File System Notes ;

If a disk drive has 4096-byte blocks and uses a 64-bit addressing scheme, how many blocks can be addressed by the disk?
A 64-bit addressing scheme means that each block can be addressed using a 64-bit integer, allowing for a total of
2 ^ 64 unique block addresses.

In a file system with 16-bit block addresses and 8 KB blocks, what is the maximum size of the disk in bytes?
2^16 * block size

Formula

MaxFileSize = (DirectPointers + SingleIndirect + (doubleIndirect * doubleIndirect)) * BlockSize

MaxFileSize = if you have a 32-bit addressing scheme and a block size of 8 KB, the formula would be:
Maximum file size (in bytes)=2^32×2^13

Maximum file size = [direct DBA + Number of (Data block size/DBA)+Number of (Data block size/DBA)^2 +Number (Data block size/DBA)^3]* Data block size.
Data Block size = 1024 byte
(Dist Block size/DBA) = Number of Disk Block
address Store inside One Block.
Maximum file size = [10 + 1* (128)+ 1 *(128* 128*)+1*(128*128*128)]*1024 Byte
= Approx 2 GB.

UNIX file system (ufs) ufs supports journaling,hot fixes,the native file system is called the extended file system (ext or ext fs)

- /boot: components of the OS kernel
- /bin: system commands
- /dev: device drivers
- /usr: user software and commands
- /home: user data
- /etc: system data
- /var: a place for files that change often
- /log: system logs useful for administration

+ several others

$2^1 = 2$ $2^9 = 512$
$2^2 = 4$ $2^{10} = 1024$
$2^3 = 8$ $2^{11} = 2048$
$2^4 = 16$ $2^{12} = 4096$
$2^5 = 32$ $2^{13} = 8192$
$2^6 = 64$ $2^{14} = 16384$
$2^7 = 128$ $2^{15} = 32768$
$2^8 = 256$ $2^{16} = 65536$

A Unix-style I-node has 10 direct pointers and one single, one double and one triple indirect pointers. Disk block size is 1 Kbyte, disk block address is 32 bits, and 48-bit integers are used.
What is the maximum possible file size?
=
2^34 bytes

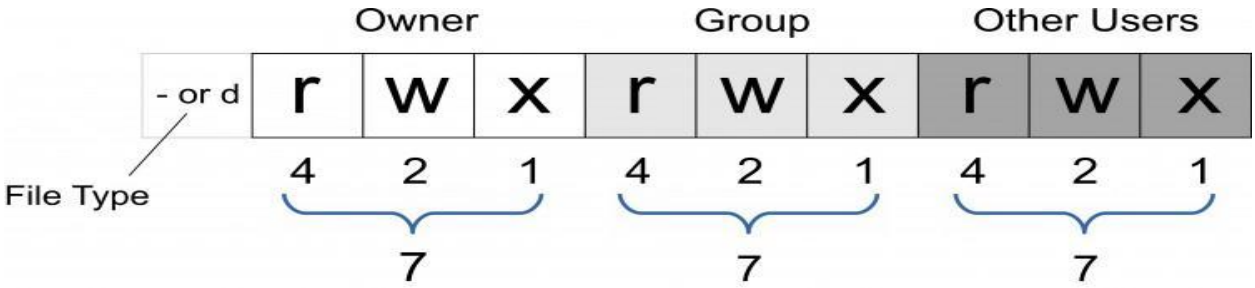
Create multiple directories, sub1 to sub9 using one command. | mkdir sub{1..9}
Create the following files (f1 f2 f22 f33 fa fb fa2 fb2 f2b f2a faa fbb) using one command. | touch f1 f2 f22 f33 fa fb fa2 fb2 f2b f2a faa fbb
copy all files starting with f and followed by a digit to sub1 | cp f[0-9] sub1
copy all files starting with f and followed by a lowercase letter to sub2 | cp f[a-z] sub2
copy all files starting with f and followed by two lowercase letters to sub3 | cp f[a-z][a-z] sub3
copy all files starting with f and followed by two digits to sub4 | cp f[0-9][0-9] sub4
copy all files starting with f and followed by letter lowercase and a digit to sub5 | cp f[a-z][0-9] sub5
copy all files starting with f and followed by any character to sub6 | cp f? sub6
copy all files starting with f and followed by letter lowercase and any character to sub7 | cp f[a-z]? sub7
copy all files starting with f and followed by two characters to sub8 | cp f?? sub8
copy all files starting with f and followed by any characters (zero or many) to sub9 | cp f* sub9
copy sub1 directory and its content into /home/sub2 directory | cp -R sub1 ~/sub2
list the current directory and all its contents recursively | ls -R
An absolute path -If a path has the root directory "/"as its first character it is an absolute path. Same for all user
A relative path-It does not have the root directory "/" as its first character. Different for different user
rm -r sub1 to remove with directory with content
[1-3]* matches all file names starting with "1","2" or "3" followed by zero or more characters.
[2-4][a-c] matches all file names starting with "2" "3" or "4" followed by "a", "b" or "c".
*[A-Z] matches file names ending in an upper case character.
[0-9]*txt* matches file names starting with a digit and containing "txt".
f[0-9][0-9]? matches all files with four character names starting in "f" followed by two digits.
[1-3]*[4-5] matches all file names starting with "1", "2" or "3" and ending in "4" or "5".
Copy to sub directory
cp g* ../../

```
#!/bin/bash

Double Quotes
All characters inserted between double quotes will keep thier
literal meaning except "$", "`" and "\ ".
```

u,g,o,a – user, group, other, all
chmod u+x filename adds execute permission for the owner of the file.
chmod u-x filename removes execute permission for the owner of the file .
chmod u=x filename adds execute permission and removes read and write permissions for the owner of the file.
chmod g+rx filename adds read and execute permission for the owners group members.
chmod g-rx filename removes read and execute permission for the owners group members.
chmod o+rx filename adds read and execute permission for other users of the file.
chmod a+rxw filename adds read, write and execute permission for all users of the file.
chmod a+w filename adds write permission for all users of the file.

Numeric file permissions



chmod 640 f1 changers the permissions of the file "f1".
chmod 640 f* changes the permissions for all files in a directory starting with "f".

- The cat command outputs the contents of a file. You can use this to view files.
- The sort command outputs the sorted contents of a file.
- The wc command outputs the number of lines, words and bytes in a file.
- The head -x command outputs the first x lines of a file.
- The tail -x command outputs the last x lines of a file.

$0 stores the name of the script.
$# stores the number of arguments.
$* stores the complete argument list.
$? the result of exit status.

The shell has several special parameters. These parameters are automatically assigned and can only be read. To read their value, use the $ sign in front of their name:
* expands to the positional parameters with which the script (or a new shell) are called (i.e. the script's arguments) as a single string. The separator (default: a space) can be chosen
1 - 9 each expands to one positional parameter (more than nine is possible)
@ expands to the positional parameters, as separate strings
# expands to the number of positional parameters
? expands to the exit status of the most recently executed command

$ expands to the process ID of the script (or shell)
0 expands to the name of the script (or shell)

```
ATEST2='moredata'
export ATEST2
```

#This is script echoes its first argument. echo $1
Change file permission from argument - chmod u+x $1 - (mv $1 $2 rename filename supplied via argument.

```
Read name from cmd and prints
echo -n "Enter your name:"
read NAME
echo "Hello $NAME"
```

if – fi or if else fi

```
if [[ $# -lt 3 ]]; then
echo "Insufficient arguments"
        exit 1
fi
echo $*
exit 0
```

compare two argument using if – fi

```
#!/bin/bash

if [[ $# -ne 2 ]]; then
   echo "Script requires exactly two arguments"
   exit 1
fi
if [[ $1 > $2 ]]; then
   echo "The first argument is greater than the second"
   exit 0
fi
if [[ $2 > $1 ]]; then
   echo "The second argument is greater than the first"
   exit 0
fi
echo "The two arguments are equal"
exit 0
```

print numbers from 1 to 10 using while loop

```
NUM=1

while [[ $NUM -le 11 ]]; do
    echo $NUM
    NUM=$((NUM+1))
Done
```

Print argument supplied in new line..using for loop

```
#!/bin/bash

if [[ $# -lt 1 ]]; then
   echo "Insufficient Arguments"
   exit 1
fi
for i in $*
do
   echo $i
done
```

String concate using for loop

```
#!/bin/bash
if [[ $# -eq 0 ]]; then
exit 0
fi
all_args=""
for arg in $*; do
all_args=$all_args$arg
done
echo $all_args
```

command > output.txt command 2> errors.txt command > output.txt 2>&1 ls file1 file2 > file22 2> file33

cat /etc/services | less

cut [options] [file(s)]
Cut columns out of input (“vertical” cutting, opposed to the “horizontal” cutting of head/tail)
Options
-c list characters to cut (example: -c 1-72)
-f list fields to cut (not universally supported)
list can be specified as: 1,5 (one and five);
or 1,10-12 (1 and 10 to 12); etc
-d field separator
Can accept input from STDIN instead of files

grep [options] regex [file(s)]
Selected options:
-c print the count of matching lines
-i ignore uppercase/lowercase differences
-h don't show file names

```
-l            only give file names when match is found
-n        display line numbers
-v        print lines except those that match

sort -k pos1[,pos2][type][-cmu] -o output [options] [file(s)]
Selected options:
-k   specify start & end fields for sort & sort type
-c        Test if is already sorted
-m   Merge two sorted files
-u        Delete all duplicate lines (unique sort)
-o        Output file name
```

Cut's -d option refers to the common columns delimiter and the -f option refers to a column number:
$ cut -d : -f 1,6 /etc/passwd
cat /etc/passwd | cut -d : -f 1 | nl or cut -d : -f 1 /etc/passwd | nl and cut -d : -f 1 /etc/passwd | nl > userlist

Arithmetic expansion $((expression))

```
#!/bin/bash

#simple if example
#Paul Sutcliffe

echo  "expecting a single argument "
echo  "command argument \$1 is $1  "

if [ $1 -eq 1 ]
then
        echo  "if test is true found 1 that is one "
else
        echo "if test is false did not find 1 "
fi

echo "will now exit"
```

Simple if else script

String tests
== tests for string equal to
!= tests for string not equal to

Other tests
if [-n $astring];then tests for string uninitialized.
if [-f $filename];then tests for file existence.

```
echo  "expecting a single argument "
echo  "command argument \$1 is $1  "

if [ $1 -eq 1 ]
then
        echo  "if test is true found 1 that is one "
else
        echo "if test is false did not find 1 "
fi
echo "will now exit"
```

inode recursive - ls -Ri or ls -i ~/*

simple calculator

```
$ cat calc.sh
#!/bin/bash
PS3="YOUR SELECTION: "
echo PLEASE SELECT ARITHMETIC SIGN:
select sign in "+" "-" "*" "/"
do
  echo Let\'s see....? The answer is: $(($1 $sign $2))
  break
done
$ ./calc.sh 4 5
```

```
#!/bin/bash

if [ -d $1 ]
then
        echo "$1 is a directory"
elif [ -f $1 ]

then
        echo "$1 is a file"

else
        echo "$1 does not exist!"
        echo -n "Create(d/f):  "
        read choice
        if [ "$choice" == "d" ]
        then
                mkdir $1
        else
                touch $1
```

```
        fi
fi

student grade –

echo -n "Student name: "
read name

if [ $1 -ge 85 ]
then
        grade="HD"
elif (( $1 >= 75))
then
        grade ="D"

elif [ $1 -ge 50 ]
then
        grade="C"

elif [ $1 -ge 50 ]
then
        grade="P"
else
        grade="Z"
fi

echo "$name mark is $1 and grade is $grade"
```

GREP

Exercise 2. Look at wild-cards. Print every line that the file 'whois.sh' with the symbol '@', followed by the string 'whois' followed by a '.', then four other characters, followed by another a '.', followed by one or more characters, followed by ';;', followed by the end of the line.

```
grep -E '@whois\..{4}\..+;;$' whois.sh
$ grep -E '@whois\..{4}\..+;;$' whois.sh
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
$
```

Notes

The dot character, '.', means any single character to grep and egrep. When we want it to mean a literal '.' we use '\.' (without the quotes).

{n} means the preceding pattern is matched n times.

+ means the preceding pattern is matched one or more times.

$ means the end of line.

The @ and ; characters have no special meaning to grep so we can simply write then as literals.

```
grep '\$' towers.sh | grep '#'
```

-i grep -i ^DA demo.txt Forgets about case sensitivity
-w grep -w "of" demo.txt Search only for the full word
-A grep -A 3 'Exception' error.log Display 3 lines after matching string
-B grep -B 4 'Exception' error.log Display 4 lines before matching string
-C grep -C 5 'Exception' error.log Display 5 lines around matching string
-r grep -r 'quickref.me' /var/log/nginx/ Recursive search (within subdirs)
-v grep -v 'warning' /var/log/syslog Return all lines which don't match the pattern
-e grep -e '^al' filename Use regex (lines starting with 'al')
-E grep -E 'ja(s|cks)on' filename Extended regex (lines containing jason or jackson)
-c grep -c 'error' /var/log/syslog Count the number of matches
-l grep -l 'robot' /var/log/* Print the name of the file(s) of matches
-o grep -o search_string filename Only show the matching part of the string
-n grep -n "go" demo.txt Show the line numbers of the matches

Options:
-i : Ignore case (case-insensitive search)
-w : Match whole word
-n : Show line numbers
-c : Count the number of matching lines
-v : Invert match, show lines that do not match
-r : Search files recursively in subdirectories
-l : Show only the filenames of matching files
-h : Do not show filenames in output
-e pattern : Use pattern as the search pattern
-f file : Read the search pattern from a file
-E : Interpret the pattern as an extended regular expression
-P : Interpret the pattern as a Perl-compatible regular expression
-m num : Stop after finding num matches
-A num : Show num lines of trailing context after the match
-B num : Show num lines of leading context before the match
-C num : Show num lines of context before and after the match

Regular Expressions:

. : Match any single character **(period)**
^ : Match the beginning of a line
$: Match the end of a line
[] : Match any character inside the brackets
[^] : Match any character NOT inside the brackets
() : Group characters together
| : Match either/or (e.g. cat|dog)
* : Match zero or more of the preceding character
+ : Match one or more of the preceding character
? : Match zero or one of the preceding character
{} : Match a range of occurrences (e.g. a{1,3} matches "a", "aa", or "aaa")

How many times?

? The preceding item must be there 0 times or 1 time
* 0 or more times
regex .* matches 0 or as many characters
+ 1 or more times
regex .+ matches 1 or as many characters
{n} n times
{n,m} at least n times, but not more than m times
A bracket expression is a list of characters enclosed between [and] (“square brackets”). It matches any **single character in that list**

Anchors force the match to occur in certain positions of the input line
the caret ^ and the dollar sign $ force the match to take place at the beginning and end of a line (or just before the newline at the end of the line)
Moreover, \b forces the match to take place at the edge of a word (only works in some versions!)

grep -E '^[^a-z]' file1
Bigger
\bad
grep -E 'g$' file1
big
bad bug
grep -E '^[^a-z]' file1
Bigger
\bad
Email Match grep -E '^[^@]+@[^@]+\.[^@]+$' file1

Examples:
grep "hello" file.txt : Search for the string "hello" in file.txt
grep -i "hello" file.txt : Search for the string "hello" in file.txt (case-insensitive)
grep -w "hello" file.txt : Search for the whole word "hello" in file.txt
grep -n "hello" file.txt : Search for the string "hello" in file.txt and show line numbers
grep -c "hello" file.txt : Count the number of lines in file.txt that contain the string "hello"
grep -v "hello" file.txt : Search for lines in file.txt that do not contain the string "hello"
grep -r "hello" /path/to/dir : Search for the string "hello" recursively in the directory /path/to/dir
grep -e "hello|world" file.txt : Search for the string "hello" or "world" in file.txt
grep -f pattern.txt file.txt : Search for the patterns in pattern.txt in file.txt

grep -E "[0-9]{3}" file.txt : Search for any three consecutive digits in file.txt
grep -P "\d{3}" file.txt : Search for any three consecutive digits in file.txt (Perl-compatible regex syntax)
`grep “hello” file.txt | head -n 5

1. **`grep 'tutor' demo.txt`**: This command searches for the string ‘tutor’ in the file `demo.txt`. It is case-sensitive.

2. **`grep -i 'tutor' demo.txt`**: The `-i` option makes the search case-insensitive. So this command will match ‘Tutor’, ‘TUTOR’, ‘tutor’, etc.

3. **`grep -wi 'tutor' demo.txt`**: The `-w` option makes the search match only whole words. So ‘tutor’ will be matched, but ‘tutorials’ will not. The `-i` option makes the search case-insensitive.

4. **`grep -win 'tutor' demo.txt`**: The `-n` option includes the line number with the matching lines.

5. **`grep -i '\\btutor\\b' demo.txt`**: The `\\b` is a word boundary, so this matches ‘tutor’ as a whole word, but not when it’s part of another word. The `-i` option makes the search case-insensitive.

6. **`grep -in '\\btutor\\b' demo.txt`**: This is similar to the previous command, but includes the line number with the matching lines.

7. **`grep -in -B 1 '\\btutor\\b' demo.txt`**: The `-B 1` option includes one line before the matching line.

8. **`grep -in -A 3 '\\btutor\\b' demo.txt`**: The `-A 3` option includes three lines after the matching line.

9. **`grep -in -C 1 '\\btutor\\b' demo.txt`**: The `-C 1` option includes one line before and one line after the matching line.

10. **`grep '[0-9]' demo.txt`**: This command matches any line that contains a digit.

11. **`grep -n '[0-9]' demo.txt`**: This is similar to the previous command, but includes the line number with the matching lines.

12. **`grep '[0-9]{3}' demo.txt`**: This command matches any line that contains exactly three digits in a row.

13. **`grep -E '[0-9]{3}' demo.txt`**: The `-E` option enables extended regular expressions, which allows the use of `{}` for specifying the number of occurrences.

14. **`grep -E '\\b[0-9]{3}\\b' demo.txt`**: This command matches any line that contains exactly three digits as a whole word.

15. **`egrep '[0-9]{3}' demo.txt`**: The `egrep` command is equivalent to `grep -E`.

16. **`grep -E '[0-9]{3,4}' demo.txt`**: This command matches any line that contains a sequence of three or four digits.

17. `grep '[A-Z]' demo.txt`: This command matches any line that contains an uppercase letter.

18. `grep '[A-Z#,:]' demo.txt`: This command matches any line that contains an uppercase letter, a hash (`#`), a colon (`:`), or a comma (`,`).

19. `grep '[#:$]' demo.txt`: This command matches any line that ends with a hash (`#`), a colon (`:`), or a dollar sign (`$`).

20. `grep '#|:|$' demo.txt`: This command matches any line that contains a hash (`#`), a colon (`:`), or a dollar sign (`$`).

21. `grep -E '#|:|$' demo.txt`: This command is equivalent to the previous command, but uses extended regular expressions.

22. `grep -E '#|:|\\$' demo.txt`: This command matches any line that contains a hash (`#`), a colon (`:`), or a literal dollar sign (`$`). The `\\$` is necessary to escape the dollar sign, which is a special character in regular expressions.

23. `grep '$' demo.txt`: This command matches any line that ends with any character, because `$` is a special character in regular expressions that matches the end of a line.

24. `grep '\\$' demo.txt`: This command matches any line that contains a literal dollar sign (`$`).

25. `grep '^' demo.txt`: This command matches any line, because `^` is a special character in regular expressions that matches the start of a line.

26. `grep '\\^' demo.txt`: This command matches any line that contains a literal caret (`^`).

27. `grep '^$' demo.txt`: This command matches any empty line, because `^$` is a regular expression that matches the start of a line immediately followed by the end of the line.

28. `grep -n '^$' demo.txt`: This is similar to the previous command, but includes the line number with the matching lines.

    - 29. `grep -v '^$' demo.txt`: The `-v` option inverts the match, so this command matches any non-empty line.
    - grep '^$' demo.txt # Search for empty lines in demo.txt.
    - grep -n '^$' demo.txt # Search for empty lines with line numbers displayed in demo.txt.
    - grep -n '^*$' demo.txt # Attempt to match lines containing only asterisks (invalid pattern).
    - grep -nE '^*$' demo.txt # Attempt to match lines containing only asterisks with extended regex (invalid pattern).
    - grep -v '^$' demo.txt # Invert match to display lines that are not empty in demo.txt.
    - grep '[#:]' demo.txt # Search for lines containing either '#' or ':' in demo.txt.
    - grep '#|:' demo.txt # Attempt to match lines containing '#|:' literally (invalid pattern).
    - grep -E '#|:' demo.txt # Search for lines containing either '#' or ':' with extended regex in demo.txt.

30. `grep -e '#' -e ':' demo.txt`: The `-e` option allows multiple patterns. This command matches any line that contains a hash (`#`) or a colon (`:`).

31. **`grep -R 'bash' ~/GitHub/USP_Tutorials/`**: The `-R` option makes the search recursive. This command searches for 'bash' in all files in the `~/GitHub/USP_Tutorials/` directory and its subdirectories.

32. **`grep -Rn 'bash' ~/GitHub/USP_Tutorials/`**: This is similar to the previous command, but includes the line number with the matching lines.

33. **`grep -Rl 'bash' ~/GitHub/USP_Tutorials/`**: This is similar to the previous command, but only prints the names of files that contain the match.

1. **`sort demo.txt`**: This command sorts the lines in the file `demo.txt` in lexicographic order.

2. **`clear`**: This command clears the terminal screen.

3. **`sort animals.txt`**: This command sorts the lines in the file `animals.txt` in lexicographic order.

4. **`sort -u animals.txt`**: This command sorts the lines in the file `animals.txt` in lexicographic order and removes duplicate lines.

5. **`sort -ui animals.txt`**: This command sorts the lines in the file `animals.txt` in lexicographic order, ignores case when comparing lines, and removes duplicate lines.

6. **`sort -ur animals.txt`**: This command sorts the lines in the file `animals.txt` in reverse lexicographic order and removes duplicate lines.

7. **`sed 's/tutor/lecturer/' demo.txt`**: This command replaces the first occurrence of 'tutor' with 'lecturer' in each line of the file `demo.txt`.

8. **`sed 's/tutor/lecturer/i' demo.txt`**: This command replaces the first occurrence of 'tutor' with 'lecturer' in each line of the file `demo.txt`, ignoring case.

9. **`sed 's/\\btutor\\b/lecturer/i' demo.txt`**: This command replaces the word 'tutor' with 'lecturer' in each line of the file `demo.txt`, ignoring case. The `\\b` word boundary ensures that only the word 'tutor' is replaced, not 'tutor' when it appears as part of another word.

10. **`sed 's/[0-9]/-/' demo.txt`**: This command replaces the first digit in each line of the file `demo.txt` with a dash.

11. **`sed 's/[0-9]/-/g' demo.txt`**: This command replaces all digits in the file `demo.txt` with a dash.

12. **`sed 's/[0-9a-z]/-/g' demo.txt`**: This command replaces all digits and lowercase letters in the file `demo.txt` with a dash.

13. **`sed '5,$d' demo.txt`**: This command deletes all lines from line 5 to the end of the file `demo.txt`.

14. **`sed '7,15d' demo.txt`**: This command deletes lines 7 to 15 in the file `demo.txt`.

15. **`sed '3,9d' demo.txt`**: This command deletes lines 3 to 9 in the file `demo.txt`.

16. **`sed '5i\\Inserted Line' demo.txt`**: This command inserts the line 'Inserted Line' before line 5 in the file `demo.txt`.

17. **`sed '2,4c\\Replaced Line' demo.txt`**: This command replaces lines 2 to 4 in the file `demo.txt` with the line ‘Replaced Line’.

18. **`sed -n '/^$/ p' demo.txt`**: This command prints only the empty lines in the file `demo.txt`.

19. **`sed -n '/^$/ !p' demo.txt`**: This command prints only the non-empty lines in the file `demo.txt`.

1. `grep 'tutor' demo.txt`: Searches for lines containing the string "tutor" in the file `demo.txt`.

2. `grep -i 'tutor' demo.txt`: Searches for lines containing the string "tutor" case-insensitively in the file `demo.txt`.

3. `grep -wi 'tutor' demo.txt`: Searches for whole words containing the string "tutor" case-insensitively in the file `demo.txt`.
4. `grep -win 'tutor' demo.txt`: Searches for whole words containing the string "tutor" case-insensitively in the file `demo.txt`, and also shows line numbers where matches are found.

5. `grep -i '\btutor\b' demo.txt`: Searches for whole words "tutor" case-insensitively in the file `demo.txt`.

6. `grep -in '\btutor\b' demo.txt`: Searches for whole words "tutor" case-insensitively in the file `demo.txt`, and also shows line numbers where matches are found.

7. `grep '[0-9]' demo.txt`: Searches for lines containing any digit in the file `demo.txt`.

8. `grep -n '[0-9]' demo.txt`: Searches for lines containing any digit in the file `demo.txt` and shows line numbers where matches are found.

9. `grep '[0-9]{3}' demo.txt`: Searches for lines containing sequences of exactly three digits in the file `demo.txt`. This command won't work as intended because `{}` is not a valid quantifier in basic regular expressions (BRE).

10. `grep -E '[0-9]{3}' demo.txt`: Searches for lines containing sequences of exactly three digits in the file `demo.txt`, using extended regular expressions.

11. `grep -E '\b[0-9]{3}\b' demo.txt`: Searches for whole words containing exactly three digits in the file `demo.txt`, using extended regular expressions.

12. `grep -E '[0-9]{3}' demo.txt`: Same as command in line 10, searches for lines containing sequences of exactly three digits in the file `demo.txt`, using extended regular expressions.

13. `egrep '[0-9]{3}' demo.txt`: Same as command in line 10, searches for lines containing sequences of exactly three digits in the file `demo.txt`, using extended regular expressions.

14. `grep -E '[0-9]{3,4}' demo.txt`: Searches for lines containing sequences of three or four digits in the file `demo.txt`, using extended regular expressions.

15. `grep -E '[0-9]{2,4}' demo.txt`: Searches for lines containing sequences of two, three, or four digits in the file `demo.txt`, using extended regular expressions.