COMPLEXITY THEORY

SIDDHANT CHAUDHARY

These are my course notes for the course COMPLEXITY THEORY that I undertook in my fourth semester. Check References for the reference books that I used in this course.

Contents

Date: January 2021.

1. An Informal Introduction

This will be a very informal introduction to some ideas in Complexity Theory. We will assume some basic knowledge of algorithms and asymptotic notations. Informally, we define the class **P** to be the class of all problems that can be solved in polynomial time. Also, we define **NP** to be the class of problems which have a *simple and efficient* certificate for a *yes* instance (we will define these more precisely). As it will turn out (and as it is intuitively clear), we have that $\mathbf{P} \subseteq \mathbf{NP}$. One of the current millenium problems is the question whether $\mathbf{P} = \mathbf{NP}$. Let's look at some examples.

Example 1.1 (**Graph Coloring**)**.** Let G be any undirected graph, and consider the k-*coloring problem* of determining whether G is k-colorable or not. When $k = 2$, i.e the problem of 2-coloring is clearly in **P**, because this is equivalent to checking whether the graph is bipartite. However, it is known that the problem of 3-coloring is **NP**-complete (whatever this means). Using this fact, we can show that k-coloring is **NP**-complete for $k \geq 3$, and this can be shown using a *reduction*. Suppose we are given a graph G, and we want to see whether this graph is k-colorable. Then, add a new vertex to G, and connect it to all other vertices of G. If the new graph is called G', then the problem is equivalent to asking whether G' is $k+1$-colorable. So, if we assume that k-colorability is **NP**-complete, then $k+1$-colorability will also turn out to be **NP**-complete.

Example 1.2 (**Boolean Satisfiability**)**.** In this problem, we ask whether a boolean formula ϕ in k-CNF is *satisfiable* (if you don't know what a boolean formula is or what CNF is, look it up), i.e whether there is an assignment to the variables in the formula which makes the formula TRUE. We first claim that 2-SAT is in **P**. To prove this, suppose we have some boolean ϕ formula in 2-CNF. An example of such a formula is

$$(x_1 \vee \neg x_2) \wedge (x_3 \vee x_1) \wedge (x_2 \vee \neg x_4)$$

where the total number of variables is 4. So, suppose there are n variables in the formula ϕ, and let the variables be $x_1, ..., x_n$. Let $(l_1 \vee l_2)$ be any clause in the formula ϕ, where l_1, l_2 are literals (i.e l_1, l_2 are either x_i or $\neg x_i$). Then, we have the following two sided implication.

$$(l_1 \vee l_2) \iff (\neg l_1 \implies l_2) \wedge (\neg l_2 \implies l_1)$$

So, we make a graph G as follows: the nodes of G are all possible literals, i.e the nodes of G are $\{x_1, x_2, ..., x_n, \neg x_1, \neg x_2, ..., \neg x_n\}$. Further, if $(l_1 \vee l_2)$ is a clause in ϕ, then there are *directed* edges $(\neg l_1, l_2)$ and $(\neg l_2, l_1)$ in G. This graph is called the *implication graph* of ϕ. We claim that

$$\phi \textit{ is satisfiable} \iff \textit{there is no pair of paths } x_i \to \neg x_i \textit{ and } \neg x_i \to x_i \textit{ in } G$$

If we are able to prove the above claim, then it is clear that 2-SAT is a problem in **P**. So, let us now prove the claim. First, suppose there is a pair of paths $x_i \to \neg x_i$ and $\neg x_i \to x_i$ in G. Then, we see that ϕ *cannot* be satisfied; if, for the sake of contradiction, ϕ was satisfiable, then it would give us a pair of implications $x_i \implies \neg x_i$ and $\neg x_i \implies x_i$, and hence any value given to x_i will lead to a contradiction. This shows the forward direction. Conversely, suppose there is no pair of such paths, and we will show that ϕ is satisfiable.

It turns out that 3-SAT is **NP**-complete.

Example 1.3. (**Subset Sum**) In this problem, we are given a set $S = \{x_1, ..., x_n\}$ of positive integers, and we are given another positive integer t. The question is whether there is a subset of $A \subseteq S$ such that the sum of elements in A is equal to t. This problem clearly is in **NP**, because for every *yes* instance, the certificate is simply the subset whose sum is t. It turns out that this problem is **NP**-complete.

2. Basic Complexity Classes

2.1. **An Important Note.** In this section and further sections, we will be assuming the knowledge of some concepts in TOC, like Turing Machines. I have made separate notes for that material.

2.2. **Running Times.** We already know what Turing Machines are, and what their computational power is. Roughly a Turing Machine is equivalent to a modern day computer, in the sense that it has memory and it has the ability to carry out algorithms. We also know that there are two kinds of TMs: deterministic and non-deterministic, and the difference between the two lies in the ability of making *choices*. However, as far as computational power is concerned, the two of them are equivalent, i.e every non-deterministic TM is equivalent to a deterministic one. We will now see how to define the notion of *running time* using this model. This will be what most of this section will focus on.

Definition 2.1. Let M be any deterministic TM, and let $f : \mathbb{N} \to \mathbb{N}$ be a function. We say that M is an $f(n)$-*time* machine if M halts on all its inputs and $f(n)$ is the maximum number of steps that M uses on any input of size n. The class of all languages which are decidable by an $O(f(n))$-time deterministic TM is denoted by **DTIME**$(f(n))$.

We can also define a notion of running time on a non-deterministic TM, and we now do this.

Definition 2.2. Let N be any non-deterministic TM, and let $f : \mathbb{N} \to \mathbb{N}$ be any function. N is said to be a *decider* if it halts on all its computation branches. If N is a *decider*, then N is said to be an $f(n)$-*time* machine if $f(n)$ is the maximum number of steps that N takes on any computation branch of any input of size n. The class of all languages that can be decided by an $O(f(n))$-time non-deterministic TM is denoted by **NTIME**$(f(n))$.

Theorem 2.1. *Let $t : \mathbb{N} \to \mathbb{N}$ be any function, where $t(n) \geq n$. Then every $t(n)$ time non-deterministic single-tape TM is equivalent to a $2^{O(t(n))}$ time deterministic single-tape TM.*

Proof. Suppose we are given a $t(n)$-time non-deterministic single tape TM. Then, we know how to convert this to a deterministic TM: we just traverse the branches of the computation tree of N in BFS order. Now, any branch of the computation tree of an input of size n has height atmost $t(n)$. If b is the maximum number of choices that the transition function of N provides, then there are atmost $b^{t(n)}$ leaves in the computation tree. Now, it takes time $O(t(n))$ to travel from the root to a leaf. So, traveling all the leaves takes time $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. This completes the proof. ■

Definition 2.3. Define the class **P** to be the set of all languages that are decidable in polynomial time by a deterministic single-tape TM. In other words,

$$P = \bigcup_{k \in \mathbb{N}} \textbf{DTIME}(n^k)$$

Definition 2.4. A *verifier* for a language A is a deterministic TM V such that

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

A verifier is said to be of *polynomial time* if it runs in polynomial time in the length of w. A language A is said to be *polynomially verifiable* if it has a polynomial time verifier. The class of polynomially verifiable languages is denoted by **NP**.

Remark 2.1.1. The string c above is also known as a *certificate*. Note that if A is in **NP**, then every certificate c for some $w \in A$ must have polynomial length (polynomial in the length of w) because a polynomial time verifier cannot even scan an input of length that is not polynomial.

Theorem 2.2. *A language is in* **NP** *if and only if it is decided by some non-deterministic polynomial time TM.*

Proof. First, suppose a language L is in **NP**. We give a non-deterministic polynomial time TM that decides the language. Suppose V is a polynomial time verifier for L that runs in time $O(n^k)$. Make a new non-determinstic machine M as follows.

(1) On input w, non-deterministically create a string c of size atmost n^k.
(2) Run the verifier V on $\langle w, c \rangle$. Accept if V accepts, and otherwise reject.

Conversely, suppose a language L is decided by a non-deterministic TM N. We can construct a polynomial time verifier V as follows: on input $\langle w, c \rangle$, simulate the machine V on w by taking c as a reference for which non-deterministic choice to make. If V accepts on this branch of the computation tree, then accept; otherwise reject. ■

Corollary 2.2.1. $\textbf{NP} = \bigcup_{k \in \mathbb{N}} \textbf{NTIME}(n^k)$.

Proof. This easily follows from the definition and the above theorem. ■

2.3. **Poly-time Reducibility and NP-Completeness.** Let us begin with a quick definition.

Definition 2.5. A function $\Sigma^* \to \Sigma^*$ is said to be *polynomial time computable* if there is some deterministic polynomial time TM M that halts with just $f(w)$ on its tape on any input w.

Definition 2.6. Let A, B be any two languages. We say that A is *polynomial time reducible* or *Karp-reducible* to B, written $A \leq_P B$, if there is some polynomially computable function $f : \Sigma^* \to \Sigma^*$ such that

$$x \in A \iff f(x) \in B$$

Definition 2.7. A language B is said to be **NP**-*complete* if B is in **NP** and every language A in **NP** is polynomially reducible to B.

2.4. **The Cook-Levin Theorem.**

Theorem 2.3 (**Cook-Levin Theorem**). *SAT is NP-Complete.*

Proof. My favorite proof of this is in Theorem 7.37 in [2]. ■

Exercise 2.1. Show that SAT $\leq_P$ 3-SAT. From this, conclude that 3-SAT is NP-complete.

Solution. First, we will show that every boolean expression can be written in conjunctive normal form. Then, we show that every formula in CNF is equivalent to a formula in 3-CNF. This will be our strategy. Complete this proof.

Theorem 2.4. *Consider the problem* CLIQUE *given by*

$$\text{CLIQUE} = \{\langle G, k\rangle \mid G \text{ has a clique of size atleast } k\}$$

Then 3SAT $\leq_P$ CLIQUE. *So, it follows that* CLIQUE *is NP-complete.*

Proof. Let ϕ be a boolean formula in 3-CNF, and suppose it has k-clauses, i.e

$$\phi = C_1 \wedge C_2 \wedge ... \wedge C_k$$

where each clause C_i contains three literals. Now, we make a graph G as follows: for each literal inside each clause, make a vertex in the graph. So, there are $3k$ vertices in G. Let x, y be any two literals in *distinct* clauses C_x and C_y. We connect x and y with an edge if x and y are *not conflicting*, i.e if $x \neq \neg y$. So, we claim that ϕ is satisfiable if and only if G has a clique of size atleast k. To show this, first suppose that ϕ is satisfiable, and let $l_1, l_2, ..., l_k$ be the literals which have an assigned value of 1 in the clauses $C_1, C_2..., C_k$. Then, the set $\{l_1, l_2, ..., l_k\}$ forms a clique of size k in G. Conversely, suppose G has a clique of size atleast k. So, by assigning a value of 1 to each of the literals belonging to the clique, we see that ϕ is satisfied, because every clause will have alteast one literal which is assigned 1, and there are no conflicts. This completes the proof. ■

2.5. **Space Complexity.** In this section, we will look at another important measure of complexity of algorithms.

Definition 2.8. Let $f : \mathbb{N} \to \mathbb{N}$ be a function, and let M be a deterministic TM that halts on all inputs. We say that M has *space complexity* $f(n)$ if $f(n)$ is the maximum number of tape cells that M scans on an input of length n. If M is a non-deterministic TM, we say that M has *space complexity* $f(n)$ if $f(n)$ is the maximum number of tape cells that M scans on any branch of its computation on any input of size n.

Definition 2.9. Let $f : \mathbb{N} \to \mathbb{N}$ be any function. The class of languages decidable by a deterministic TM of space complexity $O(f(n))$ is denoted by **SPACE**$(f(n))$, and the class of languages decidable by a non-deterministic TM of space complexity $O(f(n))$ is denoted by **NSPACE**$(f(n))$. The class **PSPACE** is defined as

$$\textbf{PSPACE} = \bigcup_{k \in \mathbb{N}} \textbf{SPACE}(n^k)$$

Theorem 2.5 (**Savitch's Theorem**). *Let $f : \mathbb{N} \to \mathbb{N}$ be a function with $f(n) \geq \log n$. Then*

$$\textbf{NSPACE}(f(n)) \subseteq \textbf{SPACE}(f^2(n))$$

or in simple words, every non-deterministic TM with space complexity $O(f(n))$ is equivalent to a deterministic TM with space complexity $O(f^2(n))$.

Proof. Let N be a non-deterministic TM that uses $O(f(n))$ space and decides a language L. We will construct a deterministic TM M that decides L and uses $O(f^2(n))$ space.

The idea is to explore the computation tree of N in a clever way. If we sequentially traverse all the branches in the computation tree of an input of size n, then we need to keep track of all of the choices we made to move on to the next branch, and each choice requires $O(f(n))$ space to store. Storing all the choices will then require $2^{O(f(n))}$ space, which we don't want.

Instead, we do the following. First, we modify the TM N so that whenever N accepts, it clears its tape contents and moves its head to the extreme left and enters a new accept state q_{accept}, and let us call this accepting configuration c_{accept}. Let c_{start} be the starting configuration of N on an input word w. We devise an algorithm REACH that takes as input three parameters: c_1, c_2 and t, where c_1,c_2 are any two configurations of N of size $O(f(n))$, and t is an integer. Moreover, REACH(c_1, c_2, t) will return 1 if the configuration c_2 is reachable from c_1 within t steps, and otherwise it returns 0. The algorithm for REACH is simple.

```
REACH (c1 , c2 , t):
    if t = 1, test directly whether c1 = c2 or whether c1 can reach
        c2 in one step using the transition function of N.
    if either is true
        return 1
    else
        return 0
    else
        for each configuration c of N using space f(n)
            k1 = REACH(c1 , c , t/2)
            k2 = REACH(c , c2 , t/2)
            return k1 and k2 //logical and
```

Now, we can define our machine M as follows. Let $\mathcal{N}$ be the maximum number of configurations of N of length $f(n)$, so we clearly see that $\mathcal{N} = 2^{O(f(n))}$ (this is where we are using the fact that $f(n) \geq \log n$), and $\mathcal{N}$ clearly depends upon N. So, our machine M operates as follows: on input w, output the result of REACH$(c_{\text{start}}, c_{\text{accept}}, \mathcal{N})$. Now, let us verify that M really runs in $O(f^2(n))$ time.

To compute REACH(c_1, c_2, t), M needs to store the configurations c_1, c_2 and the number t on a function stack (note that the function is recursive). The number t is atmost $\mathcal{N}$, and so t can be stored in $\log \mathcal{N} = O(f(n))$ bits. Since the configurations are of size atmost $f(n)$, they can be stored in $O(f(n))$ space as well. So, all these three parameters can be stored in $O(f(n))$ space. Now, here is an important point: a call to REACH uses to further calls to REACH. So even though REACH is of exponential time, the *same* space can be used to solve the two subcalls (and this is the important difference between space and time; space can be reused). Each call to REACH halves t, and hence there are $\log t = O(\log \mathcal{N}) = O(f(n))$ total calls to REACH. So, the total space used by M is $O(f(n)) \cdot O(f(n)) = O(f^2(n))$. ■

Remark 2.5.1. This proof is essentially the idea in the proof of **Savitch's Theorem 8.5** in [2], and as mentioned in the book, there is a small glitch in this proof. In particular, before M runs, it needs to know the value of $\mathcal{N}$, i.e it needs to know the value of $f(n)$ (because $\mathcal{N} = 2^{O(f(n))}$). But this can actually be remedied

in a simple way: just make M iterate over all possible values $f(n) \in \{1, 2, 3...\}$. For every i, check whether any configuration of length atleast $f(n) = i$ is even reachable or not. If not, then M just simply rejects. Otherwise continue the loop.

Corollary 2.5.1. $\textbf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{NSPACE}(n^k) = \bigcup_{k\in\mathbb{N}} \textbf{SPACE}(n^k) = \textbf{PSPACE}$

Proof. This easily follows from **Savitch's Theorem** 2.5. ■

Definition 2.10. We define the class **EXPTIME** as

$$\textbf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(2^{n^k})$$

Proposition 2.6. *The following inclusion between complexity classes holds.*

$$\textbf{P} \subseteq \textbf{NP} \subseteq \textbf{PSPACE} \subseteq \textbf{EXPTIME}$$

Proof. The inclusion $\textbf{P} \subseteq \textbf{NP}$ is already proven. Moreover, we know by **Corollary** 2.2.1 that any language in **NP** is decidable by a polynomial-time non-deterministic TM. Now, any polynomial-time TM can only access polynomially many tape cells on an input, and hence it follows that $\textbf{NP} \subseteq \textbf{NPSPACE}$ and hence by **Corollary** 2.5.1 we see that $\textbf{NP} \subseteq \textbf{NPSPACE}$. Finally, take any language in **PSPACE** decided in space $O(f(n))$. So, the total number of possible configurations are $O(f(n))2^{O(f(n))}$, since the number of tape cells being used is $O(f(n))$. Now, any TM that halts cannot repeat a configuration. So, the running time of this TM must be $O(f(n))2^{O(f(n))} = 2^{O(f(n))}$ and hence it follows that $\textbf{PSPACE} \subseteq \textbf{EXPTIME}$. This completes our proof. ■

Remark 2.6.1. It actually turns out that $\textbf{P} \neq \textbf{EXPTIME}$, and hence atleast one of the inclusions $\textbf{P} \subseteq \textbf{NP}$, $\textbf{NP} \subseteq \textbf{PSPACE}$ and $\textbf{PSPACE} \subseteq \textbf{EXPTIME}$ must be strict.

2.6. **PSPACE Completeness.** Just like **NP**-complete problems, we can define the notion of **PSPACE**-*completeness*.

Definition 2.11. A language L is **PSPACE**-*complete* if L is in **PSPACE** and every A in **PSPACE** is polynomial time reducible to L. If L satisfies only the second condition, then it is said to be **PSPACE**-*hard*.

Now, we will be introducing our first **PSPACE**-complete problem. The problem deals with *quantified* boolean formulae. In particular, we will be dealing with *fully quantified* boolean formulae in *prenex normal form*. An example of such a formula is

$$\phi = \forall x \exists y[(x \vee y) \wedge (\neg x \vee \neg y)]$$

It is easy to see that a fully quantified formula in prenex normal form is either true or false. We define the language

$$\text{TQBF} = \{\langle\phi\rangle \mid \phi \text{ is a true fully quantified boolean formula}\}$$

Theorem 2.7. TQBF *is* **PSPACE**-*complete.*

Proof. The proof that we covered in class was the proof of **Theorem 8.9** in [2]. ■

2.7. **Sublinear Spaces.** In this section, we will look at complexity classes where the space requirement is sublinear, i.e the space used is *less than* linear. For this to work, we will be considering TMs which have a separate input tape and a separate work tape, and only the work tape will contribute to the space complexity.

Definition 2.12. The class of languages decidable by deterministic TMs in logarithmic space is denoted by **L**, i.e

$$\mathbf{L} = \mathbf{SPACE}(\log n)$$

Similarly, the class of languages decidable by non-deterministic TMs in logarithmic space is denoted by **NL**, i.e

$$\mathbf{NL} = \mathbf{NSPACE}(\log n)$$

Now, we have to define the notion of an **NL-complete** problem, but to do that we have to define a notion of reduction.

Definition 2.13. A *log-space transducer* is a TM that has a read-only input tape, a read-write work tape and a write-only output tape. The work tape can use only $O(\log n)$ cells. A function $f : \Sigma^* \to \Sigma^*$ is said to be *log space computable* if there is some log space transducer M which writes just $f(w)$ on its output tape on input w. A language A is said to be *log space reducible* to B, written $A \leq_L B$, if there is some log space computable function f such that $x \in A \iff f(x) \in B$.

Proposition 2.8. *If $B \in$ **L** and $A \leq_L B$, then $A \in$ **L**.*

Proof. (This is loosely the argument in the proof of **Theorem 8.23** in [2]) In this proof, we can't apply the simple idea of polynomial time reductions, i.e given $x \in A$, we can't just compute $f(x)$ and run the machine for B on $f(x)$, and this is because the input $f(x)$ to B may have more than logarithmic length. Instead, the trick is to compute $f(x)$ bit-by-bit, and we now describe it.

Let M_A, M_B be log space machines for A, B respectively, and let T be the log space transducer for the reduction from A to B. Let x be an input given to A. As usual, the idea is to run M_B on input $f(x)$, but with limited space. To do this, everytime M_B requires the i^{th} bit of $f(x)$, we run T on the input x and return the i^{th} bit from the output tape, and give it to the machine M_B. Note that this procedure is very inefficient, because every time M_B moves its input head, we have to recompute $f(w)$ from scratch. However, the advantage of this is that we only need to store a single bit of $f(w)$ at a point, and hence the machine takes logarithmic space. This is the effect of trading time for space. ∎

Definition 2.14. A language B is said to be **NL**-*complete* if $B \in$ **NL** and every $A \in$ **NL** is log space reducible to B.

Now consider the following language.

$$\text{PATH} := \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$$

Theorem 2.9. *The language* PATH *as defined above is* **NL**-*complete.*

Proof. First, we need to show that PATH is in **NL**. Here is a non-deterministic log space TM for PATH: on input $\langle G, s, t\rangle$, the TM starts with the node s on its work tape. At each point, it non-deterministically guesses the next vertex in the required path, and writes that vertex on its work tape. The machine runs for

$|V|$ steps, where $|V|$ is the number of vertices in G. If the vertex t is encountered within $|V|$ steps, the machine accepts, otherwise it rejects. Clearly, at any point of time, we only need to store the current vertex, and if there are $|V|$ vertices, this requires $\log |V|$ space on the work tape. So, this machine is the required log space machine for PATH.

We will now show that every $A \in$ **NL** is log space reducible to PATH. Let M be any non-deterministic log space TM for A, and let w be any input to M. So, the work tape of M takes $c \log n$ space for some constant c, where n is the input length. Consider the *configuration graph* G of M on the input w; this is the graph whose vertices are all possible configurations of M of size $c \log |w|$ on the input w, and there is a directed edge between two configurations c_1, c_2 if M can reach the configuration c_2 from c_1 in one step. Let c_{start} be the starting configuration, and let c_{accept} be the accepting configuration for w, and we assume without loss of generality that c_{accept} is unique (if not, M can be made to have a unique accepting configuration). So, the reduction is simply mapping the word w to $\langle G, c_{\text{start}}, c_{\text{accept}} \rangle$. It remains to show that this is infact a log space reduction.

Note that the vertices of G can be computed in log space, because each vertex is a configuration which has length atmost $c \log |w|$. So the log space transducer can simply list all possible strings of length $c \log |w|$ sequentially, and check which ones are valid configurations, and the valid ones can be put on the output tape. Edges are also easy to list: the transducer simply lists all pairs (c_1, c_2) of valid configurations, and the pair is put on the output tape if c_2 can be reached from c_1 in one step. This completes our proof. ■

2.8. **NL vs co-NL.** In this section, we will prove a surprising result about the class **NL**.

Theorem 2.10 (**Immerman–Szelepcsényi Theorem**). **NL** $=$ **co-NL**.

Proof. From **Theorem** 2.9 we know that the problem PATH is **NL**-complete. So, it is enough to show that the problem $\overline{\text{PATH}}$ is in **NL**, because every problem in **co-NL** is log space reducible to $\overline{\text{PATH}}$.

So we need to give an algorithm that accepts input $\langle G, s, t \rangle$ if and only if there is no s to t path in G. We will do this in two steps:

(1) We will first give a non-deterministic log space algorithm that, on input $\langle G, s, t \rangle$, computes the number of vertices of G that are reachable from s. Call this number c.
(2) Next, we will give a non-deterministic log space algorithm that solves $\overline{\text{PATH}}$ on the input $\langle G, s, t, c \rangle$, i.e we pass an additional parameter c to the algorithm.

Clearly, combining these two algorithms will give us the required log space procedure to solve $\overline{\text{PATH}}$.

First, we give the procedure for (2), assuming we have solved (1). On input $\langle G, s, t, c \rangle$, we iterate over all vertices of G. At each step of the iteration, we only need to remember a vertex of G, and that takes $O(\log (|V|))$ space. We also maintain a counter c_{counter} that is initially 0. Let v_{curr} be the current vertex in the iteration. Then, we non-deterministically guess whether the v_{curr} is reachable from s or not. If the non-deterministic guess is a *no*, we move on to the next vertex. If the non-deterministic guess is a *yes*, then we non-deterministically try to obtain a path from s to v_{curr} (just as in the proof of **Theorem** 2.9), and

this takes time $O(\log |V|)$. If this non-deterministic process does not give a path from s to v_{curr}, then we *reject*. If the procedure gives us a path from s to v_{curr}, we increment c_{counter} by 1, and move on to the next vertex. Also, if at any stage, the machine exhibits a path from s to t, then the machine *rejects*. Finally, after iterating through all vertices, the machine checks whether $c_{\text{counter}} = c$. If not, then the machine *rejects*, otherwise the machine *accepts*.

Clearly, the above procedure only accepts if t is *not* reachable from s. We tweak our machine so that it accepts inputs $\langle G, s, t, c\rangle$ which are *not* valid encodings (remember that we have to decide the language $\overline{\text{PATH}}$; in particular, there will be inputs which are not valid encodings). Also, this procedure only needs to store the values of c_{counter} and v_{curr} and possibly some loop variables, and hence it requires only logarithmic space. So, this gives us a procedure for (2), assuming that we have solved (1).

Next, we will show how to solve (1). We will give a non-deterministic algorithm, one of whose branches computes the value of c correctly, and all other branches reject.

Let the input be $\langle G, s, t\rangle$. For each $0 \leq i \leq m$, denote the set of all vertices of G reachable from s within i steps by A_i, i.e $A_i = \{s\}$. Clearly, A_m is the set of all vertices reachable from s. Let $c_i = |A_i|$, so that $c_0 = 1$. We will give a non-deterministic procedure that calculates A_{i+1} from A_i, and running this procedure $m - 1$ times will give us the value $c_m = c$.

So suppose we know c_i, and we want to calculate c_{i+1}. We put $c_{i+1} = 0$ initially. We iterate through all the vertices of G, and this forms our outer loop. Suppose v_{curr} is the current vertex in this loop. Next, we form an inner loop to iterate through all the vertices of G, and let u_{curr} be the current vertex in the inner loop. We non-deterministically guess whether $u_{\text{curr}} \in A_i$ or not. If the guess is a *no*, then we simply move to the next vertex in the inner loop. If the guess is a *yes*, then we non-deterministically try to obtain a path of length atmost i from s to u_{curr}. If we don't get such a path, we *reject*. If we do get such a path, then we check whether $(u_{\text{curr}}, v_{\text{curr}})$ is an edge in G. If it is an edge, we increment c_{i+1} by 1 (because in this case $v_i \in A_{i+1}$). Through the inner loop, we also keep a count of the number of vertices found in A_i. If this count is not equal to c_i, we *reject*, because not all vertices in A_i have been found. Then, we go on to the next vertex in the outer loop.

Clearly, only a few variables need to be stored in the above procedure, and hence it takes $O(\log n)$ time. So this is the required procedure for (1), and hence this completes our proof. ■

Remark 2.10.1. The ideas here are loosely the contents of **Theorem 8.27** of [2].

3. Intractability

In this section we will see some problems which are solvable in principle, but they require a lot of space and time, which makes them practically impossible to solve. This is the notion of *intractability*.

3.1. **Hierarchy Theorems.** Let us first begin with the notion of *space constructability*.

Definition 3.1. Let $f : \mathbb{N} \to \mathbb{N}$ be a function, where $f(n)$ is atleast $O(\log n)$. $f(n)$ is said to be *space constructible* if the function that maps the string 1^n to the

binary representation of $f(n)$ is computable in $O(f(n))$ space. In simple words, $f(n)$ is *space constructible* if there is an $O(f(n))$ space TM that halts with output equal to the binary representation of $f(n)$ on input 1^n.

Theorem 3.1 (**Space Hierarchy Theorem**). *For any space constructible function $f : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(f(n))$ space but not in $o(f(n))$ space.*

Proof. (The proof is based on the ideas in **Theorem 9.3** of [2]) The idea is to make a TM that runs in $O(f(n))$ space and is *different* from every TM that runs in $o(f(n))$ space. We will accomplish this by using a *diagonalisation argument*, something very similar to what is used in proving that the halting problem is undecidable.

We make a TM D as follows: D takes as input all strings x of the form $\langle M \rangle 10^*$, where $\langle M \rangle$ is the encoding of some TM. If the input x is not of this form, then our machine D will simply reject. So, we assume that x is of the given form. So on input $x = \langle M \rangle 10^k$, D simulates the machine M on the the same input $x = \langle M \rangle 10^k$ within the space bound $f(n)$ (this is where we use space constructability of f). If M accepts x within this space bound, then D rejects. If M rejects within this space bound, then D accepts. If M does not halt within this space bound, D simply accepts.

It is clear from this description that D runs in space $O(f(n))$. Next, we need to show that D is *not* equivalent to any TM that runs in space $o(f(n))$. Let M be any TM that runs in space $o(f(n))$, and let n_0 be so large so that M runs within $f(n)$ space on all inputs of size $n \geq n_0$. Consider the string $\langle M \rangle 10^{n_0}$. Clearly, D differs in its behavior from M on input $\langle M \rangle 10^{n_0}$. This completes the proof. ■

Remark 3.1.1. There is a technical glitch in the above proof. It is possible that when D simulates M, M may go in an infinite loop within space $f(n)$. Now, any machine that halts within $f(n)$ space must run for time atmost $2^{f(n)}$. So, we configure D so that it runs for time atmost $2^{f(n)}$.

Corollary 3.1.1. *For any two functions $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ where $f_1(n)$ is $o(f_2(n))$ and f_2 is space constructible,* **SPACE**$(f_1(n)) \subsetneq$ **SPACE**$(f_2(n))$. *In simple words, if we are allowed more time, then we can decide more problems.*

Proof. This is immediate from the **Space Hierarchy Theorem** 3.1. ■

Example 3.1. It can be shown that the function n^c is space constructible for every $c \in \mathbb{Q}^+$. This immediately tells us that if $0 \leq c_1 < c_2$ are two real numbers, then **SPACE**$(n^{c_1}) \subsetneq$ **SPACE**(n^{c_2}).

Example 3.2. Using **Corollary** 3.1.1, we can show that **NL** $\subsetneq$ **PSPACE**. By **Savitch's Theorem** 2.5, we know that **NL** $\subseteq$ **SPACE**$(\log^2(n))$. Since $\log^2(n) = o(n)$, **Corollary** 3.1.1 immediately implies that **SPACE**$(\log^2 n) \subsetneq$ **SPACE**(n), and hence it follows that **NL** $\subsetneq$ **SPACE**$(n) \subsetneq$ **PSPACE**.

Definition 3.2. A function $t : \mathbb{N} \to \mathbb{N}$ such that $t(n)$ is atleast $O(n \log n)$ is said to be *time-constructible* if the function mapping 1^n to the binary representation of $t(n)$ is computable in time $O(t(n))$.

Theorem 3.2 (**Time Hierarchy Theorem, Deterministic Version**). *For any time constructible function $t : \mathbb{N} \to \mathbb{N}$ there exists a language A that is decidable in time $O(t(n))$ but is not decidable in time $o(t(n)/\log t(n))$.*

Proof. (The idea is based on **Theorem 9.10** of [2]) The proof idea is very similar to the proof of the **Space Hierarchy Theorem** 3.1. However, here there is a factor of $\log t(n)$ because we need to make sure that our machine takes atmost $t(n)$ steps to run. This will be more clear in the following algorithm.

We make a TM D which does the following, and note that this is very similar to what we did in the **Space Hierarchy Theorem** 3.1.

(1) Let the input be w, and let $n = |w|$.
(2) Since t is time constructible, compute $t(n)$ and store the value $t(n)/\log t(n)$ in a binary counter. We will use this counter to capture the number of steps; so, we decrement this counter before every step in (4) below. If the counter reaches 0, then reject.
(3) If w is not of the form $\langle M \rangle 10^*$, then simply reject.
(4) If w is of the form $\langle M \rangle 10^k$, simulate M on w. If M accepts, then reject and if M rejects, then accept.

Note that at every step, the counter in point number (2) above is decremented. The counter has size $t(n)/\log t(n)$, and hence the space used to store this counter is $\log\left(t(n)/\log t(n)\right)$, and this is $O(\log t(n))$. So, the cost of decrementing the counter at every step is $O(\log t(n))$, and hence the overall cost of decrementing the counter until it is zero is $O(\log t(n))t(n)/\log t(n) = O(t(n))$. So, it follows that our machine D runs in time $O(t(n))$. So suppose D runs in time $ct(n)$ for some constant c.

Next, we claim that the language decided by D *cannot* be decided by any TM with runtime $o(t(n)/\log t(n))$. For the sake of contradiction, suppose there is some $o(t(n)/\log t(n))$ time machine M such that $L(M) = L(D)$. Let $n_0 \in \mathbb{N}$ be such that M takes less than $ct(n_0)/\log t(n_0)$ steps to run on an input of length $\geq n_0$. This means that if D runs on input $\langle M \rangle 10^{n_0}$, then the simulation of M will be complete. But this is a contradiction, because by our definition, the behavior of D is different from M on this input. ■

Theorem 3.3 (**Time Hierarchy Theorem, Non-Deterministic Version**). *If* $f(n+1) = o(g(n))$ *then* **NTIME**$(f(n)) \subsetneq$ **NTIME**$(g(n))$.

Proof. This is given in **Theorem 3.3** of [1]. I may want to write the proof done in class here, but I think its similar to proof in Borak's book. ■

3.2. **The Polynomial Time Hierarchy.** In this section, we will review an important class of problems, which extends the classes **P** and **NP**.

Definition 3.3. Let $\Sigma_0^p =$ **P** and $\Sigma_1^p =$ **NP**. The class Σ_2^p is defined to be the set of all languages L for which there exists a polynomial time TM M and a polynomial $q(x)$ such that

$$x \in L \iff \exists u \in \{0,1\}^{q(|x|)} \forall v \in \{0,1\}^{q(|x|)} \langle x, u, v \rangle \in L(M)$$

Note the similarity of the definition of Σ_2^p with the definition of **NP**. Here, we have an extra $\forall$ quantifier. It is also clear that **NP** $\subseteq \Sigma_2^p$.

Definition 3.4. Define the class Π_2^p to be

$$\Pi_2^p := \{L : \overline{L} \in \Sigma_2^p\}$$

Equivalently, we have the following definition: a language L is in Π_2^p if there exists a polynomial time TM M and a polynomial $q(x)$ such that

$$x \in L \iff \forall u \in \{0,1\}^{q(|x|)} \exists v \in \{0,1\}^{q(|x|)} \langle x, u, v \rangle \in L(M)$$

So, we have just flipped the two quantifiers here.

Definition 3.5. For every $i \geq 1$, we say that a language L is in Σ_i^p if there is a polynomial time TM M and a polynomial $q(x)$ such that

$$x \in L \iff \exists u_1 \in \{0,1\}^{q(|x|)} \forall u_2 \in \{0,1\}^{q(|x|)} \exists u_3 .... Q_i u_i \in \{0,1\}^{q(|x|)} \langle x, u_1, u_2, ..., u_i \rangle \in L(M)$$

where Q_i is the quantifier $\exists$ if i is odd, and it is $\forall$ if i is even.

Similarly, we can define the class Π_2^p as the set of all languages L such that there exists a polynomial time TM M and a polynomial $q(x)$ such that

$$x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \forall u_3 .... Q_i u_i \in \{0,1\}^{q(|x|)} \langle x, u_1, u_2, ..., u_i \rangle \in L(M)$$

where Q_i is the quantifier $\forall$ if i is odd, and it is $\exists$ if i is even.

Remark 3.3.1. Note that $\Sigma_1^p = \mathbf{NP}$ and $\Pi_1^p = \mathbf{coNP}$. More generally, we have $\Pi_i^p = \mathbf{co}\Sigma_i^p$. Also note that $\Sigma_i^p \subseteq \Pi_{i+1}^p$.

Definition 3.6. The *polynomial hierarchy* is defined as

$$\mathbf{PH} := \bigcup_i \Sigma_i^p$$

Proposition 3.4. *For every $i \geq 1$, if $\Sigma_i^p = \Pi_i^p$ then $\mathbf{PH} = \Sigma_i^p$. So in this case, the hierarchy **PH** collapses to the i^{th} level.*

Proof. Didnt' have enough time to write the proof. But, I like the proof of **Theorem 2** in http://www.cs.umd.edu/~jkatz/complexity/f11/lecture8.pdf. ■

Proposition 3.5. $\mathbf{PH} \subseteq \mathbf{PSPACE}$.

Proof. To be completed. ■

Theorem 3.6. *Suppose there is some language L that is **PH**-complete. Then, there exists an i such that $\mathbf{PH} = \Sigma_i^p$.*

Proof. This is easy to prove: suppose L is a **PH**-complete language. So, there is some i such that $L \in \Sigma_i^p$. Then, every problem of **PH** can be polynomially reduced to L, and hence every language in **PH** is in Σ_i^p. ■

Theorem 3.7. *Let $i \in \mathbb{N}$. Define the following problems.*

$$\Sigma_i\text{SAT} := \{\varphi \mid \exists u_1 \forall u_2 \exists \cdots Q_i u_i \, \varphi(u_1, ..., u_i) = 1\}$$
$$\Pi_i\text{SAT} := \{\varphi \mid \forall u_1 \exists u_2 \forall \cdots S_i u_i \, \varphi(u_1, ..., u_i) = 1\}$$

Then, the problem Σ_iSAT is a complete problem for Σ_i^p and the problem Π_iSAT is a complete problem for Π_i^p.

Proof. To be completed. ■

3.3. **Oracle Computations.** Now let us look at an important abstract idea.

Definition 3.7. An *oracle* for a language A is a device that can report if a string w is a member of A or not. An *oracle Turing Machine* M^A is a TM which can query the oracle for A, and has a special *oracle tape*: whenever M^A writes a string w on its oracle tape, the oracle for A can report to M whether $w \in A$ in a single computational step. M^A is said to have *access* to an oracle for A.

Remark 3.7.1. The best way to think about *oracles* is to think about them as being *black boxes* to solve certain problems for free. This is also called *relativisation*, i.e we study properties of computation relative to a given problem, i.e if we can solve a certain problem for free, how much power do we have to solve other problems.

Definition 3.8. Let A be any language. Denote by $\mathbf{P}^A$ the class of all languages solvable by polynomial time TMs having access to the oracle A. Similarly, the class $\mathbf{NP}^A$ is defined.

Theorem 3.8. *There exists oracles A, B such that $\mathbf{P}^A \neq \mathbf{NP}^A$ and $\mathbf{P}^B = \mathbf{NP}^B$.*

Proof. Let $B =$ TQBF. Then we have the following inclusions.

$$\mathbf{NP}^{\text{TQBF}} \subseteq \mathbf{NSPACE} \subseteq \mathbf{PSPACE} \subseteq \mathbf{P}^{\text{TQBF}}$$

The first inclusion is true because any problem in $\mathbf{NP}^{\text{TQBF}}$ can simply solve any instance of TQBF in polynomial space instead of using the oracle for it. The second inclusion is true by **Savitch's Theorem** 2.5, and the last inclusion is true because TQBF is **PSPACE**-complete. This shows that $\mathbf{P}^B = \mathbf{NP}^B$.

Next, we will exhibit the oracle A. Complete this proof! The proof done in class was **Theorem 9.20** of [2]. ■

3.4. **Circuit Complexity.** First, we review *boolean circuits*.

Definition 3.9. Formally, a *boolean circuit* is a directed acyclic graph in which the leaves are labelled by variables $x_1, ..., x_n$ and the rest of the vertices are labelled by logical gates (AND, OR and NOT). The edges of a boolean circuit are called *wires*.

An example of a boolean circuit is the following.

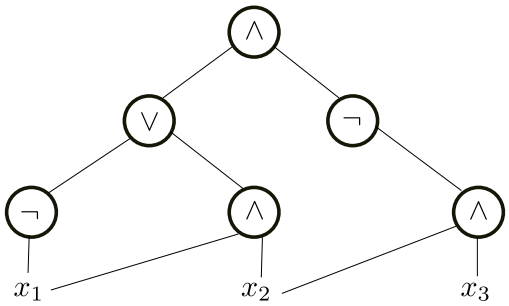


Figure 1. Circuit having three variables

Definition 3.10. Let C be a boolean circuit with n variables. We associate a function $f_C : \{0,1\}^n \to \{0,1\}$ as follows: for any $(x_1, ..., x_n) \in \{0,1\}^n$, we let $f(x_1, ..., x_n)$ to be the output of the circuit C on these input variables.

Definition 3.11. A *circuit family* C is an infinite sequence of circuites $\{C_n\}_{n\in\mathbb{N}}$. We say that the family C *recognizes* a language $L \subseteq \{0,1\}^*$ if circuit C_n correctly decides the membership of the language $L_{=n}$, where

$$L_{=n} := \{x \in L \mid |x| = n\}$$

Definition 3.12. The *size* of a circuit C (denoted by $|C|$) is the number of gates that it contains. The *depth* of a circuit C is the length of the longest path from an input variable to an output gate. A language L is said to have a $T(n)$*-size circuit family* if there is a circuit family $C = \{C_n\}$ recognizing L such that $|C_n| \leq T(n)$ for all $n \in \mathbb{N}$. A language A is said to have *circuit complexity* $T(n)$ if the size-minimal family of circuits recognizing A has size $T(n)$.

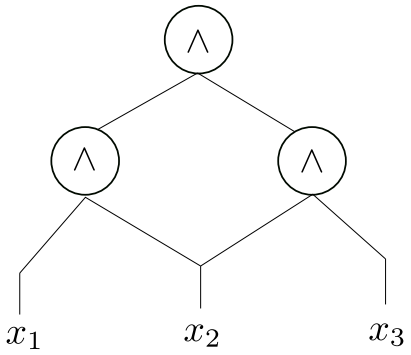
Definition 3.13. Define **P**/**poly** to be the class of problems that can be decided by polynomial size circuit families.

Theorem 3.9. **P** $\subseteq$ **P**/**poly**. *More precisely, let* $t : \mathbb{N} \to \mathbb{N}$ *be a function with* $t(n) \geq n$. *if* $A \in$ **DTIME**$(t(n))$, *then* A *has circuit complexity* $O(t^2(n))$.

Proof. In class, we covered the proof done in **Theorem 9.30** of [2]. ■

Theorem 3.10. *The class* **P**/**poly** *contains an undecidable problem. Hence,* **P** $\subsetneq$ **P**/**poly**.

Proof. First, we show that every unary language $L \subseteq \{1\}^*$ is contained in **P**/**poly**. For $n \in \mathbb{N}$, let C_n be the trivial circuit which rejects every input if $1^n \notin L$, and if $1^n \in L$, let C_n be the boolean circuit which consists of only AND gates. For instance, if $1^3 \in L$ then the circuit C_3 looks something like the circuit given below.



(Important point: we are *not saying* that we know how to *find* the $n \in \mathbb{N}$ for which $1^n \in L$. We are just claiming the existence of a polynomial-size circuit family that accepts this language). Clearly, the circuit family $\{C_n\}$ is polynomial sized, and it accepts L. This shows that $L \in$ **P**/**poly**.

Now, consider the following unary language:

$$u - \text{HALT} := \{1^n \mid n \text{ encodes } \langle M, x\rangle \text{ in binary such that } M \text{ halts on } x\}$$

Being a unary language, $u - \text{HALT} \in$ **P**/**poly**. We claim that $u - \text{HALT} \notin$ **P** (and infact, this problem is undecidable), and this will complete the proof of the claim. The fact that this problem is undecidable is clear, because if this problem is decidable, it would mean that the halting problem HALT is also decidable, which is a contradiction. ■

Theorem 3.11. *The class* **P** *is equal to the class of problems decidable by polynomial-size circuit families whose descriptions can be obtained in polynomial time.*

Remark 3.11.1. Such circuit families are called *uniformly-generated* circuit families.

Proof. To be completed. ■

Theorem 3.12 (**Karp-Lipton Theorem**). *If* **NP** $\subseteq$ **P**/**poly** *then* **PH** $= \Sigma_2^p$.

Remark 3.12.1. This theorem is basically saying that it is very unlikely that **NP** is a subset of **P**/**poly**.

Proof. Recall the if it is true that $\Pi_i^p \subseteq \Sigma_i^p$, then **PH** $= \Sigma_2^p$ (see **Proposition** 3.4). To prove this theorem, we will show that if **NP** $\subseteq$ **P**/**poly**, then $\Pi_2^p \subseteq \Sigma_2^p$, and that will complete our proof. Complete this. ■

4. Modern Complexity

4.1. **Randomized Complexity.**

First, we will define the notion of a *probabilistic* TM.

Definition 4.1. A *Probabilistic Turing Machine* is a non-deterministic TM in which each non-deterministic step is based on a *coin toss*. If a branch of computation tosses a coin t times, then the probability associated to that branch is $1/2^t$.

Definition 4.2. We say a probabilistic TM M accepts a string x if and only if

$$\mathbb{P}(M \text{ accepts } x) \geq \frac{2}{3}$$

We define the class **BPP** (*Bounded Probabilistic Polymomial time*) to be the set of all languages which are accepted by a probabilistic polynomial time TM.

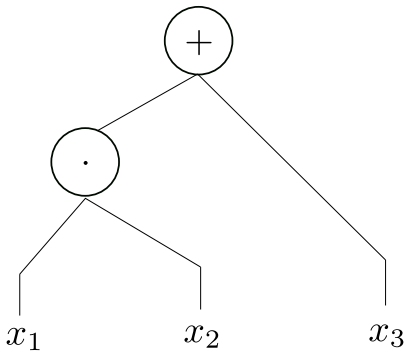
Proposition 4.1. **BPP** $\subseteq$ **EXP**

Proof. The idea is simple: just traverse the computation tree in a BFS fashion. Clearly, this takes exponential time, because the number of nodes in the computation tree for a string will be exponential in the size of the string. ■

Theorem 4.2. **BPP** $\subseteq \Sigma_2^p \cap \Pi_2^p$

4.2. **Algebraic Circuits.**

An *algebraic circuit* is just like a boolean circuit, where the gates are + and - gates (which represent addition and subtraction). These gates compute multivariate polynomials. For example, consider the following algebraic circuit.



The above boolean circuit will compute the polynomial $x_1x_2 + x_3$, which is a polynomial over three variables. The good thing about these circuits is that small algebraic circuits can compute big polynomials. For example, consider the polynomial

$$p(x_1, ..., x_n) = (1 + x_1)(1 + x_2) \cdots (1 + x_n)$$

It can be checked that this polynomial has $O(2^n)$ monomials. However, there is a simple algebraic circuit to compute this polynomial which has only $n+1$ gates (finding such a circuit is almost trivial).

4.3. **Polynomial Identity Testing.** In this section, we will look at a problem which has a fairly simple randomized algorithm that solves it.

Suppose we are given an algebraic circuit C which computes a polynomial P of degree $\leq d$. We need to determine whether $P \equiv 0$, i.e whether P is the zero polynomial. It is an *open problem* to determine whether PIT $\in$ **P**.

Lemma 4.3 (**Schwartz-Zippel Lemma**). *Let $P(x_1, ..., x_n) \in \mathbb{Q}[x_1, ..., x_n]$ be a non-zero polynomial of degree d. Let $S \subseteq \mathbb{Q}$ be any finite subset of $\mathbb{Q}$. If $a_1, a_2, ..., a_n$ are picked uniformly at random from the set S, then*

$$\mathbb{P}[P(a_1, ..., a_n) = 0] \leq \frac{d}{|S|}$$

Remark 4.3.1. For polynomials in one variable, this lemma is trivial, because in that case P can have atmost d roots (since we are working over a field).

Proof. We prove this by induction on n. As mentioned in the remark, the base case $n = 1$ is trivially true, because a polynomial of degree D can have atmost d roots. Assume that the lemma holds for those cases when the number of variables is $\leq n - 1$. Now, we can write the polynomial $P(x_1, ..., x_n)$ as a polynomial over the variable x_n, where the coefficients will be taken from the ring $\mathbb{Q}[x_1, ..., x_{n-1}]$; more formally, we are using the isomorphism $\mathbb{Q}[x_1, ..., x_n] \cong \mathbb{Q}[x_1, ..., x_{n-1}][x_n]$. We will use the notation $\overline{x} = (x_1, x_2, ..., x_n)$. So, let

$$P(\overline{x}) = \sum_{i=0}^{\tilde{d}} x_n^i P_i(x_1, ..., x_{n-1})$$

where each $P_i(x_1, ..., x_{n-1}) \in \mathbb{Q}[x_1, ..., x_{n-1}]$ and $0 \leq \tilde{d} \leq d$. Since $P(\overline{x}) \not\equiv 0$, there is a maximum index $j \leq d$ such that such that $P_j(x_1, ..., x_{n-1}) \not\equiv 0$. So, we can write

$$P(\overline{x}) = \sum_{i=0}^{j} x_n^i P_i(x_1, ..., x_{n-1})$$

Also, note that $\deg P_j \leq d - j$. By induction hypothesis, we see that

$$\mathop{\mathbb{P}}_{a_1,...,a_{n-1} \in S}[P_j(a_1, ..., a_{n-1}) = 0] \leq \frac{d-j}{|S|}$$

Now, we have

$$P(a_1, ..., a_{n-1}, x_n) = \sum_{i=0}^{j} x_n^i P_i(a_1, ..., a_{n-1})$$

Now, we condition the event $P(a_1, ..., a_{n-1}, a_n) = 0$ into two events.

(1) In the first case, $P(a_1, ..., a_{n-1}, a_n) = 0$ and $P_j(a_1, ..., a_{n-1}) = 0$. We have

$$\mathbb{P}[P(a_1, ..., a_{n-1}, a_n) \wedge P_j(a_1, ..., a_{n-1}) = 0] \leq \mathbb{P}(P_j(a_1, ..., a_{n-1}) = 0) \leq \frac{d-j}{|S|}$$

(2) In the second case, $P(a_1, ..., a_{n-1}, a_n) = 0$ and $P_j(a_1, ..., a_{n-1}) \neq 0$. Now, observe

$$P(a_1, ..., a_{n-1}, x_n) = \sum_{i=0}^{j} x_n^i P_i(a_1, ..., a_{n-1})$$

and the above polynomial is a polynomial in one variable. So in this case we have

$$\mathbb{P}[P(a_1, ..., a_{n-1}, a_n) = 0] \leq \frac{j}{|S|}$$

So, it follows that

$$\mathbb{P}[P(a_1, ..., a_{n-1}, a_n) = 0] \leq \frac{d-j}{|S|} + \frac{j}{|S|} = \frac{d}{|S|}$$

and this completes the proof. ■

Corollary 4.3.1. *The number of zeroes of $P(x_1, ..., x_n)$ in the box $S \times S \times \cdots \times S = S^n$ is atmost $d \cdot |S|^{n-1}$.*

Proof. The proof is immediate; suppose k is the number of roots in the box $S \times S \times \cdots \times S = S^n$. Then, the above lemma implies

$$\frac{k}{|S|^n} \leq \frac{d}{|S|}$$

and hence it follows that $k \leq d \cdot |S|^{n-1}$, and this is what we wanted to prove. ■

4.3.1. *A simple randomized algorithm for PIT.* Using **Corollary** 4.3.1, we can devise a simple randomized algorithm for the polynomial identity testing problem.

(1) Suppose the input is a circuit for $f(x_1, ..., x_n) \in \mathbb{Q}[x_1, ..., x_n]$, where $\deg(f) \leq d$.
(2) Take $S \subseteq \mathbb{Q}$ such that $|S| = 100d$.
(3) Pick a vector $(a_1, ..., a_n) \in S^n$ uniformly at random. Use this input in the circuit, and check whether the output of the circuit is non-zero.
(4) If the output of the circuit is zero, return true. Otherwise return false.

By the **Schwartz-Zippel Lemma** 4.3, we see that if $f \not\equiv 0$, then the probability that the above algorithm returns true is less than $\frac{d}{100d} = \frac{1}{100}$. So, this algorithm is the required randomized algorithm for polynomial identity testing.

Remark 4.3.2. Finding an efficient deterministic algorithm for PIT is still an open problem.

4.4. **Bipartite Matching.** Suppose we are given a bipartite graph G with bipartition $V_1 \cup V_2$ such that $|V_1| = |V_2| = n$. Make a matrix A_G as follows, which we will call the *symbolic adjacency matrix* of G (this will not be the usual adjacency matrix): suppose the vertices in V_1 and V_2 are both labelled with labels $1, 2, ..., n$. Consider n^2 variables x_{ij} for $1 \leq i, j \leq n$. A_G is an $n \times n$ matrix such that

$$A_G[ij] = \begin{cases} x_{ij} & , \text{ if vertex } i \text{ in } V_1 \text{ is connected to vertex } j \text{ in } V_2 \\ 0 & , \text{ otherwise} \end{cases}$$

Now, consider $\det(A_G)$, and look at it as a polynomial over n^2 variables. The following equivalence is clear using the permutation expansion of the determinant:

$$\det(A_G) \neq 0 \iff G \text{ has a perfect matching}$$

So, we immediately get a randomized algorithm for existence of perfect matchings: use the polynomial identity testing to check whether $\det(A_G) \not\equiv 0$.

4.5. **Error Reduction and Adelman's Theorem.** One of the simplest statistical tricks to reduce the error of a randomized machine M is to run many independent trials of the machine M on the input. In regard to this, we have the following lemma.

Proposition 4.4 (**Amplification Lemma**). *Let ϵ be a fixed number between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$, a probabilistic polynomial time TM M_1 that operates with an error ϵ has an equivalent probabilistic polynomial time TM M_2 that operates with an error of $2^{-p(n)}$.*

Proof. The main idea is to run many independent trials of the machine M_1. The details of the proof were not covered, but are given in **Lemma 10.5** of [2]. ■

Theorem 4.5 (**Adelman**). **BPP** $\subseteq$ **P**/**poly**

Proof. Suppose $L \in$ **BPP**. By the **Amplification Lemma** 4.4, there is a TM M that on inputs of size n uses m random bits and satisfies

$$\mathbb{P}_{r\in\{0,1\}^m}[M(x,r) \neq L(x)] \leq \frac{1}{2^{n+2}}$$

Say a random bit $r \in \{0,1\}^m$ is *bad* if for an input $x \in \{0,1\}^n$, $M(x,r)$ is an incorrect answer, i.e $M(x,r) \neq L(x)$. Otherwise, we say it is *good* for x. By the given probability bound, at most $2^m/2^2n + 2$ values of r are bad for x. Adding over all $x \in \{0,1\}^n$, we conclude that there are atmost $2^n \times 2^m/2^{n+2} < 2^m$ strings r which are bad for some x. So, there is some string r that is good for every $x \in \{0,1\}^n$. We can hardwire such a string r into a circuit C_n that correclty decides the language L. ■

4.6. **A better bound on BPP.** In this section, we will prove a good upper bound for the class **BPP**.

Theorem 4.6. **BPP** $\subseteq \Sigma_2 \cap \Pi_2$

Proof. It is enough to show that **BPP** $\subseteq \Sigma_2$, since **BPP** is closed under complementation.

Suppose $L \in$ **BPP**. So, there is a randomized polynomial time TM M such that

$$\mathbb{P}_r[\text{error}] \leq \frac{1}{2^n}$$

for all inputs $x \in \{0,1\}^n$ and random bits $r \in \{0,1\}^m$. For $x \in \{0,1\}^n$, let

$$S_X := \{r \in \{0,1\}^m \mid M(x,r) = 1\}$$

By the given probability bound, we immediately see that

$$x \in L \implies |S_x| \geq \left(1 - \frac{1}{2^n}\right) 2^m$$

$$x \notin L \implies |S_x| \leq \frac{2^m}{2^n} = 2^{m-n}$$

Let $S \subseteq \{0,1\}^m$. Given any $y \in \{0,1\}^m$, the translate $S + y$ of S is defined as

$$S + y := \{s + y \mid s \in S\}$$

We will show the following: if the size of S is *small* then a *small* number of translates of S can not cover $\{0,1\}^m$ (we will make this more precise in a moment). However, if the size of S is *sufficiently large*, then there is a small set of translates of S which can cover $\{0,1\}^m$. Let us now show this.

Let $k = \frac{m}{n} + 1$. Suppose $S \subseteq \{0,1\}^m$ such that $|S| \leq 2^m/2^n$ (so that S is a *small* set). Let $S + u_1, ..., S + u_k$ be any translates of S. Then we have

$$\left| \bigcup_{i=1}^{k} S + u_i \right| \leq \left(\frac{m}{n} + 1 \right) \frac{2^m}{2^n} < 2^m$$

and hence any k translates of S cannot cover $\{0,1\}^m$ (this proves the first part of the statement made at the end of the above paragraph).

Next, we prove the second statement. We will be using the so-called *probabilistic method*. The key observation is this: if S is any subset of $\{0,1\}^m$ then

$$r \in S + u \iff r + u = s \in S$$

and this is just a consequence of addition modulo 2. Now, fix $r \in \{0,1\}^m$. Then by this observation,

$$\mathop{\mathbb{P}}_{u \in \{0,1\}^m}[r \in S + u] = \mathop{\mathbb{P}}_{u \in \{0,1\}^m}[r + u \in S] \leq \frac{|S|}{2^m}$$

Now, let $S \subseteq \{0,1\}^m$ be such that

$$|S| \geq \left(1 - \frac{1}{2^n}\right) 2^m$$

(i.e S is a sufficiently big set). Fix $r \in \{0,1\}^m$. Then,

$$\mathop{\mathbb{P}}_{u_1,...,u_k \in \{0,1\}^m} \left[r \notin \bigcup_{i=1}^{k} S + u_i \right] \leq \left(\frac{2^m/2^n}{2^m} \right)^k = \frac{1}{2^{nk}} = 2^{-nk}$$

Now, if $\left(\bigcup_{i=1}^{k} S + u_i \right)$ is not a cover of $\{0,1\}^m$ then there is some $r \in \{0,1\}^m$ such that $r \notin \bigcup_{i=1}^{k} S + u_i$ For each $r \in \{0,1\}^m$, let E_r be the event that $r \notin \bigcup_{i=1}^{k} S + u_i$. Above we have shown that

$$\mathbb{P}[E_r] \leq 2^{-nk}$$

and hence it follows that

$$\mathbb{P}\left[\bigcup_r E_r \right] \leq \sum_r \mathbb{P}[E_r] \leq 2^m 2^{-nk} = 2^m 2^{-n\left(\frac{m}{n}+1\right)} < 1$$

So, it follows that there are $u_1, ..., u_k$ such that

$$\bigcup_{i=1}^{k} S + u_i = \{0,1\}^m$$

and this proves the second statement.

Now, returning back to our original problem. By what we have shown thus far, we have

$$x \in L \iff |S_x| \geq \left(1 - \frac{1}{2^n}\right) 2^m \iff \exists u_1, ..., u_k \; \forall r \in \{0,1\}^m \; r \in \bigcup_{i=1}^{k} S_x + u_i$$

This is very close to a language in Σ_2. The only thing we need to do is convert the statement

$$r \in \bigcup_{i=1}^{k} S_x + u_i$$

to a logical predicate. We use the machine M to do this. Again, observe that

$$r \in u_i + S_x \iff r + u_i \in S_x \iff M(x, r + u_i) = 1$$

and hence

$$x \in L \iff \exists u_1, ..., u_k \ \forall r \in \{0,1\}^m \ \vee_{i=1}^k M(x, r + u_i)$$

■

4.7. **Chernoff Bound and More on Randomized Algorithms.** First, let us begin with the *Chernoff Bound*.

Proposition 4.7 (**Chernoff Bound**). *Let $X_1, ..., X_k$ be independent boolean random variables such that*

$$\mathbb{P}[X_i = 1] = p$$

for each $1 \leq i \leq k$. Let

$$X = \sum_{i=1}^{k} X_i$$

and we see that

$$\mathbb{E}[X] = \sum_{i=1}^{k} \mathbb{E}[X_i] = pk$$

Then

$$\mathbb{P}\left[|X - \mu| > \delta\mu\right] \leq e^{\frac{-\delta^2}{4}\mu}$$

Proof. Look up a standard probability textbook. ■

Now, suppose we have a randomized algorithm A which works within a two-sided error bound of $1/3$. As in the **Amplification Lemma** 4.4, we can always reduce the error of this algorithm by repeating the algorithm independently many times, and judging my the major outcome (i.e if majority of times the algorithm accepts, then we accept, otherwise we reject). Let us look into this in a bit more detail. So suppose we have a string x with $|x| = n$, and suppose we run the algorithm A on x with k random bits $r_1, ..., r_k$ which are picked uniformly at random. For each $1 \leq i \leq k$, let

$$X_i = \begin{cases} 1 & \text{if } A(x, r_i) \text{ decides the membership of } x \text{ correctly} \\ 0 & \text{otherwise} \end{cases}$$

Clearly, we see that

$$\mathbb{E}[X_i] \geq \frac{2}{3}$$

If we put $X = \sum_i X_i$ then we have

$$k \geq \mathbb{E}[X] \geq \frac{2}{3}k$$

(1) Consider the case when $x \in L$, and the algorithm A incorrectly decides the membership of L. Clearly, this can happen only if $X \leq \frac{1}{2}k$ (i.e more than half random bits incorrectly decide the membership of x in L). In this case, we have

$$|X - \mathbb{E}[X]| \geq \left|\frac{1}{2}k - \frac{2}{3}k\right| = \frac{1}{6}k \geq \frac{1}{6}\mathbb{E}[X]$$

and then one can use the **Chernoff Bound** 4.7 to show that

$$\mathbb{P}\left[X \leq \frac{1}{2}k\right] \leq e^{-\Omega(k)}$$

If k is some polynomial of n, then

$$e^{-\Omega(k)} \leq \frac{1}{\exp(n)}$$

This shows that repeating the algorithm A polynomially many times will give an exponentially low probability of error.

4.8. **Randomized Space Bounded Computation.** First we begin with a definition.

Definition 4.3. **RL** denotes the class of all languages which are recognizable by a randomized log-space machine with a one-sided error bound.

Example 4.1. Consider the problem UREACH defined as

$$\text{UREACH} := \{\langle G, s, t\rangle \mid \text{There is a path from } s \text{ to } t \text{ in the undirected graph } G\}$$

In 2004, it was proven that UREACH$\in$ **L**, a result which is very hard to prove. The following theorem is a bit easier to prove.

Theorem 4.8. **UREACH** $\in$ **RL**

Proof. Our randomized algorithm will have this following vague description: suppose the input is $\langle G, s, t\rangle$.

(1) Start a random walk from s.
(2) If at any time you hit the vertex t, accept. Continue the random walk for a certain number of steps, which will be polynomial in n, where n is the length of our input.
(3) If t is never found, reject.

Clearly, the algorithm above will only have a one-sided error: if s and t are in different connected components, then the above algorithm will never make a mistake. Let us explore this in more detail.

First, let us restrict ourselves to the case when G is a d-regular graph. Consider A_G, the adjacency matrix of G. Now, we consider the normalised matrix A_G/d. Clearly, this is a *doubly stochastic matrix*, i.e the sum of the entries in each row and column of this matrix is 1. Now, A_G is an $n \times n$ symmetric matrix, where n is the number of vertices in G. So, all the eigenvalues of A_G are real, and we can find a set of eigenvectors which form an orthonormal basis of $\mathbb{R}^n$. Now, it can be easily checked that the vector $u = (1/\sqrt{n}, \cdots, 1/\sqrt{n}) \in \mathbb{R}^n$ is a unit eigenvector of A_G with eigenvalue 1. So, 1 is an eigenvalue of A_G. It can be shown that all eigenvalues of a stochastic matrix have absolute value atmost 1. Now, let $1 = \lambda_1, \lambda_2, ..., \lambda_n$ be all the eigenvalues of A_G, arranged in an order so that

$$1 \geq |\lambda_1| \geq \cdots \geq |\lambda_n|$$

Complete this analysis!

Now, suppose that we have a graph G which is not necessarily regular. We will convert this graph to a 3-regular graph, such that the reachability between s, t in G is preserved in this new graph. The idea is to create a new graph G' as follows: for every vertex v in the graph with degree d_v, create a d_v cycle, and

add this cycle to G'. Then, if (u, v) is an edge in G, add an edge between any vertex in u's cycle, and any vertex in v's cycle. It is easily seen that this graph G' is 3-regular. Moreover, s, t are reachable in G if and only if they are reachable in G' (where s and t are identified with any vertex on their corresponding cycle in G'). Clearly, the size of the graph G' is polynomial in the size of G. Now, we can apply the above algorithm to G', and the claim follows. ■

4.9. **Interactive Proofs.** We will begin with a simple definition.

Definition 4.4. A language $L \in$ **IP** if the following are true.

$$x \in L \implies \exists \text{ prover } P \text{ such that } \mathbb{P}[\text{verifier } V \text{ accepts } x] \geq 2/3$$
$$x \notin L \implies \forall \text{ provers } P, \mathbb{P}[\text{verifier } V \text{ accepts } x] \leq 1/3$$

The first condition is called *completeness* and the second condition is called *soundness*. The class **dIP** is similarly defined, except in that case the verifier V is deterministic. For more information, see the section 8.2 in [1].

Proposition 4.9. **IP** $\subseteq$ **PSPACE**.

Proof. To be completed. ■

Definition 4.5. Define the language #SAT$_E$ as follows.

$$\#\text{SAT}_E := \{\langle \phi, k \rangle \mid \phi \text{ is a boolean formula having exactly } k \text{ satisfying assignments}\}$$

Theorem 4.10. #SAT$_E \in$ **IP**.

Proof. For simplicity, let ϕ be a boolean formula in 3CNF. Suppose the number of variables in ϕ is n. Then, observe that

$$\phi(b_1, ..., b_n) = 1 \iff \{b_1, ..., b_n\} \text{ is a satisfying assignment of } \phi$$

So, it follows that

$$\sum_{b_1 \in \{0,1\}} \cdots \sum_{b_n \in \{0,1\}} \phi(b_1, ..., b_n) = k$$

where k is the number of satisfying assignments of ϕ.

Next, suppose C is a clause of the form $t_i \vee t_j \vee t_k$, where t_i, t_j, t_k are literals. Let variables be denoted by $x_1, ..., x_n$. An example of C might be

$$C = x_i \vee \overline{x_j} \vee x_k$$

For this C, define a polynomial P_C as follows.

$$P_C(x_i, x_j, x_k) = 1 - (1 - x_i)x_j(1 - x_k)$$

Similarly, for any clause C, we can define a polynomial P_C similarly. Then note that C agrees with the polynomial P_C on $\{0, 1\}$ values.

So, if ϕ is a formula in 3CNF given by

$$\phi = C_1 \wedge C_2 \wedge \cdots C_m$$

then define

$$P_\phi(x_1, ..., x_n) = P_{C_1} \cdot P_{C_2} \cdots P_{C_m}$$

Then, observe that

$$\sum_{b_1,...,b_n \in \{0,1\}} \phi(b_1, ..., b_k) = k \iff \sum_{b_1,...,b_n \in \{0,1\}} P_\phi(x_1, ..., x_n) = k$$

So, we have converted our problem about satisfiability of a boolean expression to an algebraic problem.

Now, choose some n such that $k \leq 2^n$ (recall that k is the number of satisfying assignments of ϕ). Now, choose a prime number p (sufficiently large) such that

$$2^n < p \leq 2^{2n}$$

It is clear that p can be represented with $2n$ bits in binary. Now, look at the polynomial P_ϕ as an element of $\mathbb{F}_p[x_1, ..., x_n]$. Then,

$$\sum_{b_1,...,b_n \in \{0,1\}} P_\phi(b_1, ..., b_n) = k \iff \sum_{b_1,...,b_n \in \{0,1\}} P_\phi(b_1, ..., b_n) = k \pmod p$$

and this is because of our choice of p. So, our problem of determining whether ϕ has exactly k satisfying assignments has turned to a problem of checking whether the right hand sum above is equal to k modulo p, which is an algebraic problem.

We now return to the main proof. We will describe the so called *sumcheck protocol*. Couldn't complete this. The sumcheck protocol is discussed on this link. ■

Theorem 4.11. PSPACE $\subseteq$ IP.

Proof. Since we already know that TQBF is **PSPACE**-complete, it suffices to show that TQBF $\in$ **IP**. So, suppose we have a fully quantified boolean formula Ψ. For simplicity, we assume that Ψ is of the form

$$\Psi = \exists x_1 \forall x_2 \exists x_2 \cdots x_n \phi(x_1, ..., x_n)$$

Then, the idea is as follows.

(1) Covert ϕ to a polynomial P_ϕ as in the previous theorem.
(2) Ψ is true if and only if

$$\sum_{x_1 \in \{0,1\}} \prod_{x_2 \in \{0,1\}} \sum_{x_3 \in \{0,1\}} \cdots P_\phi(x_1, ..., x_n) = K \neq 0$$

where above, we use a summation Σ for the $\exists$ quantifier, and a product $\prod$ for the $\forall$ quantifier.

Then essentially the same idea as in the previous theorem works, with a little modification. We need to handle the *degree blowup* as well, which results because of the multiplication above. This is well explained in section **8.5.3** of [1]. ■

4.10. **Public coins and the class AM.** In this section, we will define the *Arthur-Merlin* complexity class, which is denoted by **AM**.

Definition 4.6. The class **AM** is the class of languages that can be decided by a two round interactive proof which works as follows: on input x, the verifier V first sends a message y consisting of random bits to the prover. Then, the prover sends a certificate z back to the verifier. Then, the verifier runs a polynomial time deterministic machine $M(x, y, z)$ to decide the membership of x. Moreover,

$$x \in L \iff \mathbb{P}_y[\exists z \mid M(x, y, z) = 1] \geq 2/3$$
$$x \notin L \iff \mathbb{P}_y[\forall z \mid M(x, y, z) = 1] \leq 1/3$$

Remark 4.11.1. This protocol is also classified as being *public*, because the verifier's random bits are known to the prover.

Theorem 4.12. $\overline{\text{GISO}} \in$ **AM**, *where* GISO *is the graph isomorphism problem.*

We now introduce the so called *set lower bound protocol*. Suppose we have a set S whose description is known to us. Also, we assume that any $x \in S$ has a *short* membership proof, where *short* means a polynomial sized certificate for a proof. Let K be an integer such that

$$2^{k-2} \leq K \leq 2^{k-1}$$

In this protocol, the prover proves to the verifier that S has cardinality atleast K upto accuracy factor of 2, i.e

(1) If $|S| \geq K$, then the prover should have a strategy to convince the verifier with high probability to accept.
(2) If $|S| \leq K/2$, then verifier should reject with high probability.

4.11. **Pairwise Independent Hash Family.** Let $\mathcal{H}_{m,k}$ be a family of functions $\{0,1\}^m \to \{0,1\}^k$ with the following properties.

(1) For any $y \in \{0,1\}^k$ and $x \in \{0,1\}^m$,

$$\mathbb{P}_{h \in \mathcal{H}_{m,k}}[h(x) = y] = \frac{1}{2^k}$$

(2) For any $y_1, y_2 \in \{0,1\}^k$ and $x_1 \neq x_2 \in \{0,1\}^m$,

$$\mathbb{P}_{h \in \mathcal{H}_{m,k}}[h(x_1) = y_1 \wedge h(x_2) = y_2] = \mathbb{P}_{h \in \mathcal{H}_{m,k}}[h(x_1) = y_1] \cdot \mathbb{P}_{h \in \mathcal{H}_{m,k}}[h(x_2) = y_2]$$

Such a family is called a *pairwise independent hash family*. We will now see how to construct such a family.

First, let us construct $\mathcal{H}_{m,m}$, i.e here $k = m$. Consider the finite field $\mathbb{F}_{2^m}$. Then, consider the family $\mathcal{H}_{m,m}$ of all maps $h_{a,b} : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ defined by

$$h_{a,b}(x) = ax + b$$

where $a, b \in \mathbb{F}_{2^m}$. Clearly, we see that $|\mathcal{H}_{m,m}| = 2^m \cdot 2^m = 2^{2m}$

Proposition 4.13. *$\mathcal{H}_{m,m}$ as defined above is a pairwise independent hash family.*

Proof. Pick any $y_1, y_2 \in \{0,1\}^m$ and pick $x_1 \neq x_2 \in \{0,1\}^m$. Now,

$$\begin{aligned}
\mathbb{P}_{h \in \mathcal{H}_{m,m}}[h(x_1) = y_1 \wedge h(x_2) = y_2] &= \mathbb{P}_{a,b \in \mathbb{F}_{2^m}}[ax_1 + b = y_1 \wedge ax_2 + b = y_2] \\
&= \mathbb{P}_{a,b \in \mathbb{F}_{2^m}}\left[\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right] \\
&= \mathbb{P}_{a,b}\left[\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right] \\
&= \frac{1}{2^m} \cdot \frac{1}{2^m} \\
&= \frac{1}{2^{2m}}
\end{aligned}$$

because x_1, x_2, y_1, y_2 are fixed in the above chain of equations. ∎

Exercise 4.1. If $k < m$, show that a simple way to construct a pairwise independent hash family $\mathcal{H}_{m,k}$ is to ignore the last $m - k$ bits of the functions in $\mathcal{H}_{m,m}$ constructed above.

4.12. **GS Set Lowerbound Protocol.** The *Goldwasser-Sipser Set Lowerbound Protocol* works as follows.

(1) $S \subseteq \{0,1\}^m$ is a set such that membership of S can be efficiently proven. The prover and the verifier both know a number K. The prover wants to convince the verifier that $|S| \geq K$ and the verifier should reject if $|S| \leq \frac{K}{2}$. k is a number such that $2^{k-2} \leq K \leq 2^{k-1}$.
(2) The verifier randomly picks a function $h \in \mathcal{H}_{m,k}$, where $\mathcal{H}_{m,k}$ is a pairwise independent hash family. The verifier also picks some $y \in \{0,1\}^k$ randomly. Then, the verifier sends h, y to the prover.
(3) The prover produces $x \in S$ such that $h(x) = y$ and sends a certificate of membership of x in S.
(4) The verifier checks whether $h(x) = y$ and also checks the certificate for membership of x. If yes, then verifier accepts, otherwise rejects.

Now consider the following.

(1) Suppose $|S| \leq K/2$ and that $|S| \leq 2^{k-1}$. For $x \in S$, define the event $E_x := h(x) = y$. Then,

$$\begin{aligned}
\mathbb{P}\left(\bigcup_{x \in S} E_x\right) &\leq \sum_{x \in S} \mathbb{P}[E_x] \\
&= \sum_{x \in S} \mathbb{P}_{h \in \mathcal{H}_{m,k}}[h(x) = y] \\
&= \sum_{x \in S} \frac{1}{2^k} \\
&\leq \frac{|S|}{2^k} \\
&\leq \frac{K}{2^{k+1}}
\end{aligned}$$

(2) Next, suppose $|S| \geq K$. Then, we see that

$$\begin{aligned}
\mathbb{P}\left(\bigcup_{x\in S} E_x\right) &\geq \sum_{x\in S}\mathbb{P}[E_x] - \sum_{x_1\neq x_2\in S}\mathbb{P}[E_{x_1}\cap E_{x_2}] \\
&= \frac{|S|}{2^k} - \sum_{x_1\neq x_2\in S}\mathbb{P}[h(x_1)=y\wedge h(x_2)=y] \\
&= \frac{|S|}{2^k} - \sum_{x_1\neq x_2\in S}\mathbb{P}_h[h(x_1)=y]\mathbb{P}_h[h(x_2)=y] \\
&= \frac{|S|}{2^k} - \binom{|S|}{2}\frac{1}{2^{2k}} \\
&\geq \frac{|S|}{2^k} - \frac{|S|^2}{2}\cdot\frac{1}{2^{2k}} \\
&= \frac{|S|}{2^k}\left(1-\frac{|S|}{2^{k+1}}\right) \\
&\geq \frac{3}{4}\cdot\frac{|S|}{2^k} \\
&= \frac{3}{2}\cdot\frac{|S|}{2^{k+1}} \\
&\geq \frac{3}{2}\frac{|K|}{2^{k+1}}
\end{aligned}$$

4.13. **Graph Non-Isomorphism is in AM.** Consider the $\overline{GISO}$ problem.

Theorem 4.14. $\overline{\mathsf{GISO}} \in$ **AM**.

Proof. We will use the Goldwasser-Sipser Set Lowerbound Protocol here. The idea is to try to construct a set S and some number K such that if $G_1 \ncong G_2$ then $|S| \geq K$ and if $G_1 \cong G_2$ then $|S| \leq K/2$, and such that membership of S can be proven via an efficient polynomial-sized certificate.

So, suppose G_1, G_2 are input graphs given to us. Define

$$S := \{(H,\pi) \mid H \cong G_1 \text{ or } H \cong G_2, \quad \pi \in \mathsf{Aut}(H)\}$$

Notice that the certificate for proving membership in S is polynomial sized (because we only need to exhibit bijections).

Let $K = n!$. We claim that if $G_1 \ncong G_2$ then $|S| = 2n!$ and if $G_1 \cong G_2$, then $|S| = n!$. For a graph G, define

$$\mathsf{Iso}(G) := \{\pi(G) \mid \pi \in S_n\}$$

We know that $\mathsf{Aut}(G) \leq S_n$. So, $\mathsf{Aut}(G)$ partitions S_n into cosets. Let $\{\rho_1, ..., \rho_l\}$ be the coset representatives. Now, we see that $l = |\mathsf{Iso}(G)|$, and hence

$$|\mathsf{Iso}(G)| \cdot |\mathsf{Aut}(G)| = n!$$

If $G_1 \cong G_2$, then clearly the cardinality of the set S will be $n!$ by the above identity. If $G_1 \ncong G_2$, then again by the above identity we see that $|S| = 2n!$. Then, we just apply the GS protocol as usual. ∎

4.14. **Permanent of a Matrix.** First, let us begin with a simple definition. For an $n \times n$ matrix A, we define

$$\mathsf{Perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

and this quantity is called the *permament* of A. Note the similarity with the determinant.

Just like the determinant, the permanent can be expanded along a row, say the first row.

$$\mathsf{Perm}(A) = \sum_{i=1}^{n} a_{1i}[\mathsf{Perm}(A_{1i})]$$

where for each i, A_{1i} is the matrix obtained after deleting the first row and the i^{th} column.

Now suppose we have a randomized algorithm $\mathscr{A}$ which takes as input an $n \times n$ matrix A over a field $\mathbb{F}$ and correctly computes the permanent of the input matrix on $1 - 1/3n$ fraction of the input. Then, we claim that there exists another algorithm $\tilde{\mathscr{A}}$ which can compute the permanent of the input matrix with high probability *for all* inputs A. Let us see how the algorithm $\tilde{\mathscr{A}}$ works.

(1) Let the input matrix be A. $\tilde{\mathscr{A}}$ samples a random matrix $R \in \mathbb{F}^{n \times n}$, which is sampled uniformly at random.
(2) $\tilde{\mathscr{A}}$ then constructs $B = A + xR$, where x is a variable. Let

$$\mathsf{Perm}(B) = \mathsf{Perm}(A + xR) = g(x)$$

where $g(x)$ is a polynomial. Note that $\deg(g(x)) \leq n$.
(3) If we fix $x = a$, then note that $B(a) = A + aR$ is a random matrix, because A, a are fixed, and R is random. Note that $g(0) = \mathsf{Perm}(A)$.
(4) Fix any $n + 1$ distinct elements $a_1, a_2, ..., a_{n+1} \in \mathbb{F}$ of the field. Consider the matrices $B(a_1), ..., B(a_{n+1})$. Since R is a random matrix, all these matrices are also random. Now, we know that the algorithm $\mathcal{A}$ makes a mistake in computing the permanent of a random matrix with probability less than $\dfrac{1}{3n}$. By a simple union bound, $\mathcal{A}$ makes a mistake on any of $B(a_1), ..., B(a_{n+1})$ with probability less than $\dfrac{n+1}{3n} \sim \dfrac{1}{3}$. This means that with probability $\geq \dfrac{2}{3}$, we get the correct permanent for $B(a_1), ..., B(a_{n+1})$ by using $\mathcal{A}$. Note that

$$g(a_1) = \mathsf{Perm}(B(a_1)), ..., g(a_{n+1}) = \mathsf{Perm}(B(a_n))$$

Now suppose $g(x) = g_0 + g_1 x + ... + g_n x^n$. The above equations give us the following system of linear equations.

$$\begin{bmatrix} 1 & a_1 & a_1^2 & \cdots & a_1^n \\ 1 & a_2 & a_2^2 & \cdots & a_2^n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & a_{n+1} & a_{n+1}^2 & \cdots & a_{n+1}^n \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} \mathsf{Perm}(B(a_1)) \\ \mathsf{Perm}(B(a_2)) \\ \vdots \\ \mathsf{Perm}(B(a_{n+1})) \end{bmatrix}$$

The first matrix on the left hand side above is an example of a *Vandermonde Matrix*. Since all the a_i's are chosen to be distinct, this matrix

is invertible. So, we can recover the values $g_0, ..., g_n$, and hence we can recover $g(0)$, which is what we wanted to do.

So, with probability $\geq 2/3$, the algorithm $\tilde{A}$ computes Perm(A) correctly for all matrices A.

4.15. **PCPs.** Was unable to make notes on this, but can be found in section 18.1 of [1].

References

[1] Sanjeev Arora, Boaz Barak, Computational Complexity: A Modern Approach, Draft of a book: Dated January 2007.
[2] Michael Sipser, Introduction to the Theory of Computation, Thomson South-Western (2012)