

(Downloaded from <https://cs.stanford.edu/~knuth/programs.html> and typeset on September 17, 2017)

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 $i = 0, 1, \dots, t-1$. 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 $\leq N$ by starting with the set $\{2, 3, \dots, N\}$ 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 $p_k^2 \leq s_t$. 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

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

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

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

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

In this “sparse” version I actually consider only integers of the form $4m+1$, and I require δ to be a multiple of 256. I also require s_0 to be a multiple of 4. Thus the sieve now contains $\delta/256$ octabytes. Reason: A gap of size g between ordinary primes implies a gap of size $\geq g$ between primes of the form $4m+1$. If $g \geq 1000$, such gaps are sufficiently rare that I think it’s faster to check their true size by brute force, because we save a factor of two with the sparse sieve.

“Brute force” in the previous paragraph means actually a pseudoprime test, using Miller and Rabin’s method. If that test passes, the probability exceeds $1 - 2^{-64}$ that I’ve incorrectly classified a composite number as a prime.

Although I haven’t had much time to experiment with this program, limited experience has shown that the cache size of the host computer has a significant effect on speed. Therefore — counterintuitively — it proves to be best to work with rather small segments. In fact, for numbers in the range of current interest to me (say 4×10^{17} , most of the primes may well exceed 50δ).

So this program uses an idea that I found on Tomás Oliveira e Silva’s web site: There’s a cyclic queue of size q , with lists of the primes that become relevant in each future segment and their starting places.

2* The sieve size δ and queue size q are specified at compile time. They are preferably powers of two, because we'll want to divide by δ and compute remainders modulo q .

The other fundamental parameters s_0 and t are specified on the command line when this program is run. And there 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, \dots , 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 in this “bootstrap” version is a list of all primes $\leq s_t$, in a format suitable for use as the input file in the regular version.

The compile-time parameter $lsize$ is somewhat delicate. We need $8qsize \times lsize$ bytes of RAM, so we don't want $lsize$ to be too large. On the other hand $lsize$ has to be large enough to accommodate the queue lists as the program runs. A large $lsize$ might force $qsize$ to be small, and that will slow things down because primes will be before they're needed.

```
#define del ((long long)(1 << 23)) /* the segment size  $\delta$ , a multiple of 256 */
#define qsize (1 << 7) /* the queue size  $q$  */
#define kmax 35000000 /* an index such that  $p_{kmax}^2 > s_t$  */
#define ksmall 156000 /* an index such that  $p_{ksmall} > \delta/4$  */
#define bestgap 1000 /* lower bound for gap reporting,  $\geq 512$ , a multiple of 4 */
#define lsize (1 << 20) /* size of queue lists, hopefully big enough */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

FILE *infile, *outfile;
unsigned int prime[kmax]; /*  $prime[k] = p_{k+1}$  */
unsigned int start[ksmall]; /* indices for initializing a segment */
unsigned int plist[qsize][lsize]; /* primes queued for a segment */
unsigned int slist[qsize][lsize]; /* their relative starting points */
int count[qsize]; /* number of entries in queue lists */
int countmax; /* the largest count we've needed so far */
unsigned long long sieve[2 + del/256];
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 */
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 */
int timer, starttime;

<Subroutines 22>

main(int argc, char *argv[])
{
    register j, jj, k;
    unsigned long long x, xx, y, z, s, ss;
    int d, dd, ii, kk, qq;

    starttime = timer = time(0);
    <Initialize the bit-counting table 18>;
    <Initialize the random number generator 24>;
    <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*>;
}
```

```

printf("Finished; the last segment took %d sec; total time %.6g hours.)\n",
      time(0) - timer, ((double)(time(0) - starttime))/3600.0);
printf("The maximum list size needed was %d.\n", countmax);
}

```

3* \langle Process the command line and input the primes **3*** $\rangle \equiv$

```

if (argc  $\neq$  5  $\vee$  sscanf(argv[1], "%llu", &s0)  $\neq$  1  $\vee$  sscanf(argv[2], "%d", &tt)  $\neq$  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], "wb");
if (!outfile) {
    fprintf(stderr, "I can't open %s for binary output!\n", argv[4]);
    exit(-3);
}
st = s0 + tt * del;
if (st > #ffffffff) {
    fprintf(stderr, "Sorry, %s[t] = %llu exceeds 32 bits!\n", st);
    exit(-69);
}
if (del % 256) {
    fprintf(stderr, "Oops: The sieve size %d isn't a multiple of 256!\n", del);
    exit(-4);
}
if (s0 & 3) {
    fprintf(stderr, "The starting point %llu isn't a multiple of 4!\n", s0);
    exit(-5);
}
if (s0 * s0 < del) {
    fprintf(stderr, "The starting point %llu is less than sqrt(%llu)!\n", s0, del);
    exit(-6);
}
 $\langle$  Input the primes 4  $\rangle$ ;
printf("Sieving between %s[0] = %llu and %s[t] = %llu:\n", s0, st);

```

This code is used in section **2***.

4. Primes are divided into three classes: small, medium, and large. The small primes (actually “tiny”) are less than 32; they appear at least twice in every octabyte of the sieve. The large primes are greater than $\delta/4$; they appear at most once in every segment of the sieve.

Since our sieve represents integers of the form $4k + 1$, every segment consists of $\delta/256$ octabytes.

```
#define ddel (del/4) /* number of bits per segment */
⟨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 (prime[k] < ddel) {
            if (k ≥ ksmall) {
                fprintf(stderr, "Oops: Please recompile me with ksmall>%d!\n", ksmall);
                exit(-11);
            }
            dd = k + 1; /* dd will be the index of the first large prime */
        }
        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 multiple of p_k that exceeds s_i and is congruent to 1 modulo 4. We let $start[k]$ be $(n_k - s_i - 1)/4$, 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.

(Actually $start[k]$ is computed explicitly only for the small and medium-sized primes. An equivalent starting point for each large active prime is recorded in its appropriate queue list.)

⟨Initialize the active primes 5⟩ ≡

```

for ( $k = 1$ ; ((unsigned long long)  $prime[k]$ ) *  $prime[k] < s0$ ;  $k++$ ) {
     $j = (((\mathbf{long\ long})(prime[k] \& 3) * prime[k]) \gg 2) - (\mathbf{long\ long})(s0 \gg 2) \% prime[k]$ ;
    if ( $j < 0$ )  $j += prime[k]$ ;
    if ( $k < dd$ )  $start[k] = j$ ;
    else {
         $jj = (j / ddel) \% qsize$ ;
        if ( $count[jj] \equiv countmax$ ) {
             $countmax++$ ;
            if ( $countmax \geq lsize$ ) {
                 $fprintf(stderr, "Oops: \_Please\_recompile\_me\_with\_lsize> \%d! \backslash n", lsize)$ ;
                 $exit(-12)$ ;
            }
        }
         $plist[jj][count[jj]] = prime[k]$ ;
         $slist[jj][count[jj]] = j$ ;
         $count[jj]++$ ;
    }
}
 $kk = k$ ;

```

⟨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 ( $k = 1$ ;  $prime[k] < 32 \wedge k < kk$ ;  $k++$ ) {
    for ( $x = 0, y = 1_{LL} \ll start[k]$ ;  $x \neq y$ ;  $x = y, y |= y \ll prime[k]$ ) ;
         $sv[k] = x, rem[k] = 64 \% prime[k]$ ;
}
 $d = k$ ; /*  $d$  is the smallest nontiny prime */

```

This code is used in section 5.

7*: ⟨Get ready for the first segment 7*⟩ ≡

```

⟨Output the primes that precede the first segment 33*⟩;
⟨Initialize the active primes 5⟩;
 $ss = s0$ ; /* base address of the next segment */
 $sieve[1 + del/256] = -1$ ; /* store a sentinel */

```

This code is used in section 2*.

```

8*  ⟨ Do segment ii 8* ⟩ ≡
{
    s = ss, ss = s + del, qq = ii % qsize;    /* s = si, ss = si+1 */
    if (qq ≡ 0) {
        j = time(0);
        printf("Beginning segment %llu (after %d sec)\n", s, j - timer);
        fflush(stdout);
        timer = j;
    }
    ⟨ Initialize the sieve from the tiny primes 9 ⟩;
    ⟨ Sieve in the previously active primes 10 ⟩;
    ⟨ Sieve in the newly active primes 12 ⟩;
    ⟨ Output the primes in the current segment 34* ⟩;
}

```

This code is used in section **2***.

```

9.  ⟨ Initialize the sieve from the tiny primes 9 ⟩ ≡
for (j = 0; j < del/256; 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 $s_i + 4j + 1$ is $1 \ll (j \& \#3f)$ in $sieve[j \gg 6]$, for $0 \leq j < \delta/4$.

```

⟨ Sieve in the previously active primes 10 ⟩ ≡
if (dd ≥ kk) { /* no large primes are active */
    for (k = d; k < kk; k++) {
        for (j = start[k]; j < ddel; j += prime[k]) sieve[j >> 6] |= 1LL << (j & #3f);
        start[k] = j - ddel;
    }
} else {
    for (k = d; k < dd; k++) {
        for (j = start[k]; j < ddel; j += prime[k]) sieve[j >> 6] |= 1LL << (j & #3f);
        start[k] = j - ddel;
    }
    ⟨ Sieve in the enqueued large primes 11 ⟩;
}

```

This code is used in section **8***.

11. Each *slist* entry is an offset relative to the beginning of the previous segment with $qq = 0$. Thus, for example, *slist*[1] holds numbers of the form $ddel + x$, $ddel * (1 + qsize) + x$, $ddel * (1 + 2 * qsize) + x$, etc., where $0 \leq x < ddel$.

⟨ Sieve in the enqueued large primes 11 ⟩ ≡

```

for ( $j = k = 0$ ;  $k < count[qq]$ ;  $k++$ ) {
    if ( $slist[qq][k] \geq (qq + 1) * ddel$ ) /* big big prime has “looped” the queue */
         $plist[qq][j] = plist[qq][k]$ ,  $slist[qq][j] = slist[qq][k] - qsize * ddel$ ,  $j++$ ;
    else {
        register unsigned int  $nstart$ ;
         $jj = slist[qq][k] \% ddel$ ;
         $sieve[jj \gg 6] |= 1_{LL} \ll (jj \& \#3f)$ ;
         $nstart = slist[qq][k] + plist[qq][k]$ ;
         $jj = (nstart / ddel) \% qsize$ ; /* possibly  $jj = qq$ ; that’s no problem */
        if ( $count[jj] \equiv countmax$ ) {
             $countmax++$ ;
            if ( $countmax \geq lsize$ ) {
                 $fprintf(stderr, "Oops: \_Please\_recompile\_me\_with\_lsize> \%d! \backslash n", lsize)$ ;
                 $exit(-13)$ ;
            }
        }
         $plist[jj][count[jj]] = plist[qq][k]$ ;
         $slist[jj][count[jj]] = (jj \geq qq ? nstart : nstart - qsize * ddel)$ ;
         $count[jj]++$ ;
    }
}
 $count[qq] = j$ ;

```

This code is used in section 10.

12. The test here is ‘ $jj > qq$ ’ when we construct an *slist* entry, not ‘ $jj \geq qq$ ’ as before. Do you see why?

⟨ Sieve in the newly active primes 12 ⟩ ≡

```

for ( $k = kk$ ;  $((\text{unsigned long long}) prime[k]) * prime[k] < ss$ ;  $k++$ ) {
    for ( $j = (((\text{unsigned long long}) prime[k]) * prime[k] - s - 1) \gg 2$ ;  $j < ddel$ ;  $j += prime[k]$ )
         $sieve[j \gg 6] |= 1_{LL} \ll (j \& \#3f)$ ;
    if ( $k < dd$ )  $start[k] = j - ddel$ ;
    else {
         $j += qq * ddel$ ;
         $jj = (j / ddel) \% qsize$ ; /* possibly  $jj = qq$ ; that’s no problem */
        if ( $count[jj] \equiv countmax$ ) {
             $countmax++$ ;
            if ( $countmax \geq lsize$ ) {
                 $fprintf(stderr, "Oops: \_Please\_recompile\_me\_with\_lsize> \%d! \backslash n", lsize)$ ;
                 $exit(-14)$ ;
            }
        }
         $plist[jj][count[jj]] = prime[k]$ ;
         $slist[jj][count[jj]] = (jj > qq ? j : j - qsize * ddel)$ ;
         $count[jj]++$ ;
    }
}
 $kk = k$ ;

```

This code is used in section 8*.

13. Processing gaps. If $p_{k+1} - p_k \geq 512$, 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 for a potential "kilogap" (a gap of length 1000 or more).

Complications occur if the gap appears at the very beginning or end of a segment, or if an entire segment is prime-free. Further complications arise because our sieve contains only half of the potential primes. 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: $sieve[del/256] = 0$ and $sieve[1 + del/256] = -1$.

```

⟨Look for large gaps 13⟩ ≡
  j = 0, k = -100;
  while (1) {
    for ( ; sieve[j] ≡ -1; j++ ) ;
    if (j ≡ del/256) x = ss;
    else ⟨Set x to the smallest prime in sieve[j] 15⟩;
    if (k ≥ 0) ⟨Set lastprime to the largest prime in sieve[k] 16⟩
    else if (lastprime ≡ 0) ⟨Set lastprime to the smallest prime ≥ s0 14⟩;
    ⟨Look for and report any large gaps between lastprime and x 19⟩;
    if (j ≡ del/256) break;
    for (j++; sieve[j] ≠ -1; j++ ) ;
    if (j < del/256) k = j - 1;
    else { /* j = 1 + del/256 and sieve[del/256 - 1] ≠ -1 */
      k = del/256 - 1;
      ⟨Set lastprime to the largest prime in sieve[k] 16⟩;
      break;
    }
  }
  for (z = ss - 1; z > lastprime; z -= 4)
    if (isprime(z)) {
      lastprime = z; break;
    }
  donewithseg:

```

```

14. ⟨Set lastprime to the smallest prime ≥ s0 14⟩ ≡
  {
    for (z = s + 3; z < x; z += 4)
      if (isprime(z)) {
        lastprime = z; goto got_it;
      }
    if (x ≡ ss) goto donewithseg; /* no primes at all below ss! */
    lastprime = x;
    got_it: fprintf(outfile, "The first prime is %llu = s[0] + %d\n", lastprime, lastprime - s0);
    fflush(outfile);
  }

```

This code is used in section 13.

15. \langle Set x to the smallest prime in $sieve[j]$ **15** $\rangle \equiv$
 $\{$
 $y = \sim sieve[j];$
 $y = y \& -y; \quad /* \text{ extract the rightmost 1 bit } */$
 \langle Change y to its binary logarithm **17** $\rangle;$
 $x = s + (j \ll 8) + (y \ll 2) + 1; \quad /* \text{ this upperbounds the first prime after a gap } */$
 $\}$

This code is used in section **13**.

16. \langle Set $lastprime$ to the largest prime in $sieve[k]$ **16** $\rangle \equiv$
 $\{$
 $\text{for } (y = \sim sieve[k], z = y \& (y - 1); z; y = z, z = y \& (y - 1)) ; \quad /* \text{ the leftmost 1 bit } */$
 \langle Change y to its binary logarithm **17** $\rangle;$
 $lastprime = s + (k \ll 8) + (y \ll 2) + 1;$
 $\}$

This code is used in section **13**.

17. 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).

\langle Change y to its binary logarithm **17** $\rangle \equiv$
 $y--;$
 $y = nu[y \& \#ffff] + nu[(y \gg 16) \& \#ffff] + nu[(y \gg 32) \& \#ffff] + nu[(y \gg 48) \& \#ffff];$

This code is used in sections **15**, **16**, and **34***.

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

\langle Initialize the bit-counting table **18** $\rangle \equiv$
 $\text{for } (j = 0; j \leq 16; j++) \text{ } nu[((1 \ll j) - 1)] = j;$

This code is used in section **2***.

19. 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 $256|j - k| - 126 \leq g \leq 256|j - k| + 126$.

If $k < 0$ and $lastprime \neq 0$, there are no primes between $lastprime$ and s .

Two or more large gaps may actually be present, in a long interval where the only primes are of the form $4m + 3$. (I doubt if this actually occurs until the numbers get much larger than I can handle, but I'm trying to make the program correct.)

⟨Look for and report any large gaps between $lastprime$ and x 19⟩ \equiv

```

if ( $j \geq k + bestgap/256$ ) {
     $xx = x$ ;
    zloop: if ( $x - lastprime < bestgap$ ) goto done_here;
     $y = (k \geq 0 ? lastprime : s)$ ;
    for ( $z = ((lastprime \& \sim 2) + bestgap - 2)$ ;  $z > y$ ;  $z -= 4$ )
        if (isprime( $z$ )) {
             $lastprime = z, k = 0$ ; goto zloop;
        }
     $z = (lastprime \& \sim 2) + bestgap + 2$ ;
    if ( $z < s$ )  $z = s + 3$ ;
    for ( ;  $z < x$ ;  $z += 4$ )
        if (isprime( $z$ )) {
             $x = z$ ; break;
        }
    if ( $x \equiv ss$ ) goto donewithseg; /*  $lastprime$  is the largest prime less than  $x$  */
    ⟨Report a gap, if it's big enough 20⟩;
     $lastprime = x, x = xx$ ; goto zloop;
}
done_here:
```

This code is used in section 13.

20. ⟨Report a gap, if it's big enough 20⟩ \equiv

```

{
    if ( $x - lastprime \geq bestgap$ ) {
        fprintf(outfile, "%llu is followed by a gap of length %d\n",  $lastprime, x - lastprime$ );
        fflush(outfile);
    }
}
```

This code is used in section 19.

21. ⟨Report the final prime 21⟩ \equiv

```

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

22. Random numbers. The following code comes directly from `rng.c`, the random number generator in Section 3.6.

```
#define KK 100      /* the long lag */
#define LL 37       /* the short lag */
#define MM (1_L << 30) /* the modulus */
#define mod_diff(x,y) (((x) - (y)) & (MM - 1)) /* subtraction mod MM */

⟨Subroutines 22⟩ ≡
    long ran_x[KK]; /* the generator state */
    void ran_array(long aa[], int n)
    {
        register int i, j;
        for (j = 0; j < KK; j++) aa[j] = ran_x[j];
        for (; j < n; j++) aa[j] = mod_diff(aa[j - KK], aa[j - LL]);
        for (i = 0; i < LL; i++, j++) ran_x[i] = mod_diff(aa[j - KK], aa[j - LL]);
        for (; i < KK; i++, j++) ran_x[i] = mod_diff(aa[j - KK], ran_x[i - LL]);
    }
```

See also sections 23, 25, 26, and 27.

This code is used in section 2*.

```

23. #define QUALITY 1009 /* recommended quality level for high-res use */
#define TT 70 /* guaranteed separation between streams */
#define is_odd(x) ((x) & 1) /* units bit of x */
⟨Subroutines 22⟩ +≡
    long ran_arr_buf[QUALITY];
    long ran_arr_dummy = -1, ran_arr_started = -1;
    long *ran_arr_ptr = &ran_arr_dummy; /* the next random number, or -1 */
    void ran_start(long seed)
    {
        register int t, j;
        long x[KK + KK - 1]; /* the preparation buffer */
        register long ss = (seed + 2) & (MM - 2);
        for (j = 0; j < KK; j++) {
            x[j] = ss; /* bootstrap the buffer */
            ss <<= 1;
            if (ss ≥ MM) ss -= MM - 2; /* cyclic shift 29 bits */
        }
        x[1]++; /* make x[1] (and only x[1]) odd */
        for (ss = seed & (MM - 1), t = TT - 1; t; ) {
            for (j = KK - 1; j > 0; j--) x[j + j] = x[j], x[j + j - 1] = 0; /* "square" */
            for (j = KK + KK - 2; j ≥ KK; j--)
                x[j - (KK - LL)] = mod_diff(x[j - (KK - LL)], x[j]), x[j - KK] = mod_diff(x[j - KK], x[j]);
            if (is_odd(ss)) { /* "multiply by z" */
                for (j = KK; j > 0; j--) x[j] = x[j - 1];
                x[0] = x[KK]; /* shift the buffer cyclically */
                x[LL] = mod_diff(x[LL], x[KK]);
            }
            if (ss) ss >>= 1;
            else t--;
        }
        for (j = 0; j < LL; j++) ran_x[j + KK - LL] = x[j];
        for (; j < KK; j++) ran_x[j - LL] = x[j];
        for (j = 0; j < 10; j++) ran_array(x, KK + KK - 1); /* warm things up */
        ran_arr_ptr = &ran_arr_started;
    }

```

24. ⟨Initialize the random number generator 24⟩ ≡

```
ran_start(314159L);
```

This code is used in section 2*.

25. After calling *ran_start*, we get new randoms by saying “*x* = *ran_arr_next*()”.

```
#define ran_arr_next() (*ran_arr_ptr ≥ 0 ? *ran_arr_ptr++ : ran_arr_cycle())
```

⟨Subroutines 22⟩ +≡

```

    long ran_arr_cycle()
    {
        if (ran_arr_ptr ≡ &ran_arr_dummy) ran_start(314159L); /* the user forgot to initialize */
        ran_array(ran_arr_buf, QUALITY);
        ran_arr_buf[KK] = -1;
        ran_arr_ptr = ran_arr_buf + 1;
        return ran_arr_buf[0];
    }

```

26. Double precision multiplication. We'll need a subroutine that computes the 128-bit product of two 64-bit integers. The product goes into *acc_hi* and *acc_lo*.

⟨Subroutines 22⟩ +≡

```

unsigned long long acc_hi, acc_lo;
void mult(unsigned long long x, unsigned long long y)
{
    register unsigned int xhi, xlo, yhi, ylo;
    unsigned long long t;
    xhi = x >> 32, xlo = x & #ffffff;
    yhi = y >> 32, ylo = y & #ffffff;
    t = ((unsigned long long) xlo) * ylo, acc_lo = t & #ffffff;
    t = ((unsigned long long) xhi) * ylo + (t >> 32), acc_hi = t >> 32;
    t = ((unsigned long long) xlo) * yhi + (t & #ffffff);
    acc_hi += ((unsigned long long) xhi) * yhi + (t >> 32);
    acc_lo += (t & #ffffff) << 32;
}
```

27. Prime testing. I've saved the most interesting part of this program for last. It's a subroutine that tries to decide whether a given **long long** number z is prime. In the experiments I'm doing, z lies between 2^{58} and 2^{59} (but the program does not require that z be in this range).

If it's easy to determine that z is definitely not prime, the subroutine returns 0.

But if z passes the Miller–Rabin test for 32 different random witnesses, the subroutine returns 1.

A nonprime number almost never returns 1. In fact, a nonprime number that passes the test even once is sufficiently interesting that I'm printing it out.

Here I implement Algorithm 4.5.4P, using the fact that $z \bmod 4 = 3$, and using “Montgomery multiplication” for speed (exercise 4.3.1–41).

⟨Subroutines 22⟩ +≡

```

int isprime(unsigned long long z)
{
    register int k, lgz, rep;
    long long x, y, q;
    unsigned long long m, zp, goal;
    ⟨If  $z$  is divisible by a prime  $\leq 53$ , return 0 32⟩;
    ⟨Get ready for Montgomery's method 28⟩;
    for (rep = 0; rep < 32; rep++) {
        P1:  $x = \text{ran\_arr\_next}()$ ;
        P2:  $q = z \gg 1$ ;
        for ( $y = x, m = 1_{\text{LL}} \ll (lgz - 2)$ ;  $m; m \gg = 1$ ) {
            ⟨Set  $y \leftarrow (y^2/2^{64}) \bmod z$  30⟩;
            if ( $m \& q$ ) ⟨Set  $y \leftarrow (xy/2^{64}) \bmod z$  31⟩;
        }
        if ( $y \neq \text{goal} \wedge y \neq z - \text{goal}$ ) {
            if (rep) {
                 $\text{fprintf}(\text{outfile}, "( \%lld\_is\_a\_pseudoprime\_of\_rank\_d)\n", z, \text{rep})$ ;
                 $\text{fflush}(\text{outfile})$ ;
            }
            return 0;
        }
    }
    return 1;
}

```

28. Miller and Rabin's algorithm is based on the fact that $x^q \equiv \pm 1$ (modulo z) when z is prime and $q = (z - 1)/2$. The loop above actually computes $(2^{64}(x/2^{64})^q) \bmod z$, so the result should be $(\pm 2^{64}) \bmod z$.

Montgomery's method also needs the constant z' such that $zz' \equiv 1$ (modulo 2^{64}).

⟨Get ready for Montgomery's method 28⟩ ≡

```

for (lgz = 63, m = #8000000000000000; ( $m \& z$ )  $\equiv 0$ ;  $m \gg = 1, lgz--$ ) ;
for ( $k = lgz, \text{goal} = m; k < 64; k++$ ) {
     $\text{goal} += \text{goal}$ ;
    if ( $\text{goal} \geq z$ )  $\text{goal} -= z$ ;
} /* now  $\text{goal} = 2^{64} \bmod z$  */
⟨Set zp to the inverse of  $z$  modulo  $2^{64}$  29⟩;

```

This code is used in section 27.

29. Here I'm using "Newton's method." (If $z \bmod 4 = 1$, the first step should be changed to $zp = (z \& 4 ? z \oplus 8 : z)$.)

```

⟨ Set  $zp$  to the inverse of  $z$  modulo  $2^{64}$  29 ⟩ ≡
{
     $zp = (z \& 4 ? z : z \oplus 8)$ ;      /*  $zz' \equiv 1 \pmod{2^4}$ , because  $z \bmod 4 = 3$  */
     $zp = (2 - zp * z) * zp$ ;          /* now  $zz' \equiv 1 \pmod{2^8}$  */
     $zp = (2 - zp * z) * zp$ ;          /* now  $zz' \equiv 1 \pmod{2^{16}}$  */
     $zp = (2 - zp * z) * zp$ ;          /* now  $zz' \equiv 1 \pmod{2^{32}}$  */
     $zp = (2 - zp * z) * zp$ ;          /* now  $zz' \equiv 1 \pmod{2^{64}}$  */
}

```

This code is used in section 28.

30. To compute $xy/2^{64} \bmod z$, we compute the 128-bit product $xy = 2^{64}t_1 + t_0$, then subtract $(z't_0 \bmod 2^{64})z$ and return the leading 64 bits.

```

⟨ Set  $y \leftarrow (y^2/2^{64}) \bmod z$  30 ⟩ ≡
{
     $mult(y, y)$ ;
     $y = acc\_hi$ ;
     $mult(zp * acc\_lo, z)$ ;
    if ( $y < acc\_hi$ )  $y += z - acc\_hi$ ;
    else  $y -= acc\_hi$ ;
}

```

This code is used in section 27.

```

31. ⟨ Set  $y \leftarrow (xy/2^{64}) \bmod z$  31 ⟩ ≡
{
     $mult(x, y)$ ;
     $y = acc\_hi$ ;
     $mult(zp * acc\_lo, z)$ ;
    if ( $y < acc\_hi$ )  $y += z - acc\_hi$ ;
    else  $y -= acc\_hi$ ;
}

```

This code is used in section 27.

32. The following simple test for nonprimality will rule out most cases before we need to resort to the Miller–Rabin scheme. Algorithm 4.5.2B is a nice divisionless method to use here. (Note that the product $3 \cdot 5 \cdot \dots \cdot 53$ is between 2^{63} and 2^{64} , so it would be considered “negative” as a **long long**.)

```
#define magic
      ((3LL*5LL*7LL*11LL*13LL*17LL*19LL*23LL*29LL*31LL*37LL*41LL*43LL*47LL*(unsigned
      long long) 53) >> 1)
⟨ If  $z$  is divisible by a prime  $\leq 53$ , return 0 32 ⟩ ≡
{
    long long  $u, v, t$ ;
     $t = \text{magic} - (z \gg 1)$ ;
     $v = z$ ;
B4: while  $((t \& 1) \equiv 0)$   $t \gg= 1$ ;
B5: if  $(t > 0)$   $u = t$ ; else  $v = -t$ ;
B6:  $t = (u - v)/2$ ;
    if  $(t)$  goto B4;
    if  $(u > 1)$  return 0;
}
```

This code is used in section 27.

33* ⟨ Output the primes that precede the first segment 33* ⟩ ≡
 for $(k = 0; \text{prime}[k] < s0; k++)$ $\text{fwrite}(\&\text{prime}[k], \text{sizeof}(\text{int}), 1, \text{outfile})$;

This code is used in section 7*.

34* ⟨ Output the primes in the current segment 34* ⟩ ≡
 for $(j = 0; j < \text{del}/128; j++)$ {
 for $(x = \sim \text{sieve}[j]; x; x = x \& (x - 1))$ {
 $y = x \& -x$; /* extract the rightmost 1 bit */
 ⟨ Change y to its binary logarithm 17 ⟩;
 $\text{lastprime} = s + (j \ll 7) + y + y + 1$; /* this is the first prime after the gap */
 $\text{fwrite}(\&\text{lastprime}, \text{sizeof}(\text{int}), 1, \text{outfile})$;
 }
 }

This code is used in section 8*.

35* Index.

The following sections were changed by the change file: [2](#), [3](#), [7](#), [8](#), [33](#), [34](#), [35](#).

aa: [22](#).
acc_hi: [26](#), [30](#), [31](#).
acc_lo: [26](#), [30](#), [31](#).
argc: [2*](#), [3*](#).
argv: [2*](#), [3*](#), [4](#).
bestgap: [2*](#), [19](#), [20](#).
B4: [32](#).
B5: [32](#).
B6: [32](#).
count: [2*](#), [5](#), [11](#), [12](#).
countmax: [2*](#), [5](#), [11](#), [12](#).
d: [2*](#).
dd: [2*](#), [4](#), [5](#), [10](#), [12](#).
ddel: [4](#), [5](#), [10](#), [11](#), [12](#).
del: [2*](#), [3*](#), [4](#), [7*](#), [8*](#), [9](#), [13](#), [34*](#).
done_here: [19](#).
donewithseg: [13](#), [14](#), [19](#).
exit: [3*](#), [4](#), [5](#), [11](#), [12](#).
fflush: [8*](#), [14](#), [20](#), [27](#).
fopen: [3*](#).
fprintf: [3*](#), [4](#), [5](#), [11](#), [12](#), [14](#), [20](#), [21](#), [27](#).
fread: [4](#).
fwrite: [33*](#), [34*](#).
goal: [27](#), [28](#).
got_it: [14](#).
i: [22](#).
ii: [2*](#), [8*](#).
infile: [2*](#), [3*](#), [4](#).
is_odd: [23](#).
isprime: [13](#), [14](#), [19](#), [27](#).
j: [2*](#), [22](#), [23](#).
jj: [2*](#), [5](#), [11](#), [12](#).
k: [2*](#), [27](#).
kk: [2*](#), [5](#), [6](#), [10](#), [12](#).
KK: [22](#), [23](#), [25](#).
kmax: [2*](#), [4](#).
ksmall: [2*](#), [4](#).
lastprime: [2*](#), [13](#), [14](#), [16](#), [19](#), [20](#), [21](#), [34*](#).
lgz: [27](#), [28](#).
LL: [22](#), [23](#).
lsize: [2*](#), [5](#), [11](#), [12](#).
m: [27](#).
magic: [32](#).
main: [2*](#).
MM: [22](#), [23](#).
mod_diff: [22](#), [23](#).
mult: [26](#), [30](#), [31](#).
n: [22](#).
nstart: [11](#).
nu: [2*](#), [17](#), [18](#).
outfile: [2*](#), [3*](#), [14](#), [20](#), [21](#), [27](#), [33*](#), [34*](#).
plist: [2*](#), [5](#), [11](#), [12](#).
prime: [2*](#), [4](#), [5](#), [6](#), [9](#), [10](#), [12](#), [33*](#).
printf: [2*](#), [3*](#), [4](#), [8*](#).
P1: [27](#).
P2: [27](#).
q: [27](#).
qq: [2*](#), [8*](#), [11](#), [12](#).
qsize: [2*](#), [5](#), [8*](#), [11](#), [12](#).
QUALITY: [23](#), [25](#).
ran_arr_buf: [23](#), [25](#).
ran_arr_cycle: [25](#).
ran_arr_dummy: [23](#), [25](#).
ran_arr_next: [25](#), [27](#).
ran_arr_ptr: [23](#), [25](#).
ran_arr_started: [23](#).
ran_array: [22](#), [23](#), [25](#).
ran_start: [23](#), [24](#), [25](#).
ran_x: [22](#), [23](#).
rem: [2*](#), [6](#), [9](#).
rep: [27](#).
s: [2*](#).
seed: [23](#).
sieve: [1](#), [2*](#), [7*](#), [9](#), [10](#), [11](#), [12](#), [13](#), [15](#), [16](#), [19](#), [34*](#).
slist: [2*](#), [5](#), [11](#), [12](#).
ss: [2*](#), [7*](#), [8*](#), [12](#), [13](#), [14](#), [19](#), [23](#).
sscanf: [3*](#).
st: [2*](#), [3*](#), [4](#), [21](#).
start: [2*](#), [5](#), [6](#), [10](#), [12](#).
starttime: [2*](#).
stderr: [3*](#), [4](#), [5](#), [11](#), [12](#).
stdout: [8*](#).
sv: [2*](#), [6](#), [9](#).
s0: [2*](#), [3*](#), [5](#), [7*](#), [14](#), [33*](#).
t: [23](#), [26](#), [32](#).
time: [2*](#), [8*](#).
timer: [2*](#), [8*](#).
tt: [2*](#), [3*](#).
TT: [23](#).
u: [32](#).
v: [32](#).
x: [2*](#), [23](#), [26](#), [27](#).
xhi: [26](#).
xlo: [26](#).
xx: [2*](#), [19](#).
y: [2*](#), [26](#), [27](#).
yhi: [26](#).
ylo: [26](#).
z: [2*](#), [27](#).
zloop: [19](#).

zp : [27](#), [29](#), [30](#), [31](#).

⟨ Change y to its binary logarithm 17 ⟩ Used in sections 15, 16, and 34*.
 ⟨ Do segment ii 8* ⟩ Used in section 2*.
 ⟨ Get ready for Montgomery's method 28 ⟩ Used in section 27.
 ⟨ Get ready for the first segment 7* ⟩ Used in section 2*.
 ⟨ If z is divisible by a prime ≤ 53 , **return** 0 32 ⟩ Used in section 27.
 ⟨ Initialize the active primes 5 ⟩ Used in section 7*.
 ⟨ Initialize the bit-counting table 18 ⟩ Used in section 2*.
 ⟨ Initialize the random number generator 24 ⟩ Used in section 2*.
 ⟨ Initialize the sieve from the tiny primes 9 ⟩ Used in section 8*.
 ⟨ Initialize the tiny active primes 6 ⟩ Used in section 5.
 ⟨ Input the primes 4 ⟩ Used in section 3*.
 ⟨ Look for and report any large gaps between $lastprime$ and x 19 ⟩ Used in section 13.
 ⟨ Look for large gaps 13 ⟩
 ⟨ Output the primes in the current segment 34* ⟩ Used in section 8*.
 ⟨ Output the primes that precede the first segment 33* ⟩ Used in section 7*.
 ⟨ Process the command line and input the primes 3* ⟩ Used in section 2*.
 ⟨ Report a gap, if it's big enough 20 ⟩ Used in section 19.
 ⟨ Report the final prime 21 ⟩
 ⟨ Set $y \leftarrow (xy/2^{64}) \bmod z$ 31 ⟩ Used in section 27.
 ⟨ Set $y \leftarrow (y^2/2^{64}) \bmod z$ 30 ⟩ Used in section 27.
 ⟨ Set $lastprime$ to the largest prime in $sieve[k]$ 16 ⟩ Used in section 13.
 ⟨ Set $lastprime$ to the smallest prime $\geq s_0$ 14 ⟩ Used in section 13.
 ⟨ Set x to the smallest prime in $sieve[j]$ 15 ⟩ Used in section 13.
 ⟨ Set zp to the inverse of z modulo 2^{64} 29 ⟩ Used in section 28.
 ⟨ Sieve in the enqueued large primes 11 ⟩ Used in section 10.
 ⟨ Sieve in the newly active primes 12 ⟩ Used in section 8*.
 ⟨ Sieve in the previously active primes 10 ⟩ Used in section 8*.
 ⟨ Subroutines 22, 23, 25, 26, 27 ⟩ Used in section 2*.

PRIME-SIEVE-SPARSE-BOOT

| | Section | Page |
|---------------------------------------|--------------------|------|
| Intro | 1 | 1 |
| Sieving | 5 | 5 |
| Processing gaps | 13 | 8 |
| Random numbers | 22 | 11 |
| Double precision multiplication | 26 | 13 |
| Prime testing | 27 | 14 |
| Index | 35 | 17 |