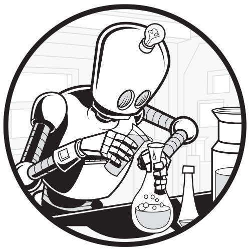
4

SOLVING MATHEMATICAL PROBLEMS WITH CODE



After our thorough overview of the fundamentals of the Kotlin programming language and the JavaFX graphics tools adapted for use in Kotlin, we're now prepared to tackle a series of math problems in the form of mini projects. The projects will grow in complexity over the chapter, but they require only high school math skills. Our journey will take us from the ancient civilizations of Babylon, Greece, and Egypt to the modern-day world of cryptography.

The main goal of these projects is to enhance your Kotlin programming skills. We'll discuss the context and mathematics behind each problem in detail, but the heart of each project will be developing an appropriate algorithm or problem-solving strategy and then implementing it in well-organized code. In doing so, you'll gain a deeper understanding of both programming and math, preparing you to solve the more sophisticated problems introduced later in this book.

Project 9: Find the Square Root with the Babylonian Algorithm

We have several methods of finding the square root of a number. In this project, I'll focus on the Babylonian square root algorithm, one of the most widely used methods today.

The Babylonian square root algorithm dates back to around 1800 BCE. It's believed the Babylonians used it for practical purposes, such as land surveying. The algorithm was later refined by the Greeks, who used it to calculate square roots to a high degree of accuracy. The Greek mathematician Heron of Alexandria described the algorithm in his work *Metrica*, written in the first century CE.

Despite its ancient origins, the Babylonian square root algorithm remains a valuable tool for understanding the history of mathematics and the development of numerical methods. The algorithm's enduring usefulness is a testament to the ingenuity of ancient mathematicians and the power of mathematical techniques.

THE ORIGIN OF THE BABYLONIAN ALGORITHM

The Yale Babylonian Collection, part of the Yale University Library in New Haven, Connecticut, houses a Babylonian clay tablet called YBC 7289, which contains the earliest known description of the Babylonian square root algorithm. This tablet has been a significant object of study for scholars of ancient mathematics and the history of science since it was acquired in the early 20th century. The Yale Babylonian Collection also includes a number of other important artifacts, including cuneiform tablets, cylinder seals, and other objects that provide insight into the culture, society, and technology of ancient Mesopotamia.

We can use the Babylonian algorithm to approximate the square root of a positive number in a few simple and iterative steps. The algorithm starts with an initial guess and then refines that guess until it's close enough to the actual square root. Here's how the algorithm works:

1. Start with an initial estimate, `guess`, for the square root of a positive number, `N`. This is customarily set to `N / 2`.
2. Check to see if the absolute value of `(guess * guess - N)` is less than the tolerance value. If yes, then terminate the loop and return the estimated square root.
3. Otherwise, update the guess using the formula `guess = (guess + N / guess) / 2.0`.
4. Repeat steps 2 and 3 until the stopping condition is met.

The Code

The Babylonian algorithm is simple enough that we can write a concise code segment to find a square root. However, it's a good practice to create a separate function for a process like this and then call that function from `main()`. This makes the code more reusable and easier to read.

Here's an example Kotlin function for calculating the square root of a positive number using the Babylonian algorithm:

```
fun babylonianSquareRoot(num: Double): Double {
    val TOL = 0.000001
    var iter = 1
    var guess = num / 2.0

    while(Math.abs(guess * guess - num) > TOL) {
        println("iter: $iter  guess=$guess")
        guess = (guess + num / guess) / 2.0
        iter ++
    }
    return guess
}
```

This `babylonianSquareRoot()` function takes a positive double-precision number `num` as its single argument. It sets the tolerance value `TOL` to `0.000001`, initializes a variable `iter` to `1` to track the number of iterations, and makes a starting `guess` of `num / 2.0`. The function then follows the Babylonian algorithm I described, using a `while` loop to refine the value of `guess` until the result is within the tolerance. To help visualize the convergence process, the intermediate values of `iter` and `guess` are printed at each iteration.

To use this function, call it from the `main()` function and provide the value of the number you want to find the square root of, like so:

```
fun main() {
    println("\n*** Finding Square Root Using Babylonian Algorithm ***\n")
    println("Enter a number (>=1) to find its square root:")
    val num = readln().toDouble()
    println("You have entered: $num\n")
    val squareRoot = babylonianSquareRoot(num)
    println("\nThe estimated square root of $num is: $squareRoot\n")
}
```

In this case, the user is asked to enter the value of a positive number, which is read as a string and converted into a number of type `Double` before its square root is estimated. We're assuming that the user will enter a valid number, which is greater than or equal to 1. If the user enters characters that cannot be converted into a number of type `Double`, the program is terminated with an error message. Also, if the user enters a valid negative number, the algorithm will not converge to a real solution.

In Chapter 1, you learned how to handle such errors or exceptions. Feel free to experiment with this code and to make it error-proof by using a `try...catch` block.

The Result

Without further ado, let's try running the algorithm! If `N` is set to `25`, the code should output the following:

```
*** Finding Square Root Using Babylonian Algorithm ***

Enter a number (>=1) to find its square root:
25
You have entered: 25.0

iter: 1  guess=12.5
iter: 2  guess=7.25
iter: 3  guess=5.349137931034482
iter: 4  guess=5.011394106532552
iter: 5  guess=5.000012953048684

The estimated square root of 25.0 is: 5.000000000016778
```

Of course, the exact value of the square root of 25 is 5. The Babylonian algorithm, like any other numerical algorithm, provides only an approximation. The accuracy of this approximation is determined by the value of tolerance (`TOL`), which can be adjusted to make the approximation more or less precise.

Keep in mind that a more accurate square root approximation will take longer to compute since the algorithm needs to go through more iterations. This sort of trade-off between accuracy and computational time is common.

Project 10: Create Pythagorean Triples with Euclid's Formula

Pythagoras was a Greek philosopher and mathematician who lived in the sixth century BCE. He believed in the idea of a harmonious universe and saw numbers, mathematics, and geometry as key elements in revealing the universe's mysteries. He's best known for the Pythagorean theorem, which states that in a right-angled triangle, the square of the length of the hypotenuse is equal to the sum of the squares of the other two sides (see Figure 4-1). Perhaps you've seen this theorem summarized as $a^2 + b^2 = c^2$.

Pythagorean triples are sets of three positive integers (a, b, c) that satisfy the Pythagorean theorem. A familiar example is (3, 4, 5): $3^2 + 4^2$ equals $9 + 16$, which equals 25, or 5^2. Pythagorean triples are used in many areas of mathematics, science, and engineering, including geometry, number theory, cryptography, physics, and computer graphics. Throughout history, mathematicians have come up with different ways of generating Pythagorean triples. In this project, we'll check out one of the earliest methods, Euclid's formula, and use it to create Pythagorean triples. Here are the steps involved:

1. Choose an arbitrary positive integer k.
2. Choose a pair of positive integers m and n, such that $m > n > 0$.
3. Calculate $a = k(m^2 - n^2)$, $b = 2kmn$, and $c = k(m^2 + n^2)$.
4. The values a, b, and c form a Pythagorean triple (a, b, c).

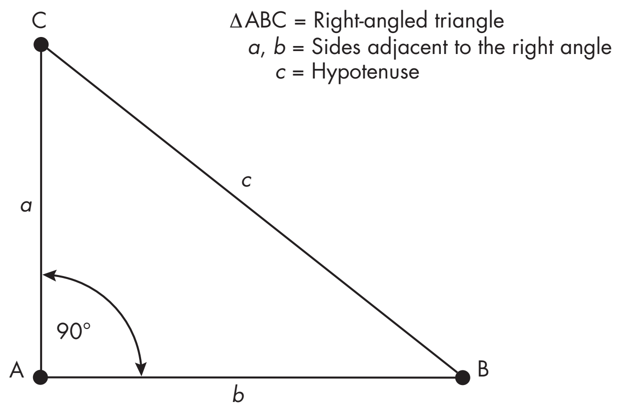

ΔABC = Right-angled triangle
a, b = Sides adjacent to the right angle
c = Hypotenuse
C
c
a
90°
A
B
b

Figure 4-1: The Pythagorean theorem

A Pythagorean triple is considered *primitive* if its members are all coprime, meaning they share no common factors other than 1. For example, (3, 4, 5) and (6, 8, 10) are both Pythagorean triples, but only (3, 4, 5) is primitive, since 6, 8, and 10 have a common factor of 2. Euclid's formula will generate a primitive Pythagorean triple if and only if the two integers *m* and *n* are coprime, and one of them is even. If both *m* and *n* are odd, then the values of *a*, *b*, and *c* will all be even, and the triple won't be primitive. However, as long as *m* and *n* are coprime, dividing the values of *a*, *b*, and *c* by 2 will result in a primitive Pythagorean triple.

The Code

Here's a Kotlin function that generates a Pythagorean triple using Euclid's formula:

```
fun generatePythagoreanTriple(m: Int, n: Int):
    Triple<Int, Int, Int> {
    val a = m * m - n * n
    val b = 2 * m * n
    val c = m * m + n * n
    return Triple(a, b, c)
}
```

The function takes in two integers *m* and *n*, then uses them to calculate and return *a*, *b*, and *c*. By not explicitly including a value for *k* here, we're implicitly assuming *k* = 1.

We can call this function repeatedly from the `main()` function, using a `for` loop to generate Pythagorean triples for an arbitrary number of pairs of successive integers, as shown in the following code:

```
fun main() {
    var m = 2              // value of m
    var n = 1              // value of n
    val numTriples = 10    // number of triples
    println("\n*** Pythagorean Triples Using Euclid's Formula ***\n")
    println("Number of Pythagorean triples: $numTriples\n")

    // Generate the first "numTriples" triples.
    for (i in 1..numTriples) {
        val pythagoreanTriple =
            generatePythagoreanTriple(m, n)
        print("i=${"%2d".format(i)}    " +
              "m=${"%2d".format(m)}    n=${"%2d".format(n)}  ")
        println("Pythagorean triple: $pythagoreanTriple")
        n++
        m++
    }
}
```

Notice that the first triple is generated using an `m` of `2` and an `n` of `1`. These are the smallest possible values of `m` and `n`. (Recall the stipulation in Euclid's formula that *m* and *n* must be positive integers such that $m > n > 0$.) These values are passed on as arguments to the `generatePythagoreanTriple()` function, which returns the elements of the Pythagorean triple as a `Triple` object in Kotlin. Successive inputs are generated by incrementing both `m` and `n` inside the `for` loop that repeats `numTriples` times. Since `m` and `n` are consecutive, one of them will always be even, and they can't possibly share any factors, so every triple will be primitive.

The Result

If you don't change any of the program parameters, the program will produce the following output that shows the first 10 Pythagorean triples:

```
*** Pythagorean Triples Using Euclid's Formula ***

Number of Pythagorean triples: 10

i= 1    m= 2    n= 1  Pythagorean triple: (3, 4, 5)
i= 2    m= 3    n= 2  Pythagorean triple: (5, 12, 13)
i= 3    m= 4    n= 3  Pythagorean triple: (7, 24, 25)
i= 4    m= 5    n= 4  Pythagorean triple: (9, 40, 41)
i= 5    m= 6    n= 5  Pythagorean triple: (11, 60, 61)
i= 6    m= 7    n= 6  Pythagorean triple: (13, 84, 85)
i= 7    m= 8    n= 7  Pythagorean triple: (15, 112, 113)
i= 8    m= 9    n= 8  Pythagorean triple: (17, 144, 145)
i= 9    m=10    n= 9  Pythagorean triple: (19, 180, 181)
i=10    m=11    n=10  Pythagorean triple: (21, 220, 221)
```

In this example, the starting values of `m` and `n` were hardcoded in the `main()` function. You might modify the code to allow the user to input values of `m` and `n` (ensuring $m > n > 0$). That would allow the user to generate a wider range of Pythagorean triples based on their requirements.

Project 11: Identify Prime Numbers with the Sieve of Eratosthenes

Eratosthenes was an ancient Greek scholar who lived in the third century BCE. He was an accomplished mathematician, astronomer, geographer, and poet. In this project, we'll explore one of Eratosthenes's many mathematical discoveries: the *sieve of Eratosthenes*, an intuitive algorithm for identifying all the prime numbers up to a given limit. (We'll explore another of his ingenious discoveries in the next project.) It's remarkable to think that Eratosthenes conceived this strategy more than two millennia ago, during a time when few individuals could read or write, let alone think about algorithms and solve abstract mathematical problems.

WHAT IS A PRIME NUMBER?

A *prime number* is a positive integer greater than 1 that has no positive integer divisors other than 1 and itself. In other words, the number can be divided evenly only by 1 and itself. For example, 5 is a prime number because it can be divided evenly only by 1 and 5. It has no other positive integer divisors. On the other hand, 6 isn't a prime number, because it can be divided evenly by 1, 2, 3, and 6. Instead, we call it a *composite number*. The first several prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, and so on. Prime numbers have many interesting properties and play a central role in number theory and cryptography, among other fields.

The Strategy

To sieve for primes like Eratosthenes, start by creating a list of all integers from 2 up to some limit. Then, starting from 2, iteratively mark off all multiples of each prime number as composite. The unmarked numbers that remain at the end of the process are all prime numbers. Here are the steps to implement this algorithm:

1. Create a list of consecutive integers from 2 through the given limit.
2. Starting with 2 (the first prime number), mark all its multiples as composite.
3. Find the next number in the list that isn't marked as composite. This will be the next prime number.
4. Mark all multiples of the prime number found in step 3 as composite.
5. Repeat steps 3 and 4 until the square of the next prime number exceeds the given limit.
6. The unmarked numbers in the list are all prime numbers.

We can optimize the sieve algorithm by marking multiples of each prime number, starting from its square. For example, when marking multiples of 3, we can start at $3^2 = 9$, since all multiples of 3 less than 9 will already have been marked as composite. In this case, 6 will have been marked while going through the multiples of 2. Similarly, when we get to multiples of 5, we can skip 10, 15, and 20 as being multiples of either 2 or 3 and start marking off composites from 25.

The Code

As before, we'll start by writing a function to implement the algorithm and then use the `main()` function to call this function and list the prime numbers. Here's the Kotlin code for the `sieveOfEratosthenes()` function:

```
fun sieveOfEratosthenes(n: Int): List<Int> {
    // Create a Boolean array with all values set to true.
    val primes = BooleanArray(n + 1) { true }
    // Create a mutable list of integers to save prime numbers.
    val primeNumbers = mutableListOf<Int>()

    // Set 0 and 1 to not be prime.
    primes[0] = false
    primes[1] = false

    // Iterate over all numbers until i^2 > N.
    var i = 2
  ❶ while (i*i <= n) {
        // If i is prime, mark all multiples of i as not prime.
      ❷ if (primes[i]) {
          ❸ for (j in i * i..n step i) {
                primes[j] = false
            }
        }
        i++
    }

    // Collect all prime numbers into a list and return it.
  ❹ for ((index, value) in primes.withIndex())
        if (value) primeNumbers.add(index)

  ❺ return primeNumbers
}
```

The `sieveOfEratosthenes()` function takes an integer `n` as input and returns a list of prime numbers up to `n`. For that, the function creates a Boolean array `primes` with a length of `n + 1` and initializes each element's value to `true`. Over the course of the function, we'll change elements to `false` if their indices aren't prime. The function also creates a mutable list `primeNumbers` of type `Int` to save the prime numbers.

To begin, we set the first two values of the `primes` array to `false` because 0 and 1 aren't prime. We then iterate over the numbers from `2` to the square root of `n` (we do this by ensuring `i*i <= n`) ❶. For each number `i` in this range, if `i` is marked as prime (that is, `true`) in the `primes` array ❷, the function marks all multiples of `i` in the `primes` array as composite (`false`). To reach all the multiples of `i`, we use a `for` loop with a step size of `i` ❸.

To collect the prime numbers, we use a `for` loop ❹ to go over all `primes` elements and add the corresponding index to `primeNumbers` when the value of the element is `true`. Finally, we return the prime numbers to `main()` as a list of integers for postprocessing ❺.

Now that our sieving function is good to go, we can use the `main()` function to retrieve a list of prime numbers up to `n` and print it out. We'll also create a `printPrimes()` helper function to manage the printing.

```
fun main() {
    println("\n*** Find All Prime Numbers Up to 'n' ***\n")
    println("Enter a number > 2 to generate the list of primes:")
    val num = readln().toInt()
    println("You have entered: $num")

    val primeNumbers = sieveOfEratosthenes(num)
    println("\nThe prime numbers <= $num are:")
    printPrimes(primeNumbers)
}

fun printPrimes(primeNumbers: List<Int>) {
    for (i in primeNumbers.indices) {
        if (i != 0 && i % 6 == 0) println()
        print("${"%8d".format(primeNumbers[i])} ")
    }
}
```

The `main()` function is similar to the one we used for the Babylonian square root algorithm in Project 9. It takes a user input for the limit `num`, uses it to create a list of prime numbers with the `sieveOfEratosthenes()` function, and then calls `printPrimes()` to print the list. To make the output look nice, `printPrimes()` organizes the numbers into rows of six and uses string formatting to create neatly aligned columns.

The Result

Here's a look at the program output up to an arbitrary limit `num` of `251`:

```
*** Find All Prime Numbers Up to 'n' ***

Enter a number > 2 to generate the list of primes:
251
You have entered: 251
```

```
The prime numbers <= 251 are:
       2         3         5         7        11        13
      17        19        23        29        31        37
      41        43        47        53        59        61
      67        71        73        79        83        89
      97       101       103       107       109       113
     127       131       137       139       149       151
     157       163       167       173       179       181
     191       193       197       199       211       223
     227       229       233       239       241       251
```

Other methods for generating prime numbers include the sieve of Sundaram, the sieve of Atkin, and trial division. I encourage you to do some online research and experiment with these methods to enhance your Kotlin coding skills and gain additional insight into prime number generation.

Project 12: Calculate Earth's Circumference the Ancient Way

One of Eratosthenes's most famous achievements was calculating Earth's circumference. He accomplished this by measuring the angle of the sun's rays at noon on the summer solstice at two locations, Alexandria and Syene (modern-day Aswan), which were known to be on the same meridian, or longitude. Figure 4-2 shows an abstraction of some of the geometry involved in this brilliant experiment.

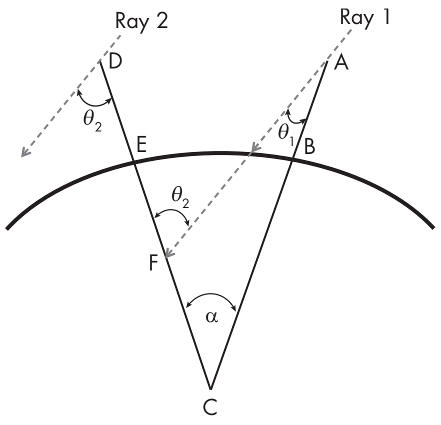


Figure 4-2: Measuring Earth's circumference

In this diagram, we can think of points E and B as two locations on Earth's surface (for example, Alexandria and Syene). We'll assume for the sake of simplicity that Earth is a perfect sphere and that both of these locations are on the same meridian. Let's also assume that *AB* and *DE* are two tall poles or towers that are sufficiently far apart that when extended to Earth's center (C), they create a small but measurable angle α. Two parallel rays coming from the sun just miss the tops of the poles and will hit the ground at slightly different angles. Because Earth's surface is curved, the angle θ_2 between ray 2 and *DE* will be slightly greater than the angle θ_1 between ray 1 and *AB*. As a result, the shadow of *DE* on the ground will be longer than that of *AB*, even if the poles themselves have the same height.

We also assume that at both locations the poles are positioned vertically relative to the ground surface, which can be thought of as flat in the vicinity of the poles. This last assumption allows us to measure the angle of a ray of light relative to a pole. Assuming that the length of the shadow is *s* and the height of the pole is *h*, the angle θ between the ray and the pole can be expressed as follows:

$$\theta = \tan^{-1}\left(\frac{s}{h}\right) \tag{4.1}$$

Finally, consider triangle *ACF* in Figure 4-2. According to the *exterior angle theorem*, the triangle's exterior angle *AFE* must be equal to the sum of the two interior opposite angles, *ACF* and *CAF*. Meanwhile, *AFE* and θ_2 are equal because these are the *alternate interior angles* of the line *DF* that intersects the two parallel solar rays. Therefore, the following must be true:

$$\angle AFE = \angle ACF + \angle CAF$$

$$\theta_2 = \theta_1 + \alpha$$

Rearranging the latter, we get:

$$\alpha = \theta_2 - \theta_1 \tag{4.2}$$

This final equation is what we need to estimate Earth's circumference. We'll achieve that by using another geometric relationship that connects the length *d* of an arc of a circle to the angle α (in radians) that the arc creates at the center of the circle:

$$\frac{\alpha}{d} = \frac{2\pi}{\text{Circumference}}$$

Solving for the circumference gives us this equation:

$$\text{Circumference} = \frac{2\pi d}{\alpha} \tag{4.3}$$

What Eratosthenes did was quite ingenious. He knew that at noon on the summer solstice, the sun would be directly overhead at Syene (point B in Figure 4-2), so a vertical pole there would cast no shadow, meaning $\theta_1 = 0$, and therefore $\alpha = \theta_2$ per Equation 4.2. In Alexandria (point E), however, the sun would be at an angle, so a pole would cast a shadow on the ground. By measuring the length of this shadow, Eratosthenes was able to calculate the angle between the sun's ray (ray 2) and the pole (*DE*) using Equation 4.1. He found this angle to be about 7.2 degrees, or 0.12566370614 radians.

Eratosthenes was aware of the distance between Alexandria and Syene—he estimated it to be 5,000 stadia (about 800 kilometers). With this information and the angle of the shadow, he calculated Earth's circumference (using Equation 4.3) and arrived at a value of approximately 40,000 kilometers. Once he determined the circumference, he could also calculate Earth's radius. For any circle, the radius r can be calculated from the circumference as:

$$r = \frac{\text{Circumference}}{2\pi}$$

Filling in the circumference formula from Equation 4.3, Earth's radius R is:

$$R = \frac{d}{\alpha} \tag{4.4}$$

This calculation gave Eratosthenes a value of 6,370 kilometers, which is remarkably close to the actual value of approximately 6,371 kilometers.

The Code

Let's write some code to imitate the method Eratosthenes used to calculate Earth's circumference and radius. We'll make our program more flexible by allowing the sun to not be directly overhead at the first location. To do this, we'll use Equation 4.1 to figure out the shadow angles, Equation 4.2 to get the arc angle, Equation 4.3 to calculate the circumference, and finally, Equation 4.4 to calculate the radius.

```
import kotlin.math.atan

data class Earth(
    val alpha: Double,
    val circumference: Int,
    val radius: Int
)

fun calculateEarthMetrics(s1: Double, h1: Double,
                          s2: Double, h2: Double, d: Double): Earth {
    // Calculate the angles of the shadows.
    val theta1 = atan(s1 / h1)
    val theta2 = atan(s2 / h2)

    // Calculate the angle at the center of Earth.
    val alpha = theta2 - theta1

    // Calculate the circumference and radius.
    val circumference = (2 * Math.PI * d / alpha).toInt()
    val radius = (d / alpha).toInt()

    return Earth(alpha, circumference, radius)
}

fun main() {
    // known values
    val shadow1 = 0.0    // m
    val height1 = 7.0    // m
    val shadow2 = 0.884  // m
    val height2 = 7.0    // m
    val distanceBetweenCities = 800.0 // in km
    val (alpha, circumference, radius) =
        calculateEarthMetrics(s1=shadow1, h1=height1,
            s2=shadow2, h2=height2,
            d=distanceBetweenCities)

    // Output the estimated circumference and radius.
    println("\n*** Measuring Earth's Circumference and Radius ***\n")
    println("Angle (alpha): ${"%7.5f".format(alpha)} radian")
    println("Circumference: $circumference kilometers")
    println("Radius: $radius kilometers")
}
```

The code segment starts by importing the required math function and defining a data class `Earth` with three properties: `alpha`, `circumference`, and `radius`. This data class allows us to conveniently package up the values estimated inside the `calculateEarthMetrics()` function and return them via a single `Earth` object.

The `calculateEarthMetrics()` function has five named parameters that represent the shadow lengths (`s1` and `s2`) and heights (`h1` and `h2`) for the two locations, and the distance (`d`) between these two locations. Then the function follows the steps described on page 137: calculating `theta1` and `theta2`, using them to calculate `alpha`, and using `alpha` to estimate `circumference` and `radius`. Since these are large numbers (when expressed in kilometers), we convert both `circumference` and `radius` into integers (which is how they were defined in the `Earth` class).

The `main()` function's job is quite simple: call the `calculateEarthMetrics()` function; receive the values of `alpha`, `circumference`, and `radius` by deconstructing the returned object; and print them with appropriate annotations and format.

The Result

For the given parameter values in this example—the same ones Eratosthenes used—the output of our program looks like this:

```
*** Measuring Earth's Circumference and Radius ***

Angle (alpha): 0.12562 radian
Circumference: 40013 kilometers
Radius: 6368 kilometers
```

Feel free to use this tool to experiment with different parameter values. For example, you could try using shadow lengths and angles measured on an exoplanet and find out how large or small the planet is!

Project 13: Code the Fibonacci Sequence

Leonardo of Pisa, commonly known as Fibonacci, was an Italian mathematician born c. 1170. From an early age, he showed a keen interest in mathematics, and his travels to North Africa and the Middle East exposed him to advanced mathematical concepts that weren't yet known in Europe. Fibonacci's most significant contribution to mathematics was the introduction of the Indo-Arabic numeral system to the Western world, which included the use of zero. This system replaced the previously used Roman numerals and revolutionized arithmetic calculations, making them significantly more efficient.

Fibonacci is widely recognized for introducing the Fibonacci sequence, a series of numbers where each number is the sum of the two preceding numbers. To explain this concept in his book *Liber Abaci*, Fibonacci used a colorful analogy involving a pair of rabbits. Imagine placing a pair of rabbits in an enclosed area. The rabbits can mate when they're one month old and can produce a new pair of rabbits when they're two months old. Therefore, it takes one month for each new pair to mature and an additional month to give birth to a new pair. If the rabbits never die and the mating continues, how many pairs of rabbits will there be after each month?

The solution to this problem forms the Fibonacci sequence. If we start with (1, 1) representing the starting pair over the first two months, the sequence will look like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on. These numbers can be linked to Fibonacci's rabbit example as explained in Table 4-1.

Table 4-1: Fibonacci's Rabbits

Month	Young pairs	Mature pairs	Total pairs	Explanation
0	1	0	1	Start with a newborn pair, no mature pair, and no offspring.
1	0	1	1	The first pair becomes mature and will reproduce at the end of this period.
2	1	1	2	The first pair of offspring is born. One mature pair will reproduce again at the end of this period.
3	1	2	3	The second pair of offspring is born. Two mature pairs will reproduce at the end of this period.
4	2	3	5	Two new pairs are born. Three mature pairs will reproduce at the end of this period.
5	3	5	8	Three new pairs are born. Five mature pairs will reproduce at the end of this period.

Say we want to determine how many rabbit pairs will exist after a certain number of generations. Here are the steps we can take, using the Fibonacci sequence:

1. Set the first two numbers in the sequence. By convention, these are usually 0 and 1 rather than 1 and 1.
2. Add the first two numbers to get the third number in the sequence.
3. Generate the next number by adding the two preceding numbers. This step can be mathematically expressed as $F_n = F_{n-1} + F_{n-2}$, where $n \geq 2$.
4. Repeat step 3 until the stopping condition is met.

The Fibonacci sequence has become a classic example of recursive sequences and is used to illustrate many mathematical concepts in various fields. Before getting into how to code this sequence, I'll introduce you to two other related concepts: the golden ratio and the Fibonacci spiral. These concepts will be illustrated in our Fibonacci code.

The Golden Ratio

The *golden ratio*, also known as the golden mean, is a mathematical ratio commonly found in nature, art, and architecture. The ratio is approximately 1.61803398875 and is denoted by the Greek letter φ.

The golden ratio is linked to the Fibonacci sequence: as the sequence continues, the ratio between each successive pair of numbers approaches the golden ratio. Starting with 1 (we can't start with 0, as the ratio of 1 over 0 is infinity), if we calculate and plot this ratio for each successive pair, it will rapidly converge on φ, as shown in Figure 4-3.

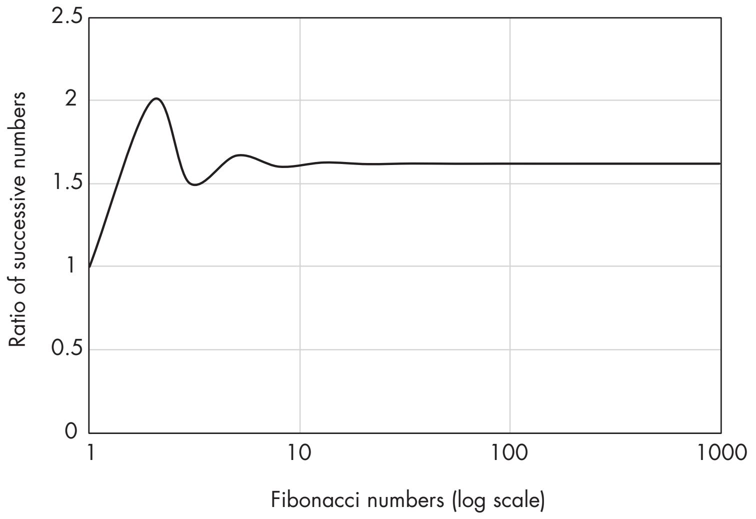


Figure 4-3: Convergence to the golden ratio

Well-known manifestations of the golden ratio in nature include nautilus shells, the arrangements of seeds in a sunflower and scales on a pine cone, and the proportions of the human body (for example, the ratio of the length of the forearm to the hand, and the ratio of the overall height to the height of the navel). The ratio has also been extensively exploited by artists, musicians, photographers, product designers, and architects in their work. In architecture, for example, it might determine the ratio of width to height for a building's facade.

The Fibonacci Spiral

The Fibonacci spiral is a geometric pattern derived from the Fibonacci sequence. It's created by drawing a series of quarter circles inside squares that are based on the numbers in the Fibonacci sequence. To draw the Fibonacci spiral, follow these steps:

1. Draw a small square with a side length of 1.
2. Draw another square of side length 1 adjacent to the first square, sharing a side.
3. Draw a third square of side length 2 adjacent to the second square, sharing a side.
4. Draw a fourth square of side length 3 adjacent to the third square, sharing a side.
5. Continue this process, drawing squares with side lengths equal to the sum of the two preceding squares, adjacent to the last drawn square, sharing a side.
6. Draw a quarter circle inside each of the squares, connecting the opposite corners of each square. The quarter circles will form a smooth curve: the Fibonacci spiral.

If you follow these steps and draw the spiral for the first eight numbers (starting from 1), the result will look like Figure 4-4.

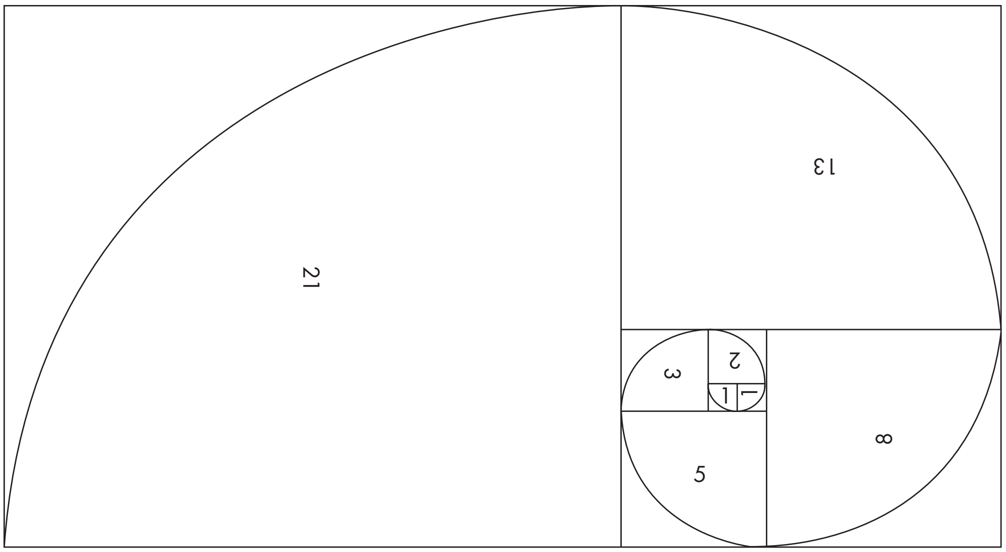


Figure 4-4: A Fibonacci spiral for the first eight numbers

The Fibonacci spiral, often associated with the golden ratio, is a recurring pattern in nature, appearing in various forms such as seashells, leaf arrangements, and even the spirals of distant galaxies! While the golden ratio is not an absolute prerequisite for beauty or efficiency in design, it undeniably holds an enduring charm as a mathematical concept that continues to capture our imagination.

The Code

Generating and printing the Fibonacci sequence up to a certain limit can be accomplished with just a few lines of code. Let's take this project a step further: along with generating the sequence itself, we'll also draw the Fibonacci spiral. This way, we'll be able to practice some of the data visualization techniques covered in the previous chapter and anticipate future projects where we'll gain deeper insight into a problem by visualizing the program output.

Setting Up

To begin, we'll write the global components of the code, including the import block needed for the visualization, the `FibonacciSpiral` application class, and the `main()` function.

```
import javafx.application.Application
import javafx.scene.Scene
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.scene.layout.Pane
import javafx.scene.paint.Color
import javafx.scene.shape.ArcType
import javafx.scene.text.Font
import javafx.stage.Stage

// number of Fibonacci numbers in the list
val N = 9
val fibs = mutableListOf<Int>()

// canvas-related parameters
val canvasW = 1000.0
val canvasH = 750.0

// Scaling parameters: adjust as needed.
val xOffset = 150
val yOffset = 50
val amplify = 25.0

class FibonacciSpiral : Application() {
    override fun start(stage: Stage) {
        val root = Pane()
        val canvas = Canvas(canvasW, canvasH)
        val gc = canvas.graphicsContext2D
        gc.translate(canvas.width / 2 + xOffset,
            canvas.height / 2 + yOffset)
        root.children.add(canvas)

        val scene1 = Scene(root, canvasW, canvasH)
        scene1.fill = Color.WHITE
        with(stage) {
            title = "Fibonacci Spiral"
            scene = scene1
            show()
        }

        // code for Fibonacci sequence and spiral
        generateFibonacciNumbers()
        drawFibonacciSpiral(gc)
        printFibonacciSequenceAndRatios()
    }
}

fun main() {
    Application.launch(FibonacciSpiral::class.java)
}
```

The code segment starts with an import block that provides access to a number of JavaFX graphics features that we'll use to draw the Fibonacci spiral on a `canvas` object. See Chapter 3 for a review of these features. Coding in IntelliJ IDEA means you don't need to memorize which library features you need to import; as you use the default template and add code that may require additional graphics elements, the IDE will import those features automatically.

Following the import block, we set up some global parameters. First, we create a variable called `N` to set how far into the Fibonacci sequence we'll go (starting from 0). Then, we create a mutable list of type `Int` named `fibs`, which will store the Fibonacci sequence as we calculate it. We also set several parameters to create a canvas where we'll draw the Fibonacci spiral. To define the size of the canvas, we use the values `canvasW` and `canvasH`, and to set the starting location of the origin of the coordinate system, we use `xOffset` and `yOffset`. For this particular project, I've set the canvas size to 1,000 pixels wide and 750 pixels high, which should be suitable for most screen sizes and resolutions.

It's important to note that the length of a line or the side of a rectangle on the canvas is measured in pixels. To plot the Fibonacci spiral, we'll start with a square of size 1. However, drawing a square of 1 pixel would result in a tiny dot on the screen, which we don't want. To avoid this, we'll use an amplification factor called `amplify` and set it to `25`. Therefore, the first square will be 25 pixels in size, and all subsequent squares will be amplified by the same factor. This ensures that the end result is a Fibonacci spiral that fills the canvas nicely.

In the `FibonacciSpiral` application class, we first create a layout container called `root` of type `Pane()`, which is the most basic layout container in JavaFX to hold and position nodes (user interface components) in a scene. We use `root` to hold the `canvas` on which the spiral will be drawn. Notice how we use the `translate` property of the graphics context `gc` to shift the initial position of the origin from the top-left corner (the default) to a position offset a bit from the middle of the canvas, where we'll draw the first Fibonacci square (see Figure 4-4 to get a sense of where that is). The rest of the class is routine JavaFX: we assign `canvas` to `root`, which is assigned to `scene1`, which connects to `stage`, the primary display window for this application.

Next, we move to the problem-specific segment of the application class, which consists of calls to three separate functions: generateFibonacciNumbers(), drawFibonacciSpiral(), and printFibonacciSequenceAndRatios(). These functions do exactly what their names suggest, and we'll discuss them in detail shortly.

Finally, the main() function contains a single line of code that launches a JavaFX application by calling the launch() method of the Application class, passing it the FibonacciSpiral class as an argument.

Generating the Fibonacci Sequence

The generateFibonacciNumbers() function generates the Fibonacci sequence as discussed earlier.

```
fun generateFibonacciNumbers() {
    // Add the starting pair.
    fibs.add(0)
    fibs.add(1)

    // Generate the sequence.
    for (i in 2 until N) {
        fibs.add(fibs[i-1] + fibs[i-2])
    }
}
```

First, $F(0)$ and $F(1)$ are set to 0 and 1, respectively, and then the rest of the sequence is generated using $F_n = F_{n-1} + F_{n-2}$, where $n \geq 2$. We add all generated numbers to the mutable list fibs using its fibs.add() method.

Drawing the Fibonacci Spiral

The drawFibonacciSpiral() function drives the process of drawing the Fibonacci spiral using the generated sequence of Fibonacci numbers, with support from two other helper functions that label each square with its corresponding Fibonacci number and draw the quarter circles.

```
fun drawFibonacciSpiral(gc: GraphicsContext) {
    for (i in 1 until N) {
      ❶ val side = fibs[i] * amplify
      ❷ with(gc) {
            strokeRect(0.0, 0.0, side, side)
            drawText(i, gc, side)
            drawArc(gc, side)
            // Move to the opposite corner by adding
            // side to both x- and y-coordinates.
            translate(side, side)
            // Rotate the axes counterclockwise.
            rotate(-90.0)
        }
    }
}
```

```
fun drawText(i: Int, gc: GraphicsContext, side: Double) {
    gc.fill = Color.BLACK
    with(gc) {
      ❸ font = when {
            i <= 2 -> Font.font(12.0)
            else -> Font.font(24.0)
        }
        fillText(fibs[i].toString(), side/2, side/2)
    }
}

fun drawArc(gc: GraphicsContext, side: Double) {
    val x = 0.0
    val y = -side
    with(gc) {
        lineWidth = 3.0
        strokeArc(x, y, 2*side, 2*side,
            -90.0, -90.0, ArcType.OPEN)
    }
}
```

The `drawFibonacciSpiral()` function uses a `for` loop that starts with `1` (we cannot draw a square of size `0`) to iterate over the sequence of numbers. In the loop, we retrieve the current number and multiply it by `amplify` to properly scale the squares on the screen ❶. Then we implement the rest of the process inside a `with(gc)` block ❷ where we draw a square, annotate it, and draw an arc. At the end of each cycle, we move the canvas's origin to the next location and rotate the coordinate system counterclockwise by 90 degrees. This way, the squares will spiral outward, as shown in Figure 4-4, but we'll still be able to draw each one with the same `strokeRect(0.0, 0.0, side, side)` call.

In the `drawText()` function, we use the argument `i`, which represents the index of the current Fibonacci number, to set the font size of the text used for annotation ❸. This ensures that the first two numbers fit inside a square of size 25 pixels. We then use the `fillText()` method of the graphics context `gc` to draw the number in the middle of its corresponding square.

The `drawArc()` function sets up the parameter values needed by the `strokeArc()` method of the graphics context. These parameters include the top-left corner of the rectangle, its width and height, the starting angle with respect to the x-axis in degrees, and the length of the arc in degrees. We also specify the arc type as `OPEN`, which means the two endpoints won't be connected with a line.

For drawing the arc, think of the imaginary box inside which the arc will be drawn as a stand-alone object that has its own coordinate system whose origin is at the center of the box. Inside this box, the positive x-axis points east and the positive y-axis points north. (Note that this isn't the same as the default convention used by the JavaFX canvas.) Taking this into account, drawing an arc counterclockwise is considered the positive direction, and this is how the starting angle and arc length are specified. For example, we've specified the start angle as –90 degress and the arc length as –90 degrees (both in the clockwise direction relative to the positive x-axis). Alternatively, we could have specified the start angle as +180 degress and the arc angle as +90 degrees (both counterclockwise) to produce the same result.

Printing the Sequence

We have one more function that prints the Fibonacci sequence, as well as the ratios between successive terms in the sequence, to illustrate how these values converge on the golden ratio.

```
private fun printFibonacciSequenceAndRatios() {
    println("\n*** Fibonacci sequence and ratios ***\n")
    println("Length of Fibonacci sequence=${fibs.size}")
    println("Generated sequence:")
  ❶ println(fibs)
    println("\nRatio F(n+1)/F(n) [starting from (1,1) pair]:")
    for (i in 2 until fibs.size) {
        println("%5d".format(fibs[i-1]) +
                "%5d".format(fibs[i]) +
                "%12.6f".format(fibs[i].toDouble()/fibs[i-1])
        )
    }
}
```

The function first prints a header message and the length of the generated Fibonacci sequence. Next, it prints the generated sequence itself using `println()` ❶. Finally, a `for` loop calculates and prints the ratios of adjacent numbers in the sequence, using `format()` to show the values with appropriate spacing and precision.

The Result

When you run the code, the text portion of the output should appear as follows:

```
*** Fibonacci sequence and ratios ***

Length of Fibonacci sequence=9
Generated sequence:
[0, 1, 1, 2, 3, 5, 8, 13, 21]

Ratio F(n+1)/F(n) [starting from (1,1) pair]:
    1    1    1.000000
    1    2    2.000000
    2    3    1.500000
    3    5    1.666667
    5    8    1.600000
    8   13    1.625000
   13   21    1.615385
```

Notice how the ratios initially zigzag around the value of 1.61803398875 but quickly approach the golden ratio once we reach the 10th pair in the sequence.

Of course, the app also displays a beautiful Fibonacci spiral drawn on the canvas using JavaFX. It should look exactly like Figure 4-4—that figure was generated with this very code!

EXERCISE

A concept related to the Fibonacci sequence and the golden ratio is Pascal's triangle, named after the French mathematician Blaise Pascal (although it was known to Chinese mathematicians over 500 years earlier). It has many interesting properties and applications in mathematics, including its use in calculating binomial coefficients, which arise in probability theory and other fields.

Pascal's triangle starts with a row containing a single number 1. Each subsequent row has one more number than the row above it. Each number is determined by adding the adjacent pair of numbers directly above it (except for a 1 on either end of each row). The first seven rows of Pascal's triangle are shown here:

```
                  1
                1   1
              1   2   1
            1   3   3   1
          1   4   6   4   1
        1   5   10  10  5   1
      1   6   15  20  15  6   1
```

Try writing the Kotlin code to generate Pascal's triangle up to some arbitrary length (say, 10 rows). Display the result as text output formatted to look like a triangle.

Project 14: Find the Shortest Distance Between Two Locations on Earth

We use the Pythagorean theorem to calculate distances between points on the same plane. However, for points on Earth's surface, this method isn't accurate over long distances, because it doesn't consider Earth's curved shape. That's where the *haversine formula* comes in. It calculates the shortest distance between two points on the surface of a sphere using the latitude and longitude coordinates of the points. In the case of Earth, the formula isn't totally accurate, since Earth isn't perfectly spherical, but it still offers a reasonable distance approximation for many practical applications, including in navigation, astronomy, and geography.

The haversine formula revolves around the concept of a *great circle*, the largest circle that can be drawn on a sphere. It's formed by the intersection of the sphere's surface with a plane that passes through the sphere's center.

The great circle divides the sphere into two equal halves, and its circumference matches the circumference of the sphere itself.

Figure 4-5 showcases two prominent great circles: the equator and the prime meridian. The equator acts as a dividing line between the northern and southern hemispheres, while the prime meridian (which passes through Greenwich, England) separates the Eastern and Western Hemispheres on Earth's surface. These two great circles serve as references for latitude and longitude, which together define the locations of points on Earth's surface.

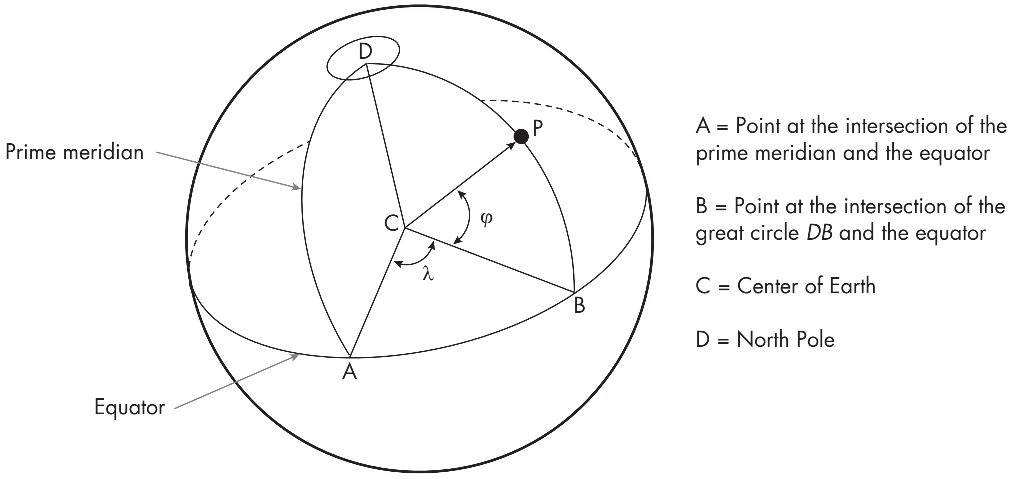

D
P
Prime meridian
C
φ
λ
B
A
Equator
A = Point at the intersection of the prime meridian and the equator
B = Point at the intersection of the great circle *DB* and the equator
C = Center of Earth
D = North Pole

Figure 4-5: The latitude and longitude of a point P

Latitude measures the distance north or south of the equator. It's expressed in degrees, with the equator being 0 degrees latitude, the North Pole 90 degrees north (90°N), and the South Pole 90 degrees south (90°S). In Figure 4-5, the latitude of point P would be denoted as φ°N, as it lies φ degrees north of the equator along a great circle that intersects with P and the North Pole. *Longitude* measures the distance east or west of the prime meridian. The prime meridian itself has a longitude of 0 degrees, and longitude values range from –180 degrees west of the prime meridian to 180 degrees east of the prime meridian. In the case of point P, its longitude would be λ°E, indicating its great circle is λ degrees east of the prime meridian.

NOTE *You may be used to seeing all latitude and longitude values expressed as positive numbers, but for the haversine formula to work, southern latitudes and western longitudes must be negative. Our program will automatically convert coordinates that don't follow this convention.*

Given any two points on the surface of a sphere, you can draw a great circle that intersects with both points, and that great circle will define the shortest path between the two points. If you know the angle θ (in radians) between the points—that is, the angle formed at the sphere's center by the arc connecting the points—and if you know the radius r of the sphere, you can calculate the distance along the sphere's surface between the two points as follows:

$$d = r\theta \tag{4.5}$$

In the case of Earth, we know the radius R to be about 6,371.009 kilometers, but how do we know the angle between two points on Earth's surface? This is where the haversine formula comes in. It uses the points' latitude and longitude coordinates, and a bit of trigonometry, to determine that angle, which in turn lets us calculate the distance between the points. The formula involves a little-known trigonometric function called the haversine function. The haversine of an angle θ is defined as follows:

$$\text{hav}(\theta) = \sin^2\left(\frac{\theta}{2}\right) \tag{4.6}$$

The haversine formula calculates a, the haversine of the angle between two points on Earth's surface, as follows:

$$a = \text{hav}(\varphi_2 - \varphi_1) + \cos(\varphi_1)\cos(\varphi_2)\,\text{hav}(\lambda_2 - \lambda_1)$$

Here (φ_1, λ_1) and (φ_2, λ_2) are the latitude and longitude coordinates of the two points, expressed in radians. To convert from degrees to radians, simply multiply by π and divide by 180.

We now have the haversine of the angle between the two points, but not the angle itself. For that, we can use the a we just calculated and the definition of haversine (Equation 4.6) to solve for the arc angle c:

$$c = 2\sin^{-1}\left(\sqrt{a}\right)$$

Now that we have the angle c, we have everything we need to calculate the distance between the points using Equation 4.5:

$$d = Rc$$

There's one catch, however: d works out to a real number only when $0 \le a \le 1$, but sometimes a can be pushed outside this range due to a floating-point error. To avoid this, we should instead express c as:

$$c = 2\sin^{-1}\sqrt{\max(0.0, \min(a, 1.0))}$$

This constrains the value of a to a range from 0 to 1, preventing any unrealistic results.

The Code

We now have everything we need to write a Kotlin program that calculates the shortest distance between two locations on Earth. For this example, I've hardcoded the locations of two well-known landmarks, Big Ben in London and the Statue of Liberty in New York, but you can use any locations you want. The code consists of four main segments: an import block and global declarations, the `main()` function, the `printLatLong()` function, and the `haversineDistance()` function. I'll discuss them in the same order.

```
// Import math functions.
import kotlin.math.sin
import kotlin.math.asin
import kotlin.math.cos
import kotlin.math.PI
import kotlin.math.max
import kotlin.math.min
import kotlin.math.pow
import kotlin.math.sqrt

// Define a Location data class.
data class Location(
    val name: String = "",
    var lat: Double,
    val latDir: String,
    var lon: Double,
    val lonDir: String
)

//  global variables and parameters
//  N = north, S = south, E = east, W = west
val L1 = Location(name = "Big Ben", lat=51.5004,
    latDir = "N", lon=0.12143, lonDir = "W")
val L2 = Location(name="Statue of Liberty", lat = 40.689978,
    latDir = "N", lon = 74.045448, lonDir = "W")
val locations = listOf(L1, L2)

val R = 6371.009  // radius of Earth in km

fun main() {
    println("\n*** Measuring Distance Between Two Locations on Earth ***\n")
    printLatLong(category = "input", locations)
    val d = haversineDistance()
    printLatLong(category = "adjusted", locations)
    println("\nThe distance between the two given locations:")
    println("d = ${"%10.2f".format(d)} km")
}

fun printLatLong(category: String, locationsToPrint: List<Location>) {
    when(category) {
        "input" ->
            println("...inputted coordinates...\n")
        "adjusted" ->
            println("\n...adjusted coordinates...\n")
    }
    locationsToPrint.forEach { location -> println(location)}
}
```

```
fun haversineDistance(): Double {
    // Adjust signs based on N-S and E-W directions.
  ❶ for (location in locations) {
        with(location) {
            if(latDir == "S" && lat > 0.0) lat = - lat
            if(lonDir == "W" && lon > 0.0) lon = - lon
        }
    }
    // Calculate the angles in radians.
    val phi1 = L1.lat * PI/180
    val phi2 = L2.lat * PI/180
    val delPhi = phi2 - phi1
    val delLambda = (L2.lon - L1.lon) * PI/180

    // Calculate the distance using haversine formula.
  ❷ val a = sin(delPhi/2).pow(2) +
            cos(phi1) * cos(phi2) *
            sin(delLambda/2).pow(2)
    // Ensure that 0 <= a <= 1 before calculating c.
  ❸ val c = 2 * asin(sqrt(max(0.0, min(1.0, a))))
  ❹ val d = R * c
    return d
}
```

Since the haversine formula needs to use quite a few math functions, we begin by importing those from the `kotlin.math` package. Next, we declare a data class `Location` with five properties: `name` (the name of the location), `lat` (the latitude), `latDir` (the direction of the latitude), `lon` (the longitude), and `lonDir` (the direction of the longitude). We then create two `Location` objects, `L1` and `L2`, representing Big Ben and the Statue of Liberty. We store them together in a list called `locations` so we can efficiently iterate over the locations.

Notice that I've provided the latitude and longitude values as positive numbers in degrees, regardless of which direction they're in. I'm relying on the `latDir` and `lonDir` properties to communicate that extra information. The convention is `N` for north, `S` for south, `E` for east, and `W` for west. In cases where the latitude or longitude of a location is exactly 0, the corresponding direction can be set to `EQ` (equator) or `PM` (prime meridian), although this won't impact the final result. Later, in the `haversineDistance()` function, we ensure that when the `lat` or `lon` direction is `S` or `W`, respectively, the corresponding values are always negative ❶.

The `main()` function prints the provided latitude and longitude values before and after adjustments (if any) using the `printLatLong()` function, makes a single call to the `haversineDistance()` function, and prints the result.

The `printLatLong()` function takes one argument, `category`, which is of type `String`. The `category` parameter is passed to a `when` block to determine which of two messages to print, indicating whether the coordinates have been adjusted for their direction properties. The locations themselves are then printed one at a time using the `forEach()` method of the `locationsToPrint` list. We could have used a regular `for` loop here, but some Kotlin enthusiasts consider `forEach()` to be more idiomatic.

Finally, the haversineDistance() function calculates the shortest distance between the two locations on a spherical surface. It first iterates over the locations and negates the latitudes and longitudes if needed ❶, then converts all the latitude and longitude values from degrees to radians. Next, it steps through the equations we discussed on page 150, using the coordinates to calculate *a* ❷, using *a* (constrained to between 0 and 1) to calculate the angle *c* ❸, and using *c* to calculate and return the haversine distance *d* between the points ❹.

The Result

When you run the program for the set location and parameter values, the output should appear as follows:

```
*** Measuring Distance Between Two Locations on Earth ***

...inputted coordinates...

Location(name=Big Ben, lat=51.5004, latDir=N, lon=0.12143, lonDir=W)
Location(name=Statue of Liberty, lat=40.689978, latDir=N, lon=74.045448,
lonDir=W)

...adjusted coordinates...

Location(name=Big Ben, lat=51.5004, latDir=N, lon=-0.12143, lonDir=W)
Location(name=Statue of Liberty, lat=40.689978, latDir=N, lon=-74.045448,
lonDir=W)

The distance between the two given locations:
d =   5575.08 km
```

As mentioned on page 148, the haversine calculation assumes that Earth is a perfect sphere, which is not true. In fact, Earth is an oblate spheroid, slightly flattened at the poles and bulging at the equator. To get around this wrinkle, you could use Vincenty's formula, which takes into account the oblate spheroidal shape of Earth by considering Earth's equatorial and polar diameters. Which formula to use really depends on the nature of the problem, as both methods have their strengths and their weaknesses.

EXERCISE

Every time you fly to a distant location for business or pleasure, you might wonder about the minimum flight distance between the departing and arriving cities. Now that you know how to calculate the haversine distance, you can find the answer if you know the latitude and longitude coordinates of the locations. Try finding the shortest distance between the locations listed here. (You can look up the coordinates for these locations online; for the last two cases, you should already know what they are.)

(continued)

- Tokyo, Japan, and Sydney, Australia
- Paris, France, and Lima, Peru
- Dublin, Ireland, and Ankara, Turkey
- The North Pole and South Pole
- Two points where the equator and the prime meridian intersect, located on opposite sides of the globe

As an added step, use the calculated distances and an average speed of air travel to estimate the flight times between these locations.

Project 15: Do Encryption with the Hill Cipher

In today's interconnected world, we constantly share sensitive data like personal information, financial details, and confidential messages. What's to stop unauthorized parties from accessing that information? The answer is *encryption*, a set of techniques for scrambling our data into gibberish that can be deciphered only with the right key. Encryption protects our privacy, safeguards against hackers and cybercriminals, and secures our online transactions.

There are a variety of encryption algorithms in use today. In this project, we'll focus on a particular algorithm called the Hill cipher, developed by American mathematician Lester S. Hill in 1929. According to this method, the *plaintext* (text in plain English or any other language) is divided into blocks of fixed size and represented as vectors. These vectors are then multiplied by a square matrix called the *encryption key*, modulo a specified number, to obtain the *ciphertext* (encrypted text). For decryption, the ciphertext vectors are multiplied by the inverse of the encryption key matrix, modulo the same specified number.

Hill's encryption method can be vulnerable to attacks if we don't choose the encryption key matrix carefully. While it's no longer employed as the sole encryption mechanism, it can still be incorporated into more sophisticated methods and remains a valuable concept to grasp. Plus, exploring Hill's method provides an excellent opportunity to apply and enhance our coding skills in the crucial field of cryptography.

How It Works

The Hill cipher revolves around concepts from linear algebra and modulo operations. I don't expect you to have an in-depth knowledge of these areas of mathematics, but you may wish to review these topics to gain a better insight into how the Hill cipher actually works, as well as its strengths and weaknesses. Here are brief definitions of the key terms that we'll use in this project:

Vector

A one-dimensional sequence of values. For example, [1, 3, 5] is a row vector with three elements.

Matrix

A two-dimensional collection of values, arranged in rows and columns. For example, a 3×3 matrix has three rows and three columns, and a total of nine elements (numbers) that can be real or complex.

Determinant

A single value calculated using the elements of a matrix. The matrix must be square, meaning it has the same number of rows and columns. Say we have the following square matrix A:

$$A = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

Its determinant, denoted by $\det(A)$, $\det A$, or $|A|$, can be calculated as follows:

$$\det(A) = aei + bfg + cdh - ceg - bdi - afh$$

Identity matrix

A square matrix, often denoted as I, where all the elements along a diagonal from the top left to the bottom right have a value of 1 and all other elements have a value of 0. A 3×3 identity matrix looks like this:

$$I = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Inverse matrix

For a given matrix A, its inverse A^{-1} is another matrix such that multiplying the two matrices results in the identity matrix (that is, $AA^{-1} = I$). A matrix must be square to have an inverse, although not all square matrices have one.

Modulo

An operation represented by the symbol % that finds the remainder when one number is divided by another. For example, 5 % 2 is 1. Modulo (*mod* for short) is a multipurpose operator used in various applications, such as determining divisibility, cycling through a range of values, and handling periodic patterns. Hill's algorithm relies on the modulo operation to keep the encrypted and decrypted texts within the same alphabet as the plaintext. Thus, the size of the alphabet serves as the *base* or *modulus* (the number after the % operator) for these operations, guaranteeing valid ciphertext and plaintext representations.

Modular multiplicative inverse (MMI)

For a given integer a and a modulus m, the MMI is positive integer x such that $ax \% m = 1$. The value of x must be less than the modulus. For example, the MMI for 5 modulo 11 is 9, because $(5 * 9) \% 11 = 1$, and 9 is less than 11.

Armed with these definitions, let's now dive into the core encryption and decryption steps employed by the Hill cipher and highlight some of our Kotlin implementation.

For Encryption

1. Define the alphabet. Choose which letters are to be used for writing plain and encrypted messages. For messages written in English, the alphabet size should be at least 26 to include all lowercase letters. We'll also include a period, a space, and a question mark, giving us an alphabet of 29 characters total. The size of the alphabet serves as the modulus.
2. Choose a block size. During encryption and decryption, the message is divided into small blocks of characters, each of the same length. In this exercise, we'll have three characters per block.
3. Generate the encryption key matrix. For the purposes of this project, I've generated the encryption key matrix for you, but if you're curious, it must adhere to these rules:
   a. The matrix must be square and have the same dimension as the block size chosen in step 2. In this case, with a block size of 3, we need a 3×3 matrix.
   b. The determinant of the matrix can't be 0.
   c. The determinant must not share a factor, other than 1, with the modulus from step 1.
4. Prepare the plaintext. Divide the plaintext message into blocks based on the chosen block size. If the last block is smaller than the fixed size, pad it with filler characters. We'll use spaces for padding to ensure that the message remains the same after decryption, with no extra visible characters.
5. Create vectors from the plaintext. Each block of the plaintext must be converted into a numerical vector with the same length as the block size. To assign numerical values to characters, we'll save the alphabet in a single `String` object. We can then map each character in the plaintext to that character's corresponding index in the string. For example, `a` will be mapped to `0`, `b` to `1`, and so on. This way, the block `cab` will become `(2, 0, 1)`, a vector of size 3.
6. Encrypt the message. For each block, carry out the following steps:
   a. Multiply the block's plaintext vector by the key matrix, modulo 29, to generate a ciphertext vector.

   b. Convert the numerical values in the ciphertext vector back to text characters using the reverse mapping scheme.
   c. Add the ciphered characters to a mutable list of characters, which will become the encrypted message (ciphertext) once all the blocks have been processed.

For Decryption

1. Generate the decryption key matrix. To decipher the encrypted text, we must first create the inverse of the encryption key matrix modulo the specified number. This process involves multiple linear algebraic steps and modulo operations; for simplicity, I've provided the inverse matrix. If you want to work with a different set of key and inverse matrices, you can look up the online tools that will generate those for you.
2. Prepare the ciphertext. Divide the ciphertext into blocks of the specified size used during encryption (3 for this project). This time, no padding is needed, since the padding was already done during encryption.
3. Create vectors from the ciphertext. Use the same character-numbering scheme to convert the ciphertext blocks into ciphertext vectors of size 3.
4. Decrypt the message. For each block, carry out the following steps:
   a. Multiply the ciphertext vector by the decryption key matrix, modulo 29, to generate a deciphered vector.
   b. Convert the numerical values in the deciphered vector back to text characters using the reverse mapping scheme.
   c. Add the deciphered characters to a mutable list of characters, which will become the decrypted message (plaintext) once all the blocks have been processed.

Finally, keep in mind that it's customary to pick an encryption key matrix made up of only integers, preferably falling between 0 and the modulus.

The Code

We're ready to implement the Kotlin code for Hill's encryption and decryption method. The code is organized in a top-down manner, starting with global declarations, continuing with the `main()` function, and ending with a series of short helper functions. We'll review everything in sequence.

Variables and Data Structures

We begin by declaring the variables and data structures needed to implement Hill's method.

```
/*  --- Hill's method for encrypting and decrypting texts --- */

// Declare the key matrix and its inverse.
// keyInv is based on mod 29.
val key = arrayOf(
    intArrayOf(13, 11, 6),
    intArrayOf(15, 21, 8),
    intArrayOf(5, 7, 9)
)

val keyInv = arrayOf(
    intArrayOf(1, 12, 8),
    intArrayOf(20, 0, 6),
    intArrayOf(0, 3, 20)
)

val dim = key.size
const val alphabet = "abcdefghijklmnopqrstuvwxyz .?"

data class Block(
    val t1: Char,
    val t2: Char,
    val t3: Char,
)

val indexVector = IntArray(dim)
val processedVector = IntArray(dim)
val blocks = mutableListOf<Block>()
val processedText = mutableListOf<Char>()
```

First, we create the matrices for encryption and decryption (`key` and `keyInv`). For this project, we'll accept these as given, but you can use online tools to create a different encryption key matrix that meets the required conditions and calculate the corresponding inverse matrix. The size of these square matrices is captured in the parameter `dim`, which is later used as the block size for processing messages. We also define a string called `alphabet` that stores all the valid letters that can be used in the plaintext and ciphertext.

Next, we introduce a data class called `Block`, which we'll use to store the text blocks generated while processing the message. These blocks will be stored as a mutable list named `blocks`. We also create a few other collections to temporarily hold and manipulate the vectors created during encryption and decryption operations, along with a mutable list named `processedText` to store the final list of characters. Since the encryption and decryption processes are very similar, we'll be able to use these variables and collections during both processes to store the intermediate and final values.

The main() Function

The `main()` function calls a series of helper functions to coordinate the overall encryption or decryption process.

```
fun main() {
    println("\n*** Cryptography with Hill's Method ***\n")
    runValidation()
    println("\nEnter 1 for encryption or 2 for decryption:")

  ❶ when(val choice = readln().toInt()) {
        1 -> {
            println("You have chosen encryption\n")
            getText()
            encrypt()
            printProcessedText(choice)
        }
        2 -> {
            println("You have chosen decryption\n")
            getText()
            decrypt()
            printProcessedText(choice)
        }
        else -> println("\nInvalid choice...exiting program\n")
    }
}
```

In `main()`, we first call the `runValidation()` function, which uses matrix multiplication (mod 29) to ensure that the encryption and decryption matrices are valid. We then prompt the user to choose which operation to carry out: encryption (enter **`1`**) or decryption (enter **`2`**). Based on the choice, we use a `when` block ❶ to implement the steps to encrypt or decrypt a message.

For both choices, we start with the `getText()` function, which takes in the message to be encrypted or decrypted from the user as a text string and divides it into blocks. We then call `encrypt()` or `decrypt()`, depending on the choice made earlier. Finally, we display the result with help from the `printProcessedText()` function.

The Helper Functions

There are several helper functions called from within the `main()` function. We'll turn to those next, starting with the functions that help validate the matrices.

```
fun runValidation() {
    println("key matrix dimension:")
    println("${key.size}  x  ${key[0].size}\n")

    // validation of key and keyInv
    val productMatrix = multiplyMatricesMod29(key, keyInv,
        r1=dim, c1=dim, c2=dim)
    displayProduct(productMatrix)
}

fun multiplyMatricesMod29(firstMatrix: Array <IntArray>,
                          secondMatrix: Array <IntArray>,
                          r1: Int,
                          c1: Int,
                          c2: Int): Array <IntArray> {
    val product = Array(r1) { IntArray(c2) }
    for (i in 0 until r1) {
        for (j in 0 until c2) {
            for (k in 0 until c1) {
                product[i][j] += (firstMatrix[i][k] *
                                  secondMatrix[k][j])
            }
          ❶ product[i][j] = product[i][j] % 29
        }
    }
    return product
}

fun displayProduct(product: Array <IntArray>) {
    println("[key * keyInv] mod 29 =")
    for (row in product) {
        for (column in row) {
            print("$column    ")
        }
        println()
    }
}
```

The `runValidation()` function displays the size of the key matrices. It then calls `multiplyMatricesMod29()` to do the validation check and shows the results with `displayProduct()`. The matrices are considered valid if one is the inverse of the other, modulo 29. If this is the case, the product of the two matrices, modulo 29, should be an identity matrix where all elements are zeros, except for ones along the diagonal from the top left to the bottom right.

In `multiplyMatricesMod29()`, we test this out, using three nested `for` loops to multiply the encryption and decryption key matrices, taking modulo 29 of each resulting value before putting it in the `product` matrix ❶. See the "Multiplying Two Matrices" box for details about the math behind this process.

MULTIPLYING TWO MATRICES

To multiply two matrices, we must first ensure that their shapes (number of rows and columns) are compatible: the number of columns in the first matrix must be equal to the number of rows in the second matrix. As a result, the product matrix will have the same number of rows as the first matrix and the same number of columns as the second matrix.

Once this condition is met, we work through the rows of one matrix and the corresponding columns of the other, multiplying and summing the values. Say we have the following two 3×3 matrices, *A* and *B*:

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{vmatrix}, \quad B = \begin{vmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{vmatrix}$$

Here are the steps to find their product matrix *C*:

1. Multiply the elements in the first row of matrix *A* by the elements in the first column of matrix *B* and add the results together: $(1 \times 9) + (2 \times 6) + (3 \times 3) = 9 + 12 + 9 = 30$. This is the value at *C*[1,1], the first row and first column of the product matrix.
2. Multiply the elements in the first row of matrix *A* by the elements in the second column of matrix *B* and add the results together: $(1 \times 8) + (2 \times 5) + (3 \times 2) = 8 + 10 + 6 = 24$. This is the value at *C*[1,2], the first row and second column of the product matrix.
3. Multiply the elements in the first row of matrix *A* by the elements in the third column of matrix *B* and add the results together: $(1 \times 7) + (2 \times 4) + (3 \times 1) = 7 + 8 + 3 = 18$. This is the value at *C*[1,3], the first row and third column of the product matrix.
4. Repeat steps 1 through 3 for the second and third rows of matrix *A* to get the values in the second and third rows of the product matrix *C*.

   After performing all these calculations, we get this result:

$$A = \begin{vmatrix} 30 & 24 & 18 \\ 84 & 69 & 54 \\ 138 & 114 & 90 \end{vmatrix}$$

The `displayProduct()` function neatly formats and prints the contents of the product matrix. As you'll later see in the example output, the result should indeed be an identity matrix.

Here's the `getText()` function, which we call from `main()` at the start of the encryption or decryption process:

```
fun getText() {
    println("Enter text for processing:")
    var text = readln().lowercase()
    val tmp = " " // Use a space for padding.

  ❶ when(text.length % 3) {
        1 -> text = text + tmp + tmp
        2 -> text += tmp
    }
    for (i in text.indices step 3)
        blocks.add(Block(text[i], text[i+1], text[i+2]))
}
```

The function uses `readln()` to take in the plaintext or ciphertext from the user. We convert all the characters to lowercase since we have only lowercase letters in our alphabet. We then check if the input string is divisible by `3` ❶ and pad it with spaces if not. Finally, we use a `for` loop with a step size of `3` to break the text into three-character blocks. Each one is stored in a `Block` object and added to the `blocks` mutable list.

The remainder of the helper functions do the work of actually encrypting and decrypting the text.

```
fun encrypt() {
    for (block in blocks) {
        getIndexBlock(block)
        encryptIndexBlock()
        addToProcessedText()
    }
}

fun decrypt() {
    for (block in blocks) {
        getIndexBlock(block)
        decryptIndexBlock()
        addToProcessedText()
    }
}

fun getIndexBlock(block: Block) {
    val (x,y,z) = block
    indexVector[0] = alphabet.indexOf(x)
    indexVector[1] = alphabet.indexOf(y)
    indexVector[2] = alphabet.indexOf(z)
}

fun encryptIndexBlock() {
    for (j in 0 until  3) {
        var sum = 0
        for (i in 0 until  3) {
            sum += indexVector[i] * key[i][j]
        }
        processedVector[j] = sum % 29
    }
}

fun decryptIndexBlock() {
    for (j in 0 until  3) {
        var sum = 0
        for (i in 0 until  3) {
            sum += indexVector[i] * keyInv[i][j]
        }
        processedVector[j] = sum % 29
    }
}

fun addToProcessedText() {
    processedVector.forEach { i ->
        processedText += alphabet[i]
    }
}
```

```
fun printProcessedText(choice: Int) {
    when(choice) {
        1 -> println("\nHere is the encrypted text:")
        2 -> println("\nHere is the decrypted text:")
    }
    print(processedText.joinToString(""))
}
```

The `encrypt()` and `decrypt()` functions both iterate through the `Block` objects in the `blocks` list and call a series of helpers to process them. The first helper called is the `getIndexBlock()` function, which looks up each character's index in the `alphabet` string, thereby converting each character to an integer. The values are stored in the `indexVector` array.

Next, we call `encryptIndexBlock()` or `decryptIndexBlock()`, which converts the plaintext vector into a cipher (encrypted) vector or vice versa by multiplying the vector by the appropriate matrix (`key` or `keyInv`), modulo 29. Multiplying a vector by a matrix is much like multiplying two matrices, but in this case, we need only two levels of `for` loops. The result goes in the `processedVector` array.

Our last encryption and decryption helper is the `addToProcessedText()` function, which takes each number from the `processedVector` array, looks up the corresponding character from the `alphabet` string, and adds that character to `processedText`, a mutable list. In the end, once all the vectors are processed, this list contains the final encrypted or decrypted text. Back in `main()`, we call the `printProcessedText()` function, which concatenates all the characters stored in the `processedText` list into a single string for easy printing.

The Result

Here's a sample run of the program in encryption mode:

```
*** Cryptography with Hill's Method ***

key matrix dimension:
3  x  3

[key * keyInv] mod 29 =
1    0    0
0    1    0
0    0    1

Enter 1 for encryption or 2 for decryption:
1
You have chosen encryption

Enter text for processing:
Code is like humor. It is bad code when you have to explain it.

Here is the encrypted text:
tsgsiomjjnhtvwpqxs.ahk?ru gbn tsgbtynurosksdoqfb a?ujsmtexvjcji
```

First, notice the validation check: the product of the two matrices is indeed an identity matrix, with ones running along the diagonal and zeros everywhere else. Then notice the final output, where the program has turned the readable plaintext into unreadable gibberish. The process also works in reverse: if you choose the decryption option (enter **2**) and input the encrypted text, the program will instantly convert the ciphertext back to the original plaintext.

Currently, the final result is displayed in all lowercase letters. I invite you to improve the `printProcessedText()` function so that the final result is capitalized as needed before printing. If you're thorough, you'll soon realize that implementing a complete set of capitalization rules isn't as simple as it sounds.

Project 16: Simulate a One-Dimensional Random Walk

So far in this chapter, the projects have all been *deterministic*, meaning there's a unique solution for a given set of input parameters. If we were to run the code multiple times with the same input, the output would remain unchanged. In this project, we'll explore a different kind of problem, one that's *stochastic* in nature. In a stochastic problem, the output for a given set of inputs isn't predetermined. We may be aware of various possible outcomes, or a range within which the output will fall, but the specific value generated by an individual instance of the experiment is determined purely by chance. To illustrate this concept, we'll probe the idea of a random walk.

A *random walk* is a process made up of a series of *random steps*, actions with multiple possible outcomes. We know the probability of each potential outcome, but the actual outcome is determined randomly. For example, rolling a die is a type of random step. Assuming the die is fair, each of its six sides will have the same likelihood of landing face up (one-sixth, or approximately 16.67 percent). Therefore, when we actually roll the die, we can't know for sure what number we'll get. Our guesses will be correct only 16.67 percent of the time.

Random walks can be described using a *mathematical space* with a certain number of dimensions, depending on the nature of the random step. Let's say we're considering the movement of a heavily inebriated person who has just come out of a pub. The street in front of the pub runs east–west. This person is totally disoriented and is taking random steps along the street in both directions. We can mathematically describe the distance the person travels over time as the sum of individual steps along the x-direction (the x-axis being the east–west line). We could record each step toward the east as +1 and a step in the opposite direction as –1 (assuming all steps cover the same distance). This is an example of a one-dimensional random walk—we need only the x-axis to describe it mathematically.

Now suppose the person has been drinking in the middle of an open field and has started to wander randomly in different directions. The person's steps can now have both an x-component (east or west) and a y-component (north or south). In this case, to measure the distance traveled from the center of the field, we'll have to track the person's movements in a two-dimensional space, which will make this a two-dimensional random walk problem.

A well-known example of a random walk is *Brownian motion*, named for Robert Brown, a 19th-century Scottish botanist. Using a microscope, Brown was observing grains of pollen immersed in water when he noticed that the grains were constantly moving in random directions. In fact, we can find similar movements whenever very small particles are injected in a fluid medium, such as dust or smoke particles in the air or the movement of particles in a colloidal suspension such as milk or paint. Brown's observation was an important scientific discovery that remained unexplained for more than half a century until 1905, when Albert Einstein explained that Brownian motion was caused by the continuous bombardment of the pollen grains by the surrounding water molecules.

In this exercise, we'll build and simulate a 1D random walk model in Kotlin. This will allow us to gain a deeper insight into how particles or objects move in one dimension through random steps. In particular, by repeating the simulation many times and plotting the results, we'll be able to identify patterns and explore the statistical properties that underlie this dynamic behavior.

A One-Dimensional Model

Imagine a single particle moving randomly along a line in small steps. For simplicity's sake, we'll assume that the particle's step size remains constant and that steps are made at steady time intervals (we do not need to use time as an explicit variable in our model). Physicists often call this scenario a *free diffusion* problem in one dimension. The process is schematically shown in Figure 4-6.

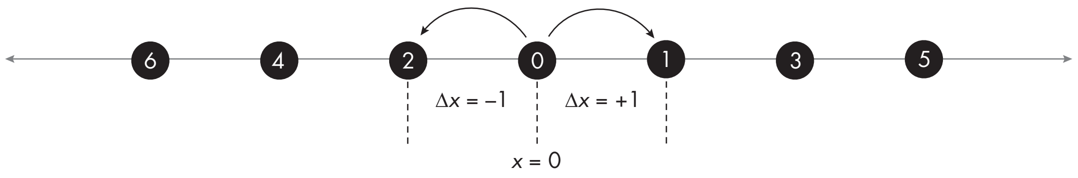


Figure 4-6: A random walk in one dimension, starting at x = 0

The particle starts at location $x = 0$ and moves in discrete steps of length $\Delta x = \pm 1$. The direction of the particle's movement is random, so its next position after 0 can be either 1 (with a displacement of +1) or 2 (with a displacement of −1). The probability p of choosing either direction is equal, so $p = 0.5$. Notice that at any given location, the particle can change its direction, so it's possible for the particle to take several random steps and end up back where it started.

The question we're interested in exploring is this: After making an arbitrary number of steps, n, how far on average will the particle have moved from its starting position? To answer this question, we'll need to simulate many random walks—say, 500 walks of 1,000 steps each—and analyze the results. We can't simply take the average of the cumulative distances traveled in the different simulations, however; the particle can drift in both the positive and negative x-directions, so the net-positive and net-negative distances would likely cancel each other out, giving us an average distance of roughly 0. Instead, we'll use the *root-mean-square (RMS) distance*, which is calculated in three steps:

1. Square all the distances from all the simulations after a given number of steps *n*. This converts any negative numbers to positive numbers.
2. Add all the results from step 1 and divide by the number of simulations to find the mean (average) of the squared distances.
3. Take the square root of step 2's result to arrive at the RMS distance.

Based on past research conducted on one-dimensional random walks, we know that the RMS distance exhibits a nonlinear relationship with the number of steps taken; in theory, the RMS distance after the *n*th step should equal the square root of *n*. To validate this notion, we'll compute the RMS distance (the *simulated* RMS) and the square root of *n* (the *theoretical* RMS) and plot them both against the number of steps, *n*. Hopefully, the two plots will be similar. We'll also plot the mean distance traveled at each time step, which should remain close to 0.

In a separate graph, we'll visualize the trajectories of the 500 random walks themselves. This should help illustrate the random nature of the outcomes and give further support to our theories about the cumulative and RMS distances traveled.

The Code

I'll present the code segments for this project in a top-down sequence, starting with some general setup code. Since we want to visualize the random walks from different simulations and examine the relationship between the cumulative, mean, and RMS distances with the number of steps, we'll use the JavaFX template that we developed and used in Chapter 3.

```
// import block
import javafx.application.Application
import javafx.scene.Node
import javafx.scene.Scene
import javafx.scene.chart.LineChart
import javafx.scene.chart.NumberAxis
import javafx.scene.chart.XYChart
import javafx.scene.control.ScrollPane
import javafx.scene.layout.Background
import javafx.scene.layout.BackgroundFill
import javafx.scene.layout.CornerRadii
import javafx.geometry.Insets
import javafx.scene.layout.VBox
```

```
  import javafx.scene.paint.Color
  import javafx.stage.Stage
  import kotlin.math.sqrt

  // data class
❶ data class State(
      var step: Double,
      var dist: Double
  )

  // global parameters
  val numStep = 1000
  val numSim = 500

❷ // Create lists needed for plotting line charts.
  val xList  : List<State> = List(numStep) { State(0.0, 0.0) }
  val avgList: List<State> = List(numStep) { State(0.0, 0.0) }
  val rmsList: List<State> = List(numStep) { State(0.0, 0.0) }
  val expList: List<State> = List(numStep) { State(0.0, 0.0) }

  val states1 = mutableListOf<List<State>>()
  val states2 = mutableListOf<List<State>>()

  class RandomWalk1D : Application() {
      override fun start(primaryStage: Stage) {
        ❸ val root = VBox()
          /*------------------------------------------*/
        ❹ root.styleClass.add("color-palette")
          root.background = Background(BackgroundFill(Color.WHITE,
               CornerRadii.EMPTY, Insets.EMPTY))
          /*------------------------------------------*/
        ❺ val scroll = ScrollPane()
          scroll.setContent = root
          val scene = Scene(scroll, 550.0, 850.0, Color.WHITE)
          primaryStage.title = "1D random Walk Simulation"
          primaryStage.scene = scene
          primaryStage.show()

          // ----- Random walk simulation starts here. -----
          // Call random walk function.
          randomWalk1d()
          // Get the theoretical RMS values.
          calcRMS1d()
          // Create line charts.
          createRWChart1(root)
          createRWChart2(root)
      }
  }

  fun main() {
      Application.launch(RandomWalk1D::class.java)
  }
```

The code segment starts with the import block. Since this project will use the XY charting features of JavaFX instead of the canvas feature, the import block is somewhat different from what we needed for Project 13, and it includes a few extra lines of code to import the `Background`, `BackgroundFill`, `CornerRadii`, and `Insets` features, which we'll use to set the chart background to white.

Next, we declare a simple data class `State` ❶ for holding individual data points during the simulation. Its `step` property represents the number of steps taken since the beginning of the random walk, and `dist` is the cumulative distance traveled after that many steps. We then declare two global parameters: `numStep`, to specify the maximum number of steps per simulation, and `numSim`, to set the maximum number of simulations.

We'll accumulate data in a number of lists ❷, each of size `numStep` and type `State`, as follows:

`xList` Stores the cumulative distance traveled after each step for a particular simulation

`avgList` Stores the arithmetic average (mean) of the cumulative distances traveled across all simulations after each step

`rmsList` Stores the RMS distance calculated across all simulations after each step

`expList` Stores the theoretical (exponential) RMS distance after each step

All these lists are initialized to `(0.0, 0.0)`, meaning all simulations start at step number 0 and position 0. In addition to these lists, we also create two mutable lists, `states1` and `states2`, which we'll use for charting purposes.

Inside the `RandomWalk1D` application class, we use a `VBox` container ❸ to hold the chart objects, as we'll generate two sets of charts that will be placed vertically inside the `VBox`. Notice the additional lines of code for setting the background of the container to white programmatically ❹, without using the cascading style sheets needed for more extensive customizations. We've also introduced the `ScrollPane` feature ❺, which will allow us to scroll the chart window to view the top or the bottom chart, as needed. We can also enlarge the window to make both charts visible at the same time.

After setting up the graphics window, we call three custom functions that will run the simulation and help visualize the results. The first call is to the `randomWalk1d()` function, which simulates `numSim` one-dimensional random walks over `numStep` steps. Here's how it works:

```
fun randomWalk1d() {
    // Create local arrays.
  ❶ val s = Array (numSim) {DoubleArray(numStep)}
    val sumX = DoubleArray(numStep)
    val sumX2 = DoubleArray(numStep)

    // Walk numStep steps numSim times.
    for (i in 0 until numSim) {
        var draw: Int
        var step: Int
        for (j in 1 until numStep) {
          ❷ draw = (0..1).random()
            step = if (draw == 0) -1 else 1
          ❸ s[i][j] = s[i][j-1] + step
            sumX[j] += s[i][j]
            sumX2[j] += (s[i][j] * s[i][j])
            xList[j].step = j.toDouble()
            xList[j].dist = s[i][j]
        }
      ❹ states1.add(xList.map {it.copy()})
    }

    // Create average (mean) and RMS for distances traveled.
    for (j in 0 until numStep) {
        avgList[j].step = j.toDouble()
        avgList[j].dist = sumX[j] / numSim
        rmsList[j].step = j.toDouble()
        rmsList[j].dist = sqrt(sumX2[j] / numSim)
    }
  ❺ states2.addAll(listOf(avgList, rmsList))
}
```

The function body starts by creating three local arrays of type DoubleArray. The first, s, is a two-dimensional array that stores the cumulative distance traveled at each step of each simulation ❶. The others are one-dimensional arrays, sumX and sumX2, to save the running sums of the cumulative distances at each step and the sums of squared distances at each step, respectively. We'll use these values to get the mean and RMS distances.

The random walks are implemented inside a nested for loop. The outer loop controls the number of simulations, and the inner loop makes the particle take numStep steps in succession. During each step, a local variable draw is randomly set to either 0 or 1 with equal likelihood ❷. Based on the outcome, step (referred to as Δx in Figure 4-6) is set to -1 or 1, which is then added to the cumulative distance traveled up to the previous step of the simulation ❸. These cumulative distances are used to create the elements of xList, which is then copied and passed on to states1 once per simulation ❹. Notice how we're reusing the memory allocated for xList during each simulation by overwriting the values of its elements. In the end, states1 has all the data we need to visualize the random walks themselves.

Once we're done with the random walks, we use the resulting lists, sumX and sumX2, to create the avgList and rmsList inside another for loop by dividing the elements of sumX and sumX2 by numSim. Here sumX[j] is the sum of all the elements in column j of the s[i][j] matrix, where i represents the simulation number and j represents the number of steps taken so far. (Likewise, sumX2[j] is the same, squared.) Finally, avgList and rmsList are passed on as elements of states2 ❺, which we defined earlier as a list of lists.

The second function call inside the application class is calcRMS1d(). It generates the theoretical RMS distance at each step:

```
fun calcRMS1d() {
    // Create the theoretical (exponential) rms/list.
    for (j in 0 until numStep) {
        expList[j].step = j.toDouble()
      ❶ expList[j].dist = sqrt(j.toDouble())
    }
    states2.add(expList)
}
```

We know from the theoretical analysis of the one-dimensional random walk problem that the RMS distance is a nonlinear function of the number of steps n, which can be expressed as $x_n = \sqrt{n}$, where x_n is the RMS distance for the nth step (n is equivalent to looping variable `j` in the code). We use this relationship in the `calcRMS1d()` function to calculate the theoretical RMS distances and update `expList` ❶. We'll use this list to create a side-by-side plot of the theoretical and simulated RMS distances to see how closely they follow each other.

In the last two lines of the application class, we make two successive calls to the `createRWChart1()` and `createRWChart2()` functions, shown here:

```
fun createRWChart1(root: VBox) {
    val xyChart1 =
        singleXYChart(states1,
            title = "Random Walk 1D Experiment",
            xLabel = "Steps",
            yLabel = "Cumulative distance traveled")
    root.children.add(xyChart1)
}

fun createRWChart2(root: VBox) {
    val xyChart2 =
        singleXYChart(states2,
            title = "Random Walk 1D Experiment",
            xLabel = "Steps",
            yLabel = "Mean and RMS distance traveled")
    root.children.add(xyChart2)
}
```

Other than the chart labels, the only difference between these two functions is that the first one uses `states1` and the second one uses `states2` to generate the respective charts. Both of these functions call the `singleXYChart()` function (which we discussed and used in Chapter 3) to draw the line charts and stack them inside a scroll pane.

The Result

When you run the full code on your device, you should see a single scrollable window pop up with two separate charts. Let's first consider the visualization of the random walks themselves, shown in Figure 4-7.

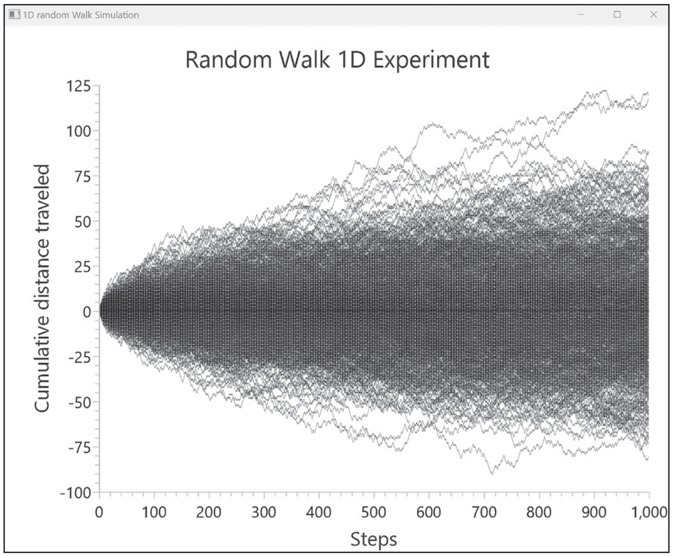


Figure 4-7: The random walk paths from 500 simulations of 1,000 steps

This chart shows all 500 random paths generated by the `randomWalk1d()` function (one per simulation, based on our global `numSim` parameter). These paths show a number of key features of one-dimensional random walks:

- Each random path is unique, evident from the tangled web of lines moving across the chart.
- Most random walks tend to stay close to their starting position, even after many steps. We can see this in the darker band along the x-axis.
- The paths are equally dispersed on both sides of $x = 0$, as expected. You could confirm this by creating histograms of the cumulative distances traveled at different numbers of steps. (I'll leave this for you to try out as an exercise.)
- For any given number of steps, if we add the cumulative distances from all simulations, the sum will be close to zero because positive and negative distances will cancel each other. For the same reason, the arithmetic mean will also be close to zero.
- The RMS distance increases with the number of steps, as confirmed by the gradual widening of the band that envelops all random paths. The RMS distance is therefore a better measure for the average distance traveled than the arithmetic mean, as we don't care about the direction of the movement.

All these points collectively provide the answer we were seeking at the start of this project. A particle moving randomly in one dimension will likely trace a path that will initially stay close to its starting position.

However, if we follow the particle for a long time, it may gradually move farther away. Again, we can't predict exactly how far a particular particle will move, but if we measure the RMS distance from many different particles, we'll see that the RMS distance increases with the number of steps. Our other chart, shown in Figure 4-8, helps us explore this last point further.

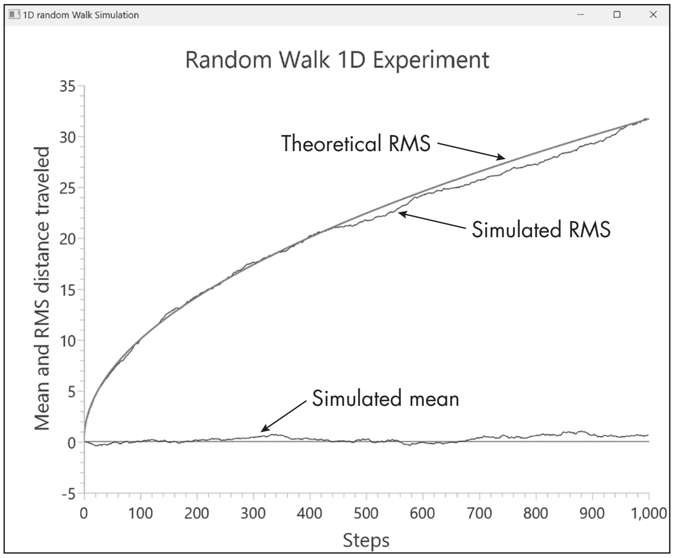


Figure 4-8: The mean and RMS distances traveled for a given number of steps

Figure 4-8 charts three lines. First, we have the line labeled "Simulated mean," generated from the data in `avgList`. This line stays very close to zero, confirming one of the key points made based on Figure 4-7: that the arithmetic average or mean for any number of steps will be zero if we have a sufficiently large number of observations. Second, we have the "Simulated RMS" line, generated from the data in `rmsList`, which clearly shows the RMS distance increasing (although at a decreasing rate) with the number of steps. Third, the smooth "Theoretical RMS" line represents the theoretical RMS distances from `expList`, calculated by simply taking the square root of the number of steps. Again, we can visually confirm that the simulated RMS values are very close to the theoretically expected RMS values.

The minor discrepancies we see between the simulated and theoretical RMS values are to be expected. The simulated RMS values will approach the theoretical values as the number of simulations approaches infinity. I invite you to run the code again, this time setting `numSim` to `5000`. Make sure that you comment out the call to the `createRWChart1()` function before doing that. The default implementation of JavaFX is memory and computation intensive, and trying to plot 5,000 lines, each with 1,000 data points, can take a while depending on your processor and memory configuration. However, if you do this experiment as suggested, you'll see that with the added random walks, the simulated and theoretical RMS lines become virtually the same. If you go further by setting `numSim` to `50000`, you'll see only one line.

EXERCISE

Now that you've seen how to model a random walk in one dimension, try extending your random walk app to model molecular diffusion in two dimensions. The effect should be similar to adding a drop of food coloring to the center of a dish of water and watching the color spread.

Hint: To be able to track distances in 2D, the `State` data class has to be changed to replace `dist` with two values for the x- and y-components—say, `distX` and `distY`. At each step in the random walk, treat these values separately. For the x-component, for example, assume that a particle can either stay still, move toward the positive x-direction, or move toward the negative x-direction. You can code this as `stepX = (-1..1).random()`, meaning `stepX` is equally likely to be `-1`, `0`, or `1`. Repeat the same for the y-component.

Define how large you want the dish to be (set an appropriate diameter), and decide what happens when a particle hits the boundary of the dish. (Perhaps the particle should bounce off the wall.) Update the particle positions inside a nested `for` loop, and update the series for charting using the `ScatterChart` feature of JavaFX. Start with 300 particles. To show particle positions dynamically (which looks really cool), run the random walk function and update the data points as an animation, using `Timeline` and `KeyFrame`, as discussed in Chapter 3.

Summary

In this chapter, we used Kotlin code and custom algorithms to solve math-related problems. The problems weren't just theoretical; they also had practical applications in fields like mathematics, geodesy, navigation, and cryptography. Throughout our journey, we employed various mathematical concepts, operations, and tools, including basic arithmetic, math and trigonometric functions, the Pythagorean theorem, the Fibonacci sequence, the haversine formula, modulo operations, and linear algebra. We also probed the realm of stochastic processes, exploring the generation and utilization of random numbers to simulate random phenomena.

Along the way, we used many core features of Kotlin, such as variables and collections, data classes, and conditional and iterative structures like `if`, `when`, `for`, and `while`. We also discovered the convenience of functions and lambdas, along with the rich set of mathematical and graphics library functions at our disposal.

Resources

Ayars, Eric. "Stochastic Methods." In *Computational Physics with Python*, 131–139. August 18, 2013. Accessed June 15, 2024. *https://belglas.files.wordpress.com/2018/03/cpwp.pdf.*

Dutka, Jacques. "Eratosthenes' Measurement of the Earth Reconsidered." *Archive for History of Exact Sciences* 46, no. 1 (1993): 55–66. Accessed June 15, 2024. *http://www.jstor.org/stable/41134135.*

Eisenberg, Murray. "Hill Ciphers and Modular Linear Algebra." November 3, 1999. Accessed June 15, 2024. *https://apprendre-en-ligne.net/crypto/hill/Hillciph.pdf.*

Harder, Douglas. "Project H.1: Sieve of Eratosthenes." University of Waterloo. Accessed June 15, 2024. *https://ece.uwaterloo.ca/~ece150/Programming_challenges/H/1/.*

Kereki, Federico. "A Modern Look at Square Roots in the Babylonian Way." *Cantor's Paradise*. December 7, 2020. Accessed June 15, 2024. *https://medium.com/cantors-paradise/a-modern-look-at-square-roots-in-the-babylonian-way-ccd48a5e8716.*

"Pythagorean Triples." Prime Glossary. Accessed June 15, 2024. *https://t5k.org/glossary/page.php?sort=PrmPythagTriples.*

Reich, Dan. "The Fibonacci Sequence, Spirals and the Golden Mean." Department of Mathematics, Temple University. Accessed June 15, 2024. *https://math.temple.edu/~reich/Fib/fibo.html.*

Van Brummelen, Glen. *Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry*. Princeton, NJ: Princeton University Press, 2013.