LAB Assignment No.-9

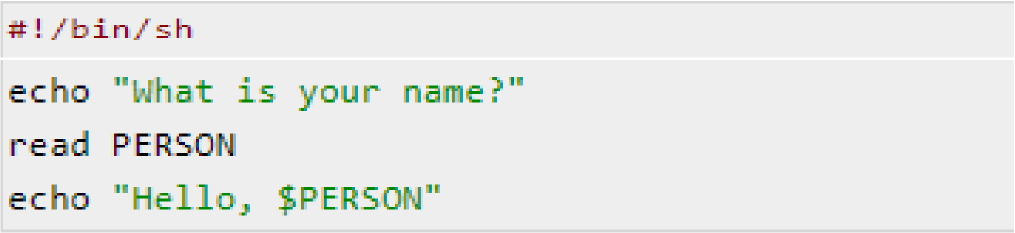
Operating Systems

(UCS - 303)

Payas Jain
101917016
CSE-1

1. Shell script –

- A shell script is a computer program designed to be run by the Linux shell which could be one of the following:
    - The Bourne Shell
    - The C Shell
    - The Korn Shell
    - The GNU Bourne-Again Shell
- A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.
- Shell scripts have several required constructs that tell the shell environment what to do and when to do it.
- As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**.
-
- Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with **.sh** file extension eg. **myscript.sh**

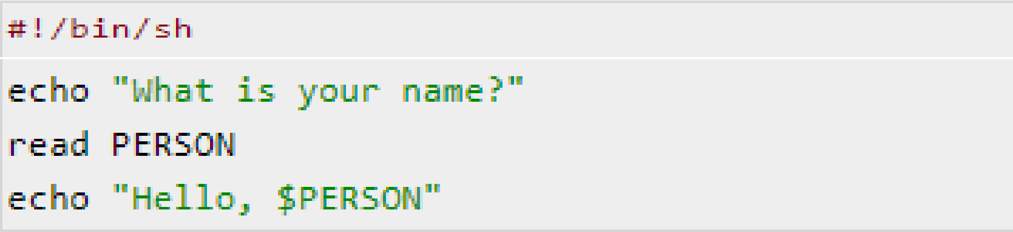


```
#!/bin/sh
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

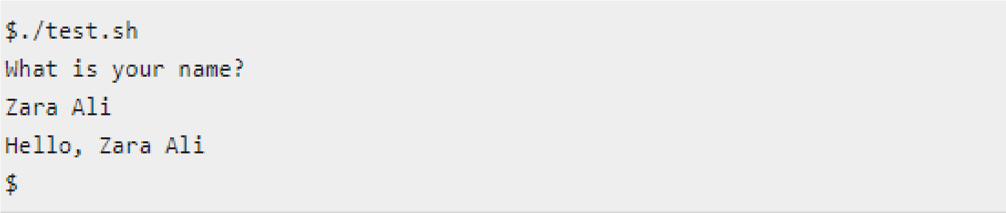
2. Read operation in shell script –

- **read command** in Linux system is used to read from a file descriptor.
- Basically, this command read up the total number of bytes from the specified file descriptor into the buffer.

- If the number or count is zero then this command may detect the errors. But on success, it returns the number of bytes read.
- Zero indicates the end of the file. If some errors found then it returns -1.



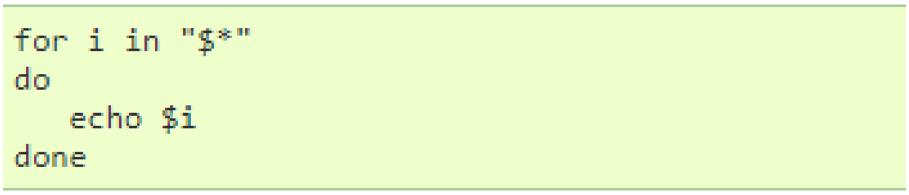
```
#!/bin/sh
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```



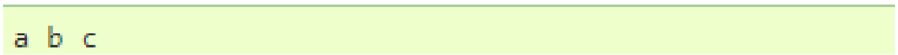
```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

3. Using command line argument –

- Arguments or variables may be passed to a shell script.
- It simply lists the arguments on the command line when running a shell script.
- In the shell script, $0 is the name of the command run (usually the name of the shell script file); $1 is the first argument, $2 is the second argument, $3 is the third argument, etc...



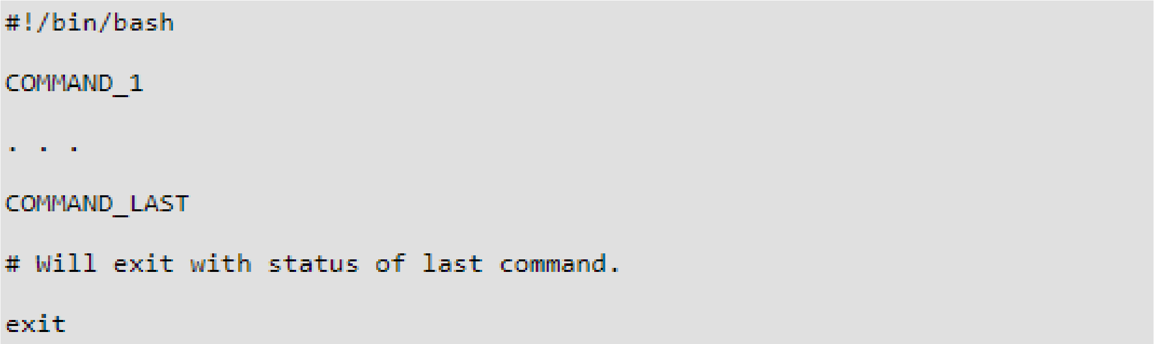
```
for i in "$*"
do
   echo $i
done
```



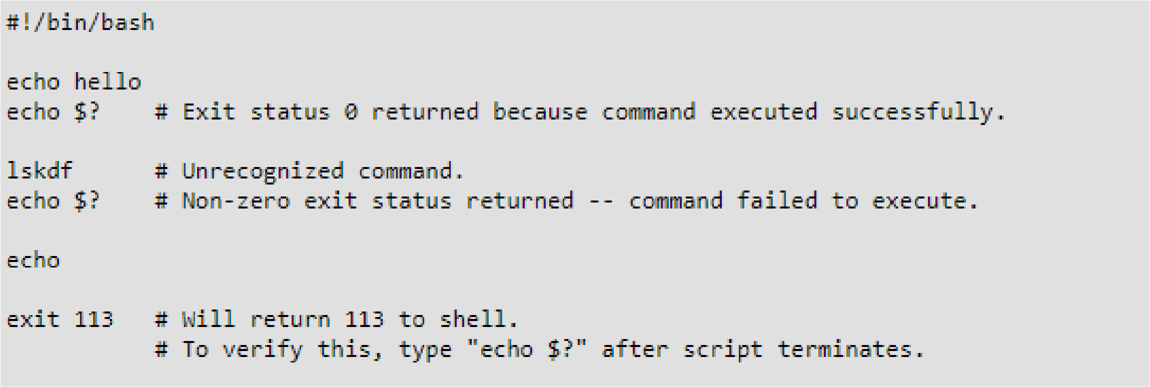
```
a b c
```

4. exit and EXIT status of commands –

- The **exit** command terminates a script. It can also return a value, which is available to the script's parent process.
- Every command returns an *exit status* (sometimes referred to as a *return status* or *exit code*). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an *error code*.
- Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status.
- When a script ends with an exit that has no parameter, the exit status of the script is the exit status of the last command executed in the script.



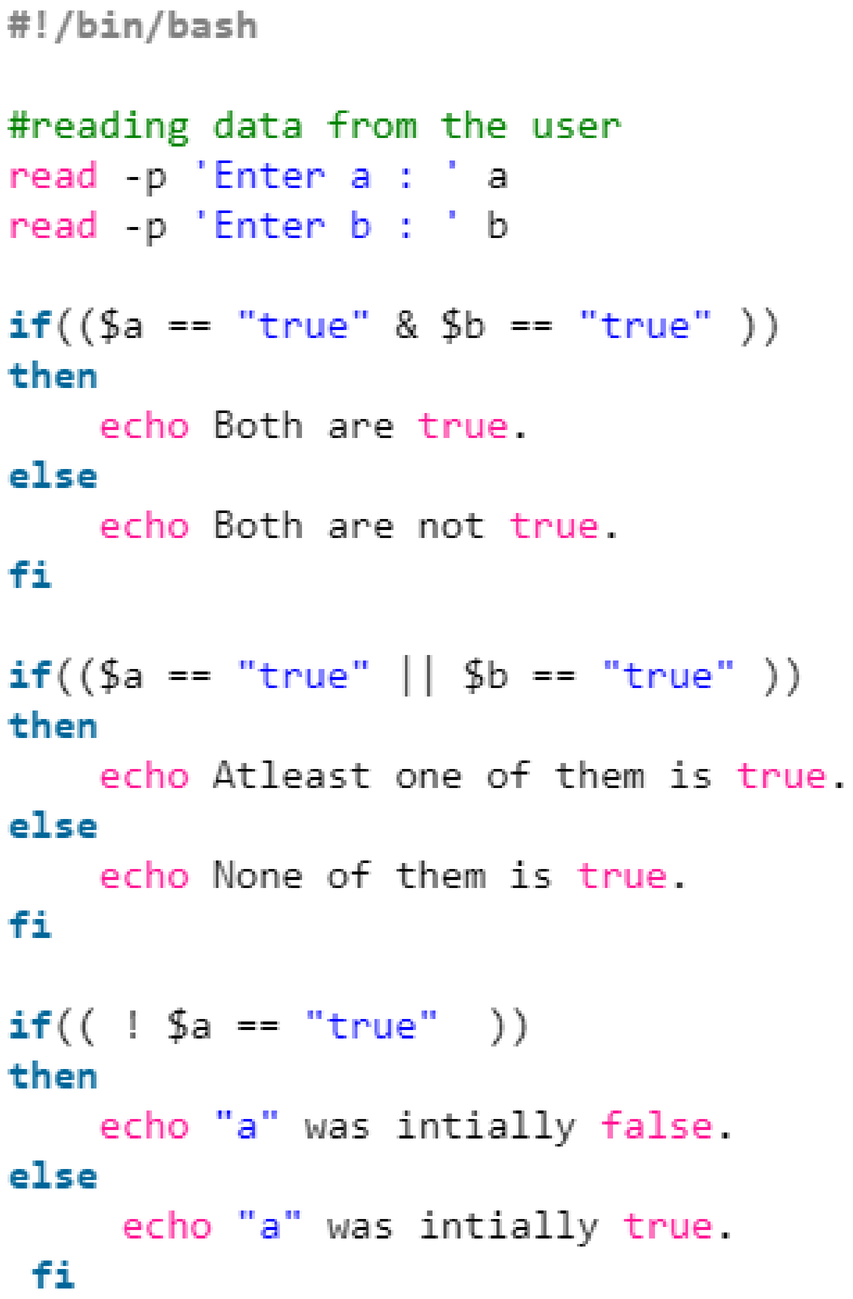
```
#!/bin/bash

COMMAND_1

. . .

COMMAND_LAST

# Will exit with status of last command.

exit
```



```
#!/bin/bash

echo hello
echo $?    # Exit status 0 returned because command executed successfully.

lskdf      # Unrecognized command.
echo $?    # Non-zero exit status returned -- command failed to execute.

echo

exit 113   # Will return 113 to shell.
           # To verify this, type "echo $?" after script terminates.
```

5. Logical operators &&, || -

- They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:
- **Logical AND (&&)** : This is a binary operator, which returns true if both the operands are true otherwise returns false.
- **Logical OR (||)** : This is a binary operator, which returns true is either of the operand is true or both the operands are true and returns false if none of then is false.

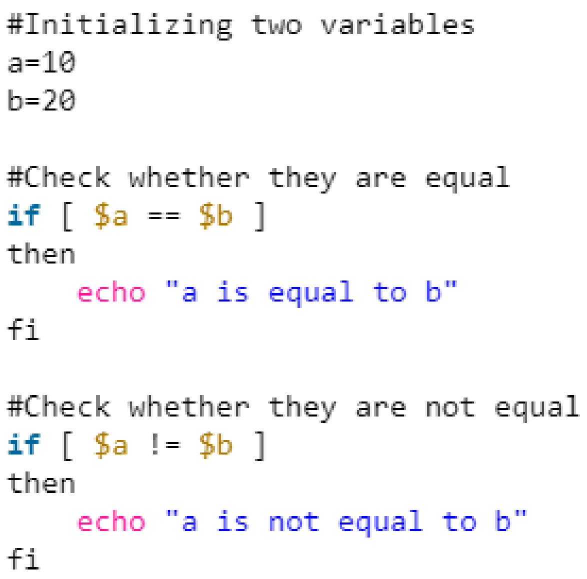


```
#!/bin/bash

#reading data from the user
read -p 'Enter a : ' a
read -p 'Enter b : ' b

if(($a == "true" & $b == "true" ))
then
    echo Both are true.
else
    echo Both are not true.
fi

if(($a == "true" || $b == "true" ))
then
    echo Atleast one of them is true.
else
    echo None of them is true.
fi

if(( ! $a == "true"  ))
then
    echo "a" was intially false.
else
     echo "a" was intially true.
 fi
```

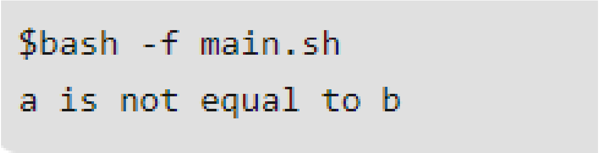
6. Conditional construct – If –

- if statement
  This block will process if specified condition is true.
- **Syntax-**

```
if [ expression ]
then
  statement
```



```
#Initializing two variables
a=10
b=20

#Check whether they are equal
if [ $a == $b ]
then
    echo "a is equal to b"
fi

#Check whether they are not equal
if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```



```
$bash -f main.sh
a is not equal to b
```

fi

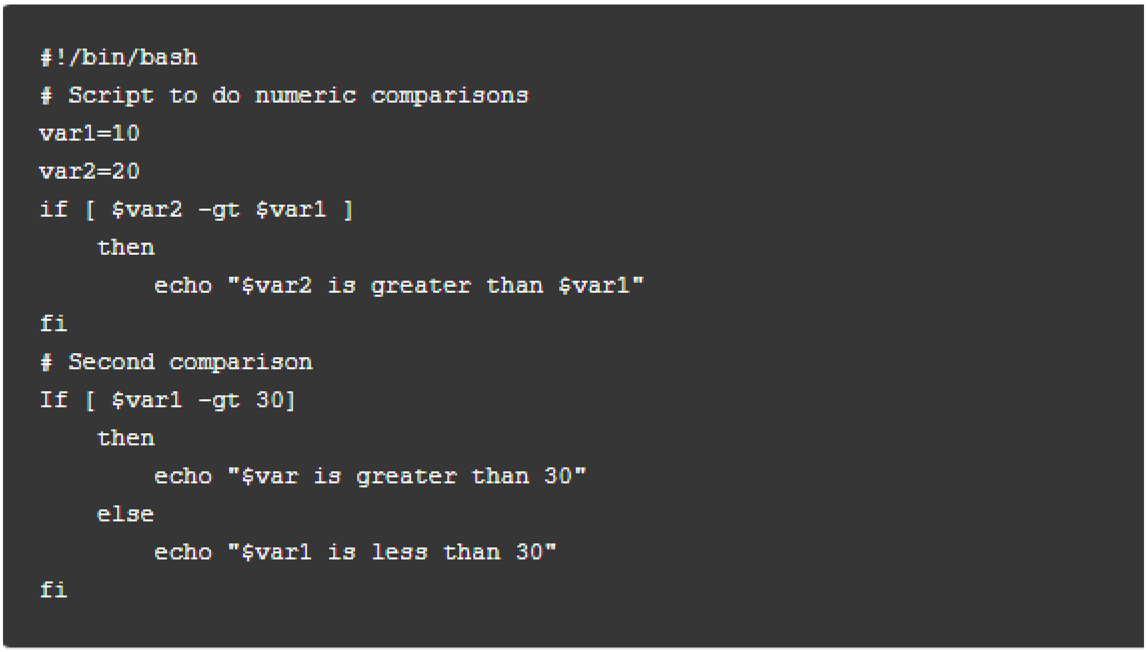
7. Using test AND [ ] to evaluate an expression –

- We can use either the test command or just an expression ([expression]) to evaluate whether an expression is true (zero) or false (non-zero). As the following example illustrates, you can accomplish the same expression evaluation different ways.
- **Evaluating expressions**

```
$ x=3

$ y=7

$

$ test $x -lt $y && echo " Option 1 -- ${x} is less than ${y} "
```

8. Numeric Comparison –

- ➢ This is one the most common evaluation method i.e. comparing two or more numbers. We will now create a script for doing numeric comparison, but before we do that we need to know the parameters that are used to compare numerical values . Below mentioned is the list of parameters used for numeric comparisons
- **num1 -eq num2** check if 1st number is equal to 2nd number
- **num1 -ge num2** checks if 1st number is greater than or equal to 2nd number
- **num1 -gt num2** checks if 1st number is greater than 2nd number
- **num1 -le num2** checks if 1st number is less than or equal to 2nd number
- **num1 -lt num2** checks if 1st number is less than 2nd number
- **num1 -ne num2** checks if 1st number is not equal to 2nd number



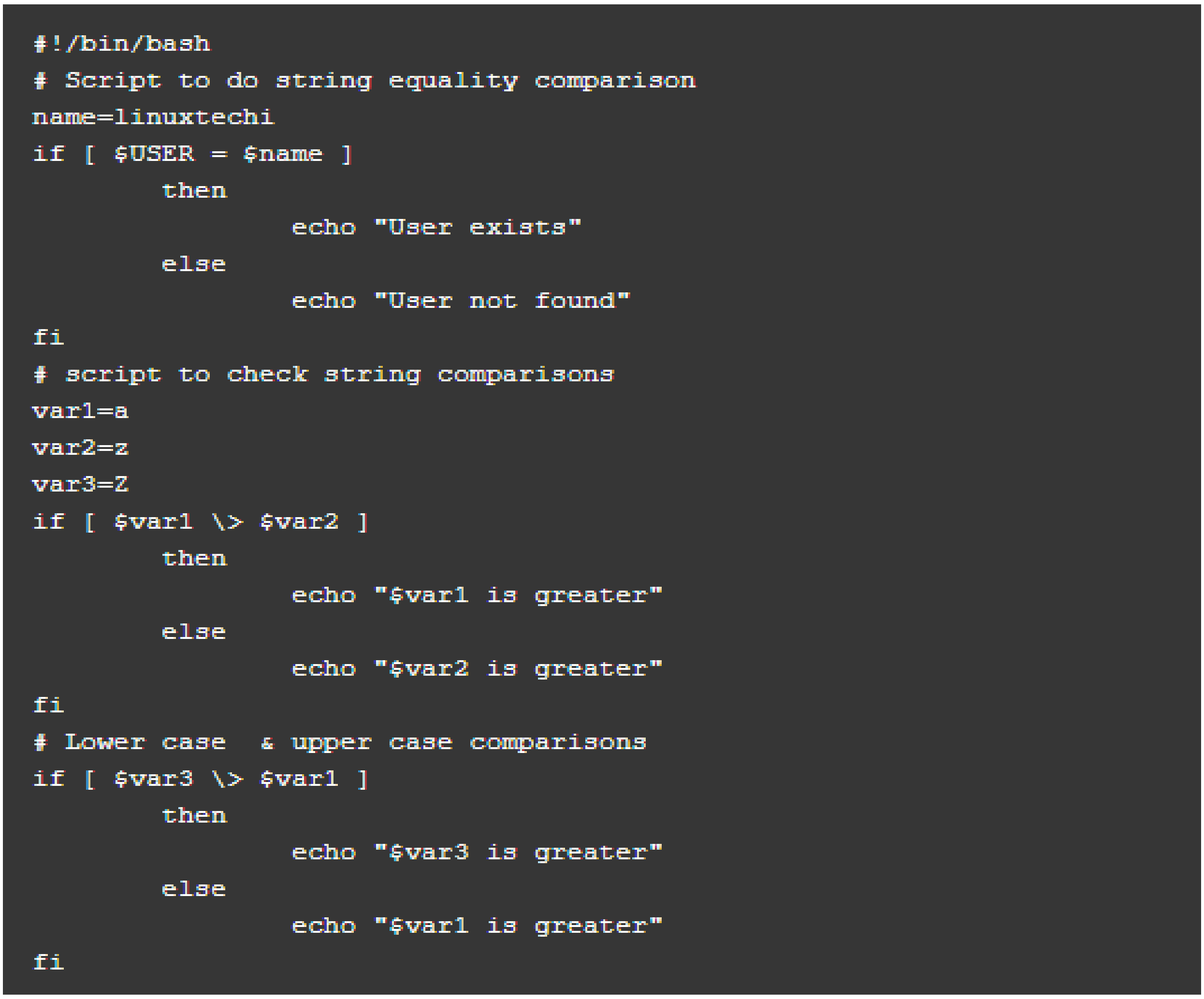
```
#!/bin/bash
# Script to do numeric comparisons
var1=10
var2=20
if [ $var2 -gt $var1 ]
    then
        echo "$var2 is greater than $var1"
fi
# Second comparison
If [ $var1 -gt 30]
    then
        echo "$var is greater than 30"
    else
        echo "$var1 is less than 30"
fi
```

9. String Comparison –

- ➢ When creating a bash script, we might also be required to compare two or more strings & comparing strings can be a little tricky. For doing strings comparisons, parameters used are –

- var1 = var2 checks if var1 is the same as string var2
- var1 != var2 checks if var1 is not the same as var2

- var1 < var2 checks if var1 is less than var2
- var1 > var2 checks if var1 is greater than var2
- -n var1 checks if var1 has a length greater than zero
- -z var1 checks if var1 has a length of zero



```
#!/bin/bash
# Script to do string equality comparison
name=linuxtechi
if [ $USER = $name ]
        then
                echo "User exists"
        else
                echo "User not found"
fi
# script to check string comparisons
var1=a
var2=z
var3=Z
if [ $var1 \> $var2 ]
        then
                echo "$var1 is greater"
        else
                echo "$var2 is greater"
fi
# Lower case  & upper case comparisons
if [ $var3 \> $var1 ]
        then
                echo "$var3 is greater"
        else
                echo "$var1 is greater"
fi
```

LAB Assignment No.-11

Operating Systems

(UCS - 303)

Payas Jain
101917016
CSE-1

1. C++ program to simulate the Banker's algorithm for deadlock avoidance –

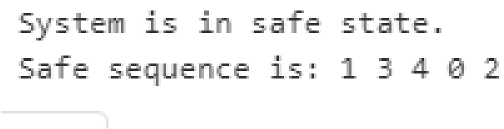
- **Code** –

```
// C++ program to illustrate Banker's Algorithm
#include<iostream>
using namespace std;

// Number of processes
const int P = 5;

// Number of resources
const int R = 3;

// Function to find the need of each process
void calculateNeed(int need[P][R], int maxm[P][R],
                int allot[P][R])
{
    // Calculating Need of each P
    for (int i = 0 ; i < P ; i++)
        for (int j = 0 ; j < R ; j++)

            // Need of instance = maxm instance -
            //                allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
}

// Function to find the system is in safe state or not
bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
    int need[P][R];

    // Function to calculate need matrix
```

```
calculateNeed(need, maxm, allot);

// Mark all processes as infinish
bool finish[P] = {0};

// To store safe sequence
int safeSeq[P];

// Make a copy of available resources
int work[R];
for (int i = 0; i < R ; i++)
    work[i] = avail[i];

// While all processes are not finished
// or system is not in safe state.
int count = 0;
while (count < P)
{
    // Find a process which is not finish and
    // whose needs can be satisfied with current
    // work[] resources.
    bool found = false;
    for (int p = 0; p < P; p++)
    {
        // First check if a process is finished,
        // if no, go for next condition
        if (finish[p] == 0)
        {
            // Check if for all resources of
            // current P need is less
            // than work
            int j;
            for (j = 0; j < R; j++)
                if (need[p][j] > work[j])
                    break;

            // If all needs of p were satisfied.
            if (j == R)
            {
                // Add the allocated resources of
                // current P to the available/work
                // resources i.e.free the resources
                for (int k = 0 ; k < R ; k++)
                    work[k] += allot[p][k];

                // Add this process to safe sequence.
                safeSeq[count++] = p;

                // Mark this p as finished
                finish[p] = 1;

                found = true;
            }
        }
    }

    // If we could not find a next process in safe
    // sequence.
    if (found == false)
    {
        cout << "System is not in safe state";
        return false;
    }
}

// If system is in safe state then
// safe sequence will be as below
cout << "System is in safe state.\nSafe"
     " sequence is: ";
for (int i = 0; i < P ; i++)
    cout << safeSeq[i] << " ";

return true;
}

// Driver code
int main()
{
    int processes[] = {0, 1, 2, 3, 4};
```

```
        // Available instances of resources
        int avail[] = {3, 3, 2};

        // Maximum R that can be allocated
        // to processes
        int maxm[][R] = {{7, 5, 3},
                         {3, 2, 2},
                         {9, 0, 2},
                         {2, 2, 2},
                         {4, 3, 3}};

        // Resources allocated to processes
        int allot[][R] = {{0, 1, 0},
                          {2, 0, 0},
                          {3, 0, 2},
                          {2, 1, 1},
                          {0, 0, 2}};

        // Check system is in safe state or not
        isSafe(processes, avail, maxm, allot);

        return 0;
    }
```

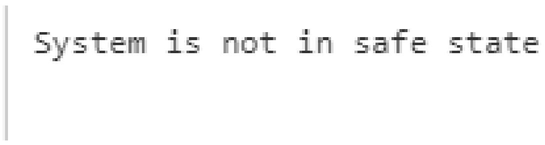
- **Output –**



```
System is in safe state.
Safe sequence is: 1 3 4 0 2
```

When available instance of resources are:

int avail[] = {1, 1, 1};



```
System is not in safe state
```

LAB Assignment No.-12

Operating Systems

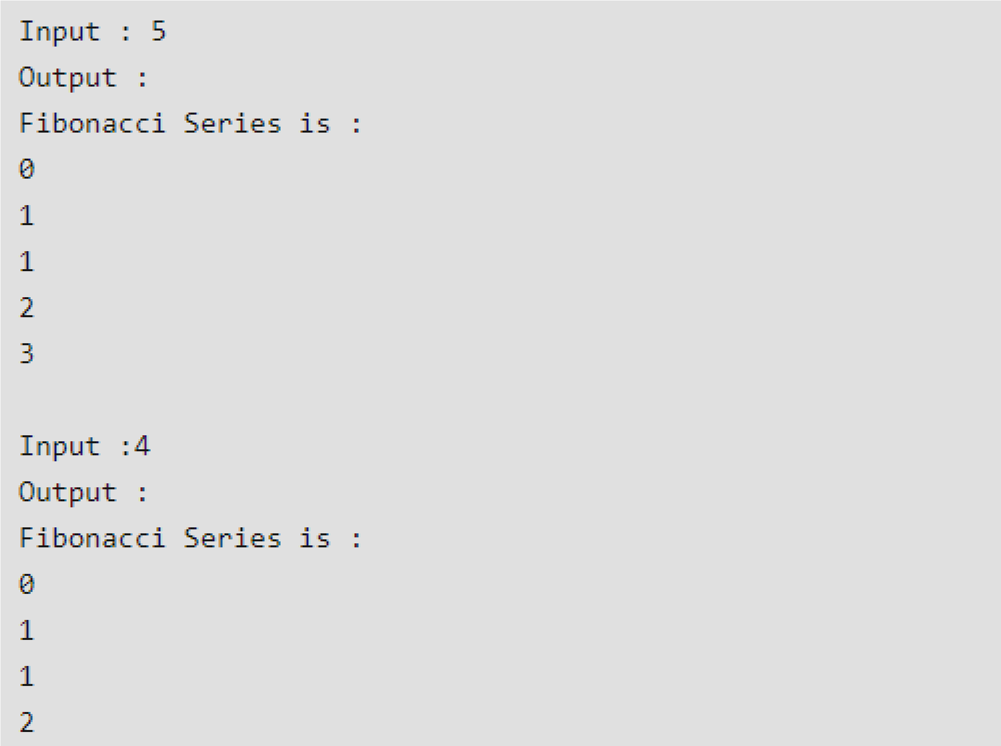
(UCS - 303)

Payas Jain
101917016
CSE-1

1. Shell script to find Fibonacci series –

```
echo "How many number of terms to be generated ?"
read n
x=0
y=1
i=2
echo "Fibonacci Series up to $n terms :"
echo "$x"
echo "$y"
while [ $i -lt $n ]
do
i=`expr $i + 1 `
z=`expr $x + $y `
echo "$z"
x=$y
y=$z
done
```

- **Output –**

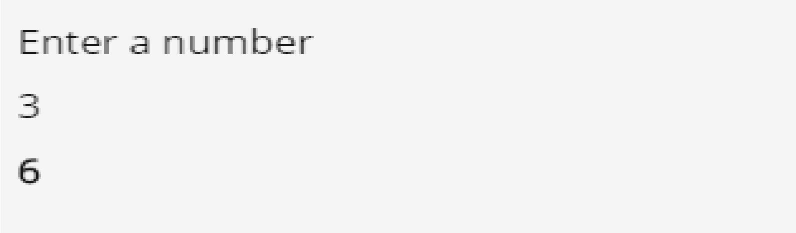


```
Input : 5
Output :
Fibonacci Series is :
0
1
1
2
3

Input :4
Output :
Fibonacci Series is :
0
1
1
2
```

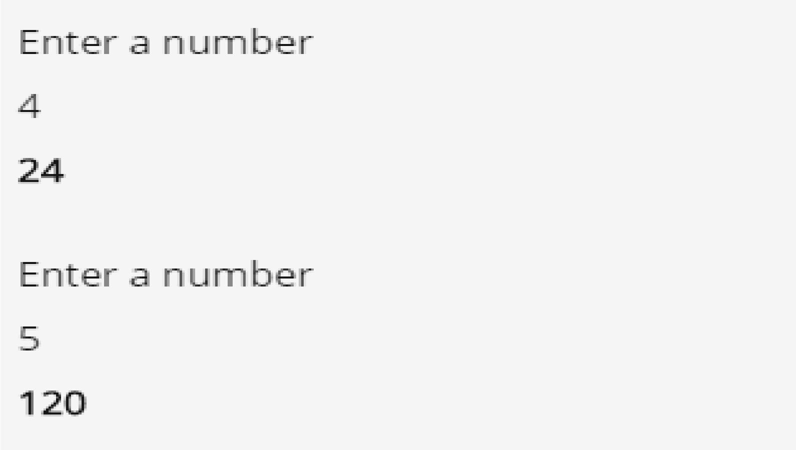
2. Shell script to find factorial of a number –

```
#shell script for factorial of a number
#factorial using while loop

echo "Enter a number"
read num

fact=1

while [ $num -gt 1 ]
do
  fact=$((fact * num))  #fact = fact * num
  num=$((num - 1))      #num = num - 1
done


echo $fact
```

Output –



```
Enter a number
3
6
```



```
Enter a number
4
24

Enter a number
5
120
```

LAB Assignment No.-13

Operating Systems

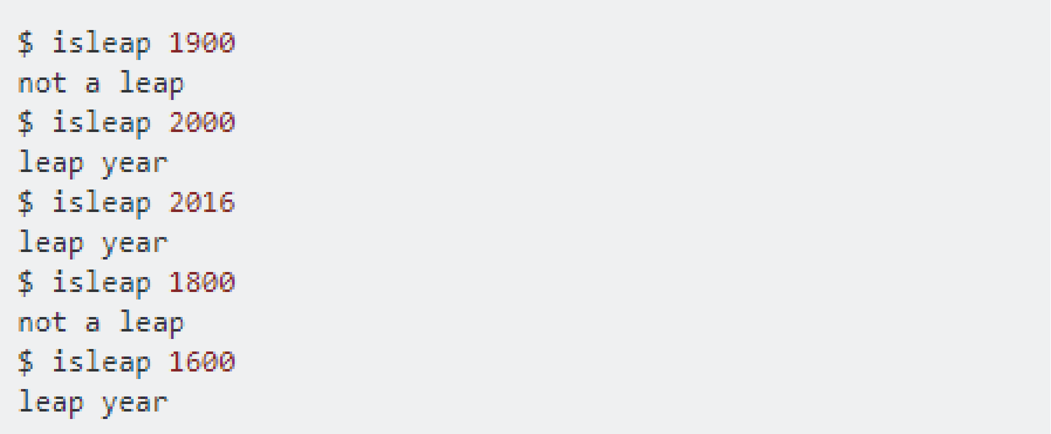
(UCS - 303)

1. Shell script to find whether a given year is leap year or not –

- A year is a leap year if it can be evenly divided by four. For example, 1996 was a leap year. But a year is not a leap year if can be evenly divided by 100 and not by 400. This is why 1700, 1800, 1900 were not leap years, but 2000 was.
- **Code** –

```
#!/bin/bash
# Shell program to read any year and find whether leap year or not
# store year
yy=0
isleap="false"

echo -n "Enter year (yyyy) : "
read yy

# find out if it is a leap year or not

if [ $((yy % 4)) -ne 0 ] ; then
   : #  not a leap year : means do nothing and use old value of isleap
elif [ $((yy % 400)) -eq 0 ] ; then
   # yes, it's a leap year
   isleap="true"
elif [ $((yy % 100)) -eq 0 ] ; then
   : # not a leap year do nothing and use old value of isleap
else
   # it is a leap year
   isleap="true"
fi
if [ "$isleap" == "true" ];
then
   echo "$yy is leap year"
else
```

```
  echo "$yy is NOT leap year"
fi
```



```
$ isleap 1900
not a leap
$ isleap 2000
leap year
$ isleap 2016
leap year
$ isleap 1800
not a leap
$ isleap 1600
leap year
```

- **Output –**

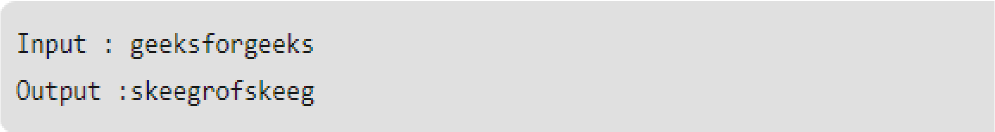
2. Shell script to print the line in opposite order –

- **Code –**

```
// reverse a string using shell script
// reverse a string is in linux

#!/ bin / bash
// reading a string
// using via user input
read - p "Enter string:" string
   // getting the length of given string
   len
   = $
{
   #string
}
// looping for reversing a string
// initialize i=len-1 for reversing a string and run till i=0
// printing in the reverse order of the given string
for ((i = $len - 1; i >= 0; i--))
   do
   // "${string:$i:1}"extract single single character from string.
```

```
reverse = "$reverse${string:$i:1}" done
   echo "$reverse"
```

- **Output –**



```
Input : geeksforgeeks
Output :skeegrofskeeg
```