Complexity Theory

Dominic Moylett

Quantum Engineering CDT
University of Bristol

dominic.moylett@bristol.ac.uk

December 11, 2015

Summary of Presentation

This presentation will consist of two parts:

- Part one will look at the building blocks of complexity classes
- And part two will look at the consequences of the complexity classes we defined

We'll finish by exploring other aspects of complexity theory, and giving examples of other complexity classes and open problems out there.

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.

How do we measure performance?

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?

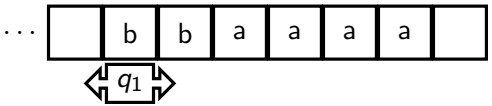
Complexity theory is the study of what problems are easy to solve with a computer.

What is a computer?

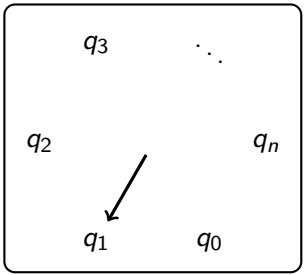
Structure of part one

- What is a computer?
- What is a problem?
- How do we measure performance?

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 $(Q, \Sigma, \Gamma, q_0, q_{accept}, q_{reject}, \delta)$:

- Q is the set of all possible states

Formal definition of a computer

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

- Q is the set of all possible states
- Σ is the input alphabet for the tape. Note that Σ cannot contain the blank symbol $\sqcup$

Formal definition of a computer

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

- 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 $\sqcup \in \Gamma$

Formal definition of a computer

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

- 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

Formal definition of a computer

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

- 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

Formal definition of a computer

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

- 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 $q_{reject} \neq q_{accept}$

Formal definition of a computer

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

- 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 $q_{reject} \neq q_{accept}$
- $\delta : Q \times \Gamma \to Q \times \Sigma \times \{L, R\}$ 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 $q \notin \{q_{accept}, q_{reject}\}$ **do**
  $(q, T_h, i) = \delta(q, T_h)$
  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 M halts in the accept state, then M accepted the input.

If M halts in the reject state, then M rejected the input.

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.

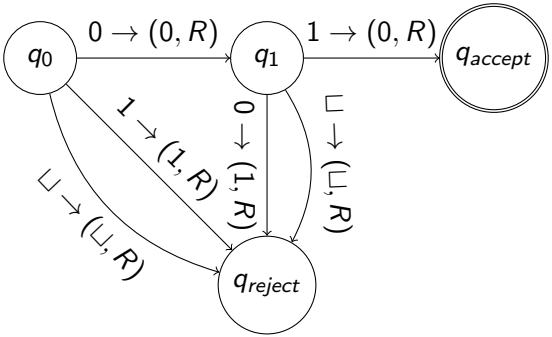
Not only that, but we could manipulate this integer such that it matched performing the next step of the computation.

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

The Church-Turing Thesis

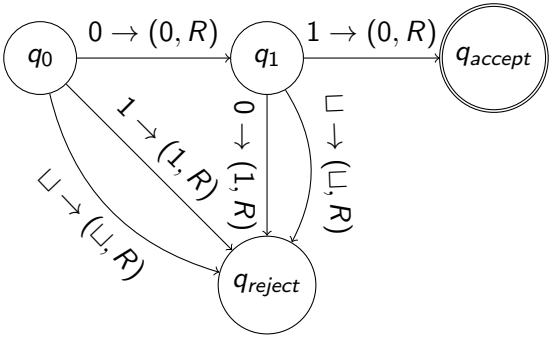
"[A]ll effectively calculable sequences are computable", *Turing*

Quiz Time!



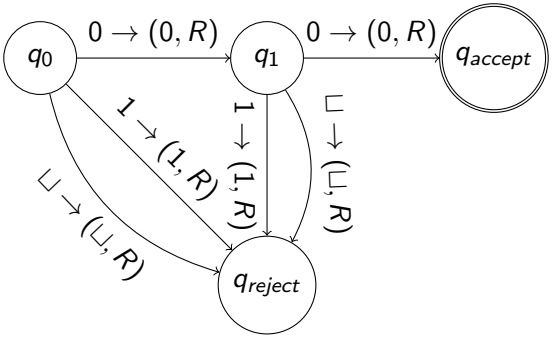
Question: Does M accept or reject 01?

Quiz Time!



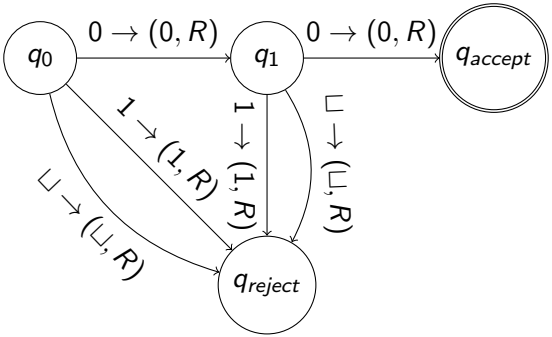
Answer: M accepts 01!

Quiz Time!



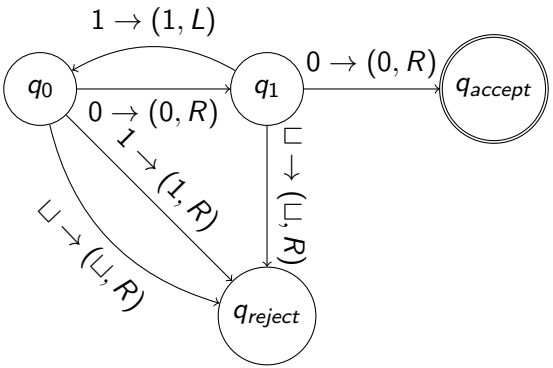
Question: Does M accept or reject 01?

Quiz Time!



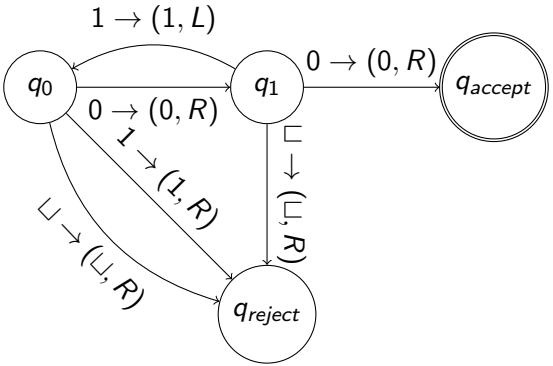
Answer: M rejects 01!

Quiz Time!



Question: Does M accept or reject 01?

Quiz Time!



Answer: M doesn't halt on 01.

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 *TM* M and $w \in \Sigma_M^*$:

- H halts in the accept state if M halts on w and
- H halts in the reject state if M does not halt on w.

The halting problem

Assume H exists. Let's build another machine H' which starts with a machine M on its tape.

H' is described as follows:
if H *accepts* (M, M) **then**
| go into an infinite loop
else
| accept
end

What happens if we run H' on H'?

The halting problem

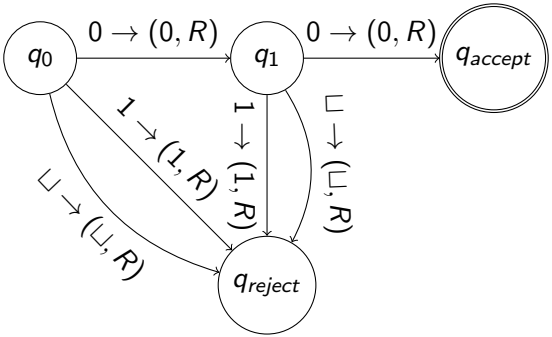
H cannot determine whether or not H' halts on H', so it is impossible to define a machine that halts on every input.

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.

The reject state

The reject state is often defined implicitly for convenience.

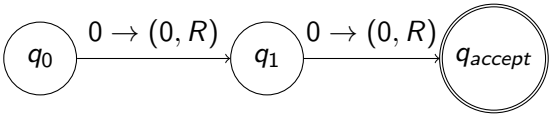
For any state except the accept state that doesn't have transitions for every symbol in the tape alphabet, it is assumed that there is a transition to the reject state.



The reject state

The reject state is often defined implicitly for convenience.

For any state except the accept state that doesn't have transitions for every symbol in the tape alphabet, it is assumed that there is a transition to the reject state.



Nondeterminism: Spot the difference!

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

- Q is the set of all possible states

Nondeterminism: Spot the difference!

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

- Q is the set of all possible states
- Σ is the input alphabet for the tape. Note that Σ cannot contain the blank symbol $\sqcup$

Nondeterminism: Spot the difference!

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

- 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 $\sqcup \in \Gamma$

Nondeterminism: Spot the difference!

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

- 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

Nondeterminism: Spot the difference!

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

- 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

Nondeterminism: Spot the difference!

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

- 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 $q_{reject} \neq q_{accept}$

Nondeterminism: Spot the difference!

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

- 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 $q_{reject} \neq q_{accept}$
- $\delta : Q \times \Gamma \to (Q \times \Sigma \times \{L, R\})^+$ is the transition function

What, what's the difference?

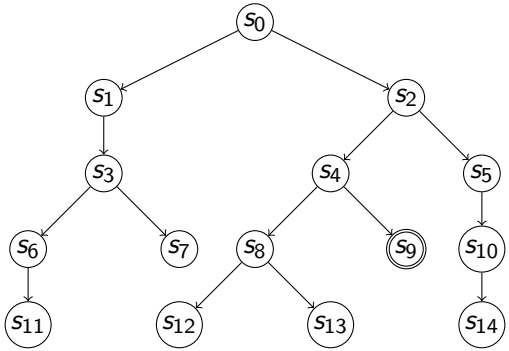
*NTM*s are different because of the transition function.

In deterministic *TM*s, 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.

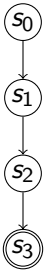
Computation Tree



Vertices are single setups (current state, tape contents and head position) for an *NTM* running on an input. Edges are transitions.

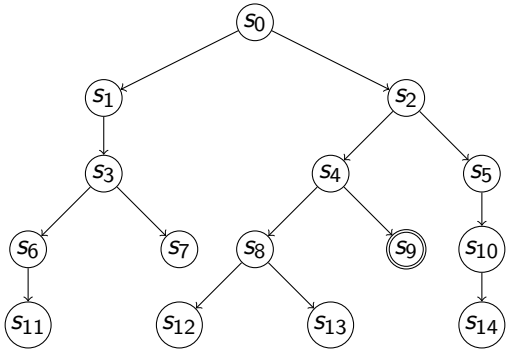
Power of Nondeterminism

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



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

Using *TM*s to simulate *NTM*s



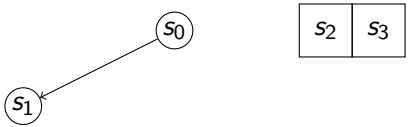
We can use Breadth-First Search[2] to search the computation tree until we find an accept state.

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

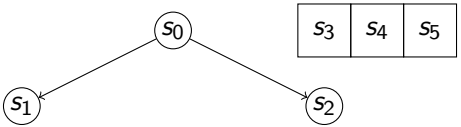
Using *TM*s to simulate *NTM*s

Using *TM*s to simulate *NTM*s

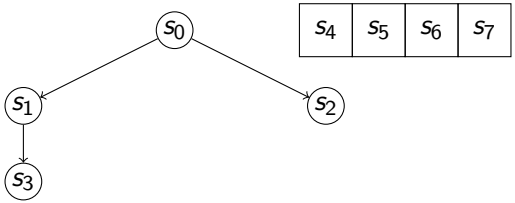
Using *TM*s to simulate *NTM*s



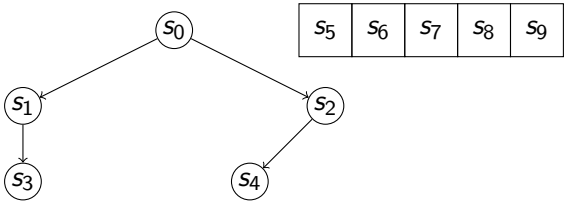
Using *TM*s to simulate *NTM*s



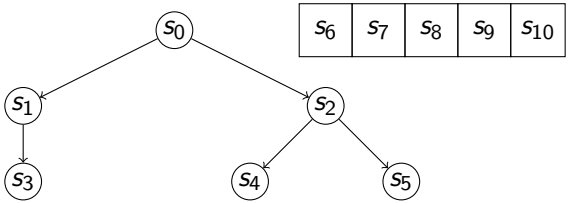
Using *TM*s to simulate *NTM*s



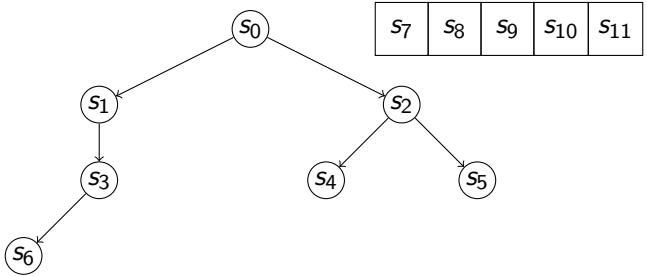
Using *TM*s to simulate *NTM*s



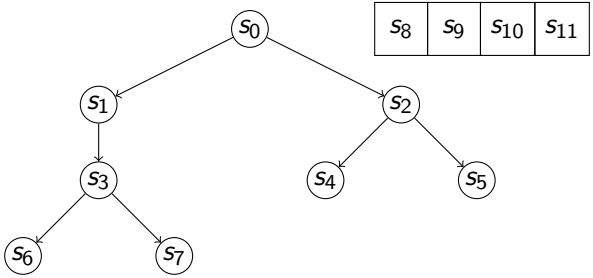
Using *TM*s to simulate *NTM*s



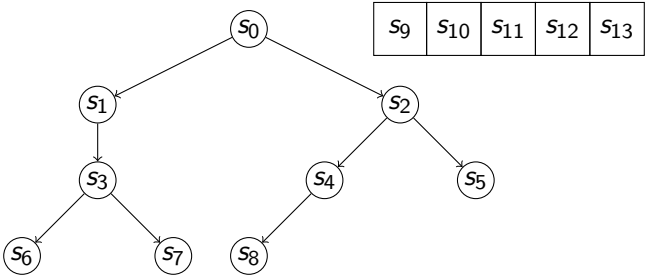
Using *TM*s to simulate *NTM*s



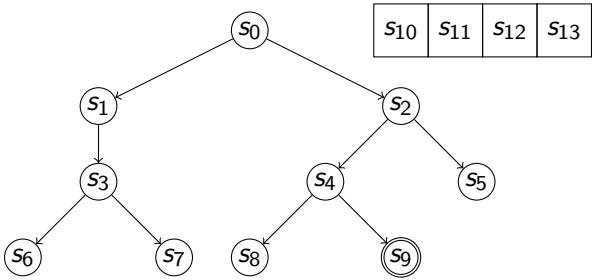
Using *TM*s to simulate *NTM*s



Using *TM*s to simulate *NTM*s



Using *TM*s to simulate *NTM*s



Structure of part one

- What is a computer? *Deterministic Turing Machine, Nondeterministic Turing Machine*
- What is a problem?
- How do we measure performance?

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 halts when given input w

Decidable languages

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

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

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

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*
- How do we measure performance?

Performance of a machine

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

To do this, we'll 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 << 1)) {
            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->rows[i].
                    row_size << 1), matcher->tmp);
                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;
            }
```

[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:

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

We use this notation, called *Big-O*, *Big-Omega* and *Big-Theta*, to simplify our complexity functions by rounding up for upper bounds (Big-O) and down for lower bounds (Big-Omega).

Solution: Approximate!

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

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

We use this notation, called *Big-O*, *Big-Omega* and *Big-Theta*, to simplify our complexity functions by rounding up for upper bounds (Big-O) and down for lower bounds (Big-Omega).

Solution: Approximate!

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

- $f(n) \in O(g(n))$ iff $\exists c > 0, n_0 \geq 0$ s.t. $\forall n \geq n_0, f(n) \leq c \cdot g(n)$
- $f(n) \in \Omega(g(n))$ iff $\exists c > 0, n_0 \geq 0$ s.t. $\forall n \geq n_0, f(n) \geq c \cdot g(n)$
- $f(n) \in \Theta(g(n))$ iff $\exists c_1, c_2 > 0, n_0 \geq 0$ s.t. $\forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

We use this notation, called *Big-O*, *Big-Omega* and *Big-Theta*, to simplify our complexity functions by rounding up for upper bounds (Big-O) and down for lower bounds (Big-Omega).

Examples

We can use the definitions from before to prove each of the following statements:

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

Examples

We can use the definitions from before to 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

We can use the definitions from before to prove each of the following statements:

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

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

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

Examples

We can use the definitions from before to prove each of the following statements:

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

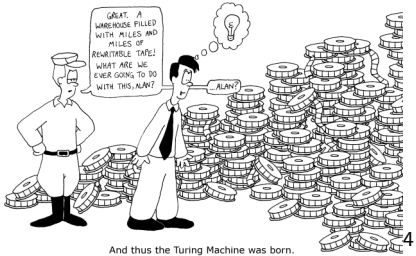
$$2^n \in \Omega(n^2) : c = 1, n_0 = 0$$

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

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 performance? *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*.

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.

Deterministic polynomial time

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

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

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

Nondeterministic polynomial time

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

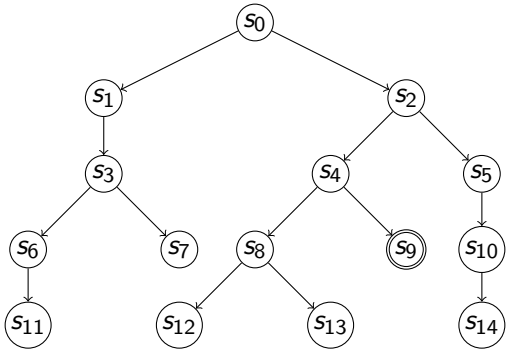
$$\bigcup_{k=0}^{\infty} \mathrm{NTIME}(n^k)$$

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

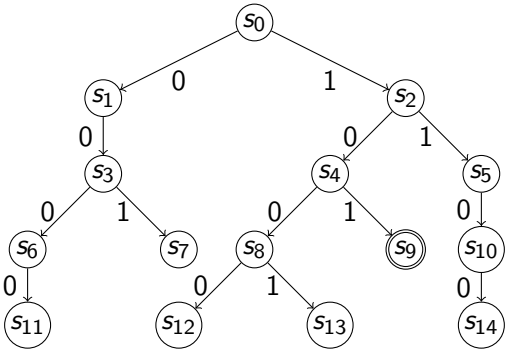
Recall the computation tree for an NTM.



We label each branch with an integer such that the labels are unique between siblings.

From nondeterminism to verification

Recall the computation tree for an NTM.



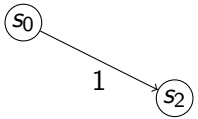
We label each branch with an integer such that the labels are unique between siblings.

From nondeterminism to verification

Our certificate c is now the polynomial length sequence that leads to the accepting state. In this case $c = 101$, as can be verified in polynomial time by following the certificate down the tree:

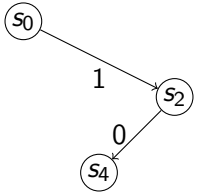
From nondeterminism to verification

Our certificate c is now the polynomial length sequence that leads to the accepting state. In this case $c = 101$, as can be verified in polynomial time by following the certificate down the tree:



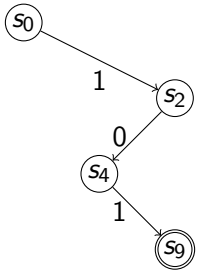
From nondeterminism to verification

Our certificate c is now the polynomial length sequence that leads to the accepting state. In this case $c = 101$, as can be verified in polynomial time by following the certificate down the tree:



From nondeterminism to verification

Our certificate c is now the polynomial length sequence that leads to the accepting state. In this case $c = 101$, as can be verified in polynomial time by following the certificate down the tree:



From verification to nondeterminism

Since the verifier runs in polynomial time, at most a polynomial number of characters can be read from the certificate.

We use nondeterminism to brute force the certificate in polynomial time.

And then we run the verifier on every certificate in polynomial time.

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]

Became known as P versus NP after Cook's work 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 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?*

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?

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?

Polynomial time computable functions

A function $f : \Sigma^* \to \Sigma^*$ is polynomial time computable if there exists a polynomial time Turing machine M that, when started with input w on its tape, halts with only $f(w)$ on its tape.

Polynomial time reducible

Take two languages A and B. We say that $A \leq_P B$ if there exists a polynomial time computable function f such that:

- $\forall w \in A, f(w) \in B$
- $\forall w \notin A, f(w) \notin B$

This means that if we have a polynomial time solution for B, then we also have a polynomial time solution for A.

Note that this property is transitive: If $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$

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, so P would be equal to NP.

Problem: There are lots of *NP* problems

There are an infinite number of *NP* problems, so it is impossible to provide a reduction to each individual problem.

So how can we reduce all of them to a specific problem?

Satisfiable boolean formulae

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

We say that f is *satisfiable* (also known as a tautology) if we can assign $\bot$ or $\top$ to each variable such that $f = \top$.

Examples:

- f is satisfiable: $(a = \top, b = \top, c = \bot)$
- 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.

$SAT \in NP$

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.

$SAT \in NP$

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 $\top$, otherwise reject.

Cook-Levin Theorem

Recall that any language in NP can be decided by an NTM in polynomial time.

Cook showed that any NTM M that halts in polynomial time running on input w can be converted in polynomial time to a boolean formula such that:

- If M accepts w, then the formula is satisfiable
- and if M rejects w, 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 now if our problem B is in *NP* and we can find one problem $A \in$ *NP*-Complete such that $A \leq_P B$, then $B \in$ *NP*-Complete

Other *NP*-Complete problems

- $3SAT$

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs
- Sudokus

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- Sudokus
- Knapsack

Other *NP*-Complete problems

- $3SAT$
- Hamiltonian graphs
- Sudokus
- Knapsack
- Tetris

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- Sudokus
- Knapsack
- Tetris
- Super Mario Brothers

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- Sudokus
- Knapsack
- Tetris
- Super Mario Brothers
- Lemmings

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- Sudokus
- Knapsack
- Tetris
- Super Mario Brothers
- Lemmings
- Bejeweled

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- Sudokus
- Knapsack
- Tetris
- Super Mario Brothers
- Lemmings
- Bejeweled
- Candy Crush Saga

Other *NP*-Complete problems

- 3*SAT*
- Hamiltonian graphs
- Sudokus
- Knapsack
- Tetris
- Super Mario Brothers
- Lemmings
- 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?
- How do we measure performance?

What if we answered these differently?

Different restrictions on time

Exponential time: EXP

Linear time: LIN

How do we measure performance?

Space complexity: $PSPACE$, $EXPSPACE$

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

Parallel Computing: NC

Quantum computers: EQP, BQP

Time travel: P_{CTC}

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 $NP = \text{co-}NP$?
- And lots more...

The Conclusion

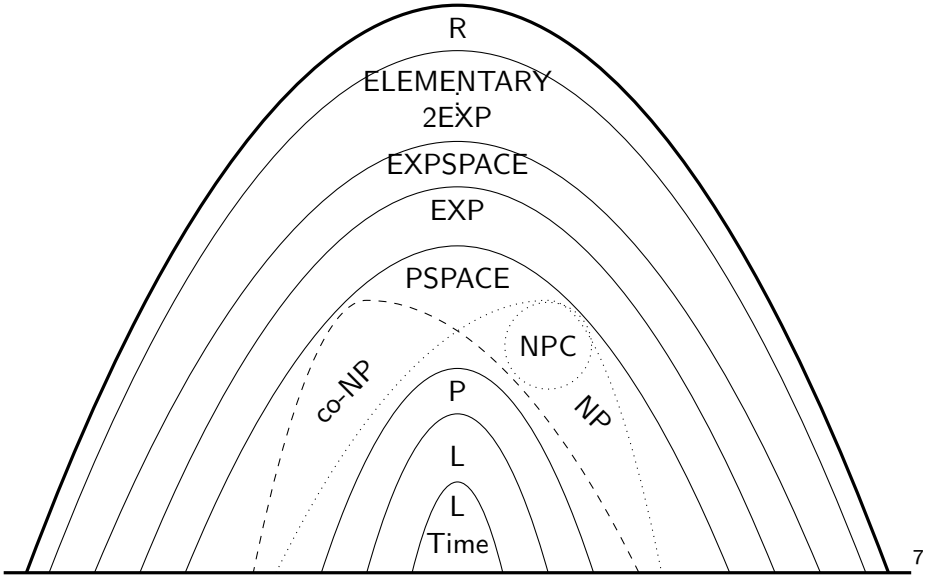
Complexity theory is a large area about what problems are solvable under different models of a computer and different performance requirements.

There are lots of complexity classes out there – far more than have been mentioned in this presentation – and very quickly relating them in a simple equation like this:

$$P \subseteq NP$$

Becomes this:

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



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

The end

PROOF:

$e^{i \cdot P_i} = -1$

And,

$P_i = P \cdot i$

So,

$e^{i \cdot P_i} = e^{P \cdot i \cdot i} = e^{-P}$

So,

$e^{-P} = -1$

Squaring both sides,

$e^{-2P} = 1$

Which leaves

$P = 0$

Thus,

$P = NP$

QED

8

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

The end

[9]

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

Suggested books

- *Introduction to the Theory of Computation* by Michael Sipser is the recommended textbook for most computability and complexity theory courses
- *Computers and Intractability: A Guide to the Theory of NP-Completeness* by Michael R. Garey and David S. Johnson covers a lot about *NP*-Complete problems, including a very large collection of *NP*-Complete problems.
- *Quantum Computing Since Democritus* by Scott Aaronson offers a broad overview of many complexity classes

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

Proofs for *NP*-Complete problems

- Takayuki Yato and Takahiro Seta: Complexity and Completeness of Finding Another Solution and Its Application to Puzzles, IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, Vol.E86-A, No.5, pp.1052-1060
- Erik D. Demaine, Susan Hohenberger and David Liben-Nowell: Tetris is Hard, Even to Approximate, `http://arxiv.org/abs/cs/0210020`
- Graham Cormode: The Hardness of the Lemmings Game, or Oh no, more NP-Completeness Proofs, Proceedings of the 3rd International Conference on Fun with Algorithms, May 2004, pages 65-76
- Greg Aloupis, Erik D. Demaine, Alan Guo and Giovanni Viglietta: Classic Nintendo Games are (Computationally) Hard, `http://arxiv.org/abs/1203.1895`
- Luciano Gual, Stefano Leucci and Emanuele Natale: Bejeweled, Candy Crush and other Match-Three Games are (NP-)Hard, `http://arxiv.org/abs/1403.5830`
- Raphaël Clifford, Markus Jalsenius, Ashley Montanaro and Benjamin Sach: The Complexity of Flood Filling Games, `http://arxiv.org/abs/1001.4420`