1 Introduction

How do we measure (and compare) the performance of various algorithms for solving computationally interesting problems?

The "real world" answer is "implement and test" but this can certainly lead to wasted effort if no efficient algorithm exists and/or misleading results if the test cases are not representative.

It turns out that in trying to draw the line between realistic problems and unrealistic ones (i.e., ones that we know how to solve, but cannot afford to wait for the solution) that our model of computation is not generally relevant.

Church-Turing thesis *Any two reasonable models of algorithms or computation will produce the same set of computable functions or languages. In particular, any problem that can be solved by an algorithm can be solved using a Turing machine.*

A stronger (and arguably less well-justified) version of this thesis is:

Any two reasonable models of algorithms or computation will produce the same set of efficiently computable functions or languages. In particular, any problem that can be solved by an efficient algorithm can be solved efficiently using a Turing machine.

2 Time complexity of Turing machines

Before we start to think about what "efficient algorithm" means, we have to find a way of measuring the performance of (an implementation of) an algorithm (as a Turing machine). Fortunately, this is pretty straightforward: each transition of a Turing machine is assumed to take "one tick" (for a suitable value of "tick"), and we just count the number of ticks that happen before the machine halts.

We're only interested in Turing machines that halt on all possible inputs, so suppose that we have such a machine M. The parameter of interest in describing the running time is the size, i.e., length, of the input string. So we define the *time complexity*, tc_M, of M to be a 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.

The *qualitative* behaviour of tc_M is much more important than its *quantitative* behaviour for a variety of reasons. The first reason is captured in the table below:

n	$1000n$	n^2	n^{10}	$(1.01)^n$
1	10^3	1	1	1
10	10^4	10^2	10^{10}	1
10^2	10^5	10^4	10^{20}	2
10^3	10^6	10^6	10^{30}	2×10^4
10^4	10^7	10^8	10^{40}	10^{43}
10^5	10^8	10^{10}	10^{50}	10^{433}

For "large" values of n the order of the values of these functions are determined by their form (linear, quadratic, tenth-power, exponential) and this would be unchanged by multiplication by any constant, the addition of any number of smaller terms, or (in the last case) the change of the base of the exponential to any constant greater than 1.

But, a more important reason that we should only be concerned with a very rough qualitative measure of time-complexity for Turing machines is that it's very sensitive to precisely what sort of Turing machine it is that we're using.

3 The overhead of different computing models

In a one-tape Turing machine if we're trying to keep track of a number of different items of data then there's an awful lot of shuttling around on the tape that happens, since we frequently need to move all the way along the tape to the place where we've saved the information that we're looking for. On a two-tape or multi-tape machine we can eliminate a lot of that.

For example, consider the long division problem:

LONG DIVISION
Instance: A pair of positive integers a and b.

Problem: Determine, using long division, integers q and r such that $a = qb + r$ and $0 \leqslant r < b$.

Note this isn't a decision problem and it's weird to specify the algorithm to be used but the point of the example is to consider the different overheads that might arise when we use the same algorithm in different models of a Turing machine.

Aside: Let's agree for once and for all that from now on we're going to be representing integers in binary notation (or any non-unary base). This is actually quite important from the standpoint of measuring complexity. The point is this: to represent the integer n in unary requires space n whereas to represent it in binary requires space only $\log_2 n$. Therefore, in measuring the time-complexity of Turing machines that are operating on integer input the *range* of integers under consideration is much bigger in the binary case (roughly speaking, in the binary case $tc_M(n)$ represents the worst-case running time for integer inputs up to 2^n, while in the unary case only for inputs up to n.)

Now returning to long division – basically, we left-shift b as far as we can (keeping at or below a) before subtracting that from a (recording the high-order bit of the quotient, q) and then carry on much the same way. Let's consider how this would happen in two different models:

- In a one-tape Turing machine we're going to need to shuttle across the entire word representing a as we process each bit of b. We can economise a bit by using "finite state as memory" to process b in blocks of some fixed length, but that just reduces the number of times we need to shuttle back and forth by a constant factor.
- In a two-tape Turing machine we could store b on one tape and a on the other. Then we could basically process a from left to right only needing to shuttle backwards and forwards across b as we compute a bit of the quotient.
- In either case we could work in "bytes" "32-bit words", "64-bit words", "4096-bit words" or whatever, but this reduces the total amount of work by a constant factor.

Now let's consider (very) roughly what the difference in performance of the same algorithm is in the two cases. Using $|x|$ to stand for the length in bits of an integer x, and n

for the total length of our input, we have:

- In the first model, each bit of a and each bit of b might be responsible for a complete shuttle across the input word. So, worst case, we'll use at least some constant multiple of $(|a| + |b|)|a||b|$ "ticks" (total length of the input, $|a| + |b|$, multiplied by number of pairs of bits, one from a and one from b). Since all three of these factors could be proportional to n the best we could hope for would be something of order n^3 (and I have a sneaking suspicion that I may have lost a power of n there, but details aren't the important thing at the moment!)
- In the second model we don't shuttle across a at all and only do so across b a number of times proportional to the length of q (which is roughly $|a| - |b|$. So again, very very roughly, we're looking at something like $|b|(|a| - |b|)$, i.e., order n^2.

There's no doubt that order n^2 is a whole lot better than order n^3, so if we want a model of "efficiency" that's robust to our choice of computing model we have to allow for that degree of wiggle room.

Aside: There's an extensive literature on optimising the complexity of division or the arithmetic operations in general (since this is obviously of practical importance particularly in cryptographic applications). The Wikipedia page on the computational complexity of mathematical operations is a good starting point. Suprisingly, there was a major breakthrough in 2015 to hit the obvious bound of $O(n \log n)$ – though the devil here is in the details. The overhead involved in squeezing down the exponent on n is such that the algorithm accomplishing this is probably not more practical than those presently in use for integers that can be stored in the observable universe – but that too may change.

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.

The word "deterministic" in that statement is essential. To simulate a non-deterministic Turing machine with a deterministic one definitely seems to require exponential overhead (we need to consider each possible input on the oracle tape and simulate the operation of the underlying deterministic machine on that input). Likewise, to simulate the operation of a quantum computer in a classical one requires exponential overhead (but only if the algorithm being simulated makes fundamental use of the quantum properties – aside from integer factoring "practical" exponential speedup in a quantum computer seems difficult, if not impossible, to achieve – never mind the slight hitch that we don't know how to build them to any reasonable size yet.)

4 Efficient algorithms

What does the discussion of the previous section imply about a definition of *efficient algorithm*? Let me turn to the literature ...

> 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.

[1] Jack Edmonds, *Paths, trees, and flowers*, Canad. J. Math. **17** (1965), 449–467.

> For several reasons the class **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 **P** having some confidence it characterizes correctly our informally defined class.

[1] Alan Cobham, *The intrinsic computational difficulty of functions*, Logic, Methodology and Philos. Sci. (Proc. 1964 Internat. Congr.), 1965, pp. 24–30.

What both these authors are saying (and what has become the widely-agreed definition) is that an algorithm should be regarded as efficient if its worst-case run time is bounded by (any!) polynomial in the input size.

Aside: This is *not* a practical definition of efficiency. We really aren't interested in algorithms that require n^{400} or even n^6 operations on input of size n. But it is a *robust* definition. And, experience has shown that once you know a problem is "efficient in the sense of the definition" then, if it is of sufficient practical interest, someone will soon come up with an algorithm that is "efficient in practice".

5 The class P

Rather than talking about "efficient algorithms" we really want to talk about "problems for which efficient algorithms exist". So, as usual, we will be thinking about *decision problems*, i.e, subsets $L \subseteq \Sigma^*$, and the associated problems:

DECIDE-L
Instance: A word $w \in \Sigma^*$.
Problem: Determine whether $w \in L$.

Given a decision problem, PROB (i.e., some instance of DECIDE-L for some L) we say that PROB $\in \mathbf{P}$ if PROB has a decision algorithm represented by a deterministic Turing machine M that halts on all inputs such that

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

for some constant c.

Remember that this means that there is a constant $A \geq 0$ such that

$$tc_M(n) \leq An^c$$

for all $n > 0$, i.e., the running time of the machine is bounded above by a polynomial of the input size.

Once more (because it's important!):

> $\mathbf{P}$ *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".

6 The class *NP*

Consider the following decision problem:

FAIR DIVISION
Instance: A finite list of positive integers.
Problem: Can the list be partitioned into two sublists A and B whose sums are equal?

A *non-deterministic* Turing machine can easily solve this in polynomial time:

- Set up two variables $a = 0$, $b = 0$.
- For each item in the list, guess whether it belongs to A. If so, add its value to a, if not add it to b.
- If, after processing the whole list, $a = b$ then answer **Yes**, otherwise **No**.

The power of non-determinism is in the "guessing" part of the procedure – a non-deterministic TM is entitled to guess correctly*if at all possible*. We could (and perhaps should) describe this process by means of a deterministic Turing machine with oracle tape. All we need to record on the oracle tape is a series of bits with "0" meaning "include this item in A" and "1" meaning "include this item in B."

This leads to the following natural definition:

> **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***.

Remember that an important aspect of non-determinism is that *we must be able to verify* positive instances – i.e., we must be sure that when the genie writes some information on the oracle tape that gives rise to an accepting computation then we really should have accepted. Otherwise the "heads **Yes**, tails **No**" machine would show that every problem was in **NP** which would not be very useful!

For this reason non-deterministic machines that solve decision problems are frequently called ?? (and often, as in the case of FAIR-DIVISION above that's what they do!)

7 Tutorial problems

We'll be using the following decision problems:

PRIME
Instance: A positive integer n.
Problem: Is n prime?

FACTOR-BOUND
Instance: A pair of positive integers m and n.
Problem: Does m have a divisor d such that $1 < d \leqslant n$?

Whenever one has a property, X, of languages, its "co-"property, co-X holds of a language L exactly when $\Sigma^* \setminus L$, i.e., the complement of L, has property X.

1. Why are co-**P**and **P** the same set of languages?
2. Please tell me if you have determined whether or not co-**NP**and **NP** are the same set of languages.
3. Show that PRIME is in co-**NP**. In fact, though this is by no means obvious, PRIME is in **P**.
4. Show that FACTOR-BOUND is in **NP**.
5. Show that FACTOR-BOUND is in co-**NP** (hint(?): this requires knowing that PRIME is in **P**).

6. **PSPACE** is the class of all decision problems that can be solved on a Turing machine with a polynomial bound on the use of space (that is, there are some constants A and c such that if the input is of size n, the tape-head never moves more than An^c cells away from the left-hand end of the tape).

   CLOBBER is a two-player game played with black and white stones occupying (some finite set of) squares within an infinite grid. One player, Bella (she/her) controls the black stones and the other, William (he/him) controls the white ones. A move for Bella is to take one of her stones, move it to an adjacent square occupied by one of William's stones, and remove his stone from the board. William moves similarly, but using his stones to clobber Bella's. The game is over when one of the players is unable to make a move (because there are no pairs of stones of opposite colours next to each other), and the player who made the last move wins. Consider the problem:

   WIN-CLOBBER
   Instance: A CLOBBER position
   Problem: Does Bella have a winning strategy moving first?

   Show that WIN-CLOBBER is in **PSPACE**. Does it seem likely to be in **P**, **NP**, or co-**NP**?