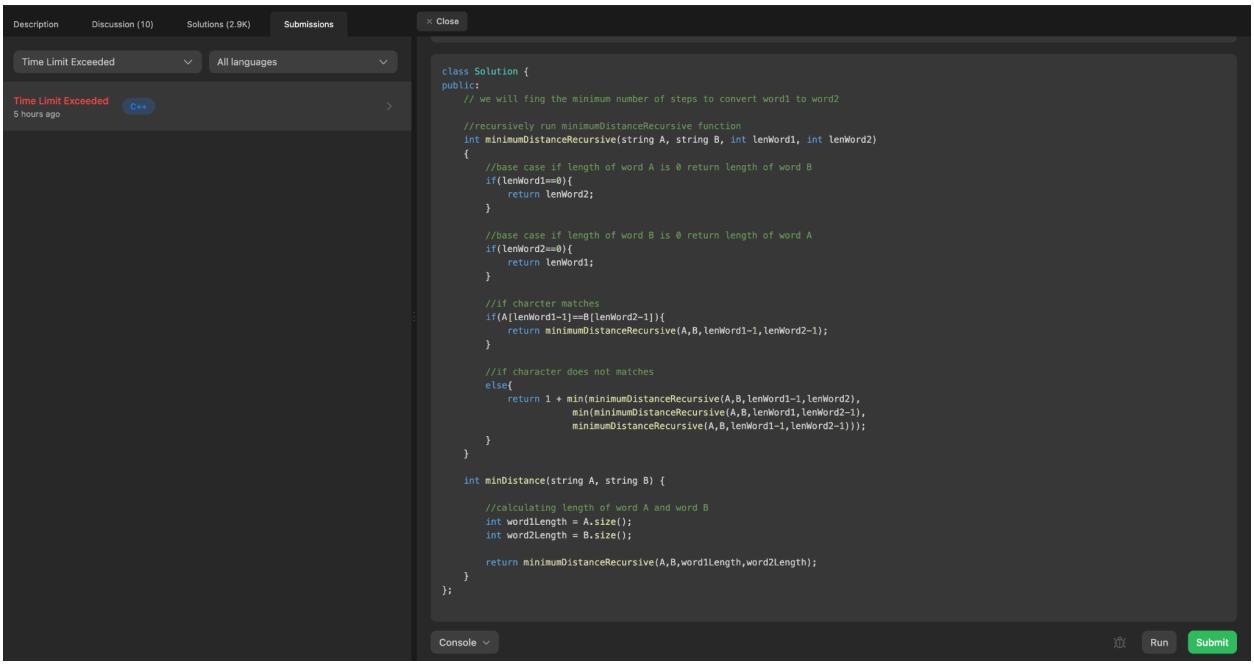
Implement the edit distance algorithm on Leetcode.
Link: https://leetcode.com/problems/edit-distance/
You need to do the following:
1) provide your code in the homework submission,
2) after your code successfully passed all the test cases when you click on the “submit” button, Click on the “details” button on the top left, you will see a report of the performance of your code (which would look similar to the picture below). Include that in your submission.

For the 5th question, we had initially used a recursion method to solve the same. We got a time limit exceeded as you can see in the screenshot below.

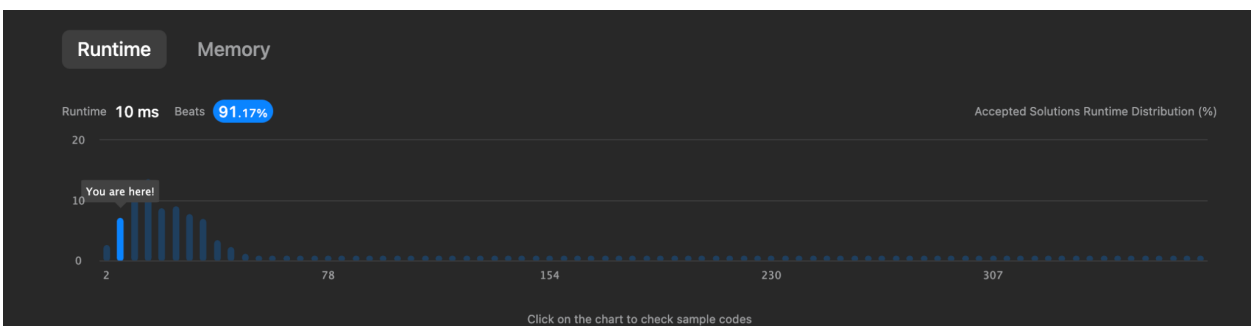


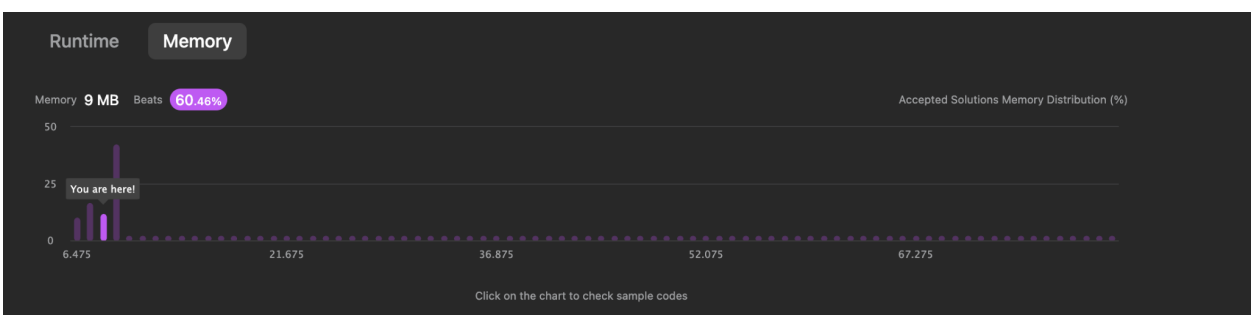
Then, we tried changing our technique to memoization and we were able to achieve the following results.

```cpp
class Solution {
public:
    // we will fing the minimum number of steps to convert word1 to word2

    //recursively run minimumDistanceRecursive function
    int minimumDistanceRecursive(int word1Length, int word2Length, string &wordA, string &wordB, vector<vector<int>>&table)     {
            //if length of word 2 is empty
            if(word2Length<0){
                return word1Length+1;
            }
            //if length of word 1 is empty
            if(word1Length<0){
                return word2Length+1;
            }
            //if length of word 2 is empty
            if(table[word1Length][word2Length]!=-1){
                return table[word1Length][word2Length];
            }
            //if character matches
            if(wordA[word1Length]==wordB[word2Length]){
                return table[word1Length][word2Length] = minimumDistanceRecursive(word1Length-1,word2Length-1,wordA,wordB,table);
            }
            //if character does not match
            else{
                int insertOperationCost = 1 + minimumDistanceRecursive(word1Length,word2Length-1,wordA,wordB,table);
                int replaceOperationCost = 1 + minimumDistanceRecursive(word1Length-1,word2Length-1,wordA,wordB,table);
                int deletedOperationCost = 1 + minimumDistanceRecursive(word1Length-1,word2Length,wordA,wordB,table);
                return table[word1Length][word2Length] = min(min(insertOperationCost,deletedOperationCost),replaceOperationCost);
            }
    }

    int minDistance(string A, string B) {

        //calculating length of word A and word B
        int word1Length = A.size();
        int word2Length = B.size();

        //Recursion with memoization
        vector<vector<int>>table(word1Length,vector<int>(word2Length,-1));
        return minimumDistanceRecursive(word1Length-1,word2Length-1,A,B,table);
    }
};
```

Console

Run Submit





Later, in order to optimize our outputs we used dynamic programming taught in class and implemented it practically. We were able to achieve the following results.

```
class Solution {
public:
    // we will fing the minimum number of steps to convert word1 to word2

    int minDistance(string A, string B) {

    //word1Length stores length of A
    //word2Length stores length of B
    int word1Length=A.length();
    int word2Length=B.length();

    //creating DP table
    int table[word1Length+1][word2Length+1];

    for(int i=0;i<=word1Length;i++)
    {
        for(int j=0;j<=word2Length;j++)
        {
            //base case (for row 0)
            if(i==0)
            {
                table[i][j]=j;
            }
            //base case (for column 0)
            else if(j==0)
            {
                table[i][j]=i;
            }
            //if character matches copy the diagonal value
            else if(A[i-1]==B[j-1])
            {
                table[i][j]=table[i-1][j-1];
            }
            //if character does not match take minimum of upper, left and diagonal value
            else
            {
                table[i][j]=1+min(table[i][j-1],min(table[i-1][j],table[i-1][j-1]));
            }
        }
    }
    //return result
    return table[word1Length][word2Length];
    }
};
```

Console Run Submit