ASSIGNMENT- 1

SUBMIT IT WITHIN 20 DAYS

I HAVE A SOFTWARE TO TEST HOW MUCH PORTION OF THE CODE ARE COPIED FROM YOUR CLASSMATES' CODE OR INTERNET.SO PLEASE DON'T TRY TO COPY.OTHERWISE YOUR ASSIGNMENT WILL BE CANCELLED AND DURING THE SEMESTER I FIXED YOU AS A COPYMAN/COPYWOMEN AND DEAL ACCORDINGLY.YOU CAN TAKE HELP FROM YOUR CLASSMATES, INTERNET OR SENIORS BUT WRITE YOUR CODE IN YOUR OWN WAY.IF YOU DON'T ABLE TO WRITE CODE IN C WRITE THE LOGIC IN ENGLISH LIKE ALGORITHM.BUT **DON'T COPY.**

NOTE FOR YOUR PROBLEM:

- *Fibonacci numbers* — Defined by the recurrence $F_n = F_{n-1} + F_{n-2}$ and the initial values $F_0 = 0$ and $F_1 = 1$, they emerge repeatedly because this is perhaps the simplest interesting recurrence relation. The first several values are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...The Fibonacci numbers lend themselves to an amazing variety of mathematical identities, and are just fun to play with. They have the following hard-to-guess but simple-to-derive closed form:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

  This closed form has certain important implications. Since $(1-\sqrt{5})/2$ is between 0 and 1, raising it to any power leaves a number in this range. Thus the first term, ϕ^n where $\phi = (1+\sqrt{5})/2$ is the driving quantity, and can be used to estimate F_n to within plus or minus 1.

ASSIGNMENT 1:

Recall the definition of the Fibonacci numbers:

$$\begin{aligned} f_1 &:= 1 \\ f_2 &:= 2 \\ f_n &:= f_{n-1} + f_{n-2} \qquad (n \geq 3) \end{aligned}$$

Given two numbers a and b, calculate how many Fibonacci numbers are in the range $[a, b]$.

Input

The input contains several test cases. Each test case consists of two non-negative integer numbers a and b. Input is terminated by $a = b = 0$. Otherwise, $a \leq b \leq 10^{100}$. The numbers a and b are given with no superfluous leading zeros.

Output

For each test case output on a single line the number of Fibonacci numbers f_i with $a \leq f_i \leq b$.

Sample Input

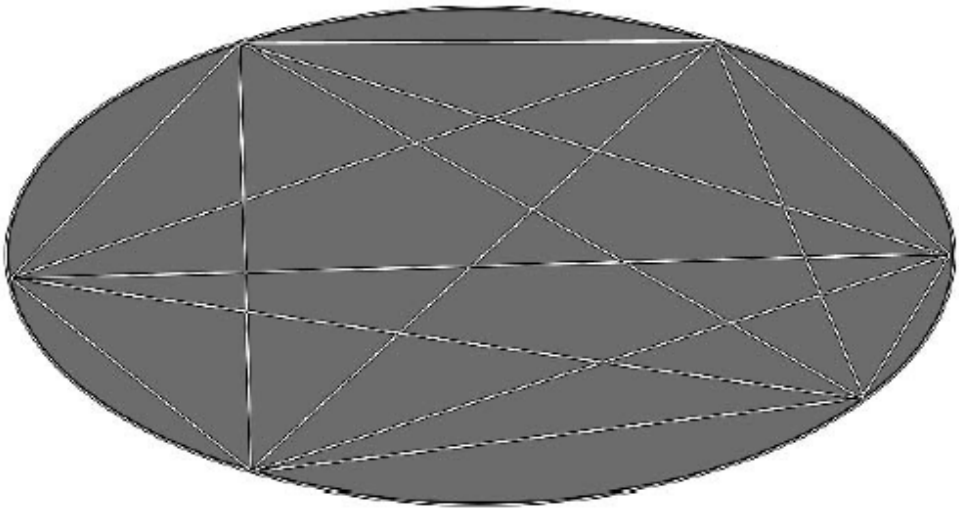
```
10 100
1234567890 9876543210
0 0
```

Sample Output

```
5
4
```

ASSIGNMENT 2:

You are given an elliptical-shaped land and you are asked to choose n arbitrary points on its boundary. Then you connect each point with every other point using straight lines, forming $n(n-1)/2$ connections. What is the maximum number of pieces of land you will get by choosing the points on the boundary carefully?



Dividing the land when $n = 6$.

Input

The first line of the input file contains one integer s $(0 < s < 3,500)$, which indicates how many input instances there are. The next s lines describe s input instances, each consisting of exactly one integer n $(0 \leq n < 2^{31})$.

Output

For each input instance output the maximum possible number pieces of land defined by n points, each printed on its own line.

Sample Input	*Sample Output*
4	1
1	2
2	4
3	8
4	

ASSIGNMENT 3:

Gustavo knows how to count, but he is just now learning how to write numbers. He has already learned the digits 1, 2, 3, and 4. But he does not yet realize that 4 is different than 1, so he thinks that 4 is just another way to write 1.

He is having fun with a little game he created: he makes numbers with the four digits that he knows and sums their values. For example:

`132 = 1 + 3 + 2 = 6`
`112314 = 1 + 1 + 2 + 3 + 1 + 1 = 9` (remember that Gustavo thinks that `4 = 1`)

Gustavo now wants to know how many such numbers he can create whose sum is a number n. For $n = 2$, he can make 5 numbers: 11, 14, 41, 44, and 2. (He knows how to count up beyond five, just not how to write it.) However, he can't figure out this sum for n greater than 2, and asks for your help.

Input

Input will consist of an arbitrary number of integers n such that $1 \leq n \leq 1,000$. You must read until you reach the end of file.

Output

For each integer read, output an single integer on a line stating how many numbers Gustavo can make such that the sum of their digits is equal to n.

Sample Input

2
3

Sample Output

5
13

NOTE FOR YOUR ASSIGNMENT:

FINDING PRIMES:

The easiest way to test if a given number x is prime uses repeated division. Start from the smallest candidate divisor, and then try all possible divisors up from there. Since 2 is the only even prime, once we verify that x isn't even we only need try the odd numbers as candidate factors. Further, we can bless n as prime the instant we have shown that it has no non-trivial prime factors below $\sqrt{n}$. Why? Suppose not – i.e., x is composite but has a smallest non-trivial prime factor p which is greater than $\sqrt{n}$. Then x/p must also divide x, and must be larger than p, or else we would have seen it earlier. But the product of two numbers greater than $\sqrt{n}$ must be larger than n, a contradiction.

Computing the prime factorization involves not only finding the first prime factor, but stripping off all occurrences of this factor and recurring on the remaining product:

```
prime_factorization(long x)
{
        long i;                 /* counter */
        long c;                 /* remaining product to factor */

        c = x;
        while ((c % 2) == 0) {
                printf("%ld\n",2);
                c = c / 2;
        }

        i = 3;
        while (i <= (sqrt(c)+1)) {
                if ((c % i) == 0) {
                        printf("%ld\n",i);
                        c = c / i;
                }
                else
                        i = i + 2;
        }

        if (c > 1) printf("%ld\n",c);
}
```

Testing the terminating condition $i > \sqrt{c}$ is somewhat problematic, because `sqrt()` is a numerical function with imperfect precision. Just to be safe, we let i run an extra iteration. Another approach would be to avoid floating point computation altogether and terminate when `i*i > c`. However, the multiplication might cause overflow when working on very large integers. Multiplication can be avoided by observing that $(i+1)^2 = i^2 + 2i + 1$, so adding $i + i + 1$ to i^2 yields $(i + 1)^2$.

For higher performance, we could move the `sqrt(c)` computation outside the main loop and only update it when `c` changes value. However, this program responds instantly on my computer for the prime 2,147,483,647. There exist fascinating randomized algorithms which are more efficient for testing primality on very large integers, but these are not something for us to worry about at this scale – except as the source of interesting contest problems themselves.

[1]Now a little puzzle to test your understanding of the proof. Suppose we take the first n primes, multiply them together, and add one. Does this number have to be prime? Give a proof or a counterexample.

DIVISIBILITY

Number theory is the study of integer divisibility. We say b *divides* a (denoted $b|a$) if $a = bk$ for some integer k. Equivalently, we say that b is a *divisor of* a or a is a *multiple* of b if $b|a$.

As a consequence of this definition, the smallest natural divisor of every non-zero integer is 1. Why? It should be clear that there is in general no integer k such that $a = 0 \cdot k$.

How do we find all divisors of a given integer? From the prime number theorem, we know that x is uniquely represented by the product of its prime factors. Every divisor is the product of some subset of these prime factors. Such subsets can be constructed using backtracking techniques as discussed in Chapter 8, but we must be careful about duplicate prime factors. For example, the prime factorization of 12 has three terms (2, 2, and 3) but 12 has only 6 divisors (1, 2, 3, 4, 6, 12).

GREATEST COMMON DIVISOR

Since 1 divides every integer, the least common divisor of every pair of integers a, b is 1. Far more interesting is the *greatest common divisor*, or *gcd*, the *largest* divisor shared by a given pair of integers. Consider a fraction x/y, say, 24/36. The reduced form of this fraction comes after we divide both the numerator and denominator by $gcd(x, y)$, in this case 12. We say two integers are *relatively prime* if their greatest common divisor is 1.

Euclid's algorithm for finding the greatest common divisor of two integers has been called history's first interesting algorithm. The naive way to compute gcd would be to test all divisors of the first integer explicitly on the second, or perhaps to find the prime factorization of both integers and take the product of all factors in common. But both approaches involve computationally intensive operations.

Euclid's algorithm rests on two observations. First,

If $b|a$, then $gcd(a, b) = b$.

This should be pretty clear. If b divides a, then $a = bk$ for some integer k, and thus $gcd(bk, b) = b$. Second,

If $a = bt + r$ for integers t and r, then $gcd(a, b) = gcd(b, r)$.

Why? By definition, $gcd(a,b) = gcd(bt+r,b)$. Any common divisor of a and b must rest totally with r, because bt clearly must be divisible by any divisor of b.

Euclid's algorithm is recursive, repeated replacing the bigger integer by its remainder mod the smaller integer. This typically cuts one of the arguments down by about half, and so after a logarithmic number of iterations gets down to the base case. Consider the following example. Let $a = 34398$ and $b = 2132$.

$$\begin{aligned} gcd(34398, 2132) &= gcd(34398 \bmod 2132, 2132) = gcd(2132, 286) \\ gcd(2132, 286) &= gcd(2132 \bmod 286, 286) = gcd(286, 130) \\ gcd(286, 130) &= gcd(286 \bmod 130, 130) = gcd(130, 26) \\ gcd(130, 26) &= gcd(130 \bmod 26, 26) = gcd(26, 0) \end{aligned}$$

Therefore, $gcd(34398, 2132) = 26$.

However, Euclid's algorithm can give us more than just the $gcd(a,b)$. It can also find integers x and y such that

$$a \cdot x + b \cdot y = gcd(a,b)$$

which will prove quite useful in solving linear congruences. We know that $gcd(a,b) = gcd(b,a')$, where $a' = a - b\lfloor a/b \rfloor$. Further, assume we know integers x' and y' such that

$$b \cdot x' + a' \cdot y' = gcd(a,b)$$

by recursion. Substituting our formula for a' into the above expression gives us

$$b \cdot x' + (a - b\lfloor a/b \rfloor) \cdot y' = gcd(a,b)$$

and rearranging the terms will give us our desired x and y. We need a basis case to complete our algorithm, but that is easy since $a \cdot 1 + 0 \cdot 0 = gcd(a,0)$.

For the previous example, we get that $34398 \times 15 + 2132 \times -242 = 26$. An implementation of this algorithm follows below:

```
/*      Find the gcd(p,q) and x,y such that p*x + q*y = gcd(p,q)      */

long gcd(long p, long q, long *x, long *y)
{
        long x1,y1;                     /* previous coefficients */
        long g;                         /* value of gcd(p,q) */

        if (q > p) return(gcd(q,p,y,x));

        if (q == 0) {
                *x = 1;
                *y = 0;
                return(p);
        }

        g = gcd(q, p%q, &x1, &y1);

        *x = y1;
        *y = (x1 - floor(p/q)*y1);

        return(g);
}
```

MODULAR ARITHMETIC

The number we are dividing by is called the *modulus*, and the remainder left over is called the *residue*. The key to efficient modular arithmetic is understanding how the basic operations of addition, subtraction, and multiplication work over a given modulus:

- *Addition* — What is $(x + y) \bmod n$? We can simplify this to

$$((x \bmod n) + (y \bmod n)) \bmod n$$

to avoid adding big numbers. How much small change will I have if given \$123.45 by my mother and \$94.67 by my father?

$$(12,345 \bmod 100) + (9,467 \bmod 100) = (45 + 67) \bmod 100 = 12 \bmod 100$$

- *Subtraction* — Subtraction is just addition with negative values. How much small change will I have after spending \$52.53?

$$(12 \bmod 100) - (53 \bmod 100) = -41 \bmod 100 = 59 \bmod 100$$

Notice how we can convert a negative number mod n to a positive number by adding a multiple of n to it. Further, this answer makes sense in this change example. It is usually best to keep the residue between 0 and $n - 1$ to ensure we are working with the smallest-magnitude numbers possible.

- *Multiplication* — Since multiplication is just repeated addition,

$$xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$$

How much change will you have if you earn \$17.28 per hour for 2,143 hours?

$$(1,728 \times 2,143) \bmod 100 = (28 \bmod 100) \times (43 \bmod 100) = 4 \bmod 100$$

Further, since exponentiation is just repeated multiplication,

$$x^y \bmod n = (x \bmod n)^y \bmod n$$

Since exponentiation is the quickest way to produce really large integers, this is where modular arithmetic really proves its worth.

- *Division* — Division proves considerably more complicated to deal with, and will be discussed in Section 7.4.

Modular arithmetic has many interesting applications, including:

- *Finding the Last Digit* — What is the last digit of 2^{100}? Sure we can use infinite precision arithmetic and look at the last digit, but why? We can do this computation by hand. What we really want to know is what $2^{100} \bmod 10$ is. By doing repeated squaring, and taking the remainder mod 10 at each step we make progress very quickly:

$$
\begin{aligned}
2^3 \bmod 10 &= 8 \\
2^6 \bmod 10 &= 8 \times 8 \bmod 10 \to 4 \\
2^{12} \bmod 10 &= 4 \times 4 \bmod 10 \to 6 \\
2^{24} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{48} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{96} \bmod 10 &= 6 \times 6 \bmod 10 \to 6 \\
2^{100} \bmod 10 &= 2^{96} \times 2^3 \times 2^1 \bmod 10 \to 6
\end{aligned}
$$

- *RSA Encryption Algorithm* — A classic application of modular arithmetic on large integers arises in public-key cryptography, namely, the RSA algorithm. Here, our message is encrypted by coding it as an integer m, raising it to a power k, where k is the so-called public-key or encryption key, and taking the results mod n. Since m, n, and k are all huge integers, computing m^k mod n efficiently requires the tools we developed above.

- *Calendrical Calculations* — As demonstrated with the birthday example, computing the day of the week a certain number of days from today, or the time a certain number of seconds from now, are both applications of modular arithmetic.

CONGRUENCES:

Congruences are an alternate notation for representing modular arithmetic. We say that $a \equiv b (\text{mod } m)$ if $m|(a-b)$. By definition, if a mod m is b, then $a \equiv b (\text{mod } m)$.

Congruences are an alternate notation for modular arithmetic, not an inherently different idea. Yet the notation is important. It gets us thinking about the *set* of integers with a given remainder n, and gives us equations for representing them. Suppose that x is a variable. What integers x satisfy the congruence $x \equiv 3 (\text{mod } 9)$?

For such a simple congruence, the answer is easy. Clearly $x = 3$ must be a solution. Further, adding or deleting the modulus (9 in this instance) gives another solution. The set of solutions is all integers of the form $9y + 3$, where y is any integer.

What about complicated congruences, such as $2x \equiv 3 (\text{mod } 9)$ and $2x \equiv 3 (\text{mod } 4)$? Trial and error should convince you that exactly the integers of the form $9y + 6$ satisfy the first example, while the second has no solutions at all.

There are two important problems on congruences, namely, performing arithmetic operations on them, and solving them. These are discussed in the sections below.

Operations on Congruences

Congruences support addition, subtraction, and multiplication, as well as a limited form of division – provided they share the same modulus:

- *Addition and Subtraction* — Suppose $a \equiv b (\text{mod } n)$ and $c \equiv d (\text{mod } n)$. Then $a + c \equiv b + d (\text{mod } n)$. For example, suppose I know that $4x \equiv 7 (\text{mod } 9)$ and $3x \equiv 3 (\text{mod } 9)$. Then

$$4x - 3x \equiv 7 - 3 (\text{mod } 9) \to x \equiv 4 (\text{mod } 9)$$

- *Multiplication* — It is apparent that $a \equiv b (\text{mod } n)$ implies that $a \cdot d \equiv b \cdot d (\text{mod } n)$ by adding the reduced congruence to itself d times. In fact, general multiplication also holds, i.e., $a \equiv b (\text{mod } n)$ and $c \equiv d (\text{mod } n)$ implies $ac \equiv bd (\text{mod } n)$.

- *Division* — However, we cannot cavalierly cancel common factors from congruences. Note that $6 \cdot 2 \equiv 6 \cdot 1 (\text{mod } 3)$, but clearly $2 \not\equiv 1 (\text{mod } 3)$. To see what the problem is, note that we can redefine division as multiplication by an inverse, so a/b is equivalent to ab^{-1}. Thus we can compute $a/b (\text{mod } n)$ if we can find the inverse b^{-1} such that $bb^{-1} \equiv 1 (\text{mod } n)$. This inverse does not always exist – try to find a solution to $2x \equiv 1 (\text{mod } 4)$.

  We *can* simplify a congruence $ad \equiv bd (\text{mod } dn)$ to $a \equiv b (\text{mod } n)$, so we can divide all three terms by a mutually common factor if one exists. Thus $170 \equiv 30 (\text{mod } 140)$ implies that $17 \equiv 3 (\text{mod } 14)$. However, the congruence $a \equiv b (\text{mod } n)$ must be false (i.e., has no solution) if $gcd(a, n)$ does not divide b.

ASSIGNMENT 4:

There is man named Mabu who switches on-off the lights along a corridor at our university. Every bulb has its own toggle switch that changes the state of the light. If the light is off, pressing the switch turns it on. Pressing it again will turn it off. Initially each bulb is off.

He does a peculiar thing. If there are n bulbs in a corridor, he walks along the corridor back and forth n times. On the ith walk, he toggles only the switches whose position is divisible by i. He does not press any switch when coming back to his initial position. The ith walk is defined as going down the corridor (doing his peculiar thing) and coming back again. Determine the final state of the last bulb. Is it on or off?

Input

The input will be an integer indicating the nth bulb in a corridor, which is less than or equal to $2^{32} - 1$. A zero indicates the end of input and should not be processed.

Output

Output "`yes`" or "`no`" to indicate if the light is on, with each case appearing on its own line.

Sample Input

```
3
6241
8191
0
```

Sample Output

```
no
yes
no
```

ASSIGNMENT 5:

Certain cryptographic algorithms make use of big prime numbers. However, checking whether a big number is prime is not so easy.

Randomized primality tests exist that offer a high degree of confidence of accurate determination at low cost, such as the Fermat test. Let a be a random number between 2 and $n-1$, where n is the number whose primality we are testing. Then, n is *probably* prime if the following equation holds:

$$a^n \bmod n = a$$

If a number passes the Fermat test several times, then it is prime with a high probability.

Unfortunately, there is bad news. Certain composite numbers (non-primes) still pass the Fermat test with every number smaller than themselves. These numbers are called Carmichael numbers.

Write a program to test whether a given integer is a Carmichael number.

Input

The input will consist of a series of lines, each containing a small positive number n ($2 < n < 65,000$). A number $n = 0$ will mark the end of the input, and must not be processed.

Output

For each number in the input, print whether it is a Carmichael number or not as shown in the sample output.

Sample Input	*Sample Output*
1729	The number 1729 is a Carmichael number.
17	17 is normal.
561	The number 561 is a Carmichael number.
1109	1109 is normal.
431	431 is normal.
0	