<https://leetcode.com/problems/minimum-ascii-delete-sum-for-two-strings/description/>

Given two strings s1 and s2, return the lowest ASCII sum of deleted characters to make two strings equal.

**Example 1:**

**Input:** s1 = "sea", s2 = "eat"

**Output:** 231

**Explanation:** Deleting "s" from "sea" adds the ASCII value of "s" (115) to the sum.

Deleting "t" from "eat" adds 116 to the sum.

At the end, both strings are equal, and 115 + 116 = 231 is the minimum sum possible to achieve this.

**Example 2:**

**Input:** s1 = "delete", s2 = "leet"

**Output:** 403

**Explanation:** Deleting "dee" from "delete" to turn the string into "let",adds 100[d] + 101[e] + 101[e] to the sum.

Deleting "e" from "leet" adds 101[e] to the sum.At the end, both strings are equal to "let", and the answer is 100+101+101+101 = 403.

If instead we turned both strings into "lee" or "eet", we would get answers of 433 or 417, which are higher.

**Constraints:**

1 <= s1.length, s2.length <= 1000

s1 and s2 consist of lowercase English letters.

**Question before we start work on the code**

**Can we translate Leetcode 712 problem into "find the LCS (longest common subsequence)" ? Because after finding the LCS, the remain part will give the minimum ASCII sum in each string ? Is this understanding correct or wrong ?**

Yes, you are on the right track, and your understanding is partially correct. The problem of **LeetCode 712** can indeed be related to the **Longest Common Subsequence (LCS)**, but there are some additional considerations when transforming the problem into the LCS context.

**How LCS Relates to the Problem:**

The **Longest Common Subsequence (LCS)** of two strings gives the longest sequence of characters that appear in both strings in the same order, but not necessarily contiguously.

Once you find the LCS, the characters that are not part of the LCS in each string must be deleted in order to make the two strings identical.

**Explanation:**

If you find the LCS of two strings s1 and s2, the characters that are **not part of the LCS** in each string need to be deleted.

The **ASCII sum** of these deleted characters would then give the total deletion cost for both strings.

Thus, the idea is:

Find the LCS of s1 and s2.

Calculate the total ASCII value of the characters in s1 and s2.

Subtract the total ASCII value of the LCS from the sum of the ASCII values of the original strings. The remainder gives you the minimum deletion cost.

**Why LCS Is Helpful, But Not Enough by Itself:**

Although this approach is conceptually correct, the solution is **not simply about finding the LCS**. You also need to account for the total ASCII deletion cost, which is not directly handled by the LCS calculation itself. That’s why the problem uses dynamic programming to specifically track and minimize the **ASCII values of the characters being deleted**.

**LCS Approach vs. Minimum ASCII Delete Sum:**

You're correct in thinking that the LCS gives us the common characters, and deleting characters not in the LCS seems like a reasonable approach to minimize deletions. However, the **LCS** only tells us **which characters remain**, but it doesn't directly help in minimizing the **ASCII sum of deletions**. Here's why:

**LCS Focuses Only on Character Matching, Not Costs:**

LCS finds the longest common subsequence by matching characters in two strings.

However, LCS only maximizes the length of the subsequence. It does **not consider the ASCII values of the characters that are left out**.

**Two different subsequences could have the same length, but the ASCII values of the deleted characters can be different, and that's critical in this problem.**

It is possible for two strings to have more than one **Longest Common Subsequence (LCS)** of the same length. The LCS is not necessarily unique—there can be multiple subsequences of the same maximum length that qualify as the LCS.

**Example:**

Consider the strings:

s1 = "abc"

s2 = "acb"

**LCS Calculation:**

The **length** of the LCS is 2, and both of the following subsequences are valid LCSs:

"ab" from s1 and "ab" from s2

"ac" from s1 and "ac" from s2

Both subsequences have the same length (2), so they are both valid **LCSs**.

**If we only focus on find s1 and s2's LCS, and get the LCS = "ab", then the remain part is "c", the cost to delete is 2 \* 99 (ASCII of "c") = 198, but the correct answer is to choose LCS = "ac", then the remain part is "b", the cost to delete is 2 \* 98 (ASCII of "b") = 196. So finding the LCS of two strings cannot guarantee directly find the minimum ASCII delete. We still need another more optimal DP solution to find answer directly instead of find LCS first then find remain ASCII sum.**

**Attempt 1: 2024-9-9**

**Solution 1: Native DFS (180 min, not easy to come up with base condition and recursive condition, TLE 63/93)**

class Solution {

    public int minimumDeleteSum(String s1, String s2) {

        return helper(s1, s2, 0, 0);

    }

    private int helper(String s1, String s2, int i, int j) {

        // Base case: If one of the strings is exhausted,

        // delete all remaining characters from the other string

        if(i == s1.length()) {

            int sum = 0;

            for(int k = j; k < s2.length(); k++) {

                sum += s2.charAt(k);

            }

            return sum;

        }

        if(j == s2.length()) {

            int sum = 0;

            for(int k = i; k < s1.length(); k++) {

                sum += s1.charAt(k);

            }

            return sum;

        }

        // If characters match, move both pointers without deleting to

        // minimize the cost of deletion

        if(s1.charAt(i) == s2.charAt(j)) {

            return helper(s1, s2, i + 1, j + 1);

        }

        // If characters don't match, either delete from s1 or delete from s2

        int delete1 = s1.charAt(i) + helper(s1, s2, i + 1, j); // delete s1[i]

        int delete2 = s2.charAt(j) + helper(s1, s2, i, j + 1); // delete s2[j]

        // Return the minimum cost of deletions

        return Math.min(delete1, delete2);

    }

}

Time Complexity: O(2^(n + m))

Space Complexity: O(2^(n + m))

**Refer to chatGPT**

Here is a **pure DFS** solution for LeetCode problem **712 - Minimum ASCII Delete Sum for Two Strings** in Java. The idea behind this solution is to recursively explore all possible deletions, keeping track of the minimum ASCII delete sum. It compares characters from the two strings at each step, either deleting one or the other, until we reach the end of one or both strings.

**Java DFS Solution:**

public class Solution {

public int minimumDeleteSum(String s1, String s2) {

return dfs(s1, s2, 0, 0);

}

private int dfs(String s1, String s2, int i, int j) {

// Base case: If one of the strings is exhausted, delete all remaining characters from the other string

if (i == s1.length()) {

int sum = 0;

for (int k = j; k < s2.length(); k++) {

sum += s2.charAt(k);

}

return sum;

}

if (j == s2.length()) {

int sum = 0;

for (int k = i; k < s1.length(); k++) {

sum += s1.charAt(k);

}

return sum;

}

// If characters match, move both pointers without deleting

if (s1.charAt(i) == s2.charAt(j)) {

return dfs(s1, s2, i + 1, j + 1);

}

// If characters don't match, either delete from s1 or delete from s2

int deleteS1 = s1.charAt(i) + dfs(s1, s2, i + 1, j); // delete s1[i]

int deleteS2 = s2.charAt(j) + dfs(s1, s2, i, j + 1); // delete s2[j]

// Return the minimum cost of deletions

return Math.min(deleteS1, deleteS2);

}

}

**Explanation:**

**Base Case:**

If we've reached the end of one of the strings (i == s1.length() or j == s2.length()), **we must delete all remaining characters in the other string**. This is handled by summing up the ASCII values of the remaining characters in the other string.

**Recursive Case:**

If the characters at the current positions i and j of s1 and s2 match (s1.charAt(i) == s2.charAt(j)), **no deletion is needed**, so we move both pointers forward (dfs(s1, s2, i + 1, j + 1)).

If they don't match, we try two options:

Delete **s1[i]** and move forward in s1, adding the ASCII value of s1[i] to the result.

Delete **s2[j]** and move forward in s2, adding the ASCII value of s2[j] to the result.

We take the **minimum** of the two options.

**Time Complexity:**

Without memoization, this solution has an exponential time complexity O(2^(n + m)), where n and m are the lengths of the two strings. It checks all combinations of deletions.

**Optimization:**

This DFS approach can be optimized by adding **memoization** to avoid recomputing the results for the same pairs of indices. However, this example is a **pure DFS solution** without memoization.

**Solution 2: DFS + Memoization (10 min)**

class Solution {

    public int minimumDeleteSum(String s1, String s2) {

        Integer[][] memo = new Integer[s1.length() + 1][s2.length() + 1];

        return helper(s1, s2, 0, 0, memo);

    }

    private int helper(String s1, String s2, int i, int j, Integer[][] memo) {

        // Base case: If one of the strings is exhausted,

        // delete all remaining characters from the other string

        if(i == s1.length()) {

            int sum = 0;

            for(int k = j; k < s2.length(); k++) {

                sum += s2.charAt(k);

            }

            return sum;

        }

        if(j == s2.length()) {

            int sum = 0;

            for(int k = i; k < s1.length(); k++) {

                sum += s1.charAt(k);

            }

            return sum;

        }

        if(memo[i][j] != null) {

            return memo[i][j];

        }

        // If characters match, move both pointers without deleting to

        // minimize the cost of deletion

        if(s1.charAt(i) == s2.charAt(j)) {

            return memo[i][j] = helper(s1, s2, i + 1, j + 1, memo);

        }

        // If characters don't match, either delete from s1 or delete from s2

        int delete1 = s1.charAt(i) + helper(s1, s2, i + 1, j, memo); // delete s1[i]

        int delete2 = s2.charAt(j) + helper(s1, s2, i, j + 1, memo); // delete s2[j]

        // Return the minimum cost of deletions

        return memo[i][j] = Math.min(delete1, delete2);

    }

}

**Above DFS solution get time limit exceed (TLE), based on above Native DFS solution i have a memoization solution, it able to pass all test cases, how the memo helps ?**

**Refer to chatGPT**

In your **memoized solution**, the key improvement over the pure DFS solution is **caching** the results of previously computed subproblems, which avoids redundant calculations. This helps prevent the **time limit exceeded (TLE)** issue.

**How Memoization Helps:**

In the pure DFS approach, the same subproblem (i.e., the pair of indices (i, j) in the strings s1 and s2) can be recomputed multiple times during recursion, leading to an **exponential time complexity**. Memoization addresses this by storing (or caching) the results of these subproblems in a 2D array (memo[i][j]), so that each subproblem is solved only once.

**Detailed Steps:**

**Subproblem Reuse:**

At each recursive call helper(s1, s2, i, j), the algorithm checks if the result for that subproblem (i, j) has already been computed (i.e., if memo[i][j] != null).

If the result is cached in the memo array, it is returned immediately without further recursive exploration.

This avoids the need to recompute the same subproblem multiple times, which is common in the pure DFS approach.

**Reduced Redundancy:**

Without memoization, the DFS approach might explore the same branches of the recursion tree over and over. For example, if both s1 and s2 are of length n, the algorithm could end up exploring a vast number of subproblems, many of which are repeated.

Memoization reduces the time complexity from **exponential** to **polynomial** by storing results and reusing them. In your case, the **2D memoization table** ensures that each subproblem is solved only once, leading to significant time savings.

**Example:** Let's say s1 = "abc" and s2 = "ab". During recursion:

The subproblem (i = 1, j = 1) (corresponding to the suffixes "bc" and "b") might be encountered multiple times if there is no memoization.

With memoization, the result of helper(s1, s2, 1, 1) is computed once and stored in memo[1][1]. The next time the subproblem (1, 1) is encountered, the stored result is retrieved, avoiding unnecessary recalculations.

**Time Complexity:**

In the memoized solution, the time complexity becomes **O(n \* m)**, where n is the length of s1 and m is the length of s2. This is because each pair of indices (i, j) is processed only once.

Without memoization (in pure DFS), the time complexity is **O(2^(n + m))** due to the repeated branching of recursive calls.

**Summary:**

Memoization ensures that each subproblem is solved just once, drastically improving the efficiency of the solution by reducing redundant work. Instead of exponential time complexity, the solution now runs in polynomial time, which allows it to handle larger input sizes and pass all test cases without exceeding time limits.

**Solution 3: DP (60 min)**

class Solution {

    public int minimumDeleteSum(String s1, String s2) {

        int len1 = s1.length();

        int len2 = s2.length();

        // dp[i][j] represents the minimum ASCII delete sum

        // required to make the substrings s1[i:] and s2[j:] equal

        int[][] dp = new int[len1 + 1][len2 + 1];

        // row -> s1(i), col -> s2(j)

        // e.g s1 = "sea", s2 = "eat"

        // Initialize last row and col

        //    j  e   a   t  ""

        // i     0   1   2   3    ^

        // s  0              313  |

        // e  1              198  |

        // a  2              97   |

        // "" 3 314 213 116  0    |

        //    <---------------

        // s1 = empty string -> remove all remain chars in s2

        for(int j = len2 - 1; j >= 0; j--) {

            dp[len1][j] = dp[len1][j + 1] + s2.charAt(j);

        }

        // s2 = empty string -> remove all remain chars in s1

        for(int i = len1 - 1; i >= 0; i--) {

            dp[i][len2] = dp[i + 1][len2] + s1.charAt(i);

        }

        // Fill the DP table from bottom-right to top-left

        for(int i = len1 - 1; i >= 0; i--) {

            for(int j = len2 - 1; j >= 0; j--) {

                if(s1.charAt(i) == s2.charAt(j)) {

                    // If characters match, no deletion needed

                    dp[i][j] = dp[i + 1][j + 1];

                } else {

                    // Otherwise, take the minimum of deleting s1[i] or s2[j]

                    int delete1 = s1.charAt(i) + dp[i + 1][j];

                    int delete2 = s2.charAt(j) + dp[i][j + 1];

                    dp[i][j] = Math.min(delete1, delete2);

                }

            }

        }

        // The answer for the entire strings is in dp[0][0]

        return dp[0][0];

    }

}

Time Complexity: O(n \* m)

Space Complexity: O(n \* m)

**Write 2D DP solution based on above pure DFS solution**

**Refer to chatGPT**

Here is a **2D Dynamic Programming (DP)** solution based on the pure DFS approach for LeetCode 712 ("Minimum ASCII Delete Sum for Two Strings"). The DP table will store the minimum ASCII delete sum for every possible suffix combination of s1 and s2.

**DP Table Explanation:**

We use a **2D table dp[i][j]** where dp[i][j] represents the minimum ASCII delete sum required to make the substrings s1[i:] and s2[j:] equal.

The idea is to build this table **bottom-up**, meaning we start from the base cases (where one of the strings is empty) and work our way to the final result (dp[0][0]), which will store the minimum delete sum for the full strings.

**Transition:**

If the characters at s1[i] and s2[j] match, then we can skip both characters and look at dp[i+1][j+1] (i.e., the result for the remaining substrings).

If the characters do not match, we have two choices:

Delete the character s1[i] and add its ASCII value, then move to dp[i+1][j].

Delete the character s2[j] and add its ASCII value, then move to dp[i][j+1].

We take the minimum of these two options to fill in dp[i][j].

public class Solution {

public int minimumDeleteSum(String s1, String s2) {

int n = s1.length();

int m = s2.length();

// Create a DP table to store the minimum ASCII sum of deletions

int[][] dp = new int[n + 1][m + 1];

// Base case: If s1 is exhausted, we need to delete all characters from s2

for (int j = m - 1; j >= 0; j--) {

dp[n][j] = dp[n][j + 1] + s2.charAt(j);

}

// Base case: If s2 is exhausted, we need to delete all characters from s1

for (int i = n - 1; i >= 0; i--) {

dp[i][m] = dp[i + 1][m] + s1.charAt(i);

}

// Fill the DP table from bottom-right to top-left

for (int i = n - 1; i >= 0; i--) {

for (int j = m - 1; j >= 0; j--) {

if (s1.charAt(i) == s2.charAt(j)) {

// If characters match, no deletion is needed

dp[i][j] = dp[i + 1][j + 1];

} else {

// If characters don't match, we either delete from s1 or s2

dp[i][j] = Math.min(

s1.charAt(i) + dp[i + 1][j], // delete s1[i]

s2.charAt(j) + dp[i][j + 1] // delete s2[j]

);

}

}

}

// The result for the entire strings is in dp[0][0]

return dp[0][0];

}

}

**Explanation:**

Base Case:

If one string is exhausted, we simply sum the ASCII values of the remaining characters from the other string.

For dp[n][j], this corresponds to deleting all remaining characters in s2 when s1 is exhausted.

For dp[i][m], this corresponds to deleting all remaining characters in s1 when s2 is exhausted.

Filling the DP Table:

We fill the table from the bottom-right to the top-left.

If the characters s1[i] and s2[j] match, we move diagonally (no deletion needed).

If they don't match, we consider two options:

Deleting s1[i] and adding its ASCII value.

Deleting s2[j] and adding its ASCII value.

We take the minimum of the two possible deletion costs.

Final Result:

The final result is stored in dp[0][0], which represents the minimum delete sum for transforming the entire strings s1 and s2.

**Time Complexity:**

Time Complexity: O(n \* m), where n is the length of s1 and m is the length of s2.

Space Complexity: O(n \* m) due to the 2D DP table.

**Example:**

For the input:

s1 = "sea"

s2 = "eat"

The correct output is 231, which the updated solution now returns.

**Breakdown:**

LCS: "ea".

Characters to delete:

Delete 's' from s1 (ASCII value = 115).

Delete 't' from s2 (ASCII value = 116).

Total minimum delete sum = 115 + 116 = 231.

Now the solution should work correctly for this test case!

**Refer to**

<https://leetcode.com/problems/minimum-ascii-delete-sum-for-two-strings/solutions/1303128/dp-approach-easy-explanation-with-diagram/>

Here we will use dynamic programming tabluation method to solve this problem.

We will create a 2D matrix with n+1 and m+1 size and will initialize first row and first column using s1 and s2.

So this 0th row/column repesents an empty string(""), let's take an example.

case 1:

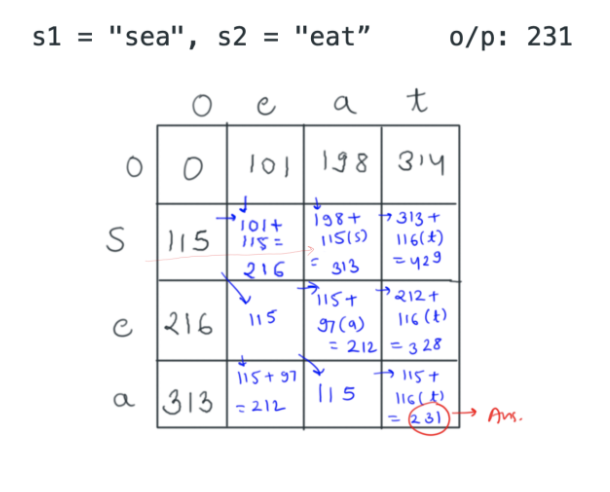
s1="" and s2="e", then we just have to delete e to make both string equal, so our result will be 101.

case 2:

s1="" and s2="ea", then we'll have to delete both "e" and "a" , so our result will be 101+97 => 198.

and in the similar way,

if s1="s" and s2="", then we will have to delete "s", so our output will be 115.



public int minimumDeleteSum(String s1, String s2) {

int m = s1.length();

int n = s2.length();

int[][] dp = new int[m+1][n+1];

//initialize first column with s1 values

for(int i=1; i<=m; i++){

dp[i][0] = dp[i-1][0] + s1.charAt(i-1);

}

//initialize first row with s2 values

for(int i=1; i<=n; i++){

dp[0][i] = dp[0][i-1] + s2.charAt(i-1);

}

for(int i=1; i<=m; i++){

for(int j=1; j<=n; j++){

// if both characters are equal then use previous diagonal value

if(s1.charAt(i-1) == s2.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}else{

dp[i][j] = Math.min(

dp[i-1][j] + s1.charAt(i-1),

dp[i][j-1] + s2.charAt(j-1)

);

}

}

}

return dp[m][n];

}

**Refer to**

<https://leetcode.com/problems/minimum-ascii-delete-sum-for-two-strings/solutions/108811/java-dp-with-explanation/>

Very Similar to Longest Common Subsequence Problem.

Let, s1 & s2 be the two strings with 1 based indexes.

Now assume, dp[i][j] = minimumDeleteSum( s1[0,i], s2[0,j])

**Base case**:

When either of the strings is empty, then whole of the other string has to be deleted.

for e.g. if s1 = "", s2 = "abc", then only way we could match these strings by deleting characters is by dropping 'a','b','c' of s2 to make it empty like s1.

Thus, whenever one of them is empty(i.e. i==0 or j==0) then answer is sum of ASCII code of the characters of the other string.

Hence the **1st** rule: **dp[i][j] =**

sum\_ascii(s2) -> if i==0

sum\_ascii(s1) -> if j==0

**Non-Base case**

Of the two strings, if both of their last characters match then certainly the answer comes from skipping those characters.

i.e. Answer("zca","bza") = Answer("zc","bz")

Hence the **2nd** rule: **dp[i][j] =**

dp[i-1][j-1] -> if s1[i]==s2[j]

Finally, if the last characters are different then its one of the three situations:

drop s1's last character (ASCII(s1's last) + dp[i-1][j])

drop s2's last character (ASCII(s2's last) + dp[i][j-1])

drop both last characters (ASCII(s1's last) + ASCII(s2's last) + dp[i-1[[j-1])

Hence the **3rd** rule: **dp[i][j] =**

Min((ASCII(s1's last) + dp[i-1][j]),(ASCII(s2's last) + dp[i][j-1]),(ASCII(s1's last) + ASCII(s2's last) + dp[i-1[[j-1]))

Combining these 3 rules gives us an elegant solution.

public int minimumDeleteSum(String s1, String s2) {

int m = s1.length();

int n = s2.length();

int[][] dp = new int[m+1][n+1];

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

for(int j=0;j<=n;j++){

if(i==0 || j==0){

int a = 0;

for(int z=1;z<=Math.max(j,i);z++){

a += (i==0?s2.charAt(z-1):s1.charAt(z-1));

}

dp[i][j] = a;

}

else if(s1.charAt(i-1)==s2.charAt(j-1)){

dp[i][j] = dp[i-1][j-1];

}

else{

dp[i][j] = Math.min(s1.charAt(i-1)+dp[i-1][j],s2.charAt(j-1)+dp[i][j-1]);

dp[i][j] = Math.min(dp[i][j],s1.charAt(i-1)+s2.charAt(j-1)+dp[i-1][j-1]);

}

}

}

return dp[m][n];

}

**Refer to**

[L72.Edit Distance (Ref.L115)](note://998ECECFA22D4E1E9533D9AD624A6345)

[L583.Delete Operation for Two Strings (Ref.L712,L72,L1143)](note://WEB12ebad84cb577589b9576d7f802df5ee)

[L1143.Longest Common Subsequence (Ref.L516,L583,L712)](note://F9909063BABB4A67B28268E5F3054AED)

[L673.Number of Longest Increasing Subsequence (Ref.L300)](note://BB31AB0B228C46F4A4C32F0148810E70)