Despina Patronas
Assignment 4 Bit Vectors and Primes
Design Document

Pre-Lab Part 1:

Fibonacci prime:

```
//Will test 20 fib numbers since tests are viable length
// Create a array of known fib numbers starting at 2 to compare with the known primes

//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
Test_Primes ( int length, Bitvector * new_vec)

        Declare int known_fib[20] ={0}          // set and initialize array for fib numbers

        Iterate from int index [0,20)           // 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 [0,20)                     //access array of Fibonacci primes
                                If ( known_fibs [f_index] == index )    // compare to the known prime
                                        Display Fibonacci found
                                        Increment f_index               //check for the next f_index
                                End if
                        End iteration f_index
                EndIf

        End Find Prime Loop Iteration
```

Lucas prime:

//Similar to Fibonacci numbers will test 20 lucas numbers excluding 1 (since we know 1 is not a prime)
//so we will follow similar steps to determine Lucas primes from a Lucas array list

//Finding the BitVector Lucas primes snippet code:
//continuation of test_primes as seen above

```
//declare outside of loop
Declare and initialize known_Lucas[20] = {2,3,4,7,11,18,29,47,76,123,199,322,521,
                                          843,1364,2207,3571,5778,9349,15127}

// within find prime loop (as seen above)

    //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

        Iterate from L_index[0,20)                     //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

    End if prime block

End within find prime loop
```

Mersenne Prime:

```
//Mersenne # Formula is (2^n) -1
        //since both ((2^0) -1) and ((2^1)-1) are 1 we can exclude these values
//due to variable tests will go up to 15 mersenne numbers

//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 (continuation of function test_primes as seen above)

                //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 Mersenne Prime Loop
                        Iterate from M_index[0,15]              //access array of Lucas primes

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

                        End M Prime Loop iteration

                End if Prime Block

        End Find Prime Loop

End Test_Prime
```

Pre-Lab Part 1
2)

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

//in order to determine palindrome, will use a char string and iterate through the elements
//in order to do this will need to convert the integer into a char
//for the base, we technically do not have to do a base change since the index where the prime is found is base 10

base 16	quotient	remainder (decimal)	remainder (hexadecimal)
7562/16	472	10	A
472/16	29	8	8
29/16	1	13	D
1/16	0	1	1

//Step 1: build char string in desired base
this function will build a char string from
//input decimal integer converted into input base

```
Buildstring (char str array, int input, int base )
        Declare int index = 0
        While (input > 0)                                   //keep finding remainders while quotient > 0
                Set remainder = input MOD base
                If ( remainder >= 0 AND remainder <= 9 )    //check remainder range for value [0-9]
                        Str[index++] = casted char (remainder + '0')   //set element to char digit
                Otherwise                                   //if not in range [0,9] use hexa values
                        Str[index++] = casted char(remainder – 10 + 'A') // element base ascii char = 'A'
                Endif
                Input = input/base                          //divide input by base and set
        end while
        str[index] = NULL                                   //the last element of the string = 0
        return str
end Buildstring
```

//Check if the created string is a palindrome
// Inspired by the assignment 4 lab manual snippet:

```
def isPalindrome(s):
    f = True
    for i in range(len(s) / 2):
        if s[i] != s[-(i + 1)]:
            f = False
    return f

w = raw_input("word = ")
if isPalindrome(w):
    print w, "is a palindrome"
else:
    print w, "is not a palindrome"
```

isPalindrome(char str array)

boolean res = 1

set end = length of string(str) -1

set start = 0

Iterate from [0,end/2]

If (str[start] != str[end]

Res = 0

Endif

Increment start

Decrement end

End iteration

Return res

End isPalindrome

Pre-Lab Part 2

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)

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

//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)

End Setbit

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

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

//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))

End ClearBit

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

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

//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)

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
Delete any extraneous heap allocations (example arrays made for palindrome testing)

3) How to improve the sieve() algorithm?

The sieve function could be improved by eliminating the redundancy of setting bit 2 twice.

The rest of the program

```
//bv.h, sieve.h and sieve.c are defined based on the lab manual for assignment 4
//in sequence.c helpful functions to use:

//function to print the character to string
Printstring ( char str array )

        Int str_length = strlen(str)
        Display “String: “

        Iterate from [0,str_length)
                Display string array element[iteration]
        End iteration
```

End printstring

```
//function for the palindromic primes test
Test_palindrome( int length, BitVector * new_vec)

        //Declare x char strings allocated on the heap for your integer base tests
        Char * string_binary = (casted character pointer) allocate ( 10 * sizeof(char))

        Declare index = 2                       //the start of prime numbers
        While ( index < length )

                If (getBit(new_vec, index) == 1)

                        Buildstring( string_binary, index, 2)       //build for the newly created char array at index prime
                                                                    //appears for base 2

                        If ( isPalindrome ( string_binary )
                                Printstring( string_binary)
                        End if

                End if
                Index++

        End while

        //free the heap
        Delete( new_vec)        //the function in bv.h
        free (string_binary)    //your languages deallocate heap function

end test_palindrome


//testing the prime and special primes
Test_Primes( int length, Bitvector *new_vec)

        //the implementation for the body aka: each special prime test can be seen on the 1st 2nd and 3rd  page

        //important to not forget to delete the new_vec when function is finished !!

        Delete( new_vec)

End test_primes
```

```
//finally the main which includes the getopts and calling of functions

#Define OPTIONS “spn:”

Int main ( int argc, char ** argv )

        Declare c = 0
        Declare length = 1000    //default

        //optargs
        Declare Boolean spec_Primes = F
        Declare Boolean palnPrimes = F

        Char *input_length = NULL

        While ( ( c = getopt(argc, argv, OPTIONS )) != -2 )

                Condition Case (c )

                        Case ‘s’                        //run Test special primes
                                spec_Primes = T
                                break
                        Case ‘p’                        //run Test palindrome primes
                                palnPrimes = T
                                break
                        Case ‘n’                        //set length to user input
                                Input_length = optard

                                If (casted int (char input_length) > 0 )     //length shouldn’t be negative or 0
                                        Set length = casted int (char input_length)
                                End if
                                Break
                End Condition Case
        End while

        BitVector *new_vec = create(length)     //create the bitvector and array on heap with the length

        Sieve(new_vec)                          //generate the primes

        If (argc ==1)               //check bad case
                Return -1
        endif

        If (specPrimes)             // if true run the test for special primes
                Test_Primes(length, new_vec)
        endif

        If (palnPrimes)
                Test_palindrome( length, new_vec)
        End if
        Return 0

End main
```