1. Lattice siever for the number field sieve.

format *mpz_t* *int*
format *u32_t* *int*
format *pr32_struct* *int*

2. Copying.

3.

```
#include <stdio.h>
#include <sys/types.h>
#ifdef SI_MALLOC_DEBUG
#include <fcntl.h>
#include <sys/mman.h>
#endif
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>
#include <time.h>
#ifdef LINUX
#include <endian.h>
#endif
#include <gmp.h>
#include <signal.h>
#include <setjmp.h>
#include "asm/siever-config.h"
#ifndef TDS_MPQS
#define TDS_MPQS TDS_SPECIAL_Q
#endif
#ifndef TDS_PRIMALITY_TEST
#define TDS_PRIMALITY_TEST TDS_IMMEDIATELY
#endif
#ifndef FB_RAS
#define FB_RAS 0
#endif
```

4.

```
#include "if.h"
#include "primgen32.h"
#include "asm/32bit.h"
#include "asm/64bit.h"
#include "redu2.h"
#include "recurrence6.h"
#include "fbgen.h"
#include "real-poly-aux.h"
#include "gmp-aux.h"
#include "lasieve-prepn.h"
```

5. These are the possible values for TDS_PRIMALITY_TEST and TDS_MPQS, which control when the primality tests and mpqs for trial division survivors are done.

```
#define TDS_IMMEDIATELY 0
#define TDS_BIGSS 1
#define TDS_ODDNESS_CLASS 2
#define TDS_SPECIAL_Q 3
```

6.

```
#define GCD_SIEVE_BOUND 10
#include "asm/siever-config.c"
#include "asm/lasched.h"
#include "asm/medsched.h"
#define L1_SIZE (1UL << L1_BITS)
#if 0
#define ZSS_STAT
  u32_t nss = 0, nzss[3] = {0,0,0};
#endif
  static float FB_bound[2], sieve_report_multiplier[2];
  static u16_t sieve_min[2], max_primebits[2], max_factorbits[2];
  static u32_t *(FB[2]), *(proots[2]), FBsize[2];   /* Some additional information (which can be
      considered to be the part of the factor base located at the infinite prime). */
  ⟨ Declarations for the archimedean primes 51 ⟩
  static u64_t first_spq, first_spq1, first_root, last_spq, sieve_count;
  static u32_t spq_count;
  static mpz_t m, N, aux1, aux2, aux3, sr_a, sr_b;   /* The polynomial. */
  static mpz_t *(poly[2]);
    /* Its floating point version, and some guess of how large its value on the sieving region could be. */
  double *(poly_f[2]), poly_norm[2];   /* Its degree. */
  i32_t poldeg[2], poldeg_max;   /* Should we save the factorbase after it is created? */
  u32_t keep_factorbase;
#define MAX_LPFACTORS 3
  static mpz_t rational_rest, algebraic_rest;
  mpz_t factors[MAX_LPFACTORS];
  static u32_t yield = 0, n_mpqsfail[2] = {0,0}, n_mpqsvain[2] = {0,0};
  static i64_t mpqs_clock = 0;
  static i64_t sieve_clock = 0, sch_clock = 0, td_clock = 0, tdi_clock = 0;
  static i64_t cs_clock[2] = {0,0}, Schedule_clock = 0, medsched_clock = 0;
  static i64_t si_clock[2] = {0,0}, s1_clock[2] = {0,0};
  static i64_t s2_clock[2] = {0,0}, s3_clock[2] = {0,0};
  static i64_t tdsi_clock[2] = {0,0}, tds1_clock[2] = {0,0}, tds2_clock[2] = {0,0};
  static i64_t tds3_clock[2] = {0,0}, tds4_clock[2] = {0,0};
  static u32_t n_abort1 = 0, n_abort2 = 0;
  char *basename;
  char *input_line = Λ;
  size_t input_line_alloc = 0;
```

7. This array stores the candidates for sieve reports.

```
  static u32_t ncand;
  static u16_t *cand;
  static unsigned char *fss_sv, *fss_sv2;
```

8. It will also be necessary to sort them.

```
  static int tdcand_cmp(const void *x, const void *y) { return (int)( * ( ( u16_t * ) x ) ) − (int)( * (
      ( u16_t * ) y ) ) ; }
```

9. For sieving with prime powers, we have two extra factor bases. Intuitively, the meaning is the following: The numbers q and qq are powers of the same prime, and the sieving event occurs iff qq divides the second coordinate j and if $i \% q \equiv (r * j / qq) \% q$. The sieve value is l.

More precisely, it is necessary that the sieving event occurs if and only if $(qq * i) \% pp \equiv (r * j) \% pp$ with $pp \equiv q * qq$ and $gcd(r, qq) \equiv 1$. This makes it necessary to put $r \equiv 1$ if $q \equiv 1$.

```
typedef struct xFBstruct {
  u32_t p, pp, q, qq, r, l;
} *xFBptr;
static volatile xFBptr xFB[2];
static volatile u32_t xFBs[2];
```

10. For lattice sieving, these are transformed from (a,b)-coordinates to (i,j)-coordinates. The translation function also accesses to the static variables holding the reduced sublattice base. The function also calculates the residue class of the first sieving event in each of the three sublattices, as well as the first j for which a sieving event occurs in each of the three cases.

Using the transformed factor base structure $*rop$, the next sieving event can be calculated by adding *rop→qq* to the current value of the sublattice coordinate j. The sieving events on this new j-line occur in the residue class modulo *rop→q* of $r +$ (*rop→r*), where r is the i-coordinate of an arbitrary sieving event on the current j-line. Since only this property of $*rop$ will be used, it is no longer necessary to bother about the value of *rop→r* in the case *rop→q* $\equiv 1$.

In the case of an even prime power, this means that the transformation of *op* into *rop* also involves a lowering of the index *op→pp* of the sublattice. Otherwise, $*rop$ is just the image of $*op$ in the reduced lattice coordinates.

```
static void xFBtranslate(u16_t * rop, xFBptr op);
static int xFBcmp(const void *, const void *);
```

11. The following function is used for building the extended factor base on the algebraic side. It investigates $s = *xaFB[xaFBs - 1]$ and determines the largest power l of $s.p$ satisfying $l < pp\_bound$ and dividing the value of A at all coprime pairs of integers (a, b) for which the image of (a, b) in $\mathbf{P}^1(\mathbf{Z})$ specializes to the element of $\mathbf{P}^1(\mathbf{Z}/q\mathbf{Z})$ determined by s, where q is the value of $s.pp$. In addition, elements are added to the factor base which determine the locations inside the residue class determined by s for which the value of the polynomial is divisible by a higher power of p. The value of l is placed in $s.l$.

12.

```
static u32_t add_primepowers2xaFB(size_t *aFB_alloc_ptr, u32_t pp_bound, u32_t side, u32_t
    p, u32_t r);
```

13.

```
static u64_t nextq64(u64_t lb);
```

14. The reduced basis of the sublattice consisting of all (a,b)-pairs which are divisible by the special q is (*a0*, *b0*), (*a1*, *b1*). The lattice reduction is done with respect to the scalar product

$$(a,b)\cdot(a',b') = aa' + sigma\,bb'$$

. It is assumed that the first basis vector is not longer than the second with respect to this scalar product. The sieving region is over $-2^{a-1} \le i < 2^{a-1}$ and $0 \le j \le 2^b$, where i and j are the coefficient of the first and the second vector of the reduced basis. We store a in *I_bits*, b in *J_bits*, 2^{a-1} in *i_shift*, 2^a in *n_I* and 2^b in *n_J*.

It is also necessary to make *root_no* a static variable which *trial_divide* can use if the special q is on the algebraic side.

```
  i32_t a0, a1, b0, b1;
#if 0
  u32_t I_bits;
#endif
  u32_t J_bits, i_shift, n_I, n_J;
  u32_t root_no;
  float sigma;
#include "strategy.h"
  strat_t strat;
```

15. In this version of the lattice siever, we split the sieving region into three pieces corresponding to the three non-vanishing elements of $\mathbf{F}_2^2$. The first contains all sieving events with i odd and j even, the second those with i even and j odd, the third those for which i and j are both odd. This oddness type is stored in a global variable *oddness_type* which assumes the three values 1, 2 and 3.

Since the oddness type of both lattice coordinates is fixed in each of the three subsieves, the sieving range for the subsieves is given by $n\_i = n\_I/2$ and $n\_j = n\_J/2$.

```
  static u32_t oddness_type;
  static u32_t n_i, n_j, i_bits, j_bits;
```

16.

```
⟨ Global declarations 20 ⟩
⟨ Trial division declarations 121 ⟩
void Usage()
{
  complain("Usage");
}
static u32_t n_prereports = 0, n_reports = 0, n_rep1 = 0, n_rep2 = 0;
static u32_t n_tdsurvivors[2] = {0, 0}, n_psp = 0, n_cof = 0;
static FILE *ofile;
static char *ofile_name;
#ifdef STC_DEBUG
  FILE *debugfile;
#endif
static u16_t special_q_side, first_td_side, first_sieve_side;
static u16_t first_psp_side, first_mpqs_side, append_output, exitval;
static u16_t cmdline_first_sieve_side = USHRT_MAX;
static u16_t cmdline_first_td_side = USHRT_MAX;
#define ALGEBRAIC_SIDE 0
#define RATIONAL_SIDE 1
#define NO_SIDE 2
static pr32_struct special_q_ps;
u64_t special_q;
double special_q_log;
volatile u64_t modulo64;
#define USER_INTERRUPT 1
#define SCHED_PATHOLOGY 2
#define USER_INTERRUPT_EXITVAL 2
#define LOADTEST_EXITVAL 3
jmp_buf termination_jb;
static void terminate_sieving(int signo)
{
  exitval = USER_INTERRUPT_EXITVAL;
  longjmp(termination_jb, USER_INTERRUPT);
}
static clock_t last_clock;
#ifdef MMX_TDBENCH
  extern u64_t MMX_TdNloop;
#endif
main(int argc, char **argv)
{
  u16_t zip_output, force_aFBcalc;
  u16_t catch_signals;
  u32_t all_spq_done;
  u32_t n_spq, n_spq_discard;
  char *sysload_cmd;
  u32_t process_no;
#ifdef STC_DEBUG
  debugfile = fopen("rtdsdebug", "w");
#endif
```

```
⟨ Getopt 23 ⟩
siever_init( );
⟨ Open the output file 25 ⟩
⟨ Generate factor bases 26 ⟩
⟨ Rearrange factor bases 32 ⟩
if (sieve_count ≡ 0) exit(0);
⟨ Prepare the factor base logarithms 36 ⟩
⟨ Prepare the lattice sieve scheduling 42 ⟩
⟨ TD Init 122 ⟩
read_strategy(&strat, max_factorbits, basename, max_primebits);
all_spq_done = 1;
⟨ Do the lattice sieving between first_spq and last_spq 17 ⟩
if (sieve_count ≠ 0) {
  if (zip_output ≠ 0) pclose(ofile);
  else fclose(ofile);
}
logbook(0, "%u␣Special␣q,␣%u␣reduction␣iterations\n", n_spq, n_iter);
if (n_spq_discard > 0) logbook(0, "%u␣Special␣q␣discarded\n", n_spq_discard);
⟨ Diagnostic output for four large primes version 22 ⟩
if (special_q ≥ last_spq ∧ all_spq_done ≠ 0) exit(0);
if (exitval ≡ 0) exitval = 1;
exit(exitval);
}
```

17.

⟨ Do the lattice sieving between *first_spq* and *last_spq* 17 ⟩ ≡

```
{
  u64_t *r;      /* The prime ideals above this special q number. */
  initprime32(&special_q_ps);
  last_clock = clock();
  n_spq = 0;
  n_spq_discard = 0;
  r = xmalloc(poldeg_max * sizeof (*r));
  if (last_spq >> 32) special_q = nextq64(first_spq1);
  else special_q = (u64_t)pr32_seek(&special_q_ps, (u32_t) first_spq1);
  if (catch_signals != 0) {
    signal(SIGTERM, terminate_sieving);
    signal(SIGINT, terminate_sieving);
  }
  for ( ; special_q < last_spq && special_q != 0; special_q = (last_spq >> 32 ? nextq64(special_q + 1) :
        nextprime32(&special_q_ps)), first_root = 0) {
    u32_t nr;
    special_q_log = log(special_q);
    if (cmdline_first_sieve_side == USHRT_MAX) {
#if 1
      double nn[2];
      u32_t s;
      for (s = 0; s < 2; s++) {
        nn[s] = log(poly_norm[s] * (special_q_side == s ? 1 : special_q));
        nn[s] = nn[s]/(sieve_report_multiplier[s] * log(FB_bound[s]));
      }
      if (nn[0] < nn[1]) first_sieve_side = 1;
      else first_sieve_side = 0;
#else
      if (poly_norm[0] * (special_q_side == 0 ? 1 : special_q) < poly_norm[1] * (special_q_side == 1 ? 1 :
            special_q)) {
        first_sieve_side = 1;
      }
      else {
        first_sieve_side = 0;
      }
#endif
    }
    else {
      first_sieve_side = cmdline_first_sieve_side;
      if (first_sieve_side >= 2)
        complain("First sieve side must not be %u\n", (u32_t) first_sieve_side);
    }
    logbook(1, "First sieve side: %u\n", (u32_t) first_sieve_side);
    if (cmdline_first_td_side != USHRT_MAX) first_td_side = cmdline_first_td_side;
    else first_td_side = first_sieve_side;
#if 0
    if (poldeg[special_q_side] > 1) {
      nr = root_finder(r, poly[special_q_side], poldeg[special_q_side], special_q);
      if (nr == 0) continue;
```

```
      if (r[nr − 1] ≡ special_q) {
          /* Dont bother about special q roots at infinity in the projective space. */
        nr−−;
      }
    }
    else {
      u32_t x = mpz_fdiv_ui(poly[special_q_side][1], (unsigned long int) special_q);
      if (x ≡ 0) {
        n_spq_discard++;
        continue;
      }
      modulo32 = special_q;
      x = modmul32(modinv32(x), mpz_fdiv_ui(poly[special_q_side][0], (unsigned long int) special_q));
      r[0] = x ≡ 0 ? 0 : special_q − x;
      nr = 1;
    }
#endif
    nr = root_finder64(r, poly[special_q_side], poldeg[special_q_side], special_q);
    if (nr ≡ 0) continue;
    if (r[nr − 1] ≡ special_q) {
        /* Dont bother about special q roots at infinity in the projective space. */
      nr−−;
    }
    for (root_no = 0; root_no < nr; root_no++) {
      u32_t termination_condition;
      if (r[root_no] < first_root) continue;
      if ((termination_condition = setjmp(termination_jb)) ≠ 0) {
        if (termination_condition ≡ USER_INTERRUPT) ⟨Save this special q and finish 19⟩
        else {    /* termination_condition ≡ SCHED_PATHOLOGY */
          char *cmd;
          asprintf(&cmd, "touch␣badsched.%s.%u.%llu.%llu", basename, special_q_side, special_q,
              r[root_no]);
          system(cmd);
          free(cmd);
          continue;    /* Next root_no. */
        }
      }
      if (sysload_cmd ≠ Λ) {    /* Abort if the system load is too large. */
        if (system(sysload_cmd) ≠ 0) {
          exitval = LOADTEST_EXITVAL;
          longjmp(termination_jb, USER_INTERRUPT);
        }
      }
      if (reduce2(&a0, &b0, &a1, &b1, (i64_t)special_q, 0, (i64_t)r[root_no], 1, (double)(sigma * sigma)))
        {
        n_spq_discard++;
        continue;
      }
      n_spq++;
      ⟨Calculate spq_i and spq_j 21⟩
      fprintf(ofile, "#␣Start␣%llu␣%llu␣(%d,%d)␣(%d,%d)\n", special_q, r[root_no], a0, b0, a1, b1);
```

```
      ⟨ Do the sieving and td 48 ⟩
      fprintf(ofile, "#␣Done␣%llu␣%llu␣(%d,%d)␣(%d,%d)\n", special_q, r[root_no], a0, b0, a1, b1);
      if (n_spq ≥ spq_count) break;
    }
    if (root_no < nr) {    /* The program did break out of the for-loop over root_no, probably
          because it received a SIGTERM or because the loadtest failed. */
      break;    /* Out of the loop over special_q. */
    }
    if (n_spq ≥ spq_count) break;
  }
  free(r);
}
```

This code is used in section 16.

18.

```
static u64_t nextq64(u64_t lb)
{
  u64_t q, r;
  if (lb < 10) {
    if (lb < 2) return 2;
    if (lb < 3) return 3;
    if (lb < 5) return 5;
    if (lb < 7) return 7;
    return 11;
  }
  q = lb + 1 − (lb & 1);
  r = q % 3;
  if (¬r) {
    q += 2;
    r = 2;
  }
  if (r ≡ 1) r = 4;
  while (1) {
    mpz_set_ull(aux3, q);
    if (psp(aux3) ≡ 1) break;
    q += r;
    r = 6 − r;
  }
  return q;
}
```

19.

⟨ Save this special q and finish 19 ⟩ ≡

```
{
  char *hn, *ofn;
  FILE *of;
  hn = xmalloc(100);
  if (gethostname(hn, 99) ≡ 0) asprintf(&ofn, "%s.%s.last_spq%d", basename, hn, process_no);
  else asprintf(&ofn, "%s.unknown_host.last_spq%d", basename, process_no);
  free(hn);
  if ((of = fopen(ofn, "w")) ≠ 0) {
    fprintf(of, "%llu\n", special_q);
    fclose(of);
  }
  free(ofn);
  all_spq_done = 0;
  break;
}
```

This code is used in section 17.

20.

⟨ Global declarations 20 ⟩ ≡

```
u64_t spq_i, spq_j, spq_x;
```

See also sections 30, 31, 34, 35, 38, 39, 40, 41, 45, 47, 55, 58, 59, 60, 61, 62, 63, 64, 102, 110, 117, 131, 148, 150, 158, and 161.

This code is used in section 16.

21. The purpose of these numbers is the following: For the number field sieve, it is necessary not to consider (a,b)-pairs which are not coprime. Therefore, before an element (i,j) of the special-q lattice Γ is considered for trial division, we check that these numbers are coprime. Unfortunately, this does not exclude the case that both a and b are divisible by the special q. The sublattice $\Gamma' \subset \Gamma$ of all (i,j)-pairs for which this happens has index q in the special-q lattice. What we need to test membership in Γ' is a pair (*spq_i*, *spq_j*) of **u32_t** integers whose image in $\Gamma/q\Gamma$ is not zero and orthogonal (with respect to the standard scalar product) to $\Gamma'/q\Gamma$.

Since we will not work with i directly but with i + *i_shift*, it also useful to store *spq_x*, the product of *i_shift* and *spq_i* modulo q.

⟨ Calculate *spq_i* and *spq_j* 21 ⟩ ≡

```
{
  if (((i64_t) b0) % ((i64_t) special_q) ≡ 0 ∧ ((i64_t) b1) % ((i64_t) special_q) ≡ 0) {
    i64_t x;
    x = ((i64_t) a0) % ((i64_t) special_q);
    if (x < 0) x += (i64_t) special_q;
    spq_i = (u64_t) x;
    x = ((i64_t) a1) % ((i64_t) special_q);
    if (x < 0) x += (i64_t) special_q;
    spq_j = (u64_t) x;
  }
  else {
    i64_t x;
    x = ((i64_t) b0) % ((i64_t) special_q);
    if (x < 0) x += (i64_t) special_q;
    spq_i = (u64_t) x;
    x = ((i64_t) b1) % ((i64_t) special_q);
    if (x < 0) x += (i64_t) special_q;
    spq_j = (u64_t) x;
  }
  modulo64 = special_q;
  spq_x = modmul64(spq_i, (u64_t) i_shift);
}
```

This code is used in section 17.

22.

⟨ Diagnostic output for four large primes version 22 ⟩ ≡

```
  {
    u32_t side;
    logbook(0, "reports:␣%u->%u->%u->%u->%u->%u->%u␣(%u)\n", n_prereports, n_reports, n_rep1, n_rep2,
        n_tdsurvivors[first_td_side], n_tdsurvivors[1 − first_td_side], n_cof, n_psp);
      /* logbook(0,"Number of relations with k rational and l algebraic primes for (k,l)=:"); */
    sieve_clock = rint((1000.0 * sieve_clock)/CLOCKS_PER_SEC);
    sch_clock = rint((1000.0 * sch_clock)/CLOCKS_PER_SEC);
    td_clock = rint((1000.0 * td_clock)/CLOCKS_PER_SEC);
    tdi_clock = rint((1000.0 * tdi_clock)/CLOCKS_PER_SEC);
    Schedule_clock = rint((1000.0 * Schedule_clock)/CLOCKS_PER_SEC);
    medsched_clock = rint((1000.0 * medsched_clock)/CLOCKS_PER_SEC);
    mpqs_clock = rint((1000.0 * mpqs_clock)/CLOCKS_PER_SEC);
    for (side = 0; side < 2; side++) {
      cs_clock[side] = rint((1000.0 * cs_clock[side])/CLOCKS_PER_SEC);
      si_clock[side] = rint((1000.0 * si_clock[side])/CLOCKS_PER_SEC);
      s1_clock[side] = rint((1000.0 * s1_clock[side])/CLOCKS_PER_SEC);
      s2_clock[side] = rint((1000.0 * s2_clock[side])/CLOCKS_PER_SEC);
      s3_clock[side] = rint((1000.0 * s3_clock[side])/CLOCKS_PER_SEC);
      tdsi_clock[side] = rint((1000.0 * tdsi_clock[side])/CLOCKS_PER_SEC);
      tds1_clock[side] = rint((1000.0 * tds1_clock[side])/CLOCKS_PER_SEC);
      tds2_clock[side] = rint((1000.0 * tds2_clock[side])/CLOCKS_PER_SEC);
      tds3_clock[side] = rint((1000.0 * tds3_clock[side])/CLOCKS_PER_SEC);
      tds4_clock[side] = rint((1000.0 * tds4_clock[side])/CLOCKS_PER_SEC);
    }
    logbook(0, "\nTotal␣yield:␣%u\n", yield);
    if (n_mpqsfail[0] ≠ 0 ∨ n_mpqsfail[1] ≠ 0 ∨ n_mpqsvain[0] ≠ 0 ∨ n_mpqsvain[1] ≠ 0) {
      logbook(0, "%u/%u␣mpqs␣failures,␣%u/%u␣vain␣mpqs\n", n_mpqsfail[0], n_mpqsfail[1],
          n_mpqsvain[0], n_mpqsvain[1]);
    }
    logbook(0, "milliseconds␣total:␣Sieve␣%d␣Sched␣%d␣medsched␣%d\n", (int) sieve_clock, (int)
        Schedule_clock, (int) medsched_clock);
    logbook(0, "TD␣%d␣(Init␣%d,␣MPQS␣%d)␣Sieve-Change␣%d\n", (int) td_clock, (int) tdi_clock, (int)
        mpqs_clock, (int) sch_clock);
    for (side = 0; side < 2; side++) {
      logbook(0, "TD␣side␣%d:␣init/small/medium/large/search:␣%d␣%d␣%d␣%d␣%d\n", (int) side, (int)
          tdsi_clock[side], (int) tds1_clock[side], (int) tds2_clock[side], (int) tds3_clock[side], (int)
          tds4_clock[side]);
      logbook(0, "sieve:␣init/small/medium/large/search:␣%d␣%d␣%d␣%d␣%d\n", (int)
          si_clock[side], (int) s1_clock[side], (int) s2_clock[side], (int) s3_clock[side], (int) cs_clock[side]);
    }
    logbook(0, "aborts:␣%u␣%u\n", n_abort1, n_abort2);
    print_strategy_stat();
#ifdef MMX_TDBENCH
    fprintf(stderr, "MMX-Loops:␣%qu\n", MMX_TdNloop);
#endif
#ifdef ZSS_STAT
    fprintf(stderr, "%u␣subsieves,␣zero:␣%u␣first␣sieve,␣%u␣second␣sieve␣%u␣first␣td\n", nss,
        nzss[0], nzss[1], nzss[2]);
#endif
  }
```

This code is used in section 16.

23.

```
⟨ Getopt 23 ⟩ ≡
  { i32_t option;
  FILE *input_data;
  u32_t i;
  ofile_name = Λ;
  zip_output = 0;
  special_q_side = NO_SIDE;
  sigma = 0;
  keep_factorbase = 0;
  basename = Λ;
  first_spq = 0;
  sieve_count = 1;
  force_aFBcalc = 0;
  sysload_cmd = Λ;
  process_no = 0;
  catch_signals = 0;
  first_psp_side = 2;
  first_mpqs_side = 0;
  J_bits = U32_MAX;
  rescale[0] = 0;
  rescale[1] = 0;
  spq_count = U32_MAX;
#define NumRead64(x) if (sscanf(optarg, "%llu", &x) ≠ 1) Usage()
#define NumRead(x) if (sscanf(optarg, "%u", &x) ≠ 1) Usage()
#define NumRead16(x) if (sscanf(optarg, "%hu", &x) ≠ 1) Usage()
  append_output = 0;
  while ((option = getopt(argc, argv, "C:FJ:L:M:N:P:R:S:ab:c:f:i:kn:o:q:rt:vz")) ≠ −1) {
    switch (option) {
    case 'C':
      if (sscanf(optarg, "%u", &spq_count) ≠ 1) Usage();
      break;
    case 'F': force_aFBcalc = 1;
      break;
    case 'J': NumRead(J_bits);
      break;
    case 'L': sysload_cmd = optarg;
      break;
    case 'M': NumRead16(first_mpqs_side);
      break;
    case 'P': NumRead16(first_psp_side);
      break;
    case 'R':
      if (sscanf(optarg, "%u:%u", rescale, rescale + 1) ≠ 2) {
        rescale[1] = 0;
        if (sscanf(optarg, "%u", rescale) ≠ 1) Usage();
      }
      break;
    case 'S':
      if (sscanf(optarg, "%f", &sigma) ≠ 1) {
        errprintf("Cannot␣read␣floating␣point␣number␣%s\n", optarg);
```

```
      Usage( );
    }
    break;
  case 'a':
    if (special_q_side ≠ NO_SIDE) {
      errprintf("Ignoring␣-a\n");
      break;
    }
    special_q_side = ALGEBRAIC_SIDE;
    break;
  case 'b':
    if (basename ≠ Λ) errprintf("Ignoring␣-b␣%s\n", basename);
    else basename = optarg;
    break;
  case 'c': NumRead64(sieve_count);
    break;
  case 'f':
    if (sscanf(optarg, "%llu:%llu:%llu", &first_spq, &first_spq1, &first_root) ≠ 3) {
      if (sscanf(optarg, "%llu", &first_spq) ≡ 1) {
        first_spq1 = first_spq;
        first_root = 0;
      }
      else Usage( );
    }
    else append_output = 1;
    break;
  case 'i':
    if (sscanf(optarg, "%hu", &cmdline_first_sieve_side) ≠ 1) complain("-i␣%s␣???\n", optarg);
    break;
  case 'k': keep_factorbase = 1;
    break;
  case 'n': catch_signals = 1;
  case 'N': NumRead(process_no);
    break;
  case 'o': ofile_name = optarg;
    break;
  case 'q': NumRead16(special_q_side);
    break;
  case 'r':
    if (special_q_side ≠ NO_SIDE) {
      errprintf("Ignoring␣-r\n");
      break;
    }
    special_q_side = RATIONAL_SIDE;
    break;
  case 't':
    if (sscanf(optarg, "%hu", &cmdline_first_td_side) ≠ 1) complain("-t␣%s␣???\n", optarg);
    break;
  case 'v': verbose++;
    break;
  case 'z': zip_output = 1;
    break;
```

```
    }
  }
  if (J_bits ≡ U32_MAX) J_bits = I_bits − 1;
  if (first_psp_side ≡ 2) first_psp_side = first_mpqs_side;
#ifndef I_bits
#error Must # define I_bits
#endif
  last_spq = first_spq + sieve_count;
#if 0
  if (last_spq ≥ I32_MAX) {    /* CAVE: Maybe this can be relaxed somewhat without invalidating our
        reduction code, but better err on the safe side. */
    complain("Cannot␣handle␣special␣q␣>=␣%d\n", I32_MAX/2);
  }
#endif
  if (optind < argc ∧ basename ≡ Λ) {
    basename = argv[optind];
    optind++;
  }
  if (optind < argc) fprintf(stderr, "Ignoring␣%u␣trailing␣command␣line␣args\n", argc − optind);
  if (basename ≡ Λ) basename = "gnfs";
  if ((input_data = fopen(basename, "r")) ≡ Λ) {
    complain("Cannot␣open␣%s␣for␣input␣of␣nfs␣polynomials:␣%m\n", basename);
  }
  mpz_init(N);
  mpz_init(m);
  mpz_init(aux1);
  mpz_init(aux2);
  mpz_init(aux3);
  mpz_init(sr_a);
  mpz_init(sr_b);
  mpz_ull_init();
  mpz_init(rational_rest);
  mpz_init(algebraic_rest);
  input_poly(N, poly, poldeg, poly + 1, poldeg + 1, m, input_data);
#if 0
  if (poldeg[1] > 1) {
    if (poldeg[0] ≡ 1) {
      mpz_t *X;

      poldeg[0] = poldeg[1];
      poldeg[1] = 1;
      X = poly[0];
      poly[0] = poly[1];
      poly[1] = X;
    }
    else {
      complain("Degrees␣>1␣on␣both␣sides␣not␣implemented\n");
    }
  }
#endif
  skip_blanks_comments(&input_line, &input_line_alloc, input_data);
  if (input_line ≡ Λ ∨ sscanf(input_line, "%hu␣%f␣%f␣%hu␣%hu\n", &(sieve_min[1]), FB_bound + 1,
        &(sieve_report_multiplier[1]), &(max_primebits[1]), &(max_factorbits[1])) ≠ 5) {
```

```
    errprintf("Rational␣sieve␣parameters␣like\n");
    errprintf("sievemin␣FBbound␣sieverest_multiplier␣max_prime_bits␣max_factorbits\n");
    errprintf("eg.␣20␣␣␣␣␣1e6␣␣␣␣3.2␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣30␣␣␣␣␣␣␣␣␣␣␣␣␣50\n");
    complain("lacking␣from␣input␣file␣%s\n", basename);
  }
  if (fscanf(input_data, "%hu␣%f␣%f␣%hu␣%hu\n", &(sieve_min[0]), FB_bound, &(sieve_report_multiplier[0]),
        &(max_primebits[0]), &(max_factorbits[0])) ≠ 5) {
    errprintf("Algebraic␣sieve␣parameters␣like\n");
    errprintf("sievemin␣FBbound␣sieverest_multiplier␣max_prime_bits␣max_factorbits\n");
    errprintf("eg.␣20␣␣␣␣␣1e6␣␣␣␣3.2␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣30␣␣␣␣␣␣␣␣␣␣␣␣␣50\n");
    complain("lacking␣from␣input␣file␣%s\n", basename);
  }
  for (i = 0; i < 2; i++) {
    if (FB_bound[i] < 4 ∨ sieve_report_multiplier[i] ≤ 0) {
      complain("Please␣set␣all␣bounds␣to␣reasonable␣values!\n");
    }
#if 0
    if (max_primebits[i] > 33) {
      complain("Only␣large␣primes␣up␣to␣33␣bits␣are␣allowed.\n");
    }
#endif
  }
  if (sieve_count ≠ 0) {
    if (sigma ≡ 0) complain("Please␣set␣a␣skewness\n");
    if (special_q_side ≡ NO_SIDE) {
      errprintf("Please␣use␣-a␣or␣-r\n");
      Usage();
    }
    if ((u64_t)(FB_bound[special_q_side]) > first_spq) {
      complain("Special␣q␣lower␣bound␣%llu␣below␣rFB␣bound␣%g\n", first_spq,
          FB_bound[special_q_side]);
    }
  }
  fclose(input_data);
  if (poldeg[0] < poldeg[1]) poldeg_max = poldeg[1];
  else poldeg_max = poldeg[0];
      /* CAVE You should better make sure that sieving is carried out only if both I_bits and J_bits are
        at least 2, although smaller values could be accepted to turn on diagnostic output. */
  i_shift = 1 ≪ (I_bits − 1);
  n_I = 1 ≪ I_bits;
  n_i = n_I/2;
  i_bits = I_bits − 1;
  n_J = 1 ≪ J_bits;
  n_j = n_J/2;
  j_bits = J_bits − 1;
  ⟨ Get floating point coefficients 24 ⟩
  }
```

This code is used in section 16.

24.

⟨ Get floating point coefficients 24 ⟩ ≡

```
{
  u32_t i, j;
  double x, y, z;
  x = sqrt(first_spq * sigma) * n_I;
  y = x/sigma;
  for (j = 0; j < 2; j++) {
    poly_f[j] = xmalloc((poldeg[j] + 1) * sizeof (*poly_f[j]));
    for (i = 0, z = 1, poly_norm[j] = 0; i <= poldeg[j]; i++) {
      poly_f[j][i] = mpz_get_d(poly[j][i]);
      poly_norm[j] = poly_norm[j] * y + fabs(poly_f[j][i]) * z;
      z *= x;
    }
  }
}
```

This code is used in section 23.

25. CAVE protect against overwriting existing files?

⟨ Open the output file 25 ⟩ ≡

```
if (sieve_count ≠ 0) {      /* Output file was given as a command line option. */
  if (ofile_name ≡ Λ) {
    if (zip_output ≡ 0) {
      asprintf(&ofile_name, "%s.lasieve-%u.%llu-%llu", basename, special_q_side, first_spq, last_spq);
    }
    else {
      asprintf(&ofile_name,
          append_output ≡ 0 ? "gzip --best --stdout > %s.lasieve-%u.%llu-%llu.gz" :
          "gzip --best --stdout >> %s.lasieve-%u.%llu-%llu.gz", basename, special_q_side,
          first_spq, last_spq);
    }
  }
  else {
    if (strcmp(ofile_name, "-") ≡ 0) {
      if (zip_output ≡ 0) {
        ofile = stdout;
        ofile_name = "to stdout";
        goto done_opening_output;
      }
      else ofile_name = "gzip --best --stdout";
    }
    else {
      if (fnmatch("*.gz", ofile_name, 0) ≡ 0) {
        char *on1;

        zip_output = 1;
        on1 = strdup(ofile_name);
        asprintf(&ofile_name, "gzip --best --stdout > %s", on1);
        free(on1);
      }
      else zip_output = 0;
    }
  }
  if (zip_output ≡ 0) {
    if (append_output > 0) {
      ofile = fopen(ofile_name, "a");
    }
    else ofile = fopen(ofile_name, "w");
    if (ofile ≡ Λ) complain("Cannot open %s for output: %m\n", ofile_name);
  }
  else {
    if ((ofile = popen(ofile_name, "w")) ≡ Λ)
      complain("Cannot exec %s for output: %m\n", ofile_name);
  }
done_opening_output: fprintf(ofile, "F 0 X %u 1\n", poldeg[0]);
}
```

This code is used in section 16.

26. For this version of the siever, we strive for cache efficiency of sieving and hence break the sieve interval into pieces of size L1_SIZE. It is then necessary to keep information about the first factor base primes on both sides which are > L1_SIZE.

```
⟨ Generate factor bases 26 ⟩ ≡
{
  size_t FBS_alloc = 4096;
  u32_t prime;
  pr32_struct ps;
  char *afbname;
  FILE *afbfile;
  u32_t side;

  initprime32(&ps);
  for (side = 0; side < 2; side++) {
    if (poldeg[side] ≡ 1) {
      u32_t j;

      FB[side] = xmalloc(FBS_alloc * sizeof(u32_t));
      proots[side] = xmalloc(FBS_alloc * sizeof(u32_t));
      prime = firstprime32(&ps);      /* Prime 2 is given special treatment. */
      for (prime = nextprime32(&ps), fbi1[side] = 0, FBsize[side] = 0; prime < FB_bound[side];
            prime = nextprime32(&ps)) {
        u32_t x;

        x = mpz_fdiv_ui(poly[side][1], prime);
        if (x > 0) {
          modulo32 = prime;
          x = modmul32(modinv32(x), mpz_fdiv_ui(poly[side][0], prime));
          x = x > 0 ? prime − x : 0;
        }
        else x = prime;
        if (prime < L1_SIZE) fbi1[side] = FBsize[side];
        if (prime < n_i) fbis[side] = FBsize[side];
        if (FBsize[side] ≡ FBS_alloc) {
          FBS_alloc *= 2;
          FB[side] = xrealloc(FB[side], FBS_alloc * sizeof(u32_t));
          proots[side] = xrealloc(proots[side], FBS_alloc * sizeof(u32_t));
        }
        proots[side][FBsize[side]] = x;
        FB[side][FBsize[side]++] = prime;
      }      /* Also, provide read-ahead safety for some functions. */
      proots[side] = xrealloc(proots[side], FBsize[side] * sizeof(u32_t));
      FB[side] = xrealloc(FB[side], FBsize[side] * sizeof(u32_t));
      fbi1[side]++;
      fbis[side]++;
      if (fbi1[side] < fbis[side]) fbi1[side] = fbis[side];
    }
    else {
      u32_t j, k, l;

      asprintf(&afbname, "%s.afb.%u", basename, side);
      if (force_aFBcalc > 0 ∨ (afbfile = fopen(afbname, "r")) ≡ Λ) {
        ⟨ Generate aFB 27 ⟩
        if (keep_factorbase > 0) ⟨ Save aFB 29 ⟩
      }
```

```
      else {
        ⟨ Read aFB 28 ⟩
      }
      for (j = 0, k = 0, l = 0; j < FBsize[side]; j++) {
        if (FB[side][j] < L1_SIZE) k = j;
        if (FB[side][j] < n_i) l = j;
        if (FB[side][j] > L1_SIZE ∧ FB[side][j] > n_I) break;
      }
      if (FBsize[side] > 0) {
        if (k < l) k = l;
        fbis[side] = l + 1;
        fbi1[side] = k + 1;
      }
      else {
        fbis[side] = 0;
        fbi1[side] = 0;
      }
    }
  }    /* CAVE clearprime */
  {
    u32_t i, srfbs, safbs;
    for (i = 0, srfbs = 0; i < xFBs[1]; i++) {
      if (xFB[1][i].p ≡ xFB[1][i].pp) srfbs++;
    }
    for (i = 0, safbs = 0; i < xFBs[0]; i++) {
      if (xFB[0][i].p ≡ xFB[0][i].pp) safbs++;
    }
    logbook(0, "FBsize␣%u+%u␣(deg␣%u),␣%u+%u␣(deg␣%u)\n", FBsize[0], safbs, poldeg[0], FBsize[1],
        srfbs, poldeg[1]);
  }    /* free(afbname); */    /* Archimedean part of the algebraic factor base. */
  ⟨ Init for the archimedean primes 52 ⟩
}
```

This code is used in section 16.

27.

```
⟨ Generate aFB 27 ⟩ ≡
  u32_t *root_buffer;
  size_t aFB_alloc;
  root_buffer = xmalloc(poldeg[side] * sizeof (*root_buffer));
  aFB_alloc = 4096;
  FB[side] = xmalloc(aFB_alloc * sizeof (**FB));
  proots[side] = xmalloc(aFB_alloc * sizeof (**proots));
  for (prime = firstprime32(&ps), FBsize[side] = 0; prime < FB_bound[side]; prime = nextprime32(&ps))
  {
    u32_t i, nr;
    nr = root_finder(root_buffer, poly[side], poldeg[side], prime);
    for (i = 0; i < nr; i++) {
      if (aFB_alloc ≤ FBsize[side]) {
        aFB_alloc *= 2;
        FB[side] = xrealloc(FB[side], aFB_alloc * sizeof (**FB));
        proots[side] = xrealloc(proots[side], aFB_alloc * sizeof (**proots));
      }
      FB[side][FBsize[side]] = prime;
      proots[side][FBsize[side]] = root_buffer[i];
      if (prime > 2) FBsize[side]++;
    }
  }
  FB[side] = xrealloc(FB[side], FBsize[side] * sizeof (**FB));
  proots[side] = xrealloc(proots[side], FBsize[side] * sizeof (**proots));
  free(root_buffer);
```

This code is used in section 26.

28.

```
⟨ Read aFB 28 ⟩ ≡
  if (read_u32(afbfile, &(FBsize[side]), 1) ≠ 1) {
    complain("Cannot␣read␣aFB␣size␣from␣%s:␣%m\n", afbname);
  }   /* Also, provide read-ahead safety for some functions. */
  FB[side] = xmalloc(FBsize[side] * sizeof(u32_t));
  proots[side] = xmalloc(FBsize[side] * sizeof(u32_t));
  if (read_u32(afbfile, FB[side], FBsize[side]) ≠ FBsize[side] ∨ read_u32(afbfile, proots[side],
        FBsize[side]) ≠ FBsize[side]) {
    complain("Cannot␣read␣aFB␣from␣%s:␣%m\n", afbname);
  }
  if (read_u32(afbfile, &xFBs[side], 1) ≠ 1) {
    complain("%s:␣Cannot␣read␣xFBsize\n", afbname);
  }
  fclose(afbfile);
```

This code is used in section 26.

29.

⟨ Save *aFB* 29 ⟩ ≡

```
{
  if ((afbfile = fopen(afbname, "w")) ≡ Λ) {
    complain("Cannot␣open␣%s␣for␣output␣of␣aFB:␣%m\n", afbname);
  }
  if (write_u32(afbfile, &(FBsize[side]), 1) ≠ 1) {
    complain("Cannot␣write␣aFBsize␣to␣%s:␣%m\n", afbname);
  }
  if (write_u32(afbfile, FB[side], FBsize[side]) ≠ FBsize[side] ∨ write_u32(afbfile, proots[side],
        FBsize[side]) ≠ FBsize[side]) {
    complain("Cannot␣write␣aFB␣to␣%s:␣%m\n", afbname);
  }
  if (write_u32(afbfile, &xFBs[side], 1) ≠ 1) {
    complain("Cannot␣write␣aFBsize␣to␣%s:␣%m\n", afbname);
  }
  fclose(afbfile);
}
```

This code is used in section 26.

30. The variables which keep the information about how many factor base primes are < `L1_SIZE`.

⟨ Global declarations 20 ⟩ +≡

```
u32_t fbi1[2];
```

31. The variables which keep the information about how many factor base primes are < *n_I*.

⟨ Global declarations 20 ⟩ +≡

```
u32_t fbis[2];
```

32.

⟨ Rearrange factor bases 32 ⟩ ≡

```
  {
    i32_t side, d;
    u32_t *fbsz;
    fbsz = xmalloc((poldeg[poldeg[0] < poldeg[1] ? 1 : 0] + 1) * sizeof (*fbsz));
    for (side = 0; side < 2; side++) {
      u32_t i, p, *FB1, *pr1;
      deg_fbibounds[side] = xmalloc((poldeg[side] + 1) * sizeof (*(deg_fbibounds[side])));
      deg_fbibounds[side][0] = fbi1[side];
      bzero(fbsz, (poldeg[side] + 1) * sizeof (*fbsz));
      for (i = fbi1[side]; i < FBsize[side]; ) {
        u32_t p;
        d = 0;
        p = FB[side][i];
        do {
          i++;
          d++;
        } while (i < FBsize[side] ∧ FB[side][i] ≡ p);
#ifdef MAX_FB_PER_P
        while (d > MAX_FB_PER_P) {
          fbsz[MAX_FB_PER_P]++;
          d = d − MAX_FB_PER_P;
        }
#endif
        fbsz[d]++;
      }
      logbook(0, "Sorted␣factor␣base␣on␣side␣%d:", side);
      for (d = 1, i = fbi1[side]; d ≤ poldeg[side]; d++) {
        if (fbsz[d] > 0) logbook(0, "␣%d:␣%u", d, fbsz[d]);
        i += d * fbsz[d];
        deg_fbibounds[side][d] = i;
        fbsz[d] = deg_fbibounds[side][d − 1];
      }
      logbook(0, "\n");
      if (deg_fbibounds[side][1] ≡ deg_fbibounds[side][poldeg[side]]) {
#if FB_RAS > 0
        FB[side] = xrealloc(FB[side], (FBsize[side] + FB_RAS) * sizeof (*FB[side]));
        proots[side] = xrealloc(proots[side], (FBsize[side] + FB_RAS) * sizeof (*proots[side]));
        goto fill_in_read_ahead_safety;
#else   /* No rearrangement required. */
        continue;
#endif
      }
      FB1 = xmalloc((FBsize[side] + FB_RAS) * sizeof (*FB1));
      pr1 = xmalloc((FBsize[side] + FB_RAS) * sizeof (*pr1));
      for (i = 0; i < fbi1[side]; i++) {
        FB1[i] = FB[side][i];
        pr1[i] = proots[side][i];
      }
      for (i = fbi1[side]; i < FBsize[side]; ) {
```

```
        u32_t p, j;
        d = 0;
        p = FB[side][i];
        j = i;
        do {
          j++;
          d++;
        } while (j < FBsize[side] ∧ FB[side][j] ≡ p);
#ifdef MAX_FB_PER_P
        while (j > i + MAX_FB_PER_P) {
          u32_t k;
          k = i + MAX_FB_PER_P;
          while (i < k) {
            FB1[fbsz[MAX_FB_PER_P]] = p;
            pr1[fbsz[MAX_FB_PER_P]++] = proots[side][i++];
          }
          d = d − MAX_FB_PER_P;
          i = k;
        }
#endif
        while (i < j) {
          FB1[fbsz[d]] = p;
          pr1[fbsz[d]++] = proots[side][i++];
        }
      }
      free(FB[side]);
      free(proots[side]);
      FB[side] = FB1;
      proots[side] = pr1;
#if FB_RAS > 0
    fill_in_read_ahead_safety:
      for (i = 0; i < FB_RAS; i++) {    /* safe values */
        FB[side][FBsize[side] + i] = 65537;
        proots[side][FBsize[side] + i] = 0;
      }
#endif
    }
    free(fbsz);
  }
```

This code is used in section 16.

33. The factor base elements belonging to primes $p \geq$ `L1_SIZE` for which there are d different projective roots are `FB`$[s][fbi]$ with $deg\_fbibounds[s][d-1] \leq fbi$ and $fbi < deg\_fbibounds[s][d]$.

34.

```
⟨ Global declarations 20 ⟩ +≡
  u32_t *(deg_fbibounds[2]);
```

35. A factor base Element has sieve logarithm l iff its factor base index is $\geq$ *fbi_logbounds*[*side*][*d*][*l*] and $<$ *fbi_logbounds*[*side*][*d*][$l+1$].

⟨ Global declarations 20 ⟩ +≡
u32_t $**$(*fbi_logbounds*[2]);

36. CAVE Provisorium bei Wahl der Siebmultiplikatoren!

⟨ Prepare the factor base logarithms 36 ⟩ ≡

```
{
  u32_t side, i;
  for (side = 0; side < 2; side++) {
    u32_t prime, nr, pp_bound;
    struct xFBstruct *s;
    u32_t *root_buffer;
    size_t xaFB_alloc = 0;
    FB_logs[side] = xmalloc(fbi1[side]);
    FB_logss[side] = xmalloc(fbi1[side]);
    sieve_multiplier[side] = (UCHAR_MAX − 50)/log(poly_norm[side]);
    sieve_multiplier_small[side] = sieve_multiplier[side];
    for (i = 0; i < rescale[side]; i++) sieve_multiplier_small[side] *= 2.;
    pp_bound = (n_I < 65536 ? n_I : 65535);
    root_buffer = xmalloc(poldeg[side] * sizeof (*root_buffer));
    prime = 2;
    nr = root_finder(root_buffer, poly[side], poldeg[side], prime);
    for (i = 0; i < nr; i++) {
      adjust_bufsize((void **) &(xFB[side]), &xaFB_alloc, 1 + xFBs[side], 16, sizeof (**xFB));
      s = xFB[side] + xFBs[side];
      s→p = prime;
      s→pp = prime;
      if (root_buffer[i] ≡ prime) {
        s→qq = prime;
        s→q = 1;
        s→r = 1;
      }
      else {
        s→qq = 1;
        s→q = prime;
        s→r = root_buffer[i];
      }
      xFBs[side]++;
      add_primepowers2xaFB(&xaFB_alloc, pp_bound, side, 0, 0);
    }
    free(root_buffer);
    for (i = 0; i < fbi1[side]; i++) {
      double l;
      u32_t l1;
      prime = FB[side][i];
      if (prime > n_I/prime) break;
      l = log(prime);
      l1 = add_primepowers2xaFB(&xaFB_alloc, pp_bound, side, prime, proots[side][i]);
      FB_logs[side][i] = rint(l1 * l * sieve_multiplier[side]);
      FB_logss[side][i] = rint(l1 * l * sieve_multiplier_small[side]);
    }
    while (i < fbi1[side]) {
      double l;
      l = log(FB[side][i]);
      if (l > FB_maxlog[side]) FB_maxlog[side] = l;
```

```
      FB_logss[side][i] = rint(sieve_multiplier_small[side] * l);
      FB_logs[side][i++] = rint(sieve_multiplier[side] * l);
    }
    qsort(xFB[side], xFBs[side], sizeof (*(xFB[side])), xFBcmp);
    ⟨ Generate fbi_logbounds 37 ⟩
    FB_maxlog[side] *= sieve_multiplier[side];
  }
}
```

This code is used in section 16.

37.

```
⟨ Generate fbi_logbounds 37 ⟩ ≡
{
  u32_t l, ub;
  double ln;
  int d;
  fbi_logbounds[side] = xmalloc((poldeg[side] + 1) * sizeof (*(fbi_logbounds[side])));
  for (d = 1; d ≤ poldeg[side]; d++) {
    fbi_logbounds[side][d] = xmalloc(257 * sizeof (**(fbi_logbounds[side])));
    if (deg_fbibounds[side][d] > 0) {
      double ln;
      ln = log(FB[side][deg_fbibounds[side][d] − 1]);
      if (ln > FB_maxlog[side]) FB_maxlog[side] = ln;
    }
    ub = deg_fbibounds[side][d − 1];
    fbi_logbounds[side][d][0] = ub;
    for (l = 0, ub = deg_fbibounds[side][d − 1]; l < 256; l++) {
      u32_t p_ub;
      p_ub = ceil(exp((l + 0.5)/sieve_multiplier[side]));
      if (ub ≥ deg_fbibounds[side][d] ∨ FB[side][ub] ≥ p_ub) {
        fbi_logbounds[side][d][l + 1] = ub;
        continue;
      }
      while (ub < deg_fbibounds[side][d] ∧ FB[side][ub] < p_ub) ub += SCHEDFBI_MAXSTEP;
      while (ub > deg_fbibounds[side][d] ∨ FB[side][ub − 1] ≥ p_ub) ub−−;
      fbi_logbounds[side][d][l + 1] = ub;
    }
  }
  logbook(−1, "Side␣%u␣maxl␣%lf\n", side, FB_maxlog[side]);
}
```

This code is used in section 36.

38. The sieve schedule is a *u16_t* ∗ ∗∗*xschedule*, where x stands for r or a. There are as many schedule parts as horizontal strips of the sieving lattice that fit into the L1 cache. For each schedule part, there are as many arrays of *u16_t* values as there are logarithms of factor base primes. For each schedule part i and each factor base logarithm $xl1 + l$, $xschedule[i][l]$ stores the sieving events with factor base logarithm $xl1 + l$ which fall into the i-th subsieve strip. This done by storing the number $j\_offset * n\_i + i$, where the meaning of n_i has been explained below, i is the first coordinate of the sieving event, and *j_offset* is the offset of the j-coordinate of the sieving event from the from the beginning to the horizontal subsieve strip. The information about the number of such events is stored indirectly as $xschedule[i][l]$ for the first l for which there is no corresponding factor base element.

For the primes below *n_I*, sieving is done in a rather conventional way (strip by strip) explained (CAVE) below.

For a projective root r belonging to p, if $0 \leq r < p$ then the sieving event occurs precisely for $i \cong rj \pmod p$. If $r = p$, then the sieving event occurs if p divides j. These sieving events are not carried out explicitely but are accumulated in a short array *horizontal_sievesums*. The speedup achieved by this simplification is probably negligible but this case needs a special treatment anyway.

For all primes above *n_I*, a recurrence information as explained in the file `recurrence2.w` is calculated. If the prime is below `L1_SIZE`, then this information is touched once for each subsieve strip. If it is larger, then this information is used at the beginning of sieving, when these sieving events are scheduled. This scheduling happens right after the recurrence information has been calculated. The recurrence information is then reused only at the end of sieving, when we perform the trial division. For the primes above *n_I* and below `L1_SIZE`, the first sieving event which may occur inside the current sieving strip is stored as two adjacent entries in *x_current_ij*. For each of the three oddness types, it is necessary to store the first sieving event. This also done by *get_recurrence_info*, and the result is stored as two short integers starting from $first\_event[side][oddness\_type - 1] + 2 * fbi$.

The recurrence information for the primes above `L1_SIZE` is stored starting from *LPri1*[*side*].

```
⟨ Global declarations 20 ⟩ +≡
  static u32_t j_per_strip, jps_bits, jps_mask, n_strips;
  static struct schedule_struct {
    u16_t ∗ ∗∗schedule;
    u32_t ∗fbi_bounds;
    u32_t n_pieces;
    unsigned char ∗schedlogs;
    u16_t n_strips, current_strip;
    size_t alloc, alloc1;
    u32_t ∗ri;
    u32_t d;    /∗ Number of factor base elements belonging to one and the same prime. ∗/
  } ∗(schedules[2]);
  u32_t n_schedules[2];
```

39.

```
⟨ Global declarations 20 ⟩ +≡
  static u32_t ∗(LPri[2]);    /∗ Recurrence information. ∗/
#define RI_SIZE 2
```

40. The array containing the first sieving events from the current strip upward.

```
⟨ Global declarations 20 ⟩ +≡
  static u32_t ∗(current_ij[2]);
```

41. Size of a schedule entry in units of *u16_t*s. This is one if the schedule is only used for sieving, two if it is used (as proposed by T. Kleinjung) to eliminate part of the trial division sieve.

```
⟨ Global declarations 20 ⟩ +≡
  static size_t sched_alloc[2];
#define SE_SIZE 2
#define SCHEDFBI_MAXSTEP #10000
```

42.

```
⟨ Prepare the lattice sieve scheduling 42 ⟩ ≡
#ifndef SI_MALLOC_DEBUG
  sieve_interval = xvalloc(L1_SIZE);
#else
  {
    int fd;
    if ((fd = open("/dev/zero", O_RDWR)) < 0) complain("xshmalloc cannot open /dev/zero: %m\n");
        /* Shared memory buffer which they use to communicate their results to parent process. */
    if ((sieve_interval = mmap(0, L1_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) ≡ (void *) −1)
      complain("xshmalloc cannot mmap: %m\n");
    close(fd);
  }
#endif
  cand = xvalloc(L1_SIZE * sizeof (*cand));
  fss_sv = xvalloc(L1_SIZE);
  fss_sv2 = xvalloc(L1_SIZE);
  tiny_sieve_buffer = xmalloc(TINY_SIEVEBUFFER_SIZE);
  if (n_i > L1_SIZE) complain("Strip length %u exceeds L1 size %u\n", n_i, L1_SIZE);
  j_per_strip = L1_SIZE/n_i;
  jps_bits = L1_BITS − i_bits;
  jps_mask = j_per_strip − 1;
  if (j_per_strip ≠ 1 ≪ jps_bits)
    Schlendrian("Expected %u j per strip, calculated %u\n", j_per_strip, 1 ≪ jps_bits);
  n_strips = n_j ≫ (L1_BITS − i_bits);
  rec_info_init(n_i, n_j);
  ⟨ Small sieve initializations 65 ⟩
  {
    u32_t s;
    for (s = 0; s < 2; s++) {
      if (sieve_min[s] < TINY_SIEVE_MIN ∧ sieve_min[s] ≠ 0) {
        errprintf("Sieving with all primes on side %u since\n", s);
        errprintf("tiny sieve procedure is being used\n");
        sieve_min[s] = 0;
      }
      current_ij[s] = xmalloc((FBsize[s] + FB_RAS) * sizeof (*current_ij[s]));
      LPri[s] = xmalloc((FBsize[s] + FB_RAS) * sizeof (**LPri) * RI_SIZE);
    }
  }
```

See also sections 43 and 103.

This code is used in section 16.

43. The reason for keeping two different sizes *allocate* and *alloc1* is that the first part of a schedule sometimes gets more sieving events than the other parts. This is due to the fact that whenever one has a prime ideal below the factor base bounds which defines a projective root in the (i,j)-coordinates, then its sieving events all go into the first part of the schedule. It is easy to prove an upper bound for their number: They all divide the value of the polynomial at (a_1, b_1), whose absolute value is bounded by the product of *poly_norm*[*side*] and the *poldeg*[*side*]-th power of the maximum of $a1/sqrt(sigma)$ and $b1 * sqrt(sigma)$. This is bounded by the product of the special q and the maximum of $sqrt(sigma)$ and $1/sqrt(sigma)$.

⟨ Prepare the lattice sieve scheduling 42 ⟩ +≡

```
  { u32_t s;
  size_t total_alloc;
  u16_t ∗sched_buf;
  double pvl_max[2];
  total_alloc = 0; for (s = 0; s < 2; s++) { u32_t i, d, nsched_per_d;
  if (sigma ≥ 1) pvl_max[s] = poldeg[s] ∗ log(last_spq ∗ sqrt(sigma));
  else pvl_max[s] = poldeg[s] ∗ log(last_spq/sqrt(sigma));
  pvl_max[s] += log(poly_norm[s]);
  if (fbi1[s] ≥ FBsize[s] ∨ i_bits + j_bits ≤ L1_BITS) {
    n_schedules[s] = 0;
    continue;
  }
  for (i = 1, d = 0; i ≤ poldeg[s]; i++)
    if (deg_fbibounds[s][i − 1] < deg_fbibounds[s][i]) d++;
  for (i = 0; i < N_PRIMEBOUNDS; i++)
    if (FB_bound[s] ≤ schedule_primebounds[i] ∨ i_bits + j_bits ≤ schedule_sizebits[i]) {
      break;
    }
  n_schedules[s] = d ∗ (i + 1);
  nsched_per_d = i + 1;
  schedules[s] = xmalloc(n_schedules[s] ∗ sizeof (∗∗schedules)); for (i = 0, d = 1; d ≤ poldeg[s]; d++) {
      u32_t j, fbi_lb;
  fbi_lb = deg_fbibounds[s][d − 1]; for (j = 0; j < N_PRIMEBOUNDS; j++) { u32_t fbp_lb, fbp_ub;
    /∗ Lower and upper bound on factor base primes. ∗/
  u32_t lb1, fbi_ub;   /∗ Factor base index bounds. ∗/
  u32_t l;   /∗ Sieve logarithm. ∗/
  u32_t sp_i;   /∗ Sieve piece index. ∗/
  u32_t n, sl_i;   /∗ Schedule log index. ∗/
  u32_t ns;   /∗ Number of strips for this schedule. ∗/
  size_t allocate, all1;
  if (fbi_lb ≥ deg_fbibounds[s][d]) break;
  if (j ≡ nsched_per_d − 1) fbp_ub = FB_bound[s];
  else fbp_ub = schedule_primebounds[j];
  if (j ≡ 0) fbp_lb = FB[s][fbi_lb];
  else fbp_lb = schedule_primebounds[j − 1];
  if (fbp_lb ≥ FB_bound[s]) continue;
  if (i_bits + j_bits < schedule_sizebits[j]) ns = 1 ≪ (i_bits + j_bits − L1_BITS);
  else ns = 1 ≪ (schedule_sizebits[j] − L1_BITS);
  schedules[s][i].n_strips = ns;   /∗ Allocate twice the amount predicted by Mertens law and the
      statistical independence of sieving events. ∗/
#ifndef SCHED_TOL
#ifndef NO_SCHEDTOL
```

```
#define SCHED_TOL 2
#endif
#endif
#ifdef SCHED_TOL
  allocate = rint(SCHED_TOL * n_i * j_per_strip * log(log(fbp_ub)/log(fbp_lb)));
#else
  allocate = rint(sched_tol[i] * n_i * j_per_strip * log(log(fbp_ub)/log(fbp_lb)));
#endif
  allocate *= SE_SIZE;    /* It is easy to convince oneself that the second summand is large enough to
      deal with the problem mentioned at the beginning of this module. */
  all1 = allocate + n_i * ceil(pvl_max[s]/log(fbp_lb)) * SE_SIZE;
  schedules[s][i].alloc = allocate;
  schedules[s][i].alloc1 = all1;    /* Determine number of schedule fbi bounds. */
  n = 0;
  lb1 = fbi_lb;
  for (l = 0, n = 0; l < 256; l++) {
    u32_t ub;
    ub = fbi_logbounds[s][d][l + 1];
    fbi_ub = ub;
    while (ub > lb1 ∧ FB[s][ub − 1] ≥ fbp_ub) ub−−;
    if (ub ≤ lb1) continue;
    n += (ub + SCHEDFBI_MAXSTEP − 1 − lb1)/SCHEDFBI_MAXSTEP;
    lb1 = ub;
    if (ub ≥ deg_fbibounds[s][d] ∨ FB[s][ub] ≥ fbp_ub) break;
  }
  fbi_ub = lb1;
  schedules[s][i].n_pieces = n;
  schedules[s][i].d = d;
  n++;
  schedules[s][i].schedule = xmalloc(n * sizeof (*(schedules[s][i].schedule)));
  for (sl_i = 0; sl_i < n; sl_i++)
    schedules[s][i].schedule[sl_i] = xmalloc(ns * sizeof (**(schedules[s][i].schedule)));
  schedules[s][i].schedule[0][0] = ( u16_t * ) total_alloc;
  total_alloc += all1; for (sp_i = 1; sp_i < ns; sp_i++) { schedules[s][i].schedule[0][sp_i] = ( u16_t * )
      total_alloc;
  total_alloc += allocate; } schedules[s][i].fbi_bounds = xmalloc(n * sizeof (*(schedules[s][i].fbi_bounds)));
  schedules[s][i].schedlogs = xmalloc(n);
  n = 0;
  lb1 = fbi_lb;
  l = fbi_lb;
  for (l = 0, n = 0; l < 256; l++) {
    u32_t ub, ub1;
    ub = fbi_logbounds[s][d][l + 1];
    while (ub > lb1 ∧ FB[s][ub − 1] ≥ fbp_ub) ub−−;
    if (ub ≤ lb1) continue;
    if (ub > fbi_ub) ub = fbi_ub;
    for (ub1 = lb1; ub1 < ub; ub1 += SCHEDFBI_MAXSTEP) {
      schedules[s][i].fbi_bounds[n] = ub1;
      schedules[s][i].schedlogs[n++] = l;
    }
    lb1 = ub;
    if (ub ≥ deg_fbibounds[s][d] ∨ FB[s][ub] ≥ fbp_ub) break;
```

```
  }
  if (fbi_ub ≠ lb1) Schlendrian("Expected␣%u␣as␣fbi␣upper␣bound,␣have␣%u\n", fbi_ub, lb1);
  if (n ≠ schedules[s][i].n_pieces)
    Schlendrian("Expected␣%u␣schedule␣pieces␣on␣side␣%u,␣have␣%u\n", schedules[s][i].n_pieces, s, n);
  schedules[s][i].fbi_bounds[n++] = fbi_ub;
  schedules[s][i].ri = LPri[s] + (schedules[s][i].fbi_bounds[0] − fbis[s]) ∗ RI_SIZE;
  fbi_lb = fbi_ub;
  i++; } }
  if (i ≠ n_schedules[s])
    Schlendrian("Expected␣to␣create␣%u␣␣schedules␣on␣side␣%d,␣have␣%u\n", n_schedules[s], s, i);
  } ⟨ Allocate space for the schedule 44 ⟩
  ⟨ Prepare the medsched 46 ⟩
  }
```

44. This should be done in such a way that we will not get a core dump, even in very bizarre situations. The scheduling algorithm writes SE_SIZE *u16_t* numbers for each sieving event. This is done in steps over intervals of factor base indices given by *schedule_fbi_bounds*. The difference between adjacent factor base bounds is at most 65536, and there is at most one sieving event per factor base prime and *i*-line. Therefore, if we leave 65536 ∗ SE_SIZE ∗ *j_per_strip* headroom at the end of the schedule buffer, it is not hard to guarantee that we will never cause a core dump by writing past its end. We may, however, encounter a situation where writing to the piece of the schedule belonging to one L1-strip extended past its end, into the storage space assigned to another L1-strip (but not past the end of the schedule buffer). In this case, which should be wildly unlikely because of our selection of *allocate*, we are forced to give up this special q but may still continue work on the other special q specified on the command line.

```
⟨ Allocate space for the schedule 44 ⟩ ≡
  sched_buf = xmalloc((total_alloc + 65536 ∗ SE_SIZE ∗ j_per_strip) ∗ sizeof (∗∗∗((∗∗schedules).schedule)));
  for (s = 0; s < 2; s++) {
    u32_t i;
    for (i = 0; i < n_schedules[s]; i++) {
      u32_t sp_i;
      for (sp_i = 0; sp_i < schedules[s][i].n_strips; sp_i++)
        schedules[s][i].schedule[0][sp_i] = sched_buf + (size_t)(schedules[s][i].schedule[0][sp_i]);
    }
  }
```

This code is used in section 43.

45.

```
⟨ Global declarations 20 ⟩ +≡
#define USE_MEDSCHED
#ifdef USE_MEDSCHED
  static u16_t ∗∗(med_sched[2]);
  static u32_t ∗(medsched_fbi_bounds[2]);
  static unsigned char ∗(medsched_logs[2]);
  static size_t medsched_alloc[2];
  static u16_t n_medsched_pieces[2];
#endif
```

46.

⟨ Prepare the medsched 46 ⟩ ≡

```
#ifdef USE_MEDSCHED
  {
    u32_t s;
    for (s = 0; s < 2; s++) {
      if (fbis[s] < fbi1[s]) {
        u32_t fbi;    /* Factor base index. */
        u32_t n;
        unsigned char oldlog;    /* Allocate one sieving event per line and factor base prime. */
        medsched_alloc[s] = j_per_strip * (fbi1[s] − fbis[s]) * SE_SIZE;
          /* In addition, deal with the problem explained 'Sched:alloc'. */
        medsched_alloc[s] += n_i * ceil(pvl_max[s]/log(n_i)) * SE_SIZE;
        n_medsched_pieces[s] = 1 + FB_logs[s][fbi1[s] − 1] − FB_logs[s][fbis[s]];
        med_sched[s] = xmalloc((1 + n_medsched_pieces[s]) * sizeof (**med_sched));
        med_sched[s][0] = xmalloc(medsched_alloc[s] * sizeof (***med_sched));
        medsched_fbi_bounds[s] = xmalloc((1 + n_medsched_pieces[s]) * sizeof (**medsched_fbi_bounds));
        medsched_logs[s] = xmalloc(n_medsched_pieces[s]);
        for (n = 0, fbi = fbis[s], oldlog = UCHAR_MAX; fbi < fbi1[s]; fbi++) {
          if (FB_logs[s][fbi] ≠ oldlog) {
            medsched_fbi_bounds[s][n] = fbi;
            oldlog = FB_logs[s][fbi];
            medsched_logs[s][n++] = oldlog;
          }
        }
        if (n ≠ n_medsched_pieces[s])
          Schlendrian("Expected␣%u␣medium␣schedule␣pieces␣on␣side␣%u,␣have␣%u\n",
              n_medsched_pieces[s], s, n);
        medsched_fbi_bounds[s][n] = fbi;
      }
      else {    /* Very small factorbase. */
        n_medsched_pieces[s] = 0;
      }
    }
  }
#endif
```

This code is used in section 43.

47.

⟨ Global declarations 20 ⟩ +≡

```
  static unsigned char *sieve_interval = Λ, *(FB_logs[2]), *(FB_logss[2]);
  static unsigned char *tiny_sieve_buffer;
#define TINY_SIEVEBUFFER_SIZE 420
#define TINY_SIEVE_MIN 8
  static double sieve_multiplier[2], sieve_multiplier_small[2], FB_maxlog[2];
  static u32_t rescale[2];
  static u32_t j_offset;
```

48. Note that we dont sieve with respect to 2.

⟨ Do the sieving and td 48 ⟩ ≡

```
{
  u32_t subsieve_nr;
  ⟨ Prepare the auxilliary sieving data 49 ⟩
  for (oddness_type = 1; oddness_type < 4; oddness_type++) {
    ⟨ Prepare the medium and small primes for oddness_type. 72 ⟩
    j_offset = 0;
    ⟨ Scheduling job for the large FB primes 54 ⟩
#ifdef ZSS_STAT
    nss += n_strips;
#endif
    for (subsieve_nr = 0; subsieve_nr < n_strips; subsieve_nr++, j_offset += j_per_strip) {
      u16_t s, stepno;
#ifdef USE_MEDSCHED
      ⟨ Medsched 101 ⟩;
      {
        clock_t new_clock;

        new_clock = clock();
        medsched_clock += new_clock - last_clock;
        last_clock = new_clock;
      }
#endif
      for (s = first_sieve_side, stepno = 0; stepno < 2; stepno++, s = 1 - s) {
        clock_t new_clock, clock_diff;

        ⟨ Prepare the sieve 88 ⟩
#ifdef ZSS_STAT
        if (s ≡ 1 ∧ ncand ≡ 0) nzss[0]++;
#endif
        new_clock = clock();
        clock_diff = new_clock - last_clock;
        si_clock[s] += clock_diff;
        sieve_clock += clock_diff;
        last_clock = new_clock;
        ⟨ Sieve with the small FB primes 93 ⟩
        new_clock = clock();
        clock_diff = new_clock - last_clock;
        s1_clock[s] += clock_diff;
        sieve_clock += clock_diff;
        last_clock = new_clock;
        if (rescale[s]) {
#ifndef ASM_RESCALE
          u32_t rsi, r;

          r = (1 ≪ rescale[s]) - 1;
          for (rsi = 0; rsi < L1_SIZE; rsi++) {
            sieve_interval[rsi] += r;
            sieve_interval[rsi] ≫= rescale[s];
          }
          for (rsi = 0; rsi < j_per_strip; rsi++) {
            horizontal_sievesums[rsi] += r;
            horizontal_sievesums[rsi] ≫= rescale[s];
```

```
          }
#else
          if (rescale[s] ≡ 1) {
            rescale_interval1(sieve_interval, L1_SIZE);
            rescale_interval1(horizontal_sievesums, j_per_strip);
          }
          else if (rescale[s] ≡ 2) {
            rescale_interval2(sieve_interval, L1_SIZE);
            rescale_interval2(horizontal_sievesums, j_per_strip);
          }
          else Schlendrian("rescaling␣of␣level␣>2␣not␣implemented␣yet\n");
#endif
        }
#ifdef BADSCHED
        ncand = 0;
        continue;
#endif
        ⟨Sieve with the medium FB primes 100⟩
        new_clock = clock();
        clock_diff = new_clock − last_clock;
        s2_clock[s] += clock_diff;
        sieve_clock += clock_diff;
        last_clock = new_clock;
        ⟨Sieve with the large FB primes 104⟩
#if 0
        dumpsieve(j_offset, s);
#endif
        new_clock = clock();
        clock_diff = new_clock − last_clock;
        sieve_clock += clock_diff;
        s3_clock[s] += clock_diff;
        last_clock = new_clock;
        if (s ≡ first_sieve_side) {
#ifdef GCD_SIEVE_BOUND
          gcd_sieve();
#endif
          ⟨Candidate search 105⟩
        }
        else ⟨Final candidate search 108⟩
        new_clock = clock();
        clock_diff = new_clock − last_clock;
        sieve_clock += clock_diff;
        cs_clock[s] += clock_diff;
        last_clock = new_clock;
      }
#ifndef BADSCHED
      trial_divide();
#endif
      {
        clock_t new_clock;

        new_clock = clock();
        td_clock += new_clock − last_clock;
```

```
          last_clock = new_clock;
        }
#if TDS_MPQS ≡ TDS_BIGSS
#error "MPQS␣at␣BIGSS␣not␣yet␣for␣serial␣siever"
        output_all_tdsurvivors();
#else
#if TDS_PRIMALITY_TEST ≡ TDS_BIGSS
#error "MPQS␣at␣BIGSS␣not␣yet␣for␣serial␣siever"
        primality_tests_all();
#endif
#endif
      }
#if TDS_MPQS ≡ TDS_ODDNESS_CLASS
      output_all_tdsurvivors();
#else
#if TDS_PRIMALITY_TEST ≡ TDS_ODDNESS_CLASS
      primality_tests_all();
#endif
#endif
#if TDS_MPQS ≡ TDS_ODDNESS_CLASS ∨ TDS_PRIMALITY_TEST ≡ TDS_ODDNESS_CLASS
      {
        clock_t new_clock;

        new_clock = clock();
        td_clock += new_clock − last_clock;
        last_clock = new_clock;
      }
#endif
    }
#if TDS_MPQS ≡ TDS_SPECIAL_Q
    output_all_tdsurvivors();
#else
#if TDS_PRIMALITY_TEST ≡ TDS_SPECIAL_Q
    primality_tests_all();
#endif
#endif
#if TDS_MPQS ≡ TDS_SPECIAL_Q ∨ TDS_PRIMALITY_TEST ≡ TDS_SPECIAL_Q
    {
      clock_t new_clock;

      new_clock = clock();
      td_clock += new_clock − last_clock;
      last_clock = new_clock;
    }
#endif
  }
```

This code is used in section 17.

49. This involves the following tasks:

- For the factor base primes below *n_i*, calculate the corresponding entry of *x_roots* as explained above.
- For the factor base primes above *n_i*, calculate the recurrence information and the first sieving events with each of the three oddness types.
- For the factor base primes above `L1_SIZE`, schedule the sieving events (in addition to the previous task). For the ones between *n_i* and `L1_SIZE`, initialize *x_current_ij*.

For the first task, note that if p is the prime and r the entry in *x_proots*, then the sieving event takes place iff $a \cong rb \pmod{p}$, and for elements of the sieving sublattice this translates into $a_0 i + a_1 j \cong r(b_0 i + b_1 j) \pmod{p}$ or

$$(a_0 - rb_0)i \cong (rb_1 - a1)j \pmod{p}$$

, hence $i \cong r'j \pmod{p}$ with $r' = (rb_1 - a1)/(a_0 - rb_0)$. If the denominator is zero, we formally put $r' = p$ to indicate the infinity element of $\mathbf{P}^1(\mathbf{F}_p)$.

Of course, the calculation of r' also has to be carried out before the more complicated tasks for the larger primes start.

```
⟨ Prepare the auxilliary sieving data 49 ⟩ ≡
  { u32_t absa0, absa1, absb0, absb1;
  char a0s, a1s;
  clock_t new_clock;
#define GET_ABSSIG(abs, sig, arg)
  if (arg > 0) {
    abs = (u32_t) arg;
    sig = '+';
  }
  else {
    abs = (u32_t)(−arg);
    sig = '-';
  }
  GET_ABSSIG(absa0, a0s, a0);
  GET_ABSSIG(absa1, a1s, a1);
  absb0 = b0;
  absb1 = b1;
  ⟨ Preparation job for the small FB primes 67 ⟩
  ⟨ Preparation job for the medium and large FB primes 50 ⟩
  ⟨ Preparations at the Archimedean primes 53 ⟩
  new_clock = clock();
  sch_clock += new_clock − last_clock;
  last_clock = new_clock; }
```

This code is used in section 48.

50.

⟨ Preparation job for the medium and large FB primes 50 ⟩ ≡

```
  {
    u32_t s;
    for (s = 0; s < 2; s++) {
      i32_t d;
      lasieve_setup(FB[s] + fbis[s], proots[s] + fbis[s], fbi1[s] − fbis[s], a0, a1, b0, b1, LPri[s], 1);
#ifndef SCHEDULING_FUNCTION_CALCULATES_RI
      for (d = 1; d ≤ poldeg[s]; d++) {
        if (deg_fbibounds[s][d − 1] < deg_fbibounds[s][d]) lasieve_setup(FB[s] + deg_fbibounds[s][d − 1],
              proots[s] + deg_fbibounds[s][d − 1], deg_fbibounds[s][d] − deg_fbibounds[s][d − 1], a0, a1, b0,
              b1, LPri[s] + RI_SIZE ∗ (deg_fbibounds[s][d − 1] − fbis[s]), d);
      }
#endif
    }
  }
```

This code is used in section 49.

51. Preparations at the archimedean primes.

We translate the polynomial into the (i,j)-coordinates, such that $A(a,b) = \tilde{A}(i,j)$ if $a = a_0 * i + a_1 * j$ and $b = b_0 * i + b_1 * j$. Now, let s be $2 *$ CANDIDATE_SEARCH_STEPS, I the value of n_I, J the value of n_J. For most k, we put a lower bound to the logarithm of $\tilde{A}(x,J)$ with real x between $k * s - J$ and $(k+1) * s - J$ into *plog_lb1*[k], where k is a non-negative integer less than $2 * J/s$. For some k, there are integers i such that values of x in the sub-interval from i to $i+1$ have to be excluded. In this case, the lower bound *plog_lb1*[k] excludes all x from such exceptional subintervals. There are *n_rroots1* such exceptions (although we dont make sure that each exception belongs to a real root), and the exceptions can be found in *rroots1*.

Similarly, a lower bound for $\tilde{A}(J,y)$ with real y between $k * s - J$ and $(k+1) * s - J$ is in *plog_lb2*[k], where k is non-negative and less than $2 * J/s$. For certain i, the x between i and $i+1$ are excluded, and the list of these i is in *rroots2*.

Not the lower bounds *lb* for the logarithms themselves are stored but integer approximations to *sieve_multiplier*[0]∗ *lb*.

⟨ Declarations for the archimedean primes 51 ⟩ ≡

```
  static double ∗(tpoly_f[2]);
#define CANDIDATE_SEARCH_STEPS 128
  static unsigned char ∗∗(sieve_report_bounds[2]);
  static i32_t n_srb_i, n_srb_j;
```

This code is used in section 6.

52.

⟨ Init for the archimedean primes 52 ⟩ ≡

```
{
  u32_t i;
  size_t si, sj;
  n_srb_i = 2 * ((n_i + 2 * CANDIDATE_SEARCH_STEPS - 1)/(2 * CANDIDATE_SEARCH_STEPS));
  n_srb_j = (n_J + 2 * CANDIDATE_SEARCH_STEPS - 1)/(2 * CANDIDATE_SEARCH_STEPS);
  sj = n_srb_j * sizeof (*(sieve_report_bounds[0]));
  si = n_srb_i * sizeof (**(sieve_report_bounds[0]));
  for (i = 0; i < 2; i++) {
    u32_t j;
    tpoly_f[i] = xmalloc((1 + poldeg[i]) * sizeof (**tpoly_f));
    sieve_report_bounds[i] = xmalloc(sj);
    for (j = 0; j < n_srb_j; j++) sieve_report_bounds[i][j] = xmalloc(si);
  }
}
```

This code is used in section 26.

53.

⟨ Preparations at the Archimedean primes 53 ⟩ ≡

```
{
  u32_t i, k;
  for (i = 0; i < 2; i++) {
    double large_primes_summand;
    tpol(tpoly_f[i], poly_f[i], poldeg[i], a0, a1, b0, b1);
    large_primes_summand = sieve_report_multiplier[i] * FB_maxlog[i];
    if (i ≡ special_q_side) large_primes_summand += sieve_multiplier[i] * log(special_q);
    get_sieve_report_bounds(sieve_report_bounds[i], tpoly_f[i], poldeg[i], n_srb_i, n_srb_j,
        2 * CANDIDATE_SEARCH_STEPS, sieve_multiplier[i], large_primes_summand);
  }
}
```

This code is used in section 49.

54.

```
⟨ Scheduling job for the large FB primes 54 ⟩ ≡
#ifndef NOSCHED
  {
    u32_t s;
    clock_t new_clock;
    for (s = 0; s < 2; s++) {
      u32_t i;
      for (i = 0; i < n_schedules[s]; i++) {
        u32_t ns;    /* Number of strips for which to schedule */
        ns = schedules[s][i].n_strips;
        if (ns > n_strips) ns = n_strips;
        do_scheduling(schedules[s] + i, ns, oddness_type, s);
        schedules[s][i].current_strip = 0;
      }
    }
#ifdef GATHER_STAT
    new_clock = clock();
    Schedule_clock += new_clock - last_clock;
    last_clock = new_clock;
#endif
  }
#else    /* NOSCHED */
#define BADSCHED
#endif
```

This code is used in section 48.

55.

```
⟨ Global declarations 20 ⟩ +≡
  void do_scheduling(struct schedule_struct *, u32_t, u32_t, u32_t);
```

56.

```
#ifndef NOSCHED
  void do_scheduling(struct schedule_struct *sched, u32_t ns, u32_t ot, u32_t s){ u32_t ll, n1_j,
        *ri;
      ;
      n1_j = ns << (L1_BITS - i_bits); for (ll = 0, ri = sched->ri; ll < sched->n_pieces; ll++) { u32_t
          fbi_lb, fbi_ub, fbio; memcpy (sched->schedule[ll + 1], sched->schedule[ll], ns * sizeof ( u16_t * *
          ) ) ;
      fbio = sched->fbi_bounds[ll];
      fbi_lb = fbio;
      fbi_ub = sched->fbi_bounds[ll + 1];
#ifdef SCHEDULING_FUNCTION_CALCULATES_RI
      if (ot == 1) lasieve_setup(FB[s] + fbi_lb, proots[s] + fbi_lb, fbi_ub - fbi_lb, a0, a1, b0, b1,
            LPri[s] + (fbi_lb - fbis[s]) * RI_SIZE, sched->d);
#endif
      ri = lasched(ri, current_ij[s] + fbi_lb, current_ij[s] + fbi_ub, n1_j, (u32_t **)(sched->schedule[ll + 1]),
          fbi_lb - fbio, ot);
      ⟨ Check schedule space 57 ⟩
      } }
#endif
```

57.

```
⟨ Check schedule space 57 ⟩ ≡
  {
    u32_t k;
    for (k = 0; k < ns; k++)
      if (sched->schedule[ll + 1][k] >= sched->schedule[0][k] + sched->alloc) {
        if (k == 0 && sched->schedule[ll + 1][k] < sched->schedule[0][k] + sched->alloc1) continue;
        longjmp(termination_jb, SCHED_PATHOLOGY);
      }
  }
```

This code is used in section 56.

58. Sieving with the small primes.
This array holds information about odd factor base primes which are, in the transformed lattice coordinates depending on the special q, not located at infinity. The format of an entry e is as follows:
- $e[0]$ the prime

⟨ Global declarations 20 ⟩ +≡
static *u16_t*∗(*smallsieve_aux*[2]), ∗(*smallsieve_auxbound*[2][5]);
static *u16_t*∗(*smallsieve_tinybound*[2]);

59. This is for prime powers. In this case, there exist roots which are neiter affine nor infinity. This may happen if one has homogeneous coordinates (i, j) such that i is prime to p and j is divisible by p but not by p^2 (or some higher power of p which is under consideration).

⟨ Global declarations 20 ⟩ +≡
static *u16_t*∗(*smallsieve_aux1*[2]), ∗(*smallsieve_aux1_ub_odd*[2]);
static *u16_t*∗(*smallsieve_aux1_ub*[2]), ∗(*smallsieve_tinybound1*[2]);

60. The entries of the array *smallsieve_aux2_ub*[*side*] have the same format as above, but hold powers of two which are only used in connection with the tiny sieve buffer.

⟨ Global declarations 20 ⟩ +≡
static *u16_t*∗(*smallsieve_aux2*[2]), ∗(*smallsieve_aux2_ub*[2]);

61. This is for odd primes or prime powers for which the sieving event occurs precisely if j is divisible by p. The primes are from *smallpsieve_aux*[*side*] to < *smallpsieve_aux1*[*side*]. The prime powers start there, and are bounded by *smallpsieve_aux_ub_odd*[s]. Finally, starting from this location the array also holds powers of prime ideals of norm two defining a system of congruences of one of the following two types:
- $i \cong r * j \pmod 2$
- $j \cong 0 \pmod 2$
- $j \cong 0 \pmod 2$ and $i \cong r * (j/2) \pmod 2$. This tail of the array is bounded by *smallpsieve_aux*[s], and its contents will depend on the oddness type.

We also need a temporary buffer which is large enough to hold all odd prime powers in this array.

⟨ Global declarations 20 ⟩ +≡
static *u16_t*∗(*smallpsieve_aux*[2]), ∗(*smallpsieve_aux_ub_pow1*[2]);
static *u16_t*∗(*smallpsieve_aux_ub_odd*[2]), ∗(*smallpsieve_aux_ub*[2]);
static unsigned char ∗*horizontal_sievesums*;

62. Representation in the special q lattic coordinates of powers of prime ideals of norm two.

⟨ Global declarations 20 ⟩ +≡
static *u16_t*∗(*x2FB*[2]), *x2FBs*[2];

63. The following array is used in connection with trial division, *smalltdsieve_aux*[s][k][i] being $k+1$ times the projective roots for the i-th record in *smallpsieve_aux*[s]. If we have a special MMX function for trial division, this information is needed only for k equal to $j\_per\_strip - 1$.

⟨ Global declarations 20 ⟩ +≡
```
  static u16_t *tinysieve_curpos;
#ifndef MMX_TD
  static u16_t **(smalltdsieve_aux[2]);
#ifdef PREINVERT
  static u32_t *(smalltd_pi[2]);
#endif
#endif
```

64. One of the improvements due to T. Kleinjung is the fact that removing candidates with a small common divisor by a sieve-like procedure also provides a speedup.

```
⟨ Global declarations 20 ⟩ +≡
#ifdef GCD_SIEVE_BOUND
  static u32_t np_gcd_sieve;
  static unsigned char *gcd_sieve_buffer;
  static void gcd_sieve(void);
#endif
```

65.

```
⟨ Small sieve initializations 65 ⟩ ≡
  {
    u32_t s;
#define MAX_TINY_2POW 4
    if (poldeg[0] < poldeg[1]) s = poldeg[1];
    else s = poldeg[0];
    tinysieve_curpos = xmalloc(TINY_SIEVE_MIN * s * sizeof (*tinysieve_curpos));
    horizontal_sievesums = xmalloc(j_per_strip * sizeof (*horizontal_sievesums));
    for (s = 0; s < 2; s++) {
      u32_t fbi;
      size_t maxent;
      smallsieve_aux[s] = xmalloc(4 * fbis[s] * sizeof (*(smallsieve_aux[s])));
#ifndef MMX_TD
#ifdef PREINVERT
      smalltd_pi[s] = xmalloc(fbis[s] * sizeof (*(smalltd_pi[s])));
#endif
      smalltdsieve_aux[s] = xmalloc(j_per_strip * sizeof (*(smalltdsieve_aux[s])));
      for (fbi = 0; fbi < j_per_strip; fbi++)
        smalltdsieve_aux[s][fbi] = xmalloc(fbis[s] * sizeof (**(smalltdsieve_aux[s])));
#else    /* The MMX specific initialization procedures may be machine dependent. */
      MMX_TdAllocate(j_per_strip, fbis[0], fbis[1]);
#endif
      smallsieve_aux1[s] = xmalloc(6 * xFBs[s] * sizeof (*(smallsieve_aux1[s])));
        /* This is very unlikely, but in principle all factor base elements could define projective roots. */
      maxent = fbis[s];
      maxent += xFBs[s];
      smallpsieve_aux[s] = xmalloc(3 * maxent * sizeof (*(smallpsieve_aux[s])));
      maxent = 0;
      for (fbi = 0; fbi < xFBs[s]; fbi++) {
        if (xFB[s][fbi].p ≡ 2) maxent++;
      }
      smallsieve_aux2[s] = xmalloc(4 * maxent * sizeof (*(smallsieve_aux2[s])));
      x2FB[s] = xmalloc(maxent * 6 * sizeof (*(x2FB[s])));
    }
  }
```

See also section 66.

This code is used in section 42.

66.

```
⟨ Small sieve initializations 65 ⟩ +≡
#ifdef GCD_SIEVE_BOUND
  {
    u32_t p, i;
    firstprime32(&special_q_ps);
    np_gcd_sieve = 0;
    for (p = nextprime32(&special_q_ps); p < GCD_SIEVE_BOUND; p = nextprime32(&special_q_ps))
      np_gcd_sieve++;
    gcd_sieve_buffer = xmalloc(2 * np_gcd_sieve * sizeof (*gcd_sieve_buffer));
    firstprime32(&special_q_ps);
    i = 0;
    for (p = nextprime32(&special_q_ps); p < GCD_SIEVE_BOUND; p = nextprime32(&special_q_ps))
      gcd_sieve_buffer[2 * i++] = p;
  }
#endif
```

67.

⟨ Preparation job for the small FB primes 67 ⟩ ≡

```
{
  u32_t s;
  for (s = 0; s < 2; s++) {
    u32_t fbi;
    u16_t *abuf;   /* Affine */
    u16_t *ibuf;   /* Infinity. */
    abuf = smallsieve_aux[s];
    ibuf = smallpsieve_aux[s];
    for (fbi = 0; fbi < fbis[s]; fbi++) {
      u32_t aa, bb;
      modulo32 = FB[s][fbi];
      aa = absa0 % FB[s][fbi];
      if (a0s ≡ '-' ∧ aa ≠ 0) aa = FB[s][fbi] − aa;
      bb = absb0 % FB[s][fbi];
      if (proots[s][fbi] ≠ FB[s][fbi]) {
        u32_t x;
        x = modsub32(aa, modmul32(proots[s][fbi], bb));
        if (x ≠ 0) {
          aa = absa1 % FB[s][fbi];
          if (a1s ≡ '-' ∧ aa ≠ 0) aa = FB[s][fbi] − aa;
          bb = absb1 % FB[s][fbi];
          x = modmul32(asm_modinv32(x), modsub32(modmul32(proots[s][fbi], bb), aa));
          abuf[0] = (u16_t)(FB[s][fbi]);
          abuf[1] = (u16_t)x;
          abuf[2] = (u16_t)(FB_logss[s][fbi]);
          abuf += 4;
        }
        else {
          ibuf[0] = (u16_t)(FB[s][fbi]);
          ibuf[1] = (u16_t)(FB_logss[s][fbi]);
          ibuf += 3;
        }
      }
      else {   /* Root is projective in (ab) coordinates. */
        if (bb ≠ 0) {
          u32_t x;
          x = modulo32 − bb;
          bb = absb1 % FB[s][fbi];
          abuf[0] = (u16_t)(FB[s][fbi]);
          abuf[1] = (u16_t)(modmul32(asm_modinv32(x), bb));
          abuf[2] = (u16_t)(FB_logss[s][fbi]);
          abuf += 4;
        }
        else {
          ibuf[0] = (u16_t)(FB[s][fbi]);
          ibuf[1] = (u16_t)(FB_logss[s][fbi]);
          ibuf += 3;
        }
      }
```

```
        }
        smallsieve_auxbound[s][0] = abuf;
        smallpsieve_aux_ub_pow1[s] = ibuf;
      }
    }
```

See also sections 68, 69, and 71.
This code is used in section 49.

68.

⟨ Preparation job for the small FB primes 67 ⟩ +≡

```
  {
    u32_t s;
    for (s = 0; s < 2; s++) {
      u32_t i;
      u16_t *buf;   /* odd. */
      u16_t *buf2;   /* even. */
      u16_t *ibuf;   /* odd, infinity, prime */
      buf = smallsieve_aux1[s];
      buf2 = x2FB[s];
      ibuf = smallpsieve_aux_ub_pow1[s];
      for (i = 0; i < xFBs[s]; i++) {
        if (xFB[s][i].p ≡ 2) {
          xFBtranslate(buf2, xFB[s] + i);
          buf2 += 4;
        }
        else {
          xFBtranslate(buf, xFB[s] + i);
          if (buf[0] ≡ 1) {
            ibuf[1] = xFB[s][i].l;
            ibuf[0] = xFB[s][i].pp;
            ibuf += 3;
          }
          else buf += 6;
        }
      }
      x2FBs[s] = (buf2 − x2FB[s])/4;
      smallpsieve_aux_ub_odd[s] = ibuf;
      smallsieve_aux1_ub_odd[s] = buf;
    }
  }
```

69.

⟨ Preparation job for the small FB primes 67 ⟩ +≡

```
  {
    u32_t s;
#ifndef MMX_TD
    for (s = 0; s < 2; s++) {
      u32_t i;
      u16_t *x;
      for (i = 0, x = smallsieve_aux[s]; x < smallsieve_auxbound[s][0]; i++, x += 4) {
        u32_t k, r, pr;
        modulo32 = *x;
        r = x[1];
        pr = r;
        for (k = 0; k < j_per_strip; k++) {
          smalltdsieve_aux[s][k][i] = r;
          r = modadd32(r, pr);
        }
#ifdef PREINVERT
        ⟨ Preinvert modulo32 70 ⟩
#endif
      }
    }
#endif
  }
```

70. Determine an inverse of p modoulo $1 + \texttt{U32\_MAX}$. Note that p is inverse to itself modulo 8. A Hensel step squares the precision of the inverse. Four Hensel steps are sufficient unless CAVE the size of **u32_t** is 64 bits.

⟨ Preinvert modulo32 70 ⟩ ≡

```
  {
    u32_t pinv;
    pinv = modulo32;
    pinv = 2 * pinv - pinv * pinv * modulo32;
    pinv = 2 * pinv - pinv * pinv * modulo32;
    pinv = 2 * pinv - pinv * pinv * modulo32;
#if 0
    pinv = 2 * pinv - pinv * pinv * modulo32;
#endif
    smalltd_pi[s][i] = 2 * pinv - pinv * pinv * modulo32;
  }
```

This code is used in section 69.

71. Finally, it is necessary to set some bounds on the arrays which we have just filled in to their correct values. Note that some of them (namely *smallsieve_aux1_ub* and *smallsieve_aux2_ub*) depend on *oddness_type* and are calculated at the beginning of each of the three subsieves.

⟨ Preparation job for the small FB primes 67 ⟩ +≡

```
{
  u32_t s;
  for (s = 0; s < 2; s++) {
    u16_t *x, *xx, k, pbound, copy_buf[6];
    k = 0;
    pbound = TINY_SIEVE_MIN;
    for (x = smallsieve_aux[s]; x < smallsieve_auxbound[s][0]; x += 4) {
      if (*x > pbound) {
        if (k ≡ 0) smallsieve_tinybound[s] = x;
        else smallsieve_auxbound[s][5 − k] = x;
        k++;
        if (k < 5) pbound = n_i/(5 − k);
        else break;
      }
    }
    while (k < 5) smallsieve_auxbound[s][5 − (k++)] = x;
    for (x = (xx = smallsieve_aux1[s]); x < smallsieve_aux1_ub_odd[s]; x += 6) {
      if (x[0] < TINY_SIEVE_MIN) {
        if (x ≠ xx) {
          memcpy(copy_buf, x, 6 * sizeof (*x));
          memcpy(x, xx, 6 * sizeof (*x));
          memcpy(xx, copy_buf, 6 * sizeof (*x));
        }
        xx += 6;
      }
    }
    smallsieve_tinybound1[s] = xx;
  }
}
```

72.

⟨ Prepare the medium and small primes for *oddness_type*. 72 ⟩ ≡

```
{
  u32_t s;
  for (s = 0; s < 2; s++) {
    switch (oddness_type) {
      u16_t *x;
    case 1: ⟨ Small sieve preparation for oddness type 1 75 ⟩
      break;
    case 2: ⟨ Small sieve preparation for oddness type 2 76 ⟩
      break;
    case 3: ⟨ Small sieve preparation for oddness type 3 77 ⟩
      break;
    }
  }
}
```

See also section 73.

This code is used in section 48.

73. In *gcd_sieve_buffer*$[2*i+1]$ we keep the offset from the current strip of the first j-line with j divisible by *gcd_sieve_buffer*$[2*i]$.

⟨ Prepare the medium and small primes for *oddness_type*. 72 ⟩ +≡

```
#ifdef GCD_SIEVE_BOUND
{
  u32_t i;
  for (i = 0; i < np_gcd_sieve; i++) {
    gcd_sieve_buffer[2*i+1] = (oddness_type/2) * (gcd_sieve_buffer[2*i]/2);
  }
}
#endif
```

74.

```
#ifdef GCD_SIEVE_BOUND
  static void gcd_sieve()
  {
    u32_t i;
    for (i = 0; i < np_gcd_sieve; i++) {
      u32_t x, p;
      x = gcd_sieve_buffer[2 * i + 1];
      p = gcd_sieve_buffer[2 * i];
      while (x < j_per_strip) {
        unsigned char *z, *z_ub;
        z = sieve_interval + (x << i_bits);
        z_ub = z + n_i - 3 * p;
        z += oddness_type ≡ 2 ? (n_i/2) % p : ((n_i + p - 1)/2) % p;
        while (z < z_ub) {
          *z = 0;
          *(z + p) = 0;
          z += 2 * p;
          *z = 0;
          *(z + p) = 0;
          z += 2 * p;
        }
        z_ub += 3 * p;
        while (z < z_ub) {
          *z = 0;
          z += p;
        }
        x = x + p;
      }
      gcd_sieve_buffer[2 * i + 1] = x - j_per_strip;
    }
  }
#endif
```

75. Odd factor base primes. This is fairly straightforward.

```
⟨Small sieve preparation for oddness type 1 75⟩ ≡
  for (x = smallsieve_aux[s]; x < smallsieve_auxbound[s][0]; x += 4) {
    u32_t p;
    p = x[0];
    x[3] = ((i_shift + p)/2) % p;
  }
```

See also sections 78, 81, and 85.

This code is used in section 72.

76.

⟨ Small sieve preparation for oddness type 2 76 ⟩ ≡

```
for (x = smallsieve_aux[s]; x < smallsieve_auxbound[s][0]; x += 4) {
    u32_t p, pr;
    p = x[0];
    pr = x[1];
    x[3] = (pr % 2 ≡ 0 ? ((i_shift + pr)/2) % p : ((i_shift + pr + p)/2) % p);
}
```

See also sections 79, 82, and 86.

This code is used in section 72.

77.

⟨ Small sieve preparation for oddness type 3 77 ⟩ ≡

```
for (x = smallsieve_aux[s]; x < smallsieve_auxbound[s][0]; x += 4) {
    u32_t p, pr;
    p = x[0];
    pr = x[1];
    x[3] = (pr % 2 ≡ 1 ? ((i_shift + pr)/2) % p : ((i_shift + pr + p)/2) % p);
}
```

See also sections 80, 83, and 87.

This code is used in section 72.

78. Odd factor base prime powers. This is just slightly more complicated.

⟨ Small sieve preparation for oddness type 1 75 ⟩ +≡

```
for (x = smallsieve_aux1[s]; x < smallsieve_aux1_ub_odd[s]; x += 6) {
    u32_t p;
    p = x[0];
    x[4] = ((i_shift + p)/2) % p;
    x[5] = 0;
}
```

79.

⟨ Small sieve preparation for oddness type 2 76 ⟩ +≡

```
for (x = smallsieve_aux1[s]; x < smallsieve_aux1_ub_odd[s]; x += 6) {
    u32_t p, d, pr;
    p = x[0];
    d = x[1];
    pr = x[2];
    x[4] = (pr % 2 ≡ 0 ? ((i_shift + pr)/2) % p : ((i_shift + pr + p)/2) % p);
    x[5] = d/2;
}
```

80.

⟨ Small sieve preparation for oddness type 3 77 ⟩ +≡

for $(x = smallsieve\_aux1[s];\ x < smallsieve\_aux1\_ub\_odd[s];\ x \mathrel{+}= 6)$ {
  u32_t $p,\ d,\ pr$;
  $p = x[0]$;
  $d = x[1]$;
  $pr = x[2]$;
  $x[4] = (pr \,\%\, 2 \equiv 1\ ?\ ((i\_shift + pr)/2) \,\%\, p : ((i\_shift + pr + p)/2) \,\%\, p)$;
  $x[5] = d/2$;
}

81. Roots blonging to odd prime powers located precisely at infinty in the projective space.

⟨ Small sieve preparation for oddness type 1 75 ⟩ +≡

for $(x = smallpsieve\_aux[s];\ x < smallpsieve\_aux\_ub\_odd[s];\ x \mathrel{+}= 3)\ x[2] = 0$;

82.

⟨ Small sieve preparation for oddness type 2 76 ⟩ +≡

for $(x = smallpsieve\_aux[s];\ x < smallpsieve\_aux\_ub\_odd[s];\ x \mathrel{+}= 3)\ x[2] = (x[0])/2$;

83.

⟨ Small sieve preparation for oddness type 3 77 ⟩ +≡

for $(x = smallpsieve\_aux[s];\ x < smallpsieve\_aux\_ub\_odd[s];\ x \mathrel{+}= 3)\ x[2] = (x[0])/2$;

84. Powers of two.

This is somewhat unpleasent. The follwoing two cases have to be distinguished:

a. In the original lattice coordinates, the sieving event occurs if $d \mid j$ and $i \cong rj/d \pmod p$, where $d > 1$ and p are powers of two. If $p = 1$, the prime goes to the horizontal factor base. Since only coprime i and j, r must be odd and j must be an odd multiple of d. Within the subsieve defined by this oddness type, we use $\tilde{j}$ with $j = 2 * \tilde{j}$, and the next possible $\tilde{j}$ after a given one is, due to the last remark, $\tilde{j} + d$. If $p = 1$ or $p = 2$, this goes to the horizontal factor base. Otherwise, r has to be added to residue class of the subsieve lattice coordinate $\tilde{i}$ and the residue class modulo $p/2$ has to be considered. In particular, if $p = 2$ the prime also goes to the horizontal factor base.

b. The sieving event occurs if $i \cong rj \pmod p$, where $p > 1$ is a power of two. If p is two, the prime ideal is treated by the horizontal factor base. Otherwise, transition to the next j-line is done by replacing $\tilde{j}$ by $\tilde{j} + 1$ and adding r to the residue class for $\tilde{i}$. Again, the residue class is not modulo p but modulo $p/2$. The oddness type is 2 for even and 3 for odd r.

85.

⟨ Small sieve preparation for oddness type 1 75 ⟩ +≡

```
{
  u16_t *x, *y, *z;
  u32_t i;
  x = smallsieve_aux1_ub_odd[s];
  y = smallpsieve_aux_ub_odd[s];
  z = smallsieve_aux2[s];
  for (i = 0; i < 4 * x2FBs[s]; i += 4) {
    u32_t p, pr, d, l;
    u16_t **a;
    d = x2FB[s][i + 1];
    if (d ≡ 1) continue;
    p = x2FB[s][i];
    pr = x2FB[s][i + 2];
    l = x2FB[s][i + 3];
    if (p < 4) {
      if (p ≡ 1) {
        *y = d/2;
        *(y + 2) = 0;
      }
      else {
        *y = d;
        *(y + 2) = d/2;
      }
      *(y + 1) = l;
      y += 3;
      continue;
    }
    p = p/2;
    if (p ≤ MAX_TINY_2POW) a = &z;
    else a = &x;
    **a = p;
    *(1 + *a) = d;
    *(2 + *a) = pr % p;
    *(3 + *a) = l;
    *(4 + *a) = ((i_shift + pr)/2) % p;
    *(5 + *a) = d/2;
    *a += 6;
  }
  smallsieve_aux1_ub[s] = x;
  smallpsieve_aux_ub[s] = y;
  smallsieve_aux2_ub[s] = z;
}
```

86.

⟨ Small sieve preparation for oddness type 2 76 ⟩ +≡

```
{
  u16_t *x, *y, *z;
  u32_t i;
  x = smallsieve_aux1_ub_odd[s];
  y = smallpsieve_aux_ub_odd[s];
  z = smallsieve_aux2[s];
  for (i = 0; i < 4 * x2FBs[s]; i += 4) {
    u32_t p, pr, d, l;
    u16_t **a;
    d = x2FB[s][i + 1];
    if (d ≠ 1) continue;
    pr = x2FB[s][i + 2];
    if (pr % 2 ≠ 0) continue;
    p = x2FB[s][i];
    l = x2FB[s][i + 3];
    if (p < 4) {   /* Horizontal. */
      if (p ≡ 1) {
        Schlendrian("Use␣1=2^0␣for␣sieving?\n");
      }
      *y = d;
      *(y + 1) = l;
      *(y + 2) = 0;
      y += 3;
      continue;
    }
    p = p/2;
    if (p ≤ MAX_TINY_2POW) a = &z;
    else a = &x;
    **a = p;
    *(1 + *a) = d;
    *(2 + *a) = pr % p;
    *(3 + *a) = l;
    *(4 + *a) = ((i_shift + pr)/2) % p;
    *(5 + *a) = 0;
    *a += 6;
  }
  smallsieve_aux1_ub[s] = x;
  smallpsieve_aux_ub[s] = y;
  smallsieve_aux2_ub[s] = z;
}
```

87.

⟨ Small sieve preparation for oddness type 3 77 ⟩ +≡

```
{
  u16_t *x, *y, *z;
  u32_t i;
  x = smallsieve_aux1_ub_odd[s];
  y = smallpsieve_aux_ub_odd[s];
  z = smallsieve_aux2[s];
  for (i = 0; i < 4 * x2FBs[s]; i += 4) {
    u32_t p, pr, d, l;
    u16_t **a;
    d = x2FB[s][i + 1];
    if (d ≠ 1) continue;
    pr = x2FB[s][i + 2];
    if (pr % 2 ≠ 1) continue;
    p = x2FB[s][i];
    l = x2FB[s][i + 3];
    if (p < 4) {     /* Horizontal. */
      if (p ≡ 1) {
        Schlendrian("Use␣1=2^0␣for␣sieving?\n");
      }
      *y = d;
      *(y + 1) = l;
      *(y + 2) = 0;
      y += 3;
      continue;
    }
    p = p/2;
    if (p ≤ MAX_TINY_2POW) a = &z;
    else a = &x;
    **a = p;
    *(1 + *a) = d;
    *(2 + *a) = pr % p;
    *(3 + *a) = l;
    *(4 + *a) = ((i_shift + pr)/2) % p;
    *(5 + *a) = 0;
    *a += 6;
  }
  smallsieve_aux1_ub[s] = x;
  smallpsieve_aux_ub[s] = y;
  smallsieve_aux2_ub[s] = z;
}
```

88.

⟨ Prepare the sieve 88 ⟩ ≡

```
{
  u32_t j;
  u16_t *x;
  for (x = smallsieve_aux[s], j = 0; x < smallsieve_tinybound[s]; x += 4, j++) {
    tinysieve_curpos[j] = x[3];
  }
  for (j = 0; j < j_per_strip; j++) {
    unsigned char *si_ub;
    bzero(tiny_sieve_buffer, TINY_SIEVEBUFFER_SIZE);
    si_ub = tiny_sieve_buffer + TINY_SIEVEBUFFER_SIZE;
    ⟨ Sieve tiny_sieve_buffer 89 ⟩
    ⟨ Spread tiny_sieve_buffer 92 ⟩
  }
  for (x = smallsieve_aux[s], j = 0; x < smallsieve_tinybound[s]; x += 4, j++) {
    x[3] = tinysieve_curpos[j];
  }
}
```

This code is used in section 48.

89.

⟨ Sieve *tiny_sieve_buffer* 89 ⟩ ≡

```
{
  u16_t *x;
  for (x = smallsieve_aux[s]; x < smallsieve_tinybound[s]; x += 4) {
    u32_t p, r, pr;
    unsigned char l, *si;
    p = x[0];
    pr = x[1];
    l = x[2];
    r = x[3];
    si = tiny_sieve_buffer + r;
    while (si < si_ub) {
      *si += l;
      si += p;
    }
    r = r + pr;
    if (r ≥ p) r = r - p;
    x[3] = r;
  }
}
```

See also sections 90 and 91.

This code is used in section 88.

90.

```
⟨ Sieve tiny_sieve_buffer 89 ⟩ +≡
  {
    u16_t *x;
    for (x = smallsieve_aux2[s]; x < smallsieve_aux2_ub[s]; x += 6) {
      u32_t p, r, pr, d, d0;
      unsigned char l, *si;
      p = x[0];
      d = x[1];
      pr = x[2];
      l = x[3];
      r = x[4];
      d0 = x[5];
      if (d0 > 0) {
        x[5]--;
        continue;
      }
      si = tiny_sieve_buffer + r;
      while (si < si_ub) {
        *si += l;
        si += p;
      }
      r = r + pr;
      if (r ≥ p) r = r - p;
      x[4] = r;
      x[5] = d - 1;
    }
  }
```

91.

⟨ Sieve *tiny_sieve_buffer* 89 ⟩ +≡

```
{
  u16_t *x;
  for (x = smallsieve_aux1[s]; x < smallsieve_tinybound1[s]; x += 6) {
    u32_t p, r, pr, d, d0;
    unsigned char l, *si;
    p = x[0];
    d = x[1];
    pr = x[2];
    l = x[3];
    r = x[4];
    d0 = x[5];
    if (d0 > 0) {
      x[5]--;
      continue;
    }
    si = tiny_sieve_buffer + r;
    while (si < si_ub) {
      *si += l;
      si += p;
    }
    r = r + pr;
    if (r ≥ p) r = r - p;
    x[4] = r;
    x[5] = d - 1;
  }
}
```

92.

⟨ Spread *tiny_sieve_buffer* 92 ⟩ ≡

```
{
  unsigned char *si;
  si = sieve_interval + (j ≪ i_bits);
  si_ub = sieve_interval + ((j + 1) ≪ i_bits);
  while (si + TINY_SIEVEBUFFER_SIZE < si_ub) {
    memcpy(si, tiny_sieve_buffer, TINY_SIEVEBUFFER_SIZE);
    si += TINY_SIEVEBUFFER_SIZE;
  }
  memcpy(si, tiny_sieve_buffer, si_ub - si);
}
```

This code is used in section 88.

93. This is for primes which will occur at least four times in each line.

⟨ Sieve with the small FB primes 93 ⟩ ≡

```
#ifdef ASM_LINESIEVER
  slinie(smallsieve_tinybound[s], smallsieve_auxbound[s][4], sieve_interval);
#else
  {
    u16_t *x;
    for (x = smallsieve_tinybound[s]; x < smallsieve_auxbound[s][4]; x += 4) {
      u32_t p, r, pr;
      unsigned char l, *y;
      p = x[0];
      pr = x[1];
      l = x[2];
      r = x[3];
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy, *yy_ub;
        yy_ub = y + n_i - 3 * p;
        for (yy = y + r; yy < yy_ub; yy = yy + 4 * p) {
          *(yy) += l;
          *(yy + p) += l;
          *(yy + 2 * p) += l;
          *(yy + 3 * p) += l;
        }
        while (yy < y + n_i) {
          *(yy) += l;
          yy += p;
        }
        r = r + pr;
        if (r ≥ p) r = r - p;
      }
#if 0
      x[3] = r;
#endif
    }
  }
#endif
```

See also sections 94, 95, 96, 97, 98, and 99.

This code is used in section 48.

94. FB primes occuring three or four times.

⟨ Sieve with the small FB primes 93 ⟩ +≡

```
#if 1
#ifdef ASM_LINESIEVER3
  slinie3(smallsieve_auxbound[s][4], smallsieve_auxbound[s][3], sieve_interval);
#else
  {
    u16_t *x;
    for (x = smallsieve_auxbound[s][4]; x < smallsieve_auxbound[s][3]; x += 4) {
      u32_t p, r, pr;
      unsigned char l, *y;
      p = x[0];
      pr = x[1];
      l = x[2];
      r = x[3];
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy;
        yy = y + r;
        *(yy) += l;
        *(yy + p) += l;
        *(yy + 2 * p) += l;
        yy += 3 * p;
        if (yy < y + n_i) *(yy) += l;
        r = r + pr;
        if (r ≥ p) r = r − p;
      }
#if 0
      x[3] = r;
#endif
    }
  }
#endif
#endif
```

95. FB primes occuring two or three times.

```
⟨ Sieve with the small FB primes 93 ⟩ +≡
#if 1
#ifdef ASM_LINESIEVER2
  slinie2(smallsieve_auxbound[s][3], smallsieve_auxbound[s][2], sieve_interval);
#else
  {
    u16_t *x;
    for (x = smallsieve_auxbound[s][3]; x < smallsieve_auxbound[s][2]; x += 4) {
      u32_t p, r, pr;
      unsigned char l, *y;

      p = x[0];
      pr = x[1];
      l = x[2];
      r = x[3];
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy;

        yy = y + r;
        *(yy) += l;
        *(yy + p) += l;
        yy += 2 * p;
        if (yy < y + n_i) *(yy) += l;
        r = r + pr;
        if (r ≥ p) r = r − p;
      }
#if 0
      x[3] = r;
#endif
    }
  }
#endif
#endif
```

96. FB primes occuring once or twice.

```
⟨Sieve with the small FB primes 93⟩ +≡
#if 1
#ifdef ASM_LINESIEVER1
  slinie1(smallsieve_auxbound[s][2], smallsieve_auxbound[s][1], sieve_interval);
#else
  {
    u16_t *x;
    for (x = smallsieve_auxbound[s][2]; x < smallsieve_auxbound[s][1]; x += 4) {
      u32_t p, r, pr;
      unsigned char l, *y;
      p = x[0];
      pr = x[1];
      l = x[2];
      r = x[3];
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy;
        yy = y + r;
        *(yy) += l;
        yy += p;
        if (yy < y + n_i) *(yy) += l;
        r = r + pr;
        if (r ≥ p) r = r − p;
      }
#if 0
      x[3] = r;
#endif
    }
  }
#endif
#endif
```

97. FB primes occuring at most once.

⟨ Sieve with the small FB primes 93 ⟩ +≡

```
#if 0
  {
    u16_t *x;
    for (x = smallsieve_auxbound[s][1]; x < smallsieve_auxbound[s][0]; x += 4) {
      u32_t p, r, pr;
      unsigned char l, *y;

      p = x[0];
      pr = x[1];
      l = x[2];
      r = x[3];
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        if (r < n_i) *(y + r) += l;
        r = r + pr;
        if (r ≥ p) r = r − p;
      }
#if 0
      x[3] = r;
#endif
    }
  }
#endif
```

98. Same thing for prime powers.

```
⟨ Sieve with the small FB primes 93 ⟩ +≡
#if 1
  {
    u16_t *x;
    for (x = smallsieve_tinybound1[s]; x < smallsieve_aux1_ub[s]; x += 6) {
      u32_t p, r, pr, d, d0;
      unsigned char l;

      p = x[0];
      d = x[1];
      pr = x[2];
      l = x[3];
      r = x[4];
      for (d0 = x[5]; d0 < j_per_strip; d0 += d) {
        unsigned char *y, *yy, *yy_ub;

        y = sieve_interval + (d0 ≪ i_bits);
        yy_ub = y + n_i − 3 * p;
        for (yy = y + r; yy < yy_ub; yy = yy + 4 * p) {
          *(yy) += l;
          *(yy + p) += l;
          *(yy + 2 * p) += l;
          *(yy + 3 * p) += l;
        }
        while (yy < y + n_i) {
          *(yy) += l;
          yy += p;
        }
        r = r + pr;
        if (r ≥ p) r = r − p;
      }
      x[4] = r;
      x[5] = d0 − j_per_strip;
    }
  }
#endif
```

99. Finally, the primes ideals or prime ideal powers defining the root projective infinity.

⟨ Sieve with the small FB primes 93 ⟩ +≡

```
#if 1
  {
    u16_t *x;
    bzero(horizontal_sievesums, j_per_strip);
    for (x = smallpsieve_aux[s]; x < smallpsieve_aux_ub[s]; x += 3) {
      u32_t p, d;
      unsigned char l;

      p = x[0];
      l = x[1];
      d = x[2];
      while (d < j_per_strip) {
        horizontal_sievesums[d] += l;
        d += p;
      }
#if 0
      x[2] = d - j_per_strip;
#endif
    }
  }
#else
  bzero(horizontal_sievesums, j_per_strip);
#endif
```

100. Sieving with the medium sized primes. On the little endian machine on which this siever was originally developed, a schedule entry lookes like (sieve interval index, factor base index). But on a big endian machine it may be more convenient to store things the other way around. This motivates the following # **define**ition.

```
⟨Sieve with the medium FB primes 100⟩ ≡
#ifndef MEDSCHE_SI_OFFS
#ifdef BIGENDIAN
#define MEDSCHED_SI_OFFS 1
#else
#define MEDSCHED_SI_OFFS 0
#endif
#endif
#ifdef ASM_SCHEDSIEVE1
  schedsieve(medsched_logs[s], n_medsched_pieces[s], med_sched[s], sieve_interval);
#else
  {
    u32_t l;
    for (l = 0; l < n_medsched_pieces[s]; l++) {
      unsigned char x;
      u16_t *schedule_ptr;
      x = medsched_logs[s][l];
#ifdef ASM_SCHEDSIEVE
      schedsieve(x, sieve_interval, med_sched[s][l], med_sched[s][l + 1]);
#else
      for (schedule_ptr = med_sched[s][l] + MEDSCHED_SI_OFFS; schedule_ptr + 3 * SE_SIZE <
            med_sched[s][l + 1]; schedule_ptr += 4 * SE_SIZE) {
        sieve_interval[*schedule_ptr] += x;
        sieve_interval[*(schedule_ptr + SE_SIZE)] += x;
        sieve_interval[*(schedule_ptr + 2 * SE_SIZE)] += x;
        sieve_interval[*(schedule_ptr + 3 * SE_SIZE)] += x;
      }
      for ( ; schedule_ptr < med_sched[s][l + 1]; schedule_ptr += SE_SIZE)
        sieve_interval[*schedule_ptr] += x;
#endif
    }
  }
#endif
```

This code is used in section 48.

101.

```
⟨Medsched 101⟩ ≡
#ifndef NOSCHED
  for (s = 0; s < 2; s++) { u32_t ll, *sched, *ri;
  if (n_medsched_pieces[s] ≡ 0) continue;
  for (ll = 0, sched = (u32_t *) med_sched[s][0], ri = LPri[s]; ll < n_medsched_pieces[s]; ll++) {
      ri = medsched(ri, current_ij[s]+medsched_fbi_bounds[s][ll], current_ij[s]+medsched_fbi_bounds[s][ll+1],
      &sched, medsched_fbi_bounds[s][ll], j_offset ≡ 0 ? oddness_type : 0); med_sched[s][ll + 1] = ( u16_t * )
      sched; } }
#endif
```

This code is used in section 48.

102. Use this to present the asm schedsieve function its arguments in a convenient way.

⟨ Global declarations 20 ⟩ +≡

```
u16_t **schedbuf;
```

103.

⟨ Prepare the lattice sieve scheduling 42 ⟩ +≡

```
{
  u32_t s;
  size_t schedbuf_alloc;
  for (s = 0, schedbuf_alloc = 0; s < 2; s++) {
    u32_t i;
    for (i = 0; i < n_schedules[s]; i++)
      if (schedules[s][i].n_pieces > schedbuf_alloc) schedbuf_alloc = schedules[s][i].n_pieces;
  }
  schedbuf = xmalloc((1 + schedbuf_alloc) * sizeof (*schedbuf));
}
```

104.

⟨ Sieve with the large FB primes 104 ⟩ ≡

```
#ifndef SCHED_SI_OFFS
#ifdef BIGENDIAN
#define SCHED_SI_OFFS 1
#else
#define SCHED_SI_OFFS 0
#endif
#endif
  {
    u32_t j;
    for (j = 0; j < n_schedules[s]; j++) {
      if (schedules[s][j].current_strip ≡ schedules[s][j].n_strips) {
        u32_t ns;   /* Number of strips for which to schedule */
        ns = schedules[s][j].n_strips;
        if (ns > n_strips − subsieve_nr) ns = n_strips − subsieve_nr;
        do_scheduling(schedules[s] + j, ns, 0, s);
        schedules[s][j].current_strip = 0;
      }
    }
#ifdef GATHER_STAT
    new_clock = clock();
    Schedule_clock += new_clock − last_clock;
    last_clock = new_clock;
#endif
    for (j = 0; j < n_schedules[s]; j++) {
#ifdef ASM_SCHEDSIEVE1
      u32_t i, k;
      k = schedules[s][j].current_strip;
      for (i = 0; i ≤ schedules[s][j].n_pieces; i++) {
        schedbuf[i] = schedules[s][j].schedule[i][k];
      }
      schedsieve(schedules[s][j].schedlogs, schedules[s][j].n_pieces, schedbuf, sieve_interval);
#else
      u32_t l, k;
      k = schedules[s][j].current_strip;
      l = 0;
      while (l < schedules[s][j].n_pieces) {
        unsigned char x;
        u16_t *schedule_ptr, *sptr_ub;
        x = schedules[s][j].schedlogs[l];
        schedule_ptr = schedules[s][j].schedule[l][k] + SCHED_SI_OFFS;
        while (l < schedules[s][j].n_pieces)
          if (schedules[s][j].schedlogs[++l] ≠ x) break;
        sptr_ub = schedules[s][j].schedule[l][k];
#ifdef ASM_SCHEDSIEVE
        schedsieve(x, sieve_interval, schedule_ptr, sptr_ub);
#else
        while (schedule_ptr + 3 * SE_SIZE < sptr_ub) {
          sieve_interval[*schedule_ptr] += x;
```

```
        sieve_interval[*(schedule_ptr + SE_SIZE)] += x;
        sieve_interval[*(schedule_ptr + 2 * SE_SIZE)] += x;
        sieve_interval[*(schedule_ptr + 3 * SE_SIZE)] += x;
        schedule_ptr += 4 * SE_SIZE;
      }
      while (schedule_ptr < sptr_ub) {
        sieve_interval[*schedule_ptr] += x;
        schedule_ptr += SE_SIZE;
      }
#endif
    }
#endif
  }
}
```

This code is used in section 48.

105. Copyright (C) 2001 Jens Franke. This file is part of gnfs4linux, distributed under the terms of the GNU General Public Licence and WITHOUT ANY WARRANTY.

You should have received a copy of the GNU General Public License along with this program; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

```
⟨ Candidate search 105 ⟩ ≡
#ifdef ASM_SEARCH0
  {
    unsigned char *srbs;
    u32_t i;

    srbs = sieve_report_bounds[s][j_offset/CANDIDATE_SEARCH_STEPS];
    ncand = lasieve_search0(sieve_interval, horizontal_sievesums, horizontal_sievesums + j_per_strip, srbs,
          srbs + n_i/CANDIDATE_SEARCH_STEPS, cand, fss_sv);
    for (i = 0; i < ncand; i++) fss_sv[i] += horizontal_sievesums[cand[i] >> i_bits];
  }
#else
  {
    unsigned char *srbs;
    u32_t i;

    srbs = sieve_report_bounds[s][j_offset/CANDIDATE_SEARCH_STEPS];
    ncand = 0;
    for (i = 0; i < n_i; i += CANDIDATE_SEARCH_STEPS) {
      unsigned char st;
      u32_t j;

      st = *(srbs++);
      for (j = 0; j < j_per_strip; j++) {
        unsigned char *i_o, *i_max, st1;

        i_o = sieve_interval + (j << i_bits) + i;
        i_max = i_o + CANDIDATE_SEARCH_STEPS;
        if (st ≤ horizontal_sievesums[j]) {
          while (i_o < i_max) {
            cand[ncand] = i_o − sieve_interval;
            fss_sv[ncand++] = *(i_o++) + horizontal_sievesums[j];
          }
          continue;
        }
        st1 = st − horizontal_sievesums[j];
        ⟨ MMX Candidate searcher 106 ⟩
      }
    }
  }
#endif
#if 0
  {
    char *ofn;
    FILE *of;

    asprintf(&ofn, "cdump.%u.%u.j%u.ot%u", special_q, r[root_no], j_offset, oddness_type);
    if ((of = fopen(ofn, "w")) ≠ Λ) {
      u32_t i;

      fprintf(of, "%u␣candidates\n", ncand);
```

```
      for (i = 0; i < ncand; i++) fprintf(of, "%u␣%u\n", cand[i], fss_sv[i]);
      fclose(of);
    }
    else errprintf("Cannot␣open␣debug␣file␣%s:␣%m\n", ofn);
    free(ofn);
  }
#endif
```

This code is used in section 48.

106.

```
⟨ MMX Candidate searcher 106 ⟩ ≡
#ifndef HAVE_SSIMD
#ifdef GNFS_CS32  /* Use 32 bit registers for candidate search. */
#define bc_t unsigned long
#define BC_MASK #80808080
#else
#define bc_t unsigned long long
#define BC_MASK #8080808080808080
#endif
    { if (st1 < #80) { bc_t bc, *i_oo;
    bc = st1;
    bc = (bc ≪ 8) | bc;
    bc = (bc ≪ 16) | bc;
#ifndef GNFS_CS32
    bc = (bc ≪ 32) | bc;
#endif
    bc = BC_MASK − bc; for ( i_oo = ( bc_t * ) i_o; i_oo < ( bc_t * ) i_max; i_oo++ )
    {
      bc_t v = *i_oo;
      if (((v & BC_MASK) | ((v + bc) & BC_MASK)) ≡ 0) continue;
      for (i_o = (unsigned char *) i_oo; i_o < (unsigned char *)(i_oo + 1); i_o++) {
        if (*i_o ≥ st1) {
          ⟨ Store survivor 107 ⟩
        }
      }
    }
    } else { bc_t * i_oo; for ( i_oo = ( bc_t * ) i_o; i_oo < ( bc_t * ) i_max; i_oo++ )
    {
      if ((*i_oo & BC_MASK) ≡ 0) continue;
      for (i_o = (unsigned char *) i_oo; i_o < (unsigned char *)(i_oo + 1); i_o++) {
        if (*i_o ≥ st1) {
          ⟨ Store survivor 107 ⟩
        }
      }
    }
    } }
#else
    { unsigned long long x;
    x = st1 − 1;
    x |= x ≪ 8;
    x |= x ≪ 16;
    x |= x ≪ 32; while (i_o < i_max) { asm volatile (
        "movq␣(%%eax),%%mm7\n""1:\n""movq␣(%%esi),%%mm1\n""movq␣8(%%esi),%%mm0\
        \n""pmaxub␣16(%%esi),%%mm1\n""pmaxub␣24(%%esi),%%mm0\n""pmaxub␣%%mm7,%%mm1\\
        n""pmaxub␣%%mm1,%%mm0\n""pcmpeqb␣%%mm7,%%mm0\n""pmovmskb␣%%mm0,%%ea\
        x\n""cmpl␣$255,%%eax\n""jnz␣2f\n""leal␣32(%%esi),%%es\
        i\n""cmpl␣%%esi,%%edi\n""ja␣1b\n""2:\n""emms":
    "=S"(i_o): "a"(&x), "S"(i_o), "D"(i_max) ) ;
    if (i_o < i_max) {
      unsigned char *i_max2 = i_o + 32;
```

```
          while (i_o < i_max2) {
            if (*i_o ≥ st1) {
              ⟨ Store survivor 107 ⟩
            }
            i_o++;
          }
        }
      } }
#endif
```

This code is used in section 105.

107.

⟨ Store survivor 107 ⟩ ≡

```
  cand[ncand] = i_o − sieve_interval;
  fss_sv[ncand++] = *i_o + horizontal_sievesums[j];
```

This code is used in section 106.

108.

⟨ Final candidate search 108 ⟩ ≡

```
  {
    u32_t i, nc1;
    unsigned char *srbs;
    static u32_t bad_pvl = 0;
    double sr_inv;

    srbs = sieve_report_bounds[s][j_offset/CANDIDATE_SEARCH_STEPS];
    n_prereports += ncand;
    if (ncand) sr_inv = 1./(M_LN2 * sieve_multiplier[s]);
    for (i = 0, nc1 = 0; i < ncand; i++) {
      u16_t st_i, t_j, ii, jj, j;

      double pvl;

      j = cand[i] >> i_bits;
#ifndef DEBUG_SIEVE_REPORT_BOUNDS
      if (sieve_interval[cand[i]] + horizontal_sievesums[j] < srbs[(cand[i] & (n_i −
            1))/CANDIDATE_SEARCH_STEPS]) continue;
#endif
      jj = j_offset + j;
      ii = cand[i] & (n_i − 1);
      st_i = 2 * ii + (oddness_type ≡ 2 ? 0 : 1);
      t_j = 2 * jj + (oddness_type ≡ 1 ? 0 : 1);
#if 1
      pvl = log(fabs(rpol_eval(tpoly_f[s], poldeg[s], (double) st_i − (double) i_shift, (double) t_j)));
#else
      pvl = log(fabs(rpol_eval0(tpoly_f[s], poldeg[s], (i32_t)st_i − (i32_t)i_shift, t_j)));
#endif
      if (special_q_side ≡ s) pvl −= special_q_log;
      pvl *= sieve_multiplier[s];
      pvl −= sieve_report_multiplier[s] * FB_maxlog[s];
      if ((double)(sieve_interval[cand[i]] + horizontal_sievesums[j]) ≥ pvl) {
          /* In fss_sv2 we save an approximation of the number of bits of the cofactor: */
        pvl += sieve_report_multiplier[s] * FB_maxlog[s];
        pvl −= (double)(sieve_interval[cand[i]] + horizontal_sievesums[j]);
        if (pvl < 0.) pvl = 0.;
        pvl *= sr_inv;      /* pvl/=(M_LN2 * sieve_multiplier[s]); */      /* pvl/=M_LN2; */
        fss_sv2[nc1] = (unsigned char)(pvl);
#ifdef DEBUG_SIEVE_REPORT_BOUNDS
        ⟨ Test correctness of sieve report bounds 109 ⟩
#endif
        fss_sv[nc1] = fss_sv[i];
        cand[nc1++] = cand[i];
      }
    }
    rpol_eval_clear();
    ncand = nc1;
  }
```

This code is used in section 48.

109.

⟨ Test correctness of sieve report bounds 109 ⟩ ≡

```
if (sieve_interval[cand[i]] + horizontal_sievesums[j] < srbs[(cand[i] & (n_i - 1))/CANDIDATE_SEARCH_STEPS])
  {
  double pvl1;
  pvl = fabs(rpol_eval(tpoly_f[s], poldeg[s], (double) st_i - (double) i_shift, (double) t_j));
  fprintf(stderr, "Bad pvl min %u at (%f,%f),spq=%u\npvl: %.5g->", bad_pvl++, (double)
      st_i - (double) i_shift, (double) t_j, special_q, pvl);
  pvl = log(pvl);
  fprintf(stderr, "%.3f->", pvl);
  pvl = sieve_multiplier[s] * pvl;
  fprintf(stderr, "%.3f->", pvl);
  if (special_q_side == s) pvl -= sieve_multiplier[s] * special_q_log;
  fprintf(stderr, "%.3f->", pvl);
  pvl -= sieve_report_multiplier[s] * FB_maxlog[s];
  fprintf(stderr, "%.3f\nLower bound was %u sv was %u=%u+%u\n", pvl, (u32_t)
      srbs[(cand[i] & (n_i - 1))/CANDIDATE_SEARCH_STEPS], (u32_t) sieve_interval[cand[i]] + (u32_t)
      horizontal_sievesums[j], (u32_t) sieve_interval[cand[i]], (u32_t) horizontal_sievesums[j]);
}
```

This code is used in section 108.

110.

⟨ Global declarations 20 ⟩ +≡

```
static void store_candidate(u16_t, u16_t, unsigned char);
```

111.

```
static void xFBtranslate(u16_t *rop, xFBptr op)
{
  u32_t x, y, am, bm, rqq;
  modulo32 = op→pp;
  rop[3] = op→l;
  am = a1 > 0 ? ((u32_t) a1) % modulo32 : modulo32 − ((u32_t)(−a1)) % modulo32;
  if (am ≡ modulo32) am = 0;
  bm = b1 > 0 ? ((u32_t) b1) % modulo32 : modulo32 − ((u32_t)(−b1)) % modulo32;
  if (bm ≡ modulo32) bm = 0;
  x = modsub32(modmul32(op→qq, am), modmul32(op→r, bm));
  am = a0 > 0 ? ((u32_t) a0) % modulo32 : modulo32 − ((u32_t)(−a0)) % modulo32;
  if (am ≡ modulo32) am = 0;
  bm = b0 > 0 ? ((u32_t) b0) % modulo32 : modulo32 − ((u32_t)(−b0)) % modulo32;
  if (bm ≡ modulo32) bm = 0;
  y = modsub32(modmul32(op→r, bm), modmul32(op→qq, am));
  rqq = 1;
  if (y ≠ 0) {
    while (y % (op→p) ≡ 0) {
      y = y/(op→p);
      rqq *= op→p;
    }
  }
  else {
    rqq = op→pp;
  }
  modulo32 = modulo32/rqq;
  rop[0] = modulo32;
  rop[1] = rqq;
  if (modulo32 > 1) rop[2] = modmul32(modinv32(y), x);
  else rop[2] = 0;
  rop[4] = op→l;
}
```

112.

```
static int xFBcmp(const void *opA, const void *opB)
{
  xFBptr op1, op2;
  op1 = (xFBptr) opA;
  op2 = (xFBptr) opB;
  if (op1→pp < op2→pp) return −1;
  if (op1→pp ≡ op2→pp) return 0;
  return 1;
}
```

113. The function is implemented by recursive calls to itself.

```
static u32_t add_primepowers2xaFB(size_t *xaFB_alloc_ptr, u32_t pp_bound, u32_t s, u32_t p, u32_t
        r)
{
  u32_t a, b, q, qo, *rbuf, nr, *Ar, exponent, init_xFB;
  size_t rbuf_alloc;

  if (xFBs[s] ≡ 0 ∧ p ≡ 0) Schlendrian("add_primepowers2xaFB␣on␣empty␣xaFB\n");
  rbuf_alloc = 0;
  Ar = xmalloc((1 + poldeg[s]) * sizeof (*Ar));
  if (p ≠ 0) {
    init_xFB = 0;
    q = p;
    if (r ≡ p) {
      a = 1;
      b = p;
    }
    else {
      a = r;
      b = 1;
    }
  }
  else {
    init_xFB = 1;
    q = xFB[s][xFBs[s] − 1].pp;
    p = xFB[s][xFBs[s] − 1].p;
    a = xFB[s][xFBs[s] − 1].r;
    b = xFB[s][xFBs[s] − 1].qq;
  }
  qo = q;
  exponent = 1;
  for ( ; ; ) {
    u32_t j, r;

    if (q > pp_bound/p) break;
    modulo32 = p * q;
    for (j = 0; j ≤ poldeg[s]; j++) Ar[j] = mpz_fdiv_ui(poly[s][j], modulo32);
    if (b ≡ 1) ⟨Determine affine roots 114⟩
    else ⟨Determine projective roots 115⟩
    if (qo * nr ≠ modulo32) break;
    q = modulo32;
    exponent++;
  }
  if (init_xFB ≠ 0)
    xFB[s][xFBs[s] − 1].l = rint(sieve_multiplier_small[s] * log(q)) − rint(sieve_multiplier_small[s] *
          log(qo/p));
  if (q ≤ pp_bound/p) {
    u32_t j;

    for (j = 0; j < nr; j++) {
      ⟨Create xaFB[xaFBs] 116⟩
      xFBs[s]++;
      add_primepowers2xaFB(xaFB_alloc_ptr, pp_bound, s, 0, 0);
    }
```

```
    }
    if (rbuf_alloc > 0) free(rbuf);
    free(Ar);
    return exponent;
  }
```

114.

⟨ Determine affine roots 114 ⟩ ≡

```
  {
    for (r = a, nr = 0; r < modulo32; r += qo) {
      u32_t pv;
      for (j = 1, pv = Ar[poldeg[s]]; j ≤ poldeg[s]; j++) {
        pv = modadd32(Ar[poldeg[s] − j], modmul32(pv, r));
      }
      if (pv ≡ 0) {
        adjust_bufsize((void **) &rbuf, &rbuf_alloc, 1 + nr, 4, sizeof (*rbuf));
        rbuf[nr++] = r;
      }
      else if (pv % q ≠ 0) Schlendrian("xFBgen:␣%u␣not␣a␣root␣mod␣%u\n", r, q);
    }
  }
```

This code is used in section 113.

115.

⟨ Determine projective roots 115 ⟩ ≡

```
  {
    for (r = (modmul32(b, modinv32(a))) % qo, nr = 0; r < modulo32; r += qo) {
      u32_t pv;
      for (j = 1, pv = Ar[0]; j ≤ poldeg[s]; j++) {
        pv = modadd32(Ar[j], modmul32(pv, r));
      }
      if (pv ≡ 0) {
        adjust_bufsize((void **) &rbuf, &rbuf_alloc, 1 + nr, 4, sizeof (*rbuf));
        rbuf[nr++] = r;
      }
      else if (pv % q ≠ 0) Schlendrian("xFBgen:␣%u^{-1}␣not␣a␣root␣mod␣%u\n", r, q);
    }
  }
```

This code is used in section 113.

116.

⟨ Create $xaFB[xaFBs]$ 116 ⟩ ≡

```
xFBptr f;
adjust_bufsize((void **) &(xFB[s]), xaFB_alloc_ptr, 1 + xFBs[s], 16, sizeof (**xFB));
f = xFB[s] + xFBs[s];
f→p = p;
f→pp = q * p;
if (b ≡ 1) {
  f→qq = 1;
  f→r = rbuf[j];
  f→q = f→pp;
}
else {
  modulo32 = (q * p)/b;
  rbuf[j] = rbuf[j]/b;
  if (rbuf[j] ≡ 0) {
    f→qq = f→pp;
    f→q = 1;
    f→r = 1;
  }
  else {
    while (rbuf[j] % p ≡ 0) {
      rbuf[j] = rbuf[j]/p;
      modulo32 = modulo32/p;
    }
    f→qq = (f→pp)/modulo32;
    f→q = modulo32;
    f→r = modinv32(rbuf[j]);
  }
}
```

This code is used in section 113.

117. Trial division and output code.

```
⟨ Global declarations 20 ⟩ +≡
  void trial_divide(void);
```

118.

```
  void trial_divide()
  {
    u32_t ci;    /* Candidate index. */
    u32_t nc1;    /* Survivors of current TD step. */
    u16_t side, tdstep;
    clock_t last_tdclock, newclock;
#ifdef NO_TDCODE
    return;
#endif
    ⟨ gcd and size checks 119 ⟩
#ifdef ZSS_STAT
    if (ncand ≡ 0) nzss[1]++;
#endif
    last_tdclock = clock();
    tdi_clock += last_tdclock − last_clock;
    ncand = nc1;
    qsort(cand, ncand, sizeof (*cand), tdcand_cmp);
    td_buf1[0] = td_buf[first_td_side];
    for (side = first_td_side, tdstep = 0; tdstep < 2; side = 1 − side, tdstep++) {
#ifdef ZSS_STAT
      if (tdstep ≡ 1 ∧ ncand ≡ 0) nzss[2]++;
#endif
      ⟨ td for this side 123 ⟩
    }
  }
```

119.

```
⟨ gcd and size checks 119 ⟩ ≡
  {
    for (ci = 0, nc1 = 0; ci < ncand; ci++) {
      u16_t strip_i, strip_j;
      u16_t st_i, true_j;      /* Semi-true i and true j */
      u16_t s;      /* Side. */
      double pvl, pvl0;
      ⟨ Calculate st_i and true_j 120 ⟩
      n_reports++;
      s = first_sieve_side;
#ifdef STC_DEBUG
      fprintf(debugfile, "%hu␣%hu\n", st_i, true_j);
#endif
      if (gcd32(st_i < i_shift ? i_shift − st_i : st_i − i_shift, true_j) ≠ 1) continue;
      n_rep1++;
#if 1
      pvl = log(fabs(rpol_eval(tpoly_f[s], poldeg[s], (double) st_i − (double) i_shift, (double) true_j)));
#else
      pvl = log(fabs(rpol_eval0(tpoly_f[s], poldeg[s], (i32_t)st_i − (i32_t)i_shift, true_j)));
#endif
      if (special_q_side ≡ s) pvl −= special_q_log;
      pvl0 = pvl;
      pvl *= sieve_multiplier[s];
      if ((double) fss_sv[ci] + sieve_report_multiplier[s] * FB_maxlog[s] < pvl) continue;
#if 1
      {
        u32_t n0, n1;
        pvl0 −= (double)(fss_sv[ci])/sieve_multiplier[s];
        if (pvl0 < 0.) pvl0 = 0.;
        pvl0 /= M_LN2;
        if (s ≡ special_q_side) {
          n0 = (u32_t) pvl0;
          n1 = (u32_t)(fss_sv2[ci]);
        }
        else {
          n0 = (u32_t)(fss_sv2[ci]);
          n1 = (u32_t) pvl0;
        }
        if (n0 > max_factorbits[0]) n0 = max_factorbits[0];
        if (n1 > max_factorbits[1]) n1 = max_factorbits[1];
        if (strat.bit[n0][n1] ≡ 0) {
          n_abort1++;
          continue;
        }
      }
#endif
      n_rep2++;
        /* Make sure that the special q is not a common divisor of the (a,b)-pair corresponding to (i,j). */
      modulo64 = special_q;
      if (modadd64(modmul64((u64_t)st_i, spq_i), modmul64((u64_t)true_j, spq_j)) ≡ spq_x) continue;
```

```
      cand[nc1++] = cand[ci];
    }
    rpol_eval_clear();
  }
```

This code is used in section 118.

120.

⟨ Calculate *st_i* and *true_j* 120 ⟩ ≡

```
  {
    u16_t jj;
    strip_j = cand[ci] ≫ i_bits;
    jj = j_offset + strip_j;
    strip_i = cand[ci] & (n_i − 1);
    st_i = 2 ∗ strip_i + (oddness_type ≡ 2 ? 0 : 1);
    true_j = 2 ∗ jj + (oddness_type ≡ 1 ? 0 : 1);
  }
```

This code is used in sections 119 and 123.

121. The buffers *td_buf*[0] and *td_buf*[1] hold the primes below the factor base bound (not the factor base indices!) for all trial division candidates in the current subsieve. When we do the trial division on the second side s, *td_buf1*[j] will point to the first entry of the j-th candidate in *td_buf*[$1 - s$]. Note that *td_buf*[s] will never be used for more than one candidate, since the surviving candidates of this trial division pass are output immediately.

⟨ Trial division declarations 121 ⟩ ≡

```
  u32_t ∗(td_buf[2]), ∗∗td_buf1;
  size_t td_buf_alloc[2] = {1024, 1024};
```

See also sections 126 and 146.

This code is used in section 16.

122.

⟨ TD Init 122 ⟩ ≡

```
  td_buf1 = xmalloc((1 + L1_SIZE) ∗ sizeof (∗td_buf1));
  td_buf[0] = xmalloc(td_buf_alloc[0] ∗ sizeof (∗∗td_buf));
  td_buf[1] = xmalloc(td_buf_alloc[1] ∗ sizeof (∗∗td_buf));
```

See also sections 127 and 147.

This code is used in section 16.

123. We store a one byte value (which is always positive) for each trial division candidate in *fss_sv*. This is also written to the corresponding location of the sieve interval. The other entries of the sieve interval are set to zero. The trial division sieve stores all large prime indices which are relevant for the location i of the sieve interval in an array *tds_fbi*[*sieve_interval*[*i*]].

⟨ td for this side 123 ⟩ ≡

```
  {
    u32_t nfbp;    /* Total number of factor base primes stored in td_buf[side] so far. */
    u32_t p_bound;    /* Bound for factor base primes which are treated by sieving. */
    u16_t last_j, strip_i, strip_j;
    u16_t *smalltdsieve_auxbound;
    nfbp = 0;    /* * Feel free to EXPERIMENT with this bound, by trying versions * like
        p_bound = (2 * n_i * j_per_strip)/(5 * ncand) * or p_bound = (n_i * j_per_strip)/(3 * ncand). */
#ifndef SET_TDS_PBOUND
    if (ncand > 0) p_bound = (2 * n_i * j_per_strip)/(5 * ncand);
    else p_bound = U32_MAX;
#else
    p_bound = SET_TDS_PBOUND(n_i, j_per_strip, ncand);
#endif
    ⟨ tds init 124 ⟩
    newclock = clock();
    tdsi_clock[side] += newclock − last_tdclock;
    last_tdclock = newclock;
    ⟨ td sieve 128 ⟩
    last_j = 0;
    for (ci = 0, nc1 = 0; ci < ncand; ci++) {
      u32_t *fbp_buf;    /* Buffer for primes < FB_bound for current candidate. */
      u32_t *fbp_ptr;    /* Current position in this buffer. */
      u16_t st_i, true_j;
      i32_t true_i;
      u32_t coll;    /* Did a hash collision occur in the tdsieve for this ci ? */
      ⟨ Calculate st_i and true_j 120 ⟩
      if (strip_j ≠ last_j) {
        u16_t j_step;
        if (strip_j ≤ last_j) Schlendrian("TD:␣Not␣sorted\n");
        j_step = strip_j − last_j;
        last_j = strip_j;
        ⟨ Update smallsieve_aux for TD 134 ⟩
      }
      true_i = (i32_t)st_i − (i32_t)i_shift;
      ⟨ Calculate sr_a and sr_b 135 ⟩
      ⟨ Calculate value of norm polynomial in aux1 138 ⟩
      if (td_buf_alloc[side] < nfbp + mpz_sizeinbase(aux1, 2)) {
        td_buf_alloc[side] += 1024;
        while (td_buf_alloc[side] < nfbp + mpz_sizeinbase(aux1, 2)) {
          td_buf_alloc[side] += 1024;
        }
        td_buf[side] = xrealloc(td_buf[side], td_buf_alloc[side] * sizeof (**td_buf));
        if (side ≡ first_td_side) {
          u32_t i, *oldptr;
          oldptr = td_buf1[0];
          for (i = 0; i ≤ nc1; i++) td_buf1[i] = td_buf[side] + (td_buf1[i] − oldptr);
```

```
          }
        }
        if (side ≡ first_td_side) fbp_buf = td_buf1[nc1];
        else fbp_buf = td_buf[side];
        fbp_ptr = fbp_buf;
        ⟨td by sieving 139⟩
        ⟨Small FB td 140⟩
        ⟨Special q td 142⟩
        ⟨Execute TD 143⟩
        ⟨Special q 64 bit td 144⟩
        ⟨rest of td 145⟩
      }
#ifndef MMX_TD
      {
        u16_t j_step;
        j_step = j_per_strip − last_j;
        ⟨Update smallsieve_aux for TD 134⟩
      }
#else
      {
        u16_t *x, j_step;
        j_step = j_per_strip − last_j;
        for (x = smallpsieve_aux[side]; x < smallpsieve_aux_ub[side]; x += 3) {
          modulo32 = x[0];
          x[2] = modsub32(x[2], (j_step) % modulo32);
        }
      }
#endif
      newclock = clock();
      tds4_clock[side] += newclock − last_tdclock;
      last_tdclock = newclock;
      ncand = nc1;
    }
```

This code is used in section 118.

124.

```
⟨ tds init 124 ⟩ ≡
  {
    unsigned char ht, allcoll;
    bzero(sieve_interval, L1_SIZE);
    bzero(tds_coll, UCHAR_MAX − 1);
    for (ci = 0, ht = 1, allcoll = 0; ci < ncand; ci++) {
      unsigned char cht;
      cht = sieve_interval[cand[ci]];
      if (cht ≡ 0) {
        cht = ht;
        if (ht < UCHAR_MAX) ht++;
        else {
          ht = 1;
          allcoll = 1;
        }
        tds_coll[cht − 1] = allcoll;
        sieve_interval[cand[ci]] = cht;
      }
      else {
        tds_coll[cht − 1] = 1;
      }
      fss_sv[ci] = cht − 1;
    }
  }
```

See also section 125.

This code is used in section 123.

125. If we use MMX or similar instructions for trial division, it is necessary to arrange the information contained in *smallsieve_aux* in a form which is suitable for use by these instructions. The function which does this should update the location of the sieving event for these factor base primes. In addition, it may also change *pbound* since the number of factor base elements treated by MMX instructions may be required to be even or divisible by four. This is the reason for calling this initialization function before starting the trial division sieve.

```
⟨ tds init 124 ⟩ +≡
#ifdef MMX_TD
  smalltdsieve_auxbound = MMX_TdInit(side, smallsieve_aux[side], smallsieve_auxbound[side][0],
        &p_bound, j_offset ≡ 0 ∧ oddness_type ≡ 1);
#else
  {
    u16_t *x, *z;
    x = smallsieve_aux[side];
    z = smallsieve_auxbound[side][0];
    if (*x > p_bound) smalltdsieve_auxbound = x;
    else {
      while (x + 4 < z) {
        u16_t *y;
        y = x + 4 * ((z − x)/8);
        if (y ≡ smallsieve_auxbound[side][0] ∨ *y > p_bound) z = y;
        else x = y;
      }
      smalltdsieve_auxbound = z;
    }
  }
#endif
```

126.

```
⟨ Trial division declarations 121 ⟩ +≡
  static unsigned char tds_coll[UCHAR_MAX];
  u32_t **tds_fbi = Λ;
  u32_t **tds_fbi_curpos = Λ;
#ifndef TDFBI_ALLOC
#define TDFBI_ALLOC 256
  static size_t tds_fbi_alloc = TDFBI_ALLOC;
#endif
```

127.

```
⟨ TD Init 122 ⟩ +≡
  {
    u32_t i;
    if (tds_fbi ≡ Λ) {
      tds_fbi = xmalloc(UCHAR_MAX * sizeof (*tds_fbi));
      tds_fbi_curpos = xmalloc(UCHAR_MAX * sizeof (*tds_fbi));
      for (i = 0; i < UCHAR_MAX; i++) tds_fbi[i] = xmalloc(tds_fbi_alloc * sizeof (**tds_fbi));
    }
  }
```

128. Set number of factor base indices which are stored so far to zero.

⟨ td sieve 128 ⟩ ≡

```
memcpy(tds_fbi_curpos, tds_fbi, UCHAR_MAX * sizeof (*tds_fbi));
```

See also sections 129, 130, 132, and 133.

This code is used in section 123.

129. Store the factor base indices ≥ *fbis*[*side*] but < *fbi1*[*side*]

⟨ td sieve 128 ⟩ +≡

```
#ifdef ASM_SCHEDTDSIEVE
  {
    u32_t x, *(y[2]);
    x = 0;
    y[0] = med_sched[side][0];
    y[1] = med_sched[side][n_medsched_pieces[side]];
    schedtdsieve(&x, 1, y, sieve_interval, tds_fbi_curpos);
  }
#else
  {
    u32_t l;
    for (l = 0; l < n_medsched_pieces[side]; l++) {
      u16_t *x, *x_ub;
      x_ub = med_sched[side][l + 1];
      for (x = med_sched[side][l] + MEDSCHED_SI_OFFS; x + 6 < x_ub; x += 8) {
        unsigned char z;
        if ((sieve_interval[*x] | sieve_interval[*(x+2)] | sieve_interval[*(x+4)] | sieve_interval[*(x+6)]) ≡ 0)
        {
          continue;
        }
        if ((z = sieve_interval[*x]) ≠ 0) *(tds_fbi_curpos[z − 1]++) = *(x + 1 − 2 * MEDSCHED_SI_OFFS);
        if ((z = sieve_interval[*(x + 2)]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = *(x + 3 − 2 * MEDSCHED_SI_OFFS);
        if ((z = sieve_interval[*(x + 4)]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = *(x + 5 − 2 * MEDSCHED_SI_OFFS);
        if ((z = sieve_interval[*(x + 6)]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = *(x + 7 − 2 * MEDSCHED_SI_OFFS);
      }
      while (x < x_ub) {
        unsigned char z;
        if ((z = sieve_interval[*x]) ≠ 0) *(tds_fbi_curpos[z − 1]++) = *(x + 1 − 2 * MEDSCHED_SI_OFFS);
        x += 2;
      }
    }
  }
#endif
  newclock = clock();
  tds2_clock[side] += newclock − last_tdclock;
  last_tdclock = newclock;
```

130. Next, store the factor base indices ≥ *fbi1* [*side*].

⟨ td sieve 128 ⟩ +≡

```
  {
    u32_t j;
    for (j = 0; j < n_schedules[side]; j++) {
#ifdef ASM_SCHEDTDSIEVE
      u32_t i, k;
      k = schedules[side][j].current_strip++;
      for (i = 0; i ≤ schedules[side][j].n_pieces; i++) {
        schedbuf[i] = schedules[side][j].schedule[i][k];
      }
      schedtdsieve(schedules[side][j].fbi_bounds, schedules[side][j].n_pieces, schedbuf, sieve_interval,
          tds_fbi_curpos);
#else
#if 1
      u32_t k, l, fbi_offset;
      u16_t *x, *x_ub;
      k = schedules[side][j].current_strip++;
      x = schedules[side][j].schedule[0][k] + SCHED_SI_OFFS;
      x_ub = schedules[side][j].schedule[schedules[side][j].n_pieces][k];
      l = 0;
      fbi_offset = schedules[side][j].fbi_bounds[l];
      while (x < x_ub) {
        u16_t **b0, **b1, **b0_ub;
#ifdef ASM_SCHEDTDSIEVE2
        b0 = tdsieve_sched2buf(&x, x_ub, sieve_interval, sched_tds_buffer,
            sched_tds_buffer + SCHED_TDS_BUFSIZE − 4);
#else
        b0 = sched_tds_buffer;
        b0_ub = b0 + SCHED_TDS_BUFSIZE;
        for ( ; x + 6 < x_ub; x = x + 8) {
          if ((sieve_interval[*x] | sieve_interval[*(x+2)] | sieve_interval[*(x+4)] | sieve_interval[*(x+6)]) ≡
                0) continue;
          if (sieve_interval[x[0]] ≠ 0) *(b0++) = x;
          if (sieve_interval[x[2]] ≠ 0) *(b0++) = x + 2;
          if (sieve_interval[x[4]] ≠ 0) *(b0++) = x + 4;
          if (sieve_interval[x[6]] ≠ 0) *(b0++) = x + 6;
          if (b0 + 4 > b0_ub) goto sched_tds1;
        }
        for ( ; x < x_ub; x += 2) {
          if (sieve_interval[*x] ≠ 0) *(b0++) = x;
        }
#endif
      sched_tds1:
        for (b1 = sched_tds_buffer; b1 < b0; b1++) {
          u16_t *y;
          u32_t fbi;
          y = *(b1);
          if (schedules[side][j].schedule[l + 1][k] ≤ y) {
            do {
```

```
            l++;
            if (l ≥ schedules[side][j].n_pieces) Schlendrian("XXX\n");
          } while (schedules[side][j].schedule[l + 1][k] ≤ y);
          fbi_offset = schedules[side][j].fbi_bounds[l];
        }
        fbi = fbi_offset + *(y + 1 − 2 * SCHED_SI_OFFS);
        *(tds_fbi_curpos[sieve_interval[*y] − 1]++) = fbi;
#ifdef TDS_FB_PREFETCH
        TDS_FB_PREFETCH(FB[side] + fbi);
#endif
      }
    }
#else
    u32_t l, k;
    k = schedules[side][j].current_strip++;
    for (l = 0; l < schedules[side][j].n_pieces; l++) {
      u16_t *x, *x_ub;
      u32_t fbi_offset;
      x_ub = schedules[side][j].schedule[l + 1][k];
      fbi_offset = schedules[side][j].fbi_bounds[l];
      for (x = schedules[side][j].schedule[l][k] + SCHED_SI_OFFS; x + 6 < x_ub; x += 8) {
        unsigned char z;
        if ((sieve_interval[*x] | sieve_interval[*(x+2)] | sieve_interval[*(x+4)] | sieve_interval[*(x+6)]) ≡
              0) {
          continue;
        }
        if ((z = sieve_interval[*x]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = fbi_offset + *(x + 1 − 2 * SCHED_SI_OFFS);
        if ((z = sieve_interval[*(x + 2)]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = fbi_offset + *(x + 3 − 2 * SCHED_SI_OFFS);
        if ((z = sieve_interval[*(x + 4)]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = fbi_offset + *(x + 5 − 2 * SCHED_SI_OFFS);
        if ((z = sieve_interval[*(x + 6)]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = fbi_offset + *(x + 7 − 2 * SCHED_SI_OFFS);
      }
      while (x < x_ub) {
        unsigned char z;
        if ((z = sieve_interval[*x]) ≠ 0)
          *(tds_fbi_curpos[z − 1]++) = fbi_offset + *(x + 1 − 2 * SCHED_SI_OFFS);
        x += 2;
      }
    }
#endif
#endif
  }
}
newclock = clock();
tds3_clock[side] += newclock − last_tdclock;
last_tdclock = newclock;
```

131.

⟨ Global declarations 20 ⟩ +≡

```
#ifndef SCHED_TDS_BUFSIZE
#define SCHED_TDS_BUFSIZE 1024
#endif
  u16_t *(sched_tds_buffer[SCHED_TDS_BUFSIZE]);
```

132. So far, we have stored factor base indices. Replace them by the factor base elements they point to:

⟨ td sieve 128 ⟩ +≡

```
  {
    u32_t i;
    for (i = 0; i < UCHAR_MAX ∧ i < ncand; i++) {
      u32_t *p;
      for (p = tds_fbi[i]; p < tds_fbi_curpos[i]; p++) *p = FB[side][*p];
    }
  }
```

133.

```
⟨ td sieve 128 ⟩ +≡
  {
    u16_t *x;
#ifdef ASM_TDSLINIE
    x = smalltdsieve_auxbound;
    if (x < smallsieve_auxbound[side][4]) {
      tdslinie(x, smallsieve_auxbound[side][4], sieve_interval, tds_fbi_curpos);
      x = smallsieve_auxbound[side][4];
    }
#else
    for (x = smalltdsieve_auxbound; x < smallsieve_auxbound[side][4]; x = x + 4) {
      u32_t p, r, pr;
      unsigned char *y;
      p = x[0];
      pr = x[1];
      r = x[3];
      modulo32 = p;
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy, *yy_ub;
        yy_ub = y + n_i − 3 * p;
        yy = y + r;
        while (yy < yy_ub) {
          unsigned char o;
          o = (*yy) | (*(yy + p));
          yy += 2 * p;
          if ((o | (*yy) | (*(yy + p))) ≠ 0) {
            yy = yy − 2 * p;
            if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
            if (*(yy + p) ≠ 0) *(tds_fbi_curpos[*(yy + p) − 1]++) = p;
            yy += 2 * p;
            if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
            if (*(yy + p) ≠ 0) *(tds_fbi_curpos[*(yy + p) − 1]++) = p;
          }
          yy += 2 * p;
        }
        yy_ub += 2 * p;
        if (yy < yy_ub) {
          if (((*yy) | (*(yy + p))) ≠ 0) {
            if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
            if (*(yy + p) ≠ 0) *(tds_fbi_curpos[*(yy + p) − 1]++) = p;
          }
          yy += 2 * p;
        }
        yy_ub += p;
        if (yy < yy_ub) {
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
        }
        r = modadd32(r, pr);
      }
      x[3] = r;
```

```
    }
#endif
#ifdef ASM_TDSLINIE3
    if (x < smallsieve_auxbound[side][3]) {
      tdslinie3(x, smallsieve_auxbound[side][3], sieve_interval, tds_fbi_curpos);
      x = smallsieve_auxbound[side][3];
    }
#else
    for ( ; x < smallsieve_auxbound[side][3]; x = x + 4) {
      u32_t p, r, pr;
      unsigned char *y;

      p = x[0];
      pr = x[1];
      r = x[3];
      modulo32 = p;
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy, *yy_ub;

        yy_ub = y + n_i;
        yy = y + r;
        if (((*yy) | (*(yy + p)) | (*(yy + 2 * p))) ≠ 0) {
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
          if (*(yy + p) ≠ 0) *(tds_fbi_curpos[*(yy + p) − 1]++) = p;
          if (*(yy + 2 * p) ≠ 0) *(tds_fbi_curpos[*(yy + 2 * p) − 1]++) = p;
        }
        yy += 3 * p;
        if (yy < yy_ub) {
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
        }
        r = modadd32(r, pr);
      }
      x[3] = r;
    }
#endif
#ifdef ASM_TDSLINIE2
    if (x < smallsieve_auxbound[side][2]) {
      tdslinie2(x, smallsieve_auxbound[side][2], sieve_interval, tds_fbi_curpos);
      x = smallsieve_auxbound[side][2];
    }
#else
    for ( ; x < smallsieve_auxbound[side][2]; x = x + 4) {
      u32_t p, r, pr;
      unsigned char *y;

      p = x[0];
      pr = x[1];
      r = x[3];
      modulo32 = p;
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy, *yy_ub;

        yy_ub = y + n_i;
        yy = y + r;
        if (((*yy) | (*(yy + p))) ≠ 0) {
```

```
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
          if (*(yy + p) ≠ 0) *(tds_fbi_curpos[*(yy + p) − 1]++) = p;
        }
        yy += 2 * p;
        if (yy < yy_ub) {
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
        }
        r = modadd32(r, pr);
      }
      x[3] = r;
    }
#endif
#ifdef ASM_TDSLINIE1
    if (x < smallsieve_auxbound[side][1]) {
      tdslinie1(x, smallsieve_auxbound[side][1], sieve_interval, tds_fbi_curpos);
      x = smallsieve_auxbound[side][1];
    }
#else
    for ( ; x < smallsieve_auxbound[side][1]; x = x + 4) {
      u32_t p, r, pr;
      unsigned char *y;

      p = x[0];
      pr = x[1];
      r = x[3];
      modulo32 = p;
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy, *yy_ub;

        yy_ub = y + n_i;
        yy = y + r;
        if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
        yy += p;
        if (yy < yy_ub) {
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
        }
        r = modadd32(r, pr);
      }
      x[3] = r;
    }
#endif
#ifdef ASM_TDSLINIE0
    if (x < smallsieve_auxbound[side][0]) {
      tdslinie0(x, smallsieve_auxbound[side][0], sieve_interval, tds_fbi_curpos);
      x = smallsieve_auxbound[side][0];
    }
#else
    for ( ; x < smallsieve_auxbound[side][0]; x = x + 4) {
      u32_t p, r, pr;
      unsigned char *y;

      p = x[0];
      pr = x[1];
      r = x[3];
```

```
      modulo32 = p;
      for (y = sieve_interval; y < sieve_interval + L1_SIZE; y += n_i) {
        unsigned char *yy, *yy_ub;
        yy_ub = y + n_i;
        yy = y + r;
        if (yy < yy_ub) {
          if (*yy ≠ 0) *(tds_fbi_curpos[*yy − 1]++) = p;
        }
        r = modadd32(r, pr);
      }
      x[3] = r;
    }
#endif
    newclock = clock();
    tds1_clock[side] += newclock − last_tdclock;
    last_tdclock = newclock;
  }
```

134. Advance the location of roots which we use for trial division. But for the primes used in the td sieve, this was already done.

```
⟨ Update smallsieve_aux for TD 134 ⟩ ≡
#ifdef MMX_TD
  MMX_TdUpdate(side, j_step);
#else
  {
    u32_t i;
    u16_t *x, *y;
    y = smalltdsieve_aux[side][j_step − 1];
    for (i = 0, x = smallsieve_aux[side]; x < smallsieve_auxbound[side][0]; i++, x += 4) {
      modulo32 = x[0];
      if (modulo32 > p_bound) break;
      x[3] = modadd32((u32_t) x[3], (u32_t) y[i]);
    }
  }
#endif
  {
    u16_t *x;
    for (x = smallpsieve_aux[side]; x < smallpsieve_aux_ub[side]; x += 3) {
      modulo32 = x[0];
      x[2] = modsub32(x[2], (j_step) % modulo32);
    }
  }
```

This code is used in section 123.

135.

⟨ Calculate *sr_a* and *sr_b* 135 ⟩ ≡

```
mpz_set_si(aux1, true_i);
mpz_mul_si(aux1, aux1, a0);
mpz_set_si(aux2, a1);
mpz_mul_ui(aux2, aux2, (u32_t) true_j);
mpz_add(sr_a, aux1, aux2);
```

See also sections 136 and 137.

This code is used in section 123.

136. What we have done so far would amount to $sr\_a = a0 * true\_i + a1 * (\textbf{int})\ true\_j$; for ordinary integers.

⟨ Calculate *sr_a* and *sr_b* 135 ⟩ +≡

```
mpz_set_si(aux1, true_i);
mpz_mul_si(aux1, aux1, b0);
mpz_set_si(aux2, b1);
mpz_mul_ui(aux2, aux2, (u32_t) true_j);
mpz_add(sr_b, aux1, aux2);
```

137. Now we have also put $sr\_b = b0 * true\_i + b1 * (\textbf{int})\ true\_j$;

⟨ Calculate *sr_a* and *sr_b* 135 ⟩ +≡

```
if (mpz_sgn(sr_b) < 0) {
  mpz_neg(sr_b, sr_b);
  mpz_neg(sr_a, sr_a);
}
```

138.

⟨ Calculate value of norm polynomial in *aux1* 138 ⟩ ≡

```
{
  u32_t i;

  i = 1;
  mpz_set(aux2, sr_a);
  mpz_set(aux1, poly[side][0]);
  for ( ; ; ) {    /* CAVE: Exclude polynomials of degree zero somewhere. */
      /* aux1 = b * aux1 + poly[side][j] * aux2; */
    mpz_mul(aux1, aux1, sr_b);
    mpz_mul(aux3, aux2, poly[side][i]);
    mpz_add(aux1, aux1, aux3);
    if (++i > poldeg[side]) break;    /* aux2 *= a; */
    mpz_mul(aux2, aux2, sr_a);
  }
}
```

This code is used in section 123.

139.

```
⟨ td by sieving 139 ⟩ ≡
  {
    int np, x;
    x = fss_sv[ci];
    np = tds_fbi_curpos[x] − tds_fbi[x];
    memcpy(fbp_ptr, tds_fbi[x], np ∗ sizeof (∗fbp_ptr));
    fbp_ptr += np;
  }
```

This code is used in section 123.

140.

```
⟨ Small FB td 140 ⟩ ≡
  {
    u16_t ∗x;
#ifndef MMX_TD
#ifdef PREINVERT
    ⟨ Small td by preinversion 141 ⟩
#else
    for (x = smallsieve_aux[side]; x < smallsieve_auxbound[side][0] ∧ ∗x ≤ p_bound; x += 4) {
      u32_t p;
      p = ∗x;
      if (strip_i % p ≡ x[3]) ∗(fbp_ptr ++) = p;
    }
#endif
#else
    fbp_ptr = MMX_Td(fbp_ptr, side, strip_i);
#endif
    for (x = smallpsieve_aux[side]; x < smallpsieve_aux_ub_pow1[side]; x += 3) {
      if (x[2] ≡ 0) {
        ∗(fbp_ptr ++) = ∗x;
      }
    }
  }
```

This code is used in section 123.

141.

```
⟨ Small td by preinversion 141 ⟩ ≡
  {
    u32_t ∗p_inv;
    p_inv = smalltd_pi[side];
    for (x = smallsieve_aux[side]; x < smallsieve_auxbound[side][0] ∧ ∗x ≤ p_bound; x += 4, p_inv ++) {
      modulo32 = ∗x;
      if (((modsub32((u32_t) strip_i, (u32_t)(x[3])) ∗ (∗p_inv)) & #ffff0000) ≡ 0) {
        ∗(fbp_ptr ++) = ∗x;
      }
    }
  }
```

This code is used in section 140.

142. Special q.

```
⟨ Special q td 142 ⟩ ≡
  if (side ≡ special_q_side) {
    if (special_q < U32_MAX) *(fbp_ptr++) = special_q;
  }
```

This code is used in section 123.

143.

```
⟨ Execute TD 143 ⟩ ≡
  fbp_ptr = mpz_trialdiv(aux1, fbp_buf, fbp_ptr − fbp_buf, tds_coll[fss_sv[ci]] ≡ 0 ? "td␣error" : Λ);
```

This code is used in section 123.

144. Special q above 32 bit: it is removed once. If it divides the value of the polynomial more than once, it has to be found by the cofactorisation functions.

```
⟨ Special q 64 bit td 144 ⟩ ≡
  if (side ≡ special_q_side) {
    if (special_q ≫ 32) {
      mpz_set_ull(aux3, special_q);
      mpz_fdiv_qr(aux2, aux3, aux1, aux3);
      if (mpz_sgn(aux3))
        Schlendrian("Special␣q␣divisor␣does␣not␣divide␣value␣of␣polynomial\n");
      mpz_set(aux1, aux2);
    }
  }
```

This code is used in section 123.

145. Finally, if the candidate is a survivor, store it if this is the first trial division side. Otherwise, output it.

```
⟨rest of td 145⟩ ≡
  if (mpz_sizeinbase(aux1, 2) ≤ max_factorbits[side]) {
    n_tdsurvivors[side]++;
    if (side ≡ first_td_side) {
      if (mpz_sgn(aux1) > 0) mpz_set(td_rests[nc1], aux1);
      else mpz_neg(td_rests[nc1], aux1);
      cand[nc1++] = cand[ci];
      td_buf1[nc1] = fbp_ptr;
      nfbp = fbp_ptr − td_buf[side];
      continue;
    }
    if (mpz_sgn(aux1) < 0) mpz_neg(aux1, aux1);
#if TDS_MPQS ≡ TDS_IMMEDIATELY
    output_tdsurvivor(td_buf1[ci], td_buf1[ci + 1], fbp_buf, fbp_ptr, td_rests[ci], aux1);
#else
#if TDS_PRIMALITY_TEST ≡ TDS_IMMEDIATELY
    mpz_set(large_factors[first_td_side], td_rests[ci]);
    mpz_set(large_factors[1 − first_td_side], aux1);
    if (primality_tests() ≡ 1) {
      store_tdsurvivor(td_buf1[ci], td_buf1[ci + 1], fbp_buf, fbp_ptr, large_factors[first_td_side],
          large_factors[1 − first_td_side]);
    }
#else
    store_tdsurvivor(td_buf1[ci], td_buf1[ci + 1], fbp_buf, fbp_ptr, td_rests[ci], aux1);
#endif    /* primality test now ? */
#endif    /* MPQS now ? */
  }
  else continue;
```

This code is used in section 123.

146.

```
⟨Trial division declarations 121⟩ +≡
  static mpz_t td_rests[L1_SIZE];
  static mpz_t large_factors[2], *(large_primes[2]);
  static mpz_t FBb_sq[2], FBb_cu[2];    /* Square and cube of factor base bound. */
```

147.

⟨ TD Init 122 ⟩ +≡

```
{
    u32_t s, i;
    for (i = 0; i < L1_SIZE; i++) {
        mpz_init(td_rests[i]);
    }
    for (s = 0; s < 2; s++) {
        mpz_init(large_factors[s]);
        large_primes[s] = xmalloc(max_factorbits[s] * sizeof (*(large_primes[s])));
        for (i = 0; i < max_factorbits[s]; i++) {
            mpz_init(large_primes[s][i]);
        }
#if 0
        mpz_init_set_d(FBb_sq[s], FB_bound[s]);
        mpz_mul(FBb_sq[s], FBb_sq[s], FBb_sq[s]);
#else
        mpz_init_set_d(FBb_cu[s], FB_bound[s]);
        mpz_init(FBb_sq[s]);
        mpz_mul(FBb_sq[s], FBb_cu[s], FBb_cu[s]);
        mpz_mul(FBb_cu[s], FBb_cu[s], FBb_sq[s]);
#endif
    }
}
```

148.

⟨ Global declarations 20 ⟩ +≡

```
u32_t *mpz_trialdiv(mpz_t N, u32_t *pbuf, u32_t ncp, char *errmsg);
```

149.

```
#ifndef ASM_MPZ_TD
  static mpz_t mpz_td_aux;
  static u32_t initialized = 0;
  u32_t *mpz_trialdiv(mpz_t N, u32_t *pbuf, u32_t ncp, char *errmsg)
  {
    u32_t np, np1, i, e2;
    if (initialized ≡ 0) {
      mpz_init(mpz_td_aux);
      initialized = 1;
    }
    e2 = 0;
    while ((mpz_get_ui(N) % 2) ≡ 0) {
      mpz_fdiv_q_2exp(N, N, 1);
      e2++;
    }
    if (errmsg ≠ Λ) {
      for (i = 0, np = 0; i < ncp; i++) {
        if (mpz_fdiv_q_ui(N, N, pbuf[i]) ≠ 0)
          Schlendrian("%s␣:␣%u␣does␣not␣divide\n", errmsg, pbuf[i]);
        pbuf[np++] = pbuf[i];
      }
    }
    else {
      for (i = 0, np = 0; i < ncp; i++) {
        if (mpz_fdiv_q_ui(mpz_td_aux, N, pbuf[i]) ≡ 0) {
          mpz_set(N, mpz_td_aux);
          pbuf[np++] = pbuf[i];
        }
      }
    }
    np1 = np;
    for (i = 0; i < np1; i++) {
      while (mpz_fdiv_q_ui(mpz_td_aux, N, pbuf[i]) ≡ 0) {
        mpz_set(N, mpz_td_aux);
        pbuf[np++] = pbuf[i];
      }
    }
    for (i = 0; i < e2; i++) pbuf[np++] = 2;
    return pbuf + np;
  }
#endif
```

150. Primality tests, mpqs, and output of candidates.

⟨ Global declarations 20 ⟩ +≡

```
static void output_tdsurvivor(u32_t *, u32_t *, u32_t *, u32_t *, mpz_t, mpz_t);
static void store_tdsurvivor(u32_t *, u32_t *, u32_t *, u32_t *, mpz_t, mpz_t);
static int primality_tests(void);
static void primality_tests_all(void);
static void output_all_tdsurvivors(void);
static u32_t *tds_fbp_buffer;
static i64_t*tds_ab;
static mpz_t *tds_lp;
static size_t max_tds = 0, *tds_fbp, tds_fbp_alloc = 0, total_ntds = 0;
#define MAX_TDS_INCREMENT 1024
#define TDS_FBP_ALLOC_INCREMENT 8192
```

151.

```
static void store_tdsurvivor(fbp_buf0, fbp_buf0_ub, fbp_buf1, fbp_buf1_ub, lf0, lf1)
    u32_t *fbp_buf0, *fbp_buf1, *fbp_buf0_ub, *fbp_buf1_ub;
    mpz_t lf0, lf1;
{
  size_t n0, n1, n;
  ⟨ Check max_tds 152 ⟩
  if (mpz_sizeinbase(lf0, 2) > max_factorbits[first_td_side] ∨ mpz_sizeinbase(lf1,
        2) > max_factorbits[1 − first_td_side]) {
    fprintf(stderr, "large␣lp␣in␣store_tdsurvivor\n");
    return;
  }
  mpz_set(tds_lp[2 * total_ntds], lf0);
  mpz_set(tds_lp[2 * total_ntds + 1], lf1);
  n0 = fbp_buf0_ub − fbp_buf0;
  n1 = fbp_buf1_ub − fbp_buf1;
  n = tds_fbp[2 * total_ntds];
  ⟨ Check tds_fbp_alloc 153 ⟩;
  memcpy(tds_fbp_buffer + n, fbp_buf0, n0 * sizeof (*fbp_buf0));
  n += n0;
  tds_fbp[2 * total_ntds + 1] = n;
  memcpy(tds_fbp_buffer + n, fbp_buf1, n1 * sizeof (*fbp_buf1));
  tds_fbp[2 * total_ntds + 2] = n + n1;
  tds_ab[2 * total_ntds] = mpz_get_sll(sr_a);
  tds_ab[2 * total_ntds + 1] = mpz_get_sll(sr_b);
  total_ntds++;
}
```

152.

⟨ Check *max_tds* 152 ⟩ ≡

```
if (total_ntds ≥ max_tds) {
  size_t i;
  if (max_tds ≡ 0) {
    tds_fbp = xmalloc((2 * MAX_TDS_INCREMENT + 1) * sizeof (*tds_fbp));
    tds_fbp[0] = 0;
    tds_ab = xmalloc(2 * MAX_TDS_INCREMENT * sizeof (*tds_ab));
    tds_lp = xmalloc(2 * MAX_TDS_INCREMENT * sizeof (*tds_lp));
  }
  else {
    tds_fbp = xrealloc(tds_fbp, (2 * (MAX_TDS_INCREMENT + max_tds) + 1) * sizeof (*tds_fbp));
    tds_ab = xrealloc(tds_ab, 2 * (MAX_TDS_INCREMENT + max_tds) * sizeof (*tds_ab));
    tds_lp = xrealloc(tds_lp, 2 * (MAX_TDS_INCREMENT + max_tds) * sizeof (*tds_lp));
  }
  for (i = 2 * total_ntds; i < 2 * (MAX_TDS_INCREMENT + max_tds); i++) mpz_init(tds_lp[i]);
  max_tds += MAX_TDS_INCREMENT;
}
```

This code is used in section 151.

153.

⟨ Check *tds_fbp_alloc* 153 ⟩ ≡

```
if (n + n0 + n1 > tds_fbp_alloc) {
  size_t a;
  a = tds_fbp_alloc;
  while (a < n + n0 + n1) a += TDS_FBP_ALLOC_INCREMENT;
  if (tds_fbp_alloc ≡ 0) tds_fbp_buffer = xmalloc(a * sizeof (*tds_fbp_buffer));
  else tds_fbp_buffer = xrealloc(tds_fbp_buffer, a * sizeof (*tds_fbp_buffer));
  tds_fbp_alloc = a;
}
```

This code is used in section 151.

154.

```
static int primality_tests()
{
  int s;
  int need_test[2];
  size_t nbit[2];
  for (s = 0; s < 2; s++) {
    size_t nb;
    need_test[s] = 0;
    nb = mpz_sizeinbase(large_factors[s], 2);
    nbit[s] = nb;
    if (nb ≤ max_primebits[s]) {
      nbit[s] = 0;
      continue;
    }
    if (mpz_cmp(large_factors[s], FBb_sq[s]) < 0) return 0;
    if (nb ≤ 2 * max_primebits[s]) {
      need_test[s] = 1;
      continue;
    }
    if (mpz_cmp(large_factors[s], FBb_cu[s]) < 0) return 0;
    need_test[s] = 1;
  }
  if (strat.stindex[nbit[0]][nbit[1]] ≡ 0) {
    n_abort2++;
    return 0;
  }
  for (s = 0; s < 2; s++) {
    i16_t is_prime;
    u16_t s1;
    s1 = s ⊕ first_psp_side;
    if (¬need_test[s1]) continue;
    n_psp++;
    if (psp(large_factors[s1], 1) ≡ 1) return 0;
    mpz_neg(large_factors[s1], large_factors[s1]);
  }
  return 1;
}
```

155.

```
#if (TDS_PRIMALITY_TEST ≠ TDS_IMMEDIATELY) ∧ (TDS_PRIMALITY_TEST ≠ TDS_MPQS)
  static void primality_tests_all( )
  {
    size_t i, j;
    for (i = 0, j = 0; i < total_ntds; i++) {
      mpz_set(large_factors[first_td_side], tds_lp[2 * i]);
      mpz_set(large_factors[1 − first_td_side], tds_lp[2 * i + 1]);
      if (primality_tests( ) ≡ 0) continue;
      mpz_set(tds_lp[2 * j], large_factors[first_td_side]);
      mpz_set(tds_lp[2 * j + 1], large_factors[1 − first_td_side]);
      tds_fbp[2 * j + 1] = tds_fbp[2 * i + 1];
      tds_fbp[2 * j + 2] = tds_fbp[2 * i + 2];
      tds_ab[2 * j] = tds_ab[2 * i];
      tds_ab[2 * j + 1] = tds_ab[2 * i + 1];
      j++;
    }
    total_ntds = j;
  }
#endif
```

156.

```
#if TDS_MPQS ≠ TDS_IMMEDIATELY
  static void output_all_tdsurvivors( )
  {
    size_t i;
    for (i = 0; i < total_ntds; i++) {
      mpz_set_sll(sr_a, tds_ab[2 * i]);
      mpz_set_sll(sr_b, tds_ab[2 * i + 1]);
      output_tdsurvivor(tds_fbp_buffer + tds_fbp[2 * i], tds_fbp_buffer + tds_fbp[2 * i + 1],
            tds_fbp_buffer + tds_fbp[2 * i + 1], tds_fbp_buffer + tds_fbp[2 * i + 2], tds_lp[2 * i], tds_lp[2 * i + 1]);
    }
    total_ntds = 0;
  }
#endif
```

157.

```
  static void output_tdsurvivor(fbp_buf0, fbp_buf0_ub, fbp_buf1, fbp_buf1_ub, lf0, lf1)
      u32_t *fbp_buf0, *fbp_buf1, *fbp_buf0_ub, *fbp_buf1_ub;
      mpz_t lf0, lf1;
  {
    u32_t s, *(fbp_buffers[2]), *(fbp_buffers_ub[2]);
    u32_t nlp[2];
    clock_t cl;
    int cferr;
    s = first_td_side;
    fbp_buffers[s] = fbp_buf0;
    fbp_buffers_ub[s] = fbp_buf0_ub;
    fbp_buffers[1 − s] = fbp_buf1;
    fbp_buffers_ub[1 − s] = fbp_buf1_ub;
    mpz_set(large_factors[s], lf0);
    mpz_set(large_factors[1 − s], lf1);
#if TDS_PRIMALITY_TEST ≡ TDS_MPQS
    if (primality_tests( ) ≡ 0) return;
#endif
    cl = clock( );
    n_cof ++;
#if 1
    cferr = cofactorisation(&strat, large_primes, large_factors, max_primebits, nlp, FBb_sq, FBb_cu);
    mpqs_clock += clock( ) − cl;
    if (cferr < 0) {
      fprintf(stderr, "cofactorisation␣failed␣for␣");
      mpz_out_str(stderr, 10, large_factors[0]);
      fprintf(stderr, ",");
      mpz_out_str(stderr, 10, large_factors[1]);
      fprintf(stderr, "␣(a,b):␣");
      mpz_out_str(stderr, 10, sr_a);
      fprintf(stderr, "␣");
      mpz_out_str(stderr, 10, sr_b);
      fprintf(stderr, "\n");
      n_mpqsfail[0]++;
    }
    if (cferr) return;
#else
    for (s = 0; s < 2; s++) {
      u16_t s1;
      i32_t i, nf;
      mpz_t *mf;
      s1 = s ⊕ first_mpqs_side;
      if (mpz_sgn(large_factors[s1]) > 0) {
        if (mpz_cmp_ui(large_factors[s1], 1) ≡ 0) nlp[s1] = 0;
        else {
          nlp[s1] = 1;
          mpz_set(large_primes[s1][0], large_factors[s1]);
        }
        continue;
      }
```

```
        mpz_neg(large_factors[s1], large_factors[s1]);
        if (mpz_sizeinbase(large_factors[s1], 2) > 96)
#if 0
          nf = mpqs3_factor(large_factors[s1], max_primebits[s1], &mf);
#else
        nf = −1;
#endif
        else nf = mpqs_factor(large_factors[s1], max_primebits[s1], &mf);
        if (nf < 0) {
          fprintf(stderr, "mpqs␣failed␣for␣");
          mpz_out_str(stderr, 10, large_factors[s1]);
          fprintf(stderr, "(a,b):␣");
          mpz_out_str(stderr, 10, sr_a);
          fprintf(stderr, "␣");
          mpz_out_str(stderr, 10, sr_b);
          fprintf(stderr, "\n");
          n_mpqsfail[s1]++;
          break;
        }
        if (nf ≡ 0) {      /* One factor exceeded bit limit. */
          n_mpqsvain[s1]++;
          break;
        }
        for (i = 0; i < nf; i++) mpz_set(large_primes[s1][i], mf[i]);
        nlp[s1] = nf;
      }
      mpqs_clock += clock() − cl;
      if (s ≠ 2) return;
#endif
      yield++;
#ifdef OFMT_CWI
#define CWI_LPB #100000
#define OBASE 10
      {
        u32_t nlp_char[2];
        for (s = 0; s < 2; s++) {
          u32_t *x, nlp1;
          for (x = fbp_buffers[s], nlp1 = nlp[s]; x < fbp_buffers_ub[s]; x++)
            if (*x > CWI_LPB) nlp1++;
          if ((nlp_char[s] = u32_t2cwi(nlp1)) ≡ '\0') break;
        }
        if (s ≡ 0) {
          errprintf("Conversion␣to␣CWI␣format␣failed\n");
          continue;
        }
#ifdef OFMT_CWI_REVERSE
        fprintf(ofile, "01%c%c␣", nlp_char[1], nlp_char[0]);
#else
        fprintf(ofile, "01%c%c␣", nlp_char[0], nlp_char[1]);
#endif
      }
#else
```

```
    fprintf(ofile, "W␣");
#define OBASE 16
#endif
    mpz_out_str(ofile, OBASE, sr_a);
    fprintf(ofile, "␣");
    mpz_out_str(ofile, OBASE, sr_b);
#ifndef OFMT_CWI_REVERSE
    for (s = 0; s < 2; s++) {
      u32_t i, *x;
#ifndef OFMT_CWI
      fprintf(ofile, "\n%c", 'X' + s);
#endif
      if (s ≡ special_q_side) {
        if (special_q ≫ 32)
#ifndef OFMT_CWI
          fprintf(ofile, "␣%llX", special_q);
#else     /* CAVE: not tested */
        if (special_q > CWI_LPB) fprintf(ofile, "␣%llu", special_q);
#endif
      }
      for (i = 0; i < nlp[s]; i++) {
        fprintf(ofile, "␣");
        mpz_out_str(ofile, OBASE, large_primes[s][i]);
      }
      for (x = fbp_buffers[s]; x < fbp_buffers_ub[s]; x++) {
#ifndef OFMT_CWI
        fprintf(ofile, "␣%X", *x);
#else
        if (*x > CWI_LPB) fprintf(ofile, "␣%d", *x);
#endif
      }
    }
#else
    for (s = 0; s < 2; s++) {
      u32_t i, *x;
      for (i = 0; i < nlp[1 - s]; i++) {
        fprintf(ofile, "␣");
        mpz_out_str(ofile, OBASE, large_primes[1 - s][i]);
      }
      for (x = fbp_buffers[1 - s]; x < fbp_buffers_ub[1 - s]; x++) {
        if (*x > CWI_LPB) fprintf(ofile, "␣%d", *x);
      }
      if (1 - s ≡ special_q_side) {
        if (special_q ≫ 32)
          if (special_q > CWI_LPB) fprintf(ofile, "␣%llu", special_q);
      }
    }
#endif
#ifndef OFMT_CWI
    fprintf(ofile, "\n");
#else
    fprintf(ofile, ";\n");
```

```
#endif
  }
```

158.

```
⟨ Global declarations 20 ⟩ +≡
#if 0
#define OFMT_CWI
#endif
#ifdef OFMT_CWI
  static char u32_t2cwi(u32_t);
#endif
```

159.

```
#ifdef OFMT_CWI
  static char u32_t2cwi(u32_t n)
  {
    if (n < 10) return '0' + n;
    n = n - 10;
    if (n < 26) return 'A' + n;
    n = n - 26;
    if (n < 26) return 'a' + n;
    return '\0';
  }
#endif
```

160.

```
#ifdef DEBUG
  int mpout(mpz_t X)
  {
    mpz_out_str(stdout, 10, X);
    puts("");
    return 1;
  }
#endif
```

161.

```
⟨ Global declarations 20 ⟩ +≡
  void dumpsieve(u32_t j_offset, u32_t side);
```

162.

```
void dumpsieve(u32_t j_offset, u32_t side)
{
  FILE *ofile;
  char *ofn;
  asprintf(&ofn, "sdump4e.ot%u.j%u.s%u", oddness_type, j_offset, side);
  if ((ofile = fopen(ofn, "w")) ≡ Λ) {
    free(ofn);
    return;
  }
  fwrite(sieve_interval, 1, L1_SIZE, ofile);
  fclose(ofile);
  free(ofn);
  asprintf(&ofn, "hzsdump4e.ot%u.j%u.s%u", oddness_type, j_offset, side);
  if ((ofile = fopen(ofn, "w")) ≡ Λ) {
    free(ofn);
    return;
  }
  fwrite(horizontal_sievesums, 1, j_per_strip, ofile);
  fclose(ofile);
  free(ofn);
}
```

GNFS-LASIEVE4E