USP
17CS35

IAT-3 (NOV 2018) Solution

1 (a) Explain the following in PERL with examples.
i) foreach looping construct ii) join [06] CO5 L4

i) The foreach looping construct

This control structure is used to execute a set of statements repeatedly. The general format is

Example 1:

```
foreach $num (1,3,5)
{
        print "The number is $num. \n";
}
```

Output:
The number is 1
The number is 3
The number is 5

Example 2:

```
foreach(1..5)
{
$sum=$sum+$_;
print "The step number is $_ and the sum is $sum\n";
}
```

In the above example, the set of statements appearing within the curly braces are executed five times with $_ assuming the values 1, 2, 3, 4 and 5 one after the other, by default.

The step number is 1 and the sum is 1
The step number is 2 and the sum is 3
The step number is 3 and the sum is 6
The step number is 4 and the sum is 10
The step number is 5 and the sum is 15

(ii) The join function

- ✓ The join function glues or pastes the elements of a list into a string.

- ✓ Join does the opposite of split function
- ✓ The syntax of join function is

  join (EXPR, LIST);

  The first argument EXPR may be any string. The join function puts the EXPR string between the individual elements of the LIST and returns the resulting string.

  Examples:

  1.

```
$ perl -e '
@week_array=("Mon","Tue","Wed","Thu","Fri","Sat","Sun");
$week_string=join(" ",@week_array);
print  $week_string;'
Mon Tue Wed Thu Fri Sat Sun
```

  2.

```
$ perl -e '
> $result=join " ", ("This", "is", "an", "example");
> print $result
> '
This is an example
```

  Note: Note that the EXPR appears only between the elements of the LIST and never before or after them.

  3.

```
$ perl -e '
> $x=join ":",3,4,12,15;
> print "$x\n";
> @y=split /:/,$x;
> print "@y\n";
> $x=join "-",@y;
> print "$x\n"; '
3:4:12:15
3 4 12 15
3-4-12-15
```

1 (b) Write short notes on find and sort commands with examples. **[04] CO4 L1**

***find* Command:**

find command recursively examines a directory tree to search for files matching some criteria, and then takes some action on the selected files.

Syntax:
find path_*list* *selection_criteria* *action*

How find works
1) First, it recursively examines all files in the directories specified in path_list.
2) It then matches each file for one or more selection_criteria
3) Finally, it takes some action on those selected files

Path_list:
Path_list is one or more subdirectories separated by whitespace.

selection_criteria:
The selection criteria always consists of an expression in the form '-operator argument'. If the expression matches the file, the file is selected. We can use more than one selection criteria to match a file.

Action:
Action specifies what action has to be taken on the selected files. We can specify multiple actions.

Example:

$find . -name test -print
./test

In the above command the *path_list* indicates that the search should start from current directory.
The selection criterion is "-name test". Each file in the list is matched against this selection criteria. If the expression matches the file, i.e., if the file name is test, then the file is selected.

The action to be taken on the selected files in the above example is to just print the selected files.

Note:
1. The default path_list is current directory.
2. The default action is to print.
3. when find is used to match a group of filenames with a wild-card pattern, the pattern must be enclosed in double quotes.

Selection Criteria:

1. Locating a file by inode number (inum) :

$find / -inum 4981488

prints all file names have the inode number 4981488.

2. File type (-type)
c can be any of the following character, then file is of type c:
1. d - directory
2. f - regular file
3. p - name pipe (FIFO)
4. s - socket
5. l - symbolic link

$find . -type d -print

3. perm (-perm):
-perm specifies the permission to match. For instance, -perm 666 selects files having read and write permission for all categories of users

find / -perm 775 -type d 2>/dev/null

Note: find uses an AND condition (an implied -a operator between -perm and -type) to select directories that have permission 775. Its selects files only if both selection criteria are fulfilled.

4. Finding unused files (-mtime and -atime)
-mtime n
    File's data was last modified n*24 hours ago.
-atime n
    File was last accessed n*24 hours ago

The find operators (!, -o and -a)
The ! Operator is used before an option to negate its meaning. So,

find . ! -name *.c
selects all but the c program files.

The -o operator represents the OR condition and the -a represents the AND condition. -a is implied by default when two selection criteria are placed together.

find $HOME \(-name "*.sh" -o -name "*.pl" \) -print

find $HOME \(-perm 777 -a -type d \) is same as
find $HOME -perm777 -type d

Options available in the Action Component

Different actions are
1. -print -> prints selected file on standard output
2. -ls -> executes ls -lids command on selected files
3. -exec cmd -> Execute UNIX command cmd followed by {} \;
4. -ok > makes the command interactive. Not all commands are interactive, so to make a command interactive we use -ok option.

Sorting on primary key (-k):

Example 1: To sort on the second field (name) of file shortlist
$ sort -t "|" -k 2 shortlist

Example 2: To reverse the sort order -r:
$ sort -t "|" -r -k 2 shortlist
or
$ sort -t "|" -k 2r shortlist

Sorting on secondary key (-k):
We can sort on more than one key, that is, we can provide a secondary key to sort.
If the primary key is the third field, and the secondary key is the second field, then we need to specify for every -k option, where the sort end.

Example 3:
$ sort -t"|" -k 3,3 -k 2,2 shortlist

The above command sorts the file by designation and name. -k 3,3 indicates that sorting starts on third field and ends on the same field.

Example 4: To sort on columns:

We can also specify a character position within a field to be the beginning of sort.

To sort the file according to the **year of birth**, we need to sort on the seventh and eighth column positions within the fifth field.

$ sort -t"|" -k 5.7,5.8 shortlist
5678|sumit chakrabarty |d.g.m. |marketing |04/09/43|6000
2365|barun sengupta |director |personnel |05/11/47|7800
9876|jai sharma |director |production|03/12/50|7000
2233|a.k. shukla |g.m. |sales |12/12/52|6000
5423|n.k. gupta |chairman |admin |08/30/56|5400

The -k option also uses the form -k m.n, where n is the character position in the mth field. **So 5.7,5.8 means that sorting starts on column 7 of the fifth field and ends on column 8.**

Example 5: Numeric Sort (-n):

$ sort -n numfile
2
4
10
27

Example 6: Removing Repeated Lines (-u):

The -u (unique) option removes repeated lines from a file.

To specify the output file (-o):
Even though sort's output can be redirected to a file, it provides -o option to specify the output filename. **The input and the output filenames can be the same.**

Example 7: The output file is different from input file.
$ sort -o sortedlist -k 3 shortlist

Example 8: The output file is same as input file:
$ sort -t "|" -o shortlist -k 4 shortlist

Example 9: To check whether the file has actually been sorted in the default order, we use the -c (check) option:

2 (a) Explain the arrays and lists in PERL with examples.
[05] CO5 L4

Arrays

- Arrays are the placeholders of lists. An array is created by assigning a list to it as shown below.
  @subjects=("physics","chemistry","maths");
- All the elements of a list now are available under a common name, the array variable name. In the above example, subjects is the array name.
- Array variable names begin with the @ character. Array name - @subjects
- Individual elements of an array are accessed using indexes. These indexes begin with a 0 (zero) and progress in steps of one as 0, 1, 2, 3 and so on. Thus the value of $subjects[0] will be physics, $subject[1] will be chemistry and so on.

- Individual elements of an array are accessed as scalars, that is, $ prefix is used with the variable name rather than the @ prefix.
- Arrays are not of fixed size. They grow and shrink dynamically as elements are added and deleted.
- Array assignment is quite flexible in perl. We can use the range operator or even assign values selectively in a single statement.
- When used as the rvalue of an assignment, @subjects evaluates to the length of the array.
  $length = @subjects

  $ perl -e '@subjects=("physics","chemistry","maths"); $length=@subjects; print $length;'
  3$
- The $# prefix to an array name signifies the last index of the array. Its always one less than the size of the array. The $# mechanism can be used to set the array to a specific size or delete all its elements. Previously defined array elements that fall outside the new index are deleted.

  $last_index=$#subjects;

  $ perl -e '@subjects=("physics","chemistry","maths"); $last_index=$#subjects; print $last_index;'
  2$
  $#subjects=10; Array size is now 11
  $#subjects=-1; No elements

Examples:

1.
$ perl -e '@x=(1..12); print @x;'
123456789101112$

2.
$ perl -e '@month[1,3..5,12]=("Jan", "Mar", "Apr", "May","Dec"); print $month[0];'
$

$ perl -e '@month[1,3..5,12]=("Jan", "Mar", "Apr", "May","Dec"); print $month[1];'
Jan

$ perl -e '@month[1,3..5,12]=("Jan", "Mar", "Apr", "May","Dec"); print @month;'
JanMarAprMayDec$

3.
$ perl -e '
@subjects=("physics","chemistry","maths"); print **@subjects**; print "\n";'

physicschemistrymaths

4.
$ perl -e '
@subjects=("physics","chemistry","maths"); print **@subjects[0]**; print "\n";'
physics

6.
$ perl -e '@month=(qw/Jan Feb Mar Apr May/); print @month;'
JanFebMarAprMay$

Lists
List is a collection of scalar.

The following is an example of list:

(“Jan”, 123, “How are you”, -34.56, Dec)
A list need not contain data of the same type.
The elements in a list are separated by comma (,) and the entire collection of scalars is enclosed within parentheses. The use of parentheses is optional.

Assigning values to the elements of a list:

Values to the elements of a list can be assigned individually to every scalar variable of the list one by one or at a single stretch by using the following

Syntax

($stone, $wood,$liquid,$age) = (“marble”,”teak”,”Hg”,29);

In the above statement, $stone will get the value “marble”, $wood will get “teak”, $liquid will get “Hg” and so on.

Sometimes the number of elements on LHS and the number of values on the RHS may not be equal. In such cases – excess list elements in the list remain undefined. However, in case there are more values (on RHS), values in excess will be just neglected.

($stone, $wood, $liquid, $age) = (“marble”,”teak”);   #liquid and age remain undefined

($a,$b) = ($b,$a);

The above assignment statement swaps the contents of the two scalar variables $a and $b.

2 (b) Explain file handling in PERL. [05] CO5 L4

File handling in Perl:

- Perl provides low-level file handling functions that allow us to hard-code the source and destination of the data stream in the script itself.
- A filehandle is similar to a file descriptor in C.
- By convention, filehandle is represented in uppercase letters.
- As in UNIX, Perl also has three standard input/output streams. They are standard input, standard output and standard error streams. The respective filehandles are STDIN (connects to the keyboard), STDOUT (connects to the display screen), and STDERR (also connects to the display screen).
- Apart from the above mentioned three file handles, Perl has four more reserved file handles. They are NULL, DATA, ARGV and ARGVOUT.

The NULL filehandle

- This is a special filehandle that allows scripts to get input from either STDIN of from each file listed on the command line.
- It is represented by the symbol '<>' and is called diamond operator or line-reading operator or angle operator.

- Examples:
  1. $perl -e 'while (<>) {print;}'
  2. $perl -e 'while (<>) {print;}' test.txt
  3. $perl -e 'while (<>) {print uc $_;}'
  4. $perl -n -e '$_ = uc $_; print;'
  5. $perl -p -e '$_= uc $_;
  6. $perl -pe '$_ = uc;' test.txt

- In examples 1,3,4 and 5 the input is accepted from the keyboard. Here, the input session is terminated by <ctrl-d> keys.
- In examples 2 and 6 the input is taken from the file test.txt specified in the command line.
- The function uc() converts its arguments to uppercase.
- In some examples no argument is given to print function (1,2 and 4) and in example 6 no argument is given to uc function. In these two case the functions act upon the default, special variable $_. **Also note that it is not mandatory to use parentheses surrounding the arguments of perl functions.**
- The **-n** option implies the existence of the while (<>) {...}. So no explicit while loop is required. (example 4)
- The **-p** option eliminates the explicit use of both while (<>) {...} and print. Example 5 and 6.

The open() function

In Perl, file is opened using the open() function. The general format of this function is

open (FILEHANDLE,"[>|>>] filename");

> - opens the mentioned file for writing. If the file doesn't exist, it will be created. If it exists, it will be over written.

>> - The mentioned file is opened in append mode. If the file doesn't exist it will be created. If it already exists, it will start writing from the end of file existing file.)
| - The use of pipe (|) option directs the output to the mentioned destination as a filter.

Default option - '<' : If none of the above option is specified, the file is opened in read mode. One can use the option < to explicitly mention the read mode.

+< - The use of option +< opens the file for both reading and writing.
+> - The use of the option +> creates a new file that has read permission.

Example:
open (INFILE,">openeg.txt");

The INFILE is the file handle of the file openeg.txt. Once the file is opened functions that read and write the file will use the filehandle to access the file.

open(OUTFILE, ">openegout.txt");
open(OUTFILE, ">>openegappend.txt);

open(INFILE, "sort emplist |"); # input from sort output
open(OUTFILE, "| lp"); # output to print spooler

The close() function:

An open file has to be closed after the necessary processing has been done. This is done by using the close() function.

The syntax is
close(FILEHANDLE);

Closes the file referred to by the FILEHANDLE.
Opening a file that is already open, closes the file and reopens it.

Example program: To read from one file and write the data read to another file

```
#!/usr/bin/perl
```

```
#Script : fileioeg.pl - shows use of low level I/O available in perl.
#reads from one  file and writes the data read to another file.

open(INFILE,"emp.lst") || die ("Cannot open file emp.lst");
open(OUTFILE,">out.txt");
while (<INFILE>)
{
      print OUTFILE;
}

close(INFILE);
close(OUTFILE);
```

Note:

1. If we don't close the files before terminating the script, perl closes them on its own.
2. If the number of print statements have to write to the same filehandle, say FILEOUT, we can assign this filehandle as the default using **select(**FILEOUT**);** Then subsequent **print** statements don't need to use the FILEOUT argument.

File Tests:

Before doing file related operations, it is recommended practice to find out whether the target file can be manipulated or not. For this, certain properties of files must be tested before carrying out any type of operation on them.

Following is a list of some of the basic and more useful file tests

Sl. No.	Operator	Meaning
1	-e	File or directory exists
2	-r	File or directory is readable
3	-w	File or directory is writable
4	-x	File or directory is executable
5	-f	Regular file
6	-d	Directory file

7	-s	File has non-zero size
8	-T	Text file
9	-B	Binary file
10	-M	Returns the time elapsed in hours since the file was last modified

Example:
$ perl -e '
$x="emp.lst";
print "File - $x is readable\n" if -r $x;
print "File - $x is executable\n" if -x $x;
print "File - $x has non-zero size\n" if -s $x;
print "File - $x exists\n" if -e $x;
print "File - $x is a text file\n" if -T $x;
print "File - $x is a binary file\n" if -B $x;
print "File - $x is a file\n" if -f $x;
print "File - $x is writable\n", if -w $x; '
File - emp.lst is readable
File - emp.lst has non-zero size
File - emp.lst exists
File - emp.lst is a text file
File - emp.lst is a file
File - emp.lst is writable

2.
$ perl -e '
$x="emp1.lst"; printf "file - $x was modified %0.3f days back\n", -M $x'
file - emp1.lst was modified 1.973 days back

The die() function:

When an attempt to open a file with a wrong name or path is made, it is necessary to trap such situations and display proper messages. In Perl, this is achieved by using the function called die();
Its usage is shown in the above perl program – fileioeg.pl.

3 (a) Explain the mechanism of process creation. [06] CO6 L4
Mechanism of Process Creation:
There are three distinct phases in the creation of a process and uses three important system calls or functions viz., **fork**, **exec** and **wait**.

1. fork:

A process in UNIX is created with the **fork** system call. Fork creates a copy of the process that invokes it. The process image is practically identical to that of the calling process, except for a few parameters like the PID. When a process is forked in this way, the child gets a new PID. The forking mechanism is responsible for the multiplication of processes in the system.

2. exec:
fork creates a process but it is not enough to run a new program. To run a program, the forked child calls exec system call which overwrite its own image with the code and data of the new program. This mechanism is called exec, and the child process is said to exec a new program. No new process is created here, the PID and PPID of the exec'd process remain unchanged.

3. wait:
The parent then executes the wait system call to wait for the child to complete. It picks up the exit status of the child and then continues with its other functions. Note that a parent may not decide to wait for the child to terminate.

Execution of a command:

- When we run a command (say cat) from the shell, the shell first creates a new process by calling **fork system call**.
- In the newly forked child process the **exec system call** is called to execute the cat program. **The exec system call replaces the shell program and data with the program and data of cat and executes cat command.**
- The parent calls the wait system call which waits for cat to terminate and then picks up the exit status of the child. This is the number returned by the child to the kernel and has great significance in both shell programming and systems programming.

3 (b) What chop() and split() functions do? Explain. [04] CO5 L1

chop()
Removes the last character from a string or array of strings completely.

Examples:
1.
$ perl -e 'chop($name=<STDIN>); print $name;'
cmrit
cmrit$

In the above example chop removed the newline character. Hence the prompt is displayed in the same line of output.

2.
$ perl -e 'chop($name="CMRIT"); print $name;'
CMRI$

In the above example chop removed the last character ‘T’ from “CMRIT.

3.
$ perl -e 'chomp($name="CMRIT"); print $name;'
CMRIT$

In the above example chomp did not remove the last character from “CMRIT” as it does not end with new line character.

The split Function: Splitting into a list or array

- This function breaks a string into its constituent elements according to a separator or breaks up a line or expression into fields. These fields are assigned either to variables or an array.
- The separator can be anything like a white space, tab, colon or any character.
- The syntax is -

  ($var1, $var2, $var3,...) = split (/separator/,$string);

  @array=split(/separator/,$string);

- Anything mentioned as separator between the two forward slashes will be the regular expression. It splits the string $string on separator.

- $string is optional and in its absence, $_ is used as default.
  @colors=split(/:/); # $_ is the default string
- When no field separator is mentioned explicitly white spaces are taken as the field separator by default.
  split($string); # considers white space as the field separator
- Split can also be used without an explicit assignment, in which case it populates the built-in array,
  split(/:/); # returned fields are stored in the array @_
  split(); # white space is the separator, $_ is the string and the returned fields are stored in the array @_

- **Example: Splitting into variables**
  1.
  $ perl -e '

```
$colors=("red,blue,green");
($v1,$v2,$v3)=split(/,/,$colors);
print ($v1.$v2.$v3);'
redbluegreen
```

4 (a) Using command line argument, write a PERL program to find whether a given number is a leap year or not. [05] CO5 L1

```
#!/usr/bin/perl
# script: leap.pl - determines whether a year is leap year or not
#
die("You have not entered the year\n") if (@ARGV == 0);
$year = $ARGV[0];
$last2digits = substr($year,-2,2);
if ($last2digits eq "00")
{
   $yesorno = ($year % 400 == 0 ? "certainly" : "not");
}
else
{
   $yesorno = ($year % 4 == 0 ? "certainly" : "not");
}
print("$year is " . $yesorno . " a leap year\n");
```

4 (b) Explain the following commands:
i) kill ii) cron iii) splice iv) crontab v) nice [05] CO6 L4

i) kill COMMAND:

- The kill command sends a signal to a process.
- The kill command uses one or more PIDs as its argument.
- By default, the kill command uses the SIGTERM (15) signal.
  $ kill 105
  is same as
  $ kill -s TERM 105
  The above command terminates or kills the job having PID 105.
- A user can kill only those processes he owns and can't kill processes of other users. Certain system processes having the PIDs 0, 1, 2, 3 and 4 can't be killed.

Example:
$ kill 121 122 125 132 138 144
Kills all processes having the PID 121, 122, 125, 132, 138 and 144.

Using kill with other signals (-s option):

By default, kill uses the SIGTERM signal (15) to terminate the process. Some programs simply ignore it and continue execution normally. In that case the process can be killed with the SIGKILL signal (9). This signal can't be generated at the press of a key. We must use kill with the signal name (without the SIG) preceded by the -s option.

$ kill -s KILL 121
or
$ kill -9 121

ii) cron

- **cron executes programs at regular intervals.**
- cron is mostly dormant, but every minute it wakes up and looks in a control file (the crontab file) in **/var/spool/cron/crontabs** for instructions to be performed at that instant.
- After executing them, it goes back to sleep, only to wake up the next minute.
- **A user can also place a crontab file named after his/her login name in the crontabs directory. For example, the user cmrit has to place crontab commands in the file /var/spool/cron/crontabs/cmrit**
- The specimen entry in the file cmrit is as shown

```
30 12 * * 1-6 clear; echo -e "\n\n\nIts lunch time.....\n\n\n" > /dev/pts/2
```

- so the following line indicates
  30 12 * * 1-6 clear; echo -e "\n\n\nIts lunch time.....\n\n\n" > /dev/pts/2
  From Monday to Saturday (1-6) at 12 hour 30 minutes in all months execute the commands - **clear; echo -e "\n\n\nIts lunch time.....\n\n\n"**

iii) splice Operator

➔ The splice function allows adding or removing items even from the middle of an array, allowing the array to grow or shrink as required. This effectively eliminates the need for linked lists in Perl.

➔ Splice function works with a maximum of four arguments. The syntax of the operator is shown below

splice (@array, $offset, [$length], [$list]);

➔ The first argument is the array on which the splice works and this argument must be present.

➔ The second argument is the offset from where the insertion or deletion begins.

- ➔ As shown in the syntax the $length and $list are optional arguments. If present, $length indicates the number of items or elements to be removed or inserted.

- ➔ When $list argument is present, splice replaces the items removed by the items present in the $list. If $list is not present, then nothing will be inserted.

- ➔ The offset value can be negative also. Whenever the $offset is negative, the counting starts from the end of the @array argument (from the high index value side) and proceeds backwards.

- ➔ The return value of splice function depends on the context in which it is used. If $list is specified, splice returns the elements removed from the @array. If $list is not specified, splice returns the last element removed.

- ➔ We can insert an element anywhere by making the value of the length argument 0 (zero).

Examples:

```
$perl -e '@subjects=qw/physics chemistry maths/;
$x=splice(@subjects,2,1,"computer"); print $x'
maths

$ perl -e '@subjects=qw/physics chemistry maths/;
$x=splice(@subjects,2,1,"computer"); print @subjects'
physicschemistrycomputer
```

To insert an element, length argument must be made zero -

```
$ perl -e '@subjects=qw/physics chemistry maths/;
$x=splice(@subjects,2,0,"computer"); print @subjects'
physicschemistrycomputermaths
```

iv) crontab

There are two ways of creating crontab file

FIRST METHOD OF CREATING crontab FILE:

Following are the steps to create a crontab file.

1. using vi editor, create a file name cron.txt (any name will do) with the following data

vi cron.txt

30 12 * * 1-6 clear; echo -e "\n\n\nIts lunch time.....\n\n\n" > /dev/pts/2
10 12 10 11 6 clear; echo -e "\n\n\nMESSAGE FROM CRONTAB.....\n\n\n" > /dev/pts/2

2. Place the file **cron.txt** in the directory /var/spool/cron/crontabs for cron to read the file. This is done by the **crontab** command.

$ crontab cron.txt

Note that the file cron.txt is placed in /var/spool/cron/crontabs directory with the name same as that of the user login name. In this example, the login name is 'cmrit'.

This way different users can have crontab files named after their user-ids.

SECOND METHOD OF CREATING crontab FILE:

The second way to enter cron commands is by using crontab with the -e option. crontab calls the editor defined in EDITOR variable. After editing the commands and quitting vi, the commands are automatically scheduled for execution.

$ crontab -e

Select an editor. To change later, run 'select-editor'.
1. /bin/ed
2. /bin/nano <---- easiest
3. /usr/bin/vim.tiny

Choose 1-3 [2]: 3
selecting the 3 option (vi editor) edit, save and quit vi. All the commands are scheduled for execution.

To see the contents of crontab file:
The -l otption is used to see the contents of the crontab file.
$ crontab -l

To remove the crontab file:
The -r option removes the crontab file.
$ crontab -r

v) nice command:

- The nice command runs a program with modified scheduling priority.
- Nice reduces the priority of any process there by raising its nice value.
- The nice command is used with the & operator to reduce the priority of jobs.

- By doing so, more important jobs can then have greater access to the system resources (**being "nice" to your neighbours)**.
- To run a job with low priority, the command name should be prefixed with nice.
  $ nice wc -l uxmanual
  or
  $ nice wc -l uxmanual &

- With no COMMAND, nice prints the current niceness.
  $ nice
  0
- Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process).
- A higher nice value implies a lower priority.
- We can specify the nice value explicitly with the -n option
  $ nice -n 5 wc -l uxmanual &
- In the above command the nice value is increased by 5 units.
- A non privileged user can't increase the priority of a process, only a superuser can do.

5 (a) Explain the two special files /dev/null and /dev/tty. [04] CO6 L4

/dev/null:
Sometimes the user wants to check whether program runs successfully without seeing its output on the screen. Also, the user doesn't want to save the output in a file.

In that situation, there is a special file that simply accepts any stream without growing in size – this special file is called **/dev/null.**

$cmp file1 file2 2> /dev/null
$cat /dev/null

The size of the file /dev/null always remains zero. /dev/null simply incinerates or completely destroys all output written to it. Whether we direct or append output to this file, its size always remains zero.

This facility is useful in redirecting error messages away from the terminal so that they don't appear on the screen.

/dev/null is actually a pseudo-device because it is not associated with any physical device.

/dev/tty:

The second special file in the UNIX system is the one indicating the terminal **/dev/tty**. This is not the file which represents standard output or standard error. Commands generally don't write to this file, but the user can redirect some statements in shell script to this file.

If user1 is working on terminal **/dev/pts/1** and user2 on **/dev/pts/2**, then both can refer to their own terminals with the same filename - /dev/tty. If user1 issues the command

$who > /dev/tty

the list of current users is sent to the terminal user1 is currently using that is, /dev/pts/1.

Similarly, if user2 issues the same command, the output is sent to user2 terminal that is, /dev/pts2.

Like /dev/null, /dev/tty can be accessed independently by several users without any conflict.

Redirecting output to one's own terminal:

If we redirect a shell script to a file, say by using
foo.sh > foo.out
Redirecting a script implies redirecting the standard output of all statements in the script. But, the script may contain some **echo** commands that provide helpful messages for the user and the user wants to see them on terminal. If such statements are explicitly redirected to /dev/tty inside the script, redirecting the script won't affect these statements. All echo commands whose output is redirected to /dev/tty will display the output on the terminal even though the output of shell script is directed to a file.

5 (b) Write a PERL program to find the number of characters, words in a given sentence as well as to print the sentence in reverse order. [06] CO5 L1

```
#!/usr/bin/perl
# Program to find the number of characters, words as well
# as to print the Reverse of a given sentence.
#
print "Enter a sentence :";
chomp($in=<STDIN>);

$no_chars = length $in;
print "The number of characters in the sentence is $no_chars\n";
```

```
@sent=split(" ",$in);
$words=@sent;
print "The number of words in the sentence is $words\n";

@reverse_sent= reverse @sent;
print "The reversed string is @reverse_sent\n";
```

6 (a) Explain the following with example:
1. head 2. tail 3. cut 4. paste 5. sort [05] CO4 L4

head command:
The **head** command displays the top (first part) of the file. When used without any option, it displays the **first ten lines** of the file.

Example 1: The command **head** by default displays first 10 line of the file
$ head sales

OPTION -n:
We can use the -n option to specify the line count and display first n lines of the file

Example 2: To display first 3 lines of the file using -n option
$ head -n 3 sales

tail command:
The **tail** command displays the end (last part) of the file. By default, it displays the **last ten lines** of the file.

Example1: tail command by default displays last 10 lines of the file
$ tail sales

OPTION -n:
Example2: To display last 3 lines of the file using -n option
$ tail -n 3 sales

OPTION -n +count:
The tail command can also display lines from the beginning of the file instead of the end. This is done using the -n +count option, where count represents the line number from where the command must start to display.

Example: To display from line 5 to end of the file
$ tail -n +5 sales

With -c option, the tail command displays in terms of characters or bytes rather than lines.

The cut command:
The cut command removes sections from each line of files. **It slits a file vertically.**

Cutting columns: (-c):
To extract specific columns, we need to specify the -c option with a list of column numbers, delimited by a comma. Ranges can also be used using the hyphen. **The -c option is useful for fixed-length lines.**

Example 1: To extract name and designation from file shortlist

```
$ cut -c 6-22,24-32 shortlist
a.k. shukla          |g.m.
jai sharma           |director
sumit chakrabarty |d.g.m.
barun sengupta     |director
n.k. gupta           |chairman
```

Note: There should be no whitespace in the column list (6-22,24-32). The cut command selects the column from the beginning and up to the end of a line. The column list must be an ascending list.

Example 2: To cut the second and the third field of shortlist

```
$ cut -d \| -f 2,3 shortlist
                or
$ cut -d "|"  -f  2,3  shortlist
```

In the above command the | (pipe) is escaped to prevent the shell from interpreting it as the pipeline character. We can also use -d "|".

THE paste COMMAND:

The paste command merges lines of files. **Its pastes files vertically rather than horizontally.** We can view two files side by side by pasting them.

Example 1: To paste files cutlist1 and cutlist2

```
$ paste cutlist1 cutlist2
a.k. shukla          |g.m.          2233|sales       |12/12/52|6000
jai sharma           |director      9876|production|03/12/50|7000
sumit chakrabarty |d.g.m.        5678|marketing |04/09/43|6000
barun sengupta     |director      2365|personnel |05/11/47|7800
```

n.k. gupta |chairman 5423|admin |08/30/56|5400

The pasting has taken place on whitespace. Like cut command, the **paste command also uses the tab as the default delimiter**, but the user can specify one or more delimiters with -d option.

Example 2: paste cutlist1 and cutlist2 with | (pipe) as delimiter
$ paste -d "|" cutlist1 cutlist2
a.k. shukla |g.m. |2233|sales |12/12/52|6000
jai sharma |director |9876|production|03/12/50|7000
sumit chakrabarty |d.g.m. |5678|marketing |04/09/43|6000
barun sengupta |director |2365|personnel |05/11/47|7800
n.k. gupta |chairman |5423|admin |08/30/56|5400

THE sort COMMAND:

The sort command sorts lines of text files. Like cut command, the sort command can sort on specified fields.

Syntax:
sort [OPTION]... [FILE]...

Example 1:
$ sort shortlist
2233|a.k. shukla |g.m. |sales |12/12/52|6000
2365|barun sengupta |director |personnel |05/11/47|7800
5423|n.k. gupta |chairman |admin |08/30/56|5400
5678|sumit chakrabarty |d.g.m. |marketing |04/09/43|6000
9876|jai sharma |director |production|03/12/50|7000

By default, sort reorders lines in ASCII collating sequence – whitespace first, then numerals, uppercase letters and finally lowercase letters.

The default sorting sequence can be altered by using certain options. We can also sort on one or more keys (fields) or use a different ordering rule.

OPTIONS OF sort COMMAND:

sort uses one or more contiguous spaces as the default field separator.

Sl. No.	OPTION	DESCRIPTION
1	-t*char*	Uses delimiter *char* to identify fields

2	-k *n*	Sorts on *n*th field
3	-k *m,n*	Starts sort o *m*th field and ends on *n*th filed
4	-k *m.n*	Starts sort on *n*th column of *m*th field
5	-u	Removes repeated lines
6	-n	Sorts numerically
7	-r	Reverses sort order
8	-f	Folds lowercase to equivalent uppercase (case-insensitive sort)
9	-m *list*	Merges sorted file in *list*
10	-c	Checks if file is sorted
11	-o *flname*	Places output in file *flname*

6 (b) Write a PERL program to print numbers that are accepted from keyboard using while and array construct. [05] CO5 L1

```
# Write a PERL program to print numbers that are
# accepted from keyboard using while and array construct.

print ("Enter numbers:");
while (<>)
{
      push(@x,$_);
}

print ("\nThe numbers entered are \n");
print @x;
```

7 (a) Explain the following string handling functions in PERL
i) length ii) index iii) substr iv) reverse v) push [06] CO5 L4

(i) length
Returns the length of the string in bytes.

```
$perl -e '$x=abcdefgh" ; print length($x);'
8
```

(ii) index
This function returns the position of the first occurrence of the spcicfied SEARCH string. If POSITION is specified, the occurrence at or after the position is returned. The value -1 is returned if the SEARCH string is not found.

$perl -e ‘$x=”abcdijklm”; print index($x,j);’
5

$perl -e ‘$x=abcdefghijklm”; print index($x,j,2);’
9

(iii) substr

This function supports three sets of passed values as follows

a) substr(STRING, OFFSET)

This function returns all characters in the string after the designated offset from the start of the STRING.

$ perl -e '$x="Hello World"; print substr($x,6);'
World

b) substr(STRING, OFFSET, LEN)

This function returns all characters in the STRING designated after the OFFSET from the start of the STRING up to the number of characters designated by LEN.

perl -e '$x="Hello World"; print substr($x,1,7);'
ello Wo

$ perl -e '$x="Cat is in the hat"; print substr($x,-3,3);'
hat

$ perl -e '$x="Cat is in the hat"; print substr($x,-7,3);'
the

c) substr(STRING, OFFSET, LEN, REPLACEMENT)

This replaces the part of the string beginning at OFFSET of the length LEN with the REPLACEMENT string.

$ perl -e '$x="Hello World"; substr($x,6,7,"SAVITHA"); print("\n".$x."\n");'

Hello SAVITHA

$ perl -e '$x="Cat is in the hat"; substr($x,0,3,"Rat"); print("\n".$x."\n");'

Rat is in the hat

$ perl -e '$x="Cat is in the hat"; substr($x,14,3,"Box"); print("\n".$x."\n");'

Cat is in the Box

iv) reverse
Reverses the characters in str and returns the reversed string.

$ perl -e ' $x="CMRIT"; $y=reverse($x); print($y);'
TIRMC

v) push Operator

The push operator works with two arguments, the first argument is an array variable name and the second is the element to be pushed.

$ perl -e '@x=(1,2,3,4); **push(@x,5)**; print @x;'
12345

7 (b) Explain the default variable $_ and $. [04] CO5 L4

The Dollar-Underscore ($_) Variable – Current Line

- This is one of the most commonly used Perl's special variables.
- This special variable always holds the current line.
- Many functions use $_ as a default argument when no argument is mentioned explicitly. For example, the print function normally expects some variable, or a list of variables or a string, the value of which is expected to be printed. However, if no argument is provided, the print function prints the value of the default variable, $_.

Examples:

1.
$ perl -e ' $_="the cat is on the Mat\n"; print;'
the cat is on the Mat

2.
$ perl -e '
foreach (1..5)
{
$sum=$sum+$_;
print "The step number is $_ and the sum is $sum\n";
> }'
The step number is 1 and the sum is 1

The step number is 2 and the sum is 3
The step number is 3 and the sum is 6
The step number is 4 and the sum is 10
The step number is 5 and the sum is 15

The Dollar-dot ($.) Variable – Current Line Number

Pearl stores the current line number in a special variable, $. ($ followed by a dot). Generally, line numbers are used as line addresses to select required lines from anywhere in a file.

Examples:
1.
$ perl -ne 'print if**($.<4)**' first_script.pl #like head -n 3
output: prints the first three lines of file – first_script.pl

2.
$ perl -ne 'print if**($.>7 && $.<9)**' first_script.pl
output: print the eighth line of the file – first_script.pl

8 (a) Differentiate between hard link and soft link.. [05] CO4 L1

The below table shows the difference between hard link and soft link:

Sl. No.	Hard Link	Soft /Symbolic Link
1.	A hard link is an additional name of the original file which refers inode to access the target file. It is an alias to the original file.	A file that can be accessed through different references pointing to it is known as soft link or symbolic link. Soft link simply provides the pathname of the file that actually has the contents.
2.	When the original file is deleted, link file can still be accessed	When the original file is deleted soft link becomes invalid and the file cannot be accessed as the symbolic link file would point to a nonexistent file and becomes a **dangling symbolic link**.
3.	The command to create hard link: ln <filename> <newfilename>	The command to create soft link; ln -s <filename> <newfilename>
4.	When two files are hard linked the two files will have the same inode number.	When two files are soft linked a new file of type symbolic link is created with a separate inode number.

5.	When two files are hard linked the link count of the two files will be two	The link count of the two files will be one.
6.	In hard link the newly created link file will not occupy any space in hard disk	In soft link the newly created symbolic link file occupies space on disk and stores the pathname of the original file. The size of symbolic link is equal to the length of original file name.
7.	Hard links are restricted to its own partitions.	Can be linked to any other file system even network file system.
8.	Memory consumption is less	Memory consumption is more
9.	Hard links cannot be implemented on directories	We can link directories using soft link
10.	The relative path is not allowed in hard link.	Relative path and absolute path both are allowed in soft link.

8 (b) Explain with example set and shift commands in UNIX to manipulate positional parameters. [05] CO4 L4

(i) set command:

- The set command assigns its arguments to the positional parameters $1, $2 and so on.
- It also sets the other parameters $# and $*.
- This feature is useful for picking up individual fields from the output of a program.

Example 1:
$ set 9876 2345 6213

The above command assigns the value 9876 to the positional parameter $1, 2345 to $2 and 6213 to $3.

$ echo $1 $2 $3
9876 2345 6213

$ echo "The $# arguments are $*"
The 3 arguments are 9876 2345 6213

Example 2: To use command substitution to extract individual fields from date output:
$ set `date`

$ echo $*

or

$ echo $@
Tue Oct 23 01:10:20 IST 2018

$1 is set to **Tue**
$2 is set to **Oct**
$3 is set to **23**
$4 is set to **01:10:20**
$5 is set to **IST**
$6 is set to **2018**

$ echo "Today's date is $3 $2 $6"
Today's date is 23 Oct 2018

(ii) shift command:

- **shift** command transfers the contents of a positional parameter to its immediate lower numbered one.
- This is done as many times as the shift command is called. When called once, $2 becomes $1, $3 becomes $2 and so on.
- The contents of the leftmost parameter $1, is lost every time shift is invoked.
- We can access 10^{th} argument by first using the shift command once and then using $9.
- If the script uses 12 arguments, we can shift 3 times and then use the $9 parameter to access 12^{th} argument.
- Every time we use shift, the leftmost variable gets lost; so it should be saved in a variable before using shift. If we have to start iteration from the fourth argument, then, save the first three parameters and then use shift 3.

Example:

$ set `date`

$ echo $@
Tue Oct 23 01:32:41 IST 2018

$ echo **$1** $2 $3
Tue Oct 23

$ shift
$ echo **$1** $2 $3
Oct 23 01:32:41

$ shift 2
$ echo $1 $2 $3

01:32:41 IST 2018