**30. 串联所有单词的子串**

给定一个字符串 s 和一些长度相同的单词 words。找出 s 中恰好可以由 words 中所有单词串联形成的子串的起始位置。

注意子串要与 words 中的单词完全匹配，中间不能有其他字符，但不需要考虑 words 中单词串联的顺序。

示例 1：

输入：

s = "barfoothefoobarman",

words = ["foo","bar"]

输出：[0,9]

解释：

从索引 0 和 9 开始的子串分别是 "barfoo" 和 "foobar" 。

输出的顺序不重要, [9,0] 也是有效答案。

示例 2：

输入：

s = "wordgoodgoodgoodbestword",

words = ["word","good","best","word"]

输出：[]

|  |
| --- |
| class Solution  {  public:  vector<int> findSubstring(string s, vector<string>& words)  {  if(words.empty())  {  return {};  }  unordered\_map<string,int> hash;  unordered\_map<string,int> found;  for(auto str : words)  {  hash[str]++;  }  const int n = words.size(); //当前有多少个单词  const int len = words[0].length(); //每个单词的长度  //n \* len就是我们要在s中截取的长度  vector<int> ans;  for(int i = 0;i <= (int)s.length() - n \* len;i++)  {  int num = 0;  found.clear();  for(num = 0;num < n;num++)  {  //截取一个单词的长度  string str = s.substr(i + num \* len,len);  if(hash.count(str))  {  found[str]++;  if(found[str] > hash[str])  {  break;  }  }  else  {  break;  }  }  if(num == n)  {  ans.push\_back(i);  }  }  return ans;  }  };  可以对字符进行hash，也可以对词进行hash  用hash记录目标值，从每个下标开始搜索当前字符串是否满足条件，满足则加入结果。 |

**36. 有效的数独**

判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。

数字 1-9 在每一行只能出现一次。

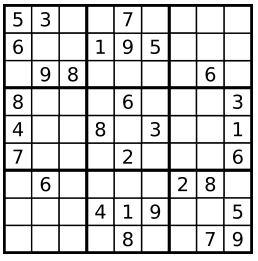
数字 1-9 在每一列只能出现一次。

数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。

上图是一个部分填充的有效的数独。

数独部分空格内已填入了数字，空白格用 '.' 表示。

示例 1:



输入:

[

["5","3",".",".","7",".",".",".","."],

["6",".",".","1","9","5",".",".","."],

[".","9","8",".",".",".",".","6","."],

["8",".",".",".","6",".",".",".","3"],

["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"],

[".","6",".",".",".",".","2","8","."],

[".",".",".","4","1","9",".",".","5"],

[".",".",".",".","8",".",".","7","9"]

]

输出: true

|  |
| --- |
| class Solution {  public boolean isValidSudoku(char[][] board) {  // 记录某行，某位数字是否已经被摆放  boolean[][] row = new boolean[9][9];  // 记录某列，某位数字是否已经被摆放  boolean[][] col = new boolean[9][9];  // 记录某 3x3 宫格内，某位数字是否已经被摆放  boolean[][] block = new boolean[9][9];  for (int i = 0; i < 9; i++) {  for (int j = 0; j < 9; j++) {  if (board[i][j] != '.') {  int num = board[i][j] - '1';  **int blockIndex = i / 3 \* 3 + j / 3;**  **if (row[i][num] || col[j][num] || block[blockIndex][num]) {**  return false;  } else {  row[i][num] = true;  col[j][num] = true;  block[blockIndex][num] = true;  }  }  }  }  return true;  }  } |

**49. 字母异位词分组**

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

输入: ["eat", "tea", "tan", "ate", "nat", "bat"]

输出:

[

["ate","eat","tea"],

["nat","tan"],

["bat"]

]

|  |
| --- |
| class Solution {  public:  vector<vector<string>> groupAnagrams(vector<string>& strs) {  map<string, vector<string>> anagrams;  for (const auto& str : strs) {  string key = str;  sort(key.begin(), key.end());  anagrams[key].push\_back(str);  }  vector<vector<string>> res;  for (const auto& item : anagrams) {  res.push\_back(item.second);  }  return res;  }  };  对排序后的字符串进行hash。 |

**347. 前 K 个高频元素**

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

示例 1:

输入: nums = [1,1,1,2,2,3], k = 2

输出: [1,2]

示例 2:

输入: nums = [1], k = 1

输出: [1]

|  |
| --- |
| class Solution {  public:  vector<int> topKFrequent(vector<int>& nums, int k) {  unordered\_map<int, int> freq;  for (auto e : nums) {  freq[e]++;  }  auto func = [](const pair<int, int>& l, const pair<int, int>& r) {  return l.second > r.second;  };  priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(func)> q(func);  for (const auto& item : freq) {  q.push(item);  if (q.size() > k) {  q.pop();  }  }  vector<int> res;  while (!q.empty()) {  res.push\_back(q.top().first);  q.pop();  }  reverse(res.begin(), res.end());  return res;  }  }; |

**467. 环绕字符串中唯一的子字符串 ！！！！**

把字符串 s 看作是“abcdefghijklmnopqrstuvwxyz”的无限环绕字符串，所以 s 看起来是这样的："...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".

现在我们有了另一个字符串 p 。你需要的是找出 s 中有多少个唯一的 p 的非空子串，尤其是当你的输入是字符串 p ，你需要输出字符串 s 中 p 的不同的非空子串的数目。

注意: p 仅由小写的英文字母组成，p 的大小可能超过 10000。

示例 1:

输入: "a"

输出: 1

解释: 字符串 S 中只有一个"a"子字符。

示例 2:

输入: "cac"

输出: 2

解释: 字符串 S 中的字符串“cac”只有两个子串“a”、“c”。.

**详细解释**

给定字符串s="abcdefghijklmnopqrstuvwxyzabcd..."(题中内置)

1. 输入一个字符串 p，输出 p 的子串的数量，满足条件

1)子串同时是 s 的子串

2)重复的子串不计数

3)子串在 p 中是连续的

4)子串在 s 中是连续的，且s字符串a-z循环自增

5)子串是非空的

注意：p 不是无限循环字符串的根

输入 p="a"，p 的子串有 "", "a"

满足条件的子串的数量是 1

输入 p="cac"，p的子串有 "", "c", "a", "ca", "ac", "cac"

满足条件的子串的数量是 2，也就是说 "ca"、"ac"、"cac"不是 s 的子串

输入 p="zab"，p的子串有 "", "z", "a", "b", "za", "ab", "zab"

满足条件的子串的数量是 6，也就是说 "z", "a", "b", "za", "ab", "zab" 都是 s 的子串

1. 为何连续子串长度等于不重复子串的数量

**个数为：以索引为 0 结尾的子数组个数 + 以索引为 1 结尾的子数组个数 + ... + 以索引为 n - 1 结尾的子数组个数，这无疑是完备的**

例如:abcdbcd, 对于以d结尾的连续字符串长度为4：以d为后缀，有abcd, bcd, cd和d,数量为4， 而bcd产生的序列都会被abcd所覆盖

总和即以a、b、c和d结尾的所有连续最长序列1 + 2 + 3 + 4 = 10

|  |
| --- |
| class Solution {  public int findSubstringInWraproundString(String p) {  int n = p.length();  if(n < 1) return 0;  int ret = 0;  int[] count = new int[26];  char[] str = p.toCharArray();  int curMaxLen = 1;  for(int i = 0; i < n; ++i) {  if(i > 0 && (str[i]-str[i-1] == 1 || str[i-1]-str[i] == 25))  curMaxLen++;  else  curMaxLen = 1;  count[str[i]-'a'] = Math.max(count[str[i]-'a'], curMaxLen);  }  for(int temp : count)  ret += temp;  return ret;  }  }  统计以每个字符作为结尾的最长连续序列(可以覆盖掉重复的短序列的情况), 他们的和即为所求 |

**560. 和为K的子数组**

给定一个整数数组和一个整数 k，你需要找到该数组中和为 k 的连续的子数组的个数。

示例 1 :

输入:nums = [1,1,1], k = 2

输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。

说明 :

数组的长度为 [1, 20,000]。

数组中元素的范围是 [-1000, 1000] ，且整数 k 的范围是 [-1e7, 1e7]。

使用哈希表记录之前出现的前缀。

|  |
| --- |
| class Solution {  public:  int subarraySum(vector<int>& nums, int k) {  map<int, int> sumCounts;//记录0-i之前出现的前缀之和出现的数量  int res = 0;  int sumPrefix = 0;  int diff = 0;  sumCounts[0] = 1;  for (size\_t i = 0; i < nums.size(); i++) {  sumPrefix += nums[i];  diff = sumPrefix - k;  if (sumCounts.find(diff) != sumCounts.end()) {  res += sumCounts[diff];  }  sumCounts[sumPrefix]++;  }  return res;  }  }; |

**523. 连续的子数组和**

给定一个包含 非负数 的数组和一个目标 整数 k，编写一个函数来判断该数组是否含有连续的子数组，其大小至少为 2，且总和为 k 的倍数，即总和为 n\*k，其中 n 也是一个整数。

示例 1：

输入：[23,2,4,6,7], k = 6

输出：True

解释：[2,4] 是一个大小为 2 的子数组，并且和为 6。

示例 2：

输入：[23,2,6,4,7], k = 6

输出：True

解释：[23,2,6,4,7]是大小为 5 的子数组，并且和为 42。

连续子数组和为k的倍数，可以对前缀和 sum 进行取余，取余之后的 sum 必然小于k，此时若在当前位置之前存在一个前缀和等于 sum的情况

余数相等相减，必然能整除

1.sum 保存到当前位置 i 的前缀和，mp:map<int, int> 保存 <前缀和对k的取余, 第一次出现 sum%k 的位置>

2.每次对sum取余，判断在位置 i之前是否存在位置使得前缀和取余后也等于 sum%k 如存在，且位置之差>1（至少有两个数）则返回true

3.如果一直遍历完，说明不符合条件，返回false

|  |
| --- |
| class Solution {  public:  bool checkSubarraySum(vector<int>& nums, int k) {  map<int, int> mp;  mp[0] = -1;  int sum = 0;  for (int i = 0; i < nums.size(); i++) {  sum += nums[i];  if (k != 0) sum %= k;  if (mp.find(sum) != mp.end()) {  if (i - mp[sum] > 1) return true;  } else {  mp[sum] = i;  }  }  return false;  }  }; |

**890. 查找和替换模式**

你有一个单词列表 words 和一个模式 pattern，你想知道 words 中的哪些单词与模式匹配。如果存在字母的排列 p ，使得将模式中的每个字母 x 替换为 p(x) 之后，我们就得到了所需的单词，那么单词与模式是匹配的。

（回想一下，字母的排列是从字母到字母的双射：每个字母映射到另一个字母，没有两个字母映射到同一个字母。）返回 words 中与给定模式匹配的单词列表。

示例：

输入：words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"

输出：["mee","aqq"]

解释：

"mee" 与模式匹配，因为存在排列 {a -> m, b -> e, ...}。

"ccc" 与模式不匹配，因为 {a -> c, b -> c, ...} 不是排列。

因为 a 和 b 映射到同一个字母。

|  |
| --- |
| class Solution {  public:  vector<string> findAndReplacePattern(vector<string>& words, string pattern) {  vector<string> res;  for(int i=0;i<words.size();i++){  if(check(words[i],pattern)) res.push\_back(words[i]);  }  return res;  }  bool check(string word,string pattern)  {  if(word.length()!=pattern.length()) return false;  for(int i=0;i<pattern.length();i++)  {  if(word.find(word[i])!=pattern.find(pattern[i])) return false;  }  return true;  }  }; |

|  |
| --- |
| class Solution {  public List<String> findAndReplacePattern(String[] words, String pattern) {  List<String> res = new ArrayList<>();  for (String word : words) {  int[] p = new int[27];  int[] p2 = new int[27];  boolean match = true;  for (int i=0;i<pattern.length();i++){  int index = word.charAt(i) - 'a' + 1;  int value = pattern.charAt(i) - 'a' + 1;  if ((p[index] != 0 && p[index] != value)||(p2[value] != 0 && p2[value] != index)){  match = false;  break;  }  p[index] = value;  p2[value] = index;  }  if (match) {  res.add(word);  }  }  return res;  }  }维护两个哈希，a中第i位数字和b中第i位数字相互映射，检测之后的相同数字是否也满足这种映射。 |

**895. 最大频率栈**

实现 FreqStack，模拟类似栈的数据结构的操作的一个类。

FreqStack 有两个函数：

push(int x)，将整数 x 推入栈中。

pop()，它移除并返回栈中出现最频繁的元素。

如果最频繁的元素不只一个，则移除并返回最接近栈顶的元素。

示例：

输入：

["FreqStack","push","push","push","push","push","push","pop","pop","pop","pop"],

[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

输出：[null,null,null,null,null,null,null,5,7,5,4]

解释：

执行六次 .push 操作后，栈自底向上为 [5,7,5,7,4,5]。然后：

pop() -> 返回 5，因为 5 是出现频率最高的。

栈变成 [5,7,5,7,4]。

pop() -> 返回 7，因为 5 和 7 都是频率最高的，但 7 最接近栈顶。

栈变成 [5,7,5,4]。

pop() -> 返回 5 。

栈变成 [5,7,4]。

pop() -> 返回 4 。

栈变成 [5,7]。

|  |
| --- |
| class FreqStack {  public:  unordered\_map<int, int> data\_to\_frequency;  unordered\_map<int, stack<int>> frequency\_of\_diffrent\_data\_set;  int max\_frequency;  FreqStack()  {  max\_frequency = 0;  }  void push(int x)  {  int tmp\_fre = 0;  data\_to\_frequency[x]++;  max\_frequency = max(max\_frequency, data\_to\_frequency[x]);  frequency\_of\_diffrent\_data\_set[data\_to\_frequency[x]].push(x);  return;  }  int pop()  {  int single\_num = frequency\_of\_diffrent\_data\_set[max\_frequency].top();  frequency\_of\_diffrent\_data\_set[max\_frequency].pop();  if (frequency\_of\_diffrent\_data\_set[max\_frequency].empty()) max\_frequency--;  data\_to\_frequency[single\_num]--;//频率降低  return single\_num;  }  };  维护两个哈希。一个哈希记录每个数字出现的频率，一个哈希记录每个频率出现的数字，因为可能有多个，可存入栈中。每次取最大频率对应的栈顶值，并移除。如果最大频率变为0，那么移除这一频率。  1个数字push 3次，在map中保存多份数据 |

**930. 和相同的二元子数组**

在由若干 0 和 1 组成的数组 A 中，有多少个和为 S 的非空子数组。

示例：

输入：A = [1,0,1,0,1], S = 2

输出：4

解释：

如下面黑体所示，有 4 个满足题目要求的子数组：

[**1,0,1**,0,1]

[**1,0,1,0**,1]

[1,**0,1,0,1**]

[1,0,**1,0,1**]

|  |
| --- |
| class Solution {  public:  int numSubarraysWithSum(vector<int>& A, int S)  {  map<int, int> hash;  int sum = 0;  hash[0] = 1;  int res = 0;  for (size\_t i = 0; i < A.size(); i++) {  sum += A[i];  if (hash.find(sum - S) != hash.end()) {  res += hash[sum - S];  }  hash[sum]++;  }  return res;  }  };  记录连续0出现的数字，计算所有(zeros[i] + 1) \* (zeros[i + S] + 1)的累加。 |

**954. 二倍数对数组**

给定一个长度为偶数的整数数组 A，只有对 A 进行重组后可以满足 “对于每个 0 <= i < len(A) / 2，都有 A[2 \* i + 1] = 2 \* A[2 \* i]” 时，返回 true；否则，返回 false。

输入：[3,1,3,6]

输出：false

输入：[2,1,2,6]

输出：false

输入：[4,-2,2,-4]

输出：true

解释：我们可以用 [-2,-4] 和 [2,4] 这两组组成 [-2,-4,2,4] 或是 [2,4,-2,-4]

class Solution:

|  |
| --- |
| def canReorderDoubled(self, A: List[int]) -> bool:  dic = collections.Counter(A)  for num in sorted(A,key = abs):#绝对值进行排序  if dic[num] == 0:continue # 后面的数4 -4，无需再判断跳过，也可以只比到n/2  if dic[num\*2] == 0:return False  dic[num] -= 1  dic[num\*2] -= 1  return True |

**974. 和可被 K 整除的子数组**

给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。

输入：A = [4,5,0,-2,-3,1], K = 5

输出：7

解释：

有 7 个子数组满足其元素之和可被 K = 5 整除：

[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

|  |
| --- |
| class Solution {  public:  int subarraysDivByK(vector<int>& A, int K)  {  map<int, int> counts;  int res = 0;  counts[0] = 1;  int sum = 0;  for (auto num : A) {  sum = ((sum + num) % K + K) % K;  if (counts.find(sum) != counts.end()) {  res += counts[sum];  }  counts[sum]++;  }  return res;  }  };  累积和的余数相同，则差值为K倍，但是余数为负数时，要加上余数在取余 |

**1072. 按列翻转得到最大值等行数**

给定由若干 0 和 1 组成的矩阵 matrix，从中选出任意数量的列并翻转其上的 每个 单元格。翻转后，单元格的值从 0 变成 1，或者从 1 变为 0 。

返回经过一些翻转后，行上所有值都相等的最大行数。

输入：[[0,1],[1,1]]

输出：1

解释：不进行翻转，有 1 行所有值都相等。

示例 2：

输入：[[0,1],[1,0]]

输出：2

解释：翻转第一列的值之后，这两行都由相等的值组成。

示例 3：

输入：[[0,0,0],[0,0,1],[1,1,0]]

输出：2

解释：翻转前两列的值之后，后两行由相等的值组成。

|  |
| --- |
| class Solution {  public:  int maxEqualRowsAfterFlips(vector<vector<int>>& matrix)  {  map<string, int> count;  for (const auto& row : matrix) {  string str;  if (row[0] == 0) {  for (auto val : row) {  str.push\_back(val);  }  } else {  for (auto val : row) {  str.push\_back(1 - val);  }  }  count[str]++;  }  int res = 0;  for (auto item : count) {  res = max(res, item.second);  }  return res;  }  };  每行为1种模式，第一个元素为1时整体反转，查找模式相同最大的数量。 |

**1124. 表现良好的最长时间段**

给你一份工作时间表 hours，上面记录着某一位员工每天的工作小时数。

我们认为当员工一天中的工作小时数大于 8 小时的时候，那么这一天就是「劳累的一天」。

所谓「表现良好的时间段」，意味在这段时间内，「劳累的天数」是严格 大于「不劳累的天数」。

请你返回「表现良好时间段」的最大长度。

输入：hours = [9,9,6,0,6,6,9]

输出：3

解释：最长的表现良好时间段是 [9,9,6]。

等价于找和大于0的最长区间。

|  |
| --- |
| class Solution {  public:  int longestWPI(vector<int>& hours) {  int n = hours.size();  unordered\_map<int, int> count;  int cur = 0;  int res = 0;  for (int i = 0; i < n; ++i) {  if (hours[i] > 8) {  cur++;  } else {  cur--;  }  if (cur > 0) res = i + 1;  else {  if (count.count(cur) < 1) count[cur] = i;  if (count.count(cur-1) > 0) res = max(res, i - count[cur-1]);  }  }  return res;  }  }; |

用一个 cur 变量记录前缀和，当大于8时，cur++, 小于8时，cur--。

由于从前向后遍历，当 cur > 0时，说明从开始到现在满足条件，时间必然是最长的，直接更新 res = i + 1。

当 cur <= 0时呢？关键来了

这里用一个 字典记录所有 cur <= 0的最小下标，所谓最小，就是后面如果再碰到了同样的 cur，不需要更新，如果没有碰到过，则把这个下标记录下来。

然后用 cur - 1 去字典里找，如果找到了下标j，那么就说明从0到 j 的前缀和是 cur-1，而从0到 i 的前缀和是 cur，那么显然从 j 到 i的和是（cur - (cur - 1)） = 1 > 0，也就是说从 j+1到 i 的表现肯定是满足的，并且由于 j 是 cur-1中最小的，所以 i-j 是最大的。

此时再跟 res 比较看是否需要更新。

上面为什么只需要查找 cur-1？因为满足条件的前缀和只能是小于等于cur-1的，也就是说其实也可以查找 cur-2,cur-3...，但是，cur-2的下标一定不可能在 cur-1的下标左边。使用反证法，前提是cur-1代表的是最小下标，那么如果 cur-2在 cur-1左边，而cur-2的左边一定还会有 cur-1出现（cur值是从0开始的），这就和最小下标的前提矛盾了。

那么问题又来了，如果 cur-1不存在，是否要查找 cur-2,cur-3...呢？

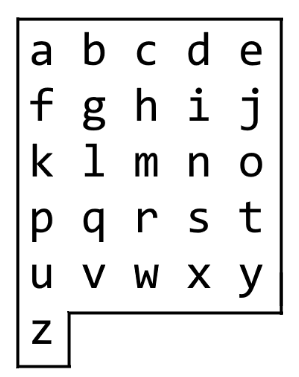
也不需要，思路跟上面是一样的，如果 cur-1不存在，cur-2,cur-3...一定也不存在。举个例子，不可能从0跳到-2，-3，而中间没有-1。

**1138. 字母板上的路径**

我们从一块字母板上的位置 (0, 0) 出发，该坐标对应的字符为 board[0][0]。

在本题里，字母板为board = ["abcde", "fghij", "klmno", "pqrst", "uvwxy", "z"]，如下所示。

我们可以按下面的指令规则行动：



如果方格存在，'U' 意味着将我们的位置上移一行；

如果方格存在，'D' 意味着将我们的位置下移一行；

如果方格存在，'L' 意味着将我们的位置左移一列；

如果方格存在，'R' 意味着将我们的位置右移一列；

'!' 会把在我们当前位置 (r, c) 的字符 board[r][c] 添加到答案中。

（注意，字母板上只存在有字母的位置。）

返回指令序列，用最小的行动次数让答案和目标 target 相同。你可以返回任何达成目标的路径。

|  |
| --- |
| class Solution {  public:  string alphabetBoardPath(string target) {  int l = target.size();  string res;  int curi = 0, curj = 0;  for(int i = 0; i < l; i++) {  int num = target[i] - 'a', r = num / 5, c = num % 5;  if('z' != target[i]) {  while(curi < r) res += "D", curi++;  while(curi > r) res += "U", curi--;  }  while(curj < c) res += "R", curj++;  while(curj > c) res += "L", curj--;  if('z' == target[i]) {  while(curi < r) res += "D", curi++;  while(curi > r) res += "U", curi--;  }  res += "!";  }  return res;  }  }; |

哈希记录每个字母的下标。按照曼哈顿距离挪过去。

**1074. 元素和为目标值的子矩阵数量**

给出矩阵 matrix 和目标值 target，返回元素总和等于目标值的非空子矩阵的数量。

子矩阵 x1, y1, x2, y2 是满足 x1 <= x <= x2 且 y1 <= y <= y2 的所有单元 matrix[x][y] 的集合。

如果 (x1, y1, x2, y2) 和 (x1', y1', x2', y2') 两个子矩阵中部分坐标不同（如：x1 != x1'），那么这两个子矩阵也不同。

输入：matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0

输出：4

解释：四个只含 0 的 1x1 子矩阵。

输入：matrix = [[1,-1],[-1,1]], target = 0

输出：5

解释：两个 1x2 子矩阵，加上两个 2x1 子矩阵，再加上一个 2x2 子矩阵。

|  |
| --- |
| class Solution {  public int numSubmatrixSumTarget(int[][] matrix, int target) {  if (matrix == null || matrix.length == 0 || matrix[0].length == 0) return 0;  // 预处理二维矩阵前缀和  int m = matrix.length;  int n = matrix[0].length;  int[][] sum = new int[m + 1][n + 1];  for (int i = 1; i <= m; i++) {  for (int j = 1; j <= n; j++) {  sum[i][j] = matrix[i - 1][j - 1] + sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1];  }  }  // 枚举所有的子矩阵  int ans = 0;  for (int x1 = 0; x1 < m; x1++) { // 枚举左上角  for (int y1 = 0; y1 < n; y1++) {  for (int x2 = x1; x2 < m; x2++) { // 枚举右下角  for (int y2 = y1; y2 < n; y2++) {  int temp = sum[x2 + 1][y2 + 1] - sum[x1][y2 + 1] - sum[x2 + 1][y1] + sum[x1][y1];  if (temp == target) ans++;  }  }  }  }  return ans;  }  }  对任何两行/列的前缀和，求子数组的前缀和 |

**1044. 最长重复子串**

给出一个字符串 S，考虑其所有重复子串（S 的连续子串，出现两次或多次，可能会有重叠）。

返回任何具有最长可能长度的重复子串。（如果 S 不含重复子串，那么答案为 ""。）

输入："banana"

输出："ana"

输入："abcd"

输出：""

|  |
| --- |
| class Solution {  public:  string longestDupSubstring(string s) {  long long mod = 1LL << 32;  function<int(int)> jud = [&](int length) {  long long h = 0;  long long aL = 1;  for (int i = 0; i < length; i++) {  // 0×26^3 + 1×26^2 + 2×26^1 + 3×26^0  h = (h \* 26 + (s[i] - 'a')) % mod;  aL = aL \* 26 % mod; // 26 ^ length  }  unordered\_set<long long> se = { h };  for (int i = 1; i < s.length() - length + 1; i++) {  h = (h \* 26 - (s[i - 1] - 'a')\* aL % mod + mod) % mod;  h = (h + (s[i + length - 1] - 'a')) % mod;  if (se.count(h)) return i;  else se.insert(h);  }  return -1;  };  int l = 1, r = s.length();  while (l < r) {  int m = l + (r - l) / 2;  if (jud(m) == -1) r = m;  else l = m + 1;  }  int i = jud(l - 1);  return i == -1 ? "" : s.substr(i, l - 1);  }  };  哈希字符串匹配。先二分，再哈希查找是否存在长度为k的重复字符串。 |