COMPUTATION: DAY 8

BURTON ROSENBERG
UNIVERSITY OF MIAMI

1. Complexity

1.1 Associated with each deciding Turing Machine is a "run time" which the number of transitions taken during the computation. Complexity Theory uses an abstracted version of this step count to associate with problems a complexity. Solutions to problems in computation are *algorithms*, but the order of difficulty of the problem captured in the run time of the algorithm appears to measure a difficulty in the problem itself. This insight is the theory of complexity.

1.2 Measure of difficulty is either in the number of steps required to solve an instance of the problem, drawn from the space of instances of the problem, or of the number cells utilized. We will focus on the number of steps, transitions, taken by the Turing machine in it's computation.

1.3 The situation requires that the problem is an infinite family of problems, so creating a list of answers is insufficient as a solution to the problem. There must be an algorithm that answers any instance of the problem drawn from an infinity of instances. We further recognize,

(1) The bound will be a function of the length of the input. Longer inputs are given more steps.

(2) The bound will be the maximum number of steps needed to decide a string, over all strings of a given length.

(3) The bound, a function $t(n)$ where n is the input length, will be categorized by its asymptotic order of growth.

Definition 1.1. The *time complexity* of a language A is the function $t(n)$ of minimum order so that there is a TM with runtime $t(n)$ that decides A.

1.4 While this quantifies over all algorithms it does so in a certain model of computation. With this definition of complexity, the complexity of the problem is not

Date: Tuesday, May 1, 2026.

truly a property of the language only — there is a dependence of the computational model.

1.5 We previously described a k-tape TM, and a simulation of a k-tape machine with a one tape machine. What this simulation result achieved was the show that the definition of computability was not changed by the additional abilities of a k-tape machine. However, the run time was. We showed that a k-tape machine of run time $O(f(n))$, when simulated by a 1-tape machine, that 1-tape machine ran in time $O(f(n)^2)$. So, to say a problem has complexity $O(n^d)$ will depend on whether we mean the complexity on a k-tape machine or on a 1-tape machine.

1.6 Likewise, we previously described the simulation of a non-deterministic TM with run time $O(f(n))$ by a deterministic machine of run time $O(2^{af(n)})$. But there is a difference here. While adding a tape seems intuitively to not be a conceptual leap, one might question whether a non-deterministic algorithm is truly an algorithm. A notion of complexity that blurs the number of tapes but distinguishes determinism from non-determinism has a sense of reasonability to it.

1.7 The exact order of run time is important, once the machine model is fixed. For this section of the course we would rather disregard some details of this order so as to understand the larger outline of complexity theory. We would like, as an example, to not be concerned about the difference between k and one tape machines, but retain the more substantial difference between deterministic and non-deterministic machines.

The Church–Turing hypothesis is that any intuitively computable function is computable on a Turing Machine. We will go on to say something more speculative. That any reasonable model of computation computes functions in a time polynomially related to the time to compute on the standard model one tape Turing Machine. We consider adding tapes to be a reasonable variant, but non-determinism to be unreasonable. This statement has been challenged (but not decimated) by the creation of Quantum Computing.

2. Algorithms

An algorithm is the idea of a program on a Turing machine or otherwise reasonably related model of computation. The notion of reasonably related is the strong Turing-Church hypothesis concerning the number of steps taken by the algorithm compared between the two computational models. While what is being counted is the number of steps, this is referred to as "time" as it is considered each step, if the model were physically run, takes a certain unit of time.

```
def gcd_exp(x,y):
    for d in range(min(x,y),1,-1):
        if x%d==0 and y%d==0: return d
    return 1

def gcd_euclid(x,y):
    while y!=0: x,y = y,x%y
    return x
```

FIGURE 1. Greatest Common Divisor

For a problem to fit into our schema, a problem (such as sorting) as instances (such as sorting a particular array of numbers). The problem is the membership problem for the set $A \subseteq \Sigma^*$ which encodes all instances, and identifies which $a \in A$ are accepted and which are not. The number of steps depends on the problem size, $|a|$, and solving a problem in time t is to solve it in time $t(|a|)$,

$$a \in A \iff M_A(a; t(|a|)) = T$$

Example:

Two algorithms to calculate the greatest common divisor of two positive integers are given in Figure 1. They are both algorithms solving the problem, but `gcd_exp` runs in time exponential in the size of the inputs, the length in digits of values stored in variables `a` and `b`; where as `gcd_euclid`[1] has time no more than cubic in this size.

2.1. **Asymptotic Order of Growth.** You should be familiar with Big-Oh notation. It puts functions into a linear order according to how fast they grow, disregarding constant factors and behavior on small inputs.

Definition 2.1. Let $f(n)$ be a function on the naturals, $f : \mathbb{N} \to \mathbb{N}$. The order of $f(n)$ is the set,

$$O(f(n)) = \{\, g : \mathbb{N} \to \mathbb{N} \mid \exists\, c, n_o > 0, \forall n \geq n_o, c\, f(n) \geq g(n) \,\}$$

Customarily we write $g = O(f(n))$ when more formally we would write $g \in O(f(n))$.

Lemma 2.1. For functions $f, g, h : \mathbb{N} \to \mathbb{N}$,

- $f = O(f)$,
- If $g = O(f)$ and $h = O(g)$ then $h = O(f)$.

[1]This algorithm is found in *Euclid's Elements*, (300 B.C.) and is called the Euclidean Algorithm.

- If $g, h = O(f)$ then $g + h = O(f)$ and $g\,h = O(f^2)$.
- $0 = O(f)$, for any f.

Proof: Left as an exercise for the reader. □

3. The class P

Definition 3.1. An algorithm is called polynomial if its run time is of the form $O(n^d)$ for some non-negative integer d. The class P is the class of all problems that can be solved with a polynomial time algorithm on any reasonable model of computation.

The following properties of languages have polynomial-time algorithms that decide the property.

- The acceptance problem for regular languages.
- The emptiness problem for regular languages,
- The equality problem for regular languages.
- The acceptance problem for context free languages.

Theorem 3.1. Let $A \subseteq \Sigma^*$ be a regular language. The problem of determining if $a \in A$ is in the class P.

Proof: Since the language is regular, there is a FA which accepts the language. Write down a description of this language and simulate it on a TM on the input a. The TM will halt with a decision.

Since the FA is a fixed machine, the length of its description is fixed, and we consider inputs of length longer than the length of the description. Each character of the input causes a bounded number of machine steps in the simulator. Hence the run time is $O(n)$. □

Theorem 3.2. Let $\langle A \rangle$ be the description of a FA A. The set,

$$E_{FA} = \{ \langle A \rangle \mid \mathcal{L}(A) = \emptyset, \}$$

is in the class P.

3.1 [Proof] We give a polynomial time algorithm to decide if machine A accepts any string at all. The algorithm searches backwards from all accept states to states that can lead to an accept state. Repeatedly iterate over all states. In the first iteration, mark the accept states; in the second and subsequent iterations mark any state with

a transition to a marked state. When an iteration does not mark any states halt and return that the language is not empty if and only if the start state is marked.

The description will be of size proportional to the number of states. Each iteration will take states proportional to the number of states; and after as many iterations as states the algorithm must have marked all the states it ever can mark. So the algorithm runs in time $O(n^2)$, where n can be taken to be the number of states in A. □

Theorem 3.3. Let $\langle A \rangle$ and $\langle B \rangle$ be the descriptions of two FA 's. The set,

$$EQ_{FA} = \{ \langle A \rangle, \langle B \rangle \mid \mathcal{L}(A) = \mathcal{L}(B), \}$$

is in the class P.

3.2 [**Proof**] We observe that,

$$\mathcal{L}(A_1) = \mathcal{L}(A_2) \Leftrightarrow \mathcal{L}(A_1 \oplus A_2) = \emptyset$$

The machine for $\mathcal{L}(A_1 \oplus A_2)$ can be constructed from $\langle A \rangle$ and $\langle B \rangle$, by the product construction which has as accepting states when only one of the machines would accept. Then we use the algorithm above to see if this machine accepts nothing. □

Theorem 3.4. Let $A \subseteq \Sigma^*$ be a context free language. The problem of determining if $a \in A$ is in the class P.

3.3 [**Proof**] Since the language is context free, there is a Chomsky Normal Form which accepts the language. Since this is a fixed grammar, consider inputs of length longer than the length of the grammar. The CYK algorithm decides in time $O(n^3)$ in the length of the string. □

4. Properties of Languages: some are Decidable

4.1 If we know that a language is Regular or Context Free, some properties of these languages are decidable. We can know that the language is these classes because they are presented in particular ways. For instance, if we are given a finite automata, we know the language is regular. If we are given a Context Free Grammar, we know the language is a Context Free Language.

4.2 Another way to describe the situation is that the index is of a Turing machine, and we are promised that the Turing machine accepts are Regular (or Context Free) language. It's a bit non-constructive, it could happen that the Turing machine is a Finite Automata simulator with a finite automata program baked right in. However I mention it here so that these results have a wider context.

5. Regular Languages

5.1 While the set REG is undecidable, many properties of regular languages are decidable.

Theorem 5.1. It is decidable if a regular language is empty, or contains all strings; and if two regular languages are equal.

5.2 [**Proof**] Given a regular language there is a finite automata accepting that language. Working backwards from accept states will provide a path, if any, for some accepted string. This decides emptiness.

The language accepts all strings if the complement is empty. Given the finite automata for the language, exchange the accept and non-accept states and determine if the resulting language is empty.

Given two regular languages there are two finite automata F_A and F_B accepting the respective languages. A finite automata can be constructed that accepts

$$F_A = F_B \iff (F_A \wedge \neg F_B) \vee (\neg F_A \wedge F_B) = \emptyset$$

and then decide the emptiness of this regular language. The first term is the empty set when $F_A \subseteq F_B$, and the second term is the empty set when $F_B \subseteq F_A$. For the sum to be empty, both these conditions need to be true. □

6. Context Free Languages

6.1 The same set of questions for Context Free Languages is more difficult.

Theorem 6.1. Non-equality for Context Free Languages is recursively enumerable.

6.2 [**Proof**] A non-deterministic Turing machine can guess the string on one language and not the other. Determining membership is recursive. If we enumerate CFG's as G_i, then the language is,

$$\neg EQ_{CFL} = \{\, i, j \mid \exists s, G_i(s) \neq G_j(s) \,\}$$

and the complement is co-recursively enumerable,

$$EQ_{CFL} = \{\, i, j \mid \forall s, G_i(s) = G_j(s) \,\}$$

Since the G_i is recursive, respectively these languages are class Σ_1 and Π_1. □

Theorem 6.2. Non-equality for Context Free Languages is undecidable.

Theorem 6.3. Emptiness for Context Free Languages is decidable.

6.3 [**Proof**] The the grammar in Chomsky Normal Form, apply a marking algorithm that determines for each variable whether that variable is *productive*, that is can become by the grammar a string of terminals. If the rule $S \to \varepsilon$ is present the language is non-empty. Mark any variable V if it appears in the rule $V \to v$, with v a terminal. Mark any variable V if it appears in the rule $V \to A\,B$, where A and B are both marked. Continue until no more markings can be made. The language is non-empty if and only if S is marked. □

Theorem 6.4. Whether a Context Free Languages is all strings is undecidable, and Π_1, as it is the predicate for the i-th language G_i, $\forall s, G_i(s)$ with $G_i(s)$ recursive.

Theorem 6.5. Acceptance for a CFL is decidable.

6.4 [**Proof**] Convert to CNF. Talk about the blow up in the conversion. Now use CYK , an $O(n^3)$ dynamic programming algorithm to decide acceptance. □

7. The class NP

7.1 The extended Church–Turing hypothesis kept as distinct those languages decided by a nondeterministic TM. The search for advice required exponential time in the length of the calculation, as all computation paths might be explored. This distinction continues in the case of polynomial algorithms, but considering problems that might not be solved in polynomial time, but give a solution, that solution can be verified correct in polynomial time. The connection with nondeterministic machines is that can consider this solution by guess. But we must be able to prove the guess correct.

7.2 Any P time algorithm is NP. Rather than guess we construct the answer, then verify our own answer. If the algorithm to construct the answer is correct there is no need to verify, but I present it this way just to continue in the format of an NP algorithm.

Definition 7.1. A language A is in NP if there is a polynomial time machine V taking two inputs and,

$$a \in A \iff \exists\, w \text{ s.t. } V(a, w) = T.$$

and w is polynomial length in the length of a.

7.3 An example of an NP problem is the problem if a Hamiltonian Circuit. Given a graph $G = \langle V, E \rangle$, a path between s and t, $s, t \in V$ is a sequence of edges in E that begins a s and ends at t, and subsequent edges share exactly one vertex. The

problem of a Hamiltonian Circuit is given a presentation of G, s and t, is there are path from s to t that visits every vertex V in G exactly once.

7.4 Given a path, it is very quick, a polynomial of low degree in the description of the graph G to determine if the path goes from s to t, following edges in appropriately, and visiting each vertex exactly once. However, at this moment, no P time algorithm is know to in general find such a path.

7.5 Because one has a polynomial time verifier, disregarding time, there is a solution to Hamiltonian Circuit which follows a pattern common to all NP class problems. A Hamiltonian Circuit for a graph of n vertices will have $n-1$ edges. Write a program to enumerate all $n-1$ sized subsets from graph's set of edges. As they are being enumerated, test them one by one with the verifier. If and when the verifier accepts, the solution has been found. If the enumeration completes with no proposed set of $n-1$ edges being a Hamiltonian Circuit, then reject the graph, as it has no Hamiltonian Circuit.

7.6 The string w can be called the witness. The class NP is the class of problems for which a solution has a polynomial sized witness. The witness is a membership proof.

8. Polynomial Reduction

Definition 8.1. A *polynomial time function* $f : A \to B$ is a Turing machine (in any reasonable mode of computation) that when started with an $a \in A$ on the time, halts in time $t(|a|)$ with only a $b \in B$ written on the tape, and t is of order $O(n^d)$ for some $d \in \mathcal{N}$.

Definition 8.2. A *polynomial time reduction* between two languages $A \subseteq \Sigma^*$ and $B \subset \Sigma^*$ is a polynomial time function $f : \Sigma^* \to \Sigma^*$ such that $f(A) \subseteq B$ and $f(\overline{A}) \subseteq \overline{B}$. It is denoted $A \leq_P B$.

Lemma 8.1. If $A \leq_P B$ and B can be solved in polynomial time, than A can be solved in polynomial time.

8.1 [**Proof**] Because B is solvable in polynomial time, there is a polynomial time function $g : \Sigma^* \to \{ T, F \}$, that decides the set $B \subseteq \Sigma^*$. Let f the reduction. Then the composition function $g \circ f$ is a polynomial time function that decides A. $\square$.

Definition 8.3. A language $A \subseteq \Sigma^*$ is *NP complete* if,

(1) The language A is in NP, and

(2) for every language B in NP, there is a reduction $B \leq_P A$.

9. NP Completeness

9.1 All P languages are in NP. Whether there are NP languages not in P is an open problem. Most people think that there are. That these language classes are distinct. The progress to solving the P versus NP problem involves the theory of *NP completeness*. In this theory a reduction is defined that organizes NP problems into those which are at least as hard, and then shows that this ordering has "hardest problems" in NP. Those hardest problems are called NP complete, and under this ordering they are known to be all as hard as each other.

10. Cook–Levin Theorem

Theorem 10.1 (Cook–Levin)**.** Given A, a language in NP, then $A \leq_P SAT$, where SAT is the general problem of finding a satisfying assignment to a boolean formula.

10.1 [**Proof**] Given a problem in NP, there is a polynomial time Turing machine that decides it. From this machine and a given string s, in polynomial time a set of boolean variables can be defined that represent all stages of the computation of the machine, and equations written down that verify that the setting of the variables corresponds to a correct computation of the the machine which accepts or rejects s.

Therefore this problem of finding an assignment to boolean variables to satisfy boolean equations is universal for the class NP. It is also in NP, since a satisfying assignment, if it exists, can be verified in polynomial time. □

10.2 It was more specifically shown that the collection of boolean equations can be collected into a single equation in what is known as 3 Conjunctive Normal Form. In this form the problem is called *3-SAT*, so the statement is: 3-SAT in NP Complete.

10.3 We then showed that 3-SAT can be reduced to k-CLIQUE and the HAMILTONIAN-PATH. So both these are also NP Complete problems. The reductions are clever and in a certain sense very direct.

Theorem 10.2. SAT, k-CLIQUE and k-COVER are NP-complete.

10.4 [**Proof**] A language is NP complete is it is an NP language, and any NP language can be reduced the that problem. The Cook-Level theorem shows SAT is NP complete. It can be shown (not done here) that any SAT can be written as a 3-SAT in polynomial time and (consequently) without more than a polynomial increase in size. . Hence,

$$\text{SAT} \leq_P \text{3-SAT}$$

By the properties of polynomial reductions, 3-SAT is NP complete. 3-SAT is reducible to the other languages in the theorem statement, and hence as also NP complete. □

10.5 A problem such that an NP language can reduce to it, but the problem is not necessarily in NP is called *NP hard*.

11. Randomized algorithms and BPP

11.1 The issue of the complements of NP languages is next considered.

Definition 11.1 (co-NP)**.** A *co-NP* language is the complement of an NP language. It is defined as, a language A such that there exists a PPT $\overline{V}$,

$$A = \{\, x \mid \forall y, \overline{V}(y, x) = T \,\}$$

11.2 There might not be (depending if P equals NP) a P-time verifier deciding membership in a co-NP language. To address this we need to go further in our machine models. The step we take now is to introduce a probabilistic Turing machine. The model we adopt is a Turing machine with a read-only tape with contents chosen randomly and uniformly over the space of possible contents. Since the machine will be P-time bound, for a suitable constants d and c it is sufficient to fill cn^d cells with uniformly and independently chosen coin flips.

11.3 For decision problems, the output of a PPT is accept or not accept. A PPT machine *computes* when the output is the contents of the output tape. In this case, a PPT is a string-valued random variable, where the event space is the choice of contents on the randomness tape.

11.4 The class of problems accepted by a probabilistic machine will depend on what criteria on the probability is used to determine acceptance. A *bounded probabilistic polynomial time* class makes this definition.

Definition 11.2 (BPP)**.** A language A is in *bounded probabilistic polynomial time* (BPP) if there exists a probabilistic polynomial-time Turing machine V_Ω such that for all inputs x:

$$x \in A \Rightarrow \Pr{}_\Omega[V_\Omega(x) = 1] \geq \frac{2}{3},$$
$$x \notin L \Rightarrow \Pr{}_\Omega[V_\Omega(x) = 1] \leq \frac{1}{3}.$$

11.5 *Nota bene:* The constants 2/3 and 1/3 are arbitrary: by independent repetition of the algorithm and taking a majority vote, the error probability can be reduced exponentially.

11.6 The name BPP includes the three elements that define the class,

(1) B as in bounded. The acceptance criteria.

(2) P as in probabilistic. The computational model.

(3) P as in polynomial time. The resource bound.

11.7 The class BPP is a very interesting class, and it has come to be seen as the most natural class in which to define effectively computable functions. This is because some things we really wish to compute are computable in BPP, and the errors can be made negligible. It also has a relationship with the quantum computing class BQP, where we replace probabilistic computing with quantum computing. The class NP cannot be related to a quantum class, because a quantum computation is in essence probabilistic.

12. Co-NP and Interactive Proof Systems

12.1 The problem of graph isomorphism is an NP problem. Our attention is to its complement, the problem of non-isomorphic graphs.

Definition 12.1. For each n, let V_n be a set of n elements called *vertices*. A *labeled graph* on V_n is given by an edge set

$$E \subseteq V_n^{(2)},$$

where $V_n^{(2)}$ denotes the set of all two-element subsets of V_n.

The symmetric group $\mathcal{S}_n$ acts on labeled graphs by permuting vertices: for $\pi \in \mathcal{S}_n$ and $G = (V_n, E)$,

$$\pi(G) = (V_n, \{\{\pi(u), \pi(v)\} : \{u, v\} \in E\}).$$

The *orbit* of G under this action is

$$[G] = \{\pi(G) : \pi \in \mathcal{S}_n\}.$$

An *abstract graph* is an orbit under this action.

12.2 Two labeled graphs G and H are *isomorphic* if they lie in the same orbit, i.e., if there exists $\pi \in \mathcal{S}_n$ such that $H = \pi(G)$.

A permutation $\pi \in \mathcal{S}_n$ is an *automorphism* of G if $\pi(G) = G$.

12.3 [**Example**] For vertex set $V = \{1, 2, 3\}$ and the edge set $E = \{\{1, 2\}, \{2, 3\}\}$ is a member of the graph L_3. The orbit contains three elements, determined by which of the three vertices is in the middle. For each of the elements, there is an automorphism which fixes the middle vertex and swaps the other two.

12.4 The set

$$ISO = \{\, \langle G_1, G_2 \rangle \mid \text{ the two graphs } G_1 \text{ and } G_2 \text{ are isomorphic} \,\}$$

is NP. Non-isomorphism of graphs is co-NP.

12.5 Two graphs can be shown non-isomorphic in polynomial time if the model of computation is extended to a pair of Turing machines, one a PPT, the *verifier* and the other a Turing machine with no resource bound, the *prover*, the converse through a shared tape. The notation for this is,

$$\langle P, V_\Omega \rangle(x)$$

Definition 12.2 (Interactive Proof System)**.** A language L has an interactive proof system if there exists a interactive machines $\langle P, V_\Omega \rangle$ where V_Ω is a a PPT and the following completeness and soundness criteria hold.

- **Completeness:**

$$\forall x \in L, \;\; \Pr{}_\Omega[\langle P, V_\Omega \rangle(x) = 1] \geq \frac{2}{3}.$$

- **Soundness:**

$$\forall x \notin L \;\; \forall P^*, \;\; \Pr{}_\Omega[\langle P^*, V_\Omega \rangle(x) = 1] \leq \frac{1}{3}.$$

12.6 While the prover is computationally unbounded, the verifier is highly constrained: it runs in probabilistic polynomial time and must base its decision only on the interaction. The guarantees of the system are captured by completeness and soundness. Completeness requires that for every true statement, there exists a prover strategy that convinces the verifier with high probability. Soundness requires that for every false statement, no prover (even a malicious one) can convince the verifier except with small probability.

12.7 We illustrate this with graph non-isomorphism. Given two graphs G_A and G_B on a vertex set V_n, consider their orbits under relabeling. These orbits are either identical (if the graphs are isomorphic) or disjoint (if they are not). The problem is to distinguish between these two cases.

12.8 [**Algorithm**] The verifier proceeds by selecting a random bit $b \in A, B$, choosing a uniformly random permutation $\pi \in \mathcal{S}_n$, and forming the graph $H = \pi(G_b)$. The

graph H is sent to the prover, who must guess whether H originated from G_A or G_B.

12.9 If the orbits are identical, then H is distributed identically regardless of whether it was generated from G_A or G_B, and even an all-powerful prover can do no better than guessing, succeeding with probability at most $1/2$. If the orbits are disjoint, then H uniquely determines which orbit it came from; an all-powerful prover can identify the correct source with probability 1. Repeating this experiment reduces the soundness error exponentially.

13. Zero-knowledge proof systems

13.1 In some interactive proof systems, the prover may reveal more information than is strictly necessary to convince the verifier of the validity of a statement. This raises the question of whether it is possible to design protocols in which the verifier learns nothing beyond the truth of the statement being proven. Such protocols are called *zero-knowledge proofs*. Informally, a proof is zero-knowledge if whatever the verifier sees during the interaction could have been generated independently, without access to the prover.

Definition 13.1 (Perfect Zero-Knowledge)**.** An interactive proof system $\langle P, V_\Omega \rangle$ for a language L is said to be *zero-knowledge* if for every probabilistic polynomial-time verifier V_Ω^*, there exists a probabilistic polynomial-time simulator M_Ω^* such that for all $x \in L$ such that

(1) With probability at most $1/2$, $M_\Omega^*(x) = \perp$, where $\perp$ is a special "undecided" symbol.

(2) Let $m^*(x)$ me the distribution of $M_\Omega^*(x)$ conditioned on that it does not output $\perp$,

$$\Pr{}_\Omega[m^*(x) = \alpha] = \Pr{}_\Omega[M_\Omega^*(x) = \alpha \mid M_\Omega^*(x) \neq \perp]$$

then $\langle P, V_\Omega^* \rangle(x)$ and $m^*(x)$ are identically distributed random variables.

13.2 Note that this is over all V_Ω^*, which you can think of as an adversarial machine attempting to extract knowledge from P. Not only is it adversarial, it can output whatever it wants, including the prover–verifier transcript and any other derived strings. Also note that the zero-knowledge property is located entirely in the case that $x \in L$.

13.3 [Zero-Knowledge Protocol for Graph Isomorphism]

Let G_0 and G_1 be two graphs on the same vertex set V_n. The prover P knows an isomorphism ϕ such that

$$\phi(G_0) = G_1.$$

The goal is for P to convince the verifier V that $G_0 \cong G_1$ without revealing ϕ.

13.4 The protocol proceeds as follows:

(1) The prover selects a uniformly random permutation $\pi \in \mathcal{S}_n$ and computes

$$H = \pi(G_0).$$

The prover sends H to the verifier.

(2) The verifier selects a random bit $b \in 0, 1$ and sends b to the prover.

(3) The prover responds with a permutation ψ defined by

$$\psi = \begin{cases} \pi & \text{if } b = 0, \\ \pi \circ \phi^{-1} & \text{if } b = 1. \end{cases}$$

(4) The verifier checks that

$$\psi(G_b) = H.$$

If so, the verifier accepts; otherwise, it rejects.

13.5 If $G_0 \cong G_1$, the prover can always respond correctly, so completeness holds. If the graphs are not isomorphic, no prover can correctly answer both challenges for the same H, so the verifier is convinced with probability at most $1/2$. Repeating the protocol reduces the error probability exponentially.

13.6 We only show a version of zero-knowledge called *honest verifier zero-knowledge*, in which a verifier that follows the protocol can extract no knowledge. The real task is any verifier can extract any information, but it is harder to prove.

13.7 This protocol is honest verifier zero-knowledge because the verifier's view can be simulated by choosing a random b, a random permutation ψ, and setting $H = \psi(G_b)$, producing a distribution identical to that of a real interaction.