**Edit Distance**

**We'll cover the following**

* + - [Problem Statement](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#problem-statement)
    - [Basic Solution](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#basic-solution)
      * [Code](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#code)
    - [Top-down Dynamic Programming with Memoization](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#top-down-dynamic-programming-with-memoization)
      * [Code](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#code-2)
    - [Bottom-up Dynamic Programming](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#bottom-up-dynamic-programming)
      * [Code](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#code-3)

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#problem-statement)

Given strings s1 and s2, we need to transform s1 into s2 by deleting, inserting, or replacing characters. Write a function to calculate the count of the minimum number of edit operations.

**Example 1:**

Input: s1 = "bat"  
       s2 = "but"  
Output: 1  
Explanation: We just need to replace 'a' with 'u' to transform s1 to s2.

**Example 2:**

Input: s1 = "abdca"  
       s2 = "cbda"  
Output: 2  
Explanation: We can replace first 'a' with 'c' and delete second 'c'.

**Example 3:**

Input: s1 = "passpot"  
       s2 = "ppsspqrt"  
Output: 3   
Explanation: Replace 'a' with 'p', 'o' with 'q', and insert 'r'.

**Basic Solution**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#basic-solution)

A basic brute-force solution could be to try all operations (one by one) on each character of s1. We can iterate through s1 and s2 together. Let’s assume index1 and index2 point to the current indexes of s1 and s2 respectively, so we have two options at every step:

1. If the strings have a matching character, we can recursively match for the remaining lengths.
2. If the strings don’t match, we start three new recursive calls representing the three edit operations. Whichever recursive call returns the minimum count of operations will be our answer.

**Code**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#code)

Here is recursive implementation:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

  }

  private int findMinOperationsRecursive(String s1, String s2, int i1, int i2) {

    // if we have reached the end of s1, then we have to insert all the remaining characters of s2

    if(i1 == s1.length())

      return s2.length() - i2;

    // if we have reached the end of s2, then we have to delete all the remaining characters of s1

    if(i2 == s2.length())

      return s1.length() - i1;

    // If the strings have a matching character, we can recursively match for the remaining lengths.

    if(s1.charAt(i1) == s2.charAt(i2))

      return findMinOperationsRecursive(s1, s2, i1+1, i2+1);

    int c1 = 1 + findMinOperationsRecursive(s1, s2, i1+1, i2); //perform deletion

    int c2 = 1 + findMinOperationsRecursive(s1, s2, i1, i2+1); //perform insertion

    int c3 = 1 + findMinOperationsRecursive(s1, s2, i1+1, i2+1); // perform replacement

    return  Math.min(c1, Math.min(c2, c3));

  }

  public static void main(String[] args) {

    EditDistance editDisatnce = new EditDistance();

    System.out.println(editDisatnce.findMinOperations("bat", "but"));

    System.out.println(editDisatnce.findMinOperations("abdca", "cbda"));

    System.out.println(editDisatnce.findMinOperations("passpot", "ppsspqrt"));

  }

}





RUN

SAVERESET

Because of the three recursive calls, the time complexity of the above algorithm is exponential O(3^{m+n})*O*(3​*m*+*n*​​), where ‘m’ and ‘n’ are the lengths of the two input strings. The space complexity is O(n+m)*O*(*n*+*m*) which is used to store the recursion stack.

**Top-down Dynamic Programming with Memoization**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#top-down-dynamic-programming-with-memoization)

We can use an array to store the already solved subproblems.

The two changing values in our recursive function are the two indexes, i1 and i2. Therefore, we can store the results of all the subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (i1 + “|” + i2)).

**Code**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#code-2)

Here is the code for Top-down DP approach:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

    if(dp[i1][i2] == null) {

      // if we have reached the end of s1, then we have to insert all the remaining characters of s2

      if(i1 == s1.length())

        dp[i1][i2] = s2.length() - i2;

      // if we have reached the end of s2, then we have to delete all the remaining characters of s1

      else if(i2 == s2.length())

        dp[i1][i2] = s1.length() - i1;

      // If the strings have a matching character, we can recursively match for the remaining lengths

      else if(s1.charAt(i1) == s2.charAt(i2))

        dp[i1][i2] = findMinOperationsRecursive(dp, s1, s2, i1+1, i2+1);

      else {

        int c1 = findMinOperationsRecursive(dp, s1, s2, i1+1, i2); //delete

        int c2 = findMinOperationsRecursive(dp, s1, s2, i1, i2+1); //insert

        int c3 = findMinOperationsRecursive(dp, s1, s2, i1+1, i2+1); //replace

        dp[i1][i2] = 1 + Math.min(c1, Math.min(c2, c3));

      }

    }

    return dp[i1][i2];

  }

  public static void main(String[] args) {

    EditDistance editDisatnce = new EditDistance();

    System.out.println(editDisatnce.findMinOperations("bat", "but"));

    System.out.println(editDisatnce.findMinOperations("abdca", "cbda"));

    System.out.println(editDisatnce.findMinOperations("passpot", "ppsspqrt"));

  }

}





RUN

SAVERESET

**What is the time and space complexity of the above solution?** Since our memoization array dp[s1.length()][s2.length()] stores the results for all the subproblems, we can conclude that we will not have more than m\*n*m*∗*n* subproblems (where ‘m’ and ‘n’ are the lengths of the two input strings.). This means that our time complexity will be O(m\*n)*O*(*m*∗*n*).

The above algorithm will be using O(m\*n)*O*(*m*∗*n*) space for the memoization array. Other than that we will use O(m+n)*O*(*m*+*n*) space for the recursion call-stack. So the total space complexity will be O(m\*n + (m+n))*O*(*m*∗*n*+(*m*+*n*)), which is asymptotically equivalent to O(m\*n)*O*(*m*∗*n*).

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#bottom-up-dynamic-programming)

Since we want to match all the characters of the given two strings, we can use a two-dimensional array to store our results. The lengths of the two strings will define the size of the two dimensions of the array. So for every index ‘i1’ in string ‘s1’ and ‘i2’ in string ‘s2’, we will choose one of the following options:

1. If the character s1[i1] matches s2[i2], the count of the edit operations will be equal to the count of the edit operations for the remaining strings.
2. If the character s1[i1] does not match s2[i2], we will take the minimum count from the remaining strings after performing any of the three edit operations.

So our recursive formula would be:

1

2

3

4

5

6

if s1[i1] == s2[i2]

  dp[i1][i2] = dp[i1-1][i2-1]

else

  dp[i1][i2] = 1 + min(dp[i1-1][i2], // delete

                       dp[i1][i2-1], // insert

                       dp[i1-1][i2-1]) // replace





**Code**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY#code-3)

Here is the code for our bottom-up dynamic programming approach:

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gx2QMvEorYY)

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

    // if s2 is empty, we can remove all the characters of s1 to make it empty too

    for(int i1=0; i1 <= s1.length(); i1++)

      dp[i1][0] = i1;

    // if s1 is empty, we have to insert all the characters of s2

    for(int i2=0; i2 <= s2.length(); i2++)

      dp[0][i2] = i2;

    for(int i1=1; i1 <= s1.length(); i1++) {

      for(int i2=1; i2 <= s2.length(); i2++) {

        // If the strings have a matching character, we can recursively match for the remaining lengths

        if(s1.charAt(i1-1) == s2.charAt(i2-1))

          dp[i1][i2] = dp[i1-1][i2-1];

        else

          dp[i1][i2] = 1 + Math.min(dp[i1-1][i2], //delete

                            Math.min(dp[i1][i2-1], //insert

                                     dp[i1-1][i2-1])); //replace

      }

    }

    return dp[s1.length()][s2.length()];

  }

  public static void main(String[] args) {

    EditDistance editDisatnce = new EditDistance();

    System.out.println(editDisatnce.findMinOperations("bat", "but"));

    System.out.println(editDisatnce.findMinOperations("abdca", "cbda"));

    System.out.println(editDisatnce.findMinOperations("passpot", "ppsspqrt"));

  }

}





RUN

SAVERESET

The time and space complexity of the above algorithm is O(n\*m)*O*(*n*∗*m*), where ‘m’ and ‘n’ are the lengths of the two input strings.

**MARK AS COMPLETED**

[**←    Back**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/gxp0yR37zBr)

Longest Alternating Subsequence

[**Next    →**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/RMM68YXRwEK)

Strings Interleaving

Report an Issue

[Ask a Question](https://discuss.educative.io/c/grokking-dynamic-programming-patterns-for-coding-interviews-design-gurus/pattern-5-longest-common-substring-edit-distance" \t "_blank)