20T3 Final Exam Questions

Exam Conditions

- You can start reading this exam at **Saturday 28 November 12:50** Sydney time.
- You can start typing at **Saturday 28 November 13:00** Sydney time.
- You have until **Saturday 28 November 16:00** Sydney time to complete this exam
- Only submissions before **Saturday 28 November 16:00** Sydney time will be marked
- Except, students with extra exam time approved by Equitable Learning Services (ELS) can make submissions after **Saturday 28 November 16:00** within their approved extra time
- You are not permitted to communicate (email, phone, message, talk, ...) with anyone during this exam, except COMP1521 staff via **cs1521.exam@cse.unsw.edu.au**
- You are not permitted to get help from anyone but COMP1521 staff during this exam.
- This is a closed book exam.
- You are not permitted to access papers or books.
- You are not permitted to access files on your computer or other computers, except the files for the exam.
- You are not permitted to access web pages or other internet resources, except the web pages for the exam and the online language cheatsheets & documentation linked below
- Even after you finish the exam, do not communicate your exam answers to anyone or the day of the exam. Some students have extended time to complete the exam.
- Do not place your exam work in any location, including file sharing services such as Dropbox, accessible to any other person .
- Ensure during the exam no other person in your household can access your work.
- Your **zpass** should not be disclosed to any other person. If you have disclosed your **zpass**, you should change it immediately.
- **Deliberate violation of exam conditions will be referred to Student Integrity as serious misconduct**

Exam Structure

- There are **11** questions on this exam.
- Total mark of questions on this exam is **100**.
- Questions are **NOT** worth equal marks.
- All 11 questions are practical (programming) questions.
- Not all questions may have provided files, You should create any files needed for submission if they are not provided.
- Answer each question in a **SEPARATE** file. Each question specifies the name of the file to use. These are named after the corresponding question number, Make sure you use **EXACTLY** this file name.
- When you finish working on a question, submit the files using the **give** command provided in the question. **You may submit your answers as many times as you like.** The last submission **ONLY** will be marked.
- Do not leave it to the deadline to submit your answers. Submit each question when you finish working on it. Running autotests does not automatically submit your code.
- You can verify what submissions you have made with `1521 classrun -check final_q<N>`

Language Documentation

You may access this **language documentation** while attempting this test:

- C quick reference
- MIPS Quick Reference Card
- MIPS Instruction Reference
- SPIM Documentation
- MIPS Quick Tutorial

You may also access:

- manual entries via the `man` command
- Texinfo pages via the `info` command

Special Considerations

This exam is covered by the Fit-to-Sit policy. That means that by sitting this exam, you are declaring yourself well enough to do so. You will be unable to apply for special consideration after the exam for circumstances affecting you before it began. If you have questions, or you feel unable to complete the exam, contact **cs1521.exam@cse.unsw.edu.au**

If you experience a technical issue before or during the exam, you should follow the following instructions:

Take screenshots of as many of the following as possible:

- error messages
- screen not loading
- timestamped speed tests
- power outage maps
- messages or information from your internet provider regarding the issues experienced

You should then get in touch with course staff via **cs1521.exam@cse.unsw.edu.au** as soon as the issue arises

Getting Started

Set up for the exam by creating a new directory called `exam_final`, changing to this directory, and fetching the provided code by running these commands:

```
$ mkdir -m 700 exam_final
$ cd exam_final
$ 1521 fetch exam_final
```

Or you can download the provided code as a zip file or a tar file.

If you make a mistake and need a new copy of a particular file you can do the follow:

```
$ rm broken-file
$ 1521 fetch exam_final
```

Only files that don't exist will be recreated, all other files will remain untouched

Question 1 (10 MARKS)

You have been given `final_q1.s`, a MIPS assembler program that reads *one* number and then prints it.

Add code to `final_q1.s` to make it equivalent to this C program:

```
// COMP1521 20T3 final exam Q1 C reference

// print (x + y) * (x - y)

#include <stdio.h>

int main(void) {
    int x, y;

    scanf("%d", &x);
    scanf("%d", &y);
    printf("%d\n", (x + y) * (x - y));

    return 0;
}
```

In other words, it should read *two* numbers, **x** and **y**, and print **(x + y) * (x - y)**.

For example:

```
$ 1521 spim -f final_q1.s
5
8
-39
$ 1521 spim -f final_q1.s
6
5
11
$ 1521 spim -f final_q1.s
5
6
-11
$ 1521 spim -f final_q1.s
42
42
0
```

NOTE:

No error checking is required.

Your program **can** assume its input always contains *two* integers, and only *two* integers.

Your program **should** print one and only one line of output.

This line of output **should** contain one integer, a newline, and nothing else.

You **can** assume the value of the expression can be represented as a signed 32 bit value.
In other words, you **can** assume overflow/underflow does not occur.

Your solution **must** be in MIPS assembler only.

Your solution **must** be in `final_q1.s`.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q1
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q1 final_q1.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q1
```

Question 2 (9 MARKS)

You have been given `final_q2.c`, a stub of a C program.

Your task is to add code to this function in **final_q2.c**:

```
// given a uint32_t,
// return 1 iff the least significant bit
// is equal to the most significant bit
// return 0 otherwise
int final_q2(uint32_t value) {
    (void) value; // REPLACE ME WITH YOUR CODE
    return 42;    // REPLACE ME WITH YOUR CODE
}
```

Add code to the function `final_q2` so that, given a `uint32_t` value, it returns **1** iff (if and only if) the least significant (bottom) bit of `value` is equal to the most significant (top) bit of `value`

`final_q2` should return **0** otherwise.

For example, given the hexadecimal value **0x12345678** which is **00010010001101000101011001111000** in binary, `final_q2` should return **1**, because the least significant bit is **0** and the most most significant bit is **0**.

Similarly, given the hexadecimal value **0x12345679** which is **00010010001101000101011001111001** in binary, `final_q2` should return **0**, because the least significant bit is **1** and the most most significant bit is **0**.

You must only use bitwise operators to implement `final_q2`.

For example:

```
$ dcc final_q2.c test_final_q2.c -o final_q2
$ ./final_q2 0x00000000
final_q2(0x00000000) returned 1
$ ./final_q2 0x00000001
final_q2(0x00000001) returned 0
$ ./final_q2 0x00000100
final_q2(0x00000100) returned 1
$ ./final_q2 0x00000101
final_q2(0x00000101) returned 0
$ ./final_q2 0x12345656
final_q2(0x12345656) returned 1
$ ./final_q2 0x12345657
final_q2(0x12345657) returned 0
$ ./final_q2 0xffffffff
final_q2(0xffffffff) returned 1
$ ./final_q2 0xfffffffe
final_q2(0xfffffffe) returned 0
$ ./final_q2 0x7fffffff
final_q2(0x7fffffff) returned 0
$ ./final_q2 0x7ffffffe
final_q2(0x7ffffffe) returned 1
```

You can also use *make* to build your code:

```
$ make final_q2
```

NOTE:

No error checking is necessary.

You **are not** permitted to call any functions from the C standard library.

You **are not** permitted to use: division (`/`), multiplication (`*`), or modulus (`%`).

You **are not** permitted to change the `main` function you have been given.

The `main` function you have been given is in `test_final_q2.c`,
not in `final_q2.c`.

You **are not** permitted to change `final_q2`'s prototype,
(its return type and argument types).

You **may** define and call your own functions if you wish.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q2
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q2 final_q2.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q2
```

Question 3 (9 MARKS)

Write a C program, `final_q3.c`, which takes two names of environment variables as arguments. `final_q3.c` should print **1** iff (if and only if) both environment variables are set to a **similar** integer value.

`final_q3.c` should print **0** otherwise.

Two integer values are **similar** if they differ by, strictly, less than **10**.

So **37** and **42** are **similar**.

So **52** and **42** are **not similar**.

An unset environment variable should be assumed to have value **42**

The shell command `export` sets an environment variable to a value; the shell command `unset` unsets an environment variable.

In the following example, `export` is used to set the environment variables `VAR1`, `VAR2` and `VAR3` and `unset` is used to ensure environment variable `VAR4` is not set.

```
$ export VAR1=40
$ export VAR2=45
$ export VAR3=55
$ unset VAR4
$ dcc final_q3.c -o final_q3
$ ./final_q3 VAR1 VAR2
1
$ ./final_q3 VAR3 VAR2
0
$ ./final_q3 VAR2 VAR4
1
$ ./final_q3 VAR4 VAR3
0
```

NOTE:

There is **no** supplied code for this question.

Your program **can** assume it is always given 2 arguments.

You program **can** assume that if an environment variable is set it contains a valid integer and nothing else.

Your program **should** always print one line of output to stdout.

The line of output **should** contain only a single character: 0 or 1, and a newline.

Your solution **must** be in C only.

You are **not** permitted to run external programs.
You are **not** permitted to use system, popen, posix_spawn, fork or exec.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q3
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q3 final_q3.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q3
```

Question 4 (9 MARKS)

You have been given `final_q4.s`, a MIPS assembler program that reads *one* number and then prints it.

Add code to `final_q4.s` to make it equivalent to this C program:

```c
// COMP1521 20T3 final exam Q4 C reference

// print low - high

#include <stdio.h>

int main(void) {
    int low = 0;
    int high = 100;
    while (low < high) {
        int x;
        scanf("%d", &x);
        low = low + x;
        high = high - x;
    }
    printf("%d\n", low - high);
    return 0;
}
```

In other words, it should read numbers untill **low** is greater or equal to **high**, and print **low - high**.

For example:

```
$ 1521 spim -f final_q4.s
10
20
25
10
$ 1521 spim -f final_q4.s
1
2
4
8
16
32
26
$ 1521 spim -f final_q4.s
100
100
$ 1521 spim -f final_q4.s
10
10
10
10
10
0
```

NOTE:

No error checking is required.

Your program **can** assume its input contains only integers, oner per line.

Your program **can** assume these integers will result in the program terminating.

You **can** assume the value of any expressions can be represented as a signed 32 bit value.
In other words, you **can** assume overflow/underflow does not occur.

Your solution **must** be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q4
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q4 final_q4.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q4
```

Question 5 (9 MARKS)

We need to a copy a file without some of the bytes at the end of the file.

Write a C program, `final_q5.c` which takes 3 arguments.

Its first argument will be an integer, **n**, the number of bytes not to be included.
Its second argument will be the name of an existing file.
Its third argument will be the name of the file to be created.

`final_q5.c` should copy the bytes of the existing file to the new file except for the last **n** bytes.

If the existing file has **n** bytes or less, the new files should still be created, but with zero bytes.

For example:

```
$ dcc final_q5.c -o final_q5
$ echo hello >hello.txt
$ ./final_q5 2 hello.txt new.txt
$ ls -l hello.txt new.txt
-rw-r--r-- 1 z5555555 z5555555 6 Nov 26 16:28 hello.txt
-rw-r--r-- 1 z5555555 z5555555 4 Nov 26 16:29 new.txt
$ xxd hello.txt
00000000: 6865 6c6c 6f0a                           hello.
$ xxd new.txt
00000000: 6865 6c6c                                hell
```

Note **new.txt** has the same contents as **hello.txt** except it is 2 bytes shorter.

The last two bytes of **hello.txt** (ASCII `'o'` and `'\n'`) were not copied to **new.txt**.

```
$ ./final_q5 2000 hello.txt file.txt
$ ls -l hello.txt file.txt
-rw-r--r-- 1 z5555555 z5555555 0 Nov 26 16:30 file.txt
-rw-r--r-- 1 z5555555 z5555555 6 Nov 26 16:28 hello.txt
$ xxd file.txt
$
```

NOTE:

There is **no** supplied code for this question.

No error checking is required.

Your program **can** assume it is always given 3 arguments.

Your program **can** assume its first argument is always a non-negative integer.

Your program **can** assume its second argument is always the pathname of an existing ordinary file.

Your program **can** assume its third argument is always the pathname of a file that can be created.

The file may contain any byte.

Your program **should** produce no output on stdout or stderr.

Your program **should** create a single file.

Your program **should** overwrite an existing file, if there is one.

Your program **can** assume all calculations can be done with `int` variables without overflow.

Your solution **must** be in C only.

You are **not** permitted to run external programs. You are **not** permitted to use system, popen, posix_spawn, fork or exec.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q5
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q5 final_q5.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q5
```

Question 6 (9 MARKS)

We need to count the number of bits which are set in a file. A bit is said to be **set** if it is **1**.

Write a C program, `final_q6.c`, which takes a single filename as its argument.

`final_q6.c` should read the bytes of the file, counting the number of bits that are set (1) in each byte.

`final_q6.c` should print one line of output containing the total number of bits which were set.

You must match the output format in the example below exactly.

```
$ dcc final_q6.c -o final_q6
$ echo > file0
$ xxd file0
00000000: 0a                                       .
$ ./final_q6 file0
file0 has 2 bits set
$ echo hello world > file1
$ xxd file1
00000000: 6865 6c6c 6f20 776f 726c 640a            hello world.
$ ./final_q6 file1
file1 has 47 bits set
$ ./final_q6 final_q6.0.bin
final_q6.0.bin has 1024 bits set
$ ./final_q6 final_q6.1.bin
final_q6.1.bin has 8084 bits set
$ ./final_q6 final_q6.2.bin
final_q6.2.bin has 9720 bits set
```

NOTE:

There is **no** supplied code for this question.

No error checking is required.

Your program **can** assume it is always given one argument: the name of a file.

The file may contain any byte.

Your program **can** assume the number of bits set can be stored in a `int` variable without overflow.

Your solution **must** be in C only.

You are **not** permitted to run external programs. You are **not** permitted to use system, popen, posix_spawn, fork or exec.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q6
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q6 final_q6.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q6
```

Question 7 (9 MARKS)

You have been given `final_q7.s`, a MIPS assembler program which implements all but one function of this C program.

```c
// COMP1521 20T3 final exam Q7 C reference

#include <stdio.h>

void read_array(int rows, int cols, int a[rows][cols]);
void reflect(int rows, int cols, int a[rows][cols], int b[cols][rows]);
void print_array(int rows, int cols, int a[rows][cols]);

int main(void) {
    int rows;
    int cols;
    scanf("%d", &rows);
    scanf("%d", &cols);
    int array1[rows][cols];
    int array2[cols][rows];
    read_array(rows, cols, array1);
    reflect(rows, cols, array1, array2);
    print_array(rows, cols, array1);
    printf("\n");
    print_array(cols, rows, array2);
}


void read_array(int rows, int cols, int a[rows][cols]) {
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            scanf("%d", &a[r][c]);
        }
    }
}


void reflect(int rows, int cols, int a[rows][cols], int b[cols][rows]) {
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            b[c][r] = a[r][c];
        }
    }
}


void print_array(int rows, int cols, int a[rows][cols]) {
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            printf("%d ", a[r][c]);
        }
        printf("\n");
    }
}
```

The function which has not been implemented is named **reflect**.

Add MIPS instructions for the function **reflect** to `final_q7.s`.

There are comments in `final_q7.s` which indicate where the MIPS instructions should be added.

`final_q7.s` should then be equivalent to the above C program:

For example:

```
$ 1521 spim -f final_q7.s
1
2
45
37
45 37

45
37
$ 1521 spim -f final_q7.s
3
2
14
15
16
17
18
19
14 15
16 17
18 19

14 16 18
15 17 19
$ 1521 spim -f final_q7.s < final_q7.input.txt
0 1
2 3
4 5
6 7
8 9

0 2 4 6 8
1 3 5 7 9
```

NOTE:

No error checking is required.

DO NOT CHANGE the supplied code for the functions: `main`, `read_array`, or `print_array`.

You **only** have to implement the function `reflect`.

You **can** assume the supplied code works.

Your solution **must** be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q7
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q7 final_q7.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q7
```

Question 8 (9 MARKS)

You have been given `final_q8.s`, a MIPS assembler program which implements all but one function of this C program.

```c
// COMP1521 20T3 final exam Q8 C reference

// Give a filename as a command line argument
// calculate the blobby_hash of the bytes of the file
// and print it

#include <stdio.h>

int blobby_hash(int hash, int byte);

int main(int argc, char *argv[]) {
    FILE *f = fopen(argv[1], "r");
    int hash = 0;
    int c;
    while ((c = fgetc(f)) != EOF) {
        hash = blobby_hash(hash, c);
    }
    printf("%d\n", hash);
}


int blobby_hash_table[256] = {
    241, 18,  181, 164, 92,  237, 100, 216, 183, 107, 2,   12,  43,  246, 90,
    143, 251, 49,  228, 134, 215, 20,  193, 172, 140, 227, 148, 118, 57,  72,
    119, 174, 78,  14,  97,  3,   208, 252, 11,  195, 31,  28,  121, 206, 149,
    23,  83,  154, 223, 109, 89,  10,  178, 243, 42,  194, 221, 131, 212, 94,
    205, 240, 161, 7,   62,  214, 222, 219, 1,   84,  95,  58,  103, 60,  33,
    111, 188, 218, 186, 166, 146, 189, 201, 155, 68,  145, 44,  163, 69,  196,
    115, 231, 61,  157, 165, 213, 139, 112, 173, 191, 142, 88,  106, 250, 8,
    127, 26,  126, 0,   96,  52,  182, 113, 38,  242, 48,  204, 160, 15,  54,
    158, 192, 81,  125, 245, 239, 101, 17,  136, 110, 24,  53,  132, 117, 102,
    153, 226, 4,   203, 199, 16,  249, 211, 167, 55,  255, 254, 116, 122, 13,
    236, 93,  144, 86,  59,  76,  150, 162, 207, 77,  176, 32,  124, 171, 29,
    45,  30,  67,  184, 51,  22,  105, 170, 253, 180, 187, 130, 156, 98,  159,
    220, 40,  133, 135, 114, 147, 75,  73,  210, 21,  129, 39,  138, 91,  41,
    235, 47,  185, 9,   82,  64,  87,  244, 50,  74,  233, 175, 247, 120, 6,
    169, 85,  66,  104, 80,  71,  230, 152, 225, 34,  248, 198, 63,  168, 179,
    141, 137, 5,   19,  79,  232, 128, 202, 46,  70,  37,  209, 217, 123, 27,
    177, 25,  56,  65,  229, 36,  197, 234, 108, 35,  151, 238, 200, 224, 99,
    190
};

int blobby_hash(int hash, int byte) {
    return blobby_hash_table[hash ^ byte];
}
```

The function which has not been implemented is named **blobby_hash**.

Add MIPS instructions for the function **blobby_hash** to `final_q8.s`. You will also need to implement the equivalent of the `blobby_hash_table` array.

There are comments in `final_q8.s` which indicate where the MIPS instructions should be added.

`final_q8.s` should then be equivalent to the above C program:

For example:

```
$ 1521 spim -f final_q8.s final_q8.bin
125
$ echo hello > hello.txt
$ 1521 spim -f final_q8.s hello.txt
76
$ 1521 spim -f final_q8.s /dev/null
0
```

NOTE:

No error checking is required.

DO NOT CHANGE the supplied code for `main`.

You **can** assume the supplied code works.

The supplied MIPS instructions for `main` accesses the command line argument and performs file operations.
You do **not** need to understand this code.

Your code does **not** have to access command line arguments or perform file operations.

You **only** have to implement the function `blobby_hash` and create the `blobby_hash_table` array.

Your solution **must** be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q8
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q8 final_q8.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q8
```

Question 9 (9 MARKS)

You have been given `final_q9.s`, a MIPS assembler program which implements all but 3 functions of this C program.

```c
// COMP1521 20T3 final exam Q9 C reference

#include <stdio.h>

void sort(int n, int a[]);
int partition(int n, int a[]);
void swap(int *x, int *y);
void read_array(int n, int a[n]);
void print_array(int n, int a[n]);

int main(void) {
    int size;
    scanf("%d", &size);
    int array[size];
    read_array(size, array);
    sort(size, array);
    print_array(size, array);
}

void sort(int n, int a[]) {
    if (n > 1) {
        int p = partition(n, a);
        sort(p, a);
        sort(n - (p + 1), a + p + 1);
    }
}

int partition(int n, int a[]) {
    int pivot_value = a[n - 1];
    int i = 0;
    for (int j = 0; j < n; j++) {
        if (a[j] < pivot_value) {
            swap(&a[i], &a[j]);
            i = i + 1;
        }
    }
    swap(&a[i], &a[n - 1]);
    return i;
}

void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

void read_array(int n, int a[]) {
    for (int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
}

void print_array(int n, int a[]) {
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}
```

The functions which have not been implemented are named **sort**, **partition** and **swap**.

Add MIPS instructions for the functions **sort**, **partition** and **swap** to `final_q9.s`.

There are comments in `final_q9.s` which indicate where the MIPS instructions should be added.

`final_q9.s` should then be equivalent to the above C program.

For example:

```
$ 1521 spim -f final_q9.s
3
15
18
9
9 15 18
$ 1521 spim -f final_q9.s
4
19
4
81
1
1 4 19 81
$ 1521 spim -f final_q9.s
5
9
4
8
1
7
1 4 7 8 9
```

NOTE:

You **must** implement functions `sort`, `partition`, and `swap` equivalent to the supplied C.

No marks will be given for implementing other functions.

No marks will be given for implementing other sorting algorithms.

No error checking is required.

DO NOT CHANGE the supplied code for `main`, `read_array`, and `print_array`.

You **can** assume the supplied code works.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q9
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q9 final_q9.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q9
```

Question 10 (9 MARKS)

Your task is to write a program named **final_q10.c** which executes a specified command with arguments it reads from standard input.

Your program should read lines of input from stdin until end-of-file and execute the command once for each line of input.

The command to execute will be specified as your program's first command line argument.

Each line will contain the arguments to execute the command with. The arguments will be separated by one or more spaces.

You must execute the command with *posix_spawnp*.

For example:

```
$ dcc final_q10.c -o final_q10
$ ./final_q10 cp
file1 file1.backup
file2 file2.backup
file3 file3.backup
Ctrl-D
$
```

In the above example, your program should call *posix_spawnp* 3 times to execute these 3 commands:

```
cp file1 file1.backup
cp file2 file2.backup
cp file3 file3.backup
```

If your program is given more than one command-line argument, any additional arguments should be passed as the first arguments to the command to be executed followed by any arguments in each line read from standard input.

For example:

```
$ ./final_q10 cp file
file.backup1
file.backup2
file.backup3
file.backup4
Ctrl-D
$
```

In the above example, your program should call posix_spawnp 4 times to execute these 4 commands:

```
cp file file.backup1
cp file file.backup2
cp file file.backup3
cp file file.backup4
```

You should ignore any spaces at the start and finish of lines. There maybe more than one space between arguments.

For example:

```
$ ./final_q10 ls -l -a
  final_q1.s
   final_q2.c   final_q3.c      final_q4.s
Ctrl-D
-rw-r--r-- 1 z5555555 z5555555 42 Nov 28 13:08 final_q1.s
-rw-r--r-- 1 z5555555 z5555555 42 Nov 28 13:21 final_q2.c
-rw-r--r-- 1 z5555555 z5555555 42 Nov 28 13:39 final_q3.c
-rw-r--r-- 1 z5555555 z5555555 42 Nov 28 13:58 final_q4.s
$
```

In the above example, your program should call posix_spawnp twice to execute these 2 commands:

```
ls -l -a final_q1.s
ls -l -a final_q2.c final_q3.c final_q4.s
```

If your program is given no arguments it should run the command echo for each line of input.
In other words the command to run defaults to echo.

For example:

```
$ ./final_q10
   This is    a  difficult exam   question!
Ctrl-D
This is a difficult exam question!
$
```

In the above example, your program should call posix_spawnp once to execute this command:

```
echo This is a difficult exam question!
```

Note the output in the above example is from echo not **final_q10.c**.

NOTE:

There is **no** supplied code for this question.

Your solution **must** be writen in C only.

The **only** external program you may run is the specified command.
You **must** run this program with posix_spawnp

You are **not** permitted to use system, popen, fork or exec.

No error checking is necessary.

Your program **should** produce no output.
The commands it runs may or may not produce output.

Your program **should** run the specified command once for every line of input.

You **can** assume that each line of input will not exceed **65536** bytes.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q10
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q10 final_q10.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q10
```

Question 11 (9 MARKS)

We need a program to check if the same files are present in two directory trees.

Write a C program, `final_q11.c`, which is given either 2 arguments which are the pathnames of directories.

`final_q11.c` should print a single line of output containing 4 integers:

- number of files which are present at the same position in both directory trees and are the same size.
- number of files which are present at the same position in both directory trees but are different sizes.
- number of files which are present only in the first directory tree.
- number of files which are present only in the second directory tree.

Note, a file needs to be present in both directory trees at the same relative pathname to be counted.

For example, if we are comparing the directory trees **dir1** and **dir2**. and the file **dir1/1521/lab09/main.c** exists in the first directory tree,

We consider it present in the second directory tree only if the file **dir2/1521/lab09/main.c** exists.

Other files named **main.c** elsewhere in the second directory tree, e.g. **dir2/2521/lab05/main.c** are not counted.

When a file is present at the same relative pathname in both directory tree `final_q11.c` does not have check it contains the same contents (bytes), just whether both files are the same size (number of bytes).

For example:
these commands create 2 directory tree named **d1** and **d2** containing the same 3 files.

```
$ mkdir -p d1/b/c
$ echo hello andrew > d1/file1
$ echo bye andrew > d1/b/file2
$ echo 1 > d1/b/c/one
$ mkdir -p d2/b/c
$ echo HELLO andrew > d2/file1
$ echo Bye Andrew > d2/b/file2
$ echo 2 > d2/b/c/one
```

Their contents of the 3 files are different but their sizes are the same.

```
$ find d1 d2 -type f | xargs stat -c '%3s %n'
 13 d1/file1
 11 d1/b/file2
  2 d1/b/c/one
 13 d2/file1
 11 d2/b/file2
  2 d2/b/c/one
```

final_q11 reports 3 files present in both tree of the same size.

```
$ dcc final_q11.c -o final_q11
$ ./final_q11 d1 d2
3 0 0 0
```

If we change the size of `file1` in the first directory tree **final_q11** reports 2 files present in both tree of the same size. and 1 file present in both tree but different size.

```
$ echo hello everyone >d1/file1
$ ./final_q11 d1 d2
2 1 0 0
```

If we add a file to the second directory tree:

```
$ echo 3 > d2/b/c/three
$ ./final_q11 d1 d2
2 1 0 1
```

If we add a different file to the first directory tree:

```
$ echo 3 > d1/b/three
$ ./final_q11 d1 d2
2 1 1 1
```

Note, the first directory tree contains a file named **b/three** and the second tree contains a file named **b/c/three**.
This is not considered as the file being present in both directory trees, as the absolute pathname of each file is different.

WARNING

Autotest will only be of limited assistance in debugging your program.
Do not expect autotest messages to be easy to understand for this problem.
You will need to debug your program yourself.

NOTE:

There is **no** supplied code for this question.

Your solution **must** be in C only.

You are **not** permitted to run external programs.
You are **not** permitted to use system, popen, posix_spawn, fork or exec.

You are **not** permitted to use libraries other than the default C libraries.
In other words your solution can not require use of dcc's `-l` flag.
If your solution compiles without dcc's `-l` flag, you are using only the default C libraries.
All functions discussed in lectures are part of the default C libraries.

No error checking is necessary.

You **can** assume the directory trees to be compared contain only directories and regular files.
You **can** assume they do not contain links or other special files.
You **can** assume they do not contain sparse files.

You can **not** assume anything about the size or contents of files in the directory tree.
The files **may** contain any byte.
The files **may** be any size.

Your program does **not** have to consider permissions, modification times, or other file metadata.

You **can** assume the maxmimum absolute pathname length of each file or directory will not exceed **65536** bytes.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest final_q11
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 final_q11 final_q11.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check final_q11
```

Submission

When you are finished working on a question, submit your work by running **give**.

You can run **give** multiple times. Only your last submission will be marked.

Don't submit any questions you haven't attempted.

Do not leave it to the deadline to submit your answers. Submit each question when you finish working on it. Running autotests does not automatically submit your code.

You can check if you have made a submission with `1521 classrun -check final_q<N>`:

```
$ 1521 classrun -check final_q1
$ 1521 classrun -check final_q2
...
$ 1521 classrun -check final_q11
```

Remember you have until **Saturday 28 November 16:00** Sydney time to complete this exam (not including any extra time provided by ELS conditions).

Do your own testing as well as running **autotest**