Generate Divisors of N

We want to find the all divisors of N. We begin by writing the number as a product of prime factors:

N=p1q1 x p2q2 x p3q3 x . . . . . . . . . . . . x pkqk

We store the prime number and their frequency. Then, recursively generate all possible combination and store the divisors.

**#include <stdio.h>**

**#include <math.h>**

**#include <algorithm>**

**using namespace std;**

**#define SIZE\_N 1000**

**#define SIZE\_P 1000**

**bool flag[SIZE\_N+5];**

**int primes[SIZE\_P+5];**

**int seive()**

**{**

**int i,j,total=0,val;**

**for(i=2;i<=SIZE\_N;i++) flag[i]=1;**

**val=sqrt(SIZE\_N)+1;**

**for(i=2;i<val;i++)**

**if(flag[i])**

**for(j=i;j\*i<=SIZE\_N;j++)**

**flag[i\*j]=0;**

**for(i=2;i<=SIZE\_N;i++)**

**if(flag[i])**

**primes[total++]=i;**

**return total;**

**}**

**int store\_primes[100],freq\_primes[100], store\_divisor[10000], Total\_Prime, ans;**

**//Very Much Faster Divisor Function…………………………………………**

**void divisor(int N)**

**{**

**int i,val,count;**

**val=sqrt(N)+1;**

**Total\_Prime=0;**

**for(i=0;primes[i]<val;i++)**

**{**

**if(N%primes[i]==0)**

**{**

**count=0;**

**while(N%primes[i]==0)**

**{**

**N/=primes[i];**

**count++;**

**}**

**store\_primes[Total\_Prime]=primes[i];**

**freq\_primes[Total\_Prime]=count;**

**Total\_Prime++;**

**val=sqrt(N)+1; // sqrt again for fast compute.**

**}**

**}**

**if(N>1)**

**{**

**store\_primes[Total\_Prime]=N;**

**freq\_primes[Total\_Prime]=1;**

**Total\_Prime++;**

**}**

**}**

**void Generate(int cur,int num)**

**{**

**int i,val;**

**if(cur==Total\_Prime)**

**{**

**store\_divisor[ans++]=num;**

**}**

**else**

**{**

**val=1;**

**for(i=0;i<=freq\_primes[cur];i++)**

**{**

**Generate(cur+1,num\*val);**

**val=val\*store\_primes[cur];**

**}**

**}**

**}**

**int main()**

**{**

**int total=seive();**

**int n,i;**

**while(scanf("%d",&n)==1)**

**{**

**divisor(n);**

**ans=0;**

**Generate(0,1);**

**sort(&store\_divisor[0],&store\_divisor[ans]);**

**printf("Total No of Divisors: %d\n",ans);**

**for(i=0;i<ans;i++)**

**printf("%d ",store\_divisor[i]);**

**printf("\n");**

**}**

**return 0;**

**}**

**/\***

**Input:**

**20**

**30**

**15**

**Output:**

**Total No of Divisors: 6**

**1 2 4 5 10 20**

**Total No of Divisors: 8**

**1 2 3 5 6 10 15 30**

**Total No of Divisors: 4**

**1 3 5 15**

**\*/**

**Generate Number of Divisors [1 to N]**

**We know :** N=p1q1 x p2q2 x p3q3 x . . . . . . . . . . . . x pkqk =p1q1 x M [where M= p2q2 x p3q3 x . . . . . . . . . . . . x pkqk]

Number of Divisor N is:  
 D(N)=(q1+1) x (q2+1) x (q3+1) x . . . . . . . . . . . . x (qk+1)

Number of Divisor M is:  
 D(M)= (q2+1) x (q3+1) x . . . . . . . . . . . . x (qk+1)

Solve the simple math we find the relation:  
 D(N)=(q1+1) x D(M)

**Sample C++ Code:**

**int D[1000010];**

**void DivisorGenerate()**

**{**

**int i,j,val,N,M,count;**

**D[1]=1;**

**for(i=2;i<=1000000;i++)**

**{**

**N=M=i;**

**val=sqrt(N)+1;**

**for(j=0;primes[j]<val;j++)**

**{**

**if(M%primes[j]==0)**

**{**

**count=0;**

**while(M%primes[j]==0)**

**{**

**M/=primes[j];**

**count++;**

**}**

**D[N]=(count+1)\*D[M];**

**break;**

**}**

**}**

**if(M==N) //Special Case if N equal prime**

**{**

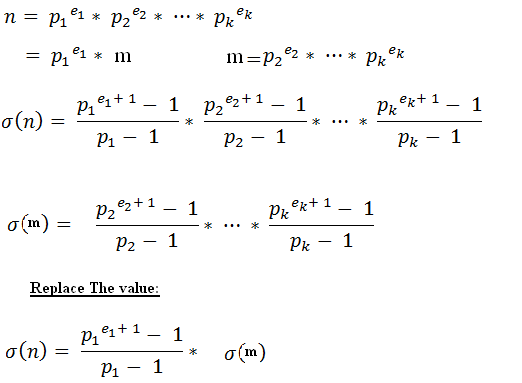
**D[N]=2;**

**}**

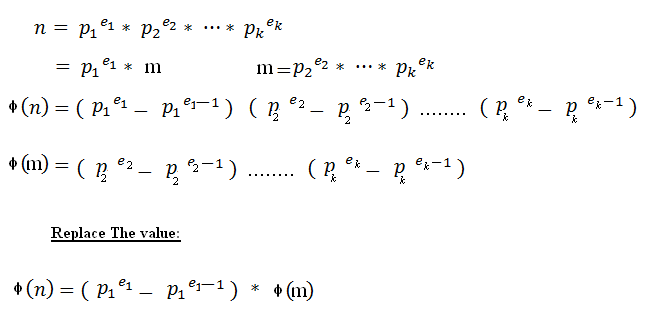
**}**

**}**

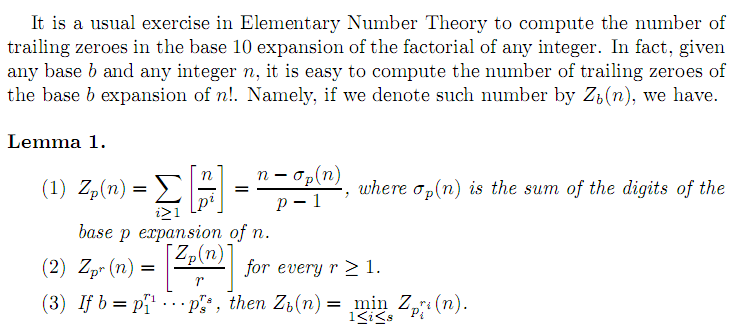
**Generate Sum of Divisors [1 to N]**



Generate Relative Prime [1 to N]



Number of Trailing Zeroes of N Factorial Base B



**C++ Code:**

**#include <stdio.h>**

**#include <math.h>**

**#include <algorithm>**

**using namespace std;**

**#define SIZE\_N 1000**

**#define SIZE\_P 1000**

**bool flag[SIZE\_N+5];**

**int primes[SIZE\_P+5];**

**int seive()**

**{**

**int i,j,total=0,val;**

**for(i=2;i<=SIZE\_N;i++) flag[i]=1;**

**val=sqrt(SIZE\_N)+1;**

**for(i=2;i<val;i++)**

**if(flag[i])**

**for(j=i;j\*i<=SIZE\_N;j++)**

**flag[i\*j]=0;**

**for(i=2;i<=SIZE\_N;i++)**

**if(flag[i])**

**primes[total++]=i;**

**return total;**

**}**

**int factors\_in\_factorial(int N,int p)**

**{**

**int sum=0;**

**while(N)**

**{**

**sum+=N/p;**

**N/=p;**

**}**

**return sum;**

**}**

**int Trailingzero\_Base\_B(int N,int B)**

**{**

**int i,ans,freq,power;**

**ans=1000000000;**

**for(i=0;primes[i]<=B;i++)**

**{**

**if(B%primes[i]==0)**

**{**

**freq=0;**

**while(B%primes[i]==0)**

**{**

**freq++;**

**B/=primes[i];**

**}**

**power=factors\_in\_factorial(N,primes[i]);**

**ans=min(ans,power/freq);**

**}**

**}**

**return ans;**

**}**

**int main()**

**{**

**int total=seive();**

**int i,N,B,zero;**

**while(scanf("%d %d",&N,&B)==2)**

**{**

**zero=Trailingzero\_Base\_B(N,B);**

**printf("%d\n",zero);**

**}**

**return 0;**

**}**

Last Non Zero Digit of Factorial

**Approach 1: Slow**

We want to last non zero digit of Factorial N. Example: 5! = 120, here last non zero digit 2. How calculate? We know any factorial can be represented by prime factor.

N! = 2q2 x 3q3 x 5q5 x 7q7 x 11q11 x 13q13……….

But we know that, pair(5,2) create a trailing zero and power of 5(q5) is less than power of 2(q2). So we discard it for this problem

N!” = 2q2-q5 x 3q3 x 7q7 x 11q11 x 13q13……….

So that last non Zero digit:

L(N)=(N!”)%10

=(( 2q2-q5 %10) x (3q3 %10) x (7q7 %10) x (11q11 %10) x (13q13%10) …….)%10

**Approach 2: Very Faster**

L(0)=1  
L(1)=1  
L(2)=2  
L(3)=6  
L(4)=4  
L(N)= (2N/5 x L(N/5) x L(N%5) )%10

= ((2N/5 %10) x L(N/5) x L(N%5) )%10

**int PTwo(int N)**

**{**

**int T[]={6,2,4,8};**

**if(N==0) return 1;**

**return T[N%4];**

**}**

**int LastNZDigit(int N)**

**{**

**int A[]={1,1,2,6,4};**

**if(N<5) return A[N];**

**return (PTwo(N/5)\*LastNZDigit(N/5)\*LastNZDigit(N%5))%10;**

**}**

Greatest Common Divisor

* If d|a and d|b, then d|(ax+by) for any integers x and y GCD(a, b) = GCD(b, a)
* GCD(a, b) = GCD(-a, b)
* GCD(a, b) = GCD(|a|, |b|)
* GCD(a,0) = |a|
* GCD(a, ka) = |a|
* If *a* and *b* are any integers, not both zero, then GCD(a,b) is the smallest positive element of the set {ax + by : x, y in **Z**} of linear combinations of *a* and *b*.
* GCD(a, b) = GCD(b, a mod b)
* LCM(a,b)=(a\*b)/GCD(a,b)

If A= p1x1 x p2x2 x p3x3 x . . . . . . . . . . . . x pkxk

B= p1y1 x p2y2 x p3y3 x . . . . . . . . . . . . x pkyk

GCD(A,B)= p1min(x1,y1) x p2min(x2,y2) x . . . . . . . . . . . . x pkmin(xk,yk)

LCM(A,B) = p1max(x1,y1) x p2max(x2,y2) x . . . . . . . . . . . . x pkmax(xk,yk)

|  |
| --- |
| **Extended Euclidean Algorithm:-**  Given **a** and **b**. Extended Euclidean Algorithm to find **x**, **y** and **gcd(a,b)**. Form This equation  **ax+by=gcd(a,b)**  **int Extended\_Euclidean(int a,int b,int &x,int &y)**  **{**  **if(b==0)**  **{**  **x=1;y=0;**  **return a;**  **}**  **int d=Extended\_Euclidean(b,a%b,y,x);**  **y=y-(a/b)\*x;**  **return d;**  **}** |

Congruence

**Definition: Two integers a and b are said to be congruence (or equivalent) module an integer m — written a ≡ b mod m — if m ∣ (a−b).**

**Example:** In class we said that 19≡2 mod 17, that 51 ≡ 0 mod 17, and that 10≡−10mod20.

1. **Reflexive:** for any integer *a* and any modulus *m*, we have *a* ≡ *a* mod *m*.
2. **Symmetric:** for any integers *a* and *b* and any modulus *m*, if *a* ≡ *b* mod *m*  then  *b* ≡ *a* mod *m*.
3. **Transitive:** for any integers *a*,*b* and *c*, and any modulus *m*, if  *a* ≡ *b* mod *m* and  *b* ≡ *c* mod *m*, then *a* ≡ *c* mod *m*.

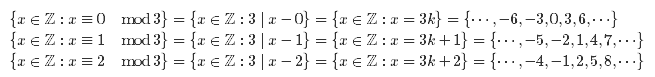
**Proof:**

To prove the reflexive property, note that *a* ≡ *a* mod *m* just means that we want to verify *m*∣(*a*−*a)*=0. We saw a while back, though, that any integer *m* divides 0, so this statement is valid.

To prove symmetry, we need to show that *a* ≡ *b* mod *m* implies *b* ≡ *a* mod *m*. If *a* ≡ *b* mod *m*, though, the definition of modular congruence tells us that *m*∣(*a*−*b)*, so that *mk*=(*a*−*b)*. But then we have *m*(−*k*)=−(*a*−*b*)=*b*−*a*, and so *m*∣(*b*−*a)*. By the definition of modular congruence, we therefore have *b* ≡ *a* mod *m*.

Finally, for transitivity we are supposed to assume that *a* ≡ *b* mod *m* and *b* ≡ *c* mod *m*, and somehow conclude that *a* ≡ *c* mod *m*. To prove this result, we note that the first two congruence conditions tells us that *m*∣(*a*−*b)* and *m*∣(*b*−*c)*. Our result on divisibility of integral linear combinations, then, tells us that *m*∣(*a*−*b*)+(*b*−*c*)=*a*−*c*. Hence the definition of modular congruence tells us that *a* ≡ *c* mod *m*.

The benefit of showing that modular congruence is an equivalence relation is that this tells us that congruence class partition the integers into distinct sets. For instance, when the modulus is 3, we know that every integer fits into one of the three collections



Coming up with a collection of integers which represent all these possible classes, then, is an important task. This leads to the following

**Definition: A collection of integers is called a complete residue system for modulus m if every integer is congruent modulo m to exactly one element from the collection.**

Inverse Modulo (Extended Euclidean Algorithm)

The **modular multiplicative inverse** of an [integer](http://en.wikipedia.org/wiki/Integer) *a* [modulo](http://en.wikipedia.org/wiki/Modular_arithmetic) *m* is an integer *x* such that

ba59f6cfe6d0f7d2da74d853e71e6585.png

That is, it is the [multiplicative inverse](http://en.wikipedia.org/wiki/Multiplicative_inverse) in the [ring](http://en.wikipedia.org/wiki/Ring_(mathematics)) of integers modulo *m*. This is equivalent to

ff91eb2d2ca2c5ba793a94a7912d638b.png

The multiplicative inverse of *a* modulo *m* exists [if and only if](http://en.wikipedia.org/wiki/Iff) *a* and *m* are [coprime](http://en.wikipedia.org/wiki/Coprime) (i.e., if gcd(*a*, *m*) = 1). If the modular multiplicative inverse of *a* modulo *m* exists, the operation of [division](http://en.wikipedia.org/wiki/Division_(mathematics)) by *a* modulo *m* can be defined as multiplying by the inverse, which is in essence the same concept as division in the [field](http://en.wikipedia.org/wiki/Field_(mathematics)) of reals.

The modular multiplicative inverse of *a* modulo *m* can be found with the [extended Euclidean algorithm](http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm).

1.png

where *a*, *b* are given and *x*, *y*, and gcd(*a*, *b*) are the integers that the algorithm discovers. So, since the modular multiplicative inverse is the solution to

2.png

by the definition of congruence, *m* | *ax* − 1, which means that m is a [divisor](http://en.wikipedia.org/wiki/Divisor) of *ax* − 1. This, in turn, means that

3.png

Rearranging produces

4.png

with *a* and *m* given, *x* the inverse, and *q* an integer multiple that will be discarded. This is the exact form of equation that the extended Euclidean algorithm solves—the only difference being that gcd(*a*, *m*) = 1 is predetermined instead of discovered. Thus, *a* needs to be [co-prime](http://en.wikipedia.org/wiki/Coprime) to the modulus, or the inverse won't exist. The inverse is *x*, and *q* is discarded.

**C++ Code Inverse Modulo Using Extended Euclidean Algorithm:**

**#include <iostream>**

**#include <algorithm>**

**#include <stdio.h>**

**using namespace std;**

**int Extended\_Euclidean(int a,int b,int &x,int &y)**

**{**

**if(b==0)**

**{**

**x=1;y=0;**

**return a;**

**}**

**int d;**

**d=Extended\_Euclidean(b,a%b,y,x);**

**y=y-(a/b)\*x;**

**return d;**

**}**

**int Inverse\_Modulo(int a,int m)**

**{**

**int x,y,d;**

**d=Extended\_Euclidean(a,m,x,y);**

**if(d==1) return (x+m)%m; //Solution Exists**

**return -1; //No Solution**

**}**

**int main()**

**{**

**int ans,a,m;**

**while(scanf("%d %d",&a,&m))**

**{**

**ans=Inverse\_Modulo(a,m);**

**printf("%d\n",ans);**

**}**

**return 0;**

**}**

**//Input:**

**98 101**

**65 79**

**Output:**

**67**

**62**

Inverse Modulo (Using Fermat Little Theorem)

The modular multiplicative inverse of an integer **a modulo m** is an integer **x** such that :

ba59f6cfe6d0f7d2da74d853e71e6585.png

That is, it is the multiplicative inverse in the ring of integers modulo m. This is equivalent to

ff91eb2d2ca2c5ba793a94a7912d638b.png

The multiplicative inverse of a modulo m exists if and only if a and m are coprime (i.e., if gcd(a, m) = 1). Let’s see, Calculate Modular Multiplicative Inverse using Fermat Little Theorem:

Fermat’s little theorem states that if **m** is a prime and **a** is an integer co-prime to m, then:

**am-1 ≡ 1 (mod m)  
a-1am-1 ≡ a-1 (mod m)  
am-2 ≡ a-1 (mod m)**

That means: **(a-1 mod m) = (am-2 mod m)** **if m is prime**

**How Can we calculate the value of (am-2 mod m) ? Using Big Mod Algorithm: Here is C++ Code:**

**int BigMod(long long B,long long P,long long M)**

**{**

**long long R=1;**

**while(P>0)**

**{**

**if(P%2==1)**

**{**

**R=(R\*B)%M;**

**}**

**P/=2;**

**B=(B\*B)%M;**

**}**

**return R;**

**}**

**int Inverse\_Modulo(int a,int m)**

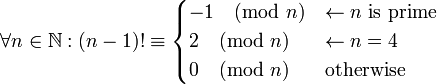
**{**

**return BigMod(a,m-2,m);**

**}**

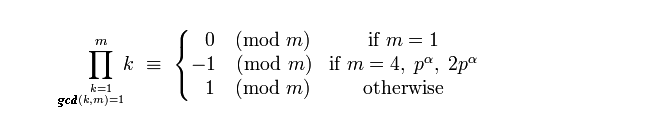
Wilson Theorem and Its Generalization

Wilson's theorem can be generalized to the following statement:



Gauss's generalization

The following is a stronger generalization of Wilson's theorem, due to [Carl Friedrich Gauss](http://en.wikipedia.org/wiki/Carl_Friedrich_Gauss):



Here **p** is an odd prime and **α** is positive integer.

**Problem ID: TJU 3618**

Divisibility Properties

**Deﬁnition:**. d | n means there is an integer k such that n = dk.

**Theorem:** (Divisibility Properties). If n, m, and d are integers then

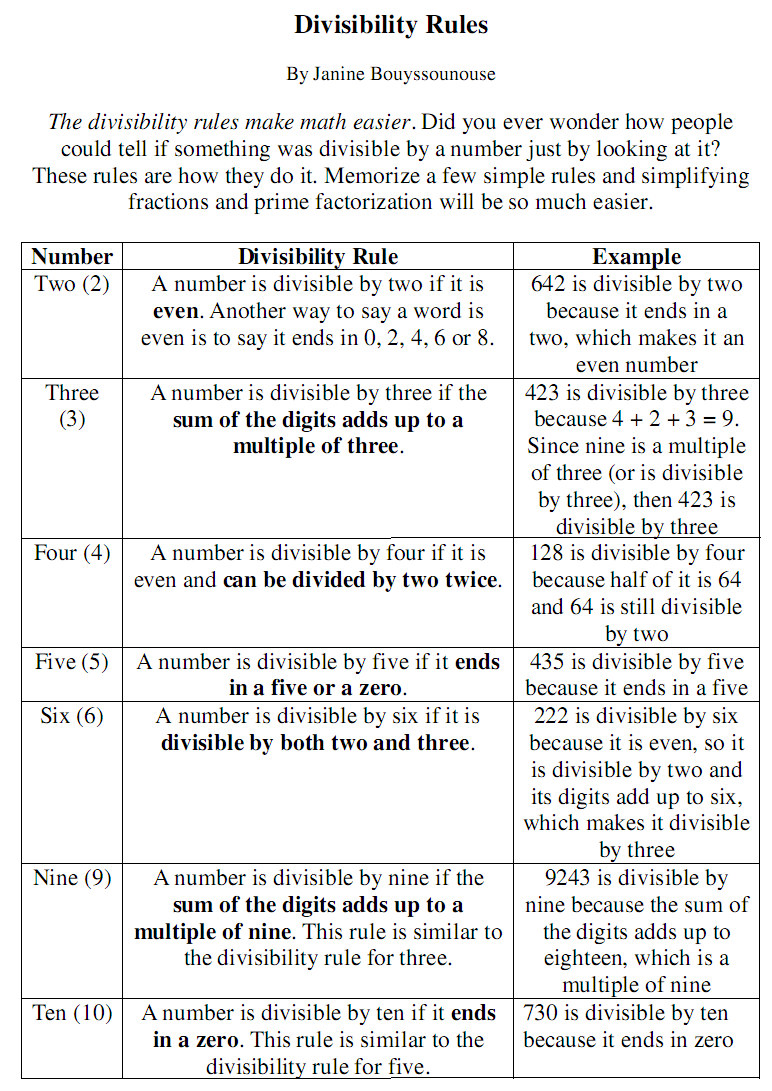
the following statements hold:

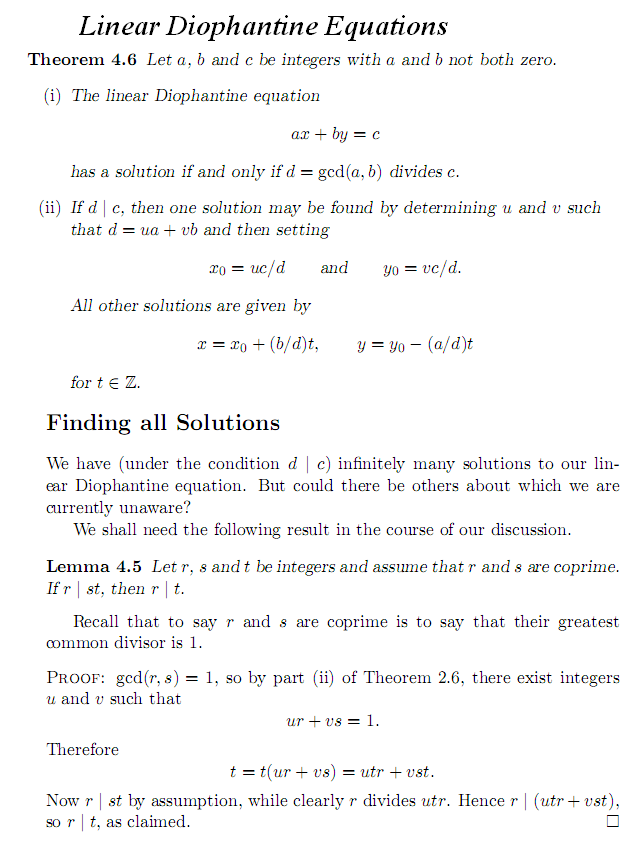
1. n | n (everything divides itself )

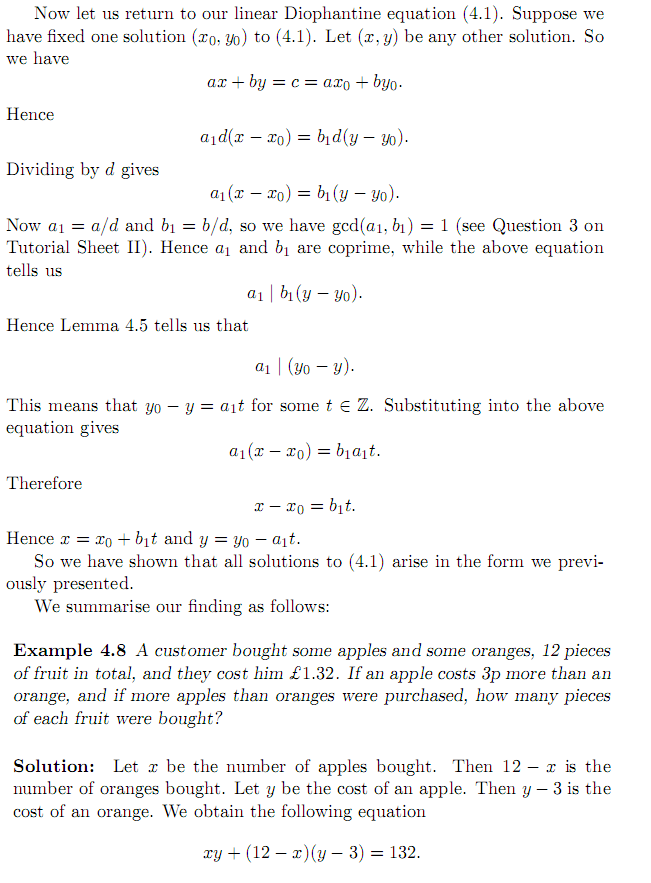
2. d | n and n | m =⇒ d | m (transitivity)

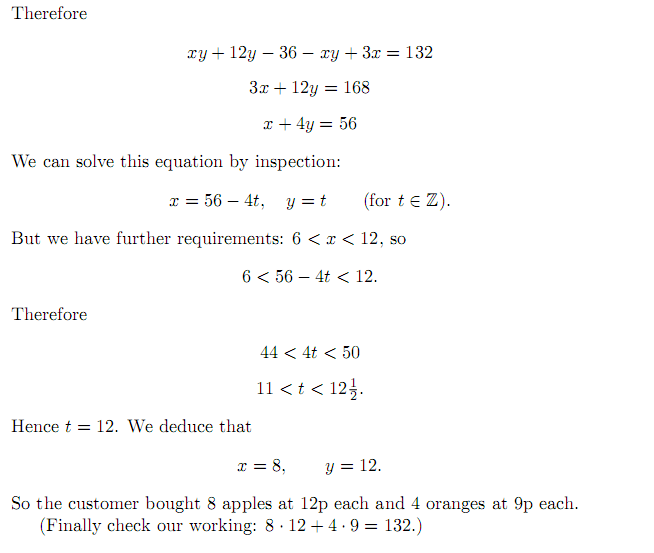
3. d | n and d | m =⇒ d | an + bm for all a and b (linearity property)

4. d | n =⇒ ad | an (multiplication property)









Problem – Marbles ([UVa 10090](http://uva.onlinejudge.org/external/100/10090.html))

|  |  |  |
| --- | --- | --- |
| 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 *c*1 Taka and can hold exactly *n*1 marbles  *Type* 2: each box costs *c*2 Taka and can hold exactly *n*2 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 <= n <= 2,000,000,000). The second line contains *c*1 and *n*1, and the third line contains *c*2 and *n*2. Here, *c*1, *c*2, *n*1and *n*2are 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 *m*1 and *m*2, where *mi*= number of *Type i* boxes required) if one exists, print "failed" otherwise.    If a solution exists, you may assume that it is unique.     |  |  | | --- | --- | | **Sample Input**  43 1 3 2 4 40 5 9 5 12 0 | **Sample Output**  13 1  failed |   **Solution:**  **n1m1 + n2m2 = n –---------------------------- (1)**    **Minimize -> c1m1 + c2m2**  **g = gcd(n1, n2)**  **n1m1' + n2m2' = g –-------------------------- (2)**  **Multiplying by (n/g) ->**  **n1m1'' + n2m2'' = n –------------------------ (3)**  **From (1) and (3) ->**  **m1 = m1'' + (n2/g) t**  **m2 = m2'' – (n1/g) t**  **Here, t is an integer parameter.**  **From the conditions -> m1 ≥ 0, m2 ≥ 0, n1 > 0, n2 > 0,**  **ceil(-m1'' g/n2) ≤ t ≤ floor(m2'' g/n1)**  **c = c1m1 + c2m2**  **= c1m1'' +c2m2'' + (c1n2/g – c2n1/g) t**  **= a + bt**  **As this is a linear function, its minimum value will be on either of the boundaries.** |

Modular Linear Equations

* Find solutions to the equation –

*ax = b (mod n)* where *a > 0* and *n > 0*

* This equation is solvable for the unknown *x* if and only if *gcd(a, n) | b*.
* This equation either has *d* distinct solutions modulo *n*, where *d = gcd(a, n)*, or it has no solutions.
* Let *d = gcd(a, n)* and suppose that *d = ax*' *+ ny*' for some integers *x'* and *y'*. If *d | b*, then the equation *ax = b (mod n)* has one of its solutions the value x0, where *x0 = x*'*(b/d) mod n*
* The other solutions of the equation would be –

*xi = x0 + i (n/d)* where *i = 0, 1, ……, d-1*

Algorithm:

**MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)**

**(d, x', y') = EXTENDED-EUCLID(a, n)**

**if d | b**

**x0 = x’(b/d) mod n**

**for i = 0 to d-1**

**print (x0 + i(n/d)) mod n**

**else**

**print “no solutions”**

Chinese Remainder Theorem

* Given a set of simultaneous congruences   
   *x = a1 (mod n1)* *x = a2 (mod n2)  
   …………….  
   …………….* *x = ai (mod ni)*
* For *i = 1, 2, …. k* and for which the *ni* are pairwise relatively prime, the unique solution of the set of congruences is –  
   *x = a1 b1 (N/n1) + …. + ak bk (N/nk) (mod N)*
* Where *N = n1 n2 …. Nk*
* And the bi are determined from

*bi (N/ni)= 1 (mod ni)*

*bi M= 1 (mod ni) [M= (N/ni)]*

*bi =M-1 (mod ni) [Multiply by M]*

*bi is the Inverse of M.*

**Practice Problems:**

**LOJ:** 1054, 1067, 1102, 1306, 1007, 1138, 1259, 1278, 1289, 1319, 1306

**UVa:** 10236, 10329, 10680, 10789, 10924, 10948, 11191, 11347, 11408, 11440, 10090,

**TJU:** 3618, 1233, 1375, 1476, 1528, 1637, 1698, 1730, 1748, 1991, 2218, 2308, 2502, 2520, 2526, 2601, 2648, 2658, 2859, 2901, 2916, 3043, 3076, 3150, 3232, 3237, 3238, 3259, 3261, 3262, 3288, 3293, 3467, 3483, 3496, 3599,3618

**Prepared By:**

Forhad Ahmed( Email: [forhadsustbd@gmail.com](mailto:forhadsustbd@gmail.com) )