<https://leetcode.com/problems/longest-happy-string/description/>

A string s is called happy if it satisfies the following conditions:

s only contains the letters 'a', 'b', and 'c'.

s does not contain any of "aaa", "bbb", or "ccc" as a substring.

s contains at most a occurrences of the letter 'a'.

s contains at most b occurrences of the letter 'b'.

s contains at most c occurrences of the letter 'c'.

Given three integers a, b, and c, return the longest possible happy string. If there are multiple longest happy strings, return any of them. If there is no such string, return the empty string "".

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

**Example 1:**

**Input:** a = 1, b = 1, c = 7

**Output:** "ccaccbcc"

**Explanation:** "ccbccacc" would also be a correct answer.

**Example 2:**

**Input:** a = 7, b = 1, c = 0

**Output:** "aabaa"

**Explanation:** It is the only correct answer in this case.

**Constraints:**

0 <= a, b, c <= 100

a + b + c > 0

**Attempt 1: 2025-02-05**

**Solution 1: Greedy + Heap (180 min)**

class Solution {

    public String longestDiverseString(int a, int b, int c) {

        // Max-heap to store characters based on their remaining counts

        PriorityQueue<Node> maxPQ = new PriorityQueue<>((x, y) -> y.count - x.count);

        // Add characters to the heap if their count is greater than 0

        if(a > 0) {

            maxPQ.offer(new Node('a', a));

        }

        if(b > 0) {

            maxPQ.offer(new Node('b', b));

        }

        if(c > 0) {

            maxPQ.offer(new Node('c', c));

        }

        StringBuilder sb = new StringBuilder();

        while(!maxPQ.isEmpty()) {

            // Get the character with the highest remaining count

            Node first = maxPQ.poll();

            // If the last two characters in the result are the same as the current character,

            // we cannot add it again to avoid three consecutive characters

            if(sb.length() >= 2 && sb.charAt(sb.length() -  1) == first.ch

                && sb.charAt(sb.length() - 2) == first.ch) {

                // If no other character is available, break

                if(maxPQ.isEmpty()) {

                    break;

                }

                // Get the second highest character

                Node second = maxPQ.poll();

                sb.append(second.ch);

                second.count--;

                // Add the second character back to the heap if its count is still greater than 0

                if(second.count > 0) {

                    maxPQ.offer(second);

                }

                // Add the first character back to the heap

                maxPQ.offer(first);

            } else {

                // Append the character to the result and reduce its count

                sb.append(first.ch);

                first.count--;

                // Add the character back to the heap if its count is still greater than 0

                if(first.count > 0) {

                    maxPQ.offer(first);

                }

            }

        }

        return sb.toString();

    }

}

// Helper class to store character and its count

class Node {

    char ch;

    int count;

    public Node(char ch, int count) {

        this.ch = ch;

        this.count = count;

    }

}

Time Complexity: (O(n)), where (n = a + b + c).

Space Complexity: (O(n)), due to the result string.

**Refer to Deepseek**

To solve the LeetCode problem 1405, "Longest Happy String," we need to construct the longest possible string using the characters 'a', 'b', and 'c' such that no three identical characters are consecutive. The input provides the maximum counts of each character, and we need to use them optimally to build the string.

**Here's a Java solution using a priority queue (max-heap) to always pick the character with the highest remaining count, ensuring that we don't have three identical characters in a row:**

import java.util.\*;

class Solution {

public String longestDiverseString(int a, int b, int c) {

// Max-heap to store characters based on their remaining counts

PriorityQueue<Pair> maxHeap = new PriorityQueue<>((x, y) -> y.count - x.count);

// Add characters to the heap if their count is greater than 0

if (a > 0) maxHeap.offer(new Pair('a', a));

if (b > 0) maxHeap.offer(new Pair('b', b));

if (c > 0) maxHeap.offer(new Pair('c', c));

StringBuilder result = new StringBuilder();

while (!maxHeap.isEmpty()) {

// Get the character with the highest remaining count

Pair first = maxHeap.poll();

// If the last two characters in the result are the same as the current character,

// we cannot add it again to avoid three consecutive characters

if (result.length() >= 2 &&

result.charAt(result.length() - 1) == first.ch &&

result.charAt(result.length() - 2) == first.ch) {

// If no other character is available, break

if (maxHeap.isEmpty()) break;

// Get the second highest character

Pair second = maxHeap.poll();

result.append(second.ch);

second.count--;

// Add the second character back to the heap if its count is still greater than 0

if (second.count > 0) maxHeap.offer(second);

// Add the first character back to the heap

maxHeap.offer(first);

} else {

// Append the character to the result and reduce its count

result.append(first.ch);

first.count--;

// Add the character back to the heap if its count is still greater than 0

if (first.count > 0) maxHeap.offer(first);

}

}

return result.toString();

}

// Helper class to store character and its count

class Pair {

char ch;

int count;

Pair(char ch, int count) {

this.ch = ch;

this.count = count;

}

}

}

**Explanation:**

**Priority Queue (Max-Heap):** We use a max-heap to always pick the character with the highest remaining count. This ensures that we use the most frequent character first, maximizing the length of the string.

**Avoiding Three Consecutive Characters:** Before appending a character, we check if the last two characters in the result are the same as the current character. If they are, we skip appending it to avoid having three consecutive identical characters.

**Appending Characters:** If the current character can be appended without violating the rule, we append it and reduce its count. If not, we append the next available character from the heap.

**Termination:** The loop continues until the heap is empty or no more characters can be appended without violating the rule.

**Example:**

For a = 1, b = 1, c = 7, the output could be "ccbccacc" or any other valid string that uses all characters without having three identical characters in a row.

This solution ensures that we construct the longest possible "happy string" given the constraints.

**Time Complexity:**

Heap Operations:

The solution uses a max-heap (priority queue) to store the characters and their counts.

Each insertion (offer) and removal (poll) operation on the heap takes ( O(log k) ), where ( k ) is the number of distinct characters. Since there are only 3 characters (a, b, c), ( k = 3 ), so each heap operation takes ( O(log 3) ), which is constant time ( O(1) ).

Number of Heap Operations:

In the worst case, we append one character to the result string in each iteration of the while loop.

The total number of characters in the result string is ( n = a + b + c ), where ( a ), ( b ), and ( c ) are the input counts.

For each character appended, we perform:

One poll operation to get the character with the highest count.

One offer operation to reinsert the character (if its count is still greater than 0).

Therefore, the total number of heap operations is proportional to ( n ).

Overall Time Complexity:

Since each heap operation takes ( O(1) ) (because ( k = 3 )), and we perform ( O(n) ) heap operations, the total time complexity is:

Here, ( n = a + b + c ).

**Space Complexity:**

Heap Storage:

The heap stores at most 3 elements (one for each character a, b, c), so the space required for the heap is ( O(1) ).

Result String:

The StringBuilder stores the result string, which can have up to ( n = a + b + c ) characters. Therefore, the space required for the result string is ( O(n) ).

Overall Space Complexity:

The dominant factor is the space required for the result string, so the space complexity is:

**Summary:**

Time Complexity: ( O(n) ), where ( n = a + b + c ).

Space Complexity: ( O(n) ), due to the result string.

**Why is the Time Complexity ( O(n) )?**

Even though heap operations typically take ( O(log k) ), in this case, ( k = 3 ) (constant), so each heap operation is ( O(1) ).

Since we perform ( O(n) ) heap operations, the total time complexity is ( O(n) ).

**Why is the Space Complexity ( O(n) )?**

The result string can have up to ( n ) characters, so the space required for the result string is ( O(n) ).

The heap uses only ( O(1) ) space, as it stores at most 3 elements.

This solution is efficient and scales linearly with the total number of characters ( n = a + b + c ).

**Any Mathematical Proof for the greedy approach ?**

The greedy approach used in the solution for LeetCode 1405 ("Longest Happy String") is based on the idea of always selecting the character with the highest remaining count at each step, while ensuring that no three identical characters are consecutive. To understand why this approach works, we can analyze it mathematically.

**Key Observations:**

Maximizing Length:

To maximize the length of the string, we should use the most frequent character as often as possible, as it contributes the most to the length.

However, we cannot use the same character more than twice in a row, as that would violate the "happy string" condition.

Balancing Characters:

If we always use the most frequent character, we might run out of other characters to break the sequence of three identical characters.

Therefore, we need to balance the usage of characters to ensure that we always have a way to break sequences of two identical characters.

**Greedy Choice Property:**

The greedy choice in this problem is to always select the character with the highest remaining count at each step, provided that adding it does not result in three identical consecutive characters. This choice is optimal because:

**It maximizes the usage of the most frequent character, which directly contributes to the length of the string.**

**It ensures that we always have a way to break sequences of two identical characters by using the next most frequent character.**

**Optimal Substructure:**

The problem exhibits optimal substructure because the optimal solution to the problem can be constructed from optimal solutions to its subproblems. Specifically:

**At each step, the decision to append a character depends only on the current state of the string and the remaining counts of the characters.**

**The optimal solution for the remaining characters (after appending a character) is independent of the choices made earlier.**

**Mathematical Proof of Correctness:**

To prove that the greedy approach works, we can use an exchange argument:

Assume an Optimal Solution Exists:

Let ( S ) be an optimal solution (the longest happy string) that does not follow the greedy approach.

Let ( S' ) be the solution constructed using the greedy approach.

Compare ( S ) and ( S' ):

If ( S ) and ( S' ) are the same, then the greedy approach is optimal.

If ( S ) and ( S' ) differ, there must be a point where ( S ) makes a non-greedy choice (i.e., it does not use the character with the highest remaining count).

Exchange Argument:

At the point where ( S ) makes a non-greedy choice, we can replace the non-greedy choice with the greedy choice (using the character with the highest remaining count).

This replacement will either:

Increase the length of the string (if the greedy choice allows more characters to be used later), or

Keep the length the same (if the greedy choice does not affect the overall length).

In either case, the new solution ( S'' ) (after the exchange) is at least as good as ( S ).

Conclusion:

Since we can always replace non-greedy choices with greedy choices without making the solution worse, the greedy approach must be optimal.

**Why the Greedy Approach Works:**

The greedy approach ensures that we always use the most frequent character as often as possible, which directly maximizes the length of the string.

By always breaking sequences of two identical characters with the next most frequent character, we avoid violating the "happy string" condition.

The priority queue (max-heap) ensures that we always have access to the character with the highest remaining count, making the greedy choice efficient.

**Example Walkthrough:**

Consider a = 1, b = 1, c = 7:

Start with the most frequent character c. Append cc.

Use the next most frequent character b to break the sequence. Append b.

Append cc again.

Use the next most frequent character a to break the sequence. Append a.

Append cc again.

The final string is ccbccacc, which is the longest possible happy string for the given counts.

**Conclusion:**

**The greedy approach is optimal because it maximizes the usage of the most frequent character while ensuring that no three identical characters are consecutive. The mathematical proof relies on the exchange argument, showing that any non-greedy choice can be replaced with a greedy choice without making the solution worse. This ensures that the greedy approach constructs the longest possible happy string.**

**Refer to**

[L984.String Without AAA or BBB (Ref.L1405,L1642)](note://WEB41677fa6ac90e3ed5b0da272fd5e7e21)

[L1642.Furthest Building You Can Reach (Ref.L1405)](note://WEB63f41eede3d2b75fae17e8adc3dd4d8e)