Algorithms and Complexity (AC)

Marie Schmidt & Tom van der Zanden

(Based on slides by Gerhard Woeginger and Jesper Nederlof)

Landelijk Netwerk Mathematische Besliskunde

LNMB, Sep–Nov 2021

Program for today

1. Introduction round
2. Course Intro
3. First lecture

Introduction round

(mentimeter)

(Preliminary) program

13 Sep : Introduction, algorithms, time complexity and computational models, P versus NP
20 Sep : reductions, NP-hardness and NP-completeness
27 Sep : Pseudopolynomial time, strong/weak NP-hardness, co-NP
30 Sept : Approximation algorithms
4 Oct : Exercise set 1
4 Oct : More on approximation algorithms
18 Oct : Exercise set 2
11 Oct : Exact algorithms for NP-hard problems
18 Oct : Exercise set 2
18 Oct : More exact algorithms for NP-hard problems
1 Nov : Exercise set 3
1 Nov : Treewidth
8 Nov : Randomized algorithms
15 Nov : Exercise set 4
15 Nov : **No lecture!!**

Disclaimer: We are discussing famous decision and optimization problems, focusing on their complexity and algorithms to solve them.
We do not, in detail, studying the application of this in real-world planning,

(Preliminary) program

13 Sep : Introduction, algorithms, time complexity and computational models, P versus NP
20 Sep : reductions, NP-hardness and NP-completeness
27 Sep : Pseudopolynomial time, strong/weak NP-hardness, co-NP
30 Sept : Approximation algorithms
4 Oct : Exercise set 1
4 Oct : More on approximation algorithms
18 Oct : Exercise set 2
11 Oct : Exact algorithms for NP-hard problems
18 Oct : Exercise set 2
18 Oct : More exact algorithms for NP-hard problems
1 Nov : Exercise set 3
1 Nov : Treewidth
8 Nov : Randomized algorithms
15 Nov : Exercise set 4
15 Nov : **No lecture!!**

Website: `https://tomvanderzanden.nl/LNMB-AC/`

First 5 lectures: Marie Schmidt (`schmidt2@rsm.nl`): **life on Zoom**,
last 4 lectures: Tom van der Zanden (`tom.vd.zanden@gmail.com`):

Life lectures

first 5 lectures are going to be life lectures
in some cases: supplemented by videos

lectures are recorded

but:

- please attend
- please turn on your camera
- please participate

Life lectures

first 5 lectures are going to be life lectures
in some cases: supplemented by videos

lectures are recorded

but:

- please attend
- please turn on your camera
- please participate

Do we need a 10 minutes break after approx. 45 minutes?

Life lectures

first 5 lectures are going to be life lectures
in some cases: supplemented by videos

lectures are recorded

but:

- please attend
- please turn on your camera
- please participate

I do not read the chat while presenting.
If anything goes wrong, please just interrupt me and speak up!
If you have a question: either interrupt me, or wait for the next 'aquamarine' slide.

Attendance

LNMB rule:

- attendance + exercises: 4 EC
- only attendance: 1 EC

Please post '[your name] is here' in the chat for documentation of attendance.

Exercises

1. first series: deadline 4-Oct-2021, 13:15:
   already available on `https://tomvanderzanden.nl/LNMB-AC/`,
   you may want to wait 1 week before solving them
   please email to `schmidt@rsm.nl`
2. second series: deadline 18-Oct-2021, 13:15
   please email to `schmidt@rsm.nl`
3. third series: deadline 1-Nov-2021, 13:15
4. fourth series: deadline 15-Nov-2021, 13:15

May be solved in groups of 2.
My strong advice: do indeed solve them in groups of two!

Questions?

1. first series: deadline 4-Oct-2021, 13:15:
   already available on `https://tomvanderzanden.nl/LNMB-AC/`,
   you may want to wait 1 week before solving them
   please email to `schmidt@rsm.nl`
2. second series: deadline 18-Oct-2021, 13:15
   please email to `schmidt@rsm.nl`
3. third series: deadline 1-Nov-2021, 13:15
4. fourth series: deadline 15-Nov-2021, 13:15

May be solved in groups of 2.
My strong advice: do indeed solve them in groups of two!

Program for the first three weeks

- algorithms,
- computational models and (worst-case) time complexity
- decision problems, P versus NP
- Reductions
- NP-hardness
- A catalogue of NP-hard problems
- pseudo-polynomial time
- strong NP-hardness & weak NP-hardness
- co-NP, co-NP versus NP

What is an algorithm?

What is an algorithm?

Algorithm

Well-defined procedure that transforms an input into an output.

What is an algorithm?

Algorithm

Well-defined procedure that transforms an input into an output.

Example: Insertion Sort

Input: A sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$
Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

What is an algorithm?

Algorithm

Well-defined procedure that transforms an input into an output.

Example: Insertion Sort

Input: A sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$
Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

for $j = 2, \ldots, n$ **do**
  $key := A[j]$
  $i := j - 1$
  while $i > 0$ and $A[i] > key$ **do**
    $A[i + 1] := A[i]$
    $i := i - 1$
  end while
  $A[i + 1] := key$
end for
return A

When we analyze an algorithm, we are interested in:

- running time of the algorithm
- space (memory) needed by the algorithm
- for optimization problems: quality of the output
  - exact algorithm
  - approximation algorithm
  - heuristic algorithm

Time and space complexity of 'Insertion Sort'

- How much time did we need?
- **How much space did we need?**
  - in our example
  - for sorting any sequence of 4 numbers
  - for sorting any sequence of n numbers

Big-O notation

We make use of big-O notation for discussing encoding space, space complexity, and time complexity.

big-O notation

$f(n)$ is $O(g(n))$ denotes
$\exists n_0, C$ such that for all $n \geq n_0, f(n) \leq C \cdot g(n)$.

Big-O notation

We make use of big-O notation for discussing encoding space, space complexity, and time complexity.

big-O notation

$f(n)$ is $O(g(n))$ denotes
$\exists n_0, C$ such that for all $n \geq n_0, f(n) \leq C \cdot g(n)$.

For example, $4n^2 + 3n \in O(n^2)$ and $7n^2 + 2 \in O(n^2)$

Does space complexity depend on the encoding?

Example: How much space does storing the number 351 need?

Does space complexity depend on the encoding?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Does space complexity depend on the encoding?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Any decimal number x with n digits can be displayed with $n \cdot \log_2(10)$ digits in binary.

Does space complexity depend on the encoding?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Any decimal number x with n digits can be displayed with $n \cdot \log_2(10)$ digits in binary. Because:

- $x < 10^n$, therefore x needs at most as many digits as 10^{n-1} in the binary representation

Does space complexity depend on the encoding?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Any decimal number x with n digits can be displayed with $n \cdot \log_2(10)$ digits in binary. Because:

- $x < 10^n$, therefore x needs at most as many digits as 10^{n-1} in the binary representation
- and $\log_2(10^n) = n \cdot \log_2(10)$

Does space complexity depend on the encoding?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Any decimal number x with n digits can be displayed with $n \cdot \log_2(10)$ digits in binary. Because:

- $x < 10^n$, therefore x needs at most as many digits as 10^{n-1} in the binary representation
- and $\log_2(10^n) = n \cdot \log_2(10)$

$\rightarrow$ Both in decimal and binary format, the number can be stored in space $O(n)$

Questions?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Any decimal number x with n digits can be displayed with $(n-1) \cdot \log_2(10)$ digits in binary.
Because:

- $x < 10^{n-1}$, therefore x needs at most as many digits as 10^{n-1} in the binary representation

Questions?

Example: How much space does storing the number 351 need?

decimal:	351	$1 \cdot 10^0 + 5 \cdot 10^1 + 3 \cdot 10^2$
binary:	101011111	$1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$

$\rightarrow$ We need $9 = \lfloor \log_2(351) \rfloor + 1$ digits to display the decimal number 351 in binary

Any decimal number x with n digits can be displayed with $(n-1) \cdot \log_2(10)$ digits in binary.
Because:

- $x < 10^{n-1}$, therefore x needs at most as many digits as 10^{n-1} in the binary representation
- and $\log_2(10^{n-1}) = (n-1) \cdot \log_2(10)$

$\rightarrow$ Both in decimal and binary format, the number can be stored in space $O(n)$

Time and space complexity of 'Insertion Sort'

- How much time did we need?
- How much space did we need?

Time and space complexity of 'Insertion Sort'

- How much time did we need?
  $\rightarrow$ measured in *elementary operations*

- How much space did we need?

Time and space complexity of 'Insertion Sort'

- How much time did we need?
  $\rightarrow$ measured in *elementary operations*
  Definition of 'elementary operation' depends on computational model

- How much space did we need?

Time complexity: computational model

Our choice: Random-access-machine (RAM) model

executes operations one after another (no concurrent operations)

Elementary steps $\hat{=}$ assumption: can be executed in constant time

- arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- data movement: load, store, copy
- control: unconditional and conditional branch, subroutine call, return

Time complexity: computational model

Our choice: Random-access-machine (RAM) model

executes operations one after another (no concurrent operations)

Elementary steps $\hat{=}$ assumption: can be executed in constant time

- arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- data movement: load, store, copy
- control: unconditional and conditional branch, subroutine call, return

For 'constant time' assumption: limit on length of each 'word of data'

Time complexity: computational model

Our choice: Random-access-machine (RAM) model

executes operations one after another (no concurrent operations)

Elementary steps $\hat{=}$ assumption: can be executed in constant time

- arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- data movement: load, store, copy
- control: unconditional and conditional branch, subroutine call, return
- exponentiation?

For 'constant time' assumption: limit on length of each 'word of data'

Time complexity: computational model

Our choice: Random-access-machine (RAM) model

executes operations one after another (no concurrent operations)

Elementary steps $\hat{=}$ assumption: can be executed in constant time

- arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
- data movement: load, store, copy
- control: unconditional and conditional branch, subroutine call, return
- exponentiation?

For 'constant time' assumption: limit on length of each 'word of data'

Why do we use RAM:

- similar to how a computer works & approximates running time of computer well

Time Complexity of Insertion Sort

- How many elementary steps did we make to sort the sequence $(5, 2, 4, 1)$ with Insertion sort?
- How many elementary steps do we need, at most, for sorting a sequence of 4 numbers?
- How many elementary steps do we need, at most, for sorting a sequence of n numbers?

Insertion Sort

Example: Insertion Sort

Input: A sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$
Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

```
for j = 2, ..., n do
    key := A[j]
    i := j − 1
    while i > 0 and A[i] > key do
        A[i + 1] := A[i]
        i := i − 1
    end while
    A[i + 1] := key
end for
return A
```

Conventions when talking about running time

1. In this course, we measure running time as a *function of the input length* n.

Conventions when talking about running time

1. In this course, we measure running time as a *function of the input length* n.
2. In this course, we consider worst-case running time, i.e., we answer the question: what is the maximum running time over all possible algorithm inputs of length n?

Conventions when talking about running time

1. In this course, we measure running time as a *function of the input length* n.
2. In this course, we consider worst-case running time, i.e., we answer the question: what is the maximum running time over all possible algorithm inputs of length n?
3. We make use of big-O notation.

big-O notation

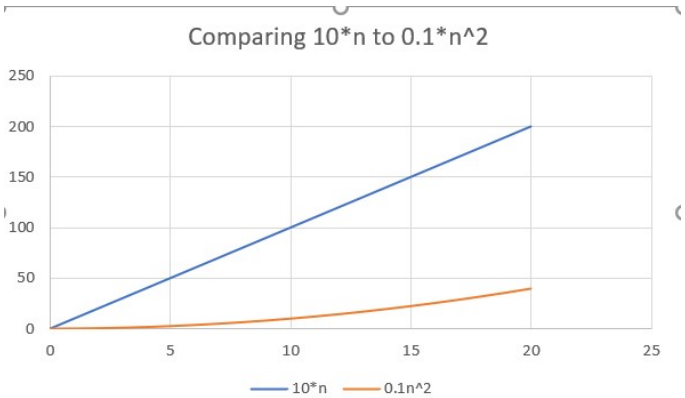
$f(n)$ is $O(g(n))$ denotes
$\exists n_0, C$ such that for all $n \geq n_0, f(n) \leq C \cdot g(n)$.

For example, $4n^2 + 3n \in O(n^2)$ and $7n^2 + 2 \in O(n^2)$

Comparing running times

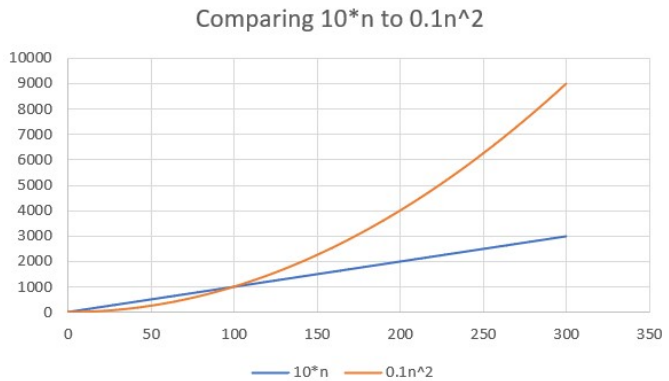
What is better: an algorithm that solves your problem in time $O(n)$, or one that solves it in time $O(n^2)$?

Comparing running times



What is better: an algorithm that solves your problem in time $O(n)$, or one that solves it in time $O(n^2)$? $\rightarrow$ it depends!

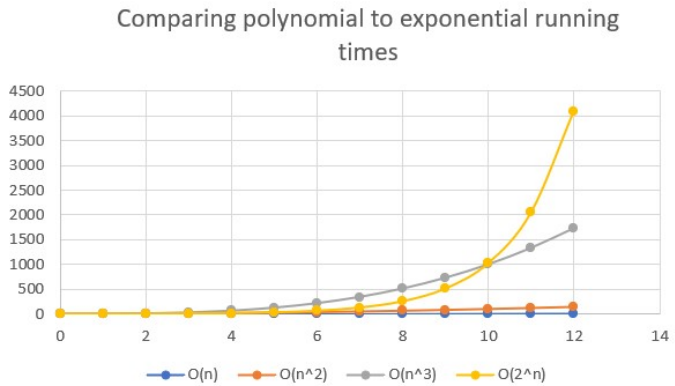
Comparing running times



What is better: an algorithm that solves your problem in time $O(n)$, or one that solves it in time $O(n^2)$? $\rightarrow$ it depends!

As we are most worried about solving large problem instances:
We consider $O(n)$ to be better than $O(n^2)$,
$O(n^2)$ to be better than $O(n^3)$
$O(n^3)$ to be better than $O(n^4)$
...

Comparing running times



What is better: an algorithm that solves your problem in time $O(n)$, or one that solves it in time $O(n^2)$? $\rightarrow$ it depends!

As we are most worried about solving large problem instances:
We consider $O(n)$ to be better than $O(n^2)$,
$O(n^2)$ to be better than $O(n^3)$
$O(n^3)$ to be better than $O(n^4)$
...
...and any *polynomial-time algorithm* to be better than an *exponential-time algorithm*

Vocabulary

Time complexity of algorithms

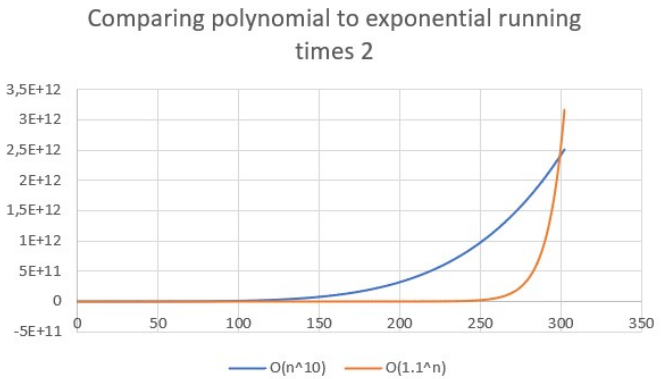
We call an algorithm *polynomial-time algorithm*, if there is a number i such that the algorithm's worst-case running time is in $O(n^i)$.
Otherwise we call it an *exponential-time algorithm*.

Exponential-time algorithms

- Refer to any algorithm whose worst-case running time is not in $O(n^i)$ for a fixed i.

Exponential-time algorithms

- Refer to any algorithm whose worst-case running time is not in $O(n^i)$ for a fixed i.
- Are considered as 'inefficient' by many people **Intuition:**
  Polynomial = desirable, good, harmless, fast, short, small
  Exponential = undesirable, bad, evil, slow, wasteful, horrible
- But: can be better than polynomial-time algorithms for specific problem instances (see also later sessions of this course)!



Other approaches to measure algorithm complexity

- Average case (instead of worst-case) complexity
  (not treated in this course)
- Complexity measured not (only) in input length n
  Alternatives:
  - size of the output (e.g., in multicriteria optimization)
  - numbers contained in program input $\rightarrow$ more on this when we talk about 'pseudo-polynomial' algorithms
  - instance characteristics $\rightarrow$ more on this when we talk about 'fixed-parameter tractability'

Questions?

- Average case (instead of worst-case) complexity
  (not treated in this course)
- Complexity measured not (only) in input length n
  Alternatives:
  - size of the output (e.g., in multicriteria optimization)
  - numbers contained in program input $\rightarrow$ more on this when we talk about 'pseudo-polynomial' algorithms
  - instance characteristics $\rightarrow$ more on this when we talk about 'fixed-parameter tractability'

Problem complexity

Problems are specified in terms of
problem input: what is given
problem output: what is searched for

Example: Insertion Sort

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Problem complexity

Problems are specified in terms of
problem input: what is given
problem output: what is searched for

Example: Insertion Sort

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Often there are many different algorithms to solve a problem

Problem complexity

Problems are specified in terms of
problem input: what is given
problem output: what is searched for

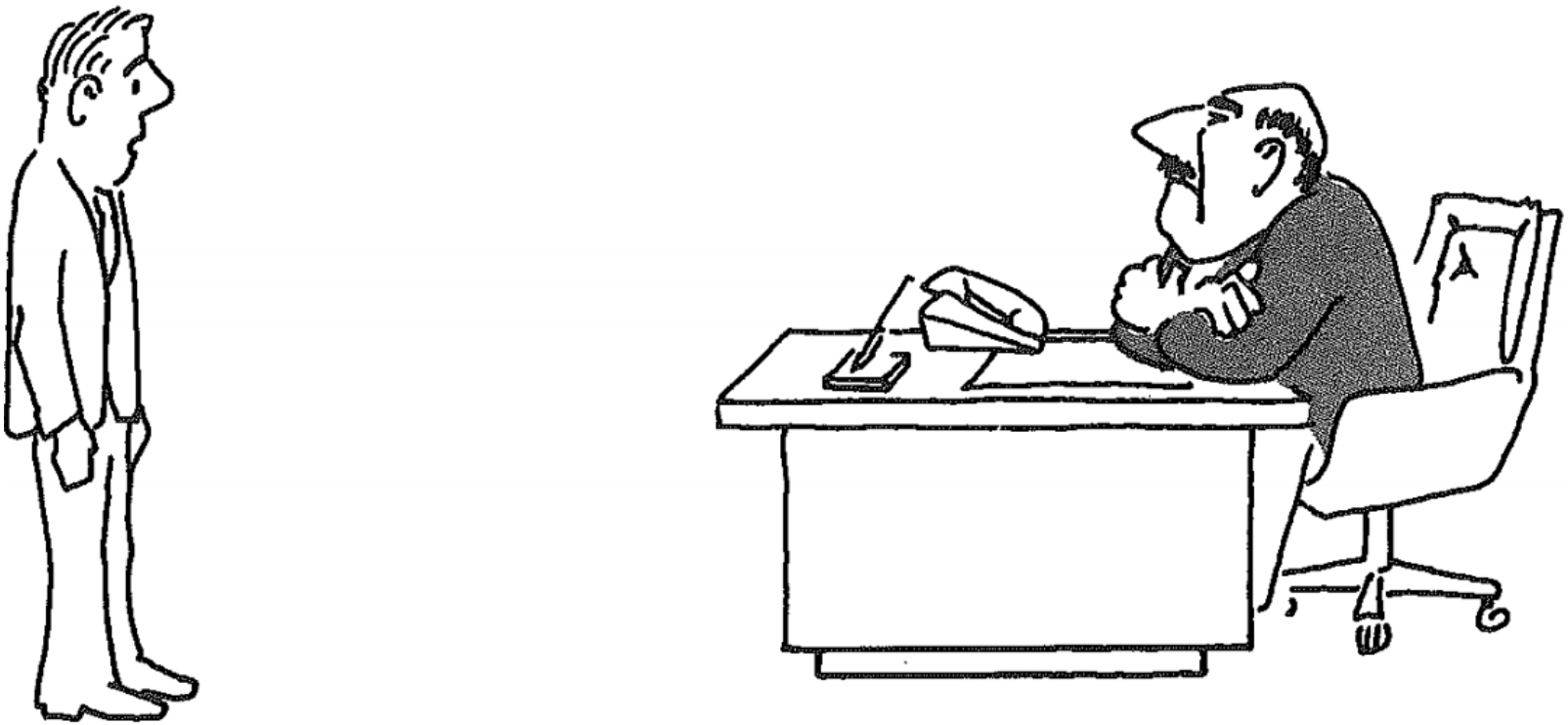
Example: Insertion Sort

Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$
Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

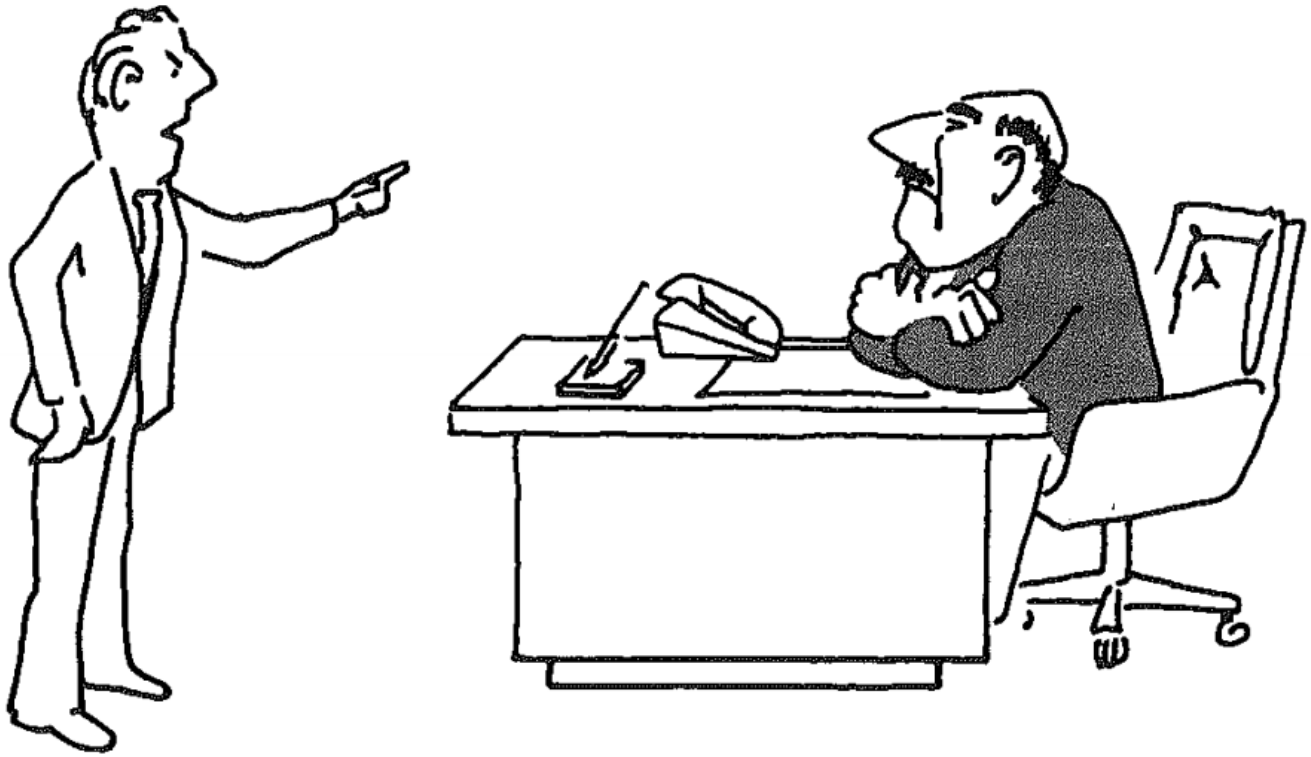
Often there are many different algorithms to solve a problem

Selection of famous sorting algorithms (see wikipedia for many more):

- Insertion sort ($O(n^2)$)
- Merge sort ($O(n \log(n))$)
- Bubble sort ($O(n^2)$)
- Shell sort ($O(n^{3/2})$)
- Bogosort ($O((n+1)!)$)



"I can't find an efficient algorithm, I guess I'm just too dumb."



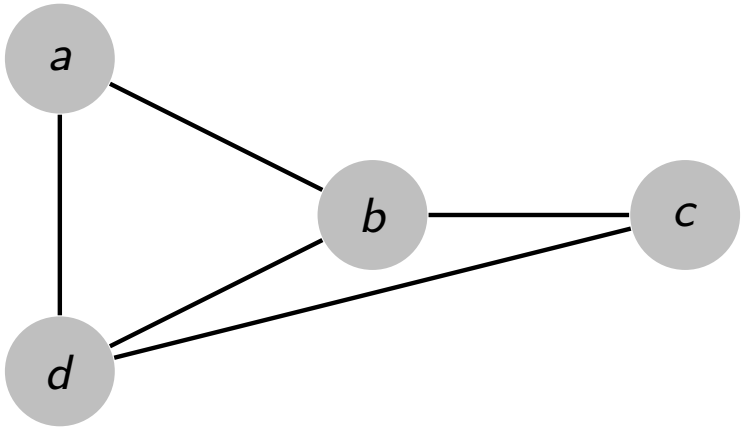
"I can't find an efficient algorithm, because no such algorithm is possible!"

Decision problems and optimization problems

In the remainder of the course, we are mostly concerned with the following two types of algorithmic problems:

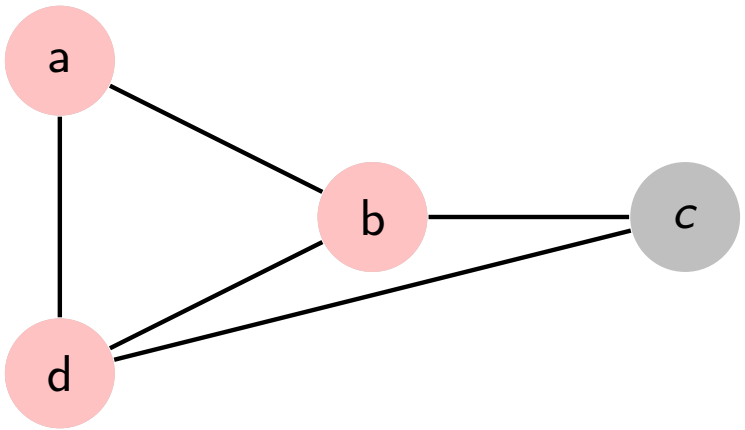
- Optimization problems (min/max)
- Decision problems (with answer YES/NO)

Decision problems and optimization problems



Reminder 'clique': set of pairwise adjacent vertices

Decision problems and optimization problems

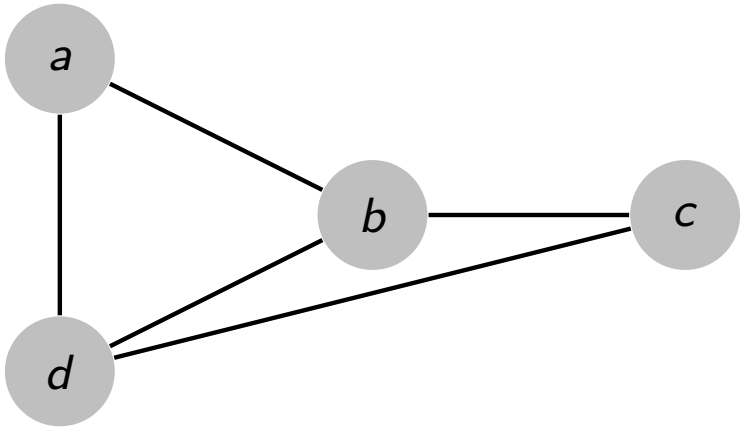


Reminder 'clique': set of pairwise adjacent vertices

Example: Decision problem CLIQUE

Instance: a graph $G = (V, E)$; a bound k
Question: does G contain a clique of size (at least) k?

Decision problems and optimization problems



Reminder 'clique': set of pairwise adjacent vertices

Example: Decision problem CLIQUE

Instance: a graph $G = (V, E)$; a bound k
Question: does G contain a clique of size (at least) k?

Example: Optimization problem CLIQUE

Instance: a graph $G = (V, E)$
Goal: find a clique of maximum size in G. / What is the maximum size of a clique in G?

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Does G contain a clique of size at least $n/2$? – YES

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Does G contain a clique of size at least $n/2$? – YES
Does G contain a clique of size at least $3n/4$? – YES

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Does G contain a clique of size at least $n/2$? – YES
Does G contain a clique of size at least $3n/4$? – YES
Does G contain a clique of size at least $7n/8$? – NO

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Does G contain a clique of size at least $n/2$? – YES
Does G contain a clique of size at least $3n/4$? – YES
Does G contain a clique of size at least $7n/8$? – NO
Does G contain a clique of size at least $13n/16$? – YES

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Does G contain a clique of size at least $n/2$? – YES
Does G contain a clique of size at least $3n/4$? – YES
Does G contain a clique of size at least $7n/8$? – NO
Does G contain a clique of size at least $13n/16$? – YES
Etc.

Search takes logarithmic number of steps –> fast and simple

Questions?

Observation

Every discrete optimization problem can be rewritten into a sequence of decision problems:

use bisection search on the interval of objective values

Example

Let G be a graph on n vertices.

Does G contain a clique of size at least $n/2$? – YES
Does G contain a clique of size at least $3n/4$? – YES
Does G contain a clique of size at least $7n/8$? – NO
Does G contain a clique of size at least $13n/16$? – YES
Etc.

Search takes logarithmic number of steps –> fast and simple

Complexity class P

Remember: Polynomial growth rate:

- $O(\text{poly}(n))$ for some polynomial poly Example: $O(n)$; $O(n \log n)$; $O(n^3)$; $O(n^{100})$

Exponential growth rate:

- everything that grows faster than polynomial

Example: 2^n; 3^n; $n!$; 2^{2^n}; n^n

Intuition:

Polynomial = desirable, good, harmless, fast, short, small
Exponential = undesirable, bad, evil, slow, wasteful, horrible

Complexity class P

Remember: Polynomial growth rate:

- $O(\text{poly}(n))$ for some polynomial poly Example: $O(n)$; $O(n \log n)$; $O(n^3)$; $O(n^{100})$

Exponential growth rate:

- everything that grows faster than polynomial

Example: 2^n; 3^n; $n!$; 2^{2^n}; n^n

Intuition:

Polynomial = desirable, good, harmless, fast, short, small
Exponential = undesirable, bad, evil, slow, wasteful, horrible

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

If you are interested in what a Turing machine is, and how it relates to algorithms, please watch the (additional) video provided on our course website

Example: The Minimum Spanning Tree (MST) problem

'Minimum Spanning Tree' (MST) - Decision version

Given: a graph $G = (V, E)$ and $w_e \in \mathbb{R}$ for every $e \in E$, a number W
Question: is there a spanning tree $T \subseteq E$ such that $\sum_{e \in T} w_e \leq W$

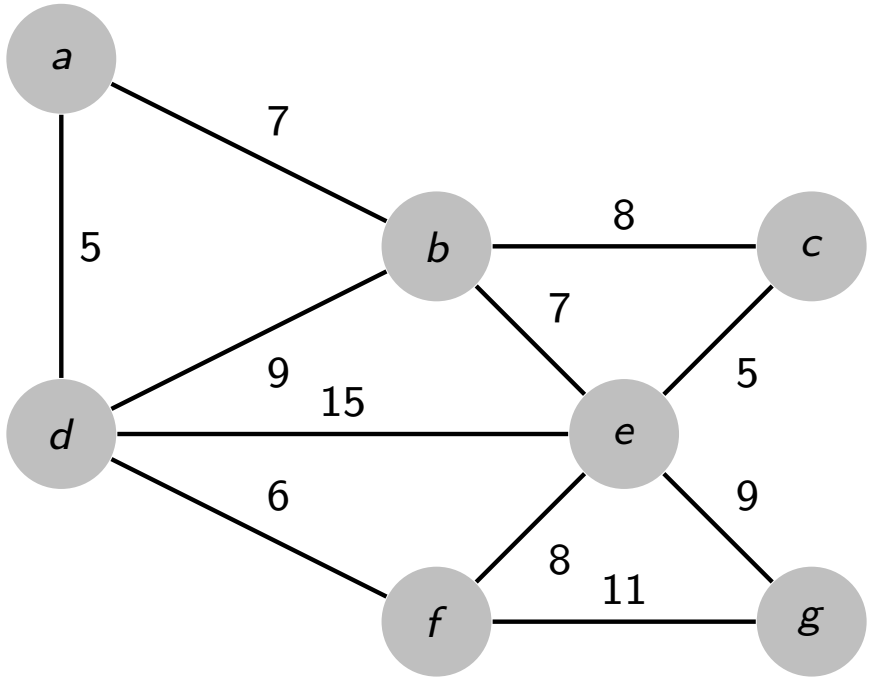
Terminology:

- tree: edge-set without cycles (e.g. at most 1 path between 2 vertices)
- spanning: all vertices are incident to an edge

Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

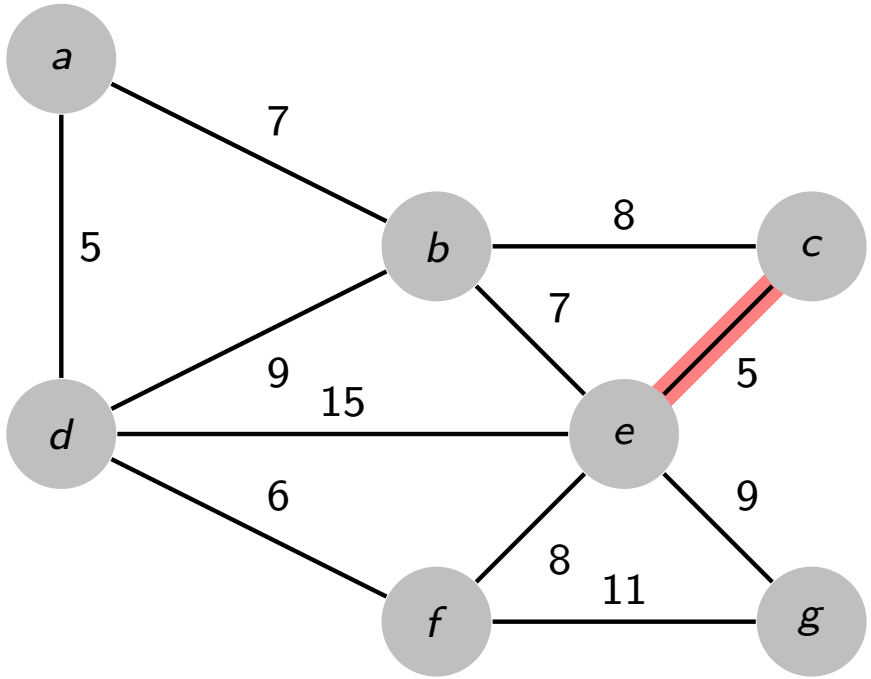
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

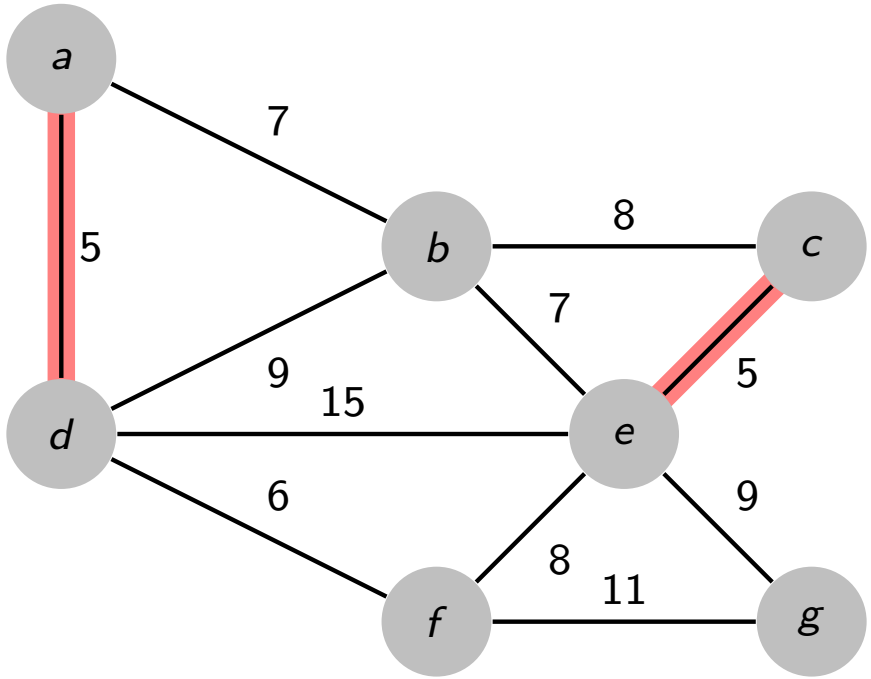
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

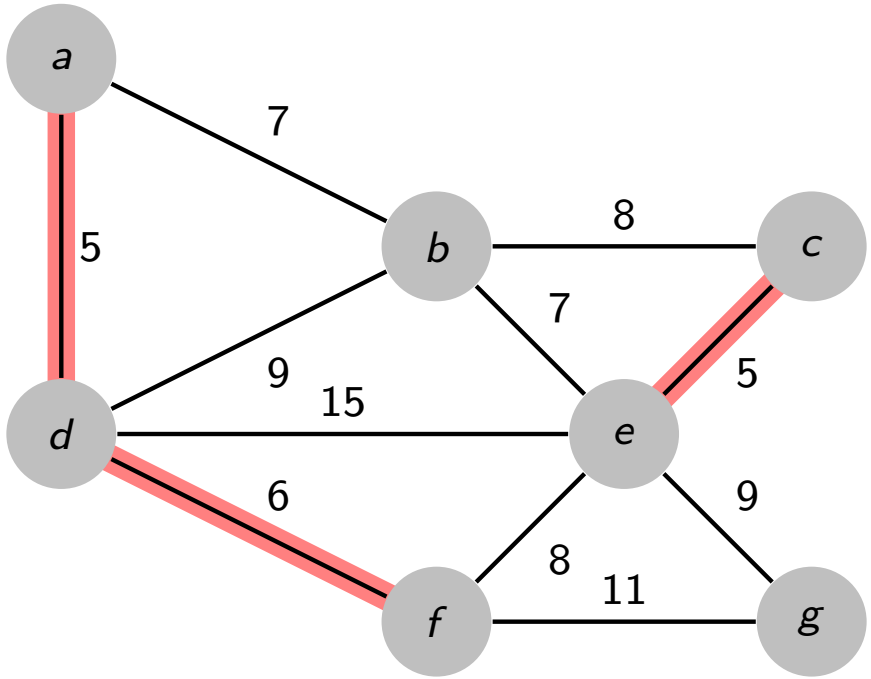
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

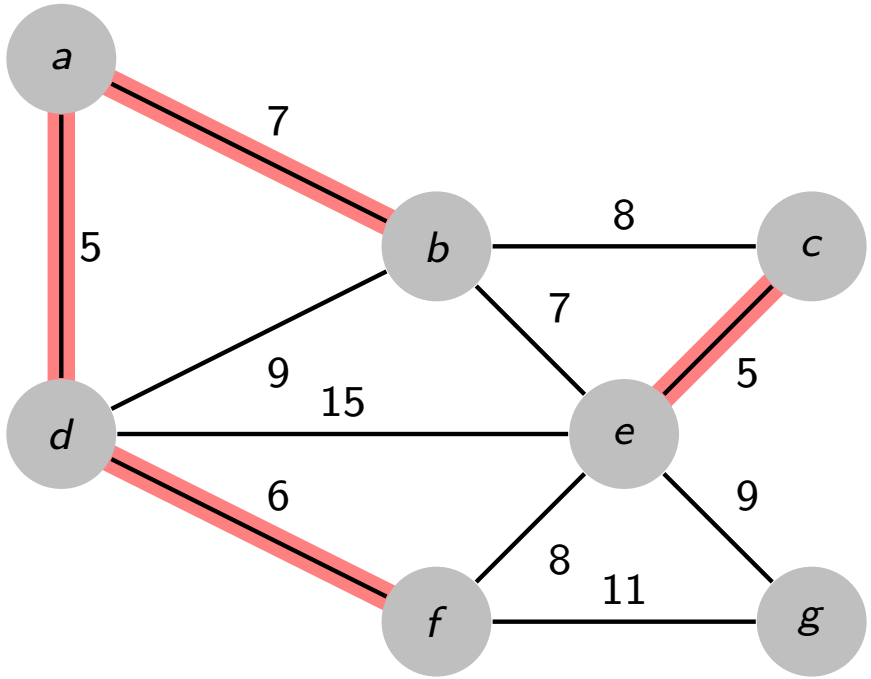
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

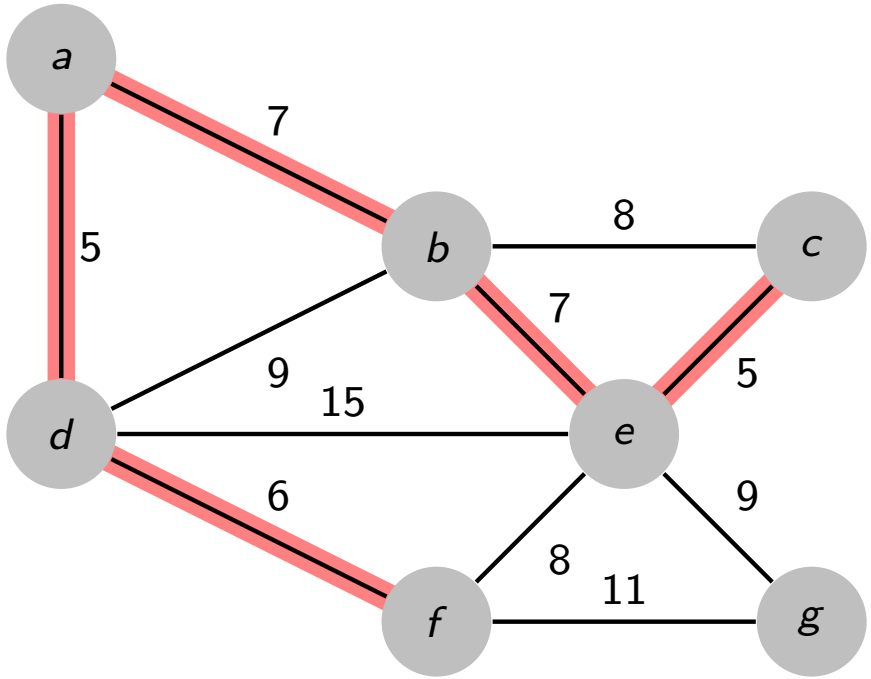
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

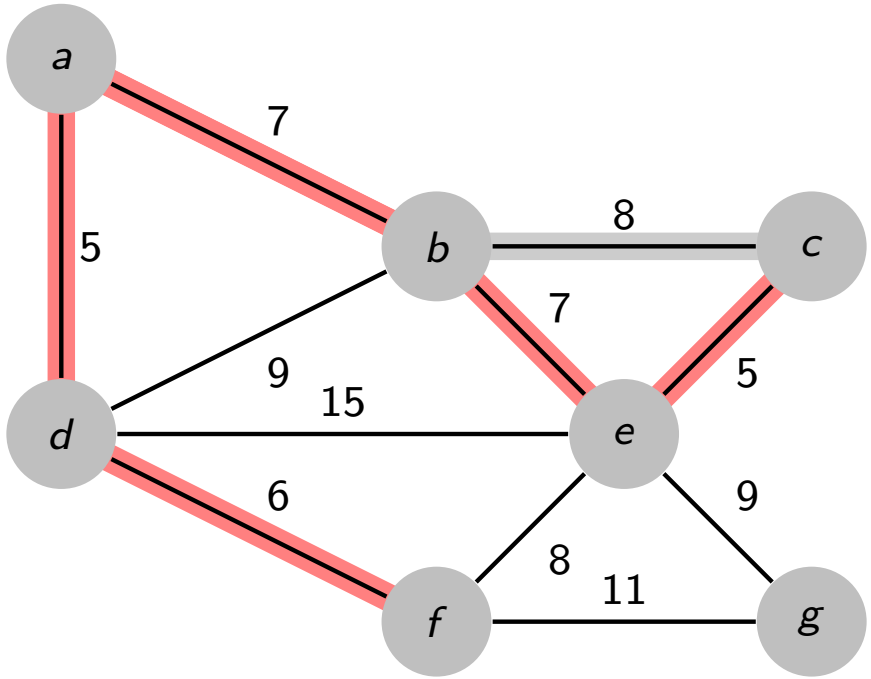
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

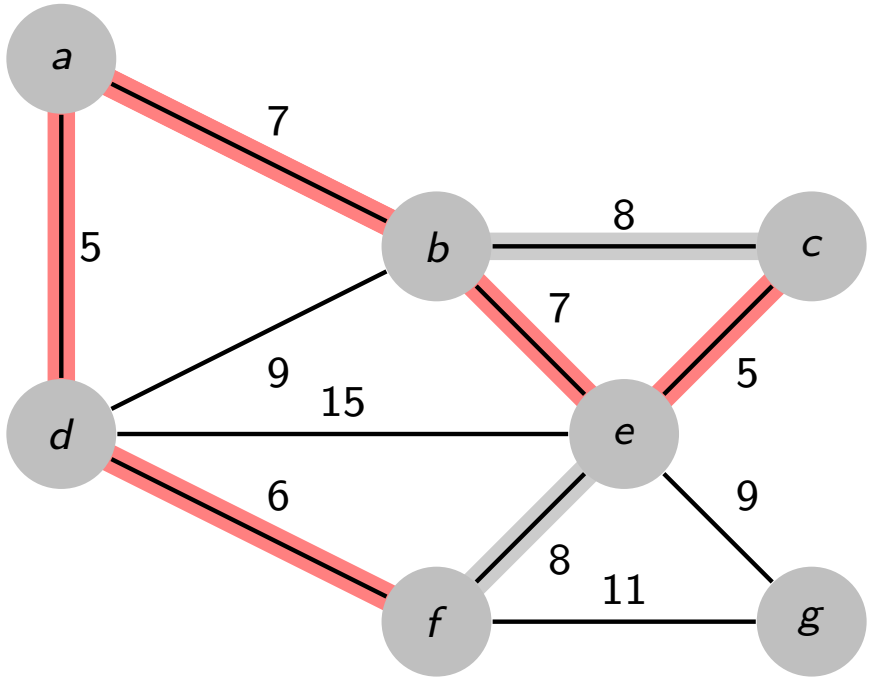
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

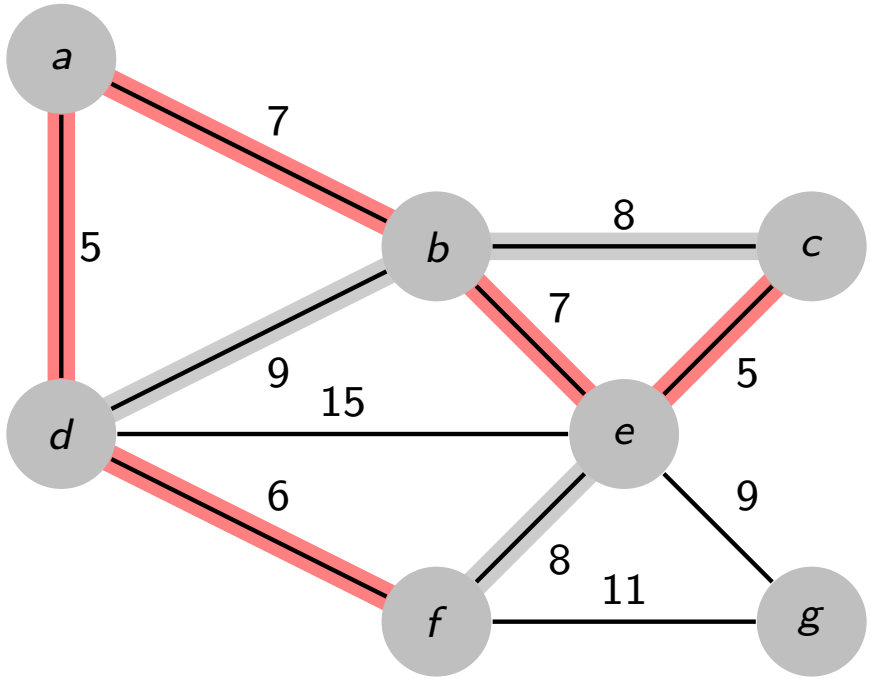
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

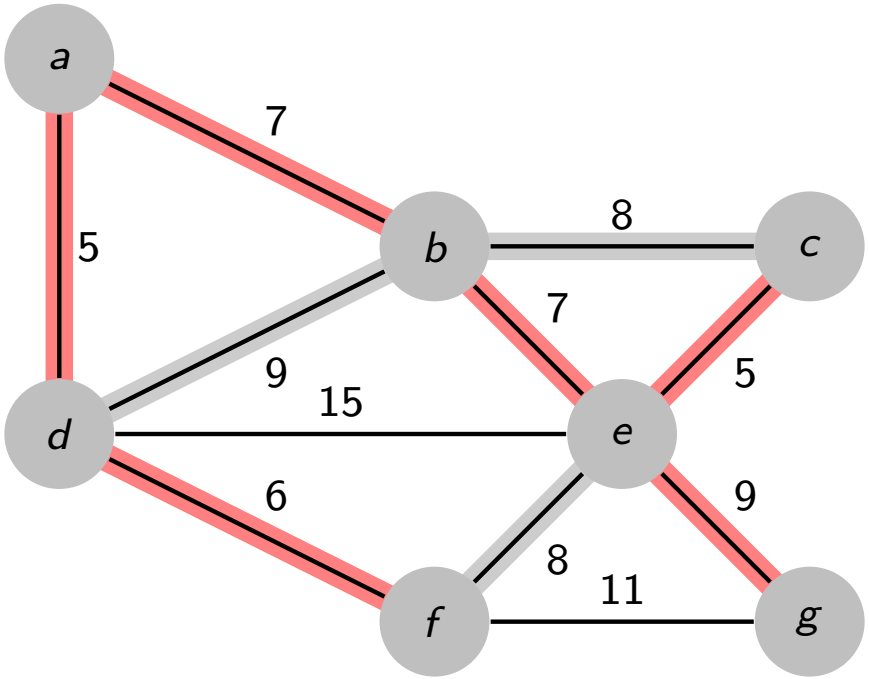
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

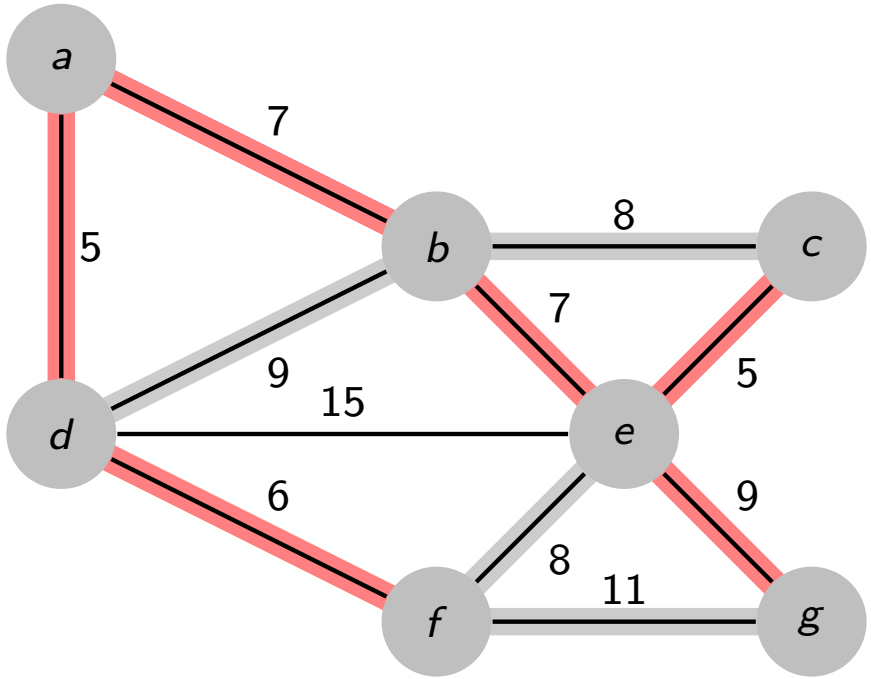
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

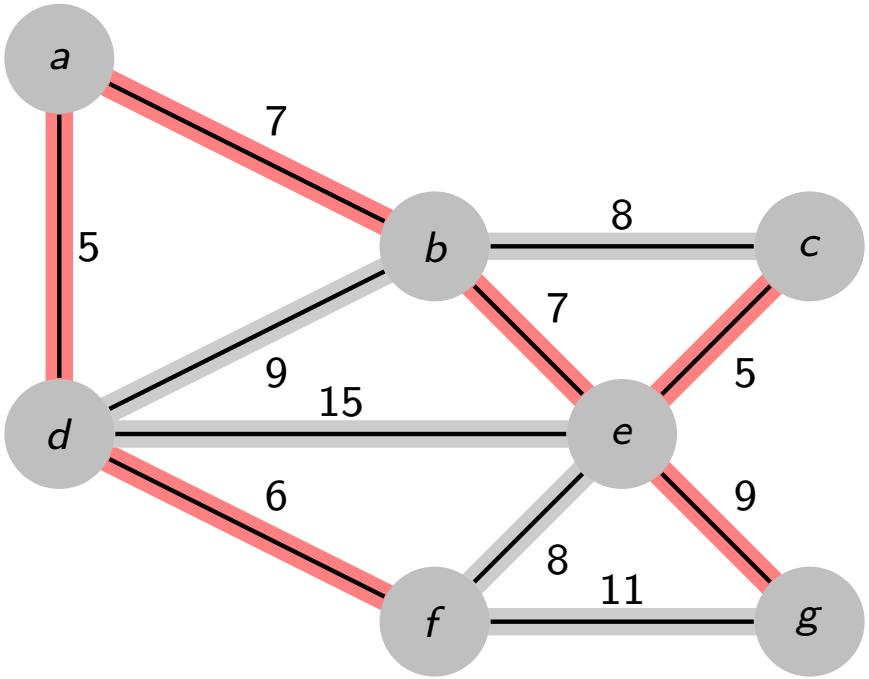
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

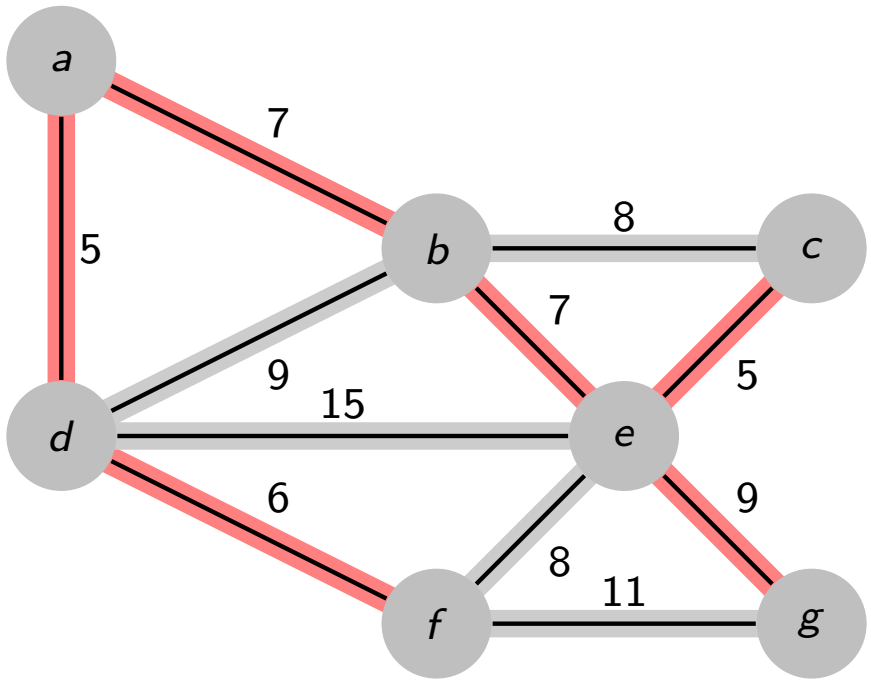
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?

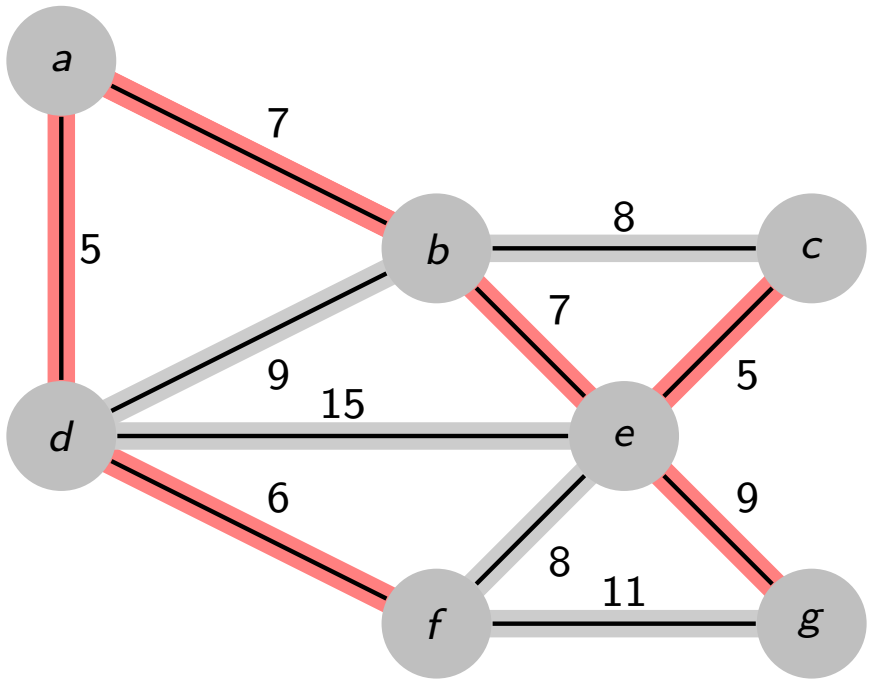


Exercise: this always gives a MST (or see Chapter 23 CLRS)

Kruskal's algorithm for Minimum Spanning Tree

Computes a Minimum Spanning Tree T using a *greedy* approach:

- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?

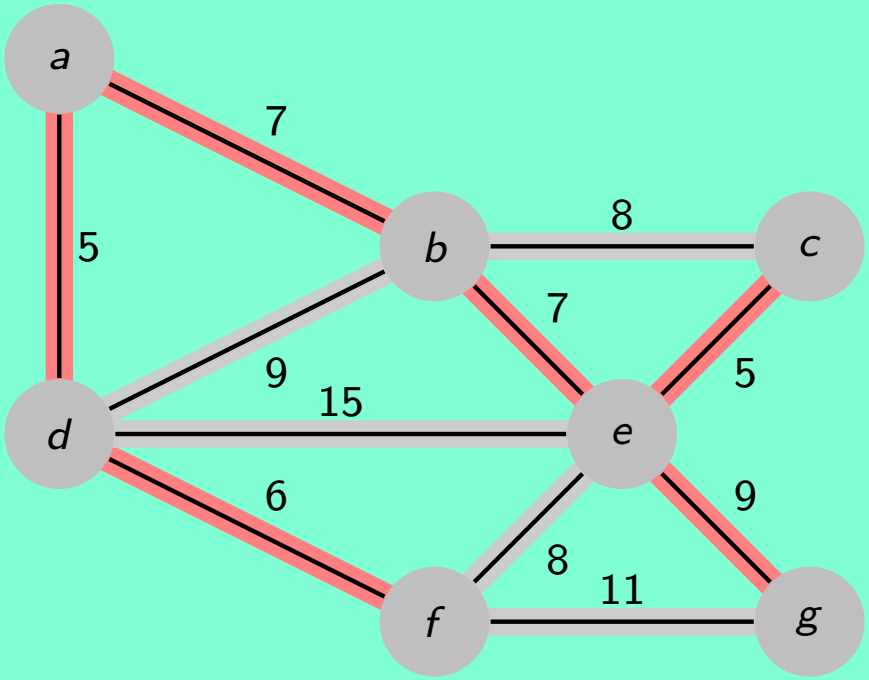


Exercise: this always gives a MST (or see Chapter 23 CLRS)
Run-time $O(|E|^2)$ (if implemented naïvely) $\Rightarrow$ **decision problem MST is in P**

Questions?

Computes a Minimum Spanning Tree T using a *greedy* approach:

- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Exercise: this always gives a MST (or see Chapter 23 CLRS)
Run-time $O(|E|^2)$ (if implemented naïvely) $\Rightarrow$ **decision problem MST is in P**

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class NP

A decision problem X lies in the complexity class NP, if

(See video on course website to learn more about Turing machines.)

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class NP

A decision problem X lies in the complexity class NP, if

- if it can be solved in polynomial time on a non-deterministic Turing machine (original, formal definition)

(See video on course website to learn more about Turing machines.)

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class NP

A decision problem X lies in the complexity class NP, if

- if it can be solved in polynomial time on a non-deterministic Turing machine (original, formal definition)
- (or, alternatively:) if it is solved by a *non-deterministic* algorithm with polynomial time complexity.

(See video on course website to learn more about Turing machines.)

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class NP

A decision problem X lies in the complexity class NP, if

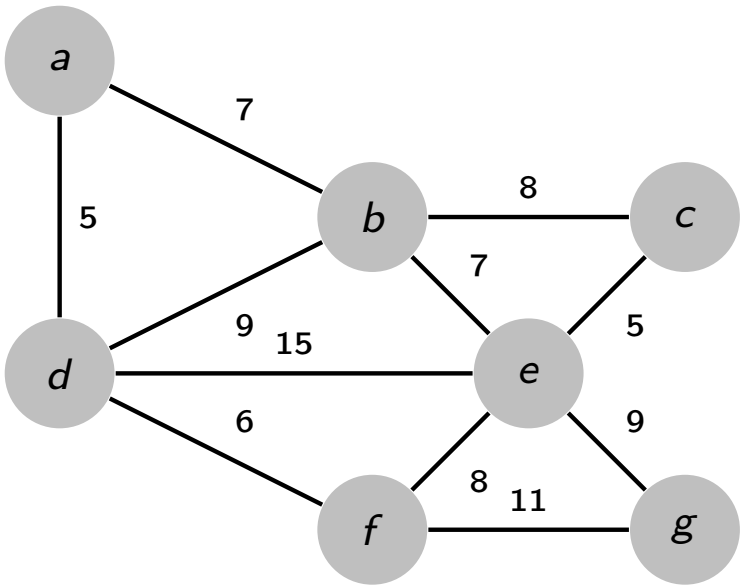
- if it can be solved in polynomial time on a non-deterministic Turing machine (original, formal definition)
- (or, alternatively:) if it is solved by a *non-deterministic* algorithm with polynomial time complexity.

Example: Traveling Salesman (decision version) is in NP.

(See video on course website to learn more about Turing machines.)

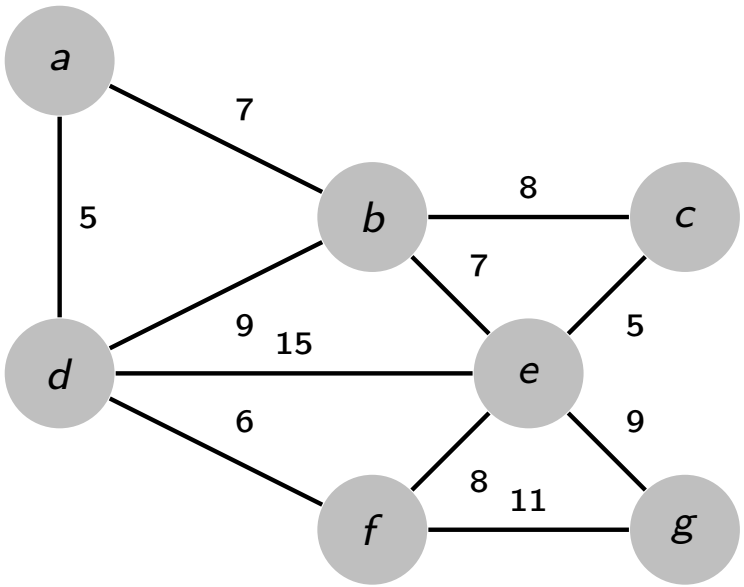
Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?



Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?



Non-deterministic algorithm for the TSP

Oracle:

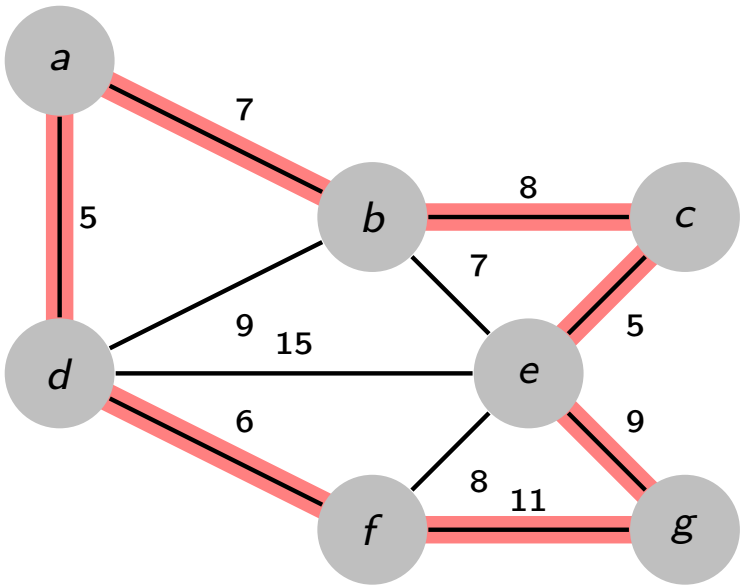
- Specify sequence of edges.

Verification:

- Verify that sequence forms a tour that visits all cities.
- Compute tour length.
- Is tour length $\leq B$?

Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?



Non-deterministic algorithm for the TSP

Oracle:

- Specify sequence of edges.

Verification:

- Verify that sequence forms a tour that visits all cities.
- Compute tour length.
- Is tour length $\leq B$?

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class NP

A decision problem X lies in the complexity class NP, if

- if it can be solved in polynomial time on a non-deterministic Turing machine (original, formal definition)
- (or, alternatively:) if it is solved by a *non-deterministic* algorithm with polynomial time complexity.

Example: Traveling Salesman (decision version) is in NP.

(See video on course website to learn more about Turing machines.)

Definition: Complexity class P

A decision problem X lies in the complexity class P,

- if it can be solved be a deterministic Turing machine in polynomial time (original, formal definition)
- (or, alternatively:) **if it is solved by an algorithm with polynomial time complexity** (definition that we use)

Definition: Complexity class NP

A decision problem X lies in the complexity class NP, if

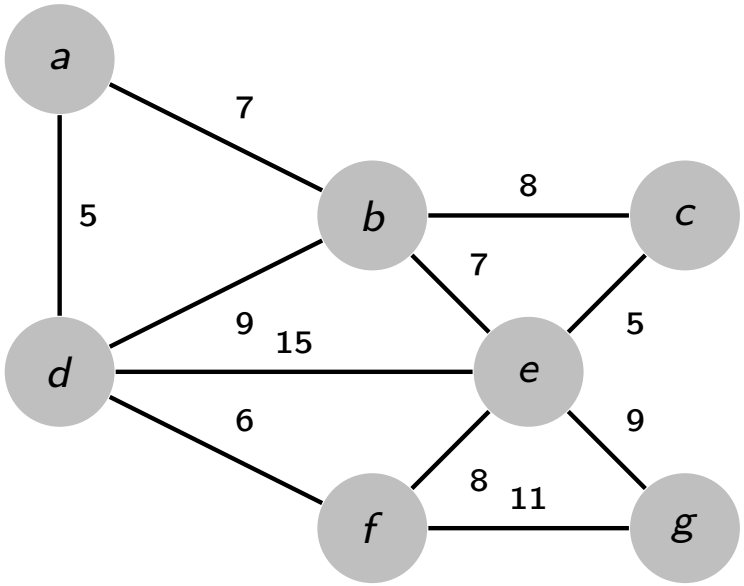
- if it can be solved in polynomial time on a non-deterministic Turing machine (original, formal definition)
- (or, alternatively:) if it is solved by a *non-deterministic* algorithm with polynomial time complexity.
- (or, alternatively:) if the YES-instances of X possess certificates of polynomial length that can be verified in polynomial time.

Example: Traveling Salesman (decision version) is in NP.

(See video on course website to learn more about Turing machines.)

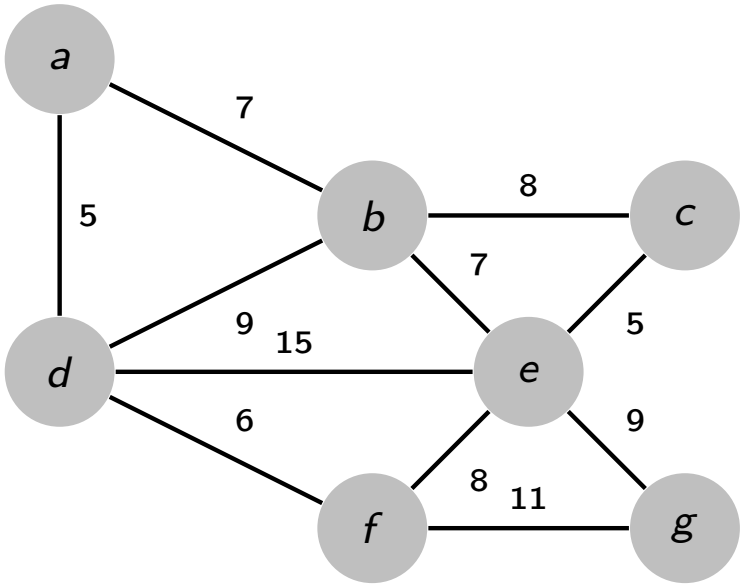
Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?



Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?

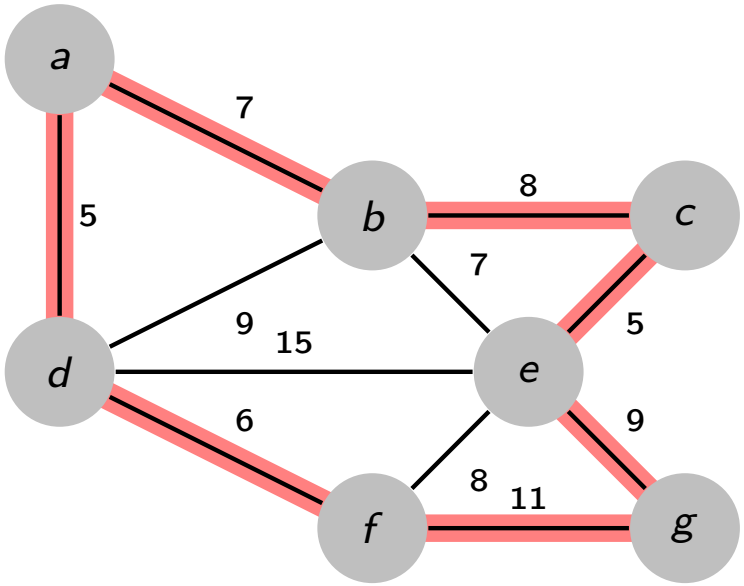


NP-certificate

- What is a NP-certificate for the TSP?
- How can it be verified in polynomial time?

Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?

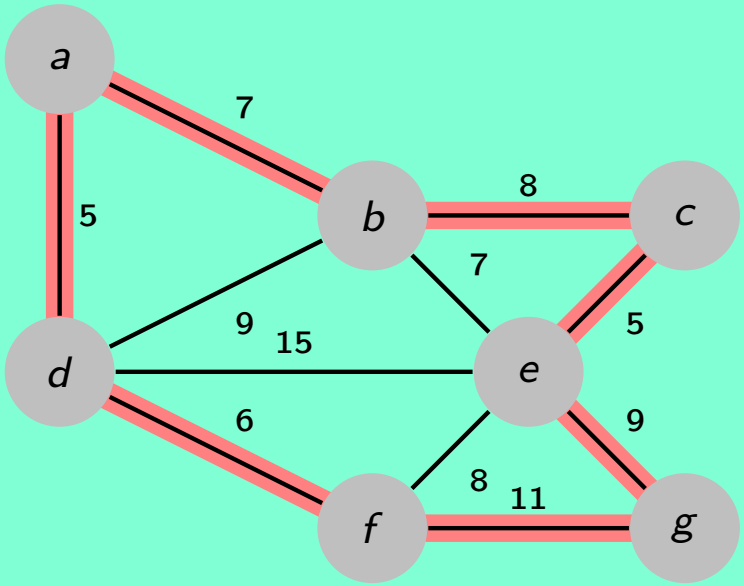


NP-certificate

- What is a NP-certificate for the TSP?
- How can it be verified in polynomial time?

Travelling Salesman Problem (TSP) - Decision version

Instance: cities $1, \ldots, n$; distances $d(i, j)$; a bound B
Question: does there exist a roundtrip of length at most B?



NP-certificate

- What is a NP-certificate for the TSP?
- How can it be verified in polynomial time?

Example: Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. x it 'true' $\Leftrightarrow$ $\neg x$ is false
- *Clause* over X: disjunction of literals $(l_1 \vee l_2 \vee \ldots l_j)$.

Example: Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. x it 'true' $\Leftrightarrow$ $\neg x$ is false
- *Clause* over X: disjunction of literals $(l_1 \vee l_2 \vee \ldots l_j)$.

Decision problem 'Satisfiability' (SAT)

Instance:
a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a set of clauses C over X

Question: does there exist a truth assignment for X that simultaneously satisfies all clauses in C?

Example: Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. x it 'true' $\Leftrightarrow$ $\neg x$ is false
- *Clause* over X: disjunction of literals $(l_1 \vee l_2 \vee \ldots l_j)$.

Decision problem 'Satisfiability' (SAT)

Instance:
a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a set of clauses C over X

Question: does there exist a truth assignment for X that simultaneously satisfies all clauses in C?

(special case 3-SAT: all clauses consist of 3 literals)

Example: Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. x it 'true' $\Leftrightarrow$ $\neg x$ is false
- *Clause* over X: disjunction of literals $(l_1 \vee l_2 \vee \ldots l_j)$.

Decision problem 'Satisfiability' (SAT)

Instance:
a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a set of clauses C over X

Question: does there exist a truth assignment for X that simultaneously satisfies all clauses in C?

(special case 3-SAT: all clauses consist of 3 literals)

Examples

$C_1 = \{(x \vee y \vee z), (\neg x \vee \neg y \vee \neg z)\}$
$C_2 = \{(x \vee y), (\neg x \vee y), (x \vee \neg y), (\neg x \vee \neg y)\}$

Example: Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \to \{true, false\}$
- *Literals:* We call x and $\neg x$ literals corresponding to variable $x \in X$. x it 'true' $\Leftrightarrow$ $\neg x$ is false
- *Clause* over X: disjunction of literals $(l_1 \vee l_2 \vee \ldots l_j)$.

Decision problem 'Satisfiability' (SAT)

Instance:
a set of logical variables $X := \{x_1, \ldots, x_n\}$ and a set of clauses C over X

Question: does there exist a truth assignment for X that simultaneously satisfies all clauses in C?

(special case 3-SAT: all clauses consist of 3 literals)

Examples

$C_1 = \{(x \vee y \vee z), (\neg x \vee \neg y \vee \neg z)\}$
$C_2 = \{(x \vee y), (\neg x \vee y), (x \vee \neg y), (\neg x \vee \neg y)\}$

Question

What's a good NP-certificate for SAT and how to verify it?