Week 01 Tutorial

Sample Answers

1. What is your tutor's name, e-mail, how long have they been at UNSW, what are they studying, what is 1 interesting thing about them?

   Answered in tute.

2. What are your class mates's names, what are they each studying, what is 1 interesting thing about each of them?

   Answered in tute.

3. Are there marks for attending lectures, tutorials or labs?

   No attendence marks for lectures, tutorials or labs.

4. What is an operating system? What operating systems are running in your tute room? What operating system do CSE lab computers run?

   An operating system is a piece of software that manages the hardware of a computer and provides an interface to the programs that run on the computer. Operating systems on phones in your tut room might include Linux (Android), IOS (iPhone), Windows. CSE's lab computers and servers run Linux.

5. a. Write a regexp to match C preprocessor commands in a C program.
   b. Write a regexp to match all the lines in a C program except preprocessor commands
   c. Write a regexp to find line in a C program with trailing white space - one or more white space at the end of line
   d. Write a regexp to match the names Barry, Harry, Larry and Parry
   e. Write a regexp to match a string containing the word `hello` followed later by the word `world`
   f. Write regexp to match the word `calendar` and all mis-spellings with 'a' replaced 'e' or vice-versa
   g. Write regexp to match a list of positive integers separated by commas, e.g. `2,4,8,16,32`
   h. Write regexp to match a C string whose last character is newline

   a. `^#`
   b. `^[^#]`
   c. `\s$`
   d. `[BHLP]arry` or `Barry|Harry|Larry|Parry`
   e. `hello.*world`
   f. `c[ae]l[ae]nd[ae]r`
   g. `[1-9][0-9]*(,[1-9][0-9]*)` (`[0-9]+(,[0-9]+)*` would match 0 as well)
   h. `"[^"]*\\n"`

6. Give five reasons why this attempt to search a file for HTML paragraph and break tags may fail.

```
$ grep <p>|<br> /tmp/index.html
```

   Give egrep commands that will work.

   The characters '<', '>' and '|' are part of the shell's syntax (meta characters) and the shell will interpret them rather than passing them to grep. This can be avoided with singleor double-quotes or backslash, e.g:

```
$ egrep '<p>|<br>' /tmp/index.html
$ egrep "<p>|<br>" /tmp/index.html
$ egrep \<p>\|<br\> /tmp/index.html
```

For historical reasons 'grep' doesn't implement alternation ('|'). You need to use 'egrep' or ('grep -E') instead to get the full power of regular expressions.

The supplied regular expression won't match the HTML tags if they are in upper case (A-Z), e.g: <P>. The match can be case insensitive by changing the regular expression or using grep's -i flag

```
$ egrep  '<[pP]>|<[bB][rR]>' /tmp/index.html
$ egrep -i '<p>|<br>' /tmp/index.html
```

The supplied regular expression also won't match HTML tags containing spaces, e.g: <p >. This can be remedied by changing the regular expression or using wgrep's -w flag.

```
$ egrep -i '< *(p|br) *>' /tmp/index.html
$ egrep -iw '<p>|<br>' /tmp/index.html
```

The HTML tag may contain attributes, e.g: <p class="lead_para">. Again can be remedied by changing the regular expression or using egrep's -w flag.

```
$ egrep -i '<(p|br)[^>]*>' /tmp/index.html
```

This will still match <pre>. This can be avoided using a more complex regex:

```
$ egrep -i '<(p|br)( [^>]*)*>' /tmp/index.html
```

The HTML tag might contain a newline. This is more difficult to handle with an essentially line-based tool like egrep.

7. For each of the regular expression below indicate how many different strings the pattern matches and give some example of the strings it matches. If possible these example should include the shortest string and the longest string.

a. `Perl`

b. `Pe*r*l`

c. `Full-stop.`

d. `[1-9][0-9][0-9][0-9]`

e. `I (love|hate) programming in (Perl|Python) and (Java|C)`

Regexp	**N Matches**	**Shortest Match**	**Longest Match**	**Other Examples**
Perl	1	Perl	Perl	Perl
Pe*r*l	Arbitrary	Pl		Pel Prl Perl
Full-stop.	same as number of characters in character set	constant length	constant length	Full-stopa Full-stopb Full-stopc
[1-9][0-9][0-9][0-9]	9000	constant length	constant length	1000 1001 1002
I (love|hate) programming in (Perl|Python) and (Java|C)	8	I love programming in Perl and C	I love programming in Python and Java	

8. This regular expression [0-9]*.[0-9]* is intended to match floating point numbers such as '42.5'. Is it appropriate?

No. The regular expression [0-9]*.[0-9]* matches strings which are not floating point numbers. It will match zero or more digits, any character, followed by zero or more digits. It also will match numbers such as 01.12
Better would be [1-9][0-9]*\.[0-9]+

9. What does the command `egrep -v .` print and why?
Give an equivalent egrep command with no options, in other words without the -v and with a different pattern.

The pattern '.' matches any character.
The option -v causes egrep to print lines which don't match the pattern

So the command `egrep -v .` prints all the empty lines in its input.

The command `egrep '^$'` would also do this.

10. Write an egrep command which will print any lines in a file `ips.txt` containing an IP addresses in the range 129.94.172.1 to 129.94.172.25

```
$ egrep '129\.94\.172\.([1-9]|1[0-9]|2[0-5])$' ips.txt
```

Or, more generally

```
$ egrep '\b129\.94\.172\.([1-9]|1[0-9]|2[0-5])\b' ips.txt
```

11. For each of the scenarios below
    - describe the strings being matched using an English sentence
    - give a POSIX regular expression to match this class of strings

    In the examples, the expected matches are highlighted in bold.

    a. encrypted password fields (including the surrounding colons) in a Unix password file entry, e.g.

```
root:ZHolHAHZw8As2:0:0:root:/root:/bin/bash
jas:nJz3ru5a/44Ko:100:100:John Shepherd:/home/jas:/bin/bash
```

`:[^:]+:`

Since encrypted passwords can contain just about any character (except colon) you could structure the pattern as "find a colon, followed by a sequence of non-colons, terminated by a colon". Note that this pattern actually matches all of the fields in the line except the first and last, but if we assume that we only look for the first match on each line, it will do.

b. positive real numbers at the start of a line (using normal fixed-point notation for reals, *not* the kind of scientific notation you find in programming languages), e.g.

```
3.141 value of Pi
90.57 maximum hits/sec
half of the time, life is silly
0.05% is the legal limit
42 - the meaning of life
this 1.333 is not at the start
```

`^[0-9]+(\.[0-9]*)?`

This pattern assumes that real numbers will consist of a sequence of digits (the integer part) optionally followed by a decimal point with the fraction digits after the decimal point. Note the use of the `^` symbol to anchor the pattern to the start of the line, the `+` to ensure that there is at least one digit in the integer part, the `\` to escape the special meaning of `.`, and the `?` to make the fractional part optional.

c. Names as represented in this file containing details of tute/lab enrolments:

```
    2134389|Wang, Duved Seo Ken         |fri15-spoons|
    2139656|Undirwaad, Giaffriy Jumis   |tue13-kazoo|
    2154877|Ng, Hinry                   |tue17-kazoo|
    2174328|Zhung, Yung                 |thu17-spoons|
    2234136|Hso, Men-Tsun               |tue09-harp|
    2254148|Khorme, Saneu               |tue09-harp|
    2329667|Mahsin, Zumel               |tue17-kazoo|
    2334348|Trun, Toyin Hong Recky      |mon11-leaf|
    2336212|Sopuvunechyunant, Sopuchue  |mon11-leaf|
    2344749|Chung, Wue Sun              |fri09-harp|
    ...
```

`[^|,]+, [^|]+`

To pick out the content without the delimiters, the first part of the name is any string without a comma or bar, then the comma and space, and then everything up to the next delimiter. Both parts of the name are non-empty, hence `+` is used rather than `*`.

d. Names as above, but without the trailing spaces (difficult). *Hint:* what are given names composed of, and how many of these things can there be?

    `[^|,]+,( [^| ]+)+`

    We couldn't just say `[^| ]+`, because that would disallow spaces inside the given names. For a space to be accepted, it has to be followed by a non-space (usually a letter). Hence the given name portion is one or more sequences of W, where W is a space followed by non-spaces and non-bars.

    When a regular expression starts to look like stupid smiley icons, you know it's complex.

12. Consider the following columnated (space-delimited) data file containing marks information for a single subject:

```
2111321 37 FL
2166258 67 CR
2168678 84 DN
2186565 77 DN
2190546 78 DN
2210109 50 PS
2223455 95 HD
2266365 55 PS
...
```

Assume that the student number occurs at the beginning of the line, that the file is sorted on student number, and that nobody scores 100.

a. Give calls to the `sort` filter to display the data:
   i. in order on student number

```
$ sort < dataFile
```

   ii. in ascending order on mark

```
$ sort -k2n < dataFile
```

   iii. in descending order on mark

```
$ sort -k2nr < dataFile
```

b. Write calls to the `egrep` filter to select details of:
   i. students who failed

```
$ egrep 'FL' < dataFile
```

   ii. students who scored above 90

```
$ egrep ' 9[0-9] ' < dataFile
```

   (although the `'[0-9] '` isn't strictly necessary)

   iii. students with even student numbers

```
$ egrep '^[0-9]*[02468] ' < dataFile
```

   (even numbers end in an even digit)

c. Write a pipeline to print:
   i. the details for the top 10 students (ordered by mark)

```
$ sort -k2nr < dataFile | head
```

   or

```
$ sort -k2n < dataFile | tail
```

ii. the details for the bottom 5 students (ordered by mark)

```
$ sort -k2n < dataFile | head -5
```

or

```
$ sort -k2nr < dataFile | tail -5
```

d. Assuming that the command `cut -d' ' -f 3` can extract just the grades (`PS`, etc.), write a pipeline to show how many people achieved each grade (i.e. the grade distribution). E.g. for the above data:

```
      1 CR
      3 DN
      1 FL
      1 HD
      2 PS
```

```
$ cut -d' ' -f 3 < dataFile | sort | uniq -c
```

13. Consider the following text file containing details of tute/lab enrolments:

```
2134389|Wang, Duved Seo Ken          |fri15-spoons|
2139656|Undirwaad, Giaffriy Jumis    |tue13-kazoo|
2154877|Ng, Hinry                    |tue17-kazoo|
2174328|Zhung, Yung                  |thu17-spoons|
2234136|Hso, Men-Tsun                |tue09-harp|
2254148|Khorme, Saneu                |tue09-harp|
2329667|Mahsin, Zumel                |tue17-kazoo|
2334348|Trun, Toyin Hong Recky       |mon11-leaf|
2336212|Sopuvunechyunant, Sopuchue   |mon11-leaf|
2344749|Chung, Wue Sun               |fri09-harp|
...
```

Assuming that the file is called `enrolments`, write pipelines to answer each of the following queries:

a. Which tute is Hinry Ng enrolled in?

```
$ egrep 'Ng, Hinry' enrolments | cut -d'|' -f3
```

b. How many different tutorials are there?

```
$ cut -d'|' -f3 enrolments | sort | uniq | wc -l
```

c. What is the number of students in each tute?

```
$ cut -d'|' -f3 enrolments | sort | uniq -c
```

d. Are any students enrolled in multiple tutes?

```
$ sort enrolments | uniq | cut -d'|' -f1,2 | uniq -c | egrep -v '^ *1      '
```