Topics in Quantum Engineering Presentation: Complexity Theory

Dominic Moylett

University of Bristol

dominic.moylett@bristol.ac.uk

November 27, 2015

Complexity theory in a nutshell

"How hard can it be?", *Clarkson, Hammond, May*

What is complexity theory?

Complexity theory is the study of what problems are easy to solve with a computer.

What is complexity theory?

Complexity theory is the study of what problems are easy to solve with a computer.

When is a problem easy?

What is complexity theory?

Complexity theory is the study of what problems are easy to solve with a computer.

What is a problem?

What is complexity theory?

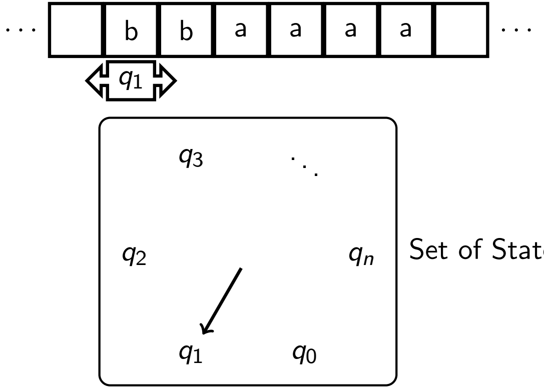
Complexity theory is the study of what problems are easy to solve with a computer.

What is a computer?

Structure of part one

- What is a computer?
- What is a problem?
- When is a problem easy?

Our model of a computer



Machine Tape (RAM) with head at letter b

Set of States (CPU) with current state q_0

[1]

[1] http://www.texample.net/tikz/examples/turing-machine-2/

Formal definition of a computer

A Turing Machine (TM) is specified as a tuple of seven components $(Q, \Sigma, \Gamma, q_0, q_{accept}, q_{reject})$:

- Q is the set of all possible states
- Σ is the input alphabet for the tape. Note that Σ cannot contain the blank symbol $\sqcup$
- Γ is tape alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma$
- $q_0 \in Q$ is the start state
- $q_{accept} \in Q$ is the accept state
- $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$
- $\delta : Q \times \Gamma \rightarrow Q \times \Sigma \times \{L, R\}$ is the transition function

But how do we run it?

The majority of computation time is spent repeating the following loop. Note that T_h is the h-th cell of the tape.

```
q = q_0
h = 0
T = w                    // Tape starts as just input, followed by blank cells
while q ∉ {q_accept, q_reject} do
    (q, h, i) = δ(q, T_h)
    if i = L then
        h = h − 1        // Move tape head to the left
    else
        h = h + 1        // Move tape head to the right
    end
end
```

What happens when it stops?

When we reach either the *accept* or *reject* state, the machine has *halted*.

If the machine *accepted*, then the machine accepts and the contents of the tape, denoted $M(w)$, is returned.

If the machine *rejected*, then the machine rejects and nothing is returned.

What can we store in a machine's memory?

- Integers

What can we store in a machine's memory?

- Integers
- Rational numbers

What can we store in a machine's memory?

- Integers
- Rational numbers
- Floating point numbers

What can we store in a machine's memory?

- Integers
- Rational numbers
- Floating point numbers
- Boolean (True/False) statements

What can we store in a machine's memory?

- Integers
- Rational numbers
- Floating point numbers
- Boolean (True/False) statements
- Text

What can we store in a machine's memory?

- Integers
- Rational numbers
- Floating point numbers
- Boolean (True/False) statements
- Text
- Graphs

What can we store in a machine's memory?

- Integers
- Rational numbers
- Floating point numbers
- Boolean (True/False) statements
- Text
- Graphs
- Other machines

Universal Turing Machines

Turing showed in his PhD thesis that we could represent any *TM* after any number of transitions – including current state, tape contents and position of tape head – as an integer.

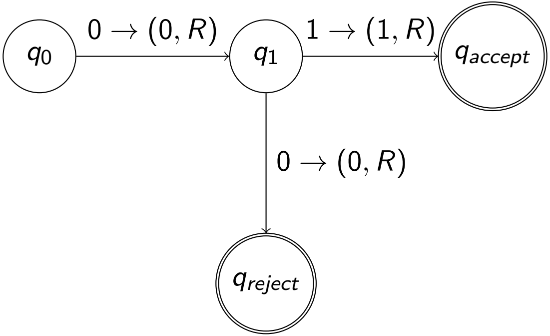
Not only that, but we could manipulate this integer such that it matched performing the next step of the computation.

This gave way to Universal Turing Machines; machines capable of running any *TM* given to them.

The Church-Turing Thesis

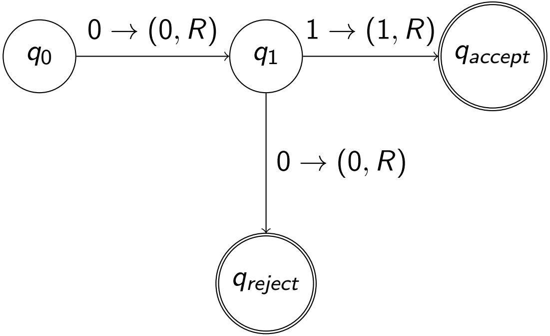
"[A]ll effectively calculable sequences are computable", *Turing*

Quiz Time!



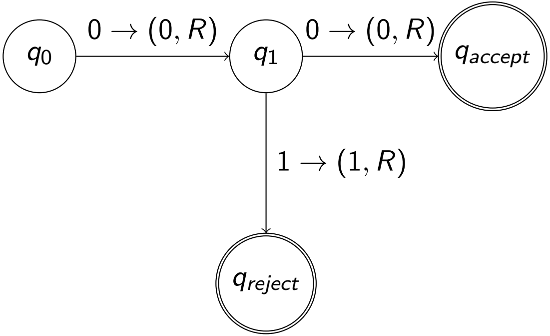
Question: Does $M(01)$ accept or reject?

Quiz Time!



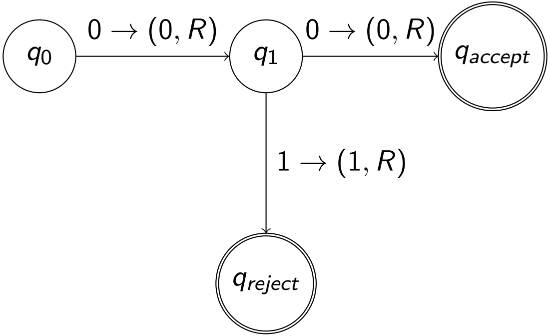
Answer: $M(01)$ accepts!

Quiz Time!



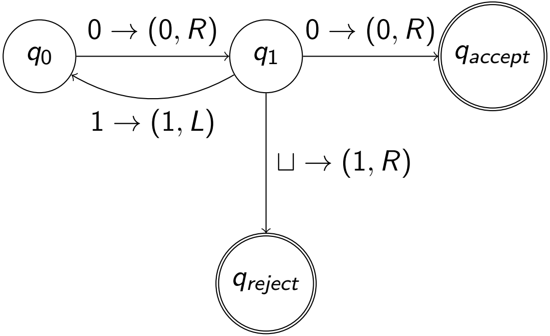
Question: Does $M(01)$ accept or reject?

Quiz Time!



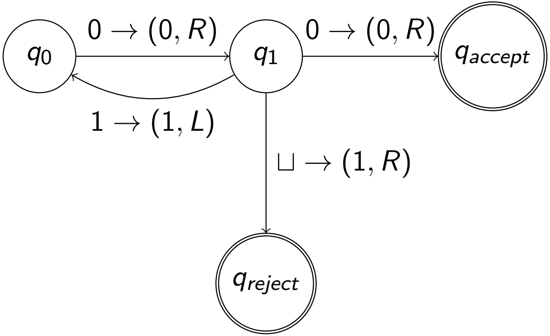
Answer: $M(01)$ rejects!

Quiz Time!



Question: Does $M(01)$ accept or reject?

Quiz Time!



Answer: $M(w)$ doesn't halt.

The halting problem

You might think it would be useful if we could tell when a machine was going to halt.

Formally, we want a TM H such that given $M \in TM$ and $w \in \Sigma_M^*$:

- $H(M, w)$ halts in the accept state if $M(w)$ halts and
- $H(M, w)$ halts in the reject state if $M(w)$ does not halt.

Sadly, Turing proved that such a machine is impossible.

There are many other unsolvable problems as well, within the area of **computability theory**. We will not cover this area, but some reading on the subject is suggested at the end.

Nondeterminism: Spot the difference!

A Nondeterministic Turing Machine (NTM) is specified as a tuple of seven components $(Q, \Sigma, \Gamma, q_0, q_{accept}, q_{reject})$:

- Q is the set of all possible states
- Σ is the input alphabet for the tape. Note that Σ cannot contain the blank symbol $\sqcup$
- Γ is tape alphabet, where $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma$
- $q_0 \in Q$ is the start state
- $q_{accept} \in Q$ is the accept state
- $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$
- $\delta : Q \times \Gamma \rightarrow (Q \times \Sigma \times \{L, R\})^*$ is the transition function

What, what's the difference?

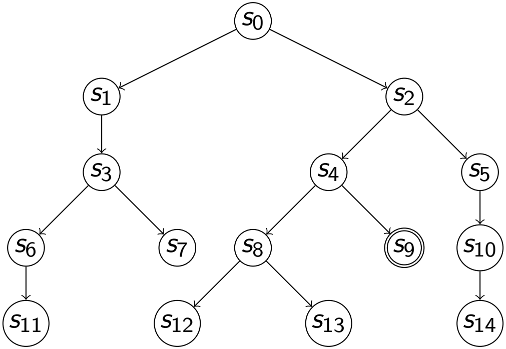
*NTM*s are different because of the transition function.

In deterministic *TMs*, the transition goes from one machine setup to another.

In *NTM*s, the transition function goes from one to many setups.

These setups are run simultaneously, and the machine accepts if one setup halts in an accepting state, or rejects if all setups halt in the rejecting state.

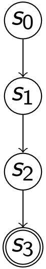
Computation Tree



Each vertex is the setup (current state, tape contents and tape head position) for an *NTM* running on an input. The edges are transitions from one setup to another.

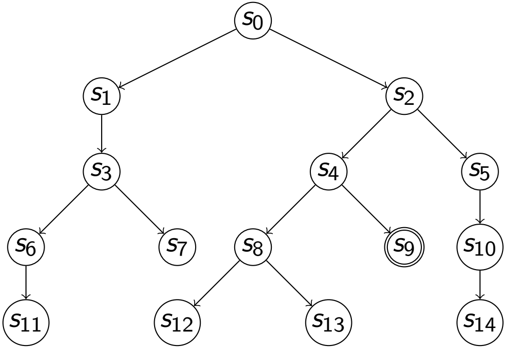
Power of Nondeterminism

Any TM is by definition an NTM. The computation tree for a TM look like this:



Question: Is there any problem that can be solved by an NTM that cannot be solved by a TM?

Using *TMs* to simulate *NTMs*



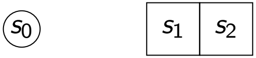
We can use Breadth-First Search[2] to search the tree until we find an accepting state.

[2] https://en.wikipedia.org/wiki/Breadth-first_search

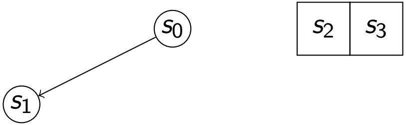
Using *TMs* to simulate *NTMs*

$$\boxed{s_0}$$

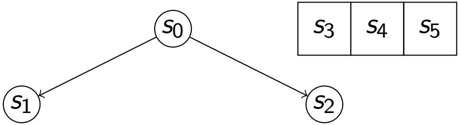
Using *TMs* to simulate *NTMs*



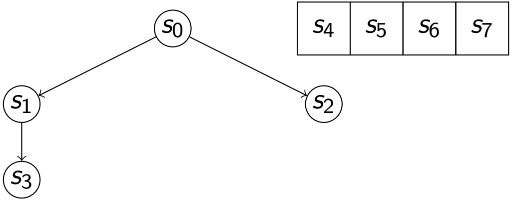
Using *TMs* to simulate *NTMs*



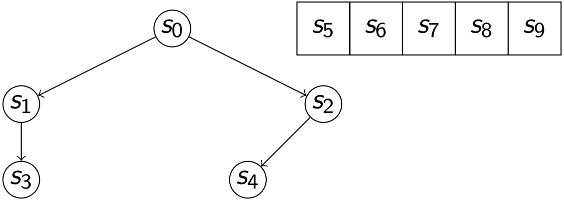
Using *TMs* to simulate *NTMs*



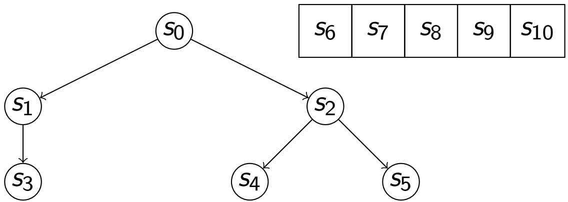
Using *TMs* to simulate *NTMs*



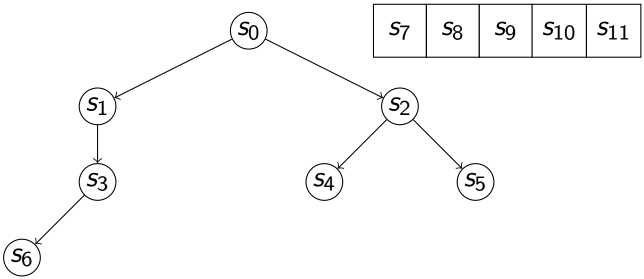
Using *TMs* to simulate *NTMs*



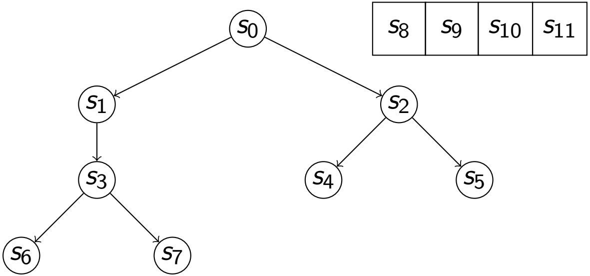
Using TMs to simulate NTMs



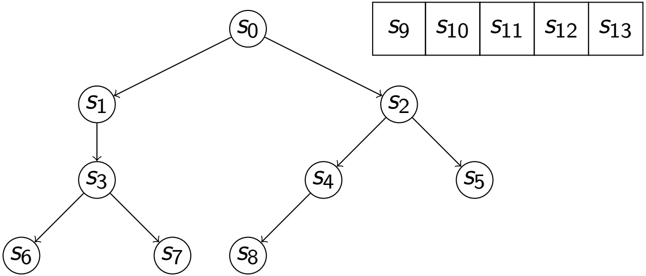
Using *TMs* to simulate *NTMs*



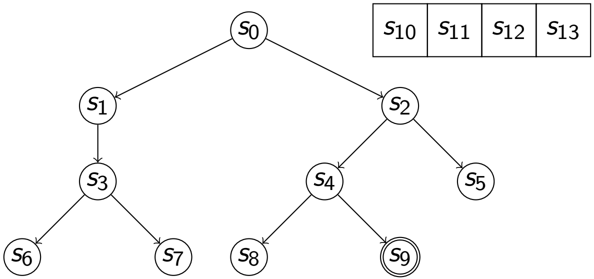
Using *TMs* to simulate *NTMs*



Using *TM*s to simulate *NTM*s



Using *TMs* to simulate *NTMs*



Structure of part one

- What is a computer? *Deterministic Turing Machine, Nondeterministic Turing Machine*
- What is a problem?
- When is a problem easy?

Describing problems as languages

A *language* L is a (potentially infinite) set.

Example languages:

- Text strings that contain the word "Hello"
- Satisfiable boolean expressions
- Eulerian graphs
- Hamiltonian graphs
- (M, w) such that $M(w)$ halts

Decidable languages

A language L is *decidable* if there exists a machine M such that:

- $\forall w \in L, M(w)$ accepts
- $\forall w \notin L, M(w)$ rejects

Verifiable languages

A language L is *verifiable* if there exists a machine V such that:

- $\forall w \in L, \exists c \in \Sigma^*$ s.t. $V(w, c)$ accepts
- $\forall w \notin L, \forall c \in \Sigma^*, V(w, c)$ rejects

Structure of part one

- What is a computer? *Deterministic Turing Machine, Nondeterministic Turing Machine*
- What is a problem? *Deciding if a word is in a language, verifying that a word is in a language*
- When is a problem easy?

Performance of a machine

Problem: We want to talk about how much time it takes for a machine to solve a problem.

Solution: Assume δ takes a constant amount of time to run and count the number of times we call δ.

To remain general, we will focus on how the number of times we call δ scales in the worst case as the input becomes larger.

We can represent this time complexity as a function of the size of the input.

Problem: Time complexity might be complicated to work out

```
while ((1 << (i + 1)) < m_max) {
    matcher->rows[i].row_size = 1 << i;
    lookup_size = 0;
    for (j = 0; j < num_patterns; j++) {
        if ((periods[j] > num_patterns) && (m[j] > num_patterns << 1) && (m[j] - num_patterns > matcher->rows[i].row_size <
            set_fingerprint(matcher->printer, &P[j][matcher->rows[i].row_size], matcher->rows[i].row_size, matcher->tmp);
            fingerprint_concat(matcher->printer, old_patterns[prev_row[j]], matcher->tmp, patterns[lookup_size]);
            prev_row[j] = lookup_size;
            if (m[j] - num_patterns <= (matcher->rows[i].row_size << 2)) {
                end_pattern[lookup_size] = num_progressions;
                set_fingerprint(matcher->printer, &P[j][matcher->rows[i].row_size << 1], m[j] - num_patterns - (matcher->ro
                fingerprint_concat(matcher->printer, patterns[lookup_size], matcher->tmp, prefix[num_prefixes]);
                progression_index[num_prefixes] = num_progressions;
                prefix_length[num_prefixes] = m[j] - num_patterns;
                num_suffixes[num_prefixes] = 1;
                set_fingerprint(matcher->printer, &P[j][m[j] - num_patterns], num_patterns, suffixes[num_prefixes][0]);
                for (k = 0; k < num_prefixes; k++) {
                    if (fingerprint_equals(prefix[k], prefix[num_prefixes])) {
                        end_pattern[lookup_size] = progression_index[k];
                        for (l = 0; l < num_suffixes[k]; l++) {
                            if (fingerprint_equals(suffixes[k][l], suffixes[num_prefixes][0])) {
                                break;
                            }
                        }
                        if (l == num_suffixes[k]) {
                            fingerprint_assign(suffixes[num_prefixes][0], suffixes[k][l]);
                            num_suffixes[k]++;
                        }
                        break;
                    }
                }
                if (k == num_prefixes) num_prefixes++;
            } else {
                end_pattern[lookup_size] = -1;
            }
            for (k = 0; k < lookup_size; k++) {
                if (fingerprint_equals(patterns[k], patterns[lookup_size])) {
                    prev_row[j] = k;
                    if ((end_pattern[k] == -1) && (end_pattern[lookup_size] != -1)) {
                        end_pattern[k] = end_pattern[lookup_size];
                        num_progressions++;
                    }
                    else if (end_pattern[lookup_size] != -1) progression_index[num_prefixes - 1] = end_pattern[k];
                    break;
                }
            }
            if (k == lookup_size) {
                lookup_size++;
                if (end_pattern[lookup_size -1] != -1) num_progressions++;
```

[3]

[3] https://github.com/djmylt/dict_matching/blob/master/dict_matching.h

Solution: Approximate!

Let f and g be functions over the real numbers. We say that:

$$f(n) \in O(g(n)) \text{ iff } \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, f(n) \leq c \cdot g(n)$$

$$f(n) \in \Omega(g(n)) \text{ iff } \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, f(n) \geq c \cdot g(n)$$

$$f(n) \in \Theta(g(n)) \text{ iff } \exists c_1, c_2 > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Examples

Prove each of the following statements:

$$n \in O(n^2)$$

$$2^n \in \Omega(n^2)$$

$$n \in \Theta(2n)$$

Examples

Prove each of the following statements:

$$n \in O(n^2) : c = 1, n_0 = 0$$

$$2^n \in \Omega(n^2)$$

$$n \in \Theta(2n)$$

Examples

Prove each of the following statements:

$$n \in O(n^2) : c = 1, n_0 = 0$$
$$2^n \in \Omega(n^2) : c = 1, n_0 = 1$$
$$n \in \Theta(2n)$$

Examples

Prove each of the following statements:

$$n \in O(n^2) : c = 1, n_0 = 0$$

$$2^n \in \Omega(n^2) : c = 1, n_0 = 1$$

$$n \in \Theta(2n) : c_1 = 0.5, c_2 = 1, n_0 = 0$$

Time Complexity Classes

Let $t : \mathbb{N} \to \mathbb{R}$ be a function.

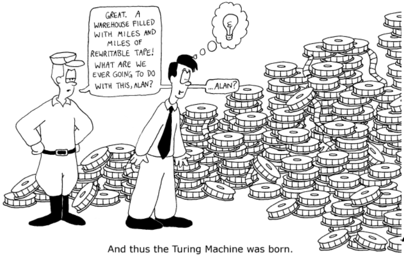
$\mathrm{TIME}(t(n))$ is all the languages that can be decided by a TM in $O(t(n))$ time.

$\mathrm{NTIME}(t(n))$ is all the languages that can be decided by a NTM in $O(t(n))$ time.

Summary of part one

- What is a computer? *Deterministic Turing Machine, Nondeterministic Turing Machine*
- What is a problem? *Deciding if a word is in a language, verifying that a word is in a language*
- How do we measure difficulty? *Upper bound of time for an input of length* n

End of part one



And thus the Turing Machine was born. [4]

[4] http://www.cs.utah.edu/~draperg/cartoons/2005/turing.html

Structure of part two

- Putting it all together!
- ...only to get another (very difficult) problem.
- How might we try to solve this new problem?

Complexity Classes

Now that we have provided our definitions for a computer, a problem and performance, we can look at what problems can be solved under these restrictions.

These are called *complexity classes*.

Deterministic polynomial time

One example of a complexity class is the set of languages that can be decided by a *TM* in polynomial time.

$$\bigcup_{k=0}^{\text{inf}} \mathrm{TIME}(n^k)$$

We shall refer to the set of these problems as P.

Nondeterministic polynomial time

One example of a complexity class is the set of languages that can be decided by a *NTM* in polynomial time.

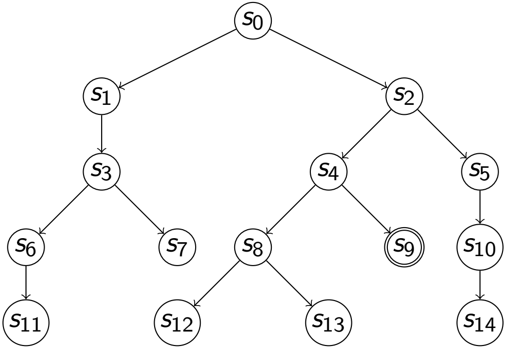
$$\bigcup_{k=0}^{\text{inf}} \mathrm{NTIME}(n^k)$$

We shall refer to the set of these problems as *NP*.

We can also define *NP* as the set of languages that can be verified by a *TM* in polynomial time.

From nondeterminism to verification

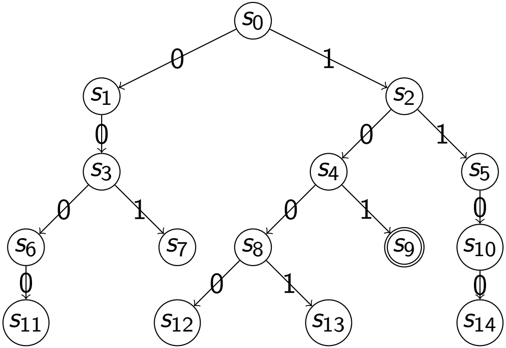
Recall the computation tree for an NTM.



We label each branch with some integer.

From nondeterminism to verification

Recall the computation tree for an *NTM*.



We label each branch with some integer.

From verification to nondeterminism

Since the verifier runs in polynomial time, the certificate must be polynomial in length.

We use nondeterminism to brute force the certificate.

And then run the verifier on every certificate.

Structure of part two

- Putting it all together! *Complexity classes, P, NP*
- ...only to get another (very difficult) problem.
- How might we try to solve this new problem?

Exercise Left For the Student

Does $P = NP$?

The P versus NP problem

Arguably first proposed by Gödel in a letter to von Neumann in 1956.[5]

First stated formally by Cook in 1971.

Solving the problem will earn you a million dollars, courtesy of the Clay Mathematics Institute.[6]

[5] https://ecommons.cornell.edu/bitstream/handle/1813/6910/89-994.pdf

[6] http://www.claymath.org/millennium-problems/p-vs-np-problem

The easy side: $P \subseteq NP$

Recall that any TM is by definition nondeterministic.

Likewise, any polynomial-time TM is also nondeterministic.

Hence $P \subseteq NP$.

The easy side: $P \subseteq NP$

Alternative proof (using verification):

Let $TM\ M$ decide L in polynomial time. Define V as follows:

$V(w, c)$:
if $M(w)$ *accepts* **then**
  accept
end
else
  reject
end

V verifies L in polynomial time. Hence $P \subseteq NP$.

The harder side: Is $NP \subseteq P$

Another way to think of this problem: *If a problem can be easily verified, can it be easily solved?*

How might we answer this question?

Why not look at the hardest problems in NP?

If $P = NP$, then even the hardest problems in NP will be solvable in polynomial time.

And if $P \subset NP$, then these are the problems that won't have a polynomial time solution, as could be checked by lower-bound analysis.

But how can we determine the hardest problems in NP?

Summary of part two

- Putting it all together! *Complexity classes, P, NP*
- ...only to get another (very difficult) problem. *Are easy to verify problems easy to solve?*
- How might we try to solve this new problem?

Polynomial time reducible

Take two languages A and B. We say that $A \leq_p B$ iff $\exists TM\ M$ such that:

- $\forall w \in A, M(w) \in B$
- $\forall w \notin A, M(w) \notin B$
- and M runs in worst-case $O(p(n))$ time for some polynomial p, where $n = |w|$

Note that this property is transitive: $A \leq_p B$ and $B \leq_q C \rightarrow A \leq_{p+q} C$

NP-Complete

A language B is NP-Complete iff:

- $B \in NP$
- and $\forall A \in NP, A \leq_p B$

If one NP-Complete language is proven to be in P, then every NP problem is also in P.

Problem: There are lots of NP problems

We know that any problem in NP can be decided in polynomial time by an NTM.

So can we convert an NTM to some other problem?

Satisfiable boolean formulae

Take a boolean formula $f = a \vee b \wedge \bar{c}$.

We say that f is *satisfiable* if we can assign 0 or 1 to each variable such that $f = 1$.

Examples:

- f is satisfiable: $(a = 1, b = 1, c = 0)$
- but $f' = x \wedge \bar{x}$ is not satisfiable

We call *SAT* the language of all satisfiable boolean formulae.

$SAT \in NP$

SAT can be verified in polynomial time:

- Make the certificate the values we assign to each variable.
- Have V substitute the value for each variable into the formula.
- Accept if the formula evaluates to 1, otherwise reject.

Cook-Levin Theorem

Cook showed that any NTM M that halts in polynomial time can be converted to a polynomial length boolean formula such that:

- If M accepts, then the formula is satisfiable
- and if M rejects, then the formula cannot be satisfied.

As a result, it was proven that $SAT \in NP$-Complete.

Proving other NP-Complete problems

Now that we have one NP-Complete problem, proving others is a lot easier.

Recall that polynomial-time reducibility is transitive.

So we can now provide a recursive definition. $B \in NP$-Complete iff:

- $B = SAT$
- or $B \in NP$ and $\exists A \in NP$-Complete s.t. $A \leq_p B$

Other *NP*-Complete problems

- *3SAT*

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs

Other NP-Complete problems

- $3SAT$
- Hamiltonian graphs
- 0-1 Integer Programming

Other NP-Complete problems

- $3SAT$
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack

Other *NP*-Complete problems

- *3SAT*
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack
- Tetris

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack
- Tetris
- Lemmings

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack
- Tetris
- Lemmings
- Super Mario Bros

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack
- Tetris
- Lemmings
- Super Mario Bros
- Bejeweled

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack
- Tetris
- Lemmings
- Super Mario Bros
- Bejeweled
- Candy Crush Saga

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs
- 0-1 Integer Programming
- Knapsack
- Tetris
- Lemmings
- Super Mario Bros
- Bejeweled
- Candy Crush Saga
- Flood-It

Summary of part two

- Putting it all together! *Complexity classes, P, NP*
- ...only to get another (very difficult) problem. *Are easy to verify problems easy to solve?*
- How might we try to solve this new problem? *NP-Complete problems*

What else is there?

Recall our three questions from part one:

- What is a computer?
- What is a problem?
- When is a problem easy?

What if we answered these differently?

When is a problem easy?

Exponential time: EXP

Linear time: LIN

Space complexity: $PSPACE$, $EXPSPACE$

Sublinear working space: L

What is a problem?

Computational problems: NP-Hard

Counting problems: $\#P$

Complementary problems: co-NP

What is a computer?

Probabilistic Turing Machines: BPP, RP

Postselection: Post-BPP

Parallel Computing: NC

Talking to another, more powerful computer: MA, IP

Oracles: $\Delta_i P, \Pi_i P, \Sigma_i P, PH$

Advice: $P/poly$, P/log

Quantum computers: EQP, BQP

Time travel: P_{CTC}

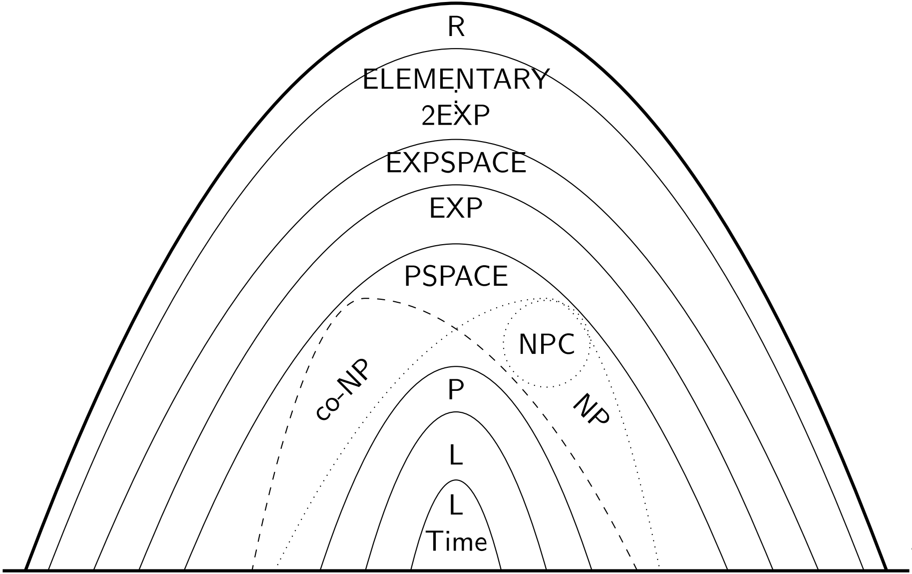
This is only the beginning

There are many more complexity classes out there, and very quickly relating them in a simple equation like this:

$$P \subseteq NP$$

Becomes this:

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE = IP = P_{CTC} \subseteq EXP \subseteq EXPSPACE$$



[7]

[7] http://www.texample.net/tikz/examples/complexity-classes/

Open problems in complexity theory

- Does $P = NP$?

Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?

Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?
- Does $BPP = P$?

Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?
- Does $BPP = P$?
- Does $BQP = BPP$?

Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?
- Does $BPP = P$?
- Does $BQP = BPP$?
- Does $NC = P$?

Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?
- Does $BPP = P$?
- Does $BQP = BPP$?
- Does $NC = P$?
- Does $P = PSPACE$?

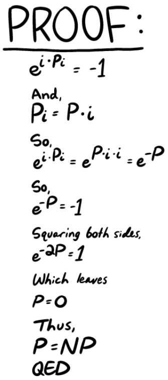
Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?
- Does $BPP = P$?
- Does $BQP = BPP$?
- Does $NC = P$?
- Does $P = PSPACE$?
- Does $NP = \text{co-}NP$?

Open problems in complexity theory

- Does $P = NP$?
- Does $L = NL$?
- Does $BPP = P$?
- Does $BQP = BPP$?
- Does $NC = P$?
- Does $P = PSPACE$?
- Does $NP =$ co-NP?
- And lots more...

The end



[8]

[8] http://www.smbc-comics.com/?id=3919

The end

[9]

[9] http://www.smbc-comics.com/?id=3919

Suggested books

- *Quantum Computing Since Democritus* by Scott Aaronson offers a broad overview of many complexity classes
- *Introduction to the Theory of Computation* by Michael Sipser is the recommended textbook for most computability and complexity theory courses

Suggested papers

- *On Computable Numbers, with an Application to the Entscheidungsproblem* by Alan Turing is Turing's PhD thesis, which provides the original definition of Turing Machines, a formal definition of the Universal Turing Machine, and a proof that the Halting problem is undecidable
- *The Complexity of Theorem Proving Procedures* by Stephen Cook provides the proof that $SAT \in NP$-Complete
- *Reducibility Among Combinatorial Problems* by Richard Karp provides 21 of the earliest problems proven to be *NP*-Complete

Suggested websites

- https://complexityzoo.uwaterloo.ca/ is a Wiki originally developed by Scott Aaronson, which provides a list of every complexity class ever stated