E0 224: Computational Complexity Theory

Naman Mishra

August 2025

Contents

Lectures

The course

Course website
Instructor: Chandan Saha
Time: Mondays and Wednesdays, 11:30–13:00
Room: CSA 252

Lecture 1.
Wednesday
August 6

Syllabus:

- P, NP and NP-completeness
- Space complexity
- Polynomial time hierarchy
- Boolean circuits
- Randomized computation
- Complexity of counting
- Basics of hardness of approximations

Resources:

(1) Computational Complexity: A Modern Approach by Sanjeev Arora and Boaz Barak
(We'll closely follow this book)

(2) Computational Complexity Theory by Steven Rudich and Avi Wigderson (Editors)

(3) Mathematics and Computation by Avi Wigderson

(4) Boolean Function Complexity by Stasys Jukna

(5) Gems of Theoretical Computer Science by Schoening and Pruim

(6) The Nature of Computation by Moore and Mertens

(7) The Complexity Theory Companion by Hemaspaandra and Ogihara

(8) Online lecture notes... (take a look at this webpage)

Evaluation:

(45%) Three assignments, one posted at the end of each month, with two weeks for completion. The submission will be via email, as a LaTeX-generated PDF file. You may freely use all resources and tools, and confer with each other, so long as you list these out.

(25%) Midterm exam

(30%) Final exam

Classify computational problems based on the amount of resources required by algorithms to solve them.

Problems

Problems come in various flavors.

Decision problem

- Is n a prime number?
- Is a vertex t reachable from a vertex s in a given graph G?

Search problem

- Search for a prime between n and $2n$.
- Find a path from s to t in graph G.
- Find a satisiying assignment for a Boolean formula.

Counting problem

- Count the number of cycles in a graph.
- Count the number of perfect matchings in a graph.

Optimization problem

- Find a minimum size *vertex cover* in a graph.
- Optimize a linear function subject to *linear inequality constraints*. (Linear programming)

Lecture 1: Turing machines

Algorithms

Algorithms are methods for solving problems, studied via formal *models of computation*, such as Turing machines.

Resources

- **Time:** Number of bit operations
- **Space:** Number of memory cells required
- **Randomness:** Number of random bits used
- **Communication:** Number of bits sent over a network

Roadmap

Structural complexity The classes P, NP, coNP, NP-completeness, et cetera. The computation must be space bounded. There is an entire polynomial hierarchy. huh?

- How hard is it to check if the largest independent set in G has size k?
- How hard is it to check if there is a circuit of size k that computes the same Boolean function as a given Boolean circuit?

Circuit complexity The internal workings of an algorithm can be viewed as a *Boolean circuit*, yet another nice combinatorial model of computation closely related to Turing machines. The size, depth and width of a circuit correspond to the sequential, parallel and space somplexity, respectively, of the algorithm it represents.

Proving $\mathsf{P} \neq \mathsf{NP}$ also reduces to showing circuit lower bounds, that is, showing the existence of Boolean functions that are hard to compute by small circuits.

Randomness We get probabilistic complexity classes such as BPP, RP, coRP, et cetera.

Access to random bits can help improve computational complexity, but to what extent? We know that Quicksort has expected running time $\Theta(n \log n)$, but worst-case time $\Theta(n^2)$. Can SAT be solved in polynomial time using randomness?

Lecture 1: Turing machines

Fact .1 (Schoening99)**.** $\mathsf{3SAT}$ *can be solved in* $O((4/3)^n)$ randomized *time.*

Brute force takes $O(2^n)$ time, and the state-of-the-art for randomized algorithms is approximately $O(1.307^n)$ time.

Counting complexity The class $\#\mathsf{P}$.

- How hard is it to count the number of perfect matchings in a graph?
- How hard is it to count the number of cycles in a graph?
- Can we compute the number of simple paths between vertices s and t in G efficiently?

In general, is counting comparable to or much harder than deciding?

Approximation A hardness of approximation result looks like the following.

Theorem .2 (Hastad, 1997)**.** *If there exists, for some* $\varepsilon > 0$*, a polynomial-time algorithm to compute an assignment that satisfies at least* $7/8 + \varepsilon$ *fraction of the clauses of an input* $\mathsf{3SAT}$*, then* $\mathsf{P} = \mathsf{NP}$*.*

In contrast, there is a polynomial-time algorithm to compute an assignment that satisfies at least $7/8$ fraction of the clauses.

Another example is that of probabilistically checkable proofs (PCPs).

Chapter I

Turing machines

Turing called them a-machines (automatic machines). Church, his doctoral advisor, named them after him.

A turing machine consists of memory tape(s) and a finite set of rules.

Definition I.1. A k-tape Turing machine M is described by a tuple (Γ, Q, δ) such that

(1) M has k one-sided memory tapes (input/work/output) with *heads*;

(2) Γ is a finite alphabet, including a special *blank* symbol $\flat$. Each memory cell contains an element of Γ.

(3) Q is a finite set of *states* with two special states: q_0 and q_∞.

(4) δ is a function from $Q \times \Gamma^k \times \{-1, 0, 1\}^k$.

The starting configuration is assumed to contain the inut string on the input tape at its beginning, following by trailing $\flat$ symbols. All other tapes contain only $\flat$s. The heads are all positioned at the beginning of each tape.

A step of computation is performed by applying δ. Once the machine enters the state q_∞, it halts computation.

Let $f\colon \{0,1\}^* \to \{0,1\}^*$, $T\colon \mathbb{N} \to \mathbb{N}$ and M a Turing machine on the alphabet $\{0, 1\}$.

Definition I.2 (Computation and running time)**.** M computes f if for every $x \in \{0,1\}^*$, M halts with $f(x)$ on its output tape once began with x on its input tape.

This computation is in T time if for every $x \in \{0,1\}^*$, M halts within $T(|x|)$ steps.

In this course, we will almost always deal with Turing machines that halt on every input, and computational problems that can be solved by a Turing machine. Can all computational problems be solved by some Turing machine? No, since the set of all Turing machines is countable but $\mathbb{N}^{\mathbb{N}}$ is not. Neither is $2^{\mathbb{N}}$, so there exist decision problems which cannot be solved via Turing machines. A more natural example is

Fact I.3 (DPRM70)**.** *There is no algorithm, realizable by a Turing machine, that decides whether a given Diophantine equation admits an integral solution.*

Finding such an algorithm was Hilbert's tenth problem.

Definition I.4 (Time constructible function)**.** A function $T \colon \mathbb{N} \to \mathbb{N}$ is *time constructible* if $T(n) \geq n$ and there is a Turing machine that computes the function $x \mapsto T(|x|)$ (expressed in binary) in $O(T(|x|))$ time.

For example, $(\cdot)^2$, $2^{\cdot}$ and $(\cdot)\log(\cdot)$ are all time constructible.

I.1 Robustness

Theorem I.5 (Binary alphabets suffice)**.** *Let $f \colon 2^* \to 2^*$ and $T \colon \mathbb{N} \to \mathbb{N}$ be a time constructible function.*

Denote the length of the input by n. If a Turing machine M over an alphabet Γ computes f, then there exists another Turing machine M' that computes f in time $4\log_2|\Gamma|T(n)$ using the alphabet $\{0, 1, \flat\}$.

Theorem I.6 (One tape suffices)**.** *Let $f \colon 2^* \to 2^*$ and $T \colon \mathbb{N} \to \mathbb{N}$ be a time constructible function.*

Denote the length of the input by n. If a Turing machine M with k tapes computes f in $T(n)$ time, then there exists a Turing machine M' with 1 tape that computes f in $5kT(n)^2$ time.

One potential way to do this would be to compute $T(n)$, and allocate $T(n)$ space for each tape, one after the other. For each step in the k-tape machine, we may need to traverse $kT(n)$ cells, and do this for $T(n)$ steps.

I.2 Universal Turing machine

Every Turing machine can be represented by a finite string over $\{0, 1\}$. Conversely, every string over $\{0, 1\}$ represents some Turing machine, by mapping initially invalid representations to the trivial Turing machine. Finally, if we allow padding with zeroes, each Turing machine has infinitely many representations. For a binary string α, we will let M_α denote the Turing machine encoded by it.

Theorem I.7 (Universal Turing machine)**.** *There exists a Turing machine U that computes $M_\alpha(x)$ for every input $\alpha|x$.*

Further, if M_α galts within T steps, U halts within $CT\log T$ steps, where C depends only on M_α.

Modern day electronic computers are physical realizations of universal Turing machines.

Lecture 2. Monday August 11

For a while, we will focus primarily on decision problems. A decision problem may be phrased as a Boolean function $f\colon \{0,1\}^* \to \{0,1\}$ or as a language over $\{0,1\}^*$, where $f \leftrightarrow \{s \in \{0,1\}^* : f(s) = 1\}$.

Definition I.8 (Decision)**.** We say that a Turing machine M *decides* a language $L \subseteq 2^*$ if M computes the indicator function $\mathbf{1}_L$ of L.

Unless otherwise stated, n will always denote the size of the input.

Definition I.9 (P)**.** Let $T\colon \mathbb{N} \to \mathbb{N}$. A language L is in $\mathsf{DTIME}(T(n))$ if there is a Turing machine that decides L in time $O(T(n))$.

The complexity class P is defined to be

$$\mathsf{P} := \bigcup_{c\in\mathbb{N}} \mathsf{DTIME}(n^c).$$

Examples.

- Cycle detection: detect if a given graph has a cycle.
- Solvability of a system of linear equations: Gaussian elimination.
- Perfect mathing [Edmonds65]: Check if a given graph has a perfect matching. This paper laid foundation for the class P, which Edmonds called "algebraically increasing" with input size.
- Planarity testing [HopcroftTarjan74]: Check if a given graph is planar.
- Primality testing [AKS02]

Definition I.10 (Polynomial-time)**.** A Turing machine M is a polynomial-time Turing machine if there is a polynomial function $q\colon \mathbb{N} \to \mathbb{N}$ such that for every input x, M halts on x within $q(|x|)$ time.

Definition I.11 (FP)**.** We say that a problem or a function $f\colon 2^* \to 2^*$ is in FP if there is a polynomial-time Turing machine that computes f.

Lecture 2: P, NP and NP-completness

Examples.

- Greatest common divisor: Euclid's algorithm
- Counting path in a DAG: Find the number of paths between two vertices in a directed acyclic graph. Perform a breadth-first search.
- Maximum matching [Edmonds65]: Find a maximal matching in a given graph. That is, a maximal set of edges such that no two edges are incident on the same vertex.
- Linear programming [Khachiyan79,Karmakar84]: Optimize a linear objective function subject to linear inequality constraints.
- Polynomial factoring [LenstraLenstraLovasz82]: Compute the irreducible factors of a univariate polynomial over $\mathbb{Q}$.

It is now known if linear programming has a *strongly* polynomial-time algorithm.

Read about the differences between weakly, strongly and pseudo polynomial-time.

I.3 NP

Solving a problem is generally harder than verifying whether a conjectured solution is indeed one.

Definition I.12 (NP)**.** A language $L \subseteq 2^*$ is in NP if there is a polynomial function $p\colon \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M (called the *verifier*) such that for every input x,

$$x \in L \iff \text{there exists a } u \in 2^{p(|x|)} \text{ such that } M(x|u) = 1.$$

Such a u if called a *certificate* or *witness* with for $x \in L$ with respect to L and M.

Examples.

- Vertex cover: Given a graph G and an integer k, check if G has vertex cover of size k.
- 0/1 integer programming: Given a ststem of linear inequalities, check if there exists a 0-1 assignment to the variables that satisy all the inequalities.

- Integer factorization: Given two numbers n and U, check if n has a prime factor less than or equal to U. The certificate is a (prime) number $p \leq U$ that divides n. The verifier only needs to check that p is prime, the inequality holds, and that it divides n.

  Is it necessary to check that p is prime?

- Graph isomorphism: Given two graphs, check if they are isomorphic.
- 2-Diophantine solvability: Given three integers a, b and c, check if there is an integer solution to $ax^2 + by + c = 0$.

  Hint: descent.

  Proof. The natural certificate is a satisfying pair (x, y). It is allowed to be of size $\mathrm{poly}(\log|a|, \log|b|, \log|c|)$. The case when $c = 0$ is trivial, as is when $a = 0$. If $b = 0$, the only possible solution has size $\Theta(\log|c/a|)$, which is also acceptable.

  The certificate size is $\log|x| + \log|y|$, give or take 4 bits. Thus $|xy|$ is allowed to be of size $2^{\mathrm{poly}(\log|ab|)}$.

  Given a solution (x, y), $(x + b, y - a(2x + b))$ is also a solution. Thus there exists a solution (x, y) with $0 \leq x < b$. In this case x takes up $\log|b|$ bits, and $|y|$ is at most $|c| + |ab^2|$. Thus $\log|y| \in \mathrm{poly}(\log|abc|)$. ■

$\mathsf{P} \subseteq \mathsf{NP}$ since the machine M deciding the problem itself gives verifier, which simply discards the witness and runs M on the input.

Read the survey "The history and status of the P versus NP problem".

I.4 Reductions

Definition I.13 (Karp reduction)**.** We say that a language $L_1 \subseteq 2^*$ is *polynomial-time reducible* or *Karp reducible* to a language $L_2 \subseteq 2^*$ if there is a polynomial-time computable function f such that $x \in L_1 \iff f(x) \in L_2$. We denote this by $L_1 \leq_p L_2$.

Lecture 3. Wednesday August 13

Note that $\leq_p$ is reflexive and transitive.

Exercise I.14. *Let $L_1 \subseteq 2^*$ be any language and $L_2 \in \mathsf{NP}$. If $L_1 \leq_p L_2$, then $L_1 \in \mathsf{NP}$.*

Proof. Let M_2 be a verifier for L_2. We can construct a verifier M_1 for L_1 as follows. M_1 takes x and a witness u, and calls M_2 with $f(x)$ and witness u.

Our hypotheses are that for some polynomials p and q,

- $x_2 \in L_2 \iff \exists u \in 2^{q|x_2|}$ such that $M_2(x_2, u) = 1$; and
- $x \in L_1 \iff f(x) \in L_2$ where $f(x)$ is computable in time $p|x|$.

The runtime of M_1 is clearly polynomial, and

$$M_1(x, u) = 1 \iff M_2(f(x), u) = 1 \iff f(x) \in L_2 \iff x \in L_1.$$

It remains to check that the witness u corresponding to $x_2 = f(x)$ has size polynomial in $|x|$. This is because $f(x)$ has size at most $p|x|$, so u has size at most $(q \circ p)|x|$. ■

The above proof without the witness components easily shows that $\leq_p$ obeys transitivity (making it a preorder).

Definition I.15 (NP-hardness)**.** A language L is NP-*hard* if it is an upper bound for NP under $\leq_p$. L is NP-*complete* if it is NP-hard and lies in NP.

Observe that if an NP-hard problem is in P, then P = NP. Moreover, an NP-complete problem is in P *if and only if* P = NP.

Most examples of NP that we discussed are known to be NP-complete. These are vertex cover, 0/1 integer programming, 3-coloring planar graphs, and 2-Diophantine solvability.

Integer factorization is not believed to be NP-complete.

Graph isomorphism was shown in [Babai15] to be *Quasi*-P. Specifically, an algorithm with runtime $2^{O(\log^3 n)}$ was developed.

Theorem I.16. *There exists an* NP*-complete problem.*

Proof. Define

$$L^\dagger := \{(\alpha, x, 1^m, 1^t) : \exists u \in 2^m \text{ such that } M_\alpha \text{ accepts } (x, u) \text{ in } t \text{ steps}\}.$$

Why is this in NP? The witness u in the description of $L^\dagger$ has length less than the input, and can be verified by simulating M_α on (x, u) for t steps, every step of which is polynomial in the input length.

Why is it NP-hard? Let $L \in$ NP. Then there exists a verifier M with runtime p and a polynomial q such that

$$x \in L \iff \exists u \in 2^{q|x|} \text{ such that } M(x, u) = 1.$$

Let α be the encoding for M. Then

$$x \in L \iff (\alpha, x, 1^{q|x|}, 1^{p(|x|+q|x|)}) \in L^\dagger.$$

■

Lecture 3

This is of course a highly unnatural language, but we will see many extremely natural ones that are also NP-complete, starting with SAT today.

Definition I.17 (Conjunctive normal form)**.** A Boolean formula is in *Conjunctive normal form* (CNF) if it is an $\wedge$ operation on many

Definition I.18 (SAT)**.** SAT is the language consisting of all satisfiable CNF formulae.

Theorem I.19 ([Cook71, Levin73] theorem)**.** SAT *is* NP*-complete.*

Proof. It is clear that SAT is in NP. It remains to show that it is NP-hard. The main idea of the proof is that computation is local.

Let $L \in \mathsf{NP}$ have verifier M with runtime and certificate size $p(|\text{input}|)$. We need a poly-time f such that $x \in L \iff f(x) \in \mathsf{SAT}$.

For any fixed x, we can capture the computation of $M(x, \cdot)$ by a CNF φ_x such that

$$\exists u \in 2^{p|x|} \text{ with } M(x,u) = 1 \iff \varphi_x \text{ is satisfiable.}$$

Claim. *Let N be a deterministic Turing machine that runs in time $T(n)$ on every input u of length n, and outputs* 0 *or* 1*. Fix an n and let u denote inputs of length n. Then,*

(1) there exists a CNF $\varphi(u, AUX)$ (where AUX is a set of auxiliary variables) of size polynomial in $T(n)$ such that for every $u \in 2^n$, $\varphi(u, AUX)$ is satisfiable as a function of the auxiliary variables iff $N(u) = 1$.

(2) φ is computable in time $\operatorname{poly}(T(n))$ *from N, T and u.*

Proof. The proof is in two steps.

(Step 1) Let N be a deterministic Turing machine that runs in time $T(n)$ on every input u of length n, and output 0 or 1. Then, for every n there exists a Boolean circuit ψ of size $\operatorname{poly}(T(n))$ such that $\psi(u) = 1$ iff $N(u) = 1$.

(Step 2) ψ is computable in time $\operatorname{poly}(T(n))$ from N, T and n.

Let $b_{s,j}$, $h_{s,j}$, and $q_{s,j}$ be defined as follows.

$$b_{s,j} \coloneqq i\text{-th bit at time } s;$$

$$h_{s,j} \coloneqq \begin{cases} 1 & \text{if the head is at position } j \text{ at time } s, \\ 0 & \text{otherwise; and} \end{cases}$$

$$q_{s,j} \coloneqq \text{state of } N \text{ when the head was last at position } j.$$

$q_{s,j}$ itself consists of constantly many bits.

For each $s \in [1, T(n)]$, $(b, h, q)_{s,j}$ only depend on $(b, h, q)_{s-1,j-1}$, $(b, h, q)_{s-1,j}$ and $(b, h, q)_{s-1,j+1}$. This gives a bunch of conditions for each bit that must be satisfied. Any Boolean equality $x = y$ can also be written as $(x \vee \neg y) \wedge (\neg x \vee y)$. □

■

Theorem I.20. 3SAT*, the language of all satisfiable* 3*-CNFs, is* NP*-complete.*

Proof. We wish to reduce SAT to 3SAT. Given a clause with $2k$ literals, one may introduce a single auxiliary variable and write it as the conjunction of two clauses with $k + 1$ literals. Write the clause as $(x_1 \vee \cdots \vee x_{2k})$. Introducing the auxiliary variable z, we may write this as

$$(x_1 \vee \cdots \vee x_k \vee z) \wedge (x_{k+1} \vee \cdots \vee x_{2k} \vee \neg z).$$

Repeating this process yields an $O(k \log k)$ algorithm to write any $2k$-clause as a conjunction of k separate 3-clauses. Performing this for every clause in the conjunction takes polynomial time and only increases the length of the input polynomially. Polynomially many auxiliary variables are required. ■

Lecture 4. Monday August 18

Examples.

- Square root mod: Given $a, b, c \in \mathbb{Z}_+$, check if there exists a natural number $x \leq c$ such that $x^2 = a \pmod{b}$. This is NP-complete.
- A variant of integer factoring: Given $L, U, N \in \mathbb{Z}_+$, check if there exists a *natural number* $d \in [L, U]$ such that $d \mid N$. This is NP-hard under randomized Karp reductions.
- Minimum circuit size problem (MCSP): Given the truth table of a Boolean function f and an integer s, check if there is a circuit of size at most s that computes f. This is in NP.

  Is it NP-complete? Who knows!? Levin (or was it Cook?) delayed his publication by a year or two trying to prove that it is, but we don't have an answer yet. [ILO20] showed that the multi-output version is NP-hard under polynomial-time randomized reductions. [Hira22] showed the same for the partial function version.

Theorem I.21 (Independent set)**.** *Deciding whether a graph G has an independent set of size k is* NP*-complete.*

We will reduce 3SAT to INDSET.

Proof. Let φ be a 3CNF with m clauses and n variables. Assume that every clause has exactly 3 literals. Associate with each such clause a clique of size $2^3 - 1 = 7$. Each vertex denotes a partial assignment: an assignment of the three literals that satisfies this clause.

This gives m cliques $C_1, \dots, C_m$ consisting of 7 vertices each. Each vertex denotes a partial assignment to the literals involved. Draw an edge between any two vertices which correspond to incompatible Boolean assignments. Call the resultant graph G.

φ is satisfiable iff G has an independent set of size m.

- If G has such an independent set, it must have exactly one vertex from each of $C_1, \dots, C_k$. Since these are all compatible with each other, this yields a global assignment. Since each partial assignment satisfies the corresponding clause, so does the global assignment.

■

Theorem I.22 (Clique)**.** *Deciding whether a graph G has a k-clique is* NP*-complete.*

Proof. G has a k-clique iff $\overline{G}$ has an independent set of size k. Thus INDSET reduces to this problem. ■

Theorem I.23 (Vertex cover)**.** *Deciding whether a graph G has a vertex cover of size k is* NP*-complete.*

Proof. G has a vertex set of size k iff G has an independent set of size $n-k$. Again INDSET reduces to this problem. ■

Theorem I.24 (0/1 integer programming)**.** *A* $0/1$ *integer program is a set of affine inequalities with rational coefficients where the variable lie in* $\{0,1\}$.

The set of satisfiable $0/1$ *integer programs is* NP*-complete.*

Proof. $\mathsf{3SAT}$ reduces naturally to bit programming. A clause $x_1 \vee \neg x_2 \vee x_3$ maps to the inequality $x_1 + (1 - x_2) + x_3 \geq 1$. ■

Theorem I.25 (Max cut)**.** *Given a graph, finding a cut with the maximum size is* NP*-hard.*

We are not stating that it is NP-complete because, the way we have stated it, NP only consists of decision problems.

Proof. We showed that the decision problem VCover is NP-hard. Thus the optimization version $\mathsf{MinVCover}$, finding a minimal vertex cover, is NP-hard. We will now reduce $\mathsf{MinVCover}$ to MaxCut. ■

I left the lecture room at this point.

Lecture 4