Bezout's Identity

Bezout's Identity states that for any two integers a and b (not both zero), there exist integers x and y such that:

$ax + by = \gcd(a, b)$

Where $\gcd(a, b)$ is the greatest common divisor of a and b.

Extended Euclidean Algorithm

The Extended Euclidean Algorithm is an extension of the Euclidean Algorithm that not only finds the Greatest Common Divisor (GCD) of two integers, but also finds integers x and y (called Bézout coefficients) such that:

$$ax + by = \gcd(a, b)$$

Prerequisite: Euclidean Algorithm

First, let's understand the basic Euclidean Algorithm for finding GCD:

```
int gcd(int a, int b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}
```

How it works:

↳ If b = 0, GCD is a

↳ Otherwise, GCD(a, b) = GCD(b, a % b)

Extended Euclidean Algorithm

1. Base Case: The End of the Recursion

The standard Euclidean algorithm stops when b becomes 0. At this point:

gcd(a, 0) = a (by definition).

We need to find x and y such that a * x + 0 * y = a.

The simplest solution is:

x = 1 (because a * 1 = a)

y = 0 (because 0 * 0 = 0)

This is our anchor. We know the answer for this simplest case,
and we'll use it to build the answer for more complex cases on our way back up the recursive calls.

```
if (b == 0) {
    x = 1;
    y = 0;
    return a; // because gcd(a, 0) is a
}
```

2. The Recursive Case: Building the Solution Backwards

This is the core insight. Let's say we are currently solving for (a, b).

↳ We make a recursive call

for the next pair in the Euclidean algorithm: (b, a % b). We trust this recursive call to correctly give us three things:

g = gcd(b, a % b) (which is the same as gcd(a, b))

x1 and y1, such that:
b * x1 + (a % b) * y1 = g

We don't know how it does it yet, we just assume it works (this is the magic of recursion!).
Our job is now to use x1 and y1 to find our own x and y for the original pair (a, b).

```
int x1, y1; // Coefficients for the smaller problem
int g = gcd(b, a % b, x1, y1); // Get gcd and coefficients for (b, a % b)
```

↳ The Key: Rewriting the Equation

We have this equation from the recursive call:

```
b * x1 + (a % b) * y1 = g
```

We know that a % b can be rewritten using the division rule:

```
a % b = a - (a / b) * b
```

(Where a / b uses integer division, e.g., floor(7/3) = 2)

Let's substitute this into the equation from the recursive call:

```
b * x1 + (a - (a / b) * b) * y1 = g
```

↳ Rearrange to match the desired pattern

Let's expand the equation:

```
b * x1 + a * y1 - (a / b) * b * y1 = g
```

Now, group the terms with a and the terms with b:

```
a * y1 + b * (x1 - (a / b) * y1) = g
```

↳ Identify the new coefficients:

Look at the equation we just derived:

```
a * (y1) + b * (x1 - (a/b)*y1) = g
```

Compare it to the equation we want to satisfy for our current call:

```
a * (x) + b * (y) = g
```

It becomes clear what x and y must be:

```
x = y1

y = x1 - (a / b) * y1
```

```
// Now update our coefficients x and y for the current (a, b)
x = y1;
y = x1 - (a / b) * y1;
return g; // Return the GCD we got from the recursive call
```

Implementation

```
int extended_gcd(int a, int b, int &x, int &y) {

    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }

    int x1, y1;
    int gcd = extended_gcd(b, a % b, x1, y1);

    x = y1;
    y = x1 - (a / b) * y1;

    return gcd;
}
```

Time: O(log min(a,b))

Space: Recursive: O(log min(a,b)) (call stack)

Practical Applications

Modular Multiplicative Inverse: If gcd(a, m) = 1, then x is the modular inverse of a mod m

Solving Linear Diophantine Equations: Equations of the form ax + by = c

Cryptography: Used in RSA algorithm

Extended Euclidean Algorithm | $ax + by + cz = \gcd(a, b, c)$

We want to find integers x, y, z such that:
a*x + b*y + c*z = gcd(a, b, c)

The solution uses the important mathematical property:

$$\gcd(a, b, c) = \gcd(\gcd(a, b), c)$$

This means we can:

First find gcd(a, b) and the coefficients for a*x + b*y = gcd(a, b)

Then find gcd(gcd(a, b), c) and coefficients for gcd(a, b)*X + c*Z = gcd(a, b, c)

Combine these results to get the final coefficients

Step-by-Step Approach

Step 1: Solve for two variables

Find d = gcd(a, b) and coefficients u, v such that:
a*u + b*v = d

Step 2: Solve with the third variable

Now find g = gcd(d, c) = gcd(a, b, c) and coefficients X, Z such that:
d*X + c*Z = g

Step 3: Combine the results

Substitute the expression for d from Step 1 into Step 2:
(a*u + b*v)*X + c*Z = g

Which simplifies to:
a*(u*X) + b*(v*X) + c*Z = g

So our final coefficients are:

x = u * X
y = v * X
z = Z

Implementation

```
// Extended GCD for two numbers (from previous implementation)
int extended_gcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int d = extended_gcd(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return d;
}

// Extended GCD for three numbers
int extended_gcd_three(int a, int b, int c, int &x, int &y, int &z) {
    // Step 1: Find gcd(a, b) and coefficients u, v
    int u, v;
    int d = extended_gcd(a, b, u, v);

    // Step 2: Find gcd(d, c) and coefficients X, Z
    int X, Z;
    int g = extended_gcd(d, c, X, Z);

    // Step 3: Combine the results
    x = u * X;
    y = v * X;
    z = Z;

    return g;
}
```