Module – Greedy Algorithms

Modified/Inspired from Stanford's CS161 by Nayyar Zaidi

Greedy algorithms

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

Today

- Four examples of greedy algorithms:
  - Activity Selection
  - Job Scheduling
  - Huffman Coding
  - Minimum Spanning Trees
    - Prim's Algorithm
    - Kruskal Algorithm

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 of the items.
  - What's the most valuable way to fill the knapsack?

🌮 🌮 💡 💡

Total weight: 10
Total value: 42

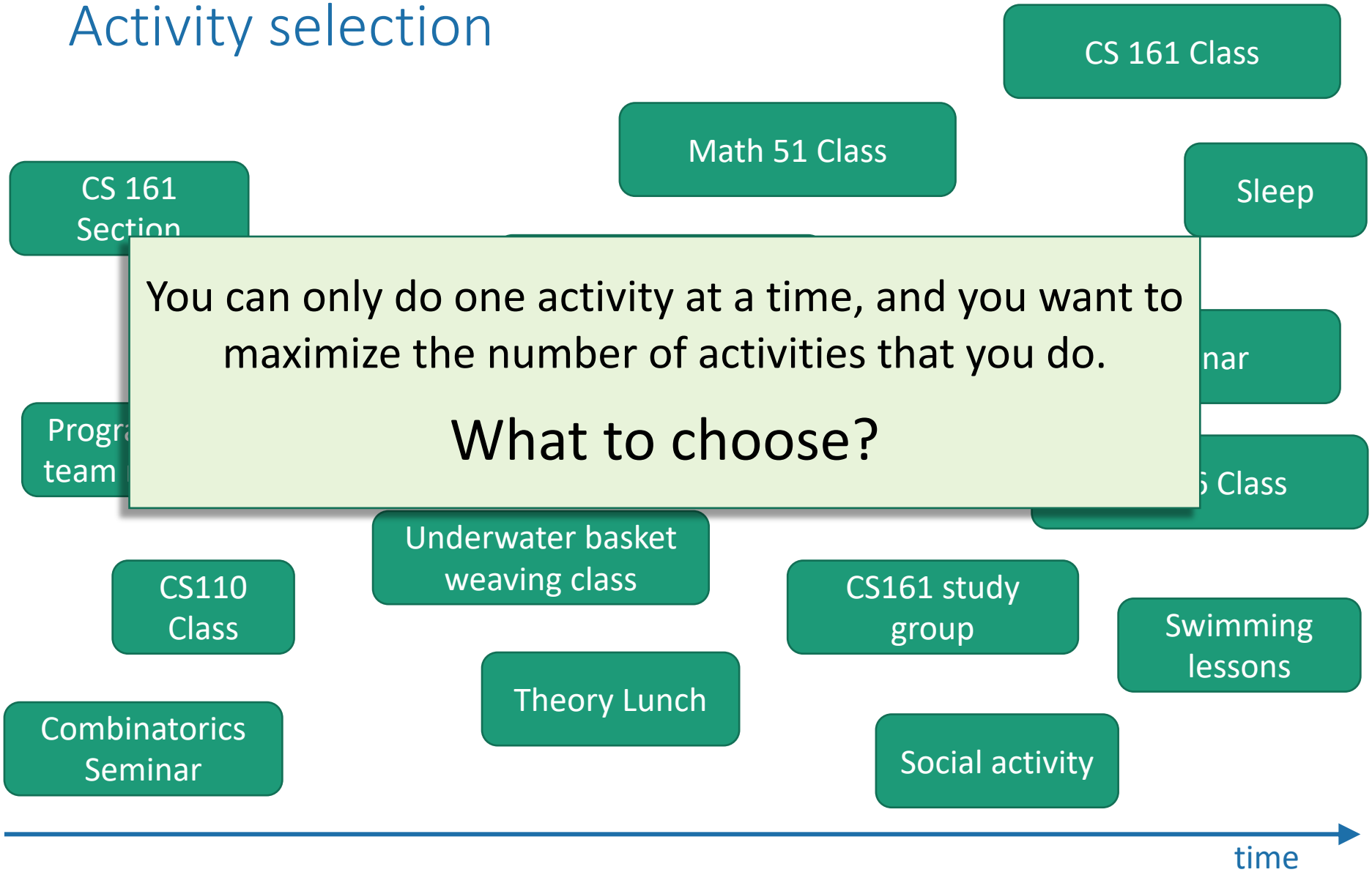
- **"Greedy"** algorithm for unbounded knapsack:
  - Tacos have the best Value/Weight ratio!
  - Keep grabbing tacos!

🌮 🌮 🌮

Total weight: 9
Total value: 39

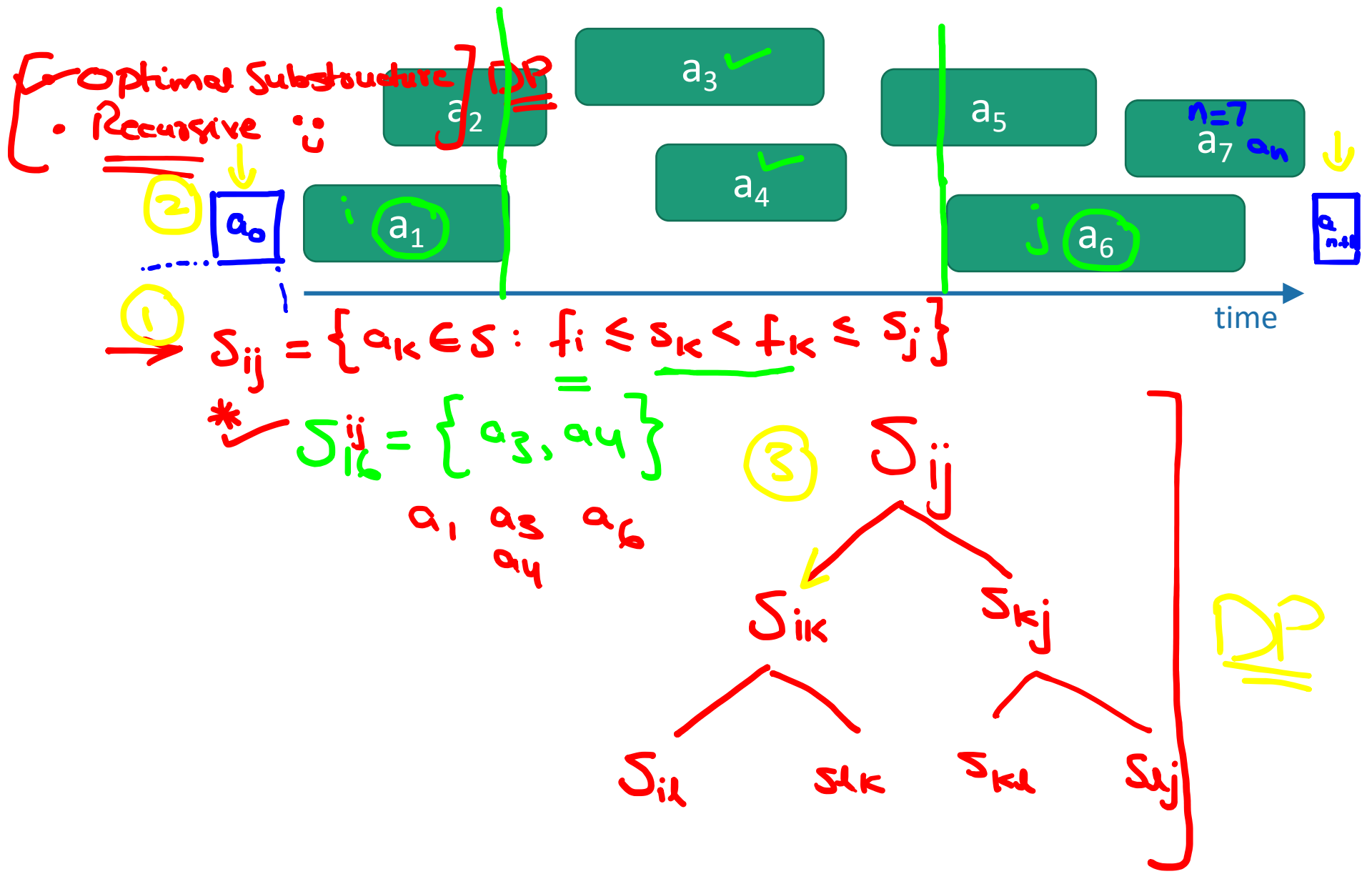
Example where greedy works

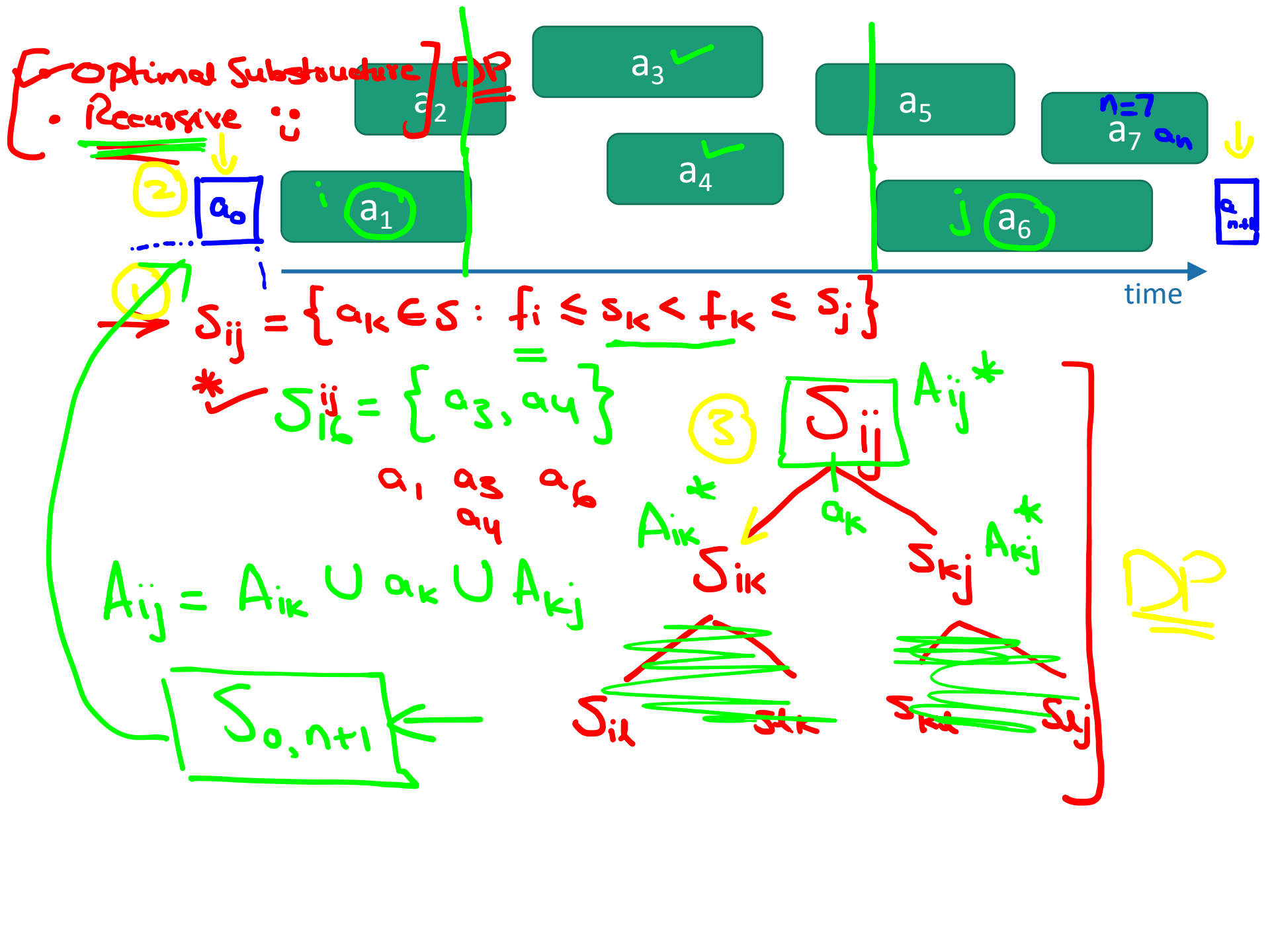
Activity selection

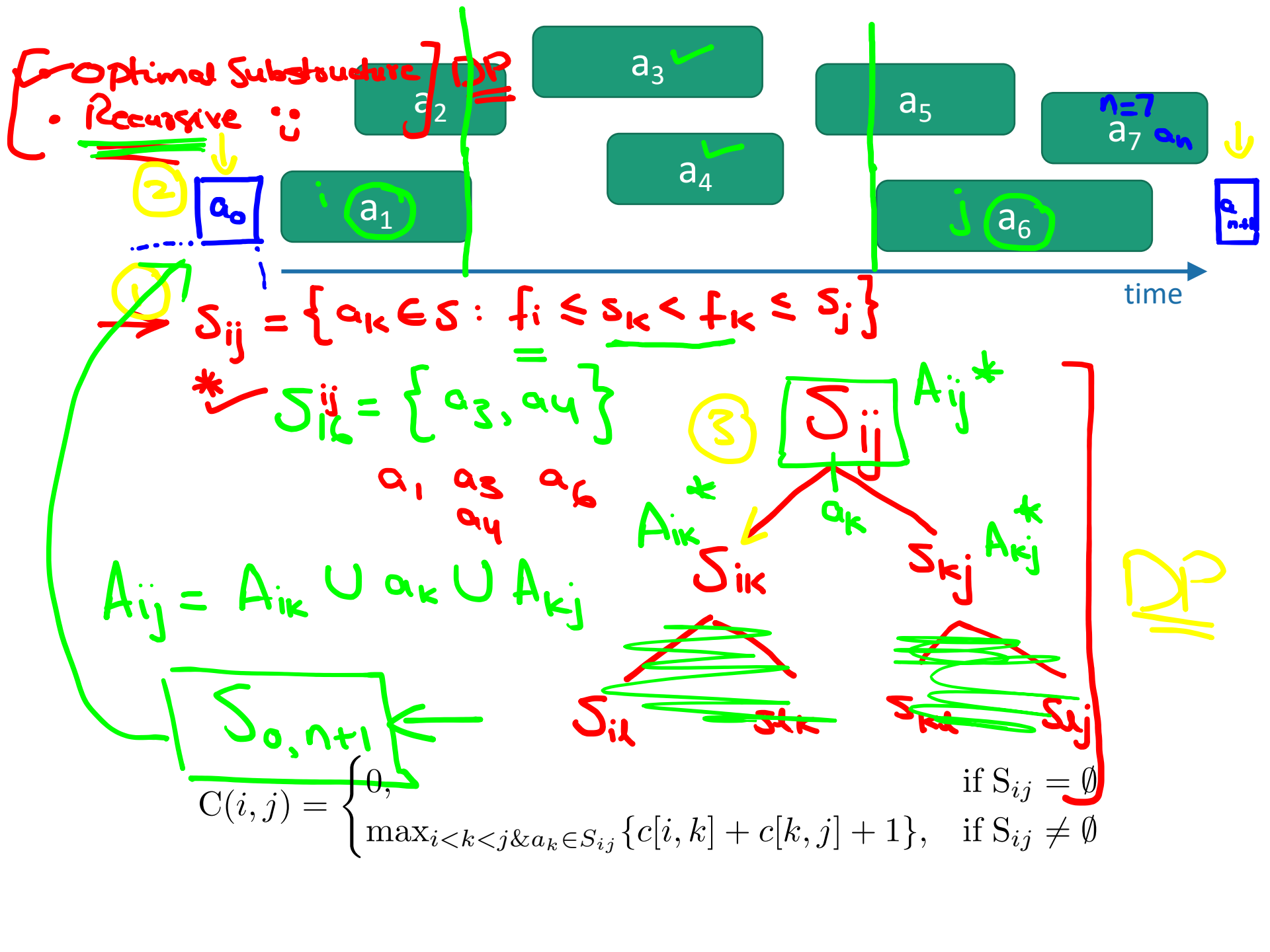


Activity selection

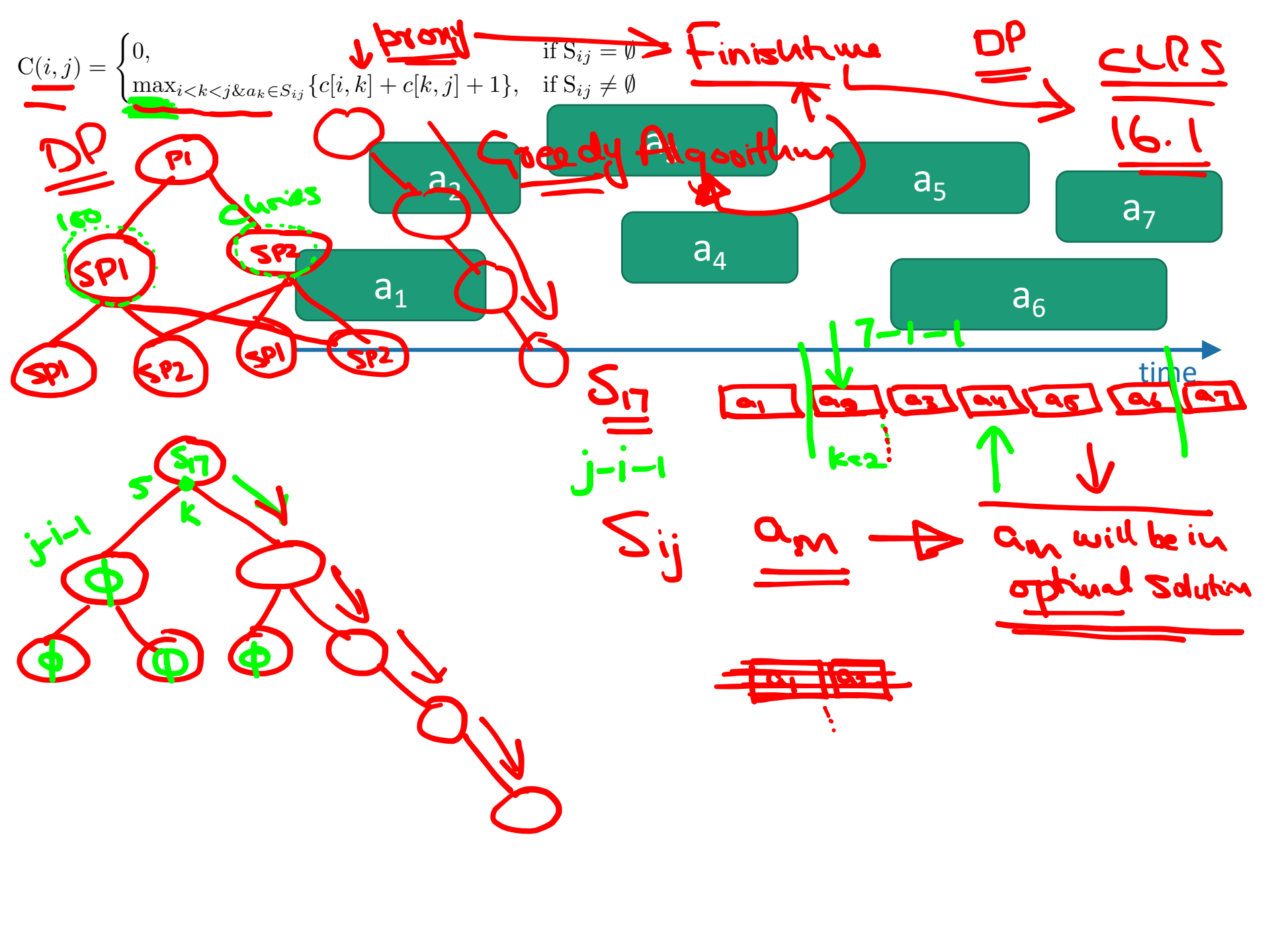
- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$
- Output:
  - How many activities can you do today?


Optimal Substructure
Recursive
DP
a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_{n+1}
n=7
i
j
time
$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$
$S_{ij} = \{a_3, a_4\}$
a_1 a_3 a_6
a_4
S_{ij}
S_{ik}
S_{kj}
S_{il}
S_{lk}
S_{kl}
S_{lj}
DP

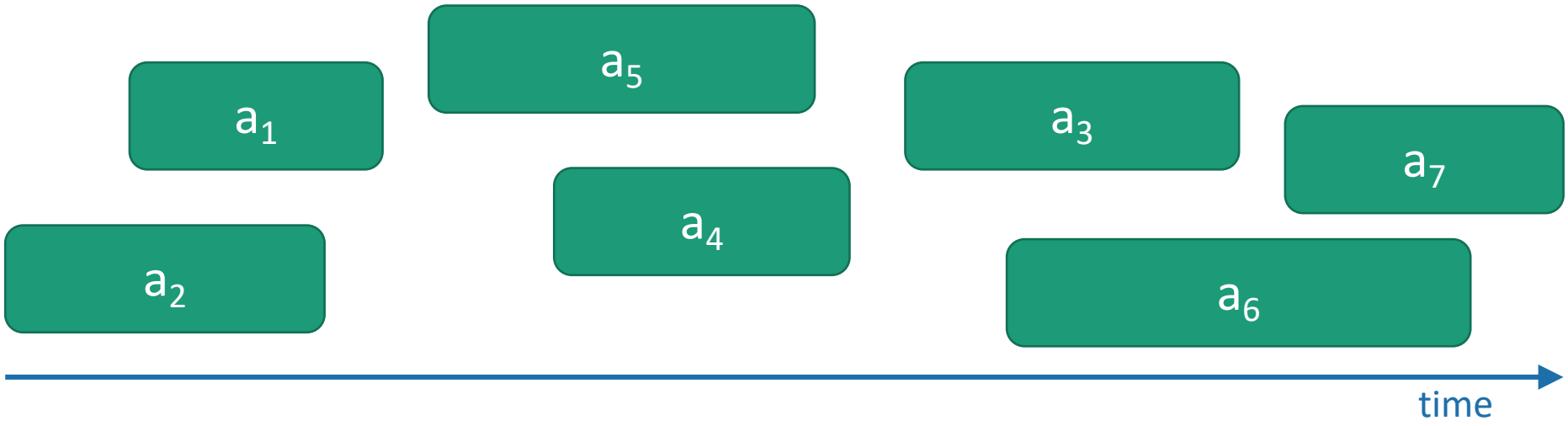

Optimal Substructure
DP
Recursive
a_0
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_{n+1}
n=7
i
j
time
$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$
$S_{ik} = \{a_3, a_4\}$
a_1 a_3 a_6 a_4
$A_{ij} = A_{ik} \cup a_k \cup A_{kj}$
$S_{0,n+1}$
S_{ij}
A_{ij}^*
A_{ik}^*
a_k
S_{ik}
S_{kj}
A_{kj}^*
DP


Optimal Substructure
DP
Recursive
a0
a1
a2
a3
a4
a5
a6
a7
an
n=7
i
j
time
Sij = {ak ∈ S : fi ≤ sk < fk ≤ sj}
Sij = {a3, a4}
a1 a3 a6
a4
Aij = Aik ∪ ak ∪ Akj
Aij*
Aik*
Akj*
Sii
Sik
Skj
So,n+1
DP
C(i, j) = 0, if Sij = ∅
max_{i<k<j & ak∈Sij} {c[i, k] + c[k, j] + 1}, if Sij ≠ ∅

$$C(i,j) = \begin{cases} 0, & \text{if } S_{ij} = \emptyset \\ \max_{i<k<j \& a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & \text{if } S_{ij} \neq \emptyset \end{cases}$$

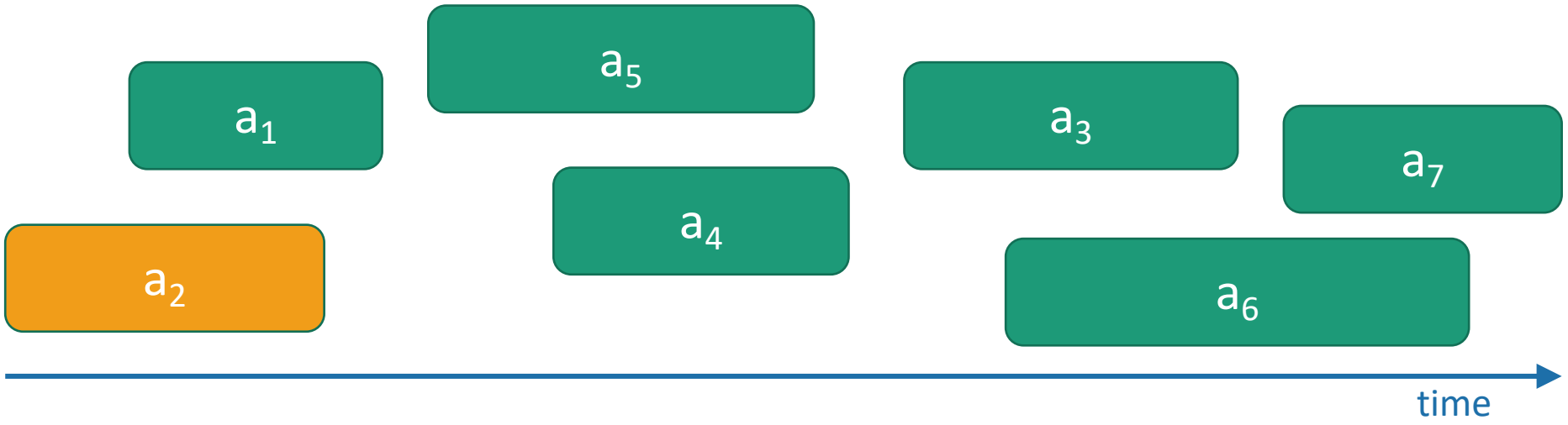


Greedy Algorithm



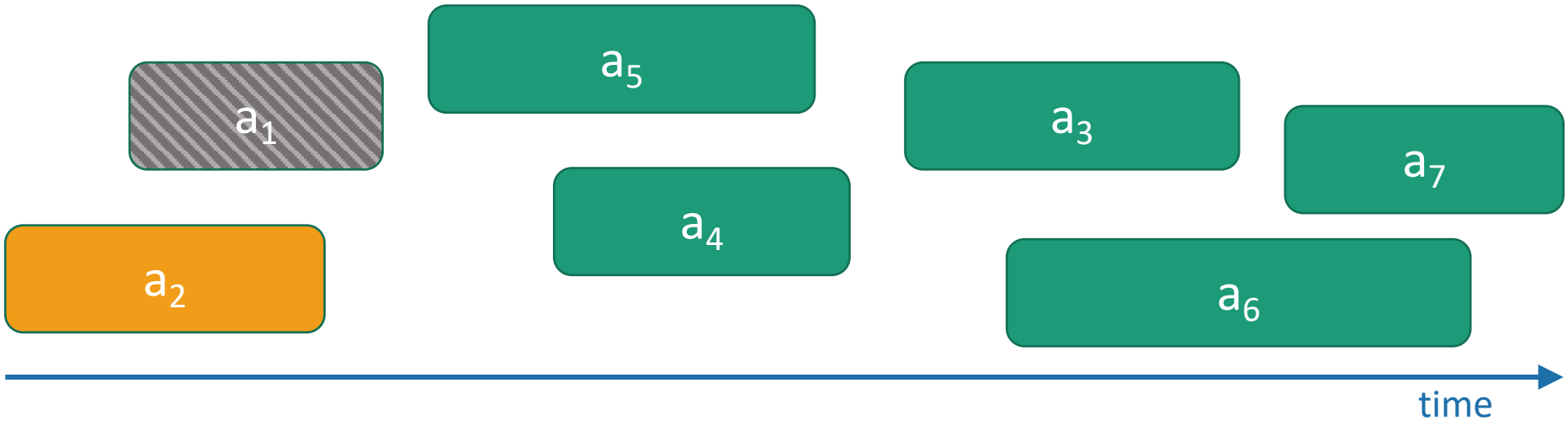
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



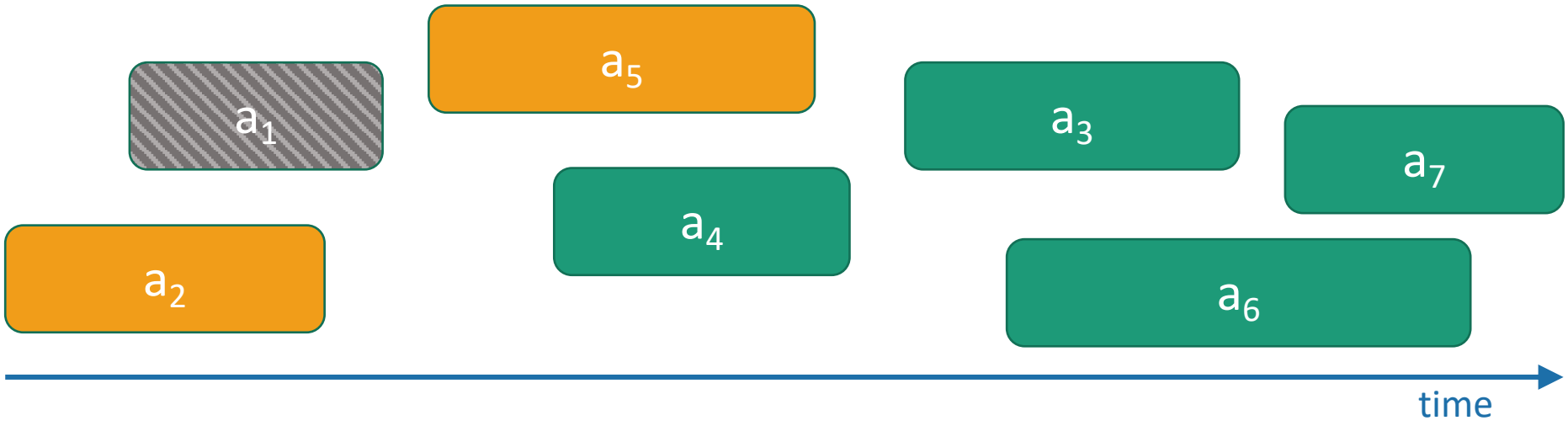
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



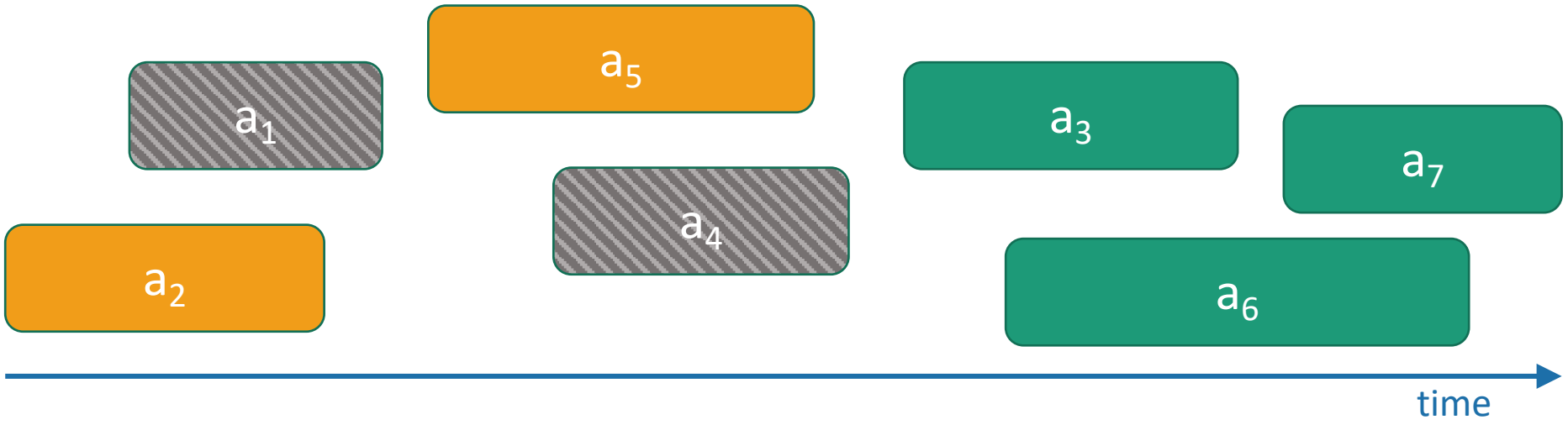
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



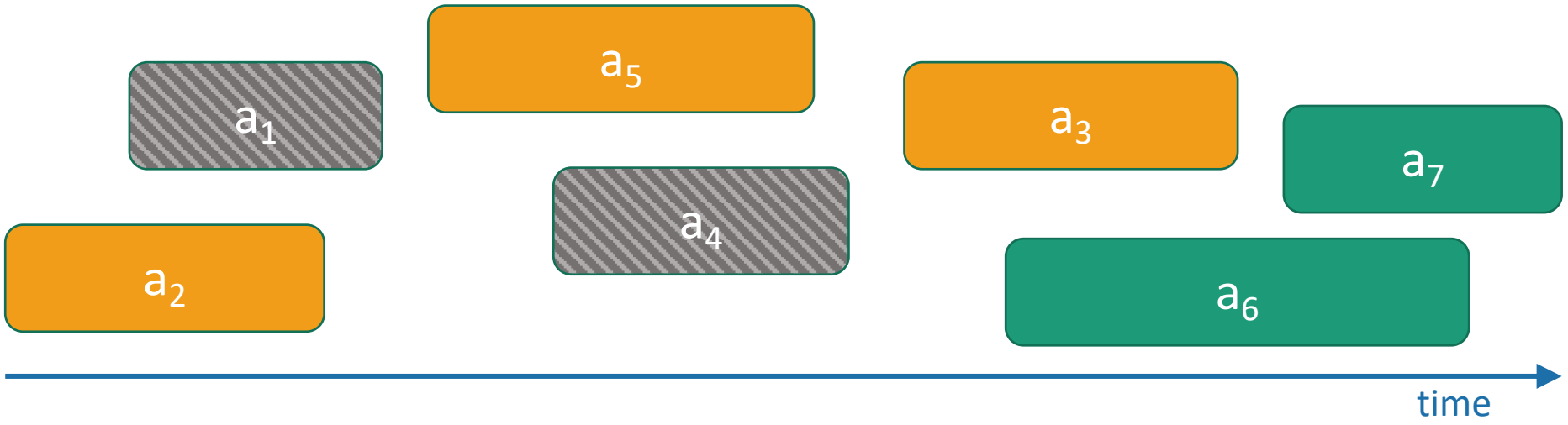
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



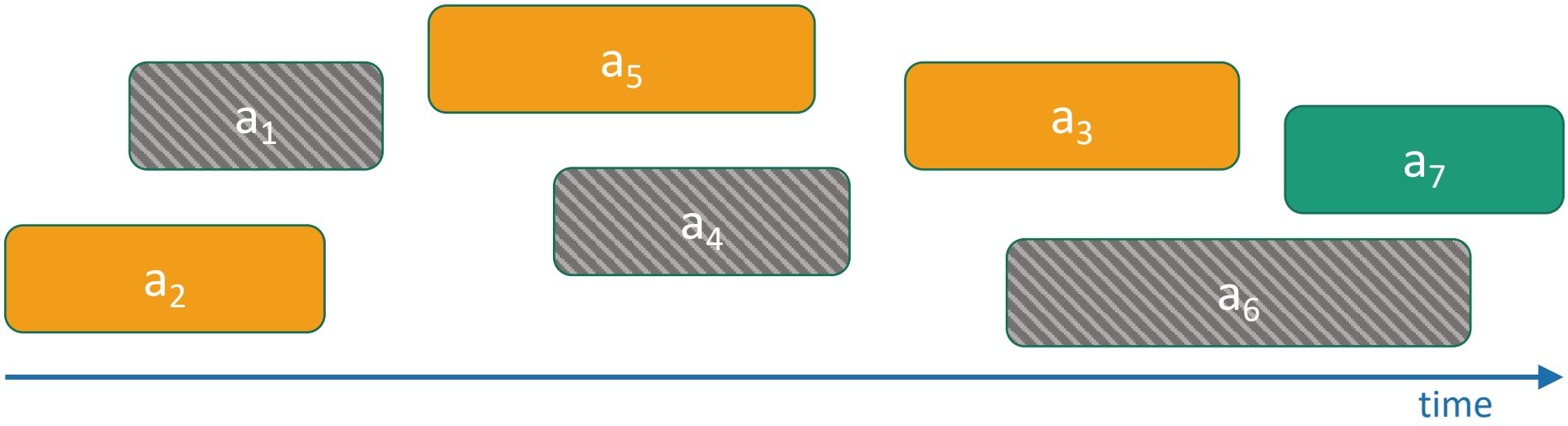
- Pick activity you can add with the smallest finish time.
- Repeat.

Greedy Algorithm



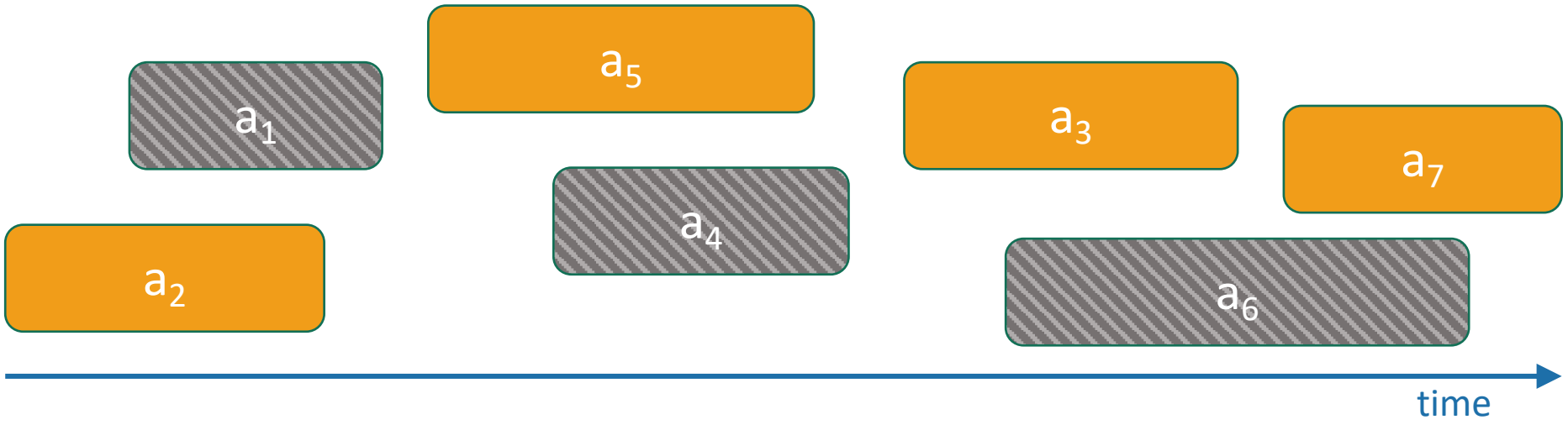
- Pick activity you can add with the smallest finish time.
- Repeat.

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

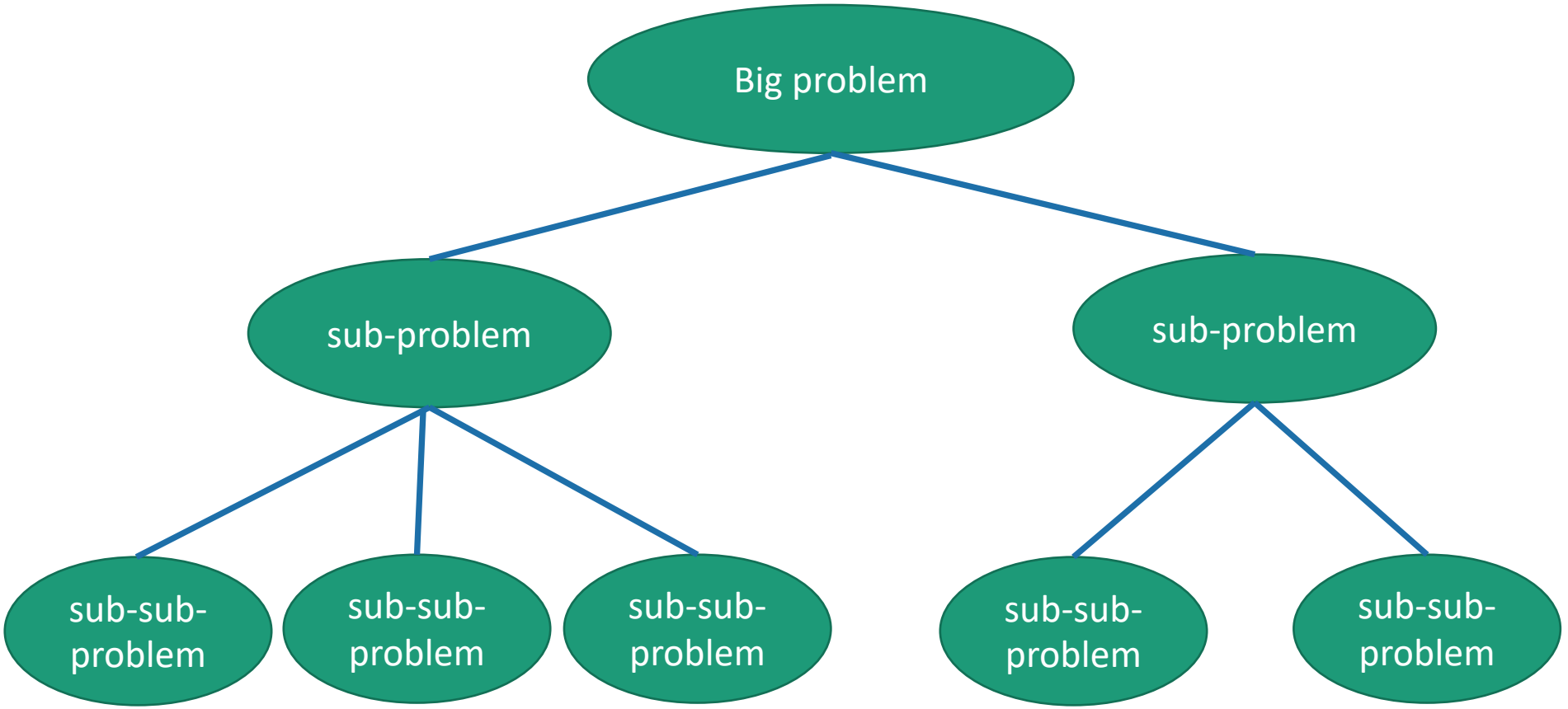
- Running time:
  - O(n) if the activities are already sorted by finish time.
  - Otherwise O(nlog(n)) if you have to sort them first.

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.

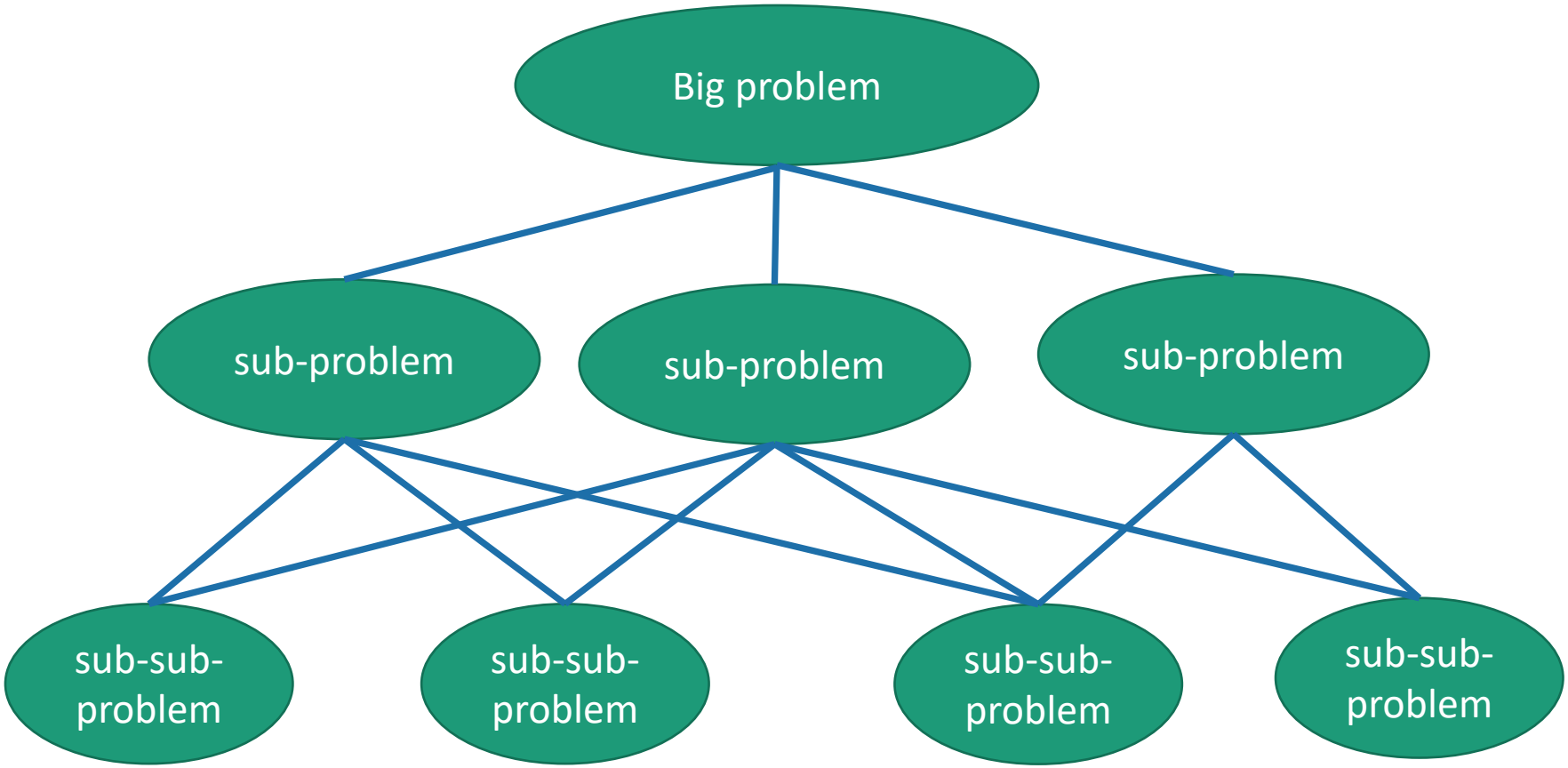
Sub-problem graph view

- Divide-and-conquer:



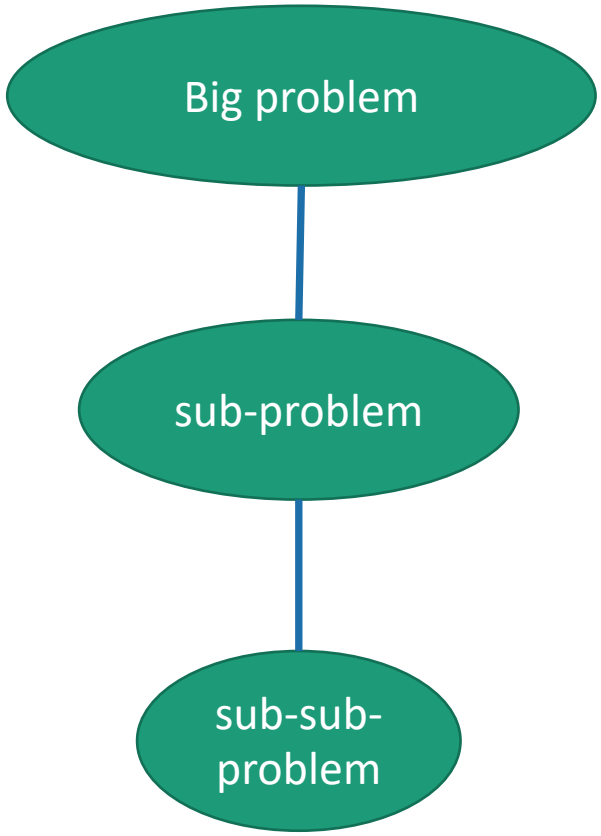
Sub-problem graph view

- Dynamic Programming:



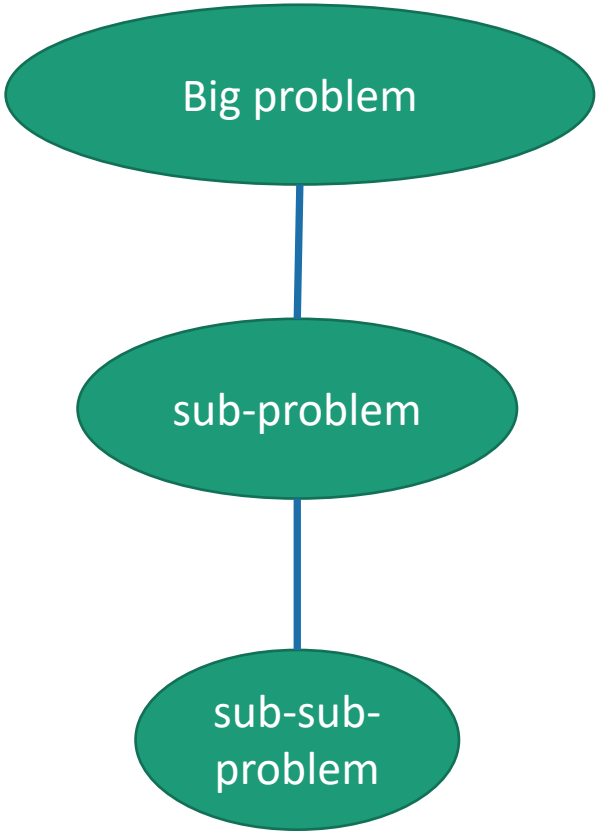
Sub-problem graph view

- 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**.

Let’s see a few more examples

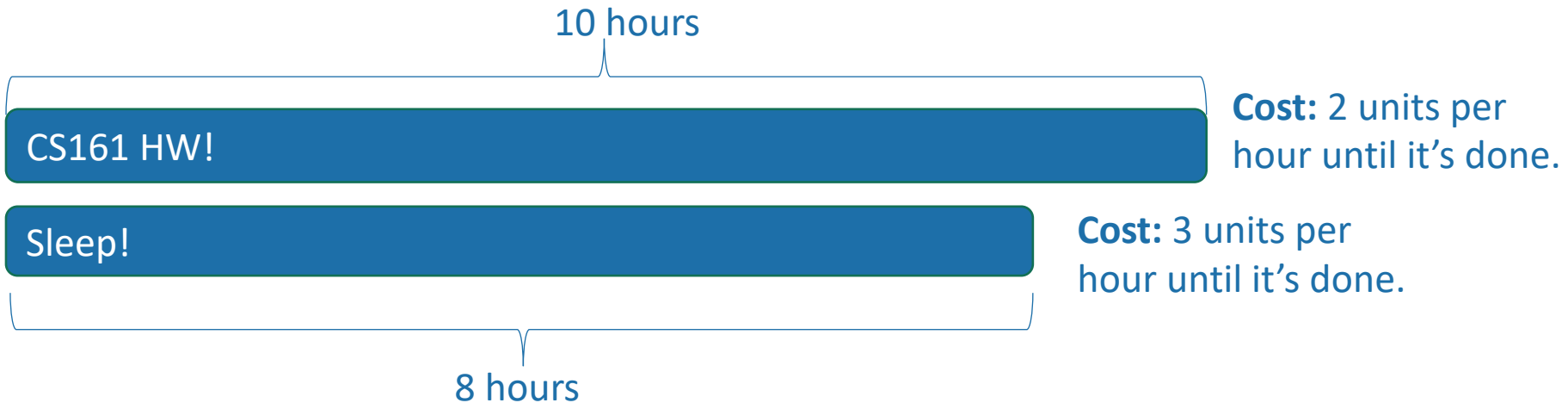
Another example:
Scheduling

- 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

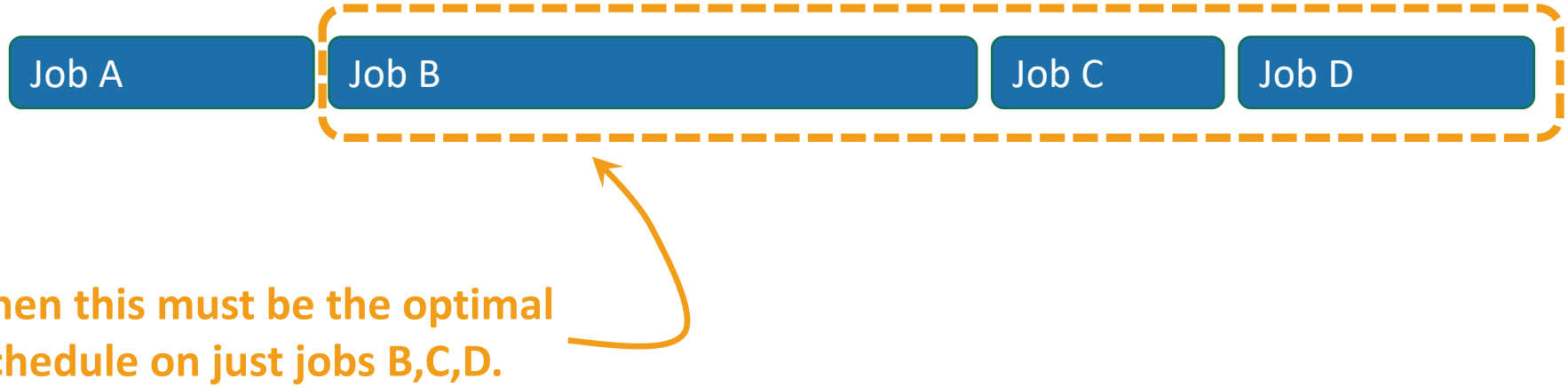


- CS161 HW, then Sleep: costs **$10 \cdot 2 + (10 + 8) \cdot 3 = 74$ units**
- Sleep, then CS161 HW: costs **$8 \cdot 3 + (10 + 8) \cdot 2 = 60$ units**

Optimal substructure

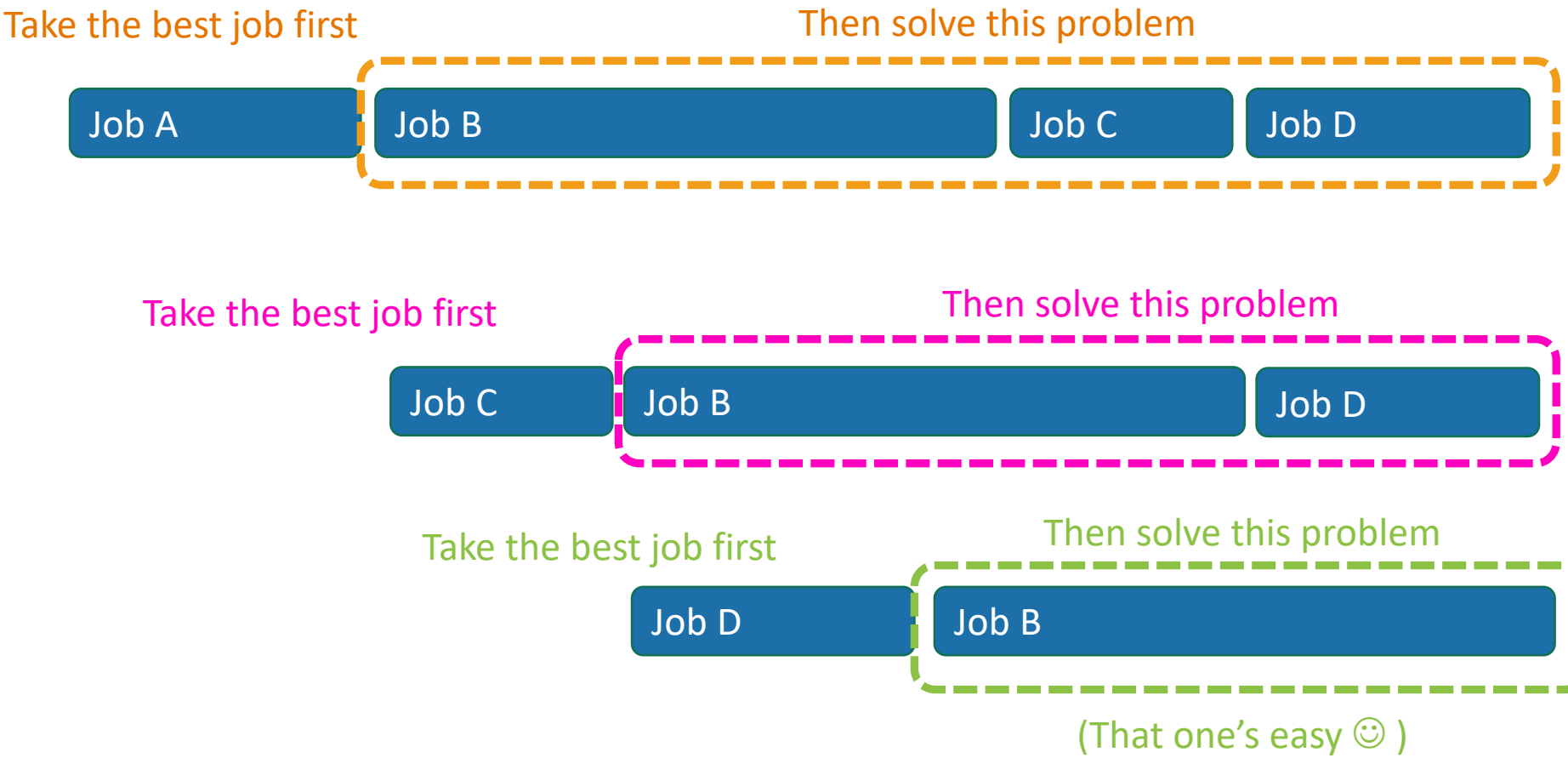
- This problem breaks up nicely into sub-problems:

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.**

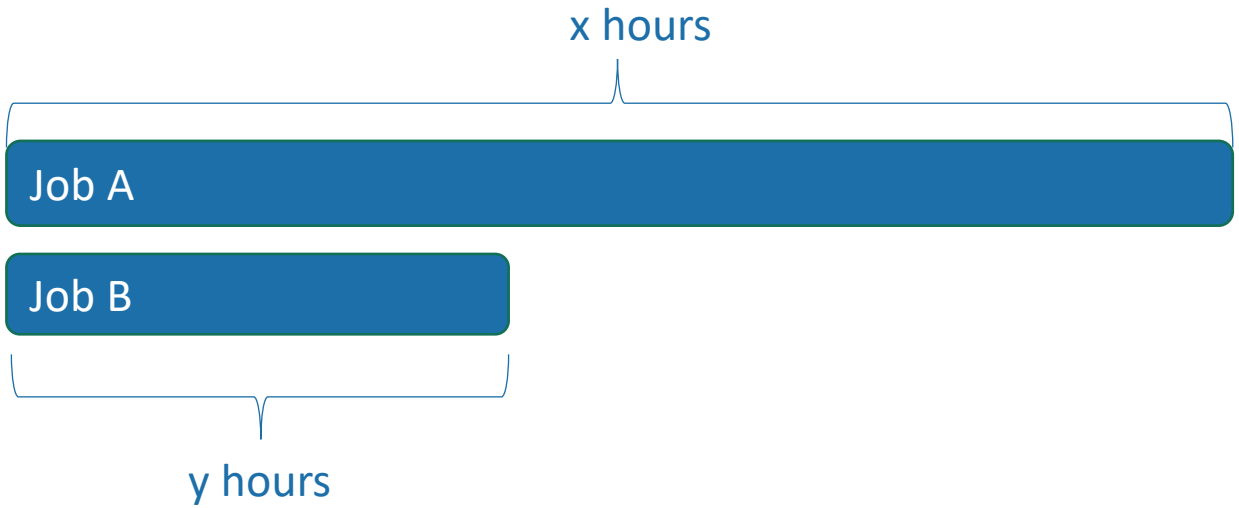
The optimal solution to this problem extends an optimal solution to the whole thing.

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:

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

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

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.

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

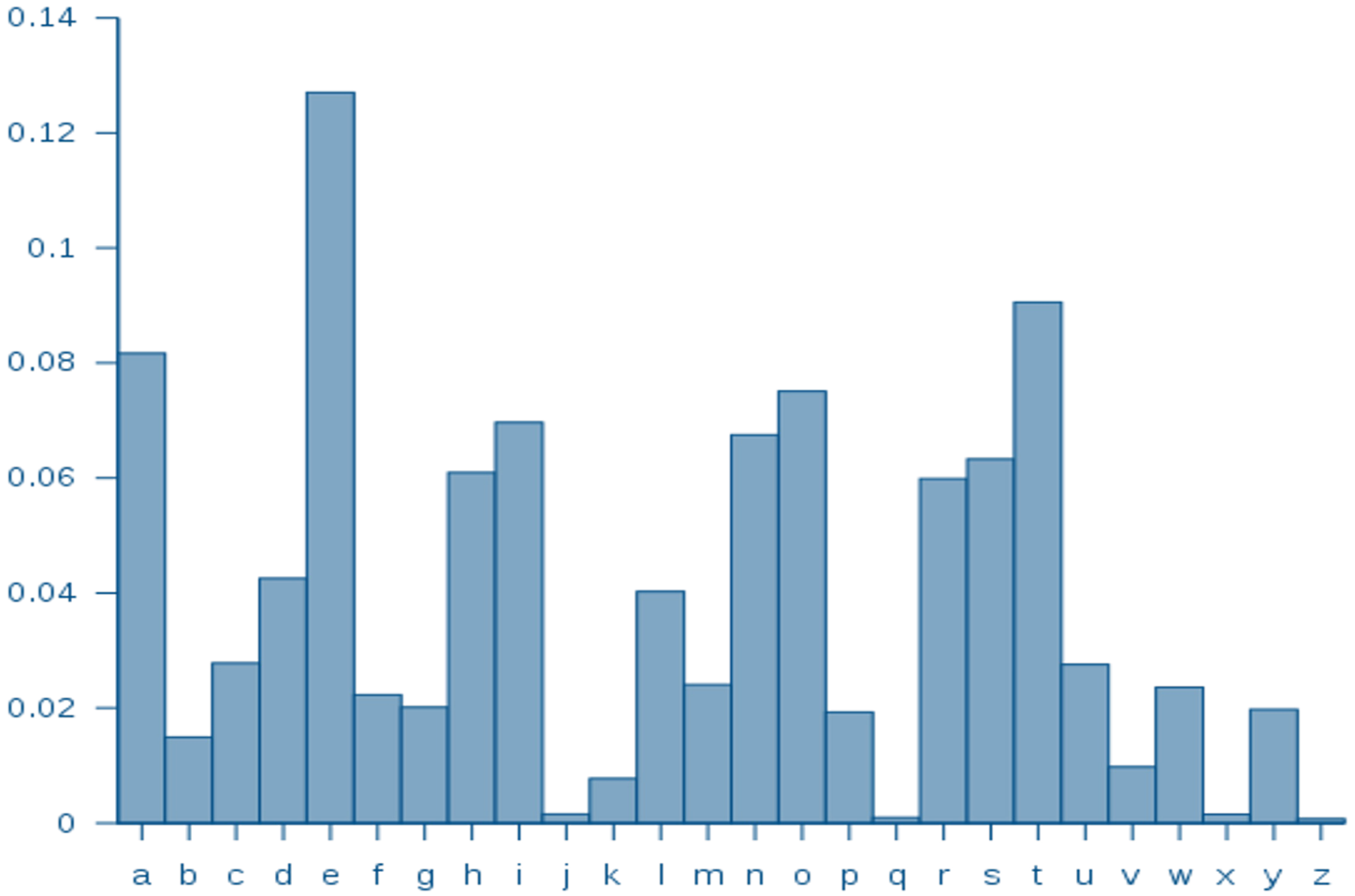
Huffman coding

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**

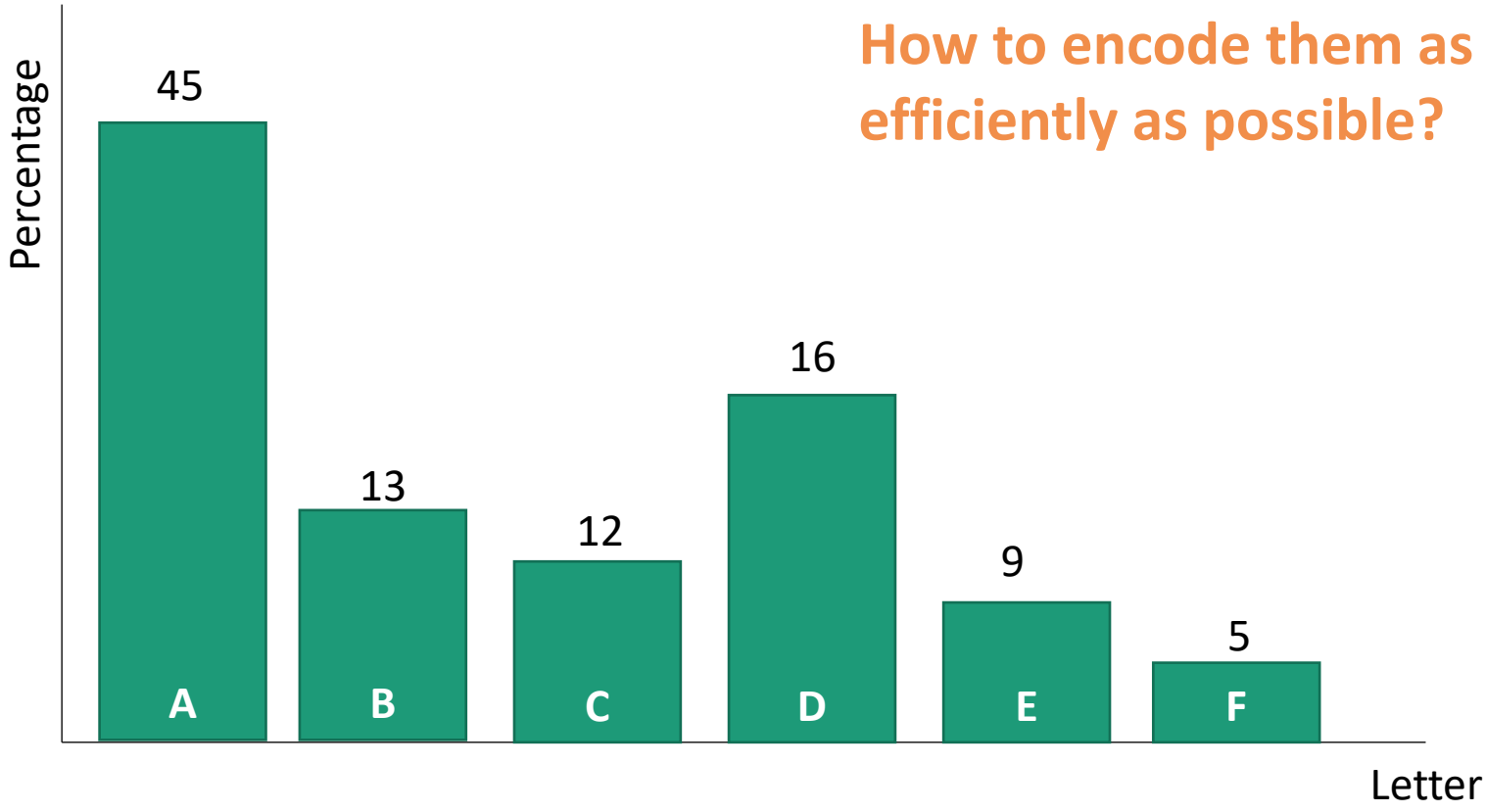
- `qwertyui_opasdfg+hjklzxcv`
- 01110001 01110111 01100101 01110010 01110100 01111001 01110101 01101001 01011111 01101111 01110000 01100001 01110011 01100100 01100110 01100111 00101011 01101000 01101010 01101011 01101100 01111010 01111000 01100011 01110110

Suppose we have some distribution on characters



Suppose we have some distribution on characters

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



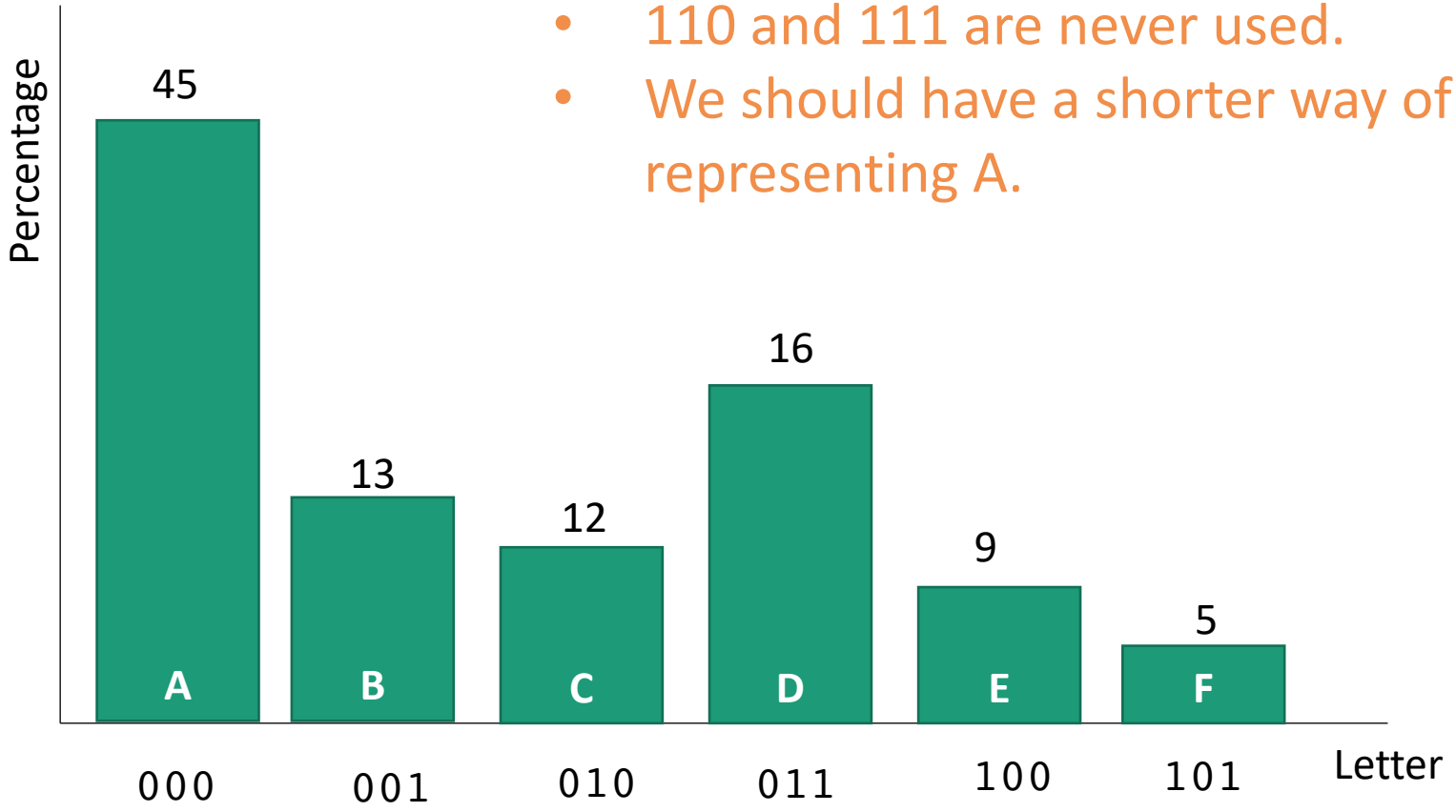
How to encode them as efficiently as possible?

Try 0
(like ASCII)

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

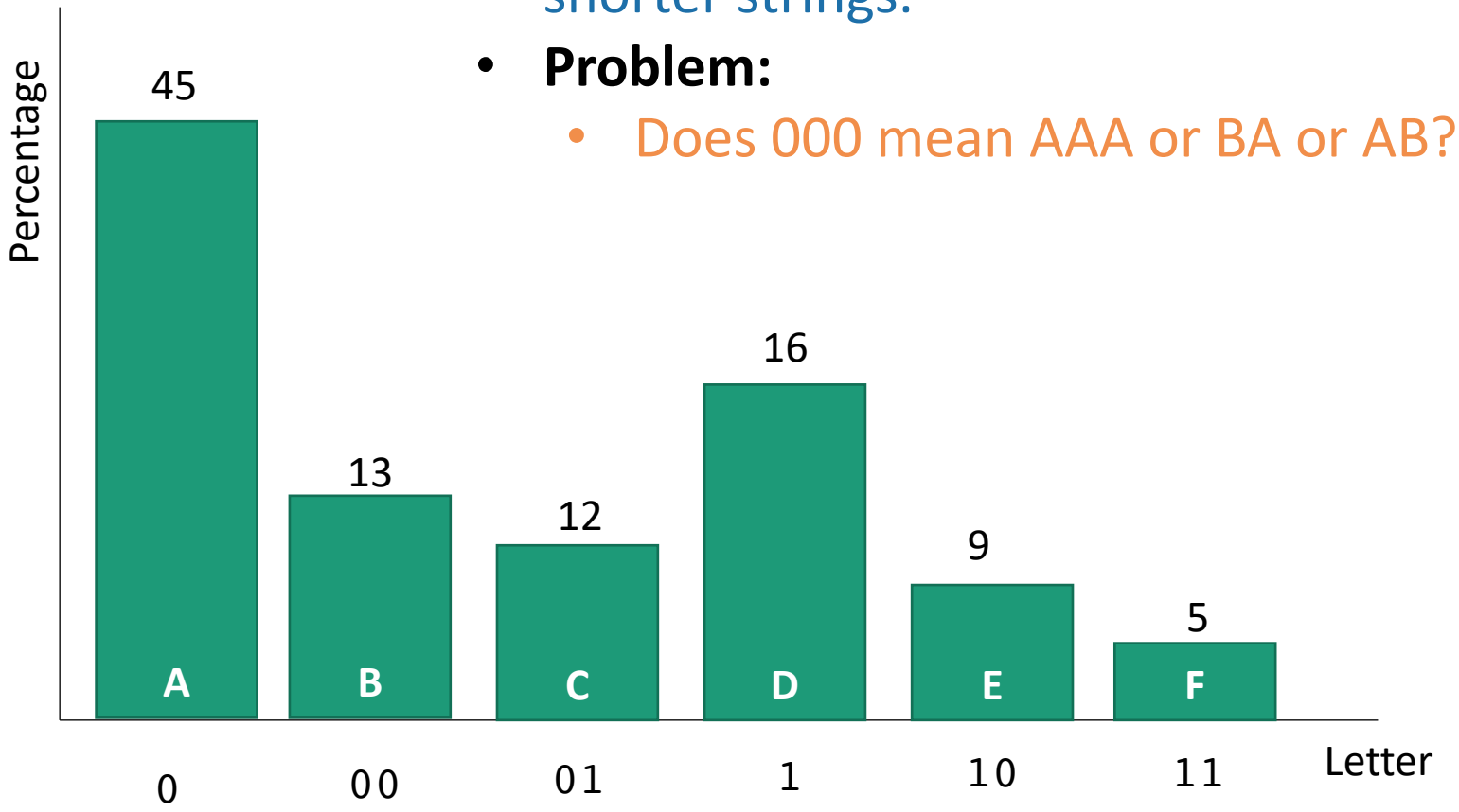
Wasteful!

- 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?

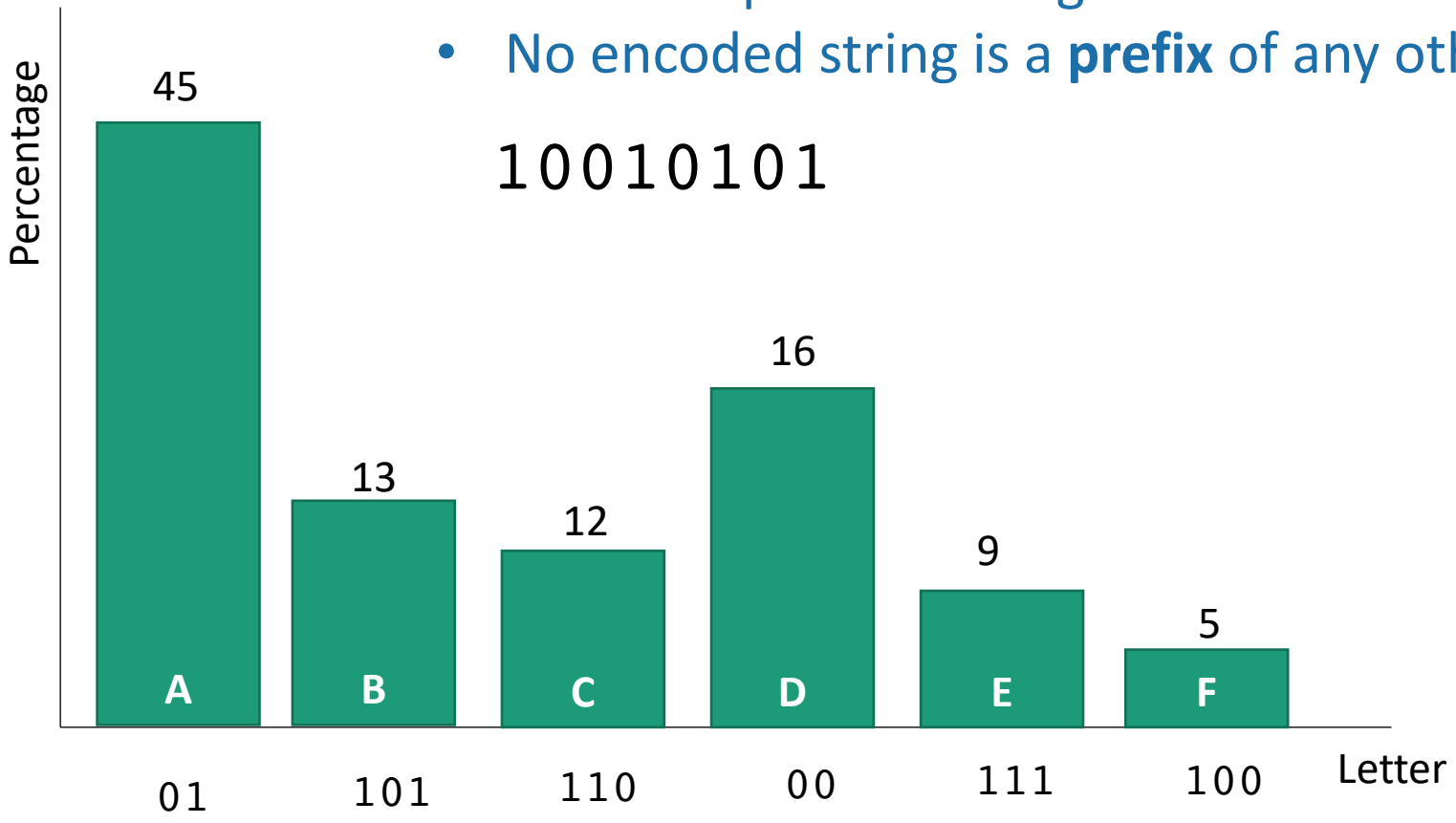


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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

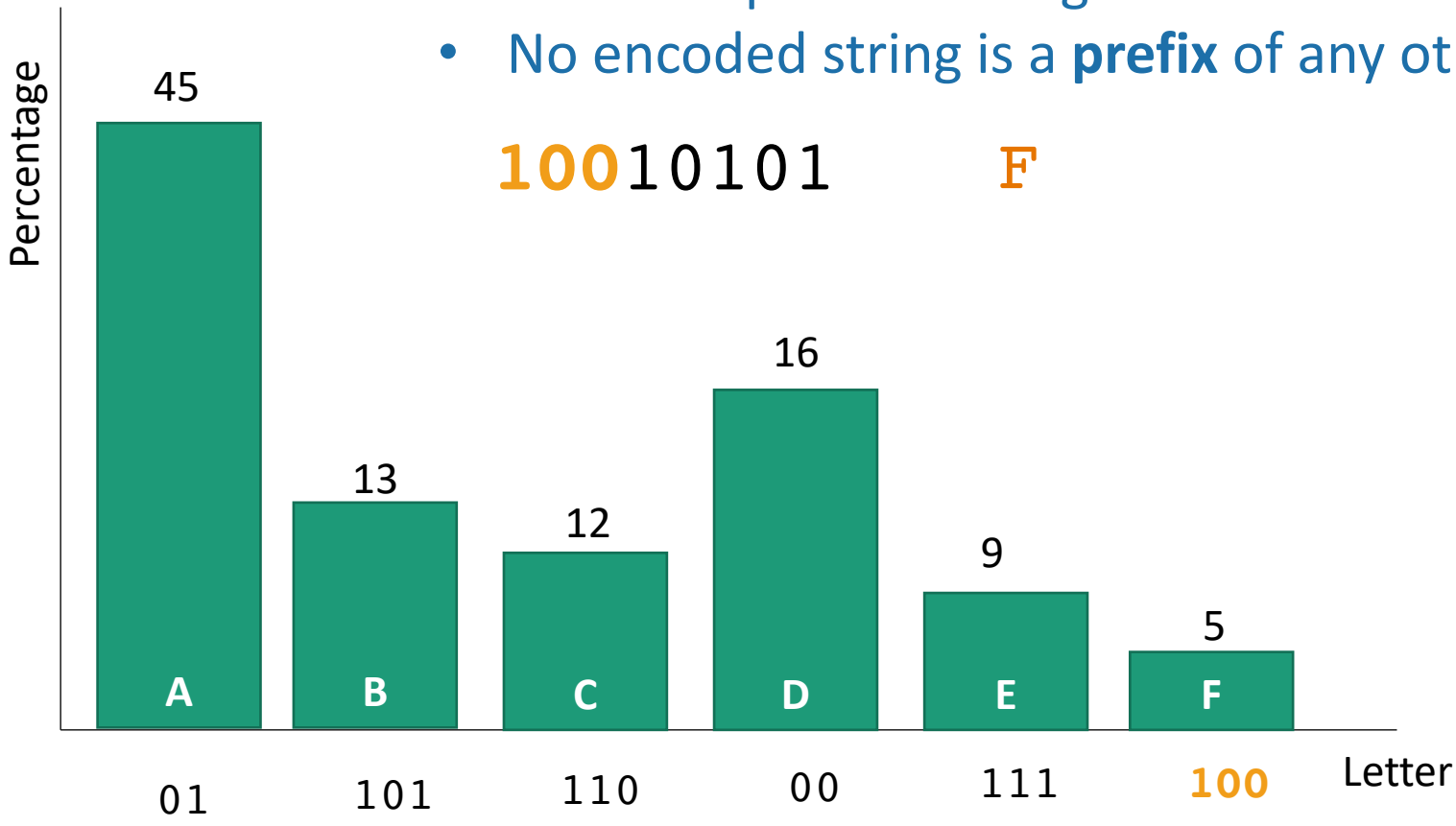


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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

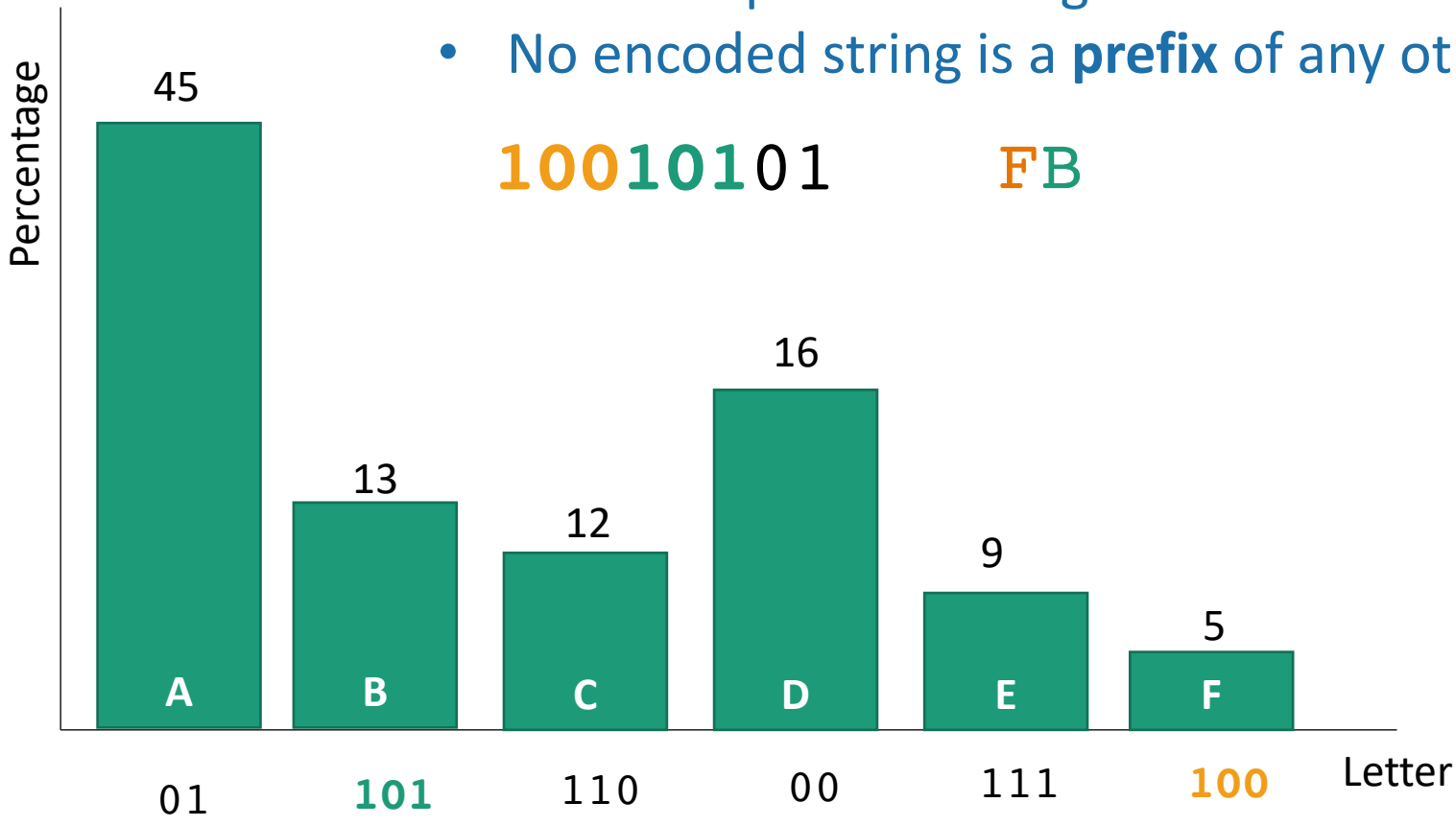


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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**

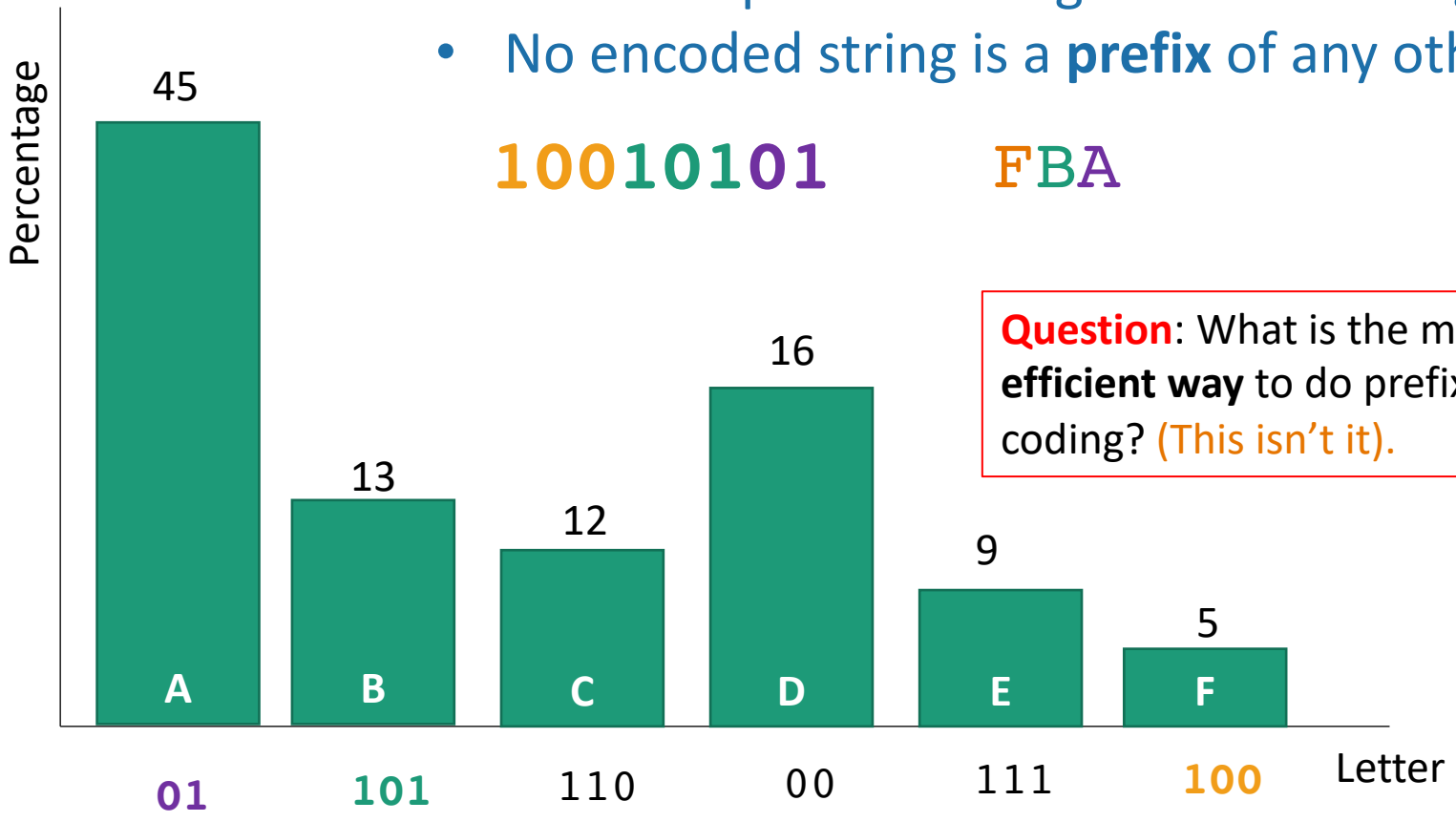


Confusingly, “prefix-free codes” are also sometimes called “prefix codes” (including in CLRS).

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.

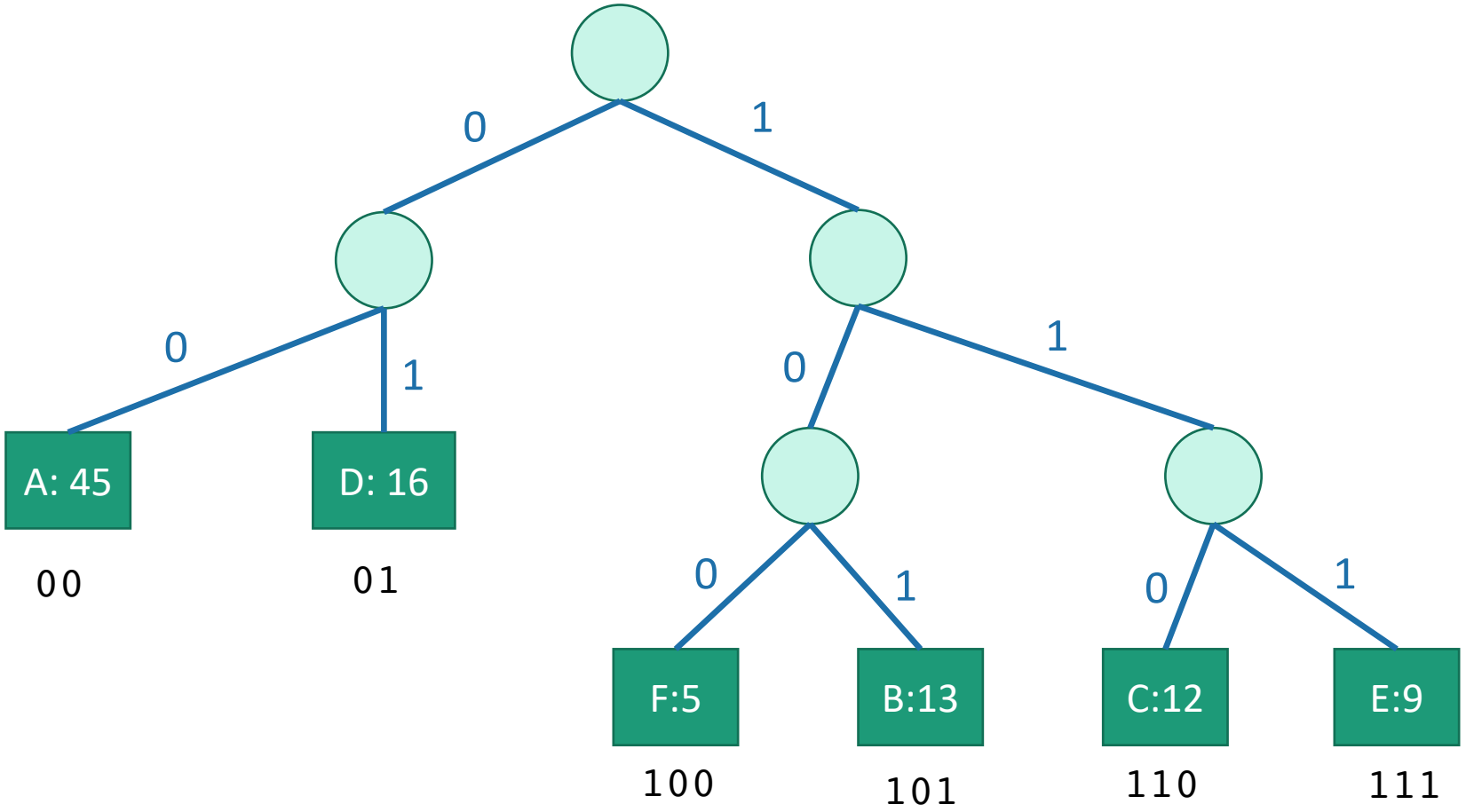
10010101 FBA



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 mea
makes up 13%
characters that e



Some trees are better than others

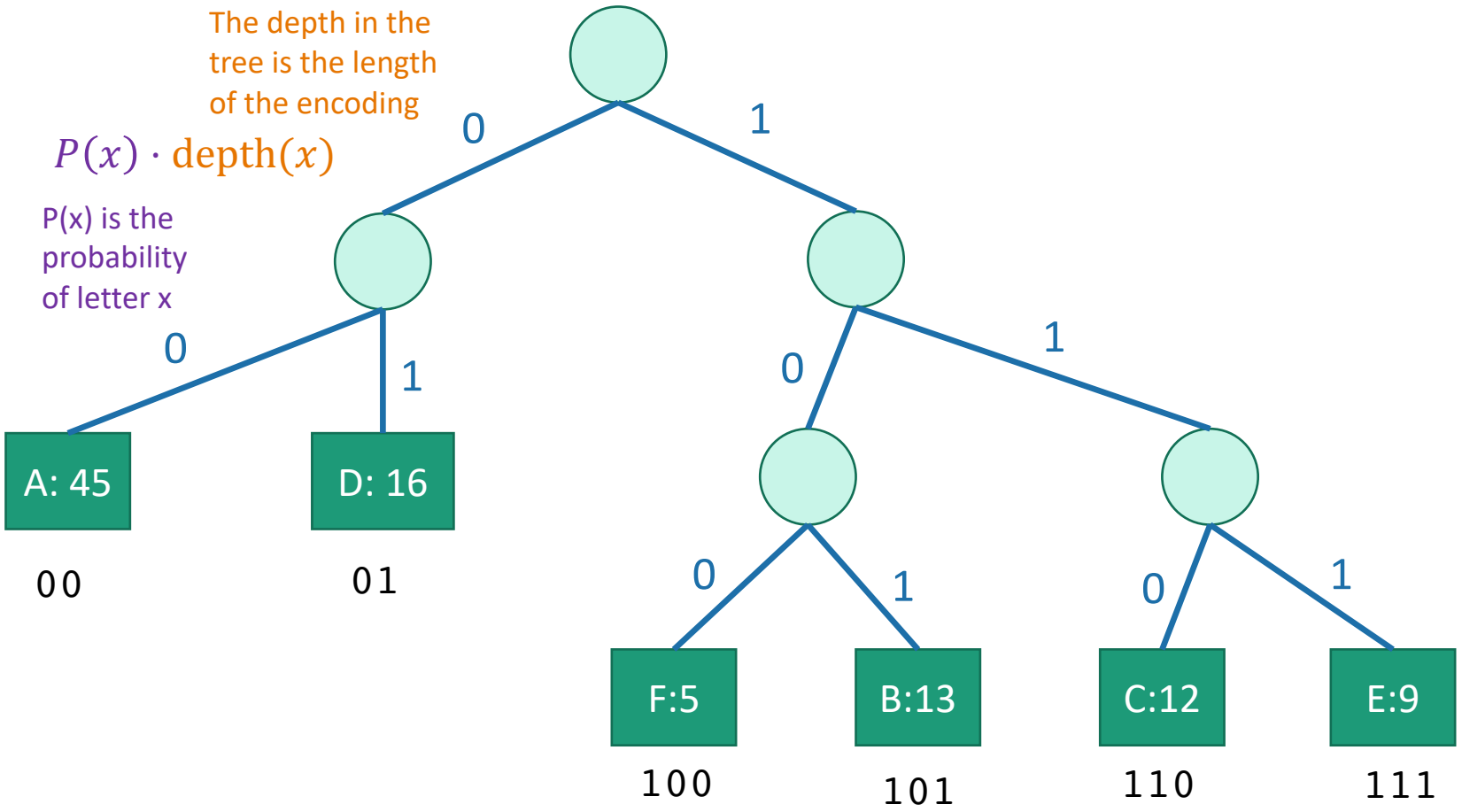
- Imagine choosing a letter at random from the language.
  - Not uniform, but according to our histogram!
- The **cost of a tree** is the expected length of the encoding of that letter.

Cost = $\sum_{leaves\ x}$

The depth in the tree is the length of the encoding

$P(x) \cdot \text{depth}(x)$

P(x) is the probability of letter x



Expected cost of encoding a letter with this tree:

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

Question

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

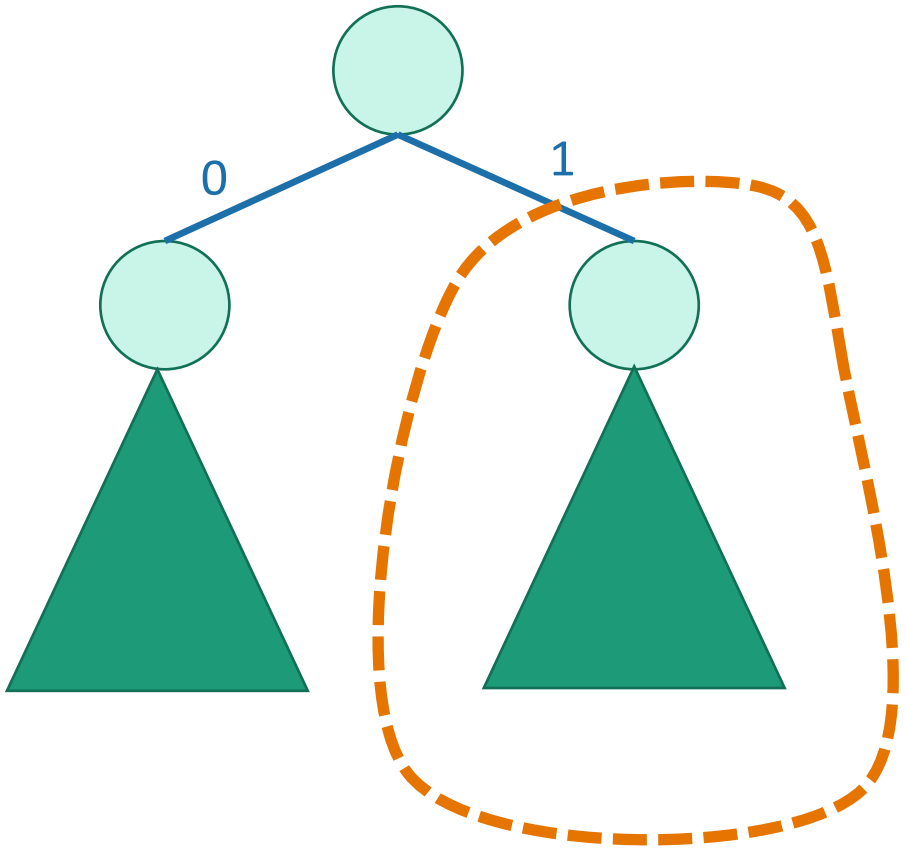
$$\text{cost(tree)} = \sum_{\text{leaves } x} P(x) \cdot \text{depth}(x)$$

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:

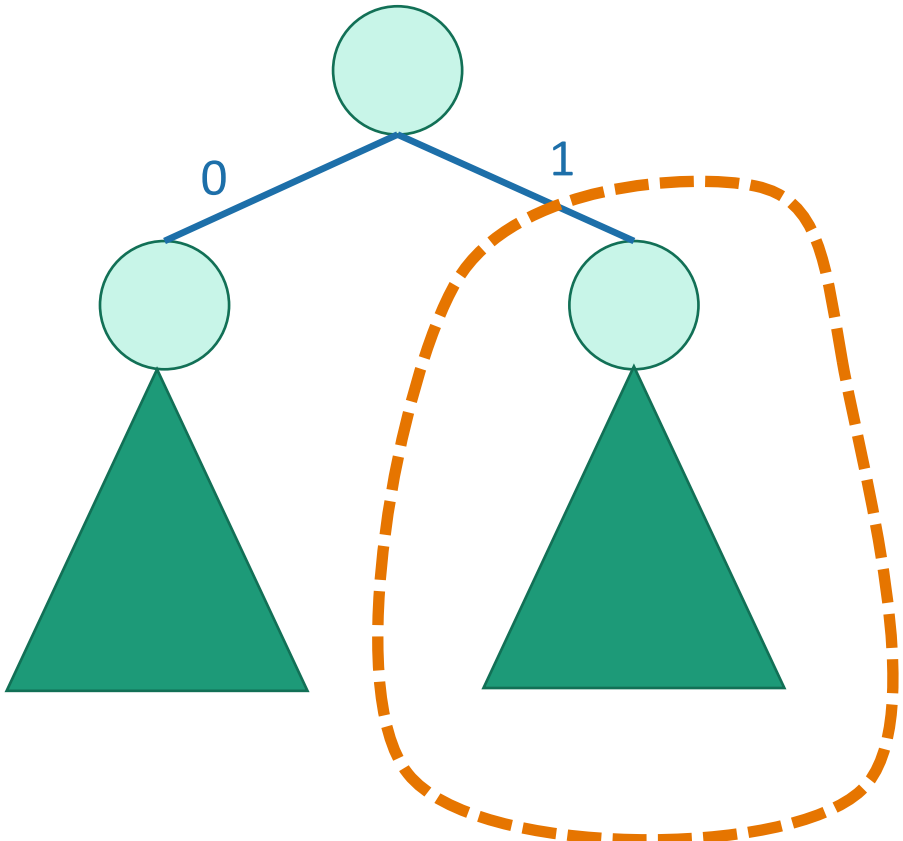


Then this is an optimal tree on fewer letters.

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...

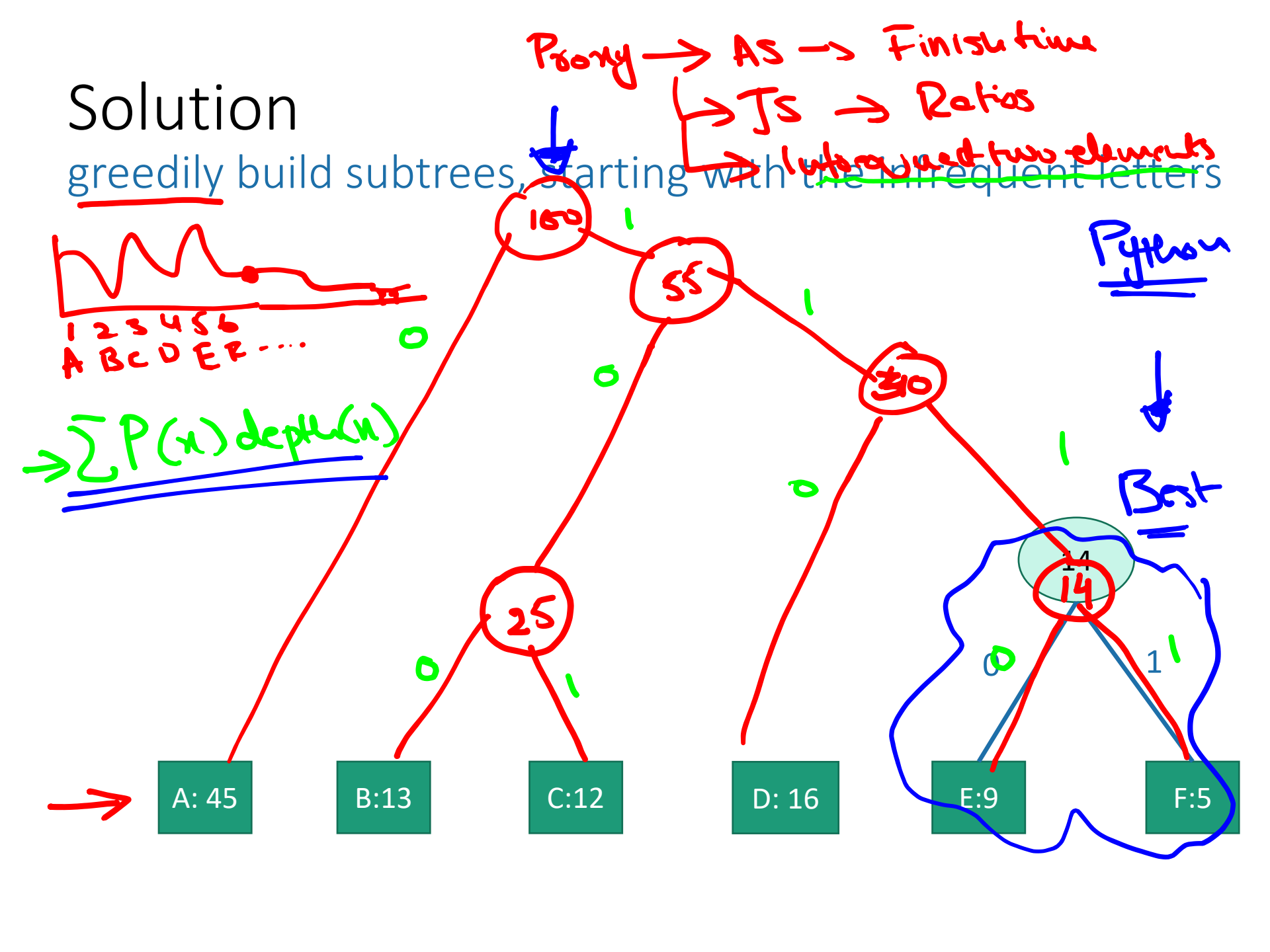


What's a **safe choice** to make for these lower sub-trees?

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

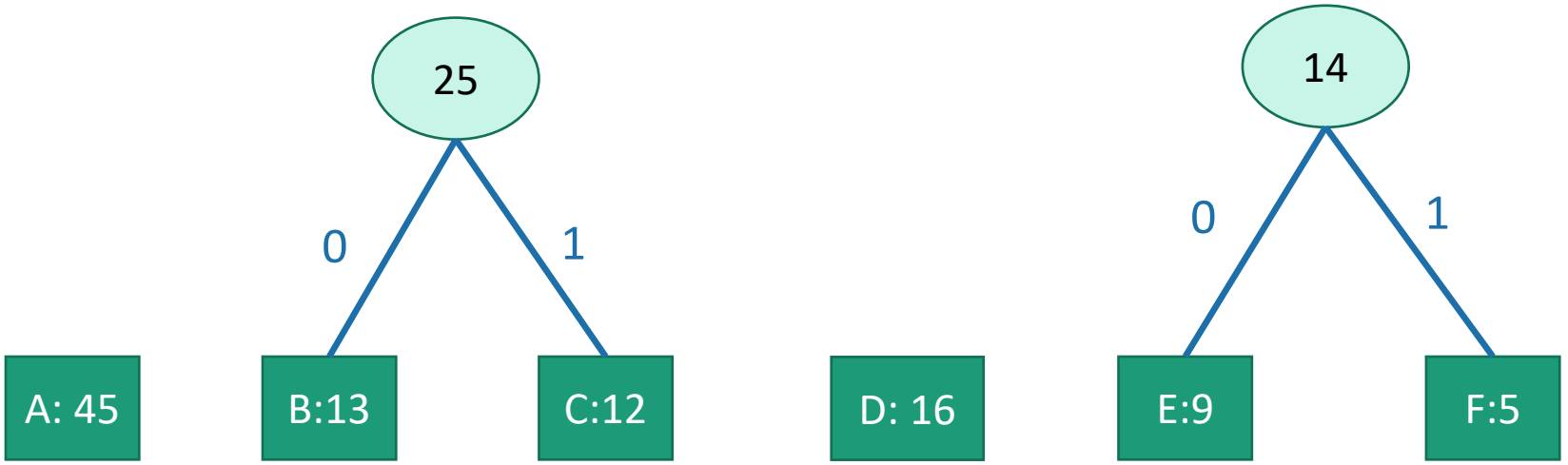
Solution

greedily build subtrees, starting with the infrequent letters



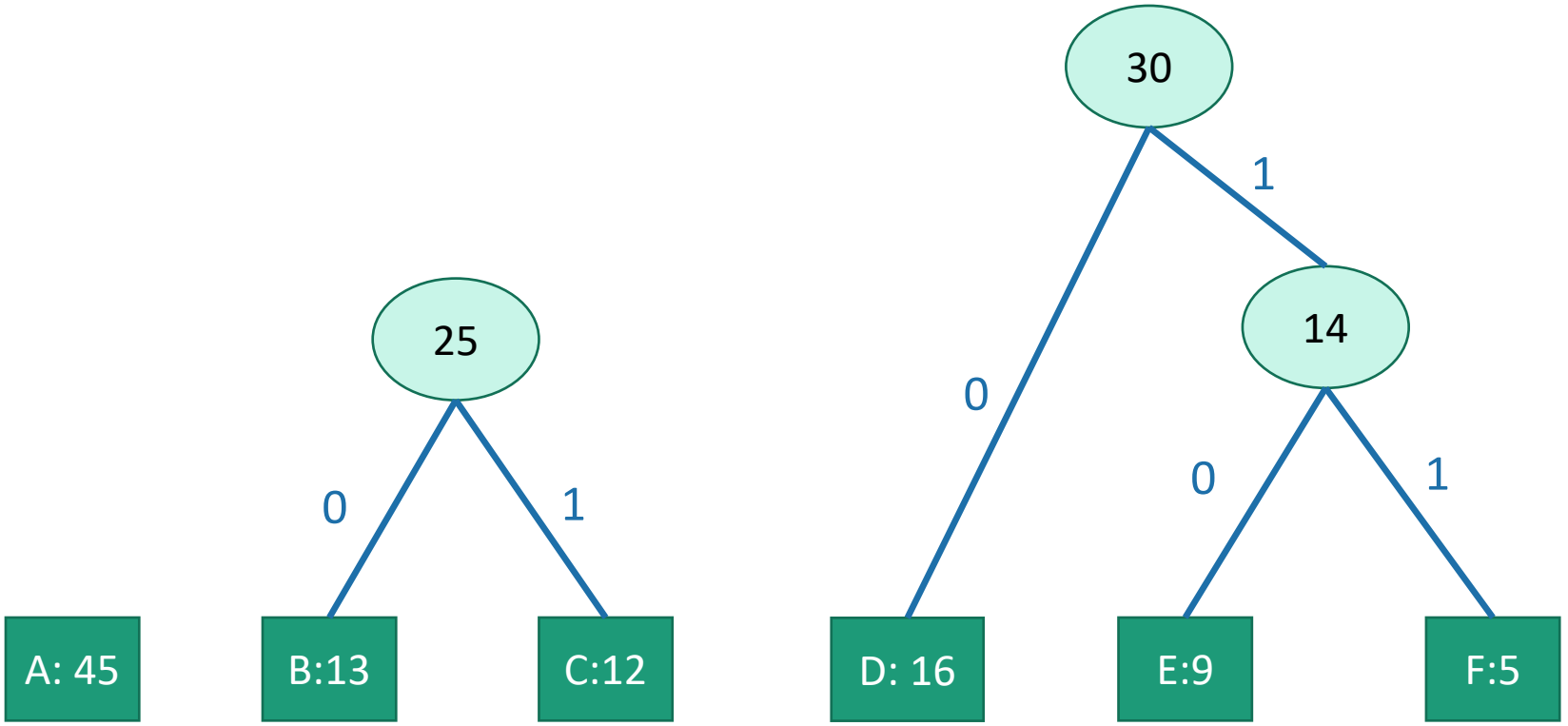
Solution

greedily build subtrees, starting with the infrequent letters



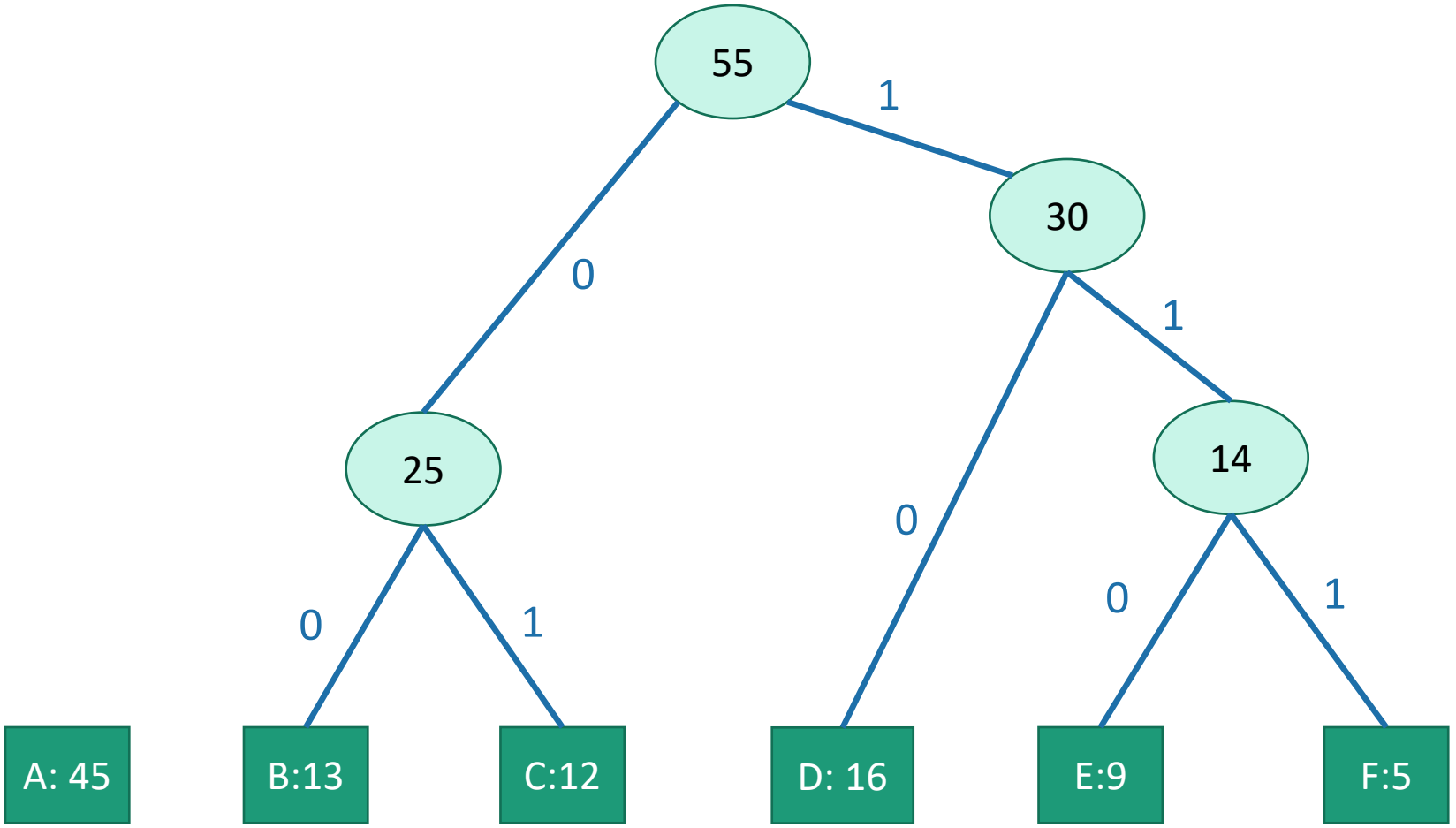
Solution

greedily build subtrees, starting with the infrequent letters



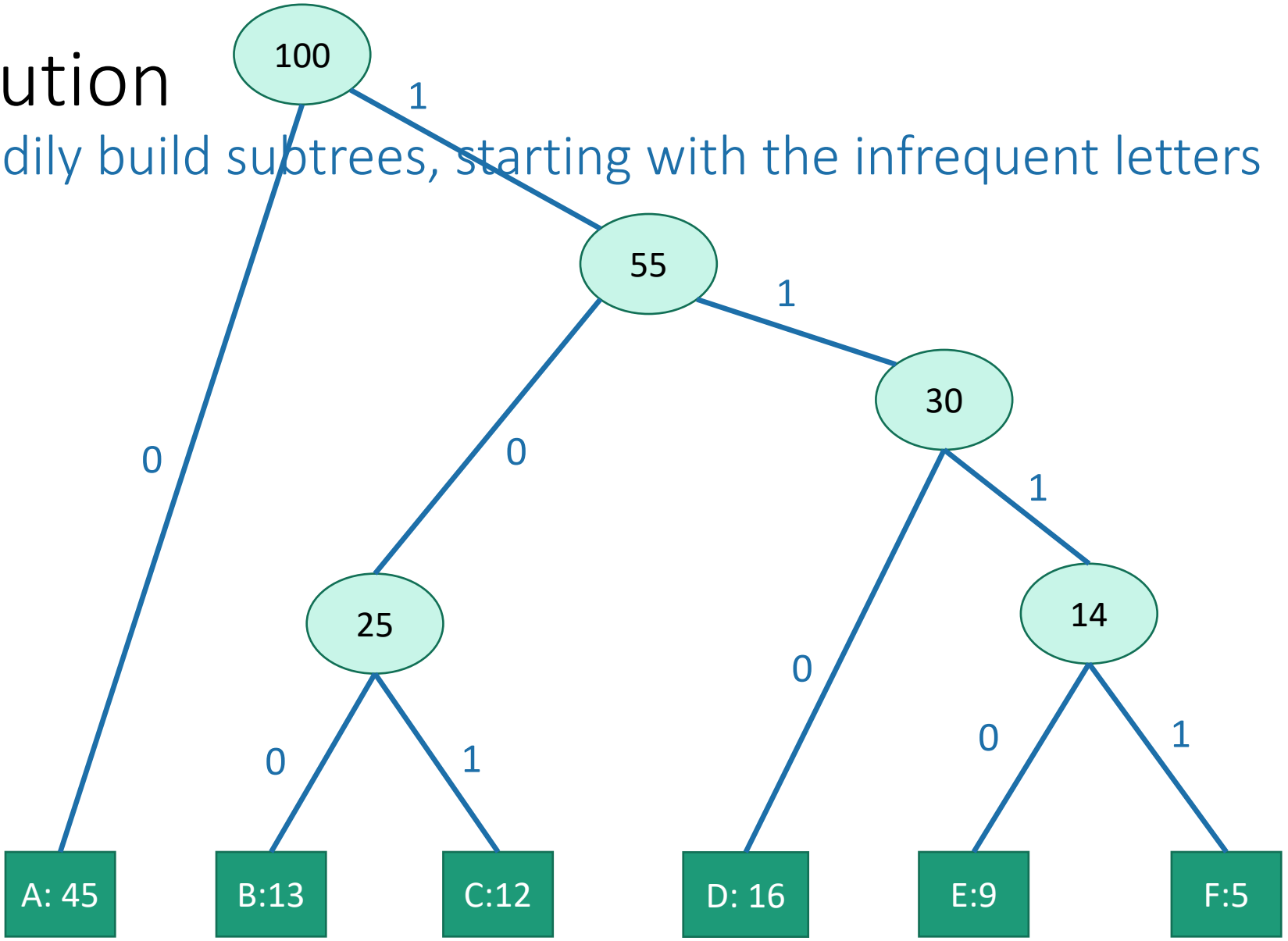
Solution

greedily build subtrees, starting with the infrequent letters



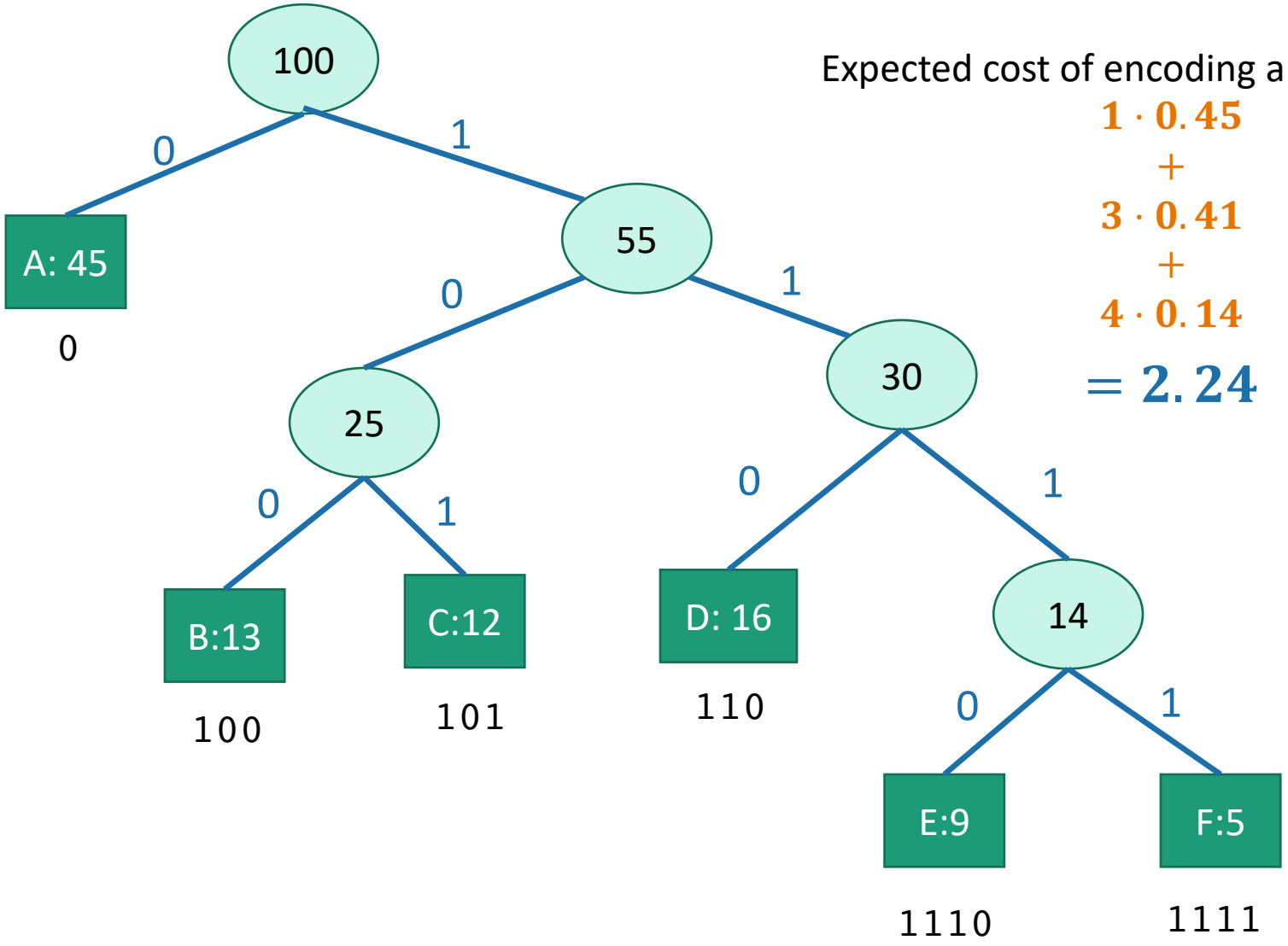
Solution

greedily build subtrees, starting with the infrequent letters



Solution

greedily build subtrees, starting with the infrequent letters



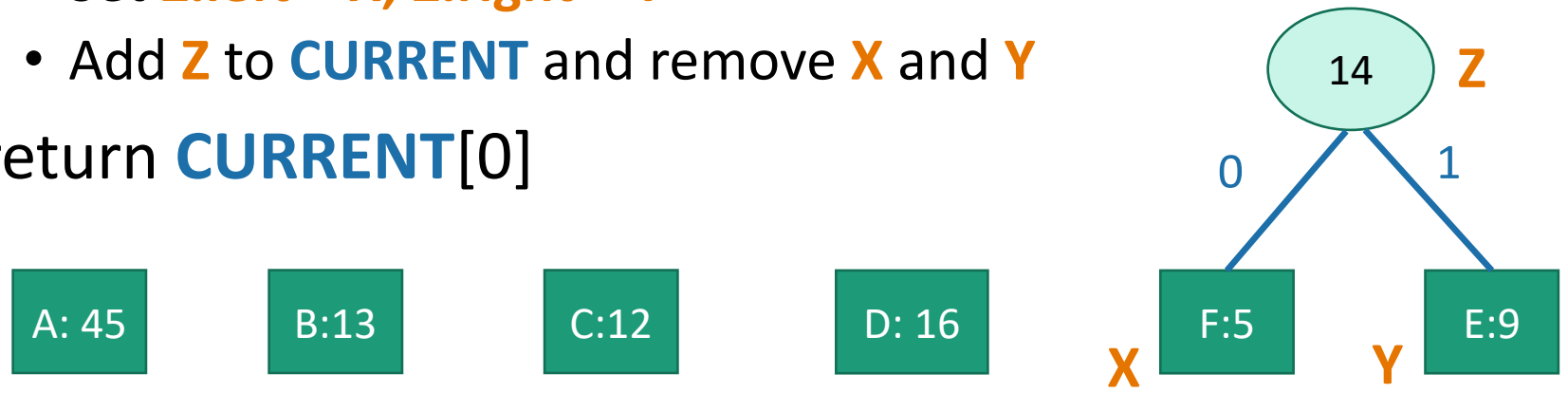
Expected cost of encoding a letter:

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

What exactly was the algorithm?

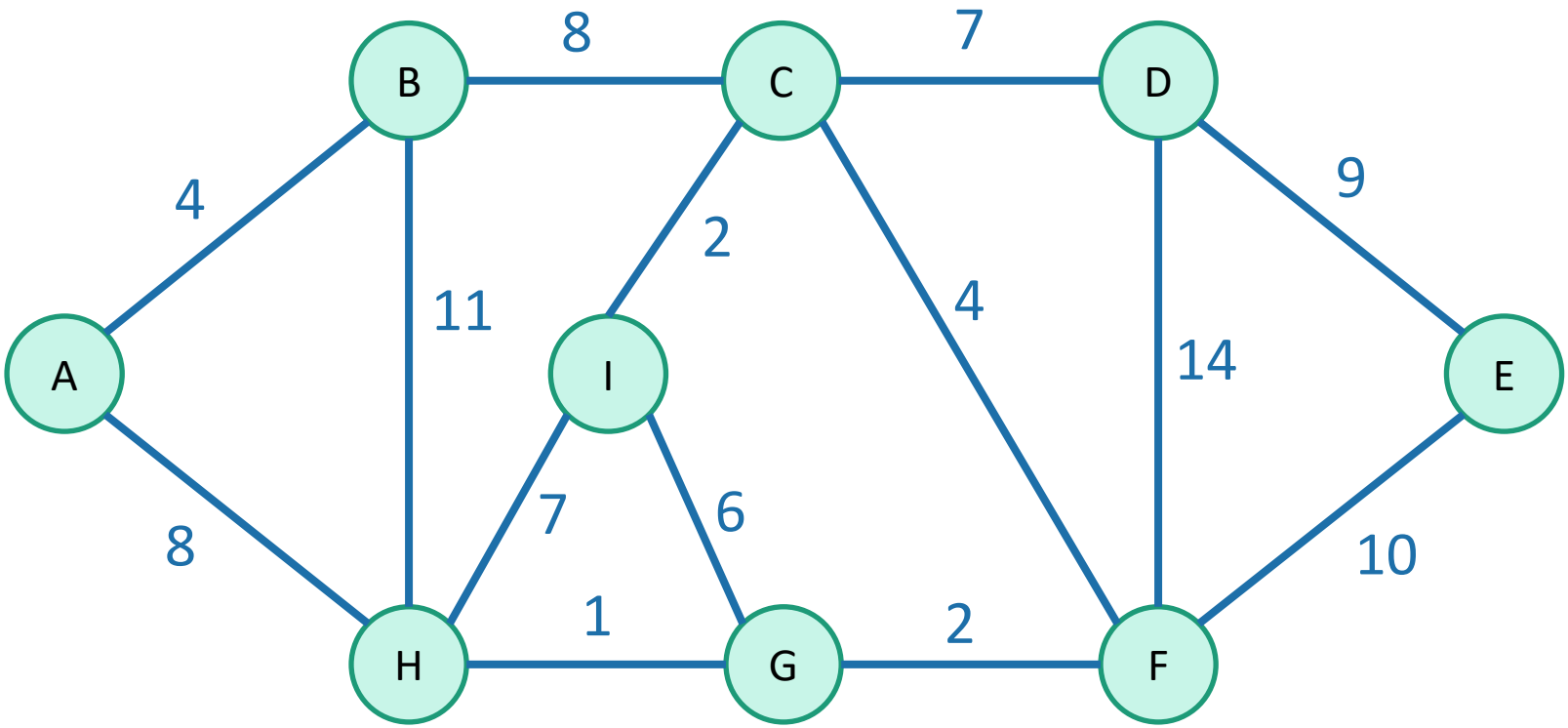
- Create a node like 

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]



Minimum Spanning Tree

Say we have an undirected weighted graph



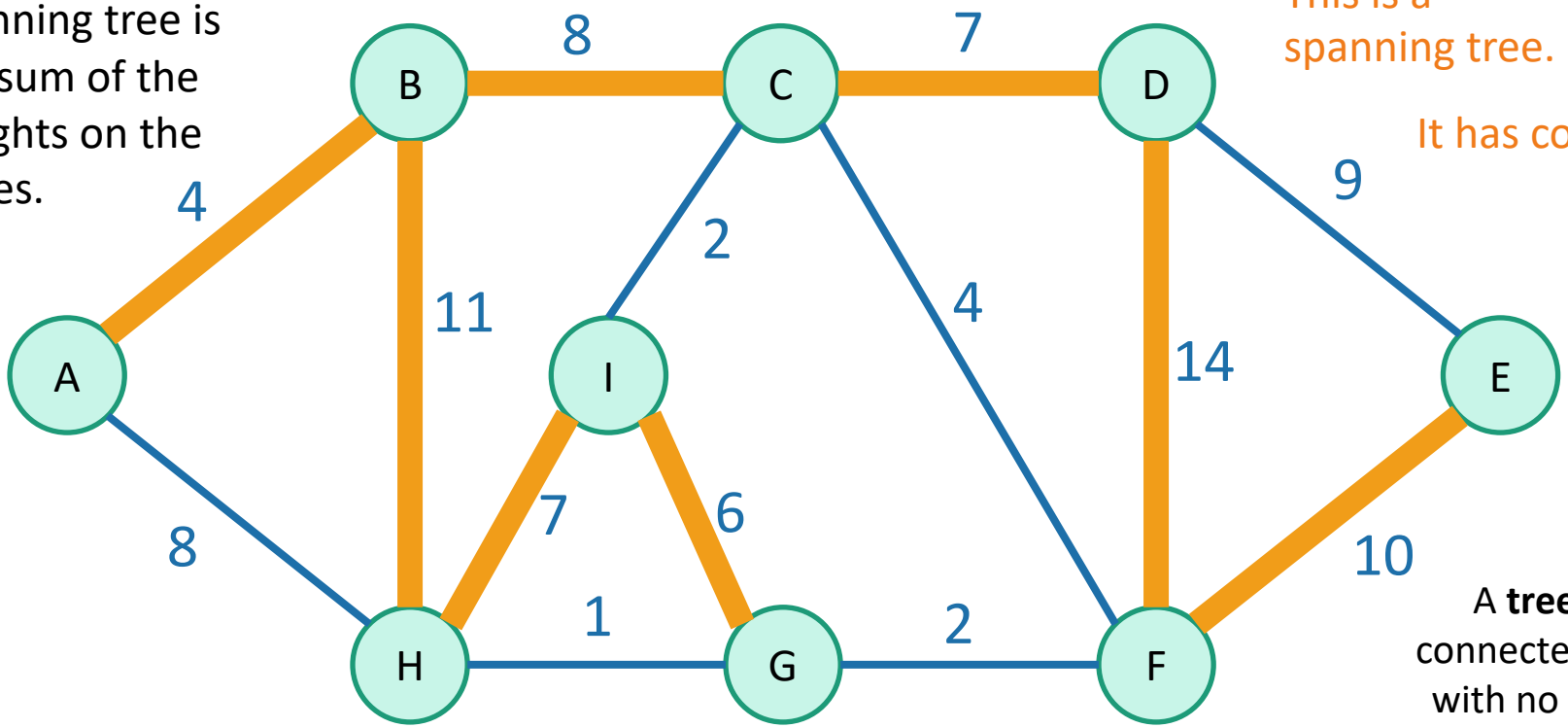
A **spanning tree** is a **tree** that connects all of the vertices.

A **tree** is a connected graph with no cycles!

Minimum Spanning Tree

Say we have an undirected weighted graph

The **cost** of a spanning tree is the sum of the weights on the edges.



This is a spanning tree.

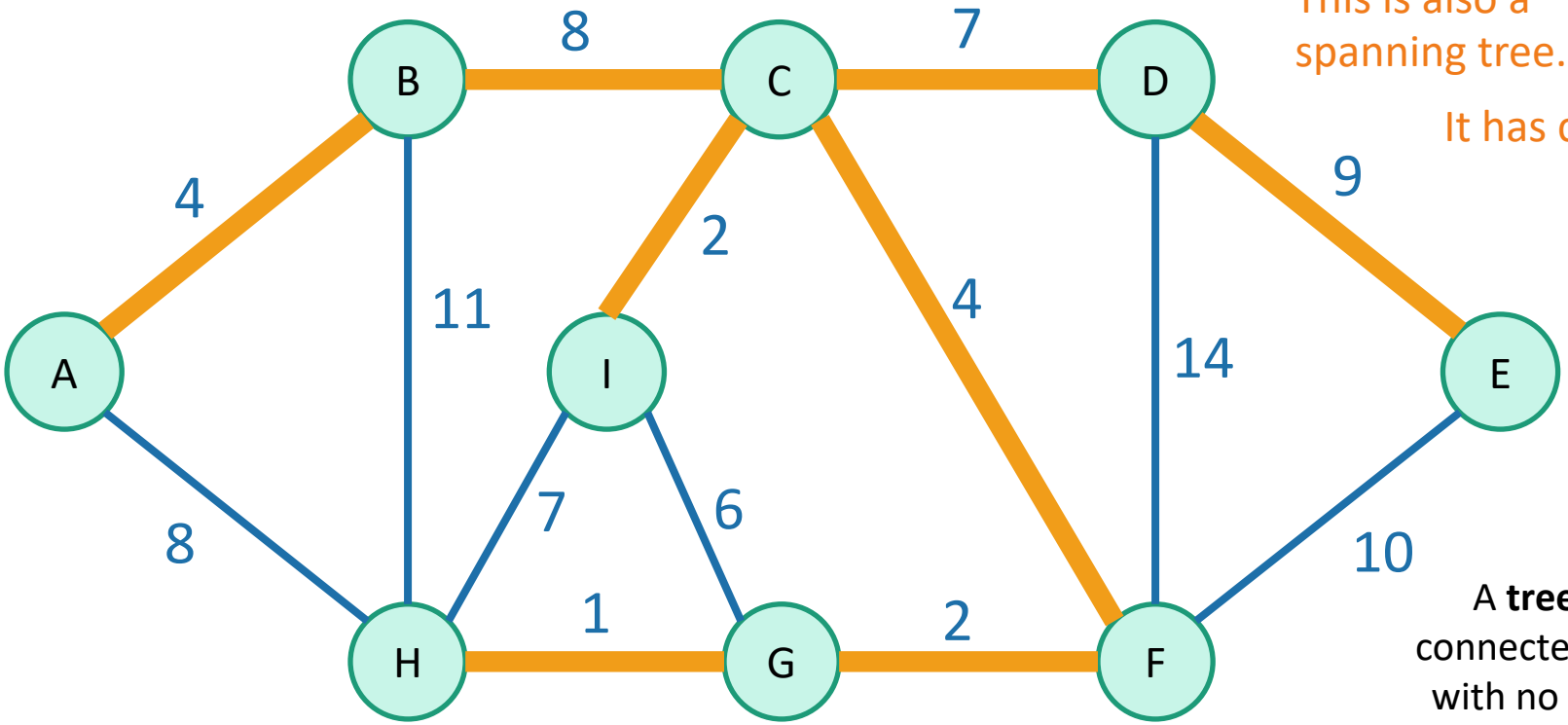
It has cost 67

A **tree** is a connected graph with no cycles!

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree

Say we have an undirected weighted graph



This is also a spanning tree.

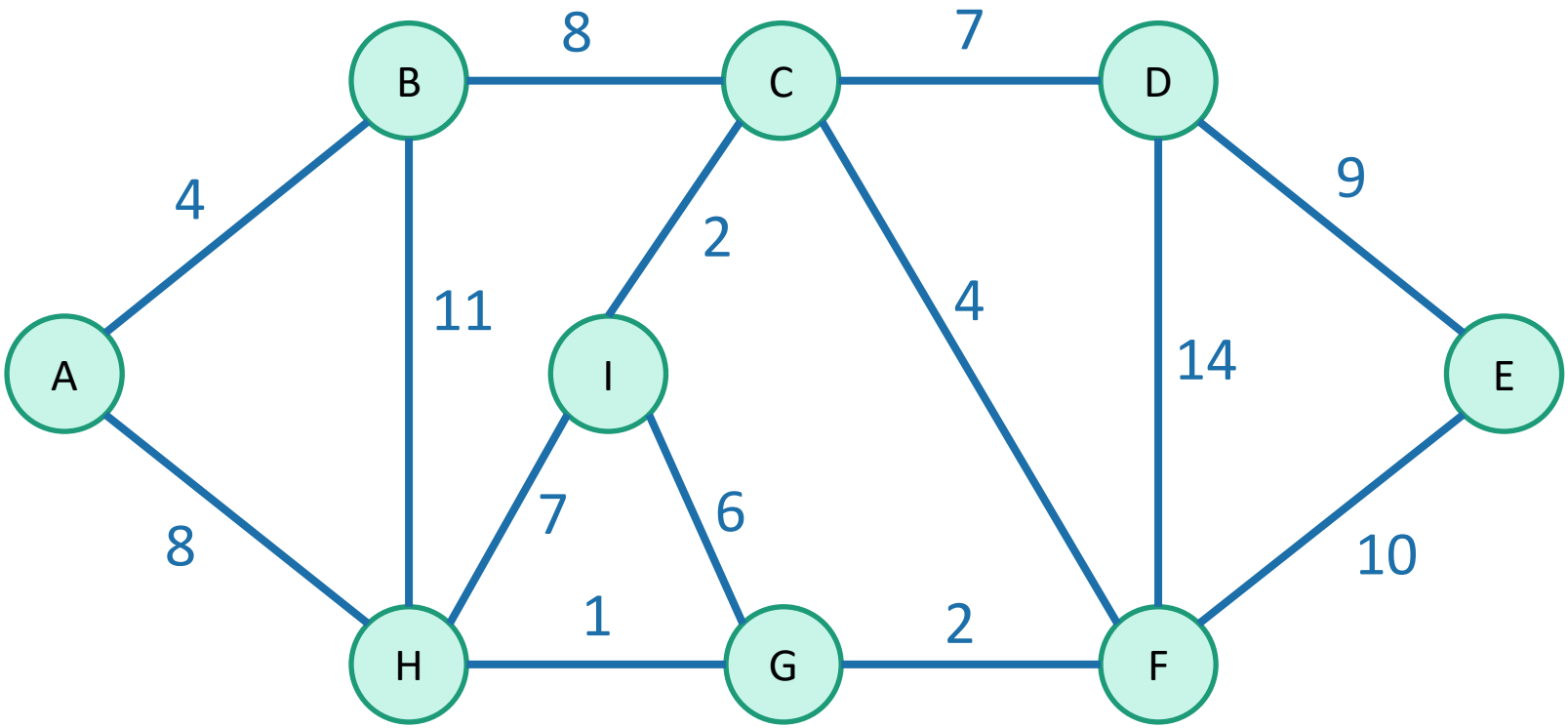
It has cost 37

A **tree** is a connected graph with no cycles!

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree

Say we have an undirected weighted graph

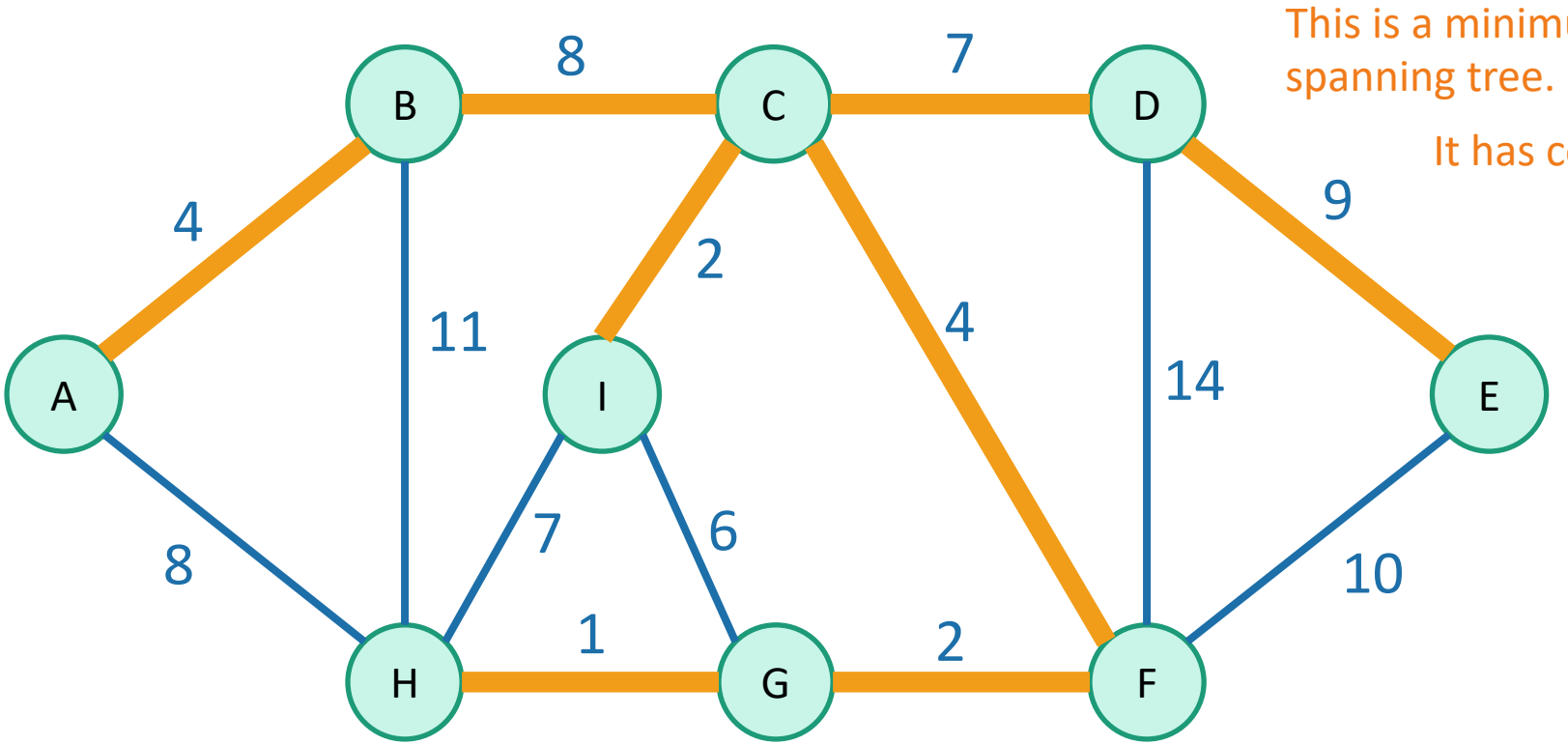


minimum **of minimal cost**

A **spanning tree** is a **tree** that connects all of the vertices.

Minimum Spanning Tree

Say we have an undirected weighted graph



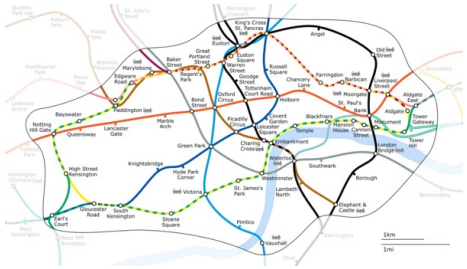
A **spanning tree** is a **tree** that connects all of the vertices.

minimum

of minimal cost

Why MSTs?

- Network design
  - Connecting cities with roads/electricity/telephone/…
- cluster analysis
  - eg, genetic distance
- image processing
  - eg, image segmentation
- Useful primitive
  - for other graph algs



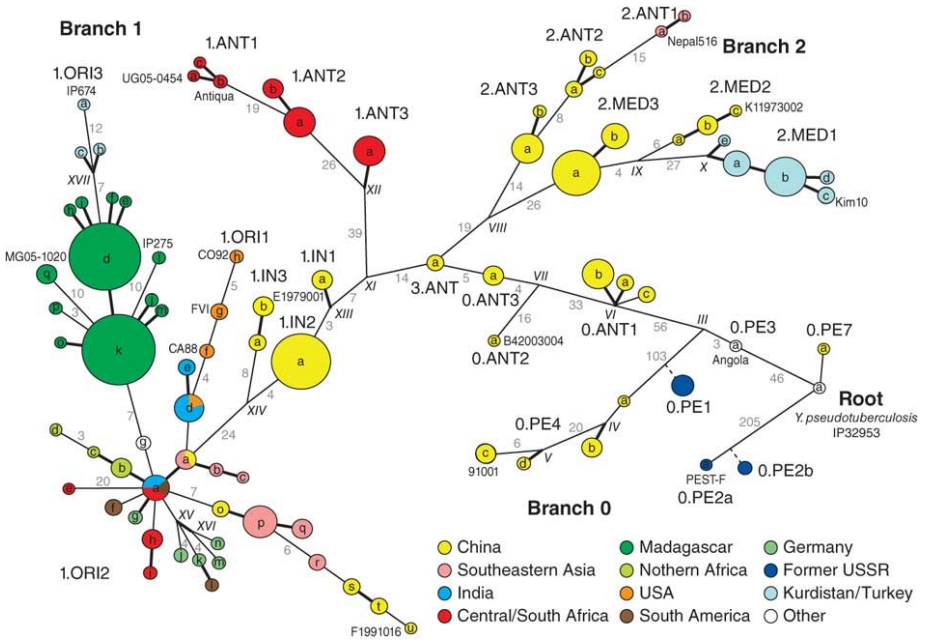


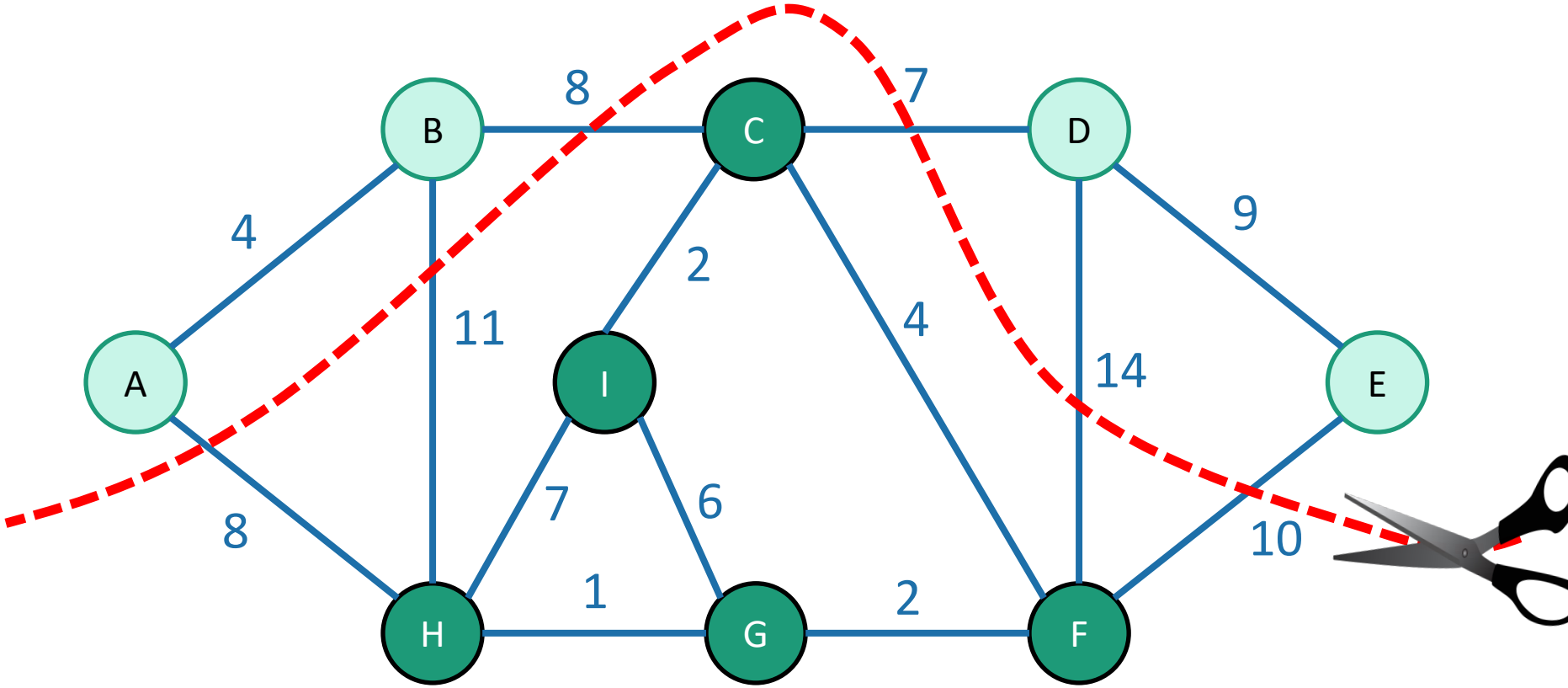
Figure 2: Fully parsimonious minimal spanning tree of 933 SNPs for 282 isolates of *Y. pestis* colored by location.
Morelli et al. Nature genetics 2010

Brief aside

for a discussion of cuts in graphs!

Cuts in graphs

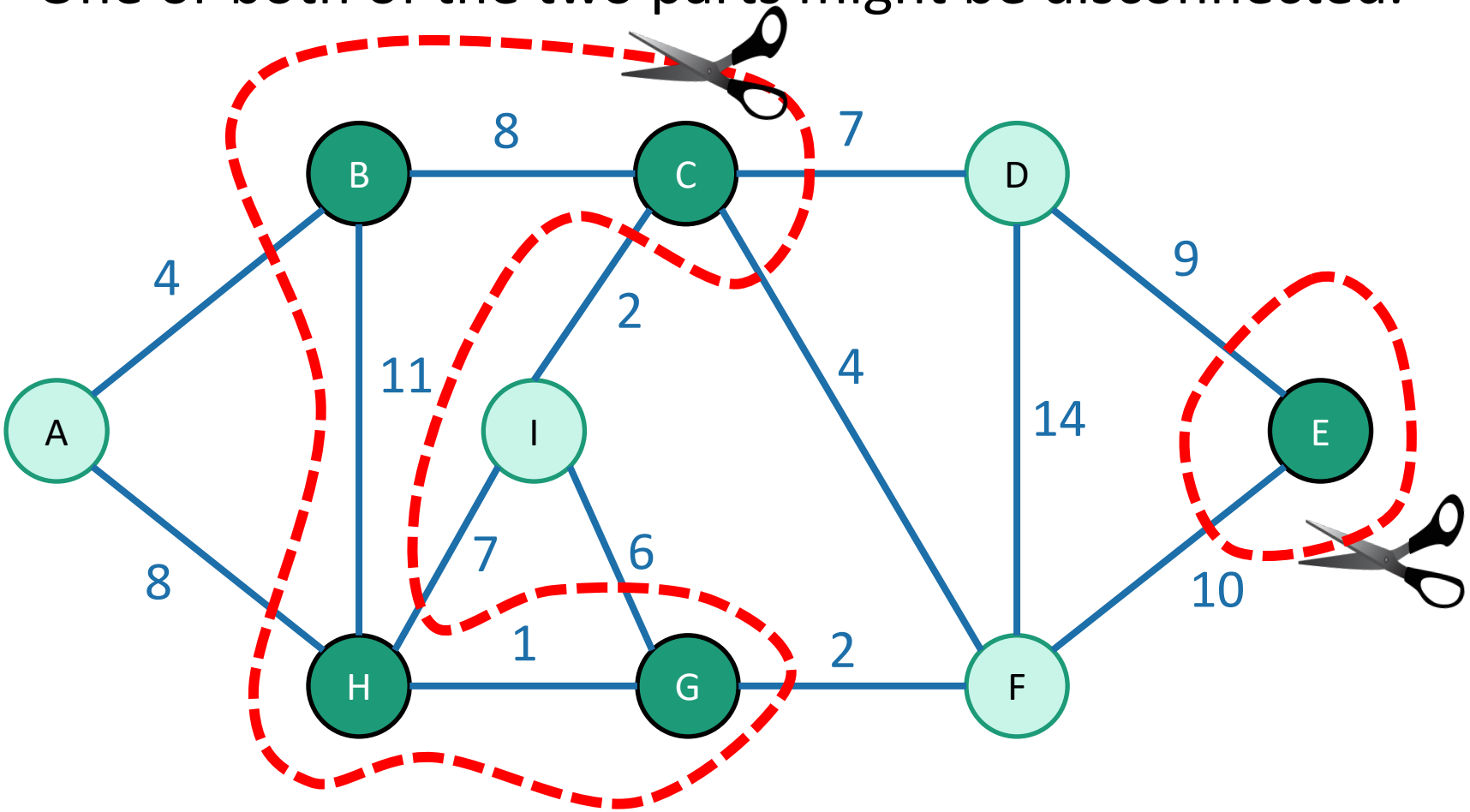
- A cut is a partition of the vertices into two parts:



This is the cut **"{A,B,D,E} and {C,I,H,G,F}"**

Cuts in graphs

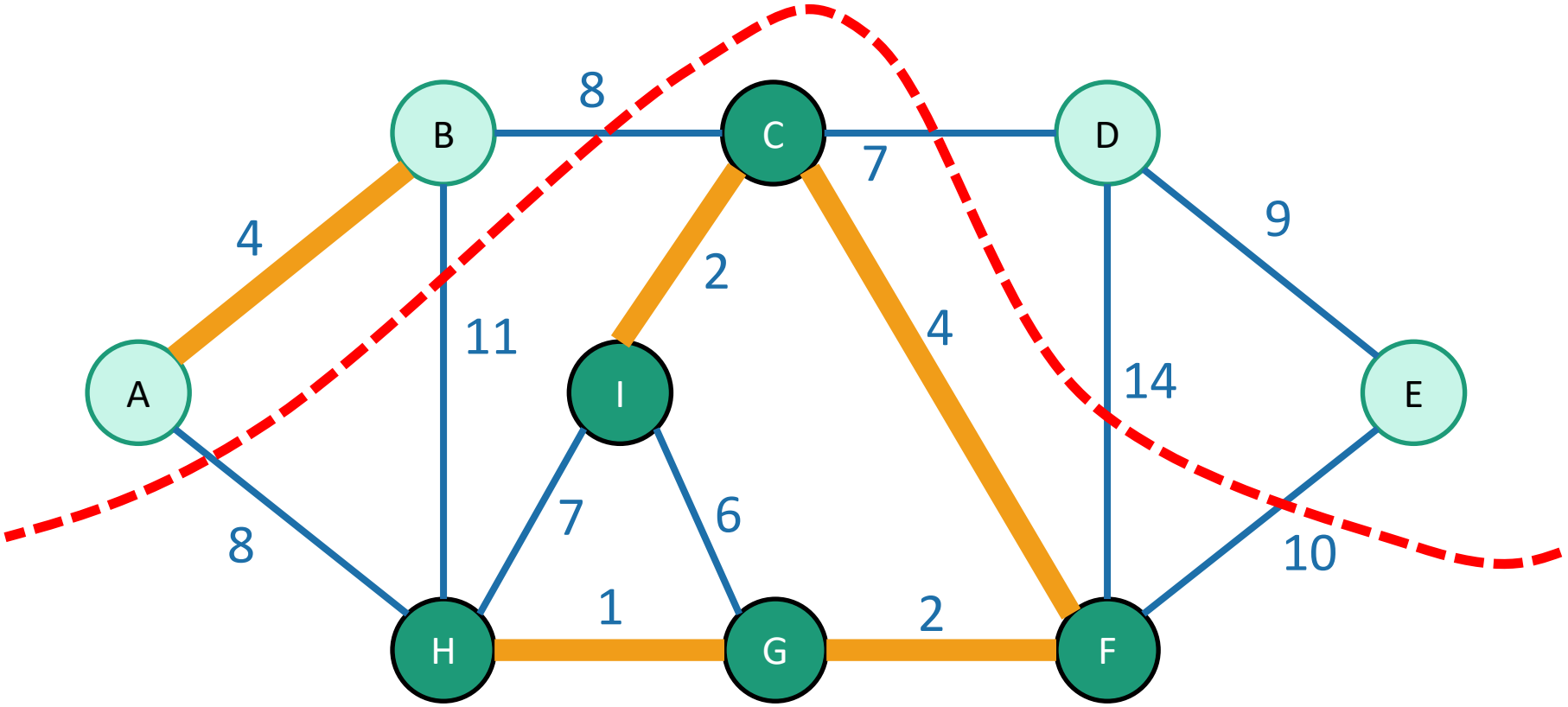
- One or both of the two parts might be disconnected.



This is the cut **"{B,C,E,G,H} and {A,D,I,F}"**

Let S be a set of edges in G

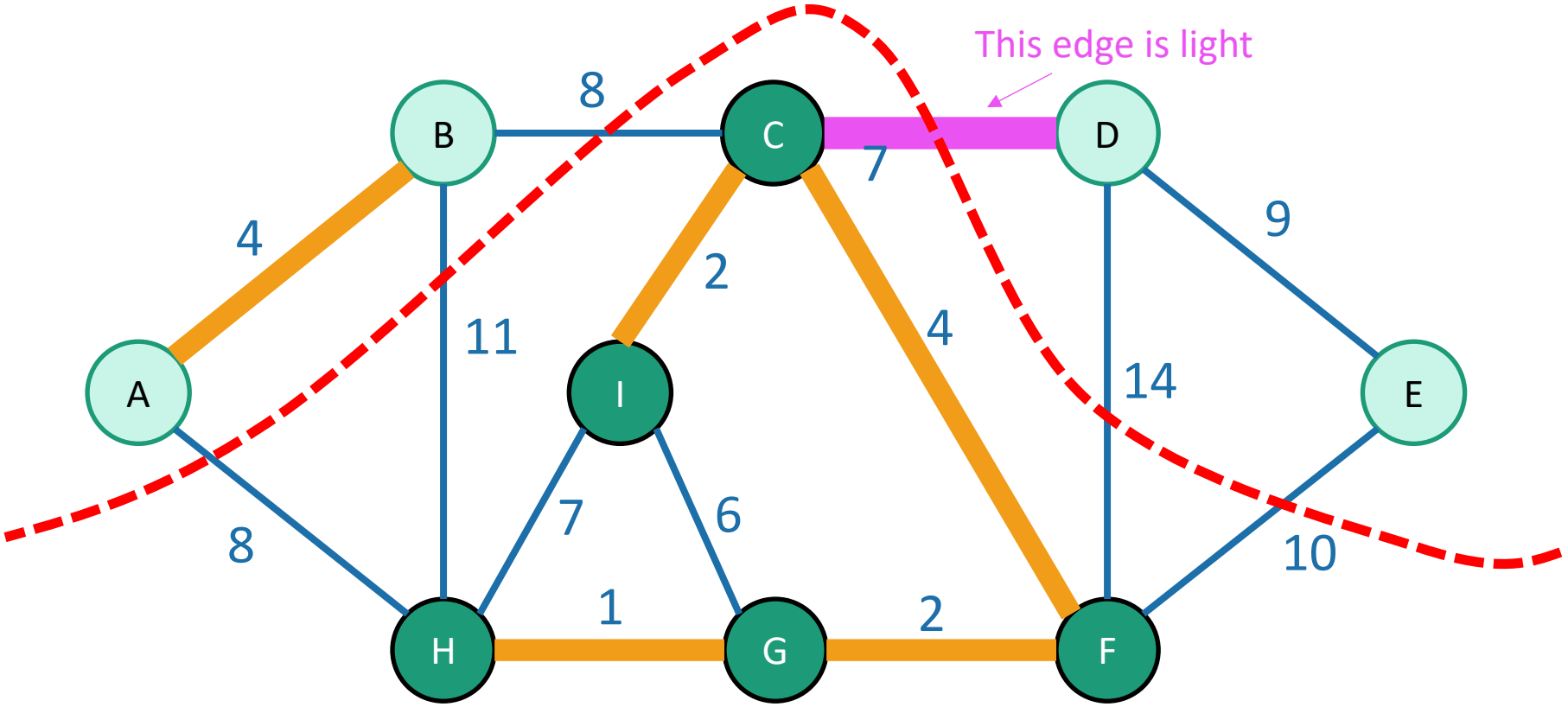
- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



S is the set of **thick orange** edges

Let S be a set of edges in G

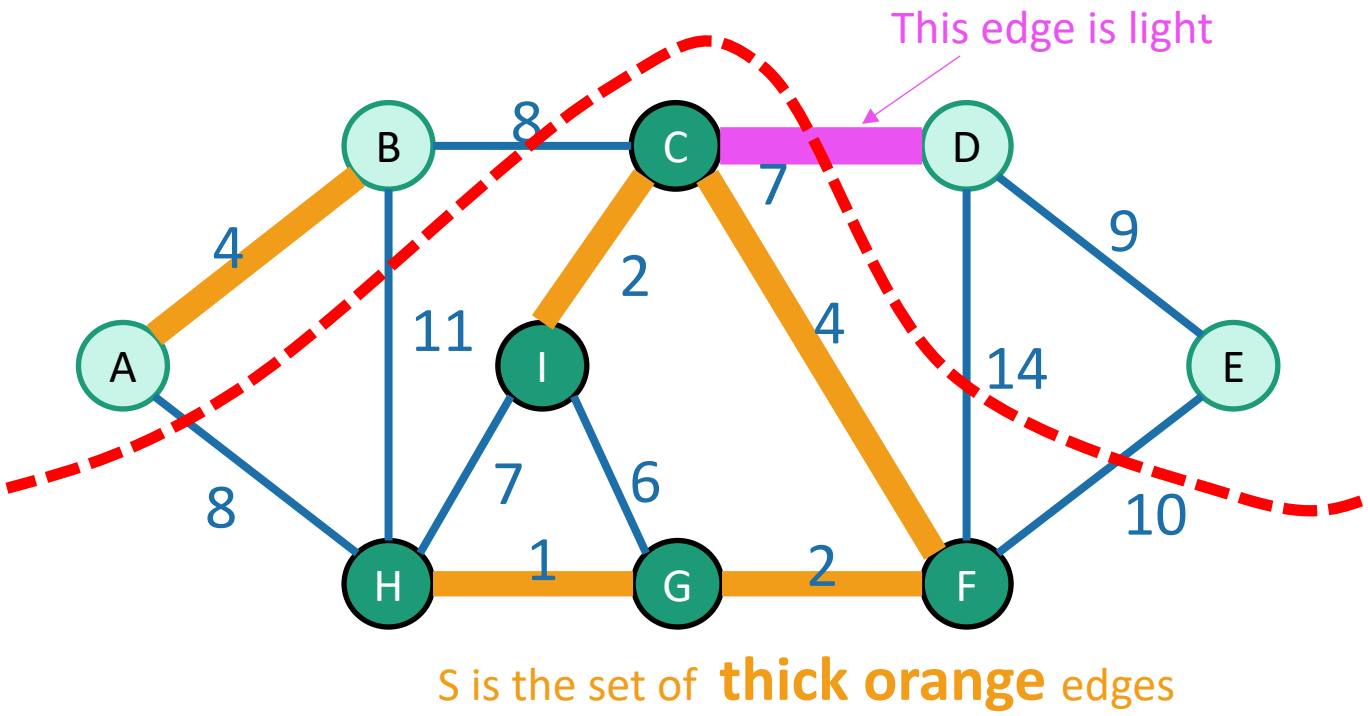
- We say a cut **respects** S if no edges in S cross the cut.
- An edge crossing a cut is called **light** if it has the smallest weight of any edge crossing the cut.



S is the set of **thick orange** edges

Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}

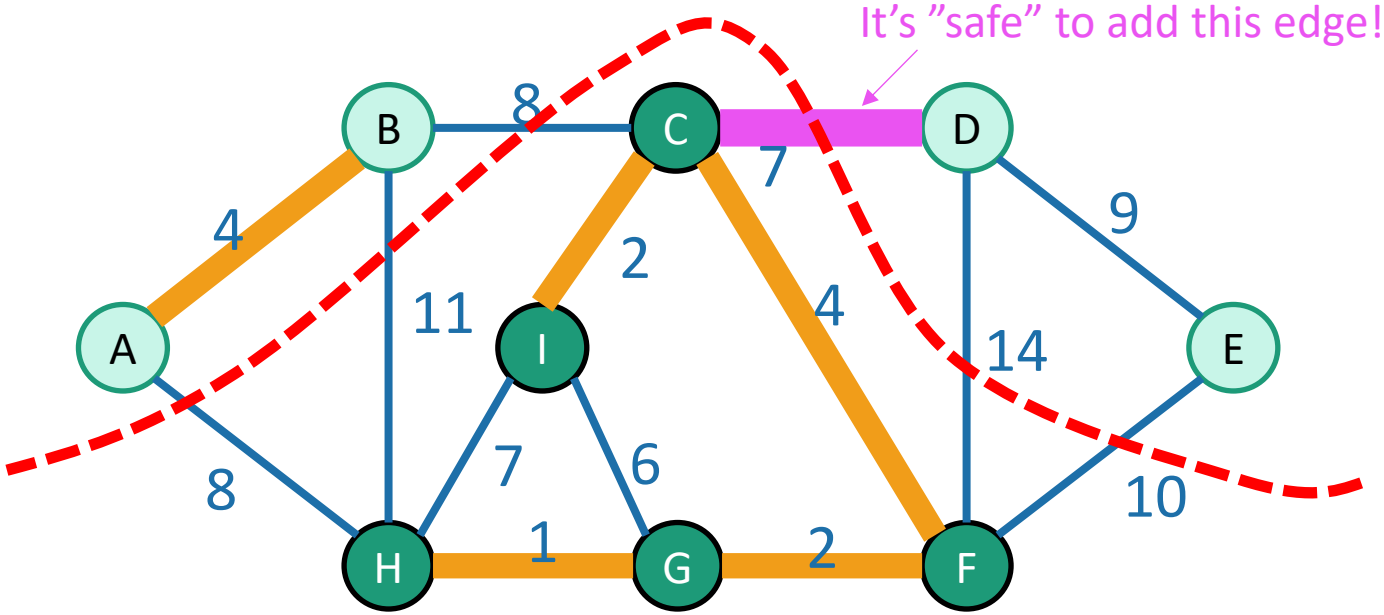


Lemma

- Let S be a set of edges, and consider a cut that respects S.
- Suppose there is an MST containing S.
- Let (u,v) be a light edge.
- Then there is an MST containing S ∪ {(u,v)}

Aka:

If we haven't ruled out the possibility of success so far, then adding a light edge still won't rule it out.



S is the set of **thick orange** edges

End aside

Back to MSTs!

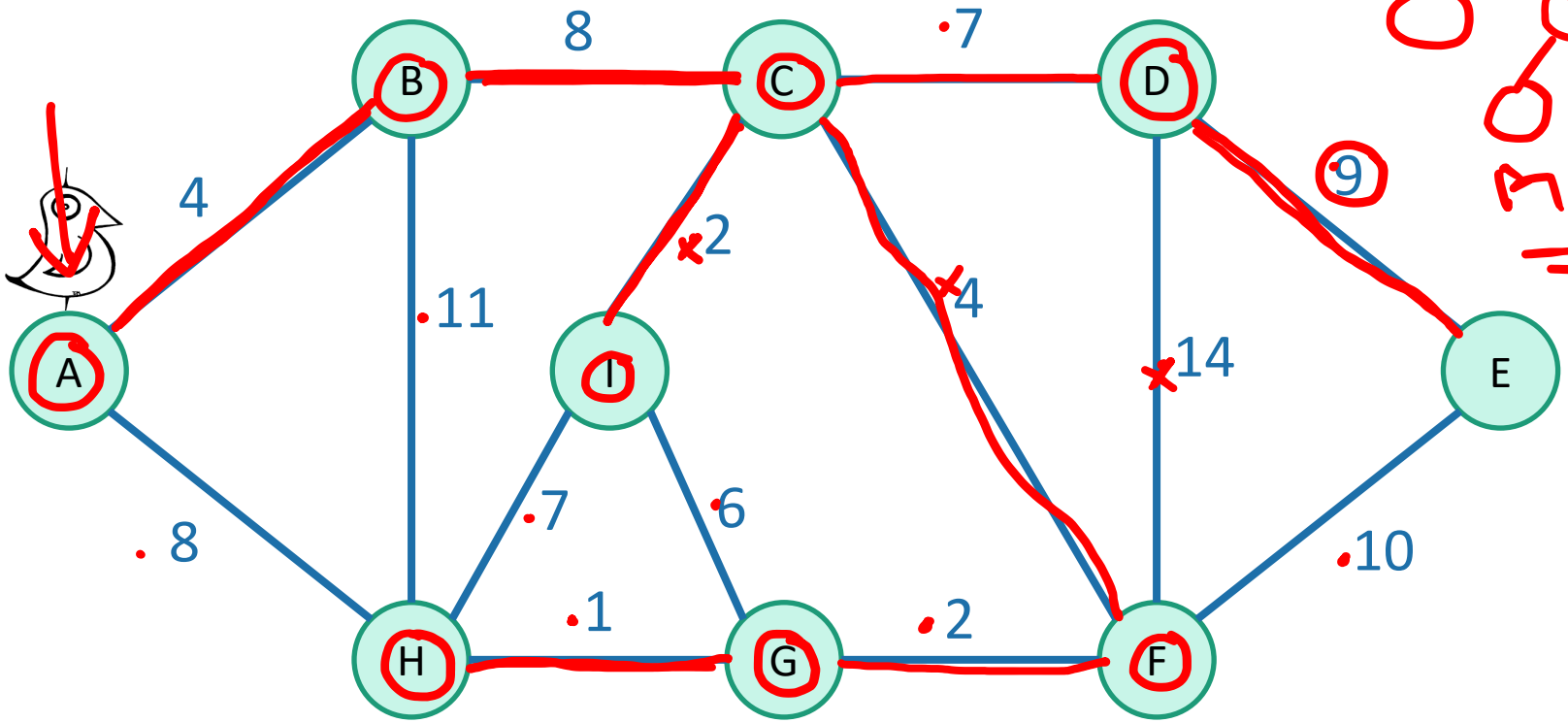
Back to MSTs

- How do we find one?
- Let us see **two greedy algorithms**.

- The strategy:
  - Make a **series of choices**, adding edges to the tree.
  - Show that each edge we add is **safe to add**:
    - we do not rule out the possibility of success
    - we will choose **light edges** crossing **cuts** and **use the Lemma**.
  - **Keep going** until we have an MST.

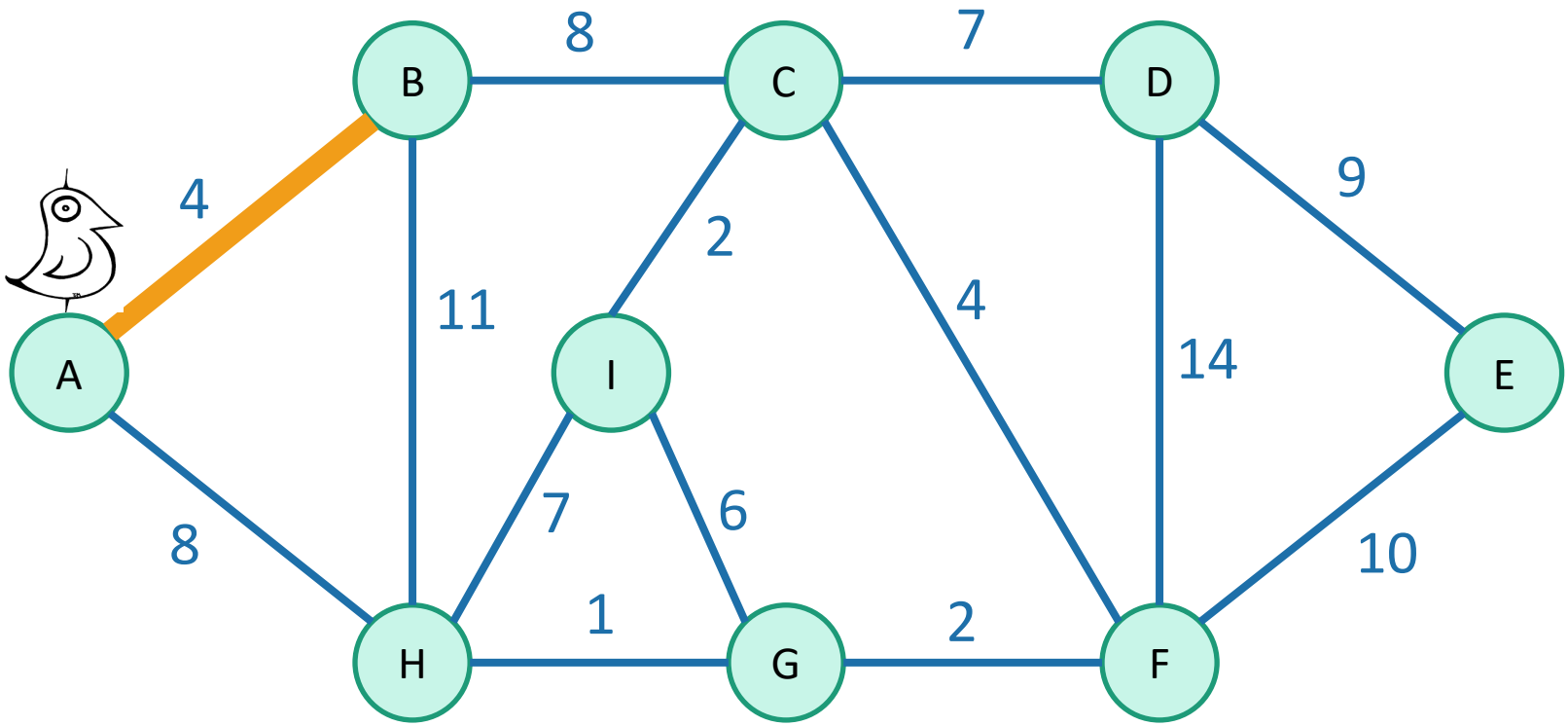
Idea 1 (MST)

Start growing a tree, greedily add the shortest edge we can to grow the tree.



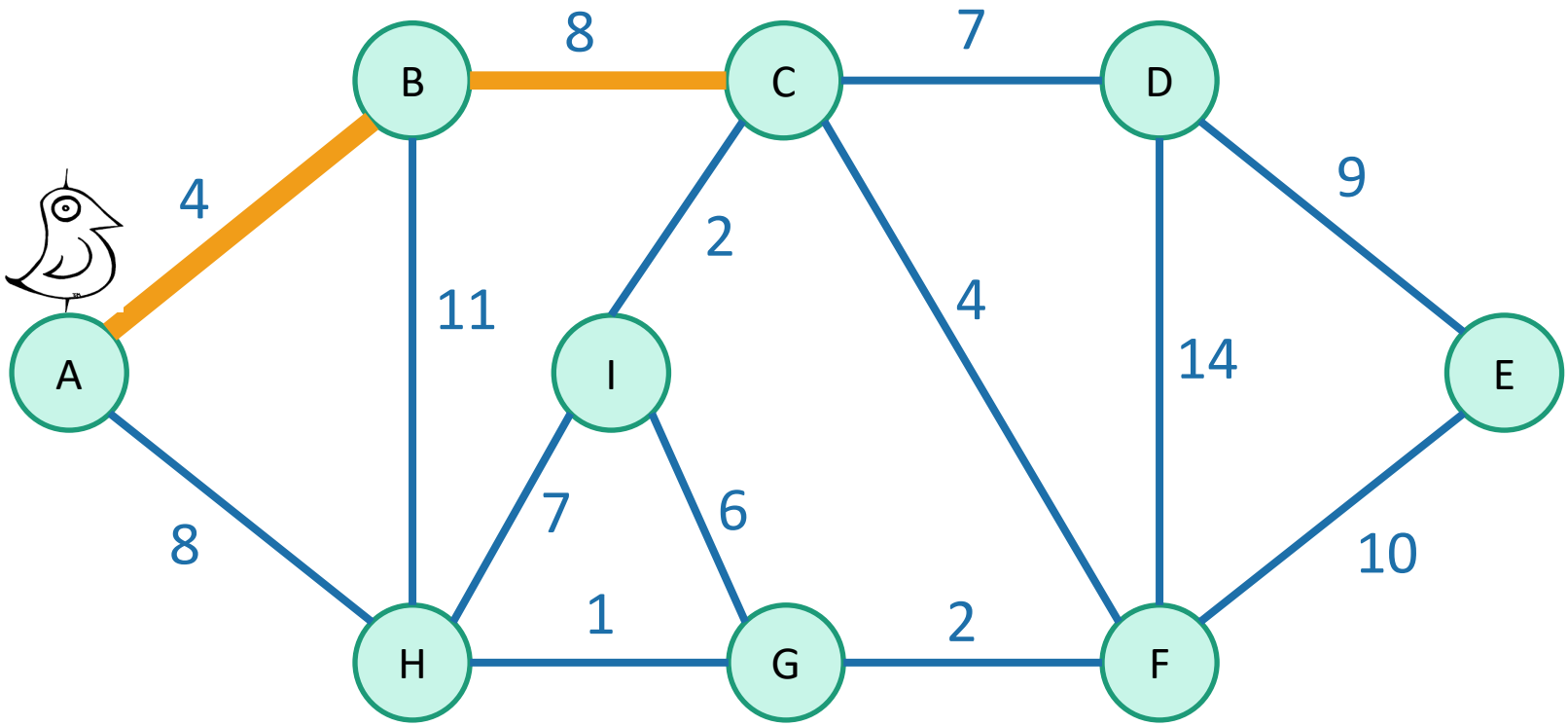
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



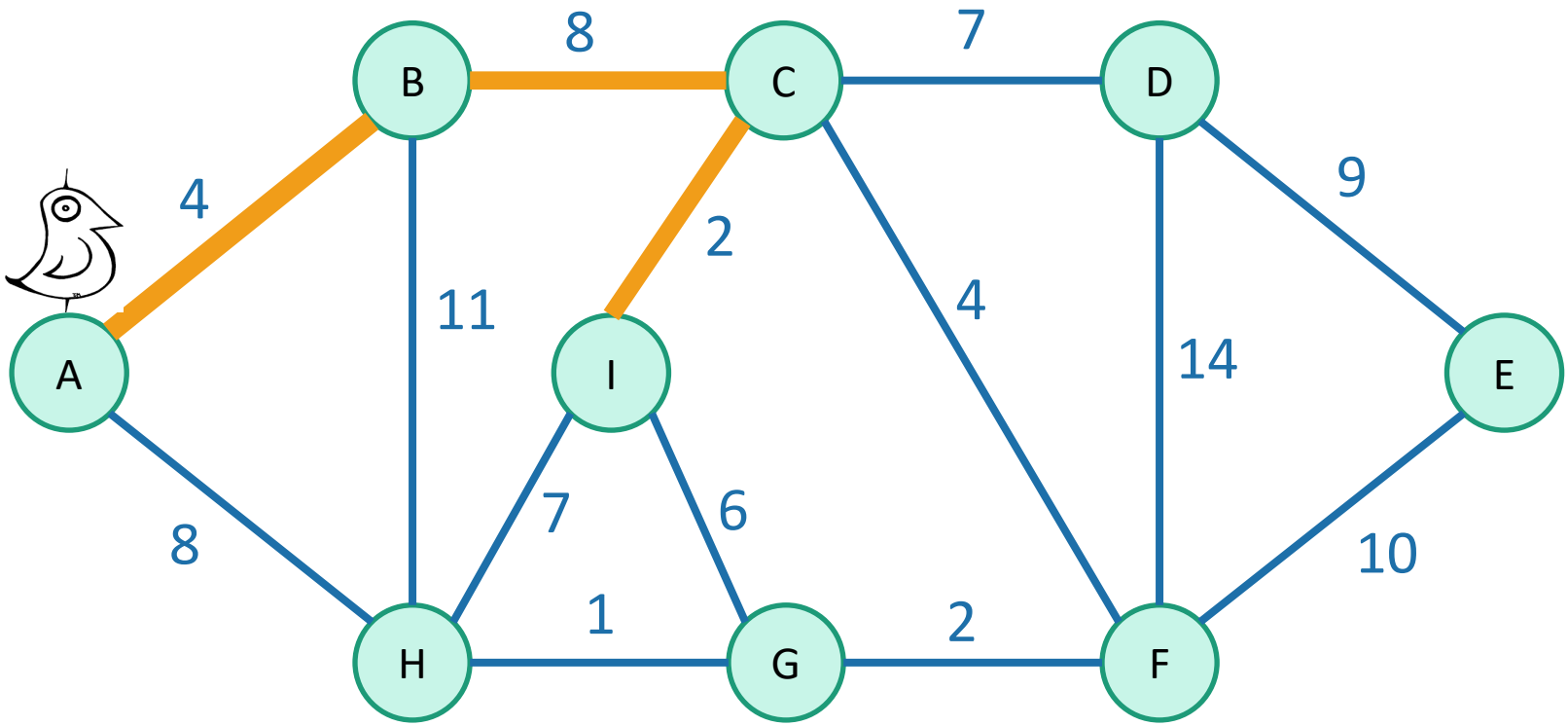
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



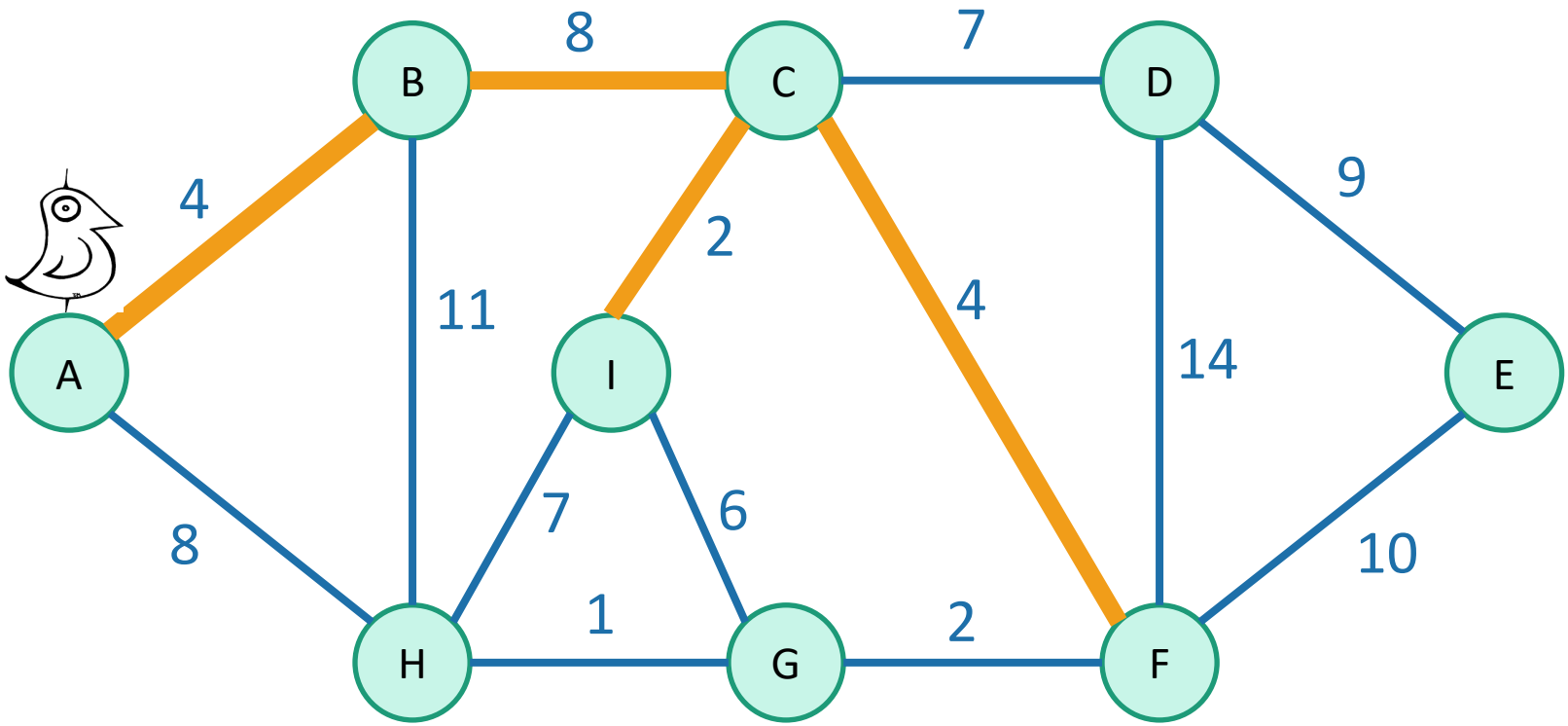
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



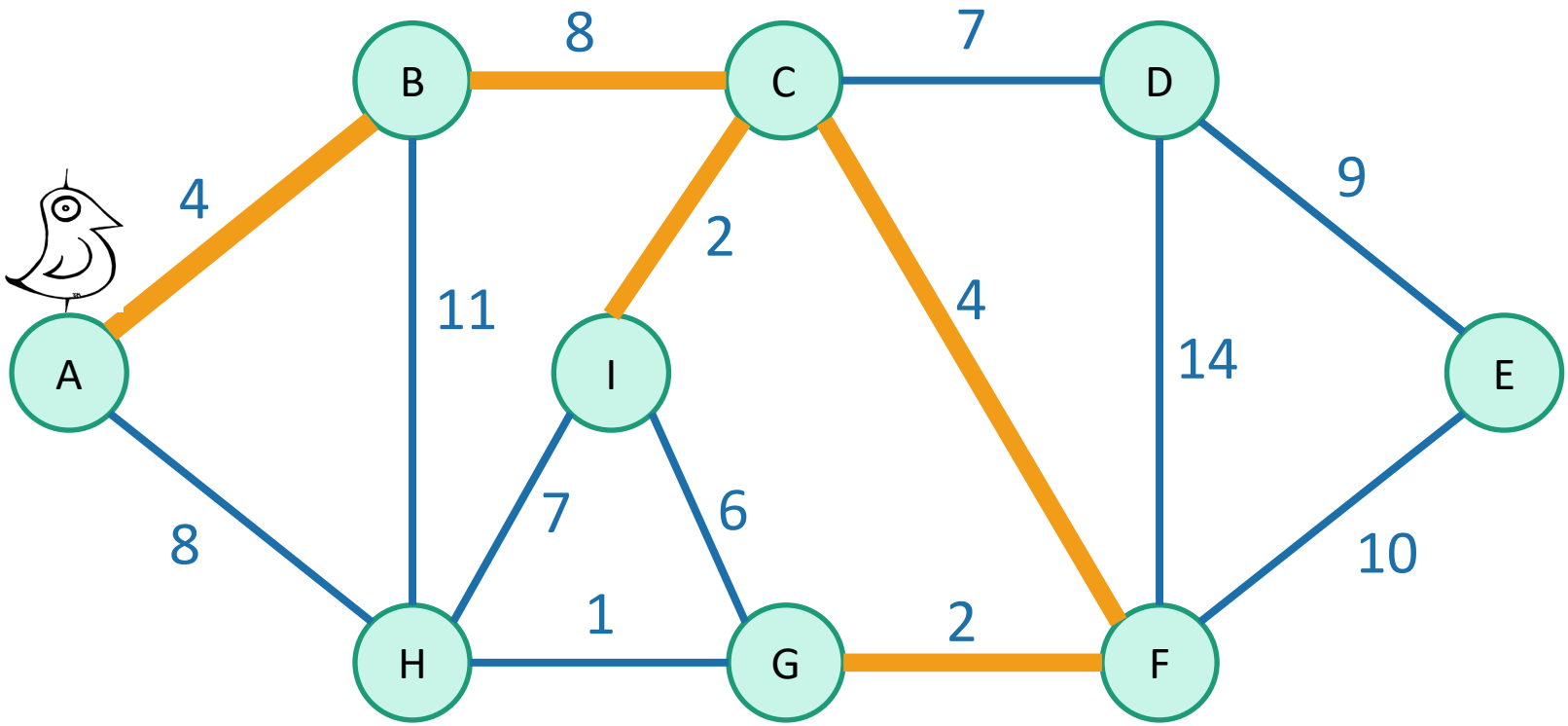
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



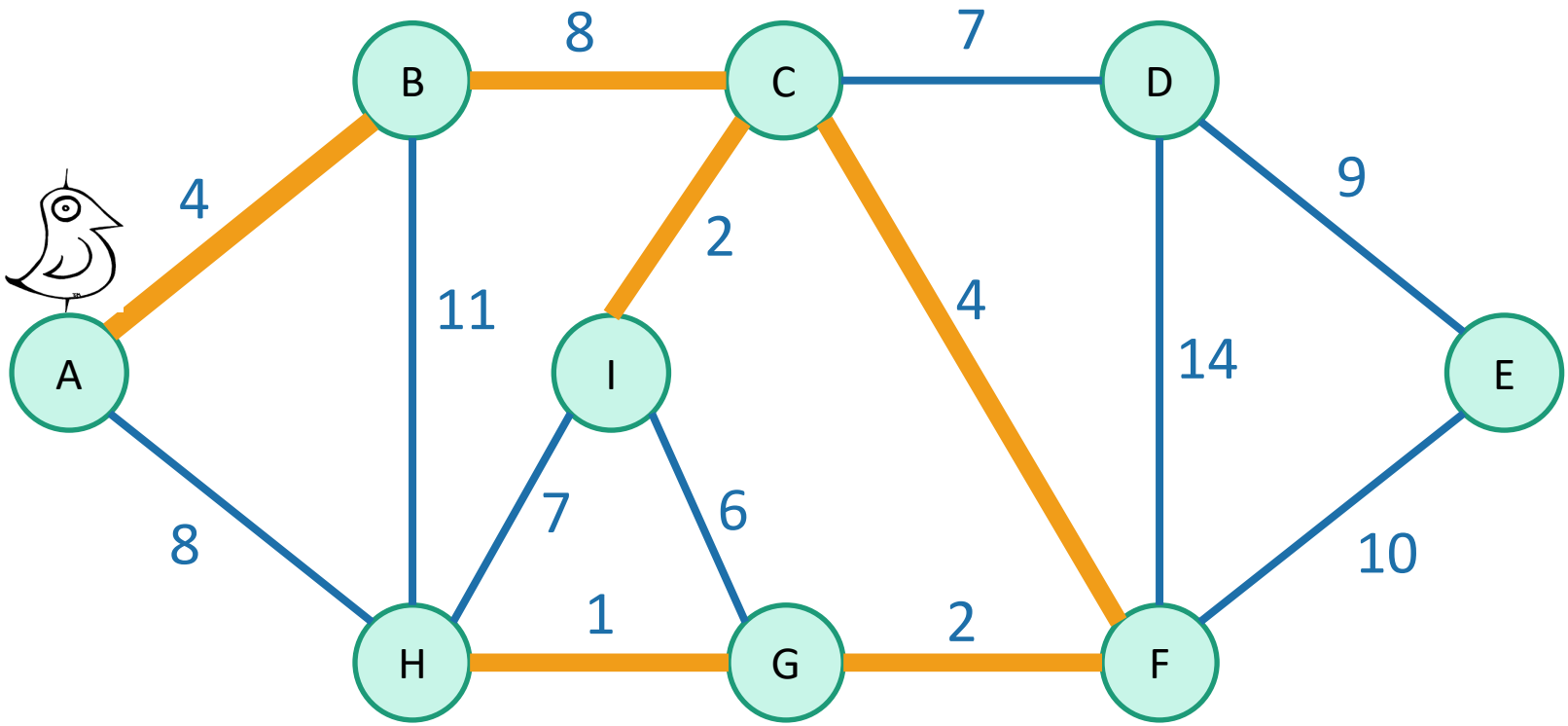
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



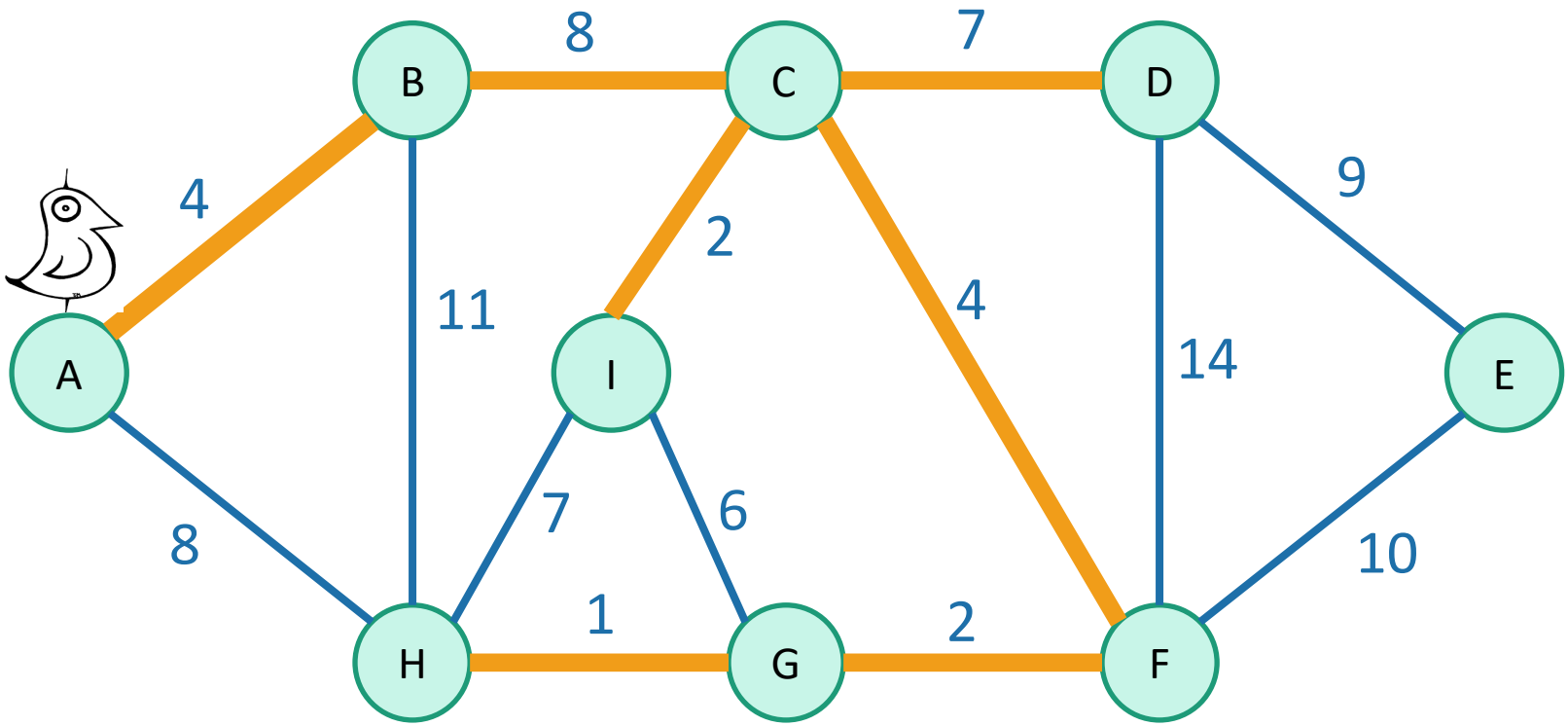
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



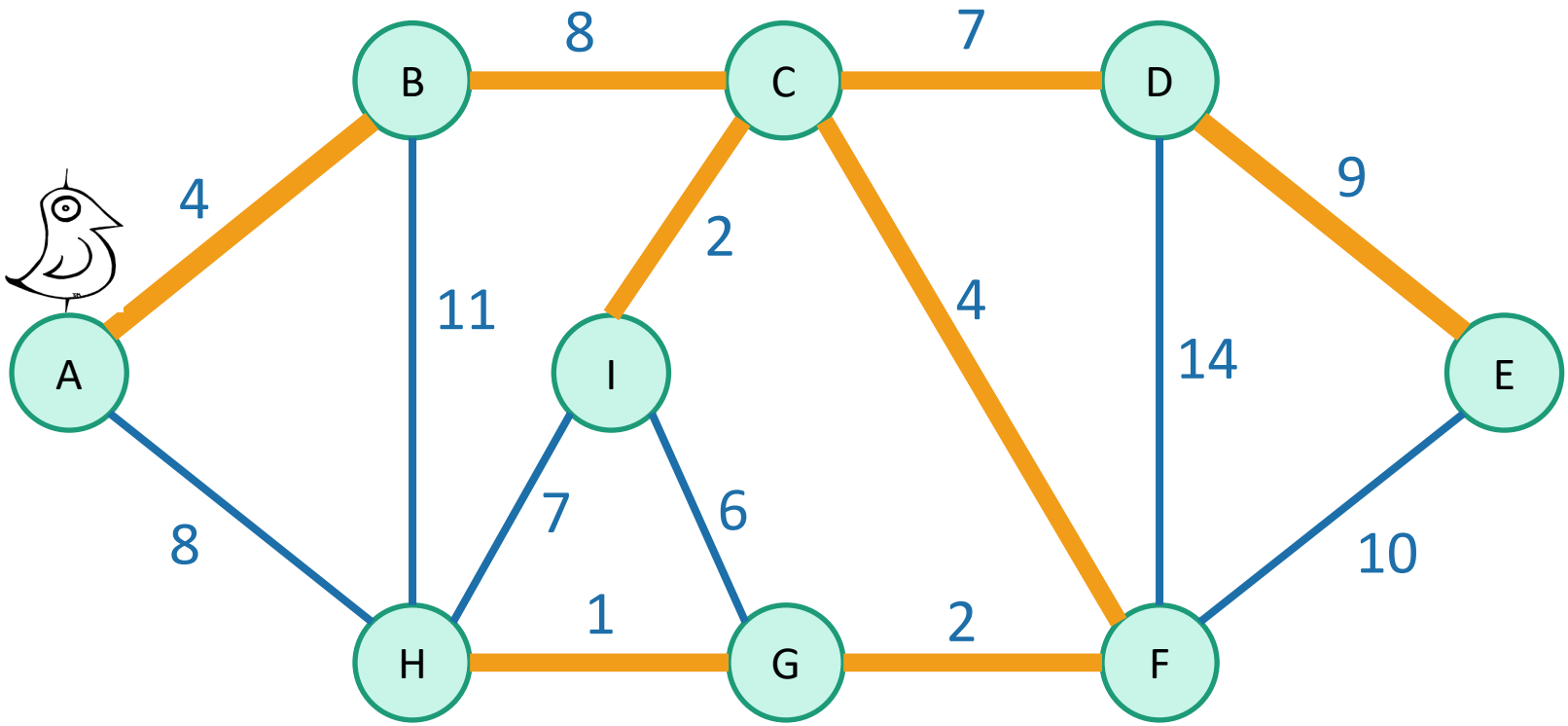
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.

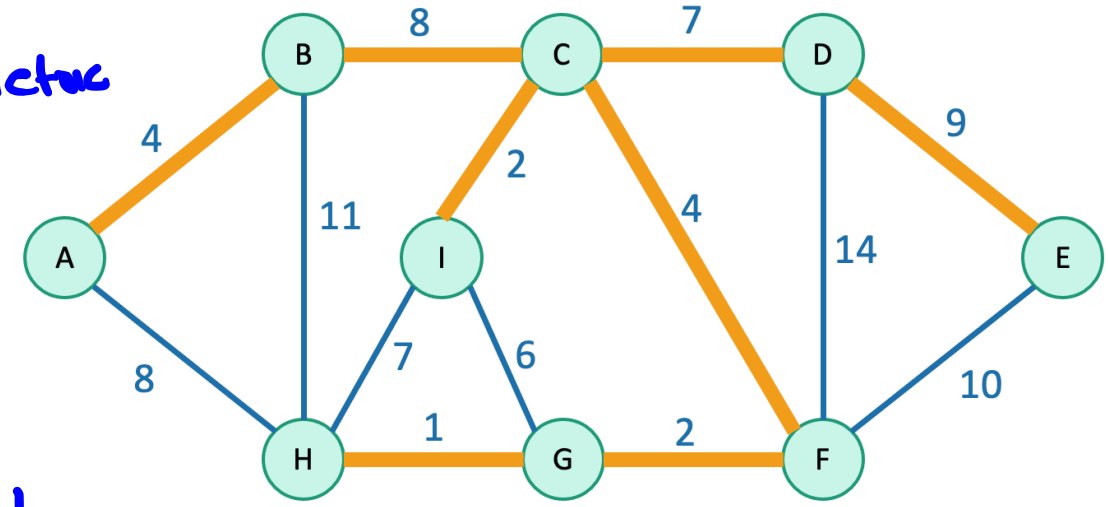


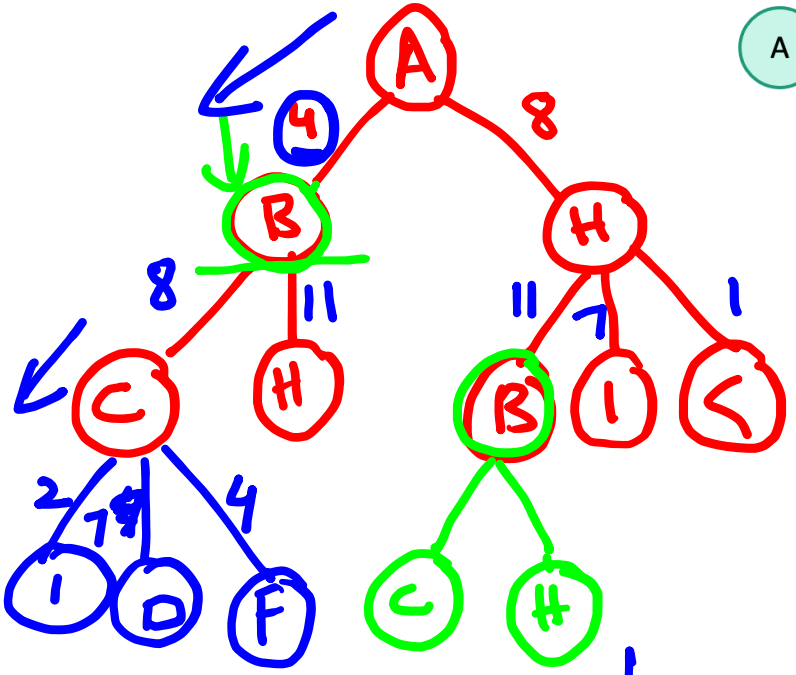
Idea 1

Start growing a tree, greedily add the shortest edge we can to grow the tree.



Prim's Algorithm





Prim

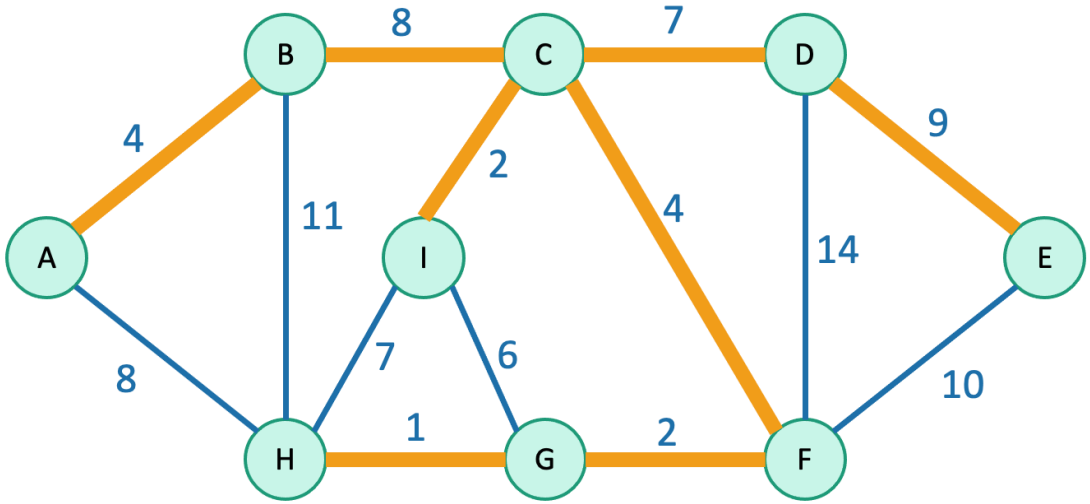
P
I
SP
I
SP

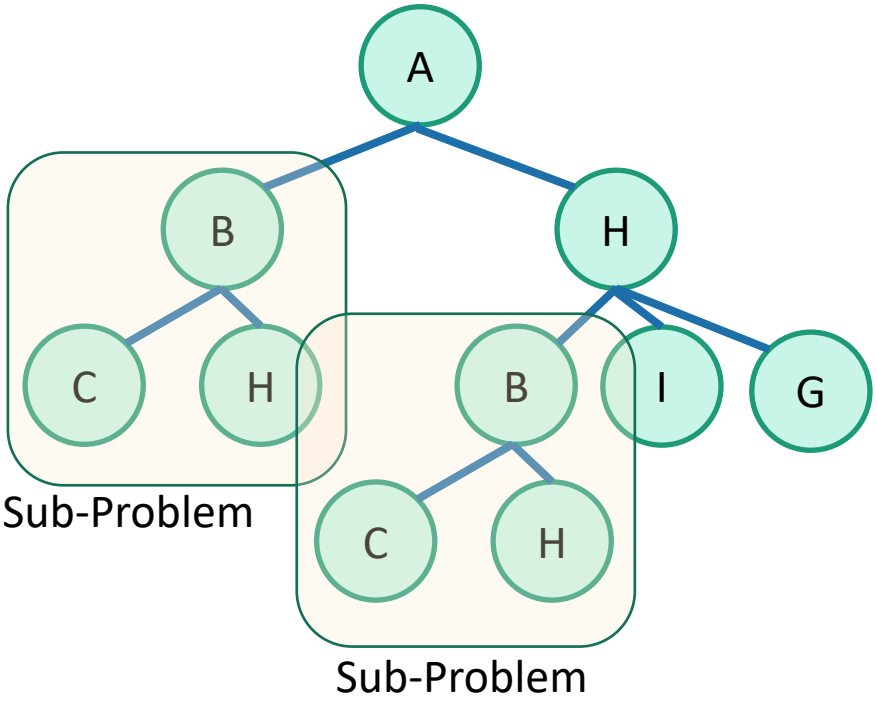
Cycle

ThinEdge

MST

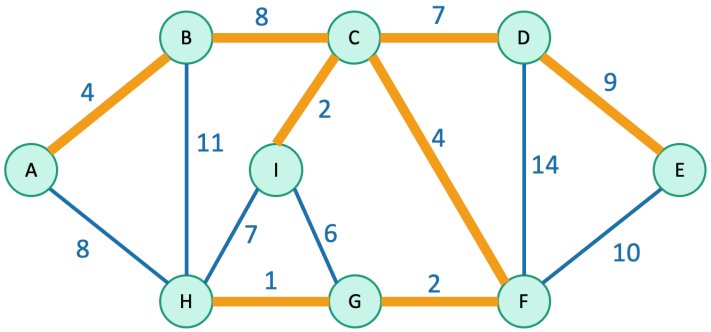
Sorted


A
B
C
D
E
F
G
H
I
4
8
7
9
11
2
4
14
8
7
6
10
1
2


A
B
H
C
H
B
I
G
C
H
Sub-Problem
Sub-Problem

We've discovered

Prim's algorithm!



- slowPrim(G = (V,E), starting vertex s):
  - Let (s,u) be the lightest edge coming out of s.
  - MST = { (s,u) }
  - verticesVisited = { s, u }
  - **while** |verticesVisited| < |V|:
    - find the lightest edge (x,v) in E so that:
      - x is in verticesVisited
      - v is not in verticesVisited
    - add (x,v) to MST
    - add v to verticesVisited
  - **return** MST

n iterations of this while loop.

Maybe take time m to go through all the edges and find the lightest.

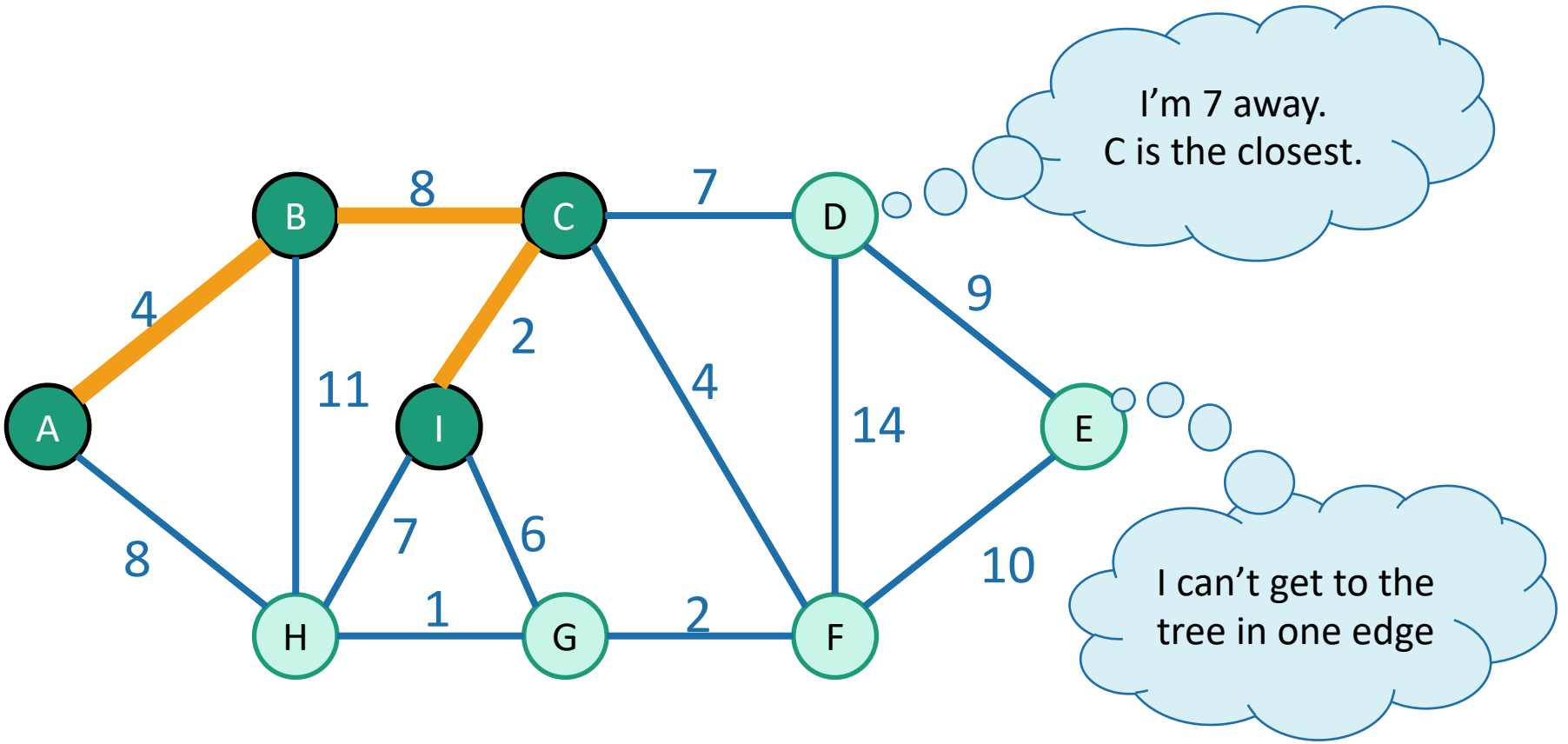
Naively, the running time is O(nm):

- For each of n-1 iterations of the while loop:
  - Maybe go through all the edges.

How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.

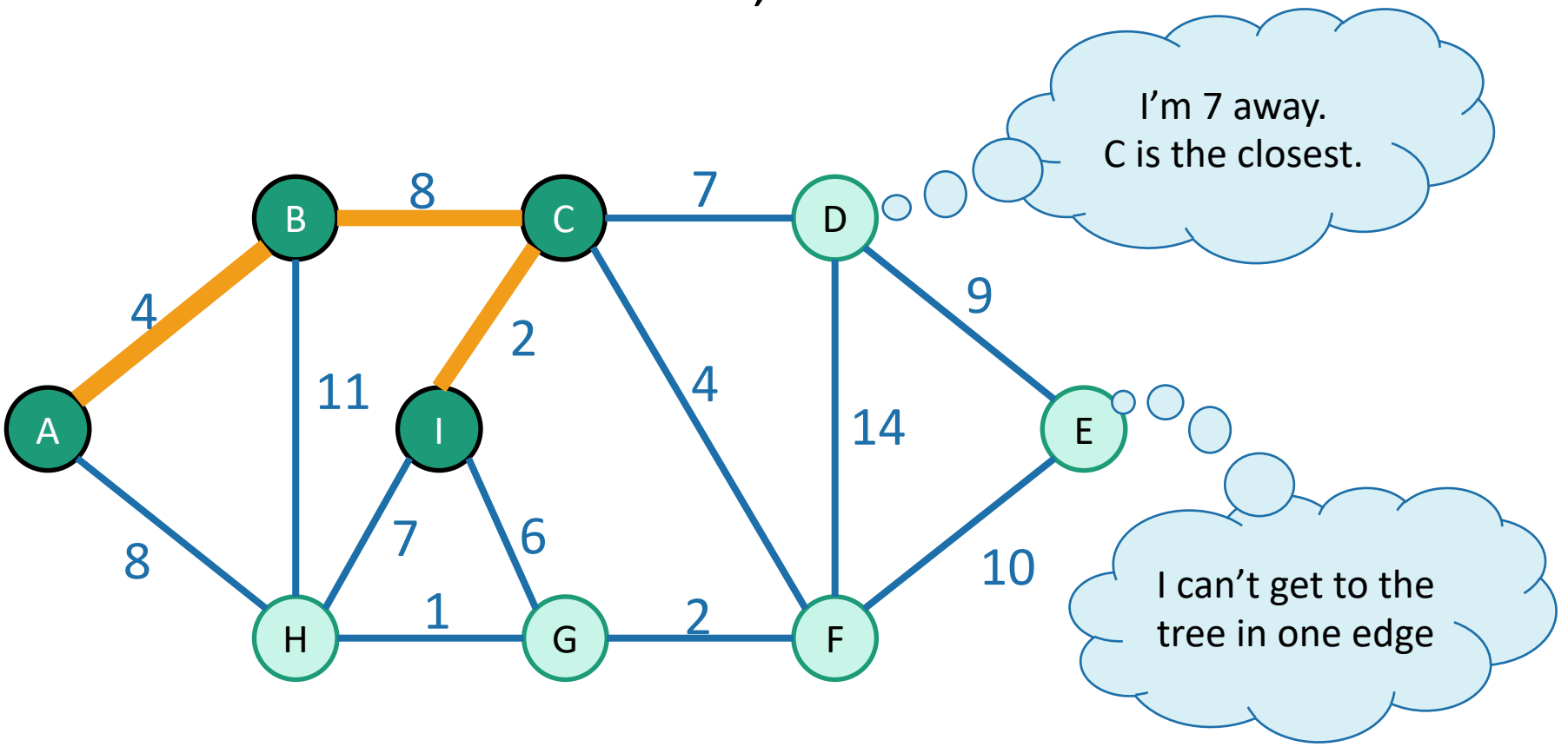
if you can get there in one edge.



How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.

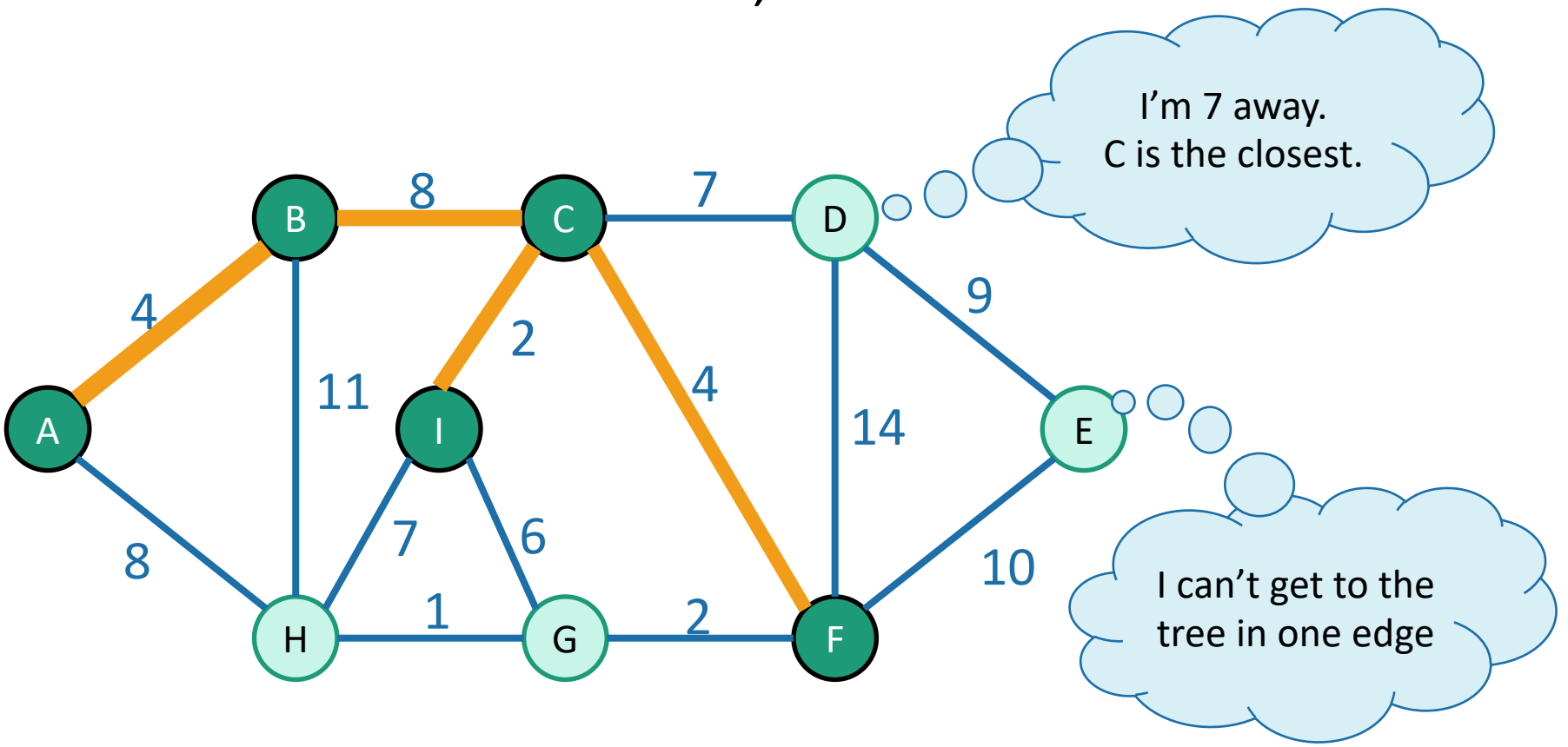
if you can get there in one edge.



How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.
- Choose the closest vertex, add it.

if you can get there in one edge.

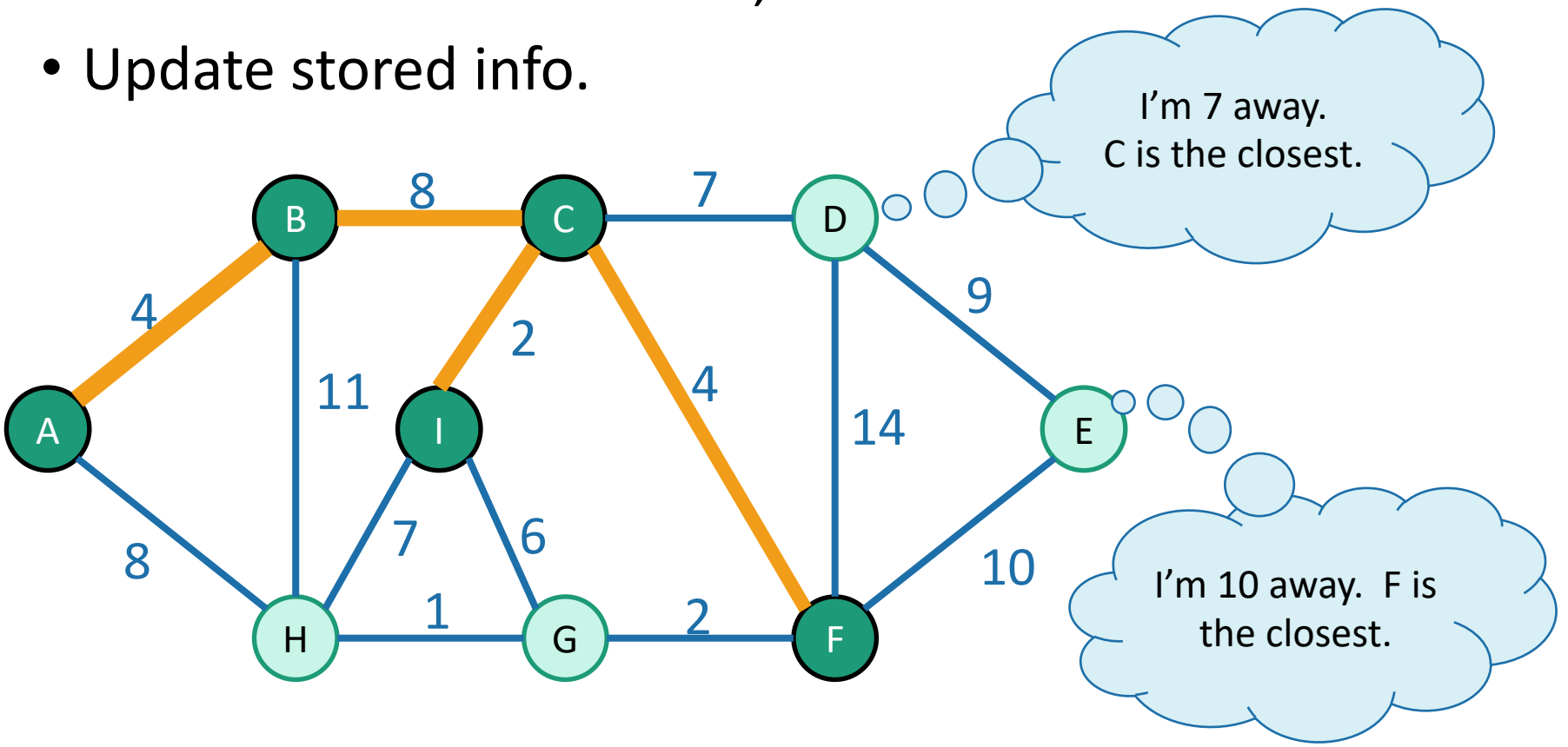


How do we actually implement this?

- Each vertex keeps:
  - the **distance** from itself to the **growing spanning tree**
  - **how to get there**.

if you can get there in one edge.

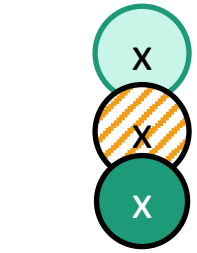
- Choose the closest vertex, add it.
- Update stored info.



Efficient implementation

Every vertex has a key and a parent

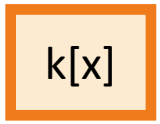
Until all the vertices are **reached:**



Can't reach x yet

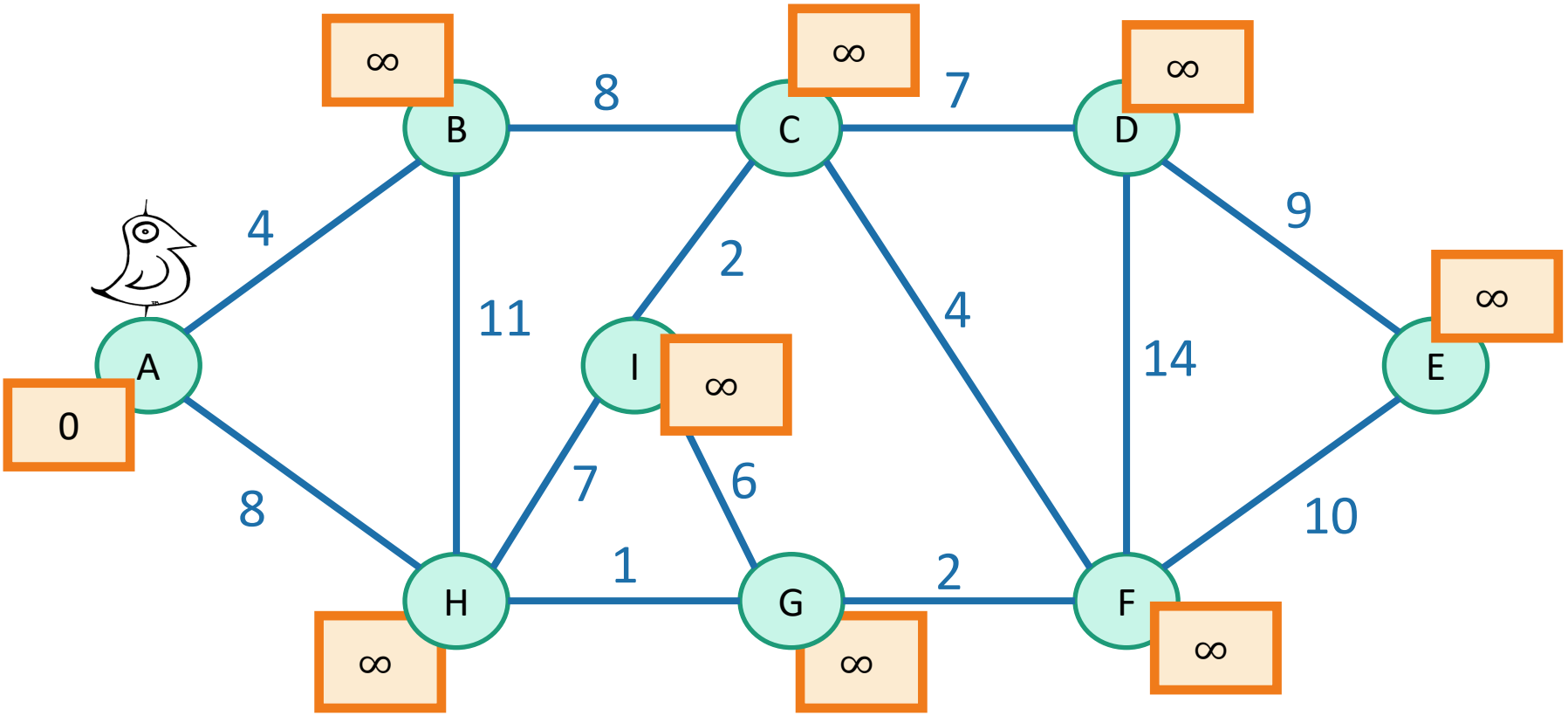
x is "active"

Can reach x



k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

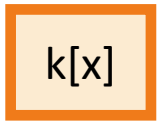
Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.

Can't reach x yet

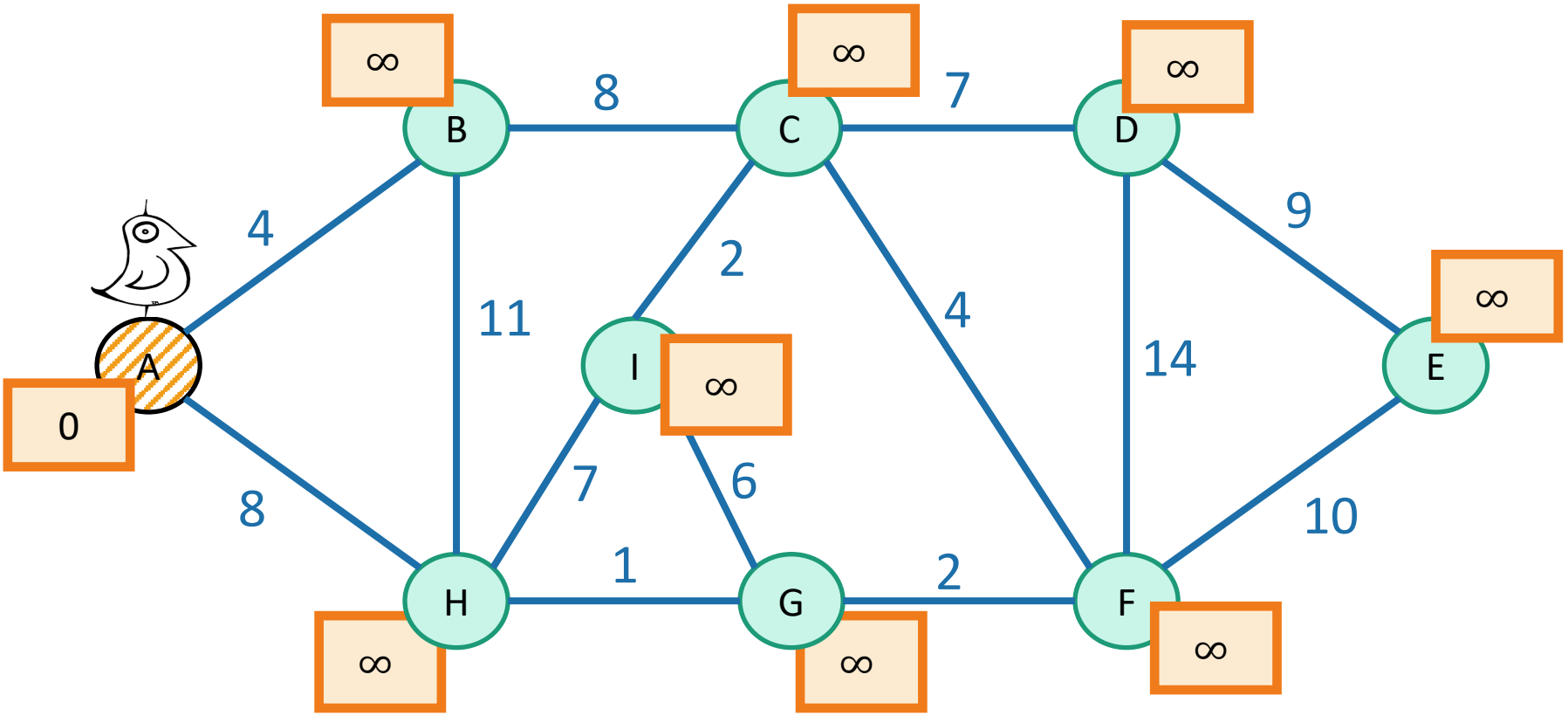
x is "active"

Can reach x



k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

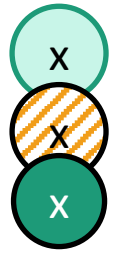


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

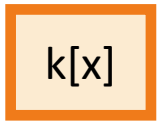
- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u



Can't reach x yet

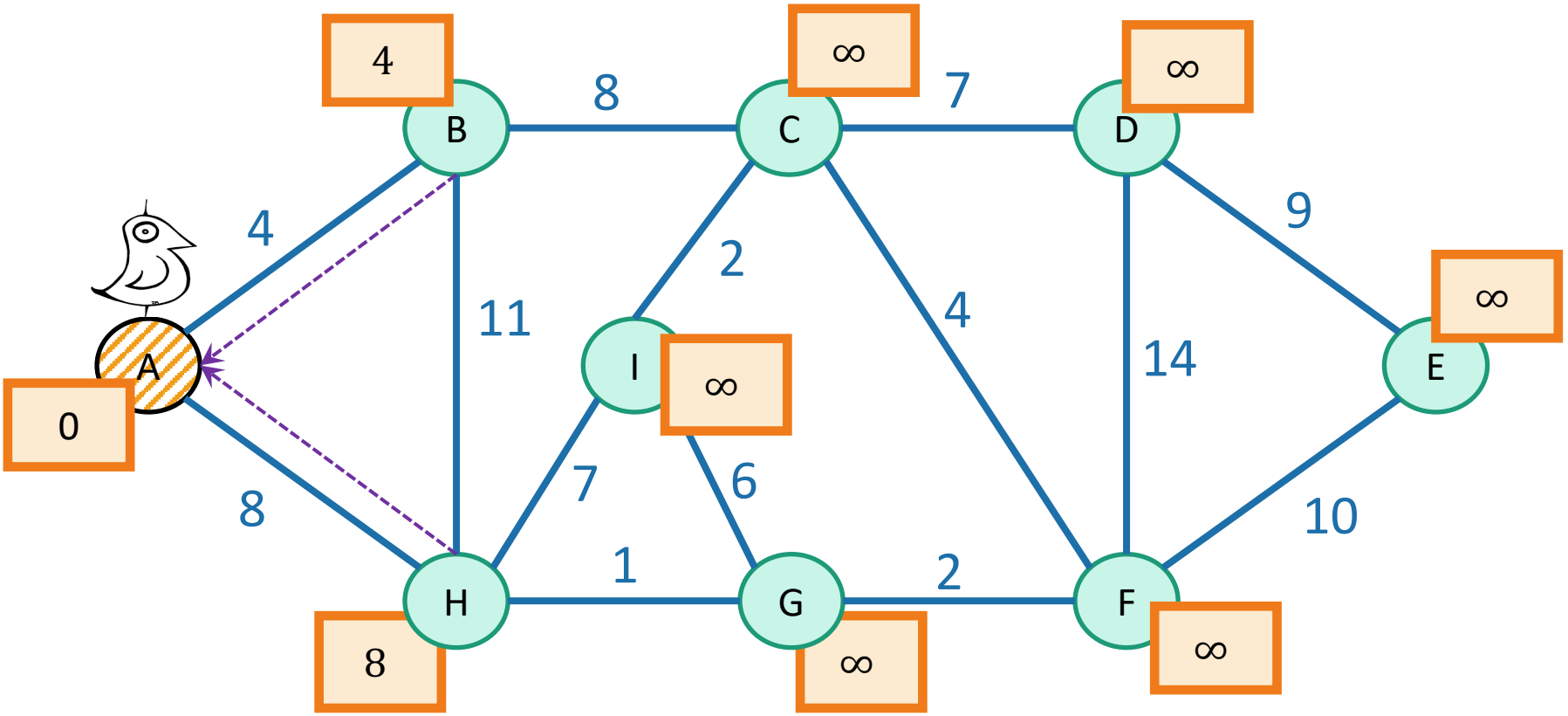
x is "active"

Can reach x



k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

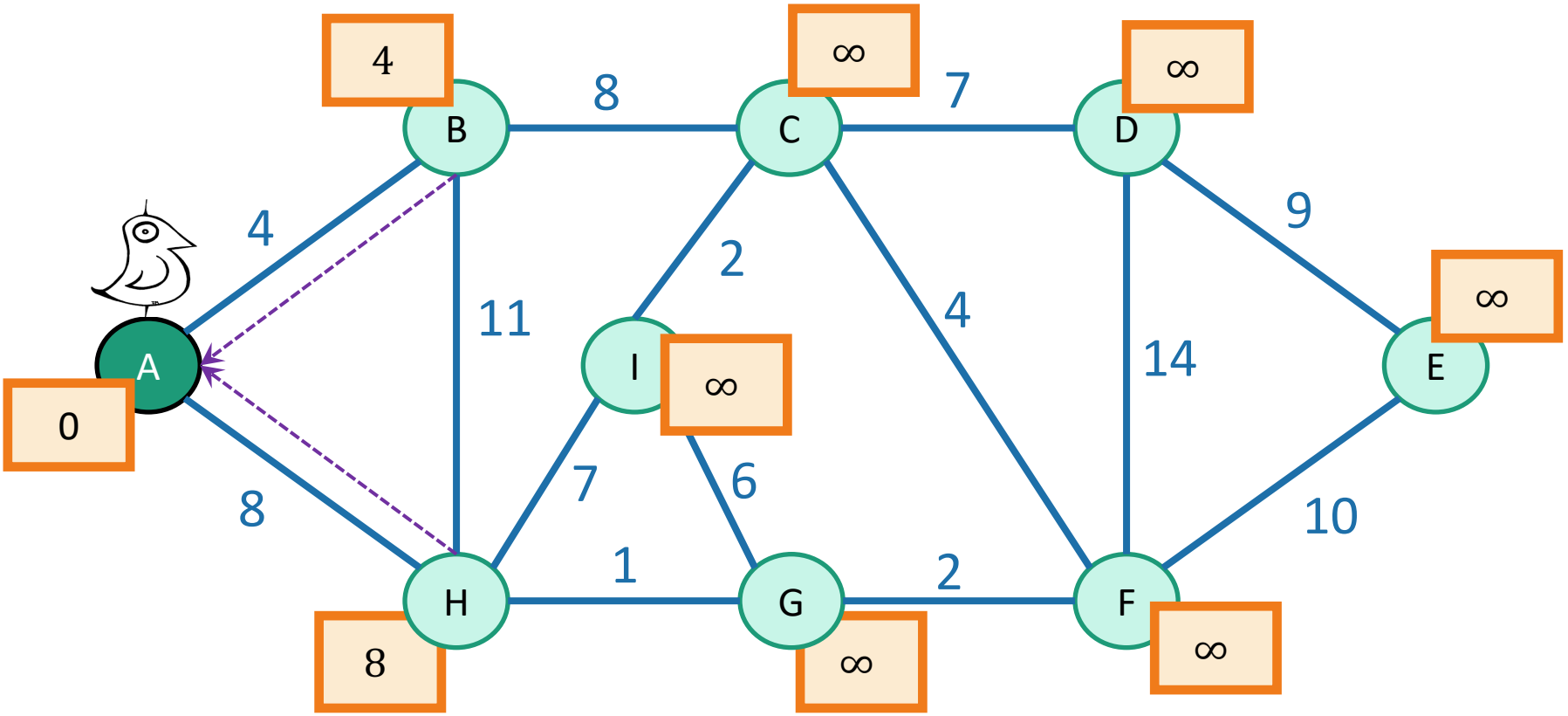
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

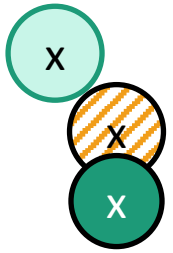


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

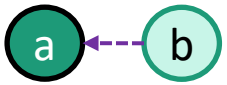


Can't reach x yet

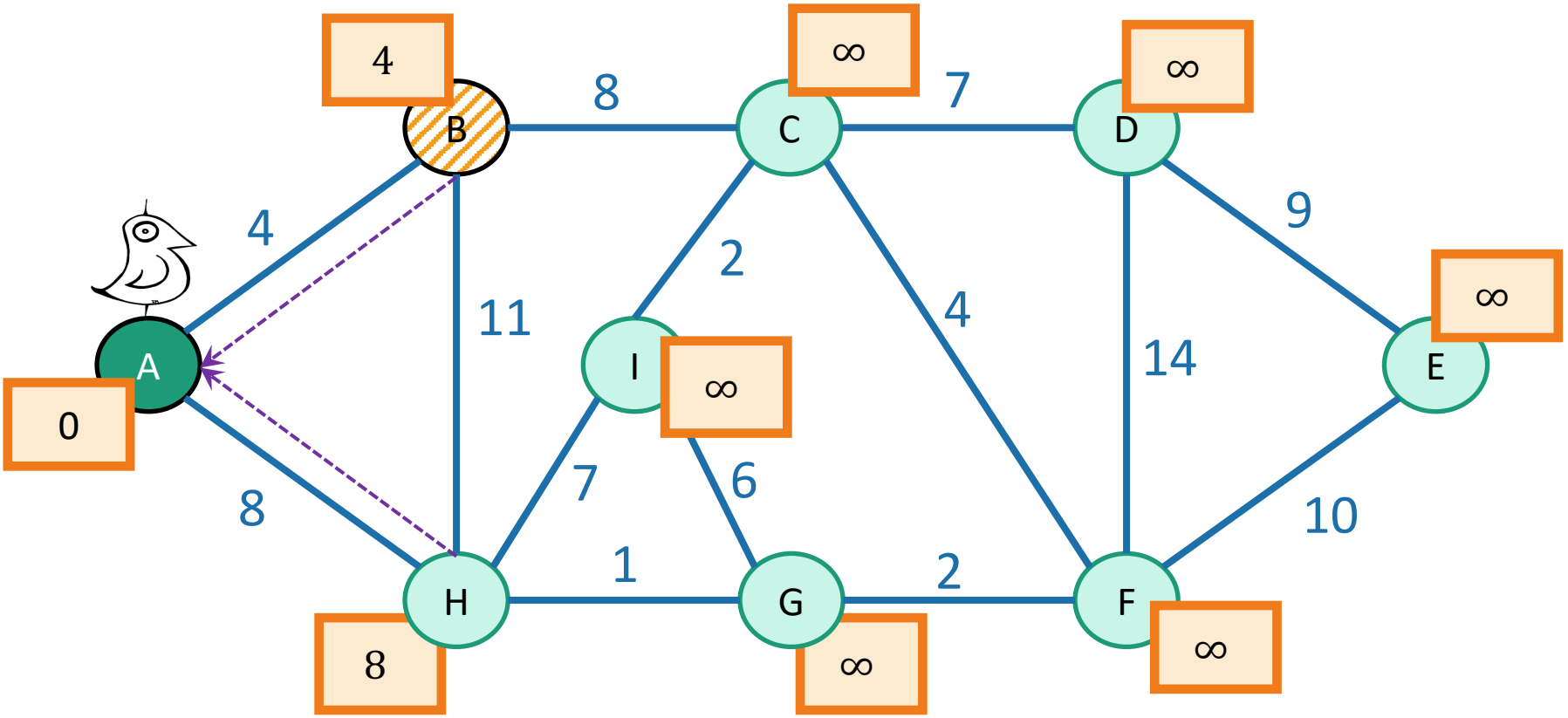
x is "active"

Can reach x

k[x] is the distance of
from the growing tre



p[b] = a, meaning th
a was the vertex tha
k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

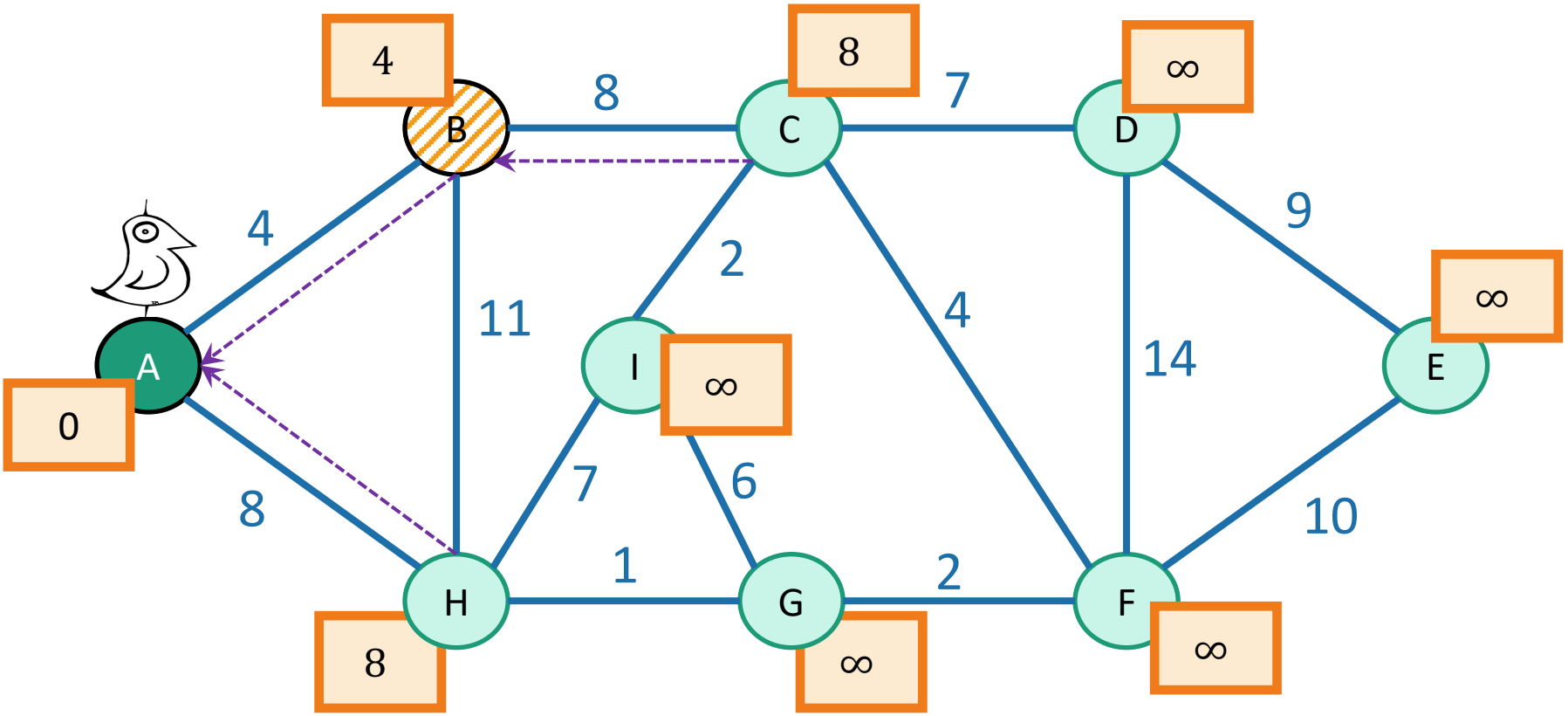
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

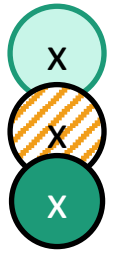


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



Can't reach x yet

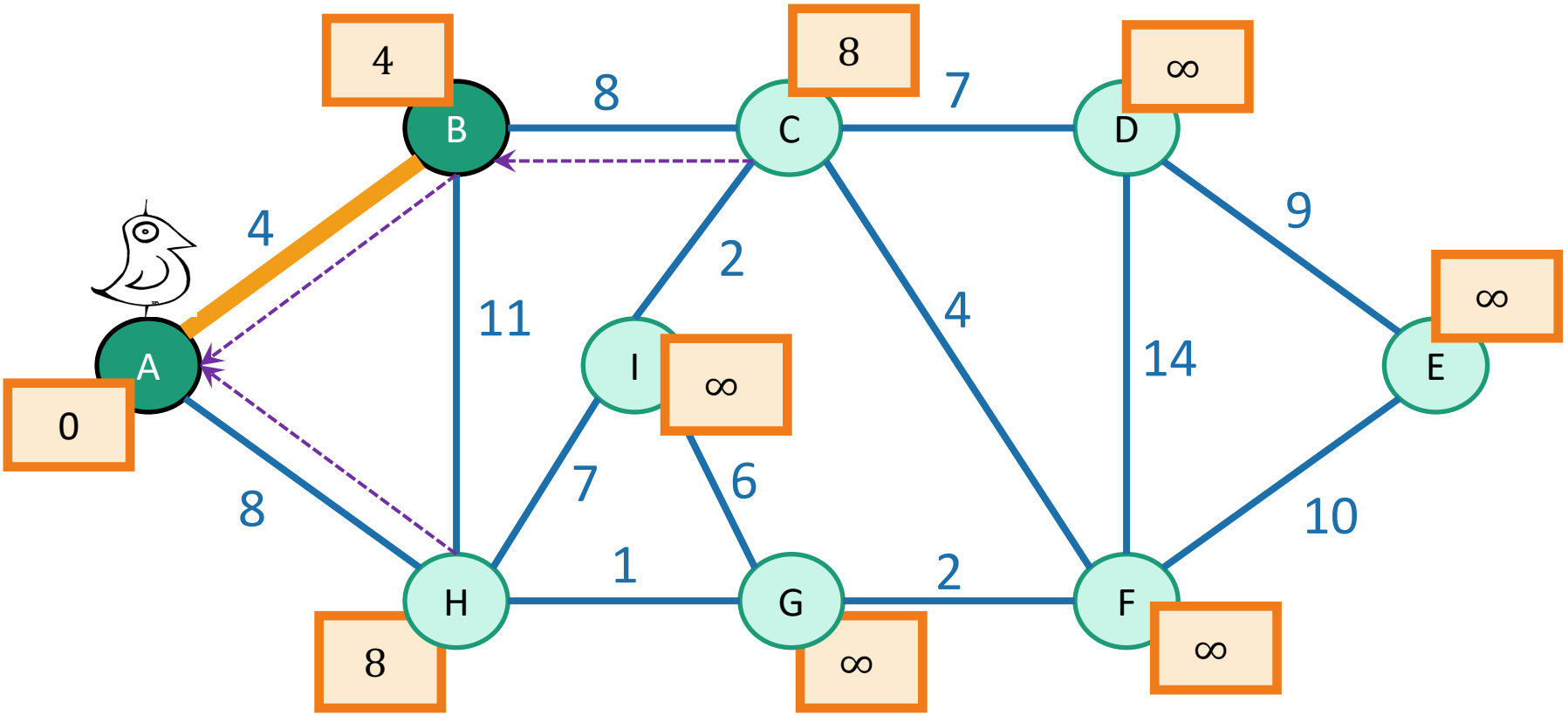
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

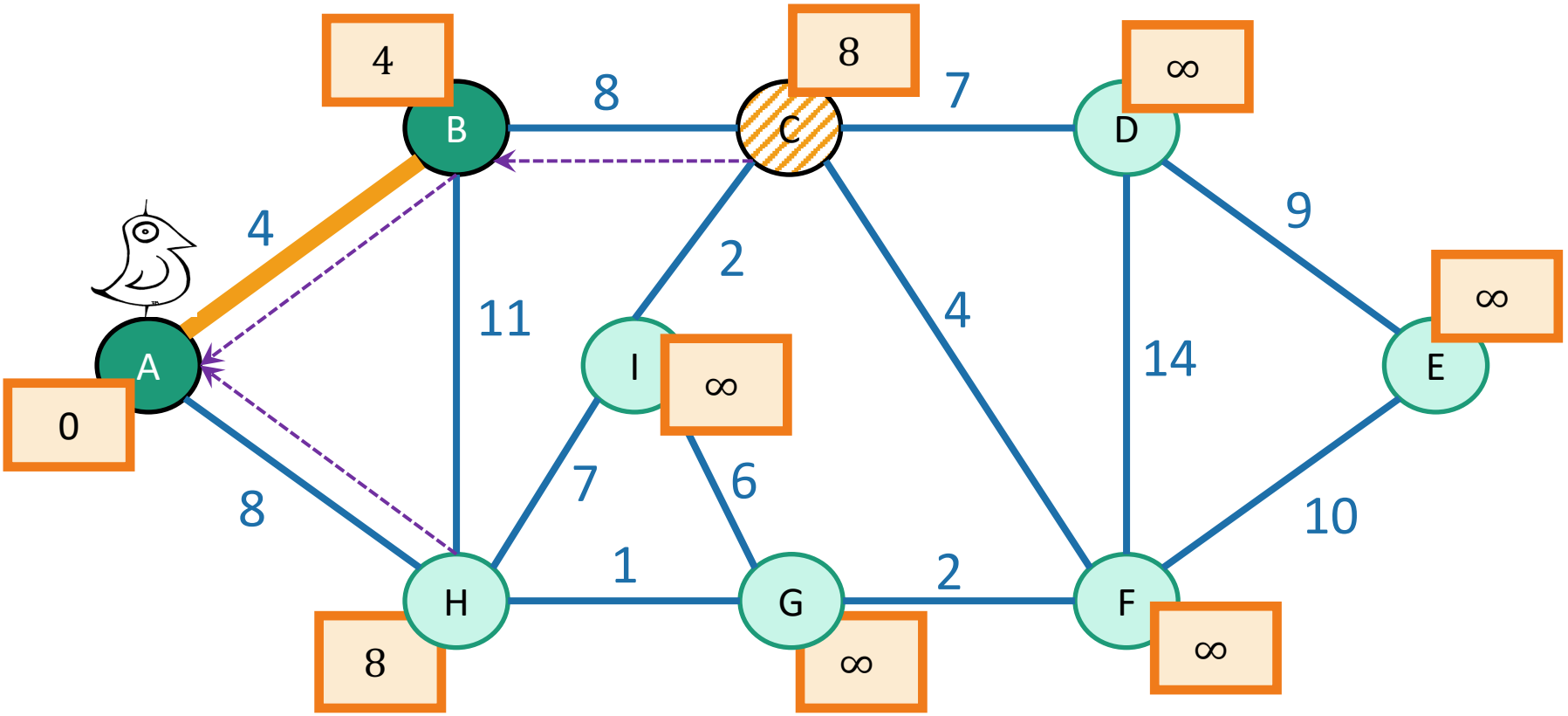
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

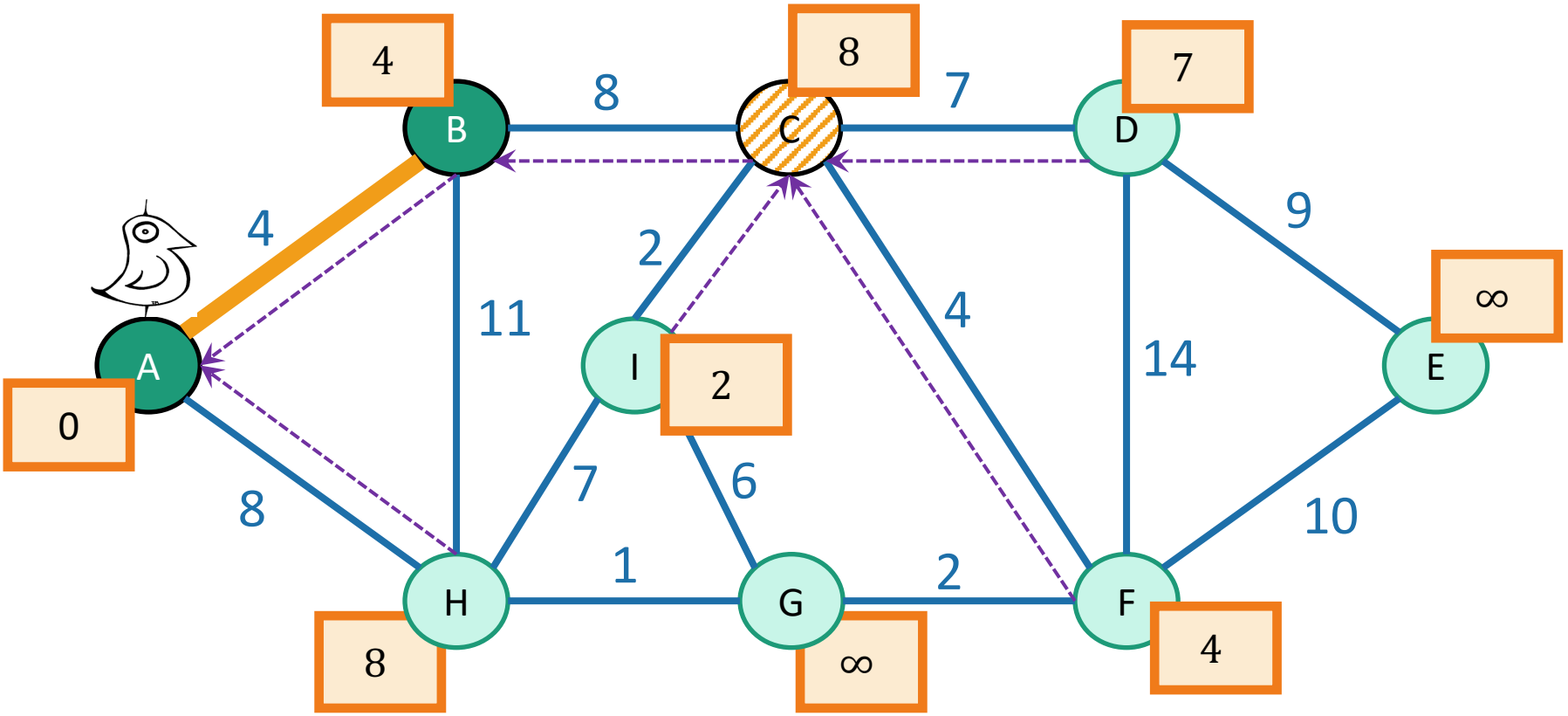
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

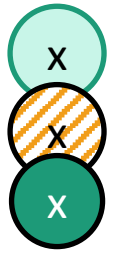


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



Can't reach x yet

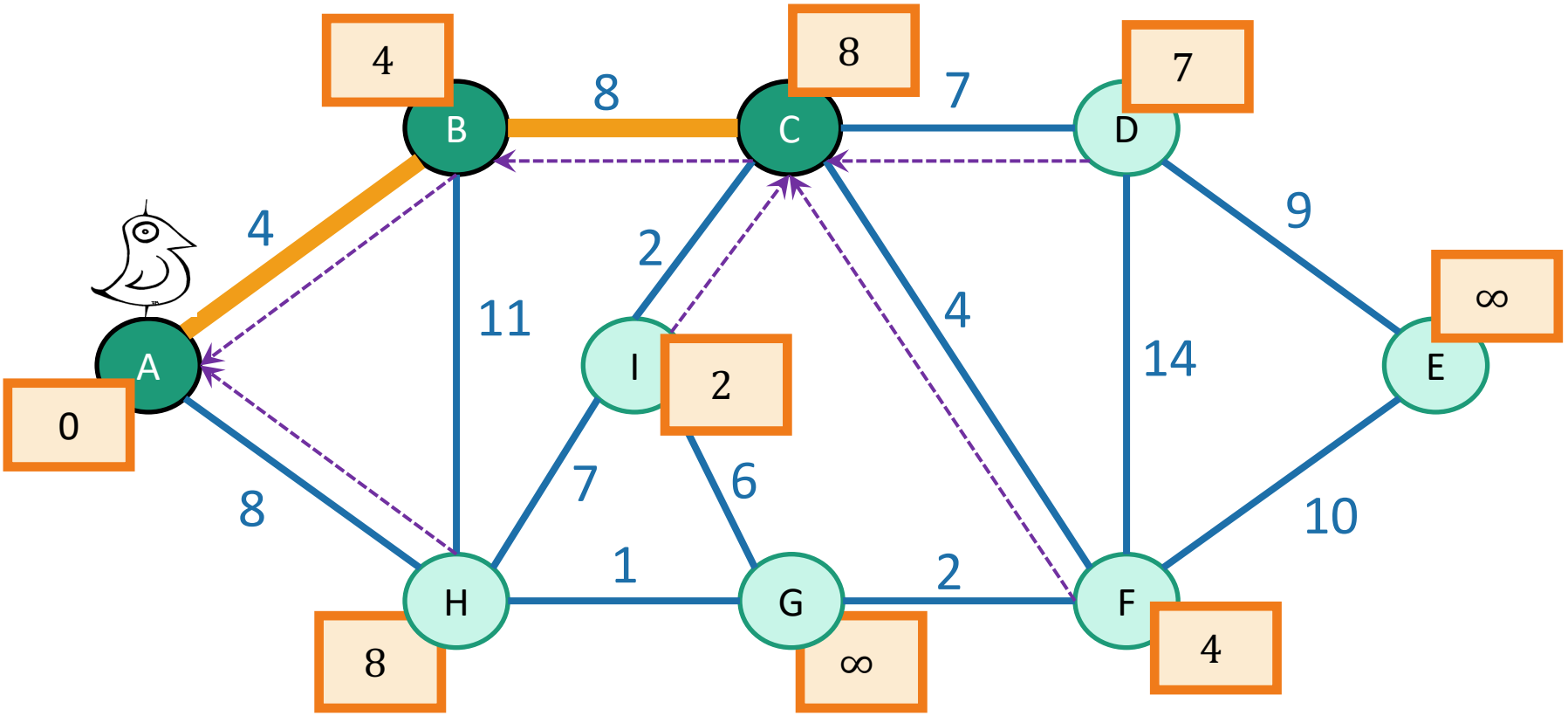
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

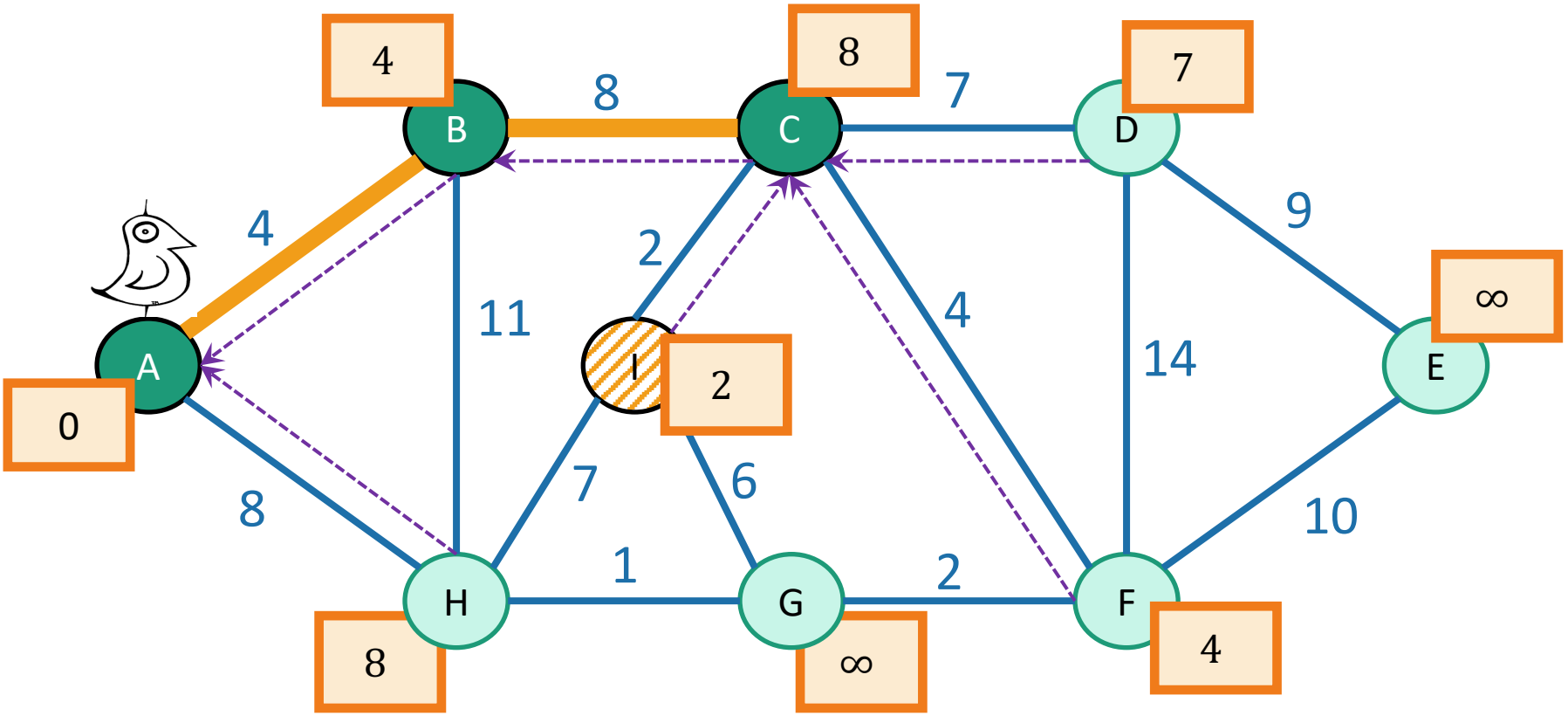
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

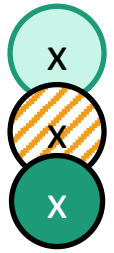


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



Can't reach x yet

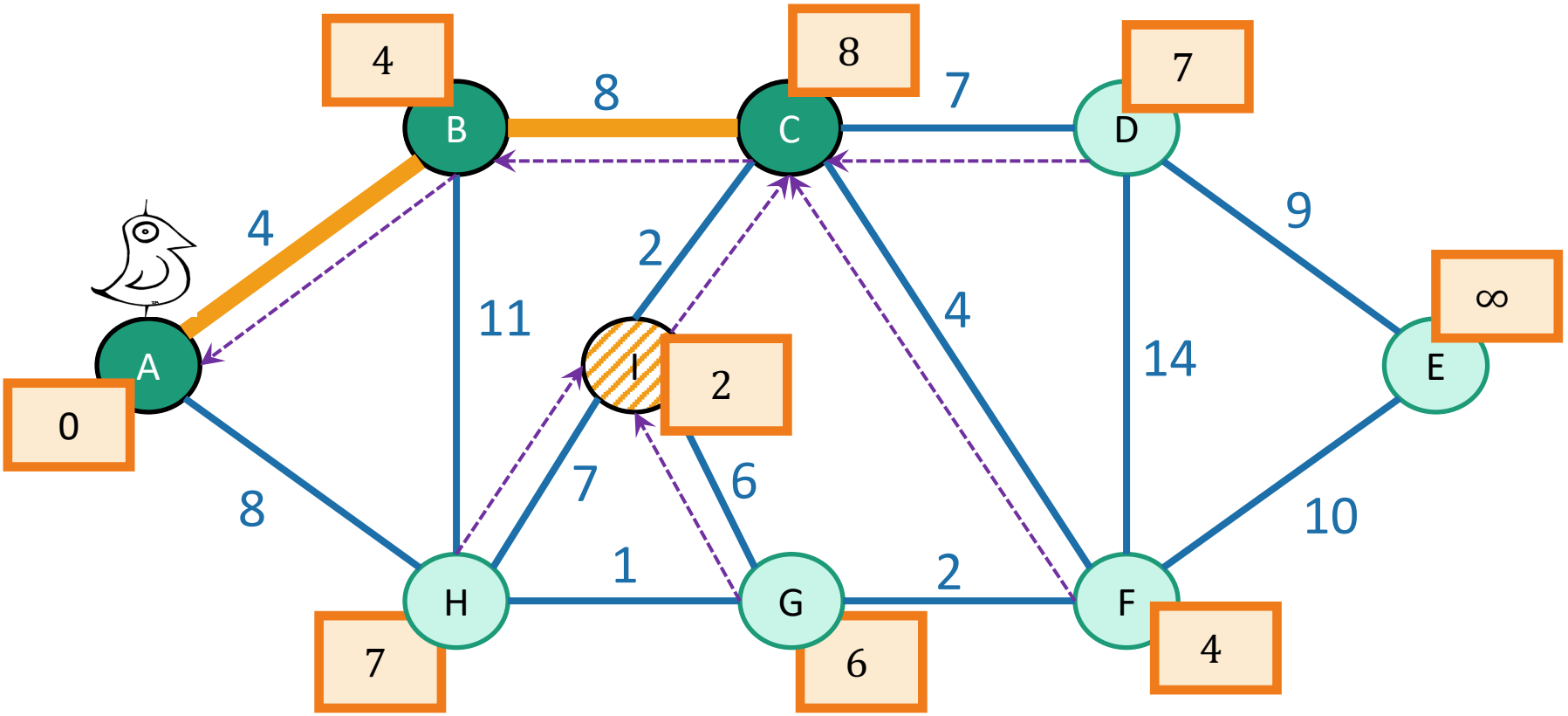
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

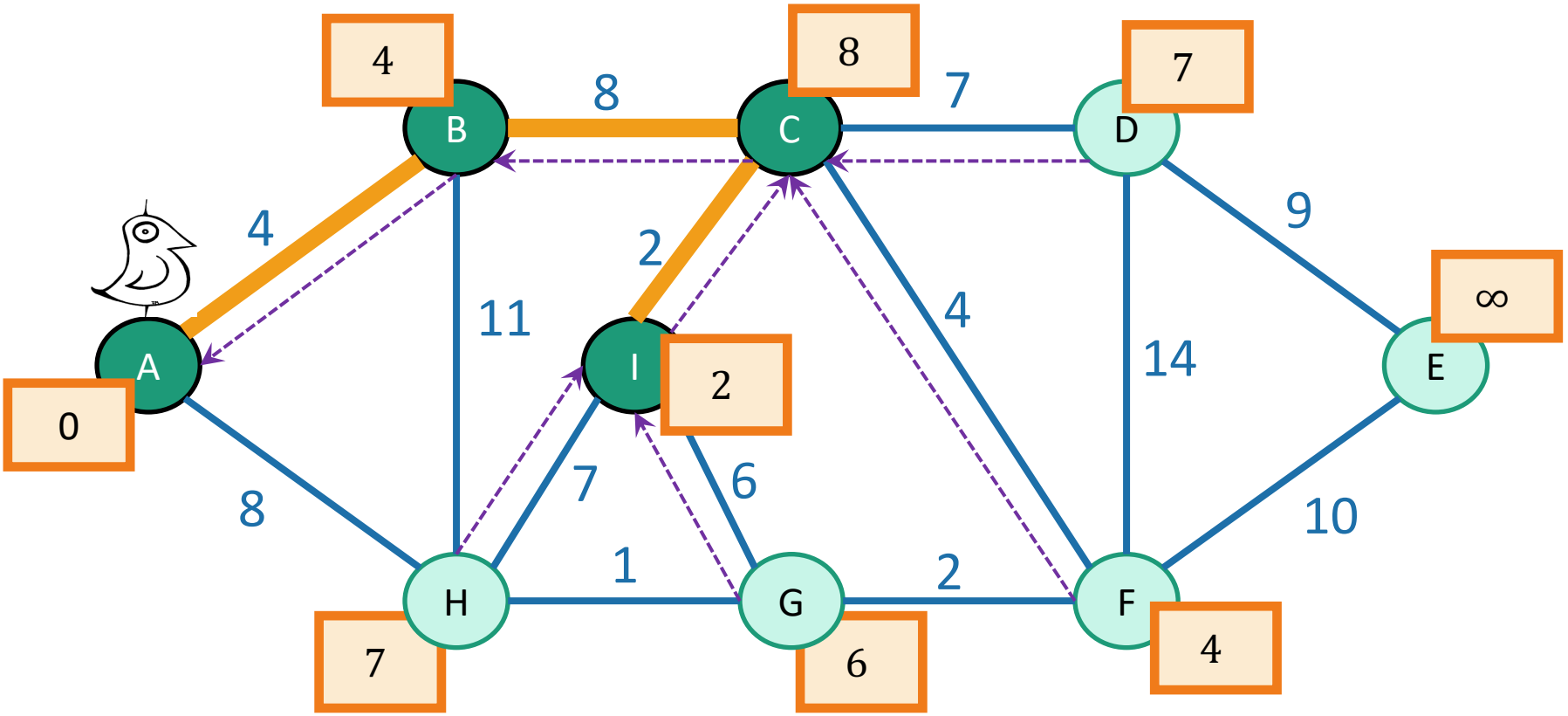
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

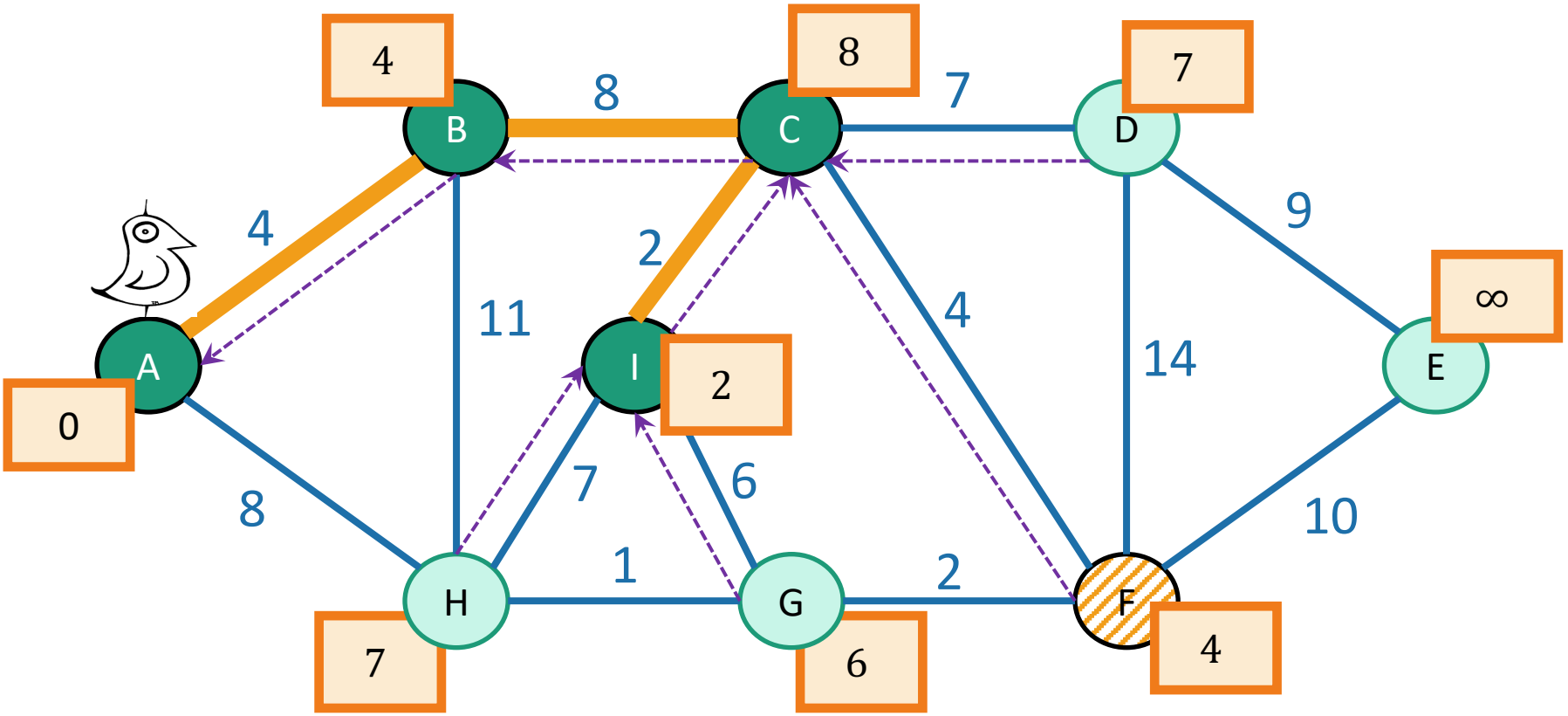
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.

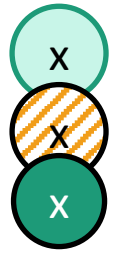


Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**



Can't reach x yet

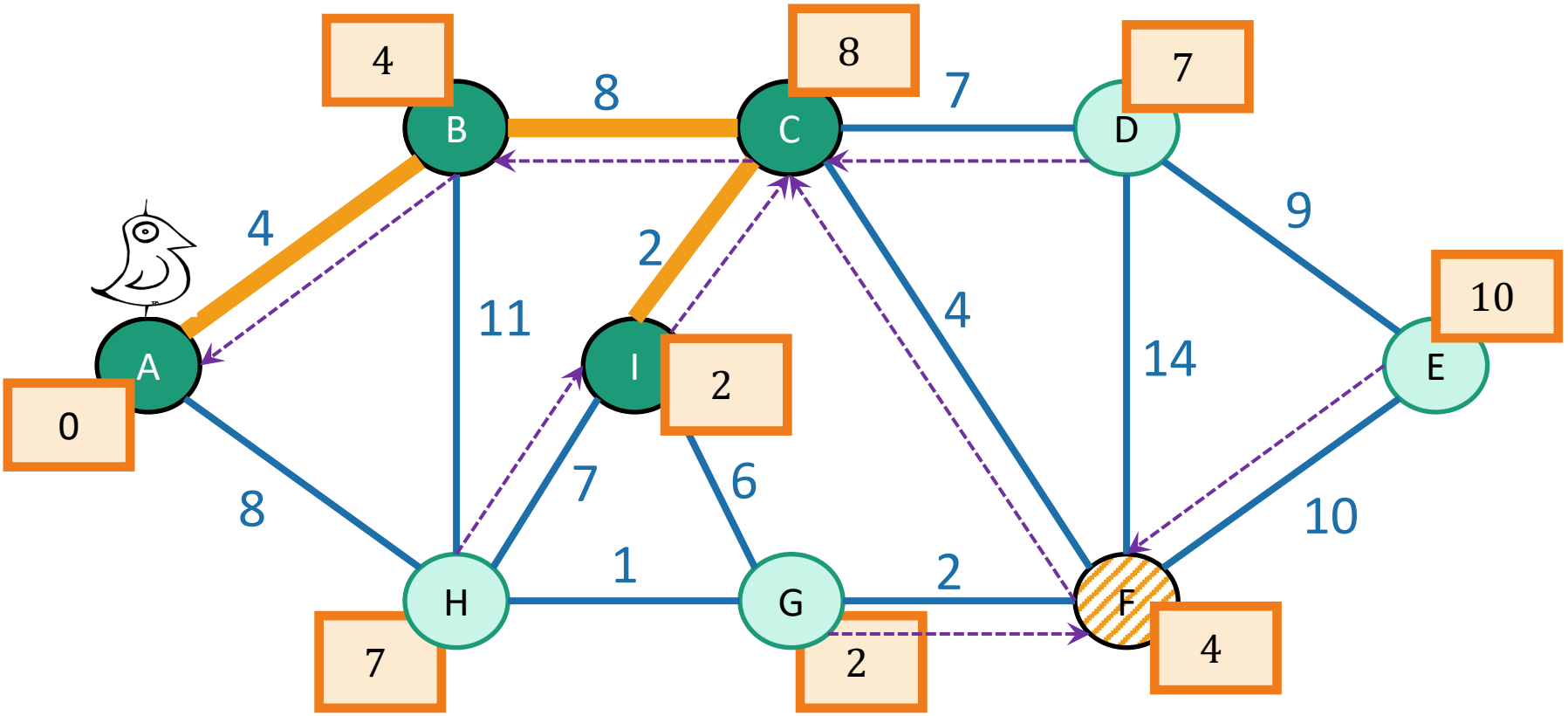
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

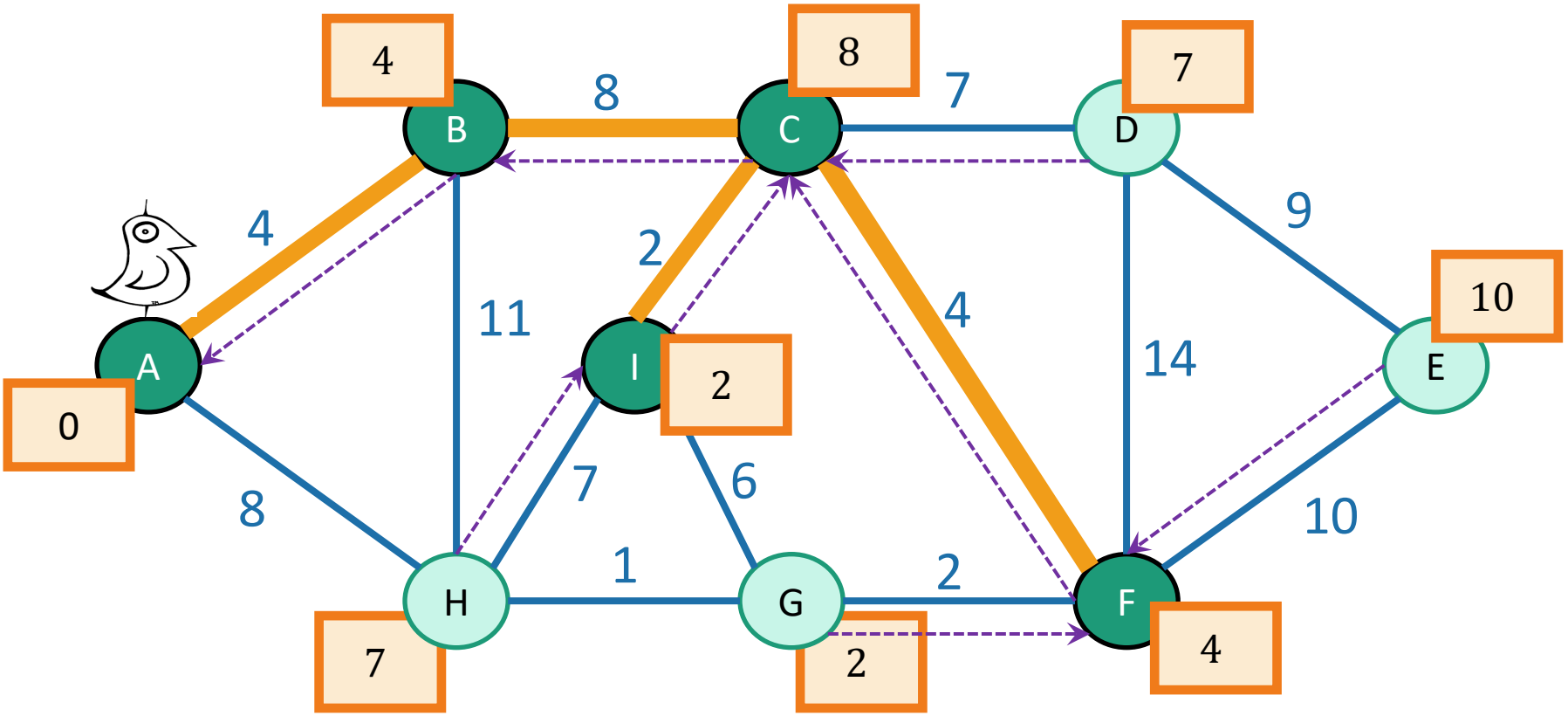
x is "active"

Can reach x

k[x]

k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



Efficient implementation

Every vertex has a key and a parent

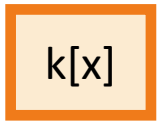
Until all the vertices are **reached:**

- Activate the **unreached** vertex u with the smallest key.
- **for each** of u's neighbors v:
  - k[v] = min(k[v], weight(u,v))
  - if k[v] updated, p[v] = u
- Mark u as **reached,** and **add (p[u],u) to MST.**

Can't reach x yet

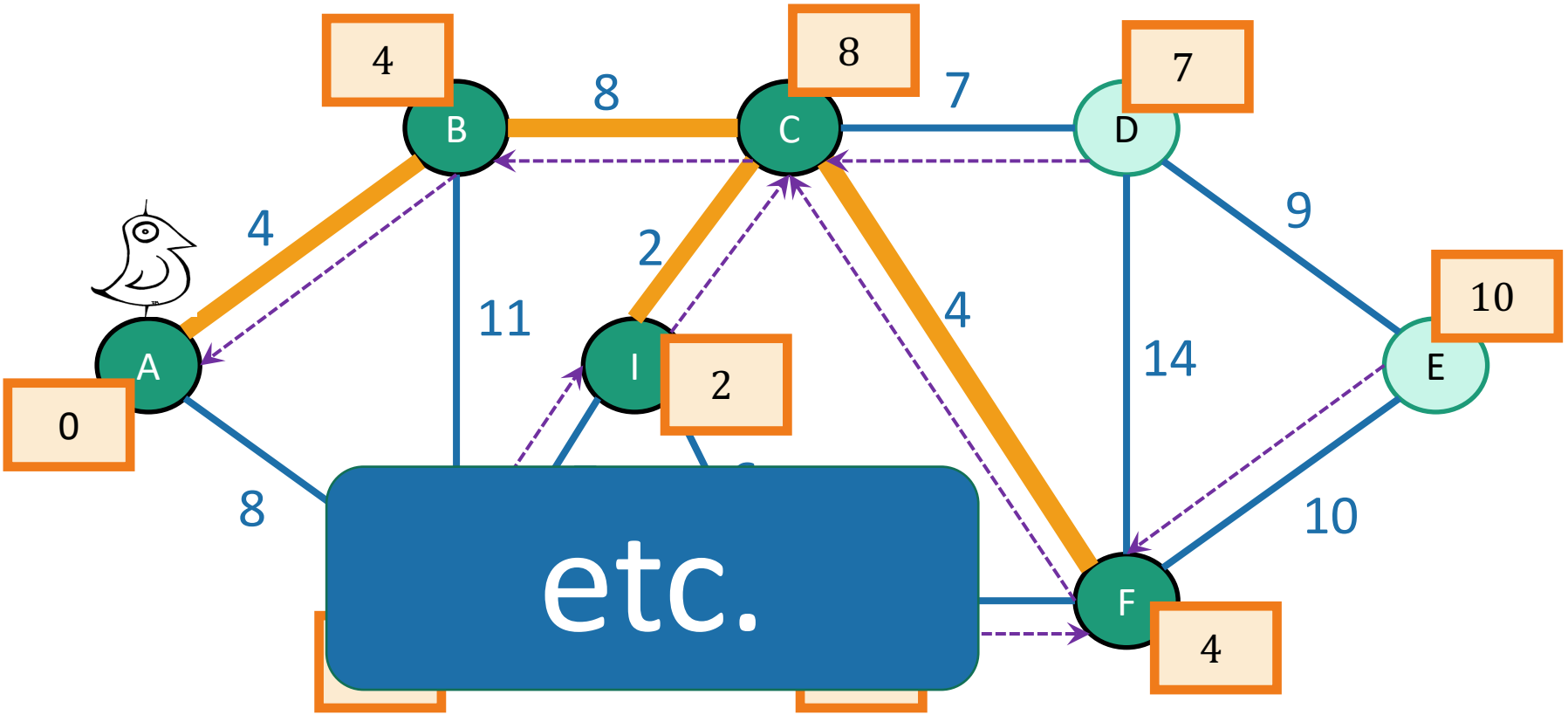
x is "active"

Can reach x



k[x] is the distance of x from the growing tree

p[b] = a, meaning that a was the vertex that k[b] comes from.



This should look pretty familiar

- Very similar to Dijkstra's algorithm!

- **Differences:**
  1. Keep track of p[v] in order to return a tree at the end
     - But Dijkstra's can do that too, that's not a big difference.
  2. Instead of d[v] which we update by
     - d[v] = min(d[v], d[u] + w(u,v))

     we keep k[v] which we update by
     - k[v] = min(k[v], w(u,v))

One thing that is similar:

Running time

- Exactly the same as Dijkstra:
  - O(mlog(n)) using a Red-Black tree as a priority queue.
  - O(m + nlog(n)) amortized time if we use a Fibonacci Heap*.

Two questions

1. Does it work?
   - That is, does it actually return a MST?
     - **Yes!**

2. How do we actually implement this?
   - the pseudocode above says “slowPrim”…
     - **Implement it basically the same way we’d implement Dijkstra!**

What have we learned?

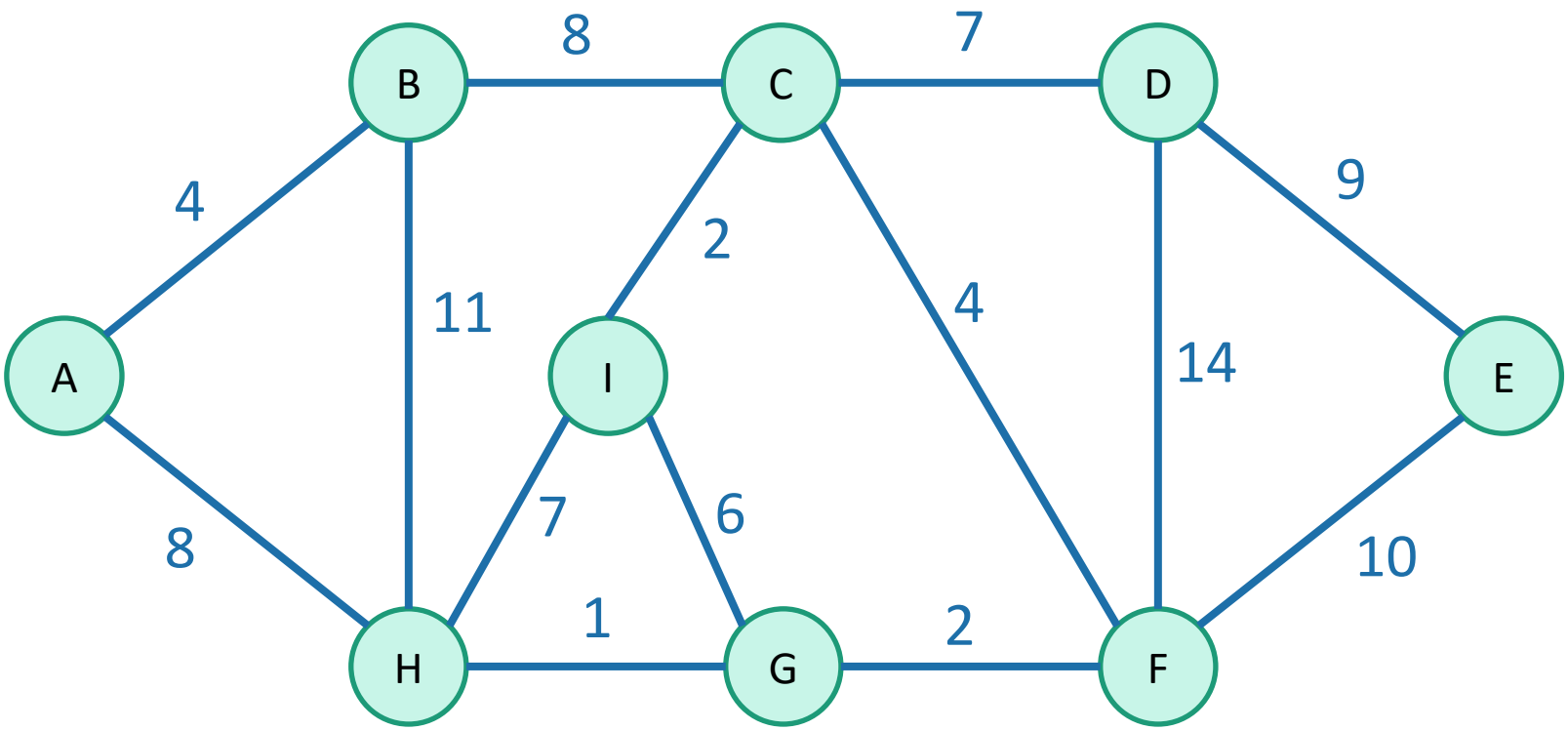
- Prim's algorithm greedily grows a tree
  - smells a lot like Dijkstra's algorithm
- It finds a Minimum Spanning Tree in time O(mlog(n))
  - if we implement it with a Red-Black Tree

- To prove it worked, we followed the same recipe for greedy algorithms we saw last time.
  - Show that, at every step, we **don't rule out success.**

That's not the only greedy algorithm

what if we just always take the cheapest edge?

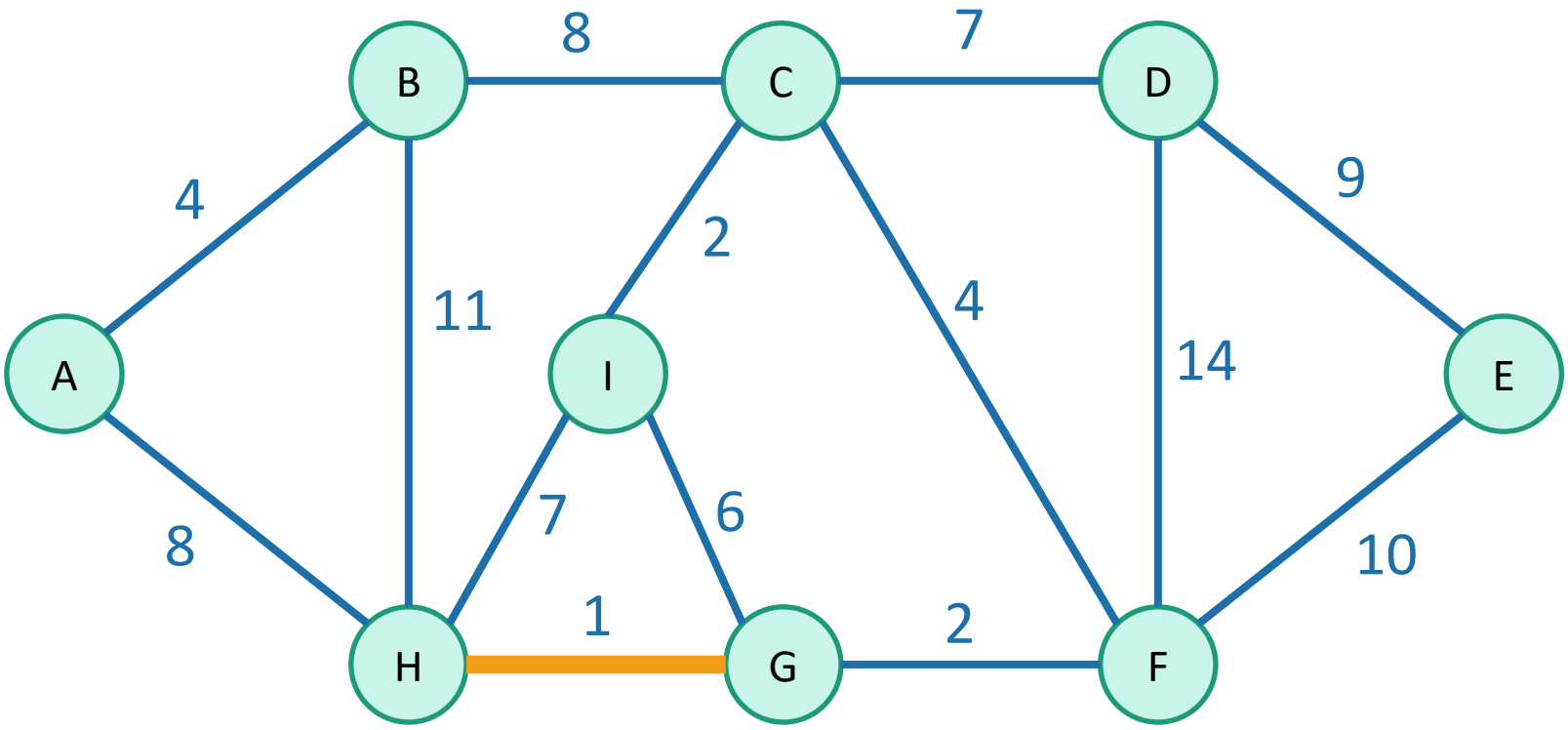
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

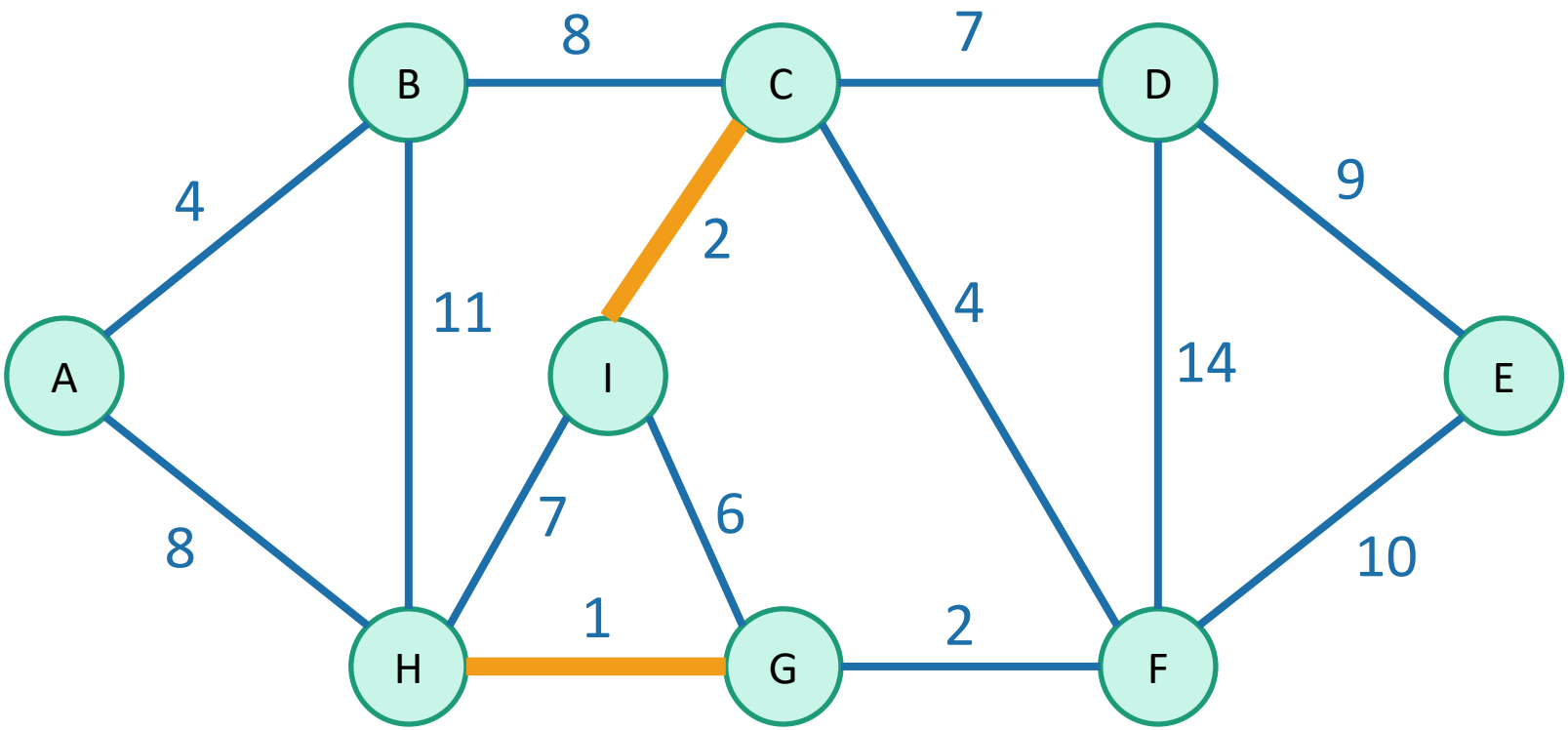
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

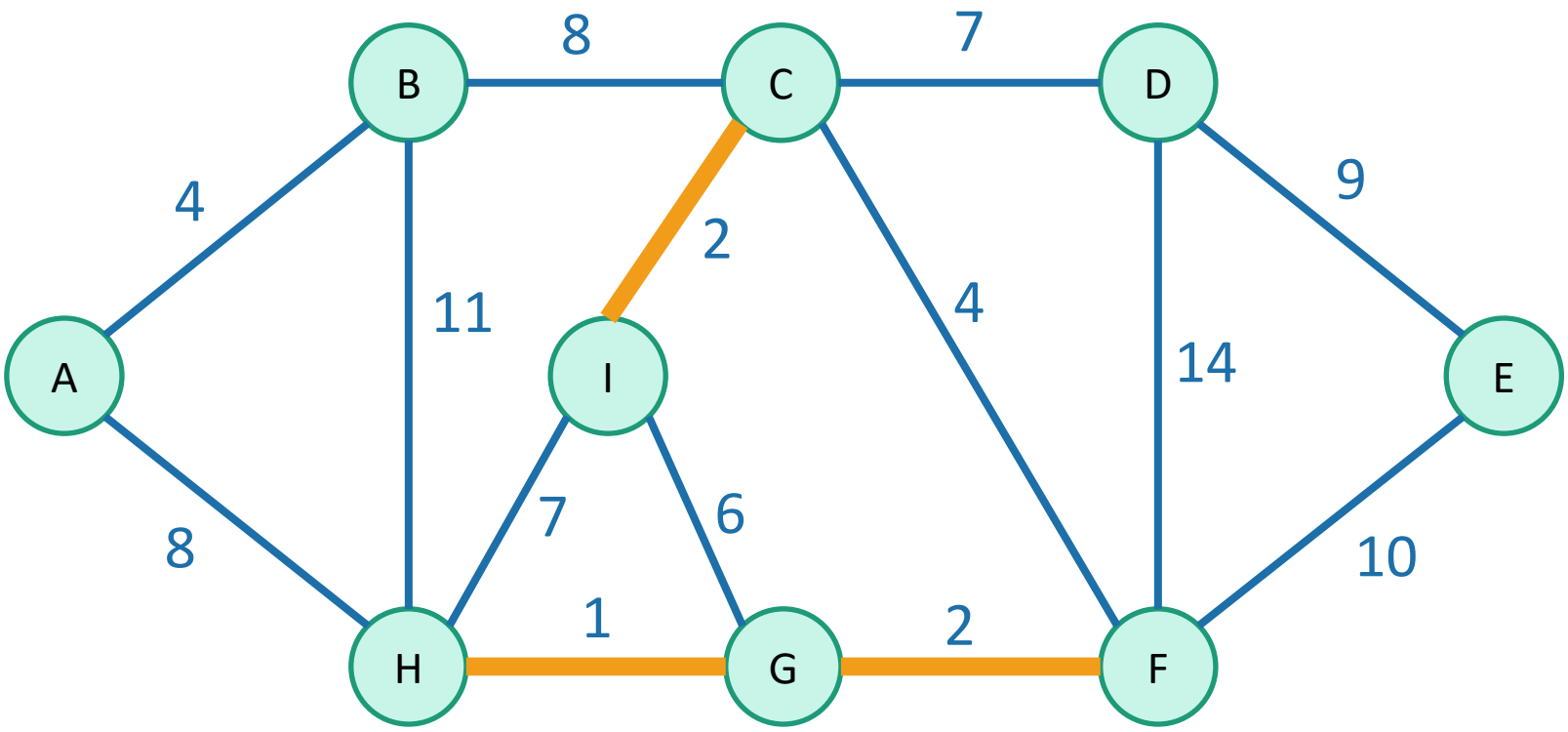
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

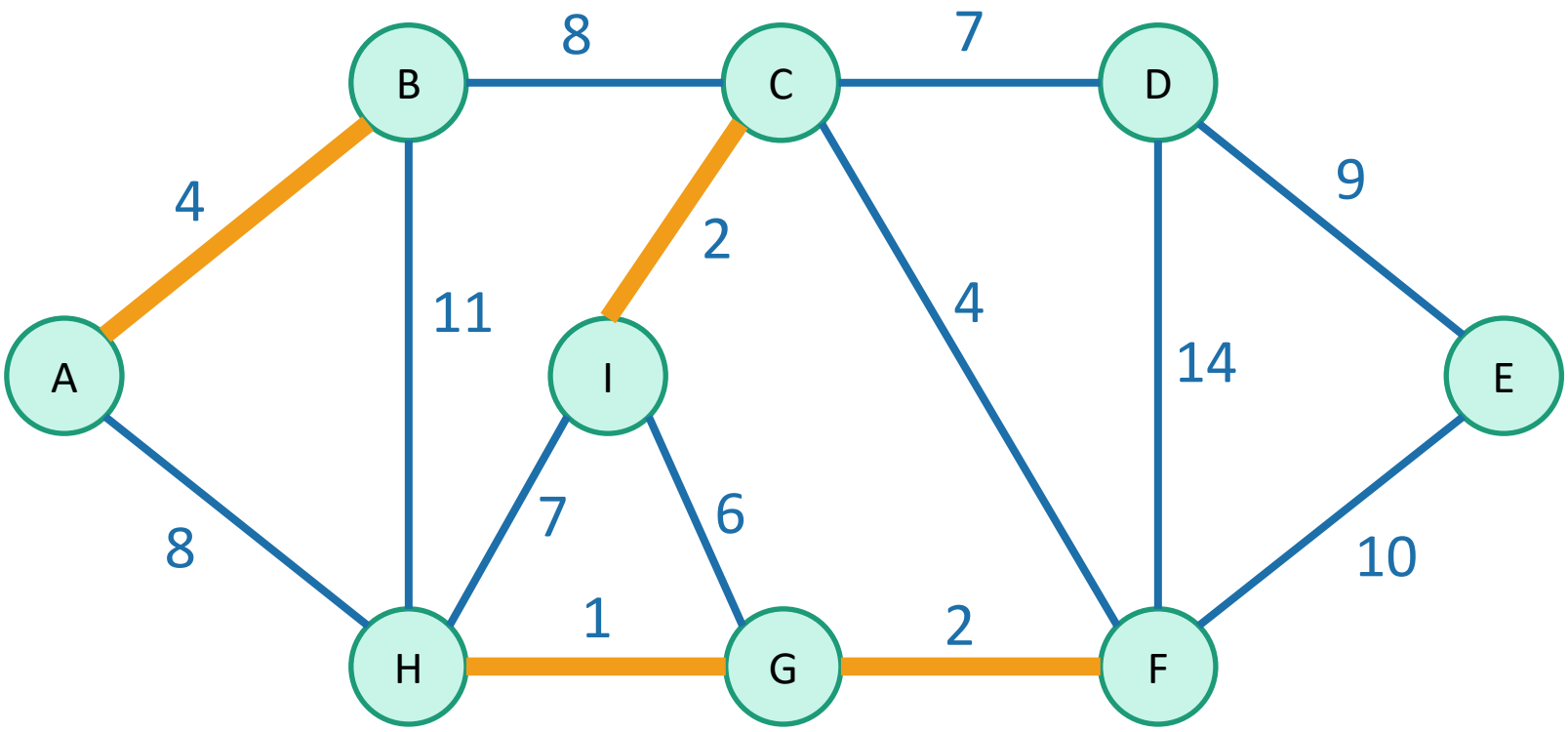
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

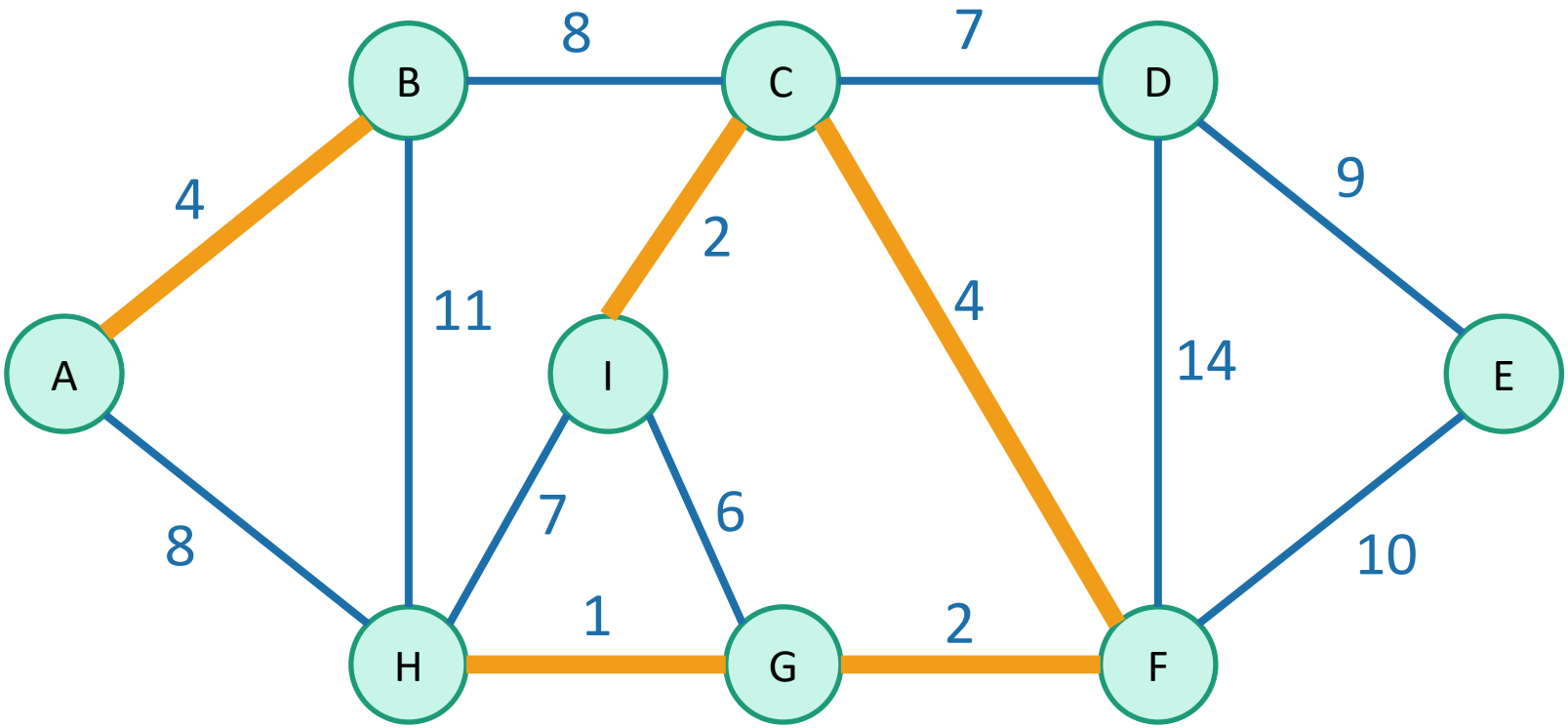
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

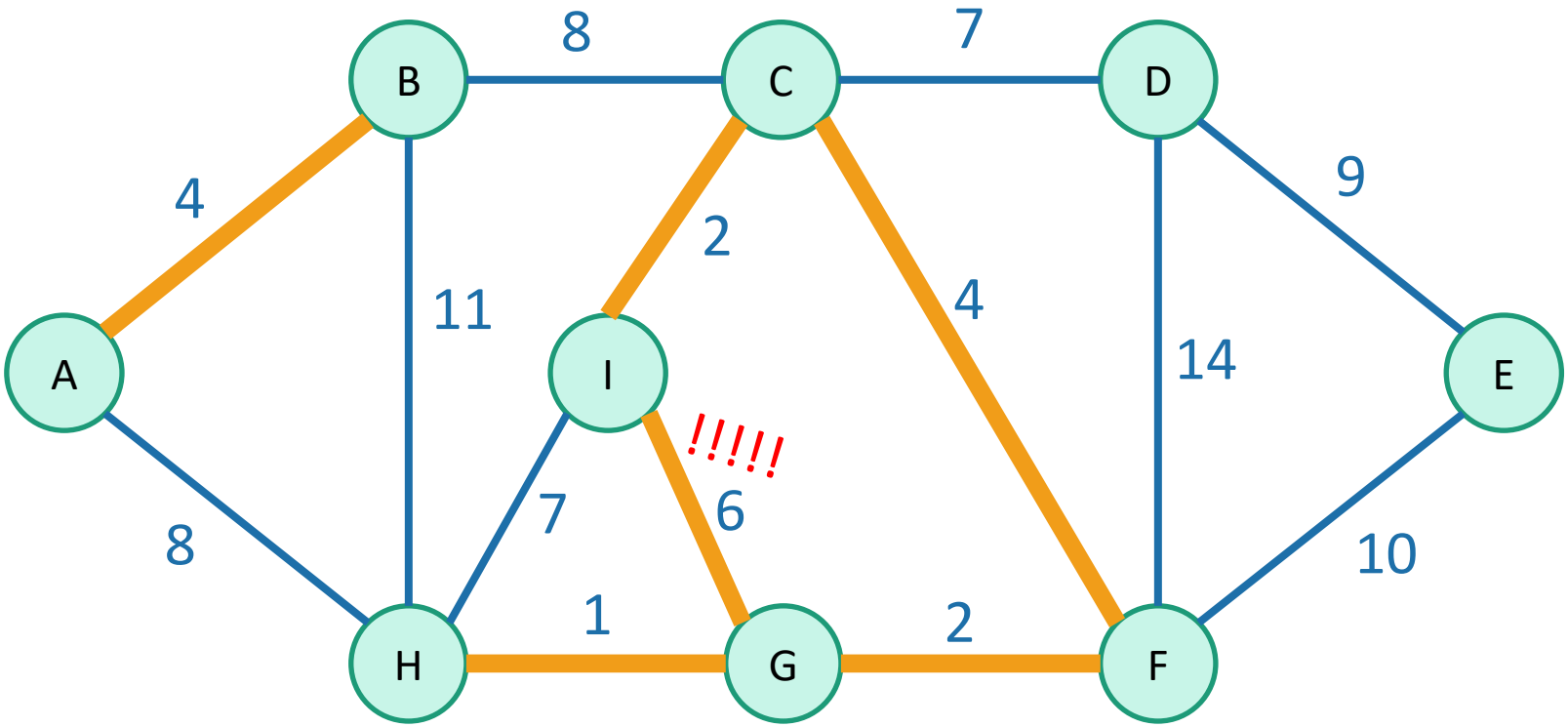


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

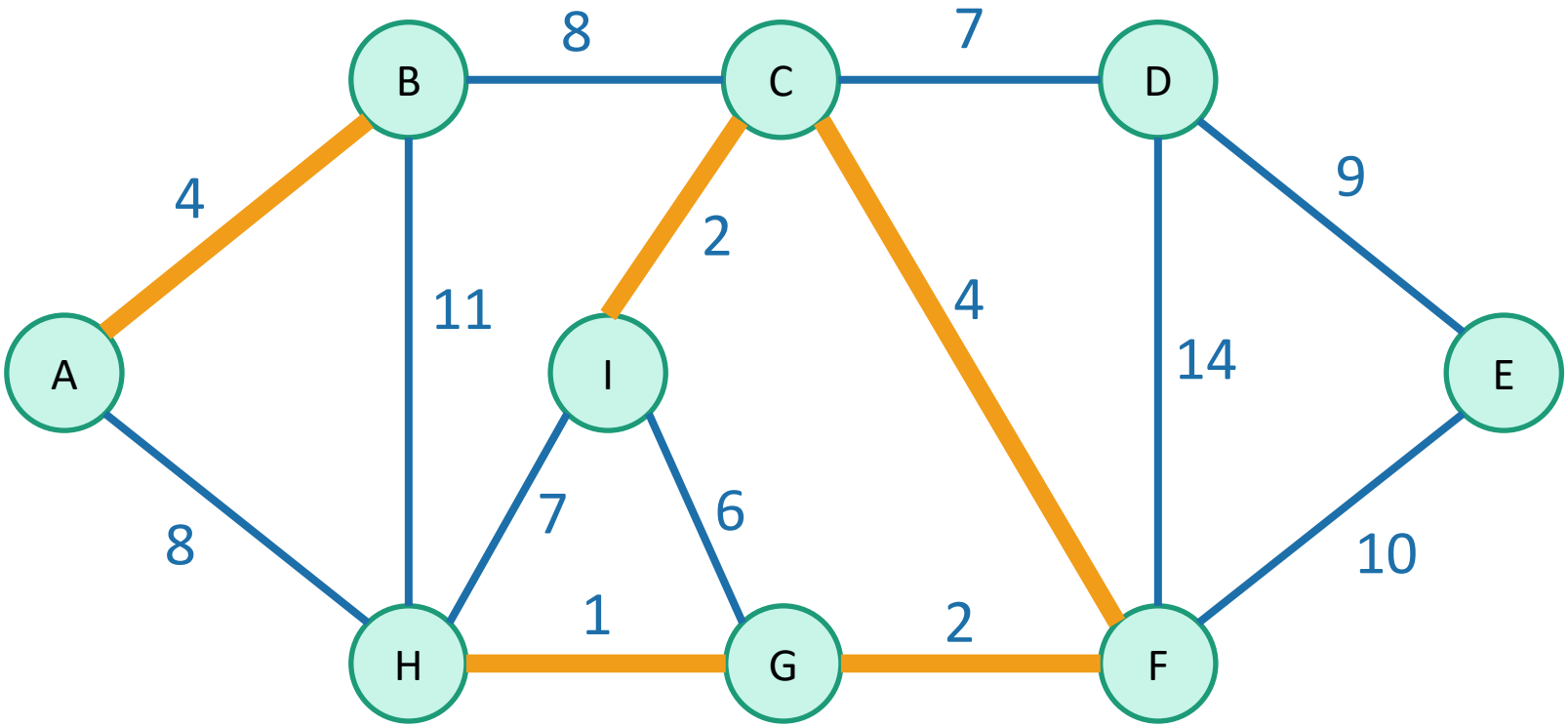


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

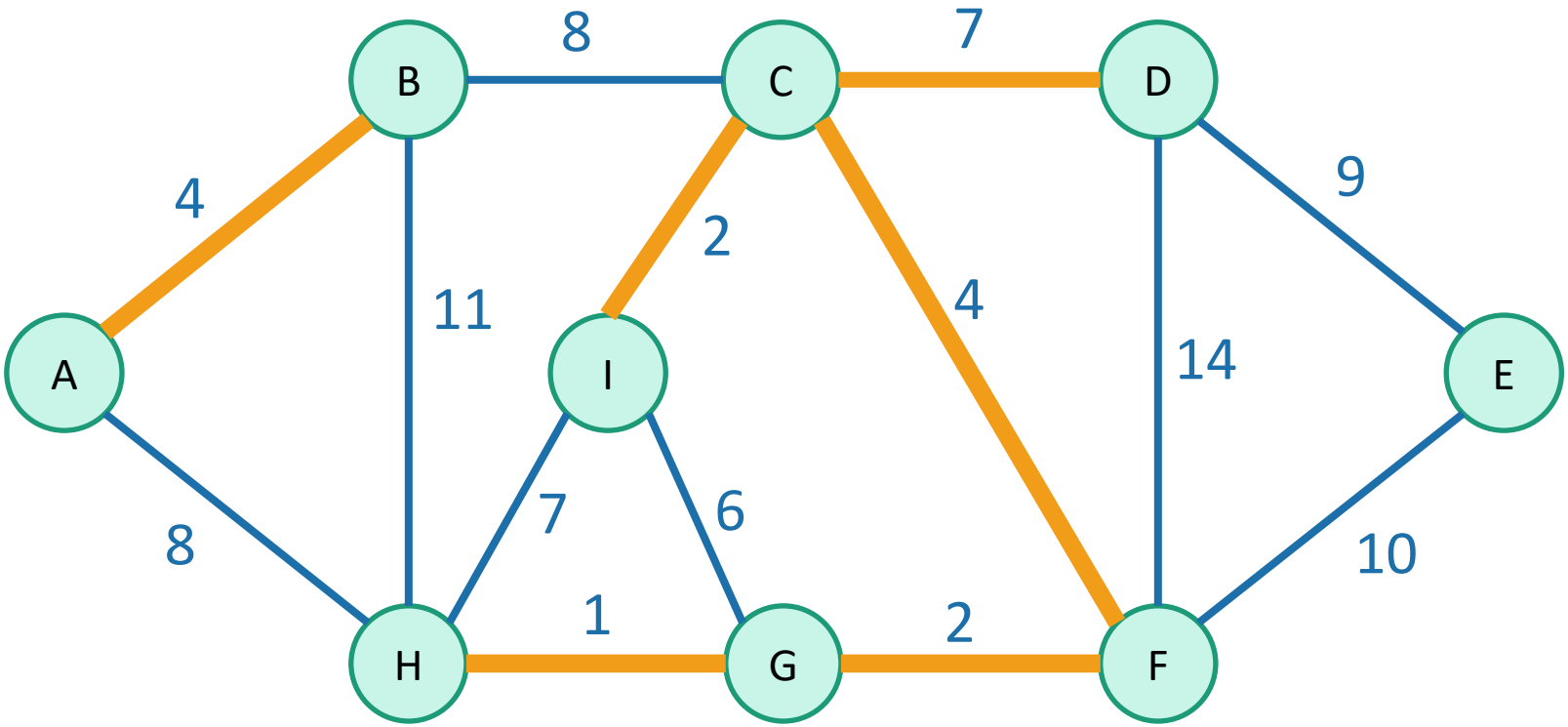


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

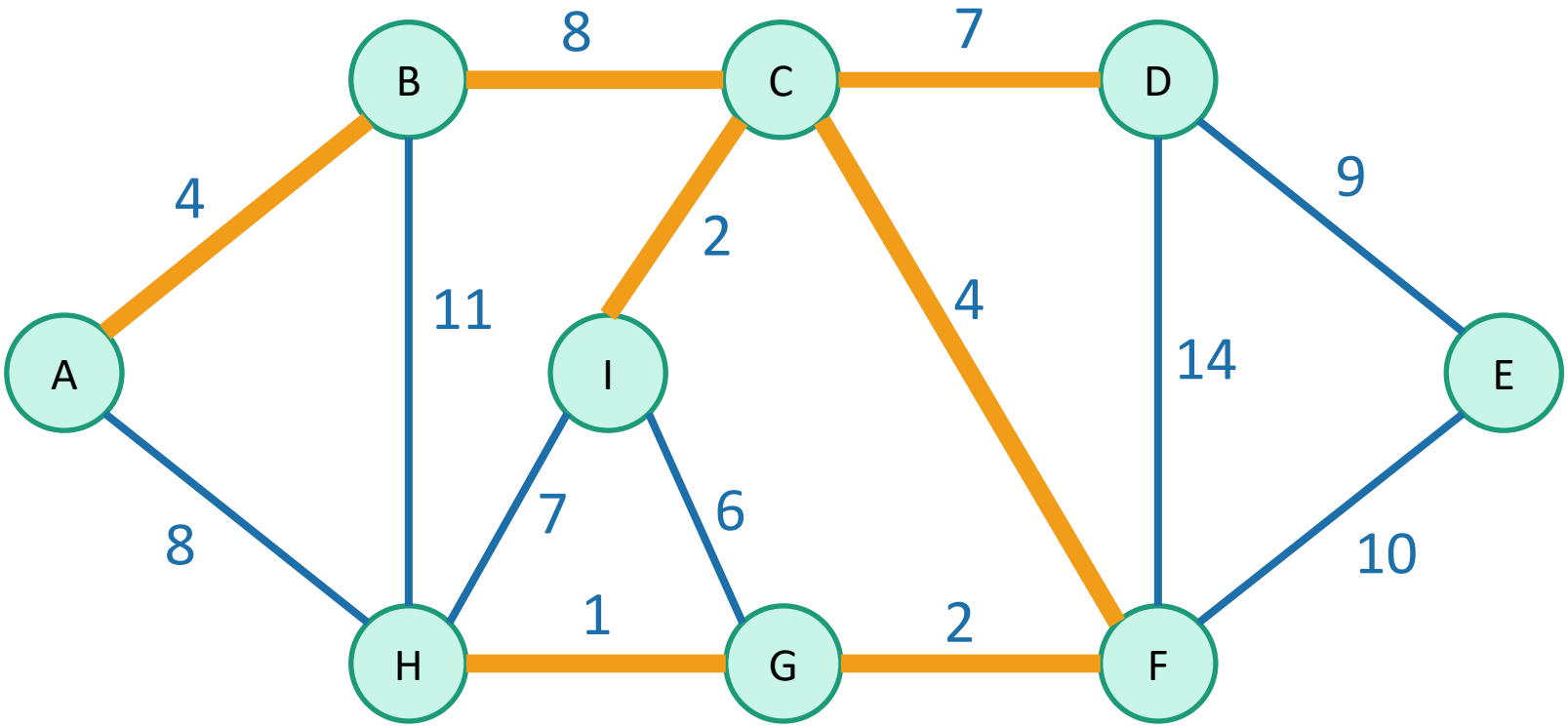


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle

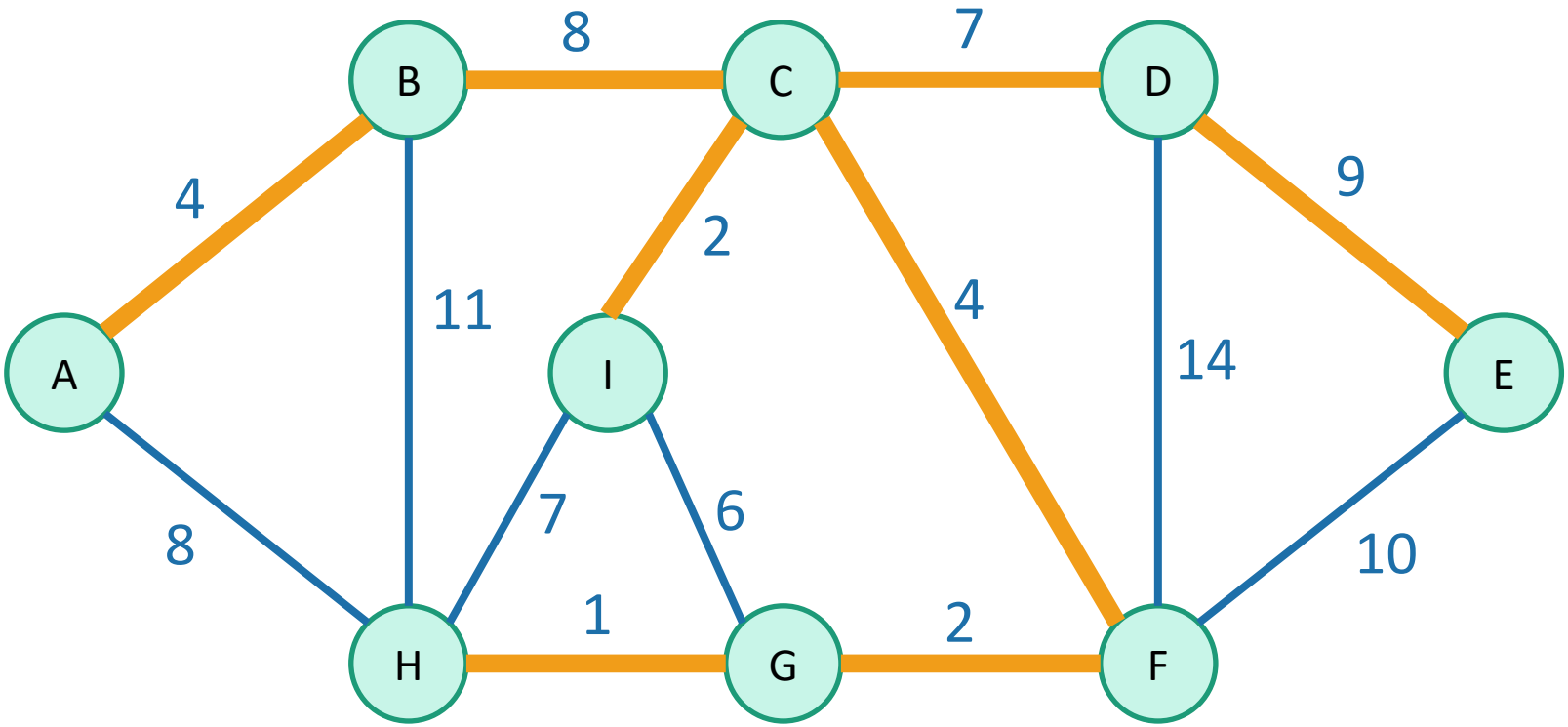


That's not the only greedy algorithm

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle



We've discovered

Kruskal's algorithm!

- **slowKruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - MST = {}
  - **for** e in E (in sorted order):
    - **if** adding e to MST won't cause a cycle:
      - add e to MST.
  - **return** MST

m iterations through this loop

How do we check this?

How **would** you figure out if added e would make a cycle in this algorithm?

Naively, the running time is ???:

- For each of m iterations of the for loop:
  - Check if adding e would cause a cycle…

Two questions

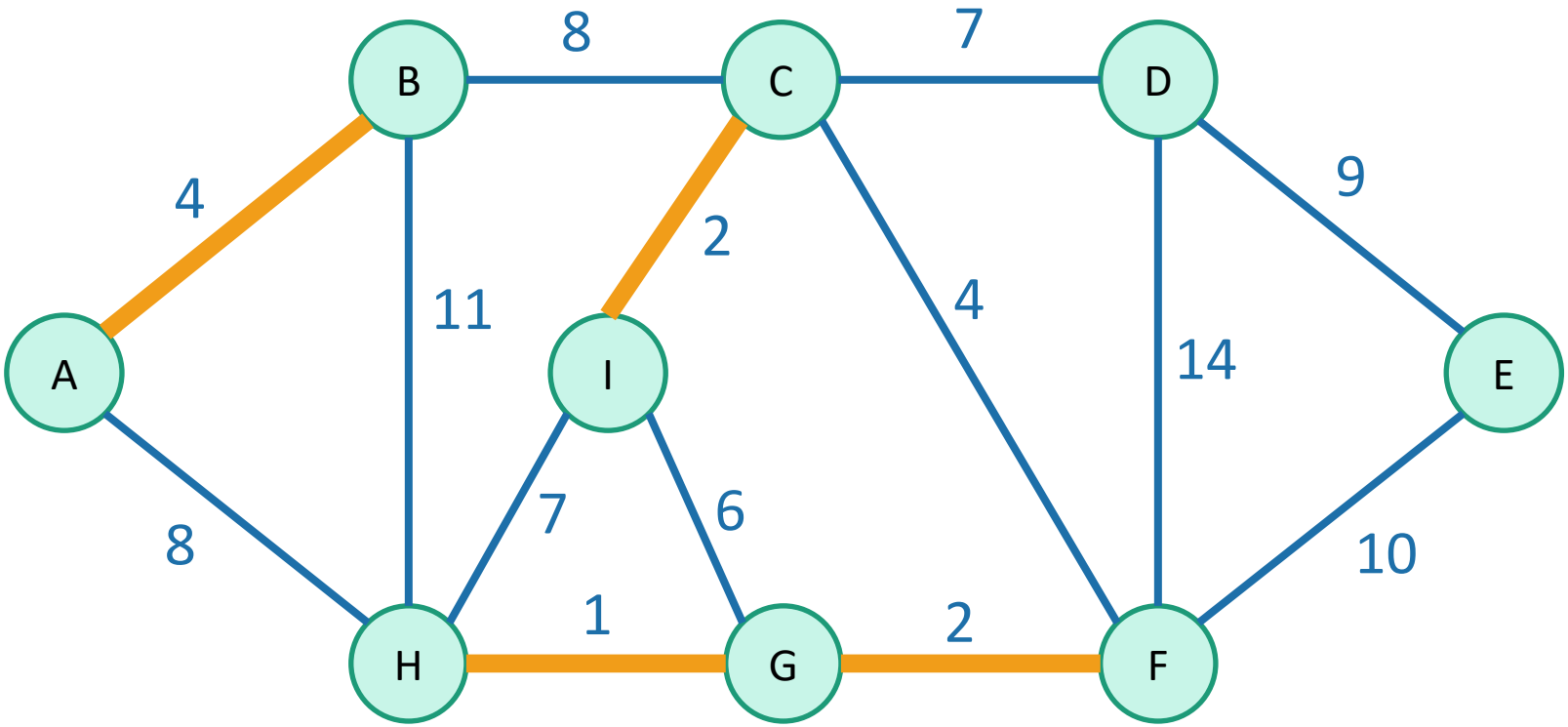
1. Does it work?
   - That is, does it actually return a MST?

2. How do we actually implement this?
   - the pseudocode above says “slowKruskal”…

Let’s do this one first

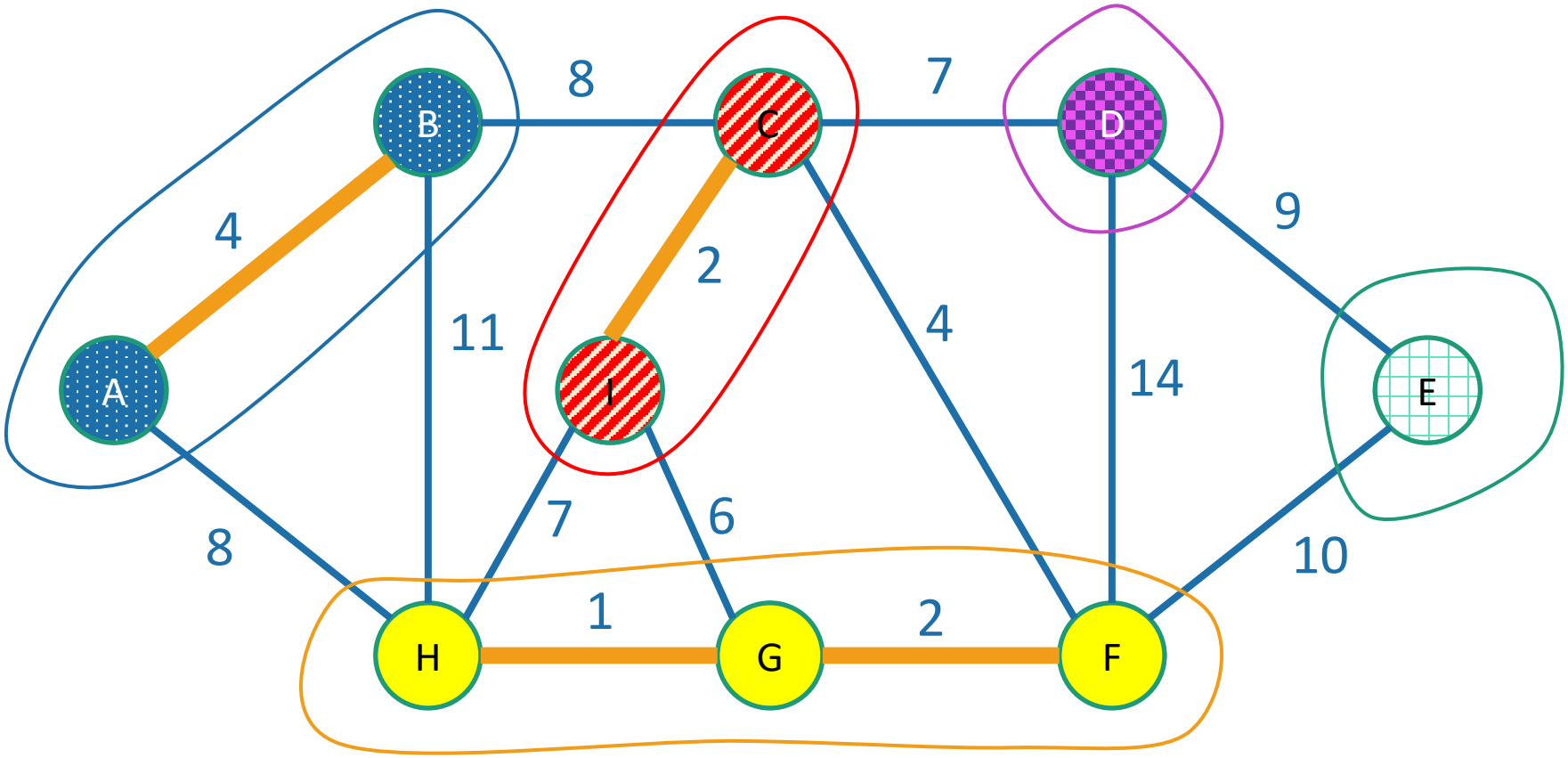
At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

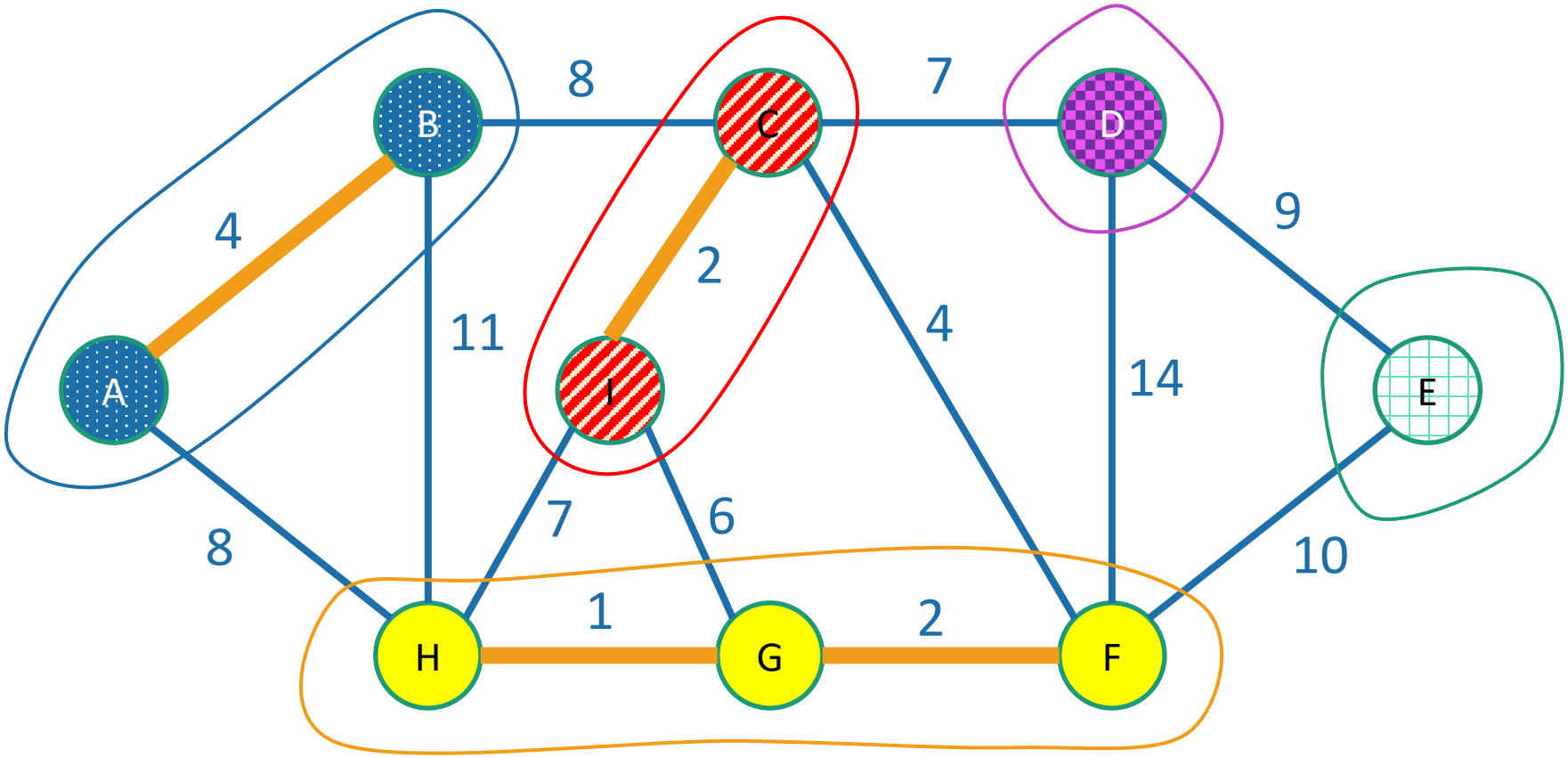
A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

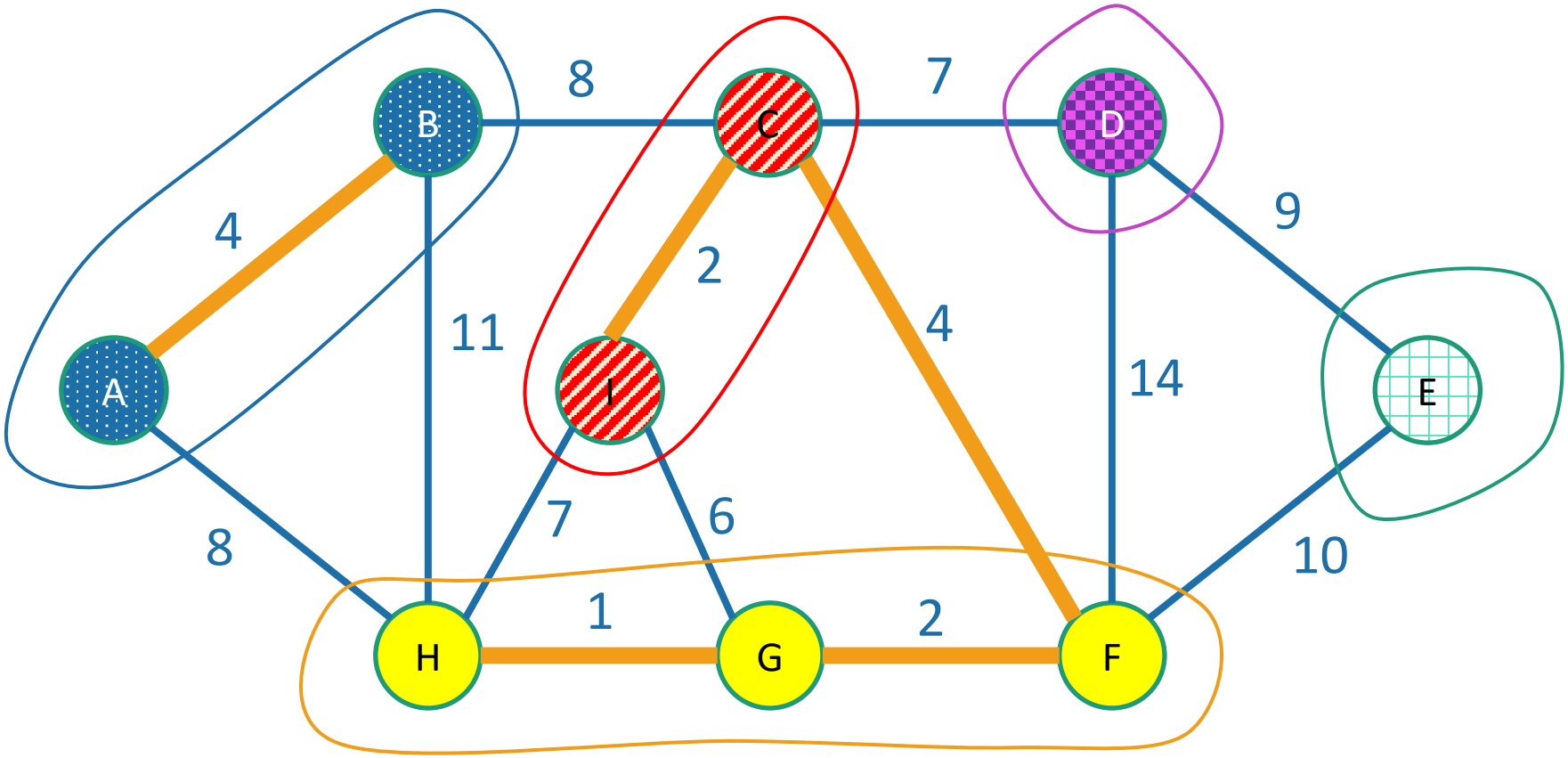
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

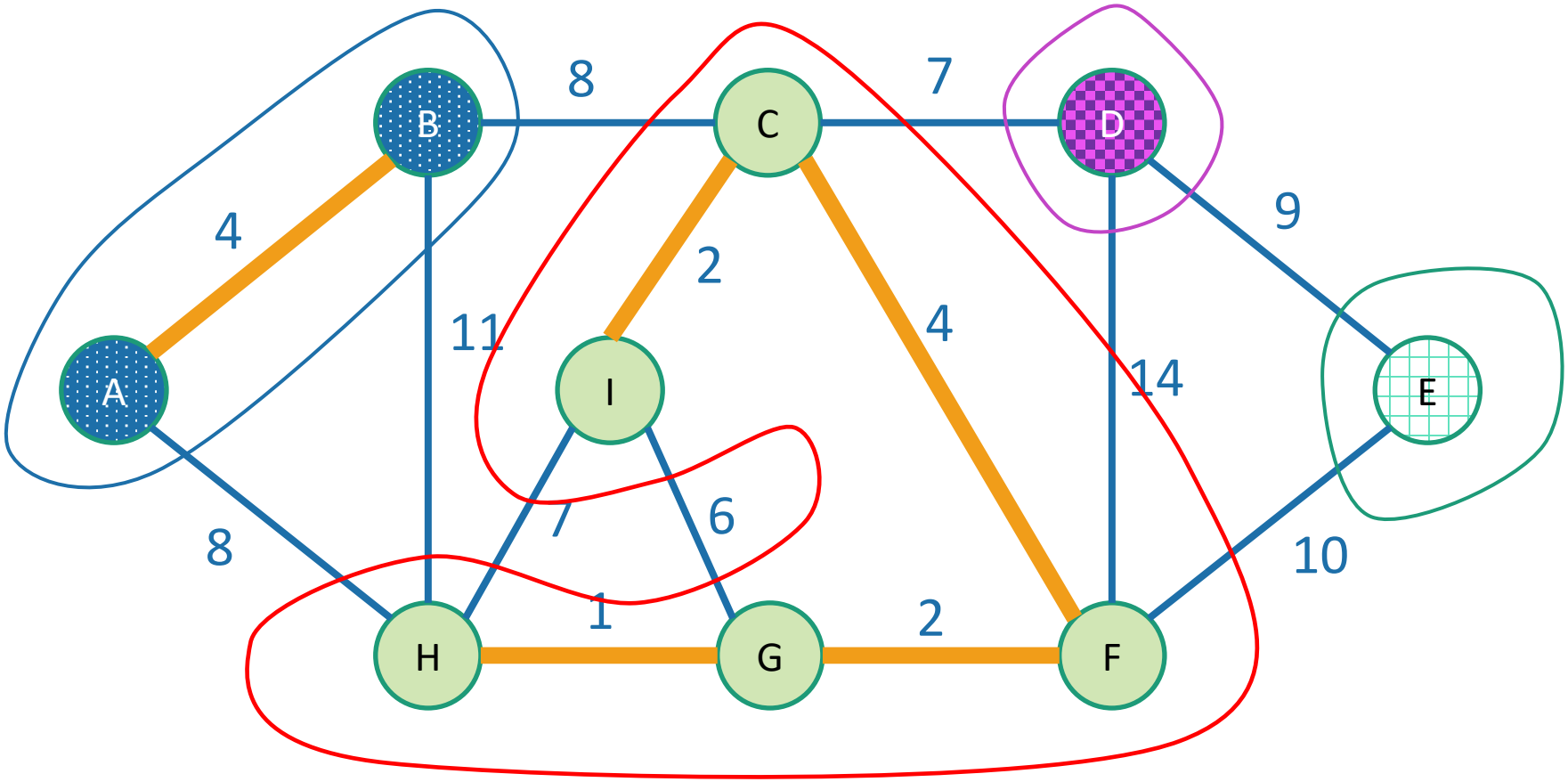
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

When we add an edge, we merge two trees:



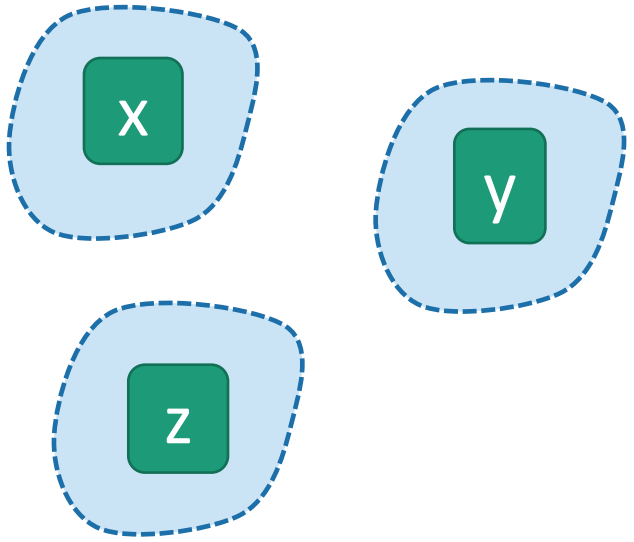
We never add an edge within a tree since that would create a cycle.

Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```

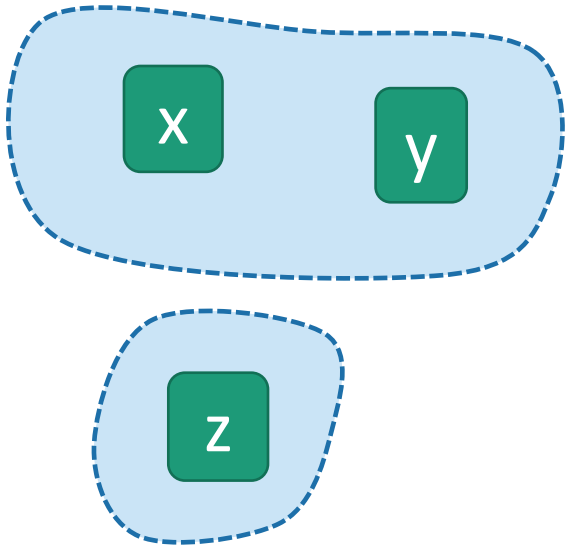


Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)
```



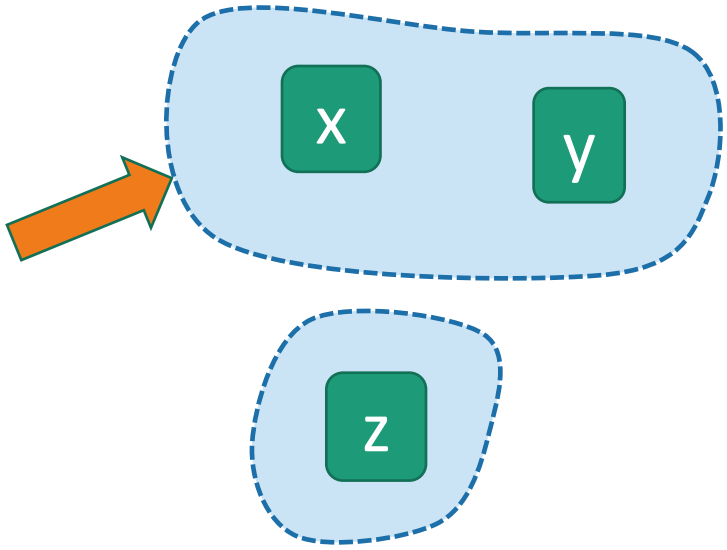
Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)
```

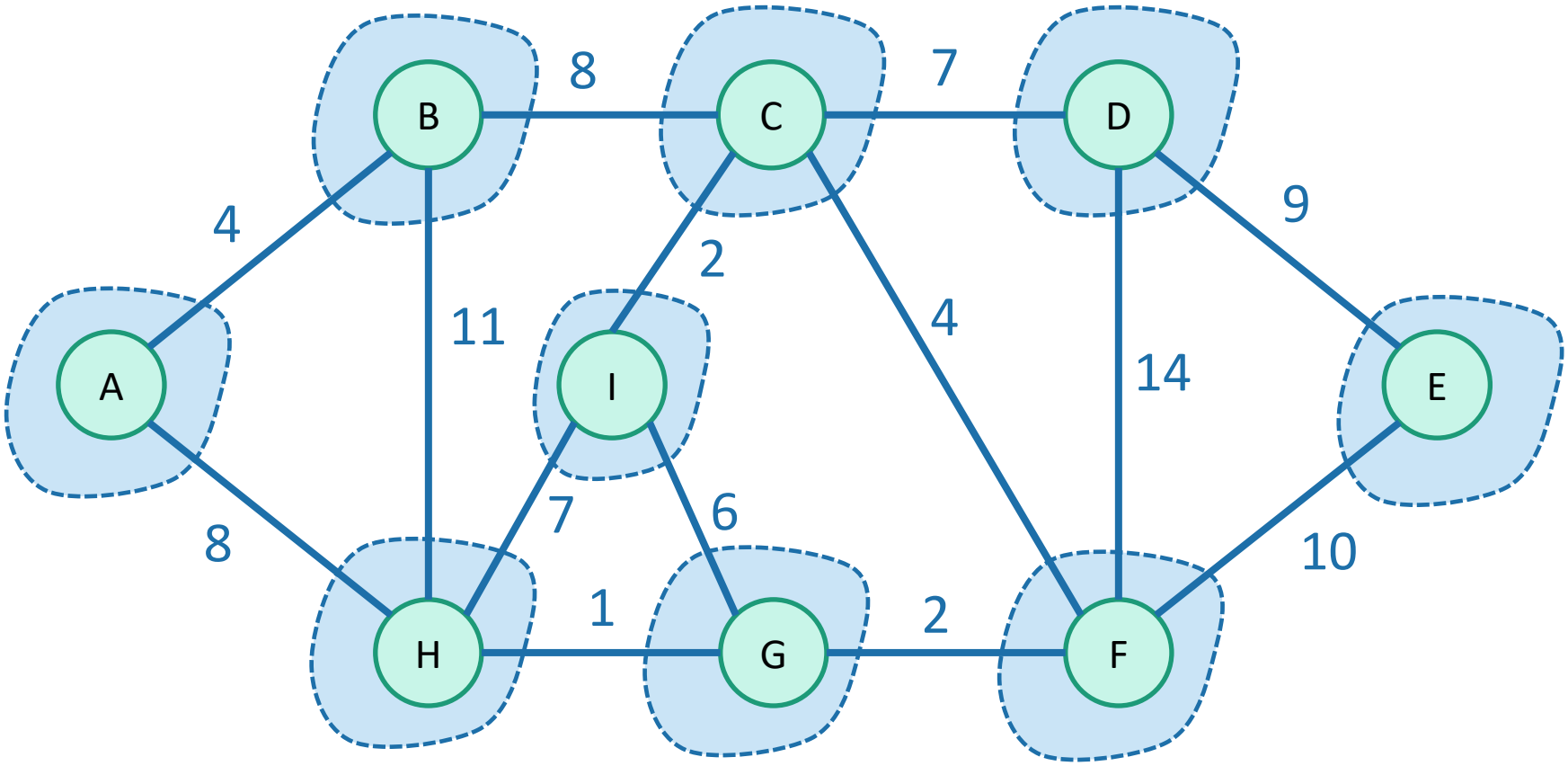


Kruskal pseudo-code

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

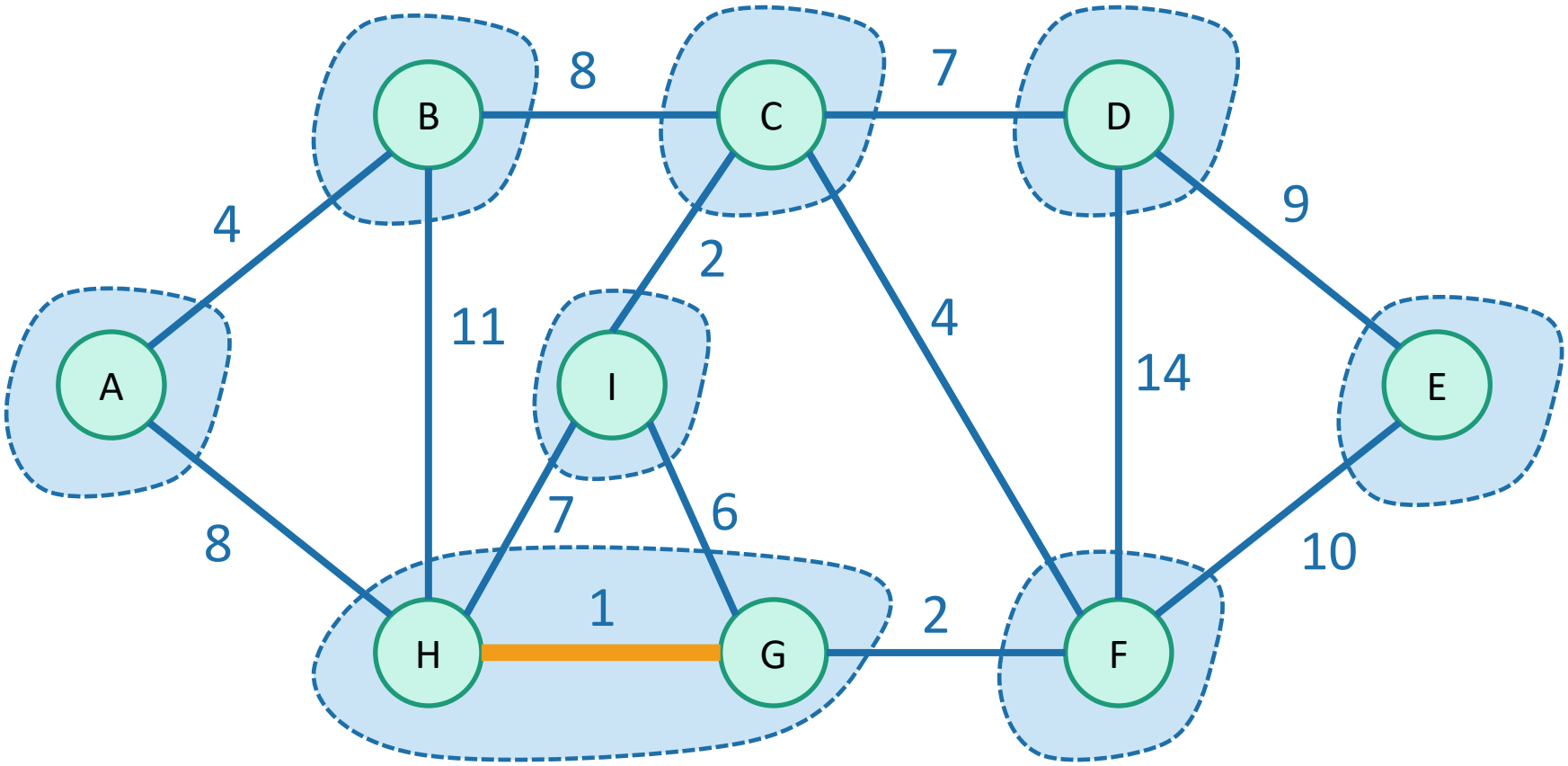
Once more…

To start, every vertex is in its own tree.



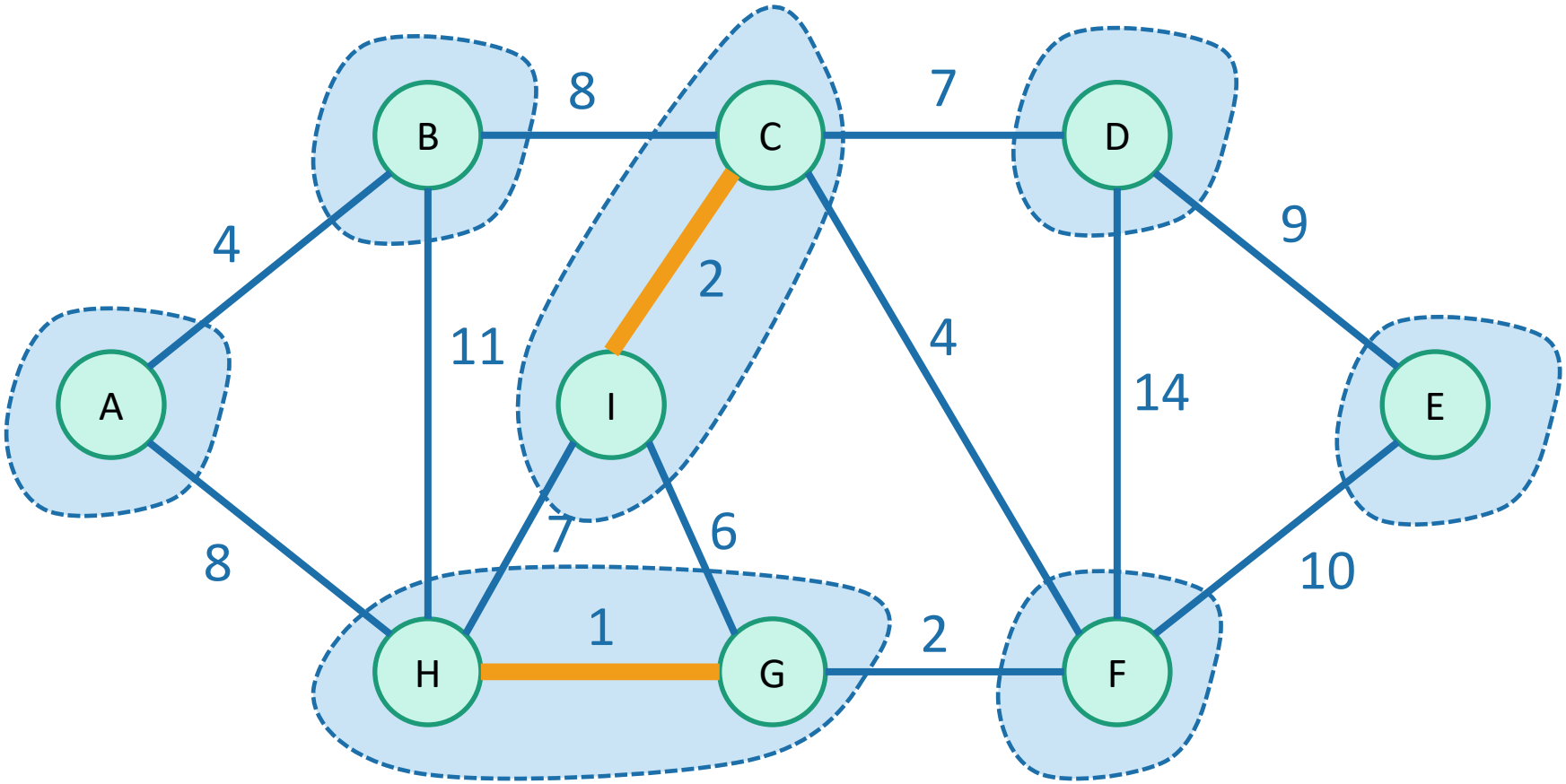
Once more…

Then start merging.



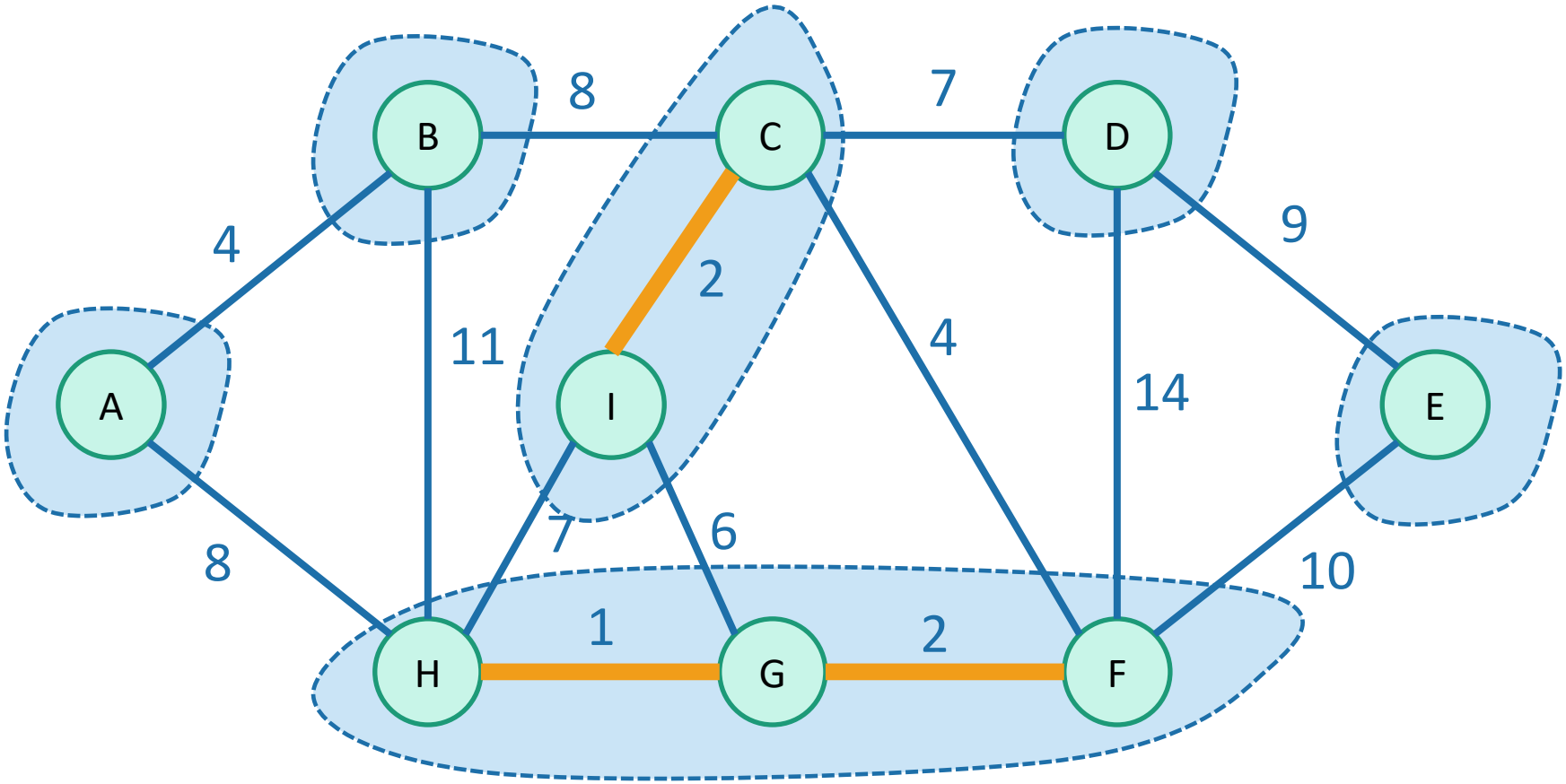
Once more…

Then start merging.



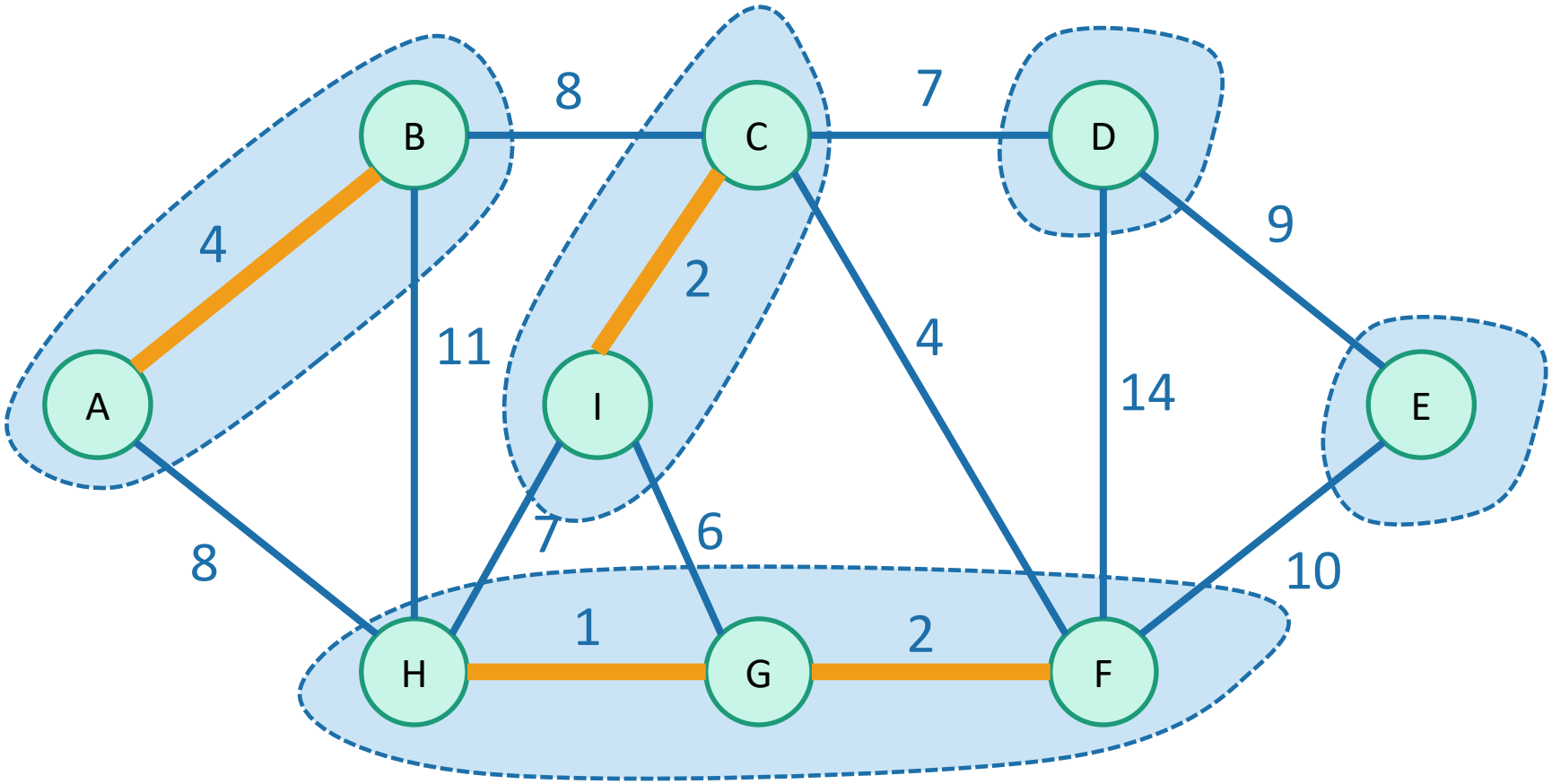
Once more…

Then start merging.



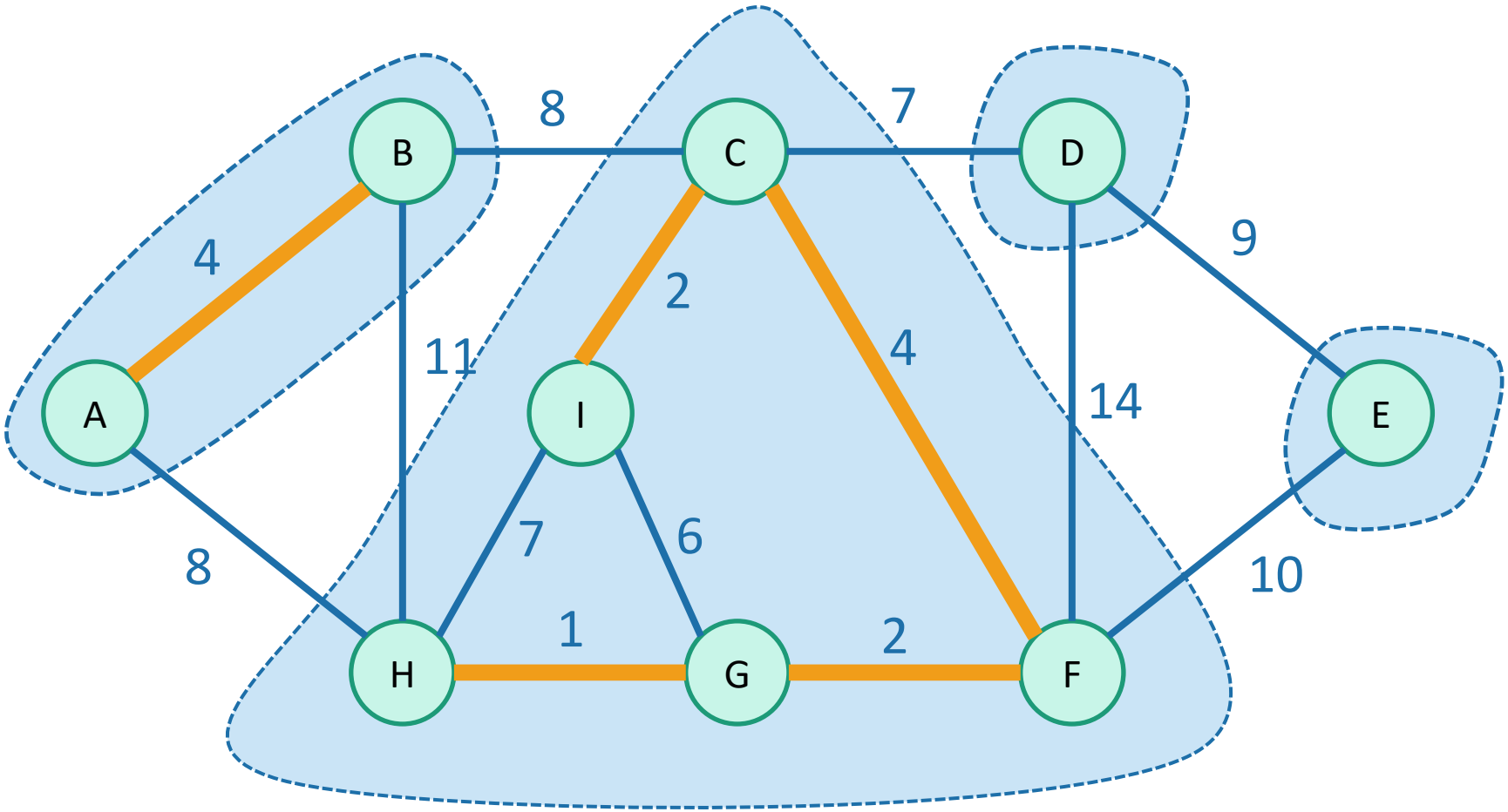
Once more…

Then start merging.



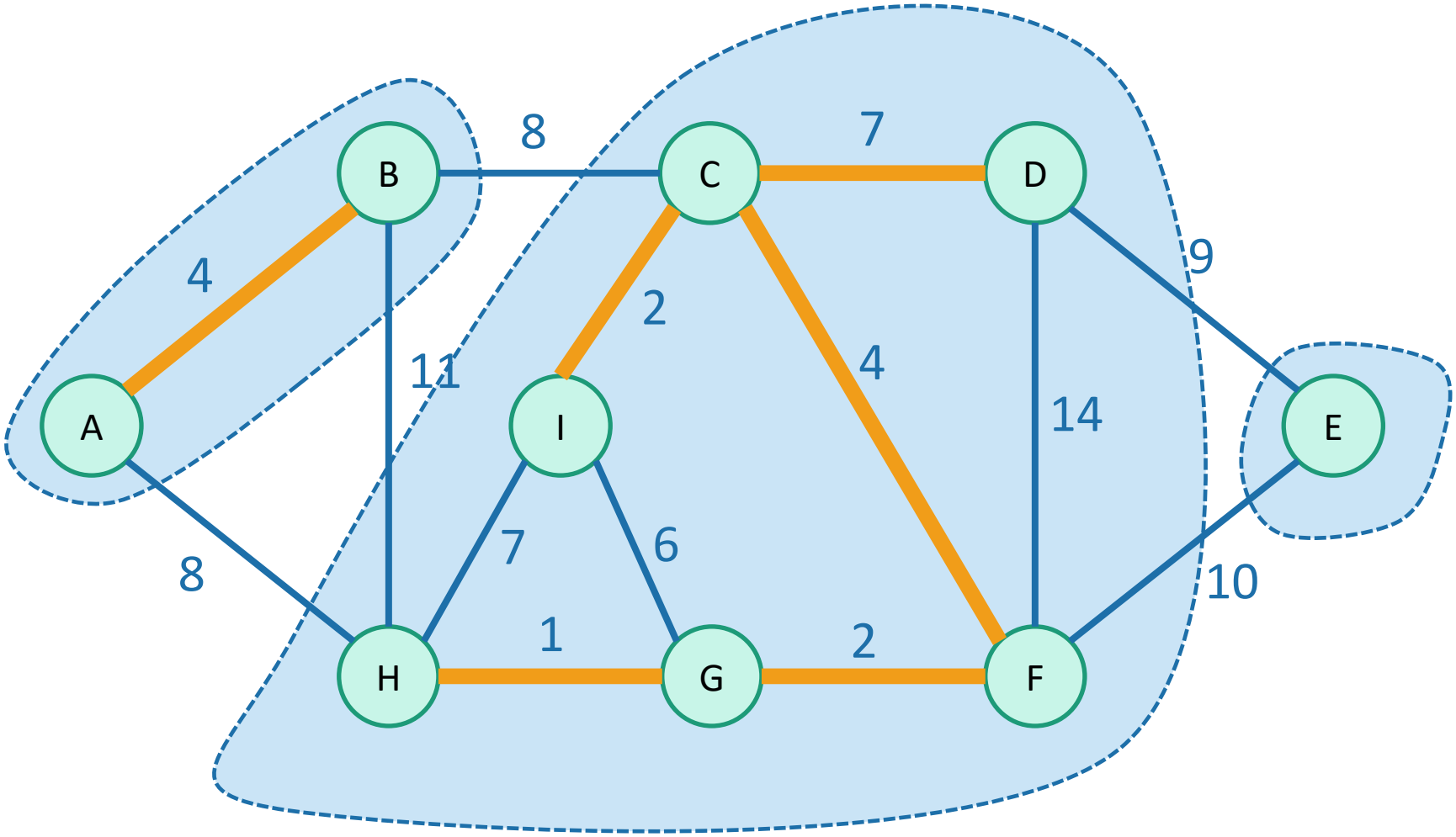
Once more…

Then start merging.



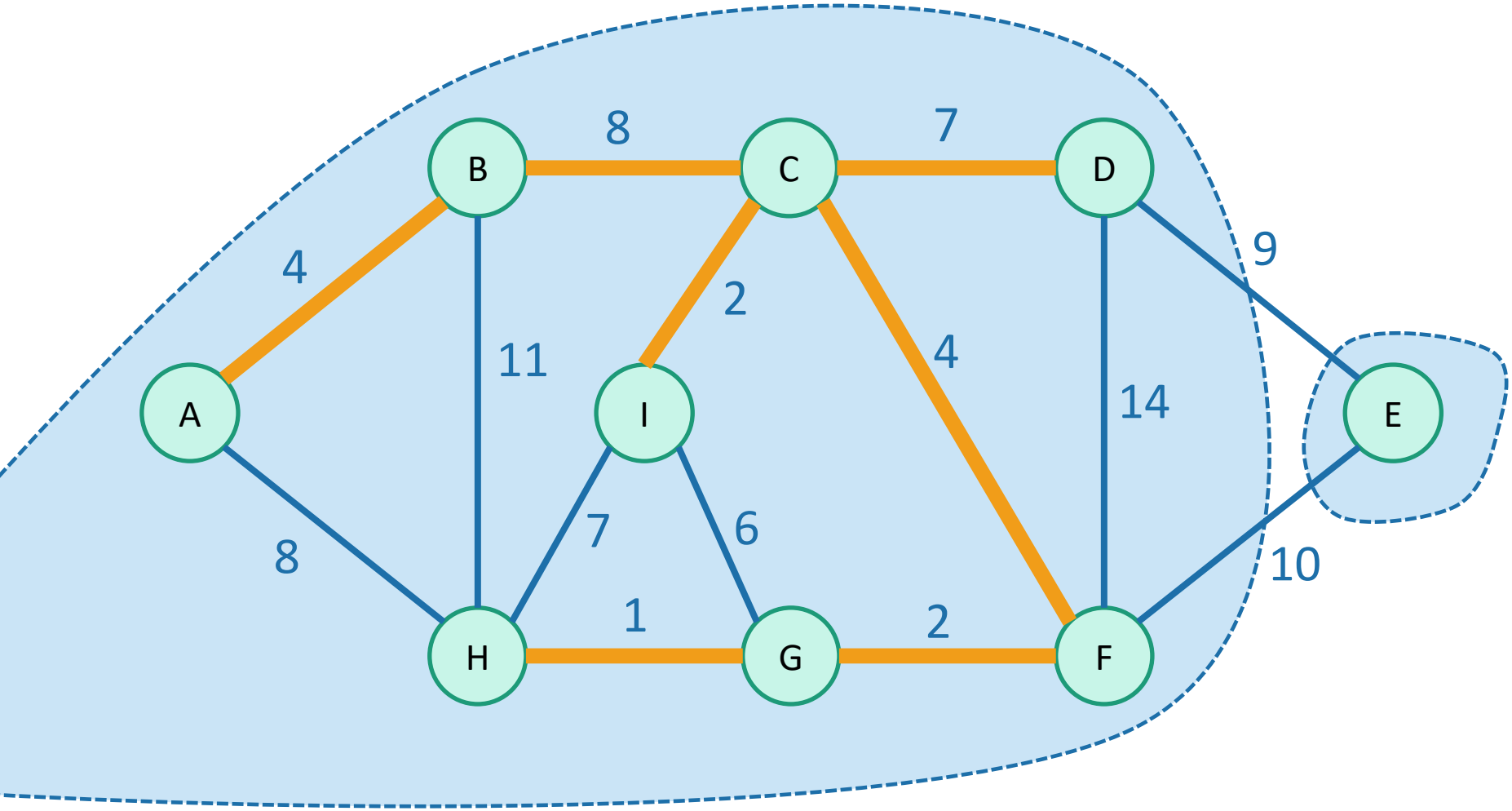
Once more…

Then start merging.



Once more…

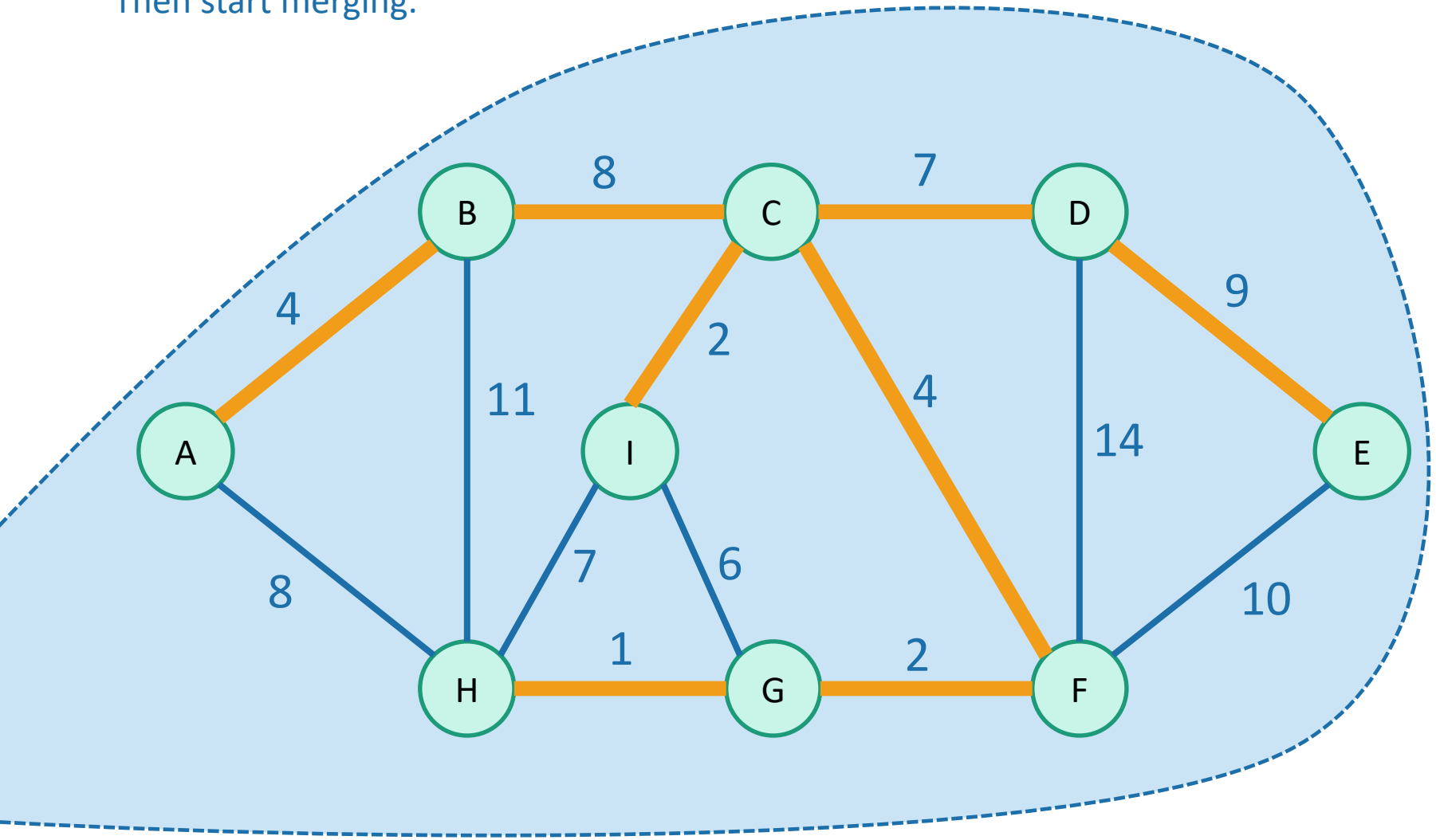
Then start merging.



Stop when we have one big tree!

Once more…

Then start merging.



Running time

- Sorting the edges takes O(m log(n))
  - In practice, if the weights are small integers we can use radixSort and take time O(m)
- For the rest:
  - n calls to **makeSet**
    - put each vertex in its own set
  - 2m calls to **find**
    - for each edge, **find** its endpoints
  - n calls to **union**
    - we will never add more than n-1 edges to the tree,
    - so we will never call **union** more than n-1 times.
- Total running time:
  - Worst-case O(mlog(n)), just like Prim.
  - Closer to O(m) if you can do radixSort

In practice, each of makeSet, find, and union run in constant time*

*technically, they run in *amortized time* O($\alpha(n)$), where $\alpha(n)$ is the *inverse Ackerman function*. $\alpha(n) \leq 4$ provided that n is smaller than the number of atoms in the universe.

Compare and contrast

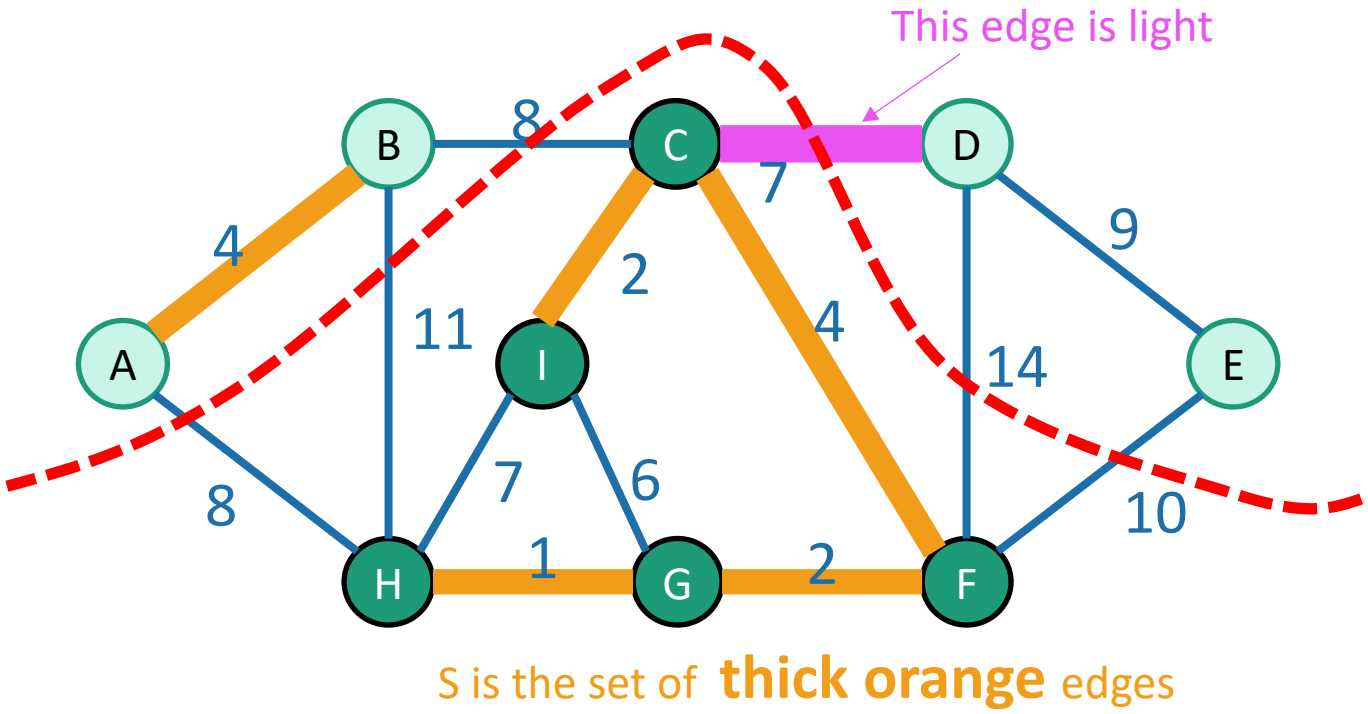
- Prim:
  - Grows a tree.
  - Time O(mlog(n)) with a red-black tree
  - Time O(m + nlog(n)) with a Fibonacci heap
- Kruskal:
  - Grows a forest.
  - Time O(mlog(n)) with a union-find data structure
  - If you can do radixSort on the edge weights, morally O(m)

Prim might be a better idea on dense graphs

Kruskal might be a better idea on sparse graphs if you can radixSort edge weights

Both Prim and Kruskal

- Greedy algorithms for MST.
- Similar reasoning:
  - Optimal substructure: subgraphs generated by cuts.
  - The way to make safe choices is to choose light edges crossing the cut.



Recap

- Two algorithms for Minimum Spanning Tree
  - Prim's algorithm
  - Kruskal's algorithm

- Both are (more) examples of greedy algorithms!
  - Make a **series of choices**.
  - Show that at each step, your choice **does not rule out success.**
  - At the end of the day, you haven't ruled out success, so **you must be successful**.