SPOJ Problem Set (classical)

74. Divisor Summation

Problem code: DIVSUM

Given a natural number n (1 <= n <= 500000), please output the summation of all its proper divisors.

Definition: A proper divisor of a natural number is the divisor that is strictly less than the number.

e.g. number 20 has 5 proper divisors: 1, 2, 4, 5, 10, and the divisor summation is: 1 + 2 + 4 + 5 + 10 = 22.

Input

An integer stating the number of test cases (equal to about 200000), and that many lines follow, each containing one integer between 1 and 500000 inclusive.

Output

One integer each line: the divisor summation of the integer given respectively.

Example

```
Sample Input:
3
2
10
20

Sample Output:
1
8
22
```

Basicamente, o problema resume-se em encontrar o somatório de todos os divisores de um número n. O algoritmo trivial seria esse:

```
int soma(int n){

	int i;

	int sum=0;



	for(i=1;i<n;i++){

		if(n%i==0){

			sum +=i;

		}

	}

	return sum;

}
```

O algoritmo trivial excede o limite de tempo estabelecido para o problema então devemos pensar em uma maneira de diminuir a complexidade do problema. Primeiramente, vamos considerar uma versão mais simples do problema: contar o número de divisores de um número para depois generalizar para resolver este problema. Um resultado importante sobre a natureza dos divisores de um número é esse:

Teorema 1: Seja d o menor divisor de um número não-primo n então d ≤ $\sqrt{n}$.

Suponha por absurdo que d > $\sqrt{n}$. Logo,

$d^2 > n$

$d > n/d$

Se d é um divisor de n então existe um k tal que n=k*d, ou seja, k=n/d. Logo, n/d também é divisor de n. Absurdo, d é o menor divisor de n.

Podemos dividir o conjunto dos números divisores de um número em três $D_{<\sqrt{n}}$, $D_{>\sqrt{n}}$ e $D_{=\sqrt{n}}$ nos divisores menores que $\sqrt{n}$, maiores que $\sqrt{n}$ e divisores iguais $\sqrt{n}$, respectivamente. Para cada divisor no conjunto $D_{<\sqrt{n}}$ podemos encontrar um divisor no conjunto $D_{>\sqrt{n}}$··. O conjunto $D_{=\sqrt{n}}$ é igual Ø se o número não é um quadrado perfeito. Baseado nessa divisão, nós podemos desenvolver o seguinte algoritmo:

```
int soma(int n){
    int i;
    int sum = 1;
    if(n==1 ) return 0;
    for(i=2;i*i<n;i++){
       if(n%i==0) {
        sum+=i; sum+=n/i;
       }
    }
    if(i*i==n) sum += i;
    return sum;
}
```

Para evitar o excesso de multiplicação dentro for podemos retirar a complexidade calculando o valor limite antes da seguinte maneira.

```
int soma3(int n){
    int i;
    int sum = 1;
    double limite = sqrt(n);
    if(n==1 ) return 0;
    for(i=2;i<limite;i++){
       if(n%i==0) {
        sum+=i; sum+=n/i;
       }
    }
    if(i*i==n)sum += i;
    return sum;
}
```

Podemos ainda desenvolver uma solução diferente para o problema considerando uma adaptação do algoritmo do crivo de Erastóstenes. O *Crivo de Eratóstenes* é um método que permite obter uma tabela de números primos até um limite escolhido da seguinte maneira.

Escreve-se a sucessão natural dos números inteiros até ao número desejado. Suprime-se o número 1. O número 2 é o menor número primo. A partir do 3, cortamos todos os múltiplos de 2. O número 3, o primeiro que não foi cortado, é primo. A partir dos que lhe seguem cortamos todos os múltiplos de três. O primeiro não riscado é 5, que será número primo, e a partir de 6 cortamos todos os múltiplos de cinco.

É fácil ver que (será que é mesmo) o corte ou crivagem dos diferentes números pode começar a fazer-se, não a partir do número que se segue a um dado primo, mas a partir do quadrado desse número primo, pois verifica-se facilmente que são primos, todos os números não riscados até ao quadrado do novo número primo, a partir do qual se devia continuar a operação. Assim, depois da supressão dos múltiplos de 2, os números não riscados 3, 5 e 7 são primos por serem inferiores a 3^2 =9.

```
int eratosthenes(int vals[DIM])
{
  int i,j;

  for(i=0;i<DIM;i++)
    vals[i]=1;

  vals[0]=0;
  vals[1]=0;
  for(i=2;i<DIM;i++)
    if(vals[i])
      for(j=i*i;j<DIM;j+=i)
        vals[j]=0;

  return 0;
}
```

A idéia básica do algoritmo que vamos desenvolver é a seguinte: vamos pré-calcular o número de divisores para todos os números da entrada em uma tabela. Inicialmente, a tabela começa com todos os valores zerados. Para cada número i, acrescentamos i para todos múltiplos de i a partir do número 2i.

```
#include <stdio.h>

#include <string.h>

#define MAX 500001

int sum[MAX];

int n,i,j,t;

int main(){

      memset(sum,0,sizeof(sum)); //zerar o vetor sum
```

```
	for(i=1;i<MAX;i++){
	  for(j=i+i;j<MAX;j=j+i){
			sum[j] +=i;
		}
	}
	scanf("%d",&t);
	while(t--){
		scanf("%d",&n);
		printf("%d\n",sum[n]);
	}
	return 0;
}
```

Será que podemos combinar as duas soluções?(Eu acho que sim!!)

Vamos considerar novamente a versão simplificada do problema: contar o número de divisores de um número. Podemos desenvolver uma outra solução utilizando o **Teorema Fundamental da Aritmética** sustenta que todos os números inteiros positivos maiores que 1 podem ser decompostos num produto de números primos, sendo esta decomposição única a menos de permutações dos fatores.

$$N = {p_1}^{\alpha_1} \ldots . {p_n}^{\alpha_n}$$

O número de divisores de N pode ser dado $d(N) = (\alpha_1 + 1)x \ldots (\alpha_n + 1)$. Para desenvolvermos uma solução para o problema original temos que entender a origem dessa fórmula. Cada divisor do número N pode ser escrito como ${p_1}^{\alpha_1'} \ldots . {p_n}^{\alpha_n'}$, onde $0 \leq \alpha_i \leq \alpha_i$ dessa maneira temos $\alpha_i + 1$ possibilidades para cada fator da multiplicação. Utilizando o **princípio fundamental da contagem** que é um princípio combinatório que indica de quantas formas se pode escolher um elemento de cada um de *n* conjuntos finitos. Se o primeiro conjunto tem k_1 elementos, o segundo tem k_2 elementos, e assim sucessivamente, então o número total T de escolhas é dado por:

$T = k_1 . k_2 . k_3 . \ldots k_n$

Obtermos a seguinte fórmula $d(N) = (\alpha_1 + 1)x \ldots (\alpha_n + 1)$.

Para encontrar a solução basta encontrar

$\sum {p_1}^{\alpha_1'} \ldots . {p_n}^{\alpha_n'}$, onde $0 \leq \alpha_i \leq \alpha_i$