Lecture Series II

The most traditional material

The partition of the rest of the lectures into two lecture series is only due to historical reasons. We start with the more traditional material, most of it is due to the 1970's and the early 1980's.

Notation: We will try to always use n to denote the length of the (main) input.

Lecture 4

Complexity classes defined by a sharp threshold

There is something appealing in defining complexity classes according to a sharp threshold like the class of problems that can be solved within time t, for some function t (e.g., $t(n) = n^3$). Contrast this definition with the class of problems that can be solved within some time t that belongs to a class of functions T (e.g., polynomials). The problem with classes defined according to a (single) sharp threshold is that they are very sensitive to the specific model of computation and may not be closed under natural algorithmic operations. Typically, these problems do not occur when defining complexity classes that correspond to a resource bounded by a class of functions, provided this class has some desirable closure properties.

4.1 Definitions

Focusing on two natural complexity measures (i.e., time and space), we may define for each function $f : \mathsf{N} \to \mathsf{N}$ classes such as $\textsc{Dtime}(f)$ and $\textsc{Dspace}(f)$ corresponding to the class of decision problems that are solvable within time and space complexity f, respectively. (That is, on input x the deciding algorithm runs for at most $f(|x|)$ steps or uses at most $f(|x|)$ bits of storage.)

We stress that when measuring the space complexity of the algorithm, we don't allow it to use its input and/or output device (i.e., tape in case of Turing machines) as temporary storage. (This is done by postulating that the input and output devices are read-only and write-only respectively.)

Note that classes as above are very sensitive to the specific model of computation. For example, the time complexity of multiple-tape Turning machines may be quadratic (but not more) in the time complexity of single-tape Turning machines (e.g., consider the set $\{xx : x \in \{0,1\}\}$).

4.2 Hierarchies and Gaps

A natural property that we may expect from complexity measures is that more resources allow for more computations. That is, if g is *sufficiently greater* than f then the class of problems solvable in time (or space) g should be *strictly larger* than the class of problems solvable in time (or space) f. This property (corresponding to a time or space hierarchy) does hold in the natural cases, where the key question is what is *sufficiently greater*. The answer will be clarified from the way such hierarchy theorems are proved, which is by using *diagonalization*.

Specifically, suppose we want to prove that $\mathrm{DTIME}(g)$ is a strict superset of $\mathrm{DTIME}(f)$. This is done by "diagonalizing against all f-time machines". That is, we construct a set L *along with a decision algorithm for it*, such that no f-time machine can correctly decide L. In order to "effectively" define L, we should be able to emulate the execution of each f-time machine. Since we cannot effectively enumerate all f-time machines, what we do instead is emulate each possible machine while using a time-up mechanism that stops the emulation at time f. In order to do this we need, in particular, to be able to compute f (relatively fast). Note that the running-time of our decision procedure for L is determined by the time it takes to compute f and the time it takes to emulate a given number of steps. Thus, *time constructible functions*, play a central rule in such proofs, where f is time constructible if on input n the value $f(n)$ can be computed within time $f(n)$. As for the emulation overhead, it depends on the specific model of computation (typically, t steps can be emulated within time $t \log t$). Similar considerations apply to space hierarchies, but here one talks about *space constructible function* and the emulation overhead is typically only linear. For simplicity, in case of multiple-tape Turing machines, we get:

Theorem 4.1 (sketch): *For any time-constructible function t, the class $\mathrm{DTIME}(t)$ is strictly contained in $\mathrm{DTIME}(\omega(t \log t))$. For any space-constructible function s, the class $\mathrm{DSPACE}(s)$ is strictly contained in $\mathrm{DSPACE}(\omega(s))$.*

The existence of functions that are not time (or space) constructible is the reason for so-called gap theorems. Typically, such theorems say that there exist functions f (which are certainly not time constructible) such that $\mathrm{DTIME}(f) = \mathrm{DTIME}(f^3)$ (or $\mathrm{DTIME}(f) = \mathrm{DTIME}(2^{2^f})$). The reason for this phenomena is that (for such a function f) there are no machines that run more than time f but less than time f^3 (or 2^{2^f}).

Lecture 5

Space Complexity

Space complexity is aimed to measure the amount of temporary storage required for a computational task. On one hand, we don't want to count the input and output themselves within the space of computation, but on the other hand we have to make sure that the input and output device cannot be abused to provide work space (which is uncounted for). This leads to the convention of postulating that the input device (e.g., a designated input-tape of a multi-tape Turing machine) is read-only, whereas the output device (e.g., a designated output-tape of a such machine) is write-only. Space complexity accounts for the amount of space on other (storage) devices (e.g., the work-tapes of a multi-tape Turing machine) that is used throughout the computation.

5.1 Deterministic space complexity

Only regular languages can be decided in constant space. This follows by combining two facts. Firstly, constant-space Turing machines are equivalent to a generalization of finite automata that can scan (parts of the) input back and forth (in both directions and for several times). Second, the latter "sweeping automata" can be simulated by ordinary finite automata (which scan the input only once, from left to right).

At first glance one may think that sub-logarithmic (deterministic) space is not more useful than constant space, because it *seems* impossible to allocate a sub-logarithmic amount of space (since measuring the input length requires logarithmic space). However, this intuition is wrong, because the input itself (in case it is of the proper form) can be used to determine its length, whereas in case the input is not of the proper form this fact may be detectable (within sub-logarithmic space). In fact:

Theorem 5.1 $\text{DSPACE}(o(\log n))$ *is a proper superset of* $\text{DSPACE}(O(1))$.

One proof consists of presenting a *double-logarithmic space* algorithm for recognizing the *non-regular* set $L = \{x_k : k \in \mathsf{N}\} \subset \{0, 1, *\}$, where x_k equals the concatenation of all k-bit long strings (in lexicographic order) separated by $*$'s (i.e., $x_k = 0^{k-2}00 * 0^{k-2}01 * 0^{k-2}10 * 0^{k-2}11 * \cdots * 1^k$). Note that $|x_k| > 2^k$, and we claim that x_k can be recognized in space $O(\log k) = O(\log\log |x_k|)$. Furthermore, the membership of any x in L can be determined in space $O(\log\log |x|)$, by iteratively checking (in space $O(\log i)$) whether $x = x_i$, for $i = 1, 2, ...$ (Details are left as an exercise.) In contrast to Theorem 5.1, double-logarithmic space is indeed the smallest amount of space that is more useful than constant space; that is:

Theorem 5.2 $\text{DSPACE}(o(\log\log n)) = \text{DSPACE}(O(1))$.

The proof proceeds by considering, for each input location, the sequence of (storage) configurations of the machine at all times that it crosses this input location. For starters, the length of this "crossing sequence" is upper-bounded by the number of possible storage configurations (i.e., in case of Turing machines, we consider the contents of the tape and the head location), which is at most $t \stackrel{\rm def}{=} 2^{s(n)} \cdot s(n)$, where s is the machine's space complexity. Thus, the number of such sequences is bounded above by t^t. But if the latter is smaller than $n/2$ then there exist three input locations that have the same sequence of configurations. Using cut-and-paste, we get a shorter input on which the machine used space $s' \stackrel{\rm def}{=} s(n)$, which is not possible in case the original n-bit long input was the shortest one on which the machine uses space at least s'. We conclude that $t^t \geq n/2$ must hold, and $s(n) = \Omega(\log t) = \Omega(\log \log n)$ follows.

Logarithmic Space. Although Theorem 5.1 asserts that "there is life under log-space", logarithmic space will be the lowest space-complexity class that we will care about. The class of sets recognizable by deterministic machines that use logarithmic space is denoted $\mathcal{L}$; that is, $\mathcal{L} \stackrel{\rm def}{=} \cup_c \textsc{Dspace}(c \log_2 n)$.

Theorem 5.3 *$\mathcal{L} \subseteq \mathcal{P}$.*

In general, if s is at least logarithmic and is computable within time 2^s then $\textsc{Dspace}(s) \subseteq \textsc{Dtime}(2^s)$. This follows as a special case from Theorem 5.6. (The phenomena that time relates exponentially to space occurs also in other settings.)

Another class of important log-space computations is the class of logarithmic space reductions; that is, reductions (or oracle machines) that use only logarithmic space (and as usual polynomial-time). In accordance with our conventions regarding input and outputs, we stress that the queries (resp., answers) are written on (resp., read from) a special device/tape that is write-only (resp., read-only) for the calling algorithm and read-only (resp., write-only) for the invoked oracle. We observe that all known (Karp-)reductions establishing NP-completeness results are in fact logarithmic space. Observe that if L' is log-space reducible to L'' and $L'' \in \mathcal{L}$ then so is L'. (See Section 5.2.3.)

Polynomial Space. As stated above, we will rarely treat computational problems that require less than logarithmic space. On the other hand, we will rarely treat computational problems that require more than polynomial space. The class of decision problems that are solvable in polynomial-space is denoted $\mathcal{PSPACE} \stackrel{\rm def}{=} \cup_c \textsc{Dspace}(n^c)$. A complete problem for $\mathcal{PSPACE}$ is presented in Section 6.5.

5.2 Non-deterministic space complexity

5.2.1 Two models of non-determinism

We discuss two models of non-deterministic machines. In the standard model, called the on-line model, the machine makes non-deterministic "on the fly" (or alternatively reads a non-deterministic input from a read-only tape *that can be read only in a uni-directional way*). Thus, if the machines wants to refer to such a non-deterministic choice at a latter stage then it must store the choice on its storage device (and be charged for it). In the so-called off-line model, the non-deterministic choices (or the bits of the non-deterministic input) are read from a special read-only record (or tape) *that can be scanned in both directions like the main input.* Although the off-line model fits better the motivations to $\mathcal{NP}$ (as presented in the first lecture), the on-line model seems more adequate for the study of non-deterministic in the context of space complexity. The latter thesis is based on observing that an off-line non-deterministic tape can be used to code computations, and in a sense allows to "cheat" with respect to the "real" space complexity of the computation. This is reflected in the fact that the off-line model can simulate the on-line model while using space that is logarithmic in the space used by the on-line model.[1] This result is tight: the on-line model can simulate the off-line model using (only) exponentially more space.

Theorem 5.4 (relating the two models, loosely stated:) *For $s : \mathsf{N} \to \mathsf{N}$ that is nice and at least logarithmic,* $\textsc{Nspace}_{\text{on-line}}(s) = \textsc{Nspace}_{\text{off-line}}(\log s)$.

To simulate the on-line model on the off-line model, use the non-deterministic input tape of the latter to encode an accepting computation of the former (i.e., a sequence of consecutive configurations leading from the initial configuration to an accepting configuration). The simulating machine (which verifies the legitimacy of the sequence of configurations recorded on the non-deterministic input tape) needs only store its *location within the current pair of configurations* that it examines, which requires space logarithmic in the length of a single configuration. On the other hand, the simulation of the off-line model by the on-line model uses a crossing-sequence argument. For starters, one shows that the length of such sequences is at most double-exponential in the space complexity of the off-line machine. Then the (on-line) non-deterministic input tape is used to encode the sequence of crossing-sequences, and the on-line machine checks that each consecutive pair is consistent. This requires holding one (or two) crossing-sequences in storage, which require space logarithmic in the number of such sequences (which, in turn, is exponential in the space complexity of the off-line machine).

5.2.2 Some basic facts about NSPACE

We let $\textsc{Nspace}(s) \stackrel{\text{def}}{=} \textsc{Nspace}_{\text{on-line}}(s)$, and focus on $\mathcal{NL} \stackrel{\text{def}}{=} \textsc{Nspace}(O(\log n))$. Suitable *upwards-translation lemmas* can be used to translate simulation results concerning $\mathcal{NL}$ (resp., $\mathcal{L}$) into general simulation results concerning non-deterministic (resp., deterministic) space. Typically, the input is padded till the concrete space allowance becomes logarithmic in the padded input (i.e., n-bit long inputs are padded to length N such that $s(n) = \log N$). Next the simulation result is applied, and finally the complexity of the obtained simulation is stated in terms of the original input length. A notable property of $\mathcal{NL}$ is that this class has a very natural complete problem:

Theorem 5.5 (Directed Connectivity is NL-complete:) *Directed Connectivity is in $\mathcal{NL}$, and every problem in $\mathcal{NL}$ is reducible to Directed Connectivity by a log-space reduction.*

Proof Sketch: A non-deterministic log-space machine may decide Directed Connectivity by guessing (and verifying) the directed path "on-the-fly". To reduce $L \in \mathcal{NL}$ to Directed Connectivity, we consider the non-deterministic log-space machine that decides L. We observe that on input x, this machine uses $\ell \stackrel{\text{def}}{=} O(\log |x|)$ space, and it may be in one out of $2^\ell \cdot |x| \cdot \ell$ possible configurations (accounting for the possible contents of its work-tape and its head locations (on the input tape and work tapes)). Consider a directed graph with these configuration as vertices, and directed edges connecting ordered pairs of possibly-consecutive configurations (relating to a possible non-deterministic move). Indeed, unlike the vertices, the edges depend on the input x. Observe that

[1] A related phenomenon is that $\textsc{Nspace}_{\text{off-line}}(s)$ is only known to be contained in $\textsc{Dtime}(2^{2^s})$, whereas (as stated in Theorem 5.6) $\textsc{Nspace}_{\text{on-line}}(s) \subseteq \textsc{Dtime}(2^s)$. In fact, the power of the off-line model emerges from the fact that its running time is not bounded (even not "without loss of generality") by an exponent in the space-complexity.

$x \in L$ if and only if there exists a directed path in this graph leading from the initial configuration to an accepting configuration. Furthermore, this graph can be constructed in logarithmic space (from the input x). Thus, L is log-space reducible to Directed Connectivity. ■

Theorem 5.6 (non-deterministic space versus deterministic time): *If s is at least logarithmic and is computable within time 2^s then* $\text{NSPACE}(s) \subseteq \text{DTIME}(2^s)$.

Proof Sketch: By a suitable upwards-translation lemma, it suffices to prove the result for logarithmic s; that is, we need to show that $\mathcal{NL} \subseteq \mathcal{P}$. Using Theorem 5.5, we just need to show that directed connectivity can be solved in polynomial-time. This fact is well known (e.g., by the directed-DFS algorithm). ■

Theorem 5.7 (Non-deterministic versus deterministic space): $\text{NSPACE}(s) \subseteq \text{DSPACE}(s^2)$ *provided that $s : \mathsf{N} \to \mathsf{N}$ is space-constructible and at least logarithmic.*

In particular, for any polynomial p, it holds that $\text{NSPACE}(p) \subset \mathcal{PSPACE}$, where the strict inclusion is due to the space hierarchy theorem (e.g., $\text{DSPACE}(n^c) \subset \text{DSPACE}(n^{c+1})$). Contrast Theorem 5.7 with the (trivial) fact that $\text{NTIME}(t) \subseteq \text{DTIME}(2^t)$, provided that $t : \mathsf{N} \to \mathsf{N}$ is time-constructible (and at least logarithmic).

Proof Sketch: Again, it suffices to show that directed connectivity can be solved in deterministic $O(\log n)^2$ space. The basic idea is that checking whether or not there is a path of length at most ℓ from u to v in G, reduces (in log-space) to checking whether there is an intermediate vertex w such that there is a path of length at most $\lceil \ell/2 \rceil$ from u to w and a path of length at most $\lfloor \ell/2 \rfloor$ from w to v. Let $p_G(u, v, \ell) \stackrel{\text{def}}{=} 1$ if there is a path of length at most ℓ from u to v in G and $p_G(u, v, \ell) \stackrel{\text{def}}{=} 0$ otherwise. Thus, $p_G(u, v, \ell)$ can be decided recursively by scanning all vertices w in G, and checking for each w whether both $p_G(u, w, \lceil \ell/2 \rceil) = 1$ both $p_G(w, v, \lfloor \ell/2 \rfloor) = 1$ hold.

Thus, suppose we are given a directed graph G and a pair of vertices (s, t), and should decide whether or not there is a path from s to t in G. Let n denote the number of vertices in G, then we need to compute $p_G(s, t, n)$. This is done by invoking a recursive procedure that computes $p_G(u, v, \ell)$ by scanning all vertices in G, and computing for each vertex w the value of $p_G(u, w, \lceil \ell/2 \rceil) \cdot p_G(w, v, \lfloor \ell/2 \rfloor)$. The amount of space taken by each level of the recursion is $\log n$ (for storing the current value of w), and the number of levels is $\log n$. The theorem follows.

We stress that when computing $p_G(u, v, \ell)$, we make polynomially many recursive calls, but all these calls re-use the same work space. That is, when we compute $p_G(u, w, \lceil \ell/2 \rceil) \cdot p_G(w, v, \lfloor \ell/2 \rfloor)$ we re-use the space that was used for computing $p_G(u, w', \lceil \ell/2 \rceil) \cdot p_G(w', v, \lfloor \ell/2 \rfloor)$ (for the previous w'). Furthermore, when we compute $p_G(w, v, \lfloor \ell/2 \rfloor)$ we re-use the space that was used for computing $p_G(u, w, \lceil \ell/2 \rceil)$. ■

5.2.3 Composition Lemmas

Indeed, as indicated by the proof of Theorem 5.7, space (unlike time!) can be re-used. In particular, if one machine makes many recursive calls to another machine then the cost in space of these calls is the *maximum space used by a single call* (whereas the cost in terms of time of these calls is the *sum of the time taken by all calls*). Put in other words: *Suppose that L_1 is s_1-space reducible to L_2 and that L_2 is in $XSPACE(s_2)$, where $X \in \{D, N\}$. Then L_1 is in $XSPACE(s_1 + s_2')$,* where $s_2'(n) = s_2(2^{s_1(n)})$ (because $2^{s_1(n)}$ is an obvious bound on the length of queries made to L_2). Proving this claim is less trivial than it seems (even in case of a single call to L_2) because we cannot afford to store the query and the answer (which may have lengths 2^{s_1} and $2^{s_2 \circ s_1}$, respectively) in the working space of the resulting machine.

For simplicity, we focus on the single-query case.[2] Let M_1 be the reduction of L_1 to L_2 and M_2 a machine solving L_2. We emulate them both as follows. We allocate each of the machines a separate work-tape, and begin by emulating M_2 without specifying its input. When M_2 wishes to read the i^{th} (e.g., first) bit of its input (which is the i^{th} bit of the query of M_1), we run machine M_1 until it produces the i^{th} bit of its query, which we hand to M_2. We stress that we do not store all previous bits of this query, but rather discard them. Thus, we run a new emulation of M_1, per each time that M_2 wishes to read a bit of its input (i.e., the query directed to it by M_1). When M_2 outputs its decision, we store it, and emulate M_1 for the last time. In this run we discard all the query bits produced by M_1, feed it with M_2's answer, and output whatever M_2 does.

The treatment of reductions to search problems is more complex, because (unless postulated differently) the calling algorithm may scan the answer provided by the oracle back and forth (rather than read it once from left to right). To treat this case we may keep two emulations of M_1, one for producing bits of the query and the other for using the bits of the answer. (Note that the second emulation corresponds to the last emulation of M_1 in the description above.) Handling of many oracle calls is performed in a query by query manner, relying of the fact that the i^{th} answer is not available after the $i+1^{\text{st}}$ query is made. For $i = 1, 2, ...$, we handle the $i+1^{\text{st}}$ query-answer by keeping a record of the temporary configurations of M_1 before it started making the i^{th} and $i+1^{\text{st}}$ queries. We maintain four emulations of M_1, the first (resp., third) for producing bits of the i^{th} (resp., $i+1^{\text{st}}$) query, and the second (resp., fourth) for using the bits of the i^{th} (resp., $i+1^{\text{st}}$) answer. Each time we need to emulate the first or second (resp., third or fourth) copy, we *start the emulation from the recorded configuration* of M_1 before making the i^{th} (resp., $i+1^{\text{st}}$) query. Once the fourth copy starts to produce the $i+2^{\text{th}}$ query, we refresh all configurations and move to the next iteration. Specifically, the configuration of the fourth copy will be used as the second temporary configuration (as it corresponds to the configuration before making the $i+2^{\text{th}}$ query), and the current second configuration (which corresponds to the configuration before making the $i+1^{\text{st}}$ query) will be used as the first temporary configuration (for iteration $i+1$).

Teaching Note. In the next subsection we will (implicitly) use a composition result, but for that specific composition we do not need the power of the above strong composition lemma. Specifically, the reduction will make queries that are very related to its input (and thus the invoked subroutine can form the query by itself from the input). Furthermore, the answers will be of logarithmic length and thus can be stored by the reduction (as in case of invoking a decision subroutine).

5.2.4 NSPACE is closed under complementation

People tend to be discouraged by the impression that "decades of research have failed to answer any of the famous open problems of complexity theory". In addition to the fact that substantial progress towards the understanding of some open problems has been achieved, people tend to forget that some famous open problems were indeed resolved. The following result relates to a famous question that was open for three decades.[3]

[2] We leave the extension to the general multiple-query case as an exercise.

[3] In particular, using the fact that the class of sets recognized by linear-space non-deterministic machines equals the set of context-sensitive languages, Theorem 5.8 resolves the question of whether the latter set is closed under complementation. This question has been puzzling researchers since the early days of research in the area of formal languages (i.e., the 1950's). We mention that Theorem 5.8 was proven in the late 1980's.

Theorem 5.8 *$\mathcal{NL} = \text{co}\mathcal{NL}$, where $\text{co}\mathcal{NL} \stackrel{\text{def}}{=} \{\{0,1\}^* \setminus L : L \in \mathcal{NL}\}$.*

Again, using an adequate upwards-translation lemma, one can derive the closure under complementation of NSPACE(s).

Proof Sketch: It suffices to show that *directed unconnectivity* (the complementation of directed connectivity) can be decided in $\mathcal{NL}$. That is, we will present a non-deterministic log-space machine M such that

- If there is *no directed path* from s to t in G then there exists a computation of M that accepts the input (G, s, t).
- If *there is a directed path* from s to t in G then all possible computations of M reject (G, s, t).

The above decision problem is log-space reducible to determining the number of nodes that are reachable from a given vertex in a given graph.[4] Thus, we focus on providing a non-deterministic log-space machine that compute the said quantity, where we say that a non-deterministic M computes the function $f : \{0,1\}^* \to \{0,1\}^*$ if the following two conditions hold:

1. For every x, there exists a computation of M that halts with output $f(x)$.
2. For every x, all possible computation of M either halt with output $f(x)$ or halt with a special "dont know" symbol, denoted $\perp$.

Fixing an n-vertex graph $G = (V, E)$ and a vertex v, we consider the set of vertices that are reachable from v by a path of length at most i. We denote this set by R_i, and observe that $R_0 = \{v\}$ and that for every $i = 1, 2, ...$, it holds that

$$R_i = R_{i-1} \cup \{u : \exists w \in R_{i-1} \text{ s.t. } (w, u) \in E\} \tag{5.1}$$

Our aim is to compute $|R_n|$. This will be done in n iterations such that at the i^{th} iteration we compute $|R_i|$. When computing $|R_i|$ we rely on the fact that $|R_{i-1}|$ is known to us, which means that we'll store $|R_{i-1}|$ (but not previous $|R_j|$'s) in memory. Our non-deterministic guess, denoted g, for $|R_i|$ will be verified as follows:

- $|R_i| \geq g$ is verified in the straightforward manner. That is, scanning V, we guess for g vertices paths of length at most i from v to them, and verify these "on-the-fly". (Indeed, we also guess for which g vertices to verify this fact.)

  (We use $\log_2 n$ bits to store the currently scanned vertex, another $\log_2 n$ bits to store an intermediate vertex on a path from v, and another $\log_2 i \leq \log_2 n$ bits to store the distance traveled so far.)

- The verification of $|R_i| \leq g$ is the interesting part of the procedure. Here we rely on the fact that we know $|R_{i-1}|$. Scanning V (again), we verify for $n - g$ (guessed) vertices that they are **not** reachable from v by paths of length at most i. Verifying that $u \notin R_i$ is done as follows.

  - We scan V guessing $|R_{i-1}|$ vertices that are in R_{i-1}, and verify each such guess in the straightforward manner. (Implicit here is a procedure that given G, v, i and $|R_{i-1}|$, produces R_{i-1} itself.)

[4] Exercise: provide such a reduction.

– For each $w \in R_{i-1}$, which was guessed and verified above, we verify that both $u \neq w$ and $(w,u) \notin E$.

By Eq. (5.1), if u passes the above verification then indeed $u \notin R_i$.

(We use $\log_2 n$ bits to store u, another $\log_2 n$ bits to count the number of vertices verified to be in R_{i-1}, another $\log_2 n$ bits to store such w, and another $2\log_2 n$ bits for verifying that $w \in R_{i-1}$.)

If any of the verifications fails, the machine halts outputting the "dont know" symbol. Exercise: assuming that the correct value of $|R_{i-1}|$ is used, prove that the above non-deterministic log-space procedure computes the value of $|R_i|$.

Observing that when computing $|R_i|$ we only need to know $|R_{i-1}|$ (and do not need $|R_j|$ for any $j < i-1$), the above yields a non-deterministic log-space machine for computing $|R_n|$. The theorem follows. ∎

Lecture 6

The Polynomial-Time Hierarchy

The Polynomial-Time Hierarchy (PH) is a hierarchy of complexity classes that extends $\mathcal{NP}$. We will present two equivalent ways of defining this hierarchy, and discuss some of its properties.

6.1 Defining PH via quantifiers

Recall that $L \in \mathcal{NP}$ if there exists a *binary* (polynomially-bounded) relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \exists y \text{ s.t. } (x, y) \in R \tag{6.1}$$

Identifying $\mathcal{NP}$ with Σ_1, we define Σ_2 as containing sets L such that there exists a *3-ary* (polynomially-bounded)[1] relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \exists y_1 \forall y_2 \text{ s.t. } (x, y_1, y_2) \in R \tag{6.2}$$

(Above and below, it is important to stress that the universal quantifiers range only over strings of the adequate length.)

In general, Σ_i is defined as the class consisting of sets L such that there exists a $(i+1)$-*ary* (polynomially-bounded)[2] relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \exists y_1 \forall y_2 \cdots Q_i y_i \text{ s.t. } (x, y_1, ..., y_i) \in R \tag{6.3}$$

where Q_i is an existential (resp., universal) quantifier in case i is odd (resp., even). That is, we have i alternating quantifiers, starting with an existential one, where each quantifier ranges over strings of length polynomial in the length of x. (Note that indeed $\Sigma_1 = \mathcal{NP}$.)

Similarly, we can define classes referring to alternating sequences of quantifiers starting with a universal quantifier. Specifically, $L \in \Pi_i$ if there exists a $(i+1)$-*ary* (polynomially-bounded) relation R such that R is polynomial-time recognizable and

$$x \in L \text{ if and only if } \forall y_1 \exists y_2 \cdots Q_i y_i \text{ s.t. } (x, y_1, ..., y_i) \in R \tag{6.4}$$

where Q_i is an existential (resp., universal) quantifier in case i is even (resp., odd).

The polynomial-time hierarchy, denoted $\mathcal{PH}$, is defined as $\cup_i \Sigma_i$. That is, $L \in \mathcal{PH}$ means that there exists an i such that $L \in \mathcal{PH}$.

[1] We say that R is polynomially-bounded if there exists a polynomial p such that for every $(x, y_1, y_2) \in R$ it holds that $|y_1| + |y_2| \leq p(|x|)$.

[2] Indeed, we say that R is polynomially-bounded if there exists a polynomial p such that for every $(x, y_1, ..., y_i) \in R$ it holds that $\sum_{j=1}^{i} |y_j| \leq p(|x|)$.

Exercises: the following facts can be verified by purely syntactic considerations:

1. For every $i \geq 1$, it holds that $\Pi_i = \text{co}\Sigma_i$ (and, in particular, $\Pi_1 = \text{co}\mathcal{NP}$).
2. For every $i \geq 1$, it holds that $\Sigma_i \subseteq \Pi_{i+1}$ and $\Pi_i \subseteq \Sigma_{i+1}$. Thus, $\mathcal{PH} = \cup_i \Pi_i$.
3. For every $i \geq 1$, it holds that $\Sigma_i \subseteq \Sigma_{i+1}$.

It is widely believed that Σ_i is a strict subset of Σ_{i+1}. See further discussion in Section 6.4.

Another Exercise: Prove that $\mathcal{PH}$ is contained in $\mathcal{PSPACE}$. See further discussion in Section 6.5.

Complete sets in PH: The above definition of Σ_i (and Π_i) gives rise to ("semi-natural") complete sets for Σ_i (and Π_i). For example, consider the set of Boolean circuits of the form C such that C takes as input i equal-length strings, denoted $x_1, ..., x_i$, and it holds that $\exists x_1 \forall x_2 \cdots Q_i x_i \, C(x_1, ..., x_i) = 1$, where Q_i is an existential (resp., universal) quantifier in case i is odd (resp., even). Clearly, this set is in Σ_i, and every set in Σ_i is Karp-reducible to this set.[3] (Hint: the x_i's correspond to the y_i's in the definition of Σ_i, whereas C corresponds to x.)

Natural Examples of sets in PH: Recall that natural NP-optimization problems are captured by NP-sets that refer only to a (one-sided) bound on the value of the optimum. For example, whereas the optimization version of `maxClique` requires to find the largest clique in a given graph, the decision problem is to tell whether or not the largest clique has size *greater than or equal to* a given number.[4] Clearly, the latter decision problem is in $\mathcal{NP}$, whereas its complement (i.e., determining whether the largest clique has size *smaller than* a given number) is in co$\mathcal{NP}$. But what about determining whether the largest clique has size *equal to* a given number? (That is, the set we refer to is the set of pairs (G, K) such that the size of the largest clique in G equals K.) Note that this problem is unlikely to be in either $\mathcal{NP}$ or co$\mathcal{NP}$ (because this will imply $\mathcal{NP} = \text{co}\mathcal{NP}$),[5] but it is certainly in Σ_2 (and in Π_2). (Exercise: Present adequate 3-ary relations for the above set.) See further discussion in the next section.

6.2 Defining PH via oracles

Recall that the general notion of a reduction is based on augmenting a (deterministic) polynomial-time machine with oracle access. A natural question is what languages can be recognized by such machines when the oracle is an arbitrary NP-set (or, equivalently an NP-complete set like SAT).[6] We denote this class by $\mathcal{P}^{\mathcal{NP}}$, standing for an arbitrary "P-machine" given oracle access to some NP-set. (As indicated below, $\mathcal{P}^{\mathcal{NP}}$ is likely to be a proper superset of $\mathcal{NP}$, whereas the class of languages that are Karp-reducible to $\mathcal{NP}$ equals $\mathcal{NP}$.)

[3] The reason that we insist on Karp-reductions here will become clear below.

[4] Actually, the decision problem is typically phrased as determining whether there exists a clique of size greater than or equal to a given number.

[5] If we could have given an NP-proof that the max-clique has size equal to a given number, then we could prove that it is strictly smaller than a given number, which is a coNP-complete problem (and $\mathcal{NP} = \text{co}\mathcal{NP}$ would follow).

[6] Exercise: Show that these two formulations are indeed equivalent.

Comment: The notation $\mathcal{P}^{\mathcal{NP}}$ is consistent with the standard notation for oracle machines. That is, for an oracle machine M, (oracle) set L and string x, we let $M^L(x)$ denote the output of M on input x and oracle access to L. (Thus, when we said that L' is reduced to L we meant that there exists a polynomial-time oracle machine M such that for every x it holds that $M^L(x) = 1$ if and only if $x \in L'$.) Thus,

$$\mathcal{P}^{\mathcal{NP}} = \{L(M^{SAT}) : M \text{ is a "P-machine"}\}$$

where $L(M^{SAT})$ denotes the set of inputs that are accepted by M when given oracle access to SAT.

Exercises:

1. Show that both $\mathcal{NP}$ and co$\mathcal{NP}$ are subsets of $\mathcal{P}^{\mathcal{NP}}$.
2. In contrast, prove that the class of languages that are Karp-reducible to $\mathcal{NP}$ equals $\mathcal{NP}$.
3. Following the above discussion, define $\mathcal{P}^{\text{co}\mathcal{NP}}$ and show that it equals $\mathcal{P}^{\mathcal{NP}}$.
4. Referring to the set of pairs (G, K) such that the size of the largest clique in G equals K, show that this set is in $\mathcal{P}^{\mathcal{NP}}$.

The definition of $\mathcal{P}^{\mathcal{NP}}$ suggests that we may define also classes such as $\mathcal{NP}^{\mathcal{NP}}$. Note that such a definition does not yield a natural notion of a reduction (to NP-sets), because the "reduction" is non-deterministic. Still, a well-defined class does emerge. Specifically, $\mathcal{NP}^{\mathcal{NP}}$ is the class of sets that are accepted by a non-deterministic polynomial-time oracle machine that is given access to some NP-set. Observe that indeed $\mathcal{P}^{\mathcal{NP}} \subseteq \mathcal{NP}^{\mathcal{NP}}$.

Defining Σ_i by oracles: As before, we let $\Sigma_1 \stackrel{\text{def}}{=} \mathcal{NP}$. For $i \geq 1$, we define

$$\Sigma_{i+1} \stackrel{\text{def}}{=} \mathcal{NP}^{\Sigma_i} . \tag{6.5}$$

Indeed, Σ_2 so defined equals $\mathcal{NP}^{\mathcal{NP}}$. As we will show in the next section, the Σ_i's as defined here coincide with the classes defined in the previous section.

A general perspective – what does ${\mathcal{C}_1}^{\mathcal{C}_2}$ mean? By the above discussion it should be clear that the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ can be defined for any two complexity classes $\mathcal{C}_1$ and $\mathcal{C}_2$, *provided that $\mathcal{C}_1$ is associated with a class of machines that extends naturally to access oracles.* Actually, the class ${\mathcal{C}_1}^{\mathcal{C}_2}$ *is not defined based on the class $\mathcal{C}_1$ but rather by analogy to it.* Specifically, suppose that $\mathcal{C}_1$ is the class of sets recognizable by machines of certain type (e.g., deterministic or non-deterministic) with certain resource bounds (e.g., time and/or space bounds). Then we consider analogous oracle machines (i.e., of the same type and with the same resource bounds), and say that $L \in {\mathcal{C}_1}^{\mathcal{C}_2}$ if there exists such an oracle machine M_1 and a set $L_2 \in \mathcal{C}_2$ such that $M_1^{L_2}$ accepts the set L.

Exercise: For $\mathcal{C}_1$ and $\mathcal{C}_2$ as above, prove that ${\mathcal{C}_1}^{\mathcal{C}_2} = {\mathcal{C}_1}^{\text{co}\mathcal{C}_2}$. Note that, in particular, $\mathcal{NP}^{\Sigma_i} = \mathcal{NP}^{\Pi_i}$.

6.3 Equivalence of the two definitions of PH

To avoid confusion, let use denote by Σ'_i the class defined via quantifiers (i.e., in Eq. (6.3)), and by Σ''_i the class defined by oracle machines (i.e., in Eq. (6.5)).

Theorem 6.1 *For every $i \geq 1$, it holds that $\Sigma'_i = \Sigma''_i$.*

Proof Sketch: The claim holds trivially for $i = 1$, Assuming that equality holds for $i \geq 1$, we show that it holds also for $i+1$. (Each of the two inclusions uses only the induction hypothesis of the same direction.)

Assuming that $\Sigma'_i \subseteq \Sigma''_i$, we prove that $\Sigma'_{i+1} \subseteq \Sigma''_{i+1}$, by looking at an $(i+2)$-ary relation, R, for a set $L \in \Sigma'_{i+1}$. Recall that $x \in L$ iff $\exists y_1 \forall y_2 \cdots Q_{i+1} y_{i+1}$ such that $(x, y_1, ..., y_{i+1}) \in R$. Define L' as the set of pairs (x, y) such that $\forall y_2 \cdots Q_{i+1} y_{i+1}$ it holds that $(x, y, y_2, ..., y_{i+1}) \in R$. Then, $L' \in \mathrm{co}\Sigma'_i$ and $x \in L$ iff there exists a y such that $(x, y) \in L'$. By using a straightforward non-deterministic oracle machine, we obtain that $L \in \mathcal{NP}^{\mathrm{co}\Sigma'_i} = \mathcal{NP}^{\Sigma'_i}$. Using the induction hypothesis, it follows that $L \in \mathcal{NP}^{\Sigma''_i} = \Sigma''_{i+1}$.

Assuming that $\Sigma''_i \subseteq \Sigma'_i$, we prove that $\Sigma''_{i+1} \subseteq \Sigma'_{i+1}$, by looking at a non-deterministic oracle machine M that accepts a set $L \in \Sigma''_{i+1}$ when using an oracle $L' \in \Sigma''_i$. By the definition of non-uniform acceptance, it follows that $x \in L$ iff there exists a computation of M on input x that accepts, when the queries are answered according to L'. Let use denote by $M^{L'}(x, y)$ the output of M on input x and non-deterministic choices y, when its queries are answered by L'. Then, $x \in L$ iff there exists a y such that $M^{L'}(x, y) = 1$. We may assume, without loss of generality, that M starts its computation by non-deterministically guessing all oracle answers (and acting according to these guesses), and that it accept only if these guesses turned out to be correct. In other words, there exists a polynomial-time computable predicate P, such that $M^{L'}(x, y) = 1$ iff $P(x, y) = 1$ and the j^{th} answer provided by the oracle in the computation $M^{L'}(x, y)$ equals the j^{th} bit of y, denoted $y^{(j)}$. Furthermore, since M acts according to the guessed answers that are part of y, the j^{th} query of M is determined (in polynomial-time) by (x, y), and is denoted $q^{(j)}(x, y)$. We conclude that $x \in L$ iff there exists a y such that $P(x, y) = 1$ and $y^{(j)} = 1$ iff $q^{(j)} \in L'$ for every j. Using the induction hypothesis, it holds that $L' \in \Sigma''_i = \Sigma'_i$, and we let R' denote the corresponding $(i+1)$-ary relation. Thus, $x \in L$ iff

$$\exists y \left((P(x,y)=1) \wedge \bigwedge_j \left(((y^{(j)}=1) \Leftrightarrow \exists y_2^{(j)} \forall y_3^{(j)} \cdots Q_{i+1} y_{i+1}^{(j)} \, (q^{(j)}(x,y), y_2^{(j)}, ..., y_{i+1}^{(j)}) \in R' \right) \right)$$

The proof is completed by observing that the above expression can be rearranged to fit the definition of Σ'_{i+1}. (Hint: we may incorporate the computation of all the $q^{(j)}(x, y)$'s into the relation R', and pull all quantifiers outside.)[7] ■

6.4 Collapses

As stated before, it is widely believed that $\mathcal{PH}$ is a strict hierarchy; that is, that Σ_i is strictly contained in Σ_{i+1} for every $i \geq 1$. We note that if a collapse occurs at some level (i.e., $\Sigma_i = \Sigma_{i+1}$

[7] Note that, for predicates P_1 and P_2, the expression $\exists y \, (P_1(y) \Leftrightarrow \exists z P_2(y, z))$ is equivalent to the expression $\exists y \, ((\neg P_1(y) \vee \exists z P_2(y, z)) \wedge ((P_1(y) \vee \neg \exists z P_2(y, z)))$, which in turn is equivalent to the expression $\exists y \exists z' \forall z'' \, ((\neg P_1(y) \vee P_2(y, z')) \wedge ((P_1(y) \vee \neg P_2(y, z'')))$. Note that pulling the quantifiers outside in $\wedge_{j=1}^t \exists y^{(j)} \forall z^{(j)} P(y^{(j)}, z^{(j)})$ yields an expression of the type $\exists y^{(1)}, ..., y^{(t)} \forall z^{(1)}, ..., z^{(t)} \wedge_{j=1}^t P(y^{(j)}, z^{(j)})$.

for some $i \geq 1$) then the entire hierarchy collapses to that level (i.e., $\Sigma_i = \mathcal{PH}$). (This fact is best verified from the oracle-based definition, and the verification is left as an exercise.)[8] In fact, a stronger statement can be proven.

Theorem 6.2 *If $\Sigma_i = \Pi_i$ holds for some $i \geq 1$ then $\mathcal{PH} = \Sigma_i$.*

In particular, $\mathcal{NP} = \text{co}\mathcal{NP}$ implies a total collapse (i.e., $\mathcal{PH} = \mathcal{NP}$). In light of the above discussion, it suffices to show that $\Sigma_i = \Pi_i$ implies $\Sigma_i = \Sigma_{i+1}$. This is easiest to prove using the quantifier-based definition, while relying on ideas used in the previous section. Specifically, for $L \in \Sigma_{i+1}$, we first derive a set $L' \in \Pi_i$ such that $x \in L$ if and only if there exists y such that $(x, y) \in L'$. By the hypothesis $L' \in \Sigma_i$, and so $x \in L$ iff

$$\exists y \exists y_1 \forall y_2 \cdots Q_i y_i \text{ s.t. } ((x, y), y_1, y_2, ..., y_i) \in R'$$

where R' is the $(i+1)$-ary relation guaranteed for L' (w.r.t the definition of Σ_i). By joining the two leftmost existential quantifiers and slightly modifying R' (into $R'' \stackrel{\text{def}}{=} \{(x, (y, y_1), y_2, ..., y_i) : ((x, y), y_1, y_2, ..., y_i) \in R'\}$), we conclude that $L \in \Sigma_i$. ∎

6.5 Comment: a PSPACE-complete problem

Recall that the complete problem of Σ_i referred to circuits that take i input strings (and to an alternating existential and universal quantification over these inputs). A natural question that arises is what happens if we drop the restriction on the number of such inputs. That is, consider the set of circuits that take a sequence of input strings, which is (of course) bounded in length by the size of the circuit. Such a circuit, denoted C, having $t = t(G)$ input strings, denoted $x_1, ..., x_t$, is in the set QC (standing for Quantified Circuits) if and only if $\exists x_1 \forall x_2 \cdots Q_t x_t \, C(x_1, ..., x_t) = 1$.

It is easy to see that $QC \in \mathcal{PSPACE}$. To show that any problem in $\mathcal{PSPACE}$ is reducible (in fact, Karp-reducible) to QC, we follow the underlying idea of the proof of Theorem 5.7. That is, let $L \in \mathcal{PSPACE}$, let M be the corresponding polynomial-space machine, and p be the corresponding polynomial space-bound. For any $x \in \{0,1\}^n$, it holds that $x \in L$ iff M passes in at most $2^{p(n)}$ steps from the initial configuration with input x, denoted $\texttt{init}(x)$, to an accepting configuration, denoted $\texttt{ACC}$. Define a Boolean predicate p_M such that $p_M(\alpha, \beta, t) = \texttt{true}$ iff M passes in at most t steps from the configuration α to configuration β. Then, we are interested in the value $p_M(\texttt{init}(x), \texttt{ACC}, 2^{p(n)})$. On the other hand, for every α and β (and $i \in \mathsf{N}$), it holds that

$$p_M(\alpha, \beta, 2^i) = \exists \gamma \, [p_M(\alpha, \gamma, 2^{i-1}) \wedge p_M(\gamma, \beta, 2^{i-1})] \tag{6.6}$$

If we were to iterate Eq. (6.6) then the length of the formula will double in each iteration, and after $\log_2 t$ iterations we'll just get a straightforward conjunction of t formulae capturing single steps of M. Our aim is to moderate the growth of the formula size during the iterations. Towards this end, we replace Eq. (6.6) by

$$\begin{aligned} p_M(\alpha, \beta, 2^i) \;\; = \;\; & \exists \gamma \forall \sigma \exists \alpha' \beta' \, [(\sigma = 0 \rightarrow (\alpha' = \alpha \wedge \beta' = \gamma)) \wedge (\sigma = 1 \rightarrow (\alpha' = \gamma \wedge \beta' = \beta)) \\ & \wedge \, p_M(\alpha', \beta', 2^{i-1}))] \end{aligned} \tag{6.7}$$

where $\sigma \in \{0, 1\}$. Observe that Eq. (6.6) is equivalent to Eq. (6.7), whereas in the latter the size of the formula grows by an additive term (rather than by a factor of 2). Thus, $p_M(\texttt{init}(x), \texttt{ACC}, 2^{p(n)})$

[8]Assuming that $\Sigma_i = \Sigma_{i+1}$, we prove by induction on $j > i$ that $\Sigma_j = \Sigma_i$. In the induction step, we have $\Sigma_{j+1} = \mathcal{NP}^{\Sigma_j} = \mathcal{NP}^{\Sigma_i} = \Sigma_{i+1} = \Sigma_i$.

can be written as a *quantified boolean formula* with $O(\log t)$ (alternating) quantifiers. The formula being quantified over will be a conjunction of ($O(\log t)$) simple logical conditions (of the type introduced in Eq. (6.7)) as well as (a single occurrence of) the formula $p_M(\cdot,\cdot,1)$. Hence, we have actually established the PSPACE-hardness of a special case of QC corresponding to Quantified Boolean Formulae, denoted QBF.

Lecture 7

Randomized Complexity Classes

So far, our approach to computing devises was somewhat conservative: we thought of them as (repeatedly) executing a deterministic rule. A more liberal and quite realistic approach pursued in this lecture considers computing devices that use a probabilistic (or randomized) rule. Specifically, we allow probabilistic rules that choose uniformly among two predetermined possibilities, and observe that the effect of more general probabilistic rules can be efficiently approximated by a rule of the former type. We still focus on polynomial-time computations, but these are *probabilistic* polynomial-time computations. Indeed, we extend our notion of *efficient computations* from *deterministic* polynomial-time computations to *probabilistic* polynomial-time computations.

Rigorous models of probabilistic machines are defined by natural extensions of the basic model; for example, we will talk of probabilistic Turing machines. Again, the specific choice of model is immaterial; as long as it is "reasonable". We consider the output distribution of such probabilistic machines on fixed inputs; that is, for a probabilistic machine M and string $x \in \{0,1\}^*$, we denote by $M(x)$ the distribution of the output of M on input x, where the probability is taken over the machine's random moves. Focusing on decision problems, three natural types of machines arise:

1. The most liberal notion is of machines with *two-sided error probability*. In case of search problems, it is required that the correct answer is output with probability that is significantly greater than 1/2 (e.g., probability at least 2/3). When this approach is applied to decision problems (solvable by probabilistic polynomial-time machines), we get the class $\mathcal{BPP}$, standing for Bounded-error Probabilistic Polynomial-time.

2. Machines with *one-sided error probability*: In case of search problems, a natural notion is of machines that output a (correct) solution (in case such exists) with probability at least 1/2, and never output a wrong solution. In case of decision problems, there are two natural cases depending on whether the machine errs on YES-instances (but not on NO-instances), or the other way around.

3. Machines that *never err*, but may output a special `don't know` symbol, say, with probability at most 1/2.

We focus on probabilistic polynomial-time machines, and on error probability that may be reduced to a "negligible" (e.g., exponentially vanishing in the input length) amount by polynomially many independent repetitions.

We comment that an alternative formulation of randomized computations is captured by (deterministic) machines that take two inputs, the first representing the actual input and the second representing the coin tosses (or the "random input"). For such machines, one considers the output distribution for any fixed first input, when the second input is uniformly distributed among the set of strings of adequate length.

7.1 Two-sided error: BPP

The standard definition of $\mathcal{BPP}$ is in terms of machines that err with probability at most 1/3. That is, $L \in \mathcal{BPP}$ if there exists a probabilistic polynomial-time machine M such that for every $x \in L$ (resp., $x \notin L$) it holds that $\Pr[M(x) = 1] \geq 2/3$ (resp., $\Pr[M(x) = 1] \leq 1/3$). In other words, letting χ_L denote the characteristic function of L, we requite that $\Pr[M(x) \neq \chi_L(x)] \leq 1/3$ for every $x \in \{0,1\}^*$. The choice of the constant 1/3 is immaterial, and any other constant smaller than 1/2 will do (and yield the very same class). In fact, a more general statement, which is proved by so-called "amplification" (see next), holds.

Error reduction (or confidence amplification). For any function $\epsilon : \mathsf{N} \to (0, 0.5)$, consider the class BPP_ϵ of sets L such that there exists a probabilistic polynomial-time machine M for which $\Pr[M(x) \neq \chi_L(x)] \leq \epsilon(|x|)$ holds. Clearly, $\mathcal{BPP} = BPP_{1/3}$. However, a wide range of other classes also equal $\mathcal{BPP}$. In particular,

1. For every positive polynomial p, the class BPP_ϵ, where $\epsilon(n) = (1/2) - (1/p(n))$, equals $\mathcal{BPP}$. That is, any error that is ("noticeably") bounded away from 1/2 (i.e., error $(1/2) - (1/\mathrm{poly}(n))$) can be reduced to an error of 1/3.

2. For every positive polynomial p, the class BPP_ϵ, where $\epsilon(n) = 2^{-p(n)}$, equals $\mathcal{BPP}$. That is, an error of 1/3 can be further reduced to an exponentially vanishing error.

Both facts are proven by applying an adequate Law of Large Numbers. That is, consider independent copies of a random variable that represents the output of the weaker machine (i.e., the machine having larger error probability). Use the adequate Law of Large Numbers to bound the probability that the average of these independent outcomes deviates from the expected value of the original random variable. Indeed, the resulting machine will invoke the original machine sufficiently many times, and rule by majority. We stress that invoking a randomized machine several times means that the random choices made in the various invocations are independent of one another.

BPP is in the Polynomial-Time Hierarchy: Clearly $\mathcal{P} \subseteq \mathcal{BPP}$, and it is commonly conjectured that equality holds (although a polynomial slow-down may occur when transforming, according to these conjectures, a probabilistic polynomial-time algorithm into a deterministic one). However, it is not known whether or not $\mathcal{BPP}$ is contained in $\mathcal{NP}$. In view of this ignorance, the following result is of interest:

Theorem 7.1 $\mathcal{BPP} \subseteq \Sigma_2$.

Proof: Suppose that $L \in \mathcal{BPP}$, and consider (by suitable error-reduction) a probabilistic polynomial-time algorithm A such that $\Pr[A(x) \neq \chi_L(x)] < 1/3\ell(|x|)$ for all $x \in \{0,1\}^*$, where $\ell(|x|)$ denotes the number of coins tossed by $A(x)$. Let us consider the residual deterministic two-input algorithm A' such that $A'(x,r)$ equals the output of A on input x and random choices $r \in \{0,1\}^{\ell(|x|)}$. We claim that $x \in L$ if and only if

$$\exists s_1, s_2, ..., s_{\ell(|x|)} \in \{0,1\}^{\ell(|x|)} \forall r \in \{0,1\}^{\ell(|x|)} \bigvee_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) = 1) \tag{7.1}$$

Once the claim is proved, the theorem follows by observing that Eq. (7.1) fits the definition of Σ_2. In order to prove the claim, we first consider the case $x \in L$. We use the Probabilistic Method to show that an adequate sequence of s_i's exists. That is, we show that most sequences of s_i's are adequate, by upper bounding the probability that a random sequence of s_i's is not adequate:

$$
\begin{aligned}
&\mathrm{Pr}_{s_1,s_2,\ldots,s_{\ell(|x|)}}[\neg\forall r \in \{0,1\}^{\ell(|x|)} \bigvee_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) = 1)] \\
=\;& \mathrm{Pr}_{s_1,s_2,\ldots,s_{\ell(|x|)}}[\exists r \in \{0,1\}^{\ell(|x|)} \bigwedge_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) \neq 1)] \\
\leq\;& \sum_{r\in\{0,1\}^{\ell(|x|)}} \mathrm{Pr}_{s_1,s_2,\ldots,s_{\ell(|x|)}}[\bigwedge_{i=1}^{\ell(|x|)} (A'(x, s_i \oplus r) \neq 1)] \\
=\;& \sum_{r\in\{0,1\}^{\ell(|x|)}} \prod_{i=1}^{\ell(|x|)} \mathrm{Pr}_{s_i}[A'(x, s_i \oplus r) \neq 1] \\
<\;& 2^{\ell(|x|)} \cdot \left(\frac{1}{3\ell(|x|)}\right)^{\ell(|x|)} \;=\; \left(\frac{2}{3\ell(|x|)}\right)^{\ell(|x|)} \;\ll\; 1
\end{aligned}
$$

where the last inequality is due to the fact that, for any fixed $x \in L$ and r, it holds that $\mathrm{Pr}_{s_i}[A'(x, s_i \oplus r) \neq 1] = \mathrm{Pr}_s[A'(x, s) \neq \chi_L(x)] < 1/3\ell(|x|)$. On the other hand, for any $x \notin L$ and every sequence of s_i's, it holds that $\mathrm{Pr}_r[\bigvee_{i=1}^{\ell(|x|)} A'(x, s_i \oplus r) = 1] < 1/3 < 1$ (since $x \notin L$). Thus, Eq. (7.1) cannot possibly hold for $x \notin L$. ■

We comment that the same proof idea yields a variety of similar statements (e.g., see Section 12.2).

7.2 One-sided error: RP and coRP

The class $\mathcal{RP}$ is defined as containing any set L such that there exists a probabilistic polynomial-time machine M satisfying the following two conditions

$$x \in L \implies \Pr[M(x) = 1] \geq \frac{1}{2} \tag{7.2}$$
$$x \notin L \implies \Pr[M(x) = 0] = 1 \tag{7.3}$$

Observe that $\mathcal{RP} \subseteq \mathcal{NP}$ (e.g., note that $\mathcal{NP}$ is obtained by replacing Eq. (7.2) with the condition $\Pr[M(x) = 1] > 0$, for every $x \in L$). Again, the specific probability threshold in Eq. (7.2) is immaterial as long as it is noticeable (and sufficiently bounded from 1).[1] Thus, $\mathcal{RP} \subseteq \mathcal{BPP}$.

Exercise: Prove that L' is in the class $\mathrm{co}\mathcal{RP} = \{\{0,1\}^* \setminus L : L \in \mathcal{RP}\}$ if and only if there exists a probabilistic polynomial-time machine M' satisfying the following two conditions

$$x \in L' \implies \Pr[M'(x) = 1] = 1 \tag{7.4}$$
$$x \notin L' \implies \Pr[M'(x) = 0] \geq \frac{1}{2} \tag{7.5}$$

[1] Exercise: Let $\mathcal{RP}_\epsilon$ denote the class obtained by replacing Eq. (7.2) by the condition $\Pr[M(x) = 1] \geq 1 - \epsilon(|x|)$, for every $x \in L$. Observe that $\mathcal{RP}_{1/2} = \mathcal{RP}$, and prove that $\mathcal{RP}_{1-(1/p(n))} = \mathcal{RP}$ and $\mathcal{RP}_{2^{-p(n)}} = \mathcal{RP}$, for any positive polynomial p. (Note that amplification is easier in this case (of one-sided error).)

The well-known randomized primality testing algorithms always accept prime numbers and rejects composite number with high probability. Thus, these algorithms establish that the set of prime numbers is in co$\mathcal{RP}$.

7.3 No error: ZPP

Whereas in case of $\mathcal{BPP}$ we have allowed two-sided errors, and in case of $\mathcal{RP}$ (and co$\mathcal{RP}$) we have allowed one-sided errors, we now allow no errors at all. Instead, we allow the algorithm to output a special `don't know` symbol, denoted $\perp$, with some bounded (away from 1) probability. The resulting class is denoted $\mathcal{ZPP}$, standing for Zero-error Probabilistic Polynomial-time. The standard definition of $\mathcal{ZPP}$ is in terms of machines that output $\perp$ with probability at most 1/2. That is, $L \in \mathcal{ZPP}$ if there exists a probabilistic polynomial-time machine M such that $\Pr[M(x) \in \{\chi_L(x), \perp\}] = 1$ and $\Pr[M(x) = \chi_L(x)] \geq 1/2$ for every $x \in \{0,1\}^*$. Again, the choice of the constant (i.e., 1/2) is immaterial, and "amplification" can be conducted as in case of $\mathcal{RP}$ (and yield the very same class). In fact, as in case of $\mathcal{RP}$, a more general statement holds.

Exercise: Prove that $\mathcal{ZPP} = \mathcal{RP} \cap \mathrm{co}\mathcal{RP}$. (Indeed, $\mathcal{ZPP} \subseteq \mathcal{RP}$ (as well as $\mathcal{ZPP} \subseteq \mathrm{co}\mathcal{RP}$) follows by a trivial transformation of the ZPP-machine. On the other hand, $\mathcal{RP} \cap \mathrm{co}\mathcal{RP} \subseteq \mathcal{ZPP}$ can be proved by combining the two machines guaranteed for a set in $\mathcal{RP} \cap \mathrm{co}\mathcal{RP}$.)

7.4 Randomized space complexity

The class $\mathcal{RL}$ (Random LogSpace) is defined analogously to the class $\mathcal{NL}$, and is indeed contained in the latter. Specifically, the syntax of Random LogSpace machines is identical to the one of Non-deterministic LogSpace machines, but the acceptance condition is probabilistic as in the case of $\mathcal{RP}$. In addition, we need to require explicitly that the machine runs in polynomial-time (or else $\mathcal{RL}$ extends up to $\mathcal{NL}$).[2]

Recall that Directed Connectivity is complete for $\mathcal{NL}$ (under log-space reductions). Below we show that undirected connectivity is solvable in $\mathcal{RL}$. Specifically, consider the set of triples (G, s, t) such that the vertices s and t are connected in the (undirected) graph G. On input (G, s, t), the randomized (log-space) algorithm starts a poly($|G|$)-long random walk at vertex s, and accepts the triplet if and only if the walk passed through vertex t. By a random walk we mean that at each step we select uniformly one of the neighbors of the current vertex and move to it. Observe that the algorithm can be implemented in logarithmic space (because we only need to store the current vertex as well as the number of steps taken so far), and that we never accept (G, s, t) in case s and t are not connected. We claim that if s and t are connected in $G = (V, E)$ then a random walk of length $O(|V| \cdot |E|)$ starting at s passes through t with probability at least 1/2. It follows that undirected connectivity is indeed in $\mathcal{RL}$.

[2] Recall that, w.l.o.g, a non-deterministic log-space machine need only run for polynomial-time. Such a computation can be simulated by a randomized log-space machine that repeatedly guesses non-deterministic moves and simulates the original machine on it. Note that we expect at most 2^t tries before we guess an accepting t-time computation, where t is polynomial in the input length. But what if there are no accepting t-time computations? To halt with a probabilistic rejecting verdict we should implement a counter that counts till 2^t, but we need to do so within space $O(\log t)$ (rather than t which is easy). In fact it suffices to have a randomized counter that with high probability counts to approximately 2^t. This can be implemented by tossing t coins until all show us HEADS. The expected number of times we need to repeat the experiment is 2^t, and we can implement this by a counter that counts till t (using space $\log_2 t$).

On proving the Random Walk Claim: (Indeed, this has little to do with the current course.) Consider the connected component of vertex s, denoted $G' = (V', E')$. For any pair, (u, v), let $T_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is first encountered. First verify that $\mathrm{E}[T_{u,v}] \leq 2|E'|$, for any (u, v) such that $\{u, v\} \in E'$.[3] Next, letting $\mathrm{cover}(G')$ be the expected number of steps in a random walk starting at s and ending when the last of the vertices of V' is encountered, and C be any directed cyclic tour that visits all vertices in G', we have $\mathrm{cover}(G') \leq \sum_{(u,v)\in C} \mathrm{E}[T_{u,v}] \leq |C| \cdot 2|E'|$. Letting C be a traversal of some spanning tree of G', we conclude that $\mathrm{cover}(G') < 4 \cdot |E'| \cdot |V'|$. Thus, with probability at least 1/2, a random walk of length $8 \cdot |E'| \cdot |V'|$ starting at s visits all vertices of G'.

[3] For example, let $C_{u,v}(n)$ be a random variable counting the number of minimal u-to-v sub-paths within a random walk of length n, where the walk starts at the stationary vertex distribution (assuming the graph is not bipartite or is sligtly modified otherwise). On one hand, $\mathrm{E}[T_{u,v}] = \lim_{n\to\infty}(n/\mathrm{E}[C_{u,v}(n)])$ (due to the memoryless property of the walk). On the other hand, $\mathrm{E}[C_{u,v}(n)] + 1$ is lower bounded by the expected number of times that the edge (v, u) was travesed (from v to u) in such a (n-step) walk, where the latter expected number equals $n/2|E'|$ (because each directed edge appears (in each step) on the walk with equal probability). It follows that $\mathrm{E}[T_{u,v}] \leq \lim_{n\to\infty}(n/((n/2|E'|) - 1)) = 2|E'|$.

Lecture 8

Non-Uniform Complexity

All complexity classes considered so far are "uniform" in the sense that each set in each of these classes was defined via one finite machine (or finite expression), which applied to all input lengths. This is indeed in agreement with the basic algorithmic paradigm of designing algorithms that can handle all inputs.

In contrast, non-uniform complexity investigates what happens when we allow to use a different algorithm for each input length. Indeed, in such a case, we must bound the description size of the algorithm (otherwise any problem can be solved by incorporating in the algorithm the answers to all finitely many inputs of the adequate length). By considering non-uniform complexity, we are placing an upper-bound on what can be done by the corresponding uniform-complexity class. The hope is that by abstracting away the ("evasive") uniformity condition, we will get a finite combinatorial structure that we may be able to understand.

8.1 Circuits and advice

Focusing on non-uniform polynomial-time, we mention two standard ways of defining non-uniform complexity classes. The first way is by considering (families of) Boolean circuits (as in Section 3.3). Specifically, L is said to be in non-uniform polynomial-time, denoted $\mathcal{P}/\text{poly}$, if there exists an infinite sequence of Boolean circuits $C_1, C_2, \ldots$ such that for some polynomial p the following three conditions hold:

1. The circuit C_n has n inputs and one output.
2. The size (e.g., number of edges) of the circuit C_n is at most $p(n)$.
3. For every $x \in \{0,1\}^n$, it holds that $C_n(x) = 1$ if and only if $x \in L$.

That is, C_n is a non-trivial algorithm (i.e., it cannot explicitly encode all 2^n answers) for deciding the membership in L of n-bit long strings. However, although C_n has size at most $p(n)$, it is not clear whether one can construct C_n in poly(n)-time (or at any time; see below).

An alternative way of defining $\mathcal{P}/\text{poly}$ proceeds by considering "machines that take advice". That is, we consider deterministic polynomial-time machines that get two inputs, where the second input (i.e., the advice) has length that is at most polynomial in the first input. The advice may only depend on the input length, and thus it cannot explicitly encode the answers to all inputs (of that length). Specifically, $L \in \mathcal{P}/\text{poly}$ if there exists a deterministic polynomial-time machine M and an infinite sequence of advice strings $a_1, a_2, \ldots$ such that for some polynomial p the following conditions hold:

1. The length of a_n is at most $p(n)$.
2. For every $x \in \{0,1\}^n$, it holds that $M(x, a_n) = 1$ if and only if $x \in L$.

Exercise: Prove that the two formulations of $\mathcal{P}/\text{poly}$ are indeed equivalent. Furthermore, prove that without loss of generality, the machine M (as above) may be a universal machine.

8.2 The power of non-uniformity

Waiving the "uniformity condition" allows non-uniform classes to contain non-recursive sets. This is true for $\mathcal{P}/\text{poly}$ as well as for most reasonable non-uniform classes, and is due to the obvious reason that there exists non-recursive unary sets. Specifically, any unary set $L \subseteq \{1\}^*$ (possibly non-recursive), can be decided by a linear-time algorithm that uses 1-bit long advice (i.e., $a_n \stackrel{\text{def}}{=} \chi_L(1^n)$ and $M(x, a_{|x|}) = 1$ if and only if both $x = 1^{|x|}$ and $a_{|x|} = 1$).

On the other hand, the existence of sets that are not in $\mathcal{P}/\text{poly}$ can be proven in a more "concrete" way than the corresponding statement for $\mathcal{P}$. Fixing any super-polynomial and sub-exponential function f, we observe that the number of possible $f(n)$-bit long advice is much smaller than the number of possible subsets of $\{0,1\}^n$, whereas these advice account for all the sets that $\mathcal{P}/\text{poly}$ may recognize (using a universal machine).

We took it for granted that $\mathcal{P} \subseteq \mathcal{P}/\text{poly}$, which is indeed true (e.g., by using empty advice strings). The fact that $\mathcal{P}/\text{poly}$ also contains $\mathcal{BPP}$ is less obvious. Before proving this fact, let use mention that it is widely believed that $\mathcal{P}/\text{poly}$ does not contain $\mathcal{NP}$, and indeed proving the latter conjecture was suggested as a good way for establishing that $\mathcal{P} \neq \mathcal{NP}$. (Whether or not this way is a good one is controversial.)

Theorem 8.1 *$\mathcal{BPP} \subset \mathcal{P}/\text{poly}$*

Proof: As in the proof of Theorem 7.1, we consider an adequate amplification of $\mathcal{BPP}$. Here, for $L \in \mathcal{BPP}$, we consider (by suitable error-reduction) a probabilistic polynomial-time algorithm A such that $\Pr[A(x) \neq \chi_L(x)] < 2^{-|x|}$. Again, let us consider the residual deterministic two-input algorithm A' such that $A'(x,r)$ equals the output of A on input x and random choices $r \in \{0,1\}^{\ell(|x|)}$. Then, by a trivial counting argument, there exists a string $r \in \{0,1\}^{\ell(n)}$ such that $A'(x,r) = \chi_L(x)$ for all x's of length n. Using this string r as the advice for n-bit long inputs, we are done. ■

8.3 Uniformity

The non-uniform aspect of the definition of $\mathcal{P}/\text{poly}$ is the lack of requirements regarding the constructibility of the circuits (resp., advice). As a sanity check, we note that requiring that these objects be polynomial-time constructible results in a cumbersome definition of $\mathcal{P}$. That is, suppose that we require that there is a polynomial-time algorithm A that given 1^n outputs the circuit C_n (resp., the advice a_n) for deciding $L \in \mathcal{P}/\text{poly}$ (as per the definition above). Then, combining A with the standard circuit-evaluation algorithm (resp., the advice-taking machine M), we obtain an ordinary polynomial-time algorithm for deciding L.

8.4 Evidence that P/poly does not contain NP

Recall that a major motivation towards studying $\mathcal{P}/\text{poly}$ is the desire to prove that $\mathcal{P}/\text{poly}$ does not contain $\mathcal{NP}$ (and thus also $\mathcal{BPP} \supseteq \mathcal{P}$ does not contain $\mathcal{NP}$). In view of the fact that $\mathcal{P}/\text{poly}$

contains non-recursive sets, one may wonder how feasible is the conjecture that $\mathcal{P}$/poly does not contain $\mathcal{NP}$. It would have been best if we knew that $\mathcal{NP} \subset \mathcal{P}$/poly if and only if $\mathcal{P} = \mathcal{NP}$. But we only know that $\mathcal{NP} \subset \mathcal{P}$/poly implies a collapse of the Polynomial-time Hierarchy. That is:

Theorem 8.2 $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ *implies that* $\mathcal{PH} = \Sigma_2$.

Proof sketch: We show that $\Pi_2 = \Sigma_2$, and the claim follows by Theorem 6.2. Suppose that $L \in \Pi_2$, and let us consider the corresponding quantified expression (for $x \in L$): $\forall y \exists z R(x,y,z) = 1$, where $y, z \in \{0,1\}^{\text{poly}(|x|)}$. Let $L' \stackrel{\text{def}}{=} \{(x,y) : \exists z R(x,y,z) = 1\}$, and observe that L' is in $\mathcal{NP}$, and thus in $\mathcal{P}$/poly. Thus, $x \in L$ if and only if for $m = \text{poly}(|x|)$ there exists a poly(m)-size circuit C_m for deciding $L' \cap \{0,1\}^m$ such that for all y's it holds that $C_m(x,y) = 1$. The above expression is almost of the adequate (i.e., Σ_2) form, except that we need to check that C is indeed correct on all inputs of length m. Suppose that L' was *downwards self-reducible*; that is, that deciding whether $w \in L'$ could be reduced to deciding membership in L' of shorter (than w) strings. Then, we could have revised the above expression and assert that $x \in L$ if and only if there exists a sequence of polynomial-size circuits $C_1, ..., C_m$ such that

1. for all y's it holds that $C_m(x,y) = 1$;
2. for $i = 1, ..., m$, the circuit C_i correctly determines membership in L', where correctness of C_i is expressed by saying that for all $w \in \{0,1\}^i$ the value of $C_i(w)$ is consistent with the values obtained by the downwards self-reduction (as answered by the already verified circuits $C_1, ..., C_{i-1}$).

However, we have no reason to assume that L' is self-reducible. What we do instead is reduce L' to SAT and apply the argument to SAT (using its polynomial-size circuits (which exist by the hypothesis) and its downwards self-reducibility (which is a very natural procedure)). Specifically, let f be a Karp-reduction of L' to SAT. Thus, $x \in L$ if and only if $\forall y\, f(x,y) \in SAT$. Using the hypothesis, we have $SAT \in \mathcal{P}$/poly, and thus there exists a sequence of polynomial-size circuits $C_1, C_2, ...$ for SAT. Now, we assert that $x \in L$ if and only if there exists a sequence of polynomial-size circuits $C_1, ..., C_m$, where $m = |f(x,y)|$, such that the following two conditions hold:

1. For all y's (of adequate length), $C_m(f(x,y)) = 1$.
2. For $i = 1, ..., m$, the circuit C_i correctly decides membership of i-bit long strings in SAT. Note that the correctness condition for C_i can be expressed as follows: For every i-long formula ϕ it holds that $C_i(\phi) = 1$ if and only if either $C_{i'}(\phi') = 1$ or $C_{i''}(\phi'') = 1$, where ϕ' (resp., ϕ'') is the formula obtained from ϕ by replacing its first variable with 0 (resp., 1), and i' (resp., i'') is the length of the resulting formula after straightforward simplifications (which necessarily occurs after instantiating a variable).

Observe that the expression obtained for membership in L is indeed of the Σ_2-form. The theorem follows. $\blacksquare$

8.5 Reductions to sparse sets

Another way of looking at $\mathcal{P}$/poly is as the class of sets that are Cook-reducible to a sparse set, where a sparse set is a set that contains at most polynomially many strings of each length. (The reason for stressing the fact that we refer to Cook-reductions will be explained below.) Let us first establish the validity of the above claim.

Proposition 8.3 *$L \in \mathcal{P}/\mathrm{poly}$ if and only if L is reducible to some sparse set.*

Proof sketch: Suppose that $L \in \mathcal{P}/\mathrm{poly}$ and suppose that n is sufficiently large. Then we can encode the nth advice string (i.e., a_n) in (the first $|a_n|$ strings of) the n-bit slice of a set S (i.e., by placing the i^{th} (n-bit) string in $S \cap \{0,1\}^n$ if and only if the i^{th} bit of a_n equals 1). Observe that S is indeed sparse (because $|a_n| = \mathrm{poly}(n)$). On input x, the reduction first retrieves the advice string $a_{|x|}$ (by making polynomially-many n-bit long queries to S), and decides according to the advice-taking $M(x, a_{|x|})$.

In case L is reducible to a sparse set S, we let the nth advice encode the list of all the strings in S that have length at most $q(n)$, where q is the polynomial bounding the running-time of the reduction. Given this advice (which is of length $\sum_{i=1}^{q(n)} |S \cap \{0,1\}^i| \cdot i = \mathrm{poly}(n)$), the advice-taking machine can emulate the answers of the oracle machine (of the reduction), and thus decide L. ■

As a direct corollary to Proposition 8.3, we obtain:

Corollary 8.4 *SAT is Cook-reducible to a sparse set if and only if $\mathcal{NP} \subseteq \mathcal{P}/\mathrm{poly}$.*

Combining Corollary 8.4 and Theorem 8.2, it follows that SAT cannot be Cook-reducible to a sparse set, unless the Polynomial-time hierarchy collapses.

Perspective: Karp-reductions to sparse sets

We have stressed the fact that we refer to Cook-reductions, because (by Corollary 8.4) SAT is Cook-reducible to a sparse set if and only if $\mathcal{NP} \subseteq \mathcal{P}/\mathrm{poly}$. In contrast, it is known that SAT is Karp-reducible to a sparse set if and only if $\mathcal{NP} = \mathcal{P}$. Thus, the difference between Cook and Karp reductions is "reflected" in the difference between $\mathcal{NP} \subseteq \mathcal{P}/\mathrm{poly}$ and $\mathcal{NP} = \mathcal{P}$.

Theorem 8.5 *SAT is Karp-reducible to a sparse set if and only if $\mathcal{NP} = \mathcal{P}$.*

Proof of a special case: Clearly, if $\mathcal{NP} = \mathcal{P}$ then SAT is Karp-reducible to any non-trivial set (e.g., to the set $\{1\}$). We establish the opposite direction only for the special case that SAT is Karp-reducible to some set S such that S is a subset of a *sparse* set $G \in \mathcal{P}$. (Such a set S is called guarded, and $S \subseteq \{1\}^*$ is indeed a special case). Specifically, using the Karp-reduction of SAT to S, we present a (deterministic) polynomial-time decision procedure for SAT. The procedure conducts a DFS on the tree of all possible partial truth assignment to the input formula, while truncating the search at nodes that are roots of sub-trees that contain no satisfying assignment (at the leaves).[1] The key observation is that each internal node (which yields a formula derived from the initial formulae by instantiating the corresponding partial truth assignment) is mapped by the reduction either to a string not in G (in which case we conclude that the sub-tree contains no satisfying assignments) or to a string in G (in which case we don't know what to do). However, once we backtrack from this internal node, we know that the corresponding element of G is not in S, and we will never extend a node mapped to this element again. Specifically, let ϕ be the input formula, and ϕ_τ denote the formula resulting from ϕ by setting its first $|\tau|$ variables according to the partial truth assignment τ. Then, the procedure proceeds as follows, using the Karp-reduction f of SAT to S:

[1] For an n-variable formulae, the leaves of the tree correspond to all possible n-bit long strings, and an internal node corresponding to τ is the parent of nodes corresponding to $\tau 0$ and $\tau 1$.

Initialization: $\tau \leftarrow \lambda$ and $B \leftarrow \emptyset$, where τ is a partial truth assignment for which we wish to determined whether or not $\phi_\tau \in SAT$, and $B \subset G \setminus S$ is a set of strings that were already "proved" not to be in S.

The following steps are recursive and return a Boolean value, representing the whether or not $\phi_\tau \in SAT$.

Internal node: Determine whether or not $\phi_\tau \in SAT$, according to the following three cases.

1. If $f(\phi_\tau) \notin G$ then return the value `false`.
(Since $S \subseteq G$, we have $f(\phi_\tau) \notin S$, and by the validity of the reduction, $\phi_\tau \notin SAT$.)
2. If $f(\phi_\tau) \in B$ then return the value `false`.
(Since $B \subseteq G \setminus S$, we have $f(\phi_\tau) \notin S$, and by the validity of the reduction, $\phi_\tau \notin SAT$.)
3. Otherwise (i.e., $f(\phi_\tau) \in G \setminus B$), invoke two recursive calls, for $\phi_{\tau 0}$ and $\phi_{\tau 1}$, respectively.[2] If both calls have returned `false` and $f(\phi_\tau) \in G \setminus B$ then add $f(\phi_\tau)$ to B (since $\phi_\tau \notin SAT$ holds). Actually, if the first call returns `true` then the second call does not take place.[3] In any case, return the OR-value of the two values returned by the recursive calls.

(We stress that only the third case invokes recursive calls.)

Bottom Level: If the constant "formula" ϕ_τ is `false` and $f(\phi_\tau) \in G \setminus B$ then add $f(\phi_\tau)$ to B. In any case, return the value of ϕ_τ.

It is easy to verify that the procedure returns the correct answer. The running-time analysis is based on the observation that if τ' and τ'' are not prefixes of one another and $f(\phi_{\tau'}) = f(\phi_{\tau''})$ then it cannot be that Case 3 was applied to both of them. Thus, the number of internal nodes for which Case 3 was applied is at most the depth of the tree times $|\cup_{i=1}^m G_i \setminus S| \leq \sum_{i=1}^m |G_i| = \text{poly}(m)$, where $G_i \stackrel{\text{def}}{=} G \cap \{0,1\}^i$ and $m = |f(\phi)| = \text{poly}(|\phi|)$. ■

[2] We may re-evaluate the condition $f(\phi_\tau) \in B$ after obtaining the answer of the first call, but this is not really necessary.

[3] Otherwise, the procedure will visit all satisfying assignments, and consequently may run for exponential time.

Lecture 9

Counting Classes

9.1 The definition of #P

A natural computational problem associated with an NP-relation R is to determine the number of solutions for a given instance; that is, given x, determine the cardinality of $R(x) \stackrel{\text{def}}{=} \{y : (x,y) \in R\}$. This problem is the counting problem associated with R. Certainly, the counting problem associated with R is not easier than the problem of deciding membership in $L_R = \{x : \exists y \text{ s.t. } (x,y) \in R\}$ (which can be casted as determining, for a given x, whether $|R(x)|$ is positive or zero).

The class $\#\mathcal{P}$ can be defined as a class of functions that count the number of solutions in NP-relations. That is, $f \in \#\mathcal{P}$ if there exists an NP-relation R such that $f(x) = |R(x)|$ for all x's. Alternatively, we can define $\#\mathcal{P}$ as a class of sets, where for every NP-relation R the set $\#R \stackrel{\text{def}}{=} \{(x,k) : |R(x)| \geq k\}$ is in $\#\mathcal{P}$. (Exercise: Formulate and show the "equivalence" between the two definitions.)

Relation to PP. The class $\#\mathcal{P}$ is related to a probabilistic class, denoted $\mathcal{PP}$, that was not defined in Lecture 7. We say that $L \in \mathcal{PP}$ if there exists a probabilistic polynomial-time algorithm A such that, for every x, it holds that $\Pr[A(x) = 1] > 1/2$ if and only if $x \in L$ (or, alternatively, $\Pr[A(x) = \chi_L(x)] > 1/2$ for every x).[1] (Recall that, in contrast, $L \in \mathcal{BPP}$ requires that $\Pr[A(x) = \chi_L(x)] > 2/3$ for every x.) Notice that any $L \in \mathcal{PP}$ can be decided by a polynomial-time oracle machine that is given oracle access to $\#R$, where R describes the actions of the PP-algorithm (i.e., $(x,r) \in R$ iff $A(x)$ accepts when using coins r). On the other hand, $\#\mathcal{P} \subseteq \mathcal{PP}$, by virtue of (a minor modification to) the following algorithm that refers to $\#R$, where $R \subseteq \cup_{n\in\mathbb{N}}\{0,1\}^n \times \{0,1\}^{m(n)}$: on input (x,k), with probability one half select y uniformly in $\{0,1\}^{m(|x|)}$ and accept iff $(x,y) \in R$, and otherwise (i.e., with probability $1/2$) accept with probability exactly $1 - (k - 0.5) \cdot 2^{-m(|x|)}$. (Exercise: Provide the missing details for all the above claims.)

9.2 #P-complete problems

We say that a computational problem is $\#\mathcal{P}$-complete if it is in $\#\mathcal{P}$ and every problem in $\#\mathcal{P}$ is reducible to it. Thus, for an NP-relation R, the problem $\#R$ (which is always in $\#\mathcal{P}$) is $\#\mathcal{P}$-complete if for any NP-relation R' it holds that $\#R'$ is reducible to $\#R$. Using the standard Karp-reductions, it is easy to show that for any known NP-complete relation R the set $\#R$ is

[1] Exercise: show the equivalence of the two formulations.

$\#\mathcal{P}$-complete. This is the case because the standard reductions (or minor modifications of them) are "parsimonious" (i.e., preserve the number of solutions). In particular:

Proposition 9.1 *$\#SAT$ is $\#\mathcal{P}$-complete, where $(\phi, k) \in \#SAT$ if and only if ϕ has at least k different satisfying assignment.*

Exercise: Verify that the standard reduction of any NP-relation to SAT is parsimonious; that is, for any NP-relation R, the standard reduction of R to SAT maps each x to a formula having exactly $|R(x)|$ satisfying assignments.

As stated above, Proposition 9.1 is merely a consequence of the nature of the reductions used in the standard context of NP-completeness results. Specifically, it is the case that the same reductions used to demonstrate NP-completeness of search problems can be used to show $\#\mathcal{P}$-completeness of the corresponding counting problems. Consequently, "hard" (i.e., NP-complete) search problems give rise to "hard" (i.e., $\#\mathcal{P}$-complete) counting problems. Interestingly, there are "hard" counting problems (i.e., $\#\mathcal{P}$-complete problems) for which the corresponding search problem is easy. For example, whereas the problem of finding a maximum matching in a given graph is "easy" (i.e., solvable in polynomial-time), the corresponding counting problem is "hard" (i.e., $\#\mathcal{P}$-complete):

Theorem 9.2 *The problem of counting the number of perfect matching in a bipartite graph is $\#\mathcal{P}$-complete. Equivalently, the problem of computing the permanent of integer matrices with 0/1-entries is $\#\mathcal{P}$-complete.*

Needless to say, the reduction used in proving Theorem 9.2 is not parsimonious (or else we could have used it to reduce $\mathcal{NP}$ to the problem of deciding whether a given graph has a perfect matching). For the same reason, the recent polynomial-time algorithm for approximating the permanent (of non-negative matrices)[2] does not yield polynomial-time approximation algorithms for all $\#\mathcal{P}$.

9.3 A randomized reduction of Approximate-#P to NP

By an approximation for a counting problem $\#R$ in $\#\mathcal{P}$, we mean a procedure that on input x outputs a "good" approximation, denoted $A(x)$, of $|R(x)|$. Specifically, we require that with high probability, the ratio $A(x)/|R(x)|$ will be bounded. For many natural NP-relations (and in particular for SAT), the following notions are all equivalent:

1. With probability at least 2/3, it holds that $A(x)$ is within a factor of 2 of $|R(x)|$ (i.e., $1 \leq A(x)/|R(x)| \leq 2$).[3]

2. With probability at least $1 - \exp(-|x|)$, it holds that $1 \leq A(x)/|R(x)| \leq 2$.

3. With probability at least $1 - \exp(-|x|)$, it holds that $1 \leq A(x)/|R(x)| \leq 1 + |x|^{-c}$, where $c > 0$ is any fixed constant.

4. With probability at least 2/3, it holds that $1 < A(x)/|R(x)| < 2^{|x|^c}$, where $c < 1$ is any fixed constant.[4]

[2] See Jerrum, Sinclair and Vigoda: A Polynomial-Time Approximation Algorithm for the Permanent of a Matrix with Non-Negative Entries, in *Proc. of the 33rd STOC*, pages 712–721, 2001.

[3] Show that this is equivalent to ability to get $A(x)$ such that $1/\sqrt{2} \leq A(x)/|R(x)| \leq \sqrt{2}$.

[4] Note that for some constant c that depends on R, the ability to approximate $|R(x)|$ to within a factor of $2^{|x|^c}$ merely requires the ability to distinguish the case $|R(x)| = 0$ from $|R(x)| > 0$ (since $|R(x)| \leq 2^{|x|^c}$ always holds). Exercise: Show that ability to approximate every $|R(x')|$ to within a factor of $2^{|x'|}$ implies ability to approximate $|R(x)|$ to within a factor of $2^{|x|^c}$.

Item 1 implies Item 2 by using straightforward error-reduction (as in case of $\mathcal{BPP}$). To show that Item 4 implies Item 1 (resp., Item 2 implies Item 3), we use the fact that for many natural NP-relations it is the case that many instances can be encoded in one (i.e., $R(\langle x_1, ..., x_t \rangle) = \{\langle y_1, ..., y_t \rangle : \forall i \, y_i \in R(x_i)\}$).[5] Thus, suppose that (for every x) we know how to approximate $|R(x)|$ to within a factor of $2^{|x|^{2/3}}$, and we want to approximate $|R(x)|$ to within a factor of 2 (for every x). Then, we form x' as a sequence of $t \stackrel{\text{def}}{=} |x|^2$ copies of x, and obtain a $2^{|x'|^{2/3}}$-factor approximation of $|R(x')| = |R(x)|^t$. Taking the t^{th} root of this approximation, we obtain $|R(x)|$ up-to a factor of $(2^{|x'|^{2/3}})^{1/t} = 2^{(t \cdot |x|)^{2/3}/t} = 2$.

In view of the above, we focus on providing any good approximation to the problem of counting the number of satisfying assignments to a boolean formula. The same techniques apply to any NP-complete problem.

Theorem 9.3 *The counting problem #SAT can be approximated up to a constant factor by a probabilistic polynomial-time oracle machine with oracle access to SAT.*

Proof Sketch: Given a formula ϕ on n variables, we approximate $|SAT(\phi)|$ by trying all possible powers of 2 as candidate approximations. That is, for $i = 1, ..., n$, we check whether 2^i is a good approximation of $|SAT(\phi)|$. This is done by uniformly selecting a "good" hashing function $h : \{0,1\}^n \to \{0,1\}^i$, and checking whether there exists a truth assignment τ for ϕ such that the following two conditions hold:

1. the truth assignment τ satisfies ϕ (i.e., $\phi(\tau) = \texttt{true}$), and
2. h hashes τ to the all-zero string (i.e., $h(\tau) = 0^i$).

These two conditions can be encoded in a new formula (e.g., by reducing the above NP-condition to SAT).[6] The new formula ϕ' is satisfiable if and only if there exists an assignment τ (to ϕ) that satisfies the above conditions. Thus, the answer to the above question (i.e., whether such a τ exists) is obtained by making a corresponding query (i.e., ϕ') to the SAT oracle.

In the analysis, we assume that the hashing function is good in the sense that for any $S \subseteq \{0,1\}^n$, with high probability, a randomly selected hashing function h satisfies $|\{e \in S : h(e) = 0^i\}| \approx |S|/2^i$. In particular, a randomly selected hashing function h maps each string to 0^i with probability 2^{-i}, and the mapping of different strings is pairwise independent. For further details, see [28, Lect. 4].

Note that if $|SAT(\phi)| < 2^{i-2}$ then a randomly selected hashing function is unlikely to map any of the (fewer than 2^{i-2}) satisfying assignment of ϕ to 0^i. Specifically, the probability that any specific assignment is mapped to 0^i equals 2^{-i}, and so the bad event occurs with probability less than 1/4, which can be further reduced by repeating the random experiment.

On the other hand, if $|SAT(\phi)| > 2^{i+2}$ then a randomly selected hashing function is likely to map some of the (more than 2^{i+2}) satisfying assignment of ϕ to 0^i. This can be proven using the pairwise independent property of the mapping induced by a random hashing function.

Thus, with high probability, the above procedure outputs a value $v = 2^i$ such that $i - 2 < \log_2 |SAT(\phi)| < i + 3$. We stress that the entire argument can be adapted to any NP-complete problem. Furthermore, smaller approximation factors can be obtained (directly) by using tricks as in the proof of Theorem 9.4. ■

[5] For example, the number of satisfying assignments to a formula consisting of t formulae over distinct variables is the product of the number of satisfying assignments to each of these formulae.

[6] Alternatively, for some popular hashing functions, the condition $h(\tau) = 0^i$ is easily transformed to CNF. Thus, we obtain the formula $\phi'(z_1, ..., z_n) = \phi(z_1, ..., z_n) \wedge (h(z_1 \cdots z_n) = 0^i)$.

9.4 A randomized reduction of SAT to Unique-SAT

The widely believed intractability of SAT cannot be due to instances that have "very many" satisfying assignments. For example, satisfying assignments for n-variable formula having at least $2^n/n$ satisfying assignments can be found in probabilistic polynomial-time by selecting n^2 assignments at random. Going to the other extreme, one may ask whether SAT instances having very few satisfying assignments (e.g., a unique satisfying assignment) can be hard. As shown below, the answer is positive. We show that ability to solve such instances yields ability to solve arbitrary instances.

In order to formulate the above discussion, we need to introduce the notion of a promise problem, which extends (or relaxes) the notion of a decision problem. A promise problem Π is a pair of disjoint subsets, denoted $(\Pi_{\text{yes}}, \Pi_{\text{no}})$. A (deterministic) machine M is said to solve such a problem if $M(x) = 1$ for every $x \in \Pi_{\text{yes}}$ and $M(x) = 0$ for every $x \in \Pi_{\text{no}}$, whereas nothing is required in case $x \notin \Pi_{\text{yes}} \cup \Pi_{\text{no}}$ (i.e., x "violates the promise"). (The notion extends naturally to probabilistic machines, oracle machines, and so on.) When we say that some problem reduces to the promise problem $\Pi = (\Pi_{\text{yes}}, \Pi_{\text{no}})$, we mean that the reduction yields the correct output regardless of the way in which queries outside of $\Pi_{\text{yes}} \cup \Pi_{\text{no}}$ are answered. (This is consistent with requiring nothing from a machine that solves Π in case the input is not in $\Pi_{\text{yes}} \cup \Pi_{\text{no}}$.)

The computational problem of distinguishing instances with a unique solution from instances with no solution yields a natural promise problem. For example, $uniqueSAT$ (or $uSAT$) is the promise problem with yes-instances being formulae having a unique satisfying assignment and no-instances being formulae having no satisfying assignment.

Theorem 9.4 *SAT is randomly reducible to $uSAT$.*

Proof Sketch: We present a probabilistic polynomial-time oracle machine that solves SAT using an oracle to $uSAT$. Actually, it is easier to first (randomly) reduce SAT to $fewSAT$, where $fewSAT$ is the promise problem with yes-instances being formulae having between 1 and 100 satisfying assignments and no-instances being formulae having no satisfying assignment.

Observe that the procedure described in the proof of Theorem 9.3 can be easily adapted to do the work. Specifically, we accept the given SAT instance ϕ if and only if any of the oracle invocations returns the value `true`. Note that the latter event may occur only if ϕ is satisfiable (because when ϕ is unsatisfiable all queries ϕ' are unsatisfiable). On the other hand, if ϕ has $k > 8$ satisfying assignments then in iteration $i = \lfloor \log_2 k \rfloor - 2$, with high probability, the query ϕ' is satisfiable and has at most $k/2^{i-2} < 32$ satisfying assignments (i.e., ϕ' is a yes-instance of $fewSAT$).[7]

To finish-up the proof we reduce $fewSAT$ to $uSAT$. Given a formula ϕ, for $i = 1, ..., 100$, we construct a formula $\phi^{(i)}$ that has a unique satisfying assignment if and only if ϕ has exactly i satisfying assignments. For example, $\phi^{(i)}$ may consist of the conjunction of i copies of ϕ over distinct variables and a condition imposing a lexicography order between the corresponding assignments.[8]
■

[7] In order to take care of the case $k \le 8 < 100$, we also query the $fewSAT$ oracle about ϕ itself.

[8] E.g., $\phi^{(2)}(x_1, ..., x_n, y_1, ..., y_n) \stackrel{\text{def}}{=} \phi(x_1, ..., x_n) \wedge \phi(y_1, ..., y_n) \wedge (\vee_{j=0}^{n-1}((x_{j+1} < y_{j+1}) \wedge (\wedge_{k=1}^{j} x_k = y_k)))$.

Lecture 10

Space is more valuable than time

This lecture was not given. The intention was to prove the following result, which asserts that any computation requires strictly less space than time.

Theorem 10.1 $\mathrm{DTIME}(t) \subseteq \mathrm{DSPACE}(t/\log t)$

That is, any given deterministic multi-tape Turing Machine (TM) of time complexity t, can be simulated using a deterministic TM of space complexity $t/\log t$. A main ingredient in the simulation is the analysis of a pebble game on directed bounded-degree graphs.

Lecture 11

Circuit Depth and Space Complexity

This lecture was not given. The intention was to explore some of the relations between Boolean circuits and Turing machines. Specifically:

- Define the complexity classes $\mathcal{NC}_i$ and $\mathcal{AC}_i$ (i.e., bounded versus unbounded fan-in circuits of polynomial-size and $O(\log^i)$-depth), and compare their computational power. Point out the connection between uniform-$\mathcal{NC}$ and "efficient" parallel computation.
- Establish a connection between the space complexity of a problem and the depth of circuits (with bounded fan-in) for the problem.

Historical Notes

For historical discussion of the material presented in Lecture 4, the reader is referred to the textbook of Hopcroft and Ullman [41]. Needless to say, the latter provides accurate statements and proofs of hierarchy and gap theorems.

Space Complexity: The emulation of non-deterministic space-bounded machines by deterministic space-bounded machines (i.e., Theorem 5.7) is due to Savitch [68]. Theorem 5.8 (i.e., $\mathcal{NL} = \text{co}\mathcal{NL}$) was proved independently by Immerman [42] and Szelepcsenyi [77].

The Polynomial-Time Hierarchy: The Polynomial-Time Hierarchy was introduced by Stockmeyer [75]. The third equivalent formulation via "alternating machines" can be found in [16].

Randomized Time Complexity: Probabilistic Turing Machines and corresponding complexity classes (including $\mathcal{BPP}, \mathcal{RP}, \mathcal{ZPP}$ and $\mathcal{PP}$) were first defined by Gill [23]. The random-walk (log-space) algorithm for deciding undirected connectivity is due to Aleliunas *et. al.* [2]. Additional examples of randomized algorithms and procedures can be found in [62] and [26, Apdx. B].

The robustness of the various classes under various error thresholds was established using straightforward amplifications (i.e., running the algorithm several times using independent random choices). Randomness-efficient amplification methods (which use related random choices in the various runs) have been studied extensively since the mid 1980's (cf. [26, Sec. 3.6]).

The fact that $\mathcal{BPP}$ is in the Polynomial-time hierarchy was proven independently by Lautemann [56] and Sipser [72]. We have followed Lautemann's proof. The ideas underlying Sipser's proof found many applications in complexity theory; in particular, they are used in the approximation procedure for $\#\mathcal{P}$ (as well as in the emulation of general interactive proofs by public-coin ones).

Non-Uniform Complexity: The class P/poly was defined by Karp and Lipton as part of a general formulation of "machines which take advise" [49]. They have noted the equivalence to the traditional formulation of polynomial-size circuits, the effect of uniformity, as well as the effect of $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ on the Polynomial-time hierarchy (i.e., Theorem 8.2). Theorem 8.5 is due to Fortune [20].

Theorem 8.1 is attributed to Adleman [1], who actually proved that $\mathcal{RP} \subseteq \mathcal{P}/\text{poly}$ using a more involved argument.

Counting Classes: The counting class $\#\mathcal{P}$ was introduced by Valiant [79], who proved that computing the permanent of 0-1 matrices is $\#\mathcal{P}$-complete (cf. Theorem 9.2). Valiant's proof first establishes the $\#\mathcal{P}$-hardness of computing the permanent of integer matrices (the entries are actually restricted to $\{-1, 0, 1, 2, 3\}$), and next reduces the computation of the permanent of integer matrices to the the permanent of 0-1 matrices. A de-constructed version of Valinat's proof can be found in [11].

The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [76], following an idea of Sipser [72]. Our exposition follows further developments in the area. The randomized reduction of SAT to uniqueSAT is due to Valiant and Vazirani [80]. Again, our exposition is a bit different.