Project 3: Prime Calculator

Discovering Prime Numbers as Application of BitVector

Revision dated 01/02/18

Educational Objectives. After successfully completing this assignment, the student should be able to accomplish the following:

- Design a class based on non-language-specific specifications
- Implement a class of your own design
- Implement constructors, copy constructor, destructor, and assignment operator for a class that has resource allocation requirements
- Global operators for a class
- Correctly separate class definition and implementation using files
- Create executables of class client programs using makefiles and the Make utility
- Test a class using specs and an existing test platform
- Use `fsu::BitVector` in designing set classes

Operational Objectives: Define and implement the class `Prime` and deliver the code in two files `prime.h` and `prime.cpp` along with a makefile for the supplied test harness. Also implement the Expand method for `fsu::BitVector` in the file `bitvect.cpp`.

Deliverables: `bitvect.cpp`, `prime.h`, `prime.cpp`, `log.txt`

Assessment Rubric

```
=====================================================
student build:
    fbitvect.x:                          [0..3]:   x
    prime_below.x                        [0..3]:   x
    fprime.x                             [0..3]:   x
assess  build:
    fbitvect.x                           [0..3]:   x
    fprime.x                             [0..3]:   x
test:
    fbitvect.x Expand                    [0..5]:   x
    fprime.x Largest                     [0..5]:   x
    fprime.x All                         [0..5]:   x
    fprime.x ResetUpperBound             [0..5]:   x
    fprime.x copychecks                  [0..5]:   x
code:
    constructor                          [0..4]:   x
    copy constructor                     [0..2]:   x
    destructor                           [0..2]:   x
    assignment operator                  [0..2]:   x
engineering etc:
   requirements                        [-20..0]: ( x)
   coding standard                     [-20..0]: ( x)
dated submission deduction          [2 pts per]: ( x)
                                                  --
total                                   [0..50]:  xx
=====================================================
```

Background

See lecture notes Chapter 4. Classes Part 1, Chapter 5. Pointers, Chapter 6. Classes Part 2, and Chapter 8. BitVectors.

The Sieve of Eratosthenes

Assume that b is a vector of bits indexed in the range [0 ... n). Denote the "value" of bit k by $b[k]$. The *Sieve of Eratosthenes* is a process that operates on a bit vector b, as follows:

1. Begin with a bitvector b indexed in the range $0 \le k < n$. Our goal is to unset bits for all composit numbers up to n, so that $b[k] = 1$ if and only if k is prime.
2. Initialize b by setting all bits.
3. Unset $b[0]$ and $b[1]$ (because 0 and 1 are not prime).
4. For k between 2 and the square root of n, stepsize 1:
   if $b[k]$ is set
     for j between $k + k$ and n, stepsize k:
       unset $b[j]$
5. Stop.

In short, unset the bits of all multiples of primes less than the square root of n.

Assertion 1. After invoking the sieve algorithm, an integer k in the range $[0 \ldots n)$ is prime iff $b[k] = 1$.

The assertion is proved by mathematical induction. The base cases $k = 0,1,2$ are each easily checked by following the first few lines of the process. For the inductive step, assume the assertion is true for all index values less than k. If $b[k] = 0$ then there was an instance of $k = a \times b$ which resulted in unsetting $b[k]$, so clearly k is composit. If $b[k] = 1$ then there was never an instance of $k = a \times b$ with a prime and $a^2 \le k$. But that is enough to prove that k is prime, because a composit number always has a factorization of the form $a \times b$ with $a \le b$ (by just writing the smaller factor first) and then we would have $k = p \times q$ where p is a prime factor of a and $q = b \times a/p$.

Remark. *What Eratosthenes was thinking?* Clearly, the big E did not use bitvectors. His approach went something like this: Imagine the numbers 1..*n* all written down in a list. We will cross all the composit numbers off of the list, so that those that are left must be all of the non-composit, that is, prime, numbers. The E-man went on to describe how to cross numbers off: first cross off 1, keep 2, and then cross off all multiples of 2. Go to the next number not crossed off (which must be prime) and cross of all of its multiples. Keep going until the list is exhausted.

Procedural Requirements:

1. Copy all of the files in `LIB/proj3` into your `cop3330/proj3` directory. Then copy the file `LIB/cpp/bitvect.cpp` into your `cop3330/proj3` directory. You should now see these files (and perhaps others) in your project directory:

   ```
   bitvect.cpp
   deliverables.sh
   fbitvect.cpp
   fprime.cpp
   prime_below.cpp
   ```

2. Begin your log file named `log.txt`. (See <u>Assignments</u> for details.)
3. Familiarize yourself with the BitVector code in your library: `LIB/cpp/bitvect.h` and `LIB/cpp/bitvect.cpp`. Both the API and implementation are discussed in the class notes.
4. Note that there is a non-functional implementation of BitVector::Expand() in the library. (This is the file you copied into your project directory.) One of your objectives is to supply functional code for this method:

   Implement the method `void fsu::BitVector::Expand(size_t numbits)` in your copy of the file `bitvect.cpp`.

5. Debug your newly augmented BitVector class with the command "co3330 bitvect".
6. Design the class Prime. Note that this is a client of `fsu::BitVector` and therefore must use the BitVector API. You are not implementing BitVector (except for the Expand method) and your Prime code cannot access the protected areas in BitVector.
7. Implement the class Prime with the class definition in file `prime.h` and the class implementation in file `prime.cpp`
8. Debug your Prime code with the commands "co3330 prime" and "co3330 bitvect".
9. Create a makefile that builds the executables for fbitvect.x, fprime.x, and prime_below.x.
10. Thoroughly test BitVector::Expand using fbitvect.x.
11. Thoroughly test Prime using fprime.x.
12. Turn in `bitvect.cpp`, `prime.h`, `prime.cpp`, `makefile`, and `log.txt` using `LIB/scripts/submit.sh` and `LIB/proj3/deliverables.sh`, following the usual procedure.

    ***Warning:** Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

Technical Requirements and Specifications - BitVector::Expand

1. Implement the method `BitVector::Expand(size_t nb)` so that it has the effect enlarging the number of bits to at least `nb` while keeping the state of each of the existing bits in the enlarged object. The necessary steps in this implementation are:
   a. Calculate `newByteArraySize` = the number of bytes required for `nb` bits
   b. Using a locally declared pointer, create a new byte array of size `newByteArraySize`
   c. Initialize the new byte array to be the same as the old one where they share indices and to be zero for the "new" bits
   d. Set `byteArraySize_` to `newByteArraySize`
   e. Delete the old `byteArray_`
   f. Point `byteArray_` to the newly allocated byte array
2. Expand(nb) should do nothing if nb does not excede the number of bits already allocated.

3. In all cases a call to Expand() should not change the bit values for any existing bits and should initialze all new bits to 0
4. When in doubt about required behavior, consult the executable `LIB/area51/fbitvect_i.x`.

Technical Requirements and Specifications - class Prime

1. The class should implement the following diagram:

Class Name:	`Prime`

Public Services :	`size_t Largest ( size_t ub ) const;`
	`void All ( size_t ub , std::ostream& os = std::cout ) const;`
	`void All ( std::ostream& os = std::cout ) const;`
	`size_t UpperBound () const;`
	`void ResetUpperBound ( size_t ub );`

Developer Services :	`void Dump ( std::ostream& os = std::cout ) const;` `// used in development & testing; displays underlying bitvector state`

Properties :	`Constructable: objects can be declared as ordinary variables, ub must be specified`
	`Assignable: objects can be assigned one to another`
	`Passable: objects can be passed by value to and returned as values from functions`

Private variables:	`fsu::BitVector bv_; // bit vector representing primes`
Private services:	`void Sieve(); // initializes bitvector to code primes`

2. The class should be a proper type, to include one 1-argument constructor and, in cases where the defaults are inadequate, the copy constructor, assignment operator, and destructor.
3. **Prime(n):** the constructor should initialize the private bitvector object in the init list and invoke Sieve() in the body, ensuring that all primes ≤ **n** are coded. (Note this requires at least **n**+1 bits.)
4. **Largest(n):** returns the largest prime that is bounded above by n. (If **n** excedes the number of bits, it is replaced by the number of bits.)
5. **All(n os):** sends all primes less than or equal to **n** to the stream os (again replacing **n** with the max number of bits if it excedes that number).
6. **All(os):** sends all primes less than or equal to **p.UpperBound()** to the stream os.
7. **UpperBound():** returns the largest bit index value stored.
8. **ResetUpperBound(n):** sets the upper bound to **n** if necessary. Calls BitVector::Expand.
9. **Sieve():** performs the Sieve of Eratosthenes on bv_.
10. **Dump(os):** is intended for use by the development and testing teams. It should display the current state of the underlying BitVector object. For example, for the object `Prime p(25)` the display from `p.Dump()` would be

    ```
    00110101000101000101000100000101
    01234567890123456789012345678901
    ```

    Study this carefully and you will understand a lot about how the implementation works. Why are there 32 bits displayed, from 0 through 31, when we only asked for 0 through 25? What is the significance of the set bits? Just looking at this dump output, can you say what is returned by `p.Largest(22)`? What is output by the call `p.All(22)`? How about `p.All(23)`?
11. Building and running the supplied `proj3/fprime.cpp` should result in output identical to the supplied executable `area51/fprime_i.x`.
12. When in doubt about required behavior, consult the executable `LIB/area51/fprime_i.x`.

Technical Requirements and Specifications - makefile

1. Create a makefile that builds the functional tests fbitvect.x, fprime.x and prime_below.x.
2. You will need intermediate targets for these object code files: `bitvect.o`, `fbitvect.o`, `fprime.o`, `prime.o`, `prime_below.o` from which you assemble the three executable targets. Here are some sample lines from a makefile to get you started:

    ```
    LIB      = /home/courses/cop3330p/LIB
    CC       = g++ -std=c++11 -I. -I$(LIB)/cpp -Wall -Wextra

    all:      fbitvect.x fprime.x prime_below.x
    ```

```
fprime.x: prime.o bitvect.o fprime.o
        $(CC) -o fprime.x prime.o bitvect.o fprime.o

bitvect.o: bitvect.cpp # bitvect.h
        $(CC) -c bitvect.cpp
```

Because you have debugged your source code files individually, it should be fairly straighforward to create the makefile. Note that we illustrate how to use the dependencies to optimize the build strategy without undue clutter: put only source files that you are creating in the dependency lists. That is why `bitvect.h` is commented out: (a) it exists in `LIB/cpp` and (b) we are not coding that file, so keeping it in the dependency list just clutters our workspace.

Hints

- Note that both fbitvect and fprime accept a command file, so you can devise tests, record them in a command file, and then repeat the test with a few keystrokes. Enter the executable name with no arguments to see what is expected.
- Note that Prime objects should always have their bitvector correctly "sieved". So the constructor needs to call Sieve(). Similarly, ResetUpperBound will need to call BitVector::Expand as well some possibly abbreviated version of Sieve().
- This is an optimizer's paradise. The basic Sieve algorithm can be optimized in several ways, such as: reducing the more-or-less wasted space and time spent dealing with even numbers and starting the inner loop at k*k instead of k+k (why?). And a "Re-Sieve" algorithm can be devised that works on only the newly allocated bits after an expansion: Re-Sieve(k) would start the sieving process at k+1 under the assumption that it has run to completion for size k. These optimizations are not required for this project, but keep them in mind for recreational code tinkering.
- Another enhancement would be to insert a timing device into the Sieve process. The files required to do this are in your LIB/cpp. An enhanced executable `LIB/area51/fprime+_i.x` is available. To activate the timer, add a second command line argument "1". (The com file argument becomes the third.) Again, just enter the command with no arguments to get the hint.
- Sieve is often used as a benchmark program. Here is some timing data we obtained for the new linprog machines, using an optimized version of Sieve:

```
   n      PrimeBelow(n)       time on old linprog2       time on new linprog7
----      -------------       --------------------       --------------------
10^2                 97          0.00 sec                    0.00 sec
10^3                997          0.00                        0.00
10^4               9973          0.00                        0.00
10^5              99991          0.00                        0.00
10^6             999983          0.02                        0.02
10^7            9999991          0.21                        0.13
10^8           99999989          2.49                        0.98
10^9          999999937         33.66                       13.31
10^10        9999999967        394.17 (6.57 min)           169.80 (2.83 min)
10^11       99999999977       4398.74 (73.31 min)         1924.85 (32.08 min)
```

Note that this data appears to show that the runtime of Sieve() is slightly slower than $\Theta(n)$ but considerably faster than $\Theta(n^2)$. See the Wikipedia entry for much more on optimization, runtime, and other Sieve topics.