Mathematics 3670: Computer Systems Introduction

Dr. Andrew Mertz

Mathematics and Computer Science Department
Eastern Illinois University

Fall 2012

Week 1: to do

What	**When**
Read Chapter 1	this week
Read Lab 1 handout	before Thursday
Determine Lab 1 Turing machine snapshots	before Thursday
Design Turing machine for halving	before Thursday
Complete Lab 1	this Thursday
Submit Lab 1 work	on/before next Thursday

The big picture

- Complex systems can be organized as a hierarchy of abstractions
- Bottom level is most basic; upper levels appear most complex
- We will study this hierarchy from the bottom up
- Key idea: how is level $N + 1$ implemented given level N?
- Ultimate aim – understand how the top level is achieved: no magic allowed!

A hierarchy

Problems

$\uparrow$

Algorithms

$\uparrow$

Language

$\uparrow$

Machine architecture (ISA)

$\uparrow$

Microarchitecture

$\uparrow$

Circuits

$\uparrow$

Devices

Problem solving with computers

- Begin with a problem statement
  - What are the inputs?
  - What are the desired outputs?
  - What is the relationship between inputs and outputs?
- Design an algorithm, which will transform inputs to outputs
- How do we express the algorithm?
  Ultimately, we want the algorithm to become a well-defined pattern of electrons flowing within a physical computer

A simple problem

- Input: A non-negative integer n
- Output: $n \bmod 3$

To design an algorithm for this problem, we need to know:

- what primitive operations are available
- what abstraction level is appropriate

Depending on the abstraction level, we may also need to be aware of data representation

The Turing machine

- Proposed in 1936 by English mathematician Alan Turing
- Can be used to formalize the idea of algorithm
- Is simple to describe
- Like modern computers, operates at a very basic level: any one step within a computation doesn't do very much

The Turing machine: basic ingredients

- A tape, divided into squares – infinite in both directions
- A read/write head which can inspect and change the contents of one square on the tape
- A finite control unit which remembers the "state"
- A set of states with one initial state
- A subset of states called final states
- A finite table of actions which controls how the machine makes one computational step

Turing machine actions

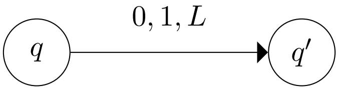
Any one action of the Turing machine is described by five components:

- current state
- current symbol
- symbol to write
- next state
- direction to move read/write head: left, right, stay

For example, $(q, 0, 1, q', L)$ tells the machine "if in state q the read/write head is scanning the symbol 0, then overwrite it with the symbol 1, switch to state q' and move the read/write head one step to the left"

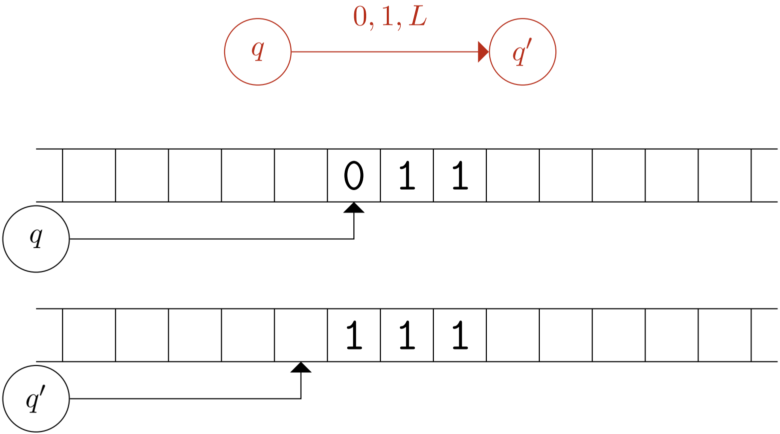
Turing machine actions

The action $(q, 0, 1, q', L)$ can be viewed as an edge in a directed graph:

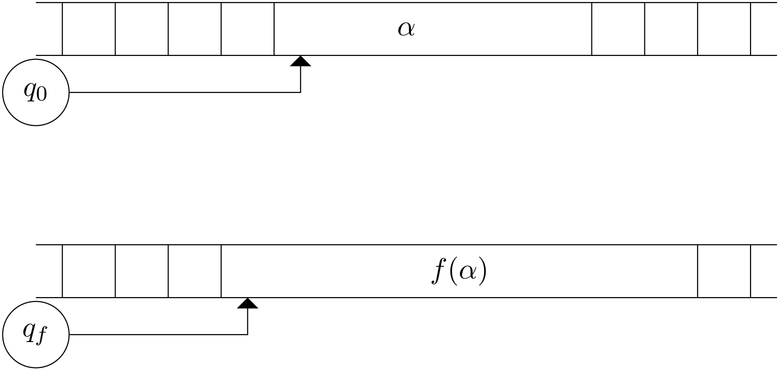


We can describe a Turing machine with a collection of actions like these, giving us a labeled, directed graph

Turing machines: one computation step



Computing functions with Turing machines

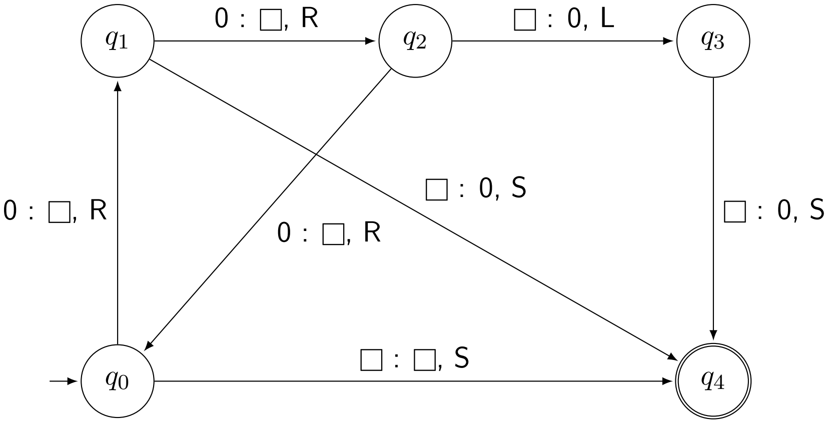


Back to the mod 3 problem

- We will represent n in unary
  For input 35, place 35 consecutive 0's on the tape
  We want to end up with $35 \bmod 3 = 2$, i.e., 2 consecutive 0's
- Division by 3 can be accomplished by repeated subtraction
- Challenge: what states and transitions are needed?

Let's think about it. . .

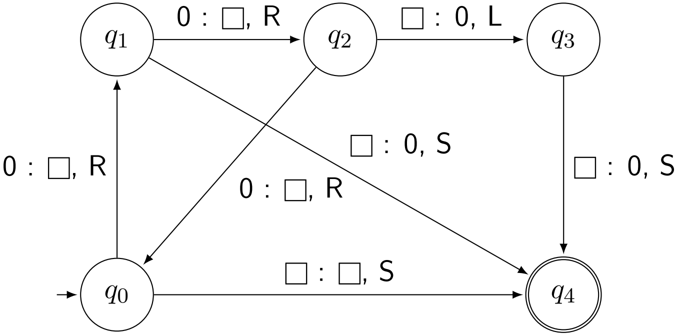
A Turing machine solution for the mod 3 problem



Simulating a Turing machine

Live demo of JFLAP

Understanding the mod 3 Turing machine



- The q_0, q_1, q_2 cycle subtracts 3 on each complete loop
- q_0 – so far, we have removed $3t$ zeros
- q_1 – so far, we have removed $3t + 1$ zeros
- q_2 – so far, we have removed $3t + 2$ zeros
- $q_2 \rightarrow q_3 \rightarrow q_4$ writes 00, then halts
- $q_1 \rightarrow q_4$ writes 0, then halts
- $q_0 \rightarrow q_4$ writes □, then halts

Turing machines as black boxes

$$n \longrightarrow \boxed{T_{\bmod 3}} \longrightarrow n \bmod 3$$

Black box view is an essential abstraction:

- Hides inessential detail
- Allows for understanding of “big picture”

Universal computational device (Turing, 1936)

- $T_{\bmod 3}$ does one task and one task only
- If you want to perform some other task, you need a different machine
- Could we design one machine that can do the work of any other machine?
- Yes! This is the machine we call U — the universal machine

$$M, x \longrightarrow \boxed{U} \longrightarrow \text{output of } M\text{, given } x$$

- U simulates what M would do — a programmable computer!

Turing's thesis

If an algorithm exists for some problem, there is an equivalent Turing machine

Turing's work provides a foundation for understanding the limits of computation — what is possible to compute and what is impossible to compute

Important idea #1 (textbook)

All computers (big, small, fast, slow, . . .) are capable of computing exactly the same things, given enough time and enough memory

Important idea #2 (textbook)

Problems we wish to solve with a computer are stated in some natural language, such as English. Ultimately, these problems are solved by electrons running around inside the computer. To achieve this, a sequence of systematic transformations takes place. At the lowest levels, very simple tasks are being performed.

The hierarchy, revisited

Problems

$\uparrow$

Algorithms

$\uparrow$

Language

$\uparrow$

Machine architecture (ISA)

$\uparrow$

Microarchitecture

$\uparrow$

Circuits

$\uparrow$

Devices

Problem statement

- Stated in a natural language, like English
- We must avoid any ambiguity
- Misunderstandings at this level will cause many issues later in the development, increasing development cost, delaying completion
- Obtaining an accurate specification of the problem is often difficult

The algorithm: essential ingredients

- Some number of inputs
- Some number of outputs
- Definiteness property: each step must be precisely defined
- Effectiveness property: each step must be something that can be carried out by a person in a finite amount of time
- Finiteness property: the algorithm, when followed, must terminate after a finite number of steps

Definiteness – you be the judge

- Suppose m and n are two arbitrary integers; positive, zero, or negative
- We are devising an algorithm that uses “integer” division, with quotient and remainder; one step is:

  let k be the remainder of $m \div n$

- For definiteness, we need a precise definition of the action
- Do we have that?

Effectiveness – you be the judge

- Suppose we are devising an algorithm which uses floating point arithmetic; one step is:

  if there are 7 consecutive 3's in the decimal expansion of π then ...

- For effectiveness, each step must be something that can be
  - basic enough to be carried out by a person
  - completed in a finite amount of time
- Do we have that?

Finiteness – you be the judge

- Suppose we are computing with exact, rational arithmetic

  Let x be 1/7
  Determine m and $d_1, d_2, \ldots, d_m$ so that
  $0.d_1 d_2 \ldots d_m = x$

- For finiteness, each step must terminate after some finite number of steps
- Do we have that?

From algorithms to programs

- To implement an algorithm, we need a programming language
  - High level: Java, C++, C, COBOL, FORTRAN, Python, Perl, Prolog, etc.
  - Low level: Assembly language
- High level – machine independent
- Low level – tied to a specific architecture

The ISA – Instruction Set Architecture

- Ultimately, programs are expressed as patterns of 0's and 1's: machine language
- Translators (compilers and assemblers) perform conversion, producing machine language from higher levels
- ISA specifies:
  - instruction set (what operations are possible?)
  - data types (e.g., integer vs. floating point, range and precision)
  - addressing modes (how is an operand located in the memory?)
- Typical ISAs: Intel x86, HP PA-RISC, Sun Sparc, ARM, Motorola 68k, etc.

Microarchitecture

- microarchitecture: implementation of an ISA
- To add x to y requires lower level details, register transfers, etc.
- There can be multiple implementations of an ISA

Logic circuits

- Fundamental building blocks: AND, OR, NOT
- Very simple: one bit operands
- Use these building blocks to create ALUs, memories, etc.

Device level

- Technologies: CMOS, NMOS, GaAs, etc.
- Solid-state physics, electrical engineering