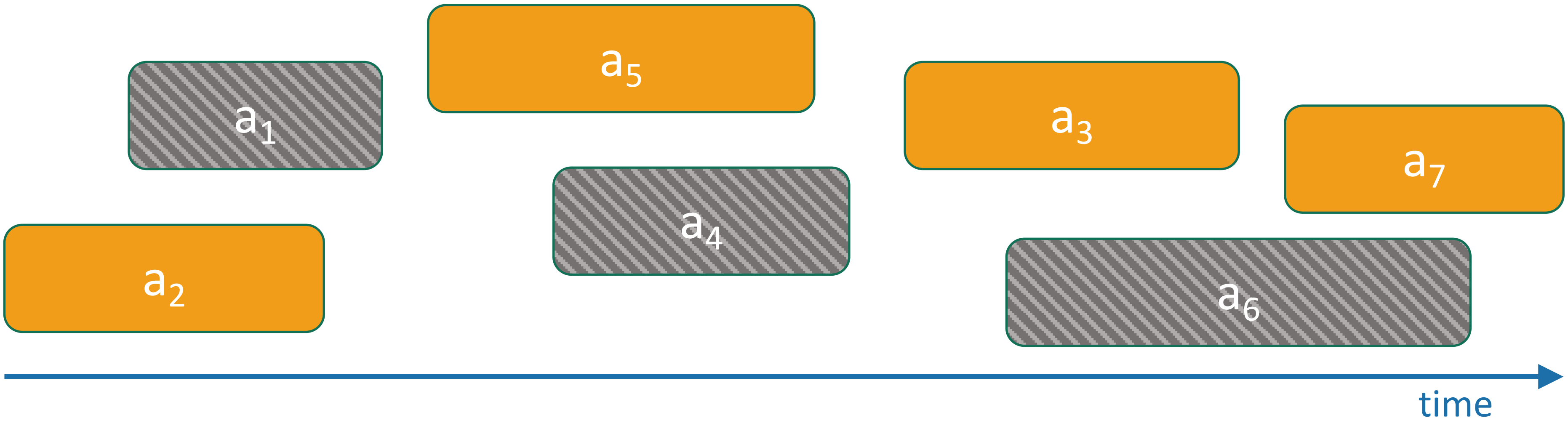
- Pick activity you can add with the smallest finish time.
- Repeat.

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?

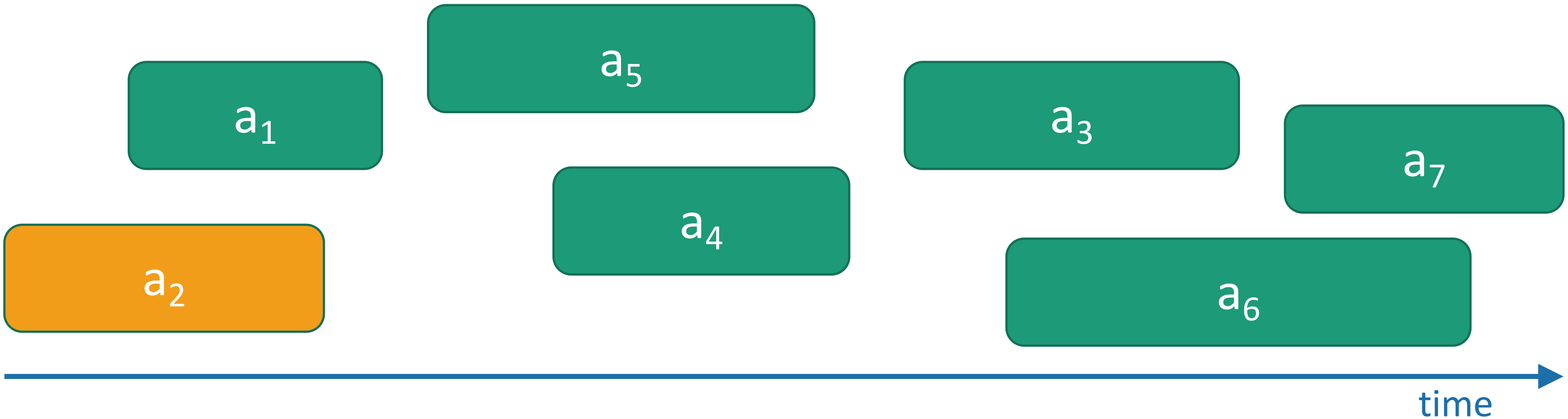
- At each step in the algorithm, make a choice.
  - Hey, I can increase my activity set by one,
  - And leave lots of room for future choices,
  - Let's do that and hope for the best!!!

- **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?
   - 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 8?
     - Proving that greedy algorithms work is often not so easy…

Back to Activity Selection



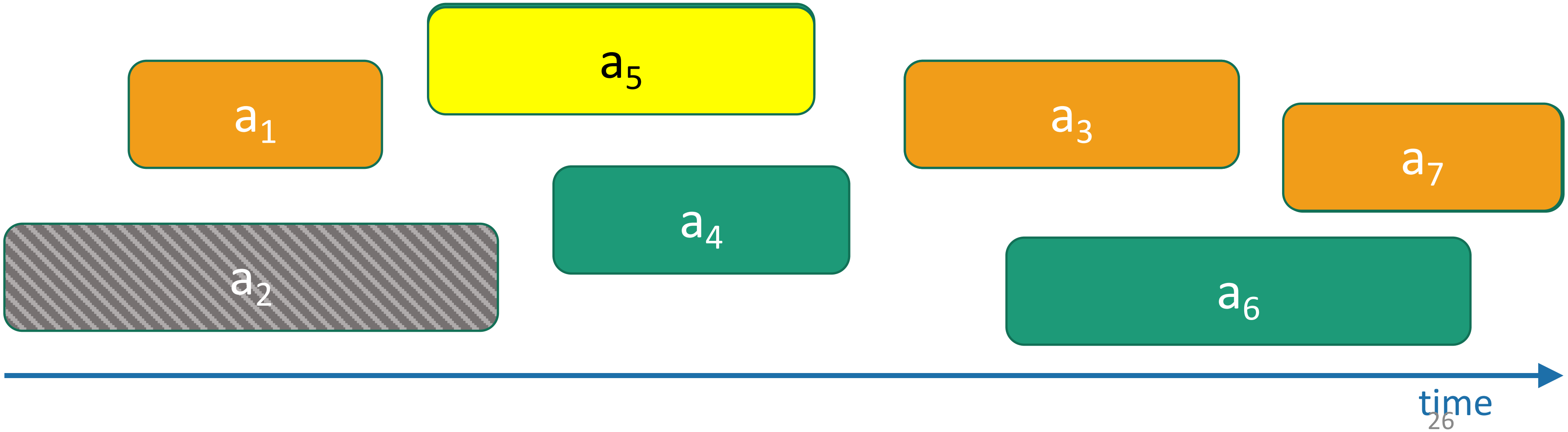
- Pick activity you can add with the smallest finish time.
- Repeat.

Why does it work?

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

Our next choice would be this one:

There's some optimal solution that contains our next choice



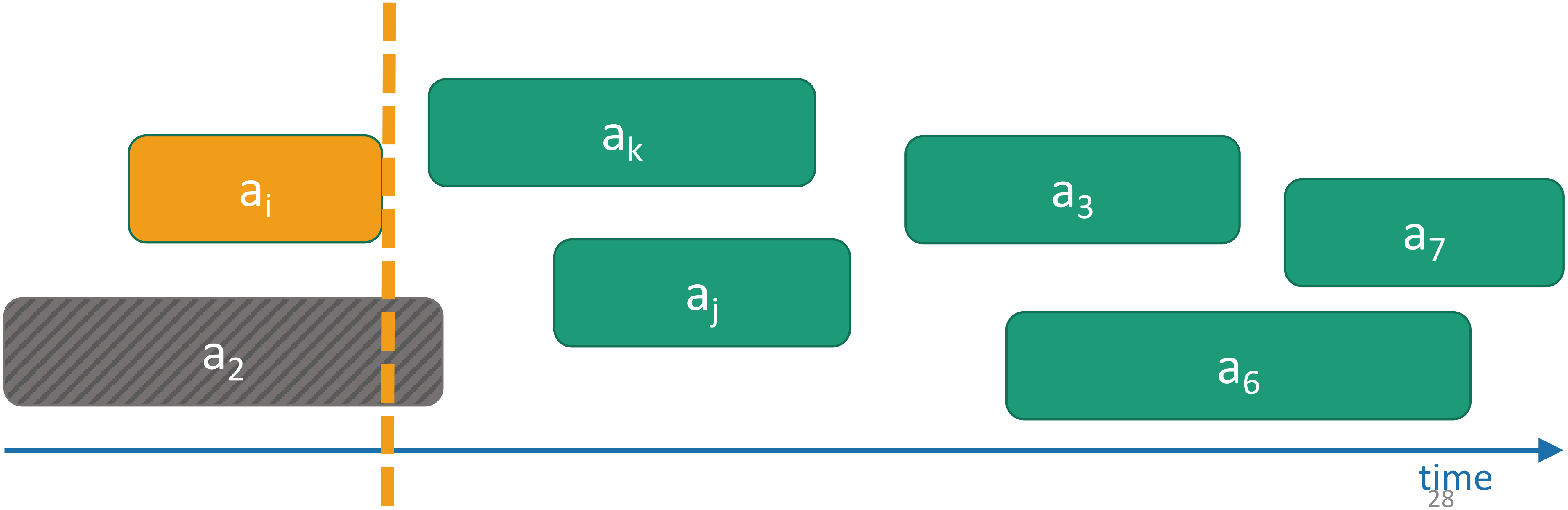
Assuming that statement…

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

Lucky the Lackadaisical Lemur

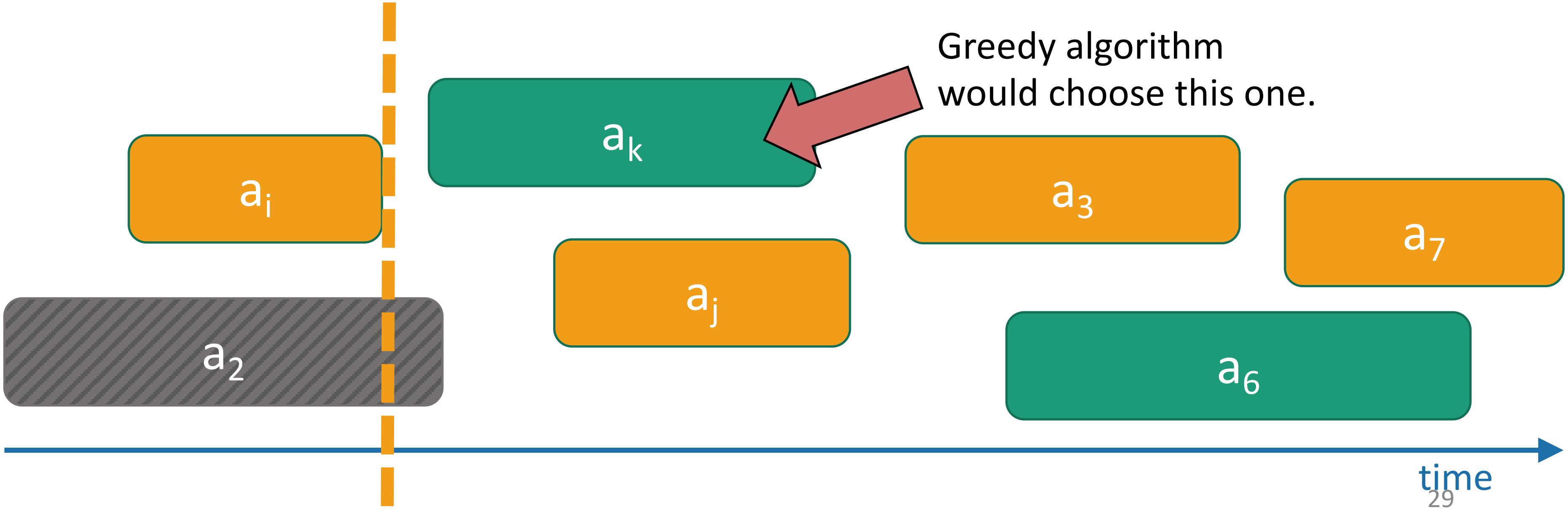
We never rule out an optimal solution

- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.



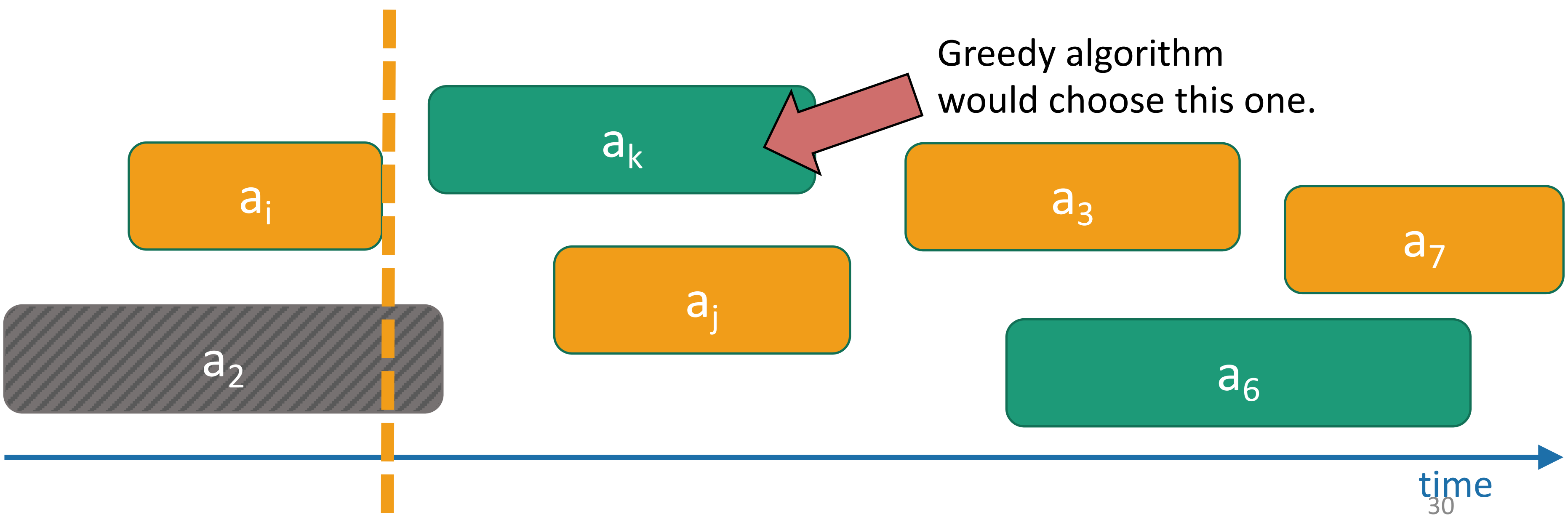
We never rule out an optimal solution

- 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.



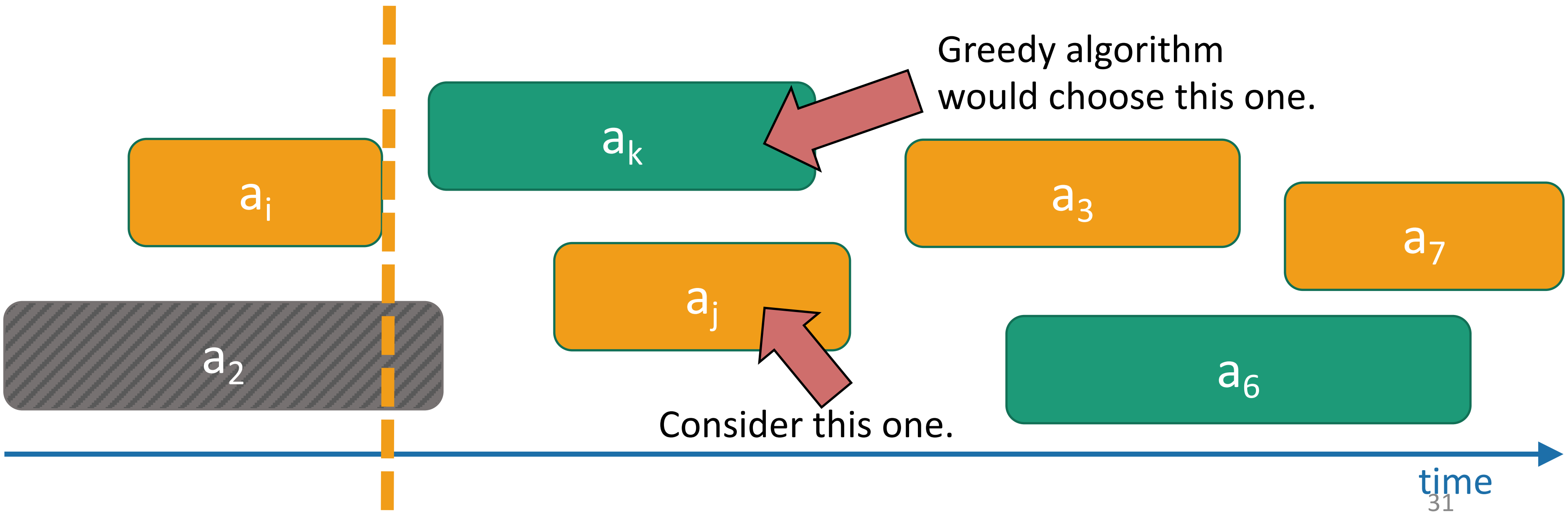
We never rule out an optimal solution

- 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*…



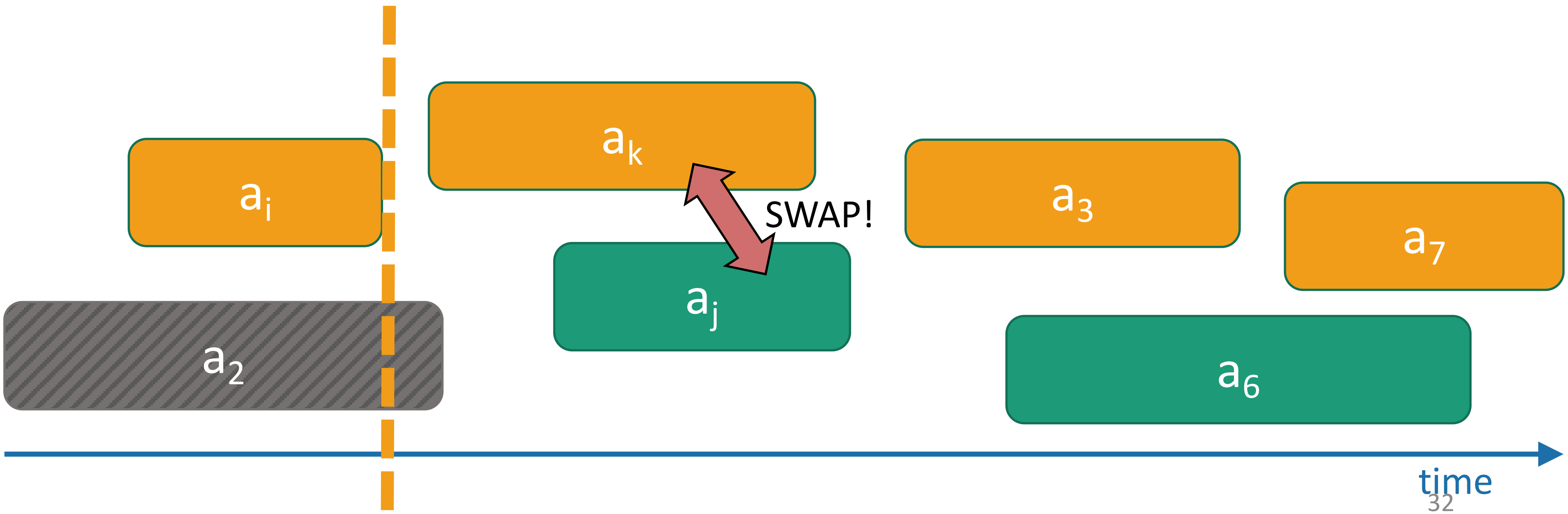
We never rule out an optimal solution ctd.

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



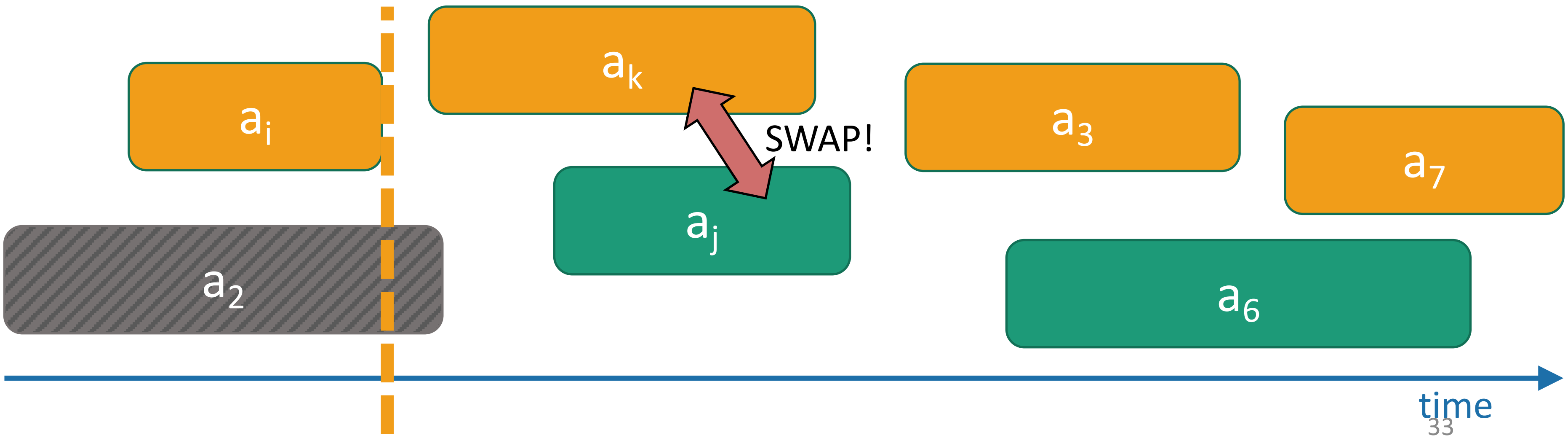
We never rule out an optimal solution
ctd.

- 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



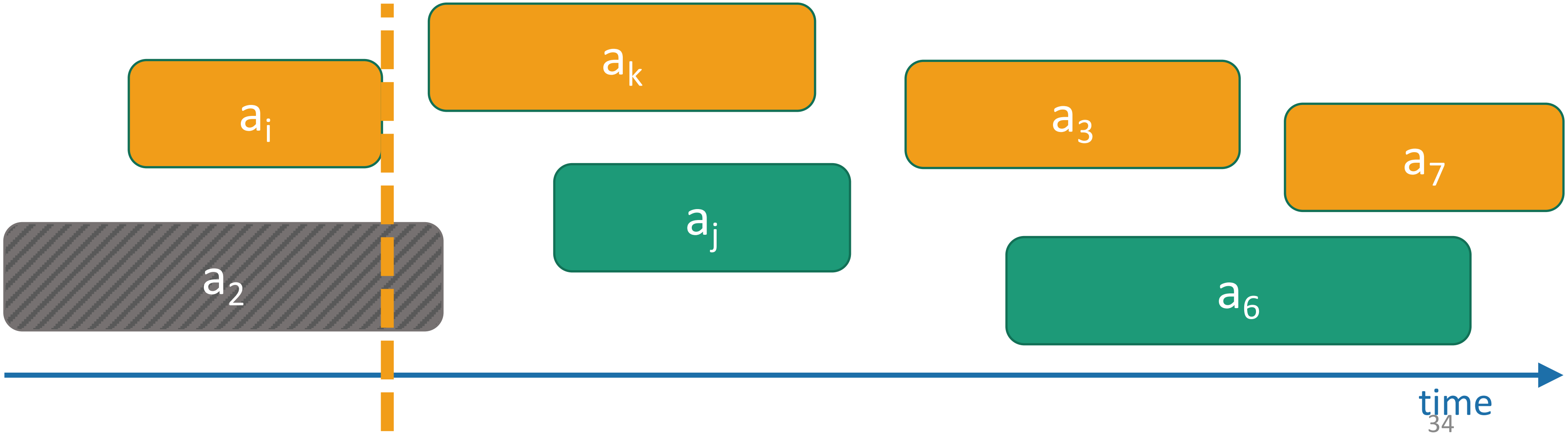
We never rule out an optimal solution
ctd.

- 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*.



We never rule out an optimal solution ctd.

- We've just shown:
  - If there was an optimal solution that extends the choices we made so far…
  - …then there is an optimal schedule that also contains our next greedy choice a_k.



So the algorithm is correct

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

Lucky the Lackadaisical Lemur

So the algorithm is correct

Plucky the Pedantic Penguin

- 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.
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 8?
     - Proving that greedy algorithms work is often not so easy…

One Common strategy
for greedy algorithms

- 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.**

One Common strategy (formally)

for greedy algorithms

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

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

One Common strategy

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* *disagrees* with your next greedy choice.
  - E.g., it *doesn't* 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.

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.

Three Questions

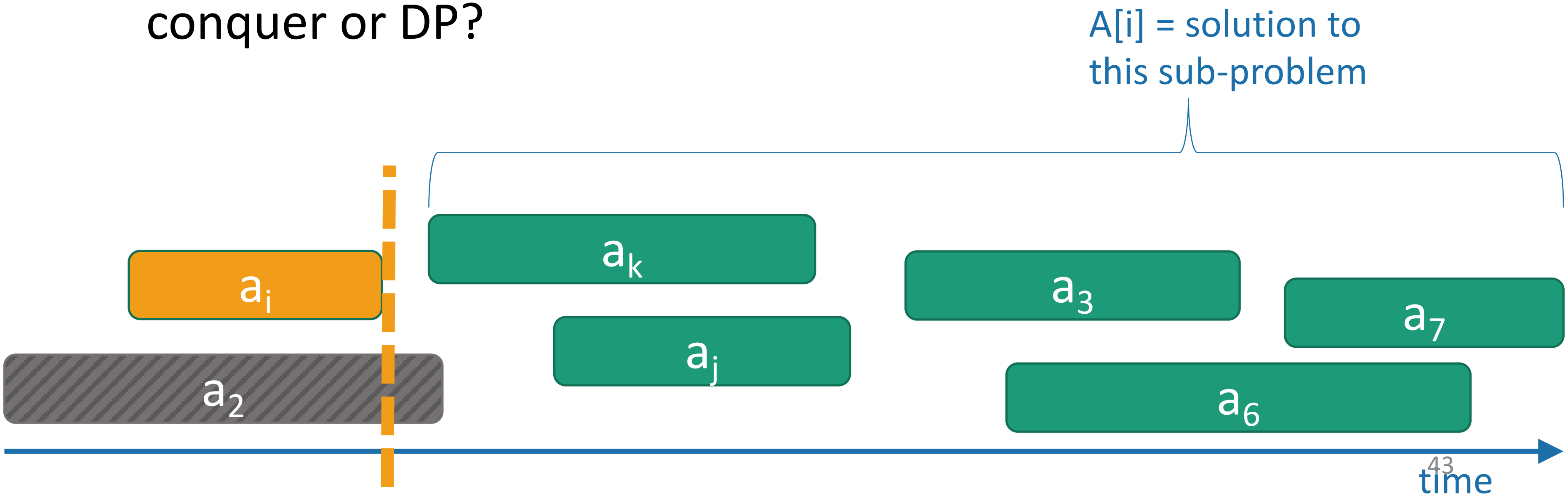
1. Does this greedy algorithm for activity selection work?
   - Yes.
2. In general, when are greedy algorithms a good idea?
   - When the problem exhibits especially nice optimal substructure.
3. The "greedy" approach is often the first you'd think of…
   - Why are we getting to it now, in Week 8?
     - Proving that greedy algorithms work is often not so easy…

Optimal sub-structure
in greedy algorithms

- Our greedy activity selection algorithm exploited a natural sub-problem structure:

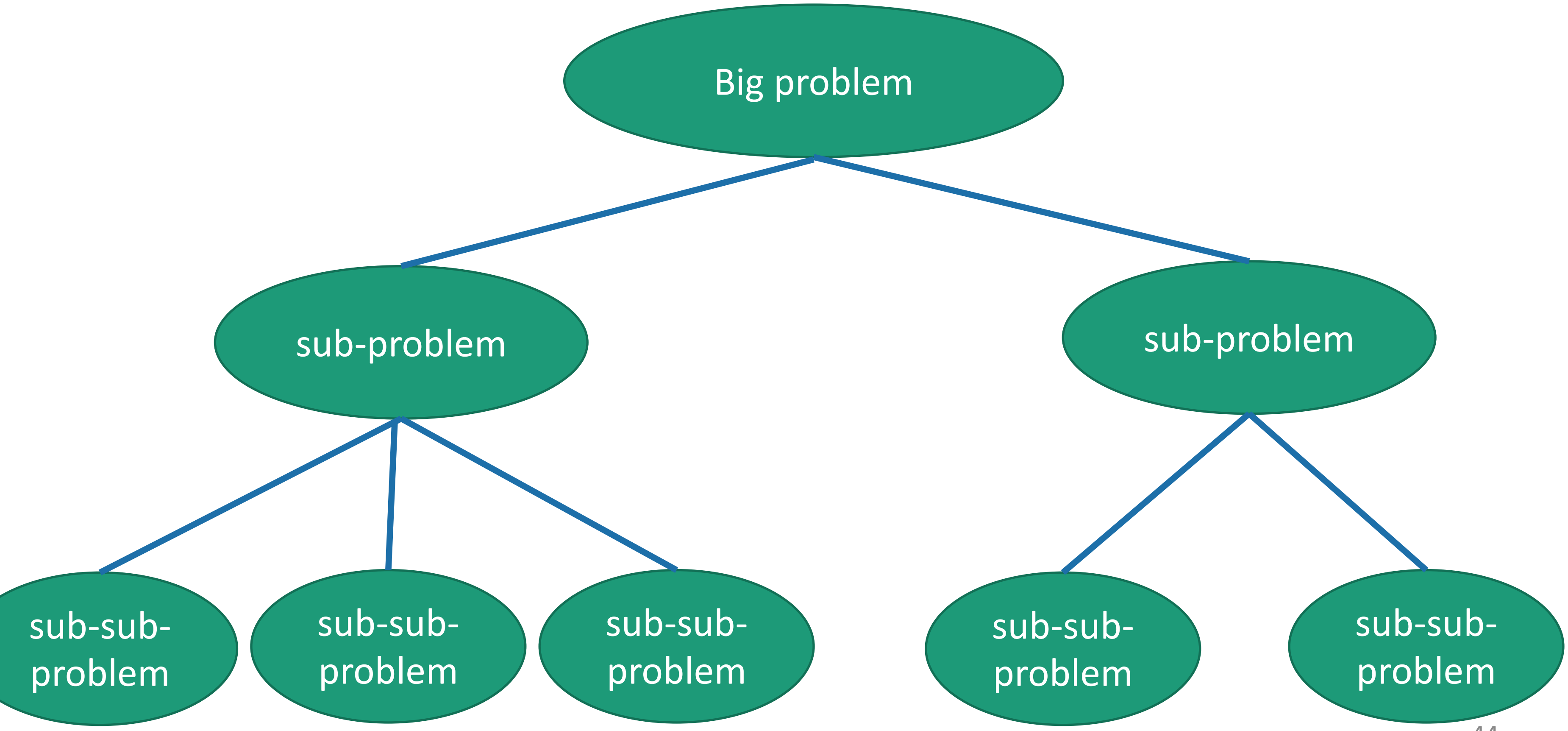
  A[i] = number of activities you can do after the end of activity i

- How does this substructure relate to that of divide-and-conquer or DP?



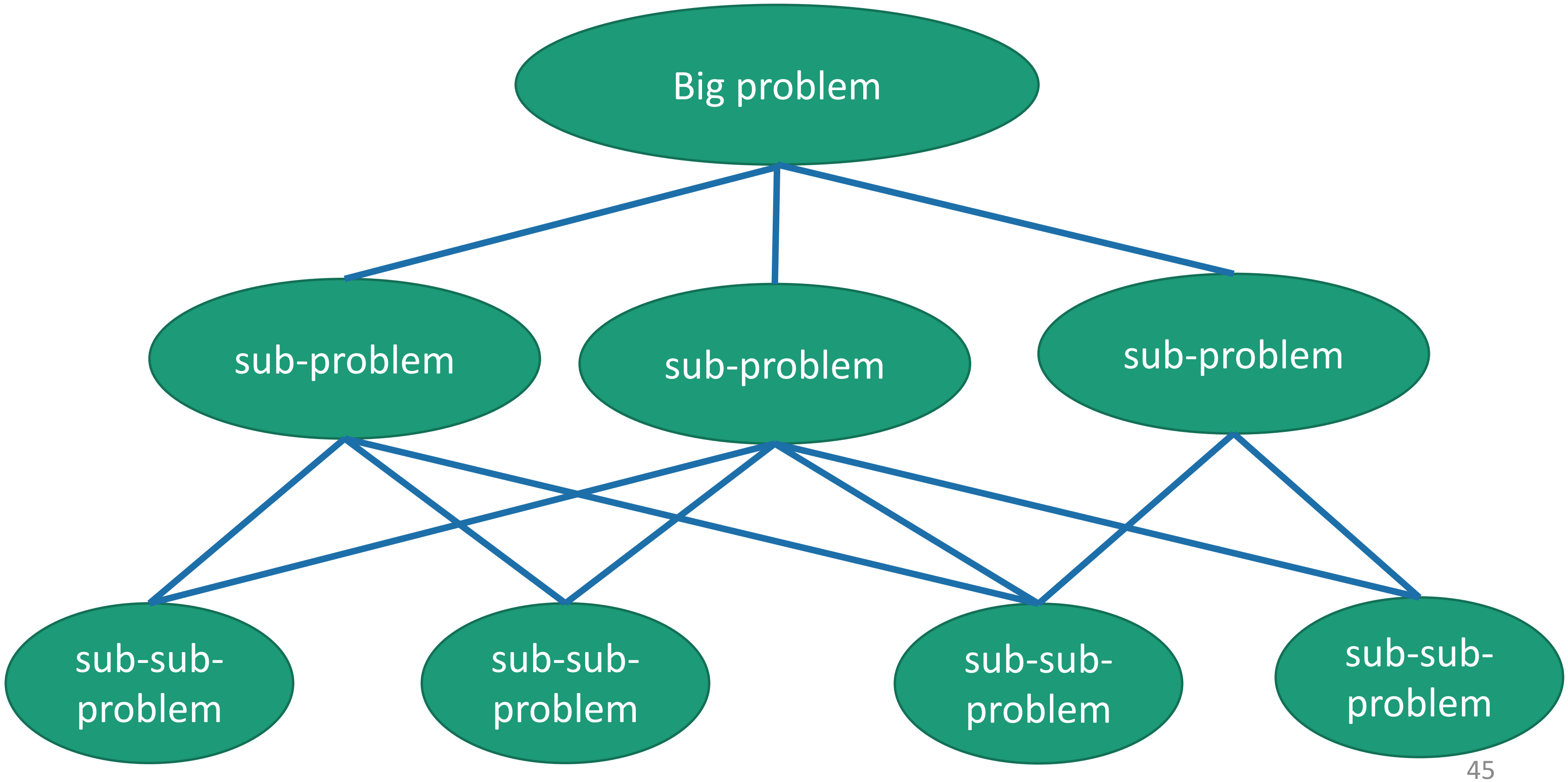
Sub-problem graph view

- Divide-and-conquer:



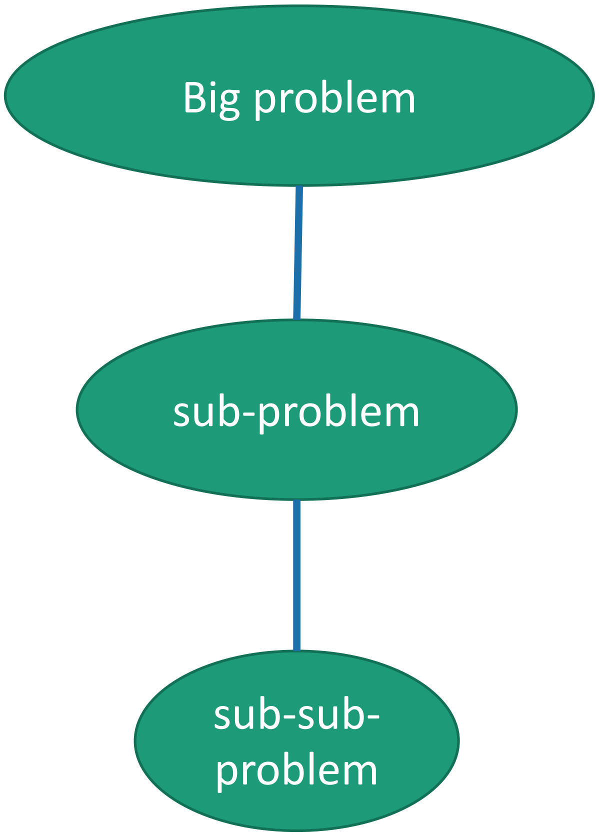
Sub-problem graph view

- Dynamic Programming:



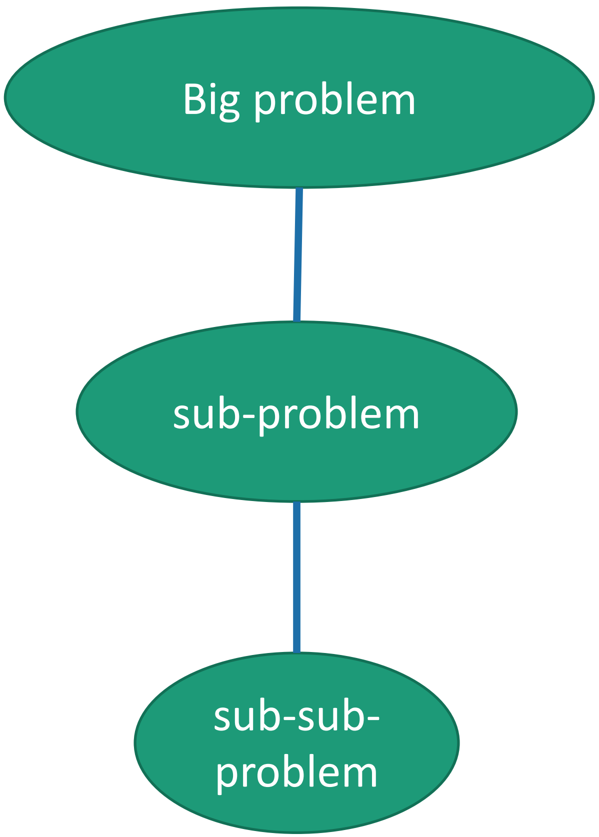
Sub-problem graph view

- Greedy algorithms:



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**.

Write a DP version of activity selection (where you fill in a table)! [See hidden slides in the .pptx file for one way]

Ollie the Over-achieving Ostrich

Three Questions

1. Does this greedy algorithm for activity selection work?
   - Yes.
2. In general, when are greedy algorithms a good idea?
   - When they exhibit 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 8?
     - Proving that greedy algorithms work is often not so easy.

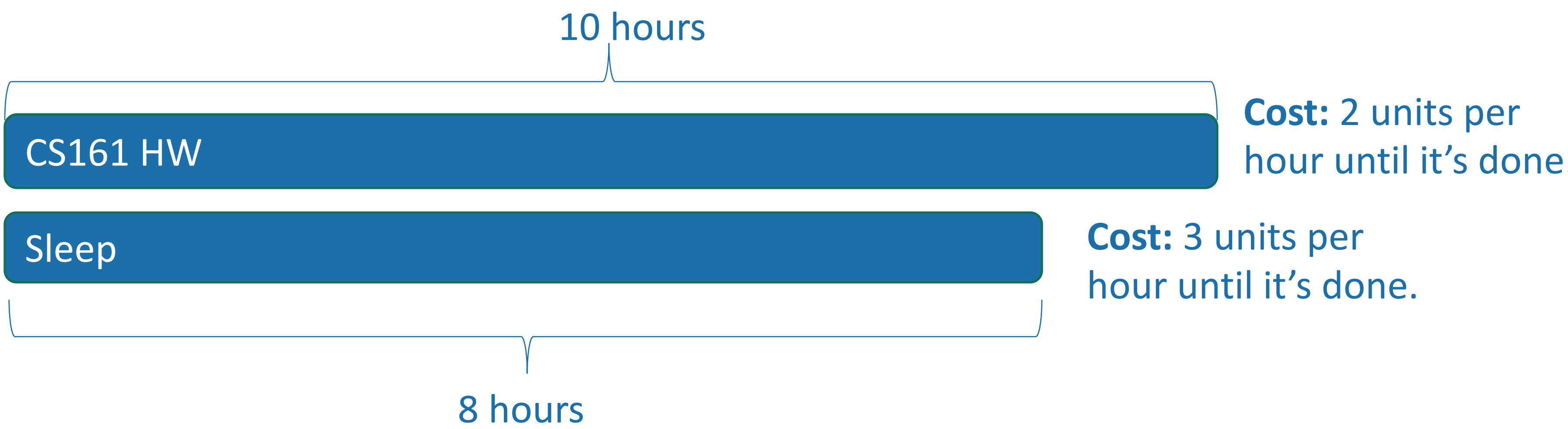
Let's see a few more examples

Another example: Scheduling

- CS161 HW
- Personal hygiene
- Math HW
- Administrative stuff for student club
- Econ HW
- Do laundry
- Meditate
- Practice musical instrument
- Read lecture notes
- Have a social life
- Sleep

Scheduling

- n tasks
- Task i takes t_i hours
- For every hour that passes until task i is done, pay c_i

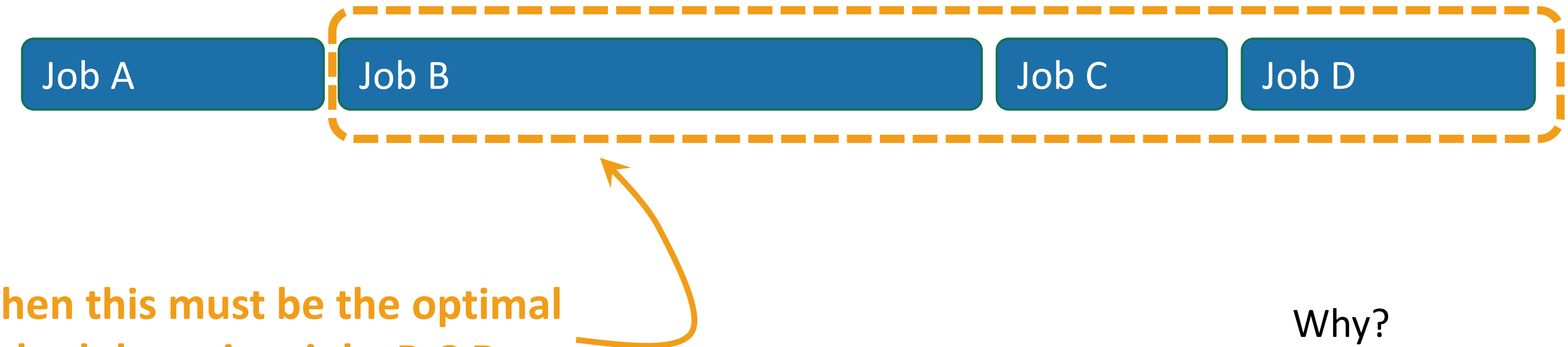


- CS161 HW, then Sleep: costs $\mathbf{10 \cdot 2 + (10 + 8) \cdot 3 = 74}$ **units**
- Sleep, then CS161 HW: costs $\mathbf{8 \cdot 3 + (10 + 8) \cdot 2 = 60}$ **units**

Optimal substructure

- This problem breaks up nicely into sub-problems:

Suppose this is the optimal schedule:



Then this must be the optimal schedule on just jobs B,C,D.

Why?

Optimal substructure

- This problem breaks up nicely into sub-problems:

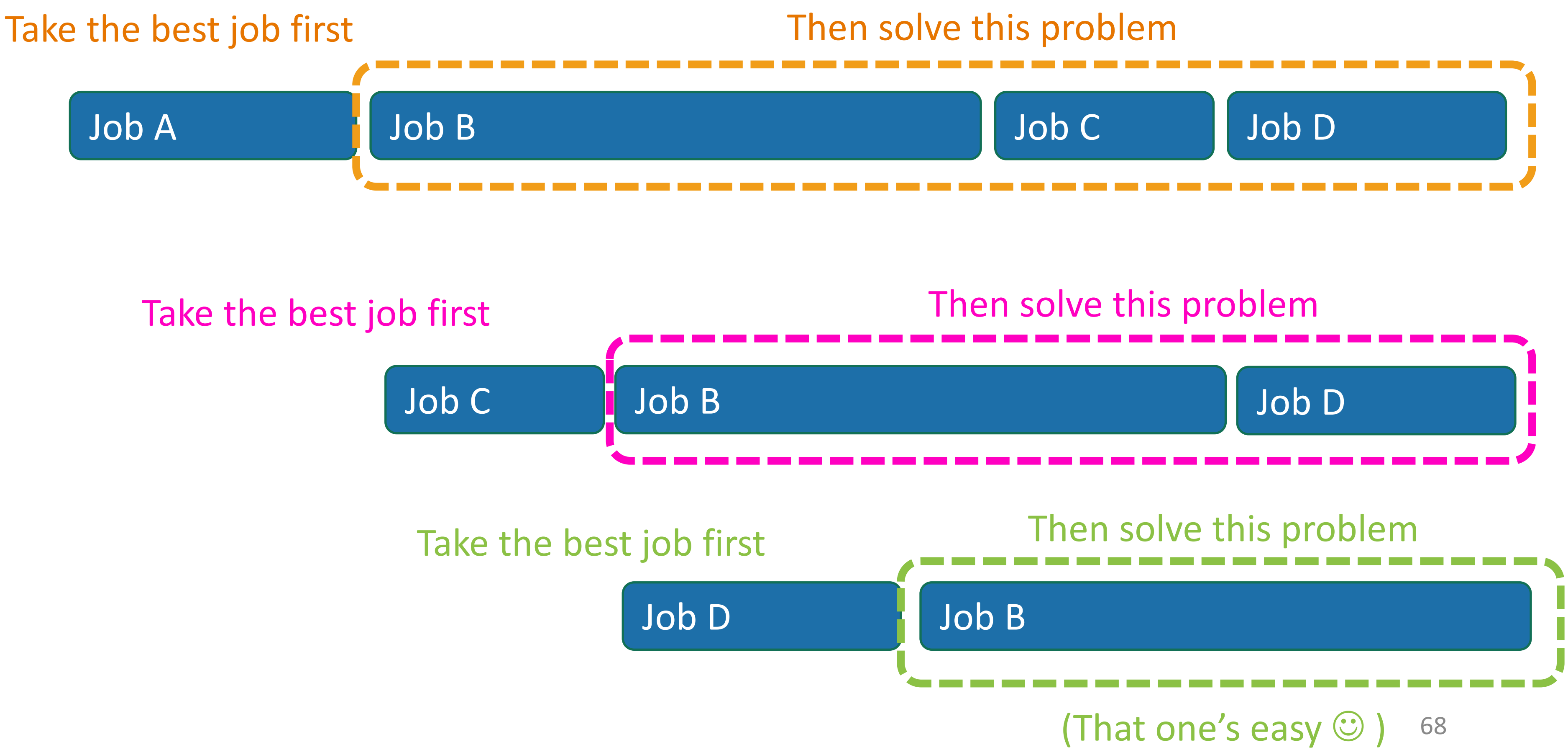
Suppose this is the optimal schedule:

Then this must be the optimal schedule on just jobs B,C,D.

If not, then rearranging B,C,D could make a better schedule than (A,B,C,D)!

Optimal substructure

- Seems amenable to a greedy algorithm:



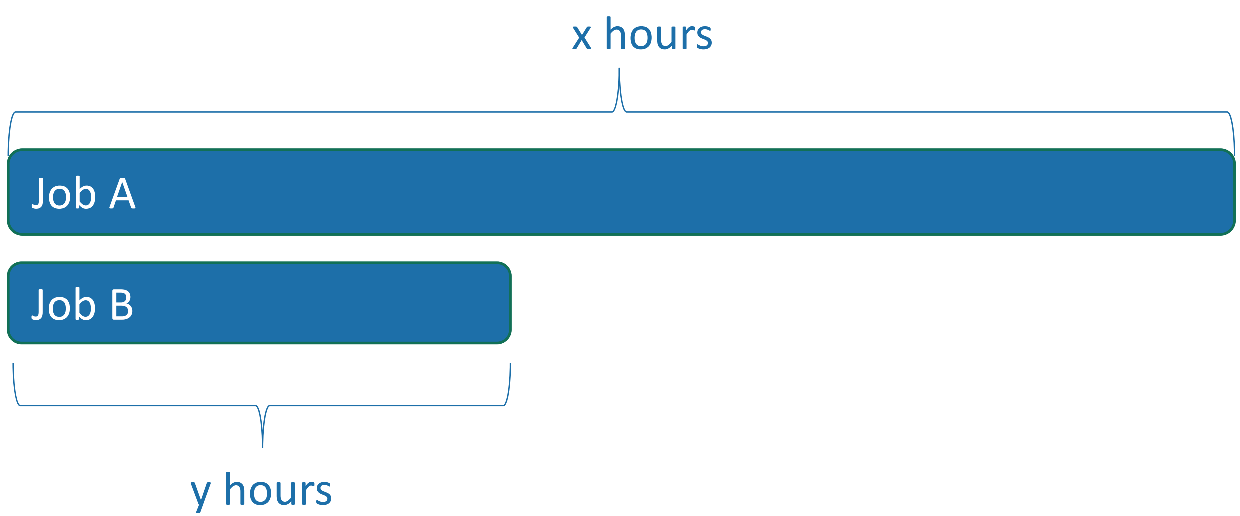
What does "best" mean?

Note: here we are defining x, y, z, and w. (We use c_i and t_i for these in the general problem, but we are changing notation for just this thought experiment to save on subscripts.)

AB is better than **BA** when:

$$xz + (x+y)w \leq yw + (x+y)z$$
$$xz + xw + yw \leq yw + xz + yz$$
$$wx \leq yz$$
$$\frac{w}{y} \leq \frac{z}{x}$$

- Of these two jobs, which should we do first?



Cost: z units per hour until it's done.

Cost: w units per hour until it's done.

- Cost(**A then B**) = x · z + (x + y) · w
- Cost(**B then A**) = y · w + (x + y) · z

What matters is the ratio:

$$\frac{\textbf{cost of delay}}{\textbf{time it takes}}$$

"Best" means biggest ratio.

Idea for greedy algorithm

- Choose the job with the biggest $\frac{\textbf{cost of delay}}{\textbf{time it takes}}$ ratio.

Lemma

This greedy choice doesn't rule out success

- Suppose you have already chosen some jobs, and haven't yet ruled out success:

Already chosen E

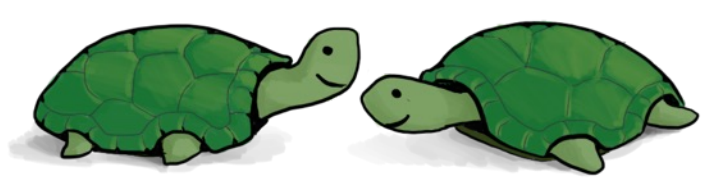
There's some way to order A, B,C, D that's optimal…

Job E | Job C | Job A | Job B | Job D

Say greedy chooses job B

- Then if you choose the next job to be the one left that maximizes the ratio **cost/time**, you still won't rule out success.
- **Proof sketch:**
  - Say Job B maximizes this ratio, but it's not the next job in the opt. soln.

How can we manipulate the optimal solution above to make an optimal solution where B is the next job we choose after E?
1 minute think; (wait) 1 minute share



Lemma

This greedy choice doesn't rule out success

- Suppose you have already chosen some jobs, and haven't yet ruled out success:

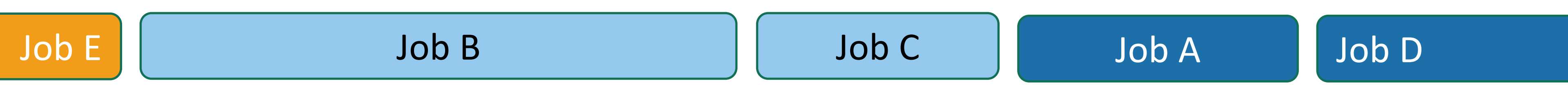
Already chosen E

There's some way to order A, B,C, D that's optimal…

Say greedy chooses job B

- Then if you choose the next job to be the one left that maximizes the ratio **cost/time**, you still won't rule out success.
- **Proof sketch:**
  - Say Job B maximizes this ratio, but it's not the next job in the opt. soln.
  - Switch A and B! Nothing else will change, and we just showed that the cost of the solution won't increase.

  - Repeat until B is first.



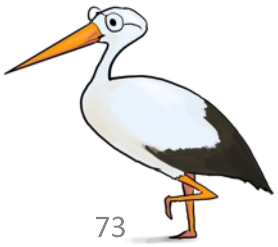
  - Now this is an optimal schedule where B is first.

Back to our framework for proving correctness of greedy algorithms

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

Just did the inductive step!

Fill in the details!



Greedy Scheduling Solution

- **scheduleJobs**(JOBS):
  - Sort JOBS in decreasing order by the ratio:
    - $r_i = \frac{c_i}{t_i} = \frac{\text{cost of delaying job i}}{\text{time job i takes to complete}}$
  - **Return** JOBS

Running time: O(n log(n))

Now you can go about your schedule peacefully, in the optimal way.

What have we learned?

- A greedy algorithm works for scheduling

- This followed the same outline as the previous example:
  - Identify **optimal substructure:**

  - Find a way to make choices that **won't rule out an optimal solution.**
    - largest cost/time ratios first.

One more example
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

One more example

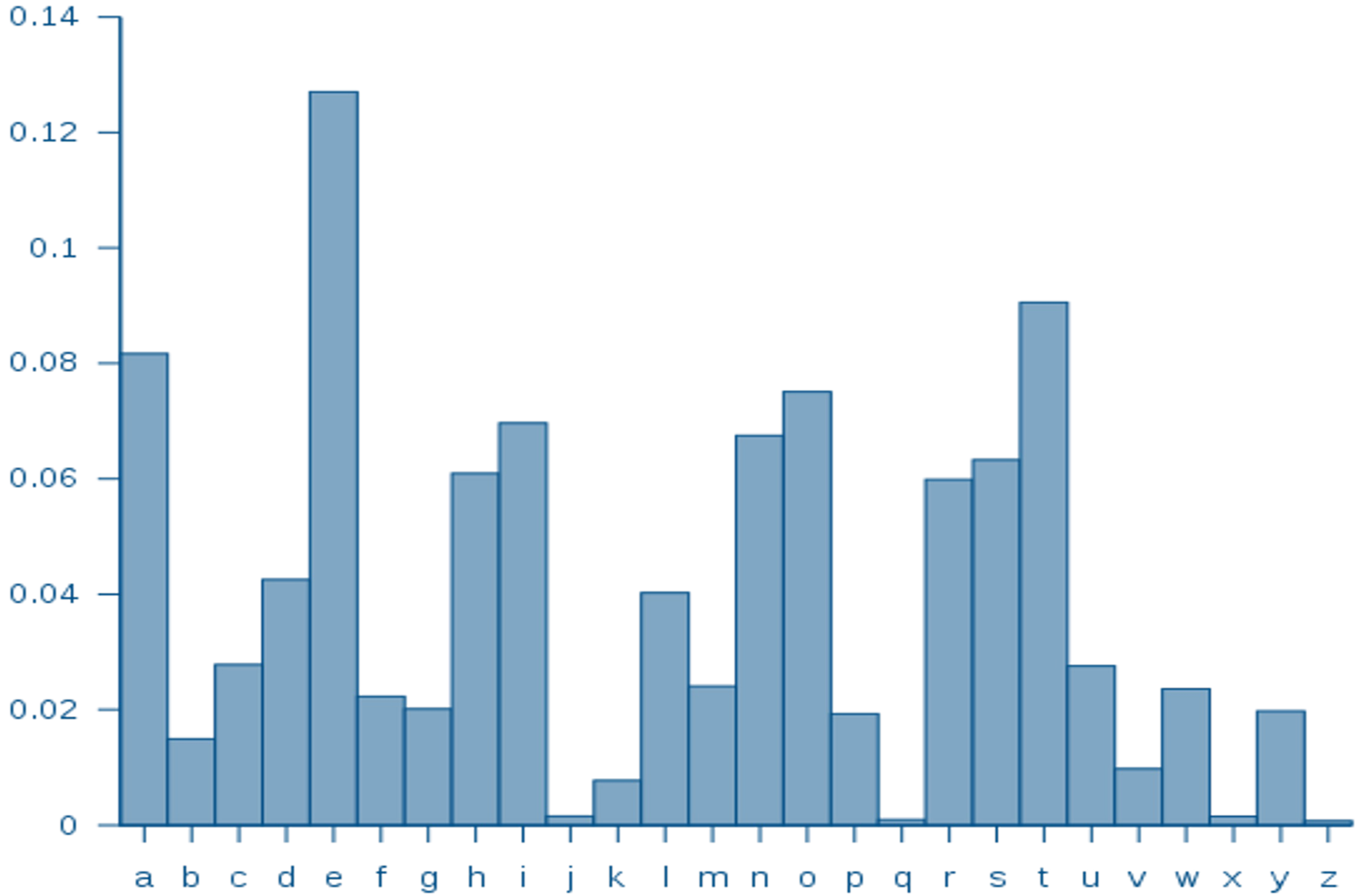
Huffman coding

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

- **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**

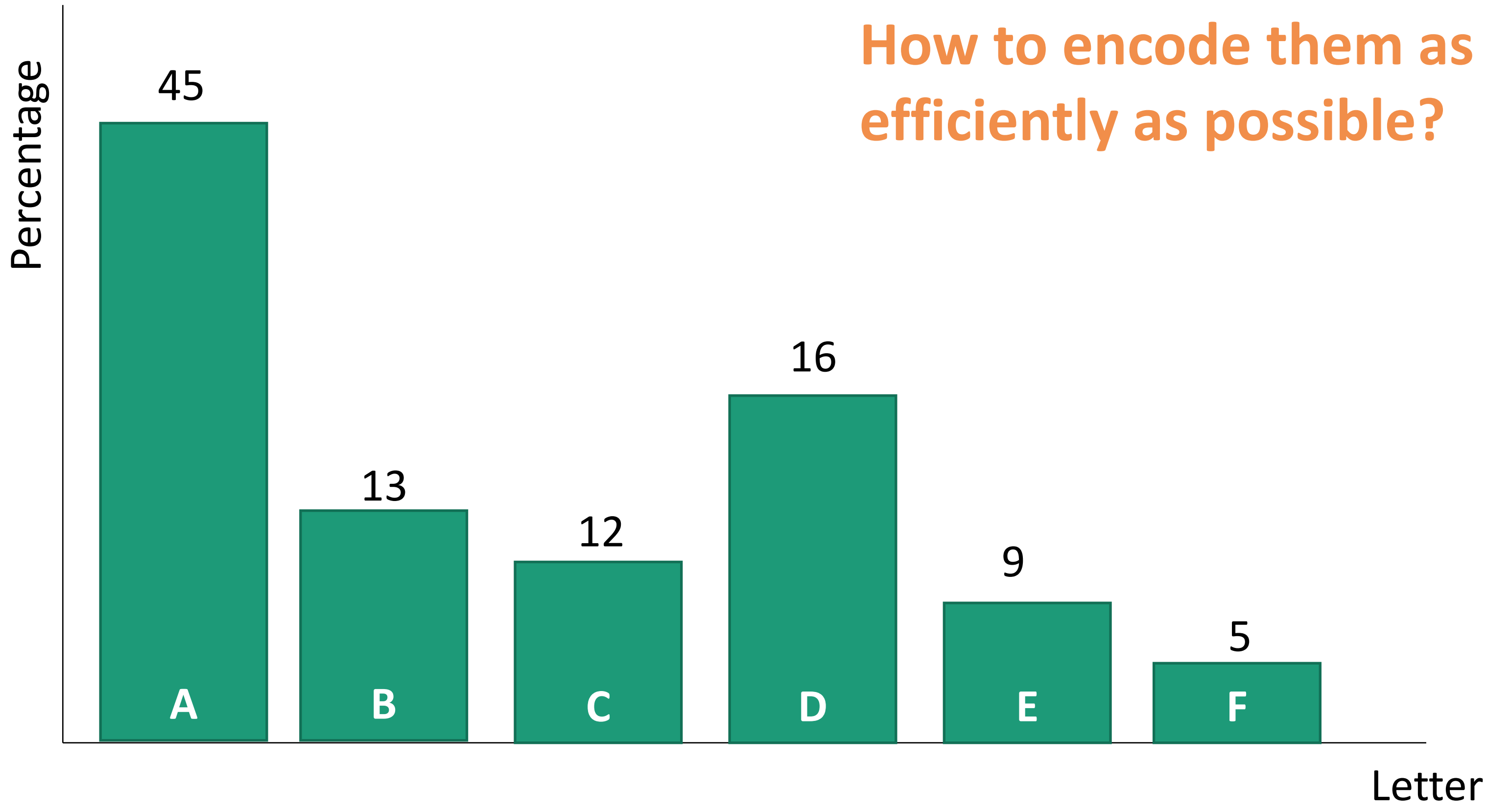
- qwertyui_opasdfg+hjklzxcv
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Suppose we have some distribution on characters



Suppose we have some distribution on characters

For simplicity, let's go with this made-up example



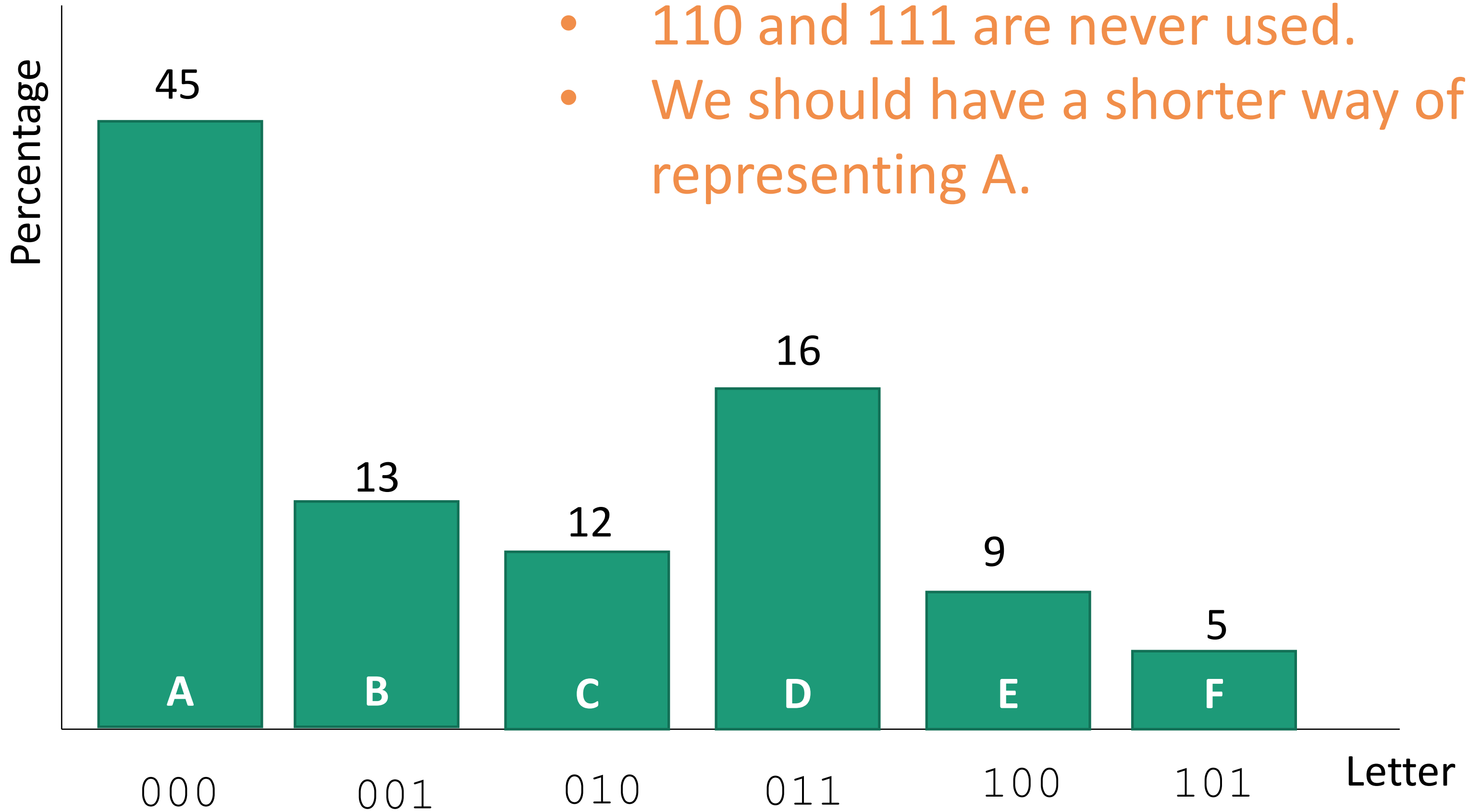
How to encode them as efficiently as possible?

Try 0
(like ASCII)

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

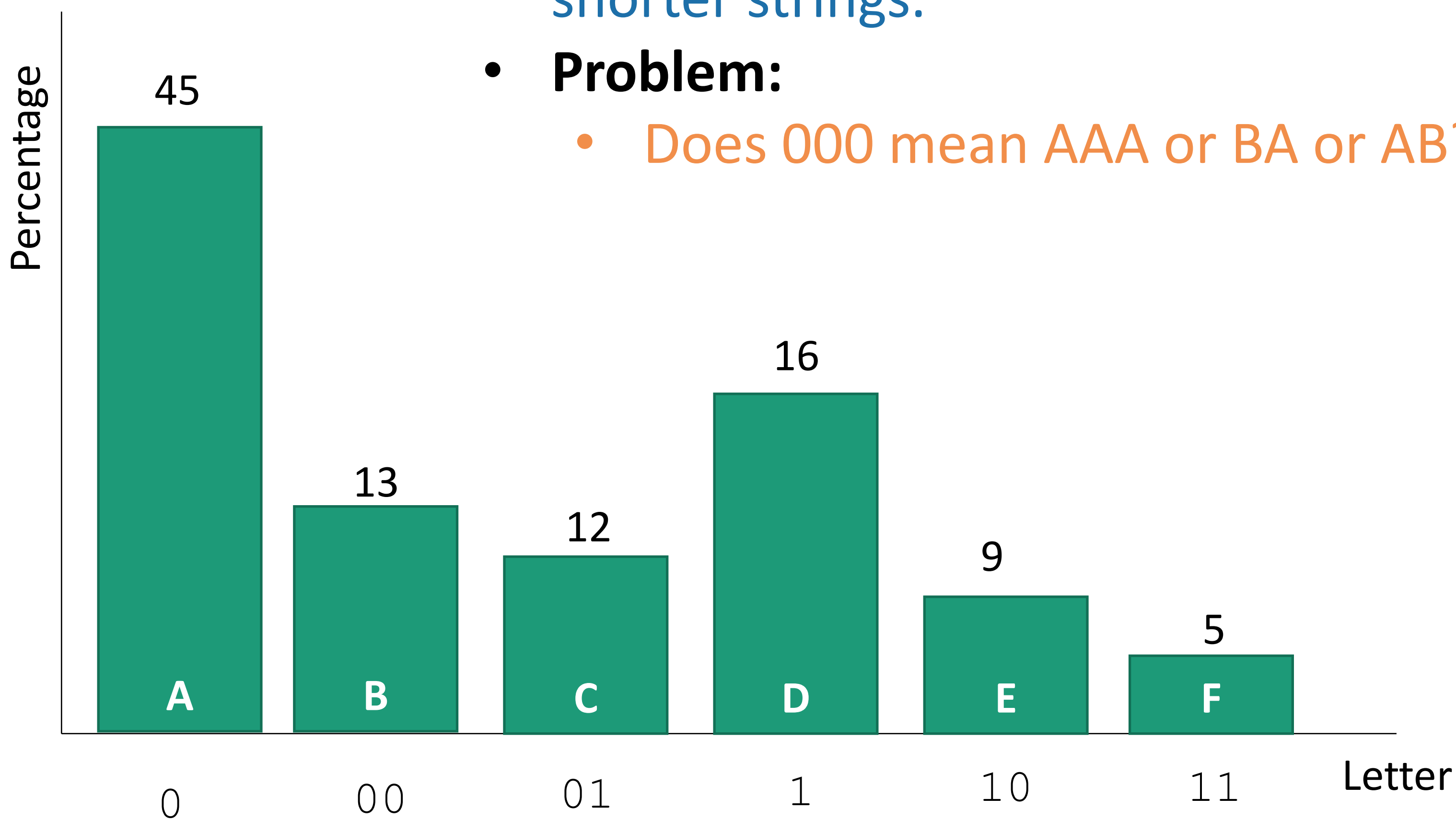
Wasteful!

- 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.
- The more frequent letters get the shorter strings.
- **Problem:**
  - Does 000 mean AAA or BA or AB?

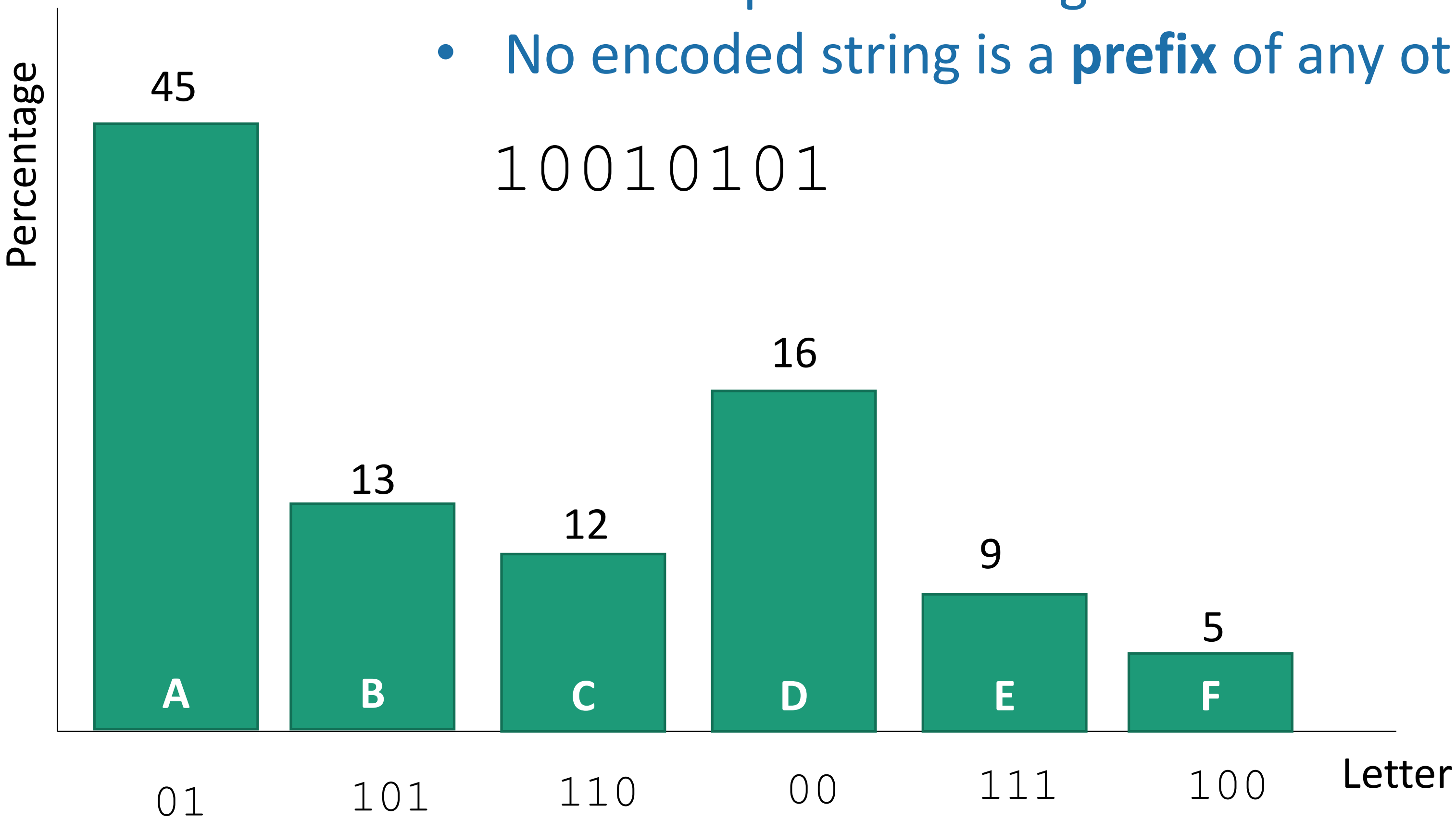


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (e.g. in CLRS).

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.

10010101

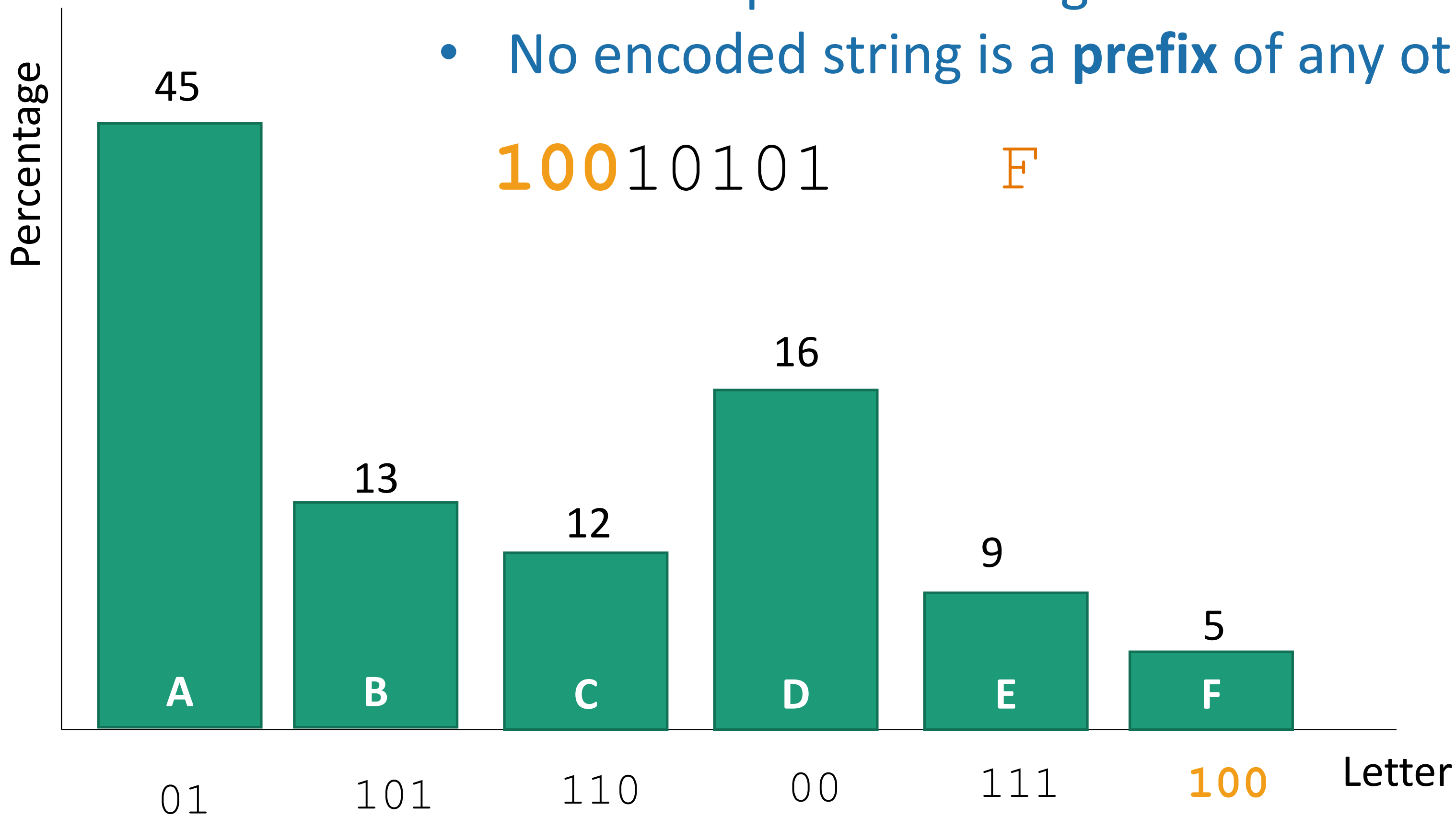


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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.

10010101 F

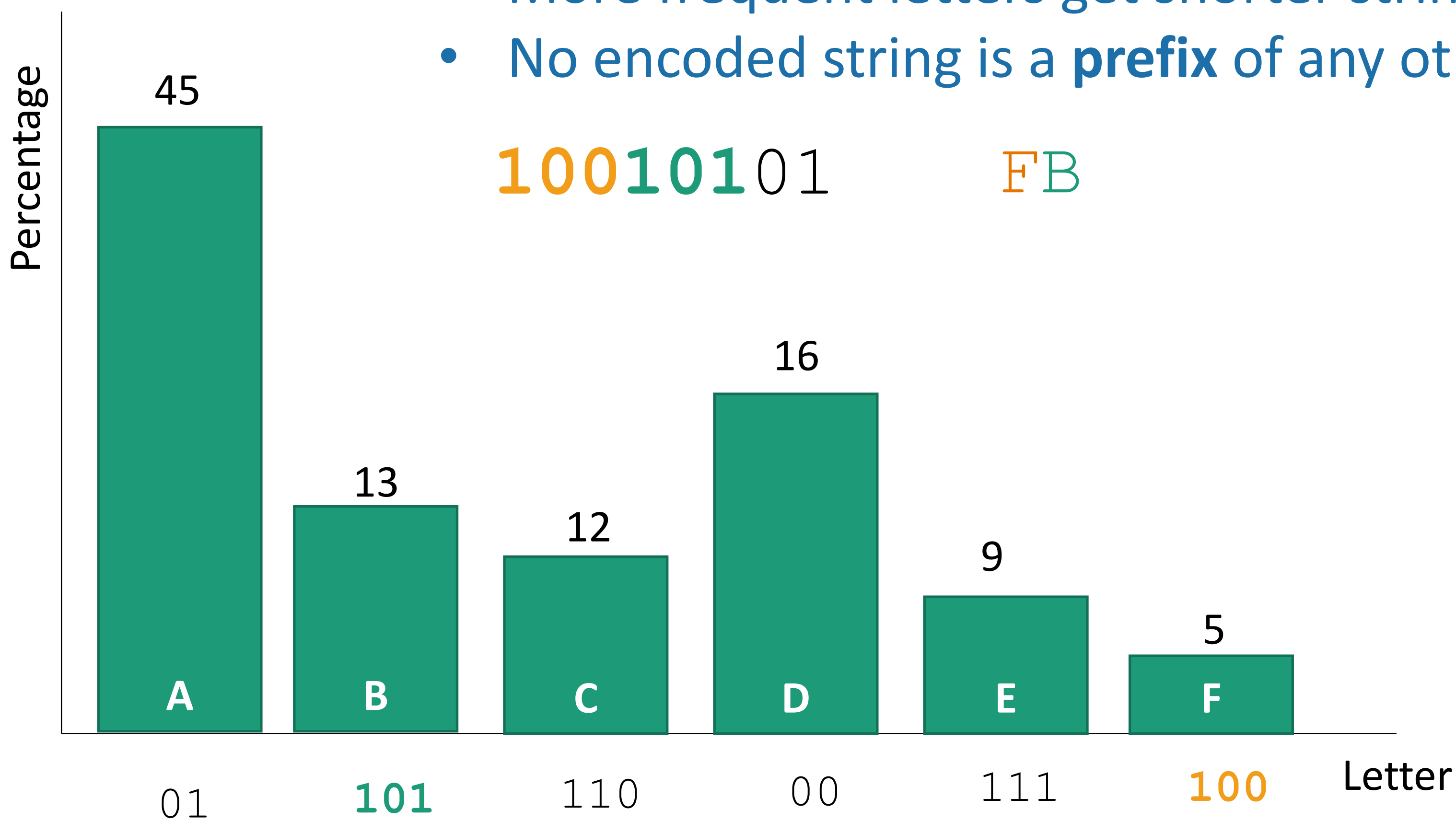


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

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.

10010101 FB

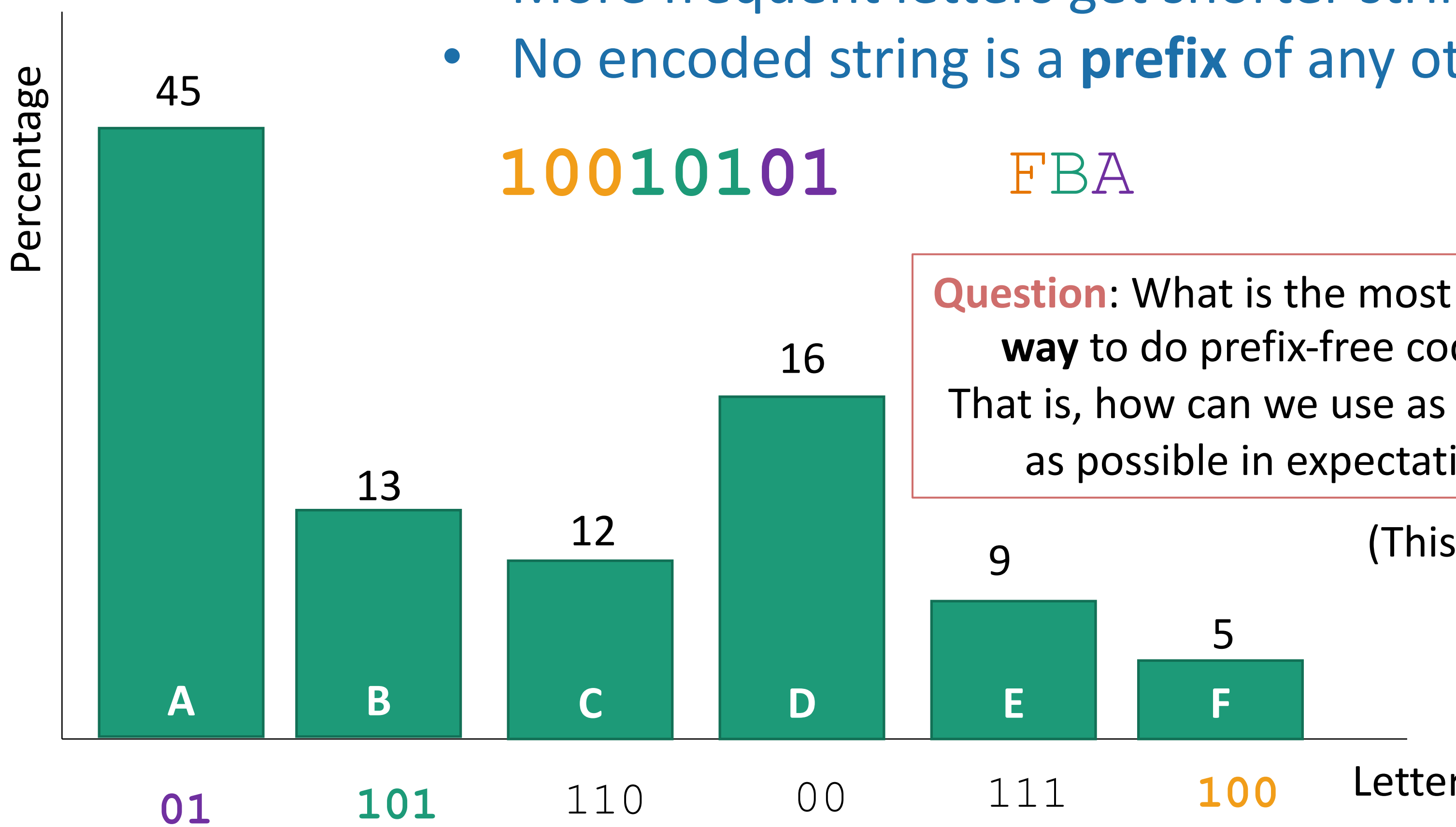


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

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.

10010101 FBA

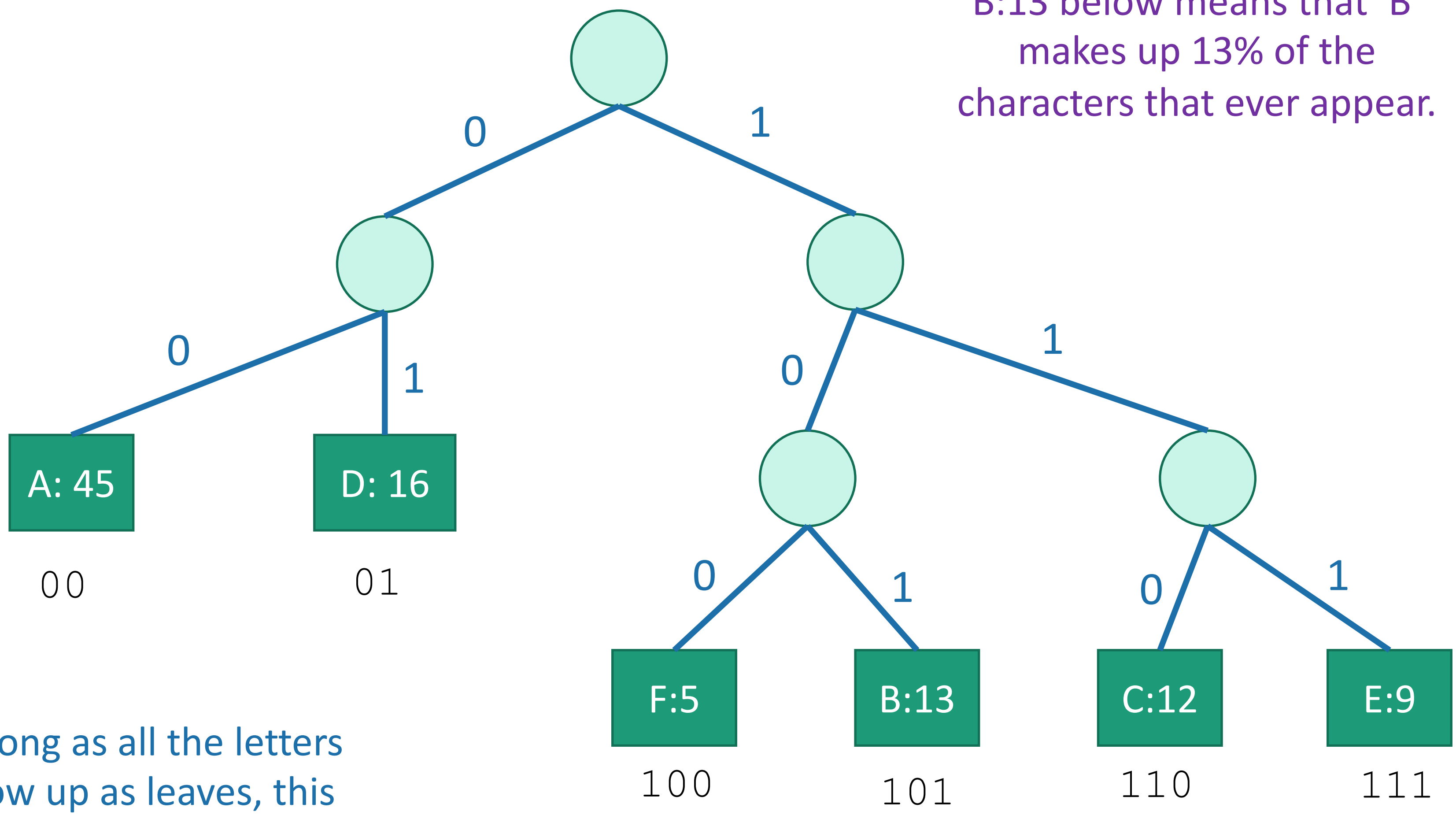


Question: What is the most **efficient way** to do prefix-free coding? That is, how can we use as few bits as possible in expectation?

(This is not it).

A prefix-free code is a tree

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



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

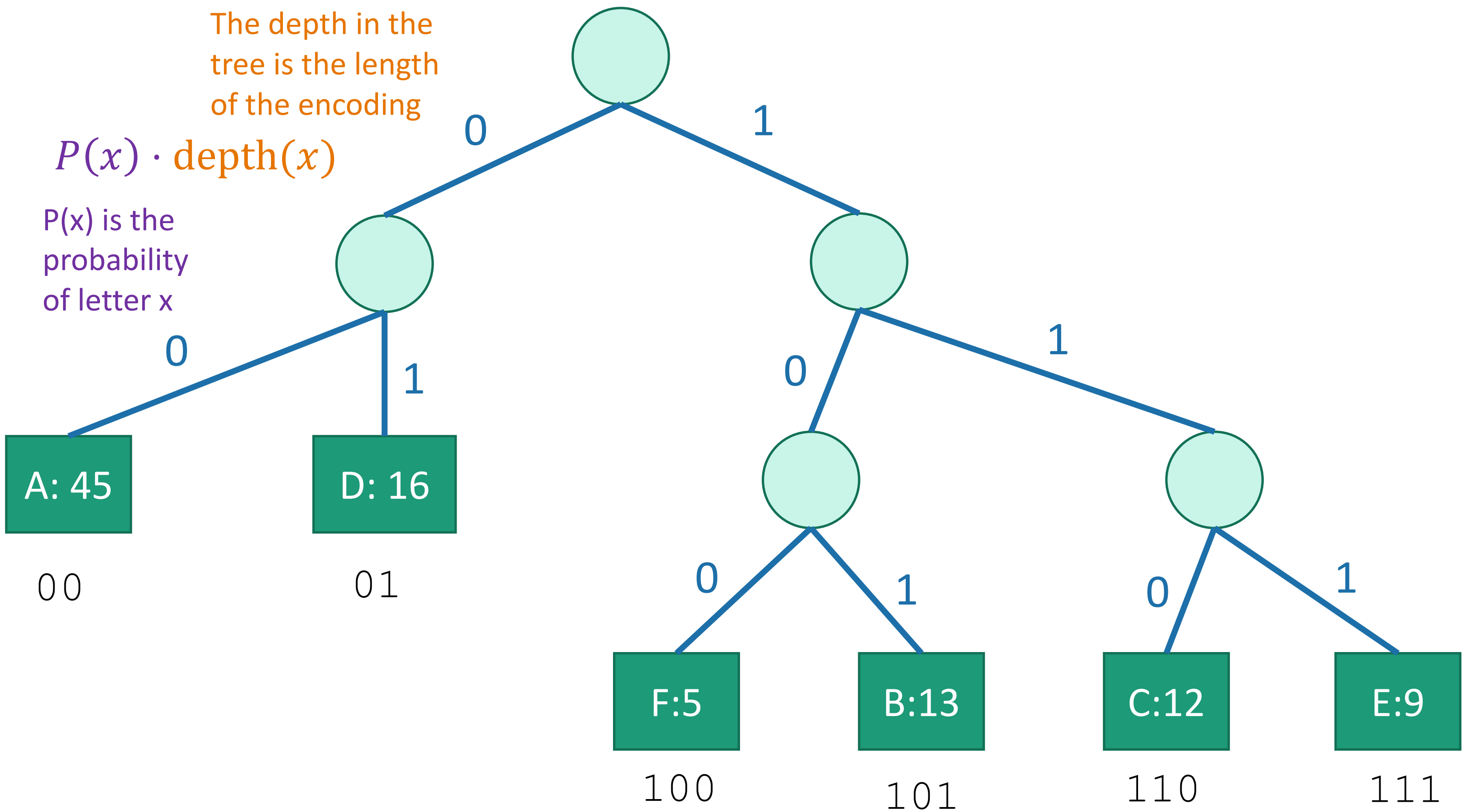
How good is a tree?

- 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.

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

The depth in the tree is the length of the encoding

P(x) is the probability of letter x



Expected cost of encoding a letter with this tree:

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

Question

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

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

P(x) is the probability of letter x

The depth in the tree is the length of the encoding

Greedy algorithm

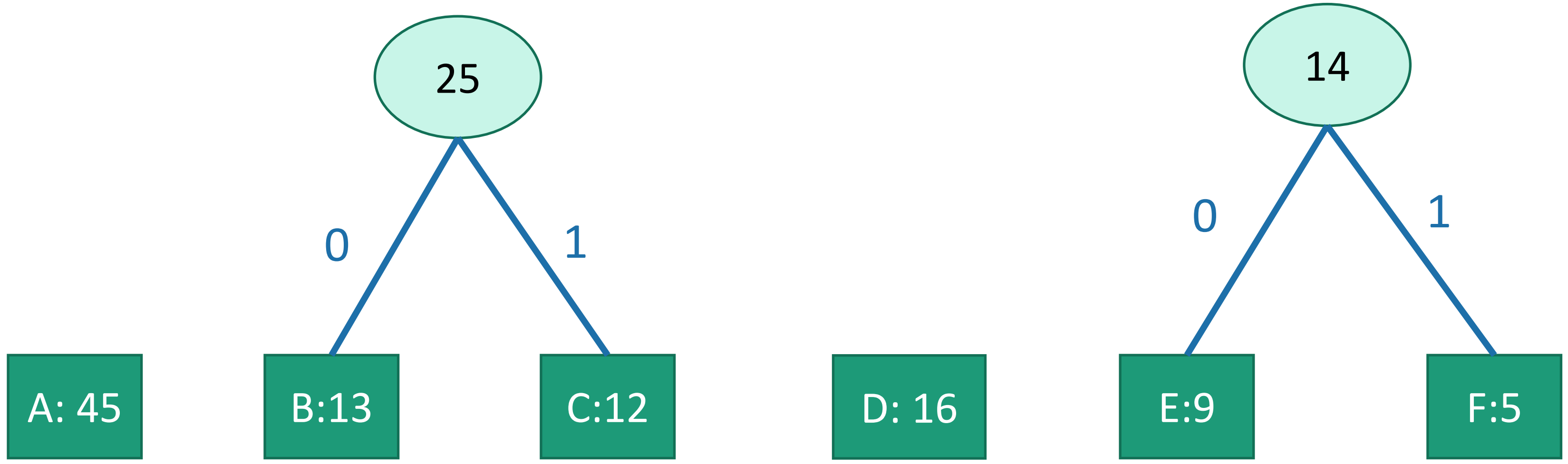
- Greedily build sub-trees from the bottom up.
- Greedy goal: less frequent letters should be further down the tree.

Solution

greedily build subtrees, starting with the infrequent letters

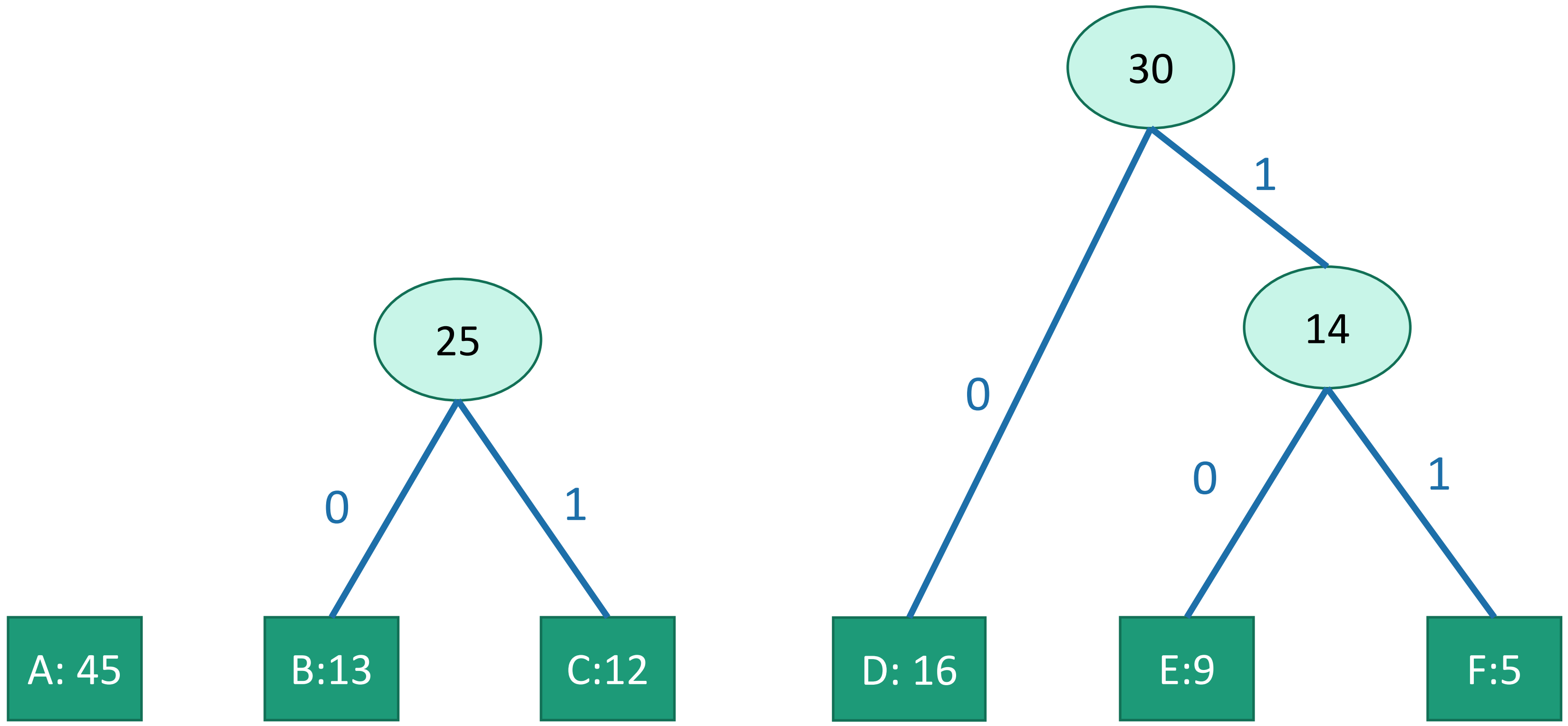
Solution

greedily build subtrees, starting with the infrequent letters



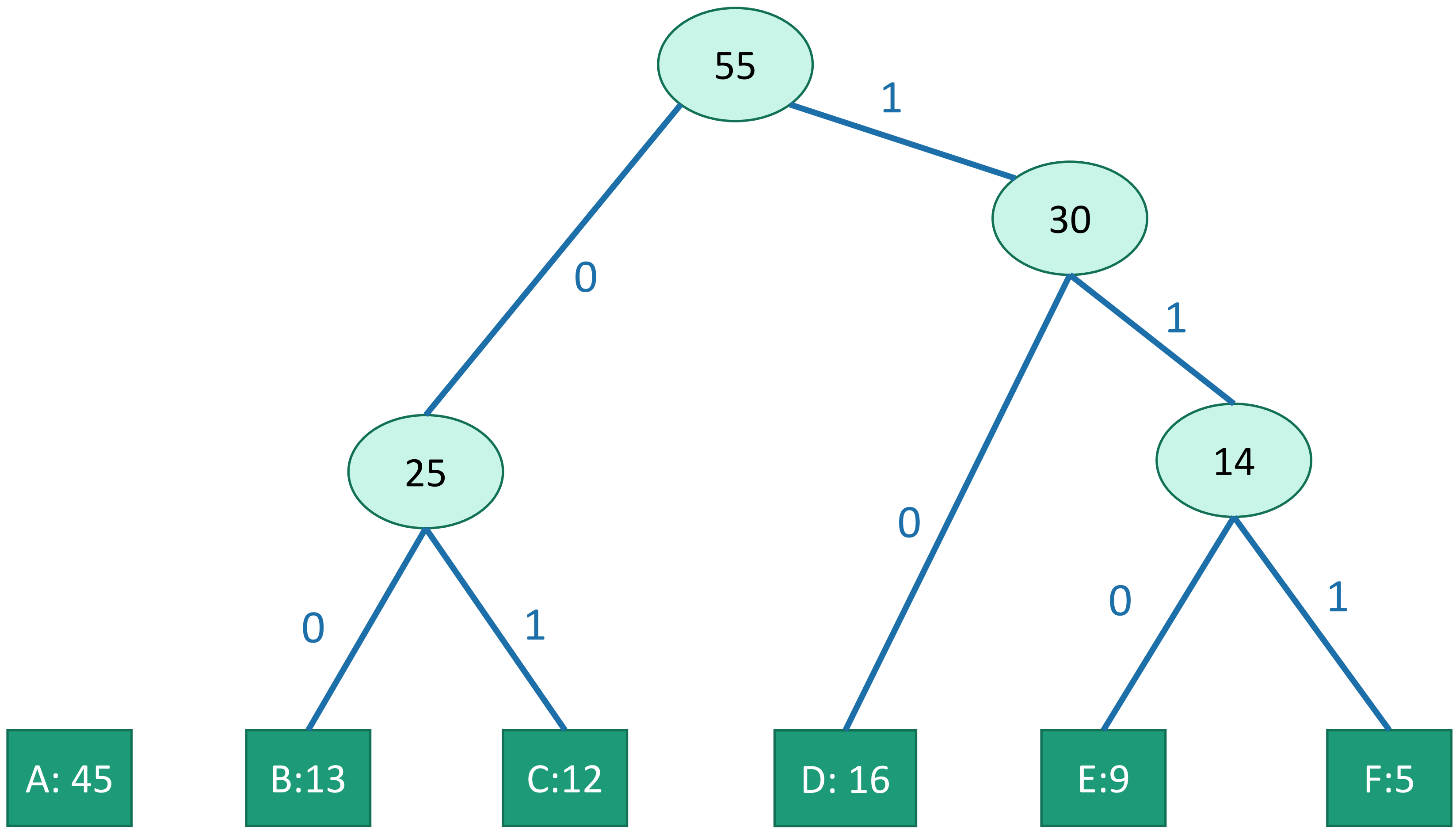
Solution

greedily build subtrees, starting with the infrequent letters



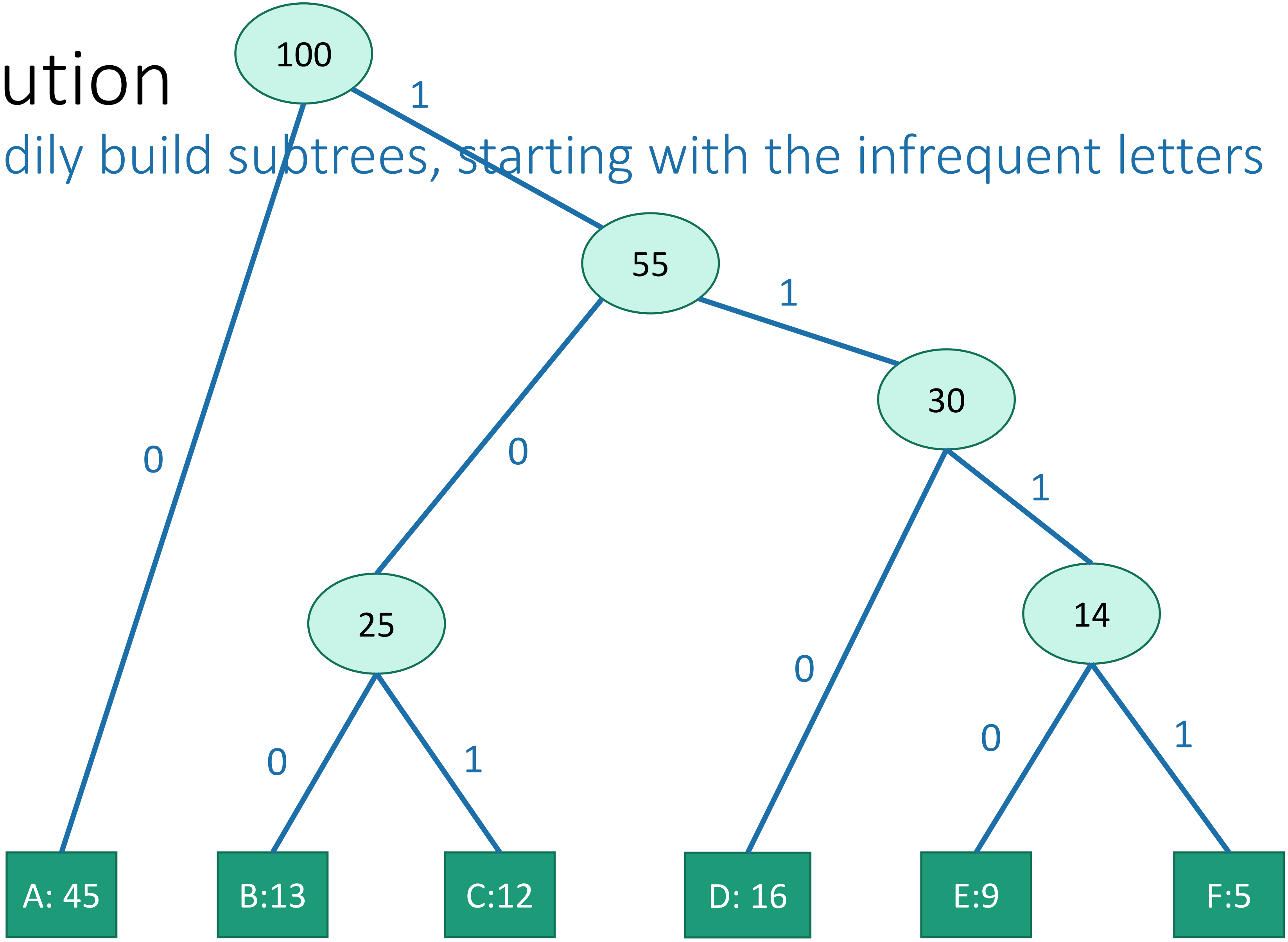
Solution

greedily build subtrees, starting with the infrequent letters



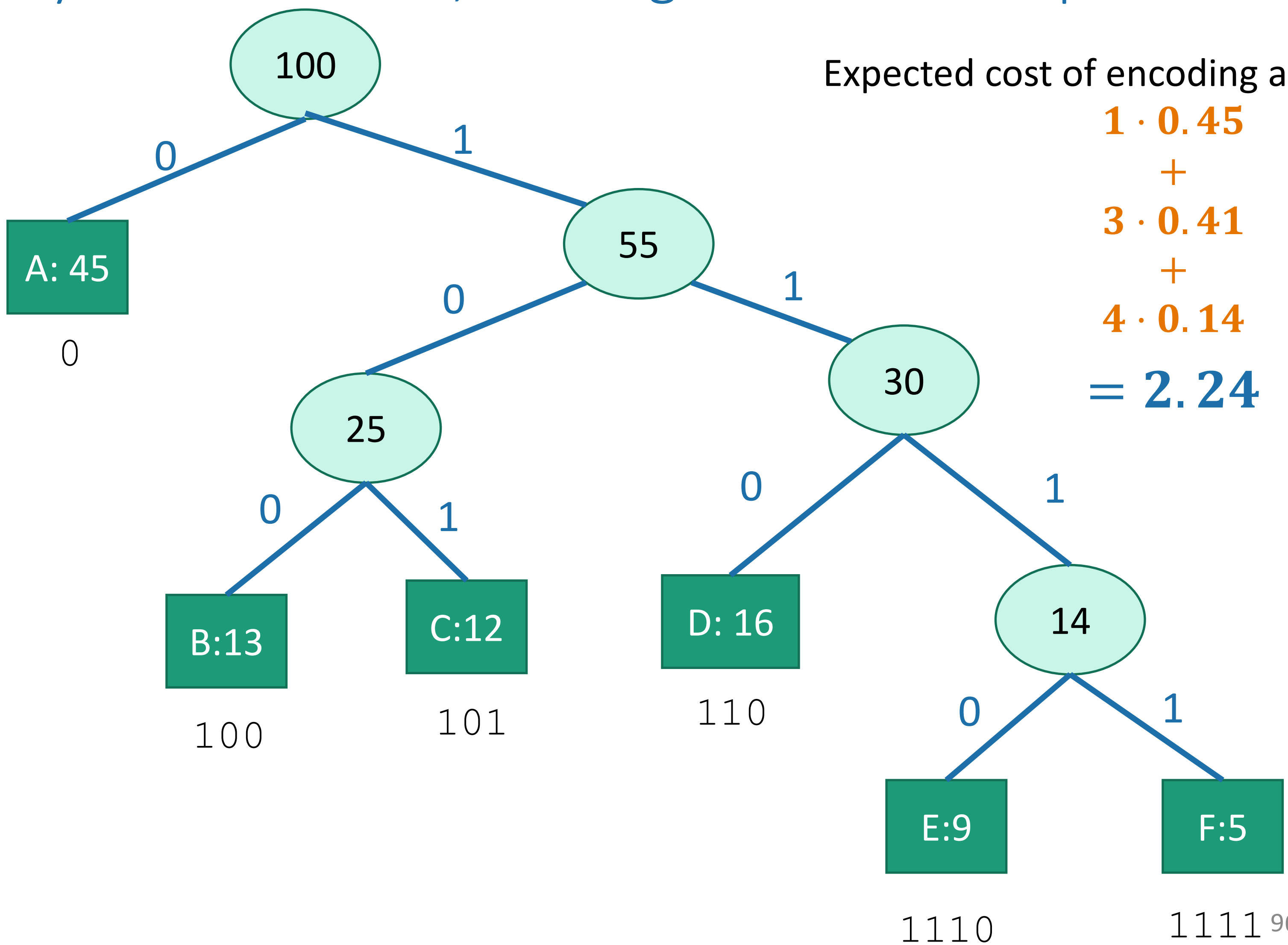
Solution

greedily build subtrees, starting with the infrequent letters



Solution

greedily build subtrees, starting with the infrequent letters

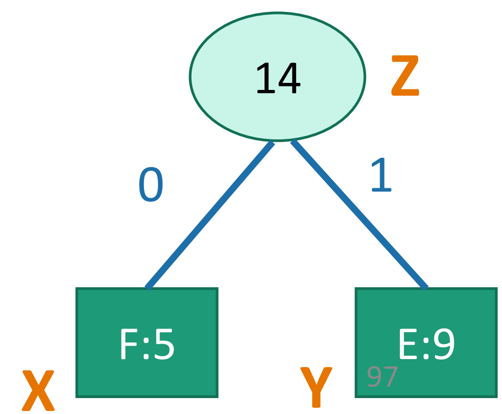


Expected cost of encoding a letter:

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

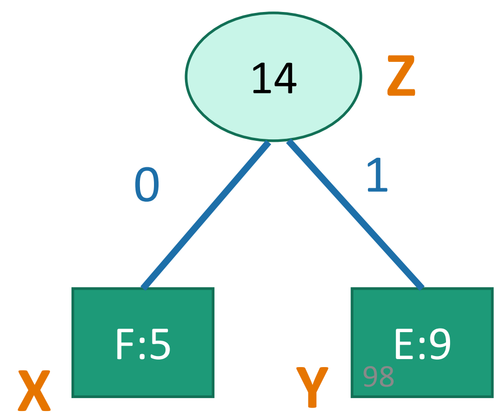
What exactly was the algorithm?

- Create a node like D: 16 for each letter/frequency
  - The key is the frequency (16 in this case)
- 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
  - The key is the frequency (16 in this case)
- 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]



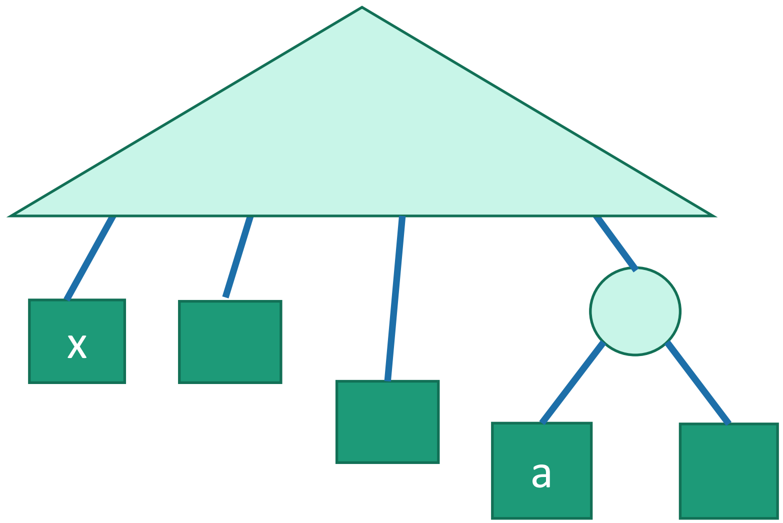
Does it work?

- Yes.
- 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.
  - Lemma:
    - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

Lemma
proof idea

If x and y are the two least-frequent letters, there is an optimal tree where x and y are siblings.

- Say that an optimal tree looks like this:



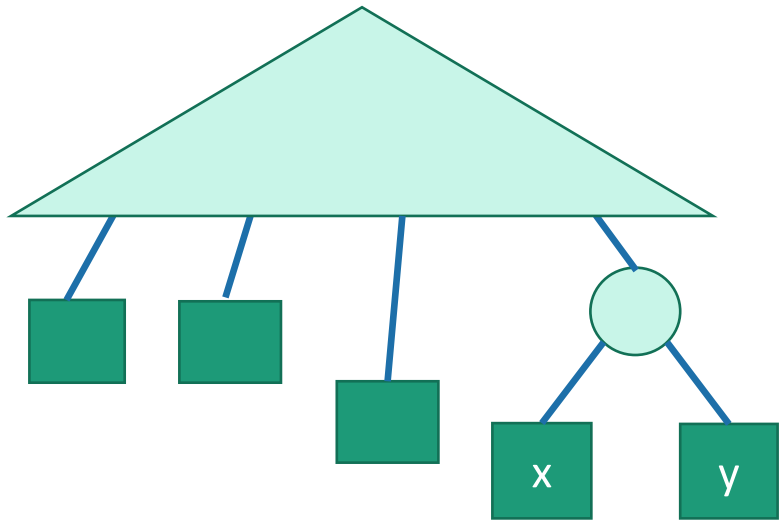
Lowest-level sibling nodes: at least one of them is neither x nor y

- 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.

Lemma
proof idea

If x and y are the two least-frequent letters, there is an optimal tree where x and y are siblings.

- Say that an optimal tree looks like this:

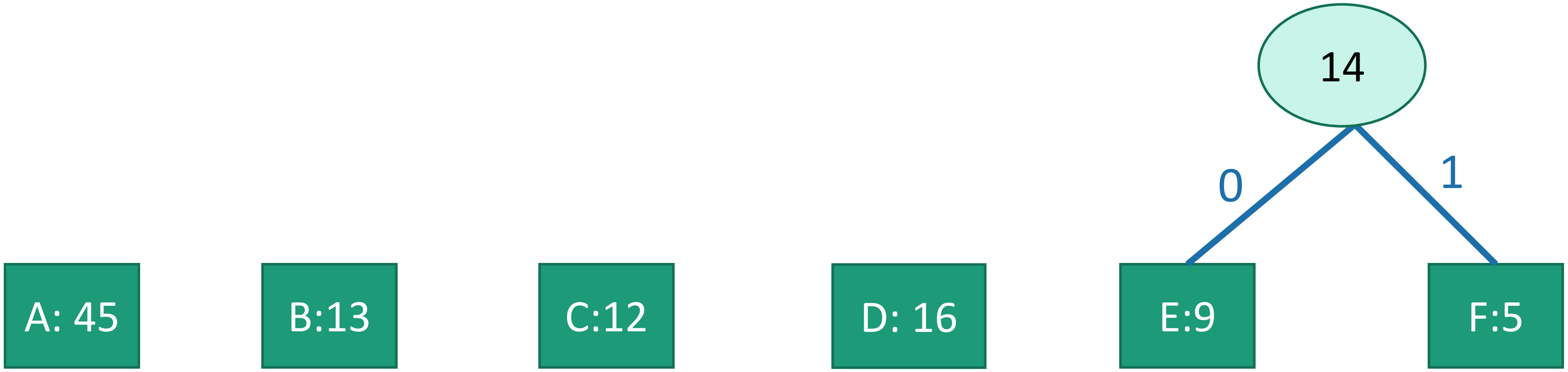


Lowest-level sibling nodes: at least one of them is neither x nor y

- 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.

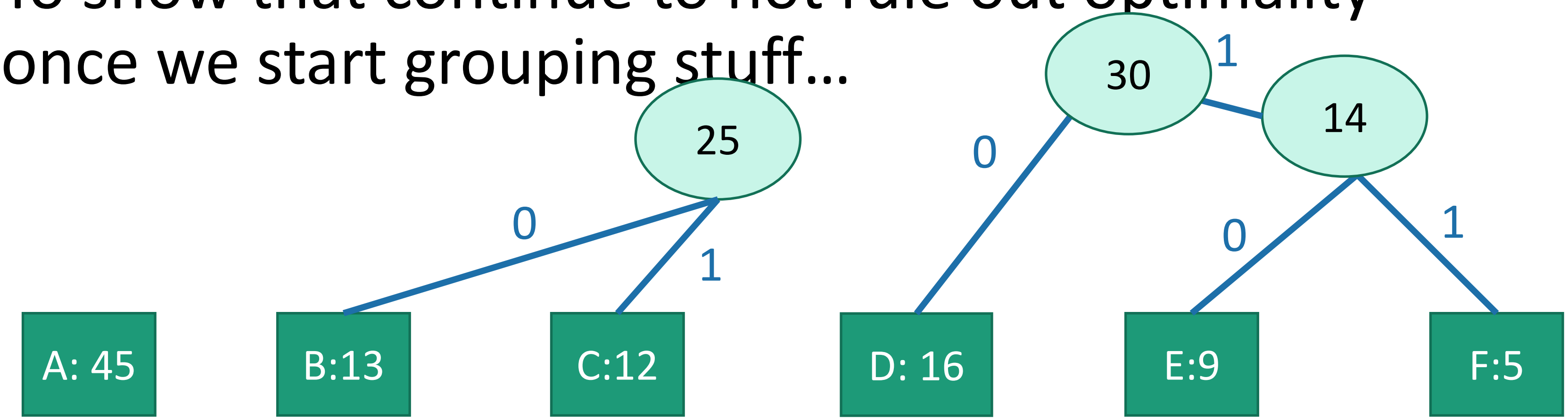
Huffman Coding Works (idea)

- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
  - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.
- That's enough to show that we don't rule out optimality on the first step.



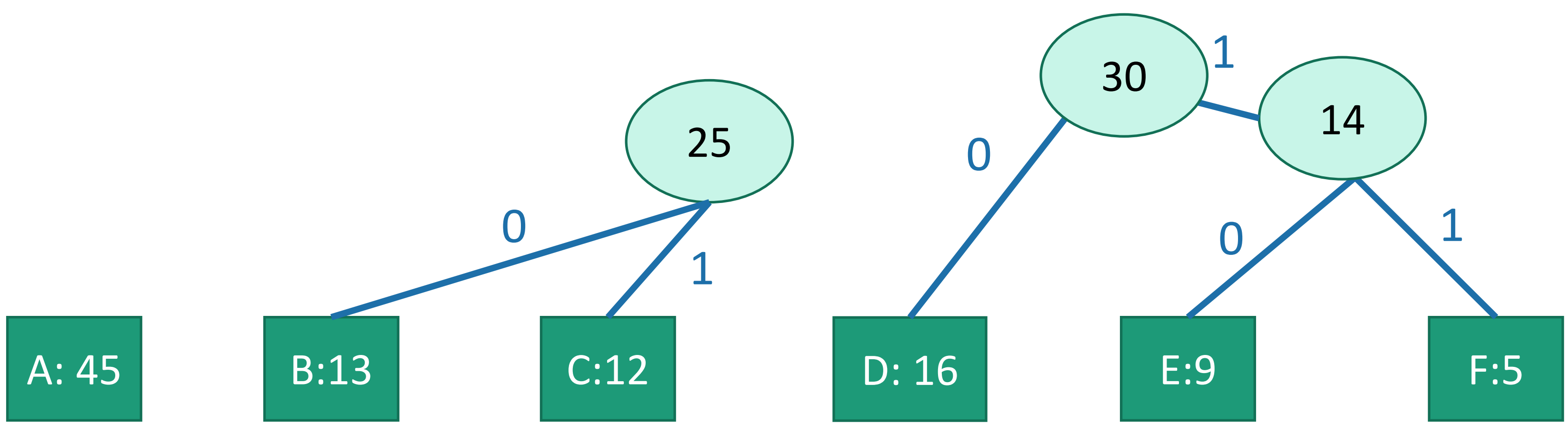
Huffman Coding Works (idea)

- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
  - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.
- That's enough to show that we don't rule out optimality on the first step.
- To show that continue to not rule out optimality once we start grouping stuff…



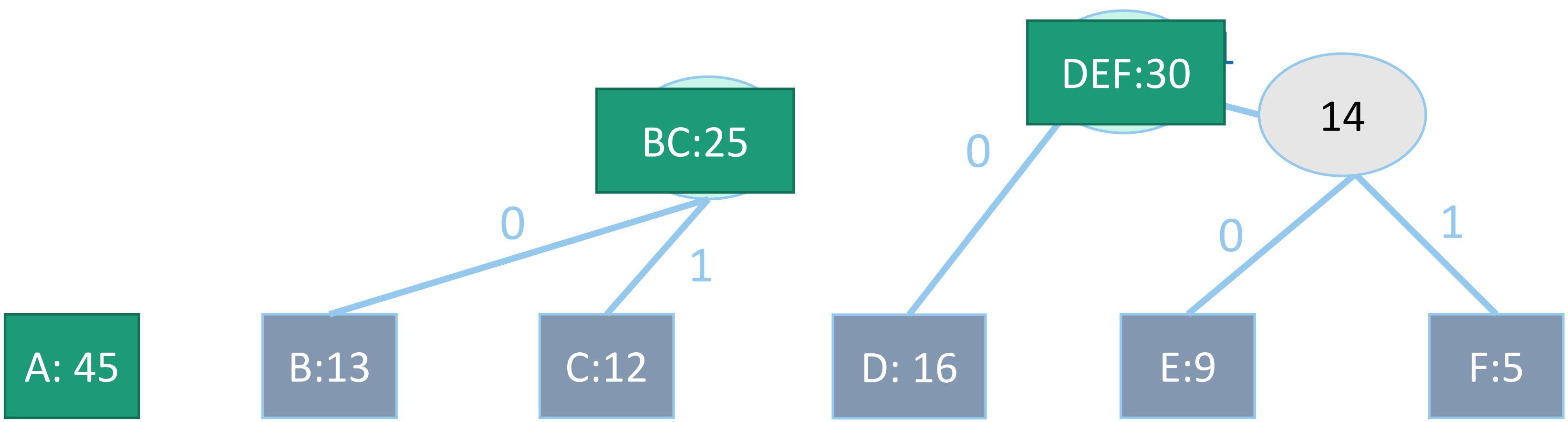
Huffman Coding Works (idea)

- To show that continue to not rule out optimality once we start grouping stuff…
- The basic idea is that we can treat the “groups” as leaves in a new alphabet.



Huffman Coding Works (idea)

- To show that continue to not rule out optimality once we start grouping stuff…
- The basic idea is that we can treat the “groups” as leaves in a new alphabet.
- Then we can use the lemma from before.



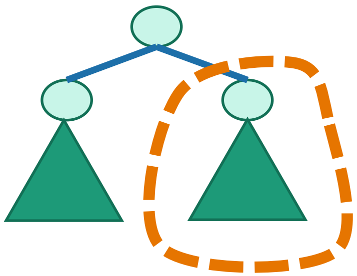
For a full proof

- See lecture notes or CLRS!

*If all we care about is number of bits.

What have we learned?

- ASCII isn't an optimal way* to encode English, since the distribution on letters isn't uniform.
- 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 I

- Greedy algorithms!
- Three examples:
  - Activity Selection
  - Scheduling Jobs
  - Huffman Coding
    - If we had time

Recap II

- Greedy algorithms!
- Often easy to write down
  - But may be hard to come up with and 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.

Next time

- Greedy algorithms for Minimum Spanning Tree!

Before next time

- Pre-lecture exercise: thinking about MSTs