COMPUTABILITY

In this chapter we

- ... highlight Decision Problems
- ... outline a cycle of interpretations.

  Writing $\preccurlyeq$ for "is computing no more than":

  $$\begin{aligned}\text{An imperative language IPS} &\preccurlyeq \text{Turing acceptors}\\ &\preccurlyeq \text{General grammars}\\ &\preccurlyeq \text{IPS}\end{aligned}$$
- ... understand the notion of universal devices
- ... evidence that "computability" has been captured (Turing-Church Thesis)

DECISION PROBLEMS

Decision problems

- A ***decision problem*** or just ***problem*** for short, is a request for an algorithm:

1. ***Instances:*** Finite discrete objects, representable textually..
2. ***Property:*** that the instances may satisfy or not.

- A ***solution*** is an algorithm deciding for each instance whether it satisfies the property.

If such an algorithm exists the problem is ***decidable,*** otherwise it is ***undecidable.***

Example: Composite numbers

- *Instances:* integers > 1.
- *Property:* “is composite”.
- A (poor) decision algorithm:
  Given input]$\boldsymbol{n}$ check for successive numbers $\boldsymbol{k} \leqslant \sqrt{\boldsymbol{n}}$
  whether $\boldsymbol{k} \mid \boldsymbol{n}$ ($\boldsymbol{k}$ divides $\boldsymbol{n}$).

Example: Integer Polynomials

- The ***Integer Polynomials*** problem has an important history, and is also known as ***Hilbert's Tenth Problem***.
- Instances: Polynomials with integer coefficients.
- Property: Evaluates to zero for some integer input.
- Examples:
  - ⋆ $x^2 + x - 2$ has solution $x = 1$ (as well as -2).
  - ⋆ $x^2 + x - 1$ has no integer solution.
  - ⋆ $x^2 + y^2 - z^2$ has solution $x = y = z = 0$ as well as $(3, 4, 5)$, $(5, 12, 13)$ ... (the *Pythagorean triplets*).
- An equivalent formulation:

  Given an equation that uses integers, + and $\times$*,*
  does it have an integer solution.

Problem about finite sets: Integer Partition

- ***Integer partition:***

  Given $S \subseteq \mathbb{N}$ is there a $P \subseteq S$ adding up to half of ΣS.
- That is:
  - ⋆ Instances: Finite set S of positive integers
  - ⋆ Property: Exists $P \subseteq S$ such that $\Sigma P = \Sigma(S - P)$
- Instances implicitly assumed to be given textually:

  $\{2, 4, 5\}$ given as `10#100#101`.
- Examples: $\{2, 3, 4, 5\}$: *yes.*

  $\{2, 3, 4, 6\}$: *no* (total is odd)

  $\{2, 3, 4, 7\}$: *no*

Exact Sum

- ***Exact Sum:***
  Given finite $S \subset \mathbb{N}$ and $t > 0$ (the *"target"*)
  is there $P \subseteq S$ such that $\Sigma P = t$?
- That is,
  - ⋆ Instances: Pairs $\langle S, t \rangle$ as above
  - ⋆ Property: Exists $P \subseteq S$ such that $\Sigma P = t$.
- Examples: $\{1, 2, 3, 5\},\ 6$: yes.
  $\{1, 3, 7\},\ 5$: no.

Decision problem about graphs: Connectivity

- A finite graph $\mathcal{G} = (V, E)$ is **connected** if every pair of distinct vertices is linked by a path.
- ***Connectivity:*** Given a finite undirected finite graph $\mathcal{G}$, is it connected?
  - ⋆ Instances: Finite undirected graph $\mathcal{G} = (V, E)$
  - ⋆ Property: $\mathcal{G}$ is connected
- Implicit assumption: graphs are given textually, e.g. by an adjacency list or a matrix.
- A (poor) decision algorithm: Exhaustive search (“brute force”): for each pair of vertices, try all possible paths.
- There are very efficient decision algorithms for CONNECTIVITY.

Graph problems: Clique

- A ***clique*** in a graph $\mathcal{G} = (V, E)$ is a set $C \subseteq V$, s.t. every distinct $u, v \in C$ are on an edge.
- The ***Clique*** problem:
  Given a finite undirected graph $\mathcal{G}$ and a target $t > 0$, is there a clique in $\mathcal{G}$ with $\geqslant t$ vertices.
- Exhaustive search (“brute force”) solution:
  Try all subsets of size t.

Equation Solvability: Strings

- String expressions (over some fixed alphabet Σ) generated from variables and fixed strings in Σ^* .
- A *solution* of $\mathbf{t} = \psi$ is a binding of string to the variables in the equation, which makes the equation true.
- Example: $x * \mathbf{01} * y = y * \mathbf{10} * x$ has as solution $x = \mathbf{11},\ y = \mathbf{1}$

*G*iven an equation between string-expressions *d*oes it have a solution?

Equation Solvability: Polynomials

- **Monomials** are products $ax_1 \cdots x_k$.
  A **Polynomial** is a sum of monomials.
- Polynomial integer solvability Problem:

> Given a polynomial $P(x_1, \ldots, x_k)$ with integer coefficients,
> does the equation $P(x_1, \ldots, x_k) = 0$ have an integer solution?

The German mathematician David Hilbert presented in 1900 a list of 20 open questions.

Finding an algorithm deciding polynomial solvability was the tenth,

In the 1970's Martin Davis, Yuri Matiyasevich, Hilary Putnam and Julia Robinson showed that this problem is undecidable.

TEXTUAL DECISION PROBLEMS

Every language is a decision problem

- Every $L \subseteq \Sigma^*$ is a decision problem:
  - $\star$ Instances: $w \in \Sigma^*$
  - $\star$ Property: $w \in L$

Every decision problem is a language

- The instances of a decision problem are finite and discrete, so they are codable as text. Examples:
- Natural numbers (e.g. the primes numbers):
  decimal coding: $\mathtt{2, 3, 5, 7, 11}$
  binary coding: $\mathtt{10, 11, 101, 111, 1011}, \ldots$
  unary coding: $\mathtt{II, III, IIIII, IIIIIII, IIIIIIIIIII}$
- Finite sets of natural numbers: $\{2, 3, 5\}$ coded by $\mathtt{10\#11\#101}$
- Matrices: $\begin{pmatrix} \mathtt{2} & \mathtt{3} \\ \mathtt{5} & \mathtt{7} \end{pmatrix}$ coded by $\mathtt{10\#11\#\#101\#111}$.
- Directed graphs: Code the adjacency matrix.
- Turing machines.
- Once we set a coding we write $a^{\#}$ for the code of a.

What about the instances?

- If we code a problem's instances as Σ-strings,
  what about Σ-strings that do not represent instances?
- E.g. for PRIME, strings $w \in \{0,1\}^*$ that are not binary numerals?
  - ⋆ Version 1:
    Does binary numeral w denote a prime ?
    Here non-numerals are not instances.
  - ⋆ Version 2:
    Does string $w \in \{0,1\}^*$ denote a prime ?
    Here non-numerals are instances, for which the answer is *no.*
- We disregard this distinction,
  because for all actual problems it is easy to tell

whether or not a string represents an instance of the problem.

EQUIVALENCE OF COMPUTATION MODELS

AN IMPERATIVE PROGRAMMING LANGUAGE

High-level programming

- Most algorithms are successfully cast in high-level programming languages that support direct representation of relevant data and operations.
- We define a simple high-level imperative language, epitomizing such languages.

Imperative programs

- Broad programming paradigms:
  imperative and *declarative*
- Imperative constructs provide access to the computer's *store,* so can be construed as powerful machines.
  Certain algorithms call for imperative constructs.
- Declarative constructs (functional, relational)
  - abstract away from store/implementation
  - tend to be less efficient
  - but easier to code, understand, verify
- Most modern higher-level programming languages combine both types of constructs.

Denoting strings by terms

- We define an imperative programming language IPS.
  over a single data-type: Σ^* for some fixed alphabet Σ.
- We assume an unbounded supply of reserved identifiers
  we call ***variables.***
- What they are is not important.
  For example we might use $\mathtt{v0}, \mathtt{v1}, \mathtt{v00}, \mathtt{v01} \ldots$,
  that is a $\mathtt{v}$ followed by a string of booleans.
- We use $x, y, z, \ldots$ (with scripts and marks) as
  discourse parameters for variables.
- The set of ***Terms*** (over Σ) is generated inductively:
  - ⋆ Basis: Variables, $\mathtt{e}$ (denoting ε), each symbol in Σ.
  - ⋆ Operations: If t and q are terms
    then so are $t * q$, $hd(t)$ and $tl(t)$.

- Intent: $*$ denotes concatenation: $\mathtt{ab} * \mathtt{ac} = \mathtt{abac}$.

  $hd(\mathtt{abc}) = \mathtt{a}$, $tl(\mathtt{abc}) = \mathtt{bc}$,

  $hd(\varepsilon) = tl(\varepsilon) = \varepsilon$
- A term is ***closed*** if it has no variables.
- Concatenation is associative,

  so parens are not needed for multiple concatenations.
- Strings are syntactic sugar: $\mathtt{011}$ stands for $\mathtt{0} * \mathtt{1} * \mathtt{1}$.

Assignments

- An **assignment** is a phrase $x := t$ (x a variable, t a term)
- Examples:

$$\begin{array}{ll} x := x * \mathsf{a} & x := x * x \\ y := tl(x) & y := tl(y) * hd(y) \\ z := tl(tl(x)) & x := hd(tl(x)) * hd(x) * tl(tl(x)) \end{array}$$

- Non-examples:

$$\begin{array}{ll} tl(x) & := \mathsf{a} * tl(x) \\ \mathsf{a} * x & := \mathsf{a} * tl(x) \end{array}$$

- The reserved identifier ***skip*** stands for an assignment $x := x$.

Composition

- Compound programs are built up using ***composition***, ***branching***, and ***iteration***.
- ***Composition:*** If P and Q are programs, then so is $P;Q$.
- Intended execution: execute P, then execute Q.
- Composition is associative, so no parens are needed:

$$P_1 \,;\, P_2 \,;\, \cdots \,;\, P_n$$

- Example: Swap values of x, y:

$$z := x; \quad x := y; \quad y := z$$

Branching

- An **equation** is a phrase $t = q$ (t, q terms).
- A **guard** is a boolean combination of equations.

- Example: $x \neq \mathsf{e}$ $\ \mathit{or}\ $ $y = x * hd(z)$
- If G is a guard and P, Q are programs, then $if\ (G)\ \{P\}\{Q\}$ is a program.
- Example:

$$if\ (\ x \neq \mathsf{e}\ )\ \{x := tl(x)\}\ \{x := y\}$$

- Using a no-op program $skip$ we define a no-branching conditional:

$$if\ (x = y)\ \{y := z\}$$

abbreviates $if\ (x = y)\ \{y := z\}\ \{skip\}$

Iteration

- If G is a guard, and P a program, then $do\ (G)\ \{P\}$ is a program.
- Example:

```
$y$ := e;
do ($x \neq$ e)
    { $y := y * hd(x) * hd(x)$;
      $x := tl(x)$
    }
```

Input and output

- We use reserved variables `in` and `out`.
  For several inputs use `in0`, `in1`, ...
- Convention: all variables are initially assigned ε,
  except for input variables, which are initialized to the inputs.
- Upon termination, the ***output*** is the value of the variable `out`.

Example: Collect `a`'s

- Task: Place in **`out`** the `a`'s in the input.
- Suggestion: Place input value in a "working variable".
  That way the input remains available for later reference.

```
$x := \mathtt{in};$
$do\ (x \neq \mathtt{e})$
   $if\ (hd(x) = \mathtt{a})$
         $\{\ y := \mathtt{a} * y\ \}$
         $\{\ skip\ \}$
         $\}$
   $x := tl(x)$
   $\}$
$\mathtt{out} := y$
```

Example: Flip

- Task: Output the flip of a string of booleans.

```
$x := \mathtt{in};$
do $(x \neq \mathsf{e})$
    $\{$ if $(hd(x) = \mathsf{e})$
        $\{y := y * 1\}$
        $\{y := y * 0\}$
    $\}$
$\mathtt{out} := y$
```

Example: Reverse

```
$x := \mathtt{in};$
$do \quad (x \neq \mathsf{e})$
        $\{\, if \;\; (x \neq \mathsf{e})$
    $\{y := hd(x) * y\}$
    $\{x := tl(x)\}$
    $\}$
$\mathtt{out} := y$
```

Example: Merge

```
$x := \mathtt{in0};\quad y := \mathtt{in1};$
$do\ (x \neq \mathsf{e}\ \ or\ \ y \neq \mathsf{e})$
        $\{z := z * hd(x);\ z = z * hd(y);$
        $x := tl(x);\quad y := tl(y)$
        $\}$
$\mathtt{out} := z$
```

Stores

- Program execution proceeds in step-by-step calculation,
  each step is changing the memory.
- The “memory” is a binding of values (strings) to identifiers (variables).
  This form of memory is called ***environment*** or ***store***.
- Writing ***Var*** for the set of variables,
  a $\boxed{\textbf{store}}$ is a function $V : \textit{Var} \to \Sigma^*$ (V for “value”.)

A term valuation function

- V can be extended to apply to all terms.
  We overload the symbol V.
  - ⋆ $V(x)$ (x a variable) is defined
    by the given function V.
  - ⋆ $V(\mathsf{e})$ is the empty string.
  - ⋆ $V(\sigma)$ is the letter σ
  - ⋆ $V(t * q)$ is $V(t) \cdot V(q)$
  - ⋆ $V(hd(t))$ is the first symbol of $V(t)$ (but ε if $V(t) = \varepsilon$)
  - ⋆ $V(tl(t))$ is the tail of $V(t)$ (but ε if $V(t) = \varepsilon$)

Program's meaning: transforming stores

- A program is a piece of text.
  Its *meaning*, also called **_semantics,_**
  is a mapping from "input" stores to "final" stores,
  that is a mapping from an initial store, with
  $V(\mathtt{in}) =$ the input, and $V(x) = \varepsilon$ for other variables x,
  to a final store (with `out` containing the output).
- That is, the meaning of a program P is a partial-function

$$[\![P]\!] : \textit{Stores} \rightharpoonup \textit{Stores}$$

where ***Stores*** is the set of stores.

Semantic brackets

- The double-brackets are called *semantic brackets*, and are a common notation of all sorts of semantic mappings, assigning a meaning to syntactic phrases.
- So $[\![P]\!](V)$ is P's output-store for the input-store V. Alternative notation: $V \Rightarrow_P V'$, (where V' is the output-store).
- Note: $[\![P]\!]$ is a *partial* function: when P on input-store V does not terminate, there is no output-store.

Program meaning: Assignments

- Programs P are generated inductively.
  $[\![P]\!]$ is defined by a corresponding recurrence.
- Assignments: If P is $x := \mathbf{t}$ then $V \Rightarrow_P V'$
  where V' is V except that $V'(x) = V(\mathbf{t})$.
- $V \Rightarrow_{\mathbf{SKIP}} V$

Semantics of composition

- $V \Rightarrow_{P;Q} V'$ iff there is a store W such that $V \Rightarrow_P W \Rightarrow_Q V'$.
- That is: $[\![\, P;Q \,]\!] = [\![\, P \,]\!] \circ [\![\, Q \,]\!]$.

  The semantic of program composition is relational composition.

Semantics of branching

- Guards are either *true* or *false* in a store.
- Example: $tl(x) = y * hd(x)$ is true in V

  iff $V(x) = V(y * hd(x))$
- If R is $\textsf{if}\ (G)\ \{P\}\{Q\}$ then

  $V \ \Rightarrow_R\ V'$ iff

  G is true in V and $V \Rightarrow_P V'$

  or

  G is false in V and $V \Rightarrow_Q V'$.

Semantics of iteration

- If R is **do** $(G)\ \{P\}$ then
  $V \Rightarrow_R V'$ iff there are stores $V_0 \ldots V_k$ such that

$$V = V_0 \Rightarrow_P V_1 \Rightarrow_P \cdots \Rightarrow_P V_k = V'$$

and G is true in each V_i $(i < k)$ and false in V_k.

$$\begin{array}{ccccccccccccc} V & \Rightarrow_P & V_1 & \Rightarrow_P & \cdots & \Rightarrow_P & V_i & \Rightarrow_P & \cdots & \Rightarrow_P & V_{k-1} & \Rightarrow_P & V' \\ \uparrow & & \uparrow & & & & \uparrow & & & & \uparrow & & \uparrow \\ G & & G & & & & G & & & & G & & G \\ \text{true} & & \text{true} & & & & \text{true} & & & & \text{true} & & \text{false} \end{array}$$

Representing stores

Store $\begin{array}{cccc} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_n \\ \downarrow & \downarrow & & \downarrow \\ w_1 & w_2 & \cdots & w_n \end{array}$

represented by string $\$\, w_1 \,\$\, w_2 \,\$ \,\cdots\, \$\, w_n \,\$$

- By induction on programs:
  for each program P obtain Turing transducer M_P
  that computes $\Rightarrow_P$.

From imperative programs to Turing transducers

Converting programs with output to Turing transducers

- We convert programs P to equivalent Turing machines T_P .
- T_P is ***equivalent*** to P in the sense that:

$$[\, u_1,\ u_2,\ \cdots\ u_n \,] \quad \Longrightarrow_P \quad [\, v_1,\ v_2,\ \cdots\ v_n \,]$$

converts to

$$> \$\, u_1 \,\$\, u_2 \,\$\, \cdots\ \$\, u_n \,\$ \quad \Longrightarrow_{T_P} \quad > \$\, v_1 \,\$\, v_2 \,\$\, \cdots\ \$\, v_n \,\$$$

- So T_P computes a coded version of $\llbracket P \rrbracket$.
- Set $\mathtt{in}, \mathtt{out}$ variables as x_1, x_2.
  The final Turing transducer interpreting P :

  1. Converts its input w to P's initial store for w :
     $> \$\, w\, \$\, \$ \cdots \$\, \$$ (w as the value of x_1, other variables set to ε.)
  2. Computes $\llbracket P \rrbracket$ over representations of the store.
  3. Converts the final store $\$\, u_1, \$, u_2\, \$ \cdots \$\, u_n\, \$$ to P's output
     $> u_2 \sqcup \ldots$.

UNIVERSAL INTERPRETERS

Interpreting Turing transducers by programs

- Showed how to convert programs to equivalent Turing transducers.
- Reverse should be trivial: interpreting the feeble by the mighty.
- It’ll still be useful!

Cfg

$\langle q,\ u\,\underline{\sigma}\,v\rangle$

represented by

Store

`state`	`left`	`cursor`	`right`
↓	↓	↓	↓
q	u^R	σ	v

- Note the reversal of `left` ,
  so that the head be the last symbol of its value.

Cfg

$\langle \mathtt{A},\ \mathtt{>0111}\,\underline{\mathtt{0}}\,\mathtt{01}\sqcup\sqcup\rangle$

represented by

Store

`state`	`left`	`cursor`	`right`
↓	↓	↓	↓
`A`	`1110>`	`0`	`01⊔⊔`

Interpreting overwrite action

- Machine action: $\mathtt{A} \xrightarrow{0\ (1)} \mathtt{B}$
- Program action:

$$\text{if} \quad (\mathtt{state} = \mathtt{A} \quad \text{and} \quad \mathtt{cursor} = 0) \\ \{\mathtt{state} := \mathtt{B};\ \mathtt{cursor} := 1\}$$

Interpreting left move

- Machine action: $\mathtt{A} \xrightarrow{\mathtt{0(-)}} \mathtt{B}$
- Program action:

```
if ($\mathtt{state} = \mathtt{A}$ and $\mathtt{cursor} = 0$)
  $\{\mathtt{state} := \mathtt{B};$
  if ($\mathtt{left} \neq \mathrm{e}$)
      $\mathtt{right} := \mathtt{cursor} * \mathtt{right};$
      $\mathtt{cursor} := hd(\mathtt{left});$
      $\mathtt{left} := tl(\mathtt{left})\}$
```

Interpreting Turing transducers (example)

- Machine transitions: $\mathtt{S} \xrightarrow{0\,(+)} \mathtt{A}$ $\mathtt{A} \xrightarrow{0\,(1)} \mathtt{B}$ $\mathtt{A} \xrightarrow{1\,(-)} \mathtt{B}$
- Initializing:

  $\mathtt{state} := \mathrm{S};$ $\mathtt{left} := \mathrm{e};$ $\mathtt{cursor} := >;$ $\mathtt{right} := \mathtt{in}$
- Execution:

```
do (a transition applies)
    { if  $(\mathtt{state} = \mathtt{S}$  and  $\mathtt{cursor} = 0)$
          $\{$    $\cdots$    $\}$ $\{\ldots\}$;
    { if  $(\mathtt{state} = \mathtt{A}$  and  $\mathtt{cursor} = 0)$
          $\{$    $\cdots$    $\}$ ...
```

THEOREM:

If a partial function is computable by a Turing transducer,
then it is computable by a program.

- What's the point of this theorem?
- Answer:
  Towards a uniform automation of compilation.
- Early 1950s' computers were programmed mannually,
  by re-switching the radio-tube components for each new task.
- See `http://www.columbia.edu/cu/computinghistory/enia`

A universal interpreter

- Consider toy machines first, all with:
  - I/O alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
    Machine alphabet: $\Gamma = \{\mathtt{0}, \mathtt{1}, \sqcup, >\}$.
  - Up to five states $\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}$ with $\mathtt{A}$ initial and $\mathtt{B}$ print state.
- Textual coding of machines:
  - Transition entry: $q\,\gamma\,\alpha\,p$
    $q, p \in \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}$,
    $\sigma \in \Gamma$,
    $\alpha \in \{\mathtt{0}, \mathtt{1}, \sqcup, -, +\}$.
  - Transition function: the transition entries separated by \$.

Example

- Transducer M is

$$\mathtt{A} \xrightarrow{>(+)} \mathtt{C}, \quad \mathtt{C} \xrightarrow{0,1(+)} \mathtt{C}, \quad \mathtt{C} \xrightarrow{\llcorner\lrcorner(1)} \mathtt{B}$$

- Represented by

$$M^{\#} \quad = \quad \mathtt{A} > + \mathtt{C} \,\$\, \mathtt{C}\, 0 + \mathtt{C} \,\$\, \mathtt{C}\, 1 + \mathtt{C} \,\$\, \mathtt{C} \sqcup 1\, \mathtt{B}$$

Program interpreters

- Program *Int* is an $\boxed{\textit{interpreter}}$ for toy Turing transducer if for each M and $w \in \{0,1\}^*$,

$$\textit{Int} \quad \text{on input} \quad M^{\#}\Box w \quad \text{returns} \quad u$$

IFF

$$M \quad \text{on input} \quad w \quad \text{returns} \quad u$$

- We use $\Box$ as a mega-separator, so as to have just one input string.

Interpreter for toy Turing transducers

- Main variables:
  - ⋆ `machine`: safely keep a copy of $M^{\#}$
  - ⋆ `state`, `left`, `cursor` and `right`
  - ⋆ `search`, and a working copy of $M^{\#}$
  - ⋆ The current transition, stored in variables
    `instate`, `insymbol`, `action`, and `outstate`.

- Broad outline of the interpreter:

```
Initialize;
do    ($\mathtt{continue} = \mathrm{e}$)
          {Yield};
          Extract
```

- **Initialize** extracts from the variable `input` the transition table of M and the initial configuration.
- The variable `continue` is ε iff a terminal cfg has been encountered.
- The program-segment **Yield** searches for a transition applicable to current cfg.
  When found, `state`, `left`, `cursor`, `right` are updated.
- **Extract** extracts the string in the “output block” of the final cfg, and assigns it to the program variable `out` .

Interpreters made more universal

- Our universal interpreter **UPT** for Turing transducers
  has two inputs: a (textually presented) Turing machine, and a binary string.
- What about Turing machines with more states?
  and with alphabets with more symbols?
- Use an alphabet containing $\mathtt{s}, \mathtt{d}, \mathtt{0}, \mathtt{1}$.
  Represent
  each state by a `s`-followed-by-boolean-string
  each letter by a `d`-followed-by-boolean-string
- Now we have a univeral program **UPT** for *all* Turing transducers.

Interpreters in other directions

- Can we have a Turing transducer interpreting *Turing transducers?*
- Absolutely! Just compile **UPT** into a Turing transducer **UTT**.
- What about a program interpreter **UPP** for all programs?
- Definition of **UPP**:
  on given program P and its input-string w:

1. Compile P into a Turing machine M.
2. Apply **UPT** to M and w.

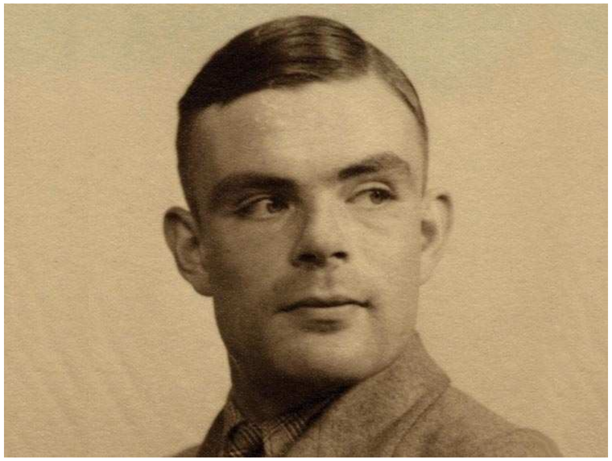
The Turing-Church Thesis

Turing-Church Thesis:

The notion of computability is completely captured

by Turing machines.

- Here a “thesis” means a declaration of faith, no rigorous proof is possible (circularity).
- It can be shaken, but never definitively confirmed.



Evidence for TC Thesis: hardware

1. Information is based on discrete and unambiguous representation,
   and can therefore be given by discrete and recognized symbols
   laid out in space.
2. Such layouts can be reduced to a one-dimensional layout,
   because a discrete space can be equipped with addresses.
3. Computation is a discrete process:
   separate steps, specific rules.
4. So any computing device has discrete *states*,
   and a finite set of *transition rules*.
5. A computing device navigates through data, and access it.
   No loss in limiting to single symbol per step.

6. Device can modify data in an implementable way.

No loss in limiting to single symbol

Evidence for TC: stability of computability

- Stability of computability.
  symbolic computation (grammars and rewrite-systems),
  functional abstraction (lambda calculus),
  recurrence and search
    (general recursive functions, functional programs)
  programming languages.
- Equivalences are implementable and feasible.
- Equivalences are uniform and systematic.

So why Turing machines (or other simple hardware) ?

- Why referring primarily to Turing machines?

1. Direct relation to hardware, hence to the analysis of computation itself.
2. Isolates the central and essential aspects of computing:
    finite number of states;
    discrete and addressable memory
    unbounded but finite
    local and discrete actions
3. Simpler computation models are more easily emulated,
    so showing universality for other models is easier using TMs.
4. Realistic estimate of resources:
    e.g. a PS can have a huge output in very few steps.