COSC 341
Theory of Computing
Lecture 16
How long will it take?

Stephen Cranefield
stephen.cranefield@otago.ac.nz

Lecture slides (mostly) by Michael Albert

Keywords: Time complexity

Time-complexity of Turing machines

- From now on a standing assumption is that the Turing machines we're considering halt on all inputs.
- The time required for a transition will be called a tick (all transitions are assumed to require the same length of time).
- This allows us to define the time complexity of M to be the function:

$$tc_M : \mathbb{N} \to \mathbb{N}$$

where $tc_M(n)$ is the maximum number of ticks required for the operation of M on any input string of length n.

Does hardware matter?

- How does the type of TM we're using affect the time-complexity?
- What does that mean exactly?
- It's a bit fuzzy but we have a feeling for what it means for two different TMs (say standard and multi-tape) to be implementing the "same" algorithm.
- The underlying hardware can make it easier to do things – for example to recognise SQUARE we saw that a two-tape machine might require only linear time, while a standard TM seems to need quadratic time.
- What has been observed again and again is this: *the extra overhead required to simulate one model of deterministic computation in another is always a polynomial in the size of the input.*

Edmonds and Cobham (1964/5)

Edmonds:

An explanation is due on the use of the words "efficient algorithm" . . . For practical purposes the difference between algebraic and exponential order is more crucial than the difference between [computable and not computable] . . . It would be unfortunate for any rigid criterion to inhibit the practical development of algorithms which are either not known or known not to conform nicely to the criterion.

Cobham:

For several reasons the class $\mathbf{P}$ *seems a natural one to consider. For one thing, if we formalize the definition relative to various general classes of computing machines we seem always to end up with the same well-defined class of functions. Thus we can give a mathematical characterization of* $\mathbf{P}$ *having some confidence it characterizes correctly our informally defined class.*

The complexity class $\mathbf{P}$

We say that a decision problem PROB $\in \mathbf{P}$ if PROB can be resolved by a deterministic Turing machine M with

$$tc_M(n) = O(n^c)$$

for some constant c.

> $\mathbf{P}$ *is the class of decision problems that can be resolved by a deterministic Turing machine* ***whose running time is bounded by a polynomial in the input size****.*

Briefly, a problem in $\mathbf{P}$ can be solved “in (deterministic) polynomial time”.

Decision problems and languages

What do decision problems have to do with languages
(and therefore Turing machines)?

- A decision problem corresponds to a subset X of Σ^*.
- The problem is to decide, given w, whether $w \in X$.

What about problems that involve constructing an answer?

For complexity analysis we can generally consider these as a combination of a decision algorithm and binary search.

The class $\mathbf{NP}$

- What happens if we allow non-determinism?
- Effectively, this allows a *guess and check* approach to our problems.
- For this reason, non-deterministic machines that solve decision problems are frequently called verifiers.
- The time-complexity of such a machine is the run-time of its deterministic part (i.e., after the guess has been entered) maximised over all inputs of a given length (but minimised over correct guesses for any given input – though this is rarely relevant)

$\mathbf{NP}$ *is the class of decision problems that can be resolved by a non-deterministic Turing machine* ***whose running time is bounded by a polynomial in the input size****.*

Note that $\mathbf{P} \subseteq \mathbf{NP}$: a problem in $\mathbf{P}$ can be 'verified' by just deciding it again.

Whether $\mathbf{P} = \mathbf{NP}$ is a hugely significant open problem.

Example problem in $\mathbf{NP}$

FAIR-DIVISION

- **Instance**: A sequence $a_1, a_2, \ldots, a_n$ of positive integers.
- **Problem**: Find a partition of the sequence into two parts with equal sums.

Note: If we represent the a_i in unary notation, e.g. $111\#11\#111111\#111\#1111$, this is in $\mathbf{P}$.

However, in time complexity problems, by convention, integers should be represented in a base > 1 (typically binary). The input size for FAIR-DIVISION is then $n \log(\max\{a_i\})$.

If we want to look at all of the 2^n possible divisions, we won't have a polynomial algorithm.

However, if a helpful genie tells us a correct division, we can easily verify it in polynomial time: we just need to add up the two groups.