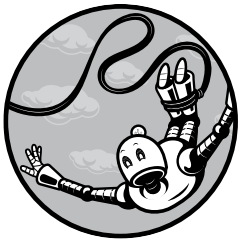
2

ALGORITHMS IN HISTORY



Most people associate algorithms with computers. This is not unreasonable; computer operating systems use many sophisticated algorithms, and programming is well suited to implementing all sorts of algorithms precisely. But algorithms are more fundamental than the computer architecture we implement them on. As mentioned in Chapter 1, the word *algorithm* dates back about a millennium, and algorithms have been described in ancient records going back much further than that. Even outside of written records, there is abundant evidence for the use of complex algorithms in the ancient world—in, for example, their construction methods.

This chapter presents several algorithms of antique provenance. They show great ingenuity and insight, especially considering that they had to be invented and verified without the aid of computers. We start by discussing

Russian peasant multiplication, a method for arithmetic that, despite the name, might be Egyptian and might not actually be associated with peasants. We continue by covering Euclid's algorithm, an important "classic" algorithm for finding greatest common divisors. Finally, we cover an algorithm from Japan that generates magic squares.

Russian Peasant Multiplication

Many people remember learning the multiplication table as a particularly painful part of their education. Young children ask their parents why learning the multiplication table is necessary, and parents usually respond that they can't multiply without knowing it. How wrong they are. *Russian peasant multiplication (RPM)* is a method that enables people to multiply large numbers without knowing most of the multiplication table.

RPM's origins are unclear. An ancient Egyptian scroll called the Rhind papyrus contains a version of this algorithm, and some historians have proposed (mostly unconvincing) conjectures about how the method could have spread from ancient Egyptian scholars to the peasants of the vast Russian hinterlands. Regardless of the details of its history, RPM is an interesting algorithm.

Doing RPM by Hand

Consider the task of multiplying 89 by 18. Russian peasant multiplication proceeds as follows. First, create two columns next to each other. The first column is called the *halving* column and starts with 89. The second column is the *doubling* column and starts with 18 (Table 2-1).

Table 2-1: Halving/Doubling Table, Part 1

Halving	Doubling
89	18

We'll fill out the halving column first. Each row of the halving column takes the previous entry and divides it by 2, ignoring the remainder. For example, 89 divided by 2 is 44 remainder 1, so we write 44 in the second row of the halving column (Table 2-2).

Table 2-2: Halving/Doubling Table, Part 2

Halving	Doubling
89	18
44	

We continue dividing by 2 until we reach 1, dropping the remainder every time and writing the result in the next row. As we continue, we find

that 44 divided by 2 is 22, then half of that is 11, then half of that (dropping the remainder) is 5, then 2, then 1. After writing these in the halving column, we have Table 2-3.

Table 2-3: Halving/Doubling Table, Part 3

Halving	Doubling
89	18
44	
22	
11	
5	
2	
1	

We've completed the halving column. As the name suggests, each entry in the doubling column will be double the previous entry. So since 18 × 2 is 36, 36 is the second entry in the doubling column (Table 2-4).

Table 2-4: Halving/Doubling Table, Part 4

Halving	Doubling
89	18
44	36
22	
11	
5	
2	
1	

We continue to add entries to the doubling column by following the same rule: just double the previous entry. We do this until the doubling column has as many entries as the halving column (Table 2-5).

Table 2-5: Halving/Doubling Table, Part 5

Halving	Doubling
89	18
44	36
22	72
11	144
5	288
2	576
1	1,152

The next step is to cross out or remove every row in which the halving column contains an even number. The result is shown in Table 2-6.

Table 2-6: Halving/Doubling Table, Part 6

Halving	Doubling
89	18
11	144
5	288
1	1,152

The final step is to take the sum of the remaining entries in the doubling column. The result is 18 + 144 + 288 + 1,152 = 1,602. You can check with a calculator that this is correct: 89 × 18 = 1,602. We have accomplished multiplication through halving, doubling, and addition, all without needing to memorize most of the tedious multiplication table that young children so despise.

To see why this method works, try rewriting the doubling column in terms of 18, the number we are trying to multiply (Table 2-7).

Table 2-7: Halving/Doubling Table, Part 7

Halving	Doubling
89	18 × 1
44	18 × 2
22	18 × 4
11	18 × 8
5	18 × 16
2	18 × 32
1	18 × 64

The doubling column is now written in terms of 1, 2, 4, 8, and so on to 64. These are powers of 2, and we can also write them as 2^0, 2^1, 2^2, and so on. When we take our final sum (adding together the doubling rows with odd entries in the halving column), we're really finding this sum:

$$18 \times 2^0 + 18 \times 2^3 + 18 \times 2^4 + 18 \times 2^6 = 18 \times (2^0 + 2^3 + 2^4 + 2^6) = 18 \times 89$$

The fact that RPM works hinges on the fact that

$$(2^0 + 2^3 + 2^4 + 2^6) = 89$$

If you look closely enough at the halving column, you can get a sense for why the preceding equation is true. We can also write this column in terms of powers of 2 (Table 2-8). When we do so, it's easier to start at the lowest entry and work upward. Remember that 2^0 is 1 and 2^1 is 2. In every

row, we multiply by 2^1, and in the rows where the halving number is odd, we also add 2^0. You can see the expression start to resemble our equation more and more as you rise through the rows. By the time we reach the top of the table, we have an expression that simplifies to exactly $2^6 + 2^3 + 2^3 + 2^0$.

Table 2-8: Halving/Doubling Table, Part 8

Halving	Doubling
$(2^5 + 2^3 + 2^2) \times 2^1 + 2^0 = 2^6 + 2^4 + 2^3 + 2^0$	18×2^0
$(2^4 + 2^2 + 2^1) \times 2^1 = 2^5 + 2^3 + 2^2$	18×2^1
$(2^3 + 2^1 + 2^0) \times 2^1 = 2^4 + 2^2 + 2^1$	18×2^2
$(2^2 + 2^0) \times 2^1 + 2^0 = 2^3 + 2^1 + 2^0$	18×2^3
$2^1 \times 2^1 + 2^0 = 2^2 + 2^0$	18×2^4
$2^0 \times 2^1 = 2^1$	18×2^5
2^0	18×2^6

If you number the rows of the halving column starting with the top row as row 0, then 1, 2, and all the way to the bottom row as row 6, you can see that the rows with odd values in the halving column are rows 0, 3, 4, and 6. Now notice the crucial pattern: those row numbers are exactly the exponents in the expression for 89 that we found: $2^6 + 2^4 + 2^3 + 2^0$. This is not a coincidence; the way we constructed the halving column means that the odd entries will always have row numbers that are the exponents in a sum of powers of 2 equaling our original number. When we take a sum of the doubling entries with those indices, we're summing up 18 multiplied by powers of 2 that sum to exactly 89, so we'll get 89×18 as our result.

The reason this works is that really, RPM is an algorithm within an algorithm. The halving column itself is an implementation of an algorithm that finds the sum of powers of 2 that equals the number at the top of the column. This sum of powers of 2 is also called the *binary expansion* of 89. Binary is an alternative way to write numbers using only 0s and 1s, and it has become extremely important in recent decades because computers store information in binary. We can write 89 in binary as 1011001, with 1s in the zeroth, third, fourth, and sixth places (counting from the right), the same as the odd rows of the halving column, and also the same as the exponents in our equation. We can interpret the 1s and 0s in a binary representation as coefficients in a sum of powers of 2. For example, if we write 100, we interpret it in binary as

$$1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

or what we would usually write as 4. If we write 1001, we interpret it in binary as

$$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

or what we would usually write as 9. After running this mini-algorithm to get the binary expansion of 89, we are poised to easily run the full algorithm and complete the multiplication process.

Implementing RPM in Python

It's relatively simple to implement RPM in Python. Let's say that we want to multiply two numbers that we will call n_1 and n_2. First, let's open a Python script and define these variables:

```
n1 = 89
n2 = 18
```

Next, we'll start our halving column. Just as described, the halving column begins with one of the numbers we want to multiply:

```
halving = [n1]
```

The next entry will be `halving[0]/2`, ignoring the remainder. In Python, we can use the `math.floor()` function to accomplish this. This function just takes the closest integer less than a given number. For example, the second row of the halving column can be calculated as follows:

```
import math
print(math.floor(halving[0]/2))
```

If you run this in Python, you'll see that the answer is 44.

We can loop through each row of the halving column, and in each iteration of our loop, we will find the next entry in the halving column in the same way, stopping when we reach 1:

```
while(min(halving) > 1):
    halving.append(math.floor(min(halving)/2))
```

This loop uses the `append()` function for concatenation. At each iteration of the `while` loop, it concatenates the `halving` vector with half of its last value, using the `math.floor()` function to ignore the remainder.

For the doubling column, we can do the same: start with 18, and then continue through a loop. In each iteration of the loop, we'll add double the previous entry to the doubling column, and we'll stop after this column is the same length as the halving column:

```
doubling = [n2]
while(len(doubling) < len(halving)):
    doubling.append(max(doubling) * 2)
```

Finally, let's put these two columns together in a dataframe called `half_double`:

```
import pandas as pd
half_double = pd.DataFrame(zip(halving,doubling))
```

We imported the Python module called `pandas` here. This module enables us to work with tables easily. In this case, we used the `zip` command, which, as suggested by its name, joins `halving` and `doubling` together like a zipper joins two sides of a garment together. The two sets of numbers, `halving` and `doubling`, start as independent lists, and after being zipped together and converted into a `pandas` dataframe, are stored in a table as two aligned columns, as shown in Table 2-5. Since they're aligned and zipped together, we can refer to any row of Table 2-5, such as the third row, and get the full row, including the elements from both `halving` and `doubling` (22 and 72). Being able to refer to and work with these rows will make it easy to remove the rows we don't want, like we did to Table 2-5 to convert it to Table 2-6.

Now we need to remove the rows whose entries in the halving column are even. We can test for evenness using the `%` (modulo) operator in Python, which returns a remainder after division. If a number `x` is odd, then `x%2` will be 1. The following line will keep only the rows of the table whose entry in the halving column is odd:

```
half_double = half_double.loc[half_double[0]%2 == 1,:]
```

In this case, we use the `loc` functionality in the `pandas` module to select only the rows we want. When we use `loc`, we specify which rows and columns we want to select in the square brackets (`[]`) that follow it. Inside the square brackets, we specify which rows and columns we want in order, separated by a comma: the format is [*`row, column`*]. For example, if we wanted the row with index 4 and the column with index 1, we could write `half_double.loc[4,1]`. In this case, we will do more than just specify indices. We will express a logical pattern for which rows we want: we want all rows where `halving` is odd. We specify the `halving` column in our logic with `half_double[0]`, since it's the column with index 0. We specify oddness with `%2 == 1`. Finally, we specify that we want all columns after the comma by writing a colon, which is a shortcut indicating that we want every column.

Finally, we simply take the sum of the remaining doubling entries:

```
answer = sum(half_double.loc[:,1])
```

Here, we are using `loc` again. We specify inside the square brackets that we want every row by using the colon shortcut. We specify that we want `doubling`, the column with index 1, after the comma. Note that the 89 × 18 example we worked through could be done more quickly and easily if we instead calculated 18 × 89—that is, if we put 18 in the halving column and 89 in the doubling column. I encourage you to try this to see the improvement. In general, RPM is faster if the smaller multiplicand is placed in the halving column and the larger one in the doubling column.

To someone who has already memorized the multiplication table, RPM may seem pointless. But besides its historical charm, RPM is worth learning for a few reasons. First, it shows that even something as dry as multiplying numbers can be done in multiple ways and is amenable to creative

approaches. Just because you've learned one algorithm for something doesn't mean that it's the only, or the best, algorithm for the purpose—keep your mind open to new and potentially better ways of doing things.

RPM may be slow, but it requires less memorization up front because it doesn't require knowledge of most of the multiplication table. Sometimes it can be very useful to sacrifice a little speed for the sake of low memory requirements, and this speed/memory tradeoff is an important consideration in many situations where we're designing and implementing algorithms.

Like many of the best algorithms, RPM also brings into focus relationships between apparently disparate ideas. Binary expansions may seem like just a curiosity, of interest to transistor engineers but not useful to a layperson or even a professional programmer. But RPM shows a deep connection between the binary expansion of a number and a convenient way to multiply with only minimal knowledge of the multiplication table. This is another reason to always keep learning: you never know when some apparently useless factoid may form the basis for a powerful algorithm.

Euclid's Algorithm

The ancient Greeks gave many gifts to humanity. One of their greatest was theoretical geometry, which was rigorously compiled by the great Euclid in his 13 books called the *Elements*. Most of Euclid's mathematical writing is in a theorem/proof style, in which a proposition is deduced logically from simpler assumptions. Some of his work is also *constructive*, meaning that it provides a method for using simple tools to draw or create a useful figure, like a square with a particular area or a tangent to a curve. Though the word had not been coined yet, Euclid's constructive methods were algorithms, and some of the ideas behind his algorithms can still be useful today.

Doing Euclid's Algorithm by Hand

Euclid's most famous algorithm is commonly known as *Euclid's algorithm*, though it is only one of many that he wrote about. Euclid's algorithm is a method for finding the greatest common divisor of two numbers. It is simple and elegant and takes only a few lines to implement in Python.

We begin with two natural (whole) numbers: let's call them a and b. Let's say that a is larger than b (if it's not, just rename a to b and rename b to a, and then a will be larger). If we divide a/b, we'll get an integer quotient and an integer remainder. Let's call the quotient q_1, and the remainder c. We can write this as follows:

$$a = q_2 \times b + c$$

For example, if we say that $a = 105$ and $b = 33$, we find that 105/33 is 3, remainder 6. Notice that the remainder c will always be smaller than both a and b—that's how remainders work. The next step of the process is to

forget about a, and focus on b and c. Just like before, we say that b is larger than c. We then find the quotient and remainder when dividing b/c. If we say that b/c is q_2, with remainder d, we can write our result as follows:

$$b = q_2 \times c + d$$

Again, d will be smaller than both b and c, since it's a remainder. If you look at our two equations here, you can start to see a pattern: we're working our way through the alphabet, shifting terms to the left every time. We started with a, b, and c, and then we had b, c, and d. You can see this pattern continue in our next step, in which we divide c/d, and call the quotient q_3 and the remainder e.

$$c = q_2 \times d + e$$

We can continue this process, proceeding as far as we need through the alphabet, until the remainder is equal to zero. Remember that remainders are always smaller than the numbers that were divided to get them, so c is smaller than a and b, d is smaller than b and c, e is smaller than c and d, and so on. This means that at every step, we're working with smaller and smaller integers, so we must eventually get to zero. When we get a zero remainder, we stop the process, and we know that the last nonzero remainder is the greatest common divisor. For example, if we find that e is zero, then d is the greatest common divisor of our original two numbers.

Implementing Euclid's Algorithm in Python

We can implement this algorithm in Python quite easily, as shown in Listing 2-1.

```
def gcd(x,y):
    larger = max(x,y)
    smaller = min(x,y)

    remainder = larger % smaller

    if(remainder == 0):
        return(smaller)

    if(remainder != 0):
❶       return(gcd(smaller,remainder))
```

Listing 2-1: Implementing Euclid's algorithm using recursion

The first thing to notice is that we don't need any of the q_1, q_2, q_3 . . . quotients. We need only the remainders, the successive letters of the alphabet. Remainders are easy to get in Python: we can use the `%` operator from the previous section. We can write a function that takes the remainder after division for any two numbers. If the remainder is zero, then the greatest common divisor is the smaller of the two inputs. If the remainder is not zero, we use the smaller of the two inputs and the remainder as inputs into the same function.

Notice that this function calls itself if the remainder is nonzero ❶. The act of a function calling itself is known as *recursion*. Recursion can seem intimidating or confusing at first; a function that calls itself may seem paradoxical, like a snake that can eat itself or a person trying to fly by pulling on their own bootstraps. But don't be scared. If you're unfamiliar with recursion, one of the best things to do is start with a concrete example, like finding the greatest common divisor of 105 and 33, and follow each step of the code as if you are the computer. You will see that in this example, recursion is just a concise way to express the steps we listed in "Doing Euclid's Algorithm by Hand" on page 20. There is always a danger with recursion that you create an infinite recursion—that a function calls itself, and while calling itself, calls itself again, and nothing ever causes the function to end, so it attempts to call itself endlessly, which is a problem because we need the program to terminate in order to get the final answer. In this case, we can feel safe because at each step we are getting smaller and smaller remainders that will eventually go down to zero and enable us to exit the function.

Euclid's algorithm is short and sweet and useful. I encourage you to create an even more concise implementation of it in Python.

Japanese Magic Squares

The history of Japanese mathematics is particularly fascinating. In *A History of Japanese Mathematics*, originally published in 1914, the historians David Eugene Smith and Yoshio Mikami wrote that Japanese math had historically possessed a "genius for taking infinite pains" and "ingenuity in untangling minute knots and thousands of them." On the one hand, mathematics uncovers absolute truths that should not vary between times and cultures. On the other hand, the types of problems that distinct groups tend to focus on and their idiosyncratic approaches to them, not to mention differences in notation and communication, provide great scope for noteworthy cultural differences, even in a field as austere as math.

Creating the Luo Shu Square in Python

Japanese mathematicians had a fondness for geometry, and many of their ancient manuscripts pose and solve problems related to finding the areas of exotic shapes like circles inscribed within ellipses and Japanese hand fans. Another steady area of focus for Japanese mathematicians throughout several centuries was the study of magic squares.

A *magic square* is an array of unique, consecutive natural numbers such that all rows, all columns, and both of the main diagonals have the same sum. Magic squares can be any size. Table 2-9 shows an example of a 3×3 magic square.

Table 2-9: The Luo Shu Square

4	9	2
3	5	7
8	1	6

In this square, each row, each column, and both main diagonals sum to 15. This is more than just a random example—it's the famous *Luo Shu square*. According to an ancient Chinese legend, this magic square was first seen inscribed on the back of a magical turtle who came out of a river in response to the prayers and sacrifices of a suffering people. In addition to the definitional pattern that each row, column, and diagonal sums to 15, there are a few other patterns. For example, the outer ring of numbers alternates between even and odd numbers, and the consecutive numbers 4, 5, and 6 appear in the main diagonal.

The legend of the sudden appearance of this simple but fascinating square as a gift from the gods is fitting for the study of algorithms. Algorithms are often easy to verify and use, but they can be difficult to design from scratch. Especially elegant algorithms, when we have the good luck to invent one, seem revelatory, as if they have come out of nowhere as a gift from the gods inscribed on the back of a magical turtle. If you doubt this, try to create an 11×11 magic square from scratch, or try to discover a general-purpose algorithm for generating new magic squares.

Knowledge of this and other magic squares apparently passed from China to Japan at least as early as 1673, when a mathematician named Sanenobu published a 20×20 magic square in Japan. We can create the Luo Shu square in Python with the following command:

```
luoshu = [[4,9,2],[3,5,7],[8,1,6]]
```

It will come in handy to have a function that verifies whether a given matrix is a magic square. The following function does this by verifying the sums across all rows, columns, and diagonals and then checking whether they are all the same:

```
def verifysquare(square):
    sums = []
    rowsums = [sum(square[i]) for i in range(0,len(square))]
    sums.append(rowsums)
    colsums = [sum([row[i] for row in square]) for i in range(0,len(square))]
    sums.append(colsums)
    maindiag = sum([square[i][i] for i in range(0,len(square))])
    sums.append([maindiag])
    antidiag = sum([square[i][len(square) - 1 - i] for i in \
range(0,len(square))])
    sums.append([antidiag])
    flattened = [j for i in sums for j in i]
    return(len(list(set(flattened))) == 1)
```

Implementing Kurushima's Algorithm in Python

In the previous sections, we discussed how to perform our algorithms of interest "by hand" before providing details of the implementation of the code. In the case of Kurushima's algorithm, we'll outline the steps and introduce the code simultaneously. The reason for this change is the relative complexity of the algorithm, and especially the length of the code required to implement it.

One of the most elegant algorithms for generating magic squares, *Kurushima's algorithm* is named for Kurushima Yoshita, who lived during the Edo period. Kurushima's algorithm works only for magic squares of *odd dimension*, meaning that it works for any $n \times n$ square if n is an odd number. It begins by filling out the center of the square in a way that matches the Luo Shu square. In particular, the central five squares are given by the following expressions, with n here referring to the dimension of the square (Table 2-10).

Table 2-10: The Center of Kurushima's Square

	n^2	
n	$(n^2 + 1)/2$	$n^2 + 1 - n$
	1	

Kurushima's algorithm for generating an $n \times n$ magic square for odd n can be described simply as follows:

1. Fill in the five central squares according to Table 2-10.
2. Beginning with any entry whose value is known, determine the value of an unknown neighboring entry by following one of the three rules (described next).
3. Repeat step 2 until every entry in the full magic square is filled in.

Filling in the Central Squares

We can begin the process of creating a magic square by creating an empty square matrix that we'll fill up. For example, if we want to create a 7×7 matrix, we can define `n=7` and then create a matrix with `n` rows and `n` columns:

```
n = 7
square = [[float('nan') for i in range(0,n)] for j in range(0,n)]
```

In this case, we don't know what numbers to put in the square, so we fill it entirely with entries equal to `float('nan')`. Here, `nan` stands for *not a number*, which we can use as a placeholder in Python when we want to fill up a list before we know what numbers to use. If we run `print(square)`, we find that this matrix by default is filled with `nan` entries:

```
[[nan, nan, nan, nan, nan, nan, nan], [nan, nan, nan, nan, nan, nan, nan],
[nan, nan, nan, nan, nan, nan, nan], [nan, nan, nan, nan, nan, nan, nan],
[nan, nan, nan, nan, nan, nan, nan], [nan, nan, nan, nan, nan, nan, nan],
[nan, nan, nan, nan, nan, nan, nan]]
```

This square is not too pretty as it is output in the Python console, so we can write a function that will print it in a more readable way:

```
def printsquare(square):
    labels = ['['+str(x)+']' for x in range(0,len(square))]
    format_row = "{:>6}" * (len(labels) + 1)
    print(format_row.format("", *labels))
    for label, row in zip(labels, square):
        print(format_row.format(label, *row))
```

Don't worry about the details of the `printsquare()` function, since it's only for pretty printing and not part of our algorithm. We can fill in the central five squares with simple commands. First, we can get the indices of the central entry as follows:

```
import math
center_i = math.floor(n/2)
center_j = math.floor(n/2)
```

The central five squares can be populated according to the expressions in Table 2-10 as follows:

```
square[center_i][center_j] = int((n**2 +1)/2)
square[center_i + 1][center_j] = 1
square[center_i - 1][center_j] = n**2
square[center_i][center_j + 1] = n**2 + 1 - n
square[center_i][center_j - 1] = n
```

Specifying the Three Rules

The purpose of Kurushima's algorithm is to fill in the rest of the `nan` entries according to simple rules. We can specify three simple rules that enable us to fill out every other entry, no matter how big the magic square is. The first rule is expressed in Figure 2-1.

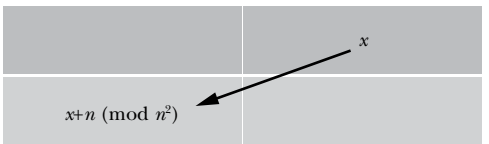


Figure 2-1: Rule 1 of Kurushima's algorithm

So for any x in the magic square, we can determine the entry that is situated in this diagonal relationship to x by simply adding n and taking the result mod n^2 (mod refers to the modulo operation). Of course, we can also go in the opposite direction by reversing the operation: subtracting n and taking the result mod n^2.

The second rule is even simpler, and is expressed in Figure 2-2.

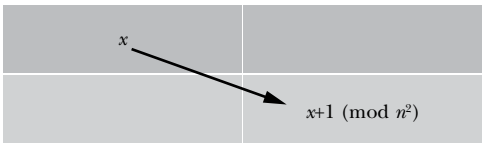


Figure 2-2: Rule 2 of Kurushima's algorithm

For any x in the magic square, the entry below and to the right of x is 1 greater than x, mod n^2. This is a simple rule, but it has one important exception: this rule is not followed when we cross from the upper-left half of the magic square to the lower-right half of the square. Another way to say this is that we do not follow the second rule if we are crossing the magic square's *antidiagonal*, the bottom-left-to-top-right line shown in Figure 2-3.

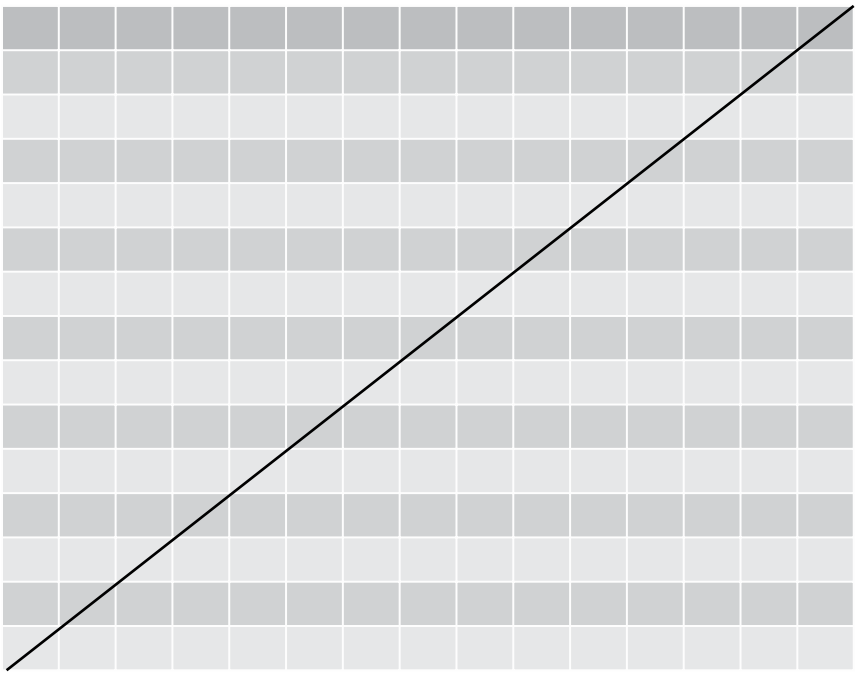


Figure 2-3: The antidiagonal of a square matrix

You can see the cells that are on the antidiagonal. The antidiagonal line passes fully through them. We can follow our normal two rules when we are dealing with these cells. We need the exceptional third rule only when starting in a cell that is fully above the antidiagonal and crossing to a cell that is fully below it, or vice versa. That final rule is expressed in Figure 2-4, which shows an antidiagonal and two cells that would need to follow this rule when crossing it.

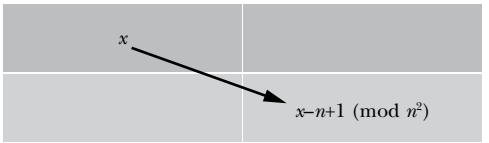


Figure 2-4: Rule 3 of Kurushima's algorithm

This rule is followed when we are crossing the antidiagonal. If we cross from the bottom right to the top left, we can follow the inverse of this rule, in which x is transformed to $x + n - 1$, mod n^2.

We can write a simple implementation of Rule 1 in Python by defining a function that takes `x` and `n` as its arguments and returns `(x+n)%n**2`:

```
def rule1(x,n):
    return((x + n)%n**2)
```

We can try this out with the central entry in the Luo Shu square. Remember, the Luo Shu square is a 3×3 square matrix, so $n = 3$. The central entry of the Luo Shu square is 5. The entry below and to the left of this entry is 8, and if we have implemented our `rule1()` function correctly we'll get an 8 when we run the following line:

```
print(rule1(5,3))
```

You should see an `8` in the Python console. Our `rule1()` function seems to work as intended. However, we could improve it by enabling it to go "in reverse," determining not only the entry on the bottom left of a given entry, but also the entry to the top right (that is, being able to go from 8 to 5 in addition to going from 5 to 8). We can make this improvement by adding one more argument to the function. We'll call our new argument `upright`, and it will be a `True`/`False` indicator of whether we're looking for the entry up and to the right of `x`. If not, we will by default look for the entry to the bottom left of `x`:

```
def rule1(x,n,upright):
    return((x + ((-1)**upright) * n)%n**2)
```

In a mathematical expression, Python will interpret `True` as 1 and `False` as 0. If `upright` is `False`, our function will return the same value as before, since $(-1)^0 = 1$. If `upright` is `True`, then it will subtract `n` instead of adding `n`, which will enable us to go in the other direction. Let's check whether it can determine the entry above and to the right of 1 in the Luo Shu square:

```
print(rule1(1,3,True))
```

It should print `7`, the correct value in the Luo Shu square.

For Rule 2, we can create an analogous function. Our Rule 2 function will take `x` and `n` as arguments, just like Rule 1. But Rule 2 is by default finding the entry below and to the right of `x`. So we will add an `upleft` argument that will be `True` if we want to reverse the rule. The final rule is as follows:

```
def rule2(x,n,upleft):
    return((x + ((-1)**upleft))%n**2)
```

You can test this on the Luo Shu square, though there are only two pairs of entries for which this doesn't run into the exception to Rule 2. For this exception, we can write the following function:

```
def rule3(x,n,upleft):
    return((x + ((-1)**upleft * (-n + 1)))%n**2)
```

This rule needs to be followed only when we're crossing the magic square's antidiagonal. We'll see later how to determine whether or not we are crossing the antidiagonal.

Now that we know how to fill the five central squares, and we have a rule to fill out the remaining squares based on knowledge of those central squares, we can fill out the rest of the square.

Filling in the Rest of the Square

One way to fill in the rest of the square is to "walk" randomly through it, using known entries to fill in unknown entries. First, we'll determine the indices of our central entry as follows:

```
center_i = math.floor(n/2)
center_j = math.floor(n/2)
```

Then, we can randomly select a direction to "walk," as follows:

```
import random
entry_i = center_i
entry_j = center_j
where_we_can_go = ['up_left','up_right','down_left','down_right']
where_to_go = random.choice(where_we_can_go)
```

Here, we've used Python's `random.choice()` function, which does random selection from lists. It takes an element from the set we specified (`where_we_can_go`), but it chooses at random (or as close to random as it can get).

After we've decided a direction to travel, we can follow whichever rule corresponds to our direction of travel. If we have chosen to go `down_left` or `up_right`, we'll follow Rule 1, choosing the right arguments and indices as follows:

```
if(where_to_go == 'up_right'):
    new_entry_i = entry_i - 1
    new_entry_j = entry_j + 1
    square[new_entry_i][new_entry_j] = rule1(square[entry_i][entry_j],n,True)
```

```
if(where_to_go == 'down_left'):
    new_entry_i = entry_i + 1
    new_entry_j = entry_j - 1
    square[new_entry_i][new_entry_j] = rule1(square[entry_i][entry_j],n,False)
```

Similarly, we'll follow Rule 2 if we have chosen to travel `up_left` or `down_right`:

```
if(where_to_go == 'up_left'):
    new_entry_i = entry_i - 1
    new_entry_j = entry_j - 1
    square[new_entry_i][new_entry_j] = rule2(square[entry_i][entry_j],n,True)

if(where_to_go == 'down_right'):
    new_entry_i = entry_i + 1
    new_entry_j = entry_j + 1
    square[new_entry_i][new_entry_j] = rule2(square[entry_i][entry_j],n,False)
```

This code is for going up-left and down-right, but we should follow it only if we're not crossing the antidiagonal. We'll have to make sure that we follow Rule 3 in the case where we are crossing the antidiagonal. There is a simple way to know if we are in an entry that is near the antidiagonal: the entries just above the antidiagonal will have indices that sum to `n-2`, and the entries just below the antidiagonal will have indices that sum to `n`. We'll want to implement Rule 3 in these exceptional cases:

```
if(where_to_go == 'up_left' and (entry_i + entry_j) == (n)):
    new_entry_i = entry_i - 1
    new_entry_j = entry_j - 1
    square[new_entry_i][new_entry_j] = rule3(square[entry_i][entry_j],n,True)


if(where_to_go == 'down_right' and (entry_i + entry_j) == (n-2)):
    new_entry_i = entry_i + 1
    new_entry_j = entry_j + 1
    square[new_entry_i][new_entry_j] = rule3(square[entry_i][entry_j],n,False)
```

Keep in mind that our magic square is finite, so we cannot, for example, travel up/left from the top row or leftmost column. By creating our list of where it's possible to travel based on our current location, we can add some simple logic to ensure that we travel only in allowed directions:

```
where_we_can_go = []

if(entry_i < (n - 1) and entry_j < (n - 1)):
    where_we_can_go.append('down_right')

if(entry_i < (n - 1) and entry_j > 0):
    where_we_can_go.append('down_left')
```

```
    if(entry_i > 0 and entry_j < (n - 1)):
        where_we_can_go.append('up_right')

    if(entry_i > 0 and entry_j > 0):
        where_we_can_go.append('up_left')
```

We have all the elements we need to write Python code that implements Kurushima's algorithm.

Putting It All Together

We can put everything together in a function that takes a starting square with some `nan` entries and travels through it using our three rules to fill them in. Listing 2-2 contains the whole function.

```
import random
def fillsquare(square,entry_i,entry_j,howfull):
     while(sum(math.isnan(i) for row in square for i in row) > howfull):
        where_we_can_go = []

        if(entry_i < (n - 1) and entry_j < (n - 1)):
            where_we_can_go.append('down_right')
        if(entry_i < (n - 1) and entry_j > 0):
            where_we_can_go.append('down_left')
        if(entry_i > 0 and entry_j < (n - 1)):
            where_we_can_go.append('up_right')
        if(entry_i > 0 and entry_j > 0):
            where_we_can_go.append('up_left')

        where_to_go = random.choice(where_we_can_go)
        if(where_to_go == 'up_right'):
            new_entry_i = entry_i - 1
            new_entry_j = entry_j + 1
            square[new_entry_i][new_entry_j] = rule1(square[entry_i][entry_j],n,True)

        if(where_to_go == 'down_left'):
            new_entry_i = entry_i + 1
            new_entry_j = entry_j - 1
            square[new_entry_i][new_entry_j] = rule1(square[entry_i][entry_j],n,False)

        if(where_to_go == 'up_left' and (entry_i + entry_j) != (n)):
            new_entry_i = entry_i - 1
            new_entry_j = entry_j - 1
            square[new_entry_i][new_entry_j] = rule2(square[entry_i][entry_j],n,True)

        if(where_to_go == 'down_right' and (entry_i + entry_j) != (n-2)):
            new_entry_i = entry_i + 1
            new_entry_j = entry_j + 1
            square[new_entry_i][new_entry_j] = rule2(square[entry_i][entry_j],n,False)

        if(where_to_go == 'up_left' and (entry_i + entry_j) == (n)):
            new_entry_i = entry_i - 1
            new_entry_j = entry_j - 1
            square[new_entry_i][new_entry_j] = rule3(square[entry_i][entry_j],n,True)
```

```
        if(where_to_go == 'down_right' and (entry_i + entry_j) == (n-2)):
            new_entry_i = entry_i + 1
            new_entry_j = entry_j + 1
            square[new_entry_i][new_entry_j] = rule3(square[entry_i][entry_j],n,False)

    ❶ entry_i = new_entry_i
      entry_j = new_entry_j

    return(square)
```

Listing 2-2: A function that enables an implementation of Kurushima's algorithm

This function will take four arguments: first, a starting square that has some `nan` entries; second and third, the indices of the entry that we want to start with; and fourth, how much we want to fill up the square (measured by the number of `nan` entries we are willing to tolerate). The function consists of a `while` loop that writes a number to an entry in the square at every iteration by following one of our three rules. It continues until it has as many `nan` entries as we have specified in the function's fourth argument. After it writes to a particular entry, it "travels" to that entry by changing its indices ❶, and then it repeats again.

Now that we have this function, all that remains is to call it in the right way.

Using the Right Arguments

Let's start with the central entry and fill up the magic square from there. For our `howfull` argument, we'll specify `(n**2)/2-4`. The reason for using this value for `howfull` will become clear after we see our results:

```
entry_i = math.floor(n/2)
entry_j = math.floor(n/2)

square = fillsquare(square,entry_i,entry_j,(n**2)/2 - 4)
```

In this case, we call the `fillsquare()` function using the existing `square` variable that we defined previously. Remember we defined it to be full of `nan` entries except for five central elements that we specified. After we run the `fillsquare()` function with that `square` as its input, the `fillsquare()` function fills in many of the remaining entries. Let's print out the resulting square and see what it looks like afterward:

```
printsquare(square)
```

The result is as follows:

```
      [0]   [1]   [2]   [3]   [4]   [5]   [6]
[0]    22   nan    16   nan    10   nan     4
[1]   nan    23   nan    17   nan    11   nan
[2]    30   nan    24    49    18   nan    12
[3]   nan    31     7    25    43    19   nan
[4]    38   nan    32     1    26   nan    20
[5]   nan    39   nan    33   nan    27   nan
[6]    46   nan    40   nan    34   nan    28
```

You'll notice that the nans occupy alternating entries, like a checkerboard. The reason for this is that the rules we have for moving diagonally give us access to only about half of the total entries, depending on which entry we started with. The valid moves are the same as in checkers: a piece that starts on a dark square can move diagonally to other dark squares, but its diagonal moving pattern will never allow it to move to any of the light squares. The nan entries we see are inaccessible if we start on the central entry. We specified `(n**2)/2 - 4` for our `howfull` argument instead of zero because we know that we wouldn't be able to fill the matrix completely by calling our function only once. But if we start again on one of the central entry's neighbors, we will be able to access the rest of the nan entries in our "checkerboard." Let's call the `fillsquare()` function again, this time starting on a different entry and specifying our fourth argument as zero, indicating that we want to completely fill our square:

```
entry_i = math.floor(n/2) + 1
entry_j = math.floor(n/2)

square = fillsquare(square,entry_i,entry_j,0)
```

If we print our square now, we can see that it is completely full:

```
>>> printsquare(square)
        [0]   [1]   [2]   [3]   [4]   [5]   [6]
  [0]    22    47    16    41    10    35     4
  [1]     5    23    48    17    42    11    29
  [2]    30     6    24     0    18    36    12
  [3]    13    31     7    25    43    19    37
  [4]    38    14    32     1    26    44    20
  [5]    21    39     8    33     2    27    45
  [6]    46    15    40     9    34     3    28
```

There is just one final change we need to make. Because of the rules of the % operator, our square contains consecutive integers between 0 and 48, but Kurushima's algorithm is meant to fill our square with the integers from 1 to 49. We can add one line that replaces 0 with 49 in our square:

```
square=[[n**2 if x == 0 else x for x in row] for row in square]
```

Now our square is complete. We can verify that it is indeed a magic square by using the `verifysquare()` function we created earlier:

```
verifysquare(square)
```

This should return `True`, indicating that we've succeeded.

We just created a 7×7 magic square by following Kurushima's algorithm. Let's test our code and see if it can create a larger magic square. If we change `n` to 11 or any other odd number, we can run exactly the same code and get a magic square of any size:

```
n = 11
square=[[float('nan') for i in range(0,n)] for j in range(0,n)]


center_i = math.floor(n/2)
center_j = math.floor(n/2)

square[center_i][center_j] = int((n**2 + 1)/2)
square[center_i + 1][center_j] = 1
square[center_i - 1][center_j] = n**2
square[center_i][center_j + 1] = n**2 + 1 - n
square[center_i][center_j - 1] = n


entry_i = center_i
entry_j = center_j

square = fillsquare(square,entry_i,entry_j,(n**2)/2 - 4)

entry_i = math.floor(n/2) + 1
entry_j = math.floor(n/2)

square = fillsquare(square,entry_i,entry_j,0)

square = [[n**2 if x == 0 else x for x in row] for row in square]
```

Our 11×11 square looks as follows:

```
>>> printsquare(square)
        [0]   [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]   [9]  [10]
  [0]    56   117    46   107    36    97    26    87    16    77     6
  [1]     7    57   118    47   108    37    98    27    88    17    67
  [2]    68     8    58   119    48   109    38    99    28    78    18
  [3]    19    69     9    59   120    49   110    39    89    29    79
  [4]    80    20    70    10    60   121    50   100    40    90    30
  [5]    31    81    21    71    11    61   111    51   101    41    91
  [6]    92    32    82    22    72     1    62   112    52   102    42
  [7]    43    93    33    83    12    73     2    63   113    53   103
  [8]   104    44    94    23    84    13    74     3    64   114    54
  [9]    55   105    34    95    24    85    14    75     4    65   115
 [10]   116    45   106    35    96    25    86    15    76     5    66
```

We can verify, either manually or with our `verifysquare()` function, that this is indeed a magic square. You can do the same with any odd `n` and marvel at the results.

Magic squares don't have much practical significance, but it's fun to observe their patterns anyway. If you're interested, you might spend some time thinking about the following questions:

- Do the larger magic squares we created follow the odd/even alternating pattern seen in the outer edge of the Luo Shu square? Do you think every possible magic square follows this pattern? What reason, if any, would there be for this pattern?
- Do you see any other patterns in the magic squares we've created that haven't been mentioned yet?
- Can you find another set of rules that create Kurushima's squares? For example, are there rules that enable one to travel up and down through Kurushima's square instead of diagonally?
- Are there other types of magic squares that satisfy the definition of a magic square but don't follow Kurushima's rules at all?
- Is there a more efficient way to write code to implement Kurushima's algorithm?

Magic squares occupied the attention of great Japanese mathematicians for several centuries, and they've found a significant place in cultures around the world. We can count ourselves lucky that the great mathematicians of the past gave us algorithms for generating and analyzing magic squares that we can easily implement on today's powerful computers. At the same time, we can admire the patience and insight that was required for them to investigate magic squares with only pen, paper, and their wits (and the occasional magical turtle) to guide them.

Summary

In this chapter, we discussed some historical algorithms that range from a few centuries to a few millenia old. Readers who are interested in historical algorithms can find many more to study. These algorithms may not be of great practical utility today, but it can be worthwhile to study them—first because they give us a sense of history, and second because they help broaden our horizons and may provide the inspiration for writing our own innovative algorithms.

The algorithms in the next chapter enable us to do some commonly needed and useful tasks with mathematical functions: maximize and minimize them. Now that we have discussed algorithms in general and algorithms in history, you should be comfortable with what an algorithm is and how one works, and you should be ready to dive into serious algorithms used in the most cutting-edge software being developed today.