Extended Euclidean Algorithm in C

By: Rodolfo “Ofo” Croes

Introduction

The extended Euclidean algorithm is an algorithm used to compute integers x and y such that ax + by = greatest common divisor of a and b. The difference between this algorithm and the Euclidean algorithm is that the Euclidean algorithm only calculates the gcd of two numbers, whereas the extended algorithm also calculates the multiplicative inverses of two numbers, as well as the gcd of the two numbers. This algorithm is particularly useful when a and b are coprime, this meaning that x is the modular multiplicative inverse of a modulo b, and y is the modular multiplicative inverse of b modulo a. In this report, I will demonstrate how this algorithm acquires the multiplicative inverses of given integers, how this can give you the answer for all x’s when ax is equivalent to congruent b (mod n), as well as how to implement this algorithm in the programming language C.

Extended Euclidean Algorithm

Given the integers x and y such that ax + by = greatest common divisor of a and b, we start by calculating the gcd(a,b) using the Euclidean Algorithm. This algorithm starts with the statement:

eqn 1. $$a = (q_0)b + p_0$$

where q_0 and p_0 are integers. Once we compute q_0 and p_0 and if $p_0 > 1$, we continue with the next calculation by shifting b and p_0 to get:

eqn 2. $$b = (q_1)\, p_0 + p_1.$$

We continue doing this shifting and calculating of q’s and p’s until the p after the + is equal to 0. Once we get to this point, we look to the previous step and see that the p after the + is equal to the gcd(a,b). Next, we use the equation where the gcd is found and rewrite it in the form:

eqn 3 . $$p_n = p_{n-2} - p_{n-1}(q_{n-1})$$

Where p_n = gcd(a,b) and n represents the steps that were taken by the Euclidean algorithm. Then we rewrite the p’s to make them all positive numbers before we begin backwards substitution:

eqn 4 . $$p_n = p_{n-2} + p_{n-1}(-q_{n-1}).$$

Once we find this, we use previous steps to backwards substitute the smallest value of p on the right side of the equation(in this case p_{n-1}) into the form:

eqn 5. $p_{n-3} - p_{n-2}(q_{n-2})$.

We simplify and continue substituting until we get the linear combination:

eqn 6. $p_n = b(B) + a(A)$

Where A and B are the multiplicative inverses of a and b respectively.

Solving ax is equivalent to congruent b mod n

When considering the extended Euclidean algorithm to solve when ax is equivalent to congruent b (mod n), we start by computing the gcd(a,n). This will gives us the amount of answers for x which can either be no solution, or 1-to-many solution(s). Once we find this out, we use the extended algorithm to find the multiplicative inverse of a which I will refer to as a^{-1}. If the amount of solutions is equal to 1, we calculate x using:

eqn 7. $x = (a^{-1} * b) \pmod n$

Thus giving us a single answer. If the gcd(a,n) is greater than one, we must take the given a, b and n and divide them by the gcd(a,n). This will gives us new values to run the extended algorithm with. This new gcd(a*, n*) will be equal to 1 and will give us a new a^{-1} when we apply the algorithm to said values. This new a^{-1} will be used, along with the new b and n to calculate the first value for x using equation 7. To find the next value(s) of x, we define the difference between each value of x as c where:

eqn 8. $c = n / \gcd(a,n)$

Using the original values for a and n. Then we add c to x to get a new unique value for x, and keep adding c to x until we have gcd(a,n) amount of unique solutions under mod n.

Implementation in C

The process to implement this algorithm in C starts with the machine taking in input from the user for a, b and n. Once this is acquired, it calculates the gcd(a, n) recursively (fig. 1). This is then returned to the main to see whether this computation has no solutions, or 1-to-many solutions. The program does not do any more computations if there is no solution and prints this out to the user. If there is one solution, the gcdExtended() function (fig. 2) is called once and returns the gcd, and the multiplicative inverses of a and n dynamically using the address' of given integers. The calculation is also done recursively. To find each of the multiplicative inverses, we start by considering invA and invB are results for inputs a and b in:

eqn 9. $a(invA) + b(invB) = gcd$

And A and B are results for inputs remainder of b/a and a in:

eqn 10. $(\text{remainder of } b/a)A + (a)B = gcd$

When we express (remainder of b/a) as $(b - (\lfloor b/a \rfloor) * a)$ we get:

eqn 11. $(b-(\lfloor b/a \rfloor)a)A + (a)B = gcd$

We expand the A into the remainder function to get:

eqn 12. $(b)A + (a)B - (\lfloor b/a \rfloor)A) = gcd$

Then, we compare the coefficients of a and b in equation 9 and equation 12 to get the following calculations for invA and invB:

eqn 13. $invA = B - \lfloor b/a \rfloor * A$

eqn 14. $invB = A$

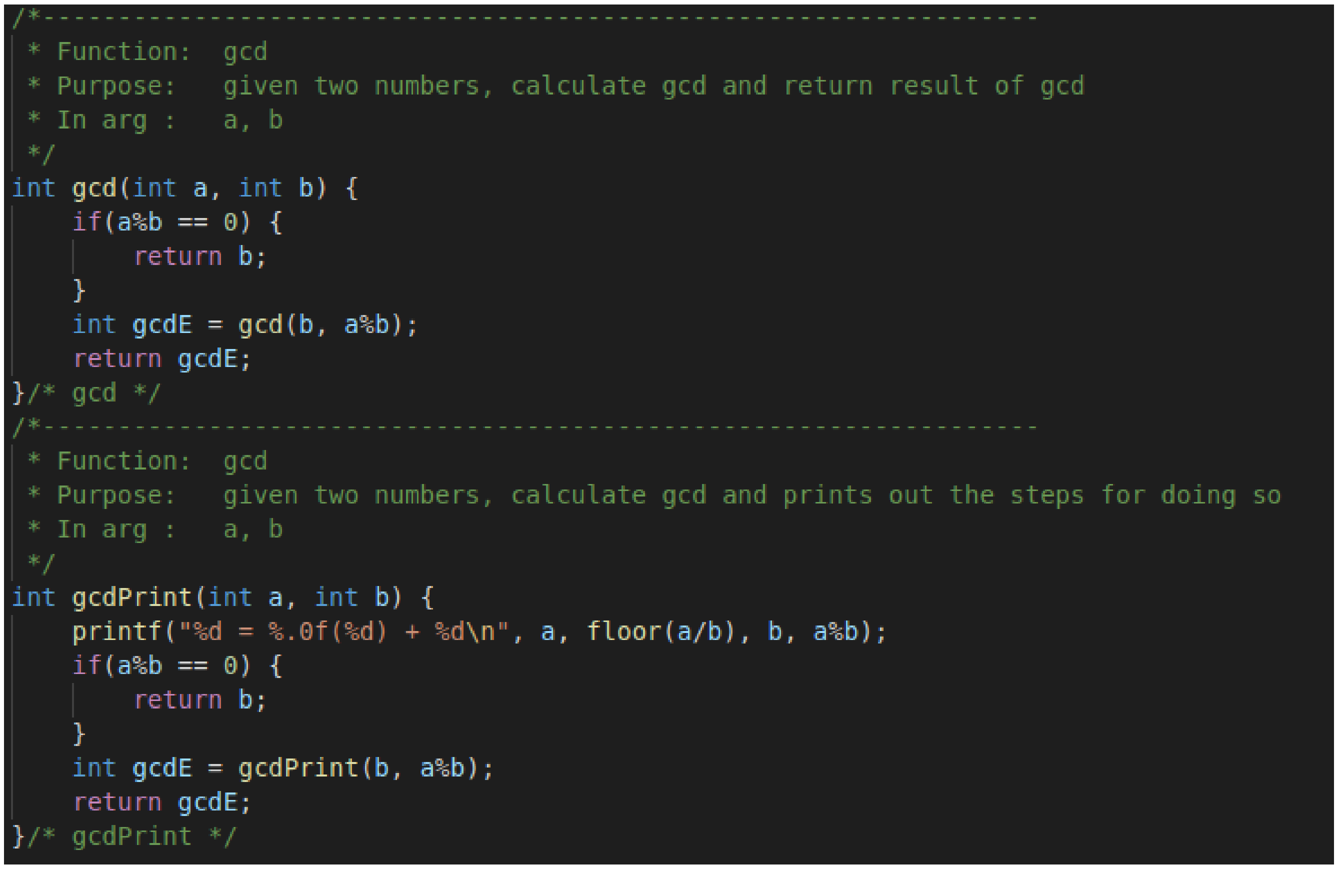
Once we have the inverses in the main function, We calculate the x and store this in an array of integers that has size of the gcd(a,n). Then we keep adding c to x with a loop until the array is completely full with unique values for x. This array is then printed out to the user showing all of the answers for x in mod n. (fig. 3)

Conclusion

In conclusion, the implementation of the Extended Euclidean Algorithm went smoothly and the amount of calculations that had to be done was less than expected. The language C did not prove to be a challenge which was unexpected given that C is not made for object-oriented programming. Some improvements I think can be made to this program is the amount of times the gcd functions are called can be less, the loops and calculations using recursion are capable of being parallelized thus improving in efficiency, and the implementation of reading in a file of many values for a, b and n, computing the values for x and printing them all out onto a file could be a feature to add. However, the way the program functions as is is satisfactory for my needs.

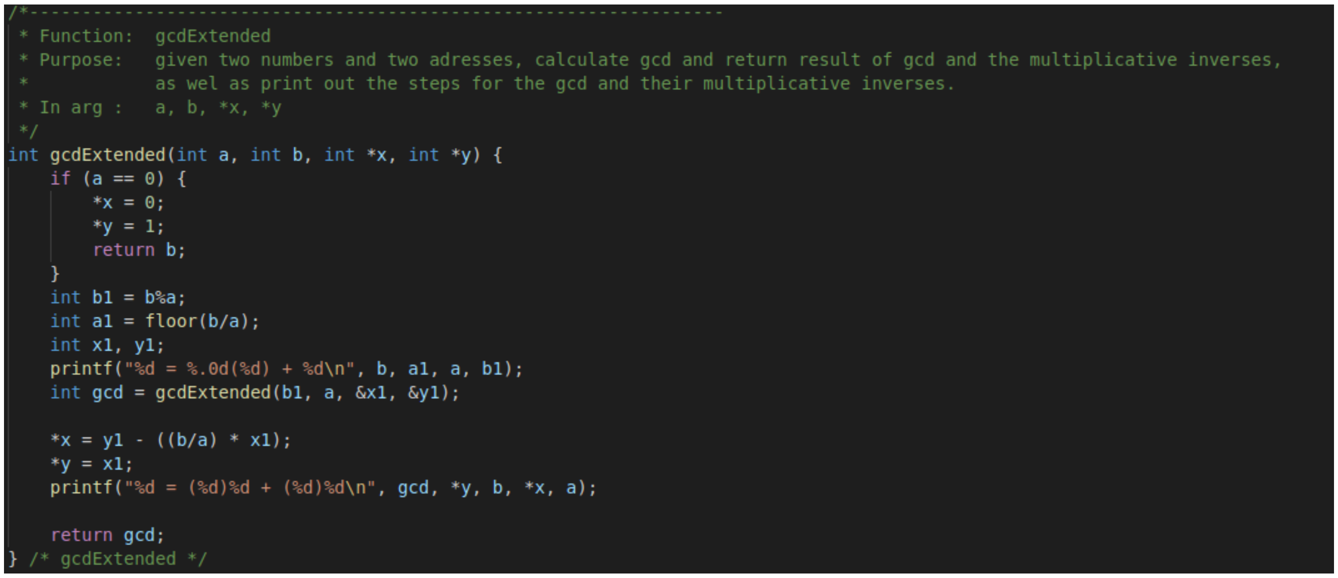
Figures

Fig. 1: gcd() methods with and without printing:



```c
/*-------------------------------------------------------------------
 * Function:  gcd
 * Purpose:   given two numbers, calculate gcd and return result of gcd
 * In arg :   a, b
 */
int gcd(int a, int b) {
    if(a%b == 0) {
        return b;
    }
    int gcdE = gcd(b, a%b);
    return gcdE;
}/* gcd */
/*-------------------------------------------------------------------
 * Function:  gcd
 * Purpose:   given two numbers, calculate gcd and prints out the steps for doing so
 * In arg :   a, b
 */
int gcdPrint(int a, int b) {
    printf("%d = %.0f(%d) + %d\n", a, floor(a/b), b, a%b);
    if(a%b == 0) {
        return b;
    }
    int gcdE = gcdPrint(b, a%b);
    return gcdE;
}/* gcdPrint */
```

Fig. 2: gcdExtended() function with printing:

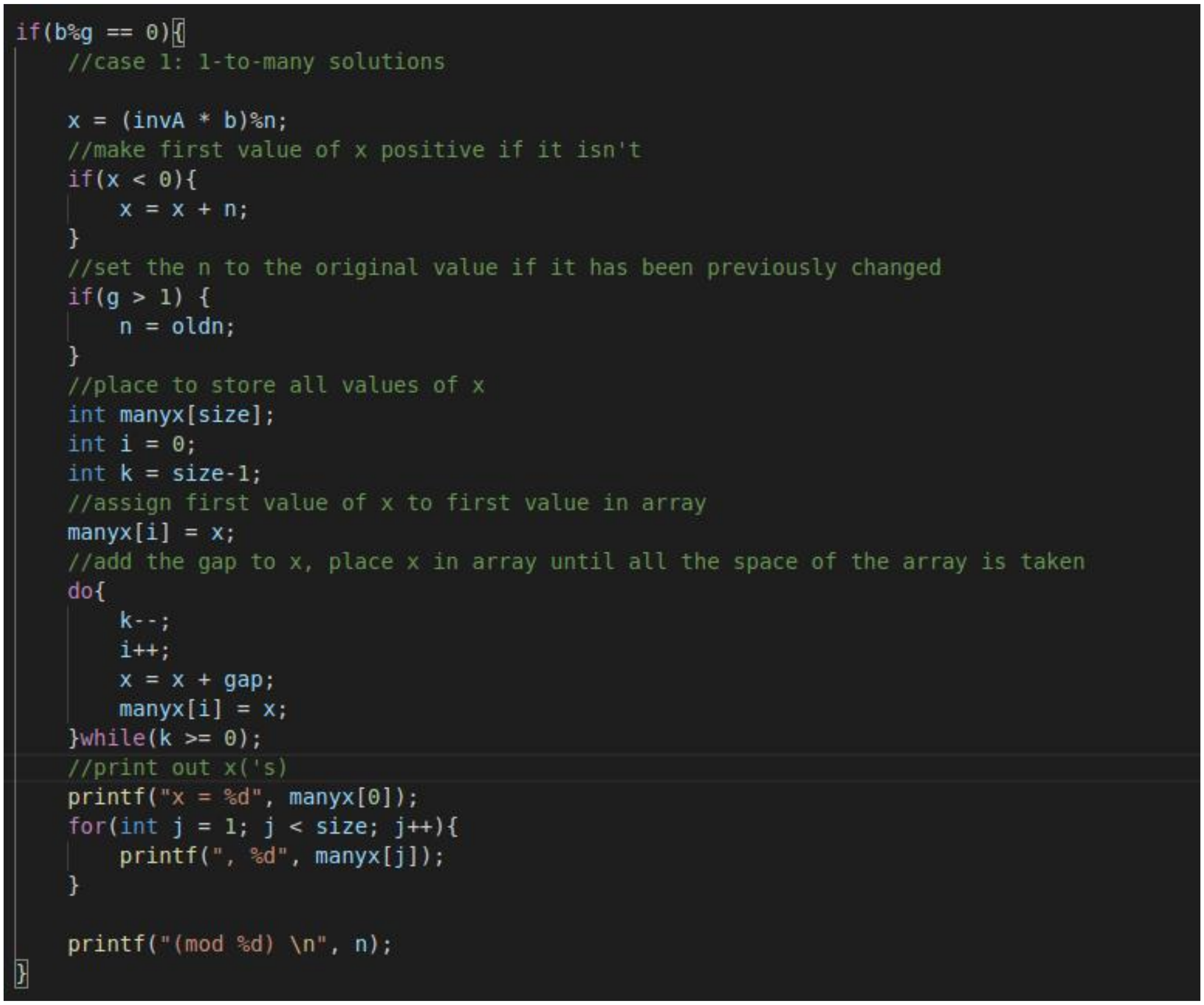


```c
/*-------------------------------------------------------------------
 * Function:  gcdExtended
 * Purpose:   given two numbers and two adresses, calculate gcd and return result of gcd and the multiplicative inverses,
 *            as wel as print out the steps for the gcd and their multiplicative inverses.
 * In arg :   a, b, *x, *y
 */
int gcdExtended(int a, int b, int *x, int *y) {
    if (a == 0) {
        *x = 0;
        *y = 1;
        return b;
    }
    int b1 = b%a;
    int a1 = floor(b/a);
    int x1, y1;
    printf("%d = %.0d(%d) + %d\n", b, a1, a, b1);
    int gcd = gcdExtended(b1, a, &x1, &y1);

    *x = y1 - ((b/a) * x1);
    *y = x1;
    printf("%d = (%d)%d + (%d)%d\n", gcd, *y, b, *x, a);

    return gcd;
} /* gcdExtended */
```

Fig. 3: Calculating, storing and printing x in main:


```c
if(b%g == 0){
    //case 1: 1-to-many solutions

    x = (invA * b)%n;
    //make first value of x positive if it isn't
    if(x < 0){
        x = x + n;
    }
    //set the n to the original value if it has been previously changed
    if(g > 1) {
        n = oldn;
    }
    //place to store all values of x
    int manyx[size];
    int i = 0;
    int k = size-1;
    //assign first value of x to first value in array
    manyx[i] = x;
    //add the gap to x, place x in array until all the space of the array is taken
    do{
        k--;
        i++;
        x = x + gap;
        manyx[i] = x;
    }while(k >= 0);
    //print out x('s)
    printf("x = %d", manyx[0]);
    for(int j = 1; j < size; j++){
        printf(", %d", manyx[j]);
    }

    printf("(mod %d) \n", n);
}
```