Code Jam 2015 - Qualification Round

Analysis: Dijkstra

We can solve this problem with the help of the following two insights. First, the product of the whole string must equal -1 (*i* â¨‰ *j* â¨‰ *k*). Second, we only need a small number of copies of the input string to check for a solution. One way to do this is by finding the shortest prefix of the whole string that reduces to *i* and the shortest subsequent prefix that reduces to *j*. If we find such two substrings, there is a solution because the rest of the string reduces to *k* thanks to the associative property. The rest of this editorial explains the two insights and provides a sample implementation.

No solution if the whole string cannot be reduced to -1

Suppose the string **S** is the whole concatenated string formed by repeating the given input string **X** times. If we can break the string S into three non-empty substrings **A**, **B**, **C** where A + B + C = S such that A reduces to *i*, B reduces to *j*, and C reduces to *k*, then the string S reduces to string "ijk", which then reduces to -1. Therefore, if the string S cannot be reduced to -1, then there is no solution. Reducing the string S can be done by simply multiplying all the characters in S into one value.

If the string S can be reduced to -1, it doesn't mean that there is a solution for S. There are many strings (e.g., "ii", "jj", etc.) that reduces to -1 but do not form a concatenation of three substrings that reduce to *i*, *j*, and *k*, respectively.

The first two substrings must reduce to *i* and *j*, respectively

From now on, we only consider whole strings that reduce to -1. To determine whether a string **S** can be broken into three non-empty substrings **A**, **B**, **C** where each reduces to *i*, *j*, *k* respectively, we only need to find the first two substrings. The last substring is guaranteed to reduce to *k* since the whole string reduces to -1.

(There are other alternatives. For example, we can find the shortest prefix and the shortest suffix that reduce to *i* and *k*, respectively. If the prefix does not overlap with the suffix, then we can reduce the rest to *j*. Care must be taken as the multiplication operator is not commutative. Exercise: Can the prefix-suffix pair actually overlap while the whole string can still be reduced to â€˜ijkâ€™? Answer: No.)

To find the first substring, we start from the first character of the string S and start multiplying it with the next characters and so on until we get the value *i*. Afterward, we repeat the procedure from the current position until we get the value *j*. The rest of the string is guaranteed to be non-empty and reduce to *k*. The complexity of finding the first and the second substrings is O(L * X), since we need to scan the whole string S of length L * X.

With these insights, we can solve the small input in O(L * X). Below is a sample implementation in Python:

```
M = [[ 0,  0,  0,  0,  0 ],
     [ 0,  1,  2,  3,  4 ],
     [ 0,  2, -1,  4, -3 ],
     [ 0,  3, -4, -1,  2 ],
     [ 0,  4,  3, -2, -1 ]]

def mul(a, b):
```

```
  sign = 1 if a * b > 0 else -1
  return sign * M[abs(a)][abs(b)]

def multiply_all(S, L, X):
  value = 1
  for i in range(X):
    for j in range(L):
      value = mul(value, S[j])
  return value

def construct_first_two(S, L, X):
  i_value = 1
  j_value = 1
  for i in range(X):
    for j in range(L):
      if i_value != 2:
        i_value = mul(i_value, S[j])
      elif j_value != 3:
        j_value = mul(j_value, S[j])
  return i_value == 2 and j_value == 3

for tc in range(input()):
  L, X = map(int, raw_input().split())
  # maps 'i' => 2, 'j' => 3, 'k' => 4
  S = [(ord(v) - ord('i') + 2) for v in raw_input()]
  ok1 = multiply_all(S, L, X) == -1
  ok2 = construct_first_two(S, L, X)
  print "Case #%d: %s" % (tc + 1,
    "YES" if ok1 and ok2 else "NO")
```

The multiplication matrix M is defined as a 5 x 5 matrix where the first column and the first row are not used (for value 0). The second row and the second column is for an identity value 1. The third, fourth and fifth (rows and columns) represent *i*, *j*, and *k*, respectively, identical to the quaternion multiplication matrix.

Optimizations for the large input

The maximum whole string length is 10^16 which is too large for an implementation of the algorithm above to finish within the time limit when it is executed in a today's computer. We need to optimize both of these functions: multiply_all() and construct_first_two().

Optimizing the multiply_all() method

Observe that the whole string is formed by repeating the input string (of length L) X times, giving the complexity of O(L * X). Since the multiplication operator is associative, we can reduce the input string into a single value before multiplying this value to itself X - 1 times. Thus, we can rewrite the multiply_all() method as follows:

```
def multiply_all(S, L, X):
  value = 1
  for i in range(L):
    value = mul(value, S[i])
  return power(value, X) # computes value^X
```

To quickly multiply the value with itself X - 1 times (that is, computing value^X), we can use the exponentiation by squaring technique which runs in O(log(X)):

```
def power(a, n):
  if n == 1: return a
  if n % 2 == 0: return power(mul(a, a), n // 2)
  return mul(a, power(mul(a, a), (n - 1) // 2))
```

With this optimizations, the multiply_all() complexity is now O(L + log(X)). Since L is at most 10000 and X is at most 10^12, the number of multiplication operations is at most 10040.

We can improve it further to O(L). Observe that the multiplication values (to itself) will always repeat to the original value after 4 consecutive multiplications, thus we only need to do at most X mod 4 multiplications to compute value^X:

```
def power(a, n):
  value = 1
  for i in range(n % 4):
    value = mul(value, a)
  return value
```

Optimizing the call to construct_first_two()

To find the prefix, we start with an identity value 1 and multiply it with the value of the first position of the input string, and so on until we get a value *i*. Supposing X is sufficiently large, if we reach the end of the input string and haven't obtained the value *i*, we repeat this procedure for the next copy of the input string. At this point, the current value may be 1, *j*, *k*, -1, *-i*, *-j*, or *-k*. If the current value is 1, we may as well stop here and declare no solution because continuing the multiplication will result in the same value 1 again. However, if the current value is not 1, we can continue the procedure.

We know from the previous section that multiplying a value to itself will repeat to its original value every 4 consecutive multiplication. Thus, if we don't encounter the value *i* after executing the procedure for 4 copies of the input string, it is impossible to construct the desired prefix. If we do encounter value *i*, then we can proceed to find the second substring where the same insight applies.

Thus, we can safely limit the call to construct_first_two() from X repeats:

```
  ok2 = construct_first_two(S, L, X)
```

to min(8, X) repeats:

```
  ok2 = construct_first_two(S, L, min(8, X))
```

This reduces the complexity of the construct_first_two() method from O(L * X) to O(L).

Therefore, the complexity of the overall algorithm is now O(L) per test case.