Week 01 Tutorial Sample Answers

1. How will online/face-to-face tuts and labs work?

> **ANSWER:**
>
> Answered in tute.

2.
   - What is your tutor's name and e-mail?
   - How long have they been at UNSW?
   - What are they studying?
   - Do they have a pet?
   - What is 1 interesting thing about them?

> **ANSWER:**
>
> Answered in tute.

3.
   - What are your class mates's names,
   - What are they each studying,
   - What is 1 interesting thing about each of them?

> **ANSWER:**
>
> Answered in tute.

4. What is an operating system?

> **ANSWER:**
>
> 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.

5. What operating systems are being used in your house/tutorial room?

> **ANSWER:**
>
> Operating systems in your room might include:
>
> - (Linux) Ubuntu, Debian, Arch, Gentoo, Red Hat, Fedora, Android, SteamOS (Arch derivative), etc, etc.
> - (Darwin (Apple)) macOS (previously: OS X (previously: mac OS X)), iOS, iPadOS, WatchOS, etc, etc.
> - (Windows (Microsoft)) Windows, Windows Phone, Windows Mobile, Xbox, etc, etc.
> - (BSD) FreeBSD, NetBSD, OpenBSD, PlaystationOS (FreeBSD derivative) etc.
>
> Family Tree.

6. What operating system(s) do CSE lab computers run?

> **ANSWER:**
>
> CSE's lab computers and servers run (Linux) Debian (currently version 4.19).
>
> This web server is currently running:
>
> ```
> $ uname -a
> Linux elgar 4.9.0-19-amd64 #1 SMP Debian 4.9.320-2 (2022-06-30) x86_64 GNU/Linux
> ```

7. Write a regex to match:

   a. C preprocessor commands in a C program source file.
   b. All the lines in a C program except preprocessor commands.
   c. All lines in a C program with trailing white space (one or more white space at the end of line).
   d. The names "Barry", "Harry", "Larry" and "Parry".
   e. A string containing the word "hello" followed, some time later, by the word "world".
   f. The word "calendar" and mis-spellings where 'a' is replaced with 'e' or vice-versa.
   g. A list of positive integers separated by commas, e.g. `2,4,8,16,32`
   h. A C string whose last character is newline.

   ANSWER:

   a.
   ```
   ^#
   ```
   b.
   ```
   ^[^#]
   ```
   c.
   ```
   \s$
   ```
   d.
   ```
   [BHLP]arry
   ```
   e.
   ```
   hello.*world
   ```
   f.
   ```
   c[ae]l[ae]nd[ae]r
   ```
   g.
   ```
   [1-9][0-9]*(,[1-9][0-9]*)*
   ```
   h.
   ```
   "[^"]*\\n"
   ```

   Note: this doesn't handle escaped quotes.

8. When should you use:

   - `fgrep` / `grep -F`
   - `grep` / `grep -G`
   - `egrep` / `grep -E`
   - `pgrep` / `grep -P`

   ANSWER:

   - `fgrep` and the same command `grep -F`.
     `fgrep` is the old command and is deprecated.
     `grep -F` should be used instead.

     `grep -F` just does string matching - not regular expression matching.
     -F stands for *Fixed String*.

     In fixed strings no characters have a special meaning.
     Each character matches exactly itself.

     Fixed string matching is faster than regular expression matching.

     Use `grep -F` when you don't need the power regular expressions.
     Eg. When you want to find a literal word in a file
     Eg. When you need to find characters in a file that would otherwise have special meaning.
   - `grep` and the same command `grep -G`.
     -G exists to reset `grep` to it's default mode after it has been changed.
     Eg. `grep -F -G`.

     `grep` implements basic regular expressions BRE (Basic Regular Expressions).

It does so for historic reasons to do with the available processing power on old computers.
Most of the time you don't want to use `grep`.
You probably want to use `grep -E` as your goto command.

- `egrep` and the same command `grep -E`.
  `egrep` is the old command and is deprecated.
  `grep -E` should be used instead.

  `grep -E` implements extended regular expressions ERE (Extended Regular Expressions).

  You probably want to use `grep -E` when you need regular expressions.

- `pgrep` isn't related to the other commands in the `grep` family.
  "*pgrep* - look up processes based on name and other attributes."

  `grep -P` implements perl compatible regular expressions PCRE (Perl Compatible/Compliant Regular Expressions).

  `grep -P` is rarely needed/used to do clever/complicated things that BRE and ERE cannot do.

9. grep takes many options (see the manual page for *grep*).

```
$ man 1 grep
```

Give 3 (or more) simple/important options grep takes and explain what they do.

ANSWER:

- For:
  - `-F, --fixed-strings`
  - `-G, --basic-regexp`
  - `-E, --extended-regexp`
  - `-P, --perl-regexp`

  See the previous question.
- `-i, --ignore-case` ignores differences in (upper/lower) case when matching characters.

  Very useful as you often want this behaviour.
- `-v, --invert-match` non-matching lines are printed.

  Very useful because it is often easier to define a regex for lines you don't want.
- `-c, --count` prints the number of lines containing a match.

  Very useful for finding how common a pattern is.
- `-w, --word-regexp` prints lines where the regex matches a whole word.

  Very useful for example for matching variables names in code.

  This is the same as adding `\b` to the start and end of your regex.
- `-x, --line-regexp` prints lines where the regex matches the whole line.

  Very useful as it is common to write an regexp to exactly match the entire line.

  This is the same as adding `^` and `$` to the start and end of your regex respectively.
- Other useful options: -o/-s/-q, -A/-B/-C, -H/-n/-b

10. Why does this command seem to be taking a long time to run:

```
$ grep -E hello
```

ANSWER:

`grep -E` is reading from stdin, printing strings that contain **hello**

`grep -E`, if it is not given any filenames as arguments, reads from stdin.

11. Why won't this command work:

```
$ grep -E int main program.c
```

ANSWER:

`grep -E` will attempt to search files **main** and **program.c** for lines containing the string **int**

This is because space has a special meaning to the shell - it separates arguments.

This will do what is likely intended - search **program.c** for lines containing the string **int main**

```
$ grep -E 'int main' program.c
```

12. 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 a `grep` command that will work.

ANSWER:

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 `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

```
$ grep -E '<[pP]>|<[bB][rR]>' /tmp/index.html
$ grep -Ei '<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

```
$ grep -Ei '< *(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.

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

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

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

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

13. 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)
```

ANSWER:

Regexp	N Matches	Shortest Match	Longest Match	Other Examples
Perl	1	"Perl"	"Perl"	
Pe*r*l	Arbitrary	"Pl"	Arbitrary	"Pel" "Prl" "Perl" "Peeeeeel" "Peerrrrrrl"
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"	"I hate programming in Perl and Java"

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

ANSWER:

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

A Better expression would be `(0|[1-9][0-9]*)\.([0-9]*[1-9]|0)`

15. What does the command `grep -Ev .` print and why?

    Give an equivalent `grep -E` command with no options,
    in other words: without the `-v` .

ANSWER:

The pattern `.` matches any character.

The option `-v` causes `grep` to print lines which don't match the pattern

So the command `grep -Ev .` prints all the empty lines in its input.

The command `grep -E '^$'` would also do this.

16. Write a `grep -E` 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`

ANSWER:

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

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

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/zsh
andrewt:ugGYU6GUJyug2:101:101:Andrew Taylor:/home/jas:/bin/dash
```

ANSWER:

```
:[^:]+:
```

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
```

ANSWER:

```
^[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|
...
```

ANSWER:

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

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?

ANSWER:

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

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.
