

CLASS NOTES

These are notes that I'm taking for Stanford's CS 254, taught by Professor Li-Yang Tan. I took CS 254 in Winter 2022, but I'm taking these notes over summer 2022, as a review of the course. Because of this, I'm basing the notes on the live notes taken by Milan Mossé in 2020, supplemented with the handwritten notes files from class lectures. Also, thanks to Kai for editing these notes and Meghna for helping me clarify the proofs in Lectures 15 and 16 ☺

TABLE OF CONTENTS

Lecture 1: Introduction	page 2
Lecture 2: CS 154 Review	page 3
Lecture 3: Space Complexity, I	page 6
Lecture 4: Space Complexity, II	page 9
Lecture 5: Space Complexity, III	page 12
Lecture 6: The Class coNP	page 14
Lecture 7: The Polynomial Hierarchy	page 16
Lecture 8: A Black-Box SAT-solver	page 20
Lecture 9: Intro To Randomness	page 23
Lecture 10: Randomized Complexity Classes	page 25
Lecture 11: Circuits	page 27
Lecture 12: More on P/poly	page 29
Lecture 13: Interactive Proofs	page 31
Lecture 14: Proving $IP = PSPACE$	page 33
Lecture 15: Merlin-Arthur and Arthur-Merlin	page 35
Lecture 16: Randomness and PH	page 38

LECTURE 1: INTRODUCTION

What is complexity theory? In the most general sense, we are trying to answer the question:

What is the most efficient way to solve a given computational task?

By **efficient**, we mean: using as little of a resource as possible.

We are used to discussing time-efficiency; in this class, we will also discuss efficiency with respect to space/memory, or random bits, or communication.

In this class, we will focus on **lower bounds**. That is, we will not necessarily find ways to solve our task, but instead focus on proving statements of the form “no algorithm for solving this problem can use fewer than this many resources.”

Complexity theory has its roots in computability theory, which was more of the focus of CS 154. Computability theory is focused on answering the question:

Which computational tasks can be solved at all?

In 1936, Turing told us that not all computational tasks can be solved (remember the Halting Problem from 154). This is one of the cool things about complexity theory: it’s introspective, in the sense that we are very aware of our own limitations.

There are many open questions in complexity theory, here are a few:

- $P \supseteq NP$? If the solution to a computational task can be verified quickly, can it also be found quickly? We guess the answer is **no**.
- $NP = coNP$? If a theorem is simple to state, does that mean it will also have a short proof? We guess the answer is **no**.
- $P \subseteq NC$? Is every efficient algorithm efficiently parallelizable? (here, we mean efficient with respect to time) We guess the answer is **no**.
- $P \subseteq L$? If we have an algorithm to quickly solve a problem, can we turn it into an algorithm to solve it using almost no memory? We guess the answer is **no**.
- $P \supseteq BPP$? Can every fast randomized algorithm be made deterministic? We guess the answer is **yes** (specifically, every fast randomized algorithm can be made deterministic with only a *polynomial* time increase)

Remember that the [Time Hierarchy Theorem](#) tells us that with a logarithmic increase in time, we can solve strictly more problems; so for example P is a strict superset of the problems solvable in $O(n \log n)$ time.

LECTURE 2: CS 154 REVIEW

Our main question last lecture referred to *computational tasks*. There are multiple kinds of computational tasks, here are two:

Definition 1. Decision problems are functions that take in a string and output **accept** or **reject** (we can also think of them as outputting 0 or 1, or **yes** or **no**...)

Example 2. One of the classic decision problems we will discuss is 3SAT, where we take in a 3CNF formula, and output **yes** if it is decidable, and **no** otherwise.

We usually think of decision problems in terms of *languages*, where a language L is the set of strings we want to accept for a given decision problem.

Definition 3. Function problems are functions that take in a string and output a string.

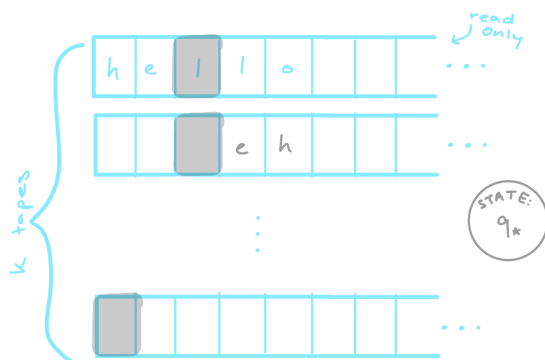
(What we mean by this is we have, in our head, a desired output string for every input string. Because we're mapping each input to an output, we can think of this as a function f . Then, we want our algorithm to receive an input x and print $f(x)$.)

Example 4. A common function problem is the problem of outputting the number of satisfying assignments of a given CNF formula.

We will focus on decision problems in this class.

Remember that the **Church-Turing thesis** says that every real-world algorithm can be simulated by a Turing machine, and this can be done with at most polynomial slow-down.

We will focus on **multitape Turing machines**:



Here, we have k tapes, where k is some constant, and the first one is a read-only input tape. There are also some set of states Q , and a transition function that at each timestep:

- decides what to write on the current space on the $k - 1$ worktapes
- moves each tapehead one spot in either direction (or keeps it at the same spot)
- changes the current state

It does so based on only the current state and the character it can currently read on each of the k tapes.

Definition 5. We say that the **time complexity** of a Turing machine is a function $t(n) : \mathbb{N} \rightarrow \mathbb{N}$, where $t(n)$ is our Turing machine's worst-case runtime on an input of length n .

Definition 6. We define the **time complexity class** $\text{TIME}(t(n))$ to be the set of all languages L such that some Turing machine decides L in $O(t(n))$.

One of our favorite time complexity classes is P , which is the set of all languages that can be decided in polynomial time, or

$$\bigcup_{c \in \mathbb{N}} \text{TIME}(n^c).$$

We also have the time complexity class EXPTIME , which is the set of languages decidable in exponential time, or

$$\bigcup_{c \in \mathbb{N}} \text{TIME}(2^{n^c}).$$

In general, we like it when we can show that things are in P and not just in EXPTIME .

The time hierarchy theorem tells us that we can solve strictly more problems with more time. Specifically:

Theorem 7 (Time Hierarchy Theorem). For any function $t(n) : \mathbb{N} \rightarrow \mathbb{N}$,

$$\text{TIME}(t(n)) \subsetneq \text{TIME}(t(n) \log(t(n))).$$

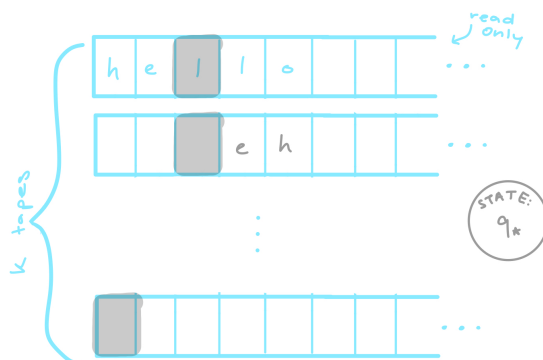
This means that linear time is strictly less powerful than quadratic time, which is strictly less powerful than cubic time, and so on. Also, it shows that $P \subsetneq \text{EXPTIME}$.

The proof of this (which we covered in 154) is based on the idea of *diagonalization*.

This is a cool theorem, because finding well-defined lower bounds like this is very rare. For example, the **relativization barrier**, which was proved in 1975, showed that ideas related to diagonalization cannot be used to separate P from NP .

Now, we will discuss nondeterminism, and define the class NP .

A nondeterministic Turing machine has almost the same structure as before:



but it has two transition functions instead of one, and nondeterministically acts according to both. So we can think of each sequence of actions as a *branch* of the NTM on the input, and if any branch leads to

acceptance, our NTM accepts.

Then, nondeterministic time complexity acts the same way as the deterministic version, except we take the worst-case time complexity across all branches.

Definition 8. Using this, we can define the class NP to be the set of all languages decidable in polynomial time by a nondeterministic Turing machine, or

$$\text{NP} = \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c).$$

Remark 9. If we have a polynomial-time deterministic Turing machine that decides a language L , it is easy to turn it into a polytime TM that decides $\neg L$; just swap the accept and reject states. So the classes P and coP are equivalent.

However, we haven't found a general way to turn a polytime NTM that decides L into a polytime NTM that decides $\neg L$ (since in this case we want to turn a mix of accept and reject branches into entirely reject branches). As we mentioned earlier, it is believed that $\text{NP} \neq \text{coNP}$, though this is still an open problem.

Some people think that $\text{NP} \cap \text{coNP} = \text{P}$, but if that were true many cryptosystems would be broken, and Li-Yang thinks it is false.

If you think this definition of NP is unintuitive, you may prefer the verifier definition:

Definition 10. A language L is NP if we have a verifier V , which is a deterministic Turing machine with the following properties:

- V runs in polynomial time in terms of the input x , but takes in two inputs, which we will call x and w
- if $x \in L$, there exists some string w such that $V(x, w)$ accepts
- if $x \notin L$, then for all strings w , $V(x, w)$ rejects

Essentially, each input provides a certificate to “prove” that it is in L , and the verifier is able to check this certificate in polynomial time. Every valid input is able to produce some valid certificate, and no invalid input is able to trick the verifier.

The final CS154 theorem we will discuss explains why we care about the SAT problem so much:

Theorem 11 (Cook-Levin). SAT is an NP-hard problem. That is, for any language L in NP , we can find a polynomial-time reduction from L to SAT.

Essentially, the proof of the Cook-Levin theorem uses the verifier definition of NP . We say that we can, in polynomial time, simulate the verifier $V(x, \cdot)$ with a CNF formula φ . Then, if there is a satisfying assignment to φ , this corresponds to a valid certificate w for x . The majority of the proof, which we will not discuss here, is about the construction of this CNF formula, based on our verifier and the given input.

Then, (since SAT and 3SAT reduce to each other) we can say that 3SAT is “representative” of NP in a sense. That is, since 3SAT is in NP , we have that:

Corollary 12. The class $\text{P} = \text{NP}$ if and only if $3\text{SAT} \in \text{P}$.

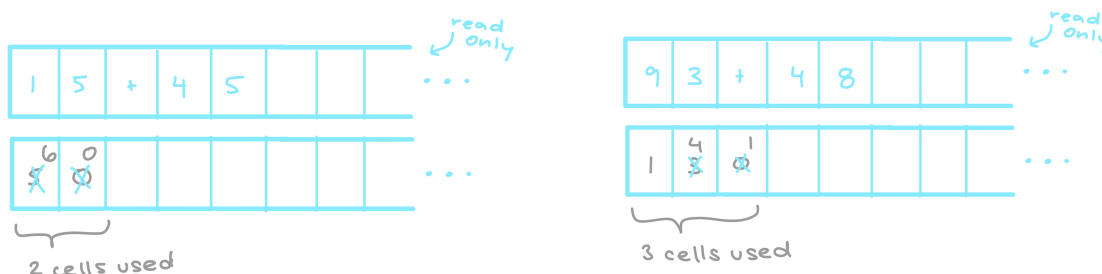
LECTURE 3: SPACE COMPLEXITY, I

For most of 154, we talked about problems related to time complexity. Many of these problems have natural analogues with respect to space complexity, and in general the space-complexity versions are easier to answer.

Intuitively, space is nicer to play with than time, because space, unlike time, can be reused.

Space complexity is defined as we expect:

Definition 13. The **space complexity** of a Turing machine is a function $s : \mathbb{N} \rightarrow \mathbb{N}$ which is defined as $s(n)$ is the worst-case number of worktape cells used by M over all inputs of length n .



We also define the space-complexity classes as we would expect:

Definition 14. The class $\text{SPACE}(s(n))$ is the set of all languages L such that we can decide L in space $O(s(n))$.

The reason we don't count the input tape in our space complexity is that many interesting problems only require *logarithmic* space in terms of the length of the input.

Exercise 15. We can construct a Turing machine that computes the length of the input (in binary) using $\log n$ space - can you see how?

Definition 16. We define LOGSPACE, or L, to be the set of all languages decidable in $\text{SPACE}(\log n)$.

A more surprising result is:

Theorem 17. Define the language STCONN to be the set of strings of the form $\langle G, s, t \rangle$ where G is a directed graph and there exists a path from s to t .

Then, STCONN is decidable in $\text{SPACE}(\log^2 n)$.

Note that we do not believe STCONN is in L, but Omer Reingold proved in 2005 that STCONN for undirected graphs is in L.

We will prove this theorem in the end of lecture, but first, let's look at a few more examples of space complexity classes.

Example 18. 3SAT is in $\text{SPACE}(n)$.

To see this, note that we can just test out all possible assignments; this should take exponential time but at most linear space, since we can reuse the space used to check an assignment.

Moreover, in general our space complexity classes capture more than our time complexity classes:

Proposition 19. For any function $t(n)$, $\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$.

For deterministic Turing machines, this follows from the fact that you can write in at most $t(n)$ cells using $t(n)$ time.

Moreover, since for any nondeterministic Turing machine we can use the above strategy of cycling through all possible certificates, so for example $\text{NP} \subseteq \text{PSPACE}$, where, as we might expect, PSPACE is $\bigcup_c \text{SPACE}(n^c)$, or the set of all languages that can be decided in polynomial space.

We also have inclusions in the other direction:

Theorem 20. When $s(n) \geq \log n$, $\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$.

Proof. We consider a *configuration* of a Turing machine to be its entire description at a given timestep. That is, its configuration consists of all worktapes, all tapehead positions, and the current state, for that timestep. Then, we can see that if our Turing machine halts on a given input, it can never reach the same configuration more than once.

Then, consider any language $L \in \text{SPACE}(s(n))$. We want to show that $L \in \text{TIME}(2^{O(s(n))})$. Specifically, if our Turing machine M decides L using $s(n)$ space, then we will show that on any input x , it uses no more than $2^{O(s(n))}$ time. We will do so by counting the total number of possible configurations for its run on x . We can see that there are $2^{(k-1) \cdot s(n)}$ possible things that could be written on the worktapes. We multiply this by the $k \cdot s(n)$ possible tapehead positions, and some constant number of possible states, to get a total of $2^{O(s(n))}$ possible configurations. Thus, M must run on x in at most $2^{O(s(n))}$ time, and generalizing, we get that $\text{SPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$. \square

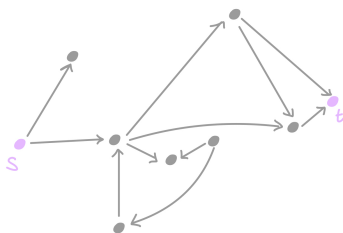
Combining these, we get the following chain:

$$\text{L} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}.$$

There is a space equivalent of the Time Hierarchy Theorem, which tells us that L is a *strict* subset of PSPACE . This means at least one of the inclusions above must be strict, but showing that any one of them is strict is still an open problem.

We will now return to proving [Theorem 17](#).

Proof. To prove this, we want an algorithm to look for a path between two points on a graph:



while using $O(\log^2 n)$ space.

We will use what we call *middle-first* search. That is, we define a subroutine **path**, which we will use to recursively look for midpoints in the path from s to t .

```
def path(x, y, k):
    if x == y:
        return True
    if k == 0:
        if edge(x, y):
            return True
        else:
            return False

    for w in G: # check if w is a midpoint
        if path(x, w, k-1) and path(w, y, k-1):
            return True

    # we reach this point if no w was a midpoint
    return False
```

Essentially, we are checking if there is a path of length at most 2^k from x to y by checking if any w is a midpoint of this path, by recursively checking if there is a path of length at most 2^{k-1} from x to w and from w to y .

Then, to solve STCONN, we just call $\text{path}(s, t, \log n)$, where n is the number of nodes in the graph. Can you see why this algorithm works?

To analyze its space usage, let us think about what information about the graph we need to store. Since s and t are given on the input tape, we just need to track the current value of k (in the smallest recursive layer), and we need to keep the current value of w for *each* open recursive layer. Since there are at most $k = \log n$ layers, and each value of w takes $\log n$ bits to write, we get a total of $O(\log^2 n)$ space, as we desired! \square

However, note that this algorithm uses a lot of time; since there are at most $2n$ recursive calls at each layer, we get a total of $(2n)^k = O(n^{\log n})$ time. It is an open question whether there is an algorithm for STCONN that uses polylog space and polynomial time.

LECTURE 4: SPACE COMPLEXITY, II

In this lecture, we will discuss nondeterministic space, and NL, which we can think of the space equivalent of NP. Unlike in NP, however, we will show that $NL = coNL$. This was extremely surprising when the proof was published!

A natural question is: is $NL = L$? This is still an open problem, but we have a decent bound on NL in terms of deterministic space, which we will get to soon.

During this lecture and the next, we will prove the following three theorems, which help us place NL in relation to all our familiar complexity classes.

Theorem 21. $NL \subseteq P$

Theorem 22. $NL \subseteq SPACE(\log^2 n)$

Theorem 23. $NL = coNL$

To start proving this, we should first make sure we have a solid idea of what NL is.

Definition 24. We say that a nondeterministic Turing machine has **space complexity** $s(n) : \mathbb{N} \rightarrow \mathbb{N}$ if, for all inputs x of length n , our Turing machine uses at most $s(n)$ space in *every* branch when deciding x .

Then, we define $NSPACE(s(n))$ to be the set of all languages that can be decided by a nondeterministic Turing machine in $O(s(n))$ space, and NL to be nondeterministic logspace, or $NSPACE(\log n)$.

To gain some intuition about NL, we return to a familiar example:

Proposition 25. $STCONN \in NL$

Proof. Since we are working nondeterministically, we can simply guess paths from s to t until we find one that works:

```
def check_connected(G, s, t):
    if s == t:
        return True

    steps = 0
    current_node = s
    while steps <= n:
        nondeterministically choose a neighbor w of current_node
        current_node = w

        if w == t: # we have completed our path to t
            return True
        steps ++

    # if the path we chose did not work
    return False
```

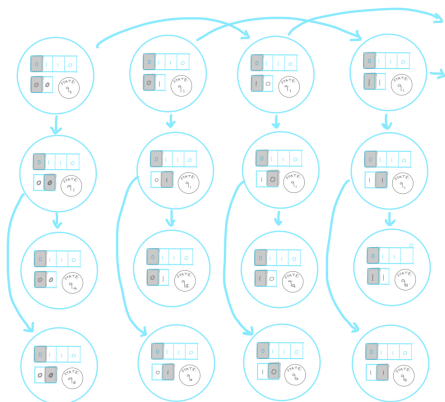
Here, we nondeterministically check every possible path from s to t . But since we only need to keep track of our current node and the steps taken so far, which uses $\log n$ space, so $\text{STCONN} \in \text{NL}$, as desired. \square

In a sense, STCONN is actually NL -complete; we will reductions from elements in NL to STCONN to show both [Theorem 21](#) and [Theorem 22](#).

Proof that $\text{NL} \subseteq \text{P}$. We want to pick an arbitrary language in NL and show that it is also in P .

Specifically, we will pick a language $L \in \text{NL}$. We know that there is some nondeterministic Turing machine N that decides L . For any input x , we will, in polynomial time: construct a graph based on $N(x)$ and use our algorithm from [Proposition 25](#) to check whether there is a valid path to an accept state in the running of $N(x)$.

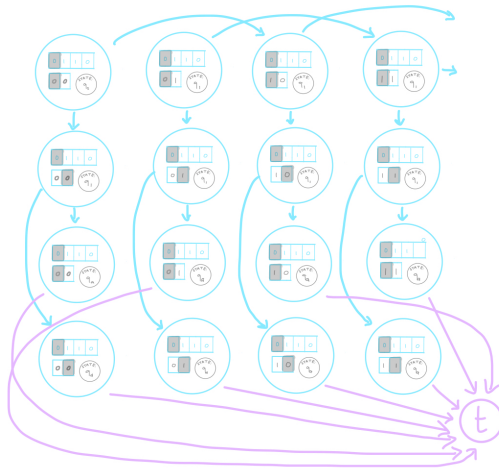
To do so, recall our proof of [Theorem 20](#). We showed that for a machine N that runs in $O(\log n)$ space on x , there are at most $\text{poly}(n)$ configurations it can reach, over all branches. We can draw a graph of these $\text{poly}(n)$ configurations, where each configuration (or node) has out-degree 2 corresponding to the configuration of the Turing machine after applying either of the two transition functions.



part of a configuration graph

Since there are $\text{poly}(n)$ configurations, and we just need to check the two transition functions for each one, constructing this graph takes $\text{poly}(n)$ time.

There is exactly one configuration that corresponds to the start state of $N(x)$, which we will call s . Then, we will draw an extra node t and connect all configurations corresponding to accept states to t to get our final graph.



Then, $N(x)$ accepts if and only if there is a path from s to t on this graph, so we have reduced to STCONN. But we can just use an algorithm like breadth-first search to decide STCONN in polynomial time, so we are done. \square

We might want to just use the same construction, and the fact that $\text{STCONN} \in \text{SPACE}(\log^2 n)$ from last time, to show that $\text{NL} \subseteq \text{SPACE}(\log^2 n)$. But we can't exactly do that; our configuration graph is much larger than $\log^2 n$, so we cannot store the entire graph in memory. Instead, we will generate pieces of the configuration graph as needed:

Proof that $\text{NL} \subseteq \text{SPACE}(\log^2 n)$. Recall the path algorithm from the proof of [Theorem 17](#).

For any language $L \in \text{NL}$ and nondeterministic Turing machine N deciding L , we can use the following modification of `path`:

```
def check_acceptance(x):
    let s be the start configuration of N(x)
    let n be the number of configurations of N(x)
    return pathtoaccept(s, log n)

def pathtoaccept(x, k):
    if x is at an accept state:
        return True

    if k == 0:
        return False

    for all configurations w of N(x):
        if pathtonode(x, w, k-1) and pathtoaccept(w, k-1):
            return True

    # no w was a midpoint
    return False

def pathtonode(x, y, k):
    if x == y:
        return True

    if k == 0:
        if either transition function maps x to y:
            return True
        else:
            return False

    for all configurations w of N(x):
        if pathtonode(x, w, k-1) and pathtonode(w, k-1):
            return True

    # no w was a midpoint
    return False
```

The code logic here is the same as that of `path`, so similar algorithm analysis applies, except we generate nodes and edges of our graph only as they become relevant, so we never use too much space. \square

LECTURE 5: SPACE COMPLEXITY, III

In this lecture, we will prove [Theorem 23](#), which says that $NL = \text{coNL}$. Specifically, we will show that $\text{STCONN} \in \text{coNL}$ (or, $\overline{\text{STCONN}} \in \text{NL}$), and since we showed last lecture that STCONN is NL -complete, this implies that $NL \subseteq \text{coNL}$, so $NL = \text{coNL}$.

This proof earned its authors the 1995 Gödel prize - it was very surprising when it was released!

We will begin by describing a verifier version of NL , in the way there is a verifier version of NP .

Definition 26. A language L is NL if we have a verifier V , which is a deterministic Turing machine with the following properties:

- V runs in logarithmic space in terms of the input x , but takes in two inputs, x and w , where the length of w is polynomial in the length of x
- x is on the regular read-only input tape, and w is on a special input tape that can only be read once, from left to right
- if $x \in L$, there exists some string w such that $V(x, w)$ accepts
- if $x \notin L$, then for all strings w , $V(x, w)$ rejects

We will leave it as an exercise to show that this is equivalent to the standard definition of NL .

Using this, we can show that $\text{STCONN} \in \text{coNL}$:

Theorem 27. The language $\overline{\text{STCONN}}$, which is the set of directed graphs $\langle G, s, t \rangle$ such that there is *no* path from s to t , is in NL .

Proof. Using the certificate definition of STCONN , we will come up with a certificate that can prove there is no path from s to t .

Then, we will begin with some notation. Let n be the number of vertices in our graph, and let R_ℓ be the set of all vertices reachable from s in at most ℓ steps. Our goal is to convince the Turing machine that t is not in R_n . But our Turing machine doesn't have the space to write down R_ℓ at any point; instead, we will have it write down $r_\ell = |R_\ell|$, which has at most $\log n$ bits.

We will individually convince the Turing machine of each r_ℓ , in order.

We can see that once the Turing machine is convinced of r_n , then we can convince it that t is not in R_n by giving it a list of paths to each of the vertices in R_n . The verifier will read through the paths in order checking that each path is a valid path to a given vertex, keeping track of the number of paths seen so far (so that we get to r_n total paths), and making sure that none of the paths we see contain t . If this is the case, we have proved that t is not reachable in n steps.

(Note that this list of paths must be in some predetermined order, or else we could trick the verifier by including the path to the same vertex twice.)

So the question then is how to get from r_ℓ to $r_{\ell+1}$?

If the Turing machine is convinced of r_ℓ , we can convince it of $r_{\ell+1}$ by giving it a certificate for each of the n vertices, saying either "this vertex is in $R_{\ell+1}$ " or "this vertex is *not* in $R_{\ell+1}$." To prove the former, we just

need a valid path from s to vertex i in at most $\ell + 1$ steps, and the verifier can verify that it is a valid path and it does not take too many steps. To prove the latter, we will first remind the verifier of all r_ℓ vertices in R_ℓ , as we did above. Then, the verifier can check that these paths are valid, it was shown exactly r_ℓ vertices, and that, for each such vertex j , it is not equal to vertex i and there is no edge $j \rightarrow i$.

As a quick check, the verifier has written down at most r_ℓ , the current vertex being checked, a counter up to ℓ , and information about the previous step in the path. This takes some constant times $\log n$ space, so the verifier runs in $\log n$ space, as desired.

Moreover, the certificate contains information about each of the n vertices, for each ℓ . For each vertex, this information contains at most n paths, each of length $\ell - 1$, for a total length of $O(n^5)$. \square

LECTURE 6: THE CLASS coNP

First, let's make sure we know the intuitive meaning of some important open questions.

If $\text{NP} = \text{coNP}$, this means that if there exist short, efficiently checkable proofs of membership, there also exist such proofs of non-membership, and vice versa.

If $\text{P} = \text{NP} \cap \text{coNP}$, this means if there exist short, efficiently checkable proofs of membership and non-membership, there exists an efficient algorithm to decide membership.

Remember our definition of coNP :

Definition 28. A language L is in coNP if \overline{L} is in NP . Equivalently, for languages $L \in \text{coNP}$, we have a polynomial-time verifier V such that for all $x \in L$, $V(x, w)$ rejects for all polynomial-length certificates w .

Remember that $\text{P} = \text{coP}$ by switching accept and reject states, but it is an open problem whether $\text{NP} = \text{coNP}$.

Example 29. A good example of a language in coNP is UNSAT . Here, $\text{UNSAT} = \overline{\text{SAT}}$ is the set of boolean formulas φ such that there exist *no* satisfying assignments for φ .

Exercise 30. We can see that $\text{P} \subseteq \text{coNP}$, and therefore if $\text{P} = \text{NP}$ then $\text{NP} = \text{coNP}$. Can you prove this is true?

A popular conjecture is that $\text{coNP} \neq \text{NP}$; from the above exercise, this would also imply that $\text{P} \neq \text{NP}$.

The coNP -complete languages are quite nice; these are just the ones whose complement are NP -complete.

Proposition 31. A language L is NP -complete if and only if \overline{L} is coNP -complete.

Proof. We will just show one direction: if L is NP -complete, this means there is a polynomial-time reduction f from any language $L' \in \text{NP}$ to L . But this means that $x \in L'$ if and only if $f(x) \in L$; equivalently, $x \in \overline{L'}$ if and only if $f(x) \in \overline{L}$. But the languages of the form $\overline{L'}$ are all languages in coNP , so f is also a polynomial-time reduction from any language in coNP to \overline{L} , and \overline{L} is coNP -complete.

The other direction follows similarly. □

This means that UNSAT , for example, is coNP -complete. This implies the following theorem:

Theorem 32. $\text{NP} = \text{coNP}$ if and only if $\text{UNSAT} \in \text{NP}$.

We leave the proof as an exercise; it should follow from the previous proposition.

In general, problems in coNP seem like a “for all” problem (“for all certificates, this is false”), while problems in NP seem like an “exists” problem (“there exists a certificate for which this is true”).

As a summary of the introduction, a language L is in NP if for every $x \in L$, there exists a short, efficiently checkable certificate that $x \in L$ (but not necessarily an efficient way of finding this certificate). If $L \in \text{coNP}$, then for every $x \notin L$, there exists such a certificate.

Then, in a sense, $\text{NP} \cap \text{coNP}$ is the set of languages we really understand, because for every element, there exists a short, efficiently checkable proof of membership or non-membership. The idea is that if $\text{P} = \text{NP} \cap \text{coNP}$,

then the existence of these certificates would imply an efficient way of finding such certificates.

Let's look at some examples of such languages:

Example 33. Let's say we are given a bipartite graph G ; that is, we can divide the nodes of the graph into disjoint subsets U and V , such that all edges map a node in U to a node in V . Then, is there a bijection $f : U \rightarrow V$ such that for every $u \in U$, there is an edge $u \rightarrow f(u)$?

This is known as the perfect matching problem.

This problem is clearly in NP; the certificate is just the actual matching.

To show it's in coNP, we must show that if G does not have a perfect matching, there exists a certificate proving this. If there is a subset $S \subseteq U$ such that the neighborhood of S , $\Gamma(S) \subseteq V$, is smaller than S , then there cannot be a perfect matching. Then, [Hall's Theorem](#) tells us that if G does not have a perfect matching, we can find $S \subseteq U$ such that $|\Gamma(S)| < |S|$. So this subset S would serve as a certificate that G does not have a perfect matching.

In 1956, which was much later, Ford and Fulkerson came up with their [max-flow algorithm](#), which shows that this problem is in P.

Example 34. The language PRIMES contains a binary encoding of all prime integers. Since the encoding of a number n uses $\log n$ bits, our certificate must be polynomial in $\log n$, not n .

PRIMES \in coNP since a valid factor $1 < m < n$ of n is a valid certificate that n is not prime.

Moreover, [Vaughan Pratt \(1975\)](#) proved that PRIMES \in NP, by turning the “for all” statement (all smaller numbers don't divide n) into an “exists” statement (there exists some $1 < m < n$ such that $m^k \not\equiv 1 \pmod{n}$ for all $k < n - 1$ but $m^{n-1} \equiv 1 \pmod{n}$).

The [AKS primality test](#), discovered in 2002, showed that PRIMES \in P.

Example 35. The language FACTORING is the set of encodings $\langle x, a, b \rangle$ such that there exists some prime $p \in [a, b]$ that divides x .

This is clearly in NP, since the prime p is a valid certificate of membership (the verifier can use the AKS test to check that p is prime, and then just divide to check that $p|x$).

Moreover, this is in coNP, since the prime factorization of x gives a valid certificate of non-membership. The verifier just checks that these numbers multiply to x , all of them are primes, and none of them are in the interval $[a, b]$.

However, FACTORING is not believed to be in P; much of cryptography is based on this assumption. The reason they use FACTORING, instead of a problem more likely to be not in P (such as SAT) is that NP-complete problems are difficult to use for cryptosystems. There are many formal reasons for why this is difficult, and it is a good topic for further research ☺

LECTURE 7: THE POLYNOMIAL HIERARCHY

Let's recall our favorite NP-complete language:

Definition 36. CLIQUE contains all representations $\langle G, k \rangle$, where the graph G contains a clique of size at least k . That is, it contains a fully-connected subgraph of at least k nodes.

This is in k because our certificate could just be the clique itself.

We can modify this to come up with a slightly stranger language:

Definition 37. ECLIQUE contains all representations $\langle G, k \rangle$ where the largest clique in G has size exactly k .

It's unclear whether ECLIQUE is in NP; we can come up with a certificate to show that it has a clique of size k , but how do we efficiently show that G has no larger cliques? Similarly, it's unclear whether ECLIQUE is in coNP.

If we're more precise about the definition, we can say that

A string $\langle G, k \rangle$ is in ECLIQUE if and only if there **exists** a clique of size k , and **for all** subgraphs S of size $k + 1$, S is not a clique.

We can consider another example:

Definition 38. The language SMALLEST-CNF consists of all CNF formulas φ such that there are no smaller encodings of the same function. More specifically:

$\langle \varphi \rangle$ is in SMALLEST-CNF if and only if **for all** φ' of size smaller than φ , there **exists** some assignment x such that $\varphi'(x) \neq \varphi(x)$.

In both of these examples, we have two quantifiers, and it's unclear how to place them in NP or coNP! So it seems like it's time to introduce more classes:

Definition 39. We say L is in Σ_2 if there is a polytime verifier V that takes in x, w_1, w_2 (where w_1 and w_2 are poly-length in x) and x is in L if and only if there **exists** w_1 such that **for all** w_2 , $V(x, w_1, w_2)$ accepts.

So ECLIQUE is in Σ_2 .

Definition 40. We say L is in Π_2 if there is a polytime verifier V that takes in x, w_1, w_2 (where w_1 and w_2 are poly-length in x) and x is in L if and only if **for all** w_1 there **exists** w_2 such that, $V(x, w_1, w_2)$ accepts.

So SMALLEST-CNF is in Π_2 .

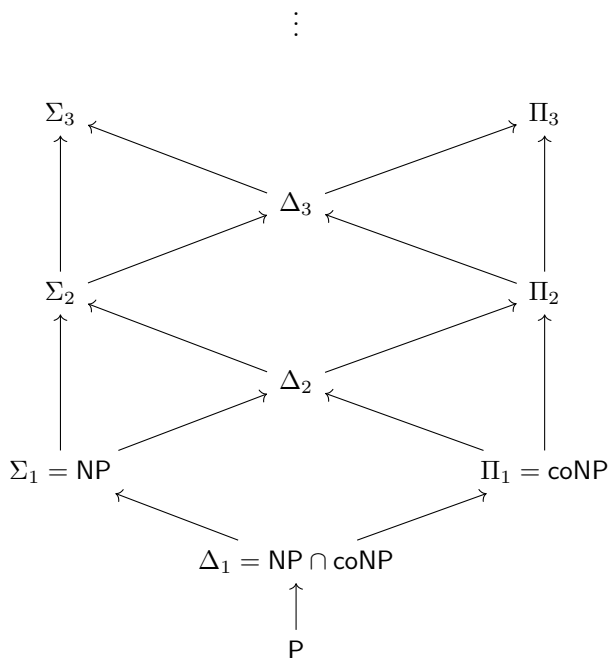
From this, we get a much more general definition:

Definition 41. A language L is in Σ_k if there is a polytime verifier V such that $x \in L$ if and only if there exists w_1 such that for all w_2 there exists w_3 such that ... $V(x, w_1, \dots, w_k)$ accepts. The class Π_k is defined the same way, with the alternating qualifiers, but starting with "for all."

We can see that this gives us a hierarchy where

$$\begin{aligned} \Sigma_k &\subseteq \Sigma_{k+1} \cap \Pi_{k+1} \\ \Pi_k &\subseteq \Sigma_{k+1} \cap \Pi_{k+1}. \end{aligned}$$

We define $\Delta_k = \Sigma_k \cap \Pi_k$, and then we have the following picture:



(As a note, I kept getting confused about why Π_k would not be a subset of Σ_k , since it seemed like if your Σ_k verifier accepted x , then a corresponding Π_k verifier would necessarily also accept x . My confusion here was that I kept thinking in terms of words in a language, rather than languages as a whole. Intuitively, if you have $L \in \Sigma_k$, it is harder to get a Π_k verifier to reject all $x \notin L$, and if you have $L \in \Pi_k$, it is harder to get a Σ_k verifier to accept all $x \in L$. This allows them to correspond to different languages, and not necessarily have one be a subset of the other.)

Definition 42. We define the **polynomial hierarchy** to be the top of this hierarchy, or

$$\text{PH} = \bigcup_k \Sigma_k = \bigcup_k \Pi_k.$$

There is a popular conjecture, which is much stronger than $\text{NP} \neq \text{coNP}$:

Conjecture 43. For all k ,

$$\Sigma_k \subsetneq \Pi_{k+1} \subsetneq \Sigma_{k+2}.$$

So all the inclusions are strict, and $\text{NP} \neq \text{coNP}$, so $\text{P} \neq \text{NP}$.

We can place PH among our familiar classes:

Theorem 44. $\text{PH} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$

Proof. The latter part is what we proved in [Theorem 20](#). For the former part, we can just iterate over all possible values of the w_i 's. At any point in time, we only need to keep track of the current value of the w_i 's that we are testing, plus simulating the verifier, which we know runs in polynomial space. Thus, in total, we need no more than polynomial space. \square

But if $P = NP$, the entire polynomial hierarchy collapses!

Theorem 45. If $P = NP$, then $PH = P$.

Intuitively, this is because if $P = NP$ then $P = \text{coNP}$ as well, and then you can get rid of each quantifier one by one.

For more detail, we will prove the following (smaller, but similar) theorem:

Theorem 46. If $P = NP$ then SMALLEST-CNF is in P .

Proof. Remember that φ is in SMALLEST-CNF if for all φ' of size smaller than φ , there exists x such that $\varphi'(x) \neq \varphi(x)$.

We will first consider the language L of all $\langle \varphi, \varphi' \rangle$ such that there exists some x for which $\varphi(x) \neq \varphi'(x)$. Clearly, L is in NP , since a valid x would be a certificate. Then, if $P = NP$, we know that A is in P , so there is a Turing machine V such that $\langle \varphi, \varphi' \rangle \in L$ if and only if $V(\varphi, \varphi') = 1$.

But this means that φ is in SMALLEST-CNF iff for all φ' , $V(\varphi, \varphi')$ accepts. But this means SMALLEST-CNF is in coNP , which equals P by our assumption. \square

More generally, we have the theorem:

Theorem 47. If $\Sigma_k = \Pi_k$ then $PH = \Pi_k = \Sigma_k$, so the polynomial hierarchy collapses to the k^{th} level.

A natural question is: are there Σ_k -complete languages? For languages we can describe naturally, it's really hard to show Σ_2 or Σ_3 completeness. This should make sense because if we have a bunch of nested quantifiers, our language probably isn't super natural sounding.

But we do have some more constructed Σ_k -completed languages.

Definition 48. We can define $\Sigma_k\text{SAT}$ to be the language containing Boolean formulas σ where there exists some z_1 such that for all z_2 there exists z_3 such that ... $\varphi(z_1, z_2, \dots, z_k)$ accepts.

(A small note here is that for $\Sigma_k\text{SAT}$ to be a valid language, the boolean formulas σ must be of any length, not just the ones which take in k variables. To account for this, if n is the number of input variables for σ , then we can say that each z_i actually represents a sequence of n/k bits.)

We define $\Pi_k\text{SAT}$ analogously.

Example 49. $\Sigma_1\text{SAT}$, or NPSAT is just SAT , and $\Pi_1\text{SAT}$, or coNPSAT , is TAUT , or the set of all cnf formulas that are tautologies.

Proposition 50. $\Sigma_k\text{SAT}$ is Σ_k -complete, and $\Pi_k\text{SAT}$ is Π_k -complete.

Remember that everything in PH has a constant number of quantifiers. If we allow more than a constant number of quantifiers, we get something that's PSPACE complete.

Proposition 51. The language TQBF , or $\Sigma_n\text{SAT}$, where we can have a separate quantifier for each variable in our boolean formula, is PSPACE -complete.

This tells us something interesting:

Theorem 52. If $\text{PH} = \text{PSPACE}$, then TQBF has a constant number of quantifiers. This means it is in Σ_k for some k . But by completeness of TQBF, this means that $\text{PH} = \Sigma_k$, so the polynomial hierarchy collapses to the k^{th} level.

So we generally believe that $\text{PH} \subsetneq \text{PSPACE}$. We can describe PSPACE using TQBF, which means we generally think of PSPACE as the space of games. Intuitively, this is because we can think of the alternating quantifiers as two players competing: for anything you can do, there exists something I can do, such that for anything you do after that ... (polynomially many times) I win.

LECTURE 8: A BLACK-BOX SAT-SOLVER

Today, we will consider the difference between having the code for an efficient SAT-solver and having a black box that solves SAT efficiently.

Last time, we showed that if Alice invents a polytime algorithm for SAT, then the entire polynomial hierarchy collapses, and we get that all of PH just equals P. Does the same thing happen if she has a black box for SAT?

Specifically, we are comparing two parallel universes. In the first, Alice has a polytime algorithm for SAT. In the second, Alice has a black box. She can feed it a cnf formula φ , and it will quickly tell her whether or not φ is satisfiable.

We can see a few things immediately:

- The first universe is at least as powerful as the second, because with the code for a SAT-solver, we can simulate the second universe easily.
- In both situations, $P = NP = \text{coNP}$, since we can reduce all problems in NP or coNP to SAT (or UNSAT) and then solve the cnf formula or feed it into our black box to get the answer.

Recall [Theorem 46](#) from last lecture, where we showed that if Alice is in the first universe, she can solve SMALLEST-CNF in polynomial time. Can she do the same thing in the second universe?

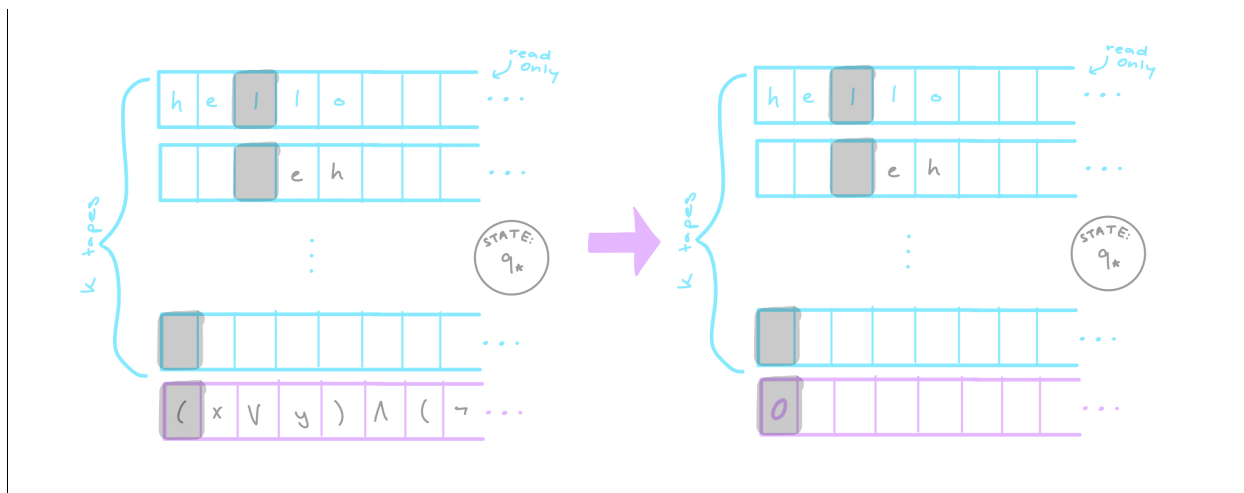
Well, we can try the same thing as before; we start with the inner language L of all $\langle \varphi, \varphi' \rangle$ such that there exists some x for which $\varphi(x) \neq \varphi'(x)$. Since L is in NP, we know we can reduce it in polynomial time to SAT and then use our SAT-solver to get an answer. We will say that $V(\varphi, \varphi')$ is the machine that decides L in polynomial time, using our black-box SAT-solver.

But then, we want to say that $\varphi \in \text{SMALLEST-CNF}$ if for every smaller φ' , $V(\varphi, \varphi')$ accepts. However, this isn't necessarily in coNP. Specifically, we cannot convert it to UNSAT using Cook-Levin, because we don't know how to model our SAT-solver oracle call in our cnf formula. So the strategy from before doesn't exactly work, and it's unclear how we would decide SMALLEST-CNF in polynomial time even with our SAT-solver.

For the rest of the lecture, we will try to figure out the exact boundaries of Alice's power in the second universe. We know that it contains NP and coNP; we will show that her power is contained within $\Sigma_2 \cap \Pi_2$ (by "her power," we mean the set of all problems she can solve in polynomial time).

Let's begin by formalizing Alice's power:

Definition 53. We say that a **SAT-oracle** Turing machine is a Turing machine with an extra "oracle" tape that it can read/write to. Specifically, it can write a cnf formula φ to the tape, and then call an **oracle** operation, which will, in constant time, replace the contents of the "oracle" tape with 1 if φ is satisfiable and 0 otherwise.



Definition 54. We say that Alice's power in the second universe is P^{SAT} , which denotes all the languages decidable in polynomial time with a SAT-oracle Turing machine.

Theorem 55. $\mathsf{P}^{\text{SAT}} \subseteq \Pi_2 \cap \Sigma_2$.

Proof. We will just show that $\mathsf{P}^{\text{SAT}} \subseteq \Sigma_2$; since it is closed under complementation, this will imply it is also a subset of Π_2 .

For any language $L \in \mathsf{P}^{\text{SAT}}$, which is decidable by a polytime SAT-oracle Turing machine M , we want to show that there exists some polytime verifier V such that $x \in L$ if and only if there exists w_1 such that for all w_2 , $V(x, w_1, w_2)$ accepts.

But before this, we will first try (and fail) to prove that $L \in \text{NP}$, because understanding why this fails is instructive:

We want some verifier V such that $x \in L$ if and only if there exists w such that $V(x, w)$ accepts. We can imagine that M makes $m = \text{poly}(n)$ oracle calls when running on x . Then, we can just set our witness to be $w = b_1, \dots, b_m$, where b_i is the answer to the i^{th} oracle call. Then, V just needs to simulate $M(x)$, but pause to reach the next bit from w whenever M makes an oracle call.

But there is a problem here - clearly if $x \in L$, there is a valid certificate which is just the correct sequence of oracle responses in the run of $M(x)$. But if $x \notin L$, we could trick the verifier by giving the wrong answer to the oracle calls at some point.

So we need to give the answers along with proof that we have the correct answer to our oracle calls. For places where the answer is 1, this is easy - just give the satisfying assignment. But for places where the answer is 0, this is UNSAT, and if we knew how to give a certificate for UNSAT, then we would know $\text{UNSAT} \in \text{NP}$. So we are stuck, and we need an extra quantifier to help us out.

In our actual problem, we want there to exist w_1 such that for all w_2 , $V(x, w_1, w_2)$ accepts. We will do the same thing as above - first have w_1 include b_1, \dots, b_m , or the m oracle call answers that M receives when running on x . Then, for each i such that $b_i = 1$, include the satisfying assignment z_i in w_1 . For each i such that $b_i = 0$, include *any* assignment z_i in w_2 . Since we want V to accept for all w_2 , we can see that we want z_i to *not* satisfy our cnf for all possible assignments z_i . This is a proof of UNSAT.

So our verifier then just simulates $M(x)$. Every time M performs an oracle call, it reads the corresponding bit b_i in its certificate. If $b_i = 1$, it checks that z_i is a satisfying assignment, and if $b_i = 0$, it checks that z_i is *not* a satisfying assignment. There are clearly valid certificates for $x \in L$. If $x \notin L$, then to trick the verifier we have to lie about at least one of the b_i 's. But if we are pretending $b_i = 1$ when it is supposed to be 0, we won't be able to provide a satisfying assignment z_i , and if we are pretending $b_i = 0$ when it is supposed to be 1, there is some satisfying assignment z_i , so V won't accept for all w_2 .

So, $L \in \Sigma_2$, and generalizing, $\text{P}^{\text{SAT}} \subseteq \Pi_2 \cap \Sigma_2$. □

LECTURE 9: INTRO TO RANDOMNESS

We now turn to the idea of randomness! We will think of randomness as a resource, so that we can ask questions such as “Can this resource be traded off with time or space?”

Specifically, we will discuss the class BPP, which for now we can think of informally as “randomized P.” Clearly, $\text{BPP} \supseteq \text{P}$. Does the reverse inclusion hold?

In the past, many complexity theorists believed BPP was a strict superset of P. However, most of them now believe that $\text{BPP} = \text{P}$, and an interesting area of research is the question of how to minimize the number of randomness used, or try to make a randomized algorithm deterministic.

We like randomness, because a lot of the fastest or most space-efficient algorithms we know are randomized. (For a cool example, look at [Karger’s algorithm](#) for the minimum cut on a graph.) Moreover, these algorithms are often very clean, and even proofs of their correctness tend to be simple!

However, randomized algorithms tend to have a bad worst-case scenario, where either they take arbitrarily long or they just return the wrong answer, and proofs of randomized algorithms are often non-constructive, so they’re less helpful for further understanding.

The reason we believe that $\text{P} = \text{BPP}$ is Impagliazzo and Wigderson proved a theorem that says: “If SAT requires exponential-sized circuits (a slight strengthening of $\text{P} \neq \text{NP}$), then $\text{P} = \text{BPP}$.”

Essentially, the proof of this statement shows that if SAT requires exponential-sized circuits, then we have a good pseudorandom generator, and we can replace any use of randomness with this pseudorandom generator. We will prove this in detail later.

Let’s look at a few examples where randomness seems useful.

Example 56 (Primality Testing). We know that AKS, which was shown in 2002, gives a deterministic $O(n^\epsilon)$ algorithm to check whether a number is prime.

Before that, in the 70s, [Miller and Rabin](#) gave an $O(n^3)$ randomized algorithm for primality testing.

Example 57. If we are given $n \times n$ matrices A, B, C , how do we verify that $AB = C$?

Freival’s randomized algorithm generates a random n -dimensional vector r and checks whether $A(Br) = Cr$. This takes $O(n^2)$ time. In contrast, the best-known deterministic algorithm takes around $O(n^{2.3})$ time.

Example 58 (Minimum Spanning Tree). As mentioned above, Karger-Klien-Tarjan came up with a randomized algorithm to find the minimum spanning tree of a graph in $O(m)$ time, where m is the number of edges. It is a big open problem whether there is an equally-fast deterministic algorithm.

Example 59 (3SAT). The best known randomized algorithm takes $O(1.31^n)$ time while the best known deterministic algorithm takes $O(1.34^n)$ time.

Example 60 (undirected STCONN). Here, we have a randomized algorithm that saves *space* instead of time. Specifically, we can decide undirected STCONN in $O(\log n)$ space by starting at s , walking randomly for $O(n^3)$ steps, and accepting if we ever see t . In 2000, Reingold came up with a deterministic algorithm for undirected STCONN that also only uses $O(\log n)$ space.

Example 61 (Polynomial Identity Testing). The question is: given two polynomials, are they equal? (For a more specific description, see the Wikipedia article [here](#).)

The randomized strategy is to evaluate both polynomials at random values, and see if they evaluate to the same thing. We don't have a good deterministic strategy.

LECTURE 10: RANDOMIZED COMPLEXITY CLASSES

Let's begin by formally defining a randomized Turing machine:

Definition 62. A randomized Turing machine, like the nondeterministic one, is a Turing machine M with two transition functions, δ_1 and δ_2 . However, instead of nondeterministically branching at each timestep, we randomly pick which transition function to apply at each timestep, each with probability $\frac{1}{2}$. We say that x is in the language of our randomized Turing machine if the probability that M accepts x is at least $2/3$; otherwise x is *not* in the language.

Definition 63. The class $\text{BPTIME}(t(n))$ is the set of all language L such that a randomized Turing machine decides L in $O(t(n))$ time.

Then, BPP, or Bounded-error Probabilistic Polynomial time, is

$$\text{BPP} = \bigcup_c \text{BPTIME}(n^c).$$

Why did we choose $2/3$? This was sort of an arbitrary choice:

Proposition 64. If our cutoff probability was instead any constant $0 < \epsilon < 1$, we would still get the same class BPP.

If our cutoff probability was 2^{-n} , where n is the input length, we also still get BPP.

If the cutoff probability was 1 (x is in the language only if the machine always accepts, and otherwise it is not) then we are working with deterministic machines, so we get P.

If the cutoff probability was 0 (x is in the language if we accept with any nonzero probability, and otherwise it is not) then we get NP.

If instead x is in the language if we accept with probability $2/3$, and not in the language if we never accept, we get a new class RP.

If we swap the probabilities, so that $x \in L$ if we always accept and not in the language if we accept with probability at most $1/3$, we get coRP.

If the accept and reject probabilities are something in between, like $(.9, .8)$ or $(.1, .01)$, we get BPP again.

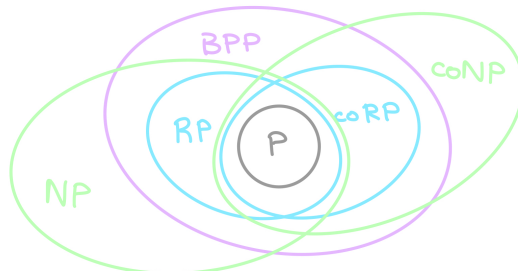
But if we have a machine M in RP or BPP, we can arbitrarily reduce the error probabilities:

Lemma 65 (RP error reduction). Given a language L , if M accepts $x \in L$ with probability $2/3$ and always rejects $x \notin L$, we can create M' which accepts $x \in L$ with arbitrarily high probability.

Proof. Have M' run $M(x)$ k times, and accept if M ever accepts. We know that we never get a false positive, so M' always rejects $x \notin L$. Moreover, since M gives us a false negative with probability $1/3$, we can see that M' gives us a false negative with probability 3^{-k} , so it accepts $x \in L$ with probability $1 - 3^{-k}$. \square

Exercise 66. We can arbitrarily reduce error probabilities for $M \in \text{BPP}$ in the same way - can you see how?

Error reduction tells us that we get the following Venn diagram of classes:



Of course, if (as we suspect) $P = \text{BPP}$, then all of these random classes collapse into P .

Actually, we can show that $\text{BPP} \subseteq \text{PH}$ (a surprising fact - we can trade out randomness for quantifiers!), and therefore if $P = \text{NP}$, $\text{BPP} = P$. For a proof that $\text{BPP} \subseteq \text{PH}$, see page 127 of Arora-Barak.

Here is an equivalent definition of a randomized Turing machine:

Definition 67. A randomized Turing machine is a deterministic Turing machine, with access to random bits as auxiliary input. That is, M takes in x and a random binary string r .

Then, a language L is in BPP if there is a polytime Turing machine M such that if $x \in L$, $M(x, r)$ accepts with probability at least $2/3$ (where the probability is over the choice of r) and if $x \notin L$, $M(x, r)$ rejects with probability at least $2/3$.

This corresponds well with our two definitions of NP ; we can think of our r as the randomized equivalent of a certificate.

Theorem 68. BPP is in EXPTIME .

Proof. If we view M as a randomized Turing machine in the above definition, then we can have a exptime M' that just brute-forces $M(x, r)$ for all choices of r , and then accepts if the majority of $M(x, r)$ accept, and rejects otherwise. \square

But we actually have a stronger statement:

Theorem 69. BPP is in PSPACE .

Proof. We just do the same thing as above, but erase our work after each r we test. \square

Ok, so what did we mean that if (something slightly stronger than) $P \neq \text{NP}$, then $\text{BPP} = P$?

Well, instead of brute-forcing over all possible poly-length strings r , we will brute-force over all possible log-length strings r , and then use a pseudo-random generator to “stretch” r into the length we need.

Specifically, we want to make a *pseudorandom generator* G that runs in $\text{poly}(n)$ time, takes as input a $\log n$ -length string, and outputs a length- n string that “fools” our randomized TM M . Specifically, for every x , the probability (over all r) that $M(x, r)$ accepts should be approximately the probability (over all s) that $M(x, G(s))$ accepts. There are some cool weird things that we do to create these pseudorandom bits in practice, but we will not discuss this in this class \ominus

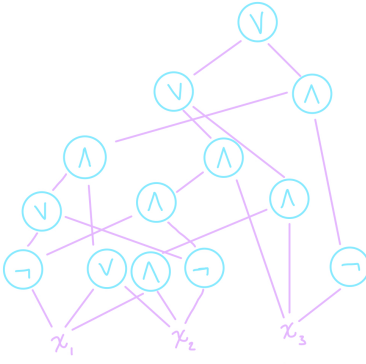
LECTURE 11: CIRCUITS

Today, we're going to prove the result we discussed last lecture:

Theorem 70 (Impagliazzo-Wigderson). If SAT requires exponential-size circuits to compute, then $P = BPP$.

Circuits are more powerful than TMs, so this is a stronger assumption than $P \neq NP$. What exactly is a circuit?

Definition 71. Boolean circuits take an input of n boolean variables, x_1, \dots, x_n , and use AND, OR, and NOT gates to generate a single output:



We can say that the circuit C computes a function $\{0, 1\}^n \rightarrow \{0, 1\}$; we call this function C as well.

We measure circuit complexity using its **size**, or the number of gates in the circuit. Morally, we can think of this as time complexity. In contrast, the **depth** of a circuit is the longest path from input to output; morally, this is parallel time complexity.

But a circuit can only compute a function on a fixed-length input, so it cannot decide a language in the same way. This motivates the following definition:

Definition 72. A **family** of circuits is a collection $\{C_n\}_{n \in \mathbb{N}}$ where we have one circuit corresponding to each input length. Each circuit family computes some language L .

A circuit family has **size complexity** $s(n)$ if for every n , the size of C_n is at most $s(n)$.

We say that circuit families are a **non-uniform** model of computation, in contrast to Turing machines, which are a **uniform** model of computation.

Definition 73. We say that $SIZE(s(n))$ is the set of all language computable by a circuit family of size $O(s(n))$. We say that P/poly is

$$P/poly = \bigcup_c SIZE(n^c),$$

so it is the circuit analogue of P.

What can we say about these size complexity classes? First of all, *every* binary language is in $SIZE(2^{O(n)})$.

Proposition 74. For any language L , there is a circuit family of size $2^{O(n)}$ that computes L .

Proof. With $2^{O(n)}$ gates, we can simply encode every word in L , and compare our input to all of them! Specifically, our circuit C_n looks like:

$$C_n(x) = \bigvee_{\substack{y \in L \\ |y|=n}} \bigwedge_{i=1}^n (x_i == y_i),$$

where we can encode $x_i == y_i$ as $(x_i \wedge y_i) \vee (\neg x_i \wedge \neg y_i)$. Our circuit says: go through every length- n word in our language. If we find one that matches every bit of x , then accept, and otherwise reject. \square

But even P/poly is more powerful than we might expect! It actually contains undecidable languages:

Proposition 75. Every unary language is in P/poly.

Proof. A unary language has at most one string of length n , for each n . Thus, our circuit C_n can simply compare this input against this one string, which it can do in $O(n)$ space. \square

But not all unary languages are decidable; for example, we can consider the language L of all unary strings of length n , where n expressed in binary encodes $\langle M, x \rangle$ such that M halts on x . This is the halting problem, so it is clearly undecidable!

So it seems like we've made P/poly way too powerful! We want something that's a bit more practical - maybe a circuit family where it's easy to generate the entire family at once.

Definition 76. A circuit family is P-uniform if it is possible to efficiently precompute each of the circuits: that is, if there is a Turing machine M that will print C_n when it gets the input 1^n .

Then, we've just described P again:

Theorem 77. The class P-uniform P/poly equals P.

Proof. One direction of this is clear: if we have a P-uniform polynomial-size circuit family $\{C_n\}$ computing L , we can consider the Turing machine that, upon receiving an input x , first computes C_n in polynomial time and then runs $C_n(x)$ in polynomial time.

For the other direction, note that using the Cook-Levin theorem, we can convert any TM in P to a polynomial-sized cnf formula, which is a polynomial-sized circuit. \square

But there is also a Turing machine definition of P/poly!

Definition 78. An **advice-taking** Turing machine is a Turing machine that also gets a read-only "advice string" of length $\text{poly}(x)$, where the advice string can depend on the length of x but not x itself.

Definition 79. A language L is in P/poly if there is some advice-taking Turing machine M , and a set of advice strings y_1, y_2, \dots such that $x \in L$ if and only if $M(x, y_{|x|})$ accepts.

This is similar to the verifier definition of NP, except we get to preset the advice strings, so we don't need to worry about our machine getting tricked by false advice.

Theorem 80. Our two definitions of P/poly are equivalent.

Proof. To get from the circuit definition to the Turing machine definition, just have the advice string be an encoding of the circuit C_n .

To get from the Turing machine definition to the circuit definition, we apply the Cook-Levin theorem again. \square

LECTURE 12: MORE ON P/POLY

Today, we will prove two main results:

Theorem 81 (Adelman). $\text{BPP} \subseteq \text{P/poly}$. Informally, we can trade randomness for non-uniformity.

Theorem 82 (Karp-Lipton). If SAT is in P/poly then the polynomial hierarchy collapses to the second level.

The second result gives us reason to believe SAT is not in P/poly, since we don't think the polynomial hierarchy collapses. So even though P/poly contains undecidable languages, it is probably not a superset of NP.

Recall that [Definition 79](#) gives us a definition of P/poly relating to advice-taking Turing machines, and [Definition 67](#) gives us a definition of BPP relating to Turing machines that take in random bits. So to prove that $\text{BPP} \subseteq \text{P/poly}$, we want to move from the statement “for every x , most random strings r give the correct answer” to “there exists some r that works for every x of a given length.” We will do that now.

Proof. Remember that we can amplify probabilities for BPP significantly with only a small increase in runtime. So, amplifying the success probability to $1 - 4^{-n}$, where n is the length of x , we can say that if $L \in \text{BPP}$, then there is a randomized Turing machine M such that for all strings x ,

$$\mathbb{P}_{r \in \{0,1\}^{\text{poly}(n)}} (M(x, r) \text{ is wrong}) \leq 4^{-n}.$$

We will say that r is bad if it causes $M(x, r)$ to be wrong for some x of length n . We want to show that for every n , we can find some r_n that is not bad, so that $M(x, r_n)$ is correct for all $|x| = n$; then r_n is our advice for M and we have converted our Turing machine to an advice-taking one.

Well, how many bad strings are there? We can take a union bound: just add up the number of bad strings for each x . We can see that for each x , the portion of r 's that are bad is at most 4^{-n} . There are 2^n strings of length n , so this means at most a $2^n \cdot 4^{-n} = 2^{-n}$ portion of the r 's are bad. So we can take any of the remaining r 's to be our advice string, and we have successfully turned M into an advice-taking Turing machine. \square

(Note that from here, if we were trying to show that $\text{BPP} \subseteq \text{P}$, we would want to show that we can efficiently/deterministically find these r_n .)

We move on to the proof of our second theorem.

Proof. We want to show that if SAT is in P/poly then $\Sigma_2 = \Pi_2$. Since $\Pi_2\text{SAT}$ is Π_2 -complete, we do this by showing that $\Pi_2\text{SAT}$ is in Σ_2 if SAT is in P/poly.

Remember that $\Pi_2\text{SAT}$ consists of all cnf formulas φ such that for all x , there exists y such that $\varphi(x, y) = 1$.

We want to show that this is in Σ_2 . In other words, we want to show that there exists a Turing machine M such that $\varphi \in \Pi_2\text{SAT}$ if and only if there exists v such that for all w ,

$$M(\varphi, v, w) = 1.$$

For any such w , define $\varphi_w = \varphi(w, \cdot)$. So $\varphi \in \Pi_2\text{SAT}$ iff there exists y such that $\varphi_w(y) = 1$. But since SAT is in P/poly, we know there is some circuit family $\{C_m\}$ that decides SAT. Specifically, there is some m such that $C_m(\varphi_w)$ will tell us if φ_w is satisfiable.

But then, we can take v to be the description of C_m , and then our machine M will create φ_w (from w) and load it into C_m (from v) to check if it is satisfiable. If it is for all w , then $\varphi \in \Pi_2\text{SAT}$, as we wanted.

But there is still a problem! If φ is not in $\Pi_2\text{SAT}$, we can still trick M by feeding it a bogus v , such as the circuit that just accepts everything. So we need to modify our description of M slightly.

Specifically, remember from pset 1 that we can go from C_m , which takes in a cnf and tells us if it is satisfiable to C'_m , which takes in a cnf and outputs a satisfying assignment. So we let v be C'_m instead, and we have M check that the assignment it outputs actually satisfies φ_w before we accept. \square

Kolmogorov had a bold conjecture: he believed that circuits are so powerful that all of P is captured by just linear-sized circuits! It is still an open problem whether this is true.

LECTURE 13: INTERACTIVE PROOFS

Let's say you can't distinguish between blue and green, and I give you the following two socks.



How do I convince you they're two different colors?

The answer is that I tell you at first that the left one is green. Then, I let you swap them without me looking, and pick out the green one again. If we repeat this process c times, and I get the answer right every time, the probability I'm lying to you is $1/2^c$.

In this lecture, we will consider interactive proofs of the above form, and look at the tradeoffs between interaction and other resources.

We begin by looking at NP as a kind of interactive proof system. Let's say there is an all-powerful prover P that wants to prove to the polytime Turing machine verifier V that $x \in L$, where L is some language in NP.

In this case, all the prover needs to do is send the certificate to V , and V can verify it in polynomial time. Since $L \in \text{NP}$, we know this can be done such that

1. an honest prover can always find a certificate to prove that $x \in L$
2. a dishonest prover cannot trick V into believing $x \in L$ when it is not

In 1985, Goldwasser, Micali, and Rackoff asked the question: what if we allow the prover and the verifier to interact?

Specifically, let's say P and V can talk for some constant k rounds. So the prover sends y_1 , the verifier responds with y_2 , and so on, and in the end, the verifier checks if $V(x, y_1, y_2, \dots, y_k)$ accepts.

We want to do this in a way such that the verifier cannot be tricked; that is,

1. if $x \in L$, then there is some prover strategy such that $V(x, y_1, \dots, y_k)$ accepts
2. if $x \notin L$ then for *all* prover strategies, $V(x, y_1, \dots, y_k)$ rejects

Theorem 83. If V is a deterministic polytime Turing machine, this is just NP.

Proof. The key is that since V is deterministic, and P is more powerful, P can just figure out y_2 on its own, generate y_3 based on that, then figure out what y_4 the verifier would have sent, and so on. So it can just generate the whole transcript (y_1, \dots, y_k) as a singular certificate, and the verifier just needs to check that y_2, y_4 , etc. are actually what it would have said before computing $V(x, y_1, \dots, y_k)$. \square

The key we used in the sock game was the verifier had access to randomness that was hidden from the prover. This allows us to have more interesting interactive proofs; the prover cannot simulate the entire conversation on its own because now the verifier knows something it does not know.

Let us consider an example of something not necessarily in NP that we can use interactive proofs for:

Consider the graph isomorphism problem: given two graphs G and H , are they isomorphic?

Clearly, this is in NP; a valid certificate is just the permutation of the vertices that will take us from G to H .

We consider the graph *non*-isomorphism problem; our language is the set of $\langle G, H \rangle$ such that G and H are not isomorphic. Clearly, this is in coNP, but it is unclear whether it is in NP. However, we can use the following interactive proof to check non-isomorphism, which is essentially the sock game:

Our prover is trying to show that G and H are non-isomorphic; this means he can distinguish between the two. The verifier randomly picks one of the two graphs, G or H . She relabels the vertices of this graph randomly, and sends it to the prover. He responds with 0 if the new graph is a permutation of G , and 1 if it is a permutation of H , and the verifier accepts if the prover is correct.

This only took two rounds of communication! We can see that the prover can always pick the correct graph if the two are not isomorphic, but he only has a $1/2$ probability of getting it right if they are isomorphic. Moreover, we can decrease this probability arbitrarily just by repeating this process, and this doesn't even increase communication complexity because we can send all of our random permutations in the same message.

We define a new class of languages:

Definition 84. $\text{IP}[k]$ is the set of all languages L that have k -round interactive proofs, with an all-powerful prover and a randomized polytime verifier such that:

- If $x \in L$, then there exists a prover strategy such that the verifier accepts with probability at least $2/3$
- If $x \notin L$, then for all prover strategies, the verifier rejects with probability at least $2/3$

Then, $\text{IP} = \text{IP}[\text{poly}(n)]$.

What is the power of IP?

Clearly, $\text{BPP} = \text{IP}[0]$ and $\text{NP} \subseteq \text{IP}[1]$. But is coNP in IP? What is the smallest class we can put IP inside?

We answer the second question first:

Proposition 85. $\text{IP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$

Proof. Note that we can simulate all prover strategies in EXPTIME and PSPACE, simply by trying all possible strings. For each of these, we can compute the probability the verifier accepts and keep track of the maximum probability. If this maximum is at least $2/3$, then we know that $x \in L$; otherwise, if it is at most $1/3$, we know that $x \notin L$. \square

We will show next lecture that actually $\text{IP} = \text{PSPACE}$; this was a very surprising result, because it means that if we allow for interaction, we can decide more than the entire polynomial hierarchy!

LECTURE 14: PROVING $IP = PSPACE$

Last lecture, we showed that $NP \subseteq IP \subseteq PSPACE$, where $PSPACE$ is very large, and contains all of PH . The question is: how large is the gap between IP and $PSPACE$?

Importantly to this question is the following problem:

#3SAT: Given a 3CNF φ , how many satisfying assignments does it have?

Note that this problem is a more difficult version of both SAT and UNSAT.

In the 1980's, it was believed that IP was not much more powerful than “randomized NP .” In 1988, Fortnow and Sipser conjectured that $\overline{3SAT}$ is not in IP , which would imply $coNP \not\subseteq IP$. But in December 1989, Lund, Fortnow, Karloff, and Nisan showed that $3SAT \in IP$ and moreover that $\#3SAT \in IP$. Two weeks later, Shamir showed that $IP = PSPACE$, so interaction is a lot more powerful than we originally thought!

Theorem 86. $\#3SAT \in IP$

Proof. Let φ be a 3CNF formula; for example

$$(x_1 \vee x_3 \vee \neg x_9) \wedge (\neg x_2 \vee \neg x_5 \vee \text{neg}x_{50}).$$

Then, note that we can represent the first clause with the polynomial

$$1 - (1 - x_1)(1 - x_3)x_9,$$

and this polynomial will equal 1 if the clause is true and 0 otherwise.

In this way, we can make polynomials for each clause, and multiply them together to get a polynomial $q(x_1, \dots, x_n)$ that represents φ (this is sometimes called the **arithmization** of φ). Both the prover and verifier can build q .

Note that $q(x) = 1$ if x satisfies φ and $q(x) = 0$ otherwise. This means that the number of satisfying assignments of φ is

$$\sum_{b_1=0}^1 \sum_{b_2=0}^1 \cdots \sum_{b_n=0}^1 q(b_1, \dots, b_n)$$

(Also, note that $q(x)$ is a polynomial of degree at most $3m$, where m is the number of clauses in our cnf.)

So the prover wants to prove that this sum equals exactly k , but the verifier cannot compute the sum by itself.

First, the prover gives the verifier a prime p with $2n$ digits (the verifier can check that p is prime in polynomial time). Then, we will show that this sum \pmod{p} equals k ; this is enough to show the sum is actually k .

Then, define the polynomial

$$r(x) = \sum_{b_2=0}^1 \sum_{b_3=0}^1 \cdots \sum_{b_n=0}^1 q(x, b_2, \dots, b_n).$$

So we want

$$r(0) + r(1) = k \pmod{p}.$$

The verifier cannot efficiently evaluate r in this form. But we know that r is a single-variable polynomial of degree at most $3m$, where each coefficient is at most $p - 1$ (and therefore takes at most $2n$ bits to write). So there is a representation of $r(x)$ as

$$\tilde{r}(x) = \sum_{i=0}^{3m} c_i x^i,$$

which will take $3m(2n) = O(mn)$ bits to write, and the verifier *can* efficiently evaluate \tilde{r} .

So the prover sends over \tilde{r} , and the verifier checks that $\tilde{r}(0) + \tilde{r}(1) = k \pmod{p}$. If this is not true, it rejects. If it is true, the prover has to convince the verifier that it sent the correct \tilde{r} . (Note that at this point, the only way the prover is lying is if it sent a \tilde{r} that wasn't equal to r .)

To prove this, the verifier sends a random $0 \leq a \leq p - 1$ to the prover; the prover then has to show that

$$\tilde{r}(a) = r(a).$$

If the prover was lying, then with high probability $r(a) \neq \tilde{r}(a)$, since the polynomials can agree on at most $3m$ values ($r(a) - \tilde{r}(a)$ has degree at most $3m$ and therefore has at most $3m$ zeroes) but $p \approx 2^{2n}$ so there are 2^{2n} possibilities for a . So if the prover is lying, then $r(a) = \tilde{r}(a)$ with probability at $\frac{3m}{2^{2n}}$, which is very small.

To show that $r(a) = \tilde{r}(a)$, we define

$$r_2(x) = \sum_{b_3=0}^1 \sum_{b_4=0}^1 \cdots \sum_{b_n=0}^1 q(a, x, b_3, \dots, b_n),$$

and now the prover has to show that

$$r_2(0) + r_2(1) = r(a) \pmod{p}.$$

(Remember that the verifier can compute $r(a)$.) Using the same strategy as above, we repeat inductively until we get to the $n = 1$ base case, which can be checked directly by the verifier.

Clearly, if the prover was honest, then the verifier will be able to see that the prover was honest by the end. If the prover was lying, then with probability

$$\left(1 - \frac{3m}{4^n}\right)^n,$$

the prover has to lie about the $n = 1$ base case, and the verifier will see that it was lying. □

LECTURE 15: MERLIN-ARTHUR AND ARTHUR-MERLIN

If BPP corresponds to P, what is the randomized analogue of NP?

We will consider two possible answers, and restate these definitions until their meaning becomes clear to us:

The first possible answer is based on the verifier definition of NP:

Definition 87. A language L is in Merlin-Arthur, or MA, if there is a randomized polytime verifier V such that:

- V takes in two inputs, x and y , and the length of y is polynomial in the length of x
- if $x \in A$, then there exists some y such that $V(x, y, r)$ accepts with probability at least $2/3$
- if $x \notin A$, then for all such y , $V(x, y, r)$ rejects with probability at least $2/3$

But here's another answer, based on the idea that any language in NP can be reduced to SAT:

Definition 88. A language L is in Arthur-Merlin, or AM, if there is a BPP reduction to SAT. That is, $L \in \text{AM}$ iff there exists a deterministic polynomial reduction R such that

- if $x \in L$, then the probability (over random strings r) that $R(x, r) \in \text{SAT}$ is at least $2/3$
- if $x \notin L$, then the probability (over random strings r) that $R(x, r) \notin \text{SAT}$ is at least $2/3$

In the Arthur-Merlin definition, we can let V be a verifier that takes in x, y , and r , and then outputs 1 iff y is a satisfying assignment for $R(x, r)$. Then, our definitions become:

Definition 89. A language L is in MA iff

- For all $x \in L$, there is some y such that $\mathbb{P}_r[V(x, y, r) = 1] \geq 2/3$ (the probability over r that $V(x, y, r)$ accepts is at least $2/3$).
- For all $x \notin L$, for all y , $\mathbb{P}_r[V(x, y, r) = 0] \geq 2/3$.

A language L is in AM iff

- For all $x \in L$, $\mathbb{P}[\exists y V(x, y, r) = 1] \geq 2/3$.
- For all $x \in L$, $\mathbb{P}[\forall y V(x, y, r) = 0] \geq 2/3$.

So we seem to have similar quantifiers, but in different locations. To help us describe this we will introduce new notation:

Definition 90. The quantifier \mathfrak{R} is defined to mean “for most,” and in this case, “with at least $2/3$ probability.”

Then, this gives us the following restatement of the definitions:

Definition 91. If $L \in \text{MA}$, then:

- if $x \in L$, $\exists y \mathfrak{R} r V(x, y, r) = 1$.
- if $x \notin L$, $\forall y \mathfrak{R} r V(x, y, r) = 0$.

If $L \in \text{AM}$, then:

- if $x \in A$, $\exists r \exists y V(x, y, r) = 1$.
- if $x \notin A$, $\forall r \forall y V(x, y, r) = 0$.

So MA corresponds to the quantifiers $(\exists \mathcal{R}, \forall \mathcal{R})$, AM corresponds to the quantifiers $(\mathcal{R} \exists, \mathcal{R} \forall)$, BPP corresponds to the quantifiers $(\mathcal{R}, \mathcal{R})$, and NP corresponds to the quantifiers (\exists, \forall) .

Exercise 92. We can think of AM as interaction between a polytime randomized Arthur and an all-powerful Merlin; Arthur first sends Merlin a random challenge, and Merlin responds with something that convinces Arthur. How would we describe MA in terms of interactive proofs?

So how do these classes relate to each other?

Well, we know already that $P \subseteq BPP$ and $NP \subseteq MA$ and $NP \subseteq AM$. We will now show the next two inclusions:

Theorem 93. $MA \subseteq AM$

Theorem 94. $MA \subseteq \Sigma_2$, while $AM \subseteq \Pi_2$

Note the second theorem has the immediate corollary:

Corollary 95. $BPP \subseteq MA \subseteq \Sigma_2 \cap \Pi_2 \subseteq PH$

since we know already that $BPP \subseteq MA$ and the first theorem tells us that $MA \subseteq AM$.

Remember that this tells us that if $P = NP$, $P = BPP = MA = AM$ as well, but the more popular conjecture is that $P = BPP$ and $P \neq NP$. A popular conjecture says that $NP = MA = AM$.

We will now prove the first theorem.

Proof. Suppose $L \in MA$. We want to prove that $L \in AM$.

First, if $x \in L$, we know from the definition of MA that

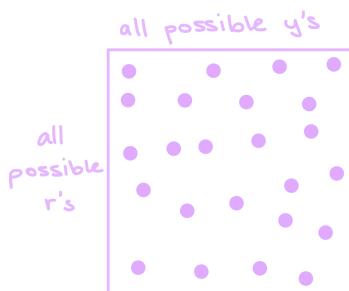
$$\exists y \mathcal{R} \forall r V(x, y, r) = 1. \quad (1)$$

We want to show that

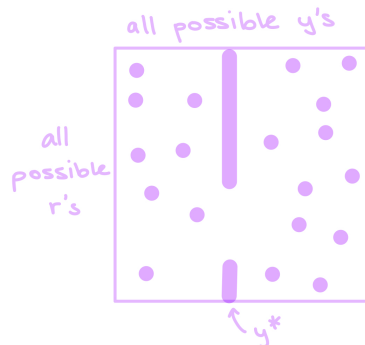
$$\mathcal{R} \exists y V(x, y, r) = 1. \quad (2)$$

But Equation 1 directly implies Equation 2. Why?

Pictorially, let's say we have a grid of all possible y 's and r 's, with points highlighted where the verifier accepts:



Then, Equation 2 says that for two thirds of the rows, we want *some* box in that row to be highlighted. But Equation 1 tells us that there's some special column y^* such that 2/3 of the column is highlighted:



This immediately tells us that $2/3$ of the rows have some box highlighted!

The problem is more difficult for $x \notin L$. When $L \in \text{MA}$, if $x \notin L$ then:

$$\forall y \exists r V(x, y, r) = 0, \quad (3)$$

and we want to show that

$$\exists r \forall y V(x, y, r) = 0. \quad (4)$$

Intuitively, Equation 3 does not imply Equation 4 when both \exists s mean “with $2/3$ probability,” because if each y corresponds to different r 's, there aren't enough r 's that cover *all* y 's.

But if we use amplification, and make the first statement have higher probability, then we should be able to find enough r 's that cover all y 's (essentially by using a fancier version of the pigeonhole principle).

Let $m = |y|$. We know that there are 2^m options for y , and we will use amplification to modify Equation 3 to say that for every y , for $1 - 4^{-m}$ of the r 's, $V(x, y, r) = 0$.

We say that an r is *bad* if it causes $V(x, y, r)$ to equal 1 for some y . If an r is not bad, it is *good*, and we want to show that at least $2/3$ of the r 's are good. (This should start to sound familiar, from the proof of Theorem 81...)

We can see that for each y , 4^{-m} of the r 's are bad, so in general at most $2^m(4^{-m}) = 2^{-m}$ of the r 's are bad. This means at least $2/3$ of the r 's are good, and Equation 4 follows. \square

Now, we will think about Theorem 94. For the rest of this lecture, we will sketch out a proof that BPP is in Σ_2 (this implies $\text{BPP} \in \Pi_2$ since it is closed under complementation, so $\text{BPP} = \text{coBPP} \subseteq \text{co}\Sigma_2 = \Pi_2$).

Remember that BPP corresponds to (\exists, \exists) while Σ_2 corresponds to $(\exists \forall, \forall \exists)$, so intuitively, we are trading out the uncertainty of “for most” for an extra quantifier.

Let $L \in \text{BPP}$. Then, for any x , let R be the set of all possible random strings, and let $R^* \subseteq R$ be the set of random strings r such that $V(x, r) = 1$. The definition of BPP tells us that if $x \in L$, then R^* is a very large subset of R , and if $x \notin L$, then R^* is a very small subset of R .

Essentially, we will make some number of “copies” of R^* , and show that when $x \in L$, these copies cover all of R , but when $x \notin L$, we can always find some r that is not in any of the copies of R^* .

We make these copies by picking a set (s_1, \dots, s_m) of elements of R , and then xoring these with our current r . So our exists-for all statement looks like: if $x \in L$, **there exists** some set (s_1, \dots, s_m) such that **for all** r , some $V(x, r \oplus s_i)$ accepts. If $x \notin L$, then **for all** sets (s_1, \dots, s_m) , **there exists** r such that no $V(x, r \oplus s_i)$ accepts.

We will formalize this idea next lecture.

LECTURE 16: RANDOMNESS AND PH

We begin with our formal proof:

Theorem 96. $\text{BPP} \subseteq \Sigma_2$

Proof. Let $L \in \text{BPP}$. Then, with amplification, we have that if $x \in L$,

$$\mathbb{P}_r[V(x, r) = 1] \geq 1 - 2^{-n}$$

and if $x \notin L$,

$$\mathbb{P}_r[V(x, r) = 1] \leq 2^{-n}.$$

We want to define a Turing machine M such that if $x \in L$, then there exists some w_1 such that for all w_2 , $M(x, w_1, w_2)$ accepts, and if $x \notin L$, then for all w_1 , there exists some w_2 such that $M(x, w_1, w_2)$ rejects.

As in last lecture, remember that V takes in x and a random string r of length $m = \text{poly}(n)$. Let R be the set of all possible random strings, so that $|R| = 2^m$, and let $R^* \subseteq R$ be the set of all r such that $V(x, r)$ accepts.

Here's what we will do: Let w_1 be a set of m binary strings s_1, \dots, s_m , each of length m . Then, note that if we xor r with some s_i , we are essentially moving to a random place in R . Let w_2 be a random string $r \in R$.

Then, for each $1 \leq i \leq m$, $M(x, (s_1, \dots, s_m), r)$ will compute $r \oplus s_i$, and then check whether $V(x, r \oplus s_i)$ accepts. If V accepts for any i , then M accepts; if V rejects for all i , then M rejects. Essentially, we are checking whether any of the s_i 's maps r to an element of R^* .

We want: for $x \notin L$, for all (s_1, \dots, s_m) , there exists r such that M rejects. This is true, because $|R^*|/|R|$ is at most 2^{-n} . So each s_i can convince M to accept at most a 2^{-n} portion of the r 's, and we get that for any given (s_1, \dots, s_m) , $M(x, (s_1, \dots, s_m), r) = 1$ for at most $m \cdot 2^{-n}$ of the r 's, so it must reject at least one of them. (Note that since $m = \text{poly}(n)$, $m < 2^n$ so our fraction is less than 1.)

Moreover, we want: for $x \in L$, there exists (s_1, \dots, s_m) such that for all r , M accepts. We will prove something stronger: with very high probability, a random tuple (s_1, \dots, s_m) will cause M to accept all r .

Let's say we have such a random tuple. For each s_i and r , the probability that $V(x, s_i \oplus r) = 1$ is at least $1 - 2^{-n}$. So the probability that, for a given r , $M(x, (s_1, \dots, s_m), r)$ rejects is the probability that $s_i \oplus r \notin R^*$ for all i , which is at most $(2^{-n})^m$. Summing over all $r \in R$ (or taking the union bound), we get that the probability that M rejects *any* r for a random choice of (s_1, \dots, s_m) is at most

$$2^m (2^{-n})^m = 2^{-(n-1)m},$$

which is very small.

Clearly, then, there must exist at least one tuple (s_1, \dots, s_m) that will cause M to accept all r . (If there wasn't, this probability would be 1, instead of a very small number.) \square

We will use the leeway given to us by this stronger conclusion to show that MA is also in Σ_2 . We return to the proof of [Theorem 94](#):

Proof. We will just show a very rough proof of the fact that $\text{MA} \subseteq \Sigma_2$. Specifically, we just showed above that we can upgrade our quantifiers, so

$$(\exists \mathcal{R}, \forall \mathcal{R}) \subseteq (\exists \mathcal{R}, \forall \mathcal{R}, \forall \mathcal{R}).$$

Then, we know that MA corresponds to the quantifiers $(\exists \mathcal{R}, \forall \mathcal{R})$. We can "plug in" the above statement to get that

$$(\exists \mathcal{R}, \forall \mathcal{R}) \subseteq (\exists \mathcal{R}, \forall \mathcal{R}, \forall \mathcal{R}) \subseteq (\exists \exists \mathcal{R}, \forall \mathcal{R}, \forall \mathcal{R}) = (\exists \mathcal{R}, \forall \mathcal{R}, \forall \mathcal{R}) \subseteq (\exists \mathcal{R}, \forall \mathcal{R}, \forall \mathcal{R}, \forall \mathcal{R})$$

where one set is contained within the other if we move to *weaker* quantifiers. Thus, $MA \subseteq \Sigma_2$. \square

Ok, so now we understand how these classes relate better. But what problems are actually in MA or AM?

We will consider the problem of approximating the number of satisfying assignments for a circuit, and show this is in AM.

Let $\#C$ be the number of strings x such that $C(x) = 1$.

We say that the APPROX problem gives us $\langle C \rangle$ as input and asks us to output some α such that $\#C \leq \alpha \leq 2\#C$.

Moreover, the PD problem gives us $\langle C, s \rangle$ as input, where s is an integer. It asks us to output **yes** if $\#C > 2s$ and **no** if $\#C \leq s$ (this is called a **promise-decision** problem because we are only asked to give a good answer for certain inputs.)

Exercise 97. Can you show that APPROX and PD are equivalent - if you can solve one, you can solve the other?

Theorem 98 (Goldwasser-Sipser). Given a circuit C and a number S , there's an AM protocol such that if $\#C > 2s$ we accept with probability at least $2/3$ and if $\#C \leq s$, we reject with probability at least $2/3$.

Proof. We can give ourselves a bit more room, so we will say that we only need to accept when $\#C > 64s$. We will also assume that $s = 2^k$ for some natural number k .

Here's how our interactive proof works:

- Arthur picks a random hash function $h : \{0, 1\}^n \rightarrow \{0, 1\}^{k+3}$, and sends this to Merlin.
- Merlin picks an x^* such that $C(x^*) = 1$ and $h(x^*) = 0$ (where 0 here means the string 0^{k+3}).
- Arthur verifies that $C(x^*) = 1$ and $h(x^*) = 0^{k+3}$.

We can see that when $\#C > 64s$, then since $\{0, 1\}^{k+3}$ has size $8s$, our hash function randomly maps these $64s$ satisfying assignments to a space of size $8s$, so with high probability, there exists some x^* within these satisfying assignments such that $h(x^*) = 0$.

Moreover, when $\#C \leq s$, then there is less than a $1/8$ probability that any of the s satisfying assignments maps to 0, so with high probability, Merlin can't find an x^* to send to Arthur. \square

There is a small but important problem with this proof, which also arises in a number of situations: Arthur doesn't actually have the computational power to compute and send over a truly random hash function h !

Instead, we will have him select a hash function from a **hash family** of $2^{O(kn)}$ possible hash functions; that way he only needs $O(kn)$ bits to describe the function he selected. But we still need our hash function to be random enough that the above argument holds!

To encode this idea that h needs to be random enough, we will require that all the hash functions in our hash family have 1-wise and pairwise independence: the former means that since 2^{k+3} is the size of our input the probability that $h(x)$ equals any given point is $2^{-(k+3)}$, and the latter means that the probability that $h(x)$ and $h(y)$ both equal a given point for any x, y is $2^{-2(k+3)}$. (Here, the probabilities are over our choice of h within our family.)

So what hash family will we choose?

For this problem, we let Arthur do the following: choose $k + 3$ random strings r_1, \dots, r_{k+3} , where each r_i has n bits. Moreover, choose $k + 3$ random bits b_1, \dots, b_{k+3} . (This is $O(nk)$ bits of information). Then, we can set the i^{th} bit of $h(x)$ to be $x \cdot r_i + b_i$, where for the \cdot operation, we take $x \oplus r_i$, and then output the parity of the resulting string (by xoring together the all the digits of the binary string).

We leave it as an exercise to show that this is has 1-wise and 2-wise independence.