QECDT Topics Presentation: Complexity theory

Dominic Moylett

University of Bristol

dominic.moylett@bristol.ac.uk

November 19, 2015

Complexity theory in a nutshell

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

What is complexity theory?

Complexity theory is the study of how difficult it is to solve a problem with a computer.

What is complexity theory?

Complexity theory is the study of how difficult it is to solve a problem with a computer.

How do we measure difficulty?

What is complexity theory?

Complexity theory is the study of how difficult it is to solve a problem with a computer.

What is a problem?

What is complexity theory?

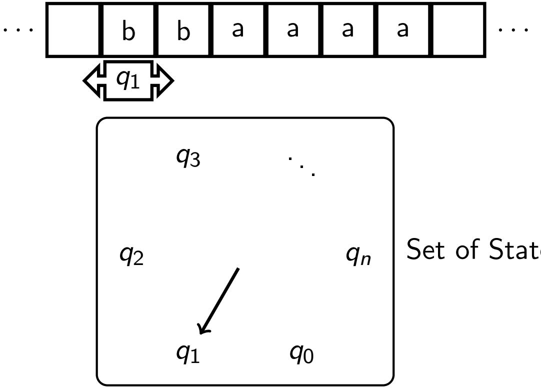
Complexity theory is the study of how difficult it is to solve a problem with a computer.

What is a computer?

Structure of part one

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

Our model of a computer



Machine Tape (RAM) with head at letter b

Set of States (CPU) with current state q_0

[1]

[1] Original version at `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 $\langle Q, \Gamma, b, \Sigma, q_0, F, \delta \rangle$:

- Q is the set of all possible states
- Γ is tape alphabet
- $b \in \Gamma$ is the blank symbol for the tape
- $\Sigma = \Gamma / b$ is the input alphabet for the tape
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states
- $\delta : Q \times \Gamma \to Q \times \Sigma \times \{L, R\}$ 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.

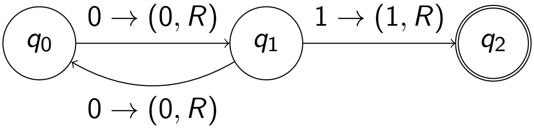
$q = q_0$
$h = 0$
$T = w$ `// Tape starts as just input, followed by blank cells`
while $\delta(q, T_h)$ *is not undefined* **do**
    $(q, h, i) = \delta(q, T_h)$
    if $i = L$ **then**
        $h = h - 1$ `// Move tape head to the left`
    else
        $h = h + 1$ `// Move tape head to the right`
    end
end

What happens when it stops?

When we reach a point that $\delta(q, T_h)$ undefined, the machine has *halted*. What happens next depends on the state the machine stopped in.

if $q \in F$ **then**
  accept
  return T // This is thought of as the output of $M(w)$
else
  reject
end

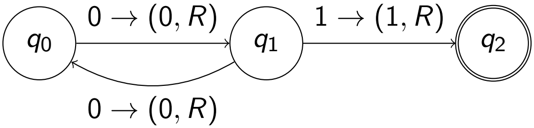
Quiz Time!



q_0 is the start state, and q_2 is the accept state.

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

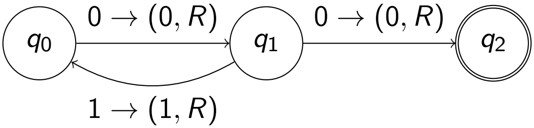
Quiz Time!



q_0 is the start state, and q_2 is the accept state.

Answer: $M(01)$ **accepts**!

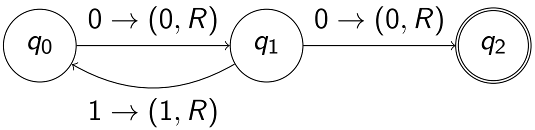
Quiz Time!



q_0 is the start state, and q_2 is the accept state.

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

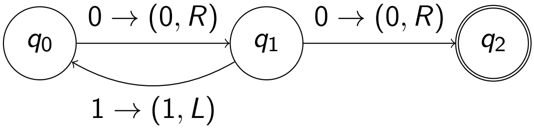
Quiz Time!



q_0 is the start state, and q_2 is the accept state.

Answer: $M(01)$ **rejects**!

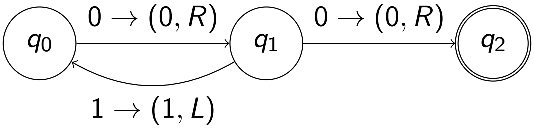
Quiz Time!



q_0 is the start state, and q_2 is the accept state.

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

Quiz Time!



q_0 is the start state, and q_2 is the accept state.

Answer: $M(01)$ **rejects**!

But wait, that last one didn't halt!

Sometimes it is possible for a *TM* to never halt on certain inputs.

In these cases, the machine rejects the input.

A related topic that comes from the idea of never terminating machines is **computability theory**. Computability theory looks at if problems can or cannot be solved by a computer, regardless of difficulty.

For the rest of this presentation, we will focus on machines that terminate on all inputs unless stated.

What can we store in a machine's memory?

- Integers
- Text
- Other machines

Universal Turing Machines

Turing showed in his PhD thesis that we could represent the current state of any TM – including current state, tape 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.

He then designed a Universal Turing Machine; a Turing Machine capable of running any other Turing Machine.

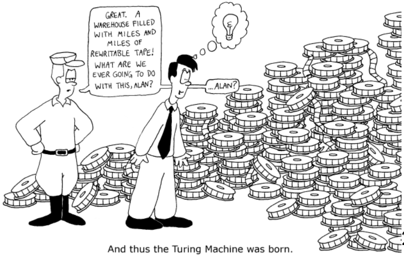
The Church-Turing Thesis

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

Summary of part one

- What is a computer? *Deterministic Turing Machine, Non-Deterministic 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.[2]

[2] 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?

Pop quiz!

Does $P = NP$?

The P versus NP problem

Arguably first proposed by Gödel in a letter to von Neumann in 1956.[3]

First stated formally by Cook in 1971.[4]

Solving the problem will earn you a million dollars, courtesy of the Clay Mathematics Institute.[5]

Aaronson has called it the most important Millennium Problem, as the answer could make the other problems significantly easier or harder to solve.[6]

[3] https://ecommons.cornell.edu/bitstream/handle/1813/6910/89-994.pdf
[4] http://dl.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=805047
[5] http://www.claymath.org/millennium-problems/p-vs-np-problem
[6] http://www.thenakedscientists.com/HTML/interviews/interview/1001376/

The easy side: $P \subseteq NP$

Recall that any TM is by definition non-deterministic.

Likewise, any polynomial-time TM is also non-deterministic.

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! *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 difficulty?

What if we answered these differently?

What is a computer?

Probabilistic Turing Machines: BPP, RP

Parallel Computing: NC

Talking to another, more powerful computer: MA, IP

Quantum computers: EQP, BQP

Time travel: P_{CTC}

What is a problem?

Computational problems: *NP-Hard*

Counting problems: $\#P$

Complementary problems: *co-NP*

How do we measure difficulty?

Exponential time: EXP

Linear time: LIN

Space complexity: $PSPACE$, $EXPSPACE$

Sublinear working space: L

This is only the beginning

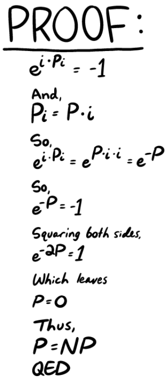
There are many more complexity classes out there, and very quickly relating them in a simple equation like this:

$$P \subseteq NP$$

Becomes this:

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

The end



[7]

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

The end

[8]

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