<https://leetcode.com/problems/longest-duplicate-substring/description/>

Given a string s, consider all *duplicated substrings*: (contiguous) substrings of s that occur 2 or more times. The occurrences may overlap.

Return **any** duplicated substring that has the longest possible length. If s does not have a duplicated substring, the answer is "".

**Example 1:**

Input: s = "banana"

Output: "ana"

**Example 2:**

Input: s = "abcd"

Output: ""

**Constraints:**

2 <= s.length <= 3 \* 10^4

s consists of lowercase English letters.

**Attempt 1: 2023-05-28**

**Solution 1: Binary Search + Rolling Harsh (120 min, TLE 66/67)**

class Solution {

    public String longestDupSubstring(String s) {

        int len = s.length();

        // 'lo' cannot start with 0 since its physical meaning is

        // substring length, the minimum length is 1

        // e.g "aa" -> if lo = 0, result = "", but expected as "a"

        //int lo = 0;

        int lo = 1;

        int hi = len - 1;

        String result = "";

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            String tmp = findDupSubstring(s, mid);

            if(tmp.isEmpty()) {

                // Current mid length substring cannot find any

                // duplicates, decrease mid length by 1 as next

                // iteration high end boundary is required

                hi = mid - 1;

            } else {

                // Current mid length substring able to find at

                // least one duplicate, but not guarantee current

                // mid length is the longest one, we can first

                // record current substring into result and move on

                result = tmp;

                lo = mid + 1;

            }

        }

        return result;

    }

    private String findDupSubstring(String s, int pattern\_len) {

        // 'mod' is a large prime number

        long mod = (1 << 31) - 1;

        // 'base' in Rabin-Karp usually take 256，since maximum ASCII of char is 255

        long base = 256;

        // 'multiplier' used for rolling hash

        long multiplier = 1;

        // Calculate 'multiplier' as base^(pattern\_len - 1)

        for(int i = 0; i < pattern\_len - 1; i++) {

            multiplier = (multiplier \* base) % mod;

        }

        // 'pattern\_hash' is the initial hash for substring as s[0, pattern\_len - 1]

        long pattern\_hash = 0;

        // Calculate 'pattern\_hash'

        for(int i = 0; i < pattern\_len; i++) {

            pattern\_hash = (pattern\_hash \* base + s.charAt(i)) % mod;

        }

        // Use map to resolve potential hash collision

        // <key, value> => <hashcode, list of all starting index with identical hash>

        Map<Long, List<Integer>> map = new HashMap<>();

        // The initial pattern\_hash mapping with index 0

        List<Integer> equalHashIdx = new ArrayList<Integer>();

        equalHashIdx.add(0);

        map.put(pattern\_hash, equalHashIdx);

        // Try all other substrings with rolling hash

        for(int i = pattern\_len; i < s.length(); i++) {

            // Update pattern\_hash with rolling harsh

            pattern\_hash = (base \* (pattern\_hash - s.charAt(i - pattern\_len) \* multiplier) + s.charAt(i)) % mod;

            if(pattern\_hash < 0) {

                pattern\_hash += mod;

            }

            // Another style for rolling hash

            //pattern\_hash = (base \* (pattern\_hash - s.charAt(i - pattern\_len) \* multiplier % mod + mod) + s.charAt(i)) % mod;

            List<Integer> list = map.get(pattern\_hash);

            if(list == null) {

                list = new ArrayList<Integer>();

                map.put(pattern\_hash, list);

            } else {

                for(int idx : list) {

                    // In case hash collision happens, double check if the newly

                    // found substring s[i - pattern\_len + 1, i + 1] match pattern

                    String pattern = s.substring(idx, idx + pattern\_len);

                    if(s.substring(i - pattern\_len + 1, i + 1).equals(pattern)) {

                        return pattern;

                    }

                }

            }

            // Adding new pattern start index into map

            list.add(i - pattern\_len + 1);

        }

        return "";

    }

}

**Solution 2: Binary Search + Rolling Harsh (10 min)**

**The prime initialize as (1 << 31) - 1 is the root cause of TLE (Time Limit Exceed on 66/67), seen from one more post and use the same way to generate prime, it works**

import java.math.BigInteger;

import java.util.Random;

class Solution {

    // 'mod' is a large prime number, but instead of setting as (1 << 31) - 1

    // and create TLE, we calculate as a random 31-bit prime

    //private static final int mod = (1 << 31) - 1;

    private static final int mod = BigInteger.probablePrime(31, new Random()).intValue();

    // 'base' in Rabin-Karp usually take 256，since maximum ASCII of char is 255

    private static final int base = 256;

    public String longestDupSubstring(String s) {

        int len = s.length();

        // 'lo' cannot start with 0 since its physical meaning is

        // substring length, the minimum length is 1

        // e.g "aa" -> if lo = 0, result = "", but expected as "a"

        //int lo = 0;

        int lo = 1;

        int hi = len - 1;

        String result = "";

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            String tmp = findDupSubstring(s, mid);

            if(tmp.isEmpty()) {

                // Current mid length substring cannot find any

                // duplicates, decrease mid length by 1 as next

                // iteration high end boundary is required

                hi = mid - 1;

            } else {

                // Current mid length substring able to find at

                // least one duplicate, but not guarantee current

                // mid length is the longest one, we can first

                // record current substring into result and move on

                result = tmp;

                lo = mid + 1;

            }

        }

        return result;

    }

    private String findDupSubstring(String s, int pattern\_len) {

        // 'multiplier' used for rolling hash

        long multiplier = 1;

        // Calculate 'multiplier' as base^(pattern\_len - 1)

        for(int i = 0; i < pattern\_len - 1; i++) {

            multiplier = (multiplier \* base) % mod;

        }

        // 'pattern\_hash' is the initial hash for substring as s[0, pattern\_len - 1]

        long pattern\_hash = 0;

        // Calculate 'pattern\_hash'

        for(int i = 0; i < pattern\_len; i++) {

            pattern\_hash = (pattern\_hash \* base + s.charAt(i)) % mod;

        }

        // Use map to resolve potential hash collision

        // <key, value> => <hashcode, list of all starting index with identical hash>

        Map<Long, List<Integer>> map = new HashMap<>();

        // The initial pattern\_hash mapping with index 0

        map.put(pattern\_hash, new ArrayList<>(Arrays.asList(0)));

        // Try all other substrings with rolling hash

        int from = 0;

        int to = pattern\_len;

        while(to < s.length()) {

            // Update pattern\_hash with rolling harsh

            pattern\_hash = (base \* (pattern\_hash - s.charAt(from++) \* multiplier) + s.charAt(to++)) % mod;

            if(pattern\_hash < 0) {

                pattern\_hash += mod;

            }

            // Another style for rolling hash

            //pattern\_hash = (base \* (pattern\_hash - s.charAt(i - pattern\_len) \* multiplier % mod + mod) + s.charAt(i)) % mod;

            List<Integer> list = map.get(pattern\_hash);

            if(list == null) {

                list = new ArrayList<Integer>();

                map.put(pattern\_hash, list);

            } else {

                for(int idx : list) {

                    // In case hash collision happens, double check if the newly

                    // found substring s[i - pattern\_len + 1, i + 1] match pattern

                    String pattern = s.substring(idx, idx + pattern\_len);

                    if(s.substring(from, to).equals(pattern)) {

                        return pattern;

                    }

                }

            }

            // Adding new pattern start index into map

            list.add(from);

        }

        return "";

    }

}

**Binary search O(n log n) average with Rabin-Karp, explained**

**Refer to**

<https://leetcode.com/problems/longest-duplicate-substring/solutions/695029/python-binary-search-o-n-log-n-average-with-rabin-karp-explained/>

This is quite difficult problem in fact and you will be very unlucky if you have this on real interview. However if you familiar with **Rabin-Karp** algorithm, it will be not so difficult, you can see it in full details on Wikipedia: <https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm>

Here I briefly explain the idea of it. Imagine, that we have string abcabcddcb, and we want to check if there are two substrings of size 3, which are equal. Idea is to **hash** this substrings, using rolling hash, where d is our **base** and q is used to avoid overflow.

for abc we evaluate [ord(a)\*d^2 + ord(b)\*d^1 + ord(c)\*d^0] % q

for bca we evaluate [ord(b)\*d^2 + ord(c)\*d^1 + ord(a)\*d^0] % q

Note, that we can evaluate rolling hash in O(1), for more details please see Wikipedia.

...

However, it can happen that we have **collisions**, we can falsely get two substrings with the same hash, which are not equal. So, we need to check our candidates.

**Binary search for help**

Note, that we are asked for the longest duplicate substring. If we found duplicate substring of length 10, it means that there are duplicate substrings of lengths 9,8, .... So, we can use **binary search** to find the longest one.

**Complexity** of algorithm is O(n log n) if we assume that probability of collision is low.

**How to read the code**

I have RabinKarp(text, M,q) function, where text is the string where we search patterns, M is the length we are looking for and q is prime modulo for Rabin-Karp algorithm.

First, we need to choose d, I chose d = 256, because it is more than ord(z). Rabin-Karp usually take 256, since maximum ASCII of char is 255

For q used 2^31 - 1 is because it is a big prime number, but not big enough, so all multiplications stay in int32 region and work fast enough. Yes, the higher it is, the less probability of collisions, but if you take it too big, it will be quite slow, you reduce probability of collision from say 1% to 0.5% and it is just not worth it. To be honest, I tried different q and choose the best working one.

Then we need to evaluate auxiliary value h, we need it for fast update of rolling hash.

Evaluate hash for first window

Evaluate hashes for all other windows in O(n) (that is O(1) for each window), using evaluated h.

We keep all hashes in **dictionary**: for each hash we keep start indexes of windows.

Finally, we iterate over our dictionary and for each unique hash we check all possible combinations and compare not hashes, but original windows to make sure that it was not a collision.

Now, to the longestDupSubstring(S) function we run binary search, where we run our RabinKarp function at most log n times.

class Solution:

    def RabinKarp(self,text, M, q):

        if M == 0: return True

        h, t, d = (1<<(8\*M-8))%q, 0, 256

        dic = defaultdict(list)

        for i in range(M):

            t = (d \* t + ord(text[i]))% q

        dic[t].append(i-M+1)

        for i in range(len(text) - M):

            t = (d\*(t-ord(text[i])\*h) + ord(text[i + M]))% q

            for j in dic[t]:

                if text[i+1:i+M+1] == text[j:j+M]:

                    return (True, text[j:j+M])

            dic[t].append(i+1)

        return (False, "")

    def longestDupSubstring(self, S):

        beg, end = 0, len(S)

        q = (1<<31) - 1

        Found = ""

        while beg + 1 < end:

            mid = (beg + end)//2

            isFound, candidate = self.RabinKarp(S, mid, q)

            if isFound:

                beg, Found = mid, candidate

            else:

                end = mid

        return Found

**Further discussion** I did some research in this domain, and there are different ways to handle this problem:

You can solve it, using **suffix array** in O(n log n) (or even O(n) with **suffix trees**) complexity, see <https://cp-algorithms.com/string/suffix-array.html> for suffix array which is a bit easier than suffix trees <https://en.wikipedia.org/wiki/Suffix_tree>

You can solve it with **Tries**, complexity To be checked.

I think there is solution where we use KMP, but I have not seen one yet.

**Step by step to understand the binary search solution**

**Refer to**

<https://leetcode.com/problems/longest-duplicate-substring/solutions/327643/step-by-step-to-understand-the-binary-search-solution/>

**Why use binary search**

Since the length of the answer must between 0 to the length of string minus 1, In the example one "banana", the answer must between 0 to 5, we can guess 3 at the first time. **We will check every possible substring with length 3 to see if we can find any duplicate.**

- ban

- ana

- nan

- ana

If we are lucky enough, like this case, 'ana' is what we want. since we want to get the longest one, so we guess 4 (middle of 3 and 5) in the next time, if we found any valid answer, we can update the old one.

**How to check duplicate substring**

The easiest way would be to use a hashmap or dictionary to store the substring (or the hash value of the substring). If the current string not in the hashmap, we put it in the hashmap, if it already existed, we return the string.

hashmap = {}

ban -> not in the hashmap -> hashmap = {ban}

ana -> not in the hashmap -> hashmap = {ban, ana}

nan -> not in the hashmap -> hashmap = {ban, ana, nan}

ana -> in the hashmap -> return 'ana'

But we will get **Memory Limit Exceeded** in leetcode if we did this. So we have to implement our own hash function by calculated a unique number for every substring then compare between them. This is kinda hacking, In the answer from @lee215,

def test(L):

  p = pow(26, L, mod)

  cur = reduce(lambda x, y: (x \* 26 + y) % mod, A[:L], 0)

  seen = {cur}

  for i in xrange(L, len(S)):

      cur = (cur \* 26 + A[i] - A[i - L] \* p) % mod

      if cur in seen: return i - L + 1

      seen.add(cur)

banana\_val -> [b,a,n,a,n,a] -> [1, 0, 13, 0, 13, 0] (we give every char a number)

ban -> ((1 \* 26) \* 26) + 13 = 689

ana -> ban + a - b = (689 \* 26) + 0 - 26 \* 26 \* 26 \* b = 338 ( the current **b** had been multiple by 26 three times)

...

Any finally we get 338 again, so we return 'ana'.

**Java Code Implementation**

**Refer to**

<https://leetcode.com/problems/longest-duplicate-substring/solutions/695419/java-o-n-log-n-rabin-karp-binary-search/comments/1415420>

Reference to rolling hash implementation in <https://algs4.cs.princeton.edu/53substring/RabinKarp.java.html> to handle hash collision and overflow:

import java.math.BigInteger;

import java.util.Random;

class Solution {

    private static final int BASE = 256;

    private static final int MOD = longRandomPrime();

    // Hint: Binary search for the length of the answer. (If there's an answer of length 10, then there are answers of length 9, 8, 7, ...)

    public String longestDupSubstring(String s) {

        String result = "";

        int low = 1, high = s.length() - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            String dup = findDupSubstring(s, mid);

            if (dup.isEmpty()) {

                high = mid - 1;

            } else {

                result = dup;

                low = mid + 1; // dup is recorded, so it's safe to move low beyond mid

            }

        }

        return result;

    }

    // Find if duplicate substring in length K by Rabin-Karp algorithm

    private String findDupSubstring(String s, int k) {

        Map<Long, List<Integer>> hashes = new HashMap<>();

        long hash = 0, power = 0;

        for (int i = 0; i < k; i++) {

            power = (i == 0) ? 1 : power \* BASE % MOD;

            hash = (hash \* BASE + s.charAt(i)) % MOD;

        }

        hashes.put(hash, new ArrayList<>(Arrays.asList(0)));

        for (int i = 1; i <= s.length() - k; i++) {

            hash = (hash + MOD - s.charAt(i - 1) \* power % MOD) % MOD;

            hash = (hash \* BASE + (s.charAt(i + k - 1))) % MOD;

            List<Integer> indexes = hashes.get(hash);

            if (indexes == null) {

                indexes = new ArrayList<>();

                hashes.put(hash, indexes);

            } else { // Check if collison or real duplicates

                String cur = s.substring(i, i + k);

                for (int from : indexes) {

                    if (cur.equals(s.substring(from, from + k))) {

                        return cur;

                    }

                }

            }

            indexes.add(i);

        }

        return "";

    }

    // a random 31-bit prime

    private static int longRandomPrime() {

        BigInteger prime = BigInteger.probablePrime(31, new Random());

        return prime.intValue();

    }

}

**Refer to**

<https://leetcode.com/problems/longest-duplicate-substring/solutions/290871/python-binary-search/comments/274729/>

class Solution {

    public String longestDupSubstring(String S) {

        // edge case

        if (S == null) {

            return null;

        }

        // binary search the max length

        int min = 0;

        int max = S.length() - 1;

        int mid;

        while (min < max - 1) {

            mid = (min + max) / 2;

            if (searchForLength(S, mid) != null) {

                min = mid;

            } else {

                max = mid - 1;

            }

        }

        String str = searchForLength(S, max);

        if (str != null) {

            return str;

        } else {

            return searchForLength(S, min);

        }

    }

    String searchForLength(String str, int len) {

        // rolling hash method

        if (len == 0) {

            return "";

        } else if (len >= str.length()) {

            return null;

        }

        Map<Long, List<Integer>> map = new HashMap<>();    // hashcode -> list of all starting idx with identical hash

        long p = (1 << 31) - 1;  // prime number

        long base = 256;

        long hash = 0;

        for (int i = 0; i < len; ++i) {

            hash = (hash \* base + str.charAt(i)) % p;

        }

        long multiplier = 1;

        for (int i = 1; i < len; ++i) {

            multiplier = (multiplier \* base) % p;

        }

        // first substring

        List<Integer> equalHashIdx = new ArrayList<Integer>();

        equalHashIdx.add(0);

        map.put(hash, equalHashIdx);

        // other substrings

        int from = 0;

        int to = len;

        while (to < str.length()) {

            hash = ((hash + p - multiplier \* str.charAt(from++) % p) \* base + str.charAt(to++)) % p;

            equalHashIdx = map.get(hash);

            if (equalHashIdx == null) {

                equalHashIdx = new ArrayList<Integer>();

                map.put(hash, equalHashIdx);

            } else {

                for (int i0: equalHashIdx) {

                    if (str.substring(from, to).equals(str.substring(i0, i0 + len))) {

                        return str.substring(i0, i0 + len);

                    }

                }

            }

            equalHashIdx.add(from);

        }

        return null;

    }

}

**Rabin Karp article refer to (in Document)**

<https://zhuanlan.zhihu.com/p/563551141>

<https://coolcao.com/2020/08/20/rabin-karp/#:~:text=Rabin%20Karp%E7%AE%97%E6%B3%95%E5%8D%B3%E6%98%AF,%E5%88%A4%E6%96%AD%E5%85%B6%E6%98%AF%E5%90%A6%E7%9C%9F%E6%AD%A3%E7%9B%B8%E7%AD%89%E3%80%82>

<https://blog.csdn.net/qq_39559641/article/details/122284388>

**Refer to**

[L214.Shortest Palindrome (Ref.L1044,L1698)](note://C98FF24B295744F888F2B8C18AD71093)

[Rabin-Karp Algorithm](note://1DFFF586710745318EE15736B2372884)

[Rabin-Karp 指纹字符串查找算法](note://49C117E848D24EB4A587297908FCDF55)