COMP2022
Models of Computation

Computational Problems and Formal Languages

Sasha Rubin

August 1, 2024

THE UNIVERSITY OF SYDNEY

Two important ideas in computer science

Computational problem:

- defines a computational task
- specifies what the input is and what the output should be

Algorithm:

- a step-by-step recipe that solves the problem, i.e., that given an input produces the output
- different from implementation

The first question you should ask when you encounter a computational problem is this:

Is there an algorithm solving it?

If "yes", the second question might be:

How "simple" need such an algorithm be?[1]

[1] What is a "simple" algorithm? COMP2123 focuses on fast (aka polynomial time) algorithms. COMP2022 gives a different but related answer.

Let's look at three computational problems...

Pattern matching

Given a string p ("pattern") and a string t ("text"), decide if p occurs in t.

"456" occurs in "1234567890", but "1569" does not.

- There is an algorithm for solving Pattern Matching. We know this because Control-F does it all day long.
- Is there an algorithm that solves Pattern Matching that has a constant amount of memory?[2]

To answer these questions, we need a mathematical model of algorithm that has constant memory.

We will define and study these in lectures 2 and 3: finite-state automata

[2] Here "constant" means that the amount of memory does not depend on the input. What if we fix the pattern, say p is always "456". Now is there such an algorithm?

Syntax Checking

Given a string t, decide if t is a legal `Python` program.

```
1   print('Hello, World!')
```

- There is an algorithm for solving Syntax Checking. We know this because the Python compiler does this all day long.
- Can Syntax Checking be solved by the the constant memory model from the previous slide? ... "No".
- What is a simple model that allows one to solve Syntax Checking?

We will define and study such a model (that can handle a part of the `Python` syntax) in lectures 4 and 5: context-free grammars

Program equivalence

Given two `Python` programs, decide if they always give the same output when run on the same input, i.e., are equivalent.

- We suspect there is no algorithm for solving Program equivalence.
- How can we be sure?

To answer this, we need a mathematical model of general-purpose algorithm.

We will define and study such a model in lectures 6, 7 and 8:
Turing machines

In Lectures 9 - 12 we will study logic.

Boolean logic is the foundation of digital systems.

Predicate logic is the foundation of reasoning about computational systems in Computer Science and about knowledge in AI.

More on these later in the semester...

Strings, strings, strings

We make two assumptions in this UoS:

Assumption 1: Inputs to computational problems are strings.[3]

Assumption 2: Outputs of computational problems return a Boolean value (`True`, `False`). These are called decision problems.[4]

[3] This is ok because all data and programs can be thought of as strings.

[4] Is this ok?

Terminology

- Strings and alphabets
- Languages
- Operations on strings and languages

Let's see what these words mean...

Let's recall strings from `Python`:

- A string is a sequence of characters, e.g., `"abc123"`.
- Concatenation of strings: `"abc"+"123"="abc123"`
- Length of strings: `len("abc") = 3`
- String of length zero: `''`

- Characters come from a finite non-empty set, we call an alphabet.
- In Python, the alphabet is the set of Unicode code points.
- We will use other alphabets, and use letters like Σ to name the alphabets:
  - $\Sigma = \{0, 1\}$ is the classic binary alphabet.
  - $\Sigma = \{a, b, c, \cdots, z\}$ is the lower-case English alphabet.
  - $\Sigma = \{0, 1, 2, +, -, \div, \times, (,)\}$ is the arithmetic expression alphabet for base 3.
  - $\Sigma =$ ASCII characters

Notation

We will use slightly different notation, so you should get used to it.

- We name strings using letters like u, v or w, and write $w = w_1 w_2 \cdots w_n$ where each w_i is a character in the alphabet
- The concatenation of strings u and v is the string uv
- We use exponential notation u^k (read 'u to the power of k') to mean "concatenate u with itself k times"
  - if $u = 011$ then $u^3 = 011011011$
  - u^0 is defined to be the empty string
- The empty string is written ϵ
- The set of all strings over the alphabet Σ is written Σ^* (read "Sigma star")[5]

[5] This notation will be explained later.

Definition

A set $L \subseteq \Sigma^*$ of strings is called a language over Σ.[6]

Example of languages (over alphabet $\Sigma = ASCII$)

- The set of strings t that contains the string 011
- The set of all strings t where t is a legal `Python` program
- The set of strings of the form (t_1, t_2) where t_1, t_2 are legal `Python` programs that always give the same output when run on the same input

[6] Sometimes these are called *formal languages* to distinguish them from natural languages, like English.

Decision Problems = Languages

Every decision problem on strings can be viewed as a language.

How?

As the set of strings for which the answer is `True`

The decision problem

> **Input**: string p over alphabet Σ
> **Output**: `True` if p is a legal `Python` program, and `False` otherwise.

has a corresponding language

$$L_{\text{Python}} = \{p \in \Sigma^* \mid p \text{ is a legal } \texttt{Python} \text{ program}\}$$

The generic problem we study is parameterised by a language $L \subseteq \Sigma^*$, called the membership problem for L:

Input: string x over alphabet Σ
Output: `True` if $x \in L$, and `False` otherwise.

Self-test

Input: pair of strings (p, t) where $p, t \in \Sigma^*$
Output: `True` if p occurs in t, and `False` otherwise.

Which of the following are in the language corresponding to this decision problem?

1. $(01, 000111)$
2. $(000111, 01)$
3. $(0011, 1100)$

Operations on languages

We now define some operations on languages, i.e., ways to form new languages from old ones.

Let A, B be languages over Σ.

The new languages will also be over Σ.

- Union of languages
- Intersection of languages
- Complement of languages
- Concatenation of languages
- Star of languages

Assumed knowledge about sets

- $A \cup B$ (read "A union B") is the set of strings that are in A, or in B, or in both.
- $A \cap B$ (read "A intersection B") is the set of strings that are in A and in B.
- $A \setminus B$ (read "A minus B") is the set of strings that are in A but not B.

Definition

Define AB (read "A concatenate B", or just "A cat B") as the set of strings formed by concatenating a string from A with a string from B.

In math:

$$AB = \{xy \in \Sigma^* : x \in A, y \in B\}$$

We use exponentiation notation A^k (read 'A to the power of k') to mean concatenate A with itself k times.

- For $k \in \mathbb{Z}^+$, define A^k to be $\overbrace{AA\cdots A}^{k}$
- Convention: A^0 is defined to be $\{\epsilon\}$.
- Note that $A^{n+m} = A^n A^m$ (even if n or m is zero).

Definition

Define A^* (read "A star") as

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \dots$$

Think of A^* as "repeated concatenation", but without fixing the number of iterations.

Sometimes you will see a plus instead of a star: A^+ (read "A plus") is $A^1 \cup A^2 \cup A^3 \cup \dots$.

Self-test

Which of the following strings are in the set $\{00, 11\}^*$?

1. 010
2. 001100
3. 11
4. ϵ

Self-test

Which of the following is equal to the set $\{0a, 1a, 0b, 1b\}$?

1. $\{0, 1\} \cup \{a, b\}$
2. $\{a, b\} \cap \{0, 1\}$
3. $\{0, 1\}\{a, b\}$
4. $\{a, b\}\{0, 1\}$

Where are we going with this?

The first half of this course is about ways to describe languages and solve their membership problems. We do this with increasingly more expressive computational models.

1. Regular expressions, finite automata
2. Context-free grammars
3. Turing-machines

COMP2022|2922
Models of Computation

Regular Expressions

Sasha Rubin

August 1, 2024

THE UNIVERSITY OF SYDNEY

- In math, we build arithmetic expressions out of numbers like $0, 1, 2, etc.$ and operations like $+$ and $\times$.
- Arithmetic expressions represent values, e.g., $(1 + 3) \times 4$ has value 16.
- Similarly, the regular expressions are built out of symbols and operations for union, concatenation, and star. They represent languages.

Regular expressions in a nutshell

Expressions that describe "simple" pattern-matching languages.

Uses:

- Text processing
- Features in (machine learning) classifiers

  `COMP5046:Natural Language Processing`
- Scanners (aka Lexical analysers, Tokenisers)
- Specification of data formats
- Foundations of query languages for graph databases

NB. Our regular expressions are based on three language operations: Union, Cat, Star. Many implementations have extra features (which we will not study in this course).

Regular expressions: Syntax

Definition

Let Σ be an alphabet. The regular expressions over Σ are defined by the following recursive process:

1. The symbols $\emptyset$ and ϵ are regular expressions
2. Each symbol a from Σ is a regular expression
3. If R_1, R_2 are regular expressions then so is $(R_1 \mid R_2)$
4. If R_1, R_2 are regular expressions then so is $(R_1 R_2)$
5. If R is a regular expressions then so is R^*

We can read:

- $(R_1 \mid R_2)$ as "R_1 or R_2"
- $(R_1 R_2)$ as "R_1 cat R_2"
- R^* as "R star"

Intuition

For example, if $\Sigma = \{a, b, c\}$ then the following is a regular expression:

$$((a^* \mid b^*)(ac))^*$$

Why?

Notation

We may drop the outermost parentheses to improve readability.

- we may write $a \mid \emptyset$ instead of $(a \mid \emptyset)$
- we may write ab^* instead of (ab^*), which is different from $(ab)^*$

What do regular expressions mean? How do they represent languages?

Language represented by a regular expression

Write $L(R)$ for the language that the regular expression R represents.

We define $L(R)$ recursively as follows:

1. $L(\epsilon) = \{\epsilon\}$
2. $L(\emptyset) = \{\}$
3. $L(a) = \{a\}$ (for $a \in \Sigma$)
4. $L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$
5. $L(R_1 R_2) = L(R_1) L(R_2)$
6. $L(R^*) = L(R)^*$

If $x \in L(R)$ we say that string x matches regular expression R

Self test

The string aab matches which of the following regexps?

1. $(ab)b$
2. $a(ab)$
3. $(aa)|b$
4. a^*b^*

Recursion

In this course, we will provide recursive procedures for defining many objects and properties. Why?

1. Precise, unambiguous.
2. Can be implemented (often as is!) as a recursive algorithm.

In the tutorial you are asked to write a function $\mathrm{Match}(R, x)$ that takes a regular expression R and a string x as input, and decides whether or not x matches R.

Notation

- $L(R)$ is called the language specified/represented/defined by R, or the language of R.
- We may drop parentheses for associative operations.
  - we may write $(R_1 \,|\, R_2 \,|\, R_3)$ instead of $((R_1 \,|\, R_2) \,|\, R_3)$
  - we may write $(R_1 R_2 R_3)$ instead of $((R_1 R_2) R_3))$
- If $A = \{a, b, c\}$ we may write A instead of $(a \,|\, b \,|\, c)$.
  - so, e.g., we may write A^* instead of $(a \,|\, b \,|\, c)^*$.
  - we may do this for any finite set A of strings.

Which languages can be specified by regular expressions?

Regular expressions can be used to specify keywords and identifiers in programming languages[7]

- The set of identifiers in `Python`
- Signed and unsigned integers
- Hexadecimal numbers prefixed with '0x'

However, for specifying expressions and statements, we will need a more powerful model (context-free grammars, later).

For instance, the set of well-formed arithmetic expressions is not the language of any regular expression! We will prove this fact, later.

[7] Keyword = reserved word; identifier = user-defined name

Matching problem for regular expressions

Input: regular expression R, string x (over alphabet Σ)
Output: `True` if x matches R, `False` otherwise

When you see a computational problem you should probably ask yourself this:

Is there a (fast) algorithm for solving the problem?

Matching problem for regular expressions

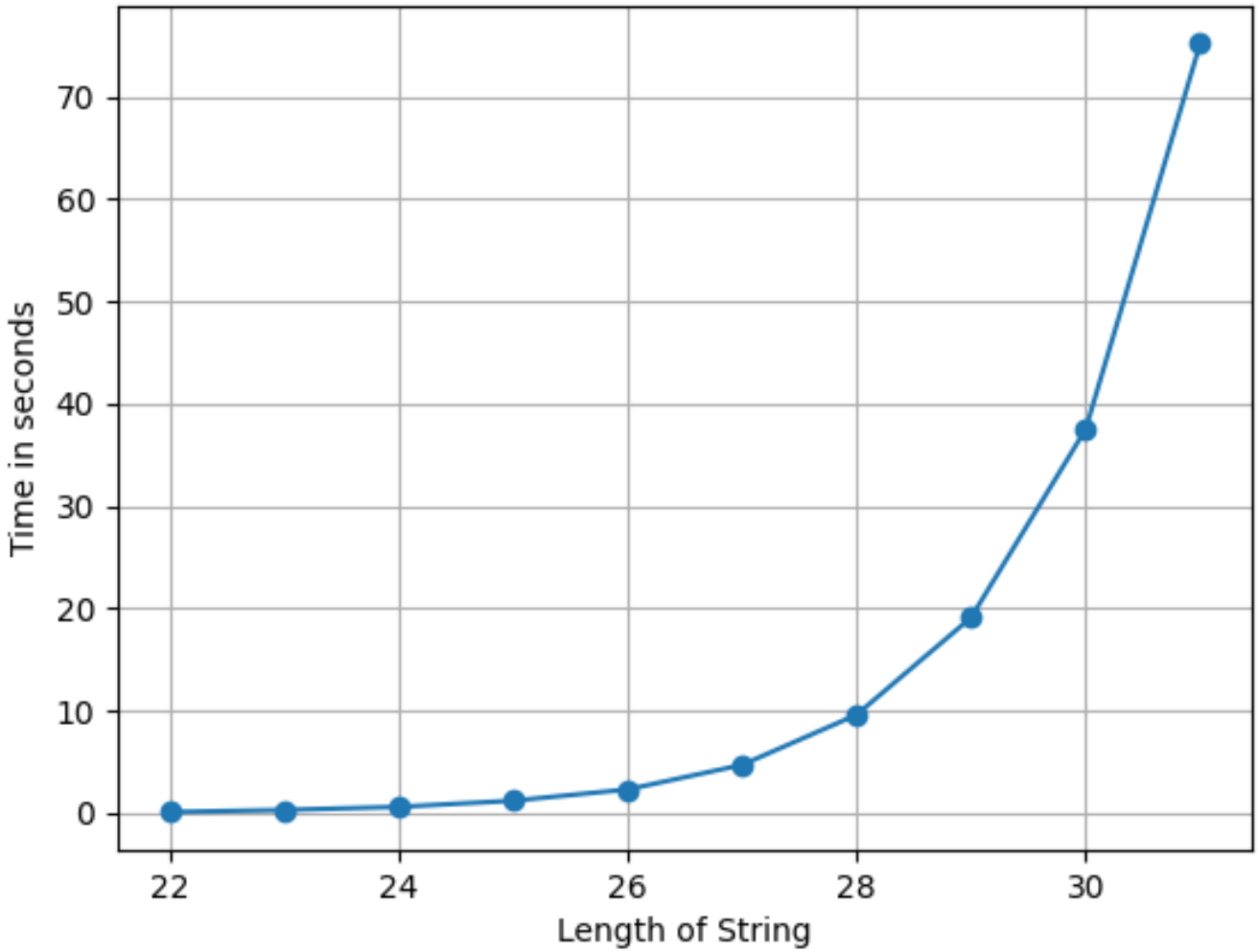
The naive recursive algorithm (Tutorial) runs in time that is exponential in the lengths R and x.

Is there an algorithm that runs in polynomial time?

What if we fix the regular expression R and only let the string x vary?

How does `Python` do?

```
1  import re
2  import time
3  r = '^(a+)+$'
4  for i in range(22,32):
5      s = 'a'*i + 'b'
6      start_time = time.time()
7      re.match(r,s)
8      print(i, (time.time() - start_time))
```


Time in seconds
0
10
20
30
40
50
60
70
22
24
26
28
30
Length of String

To design a polynomial time algorithm we will need to first learn about a model of computation called automata (next time).

Extra slides that you should read in your own time

Assumed knowledge

Here A, B are languages:

- $A \cup B$ (read "A union B") is the set of strings that are in A, or in B, or in both:

$$A \cup B = \{x \in \Sigma^* : x \in A \text{ or } x \in B\}$$

- $A \cap B$ (read "A intersection B") is the set of strings that are in A and in B:

$$A \cap B = \{x \in \Sigma^* : x \in A \text{ and } x \in B\}$$

- $A \setminus B$ (read "A minus B") is the set of strings that are in A but not B.

$$A \setminus B = \{x \in \Sigma^* : x \in A \text{ and } x \notin B\}$$

Definition

Define A^* (read "A star") as the set of strings that includes the empty string and all strings of the form $x_1 x_2 \cdots x_k$ where $k \geq 1$ and each $x_i \in A$.

In math:

$$A^* = \{x_1 x_2 \cdots x_k \in \Sigma^* : k \geq 0, \text{each } x_i \in A\}$$

We can also write this as

$$\begin{aligned} A^* &= \bigcup_{n \in \mathbb{Z}_0^+} A^n \\ &= A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots \end{aligned}$$

We define A^+ (read "A plus") to be $A^1 \cup A^2 \cup A^3 \cup \ldots$.

Think of A^* as "repeated concatenation", but without fixing the number of iterations.

Regular expressions: Semantics (wordy)

We say that a string matches a regular expression according to the following rules:

1. No string matches the regexp $\emptyset$.
2. Only the empty string matches the regexp ϵ.
3. For alphabet symbol $a \in \Sigma$, only the string a matches the regexp a.
4. A string matches $(R_1 \mid R_2)$ if it matches R_1 or if it matches R_2.
5. A string matches $(R_1 R_2)$ if it can be written as $u_1 u_2$ where u_1 matches R_1 and u_2 matches R_2.
6. A string matches R^* if either it is the empty string, or if it can be written as $u_1 u_2 \cdots u_k$ (for $k \geq 1$) where each u_i is a string that matches R.

Note that these rules are recursive!

Some edge cases

- The empty set $\emptyset$, also written $\{\}$, is a language with no elements in it
- The set $\{\epsilon\}$ is a language with one element in it, the empty string.
- The languages $\{0, 1\}$ and $\{1, 0\}$ are equal, but the strings 01 and 10 are not.

If we want to be very careful...

We are **overloading** notation for convenience.

- $\emptyset$ is a **symbol** used in regexps and the empty **set** of strings.
- ϵ is a **symbol** used in regexps and a **string**.
- $a \in \Sigma$ is a **symbol** used in regexps and an **alphabet symbol** and a **string** of length 1.
- * is a **symbol** used in regexps and an **operation** on sets of strings.
- some texts write $+$ or $\cup$ in regular expressions instead of $|$.

Equivalence problem

Input: two regular expressions R, S (over alphabet Σ)
Output: `True` if $L(R) = L(S)$, `False` otherwise

Is there even any algorithm for this problem?

It seems one would have to check that every string that matches R also matches S (and vice versa).

But such an algorithm would run forever if the two expressions did indeed match the same strings!

There is an algorithm, that will make use of automata (introduced in the next lecture).

Function problems

Many problems in programming are most naturally thought of as function problems.

Given an input x, compute and return an output $f(x)$

eg. We have a phonebook, stored as a linked list. Given a name (input), return that person's number (output) (or give an error message if they aren't in the phonebook)

Function problems seem a lot more useful than decision problems... so why do we only care about decision problems?

From functions to decisions

It turns out we can convert any function problem into a decision problem. How?

Let the function problem be "given x, return f(x)"

Assume the output is written in binary and has fixed length

Then the decision problem is "given x and a positive integer n, is the n-th bit of f(x) 1?"

If we can solve this decision problem, we can solve the function problem. (How?)

Hence, decision problems are as powerful as function problems, but usually simpler to think/reason about, so it makes sense to focus on them

Removing assumptions

What if the output is not in binary?

Option 1: encode it into binary (using eg. ASCII, Unicode, or just any mapping from strings to binary)

Option 2: change the decision problem to "given x, n, a, where a is a character, is the n-th character of f(x) a?"

What if the length of the output is not fixed?

Option 1: change the encoding of the answer to allow end of string (eg. odd-numbered bits are bits of the answer, even-numbered bits are 1 if EOS, 0 if not, so 1001011 maps to 10000010001011)

Option 2: use 2 decision problems, where the 2nd one is "given x,n, is the length of f(x) at least n?

What's the point?

Are these conversions practical? Should you turn everything into a decision problem when programming?

No - in practice, you'd just solve the function problem directly. So what's the point?

Given something large, complex, and difficult to analyse, it makes sense to break it down into something smaller and simpler

Thinking about yes/no outputs, rather than "absolutely anything" outputs, is easier