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

Write a function that reverses a string. The input string is given as an array of characters s.

You must do this by modifying the input array [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Example 1:**

**Input:** s = ["h","e","l","l","o"]

**Output:** ["o","l","l","e","h"]

**Example 2:**

**Input:** s = ["H","a","n","n","a","h"]

**Output:** ["h","a","n","n","a","H"]

**Constraints:**

1 <= s.length <= 105

s[i] is a [printable ascii character](https://en.wikipedia.org/wiki/ASCII#Printable_characters).

**Attempt 1: 2025-07-01**

**Solution 1: Two Pointers (10 min)**

class Solution {

    public void reverseString(char[] s) {

        int i = 0;

        int j = s.length - 1;

        while (i < j) {

            char temp = s[i];

            s[i] = s[j];

            s[j] = temp;

            i++;

            j--;

        }

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Solution 2: Recursion (10 min)**

**Style 1: Traversal (Do something first then recursion)**

class Solution {

    public void reverseString(char[] s) {

        helper(s, 0, s.length - 1);

    }

    private void helper(char[] s, int i, int j) {

        if(i >= j) {

            return;

        }

// Do something first

        char tmp = s[i];

        s[i] = s[j];

        s[j] = tmp;

// Recursion

        helper(s, i + 1, j - 1);

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: Divide and Conquer (Recursion first then do something)**

**Note: Because of void return required, we cannot apply traditional Divide and Conquer strategy here**

class Solution {

public void reverseString(char[] s) {

dncReverse(s, 0, s.length - 1);

}

private void dncReverse(char[] s, int start, int end) {

int n = end - start + 1;

if (n <= 1) return;

int half = n / 2;

int mid = start + half - 1;

// Divide: Recursively reverse halves

dncReverse(s, start, mid);

dncReverse(s, end - half + 1, end);

// Conquer: Swap blocks (O(n) operation)

// And because we cannot directly concatenate substrings as void return,

// so the for loop style is required, which not equals traditional Conquer

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

char temp = s[start + i];

s[start + i] = s[end - half + 1 + i];

s[end - half + 1 + i] = temp;

}

}

}

Time Complexity: O(nlogn)

Space Complexity: O(nlogn)

**Refer to Deepseek**

**Problem Statement**

Write a function that reverses a string. The input string is given as an array of characters s. You must do this by modifying the input array in-place with O(1) extra memory.

**Approach**

The solution uses a two-pointer approach to reverse the string in-place:

Initialize two pointers, one at the start (left) and one at the end (right) of the array

Swap the characters at these pointers

Move the left pointer forward and the right pointer backward

Repeat until the pointers meet in the middle

**Solution Code**

class Solution {

public void reverseString(char[] s) {

int left = 0;

int right = s.length - 1;

while (left < right) {

// Swap characters

char temp = s[left];

s[left] = s[right];

s[right] = temp;

// Move pointers

left++;

right--;

}

}

}

**Explanation**

The algorithm starts with two pointers at opposite ends of the array.

In each iteration, it swaps the characters at these pointers and moves them towards the center.

The loop continues until the pointers meet or cross each other, ensuring all characters are swapped.

This approach efficiently reverses the string in O(n) time complexity with O(1) space complexity, as it only uses a constant amount of extra space for the temporary variable during swaps.

This solution is optimal for the problem constraints and demonstrates an efficient way to perform in-place array manipulation.

**Do we have a recursive solution ?**

Yes, we can solve this problem recursively as well. Here's a recursive approach to reverse the string in-place:

**Recursive Approach**

Define a helper function that takes the character array and two indices (left and right)

Base case: if left >= right, return (we've reached or passed the middle)

Recursive case:

Swap the characters at left and right indices

Recursively call the function with left+1 and right-1

**Solution Code**

class Solution {

public void reverseString(char[] s) {

reverseHelper(s, 0, s.length - 1);

}

private void reverseHelper(char[] s, int left, int right) {

if (left >= right) {

return;

}

// Swap characters

char temp = s[left];

s[left] = s[right];

s[right] = temp;

// Recursively reverse the substring

reverseHelper(s, left + 1, right - 1);

}

}

**Explanation**

The reverseString method initiates the recursion by calling the helper with the full array range (0 to length-1)

The reverseHelper method:

Handles the base case where no more swaps are needed

Performs the swap for the current left and right positions

Recursively processes the next inner pair of characters

Each recursive call moves the left pointer forward and right pointer backward until they meet in the middle

**Note on Recursion**

**While this recursive solution works, it's important to note:**

**It uses O(n) space in the call stack (implicit memory) due to recursion depth**

**The iterative solution is generally preferred for this problem as it has O(1) space complexity**

**Recursion depth is limited by the stack size, which could be a problem for very large strings**

**The recursive approach is elegant but less efficient in terms of space compared to the iterative solution.**

**Time Complexity Analysis for the Recursive Solution**

The time complexity of the recursive solution for reversing a string is **O(n)**, where **n** is the length of the input string/array.

**Detailed Explanation:**

**Base Case**: The recursion stops when left >= right, which happens after approximately n/2 recursive calls.

**Work Per Recursive Call**: Each recursive call performs:

A constant-time comparison (left >= right)

A constant-time swap operation (3 assignments)

Two pointer adjustments (increment/decrement)

**Total Operations**:

For a string of length n, we make n/2 recursive calls

Each call does O(1) work

Total work = (n/2) \* O(1) = O(n)

**Space Complexity:**

While the time complexity is O(n), the space complexity is also O(n) due to:

The recursion stack which grows to depth n/2 ≈ O(n) in the worst case

This is less efficient than the iterative O(1) space solution

**Comparison with Iterative Solution:**

**Iterative**: O(n) time, O(1) space

**Recursive**: O(n) time, O(n) space (due to call stack)

The recursive solution has the same time complexity but worse space complexity because each recursive call adds a new stack frame, while the iterative solution just uses a few variables.

**I have seen kind of "Divide and Conquer" style solution like below, but for my understanding it requires return type not as void, right ? Currently the problem described return type is limited as void only, so our recursive solution cannot use this "Divide and Conquer" style, right ?**

public class Solution {

public String reverseString(String s) {

int length = s.length();

if(length <= 1) {

return s;

}

String leftString = s.substring(0, length / 2);

String rightString = s.substring(length / 2, length);

return reverseString(rightString) + reverseString(leftString);

}

}

You're absolutely right to notice this distinction! Let me clarify the key differences between the approaches:

**Your Observation is Correct**

**Problem Constraints**:

The LeetCode problem requires modifying the input char[] in-place with a void return type

Your shown "Divide and Conquer" solution returns a new String (not in-place)

**Key Differences**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Approach | Return Type | Modifies Input | Space Complexity | Suitable for LeetCode 344? |
| Your Divide & Conquer | String | No (creates new strings) | O(n log n) due to string copies | ❌ No (wrong signature) |
| In-place Recursive | void | Yes | O(n) call stack | ✅ Yes |
| In-place Iterative | void | Yes | O(1) | ✅ Yes (best) |

**Why the Divide & Conquer Style Doesn't Work Here**

**String vs char[]**:

The problem gives char[] but your solution works with String

Strings are immutable in Java, requiring new allocations

**Not In-Place**:

Each substring creates new strings

The concatenation (+) creates even more strings

This violates the O(1) space requirement

**Performance**:

Your approach does O(n log n) string copying work

The call stack is O(log n), but dominated by string operations

**What You Could Do Instead**

If you really wanted a Divide & Conquer approach that fits the problem constraints:

class Solution {

public void reverseString(char[] s) {

divideAndConquerReverse(s, 0, s.length - 1);

}

private void divideAndConquerReverse(char[] s, int start, int end) {

if (start >= end) return;

int mid = start + (end - start) / 2;

// Reverse left half

divideAndConquerReverse(s, start, mid);

// Reverse right half

divideAndConquerReverse(s, mid + 1, end);

// Merge by swapping halves

while (start < end) {

char temp = s[start];

s[start] = s[end];

s[end] = temp;

start++;

end--;

}

}

}

But this is unnecessarily complicated for this problem - the simple two-pointer approach is better. The true Divide & Conquer shines in problems like "Reverse Words in a String II" (LeetCode 186) where you need to reverse parts at different levels.

**But your above solution is wrong for below input:**

**Input**

**s = ["h","e","l","l","o"]**

**Output = ["l","o","e","h","l"]**

**Expected = ["o","l","l","e","h"]**

You're absolutely right! My previous Divide & Conquer implementation was incorrect - it doesn't properly reverse the string. Let me explain why it fails and how to fix it.

**Why the Previous Solution Failed**

For input ['h','e','l','l','o']:

First splits into ['h','e'] and ['l','l','o']

Recursively reverses them to ['e','h'] and ['o','l','l']

Then incorrectly swaps the halves producing ['o','l','e','h','l']

**Correct Divide & Conquer Approach**

Here's a proper implementation that actually reverses the string:

class Solution {

public void reverseString(char[] s) {

reverseHelper(s, 0, s.length - 1);

}

private void reverseHelper(char[] s, int left, int right) {

if (left >= right) return;

// Swap the outermost characters

char temp = s[left];

s[left] = s[right];

s[right] = temp;

// Recursively reverse the inner substring

reverseHelper(s, left + 1, right - 1);

}

}

**Key Differences:**

**No Midpoint Splitting**:

The correct version doesn't split the array in half

It works from both ends towards the center

**Proper Swapping**:

Always swaps the outermost characters first

Then processes what's between them

**Correct Behavior**:

['h','e','l','l','o'] becomes ['o','l','l','e','h'] as expected

Each recursive call handles one pair of characters

**Why This Works:**

Base Case: When pointers meet/cross

Recursive Case:

Swap first and last characters

Recurse on the substring between them

This is essentially the recursive version of the standard two-pointer approach, just using the call stack instead of a loop.

**Time/Space Complexity:**

Time: O(n) - Each character is swapped exactly once

Space: O(n) for the call stack (but O(1) extra memory besides that)

The iterative solution is still better for this problem, but this shows how to correctly implement a recursive reversal.

**Please re-think once, your new fixed solution is working but its not follow the Divide and Conquer style as the solution i provided, you have to recursive the main problem to sub-problem first (which is Divide step), then calculate on the minimum sub-problem then merge (which is Conquer step), but in your fixed solution its actually following the regular recursive solution as Conquer step first then Divide step, in simple, i need you create a typical Divide step before Conquer step solution but match the void return requirement on the problem, if not possible, explain why**

**You've highlighted a crucial point. The classic Divide and Conquer (D&C) approach for string reversal isn't directly applicable to LeetCode 344's constraints (in-place modification with void return type). Here's a detailed explanation:**

**Why the Classic D&C Approach Doesn't Work In-Place**

**Division Creates Subarrays**:

The D&C approach naturally divides the array into subarrays (left/right halves). This requires:

Creating new subarrays (violates in-place requirement)

Additional O(n) space for substring operations (violates O(1) space)

**Combine Step Requires Concatenation**:

**Merging reversed substrings (reverse(right) + reverse(left)) creates new strings, which:**

**Requires returning a new string (violates void return type)**

Uses O(n) space for concatenation

**Swapping Halves of Unequal Length is Problematic**:

For odd-length arrays (e.g., 5 elements):

Left half = 3 elements, right half = 2 elements

Cannot directly swap blocks of different sizes without complex rotations

**Attempt at an In-Place D&C Solution**

While technically possible, this approach is inefficient and complex:

class Solution {

public void reverseString(char[] s) {

dncReverse(s, 0, s.length - 1);

}

private void dncReverse(char[] s, int start, int end) {

int n = end - start + 1;

if (n <= 1) return;

int half = n / 2;

int mid = start + half - 1;

// Divide: Recursively reverse halves

dncReverse(s, start, mid);

dncReverse(s, end - half + 1, end);

// Conquer: Swap blocks (O(n) operation)

// And because we cannot directly concatenate substrings as void return,

// so the for loop style is required, which not equals traditional Conquer

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

char temp = s[start + i];

s[start + i] = s[end - half + 1 + i];

s[end - half + 1 + i] = temp;

}

}

}

**Test Case: ["h","e","l","l","o"]**

**First recursive call** ([0,4]):

half = 2

Reverse left block [0,1]: ["e","h","l","l","o"]

Reverse right block [3,4]: ["e","h","l","o","l"]

Swap blocks: ["o","l","l","e","h"] ✅

**But time complexity is O(n log n)** due to:

T(n) = 2T(n/2) + O(n)

(Master Theorem: O(n log n))

**Why This is Impractical**

**Inefficiency**:

Iterative solution: O(n) time

This D&C solution: O(n log n) time

**Unnecessary Complexity**:

Requires careful index management

Still uses O(log n) recursion stack space

Swapping blocks requires O(n) operations per recursion level

**Violates Problem Spirit**:

Problem expects simple O(n) solution

D&C is overkill for linear reversal

**Recommended Approach**

For this specific problem, **iterative two-pointer** or **tail recursion** is optimal:

// Iterative (Best)

public void reverseString(char[] s) {

int left = 0, right = s.length - 1;

while (left < right) {

char temp = s[left];

s[left++] = s[right];

s[right--] = temp;

}

}

// Recursive (Conceptual)

public void reverseString(char[] s) {

reverse(s, 0, s.length - 1);

}

private void reverse(char[] s, int left, int right) {

if (left >= right) return;

char temp = s[left];

s[left] = s[right];

s[right] = temp;

reverse(s, left + 1, right - 1);

}

**Conclusion**

The classic D&C reversal algorithm **cannot satisfy LeetCode 344's constraints** because:

It inherently requires creating new subarrays

Block swapping of unequal lengths is complex

Optimal solution exists without D&C (O(n) time, O(1) space)

Use the iterative two-pointer method for this problem. D&C is better suited for problems like:

Merge Sort (natural division/merging)

Reversing words in a string (LeetCode 186)

Problems where division yields independent subproblems

**Refer to**

[L186.Reverse Words in a String II (Ref.L344)](note://WEB53acf96fbd1f8e2e9b8ae54ca1963033)

[L345.Reverse Vowels of a String](note://CA879E2192A94099996642D7560B6FB9)

[L541.Reverse String II (Ref.L344,L345)](note://WEB9e50c649529029c8c3ea4ef783609472)

[两种递归分治和遍历的区别](note://F9C426C3D53242F4914CB5CA5CF171D7)