CoCoNuT - Complexity - Lecture 0

N.P. Smart

Dept of Computer Science
University of Bristol,
Merchant Venturers Building

September 29, 2014

Outline

Resources

There are

- Notes prepared for the course.
- Books, mentioned in the notes.
- Exercises, in the notes.
- Lectures.
- Classes for discussions.

Take notes in lectures.

Lectures are to set the background for your learning.

You learn outside of the lectures, not in them!

Plan

The notes have for each lectures:

- Basic material
  - You should go over this, both before and after each lecture.
  - If you dont understand these bits you have a problem.
- Supplementary notes on other related matters.
  - Trying to read these bits will help you in the long run.
- Advanced Technical Notes for those really keen.
  - Have a go, but do not be disheartened if it gets too much.

In the lecture zero we do a rapid survey of things you should have met before.

Turing Machines

Definition

A Turing machine is described by a tuple $(\Sigma, \mathbb{I}, \Gamma, \delta)$, where:

1. Σ is the set of states,
2. $\mathbb{I}$ is the input alphabet (which must not contain $\Box$ or $\triangleright$), e.g. $\mathbb{I} = \{0, 1\}$.
3. Γ is the tape alphabet, where $\{\triangleright, \Box\} \subset \Gamma$ and $\mathbb{I} \subseteq \Gamma$,
4. $\delta : \Sigma \times \Gamma \to \Sigma \times \Gamma \times \{L, R\}$ is the transition function,
5. START $\in \Sigma$ is the start state,
6. ACCEPT $\in \Sigma$ is the accept state,
7. REJECT $\in \Sigma$ is the reject state, where REJECT $\neq$ ACCEPT.
8. $\triangleright$ is a special tape symbol marking the left most end of the tape.

Turing Machines

The Church-Turing thesis states that everything which can be computed can be computed by a Turing machine.

Some evidence for this:

- Turing machines are equivalent to other models of computation such as
  - multitape Turing machines,
  - pushdown automata with two stacks
  - counter machines.

There are universal Turing machines which can simulate any other Turing machine given as input.

Languages

A language is a set of strings.

We use a Turing machine to decide whether a string x is in the language.

Example

PRIMES is the language (i.e. set of binary strings) representing integers such that the integer is prime.

A Turing Machine which “accepts” PRIMES is something which can perform a primality testing algorithm.

Computability Theory

Computability Theory refers to the problem of working out what can be computed at all.

It turns out some things cannot be computed at all

i.e. There are some languages which cannot be decided by a Turing Machine

- e.g. The Halting Problem

This has wide philosophical implications in science, maths and CS.

- The Hilbert Programme.
- Gödel's Incompleteness Results
- ....

Introduction to Complexity Theory

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.

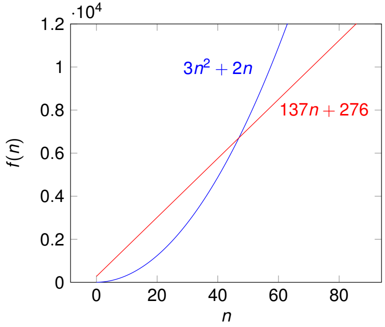
Some notes on this definition:

- 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.

Big-O notation

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 and little-o notation

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

Definition: Big-O

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 $n \geq n_0$,

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

i.e.

$$\exists c > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

Big-O and little-o notation

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

Definition: little-o

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

$$f(n) < c \cdot g(n).$$

i.e.

$$\forall c > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) < c \cdot g(n).$$

or, equivalently,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Examples: Polynomials

If f is a polynomial of degree k then

$f(n) = O(n^k)$
$f(n) = o(n^{k+1})$, but f is not $o(n^k)$

In general: if $0 < k_1 < k_2$ then $n^{k_1} = o(n^{k_2})$

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 $O(n^2)$.

Examples: Logarithms and Exponentials

Logarithms.

Recall: $a^x = y$ then $x = \log_a y$

- Typically $a = 2$: $(\lfloor \log_2 n \rfloor + 1)$ is n's length in binary.

$$\frac{(\log n)^k}{n^c} \underset{n\to\infty}{\longrightarrow} 0 \quad \text{for all } k, c > 0$$

Thus $\log n = o(n^k)$, for all $k > 0$, $n \log n = o(n^2)$, etc

Exponentials.

Any exponential function "dominates " any polynomial:

$$\frac{n^k}{c^n} \underset{n\to\infty}{\longrightarrow} 0 \quad \text{for all } k > 0, c > 1$$

Thus $n^k = o(c^n)$, for any $c > 1$

In general: if $c_1 < c_2$ then ${c_1}^n = o({c_2}^n)$

Notation: $f(n) = 2^{O(g(n))}$ iff $\exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq 2^{c \cdot g(n)}$

Time complexity classes

Definition

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

For example, consider the language

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

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

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

Computing An Algorithms Complexity

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.

Definitions

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

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

Polynomial Time

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

- On input of a string determines whether string *is or is not* in the language.

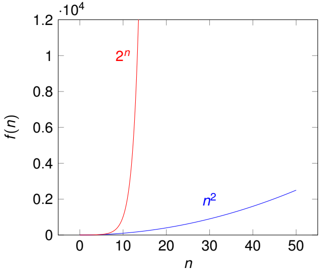
Could also define P as the class of languages which are *recognizable* in polytime.

- If $x \in L$ outputs “accept” in polytime.
- If $x \notin L$ output and runtime undefined.

Exercise: Show that a polytime algorithm to recognize a language in P can be turned into a polytime algorithm to decide a language in P.

Exponential Time

EXP is the class of languages decided by Turing machines with runtime exponential in the input size.



Gap: super-polynomial, sub-exponential ($\forall c > 1 : f(n) = o(c^n)$),
e.g.: $f(n) = n^{\log n}$.

The Class P

P is **robust** (i.e. not affected by model of computation)

P is a mathematical **model** of "realistically solvable" or **"tractable"** problems (Cobham's thesis)

- Caveat: running time $c \cdot n^k$ could have c large or k large
- Assumes no other resources (compare to BPP later in course).

Examples

- $\mathit{FACTORING} \overset{?}{\in} \mathrm{P}$
  - $\mathit{FACTORING} = \{(N, M) \mid N \text{ has a prime factor } 1 < k < M\}$
  - Brute force: $O(2^{n/2})$. Best known algo: $2^{O(n^{1/3}(\log n)^{2/3})}$
- $\mathit{PATH} \in \mathrm{P}$
  - $\mathit{PATH} = \{(G, s, t) \mid G \text{ is directed graph w/ path from } s \text{ to } t\}$
  - See next lecture for proof of $\mathit{PATH} \in \mathrm{P}$.

Are All Interesting Computational Problems Easy?

This is the central question of computer science.

If the answer is true then eventually computers will solve all problems

If false then there will always be a need for computer scientists to solve new problems.

Mathematically we can ask whether all interesting problems lie in P.

- Depends of course on our definition of interesting.
- Clearly HALT is interesting, but it is clearly not easy.

The SATisfiability problem

SAT

- Boolean variables: x take values 1 (TRUE) or 0 (FALSE)
- Boolean operations: AND ($x_1 \wedge x_2$), OR ($x_1 \vee x_2$), NOT ($\overline{x}$)
- Boolean formulas: e.g. $\phi = (\overline{x}_1 \wedge x_2) \ \vee \ ((x_1 \wedge \overline{x}_3) \vee x_2)$

A boolean formula is **satisfiable** if there exists an assignment of 0's and 1's to the variables, s.t. the formula evaluates to 1.

Note a formula with n variables has 2^n possible assignments

The SAT language is defined as

$$\textbf{\textit{SAT}} := \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

SAT

The SAT problem is pretty basic, it asks whether a formula can be made to be true.

- Practical SAT solvers, i.e. algorithms to solve SAT, have huge commercial applications today!
- From verifying software and hardware, to combinatorial optimization applications.

So we can say with certainty that SAT is interesting

Question: Is $\textbf{\textit{SAT}} \in \mathrm{P}$?