Untitled Note

44

lzl124631x ★258

Last Edit: October 8, 2018 11:18 AM

6.4K VIEWS

You can find more of my code/explanations in my github repo lzl124631x/LeetCode

It's easy to come up with a brute force solution and to find that there will be a **repetitive pattern** when matching `S2` through `S1`. The only problem is how to use the repetitive pattern to save computation.

Fact:
If `s2` repeats in `S1` `R` times, then `S2` must repeats in `S1` `R / n2` times.
Conclusion:
We can simply count the repetition of `s2` and then divide the count by `n2`.

How to denote repetition:
We need to scan `s1` `n1` times. Denote each scanning of `s1` as an `s1` segment.
After each scanning of `i`-th `s1` segment, we will have

1. The accumulative count of `s2` repeated in this `s1` segment.
2. A `nextIndex` that `s2[nextIndex]` is the first letter you'll be looking for in the next `s1` segment.

> Suppose `s1="abc"`, `s2="bac"`, `nextIndex` will be `1`; `s1="abca"`, `s2="bac"`, `nextIndex` will be `2`

It is the `nextIndex` that is the denotation of the repetitive pattern.

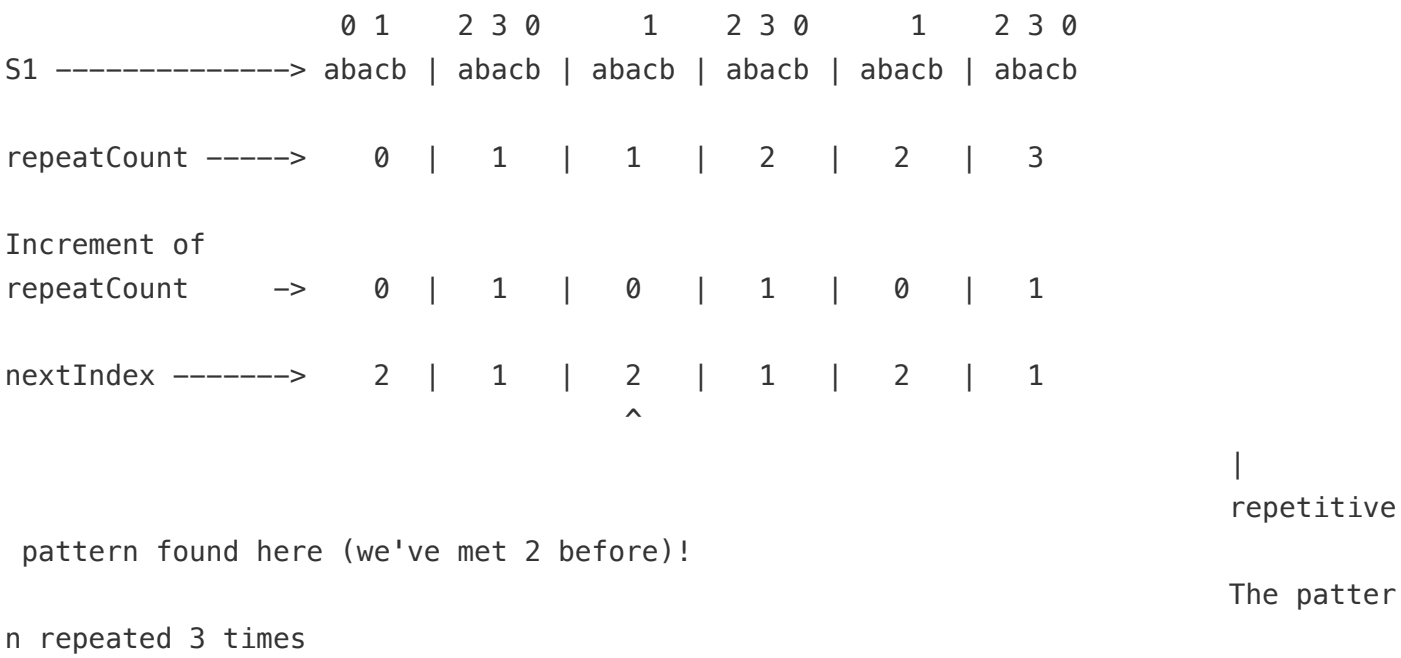
Example:

```
Input:
```

s1="abacb", n1=6
s2="bcaa", n2=1

Return:

3



The `nextIndex` has `s2.size()` possible values, ranging from `0` to `s2.size() - 1`. Due to PigeonHole principle, you must find two same `nextIndex` after scanning `s2.size() + 1` `s1` segments.

Once you meet a `nextIndex` you've met before, you'll know that the following `nextIndex`s and increments of `repeatCount` will repeat a pattern.

So let's separate the `s1` segments into 3 parts:

1. the prefix part before repetitive pattern
2. the repetitive part
3. the suffix part after repetitive pattern (incomplete repetitive pattern remnant)

All you have to do is add up the repeat counts of the 3 parts.

```
// OJ: https://leetcode.com/problems/count-the-repetitions/
// Author: github.com/lzl124631x
// Time: O(|s1| * n1) where |s1| is the length of s1
// Space: O(n1)
class Solution {
public:
    int getMaxRepetitions(string s1, int n1, string s2, int n2) {
        vector<int> repeatCount(n1 + 1, 0);
        vector<int> nextIndex(n1 + 1, 0);
```

```
        int j = 0, cnt = 0;
        for (int k = 1; k <= n1; ++k) {
            for (int i = 0; i < s1.size(); ++i) {
                if (s1[i] == s2[j]) {
                    ++j;
                    if (j == s2.size()) {
                        j = 0;
                        ++cnt;
                    }
                }
            }
            repeatCount[k] = cnt;
            nextIndex[k] = j;
            for (int start = 0; start < k; ++start) {
                if (nextIndex[start] == j) { // see if you have met this nextIndex
before
                    // if found, you can calculate the 3 parts
                    int prefixCount = repeatCount[start]; // prefixCount is the sta
rt-th repeatCount
                    // (repeatCount[k] - prefixCount) is the repeatCount of one occ
urrance of the pattern
                    // There are (n1 - start) / (k - start) occurrances of the patt
ern
                    // So (n1 - start) / (k - start) * (repeatCount[k] - prefixCoun
t) is the repeatCount of the repetitive part
                    int patternCount = (n1 - start) / (k - start) * (repeatCount[k]
 - prefixCount);
                    // The suffix contains the incomplete repetitive remnant (if an
y)
                    // Its length is (n1 - start) % (k - start)
                    // So the suffix repeatCount should be repeatCount[start + (n1
- start) % (k - start)] - prefixCount
                    int suffixCount = repeatCount[start + (n1 - start) % (k - start
)] - prefixCount;
                    return (prefixCount + patternCount + suffixCount) / n2;
                }
            }
        }
        return repeatCount[n1] / n2;
    }
};
```

Update 12/31/2018

- `int patternCount = (repeatCount[k] - repeatCount[start]) * (n1 - start) / (k - start);` to `int patternCount = (repeatCount[k] - repeatCount[start]) * ((n1 - start) / (k - start));` since `a * b / c` doesn't necessarily equal `a * (b / c)`. (the old test cases didn't cover this case)
- The size of `repeatCount` and `nextIndex` should be `n1 + 1`.
  Thanks for comments from @wxd_sjtu, @rjtsdl, @Rongch

Another version using `unordered_map` to save computation. Reduce runtime from ~80ms to ~4ms.

```
// OJ: https://leetcode.com/problems/count-the-repetitions/
// Author: github.com/lzl124631x
// Time: O(|s1| * n1) where |s1| is the length of s1
// Space: O(n1)
class Solution {
public:
    int getMaxRepetitions(string s1, int n1, string s2, int n2) {
        unordered_map<int, int> kToRepeatCount;
        unordered_map<int, int> nextIndexToK;
        kToRepeatCount[0] = 0;
        nextIndexToK[0] = 0;
        int j = 0, cnt = 0;
        for (int k = 1; k <= n1; ++k) {
            for (int i = 0; i < s1.size(); ++i) {
                if (s1[i] == s2[j]) {
                    ++j;
                    if (j == s2.size()) {
                        j = 0;
                        ++cnt;
                    }
                }
            }
            if (nextIndexToK.find(j) != nextIndexToK.end()) {
                int start = nextIndexToK[j];
                int prefixCount = kToRepeatCount[start];
                int patternCount = (n1 - start) / (k - start) * (cnt - prefixCount)
;
                int suffixCount = kToRepeatCount[start + (n1 - start) % (k - start)
] - prefixCount;
                return (prefixCount + patternCount + suffixCount) / n2;
            }
            kToRepeatCount[k] = cnt;
            nextIndexToK[j] = k;
        }
        return kToRepeatCount[n1] / n2;
```

```
    }
};
```

Comments: 17

Best Most Votes Newest to Oldest Oldest to Newest

Type comment here... (Markdown is supported)

Preview

Post

rjtsdl ★5

Last Edit: September 11, 2018 3:52 AM

Read More

@lzl124631x you get most of it right. Just this line

```
int patternCount = (repeatCount[k] - repeatCount[start]) *(n1 - start) / (k - start);
```

Correct to:

```
int patternCount = (n1 - start) / (k - start) * (repeatCount[k] - repeatCount[start]);
```

for example

3 * 3 / 2 (4) != 3/2 * 3 (3)

5

Show 1 reply

Reply

Share

Report


Rongch ★11

August 31, 2017 11:15 PM

Read More

There is an error in this line

```
int patternCount = (repeatCount[k] - repeatCount[start]) * (n1 - start) / (k - star
t);
```

It should calculate `(n1-start)/(k-start)` first

```
int patternCount = (repeatCount[k] - repeatCount[start]) * ((n1 - start) / (k - sta
rt));
```

5

Show 1 reply
Reply
Share
Report


zestypanda ★1699

July 17, 2017 9:50 AM

Read More

Great solution!
I have a minor optimization. In order to remove the inner loop to search for nextIndex[start], which is O(n^2)

and n is s2.size(), we can use a hash table visited[k]. It is used to record how many passes of s1 is required to get nextIndex of k. For example, visited[2] = p means after p passes of s1, the index in string s2 is 2. And the vector nextIndex is not necessary any more.
Also prefixCount + suffixCount = repeatCount[start + (n1 - start) % (k - start)], which can be simplified.

```
class Solution {
public:
    int getMaxRepetitions(string s1, int n1, string s2, int n2) {
        int m1 = s1.size(), m2 = s2.size();
        vector<int> repeated(m2+1,0), visited(m2+1, -1);
        visited[0] = 0;
        int k = 0, cnt = 0;
        for (int i = 1; i <= n1; i++) {
            for (int j = 0; j < m1; j++) {
                ] == s2[k]) {

                k == m2) {
                k = 0;
                cnt++;

                k] == -1) {
                repeated[i] = cnt;
                visited[k] = i;
            }
            else {
                int start = visited[k], p = i-start, t = cnt-repeated[start];
                int ans = (n1-start)/p*t + repeated[(n1-start)%p+start];
                return ans/n2;
            }
        }
        return cnt/n2;
    }
};
```

2
Show 1 reply
Reply
Share
Report

vesion ★ 215
December 30, 2016 5:36 AM
Read More

> Due to PigeonHole principle, you must find two same nextIndex after scanning s2.size() + 1 s1 segments.

Awesome! It's the key to take a good advantage of the repetitive pattern!

3
Reply
Share
Report


lufangjianle★129

May 10, 2017 7:38 AM

Read More

@lzl124631x That's a elegant solution! Actually, you can avoid the following loop by making nextIndex a map with (j, k) pair so that you can access the previous j directly. Remember to put(0, 0) in the beginning.

```
            for (int start = 0; start < k; ++start) {
                if (nextIndex[start] == j) {
                    int prefixCount = repeatCount[start];
                    int patternCount = (repeatCount[k] - repeatCount[start]) * (n1
- start) / (k - start);
                    int suffixCount = repeatCount[start + (n1 - start) % (k - start
)] - repeatCount[start];
                    return (prefixCount + patternCount + suffixCount) / n2;
1               }
            }
```

Show 1 reply

Reply

Share

Report


fill32★15

June 10, 2019 12:42 AM

Read More

peeeerfect!

0

Reply

Share

Report

contacttoakhil★530

Last Edit: May 26, 2019 12:22 AM

Read More

Same idea in Java (Better than 100 pct)

```
public int getMaxRepetitions(String s1, int n1, String s2, int n2) {
        int len1 = s1.length(), len2 = s2.length();
        int[] repeatCount = new int[len2 + 1];
        int[] nextIndex   = new int[len2 + 1];
        int j = 0, cnt = 0;
        for (int k = 1; k <= n1; k++){
            for (int i = 0; i < len1; i++){
                if (s1.charAt(i) == s2.charAt(j)){
                    j++;
                    if (j == len2){
                        j = 0;
                        cnt++;
                    }
                }
```

0

Reply

Share

Report

```
k] = cnt;
 = j;
rt = 0; start < k; start++){
Index[start] == j) {
prefixCount = repeatCount[start];
patternCount = (n1 - start) / (k - start) * (repeatCount[k]

suffixCount = repeatCount[start + (n1 - start) % (k-start)]
```

wxd_sjtu★928

Last Edit: December 30, 2018 10:00 PM

Read More

```
                int prefixCount = repeatCount[n1] - repeatCount[start];
                return (prefixCount + patternCount + suffixCount) / n2;
            }
        }
    }
    return repeatCount[n1] / n2;
}
```

Two errors in the solution, one of which is not mentioned in the comments before:

1. the size of `repeatCount` and `nextIndex2` should be `n1+1`, rather than `s2.size()+1`
2. `int patternCount = (repeatCount[k] - repeatCount[start]) * (n1 - start) / (k - start);` should be
   `int patternCount = (repeatCount[k] - repeatCount[start]) * ((n1 - start) / (k - start));`

0

Show 1 reply

Reply

Share

Report

wxd_sjtu ★928

December 30, 2018 5:49 AM

Read More

So smart solution!

0

Reply

Share

Report

Calotte★79

September 14, 2018 8:59 AM

Read More

I'm not understand that when you get the 'cnt', why not return cnt/n2 to finish?

0

Show 1 reply
Reply
Share
Report

-
- 1
- 2
-