Project 3, Program Design

1. (50 points) Given an array `a` of length `n`, write a program that find all the integers in the range [1, n] that do not appear in `a`. For example, if an input array of length 5 contains {4, 3, 3, 2, 2}. The output will be 1 and 5, all the integers in the range [1, 5] that do not appear in the input array.

   Example input/output #1:
   Enter the length of the input array: 4
   Enter the elements of the input array: 2 4 1 4
   Output: 3

   Example input/output #2:
   Enter the length of the input array: 8
   Enter the elements of the input array: 4 3 7 1 3 2 8 2

   Output: 5 6

   1) Name your program **`arrays1.c`**.
   2) In the main function, the program will ask the user to enter the length of the array and the elements of the array. Assume the elements entered are in the range of [1, n], where n is the length of the array.
   3) Include the function `find()` in the program. The `find()` function finds all the integers in the range [1, n] that do not appear in `a`.

   ```
   void find(int a[],int n, int b[]);
   ```

   Array `b` should be of size n and contains 0s and 1s. An element of b is 1 if the (index of that element + 1) appears in a, and 0 otherwise. For example, an input array of length 5 contains {4, 3, 3, 2, 2}, array b will be {0, 1, 1 , 1, 0}.

   4) The main function calls the find function and displays the output.

2. (50 points) Write a program that checks if two arrays of the same length match after some number of shifts on the first array. A shift on an array means moving the leftmost element to the rightmost position. For example, if array `a` contains {4, 6, 1, 2}, then it will be {6, 1, 2, 4} after one shift. If array b contains {1, 2, 4, 6}, then array `a` matches

array `b` after 2 shifts. Your program will display "true" if and only if array `a` can become array `b` after some number of shift on `a`. Your program will display "false" otherwise.

Example input/output #1:
Enter the length of the input array: 4
Enter the elements of the first array: 2 8 1 4
Enter the elements of the second array: 1 4 8 2
Output: false

Example input/output #2:
Enter the length of the input array: 5
Enter the elements of the first array: 4 6 7 1 3
Enter the elements of the second array: 1 3 4 6 7

Output: true

1) Name your program **`arrays2.c`**.
2) In the main function, the program will ask the user to enter the length of the arrays and the elements of each array.
3) Include the function `shift()` in the program. The `shift()` function moves the leftmost element of array `a` to the rightmost position.

```
void shift(int a[], int n);
```

4) The main function calls the shift function, evaluates the arrays, and displays the result.

Before you submit

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

gcc –Wall arrays1.c

gcc –Wall arrays2.c

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

   chmod 600 arrays1.c

   chmod 600 arrays2.c

3. Use the test scripts to test your program:

   chmod +x try_arrays1

   ./try_arrays1

   chmod +x try_arrays2

   ./try_arrays2

4. Submit arrays1.*c* and arrays2.c on Canvas.

Grading

Total points: 100 (50 points 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% (including functions implemented as required)

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.