目录

[1. Two Sum 2](#_Toc524025519)

[2. Add Two Numbers 2](#_Toc524025520)

[3. Longest Substring Without Repeating Characters 3](#_Toc524025521)

[4. Median of Two Sorted Arrays 4](#_Toc524025522)

[5. Longest Palindromic Substring 6](#_Toc524025523)

[6. ZigZag Conversion 7](#_Toc524025524)

[7. Reverse Integer 8](#_Toc524025525)

[8. String to Integer (atoi) 9](#_Toc524025526)

[9. Palindrome Number 11](#_Toc524025527)

[10. Regular Expression Matching 12](#_Toc524025528)

[11. Container With Most Water 14](#_Toc524025529)

[12. Integer to Roman 15](#_Toc524025530)

[13. Roman to Integer 17](#_Toc524025531)

[14. Longest Common Prefix 19](#_Toc524025532)

[n. xxx 20](#_Toc524025533)

[n. xxx 20](#_Toc524025534)

[n. xxx 20](#_Toc524025535)

[n. xxx 21](#_Toc524025536)

做题方法

思路方法

确定方法——考虑特殊情况（测试）

经验：

能从原字符串中截取的，不要自行重新组装字符串，过程中记录索引最后截取即可。

去重或避免重复往往和排序相关，至少先排序才能更方便去重。

14.66.70.20.26.55.88.89.17.22.46.79.42.53.21

104.191.258.226.461.122.238.202.136.108.253.231

155.206.283.141.198.172.160.349.153.110.

148.328.74.96.134.22.79

123.42.138.33.240.173.179

316.150.139

188.283.238

136.217

206.141.328.160.142

121.122.134.316

349.374.278.153.50.287.33.74.240

123.198.338

104.226.100.235.110.108

136.137.371.191.338.268.231

# 1. Two Sum

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

**Example:**

Given nums = [2, 7, 11, 15], target = 9,

Because nums[**0**] + nums[**1**] = 2 + 7 = 9,

return [**0**, **1**].

思路1：

建立一个哈希表用来存储给定数组的元素和其对应的索引index，遍历一遍数组，看是否已经存在一个项的key等于目标和当前项key的差。

代码1：

public int[] twoSum(int[] nums, int target) {

Map<Integer, Integer> map = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

int complement = target - nums[i];

if (map.containsKey(complement)) {

return new int[] { map.get(complement), i };

}

map.put(nums[i], i);

}

throw new IllegalArgumentException("No two sum solution");

}

复杂度1:

* Time complexity : O(n). We traverse the list containing n elements only once. Each look up in the table costs only O(1) time.
* Space complexity : O(n). The extra space required depends on the number of items stored in the hash table, which stores at most n elements.

# 2. Add Two Numbers

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order** and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example:**

**Input:** (2 -> 4 -> 3) + (5 -> 6 -> 4)

**Output:** 7 -> 0 -> 8

**Explanation:** 342 + 465 = 807.

思路1：

直观理解，按位相加，将进位作一个变量处理。注意特殊情况，如某个数为空、最高位进位、位数不等的两数相加。

代码1：

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

ListNode dummyHead = new ListNode(0);

ListNode p = l1, q = l2, curr = dummyHead;

int carry = 0;

while (p != null || q != null) {

int x = (p != null) ? p.val : 0;

int y = (q != null) ? q.val : 0;

int sum = carry + x + y;

carry = sum / 10;

curr.next = new ListNode(sum % 10);

curr = curr.next;

if (p != null) p = p.next;

if (q != null) q = q.next;

}

if (carry > 0) {

curr.next = new ListNode(carry);

}

return dummyHead.next;

}

复杂度1：

* Time complexity : O(max(m, n)). Assume that m*m* and n represents the length of l1*l*1 and l2*l*2respectively, the algorithm above iterates at most max(m, n) times.
* Space complexity : O(max(m, n)). The length of the new list is at most max(m,n) + 1.

# 3. Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

**Examples:**

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a **substring**, "pwke" is a *subsequence* and not a substring.

思路1：

使用一种叫做滑动窗口sliding window的算法，逐一检查每个子串，并使用经验指导优化。另外可以使用字符串所用的全部字符集charset table代替Map，不过似乎对性能提升意义不大。

代码1：

public class Solution {

public int lengthOfLongestSubstring(String s) {

int n = s.length(), ans = 0;

Map<Character, Integer> map = new HashMap<>(); // current index of character

// try to extend the range [i, j]

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

if (map.containsKey(s.charAt(j))) {

i = Math.max(map.get(s.charAt(j)), i);

}

ans = Math.max(ans, j - i + 1);

map.put(s.charAt(j), j + 1);

}

return ans;

}

}

复杂度1：

* Time complexity : O(n). Index jwill iterate n times.
* Space complexity (HashMap) : O(min(m, n)). Same as the previous approach.
* Space complexity (Table): O(m). m is the size of the charset.

# 4. Median of Two Sorted Arrays

There are two sorted arrays **nums1** and **nums2** of size m and n respectively.

Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

You may assume **nums1** and **nums2** cannot be both empty.

**Example 1:**

nums1 = [1, 3]

nums2 = [2]

The median is 2.0

**Example 2:**

nums1 = [1, 2]

nums2 = [3, 4]

The median is (2 + 3)/2 = 2.5

思路1：已排序两向量的快速中间划分，即依据中位数定义，只要满足左右两部分数目相等并且一边严格大于另一边即可完成中位数查找。

代码1：

class Solution {

public double findMedianSortedArrays(int[] A, int[] B) {

int m = A.length;

int n = B.length;

if (m > n) { // to ensure m<=n

int[] temp = A; A = B; B = temp;

int tmp = m; m = n; n = tmp;

}

int iMin = 0, iMax = m, halfLen = (m + n + 1) / 2;

while (iMin <= iMax) {

int i = (iMin + iMax) / 2;

int j = halfLen - i;

if (i < iMax && B[j-1] > A[i]){

iMin = i + 1; // i is too small

}

else if (i > iMin && A[i-1] > B[j]) {

iMax = i - 1; // i is too big

}

else { // i is perfect

int maxLeft = 0;

if (i == 0) { maxLeft = B[j-1]; }

else if (j == 0) { maxLeft = A[i-1]; }

else { maxLeft = Math.max(A[i-1], B[j-1]); }

if ( (m + n) % 2 == 1 ) { return maxLeft; }

int minRight = 0;

if (i == m) { minRight = B[j]; }

else if (j == n) { minRight = A[i]; }

else { minRight = Math.min(B[j], A[i]); }

return (maxLeft + minRight) / 2.0;

}

}

return 0.0;

}

}

复杂度1：

* Time complexity:  *O*(log(min(*m*,*n*))).  
  At first, the searching range is [0,*m*]. And the length of this searching range will be reduced by half after each loop. So, we only need log(*m*) loops. Since we do constant operations in each loop, so the time complexity is *O*(log(*m*)). Since *m*≤*n*, so the time complexity is *O*(log(min(*m*,*n*))).
* Space complexity: *O*(1).  
  We only need constant memory to store 99 local variables, so the space complexity is *O*(1).

# 5. Longest Palindromic Substring

Given a string **s**, find the longest palindromic substring in **s**. You may assume that the maximum length of **s** is 1000.

**Example 1:**

**Input:** "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

**Example 2:**

**Input:** "cbbd"

**Output:** "bb"

思路1：Expand Around Center

直观的遍历求解，子串必须是镜像的才符合

代码1：

public String longestPalindrome(String s) {

if (s == null || s.length() < 1) return "";

int start = 0, end = 0;

for (int i = 0; i < s.length(); i++) {

int len1 = expandAroundCenter(s, i, i);

int len2 = expandAroundCenter(s, i, i + 1);

int len = Math.max(len1, len2);

if (len > end - start) {

start = i - (len - 1) / 2;

end = i + len / 2;

}

}

return s.substring(start, end + 1);

}

private int expandAroundCenter(String s, int left, int right) {

int L = left, R = right;

while (L >= 0 && R < s.length() && s.charAt(L) == s.charAt(R)) {

L--;

R++;

}

return R - L - 1;

}

复杂度1：

* Time complexity : O(n^2). Since expanding a palindrome around its center could take O(n)time, the overall complexity is O(n^2).
* Space complexity : O(1).

# 6. ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:

string convert(string s, int numRows);

**Example 1:**

**Input:** s = "PAYPALISHIRING", numRows = 3

**Output:** "PAHNAPLSIIGYIR"

**Example 2:**

**Input:** s = "PAYPALISHIRING", numRows = 4

**Output:** "PINALSIGYAHRPI"

**Explanation:**

P I N

A L S I G

Y A H R

P I

思路1：按照直观的方法，将每一行作为一个容器，然后依次按照zigzag顺序填入字符。另一种方法是通过数学规律分析出编号的规律。代码以第二种思路为主，两种方法代价一致。

代码1：

class Solution {

public String convert(String s, int numRows) {

if (numRows == 1) return s;

StringBuilder ret = new StringBuilder();

int n = s.length();

int cycleLen = 2 \* numRows - 2;

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

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

ret.append(s.charAt(j + i));

if (i != 0 && i != numRows - 1 && j + cycleLen - i < n)

ret.append(s.charAt(j + cycleLen - i));

}

}

return ret.toString();

}

}

复杂度1：

* Time Complexity: *O*(*n*), where *n*==len(*s*). Each index is visited once.
* Space Complexity: *O*(*n*). For the cpp implementation, *O*(1) if return string is not considered extra space.

# 7. Reverse Integer

Given a 32-bit signed integer, reverse digits of an integer.

**Example 1:**

**Input:** 123

**Output:** 321

**Example 2:**

**Input:** -123

**Output:** -321

**Example 3:**

**Input:** 120

**Output:** 21

**Note:**  
Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231,  231− 1]. For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

思路1：逆向处理字符串等，要想到使用栈结构的特点。不过对于数字可以使用除余计算，或称为位操作来达到这个目的。

代码1：

class Solution {

public int reverse(int x) {

int rev = 0;

while (x != 0) {

int pop = x % 10;

x /= 10;

if (rev > Integer.MAX\_VALUE/10 || (rev == Integer.MAX\_VALUE / 10 && pop > 7)) return 0;

if (rev < Integer.MIN\_VALUE/10 || (rev == Integer.MIN\_VALUE / 10 && pop < -8)) return 0;

rev = rev \* 10 + pop;

}

return rev;

}

}

复杂度1：

* Time Complexity: *O*(log(*x*)). There are roughly log10​(*x*) digits in *x*.
* Space Complexity: *O*(1).

# 8. String to Integer (atoi)

Implement atoi which converts a string to an integer.

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned.

**Note:**

* Only the space character ' ' is considered as whitespace character.
* Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231,  231− 1]. If the numerical value is out of the range of representable values, INT\_MAX (231− 1) or INT\_MIN (−231) is returned.

**Example 1:**

**Input:** "42"

**Output:** 42

**Example 2:**

**Input:** " -42"

**Output:** -42

**Explanation:** The first non-whitespace character is '-', which is the minus sign. Then take as many numerical digits as possible, which gets 42.

**Example 3:**

**Input:** "4193 with words"

**Output:** 4193

**Explanation:** Conversion stops at digit '3' as the next character is not a numerical digit.

**Example 4:**

**Input:** "words and 987"

**Output:** 0

**Explanation:** The first non-whitespace character is 'w', which is not a numerical digit or a +/- sign. Therefore no valid conversion could be performed.

**Example 5:**

**Input:** "-91283472332"

**Output:** -2147483648

**Explanation:** The number "-91283472332" is out of the range of a 32-bit signed integer. Thefore INT\_MIN (−231) is returned.

思路1：

I think we only need to handle four cases:

1. discards all leading whitespaces
2. sign of the number
3. overflow
4. invalid input

代码1：

复杂度1：

# 9. Palindrome Number

Determine whether an integer is a palindrome. An integer is a palindrome when it reads the same backward as forward.

**Example 1:**

**Input:** 121

**Output:** true

**Example 2:**

**Input:** -121

**Output:** false

**Explanation:** From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

**Example 3:**

**Input:** 10

**Output:** false

**Explanation:** Reads 01 from right to left. Therefore it is not a palindrome.

**Follow up:**

Coud you solve it without converting the integer to a string?

思路1：将原数字通过逐位倒置，直至一半位数的数字被倒置，然后比较二者即可。如果满足条件，二者要么相等或差一位相等。之所以选取一半位数，是为了避免某些数字倒置后上溢。

代码1：C#

public class Solution {

public bool IsPalindrome(int x) {

// Special cases:

// As discussed above, when x < 0, x is not a palindrome.

// Also if the last digit of the number is 0, in order to be a palindrome,

// the first digit of the number also needs to be 0.

// Only 0 satisfy this property.

if(x < 0 || (x % 10 == 0 && x != 0)) {

return false;

}

int revertedNumber = 0;

while(x > revertedNumber) {

revertedNumber = revertedNumber \* 10 + x % 10;

x /= 10;

}

// When the length is an odd number, we can get rid of the middle digit by revertedNumber/10

// For example when the input is 12321, at the end of the while loop we get x = 12, revertedNumber = 123,

// since the middle digit doesn't matter in palidrome(it will always equal to itself), we can simply get rid of it.

return x == revertedNumber || x == revertedNumber/10;

}

}

复杂度1：

* Time complexity :  *O*(log10​(*n*)). We divided the input by 10 for every iteration, so the time complexity is  *O*(log10​(*n*))
* Space complexity : *O*(1).

# 10. Regular Expression Matching

Given an input string (s) and a pattern (p), implement regular expression matching with support for '.' and '\*'.

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

**Note:**

* s could be empty and contains only lowercase letters a-z.
* p could be empty and contains only lowercase letters a-z, and characters like . or \*.

**Example 1:**

**Input:**

s = "aa"

p = "a"

**Output:** false

**Explanation:** "a" does not match the entire string "aa".

**Example 2:**

**Input:**

s = "aa"

p = "a\*"

**Output:** true

**Explanation:** '\*' means zero or more of the precedeng element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

**Input:**

s = "ab"

p = ".\*"

**Output:** true

**Explanation:** ".\*" means "zero or more (\*) of any character (.)".

**Example 4:**

**Input:**

s = "aab"

p = "c\*a\*b"

**Output:** true

**Explanation:** c can be repeated 0 times, a can be repeated 1 time. Therefore it matches "aab".

**Example 5:**

**Input:**

s = "mississippi"

p = "mis\*is\*p\*."

**Output:** false

思路1：

使用迭代也好递归也好，这是一个具备最优子结构性质的问题，特殊情况下或最坏情况下不使用动态规划法会造成重复计算子问题（比如匹配aaaa和a\*a\*等，这种极端情况比较难以预见）。使用动态规划方法较佳。注意，动态规划的方法显然是递归变形设计而来。

代码1：*Bottom-Up Variation*

class Solution {

public boolean isMatch(String text, String pattern) {

boolean[][] dp = new boolean[text.length() + 1][pattern.length() + 1];

dp[text.length()][pattern.length()] = true;

for (int i = text.length(); i >= 0; i--){

for (int j = pattern.length() - 1; j >= 0; j--){

boolean first\_match = (i < text.length() &&

(pattern.charAt(j) == text.charAt(i) ||

pattern.charAt(j) == '.'));

if (j + 1 < pattern.length() && pattern.charAt(j+1) == '\*'){

dp[i][j] = dp[i][j+2] || first\_match && dp[i+1][j];

} else {

dp[i][j] = first\_match && dp[i+1][j+1];

}

}

}

return dp[0][0];

}

}

复杂度1：

* Time Complexity: Let T, P*T*,*P* be the lengths of the text and the pattern respectively. The work for every call to dp(i, j) for i=0, ... ,T*i*=0,...,*T*; j=0, ... ,P*j*=0,...,*P* is done once, and it is *O*(1) work. Hence, the time complexity is *O*(*TP*).
* Space Complexity: The only memory we use is the *O*(*TP*) boolean entries in our cache. Hence, the space complexity is *O*(*TP*).

# 11. Container With Most Water

Given *n* non-negative integers *a1*, *a2*, ..., *an*, where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

**Note:**You may not slant the container and *n* is at least 2.



The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

**Example:**

**Input:** [1,8,6,2,5,4,8,3,7]

**Output:** 49

思路1：使用两个指针，一左一右向内推进，每次推进左右两个指针中较短的那一个。看似是贪心实则不是，只是一种对于蛮力算法的预判性改进，省略无需计算的情况。通过矩阵来分析所有组合的情况可以更加清晰这种方法的便捷性与正确性，这种方法只是特殊问题的巧妙解法。

代码1：

public class Solution {

public int maxArea(int[] height) {

int maxarea = 0, l = 0, r = height.length - 1;

while (l < r) {

maxarea = Math.max(maxarea, Math.min(height[l], height[r]) \* (r - l));

if (height[l] < height[r])

l++;

else

r--;

}

return maxarea;

}

}

复杂度1：

* Time complexity : *O*(*n*). Single pass.
* Space complexity : *O*(1). Constant space is used.

# 12. Integer to Roman

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, two is written as II in Roman numeral, just two one's added together. Twelve is written as, XII, which is simply X + II. The number twenty seven is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral. Input is guaranteed to be within the range from 1 to 3999.

**Example 1:**

**Input:** 3

**Output:** "III"

**Example 2:**

**Input:** 4

**Output:** "IV"

**Example 3:**

**Input:** 9

**Output:** "IX"

**Example 4:**

**Input:** 58

**Output:** "LVIII"

**Explanation:** C = 100, L = 50, XXX = 30 and III = 3.

**Example 5:**

**Input:** 1994

**Output:** "MCMXCIV"

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

思路1：转换对象二者均为数字，也就是说都有相同的数字性质，即按照不同位的值乘以位级相加可得最终数值。

代码1：

public **static** String intToRoman(int num) {

String M[] = {"", "M", "MM", "MMM"};

String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};

String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};

**return** M[num/1000] + C[(num%1000)/100] + X[(num%100)/10] + I[num%10];

}

复杂度1：时空复杂度均为O(1)

# 13. Roman to Integer

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, two is written as II in Roman numeral, just two one's added together. Twelve is written as, XII, which is simply X + II. The number twenty seven is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer. Input is guaranteed to be within the range from 1 to 3999.

**Example 1:**

**Input:** "III"

**Output:** 3

**Example 2:**

**Input:** "IV"

**Output:** 4

**Example 3:**

**Input:** "IX"

**Output:** 9

**Example 4:**

**Input:** "LVIII"

**Output:** 58

**Explanation:** C = 100, L = 50, XXX = 30 and III = 3.

**Example 5:**

**Input:** "MCMXCIV"

**Output:** 1994

**Explanation:** M = 1000, CM = 900, XC = 90 and IV = 4.

思路1：

代码1：

class Solution {

public int romanToInt(String s) {

Map<Character, Integer> rank = new HashMap<>();

rank.put('I', 1);

rank.put('V', 5);

rank.put('X', 10);

rank.put('L', 50);

rank.put('C', 100);

rank.put('D', 500);

rank.put('M', 1000);

int ans = 0;

if(s.isEmpty() || s == null) {

return 0;

} else {

try {

for(int i = 0; i < s.length(); i++) {

if(i<(s.length() - 1) &&

rank.get(s.charAt(i))

< rank.get(s.charAt(i + 1))) {

ans -= rank.get(s.charAt(i));

} else {

ans += rank.get(s.charAt(i));

}

}

} catch (NullPointerException e) {

System.out.print("String is not valid Roman number!");

}

return ans;

}

}

}

复杂度1：

# 14. Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

**Example 1:**

**Input:** ["flower","flow","flight"]

**Output:** "fl"

**Example 2:**

**Input:** ["dog","racecar","car"]

**Output:** ""

**Explanation:** There is no common prefix among the input strings.

**Note:**

All given inputs are in lowercase letters a-z.

思路1：有从字符串列表从头至尾每两个串联比较的，也有同时对所有字符串相同纵列逐一比较的，二者时空性能渐进相同，区别只是最坏情况不同。还有分治法，渐进空间性能不如前两者，道理上和串联比较差别不大。最后是二分查找，首先选取最短的字符串作为基准，然后二分，根据左或右任一部分是否为全部字符串的前缀决定深入哪一侧继续探查，但时间性能不如前者。

代码1：第二种方法的代码，纵向并行比较法

public String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0) return "";

for (int i = 0; i < strs[0].length() ; i++){

char c = strs[0].charAt(i);

for (int j = 1; j < strs.length; j ++) {

if (i == strs[j].length() || strs[j].charAt(i) != c)

return strs[0].substring(0, i);

}

}

return strs[0];

}

复杂度1：

* Time complexity :*O*(*S*) , where S is the sum of all characters in all strings. In the worst case there will be n*n* equal strings with length m*m* and the algorithm performs S = m\*n*S*=*m*∗*n* character comparisons. Even though the worst case is still the same as Approach 1, in the best case there are at most *n*∗*minLen* comparisons where *minLen* is the length of the shortest string in the array.
* Space complexity : *O*(1). We only used constant extra space.

# 15. 3Sum

Given an array nums of n integers, are there elements a, b, c in nums such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

**Note:**

The solution set must not contain duplicate triplets.

**Example:**

Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is:

[

[-1, 0, 1],

[-1, -1, 2]

]

思路1：

先排序，目的是为了后面方便避免重复。然后遍历数组，以当前元素为固定元素，在其后的范围内进行2Sum的算法。因为排序过，所以用头尾两指针夹逼操作即可，若结果大，高处指针下移，反之类似。

代码1：

public List<List<Integer>> threeSum(int[] num) {

Arrays.sort(num);

List<List<Integer>> res = **new** LinkedList<>();

**for** (int i = 0; i < num.length-2; i++) {

**if** (i == 0 || (i > 0 && num[i] != num[i-1])) {

int lo = i+1, hi = num.length-1, sum = 0 - num[i];

**while** (lo < hi) {

**if** (num[lo] + num[hi] == sum) {

res.add(Arrays.asList(num[i], num[lo], num[hi]));

**while** (lo < hi && num[lo] == num[lo+1]) lo++;

**while** (lo < hi && num[hi] == num[hi-1]) hi--;

lo++; hi--;

} **else** **if** (num[lo] + num[hi] < sum) lo++;

**else** hi--;

}

}

}

**return** res;

}

复杂度1：

# 16. 3Sum Closest

Given an array nums of n integers and an integer target, find three integers in nums such that the sum is closest to target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

**Example:**

Given array nums = [-1, 2, 1, -4], and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

思路1：

代码1：

复杂度1：

# n. xxx

思路1：

代码1：

复杂度1：

# n. xxx

思路1：

代码1：

复杂度1：