Roleen Ferrer
Professor Maxwell Dunne
CSE 13S - Computer Systems and C Programming

Assignment 4: Bit Vectors and Primes

In this lab, I will be implementing certain functions in order to recreate the prime number list. In order to produce this list, I will be using the Sieve of Eratosthenes to generate a list of prime numbers and determine if they are also Fibonacci Prime Numbers (F), Lucas Prime Numbers (L) or Mersenne Prime Numbers (M). For each prime number I will also check if it is a palindromic prime number for the following bases:

- Base 2
- Base 9
- Base 10
- and Base F + 10, or Base 16 corresponding to the last letter of my last name.

The user will input commands in the command prompt to achieve the following:

1. -s: which will print out all of the primes and determine if they are of Fibonacci, Lucas, or Mersenne type
2. -p: which will print out the palindromic primes in the bases 2, 9, 10 and the first letter of my last name + 10 (16).
3. -n <value>: which will take a value from the user to consider in creating the prime sieve. If no "n" command is given, it will default to 1000.

PRE-LAB QUESTIONS

Part I

1) See IMPLEMENTATION - Main (sequence.c), bool lucasNumbersCheck, bool fibNumerscheck, and mersenneNumbersCheck
2) See IMPLEMENTATION - Main (sequence.c), void base10 and bool isPalindrome

Part II

1) See IMPLEMENTATION - Bit Vectors (bv.c)
2) In order to prevent memory leaks, I made sure to free elements inside the v->vector first before I freed v itself to ensure that the BitVector struct would be deleted. I also made

sure that the BitVector would also be freed when there would be an invalid uint32_t that would be passed in bv_create, to which it would free the BitVector.

3) A change I would make in the sieve code would be a condition to check if the multiple of some number is already taken. For example, when the function already checks for the index of 2, the function does not need to check for other multiples of 2, such as 4 or 6, since these would have been cleared from the first time I ran sieve().

IMPLEMENTATION

Bit Vectors (bv.c)

This source file is very important because it creates a BitVector struct that is called upon other source files such as sieve.c and sequence.c. The following functions are implemented as followed:

1. **typedef struct BitVector**
    a. which includes:
        i. uint8_t *vector; which will store the bits for the Bit Vector
        ii. uint32_t length; which stores the length of the Bit Vector
2. **BitVector *bv_create(uint32_t bit_len)**
    a. This function creates a Bit Vector when called
    b. Parameters:
        i. uint32_t bit_len, which takes in the length of the Bit Vector
    c. Using a malloc, create a Bit Vector struct to allocate the size of the Bit Vector
    d. Create conditionals that check if the Bit Vector struct was created successfully or unsuccessfully
    e. Set v->length to bit_len
    f. Set v->vector using:
        i. (uint8_t *)malloc(((bit_len / 8) + 1) * sizeof(uin8_t))
        ii. This will allocate memory to the use of a Bit Vector that can carry 8 bits per “bucket”, thus making the Bit Vector more cost efficient for this lab
3. **void bv_delete(BitVector *v)**
    a. This function will delete the Bit Vector and free memory

i. Parameters:
    1. BitVector *v, which takes in a Bit Vector
ii. Free v->vector allocated memory
iii. Then, clear v Bit Vector itself

4. **uint32_t bv_get_len(BitVector *v)**
    a. This function will return the length of the bitvector
        i. Parameters:
            1. BitVector *v, which takes in a Bit Vector
        ii. Return v->length
5. **void bv_set_bit(BitVector *v, uint32_t i)**
    a. This function will set a bit at some index to 1 (true)
    b. Parameters:
        i. BitVector *v, which takes in a Bit Vector
        ii. uint32_t i, which represents the index to set the bit
    c. Using bitwise operators, take the bit at the current index of the Bit Vector, shift 0x1 left by the index and OR each bit together.
        i. v->vector[i/8] = v->vector[i/8] | (1 << (1%8))
6. **void bv_clr_bit(BitVector *v, uint32_t i)**
    a. This function will clear a bit at some index to 0 (false)
    b. Parameters:
        i. BitVector *v, which takes in a Bit Vector
        ii. uint32_t i, which represents the index to set the bit
    c. Using bitwise operators, take the bit at the current index of the Bit Vector, shift 0x1 left by the index and bitwise NOT the result. Then, AND the bits together to clear it.
        i. v->vector[i/8] = v->vector[i/8] & ~(1 << (i%8))
7. **uint8_t bv_get_bit(BitVector *v, uint32_t i)**
    a. This function will return the bit at a given index
    b. Parameters:
        i. BitVector *v, which takes in a Bit Vector
        ii. uint32_t i, which represents the index to set the bit

c. This can be done by doing a bitwise operation AND on 0x1 shifted to the left by the index.
    i. return v->[i/8] & (1 << (i%8))

8. **void bv_set_all_bits(BitVector *v)**
    a. This function will set all the bits in a Bit Vector to 1 (true)
        i. Parameters:
            1. BitVector *v, which takes in a Bit Vector
    b. Create a for loop that will iterate through all of the index of the Bit Vector until it is greater than bv_get_len(v)
        i. for (uint32_t i = 0; i < bv_get_len(v) + 1; i++)
    c. Use bv_set_bit(v, i) to set each bit at the index.

The Sieve of Eratosthenes (sieve.c)

The Sieve of Eratosthenes will help filter out the composite numbers (0) and leave the prime numbers set (1) in a Bit Vector. The sieve works like this:

1. Set x equal to 2, which signifies the start of the sieve and the smallest prime number
2. Enumerate through multiples of p until there is not such number that can be enumerated with p
3. Repeat step 2 with the newly found prime number
4. The sieve will stop until it reaches a certain limit, which in this case, is set by length of the Bit Vector.

The code for this sieve was given in the Lab Document of this lab, to which I used and gave proper credit to:

```
//
// The Sieve of Erastothenes
// Sets bits in a BitVector representing prime numbers.
// Composite numbers are sieved out by clearing bits.
//
// v: The BitVector to sieve.
//
void sieve(BitVector *v) {
  bv_set_all_bits(v);
  bv_clr_bit(v, 0);
  bv_clr_bit(v, 1);
  bv_set_bit(v, 2);
  for (uint32_t i = 2; i < sqrtl(bv_get_len(v)); i += 1) {
    // Prime means bit is set
    if (bv_get_bit(v, i)) {
}
```

Main (sequence.c)

This source file contains the main functions that will be used to create and print the list of primes and create and print the list of palindromic primes in bases 2, 9, 10, and 16. The follow function implementations are as followed:

1. **int main(int argc, char **argv)**
    a. This main function will take in the commands from the user, in which they are able to set the amount of primes (n), print out the list of primes and their special primes (s), or print out the list of palindromic primes in bases 2, 9, 10, and 16 (p).
    b. Using getopt() will take the commands from the user.
    c. Create an int n that is set default to 1000 if the user does not input some n in the command line
    d. Create to boolean data types that are set to false
        i. This will keep track of which commands are input by the user
            1. bool print_primes = false, print_palindromes = false
    e. Create conditionals that will call functions to print the output.
        i. In each conditional, create the Bit Vector and pass it through the sieve to generate prime numbers
        ii. Call their respective helper functions to print out the correct output corresponding to the Bit Vector
        iii. Delete the Bit Vector struct to free up memory.
2. **void printPrimes(struct BitVector *primes, uint32_t numbers)**
    a. This function will be called when the print_primes = true (the user inputs the command). It will print out the primes and respective special primes.
    b. Parameters:
        i. struct BitVector *primes, which takes in a Bit Vector
        ii. uint32_t numbers, which takes in the amount of numbers the Bit Vector will find primes up to

c. Create a for loop that will iterate through the Bit Vector that has already been passed through the sieve.
    i. if the bv_get_bit(primes, index) is cleared (0), continue
d. else , print the prime number to the output
e. Create three boolean data types that will check if the following prime number is either a Mersenne, Fibonacci, or Lucas number.
    i. This will be done by calling helper functions to check if it is a special prime
f. If the boolean expression is true, print the special prime number type

3. **bool lucasNumbersCheck(double prime_number) (Helper Function)**
    a. This function find each lucas number up to a certain number that is < uint32_t and compare it to the given prime number and will return True if it matches
    b. Parameters:
        i. double prime_number, which takes in a prime number
    c. Declare variables first, second, and next to symbolise the iteration of finding the Lucas number
    d. Create a for loop that will iterate through each Lucas Number and compare it to the prime number
        i. The next variable will equal the first + second,
        ii. The first variable will equal the second
        iii. and the second variable will equal next
    e. Return True if it matches otherwise return False
4. **bool fibNumbersCheck(double prime_number) (Helper Function)**
    a. This function find each Fibonacci number up to a certain number that is < uint32_t and compare it to the given prime number and will return True if it matches
    b. Parameters:
        i. double prime_nubmer, which takes in a prime number
    c. Declare variable fib_number = 0, n_minus_2 = 1, and i
    d. Create a for loop that will iterate through each Fibonacci Number and compare it to the prime number

i. The i variable will equal the fib_numer + n_minus_2
ii. The fib_number variable will equal n_minus_2
iii. The n_minus_2 variable will equal i

e. Return True if it matches otherwise return False

5. **bool mersenneNumbersCheck(double prime_number) (Helper Function)**
    a. This function find each Fibonacci number up to a certain number that is < uint32_t and compare it to the given prime number and will return True if it matches
    b. Parameters:
        i. double prime_nubmer, which takes in a prime number
    c. Create a for loop that will iterate through each Mersenne Number and compare it to the prime number
        i. for (double n = 0; n <= 32; n++)
        ii. For each n, use pow(2, n) - 1 and compare it to the prime number
    d. Return True if it matches otherwise return False
6. **bool isPalindrome(uint_8 *array, int top_of_array)**
    a. This function will check if the current array is a palindrome. The array will be set from their respective Base functions (2, 9, 10, 16). This function was also created with the help of the pseudocode from the lab document.
    b. Parameters:
        i. uint_8 *array, which takes in the array that holds the palindromic integers
        ii. int top_of_array, which takes in the top of the array, or highest index
    c. Create a for loop that ill iterate through half of the array and check if it will equal the other half, in order
        i. for (uint8_t i = 0; i <= (top_of_array / 2); i += 1)
        ii. Create a conditional that will check if the start of the array against the end of the array.
            1. If they do not equal, return False
        iii. else, return True.
7. **void Base2(struct BitVector *primes, uint32_t numbers)**

a. This function will print out all of the prime numbers from the given n value that have a palindrome at Base 2.
    i. Parameters:
        1. struct BitVector *primes, which takes in a Bit Vector
        2. uint32_t numbers, which takes in the amount of numbers the Bit Vector will find primes up to
b. Create a for loop that will iterate through the Bit Vector that has already been passed through the sieve.
    i. if the bv_get_bit(primes, index) is cleared (0), continue
c. else , create an array that can hold 32 bits, which is enough bits to accommodate uint32_t bits in binary (base 2)
d. Declare a variable “prime” that takes in the current i from the for loop
e. Set an index variable to 0
f. Create a while loop that will iterate through the prime number as long as it is greater than 0.
    i. array[index] = prime % 2, which is the reminder of the current quotient
    ii. prime = prime / 2
    iii. index += 1
g. Check if the current array is a palindrome by passing it through isPalindrome(array, index) helper function.
h. If it is a palindrome, print out the prime number followed by the palindrome in base 2.

8. void Base9(struct BitVector *primes, uint32_t numbers)

a. This function will print out all of the prime numbers from the given n value that have a palindrome at Base 9.
    i. Parameters:
        1. struct BitVector *primes, which takes in a Bit Vector
        2. uint32_t numbers, which takes in the amount of numbers the Bit Vector will find primes up to
b. Create a for loop that will iterate through the Bit Vector that has already been passed through the sieve.

i. if the bv_get_bit(primes, index) is cleared (0), continue

c. else , create an array that can hold 32 bits, which is enough bits to accommodate uint32_t bits
d. Declare a variable “prime” that takes in the current i from the for loop
e. Set an index variable to 0
f. Create a while loop that will iterate through the prime number as long as it is greater than 0.
    i. array[index] = prime % 9, which is the reminder of the current quotient
    ii. prime = prime / 9
    iii. index += 1
g. Check if the current array is a palindrome by passing it through isPalindrome(array, index) helper function.
h. If it is a palindrome, print out the prime number followed by the palindrome in base 9.

9. void Base10(struct BitVector *primes, uint32_t numbers)

a. This function will print out all of the prime numbers from the given n value that have a palindrome at Base 2.
    i. Parameters:
        1. struct BitVector *primes, which takes in a Bit Vector
        2. uint32_t numbers, which takes in the amount of numbers the Bit Vector will find primes up to
b. Create a for loop that will iterate through the Bit Vector that has already been passed through the sieve.
    i. if the bv_get_bit(primes, index) is cleared (0), continue
c. else , create an array that can hold 32 bits, which is enough bits to accommodate uint32_t bits
d. Declare a variable “prime” that takes in the current i from the for loop
e. Set an index variable to 0
f. Create a while loop that will iterate through the prime number as long as it is greater than 0.
    i. array[index] = prime % 10, which is the reminder of the current quotient

ii. prime = prime / 10

iii. index += 1

g. Check if the current array is a palindrome by passing it through isPalindrome(array, index) helper function.

h. If it is a palindrome, print out the prime number followed by the palindrome in base 10.

10. void Base16(struct BitVector *primes, uint32_t numbers)

a. This function will print out all of the prime numbers from the given n value that have a palindrome at Base 2.

i. Parameters:

1. struct BitVector *primes, which takes in a Bit Vector
2. uint32_t numbers, which takes in the amount of numbers the Bit Vector will find primes up to

b. Create a for loop that will iterate through the Bit Vector that has already been passed through the sieve.

i. if the bv_get_bit(primes, index) is cleared (0), continue

c. else , create an array that can hold 32 bits, which is enough bits to accommodate uint32_t bits

d. Declare a variable “prime” that takes in the current i from the for loop

e. Set an index variable to 0

f. Create a while loop that will iterate through the prime number as long as it is greater than 0.

i. array[index] = prime % 16, which is the reminder of the current quotient

ii. prime = prime / 16

iii. index += 1

g. Check if the current array is a palindrome by passing it through isPalindrome(array, index) helper function.

h. If it is a palindrome, print out the prime number followed by the palindrome in base 16.

i. For each index, if it is larger than or equal to 10, print out the character from the ascii character chart.

ii. else, print out the number instead