```c
/*
** Pollard p - 1 integer factoring method
** (c) Wednesday, August 21 and also 1997
** by James Pate Williams, Jr., BA, BS, MSwE, PhD
** see "A Course in Computational Algebraic Number
** Theory" (c) Henri Cohen 1996 Third Corrected Printing
** 64-bit version largest allowed number is 2^31-1
** which is equal to 2,147,483,647 commas are not
** allowed in inputting a number to factor, however,
** exponential notation such as 2^16-1 is okay
*/

#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

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

/* 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 b = (long)(i % BITS_PER_LONG);
    long c = (long)(i / BITS_PER_LONG);
    return (sieve[c] >> (BITS_PER_LONG_1 - b)) & 1L;
}

void SetBit(long i, long v) {
    long b = (long)(i % BITS_PER_LONG);
    long c = (long)(i / BITS_PER_LONG);
```

```
53      long mask = 1L << (BITS_PER_LONG_1 - b);
54      if (v == 1)
55          sieve[c] |= mask;
56      else
57          sieve[c] &= ~mask;
58  }
59
60  void Sieve() {
61      long c, i, inc;
62      SetBit(0, 0);
63      SetBit(1, 0);
64      SetBit(2, 1);
65      for (i = 3; i <= B2; i++)
66          SetBit(i, i & 1);
67      c = 3;
68      do {
69          i = c * c, inc = c + c;
70          while (i <= B2) {
71              SetBit(i, 0);
72              i += inc;
73          }
74          c += 2;
75          while (!GetBit(c)) c++;
76      } while (c * c <= B2);
77  }
78
79  long NextPrime(long bound) {
80      long q = 0;
81      while (q == 0 && prime <= bound) {
82          if (GetBit(prime))
83              q = prime;
84          prime++;
85      }
86      return q;
87  }
88
89  long long Pow(long long x, long long b)
90  {
91      /* returns x ^ b */
92      int bits[64] = { 0 }, i = 0, l = 0;
93      long long bb = b, z = 1;
94
95      while (bb != 0) {
96          bits[l++] = (int)(bb & 1);
97          bb >>= 1;
98      }
99
100     for (i = l - 1; i >= 0; i--) {
101         z = z * z;
102         if (bits[i] == 1) z = z * x;
103     }
104
```

```
105     return z;
106 }
107
108 long long PowMod(long long x, long long b, long long n)
109 {
110     /* returns x ^ b mod n
111     ** algorithm taken from Douglas R. Stinson
112     ** "Cryptography Theory and Practice" 1995
113     ** Figure 4.4 page 127
114     **/
115     long long bits[64] = { 0 }, i = 0, l = 0;
116     long long bb = b, z = 1;
117
118     while (bb != 0) {
119         bits[l++] = (int)(bb & 1);
120         bb >>= 1;
121     }
122
123     for (i = l - 1; i >= 0; i--) {
124         z = (z * z) % n;
125         if (bits[i] == 1) z = (z * x) % n;
126     }
127
128     if (z < 0)
129         z += n;
130
131     return z;
132 }
133
134 long long GCD(long long a, long long b) {
135     /* returns greatest common divisor of a and b */
136     long long r;
137
138     if (a < 0) a = -a;
139     if (b < 0) b = -b;
140     while (b > 0)
141         r = a % b, a = b, b = r;
142     return a;
143 }
144
145 long long Rand64() {
146     long long hi = ((long long)rand()) << 32;
147     long long lo = (long long)rand();
148     return hi | lo;
149 }
150
151 long long RandomRange(long long lower, long long upper) {
152     long long a = 0;
153
154     while (1) {
155         a = lower + Rand64() % (upper + 1);
156
```

```
157         if (a <= upper)
158             return a;
159     }
160 }
161
162 int RabinMiller(long long N, long long* a) {
163     /* given an integer N >= 3
164     ** returns 0 composite
165     ** returns 1 probably prime
166     ** see Cohen Algorithm 8.2.2 page 422
167     */
168     long long N1 = N - 1, b, c = 20, e, q = N1, t = 0;
169
170     if (N == 2 || N == 3) return 1;
171     /* N - 1 = 2 ^ t * q with q odd */
172     while (!(q & 1)) q >>= 1, t++;
173 L2:
174     *a = RandomRange(2, N - 1);
175     e = 0;
176     b = PowMod(*a, q, N);
177     if (b == 1) goto L4;
178     while (b != 1 && b != N1 && e <= t - 2) {
179         b = (b * b) % N;
180         e++;
181     }
182     if (b != N1) return 0LL;
183 L4:
184     c--;
185     if (c > 0) goto L2;
186     return 1;
187 }
188
189 long long PMOFirstStage(long long N, long long* x) {
190     /* see Cohen Algorithm 8.2.2 page 439 */
191     int c = 0, i = -1, ii = 0, j = i;
192     long l = 0;
193     long long g = 0, q = 0, q1 = 0, y = 0;
194
195     *x = 2;
196     y = *x;
197     prime = 2;
198     for (ii = 0; ii < SIEVE_COUNT1; ii++)
199         p1[ii] = NextPrime(B1);
200     prime = 2;
201 L2:
202     i++;
203     if (i == SIEVE_COUNT1) {
204         g = GCD(*x - 1, N);
205
206         if (g == 1)
207             return 0;
208         else {
```

```
209             i = j;
210             *x = y;
211             goto L5;
212         }
213     }
214     else {
215         q = p1[i];
216         q1 = q;
217         l = (long)(B1 / q);
218     }
219     while (q1 <= l)
220         q1 *= q;
221     *x = PowMod(*x, q1, N);
222     c++;
223     if (c < 20)
224         goto L2;
225     g = GCD(*x - 1, N);
226     if (g == 1) {
227         c = 0;
228         j = i;
229         y = *x;
230         goto L2;
231     }
232     else {
233         i = j;
234         *x = y;
235     }
236 L5:
237     i++;
238     q = p1[i];
239     q1 = q;
240 L6:
241     *x = PowMod(*x, q, N);
242     g = GCD(*x - 1, N);
243     if (g == 1) {
244         q1 *= q;
245         if (q1 <= B1)
246             goto L6;
247         else
248             goto L5;
249     }
250     else if (g < N)
251         return g;
252     else if (g == N)
253         return 0;
254     return 0;
255 }
256
257 long long PMOSecondStage(long long N) {
258     /* see Cohen Algorithm 8.8.3 page 441 */
259     int c = 0, count = 0, i = 0, ii = 0, j = 0;
260     long long P = 0, b = 0, g = 0, x = 0, y = 0;
```

```
261
262     g = PMOFirstStage(N, &x);
263     if (g != 0)
264         return g;
265     b = x;
266     c = 0;
267     P = 1;
268     i = -1;
269     j = i;
270     y = x;
271     prime = 2;
272     for (ii = 0; ii < SIEVE_COUNT2; ii++)
273         p2[ii] = NextPrime(B2);
274     prime = 2;
275     for (ii = 0; ii < SIEVE_COUNT2 - 1; ii++) {
276         long t1 = p2[ii], t2 = 0;
277         if (t1 >= B1) {
278             t2 = p2[ii + 1];
279             d2[count] = t2 - t1;
280             b2[count] = (long)PowMod(b, (long long)d2[count], N);
281             count++;
282         }
283     }
284     x = PowMod(x, p1[SIEVE_COUNT1 - 1], N);
285 L3:
286     i++;
287     x *= PowMod(b, d2[i], N);
288     P *= (x - 1);
289     c++;
290     if (i >= SIEVE_COUNT2)
291         goto L6;
292     if (c < 20)
293         goto L3;
294     g = GCD(x - 1, N);
295     if (g == 1) {
296         c = 0;
297         j = i;
298         y = x;
299         goto L3;
300     }
301 L5:
302     i = j;
303     x = y;
304     do {
305         x = PowMod(b, d2[i], N);
306         i++;
307         g = GCD(x - 1, N);
308     } while (g == 1);
309     if (g < N)
310         return g;
311     if (g == N)
312         return 0;
```

```
313 L6:
314     g = GCD(P, N);
315     if (g == 1)
316         return 0;
317     goto L5;
318 }
319
320 void DoSingletonsSort(int ii, int jj) {
321     /* see "Sorting and Sort Systems"
322     ** by Harold Lorin (IBM book)
323     */
324     int m = 0;
325     int i = ii;
326     int j = jj;
327     int iu[50] = { 0 };
328     int il[50] = { 0 };
329     int ij = 0, l = 1, k = 0;
330     PRIMEEXPONENT t = { 0 }, tt = { 0 };
331 Label1:
332     if (i >= j)
333         goto Label8;
334     goto Label2;
335 Label2:
336     k = i;
337     ij = (j + i) / 2;
338     t = pev[ij];
339     if (pev[i].prime <= t.prime)
340         goto Label3;
341     pev[ij] = pev[i];
342     pev[i] = t;
343     t = pev[ij];
344     goto Label3;
345 Label3:
346     l = j;
347     if (pev[j].prime >= t.prime)
348         goto Label5;
349     pev[ij] = pev[j];
350     pev[j] = t;
351     t = pev[ij];
352     if (pev[i].prime <= t.prime)
353         goto Label5;
354     pev[ij] = pev[i];
355     pev[i] = t;
356     t = pev[ij];
357     goto Label5;
358 Label4:
359     pev[l] = pev[k];
360     pev[k] = tt;
361     goto Label5;
362 Label5:
363     l--;
364     if (pev[l].prime > t.prime)
```

```
365         goto Label5;
366     tt = pev[l];
367     goto Label6;
368 Label6:
369     k++;
370     if (pev[k].prime < t.prime)
371         goto Label6;
372     if (k <= l)
373         goto Label4;
374     if (l - i <= j - k)
375         goto Label7;
376     il[m] = i;
377     iu[m] = l;
378     i = k;
379     m++;
380     goto Label9;
381 Label7:
382     il[m] = k;
383     iu[m] = j;
384     j = l;
385     m++;
386     goto Label9;
387 Label8:
388     m--;
389     if (m == -1)
390         return;
391     i = il[m];
392     j = iu[m];
393     goto Label9;
394 Label9:
395     if (j - i > 10)
396         goto Label2;
397     if (i == ii)
398         goto Label1;
399     i--;
400     goto Label10;
401 Label10:
402     i++;
403     if (i == j)
404         goto Label8;
405     t = pev[i + 1];
406     if (pev[i].prime <= t.prime)
407         goto Label10;
408     k = i;
409     goto Label11;
410 Label11:
411     pev[k + 1] = pev[k];
412     k--;
413     if (t.prime < pev[k].prime)
414         goto Label11;
415     pev[k + 1] = t;
416     goto Label10;
```

```
417 }
418
419 void SingletonsSort(int n) {
420     DoSingletonsSort(0, n - 1);
421 }
422
423 long long PrimePowerTest(long long n) {
424     /* see Cohen Algorithm 1.7.5 page 42 */
425     long long a = 0, d = 0, p = 0, q = 0;
426
427     if ((n & 1) == 0) {
428         p = 2;
429         goto L4;
430     }
431     else
432         q = n;
433 L2:
434     if (RabinMiller(q, &a)) {
435         p = q;
436         goto L4;
437     }
438     d = GCD(PowMod(a, q, n) - a, q);
439     if (d == 1 || d == q)
440         return 0;
441     q = d;
442     goto L2;
443 L4:
444     if (!RabinMiller(p, &a)) {
445         q = p;
446         goto L2;
447     }
448     while (n > 1) {
449         if (n % p != 0)
450             return 0;
451         n /= p;
452     }
453     if (n == 1)
454         return p;
455     return 0;
456 }
457
458 long long ParseNumber(char numStr[], long long* n) {
459     char c = 0, expStr[32] = { 0 };
460     char addStr[32] = { 0 }, radStr[32] = { 0 };
461     char* karet = NULL, * nsign = NULL, * psign = NULL;
462     long long radix = 0, expon = 0, addme = 0;
463     long long error = 0, i = 0, j = 0, l = 0;
464
465     karet = strchr(numStr, '^');
466     nsign = strchr(numStr, '-');
467     psign = strchr(numStr, '+');
468     l = strlen(numStr);
```

```
469
470     if (karet == NULL && nsign == NULL && psign == NULL) {
471         c = numStr[i++];
472
473         while (c >= '0' && c <= '9' && i < l) {
474             radStr[j++] = c;
475             c = numStr[i++];
476         }
477
478         if (i == l)
479         {
480             *n = atoll(numStr);
481             return 1;
482         }
483
484         else
485             return -1;
486
487         if (j == 0)
488             return -2;
489     }
490
491     else if (karet) {
492         c = numStr[0];
493         i = j = 0;
494
495         while ((c >= '0' && c <= '9') && (i < l) || (c == '^')) {
496             c = numStr[i++];
497
498             if (c != '^')
499                 radStr[j++] = c;
500             else if (c == '^')
501                 break;
502         }
503
504         radStr[j] = '\0';
505         radix = atol(radStr);
506
507         if (psign && !nsign)
508             c = *(psign + 1);
509         else if (!psign && nsign)
510             c = *(nsign + 1);
511         else
512             return -3;
513
514         j = 0;
515
516         while ((c >= '0' && c <= '9') && (i < l)) {
517             c = numStr[i++];
518
519             if (c >= '0' && c <= '9')
520                 expStr[j++] = c;
```

```
521         }
522
523         if (j == 0)
524             return -4;
525
526         expStr[j++] = '\0';
527         expon = atol(expStr);
528
529         if (i == l) {
530             *n = (long long)pow(
531                 (double)radix, (double)expon) + addme;
532             return 1;
533         }
534
535         j = 0;
536         c = numStr[i];
537
538         while (
539             (c >= '0' && c <= '9') && (i < l)) {
540             if (c >= '0' && c <= '9')
541                 addStr[j++] = c;
542             c = numStr[i++];
543         }
544
545         if (j != 0)
546         {
547             addStr[j] = '\0';
548
549             if (psign != NULL)
550                 addme = atol(addStr);
551             else if (nsign != NULL)
552                 addme = -atol(addStr);
553             else
554                 return -5;
555
556             if (error == 0) {
557                 *n = (long long)pow(
558                     (double)radix, (double)expon) + addme;
559                 return 1;
560             }
561         }
562
563         return -6;
564     }
565
566     return 0;
567 }
568
569 int main(void)
570 {
571     double runtime = 0;
572     clock_t time0 = 0, time1 = 0;
```

```
573     time0 = clock();
574     Sieve();
575     time1 = clock();
576     runtime = ((double)time1 - time0) / CLOCKS_PER_SEC;
577     printf_s("prime number sieve creation\n");
578     printf_s("time in seconds = %lf\n", runtime);
579     long long em1 = PowMod(9726, 3533, 11413);
580     long long em2 = PowMod(5, 596, 1234);
581     for (;;) {
582         char numStr[32] = { 0 };
583         int i = 0, peCount = 0;
584         long long a = 0, g = 0, N = 0, power = 0;
585
586         printf_s("number to be factored or 0 to quit:\n");
587         scanf_s("%s", numStr, sizeof(numStr));
588
589         if (!ParseNumber(numStr, &N)) {
590             printf_s("number parsing error\n");
591             continue;
592         }
593
594         if (N == 0) break;
595
596         if (N > 2147483647) {
597             printf_s("number is too large must be <= 2147483647\n");
598             continue;
599         }
600
601         if (RabinMiller(N, &a)) {
602             printf_s("number is prime\n");
603             continue;
604         }
605
606         power = PrimePowerTest(N);
607
608         if (power >= 2) {
609             printf_s("prime powers are not allowed\n");
610             continue;
611         }
612
613         time0 = clock();
614         peCount = 0;
615
616         for (;;) {
617             g = PMOSecondStage(N);
618             if (g == 0)
619                 break;
620             pev[peCount].exponent = 0;
621             while (N % g == 0) {
622                 pev[peCount].exponent++;
623                 N /= g;
624             }
```

```
625             pev[peCount++].prime = g;
626             if (N == 1)
627                 break;
628             if (RabinMiller(N, &a)) {
629                 pev[peCount].exponent = 1;
630                 pev[peCount++].prime = N;
631                 break;
632             }
633         }
634         if (g == 0 && peCount == 0) {
635             printf_s("Pollard p - 1 failure\n");
636             continue;
637         }
638         SingletonsSort(peCount);
639         for (i = 0; i < peCount; i++) {
640             int rm = RabinMiller(pev[i].prime, &a);
641             char ch = rm == 1 ? 'p' : 'c';
642             printf_s("%d\t%lld\t%d\t%c\n",
643                 i + 1, pev[i].prime, pev[i].exponent, ch);
644         }
645         time1 = clock();
646         runtime = ((double)time1 - time0) / CLOCKS_PER_SEC;
647         printf_s("factoring time in seconds = %lf\n", runtime);
648     }
649     return 0;
650 }
```