Algorithms and Complexity (AC)

First Lecture (11th of September, 2023)

Paul Bouman & Tom van der Zanden (Based on slides by Gerhard Woeginger, Jesper Nederlof and Marie Schmidt)

September – November 2023

LNMB – Landelijk Netwerk Mathematische Besliskunde

Table of contents

Introduction

Program for today

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

Introduction round

Let's do some short introductions. If you want, you can mention

- Who you are?
- Where you work and what you work on?
- Who you work with?

(Preliminary) program

Date	Who	Topic
11 Sep	PB	Introduction, algorithms, time complexity and computational models, P versus NP
18 Sep	PB	Reductions, NP-hardness and NP-completeness
25 Sep	PB	Pseudopolynomial time, strong/weak NP-hardness, co-NP
2 Oct	PB	Approximation algorithms
2 Oct	You	**Deadline Exercise set 1**
9 Oct	PB	More on approximation algorithms
16 Oct	TvdZ	Exact algorithms for NP-hard problems
16 Oct	You	**Deadline Exercise set 2**
23 Oct	TvdZ	More exact algorithms for NP-hard problems
30 Oct	TvdZ	Treewidth
30 Oct	You	**Deadline Exercise set 3**
6 Nov	TvdZ	Randomized algorithms
13 Nov		**No lecture**
13 Nov	You	**Deadline Exercise set 4**

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

Lectures and Attendance

Lectures 1, 5 and 9 are not offered online. The other lectures are hybrid or fully online.

The room in Utrecht is always available during lectures.

Lectures by PB will all be on-campus or hybrid.

LNMB rule:

- Making the exercises: 4 EC
- Attendance only: 1 EC

For *attendance only* you can miss at most two lectures. Lectures taught on-campus must be attended in person. During online-only lectures your camera must be on to be considered present.

Exercises

1. First series: deadline 2-Oct-2023
   You may want to wait 1 week before solving them
   Please email to bouman@ese.eur.nl
2. Second series: deadline 16-Oct-2023
   Please email to bouman@ese.eur.nl
3. Third series: deadline 30-Oct-2023,
4. Fourth series: deadline 13-Nov-2023

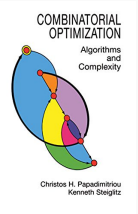
May be solved in groups of 2.

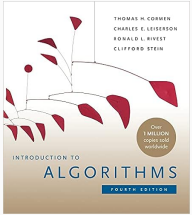
Strong advice: do indeed solve them in groups of two! We can match during the coffee break.

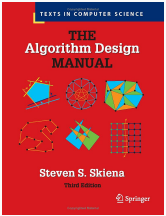
The final two sets will be handled by Tom van der Zanden.

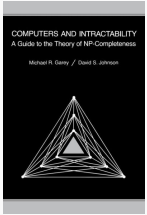
Books

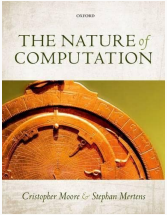
There is no required literature for this course. There are many useful resources for this course.











Combinatorial Optimization (CO) Cheap book covers many topics of this course.

Introduction to Algorithms (CLRS) Popular introductory textbook. 3rd edition is cheaper.

The Algorithm Design Manual Both introductory book on algorithms and a catalog of algorithmic problems.

Computers and Intractability The original but technical book on the topic. Contains a catalog of problems.

The Nature of Computation Modern book with modern intuitive explanations of computational complexity.

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

Algorithms

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

```
def insertion_sort(seq):
    n = len(seq)
    for j in range(1, n):
        key = seq[j]
        i = j-1
        while i>=0 and seq[i] > key:
            seq[i+1] = seq[i]
            i -= 1
        seq[i+1] = key
    return seq
```

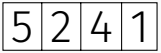
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



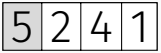
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



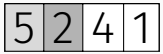
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



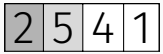
Example: Insertion Sort

Example: input sequence *[5,2,4,1]*

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



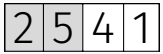
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



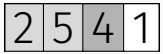
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



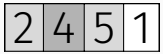
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



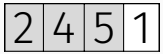
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



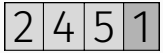
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



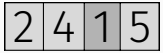
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



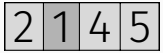
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



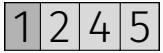
Example: Insertion Sort

Example: input sequence `[5,2,4,1]`

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



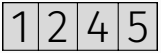
Example: Insertion Sort

Example: input sequence *[5,2,4,1]*

Light gray: part of the sequence that has been sorted

Dark gray: current key to be inserted in the sorted part of the sequence

White: part of the sequence still unsorted



What is this course about? - Algorithms...

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?

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 $\triangleq$ 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?

Example: Insertion Sort

```
def insertion_sort(seq):
    n = len(seq)
    for j in range(1, n):
        key = seq[j]
        i = j-1
        while i>=0 and seq[i] > key:
            seq[i+1] = seq[i]
            i -= 1
        seq[i+1] = key
    return seq
```

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

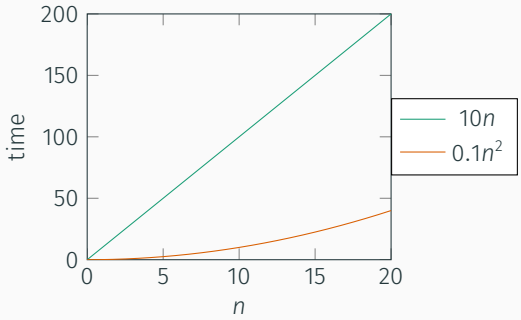
1. In this course, we measure running time as a *function of the input length n*.
2. In this course, we consider worst-case running time, i.e., we answer the question: what is the maximum running time over all possible algorithm inputs of length n?
3. We make use of big-O notation.

big-O notation

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

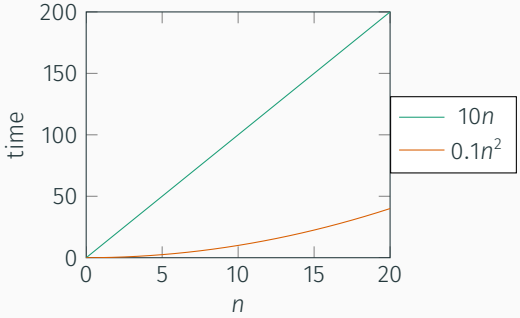
For example, $4n^2 + 3n \in O(n^2)$ and $7n^2 + 2 \in O(n^2)$

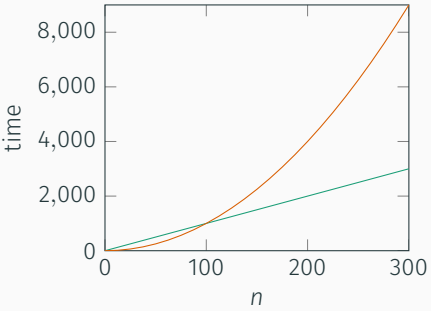
Comparing running times



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

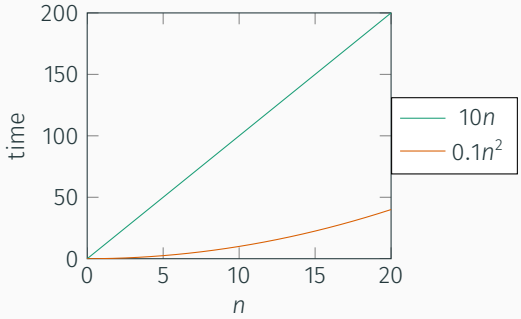
Comparing running times

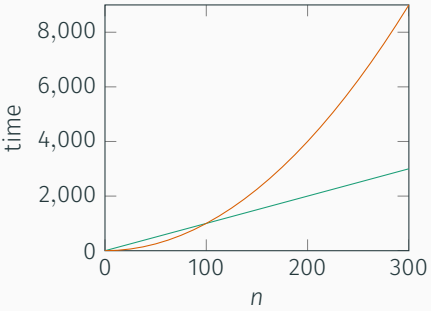




What is better: an algorithm that solves your problem in time $O(n)$, or one that solves it in time $O(n^2)$?

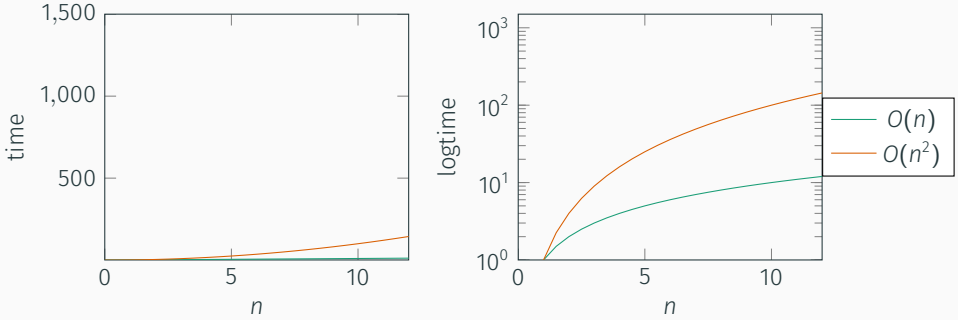
Comparing running times





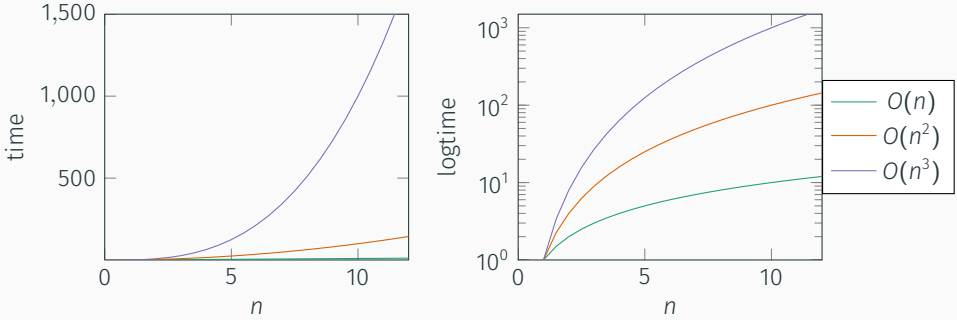
What is better: an algorithm that solves your problem in time $O(n)$, or one that solves it in time $O(n^2)$? $\rightarrow$ it depends!

Comparing running times



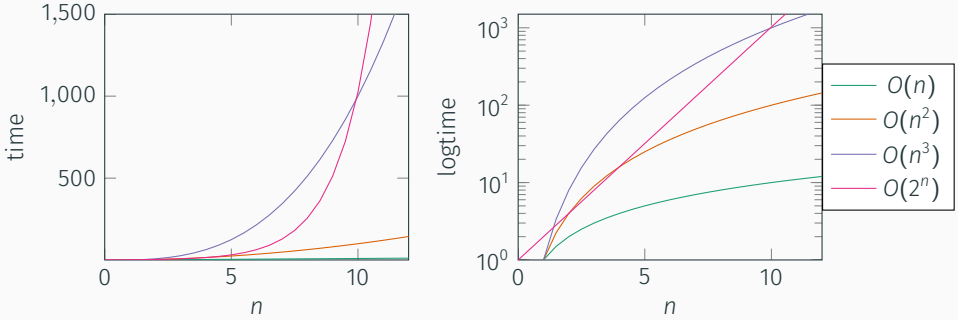
As we are most worried about solving large problem instances:
We consider $O(n)$ to be better than $O(n^2)$,

Comparing running times



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



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*

Time complexity of algorithms

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

We call an algorithm a *exponential-time algorithm*, if there is some b for which the algorithm's worst case running time is in $O(b^{f(n)})$ where $f(n)$ is some function of n.

Vocabulary

Time complexity of algorithms

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

We call an algorithm a *exponential-time algorithm*, if there is some b for which the algorithm's worst case running time is in $O(b^{f(n)})$ where $f(n)$ is some function of n.

More refined notions exist, such as *factorial time* (e.g. $O(n!)$) or *logarithmic time* (e.g. $O(\log n)$).

The focus in this course is on *polynomial time* versus *exponential time* and we take some liberties accordingly.

Exponential-time algorithms

Exponential time algorithms are considered as ’inefficient’ by many people

Intuition

Polynomial = desirable, good, harmless, fast, short, small

Exponential = undesirable, bad, evil, slow, wasteful, horrible

Exponential-time algorithms

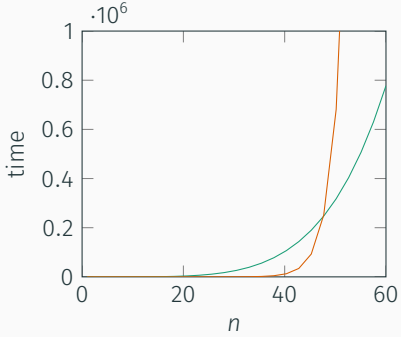
Exponential time algorithms are considered as 'inefficient' by many people

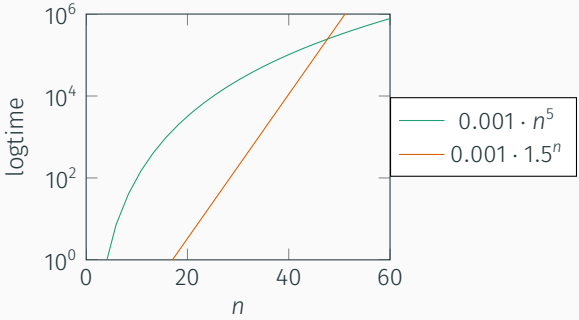
Intuition

Polynomial = desirable, good, harmless, fast, short, small

Exponential = undesirable, bad, evil, slow, wasteful, horrible

But: for smaller/specific problem instances they can be preferable! (discussed by TvdZ)





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'

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

Space needed

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

Space complexity in the RAM model

- As reading and writing uses elementary operations, *time complexity* gives an upper bound on *time complexity* in the RAM-model of computation.
- Inspecting all input data requires $O(n)$ time and is typically required in he worst case.

Because of this we mostly focus on the time complexity of algorithms and problems.

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?

System	Encoded	Expanded
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?

System	Encoded	Expanded
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 $n \cdot \log_2(10)$ bits in binary.

Does space complexity depend on the encoding?

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

System	Encoded	Expanded
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 $n \cdot \log_2(10)$ bits in binary.

Does space complexity depend on the encoding?

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

System	Encoded	Expanded
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 $n \cdot \log_2(10)$ bits in binary.

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

Problem Complexity

Problem complexity

Problems are specified in terms of

Problem input: What is given

Problem output: What does the expected answer look like

Example: Sorting

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 does the expected answer look like

Example: Sorting

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 does the expected answer look like

Example: Sorting

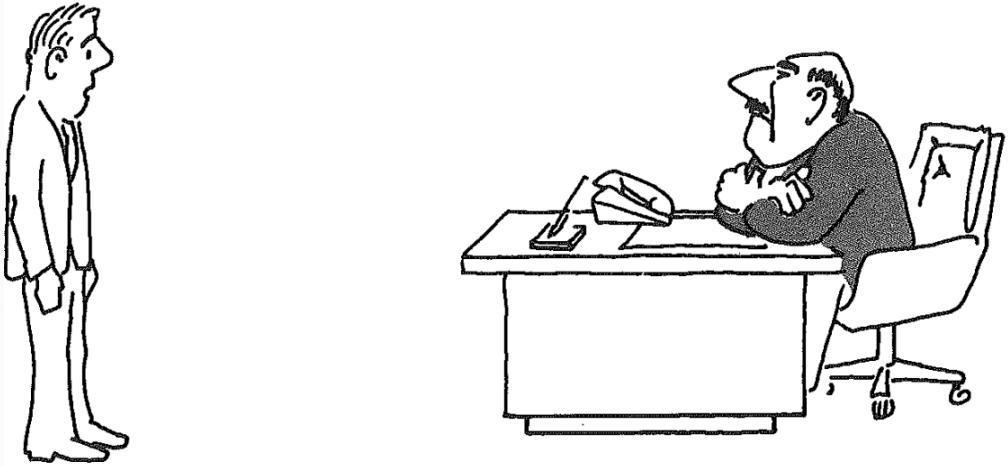
Input: A sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A permutation $(a'_1, a'_2, \ldots, a'_n)$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

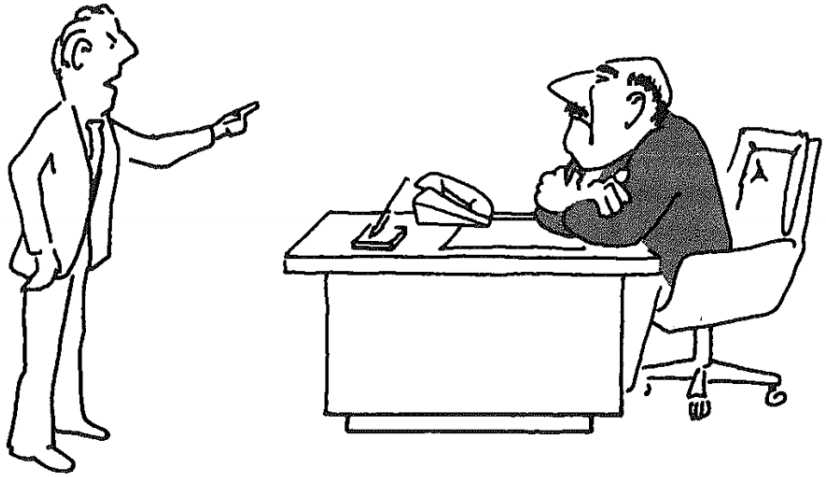
Often there are many different algorithms to solve a problem

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)

Instance

We sometimes refer to a particular *input* for a problem as an *instance*.
Example instances for sorting: `[5,2,4,1]` or `[12,3,2,5,4,2]`

Problem

One way to look at a *problem* is as an infinite set of instances.
When encoded using discrete symbols, this is a *countable infinite set*.

Some terminology

Instance

We sometimes refer to a particular *input* for a problem as an *instance*.
Example instances for sorting: `[5,2,4,1]` or `[12,3,2,5,4,2]`

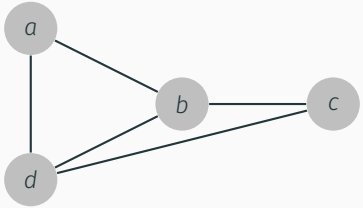
Problem

One way to look at a *problem* is as an infinite set of instances.
When encoded using discrete symbols, this is a *countable infinite set*.

In case of decision problems, we sometimes talk about YES-instances and NO-instances.

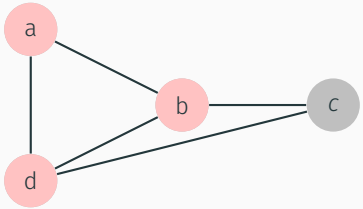
Some texts refer to *strings* or *sentences* rather than *inputs* or *instances*.

Decision problems and optimization problems



Reminder 'clique': set of pairwise adjacent vertices

Decision problems and optimization problems



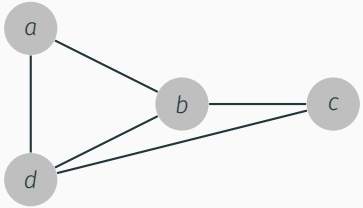
Reminder 'clique': set of pairwise adjacent vertices

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

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary search on the interval of objective values

Observation

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary search on the interval of objective values

Example

Let G be a graph on n vertices.

Decision problems and optimization problems

Observation

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary 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

Decision problems and optimization problems

Observation

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary 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

Decision problems and optimization problems

Observation

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary 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

Decision problems and optimization problems

Observation

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary 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

Decision problems and optimization problems

Observation

Discrete optimization problem can usually be rewritten into a sequence of decision problems.

Use bisection search / binary 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 $\rightarrow$ fast and simple

Problem classes P and NP

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, slides and/or a video will be provided on the site.

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 $\sum_{e \in T} w_e \leq W$

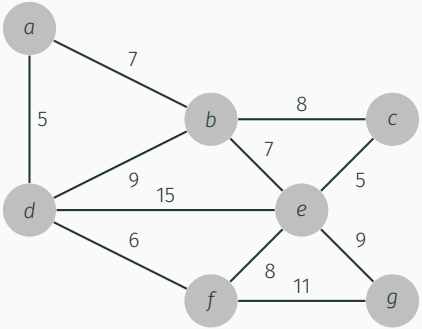
Terminology:

- 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:

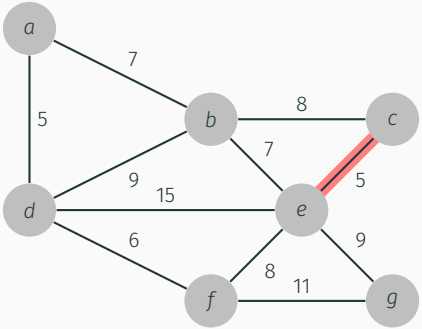
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

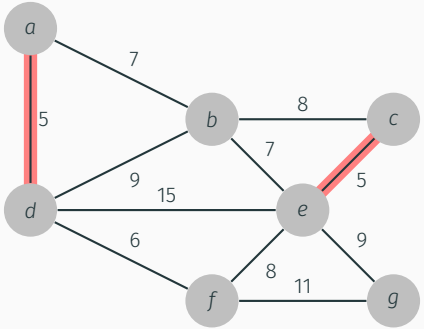
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

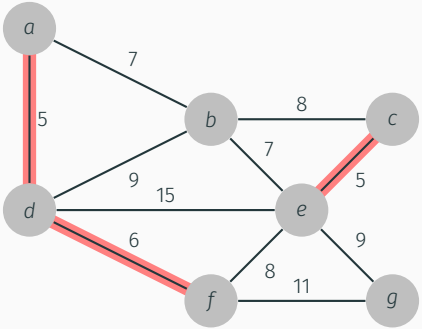
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

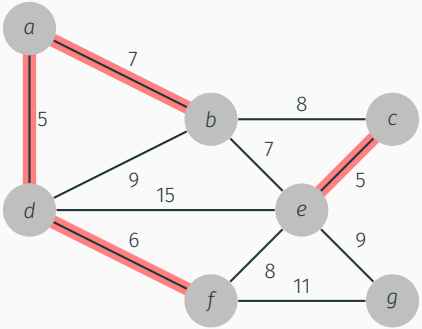
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

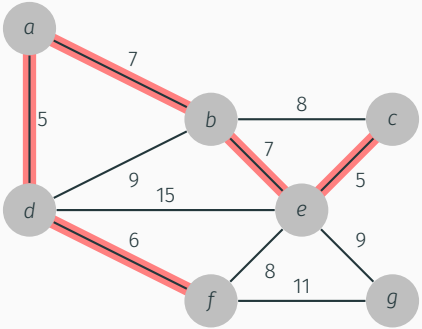
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

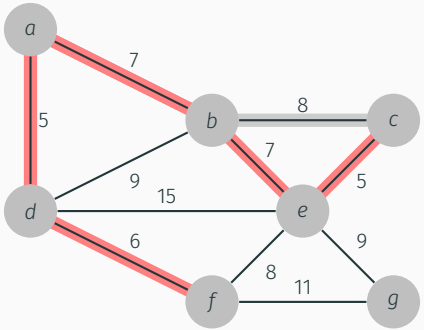
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

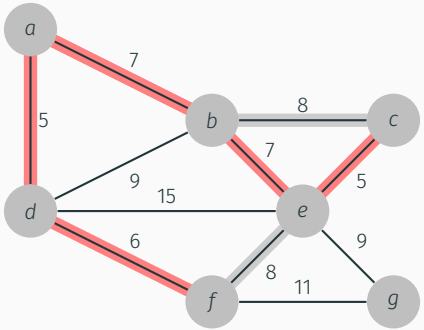
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

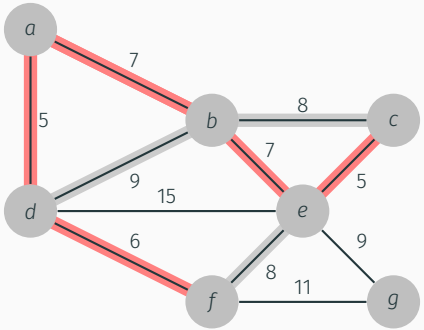
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

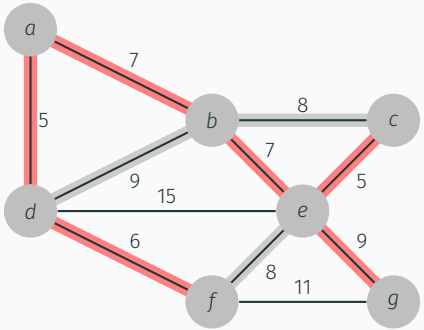
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

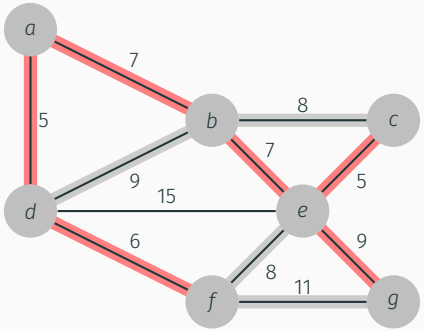
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

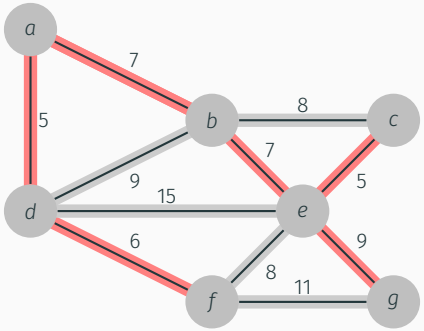
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



Kruskal's algorithm for Minimum Spanning Tree

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

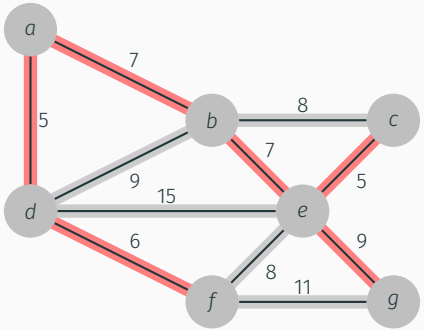
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle in T.
- Check: is $\sum_{e \in T} w_e \leq W$?



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 $\sum_{e \in T} w_e \leq W$?

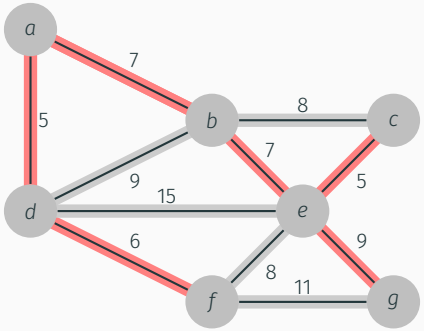


Exercise: this always gives a MST (or see CO/CLRS)

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 CO/CLRS)
Run-time $O(|E|^2)$ (if implemented naïvely) $\Rightarrow$ **decision problem MST is in P**

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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)

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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.)

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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.)

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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.)

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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.

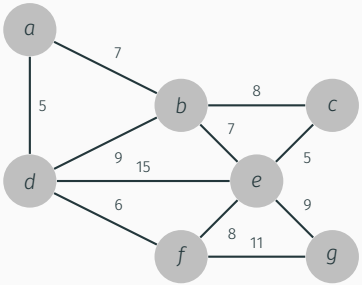
(See video on course website to learn more about Turing machines.)

Example: Travelling Salesman Problem

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?

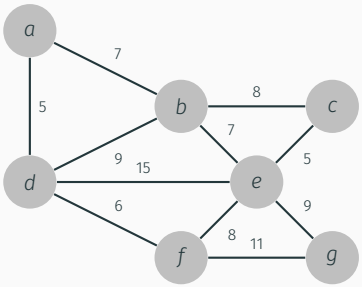


Example: Travelling Salesman Problem

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:

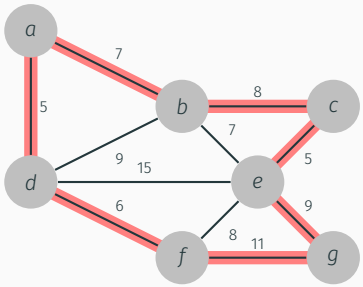
- Verify that sequence forms a tour that visits all cities.
- Compute tour length.
- Is tour length $\leq B$?

Example: Travelling Salesman Problem

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 $\leq B$?

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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.)

Complexity classes P and NP

Definition: Complexity class P

A decision problem *X* lies in the complexity class P,

- if it can by 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.

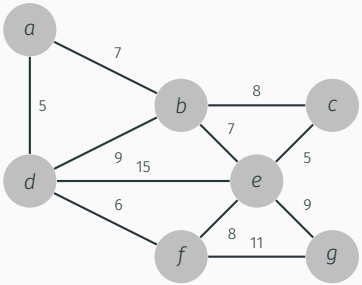
(See video on course website to learn more about Turing machines.)

Example: Travelling Salesman Problem

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?

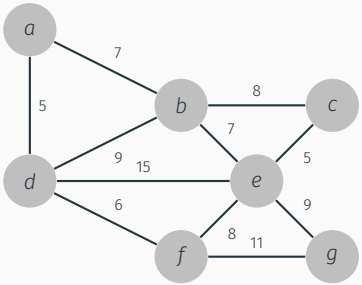


Example: Travelling Salesman Problem

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

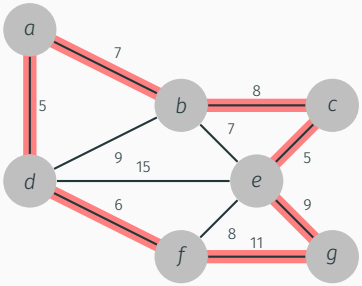
- What is a NP-certificate for the TSP?
- How can it be verified in polynomial time?

Example: Travelling Salesman Problem

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*: $t : X \to \{T, F\}$ (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 $(l_1 \vee l_2 \vee \ldots l_j)$.

Truth Tables

Truth Assignment		Expressions		
x_1	x_2	$\neg x_1$	$x_1 \wedge x_2$	$x_1 \vee x_2$
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

Example: Satisfiability

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

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

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

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?

Wrap Up

Next week we continue our discussion of P and NP, and discuss problem-reductions.

The million dollar question

P=NP?

Implication: verification in poly-time would imply we can determine answers in poly-time.