1. grep -w checks for whole words, not just substrings
2. grep -o will only return the matches (not the entire line)
3. The - character also has to be escaped
4. NR = number of total lines; NF = number of fields in the current line
5. length($1)
6. head -n 10 file.txt => prints first 10 lines
7. tail -n 10 = prints last 10 lines
8. tail --lines=+2 newfile.txt - starts from the 2nd line (1-indexed)
9. sed -i will actually replace things in the file
10. awk -F'[:]' 'BEGIN {sum = 0} $13 > $14 {sum += $13} END {print sum}' last.fake
11. \> means end of word
    a. $ means end of line
    b. These 2 can be combined
12. we can only access the first 9 arguments; we need “shift n” to get the next ones
13. lineCount=`cat file | wc -l`
14. wc -m = number of characters
    a. wc -w = number of words
15. cut -c 2-4 file.txt = only displays the characters between 2 and 4 from file.txt
16. x=`expr $x + 1` ⇔ x++
17. checks if it's even: `expr $x % 2` -eq 0
18. checks if argument is a file / directory (in an IF): “-f $arg”, or “-d $arg” (don't use "type", like in a for)
    a. -x is for executables
    b. man test shows all these types
19. chmod u=rwx,g=rx,o=r myfile (permissions for owner, group, others)
20. **elif** not else if
21. [$foundFile -gt 0 -a $foundDirectory -gt 0];
    a. -a means &&
    b. -o means ||
22. when you want to remove an entire line: sed "/pattern/d" $file
23. be careful when using sed to check for entire words, so "\<word\>" not just "word"
24. - when you want to grep a string (which is not a file name): echo $string | grep "pattern"
25. you can do: if grep "pattern" then ..

26. Use printf syntax in awk: awk '{printf "%s@scs.ubbcluj.ro", $1}'
27. use the command “file” to determine the type of a file
28. for the sleep command we need unistd.h
29. fgets(string, MAX_STRING_LENGTH, stdin) ⇔ read a line from the keyboard
30. srand(time(NULL)) ⇔ reset the random generator
31. sed "s/\([0-9]\)/\1\1/g" file.txt ⇔ prints a file with all the digits duplicated (if we also use -i => the file is actually modified)
32. grep "[aeiouAEIOU]$" aux.txt ⇔ prints the lines which end in vowels
33. echo $(dirname `pwd`) ⇔ prints the parent directory
    a. echo ~ ⇔ prints the root
34. if ! [-f $arg -o -d $arg]; ⇔ checks if the argument is not a file nor a directory (beware of the negation sign)
35. A non-zero return value in shell => false; 0 => true
36. A child process is a zombie from the moment it finishes execution until its parent calls wait or waitpid
    a. Call wait or waitpid for each process you create
37. CTRL-C ⇔ SIGINT ⇔ signal no. 2 ⇔ kill -2 pid
    a. SIGTERM = 15
    b. SIGKILL = 9
    c. man 7 signals shows all of them
38. signal(SIGINT, f) => whenever SIGINT is received, the function f is called
    a. This should be called at the beginning of the program
    b. This works for all signals but SIGKILL - this cannot be stopped or modified
    c. Needs #include <signal.h>
39. Whenever a child process stops, the parent receives a SIGCHLD
    a. signal(SIGCHLD, SIG_IGN) will basically prevent zombie creation
40. SIGUSR1 and SIGUSR are “set aside” so that we can use them in any way we want
41. Running other programs:
    a. Searches PATH for the program
        i. **execvp**(“grep”, a)
            1. char* a[] = {"grep", "/an1/gr911/", "/etc/passwd", NULL}
        ii. **execlp**("grep", "grep", "/an1/gr911/", "/etc/passwd", NULL)
    b. Doesn’t search PATH for the program
        i. **execv**("/bin/grep", a);
            1. char*a[] = {"/bin/grep", "/an1/gr911/", "/etc/passwd", NULL}

        ii. **execl**("/bin/grep", "/bin/grep", "/an1/gr911/", "/etc/passwd",NULL)
    c. NULL marks the end of the arguments
    d. Path = all the places in which an executable might be located

42. First argument (0) of a program is always the command name
43. If we close a pipe before calling fork(), the child will inherit a closed pipe
44. pipe[0] = read; pipe[1] = write
45. **popen**:
    a. We can call other programs/ scripts from a C program and also get its output or provide some input to it
    b. Basically an exec which doesn't overwrite the current program
46. **dup**/ **dup2**
    a. <u>File descriptor table</u>: basically every file in a program will have a handle, which indicates what we can do with it / knows how to operate it;
        i. e.g. "open" will open a file to reading or writing -> its handle is stored in the fd table on the first available position; its index will be the value of the handle
        ii. Positions 0, 1, 2 are usually "reserved" for (console) stdin, stdout and stderr
        iii. A pipe will have 2 of these file descriptors: one for reading and one for writing
    b. int **dup**(int oldFD) - copies the fd given as an argument to a new position in the table; so now, when we want to do a certain operation, we can do it from either of the fd's
    c. int **dup2**(int oldFD, int newFD) - overwrites the old fd with the new one (so for example we can overwrite position 0 - stdin to have the handle for a fifo); it also **closes the old fd before overwriting it**
    d. Exec will not overwrite the file descriptor table, so if we do a dup, then an exec, the new program will use the same fd table
    e. The exec will also close the files that it used (so if we have a pipe, it will close the part that it used, but obviously not both ends)
    f. **Undo** dup calls: before using the fd, make and store a copy of it using dup(), then do whatever you want with dup2(), then set the fd back to its previous value, which was stored in the beginning
47. Shared memory
    a. IPC = interprocess communication
    b. Each(IPC I think ?) has an unique ID; they remain in the system and we need to clean them up

48. **Threads**
    a. They don't copy data from the process (as opposed to fork), and will also need another stack
    b. If a thread fails, all other threads fail
    c. **Race condition**: multiple processes/ threads want to access the same resource (=critical resource) simultaneously
    d. **Semaphore** - doesn't allow more than n threads at a time to access a resource (n is set when initializing it)
    e. **Barrier** - waits for all n threads to reach it, and only then does it allow (all of) them to go on
49. **Monolithic os**
    a. they are basically one big program
50. **Microkernel os**
    a. it's like a dispatcher which transfers messages between different components; in this example, if a driver (which is one of the "components") fails, it does not bring the entire system down, like in the monolith's case
    b. The BIOS is kind of like a microkernel OS
51. Process states
    a. **HOLD** - a process which is almost ready to enter the system
    b. **READY** - the process is in the memory and is ready to be served by a processor
    c. **RUN** - the process is running
        i. This will alternate quickly with
            1. **WAIT**
                a. The process waits for a "slow" operation (usually disk operations); while it does that, it's no longer running, so that other processes can run
            2. **SWAP**
                a. The RAM is full, so the OS "dumps" a processes' memory on the disk, temporarily, so that another process can run
    d. **FINISH** - the process is done
52. **Semaphore** = pair (v(s), c(s))
    a. v = the value of the semaphore, integer; when it's >= 0 => a new process can be run, otherwise it will be made to wait
    b. c = a queue which contains all the processes that are WAITing
    c. It has 2 operations

   i. **P** will try to run a process; if it has space (v(s) > 0) => we run it, otherwise it's pushed to the queue and the process is set to WAIT
   ii. **V** will be called when ending a process; if there are processes in the queue (so if v(s) < 0), we will set one of them to READY and pop it out of the queue
   iii. In a way, P ~ lock, V ~ unlock

53. Livelock - it's not a deadlock, bc things are moving; however, they are not progressing at all
54. Segmented memory - it defines a section of memory which will have certain permissions/ properties;
55. Loading a process
    a. All at once - once it's loaded, it goes really fast; slow start-up time, might waste memory
    b. Load each page as needed - kind of faster start-up, but the execution is slower, no wasted memory
    c. "Locality principle" - usually when we load page x, we will also need pages x+1, x+2 .. (ex. in a for loop)
56. **NRU** (not recently used) algorithm - for each page we store 2 bits; 1 of them knows whether the page has been referenced recently, the other if the page has been modified recently => 4 classes, where the most important is the one with both bits 1 -> modified but not referenced -> referenced but not modified -> both bits 0
    a. Could be mixed with FIFO => better results
57. **LRU** (least recently used) - we have a matrix of size pages * pages; each time a page is used we fill its row with 1 and its column with 0 (in this order); the page with the least 1s on its row is the least used
58. When searching for where to allocate new, contiguous memory (malloc)
    a. First fit - search for the first available area in which it fits
    b. Best fit - search for an area s.t. the space left is the smallest possible; but we will be left with very small slices of memory
    c. Worst fit - opposite of best fit
    d. Buddy-system - we split the memory in chunks of size 2^k; whenever we need to allocate x, we search for the first available power of 2 >= x; if that is too large, we then divide what's left into other powers of 2, which will be free
59. **Caching**, from small and fast to the large and slow
    a. Registers -> L1 -> L2 -> L.. -> RAM -> HDD/ SSD

b. When we need to search for things/ their positions in cache =>
    i. Direct caching ~ hashing, so we will have collisions (thrashing)
    ii. No “hashing” - we just place it on the first available position, and when we need it, we search for it
    iii. Compromise (set-associative caches) - we split the cache into larger sets which are accessed directly; but inside them we go linearly, like in the no hashing method

60. Symbolic link - a new, different name for the same inode
61. Hard link - a new inode for the same data; can only be created by the root