(See https://cs.stanford.edu/~knuth/programs.html for date.)

**1. Intro.** This program constructs segments of the "sieve of Eratosthenes," and outputs the largest prime gaps that it finds. More precisely, it works with sets of prime numbers between $s_i$ and $s_{i+1} = s_i + \delta$, represented as an array of bits, and it examines these arrays for $t$ consecutive intervals beginning with $s_i$ for $i = 0, 1, \ldots\ t-1$. Thus it scans all primes between $s_0$ and $s_t$.

Let $p_k$ be the $k$th prime number. The sieve of Eratosthenes determines all primes $\le N$ by starting with the set $\{2, 3, \ldots, N\}$ and striking out the nonprimes: After we know $p_1$ through $p_{k-1}$, the next remaining element is $p_k$, and we strike out the numbers $p_k^2$, $p_k(p_k+1)$, $p_k(p_k+2)$, etc. The sieve is complete when we've found the first prime with $p_k^2 > N$.

In this program it's convenient to deal with the nonprimes instead of the primes, and to assume that we already know all of the "small" primes $p_k$ for which $p_k^2 \le s_t$. And of course we might as well restrict consideration to odd numbers. Thus, we'll represent the integers between $s_i$ and $s_{i+1}$ by $\delta/2$ bits; these bits will appear in $\delta/128$ 64-bit numbers $sieve[j]$, where

$$sieve[j] = \sum_{n=s_i+128j}^{s_i+128(j+1)} 2^{(n-s_i-128j-1)/2}\,\bigl[n \text{ is an odd multiple of some odd prime} \le \sqrt{s_{i+1}}\,\bigr].$$

We choose the segment size $\delta$ to be a multiple of 128. We also assume that $s_0$ is even, and $s_0 \ge \sqrt{\delta}$. It follows that $s_i$ is even for all $i$, and that $(s_i+1)^2 = s_i^2 + s_i + s_{i+1} - \delta \ge s_i + s_{i+1} > s_{i+1}$. Consequently we have

$$sieve[j] = \sum_{n=s_i+128j}^{s_i+128(j+1)} 2^{(n-s_i-128j-1)/2}\,\bigl[n \text{ is odd and not prime}\bigr],$$

because $n$ appears if and only if it is divisible by some prime $p$ where $p \le \sqrt{s_{i+1}} < s_i + 1 \le n$.

**2.** The sieve size $\delta$ is specified at compile time, but $s_0$ and $t$ are specified on the command line when this program is run. There also are two additional command-line parameters, which name the input and output files.

The input file should contain all prime numbers $p_1$, $p_2$, $\ldots$, up to the first prime such that $p_k^2 > s_t$; it may also contain further primes, which are ignored. It is a binary file, with each prime given as an **unsigned int**. (There are 203,280,221 primes less than $2^{32}$, the largest of which is $2^{32}-5$. Thus I'm implicitly assuming that $s_t < (2^{32}-5)^2 \approx 1.8 \times 10^{19}$.)

The output file is a short text file that reports large gaps. Whenever the program discovers consecutive primes for which the gap $p_{k+1} - p_k$ is greater than or equal to all previously seen gaps, this gap is output (unless it is smaller than 256). The smallest and largest primes between $s_0$ and $s_t$ are also output, so that we can keep track of gaps between primes that are found by different instances of this program.

```
#define del 100000000LL   /* the segment size δ, a multiple of 128 */
#define kmax 10000   /* an index such that p²_kmax > s_t */

#include <stdio.h>
#include <stdlib.h>
  FILE *infile, *outfile;
  unsigned int prime[kmax];   /* prime[k] = p_{k+1} */
  unsigned int start[kmax];   /* indices for initializing a segment */
  unsigned long long sieve[2 + del/128];
  unsigned long long s0;   /* beginning of the first segment */
  int tt;   /* number of segments */
  unsigned long long st;   /* ending of the last segment */
  unsigned long long lastprime;   /* largest prime so far, if any */
  int bestgap = 256;   /* lower bound for gap reporting */
  unsigned long long sv[11];   /* bit patterns for the smallest primes */
  int rem[11];   /* shift amounts for the smallest primes */
  char nu[#10000];   /* table for counting bits */

  main(int argc, char *argv[])
  {
    register j, k;
    unsigned long long x, y, z, s, ss;
    int d, ii, kk;
    ⟨Initialize the bit-counting table 17⟩;
    ⟨Process the command line and input the primes 3⟩;
    ⟨Get ready for the first segment 7⟩;
    for (ii = 0; ii < tt; ii++) ⟨Do segment ii 8⟩;
    ⟨Report the final prime 19⟩;
  }
```

**3.** ⟨ Process the command line and input the primes 3 ⟩ ≡

```
if (argc ≠ 5 ∨ sscanf(argv[1], "%llu", &s0) ≠ 1 ∨ sscanf(argv[2], "%d", &tt) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣s[0]␣t␣inputfile␣outputfile\n", argv[0]);
  exit(−1);
}
infile = fopen(argv[3], "rb");
if (¬infile) {
  fprintf(stderr, "I␣can't␣open␣%s␣for␣binary␣input!\n", argv[3]);
  exit(−2);
}
outfile = fopen(argv[4], "w");
if (¬outfile) {
  fprintf(stderr, "I␣can't␣open␣%s␣for␣text␣output!\n", argv[4]);
  exit(−3);
}
st = s0 + tt ∗ del;
if (del % 128) {
  fprintf(stderr, "Oops:␣The␣sieve␣size␣%d␣isn't␣a␣multiple␣of␣128!\n", del);
  exit(−4);
}
if (s0 & 1) {
  fprintf(stderr, "The␣starting␣point␣%llu␣isn't␣even!\n", s0);
  exit(−5);
}
if (s0 ∗ s0 < del) {
  fprintf(stderr, "The␣starting␣point␣%llu␣is␣less␣than␣sqrt(%llu)!\n", s0, del);
  exit(−6);
}
⟨ Input the primes 4 ⟩;
printf("Sieving␣between␣s[0]=%llu␣and␣s[t]=%llu:\n", s0, st);
```

This code is used in section 2.

**4.** ⟨ Input the primes 4 ⟩ ≡

```
for (k = 0; ; k++) {
  if (k ≥ kmax) {
    fprintf(stderr, "Oops:␣Please␣recompile␣me␣with␣kmax>%d!\n", kmax);
    exit(−7);
  }
  if (fread(&prime[k], sizeof(unsigned int), 1, infile) ≠ 1) {
    fprintf(stderr, "The␣input␣file␣ended␣prematurely␣(%d^2<%llu)!\n", k ? prime[k − 1] : 0, st);
    exit(−8);
  }
  if (k ≡ 0 ∧ prime[0] ≠ 2) {
    fprintf(stderr, "The␣input␣file␣begins␣with␣%d,␣not␣2!\n", prime[0]);
    exit(−9);
  }
  else if (k > 0 ∧ prime[k] ≤ prime[k − 1]) {
    fprintf(stderr, "The␣input␣file␣has␣consecutive␣entries␣%d,%d!\n", prime[k − 1], prime[k]);
    exit(−10);
  }
  if (((unsigned long long) prime[k]) ∗ prime[k] > st) break;
}
printf("%d␣primes␣successfully␣loaded␣from␣%s\n", k, argv[3]);
```

This code is used in section 3.

**5. Sieving.** Let's say that the prime $p_k$ is "active" if $p_k^2 < s_{i+1}$. Variable *kk* is the index of the first inactive prime. The main task of sieving is to mark the multiples of all active primes in the current segment.

For each active prime $p_k$, let $n_k$ be the smallest odd multiple of $p_k$ that exceeds $s_i$. We let *start*[*k*] be $(n_k - s_i - 1)/2$, the bit offset of the first such multiple that needs to be marked.

At the beginning, we compute *start*[*k*] by division. But we'll be able to compute *start*[*k*] for subsequent segments as a byproduct of sieving, without division; that's why we bother to keep *start*[*k*] in memory.

```
⟨ Initialize the active primes 5 ⟩ ≡
  for (k = 1; ((unsigned long long) prime[k]) * prime[k] < s0; k++) {
    j = s0 % prime[k];
    if (j & 1) start[k] = prime[k] − ((j + 1) ≫ 1);
    else start[k] = (prime[k] − j − 1) ≫ 1;
  }
  kk = k;
  ⟨ Initialize the tiny active primes 6 ⟩;
```

This code is used in section 7.

**6.** Primes less than 32 will appear at least twice in every octabyte of the sieve. So we handle them in a slightly more efficient way, unless they're initially inactive.

```
⟨ Initialize the tiny active primes 6 ⟩ ≡
  for (k = 1; prime[k] < 32 ∧ k < kk; k++) {
    for (x = 0, y = 1LL ≪ start[k]; x ≠ y; x = y, y |= y ≪ prime[k]) ;
    sv[k] = x, rem[k] = 64 % prime[k];
  }
  d = k;    /* d is the index of the smallest nontiny prime */
```

This code is used in section 5.

**7.**

```
⟨ Get ready for the first segment 7 ⟩ ≡
  ⟨ Initialize the active primes 5 ⟩;
  ss = s0;    /* base address of the next segment */
  sieve[1 + del/128] = −1;    /* store a sentinel */
```

This code is used in section 2.

**8.**

```
⟨ Do segment ii 8 ⟩ ≡
  {
    s = ss, ss = s + del;    /* s = s_i, ss = s_{i+1} */
    printf("Beginning␣segment␣%llu\n", s);
    ⟨ Initialize the sieve from the tiny primes 9 ⟩;
    ⟨ Sieve in the previously active primes 10 ⟩;
    ⟨ Sieve in the newly active primes 11 ⟩;
    ⟨ Look for large gaps 12 ⟩;
  }
```

This code is used in section 2.

**9.** ⟨ Initialize the sieve from the tiny primes 9 ⟩ ≡

```
for (j = 0; j < del/128; j++) {
  for (z = 0, k = 1; k < d; k++) {
    z |= sv[k];
    sv[k] = (sv[k] << (prime[k] - rem[k])) | (sv[k] >> rem[k]);
  }
  sieve[j] = z;
}
```

This code is used in section 8.

**10.** Now we want to set 1 bits for every odd multiple of *prime*[*k*] in the current segment, whenever *prime*[*k*] is active. The bit for the integer $s_i + 2j + 1$ is $1 \ll (j \,\&\, \#3f)$ in *sieve*[$j \gg 6$], for $0 \le j < \delta/2$.

⟨ Sieve in the previously active primes 10 ⟩ ≡

```
for (k = d; k < kk; k++) {
  for (j = start[k]; j < del/2; j += prime[k]) sieve[j >> 6] |= 1LL << (j & #3f);
  start[k] = j - del/2;
}
```

This code is used in section 8.

**11.** ⟨ Sieve in the newly active primes 11 ⟩ ≡

```
while (((unsigned long long) prime[k]) * prime[k] < ss) {
  for (j = (((unsigned long long) prime[k]) * prime[k] - s - 1) >> 1; j < del/2; j += prime[k])
    sieve[j >> 6] |= 1LL << (j & #3f);
  start[k] = j - del/2;
  k++;
}
kk = k;
```

This code is used in section 8.

**12. Processing gaps.** If $p_{k+1} - p_k \ge 256$, we're bound to find an octabyte of all 1s in the sieve between the 0 for $p_k$ and the 0 for $p_{k+1}$. In such cases, we check to see if this gap breaks or ties the current record.

Complications occur if the gap appears at the very beginning or end of a segment, or if an entire segment is prime-free. I've tried to get the logic correct, without slowing the program down. But if any bugs are present in this code, I suppose they are due to a fallacy in this aspect of my reasoning.

Two sentinels appear at the end of the sieve, in order to speed up loop termination: $sieve[del/128] = 0$ and $sieve[1 + del/128] = -1$.

```
⟨Look for large gaps 12⟩ ≡
  j = 0;
  ⟨Identify the first prime in this segment, if necessary 13⟩;
  while (1) {     /* at this point j < del/128 and sieve[j] ≠ −1 */
    for (j++; sieve[j] ≠ −1; j++) ;
    if (j < del/128) {
      k = j − 1;
      for (j++; sieve[j] ≡ −1; j++) ;
      if (j ≡ del/128) break;
      ⟨Check for a potentially interesting gap 14⟩;
    } else {     /* j = 1 + del/128 and sieve[del/128 − 1] ≠ −1 */
      k = del/128 − 1; break;
    }
  }
  ⟨Set lastprime to the largest prime in sieve[k] 15⟩;
donewithseg:
```

This code is used in section 8.

**13.** We don't need to figure out the exact value of the first prime greater than $s$ unless the present segment begins with an octabyte of all 1s, or the previous segment ends with such an octabyte, or we're in the first segment.

But in any case we'll want to go immediately to *donewithseg* if the current segment is entirely prime-free. And we always want to end this step with $j$ equal to the smallest index such that $sieve[j] \ne -1$.

```
⟨Identify the first prime in this segment, if necessary 13⟩ ≡
  if (lastprime ≤ s − 128 ∨ sieve[j] ≡ −1) {
    for ( ; sieve[j] ≡ −1; j++) ;
    if (j ≡ del/128) goto donewithseg;
    y = ∼sieve[j];
    y = y & −y;     /* extract the rightmost 1 bit */
    ⟨Change y to its binary logarithm 16⟩;
    x = s + (j ≪ 7) + y + y + 1;     /* this is the first prime of the segment */
    if (lastprime) ⟨Report a gap, if it's big enough 18⟩
    else {
      k = x − s0;
      fprintf(outfile, "The␣first␣prime␣is␣%llu␣=␣s[0]+%d\n", x, k);
      fflush(outfile);
    }
  }
```

This code is used in section 12.

**14.** When $sieve[k] \neq -1$ and $sieve[j] \neq -1$ and everything between them is $-1$ (all ones), there's a gap of size $g$ where $128|j-k| - 126 \le g \le 128|j-k| + 126$.

⟨ Check for a potentially interesting gap 14 ⟩ ≡

```
if (((j − k) ≪ 7) + 126 ≥ bestgap) {
    y = ∼sieve[j];
    y = y & −y;    /* extract the rightmost 1 bit */
    ⟨ Change y to its binary logarithm 16 ⟩;
    x = s + (j ≪ 7) + y + y + 1;    /* this is the first prime after the gap */
    ⟨ Set lastprime to the largest prime in sieve[k] 15 ⟩;
    ⟨ Report a gap, if it's big enough 18 ⟩;
}
```

This code is used in section 12.

**15.** ⟨ Set *lastprime* to the largest prime in *sieve*[*k*] 15 ⟩ ≡

```
for (y = ∼sieve[k], z = y & (y − 1); z; y = z, z = y & (y − 1)) ;
⟨ Change y to its binary logarithm 16 ⟩;
lastprime = s + (k ≪ 7) + y + y + 1;
```

This code is used in sections 12 and 14.

**16.** As far as I know, the following method is the fastest way to compute binary logarithms on an Opteron computer (which is the machine I'm targeting here).

⟨ Change *y* to its binary logarithm 16 ⟩ ≡

```
y−−;
y = nu[y & #ffff] + nu[(y ≫ 16) & #ffff] + nu[(y ≫ 32) & #ffff] + nu[(y ≫ 48) & #ffff];
```

This code is used in sections 13, 14, and 15.

**17.** With a more extensive table, I could count the 1s in an arbitrary binary word. But seventeen table entries are sufficient for present purposes.

⟨ Initialize the bit-counting table 17 ⟩ ≡

```
for (j = 0; j ≤ 16; j++) nu[((1 ≪ j) − 1)] = j;
```

This code is used in section 2.

**18.** ⟨ Report a gap, if it's big enough 18 ⟩ ≡

```
{
    if (x − lastprime ≥ bestgap) {
        bestgap = x − lastprime;
        fprintf(outfile, "%llu␣is␣followed␣by␣a␣gap␣of␣length␣%d\n", lastprime, bestgap);
        fflush(outfile);
    }
}
```

This code is used in sections 13 and 14.

**19.** ⟨ Report the final prime 19 ⟩ ≡

```
if (lastprime) {
    k = st − lastprime;
    fprintf(outfile, "The␣final␣prime␣is␣%llu␣=␣s[t]-%d.\n", lastprime, k);
} else fprintf(outfile, "No␣prime␣numbers␣exist␣between␣s[0]␣and␣s[t].\n");
```

This code is used in section 2.

**20. Index.**

# PRIME-SIEVE