DESIGN
Bit Vectors and Primes

TASK:
Generate a list of prime numbers up to a value n using a sieve, then go through each number and determine if it is a Fibonacci Prime, Lucas Prime, or Mersenne Prime.
Also be able to generate a list of prime numbers up to a value n using a sieve and then check if it is a palindrome prime for the following bases:
- Base 2
- Base 9
- Base 10
- Base 35

APPROACH:
Use getopt() to determine if user wants to find sequence primes or palindrome primes. Generate the list of primes using the Sieve of Erastothenes as well as the BitVector ADT given with the bv.h file.
Sequence: Check each prime generated and determine whether it is a Fib, Luc, and/or Mer with bool functions
Palindrome: Convert each prime into different specified base and print them if they are a palindrome using bool palindrome function given in lab manual

PRE-LAB 2:
1.
```
bv_create(bit_len)
  malloc memory the size of the BitVector
  set bit length to input bit_len
  set vector to calloc(bit_len / 8 + 1, 8 bits)
  return v;

bv_delete(v)
  free v's items
  free v

bv_get_len(v)
  return v->length

bv_set_bit(v, i)
  set i to the remainder of i / v->length
  set byte to vector[i / 8]
  set position to remainder of i / 8
  set vector[i / 8] to (byte OR position shifted left 1)

bv_clr_bit(v, i)
  set i to the remainder of i / v->length
  set byte to vector[i / 8]
  set position to remainder of i / 8
  set mask to NOT of (position shifted left 1)
  set vector[i / 8] to (byte AND mask)

bv_get_bit(v, i)
  set position to remainder of i / 8
  return (vector[i/8] AND (position shifted left 1) shifted right position)

bv_set_all_bits(v)
  for 0 - length of v
    bv_set_bit(v, i)
```

2. Memory allocated for the BitVector ADT can be used again and is not reserved for no reason, which would cause a leak.
3. Since sieve() is a nested for loop, you should optimize the inner loop as well as using a post-condition instead of a precondition to loop one less time.

PRE-LAB 1:
1.
```
bool fib_test(input number) {
  set first and second number to 0 and 1
  set third number to (first + second)

  if input is 0 or 1, return true

  else, while the third number is less than
    the input, find the next number in the
    fib sequence by making the second number
    the first and the third number the second'

  if the third number is equal to the input,
    input is a fib number, so return true

return false
}

bool luc_test(input number) {
  set first and second number to 2 and 1
  set third number to (first + second)

  if input is 1, 2, or 3 return true

  same logic for fib_test applies from then on
}

bool mer_test(input number) {
  for loop generating mersenne numbers
     (incrementing i from 0 and setting mersenne
      number = pow(2, i) - 1)
    if mersenne number is greater than input,
      return false
    else if mersenne number = input number,
      return true
}
```

2.
input will be string after base conversion

```
bool palindrome_test(input string) {
  while the left index is less than the right.
    if the left character is not equal to the right character,
      return false
    increment left index and decrement right index
    to move towards middle
  return true if while loop ends, signaling that
  each left char matches the right
}
```

General Pseudocode

```
int main(arguments) {
  while parsing through arg,
    if -n #, set # to largest number to be considered (length of BitVector),
      otherwise, default is 1000
   if -s, set sequence to true
   if -p, set palindrome to true

  if sequence is true,
    create a BitVector with n as its length
    sieve BitVector to set prime bits as 1
    iterate through BitVector, printing prime numbers
    for each prime number,
       print mersenne if mer_test(prime #) is true
       print lucas if luc_test(prime #) is true
       print fibonacci if fib_test(prime #) is true
    delete BitVector, freeing it from the heap

  if palindrome is true,
    create a BitVector with n as its length
    sieve BitVector to set prime bits as 1
    create conversion array to store converted numbers from different bases
    print header for Base 2
    iterate through BitVector
      if current # is prime,
        if palindrome_test(# converted to base 2) is true
          print (number = to its conversion)

    follow same logic in Base 2 for Base 9, 10, and 35 ('Y' + 10)
  }
```

```
char convert_base(conversion array, number, base)
  set quotient = number
  while the quotient is greater than 0
     remainder = quotient % base
     if the remainder is 0-9, add corresponding # to conv array
     else, add corresponding letter to conv array
     quotient / = base
     increment position in conv array to move onto next digit
add a null character after last converted digit in array
reverse the array as it conversion array is backwards
return conversion array
```

Design Process

Creating the fib, luc, and mer tests were fairly easy as I just had each of my helper functions create and continue each sequence, always checking if the input was in it where it would return true or the sequence past it, returning false.

I was confused as to how I should convert bases and check if they were palindromes. I knew I had to input the number as a string in order for the palindrome function to work, but I was struggling to figure out how to implement characters when converting into bases. I didn't want to use an array filled with letters and choose a specific one during the conversion process as it was brute force and i would basically have to fill an entire array with the alphabet. I decided to use asci characters and their corresponding numbers to implement the letters. During the conversion process, if a number was greater than 9, I would add (that number - 9) and the char " ` ", which would return the corresponding lowercase letter. For example, if the number was 10, the process would be (10 - 9) + " ` ", which would equal " a ".

I was able to generate prime numbers using the BitVector ADT and the Sieve of Erastothenes; the BitVector would create an array with a designated legnth. The sieve would then check each index of the BitVector array and if it was a prime number, it would set the corresponding bit to 1. When trying to find prime numbers, I would iterate through the BitVector, starting at the 2nd index as I know 0 and 1 aren't prime, and then check if each corresponding bit was 1, indicating that its index was a prime number.

Notes

bv.h, bv.c, sieve.h, and sieve.c were given and derived from Asgn4 lab manual

palindrome_test was derived from https://www.geeksforgeeks.org/c-program-check-given-string-palindrome/
  changed function to be bool type and return true or false

convert_base logic was derived from https://www.tutorialspoint.com/computer_logical_organization/number_system_conversion.htm
  used same conversion system to convert any decimal into any base