Effective Computability and Decidability

Robert M. Keller
Harvey Mudd College
April 2013

“Effective” is an informal term

- Church and Turing separately tried to formalize effective computability in 1936 and 1937, respectively.

Effective?

- Consider "effective" to mean "computable by *some* [discrete] method".
- DFAs and PDAs offer a ***limited*** form of computability. They are effective, but not every effectively-computable function is computable by a PDA.
- Computing membership in the language $\{a^n b^n c^n \mid n \geq 0\}$ is a case in point. There is a way to compute membership in this language, but not by a PDA.

History of Effective Computability

Gottfried Wilhelm Leibniz, 1646-1764

The origin of the Entscheidungsproblem goes back to Gottfried Leibniz, who in the seventeenth century, after having constructed a successful mechanical calculating machine, dreamt of building a machine that could manipulate symbols in order to determine the truth values of mathematical statements. Leibniz may have been the **first computer scientist** and information theorist.[65] Early in life, he documented the binary numeral system (base 2), then revisited that system throughout his career.[66] He anticipated Lagrangian interpolation and algorithmic information theory. His calculus ratiocinator **anticipated aspects of the universal Turing machine**.

In 1671, Leibniz began to invent a machine that could execute all four arithmetical operations, gradually improving it over a number of years. This "Stepped Reckoner" attracted fair attention and was the basis of his election to the Royal Society in 1673.
Leibniz was groping towards hardware and software concepts worked out much later by **Charles Babbage and Ada Lovelace**. In 1679, while mulling over his binary arithmetic, Leibniz imagined a machine in which binary numbers were represented by marbles, governed by a rudimentary sort of punched cards.[69] Modern electronic digital computers replace Leibniz's marbles moving by gravity with shift registers, voltage gradients, and pulses of electrons, but otherwise they run roughly as Leibniz envisioned in 1679.

http://en.wikipedia.org/wiki/Gottfried_Leibniz

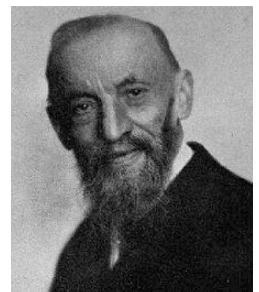
Dedekind and Peano

- **Richard Dedekind** (1831-1916) in 1888 proposed an axiomatic foundation for the natural numbers, whose primitive notions were **1** and the **successor** function.

- **Giuseppe Peano** (1858–1932) in 1889 created a simpler set of axioms (also using 1), acknowledging Dedekind.

  [Using 0 vs. 1 is a non-issue, since the axioms give an *abstract* characterization of the natural numbers.
  0 is mostly used today.]

Dedekind



Peano

Wilhelm Ackermann

- **Ackermann** (1896-1962), a student of David Hilbert, investigated properties of recursive functions over the natural numbers.
- He defined the family of functions now known as **primitive recursive**, and found an obviously-computable function that is *not* in the family (Ackermann's function).
- He proved the consistency of a set of axioms for arithmetic, and co-authored one of the first **mathematical logic books** with Hilbert.

Ackermann's Function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Sometimes used as a test-case for bignum computation.

A(4,2) = 2^{65536} - 3, an integer having 19,729 decimal digits

Kurt Gödel

- Gödel (1906-1978) proposed in 1931 a formalism for **computable functions** that extended Ackermann's primitive recursive functions.
- He called these **general recursive functions**. The formalism is similar to functional languages.

Gödel (right) with Einstein

- These played a role in his work on representing logical formulas arithmetically, and his **incompleteness theorem**.

Alonzo Church

- Church (1903–1995) was the first to prove, in 1936, that the **predicate calculus is undecidable**.
- This result is known as **Church's Theorem**.
- The problem itself then went by the name **The Entscheidungsproblem** ("Decision problem").
- Apparently his proof used some of the work of his student, Stephen Kleene.

The Lambda Calculus

- Church invented the lambda calculus in 1936. It is a system for defining **functions** and computing with them.

- Its use appears in many **functional languages**, including Lisp/Scheme and more recently Python.

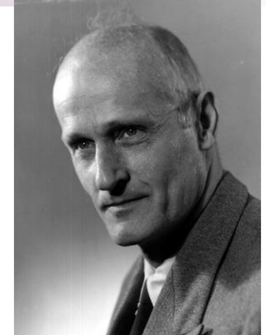
- It is the residue of a **logical system** invented by Church that was eventually shown **inconsistent**, because certain expressions were equivalent to their own negation. As computation, however, this simply expresses a form of non-terminating behavior.

Let $k = (\lambda x.\ \neg(x\ x))$, then $kk = (\lambda x.\ \neg(x\ x))k = \neg(kk)$.
The function $\neg$ is somewhat arbitrary here, since kk is further applicable.

Universality of the Lambda Calculus

- Church conjectured that the lambda calculus was equivalent to effective computability.
- However, this claim was not totally convincing, and some (e.g. Gödel) expressed skepticism.
- Eventually the lambda calculus was shown to be equivalent to Turing machines and other formalisms.

Stephen Kleene



- Kleene (1909-1994, the same Kleene who discovered **regular expressions** in 1951) was a student of Church, as was Turing.
- He wrote in 1952 the text *Introduction to Metamathematics*, which expounded on Gödel's work and also introduced the **partial recursive functions**, a more structured formalism for computability.
- [This gave rise to the saying:
  "Kleeneness is next to Gödelness."]

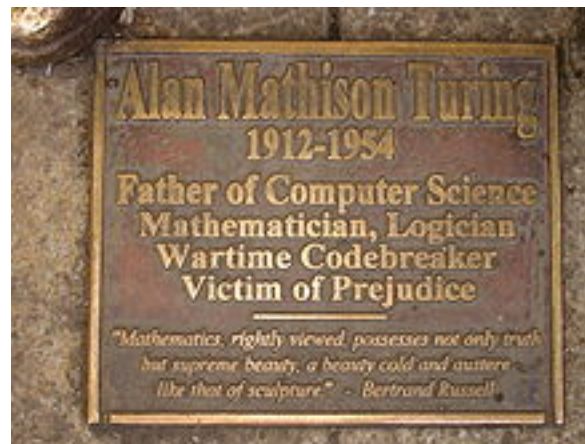
Alan Turing

- Turing (1912-1954) was Church's student at Princeton.

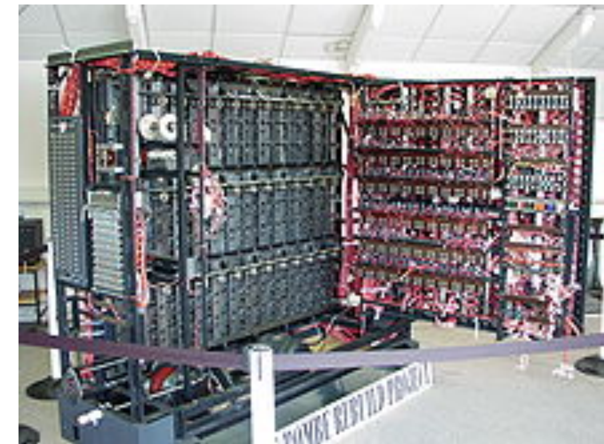
- In 1937 (a year after Church published the lambda calculus), Turing published his paper "On Computable Numbers, with an Application to the Entscheidungsproblem" (Proceedings of the London Mathematical Society 42) which defined and defended a variant of what we now call the **Turing machine**, although Turing appeared to be trying to characterize computability by a *human* "computer".

- Turing's defense was more intuitive and grounded than Church's defense of the lambda calculus, in my opinion.

Alan Turing, 1912-1954



Turing memorial plaque,
Manchester, England

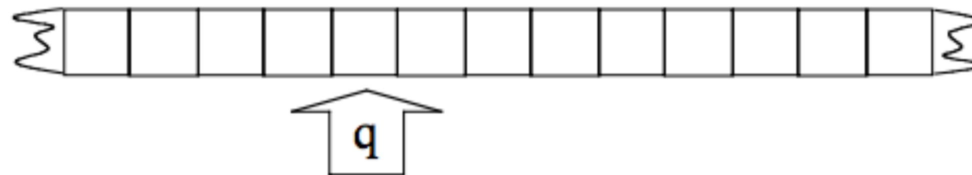


Turing's decryption computer
da Bombe, 1941
which helped win WWII

http://en.wikipedia.org/wiki/Alan_Turing

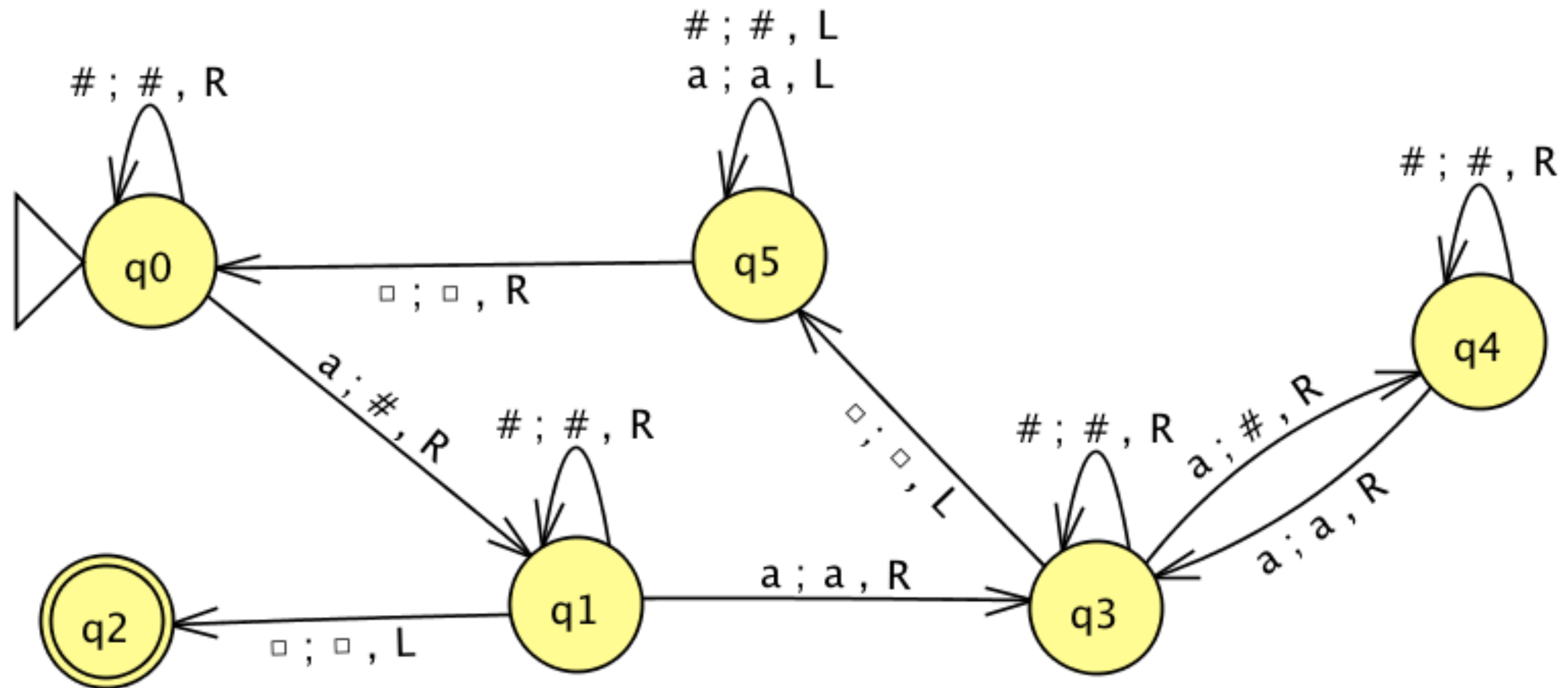
Turing Machines

- I will describe the model on the board.

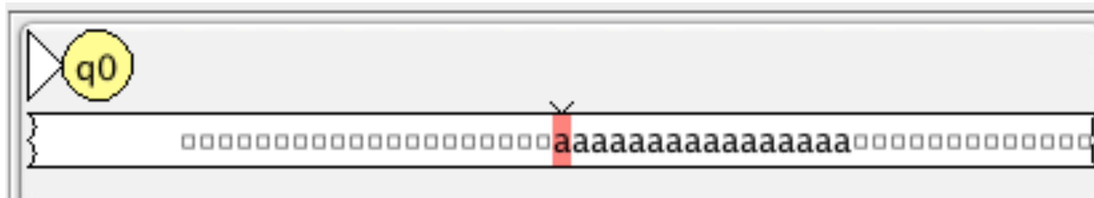


- I will give a Prolog program that simulates a Turing machine.
- See JFlap for additional examples.

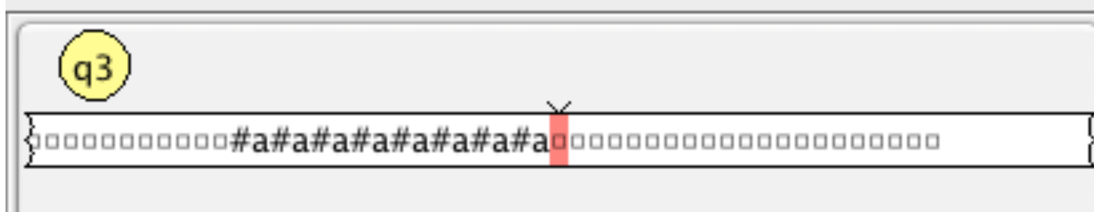
Example L = {a^n | n > 0 is a power of 2}
L(M) = L for the following machine M



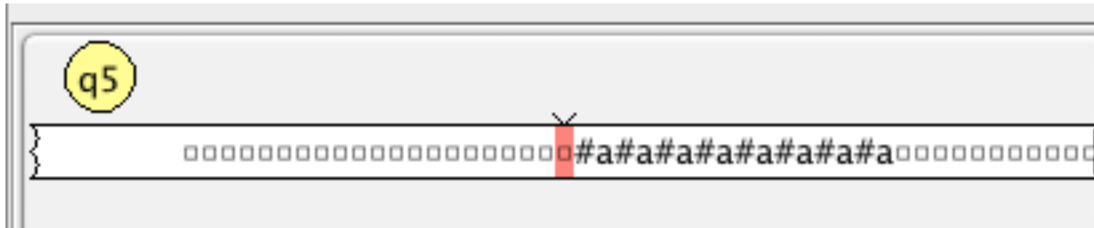
Example Configurations for L



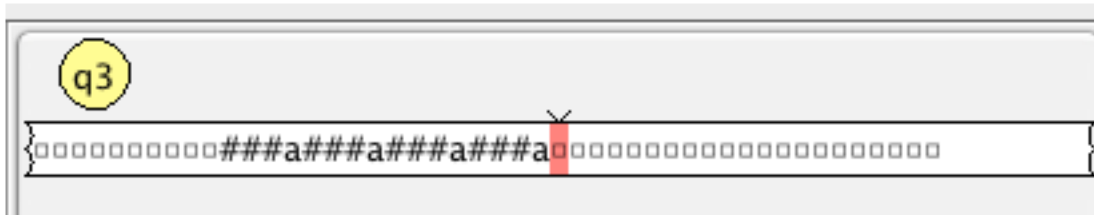
Starting on 16 a's



q3 and q4 mark off every other a replacing it with #, effectively dividing by 2

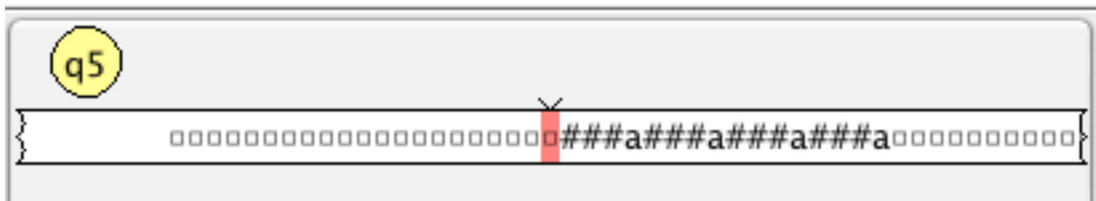


q5 returns to left end.

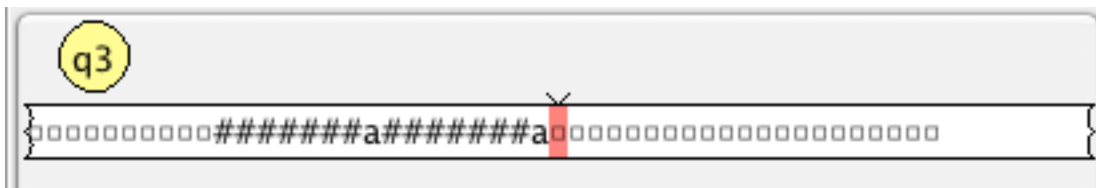


q3 and q4 again mark off every other a replacing it with #, effectively dividing by 2

Example Configurations for L

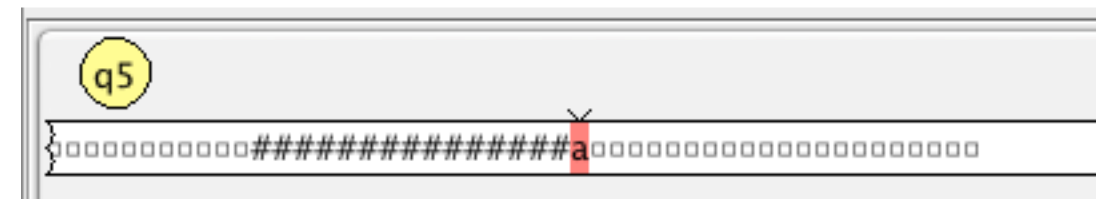


q5 returns to left end.



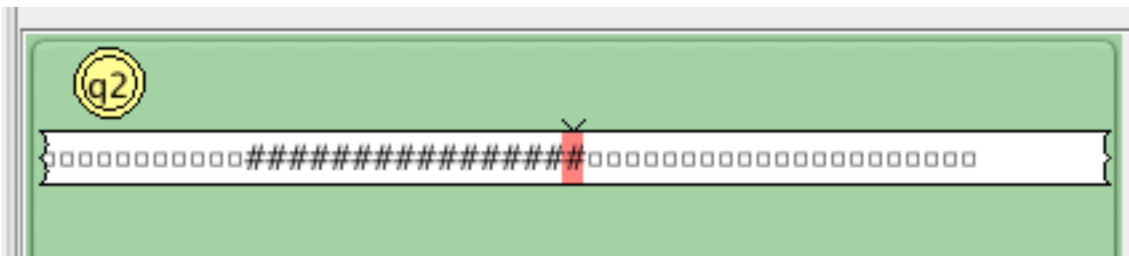
q3 and q4 again mark off every other a replacing it with #, effectively dividing by 2

q5 returns to left end.

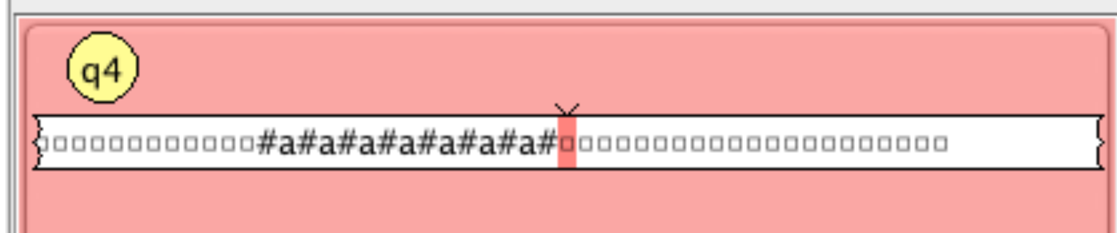


q3 and q4 again mark off every other a replacing it with #, effectively dividing by 2

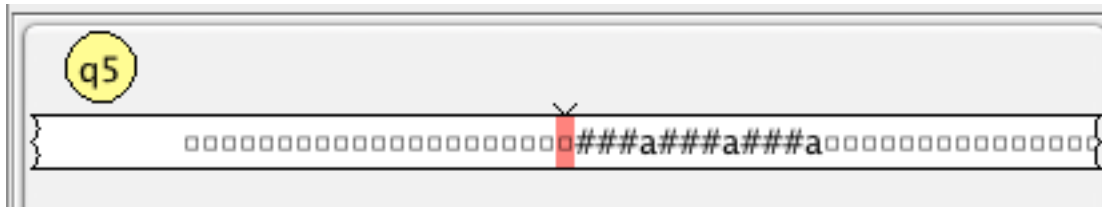
Example Configurations for L

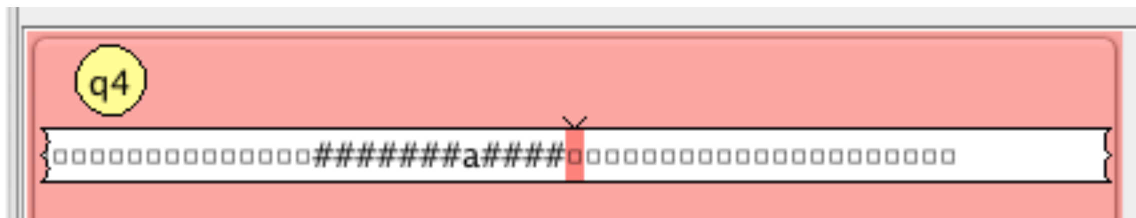


Make a pass to see if any a’s left over. If not, **accept**.



With 15 a’s,
an odd number of a’s is detected on first pass in q4, and input rejected.





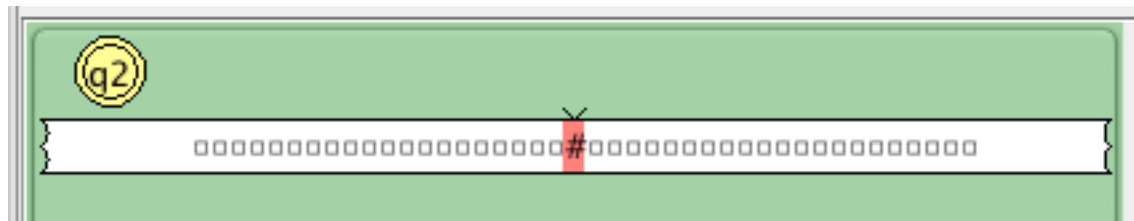
With 12 a’s,
an odd number of a’s won’t be detected until the third pass.

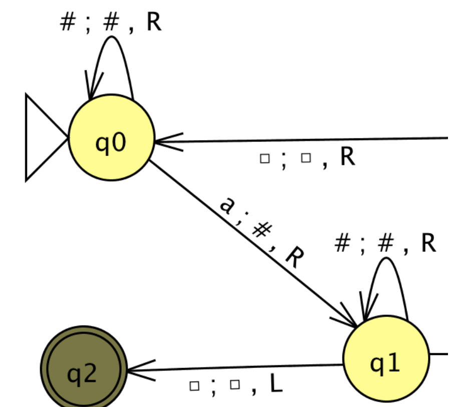
Multiple Run

Multiple Run

Input	Result
	Reject
a	Accept
aa	Accept
aaa	Reject
aaaa	Accept
aaaaa	Reject
aaaaaaaa	Accept
aaaaaaaaa	Reject
aaaaaaaaaaaaaaaa	Accept

Single **a** case is detected by q0 to q1 to q2.





Things to Remember about Turing Machines

- There is a **finite** set of **tape symbols**.
- There is a **finite** set of **control states** (so the control states cannot remember arbitrarily-large numbers).
- Single actions are **local** and bounded in complexity (read, write, move, change-state).
- At any one time, only a **finite portion** of the tape is in use. The size of the portion can increase over time however, so the **tape is unbounded**.
- There is no result until the machine **halts**.

The Church-Turing Thesis

- Effectively-computable functions are the same as Turing-computable ones.

Turing's Argument

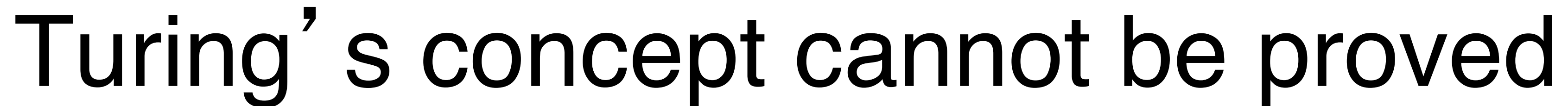
- Everyone should read the prose parts (possibly skipping some of the notation) from the original source. (It is only a few pages long.)

- Here are on-line versions:
  http://classes.soe.ucsc.edu/cmps210/Winter11/Papers/turing-1936.pdf

  http://www.abelard.org/turpap2/tp2-ie.asp (if you love awkward coloring)

Why accept Turing's concept?

- Turing gave a careful informal argument as to why his machine model captured the essence of computability.
- Other models of computabilty (a dozen or so) are provably equivalent to Turing's machines.
- No one has come up with a function that is intuitively computable that could be proved to be not computable on a Turing machine.



Turing’s concept cannot be proved

- In order to **prove** that Turing was correct, we’d have to come with **another model**, say an **X machine**, for computability that is more obviously correct, then prove that anything computable by an X machine could be computed by a Turing machine.

- But then there would remain the question of whether the **X machine** completely captured the notion of computability.

- This could lead to **infinite regress** (X machine, Y machine, Z machine, …), still with no final proof.

Turing's idea could conceivably be disproved.

- This could be done by devising a function that is **intuitively computable**, then proving that it cannot be computed by a Turing machine.
- This has not been done to satisfaction of the general community.
- However, there is work in this direction:

  http://en.wikipedia.org/wiki/Super-recursive_algorithm

Turing Computability Hedge

- To put arguments about computability on a precise formal basis, one can add the qualification "Turing" in front of "computabilty".

- When we say "computability" here, Turing computability is what we mean.

Gödel on Turing on Computability (from Oron Shagrir, 2004)

Quoting Gödel: "The greatest improvement was made possible through the **precise definition of the concept of finite procedure**, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to exactly the same concept. **The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.**"
[Gödel 1951, pp. 304–305]

Gödel on Turing on Computability (from Oron Shagrir, 2004)

The preceding is the approach Gödel recommends in his 1934 conversation with Church, in which he **rejects Church's proposal as "thoroughly unsatisfactory."**

Gödel suggests to Church that "it might be possible, in terms of an **effective calculability** as an undefined notion, to state a set of axioms which would embody the generally accepted properties of this notion, and to do something on that basis."

Emil Post

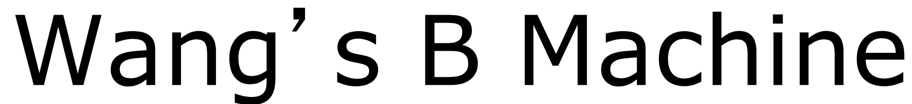
- Post (1897-1954) may have been the first to prove completeness of the propositional calculus.
- In 1936, *independently of Turing*, he developed his own type of machine similar to Turing's and conjectured that it was equivalent in power to Gödel's recursive functions.
- For this and related models, see: http://en.wikipedia.org/wiki/Post-Turing_machine

Hao Wang

- Hao Wang (1921-1955), a student of Quine, introduced in 1954 the **B-machine** model which could **not erase symbols**, only write them, yet was equivalent in power to the Turing machine.

- [How is this possible?]

- He was the first to use a program-like **statement sequence**.



As defined by Wang (1954) the B-machine has at its command only 4 instructions:

(1) → : Move tape-scanning head one tape square to the right (or move tape one square left), then continue to next instruction in numerical sequence;

(2) ← : Move tape-scanning head one tape square to the left (or move tape one square right), then continue to next instruction in numerical sequence;

(3) * : In scanned tape-square print mark * then go to next instruction in numerical sequence;

(4) Cn: Conditional "transfer" (jump, branch) to instruction "n": If scanned tape-square is marked then go to instruction "n" else (if scanned square is blank) continue to next instruction in numerical sequence.

Joachim Lambek's "Infinite Abacus" (1961)

- This machine used **registers** holding **natural numbers** instead of a tape holding symbols.
- This model is used extensively in logic texts by George Boolos and Richard Jeffrey.
- A host of similar models appeared around the same time, or in the decade after.
  - Counter machines (Shepherdson and Sturgis)
  - Register machines
  - RAM (Random-Access Machine)
  - RASP (Random-Access, Stored-Program) machine
- http://en.wikipedia.org/wiki/Counter_machine
  http://en.wikipedia.org/wiki/Register_machine

Extending Turing Machines

- What features could be added to Turing machines to try to make them more powerful?
  - Multiple tracks …

Extending Turing Machines

- Do those features make the model more powerful (as far as computability is concerned)?
  - Multiple tracks …

Universal Turing Machines

- Each Turing machine (as determined by its set of 5-tuples or rules) computes a certain function (accepts a certain language).
- A **Universal** TM can compute anything any other TM can.
- What's the catch? The UTM has to have the original TM's program written on its tape and be able to interpret it. The symbols of the original machine are encoded into symbols of the **fixed alphabet** of the UTM.

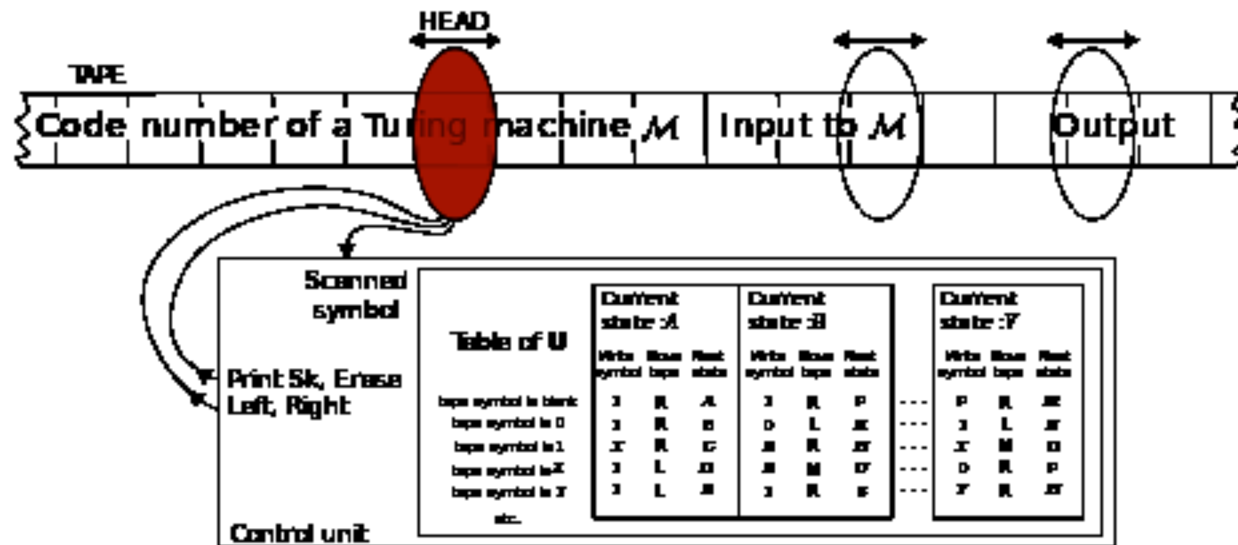
Universal Turing machines

- A universal Turing machine is one can simulate any other Turing machine.
- The alphabet of a universal TM is fixed.

  So we'd need to **encode** the alphabet of the TM being simulated.
- The program of the simulated TM can be arbitrarily large.

  So we'd need to **encode and store the program** of the simulated TM on the universal machine's tape.
- The universal machine would "jockey back and forth" between the program and the simulated tape, leaving markers, etc. in the form of tape symbols to keep track of current state and head position.

Universal Turing machine



http://en.wikipedia.org/wiki/Universal_Turing_machine

Examples of UTMs

- The Hopcroft & Ullman machine, presented in the first edition of their textbook (1969), has 40 states and 12 tape symbols, used to simulate machines with 2 tape symbols only. (This was not the first UTM.)
- Machines with more than 2 tape symbols can be transformed into ones with only 2 by encoding the symbols into binary.
- The state table of this machine is 3 pages of a text book.
- See: http://www.rdrop.com/~half/General/UTM/UTMStateTable.html

A 3-tape Universal Machine in JFLAP

q0 Match State
q1 Rewind T3
q2 Match Symbol
q3 Rewind T2
q4 Write State
q5 Rewind T3
q6 Skip 5 T1 0s
q7 Skip 4 T1 0s
q8 Skip 3 T1 0s
q9 Skip 2 T1 0s
q10 Skip 1 T1 0s
q11 Rewind T3
q12 Rewind T2
q13 Write Symbol
q14 Next 0
q15 Next 1
q16 Rewind T2
q17 Rewind T2
q18 Fast Forward T2
q19 Prev 0
q20 Prev 1
q21 Start Compression
q22 Fill Left Blank
q23 Rewind T2
q24 Moved L
q25 Rewind T1
q26 Skip Right
q27 Moved S
q28 Skip Right
q29 Moved R
q30 Fill Right Blank
q31 q1 or higher
q32 q2 or higher
q33 q2, ACCEPT!

Other UTMs

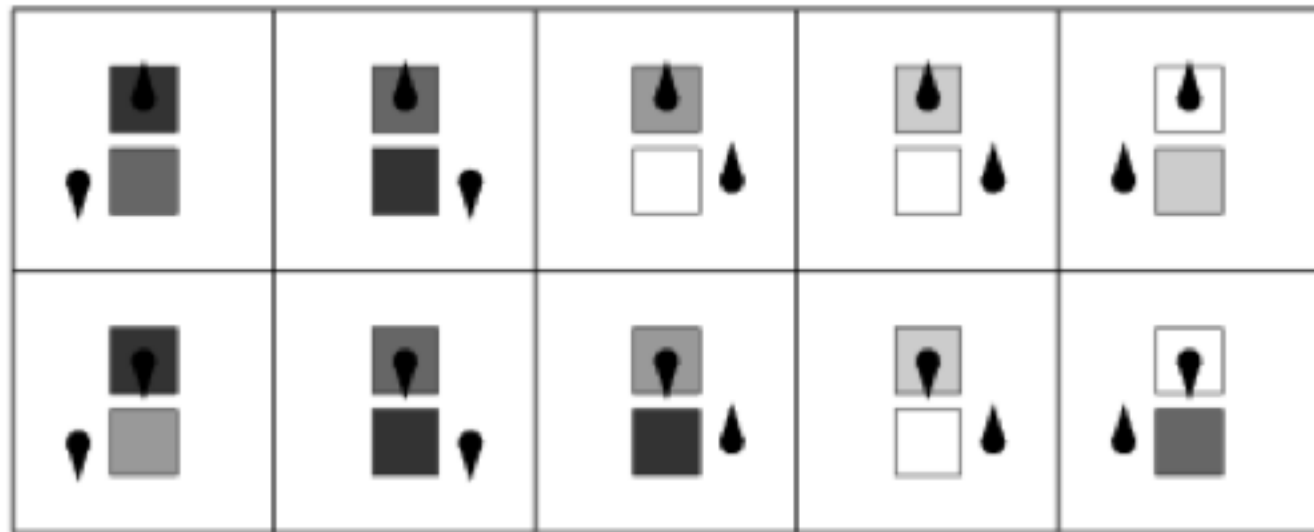
- Marvin Minsky, 1962: 4 state x 7 symbol
- (The state x symbol product is sometimes used as a measure to be reduced.)

symbols

states	1	2	3	4	5	6	7
Y	_L1	_L1	YL3	YL4	YR5	YR6	_R7
_	_L1	YR2	HALT	YR5	YL3	AL3	YR6
1	1L2	AR2	AL3	1L7	AR5	AR6	1R7
A	1L1	YR6	1L4	1L4	1R5	1R6	_R2

Other UTMs

- Stephen Wolfram, 2002: 2 state, 5 symbol machine (proof controversial)



adapted from Wolfram, S. *A New Kind of Science*. Wolfram Media, p. 707, 2002.

TMs presented informally, Algorithms

- We typically use the Church-Turing thesis to avoid having to present a Turing machine in detail.
- If we can give a description of an **algorithm**, the thesis says that there is a TM that carries out the same computation.

Programming Languages

- Functions that are computable in most ordinary programming languages (Racket, Java, C++, C, Python, …) are computable by Turing machines, and vice-versa.

- We assume that the platform on which programs in these languages executes has unbounded memory available (just like a Turing machine tape does).

Mutual Simulation

- It is relatively easy to show that all common programming languages can simulate an arbitrary Turing machine.
- It is also possible, but generally tedious, to show that a Turing machine can simulate programs in common languages.

Why do we care?

- In showing that there are functions that are not computable, it may be easier to use a programming language.

Decision Problems
TM Languages

More Examples: Argue that there is an algorithm or TM for each.

- The input is an encoding of a DFA M in a straightforward way.
- Argue, if possible, that following problems are effectively computable:
  - For a fixed $x \in \Sigma^*$, is $x \in L(M)$?
  - For a letter $\sigma \in \Sigma$, is there an $x \in L(M)$ containing σ?
  - For a fixed $x \in \Sigma^*$, is x the prefix of some string in L(M)?
  - Is $L(M) = \Sigma^*$?
  - Is $L(M) = \varnothing$?

More Decision Problems

- For two encoded DFAs, M and N
  - Is $L(M) \cap L(N) = \varnothing$?
  - Is $L(M) = L(N)$?
  - Is $L(M) \subseteq L(N)$?

More Problems: Is there an algorithm?

- For G a context-free grammar:
  - Is G in Chomsky normal form?
  - For a fixed $x \in \Sigma^*$, is $x \in L(G)$?
  - Is $L(G) = \emptyset$?
  - Is $L(G) = \Sigma^*$?
  - Is L(G) regular?
- For G and H two context-free grammars:
  - Is $L(G) = L(H)$?
  - Is $L(G) \cap L(H) = \emptyset$?

Decidability

- The previous problems are all examples of decision problems. They are asking for a **yes/no answer**.

- All can be characterized as **language acceptance problems**.

- Example: Is L(M) = L(N)? (for DFAs): This is asking whether **a pair of encoded DFAs**, one for M and one for N, is **a member of the language EDFA** of pairs of DFA's that accept the same strings.
(E for Equivalent)
- **The language of interest**, EDFA, is not primarly the languages L(M) and L(N), but rather the language of encodings.

Decidable Languages

- A language is called decidable if there is some Turing machine that accepts exactly that language.

- For any input, the machine must eventually halt and indicate one of:
  - Accept
  - Reject

Algorithms for Decision Problems

- Consider EDFA again. Suppose a problem is represented an encoded pair <M, N> for M and N respectively.
- The question is whether or not <M, N> ∈ EDFA.
- Answering the question **does not necessarily entail running** either M nor N, but rather **analyzing** <M> and <N> for certain characteristics.

Most languages are not decidable

- There is a countably-infinite set of strings over a finite alphabet, thus a countably-infinite set of Turing machine programs.
- So the set of decidable languages on a given alphabet is countably-infinite.
- But there are uncountably-many languages over a given alphabet.

What does “countably infinite” mean?

- It means that the set can be put into 1-1 correspondence with the natural numbers {0, 1, 2, 3, …}.

- For example, these are countably infinite:
  - The set of all pairs of natural numbers: {(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), …}
  - The set of all finite lists of natural numbers

The set of all languages is uncountable

- A similar observation was made by **Georg Cantor**, in his famous **Diagonal Argument**, 1891.

- **Suppose to the contrary** that the set of such languages **is** countable:

  $L_0, L_1, L_2, \ldots$

- Define "diagonal" language D such that

  $D = \{x_i \in \Sigma^* \mid x_i \notin L_i\}$ $(\Sigma^* = \{x_0, x_1, x_2, \ldots\})$

- Define “diagonal” language D such that

$$D = \{x_i \in \Sigma^* \mid x_i \notin L_i\} \qquad (\Sigma^* = \{x_0, x_1, x_2, ...\})$$

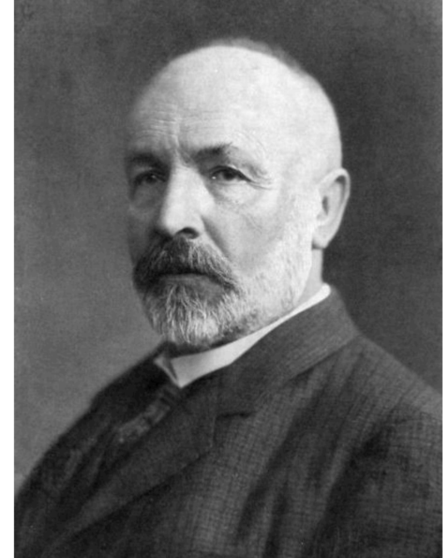
- D cannot be in the list as L_i for any i.

- Suppose D is in the list as, say, L_d.
- Then $L_d = \{x_i \in \Sigma^* \mid x_i \notin L_i\}$.

- But then $x_d \in L_d \rightarrow x_d \notin L_d$, and conversely $x_d \notin L_d \rightarrow x_d \in L_d$, both contradictions.

David Hilbert’s comment on Cantor

"No one shall expel us from the paradise that Cantor has created."



Georg Cantor,
1845-1918

David Hilbert,
1862-1943

Application to Decidable Languages

- The same diagonalization argument that showed that the set of all languages is uncountable can be applied to show that there are languages that are **not decidable**.
- Because every decidable language has a Turing machine that accepts it, and we can enumerate all Turing machines, we can enumerate their languages:

  $L_0, L_1, L_2, \ldots$
- Then define $D = \{x_i \in \Sigma^* \mid x_i \notin L_i\}$ as before. D is not in the list, hence cannot be decidable.

Turing Machines vs. Languages

An enumeration of all Turing machines (distinct)		The languages they compute (with duplicates listed)	
T_0	→	L_0	→ F_0
T_1	→	L_1	→ F_1
T_2	→	L_2	→ F_2
T_3	→	L_3	→ F_3
. . .		. . .	. . .

G

What about Turing machines that compute functions rather than languages?

- Any such machine M can be thought of computing a language, namely the set of strings for which M computes a specified value.

- In the next set, we show specific examples of undecidable languages.