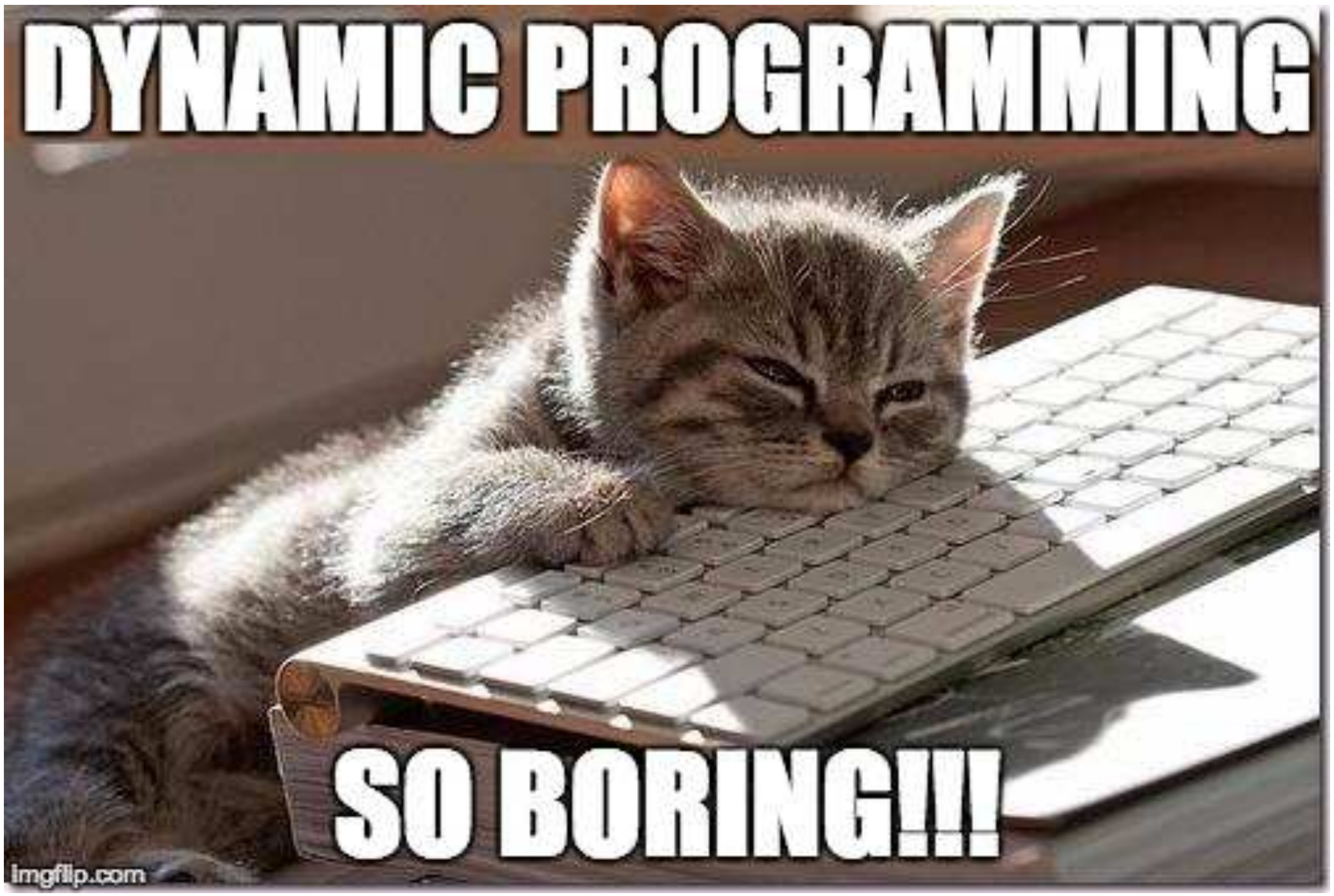
Lecture 14

Greedy algorithms!

Announcements

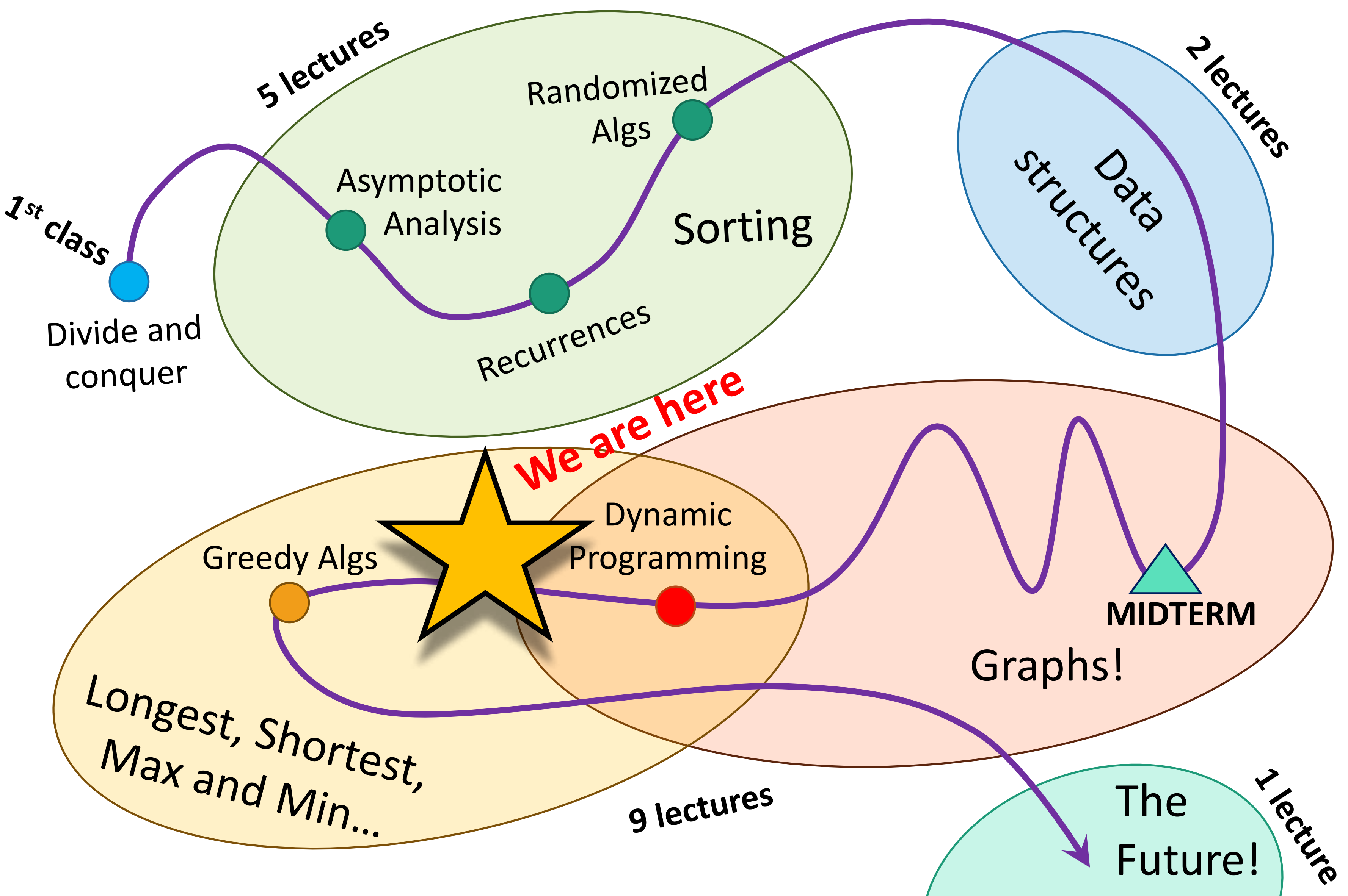
- HW6 Due Friday!
  - ***TONS OF PRACTICE ON DYNAMIC PROGRAMMING***

Last week



More detailed schedule on the website!

Roadmap



This week

- Greedy algorithms!
- Builds on our ideas from dynamic programming

Greedy algorithms

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

Today

- One **non**-example of a greedy algorithm:
  - Knapsack again
- Three examples of greedy algorithms:
  - Activity Selection
  - Job Scheduling
  - Huffman Coding

Non-example

- Unbounded Knapsack.
- (From pre-lecture exercise)

Capacity: 10

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

- Unbounded Knapsack:
  - Suppose I have infinite copies of all of the 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?

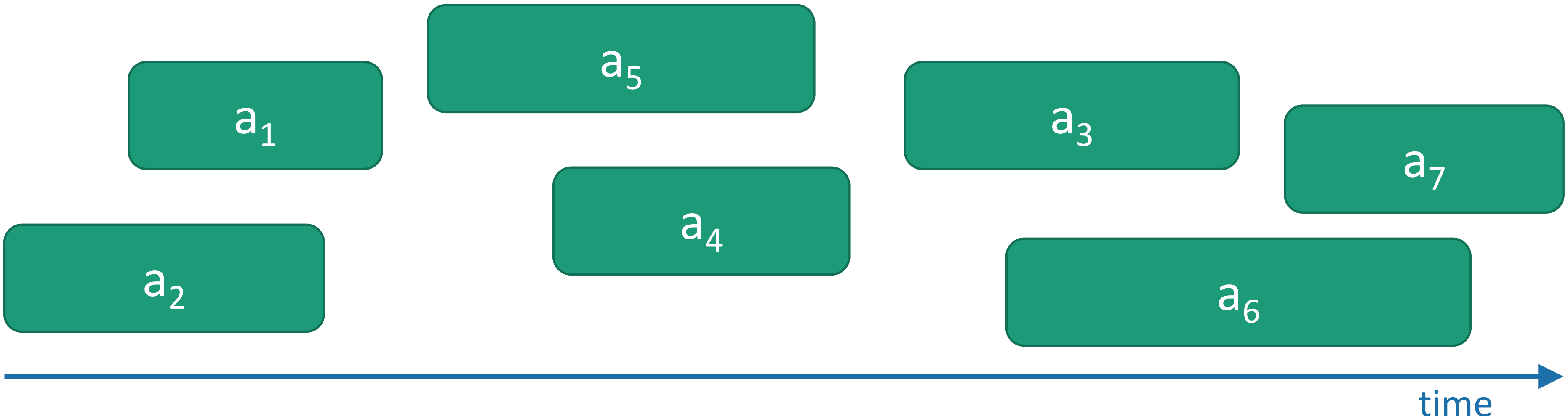
- CS 161 Class
- Math 51 Class
- CS 161 Section
- Sleep
- Underwater basket weaving class
- CS110 Class
- CS161 study group
- Swimming lessons
- Theory Lunch
- Combinatorics Seminar
- Social activity

time

Activity selection

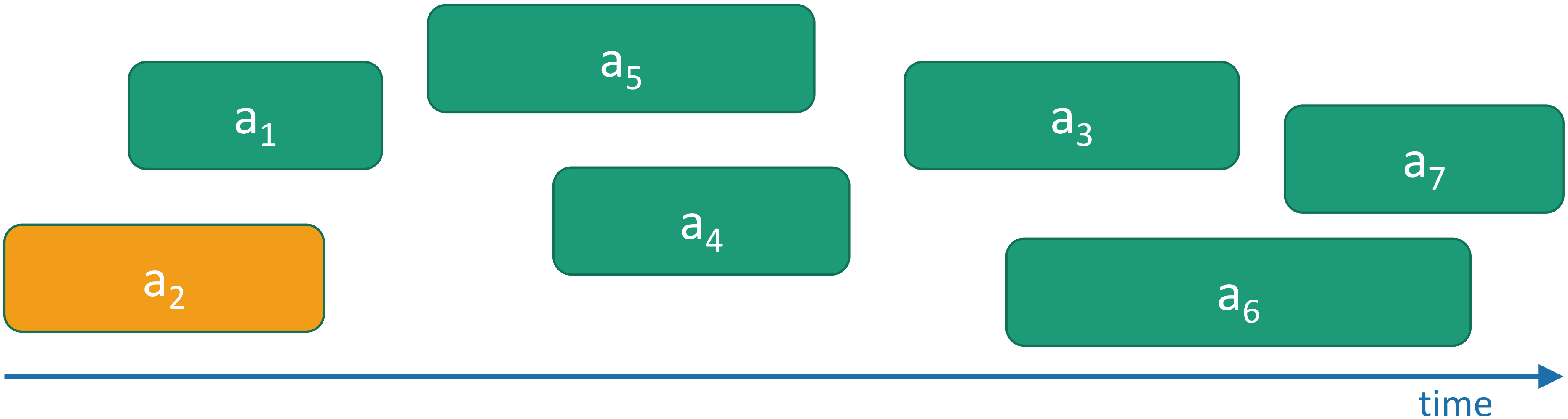
- Input:
  - Activities a_1, a_2, …, a_n
  - Start times s_1, s_2, …, s_n
  - Finish times f_1, f_2, …, f_n
- Output:
  - How many activities can you do today?

Greedy Algorithm



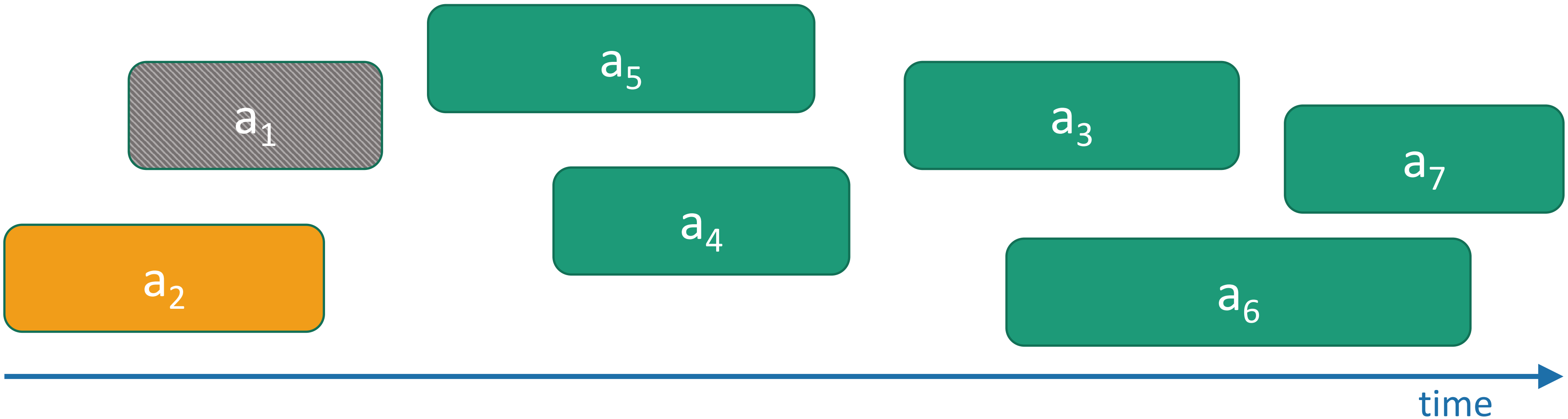
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



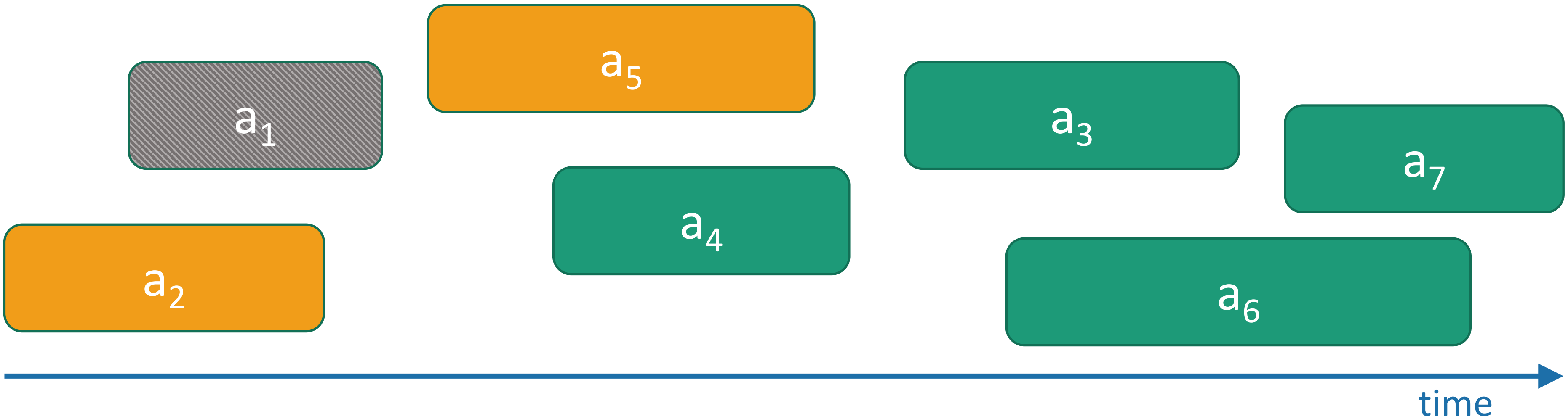
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



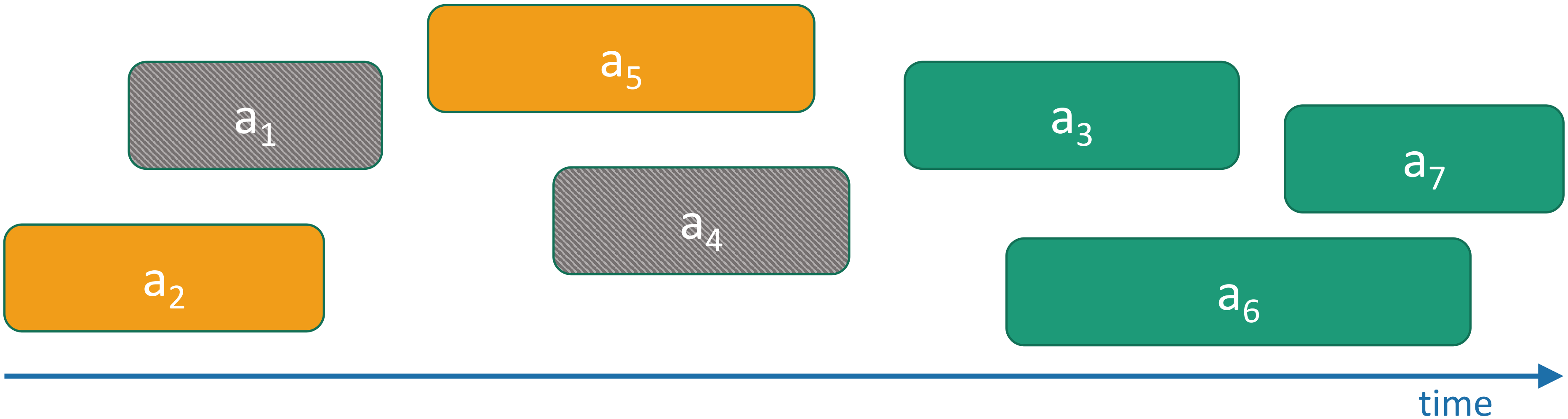
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



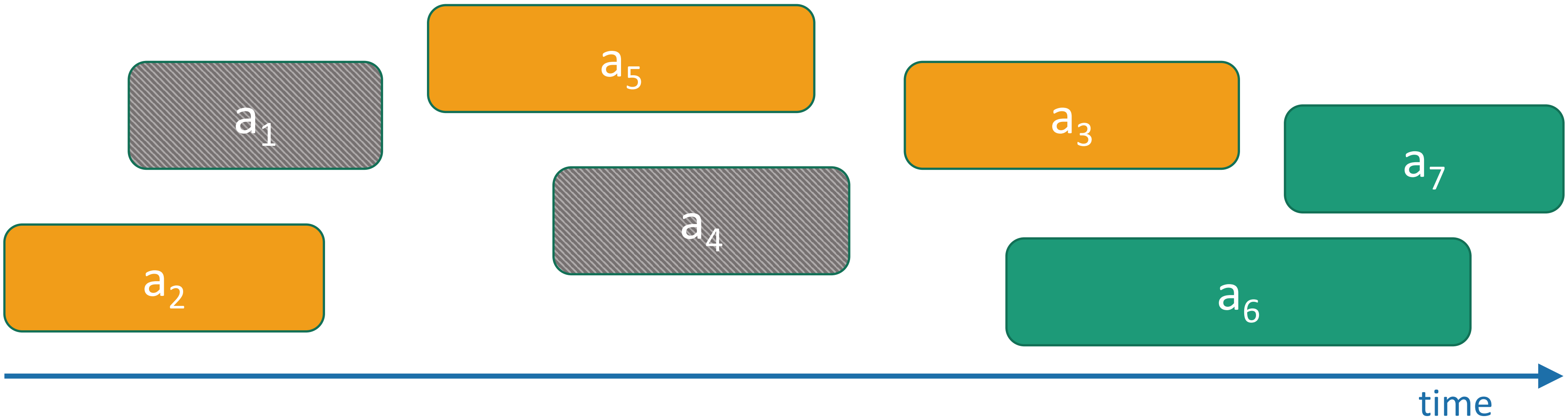
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



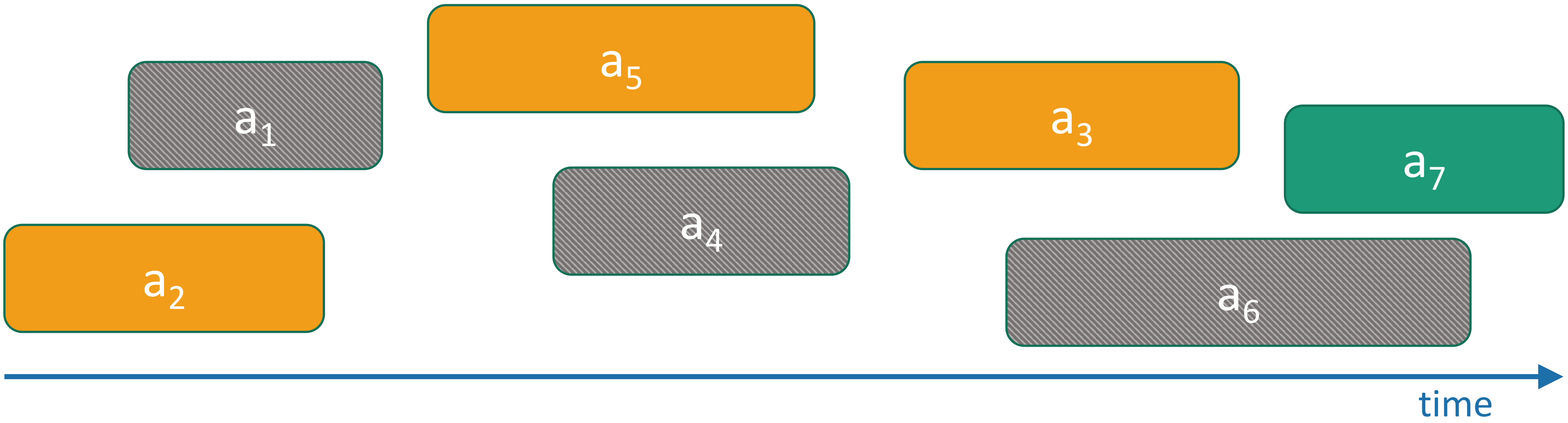
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



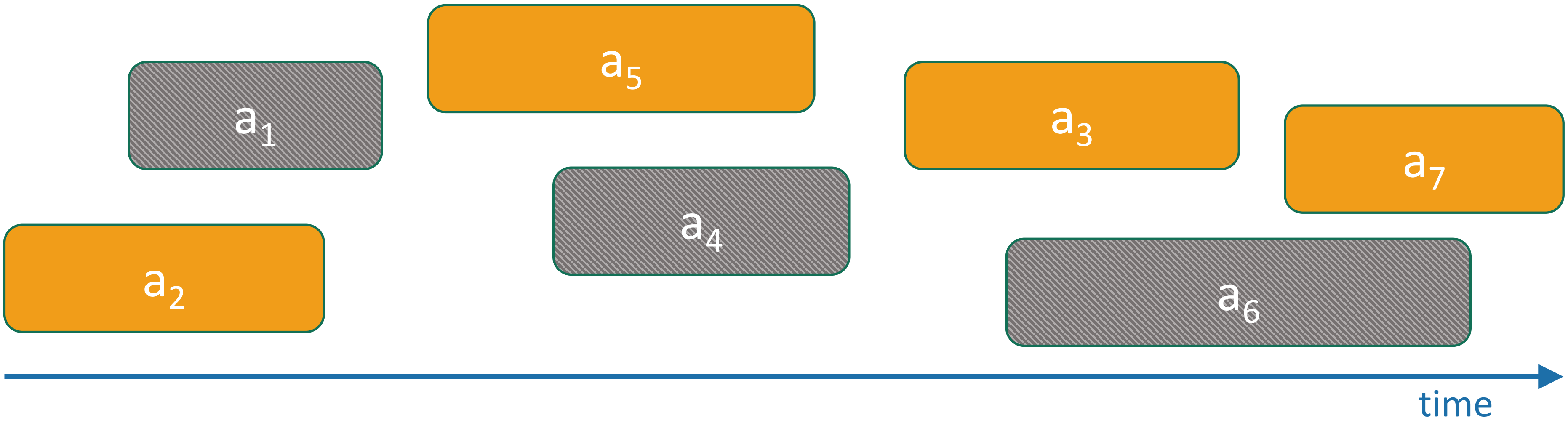
- Pick activity you can add with the smallest finish time.
- Repeat.

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.

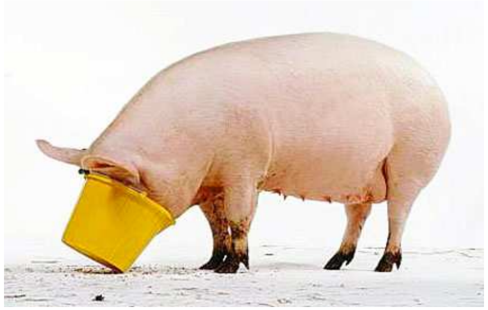
At least it’s fast

- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise O(nlog(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 8?

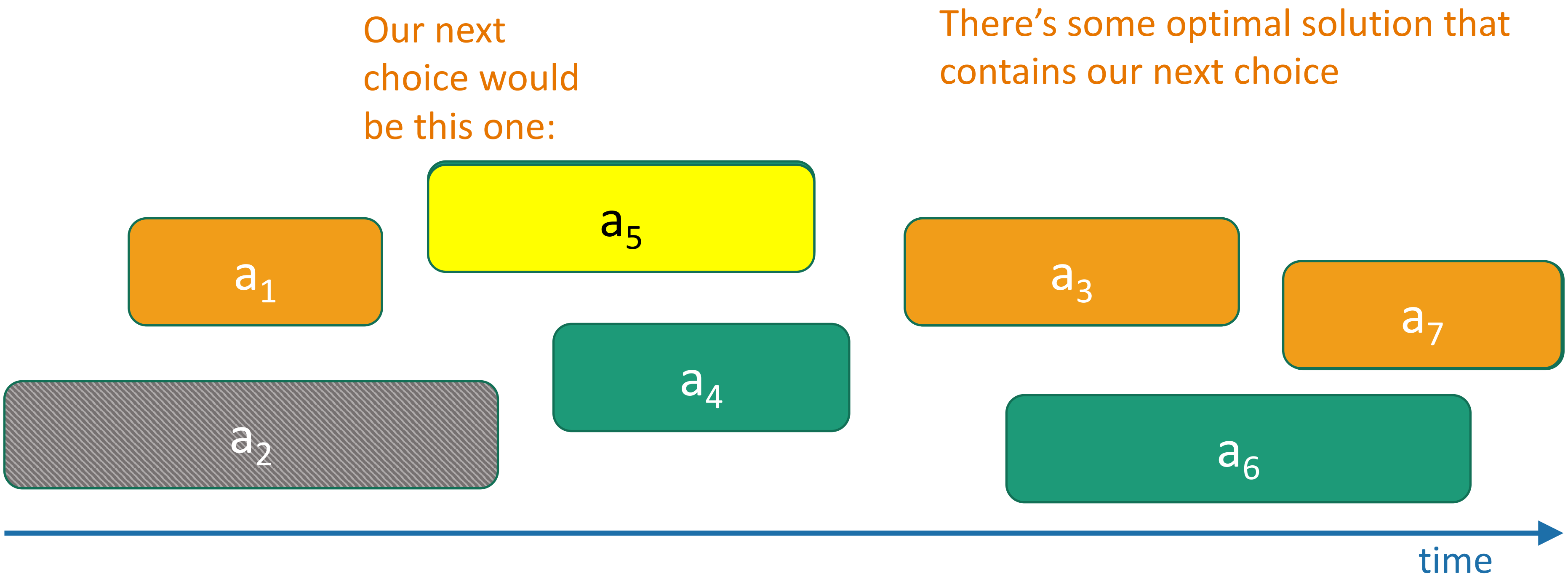
Answers

1. Does this greedy algorithm for activity selection work?
   - Yes. (Seems to: IPython notebook…) (But now let's see why…)
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?
     - Related to dynamic programming! (Which we did in Week 7).
     - Proving that greedy algorithms work is often not so easy.

Why does it work?

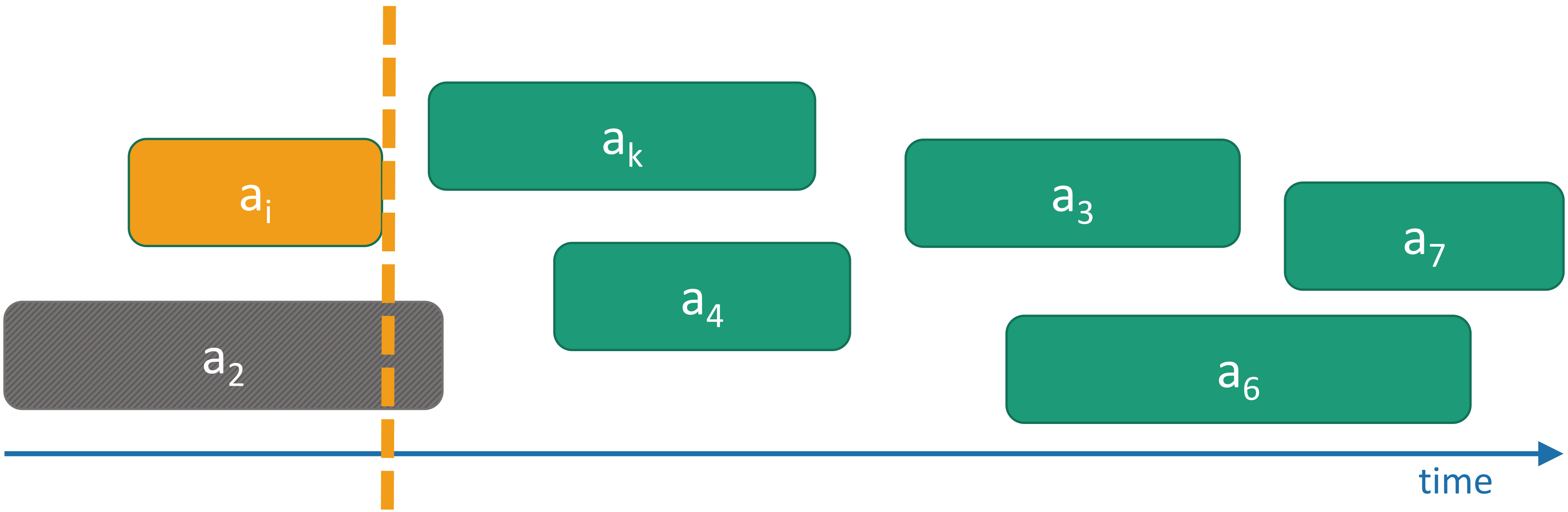
- Whenever we make a choice, **we don't rule out an optimal solution.**



To see this, consider

Optimal Substructure

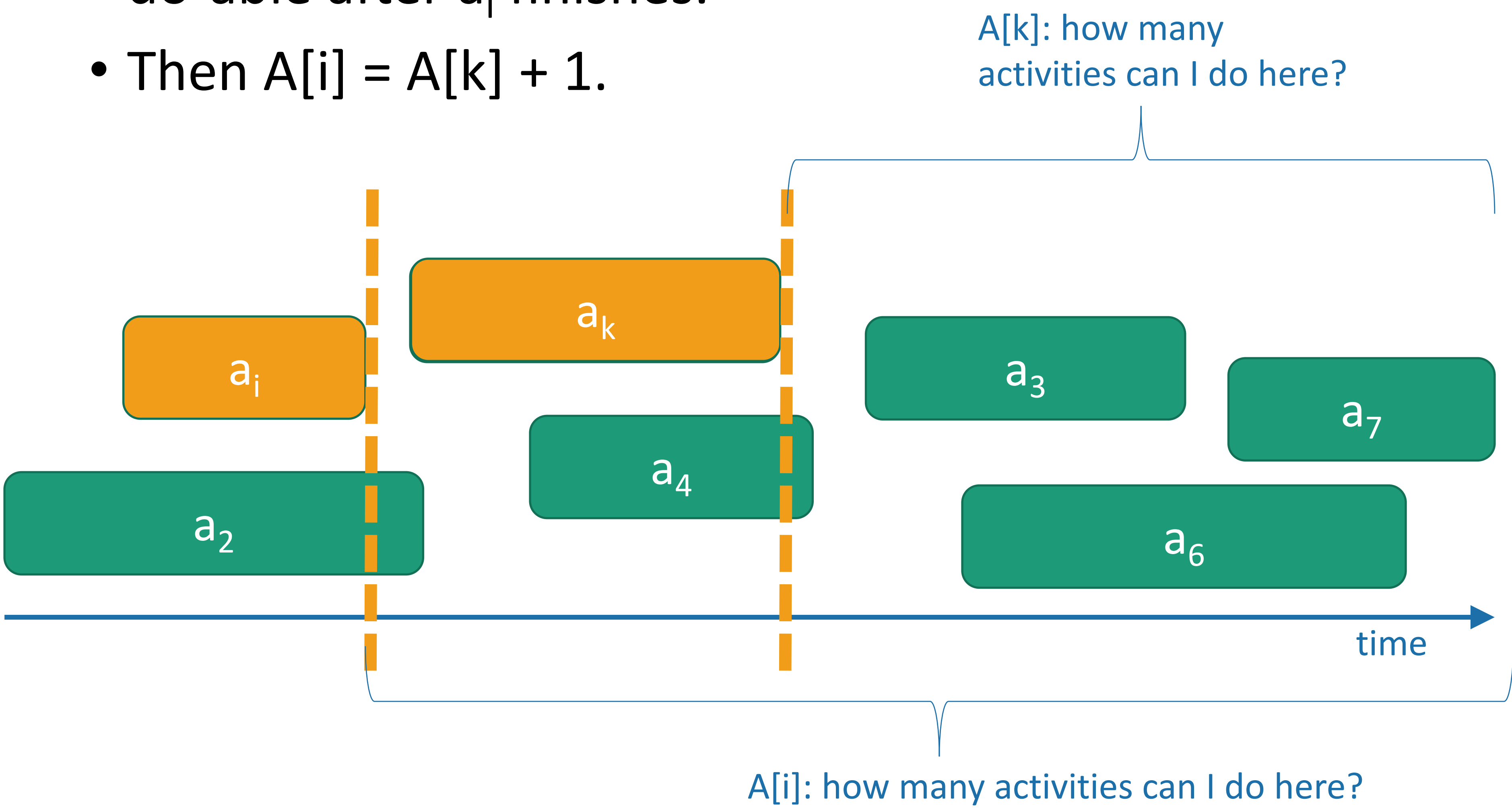
- Subproblem i :
  - A[i] = Number of activities you can do after Activity i finishes.



Want to show: when we make a choice a_k, the optimal solution to the smaller sub-problem k will help us solve sub-problem i

Claim

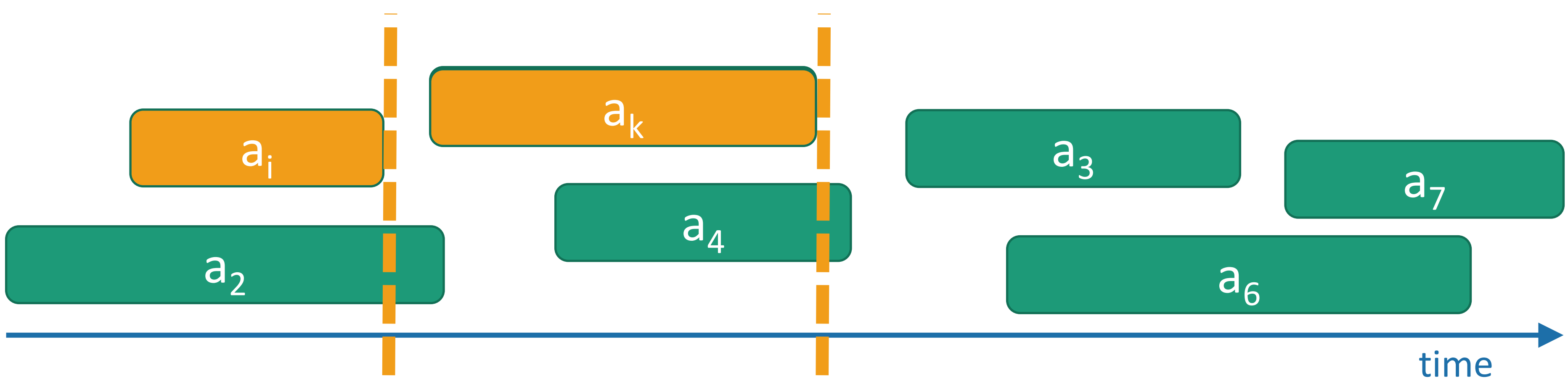
- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then A[i] = A[k] + 1.



Proof

- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then A[i] = A[k] + 1.

- Clearly A[i] $\geq$ A[k] + 1
  - Since we have a solution with A[k] + 1 activities.

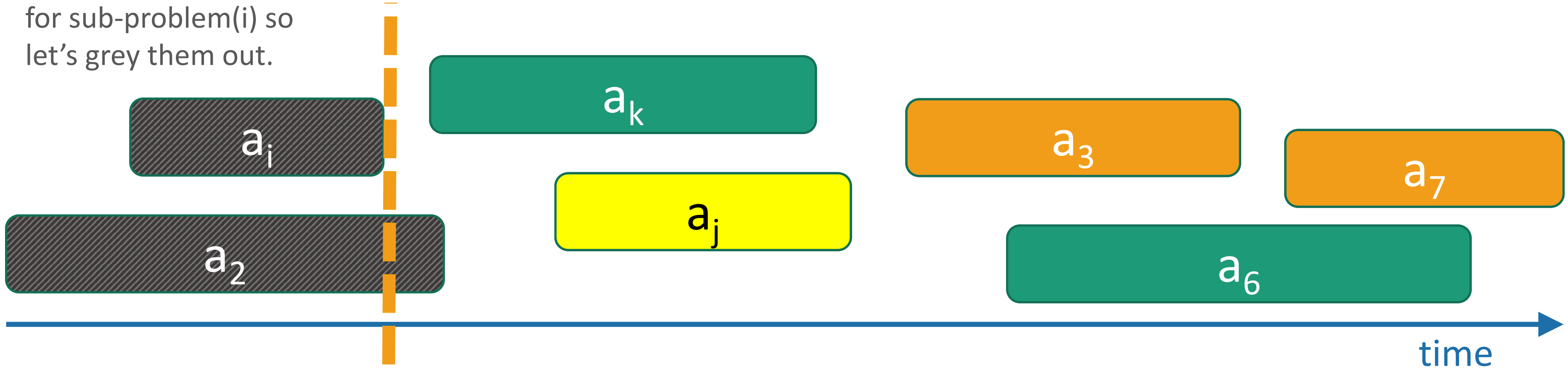


Proof

- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then A[i] = A[k] + 1.

- Suppose toward a contradiction that A[i] > A[k] + 1.
- There's some better solution to subproblem(i) that doesn't use a_k
- Say a_j ends first after a_i in that better solution.
- Remove a_j and add a_k from the better solution.

These two don't count for sub-problem(i) so let's grey them out.



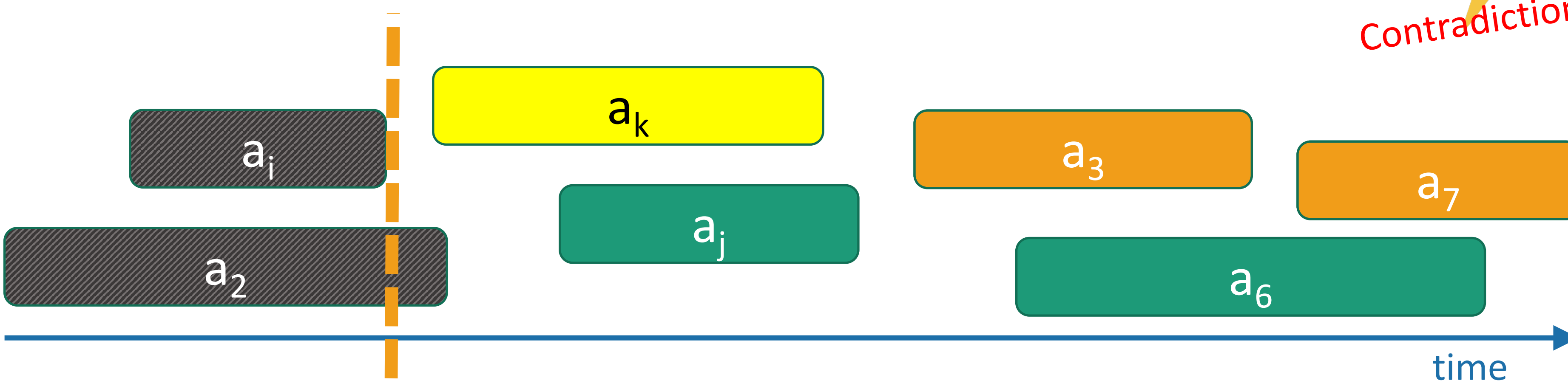
Proof

- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then A[i] = A[k] + 1.

- Suppose toward a contradiction that A[i] > A[k] + 1.
- There's some better solution to subproblem(i) that doesn't use a_k
- Say a_j ends first after a_i in that better solution.
- Remove a_j and add a_k from the better solution.
- Now you have a solution of the same size…

  but it includes a_k so it must have size $\leq$ A[k] + 1.

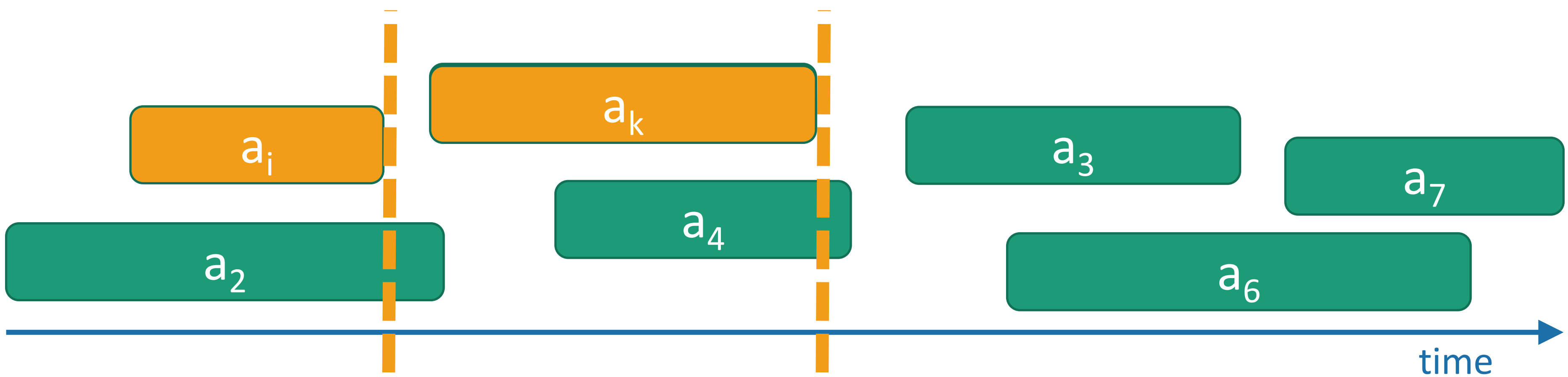
Contradiction!



Proof

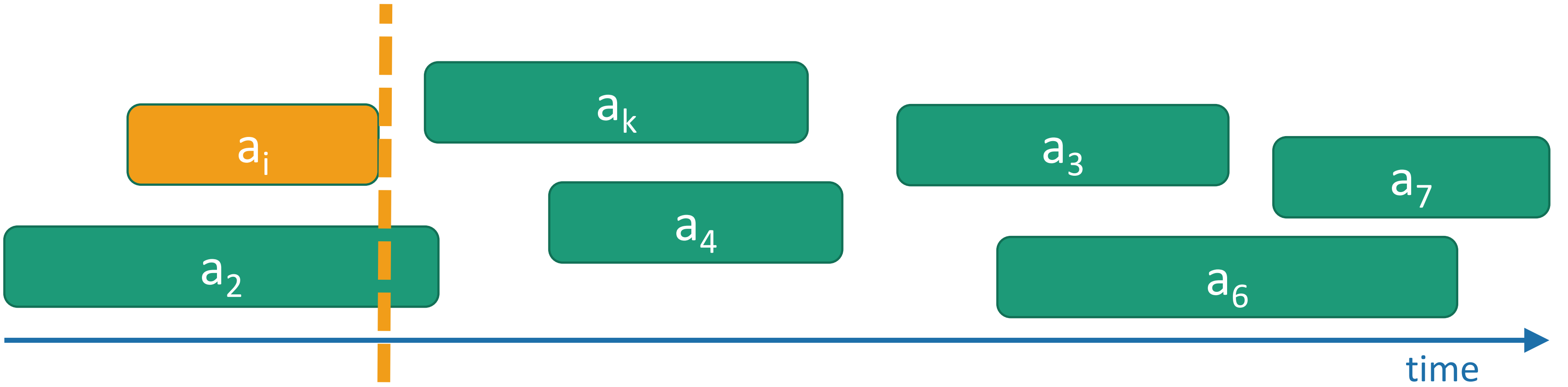
- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then A[i] = A[k] + 1.

- Clearly A[i] $\geq$ A[k] + 1
  - Since we have a solution with A[k] + 1 activities.
- And we just showed A[i] $\leq$ A[k] + 1
  - By contradiction
- That proves the claim.



We never rule out an optimal solution

- We've shown:
  - If we choose a_k have the smallest finish time among activities do-able after a_i finishes, then A[i] = A[k] + 1.
- That is:
  - Assume that we have an optimal solution up to a_i
  - By adding a_k we are still on track to hit that optimal value



So the algorithm is correct

- We never rule out an optimal solution
- At the end of the algorithm, we've got a solution.
- It's not **not** optimal.
- 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:
  - **TO DO**
- 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.

Inductive step

- Suppose that after adding the t'th thing (Activity i), there is an optimal solution:
  - X activities done and A[i] activities left.
- Then we add the (t+1)'st thing (Activity k).
- A[k] = A[i] - 1 (by the claim)
- Now:
  - X+1 activities done and A[i] – 1 activities left.
  - Same number as before!
  - Still optimal.

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:
  - **TO DO**
- 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.

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.**

Common strategy (formally)

for 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:
  - **TODO**
- Conclusion:
  - If you reach the end of the algorithm and haven’t ruled out success then you must have succeeded.

DP view of activity selection

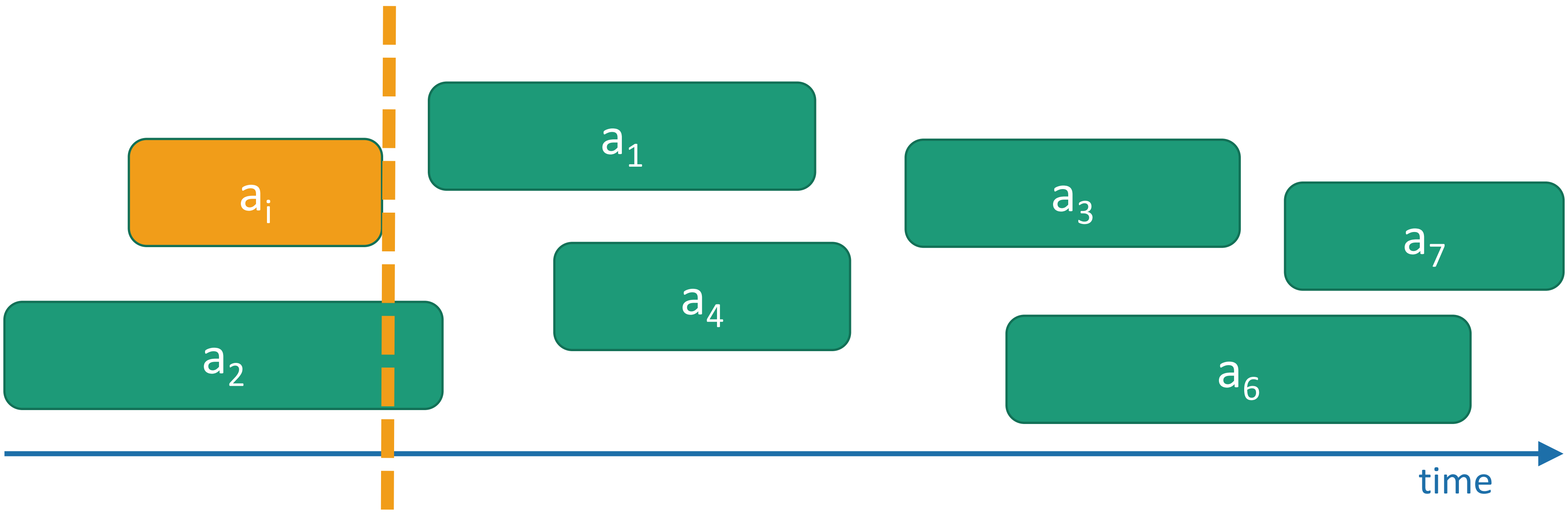
- This algorithm is most naturally viewed as a greedy algorithm.
  - Make greedy choices
  - Never rule out success
- But, we could view it as a DP algorithm
  - Take advantage of optimal sub-structure and fill in a table.
- We'll do that now.
  - Just for pedagogy!
  - (This isn't the best way to think about activity selection).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

Optimal substructure

- Subproblem i:
  - A[i] = number of activities you can do after Activity i finishes.

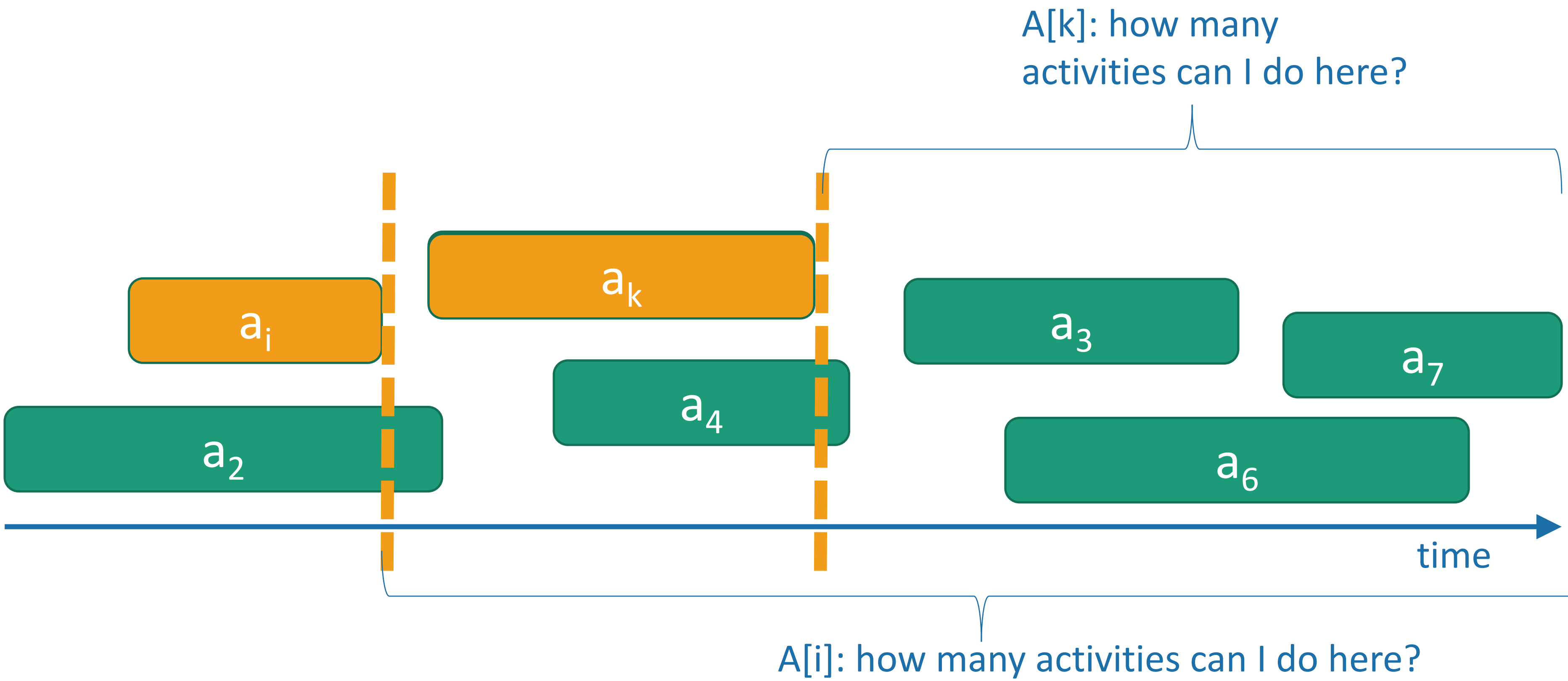


Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

We did that already

- Let a_k have the smallest finish time among activities do-able after a_i finishes.
- Then $A[i] = A[k] + 1$.



Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

See IPython notebook for implementation

Top-down DP

- Initialize a global array A to [None,…,None]
- Make a “dummy” activity that ends at time -1.
- **def** findNumActivities(i):
  - If A[i] != None:
    - **Return** A[i]
  - Let Activity k be the activity I can fit in my schedule after Activity i with the smallest finish time.
  - **If** there is no such activity k, set A[i] = 0
  - **Else,** A[i] = findNumActivities(k) + 1
  - **Return** A[i]
- **Return** findNumActivities(0)

This is a terrible way to write this!

The only thing that matters here is that the highlighted lines are our recursive relationship.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the value of the optimal solution.
- **Step 3:** Use dynamic programming to find the value of the optimal solution.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual solution.
- **Step 5:** If needed, code this up like a reasonable person.

See IPython notebook for implementation

Top-down DP

- Initialize a global array A to [None,…,None]
- Initialize a global array Next to [None, …, None]
- Make a "dummy" activity that ends at time -1.
- **def** findNumActivities(i):
    - If A[i] != None:
        - **Return** A[i]
    - Let Activity k be the activity I can fit in my schedule after Activity i with the smallest finish time.
    - **If** there is no such activity k, set A[i] = 0
    - **Else,** A[i] = findNumActivities(k) + 1 and Next[i] = k
    - **Return** A[i]
- findNumActivities(0)
- Step through "Next" array to get schedule.

This is a terrible way to write this!

The only thing that matters here is that the highlighted lines are our recursive relationship.

Let's step through it.

(See IPython notebook for code with some print statements)

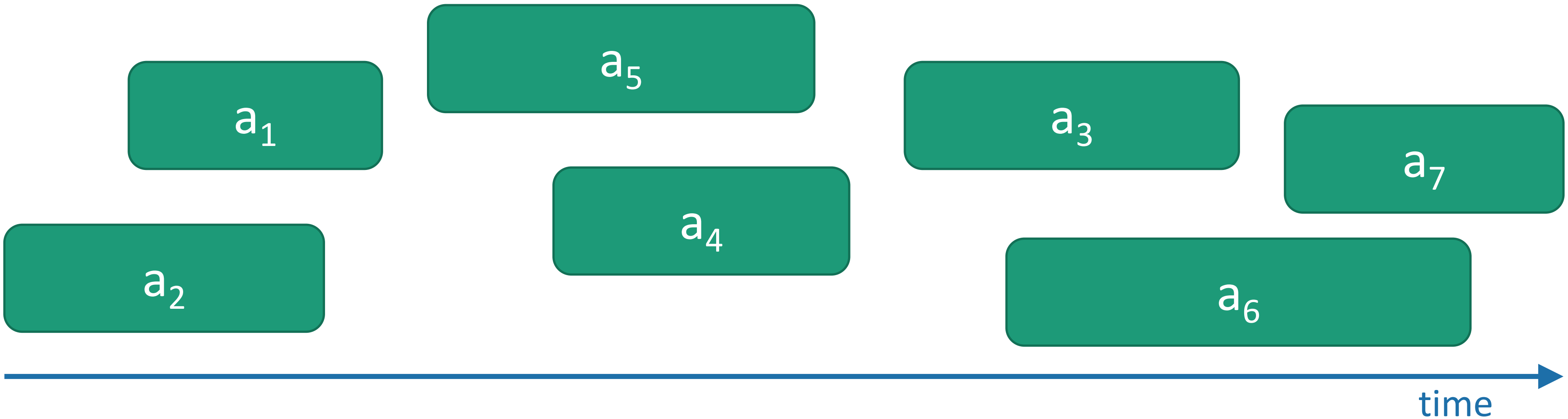
```
In [36]: activities = [ [-1,0], [1,4],[2,5],[3,6],[5,7],[3,8],[6,9],[8,10],[9,11],[5
         activityList = findBestSchedule(activities)
         print("---\n Solution:")
         for act in activityList:
             print(activities[act])

After activity [-1, 0] I am adding the next thing which is [1, 4]
After activity [1, 4] I am adding the next thing which is [5, 7]
After activity [5, 7] I am adding the next thing which is [8, 10]
After activity [8, 10] I am adding the next thing which is [13, 15]
---
 Solution:
[1, 4]
[5, 7]
[8, 10]
[13, 15]

In [ ]:
```

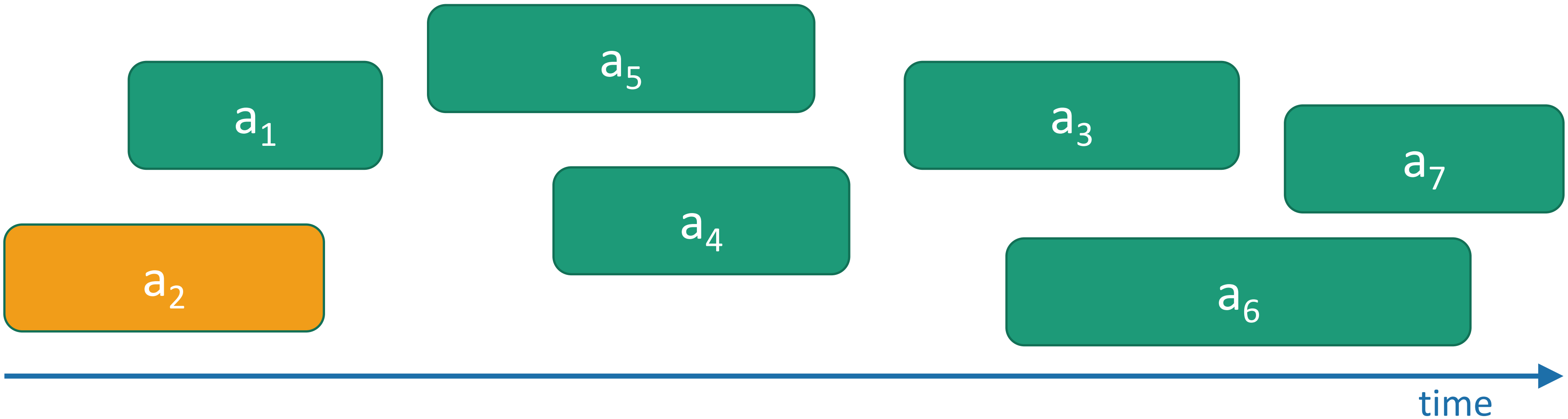
This looks pretty familiar!!

Let’s step through it.



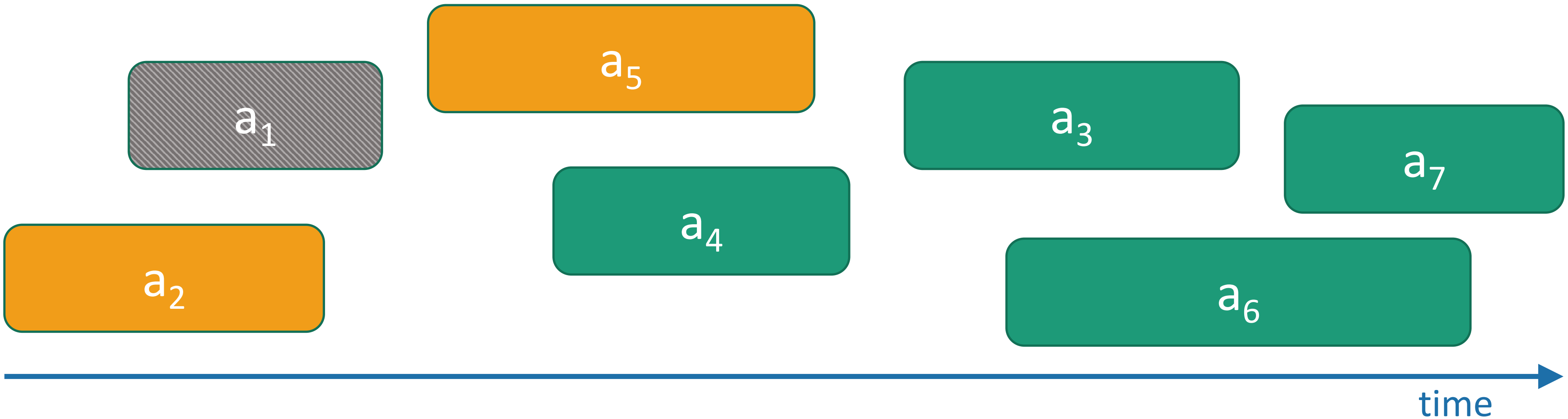
- Start with the activity with the smallest finish time.

Let's step through it



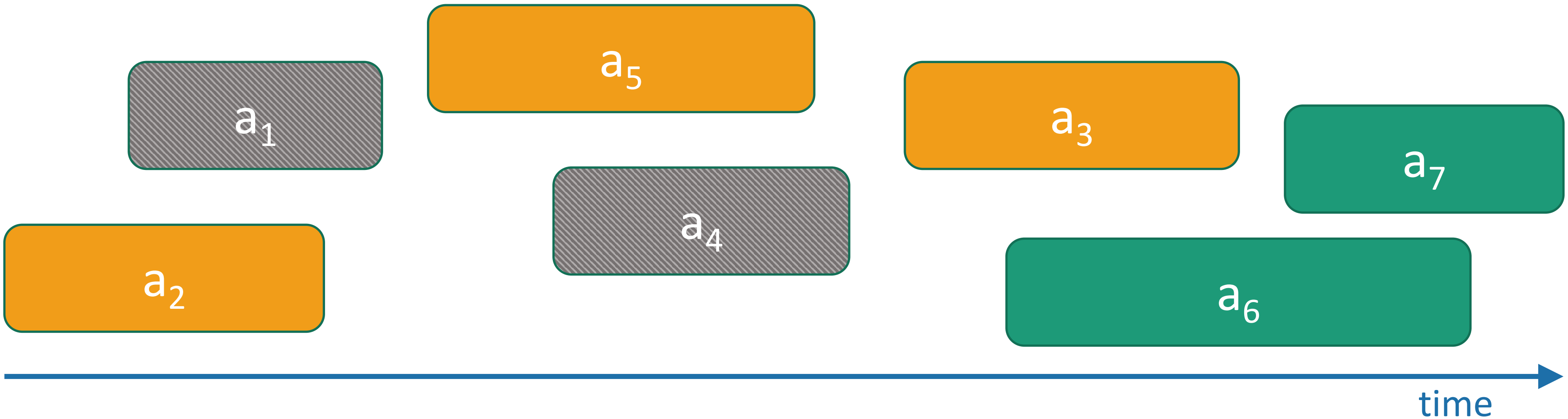
- Now find the next activity still do-able with the smallest finish time, and recurse after that.

Let’s step through it



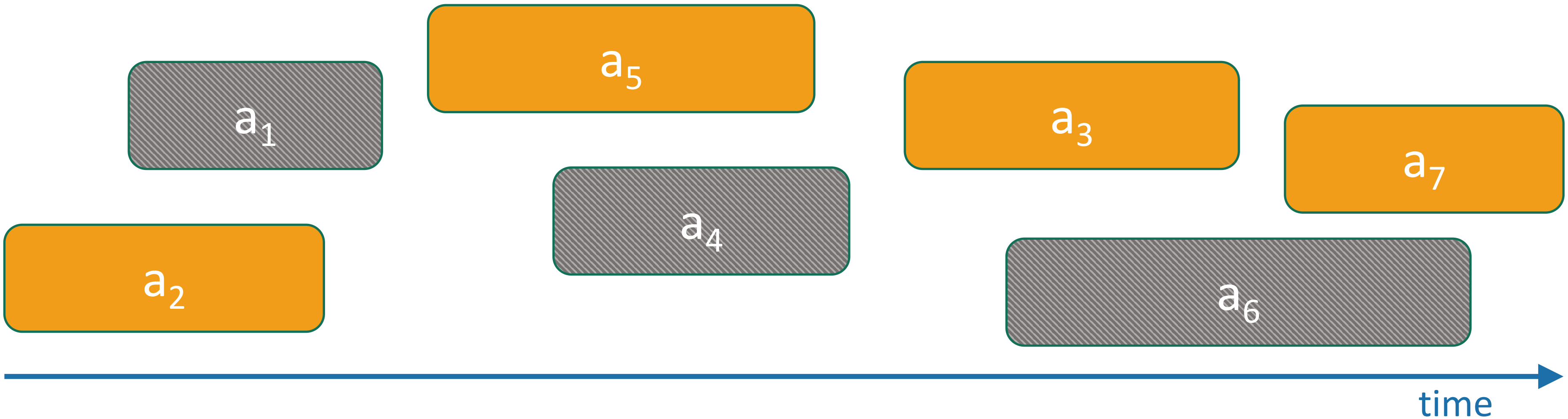
- Now find the next activity still do-able with the smallest finish time, and recurse after that.

Let’s step through it



- Now find the next activity still do-able with the smallest finish time, and recurse after that.

Let's step through it



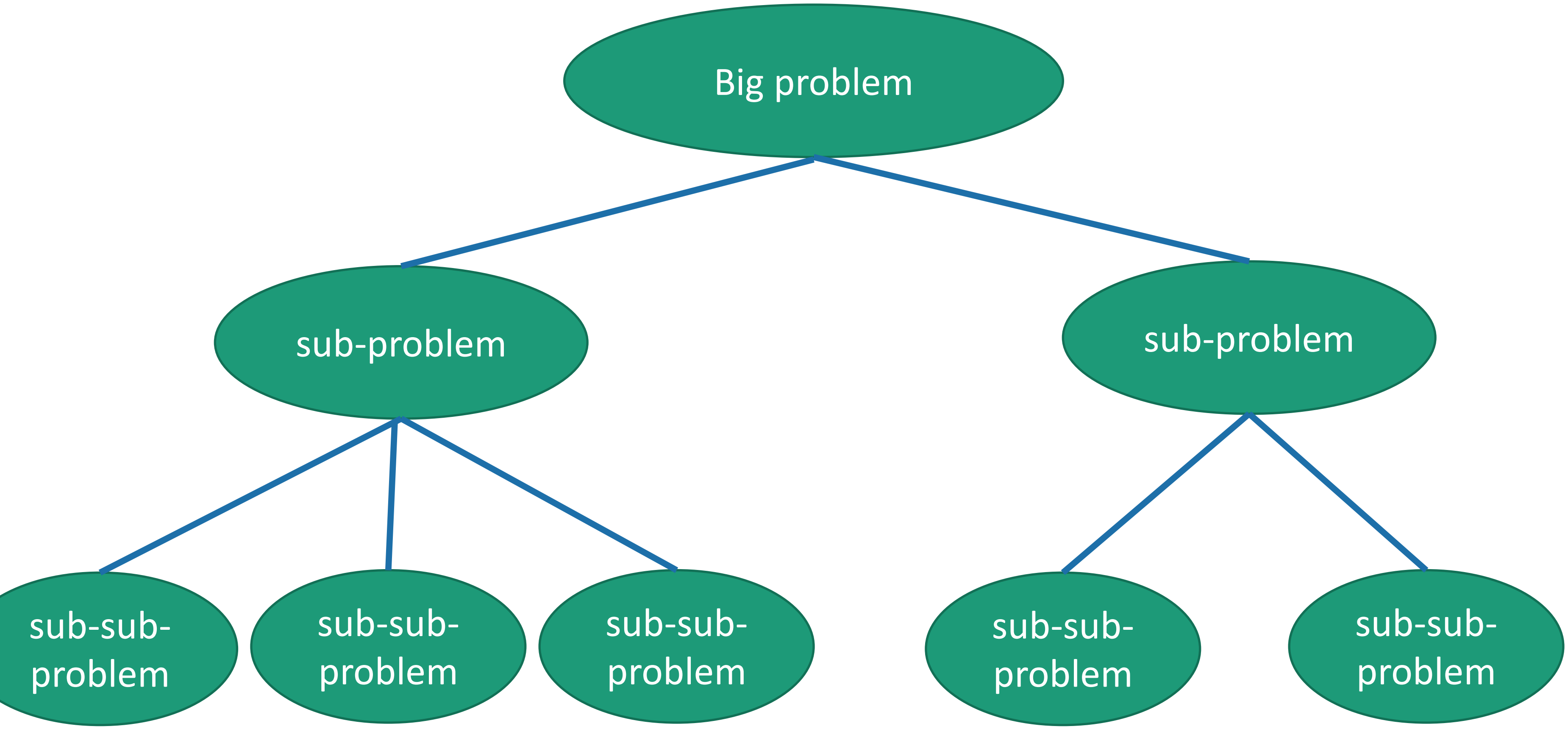
- Ta-da!

It's exactly the same* as the greedy solution!

*if you implement the top-down DP solution appropriately.

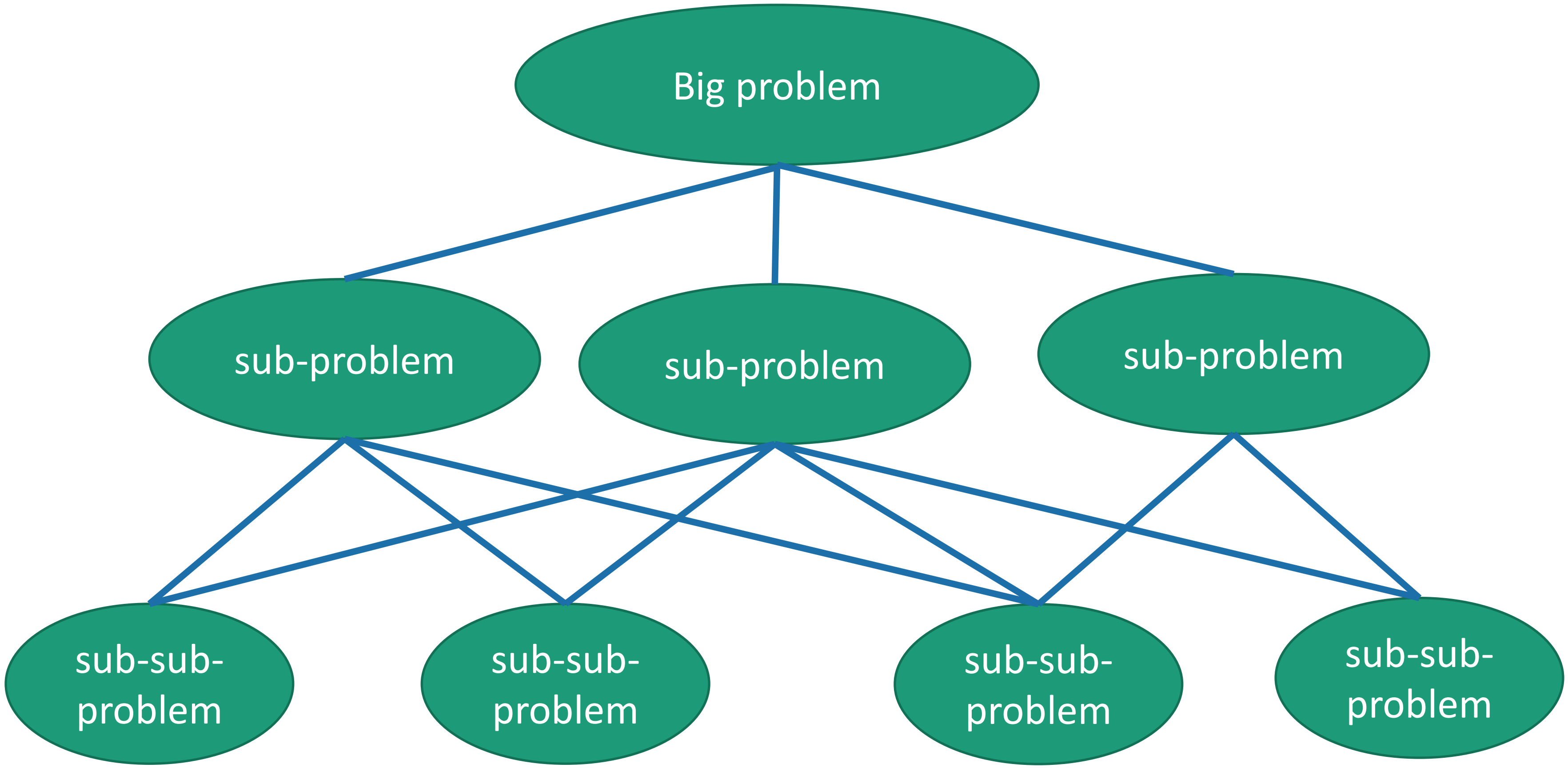
Sub-problem graph view

- Divide-and-conquer:



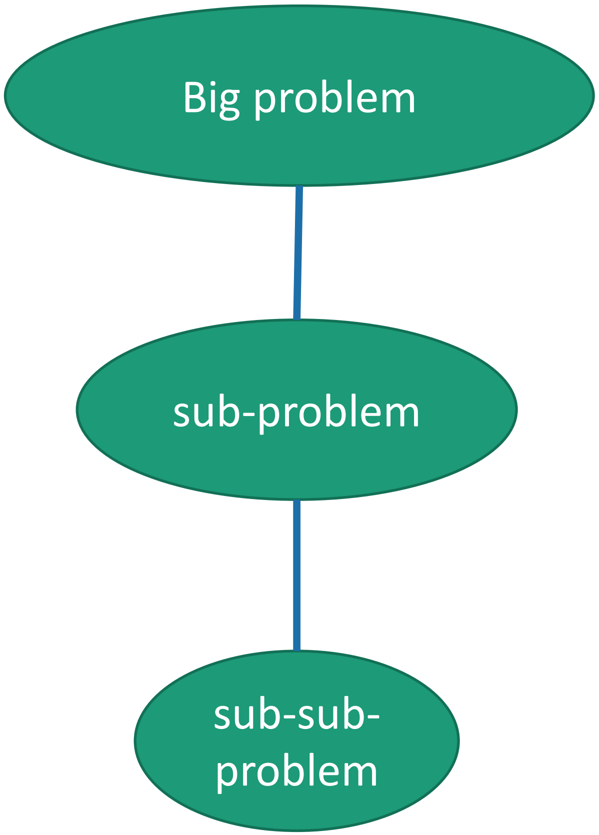
Sub-problem graph view

- Dynamic Programming:



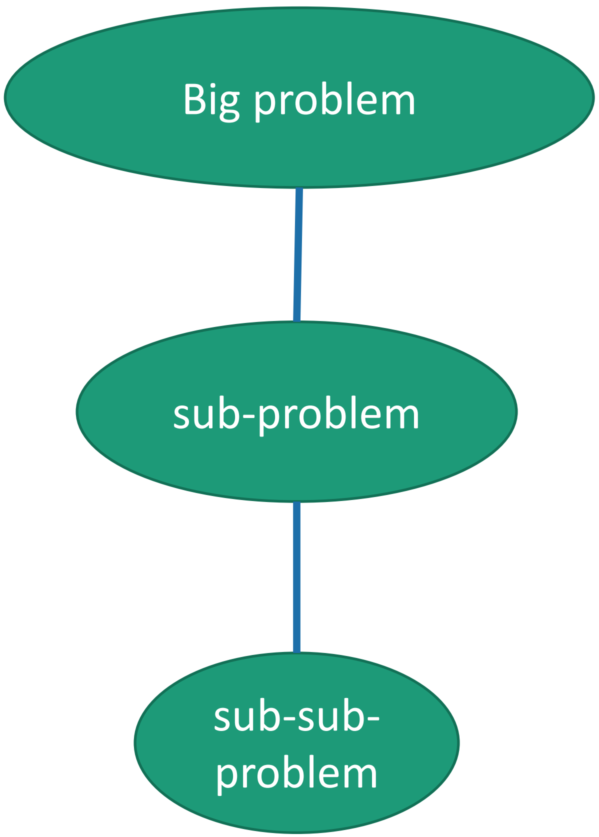
Sub-problem graph view

- 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**.

Answers

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?
     - Related to dynamic programming! (Which we did in Week 7).
     - Proving that greedy algorithms work is often not so easy.

Let’s see a few more examples

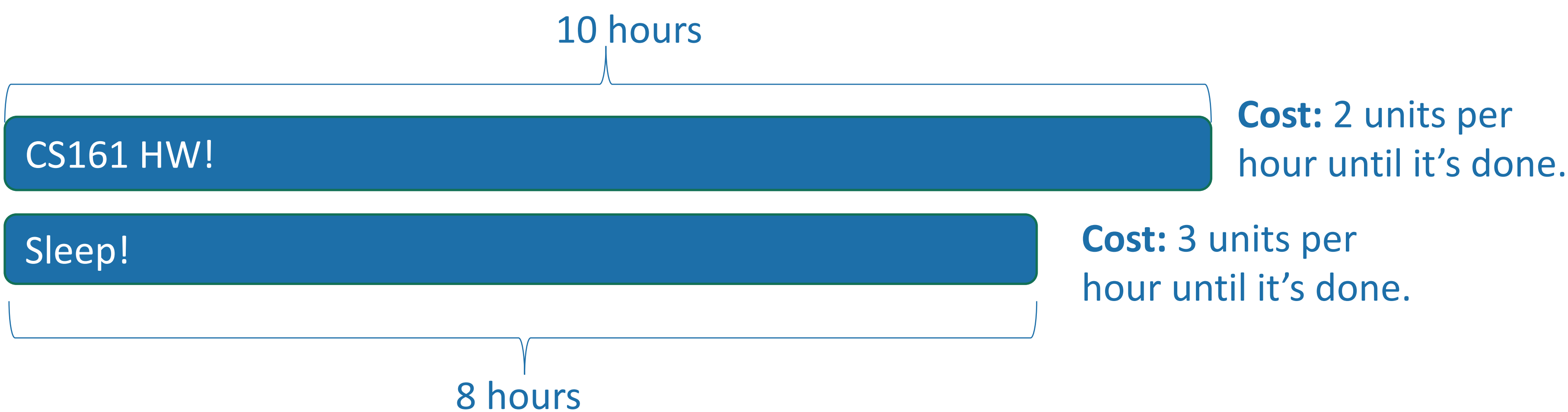
Another example:
Scheduling

- CS161 HW!
- Call your parents!
- Math HW!
- Administrative stuff for your student club!
- Econ HW!
- Do laundry!
- Meditate!
- Practice musical instrument!
- Read CLRS!
- Have a social life!
- Sleep!

Overcommitted
Stanford Student

Scheduling

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

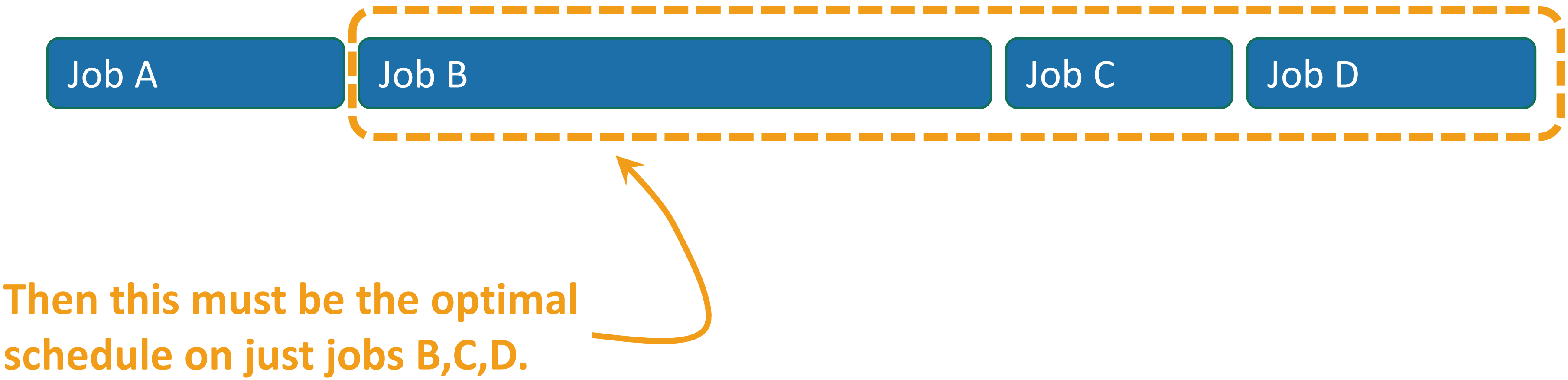


- CS161 HW, then Sleep: costs $\mathbf{10 \cdot 2 + (10 + 8) \cdot 3 = 74}$ **units**
- Sleep, then CS161 HW: costs $\mathbf{8 \cdot 3 + (10 + 8) \cdot 2 = 60}$ **units**

Optimal substructure

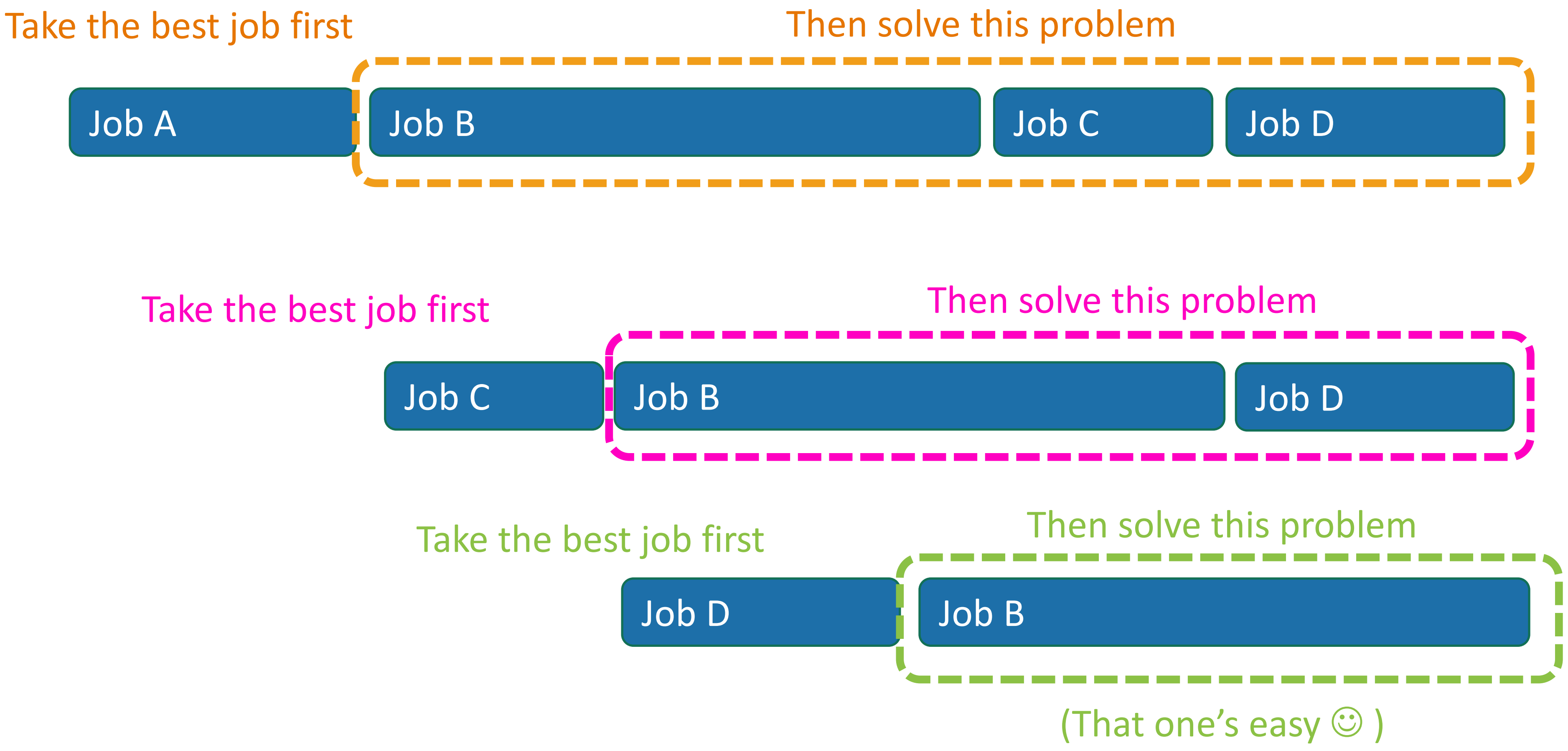
- This problem breaks up nicely into sub-problems:

Suppose this is the optimal schedule:



Optimal substructure

- Seems amenable to a greedy algorithm:



What does "best" mean?

- Recipe for greedy algorithm analysis:
  - We make a **series of choices**.
  - We 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.**

"Best" means: **won't rule out an optimal solution.**

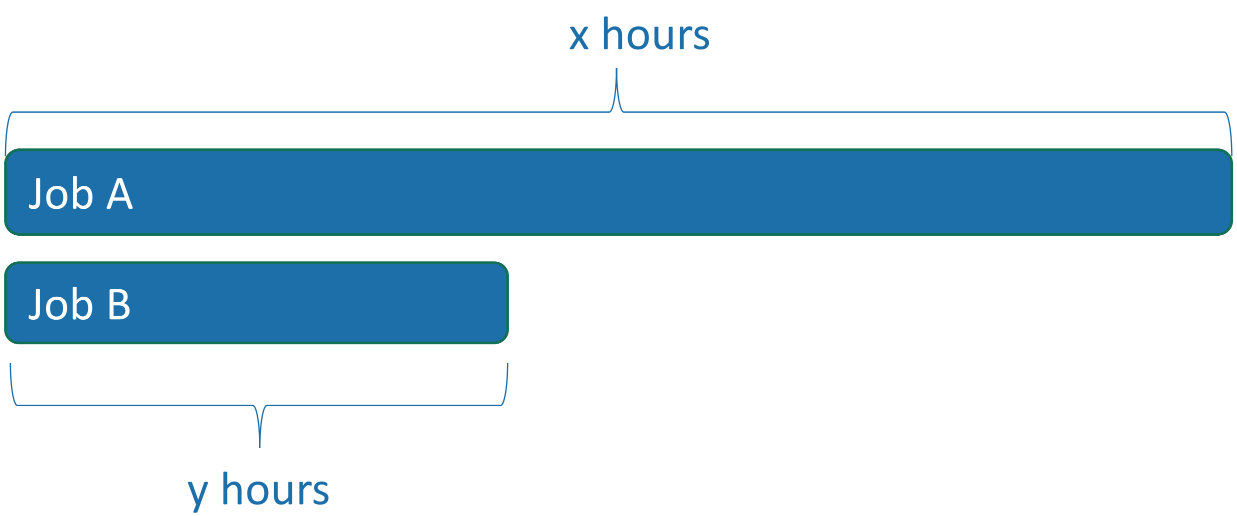
The optimal solution to this problem extends an optimal solution to the whole thing.

Head-to-head

A then B is better than **B then A** 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:

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

Do the job with the biggest ratio first.

Lemma

- Given jobs so that **Job i** takes time t_i with cost c_i ,
- There is an optimal schedule so that the first job is the one that maximizes the ratio c_i/t_i

- **Proof:**
  - Say Job B maximizes this ratio, and it’s not first:

$c_A/t_A >= c_B/t_B$

  - Switch A and B! Nothing else will change, and we showed on the previous slide that the cost won’t increase.

  - Repeat until B is first.

Choose greedily:
Biggest cost/time ratio first

- **Job i** takes time t_i with cost c_i
- There is an optimal schedule so that the first job is the one that maximizes the ratio c_i/t_i

- So if we choose jobs greedily according to c_i/t_i, we never rule out success!

Greedy Scheduling Solution

- **scheduleJobs**(JOBS):
  - Sort JOBS by the ratio:
    - $r_i = \frac{c_i}{t_i} = \frac{\text{cost of delaying job i}}{\text{time job i takes to complete}}$
  - Say that **sorted_JOBS[i]** is the job with the i'th biggest r_i
  - **Return sorted_JOBS**

The running time is O(nlog(n))

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

Formally, use induction!

SLIDE SKIPPED IN CLASS

- **Inductive hypothesis**:
  - There is an optimal ordering so that the first t jobs are **sorted_JOBS**[:t].
- **Base case**:
  - When t=0, this reads: "There is an optimal ordering so that the first 0 jobs are []"
  - That's true.
- **Inductive Step:**
  - Boils down to: there is an optimal ordering on sorted_JOBS[t:] so that sorted_JOBS[t] is first.
  - This follows from the Lemma.
- **Conclusion:**
  - When t=n, this reads: "There is an optimal ordering so that the first n jobs are **sorted_JOBS**."
  - aka, what we returned is an optimal ordering.

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 "safe" choices that **won't rule out an optimal solution.**
    - largest 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

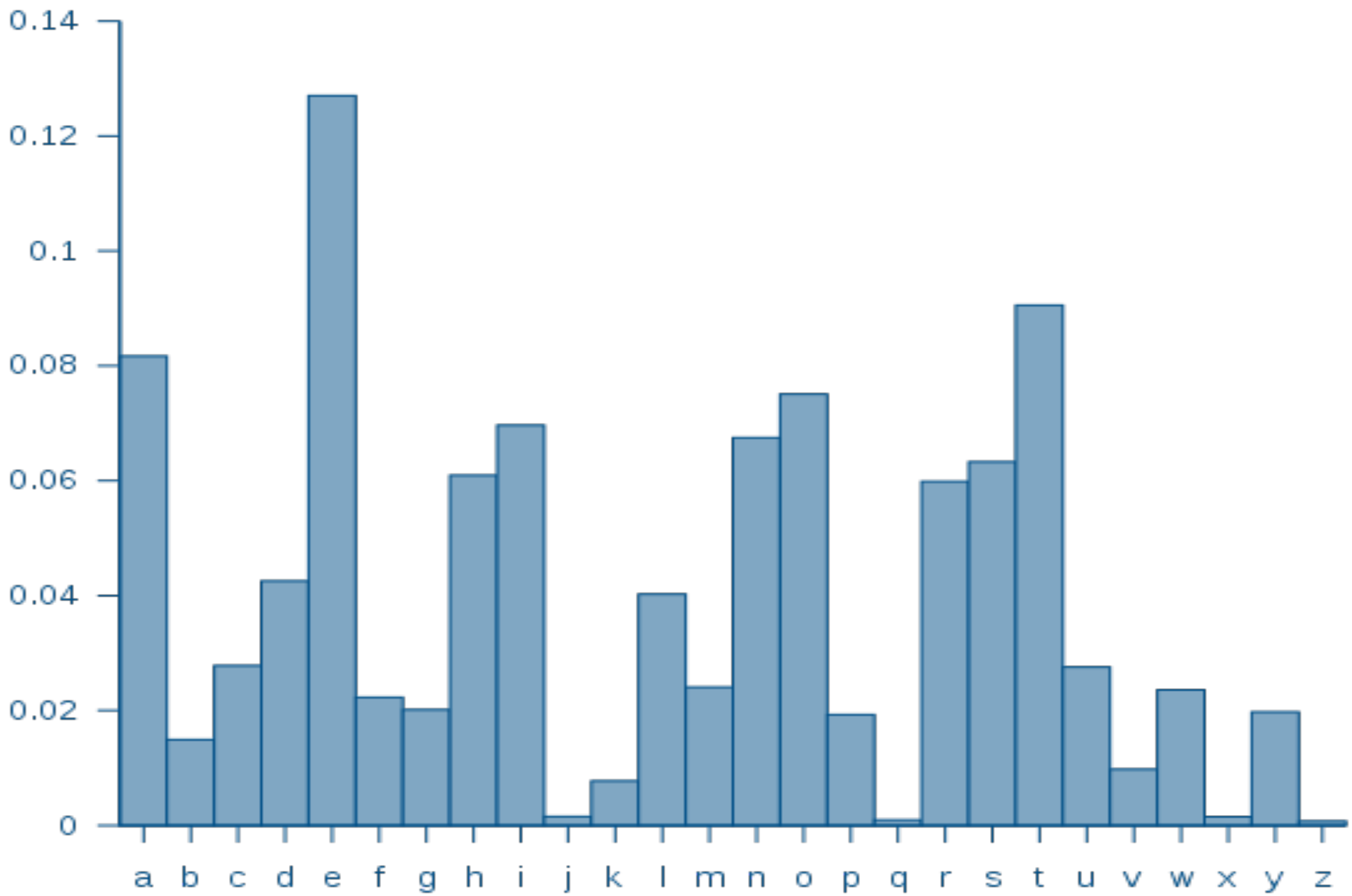
One more example
Huffman coding

ASCII is pretty wasteful. If **e** shows up so often, we should have a more parsimonious way of representing it!

- `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**

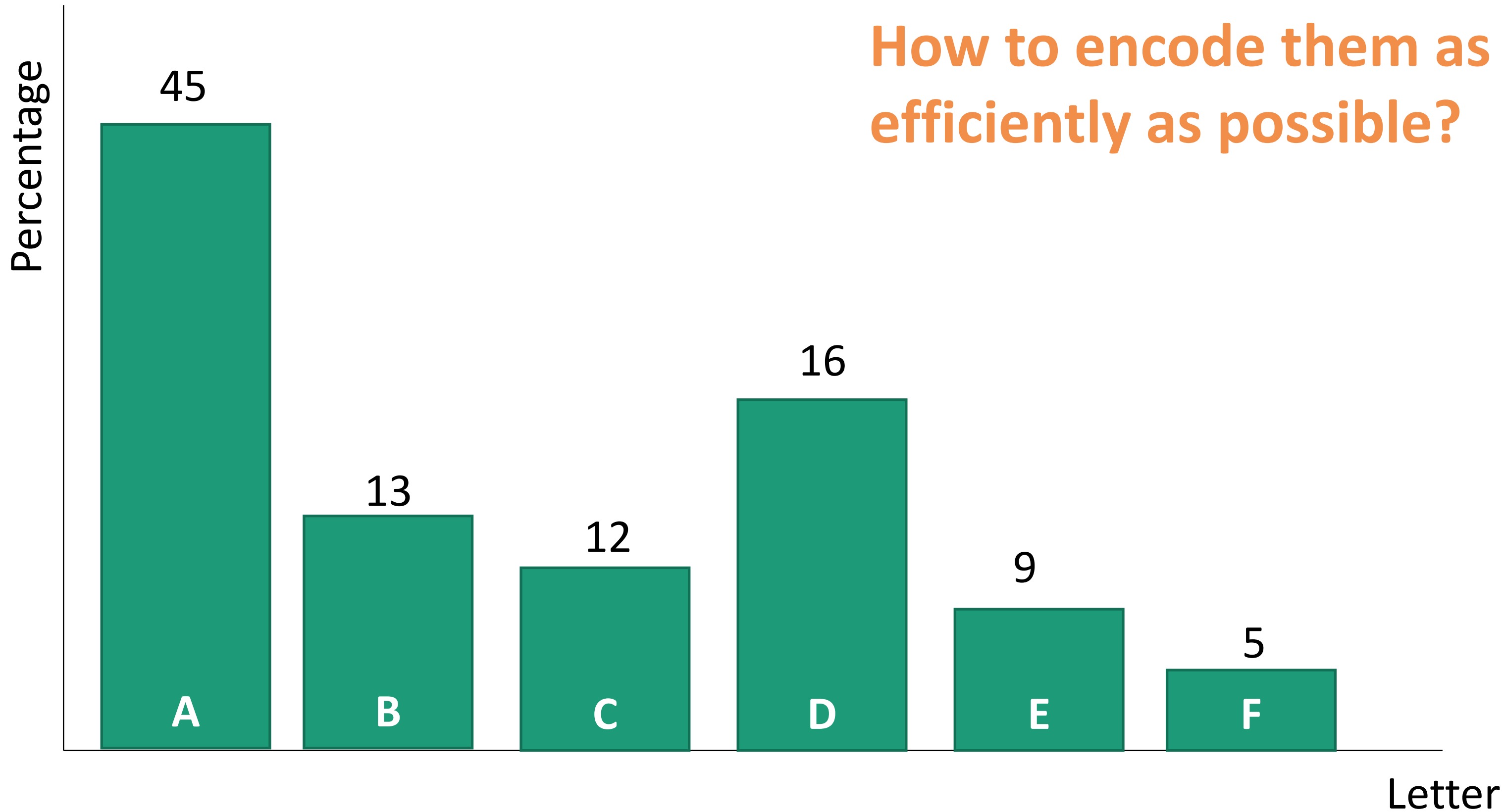
- `qwertyui_opasdfg+hjklzxcv`
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Suppose we have some distribution on characters



Suppose we have some distribution on characters

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



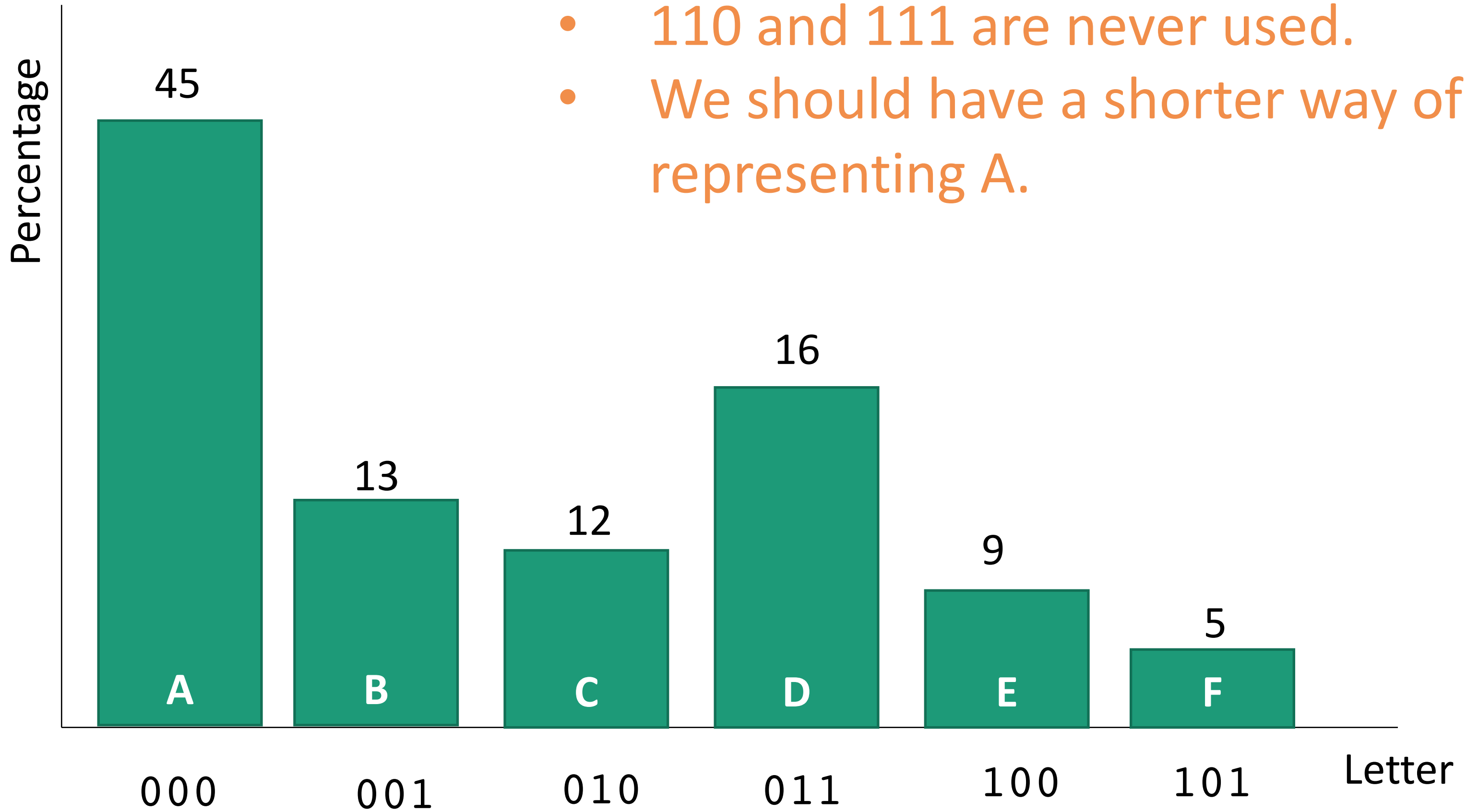
How to encode them as efficiently as possible?

Try 0
(like ASCII)

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

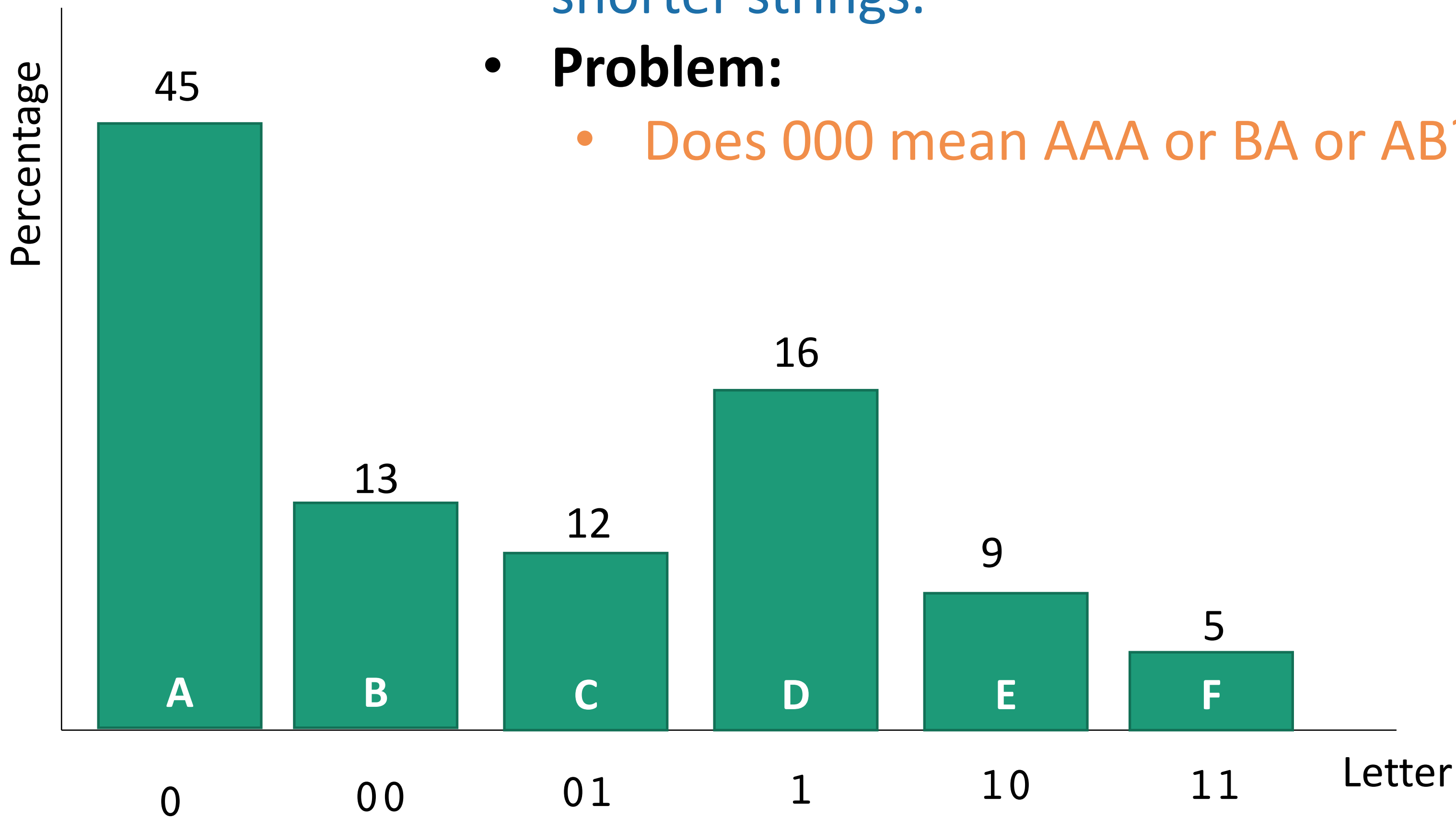
Wasteful!

- 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?

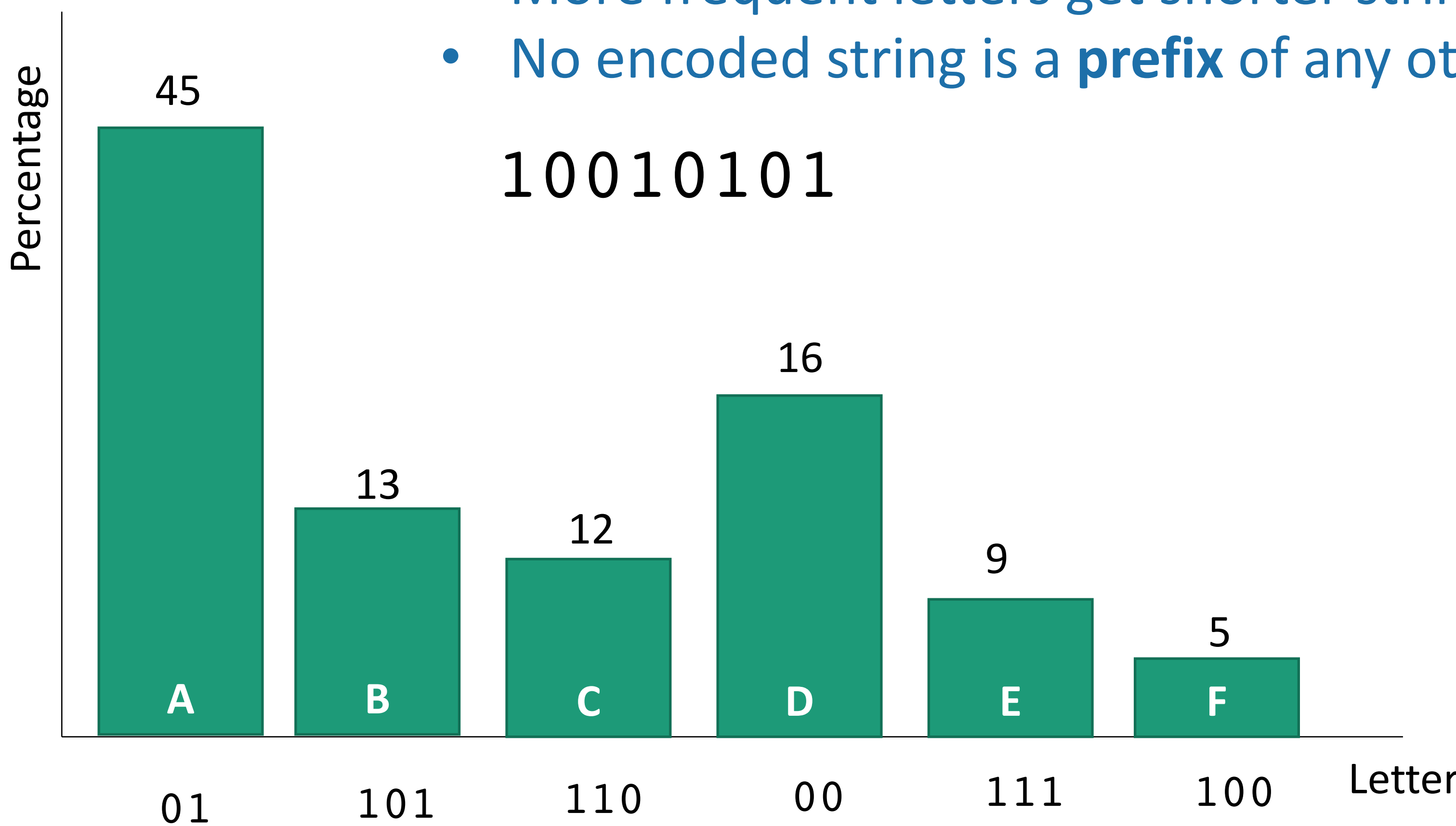


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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

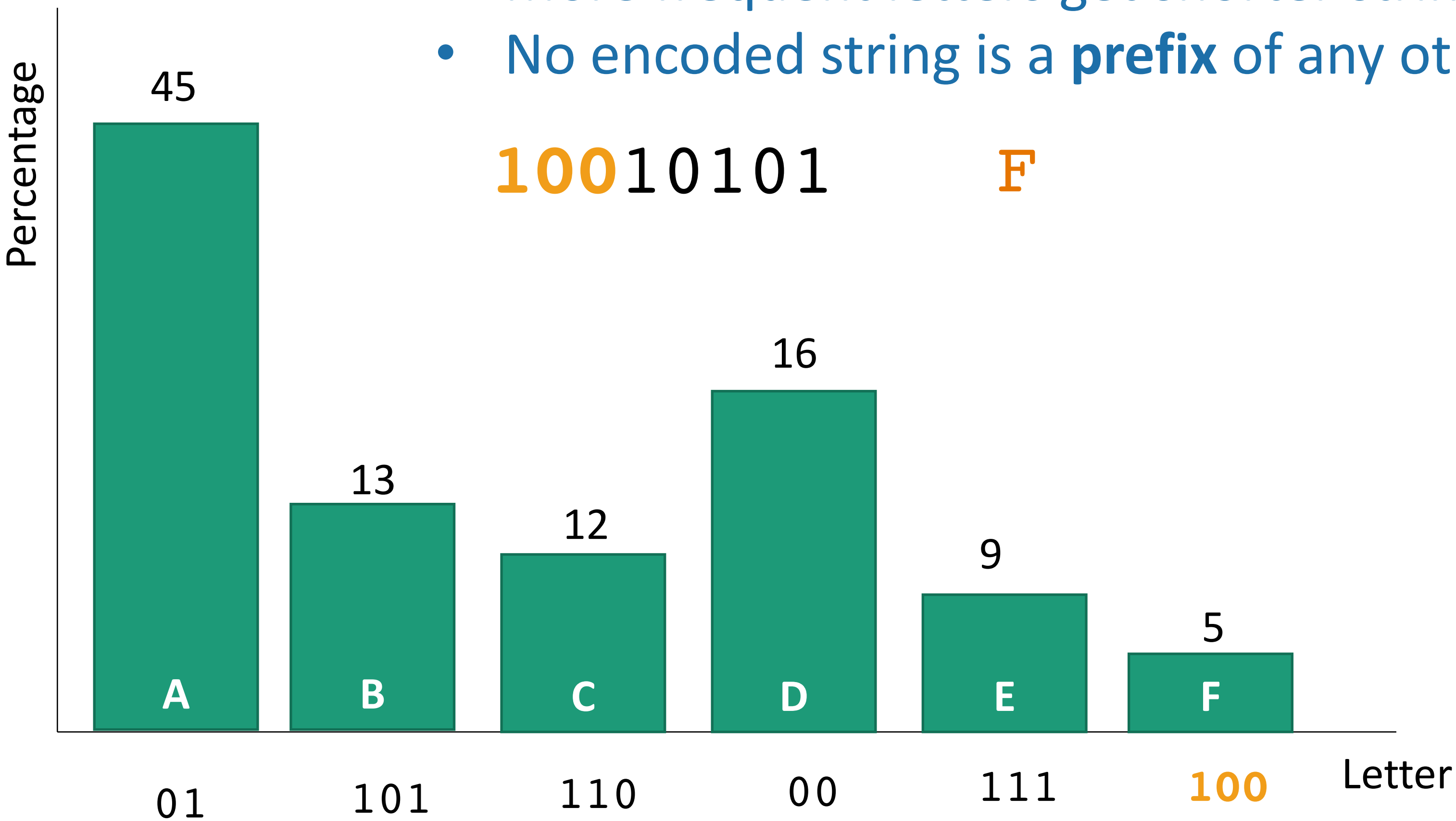


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

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**

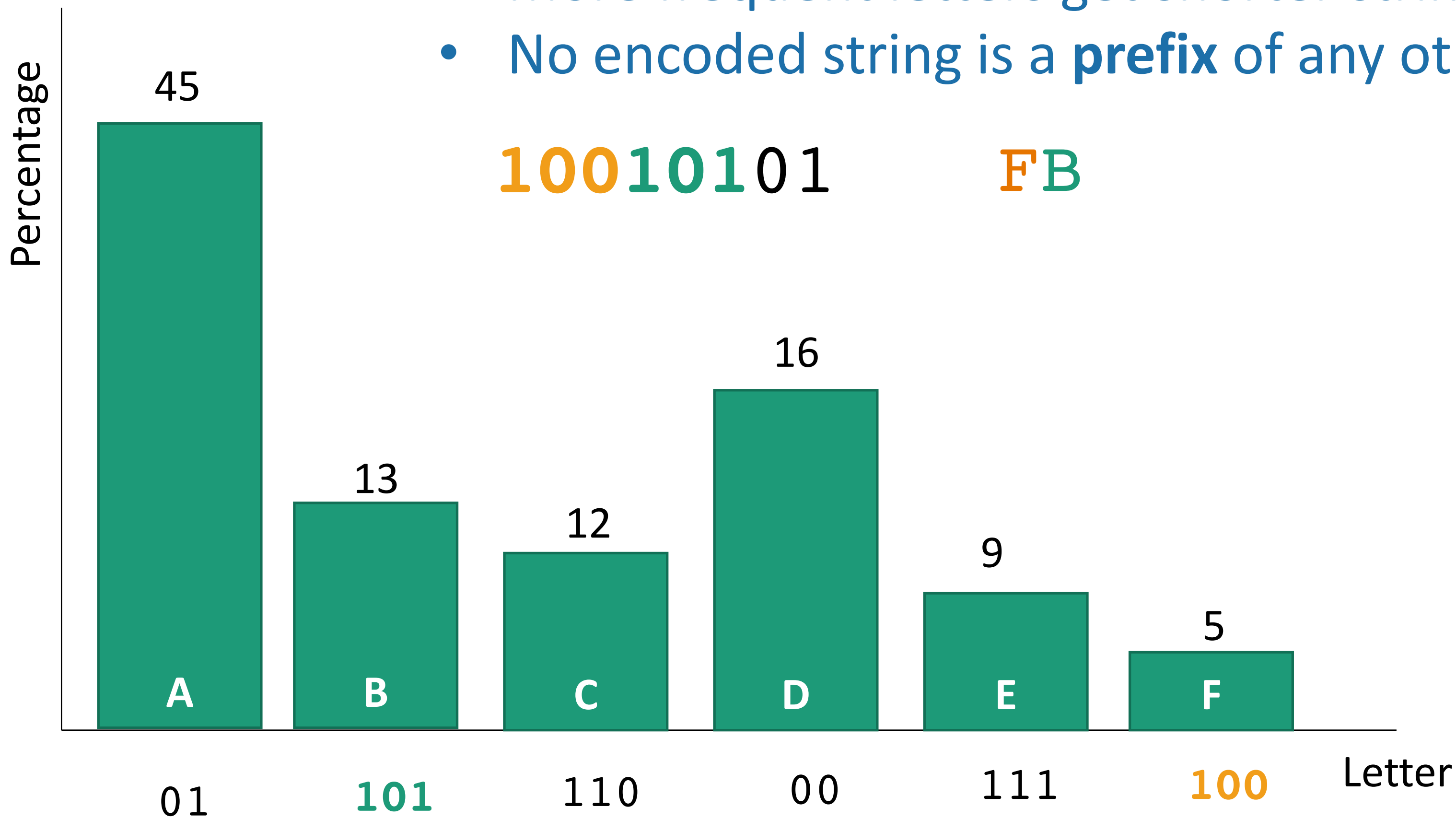


Confusingly, "prefix-free codes" are also sometimes called "prefix codes" (including in CLRS).

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

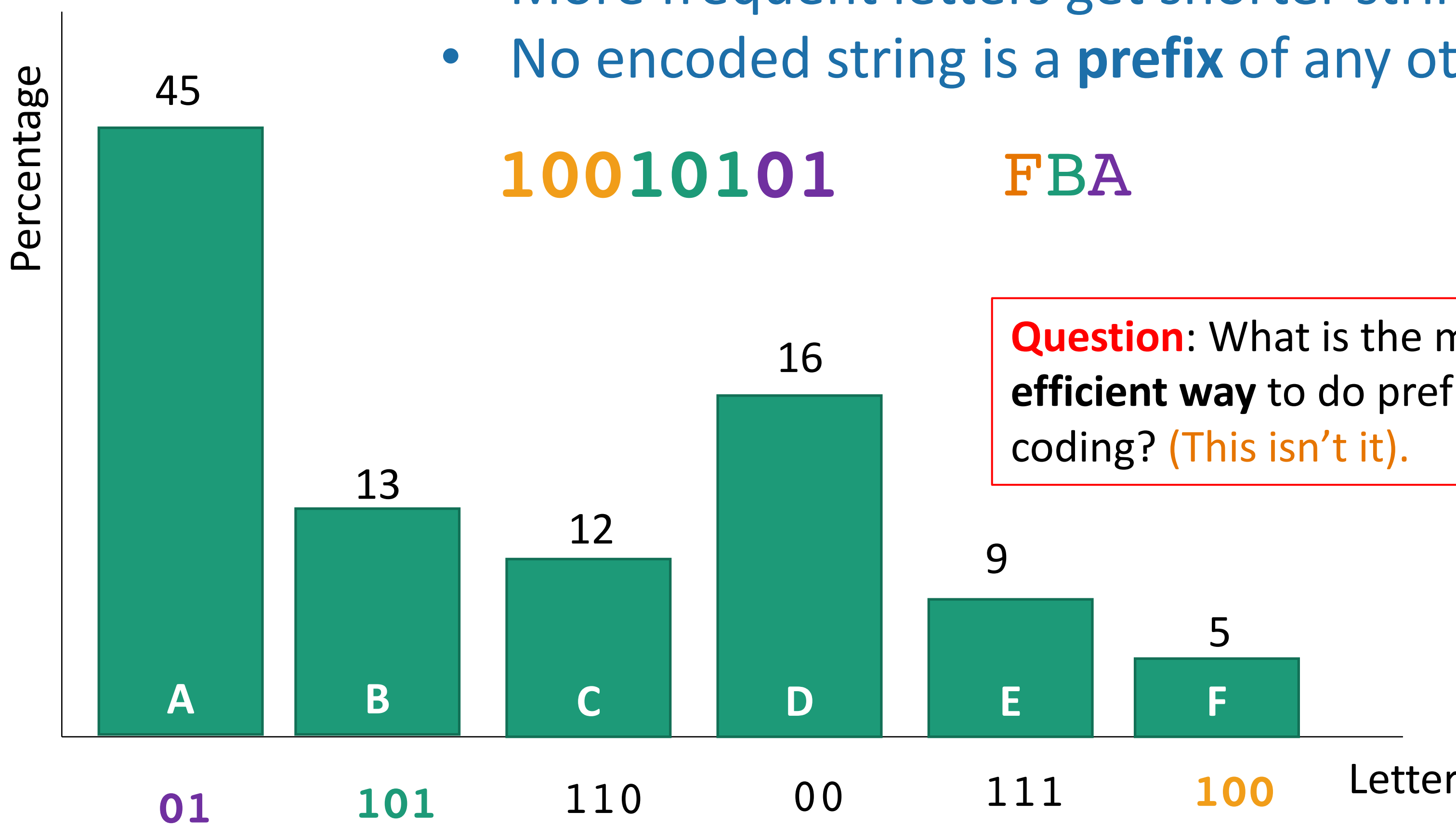


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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.

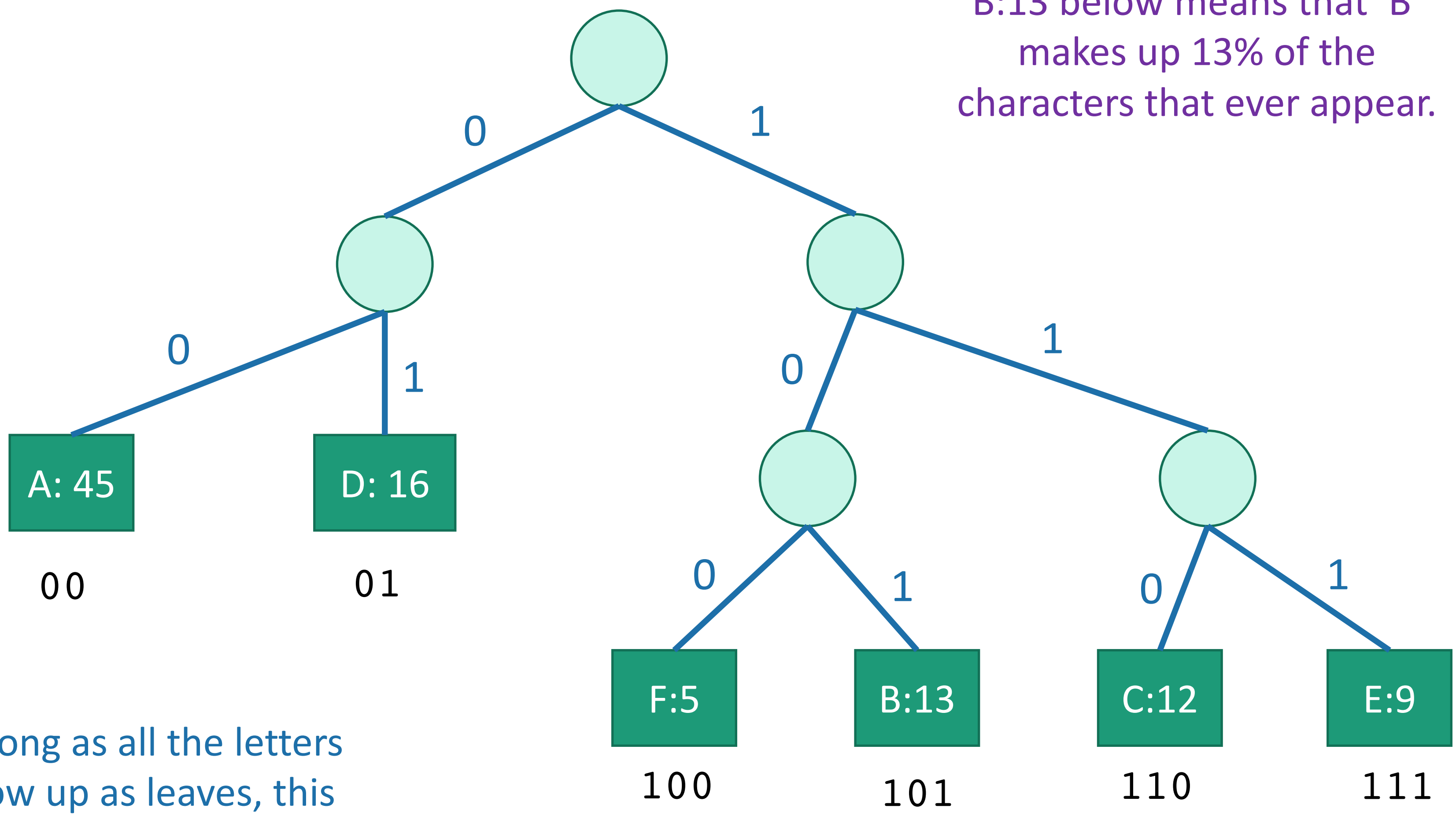
10010101 FBA



Question: What is the most **efficient way** to do prefix-free coding? (This isn’t 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**.

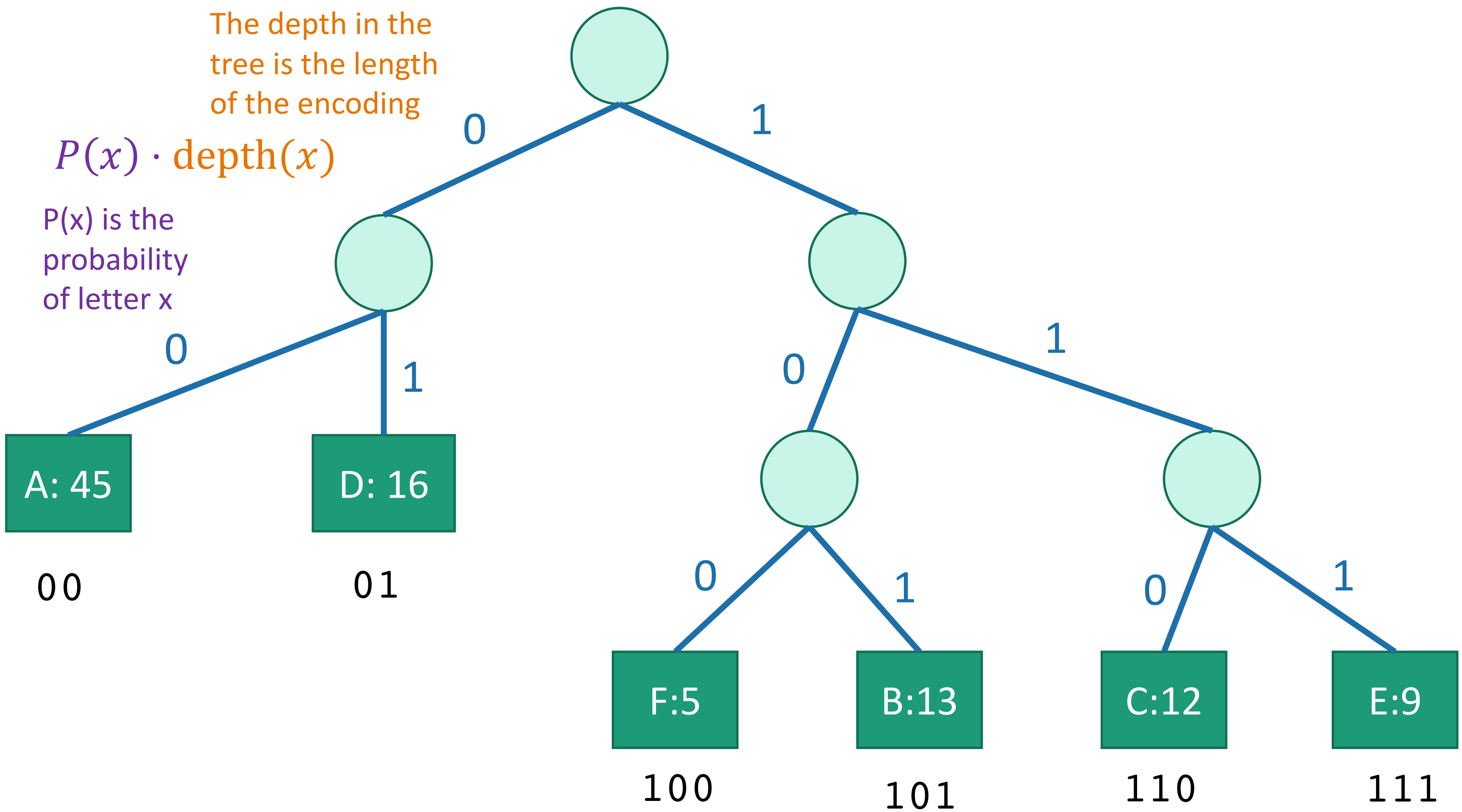
Some trees are better than others

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

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

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

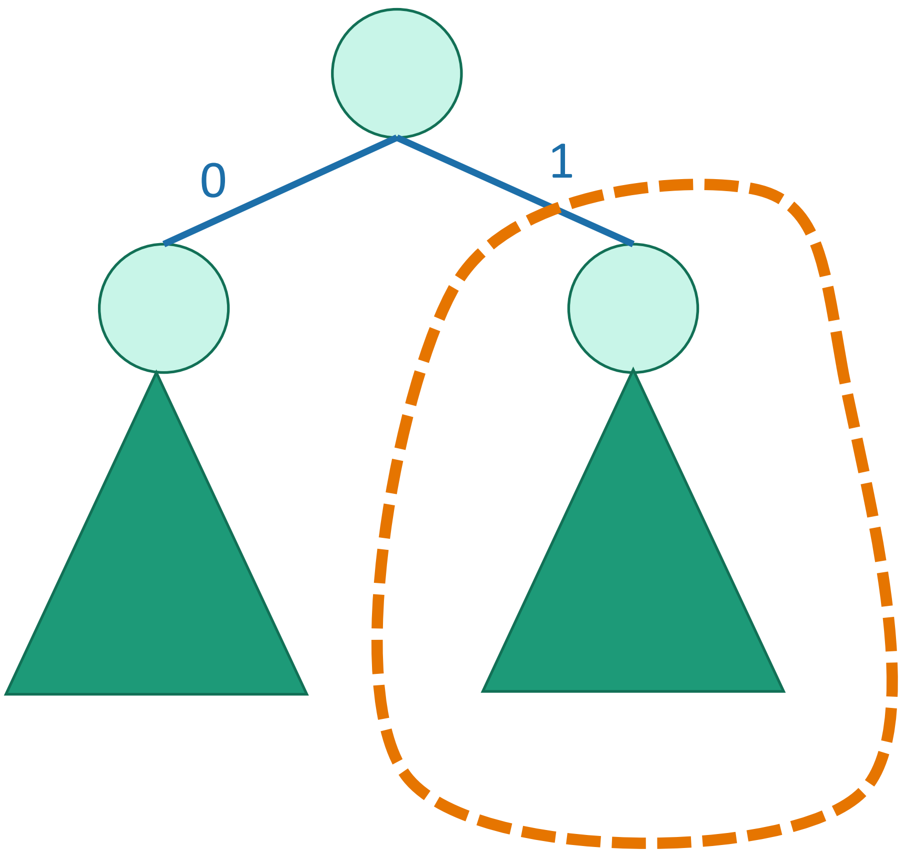
$$\text{cost(tree)} = \sum_{\text{leaves } x} P(x) \cdot \text{depth}(x)$$

P(x) is the probability of letter x

The depth in the tree is the length of the encoding

Optimal sub-structure

- Suppose this is an optimal tree:

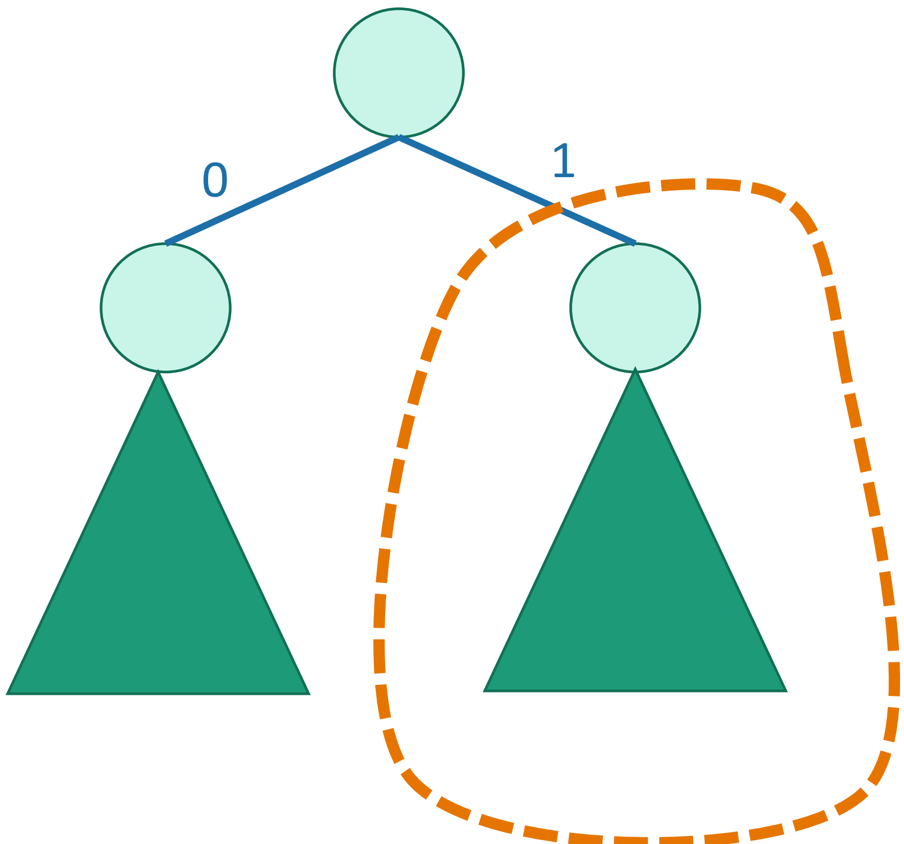


Then this is an optimal tree on fewer letters.

Otherwise, we could change this sub-tree and end up with a better overall tree.

In order to design a greedy algorithm

- Think about what letters belong in this sub-problem...



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

Infrequent elements!

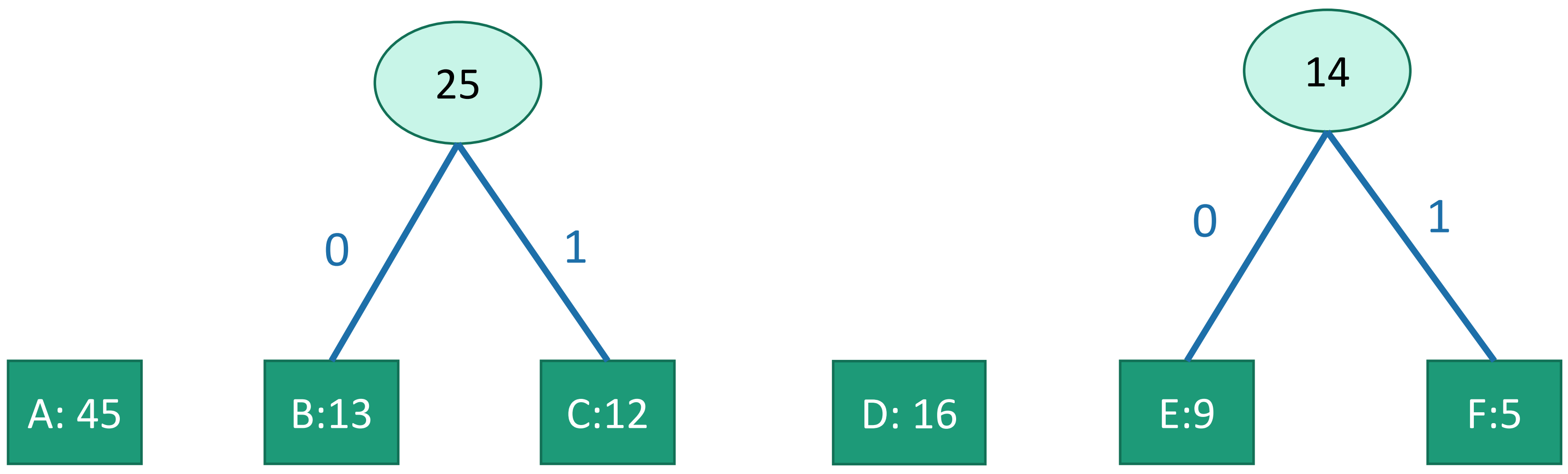
We want them as low down as possible.

Solution

greedily build subtrees, starting with the infrequent letters

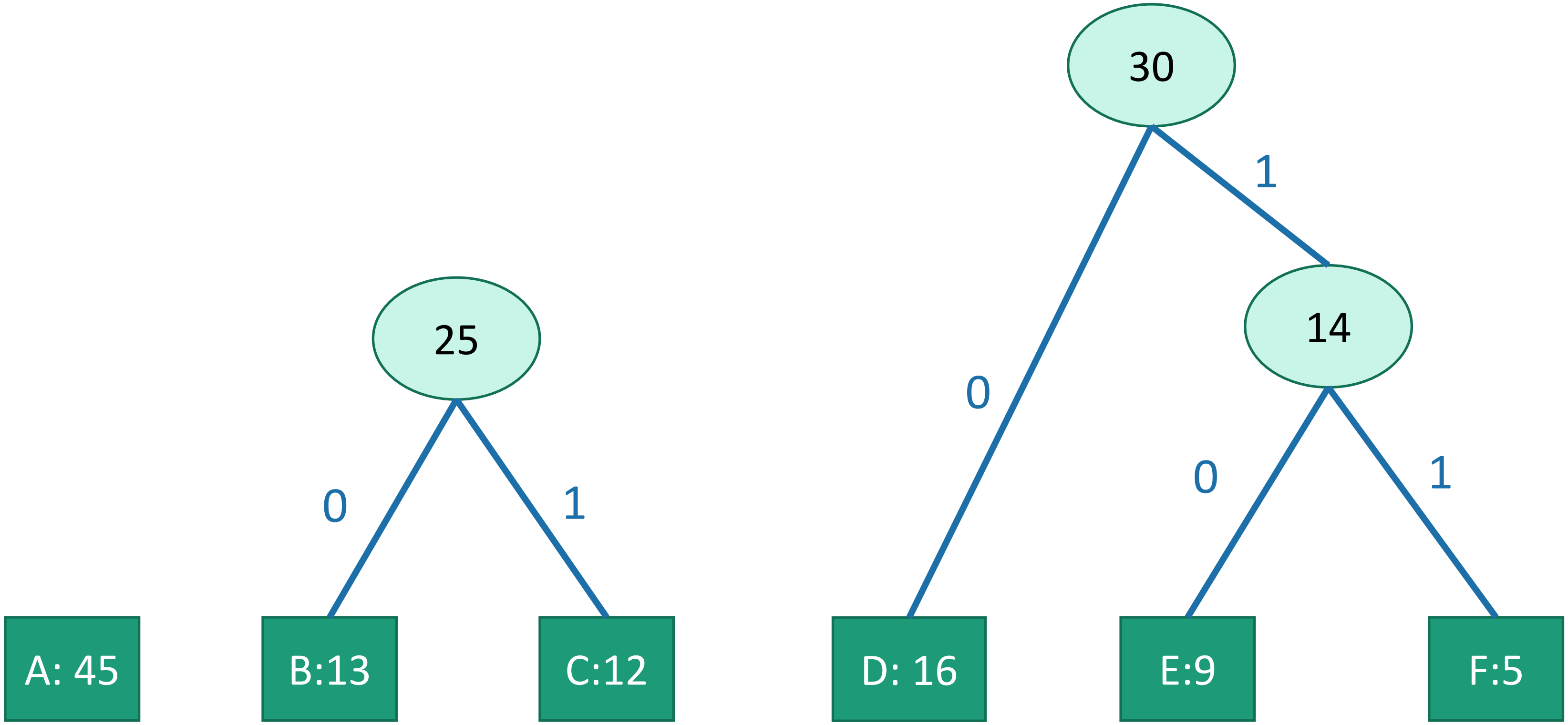
Solution

greedily build subtrees, starting with the infrequent letters



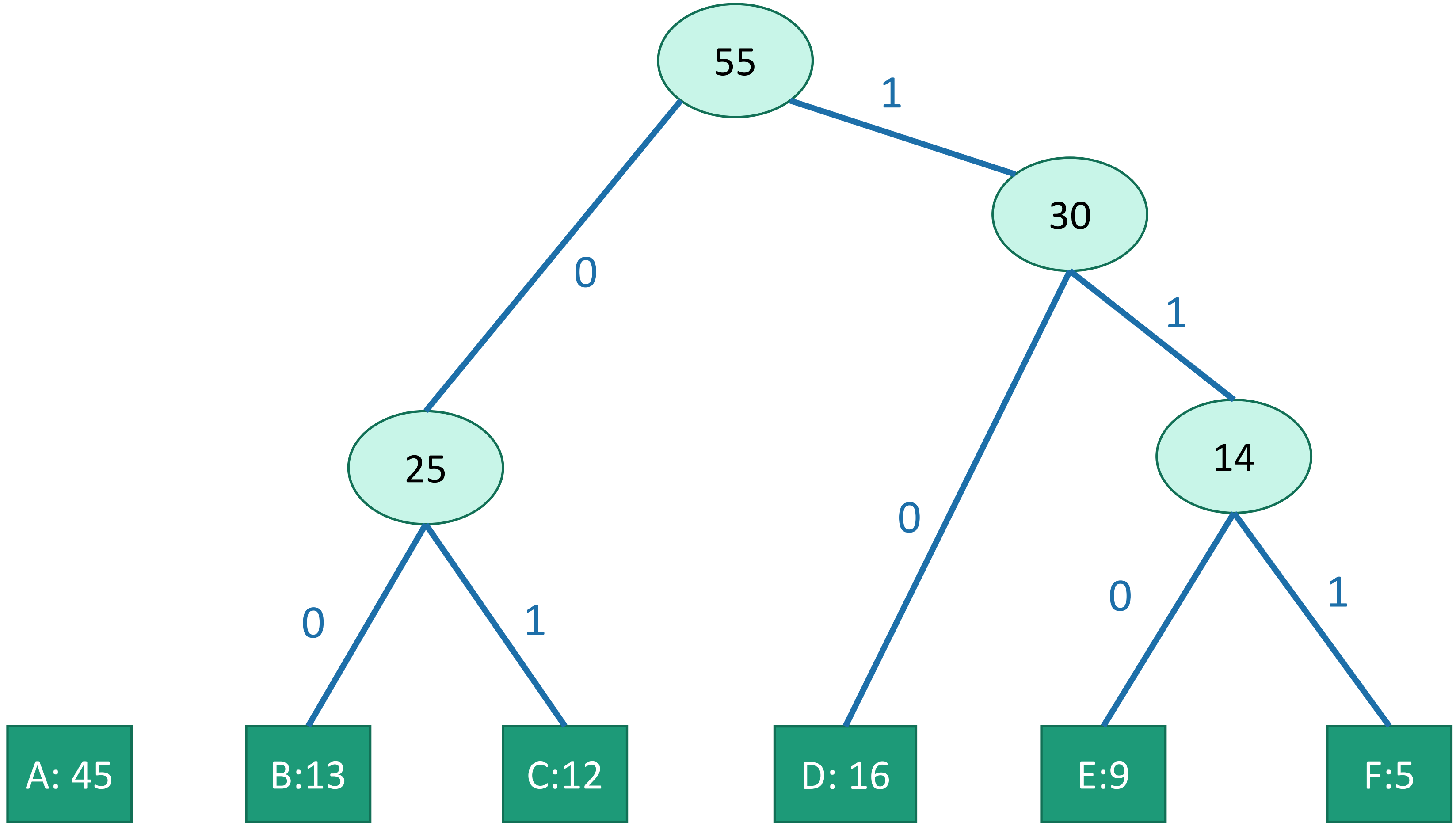
Solution

greedily build subtrees, starting with the infrequent letters



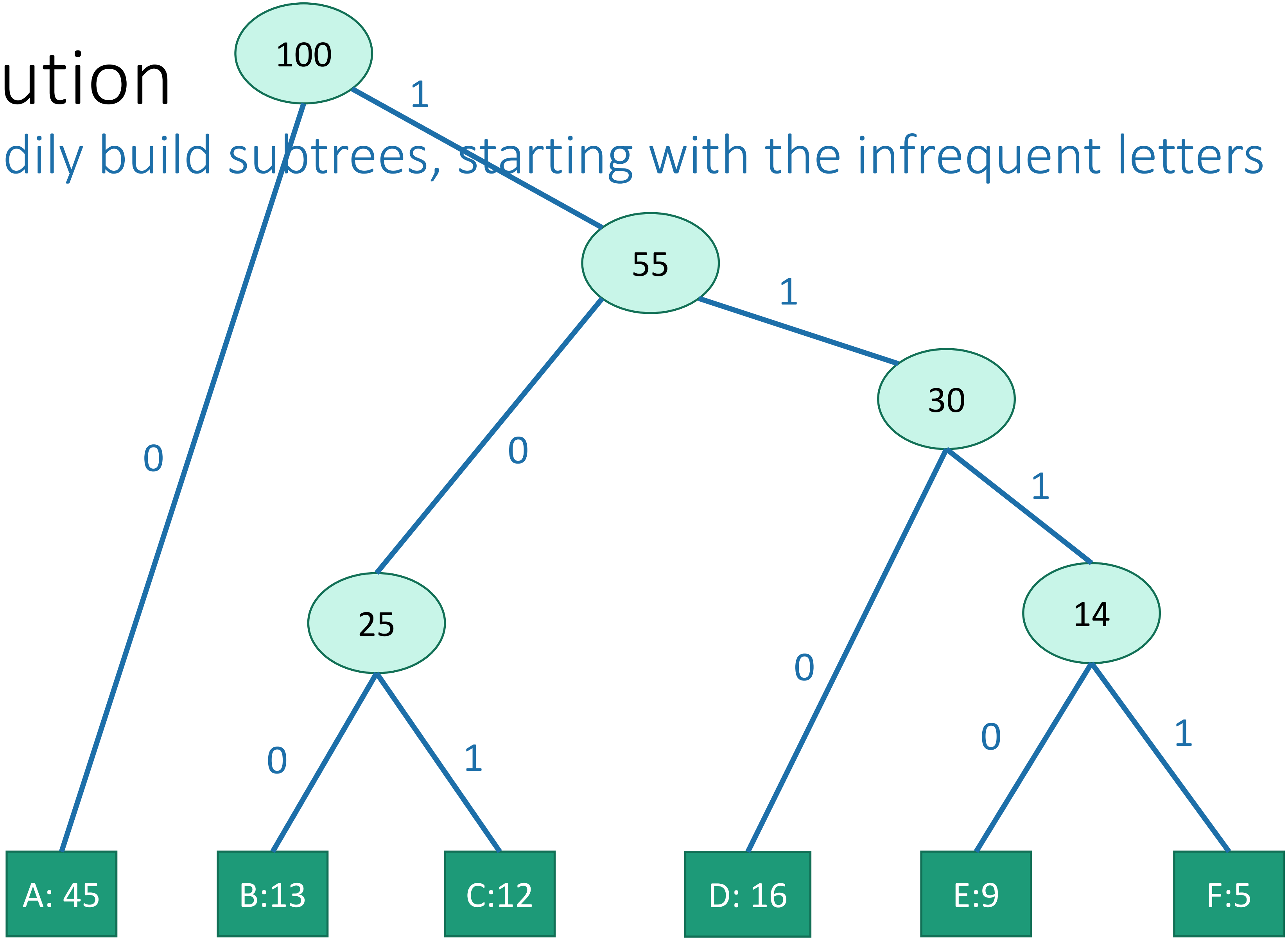
Solution

greedily build subtrees, starting with the infrequent letters



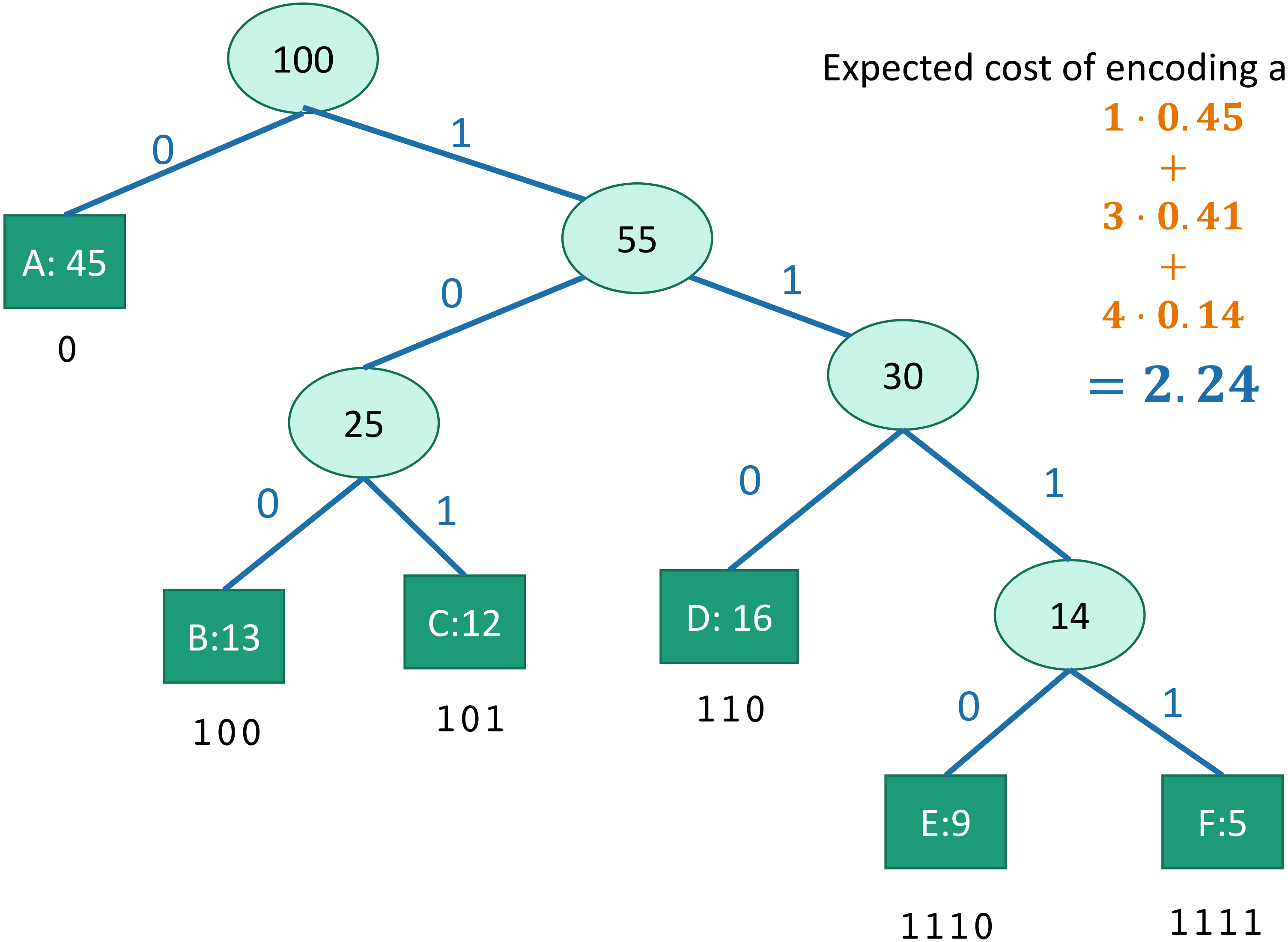
Solution

greedily build subtrees, starting with the infrequent letters



Solution

greedily build subtrees, starting with the infrequent letters



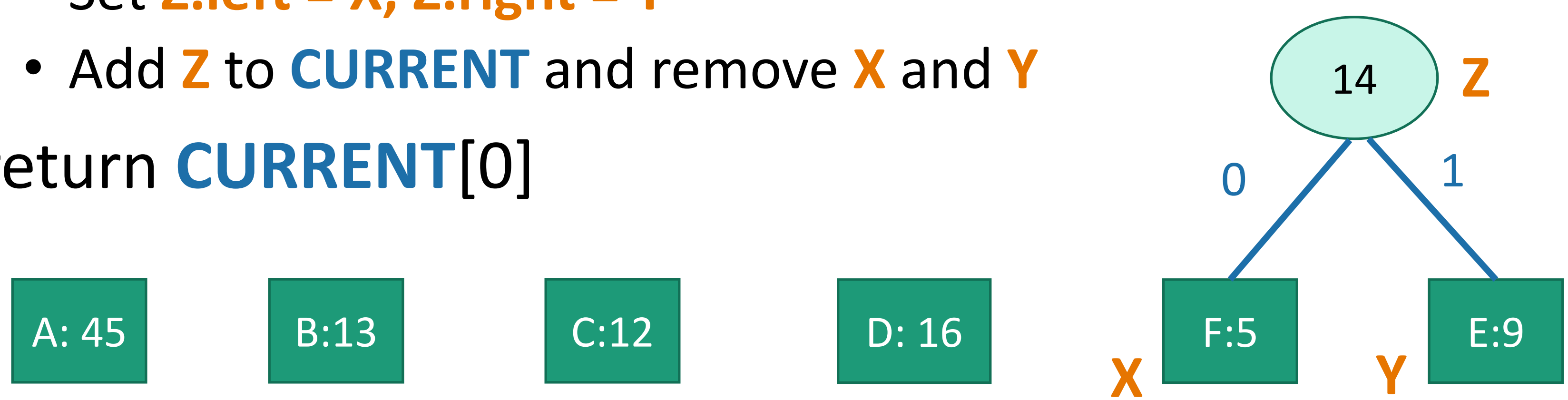
Expected cost of encoding a letter:

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

$$= \mathbf{2.24}$$

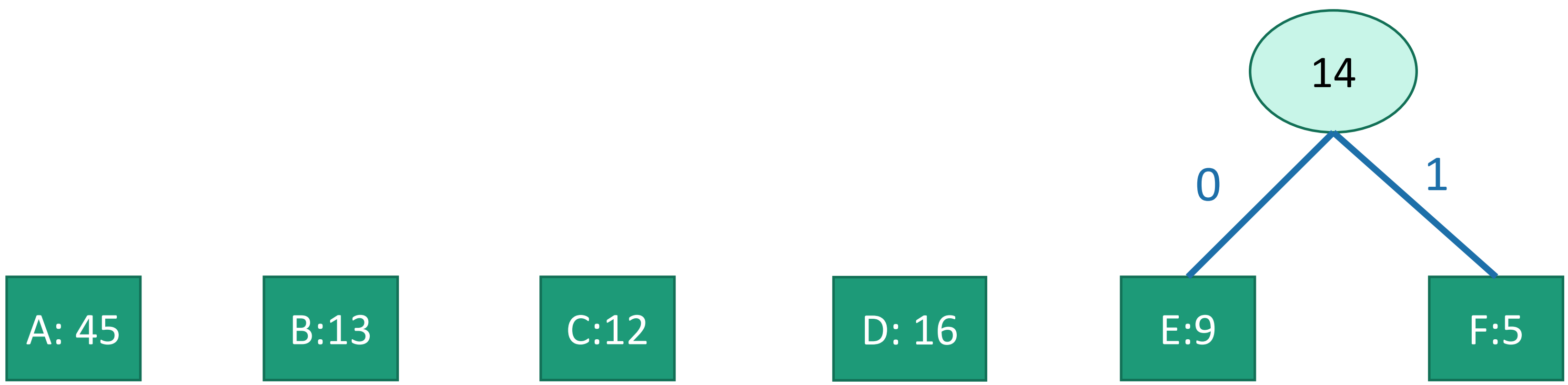
What exactly was the algorithm?

- 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]



Does it work?

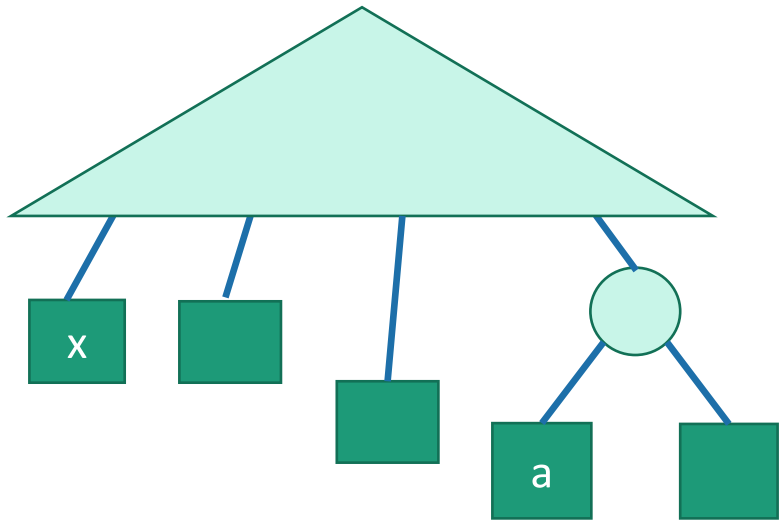
- Yes.
- Same strategy:
  - Show that at each step, the choices we are making **won't rule out** an optimal solution.
  - Lemma:
    - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.



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:



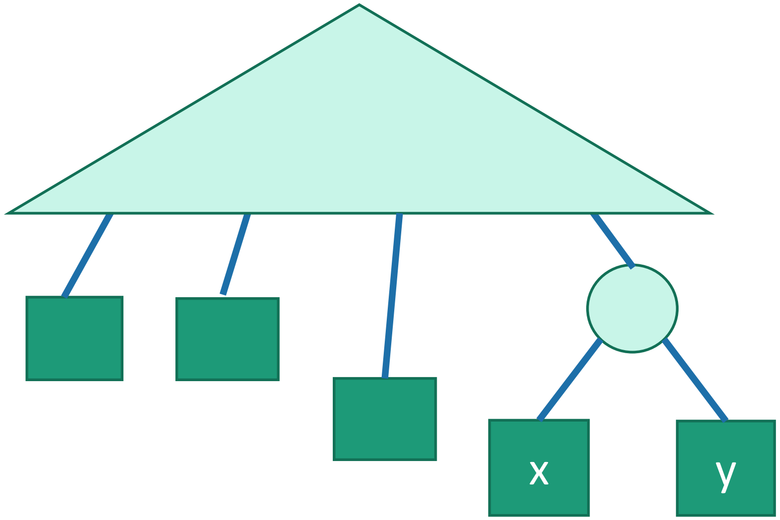
Lowest-level sibling nodes: at least one of them is neither x nor y

- 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 its encoding shorter.
- 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 its encoding shorter.
- 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.

Proof strategy
just like before

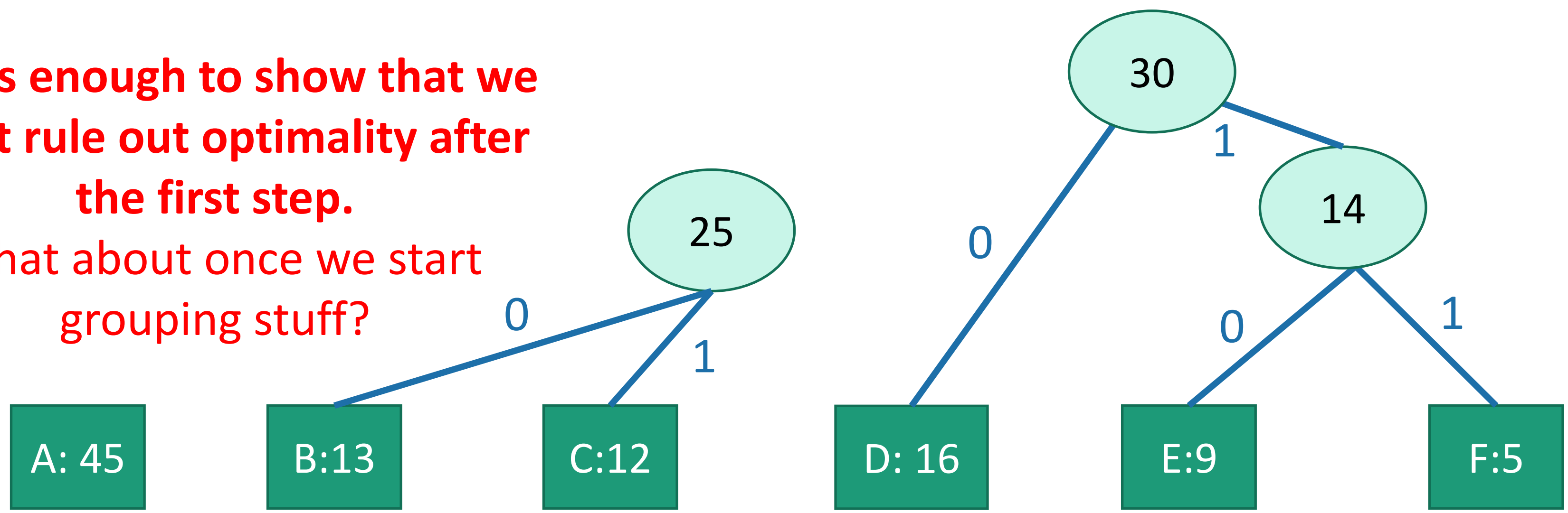
- Show that at each step, the choices we are making **won't rule out** an optimal solution.
- Lemma:
  - Suppose that x and y are the two least-frequent letters. Then there is an optimal tree where x and y are siblings.

Proof strategy
just like before

- 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.

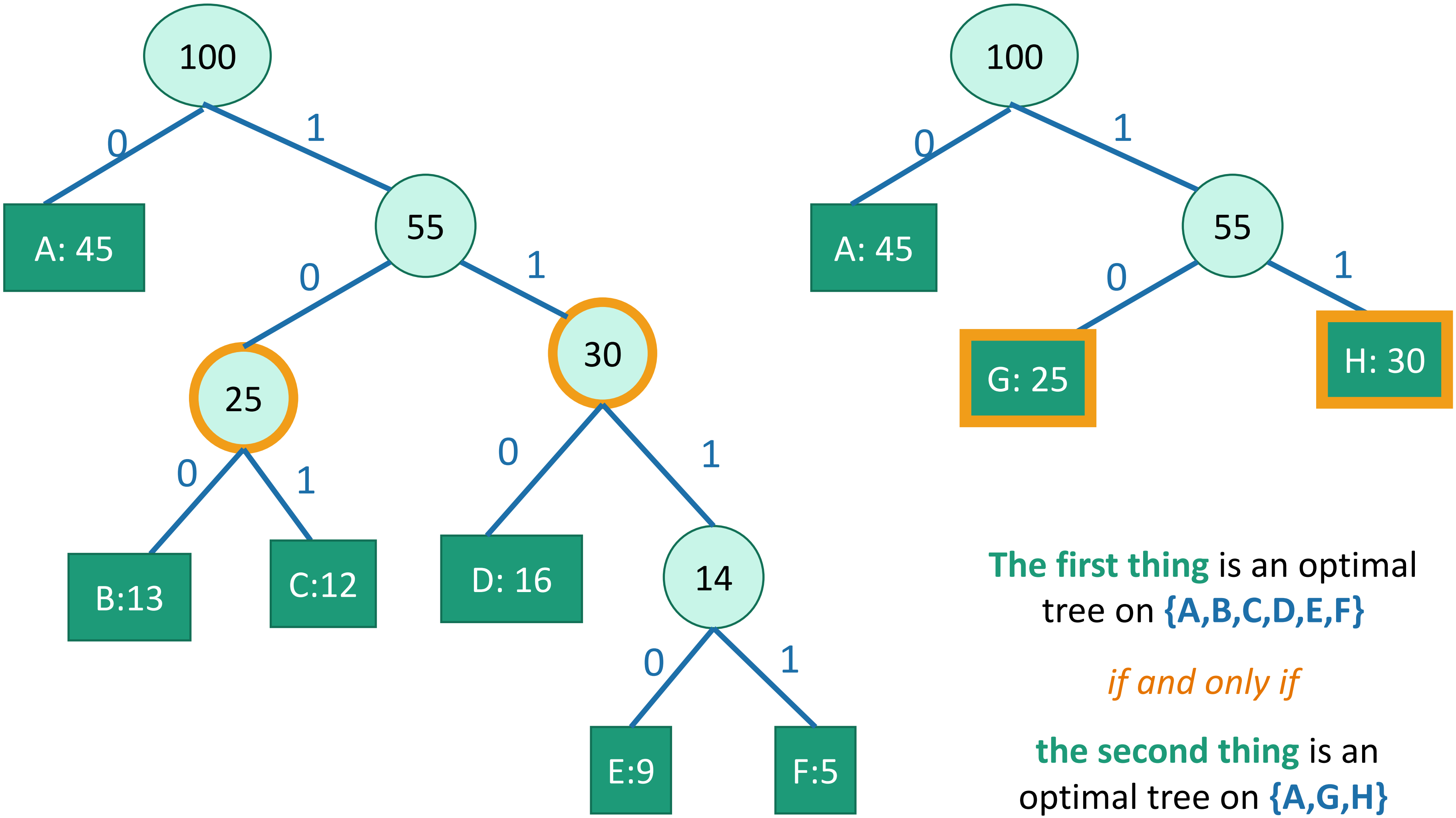
That's enough to show that we don't rule out optimality after the first step.

What about once we start grouping stuff?



Lemma 2

this distinction doesn't really matter



The first thing is an optimal tree on **{A,B,C,D,E,F}**

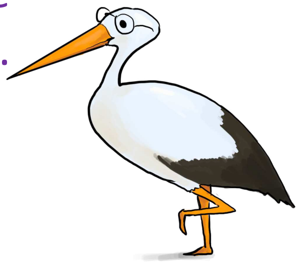
if and only if

the second thing is an optimal tree on **{A,G,H}**

Lemma 2
this distinction doesn't really matter

- For a proof:
  - See CLRS, Lemma 16.3
    - Rigorous although presented in a slightly different way
  - See Lecture Notes 14
    - A bit sketchier, but presented in the same way as here
  - Prove it yourself!
    - This is the best!

Getting all the details isn't that important, but you should convince yourself that this is true.



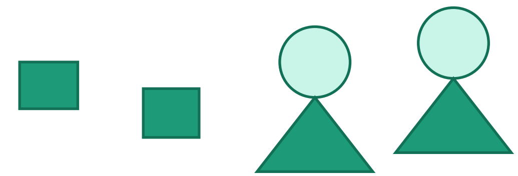
Siggi the Studious Stork

Together

- 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 2:
  - We may as well imagine that **CURRENT** contains only leaves.
- These imply:
  - At each step, our choice doesn’t rule out an optimal tree.

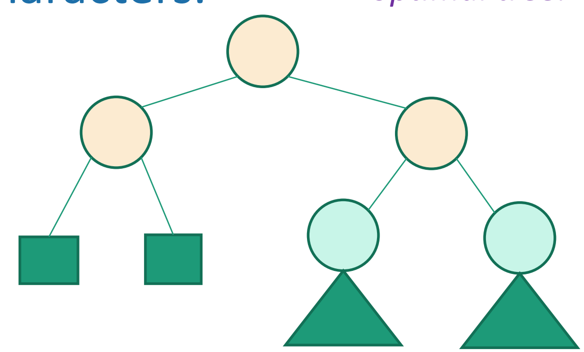
The whole argument

After the t'th step, we've got a bunch of current sub-trees:

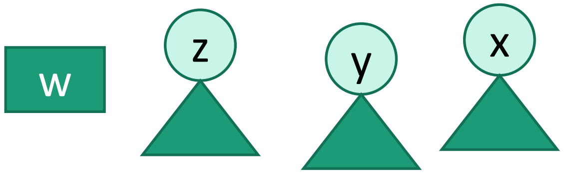


- Inductive hypothesis:
  - after the t'th step,
    - there is an optimal tree containing the current subtrees as "leaves"
- Base case:
  - after the 0'th step,
    - there is an optimal tree containing all the characters.
- Inductive step:
  - **TO DO**
- Conclusion:
  - after the last step,
    - there is an optimal tree containing this whole tree as a subtree.
  - **aka,**
    - after the last step the tree we've constructed is optimal.

Inductive hyp. asserts that our subtrees can be assembled into an optimal tree:



We’ve got a bunch of current sub-trees:



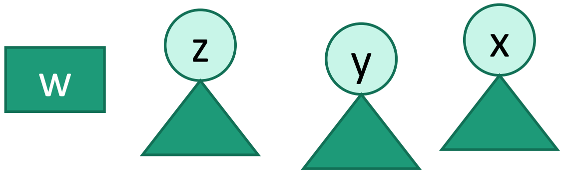
say that x and y are the two smallest.

Inductive step

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as “leaves.”
- Want to show:
  - After t steps, there is an optimal tree containing all the current sub-trees as leaves.

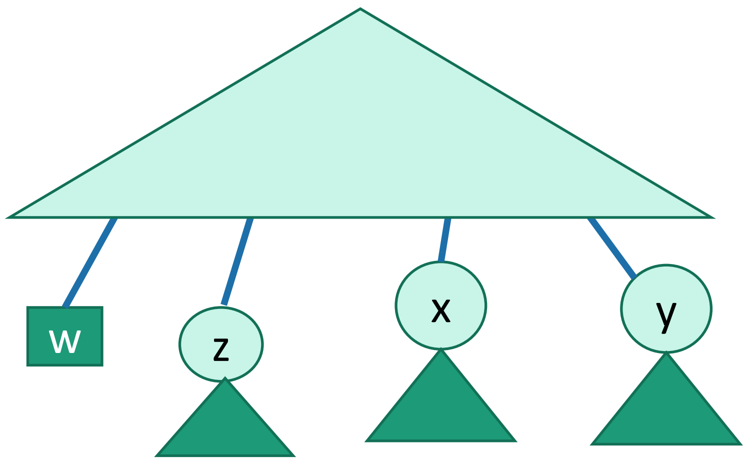
Inductive step

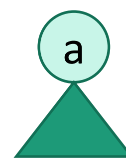
We’ve got a bunch of current sub-trees:



say that x and y are the two smallest.

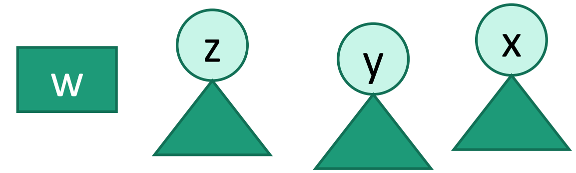
- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as “leaves.”



- By Lemma 2, may as well treat  as 

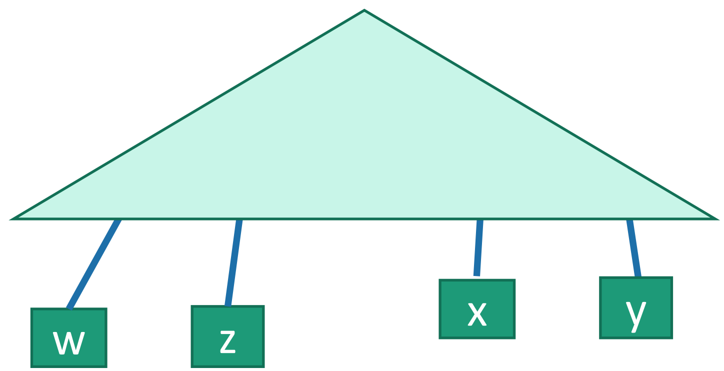
We've got a bunch of current sub-trees:

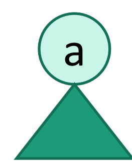
Inductive step



say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."

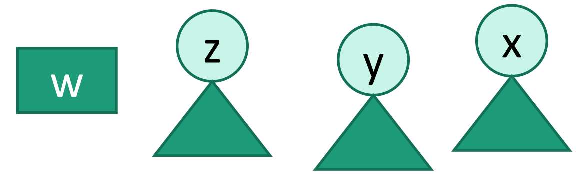


- By Lemma 2, may as well treat 

 as 

  - In particular, optimal trees on this new alphabet correspond to optimal trees on the original alphabet.

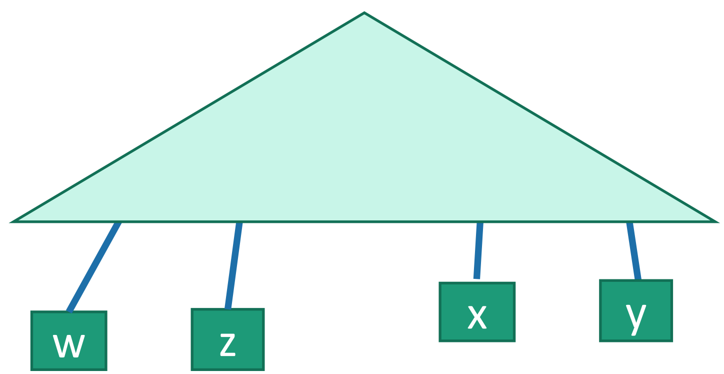
We’ve got a bunch of current sub-trees:



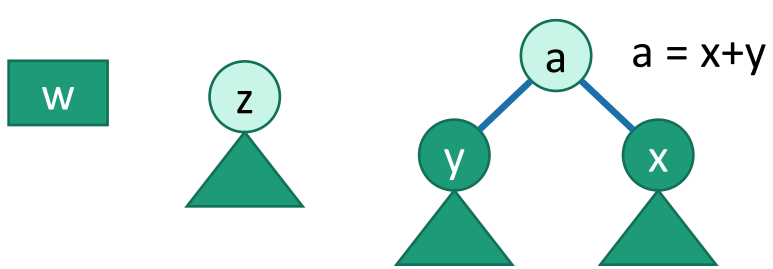
Inductive step

say that x and y are the two smallest.

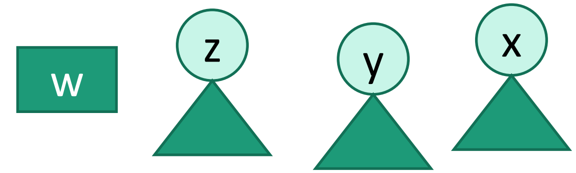
- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as “leaves.”



- Our algorithm would do this at level t:



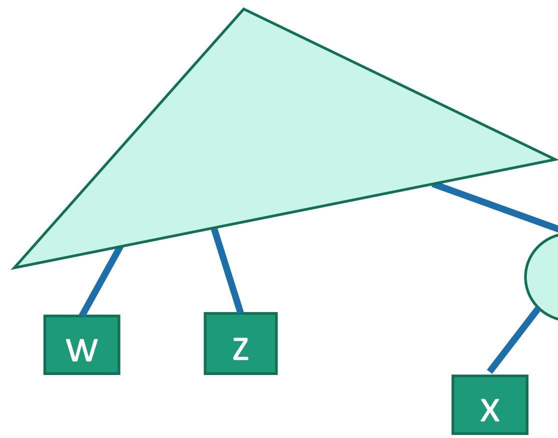
We’ve got a bunch of current sub-trees:



say that x and y are the two smallest.

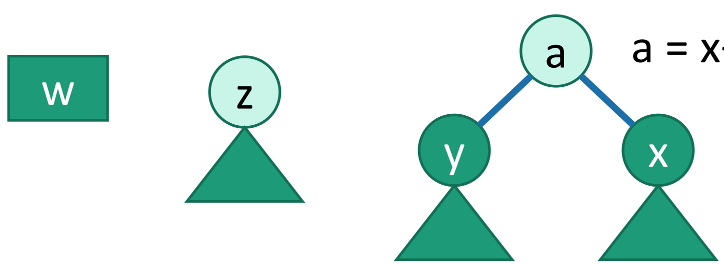
Inductive step

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as “leaves.”



Lemma 1 implies that there’s an optimal sub-tree that looks like this; aka, what our algorithm did okay.

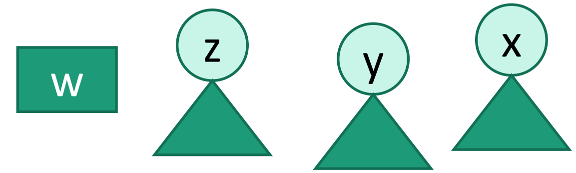
- Our algorithm would do this at level t:



a = x+y

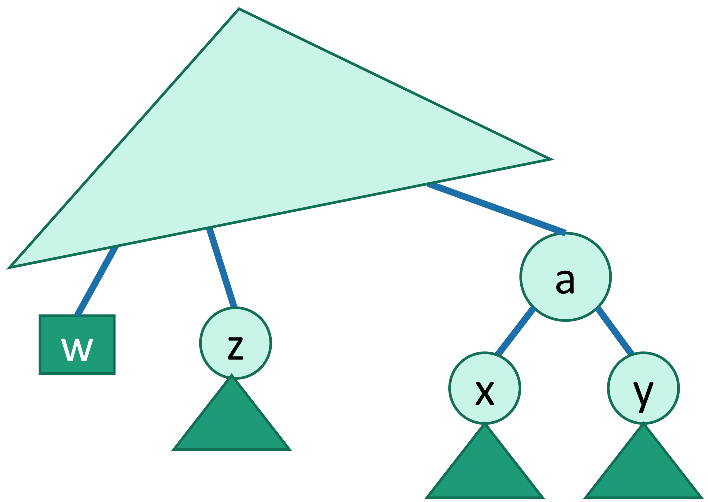
Inductive step

We’ve got a bunch of current sub-trees:



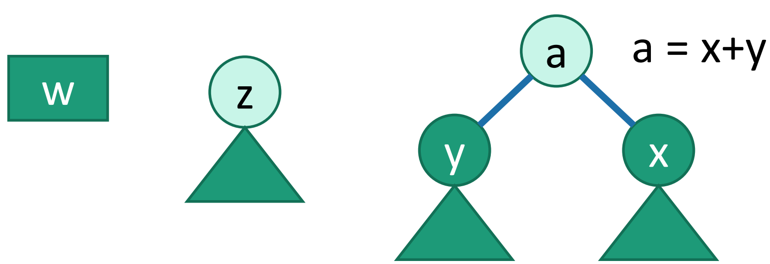
say that x and y are the two smallest.

- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as “leaves.”



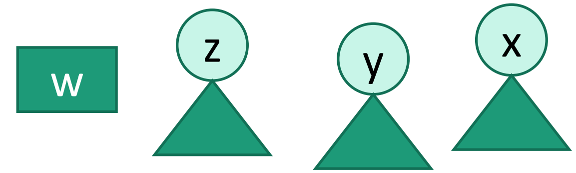
Lemma 2 again says that there’s an optimal tree that looks like this

- Our algorithm would do this at level t:



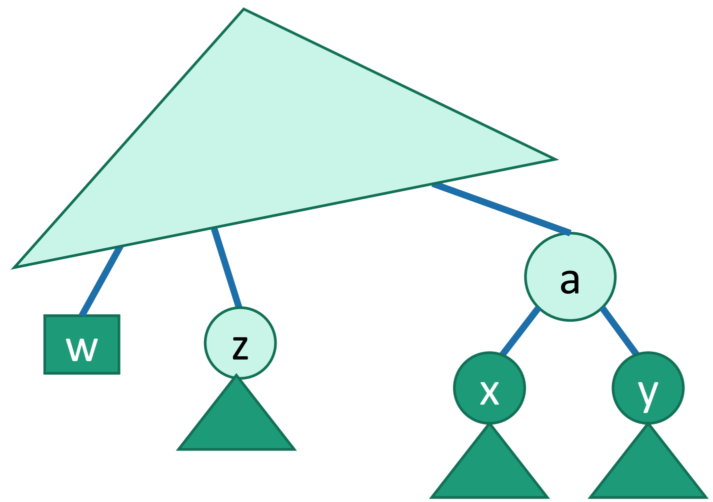
Inductive step

We've got a bunch of current sub-trees:



say that x and y are the two smallest.

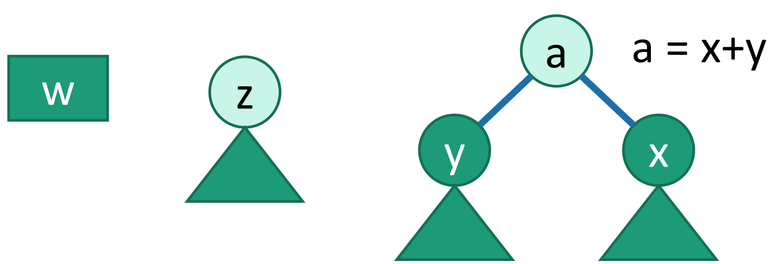
- Suppose that the inductive hypothesis holds for t-1
  - After t-1 steps, there is an optimal tree containing all the current sub-trees as "leaves."



Lemma 2 again says that there's an optimal tree that looks like this

aka, there is an optimal tree containing all the level-t sub-trees as "leaves".

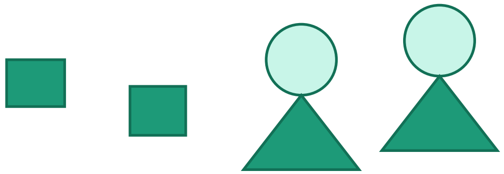
- Our algorithm would do this at level t:



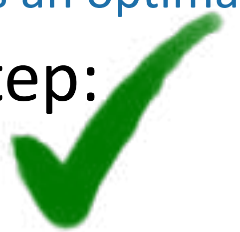
This is what we wanted to show for the inductive step.

Inductive outline:

After the t’th step, we’ve got a bunch of current sub-trees:

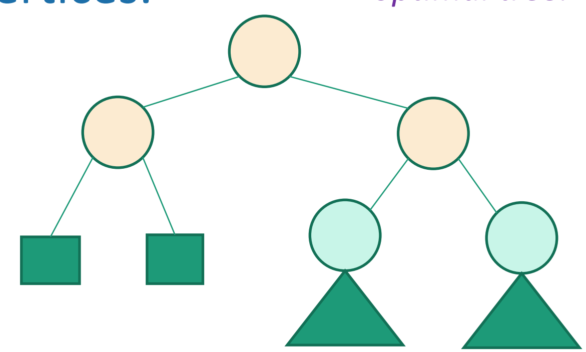


- Inductive hypothesis:
  - after the t’th step,
    - there is an optimal tree containing the current subtrees as “leaves”
- Base case:
  - after the 0’th step,
    - there is an optimal tree containing all the vertices.
- Inductive step:
  - **TO DO**



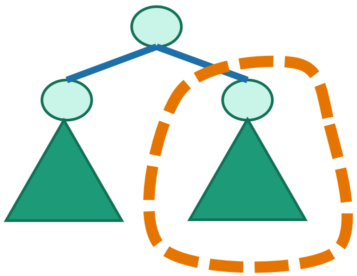
- Conclusion:
  - after the last step,
    - there is an optimal tree containing this whole tree as a subtree.
  - **aka,**
    - after the last step the tree we’ve constructed is optimal.

Inductive hyp. asserts that our subtrees can be assembled into an optimal tree:



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 "safe" 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

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: candidate greedy algorithms for MST