Maastricht University
Department of Advanced Computing Science

Algorithmic Design

Week 1:
Introduction, Greedy Algorithms, and More!

David Mestel and **Steven Chaplick**

2024-2025 Period 5

Practicalities

- Steven Chaplick (office: PHS1 C4.006, s.chaplick@maastrichtuniversity.nl)
- David Mestel (office: PHS1 C4.008, david.mestel@maastrichtuniversity.nl)

Lectures:

- Tuesdays & Thursdays, check **timetable** sometimes we have a break!
- Materials posted on Canvas under Modules

Practicalities

- Steven Chaplick (office: PHS1 C4.006, s.chaplick@maastrichtuniversity.nl)
- David Mestel (office: PHS1 C4.008, david.mestel@maastrichtuniversity.nl)

Lectures:

- Tuesdays & Thursdays, check **timetable** sometimes we have a break!
- Materials posted on Canvas under Modules

Tutorials:

- Thursdays (starting this week), check **timetable**, sometimes we have a break!
- Your chance to ask questions, work on tasks, collaborate, and clarify course material

Practicalities

- Steven Chaplick (office: PHS1 C4.006, s.chaplick@maastrichtuniversity.nl)
- David Mestel (office: PHS1 C4.008, david.mestel@maastrichtuniversity.nl)

Lectures:
- Tuesdays & Thursdays, check **timetable** sometimes we have a break!
- Materials posted on Canvas under Modules

Tutorials:
- Thursdays (starting this week), check **timetable**, sometimes we have a break!
- Your chance to ask questions, work on tasks, collaborate, and clarify course material

Assessment
- 4 Problem sets (5% each) with programming and mathematical tasks. Submissions via CodeGrade, and Canvas.
- Try using LaTeX – overleaf.com is a great way to get started :)
- Final Exam (80%)

What Is This Course About?

Background: Data Strctures and Algorithms, Discrete Mathematics

What Is This Course About?

Background: Data Strctures and Algorithms, Discrete Mathematics

Data Structures

- How to organizate and perform operations on data in memory

What Is This Course About?

Background: Data Strctures and Algorithms, Discrete Mathematics

Data Structures

- How to organizate and perform operations on data in memory

Algorithms:

- Methods uses data structures to implement functionality.

What Is This Course About?

Background: Data Strctures and Algorithms, Discrete Mathematics

Data Structures

- How to organizate and perform operations on data in memory

Algorithms:

- Methods uses data structures to implement functionality.

Complexity Analysis:

- How to measure the “speed” of an algorithm.

Here: deeper look into algorithmic paradigms, and when tasks are inherantly difficult to solve computationally.

What Is This Course About?

Background: Data Strctures and Algorithms, Discrete Mathematics

Data Structures

- How to organizate and perform operations on data in memory

Algorithms:

- Methods uses data structures to implement functionality.

Complexity Analysis:

- How to measure the “speed” of an algorithm.

Here: deeper look into algorithmic paradigms, and when tasks are inherantly difficult to solve computationally.

We will assume:

- You know how to build/use data structures
- You can do basic complexity analysis $\rightsquigarrow$ asymptotic runtime, Big-$\mathcal{O}$
- Can apply standard proof techniques $\rightsquigarrow$ cases, contradiction, induction, etc.

What Is This Course About?

To prove a statement is correct various techniques can be used:

- Proof by cases, proof by contradiction, induction, etc.

Picking the different methods can make the statement easier or more difficult to prove.

- Learning these general techniques gives us tools to find the “easy” proof.

What Is This Course About?

To prove a statement is correct various techniques can be used:

- Proof by cases, proof by contradiction, induction, etc.

Picking the different methods can make the statement easier or more difficult to prove.

- Learning these general techniques gives us tools to find the “easy” proof.

Computational tasks can be similarly approached via different algorithmic paradigms.

- Greedy, divide-and-conquer, dynammic programming, etc.

What Is This Course About?

To prove a statement is correct various techniques can be used:

- Proof by cases, proof by contradiction, induction, etc.

Picking the different methods can make the statement easier or more difficult to prove.

- Learning these general techniques gives us tools to find the “easy” proof.

Computational tasks can be similarly approached via different algorithmic paradigms.

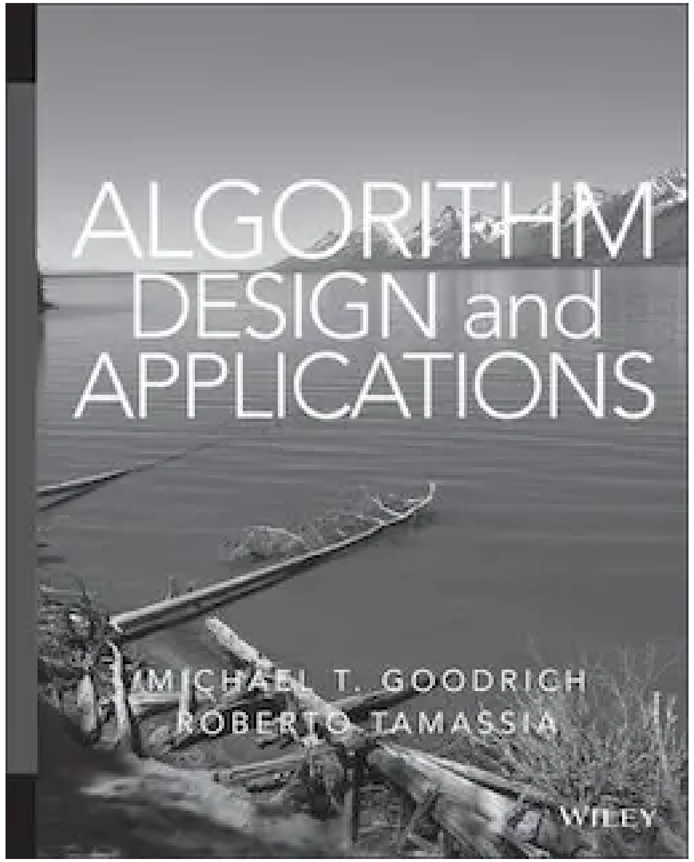
- Greedy, divide-and-conquer, dynammic programming, etc.

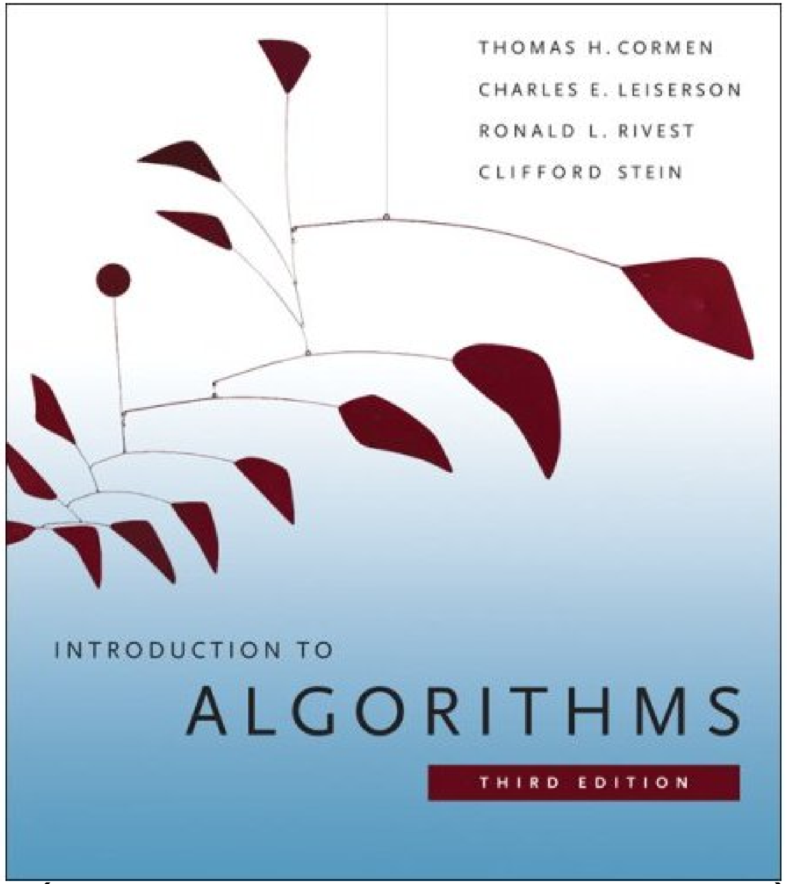
Course Outline:

- Start with these paradigms and corresponding tasks that can be solved with them
- Some tasks lack efficient algorithms to solve them, why? Classification of NP-hard problems.
- But we still need to tackle those problems $\leadsto$ approximations, heuristics, etc.

Books

The ‘official’ textbook for the course is [GT] Goodrich & Tamassia, ‘Algorithm Design and Applications’





(additional standard book) ‘Introduction to Algorithms’ [CLRS]

Week 1:

Greedy Algorithms: [GT §10, CLRS §16]

- Pick what looks the best step-by-step.

Amortised Analysis: [GT §1.4, CLRS §17]

- A different view on measuring runtime of algorithms.

Huffman Codes: [GT §10.3, CLRS §16.3]

- A simple but optimal* compression

Week 1:

Greedy Algorithms: [GT §10, CLRS §16]

- Pick what looks the best step-by-step.

Amortised Analysis: [GT §1.4, CLRS §17]

- A different view on measuring runtime of algorithms.

Huffman Codes: [GT §10.3, CLRS §16.3]

- A simple but optimal* compression

* Beware when optimality is claimed, there can be some strange conditions attached ;)

Greedy algorithms

- **Greedy** means making the choice that seems best now, without thinking about the future
- Algorithmically, “do the what looks the best each step”
- But what does **seems best now** mean? Need to pick the right measure of “best”.

Greedy algorithms

- **Greedy** means making the choice that seems best now, without thinking about the future
- Algorithmically, "do the what looks the best each step"
- But what does **seems best now** mean? Need to pick the right measure of "best".

For example, you come across a pile of change on a table. You can only take 10 coins. How can you maximize the amount of money you get?

Greedy algorithms

- **Greedy** means making the choice that seems best now, without thinking about the future
- Algorithmically, "do the what looks the best each step"
- But what does **seems best now** mean? Need to pick the right measure of "best".

For example, you come across a pile of change on a table. You can only take 10 coins. How can you maximize the amount of money you get?

A simple greedy algorithm:

1. Pick up (one of) the most valuable coin(s)
2. If you have taken 10 coins, stop. If not, go to 1

Greedy algorithms

- **Greedy** means making the choice that seems best now, without thinking about the future
- Algorithmically, “do the what looks the best each step”
- But what does **seems best now** mean? Need to pick the right measure of “best”.

For example, you come across a pile of change on a table. You can only take 10 coins. How can you maximize the amount of money you get?

A simple greedy algorithm:

1. Pick up (one of) the most valuable coin(s)
2. If you have taken 10 coins, stop. If not, go to 1

did you spot the termination bug here?

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

Now it becomes a KnapSack problem:

KnapSack
Input: Given a weight capacity $W \geq 0$ and n objects with non-negative weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, i.e., object i has weight w_i and value v_i.

Goal: Pick a subset S of the given objects of maximum value and total weight at most W.

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

Now it becomes a $\textsc{KnapSack}$ problem:

$\textsc{KnapSack}$
Input: Given a weight capacity $W \geq 0$ and n objects with non-negative weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, i.e., object i has weight w_i and value v_i.

Goal: Pick a subset S of the given objects of maximum value and total weight at most W.

How can we generalize the previous greedy approach?

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

Now it becomes a KNAPSACK problem:

KNAPSACK
Input: Given a weight capacity $W \geq 0$ and n objects with non-negative weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, i.e., object i has weight w_i and value v_i.

Goal: Pick a subset S of the given objects of maximum value and total weight at most W.

How can we generalize the previous greedy approach?

- Pick coins by *value density*: $\dfrac{v_i}{w_i}$

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

Now it becomes a $\textsc{KnapSack}$ problem:

$\textsc{KnapSack}$
Input: Given a weight capacity $W \geq 0$ and n objects with non-negative weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, i.e., object i has weight w_i and value v_i.

Goal: Pick a subset S of the given objects of maximum value and total weight at most W.

How can we generalize the previous greedy approach?

- Pick coins by *value density*: $\dfrac{v_i}{w_i}$ Counter-example to optimality of this approach?

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

Now it becomes a $\textsc{KnapSack}$ problem:

$\textsc{KnapSack}$
Input: Given a weight capacity $W \geq 0$ and n objects with non-negative weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, i.e., object i has weight w_i and value v_i.

Goal: Pick a subset S of the given objects of maximum value and total weight at most W.

How can we generalize the previous greedy approach?

- Pick coins by *value density*: $\dfrac{v_i}{w_i}$ Counter-example to optimality of this approach?

Consider 3 coins where the (w_i, v_i) are (3,5), (2,3), (2,3), and capacity $W = 4$.

When greediness goes wrong

Suppose our coins not only have a value, but also a weight.
And, we cannot just take any 10, we are limited by the **weight** of the coins we pick.

Now it becomes a KNAPSACK problem:

KNAPSACK
Input: Given a weight capacity $W \geq 0$ and n objects with non-negative weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ respectively, i.e., object i has weight w_i and value v_i.

Goal: Pick a subset S of the given objects of maximum value and total weight at most W.

How can we generalize the previous greedy approach?

- Pick coins by *value density*: $\dfrac{v_i}{w_i}$ Counter-example to optimality of this approach?

Consider 3 coins where the (w_i, v_i) are (3,5), (2,3), (2,3), and capacity $W = 4$.

Greedy gets the first coin (value$=5$), but it's better to take the other two coins (value$=6$).

When can we use a greedy approach?

Need an **Optimization problem** with a set of **configurations** and an **objective function**, and the goal is to find a configuration that maximizes (or minimizes) the value of the objective function.

When can we use a greedy approach?

Need an **Optimization problem** with a set of **configurations** and an **objective function**, and the goal is to find a configuration that maximizes (or minimizes) the value of the objective function.

Outline of a greedy algorithm:

- View the problem as a series of choices
- At every point, make the choice that most improves the objective function
- Do this until you have made enough choices to have a solution

Given a particular problem there is not one, single, greedy algorithm for it

Challenges in Greedy Design

- Different algorithms result from varying the objective function, and the starting conditions.
- How to decide which choices to make?
  - It is important that the start position and the rule for choosing a next move allow for *enough* configurations to be reached
- Deciding on an objective function:
  - Technically, the objective function only applies when the configuration is complete
  - One has to create a function to apply to incomplete configurations
  - Usually this is straightforward, but different choices can lead to different algorithms

So when do we use a greedy approach?

Generally we need three properties

1. Optimal substructure:
   - When an optimal solution can be constructed from optimal solutions of its subproblems. ["Grow" a solution without chaging the past]
2. Independent subproblems
   - Two subproblems are independent if they do not share the same resources (Alternatively, two subproblems are independent if the solution to one does not affect the solution to the other)
3. Non-overlapping subproblems
   - Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems

So when do we use a greedy approach?

Generally we need three properties

1. Optimal substructure:
   - When an optimal solution can be constructed from optimal solutions of its subproblems. [“Grow” a solution without chaging the past]
2. Independent subproblems
   - Two subproblems are independent if they do not share the same resources (Alternatively, two subproblems are independent if the solution to one does not affect the solution to the other)
3. Non-overlapping subproblems
   - Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems

Let’s try another problem, this time on graphs.

Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$,
  where each edge is a pair of vertices.

Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$,
  where each edge is a pair of vertices.

Representation?

Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$,
  where each edge is a pair of vertices.

Representation?

- Set notation

$$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$$

$$\begin{aligned} E(G) = \{&\{v_1, v_2\}, \{v_1, v_8\}, \{v_2, v_3\}, \{v_3, v_5\}, \{v_3, v_9\}, \\ &\{v_3, v_{10}\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_9\}, \{v_5, v_8\}, \\ &\{v_6, v_8\}, \{v_6, v_9\}, \{v_7, v_8\}, \{v_7, v_9\}, \{v_8, v_{10}\}, \\ &\{v_9, v_{10}\}\} \end{aligned}$$

Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$, where each edge is a pair of vertices.

Representation?

- Set notation

$$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$$

$$\begin{aligned} E(G) = \{&\{v_1, v_2\}, \{v_1, v_8\}, \{v_2, v_3\}, \{v_3, v_5\}, \{v_3, v_9\}, \\ &\{v_3, v_{10}\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_9\}, \{v_5, v_8\}, \\ &\{v_6, v_8\}, \{v_6, v_9\}, \{v_7, v_8\}, \{v_7, v_9\}, \{v_8, v_{10}\}, \\ &\{v_9, v_{10}\}\} \end{aligned}$$

- Adjacency list

v_1:	v_2, v_8	v_6:	v_4, v_8, v_9
v_2:	v_1, v_3	v_7:	v_8, v_9
v_3:	v_2, v_5, v_9, v_{10}	v_8:	$v_1, v_5, v_6, v_7, v_9, v_{10}$
v_4:	v_5, v_6, v_9	v_9:	$v_3, v_4, v_6, v_7, v_8, v_{10}$
v_5:	v_3, v_4, v_8	v_{10}:	v_3, v_8, v_9

Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$, where each edge is a pair of vertices.

Representation?

- Set notation

$$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$$

$$\begin{aligned} E(G) = \{&\{v_1, v_2\}, \{v_1, v_8\}, \{v_2, v_3\}, \{v_3, v_5\}, \{v_3, v_9\}, \\ &\{v_3, v_{10}\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_9\}, \{v_5, v_8\}, \\ &\{v_6, v_8\}, \{v_6, v_9\}, \{v_7, v_8\}, \{v_7, v_9\}, \{v_8, v_{10}\}, \\ &\{v_9, v_{10}\}\} \end{aligned}$$

- Adjacency list

v_1:	v_2, v_8	v_6:	v_4, v_8, v_9
v_2:	v_1, v_3	v_7:	v_8, v_9
v_3:	v_2, v_5, v_9, v_{10}	v_8:	$v_1, v_5, v_6, v_7, v_9, v_{10}$
v_4:	v_5, v_6, v_9	v_9:	$v_3, v_4, v_6, v_7, v_8, v_{10}$
v_5:	v_3, v_4, v_8	v_{10}:	v_3, v_8, v_9

- Adjacency matrix

$$\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{pmatrix}$$

Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$,
  where each edge is a pair of vertices.

Representation?

- Set notation

$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$

$E(G) = \{\{v_1, v_2\}, \{v_1, v_8\}, \{v_2, v_3\}, \{v_3, v_5\}, \{v_3, v_9\}, \{v_3, v_{10}\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_9\}, \{v_5, v_8\}, \{v_6, v_8\}, \{v_6, v_9\}, \{v_7, v_8\}, \{v_7, v_9\}, \{v_8, v_{10}\}, \{v_9, v_{10}\}\}$

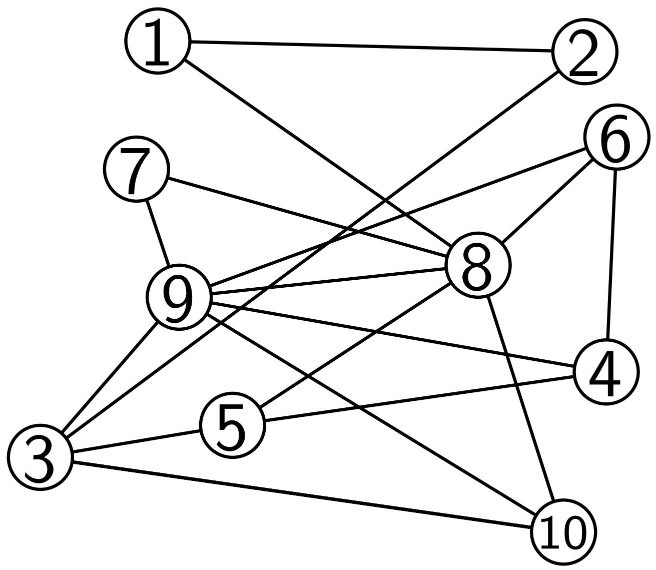
- Adjacency list

v_1: v_2, v_8
v_2: v_1, v_3
v_3: v_2, v_5, v_9, v_{10}
v_4: v_5, v_6, v_9
v_5: v_3, v_4, v_8
v_6: v_4, v_8, v_9
v_7: v_8, v_9
v_8: $v_1, v_5, v_6, v_7, v_9, v_{10}$
v_9: $v_3, v_4, v_6, v_7, v_8, v_{10}$
v_{10}: v_3, v_8, v_9

- Adjacency matrix

$$\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{pmatrix}$$

- Drawing



Graphs: (road) networks, relation diagrams, etc.

What is a graph?

- graph G
- vertex set $V(G) = \{v_1, v_2, \ldots, v_n\}$
- edge set $E(G) = \{e_1, e_2, \ldots, e_m\}$, where each edge is a pair of vertices.

Representation?

- Set notation

$$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}\}$$

$$\begin{aligned} E(G) = \{&\{v_1, v_2\}, \{v_1, v_8\}, \{v_2, v_3\}, \{v_3, v_5\}, \{v_3, v_9\}, \\ &\{v_3, v_{10}\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_4, v_9\}, \{v_5, v_8\}, \\ &\{v_6, v_8\}, \{v_6, v_9\}, \{v_7, v_8\}, \{v_7, v_9\}, \{v_8, v_{10}\}, \\ &\{v_9, v_{10}\}\} \end{aligned}$$

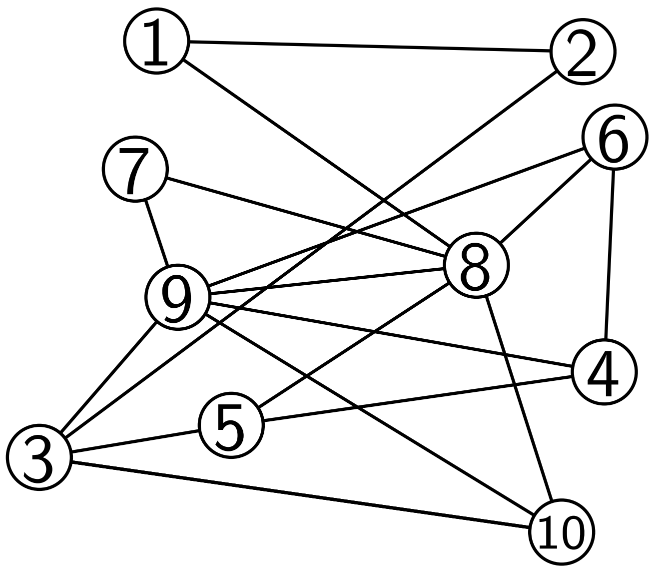
- Adjacency list

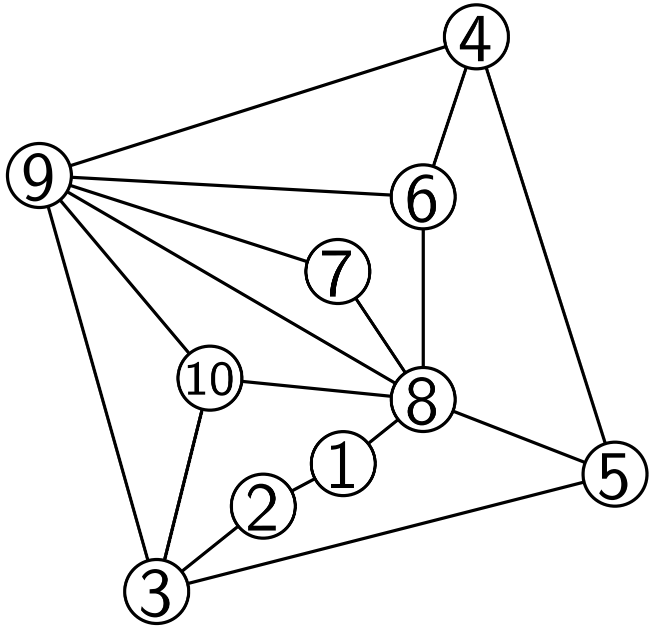
v_1:	v_2, v_8	v_6:	v_4, v_8, v_9
v_2:	v_1, v_3	v_7:	v_8, v_9
v_3:	v_2, v_5, v_9, v_{10}	v_8:	$v_1, v_5, v_6, v_7, v_9, v_{10}$
v_4:	v_5, v_6, v_9	v_9:	$v_3, v_4, v_6, v_7, v_8, v_{10}$
v_5:	v_3, v_4, v_8	v_{10}:	v_3, v_8, v_9

- Adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

- Drawing





Problem: Efficient Sub-Network Construction

- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

Problem: Efficient Sub-Network Construction

- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

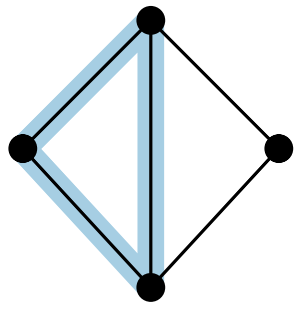
Represent road network as a **graph**: road segments are the edges, locations are the vertices.

Problem: Efficient Sub-Network Construction

- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

Represent road network as a **graph**: road segments are the edges, locations are the vertices.

Should our fibre network contain **cycles** ?



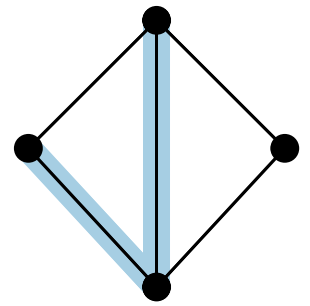
Problem: Efficient Sub-Network Construction

- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

Represent road network as a **graph**: road segments are the edges, locations are the vertices.

Should our fibre network contain **cycles** ?

No, the solution should be a **tree**.



Problem: Efficient Sub-Network Construction

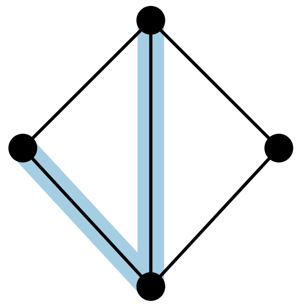
- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

Represent road network as a **graph**: road segments are the edges, locations are the vertices.

Should our fibre network contain **cycles** ?

No, the solution should be a **tree**.

How do we ensure all locations have service?



Problem: Efficient Sub-Network Construction

- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

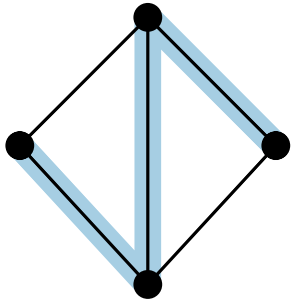
Represent road network as a **graph**: road segments are the edges, locations are the vertices.

Should our fibre network contain **cycles** ?

No, the solution should be a **tree**.

How do we ensure all locations have service?

The solution needs to **connect** all locations.



Problem: Efficient Sub-Network Construction

- Need to build a fibre optic network for my kingdom connecting all important locations.
- For ease of planning, cables are built along the road network.
  - The roads are given as segments connecting important locations, and
  - each segment has a cost to have a cable built there.
- I want to connect my kingdom at minimum total cost

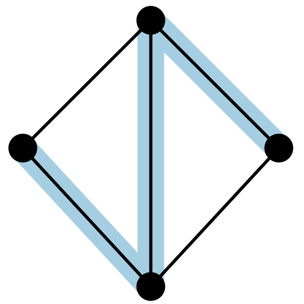
Represent road network as a **graph**: road segments are the edges, locations are the vertices.

Should our fibre network contain **cycles** ?

No, the solution should be a **tree**.

How do we ensure all locations have service?

The solution needs to **connect** all locations.



MINIMUM SPANNING TREE (MST)

Input: An n-vertex graph $G = (V, E)$, and edge costs $c : E \to \mathbb{R}$.

Goal: Return a tree T that *spans* G, i.e., $V(T) = V$ and $E(T) \subset E$, where the total edge cost, $\sum_{e \in E(T)} c(e)$, is minimized.

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

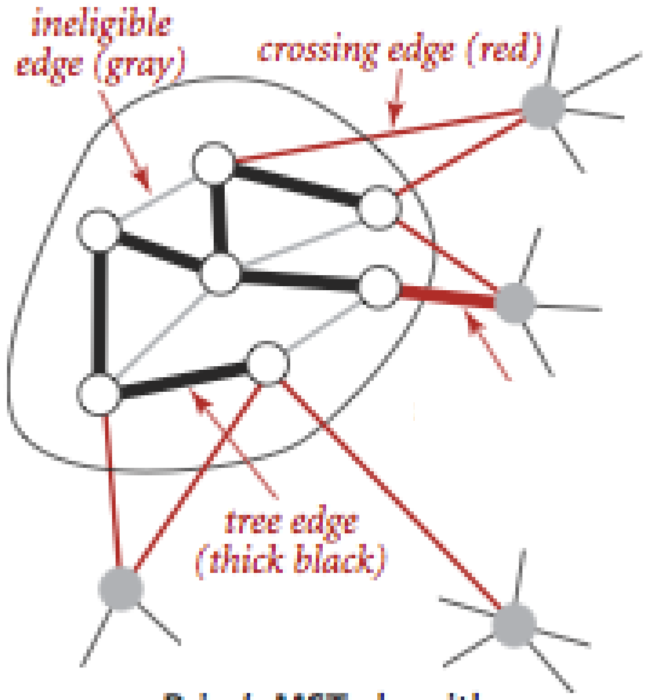
What are our partial solutions?

How do we "grow" a partial solution greedily?

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

What are our partial solutions? Trees in our graph.

How do we "grow" a partial solution greedily?



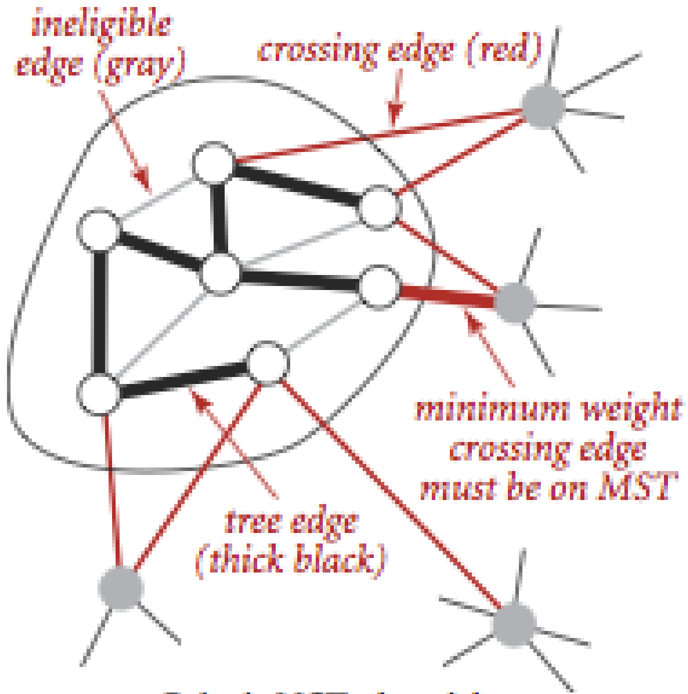
Prim's MST algorithm

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

What are our partial solutions? Trees in our graph.

How do we "grow" a partial solution greedily?

> Pick a **cheapest** (minimum cost) edge
> to add a new vertex to the tree.



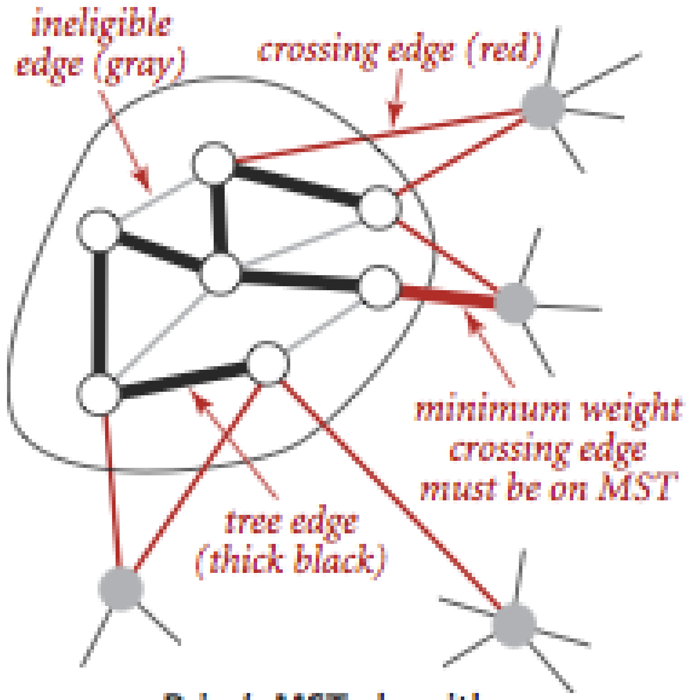
Prim's MST algorithm

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

What are our partial solutions? Trees in our graph.

How do we "grow" a partial solution greedily?

Pick a **cheapest** (minimum cost) edge
to add a new vertex to the tree.



Prim's MST algorithm

How can we do this efficiently?

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

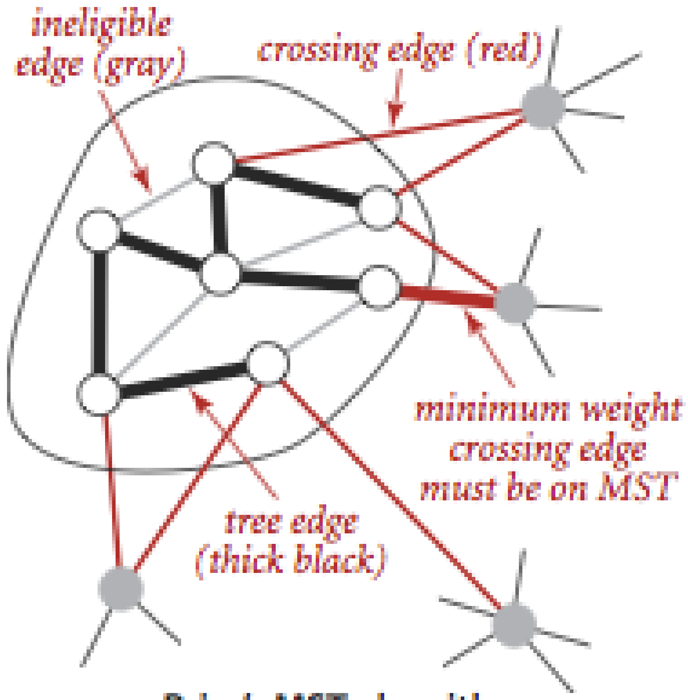
What are our partial solutions? Trees in our graph.

How do we "grow" a partial solution greedily?

Pick a **cheapest** (minimum cost) edge
to add a new vertex to the tree.

How can we do this efficiently?

Data structures to the rescue!



Prim's MST algorithm

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

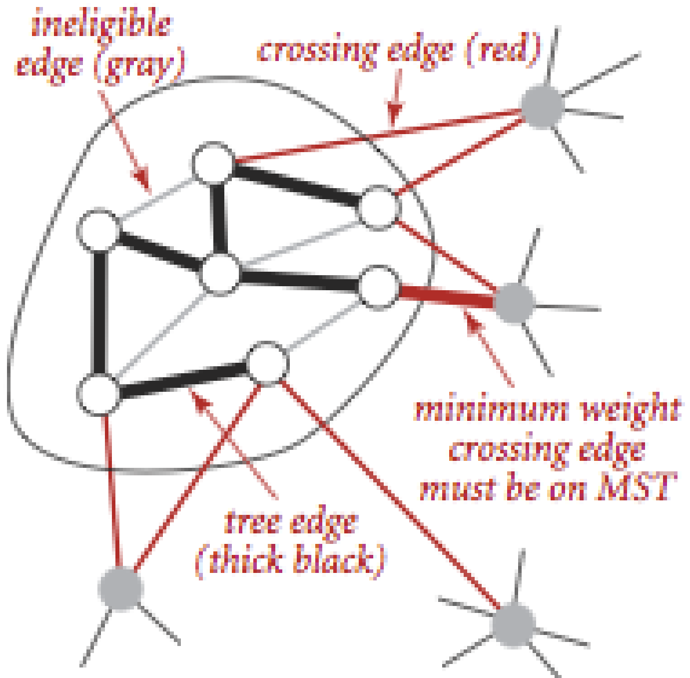
What are our partial solutions? Trees in our graph.

How do we "grow" a partial solution greedily?

Pick a **cheapest** (minimum cost) edge
to add a new vertex to the tree.

How can we do this efficiently?

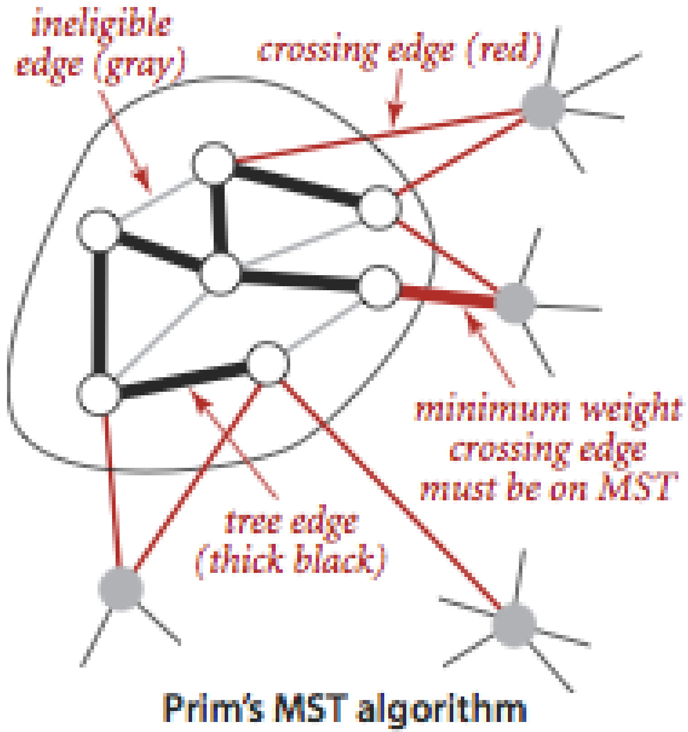
Data structures to the rescue!
Maintain **eligible** edges to be added in a **heap**.



Prim's MST algorithm

Greedy Algorithm for MST (Prim), [GT §15, CLRS §23]

What are our partial solutions? Trees in our graph.



How do we "grow" a partial solution greedily?

Pick a **cheapest** (minimum cost) edge
to add a new vertex to the tree.

How can we do this efficiently?

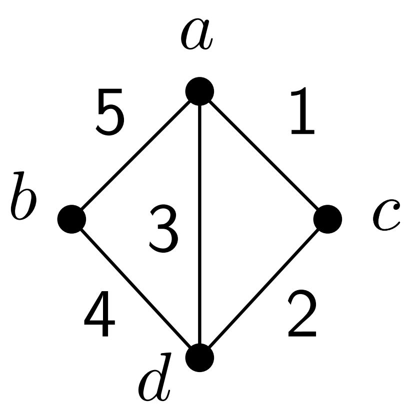
Data structures to the rescue!
Maintain **eligible** edges to be added in a **heap**.

Exercises:

- Formalize this algorithm in pseudocode.
- Analyze the runtime. Think carefully about how you should represent the graph.

But wait, does that really work?

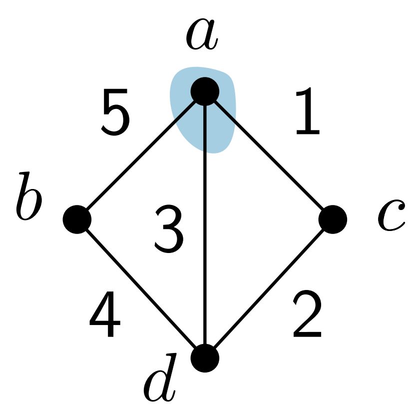
Quick sanity check:



iteration:	
Vertices in our Tree	
Edges (by cost) in the Heap:	

But wait, does that really work?

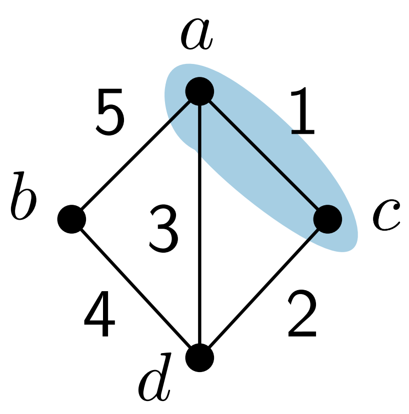
Quick sanity check:



iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$			
Edges (by cost) in the Heap:	$\{1, 3, 5\}$			

But wait, does that really work?

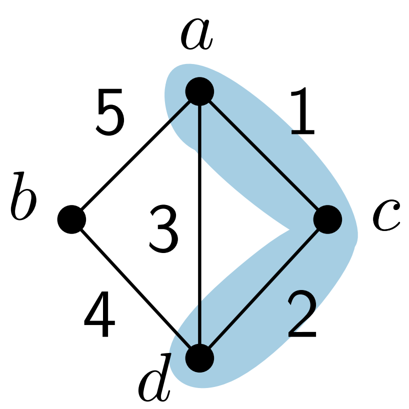
Quick sanity check:



iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$		
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$		

But wait, does that really work?

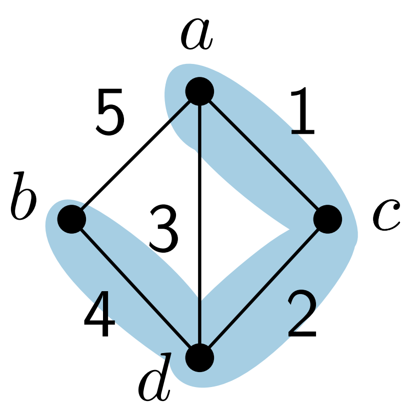
Quick sanity check:



iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$	$\{4, 5\}$	

But wait, does that really work?

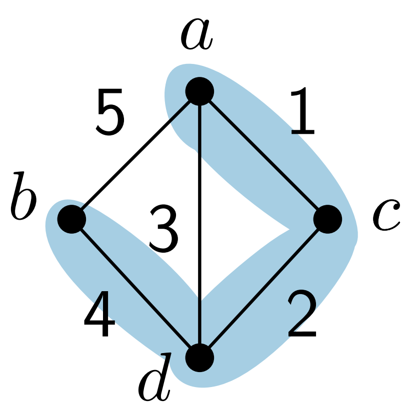
Quick sanity check:



iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	$\{a, c, d, b\}$
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$	$\{4, 5\}$	$\{\}$

But wait, does that really work?

Quick sanity check:

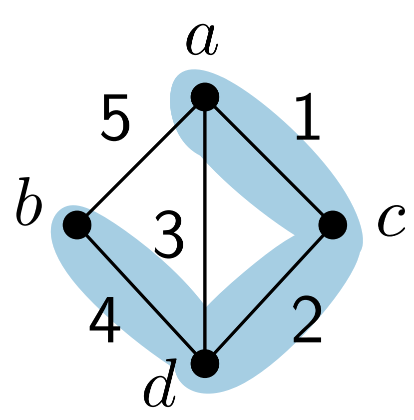


iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	$\{a, c, d, b\}$
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$	$\{4, 5\}$	$\{\}$

Looks good for this example, but can we prove it in general?

But wait, does that really work?

Quick sanity check:



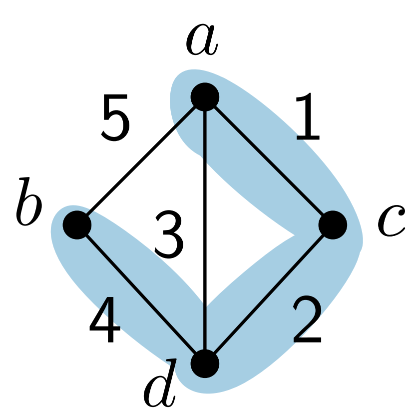
iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	$\{a, c, d, b\}$
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$	$\{4, 5\}$	$\{\}$

Looks good for this example, but can we prove it in general?

Proof: (Assuming all weights are different)
Let T be a spanning tree returned by the greedy algorithm, and let T' be a minimum spanning tree.

But wait, does that really work?

Quick sanity check:

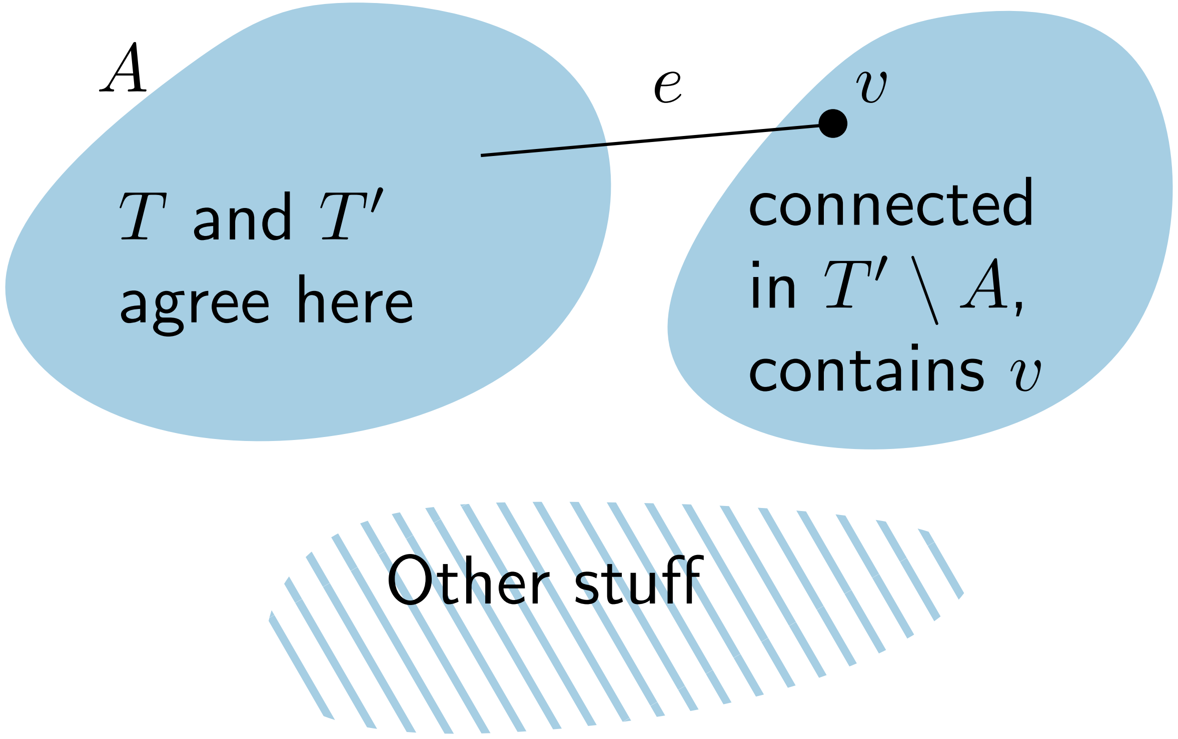


iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	$\{a, c, d, b\}$
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$	$\{4, 5\}$	$\{\}$

Looks good for this example, but can we prove it in general?

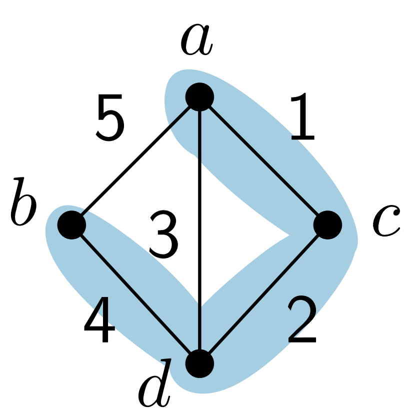
Proof: (Assuming all weights are different)
Let T be a spanning tree returned by the greedy algorithm, and let T' be a minimum spanning tree.

Let $e = uv$ be the first edge added to T by the greedy algorithm that is not an edge of T'.



But wait, does that really work?

Quick sanity check:



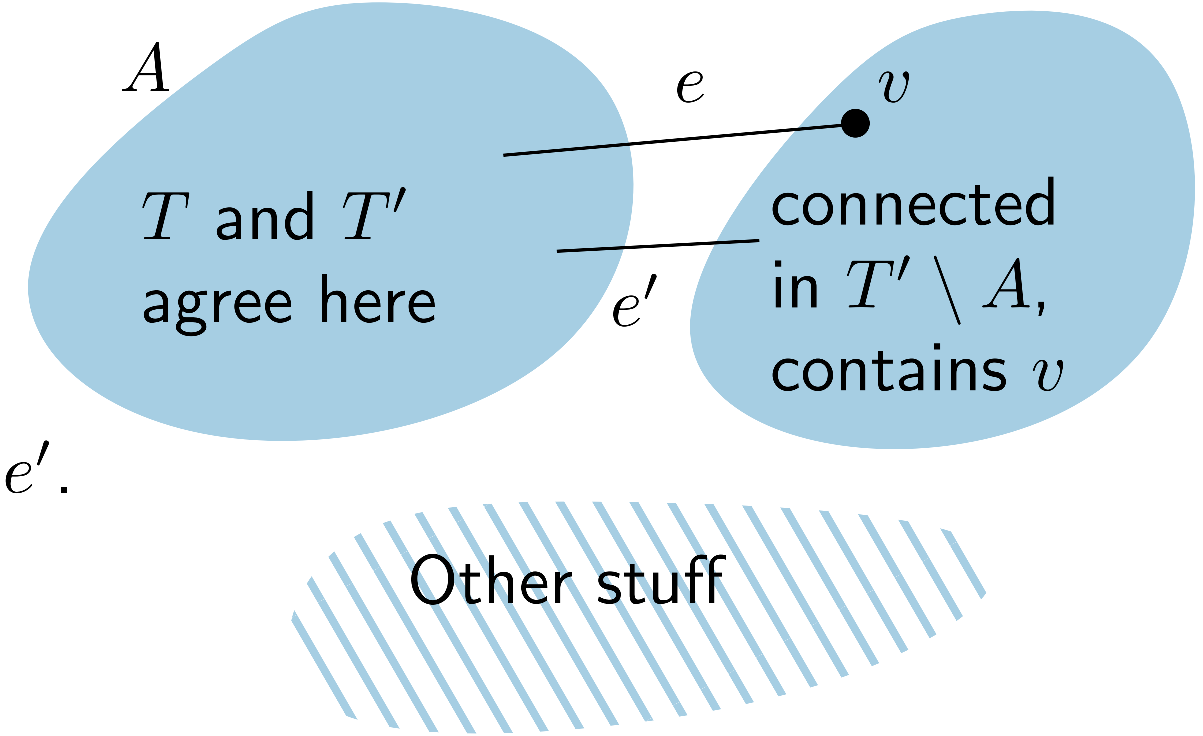
iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a, c\}$	$\{a, c, d\}$	$\{a, c, d, b\}$
Edges (by cost) in the Heap:	$\{1, 3, 5\}$	$\{2, 3, 5\}$	$\{4, 5\}$	$\{\}$

Looks good for this example, but can we prove it in general?

Proof: (Assuming all weights are different)
Let T be a spanning tree returned by the greedy algorithm, and let T' be a minimum spanning tree.

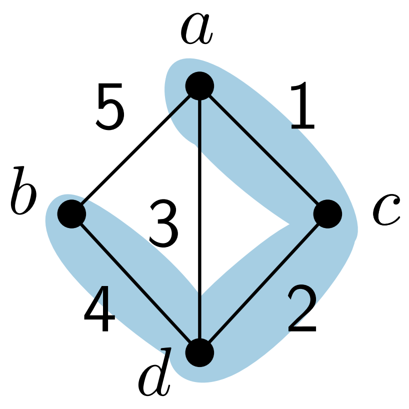
Let $e = uv$ be the first edge added to T by the greedy algorithm that is not an edge of T'.

T' also connects the left blob with the right, say by e'.
But greedy took e instead of e', so $c(e) < c(e')$.



But wait, does that really work?

Quick sanity check:

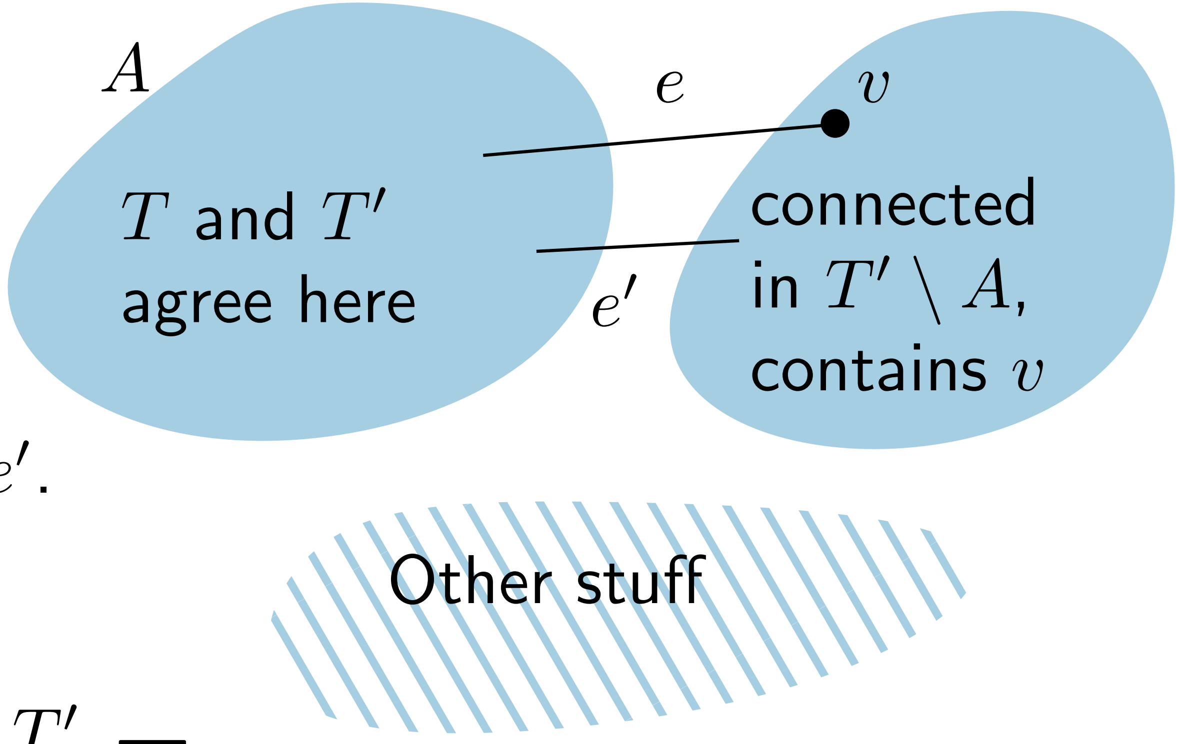


iteration:	1	2	3	4
Vertices in our Tree	$\{a\}$	$\{a,c\}$	$\{a,c,d\}$	$\{a,c,d,b\}$
Edges (by cost) in the Heap:	$\{1,3,5\}$	$\{2,3,5\}$	$\{4,5\}$	$\{\}$

Looks good for this example, but can we prove it in general?

Proof: (Assuming all weights are different)
Let T be a spanning tree returned by the greedy algorithm, and let T' be a minimum spanning tree.

Let $e = uv$ be the first edge added to T by the greedy algorithm that is not an edge of T'.



T' also connects the left blob with the right, say by e'.
But greedy took e instead of e', so $c(e) < c(e')$.
Let T'' be the result of swapping e' for e in T'.
T'' is a spanning tree contradicting the minimality of T'. ∎

But what about equal weights? (add example)

To show that the greedy T is indeed optimal, we **imagine** the following scenario.

But what about equal weights? (add example)

To show that the greedy T is indeed optimal, we **imagine** the following scenario.

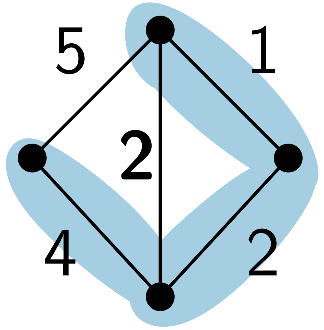
Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

But what about equal weights? (add example)

To show that the greedy T is indeed optimal, we **imagine** the following scenario.

Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

Example from before, now the “middle” edge has cost 2.

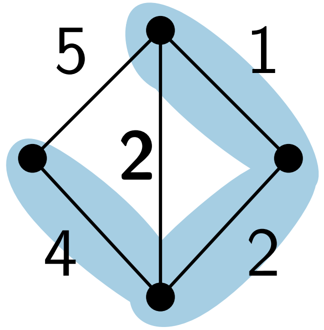


But what about equal weights? (add example)

To show that the greedy T is indeed optimal, we **imagine** the following scenario.

Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

$n = 4,\ \Delta = 1.$



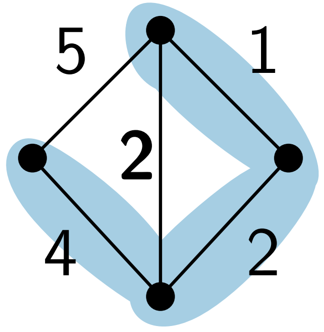
But what about equal weights? (add example)

To show that the greedy T is indeed optimal, we **imagine** the following scenario.

Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

We now make all the weights unique as follows.

$n = 4$, $\Delta = 1$.



- List out the edges in T in the order they were added to T as $e_1, \ldots, e_{n-1}$, and the remaining edges $e_n, \ldots, e_{|E|}$ in an arbitrary order.
- New cost function c' where $c'(e_i) = c(e_i) + i \cdot \epsilon$.

But what about equal weights? (add example)

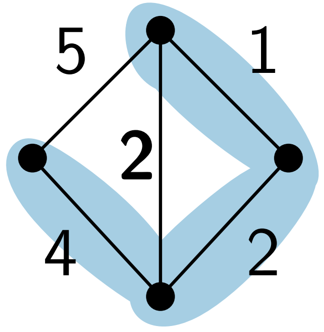
To show that the greedy T is indeed optimal, we **imagine** the following scenario.

Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

We now make all the weights unique as follows.

- List out the edges in T in the order they were added to T as $e_1, \ldots, e_{n-1}$, and the remaining edges $e_n, \ldots, e_{|E|}$ in an arbitrary order.
- New cost function c' where $c'(e_i) = c(e_i) + i \cdot \epsilon$.

$n = 4$, $\Delta = 1$.



Now the costs are unique, but how does this help us to prove our tree is optimal?

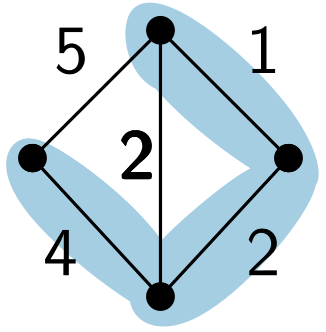
But what about equal weights? (add example)

To show that the greedy T is indeed optimal, we **imagine** the following scenario.

Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

We now make all the weights unique as follows.

$n = 4$, $\Delta = 1$.



- List out the edges in T in the order they were added to T as $e_1, \ldots, e_{n-1}$, and the remaining edges $e_n, \ldots, e_{|E|}$ in an arbitrary order.
- New cost function c' where $c'(e_i) = c(e_i) + i \cdot \epsilon$.

Now the costs are unique, but how does this help us to prove our tree is optimal?

We can now prove that the cost of the greedy tree T according to c' is less than:

$$\Delta + \text{the cost of the optimal tree } T' \text{ according to the original costs.}$$

But what about equal weights? (add example)

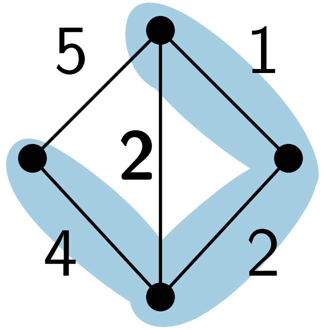
To show that the greedy T is indeed optimal, we **imagine** the following scenario.

Let ϵ be a very small value, e.g., something smaller than $\frac{1}{n^3} \cdot \Delta$ where Δ is the smallest difference between any two **distinct** edge costs.

We now make all the weights unique as follows.

- List out the edges in T in the order they were added to T as $e_1, \ldots, e_{n-1}$, and the remaining edges $e_n, \ldots, e_{|E|}$ in an arbitrary order.
- New cost function c' where $c'(e_i) = c(e_i) + i \cdot \epsilon$.

$n = 4$, $\Delta = 1$.



$(1, \quad 2, \quad \mathbf{2}, \quad 4, \quad 5) \rightarrow$

$(1+\frac{1}{4^3}, \; 2+\frac{2}{4^3}, \; \mathbf{2}+\frac{3}{4^3}, \; 4+\frac{4}{4^3}, \; 5+\frac{5}{4^3})$

Now the costs are unique, but how does this help us to prove our tree is optimal?

We can now prove that the cost of the greedy tree T according to c' is less than:

$$\Delta + \text{the cost of the optimal tree } T' \text{ according to the original costs.}$$

Exercise: Try it :) . . . also proven in the books.

Problem: Scheduling

- You are an event coordinator for a convention
- You have a list of talks $T_1, \ldots, T_n$, each T_i has a start and finish time, (s_i, f_i), $s_i < f_i$.
- The venue has an adequate number of rooms to host all talks in any manner
  - They charge an entire day for each room used, even it is only used for one short talk.
- You would like to create a schedule that
  - Minimizes the number of rooms used and tells you what talk goes in which room
- The book calls this *task scheduling*. It is also known as *interval scheduling*.

Problem: Scheduling

- You are an event coordinator for a convention
- You have a list of talks $T_1, \ldots, T_n$, each T_i has a start and finish time, (s_i, f_i), $s_i < f_i$.
- The venue has an adequate number of rooms to host all talks in any manner
  - They charge an entire day for each room used, even it is only used for one short talk.
- You would like to create a schedule that
  - Minimizes the number of rooms used and tells you what talk goes in which room
- The book calls this *task scheduling*. It is also known as *interval scheduling*.

What is a schedule?

When is a schedule **feasible**?

Problem: Scheduling

- You are an event coordinator for a convention
- You have a list of talks $T_1, \ldots, T_n$, each T_i has a start and finish time, (s_i, f_i), $s_i < f_i$.
- The venue has an adequate number of rooms to host all talks in any manner
  - They charge an entire day for each room used, even it is only used for one short talk.
- You would like to create a schedule that
  - Minimizes the number of rooms used and tells you what talk goes in which room
- The book calls this *task scheduling*. It is also known as *interval scheduling*.

What is a schedule? Function S mapping talks to rooms (e.g., represented numbers).

When is a schedule **feasible**?

Problem: Scheduling

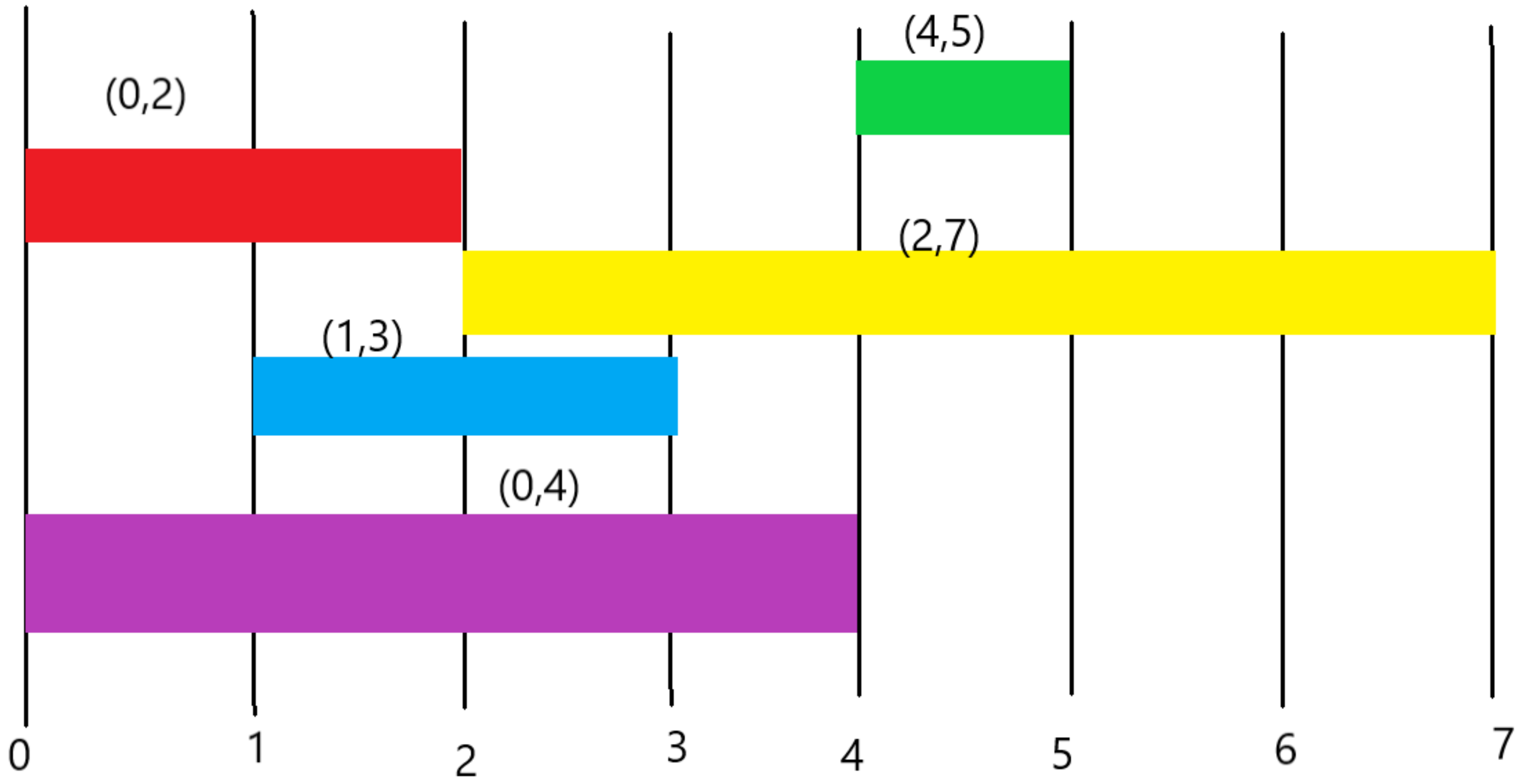
- You are an event coordinator for a convention
- You have a list of talks $T_1, \ldots, T_n$, each T_i has a start and finish time, (s_i, f_i), $s_i < f_i$.
- The venue has an adequate number of rooms to host all talks in any manner
  - They charge an entire day for each room used, even it is only used for one short talk.
- You would like to create a schedule that
  - Minimizes the number of rooms used and tells you what talk goes in which room
- The book calls this *task scheduling*. It is also known as *interval scheduling*.

What is a schedule? Function S mapping talks to rooms (e.g., represented numbers).

When is a schedule **feasible**? No **conflicting** talks are mapped to the same room.
That is, if $S(T_i) = S(T_j)$ then either $f_i \leq s_j$ or $f_j \leq s_i$.

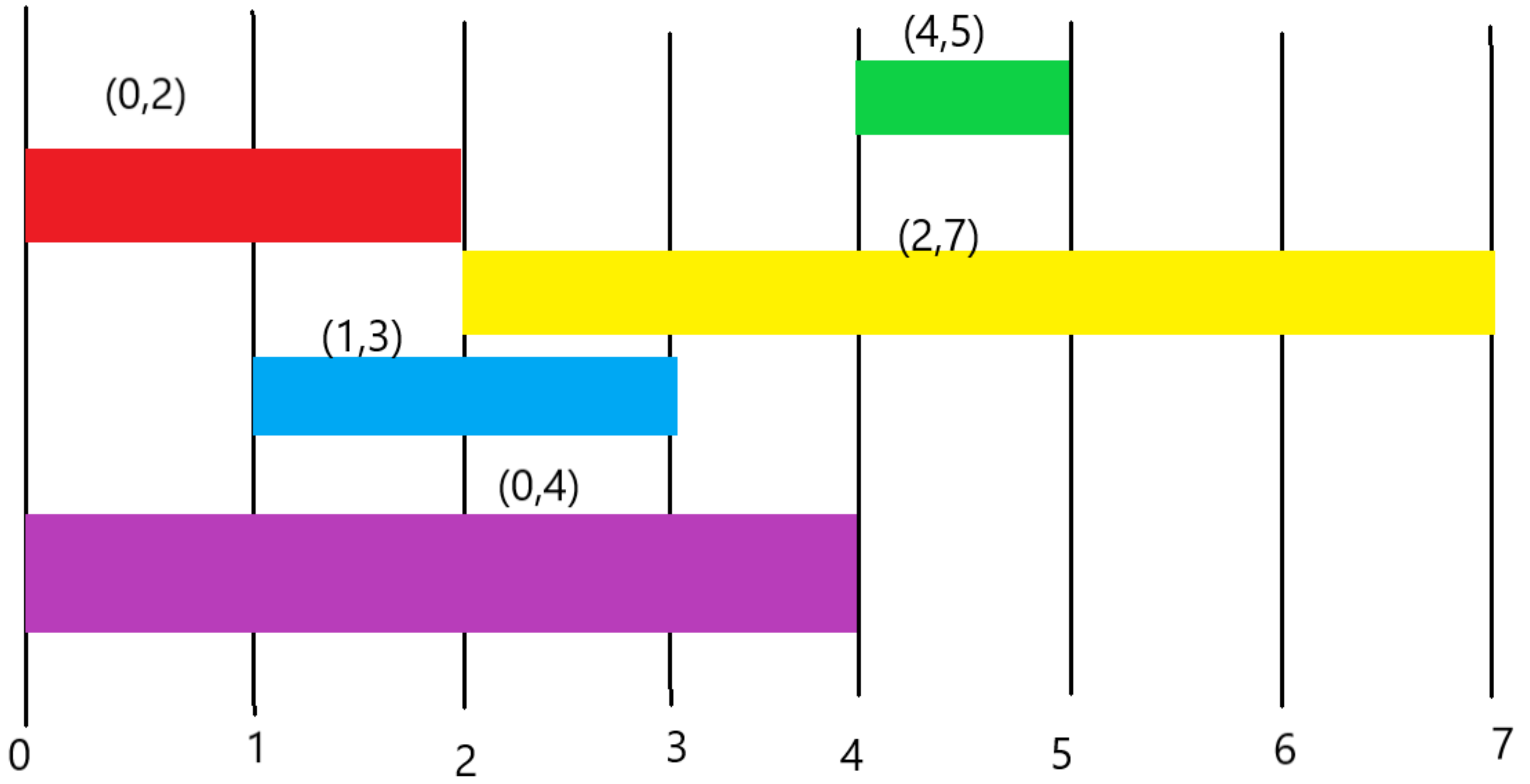
Scheduling, an example

Here is a graphical example of a set of talks
$\{(4,5),(0,2),(2,7),(1,3),(0,4)\}$



Scheduling, an example

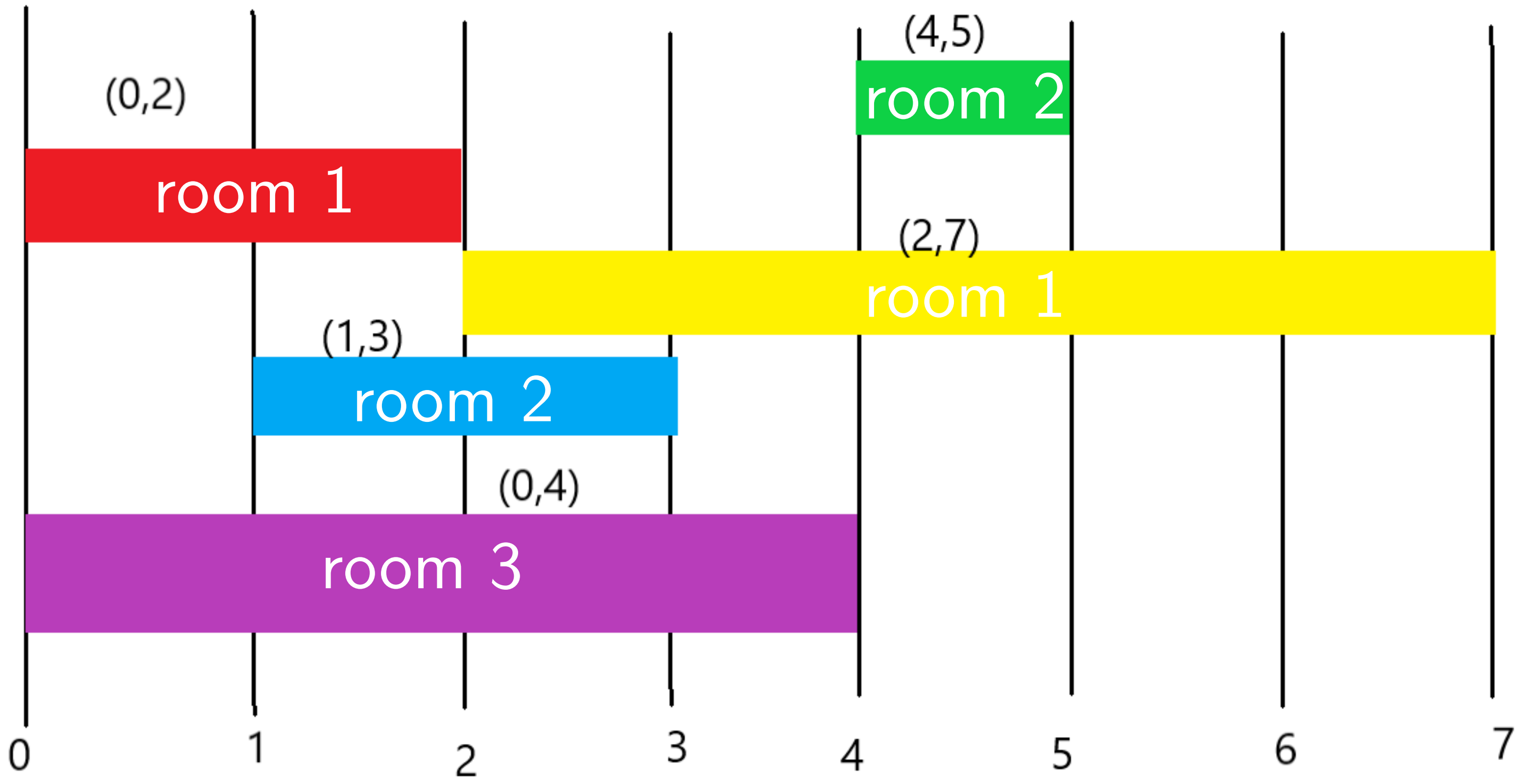
Here is a graphical example of a set of talks
$\{(4,5),(0,2),(2,7),(1,3),(0,4)\}$



How many rooms do we need?

Scheduling, an example

Here is a graphical example of a set of talks
$\{(4,5),(0,2),(2,7),(1,3),(0,4)\}$



How many rooms do we need?

Greedy Approach

Take the longest talk first, then the next longest, etc. ?

Regardless of our greedy measure, at each point the next talk scheduled either has to

- Be scheduled in room in use, to do so it cannot conflict with another talk in that room.
- Be scheduled in a new room, otherwise

Greedy Approach

Take the longest talk first, then the next longest, etc. ?

Regardless of our greedy measure, at each point the next talk scheduled either has to

- Be scheduled in room in use, to do so it cannot conflict with another talk in that room.
- Be scheduled in a new room, otherwise

Taking the longest first could leave space for the shorter talks to “fill in” the unused spots
This sort of reasoning is common when coming up with greedy algorithms

- Pick a likely-sounding rule
- See how it does

Approach #1: longest first

Not optimal :(

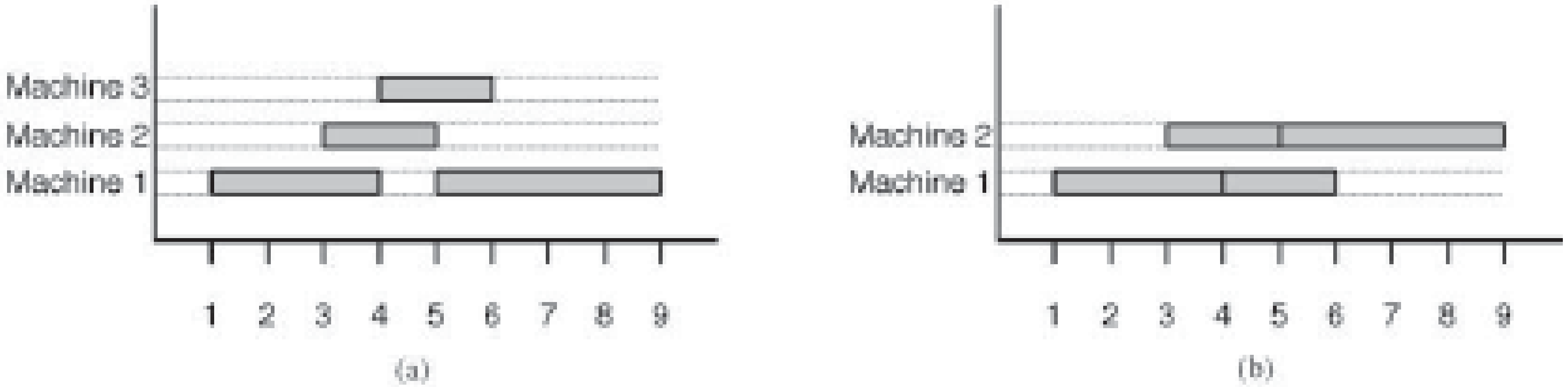


Figure 10.5: Why the longest-first strategy doesn't work, for the pairs of start and finish times in the set $\{(1,4),(5,9),(3,5),(4,6)\}$; (a) the solution chosen by the longest-first strategy; (b) the optimal solution. Note that the longest-first strategy uses three machines, whereas the optimal strategy uses only two.

Next Approach?

Other measures . . . use the start or finish times to greedily fill the schedule?

- Earliest to start first, then next earliest start time, etc.
- Earliest to finish first, then next earliest to finish, etc.
- Latest to finish first, then next latest, etc.
- . . .

Idea: scheduling the earliest talk first fills the rooms from earliest to latest.

Next Approach?

Other measures . . . use the start or finish times to greedily fill the schedule?

- Earliest to start first, then next earliest start time, etc.
- Earliest to finish first, then next earliest to finish, etc.
- Latest to finish first, then next latest, etc.
- . . .

Idea: scheduling the earliest talk first fills the rooms from earliest to latest.

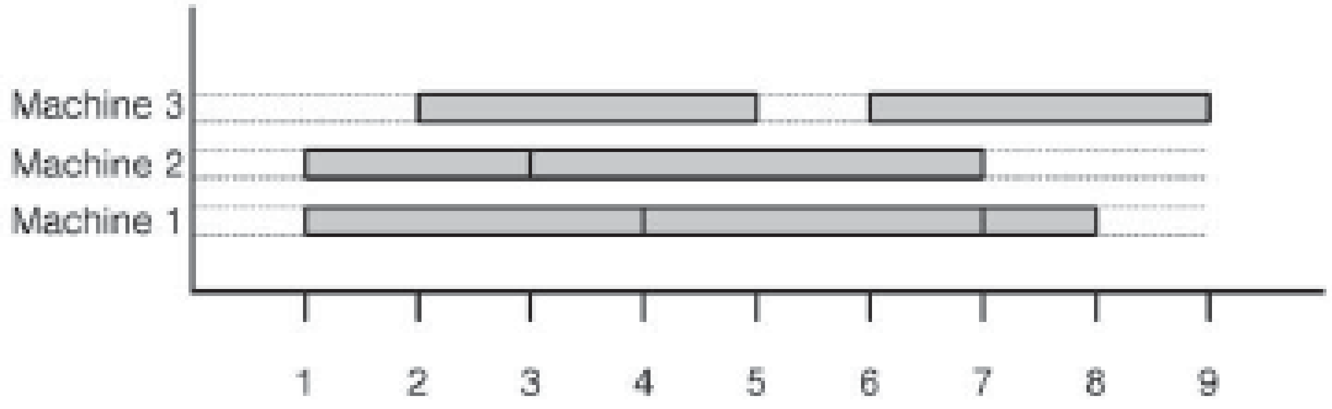


Figure 10.7: An example solution produced by the greedy algorithm based on considering tasks by increasing start times.

Next Approach?

Other measures . . . use the start or finish times to greedily fill the schedule?

- Earliest to start first, then next earliest start time, etc.
- Earliest to finish first, then next earliest to finish, etc.
- Latest to finish first, then next latest, etc.
- . . .

Idea: scheduling the earliest talk first fills the rooms from earliest to latest.

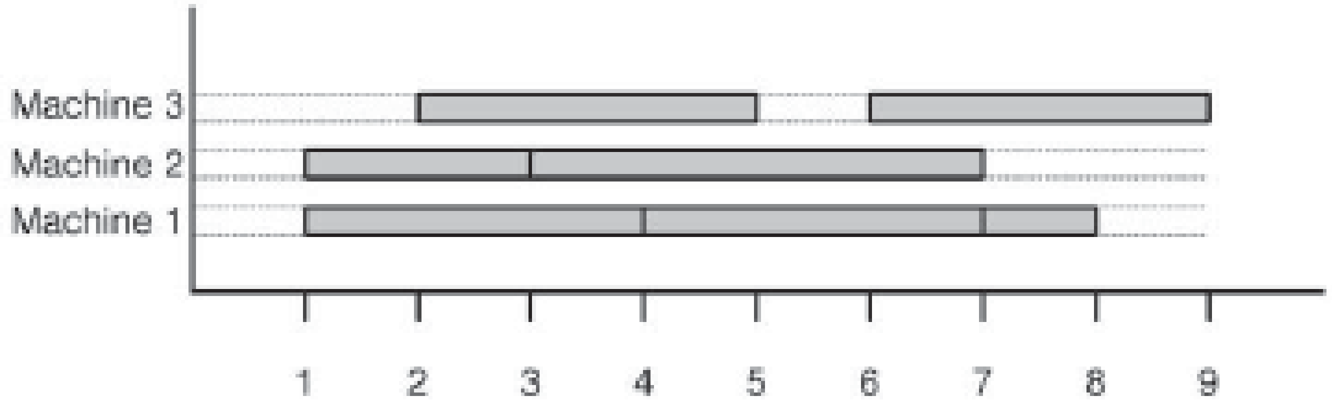


Figure 10.7: An example solution produced by the greedy algorithm based on considering tasks by increasing start times.

Looks promising!

Next Approach?

Other measures ... use the start or finish times to greedily fill the schedule?

- Earliest to start first, then next earliest start time, etc.
- Earliest to finish first, then next earliest to finish, etc.
- Latest to finish first, then next latest, etc.
- ...

Idea: scheduling the earliest talk first fills the rooms from earliest to latest.

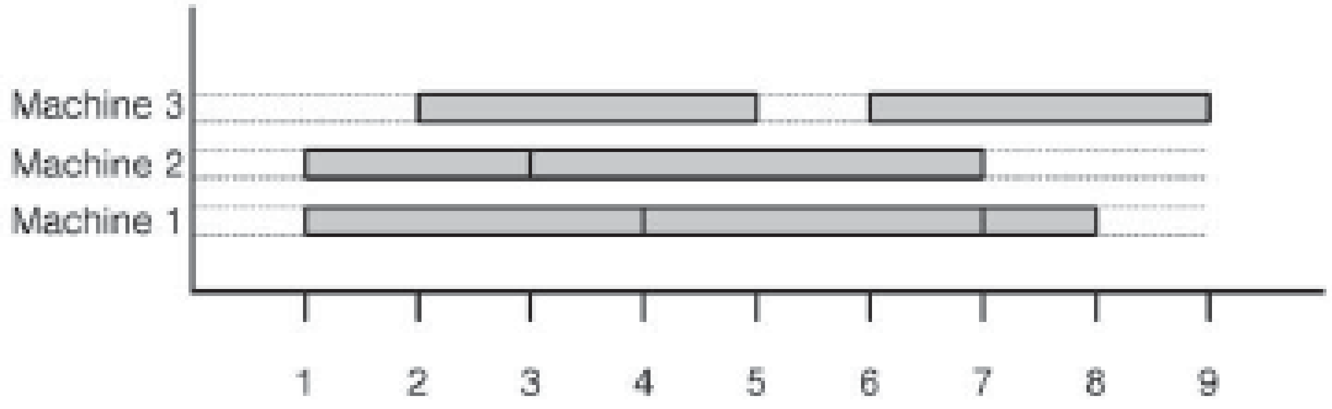


Figure 10.7: An example solution produced by the greedy algorithm based on considering tasks by increasing start times.

Looks promising!

Exercise: write out the pseudocode for this algorithm and analyze the runtime.

But does it really always work?

(The book also provides a proof of this)

Let S be a schedule given by the greedy approach of scheduling talks by earliest start time.

Let S' be an optimal schedule, that is, using the fewest rooms possible.

But does it really always work?

(The book also provides a proof of this)

Let S be a schedule given by the greedy approach of scheduling talks by earliest start time.
Let S' be an optimal schedule, that is, using the fewest rooms possible.

Suppose that S uses k rooms and S' uses ℓ rooms. For a contradiction, assume $k > \ell$.

Let T_i be the talk that causes S to start using the room $\ell + 1$. Thus, T_i cannot be added to any of the ℓ active rooms.

But does it really always work?

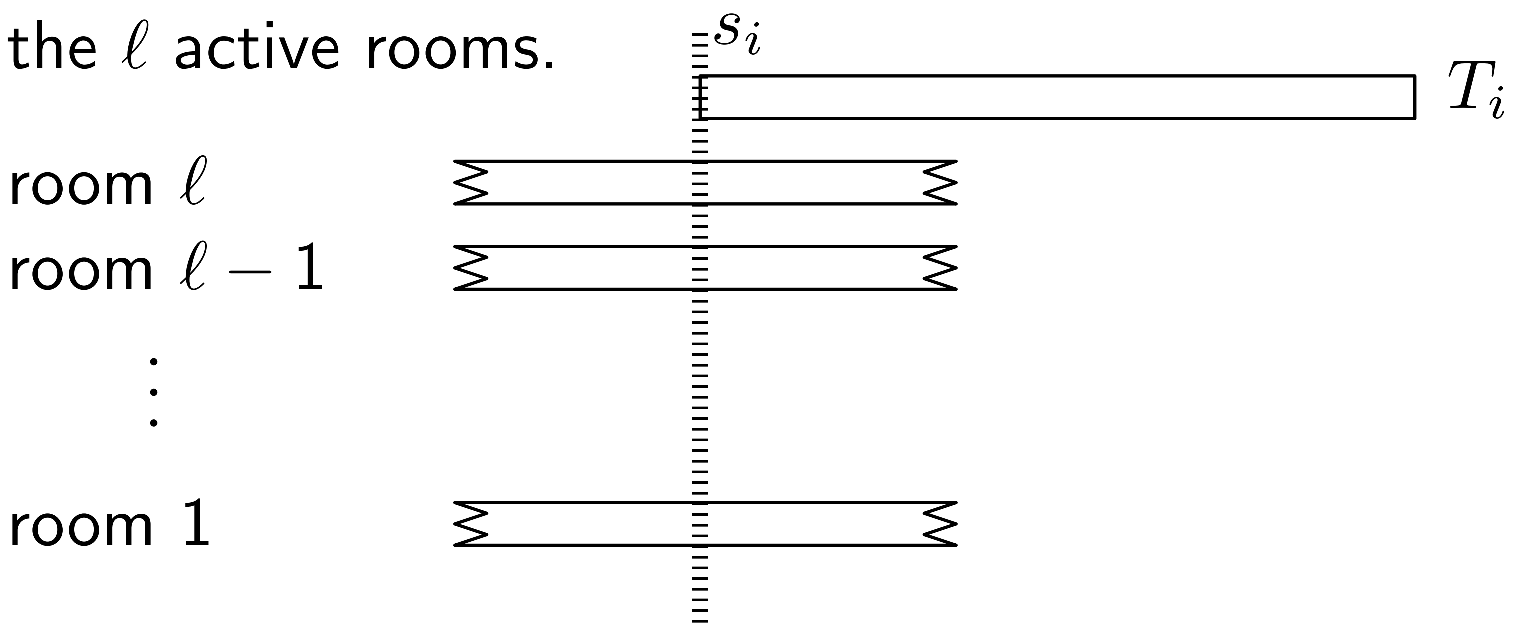
(The book also provides a proof of this)

Let S be a schedule given by the greedy approach of scheduling talks by earliest start time.

Let S' be an optimal schedule, that is, using the fewest rooms possible.

Suppose that S uses k rooms and S' uses ℓ rooms. For a contradiction, assume $k > \ell$.

Let T_i be the talk that causes S to start using the room $\ell + 1$. Thus, T_i cannot be added to any of the ℓ active rooms.



But does it really always work?

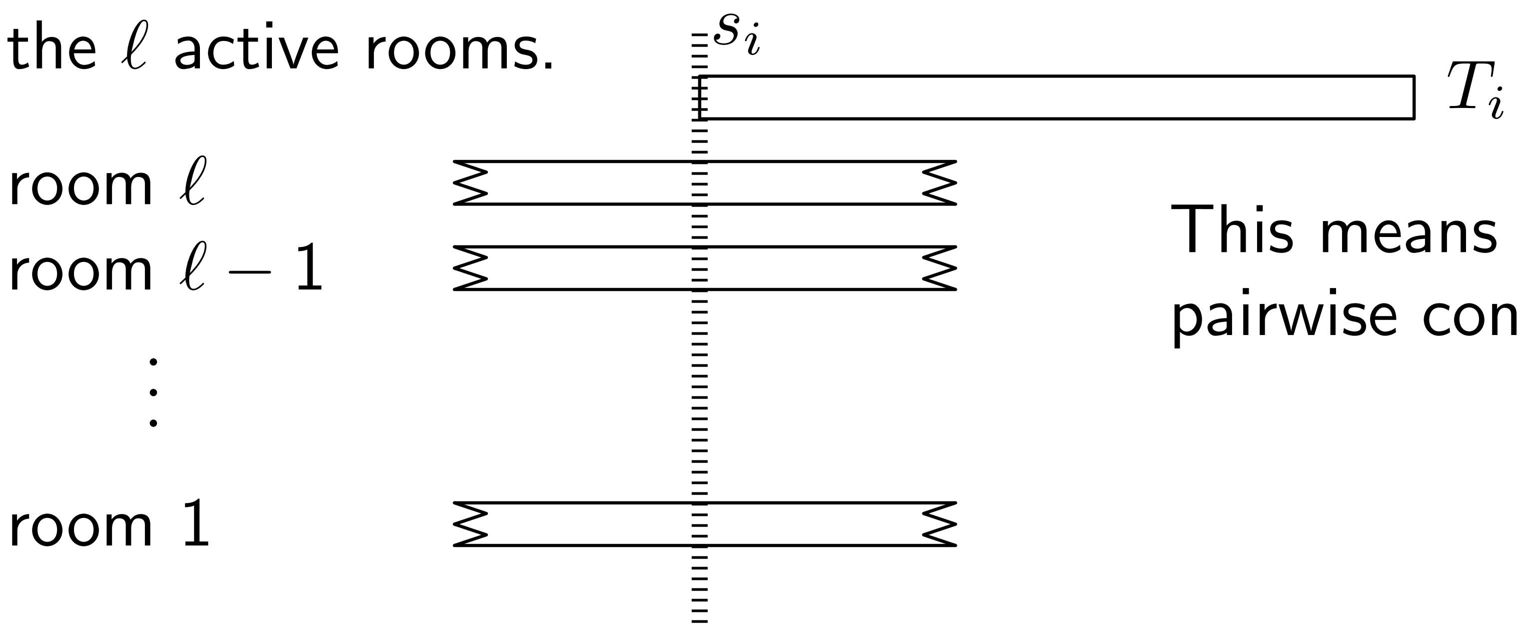
(The book also provides a proof of this)

Let S be a schedule given by the greedy approach of scheduling talks by earliest start time.

Let S' be an optimal schedule, that is, using the fewest rooms possible.

Suppose that S uses k rooms and S' uses ℓ rooms. For a contradiction, assume $k > \ell$.

Let T_i be the talk that causes S to start using the room $\ell + 1$. Thus, T_i cannot be added to any of the ℓ active rooms.



This means we have $\ell + 1$ talks that pairwise conflict.

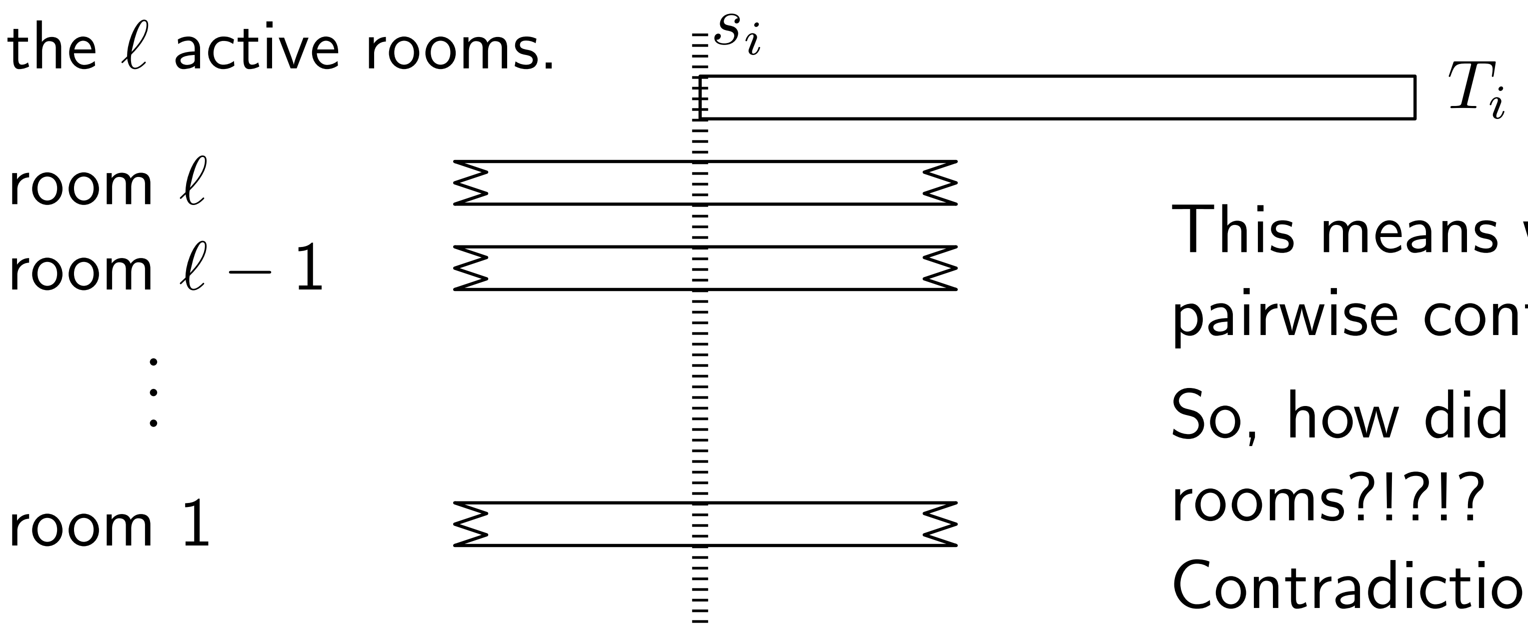
But does it really always work?

(The book also provides a proof of this)

Let S be a schedule given by the greedy approach of scheduling talks by earliest start time.
Let S' be an optimal schedule, that is, using the fewest rooms possible.

Suppose that S uses k rooms and S' uses ℓ rooms. For a contradiction, assume $k > \ell$.

Let T_i be the talk that causes S to start using the room $\ell + 1$. Thus, T_i cannot be added to any of the ℓ active rooms.



This means we have $\ell + 1$ talks that pairwise conflict.

So, how did S' schedule these in ℓ rooms?!?!?

Contradiction! :)

But does it really always work?

(The book also provides a proof of this)

Let S be a schedule given by the greedy approach of scheduling talks by earliest start time.
Let S' be an optimal schedule, that is, using the fewest rooms possible.

Suppose that S uses k rooms and S' uses ℓ rooms. For a contradiction, assume $k > \ell$.

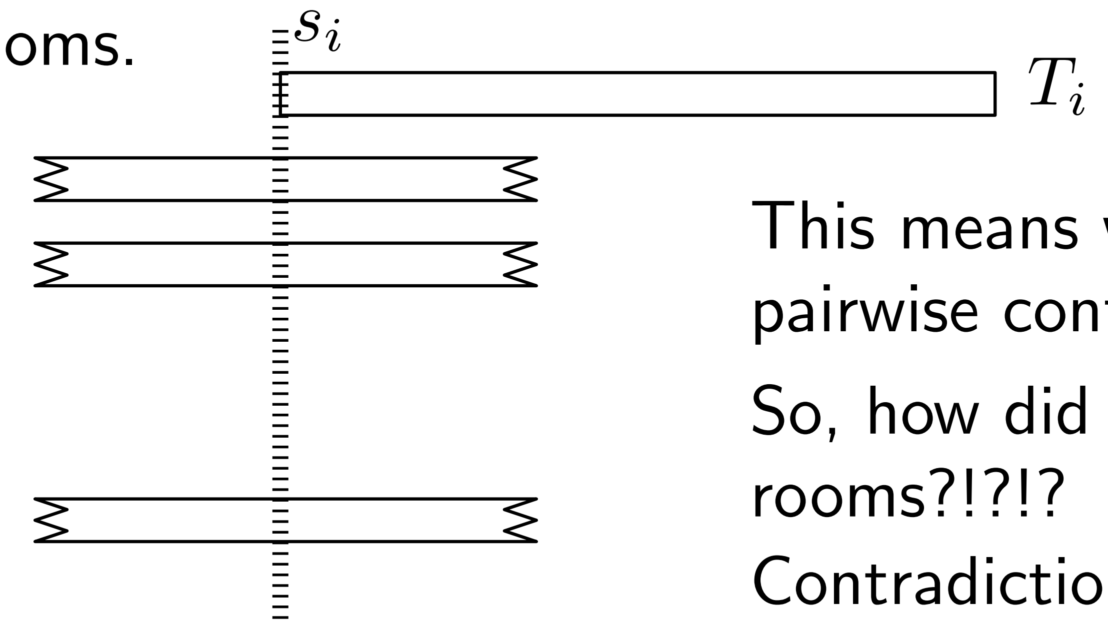
Let T_i be the talk that causes S to start using the room $\ell + 1$. Thus, T_i cannot be added to any of the ℓ active rooms.

room ℓ

room $\ell - 1$

$\vdots$

room 1



This means we have $\ell + 1$ talks that pairwise conflict.

So, how did S' schedule these in ℓ rooms?!?!?

Contradiction! :)

Thus, our greedy algorithm is optimal!

Exercise What about those other orders by start and finish times, do they also work?

Now for something different [GT §1.4, 10.3, CLRS §17.4, 16.3]

An important part of algorithmic design is understanding how fast (actually, how slow) your algorithms can be.

- Amortized analysis and Accounting. Or:
  Are you sure those worst case iterations happen all the time?

Now for something different [GT §1.4, 10.3, CLRS §17.4, 16.3]

An important part of algorithmic design is understanding how fast (actually, how slow) your algorithms can be.

GT §1.4, CLRS §17.4, (actually all of Ch. 17)

- Amortized analysis and Accounting. Or:
  Are you sure those worst case iterations happen all the time?

Now for something different [GT §1.4, 10.3, CLRS §17.4, 16.3]

An important part of algorithmic design is understanding how fast (actually, how slow) your algorithms can be.

GT §1.4, CLRS §17.4, (actually all of Ch. 17)

- Amortized analysis and Accounting. Or:
  Are you sure those worst case iterations happen all the time?

With data flooding in from all directions, we should probably remember to compress it sometimes. So, let's see a classic compression approach.

GT §10.3, CLRS §16.3

- Huffman Codes. Or:
  Why do we use the same amount of bits for all characters when some occur more often than others?

Amortized Analysis and Accounting [GT §1.4, CLRS §17.4]

The algorithmic analysis tools from the previous course are not enough to analyze some of the algorithms we will be seeing.
For example, consider a **dynamic table**:

- Similar to a regular array, but
- Can hold any number of elements, not a fixed number

Amortized Analysis and Accounting [GT §1.4, CLRS §17.4]

The algorithmic analysis tools from the previous course are not enough to analyze some of the algorithms we will be seeing.
For example, consider a **dynamic table**:

- Similar to a regular array, but
- Can hold any number of elements, not a fixed number

How to implement such a data structure?

- starting with a fixed size table (array), but when it is full:
  - Create a new table of double the size
  - Copy all the old elements into the new, bigger, table

Amortized Analysis and Accounting [GT §1.4, CLRS §17.4]

The algorithmic analysis tools from the previous course are not enough to analyze some of the algorithms we will be seeing.
For example, consider a **dynamic table**:

- Similar to a regular array, but
- Can hold any number of elements, not a fixed number

How to implement such a data structure?

- starting with a fixed size table (array), but when it is full:
  - Create a new table of double the size
  - Copy all the old elements into the new, bigger, table

```
function table_insert(T, newElem)
   if num(T) = size(T)
      U := create_table(2 * size(T))
      for each elem in T
         elementary_insert(U, elem)
         T := U
   elementary_insert(T, newElem)
```

Amortized Analysis and Accounting [GT §1.4, CLRS §17.4]

The algorithmic analysis tools from the previous course are not enough to analyze some of the algorithms we will be seeing.
For example, consider a **dynamic table**:

- Similar to a regular array, but
- Can hold any number of elements, not a fixed number

How to implement such a data structure?

- starting with a fixed size table (array), but when it is full:
  - Create a new table of double the size
  - Copy all the old elements into the new, bigger, table

```
function table_insert(T, newElem)
   if num(T) = size(T)
      U := create_table(2 * size(T))
      for each elem in T
         elementary_insert(U, elem)
         T := U
   elementary_insert(T, newElem)
```

You might ask, this looks pretty standard, why do we need new analysis tools?

Dynamic Table: Analysing an Insertion

Assume both `create_table` and `elementary_insert` are constant $O(1)$ operations.

What is the complexity of `table_insert` ?

```
function table_insert(T, newElem)
   if num(T) = size(T)
      U := create_table(2 * size(T))
      for each elem in T
         elementary_insert(U, elem)
         T := U
   elementary_insert(T, newElem)
```

Dynamic Table: Analysing an Insertion

Assume both `create_table` and `elementary_insert` are constant $O(1)$ operations.

What is the complexity of `table_insert` ?

```
function table_insert(T, newElem)
    if num(T) = size(T)
        U := create_table(2 * size(T))
        for each elem in T
            elementary_insert(U, elem)
            T := U
    elementary_insert(T, newElem)
```

Standard worse-case analysis:

- Expanding `T` is the worse-case
- Thus $O($`size(T)`$)$ per call.

Dynamic Table: Analysing an Insertion

Assume both `create_table` and `elementary_insert` are constant $O(1)$ operations.

What is the complexity of `table_insert` ?

Standard worse-case analysis:

- Expanding `T` is the worse-case
- Thus $O($`size(T)`$)$ per call.

But *most* of the time it is just $O(1)$.

```
function table_insert(T, newElem)
    if num(T) = size(T)
        U := create_table(2 * size(T))
        for each elem in T
            elementary_insert(U, elem)
            T := U
    elementary_insert(T, newElem)
```

Dynamic Table: Analysing an Insertion

Assume both `create_table` and `elementary_insert` are constant $O(1)$ operations.

What is the complexity of `table_insert` ?

Standard worse-case analysis:

- Expanding `T` is the worse-case
- Thus $O($`size(T)`$)$ per call.

But *most* of the time it is just $O(1)$.

```
function table_insert(T, newElem)
    if num(T) = size(T)
        U := create_table(2 * size(T))
        for each elem in T
            elementary_insert(U, elem)
            T := U
    elementary_insert(T, newElem)
```

$\rightsquigarrow$ Analyze the insertion time for a block of many insertions rather than just a single one.

Dynamic Table: Analysing an Insertion

Assume both `create_table` and `elementary_insert` are constant $O(1)$ operations.

What is the complexity of `table_insert` ?

```
function table_insert(T, newElem)
   if num(T) = size(T)
      U := create_table(2 * size(T))
      for each elem in T
         elementary_insert(U, elem)
         T := U
   elementary_insert(T, newElem)
```

Standard worse-case analysis:

- Expanding `T` is the worse-case
- Thus $O($`size(T)`$)$ per call.

But *most* of the time it is just $O(1)$.

$\leadsto$ Analyze the insertion time for a block of many insertions rather than just a single one.

Amortization

- Aggregate amortization (or, just, amortization)
- Accounting amortization (or, just, accounting)

(Aggregate) Amortization

Analyze the operation (insertion) over a sequence of calls.

- Can be a small, constant number of steps
- Can be a larger number of steps (to grow the table)

Goal: Find a function that describes the runtime over an **arbitrarily long** sequence of calls (insertions), we will have found our time complexity

$$\frac{\text{total runtime}}{\text{size of the input}}$$

Build Intuition, then Establish the function

Starting with an empty table, the actual costs of a series of inserts would look like:

- Call 1: 0 copy + 1 insert $\rightarrow$ 1 op. (table full)
- Call 2: 1 copy + 1 insert $\rightarrow$ 2 ops. (table full)
- Call 3: 2 copy + 1 insert $\rightarrow$ 3 ops. (table has one free space)
- Call 4: 0 copy + 1 insert $\rightarrow$ 1 ops. (table full)
- Call 5: 4 copy + 1 insert $\rightarrow$ 5 ops. (table has 3 free spaces)

Build Intuition, then Establish the function

Starting with an empty table, the actual costs of a series of inserts would look like:

- Call 1: 0 copy + 1 insert $\rightarrow$ 1 op. (table full)
- Call 2: 1 copy + 1 insert $\rightarrow$ 2 ops. (table full)
- Call 3: 2 copy + 1 insert $\rightarrow$ 3 ops. (table has one free space)
- Call 4: 0 copy + 1 insert $\rightarrow$ 1 ops. (table full)
- Call 5: 4 copy + 1 insert $\rightarrow$ 5 ops. (table has 3 free spaces)

There is a pattern here

$$1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, \ldots$$

Build Intuition, then Establish the function

Starting with an empty table, the actual costs of a series of inserts would look like:

- Call 1: 0 copy $+$ 1 insert $\rightarrow$ 1 op. (table full)
- Call 2: 1 copy $+$ 1 insert $\rightarrow$ 2 ops. (table full)
- Call 3: 2 copy $+$ 1 insert $\rightarrow$ 3 ops. (table has one free space)
- Call 4: 0 copy $+$ 1 insert $\rightarrow$ 1 ops. (table full)
- Call 5: 4 copy $+$ 1 insert $\rightarrow$ 5 ops. (table has 3 free spaces)

There is a pattern here

$$1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, \ldots$$

Let C_i denote the number of ops. performed by the i^{th} call.

Build Intuition, then Establish the function

Starting with an empty table, the actual costs of a series of inserts would look like:

- Call 1: 0 copy + 1 insert $\rightarrow$ 1 op. (table full)
- Call 2: 1 copy + 1 insert $\rightarrow$ 2 ops. (table full)
- Call 3: 2 copy + 1 insert $\rightarrow$ 3 ops. (table has one free space)
- Call 4: 0 copy + 1 insert $\rightarrow$ 1 ops. (table full)
- Call 5: 4 copy + 1 insert $\rightarrow$ 5 ops. (table has 3 free spaces)

There is a pattern here

$$1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, \ldots$$

Let C_i denote the number of ops. performed by the i^{th} call.

$$C_i = \begin{cases} \quad, & i \text{ is not a power of 2} \\ \quad, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$, where n is the number of calls.

Build Intuition, then Establish the function

Starting with an empty table, the actual costs of a series of inserts would look like:

- Call 1: 0 copy + 1 insert $\rightarrow$ 1 op. (table full)
- Call 2: 1 copy + 1 insert $\rightarrow$ 2 ops. (table full)
- Call 3: 2 copy + 1 insert $\rightarrow$ 3 ops. (table has one free space)
- Call 4: 0 copy + 1 insert $\rightarrow$ 1 ops. (table full)
- Call 5: 4 copy + 1 insert $\rightarrow$ 5 ops. (table has 3 free spaces)

There is a pattern here

1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, ...

Let C_i denote the number of ops. performed by the i^{th} call.

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$, where n is the number of calls.

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i \quad + \sum_{i \text{ is not a power of 2}}^{n} 1$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of } 2 \\ i, & i \text{ is a power of } 2 \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of } 2}^{n} i \; + \; \boxed{\sum_{i \text{ is not a power of } 2}^{n} 1}$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i \quad + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n}$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i \quad + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{??}$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$, where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

Uh, but what is that sum?

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$, where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

Uh, but what is that sum?

trick: re-write in binary

$$\begin{array}{r} 1+ \\ 10+ \\ 100 + \ldots + \\ 10 \cdots 0 \end{array}$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of } 2 \\ i, & i \text{ is a power of } 2 \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of } 2}^{n} i + \sum_{i \text{ is not a power of } 2}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of } 2}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

Uh, but what is that sum?

trick: re-write in binary

How many zeros here?

$$\begin{array}{r} 1+ \\ 10+ \\ 100 + \ldots + \\ 10\cdots0 \end{array}$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of 2}}^{n} i + \sum_{i \text{ is not a power of 2}}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of 2}}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

Uh, but what is that sum?

trick: re-write in binary

How many zeros here?

$\lfloor \log_2 n \rfloor - 1$

$$\begin{array}{r} 1+ \\ 10+ \\ 100 + \ldots + \\ 10 \cdots 0 \end{array}$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of } 2 \\ i, & i \text{ is a power of } 2 \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$, where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of } 2}^{n} i + \sum_{i \text{ is not a power of } 2}^{n} 1$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of } 2}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

Uh, but what is that sum?

trick: re-write in binary

How many zeros here?

$\lfloor \log_2 n \rfloor - 1$

$$\begin{array}{r} 1+ \\ 10+ \\ 100 + \ldots + \\ 10 \cdots 0 \end{array}$$

$$= \overbrace{11 \cdots 11}^{\lfloor \log_2 n \rfloor}$$

$$= 2^{\lfloor \log_2 n \rfloor + 1} - 1$$

$$\leq 2 \cdot 2^{\log_2 n} - 1 = 2n - 1$$

But how much is that really?

$$C_i = \begin{cases} 1, & i \text{ is not a power of } 2 \\ i, & i \text{ is a power of } 2 \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i$,
where n is the number of calls.

$$\sum_{i=1}^{n} C_i = \sum_{i \text{ is a power of } 2}^{n} i + \sum_{i \text{ is not a power of } 2}^{n} 1 \leq 2n-1 + n-(\lfloor \log_2 n \rfloor + 1)$$

$$\leq 3n$$

$$= 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor log_2 n \rfloor} = \sum_{j=1}^{\lfloor log_2 n \rfloor} 2^j$$

$$= \sum_{i=1}^{n} 1 - \sum_{i \text{ is a power of } 2}^{n} 1 = n - (\lfloor \log_2 n \rfloor + 1)$$

Uh, but what is that sum?

trick: re-write in binary

How many zeros here?

$\lfloor \log_2 n \rfloor - 1$

$$1+$$
$$10+$$
$$100 + \ldots +$$
$$10 \cdots 0$$

$$= \overbrace{11 \cdots 11}^{\lfloor \log_2 n \rfloor}$$

$$= 2^{\lfloor \log_2 n \rfloor + 1} - 1$$

$$\leq 2 \cdot 2^{\log_2 n} - 1 = 2n - 1$$

So, the amortized runtime is....

$$C_i = \begin{cases} 1, & i \text{ is not a power of } 2 \\ i, & i \text{ is a power of } 2 \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i \leq 3n$, where n is the number of calls.

- When making n calls, we make (at most) a total of $3n$ ops.
- So the amortized cost (number of ops per call) is:

$$\frac{3n}{n} = 3$$

So, the amortized runtime is....

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i \leq 3n$,
where n is the number of calls.

- When making n calls, we make (at most) a total of $3n$ ops.
- So the amortized cost (number of ops per call) is:

$$\frac{3n}{n} = 3$$

Thus, since each op can be performed in $O(1)$ time, the amortized runtime is also $O(1)$:)

So, the amortized runtime is....

$$C_i = \begin{cases} 1, & i \text{ is not a power of 2} \\ i, & i \text{ is a power of 2} \end{cases}$$

$\rightarrow$ Total Ops: $\sum_{i=1}^{n} C_i \leq 3n$,
where n is the number of calls.

- When making n calls, we make (at most) a total of $3n$ ops.
- So the amortized cost (number of ops per call) is:

$$\frac{3n}{n} = 3$$

Thus, since each op can be performed in $O(1)$ time, the amortized runtime is also $O(1)$:)

That felt like a lot of work ... let's try the other way, maybe the accountants know better.

Amotization by Accounting (aka Charging/Discharging)

- Consider your personal budget
- There are two (or more) types of days: Regular days and Vacation days
- On regular days you pay for your normal activities, and save some money for vacation
- On vacation days you pay for your normal activities, and spend the money you saved up
- This way you never run out of money If you save up enough

Amotization by Accounting (aka Charging/Discharging)

- Consider your personal budget
- There are two (or more) types of days: Regular days and Vacation days
- On regular days you pay for your normal activities, and save some money for vacation
- On vacation days you pay for your normal activities, and spend the money you saved up
- This way you never run out of money If you save up enough

The longer you wait, the more you save.

- a small vacation every month, or
- a bigger one every other month, or an even bigger one every third month, and so on
- saving long enough means affording arbitrarily-expensive vacation

Amotization by Accounting (aka Charging/Discharging)

Applied to algorithms:

- Every operation gets athe same (e.g., constant) **income**
- Simple operations pay for their cost out of this income, and put the rest in **savings**
- Expensive operations we can use **savings** (and income) while ensuring the **savings** are never negative.
- If we get a constant income, this means we have constant amortized time complexity.

Amotization by Accounting (aka Charging/Discharging)

Applied to algorithms:

- Every operation gets athe same (e.g., constant) **income**
- Simple operations pay for their cost out of this income, and put the rest in **savings**
- Expensive operations we can use **savings** (and income) while ensuring the **savings** are never negative.
- If we get a constant income, this means we have constant amortized time complexity.

For our dynamic table example, we give `table_insert` an **income of 3**.
When the table doesn't expand, It **pays 1** to do the actual insert and **saves 2** in the bank
Otherwise, it will pay the copying cost plus 1 for the next insert.

Let see how these values progress

In the table we can see how these values change as elements are added.

- **Income** is how much we get paid for each `table_insert`
- **Cost** is the number of operations required for that insertion
- +/- is amount leftover (or deficit) after paying for the operations.
- **Savings** what we have remaining in the bank after covering any deficit.

i	0	1	2	3	4	5	6	7	8	9	10
income	3	3	3	3	3	3	3	3	3	3	3
cost	1	2	3	1	5	1	1	1	9	1	1
+/-	2	1	0	2	-2	2	2	2	-6	2	2
savings	2	3	3	5	3	5	7	9	3	5	7

But how should we prove an income of 3 is enough?

Intuition:

- 1 unit of income gets the element into the table initially
- 1 unit pays for the element to be copied the first time
- 1 unit pays for a previous element to be copied again

With this in mind, we use induction on the expensive calls.

- If i is a power of 2 we need to copy over i elements, for a cost of i.
- By induction, we know that our savings was non-negative at the previous expensive call, $i/2$ calls ago.
- Thus, we have savings of 2 units from each of the last $i/2$ elements, for a total savings of $2(i/2) = i$ units.

Therefore we will never go broke.

Huffman Coding [GT §10.3, CLRS §16.3]

What is the problem to solve?
You have a huge text file (for example xml, json, some other even more horrible format).

- Often stored just as plain text, e.g., ASCII, unicode, etc.
- In ASCII all the characters are the same length - 8 bits
- This seems wasteful – maybe more common characters should get short codes.

Huffman Coding [GT §10.3, CLRS §16.3]

What is the problem to solve?
You have a huge text file (for example xml, json, some other even more horrible format).

- Often stored just as plain text, e.g., ASCII, unicode, etc.
- In ASCII all the characters are the same length - 8 bits
- This seems wasteful – maybe more common characters should get short codes.

For example, in English the most common letters are E, A, and T
Maybe we can use short bit patterns:

- "E" $\rightarrow$ 0
- "A" $\rightarrow$ 1
- "T" $\rightarrow$ 01

Huffman Coding [GT §10.3, CLRS §16.3]

What is the problem to solve?
You have a huge text file (for example xml, json, some other even more horrible format).

- Often stored just as plain text, e.g., ASCII, unicode, etc.
- In ASCII all the characters are the same length - 8 bits
- This seems wasteful – maybe more common characters should get short codes.

For example, in English the most common letters are E, A, and T
Maybe we can use short bit patterns:

- "E" $\rightarrow$ 0
- "A" $\rightarrow$ 1
- "T" $\rightarrow$ 01

Will this work?

Huffman Coding [GT §10.3, CLRS §16.3]

What is the problem to solve?
You have a huge text file (for example xml, json, some other even more horrible format).

- Often stored just as plain text, e.g., ASCII, unicode, etc.
- In ASCII all the characters are the same length - 8 bits
- This seems wasteful – maybe more common characters should get short codes.

For example, in English the most common letters are E, A, and T
Maybe we can use short bit patterns:

- "E" $\rightarrow$ 0
- "A" $\rightarrow$ 1
- "T" $\rightarrow$ 01

Will this work?
EAT $\rightarrow$ 0101 $\rightarrow$??

Huffman Coding [GT §10.3, CLRS §16.3]

What is the problem to solve?
You have a huge text file (for example xml, json, some other even more horrible format).

- Often stored just as plain text, e.g., ASCII, unicode, etc.
- In ASCII all the characters are the same length - 8 bits
- This seems wasteful – maybe more common characters should get short codes.

For example, in English the most common letters are E, A, and T
Maybe we can use short bit patterns:

- "E" $\rightarrow$ 0
- "A" $\rightarrow$ 1
- "T" $\rightarrow$ 01

Will this work?
EAT $\rightarrow$ 0101 $\rightarrow$??

1. So, not just any short codes can be used together.
2. Need to ensure the receiver of such a code can revert the it back to the original text.

Huffman Coding [GT §10.3, CLRS §16.3]

What is the problem to solve?
You have a huge text file (for example xml, json, some other even more horrible format).

- Often stored just as plain text, e.g., ASCII, unicode, etc.
- In ASCII all the characters are the same length - 8 bits
- This seems wasteful – maybe more common characters should get short codes.

For example, in English the most common letters are E, A, and T
Maybe we can use short bit patterns:

- "E" $\rightarrow$ 0
- "A" $\rightarrow$ 1
- "T" $\rightarrow$ 01

Will this work?
EAT $\rightarrow$ 0101 $\rightarrow$??

1. So, not just any short codes can be used together.
2. Need to ensure the receiver of such a code can revert the it back to the original text.

Fixing the Ambiguity

Use a **prefix code**.

- In a prefix code, no code word is a prefix of any other code word
- The problem before: the code word for "E" (0) is a prefix for the code word for "T" (01)
- The recipient is never sure if a 0 means an "E", or if it just the first bit of a "T"
- New code words:
  - "E" $\rightarrow$ 0
  - "A" $\rightarrow$ 10
  - "T" $\rightarrow$ 11
- Now "EAT" $=$ 01011
- And it can only be interpreted one way

Finally, formalizing the problem

Compute he best way to assign (prefix) code words to letters.
More formally:

- Given a message X
  - X uses characters $C = \{c_1, c_2, ..., c_n\}$
  - For each c_i, let $f(c_i)$ denote how many times c_i occurs in X.
  - Informally, the frequency of each character
- Create a code for C which allows X to be transmitted such that
  - The code allows X to be unambiguously recovered, e.g., it is a prefix code
  - The code ensures that the encoding of X uses the fewest bits possible.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Finally, formalizing the problem

Compute he best way to assign (prefix) code words to letters.
More formally:

- Given a message X
  - X uses characters $C = \{c_1, c_2, ..., c_n\}$
  - For each c_i, let $f(c_i)$ denote how many times c_i occurs in X.
  - Informally, the frequency of each character
- Create a code for C which allows X to be transmitted such that
  - The code allows X to be unambiguously recovered, e.g., it is a prefix code
  - The code ensures that the encoding of X uses the fewest bits possible.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Sounds too good to be true.... ?

Finally, formalizing the problem

Compute he best way to assign (prefix) code words to letters.
More formally:

- Given a message X
  - X uses characters $C = \{c_1, c_2, ..., c_n\}$
  - For each c_i, let $f(c_i)$ denote how many times c_i occurs in X.
  - Informally, the frequency of each character
- Create a code for C which allows X to be transmitted such that
  - The code allows X to be unambiguously recovered, e.g., it is a prefix code
  - The code ensures that the encoding of X uses the fewest bits possible.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

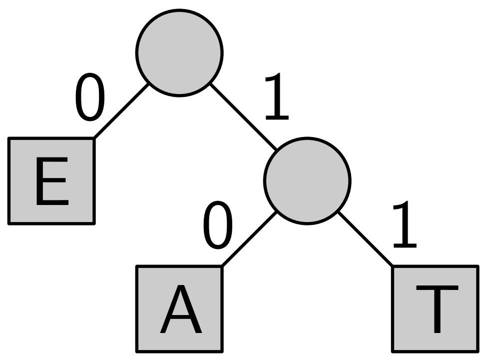
Sounds too good to be true.... ?

Fewest bits where each character is encoded separately.
Different approaches can do better when encoding more than one character at a time.
But, for single character encodings, there really is a **best** approach.

Huffman Coding

The encoding strategy is formed by the construction of a rooted binary tree where:

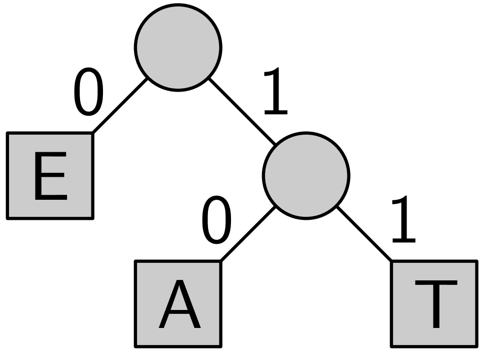
- Leaves of the tree map **bijectively** to the characters of the given string.
- For each non-leaf vertex of the tree, one of the two outgoing edges is labelled "0" and the other (if present) is labelled "1".
- The **encoding** of each character C is given by reading the edge labels along the path from the **root** to the **leaf** corresponding to C.
  - "E" $\to$ 0
  - "A" $\to$ 10
  - "T" $\to$ 11



Huffman Coding

The encoding strategy is formed by the construction of a rooted binary tree where:

- Leaves of the tree map **bijectively** to the characters of the given string.
- For each non-leaf vertex of the tree, one of the two outgoing edges is labelled "0" and the other (if present) is labelled "1".
- The **encoding** of each character C is given by reading the edge labels along the path from the **root** to the **leaf** corresponding to C.
  - "E" $\rightarrow$ 0
  - "A" $\rightarrow$ 10
  - "T" $\rightarrow$ 11



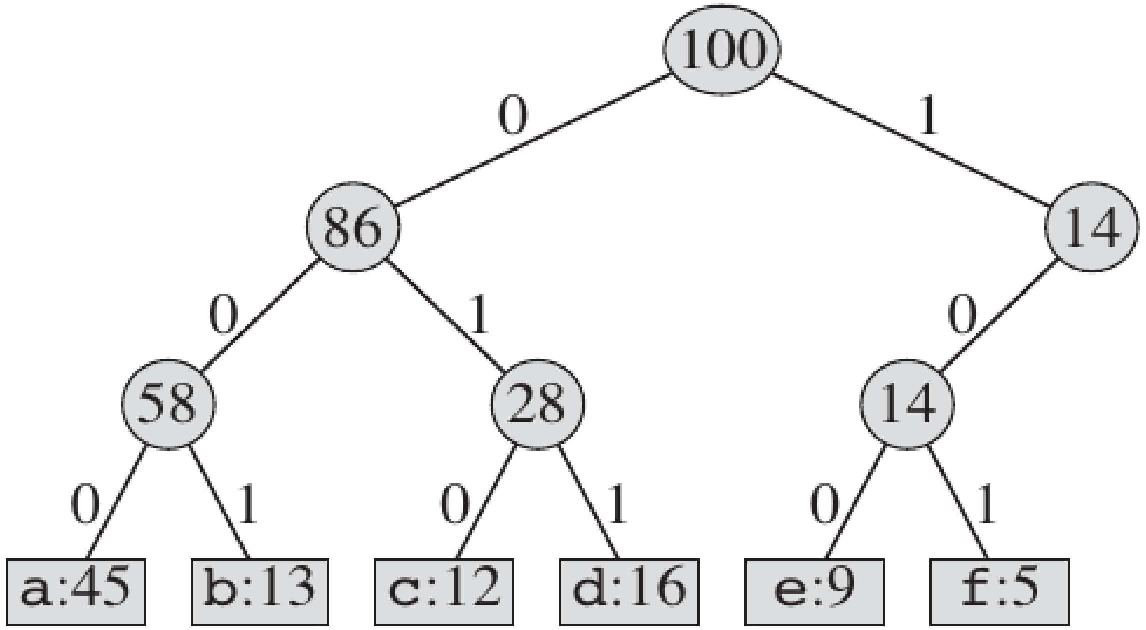
The key insights are:

- The codes arising from such a binary tree are **prefix codes**. Moreover, with such a tree we can unambiguously decode any binary string encoded with the same tree.
- Optimality: Characters whose leaves are close to the root should occur more often in the string than characters whose leaves are further away from the root.

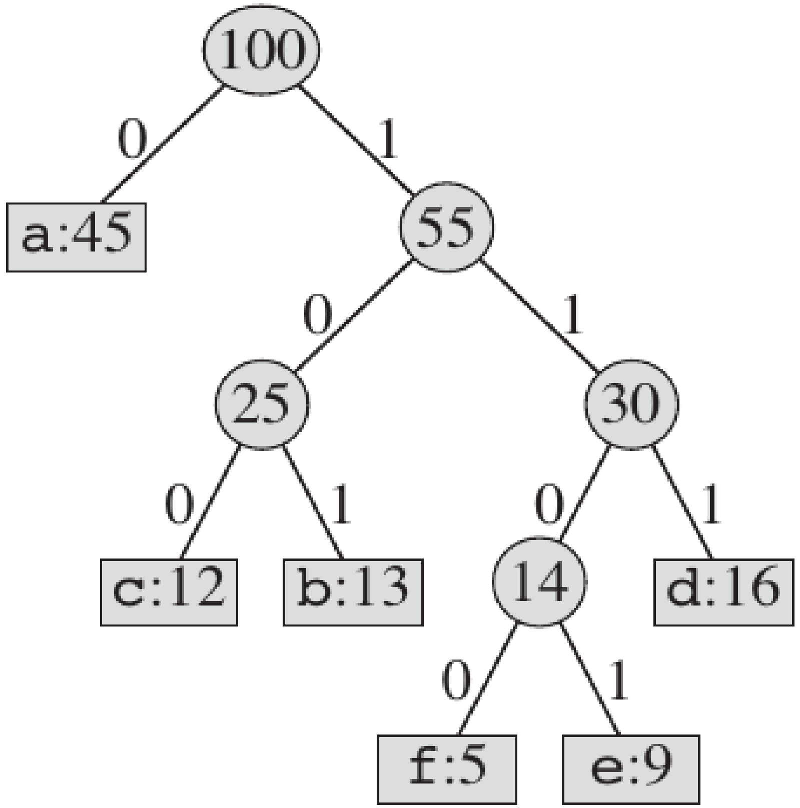
Different trees mean different encoding.

Based on a file with 100,000 characters, but only containing {a,b,c,d,e,f}

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a)

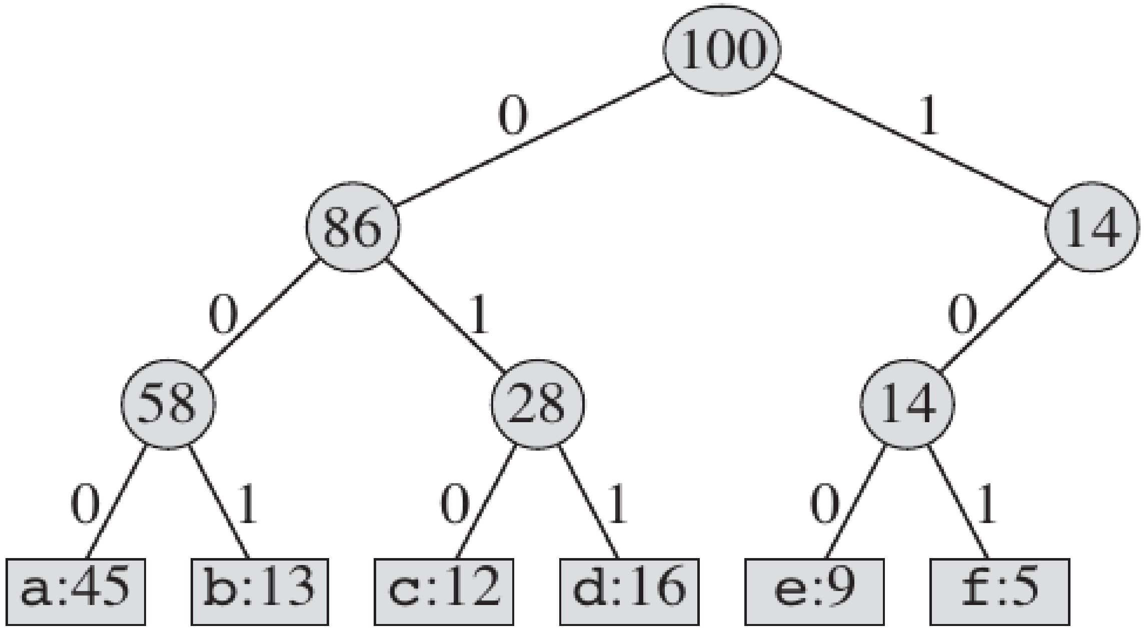


(b)

Different trees mean different encoding.

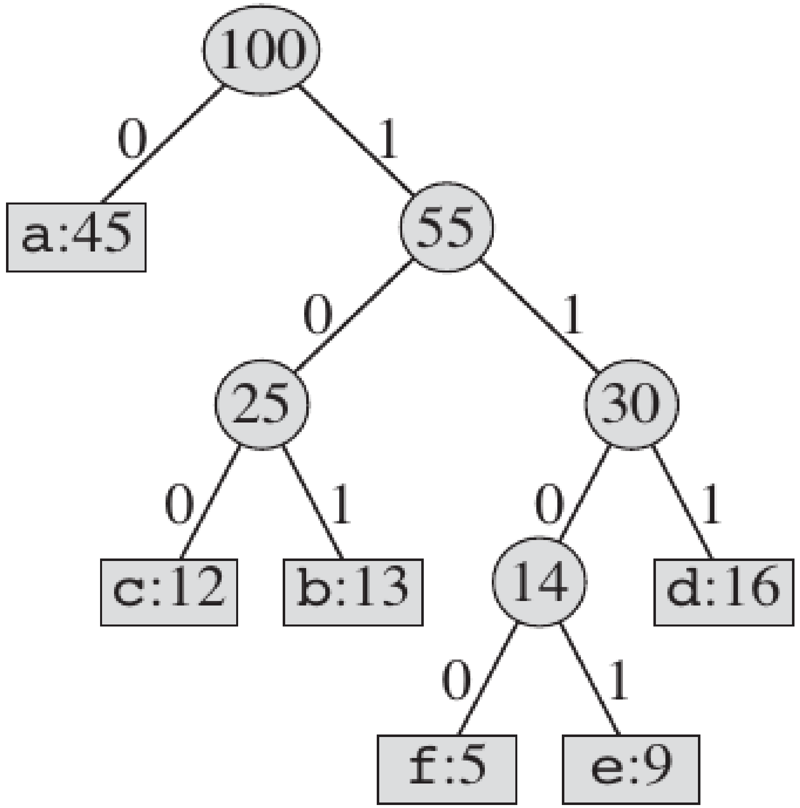
Based on a file with 100,000 characters, but only containing `{a,b,c,d,e,f}`

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



Can this tree be optimal?

(a)



(b)

How to construct an optimal tree?

Insight: low frequency characters should be **far** from the root.

How to construct an optimal tree?

Insight: low frequency characters should be **far** from the root.

So, let's try a greedy choice to ensure this.
$\rightsquigarrow$ Build the tree **bottom-up**, start with the **furthest leaves**, i.e., low frequency characters.

How to construct an optimal tree?

Insight: low frequency characters should be **far** from the root.

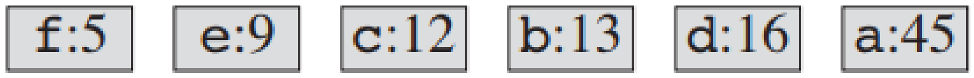
So, let's try a greedy choice to ensure this.
$\rightsquigarrow$ Build the tree **bottom-up**, start with the **furthest leaves**, i.e., low frequency characters.

How to construct an optimal tree?

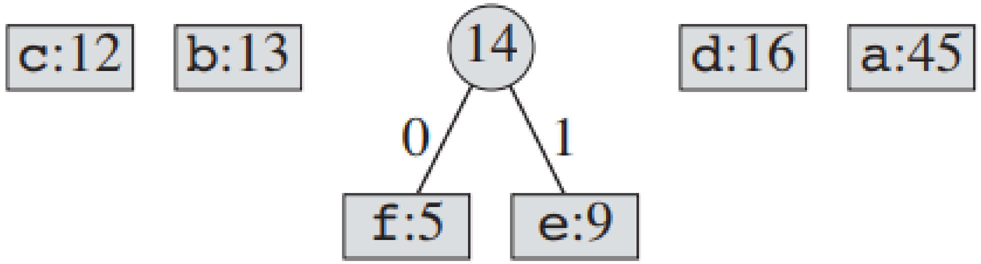
Insight: low frequency characters should be **far** from the root.

So, let's try a greedy choice to ensure this.
$\rightsquigarrow$ Build the tree **bottom-up**, start with the **furthest leaves**, i.e., low frequency characters.

(a)



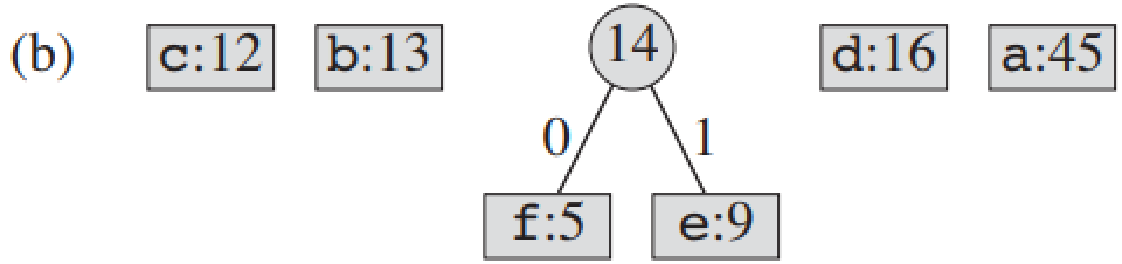
(b)

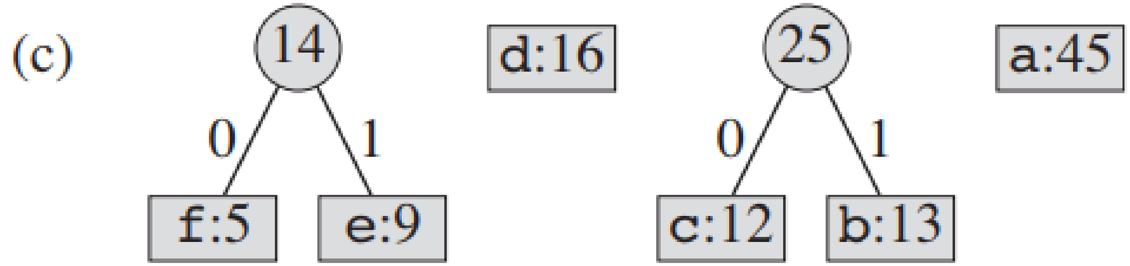


How to construct an optimal tree?

Insight: low frequency characters should be **far** from the root.

So, let's try a greedy choice to ensure this.
$\rightsquigarrow$ Build the tree **bottom-up**, start with the **furthest leaves**, i.e., low frequency characters.

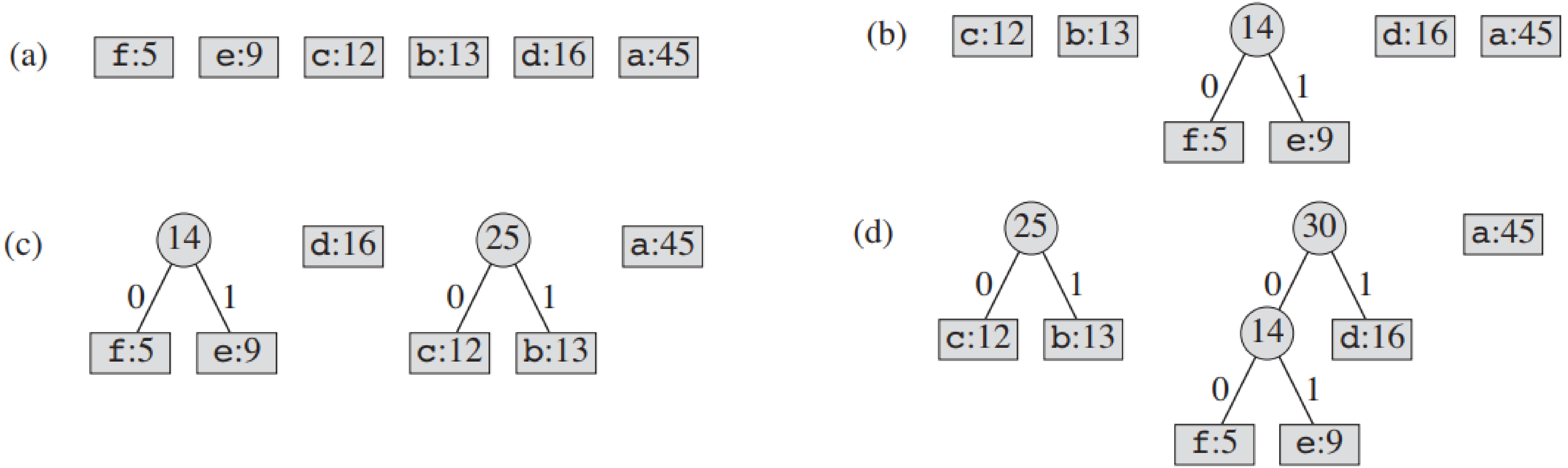




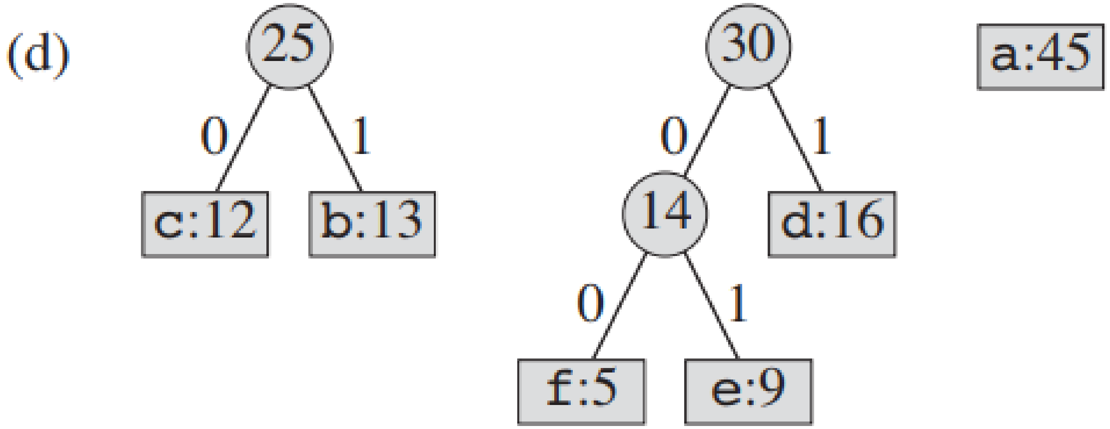
How to construct an optimal tree?

Insight: low frequency characters should be **far** from the root.

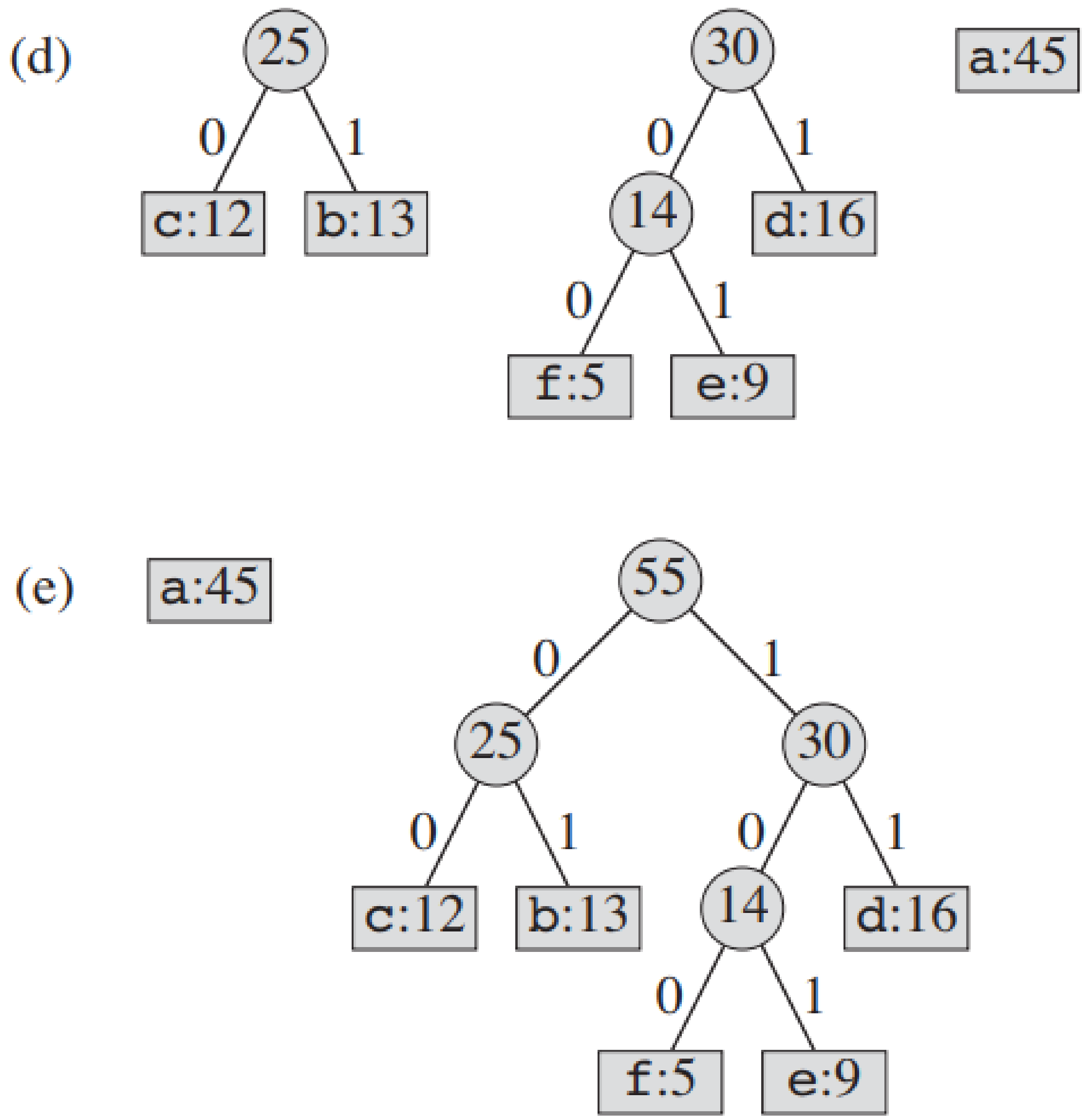
So, let's try a greedy choice to ensure this.
$\rightsquigarrow$ Build the tree **bottom-up**, start with the **furthest leaves**, i.e., low frequency characters.



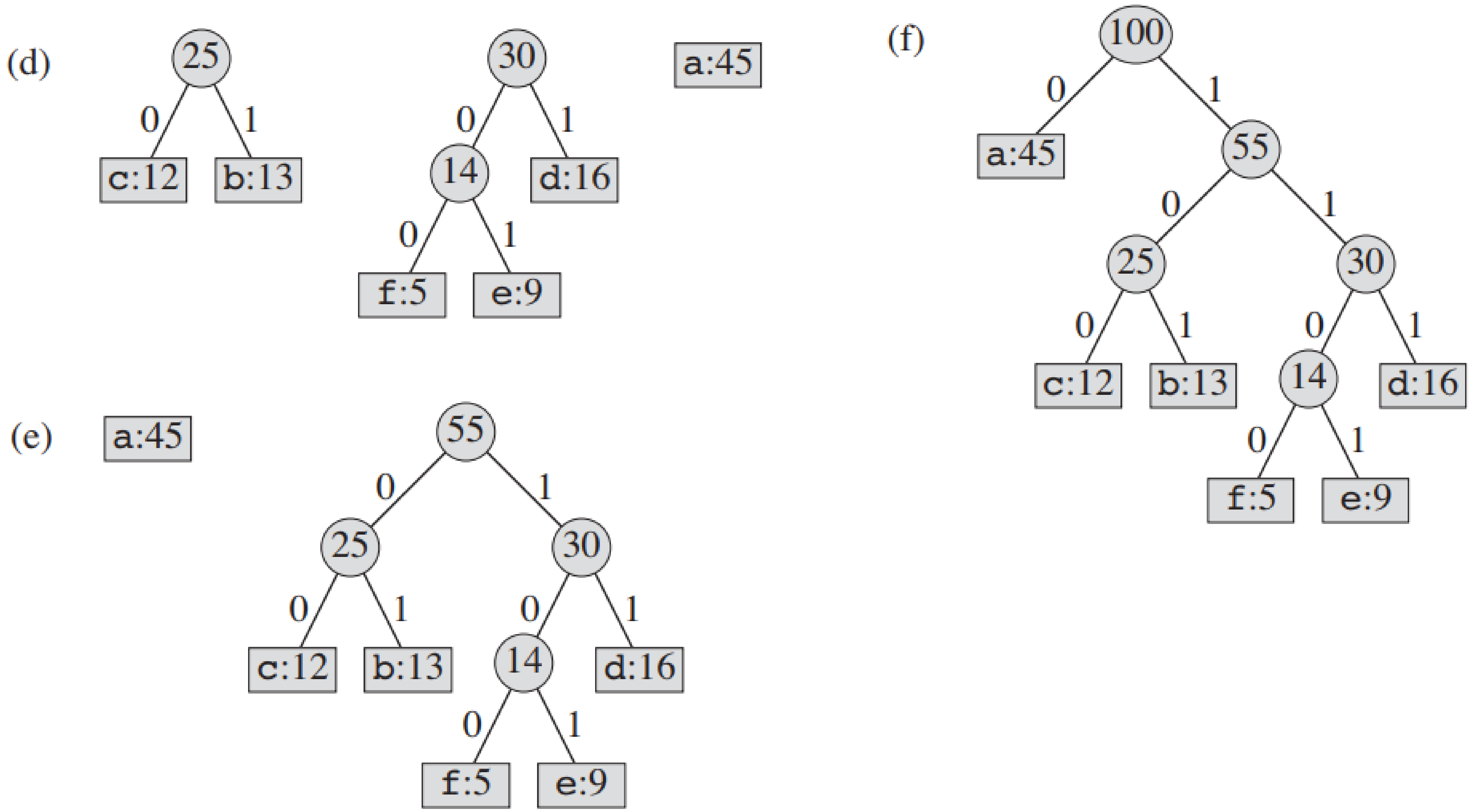
How to construct an optimal tree?



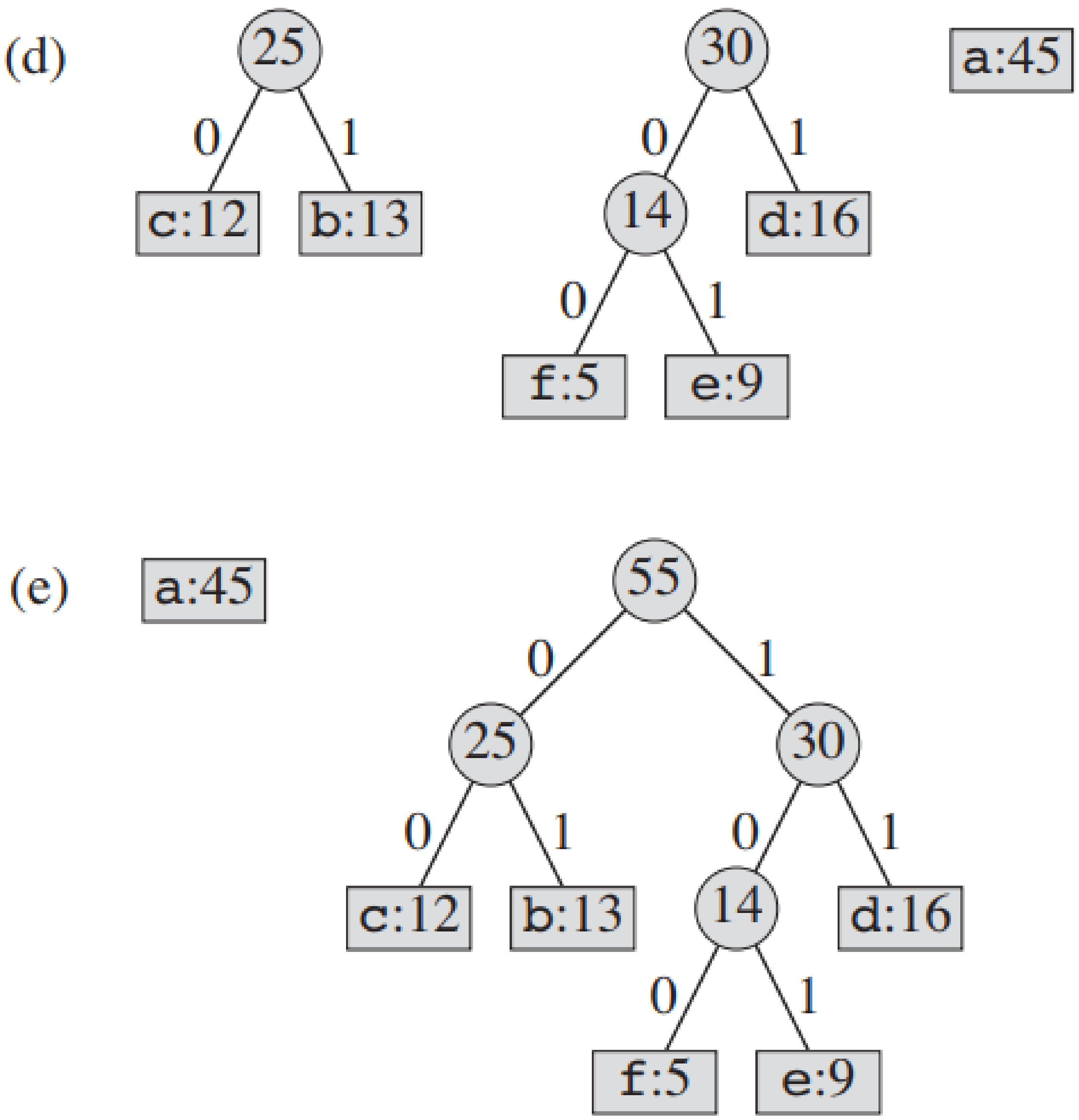
How to construct an optimal tree?

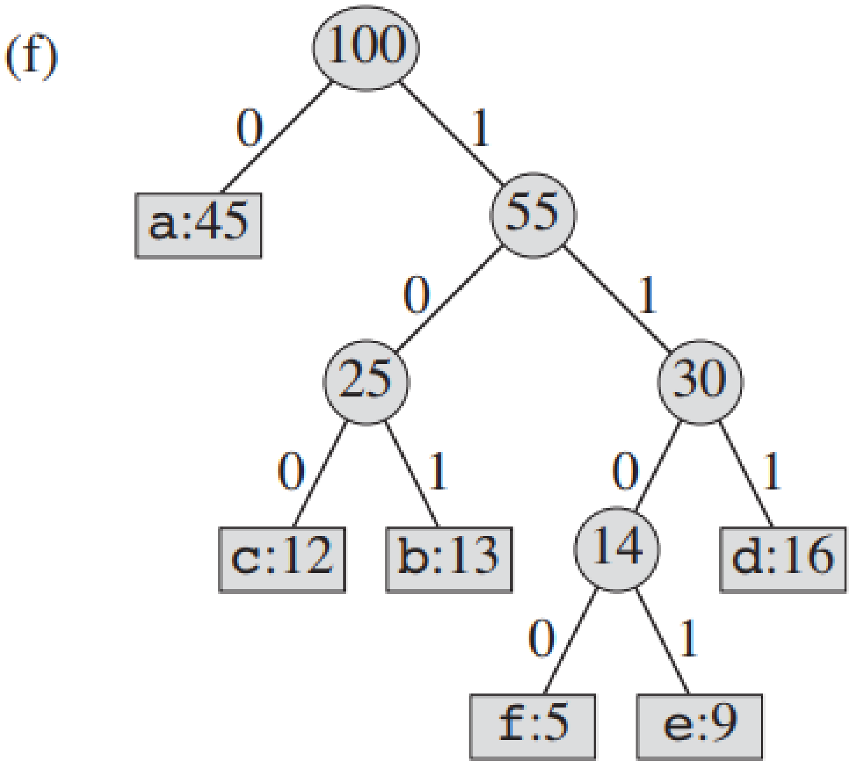


How to construct an optimal tree?



How to construct an optimal tree?





HUFFMAN(C)

$n = |C|$
$Q = C$
for $i = 1$ **to** $n - 1$
    allocate a new node z
    $z.left = x =$ EXTRACT-MIN(Q)
    $z.right = y =$ EXTRACT-MIN(Q)
    $z.freq = x.freq + y.freq$
    INSERT(Q, z)
return EXTRACT-MIN(Q)   // return the root of the tree

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

(as discussed before)

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Proof (idea) Use contradiction.
Let T be an optimal code, and let C'' be a character distinct from C, C' with the longest code. Swap the code of C (or C') for that of C''.

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Theorem: The tree constructed by our algorithm admits an optimal code.

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Theorem: The tree constructed by our algorithm admits an optimal code.

Prove by induction on the number of distinct characters.

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Theorem: The tree constructed by our algorithm admits an optimal code.

Prove by induction on the number of distinct characters.

Try it! (hint: use the two lemmas)

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Theorem: The tree constructed by our algorithm admits an optimal code.

Prove by induction on the number of distinct characters.

Try it! (hint: use the two lemmas)

Another **exercise:** What is the runtime of our algorithm?

But why is this optimal?

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)        // return the root of the tree
```

Lemma: No internal vertex of an optimal Huffman tree has out-degree 1.

Lemma: Let C, C' be the two characters with the lowest frequency in a string X, and let T be an optimal code for X. The characters C, C' must have the longest codes in T.

Theorem: The tree constructed by our algorithm admits an optimal code.

Prove by induction on the number of distinct characters.

Try it! (hint: use the two lemmas)

Another **exercise:** What is the runtime of our algorithm?
How to encode and decode once we have such a code?

Next Week

More algorithmic techniques: Divide and Conquer

e.g., as in Merge Sort.

How to analyze the runtime of recursive methods (e.g., as arising from Divide and Conquer)

The MASTER theorem