Project 4, Program Design

1. Sets of numbers can be represented using array of `0`s and `1`s. The idea is that `a[i]!=0` if `i` is in the set, and `a[i] == 0` if it is not. For example, the array `a[10] = {0, 0, 1, 0, 1, 1, 0, 0, 0, 0}` would represent the set `{2, 4, 5}` because `a[2]`, `a[4]`, and `a[5]` have the value `1`, and everywhere else a contains zeros. Since the array has a fixed bound, say `N`, the values in the set are restricted to the range `0…N-1`.

   Write a C program that reads in two sets of numbers `A` and `B`, and calculates and print their union ($A \cup$ B) and intersection ($A \cap$B). $A \cup B$ is the set of elements that appear in either A or B, and that ($A \cap B$ is the set of elements that appear in both A and B. The values in the sets are restricted to the range `0… 9`.

   1) Name your program *`sets.c`*.
   2) The program will read in the number of element in the first set, for example, 4, then read in the numbers in the set, for example, 3 6 8 9. The repeat for the second set. The two sets do not necessarily are of the same size.
   3) **The sets are stored using arrays of 0s and 1s as described above.**
   4) Calculate the union and intersection of the two sets and display the result.
   5) Sample input/output:

```
Please enter the number of elements in set A: 3

Enter the numbers in set A: 3 5 8

Please enter the number of elements in set B: 4

Enter the numbers in set B: 7 5 9 3

Output:

The union of set A and B is: 3 5 7 8 9

The intersection of set A and B is: 3 5
```

2. A binary bar code scan is a bit pattern that contains only 1s and 0s. Write a program that finds the edges of light and dark regions. Process an input bit pattern in the following manner:
   - Assign a 1 to the output bit pattern whenever two consecutive bits (one bit and it's previous bit) are different
   - Assign a 0 to the output bit pattern whenever two consecutive bit (one bit and it's previous bit) are the same
   - Assign 0 to the first output bit since there is no previous bit for the first bit

   For example, input and output bit pattern of the program that detects the edges might look like the following:

   Input: 00101101
   Output: 00111011

1) Name your program `barcode.c`.
2) Assume the input contains exactly 8 bits.
3) As part of the solution, write and call the function edge() with the prototype

```
void edge(int n, int a1[], int a2[]);
```

to perform edge detection. The arguments of the function `edge()` contain the length of the input and output arrays with the same size, the input array `a1`, and output array `a2`.

Before you submit

1. Compile both programs with –Wall. –Wall shows the warnings by the compiler. Be sure it compiles on ***circe*** with no errors and no warnings.

   gcc -Wall sets.c

   gcc –Wall barcode.c

2. Be sure your Unix source file is read & write protected. Change Unix file permission on Unix:

   chmod 600 sets.c

   chmod 600 barcode.c

3. Test the second program with the shell script try_sets and try_barcode on Unix:

   chmod +x try_sets

   ./try_sets

   chmod +x try_barcode

   ./try_barcode

4. Submit sets.c and barcode.c on Canvas.

Grading

Total points: 100 (50 point each problem)

1. A program that does not compile will result in a zero.
2. Runtime error and compilation warning 5%
3. Commenting and style 15%
4. Functionality 80%
   Problem #1, the sets stored and computed using arrays of 0s and 1s as described (60% of functionality)
   Problem #2, function edge() implemented as required (60% of functionality)

Programming Style Guidelines

The major purpose of programming style guidelines is to make programs easy to read and understand. Good programming style helps make it possible for a person knowledgeable in the application area to quickly read a program and understand how it works.

1. Your program should begin with a comment that briefly summarizes what it does. This comment should also include your **name**.
2. In most cases, a function should have a brief comment above its definition describing what it does. Other than that, comments should be written only *needed* in order for a reader to understand what is happening.
3. Variable names and function names should be sufficiently descriptive that a knowledgeable reader can easily understand what the variable means and what the function does. If this is not possible, comments should be added to make the meaning clear.
4. Use consistent indentation to emphasize block structure.
5. Full line comments inside function bodies should conform to the indentation of the code where they appear.
6. Macro definitions (#define) should be used for defining symbolic names for numeric constants. For example: **#define PI 3.141592**
7. Use names of moderate length for variables. Most names should be between 2 and 12 letters long.
8. Use underscores to make compound names easier to read: **tot_vol** or **total_volumn** is clearer than totalvolumn.