Assignment 1
A Little Light Number Theory

Prof. Darrell Long
Department of Computer Engineering

Spring 2017

1 Introduction

We are going to answer some simple but fundamental questions about the sequence of natural numbers $\mathbb{N} = 1, 2, 3, \ldots$ We are going to look at primality, composites and perfect numbers. Primes numbers are fundamental in mathematics and Computer Science, and perfect numbers are an interesting curiosity.

You will also be concerned with the execution time of your programs.

1.1 Prime Numbers

God may not play dice with the universe, but something strange is going on with the prime numbers.

—Paul Erdös

Prime numbers are among the most interesting of the natural numbers. A number $p \in \mathbb{N}$ is *prime* if it is evenly divisible by only 1 and p. That means that $(\forall m \in \mathbb{N}, 1 < m < p) m \nmid p$ or alternatively, $(\forall m \in \mathbb{N}, 1 < m < p) p \bmod m \neq 0$. The first prime number is 2, all primes except 2 must be odd, since all even numbers are divisible by 2. There are an infinity of primes, which was proven by Euclid about 300 B.C. There is no formula for finding the primes, but the *prime number theorem* tells us that the probability of a given $m \in \mathbb{N}, 1 < m \leq N$ being prime is very close to $\frac{1}{\ln N}$, since the number of primes less than N, denoted $\pi(N)$, is

$$\frac{N}{\ln N - (1-\epsilon)} < \pi(N) < \frac{N}{\ln N - (1+\epsilon)}.$$

Determining whether a number n is prime is conceptually simple, all that must be done is to try every $k \in \{2, \ldots, n-1\}$. The execution time for this method is $O(n)$, which for large n is prohibitive. A little thought shows that we do not need to check so many, and that evaluating $k \mid n$ for $k \in \{2, \ldots, \lceil\sqrt{n}\rceil\}$ is sufficient. The resulting execution time of $O(\sqrt{n})$ is a great improvement. Is that the best that we can hope for? No, but a lower bound is *unknown*.

1.2 Composite Numbers

The problem of distinguishing prime numbers from composite numbers and of resolving the latter into their prime factors is known to be one of the most important and useful in arithmetic.

—Carl Friedrich Gauss

All natural numbers that are not prime are called *composite*. The *fundamental theorem of arithmetic*, also called the *unique factorization theorem*, states that every integer $m > 1$ is either prime or a unique product of primes $(p_0, \ldots, p_k)$. That is,

$$m = p_0^{\alpha_0} \times p_1^{\alpha_1} \times \ldots \times p_k^{\alpha_k} = \prod_{i=0}^{k} p_i^{\alpha_k}.$$

For example, $83736 = 2^3 \times 3^2 \times 1163$. In 1801, the fundamental theorem of arithmetic was proved by Gauss in his book *Disquisitiones Arithmeticæ*.

Determining the prime factorization of m can be accomplished trying all primes $p_0, \ldots, p_k \leq m$. The execution time is difficult to compute, since each successful division reduces the complexity, but is it bounded from above by $O(\log m)$, provided you have a list of primes to consult.

1.3 Perfect Numbers

Perfect numbers like perfect men are very rare.

—Rene Descartes

A *perfect number* is a natural number that is equal to the sum of its proper divisors. That is,

$$m = \sum_{i|m} i.$$

For example, $1+2+3 = 6$ and $496 = 1+2+4+8+16+31+62+124+248$. The first four perfect numbers are $6, 28, 496$ and $8\,128$, and you will not find the next one until $33\,550\,336$.

It is not known whether there are any *odd* perfect numbers, but it is considered unlikely. All even perfect numbers have the form $2^{p-1}(2^p - 1)$ where p is prime and its *digital root* is 1.

2 Your Task

Your task is to go through each natural number beginning with 2 and determine whether it is *prime* (`P`), *composite* (`C`) or *perfect* (or `E` in honor of Euclid). Prime numbers cannot be factored, but you must perform a *prime factorization* of all composites. If you determine that a number is perfect, then you must list all of its *proper divisors*.

Consider the example given below. This shows the *required* format of the output.

```
2 P
3 P
4 C: 2 2
5 P
6 C: 2 3
6 E: 1 2 3
7 P
8 C: 2 2 2
9 C: 3 3
10 C: 2 5
11 P
12 C: 2 2 3
13 P
14 C: 2 7
15 C: 3 5
16 C: 2 2 2 2
17 P
18 C: 2 3 3
19 P
20 C: 2 2 5
```

Example

What you should observe immediately is that you will need to store some quantities. You may want to store the prime factors of each composite. Is that necessary? You will need to store the store the proper divisors of a composite since you will not know that a number is perfect until you have tallied its proper divisors. The most obvious data structure for storing those divisors is an *array*.

3 Specifics

Writing the program using the naïve approach of finding primes up to the value of a given number into order to find its prime factorization is, frankly, silly. You can and will do much better, and you will do so by using a *sieve*.

```
# ifndef _SIEVE
# define _SIEVE
# include "bv.h"

void sieve(bitV *); // Results in a vector of prime numbers
# endif
```

sieve.h

Your sieve will take as its sole argument a *bit vector* and it will mark with `1` the positions that correspond to a prime number, while all other positions will be marked `0`.

A bit vector is a rarely taught but essential tool in the kit of all Computer Scientists and Engineers. Your bit vector implementation *must* match the given interface.

```
# ifndef _BVector
# define _BVector
# include <stdint.h>

typedef struct bitV {
    uint8_t *vector;
    uint32_t length;
} bitV;

bitV *newVec(uint32_t); // Create a new vector of specified length

void delVec(bitV *); // Deletes a vector

void oneVec(bitV *); // Creates a vector of all 1

void setBit(bitV *, uint32_t); // Sets a specified bit

void clrBit(bitV *, uint32_t); // Clears a specified bit

uint8_t valBit(bitV *, uint32_t); // Returns the value of a specified
    bit

uint32_t lenVec(bitV *); // Return the length of the vector
# endif
```

bv.h

The header file `bv.h` defines the exported type `bitV` and its associated operations. Even though **C** will not prevent you from directly manipulating the data structure, you must avoid the temptation and *only* use the functions defined in `bv.h`—no exceptions!

You must implement each of the functions specified in the header file. Most of them are just a line of two of **C** code, but their implementation can be subtle. You are warned *again* against using code that you may find on the Internet.

One *could* write a function

```
uint64_t nextPrime(uint64_t n)
{
    \\ Find the first prime after n
}
```

but that would be foolishness, even though the prime density is approximately $\frac{1}{\ln n}$. Instead, you will use a sieve. The easiest to implement is the *Sieve of Eratosthenes* but ambitious students are encouraged to use a more sophisticated method. The number of operations to find up to N primes is

$$\log\log N - \frac{1}{\log N}\left(1 - \frac{4}{\sqrt{N}}\right) + M - \log 2,$$

where $M \approx 0.261\,497\,212\,847\,642\,783\,755\ldots$ (the Meissel-Mertens constant). Is this the best that can be done? No, the *general number field sieve* is the most efficient classical algorithm *known* for factoring integers larger than 10^{100}. Is it the best possible? We simply do not know.

```
void sieve(bitV *v)
{
    oneVec(v);    // Assume all are prime
    clrBit(v, 0); // 0 is, well, zero
    clrBit(v, 1); // 1 is unity
    setBit(v, 2); // 2 is prime
    for (uint32_t i = 2; i <= sqrtl(lenVec(v)); i += 1)
    {
        if (valBit(v, i)) // It's prime
        {
            for (uint32_t k = 0; (k + i) * i <= lenVec(v); k += 1)
            {
                clrBit(v, (k + i) * i); // Its multiple are composite
            }
        }
    }
    return;
}
```

Since you have been given the code for the Sieve of Eratosthenes, you *must* cite it and give proper credit if you use it. If, for example, you were to implement the Sieve of Sundaram, or the more modern Sieve of Atkin, you would not need to cite beyond the source of the algorithm and any pseudocode that you followed.

Submission

Your program must be *capable* of executing correctly until it would find the *sixth* perfect number. But, that would take a very long time unless you were very clever with your algorithms. So, your program will *by default* run until it reaches 100 000. Along the way it should find *four* perfect numbers and a large number of prime numbers as well.

We will test your program by comparing its output with the output of a known correct program. Example output is given at the end of the assignment, and your program should match it exactly (for as far as the example goes, the test will go to 100 000).

You *must* turn in your assignment in the following manner:

1. By *default* your program runs up through 100 000.
2. *Optionally*, you can provide for `./parfait -n K` were `K` is the largest natural number considered.
3. Have file called `Makefile` that when the grader types `make` will compile your program. At this point you will have learned about `make` and can create your own `Makefile`.
   - `CFLAGS=-Wall -Wextra -Werror -pedantic` must be included.
   - `CC=gcc` must be specified.
   - `make clean` must remove all files that are compiler generated.
   - `make` should build your program, as should `make all`.
   - Your program executable must be named `parfait`.
4. You program *must* have the source and header files:
   - `bv.h` to specify the bit vector operations and abstract data type `bitV`.
   - `bv.c` to implement the functionality.
   - `sieve.h` specifies the interface to the sieve.
   - `sieve.c` to implement the sieve algorithm of your choice.
   - `parfait.c` contains `main()` and *may* contain the other functions necessary to complete the assignment.
5. You may have other source and header files, but *do not try to be overly clever*.
6. A plain text file called `README` that describes how your program works.
7. The executable file produced by the compiler *must be called* `parfait`.
8. These files must be in the directory `assignment1`.
9. You must `commit` and `push` the directory and its contents using `git`.

Example

```
2 P
3 P
4 C: 2 2
5 P
6 C: 2 3
6 E: 1 2 3
7 P
8 C: 2 2 2
9 C: 3 3
10 C: 2 5
11 P
12 C: 2 2 3
```

```
13 P
14 C: 2 7
15 C: 3 5
16 C: 2 2 2 2
17 P
18 C: 2 3 3
19 P
20 C: 2 2 5
21 C: 3 7
22 C: 2 11
23 P
24 C: 2 2 2 3
25 C: 5 5
26 C: 2 13
27 C: 3 3 3
28 C: 2 2 7
28 E: 1 2 4 7 14
29 P
30 C: 2 3 5
31 P
32 C: 2 2 2 2 2
33 C: 3 11
34 C: 2 17
35 C: 5 7
36 C: 2 2 3 3
37 P
38 C: 2 19
39 C: 3 13
40 C: 2 2 2 5
41 P
42 C: 2 3 7
43 P
44 C: 2 2 11
45 C: 3 3 5
46 C: 2 23
47 P
48 C: 2 2 2 2 3
49 C: 7 7
50 C: 2 5 5
51 C: 3 17
52 C: 2 2 13
53 P
54 C: 2 3 3 3
55 C: 5 11
56 C: 2 2 2 7
57 C: 3 19
58 C: 2 29
59 P
60 C: 2 2 3 5
61 P
```

```
62 C: 2 31
63 C: 3 3 7
64 C: 2 2 2 2 2 2
65 C: 5 13
66 C: 2 3 11
67 P
68 C: 2 2 17
69 C: 3 23
70 C: 2 5 7
71 P
72 C: 2 2 2 3 3
73 P
74 C: 2 37
75 C: 3 5 5
76 C: 2 2 19
77 C: 7 11
78 C: 2 3 13
79 P
80 C: 2 2 2 2 5
81 C: 3 3 3 3
82 C: 2 41
83 P
84 C: 2 2 3 7
85 C: 5 17
86 C: 2 43
87 C: 3 29
88 C: 2 2 2 11
89 P
90 C: 2 3 3 5
91 C: 7 13
92 C: 2 2 23
93 C: 3 31
94 C: 2 47
95 C: 5 19
96 C: 2 2 2 2 2 3
97 P
98 C: 2 7 7
99 C: 3 3 11
100 C: 2 2 5 5
101 P
102 C: 2 3 17
103 P
104 C: 2 2 2 13
105 C: 3 5 7
106 C: 2 53
107 P
108 C: 2 2 3 3 3
109 P
110 C: 2 5 11
111 C: 3 37
```

```
112 C: 2 2 2 2 7
113 P
114 C: 2 3 19
115 C: 5 23
116 C: 2 2 29
117 C: 3 3 13
118 C: 2 59
119 C: 7 17
120 C: 2 2 2 3 5
121 C: 11 11
122 C: 2 61
123 C: 3 41
124 C: 2 2 31
125 C: 5 5 5
126 C: 2 3 3 7
127 P
128 C: 2 2 2 2 2 2 2
129 C: 3 43
130 C: 2 5 13
131 P
132 C: 2 2 3 11
133 C: 7 19
134 C: 2 67
135 C: 3 3 3 5
136 C: 2 2 2 17
137 P
138 C: 2 3 23
139 P
140 C: 2 2 5 7
141 C: 3 47
142 C: 2 71
143 C: 11 13
144 C: 2 2 2 2 3 3
145 C: 5 29
146 C: 2 73
147 C: 3 7 7
148 C: 2 2 37
149 P
150 C: 2 3 5 5
151 P
152 C: 2 2 2 19
153 C: 3 3 17
154 C: 2 7 11
155 C: 5 31
156 C: 2 2 3 13
157 P
158 C: 2 79
159 C: 3 53
160 C: 2 2 2 2 2 5
161 C: 7 23
```

```
162 C: 2 3 3 3 3
163 P
164 C: 2 2 41
165 C: 3 5 11
166 C: 2 83
167 P
168 C: 2 2 2 3 7
169 C: 13 13
170 C: 2 5 17
171 C: 3 3 19
172 C: 2 2 43
173 P
174 C: 2 3 29
175 C: 5 5 7
176 C: 2 2 2 2 11
177 C: 3 59
178 C: 2 89
179 P
180 C: 2 2 3 3 5
181 P
182 C: 2 7 13
183 C: 3 61
184 C: 2 2 2 23
185 C: 5 37
186 C: 2 3 31
187 C: 11 17
188 C: 2 2 47
189 C: 3 3 3 7
190 C: 2 5 19
191 P
192 C: 2 2 2 2 2 2 3
193 P
194 C: 2 97
195 C: 3 5 13
196 C: 2 2 7 7
197 P
198 C: 2 3 3 11
199 P
200 C: 2 2 2 5 5
201 C: 3 67
202 C: 2 101
203 C: 7 29
204 C: 2 2 3 17
205 C: 5 41
206 C: 2 103
207 C: 3 3 23
208 C: 2 2 2 2 13
209 C: 11 19
210 C: 2 3 5 7
211 P
```

```
212 C: 2 2 53
213 C: 3 71
214 C: 2 107
215 C: 5 43
216 C: 2 2 2 3 3 3
217 C: 7 31
218 C: 2 109
219 C: 3 73
220 C: 2 2 5 11
221 C: 13 17
222 C: 2 3 37
223 P
224 C: 2 2 2 2 2 7
225 C: 3 3 5 5
226 C: 2 113
227 P
228 C: 2 2 3 19
229 P
230 C: 2 5 23
231 C: 3 7 11
232 C: 2 2 2 29
233 P
234 C: 2 3 3 13
235 C: 5 47
236 C: 2 2 59
237 C: 3 79
238 C: 2 7 17
239 P
240 C: 2 2 2 2 3 5
241 P
242 C: 2 11 11
243 C: 3 3 3 3 3
244 C: 2 2 61
245 C: 5 7 7
246 C: 2 3 41
247 C: 13 19
248 C: 2 2 2 31
249 C: 3 83
250 C: 2 5 5 5
251 P
252 C: 2 2 3 3 7
253 C: 11 23
254 C: 2 127
255 C: 3 5 17
256 C: 2 2 2 2 2 2 2 2
257 P
258 C: 2 3 43
259 C: 7 37
260 C: 2 2 5 13
261 C: 3 3 29
```

```
262 C: 2 131
263 P
264 C: 2 2 2 3 11
265 C: 5 53
266 C: 2 7 19
267 C: 3 89
268 C: 2 2 67
269 P
270 C: 2 3 3 3 5
271 P
272 C: 2 2 2 2 17
273 C: 3 7 13
274 C: 2 137
275 C: 5 5 11
276 C: 2 2 3 23
277 P
278 C: 2 139
279 C: 3 3 31
280 C: 2 2 2 5 7
281 P
282 C: 2 3 47
283 P
284 C: 2 2 71
285 C: 3 5 19
286 C: 2 11 13
287 C: 7 41
288 C: 2 2 2 2 2 3 3
289 C: 17 17
290 C: 2 5 29
291 C: 3 97
292 C: 2 2 73
293 P
294 C: 2 3 7 7
295 C: 5 59
296 C: 2 2 2 37
297 C: 3 3 3 11
298 C: 2 149
299 C: 13 23
300 C: 2 2 3 5 5
301 C: 7 43
302 C: 2 151
303 C: 3 101
304 C: 2 2 2 2 19
305 C: 5 61
306 C: 2 3 3 17
307 P
308 C: 2 2 7 11
309 C: 3 103
310 C: 2 5 31
311 P
```

```
312 C: 2 2 2 3 13
313 P
314 C: 2 157
315 C: 3 3 5 7
316 C: 2 2 79
317 P
318 C: 2 3 53
319 C: 11 29
320 C: 2 2 2 2 2 2 5
321 C: 3 107
322 C: 2 7 23
323 C: 17 19
324 C: 2 2 3 3 3 3
325 C: 5 5 13
326 C: 2 163
327 C: 3 109
328 C: 2 2 2 41
329 C: 7 47
330 C: 2 3 5 11
331 P
332 C: 2 2 83
333 C: 3 3 37
334 C: 2 167
335 C: 5 67
336 C: 2 2 2 2 3 7
337 P
338 C: 2 13 13
339 C: 3 113
340 C: 2 2 5 17
341 C: 11 31
342 C: 2 3 3 19
343 C: 7 7 7
344 C: 2 2 2 43
345 C: 3 5 23
346 C: 2 173
347 P
348 C: 2 2 3 29
349 P
350 C: 2 5 5 7
351 C: 3 3 3 13
352 C: 2 2 2 2 2 11
353 P
354 C: 2 3 59
355 C: 5 71
356 C: 2 2 89
357 C: 3 7 17
358 C: 2 179
359 P
360 C: 2 2 2 3 3 5
361 C: 19 19
```

```
362 C: 2 181
363 C: 3 11 11
364 C: 2 2 7 13
365 C: 5 73
366 C: 2 3 61
367 P
368 C: 2 2 2 2 23
369 C: 3 3 41
370 C: 2 5 37
371 C: 7 53
372 C: 2 2 3 31
373 P
374 C: 2 11 17
375 C: 3 5 5 5
376 C: 2 2 2 47
377 C: 13 29
378 C: 2 3 3 3 7
379 P
380 C: 2 2 5 19
381 C: 3 127
382 C: 2 191
383 P
384 C: 2 2 2 2 2 2 2 3
385 C: 5 7 11
386 C: 2 193
387 C: 3 3 43
388 C: 2 2 97
389 P
390 C: 2 3 5 13
391 C: 17 23
392 C: 2 2 2 7 7
393 C: 3 131
394 C: 2 197
395 C: 5 79
396 C: 2 2 3 3 11
397 P
398 C: 2 199
399 C: 3 7 19
400 C: 2 2 2 2 5 5
401 P
402 C: 2 3 67
403 C: 13 31
404 C: 2 2 101
405 C: 3 3 3 3 5
406 C: 2 7 29
407 C: 11 37
408 C: 2 2 2 3 17
409 P
410 C: 2 5 41
411 C: 3 137
```

```
412 C: 2 2 103
413 C: 7 59
414 C: 2 3 3 23
415 C: 5 83
416 C: 2 2 2 2 2 13
417 C: 3 139
418 C: 2 11 19
419 P
420 C: 2 2 3 5 7
421 P
422 C: 2 211
423 C: 3 3 47
424 C: 2 2 2 53
425 C: 5 5 17
426 C: 2 3 71
427 C: 7 61
428 C: 2 2 107
429 C: 3 11 13
430 C: 2 5 43
431 P
432 C: 2 2 2 2 3 3 3
433 P
434 C: 2 7 31
435 C: 3 5 29
436 C: 2 2 109
437 C: 19 23
438 C: 2 3 73
439 P
440 C: 2 2 2 5 11
441 C: 3 3 7 7
442 C: 2 13 17
443 P
444 C: 2 2 3 37
445 C: 5 89
446 C: 2 223
447 C: 3 149
448 C: 2 2 2 2 2 2 7
449 P
450 C: 2 3 3 5 5
451 C: 11 41
452 C: 2 2 113
453 C: 3 151
454 C: 2 227
455 C: 5 7 13
456 C: 2 2 2 3 19
457 P
458 C: 2 229
459 C: 3 3 3 17
460 C: 2 2 5 23
461 P
```

```
462 C: 2 3 7 11
463 P
464 C: 2 2 2 2 29
465 C: 3 5 31
466 C: 2 233
467 P
468 C: 2 2 3 3 13
469 C: 7 67
470 C: 2 5 47
471 C: 3 157
472 C: 2 2 2 59
473 C: 11 43
474 C: 2 3 79
475 C: 5 5 19
476 C: 2 2 7 17
477 C: 3 3 53
478 C: 2 239
479 P
480 C: 2 2 2 2 2 3 5
481 C: 13 37
482 C: 2 241
483 C: 3 7 23
484 C: 2 2 11 11
485 C: 5 97
486 C: 2 3 3 3 3 3
487 P
488 C: 2 2 2 61
489 C: 3 163
490 C: 2 5 7 7
491 P
492 C: 2 2 3 41
493 C: 17 29
494 C: 2 13 19
495 C: 3 3 5 11
496 C: 2 2 2 2 31
496 E: 1 2 4 8 16 31 62 124 248
497 C: 7 71
498 C: 2 3 83
499 P
500 C: 2 2 5 5 5
```

Longer Example