Week 10 Tutorial Sample Answers

1. How is the assignment going?

   Does anyone have hints or advice for other students?

   Has anyone discovered interesting cases that have to be handled?

   Answer:

   Discussed in tutorial.

2. These commands both copy a directory tree to a CSE server.

   ```
   $ scp -r directory1/ z1234567@login.cse.unsw.edu.au:directory2/
   $ rsync -a directory1/ z1234567@login.cse.unsw.edu.au:directory2/
   ```

   What underlies them?

   How do they differ?

   Why are these differences important?

   Answer:

   Both command use `ssh` to connect to the remote machine.

   `rsync` will be more efficient if `directory2` already exists with similar contents. It will copy only files which are different and if only part of a file has changed it can copy only the changed bytes. This is is much more efficient when we are repeatedly making copies, e.g. for backups, particularly over a slow connection

   `rsync -a` also copies metadata such as file permissions and modification data which may be very important.

   `scp -rp` would copy metadata.

3. Assumes Linux kernel source tree containing thousand of source files can be found in `/usr/src/linux`

   Write a shell script that emails each of the 50,000 source (.c) files to Andrew, each as an attachment to a separate email.

   The source files may be anywhere in a directory tree than goes 10+ levels deep.

   Please don't run this script, andin general be very careful with such scripts. It is very embarassing to accidentally send thousands of emails.

   What assumptions does your script make?

   Answer:

   ```
   #!/bin/sh

   directory=/usr/src/linux

   # depends on pathnames not containing white-space

   for c_file in $(find "$directory" -type f -name '*.c')
   do
       echo mutt -s "C for you"  -a "$c_file" -- andrewt@unsw.edu.au
   done
   ```

4. Write a Shell script, `tags.sh` which prints the HTML tags it uses.

   The tag should be converted to lower case and printed in sorted order with a count of how often each is used.

   Don't count closing tags.

```
$ ./tags.sh https://www.google.com/
     18 a
      1 b
      1 body
      5 br
      1 center
     11 div
      1 form
      1 head
      1 html
      1 img
     10 input
      2 meta
      2 nobr
      1 p
      6 script
      8 span
      2 style
      1 table
      3 td
      1 title
      1 tr
      1 u
```

Discuss any assumptions you make.

Answer:

```
#!/bin/sh

# count HTML tags in the web pages given as command-line arguments
# does not handle HTML comments
# assumes <> appear only in tags
# tags spread over multiple-lines will not be detected

curl -s "$@"|               # fetch the web page
tr A-Z a-z|                 # convert every tolower case
tr '>' '\n'|                # ensure each tag is on a separate line
egrep '<'|                  # ignore lines without tags
sed 's?.*< *??'|            # remove everything up to start of tag
egrep '^[a-z]'|             # remove things that look like tags but aren't
sed 's/[^a-z].*//'|         # delete every after the tag
sort|                       # count the tags
uniq -c
```

5. Write a *shell script* called `rmall.sh` that removes all of the files and directories below the directory supplied as its single command-line argument. The script should prompt the user with `Delete` *X*? before it starts deleting the contents of any directory *X*. If the user responds `yes` to the prompt, `rmall` should remove all of the plain files in the directory, and then check whether the contents of the subdirectories should be removed. The script should also check the validity of its command-line arguments.

Sample solution

```
#!/bin/sh

# check whether there is a cmd line arg
case $# in
1) # ok ... requires exactly one arg
    ;;
*)
    echo "Usage: $0 dir"
    exit 1
esac

# then make sure that it is a directory
if test ! -d $1
then
    echo "$1 is not a directory"
    echo "Usage: $0 dir"
    exit 1
fi

# change into the specified directory
cd $1

# for each plain file in the directory
for f in .* *
do
    if test -f "$f"
    then
        rm $f
    fi
done

# for each subdirectory
for d in .* *
do
    if test -d "$d" -a "$d" != .  -a "$d" != ..
    then
        echo -n "Delete $d? "
        read answer
        if test "$answer" = "yes"
        then
             rmall "$d"
        fi
     fi
done
```

Alternative solution using case

```
#!/bin/sh

# check whether there is a cmd line arg
case $# in
1) # ok ... requires exactly one arg
    ;;
*)
    echo "Usage: $0 dir"
    exit 1
esac

# then make sure that it is a directory
if test ! -d $1
then
    echo "$1 is not a directory"
    echo "Usage: $0 dir"
    exit 1
fi

# change into the specified directory
cd $1

# for each plain file in the directory
for f in .* *
do
    case $f in
    .|..) # ignore . and ..
        ;;
    *)
        if test -f $f
        then
            rm $f
        fi
        ;;
    esac
done

# for each subdirectory
for d in .* *
do
    case $d in
    .|..) # ignore . and ..
        ;;
    *)
        if test -d $d
        then
            echo -n "Delete $d? "
            read answer
            if test "$answer" = "yes"
            then
                rmall $d
            fi
        fi
        ;;
    esac
done
```

6. Write a *shell script* called `check` that looks for duplicated student ids in a file of marks for a particular subject. The file consists of lines in the following format:

```
2233445 David Smith 80
2155443 Peter Smith 73
2244668 Anne Smith 98
2198765 Linda Smith 65
```

The output should be a list of student ids that occur 2+ times, separated by newlines. (i.e. any student id that occurs more than once should be displayed on a line by itself on the standard output).

Sample solution

```
#!/bin/sh

cut -d' ' -f1 < Marks | sort | uniq -c | egrep -v '^ *1 ' | sed -e 's/^.* //'
```

Explanation:

1. `cut -d' ' -f1 < Marks` ... extracts the student ID from each line
2. `sort | uniq -c` ... sorts and counts the occurrences of each ID
3. IDs that occur once will be on a line that begins with spaces followed by 1 followed by a TAB
4. `grep -v '^ *1 '` removes such lines, leaving only IDs that occur multiple times
5. `sed -e 's/^.* //'` gets rid of the counts that `uniq -c` placed at the start of each line

7. Write a *Perl script* **revline.pl** that reverses the fields on each line of its standard input.

   Assume that the fields are separated by spaces, and that only one space is required between fields in the output.

   For example

```
$ ./revline.pl
hi how are you
i'm great thank you
Ctrl-D
you are how hi
you thank great i'm
```

Obvious readable solution

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

while ($line = <STDIN>) {
    chomp $line;
    my @fields = split /\s+/, $line;
    @fields = reverse @fields;
    $line_out = join ' ', @fields;
    print "$line_out\n";
}
```

Or using Perls $_ variable.

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

while (<STDIN>) {
    chomp;
    my @fields = split;
    @fields = reverse @fields;
    $line_out = join ' ', @fields;
    print "$line_out\n";
}
```

or exploiting perl's -p command flags:

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

chomp;
$_ = join ' ', reverse split;
```

Revision questions

The following questions are primarily intended for revision, either this week or later in session. Your tutor may still choose to cover some of these questions, time permitting.

5. Which one of the following regular expressions would match a non-empty string consisting only of the letters `x`, `y` and `z`, in any order?

   a. `[xyz]+`

   b. `x+y+z+`

   c. `(xyz)*`

   d. `x*y*z*`

a. Correct

b. Incorrect ... this matches strings like `xxx...yyy...zzz...`

c. Incorrect ... this matches strings like `xyzxyzxyz...`

d. Incorrect ... this matches strings like `xxx...yyy...zzz...`

The difference between (b) and (d) is that (b) requires there to be at least one of each `x`, `y` and `z`.

6. Which one of the following commands would extract the student id field from a file in the following format:

```
COMP3311;2122987;David Smith;95
COMP3231;2233445;John Smith;51
COMP3311;2233445;John Smith;76
```

a. `cut -f 2`

b. `cut -d; -f 2`

c. `sed -e 's/.*;//'`

d. None of the above.

a. Incorrect ... this gives the entire data file; the default field-separator is tab, and since the lines contain no tabs, they are treated as a single large field; if an invalid field number is specified, `cut` simply prints the first

b. Incorrect ... this runs two separate commands `cut -d` followed by `-f 2`, and neither of them makes sense on its own

c. Incorrect ... this removes all chars up to and including the final semicolon in the line, and this gives the 4th field on each line

d. Correct

7. Which one of the following Perl commands would acheive the same effect as in the previous question (i.e. extract the student id field)?

a. `perl -e '{while (<>) { split /;/; print;}}'`

b. `perl -e '{while (<>) { split /;/; print $2;}}'`

c. `perl -e '{while (<>) { @x = split /;/; print "$x[1]\n";}}'`

d. `perl -e '{while (<>) { @x = split /;/; print "$x[2]\n";}}'`

a. Incorrect ... this splits the line, but doesn't save the result of the splitting, and then prints the default value, which is the whole line read

b. Incorrect ... `$2` does not refer to the second field in Perl

c. Correct ... the `split` saves the result in the `@x` list, and the index `[1]` selects the second value from the list

d. Incorrect ... the `split` saves the result in the `@x` list, but the index `[2]` selects the third value from the list

8. Consider the following Perl program that processes its standard input:

```
#!/usr/bin/perl -w
while (<STDIN>) {
    @marks = split;
    $studentID = $marks[0];
    for (i = 0; i < $#marks; i++) {
        $totalMark += $marks[$i];
    }
    printf "%s %d\n", $studentID, $totalMark;
}
```

This program has several common mistakes in it. Indicate and describe the nature of each of these mistakes, and say what the program is attempting to do.

- The `for` loop uses the "variable" `i` but forgets to prefix it with the `$` symbol, so it will be treated as a constant and an error message generated
- The iteration over the marks is incorrect; the value `$#marks` gives the index of the last array element; since the loop runs to less than `$#marks` it will miss the last element
- A related point: since the first element in the array is the student ID and not a mark, it should not be included in the `$totalMark`; the loop iteration should start from `$i = 1`.
- The value of `$totalMark` is not reset for each student, so the total simply increases continually and does not reflect the sum of marks for any individual except the first student

9. Consider the following table of student enrolment data:

StudentID	Course	Year	Session	Mark	Grade

2201440	COMP1011	1999	S1	57	PS
2201440	MATH1141	1999	S1	51	PS
2201440	MATH1081	1999	S1	60	PS
2201440	PHYS1131	1999	S1	52	PS
...	...	...	...	...	...

A file containing a large data set in this format for the years 1999 to 2001 and ordered by student ID is available in the file *data*.

Write a program that computes the average mark for a specified course for each of the sessions that it has run. The course code is specified as a command-line argument, and the data is read from standard input. All output from the program should be written to the standard output.

If no command-line argument is given, the program should write the following message and quit:

```
Usage: ex3 Course
```

The program does *not* have to check whether the argument is valid (i.e. whether it looks like a real course code). However, if the specified course code (*CCODE*) does not appear anywhere in the data file, the program should write the following message:

```
No marks for course CCODE
```

Otherwise, it should write one line for each session that the course was offered. The line should contain the course code, the year, the session and the average mark for the course (with one digit after the decimal point). You can assume that a course will not be offered more than 100 times. The entries should be written in chronological order.

The following shows an example input/output pair for this program:

Sample Input Data	Corresponding Output

`COMP1011`	`COMP1011 1999 S1 62.5`
	`COMP1011 2000 S1 69.1`
	`COMP1011 2001 S1 66.8`

Sample Perl solution

```perl
#!/usr/bin/perl

if (@ARGV < 1) {
    die "Usage: ex3 Course\n";
} else {
    $c = $ARGV[0];
}

while (<STDIN>) {
    chomp;
    my ($sid,$course,$year,$sess,$mark,$grade) = split;

    if ($course eq $c) {
        $sum{"$year $sess"} += $mark;
        $count{"$year $sess"}++;
        $nofferings++;
    }
}

if ($nofferings == 0) {
    print "No marks for course $c\n";
} else {
    foreach $s (sort keys %sum) {
        printf "$c $s %0.1f\n", $sum{"$s"}/$count{"$s"};
    }
}
```

10. Write a Perl program **frequencies.pl** that prints a count of how often each letter ('a'..'z' and 'A'..'Z') and digit ('0'..'9') occurs in its input. Your program should follow the output format indicated in the examples below exactly.

    No count should be printed for letters or digits which do not occur in the input.

    The counts should be printed in dictionary order ('0'..'9','A'..'Z','a'..'z').

    Characters other than letters and digits should be ignored.

    The following shows an example input/output pair for this program:

```
$ ./frequencies.pl
The  Mississippi is
1800 miles long!
Ctrl-D
'0' occurred 2 times
'1' occurred 1 times
'8' occurred 1 times
'M' occurred 1 times
'T' occurred 1 times
'e' occurred 2 times
'g' occurred 1 times
'h' occurred 1 times
'i' occurred 6 times
'l' occurred 2 times
'm' occurred 1 times
'n' occurred 1 times
'o' occurred 1 times
'p' occurred 2 times
's' occurred 6 times
```

Clear readable Perl solution

```
#!/usr/bin/perl -w
# courtesy aek@cse.unsw.EDU.AU
# letter count- count number of occurrences of each letter

# map letters to counts
my %lettercount = ();
while (<>) {
        chomp;

        # remove anything but letters and numbers
        s/[^A-Za-z0-9]//g;

        # split the line into an array of characters
        @chars = split //;
        foreach $letter (@chars) {
                # record count in hash table
                $lettercount{$letter}++;
        }
}

# output count of each letter, sorted on the keys (letters)
foreach $letter (sort keys %lettercount) {
        print "'$letter' occurred $lettercount{$letter} times\n";
        # (look up count for each letter from table)
}
```

Terse less-reable Perl soluton:

```
#!/usr/bin/perl -w
while (<>) {
    for (split //) {
        $count{$_}++ if /[a-zA-Z0-9]/;
    }
}
print "'$_' occurred $count{$_} times\n" for sort keys %count;
```

A minimal Perl soluton:

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

$freq{$_}++ for grep /[a-z0-9]/i, (split //, (join '', <>));
print "'$_' occurred $freq{$_} times\n" for sort keys %freq;
```

Clear readable Python solution

```python
#!/usr/bin/python

import fileinput, collections, sys, re

freq = collections.defaultdict(int)

for line in fileinput.input():
    chars = list(line)
    for c in chars:
        if c.isalnum():
            freq[c] += 1

for f in sorted(freq):
    print("'{}' occurred {} times".format(f, freq[f]))
```

A less-reable Python solution:

```python
#!/usr/bin/python

import fileinput, collections, sys, re

freq = collections.defaultdict(int)
for c in filter(lambda x: x.isalnum(), list(''.join(fileinput.input()))):
    freq[c] += 1
for f in sorted(freq):
    print("'{}' occurred {} times".format(f, freq[f]))
```

11. Write a Perl program that maps all lower-case vowels (a,e,i,o,u) in its standard input into their upper-case equivalents and, at the same time, maps all upper-case vowels (A, E, I, O, U) into their lower-case equivalents.
The following shows an example input/output pair for this program:

Sample Input Data	Corresponding Output

`This is some boring text.`
`A little foolish perhaps?`
	`ThIs Is sOmE bOrIng tExt.`
	`a lIttlE fOOlIsh pErhAps?`

Sample Perl solution

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

@lines = <STDIN>;
map {tr /aeiouAEIOU/AEIOUaeiou/} @lines;
print @lines;
```

Another Sample Shell solution

```sh
#!/bin/sh

tr aeiouAEIOU AEIOUaeiou
```

12. A "hill vector" is structured as an *ascent*, followed by an *apex*, followed by a *descent*, where
    - the *ascent* is a non-empty strictly ascending sequence that ends with the apex
    - the *apex* is the maximum value, and must occur only once
    - the *descent* is a non-empty strictly descending sequence that starts with the apex

For example, [1,2,3,4,3,2,1] is a hill vector (with apex=4) and [2,4,6,8,5] is a hill vector (with apex=8). The following vectors are not hill vectors: [1,1,2,3,3,2,1] (not strictly ascending and multiple apexes), [1,2,3,4] (no descent), and [2,6,3,7,8,4] (not ascent then descent). No vector with less than three elements is considered to be a hill.

Write a Perl program **hill_vector.pl** that determines whether a sequence of numbers (integers) read from standard input forms a "hill vector". The program should write "hill" if the input *does* form a hill vector and write "not hill" otherwise.

Your program's input will only contain digits and white space. Any amount of whitespace may precede or follow integers.

Multiple integers may occur on the same line.

A line may contain no integers.

You can assume all the integers are positive. The following shows example input/output pairs for this program:

Sample Input Data	Corresponding Output
`1 2 4 8 5 3 2`	`hill`

`1 2`	`not hill`
`1 3 1`	`hill`

`  3`
`1   1`
	`not hill`

`2 4 6 8 10 10 9 7 5 3 1`	`not hill`

Sample Perl solution

```perl
#!/usr/bin/perl -w
@n = split /\D+/, join(' ', <>);
$i= 0;
$i++ while $i < $#n && $n[$i] < $n[$i+1];
$j = $#n;
$j-- while $j > 0 && $n[$j] < $n[$j-1];
print "not " if $i != $j || $i == 0 || $j == $#n;
print "hill\n";
```

13. A list **a_1**, **a_2**, ... **a_n** is said to be **converging** if

$$a_1 > a_2 > \ldots > a_n$$

and

$$\text{for all i } a_{i-1} - a_i > a_i - a_{i+1}$$

In other words, the list is strictly decreasing and the difference between consecutive list elements always decreases as you go down the list.

Write a Perl program **converging.pl** that determines whether a sequence of positive integers read from standard input is converging. The program should write "converging" if the input is converging and write "not converging" otherwise. It should produce no other output.

Sample Input Data	Corresponding Output
`2010 6 4 3`	`converging`

`20`
`15`
`9`
	`not converging`

`1000`
`     100    10`
`     1`
	`converging`

`  6`
`  5`
`2 2`
	`not converging`

`  1 2 4 8`	`not converging`

Your program's input will only contain digits and white space. Any amount of whitespace may precede or follow integers.

Multiple integers may occur on the same line.

A line may contain no integers.

You can assume your input contains at least 2 integers.

You can assume all the integers are positive.

Sample Perl solution

```
#!/usr/bin/perl -w
@n = split /\D+/, join(' ', <>);
foreach $i (1..$#n) {
    if ($n[$i] >= $n[$i+1]) {
        print "not converging\n";
        exit;
    }
}
foreach $i (2..$#n) {
    if ($n[$i-2] - $n[$i-1] <= $n[$i-1] - $n[$i]) {
        print "not converging\n";
        exit;
    }
}
print "converging\n";
```

14. The *weight* of a number in a list is its value multiplied by how many times it occurs in the list. Consider the list `[1 6 4 7 3 4 6 3 3]`. The number 7 occurs once so it has weight 7. The number 3 occurs 3 times so it has weight 9. The number 4 occurs twice so it has weight 8.

    Write a Perl program **heaviest.pl** which takes 1 or more positive integers as arguments and prints the heaviest.

    Your Perl program should print one integer and no other output.

    Your Perl program can assume it it is given only positive integers as arguments

```
$ ./heaviest.pl 1 6 4 7 3 4 6 3 3
6
$ ./heaviest.pl 1 6 4 7 3 4 3 3
3
$ ./heaviest.pl 1 6 4 7 3 4 3
4
$ ./heaviest.pl 1 6 4 7 3 3
7
```

Sample Perl solution

```
#!/usr/bin/perl -w
foreach $n (@ARGV) {
    $w{$n * grep {$_ == $n} @ARGV} = $n;
}
print $w{(sort {$b <=> $a} keys %w)[0]}, "\n";
```