Computational complexity

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

2 May 2014

Introduction

- If we can prove that a language is decidable, that does not mean we can solve the corresponding decision problem in practice.
- Computational complexity theory studies the question of which problems we can solve given restricted resources: in particular, restricted time or space.
- We are generally interested in how the resources we need to solve a family of problems grow with problem size.
- This allows us to compare the complexity of different problems and formalise the intuitive notion that some problems are harder than others.

Time complexity

Time complexity

Let M be a deterministic Turing machine that halts on all inputs. The running time or time complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of length n.

Time complexity

Time complexity

Let M be a deterministic Turing machine that halts on all inputs. The running time or time complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of length n.

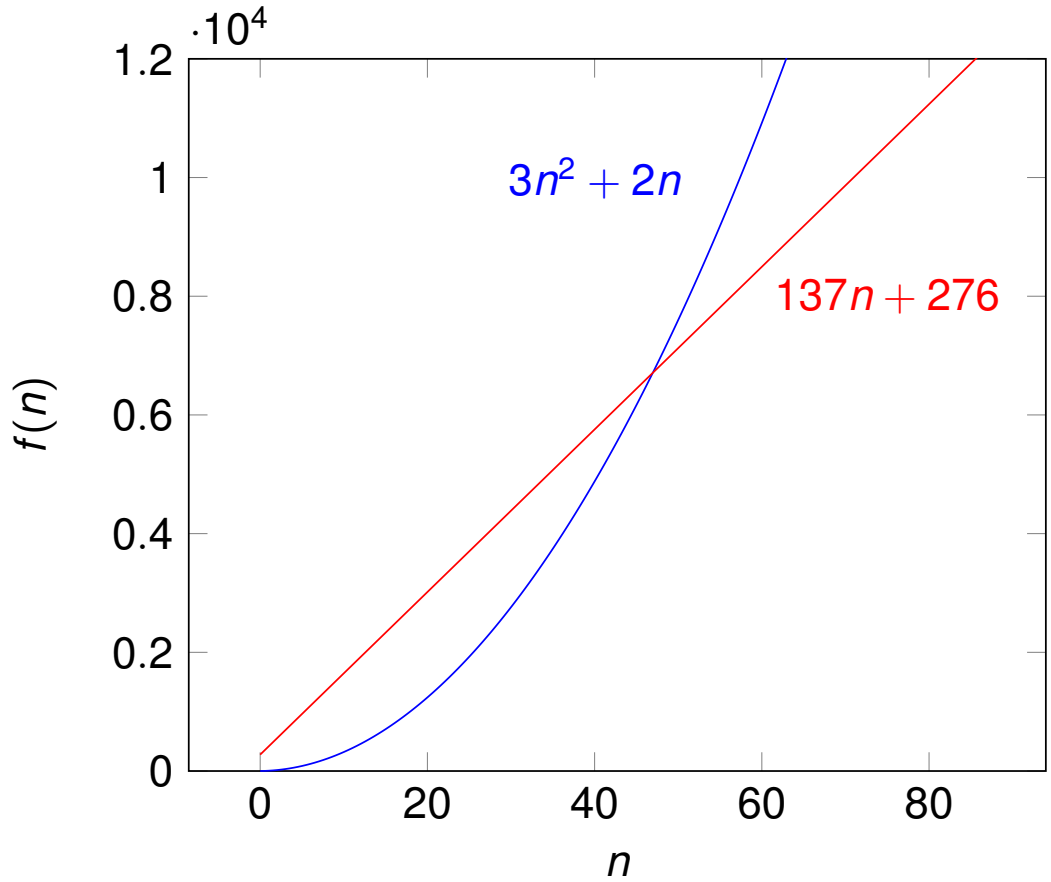
Some notes on this definition:

- A "step" is a transition, which includes reading a symbol, changing state and writing a new symbol to the tape.
- This is a worst-case notion of complexity, i.e. we define the running time of M on inputs of a certain length to be the number of steps that M takes on the worst possible input of that length.
- We usually use n to represent the length of the input.

When comparing running times we often only care about the scaling behaviour with the input size n.

When comparing running times we often only care about the scaling behaviour with the input size n.

- For example, given two Turing machines M_1 and M_2 with running times $3n^2 + 2n$ and $137n + 276$ respectively, for large n the $3n^2$ term in the running time of the first machine will dominate.



Big-O notation

Big-O notation allows us to simplify expressions for running times etc. while still retaining the important features.

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions. We write $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that, for all integers $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

Big-O notation

Big-O notation allows us to simplify expressions for running times etc. while still retaining the important features.

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions. We write $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that, for all integers $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

For example:

- $f(n) = 3n^2 + n$: $f(n) = O(n^2)$
- $f(n) = 0.01n^2 + 0.001n^3$: $f(n) = O(n^3)$, but $f(n)$ is not $O(n^2)$.

Time complexity classes

Definition

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Then TIME($t(n)$) is the set of all languages which are decidable by a Turing machine running in time $O(t(n))$.

Time complexity classes

Definition

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Then TIME($t(n)$) is the set of all languages which are decidable by a Turing machine running in time $O(t(n))$.

For example, recall the language

$$\mathcal{L}_{EQ} = \{w\#w \mid w \in \{0,1\}^*\}$$

of two equal bit-strings, separated by a $\#$ symbol.

Claim: $\mathcal{L}_{EQ} \in$ TIME(n^2).

Time complexity classes

Recall the following algorithm for deciding this language:

Algorithm for deciding $\mathcal{L}_{EQ}$ (sketch)

1. Scan forwards and backwards, testing each corresponding pair of bits either side of the $\#$ for equality in turn.
2. Overwrite each bit with an $\mathtt{x}$ symbol after checking it so we don't check the same bits twice.
3. Accept if all pairs of bits match each other; otherwise reject.

Time complexity classes

Recall the following algorithm for deciding this language:

Algorithm for deciding $\mathcal{L}_{EQ}$ (sketch)

1. Scan forwards and backwards, testing each corresponding pair of bits either side of the $\#$ for equality in turn.
2. Overwrite each bit with an $\mathtt{x}$ symbol after checking it so we don't check the same bits twice.
3. Accept if all pairs of bits match each other; otherwise reject.

Bounding the time complexity (sketch):

- In the worst case, the input string is of the form $w\#w$ for some string $w \in \{0,1\}^m$.

Time complexity classes

Recall the following algorithm for deciding this language:

Algorithm for deciding $\mathcal{L}_{EQ}$ (sketch)

1. Scan forwards and backwards, testing each corresponding pair of bits either side of the $\#$ for equality in turn.
2. Overwrite each bit with an $\mathtt{x}$ symbol after checking it so we don't check the same bits twice.
3. Accept if all pairs of bits match each other; otherwise reject.

Bounding the time complexity (sketch):

- In the worst case, the input string is of the form $w\#w$ for some string $w \in \{0,1\}^m$.
- For each symbol in the part of the string to the left of $\#$, $m+1$ moves to the right are made, the corresponding symbol to the right of $\#$ is checked and $m+1$ moves to the left are made.

Time complexity classes

Recall the following algorithm for deciding this language:

Algorithm for deciding $\mathcal{L}_{EQ}$ (sketch)

1. Scan forwards and backwards, testing each corresponding pair of bits either side of the $\#$ for equality in turn.
2. Overwrite each bit with an $\mathtt{x}$ symbol after checking it so we don't check the same bits twice.
3. Accept if all pairs of bits match each other; otherwise reject.

Bounding the time complexity (sketch):

- In the worst case, the input string is of the form $w\#w$ for some string $w \in \{0,1\}^m$.
- For each symbol in the part of the string to the left of $\#$, $m+1$ moves to the right are made, the corresponding symbol to the right of $\#$ is checked and $m+1$ moves to the left are made.
- So the algorithm runs in time $O(m^2)$, which is also $O(n^2)$.

Polynomial and exponential time

- We would like to distinguish between algorithms which are efficient (run in a reasonable length of time) and algorithms which are inefficient.
- One way to do this is via the concepts of polynomial-time and exponential-time algorithms.

Polynomial and exponential time

- We would like to distinguish between algorithms which are efficient (run in a reasonable length of time) and algorithms which are inefficient.
- One way to do this is via the concepts of polynomial-time and exponential-time algorithms.

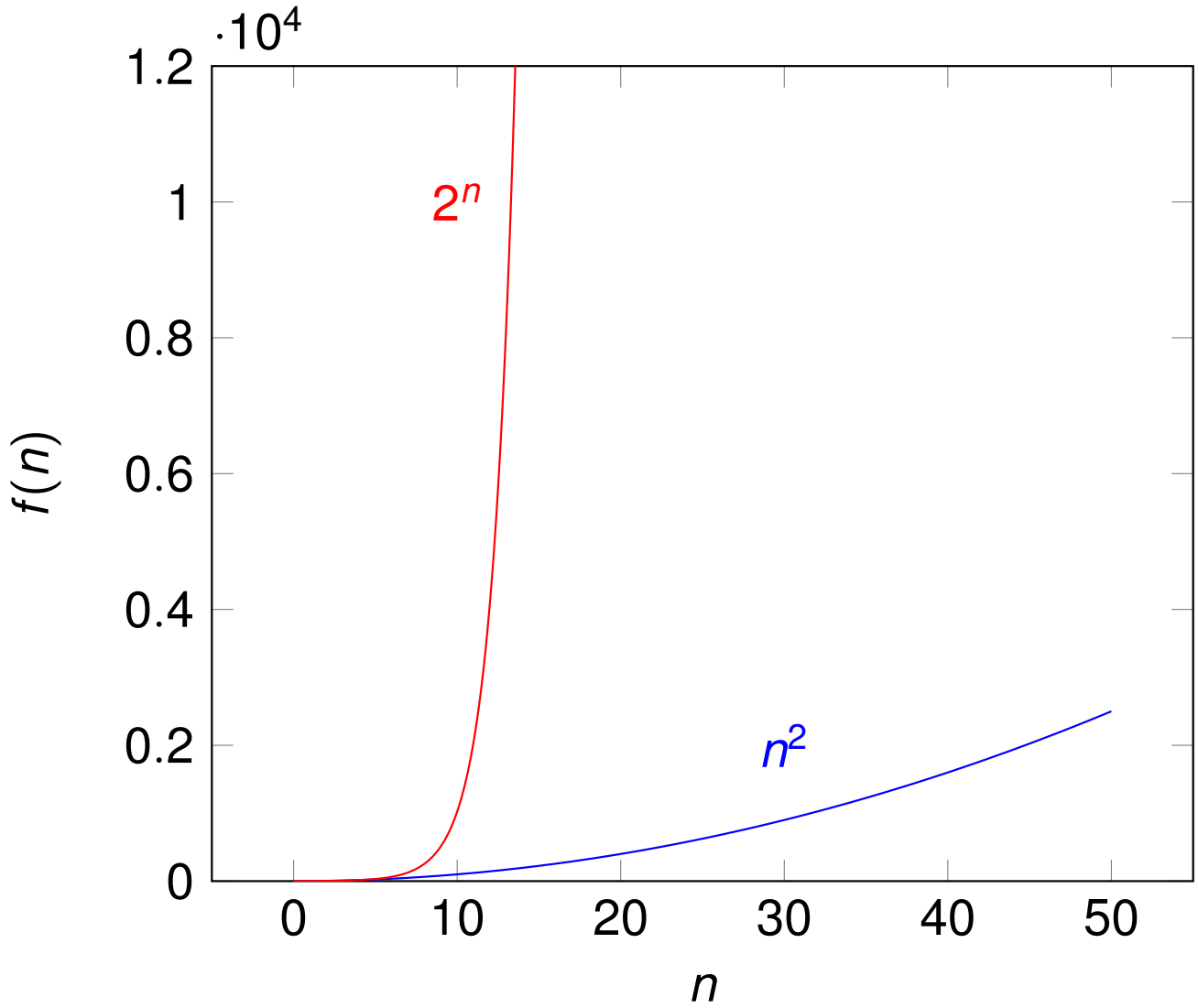
Definitions

$$\mathrm{P} = \bigcup_{k \geq 0} \mathrm{TIME}(n^k)$$

$$\mathrm{EXP} = \bigcup_{k \geq 0} \mathrm{TIME}(2^{n^k})$$

So P is the class of languages that are decided by Turing machines with runtime polynomial in the input size. EXP is the class of languages decided by Turing machines with runtime exponential in the input size.

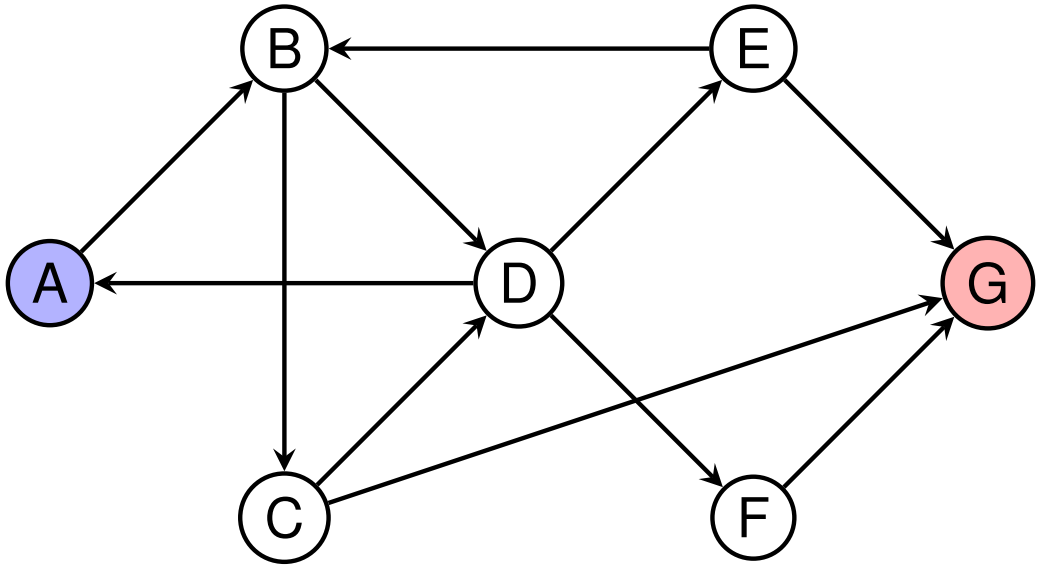
Polynomial and exponential time



Some examples of languages in P

Consider the language PATH defined by

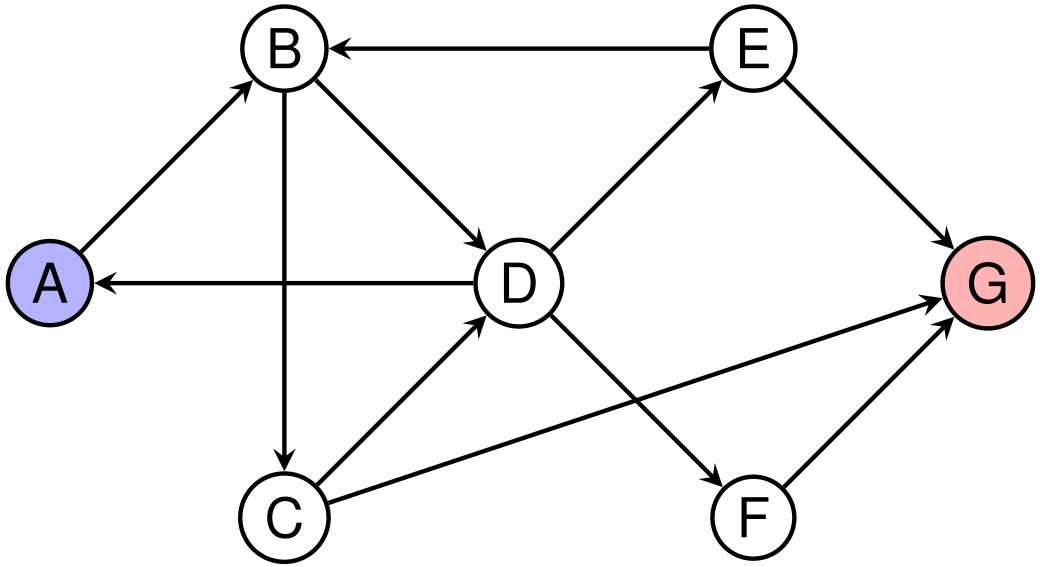
$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$



Some examples of languages in P

Consider the language PATH defined by

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$

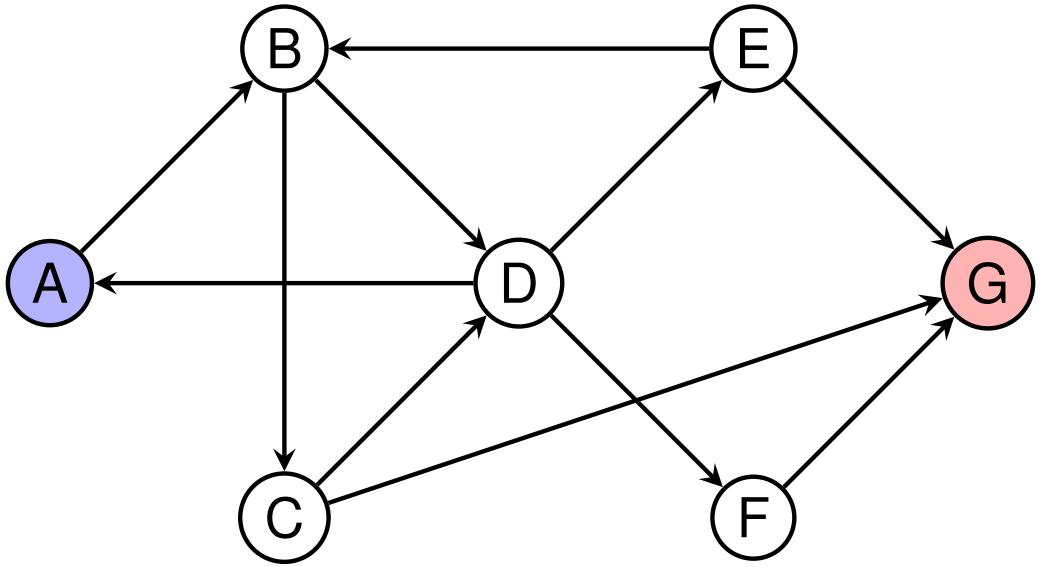


- PATH is a formalisation of the problem of determining whether there is a path between two specified points in a directed graph.

Some examples of languages in P

Consider the language PATH defined by

$$\text{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a path from } s \text{ to } t\}$$



- PATH is a formalisation of the problem of determining whether there is a path between two specified points in a directed graph.
- A naïve algorithm would be to try every possible path (sequence of vertices) from s to t in turn to see if that path exists. But if G has m vertices, in the worst case this could involve checking $\sim m^m$ paths...

Some examples of languages in P

A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex s.
2. Repeat until no further vertices are found:
   - Check all the edges in G. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If t is marked, accept; otherwise reject.

Some examples of languages in P

A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex s.
2. Repeat until no further vertices are found:
   - Check all the edges in G. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If t is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Assume the input graph G is on m vertices and is provided as an adjacency matrix, so the input size $n = O(m^2)$.

Some examples of languages in P

A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex s.
2. Repeat until no further vertices are found:
   - Check all the edges in G. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If t is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Assume the input graph G is on m vertices and is provided as an adjacency matrix, so the input size $n = O(m^2)$.
- Step 2 is repeated at most m times and checks at most m^2 edges.

Some examples of languages in P

A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex s.
2. Repeat until no further vertices are found:
   - Check all the edges in G. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If t is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Assume the input graph G is on m vertices and is provided as an adjacency matrix, so the input size $n = O(m^2)$.
- Step 2 is repeated at most m times and checks at most m^2 edges.
- Checking each edge in step 2 can be done in time $O(m^k)$ for some fixed k (depending on the computational model).

Some examples of languages in P

A polynomial-time algorithm deciding PATH

On input $\langle G, s, t \rangle$:

1. Place a mark on vertex s.
2. Repeat until no further vertices are found:
   - Check all the edges in G. If an edge is found from a marked vertex to an unmarked vertex, mark the target of the edge.
3. If t is marked, accept; otherwise reject.

A rough upper bound on the running time of this algorithm:

- Assume the input graph G is on m vertices and is provided as an adjacency matrix, so the input size $n = O(m^2)$.
- Step 2 is repeated at most m times and checks at most m^2 edges.
- Checking each edge in step 2 can be done in time $O(m^k)$ for some fixed k (depending on the computational model).
- So the running time of the algorithm is $O(m^{k+3})$, which is poly(n).

Some examples of languages in P

Many other important problems are known to be in P. For example:

- The language

  $$\text{PRIMES} = \{x \in \{0,1\}^* \mid x \text{ is a prime number written in binary}\}$$

  is in P but this was only proven in 2002 (by two undergraduate students and a professor).

Some examples of languages in P

Many other important problems are known to be in P. For example:

- The language

  $$\text{PRIMES} = \{x \in \{0,1\}^* \mid x \text{ is a prime number written in binary}\}$$

  is in P but this was only proven in 2002 (by two undergraduate students and a professor).

- Every context-free language is in P. The algorithm we presented showing decidability of CFLs does not imply this (why not?), but the CYK Algorithm you saw in Algorithms and Programming does.

Some examples of languages in P

Many other important problems are known to be in P. For example:

- The language

  $$\text{PRIMES} = \{x \in \{0,1\}^* \mid x \text{ is a prime number written in binary}\}$$

  is in P but this was only proven in 2002 (by two undergraduate students and a professor).

- Every context-free language is in P. The algorithm we presented showing decidability of CFLs does not imply this (why not?), but the CYK Algorithm you saw in Algorithms and Programming does.

- Other examples include evaluation of circuits, finding shortest paths, pattern matching, linear programming, ...

Problems whose solutions are easy to check

There are problems which may not be easy to solve, but for which it is easy to check a claimed solution.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Pics: Wikipedia/Sudoku

Problems whose solutions are easy to check

There are problems which may not be easy to solve, but for which it is easy to check a claimed solution.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Pics: Wikipedia/Sudoku

Problems whose solutions are easy to check

We can formalise this using the notion of a verifier.

Definition

A verifier for a language $\mathcal{L}$ is a Turing machine V such that

$$\mathcal{L} = \{x \mid \text{there exists a string } c \text{ such that } V \text{ accepts } \langle x, c \rangle\}.$$

- We think of c as a proof or witness that $x \in \mathcal{L}$, which V can check.

Problems whose solutions are easy to check

We can formalise this using the notion of a verifier.

Definition

A verifier for a language $\mathcal{L}$ is a Turing machine V such that

$$\mathcal{L} = \{x \mid \text{there exists a string } c \text{ such that } V \text{ accepts } \langle x, c \rangle\}.$$

- We think of c as a proof or witness that $x \in \mathcal{L}$, which V can check.
- V should accept a correct proof, but not be fooled by any claimed incorrect proof.

Problems whose solutions are easy to check

We can formalise this using the notion of a verifier.

Definition

A verifier for a language $\mathcal{L}$ is a Turing machine V such that

$$\mathcal{L} = \{x \mid \text{there exists a string } c \text{ such that } V \text{ accepts } \langle x, c \rangle\}.$$

- We think of c as a proof or witness that $x \in \mathcal{L}$, which V can check.
- V should accept a correct proof, but not be fooled by any claimed incorrect proof.
- A polynomial-time verifier is a verifier which runs in time polynomial in the length of the input x, i.e. $O(|x|^k)$ for some fixed k.

Definition

NP is the class of languages which have a polynomial-time verifier.

Problems whose solutions are easy to check

An example of a language in NP:

FACTORING $= \{\langle x, y\rangle \mid x$ is an integer with a prime factor lower than $y\}$.

- For example, $15, 4$ is in the language, but $15, 2$ is not.

Problems whose solutions are easy to check

An example of a language in NP:

FACTORING $= \{\langle x, y\rangle \mid x$ is an integer with a prime factor lower than $y\}$.

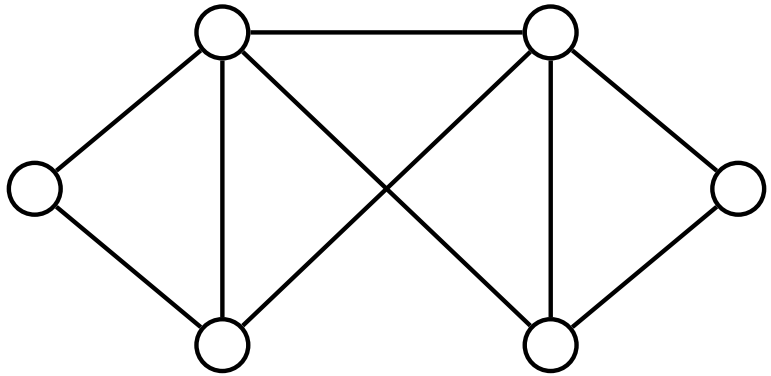
- For example, $15, 4$ is in the language, but $15, 2$ is not.
- For any x, the list of prime factors of x can be used as a proof to determine whether $\langle x, y\rangle$ is in the language.
- Given some claimed prime factors of x, we can multiply them together to determine whether we get x. If so, we check whether the smallest of them is lower than y.

Problems whose solutions are easy to check

Another example is graph colouring problems. We say a graph can be properly k-coloured if each vertex can be assigned one of k colours, such that all pairs of adjacent vertices have different colours.

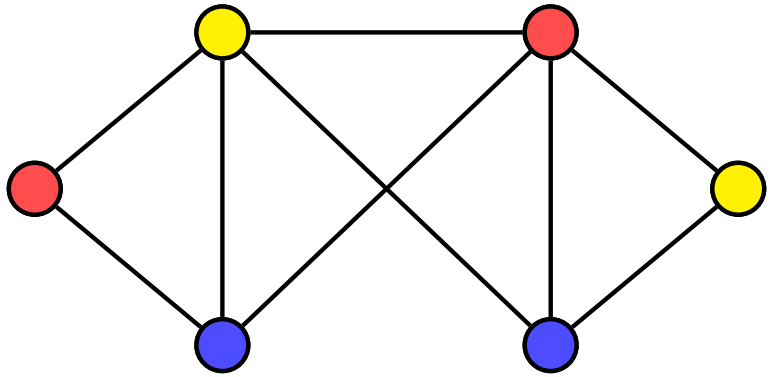
Problems whose solutions are easy to check

Another example is graph colouring problems. We say a graph can be properly k-coloured if each vertex can be assigned one of k colours, such that all pairs of adjacent vertices have different colours.



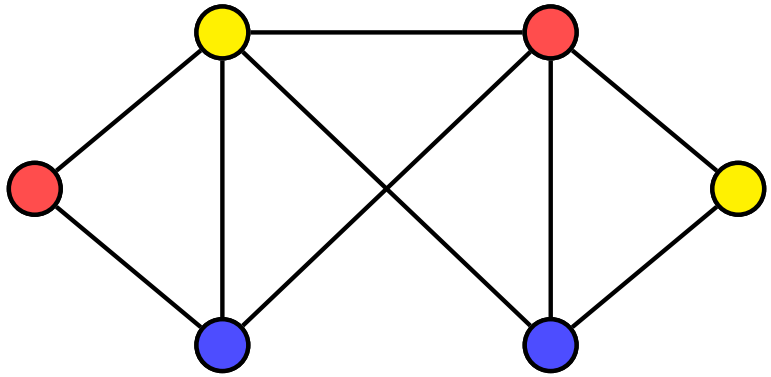
Problems whose solutions are easy to check

Another example is graph colouring problems. We say a graph can be properly k-coloured if each vertex can be assigned one of k colours, such that all pairs of adjacent vertices have different colours.



Problems whose solutions are easy to check

Another example is graph colouring problems. We say a graph can be properly k-coloured if each vertex can be assigned one of k colours, such that all pairs of adjacent vertices have different colours.



We can formalise this decision problem as the language

$$\text{3-COLOURING} = \{\langle G \rangle \mid G \text{ is a graph which can be properly 3-coloured}\}.$$

- Given a graph, we can be convinced that it can be properly 3-coloured by being given a 3-colouring and checking that it is proper; so 3-COLOURING is in NP.

P vs. NP

- Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.

P vs. NP

- Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.

- Also, every language in NP is also in EXP. If there exists an m-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.

P vs. NP

- Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.
- Also, every language in NP is also in EXP. If there exists an m-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.
- So $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP}$.

P vs. NP

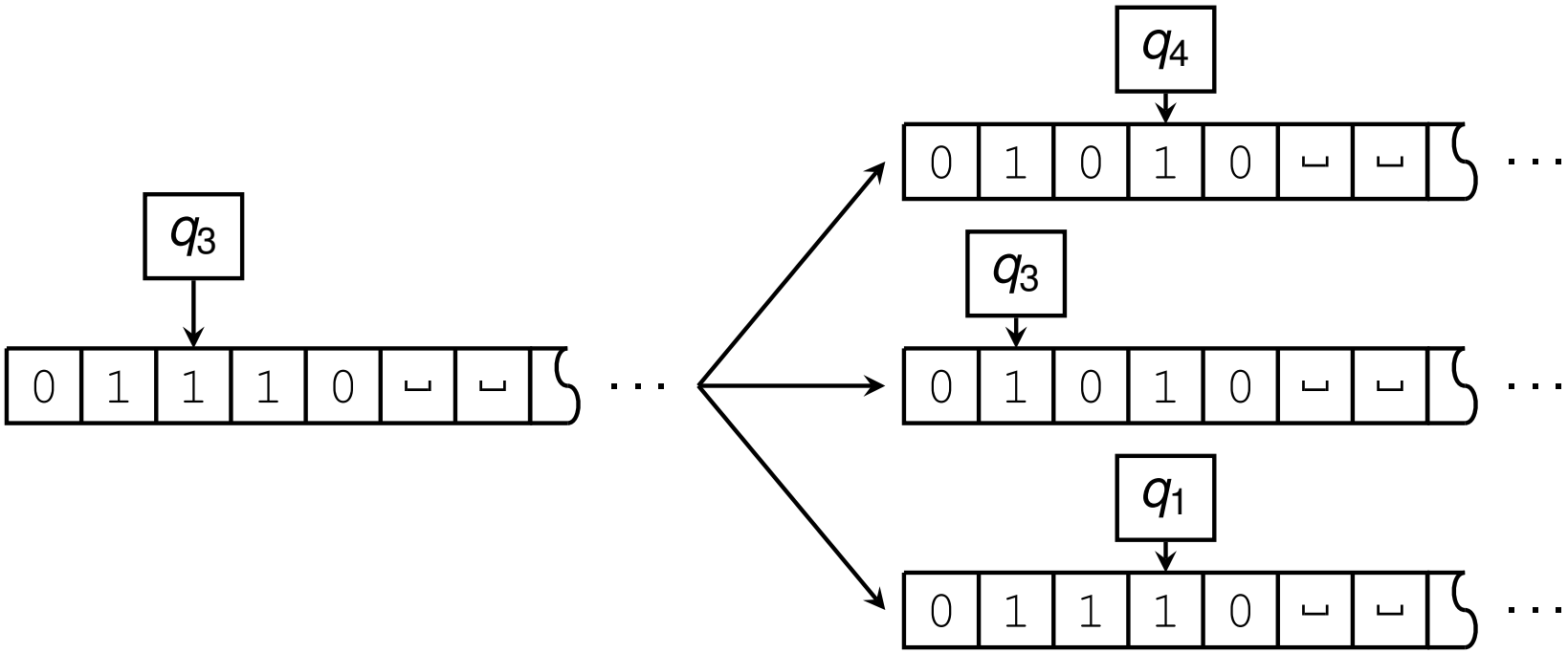
- Every language in P is also in NP. For any language in P, the verifier can ignore any claimed proof and just decide the language directly.
- Also, every language in NP is also in EXP. If there exists an m-bit witness that the input is in a language, by looping over all possible witnesses a Turing machine can find that witness in time $O(2^m)$.
- So $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP}$.

It is not known whether $\mathrm{P} = \mathrm{NP}$ and this question is considered the most important open problem in computer science!

- Resolving it would win you everlasting fame (as well as $1M).

Nondeterministic polynomial-time

Recall that a nondeterministic Turing machine (NDTM) can explore multiple computational paths simultaneously. It accepts if and only if at least one of the computational paths accepts.



Nondeterministic polynomial-time

Definitions

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Then NTIME($t(n)$) is the set of all languages which are decidable by a nondeterministic Turing machine running in time $O(t(n))$.

- We say that an NDTM runs in time $O(t(n))$ if all of its computational paths halt in time $O(t(n))$.

Nondeterministic polynomial-time

Definitions

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Then NTIME($t(n)$) is the set of all languages which are decidable by a nondeterministic Turing machine running in time $O(t(n))$.

- We say that an NDTM runs in time $O(t(n))$ if all of its computational paths halt in time $O(t(n))$.

We will now see that there is a close connection between nondeterministic Turing machines and verification of proofs.

This will explain the name NP, which stands for "Nondeterministic Polynomial-time" (and not "non-polynomial time").

Nondeterministic polynomial-time

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Nondeterministic polynomial-time

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

We first show that $\mathcal{L} \in \mathrm{NP}$ implies that $\mathcal{L}$ is decided by an NDTM running in polynomial time (i.e. $\mathrm{NP} \subseteq \bigcup_{k \geq 0} \mathrm{NTIME}(n^k)$).

Nondeterministic polynomial-time

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

We first show that $\mathcal{L} \in \mathrm{NP}$ implies that $\mathcal{L}$ is decided by an NDTM running in polynomial time (i.e. $\mathrm{NP} \subseteq \bigcup_{k \geq 0} \mathrm{NTIME}(n^k)$).

- From the definition of NP, there exists a verifier V for $\mathcal{L}$ running in time $O(n^k)$ for some fixed k.

Nondeterministic polynomial-time

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

We first show that $\mathcal{L} \in \mathrm{NP}$ implies that $\mathcal{L}$ is decided by an NDTM running in polynomial time (i.e. $\mathrm{NP} \subseteq \bigcup_{k \geq 0} \mathrm{NTIME}(n^k)$).

- From the definition of NP, there exists a verifier V for $\mathcal{L}$ running in time $O(n^k)$ for some fixed k.
- We define an NDTM N which behaves as follows:
  1. Nondeterministically guess a string c of length at most $O(n^k)$.
  2. Run V on input x, c. If V accepts, accept; otherwise, reject.

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

For the other direction, we show that if $\mathcal{L}$ is decided by an NDTM running in polynomial time, $\mathcal{L} \in \mathrm{NP}$ (i.e. $\bigcup_{k \geq 0} \mathrm{NTIME}(n^k) \subseteq \mathrm{NP}$).

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

For the other direction, we show that if $\mathcal{L}$ is decided by an NDTM running in polynomial time, $\mathcal{L} \in \mathrm{NP}$ (i.e. $\bigcup_{k \geq 0} \mathrm{NTIME}(n^k) \subseteq \mathrm{NP}$).

- From the definition of NTIME, there exists an NDTM N which decides $\mathcal{L}$ and runs in time $O(n^k)$ for some fixed k.

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

For the other direction, we show that if $\mathcal{L}$ is decided by an NDTM running in polynomial time, $\mathcal{L} \in \mathrm{NP}$ (i.e. $\bigcup_{k \geq 0} \mathrm{NTIME}(n^k) \subseteq \mathrm{NP}$).

- From the definition of NTIME, there exists an NDTM N which decides $\mathcal{L}$ and runs in time $O(n^k)$ for some fixed k.
- We define a verifier V which takes as input a string x and a witness c. c is the description of a sequence of computational path choices made by N.

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

For the other direction, we show that if $\mathcal{L}$ is decided by an NDTM running in polynomial time, $\mathcal{L} \in \mathrm{NP}$ (i.e. $\bigcup_{k \geq 0} \mathrm{NTIME}(n^k) \subseteq \mathrm{NP}$).

- From the definition of NTIME, there exists an NDTM N which decides $\mathcal{L}$ and runs in time $O(n^k)$ for some fixed k.
- We define a verifier V which takes as input a string x and a witness c. c is the description of a sequence of computational path choices made by N.
- V simulates the computation of N on input x according to c, checking whether each transition made is valid.

Theorem

$$\mathrm{NP} = \bigcup_{k \geq 0} \mathrm{NTIME}(n^k).$$

Proof (sketch)

For the other direction, we show that if $\mathcal{L}$ is decided by an NDTM running in polynomial time, $\mathcal{L} \in \mathrm{NP}$ (i.e. $\bigcup_{k \geq 0} \mathrm{NTIME}(n^k) \subseteq \mathrm{NP}$).

- From the definition of NTIME, there exists an NDTM N which decides $\mathcal{L}$ and runs in time $O(n^k)$ for some fixed k.
- We define a verifier V which takes as input a string x and a witness c. c is the description of a sequence of computational path choices made by N.
- V simulates the computation of N on input x according to c, checking whether each transition made is valid.
- If N accepts in the end, V accepts; otherwise, V rejects.

NP-completeness

Some languages in the class NP are known to actually be NP-complete.

- A language $\mathcal{L}$ is said to be NP-complete if:
  - $\mathcal{L} \in$ NP;
  - Every language $\mathcal{L}' \in$ NP reduces to $\mathcal{L}$ in polynomial time.
- So NP-complete problems are the hardest problems in NP: if we can solve them efficiently, we can solve every other problem in NP efficiently.

NP-completeness

Some languages in the class NP are known to actually be NP-complete.

- A language $\mathcal{L}$ is said to be NP-complete if:
  - $\mathcal{L} \in$ NP;
  - Every language $\mathcal{L}' \in$ NP reduces to $\mathcal{L}$ in polynomial time.
- So NP-complete problems are the hardest problems in NP: if we can solve them efficiently, we can solve every other problem in NP efficiently.

Examples of NP-complete problems include 3-colouring, clique finding in graphs, optimal packing problems . . . in fact, many of the most practically important problems in computer science.

Summary and further reading

- Computational complexity theory allows us to study the resources required to solve problems.
- Problems can be classified according to the time required to solve them in the deterministic Turing machine model.
- P is the class of problems which can be solved in time polynomial in the input size, while NP is the class of problems whose solutions can be verified in time polynomial in the input size.
- NP has an alternative definition in terms of nondeterministic Turing machines.

Further reading: Sipser §7.1–7.3.