The main goal of complexity theory is to understand the limits of efficient computation. The models we looked at so far are not strong enough to capture all efficient computations. Fairly simple functions like the determinant of a matrix are believed to be hard for both polynomial-width and logarithmic-depth circuits. To capture all efficient computation we need to drop restrictions on width and depth.

Unfortunately it is difficult to reason about unrestricted circuits. While there are many interesting functions, some of which we will see today, that are believed to require circuits of size exponential in their input, even proving a large *linear* size lower bound like $10n$ remains out of reach. This calls for a change of objective from absolute hardness (is problem A hard?) to relative hardness (is problem A at least as hard as problem B?)

When talking about general computation it is more convenient to pass to *uniform* models which can take in inputs of arbitrary length.

1 Search problems and their decision versions

Guided by the philosophy of keeping things as simple as possible we have restricted most of our discussion so far to decision problems, i.e., questions with a yes/no answer. There are many interesting problems in which the decision signifies the existence of a solution. Here are some examples:

PMATCH (Perfect matching): Given an undirected graph, does it contain a perfect matching, i.e. a collection of edges that covers every vertex exactly once?
CLIQUE: Given an undirected graph G and a number k, does G contain a clique of size k, i.e. a subset of k vertices of which ever pair is connected by an edge?
SAT (Boolean formula satisfiability): Given a boolean formula in conjunctive normal form (CNF), for instance

$$(x_1 \text{ OR } \overline{x_2}) \text{ AND } (x_1 \text{ OR } x_3 \vee \overline{x_4}) \text{ AND } (\overline{x_2} \text{ OR } \overline{x_3}) \text{ AND } (\overline{x_4})$$

is there an assignment that satisfies the formula (i.e. it satisfies all the clauses simultaneously)? For instance the assignment $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$ satisfies the above formula.

As usual we represent inputs to decision problems as binary strings. A decision problem can then be represented either as a function from the set $\{0,1\}^*$ of all possible inputs to $\{0,1\}$, or equivalently as a partition (YES, NO) of $\{0,1\}^*$. For example, the YES instances of PMATCH are those graphs (represented say by a list of indicators for each of the $\binom{n}{2}$ possible edges) that do contain a perfect matchings, while the NO instances are those that do not.

Depending on the encoding we use to represent instances there may be strings that do not represent valid inputs at all. For example, if a graph is represented by an indicator vector for each of its $\binom{n}{2}$ possible edges then strings of length 5 or 13 do not represent any inputs. One possibility is to disqualify such instances by relaxing the requirement that the sets YES and NO partition all possible strings and allow for a third possibility that certain instances are invalid. This leads to the notion of *promise problems*, to which we will come back later in the course. For now we will adopt the convention that invalid inputs be classified as NO instances, and part of the job of the algorithm is to tell whether the input string represents a valid instance.

Search problems In a search problem we not only want to know if a solution exists, but find the actual solution as well:

Given a graph, find a perfect matching, if one exists.
Given a graph G and a number k, find a clique of size k in G, if one exists.

Given a boolean formula, find an assignment that satisfies it, if one exists.

Search problems are represented as *relations* S over pairs of strings $(x, y) \in \{0,1\}^* \times \{0,1\}^*$. The input x represents the problem (the graph) while y represents potential solutions (the perfect matching). The pair (x, y) satisfies S if y is a solution of x (graph x contains the matching y). This definitional choice allows a search problem to have multiple solutions, or none at all.

Now let us jump ahead a little bit and see what kind of insights complexity theory can give us about these problems. We are looking at problems of the following kind: Given some object (a graph, a formula) find if it contains a given structure (a matching, a clique of a given size, a satisfying assignment), and find this structure if it exists. In principle, we can solve all these problems by trying out all possible structures of interest. However, even for moderately sized objects, brute-force search would take too long even for moderately-sized instances.

In 1965 Edmonds found an ingenious way of finding perfect matchings in graphs that does much better than brute-force search. For an n vertex graph, Edmonds' algorithm performs a sequence of about n^3 operations which consist of marking certain edges in the graph and looking at their neighbors, at the end of which the maximum matching is highlighted. With today's technology we can run this algorithm on graphs with hundreds of thousands of nodes, and even faster algorithms have been discovered since then.

In contrast to PMATCH, for CLIQUE and SAT we know of no better algorithm than one that essentially performs an exhaustive search of all possible solutions. Most theoretical computer scientists believe that no amount of ingenuity will yield algorithms for these problems that are substantially faster than brute-force search. This belief is supported by a famous conjecture: P does not equal NP.

2 Models of computation

To reason rigorously about the complexity of computations we need a model that captures realistic computational resources. This seems difficult because technology changes over time, sometimes dramatically. In the early 1800s the mathematician Gauss was admired for his prowess at performing difficult calculations. In the early 1900 mechanical calculating devices put the best humans to shame, and by the 1950s electronic computers were already doing calculations that were infeasible before. Moore's law correctly predicted exponential speedups in processing power in the last 40 years. How can we hope to say anything about computing power that is independent of technology?

Complexity theory takes the perspective that the difference in resources between various computational devices is insignificant compared to the vast gap between the tractable and the intractable. Then it does not really matter what model of computation we use, as long as it can both simulate and be simulated *efficiently* by a realistic computational device.

The *Turing Machine* is an important such model. This is a device with a finite *control unit* and a one-sided *infinite tape* whose cells are populated by symbols from some fixed alphabet that contains the input/output alphabet symbols 0 and 1 and the empty cell symbol $\square$. The control unit consists of a set of states, including a start state and an halting state. The tape is initialized with the input and contains a moving head which initially points to its first bit. At each step the control unit executes a deterministic *transition* that specifies the next state, the content of the active tape cell under the head, and the head position (stay or move one unit left/right) as a function of the current state and the active tape cell content. When the control unit moves to the halting state the contents of the tape (the part not covered by $\square$s) give the output.

The *running time* of a Turing Machine on a given is the number of steps that it takes for the halting state to be reached. The *space* is the index of the rightmost cell accessed during the computation. We say that the machine runs in time $t(n)$ if for every possible input $x \in \{0,1\}^*$, its running time is at most $t(|x|)$ (and similarly for space).

3 P and NP

The convention in complexity theory is to consider a computation efficient if its running time of the computation is bounded by some polynomial of the input length.

There are several reasons for choosing polynomial time as a notion of efficiency. First, we want to do better than exhaustively running through all possible solutions, a procedure that typically takes time exponential in the size of the inputs. Polynomials grow slower than exponentials, so an algorithm that runs in polynomial time must be using something about the structure of the problem that allows it to sidestep brute-force search over all possible solutions. Another reason for using polynomial time as a measure of efficiency is belief in the *Cobham-Edmonds thesis* (also known as the *extended Church-Turing thesis*), which states that any reasonable model of computation can be simulated on any other with a slowdown that is at most polynomial in the size of the input. Finally, polynomial-time computations enjoy nice closure properties: For example, if the output of one polynomial-time algorithm is the input to another algorithm then their composition also runs in time polynomial in the length of the input.

It is important to keep in mind that not all computations that can be done in polynomial time are considered efficient in practice, that there are interesting algorithms whose running time is somewhere between polynomial and exponential, and that the value of the actual exponent in exponential-time algorithms can make a big difference. Refinements of complexity-theory such as *fine-grained complexity* address these phenomena.

For decision problems, polynomial-time computation is captured by this definition:

Definition 1. The class P consists of those decision problems that are solved by some Turing Machine that runs in time $O(|x|^c)$ for every input x and some constant c.

Many decision problems of interest, like PMATCH, CLIQUE, and SAT, are derived from search problems. These search problems have two important properties: their solutions (if they exist) are *short* and they can be *verified* efficiently. A relation S is an NP-relation if it has these two properties:

1. (short solutions) There is a polynomial p such that if $(x, y) \in S$, then $|y| \leq p(|x|)$, and
2. (efficient verification) There is a polynomial-time algorithm that accepts input (x, y) iff $(x, y) \in S$

If $(x, y) \in S$, then y is called a *witness* or *certificate* for x. For example in the SAT problem, x is a boolean formula and y is a satisfying assignment for x; thus y witnesses the fact that x is satisfiable.

The decision version of a search relation S is the decision problem whose YES instances are those x for which there exists a y such that $(x, y) \in S$. (We call such x yes instances of S.) A decision problem is an NP decision problem if it is the decision version of some NP-relation. The class NP consists of all NP decision problems. In other words:

Definition 2. The class NP consists of all decision problems (YES, NO) for which there exists a polynomial-time Turing Machine V called *the verifier* and a polynomial p such that $x \in YES$ if and only if V accepts input (x, y) for some y of length at most $p(|x|)$.

4 NP-completeness

Tolstoy famously said "all happy families are alike, but every unhappy one is unhappy in its own way". In contrast, many difficult problems in NP are difficult in the same way: There is no better way to solve them than exhaustive search. NP-completeness explains why this happens.

For two search problems S and T, we say S *polynomial-time reduces* to T if there exist polynomial-time computable functions *inst* and *sol* such that (1) if x is a yes instance of S then $inst(x)$ is a yes instance of T and (2) if y is a solution of T then $sol(x, y)$ is a solution of S. Polynomial-time reductions are transitive: If S reduces to T and T reduces to U then S also reduces to U. A search problem C is NP-*complete* if C is an NP-relation and every NP-relation polynomial-time reduces to C.

A reduction is a useful tool for proving that search problem T is computationally at least as hard as search problem S. For if we had a polynomial-time algorithm for T we could then also solve S: Given instance x of S, first convert it to instance $inst(x)$ of T, solve $inst(x)$ to obtain solution y, and then apply sol to (x, y) to get back a solution for x.

Polynomials and circuits Before we see some examples of NP-complete problems it will be helpful to relate the computational power of algorithms and circuits. These are objects of different type as the set of possible inputs to an algorithm is infinite, while in the case of circuits inputs have an a priori fixed length. We can, however, simulate an algorithm by an infinite circuit family $\{C_1, C_2, \dots\}$ where circuit n takes inputs in $\{0,1\}^n$.

Theorem 3. *For every Turing Machine M that runs in time at most $t(n)$ and space at most $s(n)$ on all inputs of length n there exists an AND/OR circuit family $\{C_1, C_2, \dots\}$ where $C_n(z) = M(z)$ for every input $z \in \{0,1\}^n$ and C_n has size $O(t(n) \cdot s(n))$.*

A Turing Machine that runs in time t on a given input must also run in space at most t as it moves one cell at a time. In particular if M runs in time polynomial in its input length, C_n also has size polynomial in n: Polynomial-time algorithms can be simulated by polynomial-size circuit families.

In the other direction, it is not true that every polynomial-size circuit family can be simulated by a polynomial-time algorithm. The reason is that the set of functions computed by polynomial-size circuit families is uncountable, while the number of Turing Machines is countable as a Turing Machine is fully described by its finite control unit.

Proof Sketch. The *computation tableau* of M on input x is a table of dimensions $t(n) \times s(n)$ that describes the transcript of the computation: The cell i, j of this tableau contains the contents of the jth cell of the tape at time i, including a special marker that designates the state if the head of the Turing Machine happens to access cell j at time i. (Some of the cells may be marked blank if the Turing Machine takes less time or space.)

We associate a variable z_{ij} with cell (i, j) of the tableau. The value in cell i at time j is then determined by a function

$$z_{i,j} = C_{i,j}(z_{i-1,j-1}, z_{i-1,j}, z_{i-1,j+1})$$

that determines the contents of cell i, j as a function of the contents of the three cells above it (only two if $i = 1$ or $i = t$). Since $C_{i,j}$ is a function from one finite alphabet to another one, independent of the input length, it can be represented by a DNF of constant size. We now create a circuit C by "cascading" the circuits $C_{i,j}$ whose inputs are the first n elements in the top row and the outputs are the relevant elements in the last row. The resulting circuit has size $O(t(n) \cdot s(n))$. □

Some NP-complete problems Our first example of an NP-complete search problem is circuit satisfiability (CSAT): Given a circuit $C\colon \{0,1\}^n \to \{0,1\}$ as input, find an assignment $y \in \{0,1\}^n$ such that $C(y) = 1$ if such an assignment exists.

Theorem 4. CSAT *is* NP*-complete.*

Proof Sketch. Take any NP-relation S with verifier V. We will show that S reduces to CSAT. Let n and m denote the length of the input and solution, respectively, and let C be the circuit that simulates V on input-solution pairs (x, y) of length n and m, respectively, given by Theorem 3. The map $inst(x)$ outputs the circuit $C_x(y) = C(x, y)$ obtained by hardwiring the input x into C and sol outputs the solution y.

If x is a yes instance of S then $C_x(y) = V(x, y)$ must accept for some y, so C_x is a yes instance of CSAT. In the other direction, any solution y such that $C_x(y) = 1$ is also a solution of x with respect to the search problem S. Therefore S reduces to $CSAT$.

It remains to argue that $inst$ and sol are polynomial-time computable. For sol this is clearly true. Regarding $inst$, inspecting the proof of Theorem 3, we can conclude that the circuit C does not merely

exist, but its description can in fact be printed in time polynomial in the running time of V on input (x, y). Since m is polynomially bounded in n, C_x can be computed in time polynomial in the length of x. □

The search problem 3SAT is the same as SAT, but its yes instances are CNFs of width at most 3, namely with at most 3 variables per clause.

Theorem 5 (Cook's Theorem)**.** 3SAT *is* NP*-complete.*

Proof. By transitivity of reductions it is enough to show that CSAT polynomial-time reduces to 3SAT. Given a circuit $C\colon \{0,1\}^n \to \{0,1\}$ the map *inst* produces a CNF ϕ over variables $x_1, \ldots, x_n, y_1, \ldots, y_m$, where the x_is and y_j represent the inputs and gates of C, respectively. For each gate G of C whose inputs are represented by literals a and b and output is represented by literal c, ϕ contains four CNF clauses that are simultaneously satisfied when and only when the constraint $c = G(a, b)$ holds. For example if G is an AND gate then the four clauses are a OR b OR $\overline{c}$, a OR $\overline{b}$ OR $\overline{c}$, $\overline{a}$ OR b OR $\overline{c}$, $\overline{a}$ OR $\overline{b}$ OR c. The map *sol* outputs $x_1, \ldots, x_n$. The formula ϕ also contains the clause y_m, assuming y_m represents the output gate of C.

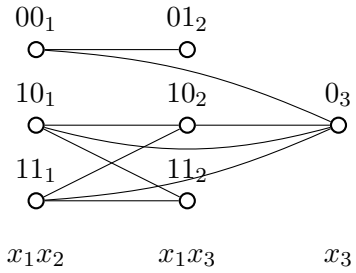
Both *inst* and *sol* are polynomial-time computable. If x is a satisfying assignment for C, and y_i is the value of the corresponding gate when C is evaluated on input x then (x, y) is a satisfying assignment for ϕ. Also, if (x, y) satisfies ϕ then x must be a satisfying assignment of C and y must encode the values at the corresponding gates, so the reduction is correct. □

Theorem 6. CLIQUE *is* NP*-complete.*

Proof. We show that 3SAT polynomial-time reduces to CLIQUE. Given a formula ϕ, *inst*(ϕ) is the following graph G and number k: For each clause c of ϕ with v variables, ϕ contains $2^v - 1$ vertices representing all *satisfying* assignments of c. Such assignments to a clause can also be viewed as partial assignments of ϕ. There is an edge between partial assignments if and only if the two are not inconsistent (they do not assign opposite values to the same variable). The number k is set to the number of clauses in ϕ. For example, if ϕ is the CNF

$$(x_1 \text{ OR } \overline{x_2}) \text{ AND } (x_1 \text{ OR } x_3) \text{ AND } (\overline{x_3})$$

then $k = 3$ and G is the graph



The vertex labels represent assignments to clauses: For example, 10_1 is the partial assignment $x_1 = 1, x_2 = 0$ to the first clause of ϕ. There is no edge between 00_1 and 10_2 because their assignments to x_1 are inconsistent.

Given a clique C of size k in G, *sol*(ϕ, C) outputs an assignment to ϕ that is consistent with all the partial assignments in the clique. In the above example, if C is the clique $\{10_1, 10_2, 0_3\}$ then *sol* would output the assignment $x_1 = 1, x_2 = 0, x_3 = 0$.

Both *inst* and *sol* are computable in polynomial time. We now argue correctness. If ϕ has a satisfying assignment x and for each clause we choose the vertex indexed by the assignment x then the corresponding set of vertices is a clique of size k in G. In the other direction, if C is a clique of size k in G then C must include one vertex for each clause of ϕ and the assignments represented by these vertices must be consistent. So *sol*(ϕ, C) is a satisfying assignment for ϕ. □

By Theorem 6, if we can design a polynomial-time algorithm for finding cliques in graphs then we would also obtain polynomial-time algorithms for finding satisfying assignments to CNFs of width 3. We do not know of such algorithms. The best currently known algorithms for 3SAT take time $2^{\Omega(n)}$ in the worst case. From the proof of Theorem 6 it follows that the existence of an algorithm for $CLIQUE$ on n vertex graphs that runs in time $2^{o(n)}$ would imply the existence of an algorithm for 3SAT that runs in time $2^{o(n)}$, where n is the number of variables. This is one reason why the existence of a $2^{o(n)}$ algorithm for $CLIQUE$ (and by Theorem 3 a circuit family for $CLIQUE$ of the same order of growth) is thought to be unlikely.

Search versus decision for NP **problems** Suppose we have an algorithm that finds perfect matchings in graphs that are guaranteed to have one. We can use this algorithm to decide if a perfect matching in a graph exists: Run the algorithm and accept if the solution checks out. This reasoning holds for SAT, CLIQUE, and every problem in NP: If we can solve any NP search problem in polynomial time, we can also solve its decision variant.

What about the other direction? There are examples of problems in NP for which the decision version is easy but we do not know of any efficient algorithms for finding solutions. Consider for example the search problem given by the relation S given by

$$(x, (y, z)) \in S \text{ if } x, y, z \text{ are integers greater than 1 such that } x = yz.$$

To solve the decision version of S we need to find out if x is composite and there is a polynomial algorithm for this. To solve the search problem we need to factor x and no polynomial-time algorithm for this is known.

However, if any NP-complete problem can be *decided* in polynomial time then solutions to all NP problems can also be *found* in polynomial time:

Theorem 7. *If* NP = P *then all* NP *search problems have polynomial-time algorithms.*

Proof. If NP = P then in particular CSAT has a polynomial-time decision algorithm D. Then CSAT also has the following polynomial-time search algorithm S: On input $C\colon \{0,1\}^n \to \{0,1\}$ obtain the circuits $C_0, C_1\colon \{0,1\}^{n-1} \to \{0,1\}$ by substituting the values 0 and 1 for x_1 in C, respectively. Run D on inputs C_0 or C_1. If D accepts C_0, set x_1 to 0 and recursively find a satisfying assignment for C_0. Otherwise, set x_1 to 1 and recursively find a satisfying assignment for C_1. If D runs in polynomial time so does S, and if C has a satisfying assignment then so must at least one of C_0 and C_1 and S is guaranteed to find it by an inductive argument.

By Theorem 4 all NP search problems polynomial-time reduce to CSAT, so if NP = P then all such problems have polynomial-time algorithms. □

5 Randomized algorithms

One resrictive aspect of the definition of polynomial time is that the algorithm does not have access to random bits. A *randomized Turing Machine* receives, in addition to its input x, a sequence of independent random bits $r_1, r_2, \cdots \sim \{0,1\}$. For every input x the output of the Turing Machine is now a random variable $M(x)$ that depends on the contents r of the random tape. When we want to make this dependence on r explicit we will write $M(x;r)$. We will say that a randomized Turing Machine runs in time at most t on a given input if its running time is at most t for all possible instantiations of r.

As for sublinear-time algorithms, randomized Turing Machines for decision problems can be of the Las Vegas or Monte Carlo type. We'll focus on the more general Monte Carlo type algorithms. This type of algorithm is required to answer correctly on all inputs most of the time.

Definition 8. Randomized Turing machine M decides f if for every input x, $\Pr[M(x) = f(x)] \geq 2/3$, where the probability is taken over the setting of the random tape of M. The class BPP consists of all decision problems f that are decided by some randomized polynomial-time Turing Machine.

We will soon see that the choice of the constant 2/3 is irrelevant; choosing any number strictly between 1/2 and 1 does not change the definition. If we replace 2/3 with 1/2 then the definition becomes meaningless as it is satisfied by any f: M can output a random coin flip as its answer. If we replace 1/2 with 1 then M is required to always output the correct answer so it can pretend the value of its random tape is fixed say to zero. So in particular all (decision) problems in P are also in BPP.

There are not too many examples of decision problems for which we know randomized efficient algorithms but no equivalent deterministic algorithm. We'll explain why next time. One important example is polynomial identity testing.

Polynomial identity testing An *arithmetic formula* over the integers is an expression built up from the variables $x_1, \ldots, x_n$, the integers, and the arithmetic operations $'+'$ and $'\times'$, for instance:

$$(x_1 + 4 \times x_3 \times x_4) \times \big((x_2 - x_3) \times (x_2 + x_3)\big) - 3 \times x_2.$$

We say that F is identically zero, in short $F \equiv 0$, if when we expand the formula and carry out all cancellations we obtain 0.

Polynomial identity testing (PIT) is the following decision problem: Given an arithmetic formula F with integer coefficients, decide if F is identically zero. The above formula is not identically zero, while this one is:

$$(x_1 + x_2) \times (x_1 + x_2) - (x_1 - x_2) \times (x_1 - x_2) - 4 \times x_1 \times x_2$$

Theorem 9. PIT *has a randomized polynomial-time algorithm (i.e., it is in* BPP*).*

Proof. Let m be the multiplication depth of F (the maximum number of $\times$ gates on all root-to-leaf paths). Consider the following randomized algorithm for PIT:

A: On input F,
Choose values $a_1, \ldots, a_n$ independently at random from the set $\{1, \ldots, 3m\}$.
Evaluate $b = F(a_1, \ldots, a_n)$.
If $b \neq 0$, reject; otherwise, accept.

If F is identically zero, A accepts F with probability 1. Now we show that if F is not identically zero then A rejects F with probability at least 1/3. This follows directly from the next lemma and the fact that an arithmetic formula with m multiplications computes a polynomial of degree at most m.

Lemma 10 (De Millo, Lipton, Schwarz, Zippel)**.** *For any nonzero polynomial p of degree d over the integers and every set $S \subseteq \mathbb{Z}$,*

$$\Pr_{a_1,\ldots,a_n \sim S}[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}$$

In our case, F is a polynomial of degree at most m and S is a set of size $2m$, so the algorithm will detect that $F \not\equiv 0$ with probability at least 1/2. □

Proof of Lemma 10. We prove it by induction on n. If $n = 1$ then p is a nonzero univariate polynomial of degree d. Such a polynomial can have at most d roots so if a was chosen at random from a set S, $p(a)$ equals zero is at most $d/|S|$. If $n > 1$ then we can expand p as

$$p(x_1, \ldots, x_n) = x_1^{d_1} \cdot p_1(x_2, \ldots, x_n) + x_1^{d_1 - 1} \cdot p_2(x_2, \ldots, x_n) + \ldots + p_k(x_2, \ldots, x_n)$$

and consider two possibilities: Either $p_1(a_2, \ldots, a_n)$ evaluates to zero or it doesn't. By the law of conditional probabilities,

$$\Pr[p(a_1, \ldots, a_n) = 0] \leq \Pr[p_1(a_2, \ldots, a_n) = 0] + \Pr[p(a_1, \ldots, a_n) = 0 \mid p_1(a_2, \ldots, a_n) \neq 0]$$

By the induction hypothesis, the first term is at most $(d - d_1)/|S|$. For the second one, conditioned on $p_1(a_2, \ldots, a_n)$ being nonzero, for any fixing of $a_2, \ldots, a_n$, $p(x_1, a_2, \ldots, a_n)$ is a univariate polynomial of

degree d_1. By our analysis of the case $n = 1$, p vanishes with probability at most $d_1/|S|$ over the choice of x_1. Therefore

$$\Pr[p(a_1, ..., a_n) = 0] \leq \frac{d - d_1}{|S|} + \frac{d_1}{|S|} = \frac{d}{|S|}. \qquad \square$$

Reducing the failure probability We now argue that the quantity that determines the failure probability of randomized algorithms for decision problems can be made arbitrarily small. In fact, any "polynomial" gap between the acceptance probability of yes and no instances can be amplified to a gap that is exponentially close to 1.

Theorem 11. *Let (YES, NO) be a decision problem and A be a polynomial-time algorithm such that*

$$\begin{aligned} x \in YES \quad &\longrightarrow \quad \Pr[A(x) = 1] \geq c(|x|) \\ x \in NO \quad &\longrightarrow \quad \Pr[A(x) = 1] \leq s(|x|), \end{aligned}$$

$c(n) - s(n) \geq 1/q(n)$ for some polynomial q, and the values $c(n), s(n)$ are computable in time polynomial in n. Then for every polynomial p there exists a polynomial-time algorithm A' such that

$$\begin{aligned} x \in YES \quad &\longrightarrow \quad \Pr[A'(x) = 1] \geq 1 - 2^{-p(|x|)} \\ x \in NO \quad &\longrightarrow \quad \Pr[A'(x) = 1] \leq 1 - 2^{-p(|x|)}. \end{aligned}$$

We now prove the theorem. On input x, the algorithm A' runs A independently for sufficiently for a number of times m to be determined later and keeps track of what fraction of the runs accept. If this fraction is closer to $c(n)$ than it is to $s(n)$ then A' accepts x, otherwise it rejects x.

Let X_i be an indicator random variable for the event that the ith run accepts and $X = X_1 + \cdots + X_m$. If x is a yes instance then the expected value $\mathrm{E}[X]$ is at least $m \cdot c(n)$, and if x is a no instance then $\mu \leq m \cdot c(n)$.

The Chernoff bound tells us that if m is large, then X is very close to its expectation with extremely high probability. There are several variants of this bound and we will apply the following one:

Theorem 12 (Chernoff bound)**.** *Let $X_1, X_2, \cdots, X_m$ are independent indicator random variables such that $\Pr[X_i = 1] = p$ for all i. Then for every $\varepsilon > 0$,*

$$\Pr\big[|X - pm| \geq \epsilon m\big] \leq 2e^{-2\epsilon^2 m}$$

We set $\epsilon = (c(n) - s(n))/2$ to obtain the following consequence:

$$\begin{aligned} x \in YES \quad &\longrightarrow \quad \Pr[X \leq (c(n) + s(n))m/2] \leq 2e^{-\delta(n)m} \\ x \in NO \quad &\longrightarrow \quad \Pr[X \geq (c(n) + s(n))m/2] \leq 2e^{-\delta(n)m} \end{aligned}$$

where $\delta(n) = (c(n) - s(n))^2/2 \geq 1/2q(n)^2$. Setting $m = 2p(n)q(n)^2 + 1$ gives the desired conclusion.

Randomized algorithms and circuits In the proof of Theorem 3 we showed an efficient simulation of deterministic algorithms by circuit families. In the setting of decision problems, randomized algorithms can also be efficiently simulated by circuit families.

Theorem 13 (Adleman's theorem)**.** *Every (decision) problem in* BPP *has a polynomial-size circuit family.*

Proof. Let f be a decision problem in BPP and M be the polynomial-time randomized Turing Machine that decides it. By Theorem 11, we may assume without loss of generality that for every $x \in \{0,1\}^*$,

$$\Pr_r[M(x; r) \neq f(x)] < 2^{-|x|}.$$

By Theorem 3 there exists a polynomial-size circuit family $\{C_1, C_2, \dots\}$ such that $C_m(x, r) = M(x; r)$ for every input x and setting of the random tape r and $m = |x| + |r|$. So for every x, $C_m(x; r)$ differs from $f(x)$ with probability less than 2^{-n} where n is the length of x.

Let's now fix an input length n. By a union bound, the probability that $C_m(x; r)$ differs from $f(x)$ for some x of length n is less than $2^n \cdot 2^{-n} = 1$ over the choice of r. In particular, it follows that for every n there exists a choice of r such that $C_m(x; r) = f(x)$ for all x of length n. If we fix this r into the circuit C_m, we obtain a new polynomal-size circuit family that decides f on all inputs. □

References

The material in this lecture can be found in most textbooks on complexity theory. A more pedagogical presentation of Turing Machines, circuits, and reductions can be found in the textbook *Introduction to the theory of computation* by Michael Sipser. Our definition of reduction between search problems is due to Levin and is technically different from the notion of a Karp reduction between decision problems from the textbook, but the two are closely related.