NP-Completeness

Peter Lammich

February 19, 2020

Tractability

- How complex is a problem?
    - How long does fastest algorithm take?

Tractability

- How complex is a problem?
  - How long does fastest algorithm take?
- Example: general sorting. $O(n \log n)$
  - no faster algorithm exists!

Tractability

- How complex is a problem?
  - How long does fastest algorithm take?
- Example: general sorting. $O(n \log n)$
  - no faster algorithm exists!
- But how about factorization?
  - find prime factors of n bit integer
  - naive algorithm: try all possible factors. $O(2^n)$
  - fastest algorithm: unknown!

Execution Time

- Recall: we count steps, wrt. some computational model
    - binary search: $O(\log n)$ steps
    - mergesort: $O(n \log n)$ steps
    - Bellman-Ford: $O(|V||E|) = O(|V|^3)$ steps

Execution Time

- Recall: we count steps, wrt. some computational model
  - binary search: $O(\log n)$ steps
  - mergesort: $O(n \log n)$ steps
  - Bellman-Ford: $O(|V||E|) = O(|V|^3)$ steps
- These are all polynomial, i.e., $O(n^k)$ for *constant* k.

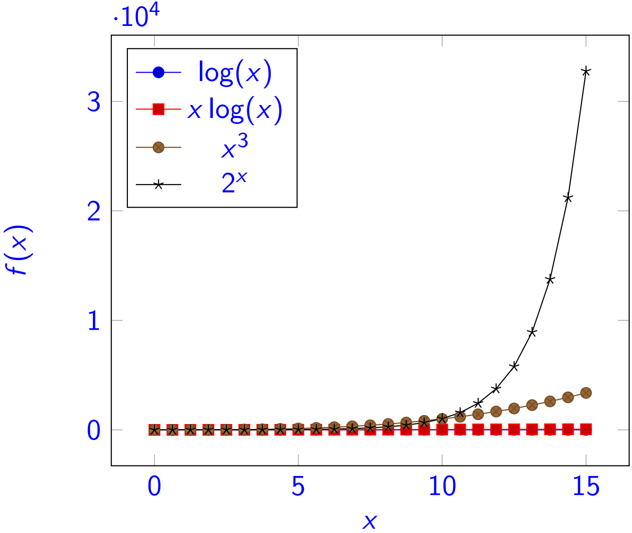
Execution Time

- Recall: we count steps, wrt. some computational model
  - binary search: $O(\log n)$ steps
  - mergesort: $O(n \log n)$ steps
  - Bellman-Ford: $O(|V||E|) = O(|V|^3)$ steps
- These are all polynomial, i.e., $O(n^k)$ for *constant* k.
- How about SAT: find a solution to a CNF formula over n variables?
  - $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge \ldots$

Execution Time

- Recall: we count steps, wrt. some computational model
  - binary search: $O(\log n)$ steps
  - mergesort: $O(n \log n)$ steps
  - Bellman-Ford: $O(|V||E|) = O(|V|^3)$ steps
- These are all polynomial, i.e., $O(n^k)$ for *constant* k.
- How about SAT: find a solution to a CNF formula over n variables?
  - $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3) \wedge \ldots$
  - Naive algorithm: try all 2^n possibilities. $O(2^n)$

Comparing Execution Times



Comparing Execution Times

- If one step is 1ns (1GHz clock), and algorithm needs $f(n)$ steps
- What input size can we handle in ... (approx values)

$f(n)$	1 second	1 hour	1 day	1 year	1 century	lifetime of universe
$\log(n)$	$10^{301029995}$	$\dots$				
n	10^9	10^{12}	10^{14}	10^{16}	10^{18}	10^{26}
n^2	10^4	10^6	10^7	10^8	10^9	10^{13}
n^5	60	300	600	2000	5000	10^5
2^n	29	41	46	54	61	88

A few numbers for comparison (approx/estimates):

- 10^{78} – 10^{82}: atoms in the known universe
- $8.8 * 10^{27}$ meters: diameter of known universe
- $5 * 10^{22}$ stars in known universe
- $4 * 10^{17}$ seconds since big bang
- 10^{13} bits of memory on a hard disk (1TiB)
- 10^{11} people ever lived
- $7.8 * 10^9$ people on earth
- $6.6 * 10^6$ people in UK

Polynomial Time

- Problem in $O(n^k)$ is called *tractable*

Polynomial Time

- Problem in $O(n^k)$ is called *tractable*
- So we don't care about polynomials
    - We treat n^2 and n^{100} algorithm the same!

Polynomial Time

- Problem in $O(n^k)$ is called *tractable*
- So we don't care about polynomials
    - We treat n^2 and n^{100} algorithm the same!
- Notation $poly(n)$: some polynomial in n

Are all Problems Tractable

Are all Problems Tractable

- No!

Are all Problems Tractable

- No!
- There are undecidable problems, e.g.
    - Halting problem: Does a program terminate for a given input?

Are all Problems Tractable

- No!
- There are undecidable problems, e.g.
    - Halting problem: Does a program terminate for a given input?
- There are problems that can only be solved in exponential time
    - Can a position in (generalized) Go be won (with Japanese ko rules)

Are all Problems Tractable

- No!
- There are undecidable problems, e.g.
  - Halting problem: Does a program terminate for a given input?
- There are problems that can only be solved in exponential time
  - Can a position in (generalized) Go be won (with Japanese ko rules)
- There are many problems for which we don't know
  - factorization, SAT, ...

Decision Problems

- Given some input of size n, compute a true/false answer
    - is a list sorted?
    - does a graph have a negative-weight cycle?
    - does a CNF formula have a solution?

Decision Problems

- Given some input of size n, compute a true/false answer
  - is a list sorted?
  - does a graph have a negative-weight cycle?
  - does a CNF formula have a solution?
- We also had optimization problems
  - compute shortest path

Decision Problems

- Given some input of size n, compute a true/false answer
  - is a list sorted?
  - does a graph have a negative-weight cycle?
  - does a CNF formula have a solution?
- We also had optimization problems
  - compute shortest path
- Usually, optimization problems have related decision problem
  - is there a path of length less than k?

Decision Problems

- Given some input of size n, compute a true/false answer
    - is a list sorted?
    - does a graph have a negative-weight cycle?
    - does a CNF formula have a solution?
- We also had optimization problems
    - compute shortest path
- Usually, optimization problems have related decision problem
    - is there a path of length less than k?
- We focus on decision problems here!

More Formally

- Describing inputs

More Formally

- Describing inputs
    - formally: sequences of bits: $\{0,1\}^*$

More Formally

- Describing inputs
  - formally: sequences of bits: $\{0, 1\}^*$
  - informally: nice notation adopted to specific problem
    - can obviously be represented as bits
    - and parsed in polynomial time
    - we don't care about the details

More Formally

- Describing inputs
  - formally: sequences of bits: $\{0,1\}^*$
  - informally: nice notation adopted to specific problem
    - can obviously be represented as bits
    - and parsed in polynomial time
    - we don't care about the details
- Decision problem: described by set of accepted words: $L \subseteq \{0,1\}^*$
  - L is called *language*
  - equivalently: decision function: $f :: \{0,1\}^* \to \{0,1\}$, $f(w) = w \in L$

More Formally

- Describing inputs
  - formally: sequences of bits: $\{0, 1\}^*$
  - informally: nice notation adopted to specific problem
    - can obviously be represented as bits
    - and parsed in polynomial time
    - we don't care about the details
- Decision problem: described by set of accepted words: $L \subseteq \{0, 1\}^*$
  - L is called *language*
  - equivalently: decision function: $f :: \{0, 1\}^* \to \{0, 1\}$, $f(w) = w \in L$
- Examples: Language of all bit-strings representing ...

More Formally

- Describing inputs
  - formally: sequences of bits: $\{0, 1\}^*$
  - informally: nice notation adopted to specific problem
    - can obviously be represented as bits
    - and parsed in polynomial time
    - we don't care about the details
- Decision problem: described by set of accepted words: $L \subseteq \{0, 1\}^*$
  - L is called *language*
  - equivalently: decision function: $f :: \{0, 1\}^* \rightarrow \{0, 1\}$, $f(w) = w \in L$
- Examples: Language of all bit-strings representing ...
  - ... sorted lists

More Formally

- Describing inputs
  - formally: sequences of bits: $\{0,1\}^*$
  - informally: nice notation adopted to specific problem
    - can obviously be represented as bits
    - and parsed in polynomial time
    - we don't care about the details
- Decision problem: described by set of accepted words: $L \subseteq \{0,1\}^*$
  - L is called *language*
  - equivalently: decision function: $f :: \{0,1\}^* \to \{0,1\}$, $f(w) = w \in L$
- Examples: Language of all bit-strings representing ...
  - ... sorted lists
  - ... graphs without negative-weight cycles

More Formally

- Describing inputs
  - formally: sequences of bits: $\{0,1\}^*$
  - informally: nice notation adopted to specific problem
    - can obviously be represented as bits
    - and parsed in polynomial time
    - we don't care about the details
- Decision problem: described by set of accepted words: $L \subseteq \{0,1\}^*$
  - L is called *language*
  - equivalently: decision function: $f :: \{0,1\}^* \to \{0,1\}$, $f(w) = w \in L$
- Examples: Language of all bit-strings representing ...
  - ... sorted lists
  - ... graphs without negative-weight cycles
  - ... CNF-formulas that have a solution

Certification

- Certificate: proof that word is in language
    - itself a bit-string

Certification

- Certificate: proof that word is in language
  - itself a bit-string
- Certificate checker $\mathit{check} : (\{0,1\}^*, \{0,1\}^*) \to \{0,1\}$
  - (sound and complete): $(\exists c.\ \mathit{check}(w, c)) \iff w \in L$
  - Certificate checking is decision problem itself!

Certification

- Certificate: proof that word is in language
  - itself a bit-string
- Certificate checker $\mathit{check} : (\{0,1\}^*, \{0,1\}^*) \to \{0,1\}$
  - (sound and complete): $(\exists c.\ \mathit{check}(w, c)) \iff w \in L$
  - Certificate checking is decision problem itself!
- Example: does CNF have solution?
  - certificate is solution (list of variables that are true)
  - certificate checker: evaluate formula

The Complexity Class NP

Definition (NP)

A decision problem is in NP, iff it has polynomial-time checkable certificates

- for word w of size n
  - certificate has size $poly(n)$
  - $check(w, c)$ executable in time $poly(n)$

The Complexity Class NP

Definition (NP)

A decision problem is in NP, iff it has polynomial-time checkable certificates

- for word w of size n
  - certificate has size $poly(n)$
  - $check(w, c)$ executable in time $poly(n)$
- Example: CNF
  - valuation of variables: polynomial size in formula
  - evaluating formula: polynomial time in formula size

The Complexity Class NP

Definition (NP)

A decision problem is in NP, iff it has polynomial-time checkable certificates

- for word w of size n
  - certificate has size $poly(n)$
  - $check(w, c)$ executable in time $poly(n)$
- Example: CNF
  - valuation of variables: polynomial size in formula
  - evaluating formula: polynomial time in formula size
- all polynomial-time solvable problems are in NP
  - the certificate is always empty, and *check* solves the problem

Reduction

Intuitively:

- Convert problem A into problem B, in polynomial time
  - if B is in $O(f(n))$, then A is in $O(poly(n) + f(poly(n)))$
  - time for conversion, and solving A

Reduction

Intuitively:

- Convert problem A into problem B, in polynomial time
  - if B is in $O(f(n))$, then A is in $O(poly(n) + f(poly(n)))$
  - time for conversion, and solving A
- A not harder than B (up to polynomial time factor)

Reduction

Intuitively:

- Convert problem A into problem B, in polynomial time
    - if B is in $O(f(n))$, then A is in $O(poly(n) + f(poly(n)))$
    - time for conversion, and solving A
- A not harder than B (up to polynomial time factor)

Formally:

Definition (polynomial-time reducible)

A language L_A is *polynomial-time reducible* to L_B, iff there is a function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that

1. $w \in L_A \iff f(w) \in L_B$
2. f can be computed in polynomial time

- Notation: $L_A \leq_p L_B$

Reduction and NP

Theorem

If $L_A \leq_p L_B$ *and* $L_B \in NP$, *then* $L_A \in NP$

Reduction and NP

Theorem

If $L_A \leq_p L_B$ *and* $L_B \in NP$*, then* $L_A \in NP$

Proof.

- $check_A(w, c) = check_B(f(w), c)$ is certificate checker.
- and executable in polynomial time.
- note: $f(w)$ can increase word size only polynomially.

□

NP-Hardness

Definition (NP-Hardness)

A problem is NP-hard, if every problem in NP can be reduced to it

NP-Hardness

Definition (NP-Hardness)

A problem is NP-hard, if every problem in NP can be reduced to it

- Intuition: harder (or equally hard) than any problem in NP

NP-Hardness

Definition (NP-Hardness)

A problem is NP-hard, if every problem in NP can be reduced to it

- Intuition: harder (or equally hard) than any problem in NP
- NP-complete = In NP and NP-hard

NP-Hardness

Definition (NP-Hardness)

A problem is NP-hard, if every problem in NP can be reduced to it

- Intuition: harder (or equally hard) than any problem in NP
- NP-complete = In NP and NP-hard
- Many interesting problems turn out to be NP-complete

NP-Completeness

- Intuitively: As hard as any problem in NP

NP-Completeness

- Intuitively: As hard as any problem in NP
- If one NP-complete problem A has poly-time algorithm

NP-Completeness

- Intuitively: As hard as any problem in NP
- If one NP-complete problem A has poly-time algorithm then all problems B in NP have poly-time algorithm!

NP-Completeness

- Intuitively: As hard as any problem in NP
- If one NP-complete problem A has poly-time algorithm then all problems B in NP have poly-time algorithm!
    - reduce B to A (in poly-time), then solve A

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
    - needs exponential time: $O(2^{poly(n)})$

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!
  - experts' expectation: $P \neq NP$
    - i.e., NP-hard problems intractable.

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!
  - experts' expectation: $P \neq NP$
    - i.e., NP-hard problems intractable.
- Why do we care then?

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!
  - experts' expectation: $P \neq NP$
    - i.e., NP-hard problems intractable.
- Why do we care then?
  - if you recognize you have an NP-hard problem:

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!
  - experts' expectation: $P \neq NP$
    - i.e., NP-hard problems intractable.
- Why do we care then?
  - if you recognize you have an NP-hard problem:
    - stop searching for efficient algorithms

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!
  - experts' expectation: $P \neq NP$
    - i.e., NP-hard problems intractable.
- Why do we care then?
  - if you recognize you have an NP-hard problem:
    - stop searching for efficient algorithms
    - can you consider easier subclass of problem instead?

Complexity of NP-complete Problems

- Naive algorithm: for word of size n, enumerate all $2^{poly(n)}$ certificates, and check each in polynomial time.
  - needs exponential time: $O(2^{poly(n)})$
- Can we do better? E.g. polynomial time?
  - we don't know!
  - P vs NP conjecture. One of the Millenium-Prize Problems!
  - experts' expectation: $P \neq NP$
    - i.e., NP-hard problems intractable.
- Why do we care then?
  - if you recognize you have an NP-hard problem:
    - stop searching for efficient algorithms
    - can you consider easier subclass of problem instead?
    - can you live with approximation? (for optimization problems)

Proving Problems NP-Complete

- to show problem is in NP

Proving Problems NP-Complete

- to show problem is in NP
    - find polynomial-time checkable certificate!

Proving Problems NP-Complete

- to show problem is in NP
    - find polynomial-time checkable certificate!
    - often, that's surprisingly easy!

Proving Problems NP-Complete

- to show problem is in NP
  - find polynomial-time checkable certificate!
  - often, that's surprisingly easy!
  - given a puzzle, finding solution much harder than checking and often, it's obvious how to check a solution!

Proving Problems NP-Complete

- to show problem is in NP
  - find polynomial-time checkable certificate!
  - often, that's surprisingly easy!
  - given a puzzle, finding solution much harder than checking and often, it's obvious how to check a solution!
- to show problem A is NP-hard

Proving Problems NP-Complete

- to show problem is in NP
  - find polynomial-time checkable certificate!
  - often, that's surprisingly easy!
  - given a puzzle, finding solution much harder than checking and often, it's obvious how to check a solution!
- to show problem A is NP-hard
  - reduce an already known NP-hard problem B to it

Proving Problems NP-Complete

- to show problem is in NP
  - find polynomial-time checkable certificate!
  - often, that's surprisingly easy!
  - given a puzzle, finding solution much harder than checking and often, it's obvious how to check a solution!
- to show problem A is NP-hard
  - reduce an already known NP-hard problem B to it
  - show $B \leq_p A$. already known: $\forall C \in NP.\ C \leq_p B$
    - by transitivity: $\forall C \in NP.\ C \leq_p A$

Proving Problems NP-Complete

- to show problem is in NP
    - find polynomial-time checkable certificate!
    - often, that's surprisingly easy!
    - given a puzzle, finding solution much harder than checking and often, it's obvious how to check a solution!
- to show problem A is NP-hard
    - reduce an already known NP-hard problem B to it
    - show $B \leq_p A$. already known: $\forall C \in NP.\ C \leq_p B$
        - by transitivity: $\forall C \in NP.\ C \leq_p A$
- Now: obtaining an initial NP-hard problem.