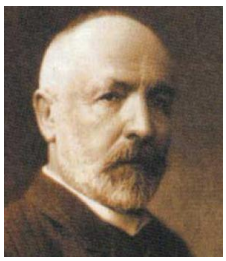
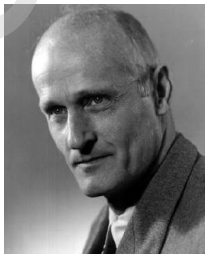
Theory of Computation





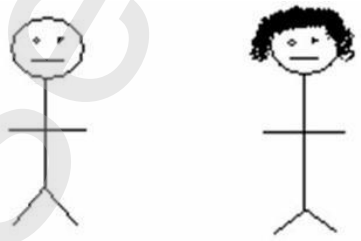

The Church Turing Thesis

This is the first "pure" course in theoretical computer science. Be concerned with the following basic questions in computer

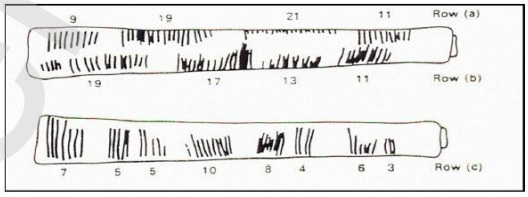
1. What is a computation?
2. Is there a universal model of computation?
3. Can everything be computed?
4. Is there a relation between computations and mathematics?
5. Can we identify problems that are not computable?
6. What resources are needed to perform a certain computation?
7. Can we identify computationally hard problems?

Earliest Models of Computation

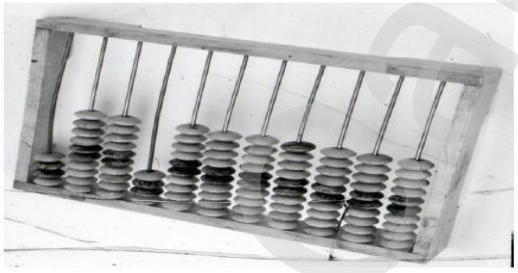
In 30 seconds draw the earliest model of computation that comes to you mind. I think there were two earliest computers. Let me show you the pictures of these models that I have made.



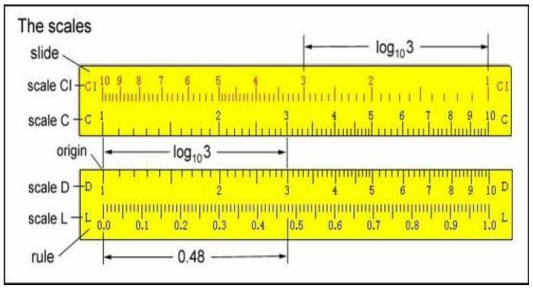
Yes, I think computing as old as human beings. In fact, the earliest known mathematical artifact is the Lebombo bone which dates back to 35,000 BC! After that humans also used clay. Tablets which are found in many civilizations.



Abacus was the next device. Here is a picture of a typical abacus.



Slide rule was another device that was invented in 1600"s.



1. Tally sticks like the Lebombo bone are the earliest computing devices. Earliest known is 35,000 BC.
2. Clay tablets were then used to store the count of livestock etc.
3. Abacus is known to exist 1000 BC and 500 BC in Mesopotamia and China.
4. Around 1620 John Napier introduced the concept of logarithms. The slide rule was invented shortly afterwards.

More than physical devices we are interested in the history of concepts in computing. We are interested in very basic questions. Let us ask the following more important questions.

When was the first algorithm devised? This is a very difficult question. Since, algorithms were not studied the way we do them today.

Earliest Algorithms
Perhaps, the first algorithm devised was unary addition thousands of years ago in some cave. The limbobo bone type devices were used to execute that algorithm. The concept of an algorithm is very well developed in Euclid‟s school. Two striking examples are:

The Euclidean Algorithm to compute the GCD. It is a non-trivial algorithm, in the sense that it is not clear at all why it computes the GCD. The constructions by ruler can compass are examples of precise algorithms. The elementary steps are well defined and the tasks are well defined.

Algorithms and Computation
The most important question in theory of computation is: What is a computation? What is an algorithm? When was this question asked?

1900 International Congress of Mathematicians

In 1900 David Hilbert and eminent mathematician was invited to give a lecture at the international congress of mathematicians. In his talk he posed 21 questions as a challenge to 20th century mathematicians. His problems spanned several areas from mathematical analysis, logic, foundation of set theory and geometry. However, one of the question the 10th problem was very interesting

Hilbert's Tenth Problem
Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise **a process** according to which it can be determined **in a finite number of operations** whether the equation is solvable in rational integers.

Note two important words:
1. Process
2. in a finite number of operations

Hilbert was asking for an algorithm!!

Let us look examine Hilbert‟s 10th question in modern language.
Consider a polynomial in many variables for example:

- $X^2 + y^2 = 0$
- $X^2 + y^2 = z^2$
- $ax + by + z^3 + 17d^3 = 14567$

These are called diophantine equations. What we want to do is to ask if there is a solution in positive integers. For example $X^2 + y^2 = z^2$ has a solution x = 3, y = 4 and z = 5. So this equation has a solution. On the other hand $x^2 + x + 2y = 39$ does not have a solution. So the question is: Given a diophantine equation does it have a solution?

Hilbert‟s Tenth Problem was the following problem in computer science: Can we write a program that will take a diophantine equation as an input and tell us if the equation has solution in positive integers? He was asking for an algorithm.

A solution to Hilbert‟s tenth problem would mean that mathematicians could solve diophantine equations mechanically. It took almost 70 years to solve this question. Eventually **Yuri Matiyasevich** was able to solve Hilbert‟s Tenth Problem. What was Matiyasevich‟s solution? Did he invent an algorithm? No.

Matiyasevich‟s proved that Hilbert‟s 10th problem was unsolvable. There is no algorithm that can tell if a diophantine equation has solutions in positive integers. Note that to understand what he did is not easy. Matiyasevich did not say he is not smart enough to find a solution. He **proved** that no such algorithm existed. He was showing impossibility of the existence of an algorithm.

Impossibility Proofs

Impossibility proofs are not very uncommon in mathematics. Let us look at some of them:

- It is impossible to write $\sqrt{2}$ as a ratio of two integers.
- It is impossible to trisect an angle with a straight edge and compass.
- It is impossible to cut a regular tetrahedron and put the pieces together into a cube of equal volume. (Here cuts are one that can be given with a hyperplane).

Matiyasevich‟s theorem belongs to this general class of theorems. But, it is of utmost importance for us since it talks about impossibility of the existence of algorithms.

In order to **prove** that algorithm cannot exist we must have a exact and mathematically precise definition of an algorithm. This is what is needed in order to show that an algorithm does not exist. Once a precise definition of an algorithm is given we can argue mathematically about them.

Turing Machines and Algorithms

Luckily the mathematical notion of what an algorithm and computation is was already developed for Mataysavich. Let us ask the most fundamental question in theory of computation.
What is an algorithm? What is a computation?

Algorithms and Computation

The seeds of answer to this question were also sown by Hilbert. In 1928, Hilbert who wanted to systematize all of mathematics had posed the **Entscheidungs** problem**.** He wanted an algorithm that would take a mathematical assertion and would decide if the assertion was true or not!!!! That would be an amazing algorithm (if it existed). In 1936, Alan Turing, Alonzo Church showed that the Entscheidungs problem was unsolvable.

The two most famous papers in this regard are:

1. Alonzo Church, "An unsolvable problem of elementary number theory", American Journal of Mathematics, 58 (1936), pp 345 - 363

2. Alan Turing, "On computable numbers, with an application to the Entscheidungs problem", Proceedings of the London Mathematical Society, Series 2, 42 (1936), pp 230 - 265.

We will work with Turing"s model. Turing came up with the idea of computing machines that we now call Turing machines. He proposed that every computation that we can perform mechanically could be performed by a Turing machine. The good thing about Turing machines was:

Turing machines are not an intuitive notion like that of an algorithm. But they are mathematically defined objects. Thus one can prove things about them. One can prove their existence and non-existence also. One can study them with the tools of mathematics and make precise claims about them.

Halting Problem

Turing in the same paper showed that a very important problem called the “Halting Problem” cannot be solved by an algorithm.

This was a breakthrough. It showed that there are mathematically defined precise problems that have no algorithmic solutions. Thus showing us that some problems cannot be computed!!!!!!

Computability Theory

Computability Theory will deal with the following questions:

- What is a computation? Can we come up with a precise definition of an algorithm or a computation? (The answer is yes).
- Can everything be computed. (The answer is no.)

Can we characterize problems that cannot be computed. (The answer is it is hard to do that but in certain cases we can identify them and prove that they are not computable). We will study computable and uncomputable problems in various different areas. We will study computability and logic. We will prove Godel"s incompleteness theorem.

Godel’s incompleteness theorem

Godel"s theorem is a corner stone in mathematical logic and foundations of mathematics. It roughly states that:

- In any (axiomatic) system that includes numbers. There are always statements that are true but not proveable.

This has had profound impact of mathematics. We will prove this theorem in our course using tools from computability theory. We will discuss this in detail when the time comes.

Complexity Theory

Complexity theory is much more practical than computability theory. The basic idea is now we not only want an algorithm but an efficient algorithm to solve problems.

Complexity Theory will deal with the following questions:

1. Given a problem can we devise an **efficient algorithm** to solve that problem?
2. Can we characterize problems that have efficient algorithms?
3. Can we characterize problems that are on the boundary of being efficiently solvable and intractable?
4. Can we prove that problems though computable are intractable?
5. Can we prove that certain problems which are practically significant are intractable?
6. Is there a unified theory of problems that are hard to solve?

One of the most important concept in Complexity Theory is that of NP-completeness. In 1971 Cook and Levin proved a very important theorem in the following papers:

- Stephen Cook (1971). "The Complexity of Theorem Proving Procedures", Proceedings of the third annual ACM symposium on Theory of computing, 151 –158.
- Leonid Levin (1973). "Universal"nye perebornye zadachi". Problemy Peredachi Informatsii 9 (3): 65 English translation, "Universal Search Problems", in B. A. Trakhtenbrot (1984).
- "A Survey of Russian Approaches to Perebor (Brute-Force Searches) Algorithms". Annals of the History of Computing 6 (4): 384400.

This theorem showed that if one given problem (called Satisfiability) could be solved efficiently then many others could also be solved efficiently.

In fact, a whole infinite class of problems can be solved efficiently. This is a very strong evidence that Satisfiability is a hard to solve problem. We are still looking for a proof!!!!!!!

In complexity theory we will talk about the celebrated P vs NP question. Which has a **$1000,000.00** award on it. We will see why the amount stated here is not enough and why this question has far reaching consequences in computer science.

Alphabet, Strings and Languages: LECTURE #2

An alphabet $\sum$ is any finite set
$\sum_1 = \{0, 1\}$
$\sum_2 = \{a, b, c, \ldots, z\}$
$\sum_3 = \{0, 1, x, y, z\}$

1. An string over an alphabet $\sum$ is a finite sequence of symbols from the alphabet.
2. Remember that whenever you talk about a string you must have an alphabet in mind.
3. 01000100 is a string over $\sum = \{0, 1\}$.
4. abracadabra is a string over $\sum_2 = \{a, \ldots, z\}$.

The number of symbols contained in a string w is called the length of w and denoted by |w|.

The string that contains no symbols is denoted by ε. In some books it is also denoted by λ. Notice that the length of this string is 0. It is called the empty string. Let us look at some strings.

1. 01011 is a string of length 5 over $\sum = \{0, 1\}$
2. ε is a string of length 0 over $\sum = \{0, 1\}$.
3. 10xx is a string of length 4 over $\sum_2 = \{0, 1, x, y\}$.
4. abpqrsxy is a string of length 8 over $\sum_3 = \{a, b, \cdots, z\}$.
5. ε is a string of length 0 over $\sum_3 = \{a, b, \cdots, z\}$

Remember that all strings have finite length.

Suppose that $x = x_1x_2 \cdots x_n$ is a string of length n and $y = y_1 \cdots y_m$ is a string of length m, then the concatenation of x and y, written as xy, is the string obtained by appending y to the end of x. Thus $xy = x_1x_2 \cdots x_ny_1y_2 \cdots y_m$.

1. The concatenation of 01 and 10 is 0110.
2. The concatenation of 111 and ε is 111.
3. The concatenation of 10xx and 00yy is 10xx00yy.
4. The concatenation of ε and abc is abc.

individual lengths.

Let x be a string. We let x^k denote the the concatenation of x with itself k times.

1. $01011_3 = 010110101101011$.
2. $\varepsilon^5 = \varepsilon$
3. $1^{10} = 1111111111$.
4. $abc^3 = abcabcabcabc$.
5. Note that $|x^k| = k|x|$.

Let A and B be two sets of strings. We define $AB = \{xy : x \in A, y \in B\}$.

1. A = {00, 11} and B = {x, y}
   - AB = {00x, 00y, 11x, 11y}
2. A = {10, 01} and B = {xx, y, xy}
   - AB = {10xx, 10y, 10xy, 01xx, 01y, 01xy}.

Note that $|AB| = |A| \times |B|$.

Let A be a sets of strings. We define $A^0 = \{\varepsilon\}$ and $A^k = A^{k-1}A$

1. A = {00, 11} then A^2 = {0000, 0011, 1100, 1111}.
2. A = {a, b} then A^2 = {aa, ab, ba, bb}
3. A = {a, b} then

   A^3 = {aaa, aab, aba, abb, baa, bab, bba, bbb}.

Note that $|A^k| = |A|^k$

Let $\sum$ be an alphabet. Then $\sum^k$ is the set of all strings of length k over the alphabet $\sum$.
If $\sum$ = {a, b, c, . . . , z} then $\sum^7$ is the set of all possible strings of length 7, I can make out of these characters. There are 26^7 of them. That is a lot of strings!
Let $\sum^* = \bigcup_{k \geq 0} \sum^k = \sum^0 \cup \sum^1 \cup \sum^2 \cup \sum^3 \ldots\ldots$

Thus $\sum^*$ is the set of all the strings over the alphabet $\sum$. So, for example, if $\sum$ = {0, 1} then {0, 1}* denotes all the strings over the alphabet {0, 1}. Examples of strings in {0, 1}* are:

1. ε
2. 101000
3. 11
4. 10100100100100101001101101001110110100100100010000 1

The lexicographical ordering of strings in $\sum^*$ is the same as the dictionary ordering, except the shorter strings precede the longer strings. Thus to write the strings of {0, 1}* in lexicographical order we first write all strings of length 0 in dictionary order, then all strings of length 1 in dictionary order and so on.....

Thus we get ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000,

There is a very neat way to write the strings of {0, 1}* in lexicographical order. It is very easy for computer scientist to understand as they know how to write numbers in binary.

First write 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ⋯⋯in binary to get 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, and now throw away the starting 1 from each string to get the lexicographical ordering as follows: ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, · · ·

A language L is a subset of $\sum^*$

1. L_1 = { ε, 00, 10, 11, 011, 10001}. L1 is a finite language.
2. L_2 = {x ∈ {0, 1}* : x starts with a 0}.
3. L_3 = {x ∈ {0, 1}* : x ends with a 1}.

Note that

1. 10001 ∈ L_1 but 000 ∉ L_1.
2. 0, 001, 01010, 010101 ∈ L_2 and 10, 11, 111, 100 ∉ L_2.
3. 1, 111, 001 are strings in L_3 and 010, 100 ∉ L_3.
4. 011 is a string that is present in all these languages.

Let us look at a complicated language over {0, 1}*
L_p = {x : x represents a prime number written in binary}.

1. 11∈L_p.
2. 101∈L_p.
3. 10001∈L_p.
4. 10∈L_p.

It is not easy to tell if a string is in L_p. Given a string you will have to check if it represents a prime only then you will find out if it is in L_p.

Machines and Turing Machines:

Consider a language for example Lp which is the set of all strings over primes. It does not seem to be an easy task to tell if a number is prime. So, it would be nice to have a machine that can tell us if a number is prime. The machine can have some input device so that we can enter a string x and ask it if x is a prime number (written in binary).

For now we want to concentrate on machines that only make decisions for us. The machine will have an input device on which we will provide the input. The machine will perform a computation and then tell us if the input belongs to a language or not.

Let us start informally and then come to the formal definition of a Turing machine. A Turing machine has a input tape which is infinite. Hence, the machine has unlimited memory. It has a read/write tape head which can move left and write and change symbols while the machine is performing its computation. The tape initially is blank everywhere. We write an input on the tape and turn the machine on by putting it in a start state.

The machine starts a computation and (may) eventually produces an output. The output can be either accept or reject. The machine produces by going into designated two designated states.

To specify a Turning machine we need to specify the following:

1. We need to specify, what are the symbols that the user is allowed to write on the input tape? In other words, we want to specify the format in which the input will be given to the machine. This is done by specifying an input alphabet ∑.
2. We need to specify what symbols the machine is allowed to write on the tape while it is performing a computation. Thus we need to specify a work alphabet Γ.
3. We need to specify what is the start state of the machine?
4. We need to specify what are the accept and reject states.
5. Lastly, and most importantly, we need to specify how the machine will perform its computation

Turing Machines:

Formally where Q, ∑, Γ are all finite sets.

1. Q is the set of states.
2.
3. Γ is the tape or work alphabet. ∑ ⊆ Γ
4. q_0 is the start state.
5. q_a is the accept state.
6. q_r is the reject state.
7. δ

Let us discuss these one by one.

1. The tape is initially empty. The user can write an input only using symbols of the input alphabet ∑ on the tape. The rest of the tape is left blank. Hence $\sqcup \notin \sum$. Which means $\sqcup$ cannot be a part of the input. Therefore, when the machine reads the first blank it knows that the input has ended.
2. Q is the set of states that the machine can be in.3
3. Γ is the set of symbols that the machine is allowed to write on the input tape. It can write a blank or any other symbol.
4. q_0 is the start state of the machine. Each time the machine performs a computation it starts from the start state.
5. q_a is the accept state, machine enters q_a it declares that it has accepted the input.
6. q_r is the reject state, machine enters q_r it declares that it has rejected the input.
7. δ is the t It describes how the machine will perform its computation.

How does a Turing machine work? We have to understand how it takes a single step. The The Turing machine looks at the current symbol under its tape head and the state that it is in. Depending on that it decides the following three things:

1. What will be its next state?
2. What will it replace the current symbol with?
3. If it will move its head left or right.

This is why the transition function has Q × Γ as its domain. It tells what the machine should do in all possible situations. The range of the transition function is Q × Γ × {L, R}. Therefore, it specifies all the three decisions that the machine has to make.

Let us look at a very simple Turing Machine M_1. In order to describe it I will need to specify all the seven things.

1. The machine has three states, thus, Q = {q_0, q_a, q_r}.
2. The input alphabet is {0, 1}.
3. The work alphabet is {0, 1, □}.
4. The start state is q_0.
5. The accept state is q_a
6. The reject state is q_r
7. The transition function δ_1 is as follows:

The transition function is given by the following table.

State	Symbol	Next State	Replace With	Move
q_0	0	q_a	0	R
q_0	1	q_r	1	R
q_0	□	q_r	□	R
q_a	0	--	--	--
q_a	1	--	--	--
q_a	□	--	--	--
q_r	0	--	--	--
q_r	1	--	--	--
q_r	□	--	--	--

Why have we not specified many entries? That is because, they are irrelevant. When the

Let us look at what this machine will do on various inputs.

1. Suppose the input is ε then the machine starts in q_0. It reads the first symbol on the tape which is a blank. Since, $\delta_1(q_0, □, R) = (q_r, □, R)$. Hence it goes into state q_r and rejects the input. So the machine rejects the input ε.
2. Suppose the input is 0 then the machine starts in q_0. It reads the first symbol which is 0. Since, $\delta_1(q_0, 0) = (q_a, □, R)$. Hence it goes into state q_a and accepts the input. So the machine accepts the string 0.
3. Suppose the input is 1 then the machine starts in q_0. It reads the first symbol on the tape which is a 1. Since, $\delta_1(q_0, 1) = (q_r, □, R)$. Hence it goes into state q_r and rejects the input. So the machine rejects the input 1.

This Turing machine will accept all the strings that start with a 0. Since from the start state it goes to either the accept state or the reject state. Hence, it only reads the first symbol of the input before accepting or rejecting.

Let us look at another example.
I want to describe M_2 so I have to again specify all the seven things.

1. The machine has five states, thus, Q = {q_0, q_1, q_2, q_a, q_r}.
2. The input alphabet is {0, 1}.
3. The work alphabet is {0, 1, □}.
4. The start state is q_0.
5. The accept state is q_a.
6. The reject state is q_r.
7. The transition function 1 is as follows:

The transition function is given by the following table.

State	Symbol	Next State	Replace With	Move
q_0	0	q_1	0	R
q_0	1	q_2	1	R
q_0	□	q_r	□	R
q_1	0	q_1	0	R
q_1	1	q_1	1	R
q_1	□	q_2	□	L
q_2	0	q_r	0	R
q_2	1	q_a	1	R
q_2	□	q_r	□	R

Let us look at what this machine will do on various inputs.

1. Suppose the input is ε then the machine starts in q_0. It reads the first symbol on the tape which is a blank. Since, Hence it goes into state q_r and rejects the input. So the machine rejects the input ε.

2. Suppose the input is 0 then the machine starts in q_0. It reads the first symbol which is 0. Since, $\delta_1(q_0, □) = (q_1, □, R)$. Hence it goes into state q_1 and replaces the current symbol with 0 (so does not change it). Now the machine is in state q_1. Next it reads the symbol □ Since $\delta_2(q_1, □) = (q_r, □, L)$ so it goes into state q_2 and moves left. Now it is reading 0 again but is in state q_2. Since $\delta_2(q_2, □) = (q_r, 0, R)$ the machine rejects this input.

3. Suppose the input is 1 then the machine starts in q_0. It reads the first symbol which is 1. Since, $\delta_1(q_0, 1) = (q_1, 1, R)$. Hence it goes into state q_1 and replaces the current symbol with 1 (so does not change it). Now the machine is in state q_1. Next it reads the symbol □ Since $\delta_2(q_2, □) = (q_r, □, L)$ so it goes into state q_2 and moves left. Now it is reading 1 again but is in state q_2. Since $\delta_2(q_2, 1) = (q_r, 1, R)$ the machine rejects this input.

This is very tedious. So let us work out a better way to following a computation of a Turing machine.

Configurations:

What will happen if we give 00101 as an input to M_2? We can represent its initial configuration with $q_0$00101. This means the machine is in the state q0 reading the symbol that is to its left. The tape contents are 00101. Next it goes into state q1 and moves right we can represent this by 0$q_1$0101

This means it is in state q_1 reading a 0. The tape contents can be easily obtained by ignoring the state in the configuration. Again since $\delta_2(q_1, 0) = (q_1, 0, R)$ the head moves to the right so

we get the next configuration which is $00q_10101$.

Actually we realize that the tape head will keep moving to the right without changing the contents till it reaches a blank. This is because, $\delta_2(q_1, 0) = (q_1, 0, R)$ and $\delta_2(q_1, 1) = (q_1, 1, R)$. Hence the machine will eventually reach the configuration $00101q_1$ at this point the tape head is reading a blank. Now, if we look up the transition function we observe that $\delta_2(q_1, \sqcup) = (q_2, \sqcup, L)$.

So the head has to move to the left now. Hence the machine goes into the configuration $0010q_21$ and since $\delta_2(q_2, 1) = (q_a, 1, R)$ the machine eventually reaches 00101qa and accepts this string.

Can you tell which strings will be accepted by this Turing machine? Yes, it a
that end with a 1. We can see this by examining the transition function again.

Let us once again make sure we understand what a Turing machine is? LECTURE#

Formally a Turing machine M is a 7-tuple, $(Q, \sum, \Gamma, \delta, q_0, q_a, q_r)$, where Q, ∑, Γ are all finite sets.

1. Q is the set of states.
2. ∑ is the input alphabet not containing the special blank symbol _.
3. Γ is the tape or work alphabet. $\sum \subseteq \Gamma$ and $\sqcup \in \Gamma$
4. q_0 is the start state.
5. q_a is the accept state.
6. q_r is the reject state.
7. $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function.

The transition function is given by the following table.

State	Symbol	Next State	Replace With	Move
q_0	0	q_0	0	R
q_0	1	q_1	1	R
q_0	□	q_r	□	R
q_1	0	q_0	0	R
q_1	1	q_1	1	R
q_1	□	q_a	□	L

One again we have not specified what the machine does when it reaches q_a or q_r . Let us see what this machine does since the machine halts when it reaches the accept or the reject state.

Let us intuitively see what each state is doing. Let us look at q0. When the machine is in q_0 represents the fact that the last scanned symbol was not a 1. Note that $\delta(q_0, 0) = (q_0, 0, R)$.

so if the machine reads a 0 it simply stays in the state q_0 and moves right (without changing the input). However, when the machine sees a 1 it changes its state to q_1 as $\delta(q_0, 1) = (q_1, 1, R)$

When the machine is in the state q_1 that represents the fact that the last scanned symbol was 1. Thus if the machine is in the state q1 and sees a 1 it stays in q_1 and if it sees a 0 it goes back to q_0. $\delta(q_1, 0) = (q_0, 0, R)$ and $\delta(q_1, 1) = (q_1, 1, R)$.

Turing Machine State Diagram:

We look the transitions $\delta(q_0, \square) = (q_r, \square, R)$ which means if the input has ended and the last symbol was not 1 then reject the input. Whereas, $\delta(q_1, \square) = (q_a, \square, R)$ if the input has ended and the last symbol was a 1 accept the input.

We can represent a Turing machine with a state diagram.

To make a state diagram follow the following rules:

1. Each state is represented by a node in the state diagram.
2. Clearly, mark the initial state, the accept and the reject state. In this course we will always mark the accept state with q_a and reject state with q_r.
3. For each state (other than the accept and reject states) there should be as many arrows coming out of the state as there are symbols in the work alphabet.
4. Each arrow should be labeled by a replace symbol and a move.

Turning Machine Configuration:

1. Its state.
2. The tape contents.
3. It head position.

Hence a configuration tells us what state

We will represent a configuration in a very special way. Suppose Γ is the tape alphabet. Let u, v $\in$Γ* and q be a state of the Turing machine. Then the configuration uqv denotes

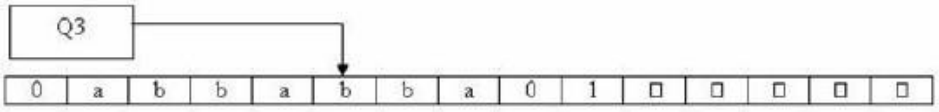
1. The current state is q.
2. The tape contents are uv.
3. The tape head is reading the first symbol of v.

It is assumed that the tape contents are all blanks after the last symbol of v. Let us look at some examples: Let us see the following picture. The machine is in state q_7. The tape contents are 101101111 and the tape head is currently reading the fifth symbol with is a 0.

So in this case, the first four symbols constitute of u and the last five symbols constitute of v. So, this configuration will be written as $1011q_701111$

Now, let us go the other way.
Suppose I tell you that a machine is in configuration: $0abbaq_3bba01$

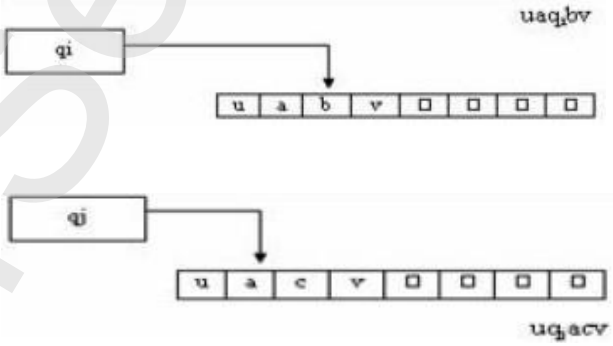
Can you draw a picture? The tape contents are easily recovered by ignoring the state in the configuration. So, the tape contents are 0abbabba01 and the machine is in state q_3. Lastly, it is reading the 6th symbol on the tape.



Let us do one more example: Suppose a TM is in the configuration $01q_2$. What does this mean? In this case, v = ε . So the machine is reading a blank symbol. The picture is given here:

The Yield Relation:

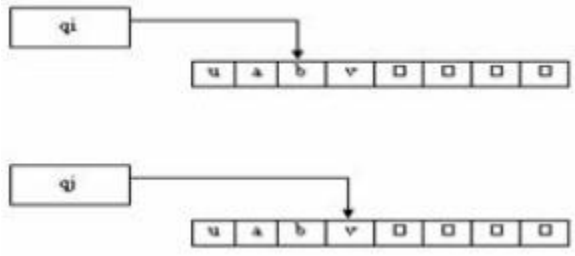
Now, we would like to formalize the way a TM computes or takes a single step. This is done using the yields relation. Suppose that a machine is in a configuration C_1. We say that the configuration C_1 yields another configuration C_2 if the machine can move legally from C_1 to C_2 in a single step. Let us be a bit more formal.



Suppose a, b $\in$ Γ and u, v $\in$ Γ* and q_i, $q_j \in Q$.

We say that uaq_ibv yields uq_jacv if $\delta(q_i, b) = (q_j, c, L)$

This only handles the left moves, for the right moves we define the yields relation as follows: We say that uaqibv yields uacqjv if $\delta(q_i, b) = (q_j, c, R)$



There cases left to

What if the head is at one of the ends of the the configuration? Let us discuss what if the head is at the left end.

- q_ibv yields q_jcv if $\delta(q_i, b) = (q_j, c, L)$
- q_ibv yields cq_jv if $\delta(q_i, b) = (q_j, c, R)$

In this case, we are agreeing that if the machine is at the left most cell it will not move its head further to the left even if the delta function requires us to do so.

What if the head is reading a blank?

- uaq_i yields uq_jac if $\delta(q_i, \square) = (q_j, c, L)$
- uaq_i yields $uacq_j$ if $\delta(q_i, b) = (q_j, c, R)$

Have we discussed all possibilities? One annoying one is left. Consider a machine in a configuration q_i

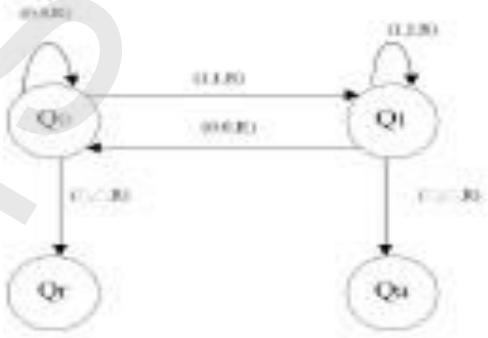
Is this a configuration? Yes, the machine is in state qi and the tape is completely blank and the machine is reading the left most symbol.

- q_i yields q_jc if $\delta(q_i, \square) = (q_j, c, L)$
- q_i yields cq_j if $\delta(q_i, \square) = (q_j, c, R)$

Summary:

- uaq_ibv yields uq_jacv if $\delta(q_i, b) = (q_j, c, L)$
- uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$

Let us look at our previous TM and its diagram and see what the yields relation gives us if we start the machine with input 1001. Let us first look at the state diagram of this TM again.



The start configuration is q01001 and if we apply the yields operation repeatedly we get $1q_1001$, $10q_001$, $100q_01$, $1001q_1$, $1001\square q_a$.

Suppose M is a TM and $w \in \sum^*$. A TM accepts w. If there exist configurations $C_1, C_2, \ldots\ldots, C_k$ such that:

1. C_1 is the start configuration of M on input w; that is, $C_1 = q_0w$
2. Each $C_i \vdash_M C_{i+1}$.
3. C_k is an accepting configuration

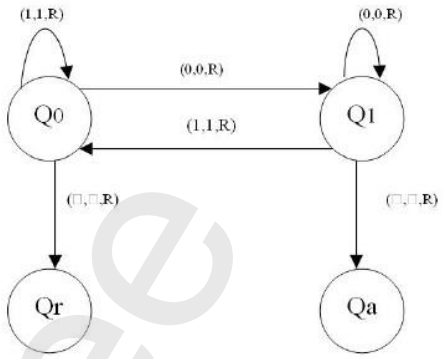
Language accepted by a TM:

The collection of all strings in ∑* that are accepted by M is the language of M and is denoted by L(M).

L(M) = {w ∈∑* : M accepts w}

So, if we go to our previous example:

Then for this TM M
L(M) = {x ∈∑* : x ends with a 0}.



Does a TM only accept or reject inputs? No.
Lets consider M‟

Hence if a string ends with a 0 the machine keeps computing forever.

$q_000 \vdash_{M'} 0q_00 \vdash_{M'} 00q_0 \vdash_{M'} \vdash_M 00\square q_0 \vdash_{M'} 00\square\square q_0$

What is the language accepted by M‟? By definition it is the set of all strings accepted by M‟. Hence,

L(M‟) = {x ∈ {0, 1}* : x ends with a 0}.

We have

-
-

We say that a Turing machine is a decider if it halts on all inputs in ∑*. Hence M is a decider and M‟ is not a decider.

Designing Turing Machine:
The most usual question in computer science can be posed as follows:

Given a language L_0 design a decider, M, such that: $L(M) = L_0$

L_0 is the given problem and M is the algorithm that solves or decides the problem. Let us look at an example of this.

Let ∑ = {0}, $L_2 = \{ 0^{2^n} : n \geq 0\}$. We want to design a TM M_2 such that $L(M_2) = L_2$. Hence we want a TM M_2 that accepts all the inputs in L_2. In order to solve this problem we must first understand the language L_2.

L_2 = {0, 00, 0000, 00000000, 0000000000000000, . . . , }

To design a TM we must have a plan in mind. How will we solve this problem? Note that a TM can only have finite number of states and therefore it cannot remember much. It does not even know how to count (unless you teach it).

If I have some number of 0"s and I cross each alternate one I will cut the number of 0"s into half.

Informal description of the M_2:

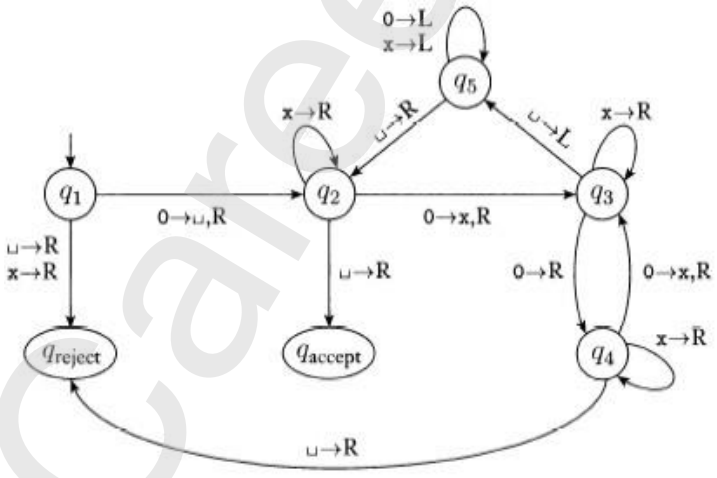
1. Sweep from the left to the right crossing off each other 0.
2. If there was a single 0 on the tape, accept.
3. If there were more then one 0"s and the number was odd, reject.
4. Return the head to the left end and repeat.

Why would this work. If the number of 0"s is 16, it will be cut into 8 and then 4 and then 2 and then 1 and then we will accept. On the other hand if the number of 0"s is say 20, it will be cut to 10 then 5 and then we will reject.

Let us look at the formal description of this Turing Machine.
The machine $M_2 = (Q, \sum, \Gamma, \delta, q_1, q_a, q_r)$ where

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_a, q_r\}$
- $\sum = 0$
- $\Gamma = \{0, x, \square\}$
- q_1 is the start state, q_a is the accept state and q_r is the reject state.
- δ is described by the following state diagram.

Let looked at the language A={ 0^{2^n} : n $\geq$ 0}, and we designed a TM for it.



$q_1$0000
␣$q_2$000
␣x$q_3$00
␣x0$q_4$0
␣x0xq_3␣
␣x0q_5x␣
␣x$q_5$0x␣
␣q_5x0x␣
q_5␣x0x␣
␣q_2x0x␣
␣x$q_2$0x␣
␣xxq_3x␣
␣xxxq_3␣
␣xxq_5x␣
␣x$q_5$xx␣
␣$q_5$xxx␣
q_5␣xxx␣
␣$q_2$xxx␣
␣x$q_2$xx␣
␣xxq_2x␣
␣xxxq_2␣
␣xxx␣q_{accept}

Now let us look at another language B = {w # w: w ∈{0, 1}*}. In order to design a TM we must understand the language first.

Let us look at some examples of strings that are in B:

0110#0110, 0#0,#, 000#000.

Let us look at some examples of strings that are not in B:

1010#1100, 0110, #0, 1#, 000#000#, 0#0#0, ε

The machine will do the following:

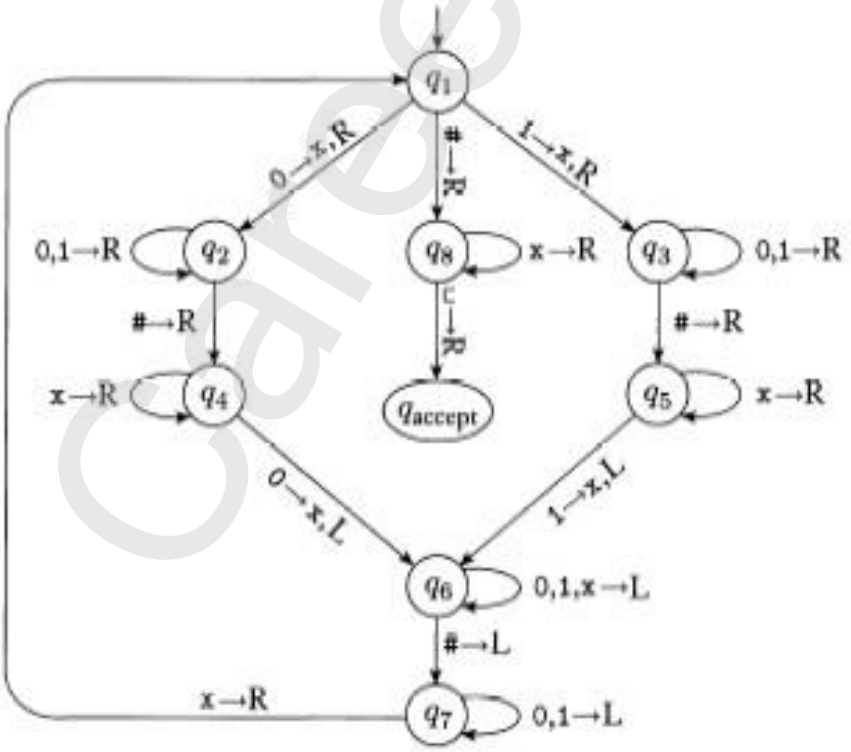
1. In the first pass the machine will make sure that the input is of the form w1#w2 where w_1, $w_2 \in \{0, 1\}^*$.
2. Then the machine will repeatedly scan the input and match each character of w_1 with one character of w_2.

We define our TM M

1. $Q = \{q_1, \ldots, q_{14}, q_a, q_r\}$
2. $\sum = \{0, 1, \#\}$
3. $\Gamma = \{0, 1, \#, x, \square\}$
4. q_1, q_a, q_r the start, accept and reject states respectively.
5. δ is given by the state diagram.

In this diagram we have not shown q_r. All transitions not defined here go to q_r.

Transition diagram of the Turing Machine.



Let us look at the computation of this TM on the input 011#011. As you can see in the first phase the machine has converted the input into □11#011.Now it is looking for a 0 in the second string. When it finds a 0 it crosses it and starts all over again. Next it crosses of the 1 and is looking for a 1 in the second string □x1#x11and is looking for a 1.

It finds a 1 after the # and crosses it. □x1#xx1 Now it goes back and similarly matches the last 1 to finally reach □xx#xxx Then it goes over the tape once and realizes that all things have been matched so it accepts this input. Put Animation here from sipser‟s book.

Let us look at the another language: C = $\{a^i b^j c^k : k = i \times j \; i, j, k \geq 1\}$.

Firstly all the strings in this language have a‟s then b‟s then c‟s. Some strings in the language are: abc, aabbcccc, aaabbcccccc. Some strings not in the langauge are ab, aabb, aabbccc, ba, cab.

Lets give an informal description of a TM that recognizes C.

On input w

1. Scan the input and make sure that w is of the form a*b*c* if it is not reject.
2. Cross off an a and scan to the right till a b occurs. Now, shuttle between b‟s and c‟s crossing one b and one c till all b‟s are crossed off. If you run out of c‟s reject.
3. If all a‟s are crossed off. Check if all c‟s are also crossed off. In that case accept.
4. If there are more a0s left. Restore all b‟s and repeat.

On input aabbbccccccc, we will first check if the string is of the form a*b*c* and it is. So in the next state we cross off an a x‟abbbccccccc. Since there are three b‟s we shuttle between b‟s can c‟s crossing them off.

To get to x‟axxxxxxccc, next there is still an a so we restore all the b‟s to get x0abbbxxxccc. Now, we cross of another a to get x‟xbbbxxxccc. Now, shuttle between b‟s and c‟s crossing them off to get x‟xxxxxxxxxx. Now check if there are any a‟s left. No so we accept.

One question.
Why is the left most x and x_0. The answer is that this way the machine will know that it has reached the left end of the tape.

Why will this work.
Note for each a we are crossing off as many c‟s as there are b‟s. So if the number of a‟s is i and the number of b‟s is j then each time j, c‟s are being crossed off. So the total number of c‟s crossed off is i × j. Thus if the number of c‟s is equal to i × j we accept.

This is a high level description of a TM. It is your home work to convert this high level description to a formal description. In your home work you have to specify 7 things.

The set of states, the start, accept and reject state, the input alphabet, the work alphabet and a transition function (for which you can draw a state diagram).

Let us look at another example given in the book.
The language E = {#x_1#x_2#. . .#x_l : each $x_i \in \{0, 1\}^*$ and $x_i \neq x_j$ for each $i \neq j$ }
This problem is called element distinctness problem. We want to make sure that all the strings x_1,……..,x_l are distinct.

Here is a high level description of the Turing machine M4 that accepts E.

On input w
1. Place a mark on top of the left most tape symbol. If that symbol is not # reject.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered and a blank is encountered accept.
3. By zig-zagging compare the two strings to the right of the marked #. If they are equal reject.
4. Move the rightmost of the two marks to the next # symbol. If no # is encountered move the left most marker to the next #. If no such # is available all strings are compared, so accept.
5. Go to state 3

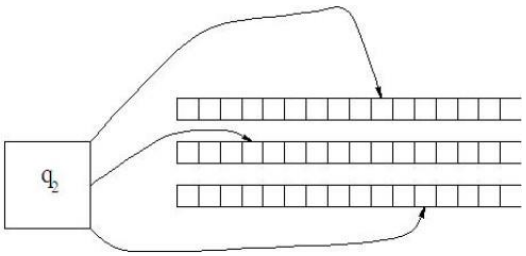
Suppose the input is #0#10#11#000. We start by marking the first # sign. This means the first string has to be compared will all the strings $\bar{\#}$0$\bar{\#}$1011#000. It is first compared with the second string. $\bar{\#}$0#10$\bar{\#}$11#000 then $\bar{\#}$0#10#11$\bar{\#}$000 then there are no more # signs. So we move onto 10. #0$\bar{\#}$10$\bar{\#}$11#000 then #0$\bar{\#}$10#11$\bar{\#}$000 then #0#10$\bar{\#}$11$\bar{\#}$000 after comparing them we are done.

There is a simple algorithm behind this TM.
Let l be the number of strings on the tape.

```
For i = 1, 2,……..,l
   For j = i + 1,……..,l
      Compare xi and xj
       If they are equal
          Reject.
       Accept.
```

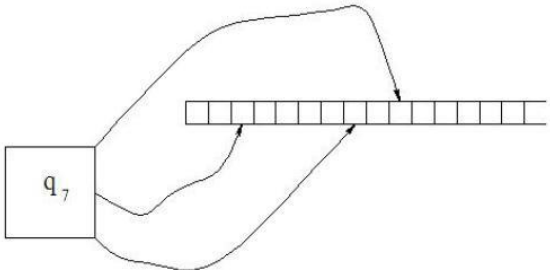
From now on. We will only give informal description of TM"s. But, you should be able to convert any informal description to a formal one. Most interesting TM have several states hundreds even thousands. However, it will be clear that an informal description can be changed to a formal one.

Variants of Turing Machines:
Let us look at an example of a multi-tape Turing machine.
A k-tape TM is a 7-tuple:
1. Q is the set of states.
2. $\sum$
3. Γ
4. q_0, q_a, q_r
5. δ is now a function from $Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R\}^k$.



Multiple head Turing machines.
A k-tape TM is a 7-tuple:

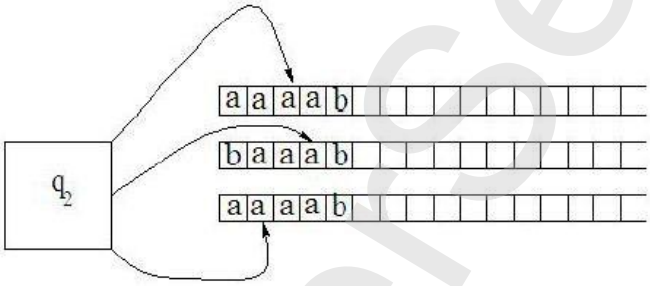
1. Q is the set of states.
2. $\sum$
3. Γ
4. q_0, q_a, q_r
5. δ is now a function from $Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L,R\}^k$.



We will show that multi-tape TMs are not more powerful than single tape TMs.

Theorem: Every multi-tape TM has an equivalent single tape TM.

A single tape TM can actually "simulate" a multi-tape TM. Let us look at a 3-tape TM.
The configuration of this TM is described as follows: q_3#aa˙aab#baa˙ab#a˙aaab
The main point is that this configuration can be written on a single tape.



The method for writing a configuration on a single tape is:

1. Write the state first followed by #.
2. Write the contents of the first tape followed by a #.
3. Write the contents of the second tape followed by a #.
4. Write the contents of the third tape followed by a #.
5. Place dots on all symbols that are currently been scanned by the TM.

Suppose that I have only a single tape. I want to know what a multi-tape TM will do on input w. I can write q_0#w"#□˙#□ ˙ Which denotes the initial configuration of M on w.

Here w" is the same string as w except the first character has a dot placed on top of it. Now, I can keep scanning the entire tape to find out which symbols are being read by M. In the second pass I can make all the modifications that M would have done in one step. Given a k tape TM M we describe a single tape TM S. S is going to simulate M by using only one tape.

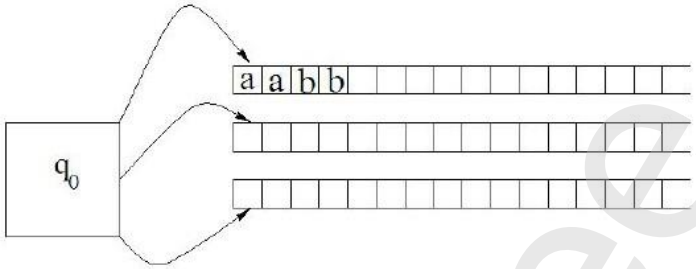
Simulation of Multi Tape Turing Machine:

Description of S:
On input w = w_1…….w_n

1. Put the tape in the format that represents the initial configuration of M; that is, #q_0w_1˙ w_2 .....w_n # □˙ # □˙……# □˙ #
2. To simulate a single move of M, S scans it" tape from left to right. In order to determine the symbols that are placed under it"s "virtual heads". Then it makes a second pass to update the contents of the tapes according to the transition function of M.

3. If at any point S has to move a head to the right on to # that is if M is moving to a previously unread black. Then S inserts □˙ by shifting symbols to the right.
4. If S reaches an accepting configuration then it accepts and if it reaches a rejecting configuration then it rejects.

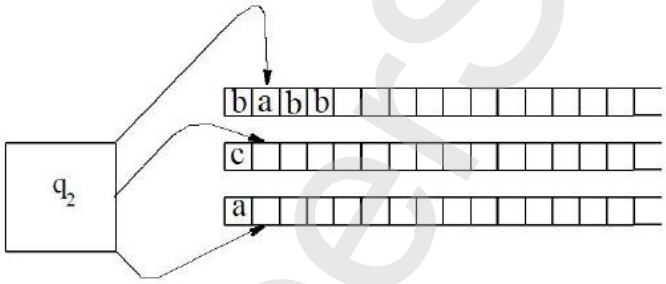
Suppose we have a three tape Turing machine and we give it the input aabb. The machine is in the following configuration:



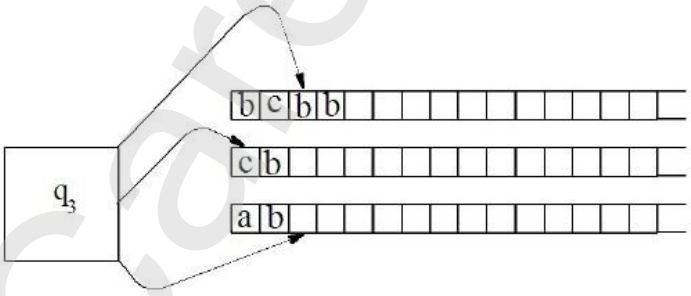
This configuration will be written on a single tape of S as: #q_0a˙abb#□˙#□˙# and suppose

$$\delta(q_0, a, □, □) = (q_2, b, c, a, R, R, R).$$

That means the next configuration will be written as: #$q_2$ba˙bb#c□˙#a□˙#. This represents the machine in the following configuration:



Now, if $\delta(q_2, a, □, □) = (q_3, c, b, b, R, L, R)$ then #$q_3$bcb˙b#c˙b#ab□˙# represents the next configuration.



In order for S to simulate a single step of M it has to make two passes.

1. In the first pass it finds out the current symbols being scanned by M"s heads.
2. In the second pass it makes the required changes to the tape contents.

Theorem: M accepts w if and only if S accepts w.

Thus we have proved the our theorem:

Theorem: Every multi-tape TM has an equivalent single tape TM.

A few words about this theorem:

1. We have only given a high level description of S. You should convince yourself that given any k-tape TM you can come up with a exact and formal definition of S. Which essentially means you can fill all the details in the above proof.
2. Work alphabet of S will be much larger than that of M.

Another important point to note in this proof:
It seems like S is not very efficient. To perform one step of M it makes two passes over the entire tape. However, efficiency is not our concern in computability theory (it will be when we study complexity theory). It is your homework to prove the following theorem.

Theorem: Every k-head TM has an equivalent single tape TM.

The proof of this theorem is almost a ditto copy of the previous theorem. However, there are some subtle differences. You must write the proof in detail to understand these differences.

Can we use this theorem to make our lives simpler? Yes. Suppose someone asks you if the language L = {w#w: w∈{0, 1}*} is decidable. It is much simpler to design a two tape TM for this language.

Description of a 2-tape TM that accepts L:

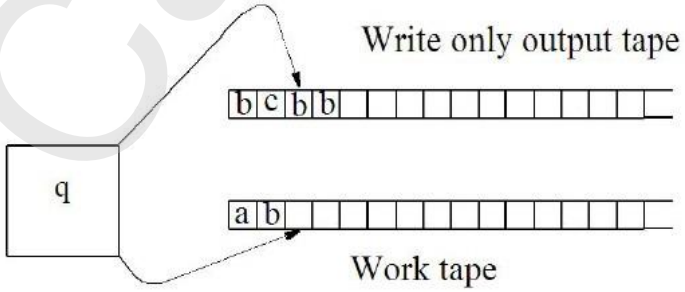
1. In the first pass make sure the input is of the form $w_1\#w_2$.
2. Copy the contents after # to the second tape.
3. Bring both tape heads all the way to the left.
4. Match symbols of w_1 and w_2 one by one to make sure they are equal.

Note that this is a much more natural solution than matching symbols by shuffling over the input.

From now on we will mostly describe multi-tape TMs. By this theorem we can always find a one-tape TM that is equivalent to our multi-tape TM. This theorem is going to be extremely useful.

Enumerators:

Let us now look at another model of computation called enumerators. Enumerators are devices that do not take any input and only produce output. Here is a picture of an enumerator:



Enumerator"s are another model of computation. They do not take any input and only produce output.

Let us define them formally:

An enumerator E = (Q, ∑, Γ, q_0, δ)

1. Q is the set of states.
2. ∑ is the output alphabet and #, □, ∉, ∑
3. Γ is the work alphabet and □∈Γ.
4. q_0 is the start state.
5. δ is the transition function.

Well what is the domain and range of δ: Q × Γ ↔ Q × Γ × {L, R} × (∑ ∪ { ε ,#})

L(E) the language enumerated by E is the set of all strings that it outputs.

Given an enumerator E Can we make a TM M such that L(E) = L(M); that is, the language accepted by M is the same as the one enumerated by E. This is easy M simply has E encoded in it. On input x it runs E and waits for x to appear. If it appears it accepts.

The answer is yes. Consider a TM M and suppose:

1. ε is accepted by M.
2. 0 is rejected by M.
3. 1 is accepted by M.
4. On 00 M loops forever.
5. 01 is accepted by M.

We want to make an enumerator that outputs ε, 1, 01, …….

We can try to make an enumerator as follows:

1. For each string x = ε, 0, 1, 00, 01, 10, 11,…….
2. Run M on x; if M accepts output x

This does not work. Since, M loops for ever on 00 thus it will never output 01!

How does the a configuration of E look like: α q β , w_1#w_2# w_{k-1}#x

Where α , β ∈(Γ ∪ □)* and x Ɛ (∑* ∪ ∑*#)

Write the definition of the yields relation for enumerators.
We say that E outputs a string w if it reaches a configuration α q β ,w_1#w_2 …. w_{k-1}#w#

L(E) the language enumerated by E is the set of all strings that it outputs. Last time we outlined a proof of the following theorem.

A language L is Turing-recognizable if and only if some enumerator enumerates it.

One side was easy: Let L be a language enumerated by an enumerator E. The L is Turing recognizable. To prove this we are given that an enumerator E enumerates L. We have to show that some Turing machine M accepts L.
The informal description of the Turing machine is as follows:

1. Input x
2. Run the Enumerator E each time it outputs a string w check if w = x. If w = x go to accept state. Otherwise, continue.

The other side was a bit tricky. But let us look at the details. If L is Turing recognizable then there is an enumerator E that enumerates L. Since L is Turing recognizable therefore some TM M accepts L. We can use M as a subroutine in our enumerator E. Here is our first attempt.

Recall that we know how to generate strings in lexicographical order as ε, 0, 1, 00, 01, 10,….,

1. For i = 1,……,
2. Simulate the machine M on x_i
3. If M accepts x_i output x_i

Why is this wrong?

Lets say

1. M accepts ε
2. M rejects 0.
3. M accepts 1.
4. M loops forever on 00.
5. M accepts 01.

Then the enumerator will never output 01. Whereas, $01 \in L$. So, we have to get around this problem.

The solution is given by dovetailing:

1. For i = 1, 2, 3, . . . ,
2. For j = 1, 2, . . . , i
3. Simulate M on x_j for i steps only.
4. If M accepts x_j output x_j.

Dovetailing:

We can use dovetailing to get over this problem. The idea is to use “time sharing” so that one bad string does not take up all the time.

Suppose $x_t \in L$. Then M accepts x_t and it must accept x_t is some finite number of steps. Lets say this number is k. Then we claim that when i = max(k, t) the string x_t will be outputted. That is because M is simulated for on x_1, . . . , xi for i steps. Since, $i \geq t$ hence $x_t \in \{x_1, \ldots, x_i\}$ and since $i \geq k$ hence M will accept x_t in k steps and x_t will be outputted.

Hence all strings that are in L are enumerated by this enumerators. To see if $x \notin L$ then E will not enumerate it, we simply observe that E only enumerates strings that are accepted by M.

A minor point to notice is that E enumerates a given string many many times. If this bothers you, you can modify E so that it enumerates each string only once.

Now, E keeps a list of all the strings it has enumerated and checks the list before enumerating every string if it has been enumerated before. Show that if L is Turing recognizable then there is an enumerator E that enumerates L. Furthermore, each string of L is enumerated by E exactly once.

Definition of an algorithm:

- Intuitive notion of algorithms equals Turing machine algorithms
- Another way to put it: An algorithm is a Turing machine that halts on all inputs.
- An algorithm is a decider.

Church-Turing Thesis:
The Church-Turing thesis states that the intuitive notion of algorithms is equivalent to the mathematical concept of a Turing machine. It gives the formal definition for answering questions like Hilbert"s tenth problem.

Encodings:
Note that we always take inputs to be strings over some alphabet $\sum^*$
Suppose $\sum$ = {a, b, c, d} we can map

1. a to 00; that is e(a) = 00.
2. b to 01; that is e(a) = 01.
3. c to 10; that is e(a) = 10.
4. d to 01; that is e(a) = 11.

This way we can encode any alphabet to {0, 1}. Note that this maps $\sum^*$ to {0, 1}* in a natural way. $e(w_1w_2\ldots\ldots.w_n) = e(w_1)\ e(w_2)\ldots\ldots\ldots e(w_n)$

Suppose $L \subset \sum^*$ we can define L_e = {e(w) : w∈L}. Designing a TM for M or designing a TM for L_e is equally difficult or easy.

We want to solve problems where the input is an interesting object such as a graph, or a matrix. However, our definitions only allow us inputs which are strings. But, this is usually not a restriction. Let us look at a few examples: L = {(i, j): $i^2 = j$}

(1, 1), (2, 4), (3, 9) are in L and (1, 3), (2, 5), (3, 1) are not in L. L is a subset of N × N.

What we can do is to encode the pairs into strings. Let us say we reserve 11 to represent a 1, 00 to represent a 0 and 01 will be the separator. Then we can encode (2, 4) to 110001110000

We represent the encoding of a pair (i, j) by < (i, j)>. Then we can solve the following problem L" = {< (i, j) >: $i^2 = j$}. This is a language over {0, 1}*.

Let us look at another example: D = {A : A is a n × n with integer entires and det(A) = 0}

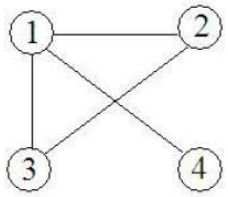
How can we encode this problem so that it becomes a language over {0, 1}*?
Lets say we have the matrix

$$\begin{pmatrix} 2 & 5 \\ 3 & 6 \end{pmatrix}$$

First we can write it as linearly as 2, 5, 3, 6. Now, we can write all the numbers in binary where 00 represents a 0, 11 represents a 1 and 01 represents the separator, So

$$\left\langle \begin{pmatrix} 2 & 5 \\ 3 & 6 \end{pmatrix} \right\rangle = 110001110011010111101111100$$

Now, we can define the following language D" = {<A> : A is a n × n with integer entires and det(A) = 0}. Similarly we can encode graphs into strings: Let us do an example: Consider the graph shown in the following figure:



This graph can be written as a vertex list followed by an edge list as follows:
(1, 2, 3, 4)((1, 2), (2, 3), (3, 1), (1, 4)).

1. 101 represents).
2. 110 represent (.
3. 011 represents ,.
4. 000 represents 0
5. 111 represents 1.

and now it is easy to encode this whole graph into a 0/1 string. The encoding starts as 1101110111110000111111110111110000001011101101110111110001010111110……. which is very long but not conceptually difficult.

Thus suppose we have L = {G : G is connected }.
We can instead look at L"= {<G> : G is connected }.

This should not be surprising for you. As computer scientist you know that everything inside a computer is represented by 0"s and 1"s.

Hilbert's Tenth Problem:
Recall from the first lecture:
Hilbert"s tenth problem was about diaphantine equations. Let us cast it in modern language.

Devise an algorithm that would decide if a (multi-variate) polynomial has integral roots

Here is an example of a multivariate polynomial: $6x^3yz^2 + 3xy^2 - x^3 - 10$.

Hilbert asked to study the language H = {<p> : p is a polynomial with integral roots}.
Once again <> mean encodings. Is there a Turing machine M that halts on all inputs and L(M) = H

Matijasevi˘c Theorem:
H is not decidable. There is no Turing machine M that halts on all inputs with L(M) = H.

Let look at a simpler language: H_1 = {<p> : p is a polynomial in x and has an integral root}. Lets see that H_1 is Turing decidable:
Here is the description of a TM

1. On input p.
2. for i = 0, 1, 2, . . . ,
3. evaluate the polynomial on x = i and x = −i. If it evaluates to 0 accept.

Note the above machine is not a decider. It is only a recognizer.
Show that H_1 is also Turing decidable.

How to describe a Turing machine:
There are three ways to do it.

1. Formal description.
2. Implementation description.
3. High-level description.

In a formal description you have to specify the whole TM using the 7-things needed. That is you have to specify $(Q, \sum, \Gamma, q_0, q_a, q_r, \delta)$.

We have seen examples of this in the last two lectures.

it moves and performs its task. Let look at an example $L = \{w\#w : w \in \{0, 1\}^*\}$.

A TM can be described as follows:

1. Scan the input and make sure it is of the form $w_1\#w_2$ where $w_1, w_2 \in \{0, 1\}^*$. Or in other words make sure there is only one # in the input.
2. Zigzag over the input matching symbols of w_1 and w_2.
3. If all symbols match then accept. If any symbol fails to match reject.

1. On input j.
2. For i = 1, . . . , j
3.
4. Reject

Note that in this case, we are assuming many things.

We are assuming that we can build a machine (subroutine) that can multiply numbers, check for equality etc.

Decidability

Decidable Languages:

Let us recall that a language A is called decidable if there exists a TM, M, that halts on all inputs and L(M) = A.

1. If $x \in A$ then M accepts x
2. If $x \notin A$ then M rejects x

Intuitively M is just an algorithm that solves the yes/no problem A.

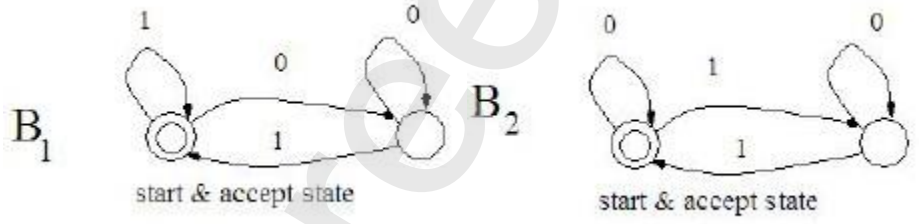
The Acceptance Problem for DFAs:

with a read only head. Let us look at the following problem:

A_{DFA} = {<B, w> : B is a DFA that accepts the input string w}.

the DFA B accepts the string w.

Note that the input will not be a DFA B but a suitable encoding of B. Let us look at the following picture which shows two DFAs:



In this case:

1. $<B_1, 1> \in A_{DFA}$.
2. $<B_1, 000> \notin A_{DFA}$.
3. $<B_2, 1> \notin A_{DFA}$.
4. $<B_2, 000> \in A_{DFA}$.

We want to show ADFA is decidable.
The question that we want to ask is: Is A_{DFA} decidable? Lets ponder on this question for a bit.

We are asking if we can make an algorithm (Turing machine) that will take as input a description of a DFA B and a string $w \in \{0, 1\}^*$ and tell us if B accepts w?

The answer is yes. We can devise such an Turing machine. Since, aTuring machine can "simulate the behavior of a DFA."Here is a high-level description of such a Turing machine M_D.

1. On input <B, w>
2. Simulate B on w.
3. If B reaches an accepting state then accept. If it reaches a rejecting state then reject.

We can give a more detailed level description of this Turing machine.
An implementation level description:

1. On input <B, w>
2. Copy the transition function of B on your second tape.
3. Write the current state on the third tape.
4. Write the w on the first tape.
5. For i = 1, . . . , |w|
6. Simulate a single step of the DFA by consulting the transition function on the first tape and the current state on the third tape.
7. If the state on the third tape is a final state accept. Else reject.

We can also look at the acceptance problem for NFAs.

A_{NFA} = {<B, w> : B is a NFA that accepts the input string w}

Is ANFA decidable? Once again think of the question intuitively.

We are asking if we can make an algorithm (Turing machine) that will take as input a description of a NFA B and a string w 2 {0, 1}* and tell us if B accepts w?

There are two ways we can approach this problem. The first one is very similar to the previous one. We can show that a Turing machine can simply "simulate" an NFA. All it has to do is to keep track of the set of states that the NFA is in.

However, there is another approach. We know that

(description of an) equivalent DFA. So we have the following solution also:

1. On input <B, w> where B is a NFA and w is a string.
2. Convert B to an equivalent DFA C.
3. Run the TM MD on input <C, w>
4. If M_D accepts accept. If it rejects reject.

Regular expressions:

We can also look another problem:
A_{REX} = {<R, w> : B is a regular expression that generates string w}

1. <0*1*, 0011>$\in A_{REX}$
2. <0*1*, 10011>$\notin A_{REX}$
3. <1(0 + 1)*, 10>$\in A_{REX}$
4. <1(0 + 1)*, 0011>$\notin A_{REX}$

Is A_{REX} decidable? Once again think of the question intuitively. We are asking if we can make an algorithm (Turing machine) that will take as input a description of a regular expression R and a string w$\in${0, 1}* and tell us if R generates w?

Once again, there is another approach. We know that every REX can be converted to an equivalent DFA. In fact, there is an algorithm that can convert any (suitable description of an) regular expression to a (description of an) equivalent DFA.

So we have the following solution also.

1. On input <R,w> where R is a regular expression and w is a string.
2. Convert R to an equivalent DFA C.
3. Run the TM MD on input <C,w>
4. If MD accepts accept. If it rejects reject.

The Emptiness Problem for DFAs:

Lets look at another problem which is more interesting. E_{DFA} = {<A> : A is a DFA and L(A) = Φ }. The input to this problem is a description of a deterministic finite automata A. We have to decide if the L(B) = Φ .

The problem on the surface looks very difficult. We have to make sure that A does not accept any strings. However, the number of potential strings is infinite. Thus it would be futile to simulate A on all the strings. The process will never end.

So we may start suspecting that this is perhaps a difficult question and designing a TM that tells us in finite time if L(A) = Φ may not be possible. However, a little thought shows that it is possible to find a TM that decides this problem.

The idea is that if we want to tell if the language of a DFA is non-empty. All we are trying to see is if it is possible to reach some accepting state. So, we can use the following method:

1. On input <A>
2. Mark the state of A.
3. For i = 1, 2,……..,n where n is the number of states.
4. Mark any state that has a transition coming into it from any state that is already marked.
5. If any accept state gets marked reject. Otherwise, accept the input.

This method can be explained in one line as follows:

1. On input <A>
2. Perform a DFS on the transition diagram of the A starting from the start state.
3. If any final state gets marked reject. Otherwise accept.

Notice that we managed to avoid a potential infinite computation by a clever method. If the problem looks difficult think again. Think of ways to avoid potential infinite computations.

The equivalence Problem for DFAs:

Lets look at another problem which is more interesting.

EQ_{DFA} = {<A, B> : A and B are DFAs and L(A) = L(B) }.

The input to this problem is a description of two deterministic finite automata A and B. We have to decide if the L(A) = L(B). That is if they accept exactly the same language.

Note the problem on the surface again looks very difficult. We have to make sure that A and B accept exactly the same strings and reject exactly the same strings. However, the number of potential strings that we can as input is infinite. Thus it would be futile to simulate A and B on all the strings. The process will never end.

So we may start suspecting that this is perhaps a difficult question and designing a TM that tells us in finite time if L(A) = Ø may not be possible. However, a little thought shows that it is possible to find a TM that decides this problem.

The idea is that if we want to tell if the language of a DFA is non-empty. All we are trying to see is if it is possible to reach some accepting state. So, we can use the following method:

1. On input <A>
2. Mark the state of A.
3. For i = 1, 2,……..,n where n is the number of states.
4. Mark any state that has a transition coming into it from any state that is already marked
5. If any accept state gets marked reject. Otherwise, accept the input.

This method can be explained in one line as follows:

1. On input <A>
2. Perform a DFS on the transition diagram of the A starting from the start state.
3. If any final state gets marked reject. Otherwise accept.

Notice that we managed to avoid a potential infinite computation by a clever method.
If the problem looks difficult think again. Think of ways to avoid potential infinite computations.

Lets look at another problem which is more interesting.
EQ_{DFA} = {<A, B> : A and B are DFAs and L(A) = L(B)}.

The input to this problem is a description of two deterministic finite automata A and B. We have to decide if the L(A) = L(B). That is if they accept exactly the same language.

Note the problem on the surface again looks very difficult. We have to make sure that A and B accept exactly the same strings and reject exactly the same strings. However, the number of potential strings that we can as input is infinite. Thus it would be futile to simulate A and B on all the strings. The process will never end.

So we may start suspecting that this is perhaps a difficult question. We have seen an example like this already.

Note that L(A) ≠ L(B) if either

1. there is an x ∈ L(A) ∩ L(B)
2. there is an x ∈ L(B) ∩ L(A)

In short if (L(A) ∩ L(B)) ∪ (L(B) ∩ L(A)) = Ø then L(A) = L(B)

Now, we know that the following properties of regular languages.

1. If P is regular and Q is regular then P ∪ Q is regular. Furthermore, given two DFAs D_1 and D_2 we can construct a DFA such that D such that L(D) = L(D_1) ∪ L(D_2).
2. If P is regular then $\overline{P}$ is regular. Furthermore, given a DFA D" we can construct a DFA D_0 such that L(D") = L(D).

Hence we can make a DFA C such that: L(C) = (L(A) $\cap$ $\overline{L}$(B)) $\cup$ (L(B) $\cap$ $\overline{L}$ (A)) = Ø

1. On input <A, B>
2. Construct the DFA C described above.
3. If L(C) = Ø accept. Else reject.

The Acceptance Problem for CFGs:
You have studied CFG in your previous automata course. Let us look at the following problem:

A_{CFG} = {<G, w>: B is a CFG that generates w}.

The input to this problem consists of two parts. The first part is going to be a CFG G. The second part is going to be a string w. We have to decide if the G generates w. Note that the input will not be a CFG G but a suitable encoding of G.

The question that we want to ask is: Is A_{CFG} decidable?
Lets ponder on this question for a bit.

We are asking if we can make an algorithm (Turing machine) that will take as input a CFG G and a string w$\in${0, 1}* and tell us if G generates w?

One idea is to try all possible derivation and check if we ever get the string w. However, we do not know how many derivations to try and this can be potentially infinite.

But we know how to convert a grammar into Chomsky Normal Form. In chomsky normal form every string of length n has a derivation of length at most 2n − 1.

1. On input <G, w>
2. Convert G into Chomsky normal form G".
3. List all derivations with 2n − 1 steps.
4. If one of them generates w accept. Otherwise reject.

Note the above method is not the most efficient. You have studied more efficient algorithms for this problem. You can recall the CYK algorithm.

The Emptiness Problem for CFGs:
Lets look at another problem which is more interesting.

E_{CFG} = {<G>: A is a CFG and L(A) = Ø}.

Note the problem on the surface looks very difficult. We have to make sure that G does not generate any strings. In other words, we would like to show that starting from the start symbol of G we can never get to a string which consists only of terminals. Since, there are potentially infinite number of derivations the problem looks difficult.

However, a little thought shows that it is possible to find a TM that decides this problem. The idea is that if we want to tell if the language of a CFG is non-empty. All we are trying to see is if it is possible to generate all terminals from the start symbol.

So, we can use the following method:

1. On input <G>
2. Mark all the terminals symbols of G.
3. For i = 1, 2,….,n where n is the number of variables.
4. If $U \rightarrow U_1 \ldots\ldots, U_k$ is a rule and $U_1,\ldots\ldots,U_k$ are all marked then mark U
5. If the start symbol gets marked reject. Otherwise, accept.

The equivalence Problem for CFGs:
Lets look at another problem which is more interesting.

$$EQ_{CFG} = \{<G_1, G_2>: G_1 \text{ and } G_2 \text{ are CFGs and } L(G_1) = L(G_2)\}.$$

Note the problem again looks very difficult. We have to make sure that G_1 and G_2 generate exactly the same strings.

Question: Is EQCFG decidable?

Acceptance Problem for TMs:
Let us define

$$A_{TM} = \{<M,w>: M \text{ is a TM that accepts } w\}.$$

This is called the acceptance problem for TMs or the halting problem.

Theorem: ATM is Turing Recognizable.

1. On Input <M, w>
2. Simulate M on w.
3. If M accepts w accept.

This machine lets call it U is a recognizer for A_{TM}. Why is this not a decider? If M loops on w then U will also loop on <M, w>.

U is a general purpose computer. M is the program and w are the data. That is what happens in a typical computer system. We make one kind of machine and run different programs on various data on it.

The fact that Universal TM may look simple to us now but it is extremely important. Notice there are no:

1. Universal DFAs
2. Universal PDAs
3. Universal CFGs

However, there is a Universal TM. Turing machines are powerful enough to simulate DFAs, PDAs and other Turing machines as well.

What happens to U if the input is a machine M and data w such that M does not halt on w then U does not halt on <M, w>.

We can ask the question is ATM decidable. So, we are asking if there is a decider H such that
1. H will accept <M, w> if M accepts w.
2. H will reject <M, w> if M does not accept w.

Cantor's approach:
To show that two sets are of the same size we can have two approaches. Let us say we want to prove that the number of passengers standing at a bus stop is equal to the number of seats in the bus.

There are two ways to do this:
1. Count the passengers and the seats and see if they are equal.
2. Seat all the passengers in the bus and see if all passengers are seated and all the seats are occupied.

One to one and onto functions:
A function f : A ! B is called one to one if each element of A is assigned a unique element B (that is, f is a function) and f(a) ≠ f(b) whenever a ≠ b. A function f : A → B if for every b in B there is an a such that f(a) = b. A correspondence (or a one-to-one correspondence) is a both one to one and onto.

Galilean Paradox and Cantor"s approach:
Lets compare the set of positive integers with the set of squares. Let us look at

f(i)	f(1)	f(2)	f(3)	f(4)	f(5)	f(6)	……
				16	25	36	……

Lets look at another example:

N = { 1, 2, 3, 4, · · · , }
B = { 2, 3, 4, · · · , }

→ B

Then we have

g(i) = i – 1
f (i) = i + 1.

and f : A → B and g : B → A.

Hilbert Hotel:

Hilbert hotel.
1.
2.

integers. In other words, there exists an function f : N ! A such that

1. f is one to one.
2. and f is onto.

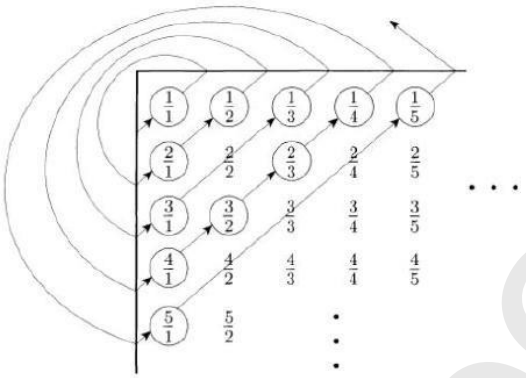
Is the set of all positive and negative integers countable? Yes.

Consider f given by:

$$f(i) = \begin{cases} i\ 2 & \text{if i is even} \\ i - 1/2 & \text{if i is odd} \end{cases}$$

Is the set of all ordered pairs of the set of positive integers countable? Yes. Consider f given by a picture.

Is the set of all positive rational numbers countable? Yes. Consider f given by a picture.



Uncountable sets and Cantor Diagonalization:

Theorem: R the set of all real numbers is uncountable.

Let x = 0.4641 · · ·
The cantor set consist of all infinite 0/1 sequences.

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
⋮	⋮

Elements of the cantor set:

- 101010101010
- 000000000000000
- 010100100100100100111

Theorem: Cantor set is not countable.

Let f be a function mapping N to C. Suppose f is onto. f maps elements as follows:

1 → 101011011 · · · ·
2 → 0010011011 · · ·
3 → 1111000101 · · ·
4 → 1010101001 · · ·
5 → 1100110011 · · ·
6 → 1010110101 · · ·
..
Let us define x a sequence $x = x_1x_2x_3 \cdots x_i$, where x_i is the i-th diagonal flipped.

1 → A_1
2 → A_2
3 → A_3
4 → A_4
5 → A_5
6 → A_6
...
x_i is different from the i-th bit of Ai by definition.

This x cannot be in the list. It will differ from each A_i. In fact, it will differ from A_i in the i-th bit.

Acceptance Problem for TMs:
Let us define

Lecture No.9

A_{TM} = {<M, w>: M is a TM that accepts w}.

This is called the acceptance problem for TMs or the halting problem.

Theorem: ATM is Turing Recognizable.

- On Input <M, w>
- Simulate M on w.
- If M accepts w accept.

This machine lets call it U is a recognizer for A_{TM}. Why is this not a decider? If M loops on w then U will also loop on <M, w>. U is called a universal Turing Machine.

Un-decidability of the Halting Problem:

Theorem: ATM is not decidable.

The proof is via contradiction. We will start by assuming that ATM is decidable and reach an absurdity. This method of proving mathematical claims is called reductio ad absurdum.

So let us assume that ATM is decidable. This means there is a decider H such that

$$H(\langle M, w\rangle) = \begin{cases} accept, & \text{if M accepts w} \\ reject, & \text{if M does not accept w} \end{cases}$$

H is a spectacular TM. Since, it halts on all inputs. So, it must be doing something very clever. Lets continue analyzing H. Note that if H exists then we can use it as a subroutine in any other TM that we design.

So, let us look at the following TM which we call D:

1. On input <M>.
2. Convert <M> into <M, <M>>
3. Run H on <M, <M>>.
4. Output the opposite of what H does; that is If H accepts reject, if H rejects accept.

Note that we can easily make this TM D provided H is given to us. So, if H exists then so does D. The main step D has to do is to convert the input <M> to <M, <M>>. What will running H on <M, <M> tell us? H accepts <M, <M>> if and only if M accepts <M>. So we are finding out if M accepts its own description or not!!

Can Turing machines accept/reject their own description? Can we run a program on itself? Well why not!

Here are some examples:

- Word count program can be used to find out how many lines of code does it have.
- A compiler for C written in C can compile itself.

So, let us find out what D is doing:

$$D(\langle M \rangle) = \begin{cases} accept, & \text{if M does not accepts w } \langle M \rangle \\ reject, & \text{if M accept } \langle M \rangle \end{cases}$$

Now, comes the interesting twist in the proof. We simply ask the question what does D do on its own description?

So, let us find out what D is doing:

$$D(\langle D \rangle) = \begin{cases} accept, & \text{if D does not accepts w } \langle D \rangle \\ reject, & \text{if D accept } \langle D \rangle \end{cases}$$

But this is absurd in both cases. Hence, we conclude that D does not exist and therefore H does not exist either.
Assume that H exists. We use H to construct D. This is easy D simply converts its input <M> into <M, <M>> and calls H. It returns the answer that is opposite of the one given by H.

1. H accepts <M, w> exactly when M accepts w.
2. D rejects <M> exactly when M accepts <M>.
3. D rejects <D> exactly when D accepts <D>.

This leads to a contradiction.

Where is the diagonalization in this proof? Well it is hidden but we can find it. Lets make a matrix as follows:

1. Label the rows with TMs as $M_1, M_2, M_3, \ldots, M_k, \ldots$
2. Label the columns with descriptions of TMs as $\langle M_1 \rangle, \ldots,$
3. Entry i, j is accept if M_i accepts $\langle M_j \rangle$

This matrix looks as follows:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	· · ·
M_1	accept	reject	accept	accept	· · ·
M_2	accept	accept	accept	accept	· · ·
M_3	accept	reject	loop	reject	· · ·
M_4	accept	accept	loop	reject	· · ·
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

1. The first three entries in the first row tell us that M_1 accepts $\langle M_1 \rangle$ and $\langle M_3 \rangle$ but not the $\langle M_2 \rangle$ and so on...
2. M_2 accepts all descriptions.....
3. Similarly the 4th row tells us that M4 accepts $\langle M_1 \rangle$ and $\langle M_2 \rangle$...

Now, since we are given that H exists we can make a matrix

1. Label the rows with TMs as $M_1, M_2, M_3, \ldots, M_k, \ldots$
2. Label the columns with descriptions of TMs as $\langle M_1 \rangle,$..... ,
3. Entry i, j is output of H on $\langle M_i , \langle M_j \rangle \rangle$

Lets look at this matrix. Note that all entries are filled.

Fill the entries according to the output of H:

	$<M_1>$	$<M_2>$	$<M_3>$	$<M_4>$	· · ·
M_1	accept	reject	accept	accept	· · ·
M_2	accept	accept	accept	accept	· · ·
M_3	accept	reject	reject	reject	· · ·
M_4	accept	accept	reject	reject	· · ·
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

The machine D is when it is given $<M_i>$ as an input looks at the entry i, i on the diagonal and outputs the opposite. Just like cantor‟s idea. Now, if D exists then there must be a row corresponding to D in this matrix. Lets look at this row and see why we have a contradiction.

Here we have filled the row for D.

$D = M_i$	accept	accept	accept	accept	· · ·

1. The first element is reject and the second is reject where as the third and fourth elements are reject.

The contradiction comes when we try to fill out the entry that corresponds to the column of D.

1. If we put an accept there, it should be reject.
2. Similarly, if we put a reject there, it should be accept.

	$<M_1>$	$<M_2>$	$<M_3>$	$<M_4>$	· · ·	· · ·	$<M_i>$	· · ·
$D = M_i$	accept	accept	reject	Reject	· · ·	· · ·	???	· · ·

1. Assume H exists.
2. Construct D. We argue that D can be constructed from H.
3. We check what D will do if it is given its own description.
4. This leads to an absurdity. Therefore, H cannot exist.
5. Quad erat demenstratum

:

Theorem: A language A is decidable if and only if A is decidable.

1. Proof is actually one line.
2. If we have a decider M that accepts A.
3. We can switch the accept and reject states to make another decider M‟.
4. M‟ accepts $\overline{A}$.

Lets recall that a language A is Turing recognizable if there is a TM M such that L(M) = A.

The TM M does not have to halt on all inputs. All we want is

$$M(x) = \begin{cases} accept, & x \in A \\ \text{does not accept}, & x \notin A \end{cases}$$

Note that the same trick we did in the previous theorem will not work here. If we switch the accept and reject states of M we get a new machine M" that does the following:

$$M'(x) = \begin{cases} \text{accept}, & x \in \overline{A} \\ \text{accept } loops, & x \notin \overline{A} \end{cases}$$

So it is sensible to talk about a Language whose complement is Turing recognizable.

Definition: A language B is called co-Turing-recognizable if B is Turing-recognizable.

Lets now prove another theorem.

Definition: If a language is Turing recognizable and co-Turing-recognizable then it is decidable.

Suppose A is Turing-recognizable and co-Turing-recognizable.

1. There is a TM M_1 such that $L(M_1) = A$.
2. There is a TM M_2 such that $L(M_2) = \overline{A}$.
3. if $x \in A$ then M_1 accepts x.
4. if $x \in \overline{A}$ then M_2 accepts x.

Lets consider the TM N now. On input x.

1. Run M_1 and M_2 on x in parallel.
2. If M_1 accepts x accept.
3. If M_2 accepts x reject.

We can use dovetailing to run both machines in parallel.

1. On input x.
2. Place the start configuration of M_1 on x on one tape.
3. Place the start configuration of M_2 on x on second tape.
4. For i = 1, 2, 3, . . . ,
5. Simulate one step of M_1. If M_1 accepts accept.
6. Simulate one step of M_2. If M_2 accepts reject.

Now what is L(N)? If $x \in A$ then N accepts x and if $x \notin A$ then N rejects x. Therefore, L(N) = A. Furthermore, N halts on all inputs. This is because $A \cup \overline{A} = \Sigma^*$

We have proved the following theorem:

Theorem: If a language is Turing recognizable and co-Turing-recognizable then it is decidable.

Lets apply this theorem to the Halting Problem. We cannot apply is directly.

What do we know about A_{TM}.

1. A_{TM} is Turing recognizable.
2. A_{TM} is not decidable.

So what can we conclude? We conclude that A_{TM} is not Turing recognizable.

Theorem: If A is not decidable the either A or $\overline{A}$ is not Turing-recognizable.

This theorem is just the contrapositive of the previous theorem.

Recall what a contrapositive is:

1. If we have a statement S : $P \Rightarrow Q$
2. Contrapositive of S is not $Q \Rightarrow$ not P

Now, suppose we have a statement: $P_1 \wedge P_2 \Rightarrow Q$
Then its contrapositive is: not $Q \Rightarrow$ not $(P_1 \wedge P_2)$.
However, Demorgan"s laws tell us that: not $(P_1 \wedge P_2) \Leftrightarrow$ not $P_1 \vee$ not P_2.
Hence, the contrapositive becomes: not $Q \Rightarrow$ not $P_1 \vee$ not P2.

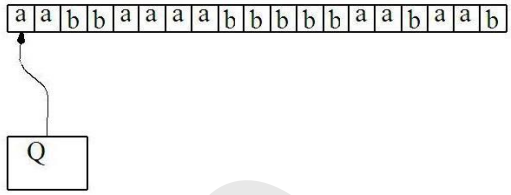
- We showed that A_{TM} is undecidable.
- We showed that if A is TR and co-TR then A is decidable.
- We concluded that $\overline{A_{TM}}$ is not TR.

LECTURE#10

Reducibility

Linear Bounded Automata:

An LBA L is a restricted type of TM. Its tape head is not permitted to move off the portion of the tape containing the input. Here is a diagram of an LBA:



As you can see the input portion of the tape is the only tape that the LBA is allowed to move its head on.

A_{LBA} = {<M, w>: M is a LBA that accepts w}
Is A_{LBA} decidable?

Acceptance Problem for LBAs
Let us define the acceptance problem for LBAs.

Theorem: Let M be an LBA with q states and g tape symbols. There are at most qng^n distinct configurations of M for a tape of length n. Recall that a configuration is completely given by

1. The state of the machine.
2. The position of its head.
3. The contents of the tape.

In this case we have
1. q possibilities for the state.
2. n possibilities for the position of the head.
3. g possibilities for each tape cell. Since, there are n of them hence there are g^n total possibilities.

Example: Suppose M has 7 states and 5 tape symbols. How many distinct configurations can we have for a tape of length 3.

1. The machine can be in any 7 states.
2. Its head can be scanning any tape cell. 3 possibilities.
   I. Cell 1 can have any symbol (5 possibilities).
   II. Cell 2 can have any symbol (5 possibilities).
   III. Cell 2 can have any symbol (5 possibilities).

So there are a total of $7 \times 3 \times 5^3$ distinct configurations.

Theorem: Let M be an LBA with q states and g tape symbols. If M takes more then qng^n steps on an input, x, of length n then the M does not halt on x.

Note that in that case one of the configurations of the LBA has repeated. Hence, M is stuck in an endless loop.

Theorem: A_{LBA} is decidable: We can simulate an LBA on its given input only for a finite amount of time to decide if it accepts the input or not.

Let L be a decider given as follows:

1. Input <M, w> where M is an LBA and w is a string.
2. Let n = |w|
3. Simulate M on w for qng^n steps.
4. If M accepts accept. Else reject.

Acceptance Problems Summary:

A_{DFA} = {<D, w>: D is a DFA that accepts w}.
A_{DFA} is decidable. Reason: we have to simulate D on w only for n steps.

A_{REX} = {<R, w>: R is a regular expression that generates w}.
A_{REX} is decidable. We convert the regular expression into a DFA and then use the decider for A_{REX} to decide A_{REX}.

A_{CFG} = {<G, w>: G is a context free grammar that generates w}.
A_{CFG} is decidable. Since we can convert G into Chomsky normal form and list all derivations of length 2n − 1.

A_{PDA} = {<P, w>: P is PDA that accepts w}.
This is decidable. Since, we can convert a PDA into a equivalent grammar and then use the decider for A_{PDA}

Emptiness Problem for LBA:

Let us define the emptiness problem for LBAs.
A_{REX} = {<M>: M is a LBA and L(M) = }
Is A_{REX} undecidable?

We will show once again that:
If A_{REX} is decidable then A_{REX} is decidable.

What would we like to do?
Given an input <M, w> where M is a TM and w is a string. We want to construct a LBA B such: B is going to accept only accepting computation history of M on w. Now, if M accepts w then there is one computation history that B will accept. On the other hand if M does not accept w then L(B) =Φ.

Computation Histories:
Let M be a TM. We have the following concepts.

1. A state. This just tells us what the current state of the machine is.
2. A configuration. This tells us what the state is and all the tape contents. So, given a configuration we can figure out what will be the next configuration.

A computation history is a sequence of configurations. A computation history on an input w will consist of configurations $C_1, C_2, \ldots, C_l$ such that

1. C_1 is the initial configuration of M on w.
2. C_i yields C_{i+1}.

A computation history of M on w is accepting if the last configuration in the history is an accepting configuration. Note that a accepting computation history of M on w is a proof that M accepts w. We will encode a computation history as follows: $C_1\#C_2\#\cdots\#C_l$.

Given a TM M and w we can construct an LBA B such that, B accepts the accepting computation history of M on w. We give an informal description of B.

On input $C_1\#\ldots\#C_l$.

1. Check if $C_1 = q0w$.
2. Check if C_l has an accept state.
3. Check C_i yields C_{i+1}.
    a. The transition function of M is hard coded in B.
    b. B zigzags between C_i and C_{i+1} to make sure if C_i yields C_{i+1}

Note that B never uses any extra memory. It only uses the portion of the input that is given to B.

B is an LBA.

1. Note that the input to B is not w but a computation history H of M.
2. In order to check if H is an computation history history of M on w it does not need more tape then the portion on which H is written.

Theorem: Let M and w be given and B be constructed as earlier.

$$L(\mathrm{B}) = \begin{cases} = \varnothing, M \text{ does not accept } w \\ \neq \varnothing, M \text{ assept } w \end{cases}$$

Theorem: E_{LBA} is undecidable: Suppose R decides E_{LBA} then consider S

1. On input <M, w>.
2. Construct and LBA B as described earlier.

Run R on B. If R accepts reject. If R rejects accepts. Note that S is a decider for A_{TM}, a contradiction.

Context Free Grammars:

Let us define the emptiness problem for CFGs.

E_{CFG} = {<G>: G is a CFG L(G) = Ø}. Is E_{CFG} decidable?

Emptiness Problem for CFGs

1. On input <G>.
2. Mark all terminals of G.
3. While no new symbols are marked.
4. If $A \longrightarrow X_1 \ldots X_r$ is a production with all symbols marked on the right hand side. Mark A.
5. If the start symbol is marked then reject else accept.

Let us define the another problem for CFGs.

ALL_{CFG} = {<G>: G is a CFG L(G) = $\sum^*$}. Is ALLCFG decidable?

On the surface the problem looks difficult. We have to find at least one string that is not generated by the grammar G. However there are infinitely many potential strings. But our goal is to PROVE that it is undecidable.

Theorem: ALL_{CFG} is undecidable.

Computation Histories: Once again we will show that If ALL_{CFG} is decidable then A_{TM} is decidable. Given a TM M and w we can construct an PDA D such that D rejects the accepting configuration history of M on w (appropriately encoded) and accepts all other strings. We give an informal description of D.

We will use a trick here. Let $C_1, \ldots, C_l$ be a computation history. We will encode it as follows: $C_1 \#C_2^r \#C_3 \#C_4^r \#\ldots\ldots\#C_l$. We will see in a minute why we are doing this.

Now consider the following PDA D. The PDA will take a computation history $H = C_1, \ldots, C_n$ and do the following.

1. If C1 is not the initial configuration of M on w accept.
2. If Cl is not a final configuration then accept.
3. If there is an i such that C_i does not yield C_{i+1} then accepts.

Note that D is a non-deterministic PDA and therefore it can guess which one of these conditions is violated. Since w is hard coded in D it can check if the first configuration is not the initial configuration. Similarly, it is easy to check if the last configuration is not an accepting configuration. How does D check if C_i does not yield C_{i+1}?

This is why we said we will use a special encoding. We had $C \#C^r_{i+1}$ Thus D can push C_i on its stack and pop it and check if C_i yields C_{i+1} or not.

Theorem: Given M and w we can construct a PDA D such that D rejects the accepting computation of M on w (properly encoded). It accepts all other strings.

Theorem: Let M and w be given and D be constructed as earlier. Let G be a grammar which generates all the strings accepted by G. Then

$$L(\mathrm{B}) = \begin{cases} = \sum^*, M \text{ does not accept } w \\ \neq \sum^*, M \text{ accepts} \end{cases}$$

Theorem: ALL_{CFG} is undecidable.

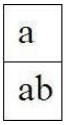
If ALL_{CFG} is decidable then there is a decider R such that $L(R) = ALL_{CFG}$
Suppose R decides ALL_{CFG} then consider S

1. On input <M, w>
2. Construct and LBA D as described earlier.
3. Convert D into an equivalent CFG G.
4. Run R on G. If R accepts reject. If R rejects accepts.

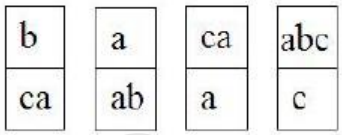
Note that S is a decider for A TM. A contradiction.

Post Correspondence Problem:

Post Correspondence Problem puzzle. We are given a collection of dominos each containing two strings, on on each side. A domino looks like this:

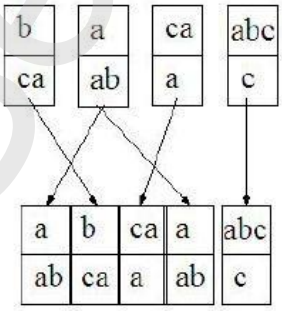


This domino has a on top and ab on the bottom. A collection of dominos looks like this.



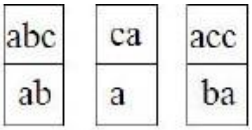
The first domino has b on top and ca on the bottom.
The second domino has a on top and ab on the bottom.
The third domino has ca on top and a on the bottom.
The fourth domino has abc on top and c on the bottom.

We make a list of dominos by placing the next to each other. We are allowed repetitions. This list is called a match if reading of the top symbols gives us the same string as reading of the bottom symbols.



For example, here we have a match.
Reading of the top symbols gives us

Lets look at another collection. This collection has three dominos.



1. The first domino has abc on top and ab on the bottom.
2. The first domino has ca on top and a on the bottom.
3. The first domino has acc on top and ba on the bottom.
4. For this collection it is not possible to find a match. Why?

The top of each domino has more letter than the bottom. So in any list the top string will be longer than the bottom string. Given a collection P of Dominos. Does P have a match?

PCP = {<P>: P has a match}.

Theorem: PCP is undecidable.

Acceptance Problem for TMs: Lecture No 11

Let us define
A_{TM} = {<M, w>: M is a TM that accepts w}.
This is called the acceptance problem for TMs or the halting problem.
We also call this the Halting Problem. We showed that this problem is undecidable.

Undecidability of the Halting Problem:

Theorem: A_{TM} is not decidable.

The language A_{TM} is undecidable. There is no TM H such that on input <M, w>

1. H accepts if M accepts w.
2. H rejects if M does not accept w.

If H exists we can use H to construct D as follows:

1. On input <M>.
2. Convert <M> into <M, <M>>
3. Run H on <M, <M>>.
4. Output the opposite of what H does; that is If H accepts reject, if H rejects accept.

Once we have constructed D we will ask how it behaves on its own description.

Assume that H exists. We use H to construct D. This is easy D simply converts its input <M> into <M, <M>> and calls H. It returns the answer that is opposite of the one given by H.

1. H accepts <M, w> exactly when M accepts w.
2. D rejects <M> exactly when M accepts <M>.
3. D rejects <D> exactly when D accepts <D>.

This leads to a contradiction.

Russell's Paradox:
In a town called Seville there lives a barber. "A man Seville is shaved by the barber if and only if the man does not shave himself?"

Question: Does the barber of Seville shaves himself.

Students are asked to make a list in which they can include the names of anyone (including themselves) in a class room. Is there a particular student wrote the names of those people in the class who did not include their own names in their own list?

Question: Does that particular student write his/her own name on his/her list?

Is it possible to have an enumerator that enumerates the description of all the enumerators that do not enumerate their own description. Suppose such an enumerator F exists.

Question: Does F enumerate its own description?

1. D that accepts (the description of) those TMs that do not accept their own description.
2. Barber who shaves all those people who do not shave themselves.
3. Student who lists all those students who do not list themselves.
4. Enumerator that enumerates (the description of) all those enumerators that do not enumerate themselves.

Such objects cannot exist.

Reducibility:

Suppose u have a language B in your mind and you can prove that. If I can make a decider for B then I can make a decider for the halting problem.

Now suppose some one claims that they have a decider for the language B. Then I can claim that I have a decider for A_{TM}. Which is absurd. From this we can conclude that B is also undecidable as we know that A_{TM} is undecidable.

$HALT_{TM}$ = {<M, w> : M is a TM that halts on w}.

Theorem: $HALT_{TM}$ is undecidable.

Suppose that $HALT_{TM}$ is decidable and R decides $HALT_{TM}$. If we have R then we have no problem on TM that loop endlessly. We can always check if a TM will loop endlessly on a given input using R. We can use R to make a decider for A_{TM}. We know that A_{TM} is undecidable. We have a contradiction and that is why R cannot exist.

Given R we can construct S which decides A_{TM} that operates as follows:

1. On input <M, w>
2. Run TM R on input <M, w>.
3. If R rejects, reject.
4. If R accepts, simulate M on w until it halts.
5. If M has accepted w, accept; if M has rejected w, reject.

Note that S will reject <M, w> even if M loops on w.

Emptiness problem for TMs:

E_{TM} = {<M>: L(M) = Ø}

Theorem: E_{TM} is undecidable.

Suppose that E_{TM} is decidable and R decides E_{TM}. R accepts <M> if the language of M is empty. We can use R to make a decider S for A_{TM}. We know that A_{TM} is undecidable. We have a contradiction and that is why R cannot exist. How can we construct S?

S gets <M, w> as an input. The first idea is to run R on input <M> and check if it accepts. If it does, we know that L(M) is empty and therefore that M does not accept w. But, if R rejects we only know that the language of R is not empty. We do not know anything about what it does on w. So the idea does not work. We have to come up with another idea.

Given M and w let us consider M_1:

1. On input x
2. If x ≠ w, reject.
3. If x = w, run M on input w and accept if M accepts w.

What is $L(M_1)$? $L(M_1) = \begin{cases} \varnothing, \text{ If } M \text{ does not accept } w \\ \{w\}, \text{ If } M \text{ accept } w \end{cases}$

Given R we can now we can construct S which decides A_{TM} that operates as follows:

On input <M, w>
1. Run TM R on input <M, w>.
2. Construct M_1
3. Run R on <M_1>
4. If R accepts <M_1>, reject. If it rejects <M_1> accept.

S is a decider for A_{TM}. Since S does not exist we conclude that R does not exist. So E_{TM} is undecidable.

Regular languages and TMs:

REG_{TM} = <<M>: L(M) is regular}

Theorem: REG_{TM} is undecidable

Suppose that REG_{TM} is decidable and R decides REG_{TM}. R accepts all TMs that accept a regular language and reject all TMs that do not accept a regular language. How can we use R to construct a decider for A_{TM}? Note that S has to get as input <M, w> and has to decide if M accepts w. The idea is as follows:

S gets <M, w> as an input. Given M and w we will construct a machine M_2 such that

$$L(M_2) = \begin{cases} \text{is regular, if M accpets w} \\ \text{not regular, if M does not accept w} \end{cases}$$

Given M and w let us consider M_2:
1. On input x
2. If $x = 0^n1^n$ then accept.
3. else run M on input w and accept x if M accepts w.

What is $L(M_2)$?

$$L(M_2) = \begin{cases} \{0^n1^n : n \geq 0\}, \text{if M does not accpets w} \\ \{0,1\}^* \quad \text{, if M accept w} \end{cases}$$

Given R we can now we can construct S which decides ATM that operates as follows:

On input <M, w>
1. Construct M_2
2. Run R on <M_2>
3. If R accepts <M_2>, accept. If it rejects <M_2> reject.

S is a decider for A_{TM}. Since S does not exist we conclude that R does not exist. So REG_{TM} is undecidable.

Equivalence problem for TMs:
So far we have only used ATM to show that other problems are undecidable. Let us do something different. Let us define:

$EQ_{TM} = \{<M_1, M_2>: L(M_1) = L(M_2)\}$

Theorem: EQ_{TM} is undecidable.
Suppose that EQ_{TM} is decidable and R decides EQ_{TM}. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages is forced to be the empty set, we end up with the problem of determining whether the language of the other machine is empty.

E_{TM} is a special case of EQ_{TM}.

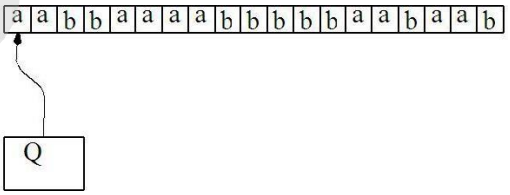
Suppose R decides EQ_{TM}. We design S to decide E_{TM}. S gets <M> as an input. Let M" be a machine that rejects all inputs.

On input <M>
1. Run R on <M, M">
2. If R accepts, accept. If it rejects, reject.

It is easy to see that the language of S is E_{TM}. As, E_{TM} is undecidable so is EQ_{TM}.

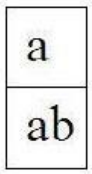
Linear Bounded Automata:

An LBA L is a restricted type of TM. Its tape head is not permitted to move off the portion of the tape containing the input. Here is a diagram of an LBA:

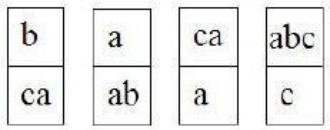


As you can see the input portion of the tape is the only tape that the LBA is allowed to move its head on.

Post Correspondence Problem:
Post Correspondence Problem (PCP) is a puzzle. We are given a collection of dominos each containing two strings, on on each side. A domino looks like this:



We make a list of dominos by placing the next to each other. We are allowed to repeat dominos. From the collection



We can make the following lists:
This list is called a match if reading of the top symbols gives us the same string as reading of the bottom symbols.

In the second list we have a match since the top reads to, abcaaabc and same is the case if we read the bottom.

b	ca	a	ca
ca	a	ab	a

a	b	ca	a	abc
ab	ca	a	ab	c

Lets look at another collection (now, written in a less fancy way). This collection has three dominos.

$$\frac{abc}{ab}\,\frac{ca}{a}\,\frac{acc}{ba}$$

The first domino has abc on top and ab on the bottom.

1. The first domino has ca on top and a on the bottom.
2. The first domino has acc on top and ba on the bottom.

For this collection it is not possible to find a match. Why? The top of each domino has more letter than the bottom. So in any list the top string will be longer than the bottom string.

Given a collection P of Dominos. Does P have a match?

PCP = {<P>: P has a match}.

Note that the Post Correspondence Problem is very different from the ones that we have studied. It apparently seems to have nothing to do with Turing machines or any other model of computation. It is just a puzzle.

Theorem: PCP is undecidable.

How will we prove this theorem? We will show that if PCP is decidable then we can use the decider for PCP to construct a decider for A TM. The main idea is that given a TM M and an input w we will describe a collection P of dominos such that, M accepts w if and only if P has a match.

How to do this? First lets us deal with a little technicality. Lets define another problem which is closely related to PCP. This problem is called MPCP (M for modified). Given a collection P of dominos does P have a match that starts with the first domino? We will show that MPCP is undecidable.

Once again the main idea is that given a TM M and an input w we will describe a collection P0 of dominos such that, M accepts w if and only if P" has a match starting with the first domino.

The main idea is that we will make a collection so that the only way to find a match will be to "simulate" a computation of M on w.

In fact a match will correspond to an accepting computation history of M on w. The construction is going to be in 7 parts.

Let M be a Turing machine where M = $(Q, \sum, \Gamma, \delta, q_0, q_a, q_r)$, and $w = w_1 \cdot \cdot \cdot w_n$

Part 1:

Put $\left[\begin{array}{c}\#\\ \hline \#\ q\ w\ w\ .....w\ \#\\ 0\ 1\ 2\qquad\quad n\end{array}\right]$ in p, as the first domino

Since this is the first domino hence the bottom string is the initial configuration of M on input w. To get a match one must choose dominos that are consistent with the bottom. We will chose these dominos in such a way that it forces the second configuration of M to appear at the bottom and so on.

Part 2:

For every a, b Ɛ Γ and q, r Ɛ Q where q ≠ qr if δ (q, a) = (r , b,R), put $\left[\begin{array}{c}qa\\ \hline br\end{array}\right]$ in p,

Part 3:

For every a, b, c Ɛ Γ and q, r Ɛ Q where q ≠ qr if δ (q, a) = (r , b, L), put $\left[\begin{array}{c}cqa\\ \hline rcb\end{array}\right]$ in p,

Part 4:

For every a Ɛ Γ put $\left[\begin{array}{c}a\\ \hline a\end{array}\right]$ in p,

The purpose of these dominos is to “simulate” the computation of M. Let us look at an example.

Example:

Lets say Γ = {0, 1, 2,□}. Say w = 0100 and the start state of M is q0. Then the initial configuration is q00100. Now, suppose $\delta(q_0, 0) = (q_7,2,R)$. Part 1 places the first domino which is

$$\left[\begin{array}{c}\#\\ \hline \#\ q_0\ 0100\ \#\end{array}\right]$$

So we have the initial configuration on the bottom. This is how the match begins. In order for the match to continue we want to pick a domino that has q00 on the top. Part 2 had put

$$\left[\begin{array}{c}q_0\\ \hline 2q_7\end{array}\right]$$

in the collection. In fact, that is the only domino that fits. So if we are to continue our match we are forced to put this domino next. The match now looks like Part 1 places the first domino which is

$$\left[\begin{array}{c}\#\ q_0\\ \hline \#\ q\ 0100\ \#\ 2q\end{array}\right]$$

Also remember we have the dominos

$$\left[\begin{array}{c}0\end{array}\right]\left[\begin{array}{c}1\\ \hline 1\end{array}\right]\left[\begin{array}{c}2\\ \hline 2\end{array}\right]\left[\begin{array}{c}\square\\ \hline \square\end{array}\right]$$

Using these we can extend this match to

$$\left[\begin{array}{c} \#q_0 0100 \\ \hline \#q_0 0100\#2q_2 100 \end{array}\right]$$

This is quite amazing! The top string is the initial configuration. The bottom has the initial configuration and the one that comes after the initial configuration. We want this to continue.

Part 5:

Put $\left[\begin{array}{c} \# \\ \hline \# \end{array}\right]$ and $\left[\begin{array}{c} \# \\ \hline \# \end{array}\right]$

1. The first one allows us to copy the # symbol that marks the separation of the configurations.
2. The second one allows us to add a blank symbol at the end of a configuration to simulate the blanks to the right that are suppressed when we write a configuration.

Lets continue with our example. Let us say we have $\delta(q_7, 1) = (q_5, 0, R)$.

Then we have the domino (from part 2) $\left[\begin{array}{c} q_7 1 \\ \hline 0 q_5 \end{array}\right]$

So we can get to the partial match $\left[\begin{array}{c} \#q_0 0100\#2q_7 100\# \\ \hline \#q_0 0100\#2q_2 100\#20q_5 00\# \end{array}\right]$

Once again note that the top has the first two configurations of M and the bottom has the first three. Lets look at what happens when the machine moves to the left.

Say we have $\delta(q_5, 0) = (q_9, 2, L)$ then we put the dominos

$$\left[\begin{array}{c} 0q_5 0 \\ \hline q_9 02 \end{array}\right]\left[\begin{array}{c} 1q_5 0 \\ \hline q_9 12 \end{array}\right]\left[\begin{array}{c} 2q_5 0 \\ \hline q_9 22 \end{array}\right]\left[\begin{array}{c} \square q_5 0 \\ \hline q_9 \square 2 \end{array}\right]$$

Now to match the bottom we must use $\left[\begin{array}{c} 0q_5 0 \\ \hline q_9 02 \end{array}\right]$

and copy dominos to get to $\left[\begin{array}{c} \#20q_5 00\# \\ \hline \#20q_5 00\#2q_9 020\# \end{array}\right]$

In order to construct a match we must simulate M on input w. However, there is a problem. The top string is always “behind” the bottom string! This is actually good for us. We will only allow the top string to catchup with the bottom one if we go into an accept state.

Part 6:
For every a Ɛ Γ put $\left[\begin{array}{c} aq_a \\ \hline q_a \end{array}\right]$ and $\left[\begin{array}{c} q_a a \\ \hline q_a \end{array}\right]$ in p'

These dominos allow us to eat up the symbols till none are left. Lets look at an example. Suppose we have the following situation.

$$\left[\begin{array}{c} ....\# \\ \hline \#21q\ 02\# \\ a \end{array}\right]$$

We can use $\left[\begin{array}{c} q_a 0 \\ \hline q_a \end{array}\right]$ to get to $\left[\begin{array}{c} ....\# \\ \hline \#21q\ 2\# \\ a \end{array}\right]$ and other dominos in part 6 to finally reach $\left[\begin{array}{c} ....\# \\ \hline \#q\# \\ a \end{array}\right]$

Part 7:
Finally we add $\left[\begin{array}{c} q_a \#\# \\ \hline \# \end{array}\right]$ in P", Which will allow us to complete the match.

Summary of this construction:

1. The first domino has the initial configuration on the bottom and only a # on the top.
2. To copy any configuration on the top. We are forced to put the next configuration on the bottom.
3. By doing this we simulate M on w.
4. When we reach the final state. We allow the top to catch up with the bottom string.

Another thing to note:

1. If M rejects w. There is no match. A match can only occur if we reach an accept state.
2. If M loops forever on w then the top string will never catch up with the bottom string.

Theorem: If P" is constructed as described then it has a match if and only if M accepts w.

Theorem: MPCP is undecidable.

Assume that MPCP is decidable. Let us say we have a decider R for MPCP. Consider the following decider S

1. On input < M, w >
2. Construct p' as described in the seven parts.
3. Run R on P".
4. If R accepts, accept.
5. If R rejects, reject.

Then S is a decider for A_{TM}, which is a contradiction to the fact that A_{TM} is undecidable.

Note that we wanted to prove that

Theorem: PCP is undecidable.

However, P" is an instance of MPCP. In fact, if we remove the restriction that the match should start with the first domino it has a very short match. Now, we can convert this instance of MPCP to an instance of PCP as follows:
Let $u = u_1 u_2 \cdots u_n$ be a string. Let us define *u, u*, *u* to be

$$*u = *u_1 * u_2 * u_3 \cdots * u_n$$
$$u* = u_1 * u_2 * u_3 \cdots * u_n *$$
$$*u* = *u_1 * u_2 * u_3 \cdots * u_n *$$

Let us say we have an instance P" of MPCP. The collection P" is given by

$$\left\{ \begin{bmatrix} t_1 \\ b_1 \end{bmatrix} \begin{bmatrix} t_2 \\ b_2 \end{bmatrix} \begin{bmatrix} t_3 \\ b_3 \end{bmatrix}, \ldots\ldots\ldots, \begin{bmatrix} t_k \\ b_k \end{bmatrix} \right\}$$

Let us define an instance P of PCP with the collection

$$\left\{ \begin{bmatrix} *t_1 \\ *b_1 \end{bmatrix} \begin{bmatrix} *t_2* \\ b_2 \end{bmatrix} \begin{bmatrix} *t_3* \\ b_3 \end{bmatrix}, \ldots\ldots\ldots, \begin{bmatrix} *t_k* \\ b_k \end{bmatrix} \begin{bmatrix} *\Delta* \\ \Delta * \end{bmatrix} \right\}$$

In this collection any match will start with the first domino. Since, it is the only one that has the same first character on the top and bottom.

So, we don"t have to insist that the match has to start with the first domino. It is built into the collection.

Other than forcing the first domino to start the match the * "s do not put any restrictions. They simply interleave with the symbols.

$\begin{bmatrix} *\Delta \\ \Delta * \end{bmatrix}$ allows the top string to add an extra * at the end of the match! Note can easily prove.

Theorem: PCP is undecidable

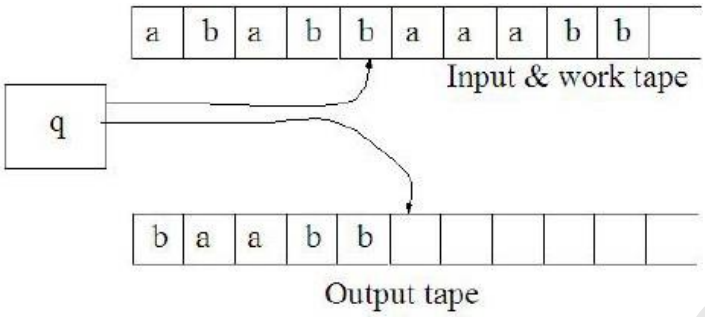
Assume that PCP is decidable. Let us say we have a decider R for PCP. Consider the following decider S

1. On input $< P_0 >$
2. Construct P as described above.
3. Run R on P.
4. If R accepts, accept.
5. If R rejects, reject.

Then S is a decider for MPCP. Which is a contradiction to the fact that MPCP is undecidable.

Computable Functions:

Now we will consider TMs that compute functions. So the TM has an input tape and an output tape. A diagram of the machine looks like as follows:



Let us consider a function f : $\sum^* \rightarrow \sum^*$.

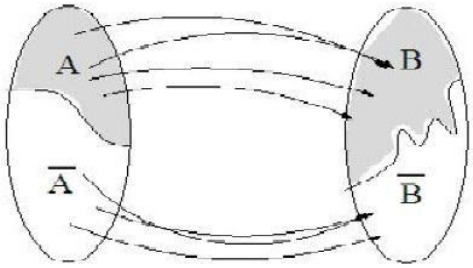
We say that f is computable if there is a TM M such that on every input w it halts by writing f(w) on its output tape. Note that some books do not require the machine to have a separate output tape. Intuitively, a computable function f is a function which can be computed by an algorithm.

Examples of computable functions:

1. f(<m, n>) = <m + n>
2. Suppose a string w encodes a description of a two tape TM. Let f be a function that constructs the description of an equivalent one tape TM. In this case, f is a computable function.

Let A be a language. We say A is mapping reducible to B, written as $A \leq_m B$, if there is a computable function f: $\sum^* \rightarrow \sum^*$ such that for every w, $w \in A \leftrightarrow f(w) \in B$.

f maps strings in A to strings in B. Further more, strings which are not in A are not mapped to B. Here is a diagram:



Note that f must be a computable function. It does not need to be one-to-one or onto.

Examples of computable functions:

1. f (<m, n>) = <m + n>
2. Suppose a string w encodes a description of a two tape TM. Let f be a function that constructs the description of an equivalent one tape TM. In this case, f is a computable function.
3. Let A be an n × n matrix then the function f (<A>) = < A^2 > is computable.

Any function f which can be computed by an algorithm is called a computable function. Computable functions are more important than Languages in practice.

Usually we want to take n input w and produce and output f (w). An algorithm (or a TM M) that computes f is what we look for.

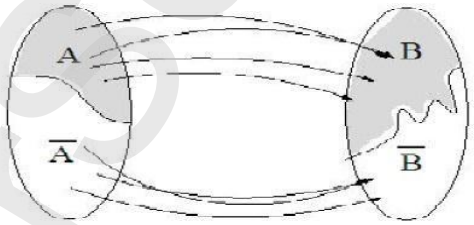
Suppose we have two problems A and B. Roughly, if the solution of B can be used to also solve A then we say A is reducible to B.

Some examples of reducibility:

- The problem of finding how to reach from a place A to B in Lahore can be reduced to the task of finding a map for the city of Lahore.
- The problem of crossing a river can be reduced to finding a boat.
- The problem of quenching ones thirst can be reduced to the problem of finding clean water.
- The problem of solving a quadratic equation can be reduced to the problem making friends with a mathematician.

Reducibility:

Let A be a language. We say A is mapping reducible to B, written as $A \leq_m B$, if there is a computable function $f : \sum^* \rightarrow \sum^*$ such that for every w, $w \in A \leftrightarrow f(w) \in B$



In this diagram:

- The left side shows $\sum^*$ and the shaded portion shows the set A.
- The right side again $\sum^*$ and the shaded portion shows the set B.
- The function must map all the strings inside A into B.
- It must also map all the strings inside $\overline{A}$ into $\overline{B}$.
- The function must be computable.

Let us make the observation that f

1. Need not be one-to-one. So, it is allowed to map several inputs to a single one.
2. Need not be onto.

Let us look at some simple reducibility, Let

A = {<n>: n is even}
B = {w∈{0, 1}* : w contains no 1"s }. Let us define f ($x_1x_2 \cdot \cdot \cdot x_n$) = x_n then f is computable.

f shows that A ≤m B since $f(\langle n \rangle) = \begin{cases} 0, \text{ if } n \text{ is even} \\ 1, \text{ if } n \text{ is odd} \end{cases}$

This shows that f(<n>)$\in$B if and only if <n>$\in$A.
Suppose

- B can be solved.
- A can be reduced to B.

We can conclude that we can also solve A.

Theorem: If A $\leq_m$ B and B is decidable, then A is decidable.

Theorem: If A $\leq_m$ B and B is decidable, then A is decidable.

Proof: Since B is decidable hence there is a decider M that decides B.

Now, consider the following decider N:

1. On input x
2. Compute y = f(x)
3. Run M on y
4. If M accepts, accept.
5. If M rejects, reject.

Since f is a computable function and M is a decider therefore N is a decider. Furthermore, x is accepted by N if and only if y is accepted by M. However, y is accepted by M if and only if y$\in$B.

Since f is a reduction therefore, x$\in$A if and only if y = f(x)$\in$B. Which implies that N accepts x if and only if x$\in$A. This shows that N is a decider for A.

Theorem: If A $\leq_m$ B and B is decidable, then A is decidable.

We can also write this theorem in the contrapositive form:

Theorem: If A $\leq_m$ B and A is undecidable, then B is undecidable.

We are more interested in using the contrapositive form of this theorem. Since, we are more interested in undecidable questions.

Suppose you have a language P that you do not know anything about. You are wondering if P is decidable or undecidable. There are two approaches you have now.

1. Find language D which is known to be decidable and show that P$\leq_m$D now you can conclude that P is also decidable.

2. Find an undecidable language U (which is known to be undecidable language) and show U $\leq_m$ P now you can conclude that P is undecidable.

Put given a language P how can you tell if it is decidable or not? How do you know which option to take? In general no one knows the answer.

We first try to find an algorithm and try to prove that P is decidable. If we fail and start suspecting that the problem is undecidable then we try to reduce some known undecidable problem to P.

In case of Hilbert"s third problem. It took 70 years to come up with the reducibility and prove that the tenth problem is undecidable.

Note to prove decidability of P We reduce P to a known decidable problem D. We show that $P \leq_m D$. Note that this is only one approach. We can also give a algorithm for P and prove that it is decidable.

To prove that P is undecidable. We reduce U to P where U is a previously known undecidable problem. We show that U ≤m P. So we are doing some thing backwards in this case!

Mapping reducibility is only one kind of reducibility. It is in some sense the simplest way to reduce a problem to another one. We will only discuss mapping reducibility.

We have already used much reducibility to show that problems are undecidable. Let us look at some of them. We have the fundamental problem.

A_{TM} = {<M, w>: M accepts w} which is undecidable. Let us consider
$HALT_{TM}$ = {<M, w>: M halts on w}

To show that $A_{TM} \leq_m HALT_{TM}$, we have to find a computable function f such that
f (<M, w>) = <M", w"> such that <M, w>$\in A_{TM}$ if and only if <M", w">$\in HALT_{TM}$.

We show a machine F that computes f as follows:

1. On input <M, w>
2. Construct M" as follows:
3. "On input x
4. Run M on x
5. If M accepts x, accept.
6. If M rejects x, loop forever."
7. Output <M", w>

Let us show $E_{TM} \leq_m EQ_{TM}$. Let M_1 be a TM that rejects all its inputs. Let us define
f(<M>) = <M, M_1> then we have L(M) = Φ if and only if L(M) = L(M_1).

In order to show that f is a reducibility we only need to argue that it is computable. It is clearly computable since all we have to do is to append the description of M1 to the description of M.

Let us show $A_{TM} \leq_m \overline{E}_{TM}$.
We define a computable function f(<M, w>) = <M_1> such that:
M accepts w if and only if L(M_1) ≠ Φ.

Here <M1> is computed by F as follows:

1. On input <M, w>
2. Let M_1 the be machine:
3. "On input x
4. If x ≠ w reject.
5. else run M on x.
6. If M accepts x accept."
7. Output M_1

Now the previous reducibility shows that $\overline{E}_{TM}$ is undecidable. From this we can also conclude that E_{TM} is undecidable. Since A is decidable if and only if $\overline{A}$ is decidable.
If $A \leq_m B$ and B is Turing recognizable then A is Turing recognizable.

Proof is almost identical to the earlier theorem. Let $A \leq_m B$ and let f be the reducibility from A to B. Furthermore, since B is Turing recognizable there is a TM M such that L(M)=B.

Consider N:
1. On input x
2. Compute y = f (x)
3. Run M on y
4. If M accepts, accept.

Then it is easy to see that L(N) = A.

We can write this theorem in the contrapositive form as:
If $A \leq_m B$ and A is not Turing recognizable then, B is not Turing recognizable.

Let us use this theorem to prove something very interesting.

Recall $EQ_{TM} = \{<M_1, M_2>: L(M_1) = L(M_2)\}$.
Let show that: $A_{TM} \leq_m EQ_{TM}$.

Given <M, w> consider two machines:

M_1:
1. On input x
2. accept

M_2:
1. On input x
2. run M on w, if M accepts w accept x.

Note that $L(M_1)=\sum^*$, and $L(M_2)=\sum^*$ if and only if M accepts w. Therefore, $L(M_1)=L(M_2)$ if and only if M accepts w. Hence if we define f (<M, w>) = $<M_1, M_2>$ then f is a reducibility from A_{TM} to EQ_{TM}.

Given <M, w> consider two machines:

M_1:
3. On input x
4. reject

M_2:
3. On input x
4. run M on w, if M accepts w accept.

Note that $L(M_1) = \Phi$ and $L(M_2) = \Phi$ if and only if M does not accepts w.
Therefore, $L(M_1)=L(M_2)$ if and only if M does accepts w. Hence if we define
f (<M, w>) = $<M_1,M_2>$ then f is a reducibility from $\overline{A}_{TM}$ to EQ_{TM}.
f can be computed as follows:
1. On input <M, w>
2. Construct M_1 and M_2 as described above.
3. Output $<M_1, M_2>$

We have shown $A_{TM} \leq_m EQ_{TM}$.
This shows that $\overline{A}_{TM} \leq_m \overline{EQ}_{TM}$ and hence $\overline{EQ}_{TM}$ is not Turing recognizable.

$A_{TM} \leq_m EQ_{TM}$

This shows that $\overline{A}_{TM} \leq_m EQ_{TM}$ and hence EQ_{TM} is not Turing recognizable.

Hence, EQ_{TM} is a language which is neither Turing-recognizable nor co-Turing recognizable.
Now, let us look at a very interesting theorem which shows us why the halting problem is o significant.

Theorem:
Every Turing recognizable language is mapping reducible to A_{TM}. This theorem says that A_{TM} is in some sense the hardest problem out of all Turing recognizable problems. If one could devise an algorithm for A_{TM} one could devise an algorithm for all Turing recognizable problems. Alas, A_{TM} is not decidable.

The proof of this theorem is very easy. let A be any Turing recognizable language. We have to show that $A \leq_m A_{TM}$ Let M be a TM such that L(M)=A. Note that M exists since A is Turing-recognizable.

Now, let us consider the function f given by f(x) = <M, x>, clearly $x \in A$ if and only if M accepts x .
Which is the same as $x \in A$ if and only if $\langle M, x\rangle \in A_{TM}$.

Is f computable?
Yes, if we hardwire M into the program that computes f. Here is the machine F that computes f

1. On input x
2. Output <M, x>

This shows that A_{TM} in some sense is the hardest Turing recognizable language. Therefore, it is the best candidate for being an undecidable language. No wonder Turing chose to work with this language and show that it is undecidable!

Recall A_{TM} = {<M, w>: M accepts w }, and
EQ_{TM} = {$\langle M_1, M_2\rangle$: M_1 and M_2 accept the same language }. We showed that

Theorem: $A_{TM} \leq_m EQ_{TM}$.

We did this as follows:
Given<M, w> we constructed two TMs:

1. The first machine, M_1, accepted everything.
2. The second machine, M_2, runs M on w. If M accepts w it accepts it‟s own input.

The reducibility was given by: f (<M>) = $\langle M_1, M_2\rangle$.

This also shows that shows that

Theorem: $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$
Since $\overline{A_{TM}}$ is not Turing recognizable therefore $\overline{EQ_{TM}}$ is also not Turing recognizable.

How do we know that A_{TM} is not Turing recognizable?

We will show that EQ_{TM} is also not Turing recognizable. We will now show that $A_{TM} \leq m \overline{EQ_{TM}}$

We are given <M, w> and construct the following machines:

1. The first machine, M_1, is going to be a TM that will reject all inputs.
2. M2, the second machine, ignores it"s input and runs M on w. If M accepts w then M accepts it"s own input.

What can we say about the languages of M_1 and M_2. The language of the first machine is the empty language; that is, $L(M_1) = \Phi$. When is the language $L(M_2)$ empty? $L(M_2)$ is empty when M does not accept w. Thus $L(M_1) = L(M_2)$ if and only if M does not accept w .

1. If M accepts w then $<M_1, M_2>$ is not in EQ_{TM}.
2. If M does not accept w then $<M_1, M_2>$ is in EQ_{TM}.

Thus f(<M, w>) = $<M_1, M_2>$ is a reducibility from A_{TM} to $\overline{EQ_{TM}}$. This shows that EQ_{TM} is not Turing recognizable and not co-Turing recognizable.

Theorem: Any language that is Turing recognizable is reducible to A_{TM}.

Let A be any Turing recognizable language. We have to show that $A \leq_m A_{TM}$. Let M be a TM such that L(M) = A. M exists as A is Turing recognizable.

Let us consider the following function f: $\sum^* \rightarrow \sum^*$ given by: f(x) = <M, x>.

All f does is appends M and x and makes the pair <M, x> and outputs it. Now,

1. If M accepts x then $x \in A$.
2. If M does not accept x then $x \notin A$.

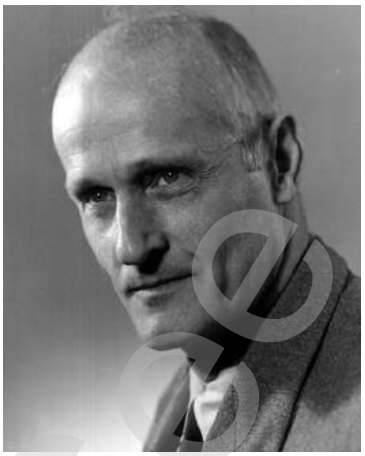
This function is computable as we can “hardwire” M in the program that computes f. Here is the machine that computes f.

1. On input x.
2. Output <M, x>

All it does it on input x produces the output <M, x>. This shows that $A \leq_m A_{TM}$.

Advance Topics in computability theory

A Lovely Puzzle
Can you write a program which when we run will produce its own description? (Here the program is not allowed to read any input device)
This puzzle will lead us to the work done by Stephen Kleene



A first try in C++
```
main() {
cout « "main() {";
cout « "cout « \"main() {\"; ";
. . . . . .
```
This is not going to end. We have to cleverer.

Let us consider the following instruction and follow it: Print the following sentence twice, second time in quotes “To be or not to be that is the question”. If we follow the instruction the result is: To be or not to be that is the question “To be or not to be that is the question”.

Let‟s change this a bit. Print the following sentence twice, second time in quotes “Hello how are you”. If we follow the instruction the result is: Hello how are you “Hello how are you”.

Now let us do something more interesting:
Print the following sentence twice, second time in quotes “Print the following sentence twice, second time in quotes”

If we follow the instruction the result is:
Print the following sentence twice, second time in quotes “Print the following sentence twice, second time in quotes”. The instruction is reproduced.

Recursion Theorem:
Programming language version

We will write a program that consists of two parts A and B.
1. A will print B.
2. B will print A.

And the program will end up printing itself.

```
A() { X="Hello";
cout « X;
B();}
```

```
B(X) { cout « "A() { X=" « X « ";";
cout « "cout « X;";
cout « "B();}";}
```

We assume that cout adds end of line automatically.

The output of B is
A() { X="Hello";
cout « X;
B();}
Which is A.

The output of A is, Hello. Thus the output of the whole program is: Hello followed by A.
If we change "Hello" to "Goodbye", the output of the whole program is Goodbye followed by A.

Thus if we change "Hello" to B. Then the output would be B followed by A.
Thus using this idea we should be able to write a program that prints itself. All we have to do is give the name main to the function A!

Recursion Theorem:
We want to make a machine that prints its own description. Let us start:

Theorem:

There is a computable function $q : \sum^* \rightarrow \sum^*$ where: q(w) = <p_w>.Where p_w is the machine that prints w, (on any input).
This is a very simple theorem. You can prove it yourself.

The algorithm is:

1. On input w.
2. Let P_w = "Print w"
3. Output <p_w>

The machine will have two parts:
B(<x>)
Find out $p_{<x>}$
Combine $p_{<x>}$ with X and output it.

B(x)
Find out the what is q(x)
Combine it with M
Print the combination.

Let us say x = <M>
On input X that is a description of a TM combine q(X) with X and output the combination.
Since B is a TM thus it has a description <B>. Let Q= q(<B>) = $p_{<B>}$

Let us define A to be Q.
Let us run A followed by B.
What will be the output?

A will put <B> on the tape.

1. B will compute q(<B>) = A
2. It will combine A with B and output it.

So it will output its own description.

Theorem (Recursion Theorem)

Let T be a Turing machine that computes a function $t : \sum^* \times \sum^* \rightarrow \sum^*$ then there is a Turing machine R that computes a function $r : \sum^* \rightarrow \sum^*$.

Where for every w, r(w) = t(<R>, w)

Let us see the proof. Let say $t : \sum^* \times \sum^* \rightarrow \sum^*$ is computed by a machine T. Let us describe a machine B. B(<X>, w)

Find out the description of q"(<x>) = p_x

Combine the description with X to make Z Write Z,w on the tape and call T.
Now, if we define A to be A = q"(<B>)

Then the combination of them is the machine promised in the recursion theorem.

1. A will put <B> on the tape.
2. B will compute q"(<B>) = A
3. It will combine A with B which is its own description.
4. Then it will call T.

Note: q' is different from q because of a slight technicality

T analyzes programs. R does the same analysis on itself. Let us look at an example. Let us define the following Turing machine.

1. On input <M>,w
2. Count the number of states in <M>
3. Print the number of states.

Thus T counts the number of states in a TM M and prints it. Hence t(<M>, w) = no of states in M. The recursion theorem says that there is a TM R that computes r such that r(w) = no of states in R. Thus there is a TM that prints a number which is the number of states in that very machine.

Lets consider another machine T.

1. On input <M>,w
2. print <M>

Thus t(<M>,w) = <M>

The recursion theorem says that there is a TM R such that: r(w) = <R>. Namely that there is a machine that prints its own description. We have already seen how to do this.

The recursion theorem says that a machine can obtain its own description and can go on to compute with it. By using recursion theorem we can add the following statement to our informal description any TM M that we are designing.
Obtain own description <SELF>

Consider the following machine M:

1. On input w
2. Obtain own description <SELF>
3. Print <SELF>

This is the machine that prints its own description. It is easy to design if we have recursion theorem at our disposal.

A computer virus is a program that is designed to spread itself among computers. A computer virus can be made via the recursion theorem. It first gets a copy of itself and then "plants" it in various places in the computer.

Lets prove
A_{TM} is undecidable
Using the recursion theorem. Suppose A_{TM} is decidable. Then there is a decider H that decides A_{TM}.

Consider the following TM B.

1. On input w
2. Obtain, via recursion theorem, own description <B>
3. Run H on $\langle B, w \rangle$. If H accepts reject w if H rejects accept w.

Clearly, B on any input w does the opposite of what H declares it does.
Therefore, H cannot be deciding A_{TM}. Simple!

Lets look at this proof closely. Consider the following TM T:

1. On input <W>, w
2. Run H on <M, w> if H accepts reject. if H rejects accept.

This TM exists as long as H exists. However, this TM is not running H on its own description but on <M>.

The recursion theorem now says that there is TM which we call B such that it computes a function b with b(w) = T(<B>, w) but T(<B>, w) = accept ↔ H rejects <B, w>. Thus B accepts w ↔ H rejects <B, w>. Hence, L(H) ≠ A_{TM}. a contradiction.

The length of the description of <M> is the number of symbols in its description. We say M is minimal if there is no TM which accepts the same language as M and has a shorter description. Let = {<M>: M is minimal}. This is a very interesting set. We will prove that

Theorem
is not Turing-Recognizable.

Two facts about MIN_{TM}.

1. MIN_{TM} is infinite.
2. MIN_{TM} contains TM"s whose length is arbitrarily large.

If MIN_{TM} is Turing Recognizable then there is a TM E that enumerates MIN_{TM}. We will use E and the recursion theorem to construct another TM C.

1. On input w
2. Obtain own description <C>
3. Run E until a machine D appears with a longer description than that of C.
4. Simulate D on input w.

All we have to note that eventually D will appear as MIN_{TM} contains TM with arbitrarily large descriptions. Now, what does C accept? L(C) = L(D). However, C has a smaller

description than D. So D cannot be in MIN_{TM}. A fixed point theorem

Theorem
Let t:$\sum^* \rightarrow \sum^*$ be a computable function. There is a TM F such that T(<B>)accepts the same language as F.

We should think t as a function that scrambles programs. This theorem says that any function that you write which scrambles programs will fail to scramble at least one program in the sense that the scramble program will continue accept the same strings as the original one.

The proof is very easy if we use the recursion theorem.
Consider F.

1. Input w.
2. Obtain own description <F>
3. Compute G = t (<F>)
4. Simulate G on w.

Clearly L(G) = L(F) and G = t (<F>). done!

Mathematical Logic is a branch of mathematics that investigates mathematics itself.
Main questions in logic are:

1. What is a theorem?
2. What is a proof?
3. What is truth?
4. Can an algorithm decide which statements are true?
5. Are all true statements

Logical Theories

Let us look at a few statements:

1. $\forall q \exists p \forall x.y\left[p \quad q \wedge \left(x.y \quad 1 \rightarrow xy \neq p \right) \right]$
2. $\forall a.b.c.n\left[\left(a,b,c \quad 0 \wedge n \quad 2 \right) \rightarrow a^n + b^n \neq c^n \right]$
3. $\forall q \exists p \forall x.y\left[p \quad q \wedge \left(x.y \quad 1 \rightarrow xy \neq p \wedge xy \neq p+2 \right) \right]$

The first statement is true a known for 2300 years. The second statement is true an took about 400 years to prove. The last statement is a famous open problem.

We want to make these notions very precise. So, we consider the alphabet

1. $\vee, \wedge, \neg$ are boolean operator
2. (,) are parenthesis
3. $\forall, \exists$ are quantifiers
4. $R_1, \ldots \ldots R_k$ are quantifiers

Note that if we need more than one variable we can use x, xx, xxx and so on. However, we will write them as x_1, x_2, ... Also, we will allow other boolean operators like $\rightarrow$.
Thus $p \rightarrow q$ is a short form for $p \vee \neg q$

$R_i(x_1, \ldots, x_j)$ is called an atomic formula. The value j is the arity of the relation symbol R_i. All appearances of R_i will have the same arity.

A formula is defined inductively as follows:
An atomic formula is a formula.
If $\varnothing_1$ and $\varnothing_2$ are formulas then $\varnothing_1 \wedge \varnothing_2, \varnothing_1 \vee \varnothing_2, \neg\varnothing_1$ are formulas and $\exists x_i [\varnothing_1]$ and $\forall x_i [\varnothing_1]$ are also formulas.

A quantifier can appear anywhere in the formula. The scope of a quantifier is the fragment of the formula that appears within the matched parenthesis or brackets. We will assume that all formulas are in prenex normal form, where all the quantifiers appear in front of the formula. A variable that is not quantified with a scope of a quantifier is called a free variable.

A sentence or a statement is a formula without any free variables. Lets look at some examples.

$$R_1(x_1) \wedge R_2(x_1, x_2, x_3).$$
$$\forall [R_1(x_1) \wedge R_2(x_1, x_2, x_3)].$$
$$\forall x_1 \exists x_2 \exists x_3 [R_1(x_1) \wedge R_2(x_1, x_2, x_3)].$$

We have only defined the syntax of formulas. Let us now give it some meaning. For that we need to specify two items.

1. One is a universe over which the variables will take values.
2. The other is an assignment of specific relations to relation symbols. Recall that a relation is an assignment of k tuples in the universe to {Ture, False}.

A universe with an assignment of relations to the relation symbols is called a model. Formally M is a tuple $(U, P_1, \ldots, P_k)$ where U is the universe and $P_1, \ldots, P_k$ are relation assigned to symbols $R_1, \ldots, R_k$. We refer to the language of a model to be the collection of formulas in which each R_i has the same airty as P_i. A sentence $\varnothing$ is either true or false in a given model. If true, we say M is a model of $\varnothing$.

Consider the sentence

$$\forall x \forall y [R_1(x, y) \vee R_1(y, x)]$$

Let us take the model $M_1 = (N, \leq)$. Thus our universe is the set of integers. The relation $\leq$ is assigned to the relation symbol R_1. This sentence is true. Since, in natural numbers either $a \leq b$ or $b \leq a$.

Now, let us take another model M_2 = (N,<). This model assigns < to R_1. Now the sentence is not true. If R_i is assigned to a customary symbol we can switch to infix notation. Thus in M_1 we can write:

$$\forall x \forall y [x \leq y \vee y \leq x]$$

Lets do another example:

$$\forall y \exists x [R_1(x, x, y)]$$

Lets assign PLUS to R_1. More precisely, PLUS(a, b, c) is true if a + b = c.
Is this sentence true? It is true if the universe is the real numbers. It is not true if the universe is integers.

If M is a model. We let TH(M) to be the theory of M. It is by definition the set of all sentences that are true sentences in the language of that M. A central point of investigation is to find out which theories are decidable and which ones are not.

We started with an alphabet.

$$\{\vee, \wedge, \neg, (,), \exists, \forall, x, R_1, R_2, \ldots, R_k\}$$

We considered formulas with this alphabet.
Some examples of formulas

1. $\exists x_1 \forall x_2 R_1(x_1, x_2) \vee R_2(x_1, x_3)$
2. $\forall x_1 \forall x_2 \exists x_3 R_2(x_1) \vee (R_3(x_1, x_3) \wedge R_1(x_2, x_2))$

A sentence is a formula without any free variables:
$\forall x_1 \exists x_2 R_1(x_1, x_2) \wedge R_2(x_2, x_1)$
This defines the syntax.
$\forall x R_1(x, x)$

To specify a model M. We have to specify

1. U the universe where the variables will take values.
2. $P_1, \ldots, P_k$ are relation assigned to symbols $R_1, \ldots, R_k$.

The language of a model M is the set of all formulas which use the relation symbols with correct airy

Suppose in the model M

1. R_1 is assigned a binary relation P_1.
2. R_2 is assigned a unary relation P_2.
3. R_3 is assigned a 5-ary relation P_3.

The formula

is in the language. Since, all relations have the right airty but
$\forall x_1 \exists x_2 (R_1(x_1, x_2) \vee R_2(x_4)) \wedge (R_3(x_2, x_1, x_3, x_4, x_5)$

$\forall x_1 \exists x_2 (R_1(x_1) \vee R_2(x_4)) \wedge (R_3(x_2, x_1)$
is not since the airty of two relations is wrong.

If M is a model. We let TH(M) denote the theory of M. It is by definition the set of all sentences that are true sentences in the language of that M.

A central point of investigation is to find out which theories are decidable and which ones are not.
Note that when we are talking about a model in which relations have some usual meaning then we shift to infix notation. We also saw examples of sentences that are true in one model and false in another model.

Let us look at one:

$$\phi = \forall y \exists x [R_1(x, x, y)]$$

1. M_1 was the model with N as the universe. R_1 was assigned PLUS. The sentence can be rewritten as and $\phi = \forall y \exists x [x + x = y]$ it is not true.

2. M_2 was the model with R as the universe. R_1 was again assigned PLUS. The sentence can be rewritten as $\phi = \forall y \exists x [x + x = y]$ and it is true.

Given a Model M we would like to know which sentences are true and which ones are not true. What we want is a decision procedure or an algorithm which will take a sentence and tell us if the sentence is true or not. One of the most interesting model is number theory where M with relation + and ×, =, ≤ with their usual meaning. This theory is undecidable. This is a major theorem by Alonzo Church building on the work of Godel.

Let us start with a simpler theory. This theory is called Presburger Arithmetic.

1. The universe is the natural numbers N.
2. Two relations + and = with their usual meaning.

So only multiplication is missing in Presburger Arithmetic.

Let us look at some sentences:

$\forall y \exists x [x + x = y]$

$\forall x \forall y \exists z [(x + y = z) \wedge ((x + x + y = z) \vee (x + y = z + z + z))]$

Note that it is possible to write 5x as x + x + x + x + x. However, we cannot write xy or x^2. So in this theory the variables are only multiplied by constants. We cannot use higher powers either.

We want to prove that this theory is decidable. So, we want to design an algorithm (or a TM) with will take as input a sentence and decide if it is true or not. How does the input look like:

$\forall y \exists z \exists x [(x + x = y) \vee (y + x = z)]$

$\forall x \forall y \exists z [(z + x = z + y) \wedge \neg (y + x = z + z])]$

How will the algorithm decide if a sentence is true.

Lets start with something simpler. Let us look at the simple atomic formula x + y = z. Can we design a DFA that accepts x, y, z if and only if they satisfy this equation?

Let us be more clear. How will the input be given to the DFA. Let us define

$$\Sigma_3 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \right\}$$

Now, we can give input to the DFA. Suppose x = 5, y = 3, z = 6 then the input to the DFA would be

$$\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

1. The top row 101 is x in binary.
2. The second row 011 is y in binary.
3. The third row 110 is z in binary.

Can we design a DFA where the input alphabet is Σ_3 and it accepts a string $I \in \Sigma_3^*$ such that the top row of the string when added to the second row of the string gives us the last row of the string?

The answer is yes. But it is easier if the machine is given the strings in reverse. Lets see how do we add in binary:

010100
001111

The way we add is we remember the carry and compute the next digit and keep moving left. If the numbers were given to us in reverse we will remember the carry and keep moving right. So, if these numbers were given to us in reverse.

c = 0
0 0 1 0 1 0
1 1 1 1 0 0
1

c = 0
0 0 1 0 1 0
1 1 1 1 0 0
1 1

c = 0
0 0 1 0 1 0
1 1 1 1 0 0
1 1 0

c = 1
0 0 1 0 1 0
1 1 1 1 0 0
1 1 0 0

c = 1
0 0 1 0 1 0
1 1 1 1 0 0
1 1 0 0 0

c = 1
0 0 1 0 1 0
1 1 1 1 0 0
1 1 0 0 0 1

If the answer were given to us. We could check it as we go along. Actually this DFA is very very simple. Let us look at it. It has only two state and a trap state. q_0, q_1 and q_r . q_0 is the accept state. Here are the transitions. When the machine is in q_0 the carry is 0. When it is in q_1 the carry is 1.

$$\delta(q_0, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}) = q_0 \qquad \delta(q_0, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}) = q_r \qquad \delta(q_0, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}) = q_r$$

$$\delta(q_0, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}) = q_0 \qquad \delta(q_0, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}) = q_1$$

So we can make a DFA which accepts all strings $I \in \Sigma_3^*$ such that
The top row of I read revered in binary + Second row of I read reversed in binary = Last row of I read reverse in binary. Thus if I represents three numbers x, y and z then the DFA accepts if and only if x + y = z.

What about other equations in Presburger Arithmetic. Let us think about
x + y = z + z + t
We can use the same trick. Let

$$\Sigma_4 = \left\{ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \cdots, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right\}$$

Each string $I \in \Sigma_4^*$ represents four numbers x, y, z and t written in reverse binary. We can make a DFA D such that D accepts I if and only if x + y = z + z + t

Given a atomic formula F (in Presburger Arithmetic) in i variables $x_1, \ldots, x_i$ variables we define $\sum_i$ to be an alphabet with 2_i characters. Each string S$\epsilon\sum_i$ defines i numbers $a_1 \ldots a_i$. We can make a DFA D such that D accepts S if and only if $a_1 \ldots a_i$ satisfy the equation F.

1. $x_1 + x_2 + x_2 + x_2 = x_3 + x_3 + x_4 + x_5$.
2. $x_1 + x_2 + x_3 + x_5 = x_4 + x_4 + x_4$

We can make DFA D_1 such which accepts strings over Σ_5^* in the first case and another DFA D_2 which accepts strings over Σ_5^*. So, in some sense DFAs can capture atomic formulas of Presburger Arithmetic.

What about non atomic formulas?

$(x_1 + x_2 + x_2 + x_2 = x_3 + x_3 + x_4 + x_5) \vee (x_1 + x_2 + x_3 + x_5 = x_4 + x_4 + x_4)$

$(x_1 + x_2 + x_2 + x_2 = x_3 + x_3 + x_4 + x_5) \wedge (x_1 + x_2 + x_3 + x_5 = x_4 + x_4 + x_4)$

$\neg(x_1 + x_2 + x_2 + x_2 = x_3 + x_3 + x_4 + x_5) \vee (x_1 + x_2 + x_3 + x_5 = x_4 + x_4 + x_4)$

In the first case we can make a DFA that accepts all the strings accepted by D_1 or D_2. Regular languages are closed under union, intersection and complementarily.

Given quantifier free formula f (in Presburger Arithmetic) in i variables $x_1 \ldots x_i$ variables we define $\sum_I$ to be an alphabet with 2_i characters. Each string S$\epsilon\sum_i$ defines i numbers $a_1 \ldots a_i$. We can make a DFA D such that D accepts S if and only if $a_1 \ldots a_i$ satisfy F.

What about formulas with quantifiers lets look at one of them:

$\exists x_3[x_1 + x_2 + x_2 = x_3]$

This formula has two free variables x_1, x_2

So lets work with $\sum_2$ and $\sum_3$. We can make a DFA D that accepts strings $S \in \Sigma_3^*$ such that S represents x_1, x_2 and x_3 which satisfy $x_1 + x_2 + x_2 = x_3$.

We would like a DFA D' that accepts strings $S' \in \Sigma_2^*$ such that S' represent x_1, x_2 and satisfy $\exists x_3[x_1 + x_2 + x_2 = x_3]$

It is much easier to make an NFA given the DFA D. The NFA N simply guesses the value of x_3 as it goes along moving to the right. Thus in each move it makes two non-deterministic moves. One pretending the current bit of x_3 is 0 and the other pretending that it is 1. We can then convert this NFA N to a DFA using a standard algorithm.

We can deal with quantifiers like this one by one. Let say we have a formula $\phi = \exists x_i \psi$

First we make an DFA D that accepts those over $\sum_i$ that represent $x_1, \ldots, x_i$ and satisfy. Then we make an NFA that uses D and guesses the value of each bit of x_i as it goes along. We convert this D to D_0 which now accepts strings over

How to handle $\phi = \forall x_i \psi$. This is easy since $\forall x_i \psi = \neg\exists\neg\psi$

First we make an DFA D that accepts those over $\sum_i$ that represent $x_1, \ldots, x_i$ and satisfy. Thus we can use the fact that regular languages are closed under complementarily to make a DFA D" that accepts ϕ. We also use the technique for handling $\exists$.

This way if we have a sentence $Q_1 x_1 Q_2 x_2 \cdots x_i \phi$
We can make a DFA D that will accept the empty string over $\sum_0$ if and only if the formula is true. In the end we run D on the empty string to see if accepts $\in$. If it does then the formula is true. Otherwise it is false.

Let us again look at these very interesting statements.

1. $\forall q \exists p \forall x, y[p > q \wedge (x, y > 1 \rightarrow xy \neq p)]$
2. $\forall a, b, c, n[(a, b, c > 0 \wedge n > 2) \rightarrow a^n + b^n \neq c^n]$
3. $\forall q \exists p \forall x, y[p > q \wedge (x, y > 1 \rightarrow xy \neq p \wedge xy \neq p + 2]$

These statements are not statements in Presburger Arithmetic.

The first two use × and the second one also uses exponentiation. Which is not allowed in Presburger Arithmetic. Let us now study a different Model. (N,×, +). In this Model

1. The universe is a set of integers.
2. We have × and + as relations with their usual meaning.

The first and the third statements are in the language of this Model. This to many mathematicians the the most important and interesting model. It is what we call number theory or the Peano Arithmetic.

Is the theroy of this Model decidable? Note that if this was the case, we would have an algorithm which could decide if Golbach conjecture is true. Such an algorithm would be amazing. It could tell us about the truth or falsehood of many mathematical statements. From now on we will refer to this model as peano arithmetic or number theory.

We will prove:

Theorem: TH(N,+,×) is not decidable.

This is both bad and good. This shows that no algorithm can be devised that can decide the truth and falsehood of general statements in number theory.

We will need the following auxiliary theorem:

Theorem: Let M be a TM and w a string. We can construct from M and w a formula $\phi_{M,w}$ in the language of (N,+,×) such that $\exists x \phi_{M,w}$ is true if and only if M accepts w.

The proof of this theorem is quite large and complicated. To make the proof a bit simpler we have provided the proof of the following theorem in the handout.

Theorem: Let M be a TM and w a string. We can construct from M and w a formula $\phi_{M,w}$ in the language of (N,+,×, ↑) (where ↑ denotes exponentiation) such that $\exists x \phi_{M,x}$ is true if and only if M accepts w.

Notice that the statement of this theorem is the same as the previous one, except that in our model now we also allow exponentiation. Even this is quite technical to prove. So, it is included as a handout for the interested students.

The basic ideas used in this theorem are:

1. The formula $\phi_{M,w}$ "says" that x is a (suitably encoded) accepting computation of M on w.
2. The operation ↑,× and + are used to extract symbols or chunks of symbols.
3. These chunks are then compared to make sure that the computation is valid.

Using this theorem we can now prove

Theorem: Th(N,+,×) is undecidable.

Proof. We can reduce A_{tm} to TH(N,+,×). Recall A_{TM} = {<M, w> : M accpets w}.
We give a mapping reduction.

1. On input <M,w>
2. Construct $_{M,w}$ and output it.

Clearly, M accepts w if and only if $_{M,w}$ is true.

We are now in a position to prove an extremely celebrated theorem of Kurt Gödel. Gödel was a German logician, perhaps, the greatest in history.

Gödel's Theorem:

He proved a theorem that shocked and delighted the mathematical world. His theorem is called the incompleteness theorem. The story goes back to Hilbert who outlined a program in 1920.

The main goal of Hilbert"s program was to provide sound foundations for all mathematics.

Hilbert"s program was

1. A formalize of all mathematics. All mathematical statements should be written in a formal language and manipulated using well defined rules.
2. Show that it is Complete. Prove that all true mathematical statements can be proved.
3. Show that it is consistent. No false statement can be proved.
4. Decidability. Show that there is an an algorithm for deciding the truth of any mathematical statement

Hilbert suspected that all mathematical statements can be proved from number theory.

Godel showed that there are statements in TH(N,+,×) which cannot be proved. TH(N,+,×) cannot prove its own consistency. This showed that Hibert"s program was doomed. Godel Theorem is considered to be a corner stone in foundation of mathematics. It is an extremely important work with far-reaching consequences.

Let us prove:
There are statements in TH(N,+,×) which cannot be proved. In order to do this we have to define what a proof is. But, instead of going into the exact definitions let us be a bit informal in our treatment. We start with a set of axioms. For example the set of axioms for numbers is the following:

1. $\forall x \forall y \forall z\,((x + y) + z = x + (y + z))$
2. $\forall x \forall y\,(x + y = y + x)$
3. $\forall x \forall y \forall z\,((x \cdot y) \cdot z = x \cdot (y \cdot z))$
4. $\forall x \forall y\,(x \cdot y = y \cdot x)$
5. $\forall x \forall y \forall z\,(x \cdot (y + z) = x \cdot y + x \cdot z)$
6. $\forall x\,((x + 0 = x) \wedge (x \cdot 0 = 0))$
7. $\forall x\,(x \cdot 1 = x)$
8. $\forall x \forall y \forall z\,((x < y \wedge y < z) \Rightarrow x < z)$
9. $\forall x\,\neg(x < x)$
10. $\forall x \forall y\,(x < y \vee x = y \vee x > y)$
11. $\forall x \forall y \forall z\,(x < y \Rightarrow x + z < y + z)$
12. $\forall x \forall y \forall z\,(0 < z \wedge x < y \Rightarrow x \cdot z < y \cdot z)$
13. $\forall x \forall y\,(x < y \Rightarrow \exists z(x + z = y))$
14. $0 < 1 \wedge \forall x\,(x > 0 \Rightarrow x \geq 1)$
15. $\forall x\,(x \geq 0)$

We add to this set of axioms $(\phi(0) \wedge \forall x(\phi(x) \Rightarrow \phi(x + 1)) \Rightarrow \forall x \phi(x))$
This is better known as the principle of mathematical induction.

Once we have defined axioms we can now define a proof.
A proof π of a statement S is a sequence of statements $S_1, \ldots, S_l$ such that

1. Each S_i is an axiom or
2. Each S_i follows from the previous S_j "s using rules of logic. Which are precise and well defined.

An Example of rules of logic is given S and S → T we can conclude T. Note that this rule is purely syntactic.

With these in mind it is easy to see:

1. The correctness of a proof can be checked mechanically. That is via a Turing machine. In otherwords $\{\langle \phi, \pi \rangle : \pi \text{ is a proof of } \phi\}$ is decidable.
2. [redacted]

Theorem: The set of proveable statements in TH(N,+,×) is Turing recognizable.

Proof.

1. On input Φ
2. Enumerate all possible proofs $\pi_1, \ldots,$.

check if π_i is a proof of Φ. If it is accept.

Theorem: Some true statements in TH(N,+,×) are not proveable.

Consider the following algorithm:

1. On input Φ
2. Enumerate all possible proofs $\pi_1, \ldots,$.
3. Check if π_i is a proof of Φ. If it is accept.
4. Check if π_i is a proof of ¬Φ. It it is reject.

If all ture statements are proveable then since each statement is either true or false hence Φ or ≠ Φ is proveable. Thus the above algorithm will be a decider for TH(N,+,×)
This is a contradiction as we have already proved that TH(N,+,×) is not decidable.

The previous theorem just says that there are true statements which are not proveable. Can we actually construct such a statement? This is what Godel Actually did. He constructed a statement which was true but not proveable. Let us also construct such a statement. We will use the recursion theorem to construct such a statement.

1. On input w
2. Obtain self description <S>.
3. Construct a statement
4. if is proveable accept w.

The fourth step can be done as the set of all proveable statements is Turing recognizable. Let us clearly see what Φ is saying. Recall that says that S accepts 0. Hence says that S does not accept 0. This statement is true if and only if S does not accept 0.

1. If S finds a proof of Ψ it will accept 0 (and all its inputs).
2. Ψ says that S does not accept 0.

Thus if S finds a proof of Ψ then Ψ is false. But a false statement cannot have a proof. Therefore Ψ is true. But that means S does not find a proof of it and hence it is u provable. Gödel also proved that TH(calN,+,×) cannot prove its own consistency. These two theorems were corner stones of logic.

Oracles:

Recall the definition of mapping reducibility.
A is mapping reducible to B, written as $A \leq_m B$ if there is a computable function $f : \sum^* \rightarrow \sum^*$ such that for all x, x Є A if and only if f(x) Є B.
Mapping reducibility is not the most general way to reduce. Recall that we proved that:

Theorem: If $A \leq_m B$ and if B is decidable the A is decidable.

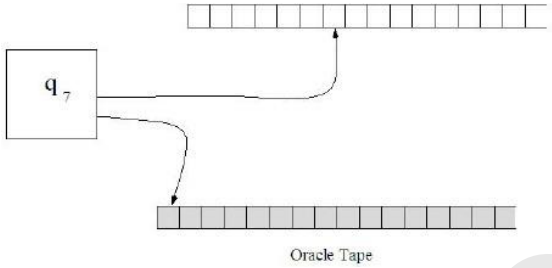
Theorem: If $A \leq_m B$ and if B is Turing-recognizable the A is Turing-recognizable.

We know that A_{TM} is Turing-recognizable. We also know that is not Turing-recognizable. From this we can conclude.

Theorem: $\bar{A}_{TM}$ is not mapping reducible to A_{TM}.

But intuitively a Language and its complement are almost the same problem. So they should be reducible to eachother. We will now define a new concept that will capture this intuition. For this we will introduce the notion of Oracles. What are Oracles? Where does the word come from?

An oracle for a language B is an external devise that solve the membership problem in B. An oracle tape TM has a special tape on which queries are written and the machine at any point in its computation write a string w on its oracle tape and enters a special state $q_?$. If w Є B then the machine enters in the state q_y otherwise it enter the state q_y. Here is a picture of an oracle tape TM.



Having an oracle B is like having access to a magical power that answers questions. A Turing machine with oracle B will be denoted by M_B.
Lets look at an example of an oracle tape TM.

We know that A_{TM} is undecidable. A_{TM} = {<M,w> : Maccpetsw} and E_{TM} is undecidable. Here E_{TM} = {<M> : L(M) = Φ}
We want to construct a oracle TM T_{ATM} that decides E_{TM}. So we are asking the following question. If we assume someone can solve A_{TM} can we use that knowledge to solve E_{TM}.
Let M be a TM. Let us consider another another TM N.

1. On input x
2. using dovetailing run M on all strings.
3. If M accepts any string then accept x.

We note that L(M) ≠ Φ then N accepts all strings. On the other hand if L(M) = Φ then N does not halt on any string.

Now consider T^$_{ATM}$.

1. On input <M>
2. Construct N
3. Ask the oracle if <N, 0> that is if N accepts 0.
4. If the oracle says yes, reject <M>.
5. If the oracle says no, then accept <M>

it is now easy to see that T_{ATM} accepts E_{TM}.

Note that we do not have an algorithm for solving E_{TM}. All we have shown is that if A_{TM} could be solved then we can solve E_{TM}. So in this case, we say that E_{TM} is decidable relative to A_{TM}.

A language A is Turing reducible to B, written as $A \leq_T B$ if A is decidable relative to B.
More precisely, there is a Turing machine with oracle B, M_B such M_B halts on all inputs and $L(M_B) = A$.

Let us ask what would happen if we could solve the halting probelm. How powerful are TM"s which have access to an oracle for A_{TM}.

A definition of information: Lecture No 19

What is information? There are many definitions of information. Let us start with a paradox which is called the Berry Paradox.

The smallest positive integer that requires at least twenty words to describe.

1. Since there are infinitely many integers hence we cannot describe all of them in under twenty words.
2. Hence there must be many integers that require more than twenty words to describe.
3. Out of them there is one which is the smallest.

This sentence is describing a number. However, it is describing a number which requires at least twenty words to describe.

The smallest positive integer that requires at least twenty words to describe.
But this sentence itself has only twelve words! This sentence is describing in twelve words a number that requires at least twenty words to describe!!!! This is a paradox.

Actually there is no paradox. When we realize that the word "describe" does not have a precise meaning in English. When we try to agree on what a description of a number is the paradox goes away. For example if we say that we describe numbers by saying their digits then this particular sentence is not a description of a number.

Let us start by looking at two strings:

A = 10101010101010101010101010101010101010

B = 10101000101001101000100100111101001011

Intuitively A seems to have less information than B. This is because I can describe A quite easily

1. The string A is 01 repeated 20 times.
2. The string B seems to have no such description.

There are two ways to describe the string A. One is to write its individual bits and the other is to simply say that it is 10 repeated fifteen times. We will be interested in the shortest description.

We will impose two restrictions.

1. We will only talk about describing strings of 0"s and 1"s.
2. The description of a string must also be a string of 0"s and 1"s.

In some sense we are trying to compress the strings.

The most general way to describe a string s is to give an algorithm A that produces that string. Thus instead of describing a string s we simply describe A. Let us make this idea more precise.

Formally and algorithm is just a TM so all we have to do is to describe a TM. Lets start carefully. Let us first consider a TM M and a binary input w to M. Suppose we want to describe the pair (M,w). How to perform this encoding? We can take the description of M that is <M> and concatenate w after it. There is a problem with this.

Suppose we send this description to a friend. The friend would not know. Where the description of M ends and where does w start from.

To avoid this problem we can have a convention. All the strings in the description of M would be

doubled. The description will be followed by 01 and then we will simply write down w. With this rule there will be no ambiguity in finding out where w starts.

Let us say we have 00110011001111110011**01**101011
In this case,

1. <M> = 1001011101
2. w = 101011

So given a TM M and a string w. We can run M on w.
Suppose the output is x then <M,w> is one possible description of x. In order to describe a string x we describe <M,w> such that running M on w outputs x. However, for a given string x there maybe many TM"s and input pairs that describe x. The minimal description of x, written d(x), is the shortest string <M,w> such that

1. M halts on w
2. Running M on w produces x

If there are several such strings we select the lexicographically first amongst them. The descriptive complexity of x, written as K(x), is K(x) = |d(x)|. It is the smallest description of x.

Lets look at a practical example: Java. You can think of hM, xi to be a java program M and its data w. Your computer downloads the whole program and the data and when you run it on your own computer with data w it produces a lot of other data.
You can download a program that produces the digits of π. Thus the digits of π have been communicated to you in a very small program.

K(x) is the smallest description of x. It is sometimes called

1. Kolmogorov complexity
2. Kolmogorov-Chaitin complexity.

It is measuring how much information is in a given string. It also makes sense to talk about descriptive complexity of infinite strings. But, we will not do so. As, an example the digits of π is an infinite string with finite descriptive complexity.

Lets prove some simple but useful facts:

1. There is a constant c such that $K(x) \leq |x| + c$
2. There is a constant c such that $K(xx) \leq K(x) + c$
3. There is a constant c such that $K(xy) \leq 2K(x) + K(y) + c$

Let us understand these first.

1. The description of x does not need to be too much larger than x.
2. The description of xx does not need to be too much larger than description of x.
3. The description of xy does not have to be too much larger than then description of y + twice the description of x.

Theorem: There is a constant c such that $K(x) \leq |x| + c$

Let M_0 be the TM that halts as soon as it starts. So, it does not change its input at all. We can always describe x by $<M_0>$, x. This description has size $2|<M_0>| + |x|$

What is important here is the $|M_0|$ is independent of the length of x. Let $c = 2|M_0| + 2$ then there is a description of x of size |x| + c. Since K(x) is the minimal length description so it cannot be larger than |x| + c.

Note that we are saying that we can append the description of the "NULL" machine to each string. Why don"t we just leave M_0 out. This is because anyone reading the description is expecting to see a description of some machine first then 01. If we agree that M_0 will be denoted by the empty string then we can simply send 01x. But even in this case c = 2.

Theorem: There is a constant c such that K(xx) ≤ K(x) + c
We maybe tempted to give the following proof. Let M_1 be a TM that doubles its string; that is,

1. On input x
2. Output xx

We can describe xx with hM_1ix. However this only shows that K(xx) ≤ |x| + c.
So we have to be more clever.

Theorem: There is a constant c such that K(xx) ≤ K(x) + c

Let M_3 be a TM that takes hNiw as inputs and doubles the output after running N on w.

1. On input <N,w>
2. Run N on w.
3. Let s be the output of N on w.
4. Write ss.

A description of xx is given by <M>d(x). Recall that d(x) is the minimal length description of x. In the previous proof d(x) was the smallest description of x. We appended the message, Double your output. To it to obtain a description of xx.

Theorem: There is a constant c such that K(xy) ≤ 2K(x) + K(y) + c
Let M be a TM that breaks a string as follows:

1. On input w.
2. Find the first 01 in w.
3. Call the string before the 01 A and the rest B.
4. Undouble the string A into A'.
5. Let s be the string described by A'.
6. Let t be the string described by B.
7. Output st.

Now consider <M> double (d(x)) 01d(y). Then it is clear that when M is run on double d(x)d(y) it produces xy hence K(xy) ≤ +|<M>| + 2|d(x)| + |d(y)| + 2,
So we may take c = |<M>| + 2. One can show

Theorem: There is a constant c such that K(xy) ≤ 2 logK(x) + K(x) + K(y) + c and we can even improve this by using more sophisticated coding techniques.

Optimality of Definition:

We have a very important question we can ask. What happens if we use some other way of describing algorithms. Lets say we say algorithms are java programs. Wouldn"t that change the descriptive complexity?

Let p : $\sum^* \rightarrow \sum^*$ be any computable function. Let us define $d_p(x)$ be the length of the lexicographically shortest string s such that p(s) = x. We define $K_p(x) = |d_p(x)|$

For example we can let $K_{java}(x)$ to be the length of the shortest java program that outputs x. The question is what happens if we use K_{java} instead of K? The answer is that descriptive complexity only changes by an additive constant. Thus K(x) is in some sense universal.

Theorem: Let p be a computable function $p : \sum^* \rightarrow \sum^*$ then there is a constant c such that $K(x) \leq K_p(x) + c$

The idea is that we can append an interpreter for java in our descriptions and this changes the description length by a constant.
Consider the following TM M

1. On input w
2. Output p(w)

Let $d_p(x)$ be the minimal description of x (with respect to p). Then consider $hMid_p(x)$. It is clear that this is a description of x. The length is $2|<M>| + 2 + d_p(x)$
So we may take $c = 2|<M>| + 2$.

Incompressible strings:

Theorem: In compressible strings of every length exists.

The proof is by counting. Let us take strings of length n. There are 2^n strings of length n.
On the other hand each description is also a string. How many descriptions are there of length $\leq n - 1$. There are $2^0 + 2^1 + \cdots + 2^{n-1} = 2^n - 1$ descriptions of length $\leq n - 1$.
Hence at least one string must be incompressible.

Theorem: At least $2^n - 2^{n-c+1} + 1$ strings of length n are incompressible by c.

The proof is again by counting. Let us take strings of length n. There are 2^n strings of length n. On the other hand each description is also a string. How many descriptions are there of length $\leq n - c$).

There are $2^0 + 2^1 + \cdots + 2^{n-c} = 2^{n-c+1-1}$ descriptions of length $\leq n - 1$. Therefore, at least $2^n - 2^{n-c+1} + 1$ strings are c-incompressible.

1. Incompressible strings look like random strings.
2. This can be made precise in many ways.

Many properties of random strings are also true for incompressible strings. For example

1. Incompressible strings have roughly the same number of 0̎s and 1̎s.
2. The longest run of 0 is of size O(log n).

Let us prove a theorem which is central in this theory.

Let us formalize this statement. A property p is a mapping from set of all strings to {TRUE, FALSE}. Thus $p:\{0, 1\}^* \rightarrow \{true, false\}$.

We say that p holds for almost all strings if

$$\frac{|\{x \in \{0,1\}^n : p(x) = \text{ false }\}|}{2^n}$$

approaches 0 as n approaches infinity.

Theorem: Let p be a computable property. If p holds for almost all strings the for any b > 0, the property p is false on only finitely many strings that are incompressible by b.

Let M be the following algorithm:

1. On input i (in binary)
2. Find the ith string s such that p(s) = False. Considering the strings lexicographically.
3. Output s.

We can use M to obtain short descriptions of string that do not have property p. Let x be a string that does not have property p. Let i_x be the index of x in the list of all strings that do not have property p ordered lexicographically. Then hM, i_x i is a description of x. The length of this description is $|i_x| + c$

Now we count. For any give b > 0 select n such that at mos

t

fraction of the strings of length ≤ n do not have property p. Note that since p holds for almost all strings, this is true for all sufficiently large n. Then we have

Therefore, $|i_x| \leq n - b - c$. Hence the length of hMii$_x$ is at most $n - b - c + c = n - b$. Thus $K(x) \leq n - b$ which means x is b compressible. Hence, only finitely many strings that fail p can be incompressible by b.

Recall that a string, x, is if $K(x) \leq |x| - b$

1. A string is b-incompressible if it is not b-compressible.
2. A string is incompressible if it is not 1-compressible.

Incompressible strings cannot be compressed. Their description is at least as long as the string itself.

1. Incompressible strings look like random strings.
2. This can be made precise in many ways.

Many properties of random strings are also true for incompressible strings. For example, we proved two theorems:

Theorem: For every n there is a incompressible string of length n.

Theorem: The number of b-incompressible strings of length b is at least $2^n - 2^{n-b+1} + 1$

The proof is simple counting!

1. Incompressible strings have roughly the same number of 0"s and 1"s.
2. The longest run of 0 is of size O(log n).

Let us prove a theorem which is central in this theory.

Theorem: A computable property that holds for "almost all strings" also holds for incompressible strings.

Let us formalize this statement. A property p is a mapping from set of all strings to {TRUE, FALSE}. Thus p:{0, 1}* → {true, false}.

We say that p holds for almost all strings if

$$\frac{|\{x \in \{0,1\}^n : p(x) = \text{ false }\}|}{2^n}$$

approaches 0 as n approaches infinity.

Theorem: Let p be a computable property. If p holds for almost all strings the for any b > 0, the property p is false on only finitely many strings that are incompressible by b.

Let M be the following algorithm:

1. On input i (in binary)
2. Find the ith string s such that p(s) = False. Considering the strings lexicographically.
3. Output s.

We can use M to obtain short descriptions of string that do not have property p. Let x be a string that does not have property p. Let i_x be the index of x in the list of all strings that do not have property p ordered lexicographically. Then <M, i_x > is a description of x. The length of this description is $|i_x| + c$

Now we count. For any give b > 0 select n such that at most

fraction of the strings of length ≤ n do not have property p. Note that since p holds for almost all strings, this is true for all sufficiently large n. Then we have

Therefore, $|i_x| \leq n - b - c$. Hence the length of <M>i_x is at most $n - b - c + c = n - b$. Thus $K(x) \leq n - b$, which means x is Hence, only finitely many strings that fail p can be incompressible by b. We can ask a very interesting question now: Is it possible to compress a string that is already compressed? The answer to this question seems to be no! Let us prove something that is quite close to this.

Theorem: There is a constant b such that for every string x the minimal description d(x) is incompressible by b.

Roughly the idea of the proof is that, If d(x) were compressible we could use the compressed version of d(x) to obtain a smaller description of x than d(x).
Consider the following TM M:

1. On input <R, y>
2. Run R on y and reject if its output is not of the form <S, z>
3. Run S on z and halt with its output on the tape.

This machine is"decoding twice". Let b = 2|<M>| + 3. We now show that b satisfies the statement of the theorem. Suppose on the contrary that d(x) is b compressible for some string. The $|d(d(x))| \leq |d(x)| - b$. But what happens if we run M on d(d(x)). Note that it outputs x.

Thus < M > d(d(x)) is a description of x. How long is this description?

$$2|<M>| + 2 + |d(d(x))| = (b - 1) + (|d(x) - b) = |d(x)| - 1$$

This contradicts the fact that d(x) is the minimal description of x.

Lecture No 21

Time

Resource Bounded Computations

Now we will look at resource bounded computations. Two of the most important resources are going to be

1. Time 2. Space

Let A = { 0^k1^k : k ≥ 0}

1. Consider the following TM that accepts A
2. Scan across the tape to check if all 0"s are before 1"s if not reject.
3. Scan across the tape crossing of a 0 and a 1 each time.
4. If 0"s remain and 1"s finished reject.
5. if 1"s remain and 0"s are finished reject.
6. if both 0"s and 1"s are finished accept.
7. Goto Step 2

We want to analyze this algorithm and talk about how many steps (time) does it take. We saw an algorithm that accepts A in time O(n^2).

A step of the Turing machine is assumed to take unit time. Lets start with a definition. Let M be a TM that halts on all inputs. The running time or the time complexity of M is a function f: N → N such that f(n) is the maximum number of steps that M uses on any input of length n.

1. We say that M runs in time f(n).
2. M is f(n)-time TM.

Big O Notation:

We want to understand how much time a TM takes in the worst case. The exact running time t(n) is going to be very complicated expression. In some cases we will not be able to compute it. But what we are interested in is getting a rough idea of how t(n) grows for large inputs. So we will use asymptotic notation.

Lets look at an example let f (n) = 5 n^3 + 13 n^2 − 6n + 48. This function has four terms. He most important term is 5 n^3 which will be the bulk if n is large. In fact 5 n^3 the constant 5 does not contribute to the growth of this function. Thus we will say that f(n) is order n^3 . We will denote this by f (n) = O(n^3). Let us state this precisely:

Let f,g:N→R+. We say that f (n)=O(g(n)) if there are positive integers c, n0 such that for all n≥ n0, f (n) ≤ cg(n) When f (n) = O(g(n)) we say that g is an asymptotic upper-bound on f . Sometime we will just say that g is an upper bound on f.

Intuitively if f (n) = O(g(n)) you can think that f is less than or equal to some multiple of g. For most functions the highest order term is is an obvious candidate to be g.

Let f(n)= 3 n^4 +7 n^2 +13 then f(n)=O(n^4).
Furthermore, f(n)=O(n^5). But it is not true that f(n)=O(n^3).

Suppose that f(n)= logb n where b is a constant. Note that logb n = $\log_2 n / \log_2 b$. Hence, we can say f(n)=O(logn). Let $f_2(n)$ = 5n log n + 200n log log n + 17, then f2(n) = O(nlogn). We can use O notation in exponents also. Thus $2^{O(n)}$ stands for a function that is upper bounded by 2^{cn} for some constant c.

We have to be careful though. For example consider $2^{O(\log n)}$. Well remember that O(log n) has a constant attached to it. Thus this function is upper bounded by $2^{c\log n}$ for some constant c. Now, we get $2^{c\log n} = (2^{\log n})^c = n^c$. Thus $2^{O(\log n)}$ is polynomial.

Bounds of the form n^c are called polynomial bounds. Where as bounds of the form 2^{nc} are called exponential bounds.

Little Oh Notation:
The Big-Oh notation roughly says that f is upper bounded by g. Now we want to say f grows strictly slower that g. Let f,g:N → R+. We say that f (n) = o(g(n)) if $\lim_{x\to\infty}(f(n)/g(n)) = 0$. Thus the ratio of f and g becomes closer and closer to 0.

1. $\sqrt{n} = o(n)$
2. n = o(n log log n)
3. n log log n = o(n log n)
4. n log n = o(n^2)
5. n^2 = o(n^3)

Lets look at a new TM that recognizes
A = { 0^k1^k : k ≥ 0}

1. Scan the tape and make sure all the 0s are before 1s.
2. Repeat till there are 0s and 1s on the tape.
    1. Scan the tape and if the total number of 0s and 1s is odd reject.
    2. Scan and cross of every other 0 and do the same with every other 1.
    3. If all 0s and 1s are crossed accept.

Thus if we have 13 0s in the next stage there would be 6. This algorithm shows that there is O(nlogn) time TM that decides A. Since, nlogn = o(n^2). This points to a real algorithmic improvement.

Time Complexity Classes Let t : N → N be a function. Define the time complexity class TIME(t(n)) to be: TIME(t(n)) = {L|L is decided by an O(t(n))-time Turing machine}.

So for example:

1. TIME(n) is the set of all languages, L, for which there is a linear time TM that accepts L.
2. TIME(n^2) is the set of all languages, L, for which there is a quadratic time TM that accepts L.

Recall A = { 0^k1^k : k ≥ 0}. Our first algorithm (or TM) showed that A Є TIME(n^2). The second one showed that A Є TIME(n log n). Notice that TIME(nlogn) _ TIME(n^2). What will happen if we allow ourselves to have a 2-tape TM.

Can we have a faster TM. Yes, if we have two tapes we can do the following:

1. Scan across to find out if all 0s are before 1s.
2. Copy all the 0s to the second tape.
3. Scan on the first tape in forward direction and the second tape in reverse direction crossing off 0s and 1s.
4. If all 0s have been crossed off and all ones have been crossed off accept else reject.

In fact, it one can design a 2-tape TM that accepts A while scanning the input only once! This leads to an important question. Can we design a 1-tape TM that accepts A in o(n log n) time, Preferably linear time? The answer is NO!

In fact there is a language L such that

1. L is accepted by a 2-tape TM in time O(n).
2. L cannot be accepted by any 1-TM TM in time o(n^2).

Thus the definition of TIME(t(n)) changes if we talk about 1-tape TMs or 2-tape TM. Fortunately, it does not change by too much as the following theorem shows.

equivalent O(t2(n))-time single tape TM.

Notice the difference from computability theory. Previously we showed:

Theorem: Every k-tape TM has an equivalent 1-tape TM.

(We ignored time). Now, we are doing more detailed analysis.

The proof of the theorem is just a more careful study of the previous proof. Remember given a k-Tape TM, M, we made a 1-tape TM, N. N works as follows:

1. On input x convert the input to #q0#x#· · ·# the start configuration of M. This configuration says that x is on the first tape. The rest of the tapes are empty and the machine is in q_0.
2. In each pass over the tape change the current configuration to the next one.
3. If an accepting configuration is reached accept
4. If a rejecting configuration is reached reject.

Now, we just have to estimate how much time does N require.

On any input x of length n, we make the following claims.

1. M uses at most t(n) cells of its k-tapes.
2. Thus any configuration has length at most kt(n) = O(t(n)).
3. Thus each pass of N requires at most O(t(n)) steps.
4. N makes at most t(n) passes on its tape (after that M must halt).

This shows that N runs in time O(t(n) × t(n)) = O(t^2(n)).
Where did we use the fact that t(n) ≥ n. In the first step the machine converts x to the initial configuration. This takes time O(n). Thus the total time is actually O(n) + O(t^2(n)) = O(t^2(n)). This is a reasonable assumption as M requires time O(n) to read its input. Which is true for most problems. The answer usually depends on the entire input.

You have studied many non-deterministic models such as:

1. N_{FA}: Non-deterministic finite automata.
2. N_{PDA}: Non-deterministic pushdown automata. Now, we will define non-deterministic TMs.

We will now define non-deterministic TMs.

A non-deterministic TM is formally given as M =(∑, Γ, q_0, qc, qr, δ) where

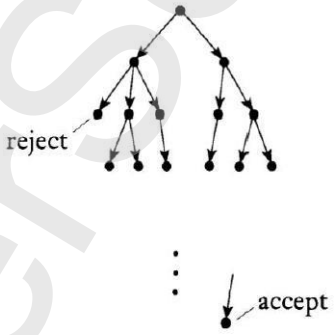
1. ∑ is the input alphabet.
2. Γ is the work alphabet.
3. q_0, qa, qr are start, accept and reject states.
4. δ is now a transition function: δ: Q × Γ :→ P(Q × Γ × {L, R}).

For example we may have $\delta(q_5, a) = \{(q_7, b, R), (q_4, c, L), (q_3, a, R)\}$.
Thus the machine has three possible ways to proceed.

We can think of a non-deterministic computation as a tree:

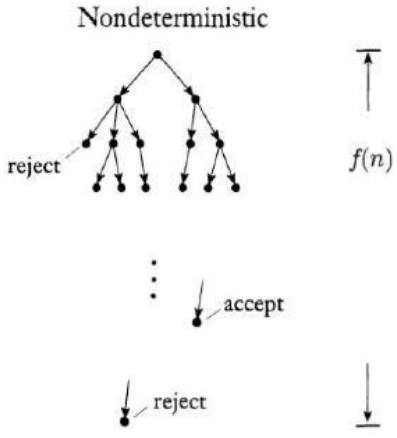
We will say that a non-deterministic TM M accepts x. If x is accepted on at least one branch.

On the other hand M rejects x if all branches of the tree reach the reject state.



The nondeterministic running time of M is the length of the longest branch. More, formally,

The running time of a nondeterministic TM M is the function f : N → N such that f (n) is the maximum number of steps. M uses on any branch of its computation on any input of length n.



A few comments on the definition:

1. The notion of non-deterministic time does not correspond to any real world machine we can make.
2. It is merely a mathematical definition.
3. We will see that this definition is extremely useful. It allows us to capture many interesting problems.

Let us prove some relation between non-deterministic time and deterministic time.

Theorem: Let t(n) be a function, where t(n) ≥ n. Then every t(n) time nondeterministic single tape TM has an equivalent 2O(t(n)) time deterministic single tape TM.

Thus this theorem says that we can get rid of non-determinism at the cost of exponentiating the time!

Let N be a non-deterministic TM that runs in time t(n). Let us first estimate how big the computation tree of N can be:

1. The tree has depth t(n).
2. Let b be the maximum number of legal choices given by Ns transition function.
3. Then this tree can have bt(n) leaves.
4. The total number of nodes can be at most 1 + b + b2 + · · · + $b^{t(n)} \leq 2b^{t(n)}$.

A deterministic TM D simulates the computation tree of N. There are several ways to do this. One is given in the book. Here is a slightly different one. To begin with D is going to be a multi-tape TM.

Consider a string over R = 1, 2, ,b of length t(n). So the string 12313412 corresponds to a computation of M where

1. Starting from the root go to the first child c1 of the root.
2. Then the second child of c_1 call it c_2.
3. Then to the third child child of c_2 call it c_3.
4. Then to the first child of c_3

Thus each such string corresponds to a branch in the tree.

Thus D is the following TM

1. On input x of length n.
2. For all strings s Є $R^{t(n)}$.
3. Simulate N making the choices dictated by s.
4. If N accepts then accept.
5. If all branches of N reject then reject.

This method assumes that t(n) can be computed quickly. However, we can also eliminate this requirement. For each string s the simulation can take O(t(n)) time. There are $b^{t(n)}$ strings to simulate. Thus the total time is O(t(n)) $b^{t(n)} = 2^{O(t(n))}$.

Our last part is to convert this multi-tape TM into a single tape TM. If we use the previous theorem then the total time would be $(2^{O(t(n))})^2 = 2^{2O(t(n))} = 2^{O(t(n))}$. We will come back to NTMs again.

P Classes:

We want to know which problems can be solved in practice. Thus we define the class P as follows:

$$p = \bigcup_k TIME(n^k)$$

or more elaborately write is as:

P = TIME(n) ∪ TIME(n^2) ∪ TIME(n^3) ∪ TIME(n^4) · · · .

The class P is important because:

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape machines.
2.

Thus in defining P we can talk about single tape TMs, Java programs, multi-tape TMs. They all give the same definition of P.

The second point is . Some people may object to this as they can point out that if a problem can be solved in time O(n^{5000}), then such an algorithm is of no practical value. This objection is valid to a certain point. However calling P to be the threshold of practical and unpractical problems has been extremely useful.

Let us consider a problem that is in P. Let us define

PATH

We want to show that PATH is in P. Lets understand the question first.

Question: Design a polynomial time algorithm that takes as input a graph G and two vertices s and t (suitably encoded) and decides if there is a path from s to t.

How will we encode the graph G. There are two ways to do this

1. Adjacency Matrix
2. Adjacency List

How do we show that PATH is in P. We have to give a polynomial time algorithm for this problem. The basic idea is “Start BFS or DFS from s and if t appears then there is a path from s to t”. Lets look at this algorithm in detail.

1. On input <G, s, t> where G is a digraph.
2. Mark s
3. Repeat till no additional nodes are marked:
4. Scan the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark b.
5. If t is marked accept. Otherwise, reject.

To analyze the algorithm we first have to compute the size of the input. The input size is at least m, where m is the number of nodes in G. We show that algorithm runs in time polynomial in m.

The repeat loop can at most run for m time. Each time all the edges are scanned. Since the number of edges is at most m2 thus step 2 takes at most m^2 time. So, the total time is at most m^3 . Hence we have shown that PATH is in P.

Let us take another problem. Let us say x and y are relatively prime if the largest integer dividing them both is 1. Let us define: RELPRIME = {<x, y> : x and y are relatively prime }.

Let us look at a bad algorithm first:

1. On input <x, y>
2. Let m = min(x, y)
3. For i = 2, . . . ,m
4. Check if i divides x and y both. If it does reject.
5. Accept.

Let analyze this algorithm. First we have to compute the size of the input. We write numbers in binary. Let us say that x < y are both k-bit numbers. So the size of the input is about 2^k .

If x and y are relatively prime then the for loop will run all the way to till x. So, how many time it will be executed? Anser is: x times, but how big is x? x is at least 2^k .

So this algorithm will run the for loop 2^k times. Which is exponential in the size of the input. So it is not a polynomial time algorithm. Lets see this point more clearly.

Let x = 1000000000000000000000000000000000000000000000000
y = 1000000000000000000000000000000000000000000000000001

In this case x = 2^{50} . The size of the input counted in bits would be about 100. So, the input is not too large. But the number of iterations will be very large.

We do not have a polynomial time algorithm yet! But, you know of a polynomial time algorithm that computes the GCD. It is called the Euclidean algorithm. Lets try that

1. On input <x, y> where x and y are numbers in binary.
2. Let r := x mod y
3. If r = 0 then GCD = y
4. Else GCD = GCD(y, r).

I like this recursive version more than the iterative one!

We have to analyze this algorithm. All other steps can be performed in polynomial time in the number of bits involving x and y. A minor point first. If x < y then they are exchanged. After that we always have x ≥ y. So lets assume x ≥ y. Note that x and y are replaced by y and r.

Theorem:

$$y \leq \frac{x}{2} \text{ or } r \leq \frac{x}{2}$$

Suppose that x is m bits and y is t bits. This shows that when we replace x and y with y and r either we get

1. m − 1 bits for y and at most t bits for r . Thus a total of at most m + t − 1 bits.
2. Or t bits for y and at most m − 1 bits for r . Thus again a total of at most m + t − 1 bits.

Hence, the number of total bits in the input is reduced by 1 each time. Hence, the algorithm will take as many recursive calls as there are bits in the input.

1. if $y \leq \frac{x}{2}$ then we are done.
2. What if $y > \frac{x}{2}$. In this case, what is r?
3. r = x mod y = x − y < x − $\frac{x}{2} = \frac{x}{2}$ done again!

Let us now state a much more general theorem.

Theorem: Every context free language is a member of P. We will not prove this theorem here. What the theorem says that for every context free language L there is a polynomial time algorithm that accepts L.

Let us define NTIME(t(n)) = {L : L is decided by a O(t(n))-time NTM N}

The Class NP

Lecture No 23

NP is the non-deterministic analog of P. So, formally

$$NP = \bigcup_k NTIME\left(n^k\right)$$

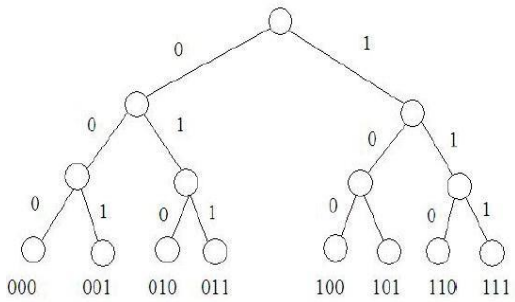
or

$$P = NTIME(n) \cup NTIME(n^2) \cup NTIME(n^3) \cup NTIME(n^4) .....$$

Let us start by looking at the following simple (and very important) non-deterministic TM. The transition function of this machine is given by

$\delta(q_0, 1) = \{(q_0, 0, R), (q_0, 1, R)\}$
$\delta(q_0, \sqcup) = \{(q_a, \sqcup, R)\}$.

Let us call this TM G (which stands for guessing). What will happen if we run this TM on input 0^n ? Since, it is a non-deterministic TM the computation is given by a computation tree. Let us look at the computation tree.



As, you can see each leaf configuration had a string of 0/1 written on the tape. This machine seems to be going through all possible 0/1 at the same time. As, if it was guessing a 0/1 string of length n. By using G as a subroutine we can make guesses. This is the power of non-determinism.

Let us now look at an example of a problem that is in NP.
Define:COMPOSITE = {<n>|n = pq for some ingeters p, q > 1}

To show that this problem in NP we have to show a non-deterministic machine that accepts all composite numbers given in binary.

Here is the TM

1. On input <n> let k be the number of bits in n.
2. Use G to guess a k-bit number p.
3. Use G to guess a k-bit number q.
4. If pq = n and p, q > 1 then accept. Else reject.

Suppose we give this machine 15 in binary. When we run G it will branch off and each branch will guess a 4 bit number. One of those branches will guess 3. So p = 3. Similarly, when we run G again it will branch off and each branch will guess a number and one of them will be 5. This particular branch will accept. Thus by definition 15 will be accepted.

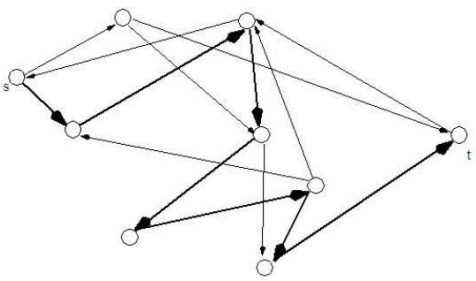
Suppose we give this machine 13 in binary. When we run G it will branch off and each branch will guess a 4 bit number. So every branch will have a p. Similarly, when we run G again it will branch off and each branch will guess a number and every branch will have a q. No branch will accept. Because, the condition 13 = pq and p, q > 1, cannot be satisfied for any p, q. Thus 13 will be rejected.

A moments thought shows that every composite will be accepted by some branch, and every prime will be accepted by all branches. Thus this NTM accepts COMPOSITES.

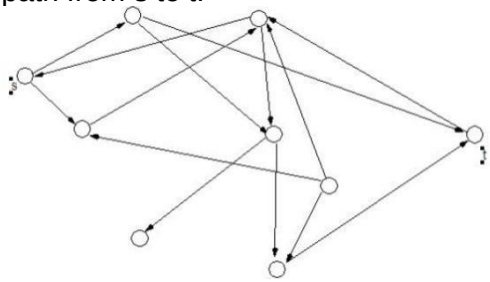
It is easy to see that the length of any branch is polynomial in the number of bits in <n>. Thus this is a non-deterministic polynomial time algorithm that accepts composites.

Let us look at another problem in NP. Let HAMPATH = {<G, s, t> : G has a hamiltonian path} We will show that this problem is in NP. Let us recall that given a digraph G. A hamiltonian path in G is a path that visits each vertex exactly once.

Here is a graph with a hamiltonian path from s to t. Show picture of a graph.



This graph does not have a hamiltonian path from s to t.



Here is a NTM that accepts HAMPATH:

1. On input <G, s, t>. Let n be the number of vertices in G.
2. Guess v1, . . . , vn a sequence of n vertices of G using the guessing machine.
3. Check if all vertices are distinct.
4. Check if v1 = s.
5. Check if vn = t.
6. For i = 1, . . . , n − 1 check if (vi , vi+1) is an edge in G.
7. If all conditions are satisfied accept else reject.

A moments thought shows that If G has a hamiltonian path from s to t then at least one branch will accept, and If G does not have a hamiltonian path from from s to t no branch will accept. Thus this NTM accepts HAMPATH. It is easy to see that the length of any branch is polynomial n. Thus this is a non-deterministic polynomial time algorithm that accepts HAMPATH.

Verifiers:

Let A be a language. A verifier for a language A is an algorithm (Turing machine) V such that
A = {w|V accepts <w, c> for some string c}.
Thus a verifier V

1. Takes two inputs <w, c>.
2. c is called the certificate or proof.
3. It accepts an <w, c> if c is a correct proof of the fact that $w \in A$.

We measure the time running time of a verifies only in terms of |w|. We say that V is a polynomial time verifier if it runs in time polynomial in |w|. If A has a polynomial time verifier then we say that it is polynomial time verifiable.

Verifiers: Composites is polynomial time verifiable.

Here is a verification algorithm.

1. On input < n, (p, q) >
2. Check if $p \leq n$ else reject.
3. Check if $q \leq n$ else reject.
4. Check if p, q > 1 else reject.
5. Check if $p \times q = n$ else reject.
6. Accept.

Note that if someone has found a factorization of a number n then it is easy for them to convince anyone that the number is composite. Thus the factorization of n is a easily verifiable proof that the number is composite.

Verification Algorithms: HAMPATH is polynomial time verifiable.

Here is a verification algorithm.

1. On input < G, s, t, (v_0, . . . , v_n) >
2. Check if $v_0 = s$.
3. Check if $v_n = t$.
4. Check if all vi "s are distinct and each vertex of G appears in the list.
5. For i = 1 to n − 1 check if (vi , vi+1) are connected with an edge of G.

Note that a hamiltonian path from s to t is a proof that <G, s, t> $\in$ HAMPATH. Thus once someone has found such a path it is very easy to verify that the path is indeed a hamiltonian path.

Polynomial time verifiers:

NP is the class of languages that have polynomial time verifiers.
NP is the class of languages that are decided by polynomial time non-deterministic Turing machies.
We want to show that these two definitions are equivalent.

For now let us define:
NP1 is the class of languages that have polynomial time verifiers.
NP2 is the class of languages that are decided by polynomial time non-deterministic Turing machines.
Let us say L $\in$ NP1 we will show that it is also in NP2.

Let L be a language in NP1 and V be a (polynomial time) verifier for L. Consider the following non-deterministic TM. Let us say V runs in time dn^k .

1. On input w of length n.
2. Guess a certificate c of length atmost dn^k.
3. Run V on <w, c>
4. if V accepts accept else reject.

This is a non-deterministic TM. It runs in time O(dn^k). Now, if w $\in$ L then there at least one certificate c such that V accepts <w, c>. Thus if w $\in$ L at least one branch of M will accept w.

If w $\notin$ L then no branch of M will accept w. This shows that L is in NP2. Now, we will show that if L is in NP2 then L is in NP1. Let L be in NP1 then there is non-deterministic TM M that accepts L. M runs in time dn^k . Let w $\in$ L. What is the way to prove that w $\in$ L. We can try to prove that M accepts w. But how?

Note that a NTM can make a choice at every step. There will be at most b possibilities for each choice. Suppose we take a string of the form 1311 $\cdot \cdot \cdot$. This string tells us to make choice number 1 in the first step. Choice number 3 in the second step and so on...

Now, let us consider the following verifier for L.

1. One input <w, c>
2. Check if |c| $\leq$ dn^k.
3. Simulate M on w by making the choices dictated by c.
4. If M accepts w accept <w, c>.

This is a deterministic verifier for L and runs in polynomial time. Now, we can remove the subscripts that we had put on these classes since we have shown they are the same.

NP is the class of languages that have polynomial time verifiers.
NP is the class of languages that are decided by polynomial time non-deterministic Turing machines.

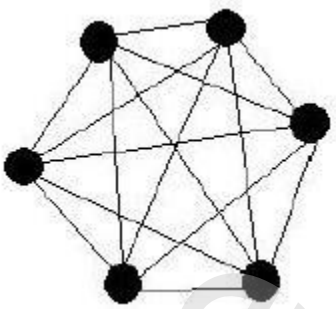
LECTURE # 23

CLIQUE Example:

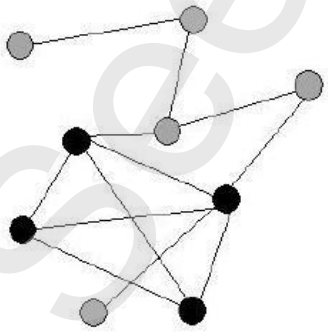
Let us define

CLIQUE = {<G, k>|G has a clique of size k}.

Here G is an undirected graph. Let us recall the definition of a clique. A clique S is a set of vertices which are all connected to each other. Here is a picture of a clique.



Here is a graph with a clique of size 4 in it. We have bolded out the clique vertices.



Is CLIQUE in NP?
You should think of making a verifier. What would be a good proof that a graph has a clique of size k?

The vertices of the clique themselves are the best possible proof.

Verfication Algorithm for CLIQUE

1. On input <G, w, S>
2. Check if the size of S is k
3. For each pair (a, b) in S there is an edge in the graph G
4. If all conditions are ture we accept otherwise reject

Let us define
NTIME(t(n)) = {L : L is decided by a O(t(n))-time NTM N}
NP is the non-deterministic analog of P. So, formally

$$NP = \bigcup_k NTIME\left(n^k\right) \qquad \text{or} \qquad NP = NTIME(n) \cup NTIME(n^2) \cup \cdots$$

Intuitively, the class NP consists of all the problems that we can solve in polynomial time if we had the power of guessing.

Let A be a language. A verifier V for A is an algorithm (Turing machine) V such that:
A = {w|V accepts <w, c> for some string c }

It is important to note that the time taken by V is measured in the length of w only. We say that V is a polynomial time verifier if on input <w, c> it takes time $O\left(|w|^k\right)$ for some k. Here c is called the certificate or the proof of w"s membership in A.

COMPOSITES Example:

Let us try to understand verifiers more closely by a simple examples. Suppose I wish to become a verifier who verifies if a given number is composite. So, I am a verifier for

COMPOSITES = {n : n is a composite number }

I can declare that I will accept only proofs or certificates of the form (p, q)
where $p \times q = n$ and $p, q > 1$.

1. If the input is (45, (5 × 9)) I will accept as $5 \times 9 = 45$.
2. If the input is (45, (15 × 3)) I will accept as $5 \times 9 = 45$.
3. If the input is (45, (7 × 9)) I will reject as $7 \times 9 \neq 45$.
4. If the input is (45, (1, 45)) I will reject as $1 \leq 1$.

Notice that in the above scheme if n is a composite number then there is always at least one way to write it as a product of two numbers that are greater than 1. Thus, for every composite number there is a proof that the number is composite.

On the other hand if n is a prime. There is no proof that n is composite. So, I will never accept any input of the form (n, (p, q)) where n is a prime.

Thus the language verified is exactly composites. Note that the verification algorithm runs in polynomial time in the size of n.

Here is the detailed version.

1. On input <n, (p, q)>
2. Check if $p \leq n$ else reject.
3. Check if $q \leq n$ else reject.
4. Check if $p, q > 1$ else reject.
5. Check if $p \times q = n$ else reject.
6. accept.

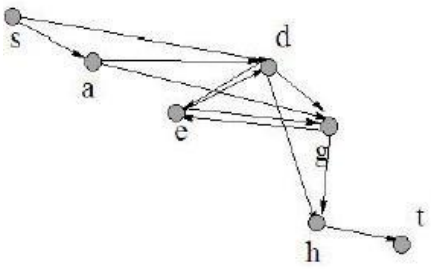
Note that if someone has found a factorization of a number n then it is easy for them to convince anyone that the number is composite. Thus the factorization of n is a easily verifiable proof that the number is composite.

Hamiltonian Path Example:

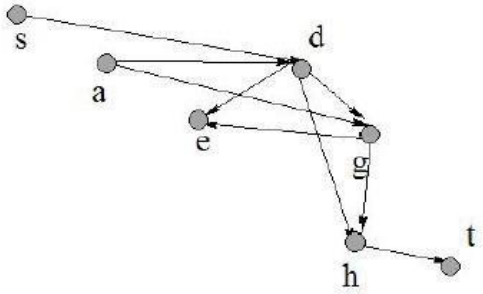
Lets look at another problem. Given a directed graph G a Hamiltonian path from s to t is a path such that:

1. It starts from s and ends at t.
2. It goes through each vertex exactly once.

Here is a graph with a Hamiltonian path
from s to t.



Path:
s, a, d, e, g, h, t
s, a, g, e, d, h, t



This graph does not have a hamiltonian path from s to t.

Let us consider the following question:

HAMPATH = {<G> : G has a hamiltonian path } Let us devise a polynomial time verification algorithm for HAMPATH. Consider the problem. We have to verify that if a graph has a hamiltonian path from s to t. What will be the simplest proof that a graph is hamiltonian. What will convince you that a given graph has a hamiltonian path. Think for a moment.

Well if someone claims that a given graph has a hamiltonian path from s to t. I will ask them to show me the path. A hamiltonian path from s to t is a clear proof or a certificate that a graph has a hamiltonian path from s to t.

Now, a verification algorithm is easy to design. The algorithm will accept inputs of the form <G, s, t, (v_0, . . . , v_n)>. It is simply check if v_0, ,v_n is indeed a hamiltonian path from s to t.

1. On input <G, s, t, (v_0, ... , v_n)>
2. Check if v_0 = s.
3. Check if v_n = t.
4. Check if all vi "s are distinct and each vertex of G appears in the list.
5. for i = 1 to n − 1 check if (vi , vi+1) are connected with an edge of G.

Clearly, this a polynomial time verification algorithm.

A language has a polynomial time verification algorithm if there is an easily (polynomial time) verifiable proof of the membership of its elements.

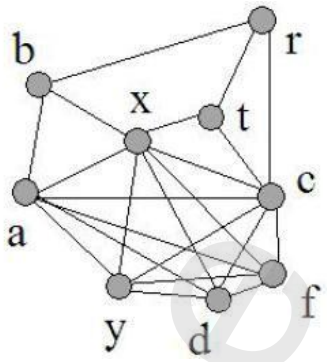
Both COMPOSITES and HAMPATH have that property.
Note that it maybe very very hard to find these proofs. But once found they are easy to verify.
Let us look at an example: 416681 = 2377 × 1735

So, (2377, 1735) is a proof that 416681 is a composite. However, it takes a while to find this proof. You can verify that the proof is correct by multiplying these two numbers in about two minutes.
Try to find a proof that 13862249 is a composite to appreciate that proofs maybe hard to find.
Similarly, it maybe hard to find a hamiltonian path (from s to t) in G. But once found it can be easily verified that it is indeed a hamiltonian path in G.

Recall that a clique in a graph is a set, C, of nodes that are all connected to eachother.

Formally, C is a clique in G if for every x, y $\in$ C with x ≠ y we have x ~ y. The clique number of a graph is the size of the largest clique in G.

Let us consider the following question:
CLIQUE = {<G, k> : G has clique number ≥ k }



Here is a graph G_0 with a clique of size 6.

Note that a, x, c, d, f , y is a clique of size 6.
There is no clique of size 7 in this graph.
The clique number of G_0 is 6.
< G_0, 6> is in CLIQUE.
< G_0, 7> is not in CLIQUE.

Let us devise a polynomial time verification algorithm for CLIQUE. Consider the problem. We have to verify that if a graph has a CLIQUE number ≥ k. What will be the simplest proof? What will convince you that a given graph has clique number ≥ k. Think for a moment.

A clique of size k itself is a clear proof or a certificate that a graph clique number ≥ k. Now, a verification algorithm is easy to design. The algorithm will accept inputs of the form <(G, k), (v0, . . . , vk)>. It simply checks if v0, . . . , vk is indeed a clique in G.

Well if someone claims that a given graph has clique number ≥ k. I will ask them to show me a clique of size k. A clique of size k is a clear proof that the clique number of a given graph is at least k.

Now, a verification algorithm is easy to design.
The algorithm will accept inputs of the form <G, k, (v0, . . . , vn)>.
It is simply check if v0, . . . , vk is indeed a clique.

1. On input <G, k, (v0, . . . , vm)>
2. Check if m ≥ k
3. Check if all vi "s are distinct and each is a vertex of G
4. for i = 1 to m
5. for j = i + 1 to m check if (vi , vj) are connected with an edge of G.

Clearly, this a polynomial time verification algorithm.

<u>SUBSETSUM Example:</u>

Let us consider a set of numbers {1, 3, 6}. This set has 2^3 subsets. The elements in the subsets sum to different numbers.

1. {} sums to 0.
2. {1} sums to 1.
3. {3} sums to 3.
4. {6} sums to 6
5. {1, 3} sums to 4
6. {1, 6} sums to 7
7. {3, 6} sums to 9
8. {1, 3, 6} sums to 10.

Note that there is no subset that sums to 5.

In the subset sum problem, we are given a set S of numbers and a target value t. The question is if there is T $\subseteq$ S such that $\sum_{a \in T} a = t$. We are asked if there is a subset of S whose elements sum to t? Formally,

SUBSETSUM = {(S, t) : There is T $\subset$ S such that $\sum_{a \in T} a = t$ }

Given a set S it has potential 2^n subsets. Thus it does not seem to be easy to tell if a given number will appear as a subset sum.

Let us devise a polynomial time verification algorithm for SUBSETSUM. Consider the problem. We have to verify that if a S has a subset T that adds up to t.

What will be the simplest proof of this. What will convince you that there is such a subset. Think for a moment.

The subset itself will be a proof! Now, a verification algorithm is easy to design. The algorithm will accept inputs of the form <S, t,T> It simply checks if T is a subset of S and adds its elements to see if they add up to t.

On input

1. <S, t,T>
2. Check if T is a subset of S if not reject.
3. Add all the elements of T. If they sum to t accept.

Clearly, this a polynomial time verification algorithm.

Let us come back to the class NP which was defined using non-deterministic turing machines. Machines that have the “magical” power of guessing.

Now, we are talking about “verification” where the proof is provided to us for free. These two notions are very close. In fact a non-deterministic machine can guess a ”proof”.

Theorem:
If A is accepted by a polynomial time verifier then A is also accepted by a polynomial time non-deterministic Turing machine.

To prove this theorem let us take a language A that is verified by a polynomial time verifier V. V runs in time n^k for some k. Let us make a non-deterministic TM for it. Consider the following non-deterministic TM N.

1. On input w of length n.
2. Non-deterministically guess a string c of length n^k.
3. Run V on <w, c> if V accepts accept.

Clearly if w ∈ A then there is a c such that V accepts <w, c> so at least one branch of N will accept. On the other hand if w ∉ A then no branch of V will accept. Clearly, this Turing machine runs in non-deterministic polytime. Hence, A ∈ NP

Another example:

As an example we can look at the set {3, 5, 7} It has the following subsets and subset sums:

1. {} sums to 0
2. {3} sums to 3
3. {5} sums to 3
4. {7} sums to 3
5. {3, 5} sums to 8
6. {3, 7} sums to 10
7. {5, 7} sums to 12
8. {3, 5, 7} sums to 15

Formally the subset sum problem is the following:
SUBSETSUM = {<S, t>: There is a subset of S that sums to t }.

We note that the simplest proof that S has a subset that sums to t is the subset itself. Thus the following is a polynomial time verification algorithm:

1. On input <S, t, T>.
2. Check if T is a subset of S.
3. If all the elements of T sum to t accept.
4. Else reject.

Once again notice that we are only saying that there is an easily verifiable certificate that an instance <S, t> is in SUBSETSUM. We are not saying anything about how hard it is to find such a proof.

Alternate Characterization of NP

Theorem:
A language, A, is in NP if and only if there exists a polynomial time verification algorithm for A.

Thus we get another characterization of NP. NP is the class of easily verifiable problems. Let us start by showing:

Theorem:
If A has a polynomial time verification algorithm then A is in NP.

Proof. Since A has an polynomial time verification algorithm let us call that algorithm V. We want to show that A is accepted by some non-deterministic Turing machine, N, that runs in non-deterministic polynomial time.

The machine N will use V as a subroutine. Suppose V runs in time n^k .
Here is a description of N:

1. On input w of length n.
2. Guess a certificate c of size at most n^k.
3. Run V on <w, c>.
4. If V accepts accept else reject.

Let us note that:

1. If w ∈ A then there is some c such that V accepts <w, c>.
2. Since, V runs in time O(n^k) hence such a c can not have length more than O(n^k).
3. Thus there is at least one branch of N that accepts w. It is the branch that correctly guesses c.

On the other hand:

1. If w ∉ A then for all c V rejects <w, c>.
2. Thus all branches of N reject w.

Hence L(N) = A. It is easy to see that N runs in polynomial time since V runs in polynomial time. The whole proof can be summarized in one line:

"We can guess the certificate using the power of non-determinism".

Now, we have to show the other side.

Theorem:
If A is in NP then it has a polynomial time verification algorithm.

Proof. Since A is in NP it is accepted by some non-deterministic Turing machine, N, that runs in non-deterministic polynomial time. We want to come up with a verification algorithm for A. We do not know much about A. All we know is that it is accepted by some non-deterministic TM.

What would be a good proof that w ∈ A? The only way to convince someone that w ∈ A is to convince them that N accepts w. But N is a non-deterministic TM. How can we convince them in polynomial time that N accepts w.

The main idea is that if we give the branch that accepts w then that is a easily verifiable proof that w is accepted by N and therefore it is a proof that w ∈ A.

Formally, the verification algorithm is the following:

1. On input <w, c>.
2. Here c specifies the branch of N.
3. Simulate N on w by taking the branching given by c.
4. If N accepts accept else reject.

Note that if w ∈ A

1. Then N accepts w. Therefore, there is at least one branch of N that accepts w.
2. Hence, we can give the specification of this particular branch to V as c. Then V will simulate N along this branch only and will accept.

If w ∉ A then

1. Then N does not accepts w. Therefore, there all branches of N reject w.
2. Hence, the specification of all branches will make V reject.

It is clear that V runs in polynomial time.

Now we have two characterization of NP

1. NP is the non-deterministic analog of P. It is the set of languages accepted by non-deterministic Turing machines that run in Non-deterministic polynomial time.
2. NP is the set of languages which have polynomial time verification algorithms.

The P versus NP question

A major question in complexity theory: Is P = NP?

The question asks if all easily verifiable languages can be solved in polynomial time?

The clay mathematical institute has offered a prize of $1, 000, 000 for a solution to this problem. This is considered to be one of the most important open questions in computer science and logic.

Polynomial time reducibility

A language A is said to be polynomial time mapping reducible to B if there is a function f:$\sum^* \rightarrow \sum^*$ such that

1. $x \in A$ if and only if $f(x) \in B$.
2. f can be computed by a Turing machine that runs in polynomial time.

This notion of reducibility is also measuring time. Now, we are saying that we should be able to compute the reducibility quickly. If A is polynomial time mapping reducible to B we write $A \leq_p B$.

Theorem:

If $B \in P$ and $A \leq_p B$ then $A \in P$.

Let us carefully prove this theorem. Since, $A \leq_p B$ hence there is a function $f : \sum^* \rightarrow \sum^*$ such that

1. $x \in A$ if and only if $f(x) \in B$.
2. f can be computed by a Turing machine, F, that runs in polynomial time. Let the running time of F be $O(n^k)$.

Furthermore, since $B \in P$ hence there is TM M that accepts B in time $O(n^l)$. Let us make a TM that accepts A in polynomial time.

1. On input x
2. Compute y = f (x) using F.
3. Run M on y. If M accepts accept if M rejects reject.

Clearly, this TM accepts A. Let us find out how much time does it take.

Let |x| = n. Then

1. Computing y = f (x) using F takes time $O(n^k)$.
2. Furthermore, $|y| \leq O(n^k)$.
3. Running M on y takes time $O(|y|^l)$.

Total time is $O(n^k) + O(|y|^l) \leq O(n^k) + O((n^k)^l) = O(n^{kl})$.

Thus this algorithm runs in polynomial time and hence A $\in$ P.

NP completeness

A language is A called NP-hard if every language L $\in$ NP. $L \leq_p A$

Notice that a NP-hard language in some sense harder than all the languages in NP.

Definition:

A language A is called NP-complete if

1. A is NP-hard.
2. A is in NP.

It is not at all clear that there are NP-complete languages. In fact, if they exist it would be an amazing fact. These languages will have "all the complexity" of the class NP. They will be extremely important because of the following theorem.

Theorem:

If A is NP-complete and A is in NP then P = NP.

So in order to solve P versus NP problems we will have two approaches. Suppose we find a NP-complete problem A. A is the hardest problem in NP. So, either we can

1. Try to prove that A is not in P. In fact, NP-complete problems are the best candidates for this.
2. Try to find a polynomial time algorithm for A. If we succeed then we will show that P = NP.

No one has been able to do any of the above!

Satisfiability

Let us see what a boolean formula is. Here is an example: $\varnothing = (x_1 \vee x_3) \wedge \overline{(x_4 \vee (\overline{x_2} \wedge x_1))}$

A boolean formula is defined as follows. Let us take variable {x1, . . . , xn}.

1. 0, 1, xi are boolean formulae.
2. If $\varnothing$ and ψ are boolean formulae then so are $\varnothing \wedge \psi$, $\varnothing \vee \psi$ and $\overline{\varnothing}$

An assignment of τ : {x1, . . . , xn} → {0, 1}. Each assignment evaluates a formula to 0 or 1 in the natural way. We substitute the variable with 0 or 1"s and evaluate the formula by interpreting $\vee$, $\wedge$ and ¬ as usual.

$$\varnothing = (x_1 \vee x_3) \wedge \overline{(x_4 \vee (\overline{x_2} \wedge x_1))}$$

Suppose the assignment is τ(x1) = τ(x2) = τ(x4) = 0, τ(x3) = 1then the formula becomes

$$\varnothing = (0 \vee 1) \wedge \overline{(\overline{0} \vee (0 \wedge 0))}$$

Which evaluates to 0. We denote the this by $\varnothing \mid \tau$

Definition: A formula $\varnothing$ is satisfiable if there exists an assignment τ such that $\varnothing \mid \tau = 1$

Note that if there is a formula with n variables there are potential 2^n assignments. A formula is satisfiable if there is at least one assignment that makes it 1 (or true).

Let us define SAT = {< $\varnothing$ > : $\varnothing$ is satisfiable}. It is easy to see that

Theorem: SAT is in NP

We can give an easy verification algorithm. The algorithm takes a formula $\varnothing$ and assignment τ and checks if τ satisfies $\varnothing$.

1. On input <$\varnothing$, τ>.
2. Substitute the variables using of $\varnothing$ using τ.
3. Evaluate the resulting formula with constants.
4. If the formula evaluates to 1 accept.

This is a verification algorithm for SAT.

Theorem: (Cook Levin): SAT is NP-complete.

Theorem: (Cook Levin): P = NP if and only if SAT$\in$P.

In order to understand the Cook-Levin theorem we have to understand what can be expressed in a boolean formula. We will do this by first reducing one problem in NP to Satisfiability.

This will give us some feel for satisifiablilty and then we can go on to the proof of the Cook-Levin theorem.

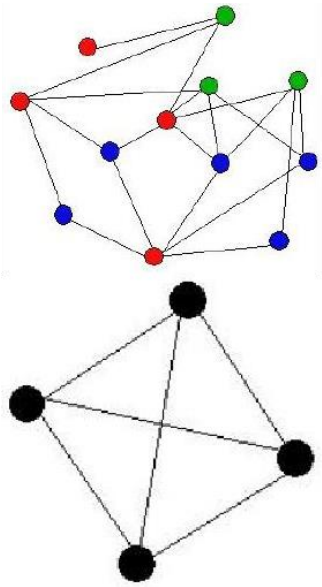
3-Color:

Let G = (V,E) be a graph. We say that G is 3-colorable if we can assign there colors (say red, blue and green) to its vertices in such a way that:

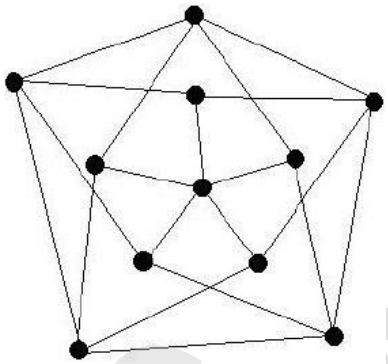
Each vertex is assigned exactly one color.
If {x, y} is an edge then the color of x is different from color of y.
Each vertex has been colored and you can check to see if it is a proper coloring.

Here is a picture of a graph that is not 3-colorable.

Here is a picture of a graph that is 3-colorable.



It is easy to see that this graph is not three colorable. Since, it has four vertices all connected to each other therefore we need at least four colors to color it properly.



Here is a picture of a graph that is not 3-colorable.

It is not easy to see that we cannot properly color it by three colors. You can exhaustively check at home that this is not a three colorable graph.

Let us now define: 3COLOR = {<G>: G is three colorable}.

It is your homework to prove that 3Color is in NP.

Theorem: 3Color ≤p SAT.

Lets first try to understand what this means! We do not want to do any of the following:

1. Design an algorithm to solve SAT.
2. Design an algorithm to solve 3Color.

We want to know the relationship between 3Color and SAT.

What we want is a a polynomial time algorithm A that will take as in input a graph G and output a Boolean formula Ǿ such that G is 3 colorable iff Ǿ is satisfiable

What we really want to do is to come up with a Translation Algorithm that translates the problem of 3-colorability to Satisfiability. In fact given a graph G we want to be able to come up with a formula Ǿ that is true if and only if the graph is 3-colorable. For that let us see what the formula Ǿ will have to "say".

The formula Ǿ will simply say

1. It is possible to assign each vertex exactly one color from red, blue and green.
2. Furthermore, if two vertices are connected then they get different colors.

Let us look at the following simple formula.

$$(x1 \vee x2)$$

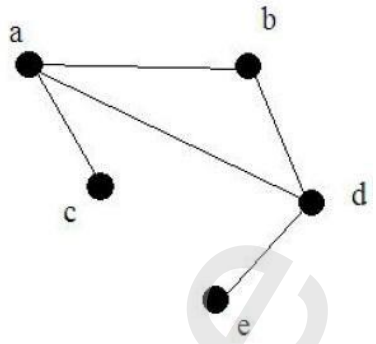
This formula is true if and only if x_1 or x_2 is true. Now, let us try to come up with a formula that is true if exactly one of the variables out of x_1 and x_2 is set to true.

A little thought shows that the formula is $(x_1 \vee x_2) \wedge \overline{(x_1 \wedge x_2)}$

Now, let us go a bit further. Let us come up with a formula which is true if and only if exactly one out of the there variables x_1, x_2, x_3 is true. A little thought shows that that formula is

$$(x_1 \vee x_2 \vee x_3) \wedge \overline{(x_1 \wedge x_2)} \wedge \overline{(x_1 \wedge x_3)} \wedge \overline{(x_2 \wedge x_3)}$$

Now, suppose we have a graph G = (V,E). As an example let us take the following graph
V = {a, b, c, d, e}.
E = {{a, b}, {a, c}, {a, d}, {b, d}, {d, e}}
Here is a picture of the graph.



Suppose I have three variables for the first vertex a. Let us say they are Ra,Ba,Ga.
They have the following interpretation:

1. Ra = true means a is colored red.
2. Ba = true means a is colored blue.
3. Ga = true means a is colored green.

How can we come up with a formula that is true if and only if the vertex a is assigned exactly one color from these three colors? Easy remember

$$(x_1 \vee x_2 \vee x_3) \wedge \overline{(x_1 \wedge x_2)} \wedge \overline{(x_1 \wedge x_3)} \wedge \overline{(x_2 \wedge x_3)}$$

So the formula is

$$(R_a \vee B_a \vee G_a) \wedge \overline{(R_a \wedge B_a)} \wedge \overline{(R_a \wedge G_a)} \wedge \overline{(B_a \wedge G_a)}$$

Similarly

$$(R_b \vee B_b \vee G_b) \wedge \overline{(R_b \wedge B_b)} \wedge \overline{(R_b \wedge G_b)} \wedge \overline{(B_b \wedge G_b)}$$

is true if and only if b is assigned exactly one color. So it is easy to find a formula that is true if and only if a given vertex is assigned one out of there colors.

Now, let us consider an edge e = {a, b}. How can we find a formula that is true if a and b are given different colors. Well that is easy

$$\overline{(R_a \wedge B_b)} \wedge \overline{(R_a \wedge B_b)} \wedge \overline{(G_a \wedge G_b)}$$

So we have learnt two things.

1. Given a vertex v we can make three variables Rv ,Bv ,Gv and make a formula that says v is assigned exactly one of the three colors, red, blue or green.
2. Given an edge {u,w} we can come up with formula that is true (or says) that u and w must have different colors.

What if we put them together. See what we get:

Let G = (V,E) be a graph and Consider the following formula:

$$\varnothing_G = \left(\bigwedge_{v \in V} (R_v \vee B_v \vee G_v) \wedge \overline{(R_v \wedge B_v)} \wedge \overline{(R_v \wedge G_v)} \wedge \overline{(B_v \wedge G_v)} \right)$$

$$\left(\bigwedge_{\{a,b\} \in E} \left(\overline{R_a \wedge R_b} \right) \wedge \left(\overline{B_a \wedge B_b} \right) \wedge \left(\overline{G_a \wedge G_b} \right) \right)$$

This formula "says" that G is three colorable. It is satisfiable true if and only if we can find a 3Coloring of G.

Consider the algorithm:

1. On input G.
2. Compute ØG and output it.

This is a polynomial-time reducibility from 3Color to SAT.

Note that we have not done the following:

1. Said any thing about how hard is it to solve a SAT.
2. Said anything about how hard it is to solve 3Color.

We have shown that in polynomial time we can reduce 3Color to SAT.

Let us look at the little theorem we have proved.

Theorem:
3Color ≤p SAT

Let us compare it with the Cook-Levin Theorem.

Theorem:
For every language A ∈ NP.

A ≤p SAT.

Thus the Cook-Levin theorem is much more general. It is talking about an infinite class of problems that is reducible to SAT.

To prove that theorem we need to understand satisfiability, non-deterministic turing machines and reducibility in a more detail.

Our goal now is to prove the Cook-Levin Theorem in its complete general form. What do we have to show? We want to show that if L ∈ NP then L ≤p SAT.

Let us start with a language L such that L ∈ NP.

What do we know about L? The only thing we know about L is that it is accepted by a non-deterministic Turing machine M in non-deterministic polynomial time.

Let us record these facts. Let M be a non-deterministic Turing machine that accepts L in time n^k

What do we want to do now?

We want to come up with a reduction. This reduction will take an instance x and convert it into a formula Ǿx such that x ∈ L if and only if Ǿx is satisfiable.

Note that if x ∈ L then at least one computation branch of M accepts x. Let us write this computation in a table. The first row of this table consist of the initial configuration.

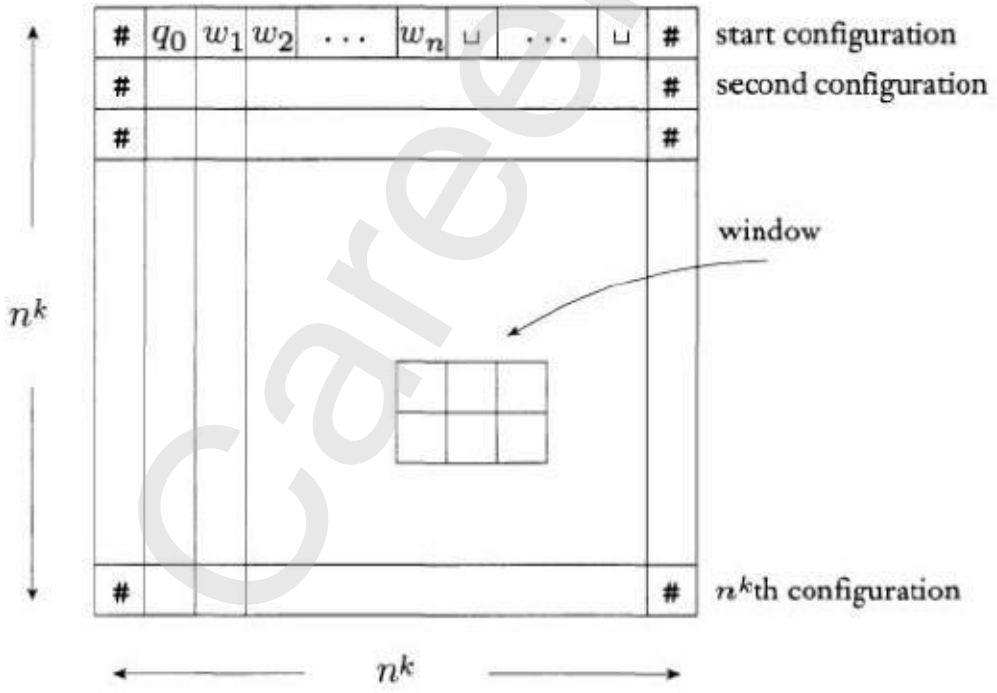
The second row of this table consist of the second configuration and so on. How does this table look like?

Let us make some observations first:

1. This table will have at most n^k rows. Since the machine accepts in time n^k .
2. The machine will scan only n^k tape symbols so we only need to keep n^k columns.

This tableau will be an $n^k \times n^k$ tableau. The tableau is a proof of the fact that M accepts x. Thus it is a proof of the fact that x ∈ L.

Let us look at a schematic diagram of this tableau.



Let us look at a schematic diagram of this tableau. If anyone has this tableau they can confirm that x∈L by checking the following.

1. Check if the first configuration is an initial configuration.
2. Check if the last configuration is an accepting configuration.
3. Check the i-th configuration yields the i + 1-st configuration.

Thus x∈L if an only if we can make a tableau with the above properties. Let us think a bit about the tableau. Each cell in the tableau has a symbol from C = Q ∪ Γ ∪ { # }

Can we make a formula which expresses the fact that each cell has exactly one symbol from C in it. The tableau is made out of cells let cell [i, j] denote the i, j-th cell. We will make variables xi,j,s for each cell where s ∈ C. If cell [i, j] = s then we can set the variable xi,j,s = true. For example if the i, j-th cell has an a then xi,j,a = ture.

Let us now look at the following formula:

$$\left(\bigvee_{s\in C} x_{i,j,s}\right)\wedge\left(\bigwedge_{s,t\in C, s\neq t}\left(\overline{x_{i,j,s}}\vee\overline{x_{i,j,t}}\right)\right)$$

The above formula says that the i, j-th cell has exactly one symbol from C in it. Let us review why this is the case.

Let us now look at the following formula:

$$\phi_{cell}=\bigwedge_{1\le i,\, j\le n^k}\left[\left(\bigvee_{s\in C} x_{i,j,s}\right)\wedge\left(\bigwedge_{s,t\in C, s\neq t}\left(\overline{x_{i,j,s}}\vee\overline{x_{i,j,t}}\right)\right)\right]$$

Let x = x1x2 . . . xn be an input. How can we say that the first row corresponds to the initial configuration? This is not that difficult. Let us consider the formula

$$\phi_{start}=x_{1,1,\#}\wedge x_{1,2,q_0}\wedge x_{1,3,x_1}\wedge x_{1,4,x_2}\wedge x_{1,5,x_3}\wedge\ldots.\wedge x_{1,n+2,x_n}\wedge x_{1,n+3,\square}\wedge x_{1,n+4,\square}\ldots.x_{1,n-1,\square}\wedge x_{1,n,\#}$$

This formula says that the first row of the tablaue is the initial configuration of M on the input x = x1 · · · xn.

Lets now look at the following formula: $\phi_{accept}=\bigvee_{1\le i,\, j,\le n^k} x_{i,j,q_{accept}}$

This formula says that the accepting state appears in the tableau.

Lastly, we want a formula ϕ_{move} this formula will say that each of the row comes from the previous row of the table by a legal move of the Turing machine M.

This is the most important part of the construction and the proof. How do we say that a move is legal. For this we have to go back to the tableau and come up with a concept of a legal window.

Consider a window of size 2 × 3 in the tableau. A window is called legal if it does not violate the actions specified by M"s transition function.

Let us look at some examples of legal windows. Suppose we have the following moves in the machine M.

$\delta(q_1, a) = \{(q_1, b, R)\}$
$\delta(q_1, b) = \{q_2, c, L), (q_2, a, R)\}$

Then the following examples are of legal windows.

a q_1 b q_2 a c	This is because the machine reading b can move left and write a c.
a q_1 b a a q_2	This is because in q1 machine reading a b can replace it with an a and move right.
a a q_1 a a b	The only information we have is that the machine is in q1 and reading something. So, it can move right.
a b a a b q_2	The head maybe too far to make a change in this part of the tape.
a b a a b q_2	The head may have moved to the left.
b b b c b b	The head may be on the left and may have changed the b to a c and moved to the left.

Now let us look at some illegal windows to get some more intuition.

a b a a a a	A symbol cannot change unless the head is near it.
a q_1 b q_1 a a	This move is not allowed by the transition function of M.
b q1 b q_2 a q_2	Wrong way to have a configuration. The head cannot become two heads!

We now want to be able to do two things.

1. First we will show that if each window in a tableau is legal then every configuration legally follows from the previous one in the table.
2. Then we will again come up with a formula that says that each window is legal.

In the end we will combine this formula with Ócell, Óstart and Óaccept to get a formula.

We want to show that if L $\in$ NP then L $\leq_p$ SAT.The idea is that if L is in NP then L is accepted by an NTM M in polynomial time.

What do we want:
We want a to show that there is a function f that we can compute in polynomial time such that

1. x $\in$ L then f (x) $\in$ SAT. We do this by creating a formula Ó$_x$ such that:
2. Ó$_x$ is satisfiable if and only if M accepts x.

What do we want:
We want a to show that there is a function f that we can compute in polynomial time such that

1. x $\in$ L then f (x) $\in$ SAT.

We do this by creating a formula Ó$_x$ such that:

1. Ó$_x$ is satisfiable if and only if M accepts x.
2. if M accepts x, we can make a tableau that shows an accepting computation of M on x.
3. This tableau will be an $n^k \times n^k$ tableau since M runs in polynomial time.
4. The formula Ó$_x$ is going to be satisfiable if and only if such a tableau exists.

The main idea is to make several variables: $x_{i,j,s}$
If the tableau contains a symbol s in the (i, j) place then we will set the variable $x_{i,j,s}$ = ture.

The tableau has several properties:

1. Each entry of the tableau has exactly one entry.
2. The first computation is the initial computation of M on x.
3. The tablature corresponds to an accepting computation.
4. Each configuration row is obtained from the previous row through the yields relation.

We try to express these properties in propositional logic.

1. Each entry of the tableau has exactly one entry. we used the formula $Ø_{cell}$ to express this.
2. The first computation is the initial computation of M on x. We used the formula $Ø_{start}$ to express this.
3. The tableau corresponds to an accepting computation. We used the formula $Ø_{accept}$ to express this.
4. Each configuration row is obtained from the previous row through the yields relation. We will use the formula $Ø_{move}$

Previousely we looked closely at $Ø_{cell}$, $Ø_{start}$ and $Ø_{accept}$. Now we want to concentrate on $Ø_{move}$.

Next was the concept of legal windows. We said a 2 × 3 window is legal if it can appear in a possible computation of the machine M. As an example, let us take a machine with
δ $(q_0, a) = \{(q_2, b,R)\}$ and
$\delta(q_0, b) = \{(q_1, a,R), (q_2, c, L)\}$.

Let us look at some legal windows now:

#	a	b
#	a	b

b	a	b
b	a	b

q_0	a	b
b	q_2	b

q_0	b	b
b	c	b

Let us look at some illegal windows.

#	a	b
#	a	#

b	a	b
b	c	b

q_0	a	b
b	q_1	b

q_0	b	b
c	q_2	b

A very critical observations are the following:

1. By observing the transition function of M we can compute the set of all legal windows.
2. Note that this has to be done once in for all. It need not be recomputed for each x. (Although it is not hard to compute also but we do not need to do that!)
3. If we have a tableau that has the first configuration which is the initial configuration and each window is legal then tableau corresponds to a computation of M on x

Let us discuss these points one by one.

By observing the transition function of M we can compute the set of all legal windows.

This is not very hard to see. In your homework, I have given you a NTM and asked you to compute all the possible legal windows. Let us nevertheless outline a simple (but time consuming method).

1. Note that these are windows of size 2 × 3. Thus they have 6 entries. The table can let us say have r possible entries.
2. Thus the total number of possible windows is r^6. Hence there are only a finite number of them.

Thus we can go through every possible window and figure out if it is legal or not.

1. For each possible window W
2. If W is legal output W.

The above algorithm will compute all legal windows. Provided we can test if a window is legal. That is not difficult to do!

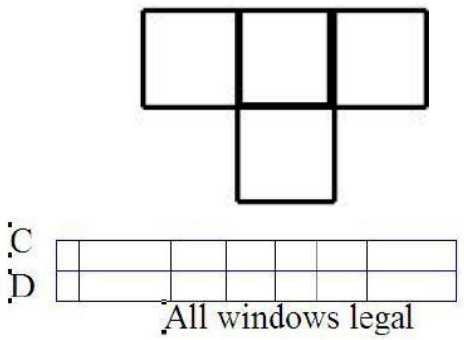
Let the output of this algorithm be L the set of all legal windows.

Note that this has to be done once in for all. It need not be recomputed for each x. (Although it is not hard to compute also but we do not need to do that!).

Note that the legality of the window does not depend on the input to the machine M. It only depends on the transition table of M. Thus the set of all legal windows can be computed once in for all.

Lastly, note that if all windows in a tableau are legal then it corresponds to a computation of M. In fact, the definition of a legal window is designed to enforce this. This is because a 2 × 3 legal window ensure that the middle element in the next configuration comes from a legal move of the NTM M.

Let us look at the following figure:

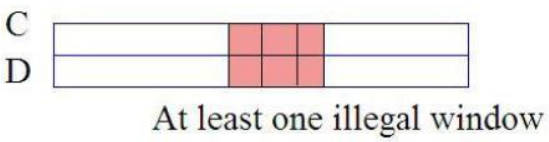


For this we need to do two exercises.
Let C and D be two configurations such that $C \vdash_M D$

Write C and D as two rows in the tableau and prove that all the 2 × 3 windows are legal in that part of the tableau.

Let C and D be two configurations such that C does not yield D

Write C and D as two rows in the tableau and prove that at least one of the 2 × 3 windows are is illegal in that part of the tableau. This is because you will have to cheat somewhere. And at that point you will have to create an illegal window.



Now we are in a position to start constructing $Ø_{cell}$. Let us go one step at a time. Let us take all the legal windows. L = {W_1,W_2, . . . ,W_t}. What we want to say is that the first window is legal. We can say that by saying....

1. The first window is W_1 or
2. The first window is W_2 or
3. The first window is W_3 or

and so on....

Let us do this by an example. Let us say that W_1 is

#	q_0	a
#	b	q_2

Then we can simply write: $x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,a} \wedge x_{2,1,\#} \wedge x_{2,2,b} \wedge x_{2,3,q_2}$

Similarly if I want to say that the (i, j)-th window is equal to W_1 I can write
$x_{i,j,\#} \wedge x_{i,j+1,q_0} \wedge x_{i,j+2,a} \wedge x_{i+1,j,\#} \wedge x_{i+1,j+1,b} \wedge x_{i+1,j+2,q_2}$

Let us denote $\mu_{i,j,l}$ to be the formula that says that the (i, j)-th window is equal to W_l. Then to say that the (i, j)-th window is legal all I have to do is to say

$$\bigvee_{l=1}^{t} \mu(i, j, l)$$

This says that the (i, j)-th window is either W_1 or W_2 or and so on. Which is equivalent to saying that the (i, j) window is legal.
So what would the following formula express:

$$\bigwedge_{i,j} \bigvee_{l=1}^{t} \mu(i, j, l)$$

This says that all windows are legal. This is our $Ø_{move}$.

Now, let us look at our grand boolean formula that we have created.
$Ø_x = Ø_{cell} \wedge Ø_{start} \wedge Ø_{accept} \wedge Ø_{move}$

This formula is satisfiable if and only if all four components are satisfiable. Let us look at them one by one:

1. $Ø_{cell}$ is satisfiable if and only if we have an assignment of the variables that corresponds to putting one symbol in each entry of the tableau. Thus the tableau is properly filled.
2. $Ø_{cell}$ is satisfied if and only if the variable assignment corresponds to filing the first row with the initial configuration of M on x.
3. $Ø_{accept}$ is satisfied if the variable assignment corresponds to having the accept sate somewhere in the tableau. That means the tableau encodes an accepting computation.
4. Lastly, and most importantly $Ø_{move}$ is satisfied if the assignment corresponds to filling rows in such a way that they follow from legal moves of the machine.

In summary $Ø_x$ is satisfiable if and only if there exists a tableau with these properties

To finish the proof of Cook-Levin Theorem let us consider the following algorithm:

1. On input x
2. Compute $Ø_x$ and output it.

This algorithm computes a function in polynomial time. Our discussion shows that this is a polynomial time reducibility from L to SAT. This shows that L $\leq_p$ SAT.

The proof of Cook-Levin Theorem is therefore complete.

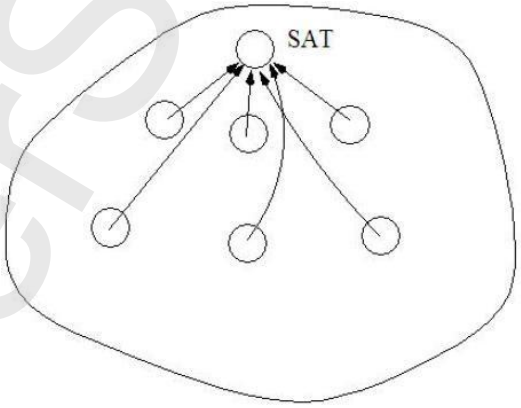
Recently we finished the proof of the celebrated Cook-Levin theorem which states that:

Definition:
SAT is NP-complete.

This means

1. SAT is in NP.
2. Every language (problem) in NP is polynomial time reducible to SAT.

One way to view this theorem is by the following picture:



An immediate question that comes to mind is the following?

Question:
Is SAT the only NP-complete problem?

Let us think about this question for a little while.

Let us for the sake of argument assume that there is another language let us call it L that is also NP-complete. What properties it will have:

1. It has to be in NP. Suppose we can establish this by giving a verification algorithm.
2. All languages $L^{/} \in$ NP will be polynomial time reducible to L.

In particular since SAT $\in$ NP hence we must also have SAT $\leq_p$ L.

Now suppose that we have a language L such that L is in NP. We suspect that L is NP-complete. If that is the case then SAT $\leq_p$ L.

Suppose we can establish the fact that SAT $\leq_p$ L. (1)

Then since SAT is itself NP-complete therefore for any language
$L^{/} \in$ NP we have $L^{/} \leq_p$ SAT (2)

If we combine (1) and (2) we get: $L^{/} \leq_p$ L there by showing that L is NP-complete.

Hence in order to show that L is NP-complete we only have to do two things.

1. Give a polynomial time verification algorithm for L. Thereby showing that L is in NP.
2. Show that SAT $\leq_p$ L. Thereby showing indirectly that every language $L' \in$ NP is polynomial time reducible to L.

Note that showing SAT $\leq_L$ is much easier than showing that all languages $L' \in$NP reduce to L in polynomial time.

Thus we will use Cook-Levin theorem to establish the NP-completeness of L. For this plan to work we need to recall the following theorem:

Definition:
If A $\leq_p$ B and B $\leq_p$ C then A $\leq_p$ C that is $\leq_p$ is a transitive relation.

The proof of this fact is so simple that we can revise it in a few minutes.

Suppose A $\leq_p$ B then there is a function f : $\sum^* \rightarrow \sum^*$ such that

1. x $\in$ A if and only if f(x) $\in$ B.
2. f can be computed in time $O(n^k)$ for some fixed k.

Similarly since B $\leq_p$ C then there is a function g : $\sum^* \rightarrow \sum^*$ such that

1. y $\in$ B if and only if g(y) $\in$ C.
2. g can be computed in time $O(n^l)$ for some fixed l.

To see that A $\leq_p$ C consider the following algorithm:

1. On input x
2. Compute y = f (x)
3. Compute z = g(y).
4. Output z.

This algorithm computes a function h(x) = g(f (x)) and it is clear that

1. x$\in$A if and only if f(x)$\in$B if and only if g(f (x))$\in$C. Therefore, x$\in$A if and only if h(x)$\in$C.
2. Can we compute h in polynomial time?

How much time does it take to compute h. Let |x| = n. Then it takes $O(n^k)$ time to compute y. and $O(|y|^l)$ time to compute z. Thus the total time is $O(n^k) + O(|y|^l)$.

Now, we realize $|y| \leq O(n^k)$ therefore, the total time is $O(n^k) + O((n^k)^l) = O(n^{kl})$

This theorem gives us the following plan to prove NP-completeness. Suppose we have a problem K that is known to be NP-complete.

Take SAT for example. Now, we have another problem W that we want to prove is NP-complete.

We can proceed as follows:

1. Show that W is in NP (This is usually easy) by giving a verification algorithm.
2. Show that K $\leq_p$ W. This shows that all languages in NP are reducible to W.

Note that we must show that K $\leq_p$ W and not the other way around. This will be our basic recipe for showing that other problems are NP-complete. Let us start with one example.

Let us start by defining a problem that we will prove is NP-complete. This problem is going to be very closely related to SAT. We will call it 3SAT. Let us say we have boolean variables $x_1, \ldots, x_n$.

We call a variable or it"s negation a literal. Now suppose we have a lot of literals that are being Or-ed together we will call that a clause. Here are examples of clauses:

$$x_1 \vee x_5 \vee \overline{x_7} \qquad \overline{x_1} \vee x_4 \vee \overline{x_6} \vee x_8 \qquad \overline{x_3} \vee \overline{x_5} \vee \overline{x_{11}}$$

A formula in conjunctive normal form (CNF) is just an and of clauses. So it looks like this:

$$\left(x_1 \vee x_5 \vee \overline{x_7}\right) \wedge \left(\overline{x_1} \vee x_4 \vee \overline{x_6} \vee x_8\right) \wedge \left(x_3 \vee \overline{x_5} \vee \overline{x_{11}}\right)$$

We will say that a formula is in 3CNF if all the clauses have exactly three literals. Let us define the following language:

3SAT = { Ø : Ø is in 3CNF and Ø is satisfiable}.

We want to show that 3SAT is NP-complete.

1. We have to show that 3SAT is in NP. That is easy and is left as a homework exercise.
2. We will show that SAT $\leq_p$ 3SAT.

Let Ø be a formula. We will show how to construct a formula in 3CNF such that

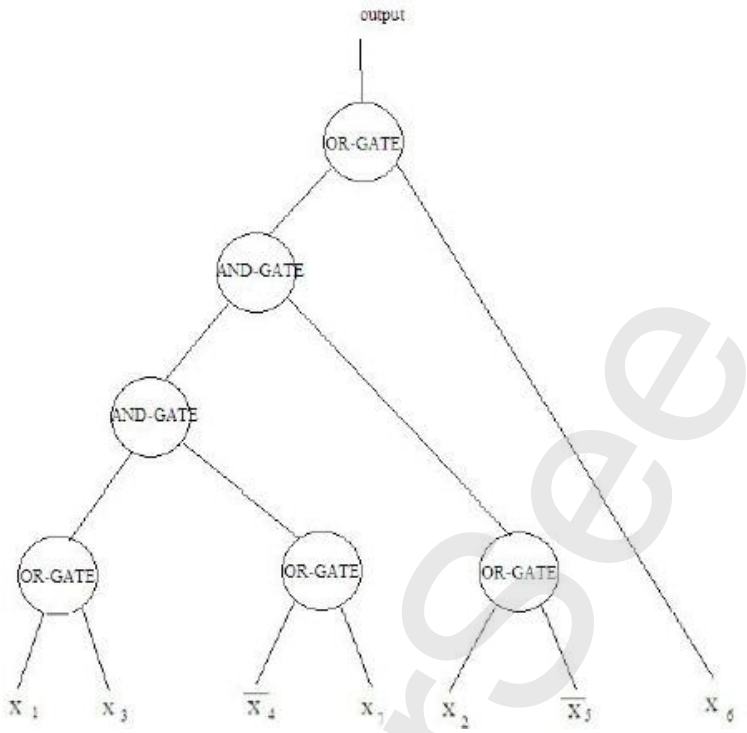
1. Ψ is in 3CNF.
2. Ø is satisfiable if and only if Ψ is satisfiable.
3. Ψ can be computed from Ø in polynomial time.

Let us illustrate the proof of this theorem by an example. I will leave the details to you. Let us say we have a boolean formula Ø given by

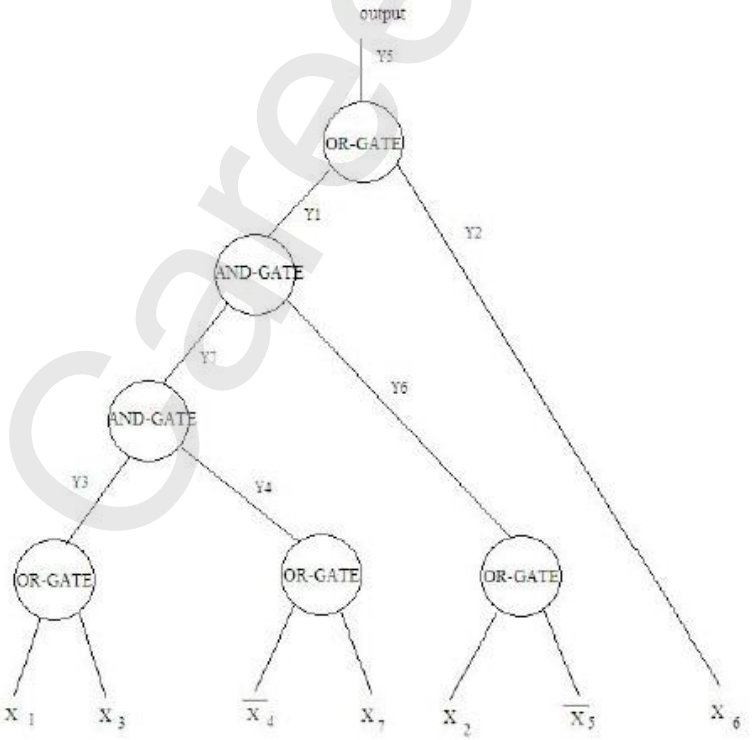
$$\left(\left(\left(x_1 \vee x_3\right)\right) \wedge \left(\left(\overline{x_4} \vee x_7\right)\right) \wedge \left(x_2 \vee \overline{x_5}\right)\right) \vee x_6$$

We will show how to make a formula that is satisfiable if and only if Ø is satisfiable. Firstly we can make a circuit corresponding to this formula. Here is a circuit.

3SAT is NP-complete

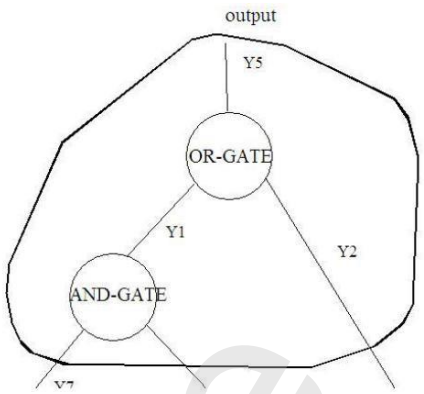


Now, we can introduce a variable for each wire of the circuit. So, we first label each wire with y' s. As shown in the figure.



Now, let us consider a part of the circuit as shown below: Here we have two wires y_1 and y_2 feeding into an or gate and the output wire is y_5.

Thus we can write y_5 is true if and only if y_1 and y_2 are true. We can express this in boolean logic as $\overline{y_5} \vee y_1 \vee y_2$. This says that either y_5 is false. Or if it is true then one of the y_1 or y_2 should be true. Note that this is a clause with three literals.



Similarly, suppose we have a and gate with y_7 and two wires y_3 and y_4 feeding into it. We can write $(\overline{y_7} \vee y_3) \wedge (\overline{y_7} \vee y_4)$ Note these are two clauses with two literals each.

So for each wire we can write a formula which is the and over all the wires. Finally we add the clause yn where yn is the output wire.

So for the example we find that the formula looks like $y_5 \wedge (\overline{y_5} \vee y_1 \vee y_2) \wedge$ .....

To satisfy this formula the output wire must be set to true. To satisfy the formula for the output wire the two wires feeding it must be set to true and so on.... Thus to make ψ true we must make an assignment of the x variables that makes Ǿ true and so on...

This shows the following. Given Ǿ we can construct ψ such that

1. Ǿ is satisfiable if and only if ψ is satisfiable.
2. ψ is in CNF.
3. Each clause in ψ has at most 3 literals.

It is your homework to show that you can now convert ψ into a formula in which each clause has exactly three literals. It is easy to see that this transformation can be done in polynomial time.

Theorem:
3SAT is NP-complete.

This is our second problem that we have shown to be NP-complete. Now, let us show that a problem from graph theory is NP-complete.

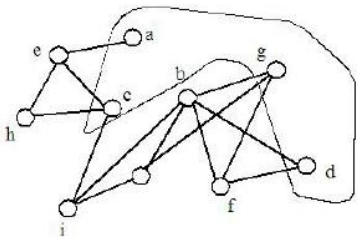
Let us start by defining a problem that we will prove is NP-complete.
Let us first recall a definition from graph theory:

Definition:
Let G = (V,E) be a graph. $I \subseteq V$ is called an independent set if the vertices in I have no edges between them.

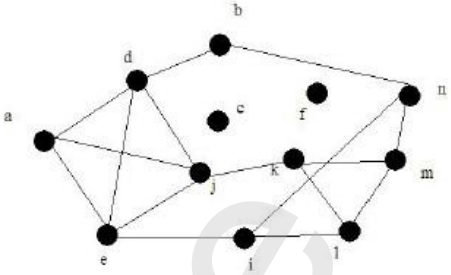
Thus for all $x, y \in I$ where $x \neq y$. $\{x, y\} \notin E$.

Independent Sets:

The figure shows a graph on 10 vertices. a, c, d, g is an independent set.



The figure shows a graph on 12 vertices. b, c, f, j,k is an independent set.



Let us define the following language:
IS = {<G, k> : G has an independent set of size k}.

So computationally we have a problem in which we will be given:

1. A graph G.
2. and an integer k.

We have to find out if G has a independent set of size k.

It is very easy to see that IS$\in$NP. We just have to give a verification algorithm. The details are your homework. Hint: It is easy to verify that a given set is an independent set in a graph.

We want to show that IS is NP-complete. Now, all we have to do is to show that SAT $\leq_p$ IS. Note that this is much easier than the proof of Cook-Levin theorem. We only have to show that one problem is reducible to IS as opposed to all the problems in NP. Let"s do that.

Theorem: SAT $\leq_p$ IS

Consequently IS is also NP-complete.
What do we want to do?

1. We will show that given a formula Ø we can construct a graph $G_{ø}$ and an integer k such that
2. Ø is satisfiable if and only if $G_{ø}$ has an independent set of size k

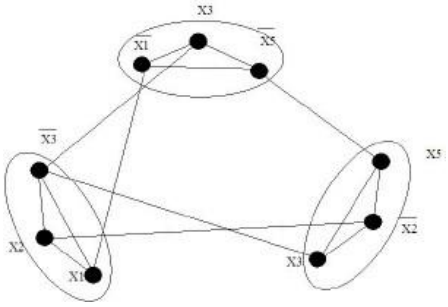
As 3SAT is NP-complete we may assume that Ø is in 3CNF.

Let us take the formula: $\left(x_1 \vee x_2 \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee x_4 \vee \overline{x_5}\right) \wedge \left(x_3 \vee \overline{x_2} \vee \overline{x_5}\right)$
and make the following graph.

For each clause place a cluster of vertices and connect them all to eachother. Label each vertex with the literal in the clause.

Between clauses connect x_i and $\overline{x_i}$. Let us see how this graph is made.

Suppose Ø has m clauses and n variables. Then we realize that $G_{ø}$ - will have m clusters.



First observation is

1. Any independent set must have at most one vertex from a cluster.
2. This means that the largest independent set is of size ≤ m.

Second observation is

1. If x_i in one cluster belongs to an independent set then no $\overline{x_i}$ belongs to the indept set.
2. If $\overline{x_i}$ in one cluster belongs to an independent set then no x_i belongs to the indept set.

Thus we can think of the vertices in the independent set as variables that are made true in that cluster.

This means if there is a independent set that contains m vertices it must have one vertex from each cluster. Thus the corresponding assignment must make the formula Ø true.

On the other hand if we Ø is satisfiable then the assignment satisfies each clause. So, it satisfies one literal in each clause. We can pick all these literals and get an independent set of size m.

Theorem:
Ø is satisfiable if and only if $G_{Ø}$ contains an independent set of size m. Where m is the number of clauses in Ø.

It is readily seen that $G_{Ø}$ can be computed in polynomial time. So this shows that 3SAT ≤p IS Thus IS is NP-complete.

Formally the reducibility is given by

1. Input <Ø>.
2. Let m be the number of clauses in Ø.
3. Compute $G_{Ø}$ and output <$G_{Ø}$,m>.

We have so far seen that SAT ≤p 3SAT ≤ IS. Is it true that IS ≤p SAT?

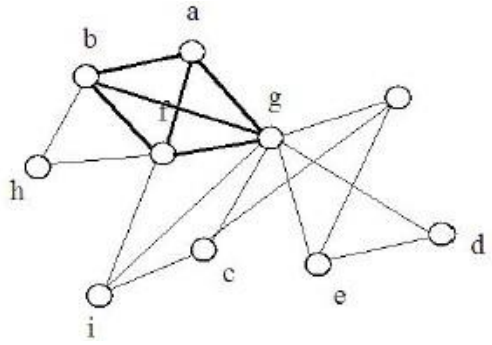
We have so far seen that SAT ≤p 3SAT ≤ IS
Is it true that IS ≤p SAT?

The answer is yes. This is a consequence of the Cook-Levin Theorem. Therefore, all these problems are polynomial time reducible to each other. Now, I want to show you two easy reducibilities. The first one is almost trivial. Let us define another problem in graph theory.

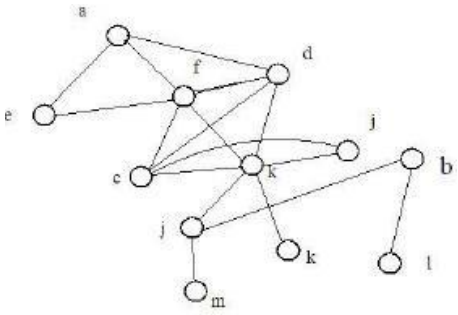
Definiation:

Let G = (V,E) be a graph. C $\subseteq$ V is called an independent set if the vertices in I have no edges between them. Thus for all x, y$\in$I where x$\neq$ y. $\{x, y\} \notin E$.

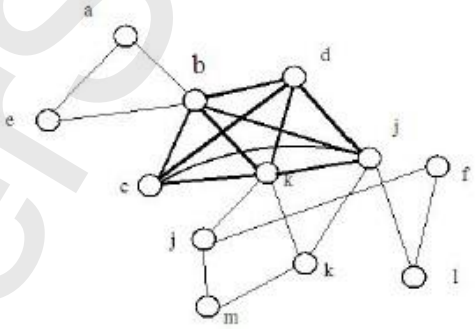
The figure shows a graph on 10 vertices. a, b, f, g is a clique.



The figure shows a graph on 12 vertices. a, d, f is a clique.



The figure shows a graph on 12 vertices. b, c, d, j,k is a clique.



Clique:

Let us define the following language:

CLIQUE = {<G, k> : G has an clique of size k}.

So computationally we have a problem in which we will be given:

1. A graph G.
2. and an integer k.

We have to find out if G has a clique of size k.

It is very easy to see that CLIQUE $\in$ NP. Again this is your homework. We want to show that CLIQUE is NP-complete.

We have three problems that we know are NP-complete.

1. SAT
2. 3SAT
3. IS.

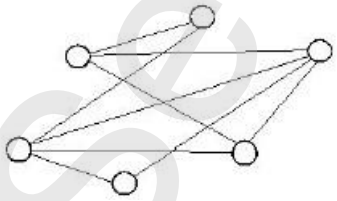
We can show any one of the following to establish the NP-completeness of CLIQUE.

1. SAT $\leq_p$ CLIQUE
2. 3SAT $\leq_p$ CLIQUE
3. IS $\leq_p$ CLIQUE

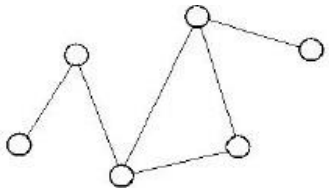
Which one should we choose. The answer is the one that is the easiest. There is no need to do more work than required. In this case, we should choose the last one because both the problems seem to be closely related.

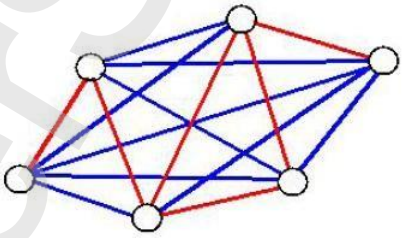
So what is the relation between independent sets and cliques? The answer is that it is very simple. Let G = (V,E) be a graph. Let us define the complement of a graph $\overline{G}$ = (V, $E^{/}$) where $E^{/}$ = {{x, y} : {x, y} $\notin$ E}. So $\overline{G}$ contains all edges that are not in G.

Put figure. Here is a graph G and its complement.



The figure is more clear if we put both the graphs together in different colors. G is in red and G is in blue.





Let us now revise the definitions of independent set and clique.

1. I is an independent set if for all x, y $\in$ I with x $\neq$ y, {x, y} $\notin$ E.
2. C is a clique if for all x, y $\in$ I with x $\neq$ y, {x, y} $\in$ E.

Thus we observe that I is an independent set in G if and only if I is a clique in $\overline{G}$!

Now the reducibility is easy to come up with:

1. On input <G, k>
2. Output < $\overline{G}$,k>

Theorem:

IS $\leq_p$ CLIQUE Consequently CLIQUE is also NP-complete.

How do we prove this. We have to give a reducibility. The reducibility takes a pair <G, k> and outputs < $\overline{G}$, k>. Clearly,

G has an independent set of size k if and only if $\overline{G}$ has a clique of size k.

Vertex Cover:

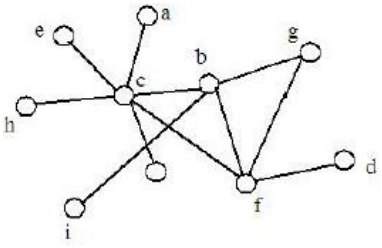
Let us define another problem in graph theory.

Definitaion:

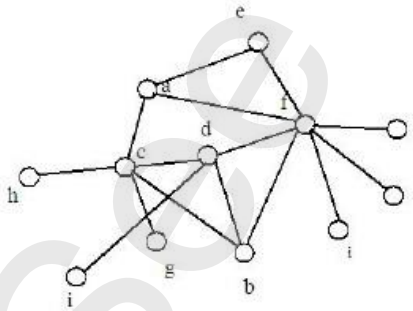
Let G = (V,E) be a graph. $W \subseteq V$ is called a vertex cover if all edges intersect W.

Thus for all $x, y \in E$ $\{x, y\} \cap W \neq Ø$.

The figure shows a graph on 10 vertices. b, c, f is a vertex cover.



The figure shows a graph on 12 vertices. a, c, d, f is a vertex cover.



Let us define the following language:

VC = {<G, r> : G has an clique of size r}.

So computationally we have a problem in which we will be given:

1. A graph G.
2. and an integer r .

We have to find out if G has a vertex cover of size r. It is very easy to see that $VC \in NP$. Again this is your homework. We want to show that VC is NP-complete. We have three problems that we know are NP-complete.

1. SAT
2. 3SAT
3. IS.
4. CLIQUE

We can show any one of the following to establish the NP-completeness of CLIQUE.

1. $SAT \leq_p VC$
2. $3SAT \leq_p VC$
3. $IS \leq_p VC$
4. $CLIQUE \leq_p VC$

Once again, we will choose the one where the reducibility is very easy to come up with!

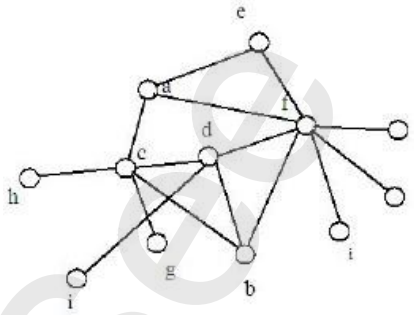
If we think about vertex cover and independent sets there is a nice relationship between them. Let G = (V,E) be a graph. Let us take W to be a vertex cover. Let us think about the complement of the vertex cover. We know that W is a vertex cover.

It"s complement is X = V \W. Thus for every edge $\{x, y\} \in E$ we have $\{x, y\} \cap W \neq Ø$. But this is the same as saying $\{x, y\}$ not $\subseteq X$. Thus no edge is present in X. So, what is X then? It is an independent set.

Theorem:
W is a vertex cover of G if and only if X = V \W is an independent set of G.

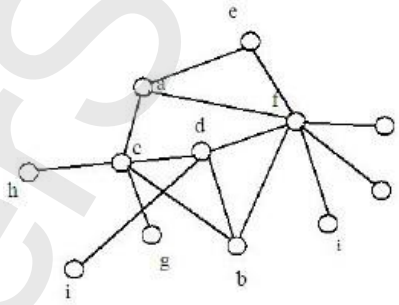
Theorem:
G contains an independent set of size k if and only if it contains a vertex cover of size n − k where n is the number of vertices in G.

Here is a graph G and with an independent set in it. The remaining vertices make a vertex cover.



The figure is more clear if we separate out the independent set from the rest of the vertices. Now the reducibility is easy to come up with:

On input <G, k>
Output <G, n – k> where n is the number of vertices in G.

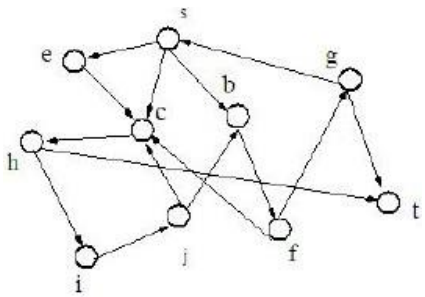


Theorem:
IS $\leq_p$ VC Consequently VC is also NP-complete.

Here is a list of the NP-complete problems we have so far.

1. SAT (Cook-Levin)
2. 3SAT (By reducing SAT to 3SAT)
3. IS (By reducing 3SAT TO IS)
4. CLIQUE (BY reducing IS to CLIQUE)
5. VC (BY reducing IS to VC).

Hamiltonian Paths:
Now, we will look at another problem and prove that it is NP-complete. This reducibility is not going to be so easy. Let us define the problem first.



Let G = (V,E) be a directed graph. Thus edges are ordered pairs and they have a direction. Here is a picture of a directed graph for you. Let us define a hamiltonian path in this graph.

A hamiltonian path from a s to t is a path in the graph:

1. It starts at s and ends at t.
2. It contains all the vertices of the graph exactly once.

NP-completeness results so far

1. SAT is NP-complete. (Cook-Levin)
2. 3SAT is NP-complete. (SAT ≤p 3SAT)
3. IS is NP-complete. (3SAT ≤p IS)
4. CLIQUE is NP-complete (IS ≤p CLIQUE)
5. VC is NP-complete (CLIQUE ≤p VC)

Our goal now is to add one more problem to this picture. We want to show that the hamiltonian path problem is also NP-Complete. The picture will then look like this.

Let G = (V,E) be a directed graph and s, t $\in$ V be two vertices. A Hamiltonian path from s to t is a path that

1. Starts and s.
2. Ends at t
3. visits all the vertices of the graph exactly once.

Let us define: HAMPATH {< G, s, t >: G has a hamiltonian path from s to t}.
Thus the computational question is that given a graph G and two vertices s and t. Does there exist a hamiltonian path from s to t. We can easily show that HAMPATH is in NP.

Theorem:
3SAT ≤p HAMPATH, Consequently HAMPATH is NP-complete.

Given a boolean formula - we will define a graph G- and two vertices s and t such that G- has a hamiltonian path from s to t if and only if Φ is satisfiable.
Let us take
$\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ and $x_1, \ldots, x_n$ be the variables in Φ. We describe how to construct G_Φ.

For each variable xi we create a diamond shape structure as shown in the figure.
For each Clause C_j we create a vertex and label it C_j . Show figure. Two special vertices s and t are added to the graph.

Now, we show how to connect the diamond vertices to the clause vertices. Each x_i has two vertices for each clause C_j . So, the diamond vertices are as follows:

Show figure. As you can see the vertices are paired up. Two verices each correspond to a clause. We label them C^L_j and C^R_j for left and right.

Now, let us discuss how to connect the vertices in the diamond to the clauses. Here is the rule. If x_i appears in C_j then:

Connect the vertex in the diamond for x_i that is labeled C^L_j with C_j . Also, connect C_j with C^R_j . Formally, all the following edges (C^L_j, C_j) and (C_j, C^R_j). Show figure.

Here is another rule
If xi appears in Cj then: Connect the vertex in the diamond for xi that is labeled C^R_j with Cj .
Also, connect Cj with C^L_j. Formally, all the following edges (C^R_j,Cj) and (Cj , C^L_j).
Note that this is backwards. Show figure.

This finishes the construction of the graph. Let us look at a complete picture for the following formula $\Phi = \left(x_1 \wedge \overline{x_2} \right)\left(x_2 \wedge x_3 \right)$

What can we say about this graph. Well, I will convince you that:

1. If Ǿ is satisfiable then this graph has a hamiltonian path from s to t.
2. If this graph has a hamiltonian path from s to t then Ǿ must be satisfiable.

Actually if we ignore the clause vertices then finding a hamiltonian cycle in this graph is easy. We can go from each diamond in two ways.

The idea is if we traverse a diamond from left to right then x_i is set to true and if we traverse it from right to left then x_i is set to false.

Thus the 2^n assingments correspond to travesing the diamonds in 2^n possible ways. Let us say we have the assignment x_1 = ture, x_2 = false, x_3 = true then we will traverse the graph as shown. Show figure.

Note that if we are traversing the diamond for xi from Left to right then we can “pick” up a clause C_j, if xi appears in C_j. This due to construction. This is because we have an edge from C^L_j to C_j and back to C^R_j . Show figure.
Note that if we are traversing the diamond for xi from Right to Left then we can “pick” up a clause C_j if xi appears in C_j. This is again by construction. In this case we have an edge from C^R_j to C_j and back to C^L_j.

Suppose that we have a satisfying assignment for Φ which satisfies all the clauses. Then we can pick up all the clauses by traversing the graph in the way specified by the satisfying assgnment.

So, if Φ is satisfiable. Take a satisfying assignment of Φ.

1. Start from s.
2. Traverse each diamond for x_i in the order specified by the assignment. If x_i is set true go from left to right. If it is set false go from right to left.
3. Along the way pick up any C_j that has not been already picked up.

This yeilds our hamiltonain path from s to t.

Lemma:
If Φ is satisfiable then G_Φ has a hamiltonian path from s to t. We have to prove the converse of this theorem next time

TSP Problems:

The traveling salesman problem is the following problem: A traveling salesman wants to start from his home and visit all the major cities in district and come back to his hometown. Given that he has found out about the fares from each city to every other city can we find the cheapest way for him to make his tour.

The TSP is not a yes/no question. It is an optimization question. It is not only asking us to find a hamiltonian path but find one with the least cost. So, intuitively it seems much harder to solve. We can make this intuition precise as follows:

Theorem:
If the TSP has a polynomial time algorithm then P = NP.

The Subset Sum Problem

Let X = {x_1, x_2,….., x_n, } be a (multi)-set of positive integers. For any S $\subseteq$ X let us define the subset sum as follows $sum(s) = \sum_{x_i \in S} X_i$

A set can have at most 2^n different subset sums. But typically they are less as many sums add to the same number.

Let X = {2, 5, 6, 7} then the subsets and subset sums are:

Φ	0	{ 5, 6 }	11
{ 2 }	2	{ 5, 7 }	12
{ 5 }	5	{ 6, 7 }	13
{ 6 }	6	{ 2, 5, 6 }	13
{ 7 }	7	{ 2, 5, 7 }	14
{ 2, 5 }	7	{ 2, 6, 7 }	15
{ 2, 6 }	8	{ 5, 6, 7 }	18
{ 2, 7 }	9	{ 2, 5, 6, 7 }	20

Let X = {1, 2, 3, 4} then the subsets and subset sums are:

Φ	0	{ 2, 3 }	5
{ 1 }	1	{ 2, 4 }	6
{ 2 }	2	{ 6, 7 }	7
{ 3 }	3	{ 3, 4 }	6
{ 4 }	4	{ 1, 2, 3 }	7
{ 1, 2 }	3	{ 1, 2, 4 }	8
{ 1, 3 }	4	{ 1, 3, 4 }	9
{ 1, 4 }	5	{ 1, 2, 3, 4 }	10

Given a set X of positive integers and a target t. Does there exist a subset of X that sums to t?

Let us say the input is: <{2, 5, 6, 7}, 13> then then there is a subset. In fact, as we saw there are two of them {6, 7} and {2, 5, 6} that sum to 13. So the answer is yes!

Let us say the input is: <{2, 5, 6, 7}, 10>: You can check that no subset sums to 10.

Let us define
SS = {<S, t>: X has a subset that sums to t}:

Thus the computational question is that given a set X and a number t. Does there exist a hamiltonian path from s to t? We can easily show that SS is in NP. This is again your homework.

Theorem:
3SAT ≤p SS, Consequently SS is NP-complete.

Given a boolean formula Φ we will define an instance <X, t> such that X has a subset S summing to t if and only if Φ is satisfiable.

Let us take Φ = C_1 ^ C_2 ^ ……. ^ C_m and x_1,……, x_n be the variables in Φ. The instance <X, t> will be given by numbers in base 10. Each number will have n + m digits. The first n digits will correspond to variables and the last m digits will correspond to clauses.

For each variable x_i we will put two numbers y_i and z_i in the list. The target value is going to be given by t. This is all shown in the following table

	1	2	3	4	⋯	l	c_1	c_2	⋯	c_k
y_1	1	0	0	0	⋯	0	1	0	⋯	0
z_1	1	0	0	0	⋯	0	0	0	⋯	0
y_2		1	0	0	⋯	0	0	1	⋯	0
z_2		1	0	0	⋯	0	1	0	⋯	0
y_3			1	0	⋯	0	1	1	⋯	0
z_3			1	0	⋯	0	0	0	⋯	1
⋮				⋱		⋮	⋮		⋮	⋮
y_l						1	0	0	⋯	0
z_l						1	0	0	⋯	0
g_1							1	0	⋯	0
h_1							1	0	⋯	0
g_2								1	⋯	0
h_2								1	⋯	0
⋮									⋱	⋮
g_k										1
h_k										1
t	1	1	1	1	⋯	1	3	3	⋯	3

Rules for making the table:

1. Make two rows y_i and z_i for each variable.
2. Place a 1 in the column corresponding to x_i in both y_i and z_i.
3. Place a 1 in the column corresponding to c_j in yi if x_i appears in c_j.
4. Place a 1 in the column corresponding to c_j in zi if $\overline{x_i}$ appears in c_j.

Lastly for each clause c_j add g_j and h_j having one exactly in the column corresponding to c_j.

t has n ones and m threes. Ones correspond to the variable columns and three correspond to the clause columns.

Note that if a subset is chosen there will be no carries. Let us see why? Let us choose all the elements shown here. Then in a given column there are only a few entries that are one. Thus there can be no carries.

Out of y_i or z_i exactly one must be chosen otherwise the target is not reached. That will correspond to setting x_i true or false.

Now the most important part. Note that if y_i is chosen then the column corresponding to the clauses that contain x_i has a 1 there. Similarly, if z_i is chosen then all the columns corresponding to $\overline{x_i}$ have a 1 in them.

Question: How can the target be reached.

Let us say we have a satisfying assignment.
We select y_i if x_i = ture and z_i otherwise. What will happen?

We get a sum that gives us all 1"s for the first n columns. Furthermore, each of the columns corresponding to a clause will have at least a sum of 1.

Now, we can pick up 0, 1 or two out of g_i , h_i to make each digit 3.

Conversely, if someone says they have a subset that sums to 1. You can be rest assured of the following things:

1. They must have chosen exactly one out of y_i and z_i . Why?
2. This defines an assignment.
3. This assignment must satisfy all the clauses. Why?

Lemma:
Φ is satisfiable if and only if X has subset that sums to t.

This lemma shows that f (Φ) = <X, t> is a polynomial time reducibility from SS to 3SAT. Thus SS is NP-complete.

Travelling Saleman Problem:

Theorem: If the TSP has a polynomial time algorithm then HAMCYCLE is in P.

Assume that TSP has a polynomial time algorithm A. Consider the following algorithm for HAMCYCLE:

1. On input < G,V >.
2. Construct a new graph K_n having all possible edges over V.
3. If (a, b) is an edge in G then give it weight 1. Otherwise, give it weight 0.
4. Use A to find the shortest tour in K_n.
5. If the shortest tour has weight n then accept. Else reject.

We just notice that any HAMCYCLE in G becomes a weight n HAMCYCLE in K_n. Thus the algorithm is polynomial time if A runs in polynomial time. This proves our theorem.

Now let us look at three versions of the TSP problem.

1. The usual TSP: Given a graph, G, with weights on edges find the minimum weight Hamcycle in G.
2. The Metric TSP: Given a complete graph with edge weights satisfying $w(a, c) \leq w(a, b) + w(b, c)$ find the minimum weight Hamcycle in G.
3. The Euclidean TSP: Given n points on the plane find the cheapest tour going through the points.

First let us turn these problems into yes/no questions:

1. The usual TSP: Given a graph, G, with weights on edges and a target t does G have a hamcycle with weight t.
2. The Metric TSP: Given a complete graph with edge weights satisfying $w(a, c) \leq w(a, b) + w(b, c)$ does G have a hamcycle with weight t.
3. The Euclidean TSP: Given n points on the plane and a target t. Is it possible to tour all the vertices by moving a distance of at most t?

These problems now become languages if we pose the appropriately. We want to study the complexity of these questions.

1. TSP is NP-complete.
2. The metric TSP is NP-complete.
3. The Euclidean TSP is NP-complete.

So it seems applying the restrictions does not seem to simplify this problem.

Theorem: The metric TSP is NP-complete.

Proof (Outline) Given a graph G we can construct a complete graph K_n on the same vertex set. If a (a, b) is an edge in G then we give this edge weight 1 otherwise we give this edge weight 2.

As we have seen: G has a hamcycle if and only if this K_n has a tour of size n. All we do is observe that this weight function satisfies the triangle inequality. Why?

We just have to verify that $w(a, c) \leq w(a, b) + w(b, c)$ Note that the wieghts are 1 or 2. So, this is always satisfied. Since $w(a, c) \leq 2$ and $w(a, b) + w(a, c) \geq 1 + 1 = 2$

Now let us talk about a different notion of algorithms. These are called approximation algorithms. Let us first understand this from a practical point of view. Suppose that you have a graph G and you want to find the cheapest Hamcycle in G. It is not good enough if someone were to tell you that the problem is NP-complete.

What do you want?

You want a tour that is cheapest. Suppose someone were to propose to you that they will find a "cheap" tour for you. But they cannot guarantee that it is the cheapest. You would take that offer since you have no other option.

Now, there is a new question. Can we devise a fast (polynomial time) algorithm that does not find the cheapest tour but a cheap enough tour? Such an algorithm will be called an approximation algorithm.

How will we say which algorithm is good or bad? How will we compare these algorithms. If we have two algorithms A_1 and A_2 you would like the one that finds a cheapest tour in your graph. But that is very subjective.

A more objective way to compare algorithm is to compare approximation ratios. Suppose that G is a graph with a cheapest tour of size t. If an algorithm A guarantees to return a tour of size t then we say A is an α approximation algorithm.

1. Thus a 2-approximation algorithm will return an answer that is at most twice the cheapest tour.
2. A 1.5 approximation algorithm will return a tour that has cost at most 50% more than the optimal.
3. 1.01 approximation algorithm will return a tour that has cost at most 1% more that the cheapest tour.

Let us look at these problems from another point of view.

Question:
What can we say about polynomial time approximation algorithms for these problems.

These problems look very different.

1. If P ≠ NP then general TSP cannot be approximated to any constant factor (much more is true).
2. The metric TSP can be approximated to a factor of 1.5.
3. The Euclidean TSP can be approximated to a factor of $1 + \alpha$ for any $\alpha > 0$. So from this point of view applying restrictions does make the problems simpler.

We will look at the first two results.

Suppose someone claims that they can approximate TSP within a factor of α. Let us call such an algorithm A. We can use A to solve the HAMCYCLE problem as follows:

1. On input G = (V,E).
2. Make a complete graph on vertex set C.
3. if (a, b) is in G give it weight $2n\alpha$.
4. Use an A to find the cheapest tour in the complete graph.
5. If the weight of this tour is less than $\alpha n + 1$ accept else reject.

Any HAMCYCLE in G becomes a tour of weight n in the complete graph. On the otherhand any other cycle in the complete graph has weight at least $n - 1 + 2\alpha n + 1 = 2(\alpha + 1)n > \alpha n$.

Thus the algorithm will return a hamiltonian cycle if it exists. Even though it is an approximate algorithm.

Hardness results for approximation algorithms are not usually as easy to prove as we have done for this problem. But now there is also unified theory of that also. We will discuss that in the later lectures. Now, let us prove a simpler theorem than we promised.

Theorem: There is a two approximation algorithm for the metric TSP problem.

We can simply state the algorithm:

1. Input a complete-graph and weights that satisfy the triangle inequality.
2. Compute the minimum spanning tree T of this graph.
3. Visit all the vertices of T using DFS and output the nodes without duplicates.

The algorithm takes polynomial time. How can we show that this gives us a tour that is at most twice the optimal tour in the graph. We need to analyze this algorithm.

The cost of the MST is less than or equal to the cost of the cheapest tour.

Proof. Delete any edge from the cheapest tour. We can a path. The path is a spanning tree (of a special kind). This path must have more cost than the MST (by definition of MST).

The second thing to observe is that: In a graph that satisfies the triangle inequality. The cost of the tour given by DFS is at most two times the cost of the tree.

We will see this using the following picture.

Take a tree.
Duplicate the edges. This gives us a tour which has all vertices but they are repeated. The cost of this is twice that of the tree. As each edge is doubled. Now, we short cut.

In short cutting we replace $v_1, \ldots, v_k$, by $v_1, \ldots, v_k$. Because of the triangle inequality the short cutting does not increase the cost of going from v_1 to v_k.

Note that $w(v_1, v_k) \leq w(v_1, v_2) + \cdots w(v_{k-1}, v_k)$.

There is a more clever algorithm that can give an approximation ratio of 1.5. Improving this ratio; that is, finding an algorithm that has an approximation ratio of less than 1.5 is very old open problem in computer science.

For the Euclidean TSP Arora has developed a wonderful algorithm. It is quite complicated and approximates it within a factor of $1 + \alpha$. The polynomial running time of the algorithm depends on α.
So we realize that even if a problem is proved to be NP-complete there is a lot that we can do about it.

If you know that a problem is NP-complete you can:

1. Restrict the problem and see which restricted versions of this problem can be solved efficiently.
2. Look for approximation algorithms for such problems.

The PCP Theorem and Hardness of Approximation

In the 1990"s several computer scientist took active interest in approximation algorithms. Usually, it is not difficult to show that problems are NP-complete. However, it is very difficult to show the approximation algorithms of problems may not exist.

This lead to one of the deepest theorems in computer science called the PCP PCP theorem. Here PCP stands for probablitically checkable proofs.

Using this theorem it is possible to prove several hardness results for approximation algorithms. It can be shown that if there are approximation algorithms for certain problems then P = NP. In some cases the best known algorithm match these hardness results.

Thus showing in some sense that we may have found the most promising approximation algorithms for these problems.

The original proof of the PCP theorem was very algebraic. Now, a combinatorial proof has also been found.

Let us explain what the theorem says. Informally talk about PCP. What a probablistically checkable proof is and what the theorem says. How spectacular it is :)

Space Complexity

We would now like to study space complexity. The amount of space or memory that is required to solve some computational problem. The following model of computation is the most convenient to study space complexity.

Consider a Turing machine M which has:

1. An input tape that is read only. The input is encloses in # symbols.
2. A work tape that is initially all blank.
3. The machine is not allowed to write a blank symbol on the work tape.

Here draw a picture of a machine:

The space used by this machine on input x is the number of non-blank symbols on the work tape at the end of the computation. Note that in this case:

1. We do not count the input as space used by the machine.
2. The machine is allowed to read the input again and again.
3. However, since the input tape is read only the machine cannot use that space to store intermediate results.

A good realistic machine to keep in mind is a computer with a CD-Rom. The input is provided on the CD-Rom and we are interested in figuring out how much memory would be required to perform a computation.

Let M be a Turing machine that halts on all inputs. We say that M runs in space s(n) if for all inputs of length n the machine uses at most s(n) space.

Let us look at a concrete example. Let us consider the language $L = \{x \in \{0, 1\}^* : x = x^r\}$ this is the set of all palindromes over {0, 1}. Lets look at two TMs that accept this language and see how much space they require.

Let us describe a TM M_1 that accepts L as follows we will give a high level description.

1. The machine copies the input to the work tape.
2. It moves one head to the left and the other to the rightmost character of the copy of the input.
3. By moving one head left to right and the other right to left it makes sure that the characters match.
4. It all of them match the machine accepts otherwise it rejects.

It is not hard to see that the space required by M_1 is linear. Thus M_1 accepts L in space O(n). The question is can we accept this language in less space.

Question: Is it possible to accept L in space o(n) ?

Now we want to save space.

How about the following machine:

1. Read the first character and put a dash on it.
2. Match it with the last character which does not have a cross on it and cross it.
3. If all characters have a dash and get matched then accept.

Here is a picture of how this machine works:

The problem is that we are not allowed to write on the input tape. Since, it is read only. Thus we must copy the contents on the input tape to the work tape. This would give a machine that takes space n at least. Can we some how do better? Here is another idea.

Let us describe a TM M_2 that accepts L.

1. The machine write 1 on its work tape. We call it the counter.
2. It matches the first symbol with the last symbol.
3. In general it has i written on the counter and it matches the i-th symbol with the i-th last symbol.
4. In order to match get to the i-th symbol the machine can use another counter that start at i and decrements each time it moves it head on the readonly head forward.
5. Similarly to get to the i last symbol it can use another counter.

This machine only uses three counters. Each counter needs to store a number in binary. The number will reach the maximum value of n.

To represent n in binary we need only $\log_2 n$ bits. Thus the machine will require only O(log n) space. This is a real improvement over the previous machine M_1 in terms of space.

A few points to remember that are made clear by this example are:

1. If you are claiming that you have a sublinear space TM then make sure that you are not overwriting the input or using that space.
2. The machine M_1 takes time O(n) and space O(n). On the other hand M_2 takes time $O(n^2)$ and space $O(\log_2 n)$. So M_1 is better in terms of time and M_2 better in terms of space. This is quite typical.
3. If you are asked to design a machine that takes less space then no consideration should be given to time.

If M is a non-deterministic TM. Suppose that all branches of M halt on on all possible input. We define the space complexity f(n) to be the maximum number of tape cells that M scans on any branch of its computation for any input of length n.

Non-deterministic space is a useful mathematical construct. It does not correspond to any realistic model of computation.

Let $f : N \rightarrow N$ be a function. We define,
SPACE(f (n)) = {L : L is accepted by an O(f (n)) space TM }.

Similarly, we define
NSPACE(f (n)) = {L : L is accepted by an O(f (n)) space N_{TM} }.

Let us consider some facts which make space complexity very different from time complexity.

A sublinear time TM cannot completely read its input. Therefore the languages accepted in sublinear time are not very interesting. This is not the case with sublinear space algorithms. In fact, it is very interesting to study sublinear space algorithms.

Let us look at an NP-complete problem. We studied CNFSAT let us see if we can come up with an space efficient algorithm for this problem. Let us say we are given a boolean formula ϕ on n variables in CNF. Let $x_1, x_2, \ldots, x_n$ be the boolean variables that appear in ϕ and $C_1, \ldots, C_m$ be the clauses of ϕ.

Consider the following algorithm (TM)

1. For each possible assignment of $x_1, x_2, \ldots, x_n$
2. For each $i = 1, \ldots, m$ if C_i is made true accept.
3. Reject.

This idea can be implemented in linear space.

1. Each assignment of $x_1, x_2, \ldots, x_n$ can be stored in n bits.
2. To make sure C_i is made true. We just have to read the clause and make sure if one of the literal"s is made true in each clause.

This shows:

Theorem: SAT is in SPACE(n).

On the other hand we know that it SAT is NP-complete. Thus that leads to the conjecture that $P \not\subset$ SPACE(n). But this is only a conjecture!

Let us look at a non-deterministic linear space machine. Let us consider the following language E_{NFA} = {<A> : A is an NFA and L(A) = ϕ }. We will devise a non-deterministic linear space algorithm that decides $\overline{E_{NFA}}$

Let us recall some facts about DFAs and NFAs

1. If D is a deterministic finite automata with k states and L(D) ≠ ϕ, then D accepts a string of length at most k.
2. If A is an NFA with q states then there exists a DFA D such that L(A) = L(D) and the number of states in D is at most 2^q.

As a consequence we get:

Theorem: If A is a NFA with q states with L(A) ≠ ϕ, then A accepts a string of length at most 2_q.

Let us consider the following non-deterministic algorithm (TM) N:

1. On input <A> a N_{TM}.
2. Let q be the number of states in A.
3. Non-deterministically guess an input x of length at most 2_q.
4. Simulate A on x.
5. If A accepts x then accept. Else reject.

Now, if L(A) = ϕ then all branches of the above N_{TM} will reject. On the other hand if L(A) ≠ ϕ then there will be at least one branch that will accept. Thus L(N) = $\overline{E_{NFA}}$

The above NTM requires 2^q space. Now, we just notice that in order to simulate an NFA we do not need to know all of x but only the current symbol of x. Thus we can reuse space. We can be more space efficient by "guessing the symbols" of x as we go along. Let us consider a new machine.

1. On input <A> a NTM.
2. Let q be the number of states in A.
3. For i = 1, . . . , 2^q
4. Non-deterministically guess the i-th symbol of x
5. Simulate A by placing markers on the current states of A.
6. If A accepts x then accept.
7. Reject.

This NTM requires a counter that can count upto 2^q. Such counter requires q bits. Thus the whole simulation can be done in linear space. Thus we have

Theorem: $\overline{E_{NFA}} \in$ SPACE(n)

It is interesting to note that ENFA is not known to be in NP or co-NP.

Let us now look at another very interesting problem. We call this reachability. Let G = (V,E) be a graph we say that a vertex t is reachable from s if there is a path from s to t. How can we solve the reachability problem.

Lets define R = {<G, s, t> : G is a graph and s is reachable from t }.

Thus the computational question is "Given a graph G and two vertices s and t decide if there is a path from s to t".

You learn two algorithms to solve this problem. One is called DFS and the other is called BFS. Rough outline is the following:

1. Mark s
2. Repeat n − 1 times.
3. If there is a edge from a marked vertex a to an unmarked vertex b then mark b.
4. If t is marked accept. Else reject.

Usually the emphasis is on the time complexity of this algorithm and DFS and BFS are elegant and efficient implementations of this idea (with very nice properties). However, now we want to view these algorithms from the point of view of space complexity.

The point to notice is that both these algorithm require us to mark the vertices (which initially are all unmarked). Therefore, we need to have one bit for each vertex and we set it when we mark that particular vertex. Thus the whole algorithm takes linear space.

Question: Can reacability be solved in sublinear space?

In fact, as we will see little later. Reachability is one of the most important problems in space complexity. Let us for now start with an ingenious space efficient algorithm for reachability.

Let G = (V,E) be a graph on n vertices and a and b be two vertices.

We say that b is reachable from a via a path of length k if there exist $a = v_0, \ldots, v_k = b$ such that $(v_i, v_{i+1}) \in E$ for all $i = 0, \ldots, k-1$.

Let us begin with the following facts:

Fact: If b is reachable from a via a path of length $k \geq 2$ then there is a vertex c such that:

1. c is reachable from a via a path of length $\lceil k/2 \rceil$
2. b is reachable from c via a path of length $\lceil k/2 \rceil$

Let us begin with the following facts:

Fact: If b is reachable from a then it is reachable via a path of length at most n. Where n is the number of vertices in the graph.

Fact: b is reachable from a via a path of length 0 if and only if b = a.

Fact: b is reachable from a via a path of length 1 if and only if $(a, b) \in E$.

Now, let us write a function reach(a,b,k). This function will return true if b is reachable from a via a path of length at most k.

```
reach(a,b,k)
1. If k = 0 then
2.      if a = b return true. Else return false.
3. If k = 1 then
4.      if a = b or (a, b) ∈ E return true. Else return false.
5. For each vertex c ∈ V
6.      if (reach(a,c, ⌈k/2⌉) and reach(c,b, ⌈k/2⌉)
7.          return true;
8. return false;
```

How much space does this program take?

We can solve the reachability problem by simply calling reach(s, t, n).
But the question is how much space will that take?

This is a recursive program and we need to analyze the space requirements of this recursive program. There are three parameters that are passed and one local variable (c). These have to be put on the stack. How much space do these variables take?

If the vertex set of the graph is $V = \{1, \ldots, n\}$ then each vertex can be stored in $\log_2 n$ bits. Similarly, the parameter k is a number between 1 and n and requires $\log_2 n$ bits to store. Thus each call will require us to store O(log n) space on the stack. So, the space requirements are O(log n) × depth of the recursion.

The depth of the recursion is only $\lceil \log_2 n \rceil$. Since, we start with n and each recursive call reduces the it by a half. Hence the total space requirements are $O(\log^2 n)$

Theorem: reachability is in SPACE($\log^2$ n).

Space Complexity Classes

Let us prove a very simple relationship between time and space complexity. Let us say that M is a deterministic TM that uses space s(n). Can we give an upper bound on the time taken by the TM M?

Let us count how many configurations of M are possible on an input of length n. Note that we can specify a configuration by writing down the tape contents of the work tape (there are at most s(n) cells used there). We also have to specify the position of the two heads.

Thus there are $n \times s(n) \times b^{s(n)}$ configurations. Here b is the number of symbols in the work alphabet of the machine.

Theorem: If M runs in space s(n) then there are at most $2^{O(s(n))}$ distinct configurations of M on x.

We only have to note that $n \times s(n) \times b^{s(n)} = 2^{O(s(n))}$

Theorem: If M accepts an input in space s(n) then it accepts it in time $2^{O(s(n))}$

This theorem is true for both deterministic and non-deterministic TMs. Let us define the analogue of P and NP from space complexity point of view:

$$PSPACE = \bigcup_k SPACE\left(n^k\right)$$

$$NSPACE = \bigcup_k NSPACE\left(n^k\right)$$

The analogue of the P vs. NP question for space computation is the following:

Question: Is PSPACE = NPSPACE?

Unfortunately there is no prize on this problem. This problem was solved by Savitch. He proved PSPACE = NPSPACE. We will prove this theorem. This theorem is a consequence of the following theorem.

Theorem: Let $f(n) \geq n$ then NSPACE(f (n)) $\subseteq$ DSPACE(f^2(n)).

We want to show that if L is accepted by non-deterministic TM in space O(f(n)) then it can be accepted by a determinitic TM that uses space $O(f^2(n))$. The main idea is similar to the one that we used for reachability.

Let N be a non-deterministic TM that accepts a language L in space f(n). We will build a deterministic TM M that accepts L in space $f^2(n)$. M will simply check if an accepting configuration of N is reachable from the starting configuration.

Let us for a moment make the assumption that we can compute f(n) easily (we will come back to this point). Let us now define a problem called yeildability. Suppose we are given two configurations c_1 and c_2 of the NTM N and a number t.

The question is Is can c_1 yeild c_2 in t steps.

We further assume that N will use space f(n).
Consider the following TM (algorithm)

CANYEILD(c_1, c_2, t)

1. If t = 1 then
2. if $c_1 = c_2$ or c_2 follows from c_1 via rules of N return true. Else return false.
3. For each configuration c_0 of length at most f (n)
4. if (CANYEILD(c_1, c'', $\lceil t/2 \rceil$) and CANYEILD(c'', c_2, $\lceil t/2 \rceil$)
5. return true;
6. return false;

Once again this is a recursive program. So, we have to see how much space does it require. It stores the configurations on the stack that takes space O(f(n)).

Furthermore, if we call CANYEILD(c_1, c_2, t) then the depth of the recursion will be $\log_2$ t. Now, given a TM M and an input x let us consider the following deterministic TM.

1. Input x.
2. Compute f (n)
3. Let c_0 be the initial configuration of M on x.
4. For all accepting configurations c_f of length at most f (n)
5. if CANYEILD(c_0, c_f , $2^{O(f(n))}$) accept.
6. reject.

It is clear that if M accepts x in space f (n) then the above algorithm will also accept x. On the otherhand if M does not accept x in space f(n) then the above algorithm will reject. The space required by the above algorithm is

$$f(n) \log(2^{O(f(n))}) = O(f(n) \times f(n)) = O(f^2(n)).$$

This completes the proof of our theorem. There is one technicality left. If you see the algorithm, we have said we will compute f(n). However, we may not know how to compute f(n). In that case, what can we do? This is just a technicality and we can add the assumption that f(n) is easily computable to our theorem. Another way to get around it is as follows:

Note that we can use the above function to compute if the initial configuration yields a configuration of length f or not. We can start with f = n and and increment its value till we find the length of the largest configuration that we can get to from the initial configuration. We can use that value of f now to see if M reaches an accepting configuration of length f.

The above theorem shows that $NSPACE(n^k) \subseteq DSPACE(n^{2k})$
and therefore, PSPACE = NPSPACE.

Let us review some relationships between space and time and non-determinism and determinism.

1. DTIME(t(n)) $\subseteq$ NTIME(t(n)).
2. DSPACE(s(n)) $\subseteq$ NSPACE(s(n)).
3. TIME(s(n)) $\subseteq$ DSPACE(s(n)).

These relationships are trivial.

Here are the non-trivial relationships.

1. NTIME(t(n)) $\subseteq$ DTIME($2^{O(t(n))}$). Simulate the non-deterministic TM.
2. NSPACE(s(n)) $\subseteq$ DSPACE($s^2(n)$). Savitch"s theorem.
3. DSPACE(s(n)) $\subseteq$ DTIME($2^{O(s(n))}$) for s(n) ≥ n.

Our favorite complexity classes:

Time:

$$P = \bigcup_k DTIME(n^k)$$

$$NP = \bigcup_k NTIME(n^k)$$

$$EXPTIME = \bigcup_k NTIME(n^{n^k})$$

Space:

$$PSPACE = \bigcup_k \mathrm{DSPACE}(n^k)$$

$$NPSPACE = \bigcup_k \mathrm{NSPACE}(n^k)$$

Thus we have the following relationships: P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE $\subseteq$ EXPTIME.

So far complexity theorist have only been able to show that P ≠ EXPTIME. Therefore, one of the above (or all) containments must be non-trivial but this all we can say now.

This picture shows these complexity classes. Put Picture from Sipser Figure 8.1 Page 282.

Motivated by the theory of NP-completeness we can ask the question if there are any PSPACE-complete problems. What kind of problems these would be? Let us make a precise definition first.

A language B is PSPACE-complete if

1. B is in PSPACE.
2. Every A in PSPACE is polynomial time reducible to B.

Pictorially we will have:

Why is it that we require that A be polynomial time reducible to B? Why not require that it is polynomial space reducible to B?

Well, what we want to say is that if a PSPACE-complete language is easy then all languages in PSPACE are easy! In other words, we claim that

Theorem: If any PSPACE-complete language is in P then PSPACE = P

This is the critical theorem (and its analogue in the theory of NP-completeness) that gives evidence that PSPACE-complete languages are hard.

The proof of this theorem reveals the answer. Let B be any PSPACE-complete language. If B is in P then there a polynomial time TM M that accepts B in time $O(n^k)$. Let $A \in$ PSAPCE.

Since $A \leq_p B$ there a polynomial time reduction f from A to B: Let N be a polynomial time TM N that computes f in time $O(n^r)$. Consider the following algorithm:

1. On input x.
2. Using N compute y = f (x).
3. Run M on y. If M accepts accept else reject.

Clearly this algorithm recognizes B. The running time of this algorithm is $O(|x|^r) + O(|y|^k)$.
If $|x| = n$ then $|y| \leq n^r$ thus the above algorithm runs in $O(n^{kr})$. This shows that A is in P.

Now the critical point is that f can be computed in polynomial time. If f were polynomial space computable we could only say that the above algorithm runs in polynomial space. We will end up proving the following uselessly trivial theorem.

If any PSPACE-complete language is in P then PSPACE = PSPACE

Note that if B is PSAPCE-complete and $A \in$ NP then $A \leq_p B$. Thus all PSPACE-complete languages are also NP-complete.

Recall the Cook-Levin Theorem.
Theorem: SAT is NP-complete.

It gave us a vast understanding of NP. It identified one hard problem in NP. What we want is to identify such a problem for PSPACE.

Recall SAT. What we want to do is consider its generalizations now. Let us look at SAT
Given a boolean formula on n variable $x_1,\ldots,x_n$, $\phi(x_1,\ldots,x_n)$ does there exist a satisfying assignment for ϕ?

Another way to phrase SAT is to say if the following is the formula $\exists x_1 \exists x_2 \exists \ldots \exists x_n \phi(x_1 \ldots X_n)$

We notice that we are only using one quantifier here. What if we allow both quantifiers?
Consider the formula: $\phi = \forall x \exists y[(x \vee y) \wedge (\bar{x} \vee \bar{y})]$
This is called a fully quantified boolean formula. A fully quantified boolean formula is also called a sentence and it is either true or false. Now we are ready to define TQBF.

TQBF $\{\langle\phi\rangle$: ϕ is a true fully quantified boolean formula $\}$

The first thing to notice is that TQBF is in PSPACE. To prove this we have to give a polynomial space algorithm that solves this problem. This is almost trivial once we define when a QBF is true.

If ϕ is a formula on 0 boolean variables then we can evaluate it to find out if it is true. For example $(0 \vee 0) \wedge (1 \vee 0)$ is false and $(1 \vee 0) \wedge (1 \vee 1)$ is true.

Let $Q_1x_1 \ldots Q_nx_n\phi(x_1 \ldots x_n)$ be a boolean formula on n variables, where $Q_i \in \{ \forall , \exists \}$. We can obtain two formulas $\phi|_{x1=0}$ and $\phi|_{x1=1}$ by first substituting $x_1 = 0$ in ϕ and then substituting $x_1 = 1$ in ϕ.

Let us work out an example. Consider

For $\Phi = \forall x \exists y \left[(x \vee y) \wedge (\bar{x} \vee \bar{y}) \right]$

We have $\Phi|_{x=0} = \exists y \left[(0 \vee y) \wedge (\bar{0} \vee \bar{y}) \right]$

And $\Phi|_{x=1} = \exists y \left[(1 \vee y) \wedge (\bar{1} \vee \bar{y}) \right]$

We define $\forall x_1\phi$ to be true if both $\phi|_{x1=0}$ and $\phi|_{x1=1}$ are true.
We define $\exists x_1\phi$ to be true if both $\phi|_{x1=0}$ and $\phi|_{x1=1}$ is true.

and both these must be true for the formula to be true.

To check $\Phi|_{x=0} = \exists y \left[(0 \vee y) \wedge (\bar{0} \vee \bar{y}) \right]$

We have to check $\Phi|_{x=0,y=0} = \left[(0 \vee 0) \wedge (\bar{0} \vee \bar{0}) \right]$

Which evaluates to false so we need to check the other one!

$$\Phi|_{x=0,y=1} = \left[(0 \vee 1) \wedge (\bar{0} \vee \bar{1}) \right]$$

This shows us that is true and we get

$$\Phi|_{x=0} = \exists y \left[(0 \vee y) \wedge (\bar{0} \vee \bar{y}) \right]$$

is true.

Both true for the formula to be true. We have checked the first one and now you can easily check the second one yourself and figure out if the original formula is true.

Note that we have given a definition of when a QBF is true. This definition leads to a recursive algorithm. Let us write down this algorithm:

Eval(ϕ, n)

1. if n = 0 then ϕ has no variables. Directly evaluate if and return the value.
2. If n > 1 then let x_1 be the first quantified variable in ϕ.
3. If $\phi = \exists x_1 \Psi$
4. Return (Eval($\phi|_{x1=0}$, n -1) $\vee$ Eval($\phi|_{x1=1}$, n - 1))
5. If $\phi = 8x_1$
6. return (Eval($\phi|_{x1=0}$, n _ 1)^ Eval($\phi|_{x1=1}$, n - 1))

Note that this is a recursive algorithm with n as its depth of recursion. It only stores a polynomial amount on the stack each time. Thus this algorithm runs in polynomial space. So this problem is in PSPACE.

Next our goal is to show that TQBF is PSPACE-hard. This is not going to be easy as we have to show that for every language A $\in$ PSPACE A $\leq_p$ TQBF.

Let us start. We do not know much about A. The only thing that we know is that it belongs to PSPACE. That means there a TM M that accepts A in polynomial space. Let us take that TM M and collect a few facts about it.

Firstly we observe that M runs in space $O(n^k)$. Therefore, it accepts in time at most $2^{O(nk)}$.

Suppose x is an input and C_0 is the initial configuration of M on x. We know that M accepts x if an accepting configuration of M is reachable from C_0 in at most $t = 2^{O(n^k)}$ steps.

Thus M accepts x if and only if there is a accepting history of M on x of length at most $2^{O(n^k)}$. Can we use the same idea as given in the Cook-Levin theorem?

In the Cook-Levin theorem each configuration was of length polynomial and the length of the computation history was also polynomial. In this case, the length is a problem. Nevertheless those ideas are going to help us. So you must review the proof of the Cook-Levin theorem in order to understand this proof completely.

A formula: $\Phi = \forall x \exists y[(x \vee y) \wedge (\bar{x} \vee \bar{y})]$.

This is called a fully quantified boolean formula. A fully quantified boolean formula is also called a sentence and it is either true or false. Now we are ready to define TQBF.

TQBF{< Φ > : Φ is a true fully quantified boolean formula g.}

The first thing to notice is that TQBF is in PSPACE. We proved this last time.

Our goal now is to show that TQBF is PSPACE-hard. This is not going to be easy as we have to show that for every language A $\in$ PSPACE A $\leq_p$ TQBF

We know is that it belongs to PSPACE. That means there a TM M that accepts A in polynomial space. Firstly we observe that M runs in space O(nk). Therefore, it accepts in time at most $2^{O(n^k)}$

Thus M accepts x if and only if there is a accepting history of M on x of length at most $2^{O(n^k)}$. We will now try to construct a QBF ΦM(x) such that Φ is true if and only if M accepts x. Furthermore (and this is the bulk of the proof) the formula will have polynomial length and would be computable in polynomial time.

Firstly, we know from the proof of the Cook-Levin theorem that we can encode configurations in boolean variables. Let use this as a black box. We could make a formula on variables X such that Con(X) is true if and only if the assignment of the variables X corresponds to a configuration.

Thus we will refer to configurations (with a bit of reminders). Strictly speaking when we talk about configurations we are talking about the underlying variables.

Now, recall that given C_1 and C_2 with corresponding variable sets X_1 and X_2 that we can make a formula yields $Y_M(X_1 , X_2)$ Which is true if C_1 yelids C_2 in one step. This is where we used the windows of size six idea.

We could make also make a formula that checked if an configuration was accepting. That is given C with corresponding variable sets X we could make a formula accept(X) which was true if and only if C was an accepting configuration.

Lastly given x and a configuration C with corresponding varaible set X we could make a formula $initial_M(X)$ which was true if and only if C was the initial configuration of M on x.

We simply put these ideas together. Our last formula was simply saying that does there exist $C_0,.....,C_m$ such that

1. Each C_i is a valid configuration.
2. C_0 is the initial configuration of M on x.
3. C_i yeilds C_{i+1}
4. C_m is a accepting configuration.

What if we do exactly the same? What goes wrong? Well m = $2^{O(n^k)}$. That is very large and not polynomial time. So we will use what is called a folding trick.

Recall that we have the quantifier $\forall$, $\exists$ that we did not have before. Let us use it. Let us make the formula $Y(C_1, C_2, k)$ if C_1 yeilds C_k in at most k steps. Now, $Y(C_1, C_2, 0)$ and $Y(C_1, C_2, 1)$ we know how to make.

We observe that $Y(C_1, C_2, k) \leftrightarrow \exists C" Y(C_1, C", k=2) \wedge Y(C", C_2, k=2)$

This is great! We can now make this formula recursively. But let us not rejoice. Let us see how long will this formula will be?

Let us say the length of each $Y(C_1, C_2, 0)$ and $Y(C_1, C_2, 1)$ is r .
Then the length of $Y(C_1, C_2, k) \leftrightarrow \exists C" Y(C_1, C", k/2) \wedge Y(C", C_2, k/2)$ will be?
Let us compute the length l_k $Y(C_1, C_2, k)$ We have $l_0 = r$, $l_1 = r$, and $l_k = 2l_{k/2} + O(1)$. Thus $l_k = rk$ at least! This is no good. Because, we want this to work for $k = 2^{O(n^k)}$. We need a new idea!

Remember that we have the quantified 8. Furthermore, let us look at the recursion again. Then the length of $Y(C_1, C_2, k) \leftrightarrow \exists C" Y(C_1, C", k/2) \wedge Y(C", C_2, k/2)$. If we can some how use 8 to make this recursion less expensive we would be home.

Consider $Y(C_1, C_2, k) \leftrightarrow \exists C" Y \forall (C, D) \in \{(C_1, C"), (C_1, C')\}$ $Y(C, D, k/2)$ this is equivalent to $Y(C_1, C_2, k) \leftrightarrow \exists C" Y(C_1, C", k/2) \wedge Y(C", C_2, k/2)$.

Again the length of each $Y(C_1, C_2, 0)$ and $Y(C_1, C_2, 1)$ is r.
Then the length of $Y(C_1, C_2, k) \leftrightarrow \exists C" \forall (C, D) \in \{(C_1, C"), (C_1, C')\}$ $Y(C, D, k/2)$will be?

Let us compute the length l_k $Y(C_1, C_2, k)$ We have $l_0 = r$, $l_1 = r$: and $l_k = l_{k=2} + O(r)$.

Thus $l_k = O(r \log k)$ at least. If $k = 2^{O(n^k)}$. This is polynomial in length. Once we have done this the proof becomes almost trivial.

Given x construct a formula

$$Y(C_i, C_f, 2^{O(n^k)}) \wedge \text{initial}_M(C_i) \wedge \text{accept}_M(C_f)$$

This QBF has now polynomial length and furthermore it is true if and only if M accepts x. Thus we have reduced A to QBF.

A formula like:

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

This is called a fully quantified boolean formula. A fully quantified boolean formula is also called a sentence and it is either true or false. Now we are ready to define TQBF.

TQBF {<Φ> : Φ is a true fully quantified boolean formula}

We proved

Theorem: TQBF is PSPACE-complete.

We want to now ask if there are other PSPACE-complete problems.

The answer is yes. Let us look at a problem that is very closely related to TQBF. Suppose we are given a QBF.

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots\ldots \exists x_n \psi.$$

We can think of a game that we call FORMULA-GAME.

In this game there are two players. Player I and Player II. The game proceeds as follows:

1. Player one assigns a value to $x_1 \in \{0, 1\}$.
2. Player one assigns a value to $x_2 \in \{0, 1\}$.
3. Player one assigns a value to $x_3 \in \{0, 1\}$.
4. Player one assigns a value to $x_4 \in \{0, 1\}$.
5. and so on.

In general:

1. If x_i is existentially quantified Player one assigns $x_i \in \{0, 1\}$.
2. If x_i is existentially quantified Player one assigns $x_i \in \{0, 1\}$.

At the end of n rounds all the variables have been assigned. Now Player I wins the game if ψ is made true for the assignment otherwise Player II wins.

This kind of game is called a two player perfect information game. An interesting question is to ask who will win this game? We have to pose this question carefully. The question asks which player will win this game if both players played perfectly.

Let us look at a simple example. Let us look at the formula $\psi(x_1, x_2, x_3)$. Let us say there are three moves. Player I chooses x_1 and then player II chooses x_2 and Player I chooses x_3. At the end of the game we have an assignment of the variables. If ψ becomes true then Player I wins. Otherwise Player II wins.

Now player one wants to make the formula true on the other hand player II is trying to make it false. If Player I wants to win then he should find an x_1 such that for any choice of x_2 he can find an x_3 such that ψ is true. Thus determining if Player I wins is equivalent to determining if the QBF $\exists\, x_1\, \forall\, x_2\, \exists\, x_3\, \psi(x_1, x_2, x_3)$ is true or not.

In general if there are many variables and have a QBF: $Q_1x_1,\ldots..,Q_nx_n\psi(x_1,\ldots,x_n)$. Where two players play n rounds and in the i-th round if $Q_i = \exists$, then Player I chooses the value of x_i otherwise Player II chooses the value of x_i .Then Player I has a winning strategy if and only if the QBF in question is true.

In general if there are many variables and have a QBF: $Q_1x_1,\ldots..,Q_nx_n\psi(x_1,\ldots,x_n)$.
Where two players play n rounds and in the i-th round if $Q_i = \exists$ then Player I chooses the value of x_i otherwise Player II chooses the value of x_i . Then Player I has a winning strategy if and only if the QBF in question is true. We call this game the game defined by Φ or we simply game the game Φ.

Every QBF Φ defines a formula.

Define, FORMULA-GAME = {<Φ>1: the game is won by Player I}. The computational question is given a game decide which player has a winning strategy. The observation that Player I wins if and only if Φ is true gives us the following theorem.

Theorem: FORMULA-GAME is PSPACE-complete.

In general there are many two player perfect information games. Most board games are like that. For example chess, checkers, Go etc. But most card games are not perfect information games. Let us think about chess.

Let us think a bit about chess. Suppose there someone were to find a winning strategy for chess. This means that whomever they play with they will win the game. This means they know a move such that no matter what the opponent does they have a move such that no matter what the opponent does the master wins. No such strategy is known for chess.

However many combinatorial games are known to be PSPACE-complete.

Let us look at a game called generalized geography. First let us recall the game geography.

Two players play the game.

1. Player I starts the game and says the name of a city/town or geographical region.
2. Player II does the same thing. The rule is player II must say a name which starts with the one that the pervious name ended with.
3. Player I does the same thing. The rule is player I must say a name which starts with the one that the pervious name ended with.
4. and so on.

Repeats are not allowed. The first player to “get stuck” loses.

Suppose that we take all the names of geographical regions and make a graph. Each vertex of the graph has a label. Two vertices a and b are connected if and only if the last letter of a is the same as the first letter of b. Here is a part of the graph.

Picture of GG graph.

Then the game can be thought of as being played on a graph. Player I chooses a vertex. Player"s continue to select the neighbors of the current vertex (without repeating vertices) untill one of them fails and loses.

This game is called Generalized Geography (GG). Formally, GG = {< G >: Player I wins on G}: The computational question again is given a directed graph G if Player I has a winning strategy.

Let us see how the winning strategy of Player I would be like? Player I has to choose a vertex such that for all choices of Player II he can choose a vertex such that for all choices of Player I. Player I wins.

So the questions seems to be just like alternating quantified formula. But is it?

We would like to show the following theorem:

Theorem: TQBF $\leq_p$ GG thus GG is PSPACE-hard.

along with this theorem we can easily see that...

We would like to show the following theorem:

Theorem: GG is in PSPACE.

Theorem: GG is in PSPACE-complete.

We will show this theorem next time. Let us also discuss some very interesting results from combinatorial games. It is known that appropriately formalized versions of many games are PSPACE-complete. These include

1. Chess
2. Go
3. Hex

and other board games.

Let us look recall the game called generalized geography.
Two players play the game.

1. Player I starts the game and says the name of a city/town or geographical region.
2. Player II does the same thing. The rule is player II must say a name which starts with the one that the pervious name ended with.
3. Player I does the same thing. The rule is player I must say a name which starts with the one that the pervious name ended with.
4. and so on.

Repeats are not allowed. The first player to “get stuck” loses.

Suppose that we take all the names of geographical regions and make a graph. Each vertex of the graph has a label. Two vertices a and b are connected if and only if the last letter of a is the same as the first letter of b. Here is a part of the graph.

Picture of GG graph.

Then the game can be thought of as being played on a graph. Player I chooses a vertex. Player"s continue to select the neighbors of the current vertex (without repeating vertices) until one of them fails and loses.

To avoid trivialities we will require the Player I to start with a designated node.

This game is called Generalized Geography (GG). Formally,
GG = {<G,b>: Player I wins on G when the game is started with b}. The computational question again is given a directed graph G if Player I has a winning strategy.

Let us see how the winning strategy of Player I would be like? Player I has to choose a vertex such that for all choices of Player II he can choose a vertex such that for all choices of Player I. Player I wins.

So the questions seems to be just like alternating quantified formula. But is it?
We would like to show the following theorems

Theorem: GG is in PSPACE.

Theorem: GG is PSPACE-hard.

A simple recursive algorithm for GG can be based on the following observation.

Theorem:
Suppose G is a graph and Player I chooses the node b in the graph. Let G_0 be the graph with b deleted from G and let $b_1,\ldots\ldots,b_k$ be the neighbors of b: Player I has a winning strategy on G with this first move as b if and only if Player I does not have a winning strategy with b_i as a move for G_0 for all i.

This observation can be converted into a simple recursive algorithm for GG.

The algorithm requires polynomial space. Although it would not be a polynomial time algorithm. This shows that GG is in PSPACE.

Theorem: FORMULA-GAME $\leq_p$ GG.

Since we know that FORMULA-GAME is PSPACE-complete therefore GG is also PSPACE-complete.

To do this we will show that given a QBF Φ we can make a graph G_Φ such that Φ is ture if and only if Player I wins G_Φ.

Let us take a formula Φ given as $\Phi = \exists x_1 \forall x_2 \exists x_3,\ldots..Qx_k\psi$.

We will assume

1. The quantifiers alternate.
2. Is in CNF.

The graph G_Φ is going be in two parts. Let us look at the first part.

1. There are n diamond structures one for each variable.
2. The start node is b.

Let us have a look at this part of the graph.

The game starts with the node b.

1. Player I chooses T_1 or F_1.
2. Player II must now choose z_1.
3. Player I now must choose b_2.
4. Player II can now choose T_2 or F_2.
5. Player I must now choose z_2.
6. Player II now must choose b_3.
7. Player I can now choose T_3 or F_3.

We observe that

1. If i is odd the Player I chooses T_i or F_i .
2. If i is even the Player II chooses T_i or F_i .

All other moves are forced.
This has the following simple interpretation.

1. Player I is choosing the existentially quantified variables.
2. Player II is choosing the universally quantified variables.

The game eventually ends at the node c. By adding an extra node before c we can make sure that it is now Player II"s turn. Let us look at the rest of the graph.
From c there are edges going to $c_1,\ldots\ldots,c_m$ where each c_i is a clause of ψ.

From clause nodes there are edges to nodes labeled by the literals of the clause. Each clause "has its own copy of the literal"

1. From the literal x_i there is an edge to T_i.
2. From the literal $\overline{x_i}$ there is an edge to F_i.

Here is a picture for clause $c_1 = (x_1 \vee \overline{x_2} \vee x_3)$.

Here is the structure of the whole graph G_Φ.

Let us consider the game from Player II"s point of view when the current vertex is c. Note that the choices of the players dictate an assignment.

1. He would like to choose a c_j such that he can win from c_j.
2. Player I will move to some literal in the clause.
3. Player II can only win if that literal is made false by the assignment.

Thus Player II is trying to move to a literal such that. All literals in the clause are made false by the assignment. Thus Player II is going to move to a clause that is made false by the assignment. Thus Player II is going to win if there is a false clause. Otherwise, player I wins.

So, while choosing the assignment Player I will try to make ψ true and Player II will try to make ψ false. This shows that

Theorem: Player I wins G_Φ starting with b if and only if Φ is true.

This establishes our reducibility. Thus we have shown that

Theorem: GG is PSPACE-complete.

Let us now consider some new complexity classes. Let us define

1. L = SPACE(log n)
2. NL = NSPACE(log n):

These turn our to be very interesting space complexity classes.

Notice that to count upto n we need a log n-bit counter. Thus you should think of L as the set of problems that can be solved using a fixed number of counters. NL is offcourse the non-deterministic analogue of this class.

We know that $L \subseteq NL \subseteq P$

Furthermore, we know from Savatich"s theorem that $NL \subseteq SPACE(\log^2 n)$.
However, Savatich"s theorem does not show us that NL = L.

To study L and NL polynomial time reductions are useless. Thus we define a new reduction. Let us make some definitions.

A LOGSPACE transducer is a TM with an input tape, a work tape and an output tape. The input tape is read only. The output tape is write only. The machine on input x of length n is allowed to use O(log n) work tape and produce its output.

Note that

1. The input is not counted as space used by the TM.
2. Only the space used on the work tape is counted as space used by the TM.

Let us design a LOGSPACE transducer that on input x compute $1^{|x|^2}$. This is very easy.

1. On input x
2. Set counter=0;
3. For each character of the input increment the counter.
4. For i = 1 to counter.
5. For j = 1 to counter.
6. print 1.

This machine uses only three counters. They need to be log n bit counters on input of length n. The output is clearly n^2 bits.

It is easy to argue that if a log space transducer produces an output f(x) then $|f(x)| \leq |x|^k$ for some k.

We say that a language A is LOGSPACE reducible to B if and only if there is a LOGSPACE transducer that computes a function f(x) such that $x \in A$ if and only if $f(x) \in B$. In this case we will write $A \leq_L B$.

We would like to show the following theorem.

Theorem: If $B \in L$ and $A \leq_L B$ then $A \in L$:

This is not trivial. Let us try to give a simple solution for this problem and see why does it not work.

We know that $A \leq_L B$ thus there is a log space transducer N computing f such that $x \in A$ if and only if f(x) is in B. Furthermore, we know that $B \in L$ so there is a TM M that uses O(log n) space and accepts B.

Let us consider the following TM T

1. Input x.
2. Using N compute y = f (x).
3. Run M on y and accept if M accepts y.

It is clear that T accepts x if and only if $x \in A$. But how much is the space used by T.

The first step is to compute f (x) and write it down. Although, it takes O(log n) space to compute f (x) but writing down f (x) can require a lot of space. Thus this is NOT a log space TM. We need a new idea.

The new idea is to use what I call the man in the middle attack! Let us see what the man in the middle attack is. Talk about man in the middle attack.

We observe that a TM reads one character at a time and then performs one step. Thus in order to simulate the TM M we need to know what is the current symbol under the tape head. Not the entire y.

We can now simulate M as follows. We find out what is the current symbol of y that is being scanned by M. To perform one step of the simulation we can rerun N on x to compute the symbol of y that is needed to take one step of M.

Thus if M needs the i-th symbol of the y we can run N on x and every time it produces a counter we increment it. We discard the output by erasing it if the counter is less than i. This way the output does not use any space. When the i-th bit is produced we use it to do one step in our simulation.

This way by using a few extra counters the space requirement of this TM is O(log n) + O(log y). The length of y is at most n^k thus the whole procedure takes log-space.

Thus we have proved our theorem.

Theorem: If $B \in L$ and $A \leq_L B$ then $A \in L$.

Notice that we would not be able to prove this theorem if we were using polynomial time reducibility. Thus log-space reducibility is what is needed to study L and NL.

A central question in space complexity is if L = NL. This is a major unresolved question in space complexity. Solution to this problem means instant fame and glory!

This problem is very interesting we can define the classes

co – P = { $\overline{L}$: L ∈ P}

co – NP = { $\overline{L}$: L ∈ NP}

It is easy to see that P = co – P. However, many people conjecture that NP ≠ co – NP.

It is easy to see that

Theorem: If co – NP ≠ NP then P ≠ NP.

Thus NP ≠ co - NP is a stronger conjecture than P ≠ NP conjecture. One way to resolve the P vs. NP conjecture is to show that NP ≠ co – NP.

On the other hand if someone proves that NP = co – NP that would be a great insight but would not resolve the P vs. NP conjecture.

We can play the same game with space complexity. Define,

co – L = { $\overline{L}$: L ∈ L}

co – NL = { $\overline{L}$: L ∈ NL}

It is easy to see that L = co – L. However, it is not clear if NL = co – NL.

Theorem: If co – NL ≠ NL then L ≠ NL.

Thus NL ≠ co – NL is a stronger conjecture than L ≠ NL conjecture.

One way to resolve the L vs. NL conjecture is to show that NL ≠ co – NL. However, there is a twist in the tale. We will prove that NL = co – NL. This is a major result in space complexity. However, the question L = NL is still wide open! Lets first start by looking at a problem

Reach = {<G, s, t> : there is a path from s to t in G}.

Lets first observe that Reach in NL. We have to give a non-deterministic algorithm for Reach.

Theorem: NL = co-NL

We will show that

1. There is a problem PATH that is NL-complete with respect to LOGSPACE reducibility.
2. We will show that $\overline{PATH}$ is in NL. This means that PATH is in co-NL.

Theorem: A $\leq_L$ B if and only if A $\leq_L$ B

This implies that $\overline{PATH}$ is co – NL complete. Since we will also show that $\overline{PATH}$ is in NL. This would tell us that NL = co – NL.

Let G be a directed graph and s and t be two designated vertices in G.
Let PATH = {<G, s, t>: there is a path from s to t in G}.

Thus given two vertices in a graph we have to decide if there is a path from s to t in a graph. This problem is also called s – t-connectivity.

Savatich"s theorem tell us that PATH $\in$ DSPACE($\log^2$ n)

We want to show that
1. PATH is in NL.
2. PATH is NL-complete.

To see that PATH is in NL we recall that non-determinism allows us to guess. Thus given s and t we can try to guess the path from s to t and accept if we find one. However, we are only allowed to use small amount of space. So we have to be a bit more careful.

Consider the following non-deterministic algorithm.
1. Input G = (V, E) and s, t $\in$ V.
2. Set current vertex v = s
3. For i = 1,.....,n
4. Guess a vertex w $\in$ V;
5. if (v, w) $\notin$ E reject;
6. else if w = t accept;
7. v = w;

The space used by this algorithm is the space to store i and v the current vertex. The maximum value of i is n and there are n vertices in the graph. Thus the total space used in O(log n).

The algorithm is trying to „guess" a path. The trick is that it does so by "guessing" one vertex at a time. It does not bother to remember the previous vertices of the path thereby saving space.

Now we would like to show that PATH is NL-complete. Let A $\in$ L then there is a non-deterministic TM M that accepts A.

On an input x the NTM M performs a computation. Let think of making a graph as follows:
1. The vertices in the graph are the configurations of M on the input x.
2. A configuration c us connected to c_0 if and only if c yields c_0 in one step.

To find out if M accepts x all we have to is to find out if from the vertex that corresponds to the initial configuration can we reach a vertex that corresponds to the final configuration.
Note that each configuration c of a NTM is specified by
1. The contents of the work tape.
2. The position of the head reading the work tape.
3. The position of the head reading the input tape.

Thus all configurations are given by c = (w, i, j) where w is the the contents of the work tape and the machine is currently reading the i-th symbol on the work tape and the j-th symbol of the input.

Since M runs in LOGSPACE thus all such configurations can be enumerated in LOGSPACE. It is also easy to see that for any given configuration c we can compute in LOGSPACE the configurations reachable from c in one step. Thus we can output the edges of this whole graph.

Let us denote by $G_M(x)$ the graph described above. Let us consider the following algorithm.

1. On input x.
2. Compute $G_M(x)$
3. Output $<G_M(x), c_i, c_f>$

Where c_i is the initial configuration of M on x and c_f is the final configuration.
This is a reducibility that shows that $A \leq_L$ PATH: This implies that PATH is NL-complete.

Next we will show that $\overline{PATH}$ is in NL. This is not a simple theorem. We have to first understand the problem very carefully.

Our goal is to design a non-deterministic TM (algorithm) that takes as input a graph G and two vertices s and t. The TM M must have the following properties:

1. If s is connected to t then all branches of M must reject.
2. If s is not connected to t then at least one branch of M must accept.
3. M must run in (non-deterministic) LOGSPACE.

What is the major difficulty? The problem is that non-deterministic TM"s are very good at finding things. However, this time a non-determnistic TM has to accept is there is no path from s to t. So, it is a counter intuitive problem from the point of view of non-determinism.

This is a difficult problem to solve. So, we start with solving a much simpler problem. Let us say we are given the following information.

1. We are given an input graph G = (V, E) and two vertices s and t.
2. We are told a number c. This is the number of vertices in G that are reachable from s.
3. We have to design a LOGSPACE NTM.
4. The TM will reject all inputs where s is connected to t.
5. At least one branch of M will accept if s is not connected to t.

This problem is easier to solve. Let us look at a overview of a non-deterministic TM that solves this problem and then we will improve it.

1. Input G, s, t and a number c (It is promised that there are exactly c vertices that are reachable from s)
2. Guess c vertices $v_1,\ldots\ldots,v_c$
3. if t = v_i for some i reject.
4. For i = 1,……..,c
5. Guess a path from s to v_i
6. Verify the path from s to t by checking edges of G.
7. If the path is not valid reject.
8. Accept.

This machine solves the problem. Lets see why! For it to reach accept state it would verify that at least c vertices are reachable from s and t is not one of them. Furthermore, if t is not reachable from s and the machine makes all the right guesses it will accept. Thus at least one branch will accept if and only if t is not reachable from s. But how much space does this algorithm take?

There are two problems in the algorithm.

1. It has to guess and remember $\{v_1, ... v_c\}$.
2. It has to guess a path from s to v_i and remember and verify it.

This is too space wasteful. So, we use the previous trick that we used to solve PATH. We do not need to guess the whole set $\{v_1,.....v_c\}$ and similarly we do not need to guess the whole path. We can guess them as we go along. Thus saving space. Let us look at these ideas clearly.

Let $V = \{v_{1,}, v_n\}$ be a set of vertices. What we want to do is think about choosing a subset of size c from these vertices. A subset can be encoded in a 0/1 vector. Let $(g_{1,.....,}g_n)$ be a 0/1 vector. Then $G = \{v_i, g_i = 1\}$ is a subset of V.

What we can do is guess the current bit of this vector one at a time. Thus if $g_i = 1$ we are guessing that the vertex is in G otherwise not. Also, as we did previously we can forget all the previous bits and only remember the number of them which were in G this way we will know the size of G at the end of the loop.

1. Input G, s, t and a number c (It is promised that there are exactly c vertices that are reachable from s)
2. Set d = 0.
3. For each vertex $x \in V$.
4. Guess a bit g.
5. If g = 1 (The algorithm has guessed that x is connected to s).
6. Set current vertex v = s;
7. For i = 1, ...,n
8. Guess a vertex $w \in V$;
9. if $(v, w) \notin E$ reject;
10. else if w = x exit the For i loop.
11. v = w;
12. if x = t reject;
13. d = d + 1;
14. if d = c then accept.

The previous algorithm tells us that given a graph G = (V, E) and a source s if we can compute in LOGSPACE a number of vertices reachable form s then we can solve the $\overline{PATH}$ problem in non-deterministic LOGSPACE.

Now, our task is a bit simpler. But we notice that it is even simpler than we thought. What we want is an non-deterministic algorithm that does the following.

1. On input G and s.
2. At least one branch of M should output a number c the number of nodes reachable from s.
3. All other branches should reject.

If we can accomplish the previous task we could run our algorithm at the branches that output c and get our algorithm. However, solving this problem requires another similar idea. We will study that idea in the next lecture.

Recall that our goal was to prove that

Theorem: NL = co – NL

We simplified this problem by observing that

Theorem: PATH is NL-complete (with respect to logspace reducibility).

Thus our goal was just to prove that

Theorem: $\overline{PATH}$ is in NL.

Thus by showing that only one problem belongs to NL we could show that two complexity classes were equal.

In the previous lecture we studied an non-deterministic algorithm. The algorithm almost did the job. But it required an extra piece of information.

1. We are given an input graph G = (V, E) and two vertices s and t.
2. We are told a number c. This is the number of vertices in G that are reachable from s.
3. The TM will reject all inputs where s is connected to t.
4. At least one branch of M will accept if s is not connected to t.

The previous algorithm tells us that given a graph G = (V, E) and a source s if we can compute in LOGSPACE a number of vertices reachable form s then we can solve the $\overline{PATH}$ problem in non-deterministic LOGSPACE.

Now, our task is a bit simpler. But we notice that it is even simpler than we thought. What we want is an non-deterministic algorithm that does the following.

1. On input G and s.
2. At least one branch of M should output a number c the number of nodes reachable from s.
3. All other branches should reject.

Today we will design this algorithm. Let us start. We first break up the problem into simpler pieces (by now you should be used to of this! Given a graph G and a vertex s let us define:

R_i = {v : there is a path from s to v of length at most i}: and define $c_i = |R_i|$.

1. R_0 is the set of all vertices reachable from s via path of length 0.
2. R_1 is the set of all vertices reachable from s via path of length 1.
3. R_2 is the set of all vertices reachable from s via path of length 2.
4. R_3 is the set of all vertices reachable from s via path of length 3.
5. and so on.

What we want to do is to find out the value of c_n.

Let us now note that $c_0 = 1$ and in fact, the only vertex reachable from s in 0 steps is s itself. Now, suppose we can design an algorithm that does the following. Given c_i as an input it at least one of its branch computes c_{i+1} correctly and all other branches reject. Then we can use this algorithm to eventually come up with an algorithm that computes c_n from c_0.

This algorithm is going to be a simple loop.

1. $c_i = 0$.
2. for i = 1,……,n – 1
3. Compute c_{i+1} from c_i.
4. if the computation succeeds continue.
5. Output c_n.

A few things about the algorithm. The indices are only there for clarity. We can forget the previous c_i "s so the space requirements are logarithmic.

Now we show how we can compute c_i+1 from c_i. For that let us try to see why a vertex will be in R_{i+1}. We observe that a vertex x is in R_{i+1} if and only if

1. It is in R_i or
2. There is a vertex $w \in R_i$ and (w, x) is an edge in the graph.

Lets start with a simple idea and then refine it. We will use the previous ideas again and again.

What we want to do is to count the number of vertices in R_i+1. There is a simple way to do this....

1. count = 0;
2. For each vertex $v \in V$
3. if $v \in R_i$ +1 count = count+1;

The only problem is to check if $v \in R_{i+1}$. We do not know R_{i+1}.

Now, given a vertex $v \in V$ all we want to do is to check if $v \in R_{i+1}$. We can use the previous observation and do this job as follows:

1. For each $w \in V$
2. If $w \in R_i$ then
3. if w = v or (w, v) an edge then $v \in R_{i+1}$.
4. $v \notin R_{i+1}$.

This idea does not work! We do not know Ri but we do know its size c_i. So, we do something cleverer.

We are given non-determinism at our disposal. So, we can use it over and over again. Let us try something.....

1. For each $w \in V$
2. Guess if $w \in R_i$
3. Check if $w \in R_i$ if not reject.
4. if w = v or (w, v) an edge then $v \in R_{i+1}$.
5. $v \notin R_{i+1}$.

The problem is how do we check if $w \in R_i$. Here we use our old trick.

Note that our subset R_i can be specified by n-bits (r_1,…….,r_n), where $r_i = 1$ if and only if $v_i \in R_i$. Suppose we could guess a bit vector (r_1, ……..,r_n).

Consider the following program.

1. count = 0
2. For w = 1,.....,n
3. Guess $r_w \in \{0, 1\}$
4. If $r_w = 1$ then
5. l = s;
6. For j = 1,..... ,i
7. guess a vertex z
8. if z = l or (z, l) $\in$ E then l = z;
9. if l ≠ w then reject.
10. count = count+1;
11. if count ≠ c_i reject;

A simplified version of this program is as follows.

1. count = 0;
2. For w = 1,……,n
3. Guess $r_w \in \{0, 1\}$
4. If $r_w = 1$ then
5. Verify if there is a path of length ≤ i from s to w.
6. If verification fails reject.
7. count = count+1;
8. If count ≠ ci reject (the guess of r_v"s was wrong).

Now we can use this program to finish our task.

1. count = 0;
2. For w = 1,....., n
3. Guess $r_w \in \{0, 1\}$
4. If $r_w = 1$ then
5. Verify if there is a path of length ≤ i from s to v.
6. if w = v or (w, v) $\in$ E then v $\in R_{i+1}$ we can set a bit b = 1.
7. if verification fails reject.
8. count = count+1;
9. if count ≠ c_i reject (the guess of r_v"s was wrong);
10. if b = 1 then v $\in R_{i+1}$ else it is not.

This non deterministic program has the following property.

1. On input v and the number c_i.
2. At least one of the branch outputs a number b $\in$ {0, 1} such that b = 1 if and only if b 2 $\in R_{i+1}$.
3. All other branches reject.

Critically we can use this program to count the size of R_{i+1}. This is what we wanted to show. Lets look at this program again. It is a highly non-deterministic program. But it only uses counters! It is one of the most wonderful programs and has only theoretical implications.

Lets revisit it. Now, we will not use indicies anymore to make it clear that only counters are used.

1. C = 1; ($c_0 = 0$ we know that)
2. For i = 1,....., n - 1
3. C has $c_i = |R_i|$ and we want to compute $D = c_{i+1}$
4. D = 0;
5. For v = 1, ... ,n (For each vertex let"s check if it is in R_{i+1})
6. b = 0;
7. count = 0;
8. For w = 1,…..... ,n
9. guess $r_w \in \{0, 1\}$ if $r_w = 1$
10. verify if there is a path of length at most i from s to w.
11. If verification fails reject.
12. else count = count+1;
13. if v = w or $(v, w) \in E$ then b = 1;
14. if count 6= C reject.
15. D = D + b
16. C = D

At the end of the algorithm we have computed c_n. Its value is stored in D. Note that one of the loop is not shown in this program. The loop that guesses the path of length at most i. Thus this is a program with four nested loops! It is quite complicated.

Once we have computed c_n we know how many vertices are reachable from s. Now, we can use the program of the previous lecture to write our non-deterministic program that we require. This ends our proof.

Computability

The most important achievement of computability theory is a formal definition of an algorithm. The Church-Turing thesis asserts that the intuitive notion of algorithm is captured by the mathematically precise notion of a Turing machine. This allows us to formally talk about algorithms and prove things about computations and algorithms.

Another extremely important theorem is the existence of a universal TM

Theorem (Turing) There exists a TM U such that U(<M, x>) = M(x)

A major part of this proof is the realization that we can encode the description of a TM. Another important idea was the fact that we can make a TM that simulates another TM by reading it"s description.

We say that a language L is decidable if there exists an TM such that halts on all inputs and accepts L. This gives us a precise definition of the notion of algorithms. We realize that that the language accepted by TM does not seem to be decidable.

We define A_{TM} = {<M, x>: M accepts x }

and a closely related problem AH = {<M, x>: M halts on x }

We note that ATM is a the language accepted by the Universal TM. However, the universal TM is not a decider. Thus the natural question that arises is, Is ATM decidable? The main technique here was digitalization. We proved the amazingly interested theorem of Turing.

Theorem (Turing): A_{TM} is not decidable.

The amazing thing about this theorem is that it points out to something which probably would not have crossed our minds. The fact that well defined computational questions can be undecidable.

Based on the undecidability of the halting problem. We were able to prove that several other problems were undecidable. A generic result called **Rice's theorem** stated that all non-trivial semantic properties of TM"s are undecidable.

We looked at a very natural question that was undecidable. This problem seemed to be a puzzle but turned out to be undecidable. That was the Post Correspondence Problem. Thus problems which "naturally popup" can also be undecidable.

We then started with the following puzzle. Can one write a program that prints itself? We were able to answer this in the affirmative and not only that we were able to come up with a technique which told us that "A program can have access to its own description". This statement was formalized in the recursion theorem. The recursion theorem becomes a very nice tool to get undecidability results.

We also looked at problems that arise from mathematical logic. In particular we looked at the decidability of logical theories. We showed using a very non-trivial algorithm that Presburger arithmetic is decidable.

On the other hand we were able to show that Paeno Arithmetic (the one that we are used to of) is undecidable. This was a shocking and wonderful result.

We then looked at Kolomogrov or Descriptive complexity of strings. This gave us another fundamental definition. It gave us a definition of information. We proved several theorems in Kolmogrov complexity. We defined incompressible strings. Furthermore we showed that "most strings" are incompressible.

Lastly we showed that any property that holds for all most all strings must be false for finitely many strings that are incompressible. Thus incompressible strings look random from the point of view of any computable property.

Complexity

Then we changed our focus and started studying Complexity Theory. Thus now we were talking about

1. Problems those are solvable in practice.
2. Thus we study computable problems and ask what resources are required to solve these problems.

We precisely defined time and space complexity. We also noted that k-tape TM"s can be simulated by a 1-tape TM with quadratic delay.

We defined non-deterministic and deterministic time and space complexity. This lead us to two naturally defined classes. The class P and the class NP.

The class P captures the problem that can be solved in "practice". However, the class NP contains a lot of practical problem that we would like to solve. We asked the question if

Question: Is P = NP?

The P vs. NP question is one of the most celebrated open questions in computer science. It is considered to be a deep question whose resolution will enhance our understanding of computation in a fundamental way. The Clay mathematical society has declared it as the problem of the millennium.

The study of P and NP has lead to the theory of NP-complexness. When we prove a problem is NP-complete we show that if it can be solved in polynomial time then all the problems in NP can be solved in polynomial time. Thus it provides us with **hard** evidence that the problem is hard. However, it does not provide us with a **proof** (at least not till the P vs. NP conjecture is open).

One of the most spectacular theorem in this area was the Cook-Levin theorem.

Theorem (Cook-Levin): SAT is NP-complete.

This theorem can be read in two ways.

1. P = NP if and only if SAT$\in$ P.
2. SAT in some sense captures the complexity of the entire class NP.

Using the Cook-Levin theorem we were able to prove that several other problems are NP-complete. These problems came from graph theory (independent set, coloring, vertex cover), arithmetic (subset sums), logic (sat) etc. Thus NP-complete is a rich class of problems.

We defined non-deterministic and deterministic space complexity classes. The class PSPACE and the class NPSPACE.

The class PSPACE captures the problem that can be solved in reasonable space. The class NPSPACE was a natural non-deterministic analogue. We asked if

Question: Is PSPACE = NPSPACE?

Theorem (Savitch): PSPACE = NPSPACE:

The study of P and PSPACE lead to the theory of PSPACE-complexness. When we prove a problem is PSPACE-complete we show that if it can be solved in polynomial time then all the problems in PSPACE can be solved in polynomial time. Thus it provides us with **hard** evidence that the problem is hard. However, it does not provide us with a **proof**.

Theorem (Cook-Levin): TQBF is SPACE-complete.

We then showed that combinatorial games like FORMULA-GAME, Generalized Geography is PSAPCE complete.

In the last part of the course we studied the classes L and NL. These were space complexity classes where the space requirement was logrithmic. We again proved that path was NL-complete.

We noticed that Savatich"s techinique does not yield a result about L and NL. However, the big surprise was the last theorem we proved was NL = co – NL, this gave us an example of a non-deterministic complexity class that is closed under complementarily.

What more to study? What have we missed? Theory of computation is a wonderful subject and it is impossible to study it in one course. There are many exciting things that you can look at in the future.
These include:

1. Randomized Complexity Classes.
2. The polynomial time hierarchy.
3. Hardness of Approximation.
4. Quantum computation and quantum complexity.
5. Cryptography.
6. and the list goes on...

You can read papers in the following conference proceedings.

1. STOC
2. FOCS
3. CCC

A few journals that you may want to look at to see what is going on in complexity theory are

1. Journal of Computational Complexity
2. Theoretical Computer Science
3. Journal of ACM