Code Jam 2015 - Round 1C

Analysis: Typewriter Monkey

This problem naturally has two parts — computing the maximum number of bananas needed, and computing the expected number of bananas needed.

Take, for example, the target string X = ABACABA. To find a string of length S containing the maximum number of copies of X, we start by putting X at the start of the string. Then to fit as many more copies as possible, we want to overlap each copy of X as much as possible with the previous copy. For this example, we could overlap with the final "A" and add "BACABA", but it is even better to overlap with "ABA" and add just "CABA" to get a second copy. To find the maximum amount of overlap, we can just try every possible amount and check which ones work, since L is small. It can also be computed in linear time using the initialization phase of the Knuth-Morris-Pratt algorithm. If the maximum amount of overlap is O, then we can fit 1+(S-L)/(L-O) copies of the string.

To find the expected number of copies, we start by computing the probability P of the word occurring at a fixed place. This is equal to the product of the probabilities for each letter of the word being correct. The probability for a single letter being correct is the fraction of keys which are that letter.

By linearity of expectation, the expected number of copies is then just P multiplied by the number of places the string can occur, which is S-L+1. This is a convenient fact to use, because we don't need to take into account that the string occurring in one position and the string occurring in an overlapping position are not independent events.

Sample implementation in Python:

```
# Find the maximum amount of overlap. We can just try
# every possible amount and check which ones work.
def max_overlap(t):
  for i in range(1, len(t)):
    if t[i:] == t[0:len(t)-i]:
      return len(t) - i
  return 0

# Returns the probability of the target word
# occurring at a fixed place.
def probability(target, keyboard):
  P = 1.0
  # Compute the product of the probabilities
  # for each letter of the word being correct.
  for i in range(len(target)):
    # The probability for a single letter being correct
    # is the fraction of keys which are that letter.
    C = keyboard.count(target[i])
    P = P * C / len(keyboard);
  return P

for tc in range(input()):
  K, L, S = map(int, raw_input().split(' '))
  keyboard = raw_input()
  target = raw_input()
  res = 0
```

```
  P = probability(target, keyboard)
  if P > 0:
    O = max_overlap(target)
    max_copies = 1.0 + (S-L) / (L-O)
    min_copies = P * (S-L+1)
    res = max_copies - min_copies
  print("Case #%d: %f" % (tc + 1, res))
```

Klockan wrote a solution in C++ that also uses this approach, which you can download from the scoreboard.

We could also use a dynamic programming algorithm for both parts of the problem, using O(LS) states, where the state is the number of characters typed and the largest number of characters of the word that are currently matched, and the value at each state is the probability of that state and the maximum number of copies of the word that could have been produced while reaching it. For each of the states where the entire word has just been matched, we add the probability of reaching that state to the expected number of copies of the word, and update the maximum number of copies that are possible. linguo wrote a solution of this type in Python.