COMPUTING SCIENCE AND TECHNOLOGY vs. THEORY OF COMPUTATION

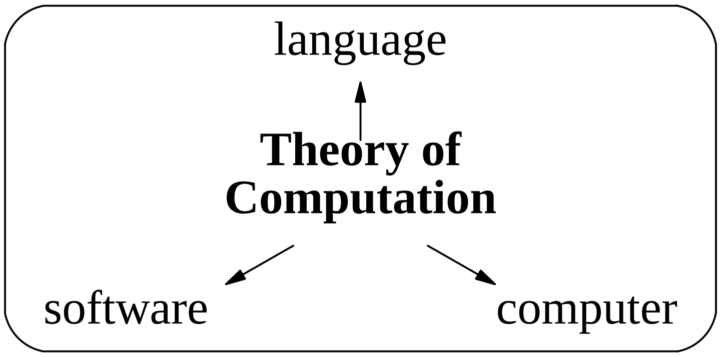
Computing Science:

Finds orders, structures, and patterns in useful (effective) computations.

Example. Each program can be built without goto's using only if-then-else, while-do loop, and sequence of statements.

Computing Technology:

Creates computers, languages, and algorithms/software with desired properties/behaviors using the computing science.



Computing Hypothesis:

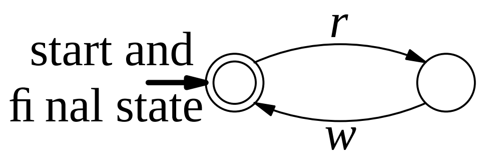
All behaviors/properties of living and non-living things can be modeled/analyzed/simulated via computation because there are "patterns" (logic) in those behaviors.

$$\boxed{\text{A program represents a potentially infinite set of computations with certain specific structures or patterns in them.}}$$

EXAMPLE OF A FINITE-STATE MODEL FOR A SIMPLE PROGRAM

```
while (not end-of-input-file) do {
      read a character;
      write that character to output-file;
}
```

Consider just the operations: read(*r*) and write(*w*):



- *r* and *w* occurs alternately, starting with *r* and ending with *w*.
- Possible sequences of computations for different input-files:

rw	Input file with one character
rwrw	Input file with two characters
rwrwrw	Input file with three characters

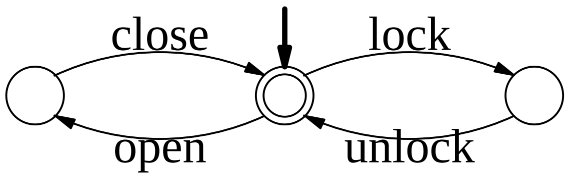
Pattern of computations: $rwrw \cdots rw = (rw)^n$ for $n \geq 0$.

Question:

•? Show the finite-state model and the pattern of computations in the above program when we also consider the operation *t* = while-test.

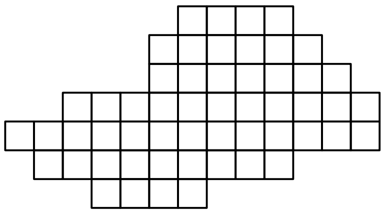
•? Give a program for printing the characters in a text-file in which the read/write operations form the pattern $r^n w^n$, $n \geq 0$.

•? What does the FSM below for the operations {open, close, lock, unlock} on a door say about which operations can be done when?

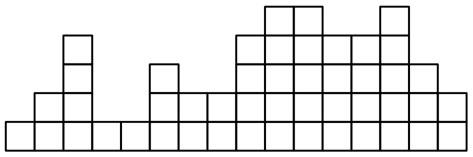


EXAMPLE QUESTIONS THAT WE WILL BE ABLE TO ANSWER

1. Consider a robot that only moves east, west, north, and south (but no rotation). Can we compute the robot's moves for traversing an arbitrary *convex* maze like the one below, using a fixed amount of memory independent of the maze's size? (Convex means the squares along each horizontal and vertical line form a continuous strip without any gap; see the maze in Problem 2.) Show two different traversals of the maze below by numbering the squares 1, 2, ⋯ in the order they wold be visited; start at the top right corner. Also, state in English your strategy in each case. Does your strategy work for a non-convex maze (perhaps having holes)?



2. Can we compute a northmost point in a skyline-shaped maze of arbitrary size using only a fixed amount of memory?



3. How much memory (and arithmetic power) do we need to determine the divisibility of binary numbers by 3?

101 = 5, not div. by 3	10101 = 21, div by 3
	11110 = 30, div by 3

Question: If 3 divides a binary number then does 3 also divide the binary number for the reverse string? (Can you prove it?)

KEY ISSUES IN THEORY OF COMPUTATION

Key issues:

- What kinds of things are computable (or non-computable)?
- What is the role of memory in computation?
- How the notion of control differs from memory?
- Can we classify computable things in some way? Are some things more non-computable than others in some way?

Inputs
Algorithm
Outputs

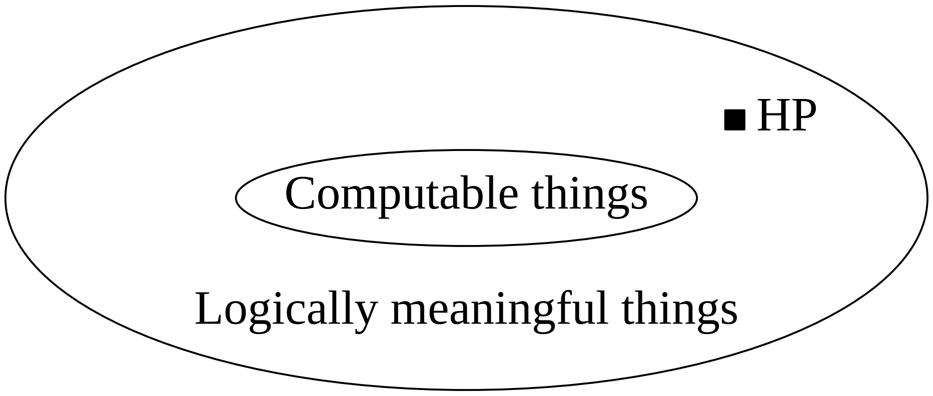
Algorithm:

- A *finite* description of a potentially *infinite* number of different computations (resulting from different inputs).
- The common pattern in those computations makes it possible to describe those computations in the form of a *finite* algorithm.

Understanding an algorithm means: understanding

- *What* is computed, i.e., the input-output relationship
- *How* the input is transformed to the output, in particular, the *pattern* in those computations

HALTING PROBLEM IS NOT COMPUTABLE



Only a small part of all logically/mathematically meaningful things are computable.

Question: Are all computable things logically meaningful?

Halting Problem (*P*, *I*): Does program *P* stop for the input *I*?

- It is a logically meaningful question, but there is no algorithm to solve the problem for all (*P*, *I*) pairs!

Question:

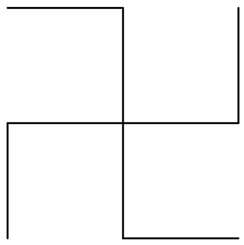
•? Why can't we just execute *P* for the input *I*?

•? Give details to show the inputs *x* for which the following program will (will not) halt.

```
void DoesItHalt(double x)
{ double y, sum;
  for (sum = y = 1.0; sum >= 1.0; sum += y)
      y = y*x;
  printf("x=%5.3f, sum=%5.3f\n", x, sum);
}
```

PATTERN

Is this a pattern – pattern of what?



Is there a pattern of the digits in the decimal form of 1/7?

$$1/7 = 0.142857142857142857\cdots$$

Is there a pattern among the strings below?

$$aa,\ aba,\ abba,\ abbba,\ \cdots$$

Is there a pattern of digits in the following?

123223131213

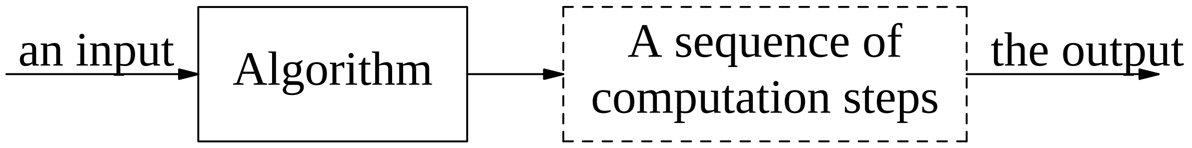
We will talk of patterns of *symbols* in a *set* of strings.

Question: Why strings when we talk of patterns?

Every thing we think or write can be expressed as a string.

Question: Can you give an example of a string without a pattern?

PATTERN IN COMPUTATION



Computation: The result of activation of an algorithm by an input.

- A *finite* sequence $x = a_1 a_2 \cdots a_n$ (or an *infinite* sequence $x = a_1 a_2 \cdots$) of *atomic* operations or steps.†
- Each $a_i \in \Sigma = \{b_1, b_2, \cdots, b_k\}$ and b_j's are the distinct atomic operations in the computations of an algorithm/program.

Example. Consider the read/write operations (i.e., $\Sigma = \{r, w\}$) in computations of the program:

```
while (not end-of-input-file) do {
      read a character;
      write that character to output-file;
}
```

- Each computation is a finite string of the operations $\Sigma = \{r, w\}$.
- It has the pattern (form): $x = rwrw \cdots rw = (rw)^n$, $n \geq 0$.

$$\boxed{\text{The computations of each algorithm have a pattern.}}$$

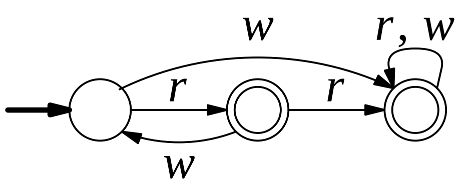
Computation Related to (Understanding) a Pattern:

- How to test if a given string fits the pattern.

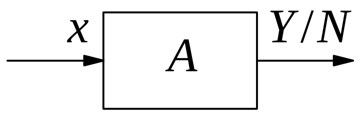
† Henceforth, a finite computation sequence x will be referred to as a *string*. The term *sequence* or *infinite string* will be used for an infinite computation.

EXERCISE

1. Let $L = \{(rw)^n\colon n \geq 0\}$. Describe the strings which are not in L, i.e., the strings in $L^c = \{x\colon x \notin L\}$. Is there a pattern in those strings? (Hint: write some example strings in L^c and then find a compact description for L^c.) If you know something about regular expressions from your Discrete Math class, then write a regular-expression for L and another regular expression for L^c. Does the following FSM describe all strings in L^c?

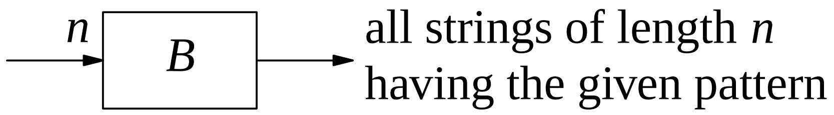


2. Suppose you have an algorithm A for testing whether a string x has a desired pattern or not. How will you compute all strings of a given length n which have that pattern? Give a brief description of your algorithm; clearly identify the inputs and the outputs.



   You can assume that the alphabet is fixed for the moment for both A and B, and are not being input.

3. Suppose now that you have an algorithm B for computing all strings of a given length n which have a certain pattern. How will you test if a string x has that pattern? Give a brief description of your algorithm.



OUR GOALS

We study:

- The basic notions of *finite-state* control and *memory*, and their role in computation.
- Abstract models (machines) for computation.

Why study these models:

- Computer hardware design (both architecture and technology) have changed frequently since the first day of computers, The changes are, however, mostly of "engineering" nature.
- The basic Turing Machine (TM) model for computation has remained unchanged since its discovery, and it is unlikely that we will ever need a change this model.

Church-Turing Hypothesis:

A computation method is an algorithm if and only if it can be modeled by a TM.

$$\boxed{\text{An algorithm = A Turing machine.}}$$

Goals:

- Learn to think in terms of "patterns".
- Classify patterns based on the characteristics of their recognition algorithms.
- Give *computable descriptions* of patterns that lead to algorithms form their recognition.
- Form complex machines/algorithms from simpler ones for recognition of complex patterns.

ABSTRACT MODELS OF COMPUTATION

- Form the basis of all hardware and software design, covering all aspects of computation:
  - Design of computers
  - Design of application and systems programs executed by computers
  - Design of compilers to process the programs to generate executable code
  - Design of programming languages to express computations/algorithms

$$\boxed{\text{A simple looking pattern can be computationally very complex.}}$$

‡ The computation of an *algorithm* may not terminate for all inputs; such a computation is sometimes called *procedural* to distinguish it from computations that always terminate. We are concerned here with computations that terminate.

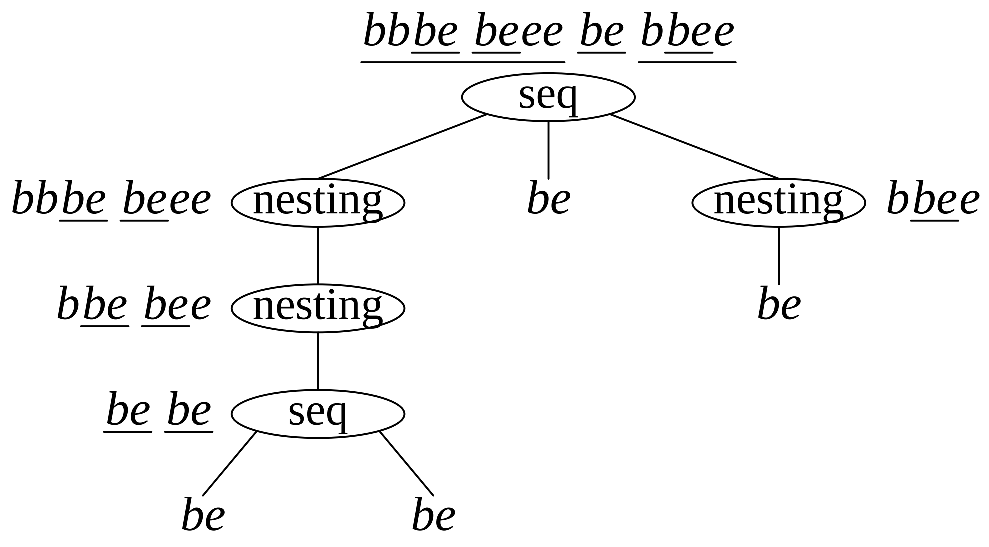
PATTERNS EVERYWHERE

Pattern of `begin-end` in a program:

```
begin        b b e e b e            rlogin
  begin      ( ( ) ) ( )              rlogin
  end                                 logout
end                                 logout
begin                               rlogin
end                                 logout
```

General Form:

- Simplest case: $b^n e^n$, $n \geq 1$ (nesting); $b^3 e^3 = \underline{b\underline{b\underline{be}e}e}$.
- More general case: $\underline{b^{n_1}e^{n_1}}\ \underline{b^{n_2}e^{n_2}}\cdots\underline{b^{n_k}e^{n_k}}$, $n_j \geq 1$, $k \geq 1$ (sequencing of nested parts).
- Most general case: $\underline{bb\underline{be}\ \underline{be}ee}\ \underline{be}\ \underline{b\underline{be}e}$, arbitrary combination of *nesting and sequencing* operations.



EXERCISE

1. Let s represent any program-statement other than the begin/end statements and variable declarations. The simplest pattern of statements in a program, when we ignore variable declarations, is $bs^n e$, $n \geq 0$. Is the following a valid pattern of statements in a program:

   $$bsbsseebse$$

   How can this be obtained from a valid b/e-pattern? State a general rule for creating valid pattern of statements in a program from valid b/e-patterns that would cover the above example.

2. If the answer to Problem 1 does not give the most general pattern of statements in a program, then find a rule for such patterns starting from valid b/e-patterns.

EXERCISE

1. Show that the following definitions of a *balanced* b/e-string x are equivalent. (Hint: First, to get a good sense of the matchings in (ii), write all possible matching for x = *bbebee* and x = *bbebeebe*. Then, show that (ii′) implies that the scheme "match the kth b from the left to the kth e from the left" gives a matching as in (ii). Can we use a similar scheme from the right instead of from the left?)

   Def. 1. (i) $\#(b, x) = \#(e, x)$ and
   (ii) There is a matching (one-to-one and onto mapping) of the b's and the e's in x such that each b is matched with an e to its right.

   Def. 2. (i′) $\#(b, x) = \#(e, x)$ and
   (ii′) For each initial part x' of x, $\#(b, x') \geq \#(e, x')$.

2. Let $N(x)$ = #(matchings for x as in Def. 1), where x is a balanced b/e-string. Show the following:

   (i) If x_1 and x_2 are balanced strings and $x = x_1x_2$, then $N(x) = N(x_1)N(x_2)$. Verify your answer by showing all possible matchings for x_1 = *bbebee*, x_2 = *bbee*, and x. (What property do you observe among the matchings for x and those for x_1 and x_2?)

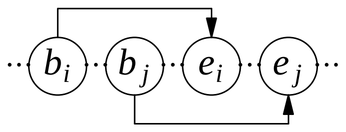
   (ii) If x cannot be factored into two smaller balanced strings as in (i), then the last e can be matched with any b. (Similarly, the first b can be matched with any of the e's.)

   (iii) Let N_j = #(b's to the left of jth e from left). Show that $N(x) = N_1(N_2 - 1)\cdots(N_n - n + 1)$, where $|x| = 2n$. Thus, $N(x) \leq n!$. Which x's give $N(x) = n!$ and $N(x) = 1$?

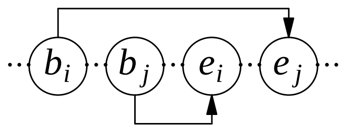
3. Find an algorithm to construct all balanced strings of length $2n$. (Do not construct all possible strings of length $2n$ over $\Sigma = \{b, e\}$ and then throw away the unbalanced ones.)

4. Find an algorithm to construct all string x of length $2n$ over $\Sigma = \{b, e\}$ such that $\#(b, x) = \#(e, x)$.

5. Consider a balanced b/e string and a matching as in Def. 1 in Problem 1. For the situation shown below on the left, we say the matched pairs $b_i: e_i$ and $b_j: e_j$ *cross* each other. We can define a new matching among $\{b_i, b_j, e_i, e_j\}$ to uncross these $b: e$ pairs as shown on the right and still satisfy the condition that "each b is matched with an e to its right". We say that $b_j: e_i$ is the *reduced pair* in the uncrossing operation and $b_i: e_j$ is the *enlarged pair*.



A crossing pair of matched b: e's.



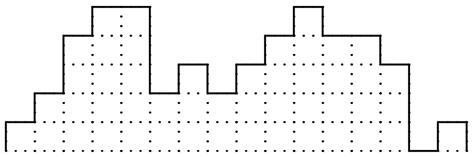
The result of uncrossing the b: e pairs.

(i) How can we apply the uncrossing in some systematic fashion and eliminate all crossings?

(ii) Argue now that given any matching satisfying Def. 1, we can create a matching with no crossing and still satisfying Def. 1.

PATTERNS EVERYWHERE (contd.)

Pattern in a skyline:



A city skyline: $\underline{ne}\ \ \underline{ne}n\underline{ne}\ \ \underline{nee}ss\underline{se}\ \ \underline{ne}\ \ \underline{se}\ \ \underline{ne}\ \ \underline{ne}\ \ \underline{ne}\ \ \underline{see}\ \ \underline{se}ss\underline{se}\ \ \underline{ne}s$

Question:

•? Is there a relationship between the skyline-pattern and any of the patterns seen previously?

•? Are there patterns in shapes of flowers and leaves? Are there patterns in paths followed by storms, cyclones, and rivers?

•? Are there patterns in human thoughts or in computations performed by programs?

$\boxed{\text{Different things may have closely related patterns when they are represented with proper abstractions.}}$

Question:

•? How do you describe the pattern of 0's and 1's in the binary strings corresponding to the even integers $n \geq 0$?

n = 0: 0	n = 4: 100	n = 8: 1000	···
n = 1: 1	n = 5: 101	n = 9: 1001	···
n = 2: 10	n = 6: 110	n = 10: 1010	···
n = 3: 11	n = 7: 111	n = 11: 1011	···

The string 0 *or* the strings starting with 1 and ending with 0.

REGULAR EXPRESSIONS: A POWERFUL METHOD FOR DESCRIBING PATTERNS

Regular expression description: 0 + 1(0 + 1)*0

+ means (logical) "or"
* means "repeat 0 or more times"

- Regular expressions provide us a way of giving a *finite* descriptions for a large variety of *infinite* sets of strings (which have a reasonably simple pattern in them).

Question:

•? What is a regular expression for all binary strings with even number of 0's?

$\{\lambda, 1, 00, 11, 001, 010, 100, 111, \cdots\}$

Reg. Exp: 1*(01*01*)*

$x = 11010001110101 = \underline{11}\cdot\underline{010}\cdot\underline{00111}\cdot\underline{0101}$

POWER OF ABSTRACTION: A METHOD OF PROBLEM REDUCTION

Divisibility by 3 of a binary number:

($*$) num(s) is divisible by 3 iff num(s') is divisible by 3, where s' is the reverse of a binary string s.

Question:

•? For what kind of s, the property ($*$) is easily proved?

•? How can we reduce an arbitrary s to this special form?

Three Reduction/Abstraction Rules ($s \neq 000\cdots0$):

(1) Remove starting and ending 0’s. (3) Remove "00".
(2) Replace "11" by "00".

Example. Rules used below: (1),(2),(3),(3),(2),(3),(1)

$101110010 \rightarrow 10111001 \rightarrow 10001001 \rightarrow 101001 \rightarrow 1011 \rightarrow 1000 \rightarrow 1.$

Question:

•? Are there other ways of applying the rules (1)-(3) to 101110010 and do we still get the same final string 1?

•? Can we replace rule (3) by a combination of rules (2)-(3)? Is this better than rule (3)?

•? Does the final string depend on how we apply the reduction rules?

•? What is the pattern of fully reduced binary strings?

What remains to show: Prove the following:

($**$) num(s) is divisible by 3 iff num(reduction(s)) is divisible by 3 on each application of rules (1)-(3).

Question: Give a rule which can also be used in reducing an s to the form (10)*1 but for which the property ($**$) does not hold.

PROOF OF (∗∗) AND (∗)

Proof of (∗∗):

- Rule (1):
  - If $s = x0$ (x is not the empty string), then num(s) = 2.num(x) and thus num(s) is divisible by 3 iff num(x) is divisible by 3.
  - If $s = 0x$, then num(s) = num(x), etc.
- Rule (2):
  - If $s = x11y$, then num(s) = num($x00y$) + $3.2^{|y|}$ and thus both or none of num(s) and num($x00y$) is divisible by 3.
- Rule (3):
  - If $s = x00y$, then

$$\begin{aligned}\text{num}(s) &= \text{num}(x).2^{|y|+2} + \text{num}(y)\\ &= 4.[\text{num}(x).2^{|y|} + \text{num}(y)] - 3.\text{num}(y)\\ &= 4.\text{num}(xy) - 3.\text{num}(y).\end{aligned}$$

  Thus 3 divides both or none of num(s) and num(xy).

Coro. If s' is the fully reduced form of s, then 3 divides both or none of num(s) and num(s').

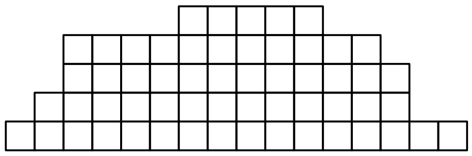
Proof of (∗):

- Follows easily now by induction on length of s and the fact that if s is fully reduced then s = reverse(s).

Question: Show the pattern of the binary numbers of the form (10)*1 that are divisible by 3.

EXERCISE

1. Show the encoding of the following convex-shaped skyline using the symbols $\{e, n, s\}$. In what way the patterns of the encoded strings for convex-shaped skyline is *simpler* than those for general skylines.



2. Shown below are 16 statements formed from the expressions E_1 = "the things he has" and E_2 = "the things he wants", by combining them in different ways with the quantifi ers "some" and "every" and the negation "not". The sentences (e)-(h) are simply the negations of (a)-(d); similarly for (e')-(h') and (a')-(d').

(a)	He has every thing that he wants.	(a′)	He wants every thing that he has.
(b)	He has some thing(s) that he wants.	(b′)	He wants some thing that he has.
(c)	He has every thing that he does not want.	(c′)	He wants every thing that he does not have.
(d)	He has some thing that he does not want.	(d′)	He wants some thing that he does not have.
(e)	He does not have every thing that he wants.	(e′)	He does not want every thing that he has.
(f)	He does not have some thing that he wants.	(f′)	He does not want some thing that he has.
(g)	He does not have every thing that he does not want.	(g′)	He does not want every thing that he does not have.
(h)	He does not have some thing that he does not want.	(h′)	He does not want some thing that he does not have.

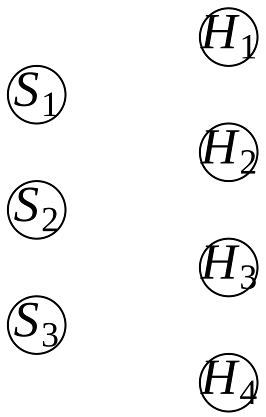
Express each of the statements (a)-(h') using H = the set of things he has, W = the set of things he wants, and Ω = the set of things under consideration (the universe). For example, (a) corresponds to $W \subseteq H$ and (b) corresponds to $H \cap W \neq \emptyset$. Avoid the use of set-complementation, as much as possible, in order to facilitate answering Problem 2 below.

3. Pair each statement in the group (a)-(h) in Problem 2 with an equivalent statement (with the same meaning) in the group (a')-(h'). No two statements within the same group are equiv-alent.

4. Which of the statements (a)-(h) in Problem 2 can be combined to express that $H = W$? Show such a combination.

5. Which of the following is a stronger statement (and hence is less likely to be true in a given situation)?
   (a) Some student got full-marks for each homework in CSC-4890.
   (b) For each homework in CSC-4890, some student got full-marks for it.

6. Consider a class of CSC-4890 with 3 students $\{S_1, S_2, S_3\}$ and 4 homeworks $\{H_1, H_2, H_3, H_4\}$. Draw lines between them to indicate who received full-marks for which homework(s) in such a way that only one of the above statements (i.e., the weaker one) holds.



COUNTABLE AND UNCOUNTABLE SET

Finite:	The set is empty, or its elements can be listed as e_1, e_2, $\cdots$, e_n for some $n \geq 1$.
Countable:	The set is infinite, but all its elements can be listed in some order as first, second, etc: e_1, e_2, e_3, $\cdots$.
	To show that a set S is countable, you must find a way of listing all items of S *exactly* once.
Uncountable:	Infinite, but not countable; cannot be list its items as first, second, etc.

Example 1. If $\Sigma = \{a\}$, then Σ^* = the set of all strings over Σ = $\{\lambda, a, aa, aaa, \cdots\}$ is countable.

Example 2. If $\Sigma = \{a, b\}$, then Σ^* is countable:

$$\underbrace{\begin{matrix}\lambda, \\ 1\end{matrix}}_{\text{length=0}} \qquad \underbrace{\begin{matrix}a, & b, \\ 2 & 3\end{matrix}}_{\text{length=1}} \qquad \underbrace{\begin{matrix}aa, & ab, & ba, & bb, \\ 4 & 5 & 6 & 7\end{matrix}}_{\text{length=2}} \qquad \cdots$$

Question:

•? Consider the set of all possible keywords, other special symbols (’;’, ’(’, ’{’, etc.), and the names of identifiers in a programming language. Is this set finite, countable, or uncountable?

•? If S is countable, show that $S^2 = \{(s_1, s_2)$: each $s_i \in S\}$ is countable by giving a systematic listing of all elements of S^2. (A similar argument shows that S^n is countable for $n \geq 2$.)

•? Show that $S_1 \cup S_2$ is countable if S_1 and S_2 are countable.

•? Show that the number of programs in a programming language is countable.

AN UNCOUNTABLE SET

The set I of all infinite sequences over $\Sigma = \{a, b\}$ is uncountable.

- I is clearly an infinite set because we have the distinct sequences: $x_n = a^n bbb\cdots$, $n \geq 1$.
- If possible, suppose $y_1, y_2, \cdots$ is a listing of all sequences in I, and let

$$\begin{array}{ll} y_1 = a_{11}a_{12}\cdots & \\ y_2 = a_{21}a_{22}\cdots & \quad \text{(each } a_{ij} = a \text{ or } b\text{)} \\ y_3 = a_{31}a_{32}\cdots & \\ \cdots & \end{array}$$

The sequence $z = \overline{a_{11}}\ \overline{a_{22}}\ \cdots$, where $\overline{a_{kk}} = \begin{cases} a, \text{ if } a_{kk} = b \\ b, \text{ if } a_{kk} = a \end{cases}$

differs from y_k in position k for each k. Thus, the listing $y_1, y_2, \cdots$ cannot be a complete listing of the sequences in I.

A similar argument shows that the infinite sequences over any alphabet of size ≥ 2 is uncountable.

Question:

•? If $y_k = a^k bbb\cdots$ for $k \geq 1$, then what is the sequence z constructed above? (Is $z \neq y_k$ for each y_k?)

•? How many infinite sequences are there over $\Sigma = \{a\}$?

•? Will $z = \overline{a_{11}}\ \overline{a_{21}}\ \overline{a_{31}}\ \cdots$ or $z = \overline{a_{12}}\ \overline{a_{21}}\ \overline{a_{34}}\ \overline{a_{43}}\ \cdots$ work in the above proof? Are there finitely or countably or uncountably many sequences that are not included in the listing $y_1, y_2, \cdots$ above?

EXERCISE

1. One way to show that two finite sets S_1 and S_2 have the same size is to find an one-one and onto mapping from S_1 to S_2. Use this method to show that the number of binary strings of length $2k - 1$ with even number of 0's is the same as the number of binary strings of length $2k - 1$ with odd number of 0's. (State in English what the mapping does to a string of length $2k - 1$ with even number of 0's, and illustrate the mapping for the strings of length $2k - 1 = 3$.)

   The same property also holds for strings of length $2k$, but the proof requires a different kind of mapping. Find such a mapping in this case also. (Hint: one possible approach is to reduce the problem to strings of length $2k - 1$ based on the leftmost bit being 0 or 1. There are other simple ways also.) Illustrate your mappings for $k = 2$.

   (An alternate method of showing this is to show that for all $m \geq 1$, one has $C_0^m + C_2^m + \cdots = C_1^m + C_3^m + \cdots$.)
2. Let $\Sigma = \{a, b, +, -\}$, and $L = \{x \in \Sigma^*$: the symbols '+' and '–' always appear paired-up as $+-$ or as $-+$, with no a or b in between the '+' and '–' of a pair$\}$. Let $f(n)$ = #(strings in L of length n). Show all strings of length 3. Prove or disprove that $f(n+2) = 2[f(n+1) + f(n)]$.
3. Consider the countable set $N = \{1, 2, 3, \cdots\}$, and let S be the set of all non-empty finite subsets of N. Find a one-to-one and onto mapping between the elements of S and the set of all binary strings ending in 1. Then, argue that S is countable. (Hint: First represent each subsets of a set of n elements by a binary string of length n. Some of these strings can be replaced by their initial parts without loosing any information.)

4. Suppose f: $S_1 \to S_2$ is an one-to-one and onto mapping between the sets S_1 and S_2. Is it true that either both S_1 and S_2 are fi nite or both are countable or both are uncountable?

5. If S_1 and S_2 are two disjoint set, then what can you say about $S_1 \cup S_2$ in each of the following cases?
   (i) Both S_1 and S_2 are fi nite.
   (ii) S_1 is countable and S_2 is fi nite.
   (iii) Both S_1 and S_2 are countable.
   (iv) S_1 is countable and S_2 is uncountable.
   (v) Both S_1 and S_2 are uncountable.

6. The following listing shows that $Q_{(0,\ 1)}$ = the set of all rational numbers in the interval (0, 1) is countable (here, m and n have no common factor in m/n):

   1/2, 1/3, 2/3, 1/4, 3/4, 1/5, 2/5, 3/5, 4/5, 1/6, 5/6, ···

   What is wrong in the following listing as an attempt to show that $Q_{(0,\ 1)}$ is countable?

   0.1, 0.2, 0.3, ···, 0.9,
   0.01, 0.02, 0.03, ···, 0.09,
   0.11, 0.12, 0.13, ···, 0.19,
   0.21, 0.22, 0.23, ···, 0.29,
   0.31, 0.32, 0.33, ···, 0.39,
   ...
   0.91, 0.92, 0.93, ···, 0.99,
   ...

   How can you modify it to show that $Q_{(0,\ 1)}$ is countable?

STRINGS AND LANGUAGES

(Finite) Alphabet:

- Σ = A *finite* (non-empty) set of symbols; e.g., $\Sigma = \{0, 1\}$

(Finite) string:

- $x = a_1a_2{\dots}a_n$, each $a_i \in \Sigma$.
- $x = 110$ (= 6 in binary form); both $x = 11$ and $x = 011$ equal 3.

Notations:

- $|x|$ = length of x; $|a_1a_2\cdots a_n| = n$ and $|\lambda| = 0$.
- Σ^* = The set of all strings over Σ.

From An Infinite Sequence $X = a_1a_2a_3\cdots$ to A Set of Strings:

- A *countable* set of finite strings: $L_X = \{a_1, a_1a_2, a_1a_2a_3, \cdots\}$.
- For any $x, y \in L_X$, one is an initial part (prefix) of the other.

Question:

•? Show L_X for $X = 101101110\cdots$.

•? How can we get back X from L_X for any X?

•? Give another way of converting an X into a countable set of finite strings so that we can get back X from L_X. Also, give a conversion method that does not always work.

> An infinite string X can be analyzed by analyzing the *infinite* language L_X.

Language: A subset of Σ^* for some Σ.

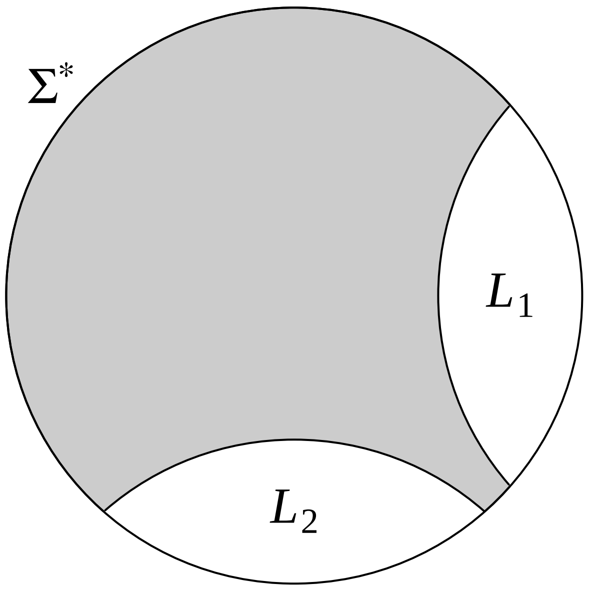
Question: Can we have an uncountable language?

EXERCISE

1. What goes wrong if we associate with the sequence $X = a_1 a_2 a_3 \cdots$ the set $\{a_1, a_2, a_3, \cdots\}$, i.e., the set of distinct symbols in X?
2. How can we associate with each finite set S of strings over an alphabet Σ a single infinite string $X(S)$ in an one-to-one fashion (so that we can get back S from $X(S)$)? (Hint: use a larger alphabet Σ' to construct $X(S)$.)
3. Repeat Problem 2 when $|\Sigma| \geq 2$ using only the symbols of Σ in $X(S)$. Explain your answer for $S = \{10, 01, 101, 1001\}$.
4. Using your solution to Problem 3, argue that one can do the same for a countable set S when $|\Sigma| \geq 2$.

LANGUAGES OVER Σ

- Each subset $L \subseteq \Sigma^*$ is a language over Σ.



COUNTABLE NUMBER OF FINITE LANGUAGES

1. For non-empty and finite Σ, Σ^* is countable: $x_1, x_2, x_3, \cdots$
2. For each $n > 0$, we can list all *finite* subsets of Σ^* which are of size n as illustrated below for $n = 1$ and $n = 2$.

   $$\begin{array}{ll} n = 1: & \{x_1\}, \{x_2\}, \{x_3\}, \cdots \\ n = 2: & \{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_3\}, \{x_1, x_4\}, \{x_2, x_4\}, \cdots \end{array}$$

   Question:

   •? Describe the method used in listing the subsets for $n = 2$.

   •? Why is the following listing not valid for $n = 2$?

   $$\{x_1, x_2\}, \{x_1, x_3\}, \{x_1, x_4\}, \cdots, \{x_2, x_3\}, \{x_2, x_4\}, \cdots, \cdots$$

3. For $n = 3$, we first list all subsets of size n which uses items $x_n = x_3$ and those preceding it, then we list all subsets of size n which uses $x_{n+1} = x_4$ and those preceding it, and so on.

   $$\underline{\{x_1, x_2, x_3\}}, \underline{\{x_1, x_2, x_4\}, \{x_1, x_3, x_4\}, \{x_2, x_3, x_4\}}, \cdots$$

4. The same idea works for all $n \geq 2$.
5. Now we first write the subsets of various sizes $n \geq 0$ as follows; for each $n \geq 1$ we have a countable many subsets of size n.

   $$\begin{array}{lllll} n = 0: & S_{01}\ (=\emptyset) & & & \\ n = 1: & S_{11}, & S_{12}, & S_{13}, & \cdots \\ n = 2: & S_{21}, & S_{22}, & S_{23}, & \cdots \\ n = 3: & S_{31}, & S_{32}, & S_{33}, & \cdots \\ \quad\cdots & \cdots & \cdots & \cdots & \cdots \end{array}$$

   Finally, we list all S_{ij}'s as follows: list all S_{ij} for a given $i + j = m \geq 1$, for successive values of m.

   $$\underline{S_{01}}, \underline{S_{11}}, \underline{S_{12}, S_{21}}, \underline{S_{13}, S_{22}, S_{31}}, \underline{S_{14}, S_{23}, S_{32}, S_{41}}, \cdots$$

EXERCISE

1. What goes wrong if we try to apply the diagonal argument to show that the finite languages are uncountable by first associating an infinite binary sequence with each finite subset as illustrated below (where we take the ith term in the sequence to be 1 if x_i belonging to the subset and 0 otherwise)? Note that these sequences has only a finite number of 1's.

$$\begin{array}{rl} \{x_1,\ x_2,\ x_6\}: & 1100010000\cdots \\ \{x_2,\ x_5,\ x_6,\ x_8\}: & 0100110100\cdots \end{array}$$

   Show that a similar problem arises if we apply the diagonal argument to the infinite languages.

2. Show that the set of all infinite languages over Σ are uncountable.

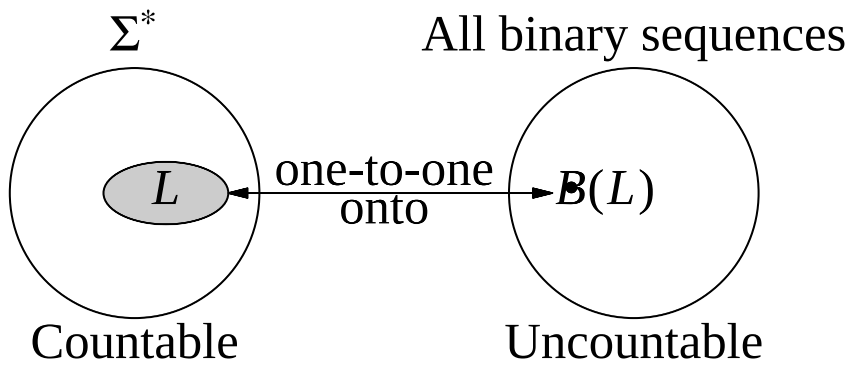
UNCOUNTABLE NUMBER OF LANGUAGES

- Let x_1, x_2, $\cdots$ be the strings in Σ^*, in some order, for an alphabet Σ.
- Given a language L over Σ, let $B(L) = b_1 b_2 b_3 \cdots$ be the associated binary *sequence*, defined by

$$b_i = \begin{cases} 1, \text{ if } x_i \in L \\ 0, \text{ otherwise} \end{cases}.$$

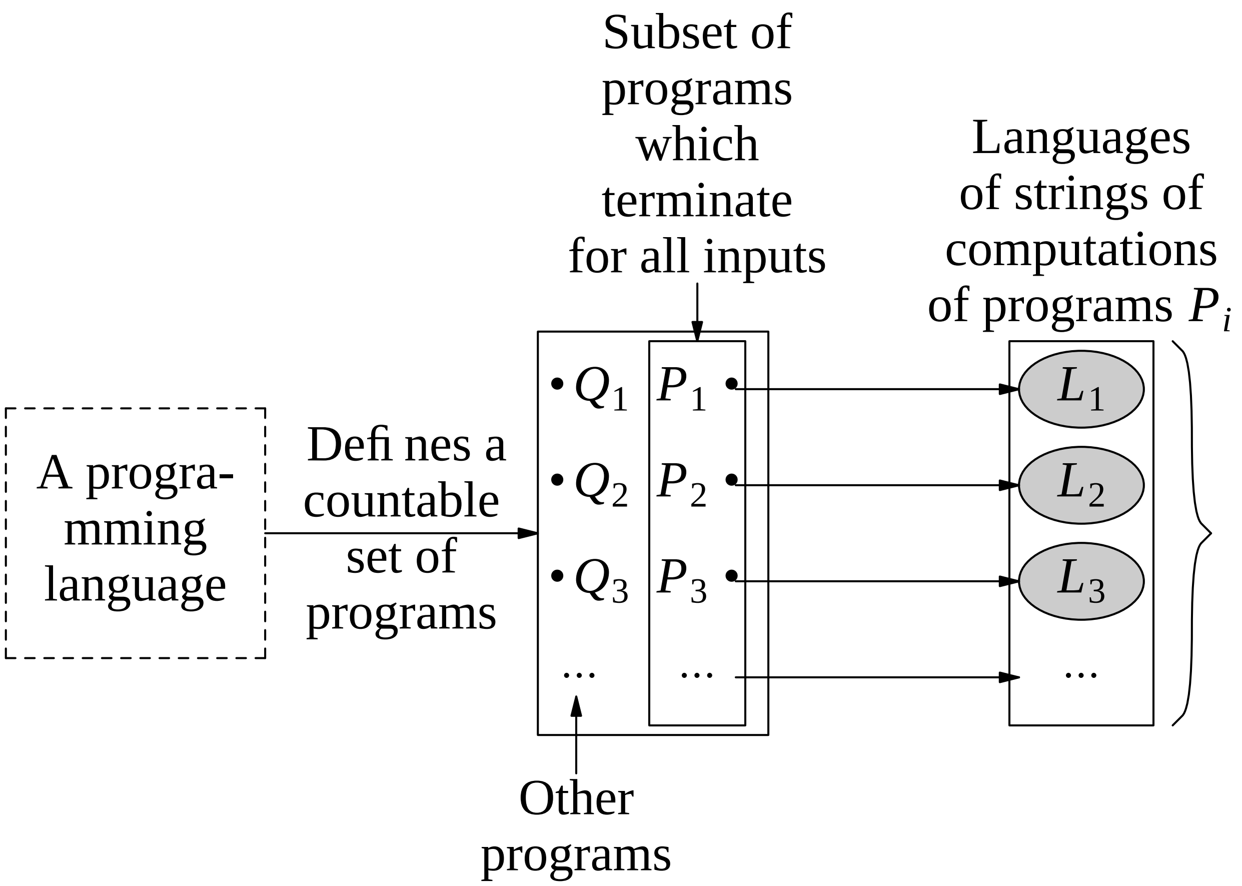
L	$B(L)$
1. $L_1 = \varnothing$ (empty language)	$B(L_1) = 000\cdots$
2. $L_2 = \Sigma^*$	$B(L_2) = 111\cdots$
3. $L_3 = \{a, aaa\}$, with $x_n = a^{n-1}$ ($n \geq 1$) and $\Sigma = \{a\}$	$B(L_3) = 0101000\cdots$
4. $L_4 = \{a, aaa\}$, with $\Sigma = \{a, b\}$ and $x_1 = \lambda$, $x_2 = a$, $x_3 = b$, $x_4 = aa$, $\cdots$	$B(L_4) = 01000001000\cdots$ (Although $L_4 = L_3$, $B(L_4) \neq B(L_3)$ because of different Σ.

Question: What is $B(L)$ for $L = \{a^n b^n \colon n \geq 1\}$, $\Sigma = \{a, b\}$, and the standard dictionary order listing of the strings in Σ^*.



There are uncountably many languages for any Σ.

COUNTABLE NUMBER OF LANGUAGES OF COMPUTATIONS


Subset of programs which terminate for all inputs
Languages of strings of computations of programs P_i
A progra-mming language
Defi nes a countable set of programs
Q_1 P_1
Q_2 P_2
Q_3 P_3
...
L_1
L_2
L_3
Other programs

- Not all L_i are distinct, and thus there are at most countably many distinct computation languages.
- Indeed, there are countably many distinct L_i's of the form $\{p^n\}$, $n \geq 1$ for programs which simply prints the integer 1 some fi xed n times (p = "print 1").
- The number of computation languages of programs is countable.

$\boxed{\text{Most languages of the fi nite computation strings do not correspond to any program.}}$

EXERCISE

1. Consider a program whose input consists of a non-empty string over $\Sigma = \{d, w\}$, where each d represents a deposit request of a unit amount and each w represents an withdraw request of a unit amount. The program outputs '+' for each successful deposit request and '−' for each successful withdraw request. Also, it prints '0' if a withdraw request fails because of unavailability of funds. Assume that there is no upper limit on the amount that can be deposited and that initial amount is 0. Which of the following are valid outputs:

```
++-+
++---
++--0-
++--0+
-
```

   Give a description of all output strings of the program; it should describe all outputs and only those. (Hint: Find a relationship with the valid *b*/*e* patterns.)

2. Consider the following variation of Problem 1. Assume that there is an upper limit of 3 for the amount held in deposit. Also, that the program outputs '1' for a failed deposit request. Is there a simple description of all output strings of the program?

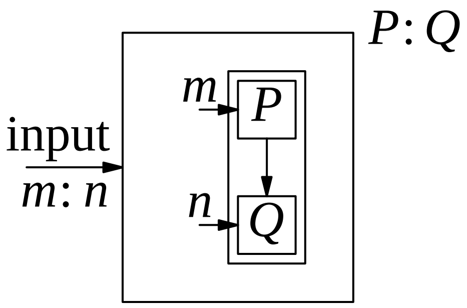
$$\boxed{\text{It is not possible, in general, to determine if two patterns-descriptions give the same set of strings.}}$$

COMBINATION OF PROGRAMS AND THEIR COMPUTATION PATTERNS

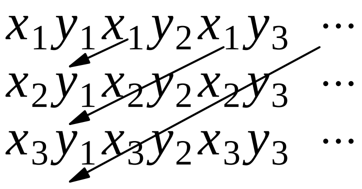
- Programs P and Q take integer inputs: 1, 2, $\cdots$ and we assume that each of them terminates for each input.
- P produces the computation-strings x_1, x_2, $\cdots$ corresponding to the inputs 1, 2, $\cdots$ in particular, the x_i's need not be distinct.
- Q produces likewise the computation-strings y_1, y_2, $\cdots$.

Sequential Combination : $P:Q$ with inputs $m:n$

The parts m and n are processed by P and Q in that order.

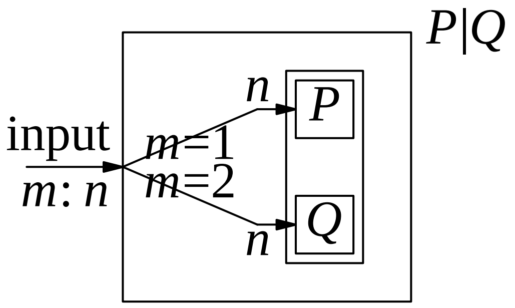


$P:Q$ has computation-strings x_1y_1, x_1y_2, x_2y_1, x_1y_3, $\cdots$.


x_1y_1 x_1y_2 x_1y_3 $\cdots$
x_2y_1 x_2y_2 x_2y_3 $\cdots$
x_3y_1 x_3y_2 x_3y_3 $\cdots$

EXERCISE

1. Consider the or-combination $P|Q$ of P and Q, where the inputs are m: n and $m = 1$ or 2. For $m = 1$, the part n is processed by P and for $m = 2$ the part n is processed by Q.



   Argue that $P|Q$ has the computation-strings x_1, y_1, x_2, y_2, $\cdots$.

2. Consider a program P whose computation language consists of the strings $\{b^m$: $m \geq 1\}$. You can think of P being a loop, where the loop-body performs the operation b; the number m can be thought of as the input to P. Let Q be another program whose computation language consists of the strings $\{a^n c^n$: $n \geq 1\}$. You can think of Q as having two loops, one loop following the other, with the loop bodies performing the operations a and c. The number n can be thought of as the input to Q; each loop in Q involves n iterations of its loop-body. What is the computation language of the program where the body of the loop in P is replaced by Q (i.e., Q replaces the operation b in P)? What does an input for the new program consist of?

3. Let $C(P)$ = the language of computations of a program P. What difficulties might arise in finding a program Q such that $C(Q)$ is the complement of $C(P)$?

4. Can you think of combining two programs P and Q in some other ways to produce different combinations of $C(P)$ and $C(Q)$?

DIFFERENT VIEWS OF AN ALGORITHM/COMPUTATION

$$\begin{tikzcd} {} \arrow[r, "\text{Input} = x"] & \boxed{\text{Algorithm}} \arrow[r, "\text{Output} = f(x)"] & {} \end{tikzcd}$$

View of a Mathematician (non-computational):

- What is the nature of a functional relationship $x \leftrightarrow f(x)$?

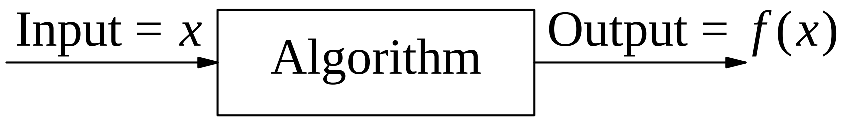
– He is not so concerned with *how to compute* the value $f(x)$ for a given x.

View of a Programmer (too low):

- How to compute $f(x)$? What are the best choices for data-structures, subroutines and their parameters, organization of data files, etc. in reducing the computation time and memory space (perhaps reduce one by using more of the other)?
- How to improve readability of the program.

– He is not so concerned with what sort of general "computing power" is *really* necessary to solve the problem. (A programmer uses whatever is available.)

CONTD.



Our view (focus on abstract/essential aspects of computation):

- Does it matter whether we read the input string from left to right vs. right to left?
- Can we compute $f(x)$ without looking at any part of the input x more than once?
- Can we avoid saving some of the intermediate results in the computation, perhaps by using extra computation time, if necessary?
- If we must save the results of intermediate computations, is there any particular order in which we might use those results to our advantage?
- Can we put limits on the number of intermediate results saved in terms of the size of the input and/or output?
- Can we distinguish between different "phases" in the computation? What would constitute such a phase? How many such phases are needed?
- Can we classify computation problems in some way based on the above issues? What are some typical problems in the various classes? How do we show which class a particular problems belongs to?

COMPLEXITY OF ORDINARY MULTIPLICATION METHOD

- Requires(1) repeated reading of parts of input, and (2) saving all intermediate results.

			5	1	9	3	(m = 4 digits)
				2	2	7	(n = 3 digits)
		3	6	3	5	1	
	1	0	3	8	6		($\leq (m+1)n = mn + n$ digits)
1	0	3	8	6			
1	1	7	8	8	1	1	($\leq m + n$ digits)

Question:

•? Can we avoid saving some of the intermediate results, say, by computing one row at a time after the first row and adding that row to the first row? Will it affect the computation time?

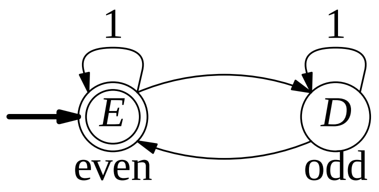
•? Is there a way to compute the successive digits of the final result from right to left without computing the intermediate rows (as was done above)?

WHAT CAN OR CANNOT BE COMPUTED?

- Is the number of 0’s in a binary string even?

100101	no
10011	yes

As we examine each symbol from left to right, say, we ignore the 1’s and pair up the 0’s. If there is an unpaired zero at the end, then the number of zero’s is odd; otherwise, it is even.



Question: What must be remembered from the symbols already seen?

- A slightly more complex problem: Are there more 0’s than 1’s in a given binary string? This cannot be done by FSM, but can be done using a push-down automata(PDA).

Question: How much memory do we need?
It depends on the length of the input string; longer the input string, more we need to remember, unlike problem (1) above.

COMPUTING AN ANSWER vs. VERIFYING A PROPOSED ANSWER

The addition problem:

```
  3 7                    3 7
  1 2                    1 2
 -----                  -----
  ? ?   (compute)        4 9   (verify)
```

- Both the computation and the verification of a given sum are easily done from right to left. (An FSM can do it.)
- The only memory needed is the "carry" and the table of the sum of two digits. The size of memory is *not dependent* of the input size.
- It is *not possible* to compute the sum from left to right. A "carry" generated some place arbitrarily far to the right might require modification of sum-digits computed earlier in left-to-right fashion, and these digits must be remembered.

```
 1 9 9 9 9
 1 0 0 0 1
 ---------
 3 0 0 0 0
```

- It is, however, *possible to verify* the correctness of a proposed sum in the left-to-right direction. We need only to remember whether a carry is to be generated.

An alternative form: 37+12=49 (verify a given sum)
37+12=? (compute sum)

- No FSM or PDA can handle two arbitrary integers in this form, either for computing the sum or for verifying a given sum.
- One must go back and forth to "pair up" the corresponding digits to the left and to the right of the ’+’ symbol to compute the sum-digits.

IMPORTANCE OF PROBLEM REPRESENTATION

- A change in the representation of a problem may make it more difficult to solve or even unsolvable.
- Two main tasks in a computation,
  (1) Which pieces of information are to be combined at any stage of a computation; this is part of algorithm design.
  (2) How to locate the required information at each stage of computation; doing it efficiently is data-structure design.

  and both can depend on problem-representation.

EXAMPLES OF INPUT-OUTPUT ALPHABETS

- Σ_1 = Input alphabet; Σ_2 = Output alphabet
- An alphabet symbol = unit of information that can be processed (input and output).

1. For the addition-computation problem in the form: 13182+57

$$\Sigma_1 = \{0, 1, 2, \cdots, 9, +\}; \qquad |\Sigma_1| = 11$$
$$\text{and } \Sigma_2 = \{0, 1, 2, \cdots, 9\}; \qquad |\Sigma_2| = 10$$

2. For the addition-verification problem in the form: 13182+57=13239

$$\Sigma_1 = \{0, 1, 2, \cdots, 9, +, =\}; \qquad |\Sigma_1| = 12$$
$$\text{and } \Sigma_2 = \{y, n\}; \qquad |\Sigma_2| = 2$$

3. For the addition-computation problem in the form: $\begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}3\\0\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix}\begin{pmatrix}8\\5\end{pmatrix}\begin{pmatrix}2\\7\end{pmatrix}$

$$\Sigma_1 = \{(i,\ j)\text{: } 0 \le i,\ j \le 9\}; \qquad |\Sigma_1| = 100$$
$$\text{and } \Sigma_2 = \{0, 1, 2, \cdots, 9\}; \qquad |\Sigma_2| = 10$$

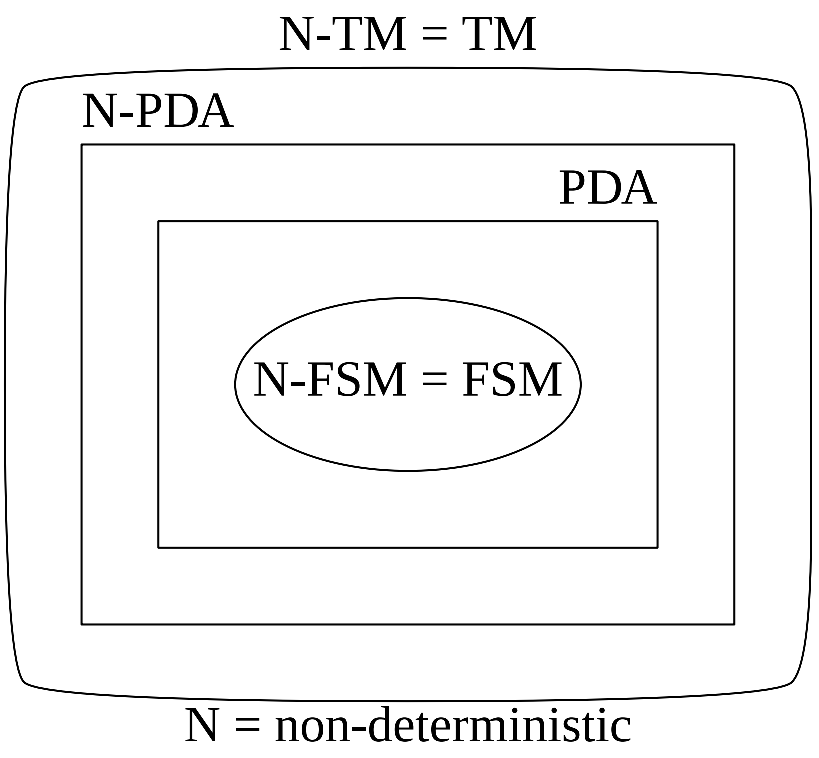
4. For the addition-verification problem in the form (the third item in the triplets give the sum-digits):

$$\begin{pmatrix}1\\0\\1\end{pmatrix}\begin{pmatrix}3\\0\\3\end{pmatrix}\begin{pmatrix}1\\0\\2\end{pmatrix}\begin{pmatrix}8\\5\\3\end{pmatrix}\begin{pmatrix}2\\7\\9\end{pmatrix}$$

$$\Sigma_1 = \{(i,\ j,\ k)\text{: } 0 \le i,\ j,\ k \le 9\}; \qquad |\Sigma_1| = 1000$$
$$\text{and } \Sigma_2 = \{y, n\}; \qquad |\Sigma_2| = 2$$

THREE MAIN COMPUTATION MODELS

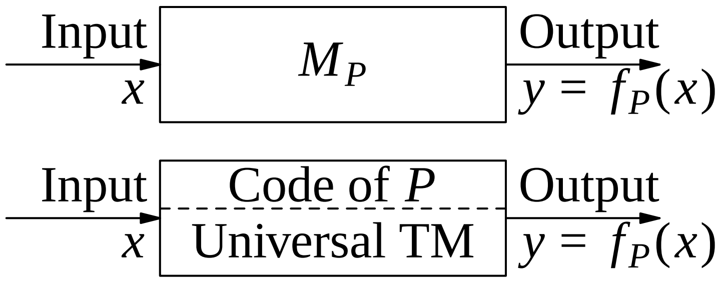
Machines	Restrictions on *external* memory size and access
Finite-state Automata (FSA), the simplest model	Zero external memory
Push-down Automata (PDA), intermediate between FSA and TM	Linearly bounded (in input length) and restricted access
Turing machine (TM), most general model	Unrestricted size and access



UNIVERSAL TURING MACHINE (PROGRAMMABLE MACHINE)

- A computer program P (in any programming language) is simply a description of a Turing machine M_P.
- A computer is a special machine, which can simulate (behave) like machine M_P given the description P.

  The computer itself is a very special TM, called a *universal* TM; it is actually a practical approximation, limited by its finite memory size, of an abstract universal TM.
- There is no other machine (screw-driver, telescope, automobile, spaceship) built/conceived by man or observed in nature having this sort of "universal" property.



- This universal nature of a computer allows us to use it in every application:
  - Robotics, manufacturing, chemical process control
  - Automobile, aircraft design
  - Design of genetically engineered bacteria and drugs
  - Text/document processing
  - Communications, etc.

EXERCISE

1. Can a universal TM solve the Halting Problem?
2. Does the following program halt for all inputs $m \geq 0$.

```
n = 1;
read(m);
Repeat the following:
        n = n+1;
        m = m/n; //integer division
            m = m*n;
    until m = 0;
```

3. Show that for each $k \geq 1$, there is some $m = m_k$ such that the program has exactly k iterations of the loop; do this by giving an algorithm to compute the smallest m_k. Illustrate your algorithm for the case of $k = 13$. Give a lower bound for the number of loop-iterations if $m = k!$; what is the connection between this result and the existence of m_k for each $k \geq 1$ in terms of which result implies the other?
4. Explain the statement "Computation, i.e., its result can be thought of as a transformation of strings to strings".

OPERATIONS ON LANGUAGES

Set-Theoretic Operations:

- Set-union $L_1 \cup L_2 = \{x\text{: } x \in L_1 \text{ or } x \in L_2\}$.
- Set-intersection $L_1 \cap L_2 = \{x\text{: } x \in L_1 \text{ and } x \in L_2\}$
- Set-complementation $L^c = \{x \in \Sigma^*\text{: } x \notin L\}$.

String-based Operations:

- *Concatenation* $x.y$ (or simply, xy) of two strings x and y.
  - For $x = 101$, $y = 1011$, and $z = 011$, $xy = 1011011 = yz$.
  - For any x, $x.\lambda = x = \lambda.x$; $\lambda.\lambda = \lambda$.
  - Associativity: $(xy)z = x(yz)$ for any x, y, and z.
  - Non-commutativty: $10 \cdot 11 = 11 \cdot 10$.

- For two languages L_1 and L_2, $L_1L_2 = \{x_1x_2\text{: } x_1 \in L_1 \text{ and } x_2 \in L_2\}$. Clearly, $L_1\{\lambda\} = L_1 = \{\lambda\}L_1$.
  - Associativity: $(L_1L_2)L_3 = L_1(L_2L_3) = \{x_1x_2x_3\text{: } x_i \in L_i\}$.
  - Short notations: $Lx = L\{x\}$, $xL = \{x\}$, $L^2 = LL$, $L^3 = LLL$, etc.

- *Kleene-star* $L^* = \{x_1x_2 \cdots x_k\text{: } k \geq 0 \text{ and each } x_j \in L\} = \{\lambda\} \cup L \cup L^2 \cup L^3 \cup \cdots$

  For $\Sigma = \{a, b\}$, $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \cdots\} = \{\lambda\} \cup \Sigma \cup \Sigma^2 \cup \cdots$

- *Reverse* of $x = a_1a_2 \cdots a_k$ is the string $x^r = a_ka_{k-1} \cdots a_1$; $\lambda^r = \lambda$. Reverse of L, $L^r = \{x^r\text{: } x \in L\}$.
- There are many other operations on languages, some of which will be seen later.

EXERCISE

1. What can you say about y in terms of x and z if $xy = yz$? What can you say about x and y if $xy = yx$?
2. Which of the following are true: (a) L^* is infinite if $L \neq \emptyset$. (b) $(L^*)^* = L^*$
3. Let $L = \{aa, bab\}$. Show that $L^* \neq \{x^n: x \in L \text{ and } n \geq 0\}$.
4. Find two non-trivial (other than $\emptyset$ and Σ^*) languages $L_1 \neq L_2$ over the alphabet $\Sigma = \{a, b\}$ such that $L_1.L_2 \subset L_2$.
5. Does $L_1^* = L_2^*$ imply $L_1 = L_2$?