Algorithms Revisited

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were lazy, you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write n - 1 as the first digit and 10 - n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

On the other hand, understanding that hard problems can be solved by step-by-step algorithmic processess is one of the major simplifying breakthroughs that has had enormous benefits. So while the execution of the algorithm may be boring and may require no intelligence, algorithmic or computational thinking is having a vast impact. It is the process of designing algorithms that is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of a step-by-step mechanical algorithm.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```
1
2
```

ActiveCode: 1 (scratch_07_03)

Run

Simple Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, *"This is great! We can use the computers to generate the tables, so there will be no errors."* That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.

Although a power of 2 table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
1 print("n",'\t',"2**n")     #table column headings
2 print("---",'\t',"-----")
3
4 for x in range(13):        # generate values for columns
5     print(x, '\t', 2**x)
6
```

ActiveCode: 2 (ch07_table1)

Run

```
n        2**n
---      -----
0        1
1        2
2        4
3        8
4        16
5        32
6        64
7        128
8        256
9        512
10       1024
11       2048
12       4096
```

The string `'\t'` represents a **tab character**. The backslash character in `'\t'` indicates the beginning of an **escape sequence**. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence `\n` represents a **newline**.

An escape sequence can appear anywhere in a string. In this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the **cursor** keeps track of where the next character will go. After a `print` function is executed, the cursor normally goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program. Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Check your understanding

itr-6: What is the difference between a tab (t) and a sequence of spaces?

a) A tab will line up items in a second column, regardless of how many characters were in the first column, while spaces will not.

b) There is no difference

c) A tab is wider than a sequence of spaces

d) You must use tabs for creating tables. You cannot use spaces.

Check Me Compare Me

Correct!! Assuming the size of the first column is less than the size of the tab width.

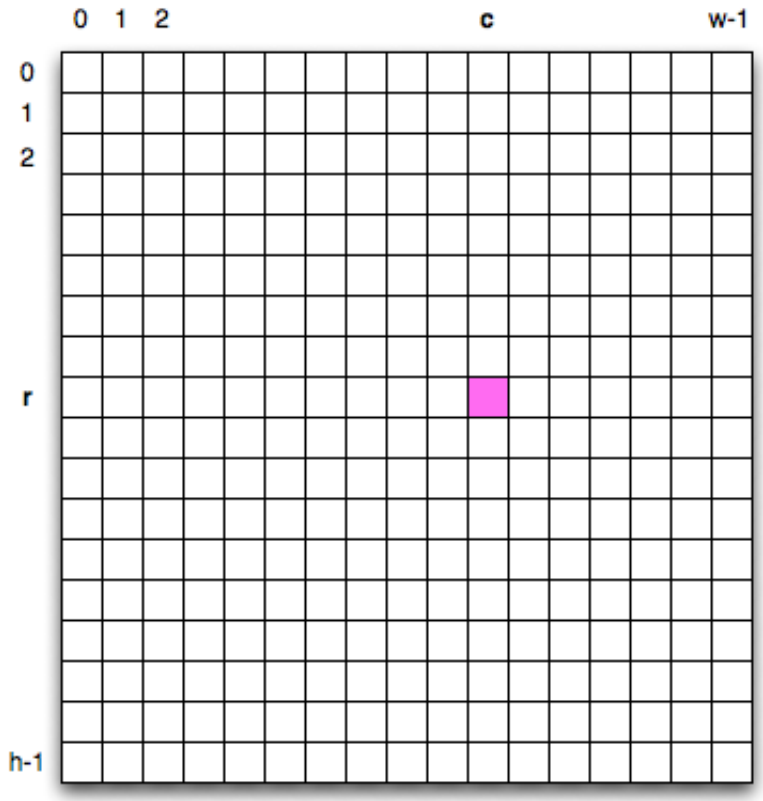
2-Dimensional Iteration: Image Processing

Two dimensional tables have both rows and columns. You have probably seen many tables like this if you have used a spreadsheet program. Another object that is organized in rows and columns is a digital image. In this section we will explore how iteration allows us to manipulate these images.

A **digital image** is a finite collection of small, discrete picture elements called **pixels**. These pixels are organized in a two-dimensional grid. Each pixel represents the smallest amount of picture information that is available. Sometimes these pixels appear as small “dots”.

Each image (grid of pixels) has its own width and its own height. The width is the number of columns and the height is the number of rows. We can name the pixels in the grid by using the column number and row number. However, it is very important to remember that computer scientists like to start counting with 0! This means that if there are 20 rows, they will be named 0,1,2, and so on thru 19. This will be very useful later when we iterate using range.

In the figure below, the pixel of interest is found at column **c** and row **r**.



The RGB Color Model

Each pixel of the image will represent a single color. The specific color depends on a formula that mixes various amounts of three basic colors: red, green, and blue. This technique for creating color is known as the **RGB Color Model**. The amount of each color, sometimes called the **intensity** of the color, allows us to have very fine control over the resulting color.

The minimum intensity value for a basic color is 0. For example if the red intensity is 0, then there is no red in the pixel. The maximum intensity is 255. This means that there are actually 256 different amounts of intensity for each basic color. Since there are three basic colors, that means that you can create 256^3 distinct colors using the RGB Color Model.

Here are the red, green and blue intensities for some common colors. Note that “Black” is represented by a pixel having no basic color. On the other hand, “White” has maximum values for all three basic color components.

Color	Red	Green	Blue
Red	255	0	0
Green	0	255	0
Blue	0	0	255
White	255	255	255
Black	0	0	0
Yellow	255	255	0
Magenta	255	0	255

In order to manipulate an image, we need to be able to access individual pixels. This capability is provided by a module called **image**. The image module defines two classes: `Image` and `Pixel`.

Each Pixel object has three attributes: the red intensity, the green intensity, and the blue intensity. A pixel provides three methods that allow us to ask for the intensity values. They are called `getRed`, `getGreen`, and `getBlue`. In addition, we can ask a pixel to change an intensity value using its `setRed`, `setGreen`, and `setBlue` methods.

Method Name	Example	Explanation
Pixel(r,g,b)	Pixel(20,100,50)	Create a new pixel with 20 red, 100 green, and 50 blue.
getRed()	r = p.getRed()	Return the red component intensity.
getGreen()	r = p.getGreen()	Return the green component intensity.
getBlue()	r = p.getBlue()	Return the blue component intensity.
setRed()	p.setRed(100)	Set the red component intensity to 100.
setGreen()	p.setGreen(45)	Set the green component intensity to 45.
setBlue()	p.setBlue(156)	Set the blue component intensity to 156.

In the example below, we first create a pixel with 45 units of red, 76 units of green, and 200 units of blue. We then print the current amount of red, change the amount of red, and finally, set the amount of blue to be the same as the current amount of green.

```
1 import image
2
3 p = image.Pixel(45,76,200)
4 print(p.getRed())
5 p.setRed(66)
6 print(p.getRed())
7 p.setBlue(p.getGreen())
8 print(p.getGreen(), p.getBlue())
9
```

ActiveCode: 3 (pixelex1a)

Run

```
45
66
76 76
```

Check your understanding

itr-7: If you have a pixel whose RGB value is (50, 0, 0), what color will this pixel appear to be?

- (•) a) Dark red
- ( ) b) Light red
- ( ) c) Dark green
- ( ) d) Light green

Check Me | Compare Me

Correct!! Because all three values are close to 0, the color will be dark. But because the red value is higher than the other two, the color will appear red.