Week 04 Laboratory Sample Solutions

Objectives

- Introduction to Perl programming.

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab04`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab04
$ cd lab04
$ 2041 fetch lab04
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE:

Deja Vu - Mapping Digits but in Perl

Write a Perl script `digits.pl` that reads from standard input and writes to standard output mapping all digit characters whose values are less than 5 into the character '<' and all digit characters whose values are greater than 5 into the character '>'. The digit character '5' should be left unchanged.

Sample Input Data	Corresponding Output
`1 234 5 678 9`	`< <<< 5 >>> >`

`I can think of 100's`
`of other things I'd rather`
`be doing than these 3 questions`
	`I can think of <<<'s`
	`of other things I'd rather`
	`be doing than these < questions`

`A line with lots of numbers:`
`12345678912345678912345678 9`
`A line with all zeroes`
`00000000000000000000000000`
`A line with blanks at the end`
`1 2 3`
	`A line with lots of numbers:`
	`<<<<5>>>>>>>>5>>>>>>>>5>>>>`
	`A line with all zeroes`
	`<<<<<<<<<<<<<<<<<<<<<<<<<<`
	`A line with blanks at the end`
	`< < <`

`Input with absolutely 0 digits in it` `Well ... apart from that one ...`	`Input with absolutely < digits in it` `Well ... apart from that one ...`
`1 2 4 8 16 32 64 128 256 512 1024` `2048 4096 8192 16384 32768 65536`	`< < < > <> << >< <<> <5> 5<< <<<<` `<<<> <<>> ><>< <><>< <<>>> >55<>`

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest perl_digits
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab04_perl_digits digits.pl
```

before **Tuesday 30 June 21:00** to obtain the marks for this lab exercise.

Sample solution for `digits.pl`

```
#!/usr/bin/perl -w
while ($line = <STDIN>) {
    $line =~ s/[0-4]/</g;
    $line =~ s/[6-9]/>/g;
    print $line;
}
```

Alternative solution for `digits.pl`

```
#!/usr/bin/perl -w
# using the implicit variable $_
while (<STDIN>) {
    s/[0-4]/</g;
    s/[6-9]/>/g;
    print;
}
```

Alternative solution for `digits.pl`

```
#!/usr/bin/perl -w
while (<STDIN>) {
    tr/0-9/<<<<<5>>>>/;
    print;
}
```

EXERCISE:

Echoing A Shell Exercise in Perl?

Write a Perl script `echon.pl` which given exactly two arguments, an integer *n* and a string, prints the string *n* times. For example:

```
./echon.pl 5 hello
hello
hello
hello
hello
hello
./echon.pl 0 nothing
./echon.pl 1 goodbye
goodbye
```

Your script should print exactly the error message below if it is not given exactly 2 arguments:

```
./echon.pl
Usage: ./echon.pl <number of lines> <string>
./echon.pl 1 2 3
Usage: ./echon.pl <number of lines> <string>
```

Also get your script to print this error message if its first argument isn't a non-negative integer:

```
./echon.pl hello world
./echon.pl: argument 1 must be a non-negative integer
./echon.pl -42 lines
./echon.pl: argument 1 must be a non-negative integer
```

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest perl_echon
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab04_perl_echon echon.pl
```

before **Tuesday 30 June 21:00** to obtain the marks for this lab exercise.

Sample solution for `echon.pl`

```perl
#!/usr/bin/perl -w
if (@ARGV != 2) {
    die "Usage: $0 <number of lines> <string>\n";
}
if ($ARGV[0] !~ /^\d+$/) {
    die "$0: argument 1 must be a non-negative integer\n";
}
foreach ($i=0; $i < $ARGV[0]; $i++) {
    print "$ARGV[1]\n";
}
```

Alternative solution for `echon.pl`

```perl
#!/usr/bin/perl -w
die "Usage: $0 <number of lines> <string>\n" if @ARGV != 2;
die "$0: argument 1 must be a non-negative integer\n" if $ARGV[0] !~ /^\d+$/;
foreach (1..$ARGV[0]) {
    print "$ARGV[1]\n";
}
```

Alternative solution for `echon.pl`

```perl
#!/usr/bin/perl -w
die "Usage: $0 <number of lines> <string>\n" if @ARGV != 2;
die "$0: argument 1 must be a non-negative integer\n" if $ARGV[0] !~ /^\d+$/;
print "$ARGV[1]\n" foreach 1..$ARGV[0];
```

Alternative solution for `echon.pl`

```perl
#!/usr/bin/perl -w
die "Usage: $0 <number of lines> <string>\n" if @ARGV != 2;
die "$0: argument 1 must be a non-negative integer\n" if $ARGV[0] !~ /^\d+$/;
print "$ARGV[1]\n" x $ARGV[0];
```

EXERCISE:

A Perl Tail

Perl file manipulation

The standard approach in Perl for dealing with a collection of files whose names are supplied as command line arguments, is something like:

```perl
#!/usr/bin/perl -w
foreach $arg (@ARGV) {
    if ($arg eq "--version") {
        print "$0: version 0.1\n";
        exit 0;
    # handle other options
    # ...
    } else {
        push @files, $arg;
    }
}

foreach $file (@files) {
    open F, '<', $file or die "$0: Can't open $file: $!\n";

    # process F

    close F;
}
```

Write a Perl script to implement the Unix `tail` command. It should support the following features of `tail`:

- read from files supplied as command line arguments
- read from standard input if no file name arguments are supplied
- display the error message `tail.pl: can't open` *`FileName`* for any unreadable file
- display the last *N* lines of each file (default *N* = 10)

- can adjust the number of lines displayed via an optional first argument *-N*
- if there is more than one named file, separate each by `==> FileName <==`

To assist with testing your solution, there are three small test files: t1.txt, t2.txt, and t3.txt. Copy these files to your current directory.

```
cp /web/cs2041/20T2/activities/perl_tail/t?.txt .
```

Using these data files, your program should behave as follows:

```
$ ./tail.pl <t1.txt
Data 1 ... Line 2
Data 1 ... Line 3
Data 1 ... Line 4
Data 1 ... Line 5
Data 1 ... Line 6
Data 1 ... Line 7
Data 1 ... Line 8
Data 1 ... Line 9
Data 1 ... Line 10
Data 1 ... Last line
$ ./tail.pl t1.txt
Data 1 ... Line 2
Data 1 ... Line 3
Data 1 ... Line 4
Data 1 ... Line 5
Data 1 ... Line 6
Data 1 ... Line 7
Data 1 ... Line 8
Data 1 ... Line 9
Data 1 ... Line 10
Data 1 ... Last line
$ ./tail.pl -5 t1.txt
Data 1 ... Line 7
Data 1 ... Line 8
Data 1 ... Line 9
Data 1 ... Line 10
Data 1 ... Last line
$ ./tail.pl -5 t2.txt
A one line file.
$ ./tail.pl -5 t1.txt t2.txt t3.txt
==> t1.txt <==
Data 1 ... Line 7
Data 1 ... Line 8
Data 1 ... Line 9
Data 1 ... Line 10
Data 1 ... Last line
==> t2.txt <==
A one line file.
==> t3.txt <==
```

Hint: use the above template for Perl file processing to get started with your script. You *must* use Perl's `-w` flag in your script, and you must write your code in such a way as to ensure that no warning messages are produced.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest perl_tail
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab04_perl_tail tail.pl
```

before **Tuesday 30 June 21:00** to obtain the marks for this lab exercise.

Sample solution for `tail.pl`

```
#!/usr/bin/perl -w

$max_lines = 10;

if (@ARGV > 0 && $ARGV[0] =~ /-([0-9]+)/) {
    $max_lines = $1;
    shift @ARGV;
}

if (@ARGV == 0) {
    @lines = <>;
    $first = @lines - $max_lines;
    $first = 0 if $first < 0;
    print @lines[$first..$#lines];
} else {
    $show_file_names = @ARGV > 1;
    foreach $file (@ARGV) {
        open my $f, '<', $file or die "$0: can't open $file\n";
        print "==> $file <==\n" if $show_file_names;
        @lines = <$f>;
        $first = @lines - $max_lines;
        $first = 0 if $first < 0;
        print @lines[$first..$#lines];
        close $f;
    }
}
```

CHALLENGE EXERCISE:

Shuffling Lines

Write a Perl script `shuffle.pl` which prints its input with the lines in random order. For example, lets create a file containing the integers 0..4.

```
seq 0 4 >numbers.txt
```

Now if we run `shuffle.pl` taking its input from this file it should print the lines in a different order each time its run, for example:

```
$ cat numbers.txt
0
1
2
3
4
$ ./shuffle.pl <numbers.txt
1
0
4
3
2
$ ./shuffle.pl <numbers.txt
3
4
2
0
1
$ ./shuffle.pl <numbers.txt
2
1
```

You are not permitted to use `List::Util` (it contains a shuffle function).

Don't look for other people's solutions - see if you can come up with your own. **Hint:** the perl function *rand* returns a floating point number between 0 and its argument. For example:

```
$ perl -e 'print rand(42), "\\n"'
38.3441749865398\n
```

```
$ perl -e 'print rand(42), "\\n"'
15.5781439648219\n
```

Hint: perl ignores the fractional part of a number if you use it to index an array

There is no autotest and no automarking of this question.

When you are finished working on this exercise, demonstrate your work to another student in your lab and ask them to enter a peer assessment. It is preferred you do this during your lab, but if this is not possible you may demonstrate your work to any other COMP(2041|9044) student before Tuesday 30 June 21:00. Note, you must also submit the work with `give`.

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab04_perl_shuffle shuffle.pl
```

before **Tuesday 30 June 21:00** to obtain the marks for this lab exercise.

Sample solution for `shuffle.pl`

```
#!/usr/bin/perl -w
# simple implementation of http://en.wikipedia.org/wiki/Fisher-Yates_shuffle
@lines = <>;
print splice(@lines, rand(@lines), 1) while @lines;
```

Alternative solution for `shuffle.pl`

```
#!/usr/bin/perl -w
use List::Util 'shuffle';
print shuffle(<>);
```

CHALLENGE EXERCISE:

Testing a Non-determinate Program

There is no dryrun test for `shuffle.pl`. Testing (pseudo)random programs is more difficult because there are multiple correct outputs for a given input.

Write a shell script `shuffle_test.sh` which tests `shuffle.pl`.

Try to test that all outputs are correct and all correct outputs are being generated.

Sample solution that just checks coverage

```
#!/bin/sh

input=/tmp/shuffle_test0$$
output=/tmp/shuffle_test1$$
sorted_output=/tmp/shuffle_test2$$
all_output=/tmp/shuffle_test3$$

number_of_lines=4
number_of_test_runs=256

# create an input file with 1 integer per line in sorted order
# and calculate how many permutations are possible
i=1
factorial=1
while test $i -le $number_of_lines
do
    echo $i
    factorial=$(($factorial * $i))
    i=$(($i + 1))
done >$input

run=1
while test $run -le $number_of_test_runs
do
    ./shuffle.pl <$input >$output
    sort -n $output >$sorted_output

    # after sorting output should be identical to input
    if diff $sorted_output $input >/dev/null
    then
        # append result of this execution to $all_output as a single line
        echo `cat $output` >>$all_output
    else
        echo Testing failed, input was:
        cat $input
        echo Testing failed, output was:
        cat $output
        exit 1
    fi
    run=$(($run + 1))
done

n_different_outputs=`sort $all_output|uniq|wc -l`
if test $n_different_outputs -eq $factorial
then
    echo All possible outputs produced
    exit 0
else
    echo In $number_of_test_runs executions only $n_different_outputs of $factorial outputs produced
    exit 1
fi

rm -f $input $output $sorted_output $all_output
```

A more elaborate solution from Donny Yang which takes a more statistical approach

```
#!/bin/sh

input=/tmp/shuffle_test0$$
output=/tmp/shuffle_test1$$
sorted_output=/tmp/shuffle_test2$$
all_output=/tmp/shuffle_test3$$

number_of_lines=4
number_of_test_runs=256

# create an input file with 1 integer per line in sorted order
# and calculate how many permutations are possible
i=1
factorial=1
while test $i -le $number_of_lines
do
    echo $i
    factorial=$(($factorial * $i))
    i=$(($i + 1))
done >$input

run=1
while test $run -le $number_of_test_runs
do
    ./shuffle.pl <$input >$output
    sort -n $output >$sorted_output

    # after sorting output should be identical to input
    if diff $sorted_output $input >/dev/null
    then
        # append result of this execution to $all_output as a single line
        echo `cat $output` >>$all_output
    else
        echo Testing failed, input was:
        cat $input
        echo Testing failed, output was:
        cat $output
        exit 1
    fi
    run=$(($run + 1))
done

n_different_outputs=`sort $all_output|uniq|wc -l`
if test $n_different_outputs -eq $factorial
then
    echo All possible outputs produced
    exit 0
else
    echo In $number_of_test_runs executions only $n_different_outputs of $factorial outputs produced
    exit 1
fi

rm -f $input $output $sorted_output $all_output
```

There is no autotest and no automarking of this question.

When you are finished working on this exercise, demonstrate your work to another student in your lab and ask them to enter a peer assessment. It is preferred you do this during your lab, but if this is not possible you may demonstrate your work to any other COMP(2041|9044) student before Tuesday 30 June 21:00. Note, you must also submit the work with `give`.

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab04_shuffle_test shuffle_test.sh
```

before **Tuesday 30 June 21:00** to obtain the marks for this lab exercise.

Sample solution for `shuffle_test.sh`

```sh
#!/bin/sh

input=/tmp/shuffle_test0$$
output=/tmp/shuffle_test1$$
sorted_output=/tmp/shuffle_test2$$
all_output=/tmp/shuffle_test3$$

number_of_lines=4
number_of_test_runs=256

# create an input file with 1 integer per line in sorted order
# and calculate how many permutations are possible
i=1
factorial=1
while test $i -le $number_of_lines
do
    echo $i
    factorial=$(($factorial * $i))
    i=$(($i + 1))
done >$input

run=1
while test $run -le $number_of_test_runs
do
    ./shuffle.pl <$input >$output
    sort -n $output >$sorted_output

    # after sorting output should be identical to input
    if diff $sorted_output $input >/dev/null
    then
        # append result of this execution to $all_output as a single line
        echo `cat $output` >>$all_output
    else
        echo Testing failed, input was:
        cat $input
        echo Testing failed, output was:
        cat $output
        exit 1
    fi
    run=$(($run + 1))
done

n_different_outputs=`sort $all_output|uniq|wc -l`
if test $n_different_outputs -eq $factorial
then
    echo All possible outputs produced
    exit 0
else
    echo In $number_of_test_runs executions only $n_different_outputs of $factorial outputs produced
    exit 1
fi

rm -f $input $output $sorted_output $all_output
```

Alternative solution for `shuffle_test.sh`

```
#!/bin/bash
# A more elaborate solution from Donny Yang which takes a more statistical approach

function runTest {
    INPUT_SIZE="$1"
    TRIALS="$2"

    # Binomial distribution
    declare -A PROBABILITIES
    EXPECTED=$((TRIALS / INPUT_SIZE))
    ONE_SIGMA=$(bc <<< "sqrt($TRIALS * ($INPUT_SIZE - 1) / $INPUT_SIZE^2)")
    TWO_SIGMA=$(bc <<< "sqrt(2^2 * $TRIALS * ($INPUT_SIZE - 1) / $INPUT_SIZE^2)")
    THREE_SIGMA=$(bc <<< "sqrt(3^2 * $TRIALS * ($INPUT_SIZE - 1) / $INPUT_SIZE^2)")
    FOUR_SIGMA=$(bc <<< "sqrt(4^2 * $TRIALS * ($INPUT_SIZE - 1) / $INPUT_SIZE^2)")

    echo "Shuffling $INPUT_SIZE values $TRIALS times ($INPUT_SIZE x $TRIALS)" >&2

    for ((INPUT=1; INPUT<=INPUT_SIZE; ++INPUT)); do
        for ((OUTPUT=0; OUTPUT<INPUT_SIZE; ++OUTPUT)); do
            PROBABILITIES["$INPUT,$OUTPUT"]=0
        done
    done

    INPUT="$(seq "$INPUT_SIZE")"
    for ((TRIAL=0; TRIAL<TRIALS; ++TRIAL)); do
        RESULT=($(<<< "$INPUT" ./shuffle.pl)) # Bottleneck here since perl takes >1 us to start
        for LINE_NUMBER in "${!RESULT[@]}"; do
            if [ "$LINE_NUMBER" -ge "$INPUT_SIZE" ]; then
                echo "Failed ($INPUT_SIZE x $TRIALS): Script outputted more lines than expected (>$INPUT_SIZE):"
>&2
                echo "$RESULT" >&2
                return 1
            fi

            ((++PROBABILITIES["${RESULT[$LINE_NUMBER]},$LINE_NUMBER"]))
        done
    done

    echo "Expected mean: $EXPECTED"
    echo $'\033[0;32m1\033[0m / \033[0;33m2\033[0m / \033[1;33m3\033[0m / \033[0;31m4\033[0m-sigma:
\033[0;32m'"$ONE_SIGMA"$'\033[0m / \033[0;33m'"$TWO_SIGMA"$'\033[0m / \033[1;33m'"$THREE_SIGMA"$'\033[0m /
\033[0;31m'"$FOUR_SIGMA"$'\033[0m'

    echo -n "   Out"$'\t'
    for ((OUTPUT=0; OUTPUT<INPUT_SIZE; ++OUTPUT)); do
        echo -n "$((OUTPUT + 1))"$'\t'
    done
    echo
    echo '  \'
    echo "In"
    for ((INPUT=1; INPUT<=INPUT_SIZE; ++INPUT)); do
        echo -n "$INPUT"$'\t'
        for ((OUTPUT=0; OUTPUT<INPUT_SIZE; ++OUTPUT)); do
            OFFSET=$((PROBABILITIES["$INPUT,$OUTPUT"] - EXPECTED))
            if [ "-$ONE_SIGMA" -le "$OFFSET" ] && [ "$OFFSET" -le "$ONE_SIGMA" ]; then
                echo -n $'\033[0;32m'"$OFFSET"$'\033[0m\t'
            elif [ "-$TWO_SIGMA" -le "$OFFSET" ] && [ "$OFFSET" -le "$TWO_SIGMA" ]; then
                echo -n $'\033[0;33m'"$OFFSET"$'\033[0m\t'
            elif [ "-$THREE_SIGMA" -le "$OFFSET" ] && [ "$OFFSET" -le "$THREE_SIGMA" ]; then
                echo -n $'\033[1;33m'"$OFFSET"$'\033[0m\t'
            elif [ "-$FOUR_SIGMA" -le "$OFFSET" ] && [ "$OFFSET" -le "$FOUR_SIGMA" ]; then
                IS_FAILED="almost"
                echo -n $'\033[0;31m'"$OFFSET"$'\033[0m\t'
            else
                IS_FAILED="yes"
                echo -n $'\033[1;31m'"$OFFSET"$'\033[0m\t'
            fi
        done
        echo
    done
```

```
    if [ "$IS_FAILED" == "yes" ]; then
        echo $'\033[1;31m'"Failed ($INPUT_SIZE x $TRIALS): Output distribution not within 4-sigma"$'\033[0m\t' >&2
        return 1
    elif [ "$IS_FAILED" == "almost" ]; then
        echo $'\033[0;31m'"Almost failed ($INPUT_SIZE x $TRIALS): Output distribution not within 3-
sigma"$'\033[0m\t' >&2
    else
        echo "Passed ($INPUT_SIZE x $TRIALS)" >&2
    fi
}

set -e
trap "trap - SIGTERM && kill -- -$$" SIGINT SIGTERM EXIT

runTest 5 2000 &
runTest 10 4000 &
runTest 5 10000 &
runTest 10 20000 &
runTest 20 40000 &

sleep 1
for N in $(seq 5); do
    echo
    wait "$(jobs -p | head -n1)"
done
trap - EXIT
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Tuesday 30 June 21:00** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 2041 classrun -sturec
```