Algorithms and Complexity (AC)

Marie Schmidt

(Based on slides by Gerhard Woeginger and Jesper Nederlof)

Landelijk Netwerk Mathematische Besliskunde

LNMB, Sep–Nov 2019

(Preliminary) program

9 Sep	: Introduction, basic concepts, time complexity and computational models, P versus NP
16 Sep	: reductions, NP-hardness and NP-completeness
23 Sep	: Pseudopolynomial time, strong/weak NP-hardness, co-NP
30 Sep	: Exercise set 1
30 Sep	: Approximation algorithms
7 Oct	: More on approximation algorithms
14 Oct	: Exercise set 2
14 Oct	: Exact algorithms for NP-hard problems
21 Oct	: More exact algorithms for NP-hard problems
28 Oct	: Exercise set 3
28 Oct	: Treewidth
4 Nov	: Randomized algorithms
11 Nov	: Exercise set 4
11 Nov	: **No lecture!!**

Website: `http://www.win.tue.nl/~jnederlo/LNMB/`

First 5 lectures: Marie Schmidt (`schmidt2@rsm.nl`), last 4 lectures: Jesper Nederlof (`j.nederlof@tue.nl`)

Program for the first three weeks

- Basic definitions: decision problems, graphs
- computational models and (worst-case) time complexity
- 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

And maybe more. . . ?

Algorithm

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

Algorithm

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

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
$a'_i \leq a'_2 \leq \ldots \leq a'_n$

Algorithm

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

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
$a'_i \leq a'_2 \leq \ldots \leq a'_n$

Algorithm

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

Example: Insertion Sort - for a human

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 $a'_i \leq a'_2 \leq \ldots \leq a'_n$

Set $A := (a_1)$
for $i = 2, \ldots, n$ **do**
  update A by inserting a_i at the 'correct' position in sorted sequence A
end for
return A

Algorithm

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

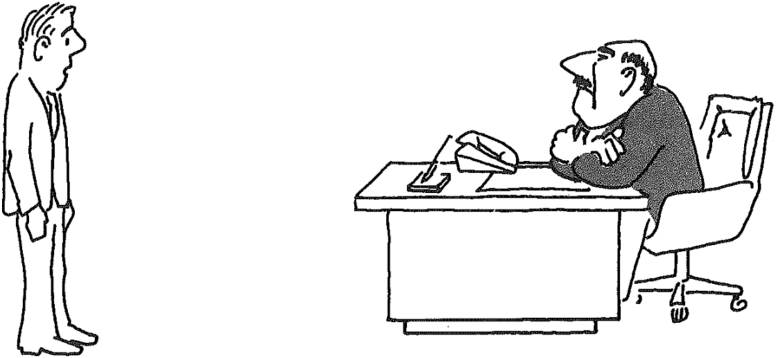
Example: Insertion Sort - for a machine

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
$a'_i \leq a'_2 \leq \ldots \leq a'_n$

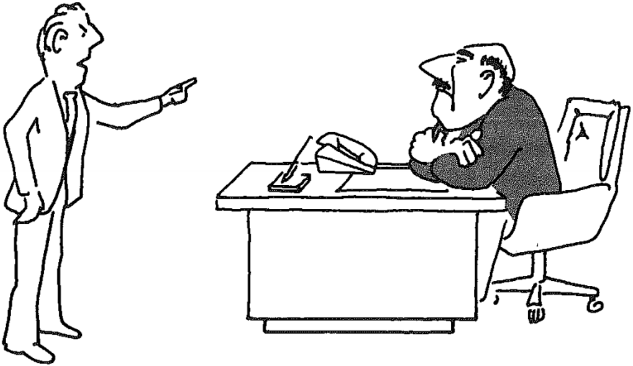
Set $A := (a_1)$
for $i = 2, \ldots, n$ **do**
  $key := A[j]$
  $i := j - 1$
  while $i > 1$ and $A[i] > key$ **do**
    $A[i+1] := A[i]$
    $i := i - 1$
  end while
  $A[i+1] := key$
end for
return A

When we analyze an algorithm, we are interested in:

- running time of the algorithm
- space (memory) needed by the algorithm (probably not treated in this course)
- for optimization problems: quality of the output
    - exact algorithm
    - approximation algorithm
    - heuristic algorithm



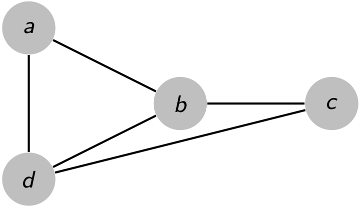
“I can’t find an efficient algorithm, I guess I’m just too dumb.”



“I can’t find an efficient algorithm, because no such algorithm is possible!”

Basic concepts: Graphs

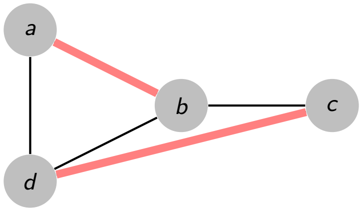
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges

Basic concepts: Graphs

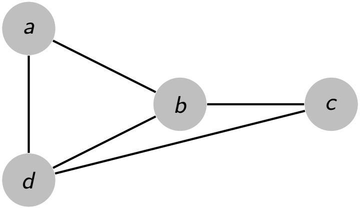
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges

Basic concepts: Graphs

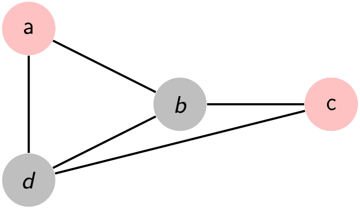
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges
Independent Set: set of pairwise non-adjacent vertices

Basic concepts: Graphs

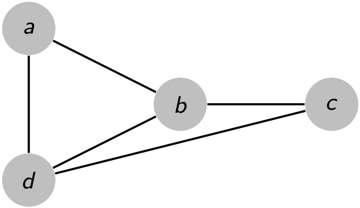
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges
Independent Set: set of pairwise non-adjacent vertices

Basic concepts: Graphs

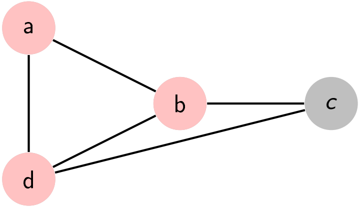
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges
Independent Set: set of pairwise non-adjacent vertices
Clique: set of pairwise adjacent vertices

Basic concepts: Graphs

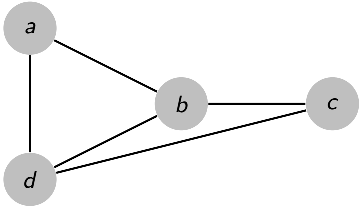
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges
Independent Set: set of pairwise non-adjacent vertices
Clique: set of pairwise adjacent vertices

Basic concepts: Graphs

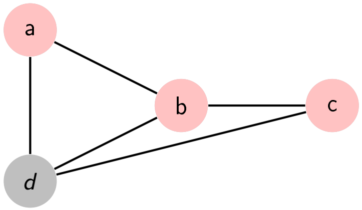
Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges
Independent Set: set of pairwise non-adjacent vertices
Clique: set of pairwise adjacent vertices
Vertex Cover: set of vertices such that each edge is incident to at least one vertex from the set

Basic concepts: Graphs

Graph: pair (V, E) where V is set of *vertices* and E is a set of pairs of vertices called *edges*



Matching: set of non-adjacent edges (no two edges share a vertex), *perfect* if $|V|/2$ edges
Independent Set: set of pairwise non-adjacent vertices
Clique: set of pairwise adjacent vertices
Vertex Cover: set of vertices such that each edge is incident to at least one vertex from the set
Much more terminology: cycles, Hamiltonian cycles, trees, forests,...

Problems

Problem instance:

- specification of problem data

Problems

Problem instance:

- specification of problem data

Example: Instance of decision version of clique

$V = \{a, b, c, d, e, f, g\}$; $k = 4$
$E = \{\{a, b\}, \{a, d\}, \{b, c\}, \{c, d\}, \{b, d\}, \{b, e\}, \{c, e\}, \{d, e\},$
$\{d, f\}, \{e, f\}, \{e, g\}, \{f, g\}\}$;

Basic concepts: Input size and asymptotics

Problem size:
- length (number of symbols) of reasonable encoding of instance (often denoted as n)

Basic concepts: Input size and asymptotics

Problem size:
• length (number of symbols) of reasonable encoding of instance (often denoted as n)

Example for encodings

- Graph: adjacency list; adjacency matrix
- Set: list of elements; bit vector
- Number: decimal; binary; hex; unary

We do not really care whether an n-vertex graph is encoded with $4n^2 + 3n$ or with $7n^2 + 2$ symbols.

We do not really care whether an n-vertex graph is encoded with $4n^2 + 3n$ or with $7n^2 + 2$ symbols.

big-Oh notation

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

We do not really care whether an n-vertex graph is encoded with
$4n^2 + 3n$ or with $7n^2 + 2$ symbols.

big-Oh 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 $7n^2 + 2 \in O(n^2)$

We do not really care whether an n-vertex graph is encoded with $4n^2 + 3n$ or with $7n^2 + 2$ symbols.

big-Oh 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 $7n^2 + 2 \in O(n^2)$

big-Omega, big-Theta

$f(n)$ is $\Omega(g(n))$ denotes that $\exists n_0, C$ such that $\forall n > n_0$ $f(n) \geq C \cdot g(n)$.
$f(n)$ is $\Theta(g(n))$ denotes that $f(n)$ is $O(g(n))$ and $\Omega(g(n))$.

Different types of algorithmic problems:

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

Different types of algorithmic problems:

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

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?

Different types of algorithmic problems:

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

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?

Example: Decision problem CLIQUE

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

Different types of algorithmic problems:

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

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?

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 (neither optimization nor decision problem) SORTING

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

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

Time complexity of an algorithm

Time complexity of an algorithm

number of elementary steps an algorithm makes

Time complexity of an algorithm

Time complexity of an algorithm

number of elementary steps an algorithm makes
$\rightarrow$ depends on 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

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' (often: in input size n: e.g., numbers $\leq c \cdot \log n$ for a constant c)

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' (often: in input size n: e.g., numbers $\leq c \cdot \log n$ for a constant c)

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' (often: in input size n: e.g., numbers $\leq c \cdot \log n$ for a constant c)

Why do we use RAM:

- similar to how a computer works & approximates running time of computer well
- easier to analyze than many alternatives

Time complexity of an algorithm

number of elementary steps an algorithm makes
$\rightarrow$ here (and in most other places): using RAM

Time complexity of an algorithm

number of elementary steps an algorithm makes
$\rightarrow$ here (and in most other places): using RAM
$\rightarrow$ normally: specified in relation to input length (n) using a *reasonable* encoding

Time complexity of an algorithm

number of elementary steps an algorithm makes
$\rightarrow$ here (and in most other places): using RAM
$\rightarrow$ normally: specified in relation to input length (n) using a *reasonable* encoding
$\rightarrow$ normally: specified in O (or Θ-notation)

Time complexity of an algorithm

number of elementary steps an algorithm makes
$\rightarrow$ here (and in most other places): using RAM
$\rightarrow$ normally: specified in relation to input length (n) using a *reasonable* encoding
$\rightarrow$ normally: specified in O (or Θ-notation)
$\rightarrow$ here: *worst-case* complexity of an algorithm: the maximum number of steps for *any* input of length n

Time complexity of an algorithm

number of elementary steps an algorithm makes
→ here (and in most other places): using RAM
→ normally: specified in relation to input length (n) using a *reasonable* encoding
→ normally: specified in O (or Θ-notation)
→ here: *worst-case* complexity of an algorithm: the maximum number of steps for *any* input of length n

BUT: there are alternatives, e.g.,

- alternative computational models
- time complexity in *output length*
- average case time complexity

What is the worst-case time complexity of InsertionSort?

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

$a'_i \leq a'_2 \leq \ldots \leq a'_n$

Set $A := (a_1)$
for $i = 2, \ldots, n$ **do**
    $key := A[j]$
    $i := j - 1$
    while $i > 1$ and $A[i] > key$ **do**
        $A[i+1] := A[i]$
        $i := i - 1$
    end while
    $A[i+1] := key$
end for
return A

Big-Oh notation

Both for encoding length, and for time complexity, we make use of big-Oh notation.

big-Oh 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-Oh notation

Both for encoding length, and for time complexity, we make use of big-Oh notation.

big-Oh 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 $7n^2 + 2 \in O(n^2)$

Note: Determining / proving the worst-case time complexity of an algorithm can be difficult!

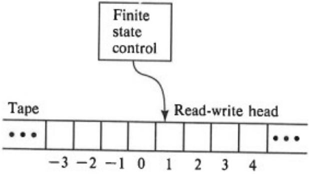
Turing machines

- Alternative mathematical models of computation
- Used in the definition of complexity classes P and NP

Turing machines

- Alternative mathematical models of computation
- Used in the definition of complexity classes P and NP

Not this.

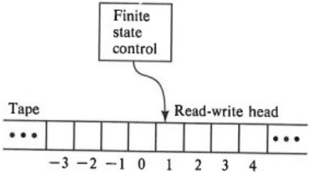


But this!

Deterministic one-tape Turing machine (DTM)

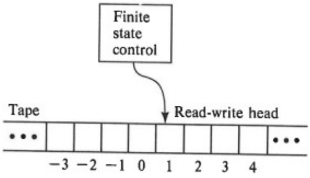
A DTM consists of

1. a *finite state control*
2. a *read-write head*
3. a *tape*: two-way infinite sequence of tape squares



A program for a DTM specifies:

1. a finite set Γ of tape symbols, including a subset $\Sigma \subset \Gamma$ of *input symbols* and a distinguished *blank symbol* $b \in \Gamma \setminus \Sigma$
2. a finite set Q of *states*, inclusing a distinguished *start state* q_0 and two distinguished *halt states* q_Y and q_N
3. a *transition function* $\delta : (Q \setminus \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$



Operation of a DTM program

Input: finite string $x \in \Sigma$

Initialize: write string in tape squares 1 to $|x|$, one symbol per square (all other tape squares are blank), state $q = q_0$, read-write head scans tape square 1

while $q \notin \{q_Y, q_N\}$ **do**

look up $(q', s'\Delta) := \delta(q, s)$ for current state q and read-write head pointing at square with symbol s

erase s

write s' in its place

move one square to the left if $\Delta = -1$, one square to the right if $\Delta = 1$

set $q := q'$

end while

if $q = q_Y$ **then**

return YES

else

return NO

end if

Each iteration of the while-loop counts as a step

A program for a DTM machine

$\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$, $Q = \{q + 0, q_1, q_2, q_Y, q_N\}$

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

$\delta(q, s)$

- Let's try this out!

A program for a DTM machine

$\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$, $Q = \{q + 0, q_1, q_2, q_Y, q_N\}$

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

$\delta(q, s)$

- Let's try this out!
- What does this program do?

A program for a DTM machine

$\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$, $Q = \{q + 0, q_1, q_2, q_Y, q_N\}$

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

$\delta(q, s)$

- Let's try this out!
- What does this program do?
- How many steps do we need?

A program for a DTM machine

$\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$, $Q = \{q + 0, q_1, q_2, q_Y, q_N\}$

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

$\delta(q, s)$

- Let's try this out!
- What does this program do?
- How many steps do we need?
- How many steps would we need at most?

A program for a DTM machine

$\Gamma = \{0, 1, b\}$, $\Sigma = \{0, 1\}$, $Q = \{q + 0, q_1, q_2, q_Y, q_N\}$

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

$\delta(q, s)$

- Let's try this out!
- What does this program do?
- How many steps do we need?
- How many steps would we need at most?
- How much space do we need (at most)?

- Would you rather own a RAM, or a DTM?

- Would you rather own a RAM, or a DTM?

Equivalence of computational models

A RAM and a DTM are equivalent in the sense that any function that can be computed on a DTM can be computed on a RAM, and vice versa.

- Would you rather own a RAM, or a DTM?

Equivalence of computational models

A RAM and a DTM are equivalent in the sense that any function that can be computed on a DTM can be computed on a RAM, and vice versa.

Church-Turing thesis

Anything that can be calculated by an *effective method* can be computed by a deterministic Turing machine.

Non-deterministic Turing machine

Non-deterministic Turing machine (NDTM)

1. guessing module: write-only head
2. checking module: deterministic Turing machine

A program for a DTM specifies:

exactly the same as a DTM program:

1. finite set of tape symbols Γ of tape symbols, including blank symbol
2. finite set Q of *states*, i
3. *transition function* δ

A program for a DTM specifies:

exactly the same as a DTM program:

1. finite set of tape symbols Γ of tape symbols, including blank symbol
2. finite set Q of *states*, i
3. *transition function* δ

Operation of a NDTM program

- write input string in tape squares 1 to $|x|$

A program for a DTM specifies:

exactly the same as a DTM program:

1. finite set of tape symbols Γ of tape symbols, including blank symbol
2. finite set Q of *states*, i
3. *transition function* δ

Operation of a NDTM program

- write input string in tape squares 1 to $|x|$
- guessing module: writes finite string of symbols from Γ in left tape squares starting from -1 in arbitrary manner

A program for a DTM specifies:

exactly the same as a DTM program:

1. finite set of tape symbols Γ of tape symbols, including blank symbol
2. finite set Q of *states*, i
3. *transition function* δ

Operation of a NDTM program

- write input string in tape squares 1 to $|x|$
- guessing module: writes finite string of symbols from Γ in left tape squares starting from -1 in arbitrary manner
- checking module: operates like a DTM

Operation of a NDTM program

- write input string in tape squares 1 to $|x|$
- guessing module: writes finite string of symbols from Γ in tape squares starting from -1 in arbitrary manner
- checking module: operates like a DTM

Note: For a given string x and a given NDTM program, there is an *infinite* number of possible computations possible (one for each 'guessed' string)

Operation of a NDTM program

- write input string in tape squares 1 to $|x|$
- guessing module: writes finite string of symbols from Γ in tape squares starting from -1 in arbitrary manner
- checking module: operates like a DTM

Note: For a given string x and a given NDTM program, there is an *infinite* number of possible computations possible (one for each 'guessed' string)

Terminology & definitions

Accepting computation: all computations that terminate in accepting state (q_Y).
Non-accepting computations: all computations that terminate in non-accepting-state (q_N) or do not terminate at all.

NDTM program M **accepts** x if *there is* an accepting computation for x on M.

The **time complexity** of an NDTM program for a string x is defined as the *minimum* running time over all accepting computations of x by M.
The worst-case time-complexity of an NDTM program is the maximum time complexity over all strings x of a certain length n that are accepted by n.

Non-deterministic algorithm

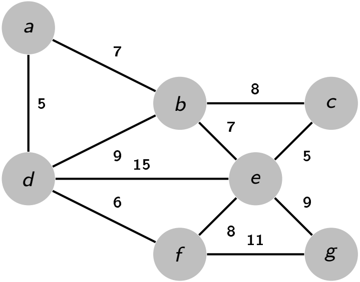
non-deterministic algorithm $\hat{=}$ program for a non-deterministic Turing machine

1. Oracle/guessing stage
2. Checking stage

time complexity of a non-deterministic algorithm
$\hat{=}$ time complexity of the corresponding program

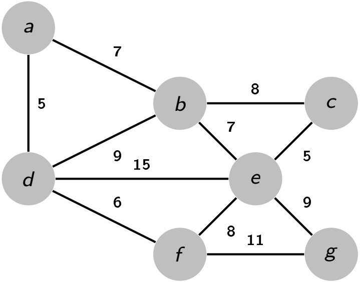
Travelling Salesman Problem (TSP) - Decision version

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.

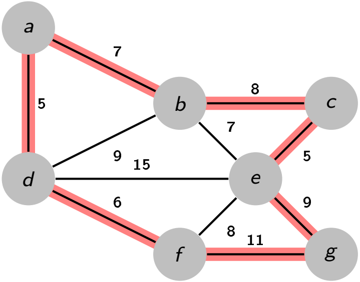
Verification:

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

- What is the time complexity of this algorithm?

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 $\leq B$?

- What is the time complexity of this algorithm?

Warnings:

1. The Church-Turing thesis relates to *deterministic* Turing machines.
2. A non-deterministic Turing machine is a *theoretical* construct, not an actual machine!

Worst-case complexity of problems

Polynomial growth rate:

- $O(\text{poly}(n))$ for some polynomial poly

Worst-case complexity of problems

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

Worst-case complexity of problems

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

Worst-case complexity of problems

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

Worst-case complexity of problems

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.
- (or, alternatively:) if it is solved by an algorithm with polynomial time complexity

Example: The Minimum Spanning Tree (MST) problem

Example of a minimization problem

- Given (adjacency list of) $G = (V, E)$ and $w_e \in \mathbb{R}$ for every $e \in E$,
- Find a spanning tree $T \subseteq E$ minimizing $\sum_{e \in T} w_e$

Example: The Minimum Spanning Tree (MST) problem

Example of a minimization problem

- Given (adjacency list of) $G = (V, E)$ and $w_e \in \mathbb{R}$ for every $e \in E$,
- Find a spanning tree $T \subseteq E$ minimizing $\sum_{e \in T} w_e$

tree: edge-set without cycles (e.g. at most 1 path between 2 vertices)

Example: The Minimum Spanning Tree (MST) problem

Example of a minimization problem

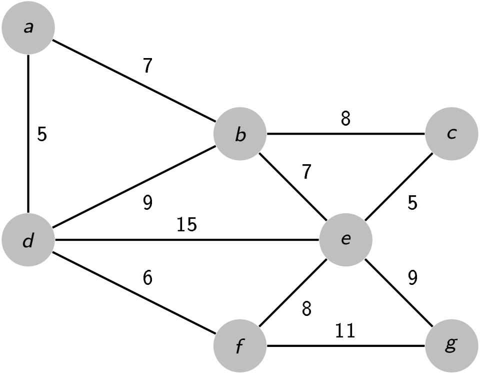
- Given (adjacency list of) $G = (V, E)$ and $w_e \in \mathbb{R}$ for every $e \in E$,
- Find a spanning tree $T \subseteq E$ minimizing $\sum_{e \in T} w_e$

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:

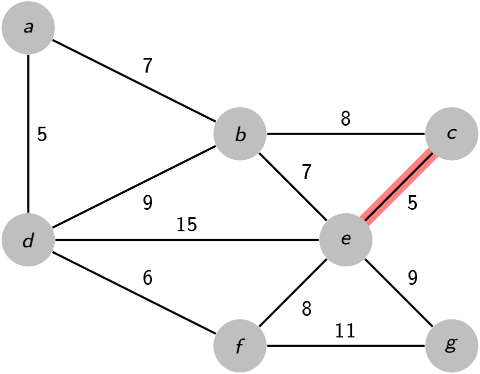
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

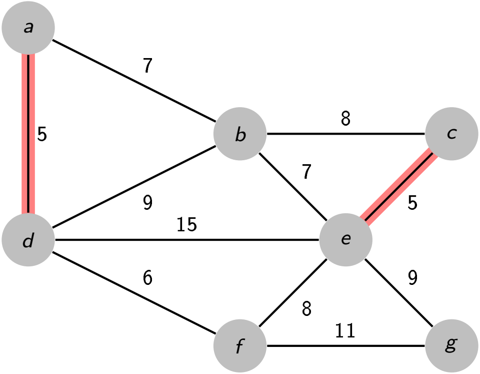
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

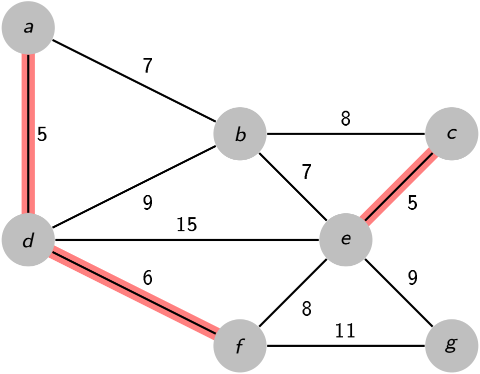
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

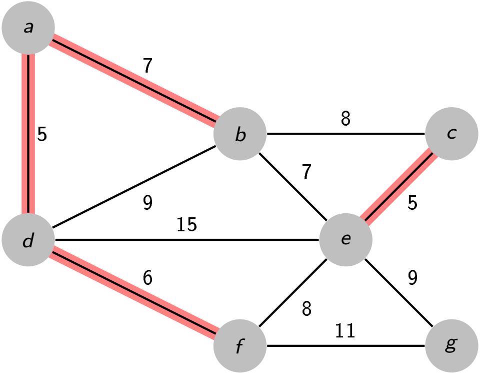
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

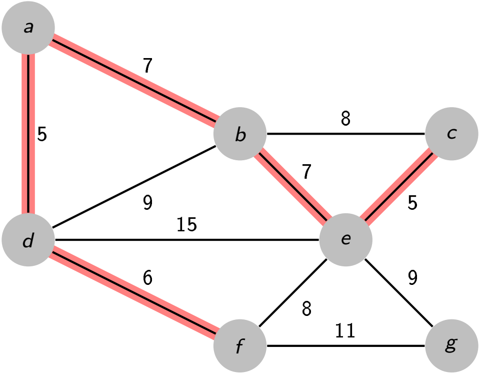
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

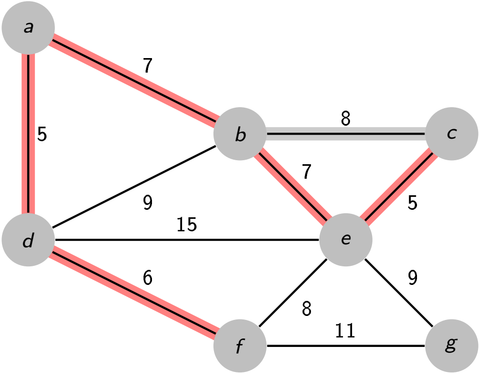
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

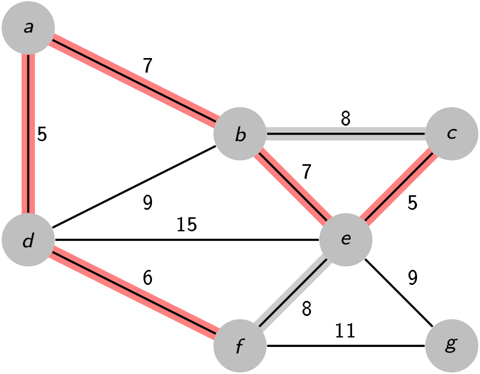
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

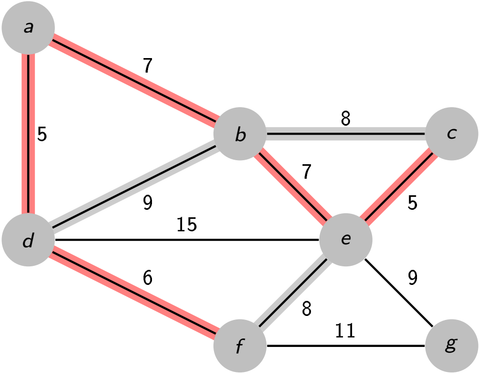
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

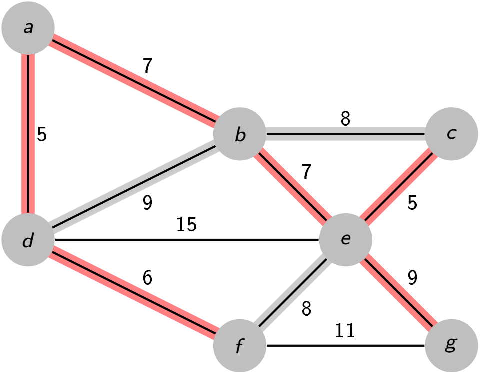
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

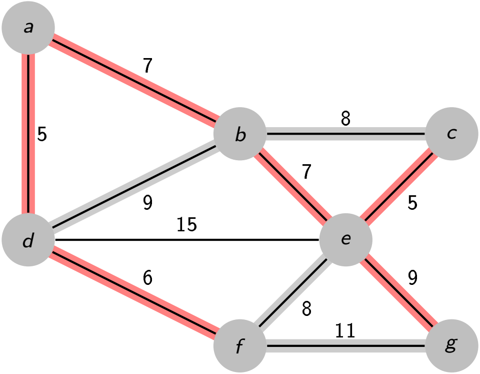
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

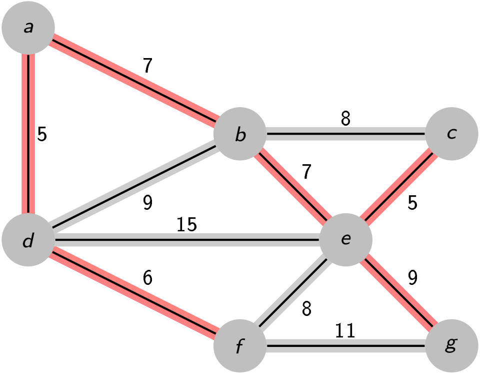
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



Kruskal's algorithm for Minimum Spanning Tree

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

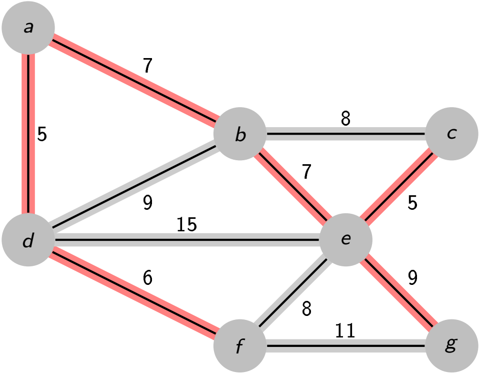
- Consider edges in ascending order of cost
- add the next edge to T unless doing so would create a cycle in T.



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.

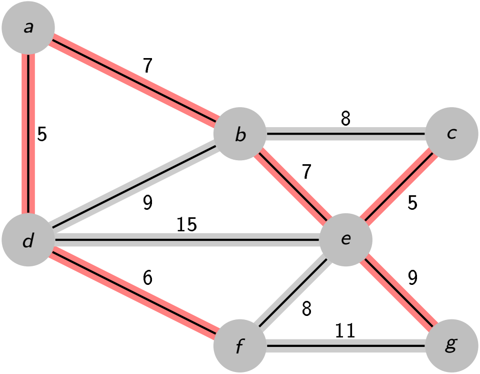


Exercise: this always gives a MST (or see Chapter 23 CLRS)

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.



Exercise: this always gives a MST (or see Chapter 23 CLRS)
Run-time $O(|E|^2)$ (if implemented naïvely); decision version 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.
- (or, alternatively:) if it is solved by an algorithm with polynomial time complexity.

Example: Minimum spanning tree (decision version) 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.
- (or, alternatively:) if it is solved by an algorithm with polynomial time complexity.

Example: Minimum spanning tree (decision version) is in P.

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.

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.
- (or, alternatively:) if it is solved by an algorithm with polynomial time complexity.

Example: Minimum spanning tree (decision version) is in P.

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.
- (or, alternatively:) if it is solved by a *non-deterministic* algorithm with polynomial time complexity.

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.
- (or, alternatively:) if it is solved by an algorithm with polynomial time complexity.

Example: Minimum spanning tree (decision version) is in P.

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.
- (or, alternatively:) if it is solved by a *non-deterministic* algorithm with polynomial time complexity.

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

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.
- (or, alternatively:) if it is solved by an algorithm with polynomial time complexity.

Example: Minimum spanning tree (decision version) is in P.

Definition: Complexity class NP

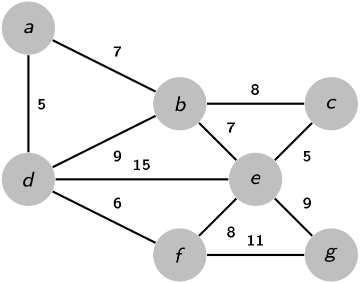
A decision problem X lies in the complexity class NP, if

- if it can be solved in polynomial time on a non-deterministic Turing machine.
- (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.

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:

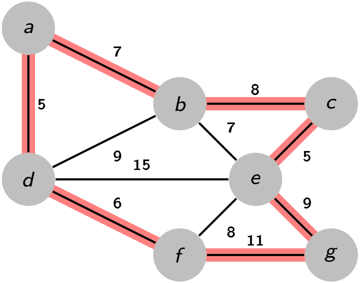
- Specify sequence of edges.

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 $\leq B$?

NP-certificate Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \rightarrow \{true, false\}$
- *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 $(l_1 \vee l_2 \vee \ldots l_j)$.

NP-certificate 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 $(l_1 \vee l_2 \vee \ldots l_j)$.

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 simulsatisfies all clauses in C?

NP-certificate Satisfiability

Let X be a set of logical *variables*.

- *Truth assignment*: $t : X \rightarrow \{true, false\}$
- *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 $(l_1 \vee l_2 \vee \dots l_j)$.

Satisfiability (SAT)

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

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

3-SAT: all clauses consist of 3 literals.

NP-certificate 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 $(l_1 \vee l_2 \vee \ldots l_j)$.

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 simulsatisfies all clauses in C?

3-SAT: all clauses consist of 3 literals.

Examples

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

NP-certificate 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 $(l_1 \vee l_2 \vee \ldots l_j)$.

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 simulsatisfies all clauses in C?

3-SAT: all clauses consist of 3 literals.

Examples

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

Question

What's a good NP-certificate for SAT?

NP-certificate Integer programming

Integer linear programming (ILP)

Instance: an integer matrix A; an integer vector b

Question: does there exist an integer vector x with $Ax \leq b$?

NP-certificate Integer programming

Integer linear programming (ILP)

Instance: an integer matrix A; an integer vector b

Question: does there exist an integer vector x with $Ax \leq b$?

Question

What's a good NP-certificate for ILP?

NP-certificate Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

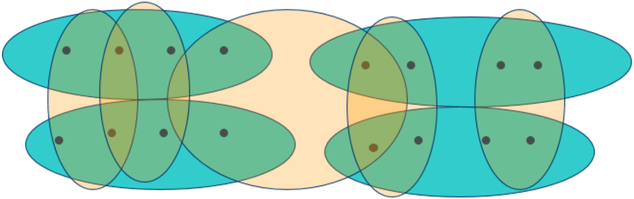
Question: do there exist some subsets S_i that form a partition of X?

NP-certificate Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \dots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

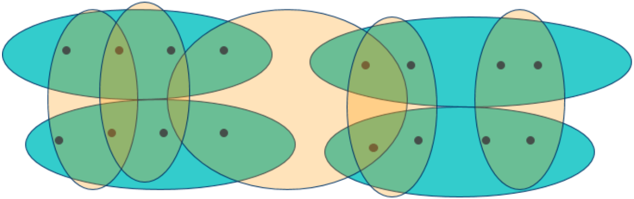


NP-certificate Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?



Question

What's a good NP-certificate for Ex-Cov?

NP-certificate Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \dots, a_n$; a bound b

Question: does there exist an index set $I \subseteq \{1, \dots, n\}$ with $\sum_{i \in I} a_i = b$?

NP-certificate Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = b$?

Example: $(a_1, \ldots, a_{12}) = (1, \ldots, 12), b = 50$. Yes or no instance?

NP-certificate Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = b$?

Example: $(a_1, \ldots, a_{12}) = (1, \ldots, 12)$, $b = 50$. Yes or no instance?
Yes: $1 + 2 + 3 + 4 + 6 + 7 + 8 + 9 + 10 = 50$.

NP-certificate Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = b$?

Example: $(a_1, \ldots, a_{12}) = (1, \ldots, 12)$, $b = 50$. Yes or no instance?
Yes: $1 + 2 + 3 + 4 + 6 + 7 + 8 + 9 + 10 = 50$.

Question

What's a good NP-certificate for SS?

P versus NP

- P = class of all problems that are easy to solve
  P stands for Polynomial Time
- NP = huge class of problems that fulfill some soft condition
  NP contains lots of interesting and important decision problems
  NP stands for Non-deterministic Polynomial Time

P versus NP

- P = class of all problems that are easy to solve
  P stands for Polynomial Time
- NP = huge class of problems that fulfill some soft condition
  NP contains lots of interesting and important decision problems
  NP stands for Non-deterministic Polynomial Time

Big open question

P=NP ????

P versus NP

- P = class of all problems that are easy to solve
  P stands for Polynomial Time
- NP = huge class of problems that fulfill some soft condition
  NP contains lots of interesting and important decision problems
  NP stands for Non-deterministic Polynomial Time

Big open question

P=NP ????

Answer YES:

- would trigger a revolution in computing
- if a short solution exists, it can be found quickly

P versus NP

- P = class of all problems that are easy to solve
  P stands for Polynomial Time
- NP = huge class of problems that fulfill some soft condition
  NP contains lots of interesting and important decision problems
  NP stands for Non-deterministic Polynomial Time

Big open question

P=NP ????

Answer YES:

- would trigger a revolution in computing
- if a short solution exists, it can be found quickly

Answer NO:

- that's what most people expect
- even very short solutions may be very hard to find

To prove that a problem

- can be solved in $O(n \log n)$, $O(n^3)$, etc
- is in P
- is in NP

is straightforward (although not always easy):

To prove that a problem

- can be solved in $O(n \log n)$, $O(n^3)$, etc
- is in P
- is in NP

is straightforward (although not always easy):

$\rightarrow$ Find an algorithm that runs in time $O(n \log n)$ / $O(n^3)$ / ... / polynomial time / non-deterministic polynomial time.

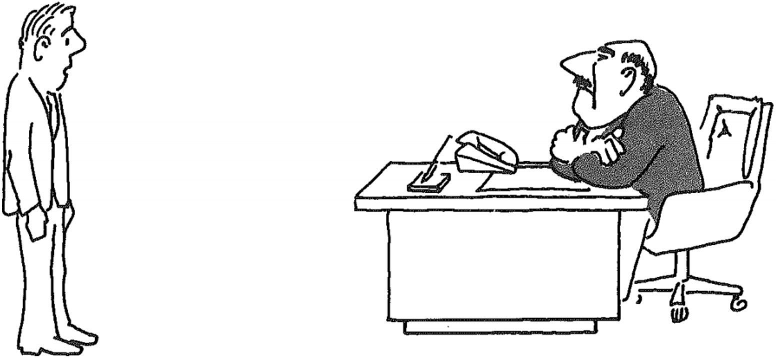
To prove that a problem

- can be solved in $O(n \log n)$, $O(n^3)$, etc
- is in P
- is in NP

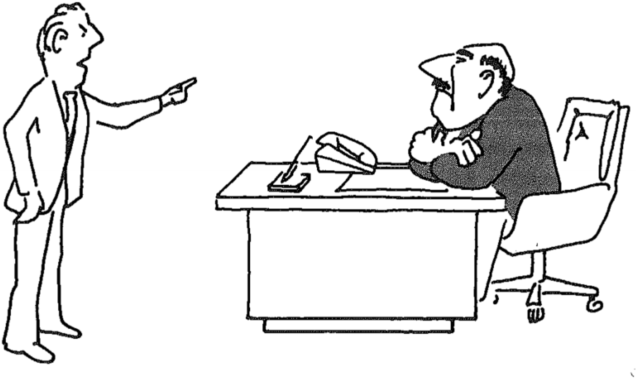
is straightforward (although not always easy):

$\rightarrow$ Find an algorithm that runs in time $O(n \log n)$ / $O(n^3)$ / ... / polynomial time / non-deterministic polynomial time.

How do we prove that a problem can**not** be solved in a certain time?



“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!"

Proving lower bounds (short note)

Example: Find maximum element from unsorted list

Input: A list of numbers $m_1, m_2, \ldots m_n$, a number M.
Question: Is there an element $\geq M$ in the list.

Proving lower bounds (short note)

Example: Find maximum element from unsorted list

Input: A list of numbers $m_1, m_2, \ldots m_n$, a number M.
Question: Is there an element $\geq M$ in the list.

How fast can you solve this problem?

Proving lower bounds (short note)

Example: Find maximum element from unsorted list

Input: A list of numbers $m_1, m_2, \ldots m_n$, a number M.
Question: Is there an element $\geq M$ in the list.

How fast can you solve this problem?

- can be done in time $O(n)$
- requires time $\Omega(n)$
- thus: $\Theta(n)$

Proving lower bounds (short note)

Example: Find maximum element from unsorted list

Input: A list of numbers $m_1, m_2, \ldots m_n$, a number M.
Question: Is there an element $\geq M$ in the list.

How fast can you solve this problem?

- can be done in time $O(n)$
- requires time $\Omega(n)$
- thus: $\Theta(n)$

Note: Most problems need time $\Omega(n)$ to be solved.
Can you think of one that does not?

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)
- Can we sort in $O(n)$?

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)
- Can we sort in $O(n)$?

Information theoretic lower bound on sorting (sketch)

There are $n!$ different permutations of n numbers.

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)
- Can we sort in $O(n)$?

Information theoretic lower bound on sorting (sketch)

There are $n!$ different permutations of n numbers.
An algorithm that sorts all of them correctly, needs to follow a different sequence of steps for each of them.

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)
- Can we sort in $O(n)$?

Information theoretic lower bound on sorting (sketch)

There are $n!$ different permutations of n numbers.
An algorithm that sorts all of them correctly, needs to follow a different sequence of steps for each of them.
Thus it needs at least $\log_2(n!)$ steps.

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)
- Can we sort in $O(n)$?

Information theoretic lower bound on sorting (sketch)

There are $n!$ different permutations of n numbers.
An algorithm that sorts all of them correctly, needs to follow a different sequence of steps for each of them.
Thus it needs at least $\log_2(n!)$ steps.

$$\log_2(n!) = \log_2(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1) = \log_2(n) + \log_2(n-1) + \ldots + \log(2) + \log(1)$$

$$= \sum_{i=1}^{n} \log_2(i) = \sum_{i=1}^{\frac{n}{2}-1} \log_2(i) \sum_{i=\frac{n}{2}}^{n} \log_2(i) \geq 0 + \sum_{i=\frac{n}{2}}^{n} \log_2(\frac{n}{2}) = \frac{n}{2} \log_2(\frac{n}{2}) = \Omega(n \log(n))$$

Sorting

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

- Insertion sort needs $O(n^2)$ in the worst case.
- Other sorting algorithms (like Merge Sort) need $O(n \log n)$. (see CLRS)
- Can we sort in $O(n)$?

Information theoretic lower bound on sorting (sketch)

There are $n!$ different permutations of n numbers.
An algorithm that sorts all of them correctly, needs to follow a different sequence of steps for each of them.
Thus it needs at least $\log_2(n!)$ steps.

$$\log_2(n!) = \log_2(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1) = \log_2(n) + \log_2(n-1) + \ldots + \log(2) + \log(1)$$

$$= \sum_{i=1}^{n} \log_2(i) = \sum_{i=1}^{\frac{n}{2}-1} \log_2(i) \sum_{i=\frac{n}{2}}^{n} \log_2(i) \geq 0 + \sum_{i=\frac{n}{2}}^{n} \log_2(\frac{n}{2}) = \frac{n}{2} \log_2(\frac{n}{2}) = \Omega(n \log(n))$$

For a more extensive proof, see here
`https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf`

3-Satisfiability (3-SAT)

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

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

3-Satisfiability (3-SAT)

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

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

From stackexchange
(https://cstheory.stackexchange.com/questions/1060/best-upper-bounds-on-sat?rq=1 and
https://cstheory.stackexchange.com/questions/93/what-are-the-best-current-lower-bounds-on-3sat)
(retrieved 13.9.19)

- Best found non-randomized algorithm (for 3-SAT) seems to be 1.32793^n
- Best found randomized algorithm similar ($O(1.321^n)$?)
- No one so far has been able to prove $\Omega(n^2)$

Lower bounds on problem complexity tend to be rare / weak / difficult to prove.

$\rightarrow$ We look at a different approach.

"I can't find an efficient algorithm, but neither can all these famous people."

Reductions

Definition

For two decision problems X and Y, we say that X (polynomially) **reduces** to Y (and we write $X \leq_p Y$)
if there exists a polynomial time transformation f that translates instance of X into instances of Y with $I \in$YES(X) $\iff$ $f(I) \in$YES(Y).

Often, we omit the word 'polynomially' and just say that X reduces to Y.

Reductions

Definition

For two decision problems X and Y, we say that X (polynomially) **reduces** to Y (and we write $X \leq_p Y$)
if there exists a polynomial time transformation f that translates instance of X into instances of Y with $I \in$YES(X) $\iff$ $f(I) \in$YES(Y).

Often, we omit the word 'polynomially' and just say that X reduces to Y.

Intuition:

- X can be modelled as a special case of Y
- the 'computational hardness' of X is upper bounded by Y's
- If Y is easy, then also X is easy
- If X is difficult, then also Y is difficult

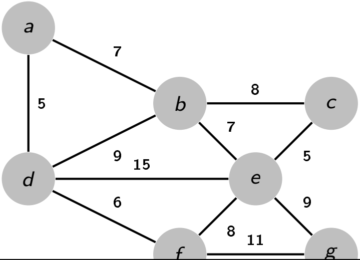
Hamiltonian cycle / TSP

Hamiltonian cycle (HC)

Instance: an undirected graph $G = (V, E)$
Question: does G contain a Hamiltonian cycle?
(a simple cycle that visits every vertex exactly once)

Travelling Salesman Problem (TSP)

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



Theorem

HC $\leq_p$ TSP.

Proof: .

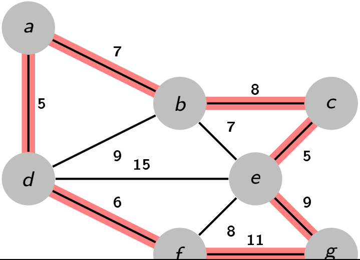
Hamiltonian cycle / TSP

Hamiltonian cycle (HC)

Instance: an undirected graph $G = (V, E)$
Question: does G contain a Hamiltonian cycle?
(a simple cycle that visits every vertex exactly once)

Travelling Salesman Problem (TSP)

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



Theorem

HC $\leq_p$ TSP.

Proof: .

Clique

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

Theorem

SAT $\leq_p$ CLIQUE.

Proof:

Clique

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

Theorem

SAT $\leq_p$ CLIQUE.

Proof: Given a set of clauses $\{c_1, c_2, \ldots, c_m\}$, over $x_1, \ldots, x_n$

Clique

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

Theorem

SAT $\leq_p$ CLIQUE.

Proof: Given a set of clauses $\{c_1, c_2, \ldots, c_m\}$, over $x_1, \ldots, x_n$
define instance instance of clique (our function f):

Clique

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

Theorem

SAT $\leq_p$ CLIQUE.

Proof: Given a set of clauses $\{c_1, c_2, \ldots, c_m\}$, over $x_1, \ldots, x_n$ define instance instance of clique (our function f):

$$V = \{(l, i) | l \text{ is a literal in } c_i\}$$
$$E = \{\{(l, i), (l', i')\} | l \neq \neg l' \wedge i \neq i'\}$$
$$k = m$$

Lemma

Reducibility is a transitive relation:
$X \leq_p Y$ and $Y \leq_p Z$ implies $X \leq_p Z$

Lemma

Reducibility is a transitive relation:
$X \leq_p Y$ and $Y \leq_p Z$ implies $X \leq_p Z$

Proof: by putting the two tranformations into series

NP-hardness

Definition

A decision problem X is *NP-hard*,
if **all** problems $Y \in NP$ can be reduced to it
(that is, if $Y \leq_p X$ holds for all $Y \in NP$)

NP-hardness

Definition

A decision problem X is *NP-hard*,
if **all** problems $Y \in NP$ can be reduced to it
(that is, if $Y \leq_p X$ holds for all $Y \in NP$)

Definition

A decision problem X is *NP-complete*,
if $X \in NP$ and X is NP-hard.

NP-hardness

Definition

A decision problem X is *NP-hard*,
if **all** problems $Y \in NP$ can be reduced to it
(that is, if $Y \leq_p X$ holds for all $Y \in NP$)

Definition

A decision problem X is *NP-complete*,
if $X \in NP$ and X is NP-hard.

Intuition:

- NP-complete problems are the hardest problems in NP
- Recall: NP is huge and contains tons of important problems
- Some people consider NP-complete problems to be intractable.

NP-hardness

Theorem

If one NP-complete problem X has a polynomial time algorithm then all NP-complete problems have polynomial time algorithms (and hence P=NP)

NP-hardness

Theorem

If one NP-complete problem X has a polynomial time algorithm then all NP-complete problems have polynomial time algorithms (and hence P=NP)

Why?

NP-hardness

Theorem

If one NP-complete problem X has a polynomial time algorithm then all NP-complete problems have polynomial time algorithms (and hence P=NP)

Why? Can reduce to X and then solve produced instance of X.

"I can't find an efficient algorithm, but neither can all these famous people."

Cook-Levin theorem (1971)

SAT is NP-complete.

Cook-Levin theorem (1971)

SAT is NP-complete.

- Stephen Cook (born 1939):
  American-Canadian computer scientist and mathematician
- Leonid Levin (born 1948):
  Russian computer scientist, discovered the result somewhat earlier

Proof of Cook-Levin

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

clause group	restriction imposed
G_1	at each time i, M is in exactly one state
G_2	at each time i, the read-write head is scanning exactly one tape square
G_3	at each time i, each tape square contains exactly one symbol from Γ
G_4	at time 0, the computation is in the initial configuration of its checking stage for input x
G_5	By time $p(n)$, M has entered state q_y and hence has accepted x
G_6	For each time i the configuration of M at time $i+1$ follows by a single application of the transition function δ from the configuration at time i

Proof of Cook-Levin

G_1: at each time i, M is in exactly one state

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert\Gamma\rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_1: at each time i, M is in exactly one state

$Q[i,0] \vee Q[i,1] \vee \ldots \vee Q[i,r]$ for all $0 \leq i \leq p(n)$
$\neg Q[i,j] \vee \neg Q[i,j']$ for all $0 \leq i \leq p(n), 0 \leq j \leq j' \leq r$

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_2: at each time i, the read-write head is scanning exactly one tape square

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_2: at each time i, the read-write head is scanning exactly one tape square

$H[i,-p(n)] \vee H[i,-p(n)+1] \vee \ldots \vee H[i,p(n)+1]$ for all $0 \leq i \leq p(n)$
$\neg H[i,j] \vee \neg H[i,j']$ for all $0 \leq i \leq p(n), -p(n) \leq j \leq j' \leq$

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q\rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert\Gamma\rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_3: at each time i, each tape square contains at least one symbol from Γ

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_3: at each time i, each tape square contains at least one symbol from Γ

$S[i,j,0] \vee S[i,j,1] \vee \ldots \vee S[i,j,|\Gamma|]$ for all $0 \le i \le p(n)$, $-p(n) \le j \le p(n)+1$

$\neg S[i,j,k] \vee \neg S[i,j,k']$ for all $0 \le i \le p(n), -p(n) \le j \le p(n)+1$, $0 \le k \le$

Variable	Range	Intended meaning
$Q[i,k]$	$0 \le i \le p(n)$, $0 \le k \le \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \le i \le p(n)$, $-p(n) \le j \le p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \le i \le p(n)$, $-p(n) \le j \le p(n)+1$, $0 \le k \le \lvert\Gamma\rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_4: at time 0, the computation is in the initial configuration of its checking stage for input x

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \|Q\|$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \|\Gamma\|$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_4: at time 0, the computation is in the initial configuration of its checking stage for input x

$Q[0,0], H[0,1], S[0,0,0]$
$S[0,1,k_1], S[0,2,k_2], \ldots, S[0,n,k_n],$ with $x = (s_{k_1}, s_{k_2}, \ldots, s_{k_n})$
$S[0,n+1,0], S[0,n+2,0], \ldots, S[0,p(n)+1,0]$

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q\rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert\Gamma\rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_5: by time $p(n)$, M has entered state q_y

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_5: by time $p(n)$, M has entered state q_y

$Q[p(n), 1]$

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_6: Changes according to transition function

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

Proof of Cook-Levin

G_6: Changes according to transition function

$$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee H[i+1,j+\Delta]$$
$$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee Q[i+1,k']$$
$$\neg H[i,j] \vee \neg Q[i,k] \vee \neg S[i,j,l] \vee S[i+1,j,l']$$

with for $q \in Q \setminus \{q_Y, q_N\}$: $\delta(q_k, s_l) = (q_{k'}, s_{l'}, \delta)$ and
for $q \in \{q_Y, q_N\}$: $\delta = 0$, $k' = k$, $l' = l$

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leq i \leq p(n)$, $0 \leq k \leq \lvert Q \rvert$	at time i, M is in state k
$H[i,j]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$	at time i, the read-write head of M scans tape square j
$S[i,j,k]$	$0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)+1$, $0 \leq k \leq \lvert \Gamma \rvert$	at time i, the entry on tape square j is s_k

NP-hardness: 3-SAT

3-SAT

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

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

Theorem

3-SAT is NP-hard (and NP-complete).

NP-hardness: 3-SAT

3-SAT

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

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

Theorem

3-SAT is NP-hard (and NP-complete).

Proof: By reduction from SAT. Let $I = (X, C)$ an instance of SAT. We construct the following instance (X', C') of 3-SAT:

- $X_0 := X$
- For each clause c_j we construct a set of variables X_j and additional clauses C_j (with 3 literals each)
- $X' := \bigcup_{j=0}^{|C|} X_j$, $C' := \bigcup_{j=1}^{|C|} C_j$

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof:

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT.

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.
We define matrix A as

$$a_{ij} = \begin{cases} \\ \\ \end{cases}$$

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.
We define matrix A as

$$a_{ij} = \begin{cases} -1 & \text{if } x_j \text{ is in } c_i \\ & \\ & \end{cases}$$

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.
We define matrix A as

$$a_{ij} = \begin{cases} -1 & \text{if } x_j \text{ is in } c_i \\ 1 & \text{if } \neg x_j \text{ is in } c_i \\ & \end{cases}$$

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.
We define matrix A as

$$a_{ij} = \begin{cases} -1 & \text{if } x_j \text{ is in } c_i \\ 1 & \text{if } \neg x_j \text{ is in } c_i \\ 0 & x_j \text{ is not in } c_i, \end{cases}$$

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.
We define matrix A as

$$a_{ij} = \begin{cases} -1 & \text{if } x_j \text{ is in } c_i \\ 1 & \text{if } \neg x_j \text{ is in } c_i \\ 0 & x_j \text{ is not in } c_i, \end{cases}$$

and $b_i = \#$negated literals in c_i -1.

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = true$.
We define matrix A as

$$a_{ij} = \begin{cases} -1 & \text{if } x_j \text{ is in } c_i \\ 1 & \text{if } \neg x_j \text{ is in } c_i \\ 0 & x_j \text{ is not in } c_i, \end{cases}$$

and $b_i = \#$negated literals in c_i -1.
encode $y_j \in \{0, 1\}$ as $0 \leq y_j \leq 1$.

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

ILP is NP-hard (and NP-complete).

Proof: by reduction from SAT. Let (X, C) with $X = x_1, \ldots, x_n$ and $C = \{c_1, c_2, \ldots, c_m\}$ be an instance of SAT.
Define A and b, use decision vars $y_j \in \{0, 1\}$ to indicate if $t(x_j) = \mathit{true}$.
We define matrix A as

$$a_{ij} = \begin{cases} -1 & \text{if } x_j \text{ is in } c_i \\ 1 & \text{if } \neg x_j \text{ is in } c_i \\ 0 & x_j \text{ is not in } c_i, \end{cases}$$

and $b_i = \#$negated literals in c_i -1.
encode $y_j \in \{0, 1\}$ as $0 \leq y_j \leq 1$.
To show: There is a satisfying truth assignment for $(X, C) \Leftrightarrow$ there is a vector y fulfilling $Ay \leq b$

NP-hardness: Integer programming

Integer programming (ILP)

Instance: an integer matrix A; an integer vector b
Question: does there exist an integer vector y with $Ay \leq b$?

Theorem

SAT $\leq_p$ ILP, and therefore ILP is NP-hard (and NP-complete).

Consequence: Every problem in NP can be modelled as an ILP.

NP-hardness: Clique

Clique

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

Theorem

CLIQUE is NP-hard (and NP-complete).

Proof: SAT is NP-hard and SAT $\leq_p$ CLIQUE.

NP-hardness: Independent set

Independent set (IS)

Instance: a graph $G = (V, E)$; an integer k
Question: does G contain an independent set of size (at least) k?
(a set of vertices that does not span any edge)

Theorem

IS is NP-hard (and NP-complete).

Proof:

NP-hardness: Independent set

Independent set (IS)

Instance: a graph $G = (V, E)$; an integer k
Question: does G contain an independent set of size (at least) k?
(a set of vertices that does not span any edge)

Theorem

IS is NP-hard (and NP-complete).

Proof: By reduction from CLIQUE:

NP-hardness: Independent set

Independent set (IS)

Instance: a graph $G = (V, E)$; an integer k
Question: does G contain an independent set of size (at least) k?
(a set of vertices that does not span any edge)

Theorem

IS is NP-hard (and NP-complete).

Proof: By reduction from CLIQUE:
Given an instance $(G = (V, E), k)$ of clique, construct the following instance of IS:
$V' := V$, $E' := \{\{i,j\} : i \neq j \in V, \{i,j\} \notin E\}$, $k' := k$.

NP-hardness: Independent set

Independent set (IS)

Instance: a graph $G = (V, E)$; an integer k
Question: does G contain an independent set of size (at least) k?
(a set of vertices that does not span any edge)

Theorem

IS is NP-hard (and NP-complete).

Proof: By reduction from CLIQUE:
Given an instance $(G = (V, E), k)$ of clique, construct the following instance of IS:
$V' := V$, $E' := \{\{i,j\} : i \neq j \in V, \{i,j\} \notin E\}$, $k' := k$.
Show:
$X \subset V$ is a clique in $G \Leftrightarrow X$ is an independent set in G'

NP-hardness: Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

Theorem

(Ex-Cov) is NP-hard (and NP-complete).

Proof:

NP-hardness: Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

Theorem

(Ex-Cov) is NP-hard (and NP-complete).

Proof: by reduction from IS.

NP-hardness: Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

Theorem

(Ex-Cov) is NP-hard (and NP-complete).

Proof: by reduction from IS.
Let (G, k) with $G = (V, E)$ be an instance of IS.

NP-hardness: Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

Theorem

(Ex-Cov) is NP-hard (and NP-complete).

Proof: by reduction from IS.
Let (G, k) with $G = (V, E)$ be an instance of IS.
Define an instance of (Ex-Cov) as follows: $X := E \cup \{1, \ldots, k\}$
and subsets
$S_{ih} := \{\{i,j\} : \{i,j\} \in E\} \cup \{h\}$ for $i \in V,\ h = 1, \ldots k$
$S_{\{i,j\}} := \{\{i,j\}\}$ for $\{i,j\} \in E$

NP-hardness: Exact cover

Exact cover (Ex-Cov)

Instance: a ground set X; subsets $S_1, \ldots, S_m$ of X

Question: do there exist some subsets S_i that form a partition of X?

Theorem

(Ex-Cov) is NP-hard (and NP-complete).

Proof: by reduction from IS.
Let (G, k) with $G = (V, E)$ be an instance of IS.
Define an instance of (Ex-Cov) as follows: $X := E \cup \{1, \ldots, k\}$
and subsets
$S_{ih} := \{\{i,j\} : \{i,j\} \in E\} \cup \{h\}$ for $i \in V,\ h = 1, \ldots k$
$S_{\{i,j\}} := \{\{i,j\}\}$ for $\{i,j\} \in E$
Show:
If S is a solution to (Ex-Cov), $i : S_{ih} \in S$ is an independent set of size k.
If $X = \{x_1, x_2, \ldots, x_k\} \subset V$ is an independent set, $\bigcup_{j=1}^{k} S_{x_j j} \cup \{\{i,j\} : i,j \notin X\}$ is a solution to (Ex-Cov).

NP-hardness: Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Theorem

SS is NP-hard (and NP-complete).

Proof:

NP-hardness: Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Theorem

SS is NP-hard (and NP-complete).

Proof: by reduction from Ex-Cov.

NP-hardness: Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Theorem

SS is NP-hard (and NP-complete).

Proof: by reduction from Ex-Cov.
Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

NP-hardness: Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \dots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \dots, n\}$ with $\sum_{j \in J} a_j = b$?

Theorem

SS is NP-hard (and NP-complete).

Proof: by reduction from Ex-Cov.
Let $(X = \{x_1, \dots, x_m\}, \{S_1, \dots, S_n\})$ be an instance of Ex-Cov.
Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

Show:
J is the index set of a solution to Ex-Cov $\Leftrightarrow$ J is the index set of a solution to SS.

NP-hardness: Subset Sum

Subset Sum (SS)

Instance: positive integers $a_1, \ldots, a_n$; a bound b

Question: does there exist an index set $J \subseteq \{1, \ldots, n\}$ with $\sum_{j \in J} a_j = b$?

Theorem

SS is NP-hard (and NP-complete).

Proof: by reduction from Ex-Cov.
Let $(X = \{x_1, \ldots, x_m\}, \{S_1, \ldots, S_n\})$ be an instance of Ex-Cov.
Define numbers a_j as $a_j := \sum_{i=1}^{m} c_{ij} \cdot d_i$ with $c_{ij} = 1$ if $x_i \in S_j$ and $d_i = (n+1)^{i-1}$.
Set $b := \sum_{i=1}^{m} (n+1)^{i-1}$.

Show:
J is the index set of a solution to Ex-Cov $\Leftrightarrow$ J is the index set of a solution to SS.
Also: argue why this is a *polynomial-time* transformation.

NP-hardness: 2-Partition

2-PARTITION

Instance: positive integers $a_1, \dots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \dots, n\}$ with $\sum_{i \in I} a_i = A$?

Theorem

2-PARTITION is NP-hard (and thus NP-complete).

Proof:

NP-hardness: 2-Partition

2-PARTITION

Instance: positive integers $a_1, \ldots, a_n$ with $\sum_{i=1}^{n} a_i = 2A$.

Question: does there exist an index set $I \subseteq \{1, \ldots, n\}$ with $\sum_{i \in I} a_i = A$?

Theorem

2-PARTITION is NP-hard (and thus NP-complete).

Proof: by reduction from SS.

Vertex cover (VC)

Instance: a graph $G = (V, E)$; an integer k
Question: does G contain a vertex cover of size (at most) k?
(a set of vertices that touches every edge)

Theorem

VC is NP-hard (and thus NP-complete).

Proof:

Vertex cover (VC)

Instance: a graph $G = (V, E)$; an integer k
Question: does G contain a vertex cover of size (at most) k?
(a set of vertices that touches every edge)

Theorem

VC is NP-hard (and thus NP-complete).

Proof: by reduction from IS.

NP-hardness: Hamiltonian cycle / TSP

Directed Hamiltonian cycle (dir-HC)

Instance: a directed graph (V, E)
Question: does this graph contain a directed Hamiltonian cycle?

Theorem

Dir-HC is NP-complete.

NP-hardness: Hamiltonian cycle / TSP

Directed Hamiltonian cycle (dir-HC)

Instance: a directed graph (V, E)
Question: does this graph contain a directed Hamiltonian cycle?

Theorem

Dir-HC is NP-complete.

Proof: Easy to see: in NP.
To show NP-hard: reduction from VC.
Given instance $G = (V, E), k$ of VC. Define $G' = (V', E')$:

NP-hardness: Hamiltonian cycle / TSP

Directed Hamiltonian cycle (dir-HC)

Instance: a directed graph (V, E)
Question: does this graph contain a directed Hamiltonian cycle?

Theorem

Dir-HC is NP-complete.

Proof: Easy to see: in NP.
To show NP-hard: reduction from VC.
Given instance $G = (V, E), k$ of VC. Define $G' = (V', E')$:

$$
\begin{aligned}
V' &= \{(i,j), \{i,j\}, (j,i) | \{i,j\} \in E\} \cup \{1, \dots, k\} \\
E' &= \{((i,j), \{i,j\}), (\{i,j\}, (i,j)), ((j,i), \{i,j\}), (\{i,j\}, (j,i)) | \{i,j\} \in E\} \\
&\quad \cup \{((i,j), q), (q, (i,j)), ((j,i), q), (q, (j,i)) | \{i,j\} \in E, q = 1, \dots, k\} \\
&\quad \cup \{((h,i), (i,j)) | \{h,i\} \in E, \{i,j\} \in E, h \neq j\} \\
&\quad \cup \{(i,j), (j,i) | 1 \leq i < j \leq k\}
\end{aligned}
$$

NP-hardness: Hamiltonian cycle / TSP

Theorem

HC is NP-hard (and thus NP-complete).

Proof: Reduction from dir-HC.

Theorem

TSP is NP-complete.

Proof: already seen: in NP and TSP$\leq_p$ HC.

Recommended reading

Garey and Johnson. 'Algorithms and Complexity'
Lenstra and Rinnooy Kan. Computational complexity of discrete optimization problems.
Annals of Discrete Mathematics 4 (pp 121-140), 1979.
Electronic copy available on website

Cormen, Leiserson, Rivest and Stein 'Introduction to Algorithms':

- Chapter 1-3 (basics)
- Chapter 23 (minimum spanning trees)
- Chapter 34 (P, NP, NP-completeness, Cook-Levin theorem, reductions)