1 Fundamental Classes

In the previous lectures, we defined the notions of time, space and non-determinism abstractly. Here we focus on certain complexity classes of interest defined by imposing specific bounds on resources.

Definition 1 *1.* $\mathsf{LOG} \equiv \mathsf{DSPACE}(\log(n))$

2. $\mathsf{NLOG} \equiv \mathsf{NSPACE}(\log(n))$

3. $\mathsf{P} \equiv \mathsf{DTIME}(\mathsf{poly}(n)) \equiv \bigcup_{k>0} \mathsf{DTIME}(n^k)$

4. $\mathsf{NP} \equiv \mathsf{NTIME}(\mathsf{poly}(n))$

5. $\mathsf{PSPACE} \equiv \mathsf{DSPACE}(\mathsf{poly}(n))$

6. $\mathsf{EXP} \equiv \bigcup_{k>0} \mathsf{DTIME}(2^{n^k})$

7. $\mathsf{NEXP} \equiv \bigcup_{k>0} \mathsf{NTIME}(2^{n^k})$

Note that by Savitch's theorem, non-deterministic polynomial space is the same class as deterministic polynomial space.

Using the general relationships between resources shown in the previous lecture notes, we have the following chain of inclusions:
$\mathsf{LOG} \subseteq \mathsf{NLOG} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$.

Some of the central open questions in complexity theory concern whether these inclusions are strict or not. Note that by the space hierarchy theorem $\mathsf{LOG} \subset \mathsf{PSPACE}$, therefore at least *one* of the first four inclusions is strict, however we do not know which one! In fact, using Savitch's theorem and the space hierarchy, we know that one of the inclusions (2)-(4) is strict, but we do not know which one. Similarly, using the time hierarchy theorem, we know that one of the inclusions (3)-(5) is strict, however we do not know which one.

The main question we will be focusing on in this part of the lecture notes is the NP vs P question, which asks if P is a strict subclass of NP. The class P is especially interesting to us because it is a model of *feasible* computation. There are many problems of great practical relevance which are known to be in NP but are not known to be in P - this is one of the main reasons for the interest in the NP vs P question.

Why do we use the class P as a model of feasible computation? There are many different reasons for this. For one, if a decision problem is in P, that often indicates that it is solvable in practice for a large range of input sizes of interest. For example, suppose there were a problem solvable in n^2 steps on inputs of length n. Then, if we have a processor which is able to perform 10^{12} steps per second, which is a fair approximation to current technology, this processor can be used to solve our problem for input sizes up to 10^6 within a second. However, if a problem requires 2^n steps on inputs of length n, then the processor could only be used to solve the problem for input sizes up to 40 in a second. This difference in input sizes is significant, and reglects the disparity between polynomial and exponential growth.

It is possible that a problem is in polynomial time with a large exponent, say it is solvable in n^{100} steps. This would not seem to suffice for the problem to be feasible in practice. However, we often find that problems solvable in polynomial time are solvable with small exponents, which is a strong indication of practical feasibility.

A second reason for being interested in P is that it is very robust under the choice of machine model. The class P remains the same irrespective of whether our underlying machine model is a multi-tape Turing machine, a single-tape Turing machine, a multi-dimensional Turing machine, a random access machine, a register machine etc. This is of course not a theorem since we do not have a formal definition of "machine model" - however it is a thesis analogous to the Church-Turing thesis. It turns out that for pretty much every "reasonable" model of computation defined so far, the model can be simulated in polynomial time by a Turing machine, and conversely. The one possible exception to this is the quantum Turing machine model - however, since quantum computation has not yet been demonstrated in the real world for significant input sizes, it is unclear whether this is a reasonable model.

A third reason which facilitates the mathematical treatment of the class is its closure under a wide variety of natural operations, such as union, intersection, complement, subroutines etc. To illustrate, one could alternatively attempt to capture the notion of "feasibly computable" by $\mathsf{DTIME}(n)$, but in addition to not being robust under the choice of machine model, this class is not closed under sub-routines. By this, we mean that if a linear-time machine makes a linear number of calls to other linear-time computations, the overall computation is not in linear time (rather, it is in quadratic time). On the other hand, by the closure properties of polynomials,if a polynomial-time procedure makes a polynomial number of calls to polynomial-time procecures, the overall computation is still polynomial-time.

There are many examples of natural problems - decision or function problems - computable in polynomial time. Examples are addition, multiplication and division of integers, sorting, finding a shortest path or minimum spanning tree in a graph and the question of whether a graph has a perfect matching. There are some problems with highly non-trivial polynomial-time solutions, eg. the Primality problem which asks if the input number is prime, and the Linear Programming problem, where we ask how to optimize a given linear function subject to given linear constraints. Primality was only shown to be in polynomial time a few years ago by Agrawal, Kayal and Saxena, while Linear Programming was shown to be in polynomial time by Khachiyan, with a more efficient algorithm due to Karmarkar.

There are however many other natural problems such as the satisfiability problem for Boolean formulae, the problem of finding a clique of a given size in a graph and the problem of factoring a number into its prime factors which are of great practical importance but are not known to be solvable in polynomial time. The class NP comes in useful in terms of relating these problems to one another and understanding their complexity. The Satisfiability and Clique problems are both in NP. The factoring problem as we have defined it is not a decision problem, but we can define a corresponding decision version which also turns out to be in NP.

In general, the class NP contains problems for which *verifying* whether a given solution is correct is easy. Contrast this with the class P, which contains problems for which *finding* solutions is easy. This distinction between finding and verifying is the essence of the NP vs P problem.

The NP vs P problem is not just central to computer science, but to mathematics as well. This is because it is intimately related to the concept of mathematical proof. Say there is a mathematical theorem T we would like to prove. Intuitively it is easy to check whether a purported proof of T is correct, but hard to generate such a proof. This corresponds to the difference between NP and P - verifying a proof is easy, but generating one is hard. In recognition of the fundamental importance of the NP vs P problem, the Clay Mathematical Institute has designated it as one of the seven *Millennium Problems*, with a prize of 1 million dollars for its solution.

2 Reductions, Hardness and Completeness

Intuitively, NP is a more powerful class than P because non-deterministic machines have the ability to explore an exponential-size search space. Justifying this intuition with a mathematical proof is notoriously hard, however. Despite strenuous efforts, limited progress has been made on resolving the NP vs P problem. Even if we don't have a mathematical proof that NP is not equal to P, we would still like to be able to say something inteesting about the "hardness" or otherwise of problems in NP, given that so many natural problems belong to NP. The notion of a reduction allows us to do this.

Reductions are used as a tool to study the *relative* complexities of problems, rather than their *absolute* complexity. A reduction from a problem L_1 to a problem L_2 means that L_1 is an "easier" problem than L_2, since a solution to L_2 would also imply a solution to L_1.

We now define reductions formally. You might have studied reductions in the context of computability theory, where a reduction is a computable function mapping one problem to another. In the context of complexity theory, we also need to bound the resources used by the reduction procedure. This can be done in different ways, and correspondingly we get different notions of reduction. The ones we are most interested in are *polynomial-time* reductions and *log-space* reductions.

We first need the notion of a *Turing machine transducer*, which computes functions rather than solving decision problems.

Definition 2 *A Turing machine transducer is a Turing machine with a designated read-only input tape and a designated write-only output tape. A Turing machine trandsucer M computes a function $f : \Sigma^* \rightarrow \Gamma^*$, where Σ is the input* *alphabet of M and Γ its tape alphabet, if for each input x, M halts with $f(x)$ written on its output tape.*

The time and space used by a Turing machine transducer are defined similarly to the time and space used by a Turing machine, with the input and output tapes not taken into account when measuring resources consumed.

Definition 3 *Let Σ be a finite alphabet. A language $L_1 \subseteq \Sigma^*$ is said to be poly-time reducible to a language L_2 if there is a function f computed by a Turing machine transducer running in polynomial time such that for each $x \in \Sigma^*$, $x \in L_1$ iff $f(x) \in L_2$.*

Definition 4 *Let Σ be a finite alphabet. A language $L_1 \subseteq \Sigma^*$ is said to be log-space reducible to a language L_2 if there is a function f computed by a Turing machine transducer using logarithmic space such that for each $x \in \Sigma^*$, $x \in L_1$ iff $f(x) \in L_2$.*

As mentioned before, reductions give a way of comparing the complexity of two languages. If L_1 is reducible to L_2, this means that L_1 is "at least as easy" as L_2, or equivalently that L_2 is "at least as hard" as L_1. We can use the notion to define hardness for complexity classes.

Definition 5 *Let C be a complexity class and L be a language. L is said to be hard for C under poly-time reductions (resp. log-space reductions) if for each $L' \in \mathsf{C}$, L' is poly-time reducible (resp. log-space reducible) to L.*

Definition 6 *Let C be a complexity class and L be a language. L is said to be complete for C under poly-time reductions (resp. log-space reductions) if for each $L' \in \mathsf{C}$, L' is poly-time reducible (resp. log-space reducible) to L.*

Intuitively, if L is complete for C, this means that L is the "hardest" language in C. A priori, it is not clear that standard complexity classes have complete languages. We show in the next Lecture Notes that not only is this the case, but a variety of *natural* problems are complete for standard classes.

The following propositions justify our claim that L_1 reducible to L_2 means that L_1 is at least as easy as L_2.

Proposition 7 *If L_1 is poly-time reducible to L_2 and $L_2 \in \mathsf{P}$, then $L_1 \in \mathsf{P}$.*

The proof of Proposition 7 is easy. A polynomial-time Turing machine M_1 for L_1 can be designed to first run the poly-time computable reduction f on its input x and then run the polynomial-time Turing machine M_2 for L_2 on $f(x)$, accepting iff M_2 accepts.

Proposition 8 *If L_1 is log-space reducible to L_2 and $L_2 \in \mathsf{LOG}$, then $L_1 \in \mathsf{LOG}$.*

The proof of Proposition 8 is trickier than that of Proposition 7. Suppose we want to design a log-space machine M_1 for L_1 for which there is a log-space reduction f to a a language $L_2 \in \mathsf{LOG}$. Say that the input to M_1 is x. Note that M_1 cannot simply store $f(x)$ on its tapes, since $|f(x)|$ could be polynomial in $|x|$, while M_1 can only afford space $O(\log(|x|))$. Instead of storing the output of the reduction, M_1 *re-computes* bits of the output as and when required. When M_1 simulates the operation of the log-space machine M_2 for L_2 on $f(x)$, at any stage in the simulation, M_2 is reading a specific input bit of $f(x)$, say the i'th bit. Instead of storing all of $f(x)$, M_1 just keeps track of i, and when it needs to simulate the move of M_2 which involves reading the i'th bit of $f(x)$, it runs the reduction on x and retains only the i'th bit of the output. Once the current move of M_2 is completed, it updates i depending on whether M_2 moves left, right or remains stationary on its input tape. Thus, by continually representing $f(x)$ in an *implicit* fashion, M_1 only needs to use $O(\log(f(|x|))) = O(\log(|x|))$ space.