Complementary Lecture Notes (3)
Background for Computational Complexity

MSc in Physics Engineering - Quantum Computation, L. S. Barbosa, 2025-26

Summary.
(1) Computational complexity. The asymptotic notation.

(2) Case study: closure operations.

(3) Complexity classes. Examples.

(4) Going quantum: The BQP class.

---

1 Analysing complexity

The focus

The focus of this lecture is *computational efficiency*, i.e. the attempt to quantify the amount of computational resources (e.g. time, space, length of messages to be exchanged) required to solve a given problem. Or, putting it in different way, to answer the question of knowing how do such resources scale with the size of the problem (for a suitable definition/measure of *size*).

Actually, the efficiency of an algorithm will be discussed by studying how its number of basic operations scales as the size of the input increases.

A fundamental message is that the efficiency of an algorithm is, up to a considerable extent, much more important than the technology used to execute it.

Example

Integer multiplication by the repeated addition algorithm (to compute $x \cdot y$, just add x to itself $y-1$ times) or by the usual grade-school algorithm illustrates the point. For example, multiplying 577 by 423 using repeated addition requires 422 additions, whereas doing it with the grade-school algorithm takes 3 multiplications of a number by a single digit and 3 additions. Even for 11-digit numbers, a pocket calculator running the grade-school algorithm would beat the best current supercomputer running repeated addition.

To study computational complexity some *abstractions* are in order. First of all, an algorithm can be abstracted as a computational task accepting as input and returning as output a sequence of bits. This is done without loss of generality because, with a linear overhead in its length, any string from any other alphabet can be encoded as a bit string.

Moreover, one may restrict attention to *decision problems*, i.e. problems that have a Boolean output, thus encoded as a single bit. For example, *given two numbers are they relatively prime?*.

A function h with a Boolean output (which is special case of a function from strings to strings) can be identified with a set

$$L_h \mathrel{\widehat{=}} \{x \mid h(x) = 1\}$$

which forms a particular *language*. Thus, computing h is equivalent to the problem of deciding language L_h, i.e. given a string s decide whether $s \in L_h$.

$\boxed{\text{Example}}$

Consider the following *dinner party problem*: Given a list of acquaintances and a list of all pairs among them who do not get along, find the largest set of acquaintances you can invite to a dinner party such that every two invitees get along with one another.

Let us represent the possible invitees through the vertices of a graph with an edge connecting any two people who don't get along. Thus, the problem becomes that of finding a maximum sized *independent set*, i.e. the bigger set of vertices without any common edges, in a given graph. The problem corresponds to the following language

$$L \mathrel{\widehat{=}} \{(G, n) \mid \exists_{M \subseteq \mathsf{vertices}(G)} .\ |M| \geq n \wedge \forall_{x,y \in M} .\ (x, y) \notin \mathsf{edges}(G)\}$$

whose words are pairs composed by a graph G and a number n for which there exists a conflict-free set of invitees, of size at least n.

Observe this a particularly difficult problem: the obvious algorithm — try all possible subsets until a subset that does not include any pair of guests who don't get along is found is highly inefficient (how long will it take for a group of 100 invitees?).

The study of computational complexity focus on how various kinds of computational resources grow as a function of the input size. This analysis can be carried on in any formalization of a computing device — e.g. Turing machines or circuits. Note that, while computability is *independent* with respect to the particular model of computation considered (cf, the (physical) Church-Turing thesis, which states that every physically realizable *computation device* can be simulated by a Turing machine), this is probably not the case when discussing computational complexity.

The corresponding conjecture is known as the *strong* Church-Turing thesis which claims that every physically realizable *computation model* can be simulated by a Turing machine with polynomial overhead. This is a bit controversial; in particular some problems are known to be solvable in polynomial time on a quantum computer, but not known to be so on a classical one. Moreover, it seems that quantum computations cannot be efficiently simulated in a classical Turing machine [3].

$\boxed{\textbf{Measuring computational complexity}}$

$\boxed{\text{Example: transitive closure of } R \subseteq A^2}$

from 'above': R^* is the smallest relations containing R that is transitive and reflexive.

from 'below':

$$R^* \ = \ \{(a,b) \mid a,b \in A \text{ there exists a path from } a \text{ to } b \text{ in } R\}$$

which suggests an algorithm:

Algorithm 1: TC1.

```
$R^* := \emptyset$;
for $i := 1..n$ do
    for each $i$-tuple $(b_1, \cdots, b_i) \in A^i$ do
        if it is a path then
            $R^* := R^* \cup \{b_1, b_i\}$
    end
end
```

The asymptotic notation to capture the growth rate of a function

To compare how the running time $h(n)$ of an algorithm grows with respect to a reference function f, one may consider the limit, when $n \longrightarrow \infty$, of the ratio

$$R = \frac{h(n)}{f(n)}$$

- R is a constant in $]0, \infty[$: $h(n) = \Theta(f(n))$
- $R < \infty$: $h(n) = \mathcal{O}(f(n))$
- $R = 0$: $h(n) = o((f(n))$

For example, $h(n)$ grows

- quadratically: $h(n) = \Theta(n^2)$
- at most quadratically: $h(n) = \mathcal{O}(n^2)$
- less than quadratically: $h(n) = o(n^2)$

Formally, define

$$\mathcal{O}(f) \ = \ \{g \in \mathbb{N}^{\mathbb{N}} \mid \exists_{c,d \in \mathbb{N}^+}. \forall_n. \ g(n) \leq c.f(n) + d\}$$

Thus, stating that $h \in \mathcal{O}(f)$ means that h *is no faster than* f. An equivalence relation relating rates of growth is defined by

$$f \sim g \ \text{ iff } \ f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f)$$

Example: $p(n) = 31n^2 + 17n + 3$

Clearly $p(n) \leq 48n^2 + 3$, because $n^2 \geq n$. Thus $p \in \mathcal{O}(-^2)$ with constants 48 and 3.
However, $-^2 \in \mathcal{O}(p)$ with constants 1 and 0.

$\boxed{\text{Theorem}}$

For any polynomial $p(n) = c_k n^k + \cdots + c_1 n + c_0$, $p \in \mathcal{O}(-^k)$ with constants $\sum_{1 \le i \le k} c_i$ and c_0.

$\boxed{\text{Theorem}}$

Any two polynomials p and q with the same degree verify $p \sim q$.

$\boxed{\text{Theorem: the exponential barrier}}$

The growth rate of function 2^n is higher than the one of an arbitrary polynomial.

Proof.
We want to show that

$$n^i \in \mathcal{O}(2^n) \quad \text{i.e. } n^i \le c2^n + d$$

Let $c = (2i)^i$ and $d = (i^2)^i$, and consider two cases:

- $n \le i^2 \Rightarrow n^i \le c2^n + d$, because $n^i \le d$
- $n > i^2 \Rightarrow n^i \le c2^n + d$, because we may prove that $n^i \le c2^n$ as follows

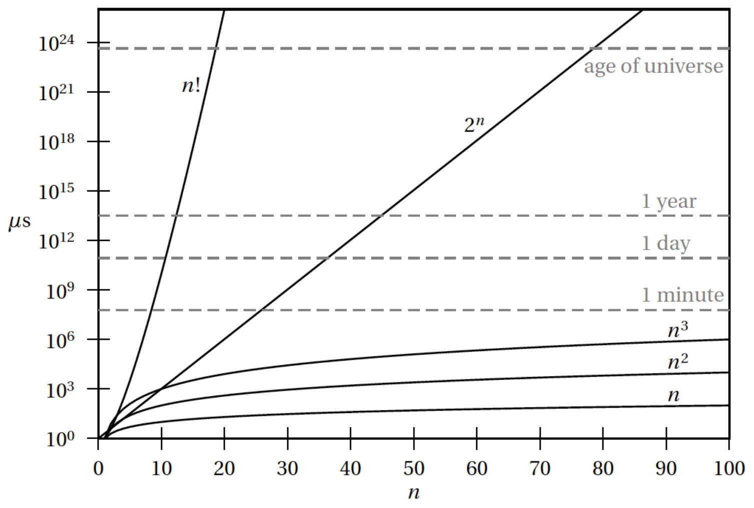
First observe that $n^i \le (iq + i)^i = i^i(q + 1)^i$, for q the integer quotient of n by i (i.e. $iq \le n \le i(q + 1)$). Now,

$$\begin{array}{ll} & i^i(q+1)^i \\ \le & \quad \{\, n \le 2^n \} \\ & i^i(2^{q+1})^i \\ \le & \quad \{\text{ definition of } c\} \\ & c2^{qi} \\ \le & \quad \{\text{ definition of } q\} \\ & c2^n \end{array}$$

Observe now that if a polynomial had the same growth rate than 2^-, then any polynomial of a higher degree would have the same rate (because we've just proved that no polynomial grows as fast as 2^-). But this leads to a contradiction because, as shown above, polynomials of different degrees have different rates of growth.

$\square$

Clearly, 2^n has a higher rate of grow than any polynomial. Other exponential functions — for example, 27^n, n^n, $n!$, 2^{n^2} or 2^{2^n} — have even higher rates of growth. The following diagram, reproduced from [5] illustrates the relevance of being polynomial (assuming execution time is 1 microsecond for $n = 1$).



To reinforce the idea, consider the famous *Travelling Salesperson* problem: Given a map with n cities and distances among them, produce an itinerary that minimizes the total distance travelled.

Clearly, the problem can be solved (e.g. systematic examination of all itineraries). But, on the other hand, it remains unsolvable in any practical sense by current computers: too many itineraries — $(n-1)!$ — to be explored. Notice that a $(n-1)!$ algorithm goes faster than 2^n. For 40 cities the number of itineraries is enormous: $39!$. Even if 10^{15} of them could be inspected per second (a value our of reach of current supercomputers) the required time for completing the calculation would be several billion lifetimes of the universe.

What is a *practically feasible algorithm*? The general answer is: it should run for a number of steps bounded by a *polynomial* in the length of the input, i.e. have a polynomial rate of growth.

Thus, an algorithm whose running time can be upper-bounded by any polynomial function is considered *efficient*. On the other hand, it is regarded as *inefficient* if it can be lower-bounded by any exponential function. There are, of course, many growth rates that fall between polynomial and exponential — for example $n \log n$ — but the polynomial / exponential separation seems to be a basic frontier for all practical purposes. One may wonder why such a *quantitative gap* seems so important and full of deep, foundational implications. The following quote by a well-known expert in quantum computation, Scott Aaronson, is in order:

> *One might think that, once we know something is computable, whether it takes* 10 *seconds or 20 seconds to compute is obviously the concern of engineers rather than philosophers. But that conclusion would not be so obvious, if the question were one of* 10 *seconds versus* $10^{10^{10}}$ *seconds! And indeed, in complexity theory, the quantitative gaps we care about are usually so vast that one has to consider them qualitative gaps as well. Think, for example, of the difference between reading a* 400*-page book and reading every possible such book, or between writing down a thousand-digit number and counting to that number.*
>
> Scott Aaronson [1].

2 Case study: Closure algorithms

Computing R^*

Algorithm 2: TC2.

```
$R^* := \emptyset$;
for $i := 1..n$ do
    for each $i$-tuple $(b_1, \cdots, b_i) \in A^i$ do
        if it is a path then
            $R^* := R^* \cup \{b_1, b_i\}$
    end
end
```

The algorithm <u>examines</u> each sequence $(b_1, \cdots, b_i)$; if this is a path <u>add</u> to the solution. Thus, the total number of basic operations (test and add) is

$$n(1 + n + n^2 + \cdots + n^n)$$

i.e. in each of the n iterations look for paths of length up to n. Therefore, TC1 $\in \mathcal{O}(n^{n+1})$.

Algorithm 3: TC3.

```
$R^* := R \cup \{(a, a) \mid a \in A\}$;
while $\exists_{a_i, a_j, a_k \in A}.\ (a_i, a_j), (a_j, a_k) \in R^*,\ (a_i, a_k) \notin R^*$ do
    $R^* := R^* \cup \{(a_i, a_k)\}$
end
```

- In each iteration one pair (if any) is <u>added</u>. Thus, the maximum number of additions corresponds to the maximum number of pairs available, i.e. n^2.
- In each iteration the algorithm <u>searches for</u> n^3 triples.

Therefore, TC3 $\in \mathcal{O}(n^2 \times n^3) = \mathcal{O}(n^5)$.

Exercise 1

The algorithm TC3 repeatedly searches for violations of the transitivity property. However, each triple must be checked again and again since the introduction of a new pair may entail new violations in triples that have already been checked. A better algorithm of $\mathcal{O}(n^2 \times n) = \mathcal{O}(n^3)$ can be obtained by imposing an order to the triples so that a new pair added does not violate the transitivity condition established for triples already considered. In the algorithm below, TC4, triples are ordered by the middle index (in increasing order). Explain why the algorithm works and its growth rate.

Algorithm 4: TC4.

```
$R^* := R \cup \{(a, a) \mid a \in A\}$;
for $j = 1, 2, \cdots, n$ do
    for $i = 1, 2, \cdots, n$ and $k = 1, 2, \cdots, n$ do
        if $(a_i, a_j), (a_j, a_k) \in R^*$ but $(a_i, a_k) \notin R^*$ then
            $R^* := R^* \cup \{(a_i, a_k)\}$
    end
end
```

Closure problems

A subset $C \subseteq A$ is *closed* for a relation $R \subseteq A^{n+1}$ if

$$b_{n+1} \in C \quad \Leftarrow \quad b_1, \cdots, b_n \in C \ \wedge \ (b_1, \cdots, b_n, b_{n+1}) \in R$$

e.g.

- $\mathbb{N}$ is closed for $+$
- the set of ancestors is closed for the relation *parent-of*
- any set is closed for $\subseteq$

Closure property: *The set* C *is closed under relations* $R_1, \cdots, R_m$

cf, the usual construction *the smallest set that contains* A *and has property* ϕ. But note that not all properties guarante the existence of a smallest set satisfying ϕ. However,

Theorem

If ϕ is a closure property defined by relations $R_1, \cdots, R_m$ on a set A and $B \subseteq A$, then there exists the smallest set C st $B \subseteq C$ and C has property ϕ.

Proof.
Let ϕ be defined by a relations $R_1, \cdots, R_m$ and S denote the set of subsets of A containing B and closed for each R_i. Clearly $S \neq \emptyset$ (why?). Then, define $C = \bigcap S$ (which is well defined because S is non empty). Then,

- $B \subseteq C$, by construction.
- C is closed under all relations $R_1, \cdots, R_m$. To see this, suppose $a_1, \cdots, a_{n-1} \in C$ and $(a_1, \cdots, a_{n-1}, a_n) \in R$. All sets in S contain $a_1, \cdots, a_{n-1}$ and because all of them are closed, all have a_n. Thus, $a_n \in C$.
- C is minimal: no strict subset C' of C exists (otherwise $C' \in S$ and $C \subseteq C'$).

$\square$

Exercise 2

Set C in the theorem above is the *closure* of B under relations $R_1, \cdots, R_m$. Determine the closure of the singleton set containing yourself under the relation *parent of*. Similarly, determine the closure of set $\{0, 1\}$ under addition and the closure of the set of natural numbers under subtraction. Note that R^*, for a relation R is the closure of R under transitivity and reflexivity.

Theorem

Any closure property over a finite set can be computed in polynomial time.

Proof.

Algorithm 5: Computing a generic closure.

```
$C^\circ := C$;
while $\exists_{1 \leq i \leq k}$ and $r_i$ elements $a_{j_1} \cdots a_{j_{r_{i-1}}} \in C^\circ$ and $a_{j_{r_i}} \in D \setminus C^\circ$ . $(a_{j_1} \cdots a_{j_{r_i}}) \in R_i$ do
    $C^\circ := C^\circ \cup \{a_{j_{r_i}}\}$
end
```

Thus, the algorithm is $\mathcal{O}(n^{r+1})$ where n is the cardinal of D and r is the greatest arity of all relations considered.

□

Theorem

Any algorithm in polynomial time can be rendered as the computation of a closure over a set for a set of relations.

3 Complexity classes

Let us classify the complexity landscape. Define

$$\mathrm{TIME}(f(n))$$

as the class of problems for which exists an algorithm solving in time $\mathcal{O}(f(n))$ instances of size n.

The class P

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

The fundamental observation is that exponents are constant with respect to n; in particular they do not grow with n (as e.g. in $n^{\log n}$). On the other hand, the class $\mathsf{EXP} = \mathsf{TIME}(2^{p(n)})$, wehre $p(n)$ stands for a polynomial in n, takes care of execution times growing exponentially with n.

P is the class of all languages for which the membership problem has a classical algorithm that runs in polynomial time and gives the correct answer with certainty. As mentioned above, polynomial computations are regarded as *tractable* or *computable in practice*. On the other hand, non-polynomial computations are regarded as *intractable* as a small variation in the input size may require resources exceeding reasonable limits (e.g. the running time may exceed the number of atoms in the universe).

P may also be characterised as the class of *polynomially decidable* languages, i.e. languages tah can be decided by a polynomially bounded Turing machine. This is a Turing machine that always halts after at most $p(n)$ steps, where $p(n)$ is a polynomial and n is the length of the input.

The class P of such languages is the quantitative analog of the class of recursive languages. As the latter, it is closed under complement, union, intersection, concatenations and Kleene star. But, on the other hand, not all recursive languages are polynomially decidable.

$\boxed{\text{Theorem}}$

$S \notin \mathsf{P}$, where

$$S = \{"M""w" \mid M \text{ accepts input } w \text{ after at most } 2^{|w|} \text{ steps}\}$$

Proof.

If $S \in \mathsf{P}$, language

$$S' = \{"M" \mid M \text{ accepts input } "M" \text{ after at most } 2^{|"M"|} \text{ steps}\}$$

and its complement are also in P. This means that there exists a polynomially bounded Turing machine B which accepts all descriptions of Turing machines that fail to accept their own description in 2^n steps, where n is the length of the description, and halt in $p(n)$ steps for a polynomial $p(n)$.

Does B accept its own description "B"?

- If YES then B fails to accept "B" within $2^{|"B"|}$ steps. However, B halts in $|"B"|$ steps (because, by assumption, the complement of S' is in P). This means that B halts much before $2^{|"B"|}$. Thus it should reject "B", which leads to a contradiction. Note that there is always an integer n_0 such that $p(n) \leq 2^n$ for all $n \geq n_0$, and we may safely assume $|"B"| \geq n_0$.
- If NO a similar argument also leads to contradiction.

The class BPP (from *bounded error probabilistic polynomial time*)

One objection raised against the definition of P is the fact that it sweeps behind the carpet the possibility of resorting to *randomness* as a computational resource. If such a thing exists in the

Universe, one may conceive algorithms involving some sort of random choices such as initializing a variable with a value chosen at random from some range. Such algorithms, classified as *randomized* ou *probabilisitc*, are basically implementations of a random number generator[1]. They can be implemented in *probabilisitc* Turing machines. These are Turing machines with two, rather than one, transition relations in their finite control. At each step one may choose, with equal probability, which of them to apply, a decision which is independent of all previous choices made.

The BPP class plays for randomized algorithms a role similar to that of P for the deterministic case: it captures efficient probabilistic computation. Formally, it is the class of all languages whose membership problem has a classical randomised algorithm that runs in polynomial time and gives the correct answer with probability at least $\frac{2}{3}$ for every input. The class corresponds to the formalisation of decision problems that are feasible on a classical computer.

The choice of $\frac{2}{3}$ above is a bit arbitrary and can be replaced by any other number $\frac{1}{2} + \delta$, with $0 < \delta < \frac{1}{2}$. This is proved as follows:

- Consider an algorithm for a decision problem that works correctly with probability $\frac{1}{2} + \delta$, and repeat its execution k times.
- Take the majority vote of all k answers as the output.
- According to the so-called *Chernoff bound* or *amplification* lemma (see [6] for a proof), this answer is correct with a probability at least $1 - e^{-2k\delta^2}$ approaching 1 exponentially fast. Thus there will be value k such that this probability will exceed $1 - \epsilon$ for any $\epsilon > 0$.
- If the original algorithm had polynomial running time $\tau(n)$, this one will have $k\tau(n)$, which is still polynomial in n.

Clearly $\mathsf{P} \subseteq \mathsf{BPP}$, but strict inclusion remains an open questions: most probably, randomized computation may be no more powerful than deterministic computation.

The class PSPACE (from *polynomial space complexity*)

Is the class of of all decision problems that can be solved within a polynomially bounded amount of space as a function of the input size. Clearly

$$\mathsf{P} \subseteq \mathsf{BPP} \subseteq \mathsf{PSPACE}$$

because any polynomial time computation occurs in polynomial space since polynomial many 1- and 2-bit gates can act on at most polynomial many bits in total. Similarly in any randomised polynomial time computation, for each fixed choice of the random bits, we can perform the associated computation in polynomial space. Then doing this sequentially in turn (re-using the same polynomial space allocation) for each of the exponentially many choices of the random bits, we can keep a running total of accept and reject answers, and thus get $\mathsf{BPP} \subseteq \mathsf{PSPACE}$.

It is not known whether any of these inclusions are strict.

[1]It turns out that generating random bits, i.e. tossing fair coins, is enough.

$\boxed{\textbf{The class } \mathsf{NP}}$

As mentioned in a previous lecture, there is no algorithm other than exhaustive search to decide if in a given graph there is an Hamiltonian path. However, if we are given a specific path it is easy to *verify* whether it is indeed a solution to this problem. The complexity class NP is precisely the class that captures the set of problems whose solutions can be efficiently *verified*, i.e. verified in polynomial time with respect to the length of the input. By contrast, the class P contains decision problems that can be efficiently *solved*. Clearly,

$$\mathsf{P} \subseteq \mathsf{NP}$$

Formally, NP is the class Q of problems such that x is a solution to Q iff there is a polynomial decision problem D and a solution (x, t) to D where the length of t is polynomial in x.

For example, for the Hamiltonian path problem, x is a graph, t is a path and $D(x, t)$ is the decision problem of checking if t is a Hamiltonian path for x.

This can also be formulated in terms of a concrete computational model, for example, Turing machines recognising languages associated to decision problems. Thus, a language $L \subseteq \{0, 1\}^*$ is in NP if there is a polynomial $p : \mathbb{N} \longrightarrow \mathbb{N}$ and a polynomial-time Turing machine M such that, for every input $x \in \{0, 1\}^*$,

$$v \in L \;\Leftrightarrow\; \exists_{t \in \{0,1\}^{p(|x|)}}.\; M(x, t) = 1$$

Turing machine M is called a *certifier* for L, and string t a *certificate*, or a *witness* for input x.

Another way to think about the complexity class NP is as the class of languages that can be decided by a polynomially bounded nondeterministic Turing machine. Note the meaning of *decision* in this context. For a nondeterministic Turing machine to decide a language means that all the computations of the machine must reject an input not in the language, whereas an input from the language must be accepted by at least one computation. Actually, the qualifier NP does not refer to non-polynomial, but to non-deterministic.

Actually, most interesting problems mentioned above for which no polynomial algorithm exists — *Traveling Salesperson, Satisfiability, Independent Set, Integer Partition*, etc., can be solved by polynomially bounded *nondeterministic* Turing machines. As in the probabilistic case, such machines have two transition functions which are arbitrarily chosen before being applied. Given an input, the machine decides if there is a sequence of (nondeterminisitic) choices leading to an acceptance state.

Although determinism and nondeterminism in the definition of Turing machines do not interfere on their expressiveness in what concerns decidability, separating determinism from nondeterminism at the polynomial level, remains unsolved. It is the famous $\mathsf{P} \neq \mathsf{NP}$ conjecture, which addresses the question whether or not the two classes are the same. How long does it take to solve a NP problem deterministically? The most general approach resort to exhaustive search, thus $\mathsf{NP} \subseteq \mathsf{EXP}$.

Finally, another way to characterise the class NP is to build on the bijection between decision problems and logical properties: a decision problem corresponds to a property $\phi(x)$ which holds if x is one of its solutions. Thus, the class of NP properties has the form

$$\phi(x) \;=\; \exists_{t \text{ polynomial in the length of } x} \,.\; D(x,t)$$

with D in class P. The existential quantifier stands for the process of finding a witness t.

Example

The *dinner party problem* discussed above in NP. Its language contains all pairs (G, n) such that graph G has a subgraph of at least n vertices with no edges between them. A polynomial-time verification algorithm proceeds as follows:

- given a pair (G, n) and a string $t \in \{0,1\}^*$, output 1 iff t encodes a list of n vertices of G such that there is no edge between any two members of the list;
- clearly, (G, n) is in the language iff there exists a t such that $M(G, n, t) = 1$. Therefore, the language is in NP. The sequence t of v vertices is the certificate that (G, n) is in the language.

Note that for G with N vertices, the certifier string t can be encoded with at most $\mathcal{O}(n \log N)$ bits. This is polynomial in the size of the representation of graph G.

Clearly $P \subseteq NP$, by making $p(|v|) = 0$ and thus reducing the certifier t to the empty string. The reverse inclusion is far less obvious. As Scott Aaronson puts it [1],

> *If* $P = NP$ *then the ability to check the solutions to puzzles efficiently would imply the ability to find solutions efficiently. An analogy would be if anyone able to appreciate a great symphony could also compose one themselves!*

Or, formulated in another way,

> *If* $P = NP$ *then whenever a theorem had a proof of reasonable length, we could find that proof in a reasonable amount of time. In such a situation, we might say that "for all practical purposes", Hilbert's dream of mechanizing mathematics had prevailed, despite the undecidability results of Gödel, Church, and Turing.*

NP**-complete problems**

The *dinner party problem* has an important completeness property: All problems (languages) in NP can be reduced to it in polynomial time (just as all recursively enumerable languages reduce to the halting problem). The intuitive meaning of a problem being reducible to another is that if one is able to solve the later quickly, one is also able to solve the former at a similar rate.

Formally, a language L *reduces* to another language H if there is a polynomial-time computable function $h : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that

$$\forall_{v \in \{0,1\}^*} .\;\; v \in L \;\Leftrightarrow\; f(v) \in H$$

Such NP problems are called NP-complete, and they are the hardest of all NP problems.

The number of real-life problems that are known to be NP-complete is enormous. Examples include

- Sudoku and jigsaw puzzles;
- the *Traveling Salesperson* problem;
- the satisfability problem for propositional formulas (typically presented in the conjunctive normal form); the result showing that each problem in NP can be reduced to the satisfability problem is known as Cook's theorem.
- more generally, the problem of finding whether a mathematical statement has a proof with a determined number of symbols or fewer, in some formal system.

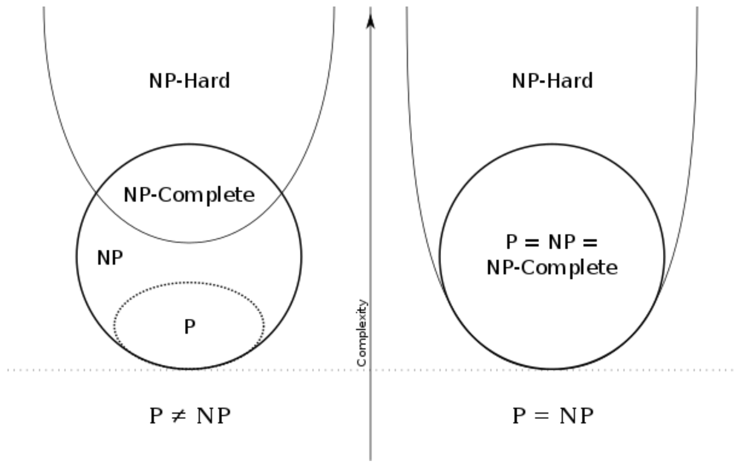
among many others. Note that each of them has a polynomial algorithm iff $\mathsf{P} = \mathsf{NP}$.

NP**-hard problems**

NP-hard problems live even beyond NP, i.e. they are at least as hard as any NP-complete problem. The precise definition here is that a problem Qis NP-hard, if there is a NP-complete problem S, such that S is reducible to Q in polynomial time.

However, differently from NP-complete problems, a NP-hard problem is not required to be in NP. Actually, there are problem which are NP-hard but not NP-complete. For example the *halting problem* studied in the previous lecture, which asks whether a given program and input will halt, is not in NP, even if all NP-complete problems can be reduced to it. On the other hand, as expected, all NP-complete problems are NP-hard.

The following diagram, borrowed from the WWW, sums up our discussion:



4 Problems, problems, problems ...

Colouring vertices in a grph. Given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?

The obvious algorithm is as follows: start with an arbitrary vertex, color it red and all of its neighbours blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color. This is a polynomial problem.

Reachability. Given two nodes of a finite graph decide if there is a path connecting them.

Is a variant of the reflexive-transitive closure problem. Can be solved by computing this closure in time $\mathcal{O}(n^3)$ and inspect the result.

A *problem* is a set of inputs, typically infinite, with a Boolean question to be asked of each input. Note that a problem needs to be encoded as a language problem so that its complexity can be analysed in a common setting. For example, the *Reachability* problem can be reduced to a decision problem for the language

$$R = \{K(G)s(i)s(j) \mid \text{there is a path in } G \text{ connecting nodes } n_i \text{ to } n_j\}$$

where K and s are suitable binary encoding functions for graphs and integers.

Euler Cycle. Given a graph is there a closed path in it that uses each edge exactly once?

Note that the path can go many times through the same node (or even not at all if there are isolated nodes). It can be proved that the necessary condition on a graph to have such a path is that i) all nodes have equal numbers of incoming and outgoing edges, and ii) for each pair of nodes, neither of which isolated, there is a path connecting them. So, clearly the corresponding language

$$G = \{K(G) \mid G \text{ has an Euler cycle}\}$$

where $K(G)$ is some encoding of graphs as strings, is in P.

Hamilton Cycle. Given a graph is there a cycle that passes through each node exactly once?

No polynomial algorithm is known. Of course the trivial one (generate all paths and choose) is not polynomial.

Equivalence of Finite Automata. Given two deterministic automata, determine whether they recognise the same language?

The problem is polynomial, as it is the variant in which only regular expressions are considered. However, one cannot conclude about the latter just by reducing to the former: actually, the generation of a finite automaton from a regular expression may increase exponentially the number of states.

Integer Partition. Given a set of n nonnegative integers represented in binary, is there a subset S of the original set such that $\sum_{i \in S} a_i = \sum_{i \notin S} a_i$?

The algorithm is $\mathcal{O}(nV)$ where V is the sum of all numbers in the original set divided by 2. In spite of its polynomial appearance, the problem is not polynomial in the length of the input. The reason is that the integers are encoded in binary: if all integers are about 2^n, then S is close to $2^n \times \frac{n}{2}$.

Satisfiability. Is a Boolean formula in conjunctive normal form satisfiable?

No polynomial algorithm is known. However, if reduced to formulas with a maximum of two literals, it becomes polynomial.

Optimisation problems require us to find the best among many possible solutions, according to some cost function. The trick to transform optimisation into language problems is to fix each input with a *bound* on the cost function. For example, the *Traveling Salesperson* problem can be rephrased as *given an integer* $n \geq 2$*, a* $n \times n$ *distance matrix, and an integer* $b \geq 0$*, find a permutation of* n *such that its cost is less or equal to* b (which, to build up intuition, may be regarded as a budget).

Independent Set. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \geq k$ such that for any two vertices in s there is no edge connecting them? This is exactly the *dinner party* problem.

Clique. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \geq k$ such that for all vertices in s there is an edge connecting each pair?

Node Cover. Given an undirected graph and an integer $k \geq 2$ is there a subset s of the set of vertices with $|s| \leq k$ such that s covers all edges of the graph, e.g. to minimise the number of guards in a museum?

Note that a set of nodes covers an edge if it contains at least one endpoint of the edge.

No polynomial algorithms are known for these problems.

5 Going quantum: The BQP class

To analyse computational complexity in the context of quantum computation it is usual considering both *time complexity*, typically measured by the total number of gates in a quantum circuit, and *query complexity*. The latter is simply the number of times a computation resorts to an *oracle*. Actually, the input is specified as an oracle that computes some (Boolean valued) function. The oracle is accessed as a black-box by supplying values and retrieving a result, but the algorithm have no access to its internals. The task is to determine whether some property of the function embodied in the oracle holds, by querying it the least possible number of times.

Examples of useful oracles used in typical quantum algorithms include

balanced function Determine whether a function is balanced or constant.

search Find the unique value k such that $f(x) = 1$.

periodicity Determine whether there is a least k such that $f(x + k) = f(x)$ for all x-

Boolean satisfability Determine whether there is an input x making $f(x) = 1$.

$\boxed{\text{The class BQP (from \textit{bounded error quantum polynomial time})}}$

This is the class of languages L such that there is a polynomial time quantum algorithm for deciding membership to L, i.e. for each input size n there is a quantum circuit whose size is bounded polynomially on n and for any input string the output answer is correct with probability at least $\frac{2}{3}$. In other, equivalente words, it is the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of at most $\frac{1}{3}$ for all instances. The error bound, as it happens in the probabilistic case, is largely arbitrary.

The classical counterpart of BQP is the class BPP discussed above. Actually, it generalizes BPP

$$\text{BPP} \subseteq \text{BQP}$$

Furthermore, we have

$$\text{P} \subseteq \text{BPP} \subseteq \text{BQP} \subseteq \text{PSPACE}$$

which means that class of problems that can be efficiently solved by quantum computers includes all problems that can be efficiently solved by deterministic classical computers but does not include any problems that cannot be solved by classical computers with polynomial space resources. There is some evidence, but not a proof, that BQP is a strict superset of P, thus asserting that there might be problems that are efficiently solvable by quantum computers but not so by deterministic classical computers. For example, the *discrete logarithm problem* is known to be in BQP, but most probably lies outside P. In any case, proving strictness in the chain of inclusions above can be anticipated to be very hard, as the problem P $\neq$ PSPACE is still to be solved.

Much is also not known about how BQP related to NP. Clearly some NP problems (namely the discrete logarithm problem mentioned above) are in BQP, but in general it is conjectured that NP $\nsubseteq$ BQP and that, in particular, BQP and the class of NP-complete problems are disjoint. Notice that, if that was not the case all NP problems would be in BQP (why?).

Another important conjecture indicates that some problems in BQP are as harder as NP-complete problems. This, jointly with the observation that many BPQ problems seem to exist outside P, supports the belief that quantum computation is indeed much more powerful that classical computation. A formal proof of this statement is still lacking as the P $\neq$ PSPACE remains unsolved.

Further study

My preference on complexity theory is Papadimitriou's wonderful book [6]; reference [4] provides an interesting alternative. Both S. Arora and B. Barak book [2] and Moore and Mertens *The Nature of Computation* [5] are more recent textbooks covering recent achievements in complexity theory (including challenges from quantum computation) and putting them in the context of the classical results.

References

[1] S. Aaronson. Why philosophers should care about computational complexity. In B. J. Copeland, C. Posy, and O. Shagrir, editors, *Computability: Turing, Gödel, Church, and Beyond*, pages 261–328. MIT Press, 2013.

[2] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[3] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A*, 400:97–117, 1985.

[4] D. Z. Du and K. I. Ko. *Theory of Computational Complexity*. Addison-Wesley, 2000.

[5] C. Moore and S. Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[6] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.