Assignment 4 Design - Jack Vento

Pre Lab

Part 1

1) Assuming you have a list of primes to consult, write pseudo-code to determine if a number is a Fibonacci prime (F), a Lucas prime (L), and/or a Mersenne prime.

My algorithm calculates all three identifiers in one pass. By starting with inital values closest to my first prime p = 2 (F = L = 2, M = 3) for our three types of primes and also tracking the previous iterations for F/L, we can avoid space inefficiency and having to do multiple loops through the Bit Vector. Then during my loop, it calculates the next F/L/M values up to my current prime using the old values. If these new values are prime, print the identifiers.

- prev_fib = 1, prev_lucas = -1
- fib = 2, lucas = 2, mers = 3
- for each prime p = [2,n]:
    - while (mers < p)
        - mers = (++mers * 2) - 1
    - if p = mers
        - print ", mersenne"
    - Print header
    - while (fib < p)
        - temp = fib
        - fib = fib + prev_fib
        - prev_fib = temp
    - if p = fib
        - print ", fibonacci"
    - Follow fibonnaci pattern for lucas.

2) Assuming you have a list of primes to consult, write pseudo-code to determine if a number in base 10 is a palindrome. Note that the technique is the same in any base.

For each number (i) in the BV, I check if it prime using bv_get_bit. If it is, pass the prime (p) and the base we are currently operating with to palindrome().

- Print the header
- for each 32 bit prime (p) in bv:
    - palindrome(to_base, p)

Palindrome(to_base, p) Design & Pseudocode:

Part 1: First of all, I will need a string to store my number conversion, and since stack strings are read only, I will allocate a string on the heap. And since we need the size of this string prior to filling it, I calculated the length using logarithms. During an internship interview the other day, I learned a neat trick for calculating digits in a number. By taking the $\log_{base}(num)$ and adding 1, you get the amount of digits in a number (num)

in whatever base you're working in (base). However, we cannot directly take $\log_2$ or $\log_{32}$, so I must perform a change of base using log division. I will also add one more to length (making it a total of +2) to account for the '\0' in C strings.

Pseudocode:

- Calculate length = 2 + $\log_{10}$(p) / $\log_{10}$(to_base)
- Allocate str of size length on the heap

Part 2: Now, using the division/mod method provided in the lab manual, I will perform a change of base and stored the result in a properly formatted C string with a null character. Notably, I will add 87 to alphabetical remainders (remainders > 9) and 48 to numerical remainders before storing it in str. (87 will offset the remainder to the start of lowercase ASCII letters and 48 will offset it to the start of ASCII numerals).

Pseudocode:

- quotient = p
- remainder = 0
- for i in range [0, length - 1]:
    - remainder = quotient % to_base
    - if (remainder > 9)
        - Store remainder + 87 in str[i]
    - else
        - Store remainder + 48 in str[i]
    - quotient = quotient / to_base
- str[len - 1] = '\0'

Part 3: Credit to the lab manual for providing a way to check the converted string. However, I did change it to account for C style strings. I subtracted two from len - i to account for the null character '/0' and the zero-based nature of C strings.

Pseudocode:

- palindrome = true
- for i in range [0, length/2]:
    - if s[i] != s[length-i-2]
        - palindrome = false
        - Break from the loop since one error means no furtherchecking needed
- if (palindrome):
    - print the prime and the converted palindrome
- Free memory allocated for str
- return

Part 2

1) Implement each BitVector ADT function.

See "Bit Vector" section for design and `bv.c` for implementations.

2) Explain how you avoid memory leaks when you free allocated memory for your BitVector ADT.

The best way to avoid memory leaks is to always call an ADT's destructor when it's no longer needed or just before it goes out of scope. In that destructor, all memory allocated inside the ADT (in the BV's case: a vector) needs to be freed and null'ed before the ADT itself is freed and null'ed.

3) While the algorithm in sieve() is correct, it has room for improvement. What change would you make to the code in sieve() to improve the runtime?

Besides the sieve unnecessarily setting position 2 prior to the loop, the sieve does not pay any respect to spatial locality: making it not very cache friendly. By dividing our range into segments less than rad(n) and operating on those segments one by one, we can use the primes generated in the lower segment to eliminate composite numbers in the next segment. This would make the algorithm significantly more cache friendly.

Bit Vector

Create(len)

To ensure I have enough bits to represent len primes, I will allocate t bytes (using uint8_t) on the heap.

- Allocate t bytes on the heap and store in ptr bv
- if bv (allocated bit vector) is valid, return bv
    - else, return null (memory allocation failed)

However, the formula in the lab manual overallocates when len is a multiple of 8 (8 bits in a byte), so I made the following change:

- if len % 8 = 0, `t = n/8`
    - else, `t = (n/8) + 1`

Delete(v)

- Free v's vector and null the pointer
- Free v and null the pointer

Get_Len(v)

- Return length

Set_Bit(v, i)

For this function, I will get the byte containing i using division then OR it with a byte containing 0x01 shifted left.

- if i >= length, return
- Set x = 1
- Set y = vector[i / 8]
- Shift x left 7 - (i % 8)
- y OR x

Clr_Bit(v, i)

This works largely the same as set, except I will NOT the shifted 1 bit and then AND it with the byte I'm operating on.

- if i >= length, return
- Set x = 1
- Set y = vector[i / 8]
- Shift x left `7 - (i % 8)` times
- x = NOT x
- y AND x

Get_Bit(v, i)

I will get the bit by setting a byte to 0x01 and shifting it to the proper position. Then, I will AND it, shift it to the end of the byte, and return it.

- Set x = 1
- Set y = vector[i / 8]
- Shift x left 7 - (i % 8)
- return (x AND y)

Set_All_Bits(v)

- for i = 0 -> i/8
    - Set vector[i] = 0x11

Sieve

Credit to the lab manual for providing the code for the Sieve and Eratosthenes of Cyrene for the original algorithm.

Design: The algorithm is quite basic. First, it clears 0 and 1. Then, starting from 2, it begins looping from [2, n]. For each element i, it checks if i is prime. If it is, it begins marking every multiple of it as composite until it reaches the length. At that point, every prime number will be represented by a 1 bit and composite a 0 bit.

Pseudocode:

- Set all bits to 1
- Clear the bits in positions 0 and 1
- for i = [2, length)
    - if get_bit(v, i) is prime
        - for k = [0, length]
            - Mark all multiples of i as composite: clr_bit(v, (k + i) * i)
- End

Sequence

Design: Similarly to asgn 3, it primarily consists of a simple getopt loop that'll decide whether to look for interesting primes or palindromes. Both options will execute the sieve to gather primes. Each respective option will follow the design/pseudocode provided in Prelab Pt1.

Like asgn 3, I decided to opt for bools to track the command line arguments rather than enums to allow for quicker debugging. Using bools better allows multiple, simultaneous inputs (and takes less memory) than tracking several different enums, so I opted for them instead.

Pseudocode:

- Declare bools (run_interest, run_pal) and size (n) to track each command line argument
- Set n to be one more than the default (n = 1000) so that the upper bound is included in the zero-indexed BitVector
- Check if there are less than 1 command-line arguments supplied, no run preference, or too many arguments. Print an error and return if so
- while there are options to parse
    - Set bool statuses for each argument.
    - Set `size = n + 1` (if n is supplied) for inclusivity.
- v = bv_create(n)
- sieve(v)
- if run_interest
    - Follow prelab 1.1 pseudocode.
- if run_pal
    - Follow prelab 1.2 pseudocode.

Design Changes

My design for the interesting primes section went along without a hitch. The only changes I ended up making were reorganizing the order in which I checked M/F/L after reading the lab manual wrong, and then clarifying the design document to better articulate why I declared my variables to what they were.

However, I had significant difficulty with printing palindromes. After struggling with properly tracking strings between functions, I opted to combine my original two helpers (convert and isPalindrome) into a single one (palindrome) that handles string/base conversion, checking, and printing. This allowed me to simply allocate it on the heap and free it once instead of having to allocate it on the stack in the main and pass pointers between two helpers. This also kept me from having to recalculate length or include <string.h>.

I also spent significant time testing a queue based solution that would allow me to track all the palindromic primes using only one pass through the BV and then store my results for later printing. However, I quickly found this solution to be highly space inefficient and decided the slight time efficiency introduced by four different searches that also tackling printing simultaneously to be worth it. I found 4 searches where I print at the same time hardly took longer than one search with four large queues and four more print loops.