A

We need to check if the first 9 or last 9 characters of s are equal to the given pattern.

You can use list slicing in Python, or `s.substr` in C++ in order to easily grab a contiguous substring of some string s.

```
s = input()
pattern = '...---...'

print("SOS" if pattern == s[:len(pattern)] or pattern == s[-len(pattern):] else "Hay SOS!")

```

B

We need to check if the given pattern appears as a *contiguous substring* anywhere in s. In Python, this is easily accomplished with the `in` operation. In C++, you can use `s.find`

```
s = input()
pattern = '...---...'

print("SOS" if pattern in s else "Hay SOS!")

```

But how does this work behind the scenes? What is its running time? For educational purposes, let's peek behind the curtain.

Let $|w|$ denote the length of some string w. For convenience, let $n := |s|$.

The string s has $|s| - |\text{pattern}| + 1$ contiguous substrings that are the same length as the given pattern (think of how many valid *starting indices* there are). Let's just check all of them and see if any are equal to our pattern.

```
s = input()
pattern = '...---...'

def is_substring(s, pattern):
    for i in range(len(s) - len(pattern) + 1):
        if s[i : i+len(pattern)] == pattern:
            return True
    return False


print("SOS" if is_substring(s, pattern) else "Hay SOS!")
```

Is string comparison free? Well, it's not a primitive operation. For educational purposes, let's unpack that as well. How does string comparison work?

- If the two strings are of different lengths then they are definitely not equal.
- Otherwise, the two strings are equal if and only if they match at each position.

You can imagine that if the strings are long, then checking that they match at *all* positions can be expensive, because we have to check *all* the indices. At worst, we end up checking almost all of the indices, which means the running time is $\mathcal{O}(|\text{pattern}|)$.

```
s = input()
pattern = '...---...'

def string_eq(a, b):
    if len(a) != len(b):
        return False
    for i in range(len(a)):
        if a[i] != b[i]:
            return False
    return True

def is_substring(s, pattern):
```

```
    for i in range(len(s) - len(pattern) + 1):
        if string_eq(s[i : i+len(pattern)], pattern):
            return True
    return False


print("SOS" if is_substring(s, pattern) else "Hay SOS!")
```

This is what's really happening under the hood when you use `in` or `find` between two strings. Thus, the running time of our solution is $\mathcal{O}(|\text{pattern}|(n - |\text{pattern}|))$[1]. Since the pattern of `...---...` in our problem is only length 9, then the running time of our solution is $\mathcal{O}(9(n-9))$, which we simplify to $\mathcal{O}(n)$, which is acceptable.

However, in general, it may be possible that $|\text{pattern}| \approx n/2$, which would cause our running time to degenerate to $\mathcal{O}\left(\frac{n}{2}(n - n/2)\right)$, which is $\mathcal{O}(n^2/4)$, which simplifies to $\mathcal{O}(n^2)$, which is too slow! In these cases, we would need to use a more advance technique like hash functions and rolling hashes. But that's overkill for this simple problem, so we won't discuss that any further here.

[1] We can drop the +1 inside because remember, Big O Notation is just a rough estimate for large input sizes

K

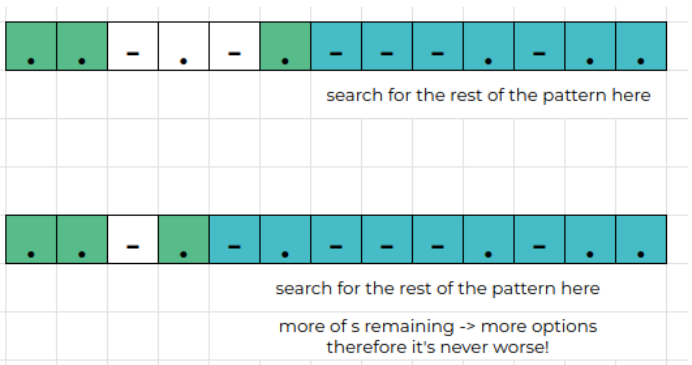
We want to check if the given pattern `...---...` appears as a not-necessarily-contiguous *subsequence* in s.

A solution for this special pattern

I claim that `...---...` appears as a subsequence in s if and only if there are exactly three dashes between the first three dots and the last three dots.

```
s = input()
n = len(s)

prefix_dots = 0
first_third_dot_at = None
for i in range(n):
    if s[i] == '.':
        prefix_dots += 1
        if prefix_dots == 3:
            first_third_dot_at = i
            break

suffix_dots = 0
last_third_dot_at = None
for i in range(n-1, -1, -1):
    if s[i] == '.':
        suffix_dots += 1
        if suffix_dots == 3:
            last_third_dot_at = i
            break

between_dashes = 0
if first_third_dot_at is not None:
    for i in range(first_third_dot_at+1, last_third_dot_at):
        if s[i] == '-':
            between_dashes += 1

print("SOS" if between_dashes >= 3 else "Hay SOS!")

```

After all, when doing a subsequence check, *why not* just always take the earliest three dots in s as the first three dots of the pattern? If I take earlier dots, then there's "more of s left over", which makes it more likely that the rest of the pattern is contained in s. Having "more of s" is never worse!



The same argument applies for always taking the last three dots.

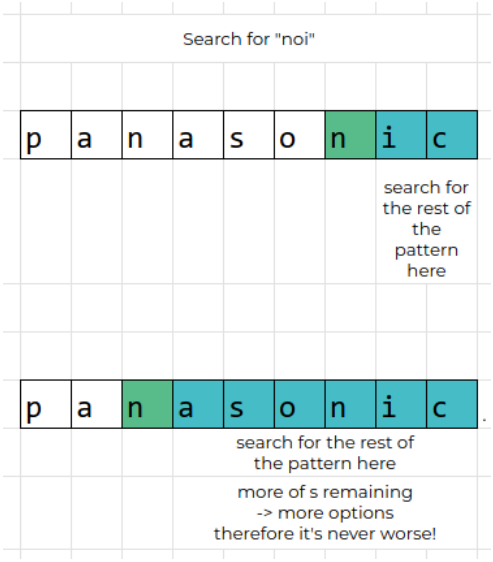
We only have to loop over the characters of s at most 3 times. Therefore, our solution runs in $\mathcal{O}(n)$.

A general solution for subsequence checks

This solution works for any s and any pattern, even with arbitrary letters and characters.

Remember when we said *why not* just always take the earliest three dots, because it's never worse? Think about it... doesn't that just always apply, in general?

Consider the first letter of the pattern. Find its earliest occurrence in s. *Why not* just always take this? It doesn't benefit us to wait, because *having more of s is always better.*



Then, repeat—for the next letter of the pattern, find its earliest occurrence in the remaining part of s... and so on. Either we exhaust all the letters of the pattern (in which case yes, the pattern is a subsequence of s) or we exhaust all the letter of s first (in which case no, the pattern is not a subsequence of s).

```
s = input()
pattern = '...---...'

j = 0
for i in range(len(s)):
    if s[i] == pattern[j]:
        j += 1
        if j == len(pattern):
            break

print("SOS" if j == len(pattern) else "Hay SOS!")
```

Since there is only one loop, it's clear that the running time is $\mathcal{O}(n + |\text{pattern}|)$.

D

First of all, we would like to begin by reminding you about the magic number: as a coarse rule of thumb, a Python program can run somewhere on the order of magnitude of $\approx 10^7$ operations per second, and a C++ can run an order of magnitude of $\approx 10^8$ operations per second. If we want our code to run under the time limit, then we need to make sure its operation count is roughly under this magic number.

We want to count how many integers satisfy both of these conditions:

- It lies between L and R (inclusive)
- It's an SOS number.

The most straightforward idea might look something like this: For each number from L to R, test each one for being an SOS number. See the following pseudocode.

```
total = 0
for each integer k from L to R:
    if is_SOS(k):
        total += 1
print(total)
```

There are $R - L + 1$ numbers to check, but in the worst case, $L = 1$ or some other small number, so we just say that there are $\mathcal{O}(R)$ candidate numbers. For each one, its check can take up to $\mathcal{O}(\text{no. of digits})$ operations (and we can show that this is $\mathcal{O}(\log_{10}(R))$. Therefore, the running time is $\mathcal{O}(R \log_{10} R)$. Unfortunately, with R possibly up to 10^{15}, this is *way* too slow.

The problem is that there are just too many integers from L to R. And if we look at the sample input, there aren't even that many SOS numbers from 1 to 10^{15} in the first place! Most of our checking just goes to waste.

"The integers from L to R" makes up our **search space**, because we are *searching* for SOS numbers in this range. The problem is that our search space is too large and too sparse. Is there maybe a better set that we can search through?

Well... how about we *swap* the search space and the check? See this pseudocode.

```
total = 0
for each SOS number k that is <= 1e15:
    if L <= k <= R:
        total += 1
print(total)
```

How large is this new search space? Well, SOS numbers $\leq 10^{15}$ must be strings of length at most 15 digits, each of which is either `5` or `0`. Actually, because of how leading zeroes work with our number system, we can just generate *all* such strings with exactly 15 digits. Because each digit can be one of two possible values, there are exactly $2^{15} = 32768$ such strings, which is not too large.

Of course, not all these strings are SOS numbers, but that's fine! Just filter out only the ones with `505` as a substring. There's still wasted work, but at least 2^{15} is a tolerable amount of work.

We can enumerate all SOS numbers using recursive backtracking.

```
all_SOS_numbers = []
def generate_SOS_numbers(i, SOS):
    if i == 0:
        if '505' in SOS:
```

```
            all_SOS_numbers.append(int(SOS))
    else:
        generate_SOS_numbers(i-1, SOS + '5')
        generate_SOS_numbers(i-1, SOS + '0')

B = 15
generate_SOS_numbers(B, '')
```

As a special hack, since we only care about binary strings, we can use the **binary representations** of the integers from 0 to $2^{15} - 1$ in order to generate all 15-length patterns with two symbols. This technique is called *bitmasking* and has loads of other applications as well.

The following code uses the "bitwise operations" left shift and bitwise AND, which you can look up. Many languages also may have their own built-in ways to extract the binary representation of an integer. For Python, you can use `bin`, and for C++, you can use `std::bitset`.

```
all_SOS_numbers = []

B = 15
for mask in range(1 << B):
    SOS = ''
    for i in range(B):
        if mask & (1 << i):  # if ith bit is 1
            SOS += '5'
        else:
            SOS += '0'

    if '505' in SOS:
        all_SOS_numbers.append(int(SOS))
```

In summary, this is our final code:

```
all_SOS_numbers = []

... # choose your preferred method of generating the SOS numbers

L, R = map(int, input().split())
total = 0
for SOS in all_SOS_numbers:
    if L <= SOS <= R:
        total += 1
print(total)
```

Let $B = \log_{10}(R)$ be the number of digits in the largest SOS number to consider. Then, our solution runs in $\mathcal{O}(B2^B)$, because there are 2^B two-character strings of length B, and testing each one for SOS-ness takes $\mathcal{O}(B)$ time, which is fast enough.

Bonus: The recursive backtracking solution can be modified so that the solution runs purely in $\mathcal{O}(2^B)$. Not that it's too important, because the bounds are small enough, but... can you think of how?

Bonus for experienced participants: Suppose there are T different test cases, each with their own L and R and asking how many SOS numbers lie in that range. Here, $1 \leq T \leq 2 \times 10^5$, so T can be quite large. Can you think of a fast way to answer each of these test cases? Say, suppose you should be able to answer each test case in $\mathcal{O}(\log(R))$ time. *Hint:* You can do some pre-processing before answering all the test cases.

Bonus Bonus for the *very* experienced participants: Suppose that $1 \leq L \leq R \leq 10^{2 \times 10^5}$, i.e. L and R can have up

to 2×10^5 *digits*. As is standard in comp prog, suppose we only want the answer modulo $10^9 + 7$. Can we answer this in $\mathcal{O}(B)$ (not $\mathcal{O}(2^B)$)?