GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: NÚMEROS PRIMOS

1. Introducción

En matemáticas, particularmente en teoría de números o aritmética existen problemas que se relacionan con los números primos o la solución de estos radica en saber identificar los números que pertenecen a dicho conjunto.

En la presente guía de aprendizaje abordaremos algoritmos para determinar si un número es primo o no así como generar todos los números primos comprendido en rango de 1 a N donde N no supera el valor de 10^7.

2. Conocimientos previos

Lo primero para continuar seguir leyendo es conocer que número primo es cualquier número natural mayor que 1 cuyo únicos divisores posibles son el mismo número primo y el factor natural 1 .A diferencia de los números primos , los números compuestos son naturales que pueden factorizarse. Ejemplo de números primos son el 2, 3, 5, 7, 11, 13. En algunas literaturas se consideran el 1 como primo, para el caso que nos ocupa no se va a considerar el 1 como no primo pero ojo cuando se enfrente a los problemas que abordan esta temática pueden encontrar algunos que lo consideran.

3. Desarrollo

3.1. Saber si un número es primo o no

En lo primero que vamos analizar es como demostra si dado un número N este es primo o no. Partiendo de la misma defición se podría idear un algoritmo que itere por todos los numeros naturales en el rango de 2 a $N-1$ incluyendo los extremos del intervalo si en dicho rango se encuentra al menos un número X que sea divisor exacto de N entonces se podría decir que N no es primo mientras si despues de iterar por todo el rango no se encuentra ningún valor que sea divisor exacto entonces se puede tener la seguridad de que N es primo.

Ahora podemos mejorar o acotar el rango de búsqueda de los supuestos divisores exacto de N. Una primera idea pudiera ser hasta $N/2$ ya que cualquier par (a, b) tal que $a * b == N$ unos de esos dos valores va ser menor o igual a $N/2$ mientras el otro va ser mayor. Pero pudieramos acotar aún mas ese rango ?. Pues si lo podemos acotar hasta $\sqrt{N}$ de esta forma si N no tiene divisores exacto en el rango de 2 a $\sqrt{N}$ podemos afirmar con seguridad que N es primo.

Lo anterior nos permite tener un algoritmo para chequear si un número no mayor que 10^{12} sea primo. Pero que hacer cuando el número es mayor ?.

3.2. Test de primalidad

La cuestión de la determinación de si un número N dado es primo es conocida como el problema de la primalidad. Un test de primalidad (o chequeo de primalidad) es un algoritmo que, dado

un número de entrada n, no consigue verificar la hipótesis de un teorema cuya conclusión es que n es compuesto.

Esto es, un test de primalidad sólo conjetura que *ante la falta de certificación sobre la hipótesis de que* n *es compuesto podemos tener cierta confianza en que se trata de un número primo*. Esta definición supone un grado menor de confianza que lo que se denomina prueba de primalidad (o test verdadero de primalidad), que ofrece una seguridad matemática al respecto.

3.2.1. Test de primalidad de Miller-Rabin

El Test de primalidad de Miller-Rabin es un test de primalidad, es decir, un algoritmo para determinar si un número dado es primo, similar al test de primalidad de Fermat. Su versión original fue propuesta por G. L. Miller, se trata de un algoritmo determinista, pero basado en la no demostrada hipótesis generalizada de Riemann; Michael Oser Rabin modificó la propuesta de Miller para obtener un algoritmo probabilístico incondicional.

Supóngase que $n > 1$ es un número impar del cual queremos saber si es primo o no. Sea m un valor impar tal que $n - 1 = 2^k m$ y a un entero escogido aleatoriamente entre 2 y $n - 2$.

Cuando se cumple que:

$a^m \equiv \pm 1 (mod n)$

o bien

$a^{2^r m} \equiv -1 (mod n)$

para al menos un r entero entre 1 y $k - 1$, se considera que n es un probable primo; en caso contrario n no puede ser primo. Si n es un probable primo se escoge un nuevo valor para a, y se itera nuevamente reduciendo el margen de error probable. Al utilizar exponenciación binaria las operaciones necesarias se realizan muy rápidamente.

Se puede demostrar que un número compuesto es clasificado *probable primo* en una iteración del algoritmo con una probabilidad inferior a $1/4$; de hecho, en la práctica la probabilidad es mucho menor.

3.2.2. Prueba de primalidad de Fermat

El pequeño teorema de Fermat establece que para un número primo p y un entero coprimo a se cumple la siguiente ecuación:

$$a^{p-1} \equiv 1 \text{ mód } p$$

En general, este teorema no se cumple para números compuestos.

Esto se puede utilizar para crear una prueba de primalidad. Elegimos un entero $2 \leq a \leq p - 2$, y verificamos si la ecuación se cumple o no. Si no se sostiene, p. $a^{p-1} \not\equiv 1 \text{ mód } p$, sabemos que p

no puede ser un número primo. En este caso llamamos a la base a un testigo de Fermat para la composición de p.

Sin embargo, también es posible que la ecuación se cumpla para un número compuesto. Entonces, si la ecuación se cumple, no tenemos una prueba de primalidad. Solo podemos decir que p es probablemente primo. Si resulta que el número es realmente compuesto, llamamos a la base a un mentiroso de Fermat.

Al ejecutar la prueba para todas las bases posibles a, podemos demostrar que un número es primo. Sin embargo, esto no se hace en la práctica, ya que requiere mucho más esfuerzo que solo hacer *división de prueba*. En su lugar, la prueba se repetirá varias veces con opciones aleatorias para a. Si no encontramos ningún testigo de la composición, es muy probable que el número sea primo.

Sin embargo, hay una mala noticia: existen algunos números compuestos donde $a^{n-1} \equiv 1 \pmod{n}$ vale para todos los a coprimos a n, por ejemplo para el número $561 = 3 \cdot 11 \cdot 17$. Tales números se llaman números de Carmichael. La prueba de primalidad de Fermat solo puede identificar estos números, si tenemos mucha suerte y elegimos una base a con $\gcd(a, n) \neq 1$.</p>

La prueba de Fermat todavía se usa en la práctica, ya que es muy rápida y los números de Carmichael son muy raros. P.ej. solo existen 646 de esos números por debajo de 10^9.

3.3. Calcular todos los primos hasta N

La idea es lograr calcular todos los primos comprendidos en el rango de 1 a N, para lograr esto vamos a ver algunos algoritmos de como lograrlo

3.3.1. Cibra de Eratóstenes

La criba de Eratóstenes es un algoritmo que permite hallar todos los números primos menores que un número natural dado N. Se forma una tabla con todos los números naturales comprendidos entre 2 y n, y se van tachando los números que no son primos de la siguiente manera: Comenzando por el 2, se tachan todos sus múltiplos; comenzando de nuevo, cuando se encuentra un número entero que no ha sido tachado, ese número es declarado primo, y se procede a tachar todos sus múltiplos, así sucesivamente. El proceso termina cuando el cuadrado del mayor número confirmado como primo es mayor que n.

Un refinamiento de la criba consiste en tachar los múltiplos del k-ésimo número primo pk, comenzando por pk2 pues en los anteriores pasos se habían tachado los múltiplos de pk correspondientes a todos los anteriores números primos, esto es, 2pk, 3pk, 5pk,..., hasta (pk-1)pk. El algoritmo acabaría cuando p^2k>n ya que no habría nada que tachar.

3.3.2. Cibra de Atkin

La criba de Atkin es un algoritmo rápido y moderno empleado en matemática para hallar todos los números primos menores o iguales que un número natural dado. Es una versión optimizada de la criba de Eratóstenes, pero realiza algo de trabajo preliminar y no tacha los múltiplos de los

números primos, sino concretamente los múltiplos de los cuadrados de los primos. Fue ideada por A. O. L. Atkin y Daniel J. Bernstein.

Así funciona el algoritmo:

- Todos los restos son módulo 60, es decir, se divide el número entre 60 y se toma el resto.
- Todos los números, incluidos x e y, son enteros positivos.
- Invertir un elemento de la lista de la criba significa cambiar el valor ("primos"o "no primos") al valor opuesto.

1. Crear una lista de resultados, compuesta por 2, 3 y 5.
2. Crear una lista de la criba con una entrada por cada entero positivo; todas las entradas deben marcarse inicialmente como "no primos".
3. Para cada entrada en la lista de la criba:
   - Si la entrada es un número con resto 1, 13, 17, 29, 37, 41, 49 ó 53, se invierte tantas veces como soluciones posibles hay para $4x^2 + y^2$ = entrada.
   - Si la entrada es un número con resto 7, 19, 31 ó 43, se invierte tantas veces como soluciones posibles hay para $3x^2 + y^2$ = entrada.
   - Si la entrada es un número con resto 11, 23, 47 ó 59, se invierte tantas veces como soluciones posibles hay para $3x^2 - y^2$ = entrada con la restricción $x > y$.
   - Si la entrada tiene otro resto, se ignora.
4. Se empieza con el menor número de la lista de la criba.
5. Se toma el siguiente número de la lista de la criba marcado como "primos".
6. Se incluye el número en la lista de resultados.
7. Se eleva el número al cuadrado y se marcan todos los múltiplos de ese cuadrado como "no primos".
8. Repetir los pasos 5 a 8.

El algoritmo ignora cualquier número divisible por 2, 3 ó 5. Todos los números con resto, módulo 60, igual a 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56 ó 58 son pares y por tanto compuestos. Los de resto 3, 9, 15, 21, 27, 33, 39, 45, 51 ó 57 son divisibles por 3 y por tanto compuestos. Finalmente, los de resto 5, 25, 35 ó 55 son divisibles entre 5 y por tanto compuestos.Todos estos restos son ignorados.

Todos los números con resto, módulo 60, igual a 1, 13, 17, 29, 37, 41, 49 ó 53 tienen un resto, módulo 4, de 1. Estos números son primos si y sólo si el número de soluciones de $4x^2 + y^2$ = n es impar y el número es libre de cuadrados.

Todos los números con resto, módulo 60, igual a 7, 19, 31 ó 43 tienen un resto, módulo 6, de 1. Estos números son primos si y sólo si el número de soluciones de $3x^2 + y^2$=n es impar y el número

es libre de cuadrados.

Todos los números con resto, módulo 60, de 11, 23, 47 ó 59 tienen un resto, módulo 12, de 11. Estos números son primos si y sólo si el número de soluciones de $3x^2$ - y^2= n es impar y el número es libre de cuadrados.

Ninguno de los candidatos a primos es divisible entre 2, 3 ó 5, por lo que no puede ser divisible entre sus cuadrados. Esta es la razón por la que las comprobaciones de si un número es libre de cuadrados no incluyen los casos 2^2, 3^2 y 5^2.

3.3.3. Cibra de lineal

Aunque hay muchos algoritmos conocidos con tiempo de ejecución sublineal (es decir,$O(N)$), el algoritmo descrito a continuación es interesante por su simplicidad: no es más complejo que el clásico criba de Eratóstenes.

Además, el algoritmo dado aquí calcula factorizaciones de todos los números en el segmento$[2, N]$ como efecto secundario, y que puede ser útil en muchas aplicaciones prácticas.

La debilidad del algoritmo dado es que usa más memoria que la criba clásica de Eratóstenes: requiere una matriz de números, mientras que para criba clásico de Eratóstenes basta con tener bits de memoria (que es 32 veces menos).

Por lo tanto, tiene sentido usar el algoritmo descrito solo hasta que para números de orden y no mayor 10^7.

La autoría del algoritmo parece pertenecer a Gries & Misra (Gries, Misra, 1978: ver referencias al final del artículo). Sin embargo, también se puede atribuir a Euler, y también se le conoce como el tamiz de Euler, quien ya utilizó una versión similar durante su trabajo.

El algoritmo se centra en calcular el factor primo mínimo para cada número en el segmento $[2, N]$.

Además, necesitamos almacenar la lista de todos los números primos encontrados, llamémoslo $pr[]$

Inicializaremos los valores $lp[i]$ con ceros, lo que significa que asumimos que todos los números son primos. Durante la ejecución del algoritmo, este vector se llenará gradualmente.

Ahora repasaremos los números del 2 al N. Tenemos dos casos para el número actual i:

1. $lp[i] = 0$ - eso significa que i es primo, es decir, no hemos encontrado factores menores para él. Por lo tanto, asignamos $lp[i] = i$ y agregamos i al final de la lista $pr[]$.
2. $lp[i]! = 0$ - eso significa que i es compuesto y que su minimo factor primo es $lp[i]$

En ambos casos actualizamos los valores $lp[]$ de para los números que son divisibles por i. Sin embargo, nuestro objetivo es aprender a hacerlo para establecer un valor como máximo una vez para cada número $lp[]$. Podemos hacerlo de la siguiente manera:

Consideremos los números $x_j = i \cdot p_j$ donde p_j son todos los números primos menores o iguales que $lp[i]$ (por eso necesitamos almacenar la lista de todos los números primos).

Estableceremos un nuevo valor $lp[x_j] = p_j$ para todos los números de esta forma.

Aunque el tiempo de ejecución de $O(n)$ es mejor que $O(n \log \log n)$ de la criba clásica de Eratóstenes, la diferencia entre ambos no es tan grande. En la práctica, la criba lineal funciona tan rápido como una implementación típica de la criba de Eratóstenes.

En comparación con las versiones optimizadas del tamiz de Eratóstenes, p. el tamiz segmentado, es mucho más lento.

Teniendo en cuenta los requisitos de memoria de este algoritmo: un vector $lp[]$ de longitud n y un vector de $pr[]$ de longitud $\frac{n}{\ln n}$, este algoritmo parece peor que la criba clásicá en todos los sentidos.

Sin embargo, su cualidad redentora es que este algoritmo calcula un vector $lp[]$, lo que nos permite encontrar la factorización de cualquier número en el segmento $[2; n]$ en el tiempo del orden de tamaño de esta factorización. Además, usar solo una matriz adicional nos permitirá evitar divisiones al buscar factorización.

Conocer las factorizaciones de todos los números es muy útil para algunas tareas, y este algoritmo es uno de los pocos que permiten encontrarlos en tiempo lineal.

4. Implementación

4.1. C++

La implementacion de saber si es primo chequeando todos los numeros en el rango de 2 hasta $\sqrt{N}$:

```
bool isPrime(int x) {
   for (int d = 2; d * d <= x; d++) {
      if (x % d == 0)
         return false;
   }
   return true;
}
```

La implementación del test primalidad Miller-Rabin en C++:

```
        int powMod (int x, int k, int m) {
                if (k == 0) return 1;
                if (k % 2 == 0) return powMod (x*x %m, k/2, m);
                else return x * powMod (x, k-1, m) % m;
        }

        bool suspect (int a, int s, int d, int n) {
```

```
        int x = powMod (a, d, n);
        if (x == 1) return true;
        for (int r = 0; r<s; ++r){
                if (x == n - 1) return true;
                x = x*x%n;
        }
        return false;
}

// {2, 7, 61, - 1} para n <4759123141 (= 2 ^ 32)
// {2, 3, 5, 7, 11, 13, 17, 19, 23, - 1} para n <10 ^ 16 (hasta el
   momento)
bool isPrime (int n) {
        if (n<=1 || (n>2 && n%2==0)) return false;
        int test [] = {2, 3, 5, 7, 11, 13, 17, 19, 23, - 1};
        int d = n - 1, s = 0;
        while (d%2==0) ++s, d/= 2;

        for (int i = 0; test[i] <n && test [i]!=-1; ++i)
        if (!suspect(test[i],s,d,n)) return false;
        return true;
}

// ----- Otra implementacion

using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64 mod) {
        u64 result = 1;
        base %= mod;
        while (e) {
                if (e & 1) result = (u128)result * base % mod;
                base = (u128)base * base % mod;
                e >>= 1;
        }
        return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
        u64 x = binpower(a, d, n);
        if (x == 1 || x == n - 1) return false;
        for (int r = 1; r < s; r++) {
                x = (u128)x * x % n;
                if (x == n - 1) return false;
        }
        return true;
};

bool MillerRabin(u64 n) {
```

```
        if (n < 4) return n == 2 || n == 3;
        int s = 0;
        u64 d = n - 1;
        while ((d & 1) == 0) {
                d >>= 1;
                s++;
        }

        for (int i = 0; i < iter; i++) {
                int a = 2 + rand() % (n - 3);
                if (check_composite(n, a, d, s)) return false;
        }
        return true;
}

// -----Otra implementacion

bool MillerRabin(u64 n) {
        if (n < 2) return false;
        int r = 0;
        u64 d = n - 1;
        while ((d & 1) == 0) {
                d >>= 1;
                r++;
        }

        for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
                if (n == a) return true;
                if (check_composite(n, a, d, r)) return false;
        }
        return true;
}
```

Implementación de Test de Fermat en C++:

```
int binpow(int a, int b) {
        int res = 1;
        while (b > 0) {
                if (b & 1)res = res * a;
                a = a * a;
                b >>= 1;
        }
        return res;
}

bool probablyPrimeFermat(int n, int iter=5) {
        if (n <= 4) return n == 2 || n == 3;

        for (int i = 0; i &lt; iter; i++) {
```

```
                int a = 2 + rand() % (n - 3);
                if (binpow(a, n - 1, n) != 1) return false;
        }
        return true;
}
```

Veamos algunas de las posibles implementaciones de la criba de Eratóstenes

```
#define MAX_N 1000001
#include <vector>

vector<int> primes;
bool mark[MAX_N];

inline void sieve(int B){
        if (B > 1) primes.push_back(2);
        for (int i=3;i<=B;i+=2){
                if(!mark[i]){
                        mark[i]=true;
                        primes.push_back(i);
                        if (i<=sqrt(B)+1)
                        for (int j=i*i;j<=B;j+=i)
                        mark[j]=true;
                }
        }
}

/* Otra implementacion pero devuelve un vector donde si
primes[i]!=0 entonces i es un numero primo es la siguiente*/

vector <int> sieve_of_eratosthenes (int n) {
        vector <int> primes (n);
        for (int i = 2; i <n; ++i)
        primes [i] = i;
        for (int i = 2; i*i<n; ++ i)
        if (primes [i])
        for (int j = i*i; j <n; j+=i)
        primes [j] = 0;
        return primes;
}
```

Ahora algunas implementaciones de la criba de Atkin:

```
void sieve_of_atkin (){
        int n;
        for (int z = 1; z <= 5; z+=4) {
                for (int y = z; y<= sqrt(N); y+=6) {
                        for (int x = 1; x <= sqrt(N) && (n=4*x*x+y*y)
                            <= N; ++x)
```

```
                        isprime [n]=!isprime[n];
                        for (int x = y + 1; x <= sqrt(N) && (n=3*x*x-y
                            *y)<=N; x+=2)
                        isprime[n]=!isprime[n];
                }
        }

        for (int z = 2; z<= 4; z+=2) {
                for (int y = z; y<=sqrt(N); y +=6) {
                        for (int x = 1; x<=sqrt(N) && (n=3*x*x+y*y)<=N
                            ;x+=2)
                        isprime[n]=!isprime[n];
                        for (int x = y+1;x<=sqrt(N) && (n=3*x*x-y*y)<=
                            N;x+=2)
                        isprime[n]=!isprime[n];
                }
        }

        for (int y = 3; y <= sqrt(N); y+=6){
                for (int z = 1; z<=2; ++z){
                        for (int x = z; x<=sqrt(N) && (n=4*x*x+y*y)<=N
                            ; x+= 3)
                        isprime[n]=!isprime[n];
                }
        }

        for (int n=5; n<=sqrt(N); ++n)
        if (isprime[n])
        for (int k=n*n; k<=N; k+=n*n)
        isprime[k]=false;
        isprime[2]=isprime[3]=true;
}

/* Otra implementacion */

void sieve_of_atkin (){
        int n;
        for (int x = 1; x<=sqrt(N); ++x) {
                for (int y = 1; y<=sqrt(N); ++y) {
                        n= 4*x*x+y*y;
                        if(n <=N && (n%12==1 || n%12==5))
                        isprime[n]=!isprime[n];
                        n=3*x*x+y*y;
                        if(n<=N && n%12==7)
                        isprime[n]=!isprime[n];
                        n=3*x*x-y*y;
                        if(x>y && n<=N && n%12==11)
                        isprime[n]=!isprime[n];
                }
        }
```

```
            for (int n = 5; n<=sqrt(N); ++n)
            if (isprime[n])
            for (int k = n*n; k<=N; k+=n*n)
            isprime[k]=false;
            isprime[2]=isprime[3]=true;
        }
```

Implementación de la criba lineal

```
        const int N = 10000000;
        vector<int> lp(N+1);
        vector<int> pr;

        for(int i=2; i <= N; ++i) {
                if(lp[i] == 0) {
                        lp[i] = i;
                        pr.push_back(i);
                }
                for(int j=0; j < (int)pr.size() && pr[j] <= lp[i] && i*pr[j]
                    <= N; ++j) {
                        lp[i * pr[j]] = pr[j];
                }
        }
```

4.2. Java

La implementacion de saber si es primo chequeando todos los numeros en el rango de 2 hasta $\sqrt{N}$:

```
public static boolean isPrime(long n) {
   if (n <= 1) return false;
   for (long i = 2; i * i <= n; i++)
      if (n % i == 0) return false;
   return true;
}
```

Otra implementaciones realizadas en Java que son optimizaciones de las cribas

```
// Generates prime numbers up to n in O(n*log(log(n))) time
public static int[] generatePrimes(int n) {
   boolean[] prime = new boolean[n + 1];
   Arrays.fill(prime, 2, n + 1, true);

   for (int i = 2; i * i <= n; i++)
      if (prime[i])
         for (int j = i * i; j <= n; j += i) prime[j] = false;
   int[] primes = new int[n + 1];
```

```
    int cnt = 0;
    for (int i = 0; i < prime.length; i++)
        if (prime[i]) primes[cnt++] = i;
    return Arrays.copyOf(primes, cnt);
}

// Generates prime numbers up to n in O(n) time
public static int[] generatePrimesLinearTime(int n) {
    int[] lp = new int[n + 1];
    int[] primes = new int[n + 1];
    int cnt = 0;
    for (int i = 2; i <= n; ++i) {
        if (lp[i] == 0) {
            lp[i] = i;
            primes[cnt++] = i;
        }
        for (int j = 0; j < cnt && primes[j] <= lp[i] && i * primes[j] <= n; ++j
            ) lp[i * primes[j]] = primes[j];
    }
    return Arrays.copyOf(primes, cnt);
}

public static boolean isPrime(long n) {
    if (n <= 1)return false;
    for (long i = 2; i * i <= n; i++)
        if (n % i == 0) return false;
    return true;
}
```

5. Complejidad

Para la primera idea de chequear si un numero es primo o la complejidad es $O\sqrt{N}$

En el caso del test Miller-Rabin asumiendo correcta la hipótesis generalizada de Riemann, se puede demostrar que, si todo valor de a hasta $2(\ln n)^2$ ha sido verificado y n todavía es clasificado como probable primo, entonces n es en realidad un número primo. Con esto se tiene un test de primalidad de costo O($(\ln n)^4$).

La complejidad del algoritmo de la criba de Eratóstenes es O($N \log \log N$) lo que hace que sea un algoritmo bastante rapido. El problema del algoritmo radica en el costo de memoria ya que necesita un arreglo de N+1 elementos siendo N el máximo número hasta donde deseamos conocer todos los números primos. Por lo que el algoritmo solo es aconsejable usarlo cuando N no es mayor de 10^6, para un intervalo mayor se puede este algoritmo junto con otras tecnicas como los test de Primalidad.

En cuanto a la complejidad de la criba de Atkin la primera implementación tiene una complejidad de O (n / log log n). Mientras en la segunda variante la complejidad es de O(n)

6. Aplicaciones

Los números primos son utilizados en diferentes tipos de problemas o ejercicios, en algunos son la parte fundamental de a solución porque el problema trata propiamente de ellos, en otros son las parte de la base o sireven de apoyo para elaborar una solución como puede ser los problemas de hashing, factorización, cantidad de divisores, inverso multiplicativo.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando aplicando los conocimientos abordados en esta guía:

- CodeForce - T-primes
- CodeForce - Prime Matrix
- Codeforce - Design Tutorial: Learn from Math
- Codeforce - Almost Prime
- DMOJ - Matriz prima
- DMOJ - Números Primos de nuevo
- DMOJ - Conjeturas de Goldbacha
- SPOJ - Prime or Not
- Codeforces - Sherlock And His Girlfriend
- Codeforces - Nodbach Problem
- Codefoces - Colliders