1. Write shell commands to do the following:

   (a) Count the number of plain files in a system

   ```
   ls / | wc -l
   ```

   (b) Delete all the files in the current directory with the substring ”sand”. (In the file contents)

   ```
   #!/bin/sh

   sand=$(grep -l "sand" *)
   for file in $(echo "$sand"); do
       rm "$file"
   done
   ```

   (c) Assuming that the names of all files in current directory which start with ‘f’ are of the form ‘f’ followed by one or more digits, rename each of them to be one more than their current name. For example, “f3” would be renamed to “f4”, and “f9” would be renamed to “f10”

   ```
   # Gets all files in current directory with f plus one or more digits
   files=$(find * -type f -regex "f[0-9][0-9]*")

   for file in $(echo "$files"); do
       # Gets number from file after "f"
       num=$(echo "$file" | cut -d "f" -f 2)
       num=$(expr $num + 1)
       mv "$file" "f$num"
   done
   ```

2. Problem: The current directory contains a bunch of plain files whose names begin with a digit, and a bunch of plain files whose names do not begin with a digit. Write shell commands to output all of the words which contain a 'q', only from all of the files whose names begin with a digit. Each word appears only once in the output no matter how many times it appears in the file(s).

```
notdigit=$(find * -not -name "[0-9][0-9]*")

for file in $(echo "$notdigit"); do
    grep -w ".*q.*" "$file" | uniq
done
```

3. (a) Write a shell script which reads zero or more integers from the standard input, one per line, and outputs their sum. You can assume that the input is correctly-formed. Example:

`(echo 2; echo 3; echo 4) | sh yourfile`

will output 9.

```
sum=0

while read num; do
    sum=$(expr $sum + $num)
done
echo $sum
```

(b) Write a shell script which takes zero or more integers as command-line arguments and outputs their sum. You can assume that the command-line arguments are all valid integers. Example:

`sh yourfile 2 3 4`

will output 9.

```
sum=0

while [ $# -gt 0 ]; do
    sum=$(expr $sum + $1)
    shift
done
echo $sum
```

4. There are three different times in an inode in the unix filesystem: the mtime (last-modified time), the atime (last-accessed time) , and the ctime (last inode change time) . Suppose that there is a plain file named ..abc" in the current directory:

   (a) Write a shell command which will update the mtime of "abc" (i.e. set it to the current time).

   ```
   echo "" >> abc # Adds newline to abc
   ```

   (b) Write a shell command which will update the atime of "abc", but not change its mtime or ctime.

   ```
   cat abc # Prints content but does not modify or change
   ```

   (c) Write a shell command which will update the ctime of "abc", but not change its mtime or atime.

   ```
   chmod a+r abc # Changes permission of file, but does not modify or access
   ```

   (d) Write a shell command which will update the mtime of the current directory.

   ```
   touch xyz | rm xyz # Adds and deletes a file from cd, thus modifying it
   ```

   (e) Write a shell command which will update the atime of the current directory.

   ```
   ls # Accesses directory
   ```

   (f) Write a shell command which will update the ctime of the current directory.

   ```
   chmod o-x . # Changes permissions of directory
   ```

5. This is normally an impossible situation:

```c
void f(int *a)
{
  // Do something with elements of the array ‘a’
}
```

because normally we would need an additional parameter which is the size of the array. However in the calls,

```c
strcpy(a, b)
```

and

```c
pipe(f)
```

there is no size parameter, How do these functions know how big their arrays are? There are three subquestions:

(a) strcpy’s first argument (the target of the copying)
strcpy will modify the array of the first argument so that it matches the characters of the string to be copied, and then adds a null terminator as the next element in array.

(b) strcpy’s second argument (the string to be copied)
This string must be a null-terminated string. This means that strcpy considers every character in the character array pointed to by the first argument until ‘\0’ to be the string being copied. strcpy only works successfully on null-terminated strings.

(c) pipe’s argument
pipe’s man page specifies that an int array of size 2 should be used as its argument.

(d) Give an example of an additional, distinct situation where a function similar to ‘f’ above knows how many elements there are in the array (or at least, how many elements to pay attention to) without a separate size parameter.
Any function that iterates through a linked list. (Functions that dealt with an array was wanted, but I drew a blank)

6. Write a complete C program (except that you can omit the #includes) which outputs all of its command-line arguments in order, one per line, but with each string reversed. Example session, where '$' is the shell prompt:

```
$ ./a.out once upon a time
ecno
nopu
a
emit
$
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void reverse(char *buf, char *arg) {
    int index = strlen(arg);
    for (int i = 0; i<index; i++){
        buf[i] = arg[(index-1) - i];
    }
}

int main(int argc, char **argv)
{
    char buf[256];
    for (int i = 1; i<argc; i++) {
        strcpy(buf, argv[i]);
        reverse(buf, argv[i]);
        printf("%s\n", buf);
    }
}
```

7. Without using popen(), write a complete C program to exec /bin/date with its standard output redirected into a pipe. Your program will read from this pipe and output just the fifth character of date's output, and a newline. (Do not use a temporary file; use a pipe.)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int fd[2];

    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }
    switch (fork())
    {
        case -1:
            perror("fork");
            exit(1);
            break;
        case 0: // Child process runs "/bin/date"
            // Close reading end of pipe
            if (close(fd[0]) == -1) {
                perror("close");
                exit(1);
            }
            // Set stdout so that writing to stdout goes to pipe
            if (dup2(fd[1], STDOUT_FILENO) == -1) {
                perror("dup2");
                exit(1);
            }
            // Close duplicate fd of writing end of pipe
            if (close(fd[1]) == -1) {
                perror("close");
                exit(1);
            }
            execl("/bin/date", "/bin/date", (char *) NULL);
            fprintf(stderr, "ERROR: exec should not return \n"); // This line is only
    reached when exec fails somehow
            break;
        default:
            // Closes writing end because process never writes to pipe
            if (close(fd[1]) == -1) {
                perror("close");
                exit(1);
            }
            char buf[5];
            // Reads 5 bytes from pipe
            if (read(fd[0], buf, 5) == -1) {
                perror("read");
                exit(1);
            }
            printf("%c\n", buf[4]);
            exit(0);
            break;
    }
    return 0;
}
```

8. The following program is a server which reads one byte (character) from each client, echoes it back, and drops the connection.

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main()
{
    int fd, clientfd;
    socklen_t len;
    struct sockaddr_in r, q;
    char c;

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        return(1);
    }

    memset(&r, '\0', sizeof r);
    r.sin_family = AF_INET;
    r.sin_addr.s_addr = INADDR_ANY;
    r.sin_port = htons(2000);

    if (bind(fd, (struct sockaddr *)&r, sizeof r) < 0) {
        perror("bind");
        return(1);
    }
    if (listen(fd, 5)) {
        perror("listen");
        return(1);
    }

    while (1) {
        len = sizeof q;
        if ((clientfd = accept(fd, (struct sockaddr *)&q, &len)) < 0) {
            perror ("accept" );
            return(1);
        }
        switch (read(clientfd, &c, 1)) {
            case -1:
                perror("read");
                return(1);
            case 1:
                if (write(clientfd, &c, 1) != 1)
                    perror("write");
            }
            close(clientfd);
    }
}
```

(a) Regarding the line "r.sin_port = htons(2000);" (line 23), what would happen if instead we simply wrote "r.sin_port = 2000;"?
The network protocols used need to have the port number stored in network order, which htons() does for us, if 2000 was used alone, the port number would be stored in host order which would cause networking problems.

(b) On which line number will this program be blocked when no client is connected?
29 (I think)

(c) On which line number will this program be blocked when a client is connected but has not yet transmitted its byte?
40

(d) In the switch statement beginning on line 40, under what circumstances will neither of the cases match? What will read()'s return value be, and why?
When connection is broken from client's side and no data is available, read will return 0 and neither of the cases will match.

(e) Change the program in place so that it mostly functions as it currently does, except that if the byte transmitted from the client is a control-B (ASCII value 2), instead of sending the control-B back it sends the string "ha!" plus a network newline.

```c
// Changed code beginning at line 14

    while (1) {
        len = sizeof q;
        if ((clientfd = accept(fd, (struct sockaddr *)&q, &len)) < 0) {
            perror ("accept" );
            return(1);
        }
        switch (read(clientfd, &c, 1)) {
            case -1:
                perror("read");
                return(1);
            case 1:
                if (c == 2) {
                    char *ha = "ha!\n"
                    if (write(clientfd, ha, 4) != 4) {
                        perror("write");
                    }
                }
                if ((c != 2) && (write(clientfd, &c, 1) != 1))
                    perror("write");
            }
            close(clientfd);
    }
}
```

9. Write a complete C program (except that you can omit the #includes) which scans the current directory, and for each item in the current directory which is itself a directory, it forks and execs 'ls' on that subdirectory. For example, if the current directory contains subdirectories 'a' and 'b', and plain files 'c' and 'd', your program will execute the commands "ls a" and "ls b". (Note: For this exam question you must exec /bin/ls; you can't just perform its function yourself.)

```c
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    DIR *directory = opendir(".");
    if (directory == NULL) {
        fprintf(stderr, "Directory could not be opened\n");
        return 1;
    }
    struct dirent *entry;
    entry = readdir(directory);
    while (entry != NULL) {
        if ((entry->d_type == DT_DIR) && (strcmp(entry->d_name, ".") != 0) && (strcmp(entry->d_name, "..") != 0)){
            switch (fork()) {
                case -1:
                {
                    perror("fork");
                    exit(1);
                    break;
                }
                case 0:
                {
                    execlp("ls", "ls", entry->d_name, (char *) NULL);
                    fprintf(stderr, "ERROR: exec should not return \n"); // This line is only reached when exec fails somehow
                    break;
                }
                default:
                {
                    int s;
                    // Waits for child to finish
                    wait(&s);
                    break;
                }
            }
        }
        entry = readdir(directory);
    }
    return 0;
}
```

Bonus qn There is a network protocol in which strings are terminated with the zero byte, just like in C. A C programmer noticed that the string is already formatted in memory with the terminating zero byte, so they decided to do something like

```
write(serverfd, s, strlen(s) + 1);
```

thus writing not only the data bytes of the string, but also the terminating zero, as is expected in this particular protocol. Unfortunately, they made a parenthesization mistake and instead wrote:

```
write(serverfd, s, strlen(s + 1));
```

If ‘s’ was the string “good morning”, exactly what was written to the socket?

“strlen(s+1)” would give the length of the string starting from ‘o’ in “good morning”, i.e. 1 less than the length of the string “good morning”, therefore “good mornin” would be written to the socket.