Introduction to Computing

Explorations in Language, Logic, and Machines

David Evans
University of Virginia

For the latest version of this book and supplementary materials, visit:

http://computingbook.org

Version: August 19, 2011

Contents

1 Computing **1**
- 1.1 Processes, Procedures, and Computers . . . 2
- 1.2 Measuring Computing Power . . . 3
  - 1.2.1 Information . . . 3
  - 1.2.2 Representing Data . . . 8
  - 1.2.3 Growth of Computing Power . . . 12
- 1.3 Science, Engineering, and the Liberal Arts . . . 13
- 1.4 Summary and Roadmap . . . 16

Part I: Defining Procedures

2 Language **19**
- 2.1 Surface Forms and Meanings . . . 19
- 2.2 Language Construction . . . 20
- 2.3 Recursive Transition Networks . . . 22
- 2.4 Replacement Grammars . . . 26
- 2.5 Summary . . . 32

3 Programming **35**
- 3.1 Problems with Natural Languages . . . 36
- 3.2 Programming Languages . . . 37
- 3.3 Scheme . . . 39
- 3.4 Expressions . . . 40
  - 3.4.1 Primitives . . . 40
  - 3.4.2 Application Expressions . . . 41
- 3.5 Definitions . . . 44
- 3.6 Procedures . . . 45
  - 3.6.1 Making Procedures . . . 45
  - 3.6.2 Substitution Model of Evaluation . . . 46
- 3.7 Decisions . . . 48
- 3.8 Evaluation Rules . . . 50
- 3.9 Summary . . . 52

4 Problems and Procedures **53**
- 4.1 Solving Problems . . . 53
- 4.2 Composing Procedures . . . 54
  - 4.2.1 Procedures as Inputs and Outputs . . . 55
- 4.3 Recursive Problem Solving . . . 56
- 4.4 Evaluating Recursive Applications . . . 64
- 4.5 Developing Complex Programs . . . 67
  - 4.5.1 Printing . . . 68

4.5.2 Tracing . . . 69
4.6 Summary . . . 73

5 Data 75
5.1 Types . . . 75
5.2 Pairs . . . 77
5.2.1 Making Pairs . . . 79
5.2.2 Triples to Octuples . . . 80
5.3 Lists . . . 81
5.4 List Procedures . . . 83
5.4.1 Procedures that Examine Lists . . . 83
5.4.2 Generic Accumulators . . . 84
5.4.3 Procedures that Construct Lists . . . 86
5.5 Lists of Lists . . . 90
5.6 Data Abstraction . . . 92
5.7 Summary of Part I . . . 102

Part II: Analyzing Procedures

6 Machines 105
6.1 History of Computing Machines . . . 106
6.2 Mechanizing Logic . . . 108
6.2.1 Implementing Logic . . . 109
6.2.2 Composing Operations . . . 111
6.2.3 Arithmetic . . . 114
6.3 Modeling Computing . . . 116
6.3.1 Turing Machines . . . 118
6.4 Summary . . . 123

7 Cost 125
7.1 Empirical Measurements . . . 125
7.2 Orders of Growth . . . 129
7.2.1 Big *O* . . . 130
7.2.2 Omega . . . 133
7.2.3 Theta . . . 134
7.3 Analyzing Procedures . . . 136
7.3.1 Input Size . . . 136
7.3.2 Running Time . . . 137
7.3.3 Worst Case Input . . . 138
7.4 Growth Rates . . . 139
7.4.1 No Growth: Constant Time . . . 139
7.4.2 Linear Growth . . . 140
7.4.3 Quadratic Growth . . . 145
7.4.4 Exponential Growth . . . 147
7.4.5 Faster than Exponential Growth . . . 149
7.4.6 Non-terminating Procedures . . . 149
7.5 Summary . . . 149

8 Sorting and Searching 153
8.1 Sorting . . . 153
8.1.1 Best-First Sort . . . 153
8.1.2 Insertion Sort . . . 157

8.1.3 Quicker Sorting 158
8.1.4 Binary Trees 161
8.1.5 Quicksort 166
8.2 Searching 167
8.2.1 Unstructured Search 168
8.2.2 Binary Search 168
8.2.3 Indexed Search 169
8.3 Summary 178

Part III: Improving Expressiveness

9 Mutation 179
9.1 Assignment 179
9.2 Impact of Mutation 181
9.2.1 Names, Places, Frames, and Environments 182
9.2.2 Evaluation Rules with State 183
9.3 Mutable Pairs and Lists 186
9.4 Imperative Programming 188
9.4.1 List Mutators 188
9.4.2 Imperative Control Structures 191
9.5 Summary 193

10 Objects 195
10.1 Packaging Procedures and State 196
10.1.1 Encapsulation 196
10.1.2 Messages 197
10.1.3 Object Terminology 199
10.2 Inheritance 200
10.2.1 Implementing Subclasses 202
10.2.2 Overriding Methods 204
10.3 Object-Oriented Programming 207
10.4 Summary 209

11 Interpreters 211
11.1 Python 212
11.1.1 Python Programs 213
11.1.2 Data Types 216
11.1.3 Applications and Invocations 219
11.1.4 Control Statements 219
11.2 Parser 221
11.3 Evaluator 223
11.3.1 Primitives 223
11.3.2 If Expressions 225
11.3.3 Definitions and Names 226
11.3.4 Procedures 227
11.3.5 Application 228
11.3.6 Finishing the Interpreter 229
11.4 Lazy Evaluation 229
11.4.1 Lazy Interpreter 230
11.4.2 Lazy Programming 232

11.5 Summary 235

Part IV: The Limits of Computing

12 Computability **237**
12.1 Mechanizing Reasoning 237
12.1.1 Gödel's Incompleteness Theorem 240
12.2 The Halting Problem 241
12.3 Universality 244
12.4 Proving Non-Computability 245
12.5 Summary 251

Indexes **253**
Index 253
People 256

List of Explorations

1.1 Guessing Numbers 7
1.2 Twenty Questions 8
2.1 Power of Language Systems 29
4.1 Square Roots 62
4.2 Recipes for π 69
4.3 Recursive Definitions and Games 71
5.1 Pascal's Triangle 91
5.2 Pegboard Puzzle 93
7.1 Multiplying Like Rabbits 127
8.1 Searching the Web 177
12.1 Virus Detection 246
12.2 Busy Beavers 249

List of Figures

1.1 Using three bits to distinguish eight possible values. 6

2.1 Simple recursive transition network. 22
2.2 RTN with a cycle. 23
2.3 Recursive transition network with subnetworks. 24
2.4 Alternate *Noun* subnetwork. 24
2.5 RTN generating "Alice runs". 25
2.6 System power relationships. 30
2.7 Converting the *Number* productions to an RTN. 31
2.8 Converting the *MoreDigits* productions to an RTN. 31
2.9 Converting the *Digit* productions to an RTN. 32

3.1 Running a Scheme program. 39

4.1 A procedure maps inputs to an output. 54
4.2 Composition. 54
4.3 Circular Composition. 57
4.4 Recursive Composition. 58
4.5 Cornering the Queen. 72

5.1 Pegboard Puzzle. 93

6.1 Computing *and* with wine. 110
6.2 Computing logical *or* and *not* with wine 111
6.3 Computing *and3* by composing two *and* functions. 112
6.4 Turing Machine model. 119
6.5 Rules for checking balanced parentheses Turing Machine. 121
6.6 Checking parentheses Turing Machine. 121

7.1 Evaluation of *fibo* procedure. 128
7.2 Visualization of the sets $O(f)$, $\Omega(f)$, and $\Theta(f)$. 130
7.3 Orders of Growth. 131

8.1 Unbalanced trees. 165

9.1 Sample environments. 182
9.2 Environment created to evaluate (*bigger* 3 4). 184
9.3 Environment after evaluating (**define** *inc* (*make-adder* 1)). 185
9.4 Environment for evaluating the body of (*inc* 149). 186
9.5 Mutable pair created by evaluating (*set-mcdr! pair pair*). 187
9.6 MutableList created by evaluating (*mlist* 1 2 3). 187

10.1 Environment produced by evaluating: 197
10.2 Inheritance hierarchy. 201
10.3 Counter class hierarchy. 206

12.1 Incomplete and inconsistent axiomatic systems. 239
12.2 Universal Turing Machine. 245
12.3 Two-state Busy Beaver Machine. 249

Image Credits

Most of the images in the book, including the tiles on the cover, were generated by the author.

Some of the tile images on the cover are from flickr creative commons licenses images from: ell brown, Johnson Cameraface, cogdogblog, Cyberslayer, dmealiffe, Dunechaser, MichaelFitz, Wolfie Fox, glingl, jurvetson, KayVee.INC, michaeldbeavers, and Oneras.

The Van Gogh *Starry Night* image from Section 1.2.2 is from the Google Art Project. The Apollo Guidance Computer image in Section 1.2.3 was released by NASA and is in the public domain. The traffic light in Section 2.1 is from iStockPhoto, and the rotary traffic signal is from the Wikimedia Commons. The picture of Grace Hopper in Chapter 3 is from the Computer History Museum. The playing card images in Chapter 4 are from iStockPhoto. The images of Gauss, Heron, and Grace Hopper's bug are in the public domain. The Dilbert comic in Chapter 4 is licensed from United Feature Syndicate, Inc. The Pascal's triangle image in Excursion 5.1 is from Wikipedia and is in the public domain. The image of Ada Lovelace in Chapter 6 is from the Wikimedia Commons, of a painting by Margaret Carpenter. The odomoter image in Chapter 7 is from iStockPhoto, as is the image of the frustrated student. The Python snake charmer in Section 11.1 is from iStockPhoto. The Dynabook images at the end of Chapter 10 are from Alan Kay's paper. The xkcd comic at the end of Chapter 11 is used under the creative commons license generously provided by Randall Munroe.

Preface

This book started from the premise that Computer Science should be taught as a liberal art, not an industrial skill. I had the privilege of taking 6.001 from Gerry Sussman when I was a first year student at MIT, and that course awakened me to the power and beauty of computing, and inspired me to pursue a career as a teacher and researcher in Computer Science. When I arrived as a new faculty member at the University of Virginia in 1999, I was distraught to discover that the introductory computing courses focused on teaching industrial skills, and with so much of the course time devoted to explaining the technical complexities of using bloated industrial languages like C++ and Java, there was very little, if any, time left to get across the core intellectual ideas that are the essence of computing and the reason everyone should learn it.

With the help of a University Teaching Fellowship and National Science Foundation grants, I developed a new introductory computer science course, targeted especially to students in the College of Arts & Sciences. This course was first offered in Spring 2002, with the help of an extraordinary group of Assistant Coaches. Because of some unreasonable assumptions in the first assignment, half the students quickly dropped the course, but a small, intrepid, group of pioneering students persisted, and it is thanks to their efforts that this book exists. That course, and the next several offerings, used Abelson & Sussman's outstanding *Structure and Interpretation of Computer Programs* (SICP) textbook along with Douglas Hofstadter's *Gödel, Escher, Bach: An Eternal Golden Braid*.

Spring 2002 CS200 Pioneer Graduates
Back row, from left: Portman Wills (*Assistant Coach*), Spencer Stockdale, Shawn O'Hargan, Jeff Taylor, Jacques Fournier, Katie Winstanley, Russell O'Reagan, Victor Clay Yount.
Front: Grace Deng, Rachel Dada, Jon Erdman (*Assistant Coach*).

I am not alone in thinking SICP is perhaps the greatest textbook ever written in any field, so it was with much trepidation that I endeavored to develop a new textbook. I hope the resulting book captures the spirit and fun of computing exemplified by SICP, but better suited to an introductory course for students with no previous background while covering many topics not included in SICP such as languages, complexity analysis, objects, and computability. Although this book is designed around a one semester introductory course, it should also be suitable for self-study students and for people with substantial programming experience but without similar computer science knowledge.

I am indebted to many people who helped develop this course and book. Westley Weimer was the first person to teach using something resembling this book, and his thorough and insightful feedback led to improvements throughout. Greg Humphreys, Paul Reynolds, and Mark Sherriff have also taught versions of this course, and contributed to its development. I am thankful to all of the Assistant Coaches over the years, especially Sarah Bergkuist (2004), Andrew Connors (2004), Rachel Dada (2003), Paul DiOrio (2009), Kinga Dobolyi (2007), Jon Erdman (2002), Ethan Fast (2009), David Faulkner (2005), Jacques Fournier (2003), Richard Hsu (2007), Rachel Lathbury (2009), Michael Lew (2009), Stephen Liang (2002), Dan Marcus (2007), Rachel Rater (2009), Spencer Stockdale (2003), Dan Upton (2005), Portman Wills (2002), Katie Winstanley (2003 and 2004), and Rebecca Zapfel (2009). William Aiello, Anna Chefter, Chris Frost, Jonathan Grier, Thad Hughes, Alan Kay, Tim Koogle, Jerry McGann, Gary McGraw, Radhika Nagpal, Shawn O'Hargan, Mike Peck, and Judith Shatin also made important contributions to the class and book.

My deepest thanks are to my wife, Nora, who is a constant source of inspiration, support, and wonder.

Finally, my thanks to all past, present, and future students who use this book, without whom it would have no purpose.

Happy Computing!

David Evans
Charlottesville, Virginia
August 2011

Spring 2003

Spring 2004

Spring 2005

1 Computing

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.
Edsger Dijkstra, 1972 Turing Award Lecture

The first million years of hominid history produced tools to amplify, and later mechanize, our physical abilities to enable us to move faster, reach higher, and hit harder. We have developed tools that amplify physical force by the trillions and increase the speeds at which we can travel by the thousands.

Tools that amplify intellectual abilities are much rarer. While some animals have developed tools to amplify their physical abilities, only humans have developed tools to substantially amplify our intellectual abilities and it is those advances that have enabled humans to dominate the planet. The first key intellect amplifier was language. Language provided the ability to transmit our thoughts to others, as well as to use our own minds more effectively. The next key intellect amplifier was writing, which enabled the storage and transmission of thoughts over time and distance.

Computing is the ultimate mental amplifier—computers can mechanize any intellectual activity we can imagine. Automatic computing radically changes how humans solve problems, and even the kinds of problems we can imagine solving. Computing has changed the world more than any other invention of the past hundred years, and has come to pervade nearly all human endeavors. Yet, we are just at the beginning of the computing revolution; today's computing offers just a glimpse of the potential impact of computing.

There are two reasons why everyone should study computing:

1. Nearly all of the most exciting and important technologies, arts, and sciences of today and tomorrow are driven by computing.
2. Understanding computing illuminates deep insights and questions into the nature of our minds, our culture, and our universe.

It may be true that you have to be able to read in order to fill out forms at the DMV, but that's not why we teach children to read. We teach them to read for the higher purpose of allowing them access to beautiful and meaningful ideas.
Paul Lockhart,
Lockhart's Lament

Anyone who has submitted a query to Google, watched *Toy Story*, had LASIK eye surgery, used a smartphone, seen a Cirque Du Soleil show, shopped with a credit card, or microwaved a pizza should be convinced of the first reason. None of these would be possible without the tremendous advances in computing over the past half century.

Although this book will touch on on some exciting applications of computing, our primary focus is on the second reason, which may seem more surprising.

Computing changes how we think about problems and how we understand the world. The goal of this book is to teach you that new way of thinking.

1.1 Processes, Procedures, and Computers

information processes

Computer science is the study of *information processes*. A process is a sequence of steps. Each step changes the state of the world in some small way, and the result of all the steps produces some goal state. For example, baking a cake, mailing a letter, and planting a tree are all processes. Because they involve physical things like sugar and dirt, however, they are not pure information processes. Computer science focuses on processes that involve abstract information rather than physical things.

The boundaries between the physical world and pure information processes, however, are often fuzzy. Real computers operate in the physical world: they obtain input through physical means (e.g., a user pressing a key on a keyboard that produces an electrical impulse), and produce physical outputs (e.g., an image displayed on a screen). By focusing on abstract information, instead of the physical ways of representing and manipulating information, we simplify computation to its essence to better enable understanding and reasoning.

procedure

algorithm

A *procedure* is a description of a process. A simple process can be described just by listing the steps. The list of steps is the procedure; the act of following them is the process. A procedure that can be followed without any thought is called a *mechanical procedure*. An *algorithm* is a mechanical procedure that is guaranteed to eventually finish.

A mathematician is a machine for turning coffee into theorems.
Attributed to Paul Erdös

For example, here is a procedure for making coffee, adapted from the actual directions that come with a major coffeemaker:

1. Lift and open the coffeemaker lid.
2. Place a basket-type filter into the filter basket.
3. Add the desired amount of coffee and shake to level the coffee.
4. Fill the decanter with cold, fresh water to the desired capacity.
5. Pour the water into the water reservoir.
6. Close the lid.
7. Place the empty decanter on the warming plate.
8. Press the ON button.

Describing processes by just listing steps like this has many limitations. First, natural languages are very imprecise and ambiguous. Following the steps correctly requires knowing lots of unstated assumptions. For example, step three assumes the operator understands the difference between coffee grounds and finished coffee, and can infer that this use of “coffee” refers to coffee grounds since the end goal of this process is to make drinkable coffee. Other steps assume the coffeemaker is plugged in and sitting on a flat surface.

One could, of course, add lots more details to our procedure and make the language more precise than this. Even when a lot of effort is put into writing precisely and clearly, however, natural languages such as English are inherently ambiguous. This is why the United States tax code is 3.4 million words long, but lawyers can still spend years arguing over what it really means.

Another problem with this way of describing a procedure is that the size of the description is proportional to the number of steps in the process. This is fine for simple processes that can be executed by humans in a reasonable amount of time, but the processes we want to execute on computers involve trillions of steps. This means we need more efficient ways to describe them than just listing each step one-by-one.

To program computers, we need tools that allow us to describe processes precisely and succinctly. Since the procedures are carried out by a machine, every step needs to be described; we cannot rely on the operator having "common sense" (for example, to know how to fill the coffeemaker with water without explaining that water comes from a faucet, and how to turn the faucet on). Instead, we need mechanical procedures that can be followed without any thinking.

A *computer* is a machine that can:

computer

1. Accept input. Input could be entered by a human typing at a keyboard, received over a network, or provided automatically by sensors attached to the computer.
2. Execute a mechanical procedure, that is, a procedure where each step can be executed without any thought.
3. Produce output. Output could be data displayed to a human, but it could also be anything that effects the world outside the computer such as electrical signals that control how a device operates.

A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and body can connect with the universe and move bits of it about.
Douglas Adams

Computers exist in a wide range of forms, and thousands of computers are hidden in devices we use everyday but don't think of as computers such as cars, phones, TVs, microwave ovens, and access cards. Our primary focus is on *universal computers*, which are computers that can perform *all* possible mechanical computations on discrete inputs except for practical limits on space and time. The next section explains what it discrete inputs means; Chapters 6 and 12 explore more deeply what it means for a computer to be universal.

1.2 Measuring Computing Power

For physical machines, we can compare the power of different machines by measuring the amount of mechanical work they can perform within a given amount of time. This power can be captured with units like *horsepower* and *watt*. Physical power is not a very useful measure of computing power, though, since the amount of computing achieved for the same amount of energy varies greatly. Energy is consumed when a computer operates, but consuming energy is not the purpose of using a computer.

Two properties that measure the power of a computing machine are:

1. *How much information* it can process?
2. *How fast* can it process?

We defer considering the second property until Part II, but consider the first question here.

1.2.1 Information

Informally, we use *information* to mean knowledge. But to understand information quantitatively, as something we can measure, we need a more precise way to think about information.

information

The way computer scientists measure information is based on how what is known changes as a result of obtaining the information. The primary unit of information is a *bit*. *One bit* of information *halves* the amount of uncertainty. It is equivalent to answering a “yes” or “no” question, where either answer is equally likely beforehand. Before learning the answer, there were two possibilities; after learning the answer, there is one.

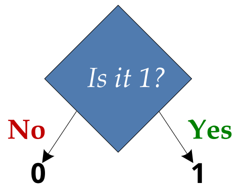
bit

binary question

We call a question with two possible answers a *binary question*. Since a bit can have two possible values, we often represent the values as **0** and 1.

For example, suppose we perform a fair coin toss but do not reveal the result. Half of the time, the coin will land “heads”, and the other half of the time the coin will land “tails”. Without knowing any more information, our chances of guessing the correct answer are $\frac{1}{2}$. One bit of information would be enough to convey either “heads” or “tails”; we can use **0** to represent “heads” and 1 to represent “tails”. So, the amount of information in a coin toss is one bit.

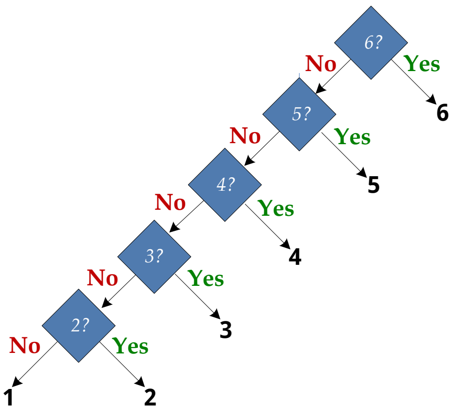
Similarly, one bit can distinguish between the values 0 and 1:



Example 1.1: Dice

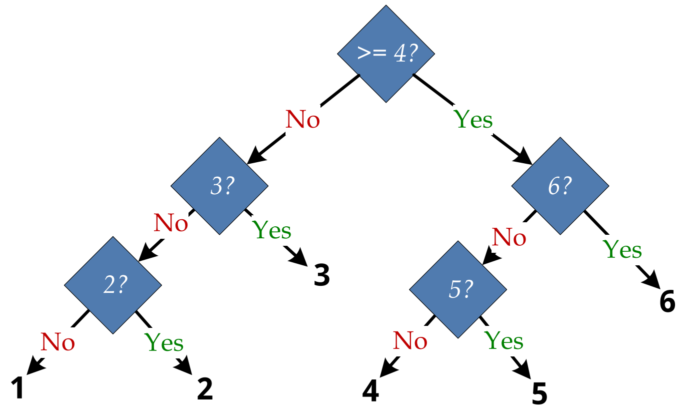
How many bits of information are there in the outcome of tossing a six-sided die?

There are six equally likely possible outcomes, so without any more information we have a one in six chance of guessing the correct value. One bit is not enough to identify the actual number, since one bit can only distinguish between two values. We could use five binary questions like this:



This is quite inefficient, though, since we need up to five questions to identify the value (and on average, expect to need $3\frac{1}{3}$ questions). Can we identify the value with fewer than 5 questions?

Our goal is to identify questions where the "yes" and "no" answers are equally likely—that way, each answer provides the most information possible. This is not the case if we start with, "Is the value 6?", since that answer is expected to be "yes" only one time in six. Instead, we should start with a question like, "Is the value at least 4?". Here, we expect the answer to be "yes" one half of the time, and the "yes" and "no" answers are equally likely. If the answer is "yes", we know the result is 4, 5, or 6. With two more bits, we can distinguish between these three values (note that two bits is actually enough to distinguish among *four* different values, so some information is wasted here). Similarly, if the answer to the first question is no, we know the result is 1, 2, or 3. We need two more bits to distinguish which of the three values it is. Thus, with three bits, we can distinguish all six possible outcomes.



Three bits can convey more information that just six possible outcomes, however. In the binary question tree, there are some questions where the answer is not equally likely to be "yes" and "no" (for example, we expect the answer to "Is the value 3?" to be "yes" only one out of three times). Hence, we are not obtaining a full bit of information with each question.

Each bit doubles the number of possibilities we can distinguish, so with three bits we can distinguish between $2 * 2 * 2 = 8$ possibilities. In general, with n bits, we can distinguish between 2^n possibilities. Conversely, distinguishing among k possible values requires $\log_2 k$ bits. The *logarithm* is defined such that if $a = b^c$ then $\log_b a = c$. Since each bit has two possibilities, we use the logarithm base 2 to determine the number of bits needed to distinguish among a set of distinct possibilities. For our six-sided die, $\log_2 6 \approx 2.58$, so we need approximately 2.58 binary questions. But, questions are discrete: we can't ask 0.58 of a question, so we need to use three binary questions.

logarithm

Trees. Figure 1.1 depicts a structure of binary questions for distinguishing among eight values. We call this structure a *binary tree*. We will see many useful applications of tree-like structures in this book.

binary tree

Computer scientists draw trees upside down. The *root* is the top of the tree, and the *leaves* are the numbers at the bottom (0, 1, 2, ..., 7). There is a unique path from the root of the tree to each leaf. Thus, we can describe each of the eight possible values using the answers to the questions down the tree. For example, if the answers are “No”, “No”, and “No”, we reach the leaf 0; if the answers are “Yes”, “No”, “Yes”, we reach the leaf 5. Since there are no more than two possible answers for each node, we call this a *binary* tree.

We can describe any non-negative integer using bits in this way, by just adding additional levels to the tree. For example, if we wanted to distinguish between 16 possible numbers, we would add a new question, “Is is $>=$ 8?” to the top of the tree. If the answer is “No”, we use the tree in Figure 1.1 to distinguish numbers between 0 and 7. If the answer is “Yes”, we use a tree similar to the one in Figure 1.1, but add 8 to each of the numbers in the questions and the leaves.

depth The *depth* of a tree is the length of the longest path from the root to any leaf. The example tree has depth three. A binary tree of depth d can distinguish up to 2^d different values.

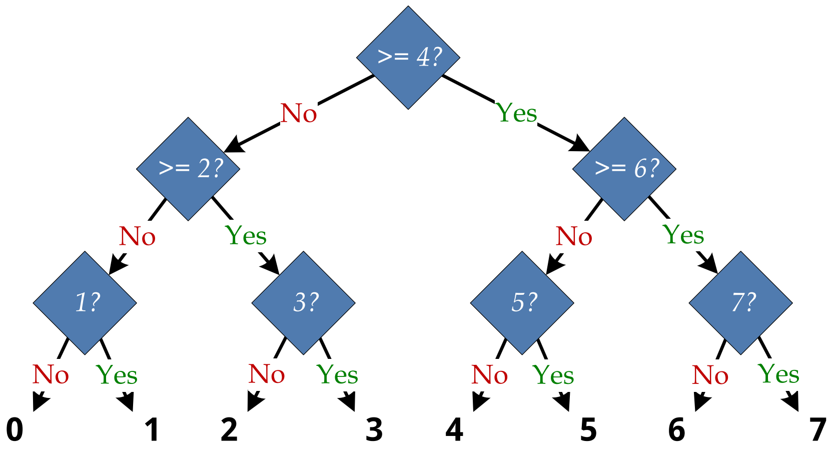


Figure 1.1. Using three bits to distinguish eight possible values.

Units of Information. One *byte* is defined as eight bits. Hence, one byte of information corresponds to eight binary questions, and can distinguish among 2^8 (256) different values. For larger amounts of information, we use metric prefixes, but instead of scaling by factors of 1000 they scale by factors of 2^{10} (1024). Hence, one *kilobyte* is 1024 bytes; one *megabyte* is 2^{20} (approximately one million) bytes; one *gigabyte* is 2^{30} (approximately one billion) bytes; and one *terabyte* is 2^{40} (approximately one trillion) bytes.

Exercise 1.1. Draw a binary tree with the minimum possible depth to:

a. Distinguish among the numbers $0, 1, 2, \ldots, 15$.

b. Distinguish among the 12 months of the year.

Exercise 1.2. How many bits are needed:

a. To uniquely identify any currently living human?

b. To uniquely identify any human who ever lived?

c. To identify any location on Earth within one square centimeter?

d. To uniquely identify any atom in the observable universe?

Exercise 1.3. The examples all use binary questions for which there are two possible answers. Suppose instead of basing our decisions on bits, we based it on *trits* where one trit can distinguish between three equally likely values. For each trit, we can ask a ternary question (a question with three possible answers).

a. How many trits are needed to distinguish among eight possible values? (A convincing answer would show a ternary tree with the questions and answers for each node, and argue why it is not possible to distinguish all the values with a tree of lesser depth.)

b. [⋆] Devise a general formula for converting between bits and trits. How many trits does it require to describe b bits of information?

Exploration 1.1: Guessing Numbers

The guess-a-number game starts with one player (the *chooser*) picking a number between 1 and 100 (inclusive) and secretly writing it down. The other player (the *guesser*) attempts to guess the number. After each guess, the chooser responds with "correct" (the guesser guessed the number and the game is over), "higher" (the actual number is higher than the guess), or "lower" (the actual number is lower than the guess).

a. Explain why the guesser can receive slightly more than one bit of information for each response.

b. Assuming the chooser picks the number randomly (that is, all values between 1 and 100 are equally likely), what are the best first guesses? Explain why these guesses are better than any other guess. (Hint: there are two equally good first guesses.)

c. What is the maximum number of guesses the second player should need to always find the number?

d. What is the average number of guesses needed (assuming the chooser picks the number randomly as before)?

e. [⋆] Suppose instead of picking randomly, the chooser picks the number with the goal of maximizing the number of guesses the second player will need. What number should she pick?

f. [⋆⋆] How should the guesser adjust her strategy if she knows the chooser is picking adversarially?

g. [⋆⋆] What are the best strategies for both players in the adversarial guess-a-number game where chooser's goal is to pick a starting number that maximizes the number of guesses the guesser needs, and the guesser's goal is to guess the number using as few guesses as possible.