CS 137: Assignment #5

Due on Friday, Nov 4, 2022, at 11:59PM

Submit all programs using the Marmoset Submission and Testing Server located at

https://marmoset.student.cs.uwaterloo.ca/

Victoria Sakhnini

Fall 2022

Notes:

- Use the examples to guide your formatting for your output. Remember to terminate your output with a newline character.
- For this assignment, you may use any content covered until the end of Module 8.

Problem 1

Define a structure `struct fraction` to represent fractions. Use this definition to implement the following seven functions:

```
struct fraction fractionCreate(int numerator, int denominator);
```

that creates a reduced fraction with the specified numerator and denominator.

Please note that the fraction is always stored as two integers representing the numerator and the denominator. Fractions are always represented in the structure as reduced a/b even if a>b.

Assume that the denominator always doesn't equal zero. If the fraction is negative, the denominator will be the negative number. Check the tests below.

`int get_numerator(struct fraction a);` returns the numerator of the fraction `a`.

`int get_denominator(struct fraction a);` returns the denominator of the fraction `a`.

`struct fraction fractionAdd(struct fraction a, struct fraction b);`
returns `a + b` (reduced fraction).

`struct fraction fractionSubtract(struct fraction a, struct fraction b);`
returns `a – b` (reduced fraction).

`struct fraction fractionMultiply(struct fraction a, struct fraction b);`
returns `a * b` (reduced fraction).

`struct fraction fractionDivide(struct fraction a, struct fraction b);`
returns `a / b` (reduced fraction).

Ensure that your program reduces the fraction after every arithmetic operation and in fractionCreate. As long as your fraction is reduced after every step, you may assume that the numerator and denominator from the result are always within ± 1,000,000,000, and the intermediate steps will never cause an overflow.

Place your definition for `struct fraction` and the declarations for the functions above in the file `fraction.h`. Place the implementation for the functions in the file `fraction.c`. Submit a zip or jar file called `fraction.zip` containing both `fraction.h` and `fraction.c`.

Do **not** submit your test program.

To create a zip file called "`fraction.zip`" containing your files, type:

```
zip fraction.zip fraction.c fraction.h
```

Sample Test program (Make sure you test your solutions for more cases)

```c
#include <assert.h>
#include "fraction.h"

int main (void)
{
    struct fraction a, b, c, r, zero;
    a = fractionCreate(40, 26);
    assert(20==get_numerator(a));
    assert(13==get_denominator(a));
   b = fractionCreate(18, 19);
   assert(18==get_numerator(b));
   assert(19==get_denominator(b));
   c = fractionCreate(360, 540);
   assert(2==get_numerator(c));
   assert(3==get_denominator(c));
   zero = fractionCreate(0, 10);
   assert(0==get_numerator(zero));
   assert(1==get_denominator(zero));

   r = fractionAdd(a, b);
   assert(614==get_numerator(r));
   assert(247==get_denominator(r));

   r = fractionSubtract(c, c);
   assert(0==get_numerator(r));
   assert(1==get_denominator(r));

   r = fractionMultiply(a, b);
   assert(360==get_numerator(r));
   assert(247==get_denominator(r));

   r = fractionDivide(c, a);
   assert(13==get_numerator(r));
   assert(30==get_denominator(r));
   r = fractionDivide(c, c);
   assert(1==get_numerator(r));
   assert(1==get_denominator(r));

   r = fractionCreate(-10,-2);
   assert(5 == get_numerator(r));
   assert(1 == get_denominator(r));
   r = fractionCreate(-2,14);
   assert(1 == get_numerator(r));
   assert(-7 == get_denominator(r));
   a = fractionCreate(1,3);
   b = fractionCreate(3,1);
   r = fractionSubtract(a,b);
   assert(8 == get_numerator(r));
   assert(-3 == get_denominator(r));
   return 0;
}
```

Problem 2

Assume the file `event.h` contains the following definitions and declarations:

```
/* A time of the day */
struct tod {
   int hour, minute;
};
/* An event with a start and end time; The start time is part of the event, but the end time isn't;
 the end time is always free. So an event running from 9:30 to 10:30 ends just before 10:30. */
struct event {
   struct tod start, end;
};
/* Returns 1 if the specified time (hour and minute) is not part of any scheduled event; returns 0
   otherwise. The value n specifies the size of the array containing the schedule.  */
int freetime (struct event schedule[], int n, int hour, int min);
```

Implement the function `freetime`. Place your implementation in the file `event.c`.

A sample program is provided to test the function, and its output is given below.

Submit both files `event.c`, containing your solution and `event.h` combined in a zip file named `event.zip`.

Sample Test program (You need to add more test cases)

```
1. #include <assert.h>
2. #include "event.h"
3.
4. int main (void)
5. {
6.   struct event schedule[] = {{{9,45},{9,55}},{{13,0},
7.                              {14,20}},{{15,0},{16,30}}};
8.
9.   assert(freetime(schedule,3,10,0));
10.  assert(!freetime(schedule,3,9,50));
11.  return 0;
12.}
13.
```

Assumptions: data is valid, and the schedule is for one day.

No assumption about the order of the events in the schedule.

Problem 3

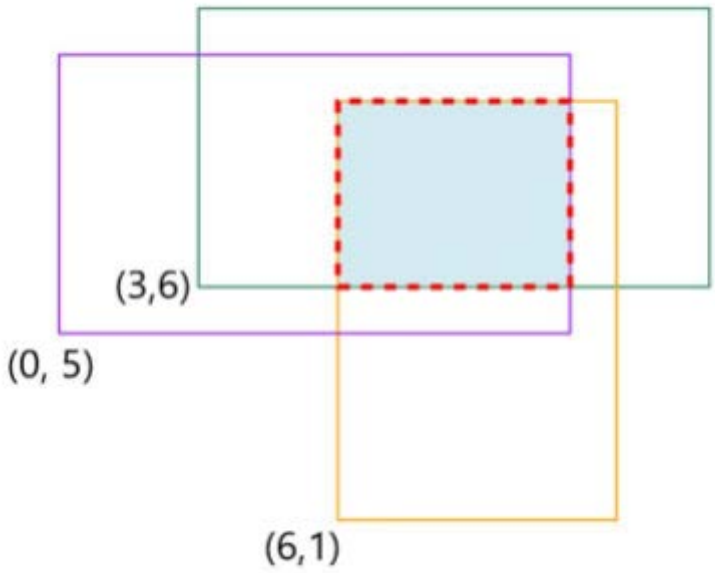
Consider the following struct.

```
typedef struct Point{
      int x;
      int y;
} point;
typedef struct Rectangle {
   point bottomLeft; // represents the bottom left corner of the rectangle
   int width;
   int height;
} rectangle;
```

Write the following function:

`rectangle intersection(rectangle rects[], int n).`

which takes in an array or `rectangle` structs of length `n`, computes the rectangle we get from the intersection of these rectangles in the array, and returns it. For instance, in the following example, for the three input rectangles (green, purple, orange), the intersection resulting from the intersection of the three rectangles is the blue rectangle with the red dotted boundary.



If all the rectangles do not intersect (i.e. an empty intersection) then return `rectangle` with the values `{{0,0},0,0}`. Otherwise return the result (the intersection) as of type `rectangle`.

Submit `rec.c` file containing only your implemented function (that is, you must delete the test cases portion/the main function). However, **you must keep the required included libraries.**

Sample Test program (You need to add more test cases)

```
1. #include <assert.h>
2.
3. int main(){
4.
5.    rectangle result;
6.    rectangle r = {{2,6},3,4};
7.    rectangle s = {{0,7},7,1};
8.    rectangle t = {{3,5},1,6};
9.    rectangle u = {{5,6},3,4};
10.
11.   // Test 1
12.   rectangle rects1[2] = {r, s};
13.   result = intersection(rects1, 2);
14.   assert(result.bottomLeft.x == 2);
15.   assert(result.bottomLeft.y == 7);
16.   assert(result.width == 3);
17.   assert(result.height == 1);
18.
19.   // Test 2
20.   rectangle rects2[3] = {r, s, t};
21.   result = intersection(rects2, 3);
22.   assert(result.bottomLeft.x == 3);
23.   assert(result.bottomLeft.y == 7);
24.   assert(result.width == 1);
25.   assert(result.height == 1);
26.
27.   // Test 3
28.   rectangle rects3[4] = {r, s, t, u};
29.   result = intersection(rects3, 4);
30.   assert(result.bottomLeft.x == 0);
31.   assert(result.bottomLeft.y == 0);
32.   assert(result.width == 0);
33.   assert(result.height == 0);
34.
35.   return 0;
36. }
```