CSE 3100 Systems Programming Exam #1

NetID: ____________________ First Name:____________________ Last Name: ____________________

The procedure on the submission has been done. TA's signature____

Note the Makefile for all the following problems is available.
Short Answers(10 points)Russian Peasant's Algorithm

The "Russian Peasant's Algorithm" uses division and multiplication by 2 (which can be done using bit operations) repeatedly to calculate the product of two integers. This is the pseudo-code to the algorithm; please translate it to C, and put it in the function `multiply` in `multiply.c`:

```
//Integers a and b are provided.
res = 0
while b > 0:
    if b % 2 == 1:
        res += a
    a = a << 1
    b = b >> 1
return res
```

What is the output of the program when you run it as follows?

```
    ./multiply 60
```

Note here we only provide one number to be multiplied. The other number is randomly generated in the program.

Code Problem 1(20 points) Estimating e

In this problem, we run random simulations to estimate e. We will not explain in detail the math behind it. Below is what we need to do.

Fill in the `count_n` function in the `value_of_e.c` file. The provided function `uniform_random` generates a random number in the range $[0, 1)$. Repeatedly call this function, keeping track of the sum of random numbers generated, and stop when the sum exceeds 1. Return the count of random numbers generated.
For example, if the generated random number is 0.4864, since it does not exceed 1, we generate another number; suppose we get 0.3738. Our sum is 0.8602. Since that does not exceed 1, we generate another number; suppose we get 0.259. Our sum is now 1.1192. Since that *does* exceed 1, we do not generate another random number. Instead, we return 3, the number of random numbers generated.

Code Problem 2(20 points) Circular Walker

In this problem we will write a program to simulate a circular walker. A walker walks along the index of an array in a circular manner. The size of the array is n. The index of the array therefore is between 0 and $n-1$. When the walker is at the index k, he will go to index k + a[k] for the next step, if $k+a[k]$ is between 0 and $n-1$. Otherwise, the modular operation is used to ensure the result is between 0 and $n-1$. Before the walk leaves the current index k, the value a[k] is decremented by 1.

For example, when $k=2$, and $n=5$, if $a[k]=4$, then $k+a[k]=2+4=6$. Applying modular operation we have $6\%n = 6\%5 = 1$. Hence the next index is 1. Before the walker goes to index 1, $a[2]$ is decremented by 1, and hence $a[2]$ becomes 3.

Note if a[k] is 0, the walker will stop.

In the starter code `rotate.c`, implement the function

```
int next_index(int a[], int k, int n)
{

}
```

This function returns the index for the next step and also updates a[k] before the function returns.

Do not change the `main()` function. Only write code inside the `next_index()` function.
If your code is correct, you will notice the walker stops at a special location.

Code Problem 3(50 points) Prime Numbers

In this assignment, we write code in the starter code prime.c to find all the prime numbers between 1 and n, where n is a positive integer.

We use a singly linked list for this purpose. Specifically, we loop i from n to 2, and for each i, we create a node for i, and insert this node at the head of a linked list. After this, the list will contain nodes for number $2, 3, \ldots, n$ and in this order. And then for each number i between 2 and $n-1$, we remove all the multiples of i, but not i itself from the linked list. Once this is done, the list should contain all the prime numbers between 1 and n. Think why this is the case.

We implement the following function for this purpose. Here we remove all the nodes whose value are multiples of k but not k itself from the list specified by head. Once a node is from the list, we should free the memeory allocated for it.

```
void remove_multiple(node **head, int k)
{

}
```

We also need to implement the following function

```
int list_length(node *head)
{

}
```

This function returns the number of items in the list. Once we have the list that contains all and only the prime numbers between 1 and n, we use the above function to print out the total number of primes in the range. Also we print out each of these prime numbers using the function `print_list()`.

Once this is done, we remove all the nodes in the list and free their corresponding memory. We implement the following function for this purpose.

```
void free_all(node **head)
{

}
```

Note do not change the main() function. Below is an example output from the program.

```
$ ./prime 9
2 3 4 5 6 7 8 9
Remove multiples of 2
2 3 5 7 9
Remove multiples of 3
2 3 5 7
Remove multiples of 4
2 3 5 7
Remove multiples of 5
2 3 5 7
Remove multiples of 6
2 3 5 7
Remove multiples of 7
2 3 5 7
Remove multiples of 8
4 primes between 1 and 9:
2 3 5 7
0 items left in the list after free_all().
```