A central goal of computational complexity is to understand the amount of time necessary to solve various computational problems. In the 1960s it was realized that some seemingly simple tasks would take an inordinate amount of time to solve on any computer, even though solutions can in principle be found. The inherent difficulty of these problems had nothing to do with the computing technology that was available in the 1960s. With today's technology we can do no better, and it is quite possible that in a hundred years time we will still be stuck. It appears that nature imposes intrinsic obstacles at performing certain computations.

1 Computational problems

There are several different types of problems that are interesting from the computational point of view. In the next few lectures we will focus on decision problems and search problems. (There are also other types like optimization problems, counting problems, sampling problems, and games.) Let's see some examples.

Decision problems   A decision problem is one that can be answered by yes or no:

PMATCH 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 as a *partition* (YES, NO) of the set $\{0,1\}^*$ of all possible inputs. 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.

An issue that arises with this representation is that it is not obvious how to classify instances that do not represent valid inputs at all. (In our graph example, these would be inputs whose length is not a number of the form $\binom{n}{2}$.) 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 of the decision problem.

A partition (YES, NO) of the set $\{0,1\}^*$ of all possible inputs can also be viewed as a function $f: \{0,1\}^* \to \{0,1\}$ where $f(x) = 1$ if $x \in YES$ and 0 if $x \in NO$. So decision problems can also be represented by functions from $\{0,1\}^*$ to $\{0,1\}$.

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

PMATCH Given a graph, find a perfect matching, if one exists.

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

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

We will represent search problems 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).

Now let us jump ahead a little bit and see what kind of insights complexity theory can give us about these problems. At a very general level, we are looking at problems of the following kind: We are given some object (a graph, a formula) and want to check if the object contains structures of a certain type (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 this would take too much time even on very fast computers.

In 1965 Edmonds found an ingenious way of finding perfect matchings in graphs that takes much less effort than looking at all possible matchings in the graph. 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.

In contrast to the matchings, for clique and formula satisfiability we know of no better algorithm than one that essentially performs an exhaustive search of all possible solutions. Most computer scientists believe that no amount of ingenuity will help us here. Rather, it is a fundamental fact of our current state of knowledge or a fundamental limitation imposed by nature (depending on who you ask) that these problems do not admit algorithms that substantially outperform exhaustive search in the worst case. Is this really true?

This is the famous P versus NP question, the most famous problem in complexity theory, and one that we will not be able to answer. Our goal will be more modest: We will try to understand the relationships between problems like matching, clique, and satisfiability with respect to different solution concepts (decision versus search), complexity measures (worst-case versus average-case), and algorithms (deterministic, randomized, quantum).

2 Models of computation

To reason rigorously about the complexity of computations we need a model that faithfully captures the resources that we have at our disposal. This appears somewhat difficult as the resources we have available change dramatically over time. 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 noone thought possible before. Today a mobile phone can do much much more than a computer from the 1990s. Given such huge disparities among the devices at our disposal, how can we hope to make any sort of general statement about computation?

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 is a reasonable one

in the sense that it can both simulate and be simulated by a realistic computational device in a relatively efficient way.

A standard model which is convenient for most purposes in complexity theory is the Turing Machine. 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 □). which is initialized with the input and terminates with the output. The tape contains a moving head, and at each step of computation the machine decides what to do (e.g., move to a different control state, move the head left or right, modify the current cell of the work tape) as a function of its current control state and the content of the tape cell that the head points to. At some point the machine goes into a special halting state and the computation is over. The *computation time* of a Turing Machine on a given is the number of steps that it takes for the halting state to be reached. The *computation space* is the index of the rightmost cell of the tape accessed during the computation.

We say that a Turing Machine solves decision problem f if at the end of the computation the tape contains the value $f(x)$ (1 if x is a *YES* instance, 0 if it is a *NO* instance). A Turing Machine solves search problem S if for every input x the machine outputs a y such that $(x, y) \in S$ provided such a y exists.

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 in question that allows it to sidestep the "brute force" procedure of going through all possible solutions. Another reason for the popularity of polynomial time as a measure of efficiency is a 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, however, to keep in mind that not all computations that can be done in polynomial time are considered efficient in practice, but also that there are interesting algorithms whose running time is somewhere between polynomial and exponential.

For decision problems, computation with worst-case polynomial time complexity is captured by the class P: 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.

A relation S is an NP-relation if it is efficiently computable and it describes a search problem whose solutions are short (if they exist). In other words,

1. There is a polynomial-time algorithm that, on input (x, y), decides if $(x, y) \in S$, and

2. There is a polynomial p such that if $(x, y) \in S$, then $|y| \leq p(|x|)$.

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 1. The class NP consists of all decision problems (YES, NO) for which there exists a polynomial-time Turing Machine V 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 $p(|x|)$.

We call Turing Machine V *the verifier*. Its task is to verify that the NP-relation is satisfied by the instance x and the candidate solution y.

4   NP completeness

Many problems in the class NP appear to be difficult in the same way: There is no better way of finding a solution than doing an exhaustive search over all possible witnesses. NP-completeness gives a partial explanation of this phenomenon. We will focus on search problems.

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 2. *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 exist.

Theorem 3. 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 2. The map $inst(x)$ outputs the circuit $C_x(y) = C(x,y)$ obtained by hardwiring the input x into C and sol simply 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 2, 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 4. 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 $\bar{c}$, a OR $\bar{b}$ OR $\bar{c}$, $\bar{a}$ OR b OR $\bar{c}$, $\bar{a}$ OR $\bar{b}$ 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. $\qquad\square$

Theorem 5. 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
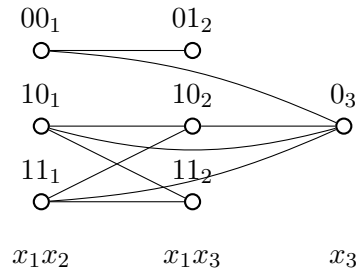
The vertex labels represent assignments to clauses: For example, 10_1 is the partial assignment $x_1 = 1, x_2 = 0$ to the first clause of ϕ. There is no edge between 00_1 and 10_2 because their assignments to x_1 are inconsistent.

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. $\qquad\square$

By Theorem 5, 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, which is a very well studied problem, take time $2^{\Omega(n)}$ in the worst case. From the proof of Theorem 5 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 2 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 6. *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 3 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 aspect in which our model of an efficient algorithm is too restrictive is that it does not allow for the use of randomness. The following extension gives the Turing Machine access to an ideal uniform source of random numbers.

A *randomized Turing Machine* contains, in addition to its other tapes, a special *random tape* which is initialized with a sequence of random and uncorrelated bits $r_1, r_2, \cdots \sim \{0, 1\}$. As soon as we allow the algorithm access to a random tape, its behavior is no longer completely determined by the input x. Now for every x the output of the Turing Machine is 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 settings of the random tape.

For decision problems, we will think of a randomized algorithm as solving the problem (in the worst case) as long as it behaves correctly on *all* inputs for "most" choices of the randomness. There are several possible definitions depending on how we require the algorithm to behave on the bad choices. We will adopt the most liberal one leading to the notion of bounded-error probabilistic polynomial time.

Definition 7. We say that randomized Turing machine M decides decision problem f if for every $x \in \{0,1\}^*$,

$$\Pr[M(x) = f(x)] \geq 2/3$$

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 and in fact any number strictly between $1/2$ and 1 leads to the same class of problems. If we replace $2/3$ with $1/2$ then the definition becomes meaningless as it is satisfied by any f (as 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 algortihms but no equivalent deterministic algorithm. The simplest (and most famous) 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 8. PIT *has a randomized polynomial-time algorithm (i.e., it is in* BPP*).*

Proof. Let m be the number of multiplications in F. 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 9 (De Millo, Lipton, Schwarz, Zippel). *For any nonzero polynomial p of degree d over the integers and every set $S \subseteq \mathbb{Z}$,*

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

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 $1/2$. $\qquad\square$

Proof of Lemma 9. 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

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

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

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

By the induction hypothesis, the first term is at most $(d - d_1)/|S|$. For the second one, conditioned on $p_1(a_2, ..., a_n)$ being nonzero, for any fixing of $a_2, \dots, a_n$, $p(x_1, a_2, \dots, a_n)$ 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 10. *Let (YES, NO) be a decision problem and A be a polynomial-time algorithm such that*

$$x \in YES \quad \longrightarrow \quad \Pr[A(x) = 1] \geq c(|x|)$$
$$x \in NO \quad \longrightarrow \quad \Pr[A(x) = 1] \leq s(|x|),$$

$c(n) - s(n) \geq 1/q(n)$ 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

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

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 $X = X_1 + \dots + X_m$. 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 11 (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:

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

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 2 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 12. *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 10, 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 2 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 the randomness r into the circuit C_m, we obtain a new polynomal-size circuit family that decides f on all inputs. $\qquad\square$

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.