Tekstmanipulatie, week 11

1. Commands for presenting text: `echo`, `cat`, `more`, `less`

'echo' echoes a text to the screen?

```
birot@garmur:~ > echo hi, I am the teacher!
hi, I am the teacher!
```

What is this useful for???

'cat' (con)catenates files (links them together, as a chain), and prints them on the screen

cat file1 [file2...]

"Pager" programs: the first one used to be 'pg' ...

'more' Prints the given file to the screen, page after page (Space: next page; Enter: next line; q= quit more; ? or h: help ==> check it for more help!)

'less' Similar to 'more' but knowing... even more. It is easier to move backwards, and based on 'more' and 'vi'.

2. Standard input and output, pipes(|, <, >)

What is 'echo' useful for? And what have we been doing with 'cat' so far?

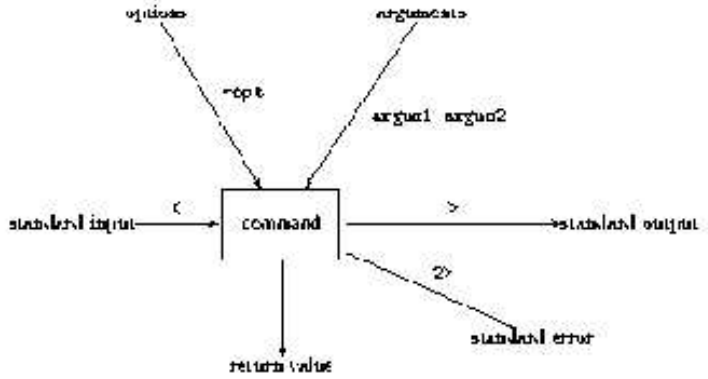
The manual of 'cat':

DESCRIPTION

Concatenate FILE(s), or standard input, to standard output.

What is that?

Remember, in UNIX everything is file... Even the screen and the keyboard! So in fact 'echo' puts a text into an output file, and if no file is specified, this file is the so-called "standard output', that is the screen. Similarly, 'cat' puts the concatenation of the input files into an output file. If no input file is specified, then this is the "standard input", that is the keyboard. And if no output is specified, this is the standard output, the screen.



In fact, each command has one input (standard input) and 2+1 outputs (standard output, standard error + return value). In addition, a program may receive information from its command line (arguments), and may read from and write to other files. However, a typical Unix program tries to avoid using other files: the Unix-philosophy is to use the standard input, the standard output, the standard error, the arguments in the command line and the return value only.

In fact, it is the shell that will feed the program's standard input and arguments, and make use of its standard output, standard error and return value. Last week , we saw how shell preprocesses the command line you typed in, for instance by resolving wild card expressions, before passing on the arguments to the program in question. This week, we are learning how you can make shell feed the standard input, and how you can make shell redirect the standard output of a command. In a few weeks from now on, we shall see how some commands (e.g. 'if', 'while', 'until') make use of the return value of other commands.

1. **Arguments** of the program: it is used to communicate a few *parameters* to the program you wish to run, but it is not a good idea to make long texts enter the program this way. However, many programs may receive file names as arguments, and then the content of these files can be used by the program.

Options are also special arguments to the Unix commands.

2. **Standard input**: a "stream" or "flow" of information that enter the program. The program can receive quite a lot of information through its standard input, as compared to the amount of information originating from the command line. The typical programs (Unix commands) we are going to use work on texts (on files seen as texts), and they read their standard input line-by-line (one line = a string of characters not containing a new-line-symbol, followed by a new-line-symbol). There are three ways you can feed the standard input of a program: either by typing something through the keyboard (by default), or by redirecting the standard input to an already existing file (<), or by redirecting the standard input to the standard output of another program / command (pipe-line: |).
3. **Standard output**: the "stream" or "flow" of information through which the results leave the program. By default, it is the screen. You can, however, redirect the standard output: to a file (>: overwrites the file, if the file already exists; >>: appends to the file, if the file already exists), or to the standard input of another command (pipe-line: |).
4. **Standard error**: another stream of information leaving the program. In most of the cases, it is used to report about possible problems arising while running the program. You may use it in order to debug your program. (In bash, you can redirect the standard error to a file by using 2>).
5. **Return value**: a number between 0 and 255 (or -127 and 128), which informs you about the result of the program. However, most often the real result is sent to the standard input, and the return value bears information rather on the way the program has run. Thus, usually if the return value is 0, it means that the command has run successfully. If an error has occurred, the return value may typically be -1 (=255), or something else, depending on the type of error encountered. (In the case of some commands, such as `grep' and `expr', the return value is defined in a different way. The usefulness of these definitions will become obvious later on.)

Remark: the C language is very closely related to Unix. The almost inevitable stdio.h include file's name refers to "standard input and output". This file includes, among others, the C commands printf and scanf, which write to the standard output and read from the standard input, respectively. Furthermore, you are always given for free the file streams stdin, stdout and stderr, which stand for standard input, standard output and standard error. Thus, when writing a C program according to the Unix-philosophy, you may use these three streams. The arguments (and options) appearing in the command line used to launch the program can be accessed through the arguments of the function `main' (consult a C manual for further details), whereas the return value of the program can be set in the return command of the function `main'.

Redirection of the input and output into files:

< the input is taken from the specified file

> the output goes to the specified file (if it already exists then it is overwritten, the previous content is lost)

>> the output goes to the specified file, but if it already exists then the output is appended to its previous content

| pipe: the output of the previous command is used as the input of the next one

`...` the expression is replaced by the output of the command line appearing between the *back quotation marks* (very useful when you want turn the output of a command or a pipe-line into the argument of another command)

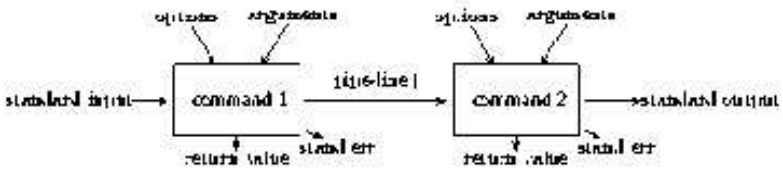
2> redirects the standard error in bash shell (in other types of shell there may be other ways to do that)

The Unix Lego game:

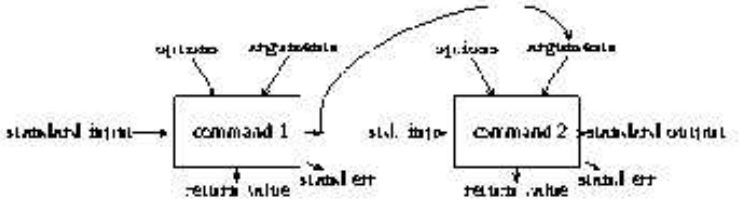
The initial idea behind Unix was something similar to the Lego game (hence, the name `Unix'). You have some basic building blocks, each of them being very simple, performing some basic task. Furthermore, you have some standardised ways of combining these building blocks, and if you are creative enough, you can solve very complex problems by combining the basic building blocks. In Lego, each building block has holes and knots of a standard size; in Unix, you have standard input, standard output, command line arguments and return values. (Standard errors are used to debug possible problems.)

How can you combine two commands?

You can use < to redirect the standard input of a command to a file. Similarly, > and >> will redirect the standard output of a command to a file. If you want to combine two commands, you can redirect the standard output of command 1 to a file, and then feed the standard input of the second command with that file. A simpler way is to use a *pipe-line*: command 1 | command 2.



Quite often, you will wish to have the standard output of some command (or command line) appear as the argument of another command. For that purpose, use the *back quotation mark* (`...`; NB: not the same as the single quotation mark used as an escape character): command2 `command1`.



Examples:

Now we understand what does 'cat file_name' and 'cat > 'file_name'.

```
echo hello! how are you? > welcome

ls -l ../Mail >> list_of_mails
```

`ls | cat` (What is the difference between this and writing simply 'ls'???)

```
echo Hi! "          " Hi again! > /dev/tty/3
```

`tty` prints the file name of the terminal connected to standard input.

Three further useful tools:

`tee` read from standard input and write both to standard output and to files: like a "T-joint" in a plumbing installation

Example:

```
cal 2002 | tee calendar2002 my_calendar | more
```

The result of this command line is that the calendar for the year 2002 will be saved both into files calendar2002 and my_calendar, on the one hand; and will be passed on further in the pipe-line (to the command 'more', which will visualise it page-by-page), on the other hand.

How to combine more commands into one line? Use the semi-colon (;) in order to make several commands run one after the other. What is the difference between ; and |? In the case of ;, the commands are run independently, whereas in the case of |, they form a pipe-line, i.e., the standard output of the one enters the standard input of the other.

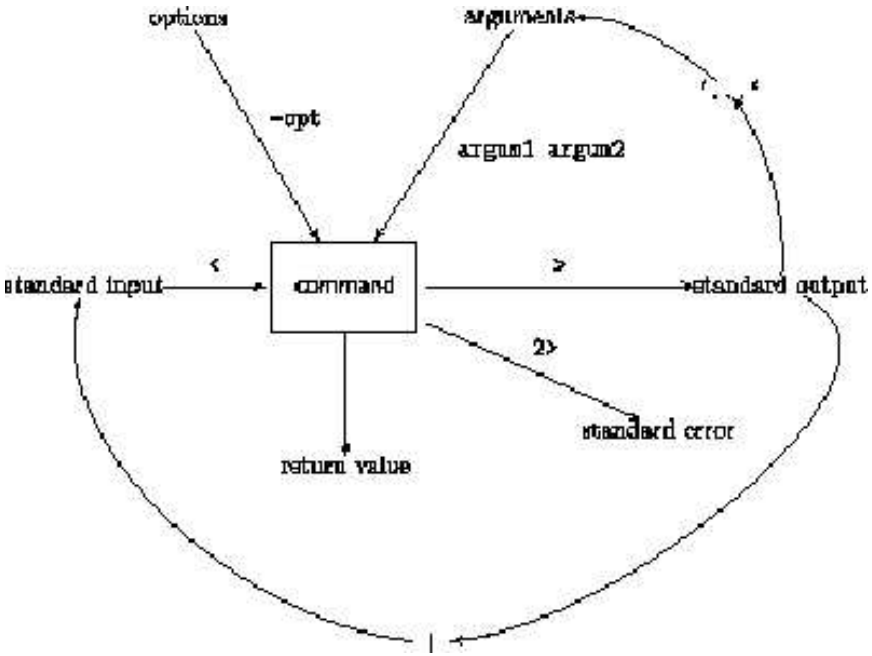
Further, you can use parentheses to combine the standard output of a series of commands. Example:

```
(ls ; ls ..) > apple
```

In this case, the standard output of the subsequent commands (`ls', followed by `ls ..') are combined together (concatenated). But, instead of going to the screen, they go to the file called `apple', due to the redirection of the standard output.

Obviously, the semi-colon and the parentheses are metacharacters, which you have to escape in cases such as a command line like `expr \( 3 + 4 \) % 6`.

A summary so far:



Information can enter a command either through standard input (keyboard by default; or file if you redirect it using <) or through its arguments (and options). Information

comes out from a command through standard output (screen by default; or file if you redirect it using >), standard error (screen by default; or file if you redirect it using 2>), and the return value. Use a pipe line (|) to redirect the standard output of a command to the standard input of another command. Use back quotation marks (`...`: in the upper left corner of most keyboards) to redirect the standard output of a command to the argument of another command.

Note that there are basically three kinds of commands:

1. 'echo' and 'expr' have nothing to do with files. They have no standard input. They receive information only from their arguments, and they see their arguments as strings of characters. They do something with their arguments, and they send the result to the standard output (that is the screen, unless you redirect it). Nevertheless, if you use wildcards in their arguments, shell will match the wildcards to file names. But then, the command itself does not know that these were file names, 'echo' and 'expr' uses them just as strings of characters..

2. 'mv', 'ls', 'cp', 'rm' do not have standard inputs, either. Even more, they do not have standard outputs, except 'ls'. They receive file names as arguments, and do something with the files. If an error occurs, the message is sent to standard error.

3. 'cat', as well as the the commands appearing in the next section, are called "**filters**": they are like filters in a real water pipe line. They receive some flow of information through standard input, process it, and then they send that modified flow of information forward to the standard output. If an error happens, the message is sent to standard error. The simplest filter is 'cat': it just sends forward what it has received. The next section is about more advanced filters, modifying the information: before sending it forward, 'head' and 'tail' may cut off some piece of information, 'tr' and 'rev' changes the information, and so on. Therefore, the key idea is that these commands receive their incoming flow of information from their standard input. But, for *most* of these commands you can also specify file names in the argument list. What happens in such cases is that the command reads the content of the files specified as its arguments, one after the other, and this will constitute the input of the command, and not the standard input. If you want to send one file

through a filter, it is almost the same if you give the file's name as a redirected input ('< filename') or as an argument (but see the difference for 'wc'). If you give more file names (for example by using wild cards), they will be catenated (linked one after the other), and it is this chain of files which will be processed by the command. Because 'cat' does not do anything to its input, 'cat' will simply catenate (link together) the content of all the files specified in its argument list into one output file (to the screen in the default case).

3. More commands for manipulating texts (`head`, `tail`, `rev`, `tr`, `wc`, `sort`,...)

Introductory remark: a file in Unix should be seen as a sequence of lines, each of them being a sequence of characters, ending with an end-of-line signal.

(Check always the manual for more options...)
(Pay attention: file names are sometimes to be put as part of the argument list, but in other cases they are the input, therefore < should be used.)

head outputs the first part of a file (first 10 line by default)

```
head -c N     prints the first N bytes
head -n N     gives the first N lines
```

tail outputs the last part of a file (first 10 line by default)

```
tail -c N     prints the last N bytes
tail -n N     gives the last N lines
```

rev reverses the lines of a file: The rev utility copies the specified files to the standard output, reversing the order of characters in every line. If no files are specified, the standard input is read.

tr translates / transforms a file (st. input to st. output) by translating, squeezing or deleting characters

`tr -d [set]` : deletes all the tokens of these characters

`tr [set1] [set2]` : replaces all the tokens of the i-th element of set1 with the i-th element of set2.

e.g. `tr [1-9] [a-i]`

`tr -s [set1] [set2]` : squeezes all repetitions of characters in set1 (in set2, if -d is present) into a single character.

e.g. `tr -s [\ ]` : all sequences of spaces are condensed into a single space

`tr -d -c [0-9]` : remove all non-numerical characters (-c : complement of set1)

Note: on some system the [ ] brackets should be left out. If something does not work, try it this way.

A very handy property of tr is that you can refer to the ASCII code of characters through it. We shall use it very often. The way to do it is: '\XXX', where the quotation marks help to escape the \ character, and XXX is the octal code of the character we want to refer to. For instance, you can change all spaces to a newline character (\n, whose ASCII code is 10, i.e. 012 in octal) in one of the following ways:

```
tr ' ' '\012'
tr ' ' '\n'
```

If you need the ASCII code of some character, use man ascii or man iso_8859-1.

wc Prints line, word, and byte counts for each FILE, and a total line if more than one FILE is specified. With no FILE, or when FILE is -, reads standard input.

- -c print the byte counts (compare with the file size given by 'ls -l' !)
- -m print the character counts
- -l print the newline counts
- -L print the length of the longest line
- -w print the word counts

sort Write sorted concatenation of all FILE(s) to standard output.

- -b ignore leading blanks in sort fields or keys
- -d consider only blanks and alphanumeric characters in keys (by sorting)
- -f fold lower case to upper case characters in keys
- -r reverse the result of comparisons
- +N shift N columns when sorting, i.e. sort according to the N+0- th column.

uniq Looks for repeating lines (next to each other!) and is able to manipulate them. If no option is given, then it removes the duplicate lines from a sorted file: discards all but one of successive identical lines from INPUT (or standard input), writing to OUTPUT (or standard output).

- -c puts a prefix before each line, giving the number of occurrences
- -d outputs only the self repeating lines, each of them only once
- -u outputs only the NON self repeating lines

Examples:

1. How to extract the middle of a file, from line number N to line number M? Solution: `head -n M | tail -n M-N+0`

2. At-bash: `tr [a-z] [zyxwvutsrqponmlkjihgfedcba]`. At-bash is a sort of technique in Jewish mysticism: you replace the first letter of the alphabet (aleph) to the last letter of the alphabet (tav), and vice-versa, then you replace the last-but-one letter of the alphabet (shin) to the second letter (beth) and vice-versa, and so on.

3. What does `ls | wc -w` ?

4. How to change the English way of giving a number (e.g. 987,654,321.33) to its European counterpart (987.654.321,33 = 987 billion 654 million 321 and 33/100)? You need to transform the .'s to ,'s and the ,'s to .'s.

5. How to reverse the order of the lines? `tr '\012' ÷ | rev | tr ÷ '\12' | rev`. How does this work? First you transform each newline symbol ('\012') to whatever symbol that does not appear in your text. This way, you get one long line, which you can reverse by using 'rev'. Then you want to get back your original lines, so you transform back the symbol used instead of the new line symbols (÷) into the real ones ('\012'). This way you will have reversed the order of the lines, but each line needs to be reversed again. Try it out, step by step.

- You have a file named 'file_name' which contains the name of another file the lines of which you want to reverse. What you can do then is: `rev` `` `cat file_name` ``. Then shell will replace `` `cat file_name` `` with the content of 'file_name' that is the name of the file that should be processed by 'rev'.

- Imagine you have a file called * in your directory (created by something like `cat > \*`). Now you type `echo `ls``. What happens? Try it out! You get all your filenames listed twice, except of the file named *, that will be listed only once. Why? Shell will first replace `` `ls` `` with the all your file names in the actual directory, that is *, file1, file2, etc. So you get a command line like `echo * file1 file2`, etc. In the next step, shell replaces * with all possible file names, so you get the command line `echo * file1 file2 file1 file2`. The arguments will then be echoed on the standard output, therefore the output will be: `* file1 file2 file1 file2`.

- You have a file "names" that contains names, some of them more than once. Imagine that you want to create two files. File "sorted" contains the names ordered alphabetically, but if a name occurs more than once, then you keep them more than once. Whereas file "once" contains the names sorted alphabetically, and each name occurs only once, but the number of occurrences in the original file are given. You can do that within one pipe-line, if you put a T-junction into it, that is you save the intermediate stage into a file, before the pipe-line goes further: `sort name | tee sorted | uniq -c > once`.

4. bc

In order to make easy calculations you can use the 'bc' command (bell's calculator). Type `bc <RETURN>` and you can immediately type in any expressions, like 3+4 or (45/3400)*100. In fact, similarly to the way we were writing short files by using 'cat', we are just using the fact that this command needs an input file, and if nothing else is specified, then it is the standard input. Therefore the program can be ended by ^d (CTRL + D: end-of-file). Or, alternatively, by ^c (CTRL + C: stop the running program).

Therefore why not doing things like:

```
echo 3+4 | bc
echo 23/46 | bc
```

Hey! Why is 23 / 46 = 0 ?! Because, if otherwise not specified, bc works with integers. Type ' scale = 4 ' to be able to receive your results with four decimals.
How to do this within one command line? You need an input file of two lines, which can be easily generated if you remember the role of the semi-colon and the parentheses:

```
(echo scale = 4; echo 5/8) | bc
```


What does

```
echo 13 % 3 | bc
```

mean? The remainder of the division. And what is the problem with this one:

```
echo (13/26)*4 | bc
```

Try rather

```
echo \(13/26\)*4 | bc
```

... and remember how you have to escape metacharacters!

Excursus

(From Henny's page.)

1. Het nut van woordenlijsten

Je kunt van een of meer teksten een alfabetische woordenlijst maken, of een frequentielijst: hoe vaak komt elk woord voor?

Een woordenlijst (unieke woorden) geeft het vocabulaire weer (interessant bij b.v. kindertaal, speciale talen). In
principe geldt: hoe meer tekst je ter beschikking hebt, hoe groter vocabulaire, maar die toename wordt natuurlijk steeds
minder, en bij b.v. een programmeertaal heb je alle mogelijke 'woorden' al heel snel binnen.
Bij kindertaal onderzoek wordt de type/token ratio gebruikt: aantal verschillende woorden gedeeld door totaal aantal
woorden. De lengte van de tekst speelt ook een rol:

Hoe groot is type/token ratio bij een tekst van 2 woorden?
Als je 6 romans van Vestdijk neemt en er nog 6 bijdoet, wat gebeurt er dan met je type-token ratio?

Daarom wordt type-token ratio meestal genormaliseerd door in tekststukken van vaste lengte te tellen, bv steeds 1000

woorden en de gevonden ratio's te middelen.
Een woordenlijst met woordfrequenties erbij is o.m. nuttig voor taalonderwijs, en voor het opzetten van
taalexperimenten (weten wat frequent gebruikte woorden zijn), voor tekstanalyses: wat zijn woorden specifiek voor
deze auteur of deze tekst? CELEX is een database met allerlei gegevens over Nederlandse woorden, o.a.
frequentiegegevens.

2. Hoe maak je een woordenlijst?

Je wilt een tekst in woorden splitsen en die gesorteerd onder elkaar zetten, of ze tellen en op frequentie sorteren. Als
je een tekst in woorden wilt splitsen moet je een aantal beslissingen nemen: zijn getallen ook woorden? Is er een
verschil tussen Kok en kok? We houden het even simpel: getallen tellen niet mee, en we maken geen onderscheid
tussen hoofd- en kleine letters. Bij het splitsen zelf duikt een probleem op: tussen woorden staan spaties, maar hoe
krijg ik de leestekens weg? Het volgende programmaatje neemt dit alle voor zijn rekening:

3. tr (translate/transform)

Transformeert 1 karakter (!!) in een ander karakter, je kunt wel een reeks wijzigingen tegelijk opgeven, maar geen
combinaties van letters zoeken en vervangen.

Als je een tr commando op een flinke file toepast, zie je vaak beter of je aan alles gedacht hebt, maar als je een
speciale opdracht even uit wilt testen (zoals in de eerste 2 opdrachten) kun je ook direct vanaf de terminal werken:

tr string1 string2
zet je tekst
veranderde tekst verschijnt

zet je tekst
veranderde tekst verschijnt

totdat je met Ctr-D of Ctr-C het programma tr weer verlaat

Als voorbeeldmateriaal in het practicum gebruiken we de teksten in directory /users1/vannoord/Federalist. Voor
achtergrond informatie over deze teksten, zie George Wellings' imposante USA project, meer in het bijzonder deze
bladzijde.

Basis syntax tr: tr string1 string2 <file1 . Voorbeelden:

```
tr e q < fed1.txt
tr eE qQ < fed1.txt | less
```

redirection van standard input, standard output; pipes.
onderscheid hoofdletter / kleine letter van belang

```
tr aeiou qqqqq < fed1.txt | less
tr A-Z a-z < fed1.txt > myfed1.txt
tr aeiou [q*5] < fed1.txt | less
tr aeiou [q*] < fed1.txt | less
tr ' ' '\012' < fed1.txt | less
tr ' ' '\n' < fed1.txt | less
```

De laatste twee commando's vervangen een spatie door een newline. ASCII tekens kun je aangeven via hun
nummer, hier \012, zie man ascii; sommige tekens kun je aanduiden via speciale karakters, zoals \n, \t (tab). De
backslash is een zgn. Escape-karakter, dat aangeeft dat het teken erachter een bijzondere betekenis heeft.
Quotes zijn soms nodig bij vreemde tekens.

Opties van tr:
-s (squeeze) perst aantal dezelfde tekens samen
-d (delete) verwijdert tekens
-c (complement) neemt alle tekens die NIET in string 1 voorkomen.

```
tr -s ' ' '\012' < fed1.txt | less
tr -d '.' < fed1.txt | less
tr -c aeiou [q*] < fed1.txt | less
tr A-Z a-z < fed1.txt | tr -cs a-z '[\012*]' > fed1.txt
```

let op: ook hier is * noodzakelijk!

De laatste opdracht is het begin van ons woordenlijstprogramma: het leest fed1.txt, verandert eerst hoofdletters in

kleine letters, verandert vervolgens alle tekens die geen kleine letter zijn (-c a-z dit betreft dus cijfers, spaties,
leestekens en newlines!) in een newline karakter \012, waarbij -s zorgt dat meerdere van die tekens samengenomen
worden (niet meer dan 1 newline achter elkaar), en schrijft het resultaat, een lijst van alle woorden, in volgorde van de
tekst, naar een nieuwe file.

4. Woordenlijst

1.programma 1: plaats ieder woord op een eigen regel:

tr (zie boven)

2.programma 2: sorteer regels (en geef alleen unieke woorden)

sort (-u)

met sort -u krijg je een alfabetische lijst unieke woorden

3.programma 3: geef unieke woorden in reeds gesorteerde lijst waar nog dubbelen in zitten (en tel ze)

uniq(-c)

met de opeenvolging sort|uniq-c krijg je een lijst unieke woorden met hun frequentie ervoor: een frequentielijst

4.programma 4: sorteer de woordenlijst met frequenties ervoor, hoogste aantal boven:

sort -nr

sorteer numeriek -n, anders krijg je de 'alfabetische' volgorde van de cijfers, b.v. 10, 100, 2 en reverse -r, dat wil
zeggen met het hoogste getal bovenaan

Gebruik de UNIX pipe om de 4 commando's te combineren.

Bíró Tamás:
e-mail
English web site
Magyar honlap

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