Despina Patronas
Assignment 4 Bit Vectors and Primes
Design Document

Pre-Lab Part 1:

**Fibonacci prime:**
**//There is 16 known Fibonacci # between [2,2000+],**

```
// Create a array of known fib numbers starting at 2 to compare with the known primes
// fib[16] = {2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597}     [2,2000+]

//function to initialize the known_fibs array
Int Fibonacci ( int index, int elem1, int elem2)

        If (index == 0)             // fib[index==0] = 2
                Return 2

        Or if (index == 1)          // fib[index==1] = 3;
                Return 3

        Otherwise                   // fib[index>1] = (index-1) + (index-2)
                Return elem1+elem2
End Fibonacci
```

**//Finding the BitVector fibonacci primes snippet code:**

```
        Declare int known_fib[16] ={0}          // set and initialize array for fib numbers

        Iterate from int index [0,16)           // set the fib array to fib values
                Set known_fib[index] = Fibonacci( index,  known_fib[i-1], known_fib[i-2])
        End iteration

        Declare int length                      // upper range of numbers to test for primality
        Declare int index = 2                   // lower range of numbers to test for primality [2,length]
        BitVector *vector = create_BV(length);  // create the bitVector
        Sieve(vector)                           // finds the primes, toggles the prime bit to 1

        //Find Prime Loop
        Iterate from [index, length]

                //If Prime block
                If find_bit(vector, index) == 1         //if the index bit of vector is 1 when it is prime
                        Display index and prime

                        //Find Fib Prime Loop
                        Iterate from f_index [0,16)                     //access array of Fibonacci primes
                                If ( known_fibs [f_index] == index )    // compare to the known prime
                                        Display Fibonacci found
                                        Increment f_index
                                End if
                        End iteration f_index
                EndIf
```

**<u>Lucas prime:</u>**

**<u>//Similar to Fibonacci numbers, there are 16 known Lucas numbers between [2,2000]</u>**
//so we will follow similar steps to determine Lucas primes from a Lucas array list
//assume this is an addendum to the above..

**<u>//Finding the BitVector Lucas primes snippet code:</u>**

```
Declare and initialize known_Lucas[16] = {2,3,4,7,11,18,29,47,76,123,199,322,521,843,1364,2707}

// within find prime loop (as seen above)
    //within if prime block (as seen above)

        //Find Lucas Prime Loop
        Iterate from L_index[0,16]                         //access array of Lucas primes

            If ( known_Lucas [L_index] == index )      // compare to the known prime
                Display Lucas found
                Increment L_index
            End if
        End L_index iteration
```

**Mersenne Prime:**

```
//Mersenne # Formula is (2^n) -1
	//since both ((2^0) -1) and ((2^1)-1) are 1 we can exclude these values
	//since 2^10 = 1024				we can exclude this value
//There are 8 viable Mersenne numbers between [2,1000] and the range of Mersenne is from [2,9]


//Helper power function
//determine a number base^power

Pow (int base, int power)
	Int result = 1				//base case x^0 = 1
	Iterate from [int I =0 to I < power)	//[0,power] to calculate the amount of time to multiply base by
		Result = result * base
	End iteration
	Return result
End power


//function to determine the Mersenne number based on input (input is the power)

Mersenne (int input)

	Return (Pow(2,input)-1)
End Mersenne
```

**//Finding the BitVector Mersenne primes snippet code:**

```
	// within find prime loop (as seen above)
		//within if prime block (as seen above)

			//Find Mersenne Prime Loop
			Iterate from M_index[0,13]			//access array of Lucas primes

				If (  Mersenne(m)== index )	// compare to the known prime
					Display Mersenne found
					Increment M_index
				End if
			End L_index iteration
```

**Pre-Lab Part 1**
**2)**

**Determine if (prime number) in Base 10 is palindrome:**

//first do the base change

//when finding the prime numbers, we print the decimal prime (which is the index)

//converting to base 1-0 is not required

| base 9 | quotient | remainder(decimal) |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |

**<u>Pre-Lab Part 2</u>**

**1)**

```
//Bit Vector Function Implementation

//Ceiling function for creating vector field
Ceiling(real n)
        //if truncated n is less than the decimal number n
        If (casted int type(n) < real n)
                //return the ceiling which is truncated n +1
                Return casted int type(n) +1
        //otherwise the two values are equal
        Otherwise
                //return the truncated n (floor of n)
                Return casted int type n
        Endif
End ceiling

//create the bitvector and initialize the fields
BitVector *create( pos int input_length)

        allocate new_ vec of type Bitvector on heap (size of 32 positive int bits)

        //check the creation is successful
        If (!new_vec)
                Return NULL
        Endif

        Set new_vec field length = input_length

        allocate new_ vec  field vector of type (positive int 8 bit) on heap (with size of 8 positive int bits* Ceiling(
                                                                                                    new_vector length /8 )
        //check the creation is successful
        if (!new_vec)
                return NULL
        endif
        return new_vec

end BitVector create

//delete the BitVector that is on the heap
Delete( BitVector *new_vec)

        //delete allocated memory of the array in new_vec
        Free(new_vec vector)

        //delete allocated memory of the Bitvector new_vec
        Free(new_vec)

End delete
```

//return the length in bits of the bitvector
**Positive int getLength (BitVector *new_vec)**

Return new_vec field length

End getLength

//set the bit at index in BitVector
**Setbit( BitVector *new_vec, positive in index)**

//accessing the byte element of the vector field
Declare Int Byte = index / 8

//access the bit element of vector field
Declare int bit = index MOD 8

// vector[byte] OR with 1 shifted left by the bit amount
Set new_vec vector[Byte] = new_vec vector[Byte] OR ( 1 << bit)

```
example:
      1010
   OR 0100  <- 0001 << 3 (index bit)
___________
      1110    index bit is set to 1
```

End Setbit

//Clears the bit at index in the BitVector from 0->1
**ClearBit( BitVector *new_vec, positive in index)**

//accessing the byte element of the vector field
Declare Int Byte = index / 8

//access the bit element of vector field
Declare int bit = index MOD 8

//& with inverse of (1 << bit)
Set new_vec vector [Byte] = new_vec vector[Byte] AND (NOT(1 << bit))

```
example:
       1100
  AND 1011  <- the inverse of ( 0001) << 3 (index)
____________
     = 1000  the index bit is cleared
```

End ClearBit

//Gets a bit from a BitVector.
**Positive int GetBit( BitVector *new_vec, positive in index)**

//accessing the byte element of the vector field
Declare Int Byte = index / 8

//access the bit element of vector field
Declare int bit = index MOD 8

//vector[byte] >> (bit & 1)
Return new_vec vector [Byte] >> (bit AND 1 )

```
example:
         0001 <- 0100 >> 3 (the index bit)
  AND  0001
  __________
       = 0001 (this would be 0 if we started with 1000)
```

End GetBit

//Sets all bits in a BitVector to 1
**SetAll( BitVector *new_vec, positive in index)**

```
//accessing the byte element of the vector field
Declare Int Byte = index / 8

//access the bit element of vector field
Declare int bit = index MOD 8

//set all bits to one in the BitVector
Iterate through index [0, new_vec length]
    SetBit(new_vec, index)
```

End SetAll

**2) How to avoid memory leaks when freeing allocated memory for BitVector ADT**

**Delete the Bitvector field vector array first**

**Then delete the BitVector itself**

**3) How to improve the sieve() algorithm?**