Pre-lab part 1

1.

This function returns the next value in the fibonacci prime sequence.

```
Uint32_t nextfib(uint32_t n) {
        Uint32_t t1 =1; t0 = 2, nextnum = 0
        While nextnum<n {
                Nextnum = t1 + t0
                T0 = t1
                T1 = nextnum
        }
        Return nextnum
}
```

This function returns the next value in the lucas prime sequence.

```
Uint32_t nextluc(uint32_t n) {
        Uint32_t t1 = 1, t0 = 2, nextnum = 0
        While (nextnum < n) {
                Nextnum = t1 +t0
                T0 = t1
                T1 = nextnum
        }
        Return nextnum
}
```

This function returns true if the integer n is mersenne prime and false otherwise

```
Bool mersenne(uint32_t t) {
        For uint32_t i =0 to i is less tahn n while i increments {
                Uint32_t v = i +1, p2 = 2
                For uint32_t a =0 til a less than v while a increments {
                        P2 *= 2
                }
                If (p2-1 > n) {
                        Return false
                }
                Else if (p2-1 == n) {
                        Return true
                }
        }
        Return false
}
```

Prints out if the digit is a mersenne, lucas or fibonacci prime

```
Void isinterest(BitVector *v) {
    For uint32_t i =0 til i < bv_get_len(v) while i increments {
        If bv_get_bit(v, i) == 1 {
            Print i prime
            If (mersenne(i)) {
                Print , mersenne
            }
            If (nexluc(i) == i) {
                Print , lucas
            }
            If (nexfib(i) == i) {
                Print , fibonacci
            }
            Print new line
        }
    }
}
```

Pre-lab part 2

1.

This function creates a new bitvector

```
BitVector *bv_create(uint32_t bit_len) {
    BitVector *newBitVector = (BitVector*) allocate space of(sizeof(BitVector))
    newBitVector->vector = (uint32_t*) allocate space of (8*bit_len)
    newBitVector->length = bit_len
    Return BitVector
}
```

This function deletes the allocated space of the bitvector

```
void bv_delete(BitVector *v) {
    Frees the v
    Frees the v vector
}

Uint32_t bv_get_len(BitVector *v) {
    Return v length
}

Void bv_set_bit(BitVector *v, uint32_t i) {
    Int f=i/8
    Double bf = i%8
```

```
        Int bf1 = (int)bf
        V vector[f] |= 1 << bf1
}

Void bv_clr_bit(BitVector *v, uint32_t i) {
        Int f=i/8
        Double bf = i%8
        Int bf1 = (int)bf
        V vector[f] &= ~(1 << bf1)
}

Uint8_t bv_get_bit(BitVector *v, uint32_t i) {
        Int f=i/8
        Double bf = i%8
        Int bf1 = (int)bf
        Return V vector[f] >> bf1 &1
}

Void bv_set_all_bits(BitVector *v) {
        For uint32_t i = 0 til i < v length while i increments {
                bv_set_bits(v, i)
        }
}
```

2. In order to avoid memory leaks, you need to dereference the pointer after freeing the allocated memory space of the BitVector ADT. Memory leak occurs because if you just free it, a pointer would still exist that points to that address.

3. There is unnecessary runtime attributed to setting the 2nd bit in the bitvector while all bits have already been set. Therefore setting the second bit is totally unnecessary and would increase the runtime.

```
Void main (int argc, character **argv) {
        int c is defined
        Int num = 1000
        Bool interest = false, palin = false
        BitVector* prime = null
        While the c = getopt of argc, argv, ‘psn:’ does not equal -1 {
                Switch of c {
                        Case of ‘s’:
                                Interest = true
                                break
                        Case of ‘p’:
```

```
                    Palin = true
                    break
                Case of ‘n’:
                    Num = atoi(optarg)
                    If num == 0{
                        Print Error: Please enter a vald “n” value \n
                    }
                    break
                default :
                    break

            }
        }
        Prime = bv_create(num)
        sieve(prime)
        If isinterest {
            isinterest(prime)
        }
        If palin {
            pcase(prime)
        }
        Return 0
}
```

This function returns true if the char* s is a palindrome and false otherwise

```
Bool isPalindrome(char *s) {
        Bool f = true
        For int i = 0 to i <= (int) (strlen(s)/2) while i increments {
            If (s[i] != s[strlen(s)-i-1]) {
                F = false
            }
        }
        Return f
}
```

This function changes the base of any uint32_t n to the desired base

```
Char* tobase(uint32_t u, int base) {
        char* based = allocate(64)
        char* tempc = allocate(64)
        Uint32_t remainder = 0
        Bool asci = false
        If (base > 10 && u > 10) {
            Asci = true
        }
        While u!=0 {
            Remainder = u%base
            U /= base
            If asci = true && remainder >= 10 {
                    Remainder += 87
                    sprintf(tempc, “%d”, remainder)
            }
            Else {
                    sprintf(tempc, “%d”, remainder)
            }
            strcat(based, tempc)
        }
        Return based
}
```

This function Prints out all the palindrome primes

```
Void pcase(BitVector *v) {
        Int bases[] = {2, 10, 14, 24}
        For int p = 0 to p<4 while p increments {
                Print base bases[p]
                Print ---- --\n
                For uint32_t i = 0 til i < bv_get_len(v) while i increments {
                        If bv_get_bit(v, i) == 1 {
                                char* palintest = tobase(i, bases[p])
                                If isPlaindrome(palintest){
                                        Print i = tobase(i, bases[p])
                                }
                        }
                }
                If p < 3 {
                        Print newline
                }
        }
}
```

This function sets the prime number index bits to 1

```
//code by DDEI
#include "sieve.h"
#include "bv.h"
#include <math.h>

//
```

```
// 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)) {
      for (uint32_t k = 0; (k + i) * i <= bv_get_len(v); k += 1) {
        bv_clr_bit(v, (k + i) * i);
      }
    }
  }
  return;
 }
```