Substring Search

OWNER: YUBO

PROBLEM STATEMENT

substring search: given a text of length n, a target string of length m, in an alphabet of size k, determine whether there is a substring of the text that matches the target string exactly

'hi my name is amelia', 'amelia' - true
'hi my name is amelia', 'foo bar baz' - false

Define complexity to be the number of character reads/comparisons
No library functions for string equality!

def isMatch(text, target):
""" returns True if target is a substring of text """
return False
print isMatch("hi my name is amelia", "amelia") # true
print isMatch("hi my name is amelia", "foo bar baz") # false

NOTES ON CALIBRATION

I usually have people start with the naive solution, which is roughly (with some Python syntactic sugar)

```
for(substring of length m in text):
    for(char1, char2 in zip(substring, target)):
        if(char1 != char2):
            break
    else:
        return True
return False
```

Compared to the naive solution: for(substring of length m in text) {substring ==? target), there are two main ways to speed up the comparison:
1) Hash substring comparison (inner loop), leads to Rabin-Karp, amortized O(n)
2) Don't check every substring (leads to Knuth-Moris-Pratt, O(n))

People who start thinking about hashing I usually push down route 1, with some hints that we need a **rolling** hash ("If we allowed transpositions how would you solve? Keep track of character count? Why is it fast to update character count?"). Usually with some pushing we can get to Rabin-Karp complete

People who start thinking about “oh, the naive is really bad at doing aabaabaabaabaaa vs aaa” I tend to push down the second track (e.g. “why can we skip straight to the third alignment for ’aazaaa’ vs ’aaa’? what if ’z’ is in the target string?”).

If a candidate catches onto the whole “suffix of substring is prefix of target string”, then I tell them to hard-code the KMP precomputation table, but usually we implement a strictly easier algorithm described in sample solution.

CANDIDATE EXAMPLES:

1 - Misses the point of the problem (e.g. talks about comparing character by character but uses string equality operators), is unable to get the naive solution working. Code looks nothing like proposed implementation
2 - Able to identify a single optimization on the naive solution but only barely scrapes by on implementing the naive solution. Lots of bugs, lots of branches. Clearly has trouble translating concepts to code.
3 - Is able to get one of the optimizations (Rabin Karp, smaller KMP) close to working, is generally pretty messy but shows an understanding of these algorithms
4 - Is able to write up one of Rabin Karp or sub-KMP to completion, with reasonably good style. Or is able to identify the KMP approach and nails the hard-coded solution.

SAMPLE INPUT

```
isMatch("aazaaa", "aaa") = true; // basic inspiration for KMP

isMatch("abababac", "ababac") = true; // counterexample for "start searching
wherever we fail"

// incidentally also counterexample for "skip to first occurrence of text char
that failed", we have to skip to the rfind

isMatch("abcbcd", "abcd") = false; // some people in KMP compute the correct
shift but only search forward not backwards

isMatch("hi my name is amelia", "amelia") = true; // note that the suffix of
"name" contains the start of "amelia"
```

SAMPLE SOLUTIONS (PYTHON)

sub-kmp is most popular, basically: suppose text substring and target have matched first j-1 characters but fail on the j-th, then align the j-th character of the text substring with the last occurrence of this character in the first j characters of the target, e.g.
text: abababac
target: ababac

we fail on the third ’b’ in the text, so the last occurrence of the b in the first six characters of the target is the second b in the target, so we next need to check alignment
text: abababac
target: __ababac

```
# def is_match(text, target):
#     """ rabin-karp """
#     BASE=101
#     def is_equal(str1, str2):
#         assert len(str1) == len(str2)
#         for idx in range(len(str1)):
#             if str1[idx] != str2[idx]:
#                 return False
#         return True

#     def hash_rk(substr):
#         """ rabin-karp substring hash. python no overflow problems """
#         if len(substr) == 1:
#             return ord(substr)
#         return ord(substr[-1]) + BASE * hash_rk(substr[:-1])

#     def update_hash(val, len_str, char_pop, char_push):
#         return BASE * (val - (ord(char_pop) ** len_str)) + ord(char_push)

#     n = len(text)
#     m = len(target)
#     target_hash = hash_rk(target)
#     text_hash = hash_rk(text[:m])
#     for text_idx in range(n - m):
#         if text_hash == target_hash and \
#             is_equal(target, text[text_idx: text_idx + m]):
#             return True
#         update_hash(text_hash, m, text[text_idx], text[text_idx + m])
#     return is_equal(target, text[n - m:n])

# def is_match(text, target):
#     """sub-KMP"""
```

```
#     n = len(text)
#     m = len(target)
#     i=0
#     while i < n - m + 1:
#         j=0
#         while j < m:
#             if text[i + j] != target[j]:
#                 break
#             j += 1
#         if j == m:
#             return True
#         i += j - target[:j].rfind(text[i + j])
#     return False

# def is_match(text, target):
#     """ hardcoded KMP for target='ababac' """
#     jump_table = [
#         { "a": 1 }, # if we see a, next compare target[1]
#         { "b": 2, "a": 1 }, # b => next compare target[2], but if a then can next compare target[1]
#         { "a": 3 },
#         { "b": 4, "a": 1 },
#         { "a": 5 },
#         { "c": 6, "b": 4, "a": 1} # c => completion, b => 'abab' so compare target[4],
#     ]
#     curr_index = 0
#     m = len(target)
#     for c in text:
#         curr_index = jump_table[curr_index].get(c, 0)
#         if curr_index == m:
#             return True
#     return False
```

```
print(is_match(”aazaaa”, ”aaa”))
print(is_match(”abcbcd”, ”abcd”))
print(is_match(”hi my name is amelia”, ”amelia”))
print(is_match(”abababac”, ”ababac”))
print(is_match(”foo bar baz”, ”ababac”))# def is_match(text, target):
#     """ naive """
#     n = len(text)
#     m = len(target)
#     for i in range(n - m + 1):
#         for j in range(m):
#             if text[i + j] != target[j]:
#                 break
#         else: # for-else means if for loop runs to completion
#             return True
#     return False

# def is_match(text, target):
#     """ rabin-karp """
#     BASE=101
#     def is_equal(str1, str2):
#         assert len(str1) == len(str2)
#         for idx in range(len(str1)):
#             if str1[idx] != str2[idx]:
#                 return False
#         return True

#     def hash_rk(substr):
#         """ rabin-karp substring hash. python no overflow problems """
#         if len(substr) == 1:
#             return ord(substr)
#         return ord(substr[-1]) + BASE * hash_rk(substr[:-1])

#     def update_hash(val, len_str, char_pop, char_push):
```

```
#         return BASE * (val - (ord(char_pop) ** len_str)) + ord(char_push)

#     n = len(text)
#     m = len(target)
#     target_hash = hash_rk(target)
#     text_hash = hash_rk(text[:m])
#     for text_idx in range(n - m):
#         if text_hash == target_hash and \
#             is_equal(target, text[text_idx: text_idx + m]):
#             return True
#         update_hash(text_hash, m, text[text_idx], text[text_idx + m])
#     return is_equal(target, text[n - m:n])

# def is_match(text, target):
#     """sub-KMP"""
#     n = len(text)
#     m = len(target)
#     i=0
#     while i < n - m + 1:
#         j=0
#         while j < m:
#             if text[i + j] != target[j]:
#                 break
#             j += 1
#         if j == m:
#             return True
#         i += j - target[:j].rfind(text[i + j])
#     return False

# def is_match(text, target):
#     """ hardcoded KMP for target='ababac' """
#     jump_table = [
#         { "a": 1 }, # if we see a, next compare target[1]
```

```
#         { "b": 2, "a": 1 }, # b => next compare target[2], but if a then can
next compare target[1]
#         { "a": 3 },
#         { "b": 4, "a": 1 },
#         { "a": 5 },
#         { "c": 6, "b": 4, "a": 1} # c => completion, b => 'abab' so compare
target[4],
#     ]
#     curr_index = 0
#     m = len(target)
#     for c in text:
#         curr_index = jump_table[curr_index].get(c, 0)
#         if curr_index == m:
#             return True
#     return False

print(is_match("aazaaa", "aaa"))
print(is_match("abcbcd", "abcd"))
print(is_match("hi my name is amelia", "amelia"))
print(is_match("abababac", "ababac"))
print(is_match("foo bar baz", "ababac"))
```