Assignment 4
Bit Vectors And Primes

<u>Pre-Lab Part 1</u>

1. Assuming you have a list of primes to consult, write pseudo-code to determine if a number is a Fibonacci prime, a Lucas prime, and/or a Mersenne prime

```
fibonacci _num = [0,1]
Lucas_num = [2,1]
Mersenne_num = []
Mersenne_tracker = 0
Mersenne_num_printed = 0
For all the numbers in prime:
    While the last num in fibonnai_num < current num loop is on:
        Add the summation of last and second last numbers in fibonacci array into
fibonacci;
        If the summation equals the current num:
            Mark it as fibonacci;
    Mersenne_num[mersenne_tracker] = 2^current_num - 1


        If that mersenne_num[mersenne_num_printed] == current_num:
            Mark it as mersenne prime;
            break;
    While the last num in lucas_num < current_num:
        If current_num is 2:
            Label it lucas and break;
        Add the summation of last and second last numbers in lucas to the array.
        If the added number == current num on loop:
            Label it lucas
```

2. Assuming you have a list of primes to consult, write pseudo-code to determine if a number in base 10 is a palindrome. Note that the technique is the same in any base.
Assuming prime numbers are stored in an array called primes.

```
boolean isPalindrome(primes):
    //using the approach to reverse the number and then check if it's the same
    For i in primes:
        Int num = i
        reversed_num = 0
        While num > 0:
            Multiply the current value in reverse_num by 10 to shift its index
left and add the remainder of num and base
            Num =/base
        If reversed_num == num:
```

```
                    Return true
```

Pre-Lab Part 2

1. Implement each BitVector ADT function(Note the function name and parameters are copied directly from the pdf)

```
BitVector * bv_create ( uint32_t bit_len ){
        Use malloc() to allocate space in memory for len of int type
        Use calloc() to allocate space of # rows needed in memory for the vector
        Set each row to have int of 8 bits
}

void bv_delete ( BitVector *v){
        Free the vector then free the entire bitvector
}

uint32_t bv_get_len ( BitVector *v){
        Return the length of v.//v->length
}

void bv_set_all_bit ( BitVector *v, uint32_t i){
        Loop through each row in vector
        Loop through each bit and set it to 1
}

void bv_clr_bit ( BitVector *v, uint32_t i){
        //Use mask technique
        Mask = 0b11111111
        Divide i by 8 to see how many rows
        set the bit at i%8 in mask to 0
        V.vector[row] = v.vector[row] & mask
}
uint8_t bv_get_bit ( BitVector *v, uint32_t i) {
        Mask = 0b00000000
        Divide i by 8 to get the row #
        Set the bit at i%8 to 1 in mask
        Do & operation between mask and v.vector[row]//This will guarantee all the other
bits will be 0 and if the bit at index i is one, the result would be one.
        if the operation value > 0, return 1 else return 0
}
```

2. Explain how you avoid memory leaks when you free allocated memory for your BitVector ADT.

   Free all the memory you allocate. Do not forget to do this. Add conditions to check that

Memory allocation using malloc(), calloc(), and realloc() is successful else they return NULL.

3. While the algorithm in sieve() is correct, it has room for improvement. What change would you make to the code in sieve() to improve the runtime?

   From what I understand, sieve() loops through the index of the bit vector and within the length of the vector, it changes the multiple of each of those indexes to be 0 implying that it’s not prime which is how it should be. One major way to improve the algorithm would be not to try to loop to eliminate the multiples of numbers whose multiples have already eliminated their multiples. Sounds strange but take this example: suppose you have a bit vector of length 50. Sieve starts at 2 and eliminates all the multiples of 2. When the outer loop goes to 4, it will try to eliminate the multiples of 4 but that’s unnecessary because all the multiples of 4 have already been eliminated in the loop of 2. Trying to avoid this would be ideal.

   One easy way is to add a condition if i is odd, only then check its bit value and eliminate the multiples. This is simple because at the first iteration of 2, it eliminates all the even numbers and their multiples as well if they are there. After all, they are also even. This saves us the time and the jump to the bv_get_bit() function. So this solution will look like this:

```
for i still sqrt(lenght):
        If i is odd or i == 2:
                If get_bit:
                        For k till length:
                                Clear bits at (k+i)*i
```

<u>Design Of Entire Program</u>

sequence.c:
main() deals with command line arguments in more or less the same way it did in assignment 2. Creates a bitvector and call sieve() on it to modify the prime indexes and then calls functions check_prime() if ‘-s’ is there and check_palindrome if ‘-p’ is there in the command argument.

The function check_prime_kind() creates a counter variable(mersene_counter
) to keep track of available space in *mersenne_num that allocates space in memory to store Mersenne numbers for every prime number. The mersenne_loading_counter is used to point at spaces at which Mersenne numbers that haven’t been printed yet. If a prime equals a Mersenne number at mersenne_loading_counter then you print it. After Mersenne, we check for lucas and Fibonacci in similar ways. There are two variables lucas_num[2 and fibonacci_num[2] that keep track of the last two numbers in the Fibonacci and Lucas series. When a prime is detected in the bitvector, the second number in the array is increased to the summation of the current two numbers in the array and the number at index 0 is changed to the number that was at index 1.

This goes on until the second number in the array is as big as the prime number or bigger. If it's equal to a prime number then you print Fibonacci or Lucas.

The function isPalindrome() is explained above in the pseudocode code. It simply

The function check_Palindrome() creates space and counter variables with to store numbers that have palindromes in base 2,9, 10, and 17(10 + letter in the alphabet of initial last name, in my case 'G'). For each index in bitvector, it checks if that index is prime then using the isPalindrome() functions, it checks its representation in bases 2, 9, 10, and 17 is palindromic and if it is then it adds it to the allocated space. Then prints the numbers that have been added in the allocated spaces using print_palindrome() function. The print_palindrome() function uses the itoa() function to convert a given number into a string of a base representation of that number and print it. A personal creation of an in-built itoa function in C++. Convert the given num into a string of base representation of that num
Got help for this function from here: https://www.geeksforgeeks.org/implement-itoa/

Bv.c: its pseudocode is given above. Used to create bitvector and its helper functions.

Sieve.c: taken from the pdf