A language class is a set of languages

Each language class has two properties
- Decision properties
- closure properties

(Usually we use the algorithms to find these properties.)

closure properties says that given a language in a class, an operation produces another language of the same class.

A language can be described in formal or informal way (such as $\{0^n 1^n \mid n \text{ is a natural}\}$) (defined by DFA or an FA)

You cannot answer a question about a language unless you have a formal description.

Thus a decision property is an algorithm that takes a formal description and tells whether some property holds

Property such as: is it finite? Can it fail? things that you cannot do with programs.

The membership property takes a string and tells whether it is accepted by the DFA. The algorithm simulates the DFA on the input. If the regular language is represented in an NFA or RE, we convert to DFA by following the circle of conversions

The emptyness property, given a regular language, does it contain any string at all?
- Assume it is represented by DFA
- Find reachable states by using BFS or DFS on the start state.
- If at least one final state is marked, then it accepts at least one input, else it is empty

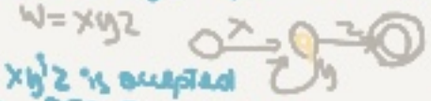
The infiniteness problem

- Start with DFA of the language
- key IDEA If the DFA has n states, and the language contains any string of length n or more then the language is infinite
- Otherwise is limite to n or less

10.06 on...

Proof of key IDEA "If the DFA accepts any string whose length is at least the number of states n, then it accepts an infinite number of strings"
- First observe that a string of length n or more has at least n+1 states along its path

If there are only n states, and there are n+1 states along the path, then two states along the path must be the same



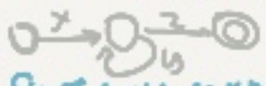
Then xy^iz is accepted by the DFA for any $i \geq 0$

Since y is not ϵ, the DFA can accept an infinite number of strings.

We still don't have an algorithm because we can't test the infinite number of strings of length $\geq n$

OTHER KEY IDEA It is sufficient to test strings between n and 2n-1

Proof



- y is the first cycle, so xy cannot be greater than n. some state in the first n+1 states is repeated.
- If w is of length 2n or more, there is still a shorter string of length $\geq n$
- Keep shortening to reach $[n, 2n-1]$

The algorithm tests for membership all strings of length between n and 2n-1. If it is accepted is infinite.

...

...

A more efficient algorithm:

- Eliminate states not reachable from start and find states
- Check for cycles in the remaining graph.
- If you don't find cycles, the language is finite

PUMPING LEMMA

For every language L, there is an integer n (number of states) such that for every string w in L of length $\geq n$ we can write $w = xyz$ such that
- $|xy| \leq n$
- $y > 0$
- For all $i \geq 0$, xy^iz is in L

(y is the label of the first substring of w that goes from the state to the same state)

- The prefix xy in w is at most n, and y is the label of the first cycle we encounter
- y is not ϵ because it connects two different occurrences of the same state along path of w
- xy^iz is in L for $i \geq 0$

Proof for containment

Given L and M, is $L \subseteq M$?
- Algorithm uses the product automaton
- We define q as final, but not r

Given a DFA, find the equivalent DFA with the fewest states
- Since we can test equivalence, given a DFA we can find the DFA with fewest states accepting L(A)

Algorithm, create a table with pairs of states and figure out which pairs are distinguishable (one goes to a final state and the other to a non.

Proof for equivalence
- Given regular languages L and M is $L = M$?
- Algorithm involves in constructing the product DFA for L and M
- Suppose have set of states Q and R
- Product has set of states for pairs, $[q, r]$ q from Q, r from R
- Start $[q_0, r_0]$
- Transitions $\delta([q,r],a) = [\delta_Q(q,a), \delta_R(q,a)]$
- Make the final state be all those pairs such that one is a final state and the other isn't. If w reaches only one final state, the two languages are not the same

---

STATE MINIMIZATION

Basis, Mark pairs with one final state

Induction, mark $[q,r]$ if for some input a, $[\delta(q,a), \delta(r,a)]$ is marked

After no more marks are possible, the unmarked pairs are equivalent and can be merged into one state

CLOSURE PROPERTIES

Given an operation between two languages, when the arguments are of a class, so is the result

- L M are regular, then L∪M is also regular
  Let R and S be the RE defined, then R+S is a RE whose language is L∪M
- RS is a RE whose language is LM — CONCAT.
- R^* is a RE whose language is R^* — KLEENE

- If L and M are regular, then so is L∪M
  "we cannot use REs very easily to prove this one"
  - construct C, the product automaton of A and B
  - make the final states be the pairs consisting of final states of both A and B

If L and M are regular, then so is L−M

- construct C ...
- make the final states be the pairs where A-state is final and B state is not

The complement of a language L is $\Sigma^* - L$

- Since we know that regular languages are closed and star and difference, we know that complement of is also closed.

HOMOMORPHISM

They are transformations on symbols which replace each symbol by a string

When a homomorphism is applied to a regular language, the result is a regular language

ie $h(0) = ab$, $h(1) = \varepsilon$
Apply it to any symbol in order and concatenate the results
$h(01010) = abababab$

Inverse homomorphism
$h^{-1}(L)$
The result is the set of strings w such that $h(w)$ is in L

CONTEX FREE

Is a class of languages

- Applications are processing natural language, and computer languages
  They'll essential with designing the parser of a language (in a compiler it runs tokens together into the proper structure)

---

context free languages are defined by context-free grammars

Every regular language has a context free grammar describing it, not all context free languages are also regular

Many of these non-regular languages, are languages that involve nested structures

---

The central elements of the grammar are variables, symbols that generate particular sets of strings

The variables, or the set of strings they generate, are defined recursively, in terms of one another

The rules that defines the language or the variables are called productions
It involves concatenation $A \to xy$ the concatenation of $L(x)$ and $L(y)$ is a subset of $L(a)$

A language may have several productions, it is defined as the union of the languages described by the right side of each production

eg. CFG for $\{0^n 1^n \mid n \geq 1\}$

productions
$S \to 01$ BASIS RULE[1]
$S \to 0S1$ INDUCTIVE RULE[2]

[1] 01 is in the language
[2] if w is in the language, then so is 0w1

CFG FORMALISM

Terminals

They are analogous to the input symbols of an automaton. They form the alphabet for the language being defined

Variables are non-terminals. They are something like the states of an automaton, but more powerful. The START symbol variable is the language of this variable that the grammar defines, and the other variables are used as auxiliaries

The productions of the grammar, which are akin to the transition function of an automaton, have the form variable (head) → string of variables and terminals (body)

CONVENTIONS

A, B, C... and S are variables
a, b, c... are terminals
..., X, Y, Z are either terminals or variables
..., w, x, y, z are strings of terminals only
α, β, γ... are strings of terminals and/or variables

eg CFG for $\{0^n 1^n \mid n \geq 1\}$
terminals 0, 1
variables {S} ← START SYMBOL
productions $S \to 01$ $S \to 0S1$

A derivation is a sequence of strings that typically have both terminals and variables.

We say $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if $A \to \gamma$ is a production

Recursive derivation
$\Rightarrow^*$ means zero or more derivation steps
BASIS $\alpha \Rightarrow^* \alpha$ for any string α
INDUCTION if $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$ then $\alpha \Rightarrow^* \gamma$

SENTENTIAL FORMS

For a grammar is any string derivable from the start symbol. Formally $S \Rightarrow^* \alpha$

The language of a grammar G is the set of terminal strings such that the start symbol derives w

a sentential form iff the production applies

eg G has productions $S \to \epsilon$ and $S \to 0S1$

$L(G) = \{0^n 1^n \mid n \geq 0\}$

CONTEX FREE LANGUAGES

A contex-f. language is a language defined by some context-free grammar.

Some languages are not contex-free!! this is because contex-f. grammars can count two things, but not three eg $0^n 1^n 2^n \mid n \geq 1$

BNF notation (Backus-Naur form)

Bgrammars, used for programming languages often

- In BNF you usually use a word to describe a variable. For example, statement for the intent is that this variable will generate all the strings that are valid statements for the programming language.
  Conventionally these words are put in between that tells they are variables
- Terminals are i + * while if

::= is used for reduction
| is used for lists productions with same left s eg $S \to 0S1 \mid 01$

... is similar to the * but means one or more, instead of zero or more

<digit> ::= 1|3|4|5|6|7|8|9
<unsignedint> ... <digit>...

TRANSLATION we can replace a... with a new variable A and productions $A \to Aa \mid a$
eg $V \to VD \mid D$ $D \to 0|1|2|3|4|5|6|7|8|9$

optional elements are added by surrounding with [ ]
eg <statement> ::= if <condition> then <statement> [; else <statement>]

TRANSLATION replace [a] by a new variable A with productions $A \to a \mid \epsilon$

$S \to iC\,tSA$ | i if | t then | e else
$A \to ;eS \mid \epsilon$

S stands for statement and C for conditions
A is an optional else
leaves; or ε otherwise

Use {} to group several different elements. eg <statement list> ::= <statement> [{; <statement>}...] ← optional one or more

For programming languages where you have a list of statements separated by a semicolon

TRANSLATE
To translate to our original notation
- Create a new variable A for {a}
- One production for A: $A \to a$
- Use A in place of a

eg $L \to S[A...]$ $A \to iS$
A stands for {; S}

$L \to SB$ $B \to A... \mid \epsilon$ $A \to ;S$
We introduce a new variable B

$L \to SB$ $B \to C \mid \epsilon$ $C \to AC \mid A$ $A \to iS$
We replace A... in B with the new variable C

Left-most and Right-most derivation

- Derivations to allow us to replace any of the variables in a string. Leads to many different derivations of the same string
- By forcing the left-most (or right-most) variable to be replaced, we avoid that "distinction without a difference"

Say $wA\delta \Rightarrow_{lm} w\beta\delta$ if w is a string of terminals only $A \to \beta$ is a production

Also $\alpha \Rightarrow^*_{lm} \beta$ if α becomes β by a sequence of zero or more $\Rightarrow_{lm}$ steps eg $S \to SS \mid (S) \mid ()$

$S \Rightarrow_{lm} SS \Rightarrow_{lm} (S)S \Rightarrow_{lm} (())S \Rightarrow (())()$

Say $\alpha A w \Rightarrow_{rm} \alpha\beta w$ if w is a string of terminals only $A \to \beta$ is a production

$S \to SS \mid (S) \mid ()$

$S \Rightarrow_{rm} SS \Rightarrow_{rm} S() \Rightarrow_{rm} (S)() \Rightarrow_{rm} (())()$

$S \Rightarrow^*_{lm} (())()$

$S \Rightarrow^*_{rm} (())()$

PARSE TREES

A parse tree is a graph that shows how some string is derived using some context-free grammar

They let us express the concept of ambiguity in grammars. Unambiguity, that the ability to provide a unique tree structure for every string in its language is vital.

interior nodes are labeled by a variable

leaves are labeled by either a terminal or by ε.

there is a production in the CFG with a node on the left and its children on the right
$A \to Bb$

The root is usually labeled as the start symbol. But not always (trees with root A)

The yield of a parse tree is the string of labels from the left

A CFG grammar is ambiguous if there is a string in the language that is the yield of two or more parse trees.

In a grammar where you can always figure out the production by scanning the given string left-to-right and looking at the next symbol is called LL(1) (Most programming languages)

There are equivalent unambiguous grammar for some language. When this is not posible, we say the language is inherent ambiguous.

NORMAL FORM

Because context-free variables can be badly designed. Analogous to the states of a finite automaton not reachable from the start state. There are productions that take many steps that obviously can be combined.

DISCOVERD ALGORITHMS

Almost all the algorithms used to find bad context-free grammars are based on the same principle

They are induction

- the basis are facts that are obvious
- discovers more facts from what it already discovered
- until no more facts can be discovered

Find variables that cannot derive terminals

Basis. If there is the production $A \to w$ where w has no variables, then A derives terminal string in one step (can be the empty string)

Induction. $A \to \alpha$ and α consists of terminals and variables previously discovered. Then A derives a terminal string

Every variable that derives a terminal string will be discovered

Algorithm to eliminate variables that derive nothing

- Find the variables that derive terminal strings
- For the other variables, remove all productions in which they appear in either head or body

Unreachable symbols

Another a terminal or variable deserves to be eliminated is if it cannot be derived from the start symbol

This algorithm is another example of the discovery algorithm

Basis, the start symbol can be reached

Induction. Suppose we discovered A, then a production $A \to \alpha$ says that we can reach all symbols in α

When we can discover no more symbols, then we have all symbols derivable from S. Remove all the other symbols.

Eliminate useless symbols

A symbol is useful if it appears in some derivation of some terminal string from the start symbol

It is useless otherwise

1. Eliminate all symbols that derive no terminal string
2. Eliminate unreachable symbols

ε productions

$A \to \varepsilon$, they can be eliminated from a context-free grammar.

The only thing is that we can no longer derive the empty string.

Theorem If L is a CFL, then $L - \{\varepsilon\}$ has a grammar with no ε-productions

Nullable symbols

We discover nullable symbols, which are variables A such that $A \Rightarrow^* \varepsilon$

Basis, if $A \to \varepsilon$, it is nullable

Induction, if $A \to \alpha$ and all symbols in α are nullable, then A is nullable

Eliminate ε productions

Turn each production $A \to X_1 \dots X_n$ into a family of productions

For each set of nullable X_i, we delete the nullable production and make a new production (except in the case all productions are nullable)

Unit productions

Where the body has a single variable.

They can be eliminated as well

If $A \Rightarrow^* B$ and $B \Rightarrow^* \alpha$ is a non-unit production

Then add the production $A \to \alpha$

Then drop all unit productions

The algorithm finds all pairs (A, B)

Basis, unit(A, A)

Induction, if we have found (B, B), and $B \to C$ is a unit production, add (A, C)

Cleaning up a grammar

If L is a CFL, then there is a grammar for $L - \{\varepsilon\}$ such that

- No useless symbols
- No ε-productions
- No unit productions

Chomsky normal form

Has only: $A \to BC$, $A \to a$

If L is a CFL, then $L - \{\varepsilon\}$ has a CFG in CNF

PUSHDOWN AUTOMATA

Equivalent in power of context-free grammars

Only the nondeterministic version defines all context-free languages

Deterministic PDAs are used to model parsers

Think of an ε-NFA with a stack to store symbols

Move are determined by

- Current state
- Current symbol (or ε)
- Current symbol on stack

A non det. PDA can have a choice of next moves

- Change of state
- Modify the stack by zero or more symbols with push or pops

A PDA is described by $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Γ: stack alphabet

Z_0: start stack symbol

Conventions

$a, b, c \dots$ are input symbols

$\dots X, Y, Z$ are stack symbols

$\dots w, x, y, z$ are strings of input

$\alpha, \beta, \dots$ are strings of stack symbols

Transition function

- Takes three arguments $\delta(q, a, Z)$

This is a set of zero or more actions, each consist of a new state p and a string α of stack symbol with which to replace the top symbol Z ...

Z: stack symbol

IF an action of a PDA is $\delta(q,a,Z)$ and contain (p,α)
... Then it can
- change state to p
- Remove a from the top of the stack
- Replace Z on the top of stack by α

Instantaneous description

An instantaneous description, or ID is a triple (q,w,α) where:
- q is the current state
- the remaining input w
- α is the stack contents (top of stack is the leftmost symbol of α)

There an analogy between derivations in a grammar and sequences of ID's for a PDA

We say ID I can become ID J in one move like that $I \vdash J$

Formally $(q,aw,X\alpha) \vdash (p,w,\beta\alpha)$ if $\delta(q,a,X)$ contains (p,β)

(top character a, w is the rest; X is top symbol, α is everything below)

We can extend $\vdash$ to $\vdash^*$, meaning zero or more moves

Language of a PDA

IF P is a PDA, then L(P) is the set of strings such that $(q_0,w,Z_0) \vdash^* (F,\varepsilon,\alpha)$

(final state; input consumed; anything left on the stack)

Another language is defined by the
- set of strings that make empty its stack

This language is called N(P) for a PDA P

Formally $(q_0,w,Z_0) \vdash^* (q,\varepsilon,\varepsilon)$ q can be non-Final

IF $L=L(P)$, there is a PDA P' such that $L=N(P')$

Deterministic PDA

There must be at least one move for any state q, input a, and stack symbol X

There must not be a choice between using ε or real input

Formally $\delta(q,a,X)$ and $\delta(q,\varepsilon,X)$ cannot both be nonempty

PUMPING LEMMA FOR CFG

We can always find two pieces of any long string and after repeating them the same number of times, we will get a string in the same language

Statement

For every context-free L there is an integer n, such that for every string z in L of length $\geq n$ there exists $z=uvwxy$ such that
- $|vwx| \leq n$
- $|vx| > 0$
- For all $i \geq 0$, uv^iwx^iy is in L

eg $\{0^i1^i \mid i \geq 1\}$ is a CFL

$\{0^i1^i0^i \mid i \geq 1\}$ is not
- We can't match two pairs or three counts as a group

Proof
- Suppose it is a CFL
- Let n be L's pumping lemma's constant (adversary picks)
- consider $z=0^n1^n0^n$
- we can write $z=uvwxy$ where $|vwx| \leq n$ and $|vx| \geq 1$ (adversary picks)

case 1, vx has no 0's
- Then at least one of them is a 1, which no string in L does

case 2, vx has at least one 0
- vwx is too short ($\leq n$) to extend all three blocks of 0's
- Thus uwy has at least one block of n 0's and at least one block with fewer than n 0's

This part of automata theory lets us show that certain tasks are impossible, or intractable

UNDECIDABILITY

Integers, strings and other, are all represented as strings of bits

KEY POINT Strings that are programs, are also strings of bits

Everything is an integer

But since binary strings can have leading 0's, there is not a unique representation

eg 101, 0101, 00101

To make the correspondence one to one, we put a one in front of any binary string, and then treat it as an integer

101 → 1101 0101 → 10101 00101 → 100101

A set is finite if it has a integer that is the count of the members on the set.

Formally, is one for which it is impossible to find a one to one correspondence between the members of the set and a subset of that set

... And an infinite set is one for which there one to one correspondence

A countable set has a one to one correspondence with positive integers (All countable sets are infinite)

Binary strings are countable. Every program can be represented with a unique integer.

We call the one to one correspondence between a countable set and a positive integer an enumeration of the set

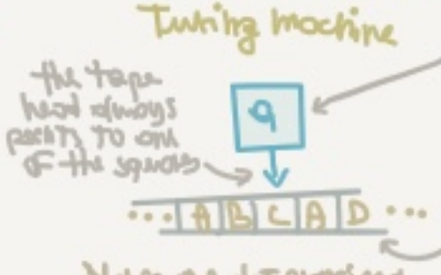
TURING-MACHINE THEORY

The languages over {0,1} are NOT countable

The purpose of this is to prove certain languages have no membership algorithm

- start with a language about Turing machines themselves
- reductions are used to prove more common questions undecidable

Turing machine



Moves are determined by the state and the symbol under the tape head

In one move it can change state, write a new tape symbol over the old one in the square it's scanning, and make the head one square Left or Right

They are as powerful as computer

... arguably even more because of the infinite tape

The transition function

$\delta(q,Z)$ for state q and tape symbol Z is either undefined, or a triple (p,Y,D) where p is a state, Y is a new tape symbol, and D is a direction L or R

Instantaneous description

Initially a TM has a tape consisting of a string of input symbol, surrounded by an infinity of blanks in both direction

The TM begin in it's start state, with the head at the left-most input symbol

An ID has the form $\alpha q \beta$ where $\alpha\beta$ includes the tape between the leftmost and rightmost nonblanks

The state q is immediately to the left of the symbol the head is now scanning

We use $\vdash$ to represent a move and $\vdash^*$ to represent any number of moves

Moves, formally

If $\delta(q,Z) = (p,Y,R)$ then $\alpha q Z \beta \vdash \alpha Y p \beta$
↳ if Z is blank, then $\alpha q \vdash \alpha Y p$

If $\delta(q,Z) = (p,Y,L)$ then for any X, $\alpha X q Z \beta \vdash$
- in addition $q Z \beta \vdash p B Y \beta$ (blank here; move the blank) $\quad \alpha p X Y \beta$

Languages of a TM

$L(M) = \{w \mid q_0 w \vdash^* I$ where I is an id with a final state$\}$

Or by halting, $H(M) = \{w \mid q_0 w \vdash^* I$ and there is no move from ID $I\}$

Formally, a TM is described by
- A finite set of states, Q
- An input alphabet, Σ.
- A tape alphabet, Γ, typically contains Σ
- A transition function, δ
- A start state, $q_0 \in Q$
- A blank symbol, B, in $\Gamma - \Sigma$ typically All the tape except for the input is blank initially
- A set of final states

a,b... are input symbols
...X,Y,Z are tape symbols
...w,x,y,z are strings of input
α,β... are strings of Tape symbols

notion of "anything we can compute" → RECURSIVELY ENUMERABLE and their subset, the RECURSIVE LANGUAGES

They both define the same class of languages

An algorithm is a TM accepting by final state, that is guaranteed to halt whether or not it accepts

IF $L(M)$ for a TM is an algorithm, then L is a recursive language

TRICK: MULTIPLE TRACKS

- Think of tape symbols as vectors with k components, each chosen from a finite alphabet
- Makes the tape appear to have k tracks
- Input symbols have the blank in all components but one

TRICK: MARKING

A good use of multitracks is to use one track for data, and another track for marks

In the mark track, almost all squares have blank value, but one or more have special symbols (marks) that indicates a place on the tape

TRICK: CACHING THE STATE

By thinking the state as a vector, with each component from some finite alphabet, we can use the first component to control operations, what we normally think as a state, and other components to hold data from a finite alphabet (typically bits or tape symbols)

eg this TM copies input w infinitely (Fairly simple)

control states
- q: mark the current position and remember the input symbol seen
- p: Run right, remembering the symbol, looking for a blank. Deposit symbol when found it
- r: Run Left looking for mark left by q, when found it remove the mark and enter state q

A state is a vector [x, y]. x is p, q or r and y is the cache with alphabet 0, 1, B

Tape symbols have the form [U, V]. The first component is the marking track, so it is either X (a mark) or B. The other component is the data track (input symbol) and it can be 0, 1 or B

transition

$\delta([q,B],[B,a]) = ([p,a],[X,a],R)$
- In state q, copy the input symbol under the head (a) into the cache
- Mark the position read with X
- Move the head right (R)

a and b represent either 0 or 1

$\delta([p,a],[B,b] = ([p,a],[B,b],R)$

In state p, if the current tape symbol does not have a blank, stay in that state and move right

$\delta([p,a],[B,B]) = ([r,B],[B,a],L)$
- when you find a B, place a (in the cache) in the data track
- go to state r and move left (L)

$\delta([r,B],[B,a]) = ([r,B],[B,a],L)$
- In r, move to the left looking for the mark

$\delta([r,B],[X,a]) = ([q,B],[B,a],R)$

When the mark is found we
- remove the mark
- go to state q
- move to the right

we're in state q, the whole cycle repeats

DECIDABILITY

To enumerate TMs, we encode them as binary strings

- our alphabet is $\{0,1\}$
- assign integer codes to the components of the TM
  - states: q_1 (start state), q_2 (final state) q_3...
  - symbols: X_1 (0), X_2 (1), X_3 (blank) X_4...
  - directions: D_1 (L), D_2 (R)

Suppose $\delta(q_i, X_j) = (q_k, X_l, D_m)$

- It is represented as $0^i 1 0^j 1 0^k 1 0^l 1 0^m$

all integers are >0, so there no consecutive 1s

1 separates; # of 0s, separated by 1s

Represent a TM by concatenating the encodings for each move, separated by 11 as punctuation

that is $code_1\ 11\ code_2\ 11\ code_3\ 11$...

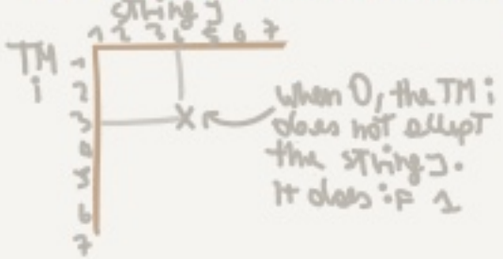
Once we have the encoding for our TMs, we can convert that binary string to a unique integer that way

- put a one in front of the binary string and treat the result as a binary integer

Thus it makes sense to talk about the i-th binary string and about the i-th TM

NOTE
Some binary strings represent flawed TMs (1s in the wrong place for example). We assume that those machines represent the empty language

Table of TMs related to the strings they accept



When 0, the TM i does not accept the string j. It does if 1

Whenever we have a table like this, we can diagonalize it

Formally $D = a_1 a_2 \ldots$ where $a_i = 0$ if the (i,i) table entry is 1 and vice-versa

D is not a row in the matrix, therefore it does not represent a language accepted by any turing machine

- Suppose it were the j-th row
- D disagrees with the j-th row at the j-th column
- Thus D is not a row

Consider the diagonalization language

$L_D = \{w \mid w \text{ is the i-th string and the i-th TM does not accept } w\}$

Since L_D has no TM, it is not recursively enumerable

Problems

Informally, a problem is a yes/no question about an infinite set of possible instances

eg Does graph G have a Hamilton cycle (cycle that touches each node exactly once)?

- Each undirected graph is an instance of the "Hamilton-cycle problem"

Here the answer is yes, there is a Hamilton cycle

- Formally, a problem is a language over some alphabet Σ
- Each string in Σ^* can be viewed as an instance of the problem once we decide on encoding
- The string is in the language if the answer to this instance of the problem is "yes".

NOTE
For flawed encodings, we assume the answer is "no"

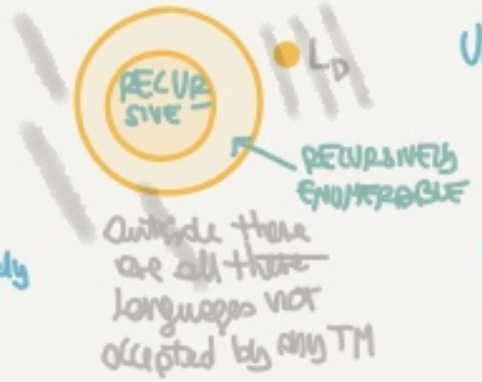
We can think of the language L_D as a problem
Does this TM not accept its own code?

Decidable problems

A problem is decidable if there is an algorithm to answer it

Then a decidable problem is a recursive language

Otherwise a problem is undecidable



Undecidable problem are either in the outer ring (RE languages but not recursive) or in the outer space.

Real world undecidable problems could be

- Can a particular line of code ever be executed?
- Is a given context-free grammar ambiguous?
- Do two CFG's generate the same language?

A recursively enumerable, but not recursive, language is the language L_U of a universal Turing machine

An UTM takes as input a binary string w and and a TM M. w is accepted if and only if M accepts w.

eg A java virtual machine takes a coded java program and an input for that program and then it executes that program on the input. A JVM is a more general UTM as it does not create a single accept output.

Designing the UTM

Inputs are of the form: code for M + 111 + w (a valid TM has never 111, so we can use it without ambiguities)

Has several tapes

- Tape 1 holds the input M111w
- Tape 2 is used to simulate the tape of M
- Tape 3 holds the state of M

...

Step 1 the UTM checks that M is a valid code for a TM

If M is not valid, its language is empty, so the UTM immediately halts without accepting

Step 2 The UTM examines M to see how many of its own tape squares it needs to represent one symbol of M

Step 3 Initialize tape 2 to represent the tape of M with input w, and initialize tape 3 to hold the start state

Step 4 Simulate M.

- Look for a move on tape 1 that matches the state on tape 3 and the tape symbol under the head on tape 2
- If found, change the tape symbol and move the head marker on tape 2 and change the state on Tape 3
- If M accepts, the UTM also accepts

INTRACTABLE PROBLEMS

Decidable problems, but they take at least exponential time as a function on their input size

In practice, those problems can only be solved for small instances

An usable problem in practice, has to run in less than exponential time, polynomial time, for some low degree polynomial

Time-Bounded TM's

A TM that, given an input of size n, always halts within T(n) moves is said to be T(n)-time bounded

- TM can be multitape
- Sometimes, it can be non-deterministic

The class P

- If a DTM M is T(n)-Time bounded for some polynomial T(n), then we say that M is polynomial time
- L(M) is said to be in the class P
- When we talk of P, doesn't matter whether we mean "by a computer" or "by a TM" (If a computer takes O(T(n)) for a given algorithm, a TM can simulate it in at most $O(T^2(n))$ steps)

Running times between polynomials

P only requires that the language be accepted by some TM whose running time is bounded above by some polynomial
(O(logn) or O(nlogn) are included)

KNAPSACK, seems to be in P but isn't

- Given positive integers $i_1 \dots i_n$, can we divide them in two sets with equals sums
- At first glance, we can solve it by using dynamic programming. Maintains table of all the differences by partitioning the first j

Although that runs in no more than the square of the sums of the integers, it doesn't tell us that knapsack is in P.
The problem is that we cannot define the input size as the sum of the integers in the input. The input size is always the # of cells that takes to write the input on a TM tape.

- Suppose we have n integers, each of which is 2^n.
- We can write integers in binary, so the input takes $O(n^2)$
- But the tables requires space $O(n2^n)$
- All n tables in time $O(n^2 2^n)$. But using n as the input size, it requires $O(n^2 \text{sqrt}(n))$

The class NP

The running time of a nondeterminist TM is the maximum number of steps taken along any branch (makes any sequence of choices)

If that time bound is polynomial, the NTM is said to be polynomial-time bounded

And it's language is said to be in the class NP

Knapsack is NP:

First uses its non-determinism to guess a partition of the integers into subsets

...then sum the subsets and compare, say yes if two partitions yield two equal sums.

P versus NP

Are P and NP the same class of languages? That is, can any problem that is solved by a NTM in polynomial time also be solved by some det. TM in polynomial time even if the degree is higher?

↳ Question posed by Steve Cook in 1970
After all, NFA can be simulated by a DFA, and NTM's can simulate det. ones. But the problem is still very very difficult.

- There are thousands of NP problems for which no algorithm in P has been found. Neither there is a proof that these problems are not in P