1. Imagine that we have just typed a shell script into the file `my_first_shell_script.sh` in the current directory. We then attempt to execute the script and observe the following:

```
$ my_first_shell_script.sh
my_first_shell_script.sh: command not found
```

Explain the possible causes for this, and describe how to rectify them.

- **problem:** you might not have the current directory in your `PATH`
  solution: add `.` to the end of your `PATH` (via `PATH=$PATH:.`)
  or type the command name as `./my_first_shell_script.sh`
- **problem:** the `my_first_shell_script.sh` file might not be executable
  solution: make the file executable (via `chmod +x my_first_shell_script.sh`)
  or execute it via the command `sh my_first_shell_script.sh` (also fixes the first problem)
- **problem:** you might have gotten the `#!/bin/sh` line wrong
  solution: check the line to make sure there are no spurious spaces or spelling mistakes and then check that the shell is actually called `/bin/sh` on your system
- **problem:** the `my_first_shell_script.sh` file has been transferred from a Windows-based computer in binary mode, and there's a `^M` (`'\\r'` in the C context) after `/bin/sh`
  solution: run the standard command `dos2unix MyScript` which will remove the pesky `^M`s.

Note that some of these problems might also result in the message: `my_first_shell_script.sh: Permission denied`, depending on which shell you're using.

2. Implement a shell script called `seq.sh` for writing sequences of integers onto its standard output, with one integer per line. The script can take up to three arguments, and behaves as follows:
   - **seq.sh *LAST*** writes all numbers from 1 up to ***LAST*** inclusive, for example:

```
$ ./seq.sh 5
1
2
3
4
5
```

   - **seq.sh *FIRST LAST*** writes all numbers from ***FIRST*** up to ***LAST*** inclusive, for example:

```
$ ./seq.sh 2 6
2
3
4
5
6
```

   - **seq *FIRST INCREMENT LAST*** writes the sequence ***FIRST***, ***FIRST + INCREMENT***, ***FIRST + 2 * INCREMENT***, up to ***p*** (where ***p*** is the largest integer in this sequence that is less than or equal to ***LAST***), for example:

```
$ ./seq.sh 3 5 24
3
8
13
18
23
```

Sample solution for seq.sh

```
#!/bin/sh
# Write the sequence of integers specified by command-line arguments
# /usr/bin/seq provides similar functionality on Linux

if test $# = 1
then
    first=1
    increment=1
    last=$1
elif test $# = 2
then
    first=$1
    increment=1
    last=$2
elif test $# = 3
then
    first=$1
    increment=$2
    last=$3
else
    cat <<EOI
Usage:
  $0 last         ... prints numbers in range 1..last
  $0 first last      ... prints numbers in range first..last
  $0 first 2nd last  ... prints numbers first,2nd,..last
                    (using difference between first and
                     2nd as the increment)
EOI
   exit 1
fi

if test $increment -lt 1
then
    echo "$0: invalid increment value: $increment"
    exit 1
fi

i=$first
while test $i -le $last
do
    echo $i
    i=`expr $i + $increment`
done
```

Another sample solution for seq.sh

```
#!/bin/bash
# Write the sequence of integers specified by command-line arguments
# /usr/bin/seq provides similar functionality on Linux

# The bash-specific syntax ((..)) is used for arithmetic
# This increases the readability but reduces the portability

if (($# == 1))
then
    first=1
    increment=1
    last=$1
elif (($# = 2))
then
    first=$1
    increment=1
    last=$2
elif (($# = 3))
then
    first=$1
    increment=$2
    last=$3
else
    cat <<EOI
Usage:
  $0 last          ... prints numbers in range 1..last
  $0 first last      ... prints numbers in range first..last
  $0 first 2nd last  ... prints numbers first,2nd,..last
                     (using difference between first and
                      2nd as the increment)
EOI
   exit 1
fi

if ((increment < 1))
then
    echo "$0: invalid increment value: $increment"
    exit 1
fi

i=$first
while ((i <= last))
do
    echo $i
    i=$((i + increment))
done
```

3. Write a shell script, **no_blinking.sh**, which removes all HTML files in the current directory which use the blink element:

```
$ no_blinking.sh
Removing old.html because it uses the <blink> tag
Removing evil.html because it uses the <blink> tag
Removing bad.html because it uses the <blink> tag
```

```
#!/bin/sh
# Removes all HTML files in the current directory which use <blink>

for file in *.html
do
    # note use of -i to ignore case and -w to ignore white space
    # however tags containing newlines won't be detected
    if egrep -iw '</?blink>' $file >/dev/null
    then
        echo  "Removing $file because it uses the <blink> tag"
        rm "$file"
    fi
done
```

Modify the shell script to instead take the HTML files to be checked as command line arguments and instead of removing them, add the suffix **.bad** to their name.

```
$ no_blinking.sh awful.html index.html terrible.html
Renaming awful.html to awful.html.bad because it uses the <blink> tag
Renaming terrible.html to terrible.html.bad because it uses the <blink
> tag
```

```
#!/bin/sh
# Removes all HTML files supplied as argument which use <blink>

for file in "$@"
do
    # note use of -i to ignore case and -w to ignore white space
    # however tags containing newlines won't be detected
    if egrep -iw '</?blink>' >/dev/null
    then
        echo  "Rename $file to $file.bad because it uses the <blink> tag"
        mv "$file" "$file.bad"
    fi
done
```

4. Write a shell script, **list_include_files.sh**, which for all the C source files (`.c` files) in the current directory prints the names of the files they include (`.h` files), for example

```
$ list_include_files.sh
count_words.c includes:
    stdio.h
    stdlib.h
    ctype.h
    time.h
    get_word.h
    map.h
get_word.c includes:
    stdio.h
    stdlib.h
map.c includes:
    get_word.h
    stdio.h
    stdlib.h
    map.h
```

```
#!/bin/sh
# list the files included by the C sources files included as arguments

for file in *.c
do
    echo "$file includes:"
    egrep '^#include' "$file"|  # find '#include lines
    sed 's/[">][^">]*$//'|      # remove the last '"' or '>' and anything after it
    sed 's/^.*["<]/    /'       # remove the first '"' or '>' and anything before it
done
```

5. Consider the following columnated (space-delimited) data file containing contact information for various CSE academic staff:

```
G Heiser        Newtown      9381-1234
S Jha           Kingsford    9621-1234
C Sammut        Randwick     9663-1234
R Buckland      Randwick     9663-9876
J A Shepherd    Botany       9665-4321
A Taylor        Glebe        9692-1234
M Pagnucco      North Ryde   9868-6789
```

Note: This data is fictitious. Do not ring these phone numbers. I have no idea whether they are real or not, but they are certainly not the correct phone numbers for the academic staff mentioned.

The data is currently sorted in phone number order. Can we use the `sort` filter to re-arrange the data into "telephone-book" order? If not, how would we need to change the file in order to achieve this?

No. We need to sort based on the family name field, but this occurs in different positions on each line of the file, depending how many initials the person has.
If not, how would we need to change the file in order to achieve this?

We need to make sure that the family name occurs in the same "field" on each line. One possiblity, make it the first field in each line, e.g.

```
Keller G       Newtown      9381-1234
Wilson W H     Kingsford    9621-1234
...
```

Another possibility, make sure that the initials form a single field, e.g.

```
G. Keller      Newtown      9381-1234
W.H. Wilson    Kingsford    9621-1234
...
```

6. Consider the Unix password file (`/etc/passwd`):

```
root:ZHolHAHZw8As2:0:0:root:/root:/bin/bash
jas:iaiSHX49Jvs8.:100:100:John Shepherd:/home/jas:/bin/bash
andrewt:rX9KwSSPqkLyA:101:101:Andrew Taylor:/home/andrewt:/bin/cat
postgres::997:997:PostgreSQL Admin:/usr/local/pgsql:/bin/bash
oracle::999:998:Oracle Admin:/home/oracle:/bin/bash
cs2041:rX9KwSSPqkLyA:2041:2041:COMP2041 Material:/home/cs2041:/bin/bash
cs3311:mLRiCIvmtI9O2:3311:3311:COMP3311 Material:/home/cs3311:/bin/bash
cs9311:fIVLdSXYoVFaI:9311:9311:COMP9311 Material:/home/cs9311:/bin/bash
cs9314:nTn.JwDgZE1Hs:9314:9314:COMP9314 Material:/home/cs9314:/bin/bash
cs9315:sOMXwkqmFbKlA:9315:9315:COMP9315 Material:/home/cs9315:/bin/bash
```

Provide a command that would produce each of the following results:

a. display the first three lines of the file
b. display lines belonging to class accounts
   (assume that their login name starts with either "cs", "se", "bi" or "en", followed by a digit)
c. display the user name of everyone whose shell is `/bin/bash`
d. create a tab-separated file `passwords.txt` containing only login name and password for all users

a. `head -3 /etc/passwd`
b. `egrep '^(cs|se|bi|en)[0-9]' /etc/passwd`
c. `egrep '/bin/bash' /etc/passwd | cut -d':' -f1`
   or a more accurate version that restricts the search to the right field:
   `cut -d':' -f1,7 /etc/passwd | egrep ':/bin/bash' | cut -d':' -f1`
d. `cut -d':' -f1,2 /etc/passwd | tr ':' '\t' > passwords.txt`

7. The following shell script emulates the `cat` command using the built-in shell commands `read` and `echo`.

```
#!/bin/sh
while read line
do
    echo "$line"
done
```

a. what are the differences between the above script and the real `cat` command?
b. modify the script so that it can concatenate multiple files from the command line, like the real `cat`

(Hint: shell control structures (e.g. `if`, `while`, `for`) are commands in their own right and can form a component of a pipeline)

a. Some differences
   - the script doesn't concatenate files named on the command line, just standard input
   - it doesn't implement all of the `cat` options
   - the appearance of lines may be altered (space at start of line is removed, and runs of multiple spaces will be compressed to a single space)
b. Shell script to concatenate multiple files specified on command line:

```
#!/bin/sh
for f in "$@"
do
    if [ ! -r "$f" ]
    then
        echo "No such file: $f"
    else
        while read line
        do
            echo "$line"
        done <$f
    fi
done
```

8. The `gzip` command compresses a text file and renames it to `oldName.gz`. The `zcat` command takes the name of a single compressed file as its argument and writes the original (non-compressed) text to its standard output.
   Write a shell script called `zshow` that takes multiple `.gz` file names as its arguments, and displays the original text of each file, separated by the name of the file.

   Consider the following example execution of `zshow`:

```
 $ zshow a.gz b.gz bad.gz c.gz
===== a =====
... original contents of file "a" ...
===== b =====
... original contents of file "b" ...
===== bad =====
No such file: bad.gz
===== c =====
... original contents of file "c" ...
```

   A simple solution which aims to make things obvious

```
#!/bin/sh
for f in "$@"   # for each command line arg
do
    f1=`echo $f | sed -e 's/\.gz//'`
    echo "===== $f1 ====="
    if test ! -r "$f"   # is the arg readable?
    then
        echo "No such file: $f"
    else
        zcat "$f"
    fi
done
```

   A solution that aims to be more robust

```
#!/bin/sh
for f in "$@"  # iterates over command line args
do
    f1=`echo $f | sed 's/\.gz//'`
    echo "===== $f1 ====="
    if test ! -r "$f"   # is the arg readable?
    then
        echo "No such file: $f"
    else
        ftype=`file -b $f | sed 's/ //'`
        if [ "$ftype" != "gzip" ]
            echo "Incorrect file type: $f"
        else
            zcat "$f"
        fi
    fi
done
```

   Notice that robustness typically adds a significant amount of code. The extra code is definitely worth it.
   The `file` command tells you what kind of file its argument is. If you don't know why we need to pipe its output through `sed`, read the relevant manual entries.

9. Consider the marks data file from last week's tutorial, and assume that it is stored in a file called `Marks`.

```
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 also that we have a file called `Students` that contains the names and student ids of for all students in the class, e.g.

```
2166258 Chen, X
2186565 Davis, PA
2168678 Hussein, M
2223455 Jain, S
2190546 Phan, DN
2111321 Smith, JA
2266365 Smith, JD
2210109 Wong, QH
...
```

Write a shell script that produces a list of names and their associated marks, sorted by name, e.g.

```
67 Chen, X
77 Davis, PA
84 Hussein, M
95 Jain, S
78 Phan, DN
37 Smith, JA
55 Smith, JD
50 Wong, QH
```

Note: there are many ways to do this, generally involving combinations of filters such as `cut`, `egrep`, `sort`, `join`, etc. Try to think of more than one solution and discuss the merits of each.

> One obvious strategy, iterate over the `Students` file using the shell's `read` command. We iterate over `Students` rather than `Marks`, since it's already in the order we want; we could iterate the other way, but then we'd have to sort the output afterwards. For each student, we can use `egrep` and `cut` (or `sed` or `awk` or `perl`) to extract their information from the `Marks` file
>
> ```
> #!/bin/sh
> while read sid name init
> do
>     mark=`egrep $sid Marks | cut -d' ' -f2`
>     echo $mark $name $init
> done <Students
> ```
>
> For the minimalists (and Haskell lovers), here's a one-liner:
>
> ```
> #!/bin/sh
> sort Students | join Marks - | sort -k4 | cut -d' ' -f2,4,5
> ```
>
> Note the use of the `-` to make the second argument to `join` come from standard input. Without this mechanism, we would need to create a temporary file containing a sorted copy of `Students`.

10. Implement a shell script called `grades.sh` that reads a sequence of (studentID, mark) pairs from its standard input and writes (studentID, grade) pairs to its standard output. The input pairs are written on a single line, separated by spaces, and the output should use a similar format. The script should also check whether the second value on each line looks like a valid grade, and output an appropriate message if it doesn't. The script can ignore any extra data occuring after the mark on each line.

    Consider the following input and corresponding output to the program:

    Input

    ```
    2212345 65
    2198765 74
    2199999 48
    2234567 50 ok
    2265432 99
    2121212 hello
    2222111 120
    2524232 -1
    ```

    Output

    ```
    2212345 CR
    2198765 CR
    2199999 FL
    2234567 PS
    2265432 HD
    2121212 ?? (hello)
    2222111 ?? (120)
    2524232 ?? (-1)
    ```

    To get you started, here is a framework for the script:

    ```
    #!/bin/sh
    while read id mark
    do
        # insert mark/grade checking here
    done
    ```

    Note that the shell's `read` operation assumes that the components on each input line are separated by spaces. How could we use this script if the data was supplied in a file that used commas to separate the (studentID, mark) components, rather than spaces?

    > Since the "mark to grade mapping problem" is a standard problem in first year tutes, working out the algorithm should not pose any problems. Hopefully the only tricky thing is getting the shell syntax right. The main aim of the exercise is to write a multiway selection statement.

We supply two solutions, one using `if`, the other using `case`. The `if` one is more natural for people who know how to program in languages like Java. The `case` version requires us to develop patterns to match all the possible inputs.

The `case` construct is a nice way for checking strings via patterns. The pattern used here catches both non-numbers and negative numbers (they start with a minus rather than a digit). Unfortunately, the `test` command doesn't support pattern-matching.

Shell supports a C-style `continue` construct for loops, which is used here to prevent processing non-numeric "mark" fields.

All of the bracket-style (`[...]`) syntax for tests could be replaced by the more conventional syntax for the `test` command, e.g. `test $mark -lt 50`.

Note that the `read` statement has 2 arguments to ensure that the mark is bundled in with the optional comment on each data line.

```
#!/bin/sh
while read stid mark extras
do
        case "$mark" in
        [0-9]*) ;;
        *)      echo "$stid ?? ($mark)"
                continue
                ;;
        esac
        if test $mark -lt 50
        then
                echo $stid FL
        elif test $mark -lt 65
        then
                echo $stid PS
        elif test $mark -lt 75
        then
                echo $stid CR
        elif test $mark -lt 85
        then
                echo $stid DN
        elif test $mark -le 100
    then
                echo $stid HD
    else
        echo "$stid ?? ($mark)"
        fi
done
```

Another possibility would to use `case` patterns to match the correct ranges of values, but this assumes that all marks are integer values. Floating point values could also be handled, but at the cost of making the patterns more complex. Also, this approach wouldn't scale up to arbitrary ranges of integers; it would become to messy to specify patterns for all possible numbers.

```
#!/bin/sh
while read stid mark extras
do
        case "$mark" in
        [0-9] | [0-4][0-9])
        echo $stid FL ;;
    5[0-9] | 6[0-4])
                echo $stid PS ;;
    6[5-9] | 7[0-4])
                echo $stid CR ;;
        7[5-9] | 8[0-4])
                echo $stid DN ;;
        8[5-9] | 9[0-9] | 100)
                echo $stid HD ;;
        *)
        echo "$stid ?? ($mark)" ;;
    esac
done
```

If the input file used comma as a separator, the easiest thing would be to run the input through `tr` to convert the commas to spaces and pipe the output into the `grades` program e.g.

```
tr ',' ' ' <data | grades
```

Alternatively, you could alter the shell's field separator via

```
IFS=,
```

11. Write a shell script `time_date.sh` that prints the time and date once an hour. It should do this until a new month is reached. Reminder the `date` command produces output like this:

```
Friday 5 August  17:37:01 AEST 2016
```

Sample solution for time_date.sh:

```
#!/bin/sh

start_month=`date | cut -d' ' -f3` # or use options to date, e.g. date +%m
while test $start_month = `date | cut -d' ' -f3`
do
    date
    sleep 3600 # i.e. one hour
done
```

Normally, we would double-quote like this:

```
while test "$start_month" = "`date | cut -d' ' -f3`"
```

but this is not needed since the month does not contain spaces

12. Consider a scenario where we have a directory containing two LaTeX files, `a.tex` and `b.tex`. The file `a.tex` is 20 lines long, and `b.tex` is 30 lines long. What is the effect of each of the commands below? How will their output differ?

```
$ wc -l *.tex
$ echo `wc -l *.tex`
```

The first command counts the number of lines in each file and writes this data out, one file per line, followed by a total. The result:

```
$ wc -l *.tex
 20 a.tex
 30 b.tex
 50 total
```

The second command starts out doing exactly the same thing as the first: counting lines in the two files. However, because of the backquotes, its output is written to a single string which is then taken by the shell and passed as arguments to the `echo` command. In the process of capturing the output, it is trimmed of trailing newlines. By the time the shell has processed it further and it is fed as arguments to the `echo` command, all newlines have been removed, and it becomes a sequence of words. The `echo` a single space.

```
$ echo `wc -l *.tex`
20 a.tex 30 b.tex 50 total
```

13. Write a shell script that displays the name and size of all files in the current directory that are bigger than (say) 100,000 bytes.

(Hint: use `wc` to do the counting, and capture its output using back-ticks. How do you get rid of the file name and/or line and word counts?)

```
#!/bin/sh
LIMIT=100000
for f in *
do
    bytes=`wc -c <"$f"`
    if test $bytes -gt $LIMIT
    then
        echo "$f has $bytes bytes"
    fi
done
```

14. What is the output of each of the following pipelines if the text

```
this is big Big BIG
but this is not so big
```

is supplied as the initial input to the pipeline?

a. `tr -d ' ' | wc -w`

This pipeline deletes (`-d`) all of the blanks between words, thus compressing each line into a single long word, which means that the number of words is the same as the number of lines (i.e. 2).

b. `tr -cs 'a-zA-Z0-9' '\n' | wc -l`

   This splits the input up so that there is one word of input on each line of output; counting the number of output lines, thus also counts the total number of words in the text (i.e. 11).

c. `tr -cs 'a-zA-Z0-9' '\n' | tr 'a-z' 'A-Z' | sort | uniq -c`

   This splits the input into words, then normalises them (by mapping all words to all upper-case), then counts the number of occurences of each distinct word. The output looks like:

```
      4 BIG
      1 BUT
      2 IS
      1 NOT
      1 SO
      2 THIS
```

15. Consider the standard "split-into-words" technique from the previous question:

```
tr -c -s 'a-zA-Z0-9' '\n' < someFile
```

    Explain how this command works (i.e. what does each argument do)

    - -c = 'complement' so it replaces everything NOT in string 1 with string 2
    - -s = 'squeeze' repeated characters, so it replaces any duplicate newlines with just one
    - the string `'a-zA-Z0-9'` is shorthand for the string of all alphahnumeric characters

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

16. Assume that we are in a shell where the following shell variable assignments have been performed, and `ls` gives the following result:

```
$ x=2  y='Y Y'  z=ls
$ ls
    a         b         c
```

    What will be displayed as a result of the following `echo` commands:

a. `$ echo a  b  c`

b. `$ echo "a  b  c"`

c. `$ echo $y`

d. `$ echo x$x`

e. `$ echo $xx`

f. `$ echo "$y"`

g. `$ echo '$y'`

h. `$ echo ` `$y` `

i. `$ echo ` `$z` `

j. `$ echo ` `echo a b c` `

The aim of this question is to clarify notions about command line arguments and the various transformations that the shell performs on the command line before executing it.
Recall that the shell performs command and variable substitution before splitting the command line into separate words to make up the arguments. Single-quotes and double-quotes perform a grouping function that overrrides the normal word-splitting.

Command + Output	**Explanation**

```
$ echo a   b
 c
a b c
```

Spaces between arguments are not preserved; `echo` uses only one space between args

```
$ echo "a   b
c"
a    b    c
```

Spaces are preserved because the quotes turns `"a  b  c"` into a single argument

```
$ echo $y
Y Y
```

`$y` expands into two separate args

```
$ echo x$x
x2
```

`$x` expands to `2` and is appended after the letter `x`

```
$ echo $xx
```

`$xx` is treated as a reference to the shell variable `xx`; since there is no such variable, it expands to the empty string

```
$ echo "$y"
Y Y
```

`$y` expands into a single argument

```
$ echo '$y'
$y
```

the single quotes prevent variable expansion

```
$ echo `$y`
Y: command
 not found
```

`$y` expands to `Y  Y` which is then executed as a command because of the backquotes; since there is no command `Y`, the error message follows

```
$ echo `$z`
a b c
```

`$z` expands to `ls` which is then executed as a command, giving the names of the files in the current directory, which are treated as three separate args

```
$ echo `echo
 a b c`
a b c
```

the inner `echo` command is executed, giving `a  b  c` which are passed as arguments to the outer `echo`

17. The following C program and its equivalent in Java both aim to give precise information about their command-line arguments.
    C:

```c
// Display command line arguments, one per line
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("#args  = %d\n", argc-1);
    for (i = 1; i < argc; i++)
        printf("arg[%d] = \"%s\"\n", i, argv[i]);
    return 0;
}
```

    Java:

```
public class args {
    public static void main(String args[]) {
        System.out.println("#args  = " + args.length);
        for (int i = 0; i < args.length; i++)
            System.out.println("arg[" + (i+1) + "] = \"" + args[i] + "\"");
    }
}
```

Assuming that the C program is compiled into a command called `args`, consider the following examples of how it operates:

```
$ args a b c
#args  = 3
arg[1] = "a"
arg[2] = "b"
arg[3] = "c"
$ args "Hello there"
#args  = 1
arg[1] = "Hello there"
```

Assume that we are in a shell where the following shell variable assignments have been performed.

```
$ x=2  y='Y Y'  z=ls
```

Assume that we are in a shell with the same variable assignments and the same current directory as the previous question. What will be the output of the following:

a. `$ args x y  z`

b. `$ args `ls``

c. `$ args $y`

d. `$ args "$y"`

e. `$ args `echo "$y"``

f. `$ args $x$x$x`

g. `$ args $x$y`

h. `$ args $xy`

This question has a similar aim to the previous one, but now we need to be more precise about what are the actual arguments that are being passed to the command. Note that we place double quotes around each argument so that we can see *exactly* what's contained in the argument, including any embedded spaces.

Command+Output	Explanation

`$ args x y z`
`#args  = 3`
`arg[1] = "x"`
`arg[2] = "y"`
`arg[3] = "z"`
	Each of the letters is a single argument (separated by spaces).

`$ args `ls``
`#args  = 3`
`arg[1] = "a"`
`arg[2] = "b"`
`arg[3] = "c"`
	The `ls` command is executed and its output is interploated into the command line; the shell then splits the command-line into arguments.

```
$ args $y
#args  = 2
arg[1] = "Y"
arg[2] = "Y"
```

$y expands to the string Y Y; when the shell splits the line into words, these two characters becomes separate args

```
$ args "$y"
#args  = 1
arg[1] = "Y Y"
```

$y expands to Y Y within the quotes, so it is treated as a single word when the shell breaks the line into args

```
$ args `echo $y`
#args  = 2
arg[1] = "Y"
arg[2] = "Y"
```

the command within the backquotes expands to Y Y, but since backquotes don't have a grouping function, the two Y's are treated as separate arguments

```
$ args $x$x$x
#args  = 1
arg[1] = "222"
```

$x expands into 2, which is concatenated with itself three times

```
$ args $x$y
#args  = 2
arg[1] = "2Y"
arg[2] = "Y"
```

$x expands to 2 and $y expands to Y Y; these two strings are concatenated to give 2Y Y and when the shell splits the line into words, the second Y becomes an arg in its own right

```
$ args $xy
#args  = 0
```

there is no variable called xy, so $xy expands to the empty string, which vanishes when the shell splits the command line into words