Paradoxes:

- A **paradox** is a self-contradictory statement that at first seems true.
- **Barber's Paradox:** The story is that there's a village that has a male barber who shaves every man in the village who does not shave himself.

  Question: Does the barber shave himself?
  Consider this:
  - If yes:
    → Recall that the barber only shaves men who don't shave themselves.
    → Hence, he does not shave himself.
  - If no:
    → Recall that the barber only shaves men who don't shave themselves.
    → Hence, he does shave himself.
  - **Note:** This is a proof by contradiction that the village in the story doesn't exist.

  Note: The problem in this paradox is that there's a self-referential aspect to what's going on.
  I.e. This is about an action that the barber does or doesn't do on himself.
  Furthermore, there's also a negation. Self-reference with negation turns out to be problematic.

  Note: This self-referential aspect with negation comes up a lot in paradoxes.
- **Liar's Paradox/Epimenides' Paradox:** Consider the proposition: **"This statement is false"**.
  Consider this:
  - If the above proposition is true, then it must be false.
  - If the above proposition is false, then it must be true.

  Notice the self-referential aspect with negation.

  The conclusion we can draw from this is that our language is so powerful that it allows us to state things that are paradoxical.
- **Russell's Paradox:** Let the set R be the set of all sets that are not members of themselves.
  I.e. $R = \{X \mid X \text{ is a set and } X \notin X\}$

  Question: Does the set R contain itself?
  Consider this:
  - If yes:
    → Recall that R is the set of all sets that are not members of themselves.
    → Hence, R doesn't contain itself.
  - If no:
    → Recall that R is the set of all sets that are not members of themselves.
    → Hence, R does contain itself.

  Again, notice the self-referential aspect with negation.
- **Gödel's completeness theorem.** (**Note:** While this is not a paradox, it is very closely connected with paradoxes and is a very important mathematical result.) Using

elementary mathematics involving just integers, addition and multiplication, Gödel was able to write a mathematical statement, S, whose meaning is:
S = "This statement is unprovable".

If S is true, then S is a true unprovable statement.
If S is false, then S is a false provable statement.

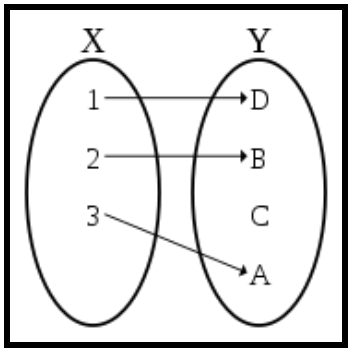
While this isn't a contradiction, it gives us two unpalatable choices.
The first choice says that our axiom system only proves true things but does not prove all true things.
The second choice says that we have an axiom system that allows us to prove false statements, which allows us to prove anything. This is useless.

Size of Infinite Sets:

- **Set Denotations:**
- N = Set of natural numbers
- Z = Set of integers
- Q = Set of rational numbers
- R = Set of real numbers
- **Example:**
- Consider the set of natural numbers, N = {0, 1, 2, …} and the set of perfect squares, PS, PS = {0, 1, 4, …}. These two sets have the same size, not because they are both infinite sets but because there is a **one-to-one correspondence** between the two sets. Take any natural number in N and you will get its corresponding perfect square value in PS. Likewise, take any perfect square in PS and you will get its corresponding square root in N. Furthermore, each value in N has exactly one corresponding value in PS and each value in PS has exactly one corresponding value in N. However, PS is clearly a **proper subset** of N. This shows that two infinite sets can have the same size, even when one is a proper subset of the other. Hence, we must be careful when dealing with infinite sets.

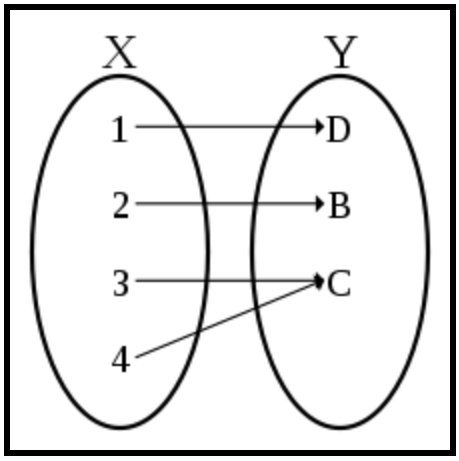
  Note: A **proper subset** of the subset "A" is a subset of "A" that is not equal to "A".
  I.e. If "B" is a proper subset of "A", then all elements of "B" are in "A", but "A" must contain at least one element that is not in "B".
  E.g. Let A = {1,3,5}
  Then, {1}, {1,3}, {1,5} and {3,5} are all proper subsets of A.
  However, {1,3,5} is not a proper subset of A, because it contains all the elements in A. It is a subset of A.
  Lastly, {1,4} is not a subset of A as it contains an element, 4, that is not in A.
- **Definitions:**
- Two sets, A and B, are **equinumerous**, meaning they have the same size or **cardinality**, iff there is a **bijection** f from A to B.
  Note: To show that A and B are equinumerous, we do |A| = |B|.
  |S| denotes the size of set S.
  Note: An **injection**, also called a **one-to-one function**, is a function that maps distinct elements of its domain to distinct elements of its codomain. In other words, every element of the function's codomain is the image of at most one element of its domain.

E.g. of an injection.



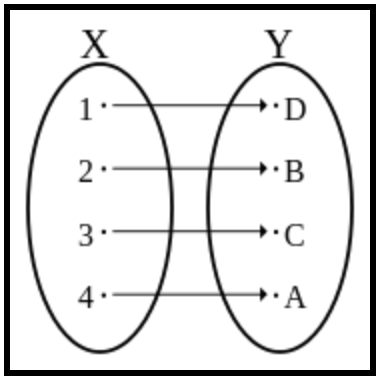
Note: A **surjection**, also called **onto**, is a function such that for every element y in the codomain Y of f, there is at least one element x in the domain X of f such that f(x) = y. It is not required that x be unique; the function f may map one or more elements of X to the same element of Y.

E.g. of an surjection.



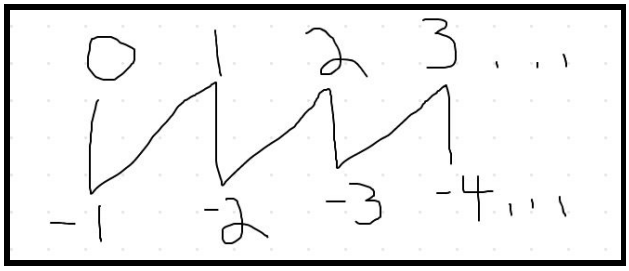
Note: A **bijection**, also called a **one-to-one correspondence**, is a function between the elements of two sets, where each element of the first set is paired with exactly one element of the second set, and each element of the second set is paired with exactly one element of the first set. There are no unpaired elements. A bijective function f: X → Y is a **one-to-one (injective)** and **onto (surjective)** mapping of a set X to a set Y. Please note that the term “one-to-one correspondence” is not the same as a “one-to-one function.”

E.g. of a bijection.

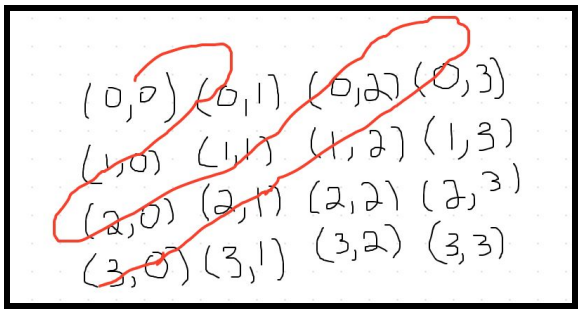


- A set “A” is **countable** or **enumerable** iff “A” is finite or it is equinumerous to N. If “A” is equinumerous to N, we say that “A” is **countably infinite**, which means that there is a bijection f from N to “A”.
  I.e. $f(n) = a_n$ where n is the n^{th} element of N and a_n is the n^{th} of “A”. (This shows the bijection from N to “A”.)

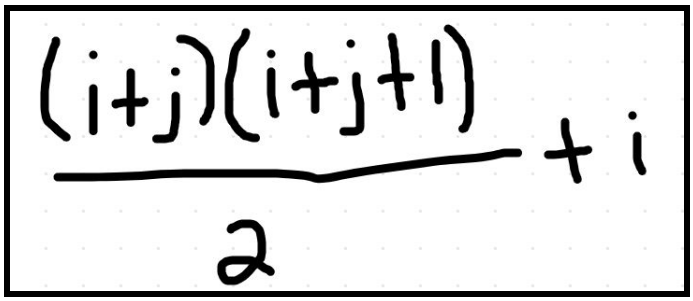
  The **enumeration of a countable set** is a sequence of all the elements in that set such that each element appears exactly once in some particular index of this sequence.
  Note: The sequence does not have to be increasing or decreasing. Any sequence suffices as long as each element appears exactly once in some particular index/position.
  E.g.
  0, 1, 2, 3, …, -1, -2, -3, … is not an enumeration because we cannot give the index of the element -1.
- **Theorem 1.1:** Z is countable.
  Note: Z = …, -3, -2, -1, 0, 1, 2, 3, … is not an enumeration. We cannot give the index of the element 0.
  This is an enumeration of Z:



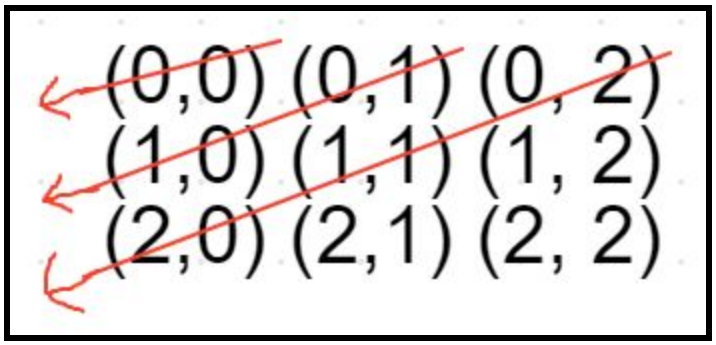
  or
  0, -1, 1, -2, 2, -3, 3, …
  Now, I can give you a specific index of any element.
  This technique is called **dovetailing**. In general, with dovetailing, you take multiple lists and merge them into a single list.
- **Theorem 1.2:** The set of the cartesian product of natural numbers, N x N, is countable.
  This set looks like the following:
  {
  (0,0), (0,1), (0,2), …
  (1,0), (1,1), (1,2), …
  (2,0), (2,1), (2,2), …
  …
  }
  I can enumerate the set like this:



I.e.
I list all the pairs (i,j) such that i+j = 0. There is 1 of these pairs.
Then, I list all the pairs (i,j) such that i+j = 1. There are 2 of these pairs.
Then, I list all the pairs (i,j) such that i+j = 2. There are 3 of these pairs.
We continue this method for all elements in this set.
To find the specific index of an element (i,j) in this set, we can do the following:



Note that this equation is for if we visit the diagonal where the sums of the pairs are equal and we start from the top row and move diagonally left, as shown below:
I.e.



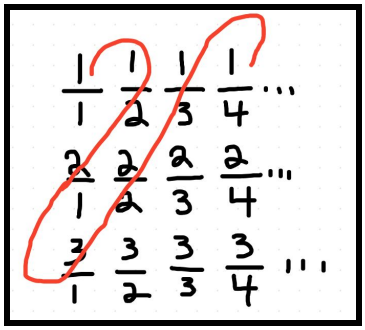
E.g.

1. Take (0,0)
   i = 0, j = 0
   ((0+0)(0+0+1)/2) + 0 = 0
2. Take (0,1)
   i = 0, j = 1
   ((0+1)(0+1+1)/2) + 0 = 1
3. Take (1,0)
   i = 1, j = 0
   ((1+0)(1+0+1)/2) + 1 = 2

This is called the **pairing function**. It takes 2 arguments, which are natural numbers, and returns the position of the 2 arguments. It encodes pairs of natural numbers by one natural number.

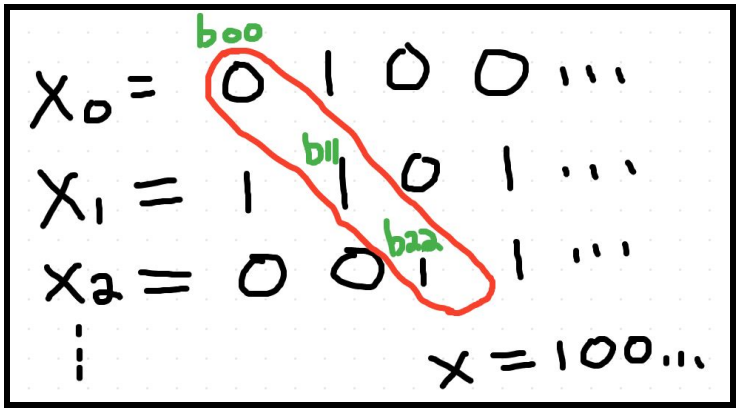
- **Theorem 1.2.1:** The set of positive rational numbers is countable. Note that this set is a **dense set**, meaning that for any two distinct elements in the set, you can find another element in between the pair. To prove that this set is countable, we will use the snaking method shown in theorem 1.2.

I.e.



- **Theorem 1.2.2:** N x N x N is countable, too. To prove this, think of the element (i,j,k) as ((i,j),k). Then, using the technique from theorem 1.2, we can prove that N x N x N is countable. Hence, this method encodes triples of natural numbers by one number.
- **Theorem 1.3:** $\forall k \in N, N^k = N * N \ldots * N$ is countable. We can use induction to prove this.
- **Theorem 1.3.1:** The set of reals in (0, 1] is not countable. To prove this, we will use the below theorem:

Theorem 1.4: The set B^∞ of infinite binary strings is not countable.
To prove this, assume for contradiction that B^∞ is countable.
Let X0, X1, X2, … be any enumeration of B^∞.
Let Bij be the j^{th} bit of Xi.
Next, we will construct an infinite binary string, x, whose i^{th} bit is $\neg$Bii, the complement of Bii.
Note that x is not in any enumeration of B^∞ because it differs from each enumeration by one bit. x differs from Xi in at least the i^{th} position, so $x \neq Xi$ for all $i \in N$. Therefore, X0, X1, …, is not an enumeration of B^∞, so B^∞ is not countable.
E.g.

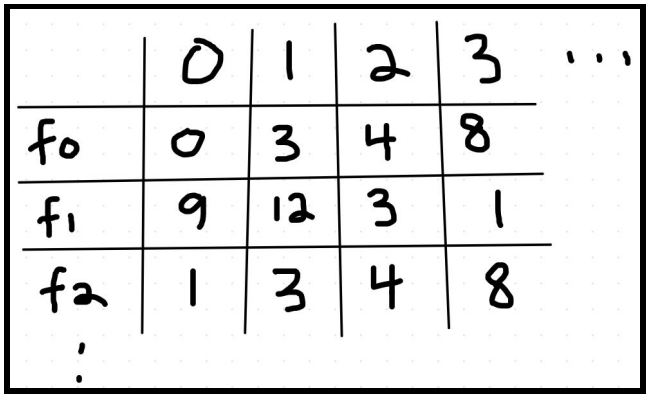


We have our enumerations, X0, X1, X2, …. Next, we take the i^{th} bit of Xi, which is Bii, and get the complement of that to construct x. In the example above, B00 = 0, B11 = 1, B22 = 1. Hence, we have 011. The complement of 011 is 100, which is what x is set to. Since x differs from Xi in at least the i^{th} position, X0, X1, …, is not an enumeration of B^∞, and as such, B^∞ is not countable.
The above method is known as the **Diagonalization Method**.

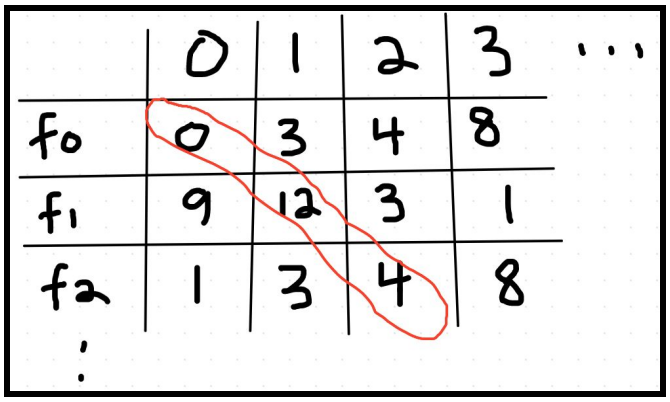
The proof for theorem 1.3.1 is similar.
Let R0, R1, R2, … be any enumeration of the set of real numbers in (0,1].
Let Bij be the j^{th} digit of Ri.
Next, we will construct an infinite number, x, whose i^{th} digit is any digit other than Bii. If Bii is 1, then the i^{th} digit of x can be any digit other than 1.
Then, we have created a real number between (0,1] that is not in any of the enumerations.
Therefore, the set of real numbers between (0,1] is uncountable.

- **Theorem 1.5:** The set of functions from N to N is uncountable. We can prove this using proof by contradiction.

By the diagonal method, suppose that it is countable.
Let f0, f1, f2, … be the functions.
Consider the picture below. It is a chart of the enumerations of the set. On the top row, we have the natural numbers and on the left column, we have the functions. In each cell, we have the result of the given function on the given natural number.
I.e. f0(0) = 0, f1(3) = 1, etc



Next, consider the diagonal shown below. The diagonal gives us the i^{th} value of the i^{th} function.



We will create a new function, f, such that f(i) ≠ fi(i).
I.e. f(0) gives any number other than f0(0) or 0.
f does not appear in any of the enumerations. Hence, the set of functions from N to N is uncountable.

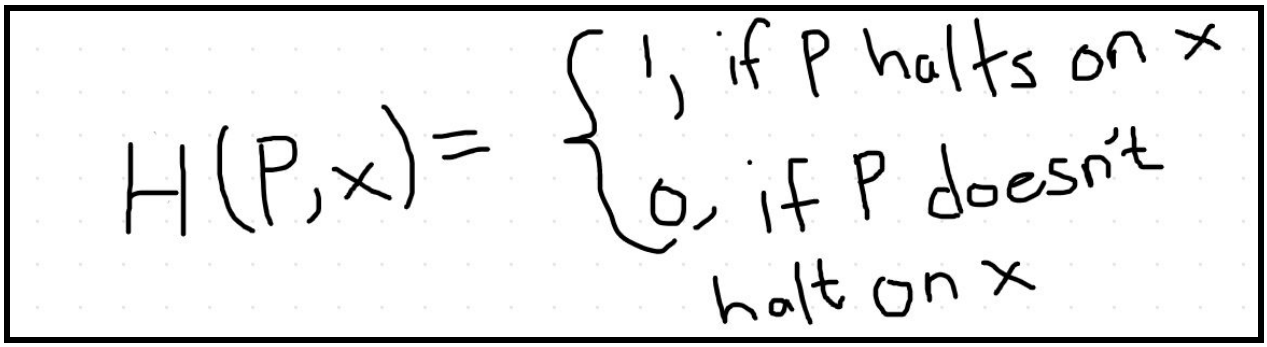
- **Theorem 1.6:** Let Γ be a (finite) alphabet, |Γ| ≥ 2. An **alphabet** is just a set of symbols. Furthermore, for our purposes, alphabets are always finite. Γ^* is the set of finite strings

over Γ. Γ^* is countable. To prove this, enumerate all strings over Γ of length 0, then length 1, then length 2, and so on. For each enumeration, we will enumerate in lexicographical order.

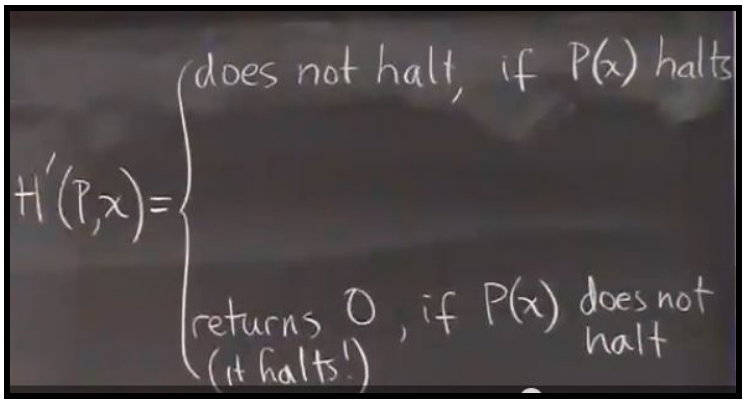
Uncomputability:

- Computers compute functions N → N. However, there are some functions N → N that computers cannot compute.
  One reason for this is because there is an uncountable number of functions from N → N but there are a countable number of programs to compute them. This is called the **counting argument** and is a cheap argument.
- **Turing's Halting Function:** Let's call the function h. h takes 2 arguments, P and x, and returns a single bit. It outputs 1 if program P halts on x. It outputs 0 if P does not halt on x.
  I.e.

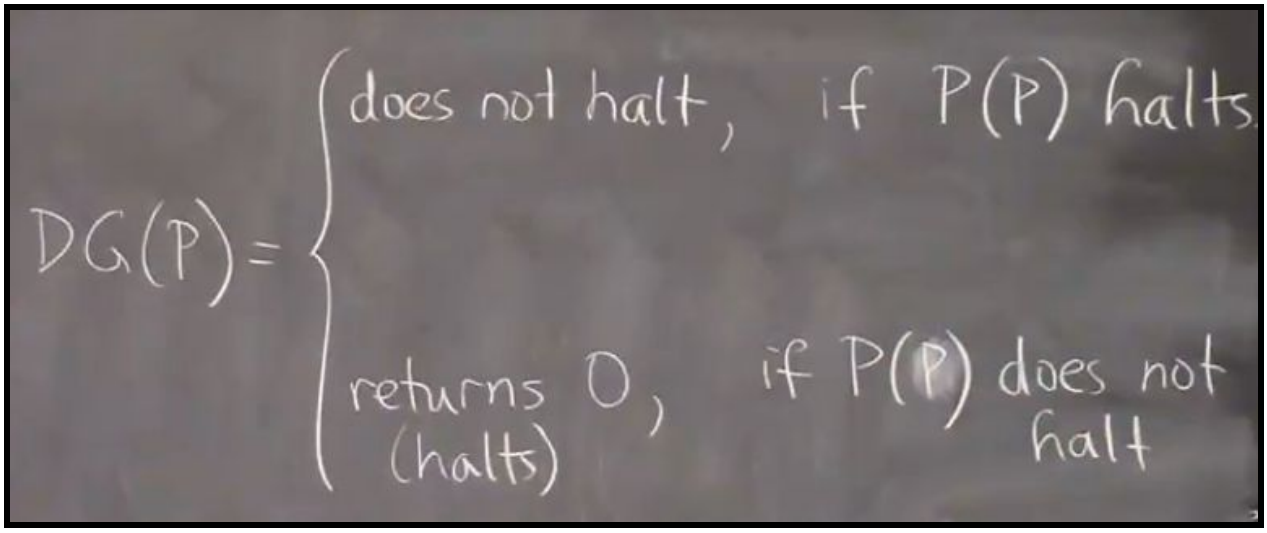
Fact: No program can compute h.
Proof:
We will prove the above fact using a proof by contradiction.
Suppose, for contradiction, that such a program exists. We'll call it H.



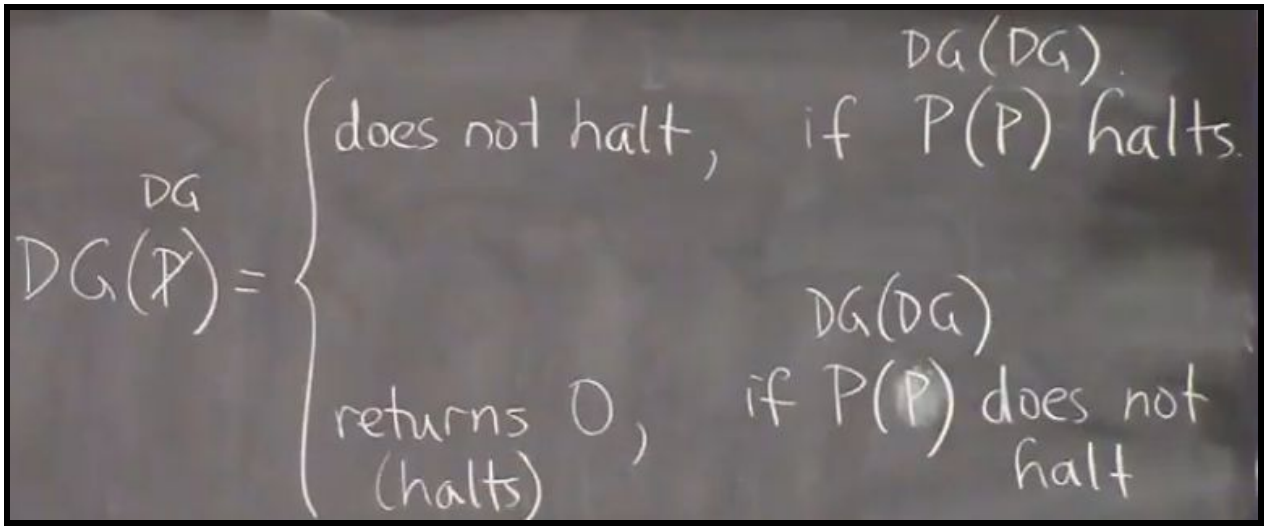
I will modify H to obtain a new program, H'. I will replace every "return b" statement in H, where b is either 1 or 0, by the following:
"if b==1 then:
    while (true): ← Notice the infinite loop here.
        x = x
else:
    return 0"



Now, I will write a new program called DG(P).
DG(P):
    return H'(P, P)



Since DG itself is a program, we can do DG(DG).



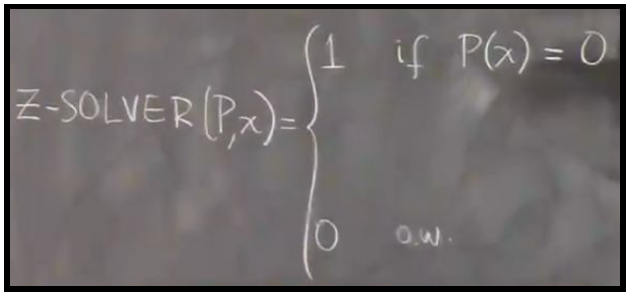
Hence, DG(DG) halts iff DG(DG) doesn't halt. This is a contradiction. Hence, h(P,x) is uncomputable.

Note: The function is called the **halting function**, but finding a program that computes the function is called the **halting problem**.

Reduction:

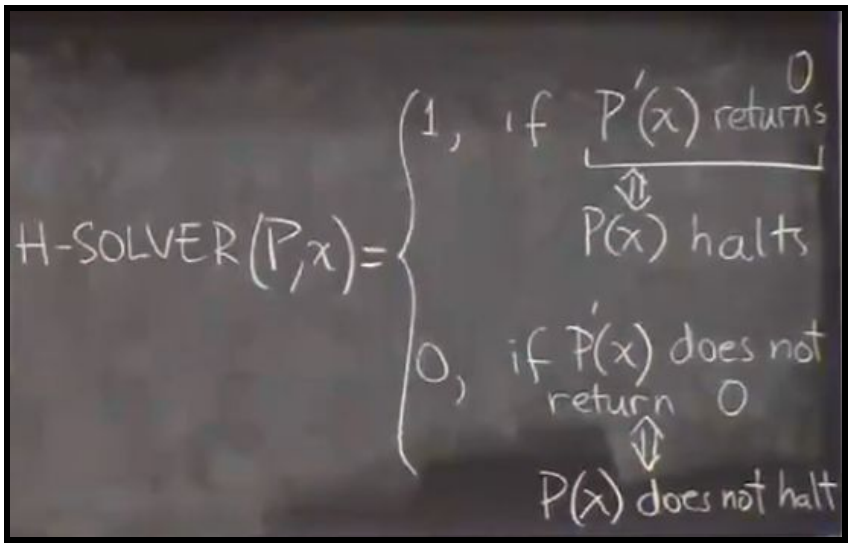
- A problem X reduces to problem Y, denoted by X ≤ Y, if we can use a given or assumed solution to Y to solve X.
- If X reduces to Y, then X is no harder than Y. This is what the ≤ sign is intended to capture. X cannot be harder than Y because if we can solve Y, we can solve X. Another way of thinking about this is that Y is at least as hard as X.
- This concept can be used in 2 ways:
    1. Positive Use: If X reduces to Y, and somebody already solved Y, I can use it to solve X.
    2. Negative Use: Suppose I have reduced X to Y. If I know that X is uncomputable, then it follows that Y is also uncomputable. I.e. This way proves that some things are hard or unsolvable. This is the way we will be using reduction for the course.
- E.g. There are 2 problems, Zero and Halt, defined below.
  Zero: Given a program P and input x, determine if P(x) returns 0.
  Halt: Is the halting problem.
  Halt ≤ Zero. Since we cannot compute Halt and Halt is no harder than Zero, we cannot compute Zero.

Proof that Halt reduces to Zero:
Suppose that Z-Solver is a program that solves Zero.



We will use Z-Solver to write a program, H-Solver, that solves the Halting Problem. I can reduce Halt to Zero by showing that Z-Solver is used to solve Halt.

Now, we will create H-Solver. It takes 2 inputs, program P and x.
In H-Solver, there is another function called P'. P' is a program obtained from P by changing every "return ___" statement to "return 0". H-Solver returns Z-Solver(P',x)



By using Z-Solver in H-Solver, it shows that Halt reduces to Zero.
Therefore, H-Solver solves the halting problem, which is impossible. Since Halt is impossible, Zero is also impossible.

Textbook Notes:

- Georg Cantor discovered the **diagonalization** technique in 1873.
- Cantor was concerned with the problem of measuring the sizes of infinite sets. If we have two infinite sets, how can we tell whether one is larger than the other or whether they are of the same size? For finite sets, of course, answering these questions is easy. We simply count the elements in a finite set, and the resulting number is its size. But if we try to count the elements of an infinite set, we will never finish! So we can't use the counting method to determine the relative sizes of infinite sets.
- For example, take the set of even integers and the set of all strings over {0,1}. Both sets are infinite and thus larger than any finite set, but is one of the two larger than the other? How can we compare their relative size?
- Cantor proposed a rather nice solution to this problem. He observed that two finite sets have the same size if the elements of one set can be paired with the elements of the

other set. This method compares the sizes without resorting to counting. We can extend this idea to infinite sets. Here it is more precisely:
Assume that we have sets A and B and a function f from A to B. f is **one-to-one** if it never maps two different elements to the same place—that is, if f(a) ≠ f(b) whenever a ≠ b. f is **onto** if it hits every element of B—that is, if for every b ∈ B there is an a ∈ A such that f(a) = b. A and B have the same size if there is a one-to-one and onto function f : A→B. A function that is both one-to-one and onto is called a **correspondence**. In a correspondence, every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.

Note: Alternative common terminology for these types of functions is **injective** for **one-to-one**, **surjective** for **onto**, and **bijective** for **one-to-one** and **onto**.

- E.g.
  Let N be the set of natural numbers {1, 2, 3,...} and let E be the set of even natural numbers {2, 4, 6,...}. Using Cantor's definition of size, we can see that N and E have the same size. The correspondence f mapping N to E is simply f(n)=2n. This is shown in the table below.

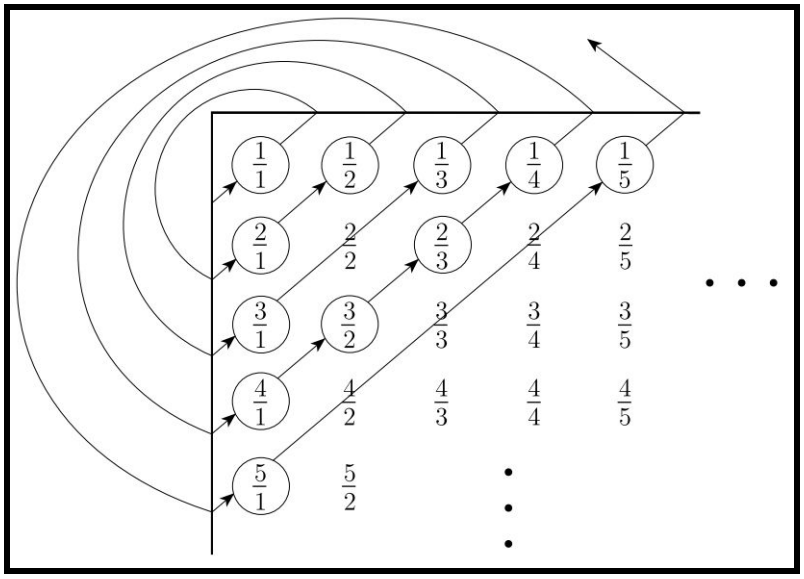
n	$f(n)$
1	2
2	4
3	6
⋮	⋮

Of course, this example seems bizarre. Intuitively, E seems smaller than N because E is a proper subset of N . But pairing each member of N with its own member of E is possible, so we declare these two sets to be the same size.

- **Definition:** A set "A" is **countable** if either it is finite or it has the same size as N.
- E.g.
  If we let Q = { $\frac{m}{n}$ | m, n ∈ N} be the set of positive rational numbers, Q seems to be much larger than N . Yet these two sets are the same size according to our definition. We give a correspondence with N to show that Q is countable. One easy way to do so is to list all the elements of Q. Then we pair the first element on the list with the number 1 from N , the second element on the list with the number 2 from N , and so on. We must ensure that every member of Q appears only once on the list. To get this list, we make an infinite matrix containing all the positive rational numbers. The ith row contains all numbers with numerator i and the jth column has all numbers with denominator j. So the number

$\frac{i}{j}$ occurs in the ith row and jth column.

Now we turn this matrix into a list. To list it, we list the elements on the diagonals, starting from the corner. The first diagonal contains the single element 1/1, and the second diagonal contains the two elements 2/1 and 1/2. So the first three elements on the list are 1/1, 2/1, and 1/2. In the third diagonal, a complication arises. It contains 3/1, 2/2, and 1/3. If we simply added these to the list, we would repeat 1/1 = 2/2. We avoid doing so by skipping an element when it would cause a repetition, so we add only the two new elements 3/1 and 1/3 . Continuing in this way, we obtain a list of all the elements of Q. This is shown in the picture below.



- **Definition:** Some infinite sets have no correspondence with N. Such sets are called **uncountable**.

  The set of real numbers is an example of an uncountable set. This will be proved below.
- **Theorem:** R is uncountable.

  Proof:

  In order to show that R is uncountable, we show that no correspondence exists between N and R. The proof is by contradiction. Suppose that a correspondence f existed between N and R. Our job is to show that f fails to work as it should. For it to be a correspondence, f must pair all the members of N with all the members of R. But we will find an x in R that is not paired with anything in N , which will be our contradiction.

  The way we find this x is by actually constructing it. We choose each digit of x to make x different from one of the real numbers that is paired with an element of N. In the end, we are sure that x is different from any real number that is paired. To do this, we will enumerate through the set R.

  Suppose that:

  R0 = 3.1415…

  R1 = 5.55555…

  R2 = 0.12345…

  R3 = 0.50000…

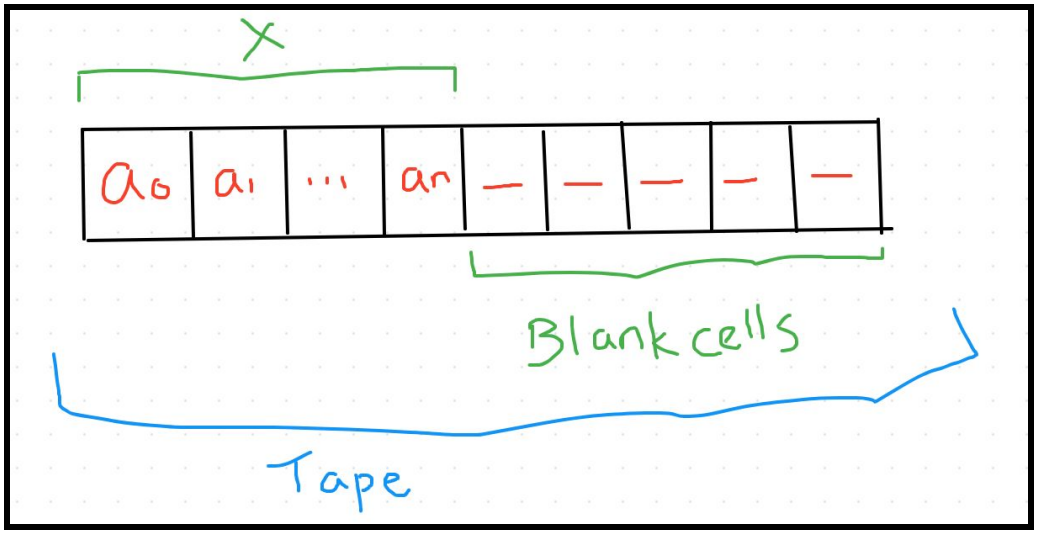
  We construct the desired x by giving its decimal representation. It is a number between 0

and 1, so all its significant digits are fractional digits following the decimal point. Hence, to begin constructing x, we will only look at the digits after the decimal in each of Ri. We will take the first digit after the decimal in R0, the second digit after the decimal in R1, and so on. In general, we will take the i+1 digit after the decimal in Ri.
In our example, for now, x = 0.1530...
Then, to make sure that x ≠ Ri, we will change all of x's digits after the decimal place. We don't care what the new digit is, as long as it is not the same as the original digit.
I.e. In our example, now x = 0.3421….
Since x differs by each Ri by at least 1 digit, x could not have appeared in the enumeration. Hence, R is uncountable.

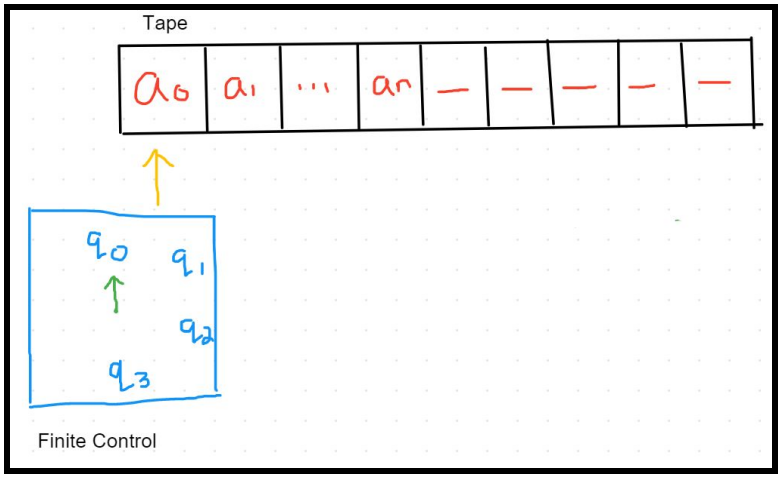
An Informal Description of Turing Machines:

- A **turing machine (TM)** is a mathematical model for what it means to perform a **mechanical computation**. **Mechanical** means that given the representation of an input, we want to produce a representation of the appropriate output by following a finite sequence of steps, each of which can be obviously carried out. **Computation** means that given an input of some sort, we want to produce an appropriate output for that input. **Note:** There could be multiple appropriate outputs for the given input.
- Examples of computations:
    a. Input: $n \in N$
       Output: n^2
       In this example, each output is unique.
    b. Input: An undirected, connected, unweighted Graph, G
       Output: A minimal spanning tree (MST) of G
       In this example, there could be multiple appropriate outputs.
    c. Input: $n \in N$
       Output: "Yes" if n is a prime number, and "No" otherwise
- When we're performing a computation, we're trying to solve a **problem**, which is an input-output relation.
- An **instance of a problem** is simply a particular input of that problem.
- A **decision problem** is a problem where the output is True/Yes or False/No.
- We can call an **instance of a decision problem** as either a **Yes-instance** or a **No-instance** depending on output for that instance.
- A **formal language** is a set of strings over some alphabet Σ.
- The subset of strings for which the problem returns Yes is a formal language, and often decision problems are defined as formal languages.
  I.e. A decision problem is a set of representations of their Yes-instances.
- E.g.
  Suppose we represent natural numbers, n, in binary. The problem is "Is n prime?". The input is a natural number and the output is Yes/No. I can think of the Yes-instances to this problem as a language.
  I.e. Lp = {10, 11, 101, 111, …} These binary strings represent the prime numbers and I can think of the problem "Is n prime" as "Does the binary representation of the number belong to Lp?"
- A Turing Machine has a 1-way infinite tape divided into cells and it consists of a finite control, which is a box that can be in a finite number of states. This finite control allows the Turing Machine to read what's on the tape and to modify what's on the tape. Each cell of the tape contains 1 symbol from a finite alphabet, including a special blank symbol. At any point in time, the finite control scans exactly one cell of the tape. This is what the Turing Machine does:
  Given the present state and the symbol on the tape, it may:
    1. Change its state and then move one cell left or one cell right.
    2. Change the symbol on the present cell and then move one cell left or one cell right.

  Note: For our purposes, the tape is infinitely long to the right. If we are at the leftmost cell, and we want to move left, we just stay in that cell.
- We start the TM with some input, x, shown below, which is a sequence of finite strings. Every other cell in the tape is a blank cell.



The input string, x, is not allowed to contain any blank cells. This is because for the TM to determine the end of the input, it starts from the leftmost cell and goes to the right until it hits a blank cell. If the input itself contains a blank cell, then the TM would never know when the input ends.
Furthermore, suppose the finite control is in state Q0.
I.e. This is what the TM looks like at the start.



Given its current state and the input, the TM either changes state or changes the input and it moves to the left or the right. The TM keeps executing these steps until the state that it's in becomes one of two special states, **Qa** or **Qr**.
Qa is the **accept state**.
Qr is the **reject state**.
Then, the TM stops and either accepts the input string, if it's in Qa, or rejects the input string, if it's in Qr.
Note: There is nothing requiring the TM to reach either Qa or Qr. It's possible that the TM will keep on going and never entering either Qa or Qr. In this case, we say that the **TM loops on the input**.
Note: Not accepting the input is not the same as rejecting the input.
We can define the language accepted by the TM as the set of input strings that lead to Qa.

A Formal Description of Turing Machines:

- A TM M is a 7-tuple, with M = (Q, Σ, Γ, δ, Q0, Qa, Qr) where
  Q is a finite set of states.
  Σ is a finite input alphabet. It does not include the blank symbol.
  Γ is another finite alphabet, which is the set of symbols that can be written on the tape.
  Note: Γ includes symbols from Σ, but may also include other symbols, such as the blank symbol. Γ is a superset of Σ. Γ is called the **tape alphabet**.
  δ is the **transition function** that tells the automation how to move. It defines the moves of the TM.
  $\delta : (Q - \{Qa, Qr\} \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R\})$
  I.e. δ takes a state that is not Qa or Qr and a tape symbol as input and outputs a new state, which may be the same as the old state, the new symbol on the tape head, which may be the same as the old symbol, and the direction in which the tape head should move (left or right).
  E.g.
  $\delta(q, a) \rightarrow \delta(p, b, L/R)$
  Given δ(q, a) where q is a state that is not Qa nor Qr, and a tape symbol "a", it returns a new state, which may be the same as the old state, a new tape symbol for the tape head, which may be the same as the old tape symbol, and a direction to move, left or right.
  If M is in state "q" and its state head is scanning the tape cell with symbol "a", then, M enters state "p" and writes "b" on the tape cell that is being scanned, and moves the tape head one cell to the left or right.
  Q0 is the initial state. $Q0 \in Q$.
  Qa is the accept state. $Qa \in Q$.
  Qr is the reject state. $Qr \in Q-\{Qa\}$.
- **Configuration/Instantaneous Description (ID):** The configuration or ID of a TM is a snapshot of the TM to describe the current situation of the TM. It provides the following information:
  - Current State
  - Contents of tape, except the trailing blanks
  - Position of the tape head

  Given the above 3 pieces of information, we know exactly what will happen next.
  I.e. We will know what will be the next state, the next contents of the tape, and next position of the tape head.

  Formally, a configuration is a string of the form "xqy" where
  $x, y \in \Gamma^*$, s.t. y does not end with a blank, and $q \in Q$.
  x is a string of tape symbols and y is a string of tape symbols not ending in blank symbols.
  q represents the current state.
  xy are the contents of the tape.
  The tape head is positioned over the first cell of string y, or if y is the empty string, then the tape head is positioned over the first of the infinitely many blank strings.

- Consider the relation, $\vdash_M$, between configurations of the TM M. Let C and C' be configurations.

  Informally, think of C $\vdash_M$ C' as "From configuration C, TM M moves to configuration C' in 1 step.

  C $\vdash_M$ C' is read as "C yields C'."
  Formally, suppose that C = xqy.
  If y = ay' (I.e. y is not empty and starts with symbol a), and δ(q,a) = (p, b, R), then

  C $\vdash_M$ C' iff C' = xbpy'.
  If y = ay and x = x'c (I.e. x is not empty and ends with symbol c), and δ(q,a) = (p, b, L), then

  C $\vdash_M$ C' iff C' = x'pcby.
  If y = ay' and x = ε (I.e. x is empty), and δ(q,a) = (p, b, L), then

  C $\vdash_M$ C' iff C' = pby'.

- We define $\vdash_M^*$ to be the transitive closure of $\vdash_M$. This means that C $\vdash_M^*$ C' iff
  - C' = C
  - There is a finite sequence of configurations C1, C2, …, Ck, such that C1 = C, Ck = C' and Ci $\vdash_M$ Ci+1 for all i ∈ $1 \leq i \leq k-1$.

- A TM M accepts string x, $x \in \Sigma^*$, iff Q0x $\vdash_M$ yQaz, where Q0x is the initial configuration and yQaz is the accepting configuration.

  A TM M rejects string x, $x \in \Sigma^*$, iff Q0x $\vdash_M$ yQrz, where Q0x is the initial configuration and yQrz is the rejecting configuration.

  If either case happens, then we say that "M halts on x".
  If M doesn't halt on x, then we say that "M loops on x".
  M loops on x if there is an infinite number of configurations C1, C2, …, such that Q0x $\vdash_M$ C1 $\vdash_M$ C2… . This is like an infinite loop.
- The language L(M) **accepted/recognized** by TM M is defined as:
  $L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$.

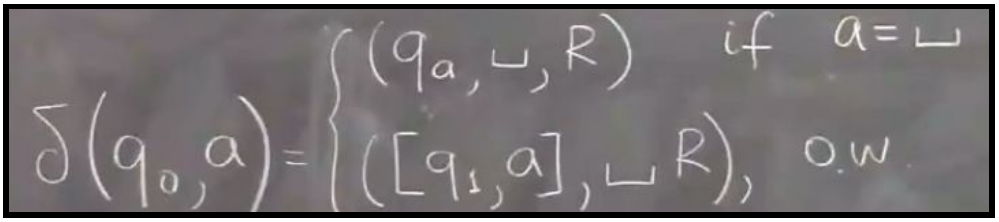
- **Note:** If M does not accept x, $x \in \Sigma^*$, this does not mean that M rejects x.
  If M does not accept x, there are 2 possibilities:
  1. M rejects x. OR
  2. M loops on x.
- Given a TM M and a string x, $x \in \Sigma^*$, there are 3 possibilities:
  1. M accepts x.
  2. M rejects x.
  3. M loops on x.
- A language L is **recognizable/semi-decidable/recursively enumerable** iff there exists a TM M such that L(M) = L. Here, M is called a **recognizer**.
- A language L is **decidable/recursive** iff there exists a TM M such that L(M) = L and M halts on every input. Here, M is called a **decider**.
- If L is a decider, then by definition, it is also a recognizer.
- Since a language is simply a set of strings, we will also say that a set is decidable/recognizable.
- Since a language is simply another way of thinking of a decision problem, we will also say that a decision problem is decidable/recognizable.
- We will create a TM that decides the language L, where
  $L = \{x \in \Sigma^* \mid x \text{ is an even length palindrome}\}$
  $\Sigma = \{0, 1, 2\}$

  E.g. ε, 00, 0220 $\in$ L

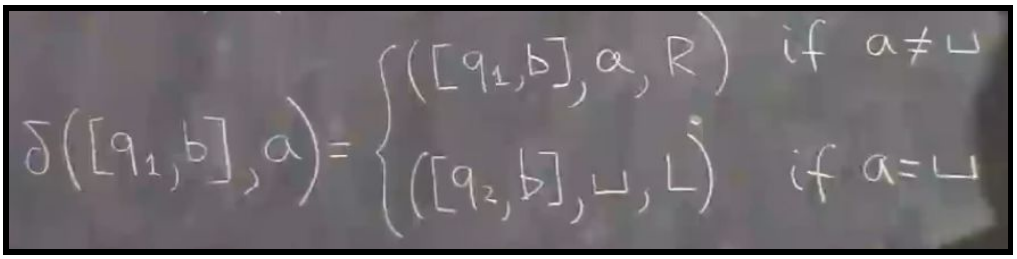
  This is a high level description of our TM:
  1. If the symbol under the tape head is the blank symbol, then accept it.
     Otherwise, “remember” that symbol, replace it with a blank and move 1 cell to the right.
  2. While the symbol scanned is not a blank symbol, move right.
  3. Move one cell to the left.
  4. If the symbol under the tape head is not the same as the one “remembered”, then reject it.
     Otherwise, replace it with a blank symbol and move 1 cell left.
  5. While the symbol scanned is not a blank symbol, move left.
  6. Move one cell to the right and go to stage 1.

  This is a formal description of our TM:
  - States of M:
    - Q0: Initial State
    - Qa/Qr: Accept/Reject State
    - [Q1, a] $\forall a \in \Sigma$: This means, keep moving right and the first symbol is a.
    - [Q2, a] $\forall a \in \Sigma$: This means, I have reached the right end and the symbol remembered is a.
    - Q3: Keep moving left.

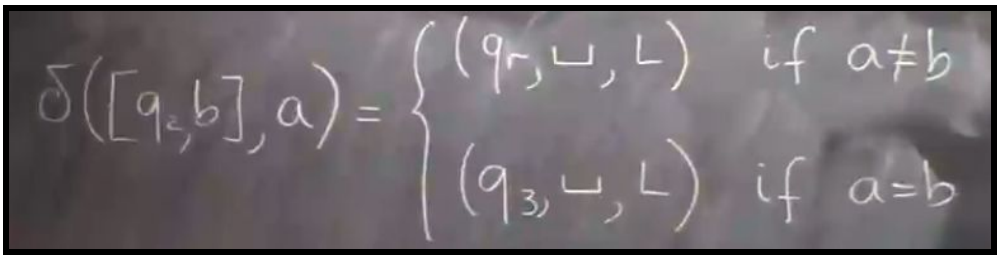
    Hence, $Q = \{Q0, Qa, Qr, Q3\} \cup \{[Q1, a] \mid a \in \Sigma\} \cup \{[Q2, a] \mid a \in \Sigma\}$
  - $\Sigma = \{0, 1, 2\}$
  - $\Gamma = \{0, 1, 2, \sqcup\}$

- 

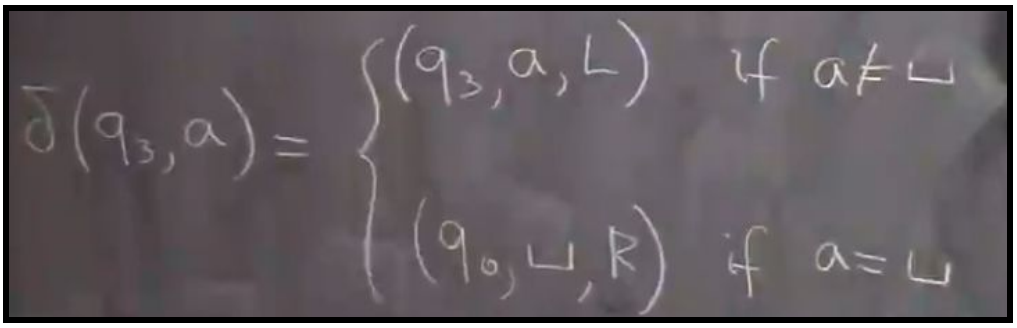
  I.e. We start at the initial state, Q0, and our symbol is a. If a is the blank symbol, then we enter the accept state, Qa. Otherwise, we enter state Q1, remember a, and replace it with a blank symbol and move 1 cell to the right.

- 

  I.e. If a is not the blank symbol, we just keep moving right. If a is the blank symbol, we enter state Q2 and move one cell left.

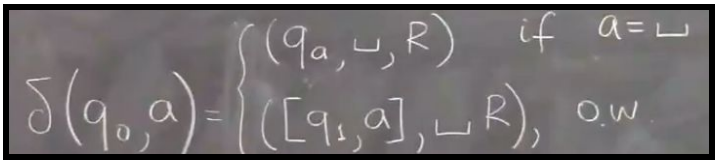
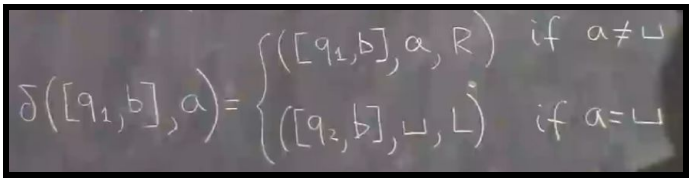
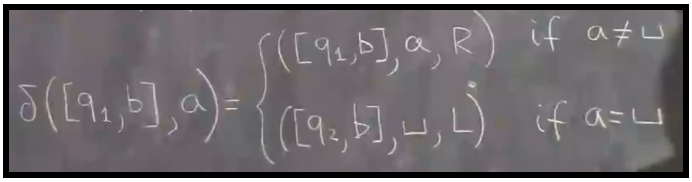
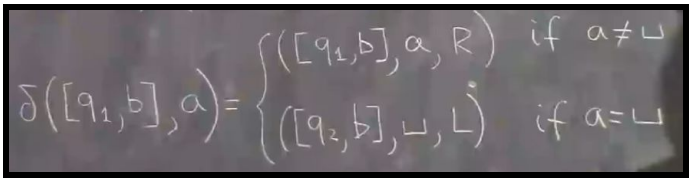
- 

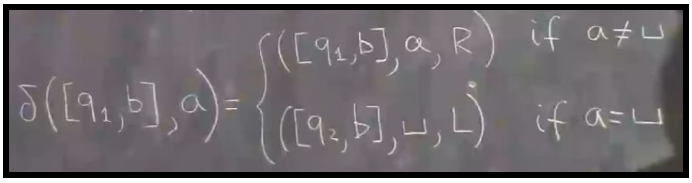
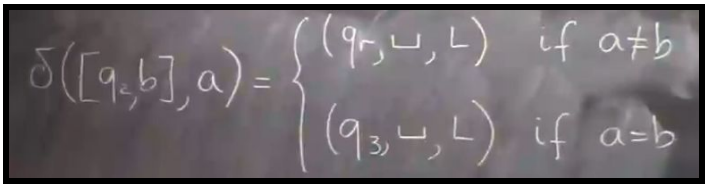
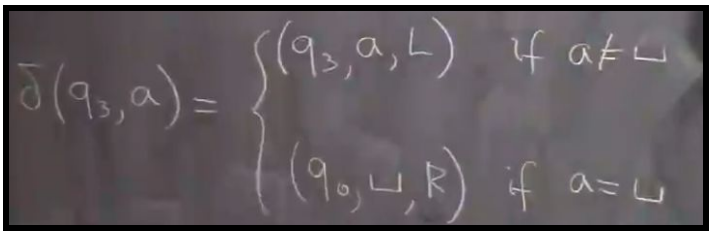
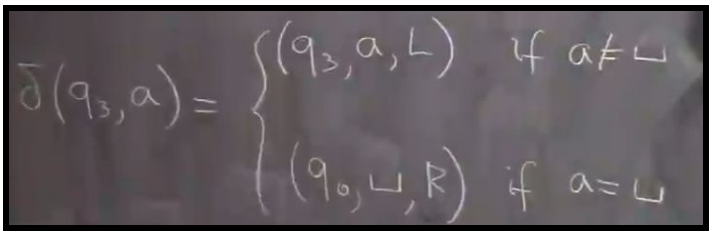
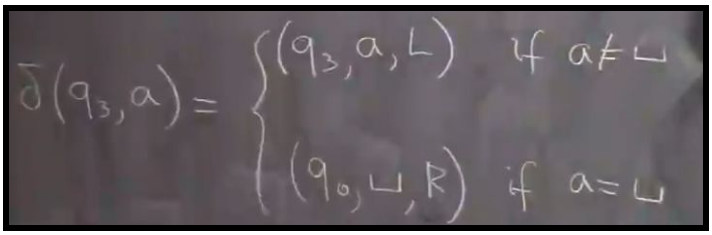
  I.e. If a is not equal to b, then we enter the reject state, Qr. Otherwise, we enter state Q3, replace a with the blank symbol, and move left.

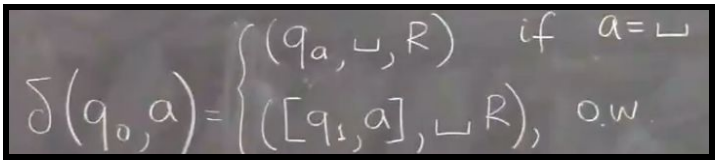
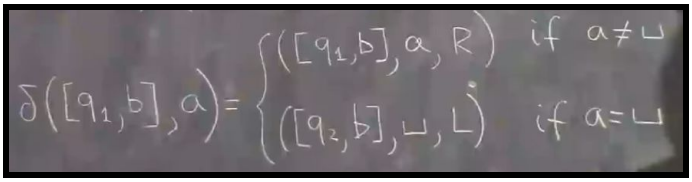
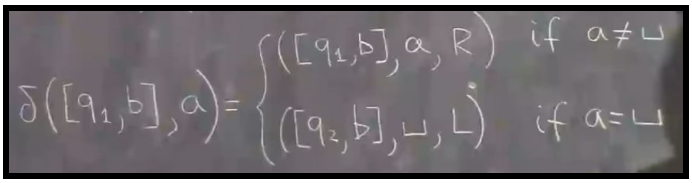
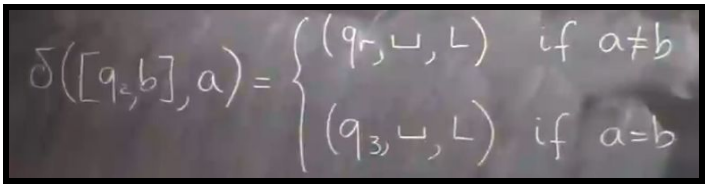
- 

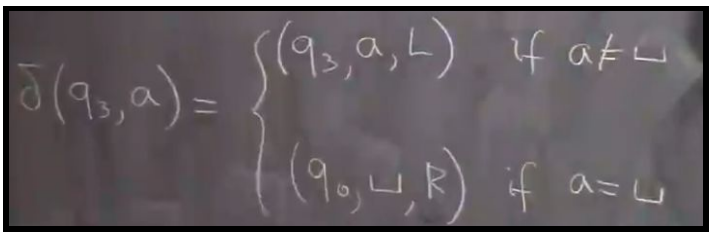
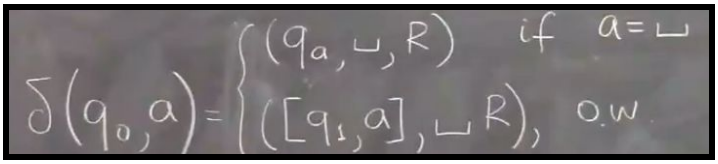
  I.e. If a is not the blank symbol, then we keep moving left. If a is the blank symbol, we enter state Q0 and move 1 cell to the right.

Example of our TM on the string 0110:
Note: To describe what the TM does on the given string, I will use a series of configurations shown below:

Configurations	**Description**
Q00110␣	We're starting at the initial configuration, Q0. Based on the transition function shown below  since a = 0, we enter state [Q1,0], replace a with ␣ and move 1 cell to the right.
␣**[Q1,0]**110␣	We're now in state [Q1,0]. Based on the transition function shown below  since a = 1, we just move 1 cell to the right.
␣1**[Q1,0]**10␣	We're still in state [Q1,0]. Based on the transition function shown below  since a = 1, we just move 1 cell to the right.
␣11**[Q1,0]**0␣	We're still in state [Q1,0]. Based on the transition function shown below  since a = 0, we just move 1 cell to the right.
␣110**[Q1,0]**␣	We're still in state [Q1,0]. Based on the transition function shown below

	 since a = ␣, we enter state [Q2, 0] and move 1 cell left.
␣11**[Q2,0]**0␣	We're now in state [Q2,0]. Based on the transition function shown below  since a = b (0 = 0), we enter state Q3, replace a with ␣ and move 1 cell left.
␣1**Q3**1␣	We're now in state Q3. Based on the transition function shown below  since a = 1, we just move 1 cell left.
␣**Q3**11␣	We're still in state Q3. Based on the transition function shown below  since a = 1, we just move 1 cell left.
Q3␣11␣	We're still in state Q3. Based on the transition function shown below 

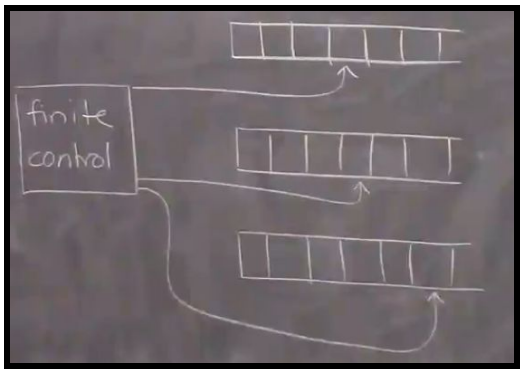
	since a = ⊔, we enter state Q0 and move 1 cell right.
⊔**Q0**11⊔	We're now in state Q0. Based on the transition function shown below  since a = 1, we enter state [Q1,1], replace a with ⊔ and move 1 cell to the right.
⊔⊔**[Q1,1]**1⊔	We're now in state [Q1,1]. Based on the transition function shown below  since a = 1, we just move 1 cell to the right.
⊔⊔1**[Q1,1]**⊔	We're still in state [Q1,1]. Based on the transition function shown below  since a = ⊔, we enter state [Q2, 1] and move 1 cell left
⊔⊔**[Q2,1]**1⊔	We're now in state [Q2,1]. Based on the transition function shown below  since a = b (1 = 1), we enter state Q3, replace a with ⊔ and move 1 cell left.
⊔**Q3**⊔	We're now in state Q3. Based on the transition function shown below

	 since a = ␣, we enter state Q0 and move 1 cell right.
␣␣**Q0**␣	We're now in state Q0. Based on the transition function shown below  since a = ␣, we enter the state Qaccept.
␣␣␣**Qaccept**	We're now in state Qaccept.

Note: The bolded parts are the states.

Variants of TMs:

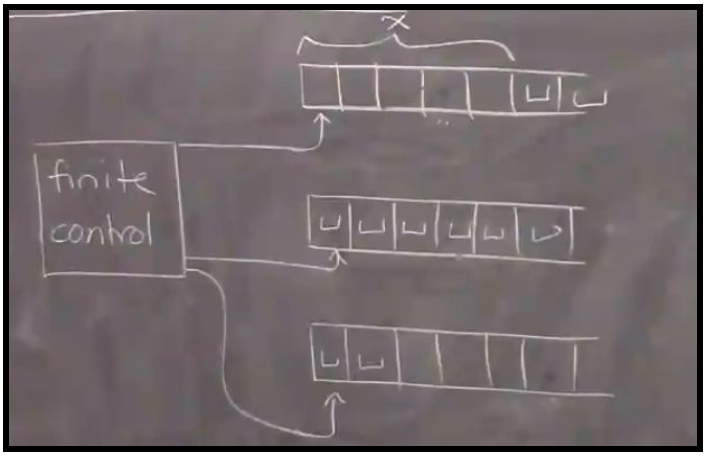
- **Multi-tape TM:**
- Looks like this:



There is one control machine and there are k tapes. Each of these k tapes has its own tape head.

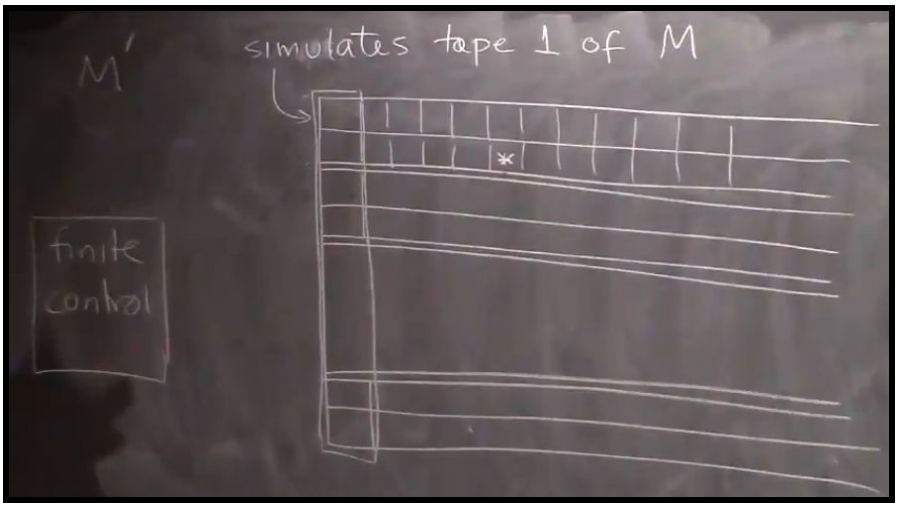
- This is how multi-tape TMs work:
  - Each tapehead starts at the leftmost cell of their respective tapes.
  - The input, X, is at the leftmost cell(s) of the first tape. Every other cells, including cells in the other tapes, are blanks.

I.e.



- The control machine is at some state and it scans each of the k tapes.
- Then, based on the machine's state and what it is currently scanning, it performs an action on that tape.
- Hence, each action is independent of other tapes.

- **Theorem 2.1:** If M is a multi-tape (k-tape) TM, there exists a basic TM M' such that L(M') = L(M).
  I.e. I can simulate a multi-tape TM with a basic TM.
  Furthermore, if M accepts/rejects input x in m moves, then M' accepts/rejects x in $O(m^2)$ moves.
  Proof that L(M') = L(M):
  - This is what M' will look like:

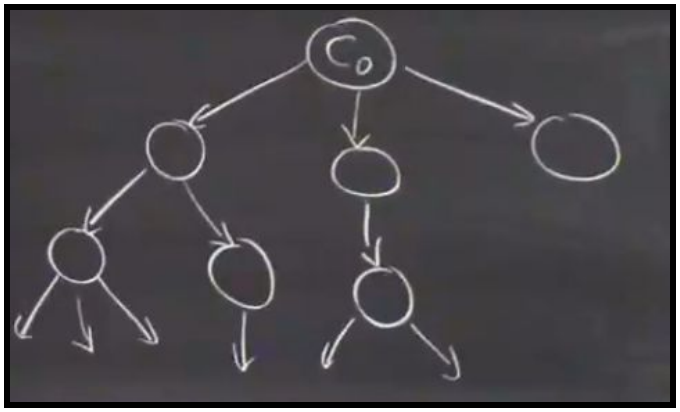


    It will have one tape, but divided up into 2k tracks. It will have 2 tracks for each tape in M. In other words, every 2 tracks in M' corresponds to 1 tape in M.
    The first track in every pair of tracks will simulate the corresponding tape of M.
    The second track in every pair of tracks is blank everywhere, except for in 1 square, where it has a star. That star represents the location of the tape head of the corresponding tape in M.
  - Remember M's state.
  - Head starts from the leftmost cell of M'.

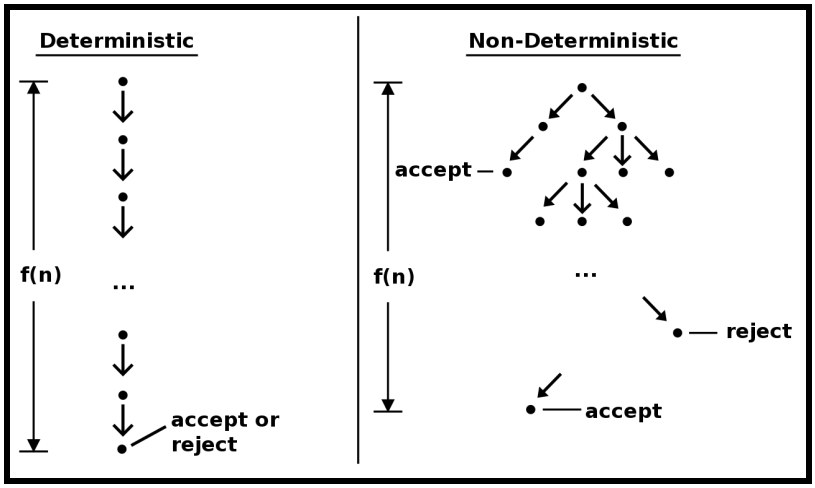
- Scan the multitrack tape of M' to the right, remembering in the state of M' all the symbols corresponding to the stars until you have seen all k stars.
- Using the state transition function of M, we determine the symbols to be written on the cell of each tape under its head. Remember them in our state.
- M' scans its multitrack (single) tape to the left performing the appropriate action at each star. M' accepts/rejects if M does.

Proof that M' takes $O(m^2)$ moves:
If M takes m moves, its heads are at most m cells apart.
The right moves of M' to simulate one move of M takes at most m steps.
The left moves of M' to simulate one move of M takes at most (m + 2k) steps. It takes m+2k steps because we might need to move 1 cell right before moving m+1 cells to the left. That's 2 extra steps. Since there are k tapes in M, at most, there would be 2k extra steps. Hence, we get m+2k steps.
Hence, one move of M takes (m) + (m+2k) or 2(m+k) moves of M'. Therefore, m moves of M takes $O(m^2)$ moves of M'.

- **<u>Non-deterministic TM (NTM):</u>**
- In a regular TM, the transition function is defined as $\delta(q, a) \rightarrow \delta(p, b, L/R)$.
  In a NTM, the transition function is defined as $\delta(q, a) \rightarrow \delta\{(p, b, L/R)\}$.
- In a regular TM, the configurations were in a linear structure.
  I.e. $C0 \vdash C1 \vdash \ldots$
  With NTMs, the configurations are in a tree-like structure.



Here's a side by side comparison of configurations for TM and NTM.

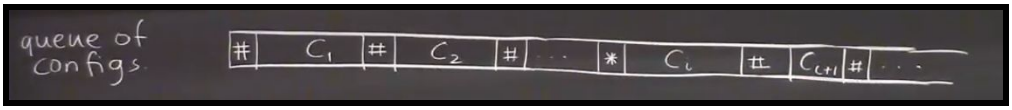


- A NTM M accepts input x iff there exists a computation path in the computation tree of M on x ending in a configuration in the accept state.
  I.e. As long as there exists one computation path that leads to the accepting state, we say that the NTM accepts the input.
- **Theorem 3.1:** A language L is recognized by a TM iff L is recognized by a NTM.

Proof:
(=>)
A TM is a special case of a NTM. Hence, if L is recognized by a TM, then it is also recognized by a NTM.

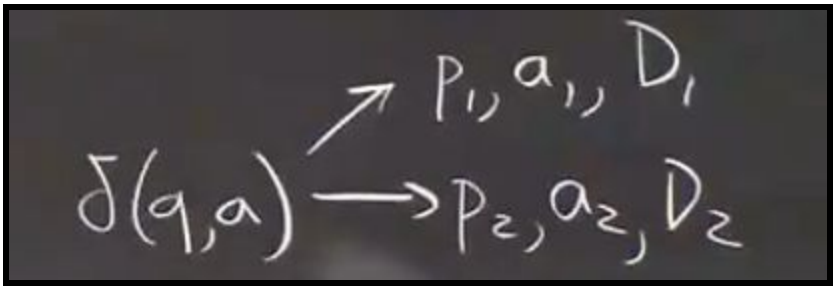
(<=)
(High level idea)

- Do a BFS of the computation tree, until you discover a configuration in the accept state, in which case we accept.
- We will use a multi-tape (2 tape) TM to simulate the NTM.
- In the multi-tape, the first tape will be a queue of configurations of the NTM. We will use a special symbol, #, to separate the configurations. However, one of the configurations will have a star, *, in front of it instead of a #. The star represents the configuration currently being looked at. Furthermore, configurations to the left of the star have already been looked at.
  The first tape is shown below.



  The second tape will be a work tape.
- Recall that configurations are of the form xqy where q is a state and xy is the contents of the NTM tape that we are simulating. Suppose that y = ay'.
  Hence, xqy = xqay'.
- The TM that's simulating the NTM scans the configuration Ci until it finds the state. Then, it consults the transition function of the NTM.

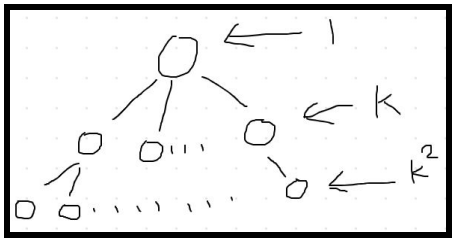
Suppose this is the transition function of the NTM.



- Then, the TM will create 2 copies of configuration Ci into the work tape. It makes 2 copies because there are 2 possibilities in the transition function. Then, it will go into each copy of the configuration in the work tape and change it based on its corresponding possibility of the transition function.
  I.e. Copy i will be changed based on the i^{th} possibility of the transition function.
- The TM will continue scanning to the right until it hits a blank symbol. Once it sees the blank symbol, it copies the contents of the work tape to the end of the queue. The work tape is now empty.
- Then, the TM goes back, looking for the star. It replaces the star with # and looks for the next #, at which it will change the # to a star and repeats for the next configuration.
- If the TM ever creates a configuration in the work tape that contains the accept state, it stops and accepts.
- **Note:** The reason why we are doing a BFS instead of a DFS is because the DFS strategy goes all the way down one branch before backing up to explore other branches. If the TM were to explore the tree in this manner, it could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design the TM to explore the tree by using BFS instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that the TM will visit every node in the tree until it encounters an accepting configuration.

- **Note:** Non-determinism does not add any powers to regular TMs.
- Suppose that a NTM M has an accepting computation path in m moves. Furthermore suppose that the maximum number of choices of M is k. How many moves does it take a simulating TM M' to accept?

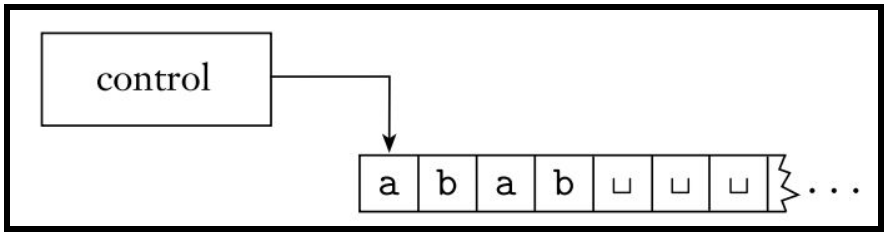
Solution:
For the root of the tree, there is 1 choice.
For the "row" after the root, there are k choices.
For the "row" after that, there are k^2 choices.
And so on.



Hence, the number of configurations explored by M' before finding the accepting configuration of M is $1 + k^2 + k^3 + \ldots + k^m$. This is $O(k^m)$.
Furthermore, if the number of moves of M is m, then the length of the configuration is m+1. This is because m symbols may have been changed, plus 1 for the state. This is O(m).
Lastly, since we copy each configuration to the work tape and back, the amount of work done for each configuration is proportional to its length.
Putting everything together, the total number of moves by M' is This is $O(m*k^m)$ which is $2^{O(m)}$.

Textbook Notes:

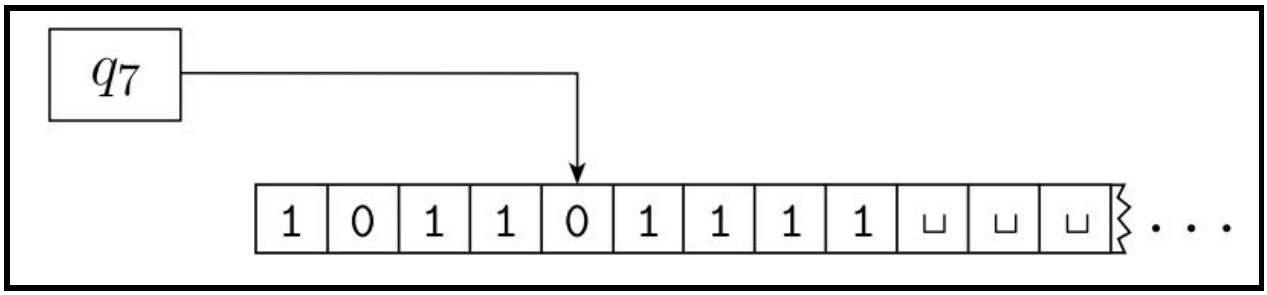
- **Introduction to Turing Machines:**
- The Turing Machine (TM) was invented by Alan Turing in 1936.
- The Turing Machine model uses an infinite tape as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape. Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs accept and reject are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.
  I.e. A schematic of a TM



- Example of a TM:
  Let's introduce a Turing machine M1 for testing membership in the language **B = {w#w| w ∈ {0,1}*}**. We want M1 to accept if its input is a member of B and to reject otherwise. Here's a high level summary of M1's algorithm on input string x:
  - Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.
  - When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept."
- **FORMAL DEFINITION OF A TM:**
- The heart of the definition of a Turing machine is the transition function δ because it tells us how the machine gets from one step to the next. For a Turing machine, δ takes the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. That is, when the machine is in a certain state q and the head is over a tape square containing a symbol a, and if δ(q, a)=(r, b, L), the machine writes the symbol b replacing the a, and goes to state r. The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case, the L indicates a move to the left.
- A **TM** is a 7-tuple, (Q, Σ, Γ, δ, Q0, Qaccept, Qreject), where Q, Σ, Γ are all finite sets and

1. Q is the set of states.
2. Σ is the input alphabet not containing the blank symbol ⊔.
3. Γ is the tape alphabet, where ⊔ ∈ Γ and Σ ⊆ Γ.
4. δ : Q × Γ→Q × Γ × {L, R} is the transition function.
5. Q0 ∈ Q is the start state.
6. Qaccept ∈ Q is the accept state.
7. Qreject ∈ Q is the reject state, where Qreject ≠ Qaccept.

- A Turing machine M = (Q, Σ, Γ, δ, Q0, Qaccept, Qreject) computes as follows. Initially, M receives its input w = w1w2 ...wn ∈ Σ^* on the leftmost n squares of the tape, and the rest of the tape is blank. The head starts on the leftmost square of the tape. Note that Σ does not contain the blank symbol, so the first blank appearing on the tape marks the end of the input. Once M has started, the computation proceeds according to the rules described by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it enters either the accept or reject states, at which point it halts. If neither occurs, M goes on forever.
- As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a **configuration** of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet Γ, we write uqv for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v.
  E.g.
  1011Q701111 represents the configuration when the tape is 101101111, the current state is Q7, and the head is currently on the second 0, as shown below:



- We say that configuration C1 **yields** configuration C2 if the Turing machine can legally go from C1 to C2 in a single step. We define this notion formally as follows:
  Suppose that we have a, b, and c in Γ, as well as u and v in Γ^* and states Qi and Qj. In that case, uaQibv and uQjacv are two configurations. We say that uaQibv yields uQjacv if in the transition function δ(Qi, b)=(Qj, c, L). That handles the case where the Turing machine moves leftward. For a rightward move, we say that uaQibv yields uacQjv if δ(Qi, b)=(Qj, c, R).
- Special cases occur when the head is at one of the ends of the configuration.
  For the left-hand end, the configuration Qibv yields Qjcv if the transition is left-moving because we prevent the machine from going off the left-hand end of the tape, and it yields cQjv for the right-moving transition.
  For the right-hand end, the configuration uaQi is equivalent to uaQi⊔ because we

assume that blanks follow the part of the tape represented in the configuration. Thus we can handle this case as before, with the head no longer at the right-hand end.

- The **start configuration** of M on input w is the configuration Q0w, which indicates that the machine is in the start state Q0 with its head at the leftmost position on the tape.
- In an **accepting configuration**, the state of the configuration is Qaccept.
- In a **rejecting configuration**, the state of the configuration is Qreject.
- Accepting and rejecting configurations are **halting configurations** and do not yield further configurations.
- A Turing machine M accepts input w if a sequence of configurations C1, C2, ... , Ck exists, where:
    1. C1 is the start configuration of M on input w,
    2. each Ci yields Ci+1, and
    3. Ck is an accepting configuration
- The collection of strings that M accepts is called **the language of M** or **the language recognized by M** and is denoted L(M).
- We call a language **Turing-recognizable** if some Turing machine recognizes it.
- When we start a Turing machine on an input, three outcomes are possible: accept, reject or **loop**. By **loop** we mean that the machine simply does not halt. Looping may entail any simple or complex behavior that never leads to a halting state.
- A Turing machine M can fail to accept an input by entering the Qreject state and rejecting or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason, we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called **deciders** because they always make a decision to accept or reject. A **decider** that recognizes some language also is said to **decide** that language. We call a language **Turing-decidable** or simply decidable if some Turing machine decides it.
- **VARIANTS OF TURING MACHINES:**
    1. **MULTI-TAPE TURING MACHINES:**
        - A **multi-tape Turing machine** is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank. The transition function is changed to allow for reading, writing, and moving the heads on some or all of the tapes simultaneously. Formally, the transition function is defined as:

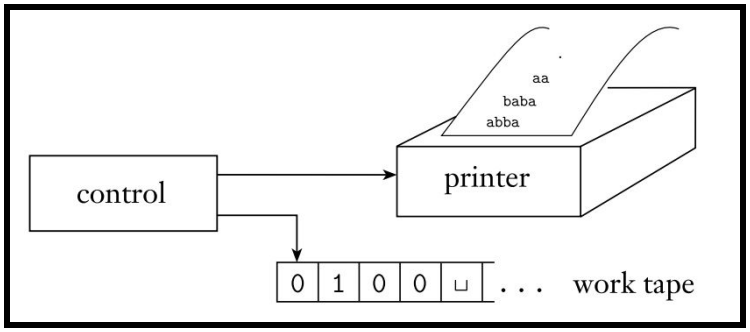
$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k$$

          where k is the number of tapes.
        - The expression δ(Qi, a1,...,ak)=(Qj, b1,...,bk,L, R,...,L) means that if the machine is in state Qi and heads 1 through k are reading symbols a1 through ak, the machine goes to state Qj, writes symbols b1 through bk, and directs each head to move left or right, or to stay put, as specified.
        - **Theorem:** Every multi-tape Turing machine has an equivalent single-tape Turing machine.
        - **Corollary:** A language is Turing-recognizable iff some multi-tape Turing machine recognizes it.

Proof:
A Turing-recognizable language is recognized by an ordinary (single tape) Turing machine, which is a special case of a multi-tape Turing machine. That proves one direction of this corollary. The other direction follows from the above theorem that every multi-tape Turing machine has an equivalent single-tape Turing machine.

2. **NONDETERMINISTIC TURING MACHINES:**
   - **Theorem:** Every nondeterministic Turing machine has an equivalent deterministic Turing machine.
   - **Corollary:** A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.
   - **Corollary:** A language is decidable if and only if some nondeterministic Turing machine decides it.

- **ENUMERATORS:**
- Loosely defined, an **enumerator** is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer.
  I.e.



- An enumerator E starts with a blank input on its work tape. If the enumerator doesn't halt, it may print an infinite list of strings. The language enumerated by E is the collection of all the strings that it eventually prints out. Moreover, E may generate the strings of the language in any order, possibly with repetitions.
- More specifically, an **enumerator** for a language $L \subseteq \Sigma^*$ is a multitape TM E_L that, if started with all tapes being blank, lists on Tape 1 every element of L preceded and followed by a special symbol #, where $\# \notin \Sigma$. The computation of E_L started in its initial state with all tapes blank is a sequence of configurations $C1 \vdash C2 \vdash \cdots \vdash Ci \vdash \cdots$, such that:
  1. If $i < j$ then the string contained in Tape 1 in Ci is a prefix of that in Cj; and
  2. #x# is a substring of the string contained in Tape 1 in some Ci if and only if $x \in L$.
- **Theorem:** A language is Turing-recognizable if and only if some enumerator enumerates it.

Proof:
=>
First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A.
The TM M works in the following way.

M = “On input w:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appear in the output of E, accept.”

Clearly, M accepts those strings that appear on E’s list.

<=

If TM M recognizes a language A, we can construct the following enumerator E for A. Say that s1, s2, s3,... is a list of all possible strings in Σ^*.

E = “Ignore the input.

Repeat the following for i = 1, 2, 3,... .

a. Run M for i steps on each input, s1, s2,...,si.
b. If any computations accept, print out the corresponding sj.”

If M accepts a particular string s, eventually it will appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1.

Here, we’re defining A as the set of all strings that are accepted by M. We may not know what these strings are given by an arbitrary M, so we run M on every possible string for every possible number of steps and see which are accepted.

Naming Conventions For The Rest of The Course:

- p, q are states
- a,b,c (letters near the beginning of the alphabet) $\in$ r (They’re tape symbols.)
- v,w,x,y,z (letters near the end of the alphabet) $\in r^*$ (They’re strings over the alphabet r.)
- ␣ denotes the blank symbol.

Note:

- A **string** is a finite set of symbols, where each symbol belongs to an **alphabet** denoted by **Σ**.
- The set of all strings that can be constructed from an alphabet Σ is $\mathbf{\Sigma^*}$.
- **ε** is the empty string.

Church's Thesis:

- Also known as **Church-Turing Thesis**.
- It is a hypothesis about computable functions.
- It says that any real-world computation can be translated into an equivalent computation involving a Turing machine.
  I.e. It states that a function on the natural numbers is computable by a human following an algorithm iff the function is computable by a Turing Machine.
- **Mathematical evidence for Church's Thesis:**
  1. **Extensions to TMs are equivalent to TMs.**
     We're dealing with different ways of looking at the same problem.
  2. **Competing formalisms for algorithms are equivalent.**
     Again, we're dealing with different ways of looking at the same problem.
     People had not bothered to define algorithms for thousands of years because it was not necessary. Then, in the space of 3 years, 3 different definitions were proposed.
     The first way is **partial recursive functions** and it was done by Gödel and Herbrandt in 1933.
     The second way is **λ-calculus (lambda calculus)** and it was done by Church and his student Kleene in 1936.
     The third way is **Turing Machines** and it was done by Turing in 1936.
  3. **Universality**

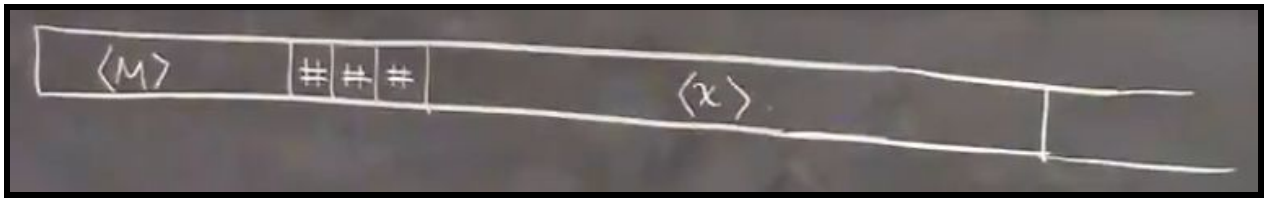
Universality:

- Consider a TM, M_u, that takes in as input a description of any TM, M, and any string, x, that M can work on and simulates the computation M on x. M_u is called the **universal TM**.
- These universal TMs have a self-referential aspect. They can talk about TMs. However, this is the source of some problems. Recall from the first lecture that many of the issues with paradoxes stemmed from the fact that they had a self-referential aspect.
- Going back to M_u, there is an issue. Like all TMs, M_u must have a set of states, an input alphabet, a tape alphabet, etc. The problem lies with its tape alphabet. The tape alphabet consists of a fixed, finite set of symbols, but it's supposed to simulate any TM that could have an entirely different set of symbols. So, I need to be able to find a way to encode an arbitrary TM that has its own states and its own tape alphabet and its own input alphabet using the fixed tape alphabet of M_u. There are many different methods we can use to encode and it doesn't matter which method we use. I will illustrate one possible encoding of TMs using the alphabet {0, 1, #}, where # acts as a separator. We will use an arbitrary TM M, s.t. M = (Q, Σ, Γ, δ, Q0, Qa, Qr). Here's how I'll encode M:
  1. Encode state q by a binary string, denoted as ⟨q⟩, of ceil($\log_2^{|Q|}$) bits.
  2. Encode Q by listing the codes of its elements separated by #'s.
     I.e. We can encode Q as ⟨Q0⟩#⟨Qa⟩#⟨Qr⟩#⟨Q1⟩#...
     We first list the initial state, followed by the accept state, then the reject state, and then the rest of the states in order.
  3. Encode symbol a ∈ Γ by a binary string, denoted as ⟨a⟩, of ceil($\log_2^{|\Gamma|}$) bits.
  4. Encode Σ by listing the codes of symbols in Σ separated by #'s.
     I.e. We can encode Σ as ⟨a⟩#⟨b⟩#⟨c⟩#...
  5. Encode Γ similarly to Σ, but we start the list with the code for the blank symbol, ␣.
  6. Encode x = a1a2a3...ak ∈ Γ^* as ⟨a1⟩⟨a2⟩⟨a3⟩…

7. Encode transition δ(q,a) = (p, b, D) as ⟨q⟩#⟨a⟩#⟨p⟩#⟨b⟩#⟨D⟩.
8. Encode δ as ⟨transition1⟩##⟨transition2⟩##⟨transition3⟩…
9. Encode M, denoted as ⟨M⟩, as ⟨Q⟩###⟨Σ⟩###⟨Γ⟩###⟨δ⟩.

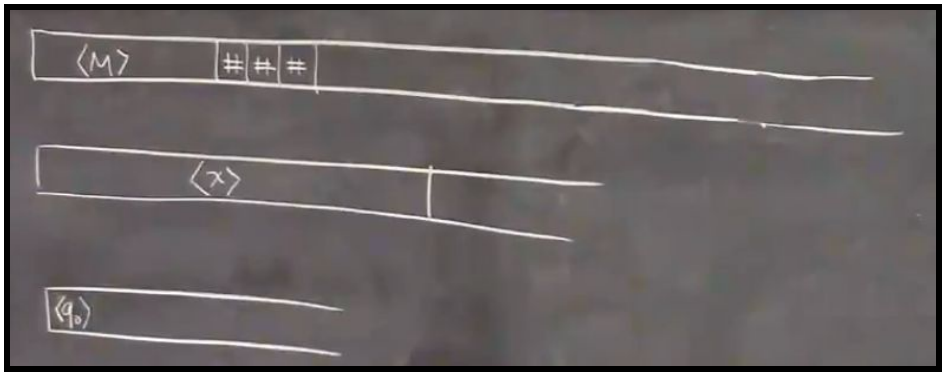
Note: Strings in $\{0,1,\#\}^*$ that don't encode a TM as above, encode TM that rejects all strings.

- The universal TM, M_u, takes as input encoding of a TM, M, and its input x and simulates M on x.
  I.e. The input of M_u is: ⟨M,x⟩
- **Theorem 3.2:** There is a TM M_u called the universal TM that:
  - accepts ⟨M,x⟩ if M accepts x
  - rejects ⟨M,x⟩ if M rejects x
  - loops on ⟨M,x⟩ if M loops on x
- We can describe M_u as a 3-tape TM. This can be converted into a regular 1-tape TM.
  The tape alphabet of M_u is {0,1,#,␣}.
  Recall that with multi-tape TMs, the input arrives in the leftmost cell(s) of tape 1 and the other tapes are left blank.
  The first tape of M_u is the input tape. The input tape contains an encoding of some TM, M, and an encoding of M's input, x.



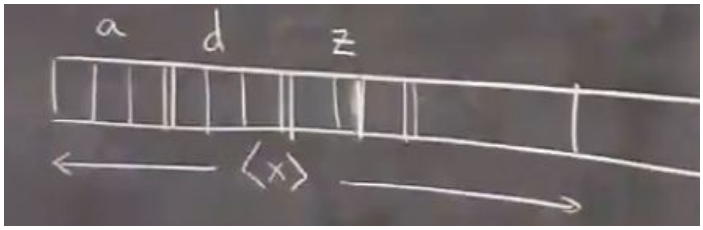
The second tape of M_u is initially blank but it will eventually contain M's encoded tape.
The third tape of M_u is initially blank but it will eventually contain the encoded state of M.

The first thing that M_u does is that it copies ⟨x⟩ onto the second tape, erases ⟨x⟩ from the first tape and puts on the third tape the encoding of the initial state of M.



Recall that every symbol in ⟨x⟩ is represented by $\text{ceil}(\log_2^{|\Gamma|})$ bits. Suppose that $\text{ceil}(\log_2^{|\Gamma|})$ is 3.
I.e. I need 3 bits to represent every tape symbol in ⟨x⟩.

Suppose that the first 3 bits represent "a", the second 3 bits represent "d" and the third 3 bits represent "z", as shown below.



Lastly, suppose that each tape head starts out by pointing to the first cell of their respective tape.
This is how M_u works:

1. It says, let me go to tape 3 and see what is the state.
2. Then, it goes to tape 2 and reads the 3 cells (bits) that make up the first symbol. Now, M_u has figured out that it is simulating M in state Q0, scanning symbol "a".
3. Then, it goes to tape 1 and finds the transition function "⟨Q0⟩#⟨a⟩" and looks for should happen next. Suppose that the entire transition function is ⟨Q0⟩#⟨a⟩#⟨p⟩#⟨b⟩#⟨R⟩.
4. Then, M_u changes the state in tape 3 to ⟨p⟩, the 3 bits that represent "a", in tape 2, to the 3 bits that represent "b", and then it moves 3 cells to the right in tape 2. It moves 3 cells to the right because every 3 cells represents one symbol.
5. M_u has now simulated 1 move of M. It then repeats steps 1-4 for the new state and symbol. It keeps repeating until it sees Qa or Qr, at which, it will either accept or reject.

- **Theorem 3.3:** If L is a decidable language, then its complement is also decidable.
I.e. The set of decidable languages is closed under complementation.

Proof:
Suppose that if L is decidable, M is a TM that decides L. Suppose that M takes an input, x. Then, M either accepts x, if $x \in L$ or rejects x, if $x \notin L$. Using M, we can construct another TM, M', that decides the complement of L, L'. If M accepts x, M' rejects x. If M rejects x, M' accepts x.

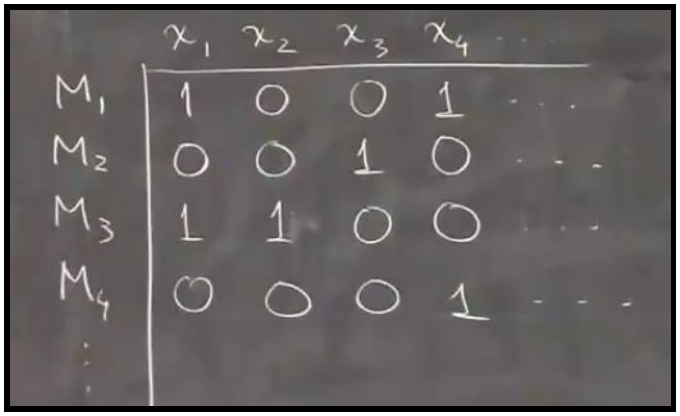
- Proof that there are languages that are not recognizable:

Fix Σ to include enough symbols to encode TMs.
E.g. Σ = {0, 1, #}
Now, consider all languages over Σ.
I will define 2 languages, U, the universal language, and D, diagonal.
U = {⟨M, x⟩ | M accepts x}
D = {⟨M⟩ | M does not accept ⟨M⟩}

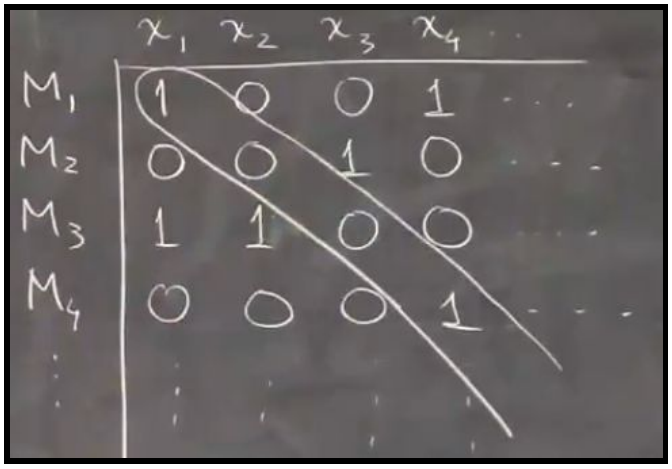
Theorem 3.4: D is not recognizable.

Proof:
Suppose for contradiction that D is recognizable. Let M_D be a TM that recognizes D.
This means that ∀ TM M, M_D accepts ⟨M⟩ iff M does not accept ⟨M⟩.
Take M = M_D. This means that M_D accepts ⟨M_D⟩ iff M_D does not accept ⟨M_D⟩.
This is a contradiction. Hence, D is not recognizable.
Another way to think about the proof:
Let's enumerate all the TMs M1, M2, M3, …, over all of their inputs.

We put a 1 if Mi accepts Xj and 0 otherwise.
An example table is shown below.



Let's look at the main diagonal of the table, shown below.



I will complement the string along the main diagonal.
I.e. 1001 becomes 0110.
Let D be the complement string (0110…).
If M_D existed, it would be in one of the enumerations, but M_D cannot be any of the TMs enumerated because it differs from every TM by at least 1 input.
Hence, M_D doesn't exist.

- **Theorem 3.5:** U, the universal language, is
  a) recognizable, but
  b) not decidable.
  U = {⟨M, x⟩ | M accepts x}

  Proof of a):
  M_u recognizes U.

  Proof of b):
  Suppose for contradiction that U is decidable. Let M_1 be a TM that decides it.
  We can use M_1 to decide the complement of D, ¬D.
  ¬D = {⟨M⟩ | M accepts ⟨M⟩}.
  Here's how we can use M_1 to decide ¬D.
  We can use M_1 to create another TM, M_2.

M_2 on input ⟨M⟩:

1. Construct ⟨M, ⟨M⟩⟩
2. Run M_1 on ⟨M, ⟨M⟩⟩
3. If M_1 accepts, accept
4. Otherwise, reject

If M_2 is a decider for ¬D, then ¬(¬D), the complement of ¬D, or D is also decidable. However, in theorem 3.4, we proved that D is not recognizable. Hence, this is a contradiction. Hence, U is not decidable.

- The proof of theorem 3.5, part b, is a **proof by reduction**.
  Problem P reduces to problem Q iff there is an algorithm to solve P that uses an assumed algorithm for problem Q as a black box.
  I.e. We can't look into the code of the algorithm to see how it works. We just give it an input and it gives back an output.

  For the proof of theorem 3.5, part b, we reduced ¬D to U, which can be written as ¬D ≤ U.
- **Theorem 3.6:** If L and ¬L, the complement of L, are both recognizable, then L and ¬L are decidable.

  Proof:
  Suppose that L and ¬L are both recognizable.
  Let M_1 be a TM that recognizes L and let M_2 be a TM that recognizes ¬L.
  Construct a TM, M, that decides L.
  M simulates M_1 for 1 step and M_2 for 1 step and repeats until either M_1 or M_2 accepts.
  This is because the input must be either in M_1 or M_2. It must either be in L or not be in L.
  If M_1 accepts, then accept.
  If M_2 accepts, then reject.
- **Corollary 3.7:** The complement of U, ¬U, is not recognizable.
  ¬U = {⟨M, x⟩ | M does not accepts x}

  Proof:
  Suppose for contradiction that ¬U is recognizable.
  In theorem 3.5 a), we said that U is recognizable.
  Hence, by theorem 3.6, both U and ¬U are decidable.
  However, in theorem 3.5 b), we said that U is not decidable.
  Hence, this is a contradiction and ¬U is not recognizable.
- **Corollary 3.8:** The set of recognizable languages is not closed under complementation.

General Turing Reductions:

- P **Turing reduces** to Q if there exists an algorithm for P that uses an algorithm for Q as a "black box". This is denoted as $P \leq_T Q$.

Halting Problem:

- Denoted as H.
- $H = \{\langle M, x\rangle \mid \text{TM } M \text{ halts on input } x\}$
  H accepts the encodings of M and x if M halts on x, and rejects the encodings otherwise.
- **Theorem 4.1:** H is
  a. recognizable but
  b. not decidable.

Proof of a):
Use M_u to simulate M on input x.
If M_u halts (either accepts or rejects), then we say yes.
If M_u doesn't halt, then it's fine because H is a recognizer, not a decider.

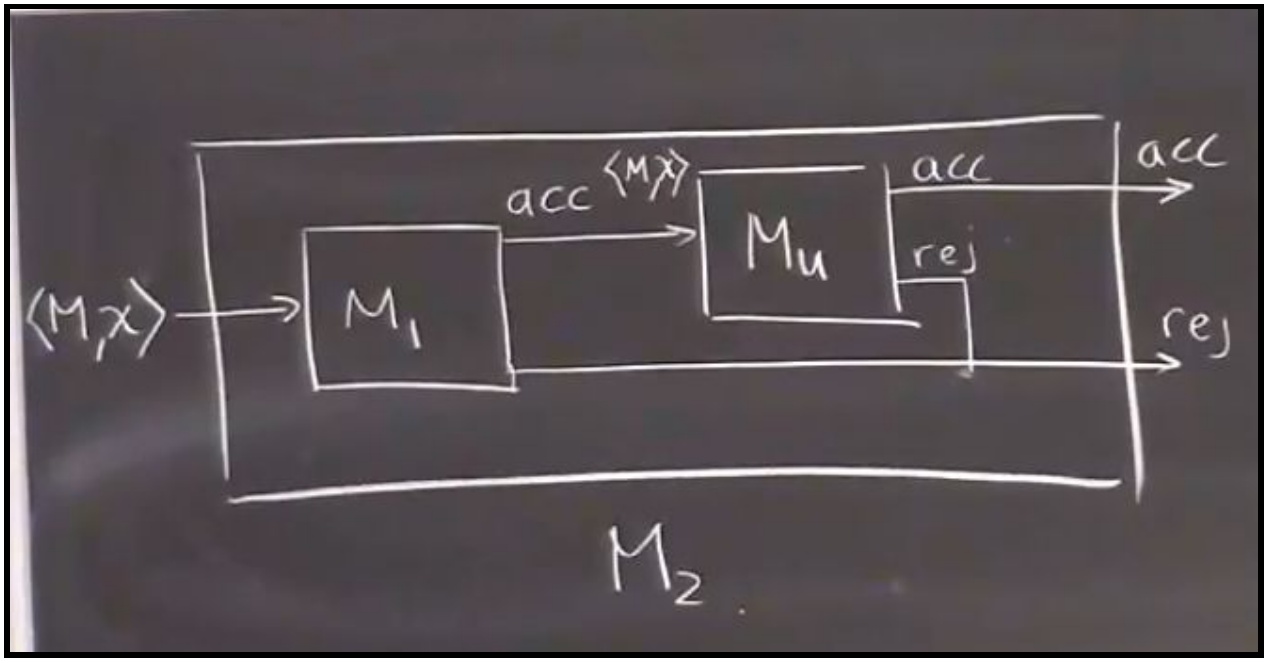
Proof of b):
We will show that $U \leq H$.
U is the universal language. In theorem 3.5, we proved that U is recognizable but not decidable.
Given an H-decider TM M_1, we will construct a U-decider TM M_2.
M_2 on input $\langle M, x\rangle$ does the following:

1. Run M_1 on $\langle M, x\rangle$
2. If M_1 accepts, then
3. Run M_u on $\langle M, x\rangle$
4. If M_u accepts $\langle M, x\rangle$, then M_2 accepts
5. Else, M_2 rejects
6. Else, M_2 reject

M_2 accepts U.
First, it runs M_1 on $\langle M, x\rangle$.
If M_1 accepts, meaning that it halts on $\langle M, x\rangle$, then we run M_u on $\langle M, x\rangle$.
If M_u accepts $\langle M, x\rangle$, then M_2 accepts $\langle M, x\rangle$.
If M_u doesn't accept $\langle M, x\rangle$, then M_2 rejects $\langle M, x\rangle$.
If M_1 doesn't accept $\langle M, x\rangle$, that means M_1 doesn't halt on $\langle M, x\rangle$, so M_2 rejects $\langle M, x\rangle$.

However, we proved in theorem 3.5 that U is not decidable, so M_2 doesn't exist.
Since M_2 relies on the existence of M_1, therefore, M_1 doesn't exist.
Therefore, H is undecidable.

Here's a diagram of the proof:

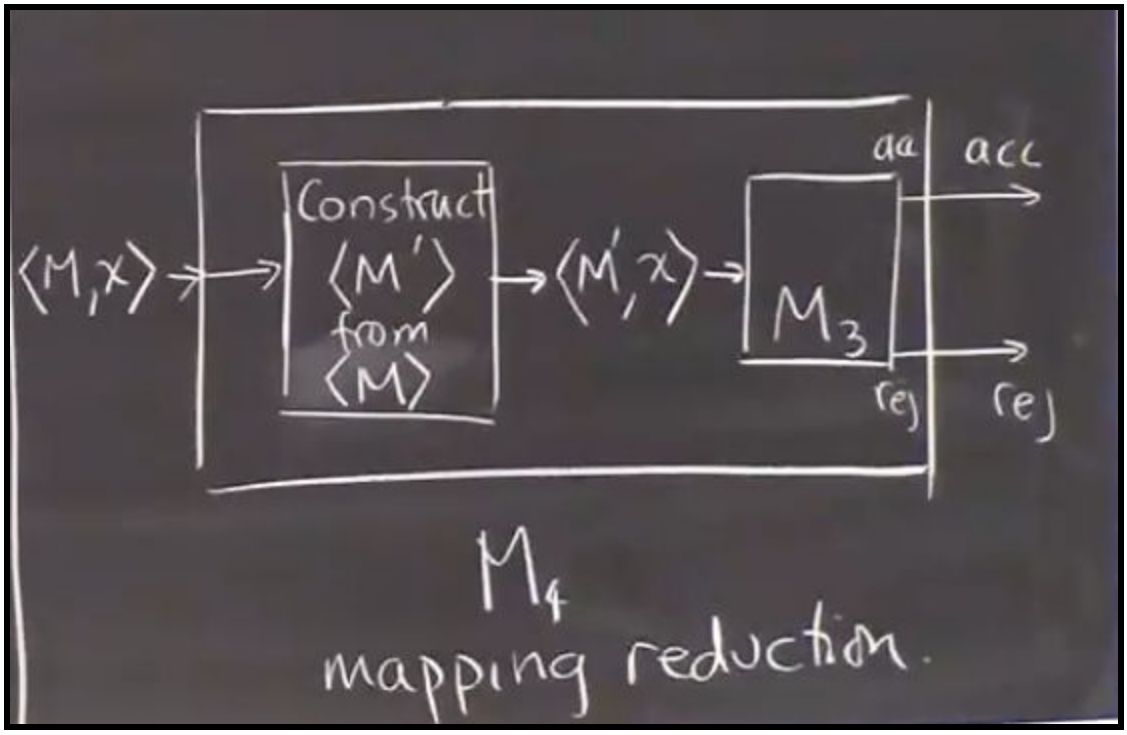


Alternative Proof of b):
Given an H-decider M_3, we can construct a U-decider M_4 as follows:
M_4 on input ⟨M, x⟩ does the following:

1. Modify M to M' by changing every transition of M to the reject state into an infinite loop.
   We know that M either accepts, rejects or loops on x.
   If M accepts x, then M' accepts x.
   If M rejects or loops on x, then M' loops on x.
2. Run M_3 on ⟨M', x⟩.
3. If M_3 accepts, then M_4 accepts.
4. Else, M_4 rejects.

M_4 accepts ⟨M, x⟩
↔ M_3 accepts ⟨M', x⟩
↔ M' halts on x
↔ M accepts x
Therefore, M_4 is a U-decider.
However, this contradicts theorem 3.5, which says that U is not decidable.
Therefore, we could not have been given an M_3 that solves the halting problem.

Here's a diagram of the proof:



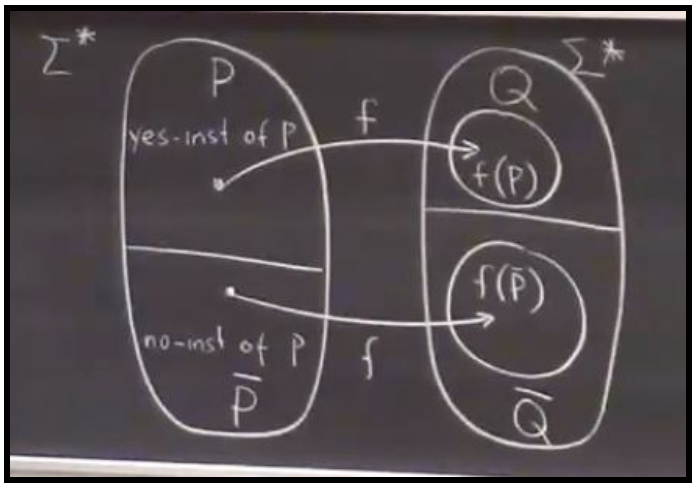
- **Corollary 4.2:** ¬H, the complement of H, is unrecognizable.
  ¬H = {⟨M, x⟩ | M doesn't halt on x}

  Proof:
  Suppose by contradiction that ¬H is recognizable.
  Recall theorem 3.6 "If L and ¬L, the complement of L, are both recognizable, then L and ¬L are decidable."
  Since both H and ¬H are recognizable, then both H and ¬H are decidable.
  However, we know that H is not decidable, which is a contradiction.
  Hence, ¬H is unrecognizable.
- If X ≤ Y and X is undecidable, then Y is also undecidable.
  However, if X ≤ Y and Y is undecidable, it doesn't tell us if X is undecidable or not.
  Note: The direction in which we are reducing things is very important.
  E.g.
  When we did U ≤ H, since we knew that U is undecidable, we could prove that H is undecidable.
  However, if we did H ≤ U, we know that U is undecidable, but we don't know if H is undecidable. We can't use this to prove that H is undecidable.
- If X ≤ Y and Y is decidable, then X is also decidable.

Mapping Reductions:

- **Definition:** Let P and Q ⊆ Σ^* be languages. P is **mapping-reducible** to Q, denoted as $P \leq_m Q$, iff there exists a computable function, $f : \Sigma^* \rightarrow \Sigma^*$, such that $x \in P$ iff $f(x) \in Q$.
  Note: The function, f, does not have to be, and is usually not, onto.
  Note: The function, f, must be computable.
  To demonstrate a computable function, we will typically write a little program or describe in English how to perform the transformation that f is supposed to do.

Note: f maps yes-instances of P to yes-instances of Q and no-instances of P to no-instances of Q.
Here is a diagram to show the definition of mapping-reducible:



Here, f maps the Yes-instances of P to a subset of the Yes-instances of Q and maps the No-instances of P to a subset of the No-instances of Q.

- E.g. Suppose that
  A = {x | x is an even integer}
  B = {x | x is an odd integer}
  Then the function f(x) = x + 1 is a mapping reduction from A to B.
  Notice that:
  $x \in A \leftrightarrow$ x is even
  $\leftrightarrow$ x + 1 is odd
  $\leftrightarrow x + 1 \in B$
  $\leftrightarrow f(x) \in B$
- All the reductions we've seen so far, with one exception, are mapping reductions.
  1. **First Reduction: Reduced ¬D (D complement) to U (Universal language)**
  - ¬D = {⟨M⟩ | M accepts ⟨M⟩}
  - f: ⟨M⟩ → ⟨M, ⟨M⟩⟩
    Here is a description of f:
    - Take the encoding of M.
    - Make a pair of itself and another encoding of M in the following way: ⟨M⟩###⟨M⟩ (The ### is used as a separator.)
  2. **Second Reduction: Reduce U to H (The Halting Problem)**
  - **Note:** This is for the "Alternative Proof of b)"
  - Given ⟨M, x⟩ we constructed ⟨M', x⟩ such that M accepts x iff M' halts on x.
    M accepts x simply means ⟨M, x⟩∈U and M' halts on x simply means ⟨M', x⟩∈H.
    So, I mapped ⟨M, x⟩ to ⟨M', x⟩ such that Yes-instances go to Yes-instances and No-instances go to No-instances.
  - **Note:** The first proof we did to prove that U reduces to H is not a mapping reduction. The difference between the first and second proof is that with the first proof, we're taking the input, ⟨M, x⟩, and running it through 2 "black boxes", M_1 and M_u. Furthermore, after running the input through the first "black box", M_1, there's a possibility that we're changing its output by running the output through the second "black box", M_u.

With the second proof, we're transforming ⟨M, x⟩ to ⟨M', x⟩, this is our function, and we're only running it through 1 TM, M_3. In the second proof, we're using M_3 in a very restricted way. We are only making 1 call to the "black box" and we're using the output of the "black box" as it is, we can't change it.

- Hence, the first proof is a **Turing reduction** while the second proof is a **mapping reduction**.

- **Theorem 4.3:** Suppose that $P \leq_m Q$. If Q is decidable, then P is decidable. If P is undecidable, then Q is undecidable.

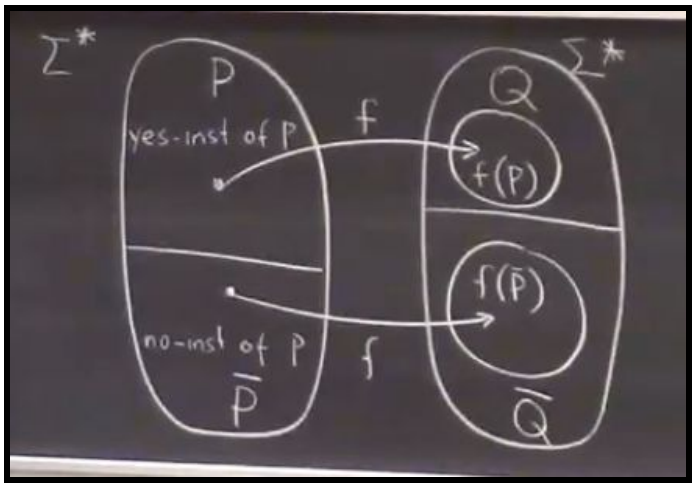
Proof of "If P is undecidable, then Q is undecidable":
Assume that $P \leq_m Q$ and P is undecidable.
Suppose for contradiction that Q is decidable.
Let D_Q be a decider for Q.
Since $P \leq_m Q$, there exists a computable function, f, such that $x \in P$ iff $f(x) \in Q$.
Then, the following algorithm is a decider for P:
D_P on input "x" does the following:

1. Computes f(x)
2. Run D_Q on f(x).
3. If D_Q accepts, then D_P accepts.
4. Else, D_P rejects.

D_P halts on all inputs, so it's a decider.
D_P decides P because it accepts x iff D_Q accepts f(x).
D_Q accepts f(x) iff $f(x) \in Q$, because D_Q is a decider for Q.
$f(x) \in Q$ iff $x \in P$, because f is a mapping reduction of P to Q.
However, this contradicts our supposition that P is undecidable.
Hence, Q is undecidable.

- **Theorem 4.4:** If $P \leq_m Q$ and Q is recognizable, then P is recognizable. If P is unrecognizable, then Q is unrecognizable.
- **Theorem 4.5:** If $P \leq_m Q$, then $\neg P \leq_m \neg Q$, where $\neg P$ is the complement of P and $\neg Q$ is the complement of Q.

Proof:
Consider the diagram below.



We know that f maps the Yes-instances of P to the Yes-instances of Q and the No-instances of P to the No-instances of Q.
However,

- The No-instances of P are the same as the Yes-instances of ¬P.
- The No-instances of Q are the same as the Yes-instances of ¬Q.
- The Yes-instances of P are the same as the No-instances of ¬P.
- The Yes-instances of Q are the same as the No-instances of ¬Q.

Hence, we can use the same function, f, as the computable function for $\neg P \leq_m \neg Q$.

- **Theorem 4.6:** If $P \leq_m Q$ and $Q \leq_m R$, then $P \leq_m R$.

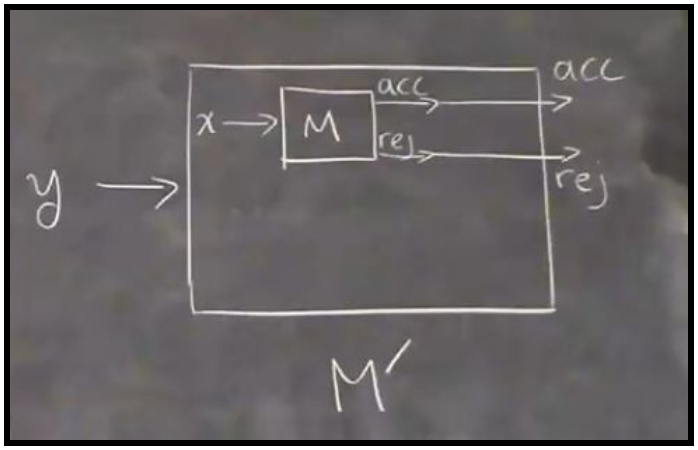
Examples of Reductions:

- To prove that a language P is unrecognizable or undecidable, it suffices to prove that $U \leq_m P$, for undecidable, and $\neg U \leq_m P$, for unrecognizable. This is by theorem 4.3 and 3.4.
- **Theorem 4.7:** Consider the following language, $E = \{\langle M \rangle \mid L(m) = \emptyset\}$. E is unrecognizable.

Proof:
It suffices to prove that $\neg U \leq_m E$.
Given $\langle M,x \rangle$, which is the input to ¬U, we want to construct $\langle M' \rangle$, which is the input to E, such that M does not accept x iff $L(M') = \emptyset$.
I.e. $\langle M,x \rangle \in \neg U$ iff $\langle M' \rangle \in E$.
We can build M' such that if M doesn't accept x, M' accepts no string, and if M accepts x, M' accepts every string.
f on input $\langle M,x \rangle$ does the following:

1. Define a machine M' that does the following on input y:
    a. Run M on x
    b. If M accepts, then M' accepts y.
    c. Else, M' rejects y.
2. Return $\langle M' \rangle$

Here's a diagram showcasing the proof.



If M does not accept x, then $L(M') = \emptyset$.
If M accepts x, then $L(M') = \Sigma^*$.

Claim: f is a mapping reduction of ¬U to E.
Proof:
To prove that f is a mapping reduction of ¬U to E, we need to verify that
⟨M,x⟩ ∈ ¬U iff ⟨M'⟩ ∈ E.
(=>) If ⟨M,x⟩ ∈ ¬U
→ M does not accept x. (M either loops on x or M rejects x.)
→ M' accepts no input.
→ L(M') = Ø
→ ⟨M'⟩ ∈ E

(<=) If ⟨M,x⟩ ∉ ¬U
→ M accepts x.
→ M' accepts all inputs.
→ L(M') = $\sum^*$ ≠ Ø
→ ⟨M'⟩ ∉ E

- **Theorem 4.8:** ¬E, the complement of E, is
    a. undecidable, but
    b. recognizable

¬E = {⟨M⟩| L(M) ≠ Ø}

Proof of a):
Suppose for contradiction that ¬E is decidable.
Then, based on theorem 3.3, which states that
"If L is a decidable language, then its complement is also decidable.
I.e. The set of decidable languages is closed under complementation.",
then E is also decidable.
However, we just proved in theorem 4.7 that E is undecidable, which is a contradiction.
Hence, ¬E is undecidable.

Proof of b):
The idea is to dovetail through all pairs (i,j). When visiting pair (i,j), run M on the i^{th} input for j steps. If it accepts, then we accept. Otherwise, visit the next pair.
If M doesn't accept or reject the i^{th} input for j steps, we simply continue the dovetailing process. This is fine because as a recognizer, it doesn't need to halt.
Note: The reason why you can't simply go down each input is because there might be an input that loops forever. Then, your machine would be stuck.

Another Proof of b):
A NTM recognizes ¬E on input ⟨M⟩ as follows:
1. Nondeterministically guess a string x.
2. Use M_u, the universal TM, to run M on x.
3. If M accepts, accept.
4. Since there's a NTM that recognizes ¬E, there's also a TM that recognizes ¬E.

- **Theorem 4.9:** Consider the following language, REG = {⟨M⟩ | L(M) is regular}. REG is undecidable.

Proof:
It suffices to prove that $U \leq_M REG$.
Given an input, ⟨M, x⟩ to U, we want to construct a machine ⟨M'⟩, which is an input to REG such that M accepts x iff L(M') is regular.
If M accepts x, then M' accepts a regular language.
If M does not accept x, then M does not accept a regular language.
f on input ⟨M, x⟩ does the following:

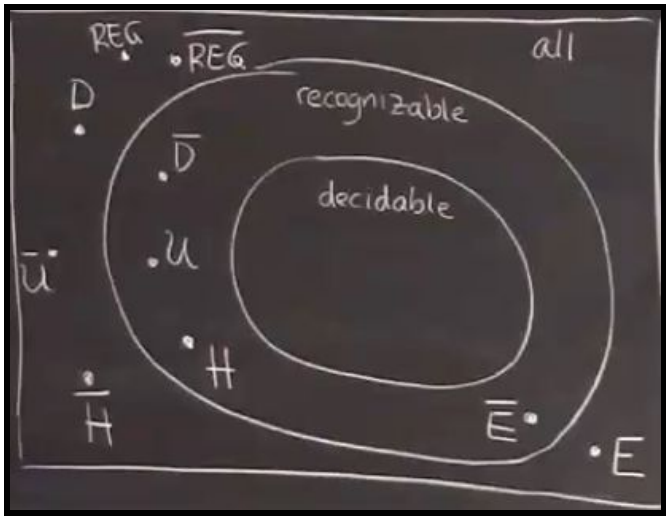
1. Define M' which on input y does the following:
    a. If $y=0^n1^n$, then accept
    b. Else, run M on x.
    c. If M accepts x, then M' accepts y.
    d. Else, M' rejects y.
2. Return ⟨M'⟩

Now, we need to verify that ⟨M, x⟩ ∈ U iff ⟨M'⟩ ∈ REG.
(=>)
If ⟨M, x⟩ ∈ U then
→ M accepts x.
→ M' accepts all inputs y. It does this in either line 1a. or line 1c. otherwise.
→ $L(M') = \sum^*$.
→ ⟨M'⟩ ∈ REG.

(<=)
If ⟨M, x⟩ ∉ U then
→ M does not accept x.
→ M' accepts all and only strings of the form 0^n1^n.
→ L(M') is not regular.
→ ⟨M'⟩ ∉ REG.
We have shown that $U \leq_M REG$, so REG is undecidable.

Note: Since $U \leq_M REG$, $\neg U \leq_M \neg REG$. This is by theorem 4.5.
Since $\neg U \leq_M \neg REG$, ¬REG is unrecognizable.
Note: $U \leq_M \neg REG$, which means that $\neg U \leq_M REG$.
Since $\neg U \leq_M REG$, REG is unrecognizable.

Picture of Recognizable and Decidable Languages:



More examples of mapping reductions:

- Consider the following set, HET = {⟨M⟩ | TM M halts on empty tape}.
  HET stands for Halts on Empty Tape. The input for HET is ε (the empty string).
- **Theorem 5.1:** HET is not decidable.

  Proof:
  It suffices to show that $U \leq_m HET$.
  Given ⟨M,x⟩ to U, we want to construct ⟨M'⟩ to HET, such that M accepts x iff M' halts on empty tape.
  If M accepts x, then M' should halt on empty tape.
  If M does not accept x, then M' should not halt on empty tape.

  f on input ⟨M,x⟩ does the following:
  1. Define M'. M' on input y does the following:
     a. Runs M on x.
     b. If M accepts, then M' accepts y.
     c. Else, loop.
  2. Returns ⟨M'⟩

  If M accepts x, then M' accepts $\sum^*$ (everything).
  In particular, M' accepts ε.
  This means that M' halts on empty tape.
  This means that ⟨M'⟩ ∈ HET.

  If M does not accept x, then M' always loops.
  M' could loop at 2 places:
  1. Line 1a. "Runs M on x". If M does not accept x because it loops, then M' loops here.
  2. Line 1c. "Else, loop" If M does not accept x because it rejects it, then M' loops here.

  In particular, M' does not halt on empty tape.
  Therefore, ⟨M'⟩ ∉ HET.

  Note: HET is recognizable.
- **Theorem 5.2:** Let ODD = {⟨M⟩ | L(M) is finite and |L(M)| is odd}. ODD is not recognizable.

  Proof:
  Suffices to prove that $\neg U \leq_m ODD$.
  Given ⟨M, x⟩ to ¬U, we want to construct ⟨M'⟩ to ODD, such that M does not accept x iff L(M') is finite and |L(M)| is odd.
  If M does not accept x, then L(M') is finite and |L(M)| is odd.
  If M accepts x, then L(M') is infinite or |L(M)| is even.

f on input ⟨M, x⟩ does the following:

1. Define M'. M' on input y does the following:
    a. If y = 010, then accept.
    b. Run M on x.
    c. If M accepts x, then M' accepts anything. (M' accepts an infinite number of strings.)
    d. Else, M' rejects y.
2. Return ⟨M'⟩

If M does not accept x, then M' will either accept {010} (line 1a.) or it will reject everything. Hence, the only string L(M') can be is {010}. Therefore, ⟨M'⟩ ∈ ODD.

If M accepts x, L(M') = $\sum^*$, which is infinite. Therefore, ⟨M'⟩ ∉ ODD.

- **Theorem 5.3:** Let FIN = {⟨M⟩ | L(M) is finite} and let INF = ¬FIN.

I.e. INF is the complement of FIN.
I.e. INF = {⟨M⟩ | L(M) is infinite}
Both FIN and INF are not recognizable.

Proof:
It suffices to show that

a. $U \leq_m FIN$
b. $U \leq_m INF$

Note: This is the reason why we can use U instead of ¬U:
Recall from theorem 4.5 that "If $P \leq_m Q$, then $\neg P \leq_m \neg Q$, where ¬P is the complement of P and ¬Q is the complement of Q."
Since we can show that $\neg U \leq_m INF$, we can show that $\neg(\neg U) \leq_m \neg INF$ or $U \leq_m FIN$.
Similarly, since we can show that $\neg U \leq_m FIN$, we can show that $\neg(\neg U) \leq_m \neg FIN$ or $U \leq_m INF$.

Proof for INF:
Given ⟨M, x⟩ to U, we want to construct ⟨M'⟩ to INF such that if M accepts x, then M' accepts an infinite language and if M doesn't accept x, then M' accepts a finite language.

f on input ⟨M, x⟩ does the following:

1. Define M'. M' on input y does the following:
    a. If y = Ø, then accept.
    b. Run M on x.
    c. If M accepts x, then M' accepts anything. (M' accepts an infinite number of strings.)
    d. Else, M' rejects y.
2. Return ⟨M'⟩

If M accepts x, then L(M') = $\sum^*$. Therefore, ⟨M'⟩ ∈ INF.
If M does not accept x, then M' will either accept Ø (line 1a.) or it will reject everything. Therefore, ⟨M'⟩ ∉ INF.

Proof for FIN:
Given ⟨M, x⟩ to U, we want to construct ⟨M'⟩ to FIN such that if M accepts x, then M' accepts a finite language and if M doesn't accept x, then M' accepts an infinite language.

Note: There's a problem here. Normally, if M doesn't accept x, M' accepts a subset of what it would accept if M accepts x. (See below). However, in this case, if M doesn't accept x, M' accepts an infinite language while if M accepts x, M' accepts only a finite language.

f on input ⟨M, x⟩ does the following:

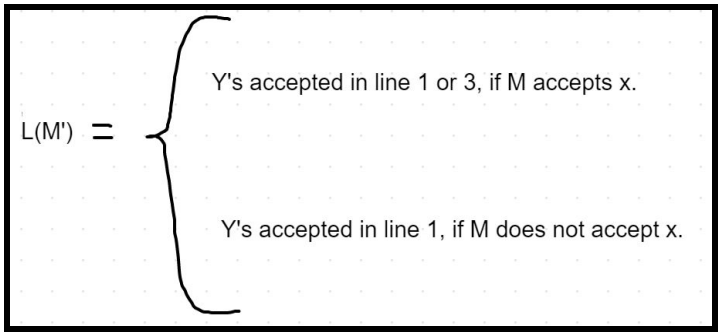
1. Define M'. M' on input y does the following:
    a. Run M on x for |y| steps.
    b. If M accepts x, in at most |y| steps, then M' reject.
    c. Else, M' accept.
2. Return ⟨M'⟩

Note: Because of line 1a., we don't have the problem that M runs on x forever.

If M accepts x, there exists a k, such that M accepts x after k steps. This means that M' accepts y if $|y| < k$ and rejects y if $|y| \geq k$. Hence, $L(M') = \{y \mid |y| < k\} \leftarrow$ finite. Therefore, $\langle M' \rangle \in$ FIN.

If M does not accept x, then $\forall$ k, M does not accept x in k steps. Therefore, $\forall\ y \in \Sigma^*$, M' accepts y (in line 1c.). Therefore, $L(M') = \Sigma^* \leftarrow$ infinite. Therefore, $\langle M' \rangle \notin$ FIN.

- Here is the general pattern for constructing "M' on input y does the following":
    1. M' might accept a certain input.
    2. Run M on x.
    3. If M accepts x, then maybe accept some more y's.
    4. Else, reject/loop

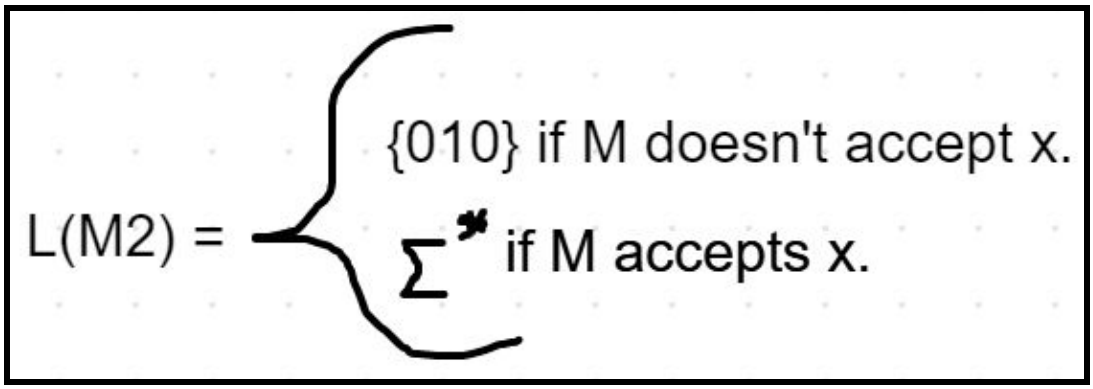


Notice that if M does not accept x, then M' only accepts a subset of what it would accept if M accepts x.

- **Theorem 5.4:** Let EQUIV = {⟨M1, M2⟩ | L(M1) = L(M2)}. EQUIV is not recognizable.

  Proof:
  The standard approach is to show that $\neg U \leq_m$ EQUIV.
  Given ⟨M, x⟩ as to $\neg U$, we want to construct ⟨M1, M2⟩ to EQUIV, such that if M does not accept x then L(M1) ≡ L(M2) and if M accepts x, then L(M1) ≠ L(M2).

  We want to fix M1 to take a specific string, which will be 010.
  I.e. L(M1) = {010}
  I.e. M1 is a TM that only accepts the string 010 and nothing else.



  f on input ⟨M, x⟩ does the following:
  1. Define M1. M1 on input y does the following:
     a. If y = 010, then accept.
     b. Else, reject.
  2. Define M2. M2 on input z does the following:
     a. If z is 010, then accept.
     b. Run M on x.
     c. If M accepts, then M2 accepts.
     d. Else, M2 rejects.
  3. Returns ⟨M1, M2⟩

  Alternative Proof:
  Here, we will prove that E $\leq_m$ EQUIV, where E = {⟨M⟩ | L(m) = Ø}.
  ⟨M⟩ ∈ E iff ⟨M, $M_{ø}$⟩ ∈ EQUIV, where $M_{ø}$ is any TM that accepts no string.
  This proof would be easier to prove than the first proof.
- **Theorem 5.5:** Let SUBSET = {⟨M1, M2⟩ | L(M1) ⊆ L(M2)}. SUBSET is not recognizable.

  Proof:
  ⟨M⟩ ∈ E iff ⟨M, $M_{ø}$⟩ ∈ SUBSET, where $M_{ø}$ is any TM that accepts no string.

Rice's Theorem:

- **Theorem 5.6:** **Rice's theorem** states that if P is a nontrivial property of recognizable languages, then L_P is undecidable.
  L_P = {⟨M⟩ | L(M) ∈ P}
  L_P is the set of TM codes whose language has property P.
- A **trivial property** of a language is a property that all languages have or no languages have.

- A **nontrivial property** of a language is a property such that there is at least one language that satisfies the property and at least one language that does not.
- A **property** P of languages is simply a set of languages.
  E.g. The set of finite languages is a property.
  E.g. The set of infinite languages is a property.
  E.g. The set of finite languages that have an odd number of strings in them is a property.
- Some trivial properties are:
  - The empty set of languages. Denoted as Ø.
    Ø is a language which has no strings. (No language has this property.)
  - The set of all languages. (Every language has this property.)

  Note: There is a difference between the following properties:
  1. Ø This means that the language has no strings. It is a trivial property.
  2. {Ø} This means that the language contains nothing. It is a nontrivial property.

  Ø is a language containing no string. {Ø} is a language containing exactly one string, the empty string, which has length 0.
- **Proof:**
  Let P be a nontrivial property of recognizable languages.
  It suffices to prove that $U \leq_m L_P$.
  Either $\{Ø\} \in P$ or $\{Ø\} \notin P$.

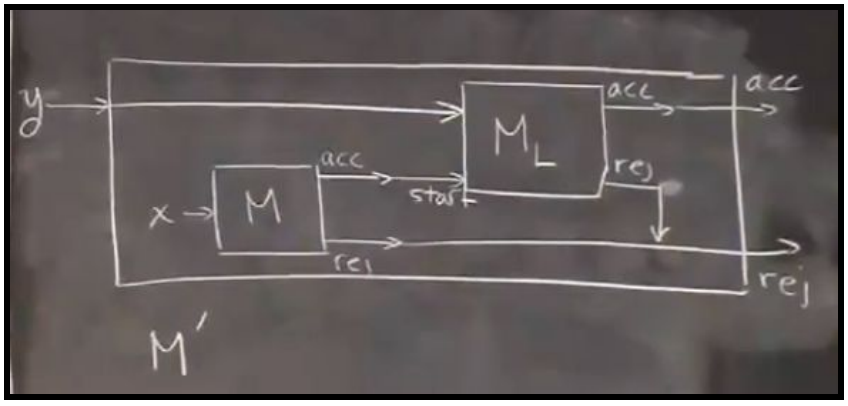
  <u>Case 1: $\{Ø\} \notin P$</u>
  Since P is nontrivial, some recognizable language, L, is in P.
  I.e. $L \in P$
  $L \neq Ø$ because L belongs to P and Ø doesn't belong to P.
  Let M_L be a TM such that $L(M_L) = L$.
  I.e. M_L recognizes L.
  We want a mapping reduction from U to L_P.

  Given ⟨M,x⟩ to U, we want to construct a machine M' to L_P such that if M accepts x then L(M') has property P and if M does not accept x, then L(M') does not have property P.

  We will use L as an example of a language that has property P and we will use the empty language, L = Ø, as a language that doesn't have property P.

  f on input ⟨M,x⟩ does the following:
  1. Define M'. M' on input y does the following:
     a. Run M on x.
     b. If M accepts, then
     c. Run M_L on y
     d. If M_L accepts, then accept.
     e. Else, reject
     f. Else, reject
  2. Return ⟨M'⟩

This is a diagram of the proof:



If M accepts x, then L(M') = L. This means that $\langle M' \rangle \in L_P$.
If M does not accept x, then L(M') = Ø. No string is accepted. This means that $\langle M' \rangle \notin L_P$.

Case 2: $\{\emptyset\} \in P$
Consider $\neg P$, the complement of property P.
$\neg P = \{L \mid L \notin P\}$
I.e. $\neg P$ is the set of languages that do not have property P.
Since $\{\emptyset\} \in P$, it follows that $\{\emptyset\} \notin \neg P$.
Now, we can apply case 1 to $\neg P$.
By case 1, $L_{\neg P}$ is undecidable.
By theorem 3.3, which states that "If L is a decidable language, then its complement is also decidable. I.e. The set of decidable languages is closed under complementation.", we know that if language L is undecidable, then its complement, $\neg L$, is also undecidable.
Hence, we know that $\neg L_{\neg P}$ is also undecidable.
$\neg L_{\neg P}$ is the set of languages that have the property P.
I.e. $\neg L_{\neg P}$ is simply L_P.
So L_P is undecidable.

Definitions:

- A language L is **recognizable** iff there exists a Turing Machine (TM) M such that L(M) = L. Here, M is called a **recognizer**.
  Note: M will halt and accept only the strings in that language. For strings not in that language, M will either reject it, in which it will halt, or M will loop. With recognizers, there is no requirement to halt.
  Note: On a given language L, a recognizer will either
  a. Halt and accept L or
  b. Halt and reject L or
  c. Loop on L
  Note: If a TM, M, doesn't accept a language, L, it doesn't mean that it rejects L. It could reject L or it could loop on L
- A language is **decidable** iff there exists a TM M such that L(M) = L and M halts on every input. Here, M is called a **decider**.
- U, the universal language, = {⟨M, x⟩ | M accepts x}.
  U is recognizable but not decidable.
- $\mathbf{M_u}$, the universal TM, takes ⟨M, x⟩ as input and simulates M on x.
- The halting problem, denoted as **H**, is H = {⟨M, x⟩ | M halts on x}.
  H is recognizable but not decidable.

General Turing Reduction:

- Reduction allows us to easily prove more languages are undecidable or unrecognizable.
- Let P and Q be languages.
- P **Turing-reduces** to Q, denoted as $P \leq_T Q$, if there exists an algorithm for P that uses an algorithm for Q as a "black box".
- Here are the general steps to prove that L is undecidable by using reduction:
  1. Assume L is decidable.
  2. Therefore, there is a TM M1 that decides it.
  3. Show that we can construct a TM M2 that uses M to decide U or some other undecidable problem.
  4. Since this contradicts that U is undecidable, L is undecidable.

Mapping Reduction:

- **Definition:** Let P and $Q \subseteq \Sigma^*$ be languages. P is **mapping-reducible** to Q, denoted as $P \leq_m Q$, iff there exists a computable function, $f : \Sigma^* \rightarrow \Sigma^*$, such that $x \in P$ iff $f(x) \in Q$. The function f is called the reduction of P to Q.
  Note: The function, f, does not have to be, and is usually not, onto.
  Note: The function, f, must be computable.
  To demonstrate a computable function, we will typically write a little program or describe in English how to perform the transformation that f is supposed to do.
  Note: f maps yes-instances of P to yes-instances of Q and no-instances of P to no-instances of Q.
- In general, when we are mapping-reducing language P to language Q, f should take an input of P as an input and output something that is an input of Q.
- **Theorems:**
  Suppose that $P \leq_m Q$
  1. **If Q is decidable, then P is decidable.**
     This is because we need to use a given solution to Q to solve P. If Q is decidable, then that means it halts on every input. Since P uses the output of Q on the input, P must halt on every input, too. Hence, P is decidable.

2. **If P is undecidable, then Q is undecidable.**
   This is because we need Q to solve P. If P isn't solvable, then neither is Q.
3. **If Q is recognizable, then P is recognizable.**
   This is because we need to use a given solution to Q to solve P. If Q is recognizable, then that means it either accepts, rejects or loops on every input. Since P uses the output of Q on the input, P must accept, reject or loop on every input, too. Hence, P is recognizable.
4. **If P is unrecognizable, then Q is unrecognizable.**
   This is because we need Q to solve P. If P isn't solvable, then neither is Q.
5. **If $P \leq_m Q$, then $!P \leq_m !Q$, where !P is the complement of P and !Q is the complement of Q.**

Note: If $P \leq_m Q$ and Q is unrecognizable, it doesn't tell us if P is recognizable or not.
Note: If $P \leq_m Q$ and Q is undecidable, it doesn't tell us if P is decidable or not.

- To prove that a language P is unrecognizable or undecidable, it suffices to prove that $U \leq_m P$, for undecidable, and $!U \leq_m P$, for unrecognizable.

Introduction to Post's Correspondence Problem (PCP):

- Let Γ be an alphabet such that $|\Gamma| \geq 2$.
- We are given a finite sequence of pairs of strings over Γ.
  I.e. (X1, Y1), (X2, Y2), …, (Xk, Yk) where $(Xi, Yi) \in \Gamma^*$.
- **Question:** Is there a finite sequence $i_1, i_2, \ldots, i_t \in \{1, \ldots, k\}$ s.t. $Xi_1Xi_2...Xi_m = Yi_1Yi_2...Yi_m$?
- **Note:** $I_1, I_2, \ldots, I_m$ does not need to contain all indices and may contain some repeatedly.
- **Example 1:** Let $\Gamma = \{0,1\}$ and k = 4.

i	**Xi**	**Yi**
1	11	111
2	101	0
3	10	01
4	0	01

This instance has a solution to the problem.

Consider the sequence 1, 2, 1.
X1X2X1 = 1110111
Y1Y2Y1 = 1110111
X1X2X1 = Y1Y2Y1
This is a solution to the problem.

Note: Solutions are not necessarily unique. Furthermore, you can have multiple solutions. You can take multiple repetitions of 1 sequence that gives a solution, and they would all be solutions.
I.e. Since 1,2,1 is a solution, then 1,2,1,1,2,1 is also a solution, and so on.

Note: The sequence 4, 2, 1, is also a solution to the problem.
X4X2X1 = 010111
Y4Y2Y1 = 010111
X4X2X1 = Y4Y2Y1

Note: The sequence 1, 3, 2, 1 is also a solution to the problem.
X1X3X2X1 = 111010111
Y1Y3Y2Y1 = 111010111
X1X3X2X1 = Y1Y3Y2Y1

- **Example 2:** Let Γ = {0,1} and k = 4.

i	**Xi**	**Yi**
1	11	111
2	101	1
3	10	01
4	0	01

This instance does not have a solution to the problem.

Proof:
First, we can never use the row i = 2. This is because when i = 1, 3, or 5, Xi and Yi have the same number of 0's whereas in i = 2, Xi and Yi have a different number of 0's. X2 has one more 0 than Y2. Hence, we can never get the same string if we use i = 2.

Second, by similar reasoning, we can never use the rows i = 1 and i = 4. For i = 1, Y1 has one more 1 than X1. For i = 4, Y4 has one more 1 than X1. Hence, we can never get the same string if we use either of i = 1 or i = 4.

Lastly, while the row i = 3 has the same number of 0's and 1's for X3 and Y3, their order is wrong. Hence, we can never get the same string if we use i = 3.

- **Theorem 5.7:** PCP is recognizable but not decidable.

Proof that PCP is not decidable:
We will prove that PCP is not decidable in 2 stages:

1. First, we will modify PCP. The modified version of PCP will be called MPCP. MPCP is the same as PCP except with the requirement that $i_1 = 1$.
   I.e. **Question:** Is there a finite sequence $i_1, i_2, \ldots, i_t \in \{1, \ldots, k\}$ s.t. $Xi_1Xi_2...Xi_m = Yi_1Yi_2...Yi_m$ and $i_1 = 1$?
   We will first prove that $U \leq_m$ MPCP.
2. We will then prove that MPCP $\leq_m$ PCP.

Proof of 1:
Given ⟨M, x⟩ to U, construct an instance P of MPCP, s.t. M accepts x iff P has an MPCP solution.
Recall that an instance of PCP/MPCP is a finite sequence of pairs of strings over Γ.
I.e. (X1, Y1), (X2, Y2), …, (Xk, Yk) where (Xi, Yi) $\in \Gamma^*$.
A **partial solution** to P is a sequence 1, $i_2, \ldots, i_m$ s.t. $X1Xi_2...Xi_m$ is a prefix of $Y1Yi_2...Yi_m$.
$X1Xi_2...Xi_m$ is referred to as the **top string**.
$Y1Yi_2...Yi_m$ is referred to as the **bottom string**.
Intuitively: In a partial solution, the top and bottom strings will be sequences of configurations of computations of TM M on x where the configurations are separated by a special symbol, #.
I.e. The configurations will look like this: #C0#C1#...

C0 = Q0x

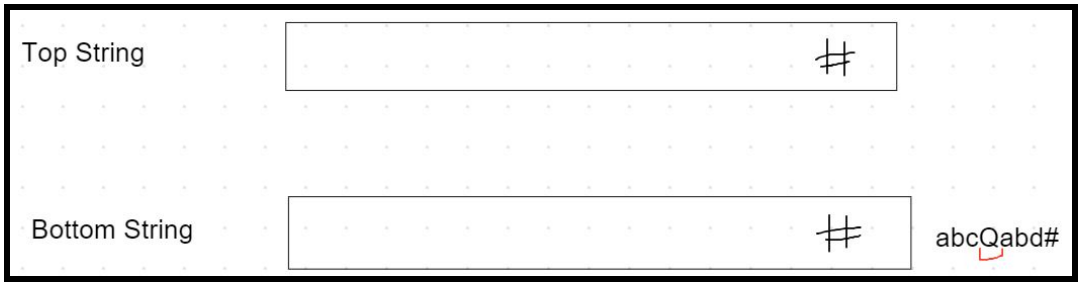
Furthermore, Ci $\vdash_M$ Ci+1.
I.e. We can go from Ci to Ci+1 in 1 move.
Furthermore, the top string is going to be 1 configuration behind the bottom string until and if we reach the accept state, Qa.
So, the partial solutions will look something like this:
Top String: #C0#C1#...#Ct-1#
Bottom String: #C0#C1#...#Ct-1#Ct#
Here, Ct is the accept state.

Lastly, If and when the bottom string finally reaches Qa, the top string will be allowed to catch up with the bottom string.

I will put pairs of strings in our instance of MPCP that will achieve the intuition behind the construction. These pairs of strings will be grouped and each group has a particular job.

Group 1:
Is the first pair and is (#, #Q0x#).
The y value is the initial configuration of TM M on input x.
Top String: #
Bottom String: #Q0x#

Group 2:
Is the second pair and will simultaneously copy portions of the last configuration in the bottom string to both the top and bottom strings.
Called “Copy Pairs”.
It needs (a, a), a ∈ Γ, and (#, #).
To understand what group 2 does, suppose I have a partial solution:



The items in the rectangles are the same, and the bottom string has the extra configuration abcQabd#. Note that the Qa doesn’t mean the accept state in this case; it’s just the state Q and the symbol a.
Now, we will copy parts of the configuration, ab bd#, to the top string and the bottom string simultaneously. The reason that we can’t copy the entire configuration, specifically the state and the symbol(s) surrounding it, is because the state could change the symbol and then it will move left/right.
I.e. In this case, we can’t copy cQa because the symbol a might get changed and then the state will move left/right. All other parts of the configuration can be copied as is.

If the state doesn't change symbol a and just moves left/right, here's what will happen:
Move Left: cQa → Qca
Move Right: cQa → caQ

Group 3:
This group is used to copy the state and the symbol(s) surrounding it to the top and bottom strings.

$\forall$ a, a', b $\in \Gamma$

This case is for if we're moving right and the symbol after the the state is not #.
It needs (qa, bp) if δ(q, a) = (p, b, R).
I.e. If we're at state q and reading symbol a, then change the a to b, go to state p and move 1 cell right.
I.e. xqay → xbpy

This case is for if we're moving right and the symbol after the the state is a #.
It needs (q#, bp#) if δ(q, ⊔) = (p, b, R).
I.e. If we're at state q and the next symbol is the blank symbol, change it to b, go to state p and move right.

This case is for if we're moving left and the symbol before the state is not #.
It needs (a'qa, pa'b) if δ(q, a) = (p, b, L).
I.e. If we're at state q and reading symbol a, then change the a to b, go to state p and move 1 cell left.
I.e. x'a'qay → x'pa'by

This case is for if we're moving left and the symbol before the state is a #.
It needs (#qa, #pb) if δ(q, a) = (p, b, L).
I.e. If we're at state q and reading symbol a, then change the a to b and go to state p. Since the state is at the leftmost position, it can't go left, so we don't move.

This case is for if we're moving left and the symbol after the the state is a # and we're replacing ⊔ with a non-blank symbol.
It needs (a'q#, pa'b#) if δ(q, ⊔) = (p, b, L).
I.e. If we're at state q and the next symbol is the blank symbol, change it to b, go to state p and move left.
Note: b is not the blank symbol. I.e. b ≠ ⊔.

This case is for if we're moving left and the symbol after the the state is a # and we're not replacing ⊔. However, the symbol preceding the state is not a blank symbol.
It needs (a'q#, pa'#) if δ(q, ⊔) = (p, b, L).
I.e. If we're at state q and the next symbol is the blank symbol, change it to b, go to state p and move left.
Note: b is the blank symbol. I.e. b = ⊔.
Note: a' is not the blank symbol. I.e. a' ≠ ⊔.

This case is for if we're moving left and the symbol after the the state is a # and we're not replacing ⊔. However, the symbol preceding the state is a blank symbol.
It needs (a'q#, p#) if δ(q, ⊔) = (p, ⊔, L).
I.e. If we're at state q and the next symbol is the blank symbol go to state p and move left. We don't change the blank symbol.
Note: b is the blank symbol. I.e. b = ⊔.
Note: a' is the blank symbol. I.e. a' = ⊔.

Group 4:
Allows the top string to catch up to the bottom string once the bottom string reaches the accept state, Qa.
Called "Catching Up Pairs"
(aQa, Qa)

(Q_aa, Qa)
∀ a ∈ Γ

Here's what the catching up pairs does:
Given:
Top String: #C0#C1#...#Cl-1
Bottom String: #C0#C1#...#Cl-1#Cl# where Cl is in the accept state.

Suppose that Cl = #abQacd#

I.e. We have
Top String: …#
Bottom String: …#abQacd

Step 1:
We will use group 2 to copy a to both the top and bottom strings.
Top String: …#a
Bottom String: …#abQacd#a

Step 2:
We will use group 4 to copy bQa to the top string and Qa to the bottom string.
Top String: …#abQa
Bottom String: …#abQacd#aQa

Step 3:
We will use group 2 to copy c to the top and bottom strings.
Top String: …#abQac
Bottom String: …#abQacd#aQac

Step 4:
We will use group 2 to copy d to the top and bottom strings.
Top String: …#abQacd
Bottom String: …#abQacd#aQacd

Step 5:
We will use group 2 to copy # to the top and bottom strings.
Top String: …#abQacd#
Bottom String: …#abQacd#aQacd#

Step 6:
We will use group 4 to copy aQa to the top string and Qa to the bottom string.
Top String: …#abQacd#aQa
Bottom String: …#abQacd#aQacd#Qa

Step 7:
We will use group 2 to copy c to the top and bottom strings.
Top String: …#abQacd#aQac
Bottom String: …#abQacd#aQacd#Qac

Step 8:
We will use group 2 to copy d to the top and bottom strings.
Top String: …#abQacd#aQacd
Bottom String: …#abQacd#aQacd#Qacd

Step 9:
We will use group 2 to copy # to the top and bottom strings.
Top String: …#abQacd#aQacd#
Bottom String: …#abQacd#aQacd#Qacd#

Step 10:
We will use group 4 to copy Qac to the top string and Qa to the bottom string.
Top String: …#abQacd#aQacd#Qac
Bottom String: …#abQacd#aQacd#Qacd#Qa

Step 11:
We will use group 2 to copy d to the top and bottom strings.
Top String: …#abQacd#aQacd#Qacd
Bottom String: …#abQacd#aQacd#Qacd#Qad

Step 12:
We will use group 2 to copy # to the top and bottom strings.
Top String: …#abQacd#aQacd#Qacd#
Bottom String: …#abQacd#aQacd#Qacd#Qad#

Step 13:
We will use group 4 to copy Qad to the top string and Qa to the bottom string.
Top String: …#abQacd#aQacd#Qacd#Qad
Bottom String: …#abQacd#aQacd#Qacd#Qad#Qa

Step 14:
We will use group 2 to copy # to the top and bottom strings.
Top String: …#abQacd#aQacd#Qacd#Qad#
Bottom String: …#abQacd#aQacd#Qacd#Qad#Qa#

Step 15:
We will use group 5 to copy Qa## to the top string and # to the bottom string.
Top String: …#abQacd#aQacd#Qacd#Qad#Qa##
Bottom String: …#abQacd#aQacd#Qacd#Qad#Qa##

Group 5:
Completes the matching.
(Qa##,#)

Now, we need to verify that M accepts x iff P has an MPCP solution.
(=>)
If M accepts x
→ There is an accepting sequence of configurations:

C0 $\vdash_M$ C1 $\vdash_M$ C2 $\vdash_M$ ... $\vdash_M$ Cl

where C0 is the initial configuration and Cl is the accepting configuration.
→ Can find a solution to MPCP where the top and bottom strings are the same and look like: #C0#C1#...#Cl#...#Qa#.
From #C0#C1#...#Cl#, we were using pairs in groups 1 - 3.
Afterwards, we were using pairs in groups 2, 4 & 5

(<=)
If an MPCP solution to P exists, the string corresponding to the solution must:

- Start with #Q0x#. This is because Group 1 puts that at the bottom.
- End with Qa##. This is because if it doesn't end with Qa##, the top string will have 1 less # than the bottom string. This is because the top string starts with 1 less # than the bottom string and every other time, the top string and bottom string has an equal number of #'s.

→ Use induction to prove an invariant:

1. The top and bottom strings in any proper partial solution have the form
   Top: #C0#C1#C2#...#C't-1
   Bottom: #C0#C1#C2#...#Ct-1#C't
   where C0, C1, …, Ct-1 are configurations of M and C't is a prefix of a configuration of M. C't-1 is a prefix of Ct-1.
2. If the state in Ct-1 is not the accepting state, then C't is a prefix of the configuration Ct s.t. Ct-1 $\vdash_M$ Ct.
3. If the state in Ct-1 is the accepting state, then the state in C't, if there is one, is also the accepting state.

→ The invariant implies that the string that corresponds to a solution to P is of the form #C0#C1#C2#...#Cl#...#Qa##, where Cl is the first configuration with an accept state and $\forall$ Ci $\vdash_M$ Ci+1, 0 ≤ i ≤ l.
→ M accepts x.

Proof of 2:
Given an instance P of MPCP, construct an instance $P^{@}$ of PCP s.t. P has an MPCP solution iff $P^{@}$ has a PCP solution.

Recall that an instance of PCP/MPCP is a finite sequence of pairs of strings over Γ.
I.e. (X1, Y1), (X2, Y2), …, (Xk, Yk) where (Xi, Yi) ∈ Γ^*.

Let @, $ ∉ Γ and @ ≠ $.
Given Z = a1a2...an, we define 2 strings, $z^{@}$ and $^{@}z$.
$z^{@}$ = a1@a2@...an@ and $^{@}z$ = @a1@a2…@an

For each pair (Xi, Yi) in P, we will define Vi = $Xi^{@}$ and Wi = $^{@}Yi$ in $P^{@}$.
We will also add a 0^{th} row in $P^{@}$, where V0 = @V1 and W0 = W1 and a row at the end in $P^{@}$, Vk+1 and Wk+1, where Vk+1 = $ and Wk+1 = @$.

Recall Example 1: Let Γ = {0,1} and k = 4. This will be our P.

i	**Xi**	**Yi**
1	11	111
2	101	0
3	10	01
4	0	01

Note that the sequence 1, 2, 1 was a solution to it. Hence, this is an instance of MPCP.

This will be our $P^{@}$.

i	**Vi**	**Wi**
0	@1@1@	@1@1@1
1	1@1@	@1@1@1
2	1@0@1@	@0
3	1@0@	@0@1
4	0@	@0@1
5	$	@$

Note that any matching you hope to achieve will have to start with V0 and W0. Otherwise, the first symbol will always be different.

Similarly, any matching you hope to achieve will have to start with V5 and W5. Otherwise, the last symbol will always be different.

If P has k pairs, then $P^{@}$ has k+2 pairs. The two pairs are (V0, W0) and (Vk+1, Wk+1).

P has an MPCP solution iff $P^{@}$ has a PCP solution.

(=>)

If P has an MPCP solution, then $P^{@}$ has a PCP solution.
Suppose that 1, i_2, i_3, …, i_m is a MPCP solution to P.
Claim: 0, i_2, i_3, i_m, i_{k+1} is a PCP solution to $P^{@}$.
Since 1, i_2, i_3, …, i_m is a MPCP solution to P, then $X1Xi_2Xi_3...Xi_m = Y1Yi_2Yi_3...Yi_m$.
Furthermore, $V1Vi_2Vi_3...Vi_m$ is almost the same as $W1Wi_2Wi_3...Wi_m$.
The differences are that the strings with Vi have a @ at the end and the strings with Wi have a @ at the beginning.
To make them equal, simply put a @ in the beginning of $V1Vi_2Vi_3...Vi_m$ and a @ at the end of $W1Wi_2Wi_3...Wi_m$.
Now, $@V1Vi_2Vi_3...Vi_m = W1Wi_2Wi_3...Wi_m@$.
Recall that @V1 = V0, W1 = W0.
Furthermore, if we add a $ to the end of the strings with Vi and Wi, we get $@V1Vi_2Vi_3...Vi_m\$$ and $W1Wi_2Wi_3...Wi_m@\$$. Recall that $ = Vk+1 and @$ = Wk+1.
Furthermore, $@V1Vi_2Vi_3...Vi_m\$ = W1Wi_2Wi_3...Wi_m@\$$.
Hence, we can substitute @V1 for V0, $ for Vk+1, W1 for W0 and @$ for Wk+1.
Now, we have $V0Vi_2...Vi_{k+1} = W0W_2...W_{k+1}$.
Hence, 0, i_2, i_3, i_m, i_{k+1} is a PCP solution to $P^{@}$.

(<=)

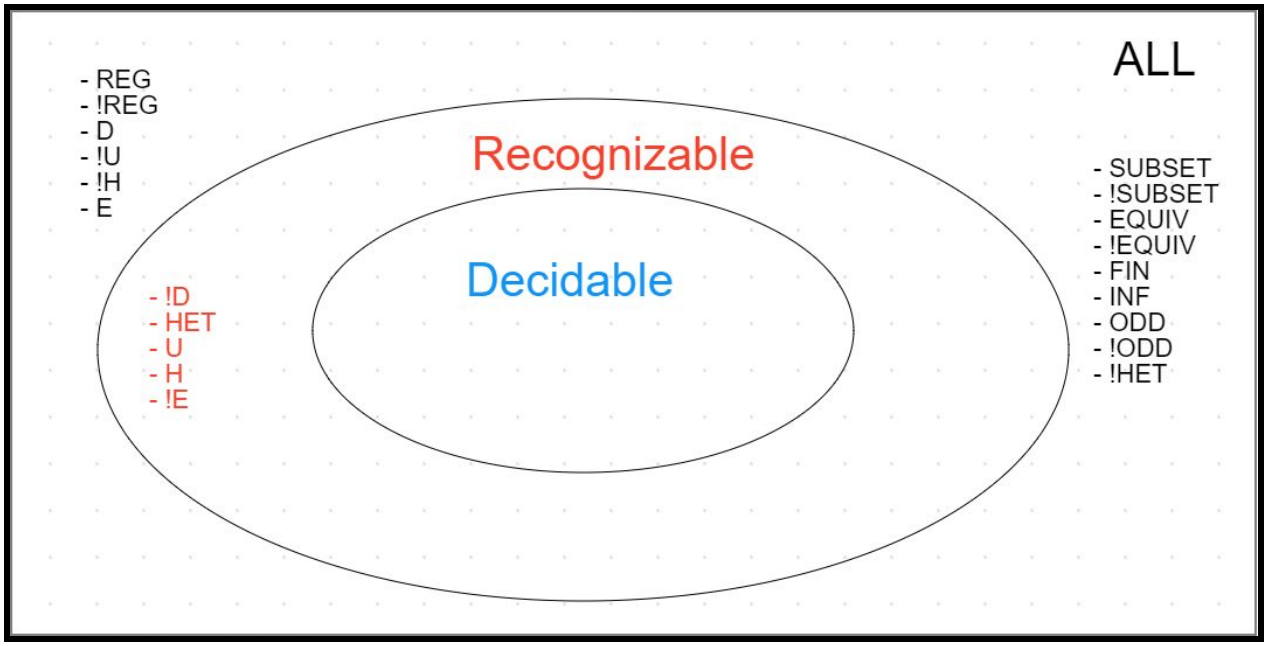
If $P^{@}$ has a PCP solution, then P has an MPCP solution.
Suppose i_1, i_2, …, i_m, i_{m+1} is a PCP solution to $P^{@}$.
It is obvious that i_1 has to be 0 because otherwise, the two strings will start with different symbols.
0 is the only row where V_i and W_i start with the same symbol.
Similarly, i_{m+1} has to be k+1.
k+1 is the only row where V_i and W_i end with the same symbol.
So, 0, i_2, …, i_{m+1}, k+1 is a PCP solution to $P^{@}$.
This means that $V0Vi_2Vi_3...Vi_mV_{k+1} = W0Wi_2Wi_3...Wi_mW_{k+1}$.
Recall that V0 = @V1, W0 = W1, V_{k+1} = $ and W_{k+1} = @$.
If we drop all the @ and $ from V_i and W_i, we get back $X1X2...Xi_m$ and $Y1Y2...Yi_m$.
$X1X2...Xi_m = Y1Y2...Yi_m$.
Therefore, 1, i_2, …, i_m, is an MPCP solution to P.

List of Languages

List of languages:

1. U = {⟨M, x⟩ | Turing machine M accepts input x} is undecidable but recognizable.
2. $\overline{U}$ = {⟨M, x⟩ | Turing machine M does not accept input x} is unrecognizable.
3. D = {⟨M⟩ | Turing machine M does not accept ⟨M⟩} is unrecognizable.
4. $\overline{D}$ = {⟨M⟩ | Turing machine M accepts ⟨M⟩} is undecidable but recognizable.
5. H = {⟨M, x⟩ | Turning machine M halts on input x} is undecidable but recognizable.
6. $\overline{H}$ = {⟨M, x⟩ | Turning machine M doesn't halt on input x} is unrecognizable.
7. E = {⟨M⟩ | Turing machine M accepts no input} is unrecognizable.
8. $\overline{E}$ = {⟨M⟩ | Turing machine M accepts some input} is undecidable but recognizable.
9. REG = {⟨M⟩ | Turing machine M accepts regular languages} is undecidable.
10. $\overline{REG}$ = {⟨M⟩ | Turing machine M does not accept regular languages} is undecidable.
11. HET = {⟨M⟩ | TM M halts on empty tape} is undecidable but recognizable.
12. $\overline{HET}$ = {⟨M⟩ | TM M doesn't halt on empty tape} is unrecognizable.
13. ODD = {⟨M⟩ | L(M) is finite and |L(M)| is odd} is not recognizable.
14. $\overline{ODD}$ = {⟨M⟩ | L(M) is infinite or |L(M)| is even} is not recognizable.
15. FIN = {⟨M⟩ | L(M) is finite} is not recognizable.
16. INF = {⟨M⟩ | L(M) is infinite} is not recognizable. (INF = $\overline{FIN}$)
17. EQUIV = {⟨M1, M2⟩ | L(M1) = L(M2)} is not recognizable.
18. $\overline{EQUIV}$ = {⟨M1, M2⟩ | L(M1) ≠ L(M2)} is not recognizable.
19. SUBSET = {⟨M1, M2⟩ | L(M1) ⊆ L(M2)} is not recognizable.
20. $\overline{SUBSET}$ = {⟨M1, M2⟩ | L(M1) ⊈ L(M2)} is not recognizable.

Here is a picture showing if each of the above languages is decidable, recognizable or unrecognizable.



Running Time of TM:

- Let M be a decider. The **running time of M/the time complexity of M** is function T: N → N s.t. T(n) is the maximum number of steps that M takes on inputs of size n before it halts.
- **Big-O** is used to describe the worst case running time for a given algorithm. It provides an upper bound.
  O(n) means that it takes at most time proportional to n.
- **Big-Ω** is used to describe the best case running time for a given algorithm. It provides a lower bound.
  Ω(n) means that it takes at least time proportional to n.
- **Big-Θ** is used to describe a tight bound for the running time for a given algorithm.
  T(n) is said to be in Θ(f(n)) if it is both in O(f(n)) and in Ω(f(n)).
  Θ(n) means that it is within a constant of n.
- Recall that the different variations of TMs, regular TMs, multi-tape TMs, NTMs, are all equivalent. However, when we are counting time, it does matter which type of TM we use.
- E.g.
  Consider the following language L = $\{0^k1^k \mid k \in N\}$.
  We want to have a recognizer for this language.
  Here is what the recognizer does:
  Given the input on the tape, the recognizer:
  1. Scans the entire input to check if there are any 0's after a 1.
  2. If there isn't:
     a. It goes back to the start of the tape.
     b. It marks the 0.
     c. It scans to the right until it sees a blank symbol, meaning that it has scanned the entire tape, or that it sees an unmarked 1.
     d. If it sees an unmarked 1, it marks it, checks if there are any more unmarked 1's and then scans to the left until it hits the last unmarked 0.
     e. If there are no more unmarked 0's, but there are still unmarked 1's, then reject.
        If there are unmarked 0's but no more unmarked 1's, then reject.
        If there are no more unmarked 0's and 1's, accept.
        Repeat steps b to e until it halts.
  3. If there is, reject.

  The running time of this algorithm, $T_1(n)$, is $\Theta(n^2)$.
  $T_1(n) = \Theta(n^2)$
  If there is a 0 after a 1, we will find it after n moves at most. This is the first sweep.
  If there is no 0 after a 1, to compare each pair of unmarked 0 and 1, will take about n/2 steps.
  There will be about n/2 of these pair matchings.
  $(n/2)*(n/2) = n^2/4$
  Hence, it takes $\Theta(n^2)$ time.

However, we can come up with another recognizer that recognizes this language faster.
Here is what the new recognizer does:
Given the input on the tape, the recognizer:

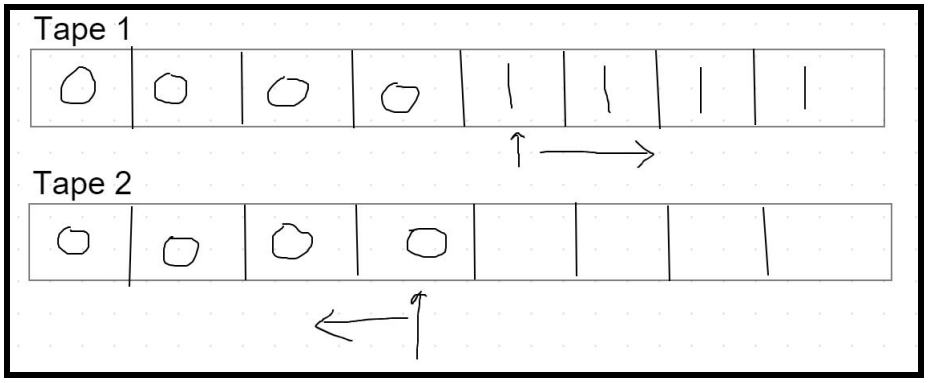
1. Scans the entire input to check if there are any 0's after a 1.
2. If there isn't:
    a. It goes back to the start of the tape.
    b. Starting from the first unmarked 0, it marks every 0 and every other 1.
    c. Repeats steps a & b until it either accepts or rejects. It accepts if there are no more 0's and 1's to mark. It rejects if there are still more 0's or 1's to mark, but no more of the opposite type to mark.
    I.e. There are 0's left but no 1's left or there are 1's left but no 0's left.
3. If there is, reject.

The running time of this algorithm, $T_2(n)$, is $\Theta(n\log(n))$.
Note: There is no standard TM that can recognize the language in fewer than $\Theta(n\log(n))$ steps.

Now, instead of using a standard TM, we will use a multi-tape (2-tape) TM.
Here is what the multi-tape recognizer does:
Given the input on the first tape, the recognizer:

1. Scans the entire input to check if there are any 0's after a 1.
2. If there isn't:
    a. It goes back to the start of the first tape.
    b. It copies all the 0's onto the second tape.
    c. Now, we will move right on tape 1 and move left on tape 2. If tape 1 arrives at the blank symbol at the same time tape 2 arrives at the beginning of the tape, then accept. Otherwise, reject.
3. If there is, reject.

Here's a picture of the 2 tapes.
Tape 1 contains the input.
Tape 2 contains all the 0's copied over from tape 1.
We start at the leftmost 1 in tape 1 and the rightmost 0 in tape 2.
At each step, we move right 1 cell in tape 1 and left 1 cell in tape 2.
If tape 1 arrives at the first blank symbol the same time that tape 2 arrives at the beginning of its tape, then accept. Otherwise, reject.



The running time of this algorithm, $T_3(n)$, is $\Theta(n)$.
We scanned the entire tape to make sure that no 0's came after 1's. (Takes n steps).
We copied all the 0's from the first tape to the second tape. (Takes n/2 steps).
We ran both tapes, tape 1 starting from the first 1 and tape 2 starting from the last 0. (Takes n/2 steps).
In total, it took $\Theta(n)$ steps.

Notice that while both single and multi-tape TMs can recognize the language, the single tape TM can do so in $\Theta(n\log(n))$ steps at best while the multi-tape TM can do so in $\Theta(n)$ steps.

Because of the fact that different variations of TMs have different running times, we want to make a very rough distinction between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time.

- We classify languages/decision problems into 2 categories:
    1. Those that are decidable/solvable in polynomial time, no matter what the exponent is.
       I.e. $O(n^k)$ where k is a constant.
    2. Those that are not decidable/solvable in polynomial time, no matter what the exponent is. These typically require exponential time.
       I.e. $O(2^n)$

Note: $O(n^{\log n})$ is not polynomial, because log n is not a constant, but is still better than $O(2^n)$.

Polynomial Time Thesis:

- **Polynomial Time Thesis:** A problem is efficiently computable/tractable/feasible iff there is a polynomial time algorithm that solves it.

Theoretical Justifications:

1. The running time is immune to models of computation up to polynomials.
   Recall theorem 2.1. It said that if a multi-tape TM takes m steps, then an equivalent regular TM takes $O(m^2)$ steps. Hence, if a multi-tape TM takes polynomial time, then its equivalent regular TM still takes polynomial time.
   Note: There is an important likely exception. If an NTM takes polynomial time, we don't know if its equivalent regular TM takes polynomial time.
2. The running time is immune to the details of encoding numbers up to polynomials, as long as we don't cheat. (The "don't cheat" part will be explained below.)
   I.e. Size of different reasonable encoding differs only by a polynomial amount.
   This is relevant because the running time is a function of the size of the input. If one way of counting the size of the input gets a very different result than another way, then we will get wildly different running times.
   However, if the size of different reasonable encoding differs only by a polynomial amount, it's ok because a polynomial of a polynomial is still a polynomial.
   E.g. 1: We want to represent the number n.
   The binary representation of n has $\text{floor}(\log_2 n)$ bits.
   The base r representation of n has $\text{floor}(\log_2 r)$ digits.
   $\log_2 n = (\log_2 r) * (\log_r n)$ where $\log_2 r$ is a constant.

E.g. 2: Consider a graph G = (V,E)
Let m = |E|
Let n = |V|
One way to calculate the size of the graph is m+n.
Another way to calculate the size of the graph is n^2.
The 2 ways are within a polynomial amount.
A third and more accurate way to calculate the size of the graph is the following:
Each node (vertex) is represented by $\log_2 n$ bits.
Hence, n nodes are represented by $n\log_2 n$ bits.
Furthermore, since each edge has 2 nodes, m edges are represented by $2m\log_2 n$ bits.
In total, we have $n\log_2 n + 2m\log_2 n$ or $\Theta((n+m)\log_2 n)$ bits.
Once again, the difference is within a polynomial amount.

Now, I will explain by "don't cheat". Don't cheat means that we don't put garbage in our encoding.
E.g. 3: Going back to our example 1, "We want to represent the number n", representing a number, x, in base 1 means that we write a symbol x many times.
If we want to represent 1 million in base 1, we have to write 1 million symbols.
Now, the difference between base 1 and base 2 representation of any number is no longer polynomial, it's exponential.

Empirical Justifications:

1. Polytime solvable problems that arise in practice usually have a "reasonable" exponent.
   For example, n^{100} can be really slow even with pretty small inputs, but in reality, most problems can be solved in n^2 or n^3 steps.
2. Furthermore, even when exponents are very high, they are still less than 2^n in most cases.
   For example, $n^{100} \leq 2^n \ \forall \ n \geq 1000$.
3. Algorithms that are not in polynomial time are usually in exponential time and exponential time algorithms are usually brute-force algorithms.

Complexity Class P:

- **P** is the set of languages/decision problems that can be decided by polynomial TMs.
  Note: NP does not mean non-polynomial time.
- Here are some examples of languages in P:
  - **E.g. 1:** SUM
    Instance: The representation/encoding of 3 numbers, x, y and z.
    I.e. Instance: $\langle x, y, z \rangle$ s.t. $x, y, z \in N$
    Question: Is $z = x + y$?
    Running time: Linear
  - **E.g. 2:** REACHABILITY
    Instance: $\langle G, s, t \rangle$ where G = (V,E) is a directed graph and s and t are nodes.
    $s, t \in V$
    Question: Does G have a path from s to t?
    Running time of BFS and DFS: $\Theta(m+n)$ where m = |E| and n = |V|

- **E.g. 3:** WEIGHTED SPANNING TREE
  Instance: ⟨G, wt, b⟩ where G = (V,E) is an undirected graph and wt is a weight function s.t. wt: E → N and b is a budget s.t. b ∈ N.
  Question: Does G have a spanning tree whose weight is no more than b?
- **E.g. 4:** PRIME
  Instance: ⟨n⟩ where n is a natural number.
  Question: Is n prime?

  Here is an algorithm, called PRIME-SOLVER, that solves this question:
  if n < 2 then return no
  for i = 2 to floor($\sqrt{n}$) do
  if n mod i = 0 then return no
  return yes

  Note: This algorithm is not in P.
  Here's the reasoning:
  - The loop runs Θ($\sqrt{n}$) or Θ($n^{1/2}$) times.
  - However, we want our program to be polynomial in respect to |⟨n⟩|. It doesn't take n bits to represent n. It takes $\log_2 n$ bits to represent n. Similarly, it takes $\log_2 n^{1/2}$ bits to represent $\sqrt{n}$.
  - $$\sqrt{n} = 2^{\log_2 \sqrt{n}} = 2^{\frac{1}{2}\log_2 n}$$
    Since $\log_2 n$ is the size of |⟨n⟩|, this algorithm is exponential in respect to |⟨n⟩|. Hence, it's not in P.

  Just because a problem is in P doesn't mean that every decider for that language is polynomial.
  PRIME is in P while PRIME-SOLVER is not in P.

- **Theorem 7.1:** Consider the language EXP = {⟨M,x⟩ | M accepts x in at most $2^{|\langle x \rangle|}$ steps}. EXP is decidable but is not in P.

  Proof that EXP is not in P:
  Suppose by contradiction that EXP is in P.
  Then, let EXP' = {⟨M⟩ | M accepts ⟨M⟩ in at most $2^{|\langle M \rangle|}$ steps}
  EXP' is in P.
  Let D = ¬EXP'. D = {⟨M⟩ | M does not accept ⟨M⟩ in at most $2^{|\langle M \rangle|}$ steps}
  If a problem is in P, then its complement is also in P.
  Hence, D is in P.
  Therefore, there exists a polytime TM M_D that decides D.
  Let P(n) = n^k be a polynomial upper bound on the running time of M_D.
  Let n_0 be a number s.t. ∀ $n \geq n_0$ $n^k \leq 2^n$.
  n_0 exists because every polynomial is less than or equal to an exponential for a sufficiently large n.
  Assume without loss of generality that $|\langle M_D \rangle| \geq n_0$.
  What does M_D do on input $\langle M_D \rangle$?

Case 1: M_D accepts $\langle M_D \rangle$.
→ This means that M_D does not accept $\langle M_D \rangle$ in $\leq 2^{|\langle MD \rangle|}$ steps.
→ But M_D on $\langle M_D \rangle$ halts in $|\langle M_D \rangle|^k$ steps and $|\langle M_D \rangle|^k \leq 2^{|\langle MD \rangle|}$.
→ M_D must reject $\langle M_D \rangle$. This is a contradiction.

Case 2: M_D rejects $\langle M_D \rangle$.
→ This means that M_D does not accept $\langle M_D \rangle$ in $\leq 2^{|\langle MD \rangle|}$ steps, since it does not accept $\langle M_D \rangle$ at all.
→ This means that M_D accepts $\langle M_D \rangle$. This is a contradiction.

This means that our original assumption that EXP is in P is false.
Hence, EXP is not in P.

- **Note:** Undecidable problems are not in P.
- **Note:** A polynomial of a polynomial is still a polynomial.

Decision, Search and Optimization Problems:

- **Travelling Salesman Problem (TSP-OPT):**
  Input: $\langle G, wt \rangle$ where $G = (V,E)$ is an undirected graph and wt is a weight function, $wt: E \rightarrow N$.
  Output: A minimum weight tour of this graph, if one exists.
  A tour means a cycle that visits each node exactly once.
  The weight of a tour is the sum of the weight of its edges.
- I can define a decision counterpart, **TSP-DEC**, that will help us solve TSP-OPT. This is a decision problem.
  Instance: $\langle G, wt, b \rangle$ where $G = (V,E)$ is an undirected graph, wt is a weight function, $wt: E \rightarrow N$, and b is a budget, $b \in N$.
  Question: Does G have a tour of $wt \leq b$?

  Solution:
  Given an oracle, which is a subroutine used as a black box, that solves TSP-DEC in one step, called TSP-DEC-SOLVER, that takes in $\langle G, wt, b \rangle$, construct an algorithm called TSP-OPT Solver, which takes in $\langle G, wt \rangle$, and solves TSP-OPT in polytime.

  TSP-OPT-SOLVER($\langle G, wt \rangle$)
  1. Use TSP-DEC-SOLVER($\langle G, wt, b \rangle$) and binary search to find the weight, b^*, of the optimal tour.

     For the binary search:
     The lower bound is $n*w_{min}$, where n is the number of edges and w_{min} is the minimum weight of any edge.
     The upper bound is $n*w_{max}$, where n is the number of edges and w_{max} is the maximum weight of any edge.
     To get the starting value for the budget, we take the middle value in the range from w_{min} and w_{max}.
     Run TSP-DEC-SOLVER($\langle G, wt, b \rangle$) to see if there's a tour that costs within that budget.
     If yes, we go in the direction of w_{min} to see if we can find a cheaper tour.
     Otherwise, we go in the direction of w_{max} to increase our budget.

2. Remove the edges one at a time from G and use TSP-DEC-SOLVER(⟨G, wt, b*⟩) to see if there exists a tour of wt b* in the remaining graph.
   If yes, continue.
   If no, put the edge back and continue.
3. Output edges left in G.

Proof that TSP-OPT-SOLVER has polynomial run time:

1. $|\langle G, wt\rangle| = O((m+n)\log n + m*\log_2 w_{max}) \rightarrow$ this is polytime w.r.t to n, m, $\log_2 w_{max}$
   $|\langle G\rangle| = O((m+n)\log n)$, as stated before
   $|\langle wt\rangle| = O(m*\log_2 w_{max})$ cause $m*w_{max}$ is the maximum total weight of the edges.

2. Time needed for steps 1 and 2 above =
   Number of calls to TSP-DEC-SOLVER * size of input

   The size of the input is $O((m+n)\log n + m*\log_2 w_{max} + \log_2(n*w_{max}))$.
   The input is ⟨G, wt, b⟩
   The worst b can be is $n*w_{max}$.
   Hence, the size of the input is $O((m+n)\log n + m*\log_2 w_{max} + \log_2(n*w_{max}))$.

   The number of calls to TSP-DEC-SOLVER is $O(\log_2(n*w_{max}) + m)$.
   This is because in step 1 from above, we make $\log_2(n*w_{max} - n*w_{min})$ calls to TSP-DEC-SOLVER.
   Setting $n*w_{min}$ to be 0, we get $\log_2(n*w_{max})$.
   Furthermore, in step 2, we call TSP-DEC-SOLVER on each edge of G.
   Since G has m edges, we call TSP-DEC-SOLVER m times.
   Hence, the number of calls to TSP-DEC-SOLVER is $O(\log_2(n*w_{max}) + m)$.

   In total, the time needed for steps 1 and 2 above =
   $((m+n)\log n + m*\log_2 w_{max} + \log_2(n*w_{max})) * (\log_2(n*w_{max}) + m)$,
   which is still in polynomial time.
3. The time needed for step 3 = $O(n*\log_2 n)$.
   This is because each edge is represented using $\log_2 n$ bits and there are n edges.

Hence, the time for TSP-OPT-SOLVER is polynomial in the size of the input assuming that TSP-DEC-SOLVER takes 1 step.
We assumed that TSP−DEC−SOLVER is constant time, but even if it did take polynomial time it would be fine. This is because a polynomial combined with a polynomial function is still polynomial.

- **Max Independent Set (MAX-IS):**
  Input: ⟨G⟩ where G = (V,E) is an undirected graph.
  Output: V' ⊆ V s.t. V' is an independent set (IS), meaning that there are no edges between any 2 nodes, of maximum size.
- I can define a decision counterpart, **IS-DEC**, that will help us solve MAX-IS. This is a decision problem.
  Instance: ⟨G, b⟩ where G = (V,E) is an undirected graph, and b is a budget, b ∈ N.
  Question: Does G have an IS of size ≥ b?

Complexity Class NP:

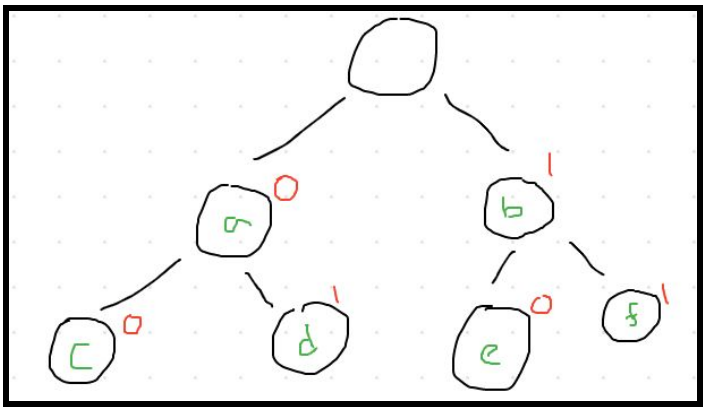
- **NP** is the set of languages/decision problems that can be decided by NTMs with polynomial running time.
- Recall that one of the theoretical justifications for the polynomial time thesis had an exception to it. That exception is that if an NTM takes polynomial time, we don't know if its equivalent regular TM takes polynomial time.
- The running time of an NTM M on input x is the maximum number of moves in any branch of M's computation of input x. Essentially, this is the height of the tree.
- The running time of an NTM M is the function T: N → N s.t. T(n) = max running time of M on any input of size n.
- **Theorem 8.1:** P ⊆ NP

  Proof:
  A TM is a special NTM where we always only have 1 choice.
  Hence, the set of languages that can be decided by a TM with polynomial running time is a subset of the set of languages that can be decided by an NTM with polynomial running time.

  Conjecture 8.2: P ≠ NP and hence P ⊂ NP
  This conjecture states that P is a proper subset of NP.
- **Thm 8.3:** Recall TSP-DEC and IS-DEC from lecture 7. TSP-DEC and IS-DEC are in NP.

  Proof that IS-DEC is in NP:
  We need to show/describe an NTM that runs in polynomial time and accepts yes-instances of the IS-DEC problem.

  A polytime NTM for IS-DEC does:
  1. Nondeterministically choose a set of b nodes of the given graph.
     Suppose there are n nodes in total in the graph.
     We can represent each node as a string of bits of length $\log_2 n$.
     Hence, a set of b nodes would have $b*\log_2 n$ bits.
     Nondeterministically choosing a set of b nodes means write down a set of b nodes.
     Here is an example of how we can nondeterministically choose a set of b nodes:



     The node with "a" in it means that the string starts with 0.
     The node with "b" in it means that the string starts with 1.

The node with “c” in it means that the string starts with 00.
The node with “d” in it means that the string starts with 01.
The node with “e” in it means that the string starts with 10.
The node with “f” in it means that the string starts with 11.
After $b*\log_2 n$ branches, we have all $b*\log_2 n$ possible strings.

2. Verify that no two nodes in the chosen set have an edge between them.
   If so, accept.
   Else, reject.

Step 1 takes polynomial time with respect to the size of the input, which is the number of nodes + the number of edges.
If we have m edges and n nodes, then the size of the input is m+n.
At most, b can be n. (b is n if we choose all n nodes.) So, at most, it takes $n*\log_2 n$ steps.
This is polynomial with respect to the size of the input.

Step 2 is completely deterministic.
Once we fix the set, we can look at every pair of nodes in the set to see if any two have an edge between them.
Hence, step 2 also takes polynomial time with respect to the size of the input.

Since each step takes polynomial time with respect to the size of the input, put together, it still takes polynomial time with respect to the size of the input.
Hence, IS-DEC is in NP.

- Here’s an alternative characterization of NP:

Consider TSP-DEC from lecture 7.
At the present time, there is no substantially better way of solving this problem than trying out every possible permutation of the n nodes in the graphs and seeing if there is a permutation that corresponds to a tour within the budget.
This method cannot be done in polynomial time.
However, if you are given a particular tour, you can easily verify if it is a proper tour and if it is within the budget. The difficult part is finding the tour.
There are a lot of problems with this property. It’s hard to find the solution, but easily to verify if a proposed solution is a valid solution or not.
All problems in NP have the property that if given a proposed solution which solves the problem in polynomial time, can be verified, in polynomial time, if it is an actual solution to the problem.

We will define a proposed solution to be a **certificate**.

TSP-DEC has a short and efficiently checkable certificate.
The certificate has a length of $n\log_2 n$ given there are n nodes in the graph. (short)
It just checks that the given nodes define a tour and that the sum of the weights of the edges is less than or equal to the budget. This takes polynomial time. (efficiently checkable)

Note: Certificates must be:

1. Comprehensive, meaning that all yes-instances have one.
2. Sound, meaning that all no-instances do not have one.
3. Short.
4. Efficiently checkable.

Definition: A polytime verifier for a language L is a polynomial time algorithm for a deterministic TM, V, s.t. $x \in L$ iff there exists a certificate for x, denoted as c_x, s.t. $|\langle c_x \rangle| = O(|\langle x \rangle^k|)$ for some constant k and V accepts $\langle x, c_x \rangle$.

You give the problem, x, and a certificate, c_x, to V and V will verify if that certificate is an actual solution to the problem.

Example: $\langle G, wt, b \rangle \in$ TSP-DEC iff there exists a list of edges s.t.

1. The list of edges form a tour of the graph.
2. The sum of the weights of the edges is at most b.

In this case our list of edges is our certificate. Steps 1 and 2 is what our TM V would do to check.

- **Theorem 8.4:** $L \in NP$ iff L has a polynomial time verifier.

Proof:

(=>)

If $L \in NP$, then it has a polynomial time verifier.

→ By definition, if a language is in NP, then there exists a polytime NTM that decides it.

→ Hence, there's a polytime NTM, M, that decides L.

→ If $x \in L$ and M is a decider for L, then there must be a sequence of configurations that accepts x. Hence, we define c_x as an accepting computation of M on x.

I.e. $c_x = C0 \vdash C1 \vdash ... \vdash Cl$ s.t. C0 is in the initial configuration of M on x and Cl is the accepting configuration of M on x.

→ $|\langle c_x \rangle|$ is short as it is polynomial in respect to $|\langle x \rangle|$

→ The verifier checks:

1. C0 is the initial configuration.
2. Cl is the accepting configuration.
3. $Ci \vdash Ci+1 \ \forall i \ 1 \leq i \leq l$

(<=)

If L has a polynomial time verifier, then $L \in NP$.

→ By definition, $x \in L$ iff there exists a certificate for x, denoted as c_x, s.t. $|\langle c_x \rangle| = O(|\langle x \rangle^k|$, for some constant k, and verifier V accepts $\langle x, c_x \rangle$ in polynomial time.

→ Construct a NTM M that on input x does the following:

1. Non-deterministically write certificate $\langle c_x \rangle$.
2. Run V on $\langle x, c_x \rangle$
    a. If V accepts, then accept.
    b. If V rejects, then reject.

M can do step 1 in polynomial time because c_x is short.
M can do step 2 deterministically in polynomial time.
Polynomial + polynomial is still polynomial.
Hence, the overall time it takes is polynomial.

- We have 2 ways of arguing that a language/decision problem is in NP:
    1. Show a polynomial time NTM for L.
    2. Show a polynomial time deterministic verifier for L.

  Note: The 2 ways are really the same thing.
- Here are 4 steps that we can use to prove that a language/decision problem is in NP:
    1. Show how to generate certificates.
       We can use an NTM to generate all of the strings and say that each one is a certificate.
    2. Argue that the certificate is short in size, meaning polynomial in size.
    3. Explain how the verifier works to validate input.
    4. Argue the verifier works in polynomial time. This is why we need the certificate to be short.

Satisfiability Problem (SAT):

- Suppose we have a proposition formula, Φ.
  A **truth assignment** tells us for every variable in the formula is it true or false.
  Once we know whether each variable is true or false, we know if the formula is true or false.

  E.g. Suppose $\Phi = (x \vee y) \wedge (\neg z \wedge \neg(x \vee \neg y))$
  Here is a truth assignment:
  Let x = true. Let y = true. Let z = false.
  Then:
  $(x \vee y) \rightarrow (T \vee T) \rightarrow T$
  $(\neg z \wedge \neg(x \vee \neg y)) \rightarrow (\neg F \wedge \neg(T \vee \neg T)) \rightarrow (T \wedge \neg(T \vee F)) \rightarrow (T \wedge \neg T) \rightarrow (T \wedge F) \rightarrow F$
  $T \wedge F \rightarrow F$
  Hence, in this case, Φ is False.

  Here's another truth assignment:
  Let x = false. Let y = true. Let z = false.
  Then:
  $(x \vee y) \rightarrow (F \vee T) \rightarrow T$
  $(\neg z \wedge \neg(x \vee \neg y)) \rightarrow (\neg F \wedge \neg(F \vee \neg T)) \rightarrow (T \wedge \neg(F \vee F)) \rightarrow (T \wedge \neg F) \rightarrow (T \wedge T) \rightarrow T$
  $T \wedge T \rightarrow T$
  Hence, in this case, Φ is True.

  A formula is **satisfiable** if there exists a truth assignment that makes the formula true.
  In our example above, Φ is satisfiable because there exists at least 1 truth assignment that makes the formula true.
- **SAT Decision Problem**
  Instance: ⟨Φ⟩, where Φ is a propositional formula.
  Question: Is Φ satisfiable?
  No one knows a polynomial time algorithm for this.

The most obvious algorithm is to test every truth assignment. However, if there are n variables, and each variable has 2 values (T or F), it would take 2^n time to test every truth assignment.

Since this is a decision problem, we need to encode the alphabet.
Some symbols we know that are in this alphabet are (,), ¬, ∧, ∨.
However, a problem arises with the variables. This is a problem because although an individual propositional formula has a finite number of variables, the set of all propositional formulas has an infinite number of variables.
We can fix this problem by representing each variable in binary with $\log_2 n$ bits, where n is the number of variables.

- **Theorem 8.5:** SAT ∈ NP

Proof:
The certificate in this case is a truth assignment that makes the formula true.
The verifier evaluates the formula given the certificate.

NP Completeness:

- In the world of computability, we divided languages/decision problems into “easy”, meaning decidable, and “hard”, meaning undecidable.
Previously, we established one “hard” problem, U. To prove that U is “hard”, we used diagonalization. Then, we proved that problem X is “hard” by proving that $U \leq_m X$.
In the world of complexity, we divide languages/decision problems into “easy”, meaning that the language is in P, and “hard”, meaning that the language is in !P∩NP.
Note: If P = NP, then !P∩NP is empty. (We don’t know if P = NP).
So, instead we use the term **NP Complete** or **NPC** to describe these problems.
NPC problems are believed to be hard and are the hardest problems in NP.

Here’s a chart to show dividing of languages in computability and complexity.

	Easy	**Hard**
Computability	Decidable	Undecidable
Complexity	In P	In NPC

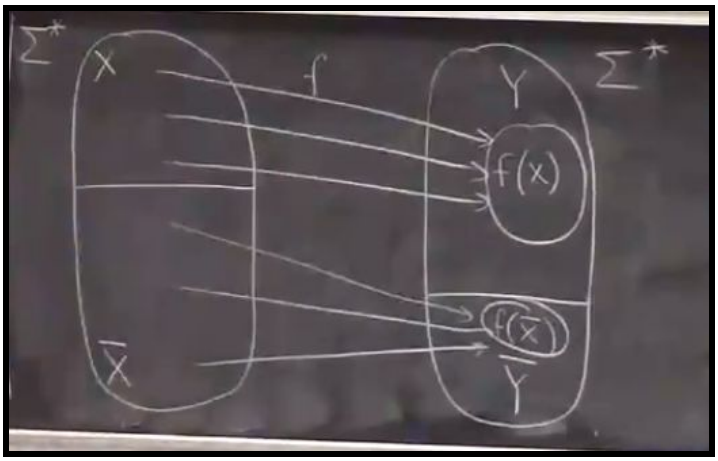
- There are 2 notations of time complexity reduction that roughly correspond to Turing reduction and Mapping reduction.
    1. **Cook Reduction:**
    - Let X, Y be problems (not necessarily decision problems). X **cook reduces** to Y , denoted as $X \rightarrow_P Y$ if there exists a polynomial time algorithm A that solves X given an oracle (blackbox subroutine) for Y where each use of the Y-oracle counts as 1 step.
      Note: While using the Y-oracle counts as 1 step per use, A has to prepare input(s) to the Y-oracle. This preparation is not part of the 1 step and is counted separately.
    - Cook reduction corresponds to Turing reduction.

- **Theorem 8.6:** If $X \rightarrow_P Y$ and there exists a polynomial time algorithm for Y, then there exists a polynomial time algorithm for X.

  Proof:
  If $X \rightarrow_P Y$, then there exists a polynomial time algorithm A that solves X, but instead of using a Y-oracle, it uses the polynomial time algorithm for Y.
  A polynomial of a polynomial is still a polynomial.
  Hence, the resulting algorithm is still polynomial time.

2. Karp Reduction:

- Let $X, Y \subseteq \sum^*$ be languages. X **karp-reduces/polytime reduces** to Y, denoted as $X \leq_P Y$, iff there exists a function $f: \Sigma^* \rightarrow \Sigma^*$ that can be completed in polynomial time s.t. $x \in X$ iff $f(x) \in Y$.
- Here's a diagram to help with the definition.



  f maps all the Yes-instances of X to a subset of the Yes-instances of Y.
  Similarly, f maps all the No-instances of X to a subset of the No-instances of Y.

- **Theorem 8.7:** If $X \leq_P Y$ and $Y \in P$ then $X \in P$.

  Proof:
  Similar to the proof of theorem 8.6.
  X-SOLVER on input x does 2 things:
  1. y = f(x)
  2. return Y-SOLVER(y)

  I.e.
  X-SOLVER(x)
  y = f(x)
  return Y-SOLVER(y)

- **Theorem 8.8:** $\leq_P$ is transitive.
  I.e. If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

  Proof:
  Given $x \in X$ iff $f(x) \in Y$ and $y \in Y$ iff $g(y) \in Z$, then $f(x) \in Y$ iff $g(f(x)) \in Z$

- **Definition:** Y is an NP-Complete (NPC) language iff
  a. $Y \in NP$
     I.e. Y is in NP.
  b. $\forall X \in NP, X \leq_P Y$
     I.e. Every problem, X, in NP must be polytime reducible to Y.
     This means that Y is the "hardest" in NP.
- **Theorem 8.9:** If Y is NPC and $Y \in P$ then P=NP.

  Proof:
  Assume that Y is NPC and $Y \in P$.
  Then, by definition, $\forall X \in NP, X \leq_P Y$.
  Then, by theorem 8.7, $X \in P$.
  Hence, every problem in NP is in P.
  Hence, P = NP.
- **Theorem 8.10:** If Y is NPC, $Z \in NP$ and $Y \leq_P Z$ then Z is NPC.

  Proof:
  Since $Z \in NP$, then part (a) of the definition of being in NPC holds.

  To show that part (b) of the definition of being in NPC holds:
  $\forall X \in NP, X \leq_P Y$. (This is because Y is NPC.)
  We also know that $Y \leq_P Z$. (This is by assumption.)
  Hence, by theorem 8.8, $\forall X \in NP, X \leq_P Z$. (This is because $\leq_P$ is transitive.)
  Hence, Z is in NPC.
- Later on we will show SAT is NPC with a specialized argument.
  After, we can use SAT as our "seed" to prove other problems are NPC.
  We will also prove that $SAT \leq_P IS$.

- **Theorem 8.11 (Ladner's Theorem):** If $P \neq NP$ then there are problems in $NP - (NPC \cup P)$.

(Cook's Theorem):

- **Theorem 9.1:** SAT is NP-Complete.

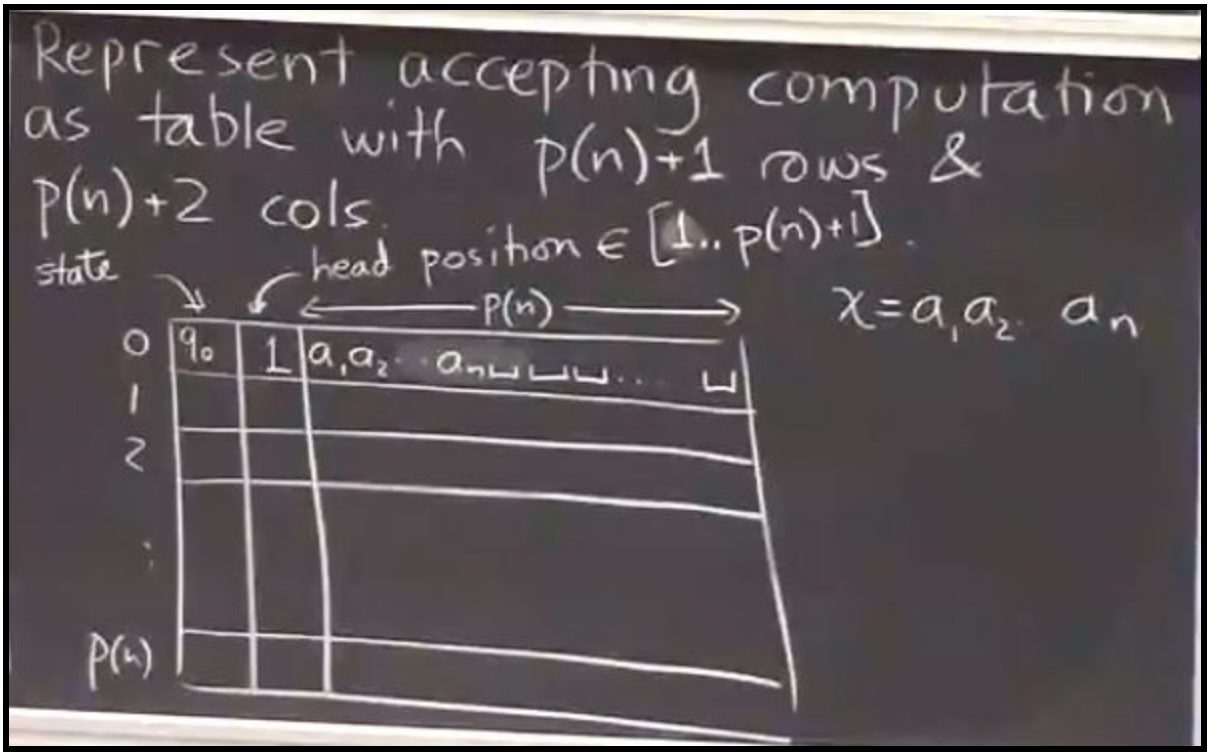
Proof:

1. **Proof that SAT ∈ NPC:**
   Note: This proof was done in lecture 8.
   The certificate in this case is a truth assignment that makes the formula true. The verifier evaluates the formula given the certificate.
2. **Proof that ∀ L ∈ NP, L ≤$_P$ SAT:**
   This part is tough because there is an infinite number of languages in NP.
   We can't prove each language ≤$_P$ SAT.
   Instead, we need to find something that all these languages have in common to help prove the reduction.
   We will call this item that all the languages have in common "a handle".
   For every language, L, in NP, the handle will be a NTM, M_L, that accepts L.
   Since the languages are in NP, we know that there exists a NTM that decides it in polynomial time.

   Given M_L and an input string x, where x ∈ ∑*, we will construct, in polytime, a function F_x s.t. x ∈ L iff F_x is satisfiable.
   We can rephrase the above statement like this: F_x is satisfiable means that there exists a truth assignment τ that satisfies F_x iff M_L on x has an accepting computation.

   Let p(n) be a polynomial bound on the running time of M_L, where n = |x|.
   This means that every computation of M_L on x takes at most p(n) steps, where n = |x|.
   M_L on x has an accepting configuration means that there exists a sequence of configurations C0⊢C1⊢…⊢Cl where C0 is the initial configuration, C0 = Q0x, and Cl is an accepting configuration.
   WLOG assume that p(n) ≥ n. (This means that the number of steps for the accepting computation is at least as long as the length of the input.)
   If p(n) < n, then we can just add useless steps to get p(n) ≥ n.
   If it was polynomial before, it's still polynomial.
   The tape on each C_t has at most p(n) symbols before the infinite number of blank symbols.
   Instead of thinking C0⊢C1⊢…⊢Cl as configurations, we will think of it as a table.
   There will be p(n)+1 rows, one row for each C_t.
   If l < p(n), I will still force the table to have p(n) + 1 rows. I will copy the last configuration to fill the remaining rows.
   Furthermore, there will be p(n)+2 columns. These columns will be used to represent the elements that are part of the configuration.
   We will put the state in the first column.
   We will put a number that represents the position of the head in the second column.
   We will put the p(n) symbols in the remaining p(n) columns, with one symbol per column.

Here's a diagram of the table:



The i^{th} row in the table is just another way to represent C_i.
We are changing the representation to make it more uniform.
ML = (Q, Σ, Γ, δ, Q0, QA, QR)
∀ q∈Q and ∀ t∈[0…p(n)], I will have a variable S_t^q, s.t. S_t^q = 1 iff at time t, the M_L is in state q. S_t^q = 0 otherwise.
∀ cell i∈[1…p(n)] and ∀ a∈Γ and ∀ t∈[0…p(n)], I will have a variable C_t^{ia}, s.t. C_t^{ia} = 1 iff at time t, cell i contains symbol a. C_t^{ia} = 0 otherwise.
∀ i∈[1…p(n)] and ∀ t∈[0…p(n)], I will have a variable H_t^i, s.t. H_t^i = 1 iff at time t, the head of the tape is on cell i. H_t^i = 0 otherwise.
The total number of variables is $\Theta(p^2(n))$.
We will use groups of formulas to help create F_x.
Group 1 (Coherence): At any time t, I cannot have:

a. M_L be in 2 different states.
b. M_L's head cannot be in 2 different cells.
c. Each cell cannot have 2 symbols.

Group 2 (Start Well): At time 0, M_L starts correctly.
Group 3 (End Well): At time p(n), M_L is in the accept state.
Group 4 (Move Well): In each step from t to t + 1:

a. Only the symbol under the head can change.
b. The state, head position, tape contents change as per δ.

For Coherence:

a. $\neg(S_t^q \wedge S_t^p)\ \forall p \neq q \in Q,\ \forall t \in [0...p(n)]$
   Note: We can express the above as: $\neg S_t^q \vee \neg S_t^p$
b. $\neg(H_t^i \wedge H_t^{i'})\ \forall i \neq i' \in [1...p(n)+1],\ \forall t \in [0...p(n)]$
   Note: We can express the above as: $\neg H_t^i \vee \neg H_t^i$
c. $\neg(C_t^{ia} \wedge C_t^{ib})\ \forall a \neq b \in \Gamma,\ \forall t \in [0...p(n)]$
   Note: We can express the above as: $\neg C_t^{ia} \vee \neg C_t^{ib}$

For Start Well:

M_L starts correctly means that at time 0,

a. The state is Q0. (S_0^{Q0})
b. The head is in cell 1. (H_0^1)
c. The tape is unchanged.
   ($C_0^{iai}\ \forall i \in [1...n]$) ← This means at time 0, cell i contains a_i for all i in 1 to n
   ($C_0^{i\_}\ \forall i \in [n+1...p(n)]$) ← This means at time 0, cell i contains a blank symbol for all i in n+1 to p(n).

For End Well:

We can represent "At time p(n), M_L is in the accept state." with $S_{p(n)}^{QA}$.

For Move Well:

a. $(C_t^{ia} \wedge \neg C_{t+1}^{ia}) \rightarrow H_t^i\ \forall i \in [1...p(n)+1],\ \forall a \in \Gamma,\ \forall t \in [0...p(n)]$
   Note: We can express the above as: $\neg C_t^{ia} \vee C_{t+1}^{ia} \vee H_t^i$.
b. $(S_t^q \wedge H_t^i \wedge C_t^{ia}) \rightarrow \vee(S_{t+1}^p \wedge H_{t+1}^{i+d} \wedge C_{t+1}^{ib})\ (p, b, R) \in \delta(q, a)$
   Note: We can express the above as:
   $\neg S_t^q \vee \neg H_t^i \vee \neg C_t^{ia} \vee (\vee(S_{t+1}^p \wedge H_{t+1}^{i+d} \wedge C_{t+1}^{ib})\ (p, b, R) \in \delta(q, a))$

$$d = \begin{cases} 1 & \text{if } D = R \\ -1 & \text{if } D = L \text{ and } i \neq 1 \\ 0 & \text{if } D = L \text{ and } i = 1 \end{cases}$$

I.e.
d = 1 if we move right.
d = -1 if we move to the left and are not in the leftmost cell.
d = 0 if we move to the left and are already in the leftmost cell.
$\forall p, q \in Q$ s.t. $q \neq QA,\ \forall i \in [1..p(n)+1],\ \forall t \in [0...p(n)]$
If q is in an accept state, there is no transition outside of an accept state.

If we are in QA, and there's empty rows in the table, we get
$(S_t^{QA} \wedge H_t^i \wedge C_t^{ia}) \rightarrow (S_{t+1}^{QA} \wedge H_{t+1}^i \wedge C_{t+1}^{ia})\ \forall i \in [1..p(n)+1],\ \forall t \in [0...p(n)]$, and $\forall a \in \Gamma$ or equivalently $\neg S_t^{QA} \vee \neg H_t^i \vee \neg C_t^{ia}) \vee (S_{t+1}^{QA} \wedge H_{t+1}^i \wedge C_{t+1}^{ia})$
Basically, we just copy everything from the row where we reach QA to all the empty rows.

Let F_x^1, F_x^2, F_x^3, F_x^4 be formulas in groups 1-4.
$F_x = F_x^1 \wedge F_x^2 \wedge F_x^3 \wedge F_x^4$
Claim: M_L accepts x iff F_x is satisfiable.
Proof:
(=>)
Suppose M_L accepts x.
Then, there exists a sequence of configurations C0, C1, …, Cl s.t.

- C0 is the initial configuration, C0 = Q0x
- $Ct \vdash Ct+1 \ \forall \ t \in [0..l-1]$
- Cl is the accepting configuration, Cl = Qa
- $l \leq p(n)$.

The contents of the table suggest a truth assignment that satisfies F_x.

(<=)
Suppose F_X is satisfiable.
That means that there exists a truth assignment that makes F_X true.
This truth assignment defines the rows of the table that correspond to an accepting computation.
Each row corresponds to a configuration of M_L. (Group 1)
The first configuration is the initial configuration of M_L on x. (Group 2)
The last configuration is an accepting configuration. (Group 3)
Each configuration follows from a previous by a legal move of M_L. (Group 4)

The reduction is polytime because the length of F_x is polynomial w.r.t |x| = n
Group 1's Formula is $O(p^3(n))$.
Group 2's Formula is $O(p(n))$.
Group 3's Formula is $O(1)$.
Group 4's Formula is $O(p^2(n))$.
Furthermore, each formula has size O(1).
$|F_x| = O(p^3(n))$ is polynomial w.r.t to |x|.

Examples of Other NPC Problems:

- We're going to prove that other problems, X, are NPC by showing $SAT \leq_p X$.
- However, before we do this we need to show that SAT is still NP C even if we restrict some formulas in a syntactic sense.
- **Definition:** F is in **Conjunctive Normal Form (CNF)** iff:
    a. It is a **literal**. A literal is a variable or the negation of a variable. OR
    b. It is a conjunction of **clauses**. A clause is a literal or a **disjunction** of literals.

  I.e. CNF is an $\wedge$ of $\vee$s, where $\vee$ is over variables or their negations (literals). An $\vee$ of literals is also called a clause.

  E.g. The following formulas are in CNF:
  $\neg X \rightarrow$ Is a literal
  $\neg X \wedge \neg Y \rightarrow$ Is a clause because it is a disjunction of 2 literals.
  $\neg x_1 \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \rightarrow$ Is a clause because there are disjunctions of literals.

  E.g. The following formulas are not in CNF:

¬(X ∨ Y) since an OR is nested within a NOT.

- **Note:** Every formula can be equivalently written as a formula in CNF.
- **CNF-SAT:**
- **Theorem 9.2:** CNF−SAT ∈ NPC.

Proof:

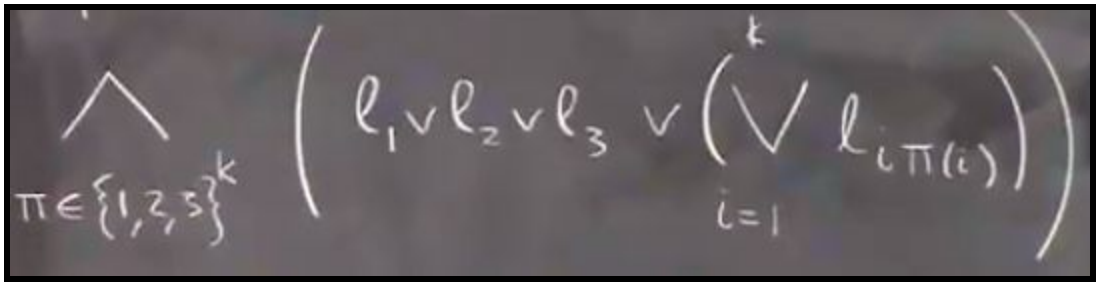
F_x is almost CNF.

The exception is in group 4.

We have $l_1 \vee l_2 \vee l_3 \vee (l_{11} \wedge l_{12} \wedge l_{13})) \vee ... \vee (l_{k1} \wedge l_{k2} \wedge l_{k3}))$ where l represent literals.

This is in DNF not CNF.

However, using distributive laws this is logically equivalent to



The size of this formula is $(3^k)(k+3)$ where $k \leq |Q||\Gamma|2 = O(1)$.

So, $(3^k)(k+3)$ is still a constant. $(3^k)(k+3) = O(1)$.

Put F_x in CNF as above.

The resulting formula has size polynomial w.r.t |x|.

Therefore, CNF−SAT ∈ NPC

- **Problem (3SAT):** CNF−SAT for formulas where each clause has ≤ 3 literals.
  Theorem 9.3: 3SAT ∈ NPC

Proof:

a. **Proof that 3SAT ∈ NP:**
   3SAT is a specific case of SAT which is in NP.
   Hence, 3SAT is in NP.
b. **CNF−SAT $\leq_p$ 3SAT:**
   Given any CNF formula F = C1 ∧ C2 ∧ ... ∧ Ck where Ci is a clause, construct in polytime a 3CNF formula F' = C'1 ∧ ... ∧ C'k.
   F' is satisfiable iff F is satisfiable.
   C'j is a 3CNF formula.
   |C'j| = O(|Cj|)
   If Cj has at most 3 literals, C'j = Cj.
   If Cj = l1 ∨ l2 ∨ … ∨ lm, where m ≥ 3, we introduce new variables z1, ..., zm−3.
   Then, let C'j = (l1 ∨ l2 ∨ z1) ∧ (¬z1 ∨ l3 ∨ z2) ∧ (¬z2 ∨ l4 ∨ z3) ∧ ... ∧ (¬zm−3 ∨ lm−1 ∨ lm).
   We are using Zi to chain the li.
   C'j is satisfiable iff Cj is satisfiable.
   Hence, F' is satisfiable iff F is satisfiable.
   |F'| = O(|F|) → CNF−SAT $\leq_p$ 3SAT

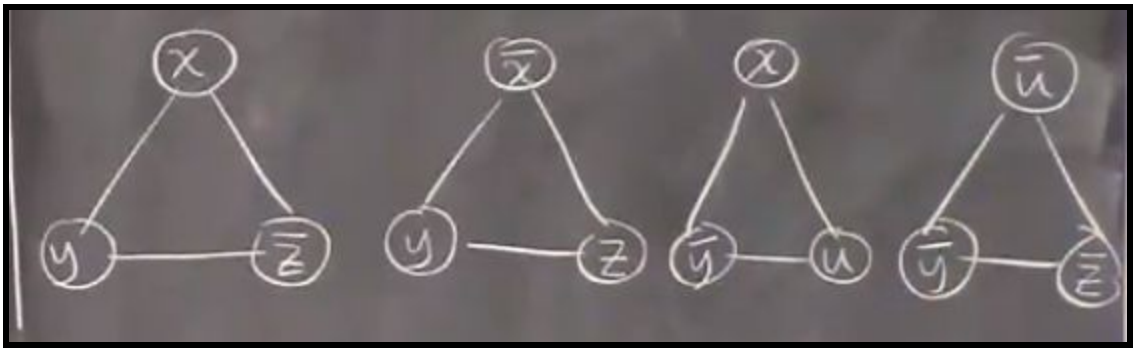
- **Fact:** 2SAT ∈ P

- **Problem (IS):**
  Instance: $\langle G, k \rangle$ where $G = (V, E)$ is an undirected graph and $k \in Z^+$.
  Question: Does G have an independent set $V' \subseteq V$ such that $|V'| \geq k$?

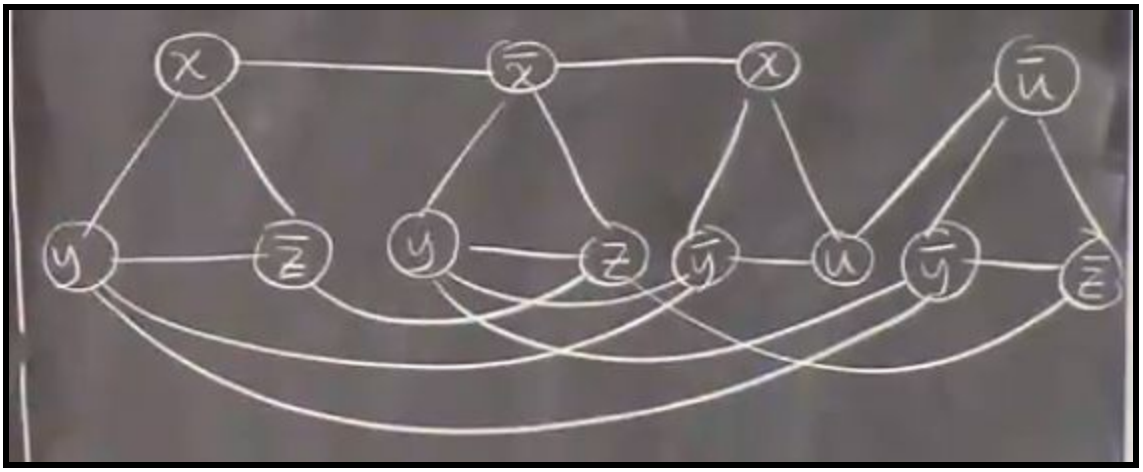
  Theorem 9.4: IS $\in$ NPC

  Proof:
  a. **Proof that IS $\in$ NP:**
     We have done this in lecture 8.
  b. **Proof that 3SAT $\leq_P$ IS:**
     Given 3CNF formula F, construct (in polynomial time in $|\langle F \rangle|$), an undirected graph G = (V, E) and $k \in Z^+$ s.t. F is satisfiable iff G has an IS of k nodes.
     Let $F = C1 \wedge ... \wedge Cm$.
     Let X1, …, Xm be variables in F.
     $Cj = (l_j^1, l_j^2, l_j^3)$ where l_j^t is a literal.

     E.g.
     Suppose we have the formula $F = (x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee u) \wedge (\neg u \vee \neg y \vee \neg z)$.
     For each clause construct a triangle.



     Selecting a node is equivalent to saying that the truth assignment for that literal is true.
     E.g. In the first triangle, if we choose $\neg z$, we're saying that $\neg z$ is true, meaning that z is false.
     We must restrict not being able to choose opposing literals (x and $\neg x$) by drawing an edge between them.
     This is because by choosing x in 1 triangle and $\neg x$ in another triangle, we're saying that we want x to be true in the first triangle and $\neg x$ to be true in the second triangle. This will cause a contradiction.
     This is shown in the picture below.



k = 4 is the number of clauses we have.

In general,
G = (V,E) where

V = $\{C_j^1, C_j^2, C_j^3 \mid 1 \le j \le m\}$

E = $\{\{\{C_j^1, C_j^2\}, \{C_j^2, C_j^3\}, \{C_j^3, C_j^1\} \mid 1 \le j \le m\} \cup$
$\{\{C_j^t, C_{j'}^{t'}\} \mid 1 \le j \ne j' \le m,\ 1 \le t, t' \le 3\}\}$ s.t. l_j^t and $l_{j'}^{t'}$ are opposite literals of the same variable.

k = m

Claim: F is satisfiable iff G has IS of size k = m.

Proof:
(=>)
Suppose F is satisfiable.
Let τ be a truth assignment that satisfies F.
τ makes true some literal, say l_j^{tj} of every clause Cj $1 \le j \le m$, $1 \le tj \le 3$.
Then $\{C_1^{t1}, C_2^{t2}, \ldots C_m^{tm}\}$ is an IS of G of size m = k.
They belong to different triangles and there are no edges between them.
If there is an edge between 2 of them, say C_i^{ti} and C_j^{tj}, then they are opposite literals of the same variable. Hence, τ would make one of them true and the other false. However, we are only choosing the literal that τ makes true, so we can't choose both C_i^{ti} and C_j^{tj}. Hence, there cannot be any edges between any 2 (C_x^{tx}, C_y^{ty}) in $\{C_1^{t1}, C_2^{t2}, \ldots C_m^{tm}\}$.

(<=)
Suppose G has an IS, V', of size m.
Because of the triangles, V' has exactly one node from each triangle.
Let V' = $\{C_1^{t1}, C_2^{t2}, ..., C_m^{tm}\}$.
We will define truth assignment τ as:

$$\tau(x) = \begin{cases} 1, & \text{if } \exists_j \text{ such that } \ell_j^{t_j} = x \\ 0, & \text{if } \exists_j \text{ such that } \ell_j^{t_j} = \overline{x} \\ 0/1 & \text{otherwise} \end{cases}$$

If l_j^{tj} = x, then we make x = 1 (x = true).
If l_j^{tj} = ¬x, then we make x = 0 (x = false), so that ¬x is true.
This is well-defined, meaning that there are no contradictory rules, that satisfies F because it solves every clause.

Remains to show that construction of ⟨G,k⟩ can be done in polynomial time in size of ⟨F⟩.
|V | = 3m
|E| = 3m + 3m(m − 1) = $O(m^2)$.
Therefore, the size of ⟨G,k⟩ is polynomial w.r.t |⟨F⟩|.
Therefore, ⟨G,k⟩ can be constructed from ⟨F⟩ is polynomial.
Therefore, 3SAT $\leq_P$ IS.
Therefore, 3SAT ∈ NPC.

- **Problem (CLIQUE):**
  Instance: ⟨G,k⟩ as in IS.
  Question: Does G have a clique of size k?
  A **clique** is a subset of nodes, V', s.t. there exists an edge between any 2 pairs of nodes in V'. V' ⊆ V.

  Theorem 9.5: CLIQUE is NPC.
  IS $\leq_P$ CLIQUE.
  Given G = (V, E), construct ¬G = (V, ¬E).
  ¬G is the complement of G.
  ¬G has an edge between 2 nodes iff G does not have an edge between those 2 nodes.
  Therefore G has an independent set iff ¬G has a clique.
- **Problem Vertex Cover (VC):**
  Instance: ⟨G,k⟩ as before.
  Question: Does G have a vertex cover of size k?
  A **vertex cover** is a set of nodes (V' ⊆ V) s.t. every edge has at least one endpoint in V'.

  Theorem 9.6: VC ∈ NPC
  G = (V, E) has an IS of size K iff G = (V, E) has a VC of size n − k, where n = |V|.
  If something is an IS, its complement must be a VC.

How to Prove B ∈ NPC:

1. Prove that B ∈ NP.
   You can do this with an NTM or certificate.
2. Choose problem A, which is known to be NPC.
3. Describe a polytime reduction of A to B, $A \leq_p B$.
   - Show how to transform any instance of A into an equivalent instance of B.
   - Argue that the transformation is polytime.

Exact Cover (XCOV):

- Instance: ⟨U, C⟩ where U is the universe set of elements and C is collection (set) of subsets of U.
- Question: Is there C' ⊆ C s.t.
  a. C' covers every element in U. (We call C' a **cover**.)
     I.e. The union of all subsets in C' is U.
  b. ∀ A, B ∈ C', s.t. A≠B, A∩B = Ø (We call C' an **exact cover**.)
     I.e. All of the sets in C' are disjoint.
     I.e. There is no element in C' that is in 2 different sets in C'.
     I.e. The intersection of any two subsets in C' should be empty. That is, each element of U should be contained in at most one subset of C'.
- **Theorem 10.1:** XCOV ∈ NPC.

Proof:

a. **Proof that XCOV ∈ NP:**
   Use a NTM to non-deterministically choose a subset of C and verify that it covers all the elements and no two sets in the chosen subset of C overlap.

b. **Show that CNF-SAT $\leq_p$ XCOV:**
   Given a CNF formula F = C1 ∧ … ∧ Cm, where each Cj is a clause, with variables X1, …, Xn, we will construct U and C s.t. F is satisfiable iff (U, C) has an exact cover.
   Let $Cj = l_j^1 \vee l_j^2 \vee \dots l_j^{kj}$, 1≤j≤m and l_j^t is a literal.

   For example, suppose F = (¬x1 ∨ ¬x2 ∨ x3) ∧ (¬x1 ∨ x2 ∨ ¬x3) ∧ (¬x1 ∨ x3) ∧ (x1 ∨ x2 ∨ x3).
   We have 3 variables, x1, x2 and x3.
   We have 4 clauses.

   Every variable gets an element in the universe.
   Every clause also gets its own element.
   Every literal inside the clause also gets its own element l_j^t.
   We define $U = \{xi \mid 1 \leq i \leq n\} \cup \{Cj \mid 1 \leq j \leq m\} \cup \{l_j^t \mid 1 \leq j \leq m,\ 1 \leq t \leq kj\}$.
   $\{xi \mid 1 \leq i \leq n\}$ represents the set of variables
   $\{Cj \mid 1 \leq j \leq m\}$ represents the set of clauses.
   $\{l_j^t \mid 1 \leq j \leq m,\ 1 \leq t \leq kj\}$ represents the set literals.

   Here are the sets in C:
   1. For each xi, 1 ≤ i ≤ n:
      a. $Pi = \{xi\} \cup \{l_j^t \mid l_j^t = xi,\ 1 \leq j \leq m,\ 1 \leq t \leq kj\}$
         Pi is the union of each variable and all the literals that are the same as the variable.

P1 = {x1, l_4^1} in our example. l_4^1 is the only literal that is the same as x1.

b. Ni = {xi} U {l_j^t |l_j^t = ¬xi , 1 ≤ j ≤ m, 1 ≤ t ≤ kj}
Ni is the union of each variable and all the literals that are the not of the variable.
N1 = {x1, l_1^1, l_2^1, l_3^1} in our example. l_1^1, l_2^1, l_3^1 are literals that are the not of x1.

2. For every clause Cj 1 ≤ i ≤ m:
S_j^t = {Cj, l_j^t} 1 ≤ t ≤ kj.
Each literal inside a clause is grouped with that clause.
E.g. S_1^1 = {C1, l_1^1}
3. For each l_j^t, 1 ≤ j ≤ m, 1 ≤ t ≤ kj:
L_j^t = {l_j^t}.
Each literal gets its own set.

Here's the intuition for coming up with this idea:

- First, to cover each of the clauses, Cj, you must choose only one S_j^t for each Cj. The idea of choosing the S_j^t is that l_j^t is a literal that will satisfy Cj.
E.g. Suppose that, from our example above, we want to use ¬x1 to make C1 true. Then, we choose S_1^1.
- Next, to cover the variables, if l_j^t is true, choose Ni and if l_j^t is false, then choose Pi.
E.g. Suppose that, from our example above, we want to use ¬x1 to make C1 true. Then, that means that x1 is false. We choose P1 instead of N1 because by choosing P1, we are preventing ourselves from using x1 to make another clause true later on. Since an exact cover forbids the same element in multiple subsets, by choosing P1 now, we are eliminating x1.

  Another way of looking at this is the fact that we need to cover each variable xi. Each variable xi is only in Pi and Ni. Suppose that l_j^t, which corresponds to xi, is true. To cover xi, we can only use Pi or Ni. If we use Pi, then l_j^t will be in 2 different sets, Pi and S_j^t, which is not allowed. Hence, we must choose Ni. The same reason applies for why we choose Pi if l_j^t is false.
- Lastly, to get the literals that we did not choose earlier, we select then from L_j^t.

c. Claim: F is satisfiable iff (U, C) has an exact cover.
Proof:
(=>)
Let τ be a truth assignment satisfying F.
Then let C' be defined as the following:

1. For each xi 1 ≤ i ≤ n:
   a. If τ(xi) = True, then put Ni in C'.
   b. If τ(xi) = False, then put Pi in C'.

2. For each clause Cj $1 \leq j \leq m$ let $l_j^{\tau j}$ be a literal such that $\tau(l_j^{\tau j}) = \text{True}$:
    a. If there are more than one literal that satisfies Cj then pick one arbitrarily.
    b. Then put $S_j^{\tau j}$ in C'.
3. For each l_j^t not covered by 1 or 2:
    a. Add L_j^t to C'.

Claim: C' is an exact cover.

a. Proof that C' covers all elements:
   This is clear as
   (1) includes all variables,
   (2) includes all the clauses and
   (3) includes all the leftover literals.
b. Proof that no two sets in C' intersect:
   The only possibility of intersection is between $S_j^{\tau j}$ and Pi or Ni.
   Suppose $l_j^{tj} \in S_j^{tj} \cap$ Ni.
   Since $l_j^{tj} \in$ Ni, $l_j^{tj} = \neg$xi.
   However, since $l_j^{tj} \in S_j^{tj}$, $l_j^{tj} = \text{True}$.
   This is a contradiction.
   If $\tau(\neg xi) = 1$, then $\tau(xi) = 0$ and Ni $\notin$ C'.

Hence, C' is an exact cover.

(<=)

Let C' be an exact cover of (U, C).
Define

$$\tau(x_i) = \begin{cases} 1 & \text{if } N_i \in \mathcal{C}' \\ 0, & \text{if } P_i \in \mathcal{C}' \end{cases}$$

These sets must contain at least one of them because these are the only sets that contain xi. It cannot contain both because then it would not be an exact cover.

Claim: τ satises F.
To prove this, it suffices to show that it satisfies every clause, Cj.
Proof:

- Let S_j^{tj} be the unique set in C' that contains Cj.
- $S_j^{tj} = \{Cj, l_j^{tj}\}$
- Suppose $l_j^t = xi$.
    - Then, C' does not contain Pi.
    - This means that C' contains Ni.
    - This means that $\tau(xi) = 1$.
    - This means that τ satisfies Cj.

- Suppose $l_j^t = \neg xi$.
  - Then, C' does not contain Ni.
  - This means that C' contains Pi.
  - This means that $\tau(\neg xi) = 0$.
  - This means that $\tau(xi) = 1$.
  - This means that τ satisfies Cj.

d. Argument for polynomial time:

The construction of ⟨U, C⟩ from ⟨F⟩ is polytime w.r.t |⟨F⟩|.
We just have to argue that |⟨U, C⟩| isn't too big.

$|\langle F\rangle| = n + \sum_{j=1}^{m} kj$

(n is the number of variables.)

($\sum_{j=1}^{m} kj$ is the total number of literals in all the clauses.
There are m clauses and each clause has kj literals.)

|⟨U, C⟩|

- $|\langle U\rangle| = n + m + \sum_{j=1}^{m} kj$, where m is the number of clauses.
- The number of all the sets in C = $\sum_{j=1}^{m} kj + 2\sum_{j=1}^{m} kj + 2n + \sum_{j=1}^{m} kj$

The first $\sum_{j=1}^{m} kj$ is for L_j^t.

The $2\sum_{j=1}^{m} kj$ is for S_j^t.

The 2n is for the xi in Pi and Ni.

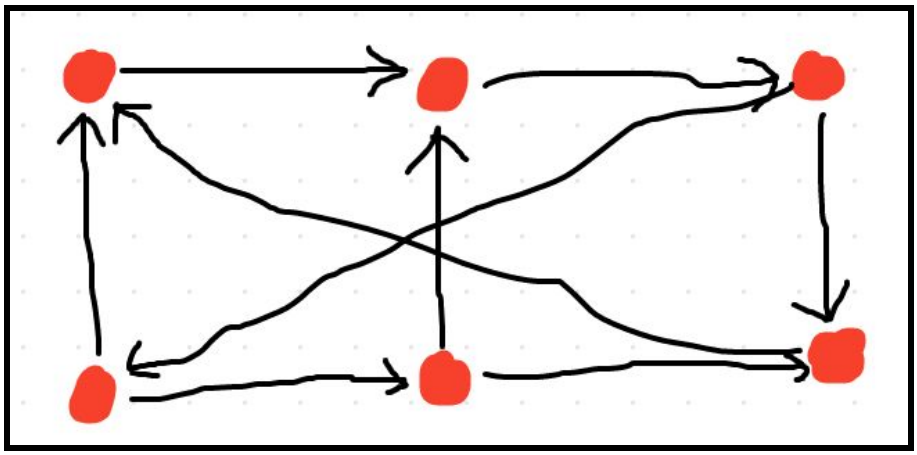
The final $2\sum_{j=1}^{m} kj$ is for each literal in Pi or Ni.

$= 2n + 4\sum_{j=1}^{m} kj$

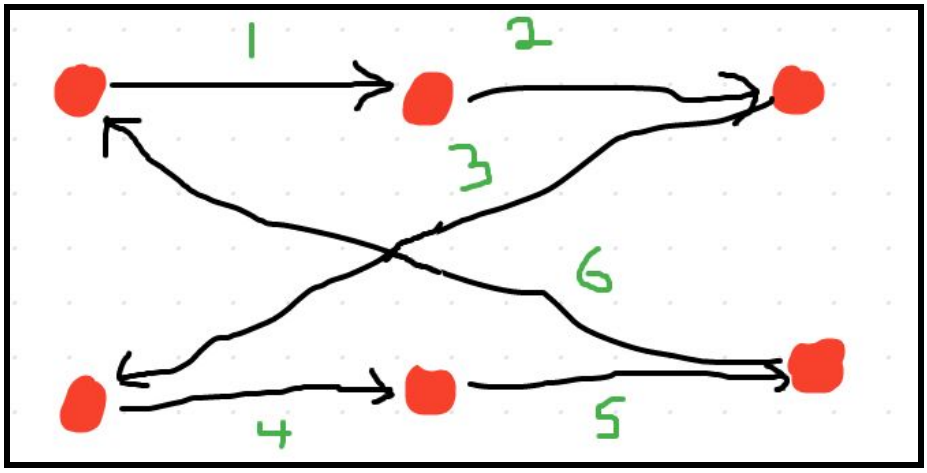
Hence, this is polynomial in |⟨F⟩|.

Directed Hamiltonian Cycle (DHC):

- Instance: ⟨G⟩, G = (V, E) is a directed graph.
- Question: Does G have a HC? A HC is a path that goes through all nodes exactly once.
  E.g. The graph below has a HC.



This is the HC:



- **Theorem 10.2:** DHC ∈ NPC

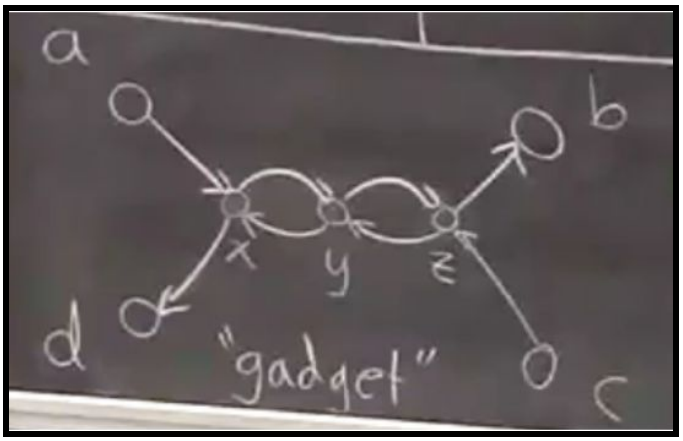
Proof:

a. **Proof that DHC ∈ NP:**
   A NTM guesses a path and a deterministic TM goes through the path and confirms it.

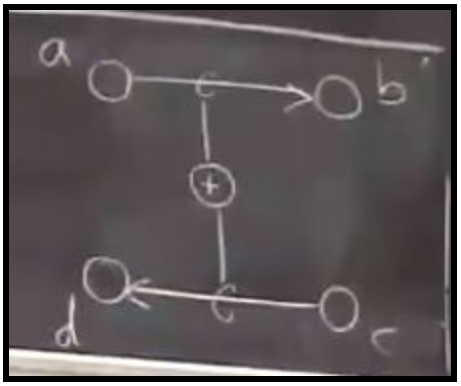
b. **Show that XCOV $\leq_p$ DHC:**
   We will use **gadgets** to show this.
   Gadgets are a sub construction that are embedded in the overall construction.
   We will define a gadget for this proof as follows:
   - We have 2 pairs of nodes (A,B) and (C,D) that connect to three nodes x, y and z.
   - Other nodes can be connected to A, B, C or D, but there are no other nodes that are connected to x, y or z.
   - If we have a HC in the larger graph the gadget is part of, it must contain one of either 2 paths:
     1. A → x → y → z → B We call this path A → B.

2. $C \rightarrow z \rightarrow y \rightarrow x \rightarrow D$ We call this path $C \rightarrow D$.

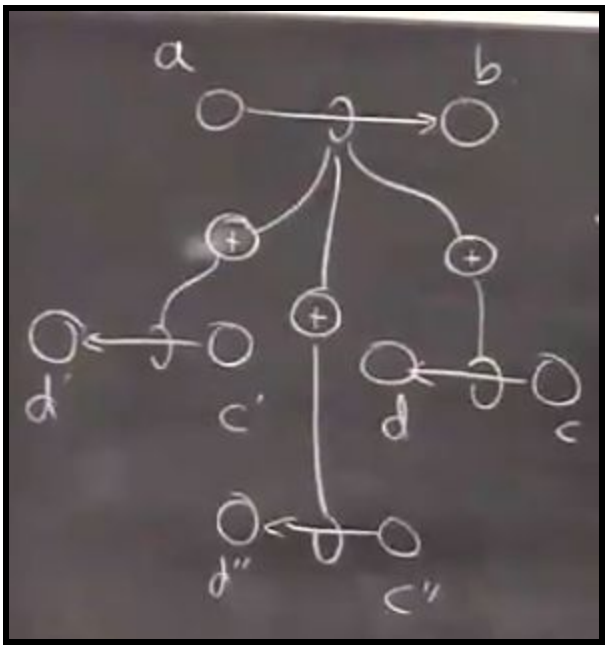
- This creates the equivalent of an xor in paths.
- Here is a picture of the gadget.



We will use the below diagram as a shorthand for the above diagram.

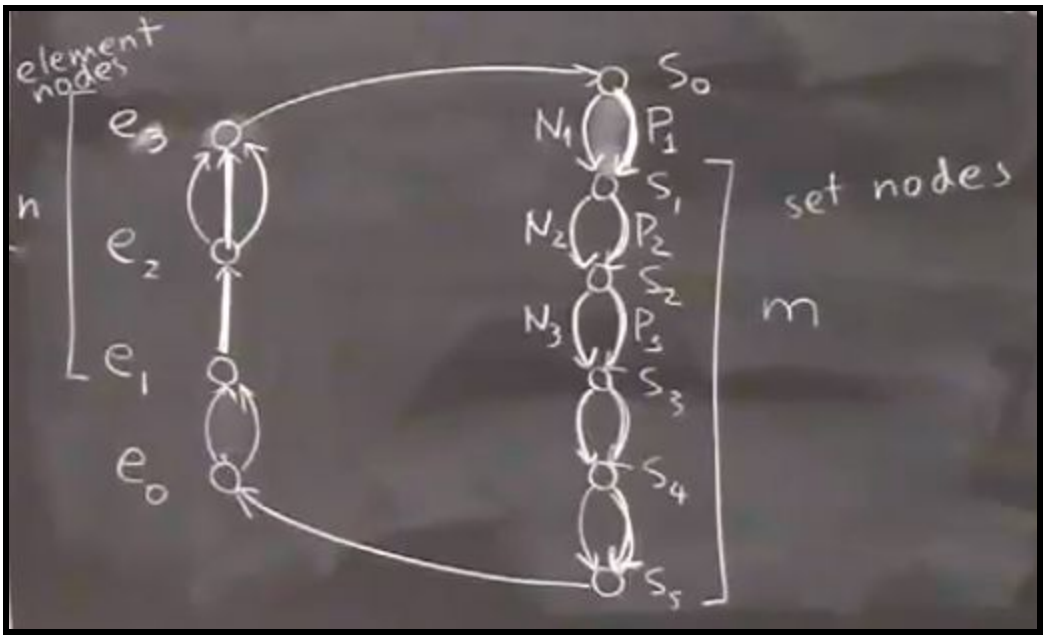


We can also have a gadget connecting more than 2 paths.



Given (U, C), which is an instance of XCOV, construct a directed graph G = (V, E) such that (U, C) has exact cover iff G has HC.
Let U = {u1, u2, ..., un} and C = {A1, ..., Am} such that Aj ⊆ U.
Construct G with n + m + 2 nodes.
n is the number of elements in U.
m is the number of sets in C.

Here is an example/informal construction of how to construct G.



The edge from e0 to e1 corresponds to u1.
The edge from e1 to e2 corresponds to u2.
The edge from e2 to e3 corresponds to u3.

The edge from s0 to s1 corresponds to A1.
The edge from s1 to s2 corresponds to A2.
The edge from s2 to s3 corresponds to A3.
The edge from s3 to s4 corresponds to A4.
The edge from s4 to s5 corresponds to A5.

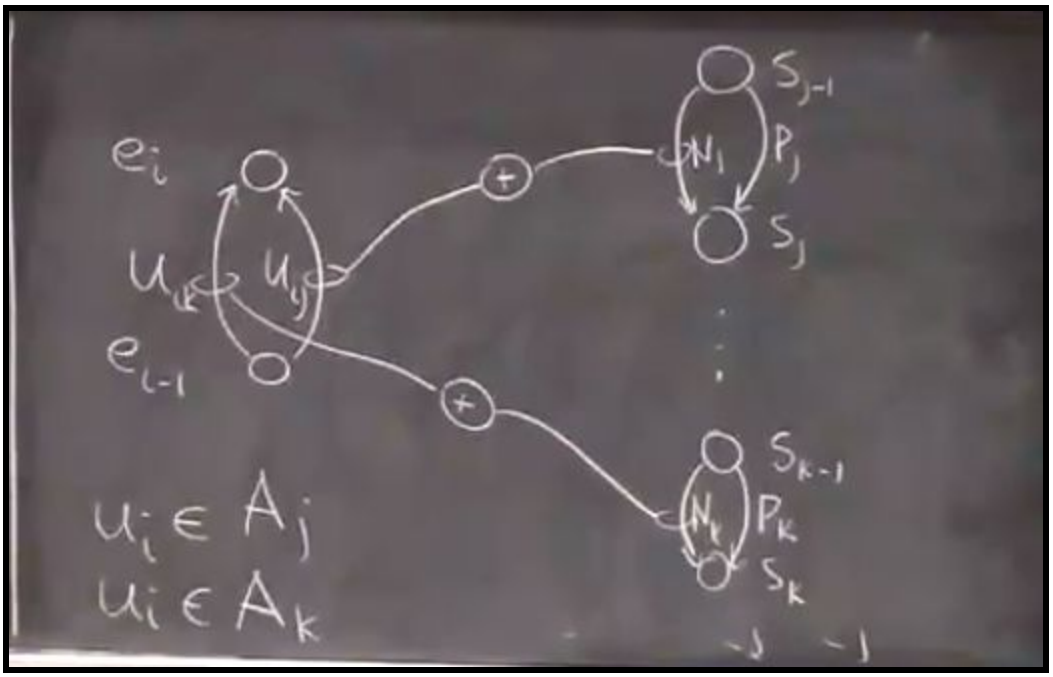
We construct set nodes, si, with an extra s0 such that there are 2 edges from si−1 to si.
We will call one edge Pi and the other edge Ni.
The Ni edges are part of a gadget.
If our HC path goes through a Pi, I will put the corresponding Ai in the exact cover. Otherwise, I won't.

We construct element nodes, ei, with an extra e0 such that there are multiple edges from ei−1 to ei.
Each element has an edge for each set it is in.
E.g. u1 is in 2 sets, so that's why there are 2 edges from e0 to e1.
E.g. u2 is in 1 set, so that's why there is 1 edge from e1 to e2.

E.g. u3 is in 3 sets, so that's why there are 3 edges from e2 to e3.
To find which set(s) each ui is in, we will couple each of the edges in ei with the corresponding Ni of the appropriate set via a gadget.
Suppose element ui ∈ Aj and ui ∈ Ak.

- We call the edges uij and uik and connect uij to Ni with a gadget and uik to Nk with a gadget.
- The number of gadgets for each si is the number of elements in the set.

Here is a picture of the above:



We have an edge from en to S0.
We have an edge from sn to e0.

Here is the formal construction of how to construct G:
G = (V, E)
V = {ei : 0 ≤ i ≤ n} U {sj : 0 ≤ j ≤ m} U [additional gadget nodes]
E = {uij : ui ∈ Aj} U {Nj,Pj : 1 ≤ j ≤ m} U {(em, s0),(sm, e0)} U [edges for the gadgets]

c. Claim: (U, C) has exact cover iff G has a HC.
Proof:
(=>)
Suppose C' is an exact cover of (U, C).
Then, choose edges as follows:

1. For every Aj ∈ C', choose Pj.
   For every Aj ∉ C', choose Nj.
   This connects the set nodes.
2. For each ui ∈ U choose a unique uij such that ui ∈ Aj and Aj ∈ C'.
   This connects the element nodes.
   This must exist because C' is an exact cover.
3. Choose edges (en, s0),(sn, e0).

These form a HC because we've connected the set nodes, the element nodes and the connecting edges.

(<=)
Suppose G has a HC H.
Let C' = {Aj |H uses Pj}
C' is an exact cover.

1. Every ui ∈ U is covered by some Aj ∈ C'.
   H must go from ei−1 to ei using uij s.t. ui ∈ Aj.
   H does not use Nj because of the coupling by the gadget.
   Therefore, H must use Pj.
   Therefore, Aj ∈ C0.
   Therefore, ui is covered by C'.
2. Sets in C' are disjoint.
   Suppose for contradiction, Aj, Ak ∈ C' such that j ≠ k, Aj ∩ Ak ≠ ∅.
   Let ui ∈ Aj ∩ Ak.
   H uses both Pj and Pk, because Aj, Ak ∈ C'.
   H does not use Nj or Nk, because it would visit the same nodes twice.
   H uses uij and uik, because of gadgets.
   H uses two edges from ei−1 to ei.
   Therefore, H is not HC. This is a contradiction.

d. **Argument for polynomial time:**

$|\langle U, C\rangle| = n + \sum_{j=1}^{m} |Aj|$

n is the number of elements in U.

$\sum_{j=1}^{m} |Aj|$ is the sum of the sizes of the various sets in C.

|G| = # of nodes + # of edges
# of nodes = n + m + 2

$\#\text{ of edges} = 2m + 2 + \sum_{j=1}^{m} |Aj|$

2m → Every set has 2 edges.
2 → There's an edge from en to s0 and e0 to sn.

$\sum_{j=1}^{m} |Aj|$ → Recall that each ei-1 to ei has x edges, where x is the number of sets ui is in.
I.e. e0 to e1 has 2 edges if u1 is in 2 sets.

|G| is polynomial w.r.t |⟨U, C⟩|.
Hence, construction of ⟨G⟩ from ⟨U, C⟩ is polytime w.r.t |⟨U, C⟩|.

Undirected Hamiltonian Cycle (UHC)

- Instance: ⟨G⟩, where G = (V, E) is undirected graph.
- Question: Does G have a HC?
- **Theorem 10.3:** UHC ∈ NPC.

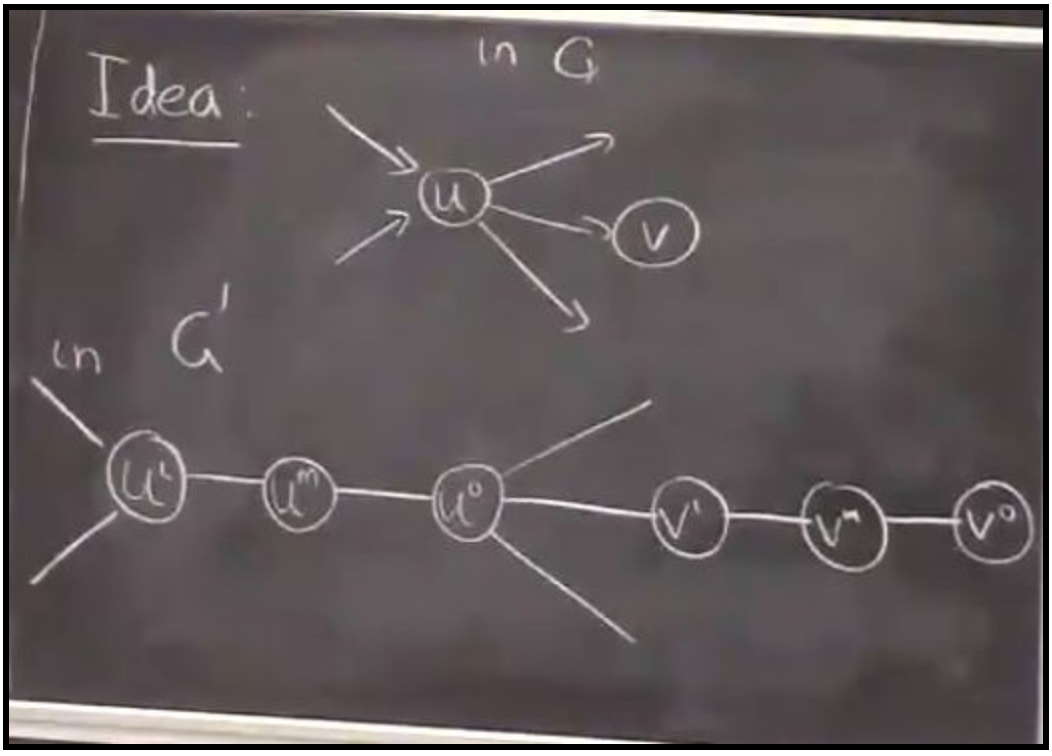
Proof:

a. **UHC ∈ NP**

b. **Show that DHC $\leq_P$ UHC:**
Given a directed graph G = (V, E), construct an undirected graph G' = (V', E') such that G has a HC iff G' has a HC.

Here's the idea:

- If G has a node "u" that's connected to a node "v" we construct 6 extra nodes: ui, um, uo and vi, vm, vo.
  ui stands for "u in".
  um stands for "u middle".
  uo stands for "u out".
  Similar naming patterns for vi, vm and vo.
- ui connects to um which connects to uo.
- vi connects to vm which connects to vo.
- uo connects to vi.
- Here's a picture:



c. **Claim: G has a HC iff G' has a HC.**

   Proof:
   (=>)
   Suppose G has a HC.
   Just use the appropriate triple of nodes that replaces the original.

   (<=)
   Suppose G' has an HC.
   Start it from um for some u.
   Without loss of generality assume the next node is uo. If its ui just reverse the direction.
   After uo the next node is vi for some v.
   The next node must be vm. This is because if vi doesn't go to vm now, in order to visit vm later, it must revisit some node.
   By the same reasoning, the next node must be vo.
   Hence, there must be this pattern of xi -- xm -- xo.
   This easily translates to the HC from G' to G as we just choose the above pattern for the appropriate nodes.

d. **Argument for polynomial time:**
   For each node, x, in G, there are 3 nodes, xi, xm and xo, in G'.
   Hence, $|\langle G' \rangle|$ is polynomial w.r.t to $|\langle G \rangle|$.

TSP:

- Instance: $\langle G, wt, b \rangle$, where G = (V,E), wt is a weight function and b is a budget.
- Question: Is there a tour, HC, of G of wt ≤ b?
- **Theorem 10.4:** TSP ∈ NPC

SubSet Sum (SUBSUM):

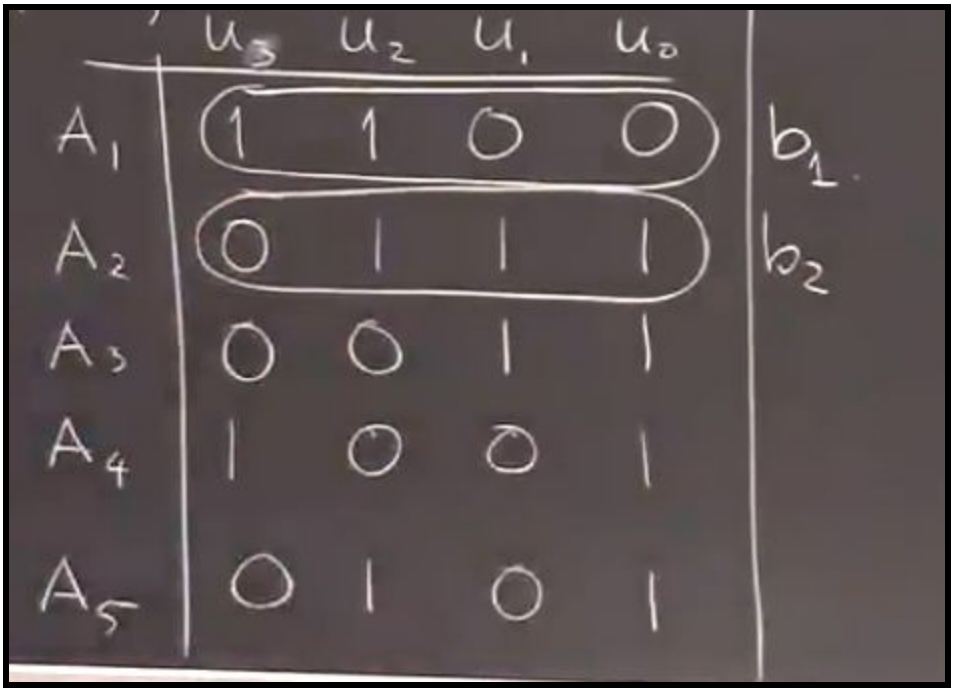
- Instance: $\langle A,k \rangle$, where A is a list of positive integers, I.e. $A = a_1, a_2, \ldots, a_n$ s.t. $a_i \in Z^+$, and k is a positive integer, $k \in Z^+$.
- Question: Is there a subsequence of A that adds up to k?
  I.e. of subsequence of A: $a_{i1}, a_{i2}, \ldots, a_{im}$ s.t. $1 \leq i1 \leq i2 \leq \ldots \leq im \leq m$.
  E.g. If A = 3, 2, 17, 11, 3, 9 and k = 15, then the answer is yes. (3+3+9 = 15) However, if k = 24, then the answer is no.
- **Theorem 11.1:** SUBSUM $\in$ NPC

  Proof:

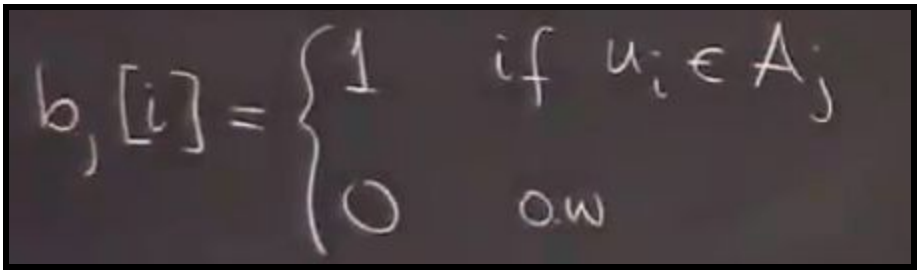
  a. **Proof that SUBSUM $\in$ NP:**
     The certificate here is the subsequence of A. The verifier simply checks that it is a subsequence of A and that the subsequence adds up to k.
  b. **Show that XCOV $\leq_p$ SUBSUM:**
     Given $\langle U,C \rangle$, which is an instance of XCOV, construct in polytime $\langle A,k \rangle$, which is an instance of SUBSUM, s.t. $\langle U,C \rangle \in$ XCOV iff $\langle A,k \rangle \in$ SUBSUM.
     Let $U = \{u_0, u_1, \ldots, u_{n-1}\}$.
     Let $C = \{A_1, A_2, \ldots, A_m\}$ s.t. $A_j \subseteq U$.

     Intuition: We will represent every element in set A_j as a binary number by viewing it in binary.
     For example:
     $U = \{u_0, u_1, u_2, u_3\}$
     $A_1 = \{u_2, u_3\}$
     $A_2 = \{u_0, u_1, u_2\}$
     $A_3 = \{u_0, u_1\}$
     $A_4 = \{u_0, u_3\}$
     $A_5 = \{u_0, u_2\}$
     We will create a matrix where every row corresponds to a set and every column corresponds to an element.
     If an element, y, is in a set, x, we put a 1 in (x,y).
     If an element, y, is not in a set, x, we put a 0 in (x,y).

This is how our matrix will look like:



Notice how A_1 has elements u_2 and u_3, but not u_0 or u_1.
Hence, the cell where u_3 and A_1 intersect has a 1.
Similarly, the cell where u_2 and A_1 intersect has a 1.
And the cell where u_1 and A_1 intersect and where u_0 and A_1 has a 0.
Each row of binary bits will be labelled b_i. For example, the first row is labelled b_1.
The below picture just references the fact that if an element is in a set, we put a 1 in the cell where they intersect, and if an element is not in a set, we put a 0 in the cell where they intersect.



Since we are looking for an exact cover, we are looking for sets s.t. when we add up their binary representations, we get a binary number s.t. all bits are 1'sr.
For example, take A_1 and A_3 from our example.
A_1 = 1100
A_3 = 0011
  1100
+0011
=1111 ← All the bits are 1

However, there's an issue with this, and the issue is carrying over 1's.
For example, take A_2, A_3, and A_5 from our example
A_2 = 0111
A_3 = 0011
A_5 = 0101

```
 0111
 0011
+0101
=1111
```

A_2, A_3, and A_5 aren't an exact cover because none of them contains U_4 and there are duplicates. However, when you add up their binary representations, you get 1111.
We can solve this by viewing b_i's in base (m + 1) because the only way to get a carry over is to sum m + 1 sets, but we only have m sets, so it's impossible to have a carry over.

c. **Claim: (U, C) has an exact cover iff there is a subsequence of b_1, ..., b_m, viewed as base (m + 1) numbers, that add up to 111...1 in base (m + 1).**

Note: 111...1 in base (m + 1) means that $k = \sum_{i=0}^{n-1} (m+1)^i = \frac{(m+1)^n - 1}{m}$.

Note: A subsequence of b_1, ..., b_m, viewed as base (m + 1) numbers means that $b_i = \sum_{j=0}^{n-1} (b_i[j])*(m+1)^j$.

Proof:
Suppose $\{A_{j1}, ..., A_{jk}\}$ is an exact cover of C.
$\leftrightarrow \forall i, 0 \le i \le n-1, \exists$ unique t, $1 \le t \le k$, s.t. $u_i \in A_{jt}$.
$\leftrightarrow \forall i, 0 \le i \le n-1, \exists$ unique t, $1 \le t \le k$, s.t. $b_{jt}[i] = 1$.
$\leftrightarrow$ the sum of b_{j1}, ..., b_{jk} = 111…1 (as base (m + 1) numbers).

This is polynomial as we just need to construct a (m × n) matrix.

Partition (PART):

- Instance: ⟨A⟩, where A is a list of positive integers, I.e. A = $a_1, a_2, \ldots, a_n$ s.t. $a_i \in Z^+$.
- Question: Is there a subsequence of A that adds up to half of the total sum of A?

  I.e. Is there a subsequence of A s.t. their sum = $\frac{\sum_{i=1}^{n} ai}{2}$?
- **Theorem 11.2:** PART ∈ NPC

  Proof:

  a. **Proof that PART ∈ NP:**
  The certificate is a sequence of numbers and the verifier verifies that this sequence is a subsequence of A and that their sum is equal to half the sum of the elements in A.

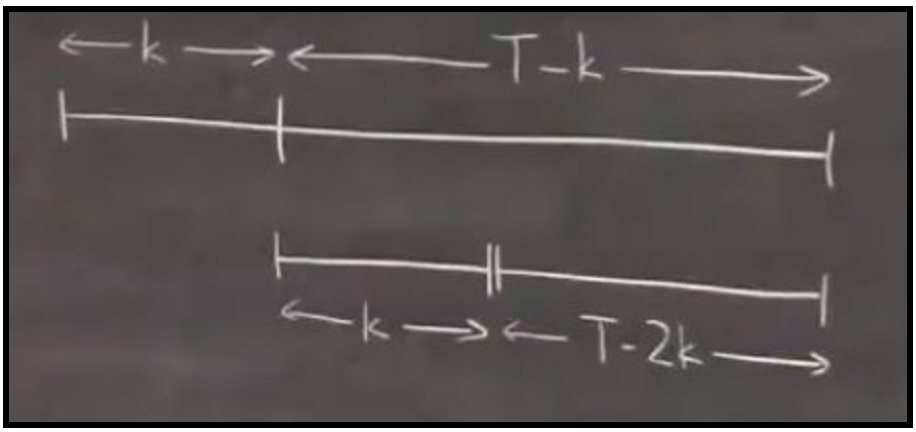
  b. **Show that SUBSUM ≤$_p$ PART:**
  Given ⟨A, k⟩, which is an instance of SUBSUM, construct in polynomial time ⟨A'⟩ s.t. ⟨A, k⟩ ∈ SUBSUM iff ⟨A'⟩ ∈ PART.

  Intuition:

  Let T = $\sum_{i=1}^{n} a_i$.

  We can add some of the elements in A and get k.
  If we take out k from our original sequence we're left with T−k. We can use that as our first half.
  If we take k from (T-k), we're left with T-2k.
  (T-k) and k add up to A.
  Now, if we add (T-2k), we can partition A into 2 halves:
  1. T-k
  2. k and (T-2k)

  Here's a picture.



  However, this picture is misleading, as it's possible that T − 2k is negative but our sequence must use a positive number. In that case, use 2k − T.
  Case 1. T − 2k < 0 → T − k ≤ k
  Here we use 2k − T.

Case 2. T − 2k = 0 → k = T/2
Then we are done, this is exactly what we are looking for.

c. Claim: ⟨A, k⟩ ∈ SUBSUM iff ⟨A'⟩ ∈ PART.

$$\text{From } A = a_1, ..., a_n \text{ and } k$$

$$A' = \begin{cases} A & \text{if } T = \sum_{i=1}^{n} a_i = 2k \\ a_1, ..., a_n, T - 2k & \text{if } T - 2k > 0 \\ a_1, ..., a_n, 2k - T & \text{if } T - k < 0 \end{cases}$$

The construction of A' from A, k is polynomial time.

Linear Programming (LP):

- An optimization Problem.
- Max/min a linear function subject to linear inequality constraints.
- For example to minimize the cost of a diet, we could just buy nothing but then we would starve. Here, the constraint is that we have sufficient nutrients in our diet.
- For example, we want to maximize profit from productive activities such that resource constraints are not exceeded.
- Example: Minimize $c_1x_1 + \ldots + c_nx_n$ where x_i are variables and c_i are constants in Z s.t.
  $a_{11}x_1 + \ldots + a_{1n}x_1 \geq b_1$
  .
  .
  .
  $a_{m1}x_1 + \ldots + a_{mn}x_m \geq b_m$
  This can be written simply as $A\bar{x} \leq \bar{b}$.
- There is a polynomial time algorithm for LP if $\bar{x} \in Q^n$.
  However, if we insist that the solution, $\bar{x}$, be integers, then it is NPC.
- The decision version of the linear programming questions works as follows:
  "Given A and $\bar{b}$, is there a vector $\bar{x}$ s.t. $A\bar{x} \leq \bar{b}$?"

Integer Linear Programming (ILP):

- Instance: ⟨A, B⟩ where A = m × n matrix of Z and b = m-vector of Z.
- Question: Is there an $\bar{x}$ s.t. $A\bar{x} \leq \bar{b}$ [$\bar{x}$ is a n-vector]

Zero or One Equations (ZOE):

- Instance: ⟨A⟩ where A = 0/1 m × n matrix.
- Question: Is there an $\vec{x}$ s.t. $A\vec{x} \leq \begin{pmatrix} 1 \\ . \\ . \\ 1 \end{pmatrix}$? [$\vec{x}$ is a n x 1 vector]
- **Theorem 11.3:** ZOE ∈ NPC

Proof:

a. **Proof that ZOE ∈ NP:**
The certificate is a 0/1 vector of length n and the verifier verifies that the dot product of any row in matrix A with $\vec{x}$ equals to 1.

b. **Show that XCOV ≤$_p$ ZOE:**
Given ⟨U, C⟩, which is an instance of XCOV, construct in polynomial time ⟨A⟩, which is an instance of ZOE, s.t. ⟨U, C⟩ ∈ XCOV iff ⟨A⟩ ∈ ZOE.
Let U = {u_1, … ,u_n}.
Let C = {A_1, … ,A_m} $A_j \subseteq$ U.

$$A\vec{x} = \begin{pmatrix} a_{11} & \dots & \dots & a_{1m} \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_m \end{pmatrix}$$

$$a_{ij} = \begin{cases} 1 & \text{if } u_i \in A_j \\ 0 & \text{otherwise} \end{cases}$$

Columns represent sets.
Rows represent elements.
1 means that the set has the element.
Each x_i in $\vec{x}$ is either 0 or 1.
x_i = 1 means that we are selecting that set for our exact cover.
x_i = 0 means that we are not selecting that set for our exact cover.
Each $\vec{x} \in \{0, 1\}^m$ defines a subset of C.
$C_{\vec{x}} = \{A_{ji} \mid x_j = 1\}$.
Consider the dot product of (The i-th row of A) and $\vec{x}$, denoted as (The i-th row of A)*$\vec{x}$.
(The i-th row of A)*$\vec{x}$ = $a_{i1}x_1 + a_{i2}x_2 + \dots + a_{im}x_m$

$$a_{ij}x_j = \begin{cases} 0 & \text{if we chose } a_{ij} \text{ but its not in } A_j \text{ or if we don't choose } a_{ij} \\ 1 & \text{if we choose } a_{ij} \text{ and its in } A_j \end{cases}$$

(The i-th row of A)* $\vec{x}$ is the number of times u, appears in sets in $C_{\vec{x}}$.

We want A$\vec{x}$ to be $\begin{pmatrix} 1 \\ . \\ . \\ 1 \end{pmatrix}$ of size n × 1.

c. Claim: ⟨U, C⟩ ∈ XCOV iff ⟨A⟩ ∈ ZOE

If ⟨U, C⟩ has an exact cover C', then $\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_m \end{pmatrix}$ where

$$x_j = \begin{cases} 1 & \text{if } A_j \in C' \\ 0 & otherwise \end{cases}$$ satisfies $$A\vec{x} = \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix}$$.

Conversely, if $\vec{x}$ satisfies $$A\vec{x} = \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix}$$ then $C_{\vec{x}} = \{A_j \mid x_j = 1\}$ is an exact cover.

Therefore, ZOE ∈ NPC.

Zero One Linear Programming (ZOLP):

- Instance: ⟨A, b⟩ where A is a m × n matrix and b is a m-vector of Z.
- Question: Is there an $\bar{x}$ s.t. $A\bar{x} \leq \bar{b}$? [$\bar{x}$ is a n-vector ∈ $\{0, 1\}^n$]
- **Note:** To turn the equality $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = 1$, we simply do $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq 1$ and $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq 1$. However, since we want everything to be in the form of $A\bar{x} \leq \bar{b}$, we multiple the second inequality by (-1). Hence, we get:
  1. $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq 1$
  2. $-a_{11}x_1 - a_{12}x_2 - \ldots - a_{1n}x_n \leq -1$

  These 2 inequalities, together, are equivalent to $a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = 1$.
- **Theorem 11.4:** ZOLP ∈ NPC

Proof:

a. **Proof that ZOLP ∈ NP**
The certificate is the vector. The verifier just multiplies the matrix with the vector and makes sure that the dot product of the i^{th} row of the matrix with the i^{th} element of the vector is less than or equal to b_i.

b. **Show that ZOE ≤$_P$ ZOLP**
Given ⟨A⟩, which is an instance of ZOE, construct in polynomial time ⟨A'⟩, which is an instance of ZOLP s.t. there exists an $\bar{x}$ where $A\bar{x} = \begin{pmatrix} 1 \\ \vdots \\ \vdots \\ 1 \end{pmatrix}$ iff there exists an $\bar{y}$ where $A'\bar{y} \leq \begin{pmatrix} 1 \\ \vdots \\ 1 \\ -1 \\ \vdots \\ -1 \end{pmatrix}$.

As shown above, $A' = \begin{pmatrix} A \\ -A \end{pmatrix}$.

Note: Since A is a m x n matrix and (-A) is also a m x n matrix, A' is a (2m) x n matrix and b is a (2m) vector where the first half of it is 1's and the second half of it is (-1)'s.

- **Corollary 11.5:** ILP ∈ NPC
  Note: ZOLP $\leq_P$ ILP
  Proof:
  a. **Show that VC $\leq_P$ ZOLP:**
     Given ⟨G, k⟩, which is an instance of VC, construct in polynomial time, ⟨A, $\bar{b}$⟩, which is an instance of ZOLP s.t. G has VC of size k iff there exists a $\bar{x}$ s.t. $A\bar{x} \leq \bar{b}$.
     Note: G = (V,E) is an undirected graph.
     Let V = {u_1, u_2, …, u_n}.
     We will introduce variables x_i, ... , x_n ∈ {0, 1}.
     Each $\bar{x}$ = (x_i, …, x_n) corresponds to a V' ⊆ V.
     We do this by having u_i ∈ V' iff x_i = 1.
     $x_1 + x_2 + \ldots + x_n \leq k$ (|V'| ≤ k)
     To show that V' covers all of the edges, we do $x_i + x_j \geq 1$ ∀{u_i, u_j} ∈ E.
     However, since we want ≤, we will turn $x_i + x_j \geq 1$ into $-x_i - x_j \leq -1$.

$$A = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 0 & -1 & -1 & 0 \end{pmatrix}$$

     A has all 1's on the first row.
     Every row afterwards has all 0's except for -1's in the i-th and j-th column.

$$A = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 0 & -1 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} k \\ -1 \\ \vdots \\ -1 \end{pmatrix}$$

3DM:

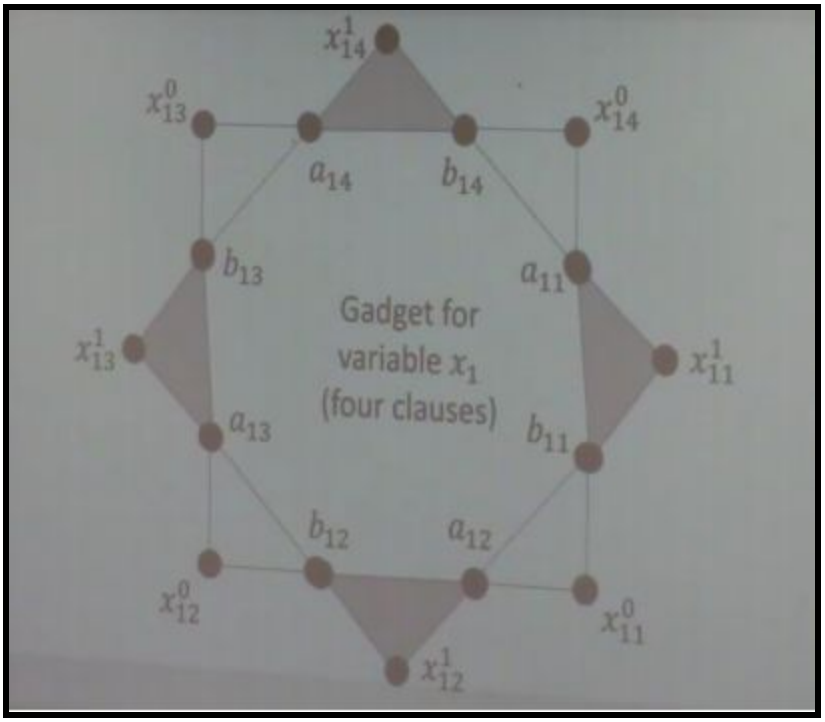
- Suppose we have a coding competition with the following rule:
  - Each team must have exactly 3 people which consists of one 2nd year student, one 3rd year student and one 4th year student.
- Suppose there are n 2nd year students, n 3rd year students and n 4th year students.
- We know that certain triples are compatible, meaning that the people in that triple get along, and certain triples are not compatible, meaning that the people in that triple don't get along.
- Can we make n teams where every team is disjoint and every team consists of compatible teams from one person in each year?
- This is an example of a problem known as **3 dimensional matching**.

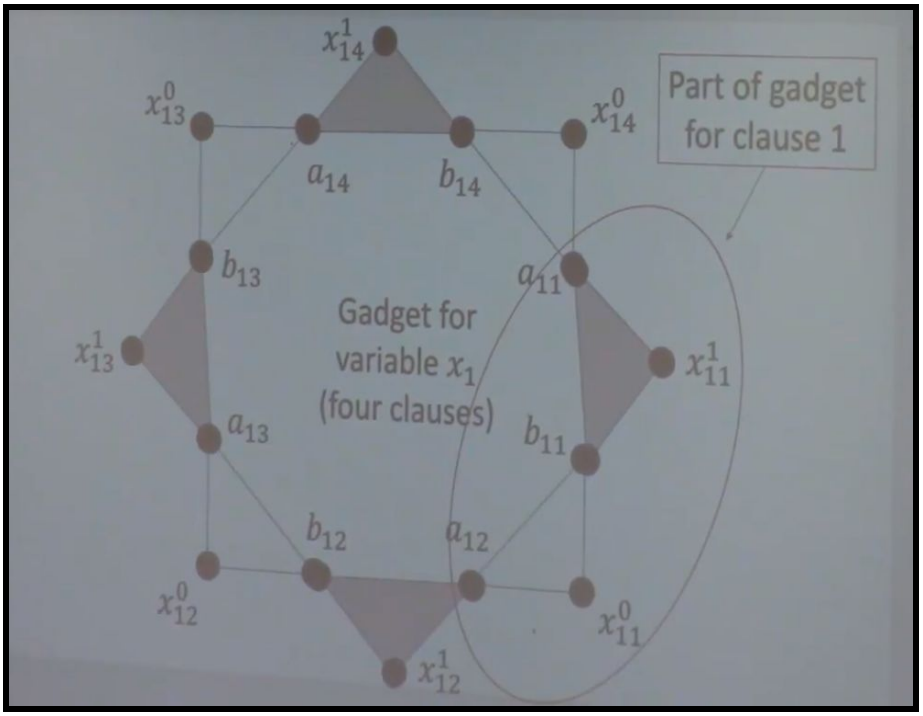
- Instance: ⟨A, B, C, M⟩ where A, b, C are disjoint sets of n elements each and $M \subseteq A \times B \times C$ (M is the compatible triples.)
- Question: Is there $M' \subseteq M$ s.t. $|M'| = n$ and $\forall$ disjoint sets (a, b, c), (a', b', c') $\in M'$, $a \neq a'$,
  $b \neq b', c \neq c'$?
  I.e. Is there a subset of M, M', s.t. there n triples/teams and the teams must consist of distinct people. **Note:** M' is called a matching.
- **Theorem 11.6:** 3DM $\in$ NPC

Proof:

a. **3DM $\in$ NP**
b. **Show that 3SAT $\leq_p$ 3DM**
   Given a 3-CNF formula F, construct in polynomial time (A, B, C, M) s.t.
   F is satisfiable iff $M' \subseteq M$ is a matching.
   F's variables are $x_1, \ldots, x_n$.
   $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$
   $Cj = l_j^1 \vee l_j^2 \vee l_j^3$

   Group I triples:
   - For every variable xi we will construct a gadget like shown below.





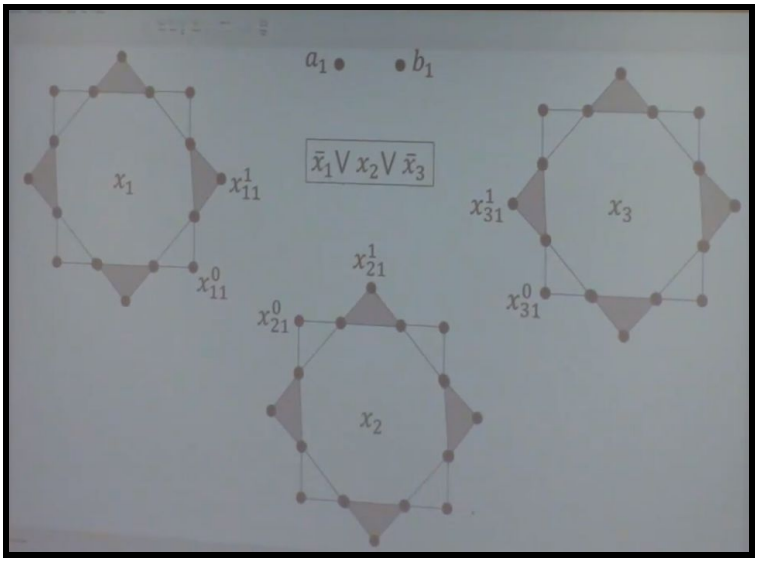
This is an example of a gadget with 4 clauses.

- For every variable and every clause, we will have 2 triples, which will be interconnected in the above pattern.
- We label x_{ij}^a for each item where a means positive or negative. If a = 0, then x_{ij}^0 is negative and if a = 1, then x_{ij}^1 is positive. Furthermore, i means that this is the i^{th} variable and j means that this is the j^{th} clause.
- The elements aij and bij they are constructed as follows:
  - a_{ij} is connected to x_{ij}^1 and b_{ij}.
  - b_{ij} is connected to x_{ij}^0 and a_{ij+1}.
- These interconnections are done in such a way that if we select the shaded triple we can not select the unshaded triple that follows.
  - If we select the triple containing x_{11}^1 then we cannot also select the triple containing x_{11}^0 because they share b_{11}.
  - If we choose one shaded triple, they need to be all shaded.
  - If we choose one clear triple, they need to be all clear.
  - This is what we want, since x_{11} can only be one truth value.
- Here's how we create the triples more precisely:
  ∀i, j where $1 \le i \le n$ and $1 \le j \le m$, we have $(a_{ij}, b_{ij}, x_{ij}^1)$ and $(a_{ij\oplus 1}, b_{ij}, x_{ij}^0)$, where $j \oplus 1 = \begin{cases} j+1 & \text{if } j < m \\ 1 & \text{if } j = m \end{cases}$.
- We have 2mn of these triples.

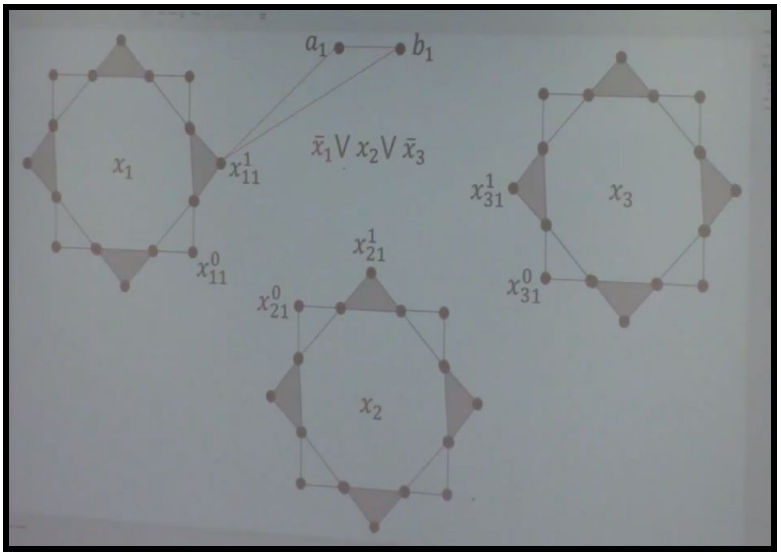
Group 2 triples:

- Let $C_j = l_j^1 \vee l_j^2 \vee l_j^3$.
- We will make one triple per literal, 3 for the entire clause.
- Take l_j^t, where t is 1, 2, or 3. There are 2 possibilities for it.
    1. l_j^t is a positive literal.
       I.e. $l_j^t = x_i$
       Hence, we create a triple (a_j, b_j, x_{ij}^0) where $a_j \in A$ and $b_j \in B$.
    2. l_j^t is a negative literal.
    3. I.e. $l_j^t = \neg x_i$
       Hence, we get (a_j, b_j, x_{ij}^1) where $a_j \in A$ and $b_j \in B$.

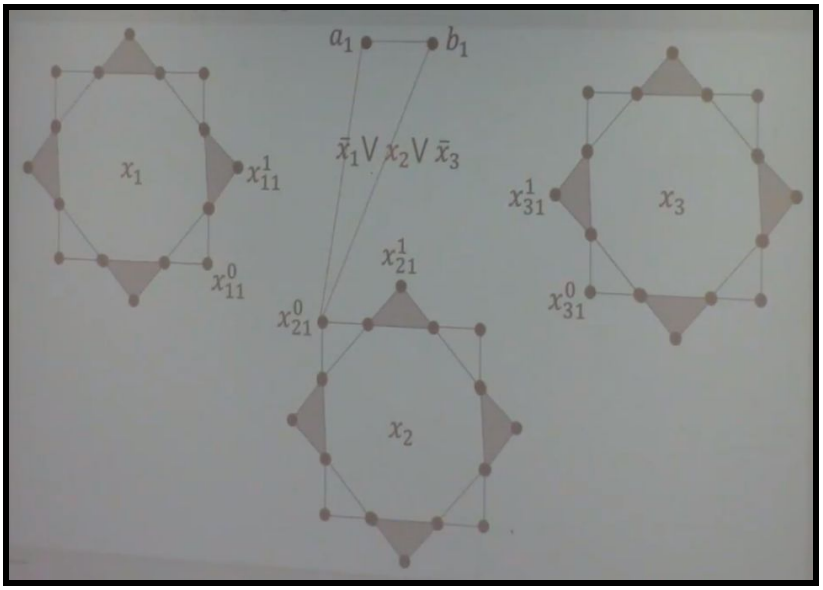
    We take the x_{ij} that's opposite to our x_i so that we take the variable that was not selected in our previous gadget.
    Here's a picture.



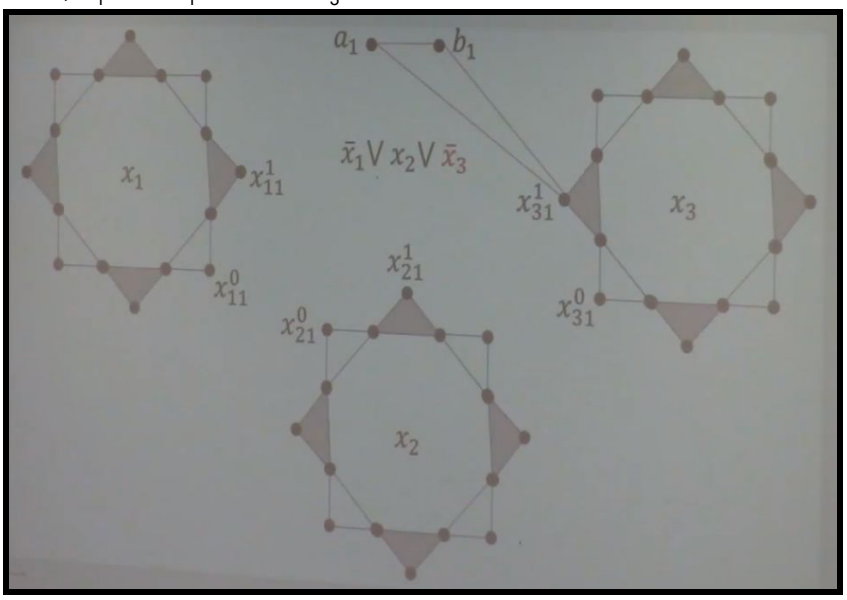
Here, a_1 and b_1 are in $\neg x_1$.



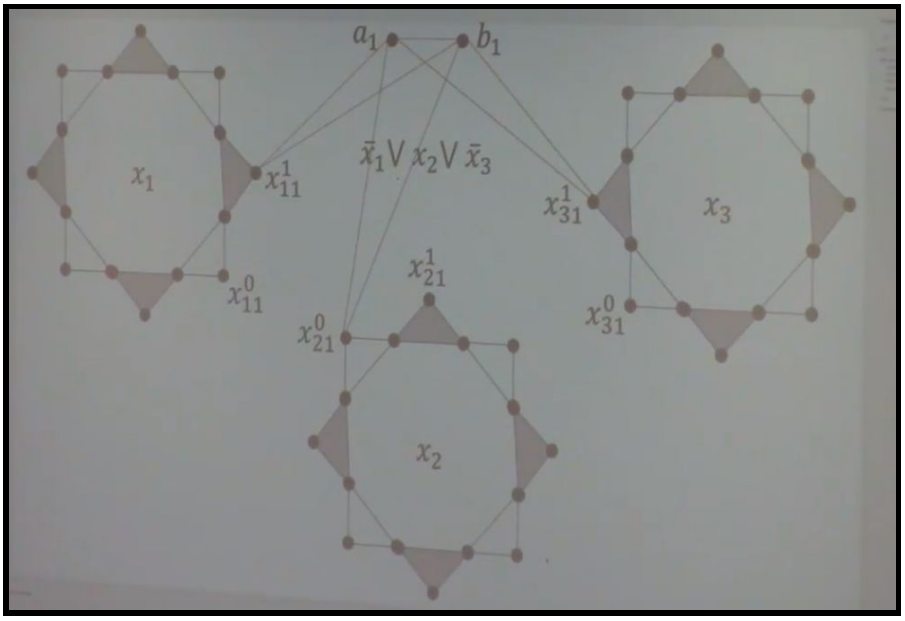
Here, a_1 and b_1 are in x_2.



Here, a_1 and b_1 are in $\neg x_3$.



The whole thing looks like this



- There are 3m such triples here. There are m clauses and each clause has 3 triples.

Group 3 Triples:

- Recall the 2mn triples from group 1.
  x_{ij}^0 and x_{ij}^1 are the tips of the triangles in group 1.
  Half of the 2mn triples (mn triples) will be covered by choosing whether or not to use x_{ij}^0 or x_{ij}^1.
  Another m triples will be covered because we have chosen certain triples, one per clause, to indicate which literal of that clause will make that whole clause true.
  2mn - mn - m = m(n-1). This means that there are m(n-1) triples that have not been covered yet. We cover them here.
- We will introduce new variables ($\sim a_k$, $\sim b_k$, x_{ij}^0) and ($\sim a_k$, $\sim b_k$, x_{ij}^1), where $1 \leq k \leq m(n-1)$. In total, we have $2m^2n(n-1)$ triples.

In total, we define:

- $A = \{a_{ij} \mid 1 \leq i \leq n\} \cup \{a_j \mid 1 \leq j \leq m\} \cup \{\sim a_k \mid 1 \leq k \leq m(n-1)\}$
- $B = \{b_{ij} \mid 1 \leq i \leq n\} \cup \{b_j \mid 1 \leq j \leq m\} \cup \{\sim b_k \mid 1 \leq k \leq m(n-1)\}$
- $C = \{c_{ij} \mid 1 \leq i \leq n\} \cup \{c_j \mid 1 \leq j \leq m\} \cup \{\sim c_k \mid 1 \leq k \leq m(n-1)\}$
- Now we need to confirm that A, B, C are all of the same size.
  $mn + m + m(n-1) = 2mn = |A| = |B| = |C|$
- # of triples:
  Group I = 2mn
  Group II = 3m
  Group III = $2m^2n(n-1) = O(m^2n^2)$.
  In total, we have $O(m^2n^2)$.

d. F is satisfiable iff (A, B, C, M) has a matching.

Sketch of Proof:
(=>)
Let τ satisfy F.
Pick triples in M' as follows:
If τ(xi) = 1, then pick the triple $(-, -, x_{ij}^{1})$ from Group 1.
If τ(xi) = 0 then pick the triple $(-, -, x_{ij}^{0})$ from Group 1.

For each C_j pick l_j^{tj} s.t. $\tau(l_j^{tj}) = 1$.
If $\tau(l_j^{tj}) = x_i$ then pick the triple $(-, -, x_{ij}^{0})$ from Group 2.
If $\tau(l_j^{tj}) = \neg x_i$ then pick the triple $(-, -, x_{ij}^{1})$ from Group 2.

For each x_{ij}^{0} or x_{ij}^{1}, not yet covered, choose the triple $(\sim a_k, \sim b_k, x_{ij}^{0/1})$ from Group 3 to cover it.

(<=)
Left it as exercise and gave the following hints:

1. Let M' be a matching.
2. $$\text{Define } \tau(x_i) = \begin{cases} 1 & \text{if } M' \text{ includes } (-, -, x_{ij}^{1}) \\ 0 & \text{if } M' \text{ includes } (-, -, x_{ij}^{0}) \end{cases}$$

<u>3XCOV Exact cover by 3-sets [sets of size 3]:</u>

- **Theorem 11.7:** 3XCOV ∈ NPC

Proof:
Left as exercise with the hint: show that $3DM \leq_p 3XCOV$.

P:

- The complexity class **P** contains all problems that can be solved in polynomial time. Polynomial time means $O(n^k)$, where k is a constant.
- Formally: P = {L | There is a TM that decides L in polynomial time.}
- **Theorem 1:** L ∈ P iff there is a polynomial-time TM for it.
  Essentially, a problem is in P iff you could solve it using a TM in polynomial time.
- **Note:** If a language L is in P, then $\overline{L}$ is also in P.
  I.e. P is closed under complementation.
- Here's how you can prove that a language, L, is in P:
  1. Construct a TM that decides L in polynomial time.
  2. Use P's closure properties.
  3. Reduce the language to a language in P.
     If $A \leq_P B$ and B ∈ P, then A ∈ P.

NP:

- The complexity class **NP** contains all problems that can be solved in polynomial time by an NTM.
- Formally: NP = {L | There is a NTM that decides L in polynomial time.}
- The NTMs we have seen so far always follow this pattern:
  1. M = On input w:
     a. Nondeterministically guess some object.
     b. Deterministically check whether this was the right guess.
     c. If so, accept. Otherwise, reject.
- **Theorem 2:** L ∈ NP iff there is a deterministic TM V with the following properties:
  1. w ∈ L iff there is some c ∈ Σ* such that V accepts ⟨w, c⟩.
  2. V runs in time polynomial in |w|.
- A TM V with the above property is called a **polytime verifie**r for L.
- The string c is called a **certificate** for w.
- You can think of V as checking the certificate that proves w ∈ L.
- Important properties of V:
  1. If V accepts ⟨w, c⟩, then we're guaranteed w ∈ L.
  2. If V does not accept ⟨w, c⟩, then either
     a. w ∈ L, but you gave the wrong c, or
     b. w ∉ L, so no possible c will work.
- Important properties of the certificate, c:
  1. c must be comprehensive, meaning that all yes-instances have one.
  2. c must be sound, meaning that all no-instances do not have one.
  3. c must be short.
  4. c must be efficiently checkable.
- Here's how you can prove that a language, L, is in NP:
  1. If there is a verifier V for L, we can build a poly-time NTM for L by nondeterministically guessing a certificate c, then running V on w.
  2. If there is a poly-time NTM for L, we can build a verifier for it. The certificate is the sequence of choices the NTM should make, and V checks that this sequence accepts.

  Note that the above 2 ways are the same.
  3. If $L1 \leq_P L2$ and L2 ∈ NP, then L1 ∈ NP.

Here are 4 steps we can do to prove that a language, L, is in NP:

1. Show how to generate certificates for L.
   We can use a NTM to generate all of the strings and say that each one is a certificate.
2. Argue that each certificate is short in size.
3. Explain how the verifier works to validate input.
4. Argue that the verifier works in polytime.

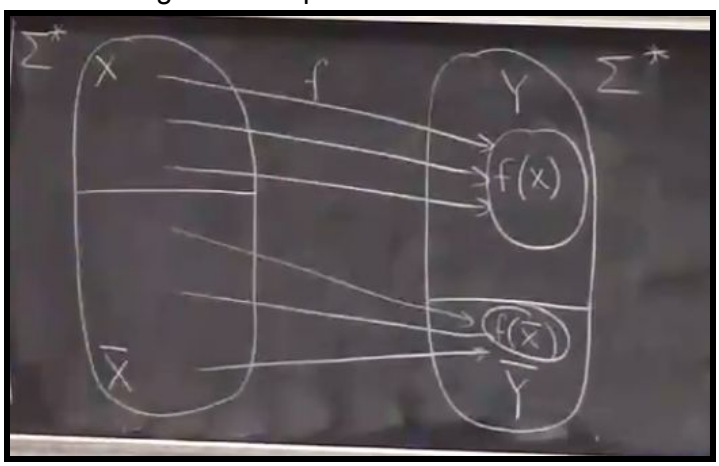
- **Theorem 3:** $P \subseteq NP$
  Conjecture 3.1: $P \neq NP$ and hence, $P \subset NP$.

Cook Reduction:

- Let X, Y be problems (not necessarily decision problems). X **cook reduces** to Y , denoted as $X \rightarrow_P Y$ if there exists a polynomial time algorithm A that solves X given an oracle (blackbox subroutine) for Y where each use of the Y-oracle counts as 1 step.
  Note: While using the Y-oracle counts as 1 step per use, A has to prepare input(s) to the Y-oracle. This preparation is not part of the 1 step and is counted separately.
- **Theorem 4:** If $X \rightarrow_P Y$ and there exists a polynomial time algorithm for Y, then there exists a polynomial time algorithm for X.

Karp Reduction/Polytime Reduction:

- Let $X, Y \subseteq \sum^*$ be languages. X **karp-reduces/polytime reduces** to Y, denoted as $X \leq_P Y$, iff there exists a function $f: \Sigma^* \rightarrow \Sigma^*$ that can be completed in polynomial time s.t. $x \in X$ iff $f(x) \in Y$.
- Here's a diagram to help with the definition.



f maps all the Yes-instances of X to a subset of the Yes-instances of Y.
Similarly, f maps all the No-instances of X to a subset of the No-instances of Y.

- **Theorem 5:** $\leq_P$ is transitive.
  I.e. If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

NPC:

- Y is a NP-Complete (NPC) language iff
    a. $Y \in NP$
       I.e. Y is in NP.
    b. $\forall X \in NP, X \leq_P Y$
       I.e. Every problem, X, in NP must be polytime reducible to Y.
       This means that Y is the “hardest” in NP.
- **Theorem 6:** If Y is NPC and $Y \in P$ then P=NP.
- **Theorem 7:** If Y is NPC, $Z \in NP$ and $Y \leq_P Z$ then Z is NPC.
- How to Prove $B \in NPC$:
    1. Prove that $B \in NP$.
       You can do this with an NTM or certificate.
    2. Choose problem A, which is known to be NPC.
    3. Describe a polytime reduction of A to B, $A \leq_p B$.
       - Show how to transform any instance of A into an equivalent instance of B.
       - Argue that the transformation is polytime.