1427. Performing String Shifts

April 14, 2020 | 19.4K views

Average Rating: 5 (10 votes)

You are given a string `s` containing lowercase English letters, and a matrix `shift`, where `shift[i] = [direction, amount]`:

- `direction` can be `0` (for left shift) or `1` (for right shift).
- `amount` is the amount by which string `s` is to be shifted.
- A left shift by 1 means remove the first character of `s` and append it to the end.
- Similarly, a right shift by 1 means remove the last character of `s` and add it to the beginning.

Return the final string after all operations.

Example 1:

```
Input: s = "abc", shift = [[0,1],[1,2]]
Output: "cab"
Explanation:
[0,1] means shift to left by 1. "abc" -> "bca"
[1,2] means shift to right by 2. "bca" -> "cab"
```

Example 2:

```
Input: s = "abcdefg", shift = [[1,1],[1,1],[0,2],[1,3]]
Output: "efgabcd"
Explanation:
[1,1] means shift to right by 1. "abcdefg" -> "gabcdef"
[1,1] means shift to right by 1. "gabcdef" -> "fgabcde"
[0,2] means shift to left by 2. "fgabcde" -> "abcdefg"
[1,3] means shift to right by 3. "abcdefg" -> "efgabcd"
```

Constraints:

- `1 <= s.length <= 100`
- `s` only contains lower case English letters.
- `1 <= shift.length <= 100`
- `shift[i].length == 2`
- `0 <= shift[i][0] <= 1`
- `0 <= shift[i][1] <= 100`

Solution

Approach 1: Simulation

Intuition

When given a shift operation, for example `[1, 4]`, we'll refer to the first part as the *shift-direction*, and the second part as the *shift-amount*.

The most obvious way of solving this problem is to simulate the shifts, one by one.

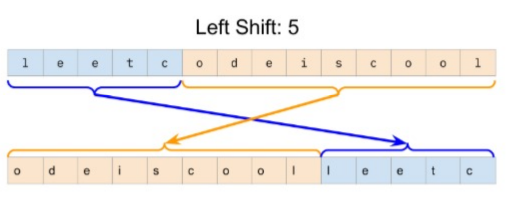
For example, suppose we have the string `"leetcodeiscool"`.

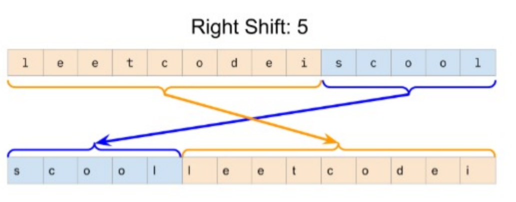
If we need to perform shift operation `[0, 5]`, then we'll shift the string *left* by 5, i.e.:

And if we need to perform shift operation `[1, 5]`, then we'll shift the string *right* by 5, i.e.:

We could write an algorithm that simulates the process in the animation for each shift operation. However, strings are immutable in many programming languages, including Java and Python. This means that once a string object has been created, it *cannot be modified*. Instead, a *new string* must be created if we want to change it. This makes "string modification" a very expensive operation, at $O(L)$, where L is the length of the string. Therefore, we don't want to make lots of little modifications: we want to combine them into one single modification that can be applied with a single new string creation.

So, how can we do that here? Well, observe that a *left shift* is equivalent to taking *shift-amount* letters off the front, and putting them onto the end. Similarly, a *right shift* is equivalent to taking *shift-amount* letters off the end, and putting them onto the front.





This will work as long as *shift-amount* is *less than the length of the string*.

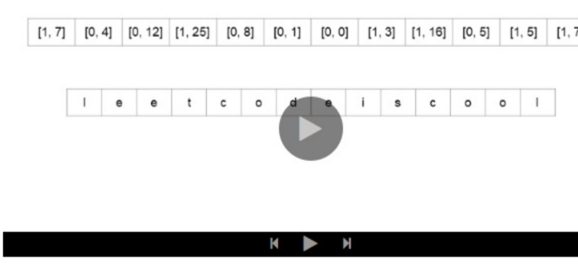
However, *shift-amounts* are allowed to be *longer* than the length of the string. For example, consider the word `"cabbage"` (which has seven letters). We need to apply the *right-shift* operation `[1, 9]` to it. If the word was more than nine letters long, we would have just taken nine letters off the end, and put them on the front. But there aren't nine letters to move! To find a solution, let's start by going back to the inefficient algorithm; moving the string by one position at a time.

Notice that after seven shifts we're back to `"cabbage"` again. Only the last two shifts mattered. The number seven is significant because it is the length of the input string. We can extend this a bit further by reasoning that if we had a *shift-amount* of 25, then we'd effectively have `7 + 7 + 7 + 4`. The `7`'s will do nothing, and it is only the final `4` that the string will be shifted by.

Recognize that the "remainder" after subtracting the length as many times as we can is simply taking the modulus of a number. i.e. `9 % 7 = 2` and `25 % 7 = 4`.

Therefore, if the *shift-amount* is greater than the length of the string, then we should start by reducing the *shift-amount* to modulo the length of the string. In fact, we don't even need the "if". For example, `6 % 7 = 6`. The modulo does nothing if the first number is smaller than the second.

To put this into a complete algorithm that solves the problem, we need to loop through the list of shift operations, doing a single concatenation for each one. Here is an animation of this process.



Algorithm

Java Python3 Copy

```java
class Solution {
  public String stringShift(String string, int[][] shift) {
    int len = string.length();
    for (int[] move : shift) {
      int direction = move[0];
      int amount = move[1] % len;
      if (direction == 0) {
        // Move necessary amount of characters from front to end
        string = string.substring(amount) + string.substring(0, amount);
      } else {
        // Move necessary amount of characters from end to front
        string = string.substring(len - amount) + string.substring(0, len - amount);
      }
    }
    return string;
  }
}
```

Complexity Analysis

Let L be the length of the string and N be the length of the `shift` array.

- Time complexity : $O(N \cdot L)$.

  Making a single modification to the input string has a cost of $O(L)$, as we need to create a new string with the modifications. We are making one modification for each shift operation. As there are N shift operations, this gives us a total time complexity of $O(N \cdot L)$.

- Space complexity : $O(L)$.

  While performing a string modification, we'll have both the original string and the new string in memory. Therefore, the space complexity is $O(L)$.

  Note that if you're using a language with mutable strings (such as C or C++), then it is possible to get the space complexity down to $O(1)$ by doing the shift operations in-place with a suitable algorithm. Approach 3 or 4 of the Rotate Strings Solution Article would be a great way of going about this.

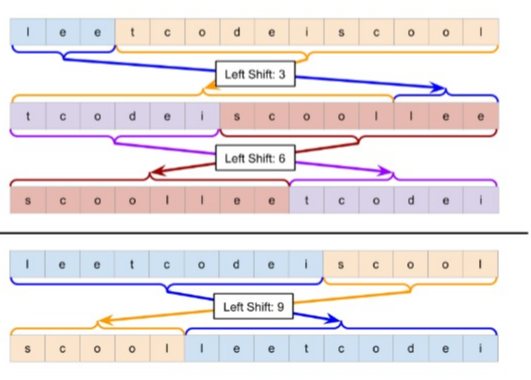
Before we came up with this approach, we briefly discussed a simpler approach where instead of doing each shift operation as a single string modification, we'd do it as *shift-amount* operations. What would the time complexity for this approach be? To simplify, we'll assume that *shift-amount* must be less than or equal to L (we can use the modulo operator to ensure this). Under this assumption, the worst case is where all the *shift-amounts* are exactly $L-1$. This means that applying a shift operation will do a $O(L)$ string modification, $L-1$ times. $(S-1) \cdot O(L) = O(L^2)$. Then with N shift operations to perform, we get a total of $O(N \cdot L^2)$. This is a lot worse!

Approach 2: Compute Net Shift

Intuition

In Approach 1, we calculated each shift at a cost of $O(L)$ each time (where L is the length of the input string). However, you might've noticed that we can combine *all* the shifts, and then perform a *single* string modification for *all of them*.

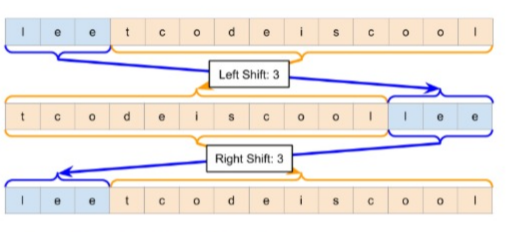
For example, if we have two left shifts, `[0, 3]` and `[0, 6]`, then we can combine them into a single left shift `[0, 3 + 6] = [0, 9]`. Then, instead of performing two separate $O(L)$ modifications, we can perform just one.



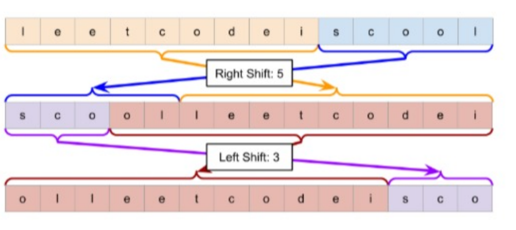
The same idea applies to the right shifts.

We could, therefore, add all the lefts together and apply them, and then add all the rights together, and apply them. This would mean we only needed to perform two $O(L)$ string modifications.

We can do even better than this though, as *left shifts and right shifts cancel each other out*. For example, a *left shift* by `3`, followed by a *right shift* by `3`, brings us back to the original string. It is, therefore, wasted computation to do these shift operations at all!



What about a *right shift* by `5`, followed by a *left shift* by `3`?



In effect, we are left with a *right shift* by `2`.

So for this approach, we'll write an algorithm that pre-processes the `shift` list, and then applies a single string modification.

Algorithm

As described above, we should go through the `shift` list adding up all the *right shift-amounts* and *left shift-amounts* into two separate sums.

And then once we have these two amounts, we need to work out which is bigger, and partially cancel it out with the other. We then need to do the relevant shift with what's remaining. If both are the same, then the string won't be changed at all.

Here is the code for this algorithm. After this code, we'll look at a slight optimization/ simplification, that uses a little more math.

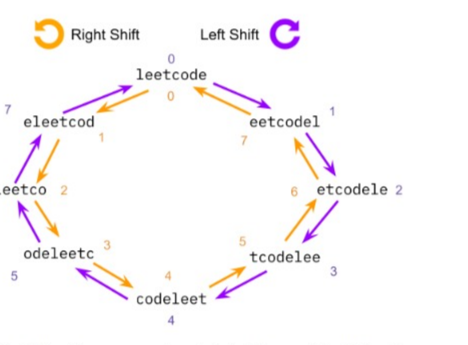
Java Python3 Copy

```java
class Solution {
  public String stringShift(String string, int[][] shift) {

    // Add up the left shifts and right shifts.
    int[] overallShifts = new int[2];
    for (int[] move : shift) {
      overallShifts[move[0]] += move[1];
    }
    int leftShifts = overallShifts[0];
    int rightShifts = overallShifts[1];

    // Determine which shift (if any) to perform.
    int len = string.length();
    if (leftShifts > rightShifts) {
      leftShifts = (leftShifts - rightShifts) % len;
      string = string.substring(leftShifts) + string.substring(0, leftShifts);
    }
    else if (rightShifts > leftShifts) {
      rightShifts = (rightShifts - leftShifts) % len;
      string = string.substring(len - rightShifts) + string.substring(0, len - rightShifts);
    }

    return string;
  }
}
```

To simplify the code further, remember that *right shifts* and *left shifts* cancel each other out. We'll just keep track of how many *left shifts* there are, by treating *right shifts* as negative *left shifts*.

If after doing this, the final value of `left_shifts` is positive, then we need to do a *left shift operation*. If it is negative, then we need to do a *right shift operation*, i.e. if it is `-5`, then we need to *right shift* by `5`.

We can simplify it further still by recognizing that the number of *right shifts* can be converted into a number of *left shifts*. Notice that all "valid" shifts of a string can be represented in a clock-like circle. For example, here is the "clock" diagram we could make for the word `"leetcode"`.



When we *left shift* the string, we are moving clockwise. When we *right shift* the string, we are moving anti-clockwise.

Now to get the overall left shift, we can simply take the total *left shifts* (regardless of whether it is positive or negative) modulo the length of the string:

`left_shifts = left_shifts % length of string`

In most programming languages, the modulo operator will add length to the shift amount until the result is positive. This is what we want. Note that the `%` operator doesn't work this way in *all* languages. In some languages, such as Java, *it does not make negative numbers positive*. Java's `Math.floorMod` function will do what we want though.

Java Python3 Copy

```java
class Solution {
  public String stringShift(String s, int[][] shift) {

    // Count the number of left shifts. A right shift is a negative left shift.
    int leftShifts = 0;
    for (int[] move : shift) {
      if (move[0] == 1) {
        move[1] = - move[1];
      }
      leftShifts += move[1];
    }

    // Convert back to a positive, do left shifts, and return.
    leftShifts = Math.floorMod(leftShifts, s.length());
    s = s.substring(leftShifts) + s.substring(0, leftShifts);
    return s;
  }
}
```

Complexity Analysis

Let L be the length of the string and N be the length of the `shift` array. Both sub-approaches here have the same complexity analysis.

- Time complexity : $O(N + L)$.

  The algorithms presented in Approach 2 both break the task into two sub-steps: calculating an overall shift and applying the shift. We'll analyze these one at a time and then combine the results.

  The first step loops through each of the N entries in the `shift` array, adding up the total number of left shifts and the total number of right shifts. Handling each entry is an $O(1)$ operation, so this first step has a total cost of $O(N)$.

  The second step applies a single string-shift operation. As discussed in the previous approach, a string-shift operation has a cost of $O(L)$.

  Because we are doing these steps one-after-the-other, and we don't know whether N or L is bigger, we add them to get a final time complexity of $O(N + L)$.

- Space complexity : $O(L)$.

  The first step uses constant extra space to keep track of the counts. This leaves us with the space complexity of modifying a string, which, as discussed before, requires auxiliary space of $O(L)$.

  As stated in the previous approach, it is possible to get the space complexity down to $O(1)$ by using a language with mutable strings.

Analysis written by @hai_dee