Let's talk about *binary numbers*. We have an n-digit binary number, b, and we denote the digit at index i (zero-indexed from right to left) to be b_i. We can find the *decimal* value of b using the following formula:

$$(b)_2 \Rightarrow b_{n-1} \cdot 2^{n-1} + \ldots + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0 = (?)_{10}$$

For example, if binary number $b = 10010$, we compute its decimal value like so:

$$(10010)_2 \Rightarrow 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = (18)_{10}$$

Meanwhile, in our well-known decimal number system where each digit ranges from 0 to 9, the value of some decimal number, d, can be expanded in the same way:

$$d = d_{n-1} \cdot 10^{n-1} + \ldots + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

Now that we've discussed both systems, let's combine decimal and binary numbers in a new system we call *decibinary*! In this number system, each digit ranges from 0 to 9 (like the decimal number system), but the *place value* of each digit corresponds to the one in the binary number system. For example, the decibinary number 2016 represents the decimal number 24 because:

$$(2016)_{decibinary} \Rightarrow 2 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 6 \cdot 2^0 = (24)_{10}$$

Pretty cool system, right? Unfortunately, there's a problem: two different decibinary numbers can evaluate to the same decimal value! For example, the decibinary number 2008 also evaluates to the decimal value 24:

$$(2008)_{decibinary} \Rightarrow 2 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 8 \cdot 2^0 = (24)_{10}$$

This is a major problem because our new number system has no real applications beyond this challenge!

Consider an infinite list of non-negative decibinary numbers that is sorted according to the following rules:

- The decibinary numbers are sorted in increasing order of the decimal value that they evaluate to.
- Any two decibinary numbers that evaluate to the same decimal value are ordered by increasing decimal value, meaning the equivalent decibinary values are strictly interpreted and compared as decimal values and the smaller decimal value is ordered first. For example, $(2)_{decibinary}$ and $(10)_{decibinary}$ both evaluate to $(2)_{10}$. We would order $(2)_{decibinary}$ before $(10)_{decibinary}$ because $(2)_{10} < (10)_{10}$.

Here is a list of first few decibinary numbers properly ordered:

x	Decibinary	Decimal
1	0	0
2	1	1
3	2	2
4	10	2
5	3	3
6	11	3
7	4	4
8	12	4
9	20	4
10	100	4
.	...	...
20	110	6

You will be given q queries in the form of an integer, x. For each x, find and print the the x^{th} decibinary number in the list on a new line.

Function Description

Complete the *decibinaryNumbers* function in the editor below. For each query, it should return the decibinary number at that one-based index.

decibinaryNumbers has the following parameter(s):

- *x*: the index of the decibinary number to return

Input Format

The first line contains an integer, q, the number of queries.
Each of the next q lines contains an integer, x, describing a query.

Constraints

- $1 \leq q \leq 10^5$
- $1 \leq x \leq 10^{16}$

Subtasks

- $1 \leq x \leq 50$ for 10% of the maximum score
- $1 \leq x \leq 9000$ for 30% of the maximum score
- $1 \leq x \leq 10^7$ for 60% of the maximum score

Output Format

For each query, print a single integer denoting the the x^{th} decibinary number in the list. Note that this must be the actual decibinary number and *not* its decimal value. Use 1-based indexing.

Sample Input 0

```
5
1
2
3
4
10
```

Sample Output 0

```
0
1
2
10
100
```

Explanation 0

For each x, we print the x^{th} decibinary number on a new line. See the figure in the problem statement.

Sample Input 1

```
7
8
23
19
16
26
7
6
```

Sample Output 1

```
12
23
102
14
111
4
11
```

Sample Input 2

```
10
19
25
6
8
20
10
27
24
30
11
```

Sample Output 2

```
102
103
11
12
110
100
8
31
32
5
```