Repeated String Match

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it. If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times (“abcdabcdabcd”), B is a substring of it; and B is not a substring of A repeated two times ("abcdabcd").

Note:
The length of `A` and `B` will be between 1 and 10000.

Solution 1

This is basically a modified version of string find, which does not stop at the end of A, but continue matching by looping through A.

```
int repeatedStringMatch(string A, string B) {
    for (auto i = 0, j = 0; i < A.size(); ++i) {
        for (j = 0; j < B.size() && A[(i + j) % A.size()] == B[j]; ++j);
        if (j == B.size()) return (i + j) / A.size() + ((i + j) % A.size() != 0 ? 1
: 0);
    }
    return -1;
}
```

As suggested by @k_j, I am also providing O(n + m) version that uses a prefix table (KMP). We first compute the prefix table using the suffix and prefix pointers. Then we are going through A only once, shifting B using the prefix table.

This solution requires O(n) extra memory for the prefix table, but it's the fastest out there (OJ runtime is 3 ms). However, we do not need extra memory to append A multiple times, as in many other solutions.

```
int repeatedStringMatch(string a, string b) {
    vector<int> prefTable(b.size());
    for (auto sp = 1, pp = 0; sp < b.size(); prefTable[sp++] = pp) {
        if (b[pp] == b[sp]) ++pp;
        else pp = prefTable[pp > 0 ? pp - 1 : 0];
    }
    for (auto i = 0, j = 0; i < a.size(); i += j + 1, j = prefTable[j > 0 ? j - 1 :
0]) {
        while (j < b.size() && a[(i + j) % a.size()] == b[j]) ++j;
        if (j == b.size()) return (i + j) / a.size() + ((i + j) % a.size() != 0 ? 1
: 0);
    }
    return -1;
}
```

written by votrubac original link here

Solution 2

Since LC has so many test cases missing, I wrote new code.
The idea is to keep string builder and appending until the length A is greater or equal to B.

```
 public int repeatedStringMatch(String A, String B) {

    int count = 0;
    StringBuilder sb = new StringBuilder();
    while (sb.length() < B.length()) {
        sb.append(A);
        count++;
    }
    if(sb.toString().contains(B)) return count;
    if(sb.append(A).toString().contains(B)) return ++count;
    return -1;
}
```

Here's the old idea I used which got accepted with multiple bugs.

~~Idea is to count all A in B as first step~~.
~~Then remove all A in B.~~
~~Then remaining B is either present in A or A+A.~~

```
public int repeatedStringMatch(String A, String B) {
    int i = 0, count = 0;
    while (i < B.length()) {
        int idx = B.indexOf(A, i);
        if (idx == -1) break;
        i = idx + A.length();
        count++;
    }
    B = B.replaceAll(A, ""); // remaining B if valid, should be smaller than A
    if (!B.isEmpty()) {
        if (A.startsWith(B)) count++; // B is substring AND first part of A
        else if(A.contains(B)) return -1; // B is substring somewhere in between
        else if ((A + A).contains(B)) count += 2; // B in rotating A
        else return -1;
    }
    return count;
}
```

written by wavy original link here

Solution 3

Let `n` be the answer, the minimum number of times `A` has to be repeated.

For `B` to be inside `A`, `A` has to be repeated sufficient times such that it is at least as long as `B` (or one more), hence we can conclude that the theoretical lower bound for the answer would be length of `B` / length of `A`.

Let `x` be the theoretical lower bound, which is `ceil(len(B) / len(A))`.

The answer `n` can only be `x` or `x + 1` (in the case where `len(B)` is a multiple of `len(A)` like in `A = "abcd"` and `B = "cdabcdab"`) and not more. Because if `B` is already in `A * n`, `B` is definitely in `A * (n + 1)`.

Hence we only need to check whether `B in A * x` or `B in A * (x + 1)`, and if both are not possible return -1.

By Yang Shun

Here's the cheeky two-liner suggested by @liping5:

```
class Solution(object):
    def repeatedStringMatch(self, A, B):
        t = -(-len(B) // len(A)) # Equal to ceil(len(b) / len(a))
        return t * (B in A * t) or (t + 1) * (B in A * (t + 1)) or -1
```

But don't do the above in interviews. Doing the following is more readable.

```
class Solution(object):
    def repeatedStringMatch(self, A, B):
        """
        :type A: str
        :type B: str
        :rtype: int
        """
        times = -(-len(B) // len(A)) # Equal to ceil(len(b) / len(a))
        for i in range(2):
          if B in (A * (times + i)):
            return times + i
        return -1
```

Thanks @ManuelP for suggesting that `times = int(math.ceil(float(len(B)) / len(A)))` can be written as `times = -(-len(B) // len(A))`.

written by yangshun original link here