GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: FACTORIZACIÓN DE ENTEROS

1. Introducción

En teoría de números, los factores primos de un número entero son los números primos divisores exactos de ese número entero. El proceso de búsqueda de esos divisores se denomina factorización de enteros, o factorización en números primos. Saber como realizar este proceso nos puede ser muy util para determinados ejercicios. A continuación abordaremos varios algoritmos para factorizar números enteros, cada uno de ellos puede ser tanto rápido como lento (algunos más lentos que otros) dependiendo de su entrada.

2. Conocimientos previos

2.1. Número primo

En matemáticas, un número primo es un número natural mayor que 1 que tiene únicamente dos divisores positivos distintos: él mismo y el 1. Por el contrario, los números compuestos son los números naturales que tienen algún divisor natural aparte de sí mismos y del 1, y, por lo tanto, pueden factorizarse. El número 1, por convenio, no se considera ni primo ni compuesto.

2.2. Criba de Eratóstenes

La Criba de Eratóstenes es un algoritmo utilizado para encontrar todos los números primos hasta un cierto límite dado. Fue desarrollado por el matemático griego Eratóstenes en el siglo III a.C. y es uno de los métodos más antiguos y eficientes para encontrar números primos.

2.3. Factorial

El factorial de un entero positivo N , el factorial de N o N factorial se define en principio como el producto de todos los números enteros positivos desde 1 (es decir, los números naturales) hasta N.

2.4. Pierre Fermat

Pierre de Fermat fue un matemático francés del siglo XVII conocido por sus contribuciones a la teoría de números, geometría analítica y cálculo. Una de las contribuciones más famosas de Fermat es el Teorema de Fermat, que enunció en una nota al margen de su ejemplar de la obra de Diofanto. El teorema establece que no existen enteros positivos a, b, c y n mayores que 2 que satisfagan la ecuación $a^n + b^n = c^n$. Este teorema fue uno de los problemas abiertos más famosos en matemáticas y se conoció como el Último Teorema de Fermat. Fue finalmente demostrado por Andrew Wiles en 1994 utilizando técnicas avanzadas de matemáticas modernas.

2.5. John Pollard

John Pollard es un matemático británico conocido por sus contribuciones en el campo de la teoría de números y la criptografía. Nació en 1926 y ha realizado importantes investigaciones

en áreas como los números primos, los algoritmos de factorización y la seguridad en sistemas criptográficos.

2.6. Multiplicación de Montgomery

La multiplicación de Montgomery es un método eficiente para realizar operaciones de multiplicación en aritmética modular. Fue desarrollada por Peter L. Montgomery en la década de 1980 y se utiliza principalmente en criptografía asimétrica, como en algoritmos de cifrado de clave pública como RSA.

2.7. Algoritmo de Euclides

El algoritmo de Euclides es un método eficiente para encontrar el máximo común divisor (MCD) de dos números enteros. Fue desarrollado por el matemático griego Euclides en el siglo III a.C. y sigue siendo uno de los algoritmos más utilizados en matemáticas y computación.

3. Desarrollo

Para descomponer un número en factores primos solo basta con seguir la idea aprendida en enseñanzas anteriores ejemplo:

4620	2
2310	2
1155	3
385	5
77	7
11	11
1	

En esta caso $4620 = 2^2 * 3^1 * 5^1 * 7^1 * 11^1$

3.1. Probando división

Este es el algoritmo más básico para encontrar una factorización prima.

Dividimos por cada divisor posible d. Se puede observar que es imposible que todos los factores primos de un número compuesto n ser más grande que $\sqrt{n}$. Por lo tanto, sólo necesitamos probar los divisores $2 \leq d \leq \sqrt{n}$.

El divisor más pequeño debe ser un número primo. Eliminamos el número factorizado y continuamos el proceso. Si no podemos encontrar ningún divisor en el rango $[2; \sqrt{n}]$, entonces el número en sí tiene que ser primo.

3.1.1. Factorización de rueda

Esta es una optimización de la división de prueba. Una vez que sabemos que el número no es divisible por 2, no necesitamos verificar otros números pares. Esto nos deja sólo $50\,\%$ de los números a comprobar. Después de factorizar 2 y obtener un número impar, podemos simplemente comenzar con 3 y contar solo los demás números impares.

Este método se puede ampliar aún más. Si el número no es divisible por 3, también podemos ignorar todos los demás múltiplos de 3 en cálculos futuros. Entonces solo necesitamos verificar los números $5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ \ldots$. Podemos observar un patrón de estos números restantes. Necesitamos verificar todos los números con $d \bmod 6 = 1$ y $d \bmod 6 = 5$. Entonces esto nos deja sólo con $33,3\,\%$ por ciento de los números a verificar. Podemos implementar esto factorizando primero los números primos 2 y 3, después de lo cual comenzamos con 5 y solo contamos los restos 1 y 5 módulo 6.

Si continuamos ampliando este método para incluir aún más números primos, se pueden alcanzar mejores porcentajes, pero las listas de omisión serán más grandes.

3.1.2. Primos precalculados

Extendiendo el método de factorización de la rueda indefinidamente, solo nos quedarán números primos para verificar. Una buena forma de comprobarlo es precalcular todos los números primos con el criba de Eratóstenes hasta que $\sqrt{n}$ y pruébelos individualmente.

3.1.3. Factorización de N

Una buena manera de verificar es precalcular todos los números primos con la criba de Eratóstenes hasta $\sqrt{N}$ y probarlos individualmente.

Para un factor primo p de N, la multiplicidad de p es el máximo exponente a para el cual p^a es un divisor de N. La factorización de un número entero es una lista de los factores primos de ese número, junto con su multiplicidad. El Teorema fundamental de la Aritmética establece que todo número entero positivo tiene una factorización de primos única.

3.1.4. Factorización de N!

Que pasa si ahora queremos descomponer N! la idea básica seria iterar desde 1 hasta N e ir descomponiendo cada número acumulando las potencias o hallar N! y luego descomponerlo, pero esto tendría una complejidad de O($N * \sqrt{N}$) el cual sería muy costoso para valores muy grandes de N. Analicemos la siguiente idea:

10!=1*2*3*4*5*6*7*8*9*10 Necesitamos hallar cuantos 2 hay.

10!	1	2	3	4	5	6	7	8	9	10	total
Descomposición de n1	1	2	3	2^2	5	2*3	7	2^3	3^2	2*5	
Primo 2	0	1	0	2	0	1	0	3	0	1	8
Primo 3	0	0	1	0	0	1	0	0	2	0	4

Para el caso del 2 seria todos los múltiplos de 2 hasta N ejemplo para 10 hay 5 que seria 10/2 luego todos los múltiplos de 4 hasta N que sería N/4 y así hasta que la potencia de 2 sea mayor que N la suma de estos seria el exponente de la potencia de 2 luego de descomponer N!, en este caso sería $2^{10/2+10/4+10/8}$=2^{5+2+1}=2^8 y este sería el procedimiento para todos los primos hasta N.

3.2. Método de factorización de Fermat

Podemos escribir un número compuesto impar $n = p \cdot q$ como la diferencia de dos cuadrados $n = a^2 - b^2$:

$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$$

El método de factorización de Fermat intenta explotar este hecho adivinando el primer cuadrado a^2, y comprobando si la parte restante, $b^2 = a^2 - n$, también es un número cuadrado. Si es así, entonces hemos encontrado los factores $a - b$ y $a + b$ de n.

3.3. Método de Pollard $P - 1$

Es muy probable que al menos un factor de un número sea B-powersmooth para B pequeños. B-powersmooth significa que cada potencia primaria d^k que divide $p - 1$ es es como máximo B. Por ejemplo, la factorización prima de 4817191 es $1303 \cdot 3697$. Y los factores son 31-powersmooth y 16-powersmooth respetablemente, porque $1303 - 1 = 2 \cdot 3 \cdot 7 \cdot 31$ y $3697 - 1 = 2^4 \cdot 3 \cdot 7 \cdot 11$. En 1974, John Pollard inventó un método para extraer B-powersmooth factores de un número compuesto.

La idea surge del pequeño teorema de Fermat . Sea una factorización de n ser $n = p \cdot q$. Dice que si a es coprimo a p , se cumple la siguiente afirmación:

$$a^{p-1} \equiv 1 \pmod{p}$$

Esto también significa que

$$a^{(p-1)^k} \equiv a^{k \cdot (p-1)} \equiv 1 \pmod{p}.$$

Entonces para cualquier M con $p-1 \mid M$ sabemos que $a^M \equiv 1$. Esto significa que $a^M - 1 = p \cdot r$, y por eso también $p \mid \gcd(a^M - 1, n)$.

Por lo tanto, si $p - 1$ por un factor p de n divide M, podemos extraer un factor usando el algoritmo de Euclides.

Está claro que el más pequeño M que es múltiplo de cada B-powersmooth numero es $\text{lcm}(1, 2, 3, 4, \ldots, B)$. O alternativamente:

$$M = \prod_{\text{prime } q \leq B} q^{\lfloor \log_q B \rfloor}$$

Aviso, si $p-1$ divide M para todos los factores primos p de n , entonces $\gcd(a^M - 1, n)$ solo será n. En este caso no recibimos ningún factor. Por lo tanto, intentaremos realizar el $\gcd$ varias veces, mientras calculamos M.

Algunos números compuestos no tienen B-powersmooth factores para pequeños B . Por ejemplo, los factores del número compuesto $100\ 000\ 000\ 000\ 000\ 493 = 763\ 013 \cdot 131\ 059\ 365\ 961$ son $190\ 753$-powersmooth y $1\ 092\ 161\ 383$-powersmooth. Tendremos que elegir $B >= 190\ 753$ para factorizar el número

Observe que este es un algoritmo probabilístico. Una consecuencia de esto es que existe la posibilidad de que el algoritmo no pueda encontrar ningún factor.

3.3.1. Algoritmo ρ de Pollard

El algoritmo Rho de Pollard es otro algoritmo de factorización de John Pollard. Sea la factorización prima de un número $n = pq$. El algoritmo analiza una secuencia pseudoaleatoria. $\{x_i\} = \{x_0,\ f(x_0),\ f(f(x_0)),\ \ldots\}$ dónde f es una función polinómica, generalmente $f(x) = (x^2 + c) \bmod n$ es elegido con $c = 1$.

En este caso no nos interesa la secuencia $\{x_i\}$. Estamos más interesados en la secuencia. $\{x_i \bmod p\}$. Desde f es una función polinómica y todos los valores están en el rango $[0;\ p)$, esta secuencia eventualmente convergerá en un bucle. La paradoja del cumpleaños en realidad sugiere que el número esperado de elementos es $O(\sqrt{p})$ hasta que comience la repetición. Si p es más pequeña que $\sqrt{n}$, la repetición probablemente comenzará en $O(\sqrt[4]{n})$.

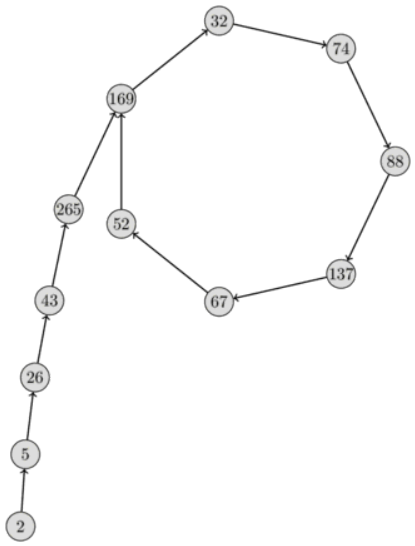
Aquí hay una visualización de tal secuencia $\{x_i \bmod p\}$ con $n = 2206637$, $p = 317$, $x_0 = 2$ y $f(x) = x^2 + 1$. Por la forma de la secuencia se puedever muy claramente por qué el algoritmo se llama algoritmo ρ de Pollard.

Sin embargo, todavía queda una pregunta abierta. ¿Cómo podemos explotar las propiedades de la secuencia? $\{x_i \bmod p\}$ a nuestra ventaja sin siquiera saber el número p ¿sí mismo?

En realidad es bastante fácil. Hay un ciclo en la secuencia $\{x_i \bmod p\}_{i \leq j}$ si y sólo si hay dos índices $s, t \leq j$ tal que $x_s \equiv x_t \bmod p$. Esta ecuación se puede reescribir como $x_s - x_t \equiv 0 \bmod p$ que es lo mismo que $p \mid \gcd(x_s - x_t, n)$.

Por tanto, si encontramos dos índices s y t con $g = \gcd(x_s - x_t, n) > 1$, hemos encontrado un ciclo y también un factor g de n. Es posible que $g = n$. En este caso no hemos encontrado un factor adecuado, por lo que debemos repetir el algoritmo con un parámetro diferente (valor inicial diferente x_0 , constante diferente c en la función polinómica f).

Para encontrar el ciclo, podemos utilizar cualquier algoritmo de detección de ciclo común.



3.3.2. Algoritmo de búsqueda de ciclos de Floyd

Este algoritmo encuentra un ciclo utilizando dos punteros que se mueven sobre la secuencia a diferentes velocidades. Durante cada iteración, el primer puntero avanzará un elemento, mientras que el segundo puntero avanzará a todos los demás elementos. Usando esta idea es fácil observar que si hay un ciclo, en algún momento el segundo puntero se encontrará con el primero durante los bucles. Si la duración del ciclo es λ y el μ es el primer índice en el que comienza el ciclo, entonces el algoritmo se ejecutará en $O(\lambda + \mu)$ tiempo.

Este algoritmo también se conoce como algoritmo de la liebre y la tortuga , basado en el cuento en el que una tortuga (el puntero lento) y una liebre (el puntero más rápido) tienen una carrera.

De hecho, es posible determinar el parámetro λ y μ usando este algoritmo (también en $O(\lambda + \mu)$ tiempo y $O(1)$ espacio). Cuando se detecta un ciclo, el algoritmo devolverá verdadero . Si la secuencia no tiene un ciclo, entonces la función se repetirá sin cesar. Sin embargo, esto se puede evitar utilizando el algoritmo Rho de Pollard.

La siguiente tabla muestra los valores de x y y durante el algoritmo para $n = 2206637$, $x_0 = 2$ y $c = 1$.

i	x_i mód n	x_{2i} mód n	x_i mód 317	x_{2i} mód 317	$\gcd(x_i - x_{2i}, n)$
0	2	2	2	2	–
1	5	26	5	26	1
2	26	458330	26	265	1
3	677	1671573	43	32	1
4	458330	641379	265	88	1
5	1166412	351937	169	67	1
6	1671573	1264682	32	169	1
7	2193080	2088470	74	74	317

Como se dijo anteriormente, si n es compuesto y el algoritmo devuelve n como factor, hay que repetir el procedimiento con diferentes parámetros x_0 y c. Por ejemplo, la elección $x_0 = c = 1$ no factorizará $25 = 5 \cdot 5$. El algoritmo volverá 25. Sin embargo, la elección $x_0 = 1$, $c = 2$ lo factorizará.

3.3.3. Algoritmo de Brent

Brent implementa un método similar al de Floyd, utilizando dos punteros. La diferencia es que en lugar de avanzar los punteros uno y dos lugares respectivamente, avanzan potencias de dos. Tan pronto como 2^i es mayor que λ y μ , encontraremos el ciclo.

La combinación de una división de prueba para números primos pequeños junto con la versión de Brent del algoritmo rho de Pollard crea un algoritmo de factorización muy potente.

4. Implementación

4.1. C++

4.1.1. Probando división

```
vector<long long> division1(long long n) {
   vector<long long> factorization;
   for (long long d = 2; d * d <= n; d++) {
      while (n % d == 0) {
         factorization.push_back(d);
         n /= d;
      }
   }
   if (n > 1) factorization.push_back(n);
   return factorization;
}
```

4.1.2. Factorización de rueda

```
vector<long long> wheel(long long n) {
```

```
    vector<long long> factorization;
    while (n % 2 == 0) {
        factorization.push_back(2);
        n /= 2;
    }
    for (long long d = 3; d * d <= n; d += 2) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    if (n > 1) factorization.push_back(n);
    return factorization;
}

vector<long long> wheelv2(long long n) {
    vector<long long> factorization;
    for (int d : {2, 3, 5}) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
    }
    static array<int, 8> increments = {4, 2, 4, 2, 4, 6, 2, 6};
    int i = 0;
    for (long long d = 7; d * d <= n; d += increments[i++]) {
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
        if (i == 8) i = 0;
    }
    if (n > 1) factorization.push_back(n);
    return factorization;
}
```

4.1.3. Primos precalculados

```
vector<long long> primes;

vector<long long> trial_division4(long long n) {
    vector<long long> factorization;
    for (long long d : primes) {
        if (d * d > n) break;
        while (n % d == 0) {
            factorization.push_back(d);
            n /= d;
        }
```

```
    }
    if (n > 1) factorization.push_back(n);
    return factorization;
}
```

4.1.4. Factorización de N

En las dos implementaciones siguientes es evidente que se necesita precalcular todos los primos al menos hasta N y tenerlos almacenados en el vector *primos*

```
//definir una tupla para almacenar a^b
//(a base)=>.first  (b expo)=>.second
typedef pair<int,int>fac;
vector<fac> Factorizar(int n){
    int l = primos.size();
    //iterar sobre todos los primos
    //hasta que algun primo sea mayor que n
    vector<fac>FP;
    for(int i=0;i<l && primos[i]<=n ;i++){
        int pi = primos[i];
        int exp = 0;
        if(n%pi==0){
            while(n%pi==0){
                exp++;//contador para el exponente
                n/=pi;//divido entre la base
            }
            // pi^exp
                // {a,b}  es lo mismo que make_pair(a,b) o Pair(a,b)
            FP.push_back({pi,exp});
             }
    }
    // si n es distinto n es un primo
    if(n>1) FP.push_back({n,1});
    return FP;
}
```

4.1.5. Factorización de N!

s

```
vector<fac> FactorizarFactorial(int n){
    int l = primos.size();
    vector<fac>FP;
    //iterar sobre todos los primos menores e iguales que n
    for(int i=0;i<l && primos[i]<=n;i++){
        //acumulador para la potencia del primo actual
        int exp = 0;
```

```
        //potencia del primo actual
        int pot = primos[i];
        //mientras la potencia sea menor que n
        while(pot<=n){
          //multiplos de la potencia menores que n
          exp+=n/pot;
          //proxima potencia del primo
          pot*=primos[i];
        }
        FP.push_back({primos[i],exp});
    }
    return FP;
}
```

4.1.6. Método de factorización de Fermat

```
int fermat(int n) {
    int a = ceil(sqrt(n));
    int b2 = a*a - n;
    int b = round(sqrt(b2));
    while (b * b != b2) {
        a = a + 1;
        b2 = a*a - n;
        b = round(sqrt(b2));
    }
    return a - b;
}
```

4.1.7. Método de Pollard $P-1$

```
long long pollards_p_minus_1(long long n) {
    int B = 10;
    long long g = 1;
    while (B <= 1000000 && g < n) {
        long long a = 2 + rand() %  (n - 3);
        g = gcd(a, n);
        if (g > 1) return g;

        // calcular a^M
        for (int p : primes) {
            if (p >= B) continue;
            long long p_power = 1;
            while (p_power * p <= B)
                p_power *= p;
                a = power(a, p_power, n);
                            g = gcd(a - 1, n);
                if (g > 1 && g < n) return g;
```

```
        }
        B *= 2;
    }
    return 1;
}
```

4.1.8. Algoritmo de ρ de Pollard

```
long long mult(long long a, long long b, long long mod) {
    long long result = 0;
    while (b) {
        if (b & 1) result = (result + a) % mod;
        a = (a + a) % mod;
        b >>= 1;
    }
    return result;
}

long long f(long long x, long long c, long long mod) {
    return (mult(x, x, mod) + c) % mod;
}

long long rho(long long n, long long x0=2, long long c=1) {
    long long x = x0;
    long long y = x0;
    long long g = 1;
    while (g == 1) {
        x = f(x, c, n);
        y = f(y, c, n);
        y = f(y, c, n);
        g = gcd(abs(x - y), n);
    }
    return g;
}
```

4.1.9. Algoritmo de Brent

```
long long brent(long long n, long long x0=2, long long c=1) {
    long long x = x0;
    long long g = 1;
    long long q = 1;
    long long xs, y;

    int m = 128;
    int l = 1;
    while (g == 1) {
        y = x;
```

```
      for (int i = 1; i < l; i++) x = f(x, c, n);
      int k = 0;
      while (k < l && g == 1) {
         xs = x;
         for (int i = 0; i < m && i < l - k; i++) {
            x = f(x, c, n);
            q = mult(q, abs(y - x), n);
         }
         g = gcd(q, n);
         k += m;
      }
      l *= 2;
   }
   if (g == n) {
     do {
        xs = f(xs, c, n);
        g = gcd(abs(xs - y), n);
     } while (g == 1);
   }
   return g;
}
```

4.2. Java

4.2.1. Probando división

```
public static ArrayList<Long> division1(long n) {
   ArrayList<Long> factorization = new ArrayList<Long>();
   for (long d = 2; d * d <= n; d++) {
      while (n % d == 0) {
         factorization.add(d);
         n /= d;
      }
   }
   if (n > 1) factorization.add(n);
   return factorization;
}
```

4.2.2. Factorización de rueda

```
public static ArrayList<Long> wheel(long n) {
   ArrayList<Long> factorization = new ArrayList<Long>();
   while (n % 2 == 0) {
      factorization.add(2L);
      n /= 2;
   }
   for (long d = 3; d * d <= n; d += 2) {
```

```
      while (n % d == 0) {
         factorization.add(d);
         n /= d;
      }
   }
   if (n > 1) factorization.add(n);
   return factorization;
}

ArrayList<Long> wheelv2(long n) {
   ArrayList<Long> factorization = new ArrayList<Long>();
   int [] pr = {2, 3, 5};
   for (int d : pr) {
      while (n % d == 0) {
         factorization.add((long)d);
         n /= d;
      }
   }
   int [] increments = {4, 2, 4, 2, 4, 6, 2, 6};
   int i = 0;
   for (long d = 7; d * d <= n; d += increments[i++]) {
      while (n % d == 0) {
         factorization.add(d);
         n /= d;
      }
      if (i == 8) i = 0;
   }
   if (n > 1) factorization.add(n);
   return factorization;
}
```

4.2.3. Primos precalculados

```
public static ArrayList<Long> trial_division4(long n, ArrayList<Long> primes)
    {
   ArrayList<Long> factorization = new  ArrayList<Long>();
   for (long d : primes) {
      if (d * d > n) break;
      while (n % d == 0) {
         factorization.add(d);
         n /= d;
      }
   }
   if (n > 1) factorization.add(n);
   return factorization;
}
```

4.2.4. Factorización de N

En las dos implementaciones siguientes es evidente que se necesita precalcular todos los primos al menos hasta N y tenerlos almacenados en el vector *primos*

```
public static Map<Integer, Integer> factorizar(int n, ArrayList<Integer>
    primos){
   int l = primos.size();
   //iterar sobre todos los primos
   //hasta que algun primo sea mayor que n
   Map<Integer, Integer> factors = new LinkedHashMap<>();
   for(int i=0;i<l && primos.get(i)<=n ;i++){
      int pi = primos.get(i);
      int exp = 0;
      if(n%pi==0){
         while(n%pi==0){
            exp++;//contador para el exponente
            n/=pi;//divido entre la base
         }
         // pi^exp
         // {a,b}  es lo mismo que make_pair(a,b) o Pair(a,b)
         factors.put(pi,exp);
      }
    }
        // si n es distinto n es un primo
        if(n>1) factors.put(n,1);
        return factors;
}
```

4.2.5. Factorización de N!

```
public static Map<Integer, Integer> factorizarFactorial(int n, ArrayList<
    Integer> primos){
   int l = primos.size();
   Map<Integer, Integer> factors = new LinkedHashMap<>();
   //iterar sobre todos los primos menores e iguales que n
   for(int i=0;i<l && primos.get(i)<=n;i++){
      //acumulador para la potencia del primo actual
      int exp = 0;
      //potencia del primo actual
      int pot = primos.get(i);
      //mientras la potencia sea menor que n
      while(pot<=n){
         //multiplos de la potencia menores que n
         exp+=n/pot;
         //proxima potencia del primo
         pot*=primos.get(i);
      }
```

```
            factors.put(primos.get(i),exp);
        }
        return factors;
    }
```

4.2.6. Método de factorización de Fermat

```
public static int fermat(int n) {
    int a = (int)Math.ceil(Math.sqrt(n));
    int b2 = a*a - n;
    int b = (int)Math.round(Math.sqrt(b2));
    while (b * b != b2) {
        a = a + 1;
        b2 = a*a - n;
        b = (int)Math.round(Math.sqrt(b2));
    }
    return a - b;
}
```

4.2.7. Método de Pollard $P-1$

```
public static long pollards_p_minus_1(long n) {
    int B = 10;
    long g = 1;
    while (B <= 1000000 && g < n) {
        Random rnd = new Random();
        long a = 2 + rnd.nextLong((n - 3))  ;
        g = gcd(a, n);
        if (g > 1) return g;

        // calcular a^M
        for (int p : primes) {
            if (p >= B) continue;
            long p_power = 1;
            while (p_power * p <= B) p_power *= p;
            a = power(a, p_power, n);
            g = gcd(a - 1, n);
            if (g > 1 && g < n) return g;
        }
        B *= 2;
    }
    return 1;
}
```

4.2.8. Algoritmo de ρ de Pollard

```
public static long mult(long a, long b, long mod) {
   long result = 0;
   while (b!=0) {
      if ((b & 1) == 1) result = (result + a) % mod;
      a = (a + a) % mod;
      b >>= 1;
   }
   return result;
}

public static long f(long x, long c, long mod) {
   return (mult(x, x, mod) + c) % mod;
}

//Cuando se invoque x0=2 y c=1
public static long rho(long n, long x0, long c) {
   long x = x0; long y = x0; long g = 1;
   while (g == 1) {
      x = f(x, c, n);
      y = f(y, c, n);
      y = f(y, c, n);
      g = gcd(Math.abs(x - y), n);
   }
   return g;
}
```

4.2.9. Algoritmo de Brent

```
//Cuando se invoque al metodo x0=2 y c=1
public static long brent(long n, long x0, long c) {
   long x = x0; long g = 1;
   long q = 1; long xs, y;
   int m = 128; int l = 1;

   while (g == 1) {
      y = x;
      for (int i = 1; i < l; i++) x = f(x, c, n);
      int k = 0;
      while (k < l && g == 1) {
         xs = x;
         for (int i = 0; i < m && i < l - k; i++) {
            x = f(x, c, n);
            q = mult(q, Math.abs(y - x), n);
         }
         g = gcd(q, n);
         k += m;
      }
      l *= 2;
```

```
    }
    if (g == n) {
        do{
            xs = f(xs, c, n);
            g = gcd(Math.abs(xs - y), n);
        } while (g == 1);
    }
    return g;
}
```

5. Aplicaciones

La factorización de enteros tiene diversas aplicaciones en diferentes campos. Algunas de las aplicaciones más comunes de la factorización de enteros son:

1. **Seguridad en criptografía:** La factorización de enteros es fundamental en la criptografía asimétrica, específicamente en el algoritmo RSA (Rivest-Shamir-Adleman). En RSA, la seguridad del sistema se basa en la dificultad computacional de factorizar grandes números enteros en sus factores primos. Si un atacante logra factorizar el número utilizado en la clave pública, podría comprometer la seguridad del sistema.
2. **Algoritmos de optimización:** En algunos problemas de optimización combinatoria, como el problema del viajante o el problema del corte máximo, la factorización de enteros puede utilizarse para encontrar soluciones óptimas o aproximadas.
3. **Matemáticas puras:** La factorización de enteros es un problema matemático interesante por sí mismo y ha sido objeto de estudio durante siglos. La teoría de números se ocupa de propiedades relacionadas con los números enteros, incluida la factorización.
4. **Compresión de datos:** En la compresión de datos, a veces se utilizan técnicas que involucran la factorización de enteros para reducir el tamaño de los datos sin perder información importante.
5. **Resolución de problemas prácticos:** En situaciones cotidianas, la factorización de enteros puede ser útil para descomponer números grandes en sus factores primos con el fin de simplificar cálculos o entender mejor la estructura de los números.

Estas son solo algunas de las aplicaciones de la factorización de enteros. Es una herramienta poderosa y versátil que se utiliza en una variedad de contextos, desde la seguridad informática hasta las matemáticas puras.

6. Complejidad

La prueba de división nos da la factorización prima en O($\sqrt{n}$). (Este es un tiempo pseudopolinomial , es decir, polinomio en el valor de la entrada pero exponencial en el número de bits de la entrada).

El método de factorización de Fermat puede ser muy rápido si la diferencia entre los dos factores p y q es pequeño. El algoritmo se ejecuta en O($|p - q|$) tiempo. Sin embargo, en la práctica este método rara vez se utiliza. Una vez que los factores se alejan más, es extremadamente lento.

Sin embargo, todavía existe una gran cantidad de opciones de optimización con respecto a este enfoque. Mirando los cuadrados a^2 módulo un número pequeño fijo, se puede observar que ciertos valores a no es necesario verlos, ya que no pueden producir un número cuadrado $a^2 - n$.

La complejidad del método Pollard $P - 1$ es O($B \log B \log^2 n$) por iteración.

El algoritmo Floyd generalmente se ejecuta en O($\sqrt[4]{n} \log(n)$) tiempo.

El algoritmo de Brent se ejectua en tiempo lineal, pero generalmente es más rápido que el de Floyd, ya que utiliza menos evaluaciones de la función f.

Observe que si el número que desea factorizar es en realidad un número primo, la mayoría de los algoritmos se ejecutarán muy lentamente. Esto es especialmente cierto para los algoritmos de factorización de Fermat, Pollard p-1 y Rho de Pollard. Por lo tanto, tiene más sentido realizar una prueba de primalidad probabilística (o determinista rápida) antes de intentar factorizar el número.

7. Ejercicios

A continuación una lista de ejercicios que la base de su solución es saber descomponer en factores primos un número.

- DMOJ - Neuronas en Acción
- DMOJ - Firmas Primas
- SPOJ - FACT0
- SPOJ - FACT1
- SPOJ - FACT2
- GCPC 15 - Divisions