Editorial:

Suppose a substring of length k appears twice in the S, clearly all substrings of length smaller than k will appear twice. Also, if no substring of length k appears twice in S, then it is impossible for substring of length greater than k to appear twice. What we can do, is binary search on the length of the substring to check if any of substring of length k repeats itself. If we find a substring of length k that repeats itself, we only need to look for substrings of length greater than k and vice versa.

So, now our problem reduces to checking the fact that given a length k, is there any substring of length k in S that repeats itself. One naive way would be to generate all the substrings of length k and store them in an unordered set, and check before every insertion if the substring already exists in the set. If it does, then we have found a repetition, else we insert the substring in the set. But, clearly, there would be at max n substrings of length k and the total complexity would turn out to be $O(n^2 \log(n))$ which would time out.

One way would be to store hashes of the substrings instead of the actual substring. Now, to calculate the hash of a substring of length k, O(k) operations are required, and there are at most n substrings, so the complexity is still $O(n^2 \log n)$.

How can we do better? It so happens that there exists a simple hash function such that if we have the hash of the first substring of length k, we can actually compute the hash of the next substring in O(1). It is called the **rolling hash** One of the major applications of rolling hash is in an algorithm for string matching, Rabin Karp. Thus, now using the rolling hash, we can check if the substring of length k repeats itself in O(n). Thus, the total complexity of the solution becomes, O(nlogn) which passes the time limit.