Programming Turing Machine example

Everything is an integer or string

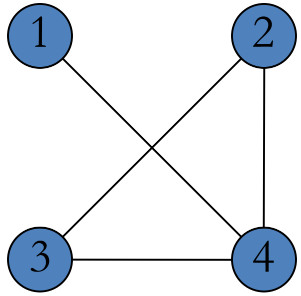
Enumerators

Programming Turing Machines

$L_4 = \{\langle G \rangle : G \text{ is a connected undirected graph}\}$

Q: How do we feed a graph into a Turing Machine?

A: We represent it by a string, e.g.



`(1,2,3,4)((1,4),(2,3),(3,4)(4,2))`

Convention for describing graphs:

(nodes) (edges)

no node must repeat

edges are pairs $(\text{node}_1, \text{node}_2)$

Programming Turing Machines

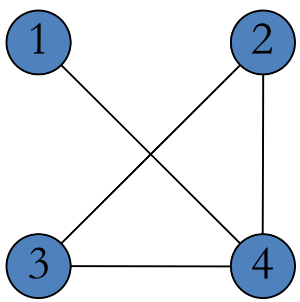
To check if $\langle G \rangle$ is in L_4:

0. Verify that $\langle G \rangle$ is the description of a graph
   (no vertex repeats; edges only go between nodes)
1. Mark the first node of G *(with dot on top)*
2. Repeat until no new nodes are marked:
   For each node, mark it if it is attached to an already marked node:
   a) Scan to find an undotted node v and mark it *(by underlining)*
   b) Scan again to find a dotted node u and underline it, too
   c) Scan edges to see if two underlined nodes are in one edge
      if they are, remove underlines, dot v, go to a)
   d) If there are no more edges, then move underline from u to next dotted node, go to c).

3. If all nodes are marked accept, otherwise reject.

Programming Turing Machines

$L_4 = \{\langle G \rangle : G \text{ is a connected undirected graph}\}$



$(\underline{\dot{1}},\underline{2},3,4)((1,4)(2,3)(3,4)(4,2))$

$(\underline{\dot{1}},2,\underline{3},4)((1,4)(2,3)(3,4)(4,2))$

$(\underline{\dot{1}},2,3,\underline{4})((1,4)(2,3)(3,4)(4,2))$

$(\dot{1},2,3,\dot{4})((1,4)(2,3)(3,4)(4,2))$

$(\underline{\dot{1}},\underline{2},3,\dot{4})((1,4)(2,3)(3,4)(4,2))$

$(\dot{1},\underline{2},3,\underline{\dot{4}})((1,4)(2,3)(3,4)(4,2))$

$(\dot{1},\dot{2},3,\dot{4})((1,4)(2,3)(3,4)(4,2))$

$(\underline{\dot{1}},\dot{2},\underline{3},\dot{4})((1,4)(2,3)(3,4)(4,2))$

$(\dot{1},\underline{\dot{2}},\underline{3},\dot{4})((1,4)(2,3)(3,4)(4,2))$

$(\dot{1},\dot{2},\dot{3},\dot{4})((1,4)(2,3)(3,4)(4,2))$

Why the following is not a description of a legitimate Turing machine?

Mbad = "On input $\langle p \rangle$, a polynomial over variables $x1, \ldots, xk$:

1. Try all possible settings of $x1, \ldots, xk$ to integer values.

2. Evaluate p on all of these settings.

3. If any of these settings evaluates to 0, accept; otherwise, reject ."

Why the following is not a description of a legitimate Turing machine?

Mbad = "On input $\langle p \rangle$, a polynomial over variables $x1, \ldots, xk$:

1. Try all possible settings of $x1, \ldots, xk$ to integer values.

2. Evaluate p on all of these settings.

3. If any of these settings evaluates to 0, accept; otherwise, reject ."

Answer:

The variables $x1, \ldots, xk$ have infinitely many possible settings.
A Turing Machine will need infinite time to try them all.
However, every stage of a TM description must be completed in a finite number of steps.

Everything can be represented as Integers or Strings

- Data types have become very important as a programming tool.
- But at another level, there is only one type, which you may think of as integers or strings.

Example: Text

- Strings of ASCII or Unicode characters can be thought of as binary strings, with 8 or 16 bits/character.
- Binary strings can be thought of as integers.
- It makes sense to talk about “the i-th string.”

Binary Strings to Integers

- There's a small glitch:
  - If you think simply of binary integers, then strings like 101, 0101, 00101,… all appear to be "the fifth string" (2^2+1=5).
- Fix by prepending a "1" to the string before converting to an integer.
  - Thus, 101, 0101, and 00101 are the 13th(1101), 21st(10101), and 37th(100101) strings, respectively.

Example: Images

- Represent an image in (say) GIF.
- The GIF file is an ASCII string.
- Convert string to binary.
- Convert binary string to integer.
- Now we have a notion of “the i-th image.”

Example: Proofs

- A formal proof is a sequence of logical expressions, each of which follows from the ones before it.
- Encode mathematical expressions of any kind in Unicode.
- Convert expression to a binary string and then an integer.

Proofs – (2)

- But a proof is a sequence of expressions, so we need a way to separate them.
- Also, we need to indicate which expressions are given.

Proofs – (3)

- Quick-and-dirty way to introduce new symbols into binary strings:
  1. Given a binary string, precede each bit by 0.
     - Example: 101 becomes 010001.
  2. Use strings of two or more 1's as the special symbols.
     - Example: 111 = "the following expression is given"; 11 = "end of expression."

Example: Encoding Proofs

111	010001	11	111	00000101	11	0101…

A given expression follows

An ex-pression

Notice this 1 could not be part of the "end"

End of expression

A given expression follows

Expression

End

Example: Programs

- Programs are just another kind of data.
- Represent a program in ASCII.
- Convert to a binary string, then to an integer.
- Thus, it makes sense to talk about “the i-th program.”
- Hmm…There aren’t all that many programs.

Finite Sets

- Intuitively, a *finite set* is a set for which there is a particular integer that is the count of the number of members.
- Example: {a, b, c} is a finite set; its *cardinality* is 3.
- It is impossible to find a 1-1 mapping between a finite set and a proper subset of itself.

Infinite Sets

- Formally, an *infinite set* is a set for which there is a 1-1 correspondence between itself and a proper subset of itself.
- Example: the positive integers {1, 2, 3,…} is an infinite set.
  - There is a 1-1 correspondence 1<->2, 2<->4, 3<->6,… between this set and a proper subset (the set of even integers).

Countable Sets

- A *countable set* is a set with a 1-1 correspondence with the positive integers (N).
  - Hence, all countable sets are infinite.
- Example: All integers.
  - 0<->1; -i <-> 2i; +i <-> 2i+1.
  - Thus, order is 0, -1, 1, -2, 2, -3, 3,…
- Examples: set of binary strings, set of Java programs.

Example: Show that the set of finite strings *S* over the lowercase letters is countable.

Proof:

Show that the strings can be listed in a sequence.

1. First list all the strings of length 0 in alphabetical order.
2. Then all the strings of length 1 in lexicographic (as in a dictionary) order.
3. Then all the strings of length 2 in lexicographic order.
4. And so on.

This implies a bijection from **N** to *S* and hence it is countable.

Example: The set of all Java programs is countable.

Proof:

Let S be the set of strings constructed from the characters which can appear in a Java program. Use the ordering from the previous example. Take each string in turn:

- Feed the string into a Java compiler. (A Java compiler will determine if the input program is a syntactically correct Java program.)
- If the compiler says YES, this is a syntactically correct Java program, we add the program to the list.
- Move on to the next string.

In this way we construct an implied bijection from N to the set of Java programs. Hence, the set of Java programs is countable.

Example: Pairs of Integers

- Order the pairs of positive integers first by sum, then by first component:
- [1,1], [2,1], [1,2], [3,1], [2,2], [1,3], [4,1], [3,2],…, [1,4], [5,1],…
- Interesting fact: this same proof means that **rational numbers** are countable.

Enumerations

- An ***enumeration*** of a set is a 1-1 correspondence between the set and the positive integers.
- Thus, we have seen enumerations for strings, programs, proofs, and pairs of integers.

How Many Languages?

- Are the languages over {0,1}* countable?
- No; here's a proof.
- Suppose we could enumerate all languages over {0,1}* and talk about "the i-th language."
- Consider the language L = { w | w is the i-th binary string and w is not in the i-th language}.

Proof – Continued

- Clearly, L is a language over {0,1}*.
- Thus, it is the j-th language for some particular j.
- Let x be the j-th string.
- Is x in L?
  - If so, x is not in L by definition of L.
  - If not, then x is in L by definition of L.

L_j

x

Recall: L = { w | w is the i-th binary string and w is not in the i-th language}.

j-th

Diagonalization Picture

Strings

Languages

	1	2	3	4	5	…
1	1	0	1	1	0	…
2		1				
3			0			
4				0		
5					1	
…						…

Diagonalization Picture

Flip each diagonal entry

Languages

Strings

	1	2	3	4	5	…
1	0	0	1	1	0	…
2		0				
3			1			
4				1		
5					0	
…						…

Can’t be a row – it disagrees in an entry of each row.

Proof – Concluded

- We have a contradiction: x is neither in L nor not in L, so our sole assumption (that there was an enumeration of the languages) is wrong.
- Thus, there are (way) more languages than programs.
- E.g., there are languages with no membership algorithm.

Non-constructive Arguments

- We have shown the **existence** of a language with no algorithm to test for membership, but we have not exhibited a particular language with that property.
- A proof that shows only that something exists without showing a way to find it or giving a specific example is a **non-constructive argument**.

Turing-Machine Theory

- The purpose of the theory of Turing machines was to prove that certain specific languages have no algorithm.

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

Why Turing Machines?

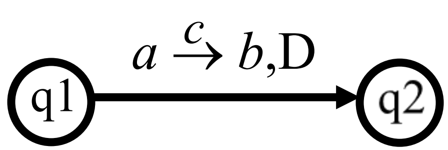
- Why not deal with C programs or something like that?
- Answer: You can, but it is easier to prove things about TM's, because they are so simple.
  - And yet they are as powerful as any computer.
    - More so, in fact, since they have infinite memory.

Variants of TM: Enumerators

An *enumerator* is a TM with two tapes: the *work tape* and the *printer*.

- Initially both tapes are blank.
- On the work tape it can read, write and move in either direction just as an ordinary TM.
- On the printer, it can only print strings.

- So that transitions look like



(if, in state q1, you see an a on the work tape, replace it with b, move in the direction D (D=L or D=R), go to state q2, print c on the printer and move right)

Enumerability vs Turing recognizability

Theorem 3.21: A language is Turing recognizable iff some enumerator enumerates it.

Proof sketch. Consider an arbitrary language L.

($\Leftarrow$): Suppose E enumerates L. Construct a TM M that works as follows:

M = "On input w:

1. Simulate E. Every time E prints a new string, compare it with w.
2. If w is ever printed, accept."

($\Rightarrow$): Suppose M recognizes L. Let $s_1, s_2, s_3, \ldots$ be the lexicographic list of all strings over the alphabet of L. Construct an enumerator E that works as follows:

E = "

1. Repeat the following for $i=1,2,3,\ldots$
2. Simulate M for i steps on each of the inputs $s_1, s_2, \ldots, s_i$.
3. If any computations accept, print out the corresponding s_j."