<https://leetcode.com/problems/substring-with-largest-variance/description/>

The **variance** of a string is defined as the largest difference between the number of occurrences of **any** 2 characters present in the string. Note the two characters may or may not be the same.

Given a string s consisting of lowercase English letters only, return **the largest variance possible among all substrings of** s.

A **substring** is a contiguous sequence of characters within a string.

**Example 1:**

**Input:** s = "aababbb"

**Output:** 3

**Explanation:**

All possible variances along with their respective substrings are listed below:

- Variance 0 for substrings "a", "aa", "ab", "abab", "aababb", "ba", "b", "bb", and "bbb".

- Variance 1 for substrings "aab", "aba", "abb", "aabab", "ababb", "aababbb", and "bab".

- Variance 2 for substrings "aaba", "ababbb", "abbb", and "babb".

- Variance 3 for substring "babbb".Since the largest possible variance is 3, we return it.

**Example 2:**

**Input:** s = "abcde"

**Output:** 0

**Explanation:**

No letter occurs more than once in s, so the variance of every substring is 0.

**Constraints:**

1 <= s.length <= 10^4

s consists of lowercase English letters.

**Attempt 1: 2024-06-08**

**Solution 1: DP + Kadane Alogrithm (180 min)**

**Style 1:**

class Solution {

    public int largestVariance(String s) {

        int maxVariance = 0;

        Set<Character> uniqueChars = new HashSet<>();

        for (char c : s.toCharArray()) {

            uniqueChars.add(c);

        }

        List<Character> unique = new ArrayList<>(uniqueChars);

        for (char a : unique) {

            for (char b : unique) {

                if (a == b) continue;

                int aAgainstBVar = solveOne(a, b, s);

                maxVariance = Math.max(aAgainstBVar, maxVariance);

            }

        }

        return maxVariance;

    }

    public int solveOne(char a, char b, String string) {

        int maxVar = 0;

        int var = 0;

        boolean hasB = false;

        // First element of the currently considered subarray is b

        boolean firstB = false;

        for (char c : string.toCharArray()) {

            if (c == a) {

                var += 1;

            } else if (c == b) {

                hasB = true;

                if (firstB && var >= 0) {

                    // "shift right" and save a 1 in the current sum to be able

                    // to properly maximize it

                    // We can only do this when we know that we have a `b` at

                    // the start of our current subarray, and we'll only ever

                    // have a single b at the start, always followed by an a,

                    // due to the next rule

                    firstB = false;

                } else if ((var - 1) < 0) {

                    // Restart the subarray from this b (inclusive) onwards

                    // This rule ensures we skip double-b's, every subarray

                    // will always end up being `ba....`, `[bb]a` would become

                    // `b[b]a` -> `b[ba]`

                    firstB = true;

                    var = -1;

                } else {

                    // var will always be >= 0 in this case

                    var -= 1;

                }

            }

            if (hasB && var > maxVar) {

                maxVar = var;

            }

        }

        return maxVar;

    }

}

**Refer to**

**"Weird Kadane" | Intuition + Solution Explained**

<https://leetcode.com/problems/substring-with-largest-variance/solutions/2579146/weird-kadane-intuition-solution-explained/>

The Solution

We're trying to solve the problem by just taking two individual letters a and b from our allowed alphabet as per the question rules: "s consists of lowercase English letters", and solving the problem for a against b. We will also end up solving it for b against a due to the loop. The final solution is the max of all the possible letters in the string solved against each other.

Idea being, you're solving:

max\_variance = 0

unique = list(set(s))

for a in unique: # see below

for b in unique: # rules dictate these 2 loops would be no more than O(26\*26)

if a == b: continue

a\_against\_b\_var = solveOne(a, b, s) # <= you want to do this efficiently

max\_variance = max(a\_against\_b\_var, max\_variance)

return max\_variance

Now you want to solve one such problem solveOne efficiently.

A naive solution for solving a against b and b against a simultaneously is a O(N^2) loop tracking the -1 and +1 as explained in the hint, and taking the abs value of (countA - countB).

But you can see that what you're really trying to do, is maximise the count of a against b and b against a for both cases. This is really just [L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA), which you can do in O(N) (so 2\*O(N) for both a against b and b against a: much better than O(N^2)), with a twist:

You can only consider your current running subarray as a valid solution if you were considering at least one b (= one -1) inside the solution. Solutions composed of just a are not valid. So you must ensure that you have bs, but you must also minimise their "impact" on the running sum.

You start with a solution not considering these bonus rules. Just basic Kadane's algorithm for the as against the bs:

def solveOne(a, b, string):

max\_var = 0

var = 0

for c in string:

if c == a:

var += 1

elif c == b:

var -= 1

var = max(0, var)

max\_var = max(max\_var, var)

return max\_var

Then you add the simple rule of "you must have a b in the considered substring"

def solveOne(a, b, string):

max\_var = 0

var = 0

has\_b = False

for c in string:

if c == a:

var += 1

elif c == b:

has\_b = True

var -= 1

var = max(0, var)

if has\_b:

max\_var = max(max\_var, var)

return max\_var

Now for the more complex rules:

Consider "baaaa". The correct solution is 3. The above algoritm would give us results:

+----------+---+---+---+---+---+

| letter | b | a | a | a | a |

+----------+---+---+---+---+---+

| variance | 0 | 1 | 2 | 3 | 4 |

+----------+---+---+---+---+---+

This is an issue, so we must ensure we track the b properly. So we can't be resetting to 0 immediately as you do in the typical Kadane algorithm.

Let's say we allow var to go to -1 (at least initially), so that seeing an a after a b will reset us back to 0 (which would be a correct result for the string "ba").

**Now what did the max(0, var) in Kadane's algorithm really mean?**

**Well, we were discarding a subarray and restarting at/after the current array item because the subarray up until the discarded number was no longer "saveable"; we would not get the max possible sum by continuing to add numbers to that running sum. Restarting from 0 with the other/remaining half could still give us a better result.**

Let's say we kept the -1 for the b's without resetting to 0, and consider:

+----------+----+---+---+---+---+

| letter | b | b | a | a | a |

+----------+----+---+---+---+---+

| variance | -1 | ? | ? | ? | ? |

+----------+----+---+---+---+---+

We're considering the subarray "[b]baaa" and start that off with a decrement. Do we want "[bb]aaa" to be considered in its entirety?

Well, bb would not give us the max sum of as if we kept both bs and kept going trying to look for as, similar to going to a negative value in the original Kadane's algorithm.

**E.g. "[bba]aa" would give us -1, "[bbaa]a" would give us 0, but "b[baa]a" would give us 1. Removing the first of those "b"s obviously doesn't break the rule of "all substrings must have at least one b" and maximises the sum for a.**

**So we want to cut off/restart our subarray start each time we get to -1 in this implementation. Such that in the example we'd be considering:**

**"[b]baaa" -> "[bb]aaa" becomes "b[b]aaa" -> "b[ba]aa" -> "b[baa]a" -> "b[baaa]" (this gives us the max!)**

+----------+----+----+---+---+---+

| letter | b | b | a | a | a |

+----------+----+----+---+---+---+

| variance | -1 | -1 | 0 | 1 | 2 |

+----------+----+----+---+---+---+

**To summarise up until now:**

**if we have a positive var value and we see an a or a b, we just increment/decrement accordingly**

**we only consider a var value towards our maximum from the point we've seen at least one b and that b is contained in our subarray**

**-1 indicates the start of a new subarray, if we get a var value under -1, we reset var to -1 which means we restart the subarray from that point**

What if we had some as before we get another b like in `"baabbbaaaaa"

+----+---+---+---+---+---+---+---+---+---+---+

| b | a | a | b | b | b | a | a | a | a | a |

+----+---+---+---+---+---+---+---+---+---+---+

| -1 | 0 | 1 | ? | | | | | | | |

+----+---+---+---+---+---+---+---+---+---+---+

Well, observe that we wouldn't get the max value by considering "[baab]bbbaaaaa" (var = 0), rather "b[aab]bbbaaaaa" (var = 1) would be more optimal and continue following the rule of having to have at least one b.

It's pretty easy to say that if we knew we had a b as our first subarray member, then when we see our next b assuming we had some a's, we benefit by simply shifting our subarray range to the right by one, which results in the var value remaining unchanged, but us having the ability to extend into a larger sum with subsequent characters.

We can also know when we have such a b at the start of our range, because as we just saw it's whenever we restart the subarray: whenever we get to var <= -1.

Should this rule take priority over the subarray restart rule? Throwback to the previous example: "[b]baaaa" -> "[bb]aaaa" becomes "b[b]aaaa" which kind of a special case of this idea, but we'd like to be immediately aware that our first subarray member is b here so that we could apply the rule again later.

So in general, the main reason to do a right shift is when we are considering a non -1 valued subarray (= not a single item that is a b) to preserve the running sum.

So we'd like to have:

"[b]aabbbaaaaa" (-1) -> "[ba]abbbaaaaa" (0) -> "[baa]bbbaaaaa" (1) -> "[baab]bbaaaaa" would be 0 but we shift right to get "b[aab]bbaaaaa" (1) -> "b[aabb]baaaaa" (0) -> "b[aabbb]aaaaa" (-1) leading to a reset starting with that last b: "baabb[b]aaaaa" (-1) -> "baabb[ba]aaaa" (0) -> "baabb[baa]aaa" (1) -> "baabb[baaa]aa" (2) -> "baabb[baaaa]a" (3) -> "baabb[baaaaa]" (4)

+----+---+---+---+---+----+---+---+---+---+---+

| b | a | a | b | b | b | a | a | a | a | a |

+----+---+---+---+---+----+---+---+---+---+---+

| -1 | 0 | 1 | 1 | 0 | -1 | 0 | 1 | 2 | 3 | 4 |

+----+---+---+---+---+----+---+---+---+---+---+

**Final Rules**

The rules we've found are, for any given char c being considered at any position in the string:

**if we have any var value and we see that the current c == a we can just increment. There's no special rules relating to incrementing**

**if we see the current c == b:**

**if we see that we have a non -1 var value (indicating a current substring containing not just b's) and assuming we could "shift right by one" to save our current var value, then we will do so. We would need to track if we do actually have a b at the start of our current substring to be able to do so.**

**if for the new var value, var - 1, (var - 1) <= -1, we set it to -1 exactly and restart the substring from the current b. We now have a first b for the previous rule to use.**

**otherwise the var value is currently positive, and adding the b would set var to 0 (minimum), not ever below 0.**

Final Code

This gets us to @votrubac's code, or here the python version of it:

def solveOne(a, b, string):

max\_var = 0

var = 0

has\_b = False

first\_b = False # first element of the currently considered subarray is b

for c in string:

if c == a:

var += 1

elif c == b:

has\_b = True

if first\_b and var >= 0: # "shift right" and save a 1 in the current sum to be able to properly maximise it

# we can only do this when we know that we have a `b` at the start of our current subarray, and we'll only ever have a single b at the start

# always followed by an a, due to the next rule

first\_b = False

elif (var - 1) < 0: # restart the subarray from this b (inclusive) onwards

# this rule ensures we skip double-b's, every subarray will always end up being `ba....`, `[bb]a` would become `b[b]a` -> `b[ba]`

first\_b = True

var = -1

else:

var -= 1 # var will always be >= 0 in this case

if has\_b and var > max\_var:

max\_var = var

return max\_var

**Style 2:**

class Solution {

    public int largestVariance(String s) {

        // Length of the input string.

        int length = s.length();

        // Variable to store the maximum variance found so far.

        int maxVariance = 0;

        // Iterate over all possible pairs of characters (a and b are different).

        for (char firstChar = 'a'; firstChar <= 'z'; ++firstChar) {

            for (char secondChar = 'a'; secondChar <= 'z'; ++secondChar) {

                if (firstChar == secondChar) {

                    // If both characters are the same, skip this iteration.

                    continue;

                }

                // Array to keep track of the frequency of character 'a'

                // f[0] is the streak of 'a's, f[1] is the max variance for the current window.

                // Initialize with 0 for streak and -n for variance because variance cannot be less than -n.

                int[] frequency = new int[] {0, -length};

                // Iterate over each character in the string.

                for (int i = 0; i < length; ++i) {

                    if (s.charAt(i) == firstChar) {

                        // If the current character is 'a', increase both frequencies.

                        frequency[0]++;

                        frequency[1]++;

                    } else if (s.charAt(i) == secondChar) {

                        // If the current character is 'b', calculate the variance.

                        frequency[1] = Math.max(frequency[0] - 1, frequency[1] - 1);

                        // Reset the streak of 'a's because 'b' is encountered.

                        frequency[0] = 0;

                    }

                    // Update the maximum variance found.

                    maxVariance = Math.max(maxVariance, frequency[1]);

                }

            }

        }

        // Return the largest variance found.

        return maxVariance;

    }

}

**Refer to**

<https://algo.monster/liteproblems/2272>

**Problem Description**

The problem focuses on finding the largest variance within any substring of a given string s, which contains only lowercase English letters. The variance of a string is defined as the largest difference in the number of occurrences between any two distinct characters in it. It's important to consider that the characters could be the same or different.

To clarify, let's say we're given a string like "abcde". In this string, each character appears exactly once, so the variance between any two characters is zero. However, if we had a string like "aabbcc", the variance could be higher. If we choose the substring "aab", the variance would be 2 because there are two 'a' characters and none of the 'b' character (or vice versa). We need to consider all possible substrings to find the one with the largest variance.

**Intuition**

The intuition behind the solution lies in realizing that the variance could only be maximized if we consider pairs of different characters in the string, as those pairs are the ones that can actually have a non-zero variance. So the first step is to iterate over all possible pairs of characters (permutations of two distinct letters from the 26 lowercase English characters).

Once a pair of characters is selected (let's say 'a' and 'b'), we apply a two-pass scanning algorithm on the string to track the difference in the occurrence counts between 'a' and 'b'. We keep two counters: one (f[0]) resets every time we encounter an 'a' after a 'b', while the other (f[1]) continues to accumulate or starts over from f[0] - 1 when a 'b' is encountered after at least one 'a'. We look to maximize the difference (f[1]), which is essentially the variance, while scanning.

In the end, we take the maximum recorded value during all scans as the answer. This approach works because by focusing on pairs of characters at a time, we are able to isolate the potential substrings that can provide the maximum variance, rather than getting lost in the overwhelming number of all possible substrings.

Along with tracking the occurrences of 'a' and 'b', there's also an understanding that we include a subtraction operation when we encounter a 'b', making sure we are actually counting a valid substring with at least one 'a' when 'b' appears, since variance requires both characters to be present. A -inf is used to handle cases where a valid substring does not exist yet. By considering every single position in the string as a potential starting or ending point for substrings with maximum variance between two specific characters, the algorithm guarantees that no possible substring is overlooked, thus ensuring the correctness of the solution.

**Solution Approach**

The implementation employs a brute force strategy enhanced with a smart iteration mechanism and memory optimization to calculate the maximum variance for all substrings of the given string. The key elements of this approach involve permutations, two comparison variables, and a linear scan through the string.

**Permutations of Characters**

Firstly, the algorithm generates all possible permutations of two distinct characters from the set of lowercase English letters. This is done using permutations(ascii\_lowercase, 2) which loops through all pairs of two distinct characters since we're looking for the difference in occurrences between two potentially different characters.

**Tracking Variance with Two Counters**

In the solution, an array f with two elements is used:

f[0] serves as a resettable counter for occurrences of character a since the last occurrence of b.

f[1] keeps track of the accumulated variance between a and b.

This dichotomy allows the algorithm to keep track of two scenarios simultaneously:

Accumulating occurrences of a when no b has interfered (handled by f[0]).

Capturing the 'variance' until a reset is needed due to a b occurrence (handled by f[1]).

Both f[0] and f[1] are reset to zero if a b appears directly after a b. However, when an a is followed by a b, f[1] takes the role of capturing the current variance by taking either the current accumulated variance minus one, or the variance just after the last a minus one, determined by max(f[1] - 1, f[0] - 1).

This handling ensures that at least one a is present before a b is counted, which is necessary for a valid substring.

**Maximal Variance Calculation**

As we scan the string s, for each character c, the algorithm performs the following actions:

If c is a, we increment both f[0] and f[1].

If c is b, we decrement f[1] and choose the larger of the two: f[1] or f[0] - 1, effectively either continuing with the current substring or starting a new substring that excludes the last a. f[0] is reset to zero as we can't start a new substring with just b.

After each update, we check if f[1] exceeds the current ans. If it does, we update ans to the value of f[1]. The algorithm avoids false starts with -inf, ensuring that f[1] only records valid substrings.

The loop through permutations of characters ensures that we consider every potential character disparity. The nested loop through the string ensures that we consider every potential substring for each character pair. Eventually, ans will contain the maximum variance achievable among all substrings of s.

The space complexity of the solution is O(1) as we only maintain a few variables and a pair of counters, and the time complexity is O(26**26**n) where n is the length of the string, due to the nested loops—26 for each character in the permutations, and n for the iterations over the string.

**Example Walkthrough**

Let's illustrate the solution approach using a small example. Suppose we are given the string s = "abbab". Our task is to find the largest variance within any substring of s.

Firstly, we generate all possible permutations of two distinct characters from the set of lowercase English letters. For simplicity, let's consider only a few pairs here, such as ('a', 'b') and ('b', 'a'), since these are the only characters in our example string.

We initiate a variable ans to store the maximum variance found. Initially, ans = -inf.

**Pair ('a', 'b'):**

Initialize the counters f[0] and f[1] to 0.

Loop through each character c in s = "abbab":

For c = 'a': Increment both f[0] and f[1] (since we initially consider the occurrence of 'a' without 'b' involved).

For c = 'b': Decrement f[1] and then set f[1] to the maximum of f[1] and f[0] - 1. Reset f[0] to 0.

Let's go through the string "abbab":

Start with 'a': f[0] = 1, f[1] = 1.

Next, 'b': f[1] = max(f[1] - 1, f[0] - 1) = 0, f[0] = 0. Update ans = max(ans, f[1]) = 1.

Next, 'b': f[1] = max(f[1] - 1, f[0] - 1) = -1, since there are no 'a' in front, we don't update ans.

Next, 'a': f[0] = 1, f[1] = max(f[1] + 1, f[0]) = 1.

Last, 'b': f[1] = max(f[1] - 1, f[0] - 1) = 0. Update ans = max(ans, f[1]) = 1.

**Pair ('b', 'a'):**

Following the same steps but reversing the roles of 'a' and 'b', we would not find a variance larger than what we already have from the pair ('a', 'b').

After considering these permutations, we find that the maximum variance is 1, which we may have obtained from the substring "ab" or "ba" in the string "abbab".

In a complete implementation, we would evaluate all permutations of character pairs, but in this small example with only 'a' and 'b' present, it is unnecessary.

The process effectively scans through every substring of s and calculates the variance for pairs of characters, updating ans when a higher variance is found. The time complexity is mainly driven by the nested loops: there are 26**26 permutations of character pairs, and for each pair, we scan the entire string s of length n, resulting in an O(26**26\*n) time complexity.

**Solution Implementation**

class Solution {

public int largestVariance(String s) {

// Length of the input string.

int length = s.length();

// Variable to store the maximum variance found so far.

int maxVariance = 0;

// Iterate over all possible pairs of characters (a and b are different).

for (char firstChar = 'a'; firstChar <= 'z'; ++firstChar) {

for (char secondChar = 'a'; secondChar <= 'z'; ++secondChar) {

if (firstChar == secondChar) {

// If both characters are the same, skip this iteration.

continue;

}

// Array to keep track of the frequency of character 'a'

// f[0] is the streak of 'a's, f[1] is the max variance for the current window.

// Initialize with 0 for streak and -n for variance because variance cannot be less than -n.

int[] frequency = new int[] {0, -length};

// Iterate over each character in the string.

for (int i = 0; i < length; ++i) {

if (s.charAt(i) == firstChar) {

// If the current character is 'a', increase both frequencies.

frequency[0]++;

frequency[1]++;

} else if (s.charAt(i) == secondChar) {

// If the current character is 'b', calculate the variance.

frequency[1] = Math.max(frequency[0] - 1, frequency[1] - 1);

// Reset the streak of 'a's because 'b' is encountered.

frequency[0] = 0;

}

// Update the maximum variance found.

maxVariance = Math.max(maxVariance, frequency[1]);

}

}

}

// Return the largest variance found.

return maxVariance;

}

}

**Refer to**

[L53.Maximum Subarray (Ref.L821)](note://872ABD256AD34054B5D1F2E9992E2CAA)