Introduction to theory of computation

Tom Carter

http://astarte.csustan.edu/~ tom/SFI-CSSS

Complex Systems Summer School

June, 2005

Our general topics: ←

- ◎ Symbols, strings and languages
- ◎ Finite automata
- ◎ Regular expressions and languages
- ◎ Markov models
- ◎ Context free grammars and languages
- ◎ Language recognizers and generators
- ◎ The Chomsky hierarchy
- ◎ Turing machines
- ◎ Computability and tractability
- ◎ Computational complexity
- ◎ References

The quotes ↞

- ◎ No royal road
- ◎ Mathematical certainty
- ◎ I had a feeling once about Mathematics
- ◎ Terminology (philosophy and math)
- ◎ Rewards

To topics ←

Introduction ←

What follows is an extremely abbreviated look at some of the important ideas of the general areas of automata theory, computability, and formal languages. In various respects, this can be thought of as the elementary foundations of much of computer science. The area also includes a wide variety of tools, and general categories of tools . . .

Symbols, strings and languages ←

- The classical theory of computation traditionally deals with processing an input string of symbols into an output string of symbols. Note that in the special case where the set of possible output strings is just {'yes', 'no'}, (often abbreviated {T, F} or {1, 0}), then we can think of the string processing as string (pattern) recognition.

  We should start with a few definitions. The first step is to avoid defining the term 'symbol' – this leaves an open slot to connect the abstract theory to the world . . .

  We define:

  1. An *alphabet* is a finite set of symbols.

2. A *string* over an alphabet A is a finite ordered sequence of symbols from A. Note that repetitions are allowed. The length of a string is the number of symbols in the string, with repetitions counted. (e.g., $|\text{aabbcc}| = 6$)

3. The empty string, denoted by ϵ, is the (unique) string of length zero. Note that the empty string ϵ is not the same as the empty set $\emptyset$.

4. If S and T are sets of strings, then $\text{ST} = \{xy |\ x \in \text{S} \text{ and } y \in \text{T}\}$

5. Given an alphabet A, we define

$$\begin{aligned} \text{A}^0 &= \{\epsilon\} \\ \text{A}^{n+1} &= \text{AA}^n \\ \text{A}^* &= \bigcup_{n=0}^{\infty} \text{A}^n \end{aligned}$$

6. A *language* L over an alphabet A is a subset of A^*. That is, $\text{L} \subset \text{A}^*$.

- We can define the natural numbers, $\mathbb{N}$, as follows:

  We let

$$\begin{aligned} 0 &= \emptyset \\ 1 &= \{\emptyset\} \\ 2 &= \{\emptyset, \{\emptyset\}\} \\ \text{and} &\ \text{in} \ \text{general} \\ n+1 &= \{0, 1, 2, \ldots, n\}. \\ \text{Then} & \\ \mathbb{N} &= \{0, 1, 2, \ldots\}. \end{aligned}$$

- Sizes of sets and countability:

  1. Given two sets S and T, we say that they are the same size ($|\mathsf{S}| = |\mathsf{T}|$) if there is a one-to-one onto function $f : \mathsf{S} \to \mathsf{T}$.

  2. We write $|\mathsf{S}| \leq |\mathsf{T}|$ if there is a one-to-one (not necessarily onto) function $f : \mathsf{S} \to \mathsf{T}$.

3. We write $|\mathsf{S}| < |\mathsf{T}|$ if there is a one-to-one function $f : \mathsf{S} \to \mathsf{T}$, but there does not exist any such onto function.

4. We call a set S

   (a) Finite if $|\mathsf{S}| < |\mathbb{N}|$

   (b) Countable if $|\mathsf{S}| \leq |\mathbb{N}|$

   (c) Countably infinite if $|\mathsf{S}| = |\mathbb{N}|$

   (d) Uncountable if $|\mathbb{N}| < |\mathsf{S}|$.

5. Some examples:

   (a) The set of integers $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ is countable.

   (b) The set of rational numbers $\mathbb{Q} = \{p/q \mid p, q \in \mathbb{Z}, q \neq 0\}$ is countable.

(c) If S is countable, then so is SxS, the cartesian product of S with itself, and so is the general cartesian product S^n for any $n < \infty$.

(d) For any nonempty alphabet A, A^* is countably infinite.

Exercise: Verify each of these statements.

6. Recall that the *power set* of a set S is the set of all subsets of S:

$$P(\mathsf{S}) = \{\mathsf{T} \mid \mathsf{T} \subset \mathsf{S}\}.$$

We then have the fact that for any set S,

$$|\mathsf{S}| < |P(\mathsf{S})|.$$

Pf: First, it is easy to see that

$$|\mathsf{S}| \leq |P(\mathsf{S})|$$

since there is the one-to-one function $f : \mathsf{S} \to P(\mathsf{S})$ given by $f(s) = \{s\}$ for $s \in \mathsf{S}$.

On the other hand, no function $f : \mathsf{S} \to P(\mathsf{S})$ can be onto. To show this, we need to exhibit an element of $P(\mathsf{S})$ that is not in the image of f. For any given f, such an element (which must be a subset of S) is

$$\mathsf{R}_f = \{x \in \mathsf{S} \mid x \notin f(x)\}.$$

Now suppose, for contradiction, that there is some $s \in \mathsf{S}$ with $f(s) = \mathsf{R}_f$. There are then two possibilities: either $s \in f(s) = \mathsf{R}_f$ or $s \notin f(s) = \mathsf{R}_f$. Each of these leads to a contradiction:

If $s \in f(s) = \mathsf{R}_f$, then by the definition of R_f, $s \notin f(s)$. This is a contradiction.

If $s \notin f(s) = \mathsf{R}_f$, then by the definition of R_f, $s \in \mathsf{R}_f = f(s)$. Again, a contradiction.

Since each case leads to a contradiction, no such s can exist, and hence f is not onto. QED

- From this, we can conclude for any countably infinite set S, $P(\mathsf{S})$ is uncountable. Thus, for example, $P(\mathbb{N})$ is uncountable. It is not hard to see that the set of real numbers, $\mathbb{R}$, is the same size as $P(\mathbb{N})$, and is therefore uncountable.

  Exercise: Show this. (Hint: show that $\mathbb{R}$ is the same size as $(0,1) = \{x \in \mathbb{R} \mid 0 < x < 1\}$, and then use the binary representation of real numbers to show that $|P(\mathbb{N})| = |(0,1)|$).

- We can also derive a fundamental (non)computability fact:

  There are languages that cannot be recognized by any computation. In other words, there are languages for which there cannot exist any computer algorithm to determine whether an arbitrary string is in the language or not.

To see this, we will take as given that any computer algorithm can be expressed as a computer program, and hence, in particular, can be expressed as a finite string of ascii characters. Therefore, since ASCII^* is countably infinite, there are at most countably many computer algorithms/programs. On the other hand, since a language is any arbitrary subset of A^* for some alphabet A, there are uncountably many languages, since there are uncountably many subsets.

No royal road ↞

There is no royal road to logic, and really valuable ideas can only be had at the price of close attention. But I know that in the matter of ideas the public prefer the cheap and nasty; and in my next paper I am going to return to the easily intelligible, and not wander from it again.

– C.S. Peirce in How to Make Our Ideas Clear, 1878

Finite automata ←

- This will be a quick tour through some of the basics of the abstract theory of computation. We will start with a relatively straightforward class of machines and languages – deterministic finite automata and regular languages.

  In this context when we talk about a machine, we mean an abstract rather than a physical machine, and in general will think in terms of a computer algorithm that could be implemented in a physical machine. Our descriptions of machines will be abstract, but are intended to be sufficiently precise that an implementation could be developed.

- A deterministic finite automaton (DFA) $\mathsf{M} = (\mathsf{S}, \mathsf{A}, s_0, \delta, \mathsf{F})$ consists of the following:

  S, a finite set of states,

  A, an alphabet,

  $s_0 \in \mathsf{S}$, the start state,

  $\delta : \mathsf{SxA} \to \mathsf{S}$, the transition function, and

  $\mathsf{F} \subset \mathsf{S}$, the set of final (or accepting) states of the machine.

  We think in terms of feeding strings from A^* into the machine. To do this, we extend the transition function to a function

  $$\hat{\delta} : \mathsf{SxA}^* \to \mathsf{S}$$

  by

  $$\begin{aligned} \hat{\delta}(s, \epsilon) &= s, \\ \hat{\delta}(s, xa) &= \hat{\delta}(\delta(s, a), x). \end{aligned}$$

We can then define the language of the machine by

$$L(\mathsf{M}) = \{x \in \mathsf{A}^* \mid \widehat{\delta}(s_0, x) \in \mathsf{F}\}.$$

In other words, $L(\mathsf{M})$ is the set of all strings in A^* that move the machine via its transition function from the start state s_0 into one of the final (accepting) states.

We can think of the machine M as a recognizer for $L(\mathsf{M})$, or as a string processing function

$$f_{\mathsf{M}} : \mathsf{A}^* \to \{1, 0\}$$

where $f_{\mathsf{M}}(x) = 1$ exactly when $x \in L(\mathsf{M})$.

- There are several generalizations of DFAs that are useful in various contexts. A first important generalization is to add a nondeterministic capability to the machines. A nondeterministic finite automaton (NFA) $\mathsf{M} = (\mathsf{S}, \mathsf{A}, s_0, \delta, \mathsf{F})$ is the same as a DFA except for the transition function:

S, a finite set of states,

A, an alphabet,

$s_0 \in$ S, the start state,

δ : SxA $\rightarrow P$(S), the transition function,

F $\subset$ S, the set of final (or accepting) states of the machine.

For a given input symbol, the transition function can take us to any one of a set of states.

We extend the transition function to $\widehat{\delta}$: SxA* $\rightarrow P$(S) in much the same way:

$$\begin{aligned} \widehat{\delta}(s, \epsilon) &= s, \\ \widehat{\delta}(s, xa) &= \bigcup_{r \in \delta(s,a)} \widehat{\delta}(r, x). \end{aligned}$$

We define the language of the machine by

$$L(\mathsf{M}) = \{x \in \mathsf{A}^* \mid \widehat{\delta}(s_0, x) \cap \mathsf{F} \neq \emptyset\}.$$

A useful fact is that DFAs and NFAs define the same class of languages. In particular, given a language L, we have that L = L(M) for some DFA M if and only if L = L(M′) for some NFA M′.

Exercise: Prove this fact.

In doing the proof, you will notice that if L = L(M) = L(M′) for some DFA M and NFA M′, and M′ has n states, then M might need to have as many as 2^n states. In general, NFAs are relatively easy to write down, but DFAs can be directly implemented.

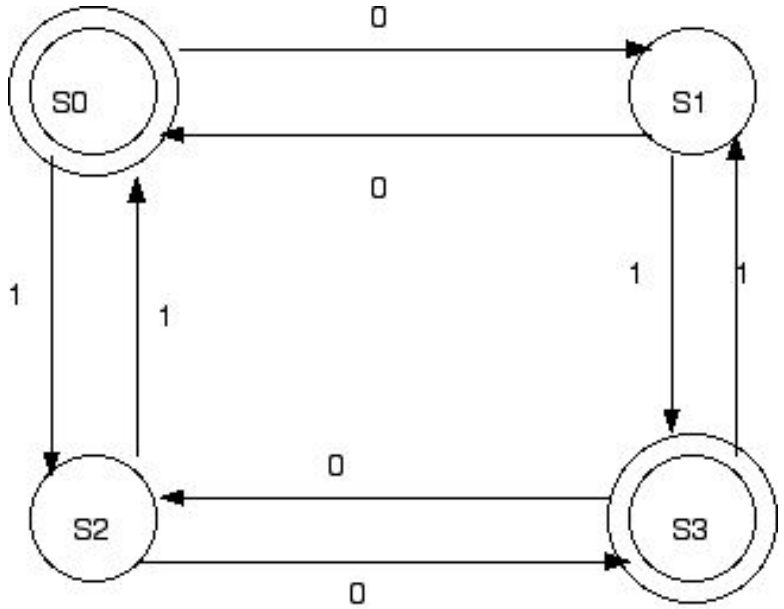
- Another useful generalization is to allow the machine to change states without any input (often called ϵ-moves). An NFA with ϵ-moves would be defined similarly to an NFA, but with transition function

$$\delta : \mathrm{S} \times (\mathrm{A} \cup \{\epsilon\}) \to P(\mathrm{S}).$$

Exercise: What would an appropriate extended transition function $\hat{\delta}$ and language L(M) be for an NFA with ϵ-moves?

Exercise: Show that the class of languages defined by NFAs with ϵ-moves is the same as that defined by DFAs and NFAs.

Here is a simple example. By convention, the states in F are double circled. Labeled arrows indicate transitions. Exercise: what is the language of this machine?



Regular expressions and languages ←

- In the preceding section we defined a class of machines (Finite Automata) that can be used to recognize members of a particular class of languages. It would be nice to have a concise way to describe such a language, and furthermore to have a convenient way to generate strings in such a language (as opposed to having to feed candidate strings into a machine, and hoping they are recognized as being in the language . . .).

  Fortunately, there is a nice way to do this. The class of languages defined by Finite Automata are called Regular Languages (or Regular Sets of strings). These languages are described by Regular Expressions. We define these as follows. We will use lower case letters for regular expressions, and upper case for regular sets.

- Definition: Given an alphabet A, the following are regular expressions / regular sets over A:

Expressions :	Sets :
$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a, for $a \in$ A	$\{a\}$, for $a \in$ A

If r and s are regular, then so are :	If R and S are regular, then so are :
r + s	R ∪ S
rs	RS
r^*	R^*

and nothing else is regular.

We say that the regular expression on the left represents the corresponding regular set on the right. We say that a language is a regular language if the set of strings in the language is a regular set.

A couple of examples:

- The regular expression

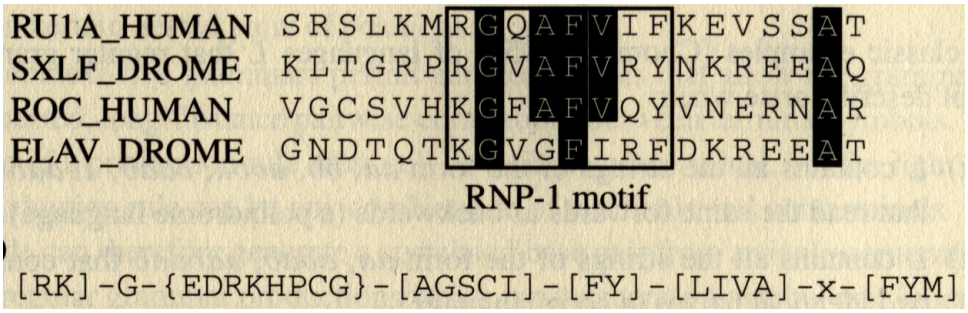
$$(00 + 11)^*(101 + 110)$$

represents the regular set (regular language)

$\{101, 110, 00101, 00110, 11101, 11110,$
$0000101, 0000110, 0011101, 0011110, \ldots\}.$

Exercise: What are some other strings in this language? Is 00110011110 in the language? How about 00111100101110?

- A protein motif pattern, described as a (slight variation of our) regular expression.



- It is a nice fact that regular languages are exactly the languages of the finite automata defined in the previous section. In particular, a language L is a regular set (as defined above) if and only if L $= L$(M) for some finite automaton M.

  The proof of this fact is relatively straightforward.

  For the first half, we need to show that if L is a regular set (in particular, if it is represented by a regular expression r), then L $= L$(M) for some finite automaton M. We can show this by induction on the size of the regular expression r. The basis for the induction is the three simplest cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$. (Exercise: find machines for these three cases.) We then show that, if we know how to build machines for R and S, then we can build machines for R $\cup$ S, RS, and R*. (Exercise: Show how to do these three – use NFAs with ϵ-moves.)

For the second half, we need to show that if we are given a DFA M, then we can find a regular expression (or a regular set representation) for $L(\mathsf{M})$. We can do this by looking at sets of strings of the form

$\mathsf{R}_{ij}^k = \{x \in \mathsf{A}^* \mid x$ takes M from state s_i to state s_j without going through (into and out of) any state s_m with $m \geq k\}$.

Note that if the states of M are $\{s_0, s_1, \ldots, s_{n-1}\}$, then

$$L(\mathsf{M}) = \bigcup_{s_j \in \mathsf{F}} \mathsf{R}_{0j}^n.$$

We also have

$$\mathsf{R}_{ij}^0 = \{a \in \mathsf{A} \mid \delta(s_i, a) = s_j\}$$

(for $i = j$, we also get $\epsilon \ldots$),

and, for $k \geq 0$,

$$\mathsf{R}_{ij}^{k+1} = \mathsf{R}_{ij}^k \cup \mathsf{R}_{ik}^k (\mathsf{R}_{kk}^k)^* \mathsf{R}_{kj}^k.$$

Exercise: Verify, and finish the proof.

Mathematical certainty ↞

I wanted certainty in the kind of way in which people want religious faith. I thought that certainty is more likely to be found in mathematics than elsewhere. But I discovered that many mathematical demonstrations, which my teachers expected me to accept, were full of fallacies, and that, if certainty were indeed discoverable in mathematics, it would be in a new field of mathematics, with more solid foundations than those that had hitherto been thought secure. But as the work proceeded, I was continually reminded of the fable about the elephant and the tortoise. Having constructed an elephant upon which the mathematical world could rest, I found the elephant tottering, and proceeded to construct a tortoise to keep the elephant from falling. But the tortoise was no more secure than the elephant, and after some twenty years of very arduous toil, I

came to the conclusion that there was nothing more that I could do in the way of making mathematical knowledge indubitable.

– Bertrand Russel in Portraits from Memory

Markov models ←

- An important related class of systems are Markov models (often called Markov chains). These models are quite similar to finite automata, except that the transitions from state to state are probabilistically determined, and typically we do not concern ourselves with final or accepting states. Often, in order to keep track of the (stochastic) transitions the system is making, we have the system emit a symbol, based either on which transition occurred, or which state the system arrived in. These machines take no input (except whatever drives the probabilities), but do give output strings.

  Markov models can often be thought of as models for discrete dynamical systems. We model the system as consisting of a finite set of states, with a certain probability of transition from a state to any other state.

The typical way to specify a Markov model is via a transition matrix:

$$\mathsf{T} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix}$$

where $0 \leq p_{ij} \leq 1$, and $\sum_j p_{ij} = 1$.

Each entry p_{ij} tells the probability the system will go from state s_i to state s_j in the next time step.

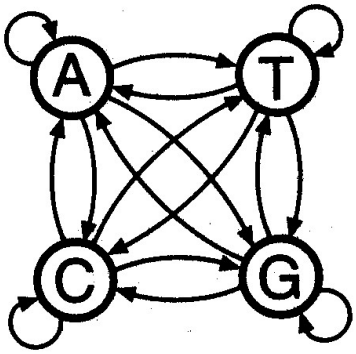
The transition probabilities over two steps are given by T^2. Over n steps, the probabilities are given by T^n.

Exercises: Suppose we run the system for very many steps. How might we estimate the relative probabilities of being in any given state?

What information about the system might we get from eigenvalues and eigenvectors of the matrix T?

- A couple of examples:

  First, a generic Markov model for DNA sequences:



  An outline for a more complex model, of a type often called a hidden Markov model.

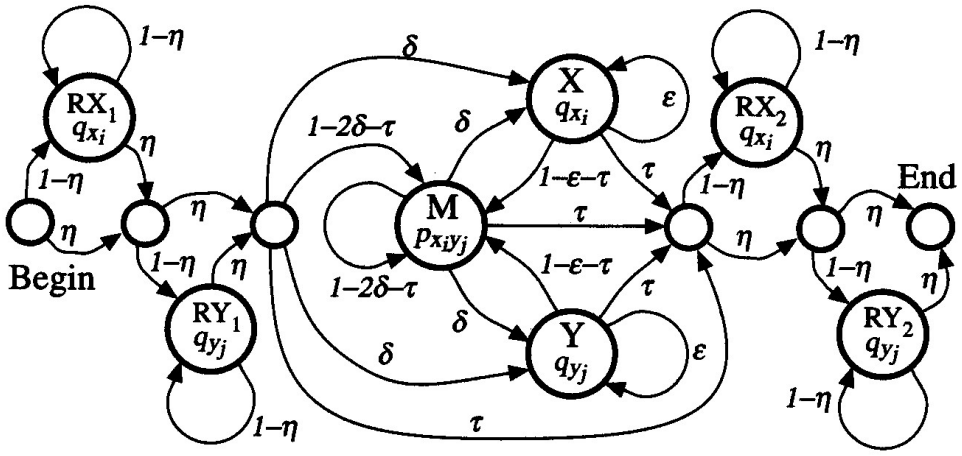


Figure 4.3 *A pair HMM for local alignment. This is composed of the global model (states* M, X *and* Y*) flanked by two copies of the random model (states* RX_1, RY_1 *and* RX_2, RY_2*).*

I had a feeling once about Mathematics ↞

I had a feeling once about Mathematics - that I saw it all. Depth beyond depth was revealed to me - the Byss and Abyss. I saw - as one might see the transit of Venus or even the Lord Mayor's Show - a quantity passing through infinity and changing its sign from plus to minus. I saw exactly why it happened and why the tergiversation was inevitable but it was after dinner and I let it go.

– Sir Winston Churchill

Context free grammars and languages ←

- While regular languages are very useful, not every interesting language is regular. It is not hard to show that even such simple languages as balanced parentheses or palindromes are not regular. (Here is probably a good place to remind ourselves again that in this context, a language is just a set of strings . . .)

  A more general class of languages is the context free languages. A straightforward way to specify a context free language is via a context free grammar. Context free grammars can be thought of as being for context free languages the analogue of regular expressions for regular languages. They provide a mechanism for generating elements on the language.

- A context free grammar G = (V, T, S, P) consists of a two alphabets V and T (called variables and terminals, respectively), an element S $\in$ V called the start symbol, and a finite set of production rules P. Each production rule is of the form $A \to \alpha$, where A $\in$ V and $\alpha \in (\mathsf{V} \cup \mathsf{T})^*$.

  We can use such a production rule to generate new strings from old strings. In particular, if we have the string $\gamma A \delta \in (\mathsf{V} \cup \mathsf{T})^*$ with A $\in$ V, then we can produce the new string $\gamma\alpha\delta$. The application of a production rule from the grammar G is often written $\alpha \underset{\mathsf{G}}{\Rightarrow} \beta$, or just $\alpha \Rightarrow \beta$ if the grammar is clear. The application of 0 or more production rules one after the other is written $\alpha \overset{*}{\Rightarrow} \beta$.

  The language of the grammar G is then

  $$L(\mathsf{G}) = \{\alpha \in \mathsf{T}^* \mid \mathsf{S} \overset{*}{\Rightarrow} \alpha\}.$$

  The language of such a grammar is called a context free language.

- Here, as an example, is the language consisting of strings of balanced parenthese. Note that we can figure out which symbols are variables, since they all occur on the left side of some production.

$$\begin{aligned} \mathsf{S} &\rightarrow \mathsf{R} \\ \mathsf{R} &\rightarrow \epsilon \\ \mathsf{R} &\rightarrow (R) \\ \mathsf{R} &\rightarrow RR \end{aligned}$$

  Exercise: Check this. How would we modify the grammar if we wanted to include balanced '[]' and '$\{\}$' pairs also?

- Here is a palindrome language over the alphabet $\mathsf{T} = \{\mathsf{a}, \mathsf{b}\}$:

$$\begin{aligned} \mathsf{S} &\rightarrow \mathsf{R} \\ \mathsf{R} &\rightarrow \epsilon \mid \mathsf{a} \mid \mathsf{b} \\ \mathsf{R} &\rightarrow \mathsf{aRa} \mid \mathsf{bRb} \end{aligned}$$

  (note the '|' to indicate alternatives . . .)

Exercise: What would a grammar for simple algebraic expressions (with terminals T = {x, y, +, -, *, (,)}) look like?

- A couple of important facts are that any regular language is also context free, but there are context free languages that are not regular.

  Exercise: How might one prove these facts?

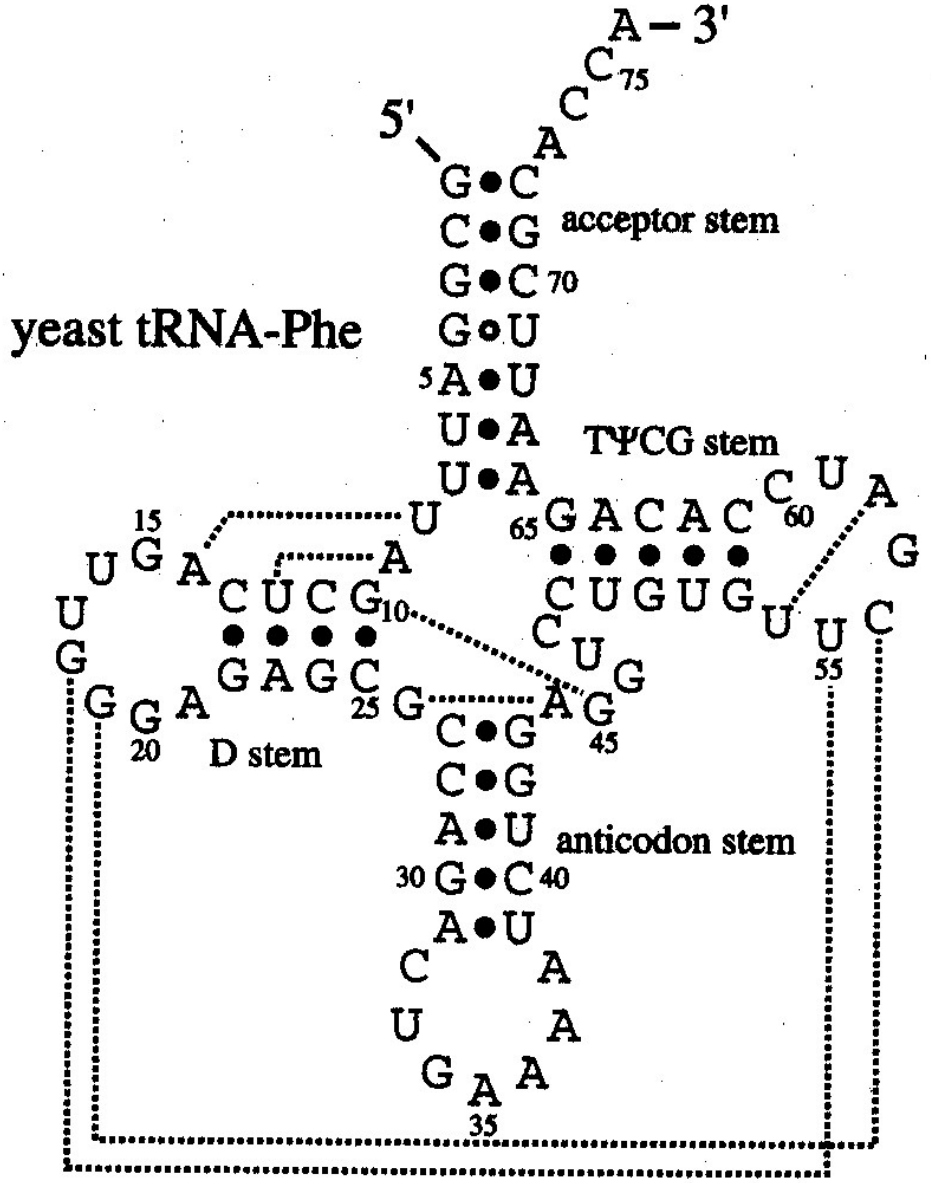
- Context free grammars can easily be used to generate strings in the corresponsing language. We would also like to have machines to recognize such languages. We can build such machines through a slight generalization of finite automata. The generalization is to add a 'pushdown stack' to our finite automata. These more powerful machines are called pushdown automata, or PDAs . . .

- Here is an example – this is a grammar for the RNP-1 motif from an earlier example. This also gives an example of what a grammar for a regular language might look like. Question: what features of this grammar reflect the fact that it is for a regular language?

  RNP-1 motif grammar

$$
\begin{aligned}
S &\rightarrow rW_1 \mid kW_1 \\
W_1 &\rightarrow gW_2 \\
W_2 &\rightarrow [afilmnqstvwy]W_3 \\
W_3 &\rightarrow [agsci]W_4 \\
W_4 &\rightarrow fW_5 \mid yW_5 \\
W_5 &\rightarrow [liva]W_6 \\
W_6 &\rightarrow [acdefghiklmnpqrstvwy]W_7 \\
W_7 &\rightarrow f \mid y \mid m
\end{aligned}
$$

- Could we build a context free grammar for the primary structure of this tRNA that would reflect the secondary structure? What features could be variable, and which must be fixed in order for the tRNA to function appropriately in context?



Terminology (philosophy and math) ↞

Somebody once said that philosophy is the misuse of a terminology which was invented just for this purpose. In the same vein, I would say that mathematics is the science of skillful operations with concepts and rules invented just for this purpose.

– Eugene Wigner

Language recognizers and generators ←

- There is a nice duality between the processes of generating and recognizing elements of a language. Often what we want is a (reasonably efficient) machine for recognizing elements of the language. These machines are sometimes called parsers, or syntax checkers, or lexical analyzers. If you have a language compiler (for C++, for example), typically a first pass of the compiler does this syntax checking.

  The other thing we are likely to want is a concise specification of the syntax rules for the language. These specifications are often called grammars. We have looked briefly at context free grammars. A more general form of grammars are the context sensitive grammars.

- A grammar for a language gives a concise specification of the pattern a string must reflect in order to be part of the language. We have described grammars as generators of the languages, in the sense that given the grammar, we can (ordinarily) generate strings in the language in a methodical way.

  In practice, we often use grammars as guides in generating language strings to accomplish particular tasks. When we write a C++ program, the grammar tells us which symbols are allowed next in our program. We choose the particular symbol – we generate the string, constrained by the grammar.

  Another thing we would like our grammar to do is provide a mechanism for constructing a corresponding language recognizing machine. This process has largely been automated for classes of

languages like the regular, context free, and context sensitive languages. For unix people, the ‘re’ in ‘grep’ stands for ‘regular expression,’ and the tool ‘lex’ is a lexical analyzer constructor.

- Of course by now you are wondering about the relationships between languages and meaning . . .

  In the context of at least some computer languages, there is a way we can begin to approach this question. We can say that the meaning of a language element is the effect it has in the world (in particular, in a computer when the program is run). Thus, for example, in C++, the statement (language element)

  $$j = 5;$$

  has the effect of finding the memory location associated with the name ‘j’, and updating that location to the value 5.

There are some useful tools, such as ‘yacc’ (yet another compiler compiler) (the Gnu public license version of this is called Bison . . .) that do a decent job of automating the process of carrying out these linkages between language elements and hardware effects . . .

Perhaps another day I’ll write another section (or another piece) on ‘meaning.’

For now, let me just leave this with Wittgenstein’s contention that meaning is use . . .

Rewards

Down the line, you can find all kinds of constraints and openings for freedom, limitations on violence, and so on that can be attributed to popular dissidence to which many individuals have contributed, each in a small way. And those are tremendous rewards. There are no President's Medals of Honour, or citations, or front pages in the New York Review of Books. But I don't think those are much in the way of genuine rewards, to tell you the truth. Those are the visible signs of prestige; but the sense of community and solidarity, of working together with people whose opinions and feelings really matter to you... that's much more of a reward than the institutionally-accepted ones.

– Noam Chomsky

The Chomsky hierarchy ←

- The tools and techniques we have been talking about here are often used to develop a specialized language for a particular task.

  A potentially more interesting (and likely more difficult) task is to work with an existing or observed language (or language system) and try to develop a grammar or recognizer for the language. One typical example of this might be to work with a natural (human?) language such as English or Spanish.

  In a more general form, we would be looking at any system that can be represented by strings of symbols, and we would attempt to discover and represent patterns and structures within the collection of observed strings (i.e., within the language of the system).

- In the 1950's, linguists applied the approaches of formal language theory to human languages. Noam Chomsky in particular looked at the relative power of various formal language systems (i.e., the range of languages that could be specified by various formal systems). He is typically credited with establishing relationships among various formal language systems – now known as the Chomsky Hierarchy.

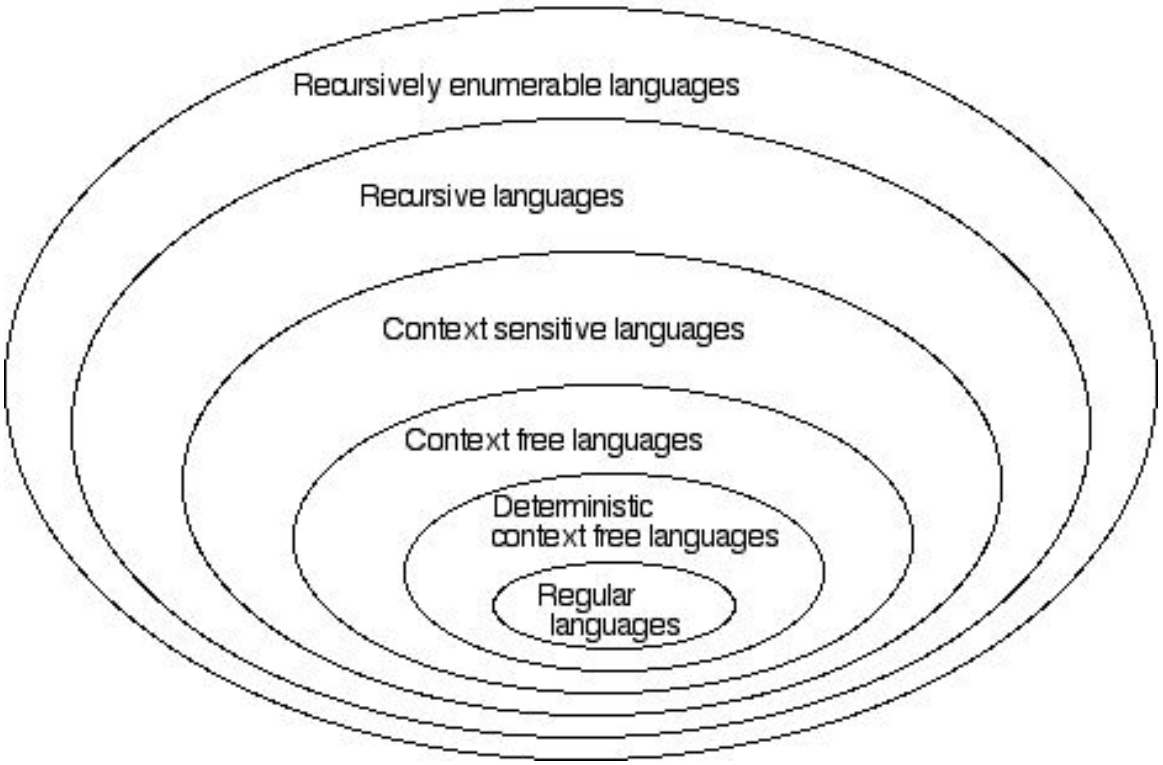
  Chomsky and other linguists had observed that very young children acquire language fluency much more rapidly and easily than one would expect from typical "learning algorithms," and therefore he hypothesized that the brain must embody some sort of "general language machine" which would, during early language exposure, be particularized to the specific language the child acquired. He therefore studied possible "general language machines." His hope was that by

characterizing various such machine types, linguists would be able to develop abstract formal grammars for human languages, and thus understand much more about languages in general and in particular.

In many respects, this project is still in process. On the next page is a slightly generalized outline of the classical Chomsky Hierarchy of formal languages. Human languages don't fit neatly into any one of the categories – at the very least, human languages are context sensitive, but with additional features. Also, though, since Chomsky's work in the 1950s, many more language categories have been identified (probably in the hundreds by now).

An interesting (and relatively deep) question is whether it fully makes sense to expect there to be a concise formal grammar for a particular human language such as English – and what exactly that would mean . . .

Elements of the Chomsky Hierarchy



Languages	Machines
Regular	DFA or NFA
Deterministic context free	Deterministic push-down automata
Context free	Nondeterministic PDA
Context sensitive	Linear bounded automata (Turing, bounded tape)
Recursive	Turing machines that halt on every input
Recursively enumerable	General Turing machines

- For purposes of research, it is worthwhile to be aware of the wide variety of formal language types that have been studied and written about. Often, though, it makes sense to restrict one's attention to a particular language class, and see which portions of the problem at hand are amenable to that class.

  For example, the regular languages are very concise, easy to work with, and many relevant tools and techniques have been developed. In particular, there has been much research on Markov models. It thus often makes sense to try to develop a regular or Markov approach to your problem, such as the work by Jim Crutchfield on ϵ-machines . . .

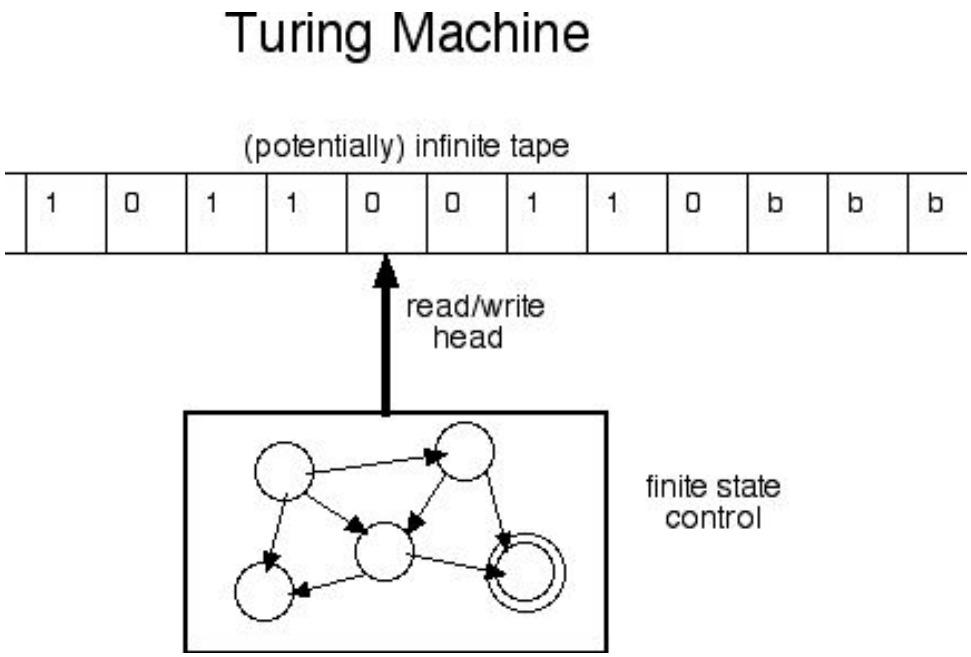
  Another important example (as indicated above) is work that has been done appliying hidden Markov models to the analysis of genome sequence problems . . .

Turing machines ←

- During the 1930s, the mathematician Alan Turing worked on the general problem of characterizing computable mathematical functions. In particular, he started by constructing a precise definition of "computability." The essence of his definition is the specification of a general computing machine – what is today called the Turing machine.

  Once having specified a particular definition of "computability," Turing was able to prove that there are mathematical functions that are not computable. Of course, one cannot "prove" that a definition itself is "correct," but can only observe that it is useful for some particular purposes, and maybe that it seems to do a good job of capturing essential features of some pre-existing general notions or intuitions. See, though, the Church-Turing thesis . . .

- The basic elements of the Turing machine are a (potentially) infinite "tape" on which symbols can be written (and re-written), and a finite state control with a read/write head for accessing and updating the tape. Turing used as his intuitive model a mathematician working on a problem. The mathematician (finite state control) would read a page (the symbol in one cell on the tape), think a bit (change state), possibly rewrite the page (change the symbol in the cell), and then turn either back or forward a page.



- More formally, a Turing machine $T = (\mathsf{S}, A, \delta, s_0, b, F)$ consists of:

  $\mathsf{S} =$ a finite set of states,
  $A =$ an alphabet,
  $\delta : \mathsf{S}\times(\mathsf{A} \cup \{\mathsf{b}\}) \rightarrow \mathsf{S}\times(\mathsf{A} \cup \{\mathsf{b}\})\times\{\mathsf{L}, \mathsf{R}\}$,
  the transition function,
  $s_0 \in \mathsf{S}$, the start state,
  b, marking unused tape cells, and
  $F \subset \mathsf{S}$, halting and/or accepting states.

  To run the machine, it is first set up with a finite input string (from A^*) written on the tape (and the rest of the tape initialized to the blank symbol 'b'). The machine is started in the start state (s_0) with its read/write head pointing at the first symbol in the input string. The machine then follows the instructions of the δ function – depending on its current state and the symbol being read in the current tape cell, the machine changes state, writes a (possibly) new symbol in

the current tape cell, and then moves the read/write head one cell either left or right. The machine continues processing until it enters a halting state (in F). Note that it is possible that the machine will go into an infinite loop and never get to a halting state . . .

There are several generic ways to think about what computation is being done by the machine:

1. Language recognizer: an input string from A^* is in the language L(T) of the machine if the machine enters a halting (accepting) state. The string is not in the language otherwise. Languages of such machines are called *recursively enumerable* languages.

2. Language recognizer (special): a subset of the halting states are declared to be 'accepting' states (and

the rest of the halting state are 'rejecting' states). If the machine halts in an accepting state, the string is in the language L(T). If the machine halts in a rejecting state, or never halts, the string is not in the language. In this special case, there are some particular machines that enter some halting state for every input string in A^*. Languages of these particular machines which halt on every input are called *recursive* languages.

3. String processor: given an input string, the result of the computation is the contents of the tape when the machine enters a halting state. Note that it may be that the machine gives no result for some input strings (i.e., never enters a halting state). For these machines, the partial function computed by the machine is a *partial recursive* function.

4. String processor (special): If the machine halts on every input string, the function computed by the machine is a *recursive* function.

- Turing proved a number of important facts about such machines. One of his deepest insights was that it is possible to encode a Turing machine as a finite string of symbols. Consider a machine $T = (\mathsf{S}, A, \delta, s_0, b, F)$. Each of $\mathsf{S}, A,$ and F is a finite set, and thus can be encoded as a single number (the number of elements in the set). We can assume that s_0 is the first of the states, so we don't need to encode that. We can assume (possibly by reordering) that the states F are at the end of the list of states, so we only need to know how many there are. The symbols in A are arbitrary, so we only need to know how many there are. The

blank symbol b can be encoded as though it were an extra last symbol in A. The only slightly tricky part is the transition function δ, but we can think of δ as being a (finite) set of ordered 5-tuples (s_i, a_i, s_j, a_j, d) where d (direction) is either L or R. There will be $|S| * (|A| + 1)$ such 5-tuples. Thus, an example Turing machine might be encoded by
$(3, 2, 1,$
$(0, 0, 0, 1, R), (0, 1, 2, 1, L), (0, 2, 2, 2, R),$
$(1, 0, 0, 0, R), (1, 1, 1, 0, R), (1, 2, 2, 2, L),$
$(2, 0, 0, 1, L), (2, 1, 2, 0, R), (2, 2, 1, 1, R))$

where there are three states, two alphabet symbols, and one halting state (state 2, but not states 0 or 1). We have encoded b as '2'.

Exercise: Draw the finite state control for this machine. (Can you figure out what the language of this machine is?)

- Turing's next insight was that since we can encode a machine as a finite string of symbols, we could use that string of symbols as the input to some other Turing machine. In fact, if we are careful, we can construct a Universal Turing Machine – a Turing machine that can simulate any other Turing machine! Turing gave the specifications of such a machine. We can call the universal Turing machine UTM.

  In particular, if we have a specific Turing machine T and a specific input string σ for the machine T, we work out the encoding for the machine T in the alphabet symbols of the UTM, and the encoding of the string σ in the alphabet symbols of the UTM, concatenate the two strings together, and use that as the input string to the UTM. The UTM then shuttles back and forth between the encoding of the string σ and the 'instructions' in the encoding of T, simulating the computation of T on σ.

- Turing proved a variety of important facts about Turing machines. For example, he proved that there cannot be a does-it-halt machine – that is, the question of whether or not a given Turing machine halts on a given input is noncomputable. No machine can be guaranteed to halt giving us a yes/no answer to this question for all machines and strings. In effect, the only sure way to tell if a given Turing machine halts on a given string is to run it – of course, the problem is that if it doesn't halt (goes into an infinite loop), you can't know for sure it's in an infinite loop until you wait forever!

  In fact, it turns out that almost all interesting questions about Turing machines (in fact, essentially any non-trivial question) is noncomputable (in the language of art, these are called *undecidable* questions). In effect, in trying to answer any non-trivial question, you

might fall into an infinite loop, and there is no sure way to know when you are in an infinite loop!

- Some other important theoretical work being done almost the same time as Turing's work was Kurt Gödel's work on the incompleteness of logical systems. Gödel showed that in any (sufficiently general) formal logical system, there are true statements that are not provable within the system.

  If we put Turing's results together with Gödel's, we can say in effect that Truth is a noncomputable function.
  Mathematically, Turing's and Gödel's fundamental results are equivalent to each other. You can't be guaranteed to know what's true, and sometimes the best you can do is run the simulation and see what happens . . .

Computability and tractability ←

- We have already observed that there are some problems that are not computable – in particular, we showed the existence of languages for which there cannot be an algorithmic recognizer to determine which strings are in the language. Another important example of a noncomputable problem is the so-called halting problem. In simple terms, the question is, given a computer program, does the program contain an infinite loop? There cannot be an algorithm that is guaranteed to correctly answer this question for all programs.

  More practically, however, we often are interested in whether a program can be executed in a 'reasonable' length of time, using a reasonable amount of resources such as system memory.

- We can generally categorize computational algorithms according to how the resources needed for execution of the algorithm increase as we increase the size of the input. Typical resources are time and (storage) space. In different contexts, we may be interested in worst-case or average-case performance of the algorithm. For theoretical purposes, we will typically be interested in large input sets . . .

- A standard mechanism for comparing the growth of functions with domain $\mathbb{N}$ is "big-Oh." One way of defining this notion is to associate each function with a set of functions. We can then compare algorithms by looking at their "big-Oh" categories.

- Given a function f, we define $O(f)$ by:

$$g \in O(f) \iff$$

there exist $c > 0$ and $N \geq 0$ such that $|g(n)| \leq c|f(n)|$ for all $n \geq N$.

- We further define $\theta(f)$ by:
  $g \in \theta(f)$ iff $g \in O(f)$ and $f \in O(g)$.

- In general we will consider the run-time of algorithms in terms of the growth of the number of elementary computer operations as a function of the number of bits in the (encoded) input. Some important categories – an algorithm's run-time f is:

  1. Logarithmic if $f \in \theta(\log(n))$.

  2. Linear if $f \in \theta(n)$.

  3. Quadratic if $f \in \theta(n^2)$.

  4. Polynomial if $f \in \theta(P(n))$ for some polynomial $P(n)$.

  5. Exponential if $f \in \theta(b^n)$ for some constant $b > 1$.

  6. Factorial if $f \in \theta(n!)$.

- Typically we say that a problem is *tractable* if (we know) there exists an algorithm whose run-time is (at worst) polynomial that solves the problem. Otherwise, we call the problem *intractable*.

- There are many problems which have the interesting property that if someone (an oracle?) provides you with a solution to the problem, you can tell in polynomial time whether what they provided you actually is a solution. Problems with this property are called Non-deterministically Polynomial, or NP, problems. One way to think about this property is to imagine that we have arbitrarily many machines available. We let each machine work on one possible solution, and whichever machine finds the (a) solution lets us know.

- There are some even more interesting NP problems which are universal for the class of NP problems. These are called NP-complete problems. A problem S is NP-complete if S is NP and, there exists a polynomial time algorithm that allows us to translate any NP problem into an instance of S. If we could find a polynomial time algorithm to solve a single NP-complete problem, we would then have a polynomial time solution for each NP problem.

- Some examples:

  1. Factoring a number is NP. First, we recognize that if M is the number we want to factor, then the input size m is approximately $\log(M)$ (that is, the input size is the number of digits in the number). The elementary school algorithm (try dividing by each number less than $\sqrt{M}$) has run-time approximately $10^{\frac{m}{2}}$, which is exponential in the number of digits. On the other hand, if someone hands you two numbers they claim are factors of M, you can check by multiplying, which takes on the order of m^2 operations.

     It is worth noting that there is a polynomial time algorithm to determine whether or not a number is prime, but for composite numbers, this algorithm does not provide a

factorization. Factoring is a particularly important example because various encryption algorithms such as RSA (used in the PGP software, for example) depend for their security on the difficulty of factoring numbers with several hundred digits.

2. Satisfiability of a boolean expression is NP-complete. Suppose we have n boolean variables $\{b_1, b_2, \ldots, b_n\}$ (each with the possible values 0 and 1). We can form a general boolean expression from these variables and their negations:

$$f(b_1, b_2, \ldots, b_n) = \bigwedge_k (\bigvee_{i,j \leq n} (b_i, \sim b_j)).$$

A solution to such a problem is an assignment of values 0 or 1 to each of the b_i such that $f(b_1, b_2, \ldots, b_n) = 1$. There are 2^n possible assignments of values. We can check an individual possible solution in polynomial time, but there are exponentially many possibilities to check. If we could develop a feasible computation for this problem, we would have resolved the traditional P$\stackrel{?}{=}$NP problem ...

Computational complexity

- Suppose we have some system we are observing for a fixed length of time. We can imagine that we have set up a collection of instruments (perhaps just our eyes and ears), and are encoding the output of our instruments as a finite string of bits (zeros and ones). Let's call the string of bits β. Can we somehow determine (or meaningfully assign a numerical value of) the complexity of the system, just by looking at the string β?

- One approach is to calculate the information entropy of β. This approach, unfortunately, carries with it the dilemma that any entropy calculation will have to be relative to a probability model for the system, and there will be many possible

probability models we could use. For example, if we use as our model an unbiased coin, then β is a possible outcome from that model (i.e., *a priori* we cannot exclude the random model), and the information entropy of β is just the number of bits in β. (What this really says is that the upper bound on the information entropy of β is the number of bits . . .) On the other hand, there is a probability model where β has probability one, and all other strings have probability zero. Under this model, the entropy is zero. Again, not particularly useful. (What this really says is that the lower bound on the entropy is zero . . .)

In order to use this approach effectively, we will have to run the system several times, and use the results together to build a probability model. What we will be looking for is the 'best' probability model for the system – in effect, the limit as we observe the system infinitely many times.

- Another approach we might take is a Universal Turing Machine approach. What we do is this. We pick a particular implementation of the UTM. We now ask, "What is the minimum length (T, σ) Turing machine / input string pair which will leave as output on the UTM tape the string (T, β)?" This minimal length is called the *computational complexity* of the string β. There is, of course, some dependence on the details of the particular UTM chosen, but we can assume that we just use some standard UTM.

  Clearly a lower bound is zero (the same as the lower bound for entropy). As an upper bound, consider the machine T_0 which just halts immediately, no matter what input it is given. Suppose we give as input to our UTM the string pair (T_0, β). This pair has the required property (it leaves (T_0, β) as output on the UTM),

and its length is essentially just the length of β (this problem is only interesting if β is reasonably long, and in fact the coding for T_0 is very short – exercise: write a typical encoding for the machine T_0). Note that the upper bound also agrees with the entropy.

In theory, the computational complexity of any string β is well-defined (and exists?). There is a nonempty set of Natural numbers which are the lengths of (T, σ) pairs which have our required property (the set is nonempty because (T_0, β) works), and therefore there is a (unique) smallest element of the set. QED. (At some level, this satisfies the mathematician in me :-)

In practice? Ah, in practice. Remember that by Turing's result on the halting problem, there is no guaranteed way to tell if a given Turing machine will even

halt on a given input string, and hence, if you give me an arbitrary (T, σ), I may just have to run it, and wait forever to know that it doesn't leave (T, β) as output. What that means is that the computational complexity is a noncomputable function! However, we may be able to reduce the upper bound, by judicious choice of machines to test, putting reasonable limits on how long we let them run before we throw them out, and pure luck. (At some level, this satisfies the computer scientist in me :-)

The two parts, theory and practice, give the philosopher in me plenty to chew over :-)

- In the case of the entropy measure, we either have to observe the system for an infinitely long time to be sure we have the right probability model, or accept an

incomplete result based on partial observations. If we can only run the system for a limited time, we may be able to improve our entropy estimates by judicious subsampling of β.

In the case of the computational complexity measure, we either have to run some machines for an infinitely long time in order to exclude them from consideration, or accept an incomplete result based on output from some but not all machines.

We are in principle prevented from knowing some things! On the other hand, it has been shown that although in practice we can't be guaranteed to get the right answer to either the entropy or computational complexity values, we can be sure that they are (essentially) equal to each other, so both methods can be useful, depending on what we know about the system, and what our local goals are.

To top ←

References

[1] Bennett, C. H. and Landauer, R., The fundamental physical limits of computation, *Scientific American,* July 38–46, 1985.

[2] Bennett, C. H., Demons, engines and the second law, *Scientific American* **257** no. 5 (November) pp 88–96, 1987.

[3] Campbell, Jeremy, *Grammatical Man*, Information, Entropy, Language, and Life, Simon and Schuster, New York, 1982.

[4] Chaitin, G., *Algorithmic Information Theory*, Cambridge University Press, Cambridge, UK, 1990.

[5] Chomsky, Noam, Three models for a description of a language, *IRE Trans. on Information Theory*, 2:3, 113-124, 1956.

[6] Chomsky, Noam, On certain formal properties of grammars, *Information and Control*, 2:2, 137-167, 1959.

[7] Church, Alonzo, An unsolvable problem of elementary number theory, *Amer. J. Math.* **58** 345–363, 1936.

[8] DeLillo, Don, *White Noise*, Viking/Penguin, New York, 1984.

[9] Durbin, R., Eddy, S., Krogh, A., Mitchison, G. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids* Cambridge University Press, Cambridge, 1999.

[10] Eilenberg, S., and Elgot, C. C., *Recursiveness*, Academic Press, New York, 1970.

[11] Feller, W., *An Introduction to Probability Theory and Its Applications*, Wiley, New York, 1957.

[12] Feynman, Richard, *Feynman lectures on computation*, Addison-Wesley, Reading, 1996.

[13] Garey M R and Johnson D S, *Computers and Intractability*, Freeman and Company, New York, 1979.

[14] Gödel, K., Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I, *Monatshefte für Math. und Physik*, 38, 173-198, 1931.

[15] Hodges, A., *Alan Turing: the enigma*, Vintage, London, 1983.

[16] Hofstadter, Douglas R., *Metamagical Themas: Questing for the Essence of Mind and Pattern*, Basic Books, New York, 1985.

[17] Hopcroft, John E. and Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Massachusetts, 1979.

[18] Knuth, D. E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed, Addison-Wesley, Reading, 1981.

[19] Linz, Peter, *An Introduction to Formal Languages and Automata*, 3rd Ed., Jones and Bartlett, Boston, 2000.

[20] Lipton, R. J., Using DNA to solve NP-complete problems, *Science*, **268** 542–545, Apr. 28, 1995.

[21] Minsky, M. L., *Computation: Finite and Infinite Machines* Prentice-Hall, Inc., Englewood Cliffs, N. J. (also London 1972), 1967.

[22] Moret, Bernard M., *The Theory of Computation* Addison-Wesley, Reading, Massachusetts, 1998.

[23] von Neumann, John, Probabilistic logic and the synthesis of reliable organisms from unreliable components, in *automata studies(Shannon, McCarthy eds)*, 1956 .

[24] Papadimitriou, C. H., *Computational Complexity*, Addison-Wesley, Reading, 1994.

[25] Rabin, M. O., Probabilistic Algorithms, *Algorithms and Complexity: New Directions and Recent Results*, pp. 21-39, Academic Press, 1976.

[26] Schroeder, Manfred, *Fractals, Chaos, Power Laws, Minutes from an Infinite Paradise*, W. H. Freeman, New York, 1991.

[27] Schroeder, M. R., 1984 *Number theory in science and communication* Springer-Verlag, New York/Berlin/Heidelberg, 1984.

[28] Turing, A. M., On computable numbers, with an application to the Entscheidungsproblem, *Proc. Lond. Math. Soc. Ser. 2* **42**, 230 ; see also *Proc. Lond. Math. Soc. Ser. 2* **43**, 544, 1936.

[29] Vergis, A., Steiglitz, K., and Dickinson, B., The Complexity of Analog Computation, *Math. Comput. Simulation 28*, pp. 91-113. 1986.

[30] Zurek, W. H., Thermodynamic cost of computation, algorithmic complexity and the information metric, *Nature* **341** 119-124, 1989.

To top ←