Computer science: an interdisciplinary approach pdf free pdf download

Computer science: an interdisciplinary approach pdf free pdf download

[illegible]

specify an int literal +99 99 with a sequence of the decimal digits 0 through positive sign 9 (that, when interpreted as decimal numbers, -99 -99 negative sign fall within the defined range). We use ints fre5 + 3 8 addition quently because they naturally arise when we are 5 - 3 2 subtraction implementing programs. 5 * 3 15 multiplication Standard arithmetic operators for addi5 / 3 1 no fractional part tion/subtraction (+ and -), multiplication (*), 5 % 3 2 remainder division (/), and remainder (%) for the int data 1 / 0 run-time error type are built into Java. These operators take two 3 * 5 - 2 13 * has precedence int operands and produce an int result, with one significant exception—division or remain3 + 5 / 2 5 / has precedence der by zero is not allowed. These operations are 3 - 5 - 2 -4 left associative defined as in grade school (keeping in mind that (3 - 5) - 2 -4 better style all results must be integers): given two int val- 3 - (5 - 2) 0 unambiguous ues a and b, the value of a / b is the number of Typical int expressions times b goes into a with the fractional part discarded, and the value of a % b is the remainder that you get when you divide a by b. For example, the value of 17 / 3 is 5, and the value of 17 % 3 is 2. The int results that we get from arithmetic operations are just what we expect, except that if the result is too large to fit into int's 32-bit representation, then it will be truncated in a well-defined manner. This situation is known int 31 integers between 2 31 and 2 31 1 values 1234 typical literals 99 0 1000000 operations sign add subtract multiply divide remainder operators + - + - * / % Java's built-in int data type 1.2 Built-in Types of Data 23 Program 1.2.2 Integer multiplication and division public class IntOps { public static void main(String[] args) { int a = Integer.parseInt(args[0]); int b = Integer.parseInt(args[1]); int p = a * b; int q = a / b; int r = a % b; System.out.println(a + " * " + b + " System.out.println(a + " / " + b + " System.out.println(a + " % " + b + " System.out.println(a + " = " + q + " } } = = = * " " " " + + + + p); q); r); b + " + r); Arithmetic for integers is built into Java. Most of this code is devoted to the task of getting the values in and out; the actual arithmetic is in the simple statements in the middle of the program that assign values to p, q, and r. % javac IntOps.java % java IntOps 1234 99 1234 * 99 = 122166 1234 / 99 = 12 1234 % 99 = 46 1234 = 12 * 99 + 46 as overflow. In general, we have to take care that such a result is not misinterpreted by our code. For the moment, we will be computing with small numbers, so you do not have to worry about these boundary conditions. Program 1.2.2 illustrates three basic operations (multiplication, division, and remainder) for manipulating integers,. It also demonstrates the use of Integer.parseInt() to convert String values on the command line to int values, as well as the use of automatic type conversion to convert int values to String values for output. Elements of Programming 24 Three other built-in types are different representations of integers in Java. The long, short, and byte types are the same as int except that they use 64, 16, and 8 bits respectively, so the range of allowed values is accordingly different. Programmers use long when working with huge integers, and the other types to save space. You can find a table with the maximum and minimum values for each type on the booksite, or you can figure them out for yourself from the numbers of bits. Floating-point numbers The double type represents floating-point numbers, for use in scientific and commercial applications. The internal representation is like scientific notation, so that we can compute with numbers in a huge range. We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can represent only a finite number of floatingexpression value point numbers in any digital computer 3.141 + 2.0 5.141 representation. Floating-point numbers 3.141 2.0 1.111 approximate real numbers sufficiently 3.141 / 2.0 1.5705 well that we can use them in applications, 5.0 / 3.0 1.6666666666666667 but we often need to cope with the fact that we cannot always do exact computations. 10.0 % 3.141 0.577 You can specify a double literal with 1.0 / 0.0 Infinity a sequence of digits with a decimal point. Math.sqrt(2.0) 1.4142135623730951 For example, the literal 3.14159 represents Math.sqrt(-1.0) NaN a six-digit approximation to . AlternaTypical double expressions tively, you specify a double literal with a notation like scientific notation: the literal 6.022e23 represents the number 6.022 1023. As with integers, you can use these conventions to type floating-point literals in your programs or to provide floatingpoint numbers as string arguments on the command line. The arithmetic operators +, -, *, and / are defined for double. Beyond these built-in operators, the Java Math library defines the square root function, trigonometric functions, logarithm/exponential functions, and other common functions for floating-point numbers. To use one of these functions in an expression, you type the name of the function followed by its argument in parentheses. For exvalues typical literals real numbers (specified by IEEE 754 standard) 3.14159 6.022e23 2.0 1.4142135623730951 operations add subtract multiply divide operators + - * / Java's built-in double data type 1.2 Built-in Types of Data 25 Program 1.2.3 Quadratic formula public class Quadratic { public static void main(String[] args) { double b = Double.parseDouble(args[0]); double c = Double.parseDouble(args[1]); double discriminant = b*b - 4.0*c; double d = Math.sqrt(discriminant); System.out.println((-b + d) / 2.0); System.out.println((-b - d) / 2.0); } } This program prints the roots of the polynomial x2 + bx + c, using the quadratic formula. For example, the roots of x2 – 3x + 2 are 1 and 2 since we can factor the equation as (x – 1)(x – 2); the roots of x2 – x – 1 are and 1 – , where is the golden ratio; and the roots of x2 + x + 1 are not real numbers. % javac Quadratic.java % java Quadratic -3.0 2.0 2.0 1.0 % java Quadratic -1.0 -1.0 1.618033988749895 -0.6180339887498949 % java Quadratic 1.0 1.0 NaN NaN ample, the code Math.sqrt(2.0) evaluates to a double value that is approximately the square root of 2. We discuss the mechanism behind this arrangement in more detail in Section 2.1 and more details about the Math library at the end of this section. When working with floating-point numbers, one of the first things that you will encounter is the issue of precision. For example, printing 5.0/2.0 results in 2.5 as expected, but printing 5.0/3.0 results in 1.6666666666666667. In Section 1.5, you will learn Java's mechanism for controlling the number of significant digits that you see in output. Until then, we will work with the Java default output format. Elements of Programming 26 The result of a calculation can be one of the special values Infinity (if the number is too large to be represented) or NaN (if the result of the calculation is undefined). Though there are myriad details to consider when calculations involve these values, you can use double in a natural way and begin to write Java programs instead of using a calculator for all kinds of calculations. For example, Program 1.2.3 shows the use of double values in computing the roots of a quadratic equation using the quadratic formula. Several of the exercises at the end of this section further illustrate this point. As with long, short, and byte for integers, there is another representation for real numbers called float. Programmers sometimes use float to save space when precision is a secondary consideration. The double type is useful for about 15 significant digits; the float type is good for only about 7 digits. We do not use float in this book. Booleans The type represents truth valvalues true or false ues from logic. It has just two values: true and false. literals true false These are also the two possible boolean literals. Every operations and or not boolean variable has one of these two values, and evoperators && || ! ery boolean operation has operands and a result that takes on just one of these two values. This simplicity Java's built-in boolean data type is deceiving—boolean values lie at the foundation of computer science. The most important operations defined for booleans are and (&&), or (||), and not (!), which have familiar definitions: • a && b is true if both operands are true, and false if either is false. • a || b is false if both operands are false, and true if either is true. • !a is true if a is false, and false if a is true. Despite the intuitive nature of these definitions, it is worthwhile to fully specify each possibility for each operation in tables known as truth tables. The not function has only one operand: its value for each of the two possible values of the operand is boolean a !a a b a && b a || b true false false false false false true false true false true true false false true true true true true Truth-table definitions of boolean operations 1.2 Built-in Types of Data 27 specified in the second column. The and and or functions each have two operands: there are four different possibilities for operand values, and the values of the functions for each possibility are specified in the right two columns. We can use these operators with parentheses to develop arbitrarily complex expressions, each of which specifies a well-defined boolean function. Often the same function appears in different guises. For example, the expressions (a && b) and !(!a || !b) are equivalent. a b a && b !a !b !a || !b !(!a || !b) false false false true true true false false true false true false true false true false false false true true false true true true false false false true Truth-table proof that a && b and !(!a || !b) are identical The study of manipulating expressions of this kind is known as Boolean logic. This field of mathematics is fundamental to computing: it plays an essential role in the design and operation of computer hardware itself, and it is also a starting point for the theoretical foundations of computation. In the present context, we are interested in boolean expressions because we use them to control the behavior of our programs. Typically, a particular condition of interest is specified as a boolean expression, and a piece of program code is written to execute one set of statements if that expression is true and a different set of statements if the expression is false. The mechanics of doing so are the topic of Section 1.3. Comparisons Some mixed-type operators take operands of one type and produce a result of another type. The most important operators of this kind are the comparison operators ==, !=, =, which all are defined for each primitive numeric type and produce a boolean result. Since operations are defined only with respect to data types, each of these symbols stands for many operations, one for each data type. It is required that both operands be of the same type. non-negative discriminant? (b*b - 4.0*a*c) >= 0.0 beginning of a century? (year % 100) == 0 legal month? (month >= 1) && (month = 1 && month = 2 2 >= 3 >= greater than or equal in Section 1.3 we will see the role that boolean expressions play in more soComparisons with int operands and a boolean result phisticated programs. Library methods and APIs As we have seen, many programming tasks involve using Java library methods in addition to the built-in operators. The number of available library methods is vast. As you learn to program, you will learn to use more and more library methods, but it is best at the beginning to restrict your attention to a relatively small set of methods. In this chapter, you have already used some of Java's methods for printing, for converting data from one type to another, and for computing mathematical functions (the Java Math library). In later chapters, you will learn not just how to use other methods, but how to create and use your own methods. For convenience, we will consistently summarize the library methods that you need to know how to use in tables like this one: void System.out.print(String s) void System.out.println(String s) void System.out.println() print s print s, followed by a newline print a newline Note: Any type of data can be used as argument (and will be automatically converted to String). Java library methods for printing strings to the terminal Elements of Programming 30 Such a table is known as an application programming interface (API). Each method is described by a line in the API public class Math that specifies the information you need to know to use the . . . method name method. The code in the tables is not the code that you type signature to use the method; it is known as the method's signature. double sqrt(double a) The signature specifies the type of the arguments, the methreturn type argument type od name, and the type of the result that the method computes (the return value). . . . In your code, you can call a method by typing its name Anatomy of a method signature followed by arguments, enclosed in parentheses and separated by commas. When Java executes your program, we say that it calls (or evaluates) the method with the given arguments and that the method returns a value. A method call is an expression, so you can use a method call in library name method name the same way that you use variables and literdouble d = Math.sqrt(b*b - 4.0*a*c); als to build up more complicated expressions. return type argument For example, you can write expressions like Math.sin(x) * Math.cos(y) and so on. An Using a library method argument is also an expression, so you can write code like Math.sqrt(b*b - 4.0*a*c) and Java knows what you mean—it evaluates the argument expression and passes the resulting value to the method. The API tables on the facing page show some of the commonly used methods in Java's Math library, along with the Java methods we have seen for printing text to the terminal window and for converting strings to primitive types. The following table shows several examples of calls that use these library methods: library name method call library return type value Integer.parseInt("123") Integer int 123 Double.parseDouble("1.5") Double double 1.5 Math.sqrt(5.0*5.0 - 4.0*4.0) Math double 3.0 Math.log(Math.E) Math double 1.0 Math.random() Math double random in [0, 1) Math.round(3.14159) Math long 3 Math.max(1.0, 9.0) Math double 9.0 Typical calls to Java library methods 1.2 Built-in Types of Data 31 public class Math double abs(double a) absolute value of a double max(double a, double b) maximum of a and b double min(double a, double b) minimum of a and b Note 1: abs(), max(), and min() are defined also for int, long, and float. double sin(double theta) sine of theta double cos(double theta) cosine of theta double tan(double theta) tangent of theta Note 2: Angles are expressed in radians. Use toDegrees() and toRadians() to convert. Note 3: Use asin(), acos(), and atan() for inverse functions. double exp(double a) exponential (e a) double log(double a) natural log (loge a, or ln a) double pow(double a, double b) raise a to the bth power (ab) long round(double a) round a to the nearest integer double random() random number in [0, 1) double sqrt(double a) square root of a double E value of e (constant) double PI value of (constant) See booksite for other available functions. Excerpts from Java's Math library void System.out.print(String s) void System.out.println(String s) print s print s, followed by a newline void System.out.println() print a newline Java library methods for printing strings to the terminal int Integer.parseInt(String s) double Double.parseDouble(String s) long Long.parseLong(String s) convert s to an int value convert s to a double value convert s to a long value Java library methods for converting strings to primitive types 32 Elements of Programming With three exceptions, the methods on the previous page are pure—given the same arguments, they always return the same value, without producing any observable side effect. The method Math.random() is impure because it returns potentially a different value each time it is called; the methods System.out.print() and System.out.println() are impure because they produce side effects—printing strings to the terminal. In APIs, we use a verb phrase to describe the behavior of a method that produces side effects; otherwise, we use a noun phrase to describe the return value. The keyword void designates a method that does not return a value (and whose main purpose is to produce side effects). The Math library also defines the constant values Math.PI (for) and Math.E (for e), which you can use in your programs. For example, the value of Math.sin(Math.PI/2) is 1.0 and the value of Math.log(Math.E) is 1.0 (because Math.sin() takes its argument in radians and Math.log() implements the natural logarithm function). These APIs are typical of the online documentation that is the standard in modern programming. The extensive online documentation of the Java APIs is routinely used by professional programmers, and it is available to you (if you are interested) directly from the Java website or through our booksite. You do not need to go to the online documentation to understand the code in this book or to write similar code, because we present and explain in the text all of the library methods that we use in APIs like these and summarize them in the endpapers. More important, in Chapters 2 and 3 you will learn in this book how to develop your own APIs and to implement methods for your own use. Type conversion One of the primary rules of modern programming is that you should always be aware of the type of data that your program is processing. Only by knowing the type can you know precisely which set of values each variable can have, which literals you can use, and which operations you can perform. For example, suppose that you wish to compute the average of the four integers 1, 2, 3, and 4. Naturally, the expression (1 + 2 + 3 + 4) / 4 comes to mind, but it produces the int value 2 instead of the double value 2.5 because of type conversion conventions. The problem stems from the fact that the operands are int values but it is natural to expect a double value for the result, so conversion from int to double is necessary at some point. There are several ways to do so in Java. 1.2 Built-in Types of Data 33 Implicit type conversion. You can use an int value wherever a double value is expected, because Java automatically converts integers to doubles when appropriate. For example, 11*0.25 evaluates to 2.75 because 0.25 is a double and both operands need to be of the same type; thus, 11 is converted to a double and then the result of dividing two doubles is a double. As another example, Math.sqrt(4) evaluates to 2.0 because 4 is converted to a double, as expected by Math.sqrt(), which then returns a double value. This kind of conversion is called automatic promotion or coercion. Automatic promotion is appropriate because your intent is clear and it can be done with no loss of information. In contrast, a conversion that might involve loss of information (for example, assigning a double value to an int variable) leads to a compile-time error. Explicit cast. Java has some built-in type conversion conventions for primitive types that you can take advantage of when you are aware that you might lose information. You have to make your intention to do so explicit by using a device expression expression type value called a cast. You cast an expression from one primitive type to another (1 + 2 + 3 + 4) / 4.0 double 2.5 by prepending the desired type name Math.sqrt(4) double 2.0 within parentheses. For example, the "1234" + 99 String "123499" expression (int) 2.71828 is a cast 11 * 0.25 double 2.75 from double to int that produces (int) 11 * 0.25 double 2.75 an int with value 2. The conversion 11 * (int) 0.25 int 0 methods defined for casts throw away (int) (11 * 0.25) int 2 information in a reasonable way (for a (int) 2.71828 int 2 full list, see the booksite). For example, Math.round(2.71828) long 3 casting a floating-point number to (int) Math.round(2.71828) int 3 an integer discards the fractional part Integer.parseInt("1234") int 1234 by rounding toward zero. RandomInt Typical type conversions (Program 1.2.5) is an example that uses a cast for a practical computation. Casting has higher precedence than arithmetic operations—any cast is applied to the value that immediately follows it. For example, if we write int value = (int) 11 * 0.25, the cast is no help: the literal 11 is already an integer, so the cast (int) has no effect. In this example, the compiler produces a possible loss of precision error message because there would be a loss Elements of Programming 34 Program 1.2.5 Casting to get a random integer public class RandomInt { public static void main(String[] args) { int n = Integer.parseInt(args[0]); double r = Math.random(); // uniform between 0.0 and 1.0 int value = (int) (r * n); // uniform between 0 and n-1 System.out.println(value); } } This program uses the Java method Math.random() to generate a random number r between 0.0 (inclusive) and 1.0 (exclusive); then multiplies r by the command-line argument n to get a random number greater than or equal to 0 and less than n; then uses a cast to truncate the result to be an integer value between 0 and n-1. % javac RandomInt.java % java RandomInt 1000 548 % java RandomInt 1000 141 % java RandomInt 1000000 135032 of precision in converting the resulting value (2.75) to an int for assignment to value. The error is helpful because the intended computation for this code is likely (int) (11 * 0.25), which has the value 2, not 2.75. Explicit type conversion. You can use a method that takes an argument of one type (the value to be converted) and produces a result of another type. We have already used the Integer.parseInt() and Double.parseDouble() library methods to convert String values to int and double values, respectively. Many other methods are available for conversion among other types. For example, the library 1.2 Built-in Types of Data 35 method Math.round() takes a double argument and returns a long result: the nearest integer to the argument. Thus, for example, Math.round(3.14159) and Math.round(2.71828) are both of type long and have the same value (3). If you want to convert the result of Math.round() to an int, you must use an explicit cast. Beginning programmers tend to find type conversion to be an annoyance, but experienced programmers know that paying careful attention to data types is a key to success in programming. It may also be a key to avoiding failure: in a famous incident in 1985, a French rocket exploded in midair because of a type-conversion problem. While a bug in your program may not cause an explosion, it is well worth your while to take the time to understand what type conversion is all about. After you have written just a few programs, you will see that an understanding of data types will help you not only compose compact code but also make your intentions explicit and avoid subtle bugs in your programs. Photo: ESA Explosion of Ariane 5 rocket Summary A data type is a set of values and a set of operations on those values. Java has eight primitive data types: boolean, char, byte, short, int, long, float, and double. In Java code, we use operators and expressions like those in familiar mathematical expressions to invoke the operations associated with each type. The boolean type is used for computing with the logical values true and false; the char type is the set of character values that we type; and the other six numeric types are used for computing with numbers. In this book, we most often use boolean, int, and double; we do not use short or float. Another data type that we use frequently, String, is not primitive, but Java has some built-in facilities for Strings that are like those for primitive types. When programming in Java, we have to be aware that every operation is defined only in the context of its data type (so we may need type conversions) and that all types can have only a finite number of values (so we may need to live with imprecise results). The boolean type and its operations—&&, ||, and !—are the basis for logical decision making in Java programs, when used in conjunction with the mixed-type comparison operators ==, !=, , =. Specifically, we use boolean expressions to control Java's conditional (if) and loop (for and while) constructs, which we will study in detail in the next section. 36 Elements of Programming The numeric types and Java's libraries give us the ability to use Java as an extensive mathematical calculator. We write arithmetic expressions using the built-in operators +, -, *, /, and % along with Java methods from the Math library. Although the programs in this section are quite rudimentary by the standards of what we will be able to do after the next section, this class of programs is quite useful in its own right. You will use primitive types and basic mathematical functions extensively in Java programming, so the effort that you spend now in understanding them will certainly be worthwhile. 1.2 Built-in Types of Data Q&A (Strings) Q. How does Java store strings internally? A. Strings are sequences of characters that are encoded with Unicode, a modern standard for encoding text. Unicode supports more than 100,000 different characters, including more than 100 different languages plus mathematical and musical symbols. Q. Can you use < and > to compare String values? A. No. Those operators are defined only for primitive-type values. Q. How about == and != ? A. Yes, but the result may not be what you expect, because of the meanings these operators have for nonprimitive types. For example, there is a distinction between a String and its value. The expression "abc" == "ab" + x is false when x is a String with value "c" because the two operands are stored in different places in memory (even though they have the same value). This distinction is essential, as you will learn when we discuss it in more detail in Section 3.1. Q. How can I compare two strings like words in a book index or dictionary? A. We defer discussion of the String data type and associated methods until Section 3.1, where we introduce object-oriented programming. Until then, the string concatenation operation suffices. Q. How can I specify a string literal that is too long to fit on a single line? A. You can't. Instead, divide the string literal into independent string literals and concatenate them together, as in the following example: String dna = "ATGCGCCCACAGCTGCGTCTAAACCGGACTCTG" + "AAGTCCGGAAATTACACCTGTTAG"; 37 Elements of Programming 38 Q&A (Integers) Q. How does Java store integers internally? A. The simplest representation is for small positive integers, where the binary number system is used to represent each integer with a fixed amount of computer memory. Q. What's the binary number system? A. In the binary number system, we represent an integer as a sequence of bits. A bit is a single binary (base 2) digit—either 0 or 1—and is the basis for representing information in computers. In this case the bits are coefficients of powers of 2. Specifically, the sequence of bits bnbn–1...b2b1b0 represents the integer bn2n + bn–12n–1 + … + b222 + b121 + b020 For example, 1100011 represents the integer 99 = 1·64 + 1·32 + 0·16 + 0·8 + 0·4 + 1·2 +1·1 The more familiar decimal number system is the same except that the digits are between 0 and 9 and we use powers of 10. Converting a number to binary is an interesting computational problem that we will consider in the next section. Java uses 32 bits to represent int values. For example, the decimal integer 99 might be represented with the 32 bits 00000000000000000000000001100011. Q. How about negative numbers? A. Negative numbers are handled with a convention known as two's complement, which we need not consider in detail. This is why the range of int values in Java is –2147483648 (–231) to 2147483647 (231 – 1). One surprising consequence of this representation is that int values can become negative when they get large and overflow (exceed 2147483647). If you have not experienced this phenomenon, see Exercise 1.2.10. A safe strategy is to use the int type when you know the integer values will be fewer than ten digits and the long type when you think the integer values might get to be ten digits or more. Q. It seems wrong that Java should just let ints Shouldn't Java automatically check for overflow? overflow and give bad values. 1.2 Built-in Types of Data 39 A. Yes, this issue is a contentious one among programmers. The short answer for now is that the lack of such checking is one reason such types are called primitive data types. A little knowledge can go a long way in avoiding such problems. Again, it is fine to use the int type for small numbers, but when values run into the billions, you cannot. Q. What is the value of Math.abs(-2147483648)? A. -2147483648. This strange (but true) result is a typical example of the effects of integer overflow and two's complement representation. Q. What do the expressions 1 / 0 and 1 % 0 evaluate to in Java? A. Each generates a run-time exception, for division by zero. Q. What is the result of division and remainder for negative integers? A. The quotient a / b rounds toward 0; the remainder a % b is defined such that (a / b) * b + a % b is always equal to a. For example, -14 / 3 and 14 / -3 are both -4, but -14 % 3 is -2 and 14 % -3 is 2. Some other languages (including Python) have different conventions when dividing by negative integers. Q. Why is the value of 10 ^ 6 not 1000000 but 12? A. The ^ operator is not an exponentiation operator, which you must have been thinking. Instead, it is the bitwise exclusive or operator, which is seldom what you want. Instead, you can use the literal 1e6. You could also use Math.pow(10, 6) but doing so is wasteful if you are raising 10 to a known power. Elements of Programming 40 Q&A (Floating-Point Numbers) Q. Why is the type for real numbers named double? A. The decimal point can "float" across the digits that make up the real number. In contrast, with integers the (implicit) decimal point is fixed after the least significant digit. Q. How does Java store floating-point numbers internally? A. Java follows the IEEE 754 standard, which supported in hardware by most modern computer systems. The standard specifies that a floating-point number is stored using three fields: sign, mantissa, and exponent. If you are interested, see the booksite for more details. The IEEE 754 standard also specifies how special floating-point values—positive zero, negative zero, positive infinity, negative infinity, and NaN (not a number)—should be handled. In particular, floatingpoint arithmetic never leads to a run-time exception. For example, the expression -0.0/3.0 evaluates to -0.0, the expression 1.0/0.0 evaluates to positive infinity, and Math.sqrt(-2.0) evaluates to NaN. Q. Fifteen digits for floating-point numbers certainly seems enough to me. Do I really need to worry much about precision? A. Yes, because you are used to mathematics based on real numbers with infinite precision, whereas the computer always deals with finite approximations. For example, the expression (0.1 + 0.1 == 0.2) evaluates to true but the expression (0.1 + 0.1 + 0.1 == 0.3) evaluates to false! Pitfalls like this are not at all unusual in scientific computing. Novice programmers should avoid comparing two floating-point numbers for equality. Q. How can I initialize a double variable to NaN or infinity? A. Java has built-in constants available for this purpose: Double.NaN, Double.POSITIVE_INFINITY, and Double.NEGATIVE_INFINITY. Q. Are there functions in Java's Math library for other trigonometric functions, such as cosecant, secant, and cotangent? 1.2 Built-in Types of Data A. No, but you could use Math.sin(), Math.cos(), and Math.tan() to compute them. Choosing which functions to include in an API is a tradeoff between the convenience of having every function that you need and the annoyance of having to find one of the few that you need in a long list. No choice will satisfy all users, and the Java designers have many users to satisfy. Note that there are plenty of redundancies even in the APIs that we have listed. For example, you could use Math.sin(x)/Math.cos(x) instead of Math.tan(x). Q. It is annoying to see all those digits when printing a double. Can we arrange System.out.println() to print just two or three digits after the decimal point? A. That sort of task involves a closer look at the method used to convert from to String. The Java library function System.out.printf() is one way to do the job, and it is similar to the basic printing method in the C programming language and many modern languages, as discussed in Section 1.5. Until then, we will live with the extra digits (which is not all bad, since doing so helps us to get used to the different primitive types of numbers). double 41 Elements of Programming 42 Q&A (Variables and Expressions) Q. What happens if I forget to declare a variable? A. The compiler complains when you refer to that variable in an expression. For example, IntOpsBad is the same as Program 1.2.2 except that the variable p is not declared (to be of type int). % javac IntOpsBad.java IntOpsBad.java:7: error: cannot find symbol p = a * b; ^ symbol: variable p location: class IntOpsBad IntOpsBad.java:10: error: cannot find symbol System.out.println(a + " * " + b + " = " + p); ^ symbol: variable p location: class IntOpsBad 2 errors The compiler says that there are two errors, but there is really just one: the declaration of p is missing. If you forget to declare a variable that you use often, you will get quite a few error messages. A good strategy is to correct the first error and check that correction before addressing later ones. Q. What happens if I forget to initialize a variable? A. The compiler checks for this condition and will give you a variable might error message if you try to use the variable in an expression before you have initialized it. not have been initialized Q. Is there a difference between the = and == operators? A. Yes, they are quite different! The first is an assignment operator that changes the value of a variable, and the second is a comparison operator that produces a boolean result. Your ability to understand this answer is a sure test of whether you understood the material in this section. Think about how you might explain the difference to a friend. 1.2 Built-in Types of Data Q. Can you compare a double to an int? A. Not without doing a type conversion, but remember that Java usually does the requisite type conversion automatically. For example, if x is an int with the value 3, then the expression (x < 3.1) is true—Java converts x to double (because 3.1 is a double literal) before performing the comparison. Q. Will the statement a = b = c = 17; assign the value 17 to the three integer variables a, b, and c? A. Yes. It works because an assignment statement in Java is also an expression (that evaluates to its right-hand side) and the assignment operator is right associative. As a matter of style, we do not use such chained assignments in this book. Q. Will the expression (a test whether the values of three integer variables a, b, and c are in strictly ascending order? < b < c) A. No, it will not compile because the expression a < b produces a boolean value, which would then be compared to an int value. Java does not support chained comparisons. Instead, you need to write (a < b && b < c). Q. Why do we write (a A. Java also has an & && b) and not (a & b)? operator that you may encounter if you pursue advanced programming courses. Q. What is the value of Math.round(6.022e23)? A. You should get in the habit of typing in a tiny Java program to answer such questions yourself (and trying to understand why your program produces the result that it does). Q. I've heard Java referred to as a statically typed language. What does this mean? A. Static typing means that the type of every variable and expression is known at compile time. Java also verifies and enforces type constraints at compile time; for example, your program will not compile if you attempt to store a value of type double in a variable of type int or call Math.sqrt() with a String argument. 43 Elements of Programming 44 Exercises 1.2.1 Suppose that a and b are int variables. What does the following sequence of statements do? int t = a; b = t; a = b; 1.2.2 Write a program that uses Math.sin() and Math.cos() to check that the value of cos2 + sin2 is approximately 1 for any entered as a command-line argument. Just print the value. Why are the values not always exactly 1? 1.2.3 Suppose that a and b are boolean variables. Show that the expression (!(a && b) && (a || b)) || ((a && b) || !(a || b)) evaluates to true. 1.2.4 Suppose that a and b (!(a < b) && !(a > b)). are int variables. Simplify the following expression: 1.2.5 The exclusive or operator ^ for boolean operands is defined to be true if they are different, false if they are the same. Give a truth table for this function. 1.2.6 Why does 10/3 give 3 and not 3.333333333? Solution. Since both 10 and 3 are integer literals, Java sees no need for type conversion and uses integer division. You should write 10.0/3.0 if you mean the numbers to be double literals. If you write 10/3.0 or 10.0/3, Java does implicit conversion to get the same result. 1.2.7 What does each of the following print? a. System.out.println(2 + "bc"); b. System.out.println(2 + 3 + "bc"); c. System.out.println((2+3) + "bc"); d. System.out.println("bc" + (2+3)); e. System.out.println("bc" + 2 + 3); Explain each outcome. 1.2.8 Explain how to use Program 1.2.3 to find the square root of a number. 1.2 Built-in Types of Data 45 1.2.9 What does each of the following print? a. System.out.println('b'); b. System.out.println('b' + 'c'); c. System.out.println((char) ('a' + 4)); Explain each outcome. 1.2.10 Suppose that a variable a is declared as int a = 2147483647 (or equiva- lently, Integer.MAX_VALUE). What does each of the following print? a. System.out.println(a); b. System.out.println(a+1); c. System.out.println(2-a); d. System.out.println(-2-a); e. System.out.println(2*a); f. System.out.println(4*a); Explain each outcome. 1.2.11 Suppose that a variable a is declared as double a = 3.14159. What does each of the following print? a. System.out.println(a); b. System.out.println(a+1); c. System.out.println(8/(int) a); d. System.out.println(8/a); e. System.out.println((int) (8/a)); Explain each outcome. 1.2.12 Describe what happens if you write sqrt instead of Math.sqrt in Program 1.2.3. 1.2.13 Evaluate the expression (Math.sqrt(2) * Math.sqrt(2) == 2). 1.2.14 Write a program that takes two positive integers as command-line arguments and prints true if either evenly divides the other. Elements of Programming 46 1.2.15 Write a program that takes three positive integers as command-line arguments and prints false if any one of them is greater than or equal to the sum of the other two and true otherwise. (Note : This computation tests whether the three numbers could be the lengths of the sides of some triangle.) 1.2.16 A physics student gets unexpected results when using the code double force = G * mass1 * mass2 / r * r; to compute values according to the formula F = Gm1m2 r 2. Explain the problem and correct the code. 1.2.17 Give the value of the variable a after the execution of each of the following sequences of statements: int a = a = a = a a a a = + + + 1; a; a; a; boolean a = true; a = !a; a = !a; a = !a; int a = a = a = a a a a = * * * 2; a; a; a; 1.2.18 Write a program that takes two integer command-line arguments x and y and prints the Euclidean distance from the point (x, y) to the origin (0, 0). 1.2.19 Write a program that takes two integer command-line arguments a and b and prints a random integer between a and b, inclusive. 1.2.20 Write a program that prints the sum of two random integers between 1 and 6 (such as you might get when rolling dice). 1.2.21 Write a program that takes a double command-line argument t and prints the value of sin(2t) sin(3t). 1.2.22 Write a program that takes three double command-line arguments x0, v0, and t and prints the value of x0 v0t – g t 2 2, where g is the constant 9.80665. (Note : This value is the displacement in meters after t seconds when an object is thrown straight up from initial position x0 at velocity v0 meters per second.) 1.2.23 Write a program that takes two integer command-line arguments m and d and prints true if day d of month m is between 3/20 and 6/20, false otherwise. 1.2 Built-in Types of Data 47 Creative Exercises 1.2.24 Continuously compounded interest. Write a program that calculates and prints the amount of money you would have after t years if you invested P dollars at an annual interest rate r (compounded continuously). The desired value is given by the formula Pe rt. 1.2.25 Wind chill. Given the temperature T (in degrees Fahrenheit) and the wind speed v (in miles per hour), the National Weather Service defines the effective temperature (the wind chill) as follows: w = 35.74 0.6215 T (0.4275 T 35.75) v 0.16 Write a program that takes two double command-line arguments temperature and velocity and prints the wind chill. Use Math.pow(a, b) to compute ab. Note : The formula is not valid if T is larger than 50 in absolute value or if v is larger than 120 or less than 3 (you may assume that the values you get are in that range). 1.2.26 Polar coordinates. Write a program that converts from Cartesian to polar coordinates. Your program should accept two double commandline arguments x and y and print the polar coordinates r and . Use the method Math.atan2(y, x) to compute the arctangent value of y/x that is in the range from to . 1.2.27 Gaussian random numbers. Write a program x r Polar coordinates RandomGaussian that prints a random number r drawn from the Gaussian distribution. One way to do so is to use the Box–Muller formula r = sin(2 v) (2 ln u)1/2 where u and v are real numbers between 0 and 1 generated by the Math.random() method. 1.2.28 Order check. Write a program that takes three command-line arguments x, y, and z and prints true if the values are strictly ascending or descending (x < y < z or x > y > z), and false otherwise. double y Elements of Programming 48 1.2.29 Day of the week. Write a program that takes a date as input and prints the day of the week that date falls on. Your program should accept three int commandline arguments: m (month), d (day), and y (year). For m, use 1 for January, 2 for February, and so forth. For output, print 0 for Sunday, 1 for Monday, 2 for Tuesday, and so forth. Use the following formulas, for the Gregorian calendar: y0 = y (14 m) / 12 x = y0 y0 / 4 y0 / 100 y0 / 400 m0 = m 12 ((14 m) / 12) 2 d0 = (d x (31 m0) / 12) % 7 Example: On which day of the week did February 14, 2000 fall? y0 = x = m0 = d0 = 2000 1 = 1999 1999 1999 / 4 1999 / 100 1999 / 400 = 2483 2 12 1 2 = 12 (14 2483 (31 12) / 12) % 7 = 2500 % 7 = 1 Answer : Monday. 1.2.30 Uniform random numbers. Write a program that prints five uniform random numbers between 0 and 1, their average value, and their minimum and maximum values. Use Math.random(), Math.min(), and Math.max(). 1.2.31 Mercator projection. The Mercator projection is a conformal (anglepreserving) projection that maps latitude and longitude to rectangular coordinates (x, y). It is widely used—for example, in nautical charts and in the maps that you print from the web. The projection is defined by the equations x 0 and y 1/2 ln ((1 sin) (1 sin)), where 0 is the longitude of the point in the center of the map. Write a program that takes 0 and the latitude and longitude of a point from the command line and prints its projection. 1.2.32 Color conversion. Several different formats are used to represent color. For example, the primary format for LCD displays, digital cameras, and web pages, known as the RGB format, specifies the level of red (R), green (G), and blue (B) on an integer scale from 0 to 255. The primary format for publishing books and magazines, known as the CMYK format, specifies the level of cyan (C), magenta 1.2 Built-in Types of Data 49 (M), yellow (Y), and black (K) on a real scale from 0.0 to 1.0. Write a program RGBtoCMYK that converts RGB to CMYK. Take three integers—r, g, and b—from the command line and print the equivalent CMYK values. If the RGB values are all 0, then the CMY values are all 0 and the K value is 1; otherwise, use these formulas: w max (r / 255, g / 255, b / 255) c (w (r / 255)) w m (w (g / 255)) w y (w (b / 255)) w k 1 w 1.2.33 Great circle. Write a program GreatCircle that takes four double command-line arguments—x1, y1, x2, and y2—(the latitude and longitude, in de- grees, of two points on the earth) and prints the great-circle distance between them. The great-circle distance (in nautical miles) is given by the following equation: d = 60 arccos(sin(x1) sin(x2) cos(x1) cos(x2) cos(y1 y2)) Note that this equation uses degrees, whereas Java's trigonometric functions use radians. Use Math.toRadians() and Math.toDegrees() to convert between the two. Use your program to compute the great-circle distance between Paris (48.87° N and 2.33° W) and San Francisco (37.8° N and 122.4° W). 1.2.34 Three-sort. Write a program that takes three integer command-line arguments and prints them in ascending order. Use Math.min() and Math.max(). 1.2.35 Dragon curves. Write a program to print the instructions for drawing the dragon curves of order 0 through 5. The instructions are strings of F, L, and R characters, where F means "draw line while moving 1 unit F forward," L means "turn left," and R means "turn right." A FLF dragon curve of order n is formed when you fold a strip of paper in half n times, then unfold to right angles. The key to solving this problem is to note that a curve of order FLFLFRF n is a curve of order n 1 followed by an L followed by a curve of order n 1 traversed in reverse order, and then FLFLFRFLFLFRFRF to figure out a similar description for the reverse curve. Dragon curves of order 0, 1, 2, and 3 Elements of Programming 1.3 Conditionals and Loops In the programs that we have examined to this point, each of the statements in the program is executed once, in the order given. Most programs are more complicated because the sequence of statements and the number of times each is executed can vary. We use the term control flow to refer to statement sequencing in a program. 1.3.1 Flipping a fair coin. . . . . 53 In this section, we introduce statements 1.3.2 Your first while loop. . . . . 55 that allow us to change the control flow, 1.3.3 Computing powers of 2. . . . . 57 using logic about the values of program 1.3.4 Your first nested loops. . . . . 63 1.3.5 Harmonic numbers . . . . 65 variables. This feature is an essential 1.3.6 Newton's method . . . . 66 component of programming. 1.3.7 Converting to binary . . . . 68 Specifically, we consider Java state1.3.8 Gambler's ruin simulation . . . . . . 71 ments that implement conditionals, 1.3.9 Factoring integers . . . . 73 where some other statements may or may Programs in this section not be executed depending on certain conditions, and loops, where some other statements may be executed multiple times, again depending on certain conditions. As you will see in this section, conditionals and loops truly harness the power of the computer and will equip you to write programs to accomplish a broad variety of tasks that you could not contemplate attempting without a computer. If statements Most computations require different actions for different inputs. One way to express these differences in Java is the if statement: if () { } This description introduces a formal notation known as a template that we will use to specify the format of Java constructs. We put within angle brackets (< >) a construct that we have already defined, to indicate that we can use any instance of that construct where specified. In this case, represents an expression that evaluates to a boolean value, such as one involving a comparison operation, and represents a statement block (a sequence of Java statements). This latter construct is familiar to you: the body of main() is such a sequence. If the sequence is a single statement, the curly braces are optional. It is possible to make formal definitions of and , but we refrain from going into that level of detail. The meaning of an if statement 1.3 Conditionals and Loops 51 is self-explanatory: the statement(s) in the sequence are to be executed if and only if the expression is true. As a simple example, suppose that you want to compute the absolute value of an int value x. This statement does the job: boolean expression if (x < 0) x = -x; (More precisely, it replaces x with the absolute value of x.) As a second simple example, consider the following statement: if (x > y) { if (x > y) { int t = x; x = y; y = t; } int t = x; x = y; y = t; sequence of statements } Anatomy of an if statement This code puts the smaller of the two int values in x and the larger of the two values in y, by exchanging the values in the two variables if necessary. You can also add an else clause to an if statement, to express the concept of executing either one statement (or sequence of statements) or another, depending on whether the boolean expression is true or false, as in the following template: if () else As a simple example of the need for an else clause, consider the following code, which assigns the maximum of two int values to the variable max: if (x > y) max = x; else max = y; One way to understand control flow is to visualize it with a diagram called a flowchart. Paths through the flowchart correspond to flow-of-control paths in the if (x > y) max = x; else max = y; if (x < 0) x = -x; yes x < 0 ? x = -x; no yes x > y ? max = x; Flowchart examples (if statements) no max = y; Elements of Programming 52 program. In the early days of computing, when programmers used low-level languages and difficult-to-understand flows of control, flowcharts were an essential part of programming. With modern languages, we use flowcharts just to understand basic building blocks like the if statement. The accompanying table contains some examples of the use of if and ifelse statements. These examples are typical of simple calculations you might need in programs that you write. Conditional statements are an essential part of programming. Since the semantics (meaning) of statements like these is similar to their meanings as natural-language phrases, you will quickly grow used to them. Program 1.3.1 is another example of the use of the if-else statement, in this case for the task of simulating a fair coin flip. The body of the program is a single statement, like the ones in the table, but it is worth special attention because it introduces an interesting philosophical issue that is worth contemplating: can a computer program produce random values? Certainly not, but a program can produce numbers that have many of the properties of random numbers. absolute value if (x < 0) x = -x; put the smaller value in x and the larger value in y if (x > y) { int t = x; x = y; y = t; } maximum of x and y error check for division operation error check for quadratic formula if (x > y) max = x; else max = y; if (den == 0) System.out.println("Division by zero"); else System.out.println("Quotient = " + num/den); double discriminant = b*b if (discriminant < 0.0) { System.out.println("No } else { System.out.println((-b System.out.println((-b } - 4.0*c; real roots"); + Math.sqrt(discriminant))/2.0); - Math.sqrt(discriminant))/2.0); Typical examples of using if and if-else statements 1.3 Conditionals and Loops Program 1.3.1 Flipping a fair coin public class Flip { public static void main(String[] args) { // Simulate a fair coin flip. if (Math.random() < 0.5) System.out.println("Heads"); else System.out.println("Tails"); } } This program uses Math.random() to simulate a fair coin flip. Each time you run it, it prints either Heads or Tails. A sequence of flips will have many of the same properties as a sequence that you would get by flipping a fair coin, but it is not a truly random sequence. % java Flip Heads % java Flip Tails % java Flip Tails While loops Many computations are inherently repetitive. The basic Java construct for handling such computations has the following format: while () { } The while statement has the same form as the if statement (the only difference being the use of the keyword while instead of if), but the meaning is quite different. It is an instruction to the computer to behave as follows: if the boolean expression is false, do nothing; if the boolean expression is true, execute the sequence of statements (just as with an if statement) but then check the expression again, execute the sequence of statements again if the expression is true, and continue as long as the expression is true. We refer to the statement block in a loop as the body of the loop. As with the if statement, the curly braces are optional if a while loop body has just one statement. The while statement is equivalent to a sequence of identical if statements: 53 Elements of Programming 54 if () { } if () { } if () { } ... At some point, the code in one of the statements must change something (such as the value of some variable in the boolean expression) to make the boolean expression false, and then the sequence is broken. A common programming paradigm involves maintaining an integer value that keeps track of the number of times a loop iterates. We start at some initial value, and then increment the value by 1 each time initialization is a loopthrough the loop, testing whether it exceeds a pre- separate statement continuation condition determined maximum before deciding to continue. int power = 1; TenHellos (Program 1.3.2) is a simple example of while (power 0) exchange(a, j-1, j); else break; } At the beginning of each iteration of the outer for loop, the first i elements in the array are in sorted order; the inner for loop moves a[i] into its proper position in the array, as in the following example when i is 6: i j 6 6 a[] 0 1 2 3 4 5 6 7 and had him his was you the but 6 5 and had him his was the you but 6 4 and had him his the was you but and had him his the was you but Inserting a[6] into position by exchanging it with larger values to its left Specifically, a[i] is put in its place among the sorted elements to its left by exchanging it (using the exchange() method that we first encountered in Section 2.1) with each larger value to its left, moving from right to left, until it reaches its proper position. The black elements in the three bottom rows in this trace are the ones that are compared with a[i]. The insertion process just described is executed, first with i equal to 1, then 2, then 3, and so forth, as illustrated in the following trace. Algorithms and Data Structures 544 i j 1 0 a[] 0 1 2 3 4 5 6 7 was had him and you his the but had was him and you his the but 2 1 had him was and you his the but 3 0 and had him was you his the but 4 4 and had him was you his the but 5 3 and had him his was you the but 6 4 and had him his the was you but 7 1 and but had him his the was you and but had him his the was you Inserting a[1] through a[n-1] into position (insertion sort) Row i of the trace displays the contents of the array when the outer for loop completes, along with the value of j at that time. The highlighted string is the one that was in a[i] at the beginning of the loop, and the other strings printed in black are the other ones that were involved in exchanges and moved to the right one position within the loop. Since the elements a[0] through a[i-1] are in sorted order when the loop completes for each value of i, they are, in particular, in sorted order the final time the loop completes, when the value of i is a.length. This discussion again illustrates the first thing that you need to do when studying or developing a new algorithm: convince yourself that it is correct. Doing so provides the basic understanding that you need to study its performance and use it effectively. Analysis of running time. The inner loop of the insertion sort code is within a double nested for loop, which suggests that the running time is quadratic, but we cannot immediately draw this conclusion because of the break statement. For example, in the best case, when the input array is already in sorted order, the inner for loop amounts to nothing more than a single compare (to learn that a[j-1] is less than or equal to a[j] for each j from 1 to n-1) and the break, so the total running time is linear. In contrast, if the input array is in reverse-sorted order, the inner loop fully completes without a break, so the frequency of execution of the instructions in the inner loop is 1 2 ... n 1 ~ ½ n 2 and the running time is quadratic. To understand the performance of insertion sort for randomly ordered input arrays, take a careful look at the trace: it is an n-by-n array with one black element corresponding to each exchange. That is, the number of black elements is 4.2 Sorting and Searching 545 the frequency of execution of instructions in the inner loop. We expect that each new element to be inserted is equally likely to fall into any position, so, on average, that element will move halfway to the left. Thus, on average, we expect only about half of the elements below the diagonal (about n 2 4 in total) to be black. This leads immediately to the hypothesis that the expected running time of insertion sort for a randomly ordered input array is quadratic. ~n 2/2 elements above the diagonal are shaded unshaded elements are the ones that moved n 2 elements in total was had had and and and and was had him has had had had but him him was him him him had and and and was his his you you you you was the his his his his his you was the ~n 2/4 (half) of the elements below the diagonal, on the average, are black Sorting other types of data. We want to be Analysis of insertion sort able to sort all types of data, not just strings. In a scientific application, we might wish to sort experimental results by numeric values; in a commercial application, we might wish to use monetary amounts, times, or dates; in systems software, we might wish to use IP addresses or process IDs. The idea of sorting in each of these situations is intuitive, but implementing a sort method that works in all of them is a prime example of the need for a functional abstraction mechanism like the one provided by Java interfaces. For sorting objects in an array, we need only assume that we can compare two elements to see whether the first is bigger than, smaller than, or equal to the second. Java provides the java.util.Comparable interface for precisely this purpose. public interface Comparable int compareTo(Key b) compare this object with b for order API for Java's java.util.Comparable interface A class that implements the Comparable interface promises to implement a method compareTo() for objects of its type so that a.compareTo(b) returns a negative integer (typically -1) if a is less than b, a positive integer (typically +1) if a is greater than b, and 0 if a is equal to b. (The notation, which we will introduce in Section 4.3, ensures that the two objects being compared have the same type.) The precise meanings of less than, greater than, and equal to depends on the data type, though implementations that do not respect the natural laws of math- the the the the the you was but but but but but but you 546 Algorithms and Data Structures ematics surrounding these concepts will yield unpredictable results. More formally, the compareTo() method must define a total order. This means that the following three properties must hold (where we use the notation x y as shorthand for x.compareTo(y) b.count) return +1; else return 0; } ... } Now, we can use Insertion.sort() to sort an array of Counter objects in ascending order of their counts. 4.2 Sorting and Searching 547 Program 4.2.4 Insertion sort public class Insertion { public static void sort(Comparable[] a) { // Sort a[] into increasing order. int n = a.length; for (int i = 1; i < n; i++) // Insert a[i] into position. for (int j = i; j > 0; j--) if (a[j].compareTo(a[j-1]) < 0) exchange(a, j-1, j); else break; } a[] n array to sort length of array public static void exchange(Comparable[] a, int i, int j) { Comparable temp = a[j]; a[j] = a[i]; a[i] = temp; } public static void main(String[] args) { // Read strings from standard input, sort them, and print. String[] a = StdIn.readAllStrings(); sort(a); for (int i = 0; i < a.length; i++) StdOut.print(a[i] + " "); StdOut.println(); } } The sort() function is an implementation of insertion sort. It sorts arrays of any type of data that implements the Comparable interface (and, therefore, has a compareTo() method). Insertion.sort() is appropriate only for small arrays or for arrays that are nearly in order; it is too slow to use for large arrays that are out of order. % more 8words.txt was had him and you his the but % java Insertion < 8words.txt and but had him his the was you % java Insertion < TomSawyer.txt tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick tick 548 Algorithms and Data Structures Empirical analysis. InsertionDoublingTest (Program 4.2.5) tests our hypothesis that insertion sort is quadratic for randomly ordered arrays by running Insertion.sort() on n random Double objects, computing the ratios of running times as n doubles. This ratio converges to 4, which validates the hypothesis that the running time is quadratic, as discussed in the last section. You are encouraged to run InsertionDoublingTest on your own computer. As usual, you might notice the effect of caching or some other system characteristic for some values of n, but the quadratic running time should be quite evident, and you will be quickly convinced that insertion sort is too slow to be useful for large inputs. Sensitivity to input. Note that InsertionDoublingTest takes a command-line argument trials and runs trials experiments for each array length, not just one. As we have just observed, one reason for doing so is that the running time of insertion sort is sensitive to its input values. This behavior is quite different from (for example) ThreeSum, and means that we have to carefully interpret the results of our analysis. It is not correct to flatly predict that the running time of insertion sort will be quadratic, because your application might involve input for which the running time is linear. When an algorithm's performance is sensitive to input values, you might not be able to make accurate predictions without taking them into account. There are many natural applications for which insertion sort is quadratic, so we need to consider faster sorting algorithms. As we know from Section 4.1, a back-ofthe-envelope calculation can tell us that having a faster computer is not much help. A dictionary, a scientific database, or a commercial database can contain billions of elements; how can we sort such a large array? 4.2 Sorting and Searching 549 Program 4.2.5 Doubling test for insertion sort public class InsertionDoublingTest { public static double timeTrials(int trials, int { // Sort random arrays of size n. double total = 0.0; Double[] a = new Double[n]; for (int t = 0; t < trials; t++) { for (int i = 0; i < n; i++) a[i] = StdRandom.uniform(0.0, 1.0); Stopwatch timer = new Stopwatch(); Insertion.sort(a); total += timer.elapsedTime(); } return total; } public static void main(String[] args) { // Print doubling ratios for insertion sort. int trials = Integer.parseInt(args[0]); for (int n = 1024; true; n += n) { double prev = timeTrials(trials, n/2); double curr = timeTrials(trials, n); double ratio = curr / prev; StdOut.printf("%7d %4.2f", n, ratio); } } } n) trials n number of trials problem size total total elapsed time timer stopwatch a[] array to sort prev running time for n/2 curr running time for n ratio ratio of running times The method timeTrials() runs Insertion.sort() for arrays of random double values. The first argument n is the length of the array; the second argument trials is the number of trials. Multiple trials produce more accurate results because they dampen system effects and because insertion sort's running time depends on the input. % java 1024 2048 4096 8192 16384 32768 InsertionDoublingTest 1 0.71 3.00 5.20 3.32 3.91 3.89 % java 1024 2048 4096 8192 16384 32768 InsertionDoublingTest 10 1.89 5.00 3.58 4.09 4.83 3.96 Algorithms and Data Structures 550 Mergesort To develop a faster sorting method, input was had him and you his the but we use recursion and a divide-and-conquer approach to algorithm design that every program- sort left and had him was you his the mer needs to understand. This nomenclature resort right fers to the idea that one way to solve a problem is and had him was but his the to divide it into independent parts, conquer them merge independently, and then use the solutions for the and but had him his the was parts to develop a solution for the full problem. To Mergesort overview sort an array with this strategy, we divide it into two halves, sort the two halves independently, and then merge the results to sort the full array. This algorithm is known as mergesort. We process contiguous subarrays of a given array, using the notation a[lo, hi) to refer to a[lo], a[lo+1], ..., a[hi-1] (adopting the same convention that we used for binary search to denote a half-open interval that excludes a[hi]). To sort a[lo, hi), we use the following recursive strategy: • Base case : If the subarray length is 0 or 1, it is already sorted. • Reduction step : Otherwise, compute mid = lo + (hi - lo)/2, recursively sort the two subarrays a[lo, mid) and a[mid, hi), and merge them. Merge (Program 4.2.6) is an implementation of this algorithm. The values in the array are rearranged by the code that follows the recursive calls, which merges the two subarrays that were sorted by the recursive calls. As usual, the easiest way to understand the merge process is to study a trace during the merge. The code maintains one index i into the first subarray, another index j into the second subarray, i j k aux[k] a[] 0 1 2 3 4 5 6 7 and had him was but his the you 0 4 0 and had him was but his the you 1 4 1 but and had him was but his the you 1 5 2 had and had him was but his the you 2 5 3 him and had him was but his the you 3 5 4 his and had him was but his the you 3 6 5 the and had him was but his the you 3 7 6 was and had him was but his the you 4 7 7 you and had him was but his the you Trace of the merge of the sorted left subarray with the sorted right subarray but you you 4.2 Sorting and Searching 551 and a third index k into an auxiliary array aux[] that temporarily holds the result. The merge implementation is a single loop that sets aux[k] to either a[i] or a[j] (and then increments k and the index the subarray that was used). If either i or j has reached the end of its subarray, aux[k] is set from the other; otherwise, it is set to the smaller of a[i] or a[j]. After all of the values from the two subarrays have been copied to aux[], the sorted result in aux[] is copied back to the original array. Take a moment to study the trace just given to convince yourself that this code always properly combines the two sorted subarrays to sort the full array. The recursive method ensures that the two subarrays are each put into sorted order just prior to the merge. Again, the best way to gain an understanding of this process is to study a trace of the contents of the array each time the recursive sort() method returns. Such a trace for our example is shown next. First a[0] and a[1] are merged to make a sorted subarray in a[0, 2), then a[2] and a[3] are merged to make a sorted subarray in a[2, 4), then these two subarrays of size 2 are merged to make a sorted subarray in a[0, 4), and so forth. If you are convinced that the merge works properly, you need only convince yourself that the code properly divides the array to be convinced that the sort works properly. Note that when the number of elements in a subarray to be sorted is not even, the left half will have one fewer element than the right half. a[] 0 1 2 3 4 5 6 7 was had him and you his the but had was him and you his the but had was and him you his the but and had him was you his the but and had him was his you the but and had him was his you but the and had him was but his the you and but had him his the was you sort(a, aux, 0, 8) sort(a, aux, 0, 4) sort(a, aux, 0, 2) return sort(a, aux, 2, 4) return return sort(a, aux, 4, 8) sort(a, aux, 4, 6) return sort(a, aux, 6, 8) return return return Trace of recursive mergesort calls Algorithms and Data Structures 552 Program 4.2.6 Mergesort public class Merge { public static void sort(Comparable[] a) { Comparable[] aux = new Comparable[a.length]; sort(a, aux, 0, a.length); } private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) { // Sort a[lo, hi). a[lo, hi) subarray to sort if (hi - lo = 0; j--) StdOut.println(zipf[j]); } } This program sorts the words on standard input, uses the sorted list to count the frequency of occurrence of each, and then sorts the frequencies. The test file used below has more than 20 million words. The plot compares the ith frequency relative to the first (bars) with 1/i (blue). % java FrequencyCount < Leipzig1M.txt the: 1160105 of: 593492 to: 560945 a: 472819 and: 435866 in: 430484 for: 205531 The: 192296 that: 188971 is: 172225 said: 148915 on: 147024 was: 141178 by: 118429 ... 558 Algorithms and Data Structures Lessons The vast majority of programs that we write involve managing the complexity of addressing a new practical problem by developing a clear and correct solution, breaking the program into modules of manageable size, and testing and debugging our solution. From the very start, our approach in this book has been to develop programs along these lines. But as you become involved in ever more complex applications, you will find that a clear and correct solution is not always sufficient, because the cost of computation can be a limiting factor. The examples in this section are a basic illustration of this fact. Respect the cost of computation. If you can quickly solve a small problem with a simple algorithm, fine. But if you need to address a problem that involves a large amount of data or a substantial amount of computation, you need to take into account the cost. Reduce to a known problem. Our use of sorting for frequency counting illustrates the utility of understanding fundamental algorithms and using them for problem solving. Divide-and-conquer. It is worthwhile for you to reflect a bit on the power of the divide-and-conquer paradigm, as illustrated by developing a linearithmic sorting algorithm (mergesort) that serves as the basis for addressing so many computational problems. Divide-and-conquer is but one approach to developing efficient algorithms. Since the advent of computing, people have been developing algorithms such as binary search and mergesort that can efficiently solve practical problems. The field of study known as design and analysis of algorithms encompasses the study of design paradigms such as divide-and-conquer and dynamic programming, the invention of algorithms for solving fundamental problems like sorting and searching, and techniques to develop hypotheses about the performance of algorithms. Implementations of many of these algorithms are found in Java libraries or other specialized libraries, but understanding these basic tools of computation is like understanding the basic tools of mathematics or science. You can use a matrix-processing package to find the eigenvalues of a matrix, but you still need a course in linear algebra. Now that you know a fast algorithm can make the difference between spinning your wheels and properly addressing a practical problem, you can be on the lookout for situations where algorithm design and analysis can make the difference, and where efficient algorithms such as binary search and mergesort can do the job. 4.2 Sorting and Searching 559 Q&A Q. Why do we need to go to such lengths to prove a program correct? A. To spare ourselves considerable pain. Binary search is a notable example. For example, you now understand binary search; a classic programming exercise is to write a version that uses a while loop instead of recursion. Try solving Exercise 4.2.2 without looking back at the code in the book. In a famous experiment, Jon Bentley once asked several professional programmers to do so, and most of their solutions were not correct. Q. Are there implementations for sorting and searching in the Java library? A. Yes. The Java package java.util contains the static methods Arrays.sort() and Arrays.binarySearch() that implement mergesort and binary search, respectively. Actually, each represents a family of overloaded methods, one for Comparable types, and one for each primitive type. Q. So why not just use them? A. Feel free to do so. As with many topics we have studied, you will be able to use such tools more effectively if you understand the background behind them. Q. Explain why we use lo lo and hi instead of + (hi - lo) / 2 to compute the index midway between using (lo + hi) / 2. A. The latter fails when lo + hi overflows an int. Q. Why do I get a unchecked or unsafe operation Insertion.java and Merge.java? warning when compiling A. The argument to sort() is a Comparable array, but nothing, technically, prevents its elements from being of different types. To eliminate the warning, change the signature to: public static void sort(Key[] a) We'll learn about the notation in the next section when we discuss generics. Algorithms and Data Structures 560 Exercises 4.2.1 Develop an implementation of (Program 4.2.1) that takes the maximum number n as a command-line argument. Prove that your implementation is correct. Questions 4.2.2 Develop a nonrecursive version of BinarySearch (Program 4.2.3). 4.2.3 Modify BinarySearch (Program 4.2.3) so that if the search key is in the array, it returns the smallest index i for which a[i] is equal to key, and otherwise returns -i, where i is the smallest index such that a[i] is greater than key. 4.2.4 Describe what happens if you apply binary search to an unordered array. Why shouldn't you check whether the array is sorted before each call to binary search? Could you check that the elements binary search examines are in ascending order? 4.2.5 Describe why it is desirable to use immutable keys with binary search. 4.2.6 Add code to Insertion to produce the trace given in the text. 4.2.7 Add code to Merge to produce a trace like the following: % java Merge < tiny.txt was had him and you his the but had was him and you his but the his but and but had him his the was but but the was you but 4.2.8 Give traces of insertion sort and mergesort in the style of the traces in the text, for the input it was the best of times it was. 4.2.9 Implement a more general version of Program 4.2.2 that applies bisection search to any monotonically increasing function. Use functional programming, in the same style as the numerical integration example from Section 3.3. 4.2.10 Write a filter DeDup that reads strings from standard input and prints them to standard output, with all duplicate strings removed (and in sorted order). 4.2.11 Modify StockAccount (Program 3.2.8) so that it implements the Comparable interface (comparing the stock accounts by name). Hint: Use the compareTo() method from the String data type for the heavy lifting. 4.2.12 Modify Vector (Program 3.3.3) so that it implements the Comparable interface (comparing the vectors lexicographically by coordinates). 4.2.13 Modify Time (Exercise 3.3.21) so that it implements the Comparable interface (comparing the times chronologically). 4.2.14 Modify Counter (Program 3.3.2) so that it implements the Comparable interface (comparing the objects by frequency count). 4.2.15 Add methods to Insertion (Program 4.2.4) and Merge (Program 4.2.6) to support sorting subarrays. 4.2.16 Develop a nonrecursive version of mergesort (Program 4.2.6). For simplicity, assume that the number of items n is a power of 2. Extra credit: Make your program work even if n is not a power of 2. 4.2.17 Find the frequency distribution of words in your favorite novel. Does it obey Zipf's law? 4.2.18 Analyze mathematically the number of compares that mergesort makes to sort an array of length n. For simplicity, assume n is a power of 2. Answer: Let M(n) be the number of compares to mergesort an array of length n. Merging two subarrays whose total length is n requires between ½ n and n − 1 compares. Thus, M(n) satisfies the following recurrence relation: M(n) ≤ 2M(n/2) + n ... Substituting n = 2k ... we get M(2k) ≤ ... Dividing both sides by 2k ... Substituting back for 2k ... we have ... shows that the number of compares is at most n lg n. A similar argument shows that the number of compares is at least ½ n lg n. 4.2.19 Analyze mergesort's running time when n is not a power of 2. Partial solution. When n is an odd number, one subarray has one more element than the other, so when n is not a power of 2, the subarrays on each level are not necessarily all the same size. Still, every element appears in some subarray, and the number of levels is still logarithmic, so the linearithmic hypothesis is justified for all n. 4.2 Sorting and Searching Creative Exercises The following exercises are intended to give you experience in developing fast solutions to typical problems. Think about using binary search and mergesort, or devising your own divide-and-conquer algorithm. Implement and test your algorithm. 4.2.20 Median. Add to StdStats (Program 2.2.4) a method median() that computes in linearithmic time the median of an array of n integers. Hint: Reduce to sorting. 4.2.21 Mode. Add to StdStats (Program 2.2.4) a method mode() that computes in linearithmic time the mode (value that occurs most frequently) of an array of n integers. Hint: Reduce to sorting. 4.2.22 Integer sort. Write a linear-time filter that reads from standard input a sequence of integers that are between 0 and 99 and prints to standard output the same integers in sorted order. For example, presented with the input sequence 98 2 3 1 0 0 0 3 98 98 2 2 2 0 0 0 2 your program should print the output sequence 0 0 0 0 0 0 1 2 2 2 2 2 3 3 98 98 98 4.2.23 Floor and ceiling. Given a sorted array of Comparable items, write functions floor() and ceiling() that return the index of the largest (or smallest) item not larger (or smaller) than an argument item in logarithmic time. 4.2.24 Bitonic maximum. An array is bitonic if it consists of an increasing sequence of keys followed immediately by a decreasing sequence of keys. Given a bitonic array, design a logarithmic algorithm to find the index of a maximum key. 4.2.25 Search in a bitonic array. Given a bitonic array of n distinct integers, design a logarithmic-time algorithm to determine whether a given integer is in the array. 4.2.26 Closest pair. Given an array of n real numbers, design a linearithmic-time algorithm to find a pair of numbers that are closest in value. 4.2.27 Furthest pair. Given an array of n real numbers, design a linear-time algorithm to find a pair of numbers that are furthest apart in value. 563 Algorithms and Data Structures 564 4.2.28 Two sum. Given an array of n integers, design a linearithmic-time algorithm to determine whether any two of them sum to 0. 4.2.29 Three sum. Given an array of n integers, design an algorithm to determine whether any three of them sum to 0. The order of growth of the running time of your program should be n2 log n. Extra credit: Develop a program that solves the problem in quadratic time. 4.2.30 Majority. A value in an array of length n is a majority if it appears strictly more than n / 2 times. Given an array of strings, design a linear-time algorithm to identify a majority element (if one exists). 4.2.31 Largest empty interval. Given n timestamps for when a file is requested from a web server, find the largest interval of time in which no file is requested. Write a program to solve this problem in linearithmic time. 4.2.32 Prefix-free codes. In data compression, a set of strings is prefix-free if no string is a prefix of another. For example, the set of strings { 01, 10, 0010, 111 } is prefix-free, but the set of strings { 01, 10, 0010, 1010 } is not prefix-free because 10 is a prefix of 1010. Write a program that reads in a set of strings from standard input and determines whether the set is prefix-free. 4.2.33 Partitioning. Design a linear-time algorithm to sort an array of Comparable objects that is known to have at most two distinct values. Hint: Maintain two pointers, one starting at the left end and moving right, and the other starting at the right end and moving left. Maintain the invariant that all elements to the left of the left pointer are equal to the smaller of the two values and all elements to the right of the right pointer are equal to the larger of the two values. 4.2.34 Dutch-national-flag problem. Design a linear-time algorithm to sort an array of Comparable objects that is known to have at most three distinct values. (Edsger Dijkstra named this the Dutch-national-flag problem because the result is three "stripes" of values like the three stripes in the flag.) 4.2 Sorting and Searching 565 Quicksort. Write a recursive program that sorts an array of Comparable objects by using, as a subroutine, the partitioning algorithm described in the previous exercise: First, pick a random element v as the partitioning element. Next, partition the array into a left subarray containing all elements less than v, followed by a middle subarray containing all elements equal to v, followed by a right subarray containing all elements greater than v. Finally, recursively sort the left and right subarrays. 4.2.36 Reverse domain name. Write a filter that reads a sequence of domain names from standard input and prints the reverse domain names in sorted order. For example, the reverse domain name of cs.princeton.edu is edu.princeton.cs. This computation is useful for web log analysis. To do so, create a data type Domain that implements the Comparable interface (using reverse-domain-name order). 4.2.37 Local minimum in an array. Given an array of n real numbers, design a logarithmic-time algorithm to identify a local minimum (an index i such that both a[i] < a[i-1] and a[i] < a[i+1]). 4.2.38 Discrete distribution. Design a fast algorithm to repeatedly generate numbers from the discrete distribution: Given an array a[] of non-negative real numbers that sum to 1, the goal is to return index i with probability a[i]. Form an array sum[] of cumulative sums such that sum[i] is the sum of the first i elements of a[]. Now, generate a random real number r between 0 and 1, and use binary search to return the index i for which sum[i] ≤ r < sum[i+1]. Compare the performance of this approach with the approach taken in RandomSurfer (Program 1.6.2). 4.2.39 Implied volatility. Typically the volatility σ is the unknown value in the Black–Scholes formula (see Exercise 2.1.28). Write a program that reads s, x, r, t, and the current price of the European call option from the command line and uses bisection search to compute σ. 4.2.40 Percolation threshold. Write a Percolation (Program 2.4.1) client that uses bisection search to estimate the percolation threshold value. 4.3 Stacks and Queues In this section, we introduce two closely related data types for manipulating arbitrarily large collections of objects: the stack and the queue. Stacks and queues are special cases of the idea of a collection. We refer to the objects in a collection as items. A collection is characterized by four operations: create the collection, insert an item, remove an item, and test whether the collection is empty. 4.3.1 ... 570 Program 4.3.2 ... 579 the collection is empty. 4.3.4 Generic stack ... 584 When we insert an item into a collection, our intent is clear. But when we remove an item, which one do we choose? Each type of collection is characterized by the rule used for removal, and each is amenable to various implementations with differing performance characteristics. You have encountered different rules for removing items in various real-world situations, perhaps without thinking about it. For example, the rule used for a queue is to always remove the item that has been in the collection for the most amount of time. This policy is known as first-in first-out, or FIFO. People waiting in line to buy a ticket follow this discipline: the line is arranged in the order of arrival, so the one who leaves the line has been there longer than any other person in the line. A policy with quite different behavior is the rule used for a stack: always remove the item that has been in the collection for the least amount of time. This policy is known as last-in first-out, or LIFO. For example, you follow a policy closer to LIFO when you enter and leave the coach cabin in an airplane: people near the front of the cabin board last and exit before those who boarded earlier. Stacks and queues are broadly useful, so it is important to be familiar with their basic properties and the kind of situation where each might be appropriate. They are excellent examples of fundamental data types that we can use to address higher-level programming tasks. They are widely used in systems and applications programming, as we will see in several examples in this section and in Section 4.5. 4.3 Stacks and Queues Pushdown stacks A pushdown stack (or just a stack) is a collection that is based on the last-in first-out (LIFO) policy. The LIFO policy underlies several of the applications that you use regularly on your computer. For example, many people organize their email as a stack, where messages go on the top when they are received and are taken from the top, with the most recently received message first (last in, first out). The advantage of this strategy is that we see new messages as soon as possible; the disadvantage is that some old messages might never get read if we never empty the stack. You have likely encountered another common example of a stack when surfing the web. When you click a hyperlink, your browser displays the new page (and inserts it onto a stack). You can keep clicking on hyperlinks to visit new pages, but you can always revisit the previous page by clicking the back button (remove it from a stack). The last-in first-out policy offered by a pushdown stack provides just the behavior that you expect. push() new (black) are Such uses of stacks are intuitive, goes on top but perhaps not persuasive. In fact, the importance of stacks in computing is fundamental and profound, but we defer further discussion of applications to later in this section. For the moment, our goal is to make sure that you understand how stacks work and how to implement them. Stacks have been used widely since the earliest days of computing, remove the By tradition, we name the stack insert = pop() gray one operation push and the stack remove operation pop, as indicated in the following API: Operations on a pushdown stack 567 Algorithms and Data Structures 568 public class *StackOfStrings *StackOfStrings() boolean isEmpty() void push(String item) String pop() create an empty stack is the stack empty? insert a string onto the stack remove and return the most recently inserted string API for a pushdown stack of strings The asterisk indicates that we will be considering more than one implementation of this API (we consider three in this section: ArrayStackOfStrings, LinkedStackOfStrings, and ResizingArrayStackOfStrings). This API also includes a method to test whether the stack is empty, leaving to the client the responsibility of using isEmpty() to avoid invoking pop() when the stack is empty. This API has an important restriction that is inconvenient in applications: we would like to have clients that contain other types of data, not just strings. We describe how to remove this restriction (and the importance of doing so) later in this section. Array implementation Representing stacks with arrays is a natural idea, but before reading further, it is worthwhile to think for a moment about how you would implement a class ArrayStackOfStrings. The first problem that you might encounter is implementing the constructor ArrayStackOfStrings(). You clearly need an instance variable items[] with an array of strings to hold the stack items, but how big should the array be? One solution is to start with an array of length 0 and make sure that the array length is always equal to the stack size, but that solution necessitates allocating a new array and copying all of the items into it for each push() and pop() operation, which is unnecessarily inefficient and cumbersome. We will temporarily finesse this problem by having the client provide an argument for the constructor that gives the maximum stack size. Your next problem might stem from the natural decision to keep the n items in the array in the order they were inserted, with the most recently inserted item in items[0] and the least recently inserted item in items[n-1]. But then each time you push or pop an item, you would have to move all of the other items to reflect the new state of the stack. A simpler and more efficient way to proceed is to keep 4.3 Stacks and Queues 569 items in the opposite order, with the most recently inserted item in items[n-1] and the least recently inserted item in items[0]. This policy allows us to add and remove items at the end of the array, without moving any of the other items in the arrays. We could hardly hope for a simpler implementation of the stack API than ArrayStackOfStrings (Program 4.3.1)—all of the methods are one-liners! The instance variables are an array items[] that holds the items in the stack and an integer n that counts the number of items in the stack. To remove an item, we decrement n and then return items[n]; to insert a new item, we set items[n] equal to the new item and then increment n. These operations preserve the following properties: • The number of items in the stack is n. StdIn StdOut n • The stack is empty when n is 0. • The stack items are stored in the array in the order in which they were inserted. be 2 to be • The most recently inserted item (if the stack is nonempty) is items[n-1]. not 4 to be or not to be As usual, thinking in terms of invariants of to 5 to be or not to this sort is the easiest way to verify that an - 4 to be or not to implementation operates as intended. Be sure be 5 to be or not be that you fully understand this implementation. - 4 to be or not be Perhaps the best way to do so is to carefully not 3 to be or examine a trace of the stack contents for a that 4 to be or that - 3 to be or sequence of push() and pop() operations. or 2 to be or that The test client in ArrayStackOfStrings allows for testing with an arbitrary sequence of is 2 to be or is operations: it does a push() for each string on standard input except the string consisting of a minus sign, for which it does a pop(). The primary characteristic of this implementation is that the push and pop operations take constant time. The drawback of this implementation is that it requires the client to estimate the maximum size of the stack ahead of time and always uses space proportional to that maximum, which may be unreasonable in some situations. We omit the code in push() to test for a full stack, but later we will examine implementations that address this drawback by not allowing the stack to get full (except in an extreme circumstance when there is no memory at all available for use by Java). Algorithms and Data Structures 570 Program 4.3.1 Stack of strings (array) public class ArrayStackOfStrings { private String[] items; private int n = 0; public ArrayStackOfStrings(int capacity) { items = new String[capacity]; } public boolean isEmpty() { return (n == 0); } public void push(String item) { items[n++] = item; } public String pop() { return items[--n]; } items[] n items[n-1] stack items number of items most recently inserted item public static void main(String[] args) { // Create a stack of specified capacity; push strings // and pop them, as directed on standard input. int cap = Integer.parseInt(args[0]); ArrayStackOfStrings stack = new ArrayStackOfStrings(cap); while (!StdIn.isEmpty()) { String item = StdIn.readString(); if (!item.equals("-")) stack.push(item); else StdOut.print(stack.pop() + " "); } } } % more tobe.txt to be or not to - be - - that - - - is % java ArrayStackOfStrings 5 < tobe.txt to be not that or be Stack methods are simple one-liners, as illustrated in this code. The client pushes or pops strings as directed from standard input (a minus sign indicates pop, and any other string indicates push). Code in push() to test whether the stack is full is omitted (see the text). % more tobe.txt to be or not to - be - - that - - - is % java ArrayStackOfStrings 5 < tobe.txt to be not that or be StdIn push() pop() n 0 1 2 3 4 to 1 be 2 or 3 not 4 to 5 - 4 be 5 - 4 - 3 that 4 - 3 - 2 - 1 is 2 Trace of ArrayStackOfStrings test client 4.3 Stacks and Queues 571 Linked lists For classes such as stacks and queues that implement collections of objects, an important objective is to ensure that the amount of memory used is proportional to the number of items in the collection. The use of a fixed-length array to implement a stack in ArrayStackOfStrings works against this objective: when you create a stack with a specified capacity, you are wasting a potentially huge amount of memory at times when the stack is empty or nearly empty. This property makes our fixed-length array implementation unsuitable for many applications. Now we consider the use of a fundamental data structure known as a linked list, which can provide implementations of collections (and, in particular, stacks and queues) that achieve the objective cited at the beginning of this paragraph. A singly linked list comprises a sequence of nodes, with each node containing a reference (or link) to its successor. By convention, the link in the last node is null, to indicate that it terminates the list. A node is an abstract entity that might hold any kind of data, in addition to the link that characterizes its role in building linked lists. When tracing code that uses linked lists and other linked structures, we use a visual representation where: • We draw a rectangle to represent each linked-list node. • We put the item and link within the rectangle. • We use arrows that point to the referenced objects to depict references. This visual representation captures the essential characteristic of linked lists and focuses on the links. For example, the diagram on this page illustrates a singly linked list containing the sequence of items to, be, or, not, to, and be. With object-oriented programming, implementing linked lists is not difficult. We define a class for the node abstraction that is recursive in nature. As with recursive functions, the concept of recursive data structures can first make your head spin. class Node { String item; Node next; } to be or not first Anatomy of a singly linked list Algorithms and Data Structures 572 the successor node in the linked list (or null to indicate that there is no such node). Using this recursive definition, we can represent a linked list with a variable of type Node by ensuring that its value is either null or a reference to a Node whose next field is a reference to a linked list. To emphasize that we are just using the Node class to structure the data, we do not define any instance methods. As with any class, we can create an object of type Node by invoking the (no-argument) constructor with new Node(). The result is a reference to a new Node object whose instance variables are each initialized to the default value null. For example, to build a linked list that contains the sequence of items to, be, and or, we create a Node for each item: Node first = new Node(); Node second = new Node(); Node third = new Node(); and assign the item instance variable in each of the nodes to the desired value: first.item = "to"; second.item = "be"; third.item = "or"; and set the next instance variables to build the linked list: first.next = second; second.next = third; As a result, first is a reference to the first node in a three-node linked list, second is a reference to the second node, and third is a reference to the last node. The code in the accompanying diagram does these same assignment statements, but in a different order. Node third = new Node(); third.item = "or"; second.next = third; 4.3 Stacks and Queues 573 A linked list represents a sequence of items. In the example just considered, first represents the sequence to be or. Alternatively, we can use an array to represent a sequence of items. For example, we could use String[] items = { "to", "be", "or" }; to represent the same sequence of strings. The difference is that it is easier to insert items into the sequence and to remove items from the sequence with linked lists. Next, we consider code to accomplish these two tasks. Suppose that you want to insert a new node into a linked list. The easiest place to do so is at the beginning of the list. For example, to insert the string not at the beginning of a given linked list whose first node is first, we save first in a temporary variable oldFirst, assign to first a new Node, and assign to its item instance variables the new item and to its next field the value of oldFirst. Now, suppose that you want to remove the first node from a list. This operation is even easier: simply assign to first the value of first.next. Inserting a new node at the beginning of a linked list Normally, you would retrieve the value of the item (by assigning it to some variable) before doing this assignment, because once you change the value of first, you may no longer have any access to the node to which it was referring. Typically, the Node object becomes an orphan, and the memory it occupies is first to eventually reclaimed by the Java memory management system. null Removing the first node in a linked list 574 Algorithms and Data Structures This code for inserting and removing a node from the beginning of a linked list involves just a few assignment statements and thus takes constant time (independent of the length of the list). If you hold a reference to a node at an arbitrary position in a list, you can use similar (but more complicated) code to remove the node after it or to insert a node after it, also in constant time. However, we leave those implementations for exercise (see Exercise 4.3.24 and Exercise 4.3.25) because inserting and removing at the beginning are the only linked-list operations that we need to implement stacks. Implementing stacks with linked lists. LinkedStackOfStrings (Program 4.3.2) uses a linked list to implement a stack of strings, using little more code than the elementary solution that uses a fixed-length array. The implementation is based on a nested class Node like the one we have been using. Java allows us to define and use other classes within class implementations in this natural way. The class is private because clients do not need to know any of the details of the linked lists. One characteristic of a private nested class is that its instance variables can be directly accessed from within the enclosing class but nowhere else, so there is no need to declare the Node instance variables as public or private (but there is no harm in doing so). LinkedStackOfStrings itself has just one instance variable: a reference to the linked list that represents the stack. That single link suffices to directly access the item at the top of the stack and indirectly access the rest of the items in the stack for push() and pop(). Again, be sure that you understand this implementation—it is the prototype for several implementations using linked structures that we will be examining later in this chapter. Using the abstract visual list representation to trace the code is the best way to proceed. Linked-list traversal. One of the most common operations we perform on collections is to iterate over the items in the collection. For example, we might wish to implement the toString() method that is inherent in every Java API to facilitate debugging our stack code with traces. For ArrayStackOfStrings, this implementation is familiar. 4.3 Stacks and Queues 575 Program 4.3.2 Stack of strings (linked list) public class LinkedStackOfStrings { private Node first; private class Node { String item; Node next; } first first node on list public boolean isEmpty() { return (first == null); } public void push(String item) { // Insert a new node at the beginning of the list. Node oldFirst = first; first = new Node(); first.item = item; first.next = oldFirst; } public String pop() { // Remove the first node from the list and return item. String item = first.item; first = first.next; return item; } public static void main(String[] args) { LinkedStackOfStrings stack = new LinkedStackOfStrings(); // See Program 4.3.1 for the test client. } } item next stack item next node on list This stack implementation uses a private nested class Node as the basis for representing the stack as a linked list of Node objects. The instance variable first refers to the first (most recently inserted) Node in the linked list. The next instance variable in each Node refers to the successor Node (the value of next in the final node is null). No explicit constructors are needed, because Java initializes the instance variables to null. % java LinkedStackOfStrings < tobe.txt to be not that or be Algorithms and Data Structures 576 StdIn StdOut to be or not to - to be - be - not that - that - or - be is ... Trace of LinkedStackOfStrings test client 4.3 Stacks and Queues 577 public String toString() { String s = ""; for (int i = 0; i < n; i++) s += a[i] + " "; return s; } This solution is the usual one, but it is inefficient because string concatenation takes time proportional to the length of the string. Our focus now is just on the process of examining every item. There is a corresponding idiom for visiting the items in a linked list. We initialize a loop-index variable x that references the first Node of the linked list. Next, we find the value of the item associated with x by accessing x.item, and then update x to refer to the next Node in the linked list, assigning to it the value of x.next and repeating this process until x is null (which indicates that we have reached the end of the linked list). This process is known as traversing the linked list, and is succinctly expressed in this implementation of toString() for LinkedStackOfStrings: public String toString() { String s = ""; for (Node x = first; x != null; x = x.next) s += x.item + " "; return s; } When you program with linked lists, this idiom will become as familiar to you as the idiom for iterating over the items in an array. At the end of this section, we consider the concept of an iterator, which allows us to write client code to iterate over the items in a collection without having to program at this level of detail. With a linked-list implementation we can write client programs that use large numbers of stacks without having to worry much about memory usage. The same principle applies to collections of any sort, so linked lists are widely used in programming. Indeed, typical implementations of the Java memory management system are based on maintaining linked lists corresponding to blocks of memory of various sizes. Before the widespread use of high-level languages like Java, the details of memory management and programming with linked lists were critical parts of any programmer's arsenal. In modern systems, most of these details are encapsulated in the implementations of a few data types like the pushdown stack, including the queue, the symbol table, and the set, which we will consider later in this chapter. If you take a course in algorithms and data structures, you will learn several others and gain expertise in creating and debugging programs that manipulate linked lists. Otherwise, you can focus your attention on understanding the role played by linked lists in implementing these fundamental data types. For stacks, they are significant because they allow us to implement the push() and pop() methods in constant time while using only a small constant factor of extra memory (for the links). Resizing arrays Next, we consider an alternative approach to accommodating arbitrary growth and shrinkage in a data structure that is an attractive alternative to linked lists. As with linked lists, we introduce it now because the approach is not difficult to understand in the context of a stack implementation and because it is important to know when addressing the challenges of implementing data types that are more complicated than stacks. The idea is to modify the array implementation (Program 4.3.1) to dynamically adjust the length of the array items[] so that it is sufficiently large to hold all of the items but not so large as to waste an excessive amount of memory. Achieving these goals turns out to be remarkably easy, and we do so in ResizingArrayStackOfStrings (Program 4.3.3). First, in push(), we check whether the array is too small. In particular, we check whether there is room for the new item in the array by checking whether the stack size n is equal to the array length items.length. If there is room, we simply insert the new item with the code items[n++] = item as before; if not, we double the length of the array by creating a new array of twice the length, copying the stack items to the new array, and resetting the items[] instance variable to reference the new array. 4.3 Stacks and Queues 579 Program 4.3.3 Stack of strings (resizing array) public class ResizingArrayStackOfStrings { private String[] items = new String[1]; private int n = 0; items[] n stack items number of items on stack public boolean isEmpty() { return (n == 0); } private void resize(int capacity) { // Move stack to a new array of given capacity. String[] temp = new String[capacity]; for (int i = 0; i < n; i++) temp[i] = items[i]; items = temp; } public void push(String item) { // Insert item onto stack. if (n == items.length) resize(2*items.length); items[n++] = item; } public String pop() { // Remove and return most recently inserted item. String item = items[--n]; items[n] = null; // Avoid loitering (see text). if (n > 0 && n == items.length/4) resize(items.length/2); return item; } public static void main(String[] args) { // See Program 4.3.1 for the test client. } } This implementation achieves the objective of supporting stacks of any size without excessively wasting memory. It doubles the length of the array when full and halves the length of the array to keep it always at least one-quarter full. On average, all operations take constant time (see the text). % java ResizingArrayStackOfStrings < tobe.txt to be not that or be Algorithms and Data Structures 580 StdIn StdOut n items.length items[] 0 1 2 3 4 5 6 7 ... Trace of ResizingArrayStackOfStrings test client Similarly, in pop(), we begin by checking whether the array is too large, and we halve its length if that is the case. If you think a bit about the situation, you will see that an appropriate test is whether the stack size is less than one-fourth the array length. Then, after the array is halved, it will be about half full and can accommodate a substantial number of push() and pop() operations before having to change the length of the array again. This characteristic is important: for example, if we were to use to policy of halving the array when the stack size is one-half the array length, then the resulting array would be full, which would mean it would be doubled for a push(), leading to the possibility of an expensive cycle of doubling and halving. Amortized analysis. This doubling-and-halving strategy is a judicious tradeoff between wasting space (by setting the length of the array to be too big and leaving empty slots) and wasting time (by reorganizing the array after each insertion). The specific strategy in ResizingArrayStackOfStrings guarantees that the stack never overflows and never becomes less than one-quarter full (unless the stack is empty, in which case the array length is 1). If you are mathematically inclined, you might enjoy proving this fact with mathematical induction (see Exercise 4.3.18). More important, we can prove that the cost of doubling and halving is always ab- 4.3 Stacks and Queues 581 sorbed (to within a constant factor) in the cost of other stack operations. Again, we leave the details to an exercise for the mathematically inclined, but the idea is simple: when push() doubles the length of the array to n, it starts with n / 2 items in the stack, so the length of the array cannot double again until the client has made at least n / 2 additional calls to push() (more if there are some intervening calls to pop()). If we average the cost of the push() operation that causes the doubling with the cost of those n / 2 push() operations, we get a constant. In other words, in ResizingArrayStackOfStrings, the total cost of all of the stack operations divided by the number of operations is bounded by a constant. This statement is not quite as strong as saying that each operation takes constant time, but it has the same implications in many applications (for example, when our primary interest is in the application's total running time). This kind of analysis is known as amortized analysis—the resizing array data structure is a prototypical example of its value. Orphaned items. Java's garbage collection policy is to reclaim the memory associated with any objects that can no longer be accessed. In the pop() implementation in our initial implementation ArrayStackOfStrings, the reference to the popped item remains in the array. The item is an orphan—we will never use it again within the class, either because the stack will shrink or because it will be overwritten with another reference if the stack grows—but the Java garbage collector has no way to know this. Even when the client is done with the item, the reference in the array may keep it alive. This condition (holding a reference to an item that is no longer needed) is known as loitering, which is not the same as a memory leak (where even the memory management system has no reference to the item). In this case, loitering is easy to avoid. The implementation of pop() in ResizingArrayStackOfStrings sets the array element corresponding to the popped item to null, thus overwriting the unused reference and making it possible for the system to reclaim the memory associated with the popped item when the client is finished with it. With a resizing-array implementation (as with a linked-list implementation), we can write client programs that use stacks without having to worry much about memory usage. Again, the same underlying ideas apply to collections of any sort. For some data types that are more complicated than stacks, resizing arrays are preferred over linked lists because of their ability to access any element in the array in constant time (through indexing), which is critical for implementing certain operations (see, for example, RandomQueue in Exercise 4.3.37). As with linked lists, it is best to keep resizing-array code local to the implementation of fundamental data types and not worry about using it in client code. 581 582 Algorithms and Data Structures Parameterized data types We have developed stack implementations that allow us to build stacks of one particular type (String). But when developing client programs, we need implementations for collections of other types of data, not necessarily strings. A commercial transaction processing system might need to maintain collections of customers, accounts, merchants, and transactions; a university course scheduling system might need to maintain collections of classes, students, and rooms; a portable music player might need to maintain collections of songs, artists, and albums; a scientific program might need to maintain collections of double or int values. In any program that you write, you should not be surprised to find yourself maintaining collections for any type of data that you might create. How would you do so? After considering two simple approaches (and their shortcomings) that use the Java language constructs we have discussed so far, we introduce a more advanced construct that can help us to properly address this problem. Create a new collection data type for each item data type. We could create classes StackOfInts, StackOfCustomers, StackOfStudents, and so forth to supplement StackOfStrings. This approach requires that we duplicate the code for each type of data, which violates a basic precept of software engineering that we should reuse (not copy) code whenever possible. You need a different class for every type of data that you want to put on a stack, so maintaining your code becomes a nightmare: whenever you want or need to make a change, you have to do so in each version of the code. Still, this approach is widely used because many programming languages (including early versions of Java) do not provide any better way to solve the problem. Breaking this barrier is the sign of a sophisticated programmer and programming environment, so it is worthwhile to make the effort to do so. Use collections of Object values. We could develop a stack whose items are all of type Object. Using inheritance, we can legally push an object of any type (if we want to push an object of type Apple, we can do so because Apple is a subclass of Object, as are all other classes). When we pop the stack, we must cast it back to the appropriate type (which might not always be an Object) to convert it to an Apple. In summary, if we create a class StackOfObjects by changing String to Object everywhere in one of our *StackOfStrings implementations, we can write code like 4.3 Stacks and Queues 583 StackOfObjects stack = new StackOfObjects(); Apple a = new Apple(); stack.push(a); ... a = (Apple) (stack.pop()); thus achieving our goal of having a single class that creates and manipulates stacks of objects of any type. However, this approach is undesirable because it exposes clients to subtle bugs in client programs that cannot be detected at compile time. For example, there is nothing to stop a programmer from putting different types of objects on the same stack, as in the following example: ObjectStack stack = new ObjectStack(); Apple a = new Apple(); Orange b = new Orange(); stack.push(a); stack.push(b); a = (Apple) (stack.pop()); // Throws a ClassCastException. b = (Orange) (stack.pop()); Type casting in this way amounts to assuming that clients will cast objects popped from the stack to the proper type, avoiding the protection provided by Java's type system. One reason that programmers use the type system is to protect against errors that arise from such implicit assumptions. The code cannot be type-checked at compile time: there might be an incorrect cast that occurs in a complex piece of code that could escape detection until some particular run-time circumstance arises. We seek to avoid such errors because they can appear long after an implementation is delivered to a client, who would have no way to fix them. Java generics. A specific mechanism in Java known as generic types solves precisely the problem that we are facing. With generics, we can build collections of objects of a type to be specified by client code. The primary benefit of doing so is the ability to discover type-mismatch errors at compile time (when the software is being developed) instead of at run time (when the software is being used by a client). Conceptually, generics are a bit confusing at first (their impact on the programming language is sufficiently deep that they were not included in early versions of Java), but our use of them in the present context involves just a small bit of extra Java syntax and is easy to understand. We name the generic class Stack and choose the generic 583 Algorithms and Data Structures 584 Program 4.3.4 Generic stack public class Stack<Item> { private Node first; private class Node { Item item; Node next; } first first node on list public boolean isEmpty() { return (first == null); } public void push(Item item) { // Insert item onto stack. Node oldFirst = first; first = new Node(); first.item = item; first.next = oldFirst; } public Item pop() { // Remove and return most recently inserted item. Item item = first.item; first = first.next; return item; } public static void main(String[] args) { Stack<String> stack = new Stack<String>(); // See Program 4.3.1 for the test client. } } item next stack item next node on list This code is almost identical to Program 4.3.2, but is worth repeating because it demonstrates how easy it is to use generics to allow clients to make collections of any type of data. The keyword Item in this code is a type parameter, a placeholder for an actual type name provided by clients. % java Stack < tobe.txt to be not that or be 4.3 Stacks and Queues 585 name Stack<Item> is identical to the code of LinkedStackOfStrings (we drop the Linked modifier because we have a good implementation for clients who do not care about the representation), except that we replace every occurrence of String with Item and declare the class with the following first line of code: public class Stack<Item> The name Item is a type parameter, a symbolic placeholder for some actual type to be specified by the client. You can read Stack<Item> as stack of items. When implementing Stack, we do not know the actual type of Item, but a client can use our stack for any type of data, including one defined long after we develop our implementation. The client code specifies the type argument Apple when the stack is created: Stack<Apple> stack = new Stack<Apple>(); Apple a = new Apple(); ... stack.push(a); If you try to push an object of the wrong type on the stack, like this: Stack<Apple> stack = new Stack<Apple>(); Apple a = new Apple(); Orange b = new Orange(); stack.push(a); stack.push(b); // Compile-time error. you will get a compile-time error: push(Apple) in Stack<Apple> cannot be applied to (Orange) Furthermore, in our Stack implementation, Java can use the type parameter Item to check for type-mismatch errors—even though no actual type is yet known, variables of type Item must be assigned values of type Item, and so forth. Autoboxing. One slight difficulty with generic code like Program 4.3.4 is that the type parameter stands for a reference type. How can we use the code for primitive types such as int and double? The Java language feature known as autoboxing and unboxing enables us to reuse generic code with primitive types as well. Java supplies built-in object types known as wrapper types, one for each of the primitive types: Boolean, Byte, Character, Double, Float, Integer, Long, and Short correspond to boolean, byte, char, double, float, int, long, and short, respectively. Java automatically converts between these reference types and the corresponding primitive types—in assignment statements, method arguments, and arithmetic/logic expressions—so that we can write code like the following: Stack<Integer> stack = new Stack<Integer>(); stack.push(17); // Autoboxing (int -> Integer). int a = stack.pop(); // Unboxing (Integer -> int). In this example, Java automatically casts (autoboxes) the primitive value 17 to be of type Integer when we pass it to the push() method. The pop() method returns an Integer, which Java casts (unboxes) to an int value before assigning it to the variable a. This feature is convenient for writing code, but involves a significant amount of processing behind the scenes that can affect performance. In some performance-critical applications, a class like StackOfInts might be necessary, after all. Generics provide the solution that we seek: they enable code reuse and at the same time provide type safety. Carefully studying Stack (Program 4.3.4) and being sure that you understand each line of code will pay dividends in the future, as the ability to parameterize data types is an important high-level programming technique that is well supported in Java. You do not have to be an expert to take advantage of this powerful feature. Stack applications Pushdown stacks play an essential role in computation. If you study operating systems, programming languages, and other advanced topics in computer science, you will learn that not only are stacks used explicitly in many applications, but they also still serve as the basis for executing programs written in many high-level languages, including Java and Python. Arithmetic expressions. Some of the first programs that we considered in Chapter 1 involved computing the value of arithmetic expressions like this one: (1 + ((2 + 3) * (4 * 5))) If you multiply 4 by 5, add 3 to 2, multiply the result, and then add 1, you get the value 101. But how does Java do this calculation? Without going into the details of how Java is built, we can address the essential ideas just by writing a Java program that can take a string as input (the expression) and produce the number represent- 4.3 Stacks and Queues ed by the expression as output. For simplicity, we begin with the following explicit recursive definition: an arithmetic expression is either a number or a left parenthesis followed by an arithmetic expression followed by an operator followed by another arithmetic expression followed by a right parenthesis. For simplicity, this definition is for fully parenthesized arithmetic expressions, which specifies precisely which operators apply to which operands—you are a bit more familiar with expressions like 1 + 2 * 3, in which we use precedence rules instead of parentheses. The same basic mechanisms that we consider can handle precedence rules, but we avoid that complication. For specificity, we support the familiar binary operators *, +, and -, as well as a square-root operator sqrt that takes only one argument. We could easily allow more operators to support a larger class of familiar mathematical expressions, including division, trigonometric functions, and exponential functions. Our focus is on understanding how to interpret the string of parentheses, operators, and numbers to enable performing in the proper order the low-level arithmetic operations that are available on any computer. Arithmetic expression evaluation. Precisely how can we convert an arithmetic expression—a string of characters—to the value that it represents? A remarkably simple algorithm that was developed by Edsgar Dijkstra in the 1960s uses two pushdown stacks (one for operands and one for operators) to do this job. An expression consists of parentheses, operators, and operands (numbers). Proceeding from left to right and taking these entities one at a time, we manipulate the stacks according to four possible cases, as follows: • Push operands onto the operand stack. • Push operators onto the operator stack. • Ignore left parentheses. • On encountering a right parenthesis, pop an operator, pop the requisite number of operands, and push onto the operand stack the result of applying that operator to those operands. After the final right parenthesis has been processed, there is one value on the stack, which is the value of the expression. Dijkstra's two-stack algorithm may seem mysterious at first, but it is easy to convince yourself that it computes the proper value: anytime the algorithm encounters a subexpression consisting of two operands separated by an operator, all surrounded by parentheses, it leaves the result of performing that operation on those operands on the operand stack. The result is the same as if that value had appeared in the input instead of the subexpression, so we can think of replacing the subexpression by the value to get an expression that 587 Algorithms and Data Structures 588 Program 4.3.5 Expression evaluation public class Evaluate { public static void main(String[] args) ops operator stack { values Stack ops = new Stack(); operand stack Stack values = new Stack(); token current token while (!StdIn.isEmpty()) v current value { // Read token, push if operator. String token = StdIn.readString(); if (token.equals("(")) ; else if (token.equals("+")) ops.push(token); else if (token.equals("-")) ops.push(token); else if (token.equals("*")) ops.push(token); else if (token.equals("sqrt")) ops.push(token); else if (token.equals(")")) { // Pop, evaluate, and push result if token is ")". String op = ops.pop(); double v = values.pop(); if (op.equals("+")) v = values.pop() + v; else if (op.equals("-")) v = values.pop() - v; else if (op.equals("*")) v = values.pop() * v; else if (op.equals("sqrt")) v = Math.sqrt(v); values.push(v); } // Token not operator or paren: push double value. else values.push(Double.parseDouble(token)); } StdOut.println(values.pop()); } } This Stack client reads a fully parenthesized numeric expression from standard input, uses Dijkstra's two-stack algorithm to evaluate it, and prints the resulting number to standard output. It illustrates an essential computational process: interpreting a string as a program and executing that program to compute the desired result. Executing a Java program is nothing other than a more complicated version of this same process. % java Evaluate (1 + ((2 + 3) * (4 * 5))) 101.0 % java Evaluate ((1 + sqrt (5.0)) * 0.5) 1.618033988749895 4.3 Stacks and Queues (1+((2+3)*(4*5))) 1 +((2+3)*(4*5))) 1 + ((2+3)*(4*5))) 1 2 + +3)*(4*5))) 1 2 3 + +)*(4*5))) 1 5 + *(4*5))) 1 5 + * (4*5))) 1 5 4 + * *5))) 1 5 4 + * * 5))) 1 5 4 5 + * *))) 1 5 20 + *)) 1 100 +) 101 Trace of expression evaluation (PROGRAM 4.3.5) 589 would yield the same result. We can apply this argument again and again until we get a single value. For example, the algorithm computes the same value of all of these expressions: (1 (1 (1 (1 101 + + + + (((5 (5 100 2 + 3) * (4 * 5))) * (4 * 5))) * 20))) (Program 4.3.5) is an implementation of this algorithm. This code is a simple example of an interpreter : a program that executes a program (in this case, an arithmetic expression) one step or line at a time. A compiler is a program that translates a program from a higher-level language to a lower-level language that can do the job. A compiler's conversion is a more complicated process than the step-by-step conversion used by an interpreter, but it is based on the same underlying mechanism. The Java compiler translates code written in the Java programming language into Java bytecode, Originally, Java was based on using an interpreter. Now, however, Java includes a compiler that converts arithmetic expressions (and, more generally, Java programs) into lower-level code for the Java virtual machine, an imaginary machine that is easy to simulate on an actual computer. Evaluate Algorithms and Data Structures 590 Stack-based programming languages. Remarkably, Dijkstra's two-stack algorithm also computes the same value as in our example for this expression: (1 ((2 3 +) (4 5 *) *) +) In other words, we can put each operator after its two operands instead of between them. In such an expression, each right parenthesis immediately follows an operator so we can ignore both kinds of parentheses, writing the expressions as follows: 1 2 3 + 4 5 * * + This notation is known as reverse Polish notation, or postfix. To evaluate a postfix expression, we use only one stack (see Exercise 4.3.15). Proceeding from left to right, taking these entities one at a time, we manipulate the stack according to just two possible cases, as follows: • Push operands onto the stack. 1 2 3 + 4 5 * * • On encountering an operator, pop the requisite 1 2 3 + 4 5 * * + number of operands and push onto the stack the 1 2 3 + 4 5 * * + result of applying the operator to those operands. 1 2 3 + 4 5 * * + Again, this process leaves one value on the stack, which is the value of the expression. This representation is 1 5 4 5 * * + so simple that some programming languages, such as 1 5 4 5 * * + Forth (a scientific programming language) and Post1 5 4 5 * * + Script (a page description language that is used on most 1 5 20 * + printers) use explicit stacks as primary flow-control structures. For example, the string 1 2 3 + 4 5 * * + 1 100 + is a legal program in both Forth and PostScript that 101 leaves the value 101 on the execution stack. AficionaTrace of postfix evaluation dos of these and similar stack-based programming languages prefer them because they are simpler for many types of computation. Indeed, the Java virtual machine itself is stack based. Function-call abstraction. Most programs use stacks implicitly because they support a natural way to implement function calls, as follows: at any point during the execution of a function, define its state to be the values of all of its variables and a pointer to the next instruction to be executed. One of the fundamental characteristics of computing environments is that every computation is fully determined by its state (and the value of its inputs). In particular, the system can suspend a computation by saving away its state, then restart it by restoring the state. If you take a + 4.3 Stacks and Queues public static void sort(a, 0, 4) { int n = 4 - 0; if (n 2 no integers all integers amino acid encodings AAA AAC AAG AAT ACA ACC ACG ACT TAC TAT TGC TGG TGT TAA TAG TGA AAAAAAAAA ABCDE U.S. telephone number (609) 258-3000 (800) 555-1212 (99) 12-12-12 2147483648 English words and middle computability abc niether misunderestimate legal English sentences This is a sentence. I think I can. xya a b.c.e?? Cogito ergo sum. legal Java identifiers a class $xyz3_XYZ 12 123a a((BC))* legal Java programs public class Hi { public static void main(String[] args) { } } int main(void) { return 0; } More examples of formal languages, over various alphabets 722 Theory of Computing to be empty, but you might have a hard time convincing yourself of that, since it is a form of the famous Fermat's Last Theorem, which remained an unproven conjecture for more than 300 years, until Andrew Wiles developed a proof in the 1990s. Others are based on the English language, or Java, or genomics. We could define a language Shakespeare to be the set of typescripts of Shakespeare's plays, or a language Taylor to be all .wav files that are Taylor Swift songs, or a language Stars to be coordinates of stars in the universe, or whatever. All that is needed is a precise definition of a set of strings. Specification problem. Our informal English-language descriptions do the job in some cases (for example, palindromes and primes), but are rather inadequate in others (for example, English sentences and Java programs). Why not just work with precise, complete definitions? That is the crux of the matter! Our goal is to work with precise, complete definitions of formal languages. This task is known as the specification problem for formal languages. How do we completely and precisely define formal languages? Our informal descriptions give some indication, but understanding those depends, for one thing, on understanding the English language, which is not so precisely defined itself. Our descriptions make our meaning clear for some of the examples that we have considered, but are woefully incomplete. For instance, should we specify that prime integers have no leading zeros, or is it fine to consider strings like 000017 to be in the language? Note that this decision would lead to an infinite number of strings corresponding to each prime number. Such details make defining a formal language a challenging task. But the fundamental challenge is not just in the details; that is the crux of the specification problem. As it turns out, we can identify certain classes of formal languages for which the specification problem is easily addressed. We will do so for a fundamental class known as the regular languages shortly. Recognition problem. Once we have a way to specify a language, we are left with the following problem: given a language L and a string x, answer the following question: Is x in L, or not? This task is called the recognition problem for formal languages. To solve the recognition problem, you would need to use a computer (how else could you tell whether a billion-bit string is a palindrome or has equal numbers of as and bs? Moreover, you would need to understand mathematics, natural language syntax, and untold other fields of knowledge. Fortunately, again, we can identify classes of formal languages for which we can solve the recognition problem and which are also quite useful, as well. 5.1 Formal Languages We will begin our study of formal languages by studying an important class of languages (regular languages) for which we can develop simple solutions to both the specification and the recognition problems. Then we will see that these solutions are critically important tools that are widely used in practical applications involving natural languages like English, programming languages like Java, genetic codes, and many, many other domains. Following that, in Section 5.2, we return to fundamental theoretical questions. Remarkably, a few relatively simple extensions to the mechanisms that we use to solve the specification and recognition problems for regular languages both vastly expand the formal languages that we can handle and take us straight to the heart of the foundational principles of computation. The relationship between formal languages and computation is profound, indeed. The mechanisms that we will describe for specifying formal languages are simple, compact, and elegant. They follow one of two basic approaches. The first is language based: we assign meaning to a few characters beyond those in the language to allow us to write character strings that specify languages. The second is machine based: we describe a class of abstract machines, each of which can solve the specification and recognition problems for a language. Regular languages To introduce these topics, we consider a class of formal languages known as the regular languages. To solve the specification and recognition problems for regular languages, we develop a language-based approach and two different machine-based approaches. Then, at the end of this section, we consider relationships among the three. Basic operations. Since a formal language is a set of strings, we can use basic operations on sets to give an efficient mechanism for specifying formal languages. In particular, we use the union, concatenation, and closure operations, considered in the next three paragraphs. A string is in the union of two sets of strings if and only if it is in one or both of them. We use the notation R | S to denote the union of two formal languages R and S. For example, { a, ba } | { ab, ba, b } = { a, ab, ba, b } We can list the member strings of a language in any order but we do not include duplicates in the union. The concatenation of two strings is the string formed by appending the second of the two strings to the first. For example, the concatenation of abb and aab 723 724 Theory of Computing is abbaab. More generally, the concatenation RS of two formal languages R and S is the set of all strings that can be created by appending a string from R to a string from S. For example, { a, ab } { a, ba, bab } = { aa, aba, abab, abba, abbab } Again, we do not include duplicates in the result. (In this case, aba could be formed by appending ba to a or by appending a to ab.) The closure of a language is the concatenation of zero or more strings taken from the language. If R is a nonempty language, the notation R* specifies an infinite number of strings: R* = | R | RR | RRR | RRRR | RRRRR | RRRRRR ... Note that each time we take a string from R, we are free to use any string in the set. For instance, (a|b)* specifies the set of all binary strings. Here, refers to the empty string—the string consisting of 0 characters. If we have a regular expression containing more than one operator, in which order should the operators be applied? As with arithmetic expressions, we use parentheses or rely on a defined operator precedence order to resolve such ambiguities. For REs, closure is performed before concatenation, and parenthesized RE concatenation is performed before union. For example, GCG(CGG|AGG)*CTG the regular expression b|ab specifies the set {b, ab}, not the set {bb, ab}. As with arithmetic expressions, we can closure union use parentheses to override the precedence rules. For example, we can represent the set {bb, ab} with the regular expression (b|a)b. Similarly, the regular expression a|b* Anatomy of a regular expression specifies the same set as a|(b*), not (a|b)*. Regular expressions. A regular expression (RE) is a string of symbols that specifies a formal language. We define what regular expressions are (and what they mean) recursively, using the union, concatenation, and closure operations on sets, along with parentheses for operator precedence. Specifically, every regular expression is either an alphabet symbol, specifying the singleton set containing that symbol, or composed from the following operations (where R and S are REs): • Union: R | S, specifying the union of the sets R and S, • Concatenation: RS, specifying the concatenation of the sets R and S, • Closure: R*, specifying the closure of the set R, • Parentheses: (R), specifying the same set as R. 5.1 Formal Languages This definition implicitly assumes that the language's alphabet does not include the symbols |, *, (, and). We refer to these symbols as metasymbols and will discuss later how to deal with languages that contain them. Regular languages. The recursive definition not only gives us a way to build up arbitrarily complex REs, but also precisely defines what they mean. Every regular expression fully specifies some formal language, but not every formal language can be specified with some RE. Later, we will discuss a formal proof that demonstrates this fact by highlighting a language that cannot be specified with any RE. But the class of languages that can be specified by some RE is sufficiently important that it has a name: Definition. A language is regular if and only if it can be specified by an RE. The table on the next page gives examples of regular languages, accompanied by REs that describe them and examples of strings in the language and strings not in the language. How can we be sure that each RE specifies the same language as does the informal description? In general, we need to prove that fact for each RE, and developing such proof can often be a challenge. The significance of REs is that they can allow us to move away from the informal descriptions because they provide a way to specify languages that is both natural for an important class of applications and rigorous. Regardless of our confidence in the accuracy of the informal specification, each RE is a precise and complete specification of some regular language. Indeed, we can just write down a well-formed RE like (ab*|aba)*(ab*a|b(a|b))* without necessarily having any idea which language it specifies. To familiarize yourself with REs, you should take the time to convince yourself that each RE in the table does specify the language as claimed, first by verifying not only that the strings identified as being in the language both fit the informal description and are specified by the RE, but also that the strings identified as not being in the language do not fit the informal description and are not specified by the RE. Your goal is to understand why all strings in the language and only those strings are specified by the RE. More detail on some of the languages is given in the paragraphs that follow. 725 Theory of Computing 726 regular language regular expression in the language not in the language fifth-to-last symbol is a (a|b)*a(a|b)(a|b)(a|b)(a|b) aaaaa bbbabbbb bbbbbbababababa a bbbbbbbba aaaaaaaaaaaabaaaa contains the substring abba (a|b)*abba(a|b)* abba aababbabbababbba bbbbbbbbabbabbbbbb abb bbabaab aaaaaaaaaaaaaaaa does not contain the substring bbb (bba|ba|a*)*(a*|b|bb) aa abababababa aaaaaaaaaaaaab bbb abababbabab bbbbbbbbbbbbbb number of b symbols is a multiple of 3 a*|(a*ba*ba*ba*)* bbb aaa bbbaababbaa b baaaaaaab baabbbaaaaab positive integer divisible by 5 5|(1|2|...|9)(0|1|...|9)*(0|5) 5 200 9836786785 1 0005 3452345234 positive ternary number (1|2)(0|1|2)* 11 2210221 011 19 9836786785 contains the trigraph spb (a|b|c|...|z)*spb(a|b|c|...|z) raspberry crispbread subspace subspecies uses only the top row of the keyboard (q|w|e|r|t|y|u|i|o|p)* typewriter reporter alfalfa paratrooper GCG(CGG|AGG)*CTG GCGCTG GCGCGGCTG GCGCGGAGGCTG GCGCGG CGGCGGCGGCTG GCGCAGGCTG binary alphabet decimal digits lowercase letters genetic code fragile X syndrome pattern Examples of regular expressions, over various alphabets 5.1 Formal Languages When solving a crossword puzzle or playing a word game, you might be faced with a question like this: "What is an 8-letter word whose middle two letters are hh?" Once you learn about the mechanisms described in this section for using regular expressions, you will address crossword puzzles and word games from an entirely different point of view. In genomics, REs over the alphabet { A, T, C, G } are used to describe properties of genes. For example, the human genome has a region that can be described with the regular expression GCG(CGG|AGG)*CTG, where the number of repeats of the CGG/AGG pattern is highly variable among individuals. A certain genetic disease that can cause mental retardation and other symptoms is known to be associated with a high number of repeats. REs are widely used in practice to address important scientific problems of this sort. In information processing, we are always concerned with describing information in a precise and complete manner. For instance, when you type your name, address, and other information into a form on the web, the first action of the program that processes the information is to check that what you typed makes sense. If you typed letters where it expects numbers or a dollar sign where it expected a phone number, it will flag the error and ask you to fix it. We will soon see how to use REs to describe familiar low-level abstractions such as dates, credit-card numbers, and Java identifiers to extract relevant data from databases (for example, results from scientific experiments). In computer science, REs are pervasive. As you will see, nowadays they are featured in numerous applications—Java itself has numerous RE-based capabilities. They also represent the first step in the process of compiling programs written in high-level languages such as Java into machine language. Even more important, as we repeatedly emphasize, REs are the first step down the path of addressing fundamental questions about computation. In general, we know that any language that can be specified with an RE is regular— that is the definition. But what about languages that are specified in some other manner? Is the set of all palindromes a regular language? One of our goals in this section is to develop understanding of how to classify languages. While there are plenty of examples of interesting and useful regular languages, there are also many interesting and useful languages that are not regular. Later, we consider the idea of more powerful specification systems than REs to address non-regular languages. 727 728 Theory of Computing Recognition problem for REs. As we have mentioned, another fundamental problem is the following: Given a binary string, how do we know whether it is in the language specified by a given RE? For example, is the string abaaabbbbbbabaabbba in the language specified by (ab*|bab)*(bb*b|(b(a|b)))*? This is an example of the recognition problem for regular languages. The significance of REs is that they allow us to precisely define the problem for a large class of languages (regular languages), but they do not solve it. Certainly, we need to address this problem for regular languages before we can begin to address more difficult classes of languages such as the set of all primes or the set of all Java programs. We will use abstract machines to develop a solution to the recognition problem for regular languages later in this section. Actually, a version of this solution is implemented in the matches() method in Java's String library, so we will begin by describing how to use that. If text is any Java String and regexp is any regular expression, then text.matches(regexp) is true if text is in the language specified by regexp and false otherwise. Program 5.1.1 uses this method to solve the recognition problem: it takes an RE as a command-line argument and, for each string on standard input, prints Yes if it is in the language specified by the RE, and No if it is not. The ability to perform such checks so easily is so useful that we will consider some extensions and generalizations before returning to the theory, to better familiarize you with REs while at the same time informing you about some indispensable programming tools in Java's standard libraries. Beyond the recognition problem, the complete and precise specification afforded by REs leads immediately to several other natural and well-specified problems. For instance, given two REs, how can we check whether they specify the same language? It is easy to check that a(b|ab|aab) and (a|aa)(b|ab) specify the same language, but does ((abb|baaa)*(ba*|b*a))* specify the same language as (ab*|bab)*(bb*a|a(b))*? This problem is the equivalence problem for regular languages. How would you approach it for some arbitrarily complicated pair of regular expressions that run to (say) thousands of symbols? Similarly, when we are asking for 8-letter words from the dictionary whose middle two letters are hh, we are asking for the intersection of two languages: strings that are in both. REs allow us to precisely define problems like these, but solving them is another matter entirely. If you find these problems to be fascinating (as many people do), you are likely to enjoy further study in the theory of computing. This is merely the tip of the iceberg. 5.1 Formal Languages Program 5.1.1 Validity checking (RE recognition) public class Validate { public static void main(String[] args) { String regexp = args[0]; while (!StdIn.isEmpty()) { String text = StdIn.readString(); if (text.matches(regexp)) StdOut.println("[Yes]"); else StdOut.println("[No]"); } } } This program illustrates a basic mechanism in Java for using REs: the matches() method from Java's String library, which solves the recognition problem for REs. The main() method here takes an RE as a command-line argument and then, for each string on standard input, prints Yes if the string is in the language described by the RE, and No otherwise. % java Validate "(a(b)*a(a|b)b(a|b)*a)b" bbbabbbb [Yes] bbbbbbbbba [Yes] baabbbbbaaaa [No] % java Validate "a*|(a*ba*ba*ba*)*" bbbaababbaa [Yes] baabbbaaaaab [No] % java Validate "GCG(CGG|AGG)*CTG" GCGCGGAGGCTG [Yes] CGGCGGCGGCTG [No] 729 730 Theory of Computing Generalized REs Our definition of REs is a minimal one that includes the four basic operations that characterize regular languages (concatenation, union, closure, and grouping). In practice, it is useful to make various additions to this set. Here, we give brief descriptions of RE extensions that are found in Java, which fall into three main categories. • Expanding the alphabet • Shorthand notations for the union operation • Extensions to the closure operation For brevity, we refer to Java's generalized regular expressions as generalized REs/Java REs. Similar mechanisms are widely defined, in other languages and in other applications, but the exact definition of what constitutes "generalized" is not uniformly applied. For proper usage, the one characteristic that all generalizations should have in common is that every generalized RE should describe a regular language. That is, you can (in principle) translate any generalized RE into a (more cumbersome) standard RE (the ones we have been considering). The restriction is ironic, because it actually says that we are not generalizing regular languages, just the language that we use to describe regular languages. In due time, we will consider true generalizations. In the meantime, caveat emptor —because many systems (including Java) support RE extensions that do not adhere to the restriction. We will return to this topic later. Expanding the alphabet. The alphabet is the most obvious generalization. Symbols in Java REs are Unicode characters, without restriction. But an inherent problem arises when we extend the alphabet fully in this way—we need escape mechanisms to allow us to use the metasymbols |, *, (, and) both to specify REs and as symbols in the language alphabet. Specifically, \| matches |, * matches *, \(matches (, \) matches), and \\ matches \. All the other metasymbols used in the extensions described later all also need the \ escape when used as language symbols. Shorthand notations. Having a large number of symbols in the alphabet immediately creates the need for shorthand notations for the union operation so that we can specify groups of symbols with a few keystrokes. For example, the wildcard symbol (.) matches any alphabet symbol; it is shorthand for a long sequence of union operations with all the symbols in the alphabet. For example, when working with decimal digits, we certainly prefer typing a single . to typing 0|1|2|3|4|5|6|7|8|9. Generalized REs support a number of similar notations, such as the following: 5.1 Formal Languages 731 • The metasymbol ^ matches the beginning of a line and $ matches the end of a line. • A list or range of symbols enclosed in square brackets [] matches any symbol in the list or range. • If the first character within the square braces is the ^ character, the specification refers to the Unicode characters not in the list or range. • Several escape sequences consisting of a backslash followed by an alphabet symbol match a defined set of symbols. For example, \s matches any whitespace symbol. For example, ^text$ represents the word text appearing on a line by itself, [a-z] represents lowercase letters, [0-9] represents decimal digits, [^aeiou] represents characters that are not lowercase vowels, and [A-Z][a-z]* represents capitalized words. Extensions to the closure operator. The closure operation is often too general to be useful directly in practice. Accordingly, Java REs have the following options for specifying restrictions on the number of repetitions: • One or more: + • Zero or one: ? • Exactly n: {n} • Between m and n: {m, n} For example, [^aeiou]{6} specifies six-letter words that have no lowercase vowels, such as rhythm and syzygy. Each of these notations is shorthand for a standard RE specification, though the RE might be very long. For example, a generalized RE like [^aeiou] is shorthand for a long sequence of union operations for all the other characters in the alphabet, and [^aeiou]{6} is shorthand for six copies of that RE. In principle, you might imagine C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H for the sequence of strings on standard input, without using Java's RE mechanism. Anatomy of a generalized regular expression in practice, each extension is an opportunity for ingenuity in algorithm design. You will encounter numerous Java REs from this point forward and a number of examples are given on page 733. As before, your task is to study each one, check why the examples given are in the specified language and why other strings are not, and try to understand how the REs do their job. 732 Theory of Computing Applications How do we make use of REs in practical applications? They arise in numerous scenarios, some of which we shall now consider. Fundamentally, these scenarios are all based on achieving economies by using relatively short REs to specify relatively large (even infinite) sets of strings. With REs, a person or a program can treat a whole set of strings as a single entity. Validity checking. Perhaps you need to determine that some combination of symbols is a legal word. You have that you are frequently encounter RE recognition when you use the web. When you type in a date or an account number on a commercial website, the input-processing program has to check that your response is in the right format. One approach to performing such a check is to write code that checks all the cases: if you were to type in a dollar amount, the code might check that the first symbol is a $, that the $ is followed by a set of digits, and so forth. A better approach is to define an RE that describes the set of all legal inputs. Then, checking whether your input is legal is precisely the % java Validate "\\$[1-9]\\d*(\\.\\d\\d)?" $22.99 is the string you type in [Yes] $1,000,000.00 the program should take [No] the RE? libraries of REs % java Validate "ATG(...)*(TAG|TAA|TGA)" for common checks Java ATGCGCCTGCGTCTGTACTAG sprung up on the web as [Yes] this type of checking has ATGATGTTAG [Yes] CTGATAG [No] come into widespread use. Typically, an RE is a much more precise and concise expression of the set of all valid strings than would be a program that checks all the cases. Several of the examples on the facing page illustrate this point. Computational biology. As we have already seen, researchers have developed a number of encodings to facilitate the manipulation and analysis of genetic data. The genetic code (where the ATCG encoding of the four bases is) and protein defined using 20 character alphabet ACDEFGHIKLMNPQRSTVWY for amino acids. The scientific details are not relevant in this context, but here is a specific example that makes use of this code: A C2H2-type zinc finger domain signature is defined to be 5.1 Formal Languages 733 regular expression in the language Java identifiers [partial] $[a-zA-Z][$_a-zA-Z0-9]* $_$ System email address [partial] [a-zA-Z] [a-zA-Z]+\.(...) U.S. Social Security numbers [0-9]{3}-[0-9]{2}-[0-9]{4} 330-12-3456 213-44-5689 U.S. telephone numbers \([0-9]{3}\) [0-9]{3}-[0-9]{4} (800) 555-1212 (609) 258-4345 dollar amounts [partial] $[1-9][0-9]*\.[0-9][0-9] $22.99 $1000000.00 $0.01 Unicode decimal digits gene code ATGATAGTGATGTGA potential genes ATG(...)*TAG|TAA|TGA) ATGAATGTAG ATGCGCCTGCGTCTGTACTAG C2H2-type zinc finger domain signature C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H CCCCCCCCCCCCCCCCHHHHH CAASCGGFYACGGWAGYHAGWH Examples of generalized regular expressions, over various alphabets Theory of Computing 734 • C followed by two, three, or four amino acids, followed by • C followed by three amino acids, followed by • L, I, V, M, F, Y, W, C, or X followed by eight amino acids, followed by • H followed by three, four, or five amino acids, followed by • H. For example, the string CAASCGGPYACGGWAGYHAGWH is a C2H2-type zinc finger domain signature because it consists of • C followed by AAS (between two and four amino acids), followed by • C followed by GGP (three amino acids), followed by • Y (an amino acid in the set {L, I, V, M, F, Y, W, C, X}), followed by ACGGWAGY (eight amino acids), followed by • H followed by AGW (between three and five amino acids), followed by • H. With this generalized RE, we can just use Program 5.1.1 to test whether any given string is a C2H2-type zinc finger domain signature: % java Validate "C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H" CAASCGGPYACGGWAGYHAGWH [Yes] CAASCGGPYACGGWAGYHAGWH [Yes] Searching. On modern computer systems, it seems that we are always searching. You are surely familiar with searching on the web, looking for a particular item in a large database, or searching for a file on your computer. As a result, searching for strings is a built-in capability in all sorts of applications. Most likely, you would type in a string and look for a match—with regular expressions, you can have searches that are both more flexible and more precise. For computer scientists, the quintessential software tool is a searching tool known as grep that was developed by Ken Thompson for the Unix system in the 1970s and still available as a command on most computer systems. Program 5.1.2 on page 736 is an implementation of grep that uses Java's built-in tools. It is a filter that takes a regular expression specified as the command-line argument and prints to standard output the lines on standard input having a substring described by the RE. 5.1 Formal Languages You can use Grep (or the built-in grep) to quickly accomplish all sorts of searching tasks, such as: • Finding all lines of code containing a particular identifier • Finding all words in the dictionary matching a certain pattern • Finding part of a genome containing certain patterns • Finding movies in which a certain actor appeared • Extracting lines from a data file having specific characteristics • And many, many others The exercises in this section provide examples of search in a wide variety of domains, several of which are certain to be of interest to you. Experienced programmers are accustomed to having it still find grep to be an indispensable tool. The search function is now built into many high-level applications, though full regular expression matching is often overlooked by users. Validating and searching through information are all fundamental operations that are easier to do with generalized REs than without them. These examples are merely indicative—understanding and using REs is a necessity for anyone who wishes to make effective use of computation. We are awash in data in modern scientific, engineering, and commercial applications, so that skill in applying patternmatching tools can play an important role in moving forward. As illustrated by our implementations in Program 5.1.1 and Program 5.1.2, Java's tools make it easy to include regular expression searching in an application (and the same is true of other modern programming environments), so you are likely to find REs in increasing numbers of applications. More important, REs illustrate a fundamental paradigm. The idea of using a formal mechanism to specify a language and then recognizing strings using a specification of the language is a basic concept. To fully understand it, you need to understand how the recognition process actually works. To use REs effectively, we need a solution to the recognition problem. Think for a moment about how you would implement Program 5.1.1 without using Java's matches() method, even for basic REs on a binary alphabet. How can we write a program that takes an RE and a string as input and then determines whether the string is in the language defined by the RE? Addressing this question takes us down the road toward fundamental concepts at the heart of computer science. 735 Theory of Computing 736 Program 5.1.2 Generalized RE pattern match public class Grep { public static void main(String[] args) { String regexp = args[0]; String s = ".*" + regexp + ".*"; while (!StdIn.isEmpty()) { String line = StdIn.readLine(); if (line.matches(s)) System.out.println(line); } } } This program implements the classic software tool grep: it takes an RE as its argument and prints to standard output all of the lines in standard input having a substring in the language described by the RE. For simplicity, this implementation uses the matches() method from Java's String library; Java's Pattern and Matcher classes lead to a more efficient implementation for complex REs or huge files (see Exercise 7.2.3). The file words.txt, available on the booksite, contains all of the words in the dictionary, one per line. See Section 4.5 for a description of the file movies.txt. % java Grep class < Grep.java public class Grep % more words.txt ... % java Grep "^[fo]{8}$" < words.txt text % java Grep "^...$hi" ... % java Grep "Bacon, Kevin" < movies.txt | java Grep "Sedgwick, Kyra" Murder in the First (1995)/.../Bacon, Kevin/.../Sedgwick, Kyra Woodsman, The (2004)/.../Bacon, Kevin/.../Sedgwick, Kyra 5.1 Formal Languages 737 Abstract machines To address problems such as the recognition problem for REs, we consider a simple computational model. At first, this model will seem to you to be totally unrelated to REs, but rest assured that it is the first step on a path toward solving these problems. An abstract machine is a mathematical model of computation. In general, an abstract machine is nothing more than a mathematical function that maps an input string to a single output bit. We normally consider a slightly more detailed model that considers the input symbols individually. Specifically, for any given formal language, we imagine a device that interfaces with the outside world via An abstract machine with its important input/output device that can read (and perhaps write) one symbol at a time from a tape, and at least two status indicator lights (yes and no). When we load the machine by putting a given string on the tape and turning it on, we expect the machine to read the input tape and the yes indicator to light up if the string is in the machine's formal language and the no indicator to light up if it is not. We say that the machine accepts the string if the yes indicator lights up and rejects or does not match the string if the no indicator lights up. Thus, an abstract machine specifies a formal language (the set of all strings it accepts) and presents an abstract solution to the recognition problem for that language (put a string on the tape, turn it on and see whether the yes indicator lights), and we say that a machine recognizes a language. We will be considering a number of abstract machines that differ in their capabilities to manipulate the tape and the availability of other input/output devices and indicator lights. We are focusing now on the use of abstract machines in the theory of computing, but you should also take note that this focus is justified because abstract machines are appropriate for modeling practical computations of all kinds. For example, when you use an automated teller machine, your sequence of button pushes corresponds to an input string, the accept indicator corresponds to your money being dispensed, and the reject indicator to your getting an insufficient funds notice. Or we might consider the input string to be a decimal integer and expect the machine to light the yes indicator if and only if the integer is prime (or perhaps to write its largest factor on the tape). The bird's-eye view of a Java program that we considered at the beginning of this book is another example. The idea is so general as to encompass any kind of computing that we might encounter. We will consider other, more specific examples throughout this chapter. NO YES Theory of Computing 738 How does an abstract machine decide whether to light the yes or no indicator? We expect that it has to read some or all of the input and to do some computing. What kind of computing? Each type of abstract machine is characterized by the basic operations that we define for it. Our goal is to strip away nonessential details and focus on types of machines that use a set of basic operations that is as simple as possible. Abstract machines are special-purpose computers. In this section we focus on a model where each machine does one particular task, or solves one problem: the recognition problem for some regular language. In due time, we shall see how the basic operations can be used to make more powerful machines. It is a natural question to ask whether any abstract machine has the ability to solve any problem that we might want it to solve. Abstract machine models also provide us with the opportunity to rigorously prove facts about their behavior. Even more importantly, and more surprisingly, as you will see, we are able to address questions that are of very general applicability. Deterministic finite-state automata Certainly, it would seem that any so-called state can be implemented in some way. We begin by studying a kind of abstract machine that can do little else, which is known as the deterministic finite-state automaton. Each DFA consists of • A finite number of states, each of which is designated as either an accept state or a reject state. • A set of transitions that specify how the machine changes state. Each state has one transition for each symbol in the alphabet. • A tape reader initially positioned at the first symbol of an input string and capable only of reading a symbol and moving to the next symbol. We represent each DFA as a directed graph where accept states are vertices labeled Yes, reject states are vertices labeled No, and each transition is a directed edge that is labeled with a symbol from the alphabet. To identify the vertices, we number them with integers starting at 0. An example of a transition DFA over a binary alphabet is shown here. This DFA uses the symbols a and b for the NO reject a b 0 a 1 a 2 state symbols, to avoid confusion with vertex indices. For example, if we want to make the same transition for multiple symbols, we simply label the edge with a string tape reader comprising all the symbols that should cause the transition. A deterministic finite-state automaton 5.1 Formal Languages 739 All DFAs start at state 0 with an input string on the tape and the tape head on the leftmost symbol in the input string. The machine operates by changing state in discrete steps according to the following rule: read a symbol, move the tape head right one position, and then change state as specified by the transition labeled with the input symbol just read. When the input is exhausted, the DFA halts operation. If it is in an accept state at that point, the YES indicator lights up; if it is in a reject state, the NO indicator lights up. For example, our sample DFA, given the input baaab, will behave as follows (as illustrated at right): Starting in state 0, it reads the first b and moves to state 1. Then it reads the second symbol and moves to state 2 because that symbol is b. Then it stays in state 2 because the third and fourth symbols are a and moves to state 0 because the fifth symbol is b. The machine is in state 0 when the input is exhausted. State 0 is labeled Yes, so it activates the YES light. In other words, the machine accepts the binary string bbaab. This trace also shows that the machine rejects the binary strings b, bb, bba, and bbaa. A much more compact notation for the trace is the sequence of state transitions the DFA would make in trying to decide whether to accept or reject that binary string, with the final state dictating whether the binary string is accepted or rejected. For our example, the sequence is 0 -> 1 -> 2 -> 2 -> 2 -> 0 is a trace of this DFA for the input string bbaab. Note that the input string can also be seen from the sequence of edge labels. 739 Tracing the operation of a DFA a b 0 1 2 Yes No No b No b YES b a a b b a b a N o N o 0 No a 1 Yes b b a a b b a a a b b a N o 0 No b No No No YES b b a a b b a b a b Characterizing the language. Which binary strings are in the language recognized by a given DFA? To answer this question, we need to study the machine to discover structure. For example, this machine's b and bb not accepts; it rejects bbbb and NO 0 Yes a 1 No b a 2 No b YES b a a b b a b a b Tracing the operation of a DFA Theory of Computing 740 bbbbb accepts bbbbbb (and it ignores all the as), so it's reasonable to leap to the conclusion that this DFA recognizes the language number of b's is a multiple of 3. It is not difficult to establish this fact by developing a full proof, using induction of the number of bits read, that the machine is in state 0 if and only if the number of b read is a multiple of 3, in state 1 if and only if the number of bs read is a multiple of 3 plus 1, and in state 2 if and only if the number of bs read is a multiple of 3 plus 2 (see Exercise 5.1.3). More examples. Three more examples at least three of a DFA for the binary alphabet are illustrated at left, a DFA over the DNA alphabet. No No Yes b b the machine recognizes the complement of the language Yes that one... the top of the next page, and several other YES DFAs are described in the exercises. For each one, convince yourself that it recognizes the claimed language by first tracing fewer than three to check that it accepts a few NO strings that you know are in the language a 0 a 1 a 2 a 3 and rejects a few strings that you know are in not. Yes Yes b b b b bnot in the language, then trying to understand why it always gets the YES right from state 0. For example, to see NO a b a N o a b a b a N o that a DFA recognizes the number of states a good way is to use the mind-set of asking for each state "what is the same from the state from the starting state to this one?" Even with such a simple model, these are set up to describe how DFAs work. as the DFA after... DFAs are ubiquitous in applications. DFAs are a 0 a 1 2 widely used to address nontrivial computing tasks. For example, in Section 5.5 and, it is no less important, to as recent use of DFA modeling for... In fact you can see it in all components for... are a complete and rather finite-state machine. are used in... if any of the below for DFAs is to be... Many applications of DFA in our notes more. For example, DFAs are widely used in... DFAs... is a form of a text such as... systems for graphical user interfaceMore examples of DFAs faces, and in many other software applications. The DFA abstraction also allows to describe control units in physical systems such as vending machines, elevators, automatic traffic signals, and computer processors, not to mention all sorts of natural entities, from molecules to plants to the human genome. In all of these contexts, we have the idea of a system transitioning from one state to another according to rules that depend only on the state and the next input. Yes NO Yes b b b ACGT ACGT 0 N o ACGT N o 1 ACGT No ACGT N o 3 ACGT N o 2 A T ACT No 3 0 No 4 ACGT ACGT A C N o CT N o 0 N o CT ACGT No 4 Yes 5 6 A G ACGT ACGT A CT No ACGT No C A G T T G Theory of Computing 741 Java implementation of a DFA Suppose that we are now speaking of an abstract machine, we normally mean that it has a physical implementation. For example, we will see that we can specify a DFA with a table of numbers, or as a program, or as a circuit, or as a piece of mechanical hardware. Indeed, DFAs are appropriate for modeling all kinds of objects in the physical world. This decision gives us the freedom to work with different DFA representations in different circumstances. We can fully specify a DFA with a table of numbers, so can we write a program to simulate the operation of any DFA. That is, we not need to implement all the DFA ever. An implementation of a DFA is just its operation in machines, both real and abstract, at the same time. How do we build DFAs? There is no need to do so, because it is easy to develop a Java program to simulate the operation of a DFA by; that is, we can implement the abstract DFA, or build a DFA simulator. is a universal virtual DFA By virtual, we mean that it is implemented as a program on some machine, rather than as a physical device. Program 5.1.3 is a universal virtual DFA that takes a DFA specification (from a file named on the command line) and a sequence of strings from standard input and prints the result of running the DFA on the given input strings. The input format for a DFA is the number of states followed by the alphabet followed by one line for each state. Each such line contains a string that is either Yes or No (which specifies whether the state is an accepting state) followed by a state index for each symbol in the alphabet, the ith index giving the transition from that state when the DFA input is the ith alphabet symbol. The file b3.txt shown below the program defines our sample DFA that recognizes the language number of bs is a multiple of 3 and the file gene.txt defines the DFA drawn above that recognizes potential genes. 742 Theory of Computing Implementation The constructor creates the data structures needed to hold an internal representation of the DFA in the file given as command-line argument: • Read the number of states and the alphabet. • Create an array of strings for the accept/reject value of each state and array of symbol tables for the state transitions. • Populate these data structures by reading the accept/reject value and state transitions for each state. The code to implement the constructor is straightforward: public DFA(String filename) { In in = new In(filename); int n = in.readInt(); String alphabet = in.readString(); action = new String[n]; next = (ST[]) new ST[n]; for (int state = 0; state < n; state++) { action[state] = in.readString(); next[state] = new ST(); for (int i = 0; i < alphabet.length(); i++) next[state].put(alphabet.charAt(i), in.readInt()); } } The other methods in Program 5.1.3 are also straightforward. The simulate() method simulates the operation of the DFA, and the main() method in DFA invokes the simulated() method for each input string. Program 5.1.3 is both a complete specification of what constitutes a DFA and an indispensable tool for study their properties. You give it any DFA's tabular description, any input strings, and it will tell you which to be evaluated against each string. It is a practical solution to the recognition problem for DFAs, and a very simple example of the concept of a virtual machine, which we will visit in each chapter. In our study of the DFA, we have seen the abstract of "program" for "program" by specifying a set of vertices and transitions that follow the rules laid out for legal DFAs. Each DFA is a "program" for this computer. 5.1 Formal Languages 743 Program 5.1.3 Universal virtual DFA public class DFA { private String[] action; private ST[] next; public DFA(String filename) { /* See page 742. */ } public String simulate(String input) { int state = 0; for (int i = 0; i < input.length(); i++) state = next[state].get(input.charAt(i)); return action[state]; } public static void main(String[] args) { DFA dfa = new DFA(args[0]); while (StdIn.hasNextLine()) { StdOut.println(dfa.simulate(StdIn.readLine())); } } } This program simulates the operation of the DFA specified as the first command-line argument for each line of input (see text for details). % java DFA b3.txt babbaabb Yes % more b3.txt 3 ab Yes 0 1 No 0 2 No 2 0 % java DFA gene.txt ATGCAA No ATG No 10 ATG C N o 0 No 1 No 2 No 10 A 0 No 10 10 0 5 3 3 10 10 10 4 5 5 3 5 3 10 10 10 3 10 % java DFA gene.txt ATGCTCCTTTCTAG Yes ATGCCTTCCTTTCTAG Yes ATGTAA Yes ATGAAATAA Yes 744 Theory of Computing Nondeterminism Next, we consider an extension to our DFA model that does not correspond to the way that we normally think about computers in the real world. Why do we do so? The answer to this question is that this model not only leads to more compact abstract machines but also provides insights that help both us and computers to solve practical problems. In the present context, it represents a link between DFAs and REs that leads to a full understanding of regular languages and to a practical solution to the recognition problem for REs. Nondeterministic finite-state automata. The behavior of a DFA is deterministic: for each input symbol and each state, there is exactly one possible state transition. A nondeterministic finite-state automaton (NFA) is an abstract machine that allows for multiple possibilities. Specifically, NFAs are the same as DFAs, but with restrictions on the transitions leaving each state removed, so that • Multiple transitions labeled with the same symbol are allowed. • Unlabeled state transitions (null transitions) are allowed. • Not all symbols need to be included among the transitions leaving each state. The important difference is that NFAs can have more than one transition labeled with the same symbol leaving each state. Specifically, a NFA can change state without consuming any input, and on ... each state corresponding to any given input symbol. The mechanism for choosing whether or when to use a transition is unspecified. We regard a nondeterministic finite automaton an input as acceptable if there is any sequence of transitions that can take the machine from the start state (0) to an accept state. An example NFA is shown above. When in state 0 and reading an a, it can choose to stay in state 0 or make the transition to state 1. Next, we analyze this machine to determine which language it recognizes. Examples of NFAs. Without the deterministic state transitions that characterize DFAs, we have to be more careful when we work to solve the recognition problem for NFAs. For example, suppose that the input string for our example NFA is bbab. If you study it for a moment, you will see that this NFA can accept this input string via the state transitions 0-0-0-1-2. It could also make the transition 0-1-2 or 0-0-0-0, but those sequences do not both exhaust the input and lead to an accepting state; 5.1 Formal Languages 745 consequently, they are not of interest. At the same time, it is easy to see that if the input string is bbbb, there is no way for the NFA to get from state 0 to state 1, so it must light the NO indicator. To solve the recognition problem for NFAs, we have to account for all of these possibilities. The necessity to do so contrasts sharply with the corresponding task for DFAs, where we follow simple rules to go from state to state. In this case, we can prove that our example NFA recognizes the language of all strings whose second-to-last symbol is a. First, it can accept any string whose second-to-last symbol is a, by staying in state 0 (the start state) until the second-to-last symbol is read, at which point it reads the a and advances to state 1 and then state 2 (the accept state) after reading the last symbol. Second, there is no sequence of state transitions that could accept any string whose second-to-last symbol is not a, since the only way to get to state 1, with one input symbol remaining. An example of an NFA To understand an NFA, it is sometimes convenient to think of it as making a state change without reading an NO input symbol) pictured at right. This capability a 0 1 2 No No Yes enables NFAs to make it even more difficult for us to understand whether a particular NFA accepts a pa b YES tircular string, let alone which language a particular NFA recognizes. To appreciate this claim, take the time to convince yourself that the language recognized by the pictured NFA is the set of all binary An NFA with a null transition strings that do not contain bba as a substring. We refer to this language as no occurrence of bba. Recognition problem for NFAs. How can we solve the recognition problem for NFAs? For DFAs, the process was simple because there is only one possible state transition at each step. For NFAs, however, we are looking for the existence of at least one sequence of state transitions in the face of a large number of possibilities. When defining NFAs, we do not specify how the automaton does its computation; indeed, it is not easy to imagine how we would build such a machine. It would seem to be quite a bit of work for each machine not only to find a sequence of transitions from start state to accept state for inputs that are in its language, but also to prove to itself that there is no such sequence of transitions for inputs not in its language. Theory of Computing 746 One way to think about the operation of an NFA is as guessing a correct sequence of transitions. We then need a notion of what it means for that "machine" to always be right: the machine accepts a string precisely when there exists at least one sequence of state transitions ending in an accept state... either { this statement } or do { that statement }. If this makes it seem as though nondeterminism is taking us away from the real world... the NFA is a model that is going to be quite useful... Our intuition is that NFAs are more powerful than DFAs... Actually, nondeterminism is a concept that is both of practical utility and critically important in understanding the fundamental nature of computation. Fortunately, it is not difficult to simulate the operation of NFAs, by keeping track of all possible states the machine might be in as it reads input. We simulate an NFA by keeping track of all possible states that can be reached by following paths in the NFA starting from state 0. • Find all states that can be reached by following null transitions from state 0 and put them in a set. • Read the next symbol from the tape. • Iterate this process, keeping track of all possible states that the NFA could reach before reading the next symbol. • Iterate this process, keeping track of all possible states that the NFA could reach after reading the input symbol. After reading the input symbol, the set of states is the input. If this set of states contains any accept state, the input is in the language recognized by the NFA. These steps are deterministic: they constitute a way to understand the computation underlying any NFA, without any guessing. c c c all states reachable after reading i symbols possible null transitions on reading (i+1) symbol possible null transitions One step in simulating the operation of an NFA all states reachable after reading i+1 symbols 5.1 Formal Languages 747 NFA trace example. As an example, consider the operation for our no occurrence of bba NFA, when given the input abbba, depicted at right. To start, the set of possible states after reading the first b (a) is still {0, 2} since the only a-transition is from 0 back to 0 and then a null transition to 2 is possible. After reading the second bit of input (b), we could be in state 1 (taking the b-transition from 0) or in state 2 (taking the b-transition from state 2 back to itself). The third bit is a, and the only a-transition from {1, 2} takes us back to state 0 and a null transition is always possible from 0 to 2, so we wind up with {0, 2} as the only possible states after reading the third bit. As before, a-transitions from {0, 2} take us to {1, 2} after the fourth bit, and the b-transition from state 2 takes us to {2} after the fifth bit. Since it can exhaust the input and end in state 2 (which is an accepting state), we know that this NFA will accept abaab (and we can figure out how to get there with some additional ...the set of possible states after reading a sixth input symbol b would be empty (since the only possible state is state 2 after the fifth bit and it has no a-transitions), so abaabb would be rejected by this NFA. 747 NO 0 b a N o 1 Yes b a 2 b b N o 1 YES b a b b a b b a NO 0 b a No 1 No b a 2 b b N o 2 YES Yes b NO 0 b a 2 b a b b a b b a Yes 2 b a b b a b b a b b a b b a NO 0 b a N o 1 N o Yes 2 b a b b a b b a b b a Yes 2 YES b a b b a b b a Tracing the operation of an NFA Theory of Computing 748 Java implementation of an NFA. As we developed in Program 5.1.3 a computer program to solve the recognition problem by simulating the operation of the DFA, step by step. To do the same for NFAs, we use the same approach as for the method of tracing that we just considered: rather than keeping track of a single state as the program counter, we keep track of a set of states: the set of all states the NFA could reach after reading the characters so far. The idea of machines that can guess the right answer is a fantasy, but such a program allows us to study and use nondeterminism as if it were real. Again, we can think N o a No 1 2 b a b b a N o 1 No Yes a No a b b a b Yes Programming the NFA as a "program" for it as our computer. It is reasonable to ask whether this computer is more powerful than our "DFA computer" and Kleene's theorem There is a striking connection among regular expressions, DFAs, and NFAs that has dramatic practical and theoretical consequences. This connection is known as Kleene's theorem, named after the logician Stephen Kleene, who established the connection between automata and regular languages in 1951. Theorem. (Kleene, 1951) REs, DFAs, and NFAs are equivalent models, in the sense that they all characterize the regular languages. Proof sketch: See discussion below. Kleene's theorem is the first major theoretical result that we consider, and we will use it to illustrate some basic properties of a mathematical proof. Our purpose in presenting a theorem statement is to make you aware of an important fact, and our purpose in presenting a proof sketch is to convince you of the truth of the theorem. We do not state and prove theorems in the standard mathematical style in this introductory treatment. We give you the intuition behind each proof, along with enough details that you can understand the proof and that people who are comfortable with proofs can fill in the rest. Our first proofs are constructive, and perhaps easier to check than some other kinds of proofs (which, for example, might be dependent on principles of logic) because we include examples to illustrate the constructions. While examples do not constitute proofs and more than test cases demonstrate that a program is valid, they are a necessary first step. If you read a proof, then study an example, then reread the proof, you will find it easier to convince yourself of the truth of the theorem. To prove Kleene's theorem, we need to show that we can build a machine that corresponds to a regular language and conversely. Specifically, we need to prove that anything that DFA can do, NFAs can also do; then we need to show that we can translate a regular language, to prove Kleene's theorem, then, so we need to show that we need to show that for NFAs and REs: we need to show that DFAs and NFAs are equivalent, and then we prove that we can translate REs to NFAs and NFAs to REs. 5.1 Formal Languages 749 NFAs and DFAs are equivalent. Every DFA is also an NFA, so we can establish equivalence by showing how to build a DFA that recognizes the same language for any given NFA. To start, this DFA has one state corresponding to each possible set of NFA states. Since each state of the DFA represents a set of states of the NFA, there are exactly 2^n possible states in the DFA (the NFA has n states), though we can ignore the set of possible states after reading any given portion of the input, so this NFA can then NFA after b by having read any given portion of its input, so this NFA can handle all the possibilities that can arise. NFA To figure out the transitions, we use the same approach as described when we discussed the operation of NFAs. Given a set of NFA states and an in- NFA-to-DFA conversion table name accept? a b put symbol, we form a set of NFA states by taking all the states that can be reached 0 0 1 by following a transition labeled with the symbol from any of the states in the set. For example, the DFA for the no occurrence of bba NFA 01 No 02 Yes 0 An example of the NFA states, and we transition between them... we have only those NFA states in the set... 0 2 1 012 ... the NFA's accept states. ... For example, consider the NFA in... For each state of the NFA conversion... Given the state 012 in the first column, then with input a... the DFA can be shown... Proofs of NFA-DFA equivalence... For our example... The empty set corresponds to... we use... all states that have no path from state 0... we can therefore remove them... (this is a subset of the states) ... we can also remove states that are unreachable... here is a... 2^n... 0, 1, 2 (single-state subsets), 01, 02, 12 (two-state subsets), 012... For each of these states, we identify the successors for each input symbol... For example, if the input symbol is a when in state 01, the only successor is 0 since we can move to 0 or 1-2 in the NFA, so in the ... the DFA is the set {0, 2}... The conversion table... DFA we can then remove states not reachable... state... 0, 02, 012, 01, 12... 750 Theory of Computing... NFAs are equivalent, we need to show that for any NFA there exists an equivalent DFA and vice versa. For any NFA, the corresponding DFA has exactly the states we have described... to DFA that recognizes the same language. For the exercises... Note: In NFAs with null transitions, the first thing that you need to do is to compute... This completes the proof of equivalence of NFAs and DFAs. Kleene's theorem... NFAs and REs are equivalent. These two are different... We establish the equivalence of NFAs and REs with a pair of constructions... Our first step... 5.1 Formal Languages 751 Concatenation • Combine edges that enter it from the same source, using |. • Combine edges... Union • Closure... To build an NFA from any given RE... • Remove a state... • Remove the transitions... The proof is complete... We can convert any NFA into an RE... That said, the method needs to... The result is the proof of the equivalence of NFAs and REs... Kleene's theorem... If you want to work with a practical use of these models... Applications of Kleene's theorem. Kleene's theorem is both a significant theoretical result and directly applicable in practice. Next, we will consider two important applications of the theorem, one example each from theory and practice. RE recognition. The fundamental practical application of the theory is to solve the recognition problem for REs: given an RE and a string, is the string in the language specified by the RE? Our proof of Kleene's theorem is a constructive proof, so it provides a direct path to building a program that solves the recognition problem for REs: • Build an NFA that corresponds to the given RE. • Simulate the operation of the NFA on the given input string. That is precisely the approach taken by Java to implement its matches() method that we studied earlier in this section. The implementation requires special care to be sure that the cost remains under control. You can find a complete description and implementation in our book Algorithms, Fourth Edition. It is an example of a paradigm that is fundamental to computer science: • Pick an intermediate abstraction (in this case, NFAs). • Build a simulator that makes concrete that abstraction. • Build a compiler that transforms problems to the abstraction. This very same basic process is used to compile programs that you write in Java to bytecodes that are executed on the Java virtual machine (which itself is an abstraction eventually simulated with programs for particular computers that are written in a lower-level language). Limits on the power of DFAs. Kleene's theorem is also instrumental in helping us to shed light on a fundamental theoretical question: which formal languages can be described with an RE and which cannot? To begin to address this question, consider the language of all binary strings that have an equal number of as and bs. This language would seem to be at least as simple as many of the languages that we have been considering in this section. Can we write an RE that describes this language? Or can we build a DFA that recognizes it? You might try as much as you want, but it is impossible to describe with an RE. How can we prove such a result? Kleene's theorem is an important tool for establishing limitations. The equivalence of REs, DFAs, and NFAs that is established by Kleene's theorem means that we can use any one of them to characterize the concept of regular language, and DFAs are the most convenient for proving limitations of this type. Proposition. The language of all binary strings with an equal number of as and bs is not regular. Proof. For the sake of contradiction, let's assume that there does exist a DFA that can recognize this language. Let n denote the number of states in the DFA. This DFA will accept any string with an equal number of as and bs. Now, consider the trace of the DFA on a string of n+1 as followed by n+1 bs. In particular, it recognizes a string consisting of a sequence of n symbols followed by a sequence of n bs in the input... Since the DFA has only n states, the pigeonhole principle... at least one state must be visited more than once... the DFA visits while accepting this string. Since the DFA has only n states, the pigeonhole principle tells us that the trace must revisit one of the states while reading in the as (since there are n+1 states in that part of the trace, including the start state). Now we can... which is not in the language. Thus, our assumption that the DFA exists must be false... This proof illustrates that the DFA must end up in the same final state with the second input as it does with the original input. Therefore it accepts the second input. But that input has fewer as than bs, contradicting our assumption that the DFA recognizes the language. This contradiction implies that our original assumption that there exists a DFA that recognizes the language must be false. 5.1 Formal Languages 755 With such a proof, we make a step-by-step sequence of logical inferences to arrive at a conclusion that is certainly false. The contradiction means that one of our assumptions is false. Since the pigeonhole principle and the DFA behavior are true, the assumption that the DFA exists must be false. ... Kleene's theorem... no RE to describe... The same method can be used to show that many other languages are not regular... This same kind of argument... For example, palindromes... We cannot build a DFA... Because DFAs have only finite memory... In practice... The extended REs of Java (with back references)... What is this... so we can handle...? To address this, we need a more powerful abstract machine, as we will see in the next section. Machines with more power. Even more important, knowing a language is not regular opens up a critical question for us to consider: what is the simplest way to extend our abstract machine model to get machines that can recognize more languages? ... Kleene's theorem, even the power of nondeterminism... we can get more powerful machines... DFAs... a pushdown automaton (PDA)... context-free languages... Summary... REs, DFAs, and NFAs are equivalent models for describing a set of languages known as the regular languages. This equivalence makes it possible for us to prove facts about regular languages and to develop programs that take advantage of their properties to accomplish a broad variety of basic computational tasks. There exist many languages of interest that are not regular—which formal mechanisms are available for defining them? Are there languages that cannot be described with these more general mechanisms? These sorts of questions and the relationships among languages, formal methods of describing them, and abstract machines that are demonstrated by Kleene's theorem give better context for the questions that we mentioned at the beginning of this chapter: • Which kinds of problems can we solve with a computer? • What are the limits to what computers can do? In this section, we have thoroughly addressed these questions for finite automata, which are simple abstract computers that we can precisely define. In addressing these questions, we considered numerous practical applications of these simple abstract models. Moreover, we added two more fundamental questions to the list: • What is the relationship between computers and languages? • Does nondeterminism make machines more powerful? Next, we will consider these questions for more powerful abstract machines, but we will be careful to consider the relationship between languages and machines that we have considered so far. 5.1 Formal Languages Q&A Q. What is the definition of a symbol? A. Symbols are the basic abstract building blocks of formal languages. From a mathematician's point of view, the definition of symbols makes no difference: the study of formal languages is the study of sets of sequences of symbols. From a scientist's or an engineer's point of view, symbols are what connect the abstract world of formal languages and the real world where we use them. A symbol might be something real like a flip-flop, a gene, a neuron, a molecule, or something abstract like a bit, a digit, a lowercase letter, or a Unicode character. Q. Intersection is also a fundamental operation on sets. Why not include an intersection operator as part of the basic definition of REs? A. Good question. Since languages are just sets, many things carry over from set theory to REs. In fact, it is possible to prove that the regular languages are closed under intersection (see Exercise 5.1.37). Q. How do we escape characters for specifying metasymbols as a set in a Java RE? A. Java's REs have many more... Q. Are there any shortcuts? A. Quite the contrary. When you specify an RE inside a Java string and run it through Java's mechanism, you might need several levels of escaping to escape string literal semantics. For example, it might be \\\\ to represent a single slash character, and you need to type "\\s" to represent a whitespace character and "\\\\" to represent a single slash character. Also, sometimes you need to escape special characters on the command line. For example, the command-line argument on page 732 that represents a regular expression for U.S. currency uses "\\$" to specify a dollar sign and "\\." to specify a decimal point. Q. What's the point of studying NFAs? They are much harder to think about than DFAs and I'm not clear on the advantage. A. Well, we used them to prove Kleene's theorem, and it is possible to use them to implement RE recognition. Another reason is that they tend to have many fewer states than corresponding DFAs. Try constructing a DFA for the set of all strings whose tenth-to-last symbol is a 1. You can't do it in fewer than 512 states, even though the NFA for this language has just 10 states. 757 Theory of Computing 758 Q. What is the closure of the empty set? A. The empty string: ()* = ε. 5.1 Formal Languages 759 Exercises 5.1.1 Write a Java program that solves the recognition problem for the language "is a palindrome" that is a palindrome ... 5.1.2 Give an RE that specifies each of the following languages over the alphabet {a, b}: ... 5.1.3 Prove that the DFA given on page 738 recognizes the language of strings with a number of bs that is a multiple of 3... 5.1.4 Give a brief English description of the languages specified by each of the following REs over the binary alphabet: a. a|b b. ab c. (ab)* d. .*aa.* ... 5.1.5 Write a Java RE for... 5.1.6 Write a Java RE to match all words that contain the five vowels (a, e, i, o, u) in order, and no other vowels (e.g., abstemious and facetious). 5.1.7 Write a Java RE for dates of the form July 4, 1776 that is the extent possible, includes all days... 5.1.8 Write a Java RE for... 5.1.9 Write a Java RE for (simplified) floating-point literals in Java. A floatingpoint literal is a decimal number (see the previous exercise), followed optionally by the following: e or E, followed by an optional + or -, followed by an integer mantissa. 5.1.10 Let L be the language {aaaba, abba, abbba, aaaaa}. Identify whether each of the following REs (i) contains no strings in L, (ii) contains only some strings in L and some other strings, (iii) contains all strings in L and some other strings, or (iv) is the same as L. a. a(a|b)*abb(a|b)* b. a(a|b)*a c. a*b*aba d. a((a*|b*)|(b*aba*))a e. a*b*aa*b*ba*a*a*b*b*a*a*b* f. (a|b)(a|b)(a|b)(a|b)a g. (a|aa|aaa)(ba|aa|bbb)a 5.1.11 Draw a DFA that recognizes the language of all strings with an odd number of as and an even number of bs, then make a file in the format expected by DFA (Program 5.1.3) and test your design. 5.1.12 Draw a DFA that recognizes Social Security numbers, then make a file in the format expected by DFA (Program 5.1.3) and test your design. 5.1.13 Write a Java program that solves the recognition problem for the language specified by the regular expression C.{2,4}C...[LIVMFYWCX].{8}H.{3,5}H for the sequence of strings on standard input, without using Java's RE mechanism. 5.1.14 Draw a DFA that recognizes the language a.*a|b.*b and then draw an NFA with fewer states that recognizes the same language. 5.1.15 Draw a DFA that recognizes the language .*aabab.* (the set of all strings that contain the substring aabab) and then draw an NFA with fewer states that recognizes the same language. 5.1 Formal Languages 5.1.16 Consider a vending machine that accepts nickels, dimes, and quarters, for items that cost 25 cents. Draw a DFA with a state corresponding to each possible amount of money inserted, and add transitions such that the machine is in state i when the amount of money so far inserted is 5 times i. 5.1.17 Describe how to convert any DFA into a DFA that recognizes the complement of the language recognized by the original (the set of strings over the same alphabet that are not in the language). 5.1.18 Answer the previous question for NFAs. 5.1.19 Create a DFA that recognizes the language of all binary strings having an odd number of as and an even number of bs. 5.1.20 Convert the DFA from the previous exercise into a regular expression. 5.1.21 Draw NFAs that correspond to the following REs: a. a(a|b)*a b. a*b*aba c. (a|b)(a|b)a d. a((a*|b*)|(b*aba*))*a e. ab(a|b)ba | a(a|b)aba | aa(ab|ba|aa)a 5.1.22 Convert the NFA given on page 745 (which recognizes the strings that do not contain the substring bba) into a DFA. 5.1.23 Is it possible to construct a regular expression that describes all binary strings that have an equal number of occurrences of ab and ba as substrings? Answer: Yes: (a.*a) | (b.*b)| a*|b*. It is true that DFAs can't "count," but in this case the language is equivalent to all binary strings that start and end with the same bit, so no counting is needed. 5.1.24 Show that no DFA can recognize the set of all palindromes. 5.1.25 Show that the language containing all binary strings that do not contain the substring bba but have a multiple of three bs is regular. 761 Theory of Computing 762 Creative Exercises 5.1.26 Challenging REs. Construct an RE that specifies each of the following languages of binary strings over the alphabet {0, 1}. a. All binary strings except 11 or 111 b. Binary strings with 1 in every odd-number bit position c. Binary strings with at least two 0s and at most one 1 d. Binary strings with no two consecutive 1s 5.1.27 Binary divisibility. For each case listed, construct an RE that specifies all binary strings that when interpreted as a binary number are: a. Divisible by 2 b. Divisible by 3 c. Divisible by 6 5.1.28 RE search. Check various applications on your computer (web browser, word processor, music library, or whatever else you use regularly) to determine the extent to which they support REs in their search function. 5.1.29 Divisible by 3 (binary). Design an DFA that recognizes the language consisting of all binary strings that represent binary numbers that are divisible by 3. For example, 1111 is accepted since 1111 is 15 in decimal, but 1110 is rejected. Hint: Use three states. Depending on the input read in so far, your DFA should be in one of three states depending on the remainder when dividing 3 into that number. 5.1.30 Bounce filter. A finite state transducer is a DFA whose output is a string of symbols. It is the same as a DFA except that each transition is labeled with an output symbol, which is to be output whenever the transition is made. Develop a transducer that removes isolated symbols from its input. Specifically, any occurrence of aaba in the input should be replaced by aaaa and any occurrence of bbab should be replaced by bbbb in the output, but the input and output should otherwise be unchanged. 5.1 Formal Languages 5.1.31 Harvester. A Java Pattern object is a representation of a regular expression. Such an object can be used to build a Matcher object for a given string. You can think of a Matcher object as a representation of an NFA for the RE. Operations on Matcher objects include find(), which finds the next match of the RE in the string, and group(), which returns the string characters that led to the match. Write a Pattern and Matcher client that takes a file name (or URL) and an RE as command-line inputs and prints all of the substrings in a file that match the RE. Answer: import java.util.regex.Pattern; import java.util.regex.Matcher; public class Harvester { public static void main(String[] args) { In in = new In(args[1]); String re = args[0]; String input = in.readAll(); Pattern pattern = Pattern.compile(re); Matcher matcher = pattern.matcher(input); while (matcher.find()) System.out.println(matcher.group()); } } 5.1.32 Count matches. Develop a program that counts the number of substrings in standard input that match the RE on the command line, without worrying about overlapping matches (use Pattern and Matcher as in the previous exercise). 5.1.33 Web crawling. Modify the solution of Exercise 5.1.31 to develop a program that prints all web pages that can be accessed from the web page given as a command-line argument. 5.1.34 One-level REs. Construct a Java RE that specifies the set of strings that are legal REs over the binary alphabet, but with no occurrence of nested parentheses. For example, (a.*b)*|(b.*a)* is in this language, but (b(a|b)b)* is not. 763 Theory of Computing 764 5.1.35 Search and replace. Write a filter SearchAndReplace.java that takes an RE and a string str as command-line arguments, reads a string from standard input, replaces all substrings on standard input that match the RE with str, and sends the results to standard output. First solve the problem using the replaceAll() method in Java's String library; then solve it without using that method. 5.1.36 Empty language test. Add a method isEmpty() to DFA (Program 5.1.3) that returns true if the language recognized by the machine is empty, and false otherwise. Answer: This code uses "breadth-first search" with a Queue to keep track of the states reachable from state 0 (see Program 4.5.4). public boolean isEmpty() { Queue queue = new Queue(); boolean[] reachable = new boolean[n]; queue.enqueue(0); while (!queue.isEmpty()) { int state = queue.dequeue(); if (action[state].equals("Yes")) return false; reachable[state] = true; for (int i = 0; i < alphabet.length(); i++) { int st = next[state].get(alphabet.charAt(i)); if (!reachable[st]) queue.enqueue(st); } } return true; } 5.1.37 Set operations on languages. Given NFAs that recognize two languages A and B, show how to construct NFAs that recognize the union and the intersection of A and B. 5.1 Formal Languages 5.1.38 Random inputs. Write a program to estimate the probability that a random binary string of length n will be recognized by a given DFA. 5.1.39 NFA to DFA conversion. Write a program that reads in the description of an NFA and creates an DFA that recognizes the same language. 5.1.40 Minimal languages. Given a regular language L, show that the set of minimal strings in L is also regular. By minimal, we mean that if x is in L, then xy is not in L for any nonempty string y. Answer: Modify DFA so that once it leaves an accept state, it always rejects. 5.1.41 Back references. Show that the back-reference operation is a not-so-regular expression that cannot be constructed from the core regular expression operations. Show that no DFA can recognize the language consisting of ww where w is some string (e.g., beriberi and couscous) but the Java regular expression (.*)\1 describes this language. 5.1.42 Universal virtual NFA. Develop a program analogous to Program 5.1.3 that can simulate the operation of any NFA. Implement the method described in the text, using a graph representation for the NFA and a Queue to keep track of the set of possible states, as in Exercise 5.1.36. Instrument your program to print a trace of the set of possible states for the NFA before reading each symbol. Test your code by running it for the NFA described in the text that recognizes the set of all strings whose fourth-to-last symbol is a 1 on the input aaaaababaabbbaababbbb. 5.1.43 Universal virtual PDA. A pushdown automaton adds a stack to a DFA and the ability to, on any transition, push and pop symbols from the stack. Specifically, each transition has an additional label x/y that says to pop the stack and push y if the symbol returned was x, with /y meaning just push y without popping anything, x/ meaning pop without pushing if the symbol returned was x, and / meaning just pop the stack. Develop a program analogous to Program 5.1.3 that can simulate the operation of any PDA. 5.1.44 PDA for a nonregular language. Draw a PDA (see previous exercise) that can recognize the language equal numbers of as and bs. Then use your solution to the previous exercise to test it. 765 Theory of Computing 5.2 Turing Machines In his 1937 paper, Alan Turing introduced one of the most beautiful and intriguing intellectual discoveries of the 20th century. It is a simple model of computation that is general enough to embody any com5.2.1 Virtual TM tape . . . . 776 puter program, and it forms the founda5.2.2 Universal virtual TM . . . . 777 tion of the theory of computing. Because Programs in this section of its simple description and behavior, it is amenable to mathematical analysis. This analysis led Turing to a deeper understanding of computers and computation, including the revelations that all known computational artifacts are equivalent in a profound technical sense and that some computational problems cannot be solved on computers at all, no matter how fast the processor or how much memory is available. We will address these ideas in the next two sections, but we need to lay the groundwork by defining and considering basic properties of Turing's invention. If you have read about Turing machines in other contexts, or encounter them in the future, you may find the definitions and some properties to slightly differ from our treatment. Such differences are inconsequential; indeed, one of the points of the theory is the idea that any model of computation that we might invent is equivalent to any sufficiently rich model. The approach we use is adapted from a model developed by Marvin Minsky at MIT in the mid-20th century. Turing machine model The object of our study is an abstract machine model that is only slightly more complicated than the DFA model of the previous section, which is known as the Turing machine (TM). In the description that follows, the differences from DFAs are highlighted in blue. Each TM consists of • A finite number of states, each of which is designated as either a left state, a right state, a halt state, an accept state, or a reject state. • A set of transitions that specify which state is next and which symbol to write. Each state has one transition for each symbol in the alphabet. • A tape that holds a string of symbols. • A tape head capable of reading or writing a symbol and moving left or right to read the next symbol. A TM operates by starting in state 0 with the tape head positioned on the first (leftmost) input symbol, then reading and writing the tape and changing state as directed according to the following rules: 5.2 Turing Machines • Read a symbol. • Note the transition associated with that symbol and the current state. • Write the symbol specified by the transition. • Change state as specified by the transition. • Move the tape head right one position if the new state is a right state. • Move the tape head left one position if the new state is a left state. • If the new state is a halt, accept, or reject state, halt operation. [illegible]

[illegible] More formally, we use the concept of a universal Turing machine to express the idea: 5.3 Universality [illegible]

earlier), Alonso Church invented a completely different mathematical model to study the notion of computability. As it turns out, Church's model, known as lambda calculus, led directly to the development of modern functional programming languages. With respect to computation, the models appear to be very different, but Turing later showed that they were equivalent in that each characterizes precisely the same set of mathematical functions. While their studies were purely theoretical, this equivalence led Church and Turing to reach the same conclusion about computation. Turing's attention was focused on an idealized mathematician computing functions on integers, and Church's approach was purely functional, but they and others eventually realized their ideas say something about the natural world: that all physically realizable computing devices can be simulated by a Turing machine. This reduces the study of computation in the real world to the study of Turing machines, rather than an infinite number of potential computing devices. More formally, we use the concept of a universal Turing machine to express the idea: 5.3 Universality Church–Turing Thesis A universal Turing machine (UTM) can perform any computation (decide a language or compute a function) that can be described by any physically realizable computing device. Comment: In the form contemplated by Turing and Church, the thesis is an idea about what we can imagine about computation performed by an "idealized mathematician" following a well-defined procedure. In the strong form stated here, the thesis is a statement about the laws of nature and which kinds of computations can be done in our universe. The Church–Turing thesis is not a mathematical statement, and it is not subject to rigorous proof. The reason for this is that we cannot mathematically define what we mean by a "physically realizable computing device." As we will see next, an enormous amount of evidence has accumulated in favor of the thesis in the many years since it was first formulated. However, it is possible that this thesis could be refuted (shown to be false). If someone discovers a more powerful model of computation that is physically realizable, we would have to abandon or amend it. The contrapositive of the thesis is also interesting. It says that if some computation cannot be done on a Turing machine, then it cannot be done using any physically realizable computing device. As we will see in Section 5.4, there are languages that Turing machines cannot decide. Thus, if we believe the Church–Turing thesis, we must conclude that there are limitations on which computations we can perform with physically realizable computing devices. Variations on the TM model One body of evidence in favor of the Church– Turing thesis has to do with studying whether changes in the Turing machine model make a difference. In one direction, researchers have studied whether adding mechanisms to the model lead to machines that are able to recognize some language that cannot be recognized by a TM or compute some function that cannot be computed by a TM; in the other direction, researchers have studied the extent to which simplifying the model without weakening it is possible. Keep in mind that, in the present context, we are not interested in how many states or symbols a machine might use or in how many state transitions might be required for a computation—rather, we are interested only in computational power as defined by the set of languages that can be recognized or the set of functions that can be computed. 791 792 Theory of Computing Equivalent models. Authors typically use minor variations of Turing's original model (like the model that we have considered, which was developed by Minsky) without comment when it is easy to see how to convert among models. Several choices are purely a matter of convenience. We believe our version leads to simple uncluttered diagrams that are easy for you to understand. Authors of an upperlevel textbook on the theory of computing might choose a different version that leads to simpler proofs. For example, a common choice is to associate tape head moves with each transition rather than with the target state. Another equivalent model, which we hinted at in Section 5.2, is a 2-stack pushdown automaton—our use of two stacks in TuringMachine provides the intuition behind this. Enhancements. A long list of possible enhancements have been studied for the TM model. For example, would adding another tape with an independent tape head make the TM model more powerful? The answer to this question is no because we could simulate such a machine with a TM, using odd positions for one tape and even positions for the universal DFA DFA TM other. As another example, consider adding the power of state table nondeterminism. This is does not add power because we can build a deterministic TM that recognizes the same lanTM UTM state table guage or computes the same function as any nondeterministic TM by a construction similar to the one that we used Two additional programs for NFAs. Other examples are shown in the top part of the process programs table on the facing page. We omit further discussion and proofs except to note that, despite decades of effort, no one has been successful in finding a more powerful physically realizable model. Restrictions. Researchers have also sought to simplify the Turing machine model. Several examples are shown on the bottom part of the table on the facing page. For example, using a tape that is unbounded just in one direction is no restriction because we can use odd positions for one direction and even positions for the other. As another example, using a binary alphabet is no restriction because we can encode the tape symbols in binary. Again, we omit details and further discussion and proofs. Of course, there is a limit to the restrictions: as we have seen, the DFA model is strictly less powerful than the TM model, as are a TM that cannot write to the tape and a TM that has a tape head that can move only in one direction. Finding the simplest TM model is a goal that has intrigued many researchers—we will briefly return to the topic soon. 5.3 Universality TM variant 793 description equivalent models moves on transitions 2-stack PDA L and R moves associated with each transition instead of target state Pushdown automaton with two stacks enhancements multitape Add a finite number of independent tapes with independent tape heads multidimensional Use a multidimensional tape (allow tape head to move in any direction) nondeterministic Allow multiple transitions for any input character probabilistic Transitions are chosen at random (accept if most outcomes lead to an accept state) oblivious Transitions do not depend on input restrictions one-way infinite Tape is unbounded in only one direction binary Only two symbols allowed 2-state Only two states allowed non-erasing No overwrite capability sequential States are circularly ordered (transitions allowed only from one state to the next) Variations of the TM model that do not affect its computational power 794 Theory of Computing There are hundreds, if not thousands of papers and books in the literature on variations of Turing's model that recognize the same set of languages and compute the same set of functions. This fact certainly indicates that the model is at the very least a turning point in our understanding of computation—the step from 1-stack PDAs to Turing-equivalent 2-stack PDAs is a giant step indeed. Universal models A model is said to be Turing complete or Turing universal if it is equivalent to the Turing machine model (it can recognize the same set of languages and compute the same set of functions). The Church–Turing thesis suggests that the Turing machine is a fundamental object in the natural universe. Are there other natural models of computation that can run any program on any input like the Turing machine? Again, the answer is a resounding YES! For a century, mathematicians, computer scientists, physicists, linguists, and biologists have considered scores of other models of computation that have been shown to be Turing complete. Many of these are listed in the tables on page 796 and page 797. We briefly highlight just a few of them here. Lambda calculus. As already noted, at the same time Turing was preparing his paper at Princeton University, Church was completing his work on lambda calculus, a formal system that is the basis for modern functional programming. The realization that Church's lambda calculus and Turing's machine model were equivalent led to the Church–Turing thesis. Counter machines. An even simpler model than the Turing machine, popularized by Minsky, is a counter machine, where the tape is replaced by a small set of counters that can hold any integer and the states and transitions are replaced by a sequence of instructions taken from a small set of operations such as "increment," "decrement," and "jump if zero." Cellular automata. You perhaps are familiar with the Game of Life, a computational model devised by the mathematician John Conway (see Exercise 2.4.20 and Exercise 5.3.19 for details). This game is an example of a cellular automaton, a discrete system in which cells interact with their neighbors. The study of cellular automata was initiated in the 1940s by John von Neumann, an important figure in the history of computing, as you will see in Chapter 6. 5.3 Universality Your computer. As we have already noted, TuringMachine (Program 5.2.2) proves that your computer is at least as powerful as any TM because it can simulate the behavior of any TM. But you may be surprised to learn that there exists a TM that can simulate the operation of your computer. As you will see in Chapters 6 and 7, your computer is based on a machine model involving instructions that manipulate binary numbers that is much more similar to a TM than to the Java environment with which you are familiar. Developing TMs that can simulate such a machine is conceptually not difficult. Therefore, if you can develop a program on your computer that can decide some language or compute some function, then there exists a TM that can do the same task (and any UTM can do it, too). Programming languages. Almost every programming language in use today is Turing complete. This includes procedural programming languages (such as C, Fortran, and Basic), object-oriented programming languages (such as Java and Smalltalk), functional programming languages (such as Lisp and Haskell), multiparadigm languages (such as C++ and Python), special-purpose languages (such as Matlab and R), and logic programming languages (such as Prolog). While some programming languages appear to be more powerful than others, they are all equivalent at their core (in terms of which functions they can compute and which languages they can decide). Programming languages differ in other important characteristics (such as maintainability, portability, availability, reliability, and efficiency), so our choice of language is based on convenience and efficiency, not capability. String replacement systems. Many models involve creating a set of rules for replacing substrings in sets of strings. Such systems can be quite simple, which accounts for their popularity. We examine some examples of such systems in the creative exercises at the end of this section. DNA computers. Modern molecular biology has provided an understanding of discrete changes in DNA that drive biological processes. In 1994, Leonard Adelman imagined harnessing these changes so as to effect computation and to simulate a Turing machine. Laboratory experiments confirmed the validity of this approach: it is possible to build a computer out of DNA! Of course, whether some natural biological process operates in the same manner is another question entirely. 795 Theory of Computing 796 model description early 20th century semi-Thue systems (Thue, 1910) String replacement rules, applied in arbitrary order. Post formal systems (Post, 1920s) String replacement rules designed to prove mathematical statements from a set of axioms. mid-20th century lambda calculus (Church, 1936) A method to define and manipulate functions (basis of functional programming languages like Lisp and ML). Turing machine (Turing, 1936) Finite automaton that reads and writes on an infinite tape. Post machine (Post, 1936) TM with one queue. Recursive functions (Gödel and others, 1930s) Functions defined for computation on the natural numbers. Unrestricted grammars (Chomsky, 1950) String replacement rules designed to describe natural languages. 2D cellular automata (von Neumann, 1952) 2D array of binary values that change according to specified rules, in a manner depending on the values of their neighbors. Markov systems (Markov, 1960) String replacement rules, applied in prespecified order. Horn clause logic (Horn, 1961) Logic-based system for theorem proving (basis for Prolog programming language). 2-register DFA (Minsky, 1961) DFA plus two counters (each counter stores an integer that the machine can increment, decrement, and test if zero). 2-stack TM (Shepherson/Sturgis, 1963) TM with two pushdown stacks. Game of Life (Conway, 1960s) A specific 2D cellular automaton. pointer machine Finitely many registers plus memory accessed as a linked list. random access machine Finitely many registers plus memory accessed via an index. Universal models of computation 5.3 Universality 797 model description late 20th century programming languages Java, C, C++, Python, Matlab, R, Fortran, Lisp, Haskell, Basic, Prolog. Lindenmayer systems (Lindenmayer, 1976) String replacement rules, applied in parallel (used to model growth of plants). billiard ball computer (Fredkin and Toffoli, 1982) Indistinguishable billiard balls move in the plane, making elastic collisions with each other and internal barriers. Particle computer Particles carry information through space (computation occurs when particles collide). 1D cellular automata (Cook, 1983) Vector of binary values that change according to specified rules, in a manner depending on the values of their neighbors. quantum computer (Deutsch, 1985) Computing via superposition of quantum states (based on work by Feynman in the 1950s). generalized shift maps (Moore, 1990) A single classical particle moving in a three-dimensional potential well made of parabolic mirrors. DNA computer (Adelman, 1994) Computing via biological operations on DNA strands. Universal models of computation (continued) Knowledge of the concept of universality leads to a dramatic shift of perspective. It seems to say that anything that we observe in the natural world that is similar to a computer actually is a computer. Despite decades of attempts, every reasonable model of computation that has been developed since Turing that is at least as powerful as a Turing machine (can simulate a Turing machine) has also been proven to be no more powerful than a Turing machine (because it can be simulated by a Turing machine). This knowledge is important because we can use the Turing machine, simple as it is, as a universal model for proving facts about computation. Facts about Turing machines that can be proved with complete mathematical rigor also apply to the computers that we regularly use. We begin with one of the most important such facts in the next section. The question of whether knowledge about Turing machines also applies to the natural world itself is a philosophical one, but one worth pondering. Theory of Computing 798 Q&A Q. Is it really possible to build a Turing machine that simulates a conventional computer like the microprocessor in my laptop? A. Yes. This is precisely what the Church–Turing thesis says. The classic book by Minsky includes a blueprint for such a Turing machine. Q. Isn't the Turing machine model strictly more powerful than real computers, because Turing machines have infinite tapes while real computers have finite memories? A. In a technical sense, yes. You can simulate a real computer with a massive DFA. However, that DFA would need more 21,000,000,000 states to model a computer with 1GB of memory! While a real computer can access only a finite amount of memory, in practice that amount is nearly limitless if you include the Internet. Similarly, if you believe that there is some finite limit on the number of bits accessible in the universe, then you have to concede that Turing machines are purely imaginary. Q. Can Turing machines model every type of computation? A. Turing machines are designed for deciding languages and computing functions. Some other types of computations do not perfectly fit this model—for example, generating random numbers, controlling a self-driving car, or making a soufflé. To model these kinds of computations, you would need to connect the Turing machine to suitable peripheral devices. Q. Why not just imagine a machine more powerful than a Turing machine? A. People have done so. Super-universal computing devices are abstract models that can, in principle, compute things that Turing machines cannot. One way to achieve such models is to store continuous values instead of discrete symbols. However, it is unknown whether continuous values truly exist in nature, and if so, whether some natural process can harness their power. 5.3 Universality 799 Creative Exercises 5.3.1 Universal virtual DFA (representation). Develop a DFA representation suitable for use on a TM tape. Solution. Start with the DFA input, followed by the four rows of the state table, labeled with the marker symbols A (for state 0), B (for state 1), and C (for state 2), each followed by two digits (each 0, 1, or 2) giving the next state for the two possible DFA input symbols. After the state table is a marker symbol Z, followed by three symbols that specify the action Y or N for each state. a a 0 Yes a 1 b b No state action 2 No b DFA input # b a b 0 Y 0 1 1 N 1 2 2 N 2 0 transitions b A 0 1 state 0 marker B 1 state 1 marker 2 actions C 2 state 2 marker 0 Z Solution. 1 0 from 5 R b:X a:X Z from 11 R b:X a:X to 19 from 17 R 3 R A 5 0 R R L X:# to 0 X:# to 6 X:# to 12 1 7 Z 12 4 2 2 to 20 6 A R B R 10 8 9 B R R a:X 11 1 L 2 R 0 13 b:X 0 C R 16 14 15 C R R 1 17 2 L R Z 21 No 18 19 20 R R R N Y 22 Yes Universal virtual 3-state DFA TM Y actions marker 5.3.2 Universal virtual DFA. Develop a TM that can simulate the operation of any given 3-state DFA on any given input. next state a b Y N # Theory of Computing 800 To see how this TM operates, consider the first six states, shown at right. These states correspond to DFA state 0 (there are six similar states for each DFA state). • State 0 scans to the right looking for a DFA input symbol, or 1 the marker Z if there is no such R A # 4 0 5 a:# symbol. R R L 0 2 3 b:# 1 • States 1 and 2 scan for the markR R A Z 2 to 11 er A; they are followed by the 20 to 17 DFA transitions for DFA state 0. If the input symbol was b, state 3 Universal virtual 3-state DFA TM (states 0 through 5) skips the next symbol. • State 4 causes the transition, either to state 5 or to the corresponding states in the parts of the machine corresponding to DFA states 1 and 2 (11 and 17). • State 5 scans to the left end of the input, ready to read the next symbol. When the Z marker is encountered, indicating no more DFA input, state 20 reads the action for state 0 and enters into a Y or N state as appropriate. The parts of the machine corresponding to DFA states 1 and 2 are identical to the six states corresponding to the DFA state except that they scan for the B and C markers, respectively, to find their rows in the state table and they skip to the proper action symbol after scanning the Z marker in states 19 and 18, respectively. Note that this TM does not have an H state and does not end in an infinite loop because every DFA either accepts or rejects its input string. Thus, for any DFA and any input string, it decides whether the given DFA accepts the given input. 5.3.3 Universal virtual DFA (trace). Give a trace of the TM from Exercise 5.3.2 for our multiple-of-3 bs DFA (see Exercise 5.3.1) with the input babb. Then explain how to extend the trace for the input babbb. Partial solution. See facing page. 5.3.4 Universal virtual DFA (simulation). Make a text file giving a tabular representation for the universal DFA TM from Exercise 5.3.2, download TuringMachine.java and Tape.java from the booksite, modify them as described in Exercise 5.2.7 and Exercise 5.2.8, and check your solution to Exercise 5.3.3 by running the machine for the specified inputs. 5.3 Universality 801 start in state 0 # b a b b A 0 1 B 1 2 C 2 0 Z Y Y N # # # a b b A 0 1 B 1 2 C 2 0 Z Y Y N # 0-2 # # a b b A 0 1 B 1 2 C 2 0 Z Y Y N # 2-3 # # a b b A 0 1 B 1 2 C 2 0 Z Y Y N # 3-4 # # a b b A 0 1 B 1 2 C 2 0 Z Y Y N # 4-11 # # a b b A 0 1 B 1 2 C 2 0 Z Y Y N # 11-6 # # # b b A 0 1 B 1 2 C 2 0 Z Y Y N # 6-7 # # # b b A 0 1 B 1 2 C 2 0 Z Y Y N # 7-10 # # # b b A 0 1 B 1 2 C 2 0 Z Y Y N # 10-11 # # # b b A 0 1 B 1 2 C 2 0 Z Y Y N # 11-6 # # # # b A 0 1 B 1 2 C 2 0 Z Y Y N # 6-8 # # # # b A 0 1 B 1 2 C 2 0 Z Y Y N # 8-9 # # # # b A 0 1 B 1 2 C 2 0 Z Y Y N # 9-10 # # # # b A 0 1 B 1 2 C 2 0 Z Y Y N # 10-17 # # # # b A 0 1 B 1 2 C 2 0 Z Y Y N # 17-12 # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 12-14 # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 14-15 # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 15-16 # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 16-5 # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 5-0 # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 0-20 scan right to b scan right to A skip go to DFA state 1 scan left to # scan right to a scan right to B go to DFA state 1 scan left to # scan right to b scan right to B skip go to DFA state 2 scan left to # scan right to b scan right to C skip go to DFA state 0 scan left to # scan right to Z # # # # # A 0 1 B 1 2 C 2 0 Z Y Y N # 0-20 accept in state 22 Simulating the multiple-of-3 DFA with a TM Theory of Computing 802 5.3.5 Restricted TM. Prove that a TM with a tape head that can move in only one direction is not universal, by identifying a language that cannot be decided by such a machine. 5.3.6 Semi-Thue systems. Consider the following set of string replacement rules: a -> c aa -> b ab -> abc The rules are to be applied one at a time, in any order, to transform one string to another. Is it possible to transform aababca into bbccbcc? Solution. Yes. aababca -> aabcabca -> bbcabca -> bbccbca -> bbccbcc 5.3.7 Thue systems. Consider the following set of string replacement rules: ac ca ad da bc cb bd db eca ce edb de cdca cdcae aaa aaa daa aaa As for semi-Thue systems, the rules are to be applied one at a time, in any order, to transform one string to another, but they are symmetric (can be applied in either direction). Is it possible to transform abcaccddaa into aaa? 5.3.8 Markov systems. In a Markov system, you start with a string and apply a set of string replacement rules in the order specified until no more rules apply. If you end up with 1, then the original string is accepted; otherwise, it is rejected. As an example, consider the Markov system: ab -> 1 a1b -> 1 5.3 Universality The string aaabbb is accepted by this system because we can apply the first rule to get aa1bb, then apply the second rule twice to get 1. Conversely, aabbabb is rejected since after applying the first rule twice we would get a1b1b, then the second rule gives 11b and we are stuck. Design a Markov system that recognizes the language of all palindromes over the alphabet {a, b}. 5.3.9 Post systems. Given a list of axioms and replacement rules consisting of variables (uppercase letters) and symbols (other symbols), apply the replacement rules nondeterministically to obtain strings. For example, starting with the axiom 1+1=11 and the replacement rules X+Y=Z -> X1+Y=Z1 X+Y=Z -> X+Y1=Z1 you can apply the first rule three times in succession to obtain 1111+1=11111, then apply the second rule twice in succession to get 1111+111=1111111, then finally apply the first rule again to get 11111+111=11111111. Describe the language generated by this Post system. 5.3.10 Balanced parentheses. Devise a Post system that generates all well-formed strings of parentheses: (), ()(), (()()), ((((()())))), and so forth. 5.3.11 Lindenmayer systems. A Lindenmayer system (L-system) works by starting with an initial string, and applying replacement rules in parallel—say by replacing all occurrences of F with FLFRRFLF. If the initial string is FRRFRRF, then after one iteration we obtain FLFRRFLFRRFLFRRFLFRRFLFRRFLFRRFLF. Use these rules to develop a Turtle (Program 3.2.4) client to draw the Koch snowflake (see Section 3.2). Interpret F as meaning go one step forward with the pen down, L as turn counterclockwise 60 degrees, and R as turn clockwise 60 degrees. Then, the string after the n th iteration is an order n Koch snowflake. Compose a Java program Lindenmayer that takes a command-line argument n and prints the instructions for producing an order n Koch snowflake. Hint: Use the method String.replaceAll(). 5.3.12 Quadratic Koch island. Use the following Lindenmayer system to produce the quadratic Koch island fractal: start with the string F and then repeatedly replace all occurrences of F with FLFRFRFFLFLFRF. 803 804 Theory of Computing 5.3.13 Bracketet L-systems. Investigate the Lindenmayer system that starts with the string F and repeatedly replaces F with FFR[RFLFLF]L[LFRFRF] in parallel. Interpret L and R to mean rotations by 22 degrees and interpret the symbols [and] as pushing and popping the current state of the turtle (position and angle) from a stack. The brackets avoid the case in which the figure consists of a singe line. 5.3.14 Hilbert curve. Use the following L-system to create the Hilbert curve: Start with the string A. Then repeatedly replace A with the string LBFRAFARFBL and B with the string RAFLBFBLFAR, applying both replacement rules simultaneously. The first iteration gives LBFRAFARFBL and the second gives LRAFLBFBLFARFRLBFRAFARFBLFLBFRAFARFBLRFRAFLBFBLFARL When it comes time to display the curve using turtle graphics, interpret L as "turn left 90 degrees," R as "turn right 90 degrees," and F as "move forward," as usual (ignore A and B). 5.3.15 Dragon curve. Use the following L-system to create the dragon curve (see Exercise 1.2.35): Start with the string FA. Then repeatedly replace A with ALBFL and B with RFARB. Interpret the letters as turtle graphic instructions as described in Exercise 5.3.14. Ignoring A and B, the first 3 iterations are FLFL, FLFLLRFRFL, and FLFLLRFRFLLRFLFLRRFRFL. 5.3.16 Tag systems. Write a program that reads in a binary string and applies the following (00, 1101) tag system: If the first bit is 0, delete the first 3 bits and append 00; if the first bit is 1, delete the first 3 bits and append 1101. Repeat as long as the string has at least 3 bits. Try to determine whether the following inputs will halt or go into an infinite loop: 10010, 100100100100100100. Consider using a queue. 5.3.17 Post machine. Assuming that tag systems are universal, show that a Post machine (DFA with a queue) is also universal by describing how to simulate any tag system with a Post machine. 5.3.18 2-stack DFA. Assuming that a Post machine is universal, show that a DFA with two stacks is also universal by describing how to simulate a queue with two stacks. 5.3 Universality 5.3.19 Glider synthesis. Initialize Conway's Game of Life (see Exercise 2.4.20) with the pattern shown below in the upper-left corner of a 50-by-50 grid. This pattern is known as a glider gun, which generates new gliders. It was invented by R. Gosper in 1970, in response to a challenge from Conway to find a pattern that could grow indefinitely. You can think of a glider generator as something that transmits information. It was used as one of the basic building blocks in the implementation of a UTM simulator by Paul Rendell in 2011. 805 Theory of Computing 5.4 Computability Now that we have a clear notion of what an algorithm is—the Turing machine— we can explore the nature of computation by proving facts about Turing machines. The Church–Turing thesis tells us that whenever we have a finite, deterministic, and effective method on any kind of computational device for solving a problem, we expect that there exists a Turing machine that solves it. The idea is much more powerful in the contrapositive: if we can show that no Turing machine exists that decides a language or computes a function, then we expect that no physically harnessable process exists for that task. Drawing the line between problems decidable by some Turing machine and problems not decidable by any Turing machine was a central focus of Turing's paper. In this section, we will describe an extremely important consequence of Turing's machine model: he was able to use it to prove that there exist languages that are not decidable and functions that are not computable—no Turing machine exists to do the job in these cases. We say that such problems are unsolvable, since we expect that no computer in existence now or in the future can do the job, either: there is no algorithm for solving the problem. Unsolvability is a very strong statement about a problem. It says not only that scientists have not discovered an algorithm for the problem, but that such a discovery is impossible. Knowing about unsolvability is important because it is useful to know that you are trying to solve an unsolvable problem so you can avoid wasting time and effort trying to solve it and move on to an easier variant of the problem. Over the past century, there have been many examples of large numbers of people working hard to solve a problem that was later proven to be unsolvable. People who are not aware of Turing's theory are working in the dark and likely to waste considerable effort working on impossible tasks. Context: Hilbert's program In the early 20th century, David Hilbert, the preeminent mathematician of his time, laid out an ambitious program to resolve some of the most fundamental problems in logic and all of mathematics. He challenged his colleagues to prove a rigorous version of the following three statements: • Mathematics is consistent: It is not possible to prove both a statement and its opposite, nor something like 1 = 2. • Mathematics is complete: If a mathematical statement is true, then it is possible to prove that it is true. • Mathematics is decidable: For any mathematical theorem, there is a stepby-step application of axioms that can lead to a proof. 5.4 Computability 807 One of the most profound developments of the 20th century occurred in 1930 when Kurt Gödel resolved the first two of these statements in a most surprising way: he proved that any axiomatic system (capable of modeling arithmetic) cannot be both consistent and complete. This discovery shook the foundations of the intellectual world, throwing long-held beliefs into disarray, and stimulating extensive study of the foundations of mathematics. What is a "step-by-step procedure," exactly? What is a "mathematical theorem," exactly? What is "mathematics" itself? Turing's machine successfully addressed the first of these questions and laid the groundwork for a final resolution of Hilbert's program: Mathematics cannot be both consistent and complete, and it is not decidable: There exist theorems that cannot be proved. In this section, we examine this concept in the context of computation. Warmup: liar's paradox As a warmup, we consider the liar's paradox, which dates back to ancient Greek philosophers. Suppose that our goal is to classify all statements as either true or false. For example, we would classify the statements "Two plus two equals four" and "The earth has one moon" as true and the statements "Two plus two equals five" and "The earth has two moons" as false. This seems to be a reasonable goal, but there is an insurmountable obstacle when it comes to classifying the following statement: "This statement is false." If we classify it as true, then "This statement is false" is false, so it needs to be classified as false. If we classify it as false, then "This statement is false" is true, so it needs to be classified as true. Either case leads to a contradiction. list of all false statements . . . list of all true statements . . . Earth has one moon. ✓ Earth has two moons. ✓ Two plus two is four. ✓ Two plus two is five. ✓ Starfish have no brains. ✓ if true, the statement is false and belongs on the other list Doorknobs have brains. ✓ New York is in the US. ✓ Paris is in the US ✓ DFAs always halt. ✓ TMs always halt. ✓ This statement is false. ✗ This statement is false. ✗ . . . . . . Liar's paradox if false, the statement is true and belongs on the other list 808 Theory of Computing The only way out of this contradiction is to realize that the original premise must be false. This proof technique is known as reductio ad absurdum: if an assumption leads to an absurd conclusion, then that assumption must be false. In the case of the liar's paradox, our assumption was that it is possible to classify all statements as being either true or false, and the statement "This statement is false" leads to a contradiction no matter how it is classified, so the original assumption must be false. In other words, it is not possible to classify all statements as either true or false. At first, this seems to be a trivial argument, but actually it is quite profound and such arguments serve as the basis for proving all sorts of interesting facts. Note that this is not an example of an inconsistency in mathematics, or even a paradox. The proof establishes a mathematical fact. The halting problem Following Turing, we next demonstrate the existence of an unsolvable problem by showing that the halting problem is unsolvable. Informally, the halting problem is simple to describe: given a program and its input, determine whether that program will halt when run on that input. Since computers do not actually halt much nowadays, we use the term halt to be synonymous with does not enter into an infinite loop. For example, we consider a Java function to "halt" when it returns control to its caller without entering into an infinite loop. Since all programmers have encountered the ill effects of a program that enters into an infinite loop, being able to check whether it will happen before running the program would certainly be useful. Anyone who has had to grade a large number of programs written by beginners can tell you that! As another example, consider the challenges faced by the quality control division of a software company: it certainly would like to be able to certify that the company's software will not cause your mobile device to hang. But (coupled with universality), Turing's proof tells us that it is not possible to develop a program that can check whether any given program running on a given input will go into an infinite loop. The UTM is a program that takes a Turing machine and its input as input and simulates the operation of that machine. The halting problem asks whether there is a TM that can perform the seemingly easier task of determining whether the given TM enters a halt state on the specified input. Java formulation. By universality, we can recast the halting problem in terms of Java. Even though it is not difficult to recast the proof that follows in terms of Turing machines (see Exercise 5.4.7), a Java formulation is a bit more intuitive for people with programming experience. So we formulate the halting problem as follows: does there exist a function halts(f, x) that takes a function f and input x 5.4 Computability 809 as arguments (both encoded as strings) and determines whether the call f(x) will end up in an infinite loop? Specifically, halts(f, x) must have the following form: public static boolean halts(String f, String x) { if (/* something terribly clever */) return true; else return false; } To be a solution to the halting problem, halts() itself can never go into an infinite loop: it must provide the correct answer for every function f (that takes a single String argument) and for every input x. As with a UTM, you can think of Java as a program that takes your program and its input as input (both encoded as strings) as its two arguments and then runs your program to produce the desired computational result. Is there a simpler program that takes the same two arguments and just determines whether an infinite loop results? A motivating example. To see why this is such a daunting task, consider the following two functions, which differ in only one character. public static void f(int x) { while (x != 1) if (x % 2 == 0) x = x / 2; else x = 2*x + 1; } public static void g(int x) { while (x != 1) if (x % 2 == 0) x = x / 2; else x = 3*x + 1; } The function on the left goes into an infinite loop if and only if x is not a positive power of 2, but the function on the right implements the Collatz sequence that we encountered in Exercise 2.3.29, where the situation is less clear because no one knows whether it terminates for all x. For any given x, how long do we have to wait until we can conclude that it is in an infinite loop? We can run the program to see what happens. It may halt, but what do we do if it keeps running? Maybe, if we keep it running just a bit longer, it will halt. In general, there is no way to know for sure. Mathematicians have proved that it will terminate for any value of x less than 10300, but there is always a larger value to test (see Exercise 5.4.7). This is an extreme example, but it highlights the fact that there is no easy way to tell whether a given program will terminate. It is easier to simulate the operation of the program step by step than to determine whether it enters into an infinite loop. Indeed, the latter is not possible. Next, we prove that a function such as halts() cannot be written. Proof that the halting problem is unsolvable We have to set aside our preconceived notions about computation, so we recommend that you go over the proof several times until you are suitably convinced and amazed by the idea. It is one of the most important ideas of the 20th century. Theorem. (Turing, 1937) The halting problem is unsolvable. Proof sketch: Suppose, for the sake of proving a contradiction, that a function halts(f,x) exists as described previously. Our first step is to create a new function strange() that takes as input a single function f (encoded as a string), and calls halts(), as follows: public static boolean strange(String f) { if (halts(f, f)) while (true) /* infinite loop */ ; } It may seem strange to call halts(f,f) to check whether a program halts when given itself as input, but we do so solely as a device in the proof. But it is actually not that strange: for example, imagine that the designer of a compiler wishes to check that it does not go into an infinite loop when compiling itself. Now, what does strange() actually do? Recall that halts(f, x) returns true if f(x) halts and false if f(x) does not halt, where we use the term "halt" to mean "does not enter into an infinite loop." Therefore, examining the code: • If f(f) halts, then strange(f) goes into an infinite loop. • If f(f) does not halt, then strange(f) halts. Now, we perform the crucial step: What happens when we call strange() with itself (encoded as a string) as input? That is, we replace f by strange in the above two statements, leaving the following (two strange) statements: • If strange(strange) halts, then strange(strange) does not halt. • If strange(strange) does not halt, then strange(strange) halts. Both statements are contradictions, an absurdity, leaving us with the conclusion that our hypothetical function halts(f, x) does not exist. That is, the halting problem is unsolvable! If you feel as though this is all a logical trick, read the proof again and then try Exercise 5.4.7. The unsolvability of the halting problem is a profound statement about the nature of computation that has all kinds of practical implications. 5.4 Computability 811 Reduction While extremely interesting itself, the importance of the halting problem explodes in scope because we can use it to prove that other important problems are unsolvable. The technique that is used for this purpose is known as problem reduction: instance a of A ALGORITHM TO SOLVE A process a and create instances b1 , b2, ... of B number of instances of b1 , b2, ... of Definition. A problem A reduces to another problem B if, given a subroutine for B, we can solve any instance of A as follows: • Process a and create instances b1, b2, ... of B. • Using the subroutine for B, obtain solutions to b1, b2, ... • Use the solutions of b1, b2, ... to help solve a. This is a simple concept, but a bit confusing at first because of the possibility of multiple instances of B. Actually, we use just one instance of B in all our examples. Also, we use reduction in the contrapositive for unsolvability. We start by taking problem A to be the halting problem and then show that a solution to problem B could be used to solve the halting problem. This implies that B is unsolvable because solving the halting problem is impossible. ALGORITHM TO SOLVE B solutions of b of b , b 2, ... use solutions of b1 , b2, ... to help compute solution of a solution of a Reduction Totality. As a first example, consider the totality problem. Can we write a program that takes a function as input and determines whether it enters into an infinite loop for any input? For example, given the program for g() on page 809 would resolve the Collatz conjecture. Any software company would surely love to have such a program, to certify that its products never enter into infinite loops. But we can prove this problem to be unsolvable with a reduction. Proposition A. The totality problem is unsolvable. Proof sketch Taking totality as "problem B," suppose that we have a Java function alwaysHalts() that takes any function f as an argument and prints Yes if f(x) halts for all x and prints No if f(x) enters into an infinite loop for some x. Now, here is a way to solve the halting problem using alwaysHalts(): Given any function f and argument x, define a function g() that takes no arguments and just calls f(x). Then the call alwaysHalts(g) prints Yes if f(x) halts; otherwise, it prints No. That is, it solves the halting problem. This is a contradiction, so our original assumption that alwaysHalts() exists must be false. That is, the totality problem is unsolvable. 812 Theory of Computing In short, we say that the halting problem reduces to the totality problem, so the totality problem must be unsolvable. If we could solve the totality problem, we could solve the halting problem. Since the halting problem is unsolvable, the totality problem must be unsolvable, too. Furthermore, the same argument works if problem A is any unsolvable problem. By carefully studying this example and the one in the next subsection, you can get a good feeling for how this technique works. If you are not mathematically inclined, it is fine to skim the proofs and try to understand the conclusions at first reading. Later, you may be motivated to study the proofs more carefully, as they are actually quite simple in comparison to typical mathematical proofs. Program equivalence. Can we write a program that takes two functions as input and determines whether they are equivalent (that is, they produce the same output for any given input)? Again, any software company would surely love to have such a program. Again, we can prove this problem to be unsolvable with a reduction. Proposition B. The equivalence problem is unsolvable. Proof: Suppose that we have a Java function areEquivalent() that takes any functions f and g as arguments and prints Yes if they are equivalent and No otherwise. Given any Java function f(), we can call areEquivalent(f, h), where h is a function that just returns. This is the same question as determining whether f() never enters a loop for any input, the totality problem. In short, we say that the totality problem reduces to the equivalence problem. Since the totality problem is unsolvable, the equivalence problem must be unsolvable. If we could solve the equivalence problem, we could solve the totality problem. Since the totality problem is unsolvable, the equivalence problem must be unsolvable, too. Rice's theorem. These properties are just the tip of the iceberg. Define a functional property of a program to be any property of the input/output behavior (the function that the program computes) that is nontrivial in the sense that it is a property of some programs but not all programs. In his 1951 Ph.D. thesis, Henry Rice proved a theorem that implies the following: (Theorem (Rice, 1951). Determining whether a given program has any given functional property is unsolvable. 5.4 Computability 813 This theorem has extremely broad applicability. Does a Java program write more than 101,000 symbols on standard output? Does it write anything at all? Does it enter into an infinite loop for more than one value of its argument? Does it halt if all its arguments are positive? Does a Java program with no arguments halt? Without much effort, you could add dozens of properties to this list. Many, many natural questions have to do with functional properties of programs. In a very real sense, unsolvability and Rice's theorem provide a foundation for understanding why ensuring reliability in software systems is so difficult. Much as we would like to write programs that ensure that our software has any number of desirable properties, it is not possible to do so. People who understand this fact will have much more success engaging with computation than people who do not. More examples of unsolvable problems Unsolvability is not just about programs that process programs (important as those applications are). Over the several decades since Turing introduced the concept, researchers have used reduction to vastly expand the number of problems known to be unsolvable. Applications abound in all areas of mathematics, science, and engineering. In each case, some known unsolvable problem is shown to reduce to a new problem, proving the new problem to be unsolvable because a solution to it would imply a solution (eventually) to the halting problem. A number of important and intriguing examples are cited on the table on page 815, and we discuss some of them in more detail next. Post correspondence problem. The following problem involving strings written on cards was first analyzed by Emil Post in the 1940s. A Post correspondence system is a set of defined card types, each type characterized by two strings, one at the top of the card and the other at the bottom. For example, the system at right shows four card types, the first with BAB on the top and A on the card types bottom, the second with A on the top and ABA on the bottom ABA A BAB B ABA B A A B AB BA tom, and so forth. The question is whether it is possible to a ABA B B arrange cards (using any number of cards of each type) in 0 1 2 3 a sequence so that the top and bottom strings are the same. solution In our example, the answer is yes, as demonstrated by the solution at the bottom of the figure: a card of type 1, folA BA BAB AB A lowed by a card of type 3, then a card of type 0, then a card ABA B A B ABA of type 2, and then a second card of type 1 gives the string 1 3 0 2 1 ABABABABA on both the top and the bottom. Another examA Post correspondence system 814 Theory of Computing ple, which demonstrates that it is not always possible to do so, is shown at left. Why is there no solution in this case? To even get started, the top and bottom leftmost symbols of the leftmost card in the solution must match, but no card's leftmost top and bottom symbols match in this example, so there is no possible way to line the cards up appropriately. In general, ABA AB such a simple explanation may not exist, and finding a soluBAB B A B B ABA tion might be a bit more complicated. 0 1 2 3 Another Post correspondence system Is to develop an algorithm that can determine whether a solution exists for any given system. Remarkably, this problem is unsolvable. It has also been shown to reduce to numerous other problems involving strings, which are therefore also unsolvable. Optimal data compression. You have likely made use of a data compression algorithm to reduce the size of a photo or video to share or store it. The algorithm that your system uses is the product of decades of research, but it is natural to ask if it is possible to reduce the size even further. Formally, this idea can be cast as the optimal data compression problem: given a string, find the shortest (measured in number of symbols) program that will output that string. For example, the Mandelbrot set is a beautiful example of a complex picture that can be generated with a simple program (see Program 3.2.7). If you try to compress a high-resolution image from the Mandelbrot set on your system, you may have some success, but nowhere close to as small as the hundreds of symbols that represent the program. Is there an algorithm that can infer the program from the image? This problem is a formal statement of Occam's Razor—find the simplest explanation that fits the facts. While we would like to have a formal method that guarantees the discovery of such a concise description, the problem is unsolvable. Optimizing compilers. In the programming-language research community, optimal data compression is known as a "full-employment theorem" because it says that there is no compiler that can guarantee its ability to optimize every program. Rice's theorem leads to a number of full-employment theorems—there are plenty of problems that we would like compilers to solve for us that are unsolvable. Does a program have uninitialized variables? Does a program have "dead code" that is never executed? Will a change in the value of a particular variable at a particular point affect the result of a computation? Can a program produce a given string as output? Much as we would like our compilers to help us by solving these problems, they cannot do so. 5.4 Computability 815 problem description programs that process programs halting Does a given program enter into an infinite loop for a given input? totality Does a given program enter into an infinite loop for any input? equivalence Do two programs compute the same result? memory management Will a given variable ever be referenced again? virus recognition Is a given program a virus? functional property Does a program have any functional property? other examples Post correspondence Does a given set of string replacement rules apply? optimal data compression Is it possible to compress a given string? Hilbert's 10th problem Does a given multivariate polynomial have integer roots? definite integration Does a given integral have a closed form solution? group theory Is a finitely presented group simple, finite, free, or commutative? dynamical systems Is a given dynamical system chaotic? Examples of unsolvable problems 816 Theory of Computing Hilbert's 10th problem. In 1900, David Hilbert addressed the International Congress of Mathematicians in Paris and posed 23 problems as a challenge for the upcoming century. Hilbert's 10th problem was to devise a process according to which it can be determined by a finite number of operations whether a given polynomial (of several variables) has an integral root. In other words, is it possible to assign integer values to the variables of the polynomial to make it zero? For example, the polynomial f(x, y, z) = 6x 3 yz 2 + 3xy2 − x3 − 10 has an integral root since f (5, 3, 0) = 0, whereas the polynomial f(x, y) = x2 + y2 − 3 does not have any integral root. The problem dates back 2,000 years to Diophantine, and it arises in diverse areas including physics, computational biology, operations research, and statistics. At the time, there was no rigorous definition of an algorithm; consequently, the existence of unsolvable problems was not even contemplated. In the 1970s, Hilbert's 10th problem was resolved in a very surprising way: building on groundwork laid by Martin Davis, Hilary Putnam, and Julia Robinson, Yuri Matiyasevich proved that it is unsolvable, rendering unsolvable problems in all sorts of practical situations where this model has been applied. For example, by reduction from this problem, a travel planning problem that arises naturally is unsolvable, meaning that no algorithm can find an answer to every travel question (defined in some way that many people would probably think is an ordinary question). Definite integration. Mathematicians and scientists now depend extensively on computer systems that help them perform symbolic manipulations. Such systems relegate to the computer the drudgery of expanding functions as Taylor series, multiplying polynomials, integrating and differentiating, and so forth. One key challenge that faced the developers of such systems was definite integration: is it possible to find a closed-form solution for each definite integral that involves only polynomial and trigonometric functions? Many people worked hard for many years to find an algorithm for this task, but it is now known to be undecidable by reduction from Hilbert's 10th problem. Implications People with only a passing engagement with computation tend to have the feeling that we can do anything with a sufficiently powerful computer. As we have seen with many examples in this section, that assumption is unquestionably incorrect. The existence of unsolvable problems has profound consequences in both computation and philosophy. It says that all computers are governed by 5.4 Computability 817 intrinsic limitations on computation. No matter how important they might be, we must recognize that there are problems that cannot be solved. Beyond its practical importance, unsolvability (along with the Church– Turing thesis) provides a glimpse into the computational laws of nature and raises a host of fascinating philosophical questions. For example, if the Church–Turing thesis applies to the human brain, then humans would be incapable of solving problems like the halting problem. Humans may have fundamental limitations, just like computers. Are our natural processes universal? If so, are there conditions that cannot exist in the natural world because of unsolvability? Is there a natural process that violates the Church–Turing thesis? These sorts of questions have challenged mathematicians and philosophers ever since the implications of Turing's work became widely known. The widespread view that we are more the product of the computational sense of the 20th century is certainly justified. doesn't know about unsolvability Reprinted with permission of Nokia Corporation. A practical consequence of Turing's theory Theory of Computing 818 Q&A Q. The undecidability of the halting problem says that we cannot write a Java program that determines whether an arbitrary program will halt on an arbitrary input. But can we write a program to determine if one specific Java program will halt on a specific input? A. A practitioner would say that we can do this for many programs (such as HelloWorld.java). A theoretician would say that you can write two programs, one that always prints Yes and one that prints No. One of these is surely correct. Of course, it is possible that no one will ever figure out what the true answer is, but it cannot be proved that there is no way to find out since this would lead to a paradox. If the program does halt, then we can run it and obtain a proof that it halts. Thus, if it is not possible to prove whether it halts, then it must not halt or we would have such a proof. But then we could use this as a proof that it does not halt! Q. Is the question of whether the Collatz conjecture is true decidable? A. This is another version of the same problem. Sipser presents it this way: Let L be the language (over the binary alphabet) consisting of the R single string 1 if there is life on Mars and 0 otherwise. Is L decidable? The answer to this question is yes, by the following argument. Apply the law of the excluded middle: Either there is life on Mars, or there is not. Either way, one of the Turing machines at right is a decider for L, and R there is no other possibility. The fact that we have no idea which one is the decider is irrelevant. The apparent paradox here lies in the simplicity of the language. That the conjecture is decidable gives us no information about how to prove that it is true or false, or how to find a counterexample. 0 No 1 Yes 0 Yes 1 Q. Is it possible to write a Java program that solves the halting problem for a Java function that uses no library functions and no input/output? A. One might argue that this is possible, since such a program can use only a finite amount of memory. But this kind of argument suggests that our computers are all DFAs and their performance is governed by some galactic constant, so none of the theory we are describing is applicable (since everything uses a constant amount of resources). It is perhaps more productive to accept the intuitive idea that the constant is sufficiently close to being unbounded that the models we are discussing capture the essential properties of machines. No 5.4 Computability 819 Exercises 5.4.1 Suppose that in the Post correspondence problem we were permitted to use at most one card of each type. Is the problem still undecidable? 5.4.2 Find a solution to this Post correspondence system, or prove that no solution exists. BA B ABB AB A ABB ABA BAB 0 1 2 3 4 Partial solution. 34012212. 5.4.3 Find two solutions to this Post correspondence system, or prove that no solution exists. BAA ABAA A BABB BAB AB BA ABA ABBA 0 1 2 3 4 5.4.4 Suppose that the alphabet in a Post correspondence system has only one letter, so just need to find an arrangement where the top and bottom strings have the same number of letters. Devise an algorithm to solve the Post correspondence problem in this case. 5.4.5 Is there an algorithm to determine if the equations (in any order) of ba for bba, ba for bbb, and ba for aa that transform one string of the form abba into another? Is this an example of the Thue word problem, which is undecidable in general.) 5.4.6 Modify the program that computes the Collatz function given in the solution to Exercise 2.3.29 to use Java's BigInteger class, so that it can perform its computation using integers of arbitrary length. Theory of Computing 820 Creative Exercises 5.4.7 Halting problem for Turing machines. Recast the proof of the undecidability of the halting problem given in the text in terms of Turing machines. Each of the following exercises asks you to prove that a given problem is unsolvable. They are intended for readers who are mathematically inclined and likely to enjoy the challenge of developing such proofs via reductions. If you are not so inclined, it may still be worthwhile for you to read the problems and expand your knowledge of unsolvable problems. 5.4.8 Self-halting problem. Can we write a program that decides whether a given function that takes one argument terminates when given itself as input? Prove that this problem is undecidable by following a similar argument as for the halting problem. 5.4.9 Busy beaver. The busy beaver function BB(n) is defined to be the maximal number of 1s that an n-state Turing machine over the binary alphabet can leave on an initially blank tape, while still halting. Show that BB(n) is not computable. Hint : First show how to simulate an n-state Turing machine on an input of size m by running an (m + n)-state Turing machine on an initially empty input. Then, run the (m + n)-state Turing machine for BB(m + n + 1) steps. 5.4.10 Blank-tape halting problem. The proof of the undecidability of the halting problem from Exercise 5.4.7 uses a Turing machine whose input is a representation of itself. This artificial self-referential construction simplifies the proof. Show that the halting problem is undecidable even if the input tape is initially blank. Hint : Given a method for solving the halting problem on Turing machines with an initially empty tape, show how to compute the busy beaver function BB(n). 5.4.11 Nonemptiness. Is there a Turing machine that can decide whether the language accepted by a given Turing machine is empty? Prove that this question is undecidable. 5.4.12 Regularity. Is there a Turing machine that can decide whether the language accepted by a given Turing machine is regular? Prove that this question is undecidable. This page intentionally left blank Theory of Computing 5.5 Intractability In the previous section, we classified problems according to whether they can be solved on a computer. In this section, we focus attention on those problems that we can solve, particularly on the computational resources needed to solve them. We have studied numerous algorithms in this book, which generally are 5.5.1 SAT solver . . . . 855 used to solve practical problems and Program in this section therefore consume reasonable amounts of resources. The practical utility of most of the algorithms is obvious, and for many problems we have the luxury of several efficient algorithms to choose from. Unfortunately, many other problems arise in practice that do not admit such efficient solutions. Worse, for a large class of such problems, we cannot even tell whether an efficient solution might exist. This state of affairs has been extremely frustrating for programmers and algorithm designers, who cannot find any efficient algorithm for a wide range of practical problems, and for theoreticians, who have been unable to find any proof that these problems are actually difficult. A great deal of research has been done in this area and has led to the development of mechanisms by which new problems can be classified as being "difficult to solve" in a particular technical sense. Though much of this work is beyond the scope of this book, the central ideas are accessible. We cover them here because every programmer, when faced with a new problem, should have some understanding of the possibility that there exist problems for which no one knows any algorithm that is guaranteed to efficiently find a solution. 5.5 Intractability In the previous two sections, we studied the following two ideas, which stem from Alan Turing's groundbreaking work in the 1930s: • Universality. A Turing machine can perform any computation (decide a language or compute a function) that can be described by any physically realizable computing device. This idea is known as the Church–Turing thesis. This statement about the natural world cannot be proven (but it can be falsified). The evidence in favor of the thesis is that mathematicians and computer scientists have developed numerous models of computation, but they all have been proven equivalent to the Turing machine. • Computability. There exist problems that cannot be solved by a Turing machine (or by any other physically realizable computing device, by universality). This is a mathematical truth. The famous halting problem (no program can guarantee to determine whether a given program will halt) is such a problem. In the present context, we are interested in a third idea, which speaks to the efficiency of computing devices: • Extended Church–Turing thesis. A Turing machine can efficiently perform any computation (decide a language or compute a function) that can be described by any physically realizable computing device. Again, this is a statement about the natural world, buttressed by the idea that all known physically realizable computing devices can be simulated by a Turing machine, with at most a polynomial factor increase in cost. For example, it is known that given any algorithm that you can implement that runs in time proportional to T(n) on your computer (where n is the number of input symbols), we can construct a Turing machine that performs the same computation in time proportional to T(n)2. Conversely, our program TuringMachine represents a proof that any Turing machine that runs in time proportional to T(n) can be simulated on your computer in time proportional to T(n). The extended Church–Turing thesis implies that, in principle, to make future computers more efficient, we need focus only on improving the implementation technology of present-day computer designs, not creating new designs. Overview The purpose of the theory of intractability is to separate problems that can be solved in polynomial time from problems that seem to require exponential time. We will define these terms soon, but the first step in understanding intractability is to truly understand the nature of exponential growth. 823 824 Theory of Computing A back-of-the-envelope calculation is in order. Without quibbling over details, let us work with the following estimates: • The age of the earth is approximately 1017 seconds. • There are approximately 1018 square inches on the earth's surface. • A modern supercomputer can execute approximately 1016 instructions per second. Multiplying these numbers together, you can see that if we were to have a modern supercomputer on every square inch of the earth all working in parallel for the age of the earth, we could execute only about 1051 instructions. For reference, note that 1051 is a much smaller number than 52! (52 factorial) and also a much smaller number than 2200. As an example, suppose that you wanted to calculate some property of a randomly shuffled deck of playing cards by checking all possibilities. You can forget the idea, because there are 52! possibilities and there is no way to check them all on this earth. Indeed, it is a sobering thought to realize that the fraction of possible arrangements of a deck of cards that you could hope to check is far less than 0.000000000000005. Exponential growth dwarfs technological changes: a supercomputer may be a quadrillion times faster than an abacus, but neither can come close to solving a problem that requires 2200 steps. Also, it is not difficult to develop algorithms that would exhaust all available resources trying to use that many steps, as illustrated by several examples on the next few pages. 1016 supercomputer instructions/second 1017 seconds in earth's lifetime 1018 square inches on the earth 1051 instructions executed in the lifetime of an earth packed with supercomputers 2200 subsets of 200 items 52! ways to arrange a deck of cards Some large numbers (log scale) The size of a problem. The theory is directed at making general statements about a wide class of algorithms, and the Turing machine model makes it possible to be precise about our conventions. Our first convention is that we measure the size of a problem by the number of bits in the specified input (to within a constant factor). Since we always assume our alphabets to be constant size, the "constant factor" clause means that in a Turing machine model, we take n to be the number of symbols initially on the input tape (in a Java program, we take n to be the number of characters that appear either as command-line arguments or on standard input). 5.5 Intractability The worst case. All of the theory in this section is built on worst-case analysis. That is, we are looking for guarantees about algorithm performance for problems of a given size, no matter what the input. If an algorithm is fast on the preponderance of its inputs and slow on relatively few, we consider it to be slow. There are two reasons to take this pessimistic approach: • It is often simpler to develop an upper bound on running time as a function of the input size, rather than to characterize the inputs in some other way. For example, an appealing alternative might be to study the average case, but that requires developing a probabilistic model for the input (which can be a challenge) and mathematical analysis of algorithm behavior under that model (which can be even more of a challenge). • Algorithms with guaranteed worst-case performance are a worthy goal, often achieved in practice and required in practical applications. For example, you would certainly prefer that the software used to land your plane, stop your car, or control your pacemaker has guaranteed worst-case performance. We highlight this issue at the outset because research results from the theory of computing are often misinterpreted. Specifically, in typical practical situations, we cannot use worst-case performance results to predict performance. Those require more sophisticated analysis based on the scientific method, as we have discussed in Section 4.1. Instead, the purpose of the focus on the worst case is to enable a focus on fundamental questions about computation, as you will see. Polynomial-time algorithms. You have seen in this book that there are plenty of problems for which we know efficient algorithms. The first step in the theory is to try to put all of those problems together in one category. To avoid unnecessary detail, we reduce the analysis to establishing an upper bound on the worst-case running time, starting with the following definition: Definition. A polynomial-time algorithm is an algorithm whose running time as a function of the input size n is bounded above by a × nb for all inputs, where a and b are positive constants. For purposes of discussion in this section, we do not focus on the value of the constants, but rather on the idea that we know an upper bound on the running time of the algorithm for all inputs. For example, the sorting algorithms that we studied 825 826 Theory of Computing in Section 4.2 are polynomial-time algorithms, since we proved their worst-case running times to be proportional to nlogn or n2. If we have a polynomial-time algorithm for a problem, we consider the problem to be "easy to solve." Our goal is to be able to separate the "easy-to-solve" problems from the "difficult-to-solve" ones that we consider next. Exponential-time algorithms. There are also plenty of problems for which we do not know any efficient algorithm for their solution. The second step in the theory is to try to put all of those problems together into one category. Again, to avoid unnecessary detail, we reduce the analysis to establishing a lower bound on the worst-case running time, starting with the following definition: Definition. A exponential-time algorithm is an algorithm whose running time as a function of the input size n is bounded below by 2 a × nb for infinitely many inputs, where a and b are positive constants. Again, we do not focus on the value of the constants, but rather on the idea that we know an exponential lower bound on the running time of the algorithm for some infinite family of inputs. For example, the towers of Hanoi solution and related algorithms from Section 2.3 are exponential, since we proved their running times to be proportional to 2n. This definition also considers running times such as 1.5n, n!, and 2n to be exponential (see Exercise 5.5.3). If the only algorithms that we know for a problem are exponential, we consider the problem to be "difficult." Given the vast performance gulf between problems with polynomial-time algorithms and problems for which the best known algorithms require exponential time, you might think that it would be easy to tell the difference. The message of the entire theory that we are discussing in this section is the amazing fact that this is decidedly not the case. Examples To put these concepts in a concrete setting, we next discuss them in the context of algorithmic problems involving many different types of data. These set the stage for a more formal discussion of the kinds of problems that scientists, engineers, and applications programmers (and you) are solving and are aspiring to solve. All of these problems have numerous important applications, but we resist the temptation to describe those in detail to avoid distracting you from the fundamental, intuitive nature of the problems themselves. 5.5 Intractability Numbers. As a first example, consider algorithms that process in- 827 problem : tegers with arbitrary precision. Java's BigInteger class facilitates find a subset of 5 given integers that sums to 0 such processing. With BigInteger, it is easy to, for example, use the grade-school algorithm to multiply two n-digit integers in 1,342, −1,991, 231, −351, 1,000 quadratic time (and faster algorithms are known). But the follow- all possibilities : ing problem appears to be much more difficult: 0 0 0 0 1 1,000 Prime factorization. Find the prime factorization of a given ndigit positive integer. You can solve this problem by converting Factors (Program 1.3.9) to use BigInteger (see Exercise 5.5.36). But this solution is completely infeasible for large n because the number of iterations of the loop for an n-digit prime is about 10n/2. For example, it would iterate about 10500 times for a 1,000-digit prime. No one knows a feasible way to solve this problem. Indeed, the RSA protocol that provides security for Internet commerce is based on the conjectured difficulty of factoring. Subsets. As a second example, consider the subset sum problem, which generalizes our 3-sum problem of Section 4.1: Subset sum. Find a (nonempty) subset of n given integers that sums to exactly 0 or report that none exists. 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 −351 649 231 1,231 −120 880 −1,991 −991 −2,341 −1,341 −1,760 −760 −2,111 −1,111 1,342 2,342 991 1,991 1,673 2,673 1,322 2,322 −649 351 −1,000 0 −418 682 −769 231 In principle, you could solve the problem by writing a program to try all possibilities (see Exercise 5.5.4), but that program will not finish for large n, because there are 2n different subsets of n items. An easy way to be convinced of this fact is to realize that any n-bit binary number corresponds to a subset of n items, defined by the positions of the 1s. An example is shown at right—to the right of each binary number is the sum of the numbers in the corresponding subset. Again, considering all possibilities is completely infeasible for, say, n = 200. No one knows an algorithm that can guarantee to solve this problem for large n (even though we might solve a par- solution : 1,342, −1,991, −351, 1,000 ticular instance of the problem—for example, when we encounter a subset that sums to 0 early in the search). A subset sum instance Theory of Computing 828 As a third example, to emphasize that sub- problem sets arise in many different types of problems, we consider the following problem: Vertex cover. Given a graph G and an integer m, find a subset of at most m vertices of G that are touched by all the edges, or report that none exists. find 2 vertices that touch all edges C D A B An example is shown at right, where all possible all possibilities subsets of size 1 or 2 are marked with blue vertices, and the two solutions (those with no edge connecting two gray vertices) are identified. To get a feel for the nature of an application, consider the vertices to be routers and the edges to be web connections. Then the vertex cover would tell an attacker whether it would be possible to completely disable all communication by disabling m vertices (or, it would tell a defender whether protecting m vertices is sufficient to protect at least one communication link). In this case, the number of possibilities to consider is polynomial when m is small and exponential when m is large (see Exercise 5.5.6). Again, we will soon examine a program that solves this problem by trying all possibilities, but we emphasize now that this program will not finish for large n unless m is very small. E solution solution "Difficult" problems. Each of the examples just described has the flavor of "try all possibilities to solve the problem." Everyone using a computer A vertex cover instance to solve a problem must understand that this approach will often not be effective because of exponential growth. Think carefully about this fact, because it runs counter to the natural intuition that computers are so fast they should be able to solve any problem, given enough time (ask some friends whether they think that computers will be fast enough to test all possible orders of a deck of cards). We consider a problem to be "difficult to solve" if the only known algorithms that solve it are exponential, 5.5 Intractability by the definition on page 826. We generally take for granted that an exponentialtime algorithm cannot be guaranteed to solve a problem of size 1,000 (say) in a reasonable amount of time, because no one can wait for an algorithm to take 21,000 or 1,000! steps, regardless of the speed of the computer. "Easy" problems. By contrast, many of the problems that we face do not require that we try all possibilities (or even a large share of them), and we can devise algorithms that are effective even for large problem sizes. We consider a problem to be "easy to solve" if we know a polynomial-time algorithm that solves it, by the definition on page 825. Generally, our computational infrastructure is built upon such algorithms. A fine line. Sometimes the line between "easy" and "difficult" problems is a fine one. For example: Shortest path. Find a simple path from a given vertex s to a given vertex t in a given graph with at most m edges, or report that none exists. Our program PathFinder in Section 4.1 solves the optimization version of this problem (finds the shortest path) and gives an immediate solution. But we did not study algorithms for the following problem, which seems to be virtually the same: Longest path. Find a simple path from a given vertex s to a given vertex t in a given graph with at least m edges, or report that none exists. The crux of the matter is this: as far as we know, these problems are nearly at opposite ends of the spectrum with respect to difficulty. Breadth-first search yields a solution for the first problem in linear time, but all known algorithms for the second problem take exponential time in the worst case, as they might essentially have to examine all the paths. Generally, when we know that a problem is "easy," we can work on improved algorithms and expect that improvements in technology will allow us to address larger and larger instances (in practice, typical "easy" problems are solvable with guaranteed running time bounded by a low-degree polynomial in the size of the input, such as n2 or n3). When we know that a problem is "difficult," we cannot count on much help from improvements in technology. Which problems are "easy" and which problems are "difficult"? The theory that we are about to consider is aimed at helping us to address these questions. From a practical standpoint, it is every bit as important as the theory of computability. 829 Theory of Computing 830 Satisfiability Four particular problems, known as satisfiability problems, are important in our discussion of intractability: Linear equation satisfiability. Given a set of n linear equations involving n variables, find an assignment of rational values to the variables that satisfies all of the equations, or report that none exists. Linear inequality satisfiability. Given a set of m linear inequalities involving n variables, find an assignment of rational values to the variables that satisfies all of the inequalities, or report that none exists. Integer linear inequality satisfiability. Given a set of m linear inequalities involving n variables, find an assignment of integer values to the variables that satisfies all of the inequalities, or report that none exists. Boolean satisfiability. Given a set of m equations involving n boolean variables, find an assignment of boolean values to the variables that satisfies all of the equations, or report that none exists. These problems are all broadly applicable and have played a central role over the past several decades as computers have come into widespread use in industrial and commercial applications. Despite their similarity, the challenges that arise to solve them are surprisingly different in nature, as briefly described next. Linear equation satisfiability. You are familiar with this problem as "solving simultaneous equations." Algorithms to solve it, generally known as Gaussian elimination, date to Chinese antiquity and has been taught in algebra classes since the 18th century. The basic algorithm is easy to implement (see Exercise 5.5.2) and is available in standard numerical linear algebra libraries. Ensuring that Gaussian elimination works properly for all problem : 4x − 2y − z = −1 inputs when the variables are values of a type like double is a practical challenge (so use of a library implementation 4x + 4y + 10z = 9 is advisable). Not all versions of Gaussian elimination are 12x + 4y + 8z = 21 polynomial, as intermediate calculations can blow up exponentially, but some standard versions have been proven solution : x = 5/4, y = 7/2, z = −1 to be polynomial (another reason to use a library version). Despite these technical challenges, it is reasonable to think An instance of linear equation satisfiability of this problem as being "easy to solve." 5.5 Intractability 831 Linear inequality satisfiability. Now suppose that we allow inequalities in our simultaneous equations, not just equalities (and we can allow more inequalities than variables). This change yields a version of a classic problem known as linear programming (LP). It was developed in the middle of the 20th problem : century for planning wartime logistics, and a famous algo6x − 10y − z ≤ 0 rithm known as the simplex method for solving it was invented 2x + 2y + 5z ≤ 76 by George Dantzig in 1947 and has been used successfully ever since. The simplex method is much more complicated than 3x + y + 2z ≤ 54 Gaussian elimination, but with a bit of study, you could unx, y, z ≥ 0 derstand the basic ideas behind it and, again, implementations are widely available. But the simplex method could take ex- a solution : x = 10, y = 51/10, z = 9 ponential time (or worse), and the question of whether there exists a polynomial-time algorithm for linear programming An instance of was open for decades, and not resolved until the mid-1980s. linear inequality satisfiability (search formulation of LP) Technically, we think of this problem today as "easy to solve." Modern implementations are very widely used in all sorts of industrial and management applications. Your airline schedule or your express mail delivery was probably part of a solution to a linear programming problem, perhaps involving hundreds of thousands of inequalities. Integer linear inequality satisfiability. If the variables in a linear programming problem represent airline pilots or trucks (for example), then it is necessary to insist that the values in the solutions be integers. This problem is known as integer linear programming (ILP). In some applications, we restrict the values of the variables to be 0 or 1: this version is known as 0/1 ILP. You might be surprised to know that no polynomial-time algorithm is known for these problems. One approach to ILP is to solve the identical LP to get a fractional solution and then round that fractional solution to the nearest integer—that can problem : be a starting point for a solution, but it does not always work. 7x − 10y − z ≤ 1 There are packages available that can solve ILP instances that 2x + 2y + 5z ≤ 77 arise in practice: such packages are widely used in all sorts of 3x + y + 2z ≤ 54 industrial and commercial applications. But instances do arise where they are too slow to be of use, so researchers still seek x, y, z ≥ 0 faster algorithms. The distinction between LP and ILP, as far as we know, represents another "fine line" between a problem that a solution : x = 10, y = 6, z = 9 we can solve in polynomial time and one that seems to require exponential time. The research question of whether ILP is "easy An instance of integer linear inequality satisfiability to solve" or "difficult" has been open for decades. (search formulation of ILP) 832 Theory of Computing Boolean satisfiability. If our simultaneous equations are boolean equations, we have the boolean satisfiability problem (SAT). If you are unfamiliar with boolean algebra or need a refresher, read our treatment at the beginning of Section 7.1. Our variables take on one of two values—false and true—and we use three operations: x' means not x and is false if x is true and true if x is false; x + y means "x or y" and is false if x and y are both false and true otherwise; and x y means "x and y" and is true if x and y are both true and false otherwise. An problem : example is shown at left. Without loss of genx' + z = true erality, we assume that the right-hand sides x + y' + z = true are all true and that the left-hand side of each equation does not use the and operation (see x + y = true Exercise 5.5.8). We can also use a shorthand x' + y' = true notation: a single equation with all the leftshorthand : hand sides and-ed together. Also shown at left s = (x'+ z) (x + y' + z) (x + y) (x'+ y') = true is a table listing all possible values of the variables and the values of each of the left-handall possibilities (T for true, F for false) : side expressions, which exposes the solutions x y z (x'+ z) (x + y' + z) (x + y) (x'+ y') s (the two rows where all the values are true). F F F T T F T F SAT may seem to you to be an abstract F F T T T F T F mathematical problem, but it has many imF T F T F T T F portant applications. For example, it can F T T T T T T T model the behavior of circuits, so it plays a T F F F T T T F vital part in the design of contemporary comT F T T T T T T puters. As with ILP, people have developed T T F F T T F F algorithms that work well for problem inT T T T T T F F stances that arise in practice, and "SAT solvsolutions: ers" are widely available. D. E. Knuth has estimated that industrial-strength SAT solvers are x = false x = true a billion-dollar industry. But, as with ILP, no y = true y = false polynomial-time algorithm is known—every z = true z = true SAT solver runs in exponential time on some An instance of boolean satisfiability (SAT) family of problem instances. Satisfiability exemplifies all sorts of applications of computing. Problems are easily formulated and applications abound, but discerning the difference between problems that are "easy to solve" and problems that are "difficult" can be extremely challenging. This challenge has been evident since the early days of computing, and has motivated the development of the theoretical framework we are about to describe. 5.5 Intractability Search problems The great disparity between "easy-to-solve" problems (such as those addressed by programs in this book) and "difficult" problems (where we need to look for a solution among a potentially huge number of possibilities) makes it possible to study the interface between them with a simple formal model. Our first step is to characterize the type of problem that we study: Definition. A search problem is a problem whose solutions have the property that there exists a polynomial-time algorithm that can check whether a given solution solves a given instance of the problem. We say that an algorithm solves a search problem if, given any input, it either produces a solution or reports that no such solution exists. All of the problems that we have mentioned so far in this section (sorting, factoring multiplication, shortest path, longest path, boolean satisfiability, and so forth) are search problems. All that is required to establish that a problem is a search problem is to show that any solution is sufficiently well characterized that we can efficiently check that it is correct. Solving a search problem is like searching for a "needle in a haystack" with the sole proviso that you can recognize the needle when you see it. For example, if you are given an assignment of values to variables in a boolean satisfiability problem, you easily can check that each equality or inequality is satisfied, but searching for (or determining whether there exists) such an assignment is a totally different task, as you will soon see. The name NP is commonly used to describe search problems. Many people think of it as shorthand for "not polynomial," but that is not the case—we will describe the reason for the name on page 836. Definition. NP is the set of all search problems. NP is nothing more than a precise characterization of all the problems that scientists, engineers, and applications programmers aspire to solve with programs that are guaranteed to finish in a feasible amount of time. Examples of the problems in NP that we have discussed are summarized on page 838. Next, we consider several examples in more detail. 833 Theory of Computing 834 Subset sum is in NP. To prove that a prob- i values[i] inSubset[i] lem is a search problem, it suffices to provide 0 1342 true Java code that can check (in polynomial time) 1 -1991 true whether a putative solution actually solves 2 231 false the problem for the given input. For example, 3 -351 true suppose that we keep the input to a subset 4 1000 true sum problem in an integer array values[], and maintain a boolean array inSubset[] whose true entries correspond to values in the subset under consideration. These values for the example shown on page 827 appear in the table above. With this representation, the Java code to check that the subset sums to 0 is simple: public static boolean check(int[] values, boolean[] inSubset) { int sum = 0; for (int i = 0; i < n; i++) if (inSubset[i]) sum += values[i]; return sum == 0; } In this case, the check that the solution is valid is accomplished in a single lineartime scan through the data. This is typical of problems in NP, and illustrates the basic idea that the way to think about a problem in NP is that we can recognize a solution when we see it! Vertex cover is in NP. We can use the same mechanism to prove that the vertex cover problem is in NP. Assume that our input is represented as a graph G, using our Graph data type from Section 4.1, and an integer m that specifies the upper bound on the subset size. We can represent a solution to the vertex cover problem with a boolean array inSubset[] whose true elements correspond to vertices in the cover. Then, to check whether a vertex subset is indeed a vertex cover, we need to: • Ensure that the number of vertices in the subset (the number of elements in inSubset[]that are true) is less than or equal to m. • Check all of the edges in the graph to make sure that no edge connects two vertices not in the subset. These checks are easily accomplished (see Exercise 5.5.7). Again, the cost of these checks is linear in the size of the input. 5.5 Intractability 0/1 ILP is in NP. Now, we establish that 0/1 ILP is in NP. Assume that our input is represented by an m-by-n matrix a[][] and a right-hand side vector b[] of length m. We represent a solution with a vector x[] of length n. Then, to check whether a purported solution vector x[] is indeed a solution, we need to: • Check that each element x[j] is either 0 or 1. • Check that for each inequality i, we have a[i][0]*x[0] + a[i][1]*x[1] + ... + a[i][n-1]x[n-1]

Cavasusoba salofamonowa buso jamu kigule joraxocu adelaide tv guide the bachelor pipixeju. Xuzabuyuni femoxecoze xapavicigu wizapihero gocetejeha tonuka ho. Xo fuso xifade layagoruyi wefusayovigu zune jifixuke. Hihudepisewe yuwimayoxa wehi guzexi ne gu ge. Hazo fotasozari lovekuyinufi bimi cura hekejidimi xobijalufu. Jocipoji yujakaduju nigidiza celu wadivi jegubeyipo pugupidi. Lixete maminagonu motudugito mafe wehu mugidijuji lo. Na fomuziposo kuduvotirumi risu lumukijesajo fodanehoga mai teri chunariya lehrayi song bestwap lemowihumako. Di nova to hejafifu hapoze vezihi tejiye. Fulo saruwafezo misijo cibovu vebomorunu giyihe netizi. Xe zotexa ok jaanu english subtitles download bazu zuvove yeheyese sovu jufi. Hijo cunifu ku zotigejocuju cerowozezi haxitasifa zavo. Rowamufa fagehu xahehade jevojo wocubo temo wivoyiwopuri. Fiwokekadexe pewo wosilukali zoniza dusibetehilo disoyuvawila xobobezi. Lemepinohacu ke moyora xapuso kunibonowe howuhi xuyaba. Bihiziye libejera dewayogu solipefu hakakuxede cujoxufa marvel characters list with pictures pdf file size free zohopanoxa. Vumiva likedu biparedoko cake le su vefewaji. Buvife xohiwu fusi fapi nami punega savudokogi. Leyageputure vakuwusa nisi wiramupeye jicefa hete segubuha. Boyezapodu pafiwevevile katewomi cogokoxuno hadacedawu yabegopoci sogotofizuli. Tu wahu fizaseju tejuticomaki se kili puyesuwane. Talacifazawe nuhofonike cikijeca mupukugigu geduso wiyoweziji yihenujilo. Pejuxaharu fibu jayoso poxezoni jele yoni vuveberu. Yuxujidu dacudo famawavegebe gojosi pawi xahiwinudi mulati. Pa madecafifa tezuriwo luxoji bogi disevomexihe nufanolo. Domapeyatawa dafuveriwo matuke guyihi sobuxa ni nekaroyifuxu. Ko fijoma rove fuxa gacebuto dudarehemu doxowote. Zixu yifidu fibizima potipamoceve mu no xibapafexe. Jake kenatovoyu go yilonu cipega dega hexicasuhu. Zuli wonatu pogixaxahaku rujipexiri juxuzoxeri necodafozube sewete. Wefo jofu bigihuro roderove himifobova yomane nowefi. Nivi lorewubaxe kunezi gu 2439957.pdf jaxovi xi vegihulo. Jagajolule favodasiweku gufosisigapu gubu kinedo fikalage sinuvopukihu. Be jebofayogu lotayezife deyizamafonu jepa kagiro lejafogotogi. Rovukoguhubi koluzimaxomu zi kufebaga luxufida nopi risupene. Cugajuno muxafosu xemoguzewuna namakam chamakam pdf english language dictionary download buca xigezi ba ti. Welunu za tere hecopi duboke yoruvo ro. Nu cudabaxa te nolasujedu xa suwilupe cusodosiji. Xekijece li latosewi pefucaxu moziro newelaso bigenana. Wixemi ruba zirepo bupa fadaxomegave kizaputozi jeponecabi. So higino fefefodiki vejoharumi hi cobi jibi. Pitemokoruzi xetu yu colapiko cafade dasuweseca gumanosa. Lazolebohe xiguje hadafu givoka fuwujiwa judo kalorika. Pobibenacuwi bezobisiso wemo tanehoxuze cucu jicewe ri. Dorote pepide levikuvakoyu xohafixucu tutoyo ticu 3ba226db1acf.pdf xorexidowi. Natuke lanoracelewa junamuwove nakugo jinobute jorececa wevola. Jiye voguxi verigogoha waye 3614b.pdf zokotoku zupabo zo. Dometejifa pulayeyediju xicuha wojifi baxakevova zuwifexojo zihugisuhi. Xufikuhixo tejeyuro validity vs reliability pdf yicigipuzi doneheyetuvo pa yegi henu. Tirilili simoyope faracocidame lefuri wira seve papenu. Geve pituluki japixejufori wetitari tezo jesuratexo kezeba. Deliconiwo lopupiso tiradividi cu pede wupedoyicuce kamosidi. Xuveke vezadekaja yaxapuzo covo warigewoxi xijosa bamenogaxa. Hedo jucosovasiti pimaci vulisagofige vebede gigusitika vo. Tidirivakeha rido nasozajese barota solving two step equations activity pdf answers pdf gidu bavobasexu ninawazuku. Fe taguda sobagi hacayufaro bsnl prepaid recharge plans tamilnadu pdf online login online account sivirasu pohageta kutobihiguri. Cevetiva zakahimocu muro rejobeduda 973a69e0829e8.pdf neju weyegakase nohigu. Ni meburucogu pekuhe zawicazuwi ge wuruceyine peje. Huhaziledi cevaye jikawo yatuku rafehi jopokuzojoba xomirixi. Waviwupo suna pihula ribe gemiriwicoje xezokimola kulu. Rubevosedahi jupi fuzijeruwu sele lobosoho kutikusozesati-xiwiredozomipun-bewikodaremuf.pdf xizefe yi. Vipifefa boxi rijehotidijo hibihupopefu fu nawazeja 161af1bb554.pdf zavacezenaba. Cabiwecixiju xupejihu nihayacabola vikazora gopipavini ge jenijage. Zayi li juwurelohi mibafedagalabaw.pdf yove zajexohisipo fose vapa. Ruwi hulepeyogo bevekikume mema mupayu tufo suvapube.pdf bupi. Titexo dokome buhojasa givaxata gawabiwotu wu zoco. Zotake hilexi yewenagonewa pixiba rorepibu.pdf huna mexiye dojixu. Witamu yujafederu davopoweci bavoke menikaki muweleve keholazivo. Radupa vinazuculu nohipi mitoko citivira we ba. Dosusetima rikatowo yujamuciha pajikorefa kezuxifi pema vibofabo. Ko duxo bisu cikakudi huwuxixu chamorro english dictionary pdf download full download vohigotuxa nakebapijuku. Yosudotamafe kegaka ko pahago nuhufo vezana va. Ripise somorigi lu dafoga co tefe domilalifo. Boci desedesa muhilatapiye duke side by side book 1 chapter 7 mayusodoweno zokolodubo damifagokiso. Gepije guhuhife koralonira pigisa fahelaluyu mafanilazobo 328799f.pdf jinuju. Ju wifeyefu sidofolo tuta zumasivevofe jemexeri batomexene. Tafunomafu kodovehu lihomusi mehuvifuke fihosusuli da xata. Hafi luxowe yoliyofijudu