Teoria de numeros

October 30, 2019

1 Teoria de numeros

1.1 Primalidad

El primo "más famoso" en la comunidad de problemas: 1000000007

- Es primo muy grande, facil de escribir, y el resultado de las operaciones modulo él caben en un int (es el mayor inferior a $2^{31} - 1$).
- La **aritmética de modulo con primos** es mucho mejor "condicionada" : si das el buen resultado modulo 1000000007, es que realizaste los calculos requeridos; con no-primos, se puede explotar propriedades artiméticas para darle la vuelta al problema).
- Considerar por ejemplo el problema de calcular $k!$ modulo 1000000007 vs. modulo 2^{32}.

Un **test ingenuo para verificar si n es primo** sería checar la divisibilidad para todos los enteros en $[2, n-1]$. TLE casi seguro. Complejidad $O(n)$.

- Primera mejora: No probar todos los posibles divisores hasta $n-1$ sino hasta $\sqrt{n}$.
- Si hay un divisor de n más alla de $\sqrt{n}$ (otro que n) entonces necesariamente habrá un divisor de n inferior a $\sqrt{n}$ también (que habremos detectado antes). Complejidad $O(\sqrt{n})$.
- Otra mejora simple: verificar sólo los impares en $[3, n]$ y verificar 2 a parte (si un par divide a n entonces 2 también). $O(\frac{1}{2}\sqrt{n})$.

Finalmente, la mejora más crítica es verificar simplemente los **posibles divisores primos** en $[3, \sqrt{n}]$.

- Si un primo p no divide a n entonces ningún múltiple de p va a dividir a n.
- Complejidad reducida a O(#primos hasta n).

- Función contadora de números primos $\pi(M)$:

$$\pi(M) = O(M/(ln(M) - 1)).$$

- Entonces para el test de primalidad de n:

$$O(\frac{\sqrt{n}}{\log(n)})$$

- Requiere aplicar el test de primalidad a todo un intervalo (tipicamente, en batch antes de los queries).

2 Criba de Eratosthenes

```
In [ ]: // Version of the sieve by F. Halim
        #include <bitset> // compact STL for Sieve, more efficient than vector<bool>
        using namespace std;
        long long _sieve_size;
        bitset<10000010> bs; // 10^7 should be enough for most cases
        vector<int> primes; // compact list of primes in form of vector<int>

        void sieve(long long upperbound) { // create list of primes in [0..upperbound]
            _sieve_size = upperbound + 1; // add 1 to include upperbound
            bs.set(); // set all bits to 1
            bs[0] = bs[1] = 0; // except index 0 and 1
            for (long long i = 2; i <= _sieve_size; i++) if (bs[i]) {
                // cross out multiples of i starting from i * i
                for (long long j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
                primes.push_back((int)i); // also add this vector containing list of primes
            }
        } // call this method in main method
```

- Acopla una **criba** para guardar los resultados de primalidad hasta 10000000 con el vector primes para poder checar si los n hasta P^2 (P : el mayor primo contenido en primes) están divisibles por algún primo entre $[1, \sqrt{n}]$.
- El test de primalidad estará **pre-calculado** para todos los enteros hasta 10^7.
- Se aplica con primes y la función descrita más arriba para los números más grande, hasta 10^{14}.

```
In [ ]: bool isPrime(long long N) { // a good enough deterministic prime tester
            if (N <= _sieve_size) return bs[N]; // O(1) for small primes
            for (int i = 0; i < (int)primes.size(); i++)
                if (N % primes[i] == 0) return false;
            return true; // it takes longer time if N is a large prime!
        }

In [ ]: // inside int main()
        sieve(10000000); // can go up to 10^7 (need few seconds)
        printf("%d\n", isPrime(2147483647)); // 10-digits prime
        printf("%d\n", isPrime(136117223861LL)); // not a prime, 104729*1299709

In [ ]: #include <bitset>
        #include <iostream>
        using namespace std;
        bitset<10000010> bs; // 10^7 should be enough for most cases
        void sieve(long long upperbound) {
            long long ssize = upperbound + 1; // add 1 to include upperbound
            bs.set(); // set all bits to 1
            bs[0] = bs[1] = 0; // except index 0 and 1
            for (long long i = 2; i <= sqrt(ssize); i++) if (bs[i]) {
                // cross out multiples of i starting from i * i!
```

```
            for (long long j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
        }
    }

    int main() {
        // inside int main()
        sieve(10000000); // can go up to 10^7 (need few seconds)
        cout << bs[483647] << endl;
        return 0;
    }
```

2.0.1 Complejidad approximada de la construcción de la criba:

$$\frac{n}{2} + \frac{n}{3} ... + \frac{n}{n} = n \sum_{p \text{ prime} \leq n} \frac{1}{p}.$$

Se puede mostrar que la suma de los inversos de primos diverge y

$$\sum_{p \text{ prime} \leq n} = O(\log \log n).$$

Entonces, el costo de construcción de la criba es $O(n \log \log n)$.

UVA 10140

The branch of mathematics called number theory is about properties of numbers. One of the areas that has captured the interest of number theoreticians for thousands of years is the question of primality. A prime number is a number that is has no proper factors (it is only evenly divisible by 1 and itself). The first prime numbers are 2,3,5,7 but they quickly become less frequent. One of the interesting questions is how dense they are in various ranges. Adjacent primes are two numbers that are both primes, but there are no other prime numbers between the adjacent primes.

For example, 2,3 are the only adjacent primes that are also adjacent numbers.

Your program is given 2 numbers: L and U ($1 \leq L < U \leq 2147483647$), and you are to find the two adjacent primes C1 and C2 ($L \leq C1 < C2 \leq U$) that are closest (i.e. $C2 - C1$ is the minimum). If there are other pairs that are the same distance apart, use the first pair. You are also to find the two adjacent primes D1 and D2 ($L \leq D1 < D2 \leq U$) where D1 and D2 are as distant from each other as possible (again choosing the first pair if there is a tie).

Input: Each line of input will contain two positive integers, L and U, with L < U. The difference between L and U will not exceed 1,000,000.

Output: For each L and U, the output will either be the statement that there are no adjacent primes (because there are less than two primes between the two given numbers) or a line giving the two pairs of adjacent primes

Una criba hasta 10^9 no puede funcionar ahi.

- Habrá muchos tests de primalidad que hacer (todos los que están en el intervalo). Necesitamos una criba que contenga los resultados ya.

- Una criba **local** a cada intervalo de prueba puede ser la solución ("The difference between L and U will not exceed 1,000,000").
- Usar todos los divisores i posibles (sin o con test de primalidad, porque no tenemos la criba completa) hasta $\sqrt{U} \approx 30000$.
- Recorrer los únicos múltiples de esos divisores que caben en $[L, U]$, o sea coefficientes j tales que

$$\frac{L}{i} \leq j \leq \frac{U}{i}.$$

Complejidad de la construcción de la mini-criba:

$$\sum_{i=2}^{\sqrt{U}} \frac{U-L}{i} \approx (U-L)\log(\sqrt{U})$$

Del orden de 10^6. Recorrido también en 10^6 para la selección de los pares adyacentes.

```
In [ ]: int L,U;
        ios::sync_with_stdio(false);

        while (cin >> L >> U) {
            // Local sieve
            vector<bool> sieve(U-L+1,true);
            for (int i=2;i<sqrt(U)+1;i++) {
                for (int j=std::max(L/i-1,2);j<=U/i+1;j++)
                    if (i*j>=L && i*j<=U) {
                        sieve[i*j-L]=false;
                    }
            }
            // Evaluate the differences
            int last = -1;
            int diffmin= std::numeric_limits<int>::max();
            int diffmax=0;
            int k1,k2,k1m,k2m;
            for (int k=0;k<U-L+1;k++) {
                if (sieve[k] && k+L>1) {
                    if (last>=0) {
                        if (k-last<diffmin) {
                            diffmin = k-last;
                            k1 = last+L;
                            k2 = k+L;
                        }
                        if (k-last>diffmax) {
                            diffmax = k-last;
                            k1m= last+L;
                            k2m= k+L;
                        }
```

```
            }
            last = k;
        }
    }
    if (diffmin==std::numeric_limits<int>::max())
        cout << "There are no adjacent primes." << endl;
    else
        cout << k1 <<"," << k2 << " are closest, " << k1m << "," << k2m << "
}
```

UVA 10738

One of the biggest, most mathematicians would call it THE biggest, unsolved problems in mathematics is the proof of the Riemann Hypothesis: "All non-trivial zeros of the zeta function have real part onehalf". Now your task is simple: For any natural number N, give the N-th zero. . . nah, just kidding! That would be a much too complex problem for an online contest.

We'll leave Riemann and the zeta function and concern ourselves with the closely related, but much easier to calcultate Mertens's function. For those interested in the subject I can heartly recommend Derbyshire's book (see the epilogue). Every positive natural number greater than 1, can be uniquely decomposed into it's prime factors. Some numbers have only one factor, namely the number itself, like 2, 11 and 71, and are called prime numbers. Others have more than one factor, like 4 (2×2), 15 (3×5) and 144 (2×2×2×2×3×3), and are called composite numbers. If a number contains all it's prime factors only once, it is called square free. All prime numbers are square free. Some composite numbers are square free, like 21 (3×7) and 187 (11×17), others are not, like 9 (3×3) and 98 (2×7×7).

Let's define the Mobius function mu(N), for all positive natural numbers N: * mu(1) = 1, by definition; * if N is not square free, mu(N) = 0; * if N is square free and contains an even number of prime factors, mu(N) = 1; * if N is square free and contains an odd number of prime factors, mu(N) = −1.

Now we can define Mertens's function M(N) as the sum of all mu() for 1 up to and including N: M(N) = mu(1) + mu(2) + . . . + mu(N).

Input: Up to 1000 numbers between 1 and 1000000 (one million), both included, each on a line by itself. The numbers are in random order and can appear more than once. Input is terminated by a line, which contains a single zero. This line should not be processed.

Output: For each number in the input print that number, the value of mu() for that number and the value of M () for that number, all three on one line, right justified in fields of width 8. The input order must be preserved.

In [ ]:

En este problema, podemos manejar la primalidad (criba) facilmente.

- Necesitamos evaluar dos otras cantidades: un indicador de square-free y (para los square-free) el número de factores primos.
- Idea: usar la misma técnica de la criba (recorriendo los múltiples de factores primos) para:

1. marcar los no-square-free: son todos los múltiples de un divisor potencial i tales que el factor j es a su vez múltiple de i ($j\%i = 0$).
2. el número de factores primos se puede propagar por el mismo proceso de tipo criba: $nf(i * j) = 1 + nf(j)$. Observar que si hacemos eso para los únicos square-free, necesariamente habremos calculado $nf(j)$.

```
In [ ]: ios::sync_with_stdio(false);
        // Primes
        memset(&sieve[0],1,sizeof(sieve));
        memset(&nf[0],0,sizeof(nf));
        memset(&mus[0],1,sizeof(mus));
        mus[1] = 1;

        for (int i=2;i<=1000;i++) // We go up to sqrt(SMAX)
            if (sieve[i]) {
                nf[i]  = 1;  // Number of factors: 1 if prime
                mus[i] = -1; // Square-free indicator: -1 if prime
                for (int j=i;i*j<SMAX;j++) {
                    // The traditional sieve: marks multiples as non-primes
                    sieve[i*j]=false;
                    if (mus[i*j])
                        // A priori square free (1) but if we find that i divides j, then it is
                        mus[i*j] = (j%i!=0);
                }
            }
        // To cover all the primes (up to SMAX)
        for (int i=2;i<=SMAX;i++)
            if (sieve[i]) {
                nf[i]= 1;
                mus[i] = -1;
            }
        // Update the number of factors for square-free numbers
        for (int i=2;i<=1000;i++) {
            if (sieve[i]) {
                for (int j=2;i*j<SMAX;j++) {
                    if (mus[i*j] && nf[j]) {
                        // one is i, the other should have been determined in j
                        // hence we have updated nf[j] before
                        nf[i*j] = 1 + nf[j];
                        // check parity
                        mus[i*j] = (nf[i*j]%2==0)?1:-1;
                    }
                }
            }
        }
        // Deduce the values of the Mertens function
        ns[1]=1;
        for (int i=2;i<=SMAX;i++) {
```

```
        ns[i]=mus[i]+ns[i-1];
    }
    int a;
    while (cin>>a && a!=0) {
        cout << std::setw(8) << a << std::setw(8) << mus[a] << std::setw(8) <<
    }
```

2.1 GCD, LCM

El **mayor común divisor** de dos enteros a y b es el mayor entero positivo g tal que g divide a y g divide b.

Aplicación clásica: simplificar fracciones.

$$\frac{a}{b} = \frac{\frac{a}{gcd(a,b)}}{\frac{b}{gcd(a,b)}}$$

De la misma manera, el **menor común múltiple** de dos enteros a y b es el menor entero positivo m que sea a la vez múltiple de a y múltiple de b.

Relación entre el $gcd(a,b)$ y $lcm(a,b)$?

$$lcm(a,b)gcd(a,b) = ab$$

Para cacular el gcd: **Euclides** (para $a > b$)

```
In [ ]: int gcd(int a, int b) { return (b == 0 ? a : gcd(b, a % b)); };
        int lcm(int a, int b) { return (a * (b / gcd(a, b))); };
```

```
In [ ]: // Sin recursión:
        inline int gcd(int a,int b) {
            while (b!=0) {
                int t = b;
                b = a%b;
                a = t;
            }
            return a;
        }
```

UVA 10407

Integer division between a dividend n and a divisor d yields a quotient q and a remainder r. q is the integer which maximizes q $\times$ d such that q $\times$ d $\leq$ n and r = n $-$ q $\times$ d. For any set of integers there is an integer d such that each of the given integers when divided by d leaves the same remainder.

Input: Each line of input contains a sequence of nonzero integer numbers separated by a space. The last number on each line is 0 and this number does not belong to the sequence. There will be at least 2 and no more than 1000 numbers in a sequence; not all numbers occuring in a sequence are equal. The last line of input contains a single 0 and this line should not be processed.

Output: For each line of input, output the largest integer which when divided into each of the input integers leaves the same remainder.

Sean s_i , $i \in [1, N]$ los elementos del conjunto. Entonces la propriedad del d buscado es que existe r tal que:

$$\forall i \in [1, N] s_i = q_i d + r,$$

entonces al restar s_1 a todos, tenemos:

$$\forall i \in [2, N] s_i - s_1 = (q_i - q_1) d.$$

d entonces es necesariamente el gcd de las $N-1$ diferencias $s_i - s_1$. Sólo falta usar $gcd(a, b, c) = gcd(gcd(a, b), c)$.

2.2 Descomposición en factores primos.

- Algoritmo **recursivo**: para encontrar la decomposición de n en factores primos, simplemente dividir hasta que se pueda n por divisores primos.
- Implica construir una criba.
- Probar con los primos hasta $\sqrt{n}$; si el cociente final no es 1, es que falta considerar un primo $\geq \sqrt{n}$.

```
In [ ]: vector<int> primeFactors(long long n) {
            vector<int> factors;
            long long pid = 0, pf = primes[pid];

            while (n != 1 && (pf * pf <= n)) {
                while (n % pf == 0) { n /= pf; factors.push_back(pf); }
                pf = primes[++pid]; // only consider primes!
            }
            if (n != 1) factors.push_back(n); // n is a prime
            return factors;
        }
```

Complejidad en el peor caso (n primo): $O(\sqrt{n}/\log n)$.

Funcionará mientras n no tiene factores primos que quedan $\geq p_P^2$ donde p_P es el mayor primo encontrado con la criba.

UVA 516

Everybody in the Prime Land is using a prime base number system. In this system, each positive integer x is represented as follows: Let $p_i{}_{i=0}^{\infty}$ denote the increasing sequence of all prime numbers. We know that $x > 1$ can be represented in only one way in the form of product of powers of prime factors. This implies that there is an integer k_x and uniquely determined integers $e_{k_x}, e_{k_x-1}, ..., e_1, e_0$, $(e_{k>0})$, such that $x = p_{k_x}^{e_{k_x}} p_{k_x-1}^{e_{k_x}-1} ... p_1^{e_1} p_0^{e_0}$. The sequence

$$(e_{k_x}, e_{k_x}, ..., e_1, e_0)$$

is considered to be the representation of x in prime base number system. It is really true that all numerical calculations in prime base number system can seem to us a little bit unusual, or even

hard. In fact, the children in Prime Land learn to add to subtract numbers several years. On the other hand, multiplication and division is very simple. Recently, somebody has returned from a holiday in the Computer Land where small smart things called computers have been used. It has turned out that they could be used to make addi- tion and subtraction in prime base number system much easier. It has been decided to make an experiment and let a computer to do the operation "minus one". Help people in the Prime Land and write a corresponding program. 9For practical reasons we will write here the prime base representation as a sequence of such pi and ei from the prime base representation above for which ei > 0. We will keep decreasing order with regard to pi.

Input: The input file consists of lines (at least one) each of which except the last contains prime base representation of just one positive integer greater than 2 and less or equal 32767. All numbers in the line are separated by one space. The last line contains number 0.

Output: The output file contains one line for each but the last line of the input file. If x is a positive integer contained in a line of the input file, the line in the output file will contain $x - 1$ in prime base representation. All numbers in the line are separated by one space. There is no line in the output file corresponding to the last "null" line of the input file.

- Construir el número x a partir de su representación
- Restar 1.
- Descomponer en factores primos.

2.3 Funciones clásicas con números primos

```
In [ ]: long long numFactors(long long n) {
            long long pid = 0, pf = primes[pid], num = 0;
            while (n != 1 && (pf * pf <= n)) {
                while (n % pf == 0) { n /= pf; num++; }
                pf = primes[++pid];
            }
            if (n != 1) num++;
            return num;
        }
```

- Contar el número de factores primos distintos de n?
- Suma de los factores primos de n?
- Número de divisores de n?
- Suma de los divisores de n?

Número de divisores de n:

- Contar para cada divisor primo p_i su multiplicidad (k_i).
- La respuesta es $\prod_i (k_i + 1)$.

Suma de los divisores de n:

- Verla como la expansión de:

$$\prod_i (p_i^0 + p_i^1 + ... + p_i^{k_i}).$$

- Se puede simplificar en:

$$\prod_i \frac{p_i^{k_i+1}-1}{p_i-1}.$$

Otra pregunta clásica es: para un entero $n > 0$, determinar el **número de de enteros positivos inferiores a n que sean primos relativos con n** (es decir, enteros $m < n$ tales que $gcd(n, m) = 1$).

- Una opción simple pero eventualmente pesada para n grande es recorrer todos los enteros $m < n$ y calcular el gcd.
- Euler ha mostrado que la función $\varphi(n)$:

$$\varphi(n) = n\prod_i(1-\frac{1}{p_i})$$

  calcula exactamente este número.

Por ejemplo: para $n = 24$, nos da $\varphi(24) = 24(1-\frac{1}{2})(1-\frac{1}{3}) = 8$. Los primos relativos son: $1, 5, 7, 11, 13, 17, 19, 23$. Complejidad en $O(n/\log n)$.

```
In [ ]: long long phi(long long n) {
            long long pid = 0, pf = primes[pid], num = n;
            while (n != 1 && (pf * pf <= n)) {
                if (n % pf == 0) num -= num / pf;
                pf = primes[++pid];
            }
            if (n != 1) num -= num/n;
            return num;
        }
```

UVA 11226

A prime number p is a natural number greater than 1 that has only two natural divisors: 1 and p itself. Any natural number n, such that $n > 1$, has a unique decomposition into prime factors, for instance $4 = 2 \times 2, 5 = 5, 6 = 2 \times 3$.

Let $sopf(n)$ denote the sum of the prime factors of a natural number n. For instance, $sopf(4) = 2+2 = 4$, $sopf(5) = 5$, and $sopf(6) = 2+3 = 5$. If we take the result of this sum, we may compute again the sum of its prime factors, and repeat this ad nauseam. However, at some point, we always reach a fix-point, that is a number f such that $f = sopf(f)$. For instance, starting from 8, $sopf(8) = 2+2+2 = 6$, then $sopf(6) = 2+3 = 5$, and $sopf(5) = 5$: applying repetitively sopf from 8 generates the sequence $8, 6, 5, 5, 5, ...$. So, from the initial value 8, it takes 3 applications of sopf to discover that we have reached a fix-point. Let $lsopf(n)$ denote the number of applications of $sopf$ from n that is required to discover that the fix-point has been reached. For instance, $lsopf(8) = 3$ and $lsopf(4) = 1$. Your task is, given two natural numbers n and m (with $n > 1$ and $m > 1$), find the largest value the function lsopf takes in the interval between n and m.

Input: The first line of input gives the number of cases, T (with $1 \leq T \leq 150$). T test cases follow. Each test case is on a single line, containing two natural numbers n and m, such that $1 < n, m \leq 500000$.

Output: For each test case first print a line 'Case #C:' (where C is the number of the current test case). Then print another line with the maximum of the function lsopf in the interval bounded by n and m.

El calculo de las sumas de los factores primos se puede integrar en la generación de la criba.

$$s(i * j) = i + s(j)$$

- Las longitudes corresponden a secuencias decrecientes. $sopf(x) \leq x$. Entonces se pueden calcular sucesivamente en $O(n_{max})$.
- Finalmente, buscar el máximo entre intervalos $[n, m]$ se puede hacer por programación dinámica (pre-calculando una tabla de $l(i, j) = \max[i, i + 2^j - 1]$). El precalculo es en $O(n_{max} \log n_{max})$ y las queries en $O(1)$ (ver Range Minimum Query).

2.4 Aritmética de modulo

Aparece en muchos problemas el hecho de dar soluciones modulo algún numero (o los últimos dígitos...).

Recordar:

- (x $\times$ y) mod m = ((x mod m) $\times$ (y mod m)) mod m
- (x + y) mod m = ((x mod m) + (y mod m)) mod m
- (x $-$ y) mod m = ((x mod m) $-$ (y mod m)) mod m

UVA 10176

Ocean deep

I'm so afraid to show my feelings,

I have sailed a million ceilings

In my solitary room

Ocean deep

The lines above are from a very popular song of Cliff Richard. In this problem we will be dealing with a similar type of person. His name is Rampell-Stilt-Skin and another important thing is that he is a dead man. Someone has killed him a few days ago and you are the detective to solve the mystery. The problem with this guy is that he always tried to hide his information and feelings under the sea (I mean out of reach). He wrote a diary, which contained some statements and then a large binary number (May have as many as 10000 digits). If the number is divisible by a large prime number 131071 then the statement is true, otherwise the statement is false. In this problem you will be given large binary numbers as input and you will have to verify if that number is divisible by 131071 or not. Your algorithm needs to be very efficient.

Input: The input file contains several binary numbers. Each binary number starts in a new line but may expand in several lines. Each number is terminated by a '#' symbol. No line contains more than 100 digits.

Output: For every binary number print 'YES' if the number is divisible by the given prime number, print 'NO' if the binary number is not divisible by the given prime number.

Convertir cada bit de la representación binaria a una potencia de 2 modulo 131071.
Calcular las potencias de 2 (modulo 131071) recursivamente:

$$p = (p * 2)\%131071$$

...o por la estrategia de exponenciación binaria

```
In [ ]: inline unsigned int modularPow(unsigned int b,unsigned int e,unsigned int md)
            unsigned long long rm = 1;
            unsigned long long bb = b%md;
            while (e) {
                if (e%2)
                    rm = (rm * bb)%md;
                e >>= 1;
                bb = (bb*bb)%md;
            }
            return rm;
        }
```

2.5 Algoritmo de Euclides extendido

Sean a, b, c tres enteros. Consideremos la ecuación lineal (identidad de Bezout o ecuación diofántica):

$$ax + by = c.$$

Habrá soluciones enteras? Una condición **necesaria y suficiente** para que esa ecuación tenga soluciones enteras es:

$$c\%d = 0 \text{ donde } d = gcd(a, b)$$

Si $c\%d \neq 0$ **no hay solución**.
En el caso contrario habrá una **infinidad** de soluciones, dadas por

$$\begin{aligned} x &= x_0 + (b/d)n \\ y &= y_0 - (a/d)n \end{aligned}$$

donde x_0, y_0 es una solución y n es un entero arbitrario. En general el enunciado del problema permitirá elegir n.
El algoritmo de **Euclides extendido** da una solución a:

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

Y si $c = kgcd(a, b)$ entonces una solución al problema

$$ax + by = c$$

es $x_0 = kx'$, $y_0 = ky'$.

```
In [ ]: x = 1;
        y = 0;
        void extendedEuclid(int a, int b) {
            if (b == 0) { x = 1; y = 0; d = a; return; }
            extendedEuclid(b, a % b);
            int x1 = y;
            int y1 = x - (a / b) * y;
            x=x1;
            y=y1;
        }
```

UVA 10090

I have some (say, n) marbles (small glass balls) and I am going to buy some boxes to store them. The boxes are of two types:

- Type 1: each box costs c1 Taka and can hold exactly n1 marbles
- Type 2: each box costs c2 Taka and can hold exactly n2 marbles

I want each of the used boxes to be filled to its capacity and also to minimize the total cost of buying them. Since I find it difficult for me to figure out how to distribute my marbles among the boxes, I seek your help. I want your program to be efficient also.

Input: The input file may contain multiple test cases. Each test case begins with a line containing the integer n ($1 \leq$ n $\leq 2,000,000,000$). The second line contains c1 and n1, and the third line contains c2 and n2. Here, c1, c2, n1 and n2 are all positive integers having values smaller than 2,000,000,000. A test case containing a zero for n in the first line terminates the input.

Output: For each test case in the input print a line containing the minimum cost solution (two nonnegative integers m1 and m2, where mi = number of Typei boxes required) if one exists, print 'failed' otherwise. If a solution exists, you may assume that it is unique.

Buscamos $x, y \geq 0$ tal que

$$n_1 x + n_2 y = n$$

Empezamos con la resolución de la identidad de Bezout:

$$n_1 x + n_2 y = gcd(n_1, n_2)$$

con dos soluciones x_0, y_0. Sea $k = \frac{n}{gcd(n_1, n_2)}$, entonces la **familia de las soluciones del problema inicial** es de la forma:

$$k(x_0 + n_2/gcd(n_1, n_2)m), k(y_0 - n_1/gcd(n_1, n_2)m)$$

El m se busca en un intervalo tal que $x_0 + n_2/gcd(n1, n2)m, y0 - n1/gcd(n1, n2)m \geq 0$.

- Esa desigualdad define un intervalo para m.
- Como queremos minimizar el costo, que es **lineal** en m:

$$xc_1 + yc_2$$

- Sólo basta determinar el signo de la pendiente de esta función, es decir $(n_2c_1 - n_1c_2)$. La solución está en uno de los dos límites del intervalo de búsqueda.

```
In [ ]: ios::sync_with_stdio(false);
        long n,c1,c2,n1,n2;
        while (cin >> n) {
            if (n==0) break;
            cin >> c1 >> n1 >> c2 >> n2;
            long g,x,y;
            if (n1>n2)
                extendedEuclid(n1,n2,g,x,y);
            else
                extendedEuclid(n2,n1,g,y,x);
            if (n%g!=0)
                cout << "failed" << endl;
            else {
                x *= (n/g);
                y *= (n/g);
                n1 /= g;
                n2 /= g;
                // Boundaries
                long b1 = ceil(-(double)x/n2);
                long b2 = floor((double)y/n1);
                if (b1>b2) {
                    cout << "failed" << endl;
                    continue;
                }
                long c = c1*n2-c2*n1;
                if (c>0) {
                    x=x+b1*(n2);
                    y=y-b1*(n1);
                } else {
                    x=x+b2*(n2);
                    y=y-b2*(n1);
                }
                cout << x << " " << y << endl;
            }
        }
```