Chapter 5 Algorithms with Perl

Learning how to write small computer programs will open a new world. If you try hard enough small programs will become full applications multiple megabytes in size. There is no real impediment to increasing application size. However, if your program has to perform a lot of calculations and if your customer or boss demands it runs as fast as possible you will need to know about *algorithms*. So what are algorithms? To some any program performs a task and automatically becomes an algorithm. Others consider tasks that can be explained mathematically to be a true algorithm. They need to be self-contained and follow operations step-by-step. Computer Scientists have over several decades discovered the best way to perform actions such as sorting lists, data processing or artificial learning.

In this chapter I will explain several important algorithms through the use of Perl, a general purpose language that is also often used as a scripting language like Python and Shell. Perl was first released in 1987 and was designed by Larry Wall and gained popularity because of its easy ability to parse texts and use regular expressions. Perl can at times be difficult to understand. With simple programs such as text parsers it is not more difficult than C or Python, but with large programs it becomes much harder to read and as such it has been referred to as *write-only language*.

Perl 5 and Perl 6

Despite the power of the Perl language its use has sadly declined. The proliferation of new languages and the continuing popularity of Python, Java and C have not done old-school languages such as Perl and Pascal any favors. What has also done irreparable harm is the divide that was created with the introduction of Perl 6 in 2000. Perl 5 development has not stopped and so the language was effectively forked. Despite Perl 6 being specified in 2000 version 1.0 has still to be released! For our examples we will be using Perl 5. If you are not certain what language version you have installed in Linux just type in

perl –v

into the command line. Chances are it is Perl 5.

Perl is an interpreted language just like Python. Programs written in Perl have the extension '.pl'. As a first simple example type in the following lines in a file called HelloWorld.pl

```
use strict;
use warnings;
print "Hello World!\n";
```

You can run the program with the command

Perl HelloWorld.pl

If you omit the first two lines the program will still print *Hello World!* Using *strict* and *warnings* makes for easy debugging. Perl is unusual in its choice of variables. Just like PHP it is weakly typed. The

language does not explicitly use casting of variables. So a string can be as easily interpreted as an integer or a float. Unlike Java, which requires the user to convert a String to an int with the Integer.parseInt() function. Perl has only three types of variables; scalars ($), arrays (@) and hashes (#). Roughly speaking they correspond to simple variables, arrays, and dictionaries. Simple variable names start with a dollar sign $. So to store the string *Hello, world!* you type in the following:

```
$sentence = "Hello, world!";
```

Input from the Terminal is collected with standard input or STDIN. The program will stop until a user types something and presses Enter.

```
$anothersentence = <STDIN>;
print "You typed in $anothersentence";
```

Programs can also read multiple lines by changing variable signal $ with @, an array.

```
@anothersentence = <STDIN>;
print "You type in the following sentences @anothersentence";
```

Now pressing Enter will no longer suffice in closing standard input. Instead, press Control-D. If you typed in multiple lines you will see each of them as output. Of course Perl also has conditionals and for loops.

```
use strict;
use warnings;

print "Enter number\n";
my $num = <STDIN>;

hello($num);

sub hello {
    for my $i (0 .. 10) {
        if($i > $num) {
            print "Number is larger than $num\n";
        } else {
            print "Number is smaller than $num\n";
        }
    }
}
```

Sorting algorithms

Sorting algorithms area amongst the most popular any programmer must know. Chances are most projects will require a sorting algorithm. With luck you can find a library to make use of an optimized sorting algorithm. However, this may not always be available or the data you are working with may be atypical. In essence a sorting algorithm will sort a list or array of values (integer, float, string etc.) in a predefined order (size or alphabetical order). For smaller applications the speed of an algorithm is not relevant. If it takes a less efficient algorithm 0.2 seconds to sort a list of a thousand bank numbers than there is little real incentive to implement an efficient algorithm that can do it in 0.1 seconds. But if you

have lists of a million records, or there is a lot of data processing than you should be able to implement the best sorting algorithm there is. For this tutorial we will start with Bubble sort, the simplest but usually also the slowest algorithm. However, first we need to define a benchmark for algorithm speed, also known as the Big O notation. Sorting algorithms can be defined by an upper bound and a lower bound to the amount of time it takes to perform their task. Generally speaking it will take an algorithm *n* steps to find an element in array if the array is sorted and the element is located at the n^{th} place. The time it takes is known as linear time, or O(n). To have a sorting algorithm work at linear time would be the Holy Grail. At this moment the only time this can occur is if the list or array is already sorted. In contrast there is also quadratic time or $O(n^2)$. This can occur with very bad algorithms that need to sort a list that is exactly the opposite if sorted. For example a list with values sorted high to low that needs to be low to high. With $O(n^2)$ if the list or array increases in size the sorting speed will increase with the power of 2.

Bubble sort

The bubble sort algorithm is easy to understand. If you have an array of 5 integer element that need to be sorted in increasing order of value the algorithm will start with the first and compare its value with the second. If the second value is less it will swap them. Then the algorithm will compare the second and third elements and check again. If the third element has a value less than the second it will swap them. It will go through this process until the end of the list or array is reached. The list will not be sorted perfectly after one loop, but it is an improvement. The list is sorted once the algorithm performs no more swapping operations. The name of the algorithm comes from the fact that high values found at the beginning of the list will be swapped all the way to the end in one loop. Bubble sort is slow, the best case scenario may be O(n) but then the list is already sorted. Average results are $O(n^2)$. Below is a Perl implementation of bubble sort

```
# to use the algorithm we create an array, the keyword 'my' makes the
@numbers variable local.
my @numbers = (96, 12, 35, 7, 82, 72, 3, 25, 43, 51);
# we call the function bubble_sort with he array
bubble_sort(@numbers);
# function bubble_sort sorts array
sub bubble_sort {
    for my $i (0 .. $#_){ #outer loop
        for my $j ($i + 1 .. $#_){ # inner loop
            # below we compare variables i and j, then we switch them
            if the condition is met
            $_[$j] < $_[$i] and @_[$i, $j] = @_[$j, $i];
        }
    }
}
```

Quicksort

One of the best sorting algorithms is Quicksort, first theorized by Tony Hoare in 1959 and implemented in 1961. Hoare is also famous for Hoare logic, a formal system of rules which can ensure the correctness of a part of a program. Quicksort is bit more difficult to understand than bubble sort, but I will do my best. Imagine you have an unordered list of 10 integers like the one shown below in brackets.

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

Now a random integer is selected. As an example we take the last one, 2. Now from the left number 6 to the right number 9 we will decide for each element if it is less or more than 2. We end up with two smaller lists, with integer 2 in the middle. The smallest list is to the left with just integer 1 and is in order.

[1,2,{6,5,3,8,4,7,9}]

As you may guess, this process is repeated for the larger list. Again we select a random number such as 6. Then we go through the remainder of the list and decide which elements are more and which are less than 6. The list will look like

[1,2,{5,3,4},6,{8,7,9}]

Again both of the sub lists will also be sorted until they are in their correct order. Quicksort has a worst case speed of $O(n^2)$, which may not seem like an improvement over bubble sort. Yet, its average performance is O(n log n) and thus decidedly better than bubble sort. Quicksort is probably one of the most popular sorting algorithms there is. Below you can find an implementation of Quicksort.

```
sub quick_sort {
    return @_ if @_ < 2; # ends the function if a sublist becomes
    smaller than 1
    my $random = splice @_, int rand @_, 1; # Select a random integer
    from the list and remove it with splice

    # we use the command grep to find all numbers lower or higher than
    $random and call quick_sort again on both lists. This process
    continuous until return statement is met
    quick_sort(grep $_ < $random, @_), $p, quick_sort(grep $_ >=
    $random, @_);
}

# create a unordered list of integers
my @quick = (6,5,1,3,8,4,7,9,2);
# call function quick_sort
@quick = quick_sort @quick;

# print the newly ordered list
print "@quick\n";
```

Recursion

A basic tenant in Computer Science is Recursion. This allows complex problems to be solved by solving many simpler problems instead. With recursion we repeatedly call the same method or function with a slightly simpler version of our problem. The quicksort algorithm used in the previous section uses recursion and is considered an excellent example of it use. Note how the unordered list is divided into two around a random number. Then for both sections the function is yet again called using the each smaller list as the new input. This is the recursive step. Of course the process cannot go on forever. We need to set a rule for when we consider a list sorted. The smallest possible list is one of size 1 and is

automatically sorted. This is called the *Base case*. This list is then returned with the **return** statement. Now this list and the random number are both sorted. If the right most recursive step eventually gets sorted it will also return back. Now the entire list is sorted.

Another example of a recursive algorithm that calls the same function over and over until a base case is found is calculating the factorial n! of a number n. An example would be 5!, which equals 120 (5*4*3*2*1). The Perl implementation would be as follows

```
use strict;
use warnings;

# request a number to factorize
print "Enter number\n";
my $num = <STDIN>;

# call the function
my $fac = fact($num,1);

# print the factorized number
print "factorial: $fac\n";

sub fact {
   my($num1, $flag) = @_;
   if($num1 == 0) { # base case
        return $flag;
   } else {
        fact($num1 - 1, $num1 * $flag);} # recursive step
}
```

I hope you have enjoyed this brief foray into the world of algorithms and the use of Perl. Remember, algorithms transcend programming languages. They form the heart of computer science and are worth devoting time on after you initial familiarization with programming.