Building Blocks of Theoretical Computer Science
Computations

Version 1.0 - January 27, 2020

Introduction

What is a computation? This is an incredibly complicated question and there is a provisional answer: "A computation is anything that can be represented by a Turing Machine." (Church-Turing Thesis.)

What requirements do we need? There is a natural idea a computation is composed of a finite number of steps, each of which does a finite amount of work. We need an input and we need an output. Each of these needs to be finite.

Input: Finite strings over a finite alphabet (set of symbols).
Output: Finite strings over a finite alphabet. (Maybe different from the input alphabet, but usually the same.

We can represent rational numbers and integers, but irrational numbers are problematic.

Function: A mapping of the input to the output.

Example Odds: Is this natural number in binary odd or even. The input is strings over {0, 1} like 0, 001, 1011, etc. The output is True or False depending on whether the string is odd.

Computation:

0) Algorithm

A specification of how to take an input and return the correct output. An output must take a finite number of steps each of which requires a finite amount of time. This means that an output is produced in a finite amount of time. This does not require us to produce an output for every input (though we may add that as a requirement for a "good" algorithm. We may not know for which inputs the algorithm will give us an output.

1) Program

Probably the first thing you thought of as a computation was a program in some language like Python or Java. With the odds example, we can write a function in Python. Often the computation requires some kind of iteration

```
def odd(binary_string):
      result = False
      for char in binary_string:
            if char == '1':
                  result = True
            else:
                  result = False

def odd(binary_string):
      result = False
      if binary_string == "":
            return result
      else binary_string[len(binary_string)-1] == "1":
            result = True
      return result
```

Each of the steps requires a finite amount of time to complete, though we have used some complicated steps that may require time that scales up with the size of the input.

2) Recursive functions (Lambda calculus).

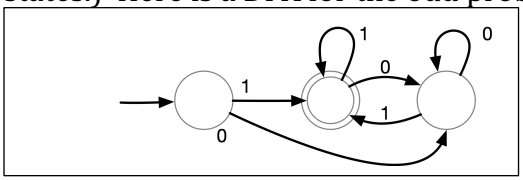
Another way that people have thought of computations is to use recursive functions without iteration. So we can define our function recusively as:

$$\text{odd(b)} = \begin{bmatrix} null & False \\ \text{"0"} & False \\ \text{"1"} & True \\ b & odd(tail(b)) \end{bmatrix}$$

Here tail is a function that returns a new string with the first character striped off. Lambda calculus was the inspiration behind the programming language Lisp which is the grandfather of functional programming languages. Note: Very few languages are purely iterative or purely functional.

3) State Transition Machines

Another way of thinking about computations is to build up a machine that has some number of states. Input will trigger a transition from one state to another state. One of the simplest of this kind of machine is a deterministic finite state automaton (DFA). (The finiteness of an algorithm, forces that we only have a finite number of states.) Here is a DFA for the odd problem:



Every time it gets a new character, it will follow the appropriate arrow to a new state. Given any finite input of length n, it will visit

at most n+1 states. When it stops, if the state is an accepting state, then the output is true. Otherwise, false.
A Turing Machine (TM) is a finite state machine with some extra memory that it can use to help in the computation.

4) Replacement Rules/Grammars
Another way of thinking about a computation is more mathematical. You apply logic steps until you reach the answer. You are given a set of replacement rules and can replace the left-hand side with the right-hand side. Repeat until no rule is applicable.

Here are a couple examples.

Example 1: In this example, we start with a non-empty input and rewrite it to 0 or 1. The truth value is True for 1 and False otherwise.

$01 \rightarrow 1$
$00 \rightarrow 0$
$10 \rightarrow 0$
$11 \rightarrow 1$

Here is a sample derivation where the left-hand side of the replacement rule marked with yellow highlight. Because each rule reduces the length of the string by 1, we can guarantee that eventually we will hit something that cannot be further rewritten.

$$1001011 \rightarrow 101011 \rightarrow 01011 \rightarrow 0101 \rightarrow 001 \rightarrow 01 \rightarrow 1$$

Example 2: In this example, we start with the symbol S and rewrite until we have just a string of 0's and 1's. Any string that can be generated is odd. Because every rule adds one character to what we are generating, we can limit the length of the derivation and decide if the string can be derived within a finite amount of time.

$S \rightarrow 1$
$S \rightarrow 0S$
$S \rightarrow 1S$

Here is a sample derivation.

$$S \rightarrow 1S \rightarrow 10S \rightarrow 100S \rightarrow 1001S \rightarrow 10010S \rightarrow 1001011$$

5) Languages

One particular kind of computation restricts the output to True/False. In this setting, we can talk about a language which is just the set of strings that have some property. So for example, we could talk about the language of odd words over the alphabet {0,1}. Odds = {1, 01, 11, 101, 111, …}. We can then build a ***recognizer*** that determines if an arbitrary word is in the language (true) or not in the language (false.)

One way of specifying a language is to use a regular expression. Regular expressions will have a finite alphabet and support the following operations.

A regular expression can be

1) the empty string
2) any single character from the alphabet
3) if α and β are regular expressions, then $\alpha\beta$ is a regular expression (concatenation)
4) if α and β are regular expressions, then $\alpha|\beta$ is a regular expression (or, choose one)
5) if α is a regular expression, then (α) is a regular expression (grouping)
6) if α is a regular expression, then α^* is a regular expression (Kleene star, zero or more copies)

By convention, the order of precedence is (), *, concatenation, and |.

So the regular expression

01*0|0

is read as

A zero followed by any number of ones, followed by a zero OR a single zero.

Which is different than

(01)*0|0

And is read as

Any number of 01's followed by a zero OR a single zero.

Which is different than

01*(0|0)

And is read as

A zero followed by any number of ones, followed by either a single zero or a single zero.

Definition: A ***regular language*** is a set of words for which there is some regular expression. The language of odd binary string is regular because it corresponds to the regular expression

(0|1)*1