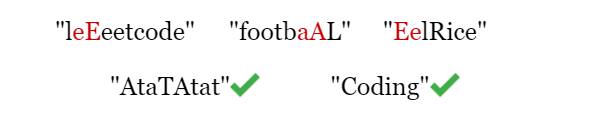
**Overview**

In this problem, we are given a string s containing some lower and upper case letters of the alphabet.

There may be two adjacent characters in the string that represent the same letter, but one of them is a lowercase letter and the other is an uppercase letter (i.e. qQ, Yy). For convenience, we refer to such two adjacent letters as **pair**.

If there is no such a pair in the string, then we can call the string a **good** string.

Take a look at some examples where strings with green checkmarks are good strings, while the other strings are **not good**, since we can find some pairs from them (pairs are colored in red).



As long as we find a pair, we can remove it from the string. Therefore, we will finally have a good string (note that an empty string is also a good string).

Here our task is to find the final good string.

**Approach 1: Iteration.**

**Intuition**

Since the answer is guaranteed to be **unique**, we don't need to worry about the order of deletion, we just keep deleting pairs until no more pairs can be found.

One thing to note is that to judge if two adjacent characters make a pair? We do easily tell that patterns like aA, Bb, cC are pairs, but how to implement the code? We can use the their **ASCII values** as reference, each character has a unique ASCII value:

* a = 97, A = 65
* b = 98, B = 66
* c = 99, C = 67 ...
* z = 122, Z = 90

Thus we can tell that two characters make a pair, when and only when their ASCII values differ by 323232 (Since the sentence only contains letters of alphabet, we do not need to consider about other speical characters). Keep This is a very common trick, keep it in mind!

**Algorithm**

1. If the size of string s is smaller than 2, return s directly.
2. Iterate over all adjacent characters in s.
   * If we find a pair, remove it from s, and start over from step 2.
   * Otherwise, we don't need to iterate. Move to step 3.
3. Return s as the final good string.

**Complexity Analysis**

Let nn*n* be the length of the input string s.

* Time complexity: O(n2)O(n^2)*O*(*n*2)
  + Each iteration for s takes O(n)O(n)*O*(*n*) time.
  + In the worst-case scenario, we can remove one pair in each iteration, there might be O(n)O(n)*O*(*n*) pairs.
  + In summary, the time complexity is O(n2)O(n^2)*O*(*n*2).
* Space complexity: O(n)O(n)*O*(*n*)
  + After we remove a pair, we concatenate the rest strings into a new string and start iterating again. Making copies of the rest of s requires O(n)O(n)*O*(*n*) space.
  + Therefore, the space complexity is O(n)O(n)*O*(*n*).

class Solution:

def makeGood(self, s: str) -> str:

# if s has less than 2 characters, we just return itself.

while len(s) > 1:

# 'find' records if we find any pair to remove.

find = False

# Check every two adjacent characters, curr\_char and next\_char.

for i in range(len(s) - 1):

curr\_char, next\_char = s[i], s[i + 1]

# If they make a pair, remove them from 's' and let 'find = True'.

if abs(ord(curr\_char) - ord(next\_char)) == 32:

s = s[:i] + s[i + 2:]

find = True

break

# If we cannot find any pair to remove, break the loop.

if not find:

break

return s

**Approach 2: Recursion**

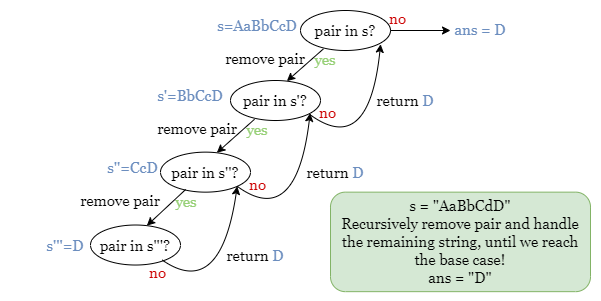
**Intuition**

We will implement the same algorithm in approach 1 using recursive method.

The trick is that each time a recursive function calls itself, it reduces the given problem into subproblems. The recursion call continues until it reaches a point where the subproblem can be solved without further recursion.

In this problem, once we find a pair that should be deleted, we are actually reducing s into a new string s' which is 2 characters smaller. Then the function calls itself for this smaller subproblem. When we can't find a pair for s, we have reached the base case where the problem can be solved by just returning s without further recursion!

Here is a brief example of the recursion approach.



**Algorithm**

Iterate over the input string s and check if a pair exists. - If we find one pair, remove it from s, and start over this step with the remaining string. - Otherwise, return s.

**Complexity Analysis**

Let nn*n* be the length of the input string s.

* Time complexity: O(n2)O(n^2)*O*(*n*2)
  + Similarly, it takes O(n)O(n)*O*(*n*) time to iterate through s to find a pair to be removed.
  + In the worst-case scenario, there might be O(n)O(n)*O*(*n*) pairs.
  + To sum up, the time complexity is O(n)O(n)*O*(*n*).
* Space complexity: O(n2)O(n^2)*O*(*n*2)
  + Recall the picture at the begining of this approach, the space complexity is proportional to the maximum depth of the recursion tree. We have up to n/2n/2*n*/2 pairs, which equals a recursion tree of depth O(n)O(n)*O*(*n*).
  + Each function call takes O(n)O(n)*O*(*n*) space.
  + Therefore, the overall space complexity is O(n2)O(n^2)*O*(*n*2).

class Solution:

def makeGood(self, s: str) -> str:

# If we find a pair in 's', remove this pair from 's'

# and solve the remaining string recursively.

for i in range(len(s) - 1):

if abs(ord(s[i]) - ord(s[i + 1])) == 32:

return self.makeGood(s[:i] + s[i + 2:])

# Base case, if we can't find a pair, just return 's'.

return s

**Approach 3: Stack**

**Intuition**

In the previous approaches, we have to make multiple iterations in some cases. Can we solve the problem using just one iteration? The answer is Yes!

Let's suppose that we find a pair in step i in the iteration, it means that the characterss[i] and s[i - 1] make a pair. We should ignore the s[i] and remove s[i - 1] from the end of the good string. Otherwise, we should add s[i] to the end of the good string.

Looks familiar? If we store all the previously visited characters in a stack, then the operations on the stack equal the operations on the end of the previous good string! Hence, let's try using a stack to store all the characters we encounter but haven't been removed.

Take the slides below as an example:

**Algorithm**

1. Initialize an empty stack stack.
2. For each character currChar in s:
   * If currChar pairs with the last character in stack, remove the character at the top of stack.
   * Otherwise, add currChar to stack.
3. Once we have finished iterating, return the string concatenated by all the remaining characters in stack.

**Complexity Analysis**

Let nn*n* be the length of the input string s.

* Time complexity: O(n)O(n)*O*(*n*)
  + We only need one iteration over s.
  + At each step, we either remove the last character from stack, or add a character to stack, both of which take constant time.
  + Therefore, the overall time complexity is O(n)O(n)*O*(*n*).
* Space complexity: O(n)O(n)*O*(*n*)
  + We use a stack to store all the characters we encounter. Since we only pop characters when finding a pair, in worst-case scenario, we may have O(n)O(n)*O*(*n*) characters stored in stack. Thus the space complexity is O(n)O(n)*O*(*n*).

**Approach 4: Two pointers, in-place modify**

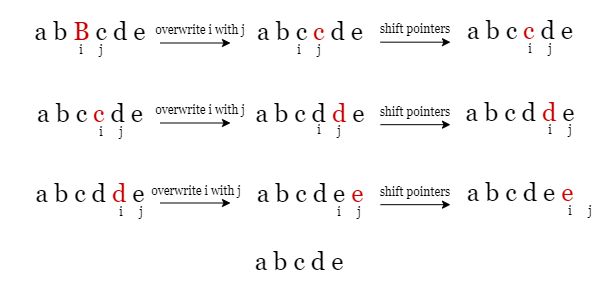
**Intuition**

In the previous approach, we use a stack to record the remaining characters. In the worst-case scenario, the stack contains up to nn*n* characters, which takes O(n)O(n)*O*(*n*) space.

We can use a similar idea on the string s itself to further improve the space complexity!

What is the core of this method? Overwriting characters.

Here is an example of 'deleting' a character at index i and modifying the string in-place. Instead of directly removing the character from the string, we overwrite it with the character at index j. After that, it's like we swapped two positions: we move the character to be kept forward, and the position representing the deleted character backward.



The red character not necessarily stands for the deleted character B, why?

This is because we only need to keep track of the exact characters that should be retained. However, for the deleted character, we just need to record the position where it should be, which is the index marked in red.

Again, we are not literally deleting them, but simply overwriting them with the characters that follow. Therefore, taken as a whole, it looks like we are gradually 'pushing' the position they occupy towards the end of the string.

The places for deleted characters, colored in red, gradually move to the end of s. Hence, when the iteration ends, the first half of the string is the good string we want!

Note that string is **mutable** in C++, thus they can be changed after the initialization. However, string is **immutable** in Python and Java, thus a slide operation on string will create a new copy which takes O(n)O(n)*O*(*n*) space. Hence, this algorithm is applicable to C++.

**Algorithm**

1. Initialize two pointers end = 0, cur = 0.
2. Iterate over s using cur. For each character s[cur].
   * If s[cur] makes a pair with the s[end], remove s[end] from the good string by decrement end by 1.
   * Otherwise, we add s[cur] to the end of the good string by overwritting s[end] by s[cur] and incrementing end by 1.
3. Once we have finished iterating, return the first half of the string s[:end] as the good string.

Note: Java and Python are **not** applicable to this solution, since their strings are **immutable**, as we mention before.

**Complexity Analysis**

Let nn*n* be the length of the input string s.

* Time complexity: O(n)O(n)*O*(*n*)
  + We only need one iteration over s.
  + In each step, we update the position of the two pointers and overwrite a character at most once, both of which take constant time.
  + To sum up, the overall time complexity is O(n)O(n)*O*(*n*).
* Space complexity: O(1)O(1)*O*(1)
  + We modify the input string s in place, thus the overall space complexity is O(1)O(1)*O*(1).