NEW HORIZON COLLEGE OF ENGINEERING

Autonomous College Permanently Affiliated to VTU, Approved by AICTE & UGC
Accredited by NAAC with 'A' Grade, Accredited by NBA

DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS

SEMESTER-I

LINUX OPERATING SYSTEM AND SHELL SCRIPTING

(24MCA15)

COURSE COORDINATOR
Dr. Arpana Prasad

2025 – 26

SAMPLE CERTIFICATE

NEW HORIZON
COLLEGE OF ENGINEERING

Autonomous College Permanently Affiliated to VTU, Approved by AICTE & UGC
Accredited by NAAC with 'A' Grade, Accredited by NBA

DEPARTMENT OF MCA

CERTIFICATE

This is to certify that <<NAME>> bearing USN No. <<USN NO >> of **MCA I SEMESTER** has satisfactorily completed the Laboratory Experiments in **LINUX OPERATING SYSTEM AND SHELL SCRIPTING**, Subject Code – **24MCA15** as

prescribed in the Syllabus **2025-26**.

---------- ---------- **Signature of**

Course Coordinator Signature of Head of the Department

---------- ---------- **Internal**

Examiner External Examiner

Date of Examination:

INDEX

6	Write a shell script that accepts valid login names as arguments and prints their corresponding home directories. If no arguments are specified, print a suitable error message.		
7	Write a shell script that displays all the links to a file specified as the first argument to the script. The second argument, which is optional, can be used to specify the directory in which the search is to begin. If this second argument is not provided, the search will begin in the current working directory. In either case, the starting directory and all its subdirectories at all levels must be searched. The script does not need to include any error checking.		
8	Write a shell script that accepts two file names as arguments. It checks if the permissions for these files are identical. If the permissions are identical, output the common permissions; otherwise, output each file name followed by its permissions.		

9	Write a shell script that takes a valid directory name as an argument, recursively descends into all the sub directories, finds the maximum length of any file in that hierarchy, and writes this maximum value to the standard output.		
10	Write a shell script that accepts a list of filenames as its arguments, counts the occurrences of each word present in the first argument file in the other argument files, and reports the counts.		
11	Write a shell script to display the calendar for the current month with the current date replaced by * or ** depending on whether the date has one digit or two digits.		
12	Write a shell script that accepts a file name, starting line number, and ending line number as arguments and displays all the lines between the given line numbers.		
13	Write a shell script that folds long lines into 40 columns. Any line that exceeds 40 characters must be broken after the 40th character, with a \ appended to indicate the line is folded, and processing should continue with the remaining text. The input should be supplied through a text file created by the user.		
14	Write an awk script that accepts a date argument in the form of dd-mm-yy and displays it in the form of month, day, and year. The script should check the validity of the argument and, in case of an error, display a suitable message.		

15	Write an awk script to delete duplicate lines from a text file while keeping the order of the original lines unchanged.		
16	Write an awk script to find the total number of books sold in each discipline as well as the total number of books sold, using an associative array. For example: Electrical 34, Mechanical 67, Electrical 80, Computer Science 43, Mechanical 65, Civil 98, Computer Science 64		

EXPERIMENT NO 1

AIM OF THE EXPERIMENT: Execute basic Linux commands such as pwd, cd, ls, mkdir, rmdir, cp, mv, rm, cat, and observe their effects on the file system.

THEORY: The basic Linux commands help navigate and manage the file system efficiently. pwd shows the current directory, while cd changes it. ls lists files, and mkdir creates directories, which can be removed using rmdir if empty. cp copies files or directories, and mv moves or renames them. rm deletes files or directories (use -r for recursive deletion), and cat displays file contents or copies them to another file. Practice these commands in a terminal to understand their effects on the file system.

This script can be saved to a file (e.g., basic.sh), made executable using chmod +x basic.sh, and executed by running ./basic.sh in the terminal.

COMMANDS USED: pwd, cd, ls, mkdir, rmdir, cp, mv, rm, cat

SAMPLE SHELL SCRIPT 1

```
#!/bin/bash
echo "Current Directory:"
pwd
mkdir demo_directory
echo "Created 'demo_directory'."
cd demo_directory
echo "Changed into 'demo_directory'. Current
Directory:" pwd
echo "This is a sample file." > sample.txt
echo "Created 'sample.txt' with content."
ls
cp sample.txt copy_sample.txt
echo "Copied 'sample.txt' to 'copy_sample.txt'."
ls
```

```
mv copy_sample.txt renamed_sample.txt
echo "Renamed 'copy_sample.txt' to
'renamed_sample.txt'." ls
echo "Content of 'renamed_sample.txt':"
cat renamed_sample.txt
cd ..
echo "Moved back to parent directory. Current
Directory:" pwd
rm -r demo_directory
echo "Removed 'demo_directory' and its contents."
```


SAMPLE SHELL SCRIPT 2

```
#!/bin/bash
mkdir my_dir && cd my_dir
echo "Hello, Linux!" > file.txt
cat file.txt
cd .. && rm -r my_dir
```

OUTPUT:

FURTHER EXPLORATION

1. Which command in Linux may be used to check the current date?
2. Check the help for date command using the 'man' command.
3. Further enhance the shell script to make more elaborate welcome message of your choice.
4. Keep note of the syntactical errors and semantical errors encountered; the student may try by varying the code and explore by writing different code for same semantic

task.

EXPERIMENT NO2

AIM OF THE EXPERIMENT: To create a shell script that when executed displays the message either "Good Morning" or "Good Afternoon" or "Good Evening" depending upon time at which the user logs in.

THEORY: In this script, the date command is used to get the current time in 24-hour format, and the %H format specifier is used to extract the hour. The hour is stored in the current_time variable. Next, the script uses an if-elif-else statement to check the value of current_time and display the appropriate greeting message. If the hour is less than 12, the script displays "Good Morning." If the hour is greater than or equal to 12 and less than 18, the script displays "Good Afternoon." Otherwise, it displays "Good Evening."
This script can be saved to a file (e.g., greet.sh), made executable using chmod +x greet.sh, and executed by running ./greet.sh in the terminal.

COMMANDS USED: date

The date command in Linux provides several options to display and set the date and time in different formats. Here are some of the commonly used options:
date: This command with no options will display the current date and time in the default format.
date +"%Y-%m-%d %H:%M:%S": This option sets the date and time format using format specifiers. In this example, the format is set to display the year, month, day, hour, minute, and seconds in the format of YYYY-MM-DD HH:MM:SS.
date -s "2023-02-15 12:30:00": This option sets the system date and time to a specific date and time. In this example, the date and time is set to February 15th, 2023 at 12:30 PM.

The %H is a format specifier used with the date command to display the current hour in 24-hour format.
To use the %H format specifier with the date command, open a terminal and type the following **Command**:
date '+%H'
Other usage of date command is as follows:

SAMPLE SHELL SCRIPT 1

```
#!/bin/bash
current_time=$(date +%H)
if [ $current_time -lt 12 ]; then
 echo "Good Morning"
elif [ $current_time -lt 18 ]; then
 echo "Good Afternoon"
else
 echo "Good Evening"
```


SAMPLE SHELL SCRIPT 2

```
echo -n "System Date : "
date
h=`date | cut -c12-13`
if [ $h -ge 0 -a $h -lt 12 ]
then
 echo "Good Morning !"
elif [ $h -ge 12 -a $h -lt 18 ]
then
 echo "Good Afternoon !"
elif [ $h -ge 18 -a $h -lt 20 ]
then
 echo "Good Evening !"
else
 echo "Good Night !"
fi
```

OUTPUT:

FURTHER EXPLORATION

1. Which command in Linux may be used to check the current date?
2. Check the help for date command using the 'man' command.
3. Further enhance the shell script to make more elaborate welcome message of your choice.
4. Keep note of the syntactical errors and semantical errors encountered; the student may try by varying the code and explore by writing different code for same semantic task.

EXPERIMENT NO 3

AIM OF THE EXPERIMENT: Create a shell script that accepts a path name and creates all the components in that path name as directories. For example, if the script is named mpc, then the command mpc a/b/c/d should create directories a, a/b, a/b/c, a/b/c/d.

THEORY: This shell script is designed to create a directory at a given path. Here's a breakdown of how it works:
The script first checks if exactly one argument is passed to it. If not, it prints an error message along with a usage statement and exits with a status code of 1.
The script then assigns the value of the first argument to a variable named path. The mkdir command is used with the -p option to create the directory specified by path. The -p option creates parent directories if they do not exist.
The script then checks the exit status of the mkdir command. If it was unsuccessful, an error message is printed, and the script exits with a status code of 1.
Finally, if the mkdir command was successful, the script prints a message indicating that the directories were created successfully in the given path.

Here's a step-by-step explanation of the code:

```
if [ $# -ne 1 ]; then
 echo "Usage: Please execute the shell script like sh $0 <path>"
exit 1
fi
```

This if statement checks if exactly one argument is passed to the script. If the number of arguments passed is not equal to 1, the script prints a usage message with the name of the script using $0, and exits with a status code of 1.

```
path=$1
```

This line assigns the first argument to a variable named path.

```
mkdir -p $path
```

This mkdir command creates the directory specified by the path variable. The -p option creates the parent directories if they do not exist.

```
if [ $? -ne 0 ]; then
 echo "Error creating directory $path"
 exit 1
```

```
fi
```
This if statement checks the exit status of the mkdir command. If the exit status is not zero, it means that the mkdir command was unsuccessful. In this case, an error message is printed, and the script exits with a status code of 1.

```
echo "Directories created successfully in $path"
```
If the mkdir command was successful, this line prints a message indicating that the directory was created successfully in the given path.

SAMPLE SHELL SCRIPT 1:

```
if test $# -ne 1
 then
 echo “Usage: Please execute the shell script like sh $0 <path>”
exit 1
fi
path=$1
mkdir -p $path
if test $? -ne 0
then
 echo “Error creating directory $path”
 exit 1
fi
echo “Directories created successfully in $path”
```

SAMPLE SHELL SCRIPT 2:

```
echo "Enter the Path Name For Creation of Directories a or a/b or
a/b/c or a/b/c/d "
read p
d=$p
p=$p"/"
while [ true ]
do
x=`echo "$p" | cut -d "/" -f1`
l=$x
if [ "$x" = "" ];
then
      echo "Created Directory Structure $d Successfully
      !" exit
fi
if [ -d $x ]
then
      echo "$x Directory is already present - Not possible to create
 !"
```

```
exit
fi
mkdir -p $x
echo "Created Sub Directory $x Successfully !"
cd $x
echo "Inside Sub Directory $x !"
p=`echo "$p" | cut -d "/" -f2-`
done
```


OUTPUT:

EXPERIMENT NO. 4

AIM OF THE EXPERIMENT: Create a shell script to implement a terminal locking mechanism similar to the lock command. The script should prompt the user to enter a password, then prompt again to confirm the password. If the passwords match, the script should lock the terminal, requiring the correct password to unlock it. Use Linux file management commands to achieve this functionality.

THEORY:

A. COMMAND USED: stty

The "stty" command is a Unix and Linux command used to view or modify terminal settings. It can be used to control various terminal features, such as echoing, flow control, and character processing. The command can be used with a variety of options to set or display various terminal attributes. For example, "stty -a" displays all the current settings for the terminal, while "stty -echo" turns off

character echoing.

The basic syntax of the "stty" command is as follows:
stty [options]

Here, "options" are the various command line options that can be used with "stty" to modify or display terminal settings, and "device" specifies the name of the terminal device to be modified (typically /dev/tty).
Some common options for the "stty" command are:
"-a": Displays all current settings for the terminal
"-echo": Turns off character echoing
Please note: The students will write the syntax, description, options and examples of sty command.

B. In Linux, a selection statement is a programming construct that allows you to execute different code paths based on a specified condition. The most commonly used selection statement is the "if" statement, which executes a block of code if a specified condition is true. Other selection statements include the "else" statement, which executes a block of code if the preceding "if" statement condition is false, and the "switch" statement, which evaluates a variable and executes different code blocks based on its value.
The syntax of the if statement in Linux shell scripting is:

```
if [ condition ]
then
 # code to execute if the condition is true
fi
```

In this syntax, the condition is placed within square brackets and the keyword then is used to specify the code block to execute if the condition is true. The code block is terminated with the fi keyword.
Alternatively, you can use the one-line syntax, which looks like this:

```
if [ condition ]; then
 # code to execute if the condition is true
fi
```

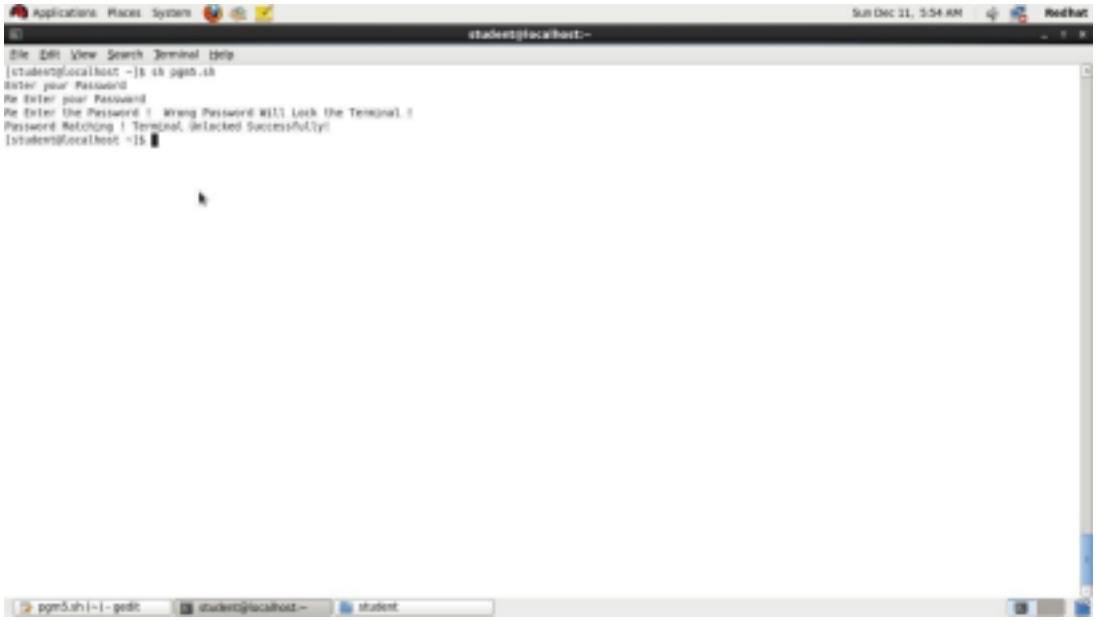
In this syntax, the then keyword is placed on the same line as the closing square bracket.

SAMPLE SHELL SCRIPT

```
echo "Enter your Password"
stty -echo
read p1
stty echo
echo "Re Enter your Password"
stty -echo
read p2
stty echo
if [ "$p1" = "$p2" ]
then
 echo "Re Enter the Password ! Wrong Password Will Lock the Terminal !"
stty -echo
 read p3
```

```
 while [ "$p2" ≠ "$p3" ]
 do
 stty echo
 echo "Terminal is Locked ! Enter the Right Password to Unlock !"
stty -echo
 read p3
 done
stty echo
echo "Password Matching ! Terminal Unlocked Successfully!"
else
echo "Password Mismatch !"
fi
```

OUTPUT:



Further Exploration:

1. Write the shell script using switch case.

EXPERIMENT NO 5

AIM OF THE EXPERIMENT: Create a script file called file-properties that reads a file name entered and outputs it properties.

THEORY:

Command Used – cut

The cut command is a command-line utility in Linux and other Unix-like operating systems that is used to extract sections or columns from a text file or input stream. It can be used to extract portions of text based on character position, byte offset, or a specific delimiter.

The basic syntax of the cut command is as follows:

cut [OPTIONS] [FILE]

Here are some common options used with the cut command:
-c: Select characters based on character position
-d: Use a specific delimiter
-f: Select fields based on a specific delimiter
--complement: Invert the selection to output everything except the selected characters or

fields Here are some examples of using the cut command:

Extract the first three characters from a file:
```
cut -c 1-3 file.txt
```
Extract the first and third character from a file:
```
cut -c 1,3 file.txt
```
Extract a specific field from a file using a delimiter:
```
cut -d ',' -f 2 file.csv
```
Extract a range of fields from a file using a delimiter:
```
cut -d ',' -f 2-4 file.csv
```

Invert the selection to output everything except a specific field:

```
cut -d ',' --complement -f 2 file.csv
```

These are just a few examples of how to use the cut command. For more information, you can refer to the cut manual page by running the command man cut in the terminal.

SAMPLE SHELL SCRIPT

```
echo "File Properties"
fname=$@
if [ $# -eq 0 ]
then
 echo " Please Pass File Names as Arguments"
exit
fi
n=$#
```

```
t=1
clear
echo "Properties of the Files :
$fname" while [ $t –le $n ]
do
 echo "File Name : $1"
 ls –l $1
 s=`ls –l $1 | tr –s " " | cut –d " " –f5`
echo "Size of $1 is : $s"
 if [ –r "$1" ]
 then
 echo "$1 is Readable."
 fi
 if [ –w "$1" ]
 then
 echo "$1 is Writable."
 fi
 if [ –x "$1" ]
 then
 echo "$1 is Executable."
 fi
 t=`expr $t + 1`
 echo "Further Information on $1"
 stat $1
 echo “Press any key to continue”
 read
 shift
done
```


OUTPUT:

EXPERIMENT NO 6

AIM OF THE EXPERIMENT: Create a shell script in Linux that accepts a valid login-id as arguments and prints the corresponding home directories, if no argument is provided display an error message. In case the argument provided by the user has invalid login id then display suitable error message.

THEORY:

A. **COMMAND: grep**
The grep command is a popular command-line tool in Linux and Unix systems used to search for text patterns in a file or a group of files. It is short for "global regular expression print."

The basic syntax of the grep command is:
grep [OPTIONS] PATTERN [FILE...]
where:

OPTIONS are any additional flags or options for the command.
PATTERN is the regular expression or string that you want to search for.
FILE... is the name of the file or files in which you want to search for the pattern. You can specify multiple files, or use wildcard characters to match multiple files.
Here are some commonly used options for the grep command:

-i: ignores case when matching the pattern.
-n: displays the line numbers of matching lines.
-v: displays all the lines that do not match the pattern.
-c: displays the count of matching lines instead of the actual lines.
-r: searches for the pattern recursively in all files and directories under a given directory.

Some examples of using the grep command:

Search for a pattern in a file:

```
grep "pattern" filename.txt
```

Search for a pattern in multiple files:
```
grep "pattern" file1.txt file2.txt file3.txt
```

Search for a pattern in all files under a directory:
```
grep -r "pattern" /path/to/directory/
```

Search for a pattern in a file and display the line numbers:
```
grep -n "pattern" filename.txt
```

Search for a pattern in a file, ignoring case:
```
grep -i "pattern" filename.txt
```

B. **/etc/passwd**

In Linux and Unix-like operating systems, /etc/passwd is a plain text file that stores information about user accounts on the system. It contains one entry per line, and each entry consists of several fields separated by colons.

The general format of a passwd file entry is:
username:password:UID:GID:GECOS:home_directory:shell
where:

username: the name of the user account.
password: the user's encrypted password, or an "x" if the password is stored in the /etc/shadow file. UID: the user's numeric user ID.
GID: the user's numeric group ID.
GECOS: a comma-separated list of additional user information, such as the user's full name, office phone number, etc.
home_directory: the path to the user's home directory.
shell: the user's default login shell.
Here's an example of a passwd file entry:
john:x:1000:1000:John Smith:/home/john:/bin/bash

In this example, the user account name is john, the password is stored in the /etc/shadow file, the UID and GID are both 1000, the user's full name is John Smith, the home directory is /home/john, and the default login shell is /bin/bash.

The passwd file can be viewed and edited by the system administrator using a text editor or command-line utilities like vi.

C. **Command:** getent

The getent command in Linux is used to retrieve information from the system databases, such as /etc/passwd, /etc/group, and /etc/hosts. It can be used to query information about users, groups, hostnames, and other system entities.
The basic syntax of the getent command is as follows:

getent database [key ...]

Here, database specifies the name of the system database to query, and key specifies the key value to search for within that database. If no key value is specified, getent will return all entries in the database.
For example, to retrieve information about a specific user from the /etc/passwd file, you can use the following command:

```
getent passwd <username>
```

Similarly, to retrieve information about a specific group from the /etc/group file, you can use the following command:

```
getent group <groupname>
```

You can also use getent to retrieve information from other databases, such as /etc/hosts, which contains information about the hostname and IP address mappings on the system. For example, to retrieve the IP address for a specific hostname, you can use the following command:

```
getent hosts <hostname>
```

Overall, getent is a useful command for retrieving information from system databases in Linux, and can be used in a variety of contexts, such as scripting and system administration.

SAMPLE SHELL SCRIPT 1:

```
echo "To Print Home Directories of given Username"
echo "Enter Valid Login Name"
read lognm
hd=`grep $lognm /etc/passwd | cut -d ":" -f6`
l=$hd
if [ $l <> 0 ]
then
 echo "Home Directory of $lognm is $hd"
else
 echo "Login Name $lognm Not Valid !"
fi
```

SAMPLE SHELL SCRIPT 2:

```
#!/bin/bash
echo "Error: No argument provided."
 exit 1
fi
```

```
for username in "$@"; do
 home_dir=$(getent passwd "$username" | cut -d: -f6)
if [ -z "$home_dir" ]; then
 echo "Error: Invalid login ID: $username"
 else
 echo "$username: $home_dir"
 fi
done
if [ $# -eq 0 ]; then
```

OUTPUT:

EXPERIMENT NO 7

AIM OF EXPERIMENT:

Create a shell script that displays all the links to a file specified as the first argument to the script. The second argument, which is optional, can be used to specify in which the search is to begin. If this second argument is not present, the search is to begin in current working directory. In either case, the starting directory as well as all its subdirectories at all levels must be searched. The script need not include any error checking.

THEORY:

`A. link=`ls -l $dir/$file | tr -s " " | cut -d " " -f2`` This is a shell command that performs the following actions:

List the files in the directory specified by the $dir variable and display detailed information about them using the ls -l command.

Use the tr command to replace any consecutive whitespace characters with a single space character. This is done to make it easier to parse the output of the ls command in the next step.

Use the cut command to extract the second field (-f2) from the output of the previous step. This field corresponds to the link count of the file specified by the $file variable.

Overall, this command is used to retrieve the link count of a file in a specified directory. The link count is the number of hard links that point to the file, and it is one of the attributes that can be displayed by the ls -l command.

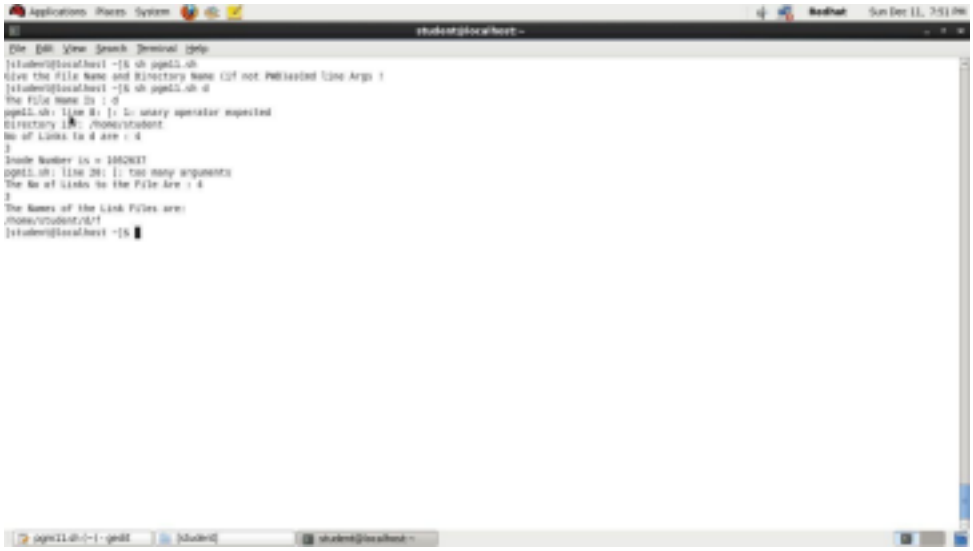
SAMPLE SHELL SCRIPT

```
if test $# -eq 0
then
 echo "Give the File Name and Directory Name (if not PWD)as Cmd line Args
!"
 exit
fi
file=$1
echo "The File Name Is : $file"
if test $# -eq 2
 then
dir=$2
 echo "Directory is : $dir"
else
dir=`pwd`
 echo "Directory is : $dir"
fi
link=`ls -l $dir/$file | tr -s " " | cut -d " " -f2` echo
"No of Links to $file are : $link"
ino=`ls -i $dir/$file | tr -s " " | cut -d " " -f1` echo
"Inode Number is = $ino"
if test $link -eq 1
then
```



```
echo " The File has No Links â€“ Default is one Link for a File !" else
echo "The No of Links to the File Are : $link"
 echo "The Names of the Link Files are: "
 find $dir -inum $ino -print
fi
```

OUTPUT:



EXPERIMENT NO 8

AIM OF THE EXPERIMENT: Create a shell script that accepts two file names as arguments, checks if the permissions for these files are identical and if the permissions are identical, output common permissions and otherwise output each file name followed by its permissions.

THEORY:

This is a shell script that checks if two file names have the same permissions.
Here's a line-by-line explanation:

```
if [ $# -eq 2 ]
then
```

This if statement checks if the number of command line arguments passed to the script is equal to 2. $# is a special variable that holds the number of arguments.

```
if [ -f $1 -a -f $2 ]
```

This if statement checks if both arguments are files using the -f test. The -a operator performs a logical AND operation.

```
p1=`ls -l $1 | cut -c1-10`
p2=`ls -l $2 | cut -c1-10`
```

These lines use ls -l to get the long listing format for each file and cut to extract the first 10 characters, which represent the file permissions. The permissions are then stored in the p1 and p2 variables.

```
if [ "$p1" = "$p2" ]
then
 echo "File Permissions are equal"
 echo "Permissions of First File : $p1"
 echo "Permissions of Second File : $p2"
else
 echo "File Permissions are not equal"
```

```
 ls -l
fi
```
This if statement compares the permissions of the two files by comparing the values of the p1 and p2 variables. If they are equal, it prints a message indicating this and the permissions of each file. Otherwise, it prints a message indicating the permissions are not equal and the ls -l command is used to list the permissions of both files.
```
else
 echo "File does not Exist !"
 exit
fi
If either of the arguments is not a file, this else block is executed,
and an error message is printed, and the script exits.
else
 echo "Pass two file names as command Line Arguments"
 exit
fi
```
If the number of arguments passed is not 2, this else block is executed, and an error message is printed, and the script exits.


```
if test -f $1 -a -f $2
 then
 p1=`ls -l $1 | cut -c1-10`
p2=`ls -l $2 | cut -c1-10`
```

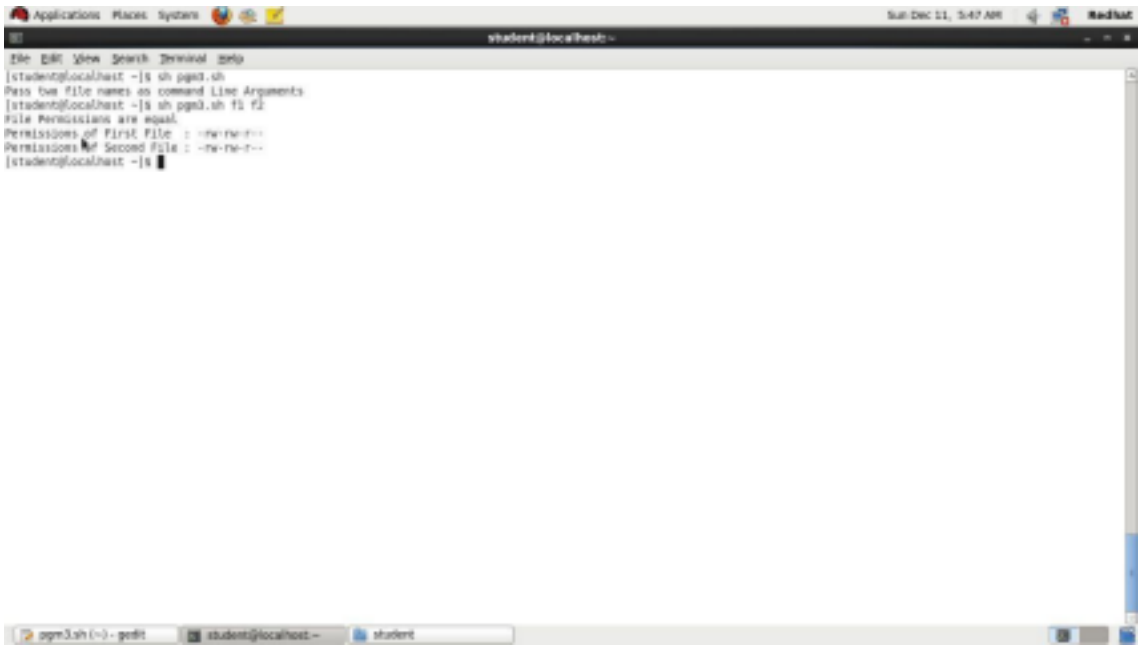

SAMPLE SHELL SCRIPT :

```
if test $# -eq 2
then
 if test "$p1" = "$p2"
 then
 echo "File Permissions are equal"
 echo "Permissions of First File : $p1"  echo "Permissions of
Second File : $p2"

 else
 echo "File Permissions are not equal"  ls -l
 fi
 else
 echo "File does not Exist !"
 exit
 fi
else
echo "Pass two file names as command Line Arguments"  exit
```

```
fi
```

OUTPUT:



EXPERIMENT NO 9

AIM OF THE EXPERIMENT: Create a shell script that takes a valid directory name as an argument and recursively descend all the sub-directories, finds the maximum length of any file in that hierarchy and writes this maximum value to the standard output.

THEORY:

There are two main commands used in the script.
A. `max=`ls -lR $d | sort -k5 -n | tail -1 | tr -s " " | cut -d " " -f5`` B. `nm=`ls -lR $d | sort -k5 -n | tail -1 | tr -s " " | cut -d " " -f9`` Explanation of the components of these commands is as follows:
This command is a shell command that finds the maximum size of a file in a directory and all its subdirectories and name of the file of maximum size in a directory and all its subdirectories.

Here is what each part of the command does:

ls -lR $d: This lists all files and directories in the specified directory and its subdirectories recursively. The -l option displays the files in long format, and the -R option lists the files recursively. sort -k5 -n: This sorts the output of the ls command by the fifth column, which represents the size of each file, in numerical order.
tail -1: This selects the last line of the sorted output, which represents the file with the largest size.
tr -s " ": This translates multiple spaces into a single space. This is useful because the cut command that follows will use spaces as delimiters.
cut -d " " -f5: This cuts the fifth field from each line, which represents the size of each file. cut -d " " -f9: This cuts the ninth field from each line, which represents the name of each file. max=...```: This assigns the result of the previous command (the largest file size) to a variable named max.
min=...```: This assigns the result of the previous command (the name of the largest file(to a variable named min.
In summary, this command lists all files and directories in the specified directory and its subdirectories, sorts them by size, selects the largest file, and outputs its size in bytes. The max=...```

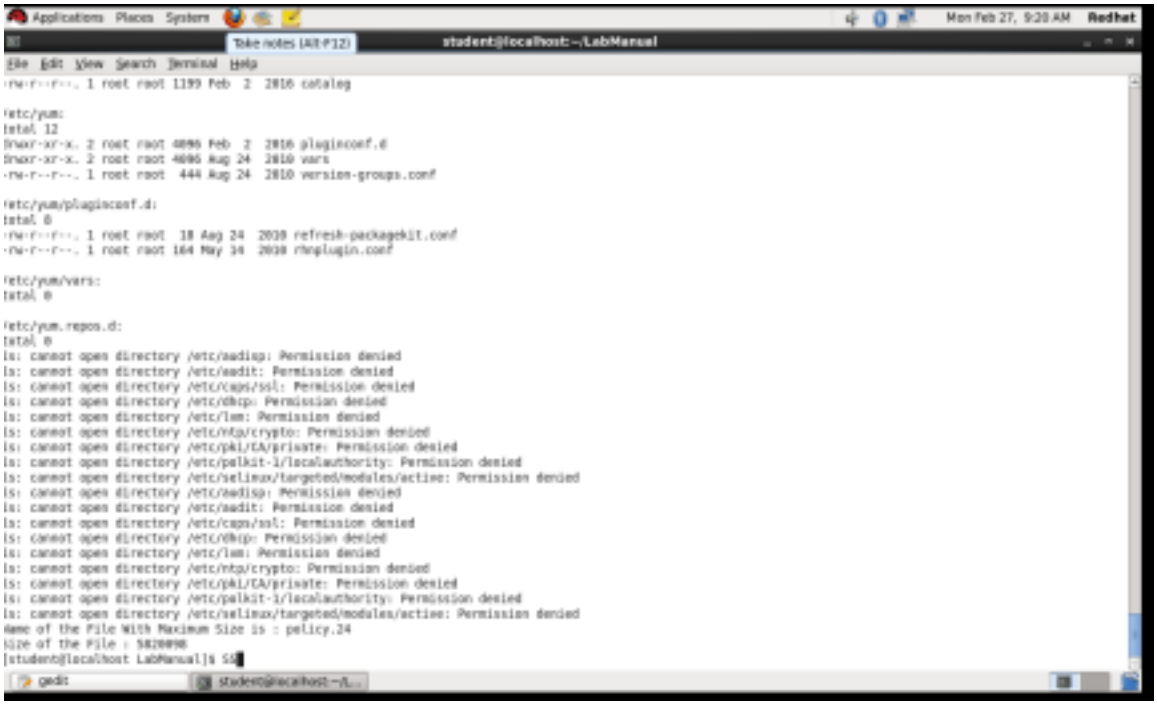
part assigns this size to the max variable for later use in the shell script.
The nm=...``` part assigns this name to the nm variable for later use in the shell script.

SAMPLE SHELL SCRIPT :

```
echo "Enter a Valid Directory Name"
read d
if [ -d $d ]
then
echo "Recursive Display with Subdirectories"
ls -lR $d
max=`ls -lR $d | sort -k5 -n | tail -1 | tr -s " " | cut -d " "
-f5`
nm=`ls -lR $d | sort -k5 -n | tail -1 | tr -s " " | cut -d " "
-f9` echo "Name of the File With Maximum Size is : $nm"
echo "Size of the File : $max"
else
 echo “Not a Valid Directory !”

fi
```


OUTPUT:



EXPERIMENT NO 10

AIM OF THE EXPERIMENT: Create a shell script that accept a list of filenames as its argument, count and report occurrence of each word that is present in the first argument file on other argument files.

THEORY:

The shell script given accepts a list of file names as arguments and counts the number of occurrences of each word in the files.
The script first checks if there are any arguments provided. If there are none, it prints an error message and exits.
The script sets the first argument as the file name to search for words in, and then shifts the argument list to exclude the first file name.
The script then loops through each word in the first file using cat $f, where $f is the first file name. For each word in the file, the script sets a counter cnt to 0 and then loops through each remaining file in the argument list.
For each file, the script uses the grep command to find all occurrences of the word in the file, using the -wo option to match only whole words and the -l option to print only the count of matching lines. The script then adds the count n of occurrences of the word in the current file to the cnt counter. After looping through all files, the script prints the total count of occurrences of the word in all files.

SAMPLE SHELL SCRIPT:

```
if test $# -eq 0
then
 echo "Enter File Names !"
 exit
fi
f=$1
shift
for wrd in `cat $f`
do
cnt=0
 for file in $*
 do
 n=`grep -wo $wrd $file | wc -l`
 cnt=`expr $cnt + $n`
 done
  echo "No of Occurrences of $wrd in All Files is : $cnt"
done
```

OUTPUT:

```
student@localhost LabManual]$ cat > tosearch
ndia
n india
ord is if
student@localhost LabManual]$
student@localhost LabManual]$ cat > insearch1
ndia indiande india jdjdj india if
f if
xit
student@localhost LabManual]$
student@localhost LabManual]$ cat > insearch2
ndia india
student@localhost LabManual]$
student@localhost LabManual]$ sh experiment11 tosearch insearch2 insearch1
o of Occurrences of india in All Files is : 5
o of Occurrences of in in All Files is : 0
o of Occurrences of india in All Files is : 5
o of Occurrences of word in All Files is : 0
o of Occurrences of is in All Files is : 0
o of Occurrences of if in All Files is : 3
student@localhost LabManual]$ SSSS
```

EXPERIMENT NO 11

AIM OF THE EXPERIMENT:

Create a shell script to display the calendar for current month with current date replaced by * or ** depending on whether the date has one digit or two digits.

THEORY:

This is a shell script that displays the current date and replaces the single-digit date with an asterisk (*) and the double-digit date with two asterisks (**).

Let's break down the code:

d=date +%e``: This assigns the current date's day of the month to the variable d. The %e option in the date command gives the day of the month as a decimal number.

echo "Today's Date is: $d": This prints the current date's day of the month.

echo "Current Date will be Replaced by * for Single Digit, ** for Double Digits ": This prints a message to inform the user that the script will replace the single-digit date with an asterisk (*) and the double digit date with two asterisks (**).

if [$d -le 9]: This is an if statement that checks if the day of the month is less than or equal to 9. c=date +%e | cut -c2``: If the day of the month is less than or equal to 9, this assigns the second character of the output of date +%e (i.e., the day of the month without the leading zero) to the variable c.

cal -h | sed " s/\b$c\b/*/ ": This pipes the output of the cal -h command to the sed command, which replaces the single-digit day of the month (represented by the value of c) with an asterisk (*). else: If the day of the month is greater than 9, this else block is executed.

cal -h | sed " s/\b$d\b/**/": This pipes the output of the cal -h command to the sed command, which replaces the double-digit day of the month (represented by the value of d) with two asterisks (**).

SAMPLE SHELL SCRIPT:

```
d=`date +%e`
echo "Today's Date is: $d"
echo "Current Date will be Replaced by * for Single Digit, ** for Double
Digits "
if [ $d -le 9 ]
then
c=`date +%e | cut -c2`
cal | sed " s/\b$c\b/*/"
else
cal | sed " s/\b$d\b/**/"
fi
```

OUTPUT:

EXPERIMENT NO 12

AIM OF THE EXPERIMENT:

Write a shell script that accept the file name, starting and ending line number as an argument and display all the lines between the given line number.

THEORY:

The sed command is a Unix/Linux utility that is used to perform text transformations on input files. The -n option tells sed to suppress the default output, meaning that only lines that are explicitly printed will be displayed.

$m and $n are variables that represent the starting and ending line numbers respectively, and p is a sed command that stands for "print".

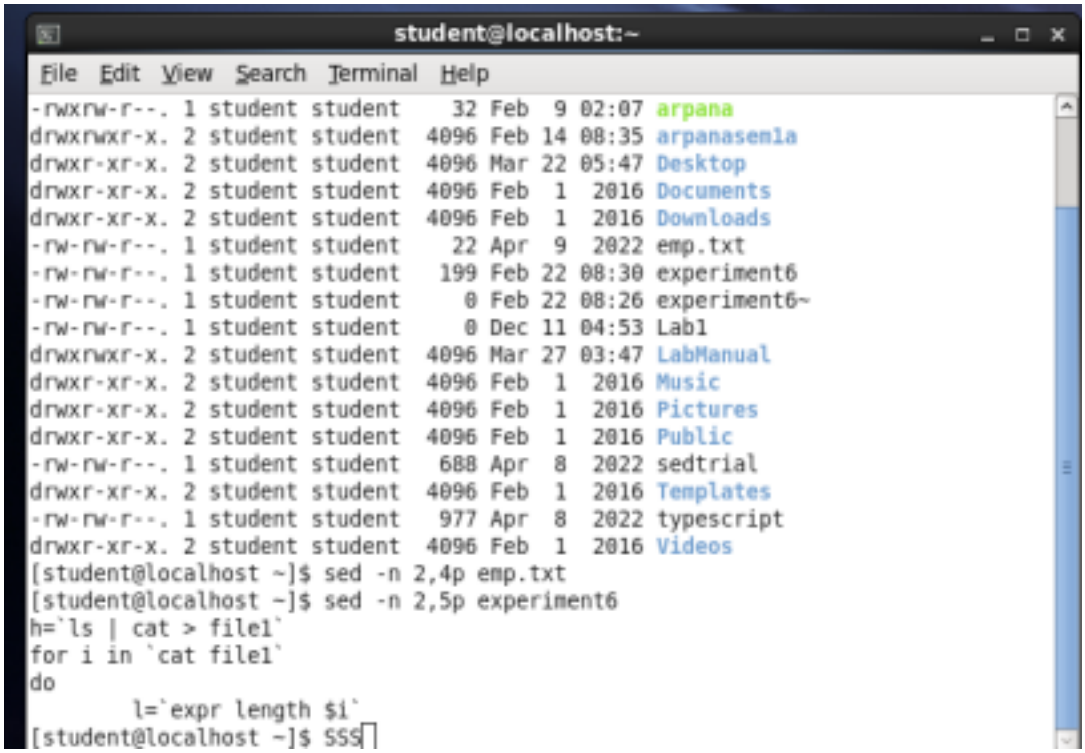
So, sed -n "$m,$n p" $fname will print all the lines in the input file $fname between line $m and line $n. The output will be printed to the terminal.

For example, if you wanted to print lines 5 through 10 of a file named "my_file.txt", you could use the following command:
sed -n "5,10 p" my_file.txt

SAMPLE SHELL SCRIPT:

```
if [ $# -ne 3 ]
then
 echo " Filename, starting line and ending line number should be
passed "
 exit
fi
fname=$1
m=$2
n=$3
echo "Filename : $fname"
echo "Starting line : $m"
echo "Ending line : $n"
echo "content of file "
cat $fname
echo "\n"
if [ $m -lt $n ]
then
 sed -n "$m,$n p" $fname
else
 echo "The order is wrong "
```


OUTPUT:



EXPERIMENT NO 13

AIM OF THE EXPERIMENT:

Create a shell script that folds long lines into 40 columns. Thus, any line that exceeds 40 characters must be broken after 40th, a "\" is to be appended as the indication of folding and the processing. The input is to be supplied through a text file created by the user.

THEORY:

This command is using the sed command-line tool to modify the content of a file called infile and save the modified content to a file called outfile.

The sed command used here is designed to substitute (or "replace") a regular expression pattern with a replacement string. In this case, the pattern to be matched is .\{40\}, which represents any sequence of 40 characters (dots represent any character). The replacement string is &\\\n, which means the matched pattern (represented by &) is followed by a backslash (\) and a newline character (\n).

Putting it all together, the sed command will find every sequence of 40 characters in infile and replace it with the same sequence followed by a backslash and a newline character. This effectively breaks up long lines into multiple lines, each no longer than 40 characters. The modified content is then saved to outfile.

For example, if infile contains a single line with 80 characters, this command would split it into two lines with 40 characters each, like this:

original:
1234567890123456789012345678901234567890123456789012345678901234567
8 90
modified: 12345678901234567890123456789012345678901\
12345678901234567890123456789012345678901

SAMPLE SHELL SCRIPT:

[Note: - Create a text file called infile and check the output of the program in a file called file]

```
sed 's/.\{40\}/&\\\n/g' infile > out.txt
echo "Contents of "infile" after appending a \ after every 40th character is: "
cat out.txt
```


OUTPUT:

EXPERIMENT NO 14

AIM OF THE EXPERIMENT:

Create an awk script that accepts date argument in the form of dd-mm-yy and displays it in the form if month, day and year. The script should check the validity of the argument and in the case of error, display a suitable message.

THEORY:

Before trying the code try the following:

Before doing practical 16

Understand the usage of command line arguments in awk command using the following code.

Step 1: create an awk file argtrial.awk using gedit with the following contents.

```
BEGIN{
for(i=0; i<ARGC;i++)
 print ARGV[i]
}
```

Step 2: on command line execute the following code

```
awk –f argtrial.awk India trial
```

The output of the command will be :

awk

India

Trial

SAMPLE SHELL SCRIPT:

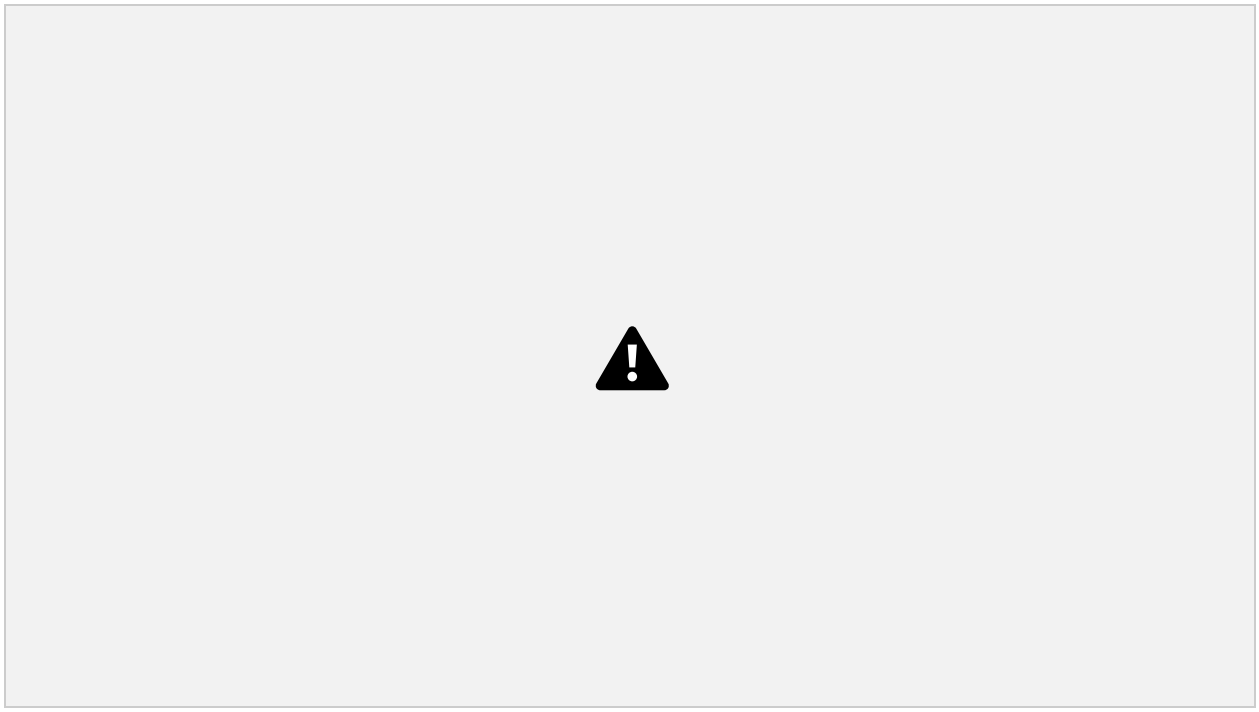
Please note: To execute the shell script as desired in practical16 do the following steps. STEP1: Write an awk script practical16.awk with the following code using gedit

```
BEGIN {
da="312831303130313130313031"
mo="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
dd=substr(ARGV[1],1,2)
mm=substr(ARGV[1],4,2)
yy=substr(ARGV[1],7,4)
l= length(ARGV[1])
if (yy%4==0)
da="312931303130313130313031"
if ((dd>substr(da,2*mm-1,2)) || (mm > 12) || (l≠10))
{
printf "Invalid Date"
exit
}
printf("The date is %d\n The month is %s\n The year is
%d\n",dd,substr(mo,3*mm -2,3),yy)
printf ("%d-%s-%d\n",dd,substr(mo,3*mm-2,3),yy)
}
```

STEP 2: After saving the above script execute the following command on command line with different dates.

```
awk -f practical16.awk 10-09-2001
```

OUTPUT:



EXPERIMENT NO 15

AIM OF THE EXPERIMENT:
Create an awk script to delete duplicated lines from a text file. The order of the original lines must remain unchanged.

THEORY:
This is an awk script that removes duplicates from a file and prints the unique values in the order they appeared in the original file. Here's an explanation of how it works:
a[++n]=$0

This line stores the current line in an array a and increments n. The array is indexed by n, so each line is stored in a separate element of the array.

```
END {
printf("List of Values in the File after removing the
Duplicates\n")  for(i=1;i≤n;i++)
{
        flag=0
        for(j=1;j<i;j++)
        {
        if (a[i] == a[j])
                {
 flag=1
```

```
                break
                }
        }
        if ( flag == 0 )
                printf("%s\n",a[i])
}
}
```

This is the END block of the awk script, which is executed after all the input has been processed. It prints the unique values in the array a in the order they appeared in the original file. The for loop iterates over each element of the array a. For each element a[i], it checks whether it has already appeared in the array up to that point (i.e., for j from 1 to i-1). If it has, the flag variable is set to 1 and the loop breaks. If flag is still 0 after the inner loop, that means a[i] is unique and it is printed to the console using printf("%s\n",a[i]).

SAMPLE SHELL SCRIPT:

Step1:

create a data file with few duplicate records of the type

empno|name|address|designation|department|9000
e001|arti|abc|manager|hr|10000
e002|anu|newgen|manager|hr|50000
e003|ishi|MDI|General Manager|Finance|500000
e004|Vihaan|IIM|CEO|Finance|1000000
e007|aradhana|hsbc|VP|5000000
e008|amit|hsbc|Assistant|400000
e007|aradhana|hsbc|VP|5000000


```
                a[++n]=$0
                END {
```

Step 2: Create a file emp.awk as given below

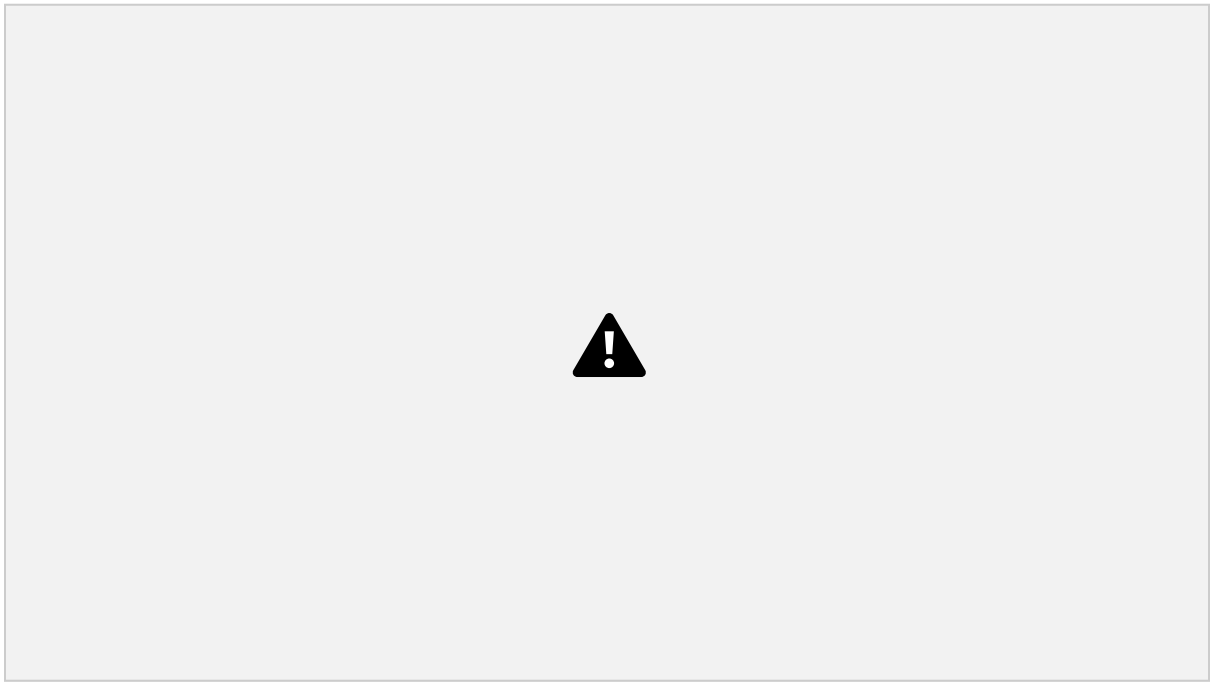
```
printf("List of Values in the File after removing the Duplicates\n")
for(i=1;i≤n;i++)
{
flag=0
for(j=1;j<i;j++)
{
if (a[i] == a[j])
{
flag=1
break
}
}
if ( flag == 0 )
printf("%s\n",a[i])

}
}
```

Step 3: Execute the code

```
awk -F "|" -f emp.awk emp.txt
```

OUTPUT:



EXPERIMENT NO 16

AIM OF THE EXPERIMENT:

Create an awk script to find out total number of books sold in each discipline as well as total book sold using associate array down table as given below: Electrical 34, Mechanical 67, Electrical 80, Computer Science 43, Mechanical 65, Civil 98, Computer Science 64.

THEORY:

The script processes some input data, likely in tabular form, and calculates the total sales for each department or category. Here's how it works:

The first line of the script, count[$1]=count[$1]+$2, initializes an associative array called count. The array uses the value in the first column of the input data as the key, and stores the sum of values in the second column associated with that key. This line essentially accumulates the total sales for each department.

The script then reads the input data line by line, and for each line, it executes the first line of the script to update the count array.

Once all the input data has been processed, the END block of the script is executed. This block calculates the total sales for all departments and prints a report.

The for loop in the END block iterates over all the keys in the count array, which correspond to the departments. It prints the department name and its total sales.

The totsales variable is initialized to 0 before the loop starts, and its value is incremented by each department's total sales.

Finally, the printf statements print a separator line and the total sales for all departments. The %d format specifier is used to print an integer value, which is the sum of all the department sales. The \n character at the end of the string is a newline, which starts a new line of output.

SAMPLE SHELL SCRIPT:

Step 1:

Create a text file 'practical18.txt' with field delimiter of type "|"

```
Electrical|34
Mechanical|67
Electrical|80
Computer Science|43
Mechanical|65
Civil|98
Computer Science|64
```

Step 2: Create an awk script with name practical18.awk with the following contents.

```
BEGIN {
printf "Department\tAmount of Books Sold\n"
printf "=============================\n"
}
{
count[$1]=count[$1]+$2
}
END {
totsales=0
for( bk in count )
{
        print bk "\t" count[bk]
        totsales=totsales+count[bk
] }
printf "=============================================\n"
printf "\nTotal Amount of Sales in all the Departments:%d" , totsales
printf "\n"
 }
```


Step 3: On command line execute the following command

```
awk –F "|" –f practical18.awk practical18.txt
```

OUTPUT: