# GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO EXTENDIDO DE EUCLIDES

# 1. Introducción

Mientras que el algoritmo euclidiano calcula sólo el máximo común divisor (GCD) de dos números enteros $a$ y $b$ , la versión extendida también encuentra una manera de representar GCD en términos de $a$ y $b$ , es decir, coeficientes $x$ y $y$ para cual:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Es importante señalar que por la identidad de Bézout siempre podemos encontrar tal representación. Por ejemplo, $\gcd(55, 80) = 5$, por lo tanto podemos representar $5$ como una combinación lineal con los términos $55$ y $80$ : $55 \cdot 3 + 80 \cdot (-2) = 5$

Una forma más general de ese problema se analiza en guía sobre Ecuaciones lineales diofánticas. Se basará en este algoritmo.

# 2. Conocimientos previos

## 2.1. Algoritmo de Euclides

El algoritmo de Euclides es un método eficiente para encontrar el máximo común divisor (gcd) de dos números enteros. El algoritmo se basa en la observación de que el gcd de dos números $a$ y $b$ es igual al gcd de $b$ y el residuo de dividir $a$ por $b$.

## 2.2. Identidad de Bézout

La identidad de Bézout es un resultado importante en teoría de números que establece lo siguiente: si $a$ y $b$ son dos enteros no nulos, entonces existen enteros $x$ e $y$ tales que $ax+by = \gcd(a, b)$, donde $\gcd(a, b)$ es el máximo común divisor de $a$ y $b$. Esta identidad es conocida como el teorema de Bézout en honor al matemático francés Étienne Bézout.

## 2.3. Ecuaciones lineales diofánticas

Una ecuación lineal diofántica es una ecuación de la forma $ax + by = c$, donde $a$, $b$, $c$ son enteros y queremos encontrar soluciones enteras para $x$ e $y$. Estas ecuaciones llevan el nombre del matemático griego Diofanto de Alejandría, quien estudió este tipo de ecuaciones en su obra.

# 3. Desarrollo

Denotaremos el GCD de $a$ y $b$ con $g$ a partir de ahora.

Los cambios al algoritmo original son muy simples. Si recordamos el algoritmo, podemos ver que el algoritmo termina con $b = 0$ y $a = g$. Para estos parámetros podemos encontrar fácilmente coeficientes, a saber $g \cdot 1 + 0 \cdot 0 = g$.

**Autor:** Luis Andrés Valido Fajardo
**Email:** luis.valido1989@gmail.com
**Teléfono:** 53694742

A partir de estos coeficientes $(x, y) = (1, 0)$ , podemos retroceder hacia arriba en las llamadas recursivas. Todo lo que tenemos que hacer es descubrir cómo los coeficientes $x$ y $y$ cambio durante la transición de $(a, b)$ a $(b, a \bmod b)$.

Supongamos que encontramos los coeficientes $(x_1, y_1)$ para $(b, a \bmod b)$ :

$$b \cdot x_1 + (a \bmod b) \cdot y_1 = g$$

y queremos encontrar la pareja $(x, y)$ para $(a, b)$:

$$a \cdot x + b \cdot y = g$$

podemos representar $a \bmod b$ como:

$$a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b$$

Sustituyendo esta expresión en la ecuación del coeficiente de $(x_1, y_1)$ da:

$$g = b \cdot x_1 + (a \bmod b) \cdot y_1 = b \cdot x_1 + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right) \cdot y_1$$

y después de reordenar los términos:

$$g = a \cdot y_1 + b \cdot \left(x_1 - y_1 \cdot \left\lfloor \frac{a}{b} \right\rfloor\right)$$

Encontramos los valores de $x$ y $y$:

$$\begin{cases} x = y_1 \\ y = x_1 - y_1 \cdot \lfloor \frac{a}{b} \rfloor \end{cases}$$

# 4. Implementación

## 4.1. Versión recursiva

La función recursiva siguiente devuelve el GCD y los valores de los coeficientes a $x$ y $y$ (que se pasan por referencia a la función). Esta implementación del algoritmo euclidiano extendido también produce resultados correctos para números enteros negativos.

#### 4.1.1. C++

```
int gcd_recursive(int a, int b, int& x, int& y) {
   if (b == 0) {
      x = 1; y = 0;
      return a;
   }
   int x1, y1;
   int d = gcd_recursive(b, a % b, x1, y1);
   x = y1;
   y = x1 - y1 * (a / b);
   return d;
}
```

#### 4.1.2. Java

```
// retorna { gcd(a,b), x, y } tal que gcd(a,b) = a*x + b*y
public static long[] gcd_recursive(long a, long b) {
   if (b == 0) return a > 0 ? new long[]{a, 1, 0} : new long[]{-a, -1, 0};
   long[] r = gcd_recursive(b, a % b);
   return new long[]{r[0], r[2], r[1] - a / b * r[2]};
}
```

### 4.2. Versión iterativa

También es posible escribir el algoritmo euclidiano extendido de forma iterativa. Como evita la recursividad, el código se ejecutará un poco más rápido que el recursivo.

Si observa detenidamente las variables $a_1$ y $b_1$ , podrá notar que toman exactamente los mismos valores que en la versión iterativa del algoritmo euclidiano normal . Entonces el algoritmo al menos calculará el GCD correcto.

Para ver por qué el algoritmo también calcula los coeficientes correctos, puede verificar que las siguientes invariantes se mantendrán en cualquier momento (antes del ciclo while y al final de cada iteración): $x \cdot a + y \cdot b = a_1$ y $x_1 \cdot a + y_1 \cdot b = b_1$ . Es trivial ver que estas dos ecuaciones se satisfacen al principio. Y puede comprobar que la actualización en la iteración del bucle seguirá manteniendo válidas esas igualdades.

Al final sabemos que $a_1$ contiene el GCD, por lo que $x \cdot a + y \cdot b = g$. Lo que significa que hemos encontrado los coeficientes requeridos.

Incluso puedes optimizar más el código y eliminar la variable $a_1$ y $b_1$ del código, y simplemente reutilizar $a$ y $b$. Sin embargo, si lo hace, perderá la capacidad de discutir sobre las invariantes.

#### 4.2.1. C++

```
int gcd_iterative(int a, int b, int& x, int& y) {
   x = 1, y = 0;
   int x1 = 0, y1 = 1, a1 = a, b1 = b;
   while (b1) {
      int q = a1 / b1;
      tie(x, x1) = make_tuple(x1, x - q * x1);
      tie(y, y1) = make_tuple(y1, y - q * y1);
      tie(a1, b1) = make_tuple(b1, a1 - q * b1);
   }
   sreturn a1;
}
```

#### 4.2.2. Java

```
// retorna { gcd(a,b), x, y } tal que gcd(a,b) = a*x + b*y
public static long[] gcd_iterative(long a, long b) {
   long x = 1, y = 0, x1 = 0, y1 = 1, t;
   while (b != 0) {
      long q = a / b; t = x;
      x = x1; x1 = t - q * x1;
      t = y; y = y1;
      y1 = t - q * y1; t = b;
      b = a - q * b; a = t;
   }
   return a > 0 ? new long[]{a, x, y} : new long[]{-a, -x, -y};
}
```

## 5. Aplicaciones

Algunas aplicaciones del algoritmo extendido de Euclides son:

1. **Encriptación RSA:** El algoritmo extendido de Euclides se utiliza en el proceso de generación de claves para el algoritmo de encriptación RSA, donde se necesitan encontrar números primos p y q y calcular la clave privada y pública.
2. **Resolución de ecuaciones diofánticas:** Las ecuaciones diofánticas son ecuaciones en las que se buscan soluciones enteras. El algoritmo extendido de Euclides se puede utilizar para encontrar soluciones enteras de ecuaciones diofánticas lineales.
3. **Reducción de fracciones:** El algoritmo extendido de Euclides se puede utilizar para simplificar fracciones, es decir, para encontrar la fracción irreducible equivalente a una fracción dada.
4. **Teoría de números:** El algoritmo extendido de Euclides es una herramienta fundamental en la teoría de números para estudiar propiedades de los números enteros, como la existencia de soluciones enteras para ciertas ecuaciones.

En resumen, el algoritmo extendido de Euclides tiene diversas aplicaciones en matemáticas y en campos como la criptografía y la teoría de números.

## 6. Complejidad

La complejidad temporal del algoritmo extendido de Euclides para encontrar el máximo común divisor (gcd) de dos números $a$ y $b$ es O($\log(\min(a, b))$), donde $\log$ representa el logaritmo en base 2 y $\min(a, b)$ es el menor de los dos números $a$ y $b$.

El algoritmo extendido de Euclides se basa en la recursión y en la división entera. En cada iteración, se realizan operaciones aritméticas simples como restas y divisiones enteras. Dado que en cada paso se reduce uno de los números $a$ o $b$ al menos a la mitad de su valor, el número de iteraciones necesarias para llegar al gcd es proporcional al logaritmo en base 2 del menor de los dos números.

Por lo tanto, la complejidad temporal del algoritmo extendido de Euclides es O($\log(\min(a, b))$). Esto significa que el tiempo de ejecución del algoritmo crece de forma logarítmica con respecto al tamaño de los números $a$ y $b$, lo cual lo hace muy eficiente para calcular el $\gcd$ de dos números enteros grandes.

## 7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver aplicando el algoritmo abordado en la presente guía:

- UVA - 10104 - Euclid Problem
- UVA - 12775 - Gift Dilemma
- GYM - (J) Once Upon A Time