

Benevento

## **2. Common Substring**

```
FUNCTION CommonSubstring(text1, text2)
CREATE
//dp table: a 2D array (text1 length + 1) x (text2 length + 1). + 1 to each length is to handle the empty
string case.

dp[][] = dp[text1 length + 1][text2 length + 1]
int maxStr = 0 //counter to keep track of the longest substring length (maximum string length)
FOR i from 1 to text.length
    FOR j from 1 to text.length
        IF text1[i - 1] == text2[j - 1]
            dp[i][j] = dp[i - 1][j - 1] + 1 //add 1 to matching string

        IF dp[i][j] > maxStr
            maxStr = dp[i][j] //if current string is larger than the max, update it

        ELSE
            dp[i][j] = 0 //reset to 0 if the next character does not match
RETURN maxStr
END
```

```
/*
Sources:
https://www.youtube.com/watch?v=6ZbPtpMpC5k
https://www.youtube.com/watch?v=BysNXJHzCEs
*/
```

# Benevento

## 5. Remove Elements, Extra Credit

Remove Element - LeetCode

Description Editorial Solutions Submissions

```
assert k == expectedNums.length;
sort(nums, 0, k); // Sort the first k elements of nums
for (int i = 0; i < actualLength; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

**Example 1:**

**Input:** nums = [3,2,2,3], val = 3  
**Output:** 2, nums = [2,2,\_,\_]  
**Explanation:** Your function should return k = 2, with the first two elements of nums being 2.  
It does not matter what you leave beyond the returned k (hence they are underscores).

**Example 2:**

**Input:** nums = [0,1,2,2,3,0,4,2], val = 2  
**Output:** 5, nums = [0,1,4,0,3,\_,\_,\_]  
**Explanation:** Your function should return k = 5, with the first five elements of nums containing 0, 0, 1, 3, and 4.  
Note that the five elements can be returned in any order.  
It does not matter what you leave beyond the returned k (hence they are underscores).

**Constraints:**

- $0 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 50$
- $0 \leq \text{val} \leq 100$

Java Auto

```
1 class Solution {
2     public int removeElement(int[] nums, int val) {
3         int k = 0; //length counter
4
5         for (int i = 0; i < nums.length; i++){
6             if(nums[i] != val){
7                 nums[k] = nums[i];
8                 k++;
9             }
10        }
11
12        return k;
13    }
14}
15
16 }
```

Restored from local. Upgrade to Cloud Saving

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[3,2,2,3]
```

val =  
3

Output

```
[2,2]
```

Expected

jbenevento Access all features with our Premium subscription!

My Lists Notebook Progress Points Try New Features Orders My Playgrounds Settings Appearance Sign Out

## **6. (text) Algorithm Analysis**

### **Common Subsequence**

Time Complexity is  $O(n*m)$  where n is the length of the string stored in text1 and m is the length of text2 because the nested loops iterate through every character in each string to create the dynamic programming table with n number of rows and m number of columns.

Space Complexity is  $O(n*m)$  where n is the length of the string stored in text1 and m is the length of text2 because it is creating a dynamic programming table with  $(n + 1) * (m + 1)$  elements. An integer is stored at each element.

```
int dp[][] = new int[text1.length() + 1][text2.length() + 1]; //O(n * m)
for (int i = 1; i <= text1.length(); i++) { //O(n)
    for (int j = 1; j <= text2.length(); j++) { //O(m)
        if (text1.charAt(i - 1) == text2.charAt(j - 1)) { //O(1)
            dp[i][j] = 1 + dp[i - 1][j - 1]; //O(1)
        } else { dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]); //O(1) }
    }
}
```

Array declaration/assignment =  $O(n*m)$   
Outer loop =  $O(n)$  Inner loop =  $O(m) * O(3) = O(3m)$   
 $O(nm) + O(n) * O(3m) = O(nm) + O(nm) = O(nm)$

### Sources:

- <https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-1-d-and-2-d-array-operations/>
- <https://stackoverflow.com/questions/68400626/big-o-notation-how-do-you-describe-a-2d-array-of-different-lengths>

Benevento

### Common Substring

Time Complexity is  $O(n*m)$  where  $n$  is the length of the string stored in  $text1$  and  $m$  is the length of  $text2$  because the nested loops iterate through every character in each string to create the dynamic programming table with  $n$  number of rows and  $m$  number of columns.

Space Complexity is  $O(n*m)$  where  $n$  is the length of the string stored in  $text1$  and  $m$  is the length of  $text2$  because it is creating a dynamic programming table with  $(n + 1) * (m + 1)$  elements. An integer is stored at each element.

```
FUNCTION CommonSubstring(text1, text2)
    CREATE dp[ text1.length + 1 ][ text2.length + 1 ] //O(n*m)
    int maxStr = 0 //O(1)

    FOR i from 1 to text1.length //O(n)
        FOR j from 1 to text2.length //O(m)
            IF text1[i - 1] == text2[j - 1] //O(1)
                dp[i][j] = dp[i - 1][j - 1] + 1 //O(1)
                IF dp[i][j] > maxStr //O(1)
                    maxStr = dp[i][j] //O(1)
            ELSE
                dp[i][j] = 0 //O(1)

    RETURN maxStr //O(1)
END
```

### Not Fibonacci

Time Complexity is  $O(N)$ , where  $n$  is the length of the sequence array, because the loop will iterate through the array once.

Space Complexity is  $O(n)$  where  $n$  is the input size because we create an array of length  $n$  to store the sequence values.

```
long[]sequence = new long[n]; //O (n)
sequence[0] = 0; //O(1)
sequence[1] = 1; //O(1)

for (int i = 2; i < n; ++i) { //O(n-2)
    double newTerm = (3 * sequence[i-1]) + (2 * sequence[i-2]); //O(1)
    sequence[i] = (long) Math.floor(newTerm); //O(1)
}

return sequence; //O(1)
```

## Where in Sequence

Time Complexity is  $O(N)$ , where  $n$  is the length of the Not Fibonacci sequence, because the loop might need to iterate through the entire sequence to find the input value. For the best case, Time Complexity is  $\Omega(1)$  if it finds the input value on the first iteration.

Space Complexity is  $O(1)$  because there is only one variable created and stored. The variable is  $i$ .

```
for(int i = 0; i < sequence.length - 1; i++){ //O(n)
    if (input == sequence[i]) { //O(1)
        return i; //index where input is located //O(1)
    } else if(input < sequence[i]){ //O(1)
        return i - 1; //return position of the closet number lower than the input //O(1)
    }
}
return -1; //O(1)
```

## Remove Elements

Time Complexity is  $O(N)$ , where  $n$  is the length of the array of doubles, because the loop needs to iterate through the entire array searching for any and all values that match the target input to be removed.

Space Complexity is  $O(1)$  because two variables are created and used. The variables are  $k$  and  $i$ .

```
int k = 0; //counter O(1)
for (int i = 0; i < nums.length; i++){ //O(n)
    if(nums[i] >= val) { //O(1)
        nums[k] = nums[i]; //O(1)
        k++; //O(1)
    }
}
return k;
```