1. (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.
Examples, if the script is in a file named "add", where the '$' is the shell prompt:

```
$ (echo 2; echo 3; echo 6) | sh add
11
$ echo 2 | sh add
2
$ sh add </dev/null
0
```

```
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 all command-line arguments are valid integers.
Examples:

```
$ sh add 2 3 6
11
$ sh add 2
2
$ sh add
0
```

```
sum=0

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

2. Write a complete shell script which behaves as follows. It must be run with at least one argument. Arguments are considered to be plain files. Appropriate messages should be printed for files which can't be read (by using unix tools appropriately so as to generate messages).
   The output of your shell script is the size in bytes of the largest of the files, with the file name. If there is a tie for largest, output any one of the files.
   Example session, if your script is in a file named "largest":

```
$ sh largest file1 file2 file3
14680 file1
$
```

```
max=0
maxname=

while [ $# -gt 0 ]; do
    # that stat format is what works on my mac, "stat --format="%s" $1" works on mathlab
    if [ $max -le $(stat -f "%z" $1) ] ; then
        max=$(stat -f "%z" $1)
        maxname=$1
    fi
    shift
done

echo "$max $maxname"
```

3. How many groups are you in? The output of the "id" command looks like this (all on one line):
   `uid=123(leephy98) gid=1008(cstudent) groups=1008(cstudent),517(csc209h),525(csc263h)`

```
  id | cut -d " " -f 3 | tr "," "\n" | wc -l
```

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).

```
1 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.

```
1 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.

```
1 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.

```
1 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.

```
1 ls # Accesses directory
```

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

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

5. Write a partial version of the *test* command in C. Your program will implement only the following features of *test*:

```
test -f file
test -d file
test -s file
```

"test -f file" tests whether the file exists and is a plain file,
"test -d file" tests whether the file exists and is a directory,
"test -s file" tests whether the file exists and is a plain file AND is non-zero-sized.

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <string.h>

int main(int argc, char *argv[])
{
    struct stat filestat;
    if (argc != 3) {
        printf("Incorrect number of arguments\n");
        return -1;
    }
    if (stat(argv[2], &filestat) == -1) {
        perror("stat");
        return -1;
    }
    if (strcmp(argv[1], "-f") == 0) {
        (S_ISREG(filestat.st_mode)) ? (printf("File exists and is a plain file\n")) :
    printf("File exists but is not a plain file\n");
    }
    else if (strcmp(argv[1], "-d") == 0) {
        (S_ISDIR(filestat.st_mode)) ? (printf("File exists and is a directory\n")) :
    printf("File exists but is not a directory\n");
    }
    else if (strcmp(argv[1], "-s") == 0) {
        ((S_ISREG(filestat.st_mode)) && ((int)filestat.st_size > 0)) ? (printf("File
    exists, is a plain file and non-zero sized\n")) : printf("File exists but is not a
    plain file and non-zero sized\n");
    }
    return 0;
}
```

6. Write a modified and partial version of the *tr* command in C. Your program requires at least two arguments (i.e. argc≥3), where invocations will look like this:

```
tr e f
tr e f file1 file2
tr 0123456789 X file
```

The first argument is a string of one or more characters to translate. All of these characters are translated to the single character which is the second argument. For this exam question, you do not need to check that argv[2] is of length 1 (although you do need to check that it exists, by checking argc).

After these two mandatory arguments, your program takes zero or more files in the usual way, where zero filename arguments means to read the standard input. As always, the output is to the standard output.

You may want to use "strchr()" to check whether a character occurs in a string.

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

#define MAX_SIZE 256

void translate(char *tr_set, char *replace, FILE *input)
{
    char buf[MAX_SIZE];
    while (fgets(buf, MAX_SIZE, input) != NULL) {
        for (int i = 0; i<strlen(buf); i++) {
            if (strchr(tr_set, buf[i]) != NULL) {   // Char exists in set, so must replace
                printf("%c", replace[0]);
            }
            else {
                printf("%c", buf[i]);
            }
        }
    }
}

int main(int argc, char *argv[])
{
    if (argc < 3) {
        fprintf(stderr, "Not enough arguments\n");
        return -1;
    }
    if (argc == 3) {
        translate(argv[1], argv[2], stdin);
    }
    else {
        for (int i = 3; i<argc; i++) {
            FILE *input = fopen(argv[i], "r");
            translate(argv[1], argv[2], input);
            if (fclose(input) == -1) {
                perror("fclose");
                return -1;
            }
        }
    }
}
```

7. Write a C program which recursively traverses the entire filesystem, starting from the root directory. It adds together the sizes, in bytes, of all files in the filesystem, including special files (including directories), and outputs this number. (After a successful stat() or lstat() call, the size of the file in bytes is present in the struct member "st_size".) You can assume that the total can be represented in an int.

   If a complete file path name reaches 1000 characters or more in length, your program may terminate with an appropriate error message; but it does have to check, and it may not exceed array bounds.

```c
#define _LARGEFILE64_SOURCE
#define _FILE_OFFSET_BITS 64

#include <stdlib.h>
#include <unistd.h>
#include <ftw.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#ifndef USE_FDS
#define USE_FDS 15
#endif

int sum = 0;

int list_entry(const char *filepath, const struct stat *info,
               const int typeflag, struct FTW *pathinfo)
{
    sum += info->st_size;
    return 0;
}


int list_directory_tree(const char *const dirpath)
{
    int result;

    /* Invalid directory path? */
    if (dirpath == NULL || *dirpath == '\0')
        return errno = EINVAL;

    result = nftw(dirpath, list_entry, USE_FDS, FTW_PHYS);
    if (result >= 0)
        errno = result;

    return errno;
}

int main(int argc, char *argv[])
{
    if (list_directory_tree("/")) {
        fprintf(stderr, "%s.\n", strerror(errno));
        return EXIT_FAILURE;
    }
    fprintf(stderr, "%d\n", sum);
    return EXIT_SUCCESS;
}
```

8. There is a computation which takes a long time to run, but is broken up into five roughly equal parts which can run simultaneously.

   Write a function in C (not a complete program; no main()) which forks five times; each of the child processes runs compute(n) for a different $0 \geq n \geq 5$; and your function wait()s for all five child processes before returning.

   (There's no exec(); all of this happens within one program.)

   You don't have to keep track of process ID numbers; you can assume for this question that there are no other child processes, so a wait() call will always wait for one of the five processes you have spawned.

```c
int fork_fn()
{
    for (int i = 0; i < 5; i++) {
        if (fork() == 0) {
            compute(i);
        }
    }
    for (int i = 0; i < 5; i++){
        // wait(NULL), waits for any child process. In for loop to wait 5 times for each child.
        wait(NULL);
    }
    return 0;
}
```

9. The program /local/t3 writes some data to file descriptor 7. Write a complete C program which execs /local/t3 with its file descriptor 7 redirected to the file "bar" (in the current directory) (the standard file descriptors 0, 1, and 2 are left alone). (All appropriate error checking is required.)

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int bar;
    if ((bar = open("bar", O_WRONLY | O_CLOEXEC | O_CREAT, 00666)) == -1) {
        perror("open");
        return -1;
    }
    int status;
    switch (fork())
    {
        case -1:
            perror("fork");
            exit(1);
            break;
        case 0:
            dup2(bar, 7);
            close(bar);
            execl("/local/t3", "/local/t3", (char *) NULL);
            break;
        default:
            wait(&status);
            exit(0);
            break;
    }
    return 0;
}
```

10. 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?
36

(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.

```
1  // Changed code beginning at line 14
2
3      while (1) {
4          len = sizeof q;
5          if ((clientfd = accept(fd, (struct sockaddr *)&q, &len)) < 0) {
6              perror ("accept" );
7              return(1);
8          }
9          switch (read(clientfd, &c, 1)) {
10             case -1:
11                 perror("read");
12                 return(1);
13             case 1:
14                 if (c == 2) {
15                     char *ha = "ha!\n"
16                     if (write(clientfd, ha, 4) != 4) {
17                         perror("write");
18                     }
19                 }
20                 if ((c != 2) && (write(clientfd, &c, 1) != 1))
21                     perror("write");
22             }
23             close(clientfd);
24     }
25 }
```