Appendix A

Answers to the exercises

All work for these was done on `ursa`, one of the C.U.S. machines running Solaris 2.3. There may be minor differences between output generated on `ursa` and other machines, especially regarding the contents of the file `/usr/dict/words`.

Answer to exercise 1:

```
$ grep hack wizard
of those who came before thee.  Hack, and hack
best hacks to the Net and gain in Repute
```

Answer to exercise 2: The file `wizard` has the string "`limb`" as part of the word "`limbo`".

```
$ grep limb virgin wizard
wizard:Delay, or linger ever in the ghastly limbo called
```

Answer to exercise 3:

```
$ grep -i with virgin
Helpline office. We tapped in with our regular BA
In common with many other airlines, Virgin rents
provided BA with critical information on Virgin's
monorail to the South Terminal. With white
with a special card to prevent hacking.''
the critical Passenger Name List with details of
$ grep -i with wizard
long, and winding, and fraught with Risks.  Thou
must Attune thyself with the Source, attaining
If thou desirest with True Desire to tread the
_Life_With_UNIX_ will set thy Feet unerringly
ed., with Guy L. Steele Jr.).
by mere Study but must become One with it through
O Nobly Born: learn, and seek within thyself.
travellest with high heart and pure intention,
that way lies only Bogosity without End.
```

```
$ grep -i with virgin wizard
virgin:Helpline office. We tapped in with our regular BA
virgin:In common with many other airlines, Virgin rents
virgin:provided BA with critical information on Virgin's
virgin:monorail to the South Terminal. With white
virgin:with a special card to prevent hacking.''
virgin:the critical Passenger Name List with details of
wizard:long, and winding, and fraught with Risks.  Thou
wizard:must Attune thyself with the Source, attaining
wizard:If thou desirest with True Desire to tread the
wizard:_Life_With_UNIX_ will set thy Feet unerringly
wizard:ed., with Guy L. Steele Jr.).
wizard:by mere Study but must become One with it through
wizard:O Nobly Born: learn, and seek within thyself.
wizard:travellest with high heart and pure intention,
wizard:that way lies only Bogosity without End.
```

Answer to exercise 4:

```
$ grep hack virgin
For the next nine months the Helpline hackers
information they were hacking out of BABS. The
with a special card to prevent hacking.''
$ grep -w hack virgin
```

Answer to exercise 5: The word "`we`" occurs twice in `virgin` with both instances on the same line. It does not occur at all in `wizard`.

```
$ grep -w we virgin wizard
virgin:``Had we only had the airline, we would have gone
```

Answer to exercise 6:

```
$ grep -vc e virgin wizard
virgin:22
wizard:34
```

Answer to exercise 7: In the first example, "`grep [Ww]izard wizard`", the shell expands the string "`[Ww]izard`" into the list of filenames that match it. Only the single filename `wizard` matches it so the command that is run is

```
$ grep wizard wizard
```

As there are no instances of "`wizard`" (with a lowercase "`w`") in the file `wizard`, no output is produced.

In the second example, "`grep '[Ww]izard' wizard`", the shell is not allowed to perform the filename expansion on "`[Ww]izard`" because it is quoted. As a result, `grep` is called to look for the regular expression, not its expansion by the shell. As the string "`Wizard`" (with an uppercase "`W`") does occur in the file `wizard`, output is produced.

Answer to exercise 8: A digit will be matched by the regular expression "`[0-9]`" so a pair of digits will match "`[0-9][0-9]`".

```
$ grep '[0-9][0-9]' wizard
```

Answer to exercise 9:

```
$ grep -c '[-y]' virgin
35
```

Answer to exercise 10:

```
$ grep -ic '[^aeiou][^aeiou][^aeiou][^aeiou]\
[^aeiou][^aeiou]' /usr/dict/words
23
$ grep -ic '[^aeiouy][^aeiouy][^aeiouy][^aeiouy]\
[^aeiouy][^aeiouy]' /usr/dict/words
3
```

Answer to exercise 11:

```
$ grep -c 's[aeiou]\{2,\}s' /usr/dict/words
25
```

Answer to exercise 12:

```
$ grep -c 'y[a-z]*y[a-z]*y' /usr/dict/words
4
```

Answer to exercise 13:

```
$ grep -ci '^[^aeiou]\{6\}' /usr/dict/words
12
```

Answer to exercise 14:

```
$ grep -ic '^z[a-z]*a$' /usr/dict/words
4
```

Answer to exercise 15:

```
$ grep -c '\<[AEIOU]' virgin
16
```

Answer to exercise 16:

```
$ grep -ic '\([aeiou]\)\1' /usr/dict/words
1018
```

Answer to exercise 17:

```
$ sed -e 's/oranges/grapes/g' < fruit
apples apples
bananas bananas
lemons limes
grapes grapes
pears pears
```

Answer to exercise 18: The trick with this question is to turn *all* the instances of "`apples`" into "`plums`" and then to turn the first one back again:

```
$ sed -e 's/apples/plums/g' -e 's/plums/apples/' < fruit
apples plums
bananas bananas
lemons limes
oranges oranges
pears pears
```

Answer to exercise 19:

```
$ sed -e 's/\<[a-z][aeiou][a-z]*\>/plums/g' < fruit
apples apples
plums plums
plums plums
oranges oranges
plums plums
```

Answer to exercise 20:

```
$ sed -e 's/\([a-z]*\) \([a-z]*\)/\2 \1/' < fruit
apples apples
bananas bananas
limes lemons
oranges oranges
pears pears
```

Answer to exercise 21: Converting the previous answers to scripts is easy. Just remember a few points:

1. The first line of the file must be "`#!/bin/sed -f`". Don't forget the "`-f`".
2. The file must be executable. (Use "`chmod a+x` *`filename`*" to make a file executable by anyone.)
3. The quotes are not `sed` syntax; they are just to protect things from the shell.
4. Multiple `sed` instruction that would have required more than one use of "`-e`" simply sit on different lines.

In this answer I have assumed that the files are all called `answern`:

```
$ ls -l answer?
-rwxr-xr-x   1 rjd4     uxsup         34 May  4 14:58 answer1
-rwxr-xr-x   1 rjd4     uxsup         47 May  4 14:58 answer2
-rwxr-xr-x   1 rjd4     uxsup         48 May  4 14:59 answer3
-rwxr-xr-x   1 rjd4     uxsup         46 May  4 15:00 answer4
$ cat answer1
#!/bin/sed -f
s/oranges/grapes/g
$ cat answer2
#!/bin/sed -f
s/apples/plums/g
s/plums/apples/
$ cat answer3
#!/bin/sed -f
s/\<[a-z][aeiou][a-z]*\>/plums/g
$ cat answer4
#!/bin/sed -f
s/\([a-z]*\) \([a-z]*\)/\2 \1/
```

Answer to exercise 22:

```
$ ls -l insert2
-rwxr-xr-x   1 rjd4     uxsup         67 May  4 15:10 insert2
$ cat insert2
#!/bin/sed -f
/lemons/i\
An inserted line\
A second inserted line
$ ./insert2 < fruit
apples apples
bananas bananas
An inserted line
A second inserted line
lemons limes
oranges oranges
pears pears
```

Answer to exercise 23:

```
$ sed -e '/^\([a-z]*\) \1$/d' < fruit
lemons limes
```

Note the use of the anchors. Without them the "`lemons limes`" line would also have matched because the "`[a-z]*`" expression could have matched *no* characters, "`\1`" would have been the empty string and the space in the middle of the line would have matched the expression.

```
$ sed -e '/\([a-z]*\) \1/d' < fruit
```

Answer to exercise 24:

```
$ awk '$3>100 {print $1}' < table
```

Answer to exercise 25: There are two possible answers here. The format strings "`%4d%5d\n`" and "`%4d %4d\n`" will both generate the correct output for this file `table`. The strings only differ in their behaviour if the value in column 3 ever reaches 5 columns. If it does the former will continue to print the output in 9 columns overall but will run the two numbers together. This is quite satisfactory if the data is being passed on to a program whose input is defined on a character-by-character basis. (Consider Fortran's "`FORMAT`" statements for an example of this.) The latter format string will cause the five digit number to overrun into the tenth column of text but will guarantee a space between the two columns of data. This is suitable for reading by eye and for programs whose input is white-space delimited. (Consider C's "`scanf()`" function for an example of this.)

```
$ awk '{printf "%4d %4d\n", $2, $3}' < table
  22   44
   2    7
  71  138
 139 1204
...
```

Answer to exercise 26:

```
$ awk '$3>100 print $0' < table
CS              71      138
CUS             139     1204
Chemistry       50      102
Materials       65      131
Thor            111     202
```

Answer to exercise 27: Again, a variety of format strings is possible depending on whether the splitting whitespace or the columns used is more important. Again, this answer inserts explicit separating whitespace.

```
$ awk '{printf "%4d %4d %8.3f\n", $2, $3, $3/$2}' < table
  22   44    2.000
   2    7    3.500
  71  138    1.944
 139 1204    8.662
...
```

Answer to exercise 28: The `-n` option on `cat` numbers each line. Note that `cat` leaves 6 characters for the line number followed by two spaces and the line itself.

```
$ cat -n table
     1  Archaeology     22      44
     2  Biotechnology   2       7
     3  CS              71      138
     4  CUS             139     1204
...
$ awk '{printf "%6d  %s\n", NR, $0}' < table
     1  Archaeology     22      44
     2  Biotechnology   2       7
     3  CS              71      138
     4  CUS             139     1204
...
```

Answer to exercise 29:

```
$ awk '$1~/^[AEIOU]/ && $2>10 {print $1}' < table
Archaeology
```