QUANTUM INFORMATION & COMPUTATION

Nilanjana Datta, DAMTP Cambridge

1 Basic notions of classical computation and computational complexity

We begin by setting down the basic notions of classical computation which will later be readily generalised to provide a precise definition of quantum computation and associated notions of quantum computational complexity classes.

Computational tasks:
The **input** to a computation will always be taken to be a bit string. The **input size** is the number of bits in the bit string. For example if the input is 0110101 then the input size is 7. Note that strings from any other alphabet can always be encoded as bit strings (with only a linear overhead in the length of the string). For example decimal integers are conventionally represented via their binary representation.

A **computational task** is not just a single task such as "is 10111 prime?" (where we are interpreting the bit string as an integer in binary) but a whole family of similar tasks such as "given an n-bit string A (for any n), is A prime?" The output of a computation is also a bit string. If this is a single bit (with values variously called 0/1 or "accept/reject" or "yes/no") then the computational task is called a **decision problem**. For computational complexity considerations (cf later), we will be especially interested in how various kinds of computational resources (principally time – number of steps, or space – amount of memory needed) grow as a function of input size n.

Let $B = B_1 = \{0, 1\}$ and let B_n denote the set of all n-bit strings. Let B^* denote the set of all n-bit strings, for all n i.e. $B^* = \cup_{n=1}^{\infty} B_n$. A subset L of B^* is called a **language**. Thus a decision problem corresponds to the recognition of a language viz. those strings for which the answer is "yes" or "accept" or 1, denoting membership of L. For example primality testing as above is the decision problem of recognising the language $L \subseteq B^*$ where L is the subset of all bit strings that represent prime numbers in binary. More general computational tasks have outputs that are bit strings of length > 1. For example the task FACTOR(x) with input bit string x is required to output a bit string y which is a (non-trivial) factor of x, or output 1 if x is prime.

Circuit model of classical computation:
There are various possible ways of defining what is meant by a "computation" e.g. the Turing machine model, the circuit (or gate array) model, cellular automata etc. Although these look quite different, they can all be shown to be equivalent for the purposes of assessing the complexity of obtaining the answer for a computational task. Here we will discuss only the circuit model, as it provides the easiest passage to a notion of quantum computation.

For each n the computation with inputs of size n begins with the input string $x = b_1 \dots b_n$ extended with a number of extra bits all set to 0 viz. $b_1 \dots b_n 00 \dots 0$. These latter bits provide "extra working space" that may be needed in the course of the computation. A **computational step** is the application of a designated Boolean operation (or Boolean gate) to designated bits, thus updating the total bit string. These elementary steps should be fixed operations and for example, not become more complicated with increasing n. We restrict the Boolean gates to be AND, OR or NOT. It can be shown that these operations are *universal* i.e. any Boolean function $f : B_m \to B_n$ at all can be constructed by the sequential application of just these simple operations. The output of the computation is the value of some designated subset of bits after the final step.

Then for each input size n we have a so-called **circuit** C_n which is just a prescribed sequence of computational steps. C_n depends only on n and not on the the particular input x of size n. In total we have a **circuit family** $(C_1, C_2, \dots, C_n, \dots)$. We think of C_n as "the computer program" or algorithm for inputs of size n. (There is actually an extra technical subtlety here that we will just gloss over: we also require that the descriptions of the circuits C_n should be generated in a suitably simple computational way as a function of n, giving a so-called uniform circuit family. This prevents us from "cheating" by coding the answer of some hard computational problem into the changing structure of C_n with n.)

Randomised classical computations:
It is useful to extend out model of classical computation to also incorporate classical probabilistic choices (for later comparison with outputs of quantum measurements, that are generally probabilistic). This is done in the circuit model as follows: for input $b_1 \dots b_n$ we extend the starting string $b_1 \dots b_n 00 \dots 0$ to $b_1 \dots b_n r_1 \dots r_k 00 \dots 0$ where $r_1 \dots r_k$ is a sequence of bits each of which is set to 0 or 1 uniformly at random. If the computation is repeated with the same input $b_1 \dots b_n$ the random bits will generally be different. The output is now a sample from a probability distribution over all possible output strings, which is generated by the uniformly random choice of $r_1 \dots r_k$. (Thus any output probability must always have the form $a/2^k$ for some integer $a \leq 2^k$). Then in specific computational algorithms we normally require the output to be correct "with suitably high probability", specified according to some desired criteria. This formalism with random input bits can be used to implement probabilistic choices of gates. For example suppose we wish to apply either AND or OR at some point, chosen with probability half. Consider the 3-bit gate whose action is as follows: if the first bit is 0 (resp. 1) apply OR (resp. AND) to the last two bits. Then we use this gate with a random input to the first bit.

Polynomial time complexity classes P and BPP

In computational complexity theory a fundamental issue is the time complexity of algorithms: how many steps (in the worst case) does the algorithm require for any input of size n? In the circuit model the number of steps on inputs of size n is taken to mean the total number of gates in the circuit C_n i.e. the *size* of the circuit C_n. Let $T(n)$ be the size of C_n, which we also interpret as a measure of the run time of the algorithm as a function of input size n. We are especially interested in the question of whether $T(n)$ is bounded by a polynomial function of n (i.e. is $T(n) < cn^k$ for all large n for some positive constants k, c?) or else, does $T(n)$ grow faster than any polynomial (e.g. exponential functions such as $T(n) = 2^n$ or $2^{\sqrt{n}}$ or $n^{\log n}$ have this property).

Remark. (Notations for growth rates in computer science (CS) literature.)
For a positive function $T(n)$ we write $T(n) = O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq cf(n)$ for all $n > n_0$, i.e. "T grows *no faster than* f". We write $T(n) = O(\text{poly}(n))$ if $T(n) = O(n^k)$ for some constant k, i.e. T grows at most polynomially with n.
Note that this use of big-O is slightly different from common usage in say calculus, where instead of $n \to \infty$ we consider $x \to 0$ e.g. writing $e^x = 1 + x + O(x^2)$.
In CS usage if $T(n)$ is $O(n^2)$ then it is also $O(n^3)$ but in calculus $O(x^2)$ terms are *not* also $O(x^3)$.
In the CS literature we also commonly find other notations: we write $T(n) = \Omega(f(n))$ to mean $T(n) \geq cf(n)$ for all $n >$ some n_0 and some positive constant c, i.e. "T grows *at least as fast as* f"; and we write $T(n) = \Theta(f(n))$ to mean $c_2 f(n) \leq T(n) \leq c_1 f(n)$ for all $n >$ some n_0 and positive constants c_1, c_2, i.e. T is both $O(f(n))$ and $\Omega(f(n))$, i.e. "T grows *at* rate f".
In this course we will use only the big-O notation (and not Ω or Θ). $\square$

Although computations with any run time $T(n)$ are computable in principle, poly time computations are regarded as "tractable" or "computable in practice". The term **efficient algorithm** is also synonymous with **poly time algorithm**. If $T(n)$ is not polynomially bounded then the computation is regarded as "intractable" or "not computable in practice" as the physical resource of time will, for fairly small n values, exceed sensibly available limits (e.g. running time on any available computer may exceed the age of the universe).

We have the following standard terminology for some classes of languages (or sometimes these terms are applied to algorithms themselves, that satisfy the stated conditions):

P ("**p**oly time"):
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.

BPP ("**b**ounded error **p**robabilistic **p**oly time"):
class of all languages whose membership problem has a classical randomised algorithm that runs in poly time and gives the correct answer with probability at least 2/3 for every input.

The class **BPP** is generally viewed as the mathematical formalisation of "decision problems that are feasible on a classical computer".

Example: Let the problem FACTOR(N, M) be the following: given an integer N of n digits and $M < N$, decide if N has a non-trivial factor less than M or not. The fastest *known* classical algorithm runs in time $\exp O(n^{\frac{1}{3}}(\log n)^{\frac{2}{3}})$ i.e. more than exponential in the cube root of the input size. Thus this problem is not known to be in **BPP**.

Remark (about the definition of **BPP**)
We have required the output to be correct with probability 2/3. However it may be shown

that "2/3" here may be replaced by any other number $1-\epsilon$ that's strictly greater than half without changing the contents of the class i.e. if there is a poly time algorithm for a problem that succeeds with probability $\frac{1}{2}+\delta$ (for any chosen $\delta > 0$, however small) then there is also a poly time algorithm that succeeds with probability 0.500001 or 0.99999 or indeed $1-\epsilon$ for any $0 < \epsilon < \frac{1}{2}$ (however small). This result relies on the following fact, sometimes called the amplification lemma (proved using the Chernoff bound for repeated Bernoulli trials cf Nielsen and Chuang p154 for a simple proof): if we have an algorithm for a decision problem that works correctly with probability $\frac{1}{2}+\delta$ then consider repeating the algorithm K times and taking the majority vote of all K answers as our final answer. Then this answer is correct with a probability at least $1-\exp(-2\delta^2 K)$, approaching 1 exponentially fast in K. Thus given any $\epsilon > 0$ this probability will exceed $1-\epsilon$ for some constant K, and if the original algorithm had poly running time $T(n)$ then our K-repetition majority vote strategy has running time $KT(n)$ which is still polynomial in n. $\square$

Remark. (Optional. Polynomial space complexity.)
If we replace the computational resource of *time* (i.e. number of gates or elementary computational steps) by that of *space* (i.e. amount of memory or number of bits needed to perform the computation) then we obtain the complexity class **PSPACE**, of all decision problems that can be solved within a polynomially bounded amount of space (as a function of input size) and no imposed restriction on time. It is easy to see that we have the inclusions $\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{PSPACE}$. Indeed any poly time computation occurs in poly space since poly many one- and two-bit gates can act on at most poly many bits in total. Similarly in any randomised poly time computation, for each fixed choice of the random bits, we can perform the associated computation in poly space. Then doing this sequentially in turn (re-using the same poly 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 $\mathbf{BPP} \subseteq \mathbf{PSPACE}$.
Astonishingly(!?) it is not known whether any of the preceding inclusions are equalities or strict inclusions!

1.1 Query complexity and promise problems

In quantum computation (cf later) and the study of its properties relative to classical computation, there is another computational scenario that is often considered. This is the formalism of "black box promise problems" with an associated measure of complexity called "query complexity".

In this scenario, instead of being given an input bit string of some length n, we are given as input a *black box* or *oracle* that computes some (here Boolean, but sometimes more general) function $f : B_m \to B_n$. We can query the black box by giving it inputs and this is the only access we have to the function and its values. No other use of the box is allowed. In particular we cannot "look inside it" to see its actual operation and learn information about the function f. Thus, at the start, it is unknown exactly which function f is, but there is often an a priori *promise* on f i.e. some stated a priori restriction on the possible form of f. Our task is to determine some desired property of

f e.g. some feature of the set of all values of f. We want to achieve this by querying the box the *least possible number of times.* In our circuits in addition to our usual gates we may use the black box as a gate, each use counting as just one step of computation. The **query complexity** of such an algorithm is simply the number of times that the oracle is used (as a function of its "size" e.g. as measured by $m + n$). In addition to the query complexity we may also be interested in the total time complexity, counting also the number of gates used to process the answers to the queries in addition to merely the number of queries themselves.

Example 1 *The following are examples of black box promise problems that will be especially relevant in this course.*

The "balanced versus constant" problem
Input: *a black box for a Boolean function $f : B_n \to B$ (one bit output).*
Promise: *f is either (a) a constant function ($f(x) = 0$ for all x or $f(x) = 1$ for all x) or (b) a "balanced" function in the sense that $f(x) = 0$ resp. 1 for exactly half of the 2^n inputs x.*
Problem: *Determine whether f is balanced or constant. We could ask for the answer to be correct with certainty or merely with some probability, say 0.99 in every case.*

Boolean satisfiability
Input: *a black box for a Boolean function $f : B_n \to B$.*
Promise: *no restriction on the form of f.*
Problem: *determine whether there is an input x such that $f(x) = 1$.*

Search
Input: *a black box for a Boolean function $f : B_n \to B$.*
Promise: *There is a unique x such that $f(x) = 1$.*
Problem: *find this special x.*

Periodicity
Input: *a black box for a function $f : \mathbb{Z}_n \to \mathbb{Z}_n$*
(where $\mathbb{Z}_n$ denotes the set of integers mod n).
Promise: *f is periodic i.e. there is a least r such that $f(x + r) = f(x)$ for all x (and $+$ here denotes addition mod n).*
Problem: *find the period r.*

In each case we are interested in how the minimum number of queries grows as a function of the natural parameter n (for quantum versus classical algorithms).