Contents:

1.Sieve of Eratosthenes

2.Segmented sieve

3.Bitwise sieve

4.How many divisors

5.Sum of divisors

6.Goldbach’s conjecture

7.Euler’s totient phi

8.How many digits x^y

9.how many digits N!

10.Trailing zeroes of factorial

11.Big mod

12.sum of digits

13.Primarity test:non deterministic(Miller-rabin)

14.Articulation points

15.Ternary search

Sieve of Eratosthenes

In [mathematics](http://en.wikipedia.org/wiki/Mathematics), the **sieve of Eratosthenes**, one of a number of [prime number sieves](http://en.wikipedia.org/wiki/Prime_number_sieve), is a simple, ancient [algorithm](http://en.wikipedia.org/wiki/Algorithm) for finding all [prime numbers](http://en.wikipedia.org/wiki/Prime_number) up to any given limit. It does so by iteratively marking as [composite](http://en.wikipedia.org/wiki/Composite_number) (i.e. not prime) the multiples of each prime, starting with the multiples of 2.

**#include <bits/stdc++.h>**

**#define Max 100**

**using namespace std;**

**bool flag[Max+5];**

**int primes[Max+5];**

**int seive()**

**{**

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

**for(i=2;i<=Max;i++) flag[i]=1;**

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

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

**if(flag[i])**

**for(j=i;j\*i<=Max;j++)**

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

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

**if(flag[i])**

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

**return total;**

**}**

**int main()**

**{**

**int total=seive();**

**printf("Total Primes: %d\n",total);**

**for(int i=0;i<total;i++)**

**printf("%d\n",primes[i]);**

**return 0;**

**}**

**// if SIZE\_N 100 Total Primes: 25**

**// if SIZE\_N 1000 Total Primes: 168**

**// if SIZE\_N 1000000 Total Primes: 78498**

**Mycode:for sieve()**

**void seive()**

**{**

**memset(table,false,sizeof(table));**

**table[0]=table[1]=true;**

**int i,j;**

**for(i=4;i<=10000000;i+=2)   table[i]=true;**

**for(i=3;i\*i<=10000000;i+=2)**

**if(table[i]==0)**

**for(j=i\*i;j<=10000000;j+=i)**

**table[j]=true;**

**prime[0]=2;**

**for(i=3,j=0;i<10000000;i+=2)**

**if(table[i]==false)  prime[++j]=i;**

**}**

**\*\*Another implementation:**

1. **#include <bits/stdc++.h>**
2. **#define ll long long**
3. **using namespace std;**
4. ***//bool flag[Max];***
5. **vector<int>primes;**
6. **void sieve()**
7. **{**
8. **const int SZ = 1e6 + 3;**
9. **bitset<SZ> status;**
10. **int rt = 1e3;**
12. **for(int i = 3; i < rt; i+=2)**
13. **if(!status[i])**
14. **for(int j = i\*i; j < SZ; j+=(i<<1))**
15. **status[j] = true;**
17. **primes.push\_back(2);**
19. **for(int i = 3; i < SZ; i+=2)**
20. **if(!status[i])**
21. **primes.push\_back(i);**
22. **}**

SEGMENTED SIEVE

For making the idea clear,the article below can be helpful

It means, that we are making segmented with the sieve we have. I once saved this text for future: "Say i need to find the prime numbers between 125 and 140 .One way is to find all primes till 140 using traditional sieves and then find how many of them are greater than 125.This will not work within limited time constraints.We need to fit the Sieve to our needs so that we run the Sieve only in that particular range.

The first prime is 2.We divide the starting number x=125 by 2 .We round off to smaller integer we get 62.We again multiply by 2 we get 124.What is 124 ?it is the first smaller number than x that is divisible by the prime 2.We start from 124,increment by 2 in each step and remove all elements between 125 and 140.That is we remove 126,128,130,132,134,136,138,140.What we did was that we did the first step in traditional sieve but just offset the starting elements to the closest composite number less than the starting point of the range.Next we do this with the 2nd smallest prime number 3.125/3=41.41\*3=123.We start from 123 go till 140 (inclusive) in steps of 3 and cut the numbers 126,129,132,135,138.Next we do with 5.But from where do we get the prime numbers 2,3,5 and so on and how long do we need to do that? That can be done well by generating the sieve upto 10^6 which can be easily done". Hope it clears!!

My code goes here:

The best article for me is in the given link:

http://zobayer.blogspot.com/2009/09/segmented-sieve.html

**#include <bits/stdc++.h>**

**#define MAX 46656**

**#define LMT 216**

**#define LEN 4830**

**#define RNG 100032**

**#define ll long long int**

**using namespace std;**

**unsigned base[MAX/64], segment[RNG/64], primes[LEN];**

**#define sq(x) ((x)\*(x))**

**#define mset(x,v) memset(x,v,sizeof(x))**

**#define chkC(x,n) (x[n>>6]&(1<<((n>>1)&31)))**

**#define setC(x,n) (x[n>>6]|=(1<<((n>>1)&31)))**

**void sieve()**

**{**

**unsigned i, j, k;**

**for(i=3; i<LMT; i+=2)**

**if(!chkC(base, i))**

**for(j=i\*i, k=i<<1; j<MAX; j+=k)**

**setC(base, j);**

**for(i=3, j=0; i<MAX; i+=2)**

**if(!chkC(base, i))**

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

**}**

**int segmented\_sieve(ll a, ll b)**

**{**

**ll i, j, k, cnt;**

**cnt=(a<=2&&2<=b) ?1:0; // for handling case a=2 and b=2 or a=1 and b=2;**

**if(b<2) return 0;**

**if(a<3) a = 3;**

**if(a%2==0) a++;**

**mset(segment,0);**

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

**{**

**j = ( (a+primes[i]-1) / primes[i] );**

**j=j\*primes[i];**

**if(j%2==0) j += primes[i];**

**for(k=primes[i]<<1; j<=b; j+=k) //seive er moto k=primes[i]\*2;**

**if(j!=primes[i])**

**setC(segment, (j-a));**

**}**

**for(i=0; i<=b-a; i+=2)**

**if(!chkC(segment, i))**

**cnt++;**

**return cnt;**

**}**

**int main(){**

**ll a,b,tc;**

**ll t=1;**

**sieve();**

**cin>>tc;**

**while(tc--){**

**cin>>a>>b;**

**a= segmented\_sieve(a, b);**

**cout<<"Case "<<t++<<": "<<a<<endl;**

**}**

**return 0;**

**}**

* BITWISE SIEVE FOR GENERATING PRIMES UPTO 10^9

Link:

* <http://paste.ofcode.org/Htv2YhvsPJ8R5rQg2eZjfb>
* <http://lbv-pc.blogspot.com/2012/08/goldbach-and-euler.html>

When you come across a problem titled after two well-known, brilliant mathematicians, you know it’s going to be good :).

What the problem is asking for can be expressed as follows: given an integer **n** (which can go as high as 108), find —if possible— two prime numbers **p1** and **p2** such that:

* *p*1+*p*2=*n*

  *p*2>*p*1

  *p*2−*p*1

* is minimised

It is clear that for this problem it is necessary to find out which numbers are primes in the range (0:108)

. Let’s focus on that first.

As you probably know, one good way to find prime numbers computationally is using the classic [Sieve of Eratosthenes](http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) (or a similar but possibly faster alternative, such as the [Sieve of Atkin](http://en.wikipedia.org/wiki/Sieve_of_Atkin)). No matter what kind of “sieve” you use, you have to implement it in a way that allows you to store information about 108 numbers, which is not a trivial job.

Let us generalise the problem and say that you want to find all prime numbers up to an integer **N**. If you write a regular implementation using a boolean (*bool* in C++ or similar) data type for each integer in the range [1:*N*] then you would end up using *N* bytes of memory, which for this problem would mean about 95MB —just “a tad” heavy.

Fortunately, there are ways to reduce the memory requirements with relatively few changes that are easy to code. The main idea is that a boolean value can be stored in a single bit, so you can store 8 boolean values in a single byte. In addition, consider that most prime numbers are odd; in fact, there is only one even prime number which is 2. With this in mind, you can further cut your memory requirements in half by simply not storing information for even numbers (you have to handle the case of 2 separately, of course).

So, with these adjustments, we can now implement a prime-finding sieve that requires only *N*÷(8∗2)

bytes. If we store this information in 4-byte integers, then we require only *N*÷64 integers. The following is a C++ implementation of this type of sieve, using the standard Eratosthenes algorithm, which has served me well for some time:

const int MAX = 100000000;  // 10^8

const int LMT =     10000;  // sqrt(MAX)

// array of data; each bit tells if a certain number is composite or not

int \_c[(MAX>>6)+1];

// we use a STL vector to store the primes

vector<int> primes;

// possibly the most important piece for a bitwise sieve

// these are the macros that check and set a bit

#define IsComp(n)  (\_c[n>>6]&(1<<((n>>1)&31)))

#define SetComp(n) \_c[n>>6]|=(1<<((n>>1)&31))

void prime\_sieve() {

    for (int i = 3; i <= LMT; i += 2)

        if (!IsComp(i))

            for (int j = i\*i; j <= MAX; j += i+i)

                SetComp(j);

    primes.push\_back(2);

    for (int i=3; i <= MAX; i += 2)

        if (!IsComp(i))

            primes.push\_back(i);

}

bool is\_prime(int n) {

    if (n < 2 || n % 2 == 0) return false;

    return ! IsComp(n);

}

As you can see, this implementation reserves about 108÷64 integers for the main array, which represents about 6MB of memory, a good deal better than the original 95MB. However, it also uses memory to store the actual prime numbers in a vector. Since there are about 5.8 million primes under 108, then this would require an extra 4×5.8×106

bytes (about 22MB).

Having solved the problem of calculating and storing the primes, now let’s move on to the problem of finding *p*1

and *p*2. The first idea that comes to mind is something like simply trying to find *p*1 starting with *n*÷2 and going downwards, checking that *p*1 and *p*2=*n*−*p*1

are primes.

However, it’s easy to see that this simple idea would take too long in practise. There’s one thing that we could do that would improve significantly the complexity of the algorithm, which is not iterating through *all* integers, not even through odd integers only, but only over the prime numbers. Since we already have a vector full of primes below 108, we can binary search the location of *n*÷2

and go down from it. That would make for a worse case of 5.8×106 iterations instead of 108÷2

.

We have improved things but, can we do even better? The discussion found in the problem statement about Goldbach’s conjecture and Euler’s ideas on the subject is very interesting on itself, but it also points to something that could be useful: an even number *can* always be expressed as the sum of two primes (well, it’s a conjecture, but it has been tested for a range of numbers much larger than (0:108)

, so we can assume it’s true for our purposes). This means that we can be reasonably confident that the process of finding *p*1 will succeed for most even numbers, hopefully quickly enough that it doesn’t have to check millions of primes. Note that I said *most* even numbers; this is because we need two *distinct* primes *p*1 and *p*2

, remember? For example, 6 can be expressed as the sum of two primes (3 + 3) but not as the sum of two distinct primes.

Okay, we have concluded that for even numbers our algorithm may not be so bad, but what about odd numbers? Well, it turns out that for odd numbers the algorithm can be *O*(1)

! If *n* is odd, then one of the two primes *p*1 and *p*2 *has* to be even, while the other has to be odd (if you don’t see this clearly, think about how even numbers are integers of the form 2*k* and odd numbers are integers of the form 2*k*+1). Do you see where this is going? If one of the primes has to be even, then that prime would have to be 2 (there are no other even primes). So, if *n* is odd we simply check if *n*−2 is prime, and if it is, then we know that *p*1=2 and *p*2=*n*−2

, otherwise there is no solution.

We have covered all possible cases now, and after testing it, it produces an answer reasonably fast. Prime numbers are lovely :).

How Many Divisors of a Number

Suppose you wish to find the number of divisors of 48. of [prime number sieves](http://en.wikipedia.org/wiki/Prime_number_sieve) Starting with 1 we can work through the set of natural numbers and test divisibility in each case, noting that divisors can be listed in factor pairs.

48 = 1×48 = 2×24 = 3×16 = 4×12 = 6×8

Hence we can see that 48 has exactly ten divisors. It should also be clear that, using this method, we only ever need to work from 1 up to the square root of the number.

Although this method is quick and easy with small numbers, it is tedious and impractical for larger numbers. Fortunately there is a quick and accurate method using the divisor.

Let D(N) be the number of divisors for the natural number, N.

We begin by writing the number as a product of prime factors:

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

Then the number of divisors: D(N) = (q1+1) x (q2+1) x (q3+1) x. . . . . . . . . x (qk+1)

The number of divisor function can be quickly demonstrated with the example we considered earlier: 48 = 24×31, therefore D(48)=5×2=10.

**#include <stdio.h>**

**#include <math.h>**

**#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 divisor(int N)**

**{**

**int i,val,count,sum;**

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

**sum=1;**

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

**{**

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

**{**

**count=0;**

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

**{**

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

**count++;**

**}**

**sum\*=(count+1);**

**}**

**}**

**if(N>1)**

**sum=sum\*2;**

**return sum;**

**}**

**int main()**

**{**

**int total=seive();**

**int n;**

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

**{**

**printf("%d No Of divisor:%d\n",n,divisor(n));**

**}**

**return 0;**

**}**

**// 5 No Of divisor:2**

**// 8 No Of divisor:4**

**// 100 No Of divisor:9**

**// 568 No Of divisor:8**

**// 48 No Of divisor:10**

**How many divisors my code:**

**ll sol(ll N)**

**{**

**ll i,cnt,sum;**

**int sz=primes.size();**

**sum=1;**

**for(i=0;primes[i]\*primes[i]<=N&&i<sz;i++)**

**{**

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

**{**

**count=0;**

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

**{**

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

**count++;**

**}**

**sum\*=(count+1);**

**}**

**}**

**if(N>1){**

**sum=sum\*2;**

**}**

**return sum;**

**}**

Sum of Divisors

Imagine you wish to work out the sum of divisors of the number 72. It would not take long to list the divisors, and then find their sum: 1 + 2 + 3 + 4 + 6 + 8 + 9 + 12 + 18 + 24 + 36 + 72 = 195.

However, this method would become both tedious and difficult for large numbers like 145600. Fortunately, there is a simple and elegant method at hand.

Let σ(n) be the sum of divisors of the natural number, n.

For any prime, p: σ(p) = p + 1, as the only divisors would be 1 and p.

Consider pa: σ(pa) = 1 + p + p2 + ... + pa (1).

Multiplying by p: pσ(pa) = p + p2 + p3 + ... + pa + 1 (2).

Subtracting (1) from (2): pσ(pa)−σ(pa) = (p−1)σ(pa) = pa+1 − 1.

Hence σ(pa) = (pa+1 − 1)/(p − 1).

For example, σ(34)=(35−1)/(3−1) = 242/2 = 121,  
and checking: 1 + 3 + 9 + 27 + 81 = 121.

Although no proof is supplied here, the usefulness of the function, σ(n), is its multiplicativity, which means that σ(a×b×...)=σ(a)×σ(b)×..., where a, b, ..., are relatively prime.

General Form:

If the prime power decomposition of an integer is

a1.png

Then we can write

a2.png

Returning to example, we use the fact that σ(72) = σ(23×32). As 23 and 32 are relatively prime, we can separately work out σ(23) = 24 − 1 = 15 and σ(32) = (33 − 1)/2 = 13. Therefore, σ(72) = 15×13 = 195.

**#include <stdio.h>**

**#include <math.h>**

**#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 Sum\_Of\_Divisor(int N)**

**{**

**int i,val,count,sum,p,s;**

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

**sum=1;**

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

**{**

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

**{**

**p=1;**

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

**{**

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

**p=p\*primes[i];**

**}**

**p=p\*primes[i];**

**s=(p-1)/(primes[i]-1);**

**sum=sum\*s;**

**}**

**}**

**if(N>1)**

**{**

**p=N\*N;**

**s=(p-1)/(N-1);**

**sum=sum\*s;**

**}**

**return sum;**

**}**

**int main()**

**{**

**int total=seive();**

**int n;**

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

**{**

**printf("%d Sum of Of divisor:%d\n",n,Sum\_Of\_Divisor(n));**

**}**

**return 0;**

**}**

**// 48 Sum of Of divisor:124**

**// 72 Sum of Of divisor:195**

Goldbach's Conjecture

For any integer *n* (*n* ≥ 4) there exist two prime numbers *p*1 and *p*2 such that *p*1 + *p*2 = *n*. In a problem we might need to find the number of essentially different pairs (*p*1, *p*2), satisfying the condition in the conjecture for a given even number*n* (4 ≤ *n* ≤ 215). (The word ‘essentially’ means that for each pair (*p*1, *p*2) we have *p*1≤*p*2.)For example, for *n* = 10 we have two such pairs: 10 = 5 + 5 and 10 = 3 + 7.

To solve this,as *n* ≤ 215 = 32768, we’ll fill an array primes[32768] using function seive. We are interested in primes, not greater than 32768.

The function *FindSol*(*n*) finds the number of different pairs (*p*1, *p*2), for which *n* = *p*1 + *p*2. As *p*1 ≤ *p*2, we have *p*1 ≤ *n*/2. So to solve the problem we need to find the number of pairs (*i*, *n* – *i*), such that *i* and *n* – *i* are prime numbers and 2 ≤ *i* ≤ *n*/2.

**int FindSol(int n)**

**{**

**int i, res=0;**

**for(i=2;i<=n/2;i++)**

**if(flag[i] && flag[n-i])**

**res++;**

**return res;**

**}**

**My solution:**

 for(i=0;prime[i]<=n/2;i++)

         {

             if(table[n-prime[i]]==**false**)    count++;

         }

Euler’s totient function

The number of positive integers, not greater than *n*, and relatively prime with *n*, equals to Euler’s totient function φ (*n*). In symbols we can state that

φ (*n*) ={*a* Î N: 1 ≤ *a* ≤ *n*, gcd(*a*, *n*) = 1}

## Properties

The following three simple properties of the Euler - enough to learn how to calculate it for any of:

* If p- is prime, then \ Phi (p) = p-1.

(This is obvious, since any number, except for the prelatively simple with him.)

* If p- a simple, a- a natural number, then \ Phi (p ^ a) = p ^ ap ^ {a-1}.

(Since the number of p ^ anot only the numbers are relatively prime form , which shares.)pk (K \ in \ mathcal {N})p ^ a / p = p ^ {a-1}

* If aand bare coprime, then \ Phi (ab) = \ phi (a) \ phi (b)("multiplicativity" of Euler's function).

(This follows from [the Chinese Remainder Theorem](http://e-maxx.ru/algo/chinese_theorem) . Let us consider an arbitrary number z \ le ab. Let xand ythe remnants of the division zat aand b, respectively. Then zis relatively prime to abif and only if zis prime to aand bindividually, or, equivalently, xone simply aand yrelatively prime b. Applying the Chinese Remainder Theorem, we see that any pair of numbers xand number-one matches , which completes the proof.)y (X \ le a, ~ y \ le b)z (Z \ le ab)

From this we can derive the function for each Euler \ It nthrough its **factorization** (decomposition ninto prime factors):

if

 n = p_1 ^ {a_1} \ cdot p_2 ^ {a_2} \ cdot \ ldots \ cdot [...]

(Where everything is p_i- simple), then

 \ Phi (n) = \ phi (p_1 ^ {a_1}) \ cdot \ phi (p_2 ^ {a_2}) \ [...]  
 = (P_1 ^ {a_1} - p_1 ^ {a_1-1}) \ cdot (p_2 ^ {a_2} - p_ [...]  
 = N \ cdot \ left (1 - {1 \ over p_1} \ right) \ cdot \ le [...]

The function fi(*n*) finds the value of φ(*n*):

For example, to find φ(616) we need to factorize the argument: 616 = 23 \* 7 \* 11. Then, using the formula, we’ll get:

φ(616) = 616 \* (1 – 1/2) \* (1 – 1/7) \* (1 – 1/11) = 616 \* 1/2 \* 6/7 \* 10/11 = 240.

Say you've got a problem that, for a given integer *n* (0 < *n* ≤ 109), asks you to find the number of positive integers less than *n* and relatively prime to *n*. For example, for *n* = 12 we have 4 such numbers: 1, 5, 7 and 11.

The solution: The number of positive integers less than *n* and relatively prime to *n* equals to φ(*n*). In this problem, then, we need do nothing more than to evaluate Euler’s totient function.

Or consider a scenario where you are asked to calculate a function Answer(*x*, *y*), with *x* and *y* both integers in the range [1, *n*], 1 ≤*n* ≤ 50000. If you know Answer(*x*, *y*), then you can easily derive Answer(*k*\**x*, *k*\**y*) for any integer *k*. In this situation you want to know how many values of Answer(*x*, *y*) you need to precalculate. The function Answer is not symmetric.

For example, if *n* = 4, you need to precalculate 11 values: Answer(1, 1), Answer(1, 2), Answer(2, 1), Answer(1, 3), Answer(2, 3), Answer(3, 2), Answer(3, 1), Answer(1, 4), Answer(3, 4), Answer(4, 3) and Answer(4, 1).

The solution here is to let res(*i*) be the minimum number of Answer(*x*, *y*) to precalculate, where *x*, *y* Î{1, …, *i*}. It is obvious that res(1) = 1, because if *n* = 1, it is enough to know Answer(1, 1). Let we know res(*i*). So for *n* = *i* + 1 we need to find Answer(1, *i* + 1), Answer(2, *i* + 1), … , Answer(*i* + 1, *i* + 1), Answer(*i* + 1, 1), Answer(*i* + 1, 2), … , Answer(*i* + 1, *i*).

The values Answer(*j*, *i* + 1) and Answer(*i* + 1, *j*), *j* Î{1, …, *i* + 1}, can be found from known values if GCD(*j*, *i* + 1) > 1, i.e. if the numbers *j* and *i* + 1 are not common primes. So we must know all the values Answer(*j*, *i* + 1) and Answer(*i* + 1, *j*) for which *j* and *i* + 1 are coprime. The number of such values equals to 2 \* φ (*i* + 1), where φ is an Euler’s totient function. So we have a recursion to solve a problem:

res(1) = 1,  
res(*i* + 1) = res(*i*) + 2 \* j (*i* + 1), *i* > 1

### Euler Totient / φ Function:

Euler Totient / Phi Function φ(n) counts the number of positive integers less than or equal to n which are relatively prime to n, i.e. do not have any common divisor with n except 1.

Formula for Euler Phi function:

[http://1.bp.blogspot.com/-KC-rhVEztZY/US5LZ7tLRmI/AAAAAAAAAWg/hmI9Z058TQg/s320/Capture.PNG](http://1.bp.blogspot.com/-KC-rhVEztZY/US5LZ7tLRmI/AAAAAAAAAWg/hmI9Z058TQg/s1600/Capture.PNG)

Here, the product is over the distinct prime numbers which divide n. Now, you can just factorize n and calculate φ(n) pretty easily. But, will that be efficient for a task such as you are asked to find φ(n) over a range of integers?

If you look closely to the formula, you will see that, we multiply n with (p-1)/p for each prime p that divides n. Now recall what do we do when we run Sieve of Eratosthenes for marking primes / non-primes. On the outer loop of the sieve, we determine if a number is prime, then in inner loop, instead of setting flags, if we can keep multiplying the number with (p-1)/p where p is the current prime number from outer loop, at the end of the iterations, we can actually generate φ(n) for each n over the range we are asked for. Here is a source code example that does exactly the same thing. Just take a look and try to understand what are the loops doing here, and how we are performing calculation and storing results.

A bit of explanation on what we are doing here: Initially the phi[] array is set to 0 (as it is declared global). We know that, phi[1] = 1 and phi[n] = n-1 when n is a prime number. So, similar to sieve algorithm, first we check if current number is prime in the outer loop, if phi[i] = 0, it means i is prime. So, we update it with i-1 accordingly. Now, for all the multiples of i greater than i, which starts from 2\*i, calling it j in inner loop, we need to multiply phi[j] by (i-1) / i. Here, we first check if phi[j] is 0, i.e. visiting it for the first time, in this case we set it with j, as I said earlier that, for φ(n) we will multiply n with (p-1)/p where p are the primes that divide n. Also, one thing to note here: a \* b / c and a / c \* b are not always same for integer calculation unless you can assure that c divides a. In this case it does, why? cause this is basically a prime factoring algorithm, and c = i here. As we are traversing i's multiples, it is guaranteed that a=phi[j] can be divided by c=i and instead of a \* b / c format, we will always use a / c \* b in these types of situations, because it will help preventing overflow many times.

Now, think about the optimizations we could apply here, and try applying them, like discarding even numbers, starting inner loop from squares, increment inner loop twice the prime number each time, won't work here. Because, we need to go through every numbers in inner loop, as we are trying to find Totient function for every n in the range 1 to MAX. Test the code on smaller range, and try to check if it is doing this correctly, play around with it.

#include <cstdio>

const int MAX = 1000001;

int phi[MAX];

void euler\_phi() {

phi[1] = 1;

for(int i=2; i<MAX; i++) {

if(!phi[i]) {

phi[i] = i-1;

for(int j=(i<<1); j<MAX; j+=i) {

if(!phi[j]) phi[j] = j;

phi[j] = phi[j]/i\*(i-1);

}

}

}

}

int main() {

euler\_phi();

for(int t=1; t<MAX; t++) printf("%d\n", phi[t]);

return 0;

}

LINK FOR EULERS PHI: http://zobayer.blogspot.com/2013/02/euler-totient-function.html

Euler’s totient theorem

If n is a positive integer and a is coprime to n, then a φ (n)  1 (mod n).

**Fermat’s little theorem**

If p is a prime number, then for any integer a that is coprime to n, we have

ap ≡ a (mod p)

This theorem can also be stated as: If p is a prime number and a is coprime to p, then

a *p*-1 ≡ 1 (mod p)

Fermat’s little theorem is a special case of Euler’s totient theorem when n is prime.

How Many Digits of XY

Let, No. of Digits D.

Then, D=floor [ log10(XY) ]+1

=floor [ Y x log10(X) ] +1

How Many Digits of N!

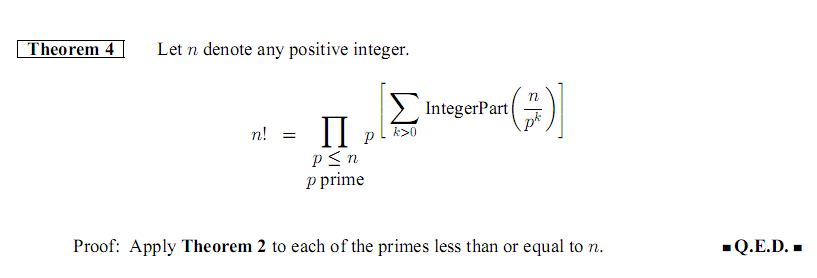
Let, Number of Digits D

D=floor[log10(N!)]+1

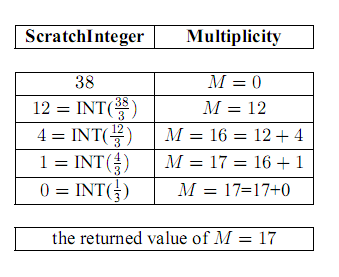
=floor[log10(1x2x3x…………xN)]+1

=floor[log10(1) + log10(2) + ……………+log10(N)]+1

Factorial Factors



**Example: 38! And prime 3**



Trailing Zeros of a Factorial

**#include <stdio.h>**

**int Trailing\_Zeros(int N)**

**{**

**int sum=0;**

**while(N)**

**{**

**sum+=N/5;**

**N=N/5;**

**}**

**return sum;**

**}**

**int main()**

**{**

**int N,ans;**

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

**{**

**ans=Trailing\_Zeros(N);**

**printf("Trailing Zeros of %d is %d\n",N,ans);**

**}**

**}**

BigMod

অনেক সময় বিভিন্ন প্রবলেম সল্ভ করার সময় আমাদের প্রায়ই বিগ মোড করার প্রয়োজন হয়। যেমন, বলা হল (2^30)%11 এর ভ্যালু কি। এর জন্য আমাদের বিগ মোড অ্যালগোরিদম জানা প্রয়োজন।

বিগ মোড অ্যালগোরিদমকে modulus multiplication এর ব্যাসিক প্রপার্টির extension বলা যায়। বিগ মোড অ্যালগোরিদমের ভেতরে যাওয়ার আগে আমাদের modulus arithmetic এর একটা সহজ প্রপার্টি জানা প্রয়োজন।

Modulus arithmetic অনুসারে (a\*b)%c কে এইভাবে লেখা যায়ঃ

(a\*b)%c = ((a%c)\*(b%c))%c

যেমন, (15\*16)%7 = 2

আবার, (15%7 \* 16%7) % 7  = (1 \* 2) % 7 = 2%7 = 2

বিগ মোড অ্যালগোরিদমের জন্য শুধুমাত্র এই সুত্রটাই যথেষ্ট।

2^30 কে আমরা ২ভাগে ভাগ করতে পারিঃ       (2^15)\*(2^15)

তাহলে, (a\*b)%c = ((a%c)\*(b%c))%c  এই সূত্র অনুসারে,

(2^30)%11  = (  (2^15) \*  (2^15) )%11

= (  ( (2^15) % 11 ) \* ( (2^15) % 11 ) )%11

আমরা 2^15 কে আবার একইভাবে ২ভাগে ভাগ করতে পারিঃ   2\*(2^14)

এখানে একটা বিষয় লক্ষণীয়, **power কখনো বেজোড় হলে তাকে জোড় করে নেওয়া হয়েছে**; এতে কাজে সুবিধা হয়।

এখন, (2^15)%11 = ( (2%11) \* ( (2^14)%11 ) ) % 11

এভাবে কাজ করতে থাকে আমরা যেটা পাই,

শুরতে, 2^30 = (2^15) \* (2^15)

Then, 2^15  = 2 \* (2^14)

Then, 2^14 = (2^7) \* (2^7)

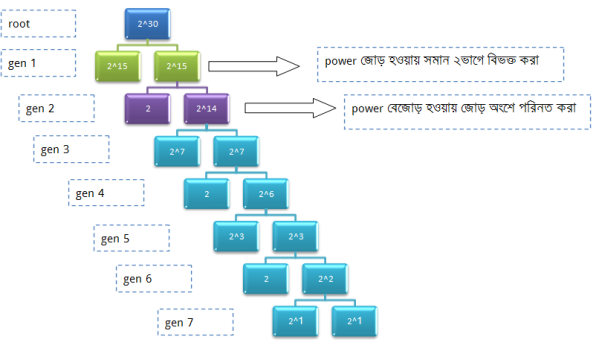
Then, 2^7 = 2 \* (2^6)

Then 2^6 = (2^3) \* (2^3)

Then 2^3 = 2 \* (2^2)

Then 2^2 = 2\*2

বিষয়টা অনেকটা এরকমঃ

[](https://imranshabijabi.files.wordpress.com/2012/11/1232.png)

এখন আমরা দেখব কিভাবে এই figure থেকে (2^30)%11 এর ভ্যালু পাওয়া যায়।

আমরা gen 7 থেকে আমাদের কাজ শুরু করব।

gen 7: ((2^1)\*(2^1))%11

2^1 = 2

2%11 = 2

সুতরাং gen 7 = ((2^1)\*(2^1))%11

= (2\*2)%11

= 4

gen 6: (2 \* (2^2)) %11 = ( (2%11) \* ((2^2)%11) )%11

এখানে, 2%11 = 2

(2^2)%11 = ((2^1)\*(2^1))%11 [যেটা প্রকতপক্ষে gen 7]

= 4 [gen 7 থেকে প্রাপ্ত]

সুতরাং gen 6 =  (2 \* (2^2)) %11 [লক্ষণীয়, 2 \* (2^2) = 2^3]

= ( (2%11) \* ((2^2)%11) )%11

= ( 2 \* 4 ) % 11

= 8

gen 5: ((2^3)\*(2^3))%11 = ( (2^3)%11) \* ((2^3)%11) )%11

এখানে, ( (2^3)%11) = (2 \* (2^2)) %11 [যেটা প্রকতপক্ষে gen 7]

= 8

তাহলে, gen 5 =  ((2^3)\*(2^3))%11

= ( (2^3)%11) \* ((2^3)%11) )%11

= ( 8 \* 8 ) % 11

=  9

gen 4: (2 \* (2^6)) %11 = ( (2%11) \* ((2^6)%11) )%11

এখন, (2^6)%11 = ((2^3)\*(2^3))%11

= gen 5

= 9

তাহলে, gen 4 = (2 \* (2^6)) %11

= ( (2%11) \* ((2^6)%11) )%11

=  ( 2 \* 9 ) % 11

= 7

gen 3: ((2^7)\*(2^7))%11 = ( (2^7)%11) \* ((2^7)%11) )%11

এখানে, ( (2^7)%11) = (2 \* (2^6)) %11 [যেটা প্রকতপক্ষে gen 4]

= 7

তাহলে, gen 3 =  ((2^7)\*(2^7))%11

= ( (2^7)%11) \* ((2^7)%11) )%11

= ( 7 \* 7 ) % 11

= 5

gen 2: (2 \* (2^14)) %11 = ( (2%11) \* ((2^14)%11) )%11

এখন, (2^14)%11 = ((2^7)\*(2^7))%11

= gen 3

= 5

তাহলে, gen 2 = (2 \* (2^14)) %11

= ( (2%11) \* ((2^14)%11) )%11

=  ( 2 \* 5 ) % 11

= 10

gen 1: ((2^15)\*(2^15))%11 = ( (2^15)%11) \* ((2^15)%11) )%11

এখানে, ( (2^15)%11) = (2 \* (2^14)) %11 [যেটা প্রকতপক্ষে gen 2]

= 10

তাহলে, gen 1 =  ((2^15)\*(2^15))%11

= ( (2^15)%11) \* ((2^15)%11) )%11

= ( 10 \* 10 ) % 11

= 1

root: (2^30)%11 = ((2^15)\*(2^15))%11

= gen 1

= 1

সুতরাং আমরা বলতে পারি, (2^30)%11 = 1

বিষয়টাকে খুব সহজে coding এ রুপান্তরিত করা যায়; এর জন্য আমরা recursion এর সাহায্য নেব ।

coding এ রুপান্তরের জন্য আমাদের ২টা বিষয় check করতে হবে : power জোড় না বেজোড় ।

Big Mod এর Function টা এরকম ঃ

 int big\_mod(int base, int power, int mod)

{

    if(power==0)    return 1;

    //কোন কিছুর power 0 হলে তার মান 1 হয়ে যায়

    else if(power%2==1) //power বেজোড় হলে

    {

        int p1 = base % mod;

        int p2 = (big\_mod(base,power-1,mod))%mod;

        return (p1\*p2)%mod;

    }

    else if(power%2==0) //power জোড় হলে

    {

        int p1 = (big\_mod(base,power/2,mod))%mod;

        return (p1\*p1)%mod;

    }

}

* Another code for this:

ll Pow(ll a,ll b) //larger power with mod;

{

ll ans=1; ///here a be the base and b be the power

while(b)

{

if(b&1)

{

b--;

ans=((ans%mod)\*a)%mod;

}

else

{

b/=2;

a=((a%mod)\*a)%mod;

}

}

return ans;

* Most efficient one:

ll big\_mod (ll a, ll b, ll c)

{

ll res = 1;

a=a%c;

while (b > 0)

{

if (b % 2 == 1)

{

res=mulmod(res,a,c);

}

a=mulmod(a,a,c);

b=b/2;

}

return res;

}

* I have used this Lightoj 1054 – Efficient Pseudo Code

|  |
| --- |
|  |

= Big Mod সংক্রান্ত প্রব্লেমঃ

* <http://uva.onlinejudge.org/external/3/374.html>
* http://uva.onlinejudge.org/external/116/11609.html

It’s Calculate: BP %M where P,B and M integer

**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;**

**}**

**UVa Problems:** 160, 294, 543, 583, 406, 686, 884,914, 10042, 10061, 10235, 10299, 10392, 10394, 10533, 10539, 10699, 10738, 10780, 10789, 10852, 11064, 11415, 11466

**LightOJ Problems:** 1045, 1014, 1028, 1035, 1045, 1054, 1067, 1090, 1098, 1109, 1213, 1214, 1245, 1282, 1336, 1340, 1341

**//Tutorial Link:**

<http://www.codechef.com/wiki/tutorial-number-theory>

<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=primeNumbers>

[http://comeoncodeon.wordpress.com](http://comeoncodeon.wordpress.com/)

<http://e-maxx.ru/algo/> (English Translation By Google Chrome)

[http://theoremoftheweek.wordpress.com](http://theoremoftheweek.wordpress.com/)

**Prepared By:**  
Forhad Ahmed (Email: [forhadsustbd@gmail.com](mailto:forhadsustbd@gmail.com) )  
Ashish Pal (Email: [**thimpu**.cse@gmail.com](mailto:thimpu.cse@gmail.com) )

Points to be noted here:

* There is no even prime except 2.
* According to goldbach’s conjecture a number can be expressed in terms of two primes.

Let,n is even and it may or may not be expressed in terms of two other primes a and b.Such that,n=a+b

But in case of odd number,it is gurranted that if that number is expressed in terms of of two primes,such as,n=a+b;

Then one of them must will be 2 and other be n-2.

### This theory is much nedded for problem Goldbach and Euler uva 10311.

SUM OF DIGITS:

Explanation link: https://www.dropbox.com/s/3s5uvec8cdto69h/Cybernauts\_Q-1\_A.pdf?dl=0

Code below:

#include <bits/stdc++.h>

#define ll long long

using namespace std;

ll sum(ll n)

{

ll pos=1,res=0,prev=0,cnt=0;

while(n)

{

ll r=n%10;

n/=10;

res+=(r\*(r-1)/2)\*pos +r\*(45\*cnt\*pos/10)+r\*(prev+1);

prev+=pos\*r;

cnt+=1;

pos\*=10;

}

return res;

}

int main()

{

ll a;

cin>>a;

sum(a);

}

* Primality test:non deterministic way(miller-rabin)

LINK:

1. <http://paste.ubuntu.com/17556987/>
2. <https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test>
3. <http://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/>

#include <bits/stdc++.h>

#define ll long long

using namespace std;

ll mulmod(ll a , ll b , ll mo)//for more uses this has to be used.

{

ll q = ((long double) a \* (long double) b / (long double) mo);

ll res = a \* b - mo \* q;

return ((res % mo) + mo) % mo;

}

ll big\_mod (ll a, ll b, ll c)//this is the best when a^b is calculated many times.

{

ll res = 1;

a=a%c;

while (b > 0)

{

if (b % 2 == 1)

{

res=mulmod(res,a,c);

}

a=mulmod(a,a,c);

b=b/2;

}

return res;

}

bool miller(ll a,ll d,ll p)

{

ll x=big\_mod(a,d,p);

if (x==1||x==p-1) return true;

while(d!=p-1)

{

x=mulmod(x,x,p);

d\*=2;

if(x==1) // x==1 means x=-1,x=1

//here p cant divide both ,then it becomes non prime;

{

return false;

}

if(x==p-1)

{

return true;

}

}

return false;

}

bool isprimes(ll p)

{

if(p<2)

{

return false;

}

if(p==2) return true;

if(p!=2&&p%2==0) return false;

ll d=p-1;

while(d%2==0) d=d/2;

for(ll i=1; i<20; i++)

{

ll a=2+rand()%(p-4);

if(!miller(a,d,p)) return false;

}

return true;

}

int main()

{

ll t,n,q,i,j,ans,y,x,f,k,tc;

//cin>>tc;

scanf("%lld",&tc);

while(tc--)

{

scanf("%lld",&n);

for(i=n-1; ; i--)

{

if(isprimes(i))

{

printf("%lld\n",i);

break;

}

}

}

}

ARTICULATION POINT:

vector<int>adj[MAXX];

int disc[MAXX];

int low[MAXX];

int parent[MAXX];

bool AP[MAXX];

bool vis[MAXX];

int tim;

void ini()

{

int i;

for(i =0; i<MAXX; i++)

{

vis[i]=AP[i]=false; // Initializing AP and vis array as false

parent[i]=-1; // Initializing parent of each vertex to -1

adj[i].clear(); // clearing the adjacency list.

low[i]=0;

}

tim=0; // initializing tim to 0

}

void dfs(int u)

{

vis[u]=true;

int i;

low[u]=disc[u]=(++tim);

int child=0;

for(i=0; i<adj[u].size(); i++)

{

int v=adj[u][i];

if(vis[v]==false)

{

child++;

parent[v]=u;

dfs(v);

low[u]=min(low[u],low[v]);

if( (parent[u]!=-1) and ( low[v]>=disc[u] ) )

AP[u]=true;

if( (parent[u]==-1) and (child>1))

AP[u]=true;

}

else if(v!=parent[u])

{

low[u]=min(low[u],disc[v]);

}

}

}

Ternary search:

*#include*<bits/stdc++.h>

*#define* ll long long

using namespace std;

ll func(ll x)

{

return 2\*x\*x - 12\*x +7;

}

ll l1,l2,ans,x;

ll ts(ll start, ll end)

{

ll l = start, r = end;

while(r-l>=3){

l1 = l + (r-l)/3;;

l2 = r - (r-l)/3;

if(func(l1) > func(l2))

{

l= l1;

}

else

{

r = l2;

}

}

if(l==r) return func(l);

else if(l+1==r) return min(func(l),func(r));

  else return min(func(l),min(func(l+1),func(l+2)));

}

int main()

{

int tc;

cin>>tc;

for(int tt=1; tt<=tc; tt++)

{

int x,y;

cin>>x>>y;

cout<<ts(x,y)<<endl;

}

}