Project 3, 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 difference of set A and B: A – B, complement of set A: $\bar{A}$ and complement of set B: $\bar{B}$. A – B is the set of elements that appear in A but not in B, and that $\bar{A}$ is the set of every element that is not in A. The values in the sets are restricted to the range `0… 9`.

   1) Name your program *`set_operations.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 difference of A and B, and complements 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 difference of set A and B is: 8

The complement of set A is: 0 1 2 4 6 7 9

The complement of set B is: 0 1 2 4 6 8
```

2. Assume the + operator is not available. Write a program (addition.c) that takes two numbers as input and display the addition of them. The program should include a function add (int n, int m) that will add two numbers **using only recursion** and the increment and decrement operators. Either of the two numbers can be zero, positive, or negative.

   Hint: add(n, m)= add(++n, --m), if m is positive, and add(n, 0)=n.

3. Modify Project 1 (knuts.c) (Write a C program that converts knuts to sickles and galleons (the currency of the Harry Potter novels). The user will enter the total number of knuts. There are 29 knuts in one sickle and 17 sickles in one galleon.) So it includes the following function:

```
void convert(int total_knuts, int *galleons, int *sickles, int
*knuts);
```

The function determines the number of galleons, sickles, and knuts for the amount of the total number of knuts. The `galleons` parameter points to a variable in which the function will store the number of galleons required. The `sickles` and `knuts` parameters are similar. Modify the `main` function so it calls `convert` to compute the smallest number of galleons, sickles, and knuts. The `main` function should display the result. Name your program knuts2.c.

Note: this program (#3) will be graded based on whether the required functionality were implemented correctly instead of whether it produces the correct output, for the functionality part (80% of the total grade)

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_operations.c

   gcc –Wall addition.c

   gcc –Wall knuts2.c

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

   chmod 600 sets_operations.c

   chmod 600 addition.c

   chmod 600 knuts2.c

3. Test your programs with the shell script try_set_operations , try_addition, and try_knuts on Unix:

chmod +x try_set_operations

./try_set_operations

chmod +x try_addition

./try_addition

chmod +x try_knuts

./try_knuts

4. Submit *sets_of_numbers.c*, *addition.c*, and knuts.c on Canvas.

Grading

Total points: 100 (problem 1: 40 points; problem 2: 30 points, problem 3: 30 points)

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%

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.