AWESOME Number

We say a number is AWESOME if it is prime and has 1 in its ones place. You are given a number N and you are asked to answer how many AWESOME numbers are not greater than N.

Input
The input consists of a single integer N (1 <= N <= 20,000,000).

Output
You should print **one line** containing a single integer, the number of awesome numbers that are no larger than N.

Example 1

```
Input:
6

Output:
0
```

Example 2

```
Input:
11

Output:
1
```

Example 3

```
Input:
100

Output:
5
```

Prime Gap

A **prime gap** is the difference between two successive prime numbers. The *n*-th prime gap, denoted g(n) is the difference between the (n+1)-th and the n-th prime numbers, i.e.

g(n) = p(n+1) - p(n)

The first 7 prime numbers are 2, 3, 5, 7, 11, 13, 17, and the first 6 prime gaps are 1, 2, 2, 4, 2, 4.

Shinya Yukimura is interested in prime gaps and he need some experimental data to verify his hypothesis. More specifically, given a closed interval [a,b], Shinya wants to find the two adjacent primes p1 and p2 (a <= p1 < p2 <= b) such that the prime gap between p1 and p2 is minimized (i.e. p2-p1 is the minimum). If there are multiple prime pairs that have the same prime gap, report the first pair. Shinya also wants to find the two adjacent primes p3 and p4 (a <= p3 < p4 <= b) that maximize the gap between p3 and p4 (choose the first pair if there are mote than one such pairs).

Please write a program to help Shinya.

Input

Two integer values a,b, with a < b. The difference between a and b will not exceed 1,000,000. 1 <= a <= b <= 2,147,483,647.

Output

If there are no adjacent primes in the interval [a,b], output -1 followed by a newline.

Otherwise the output should be 4 integers: p1,p2,p3,p4 as mentioned above separated by a space.

Example 1

```
Input:
1 20

Output:
2 3 7 11
```

Example 2

```
Input:
13 16

Output:
-1
```

In the first example test case, the prime gap between 13 and 17 also has the largest value 4, but the pair (7,11) appears before (13,17), so we output 7 11 instead of 13 17.

Reverse Polish notation

Reverse Polish notation (RPN), also known as Polish postfix notation or simply postfix notation, is a mathematical notation in which operators follow their operands, in contrast to Polish notation (PN), in which operators precede their operands. It does not need any parentheses as long as each operator has a fixed number of operands.

You need to write a program that transforms an infix expression to a equivalent RPN according to the following specifications.

1. The infix expression is in the input file in the format of one character per line, with a maximum of 50 lines. For example, `(1+1)*(4*5+1)-4` would be in the form:

```
(
1
+
1
)
*
(
4
*
5
+
1
)
-
4
```

2. There will be only one infix expression in the input file, and it will be an expression with a valid syntax.
3. All operators are binary operators `+, -, *, /` .
4. The operands will be one digit numerals: 0, 1, 2, … , 9.
5. The operators `*` and `/` have the highest precedence. The operators `+` and `-` have the lowest precedence. Operators at the same precedence level associate from left to right. Parentheses act as grouping symbols that override the operator precedence.

Input

There will be multiple lines in the input file as specified above.

Output

The output file will have a postfix expression all on one **line** with no whitespace between symbols and a single newline character at the end.

Example

```
Input:
(
1
+
1
)
*
(
4
*
5
+
1
```

```
)
-
4

Output:
11+45*1+*4-
```

Stack Puzzle

There are two sequences of stack operations converting the word TROT to TORT:

```
[
i i i i o o o o
i o i i o o i o
]
```

where `i` and `o` stands for push and pop operation respectively. In this problem, you are given two words and you are asked to find out all sequences of stack operations converting the first word to the second.

Input
The input consists of two lines, the first of which is the source word and the second is the target word.

Output
Your program should print a sorted list of valid `i/o` sequences. The list is delimited by

```
[
]
```

and the sequences are sorted in lexicographical order. Within each sequence, `i`'s' and `o`'s are separated by a single space and each sequence is terminated by a new line.

Process
Given an input word, a valid `i/o` sequence implies that every character of the word is pushed and popped exactly once, and no attempt is ever made to pop an empty stack. For example, if the word FOO is input, then the sequence:

- `i i o i o o` is valid and produces OFF
- `i i o` is not valid (too short),
- `i i o o o i` is not valid (illegal pop from an empty stack)

A valid sequence infers a permutation of the letters in the input word. For example, given the input word FOO, both sequences `i i o i o o` and `i i i o o o` give the word OOF.

Example 1

```
Input:
madam
adamm

Output:
[
i i i i o o o i o o
i i i i o o o o i o
i i o i o i o i o o
i i o i o i o o i o
]
```

Example 2

```
Input:
bahama
bahama
```

```
Output:
[
i o i i i o o i i o o o
i o i i i o o o i o i o
i o i o i o i i i o o o
i o i o i o i o i o i o
]
```

Example 3

```
Input:
long
short

Output:
[
]
```

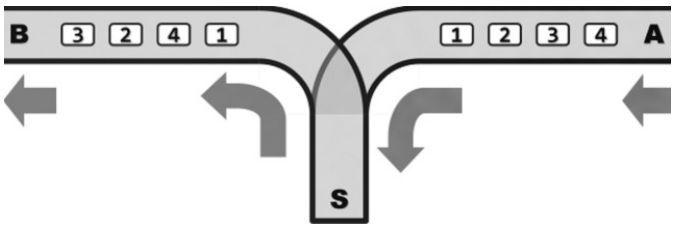
Example 4

```
Input:
eric
rice

Output:
[
i i o i o i o o
]
```

Train

This is a figure that shows the structure of a station for train dispatching.



In this station, A is the entrance for each train and B is the exit. S is the switching track. The coaches of a train can enter the switching track from direction A and must leave in direction B. Individual coaches can be disconnected from the rest of the train as they enter the switching track, so that they can be reorganized before they continue in direction B. If a coach enters the switching track from direction A, it must leave in direction B (i.e., it cannot return towards A). If a coach leaves in direction B, it cannot return back to the switching track.

Assume that a train consist of n coaches labeled {1, 2, …, n}. A dispatcher wants to know whether these coaches can pull out at B in the order of {a1, a2, …, an}.

Input

The 1st line contains an integer n (n <= 1,000) equal to the number of coaches, as described above. In each of the next lines of the input, except the last one, there is a permutation of 1, 2, . . . , n, this is the sequence {a1, a2, …, an} that the dispatcher would like to achieve as the coaches leave the switching track in direction B. The last line of the block contains just ‘0’ (to indicate the end of input).

Output

You should output the result for each permutation. If the sequence is feasible, output a "Yes", followed by a newline. If the sequence is infeasible, output a "No", followed by a newline.

Example 1 (pictured in the figure)

```
Input:
4
3 2 4 1
0

Output:
Yes
```

Example 2

```
Input:
5
1 2 3 4 5
4 5 1 3 2
0

Output:
Yes
No
```