```c
/*
** Author: James Pate Williams, Jr. (c) 1997 - 2024
** Single precision Goldwasser-Kilian primality test.
** See "A Course in Computational Algebraic Number
** Theory" by Henri Cohen Algorithm 9.2.3 page 470.
 */

#include <windows.h>
#include <tchar.h>
#include <strsafe.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define B1 2000000
#define B2 100000000
#define SIEVE_SIZE (B2 / 32 + 1)
#define BITS_PER_LONG  32
#define BITS_PER_LONG_1 31
#define SIEVE_COUNT1 148933
#define SIEVE_LIMIT1 1999853
#define SIEVE_COUNT2 5761455
#define SIEVE_LIMIT2 99999989
#define MAX_THREADS 1
#define BUF_SIZE 255

typedef struct tagPrimeExponent {
    long long prime;
    int exponent;
} PRIMEEXPONENT, * PPRIMEEXPONENT;

typedef struct tagECPoint {
    long long x, y;
} ECPOINT, * PECPOINT;

/* global data */

long prime;
long b2[SIEVE_COUNT2];
long d2[SIEVE_COUNT2];
long p1[SIEVE_COUNT1];
long p2[SIEVE_COUNT2];
long sieve[SIEVE_SIZE];
PRIMEEXPONENT pev[128];

long GetBit(long i, long* sieve) {
    long b = (int)(i % BITS_PER_LONG);
    long c = (int)(i / BITS_PER_LONG);

    return (sieve[c] >> (BITS_PER_LONG_1 - b)) & 1L;
```

```
 53  }
 54
 55  void SetBit(long i, long v, long* sieve) {
 56      long b = (int)(i % BITS_PER_LONG);
 57      long c = (int)(i / BITS_PER_LONG);
 58      long mask = 1L << (BITS_PER_LONG_1 - b);
 59
 60      if (v == 1)
 61          sieve[c] |= mask;
 62      else
 63          sieve[c] &= ~mask;
 64  }
 65
 66  void Sieve(long n) {
 67      long c, i, inc;
 68
 69      SetBit(0, 0, sieve);
 70      SetBit(1, 0, sieve);
 71      SetBit(2, 1, sieve);
 72      for (i = 3; i <= n; i++)
 73          SetBit(i, i & 1, sieve);
 74      c = 3;
 75      do {
 76          i = c * c, inc = c + c;
 77          while (i <= n) {
 78              SetBit(i, 0, sieve);
 79              i += inc;
 80          }
 81          c += 2;
 82          while (!GetBit(c, sieve)) c++;
 83      } while (c * c <= n);
 84  }
 85
 86  long NextPrime(long bound) {
 87      long q = 0;
 88      while (q == 0 && prime <= bound) {
 89          if (GetBit(prime, sieve))
 90              q = prime;
 91          prime++;
 92      }
 93      return q;
 94  }
 95
 96  long long GCD(long long a, long long b) {
 97      /* returns greatest common divisor of a and b */
 98      long long r;
 99
100      if (a < 0) a = -a;
101      if (b < 0) b = -b;
102      while (b > 0)
103          r = a % b, a = b, b = r;
104      return a;
```

```
105 }
106
107 long long ExtendedEuclidean(long long b, long long n)
108 {
109     long long b0 = b, n0 = n, t = 1, t0 = 0, temp, q, r;
110
111     q = n0 / b0;
112     r = n0 - q * b0;
113
114     while (r > 0)
115     {
116         temp = t0 - q * t;
117         if (temp >= 0) temp = temp % n;
118         else temp = n - (-temp % n);
119         t0 = t;
120         t = temp;
121         n0 = b0;
122         b0 = r;
123         q = n0 / b0;
124         r = n0 - q * b0;
125     }
126     if (b0 != 1) return 0;
127     else return t % n;
128 }
129
130 void Add(
131     long long a, long long p, ECPOINT P,
132     ECPOINT Q, ECPOINT* R) {
133     /* elliptic curve point partial addition */
134     long long i, lambda;
135
136     if (P.x == Q.x && P.y == 0 && Q.y == 0)
137     {
138         R->x = 0;
139         R->y = 1;
140         return;
141     }
142     if (P.x == Q.x && P.y == p - Q.y)
143     {
144         R->x = 0;
145         R->y = 1;
146         return;
147     }
148     if (P.x == 0 && P.y == 1)
149     {
150         *R = Q;
151         return;
152     }
153     if (Q.x == 0 && Q.y == 1)
154     {
155         *R = P;
156         return;
```

```
157     }
158     if (P.x != Q.x)
159     {
160         i = Q.x - P.x;
161         if (i < 0) i += p;
162         i = ExtendedEuclidean(i, p);
163         lambda = ((Q.y - P.y) * i) % p;
164     }
165     else
166     {
167         i = ExtendedEuclidean((2 * P.y) % p, p);
168         lambda = ((((3 * P.x * P.x) % p) + a) * i) % p;
169     }
170     if (lambda < 0) lambda += p;
171     R->x = ((lambda * lambda) % p - P.x - Q.x) % p;
172     R->y = ((lambda * (P.x - R->x)) % p - P.y) % p;
173     if (R->x < 0) R->x += p;
174     if (R->y < 0) R->y += p;
175 }
176
177 void Multiply(
178     long long a,
179     long long k,
180     long long p,
181     ECPOINT P,
182     ECPOINT* R)
183 {
184     ECPOINT S = { 0 }, T = { 0 };
185
186     R->x = 0;
187     R->y = 1;
188     S = P;
189
190     while (k != 0)
191     {
192         if (k & 1)
193         {
194             Add(a, p, *R, S, &T);
195             *R = T;
196         }
197
198         k >>= 1;
199
200         if (k != 0)
201         {
202             Add(a, p, S, S, &T);
203             S = T;
204         }
205     }
206 }
207
208 long long Jacobi(long long a, long long n)
```

```
209 {
210     long long a1, e = 0, m = 0, n1 = 0, s = 0;
211 
212     if (a == 0) return 0;
213     if (a == 1) return 1;
214     while ((a & 1) == 0)
215         a >>= 1, e++;
216     a1 = a;
217     m = n % 8;
218     if (!(e & 1)) s = 1;
219     else if (m == 1 || m == 7) s = +1;
220     else if (m == 3 || m == 5) s = -1;
221     if (n % 4 == 3 && a1 % 4 == 3) s = -s;
222     n1 = n % a1;
223     return s * Jacobi(n1, a1);
224 }
225 
226 long long Pow(long long x, long long b)
227 {
228     /* returns x ^ b */
229     int bits[64] = { 0 }, i = 0, l = 0;
230     long long bb = b, z = 1;
231 
232     while (bb != 0) {
233         bits[l++] = (int)(bb & 1);
234         bb >>= 1;
235     }
236 
237     for (i = l - 1; i >= 0; i--) {
238         z = z * z;
239         if (bits[i] == 1) z = z * x;
240     }
241 
242     return z;
243 }
244 
245 long long PowMod(long long x, long long b, long long n)
246 {
247     /* returns x ^ b mod n
248     ** algorithm taken from Douglas R. Stinson
249     ** "Cryptography Theory and Practice" 1995
250     ** Figure 4.4 page 127
251     **/
252     long long bits[64] = { 0 }, i = 0, l = 0;
253     long long bb = b, z = 1;
254 
255     while (bb != 0) {
256         bits[l++] = (int)(bb & 1);
257         bb >>= 1;
258     }
259 
260     for (i = l - 1; i >= 0; i--) {
```

```
261         z = (z * z) % n;
262         if (bits[i] == 1) z = (z * x) % n;
263     }
264
265     if (z < 0)
266         z += n;
267
268     return z;
269 }
270
271 long long Rand64()
272 {
273     long long hi = ((long long)rand()) << 32;
274     long long lo = (long long)rand();
275     return hi | lo;
276 }
277
278 long long RandomRange(long long lower, long long upper)
279 {
280     long long a = 0;
281
282     while (1) {
283         a = lower + Rand64() % (upper + 1);
284
285         if (a <= upper)
286             return a;
287     }
288 }
289
290 long long RabinMiller(long long N) {
291     /* given an integer N >= 3 returns 0 composite */
292     /* 1 probably prime */
293     long long N1 = N - 1, a, b, c = 20, e, q = N1, t = 0;
294
295     if (N == 2 || N == 3) return 1;
296     /* N - 1 = 2 ^ t * q with q odd */
297     while (!(q & 1)) q >>= 1, t++;
298 L2:
299     a = RandomRange(2, N - 1);
300     e = 0;
301     b = PowMod(a, q, N);
302     if (b == 1) goto L4;
303     while (b != 1 && b != N1 && e <= t - 2) {
304         b = (b * b) % N;
305         e++;
306     }
307     if (b != N1) return 0LL;
308 L4:
309     c--;
310     if (c > 0) goto L2;
311     return 1LL;
312 }
```

```
313
314 long long A[10000], B[10000];
315
316 void DoSingletonsSort(long long a[], int ii, int jj) {
317     /* see "Sorting and Sort Systems"
318     ** by Harold Lorin (IBM book)
319     */
320     int m = 0;
321     int i = ii;
322     int j = jj;
323     int iu[50] = { 0 };
324     int il[50] = { 0 };
325     int ij = 0, l = 1, k = 0;
326     long long t = { 0 }, tt = { 0 };
327 Label1:
328     if (i >= j)
329         goto Label8;
330     goto Label2;
331 Label2:
332     k = i;
333     ij = (j + i) / 2;
334     t = a[ij];
335     if (a[i] <= t)
336         goto Label3;
337     a[ij] = a[i];
338     a[i] = t;
339     t = a[ij];
340     goto Label3;
341 Label3:
342     l = j;
343     if (a[j] >= t)
344         goto Label5;
345     a[ij] = a[j];
346     a[j] = t;
347     t = a[ij];
348     if (a[i] <= t)
349         goto Label5;
350     a[ij] = a[i];
351     a[i] = t;
352     t = a[ij];
353     goto Label5;
354 Label4:
355     a[l] = a[k];
356     a[k] = tt;
357     goto Label5;
358 Label5:
359     l--;
360     if (a[l] > t)
361         goto Label5;
362     tt = a[l];
363     goto Label6;
364 Label6:
```

```
365      k++;
366      if (a[k] < t)
367          goto Label6;
368      if (k <= l)
369          goto Label4;
370      if (l - i <= j - k)
371          goto Label7;
372      il[m] = i;
373      iu[m] = l;
374      i = k;
375      m++;
376      goto Label9;
377  Label7:
378      il[m] = k;
379      iu[m] = j;
380      j = l;
381      m++;
382      goto Label9;
383  Label8:
384      m--;
385      if (m == -1)
386          return;
387      i = il[m];
388      j = iu[m];
389      goto Label9;
390  Label9:
391      if (j - i > 10)
392          goto Label2;
393      if (i == ii)
394          goto Label1;
395      i--;
396      goto Label10;
397  Label10:
398      i++;
399      if (i == j)
400          goto Label8;
401      t = a[i + 1];
402      if (a[i] <= t)
403          goto Label10;
404      k = i;
405      goto Label11;
406  Label11:
407      a[k + 1] = a[k];
408      k--;
409      if (t < a[k])
410          goto Label11;
411      a[k + 1] = t;
412      goto Label10;
413  }
414
415  void SingletonsSort(long long a[], int n) {
416      DoSingletonsSort(a, 0, n - 1);
```

```
417 }
418
419 long long SquareRootModPrime(
420     long long a, long long p) {
421     /* returns square root of a modulo p if it exists 0 otherwise */
422     int i;
423     long long ainv = 0, b = 0, c, d, p1 = p - 1, q = 0, r = 0;
424     long long s = 0, t = p1;
425
426     if (Jacobi(a, p) == -1)
427         return 0;
428
429     a %= p;
430
431     if (a < 0)
432         a += p;
433
434     /* find generator */
435     do
436         b = RandomRange(1, p1);
437     while (Jacobi(b, p) != -1);
438
439     /* p - 1 = 2 ^ s * t with t odd */
440     while (!(t & 1)) {
441         t >>= 1;
442         s++;
443     }
444
445     ainv = ExtendedEuclidean(a, p);
446     if ((ainv * a) % p != 1)
447         exit(-512);
448     c = PowMod(b, t, p);
449     r = PowMod(a, (t + 1) / 2, p);
450
451     for (i = 1; i <= s - 1; i++) {
452         long long e = s - i - 1;
453         long long f = PowMod(2, e, p);
454         d = PowMod((((r * r) % p) * ainv) % p, f, p);
455         if (d == p1)
456             r = (r * c) % p;
457         c = (c * c) % p;
458     }
459
460     return r;
461 }
462
463 long long Legendre(
464     long long x, long long a, long long b, long long p)
465 {
466     long long y = ((((x * x) % p) * x) % p + (a * x) % p + b) % p;
467     long long l = PowMod(y, (p - 1) / 2, p);
468
```

```
469     if (l == p - 1)
470         l = -1;
471
472     return l;
473 }
474
475 long long Cardinality(
476     long long a, long long b, long long p)
477 {
478     long long a_p = 0, x;
479
480     for (x = 0; x < p; x++)
481         a_p += Legendre(x, a, b, p);
482     return p + 1 + a_p;
483 }
484
485 int Shanks(
486     long long a, long long p,
487     long long* beta, long long* gamma,
488     ECPOINT P, ECPOINT* Q) {
489     int aCount = 0, bCount = 0;
490     long long W = (long long)ceil(sqrt(2.0) *
491         pow((double)p, 0.25));
492
493     /* baby steps */
494
495     for (*beta = 0; *beta < W; (*beta)++) {
496         long long m = (p + 1 + *beta) % p;
497         ECPOINT R;
498
499         Multiply(a, m, p, P, &R);
500
501         if (R.x != 0)
502             A[aCount++] = R.x;
503     }
504
505     /* giant steps */
506
507     for (*gamma = 0; *gamma <= W; (*gamma)++)
508     {
509         long long m = (*gamma * W) % p;
510         ECPOINT R;
511
512         Multiply(a, m, p, P, &R);
513
514         if (R.x != 0)
515             B[bCount++] = R.x;
516     }
517
518     SingletonsSort(A, aCount);
519     SingletonsSort(B, bCount);
520
```

```
521     for (int i = 0; i < aCount; i++)
522     {
523         *beta = A[i];
524 
525         for (int j = 0; j < bCount; j++)
526         {
527             *gamma = B[j];
528 
529             if (*beta == *gamma)
530             {
531                 *beta = i;
532                 *gamma = j;
533                 return 1;
534             }
535         }
536     }
537 
538     return 0;
539 }
540 
541 long long ShanksMestre(
542     long long a, long long b, long long p)
543 {
544     if (p <= 229)
545         return -1;
546 
547     int counter = 0;
548 
549     while (counter < 2048 * 2048) {
550         long long g = -1, c = 0, d = 0, g2 = 0, g3 = 0, sigma = 0;
551         long long x = -1, y = -1;
552         long long beta = -1, gamma = -1, ea = -1, m, r, t, u;
553         long long W = (long long)ceil(sqrt(2.0) *
554             pow((double)p, 0.25));
555         ECPOINT P = { 0 }, Q = { 0 }, R = { 0 };
556 
557         for (g = 1; g < p; g++)
558             if (Legendre(g, a, b, p) == -1)
559                 break;
560 
561         g2 = (g * g) % p;
562         g3 = (g * g2) % p;
563         c = (g2 * a) % p;
564         d = (g3 * b) % p;
565 
566         while (x < p)
567         {
568             while (x < p)
569             {
570                 x = RandomRange(2, p - 1);
571                 sigma = Legendre(x, a, b, p);
572 
```

```
573                 if (sigma != 0)
574                     break;
575             }
576 
577             if (sigma == 1)
578             {
579                 ea = a;
580                 y = ((((x * x) % p) * x) % p + (a * x) % p + b) % p;
581             }
582             else
583             {
584                 ea = c;
585                 x = (x * g) % p;
586                 y = ((((x * x) % p) * x) % p + (c * x) % p + d) % p;
587             }
588 
589             P.x = x;
590             P.y = SquareRootModPrime(y, p);
591 
592             if (P.y != 0)
593             {
594                 if (Shanks(ea, p, &beta, &gamma, P, &Q))
595                     break;
596             }
597         }
598 
599         if (x == p)
600             return -1;
601 
602         r = -1;
603         t = beta + gamma * W;
604         m = p + 1 + t;
605         Multiply(ea, m, p, P, &Q);
606         Multiply(a, m, p, P, &R);
607 
608         if ((ea == a && Q.x == 0 && Q.y == 1) ||
609             (ea != a && Q.x == 0 && Q.y == 1 &&
610                 !(R.x == 0 && R.y == 1)))
611             r = p + 1 + sigma * t;
612 
613         else
614         {
615             t = -beta - gamma * W;
616             m = p + 1 + t;
617             Multiply(ea, m, p, P, &Q);
618             Multiply(a, m, p, P, &R);
619 
620             if ((ea == a && Q.x == 0 && Q.y == 1) ||
621                 (ea != a && Q.x == 0 && Q.y == 1 &&
622                     !(R.x == 0 && R.y == 1)))
623                 r = p + 1 + sigma * t;
624 
```

```
625             else
626             {
627                 u = beta - gamma * W;
628                 m = p + 1 + u;
629                 Multiply(ea, m, p, P, &Q);
630                 Multiply(a, m, p, P, &R);
631
632                 if ((ea == a && Q.x == 0 && Q.y == 1) ||
633                     (ea != a && Q.x == 0 && Q.y == 1 &&
634                         !(R.x == 0 && R.y == 1)))
635                     r = p + 1 + sigma * u;
636
637                 else
638                 {
639                     u = -beta + gamma * W;
640                     m = p + 1 + u;
641                     Multiply(ea, m, p, P, &Q);
642                     Multiply(a, m, p, P, &R);
643
644                     if ((ea == a && Q.x == 0 && Q.y == 1) ||
645                         (ea != a && Q.x == 0 && Q.y == 1 &&
646                             !(R.x == 0 && R.y == 1)))
647                         r = p + 1 + sigma * u;
648                 }
649             }
650         }
651
652         if (r != -1)
653             return r;
654
655         counter++;
656     }
657
658     return -1;
659 }
660
661 long long BrentPollardRho(
662     long long N, long long limit) {
663     long c = 0, i = 0;
664     long long k = 1, l = 1, P = 1;
665     long long x = 2, y = 2, x1 = 2, g = 0;
666 L2:
667     x = (x * x + 1) % N;
668     P = (P * (x1 - x)) % N;
669
670     if (P < 0)
671         P += N;
672
673     c++;
674
675     if (c == 20) {
676         g = GCD(P, N);
```

```
677
678             if (g > 1)
679                 goto L4;
680             else {
681                 y = x;
682                 c = 0;
683             }
684         }
685
686         k--;
687
688         if (k != 0)
689             goto L2;
690
691         g = GCD(P, N);
692
693         if (g > 1)
694             goto L4;
695
696         else
697         {
698             x1 = x;
699             k = l;
700             l = 2 * l;
701
702             for (i = 0; i < k; i++)
703                 x = (x * x + 1) % N;
704
705             y = x;
706             c = 0;
707             goto L2;
708         }
709     L4:
710         do
711         {
712             y = (y * y + 1) % N;
713             g = GCD(x1 - y, N);
714         } while (g <= 1);
715
716         if (g < N)
717         {
718             int e = 0;
719
720             while (N % g == 0)
721             {
722                 e++;
723                 N /= g;
724             }
725
726             if (RabinMiller(g) && g > limit)
727                 return g;
728
```

```
729             if (RabinMiller(N) && N > limit)
730                 return N;
731         }
732
733         return 0;
734     }
735
736     long long PMOFirstStage(long long N, long long* x) {
737         /* see Cohen Algorithm 8.2.2 page 439 */
738         int c = 0, i = -1, ii = 0, j = i;
739         long l = 0;
740         long long g = 0, q = 0, q1 = 0, y = 0;
741
742         *x = 2;
743         y = *x;
744         prime = 2;
745         for (ii = 0; ii < SIEVE_COUNT1; ii++)
746             p1[ii] = NextPrime(B1);
747         prime = 2;
748     L2:
749         i++;
750         if (i == SIEVE_COUNT1) {
751             g = GCD(*x - 1, N);
752
753             if (g == 1)
754                 return 0;
755             else {
756                 i = j;
757                 *x = y;
758                 goto L5;
759             }
760         }
761         else {
762             q = p1[i];
763             q1 = q;
764             l = (long)(B1 / q);
765         }
766         while (q1 <= l)
767             q1 *= q;
768         *x = PowMod(*x, q1, N);
769         c++;
770         if (c < 20)
771             goto L2;
772         g = GCD(*x - 1, N);
773         if (g == 1) {
774             c = 0;
775             j = i;
776             y = *x;
777             goto L2;
778         }
779         else {
780             i = j;
```

```
781             *x = y;
782         }
783     L5:
784         i++;
785         q = p1[i];
786         q1 = q;
787     L6:
788         *x = PowMod(*x, q, N);
789         g = GCD(*x - 1, N);
790         if (g == 1) {
791             q1 *= q;
792             if (q1 <= B1)
793                 goto L6;
794             else
795                 goto L5;
796         }
797         else if (g < N)
798             return g;
799         else if (g == N)
800             return 0;
801         return 0;
802     }
803
804     long long PMOSecondStage(long long N) {
805         /* see Cohen Algorithm 8.8.3 page 441 */
806         int c = 0, count = 0, i = 0, ii = 0, j = 0;
807         long long P = 0, b = 0, g = 0, x = 0, y = 0;
808
809         g = PMOFirstStage(N, &x);
810         if (g != 0)
811             return g;
812         b = x;
813         c = 0;
814         P = 1;
815         i = -1;
816         j = i;
817         y = x;
818         prime = 2;
819         for (ii = 0; ii < SIEVE_COUNT2; ii++)
820             p2[ii] = NextPrime(B2);
821         prime = 2;
822         for (ii = 0; ii < SIEVE_COUNT2 - 1; ii++) {
823             long t1 = p2[ii], t2 = 0;
824             if (t1 >= B1) {
825                 t2 = p2[ii + 1];
826                 d2[count] = t2 - t1;
827                 b2[count] = (long)PowMod(b, (long long)d2[count], N);
828                 count++;
829             }
830         }
831         prime = 2;
832         x = PowMod(x, p1[SIEVE_COUNT1 - 1], N);
```

```
L3:
    i++;
    x *= PowMod(b, d2[i], N);
    P *= (x - 1);
    c++;
    if (i >= SIEVE_COUNT2)
        goto L6;
    if (c < 20)
        goto L3;
    g = GCD(x - 1, N);
    if (g == 1) {
        c = 0;
        j = i;
        y = x;
        goto L3;
    }
L5:
    i = j;
    x = y;
    do {
        x = PowMod(b, d2[i], N);
        i++;
        g = GCD(x - 1, N);
    } while (g == 1);
    if (g < N)
        return g;
    if (g == N)
        return 0;
L6:
    g = GCD(P, N);
    if (g == 1)
        return 0;
    goto L5;
}

int GoldwasserKilian(long long N, long B0) {
    /* returns 0 if N is composite 1 if prime */
    int i = 0, ssl = 0;
    double runtime = 0.0;
    long long Ni = N, a = 0, b = 0, d = 0, limit = 0;
    long long bpr = 0, m = 0, p = 0, pmo = 0, q = 0, x = 0, y = 0;
    ECPOINT P = { 0 }, P1 = { 0 }, P2 = { 0 };
    clock_t time0 = clock(), time1 = 0;
    HANDLE hStdout = NULL;
    TCHAR msgBuf[BUF_SIZE] = { 0 };
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hStdout == INVALID_HANDLE_VALUE)
        return 1;
    Sieve(SIEVE_LIMIT2);
```

```
885     time1 = clock();
886     runtime = ((double)time1 - time0) / CLOCKS_PER_SEC;
887     printf_s("Prime sieving time = %lf\n", runtime);
888 L2:
889 L3:
890     limit = (long long)sqrt(sqrt((double)Ni)) + 1;
891     if (Ni <= B0) {
892         p = 2;
893         while (p <= (long long)sqrt((double)Ni)) {
894             if (Ni % p == 0) {
895                 printf_s("1 factor = %lld\n", p);
896                 return 0;
897             }
898             else if (p > limit)
899                 break;
900             p++;
901             while (!GetBit((long)p, sieve)) p++;
902         }
903         return 1;
904     }
905     if (!RabinMiller(Ni)) goto L9;
906     for (;;) {
907         a = RandomRange(1, Ni);
908         b = RandomRange(1, Ni);
909         m = ((4 * (a * a * a)) % Ni + 27 * (b * b) % Ni) % Ni;
910         if (m != 0) break;
911     }
912     m = ShanksMestre(a, b, Ni);
913     p = 2;
914     q = 0;
915     while (p <= (long long)sqrt((double)m)) {
916         if (m % p == 0) {
917             if (p > limit) {
918                 q = p;
919                 break;
920             }
921         }
922         p++;
923         while (!GetBit((long)p, sieve)) p++;
924     }
925     if (q == 0) {
926         bpr = BrentPollardRho(m, limit);
927         if (bpr > limit)
928             q = bpr;
929         else {
930             pmo = PMOSecondStage(m);
931             if (pmo > limit)
932                 q = pmo;
933             else
934                 q = m / 2;
935         }
936     }
```

```
937     if (q != 0 && m & 1 || !RabinMiller(q))
938         goto L3;
939 L6:
940     while (1) {
941         x = RandomRange(1, Ni - 1);
942         y = (((x * x) % Ni * x) % Ni + (a * x) + b) % Ni;
943         if (y < 0) y += p;
944         if (y != 0 && Jacobi(y, Ni) != -1)
945             break;
946     }
947     y = SquareRootModPrime(y, Ni);
948     if (y == 0)
949         goto L9;
950     P.x = x;
951     P.y = y;
952     Multiply(a, m, Ni, P, &P1);
953     if (!(P1.x == 0 && P1.y == 1))
954         goto L6;
955     Multiply(a, m / q, Ni, P, &P2);
956     if (P2.x == 0 && P2.y == 1)
957         goto L6;
958     StringCchPrintf(msgBuf, 255,
959         TEXT("N[%d] = %lld\n"), i, Ni);
960     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
961     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
962         &dwChars, NULL);
963     StringCchPrintf(msgBuf, BUF_SIZE, TEXT("a = %lld\n"), a);
964     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
965     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
966         &dwChars, NULL);
967     StringCchPrintf(msgBuf, BUF_SIZE, TEXT("b = %lld\n"), b);
968     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
969     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
970         &dwChars, NULL);
971     StringCchPrintf(msgBuf, BUF_SIZE, TEXT("m = %lld\n"), m);
972     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
973     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
974         &dwChars, NULL);
975     StringCchPrintf(msgBuf, BUF_SIZE, TEXT("q = %lld\n"), q);
976     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
977     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
978         &dwChars, NULL);
979     StringCchPrintf(msgBuf, BUF_SIZE,
980         TEXT("P = (%lld, %lld)\n"), P.x, P.y);
981     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
982     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
983         &dwChars, NULL);
984     StringCchPrintf(msgBuf, BUF_SIZE,
985         TEXT("P1 = (%lld, %lld)\n"), P1.x, P1.y);
986     ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
987     WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
988         &dwChars, NULL);
```

```
    StringCchPrintf(msgBuf, BUF_SIZE,
        TEXT("P2 = (%lld, %lld)\n"), P2.x, P2.y);
    ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
        &dwChars, NULL);
    i++;
    Ni = q;
    goto L2;
L9:
    if (i == 0) return 0;
    i--;
    goto L3;
}

void test_1(void) {
    int i;
    long long a = 1, b = 6, p = 11;
    ECPOINT P = { 2, 7 }, R;

    for (i = 1; i <= 12; i++) {
        Multiply(a, i, p, P, &R);
        printf_s("%2d(%2lld, %2lld) = (%2lld, %2lld)\n",
            i, P.x, P.y, R.x, R.y);
    }
    printf_s("|E(F_%lld)| = %lld\n", p, Cardinality(a, b, p));
    printf_s("Rabin_Miller(127) = %lld\n", RabinMiller(127));
    printf_s("Rabin_Miller(499) = %lld\n", RabinMiller(499));
}

void test_2(void) {
    long long a = 1, b = 1, p = 5;
    printf_s("|E(F_%lld)| = %lld\n", p, Cardinality(a, b, p));
}

long long ParseNumber(char numStr[], long long* n) {
    char c = 0, expStr[32] = { 0 };
    char addStr[32] = { 0 }, radStr[32] = { 0 };
    char* karet = NULL, * nsign = NULL, * psign = NULL;
    long long radix = 0, expon = 0, addme = 0;
    long long error = 0, i = 0, j = 0, l = 0;

    karet = strchr(numStr, '^');
    nsign = strchr(numStr, '-');
    psign = strchr(numStr, '+');
    l = strlen(numStr);

    if (karet == NULL && nsign == NULL && psign == NULL) {
        c = numStr[i++];

        while (c >= '0' && c <= '9' && i < l) {
            radStr[j++] = c;
            c = numStr[i++];
```

```
1041            }
1042
1043            if (i == l)
1044            {
1045                *n = atoll(numStr);
1046                return 1;
1047            }
1048
1049            else
1050                return -1;
1051
1052            if (j == 0)
1053                return -2;
1054        }
1055
1056        else if (karet) {
1057            c = numStr[0];
1058            i = j = 0;
1059
1060            while ((c >= '0' && c <= '9') && (i < l) || (c == '^')) {
1061                c = numStr[i++];
1062
1063                if (c != '^')
1064                    radStr[j++] = c;
1065                else if (c == '^')
1066                    break;
1067            }
1068
1069            radStr[j] = '\0';
1070            radix = atol(radStr);
1071
1072            if (psign && !nsign)
1073                c = *(psign + 1);
1074            else if (!psign && nsign)
1075                c = *(nsign + 1);
1076            else
1077                return -3;
1078
1079            j = 0;
1080
1081            while ((c >= '0' && c <= '9') && (i < l)) {
1082                c = numStr[i++];
1083
1084                if (c >= '0' && c <= '9')
1085                    expStr[j++] = c;
1086            }
1087
1088            if (j == 0)
1089                return -4;
1090
1091            expStr[j++] = '\0';
1092            expon = atol(expStr);
```

```
1093
1094            if (i == l) {
1095                *n = (long long)pow(
1096                    (double)radix, (double)expon) + addme;
1097                return 1;
1098            }
1099
1100            j = 0;
1101            c = numStr[i];
1102
1103            while (
1104                (c >= '0' && c <= '9') && (i < l)) {
1105                if (c >= '0' && c <= '9')
1106                    addStr[j++] = c;
1107                c = numStr[i++];
1108            }
1109
1110            if (j != 0)
1111            {
1112                addStr[j] = '\0';
1113
1114                if (psign != NULL)
1115                    addme = atol(addStr);
1116                else if (nsign != NULL)
1117                    addme = -atol(addStr);
1118                else
1119                    return -5;
1120
1121                if (error == 0) {
1122                    *n = (long long)pow(
1123                        (double)radix, (double)expon) + addme;
1124                    return 1;
1125                }
1126            }
1127
1128            return -6;
1129        }
1130
1131        return 0;
1132    }
1133
1134    //https://learn.microsoft.com/en-us/windows/win32/procthread/creating-threads
1135
1136    typedef struct tagThreadData
1137    {
1138        long long N;
1139        long B0;
1140    } THREADDATA, * PTHREADDATA;
1141
1142    void ErrorHandler(LPCTSTR lpszFunction)
1143    {
1144        // Retrieve the system error message for the last-error code.
```

```
1145
1146      LPVOID lpMsgBuf = NULL;
1147      LPVOID lpDisplayBuf = NULL;
1148      DWORD dw = GetLastError();
1149
1150      FormatMessage(
1151          FORMAT_MESSAGE_ALLOCATE_BUFFER |
1152          FORMAT_MESSAGE_FROM_SYSTEM |
1153          FORMAT_MESSAGE_IGNORE_INSERTS,
1154          NULL,
1155          dw,
1156          MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
1157          (LPTSTR)&lpMsgBuf,
1158          0, NULL);
1159
1160      // Display the error message.
1161
1162      lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
1163          (lstrlen((LPCTSTR)lpMsgBuf) + lstrlen((LPCTSTR)lpszFunction) + 40) *
              sizeof(TCHAR));
1164      int ssl = StringCchPrintf((LPTSTR)lpDisplayBuf,
1165          LocalSize(lpDisplayBuf) / sizeof(TCHAR),
1166          TEXT("%s failed with error %d: %s"),
1167          lpszFunction, dw, lpMsgBuf);
1168      if (ssl == 0 || lpDisplayBuf == NULL)
1169          return;
1170      MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);
1171
1172      // Free error-handling buffer allocations.
1173
1174      LocalFree(lpMsgBuf);
1175      LocalFree(lpDisplayBuf);
1176  }
1177
1178  DWORD WINAPI MyThreadFunction(LPVOID lpParam)
1179  {
1180      double runtime = 0.0;
1181      clock_t time0 = 0, time1 = 0;
1182
1183      HANDLE hStdout;
1184      PTHREADDATA pDataArray;
1185      TCHAR msgBuf[BUF_SIZE] = { 0 };
1186      size_t cchStringSize;
1187      DWORD dwChars;
1188
1189      // Make sure there is a console to receive output results.
1190
1191      hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
1192      if (hStdout == INVALID_HANDLE_VALUE)
1193          return 1;
1194
1195      // Cast the parameter to the correct data type.
```

```
1196      // The pointer is known to be valid because
1197      // it was checked for NULL before the thread was created.
1198
1199      pDataArray = (PTHREADDATA)lpParam;
1200
1201      // Print the parameter values using thread-safe functions.
1202
1203      StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d, %lld\n"),
1204          pDataArray->B0, pDataArray->B0, pDataArray->N);
1205      int ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
1206      if (ssl != 0)
1207          WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize, &dwChars, NULL);
1208
1209      time0 = clock();
1210
1211      int gk = GoldwasserKilian(
1212          pDataArray->N,
1213          pDataArray->B0);
1214
1215      if (gk == 0)
1216          StringCchPrintf(msgBuf, BUF_SIZE,
1217              TEXT("Goldwasser-Kilian algorithm failed\n"));
1218      else if (gk == 1)
1219          StringCchPrintf(msgBuf, BUF_SIZE,
1220              TEXT("number is proven prime\n"));
1221      ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
1222      if (ssl == 0)
1223          WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
1224              &dwChars, NULL);
1225
1226      time1 = clock();
1227      runtime = ((double)time1 - time0) / CLOCKS_PER_SEC;
1228
1229      StringCchPrintf(msgBuf, BUF_SIZE,
1230          TEXT("runtime in seconds = %lf\n"), runtime);
1231      ssl = StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
1232
1233      if (ssl == 0)
1234          WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize,
1235              &dwChars, NULL);
1236      return 0;
1237  }
1238
1239  int StartThread(long B0, long long N) {
1240      PTHREADDATA pDataArray[MAX_THREADS] = { 0 };
1241      DWORD dwThreadIdArray[MAX_THREADS] = { 0 };
1242      HANDLE hThreadArray[MAX_THREADS] = { 0 };
1243      // Create MAX_THREADS worker threads.
1244      for (int i = 0; i < MAX_THREADS; i++)
1245      {
1246          // Allocate memory for thread data.
1247          pDataArray[i] = (PTHREADDATA)HeapAlloc(
```

```
1248            GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(THREADDATA));
1249
1250        if (pDataArray[i] == NULL) {
1251            // If the array allocation fails, the system is out of memory
1252            // so there is no point in trying to print an error message.
1253            // Just terminate execution.
1254            ExitProcess(2);
1255        }
1256        // Generate unique data for each thread to work with.
1257        pDataArray[i]->B0 = B0;
1258        pDataArray[i]->N = N;
1259        // Create the thread to begin execution on its own.
1260        hThreadArray[i] = CreateThread(
1261            NULL,                   // default security attributes
1262            0,                      // use default stack size
1263            MyThreadFunction,       // thread function name
1264            pDataArray[i],          // argument to thread function
1265            0,                      // use default creation flags
1266            &dwThreadIdArray[i]);   // returns the thread identifier
1267        // Check the return value for success.
1268        // If CreateThread fails, terminate execution.
1269        // This will automatically clean up threads and memory.
1270        if (hThreadArray[i] == NULL) {
1271            ErrorHandler(TEXT("CreateThread"));
1272            ExitProcess(3);
1273        }
1274    } // End of main thread creation loop.
1275    // Wait until all threads have terminated.
1276    DWORD result = WaitForMultipleObjects(MAX_THREADS,
1277        hThreadArray, TRUE, INFINITE);
1278    // Close all thread handles and free memory allocations.
1279    for (int i = 0; i < MAX_THREADS; i++)
1280    {
1281        CloseHandle(hThreadArray[i]);
1282        if (pDataArray[i] != NULL)
1283        {
1284            HeapFree(GetProcessHeap(), 0, pDataArray[i]);
1285            pDataArray[i] = NULL;    // Ensure address is not reused.
1286        }
1287    }
1288
1289    return 0;
1290 }
1291
1292 int main(void)
1293 {
1294    double runtime = 0;
1295    long long N;
1296    clock_t time0 = 0, time1 = 0;
1297 #ifdef DEBUG1
1298    test_1();
1299    test_2();
```

```
#endif
    long long pmTest1 = PowMod(9726, 3533, 11413);
    if (pmTest1 != 5761)
    {
        printf_s("fatal error in PowMod function test 1\n");
        exit(-4);
    }
    /* pmTest1 from 1995 Stinson textbook */
    long long pmTest2 = PowMod(5, 596, 1234);
    if (pmTest2 != 1013)
    {
        printf_s("fatal error in PowMod function test 2\n");
        exit(-8);
    }
    /* pmTest2 from "Handbook of Applied Cryptography */
    for (;;) {
        char numStr[32] = { 0 };
        int st = 0;
        long B0 = 0;
        printf_s("number to be tested or 0 to quit:\n");
        scanf_s("%s", numStr, sizeof(numStr));
        if (!ParseNumber(numStr, &N)) {
            printf_s("number parsing error\n");
            continue;
        }
        if (N == 0) break;
        if (N > 2147483647) {
            printf_s("number is too large must be <= 2147483647\n");
            continue;
        }
        if (GCD(N, 6) != 1) {
            printf_s("number must be coprime with 6\n");
            continue;
        }
        if (!RabinMiller(N)) {
            printf_s("number is composite\n");
            continue;
        }
        printf_s("number of primes in factor base:\n");
        scanf_s("%s", numStr, sizeof(numStr));
        B0 = atol(numStr);
        st = StartThread(B0, N);
    }
    return 0;
}
```