Tekstmanipulatie, week 13

1. sed

From the online manual:

Sed is a ***s****tream* ***ed****itor*. A stream editor is used to perform basic text transformations on an input stream (whatever shell sends to its standard input, let it be the content of a file or the output of a pipeline). The command sed makes always only one pass over the input, and this makes it more simple and more efficient. To pass over the input more times, you need more sed commands.

In other words:

The 'sed' command is like a sewing machine that goes through a file (a text), and performs some basic operations. You can define the operation to be performed in two ways:

- Either in the command line, as arguments of 'sed'
- Or in an independent file (a 'script-file'), and you run sed with the following argument: -f <filename>. (By using script files, you can perform more complicated tasks with sed. However, we are not going to use sed scripts in this course. Similarly, we are not going to learn about many advanced functionalities of sed.)

When performing the 'sed' command, the computer will take all commands appearing either in the command line or in the specified script files, and decide in which order to execute them. Therefore, if you want the operations to be executed in a specific order, and the result is not what you wanted, then you do better use a pipe line, and make sure that the different commands are executed in the order you want.

What are the basic operations you can use?

```
sed s/regex/newstring/ file1 > file2
```

In each line, this command will rewrite the first string that matches the given regular expression (for possibilities that can be used in regex, see the grep command) by newstring. The input file is file1, and the output file is file2.

```
sed expr1/s/regex/newstring/
```

This will rewrite the first string that matches the regular expression given by `newstring`, but only in the lines that include something matching `expr1`. The command

```
sed expr1/s/regex/newstring/g
```

will replace all instances regex, and not only the first one in each line ('g' stands for global). Further, the command

```
sed expr1/s/regex/newstring/15
```

will replace the 15th instance. (Any number can be given.) E.g.:

```
echo aaaaaaaaaaaaaaaaaaaaaaaa | sed -e s/a/b/15
aaaaaaaaaaaaaabaaaaaaaaaa
```

Then:

```
sed /regex/d
```

This rule will delete all lines that include regex, yielding a similar result to the -v option of grep.

If you want to delete a given string or a regular expression, you can do it by replacing it with "nothing" (to be more exact: by replacing it with the *empty string*):

```
sed s/regex//.
```

If you want to put more than one operation into the command line (or you have a script file with at least one operation in the command line), use the -e option before each operations of the command line:

E.g. `sed -e /Henry/d -e /Sally/s/Smith/White/ people.old > people`
This will delete all lines containing 'Henry" from the file called people.old, and change 'Smith' to 'White' in all lines containing 'Sally'. The result goes to the file called 'people'.

The & character has a special meaning: it stands for the string that has been matched. E.g.:

```
echo "ik ben jan" | sed -e 's/j[a-z]*n/piet&piet/'
ik ben pietjanpiet
```

You will frequently use & when you will want to put each character of the input file into a new line. Then, you have to insert a new-line character after each character. With the following command line, you can insert the cahracter @ after each character of the text:

```
sed "s/./&@/g"
```

This works because . stands for whatever character, similarly to what we have learnt concerning grep. Then, the safest way to insert a new-line character after each character is (supposing that the character @ does not occur in the original input):

```
sed "s/./&@/g" | tr @ '\012'
```

A script consists of commands, each of them in a new line (or separeted with a semi-colon). An example:

```
cat > changes
   /Sally/s/Smith/White/g
   /Henry/d
   $a\
   James Walker 112
   ^D
sed -f changes people.old > people.new
```

This script deletes all lines containing 'Henry', changes 'Smith' to 'White' in lines containg 'Sally', and add the line 'James Walker 112' to the end of the file. The a\ command stands for "add" or "append". The $ symbol stands for "last command line". (The ^D means to press CTRL-D to end the file.)

If you want to know more about sed (cycles, buffers; suppressing the output with the -n option, so that only the line including the p command should be outputed, etc.), consult any UNIX manual.

An important note:

The s/regex/string/ functionality of sed replaces always the longest string that

matches the regular expression. For instance, what is the output of the following command line?

```
echo baaaaaab | sed "s/a*/@/"
```

The command sed replaces the *first* string in the line that matches the regular expression `a*`. This is zero or more consecutive a's, and already the beginning of the line matches it (namely, zero a's). Therefore, the output will be:

```
@baaaaaab
```

However, if we require at least one character a in the regular expression (`s/aa*/@/`), then the string to be replaced by @, can be either a, or aa, or aaa, etc., yielding one of the following outputs:

```
@b@aaaaab
b@aaaab
b@aaab
b@aab
b@ab
b@b
```

Which one of these will be the output of the above command line? Here becomes the rule about the longest match possible important: for sed looks for the longest string matching the regular expression, the last one will be the output produced.

sed and tr

What is the difference between sed and tr? Which one to use? The command tr transforms or deletes *characters*, while the `s/regex/string/g` functionality of sed replaces each string that matches the regular expression to the second string. With tr, unlike with sed, you cannot change one character to a string of more characters, or a string of more characters to something else. On the other hand, because a single character is also a regular expression, you can (almost) always use sed instead of tr. It seems as if sed was just more general than tr. However, there are a few cases when you may prefer using tr, or you cannot simply avoid using it.

First, imagine that you want to replace each occurences of the character a into 1, and simultaneously each occurences of the character b into 2, and also each c's into 3. The following command lines will yield the same result, you may choose the simplest

yourself:

```
tr abc 123
sed s/a/1/g | sed s/b/2/g | sed s/c/3/g
sed -e s/a/1/g -e s/b/2/g -e s/c/3/g
```

(The last option works, because it does not matter in which order the different rules are applied. Nevertheless, if the order of applying the rules is important, use rather a pipe line.)

Second, if you want to change all occurences of a, b or c into 1, you can do one of the followings (I guess you would go again for tr):

```
tr abc 111
tr abc 1
sed s/a/1/g | sed s/b/1/g | sed s/c/1/g
sed -e s/a/1/g -e s/b/1/g -e s/c/1/g
sed s/[abc]/1/g
```

Here, the second command line is a short hand for the first one: whenever the second parameter of tr is a shorter string than the first parameter, then the last character of the second parameter is also used to replace the remaining characters of the first string. Furthermore, the regular expression [abc] in last command line refers to any one-character-long string composed of the elements of the set {a, b, c}.

Third, you can easily refer to the ASCII (or ISO 8859) code of some special characters, something that is much more complicated in sed. The last important difference is that you can delete new-line characters in tr (`tr -d '\n'` or `tr -d '\012'`). However, this is not possible in sed, since sed reads its input line-by-line, and writes its output line-by-line. Consequently, with sed you can make changes within one line; and you can delete entire lines by using the rule `/regex/d`, that is, you do not allow them get printed on the output. However, deleting the new-line characters in the file and making the entire file one long line contradicts the whole idea behind sed.

2. `cut`, `paste`

A big number of the Unix commands are so-called "filters". These are small programs that read the standard input (or the redirected standard input; and in some cases the

name of the input file can be given as an argument, too), do something with it, and writes the result to the standard output (or to the redirected standard output).

Combining filters, i.e. building a series of filters is possible with pipe-lines ('|').

The simplest filter is `cat`: it just copies the input (that can be the con**cat**anation of several files given as arguments, too) to the output.

Further filters were (cf. 3rd week): rev, sort, tr, uniq, wc, head, tail, as well as (cf. 4th week) grep. The previously introduced 'sed' command can also be seen as a filter.

Further filters are: colrm (removing columns from a file), crypt (coding and decoding data; theoretically this command shouldn't be available in systems outside the US -- for federal security reasons...), look (displaying lines beginning with a given string), spell (spell checker, not available on all systems). If you are interested, check the online manual to get more information about them.

Now we are going to deal with two further commands: `cut` and `paste`.

In Unix, "everything is a file", and "every file is a text". This means that each file is considered to be a series of lines, each endig with an end-of-line character, and a line being a series of characters (printable or not printable ones). (If our file does not contain any end-of-line character, because it is not meant to be a text, then it is seen to be an one-line-long text.)

Columns

Under Unix, a file can be seen not only as a text document composed of lines, but also as a table. Lines correspond to rows, and each character in a line is a different cell of the table. Each cell is filled with a value between 0 and 255, or, if you want, with one character. The only difference between a file and a regular table is that a file does not necessarily have the same number of characters in a line, therefore, it is like a table which may have different number of cells in each row. Now, the *n*th *column* in a table is composed of the *n*th cells in each row. Similarly, the *n*th column of a file is composed of the *n*th character of each line.

(We have already encountered this idea at the +*n* option of the `sort` command, where *n* is a number.)

The second way to define *columns* is to define what the *delimiter* between two columns is: for instance, by default, a tabulator-character (TAB, \t) may serve as the delimiter. Now, one cell (called as a *field* in this context) is what occurs between two

delimiter characters, and contains a string of characters. The *n*th column, then, is defined as the sequence of the *n*th cells (fields) of each row. (For the sake of ease, let us consider a new-line character and an end-of-file character always as a delimiter character.)

Let us see an example to the first definition of a column, where a cell is one character:

Suppose we have a file called `grades`, containing student number, the name of the student and the final grade:

```
0513678 John   8
0612942 Kathy  7
0418365 Pieter 6
0539482 Judith 9
```

Suppose you want to hang out this list, but without the names (just student no. and grade). If each character in a line is seen as a separate cell of a table, then you want to keep the first and the last columns of this "table". Therefore, you want to remove the columns 9 to 15. (The first character of a line is in column no. 1.) There are several options to do that (the `lpr` command sends its input to the printer):

```
colrm 9 15 < grades | lpr
cut -c1-8,16-18 grades | lpr
```

You also could redirect the output to another file (`> grade_without_names`), and then print it (`lpr grades_without_names`), of course.

The moral is:

> `colrm [ startcol [ endcol]]` : will remove the columns from no. `startcol` (until no. `endcol`, if specified): these are two seperate arguments of the command. The input file should be given as a redirected standard input. The output lacks the specified columns.
>
> `cut -cLIST file` : will output only the specified columns. The input file is given as an argument, and the list of columns to be cut are given as an argument after option -c. The LIST of columns to be cut out should meet the following criteria: numbers are separated by commas (but no space! because a space would mean: 'end of the argument'), and a series of neighbouring columns can be given using a minus sign. E.g. '`3,5-8,11-20,28`' would mean the column no. 3, as well as columns 5 to (t/m) 8, columns 11 to 20, and column 28.

Comming back to our example using the grades of the students, how would the story look like if we used the second definition of what a column is (a cell = a field, that is, a string of characters between to delimiter characters)? Let us use the % character as the delimiter between the fields (cells), and let us introduce this character into the file in the following way:

```
0513678 % John   % 8
0612942 % Kathy  % 7
0418365 % Pieter % 6
0539482 % Judith % 9
```

Now, the first column contains the student numbers, the second column the name, while the third one the grade. The -d option of the command cut defines what the delimiter is, and you can refer to a field by giving its number after the option -f. Thus, to cut the first column, that is, the student numbers, use `cut -d % -f 1` (or, `cut -d "%" -f 1`: escape characters are required whenever your delimiter would be otherwise understood as a metacharacter by shell). The command line `cut -d "%" -f 3 > file3` will save the third column of its input into another file.

paste

The `paste` command can be used for many purposes. The basic use of it is merging columns.

Typically, you give more file names as its arguments. These are its input files. What paste does is that it reads the first line of its first input file, followed by the first line of its second input file, etc. Then, it concatenates them (writes one after the other), and this merging of the lines will be written as the first line of its standard output. Then, the same happens to the second line of the input files, etc.

To be more precise, the standard output of paste is a table. The first cell of the first row is the first line of its first input file, etc. In general, the *i*th field (cell) in the *j*th row is the *j*th line of the *i*th input file. With the option -d, you can define what you want to be the delimiter character, or, more generally, the delimiter string (it can be more than one characters, or, even the empty string,"", too). In fact, the *j*th line of the output is the concatenation of the *j*th lines of the first input file, followed by the delimiter character or string, followed by the *j*th lines of the second input file, followed by the delimiter character or string, etc.

The delimiter character, by default, is TAB (\t), which is visualized as if you had a series of spaces (the TAB character brings to the 1st, 9th, 17th, 25th, 33rd, etc.

columns). But, remember, this is not the case: if you look at the bytes in the file, it is not the same if you have three spaces (ASCII code = 32) or one TAB character (ASCII code = 9). Note that if you look at the output of Unix commands like wc or uniq -c, you will see the same tabular structure: fields are divided by a TAB character.

If you imagine the `cat` command as linking your files "vertically", then the `paste` command does the same "horizontally" (thank to the philosophy of seeing your files as being composed of lines). Thus, the syntax of this command is:

```
paste [ -d char]  file [ file...]
```

Suppose you have a file containing 5 names (`names`), another file containing 5 birthdays (`birthdays`), and a third one containing 5 addresses (`addresses`). The following command will create a file containing the combined information:

```
paste names birthdays addresses > info
cat info
      Jane    23/11    9722EK Groningen....
      Jack    05/09    9718UW Groningen....
      ...
```

Now how can you change the order of columns of a given file? Combining `cut` and `paste`:

```
cut -c1-8 info > naam; cut -c9-14 info > jaarig; cut
-c15-40 info > woning; paste naam woning jaarig
> new_info
```

(Remark: the semi-column is used as delimiters between commands. You could put them into separate lines, too. When putting into separate lines, the Shell will deal with them separately: pre-processing and executing the first line, then pre-processing and executing the second line, etc. When putting into one line, Shell will pre-process them together, and then execute them one-by-one.)

3. N-grams

The topic of this course gives us an excellent opportunity to mention a few basic ideas of statistical linguistics. What are the motivation of statistical approaches to

linguistics?

First: why dealing only with qualitative, and not also with quantitative properties of human languages? For instance: the lyric style of a romantic poem could be explained in many cases with the statistically significant overuse of some 'soft' sounds (e.g. [l], [n]), as opposed to the 'sharp' sounds ([t], [k],...) in revolutionary poems. Furthermore, many modern linguists emphasize that linguistic statements should be checked with quantitative methods. For instance, the paradigm by Noam Chomsky (a sentence is either
grammatical or agrammatical) has been slowly changed by many to a 'corpus-based paradigm': some types are very frequent (therefore 'more correct'), others are pretty rare ('less or marginally correct'), or absolutely abscent ('incorrect'?) in a given corpus (set of texts).

A second motivation for these statistical games are real life applications. For instance, when writing a document in more languages, you don't want to change the language of your spell checker all the time; instead, you would like your spell checker to recognize the language of your sentence. Here is a demo for guessing the language a text has been written in.

A very important implementation of statistical linguistics is text classification (cf. 6th week). Imagine you are in a big news agency or in an intelligence agency and there are thousands of articles comming in day after day. You want to select them by language and by topic in order to be able to send them to the appropriate person. Instead of using human beings just to read all these materials and to categorize them, you can automatize this task if you are clever enough. This has been a huge topic of research in the last couple of decades.

The basic information you need in statistical linguistics is the frequency of some items. You should distinguish between *absolute frequency* (number of occurences) and *relative frequency* (percentage: absolute frequency / total number of items). Items can be for instance characters (e.g. the percentage of the 'w' character will be much higher in English than in French), words (e.g. the word 'computer' is much more frequent in articles related to informational sciences than in articles related to pre-school care education), or sentence types (e.g. embedded sentences are much more likely in written texts produced by university students than in an oral corpus produced by uneducated people).

A further basic idea is the distinction between *type* and *token*. You usually have several tokens of the same type: several occurences of the same character, word, expression, sentence-class, etc. For instance in my previous sentence the type

'several' is represented by two tokens. The type-token ration is the ratio of the number of types and the number of tokens in a given text. A small child (or somebody learning a new language) use few words, resulting in a low type-token ration (e.g. 10 different words in a text of 100 words, i.e. lot of repetition). While an educated speaker with a rich vocabulary might use a lot of different words (especially in written texts), resulting in an extremely high type-token ratio.

On the other hand, some words (articles, preposition, pronouns,...) will occur very frequently independently of the richness of one's vocabulary. The frequent use of the words (nouns, verbs,...) characteristic to the topic in question is also unevitable. The distribution of frequent and less frequent words follow a Zipf-function (cf. next week). It is remarkable that similar Zipf-functions can be met in many different fields (DNA, distribution of town size in a given country, etc. see also topics related to fractals, chaotic behavior, critical phenomena, etc.).

You will very often remark that you would rather use the frequency of the combination of two or more items, rather than the frequency of one item. French texts can be very easily recognized by the *relatively* high frenquency of the 'ou', 'oi', character pairs (*bigram*). The bigram 'sh' is characteristic to English, 'aa' to Dutch, the *trigram* 'sch' to German and Dutch, etc. You can also look at bigrams, trigrams, etc. (*n-grams*, in general) of words: e.g. some English speakers use much more often the bigram 'you know' than others.

About creating n-gram frequency counts, please do have a look at Henny Klein's lecture-notes from last year.

4. Making a concordance (KWIC)

(From H. Klein's web site.)

Concordantie=verzameling van voorkomens van woorden in hun context. Al van oudsher werden er b.v. (handmatig uiteraard) concordanties van de bijbel gemaakt: lijsten van woorden en namen met de plaatsen waar ze te vinden waren. Modern en electronisch: KWIC (key word in context). In dit formaat wordt voor een gevraagd key word een tabel geproduceerd met daarbij de linker- en rechtercontext tot een bepaalde diepte (bijvoorbeeld 35 characters). Voorbeeld voor het woord itself in Federalist/fed60.txt:

ts own elections to the Union itself . It is not
e into them, it would display itself in a form
preference in which itself would not be included? Or to
itself could desire. And thirdly, th

Hoe maken we zo'n KWIC tabel met behulp van UNIX utilities? Antwoord:

1. gebruik grep om relevante regels te selecteren
2. gebruik cut om context te extraheren
3. gebruik sed om contexten lang genoeg te maken (opvullen met spaties)
4. gebruik cut om contexten kort genoeg te maken (eventueel met rev).
5. gebruik paste om iedere kolom weer naast elkaar te geven

Hier gaan we dan. Stel je voor dat we net zoals boven een KWIC voor het key word itself willen maken van het bestand fed60.txt. Helaas kent sed geen optie voor 'ignore case':

1. grep -i itself ~vannoord/Federalist/fed60.txt (door met pipe:)
   | sed -e 's/[Ii][Tt][Ss][Ee][Ll][Ff]/#&#/'>lines

Het bestand lines ziet er als volgt uit. We hebben het speciale symbool # gebruikt om de key word duidelijk aan te geven (voor cut hieronder).

regulating its own elections to the Union #itself#. It is not
gain admittance into them, it would display #itself# in a form
preference in which #itself# would not be included? Or to what
#itself# could desire. And thirdly, that men accustomed to

2. We maken aparte bestanden voor contexten en key word met behulp van cut. cut haalt stukken uit een tekstregel. Er zijn allerlei opties, de belangrijkste op dit moment:
   cut -c met getallen of range getallen: de tekens op die posities, bv cut -c 2,5-8,14- haalt uit elke regel de characters op positie 2, 5 t.m. 8 en van 14 tot eind
   cut -f met getallen of range getallen haalt de gewenste fields (velden) uit de regel. Default zijn die gescheiden door tab (vgl paste) maar je kunt een eigen 'delimiter' opgeven via -d, zoals een spatie voor woordgrenzen (-d' '). Hieronder gebruiken we de toegevoegde # als delimiter.

```
cut -d# -f 1 < lines > before
cut -d# -f 2 < lines > itself
cut -d# -f 3 < lines > after
```

3.Maak de context lang genoeg door met sed bv 35 spaties toe te voegen aan begin / einde van de regel. Helaas 'verstaat' sed geen opgave van een specifiek aantal

```
sed -e 's/^/                    /' < before > before2
sed -e 's/$/                    /' < after > after2
```

4.Verkort iedere regel tot bv 35 characters. Snap je waarom de before-file hiervoor omgekeerd wordt?

```
cut -c 1-35  < after2 > after3
rev before2 | cut -c 1-35 | rev > before3
```

5.Resultaat wordt verkregen met:

```
paste before3 itself after3
```

De hele set commando's zoals je ze van de terminal zou geven bij elkaar:

```
grep -i itself ~vannoord/Federalist/fed60.txt | sed -e
's/[Ii][Tt][Ss][Ee][Ll][Ff]/#&#/'>lines
cut -d# -f 1 < lines > before
cut -d# -f 2 < lines > itself
cut -d# -f 3 < lines > after
sed -e 's/^/                    /' < before > before2
sed -e 's/$/                    /' < after > after2
cut -c 1-35  < after2 > after3
rev before2 | cut -c 1-35 | rev > before3
paste before3 itself after3
```

Beperkingen

- alleen eerste match van de zoekstring op een regel wordt gegeven
- regelnummers ontbreken(kan makkelijk worden toegevoegd, zie huiswerk)
- de context wordt in karakters i.p.v. woorden geteld (zie huiswerk)
- alleen context van dezelfde regel, geen zinnen

5. Zipf's law

It is very interesting to draw the following diagram: after having put our words in decreasing order of frequency, we don't look at the words themselves, just at the frequency `f` in the function of its rank `r`. So `f(1)` is the frequency of the most frequent word, `f(2)` is the frequency of the second most frequent word, ... `f(100)` is the frequency of the 100th most frequent word, etc.

This `f(r)` function has interesting properties. In most of the cases, it has the same form, independently of the language, style or content of the given text. Not speaking of the first couple of values (the samllest ranks), this function fits very well a hyperbole, i.e. the `y = c / x` function, where `c` is some constant. To be more precise, a good approximation is the following power law:

```
f = c / r^(-a)
```

where `a` is a little bit smaller than 1 (approx. 0.7 - 0.8). (Remark: ^ means here "to the power of...".)

The fact that the Zipf-function has such a power-law behaviour, and is independent of the kind of the text, is very surprising, and generated a nice literature since the 1930s, 1940s, when the law was discovered (without the use of computers, yet!). For more information consult me (i.e. Tamas).

You can find here a nice program in Perl, by Henny Klein, that does the Zipf analysis of a text.

Bíró Tamás:
e-mail
English web site
Magyar honlap

Last modified: Thu Jul 3 11:39:17 METDST 2003