**Cases - basic:**

Case 1:

1. Create two files, file1 and file2: touch file1 file2
2. Rename flie1 to file1: mv flie1 file1
3. Remove both files: rm file1 file2

Case 2:

1. Create a file named file1: touch file1
2. Create a new directory named MYDIR: mkdir MYDIR
3. Move file1 to the MYDIR directory: mv file1 MYDIR/
4. Remove the directory MYDIR recursively (including its contents): rm -r MYDIR

Case 3:

1. Create a file named file1: touch file1
2. Write "This is my file" to file1: echo "This is my file" > file1
3. Print the content of the file to the console: cat file1

Case 4:

1. Create a file named my\_file with the content "The original content.": echo "The original content." > my\_file
2. Copy the file my\_file with the name my\_file\_copied: cp my\_file my\_file\_copied
3. Edit the content of my\_file\_copied to "The edited content.": echo "The edited content." > my\_file\_copied

Case 5:

1. Open the terminal using keyboard shortcut: <ctrl><alt><t>
2. Show all files in /etc/cron.d/: ls /etc/cron.d/
3. Show the file rights (long listing option) in /etc/cron.d/: ls -l /etc/cron.d/

Case 6:

1. Check the help or manual of ls command: man ls
2. Find the option to list files recursively: ls -R
3. Show all files recursively in /boot: ls -R /boot

Case 7:

1. Change directory to /etc: cd /etc
2. List all files in /etc/cron.d/: ls /etc/cron.d/
3. Change directory to your home directory: cd ~

Case 8:

1. Create directory SOMEDIR: mkdir SOMEDIR
2. Rename the directory to NEWNAME\_DIR: mv SOMEDIR NEWNAME\_DIR
3. Remove the directory NEWNAME\_DIR: rm -r NEWNAME\_DIR

Case 9:

1. Create a file called date\_printer with the content date: echo "date" > date\_printer
2. Change the file permission to be an executable file: chmod +x date\_printer
3. Run the program: ./date\_printer

Case 10:

1. Show the content of .bashrc file using two different commands:

* cat ~/.bashrc
* less ~/.bashrc

1. Edit the file .bashrc by adding date in a new line: echo "date" >> ~/.bashrc

Case 11:

1. Show the file rights of all files in the current directory: ls -l

Case 12:

1. Run the top command: top
2. Kill the process using the keyboard shortcut: <ctrl><c>

**Medium Cases:**

Case 1:

1. Kill the process using the keyboard shortcut:

* ls /etc/a\*
* ls /etc | grep ^a

1. Check the last usage of the find command (in the history): history | grep find | grep -v history | tail -n1

Case 2:

1. Create files: 1a.jpg, 2.jpg, 3b.png, 4ef.svg, 5.png: touch 1a.jpg 2.jpg 3b.png 4ef.svg 5.png
2. Copy the JPG files to a newly created directory MYDIR:

* mkdir MYDIR
* cp \*.jpg MYDIR/

1. Remove all PNG files: rm \*.png

Case 3:

1. Create a file with numbers 1-10 (each number in a new line):

* seq 1 10 > numbers.txt
* echo “1<ENTER>2<ENTER>3…<ENTER>10” > numbers.txt

1. Show the content of the file: cat numbers.txt
2. Show only the first 3 lines: head -n3 numbers.txt
3. Show only the 4th line: head -n4 numbers.txt | tail -n1

Case 4:

1. Connect to your account via SSH: ssh [username@ssh.mini.pw.du.pl](mailto:username@ssh.mini.pw.du.pl)
2. Open a screen session and run the top command:

* screen
* top
* To detach from the screen session, press <ctrl><a>, then d.

1. Reattach the last screen session: screen -r

Case 5:

1. Change directory to /etc: cd /etc
2. List all files in /etc/cron.d/ using a relative path: ls cron.d/
3. List all files in Desktop (inside your home directory): ls ~/Desktop
4. Change directory to your home directory: cd ~
5. List all files in /etc/cron.d/ using a relative path: ls /etc/cron.d/

Case 6:

1. Create a variable MYVAR with value -al: MYVAR="-al"
2. Show the variable: echo $MYVAR
3. Use the variable MYVAR to list all files (with long listing) in the /etc directory: ls $MYVAR /etc

Case 7:

1. Open a screen session: screen
2. Start the script command: script
3. Show all files in /etc: ls /etc
4. Close script and the screen session:

* To close the script command, press <ctrl><d>.
* To close the screen session, press <ctrl><a>, then k.

1. Show the saved script: cat typescript

Case 8:

1. Create a directory DIR with 3 files (file1, file2, file3):

* mkdir DIR
* touch DIR/file1 DIR/file2 DIR/file3

1. Compress the directory (tar.gz): tar -czvf DIR.tar.gz DIR
2. Extract the compressed archive: tar -xzvf DIR.tar.gz

Case 9:

1. Ask a friend to open an editor (nano, pico, vi) or a file content viewer (less, more).
2. Edit the file: nano file (etc…)
3. If needed, ask them to close the opened editor/viewer and open the one they prefer.

Case 10:

1. Compose one-line command: List directory SOMEDIR, then if not exists create it, then on success list the content of current directory: ls SOMEDIR || mkdir SOMEDIR && ls

Case 11:

1. List all files in the current directory using the long listing format: ls -l
2. Find three other commands to do the same:

* ls --format=long
* ls -l .
* ls -l ./

Case 12:

1. Show the content of the .bashrc file using four different commands:

* cat ~/.bashrc
* less ~/.bashrc
* more ~/.bashrc
* head ~/.bashrc (just first few lines)

1. Edit the .bashrc file by adding a newline that will print "Today is ..." (with the current date): echo "echo 'Today is ...' \$(date)" >> ~/.bashrc

Case 13:

1. Create a new file: touch myfile.txt
2. Append the new file with "someline": echo "someline" >> myfile.txt
3. Repeat appending three times: As above but 3 times
4. Show the content of the file: cat myfile.txt
5. Overwrite the file with the only line: echo "the only line" > myfile.txt

Case 14:

1. Show the disk usage of the root directory with limited depth (1) and human-readable format: du -d 1 -h /

Case 15:

1. Create a file: touch myfile
2. Change the file rights so that only you can see, edit, and execute the file: chmod 700 myfile
3. Change the group of the file to ., which means no group ownership: chgrp . myfile
4. Delete the file: rm myfile

Case 16:

1. Open the Firefox browser.
2. How many processes connected with Firefox are running: pgrep -c firefox
3. How many resources do the processes consume? How much of them does Firefox consume: top -p $(pgrep -d',' firefox)

Case 17:

1. Run the top command: top
2. Open another terminal and find the top process ID: pgrep top
3. Kill the top process using its process ID: kill <top-process-id>

Case 18:

1. Open the file editor mousepad in the background: mousepad &
2. See all the processes connected with your terminal: ps -t $(tty)
3. Get the process into the foreground: fg %1
4. Kill the process:

* <ctrl><c>
* kill <process-id>

**Advanced Cases:**

1. Git:

* Why choose it: Git is a widely used version control system that allows you to track changes in files and collaborate with others on software development projects. It is an essential tool for developers and anyone working on code-related projects.
* Interesting feature: One of the most interesting features of Git is its ability to create branches. Branching allows you to work on different features or versions of your project simultaneously, making it easy to experiment, collaborate, and merge changes back together.

1. Regular expressions:

* Why choose it: Regular expressions are powerful patterns used for searching, matching, and manipulating text. They are used in various programming languages, command-line tools, and text editors to perform complex search and replace operations.
* Interesting feature: The most interesting feature of regular expressions is their ability to define complex patterns using a concise syntax. With regular expressions, you can search for specific patterns of characters, validate input, extract data, and perform advanced text processing tasks.

1. Bash scripts:

* Why choose it: Bash scripting is essential for automating tasks in a Unix-like environment. It allows you to write scripts that execute a series of commands, perform conditional operations, loop over data, and handle user input.
* Interesting feature: One of the most interesting features of bash scripts is the ability to handle user input. You can prompt the user for input, read their responses, and use that input to control the behavior of your script. This allows for interactive and dynamic script execution.

1. Vim:

* Why choose it: Vim is a highly configurable text editor known for its efficiency and powerful editing capabilities. It is a modal editor, meaning it has different modes for editing, navigating, and executing commands, making it a favorite among developers and power users.
* Interesting feature: One of the most interesting features of Vim is its command mode, where you can enter commands to perform various operations. With the right commands and shortcuts, you can navigate through files, search and replace text, and execute external commands without leaving the editor.

1. GCC (GNU Compiler Collection):

* Why choose it: GCC is a widely used compiler suite for programming languages like C, C++, and Fortran. It is known for its robustness, optimizations, and support for various platforms. Understanding GCC can help you build and compile software efficiently.
* Interesting feature: One of the most interesting features of GCC is its optimization capabilities. GCC provides various optimization levels that can significantly improve the performance of your compiled code. It also supports different optimization techniques, such as inlining, loop unrolling, and vectorization, which can make your programs run faster.

Git, Regex, Gcc <3. The only reasonable options <3

Git tutorial:

* git init: Initializes a new Git repository in the current directory.
* git clone <repository\_url>: Copies an existing Git repository from a remote server to your local machine.
* git add <file>: Adds a specific file to the staging area.
* git add .: Adds all modified and new files to the staging area.
* git commit -m "Commit message": Creates a new commit with the staged changes and a descriptive message.
* git status: Displays the current status of your repository, including modified files and untracked files.
* git log: Shows a log of all commits in reverse chronological order.
* git branch: Lists all branches in the repository.
* git branch <branch\_name>: Creates a new branch with the specified name.
* git checkout <branch\_name>: Switches to the specified branch.
* git merge <branch\_name>: Merges changes from the specified branch into the current branch.
* git pull <remote\_name> <branch\_name>: Fetches and merges changes from a remote repository into your current branch.
* git push <remote\_name> <branch\_name>: Pushes your local commits to a remote repository.
* git checkout -- <file>: Discards changes made to a specific file since the last commit.
* git reset HEAD <file>: Unstages a file from the staging area.
* git revert <commit\_hash>: Creates a new commit that undoes the changes made in a previous commit.

Regex tutorial:

* Matching literal characters is as simple as typing them. For example, the regex pattern cat matches the string "cat" in any text.
* Metacharacters have special meanings in regex. Some common metacharacters are: ., ^, $, \*, +, ?, (), [], {}, and \ . If you want to match a metacharacter as a literal character, you need to escape it with a backslash. For example, to match a period, use \..
* [abc] matches any single character "a", "b", or "c".
* [a-z] matches any lowercase letter.
* [A-Z] matches any uppercase letter.
* [0-9] matches any digit.
* [^0-9] matches any character that is not a digit.
* \* matches zero or more occurrences.
* + matches one or more occurrences.
* ? matches zero or one occurrence.
* {n} matches exactly n occurrences.
* {n,} matches n or more occurrences.
* {n,m} matches between n and m occurrences.
* ^ matches the start of a line.
* $ matches the end of a line.
* \b matches a word boundary.
* \d matches any digit (equivalent to [0-9]).
* \w matches any word character (equivalent to [a-zA-Z0-9\_]).
* \s matches any whitespace character.
* \. matches a period.

Bash scripts:

* Always start with: #!/bin/bash
* User input: read var

Ex.:

echo "What's your name?"

read name

echo "Hello, $name!"

* Script arguments: $n - n-th argument (command line arguments)

Ex.:

echo "Script name: $0"

echo "First argument: $1"

echo "Second argument: $2"

* Conditional statements (if): if[statement]; then something else (elif) something fi

Ex.:

echo "Enter a number:"

read number

if [ $number -gt 0 ]; then

echo "The number is positive."

elif [ $number -lt 0 ]; then

echo "The number is negative."

else

echo "The number is zero."

fi

* for loop: for item in “arr\_of\_items”; do something done

Ex.:

fruits=("apple" "banana" "orange")

for fruit in "${fruits[@]}"; do

echo "I like $fruit"

done

* for loop (2-nd version): just like in cpp

Ex.:

for ((i=1; i<=5; i++))

do

echo $i

done

* while loop: while [statement]; do something done

Ex.:

count=1

while [ $count -le 5 ]; do

echo "Count: $count"

count=$((count + 1))

done

Vim tutorial:

* Here you have as many pieces of information about Vim as my successful attempts to exit Vim (PLEASE HELP, I’M STUCK)

Gcc tutorial:

* Create the C program with multiple files: Assume you have files main.c, function.c,, function.h. main.c is dependent on function.c and function.c has a header file function.h.
* Compile the C program: gcc -c main.c functions.c to create object files, gcc main.o functions.o -o program to create executable from object files
* Run the C program: ./program