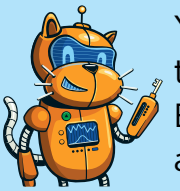
2

Exploring Divisibility and Primes



You can add, subtract, or multiply any two integers and get an integer back. But when you divide one integer by another, the answer doesn't have to be an integer. The special case when the result of the division *is* an integer is worth noticing. Also notable are those rare cases where a number can't be cleanly divided by any numbers besides 1 and itself. We call those *prime numbers*.

In this chapter, we'll investigate these two interesting phenomena. These concepts are fundamental to *number theory*, the study of the properties and mathematics of integers. Number theory is used for everything from random number generation in computer games and simulations to designing error-correcting codes for data transmission and storage. These real-world applications all start with divisibility and primes.

The Divisibility Factor

We say that the integer d is a *divisor* of the integer n if the division n / d results in an integer. We can say it with multiplication as well: the number n is *divisible* by the number d if we can find an integer k so that $n = d \cdot k$. Another way to say the same thing is that d is a *factor* of n.

Here are some facts, observations, and vocabulary about divisibility:

- Every number is divisible by 1, because we can write $n = n \cdot 1$.
- Every number n is a divisor (or factor) of itself. If we don't want to include n in the list of divisors, we can specify the others as *proper divisors*.
- Integers are *even* or *odd* depending on whether they're divisible by 2.
- Every integer divisible by 5 is guaranteed to have a last digit of 0 or 5.
- Every integer divisible by 10 ends in a 0.
- The set of positive divisors of 6 is {1, 2, 3, 6}. The number 6 is considered *perfect* because the sum of its proper divisors, 1 + 2 + 3, is 6 itself.

Programming Challenge

2.1 Fizz-Buzz is a game that can be played by any number of players seated in a circle. Players take turns counting up from 1, but if the number they're supposed to say is divisible by 5, the player says "Fizz" instead of the number. If the number is divisible by 7, the player says "Buzz." If the number is divisible by both 5 and 7, the player says "Fizz Buzz." If a player says the wrong thing, they're out, and the last player left wins. Write a program so Scratch Cat can play Fizz-Buzz with you.

Modular Arithmetic

Even though dividing one integer by another doesn't necessarily result in another integer, *modular arithmetic* gives us a way to express any division operation using integers. The answer to a modular division problem is reported as two separate integers: the *quotient* itself, with any decimal component removed, and an extra part called the *remainder*. Symbolically, we say the integer b divided by the positive integer a gives a quotient q and a remainder r, where $0 \leq r < a$. The relationship is given by the equation $b = (q \cdot a) + r$.

Division is the process of determining a quotient and remainder given b and a. The *division algorithm* identifies the quotient and remainder. Scratch has a built-in operation to capture the remainder r, called `mod`. To find the quotient q, we do the division using the / operator and indicate that we want to keep only the integer part of the result by using the `floor` operation. Figure 2-1 gives an example.

Figure 2-1: Calculating the quotient and remainder of 45/7

Here, `floor of 45 / 7` gives us a quotient of 6, and `45 mod 7` gives us a remainder of 3. To check this is right, we can plug the results into our formula:

$$\begin{aligned} b &= (q \cdot a) + r \\ &= (6 \cdot 7) + 3 \\ &= 42 + 3 \\ &= 45 \end{aligned}$$

We say two numbers x and y are *congruent modulo n* if $x \bmod n = y \bmod n$. In this case, when x and y are divided by n, they have the same remainder r. For example, 7 and 19 are congruent mod 6, because 7 and 19 divided by 6 both yield a remainder of 1. Congruence isn't as strong as equality, in that equal numbers must be congruent, but congruent numbers need not be equal. Instead of an equal sign (=), we use the triple bar symbol (≡) for congruence, so we write $7 \equiv 19 \bmod 6$.

Here are some facts connecting modular arithmetic to divisibility:

- We can test for divisibility of b by a using Scratch by seeing if $b \bmod a$ is 0.
- Odd numbers are all congruent to 1 mod 2, and even numbers are congruent to 0 mod 2.
- Numbers that end in 0 are congruent to 0 mod 10. They're also divisible by 10.
- Numbers that end in 0 or 5 are congruent to 0 mod 5. They're also divisible by 5.
- When we represent $b = (q \cdot a) + r$ by the division algorithm, the set of all possible remainders is $\{0, 1, 2, \ldots, a - 1\}$, a set of a elements. Sometimes it's more useful to use another set of a elements where every integer is congruent to one element of the set. Since Scratch numbers elements in lists starting from 1, the set $\{1, 2, 3, \ldots, a\}$ is often a good choice.

We'll explore a simple hack that uses modular arithmetic to help check the results of a calculation in the next project.

Project #5: A Trick for Checking Your Math

Casting out nines is a trick for verifying the answer to a large addition or multiplication problem. To see how it works, first notice that every power of 10 leaves a remainder of 1 when it's divided by 9. For example:

$$\begin{aligned} 10 &= 9 + 1 \\ 100 &= 99 + 1 = (11 \cdot 9) + 1 \\ 1{,}000 &= 999 + 1 = (111 \cdot 9) + 1 \end{aligned}$$

This points to a broader rule that when you divide a number n by 9, you get the same remainder as when you divide the sum of the digits of n by 9. Take the case of 347 divided by 9. To determine the remainder, we first sum the digits: 3 + 4 + 7 = 14. At this point, we could notice that $14 = (1 \cdot 9) + 5$, giving us a remainder

of 5. Or we could do the casting out nines trick a second time to get the result in an easier way: 1 + 4 = 5. (In fact, 347 divided by 9 is 38 remainder 5.)

Casting out nines is a good way to check your work after a big addition or multiplication operation, because it's much easier to do arithmetic mod 9 (by summing a number's digits) than to keep track of multidigit sums and products. Suppose, for example, you calculate 347 + 264 and get the answer 601. We've already seen that 347 mod 9 is 5. For 264, 2 + 6 + 4 = 12 and 1 + 2 = 3, so 264 mod 9 is 3. That means (347 + 264) mod 9 should be 5 + 3 = 8. But 601 mod 9 is 6 + 0 + 1 = 7, so something is wrong. It looks like somebody forgot to carry the 1 in the original addition! When we fix the sum to be 611, casting out nines works as expected.

Even though adding up the digits of a number is pretty easy mental math, let's have Scratch Cat do the work for us. The program in Figure 2-2 uses the casting out nines technique to find any number mod 9.

Figure 2-2: A program for finding `x mod 9` by calculating digit sums

The trick is to have Scratch see the number `x` as a string of digits. The `length of` operator reports how many digits the number has, and the `letter of` operator lets us pick off one digit at a time so we can add them up. The code is nested inside a `repeat until` loop that makes it continue until the length of `x` is `1`, meaning the number has only one digit. If that single digit is in the range 0 through 8, we have our answer. The single digit could also be a 9, though, which is congruent to 0 mod 9. In that case, the last `if` statement picks 0 as the answer to report instead of 9.

The Results

Figure 2-3 shows a sample run of the program, using 601 as the input.

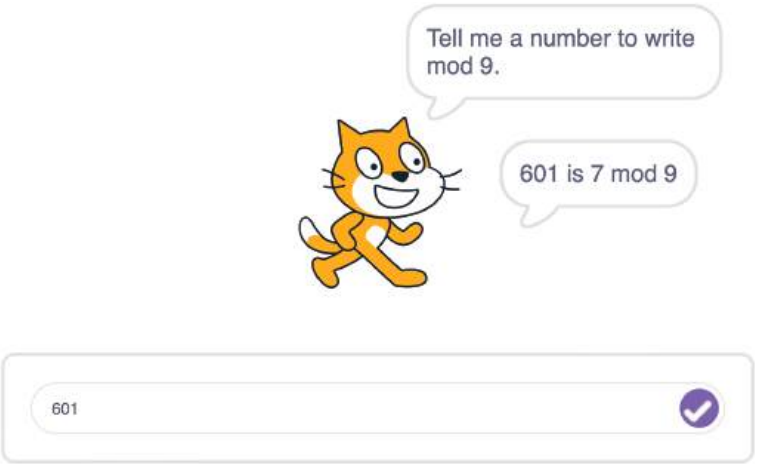


Figure 2-3: Finding 601 mod 9

The last line of the program uses `join` operations to make the output pretty, reminding us of the input number and the result of the casting out nines process.

Hacking the Code

We have the same problem here that we had in Chapter 1: Scratch is happy to run the code on input that isn't a number. The way the program is written, it even gets in trouble with what ought to be perfectly allowable inputs, like negative integers. For example, the number –3 interpreted as a string has a length of 2, and according to Scratch, the first character, the minus sign, has a numerical value of 0. So Scratch reports that the sum of the digits of –3 is 0 + 3 = 3. The trouble is that –3 mod 9 is equal to 6, not 3.

Because we'll run into problems with negative integers and non-integer inputs, before we put the code out for general use, we should make it safe by screening possible inputs to allow only the ones we want: positive integers. We can create a custom block to screen the input, as shown in Figure 2-4.

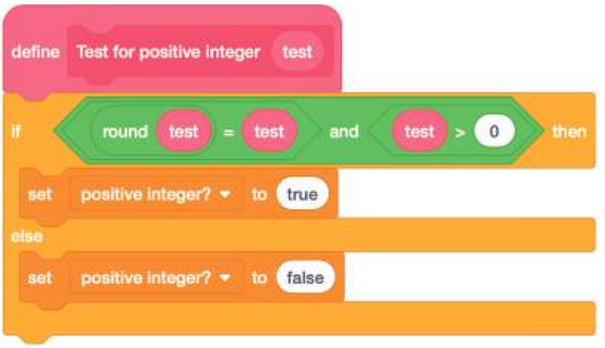


Figure 2-4: Making sure the input is a positive integer

The Boolean statement `round test = test` is a hack that lets us kill a few birds with one stone. It screens out non-numeric input (such as the word *banana*), since trying to round a non-number in Scratch produces 0 as a result. It also screens out numbers with nonzero decimal components, which will no longer be equal to themselves after rounding. Combined with `text > 0`, our `if` statement is true if the input `test` is a positive integer and false otherwise, so we can set the value of the variable `positive integer?` to `true` if the two conditions are satisfied.

NOTE *Some programming languages have special Boolean variables that can take on only the values `true` or `false`, but Scratch doesn't. Here, we simply use the words* true *and* false *instead. Some programmers prefer to use the numerical values 1 and 0 to keep track of truth values.*

Once we have a screening block, we can modify the code in Figure 2-2 to execute for only appropriate values, as shown in Figure 2-5. Paste the original program from the `repeat until` block onward into the empty slot after the `if`.

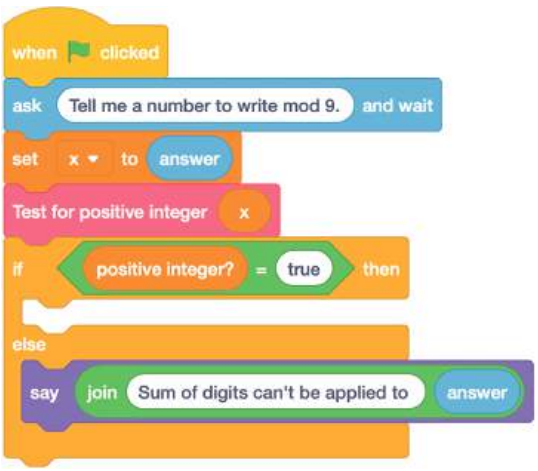


Figure 2-5: Don't let Scratch Cat make a mistake!

Of course, you don't actually have to go through the casting out nines process to perform this calculation. You can just use Scratch's `mod` block! Still, writing the program is good practice for figuring out how to solve a problem and how to analyze a number, one digit at a time. The program also generalizes to other cases, such as the one in Challenge 2.2.

Programming Challenges

2.2 Casting out nines gives a test for divisibility by 9, since if the sum of the digits is 0 or 9, the number is divisible by 9. A test for divisibility by 11 works similarly, except instead of adding all the digits, you alternately subtract and add them. For example, 1,342 is divisible by 11 because 1 – 3 + 4 – 2 = 0. Program Scratch to calculate the –/+ digit sum for a given number to see if it's divisible by 11.

2.3 Scratch has an operator that lets you pick a random number in a specified range. Write a program to pick 10 random numbers between 1 and 100. Predict how many are likely to be divisible by 9, then use Scratch to check if your prediction was right.

2.4 Sometimes when you have to enter a number into a computer form (like a credit card number or a book's ISBN code), the number includes a *check digit* to make sure you haven't made a mistake. One way to implement this is to add an extra digit at the end that's derived from the original number. For example, the extra digit could be the original number mod 9, found by casting out nines as in the program in Figure 2-2. Extend this program to give the original number with its check digit added.

2.5 When copying numbers we sometimes make *transposition* errors, where two digits are switched. For example, we might miswrite 1,467 as 1,647. Could you use the casting out nines trick to help catch this kind of mistake?

Prime Numbers

Some integers have many divisors, and some have only a few. The integer 1 is a special case, in that it's divisible only by itself. For any other number, the smallest number of divisors is two: 1 and the number itself. As mentioned at the beginning of this chapter, numbers with only two divisors are called prime numbers. Numbers with more than two divisors are called *composite numbers*.

The first few prime numbers are 2, 3, 5, 7, 11, 13, and 17. To find more, we'll turn to Scratch.

Project #6: Is It Prime?

One way to determine if a number is prime is to try out possible factors one by one, a process called *trial division*. If there aren't any other divisors between 1 and the number, then the number is prime. For the number 5, for example, we would try dividing 5 by 2, then 3, then 4. None of those numbers divide evenly into 5, so 5 is prime.

Doing trial division manually quickly gets tedious, so we'll write a program to make Scratch do it for us. Figure 2-6 shows a simple version of the code that doesn't worry about improper inputs that could cause incorrect answers (for example, strings or numbers that aren't positive integers).

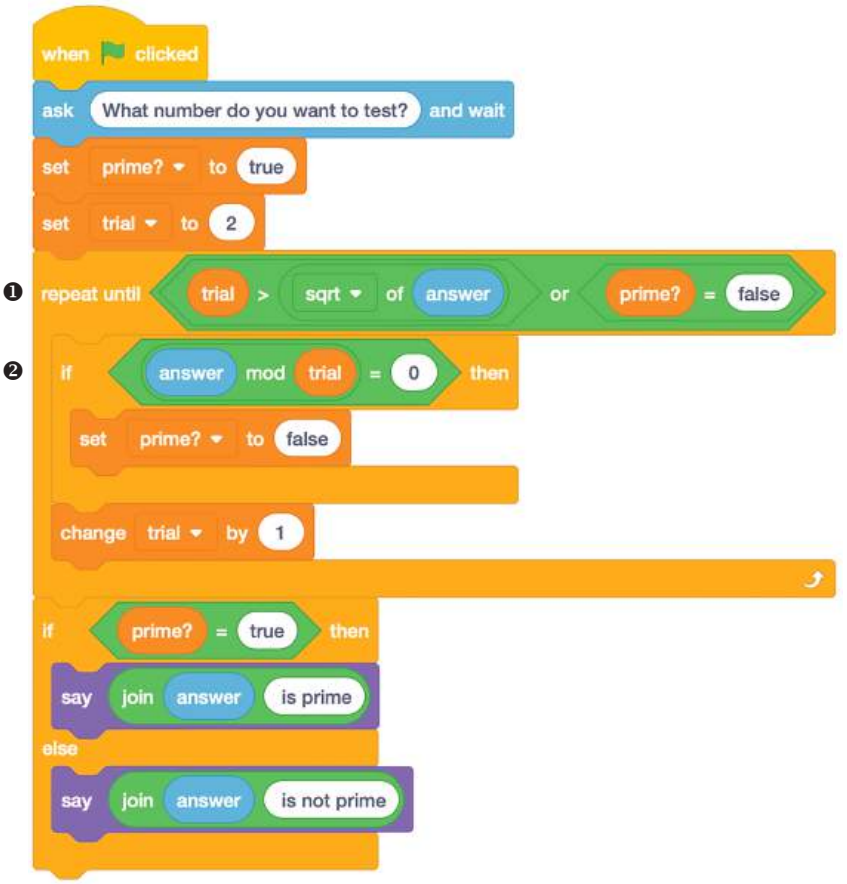


Figure 2-6: Checking for primes by trial division

The code prompts for a number to test and decides if the number is prime by working with the Boolean `prime?` variable. We perform the trial division in a `repeat until` loop ❶ by calculating `answer mod trial` ❷, where the variable `trial` is the trial divisor. If the result is `0`, we know that we have a divisor and that `answer` isn't prime, so we exit the loop. Otherwise, we add `1` to `trial` and try again. At the end, we report an answer based on whether `prime?` is `true` or `false`.

The Results

Figure 2-7 shows some sample runs of the trial division program.

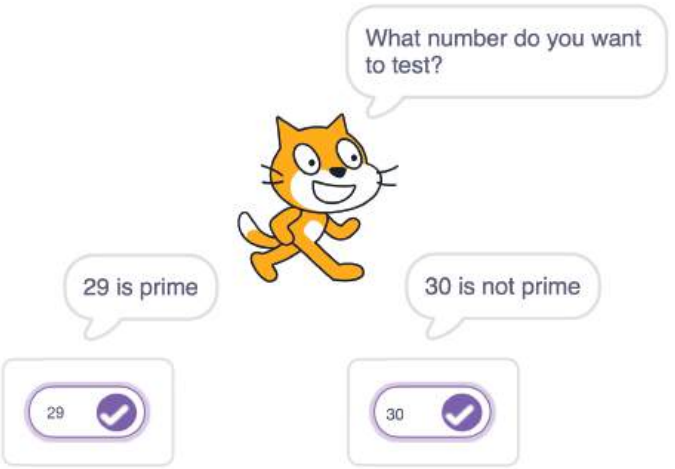


Figure 2-7: Sample runs of the trial division program

The program correctly identifies 29 as prime and 30 as not prime.

Hacking the Code

We should screen the input so Scratch is considering only positive integers. A custom block like the one we made for the casting out nines program (see Figure 2-4) would work, put into an `if` statement (as in Figure 2-5). There are a few more conditions to put into the screening code, though. First, the integer 1 is neither prime nor composite, but 1 would survive the `repeat until` loop in our trial division program and be labeled as prime. The custom block in Figure 2-8 includes an initial `if` test to disallow an input of 1.

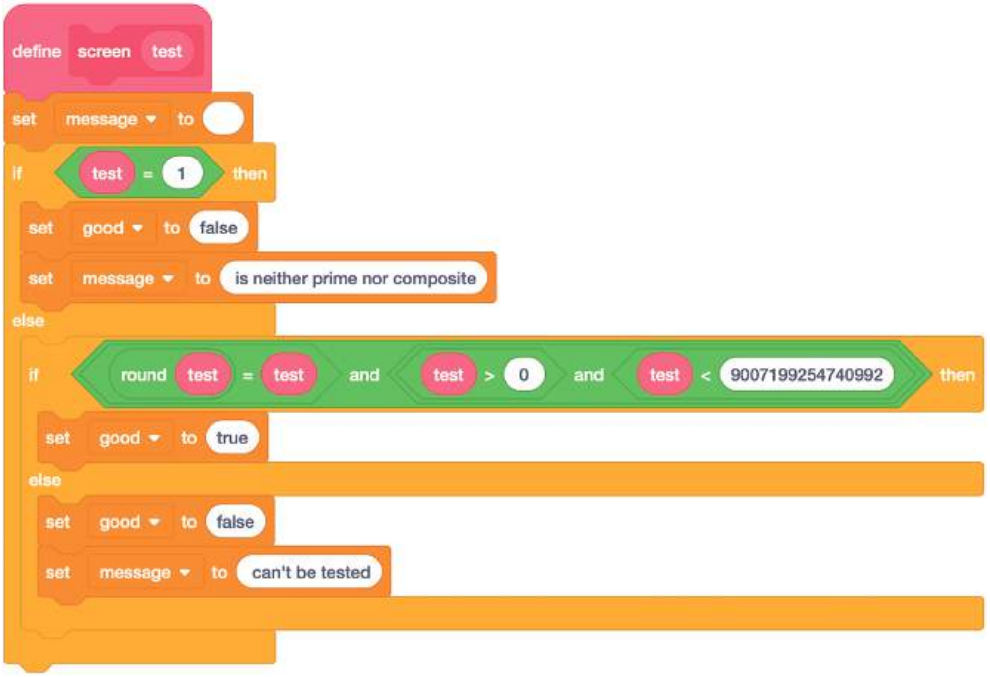


Figure 2-8: Limiting the input for the trial division program

A more subtle problem is that, as we saw in Chapter 1, integer arithmetic is exact only up to flintmax. That means the divisibility test works only for numbers up to 9,007,199,254,740,992. After that, Scratch Cat thinks every number is composite! The check code in Figure 2-8 accounts for this as well by verifying that `test` is less than flintmax. The block also returns a `message` variable giving more information for the program to report when the input can't reliably be tested.

Another consideration with this program is that trial division on large numbers potentially takes many steps—so many that even on a fast computer you might have to wait a long time to get an answer. The test in the `repeat until` loop ❶ in Figure 2-6 is a hack to speed up the process: we really have to consider only trial divisors up to the square root of the input number. This works because if a number n isn't prime, it must have a factorization $n = a \cdot b$ other than the trivial factorization $1 \cdot n$. Since $n = \sqrt{n} \cdot \sqrt{n}$, one of a or b must be bigger than $\sqrt{n}$ and one must be smaller than $\sqrt{n}$. We have to do trial division only up to $\sqrt{n}$ to find the smaller one, if it exists.

This hack provides a huge savings! We can test numbers up to 1,000,000 with no more than $\sqrt{1{,}000{,}000} = 1{,}000$ trial divisions. To speed up the code even further, once we've checked on divisibility by 2, we could test only for divisibility by odd numbers. This is because if a number n is divisible by any even number, it will also be divisible by 2.

All these improvements allow for shorter runtimes, but they also make for a longer, more complicated program. Whether the trade-off is worth it will depend

on who will be using your work, and for what. Improvements that make the program easier to use are usually worth it. Improvements that speed up runtime have to be dramatic to be noticeable, but they may be worthwhile if users will be looking for quick results.

Project #7: The Sieve of Eratosthenes

Trial division isn't the only way to find prime numbers. In this project, we'll explore a different technique: looking at a list of all numbers up to some limit and throwing away the numbers that are composite. This approach sifts, or *sieves*, out the primes and is called the *sieve of Eratosthenes* after the Greek mathematician who first used it. Scratch Cat uses sieving in Figure 2-9, where the numbers 1 through 120 have been arranged in a grid.

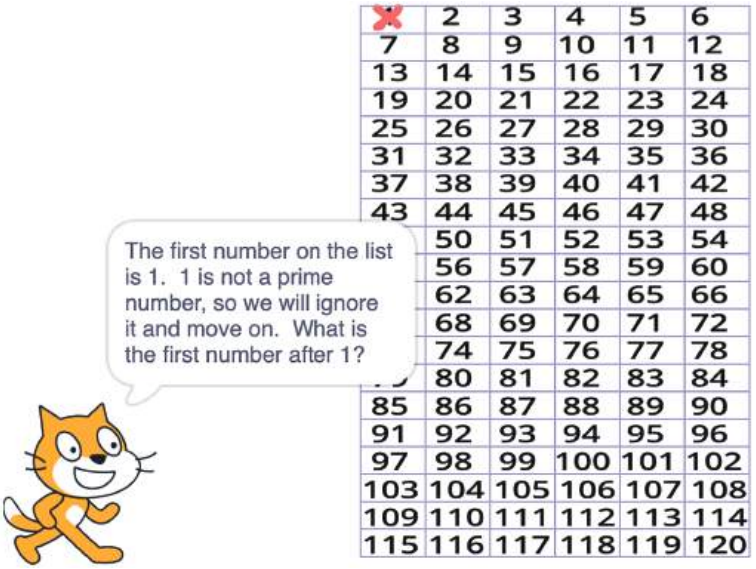


Figure 2-9: Sieving out the primes by throwing away non-primes

First, we cross out 1, which is neither prime nor composite, in red. Then, we cross out all multiples of 2 in green, as shown on the left side of Figure 2-10, and see what's left. We continue by identifying the next few primes after 2 (3, 5, and 7) and crossing out any multiples of them, as shown on the right side of Figure 2-10.

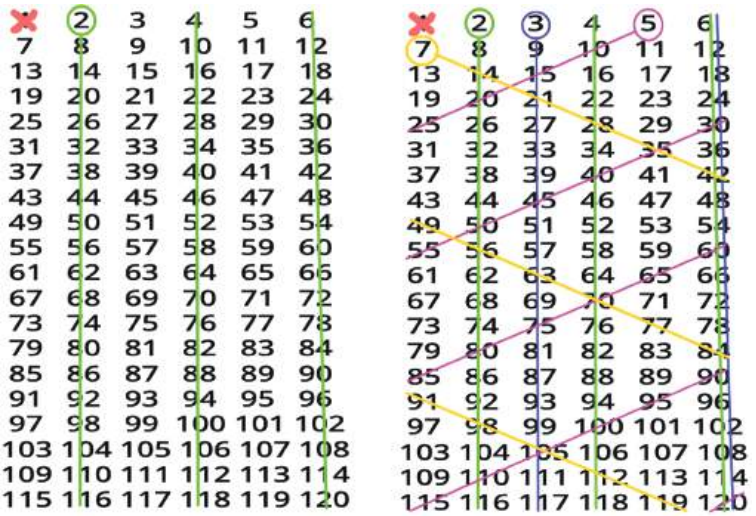


Figure 2-10: Eliminating all the even numbers after 2 (left) and all multiples of 3, 5, and 7 (right)

Notice that multiples of 2 and 3 are crossed out with vertical lines down the columns of the grid. This works because the grid is set up to be 6 numbers wide, and 6 is divisible by both 2 and 3. Multiples of 5, crossed out in pink, step backward on the diagonal. This is because to get from one multiple of 5 to the next multiple we add 5, which is 6 – 1. So to find the next multiple of 5, we go down one row for the 6 and back one column for the –1. Similarly, to find multiples of 7, crossed out in yellow, we go down one row and step one column to the right (because 7 = 6 + 1), giving us lines along the other diagonal.

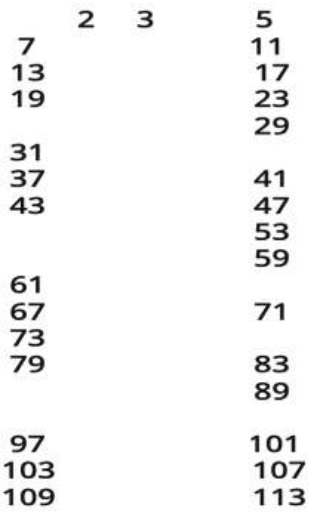


Figure 2-11: All the primes up to 120, after sieving

Here's the payoff of sieving: if a number n is composite and has a factorization $n = a \cdot b$ where $1 < a \leq b < n$, then $a \leq \sqrt{n}$. In our example, $n = 120$, so any composite number in the grid must have a prime factor less than $\sqrt{120}$, or approximately 10.95. Once we've sieved up to 7, the next number that hasn't already been crossed out is 11, which is greater than $\sqrt{120}$, so 7 is as far as we need to sieve. Every number that remains, meaning it hasn't been crossed off as a multiple of 2, 3, 5, or 7, must be a prime number (see Figure 2-11).

This is the second time the square root hack has been useful. First, it made the trial division program in Figure 2-6 run faster. Now, it's telling us when to stop sieving, allowing us (in this example) to find all primes less than 120 just by sieving up to 7.

We could use the same technique to sieve up to a much higher bound. All we have to do

is get rid of all the multiples of each prime as they're discovered, up to the square root of the bound. That's what we do in the Scratch program in Figure 2-12.

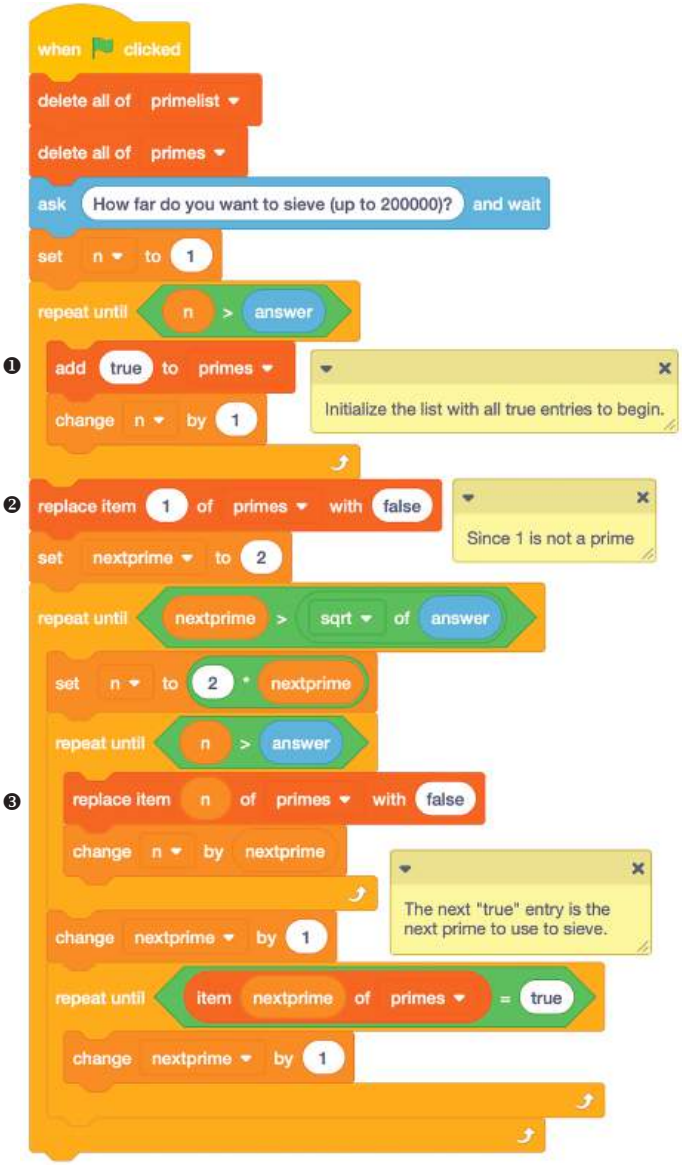


Figure 2-12: The sieve program

We start by asking how far to go, then seeding the list `primes` with that many entries ❶. (Since we're using a list, our upper bound is limited by the maximum list size that Scratch supports, which is 200,000.) Scratch indexes lists starting with 1, so the list entry at index *n* will keep track of whether *n* is a prime. Initially we set each entry to `true`, but we'll change non-prime entries to `false` as we sieve.

First, we handle the special case of 1, which is neither prime nor composite ❷. Then, we look for the next number not crossed out by sieving so far. We leave that number as `true` but set all multiples of that number to `false` ❸. We repeat this process until the next number not crossed out is greater than the square root of the limit.

Once we have a complete list, we can access it and answer questions about the prime numbers we've found. Figure 2-13 has a little piece of code to count how many primes there are up to the sieve limit.

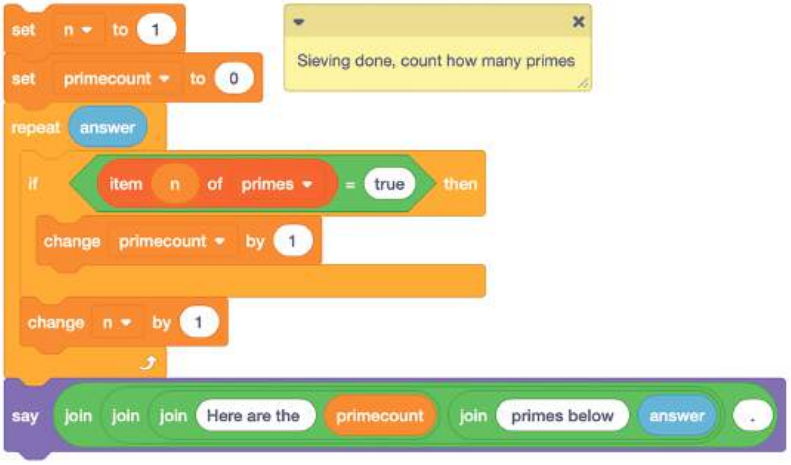


Figure 2-13: Counting primes with the sieve program

Here, we step through the list we've built and count how many `true` entries there are, incrementing the variable `primecount` each time. Figure 2-14 shows another extra piece of code that lists the primes we've found.

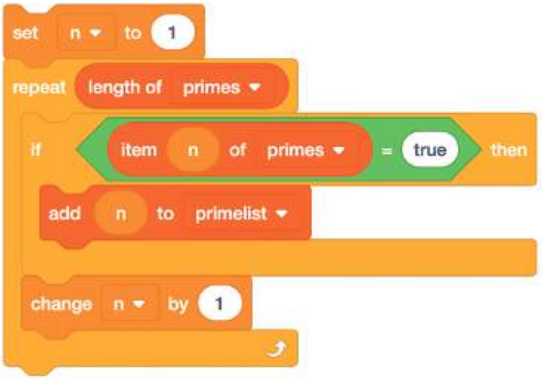


Figure 2-14: Listing primes with the sieve program

This block finds the `true` items in the list and stores their corresponding index numbers in a separate list.

Hacking the Code

Sometimes it's useful to have the data that Scratch generates as a separate file so you can import it into a text editor or a spreadsheet. Fortunately, Scratch gives us the option to import and export lists by right-clicking the list in the graphics window (see Figure 2-15). This way, you can take your sieved list of primes out of Scratch to play with it further.

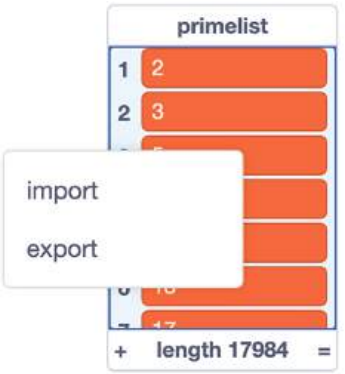


Figure 2-15: Saving the list to work on later

Text editors, word processors, and spreadsheet programs are happy to work with the text output from Scratch. Try importing your data into a spreadsheet program such as Excel, Numbers, or Open Office. If you want several entries per row, make sure you have Scratch insert commas separating the entries in your text file (using the `join` block), and then use the *CSV* format, short for *comma separated values*, to import it. The default carriage returns in the file that Scratch produces will list the entries in separate rows in the spreadsheet.

Programming Challenges

2.6 Use the sieve program to find how many primes there are between 1 and 10, 100, 1,000, 10,000, and 100,000. Keep track of the ratio between the number of primes and the size of the list, and display your results in a table. How does the relative number of primes appear to be changing as the upper bound increases?

2.7 Write a block to scan the list of integers that the sieve program produces, looking for long sequences of consecutive composite numbers. What's the longest sequence you can find?

(continued)

2.8 *Twin primes* are pairs of primes that differ by exactly 2; for example, 3 and 5 or 11 and 13. Write a block to scan the sieve program's output and count how many pairs of twin primes there are up to the sieving limit.

2.9 Rewrite the sieve program in Figure 2-12 so it displays the results in a table six entries wide, like the table in Figure 2-9. Use the language of congruences to explain why the only prime numbers that appear after the first row of the table are in columns 1 and 5.

Nothing Common About Common Divisors

Given two integers *a* and *b*, the *set of common divisors* refers to all the integers that evenly divide both *a* and *b*. There will always be at least one common divisor, the number 1, since 1 is a factor of all integers. But larger common divisors might exist as well. Of particular interest is the *greatest common divisor (GCD)*, the largest number that evenly divides *a* and *b*. If this largest common divisor is *d*, we write GCD(*a*, *b*) = *d*.

As with identifying primes, there are several methods for finding the GCD of two numbers, with varying degrees of efficiency. We'll explore two such techniques in the next two projects.

Project #8: Greatest Common Divisors the Slow Way

Here's one way to find the greatest common divisor between two integers *a* and *b*. Starting from 1, try dividing *a* and *b* by every number. If it divides evenly into both of them, you've found a common divisor. Stop once you reach *a* or *b*, whichever comes first. The highest common divisor you've found is the GCD. The program in Figure 2-16 uses this approach.

We use a custom block to identify the minimum of the two input values, `a` and `b`. Then we count up from `1` to the minimum, checking if the `mod` of both `a` and `b` is `0`. If it is, we store the current divisor in the variable `gcd`, which holds our answer when the program finishes running.

This technique of testing every number as a possible common divisor is known as a *brute-force* approach. It's like trying to guess someone's computer password by testing out every possible sequence of letters and numbers. For our GCD program, brute force is fast enough for smaller values of `a` and `b`, say up to 1 million, but it's noticeably slower for larger numbers. As the numbers being screened get closer to flintmax, it becomes especially annoying to wait. Luckily, there's a better way.

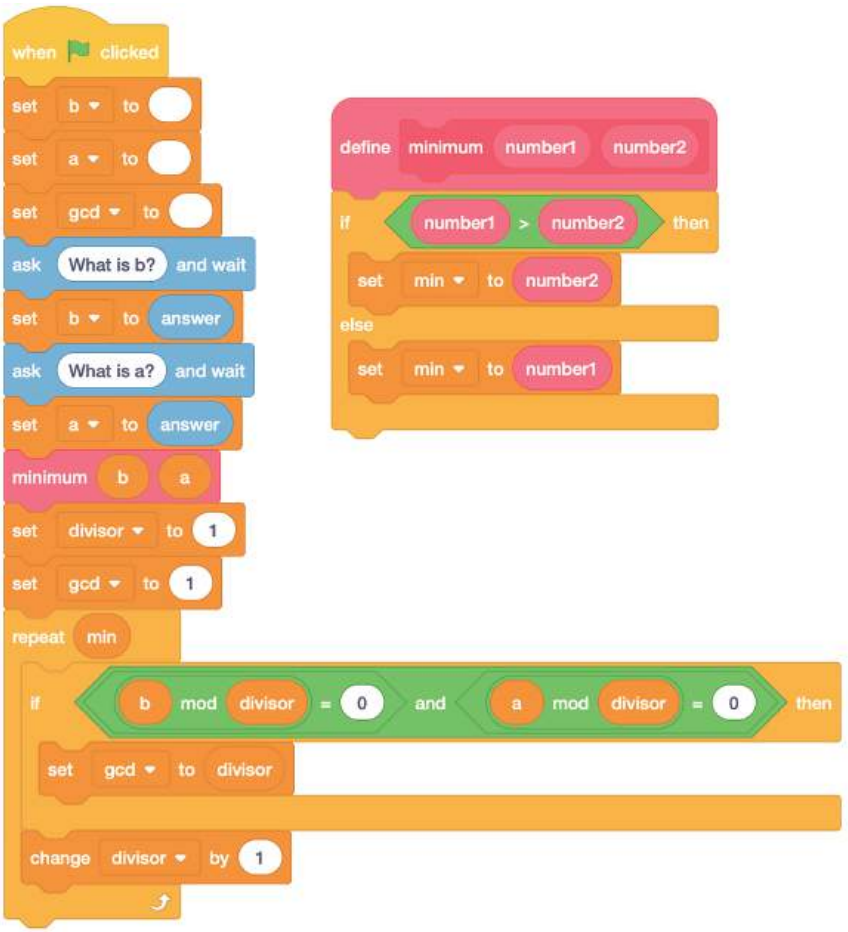


Figure 2-16: Finding the GCD the slow way

Project #9: Greatest Common Divisors the Fast Way

The Greek mathematician Euclid described a more efficient method for calculating the greatest common divisor of two numbers in his textbook *The Elements*, which was written around 300 BCE. *The Elements* covers topics in several different areas of mathematics, focusing on geometry and number theory. The book was so influential that Euclid's organization of the material was used to teach mathematics for centuries, and it continues to be used today.

Euclid's approach to calculating greatest common divisors is based on the observation that for two positive integers a and b where $a < b$, any common divisor of a and b is also a divisor of $b - a$. For example, say $a = 330$ and $b = 876$. A common divisor of 330 and 876 is 6, and 6 is also a divisor of $876 - 330 = 546$.

By extension, if we divide the larger of the two numbers, b, by the smaller, a, and keep track of the division with a quotient and remainder, $b = q \cdot a + r$, then any

common divisor of b and a is also a divisor of a and r. Then we can repeat the process with a and r, and so on until there's a last remainder of 0. At this point, the next-to-last remainder is the greatest common divisor of a and b. The sequence of divisions looks like this:

$$\begin{aligned} b &= q_1 \cdot a + r_1 \\ a &= q_2 \cdot r_1 + r_2 \\ r_1 &= q_3 \cdot r_2 + r_3 \\ &\vdots \\ r_{k-2} &= q_k \cdot r_{k-1} + r_k \\ r_{k-1} &= q_{k+1} \cdot r_k + 0 \end{aligned}$$

The remainders decrease, so $a > r_1 > r_2 > \ldots > r_k$, with $r_k = \text{GCD}(b, a)$ and $r_{k+1} = 0$.

Here are the steps to calculate that 6 is the greatest common divisor of $b = 876$ and $a = 330$, interpreted both with the division algorithm and with modular arithmetic. Notice how the values shift positions from right to left as we move from one line to the next:

$$\begin{aligned} 876 &= 2 \cdot 330 + 216 & \quad 876 \bmod 330 &= 216 \\ 330 &= 1 \cdot 216 + 114 & \quad 330 \bmod 216 &= 114 \\ 216 &= 1 \cdot 114 + 102 & \quad 216 \bmod 114 &= 102 \\ 114 &= 1 \cdot 102 + 12 & \quad 114 \bmod 102 &= 12 \\ 102 &= 8 \cdot 12 + 6 & \quad 102 \bmod 12 &= 6 \\ 12 &= 2 \cdot 6 + 0 & \quad 12 \bmod 6 &= 0 \end{aligned}$$

The Scratch program in Figure 2-17 implements Euclid's algorithm.

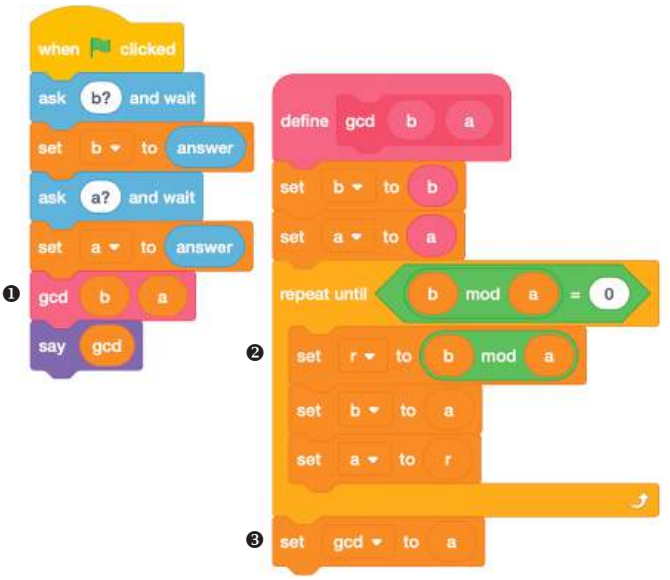


Figure 2-17: Finding the GCD with Euclid's algorithm

The program is organized so all the work of the repeated division occurs in the custom `gcd` block ❶. The block's definition is surprisingly short compared to our brute-force GCD program (Figure 2-16). Inside a `repeat until` loop, we keep taking `b mod a` ❷ and shuffling the values of `a` and `r` back into `b` and `a` until we finally get down to a remainder of 0. That's where the loop stops, and the last value of `a` can be reported as the GCD ❸.

The Results

Figure 2-18 shows a sample run of the GCD program with two very large numbers as inputs.

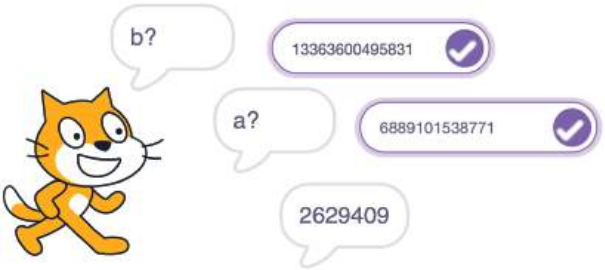


Figure 2-18: A calculation with Euclid's algorithm

Unlike our brute-force approach, the code works very quickly, even for numbers close to flintmax.

Hacking the Code

So far, the language we've used to talk about how efficient an algorithm is has been pretty general. We talk about a program running quickly or slowly, but it would be good to know just how quickly or slowly that turns out to be on your computer. It would also be useful to see how the program's performance changes as we go from working with numbers in the tens or hundreds to numbers in the thousands or millions.

Scratch has a built-in timer that measures elapsed time in seconds from the moment a program starts executing. It's accessible via the `timer` block in the Sensing section of the block menu. We can take any program and wrap it in a few lines of code to time how long an algorithm takes to run, as shown in Figure 2-19.

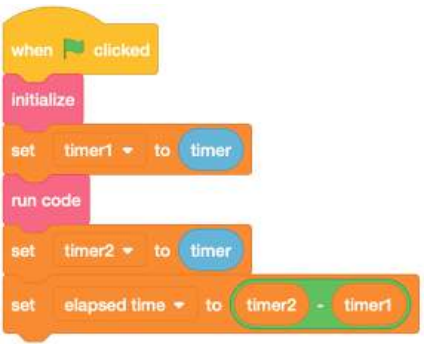


Figure 2-19: Timing how fast a program runs

Here, the `initialize` block would contain any setup code that you don't want to time, such as prompting the user for input, while the `run code` block would contain the code for the algorithm you want to time. We record the value of `timer` before and after executing `run code`, then take the difference between the two times to see how long the execution took.

Figure 2-20 shows the result of wrapping the trial division prime testing program from Figure 2-6 in the timer code, including the value of `elapsed time` when the program finishes. For a prime close to flintmax, it takes my computer a little over a minute to report.

Figure 2-20: Testing a big prime number

For many programs with small test values, the elapsed time will show as `0`, since the algorithm takes only a fraction of a second to run. The reported time might also vary across runs because your computer is doing other things in the background, which limits the amount of resources Scratch has available to do its job. To get an accurate time, run the program lots of times in a row and keep track of the cumulative runtime, then divide by the number of times you ran the program to find the average time for each run.

Programming Challenges

2.10 Use timing loops to compare the runtimes for the two GCD calculating programs (the brute-force version in Figure 2-16 and the Euclidean version in Figure 2-17).

2.11 Program a counter to count how many steps Euclid's algorithm takes. Experiment to see what numbers make the algorithm take the highest number of steps to run.

Conclusion

Computations involving divisibility are much easier and faster to do with computer assistance. If I had to work out if a number was prime by doing trial division by hand, I would probably give up after a few dozen calculations. Even if I were punching possible divisors into a calculator, I would get bored pretty quickly and probably start making mistakes ("trial and error" is mostly error!). But Scratch Cat is eager to help out for as long as I want. Scratch is a telescope that lets us look deeper into the universe of numbers than we could ever do ourselves. All we have to do is ask.