COMP[29]041 18s2

Week 13 Tutorial

COMP[29]041 18s2

Sample Answers

1. How is assignment 2 going?
   Do students who've made progress with the assignment have advice for students not so far along?

   Do students have questions that other may be able to answer?

Revision questions

The remaining tutorial questions are primarily intended for revision - either this week or later in session.
Your tutor may still choose to cover some of the questions time permitting.

2. In Javascript fetch("resource_url") ...
   a. Returns the resource immediately

   b. Returns the resource once the server responds

   c. Returns a promise immediately and calls a callback once the server responds

   d. Returns a callback immediately and invokes a promise once the server responds

   A fetch returns a promise object immediately, and once the server responds will attempt to call any attached callback. Note that the callback is the function you specify in the "then" function, i.e fetch("url").then(callback)

3. Why do we place scripts at the bottom of a html page?
   a. So if the script takes a while to load the rest of the page isn't frozen

   b. So if the javascript needs to use elements on the page, they are loaded before the javascript is run

   c. Having scripts in the head tag is not supported in older browsers

   d. A + C

   e. A + B

   Correct answer is E (A+B), the script may take a while so we want the page to load while it's getting processed but in addition if your code has a line such as getElementById("myElem") then it'll fail if the script is run before the element "myElem" exists on the page. Do note we can avoid the latter issues by using window.onload and other solutions.

4. What is the following code trying to do and what's wrong with it. How would you correct it?

```
const nums = [0,1,2,3,undefined,4,5,6,undefined,7];
const only_defined_nums = nums.filter((r)=>r)
```

   Code attempts to grab all numbers that arn't undefined from a list. it evaluates all the elements as booleans which partialy works becuse undefined == false but this is only a half solution as 0 == false too.

5. Write some js that given this snippet of HTML, will change the color of the text in the input to be red if the email is invalid and green otherwise on the button click. You may assume a valid email is `A@A.A` where A is one or more alpha numeric characters, i.e a-z,A-Z,0-9

```
<input id="email"> </input>
<button id="check">check email</button>
```

   Sample Js solution

```
// bind event
document.getElementById("check").addEventListener("click", checkEmail);

function checkEmail() {
  // get the email
  const email = document.getElementById("email");
  const value = email.value;
  // check
  if (/^[0-9a-z]+@[0-9a-z]+\.[0-9a-z]+$/i.exec(value)) {
    email.style.color = "green";
  } else {
    email.style.color = "red";
  }
}
```

6. Write some js that fetches 3 strings from the endpoints "test.com/1" "test.com/2" and "test.com/3" and prints them out together them with a space in between. If the 3 endpoints returned "Spooky", "Scary" and "Skeletons" respectively then the string "Spooky Scary Skeletons" would be printed out.

    Sample Js solution

```
let URLS = ["test.com/1", "test.com/2", "test.com/3"];
URLS = URLS.map((url) => fetch(url));
Promise.all(URLS).then((responses) => {
  const string = responses.map(r => r.text()).join(" ");
  console.log(string);
})
```

7. In the context of computing, a *shell* is
    a. part of the Unix operating system

    b. a program that arranges the execution of other programs

    c. a component of a window manager such as `fvwm`

    d. an object-oriented wrapper for a procedural program

    a. Incorrect ... the shell runs as a normal process under Unix; it is not part of the O/S

    b. Correct.

    c. Incorrect ... the shell is completely separate from any window manager; you can run a shell without having any window manager running.

    d. Incorrect ... meaningless answer; certainly not shell-related

8. 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`.

9. 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

10. 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

11. 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

12. 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
    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
```

13. 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

14. 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;
```

15. 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

```
#!/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"};
    }
}
```

16. 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;
```

17. Write a program **frequencies.py** - see previous question for details.

Clear readable Python solution

```
#!/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 soluton:

```
#!/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]))
```

18. 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

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

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

Another Sample Shell solution

```
#!/bin/sh

tr aeiouAEIOU AEIOUaeiou
```

19. 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

```
#!/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";
```

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

$$a_1 > a_2 > ... > a_n$$

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 consecuctive 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";
```

21. 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";
```