Introduction to `awk` programming

(block course)

Solutions to the exercises

Michael F. Herbst
michael.herbst@iwr.uni-heidelberg.de
http://blog.mfhs.eu

Interdisziplinäres Zentrum für wissenschaftliches Rechnen
Ruprecht-Karls-Universität Heidelberg

15th – 17th August 2016

Contents

Solutions to the exercises

Solution to 1.3

- The commands are

```
$ awk '/hunger/' resources/gutenberg/pg76.txt
```

and

```
$ awk '/hunger/' resources/gutenberg/pg74.txt
```

- One solution for the program `first.awk` is

```
# print lines matching hunger
/hunger/ {print}

# print lines matching year
/year/   {print}
```

1_first_look/sol/first.awk

which works since if either rule matches the line is printed. Unfortunately it will print the line twice, if both the words “hunger” as well as “year” are contained in the input data.

Solution to 1.4

- The file `resources/testfile` contains

```
some words
any data
some further words
somer morer things
more other thing
even more data
```

resources/testfile

whereas the output of

```
$ awk -f 1_first_look/printprint.awk resources/testfile
```

is

```
some␣words
some␣words
any␣data
any␣data
some␣further␣words
some␣further␣words
somer␣morer␣things
somer␣morer␣things
```

```
more␣other␣thing
more␣other␣thing
even␣more␣data
even␣more␣data
```

in other words, each line is shown twice. This is due to the fact, that the `awk` program `1_first_look/printprint.awk` contains the rule `{print}` twice, which is unconditionally executed. Therefore this printing instruction (which prints the whole record) is executed twice, i.e. the output contains each line twice.

- Now `awk` reads no records (empty file) and hence none of the two rules is executed. Therefore the program produces no output.

Solution to 2.2

The matching part:

- `..` matches any string that contains any two character substring, i.e. any string with two or more letters. This is everything except `g` and the empty string.
- `^..$` matches a string with exactly two characters, i.e. `ab` and `67`.
- `[a-e]` matches any string that contains at least one of the characters `a` to `e`, i.e. `ab` and `7b7`.
- `^.7*$` matches any string which starts with an arbitrary character and then has zero or more 7s following. This is `g`, `67`, `67777`, `7777` and `77777`.
- `^(.7)*$` matches any string which has zero or more consecutive substrings consisting of an arbitrary character and a 7. This is `67`, `o7x7g7`, `7777` and the empty string. Note that e.g. `77777` does not match: If we “use” the pattern `.7` three times we get `^.7.7.7$` and `77777` has one character too little to be a match for this.

Solution to 2.3

The crossword:

	`a?[3[:space:]]+b?`	`b[^eaf0-2]`
`[a-f][0-3]`	a3	b3
`[[:xdigit:]]b+`	3b	bb

Solution to 2.4

a) `ab*c` or `c$` or just `c`

b) `ab+c` or `bc$`

c) `^a.*c` or `c$`

d) `^␣*q` or `q..`

e) `^a|w` or `....`

Solution to 2.5

- Regexes for the parts:
  - sign: "`[+-]`"
  - prefactor: "`[01]\.[0-9]*`"
  - exponent: "`[0-9]+`"
- So altogether the scientific numbers need to match:

```
([+-]?)([01]\.[0-9]*)e([+-]?)([0-9]+)
```

where the parenthesis `( )` are only provided to show the individual parts, i.e.

```
[+-]?[01]\.[0-9]*e[+-]?[0-9]+
```

would be valid as well. Executing this on the `digitfile` gives

```
$ awk '/[+-]?[01]\.[0-9]*e[+-]?[0-9]+/ {print}' resources/digitfile
```

```
1.759e+15
1.5e+5da␣is␣a␣scientific␣number
-1.34e+04
```

- Introducing the fault tolerance implies:
  - We replace the plain requirement for "`e`" by the bracket expansion "`[eEdD]`".
  - Instead of "`[01]\.[0-9]*`", we require a number with an optional decimal part, i.e. "`[0-9]+(\.[0-9]*)?`"

Hence overall

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

```
$ awk '/[+-]?[0-9]+(\.[0-9]*)?[eEdD][+-]?[0-9]+/ {print}' resources/digitfile
```

```
1.759e+15
-9.3e-5
19e-5␣is␣not␣properly␣formatted␣either.
1.5e+5da␣is␣a␣scientific␣number
-1.34e+04
```

Solution to 3.1

This can be achieved using the commandline

```
$ awk '/^free/ { print $2 }' resources/gutenberg/pg1232.txt
```

which prints

```
Republic
him
he
sharing
distributed
```

Solution to 3.3

The first and second column from the matrix file can be extracted using

```
$ awk '{ print $1 "␣" $2 }' resources/matrices/lund_b.mtx
```

which gives

```
%%MatrixMarket␣matrix
%␣This
147␣147
1␣1
2␣1
8␣1

...

146␣146
147␣146
147␣147
```

If we want to exclude the first two lines (comment lines), we need to run

```
$ awk '/^[^%] { print $1 "␣" $2 }' resources/matrices/lund_b.mtx
```

instead.

Solution to 3.5

One solution to the exercise is

```
{
  res = res "␣" $0
  print res
}
```

3_basics/sol/growingconcat.awk

If additionally one wants to get rid of the leading space in each line, one could use the program

```
{
  res = res blank $0
  blank = "␣"  # set blank to be a space from here on
  print res
```

```
}
```

3_basics/sol/growingconcat_nospace.awk

The idea behind this latter script is, that for the first record `blank` and `res` are not defined, i.e. equivalent to the empty string.

Solution to 3.6

First we explain the program:

- The first line of `3_basics/exscript.awk` just causes the current value of the variable `a` to be printed. If this variable is undefined or empty it will print an empty line.
- The second line always sets `num` to the string `"false"` and increases the value of `a`.
- Third line decreases `a` and sets `num` to `"true"` if the record, which is processed contains a digit $0 \dots 9$
- In other words if the record contains a digit the value of `a` will overall remain unchanged and `num` is `"true"` before executing line 4.
- Line 4 will just print the value of num, so if this line prints `num: false` then the value of `a` is increased.

Now we look at the input.

- The first record is `4`. Here no value resides in `a`, i.e. we print an empty string. Furthermore `num` is set to `"true"` and `a` is updated to 0. The output of this record is

```

num: true
```

- Next record is a number as well. We print the 0 from the previous record and the same `num: true`. No change to `a`. The output is

```
0
num: true
```

- Next record contains no number, so `a` is increased to 1 and num is now `"false"`, which yields

```
0
num: false
```

- Finally we print the increased `a` and increase it further, since num is still `"false"`:

```
1
num: false
```

- and so on

Solution to 3.7

In order to count the number of lines which contain any digit, we can use the script

```
/[0-9]/ { c+=1 }
END { print c }
```

3_basics/sol/count_numbers.awk

This will provide us with those lines containing *any* kind of number as well, since numbers are obviously made up of digits.

The program

```
/[+-]?[01]\.[0-9]*e[+-]?[0-9]+/ { c+=1 }
END { print c }
```

3_basics/sol/count_scinumbers.awk

on the other hand counts the number of lines with scientific numbers (in the strict sense).

Solution to 3.8

We compute the column-wise averages using the program

```
{
  count++       # Count of the matrix elements
  sum1 += $1    # Sum of first column
  sum2 += $2    # Sum of second column
  sum3 += $3    # Sum of third column
}

END {
  # Compute averages and print:
  print "Average␣col1:␣" sum1/count
  print "Average␣col2:␣" sum2/count
  print "Average␣col3:␣" sum3/count
}
```

3_basics/sol/mtx_averages.awk

This results in

```
$ awk -f 3_basics/sol/mtx_averages.awk resources/matrices/3.mtx
```

```
Average␣col1:␣1.90909
Average␣col2:␣1.90909
Average␣col3:␣2.45455
```

and

```
$ awk -f 3_basics/sol/mtx_averages.awk resources/matrices/lund_b.mtx
```

```
Average␣col1:␣79.1026
Average␣col2:␣68.2924
Average␣col3:␣150.228
```

and

```
$ awk -f 3_basics/sol/mtx_averages.awk ↙
    ↪resources/matrices/bcsstm01.mtx
```

```
Average␣col1:␣24.48
Average␣col2:␣24.48
Average␣col3:␣64.96
```

If one wants to make sure to skip the first few comment lines, one can use the program

```
/^[^%]/ {
  count++       # Count of the matrix elements
  sum1 += $1    # Sum of first column
  sum2 += $2    # Sum of second column
  sum3 += $3    # Sum of third column
}

END {
  # Compute averages and print:
  print "Average␣col1:␣" sum1/count
  print "Average␣col2:␣" sum2/count
  print "Average␣col3:␣" sum3/count
}
```

3_basics/sol/mtx_averages_skip.awk

instead, where a guarding regular expression pattern makes sure that only non-comment lines are included in the average.

Solution to 3.10

- The program

```
# check if we have a comment. If not increase the line number
# and flag as a nocomment record
/^[^%]/ {
  linenr +=1
  nocomment=1
}

# Extract the number of entries and store them
linenr == 1 && nocomment {
  nentries=$3
}

# Increase the count of actual entries,
# since this an explicitly provided entry
```

```
linenr > 1 && nocomment {
  actualentries++
}
END {
  print "Expected entries:     " nentries
  print "Actual entries:       " actualentries
}
```

3_basics/sol/mtx_check_entry_count.awk

prints both the counted and the expected number of non-zero entries in the `mtx` file.

- Both values for the sparsity ratio are printed by

```
# check if we have a comment. If not increase the line number
# and flag as a nocomment record
/^[^%]/ {
  linenr +=1
  nocomment=1
}

# Extract the number of nonzeros and store them
# Compute the number of rows times colums
linenr == 1 && nocomment {
  rows = $1
  cols = $2
  nentries=$3
  total = rows*cols
}

# Increase the count of actual entries,
# since this an explicitly provided entry
linenr > 1 && nocomment {
  actualentries++
}


END {
  print "Sparsity ratio:        " (total-nentries)/total
  print "Actual sparsity ratio: " (total-actualentries)/total
}
```

3_basics/sol/mtx_sparsity_ratio.awk

- The elementwise square is computed by the program

```
# Copy all comments:
/^%/ { print $0 }

# If no comment and we have passed the first line:
/^[^%]/ && passedfirst == 1 {
  print $1 " " $2 " " ($3*$3)
}

# Copy the first non-comment line verbatim:
```

```
/^[^%]/ && passedfirst != 1 {
  passedfirst=1
  print
}
```

3_basics/sol/mtx_square_elements.awk

Solution to 3.11

One possible way to extract the excited states is:

```
#!/usr/bin/awk -f

# We use the state variable inside_block to keep track whether
# we are inside or outside an excited states block
# It's default value is 0, i.e. outside

# whenever we encounter the "  Excited state ", we
# change the flag to indicate that we are inside the table.
# also we store the state number, which sits in the third field
/^ *Excited state / { inside_block=1; state_number=$3 }

# if we find the "Term symbol" line inside the block, we store
# the term symbol which sits in $3 $4 and $5
inside_block==1 && /^ *Term symbol/ { term_symbol=$3 " " $4 " " $5 }

# if we find the "Excitation energy" line, we store the excitation energy
# and print the table, since we do not care about the rest of the
# block. Next we reset the inside_block flag for the next block to come.
inside_block==1 && /^ *Excitation energy/ {
  excitation_energy=$3

  # print the data tab-separated (for analysis with e.g. cut)
  print state_number "\t" term_symbol "\t" excitation_energy

  inside_block=0
}
```

3_basics/ex_extract_states.awk

Solution to 4.1

The following program implements one way to print duplicated words in a text:

```
#!/usr/bin/awk -f
# change the record separator to anything which is not
# an alphanumeric (we consider a different word to start
# at each alphanumeric character)
BEGIN { RS="[^[:alnum:]]+" }
```

```
# now each word is a separate record

$0 == prev { print prev }
{ prev = $0 }
```

4_parsing_input/sol/duplicated_words.awk

Solution to 4.2

- The final balance is printed by

```
#!/usr/bin/awk -f

# Change field separator:
BEGIN { FS ="," }

# Extract starting balance
/^[#] Starting balance/ { balance = $2 }

# Once in the transfer block, adjust balance:
/^[^#]/ { balance += $2 }

# Print the final balance:
END { print "Final balance: " balance }
```

4_parsing_input/sol/csv_balance.awk

- A balance column is appended by

```
#!/usr/bin/awk -f

# Change field separator:
BEGIN { FS ="," }

# Extract starting balance
/^[#] Starting balance/ { balance = $2 }

# Print all comment lines verbatim:
/^[#]/

# Once in the transfer block, adjust balance and append a column
/^[^#]/ {
  balance += $2
  print $0 "," balance
}
```

4_parsing_input/sol/csv_balance_append_column.awk

Solution to 4.5

One solution program to add all scientific numbers which occur in some input is:

```
#!/usr/bin/awk -f

BEGIN {
  # Pattern for the Sign:
  sign="[+-]?"

  # Pattern for an integer
  intp="[0-9]+"

  # For a float we may additionally have
  fpe="(\\.[0-9]+)?"

  # The optional exponent
  expe="(e[+-][0-9]+)?"

  # Build the pattern:
  FPAT=sign intp fpe expe
}

# Assume that we have no more than 5 numbers in each line
# (which is true for the digitsfile)
{ c += ($1 + $2 + $3 + $4 + $5) }
END { print "The sum is " c }
```

4_parsing_input/sol/add_digits.awk

Solution to 5.3

- One way to achieve the unfolding is to do default input parsing, but to print each field on a different line, e.g.

  ```
  $ awk 'BEGIN { OFS="\n" }; { $1=$1; print }' resources/testfile
  ```

  The other option is to treat each word as a separate record, i.e.

  ```
  $ awk 'BEGIN { RS="[[:space:]]+" }; { print }' resources/testfile
  ```

- Changing the separator character in a `csv` file from comma to semicolon can be achieved by the simple commandline

  ```
  $ awk 'BEGIN {OFS=";"; FS=","}; {$1=$1; print }' resources/data/money.csv
  ```

  which sets `FS` and `OFS` appropriately and then triggers a rebuild of the `$0$` variable.

Solution to 5.5

One solution to print the average measurement value and to exclude the erroneous apparatus 3 explicitly is

```
#!/usr/bin/awk -f

BEGIN {
  # make both space and : field separators
  FS="[ :]+"

  # Alternatively we can use FPAT to describe
  # the numbers that we expect:
  #
  # FPAT="[0-9]+|-[0-9]\\.[0-9]+"
  #
  # Problem is that this does not work for numbers
  # in the scientific format like that.
  # One would need to add another alternation.

  print("## | average")
  print("---+--------")
}

# only process if the line is no comment line
$0 !~ /^#/ {
  # $1 is the apparatus count
  # $2 to $8 is the values

  # compute average and add to total sum
  sum = ($2+$3+$4+$5+$6+$7+$8)
  avg = sum/7.

  # print avg
  printf("%2d | %.4f\n",$1,avg)
}

# Apparatus 3 is a little off, so exclude it explicitly
$1 != 3 {
  totsum+= sum
  totcount+=7
}

# Print a note about this:
$1 == 3 { print "   | Note: Not included in total sum" }

# print results:
END { printf("\ntotal avg: %.4f\n",totsum/totcount) }
```

5_printing_output/sol/analysis.awk

Solution to 6.3

If one wants to use a range pattern, this can be done using the program

```
#!/usr/bin/awk -f

# The chapter to extract, here the first
BEGIN { v=1 }

# The range: From this until the next.
$1 == "CHAPTER" && $2 == v, $1 == "CHAPTER" && $2 == (v+1)
```

6_patterns_actions_variables/sol/extract_chapter.awk

Running this like

```
$ 6_patterns_actions_variables/sol/extract_chapter.awk resources/gutenberg/pg161.txt
```

gives

```
CHAPTER 1


The family of Dashwood had long been settled in Sussex.  Their estate
was large, and their residence was at Norland Park, in the centre of

...

having much of her sense, she did not, at thirteen, bid fair to equal
her sisters at a more advanced period of life.



CHAPTER 2
```

In order to avoid the chapter heading of the next chapter to be printed, one could store the chapter number instead:

```
#!/usr/bin/awk -f
BEGIN { v=1 }

# remember chapter number
/^CHAPTER [0-9]+/ { chapter = $2 }

# Print the chapter if it is the right one
chapter == v
```

6_patterns_actions_variables/sol/extract_chapter_state.awk

Solution to 6.4

One solution is to count the number of lines inside the Davidson range:

```
#!/usr/bin/awk -f

# We know that the iteration count increases by one
# for each extra line we find in the Davidson block
# There are 7 lines containing no iterations
# (ie the headings, the guess and the summary)
# So we count all lines between "Starting Davidson"
# and "Davidson Summary" and subtract 7 to get the
# number of iterations.
/^  Starting Davidson \.\.\./, /^  Davidson Summary:/ { count+=1 }
/^  Davidson Summary:/ {
  # print count and reset
  print count-7
  count=0
}
```

6_patterns_actions_variables/sol/extract_davidson.awk

Solution to 6.5

If one wants to automatically exclude the instrument based one an upper threshold, one could use the program

```
#!/usr/bin/awk -f

BEGIN {
  # The upper threshold to include a value
  thresh_upper = -0.05

  # make both space and : field separators
  FS="[ :]+"

  print("## | average")
  print("---+---------")
}

# Skip comment lines:
/^#/ { next }

# $1 is the apparatus count
# $2 to $8 is the values

{
  # Compute the average:
  sum = ($2+$3+$4+$5+$6+$7+$8)
  avg = sum/7.

  # if the average is larger than upper threshold,
  # the apparatus is off and we skip the rest
```

```
  if (avg >= -0.05) {
    printf("%2d | %.4f > %.4f => Not included\n",$1,avg,thresh_upper)
    next
  }

  # All the records that made it here
  # should be included:
  printf("%2d | %.4f\n",$1,avg)

  totsum+= sum
  totcount+=7
}

END { printf("\ntotal avg: %.4f\n",totsum/totcount) }
```

6_patterns_actions_variables/sol/analysis_automatic.awk

Solution to 6.7

- A possible factorial program is

```
#!/usr/bin/awk -f
{
  n=+$1
  res=1
  while(n>1) {
    res=res*n
    --n
  }
  print res
}
```

6_patterns_actions_variables/sol/factorial.awk

- A couple of examples:

```
$ echo -e "20\n50\n100" | 6_patterns_actions_variables/sol/factorial.awk
```

gives

```
2432902008176640000
30414093201713375576366966406...832057064836514787179557289984
93326215443944102188325606108...311236641477561877016501813248
```

So `awk` is able to do integer arithmetic up to the point that it allows to calculate 100! using an extremely naive algorithm!

Solution to 6.9

Just replace the `while` by a `for` loop:

```
#!/usr/bin/awk -f
{
  res=1
  for(n=+$1; n>0; --n) {
    res=res*n
  }
  print res
}
```

6_patterns_actions_variables/sol/factorial_for.awk

Solution to 6.10

We generalise the program by using a `for`-loop over fields and make the code cleaner using an `if`-statement.

```
#!/usr/bin/awk -f

BEGIN {
  # The upper threshold to include a value
  thresh_upper = -0.05

  # make both space and : field separators
  FS="[ :]+"

  print("## | average")
  print("---+--------")
}

# Skip comment lines:
/^#/ { next }

# $1 is the apparatus count
# from $2 onwards are the values

# Compute the average:
{
  # Accumulate the sum:
  sum=0
  for (i=2;i<=NF;++i) {
    sum+=$i
  }
  avg = sum/(NF-1)

  # check if the average is larger than
  # upper threshold, if yes then apparatus
  # is off and we skip the rest, else we
  # print and add to the total
  if (avg > thresh_upper) {
```

```
    printf("%2d | %.4f > %.4f => Not included\n",$1,avg,thresh_upper)
    next
  } else {
    printf("%2d | %.4f\n",$1,avg)
    totsum+= sum
    totcount+=(NF-1)
  }
}

END { printf("\ntotal avg: %.4f\n",totsum/totcount) }
```

6_patterns_actions_variables/sol/analysis_general.awk

Solution to 6.11

The following script checks whether the first field of each record is a prime number.

```
#!/usr/bin/awk -f
{

  isprime=1
  n=+$1
  for (i=2; i*i < n; ++i) {
    if (n % i == 0) {
      isprime=0
      break
    }
  }

  if (isprime) {
    printf("%d is prime\n", n)
  } else {
    printf("Smallest divisor of %d is %d\n", n,i)
  }
}
```

6_patterns_actions_variables/sol/is_prime.awk

The output for

```
$ echo -e "101\n1001" | 6_patterns_actions_variables/ex_break.awk
```

is

```
101 is prime
Smallest divisor of 1001 is 7
```

Solution to 6.12

One solution to find the number of times "a" and "e" occur in a book is

```
#!/usr/bin/awk -f

BEGIN { FS="" }

/\*\*\* START OF THIS PROJECT GUTENBERG EBOOK [A-Z,-.]+ \*\*\*/ {
  # Flag that we are inside, but do not do statics on this record
  inside_book=1
  next
}

/\*\*\* END OF THIS PROJECT GUTENBERG EBOOK [A-Z,-.]+ \*\*\*/ {
  # We are at the end of the book => quit awk program
  exit
}

# if we are inside:
inside_book {
  # Ignore case such that both upper
  # and lower case characters are counted
  IGNORECASE=1

  for (i=1; i<=NF; ++i) {
    # Use a regex here, since == operator
    # is not affected by IGNORECASE
    if ($i ~ /a/) {
      acount++
    } else if ($i ~ /e/) {
      ecount++
    }
    charcount++
  }

  # Unset IGNORECASE, since the regex patters above are
  # case sensitive.
  IGNORECASE=0
}

# Print final results:
END {
  printf("total %8d\n",charcount)
  printf("a     %8d  (%6.2f%%)\n",acount,acount/charcount*100)
  printf("e     %8d  (%6.2f%%)\n",ecount,ecount/charcount*100)
}
```

6_patterns_actions_variables/sol/gutenberg_character_statistics.awk

Solution to 7.3

One solution, which also excludes all whitespace characters when performing the character counting, is

```
#!/usr/bin/awk -f

BEGIN { FS="" }

/\*\*\* START OF THIS PROJECT GUTENBERG EBOOK [ A-Z,-.]+ \*\*\*/ {
  # Flag that we are inside, but do not do statics on this record
  inside_book=1
  next
}

/\*\*\* END OF THIS PROJECT GUTENBERG EBOOK [ A-Z,-.]+ \*\*\*/ {
  # We are at the end of the book => quit awk program
  exit
}

inside_book {
  for (i=1; i<=NF; ++i) {
    # Ignore those characters which are space characters:
    # Note: Not strictly speaking required for the exercise,
    # but gives a nicer result in the end.
    if ($i ~ /[[:space:]]/) continue

    # Increase count for character and total count
    #
    # one could also use
    #       count[tolower($i)]
    # in order to map each character to its lower-case
    # equivalent and make a count over this instead.
    count[$i]++
    charcount++
  }
}

# Print final results:
END {
  printf("total %8d  (%6.2f%%)\n",charcount,charcount/charcount*100)
  print("-----------------------")

  for (c in count) {
    printf("%-5s %8d  (%6.2f%%)\n",c,count[c],count[c]/charcount*100)
  }
}
```

7_arrays/sol/gutenberg_character_statistics.awk

Solution to 8.3

If we allow ourselves to use the usual control structures one could find the maximum and absolute maximum like this

```
#!/usr/bin/awk -f

# The usual abs function
function abs(a) {
  if (a<0) return -a
  return +a
}

# Initialise max and absmax:
NR == 1 {
  max = $1
  absmax = abs($1)
}

# Loop over each field (number) and update
# max and absmax if necessary
{
  for(i=1;i<=NF;++i) {
    if ($i > max) {
      max = $i
    }
    if (abs($i) > absmax) {
      absmax=abs($i)
    }
  }
}

END {
  print "max:    " max
  print "absmax: " absmax
}
```

8_functions/sol/max_element_long.awk

Alternatively, we can change the range separator and use `awk`'s implicit loop over records to achieve the same thing in less lines of code and without a single control structure:

```
#!/usr/bin/awk -f

# The usual abs function
function abs(a) {
  if (a<0) return -a
  return +a
}

# Change record separator to repeated space chars
# so each field of the matrix becomes a record on its own.
BEGIN { RS="[[:space:]]+" }

# Initialise max and absmax with first record:
```

```
NR == 1 {
  max = +$0
  absmax = abs($0)
  next
}

# For all other record, determine if max or absmax:
+$0 > max { max = +$0 }
abs($0) > absmax { absmax = abs($0) }

END {
  print "max:    " max
  print "absmax: " absmax
}
```

8_functions/sol/max_element.awk

Solution to 9.4

- `wc -w` is equivalent to

```
#!/usr/bin/awk -f
# Split into a new record at multiple occurrences of space
# characters. Then just print the record count.
BEGIN { RS="[[:space:]]+" }
END { print NR }
```

9_practical_programs/sol/wc_w.awk

- `uniq -c` we can implement like

```
#!/usr/bin/awk -f

# Initialise buffer to be the first record:
NR == 1 { buffer=$0 }

# If repeated occurrence increase count:
buffer == $0 { count++ }

# Else print the record we had in the buffer
# and reset counter and buffer
buffer != $0 {
  printf("%5d %s\n",count,buffer)
  buffer=$0
  count=1
}

# Print what is left in the buffer
END {
  printf("%5d %s\n",count,buffer)
}
```

9_practical_programs/sol/uniq_c.awk

- `sort` is implemented using `awk`'s `asort`:

```
#!/usr/bin/awk -f

# Append all input lines to a buffer array
{ buffer[NR] = $0 }

# In the end sort using asort and print in order
END {
  nr = asort(buffer)
  for (i=1; i<=nr; ++i) {
    print(buffer[i])
  }
}
```

9_practical_programs/sol/sort.awk

- `egrep` can be mimicked using a surrounding shell script with inline `awk` code:

```
#!/bin/sh
# Store the regex (first argument to script)
regex=$1
shift

# Call awk and use DOUBLE quotes to insert the regex
# inside an awk pattern and pass the remaining
# arguments to the scripts to awk itself (as files)
#
# Whenever that regex pattern matches the default print
# action is executed (exactly like egrep does it)
awk "/$regex/" $@
```

9_practical_programs/sol/egrep.sh

For more details, how the shell command `shift` works and what the shell variables `$1` and `$@` mean, see chapter 3.2.1 and 4.6 of the lecture notes to the "advanced bash scripting" course[1].

[1] Available from `http://blog.mfhs.eu/teaching/advanced-bash-scripting-2015/`.

Licensing and redistribution