Competitive Coding @ Berkeley

DeCal Lecture #11 - Rolling Hashes

Announcements

- Last lecture next week!
    - We will do a quick review of the topics covered so far
- Final contest:
    - Will be just a longer assignment (worth the same amount of points)

Motivating Problem

Insert motivational quote here

Motivating Problem: Find all occurrences of substring

We're given a text $\mathbf{T}$ and a pattern $\mathbf{P}$ where $\mathbf{T}$ and $\mathbf{P}$ are strings made of lowercase letters in the English alphabet. Find the positions of all occurrences of $\mathbf{P}$ in $\mathbf{T}$.

Example:
$\mathbf{T}$ = `caccaccacccaaccacca`
$\mathbf{P}$ = `ccac`

Motivating Problem: Find all occurrences of substring

We're given a text $\mathbf{T}$ and a pattern $\mathbf{P}$ where $\mathbf{T}$ and $\mathbf{P}$ are strings of lowercase letters in the English alphabet. Find the positions of all occurrences of $\mathbf{P}$ in $\mathbf{T}$.

Example:
$\mathbf{T}$ = **caccaccacccaaccacca**
$\mathbf{P}$ = **ccac**

3 occurrences, at indices [2, 5, 13]:

```
0123456789012345678
caccaccacccaaccacca
caccaccacccaaccacca
caccaccacccaaccacca
```

Motivating Problem: Find all occurrences of substring

Consider the following brute force algorithm below. What's the time complexity?

```
ans = []
for each starting position i in $\mathbf{T}$ (between 0 and $|\mathbf{T}|$ - $|\mathbf{P}|$ inclusive):
    for each position j in $\mathbf{P}$:
        if T[i] != P[j]:
            break because mismatch
    if all positions match up:
        append i to ans
return ans
```

Motivating Problem: Find all occurrences of substring

Complexity of brute force is O(|**T**||**P**|)

Why?

Consider the input:
T = `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`
P = `aaaaaaaaaa`

Observation:
We're making a bunch of redundant and unnecessary comparisons when we shift just one character over and re-check the entire pattern. Is there a way to "keep" some of this information as we go along so we don't have to re-check the entire pattern?

Rolling Hashes

Rolling Hashes: Main Ideas

- We maintain a window and hash of the window as we're going through a string
    - Wow! 2 pointers!
- Whenever we slide the window over by 1 character, we update the hash using the existing hash
    - Existing hash contains the "information" of previous calculation, so we don't have to re-check the entire window
- When the same string is hashed, it will always have the same hash value. If two strings are hashed and have the same value, it is very likely that they are the same string!
    - Only false positives, no false negatives
    - Since we design a system that makes false positives unlikely, we just assume same hash ⇒ same string
- Since hashes are just integers, we can compare them and do math with them in O(1)
- Need to design a special hash function that achieves this

Rolling Hashes: Rabin-Karp

There are many ways to achieve this, but one of the most popular is polynomial rolling hashes with the Rabin-Karp algorithm.

The basic idea is that we treat the string as a modular polynomial, where the base is some prime number larger than the size of the alphabet, and the coefficients are determined by the characters of the string.

Suppose we have some string $\mathbf{S}$. Then, we define:

$$\text{hash}(\mathbf{S}) = \mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}|-1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}|-2} + \ldots + \mathbf{S}(|\mathbf{S}|-2) * \mathbf{B}^{1} + \mathbf{S}(|\mathbf{S}|-1) * \mathbf{B}^{0} \bmod \mathbf{M}$$

$$= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,$$

Where $\mathbf{B}$ (the base) and $\mathbf{M}$ (the mod) are prime numbers that we can choose, and $\mathbf{S}$(i) maps the character at the ith position of $\mathbf{S}$ into some number in $[1, \mathbf{B})$

Rolling Hashes: Computing an Example

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^{1} + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^{0} \bmod \mathbf{M}$

Compute hash($\mathbf{S}$) given the following:

$\mathbf{S}$ = `hello`

$\mathbf{S}$(i) = position of i in alphabet, so $\mathbf{S}$(`h`) = 8, $\mathbf{S}$(`e`) = 5, $\mathbf{S}$(`l`) = 12, $\mathbf{S}$(`o`) = 15

$\mathbf{B}$ = 31

$\mathbf{M}$ = 101

Rolling Hashes: Computing an Example

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^1 + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^0 \bmod \mathbf{M}$

Compute hash($\mathbf{S}$) given the following:
$\mathbf{S}$ = `hello`
$\mathbf{S}$(i) = position of i in alphabet, so $\mathbf{S}$(`h`) = 8, $\mathbf{S}$(`e`) = 5, $\mathbf{S}$(`l`) = 12, $\mathbf{S}$(`o`) = 15
$\mathbf{B} = 31$
$\mathbf{M} = 101$

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^4 + \mathbf{S}(1) * \mathbf{B}^3 + \mathbf{S}(2) * \mathbf{B}^2 + \mathbf{S}(3) * \mathbf{B}^1 + \mathbf{S}(4) * \mathbf{B}^0 \bmod \mathbf{M}$
hash($\mathbf{S}$) = $8 * 31^4 + 5 * 31^3 + 12 * 31^2 + 12 * 31^1 + 15 * 31^0 \bmod 101$
hash($\mathbf{S}$) = $7388168 + 148955 + 11532 + 372 + 15 \bmod 101$
hash($\mathbf{S}$) = 100

Rolling Hashes: Implementation

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 7;
    long long hash_value = 0;
    for (char c : s) {
        hash_value = (p * hash_value + (c - 'a' + 1)) % m;
    }
    return hash_value;
}
```

Rolling Hashes: Computing an Update

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^{1} + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^{0} \bmod \mathbf{M}$

Now that we know hash(`hello`) = 100, suppose we want to find hash(`elloh`):

$\mathbf{S}$ = `elloh`

$\mathbf{S}$(i) = position of i in alphabet, so $\mathbf{S}$(`h`) = 8, $\mathbf{S}$(`e`) = 5, $\mathbf{S}$(`l`) = 12, $\mathbf{S}$(`o`) = 15

$\mathbf{B} = 31$

$\mathbf{M} = 101$

Rolling Hashes: Computing an Update

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^{1} + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^{0} \bmod \mathbf{M}$

Now that we know hash(`hello`) = 100, suppose we want to find hash(`elloh`):
$\mathbf{S}$ = `elloh`; $\mathbf{S}$(i) = position of i in alphabet, so $\mathbf{S}$(`h`) = 8, $\mathbf{S}$(`e`) = 5, $\mathbf{S}$(`l`) = 12, $\mathbf{S}$(`o`) = 15; $\mathbf{B}$ = 31; $\mathbf{M}$ = 101

```
temp = hash("hello") // previous hash
temp -= S("hello"(0)) * B^4 // remove the first character from equation
temp *= B // shift all the coefficients to the left
temp += S("elloh"(4)) // add on the last coefficient
hash("elloh") = temp // et voila!
// btw the answer was 25 if anyone was actually calculating lol
```

Time complexity of computing an update?

Rolling Hashes: Selecting Parameters

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^{1} + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^{0} \bmod \mathbf{M}$

- **B** (the base)
  - Prime
  - At least as big as the size of your alphabet + 1
    - Otherwise, you won't be able to fit everything and you'll get collisions
    - 31 is good if alphabet is English alphabet
    - 53 is good if alphabet is alphanumeric
- **M** (the mod)
  - Prime
  - Should be large to make collisions less likely
    - 10^9+7 is typically a good number, lets you do multiplication without worrying about 64-bit overflow
    - If you use Python, feel free to go nuts (but arithmetic on bigger numbers is a bit slower)
- **S**(i) (character to integer mapping)
  - Typically, `a` => 1, `b` => 2, `c` => 3, etc
  - DANGER: DO NOT MAP CHARACTERS TO 0
    - If you map `a` to 0, then hash(`a`) = hash(`aa`) = hash(`aaa`) and bad things will happen

Rolling Hashes: Hash Collisions

- A collision happens when two strings hash into the same value
- Using our previous parameters for $\mathbf{B}$ and $\mathbf{M}$, hash(`hello`) = hash(`aaabm`)
- We want to assume (same hash) ⇒ (same strings) but this is not true on collision
- Fortunately, some very smart math people have proven that this is very rare if we pick our parameters right
    - Pick $\mathbf{M}$ to be as large as possible
    - Then, if we randomize $\mathbf{B}$, pairwise collision probability is ~$1/\mathbf{M}$
    - By birthday paradox, the probability of any collision happening when hashing $\mathbf{m}$ strings with length of $\mathbf{n}$ is $\mathbf{m}^2\mathbf{n}/\mathbf{M}$
- In practice, don't think too hard about the exact math. Just guestimate and you'll probably be fine.

Rolling Hashes: Solving the Motivating Problem

We're given a text $\mathbf{T}$ and a pattern $\mathbf{P}$ where $\mathbf{T}$ and $\mathbf{P}$ are strings of lowercase letters in the English alphabet. Find the positions of all occurrences of $\mathbf{P}$ in $\mathbf{T}$.

Example:
$\mathbf{T}$ = `caccaccacccaaccacca`
$\mathbf{P}$ = `ccac`

How can we solve this problem using rolling hashes in O(|$\mathbf{T}$|+|$\mathbf{P}$|) time and O(1) space?

Rolling Hashes: Solving the Motivating Problem

We're given a text $\mathbf{T}$ and a pattern $\mathbf{P}$ where $\mathbf{T}$ and $\mathbf{P}$ are strings of lowercase letters in the English alphabet. Find the positions of all occurrences of $\mathbf{P}$ in $\mathbf{T}$.

1. Since we have lowercase letters, we use $\mathbf{B}$ = 31. We also choose $\mathbf{M} = 10^9 + 7$ (any sufficiently large $\mathbf{M}$ is good).
2. Compute hash($\mathbf{P}$). We will check all hashes of substrings of length $|\mathbf{P}|$ in $\mathbf{T}$ against this.
3. Compute hash($\mathbf{T}[:|\mathbf{P}|]$) (the first $|\mathbf{P}|$ characters in $\mathbf{T}$).
4. Perform $\mathbf{T}$ - $\mathbf{P}$ updates where we find hash($\mathbf{T}[i:|\mathbf{P}|+i]$) for each start position i. If this is equal to hash($\mathbf{P}$), then we've found the start position of an occurrence.
5. Return the answer

We check the hashes of $\mathbf{T}$ - $\mathbf{P}$ strings. Collisions are only problematic if they collide with hash($\mathbf{P}$). The collision probability is ~10^{-9}. If $|\mathbf{T}| = 10^5$, then the probability of collision is ~$10^{-4} = 0.0001$.

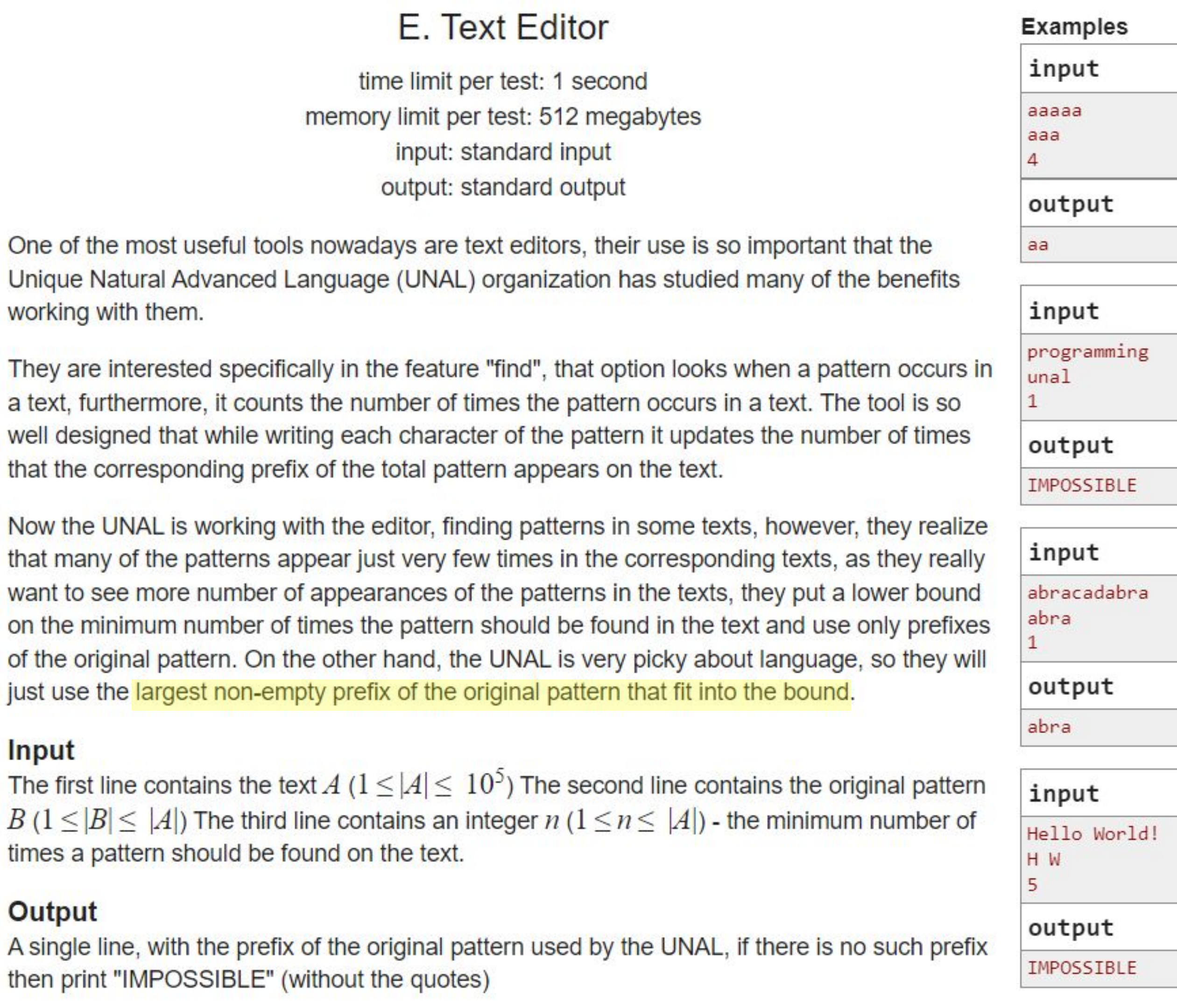
Bonus: what if we get super unlucky and we still do get a collision? How can we fix the algorithm by changing 1 character in our code?

Practice

aka free homework answers!

Text Editor

https://codeforces.com/gym/101466/problem/E



E. Text Editor

time limit per test: 1 second
memory limit per test: 512 megabytes
input: standard input
output: standard output

One of the most useful tools nowadays are text editors, their use is so important that the Unique Natural Advanced Language (UNAL) organization has studied many of the benefits working with them.

They are interested specifically in the feature "find", that option looks when a pattern occurs in a text, furthermore, it counts the number of times the pattern occurs in a text. The tool is so well designed that while writing each character of the pattern it updates the number of times that the corresponding prefix of the total pattern appears on the text.

Now the UNAL is working with the editor, finding patterns in some texts, however, they realize that many of the patterns appear just very few times in the corresponding texts, as they really want to see more number of appearances of the patterns in the texts, they put a lower bound on the minimum number of times the pattern should be found in the text and use only prefixes of the original pattern. On the other hand, the UNAL is very picky about language, so they will just use the largest non-empty prefix of the original pattern that fit into the bound.

Input

The first line contains the text A $(1 \leq |A| \leq 10^5)$ The second line contains the original pattern B $(1 \leq |B| \leq |A|)$ The third line contains an integer n $(1 \leq n \leq |A|)$ - the minimum number of times a pattern should be found on the text.

Output

A single line, with the prefix of the original pattern used by the UNAL, if there is no such prefix then print "IMPOSSIBLE" (without the quotes)

Examples

input
```
aaaaa
aaa
4
```
output
```
aa
```

input
```
programming
unal
1
```
output
```
IMPOSSIBLE
```

input
```
abracadabra
abra
1
```
output
```
abra
```

input
```
Hello World!
H W
5
```
output
```
IMPOSSIBLE
```

Text Editor: Solution

For any prefix, we can use the same algorithm for the substring occurrence search problem to count the number of occurrences in O($|\mathbf{A}|$ + $|\mathbf{B}|$) using base = 31 and mod = $10^9 + 7$.

Observe that if a prefix of $\mathbf{B}$ with length l yields k occurrences, then a prefix with length l - 1 must also yield at least k occurrences.

Since we want to find the longest prefix, this is a perfect opportunity to binary search for the answer!

Binary search over [1, $|\mathbf{B}|$] for the longest possible prefix that works. The search runs O(log($|\mathbf{B}|$)) iterations. Each iteration takes O($|\mathbf{A}|$ + $|\mathbf{B}|$) to check if it's good. Thus, we have an overall runtime complexity of O(($|\mathbf{A}|$ + $|\mathbf{B}|$)log($|\mathbf{B}|$)).

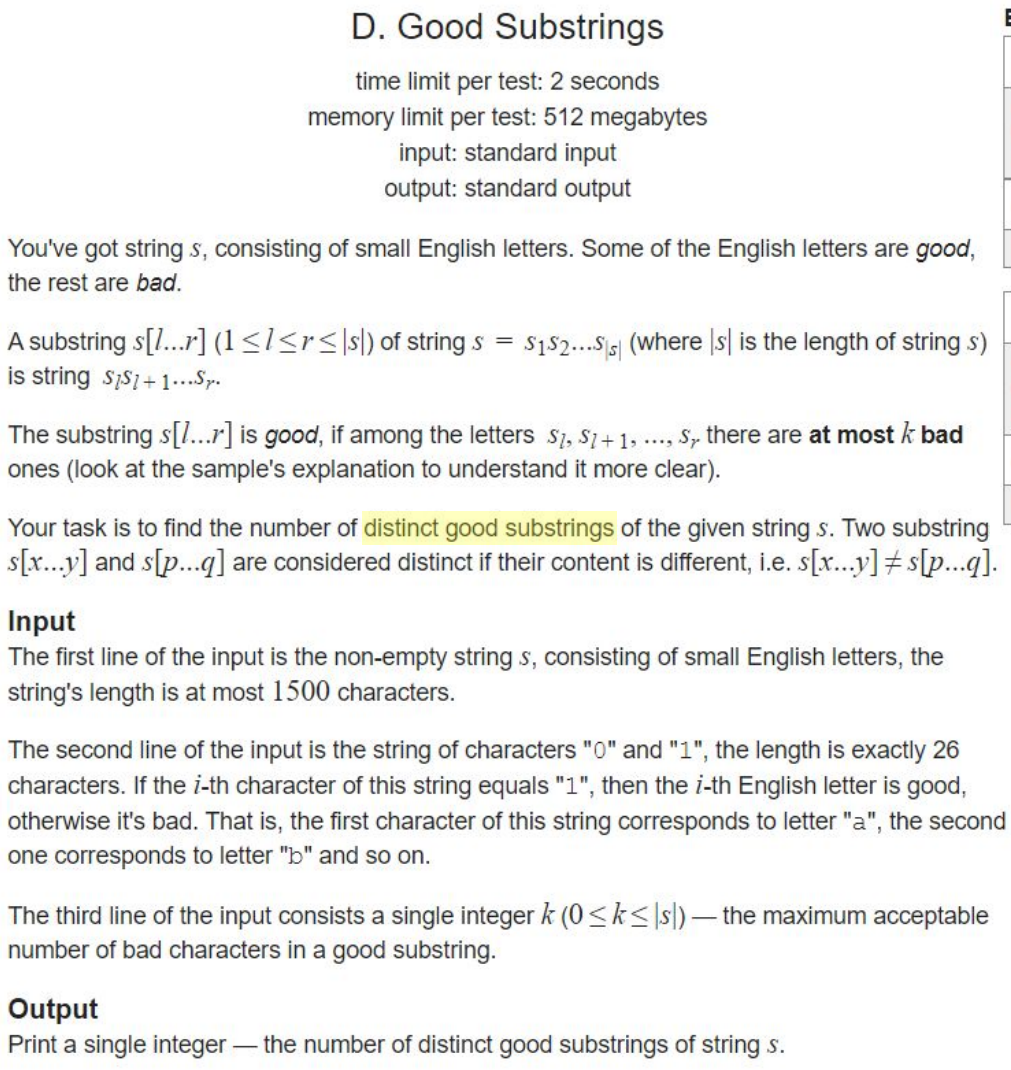
Text Editor: Collision Analysis

Collisions are only a problem within each iteration of binary search. Collisions that happen across iterations don't matter since the strings are of different lengths.

Like out previous analysis, in each iteration, we check the hashes of no more than $|\mathbf{A}|$ strings and we only care about collisions with the selected prefix of $\mathbf{B}$. The collision probability is ~10^{-9}. Since $|\mathbf{A}| <= 10^5$, the the probability of collision is ~10^{-4} = 0.0001.

Good Substrings

https://codeforces.com/contest/271/problem/D



D. Good Substrings

time limit per test: 2 seconds
memory limit per test: 512 megabytes
input: standard input
output: standard output

You've got string s, consisting of small English letters. Some of the English letters are *good*, the rest are *bad*.

A substring $s[l...r]$ $(1 \le l \le r \le |s|)$ of string $s = s_1s_2...s_{|s|}$ (where $|s|$ is the length of string s) is string $s_ls_{l+1}...s_r$.

The substring $s[l...r]$ is *good*, if among the letters $s_l, s_{l+1}, ..., s_r$ there are **at most** k **bad** ones (look at the sample's explanation to understand it more clear).

Your task is to find the number of distinct good substrings of the given string s. Two substring $s[x...y]$ and $s[p...q]$ are considered distinct if their content is different, i.e. $s[x...y] \neq s[p...q]$.

Input

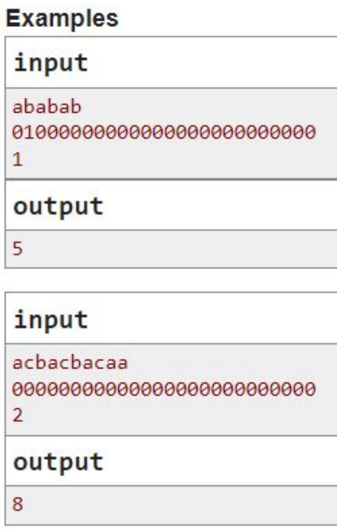
The first line of the input is the non-empty string s, consisting of small English letters, the string's length is at most 1500 characters.

The second line of the input is the string of characters "0" and "1", the length is exactly 26 characters. If the i-th character of this string equals "1", then the i-th English letter is good, otherwise it's bad. That is, the first character of this string corresponds to letter "a", the second one corresponds to letter "b" and so on.

The third line of the input consists a single integer k $(0 \le k \le |s|)$ — the maximum acceptable number of bad characters in a good substring.

Output

Print a single integer — the number of distinct good substrings of string s.



Examples

input

```
ababab
01000000000000000000000000
1
```

output

```
5
```

input

```
acbacbacaa
00000000000000000000000000
2
```

output

```
8
```

In the first example there are following good substrings: "a", "ab", "b", "ba", "bab".

In the second example there are following good substrings: "a", "aa", "ac", "b", "ba", "c", "ca", "cb".

Good Substrings: Solution

String length is at most 1500 characters. Since there are n * (n + 1) / 2 possible substrings in a string of length n, we only have 1125750 possible options.

We can do a complete search!

Consider every possible string length. Use two pointers to go through while keeping track of the number of bad letters and also keeping track of the rolling hash. If the number of bad letters is within the allowed threshold, add the hash to a set. Sum up the sizes of each set.

We only care about the number of distinct substrings, but since we know that collisions are rare, we assume that different hashes imply different strings.

Good Substrings: Collision Analysis

String length is at most 1500 characters. Since there are n * (n + 1) / 2 possible substrings in a string of length n, we only have 1125750 possible options.

If we naively put all the hashes into a set regardless of string length, we will add up to 1125750 hashes. If any of them collides with one another, we'll have an issue, so the birthday paradox comes in and screws us over.

However, if we separate our sets by length, then there are only <= 1500 distinct strings $1500^2/10^9$ = 0.00225, which is probably safe.

Post-lecture edit: there seems to be some anti-hashing tests for this problem, but they're quite weak. One way to get around this is to use double-hashing with two different $\mathbf{B}$ values. Another way (if you're using Python) is to simply use a bigger $\mathbf{M}$.

Rolling Hash Prefix Arrays

h ha has hash hashe hashes

Rolling Hash Prefix Arrays

Let's build on the idea of prefix sum arrays. By using O($\mathbf{S}$) extra space, we can create a data structure that allows us to compute the hash of *any* substring in $\mathbf{S}$ in O(1)!

hash($\mathbf{S}$) = $\mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^{1} + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^{0} \bmod \mathbf{M}$

The main idea is to build an array where arr[i] gives you hash($\mathbf{S}$[:i]). This can be done in linear time by the recurrence:

arr[i] = arr[i - 1] * $\mathbf{B}$ + $\mathbf{S}$(i) mod $\mathbf{M}$

This is basically just the same idea behind prefix sums!

Rolling Hash Prefix Arrays: Fast Substring Hashing

$\text{hash}(\mathbf{S}) = \mathbf{S}(0) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(|\mathbf{S}| - 2) * \mathbf{B}^1 + \mathbf{S}(|\mathbf{S}| - 1) * \mathbf{B}^0 \bmod \mathbf{M}$
$\text{arr}[i] = \text{hash}(\mathbf{S}[:i]) = \text{arr}[i - 1] * \mathbf{B} + \mathbf{S}(i) \bmod \mathbf{M}$

To compute the hash of a substring, we can simply use subtraction and multiply by the inverse.

$\text{hash}(\mathbf{S}[i:j]) = \mathbf{S}(i) * \mathbf{B}^{|\mathbf{S}| - 1} + \mathbf{S}(i + 1) * \mathbf{B}^{|\mathbf{S}| - 2} + \ldots + \mathbf{S}(j - 2) * \mathbf{B}^1 + \mathbf{S}(j - 1) * \mathbf{B}^0 \bmod \mathbf{M}$
$\text{hash}(\mathbf{S}[i:j]) = (\text{hash}(\mathbf{S}[:j]) - \text{hash}(\mathbf{S}[:i]) * \mathbf{B}^{j - i}) \bmod \mathbf{M}$
$\text{hash}(\mathbf{S}[i:j]) = (\text{arr}[j] - \text{arr}[i] * \mathbf{B}^{j - i}) \bmod \mathbf{M}$

This method is very useful if we don't know the length of the string beforehand, or if rolling over every possible length would take too long.

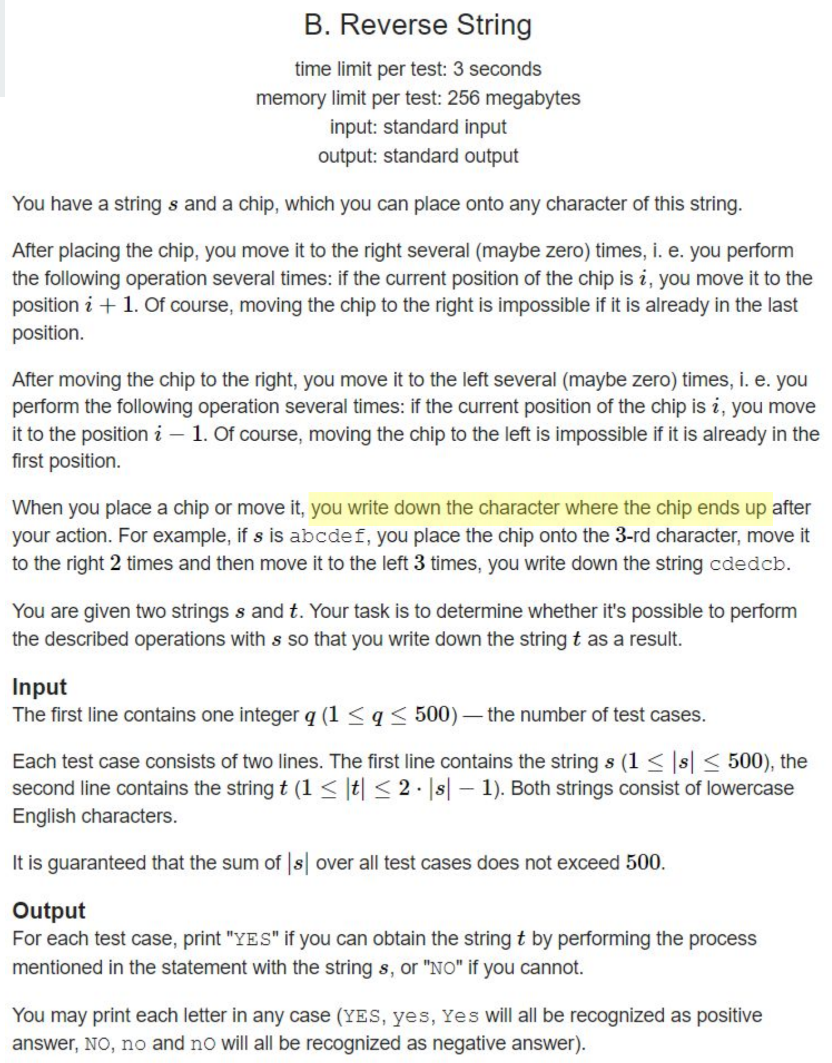
More Practice

free homework answers part 2: electric boogaloo

Reverse String

DO THIS IN O(|s|²)

https://codeforces.com/problemset/problem/1553/B



B. Reverse String

time limit per test: 3 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

You have a string s and a chip, which you can place onto any character of this string.

After placing the chip, you move it to the right several (maybe zero) times, i. e. you perform the following operation several times: if the current position of the chip is i, you move it to the position $i + 1$. Of course, moving the chip to the right is impossible if it is already in the last position.

After moving the chip to the right, you move it to the left several (maybe zero) times, i. e. you perform the following operation several times: if the current position of the chip is i, you move it to the position $i - 1$. Of course, moving the chip to the left is impossible if it is already in the first position.

When you place a chip or move it, you write down the character where the chip ends up after your action. For example, if s is `abcdef`, you place the chip onto the 3-rd character, move it to the right 2 times and then move it to the left 3 times, you write down the string `cdedcb`.

You are given two strings s and t. Your task is to determine whether it's possible to perform the described operations with s so that you write down the string t as a result.

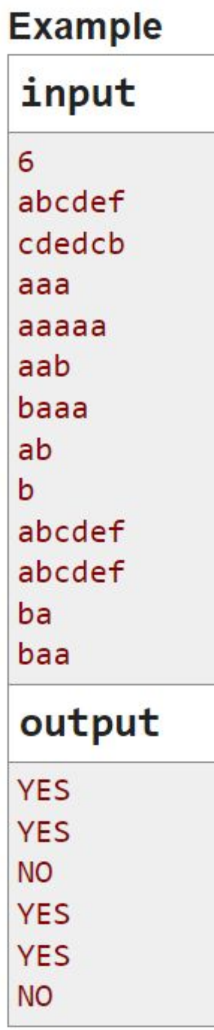
Input
The first line contains one integer q $(1 \le q \le 500)$ — the number of test cases.

Each test case consists of two lines. The first line contains the string s $(1 \le |s| \le 500)$, the second line contains the string t $(1 \le |t| \le 2 \cdot |s| - 1)$. Both strings consist of lowercase English characters.

It is guaranteed that the sum of $|s|$ over all test cases does not exceed 500.

Output
For each test case, print "`YES`" if you can obtain the string t by performing the process mentioned in the statement with the string s, or "`NO`" if you cannot.

You may print each letter in any case (`YES`, `yes`, `Yes` will all be recognized as positive answer, `NO`, `no` and `nO` will all be recognized as negative answer).



Example

input

```
6
abcdef
cdedcb
aaa
aaaaa
aab
baaa
ab
b
abcdef
abcdef
ba
baa
```

output

```
YES
YES
NO
YES
YES
NO
```

Reverse String: Solution

This problem can be solved in $O(|\mathbf{s}|^3)$, but for the sake of exercise, please solve it in $O(|s|^2)$!

Complete search again: $|\mathbf{s}|$ initial chip positions, $|t|$ places to change direction for each position. For each of the $|\mathbf{s}||\mathbf{t}|$ configurations, we can naively scan through in $O(|\mathbf{t}|)$. Since $|t| <= |\mathbf{s}|$, this solution is $O(|\mathbf{s}|^3)$.

However, we can do better! Notice that ignore initial positions, there are only $O(|\mathbf{t}|)$ different possible substrings that can yield our answer.

We can compute all of these, then for each one, check each starting position to see if it exists in s. Although the strings have different lengths, we can construct a prefix array to get the hash of any substring in O(1). This method therefore only uses $O(|\mathbf{s}|^2)$.

Collision analysis is left as an exercise. But the fact that $|\mathbf{s}| <= 500$ means that it's basically not an issue.

Variations and Final Thoughts

Variations

- Modify the hash function to support some additional properties
    - What if you didn't care about exact ordering, but only care about the count of each character in window?
- Use multiple hash functions for robustness against collisions
    - Assume strings are equal if both hash functions return the same value. Double collisions are much more rare than single collisions.
- Using other auxiliary data structures/algorithms/techniques
    - Binary search, dynamic programming, etc

Further Reading

- Codeforces blog: *On the mathematics behind rolling hashes and anti-hash tests*
  - https://codeforces.com/blog/entry/60442
  - International Grandmaster explains why this scheme is reliable. Hope you like math!
- Codeforces blog: *[Tutorial] Rolling hash and 8 interesting problems [Editorial]*
  - https://codeforces.com/blog/entry/60445
  - Some classic string problems and how to solve them using rolling hashes.
- cp-algorithms.com: *String Hashing*
  - https://cp-algorithms.com/string/string-hashing.html
  - Excellent list of even more practice problems!
- CS 176: *Algorithms for Computational Biology*
  - https://www2.eecs.berkeley.edu/Courses/CS176/
  - First 1/3 of the class is dedicated to string algorithms! Although they don't discuss hashing, they discuss a lot of other string algorithms and theory. If you're interested in string algorithms, consider taking!

Homework Tips

- You'll be given a library that implements all the methods we've discussed for you. Feel free to copy-paste this into your solution!
- Using the `**` operator on large powers in Python is slow. If you're going to mod the result anyway, use the much faster builtin `pow(number, power, modulus)` function instead.

Attendance

Thank you all for coming!