谢字希

乐于总结，乐于开源

https://github.com/XieZixiUSTC/code1024

list deque vector 序列容器 erase 指针 set map multiset multi map erase 元素关联容器 容器配接器 queue stack priority_queue

143. 重排链表（反转后面的，里面有个标准的快慢指针模板，重要，交叉合并链表，记得 mid 尾部给个 null）

```
ListNode* middleNode(ListNode* head) {//标准的快慢指针，返回 slow 重要
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast->next != nullptr && fast->next->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next; 最后 slow。Next
        }
        return slow;
    }
    这里的不用 next
 while (fast != tail) {
                slow = slow->next;
                fast = fast->next;//先走一步，看下一步
                if (fast != tail) {//不能取尾部后面是 head mid/ mid tail
                    fast = fast->next;
                }
            }
            ListNode* mid = slow;
            return merge(sortList(head, mid), sortList(mid, tail));
        }
    /大家一起走一步
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while (fast != nullptr) {
            slow = slow->next;
            if (fast->next == nullptr) {
                return nullptr;//就没有环了
            }
            fast = fast->next->next;
            if (fast == slow) {//当两者相等
                ListNode *ptr = head;//一个从起点一个从，环相遇的地方
```

```
                while (ptr != slow) {//当两者相等，就是环的入口
                    ptr = ptr->next;
                    slow = slow->next;
                }
                return ptr;
            }
        }
        return nullptr;
    }
};
```

字符串经常包含空串，所以会是 vector<>d(length+1) dp【length】

929. 独特的电子邮件地址

难度简单 147

每封电子邮件都由一个本地名称和一个域名组成，以 @ 符号分隔。

例如，在 alice@leetcode.com 中， alice 是本地名称，而 leetcode.com 是域名。

除了小写字母，这些电子邮件还可能包含 '.' 或 '+'。

如果在电子邮件地址的**本地名称**部分中的某些字符之间添加句点（'.'），则发往那里的邮件将会转发到本地名称中没有点的同一地址。例如，"alice.z@leetcode.com" 和 "alicez@leetcode.com" 会转发到同一电子邮件地址。 （请注意，此规则不适用于域名。）

如果在**本地名称**中添加加号（'+'），则会**忽略第一个加号后面的所有内**容。这允许过滤某些电子邮件，例如 m.y+name@email.com 将转发到 my@email.com。 （同样，此规则不适用于域名。）

可以同时使用这两个规则。

给定电子邮件列表 emails，我们会向列表中的每个地址发送一封电子邮件。实际收到邮件的不同地址有多少？

示例：

输入：
`["test.email+alex@leetcode.com","test.e.mail+bob.cathy@leetcode.com","testemail+david@lee.tcode.com"]`**输出：**2 **解释：**实际收到邮件的是 `"testemail@leetcode.com"` 和 `"testemail@lee.tcode.com"`。

方法：规范化表示
思路和算法

对于每个电子邮件地址，我们求出它的规范化表示（即根据 '.' 和 '+' 的规则进行处理后得到的，本地名称中仅包含小写字母的电子邮件地址）。我们对每一个地址依次进行如下的操作：

将电子邮件地址根据 '@' 分成本地名称 local 和域名 reset 两部分，其中域名部分包含 '@'，且不需要进行额外的处理；

如果本地名称中有 '+'，那么移除 '+' 以及它后面出现的所有字符；

移除本地名称中的所有 '.'；

处理完成的本地名称和域名进行连接，得到电子邮件地址的规范化表示 local + rest 。

在得到了所有电子邮件地址的规范化表示后，我们将它们放入集合（ Set ）中，就可以获知不同地址的数目。

JavaPython

```
class Solution(object):
    def numUniqueEmails(self, emails):
        seen = set()
        for email in emails:
            local, domain = email.split('@')
            if '+' in local:
                local = local[:local.index('+')]//找到 index
            seen.add(local.replace('.','') + '@' + domain)
        return len(seen)
```

作者：LeetCode
链接：https://leetcode-cn.com/problems/unique-email-addresses/solution/du-te-de-dian-zi-you-jian-di

-zhi-by-leetcode/
来源：力扣（LeetCode）

Python index() 方法检测字符串中是否包含子字符串 str ，如果指定 beg（开始） 和 end（结束） 范围，则检查是否包含在指定范围内，该方法与 python find()方法一样，只不过如果 str 不在 string 中会报一个异常。
语法

index()方法语法：

```
str.index(str, beg=0, end=len(string))
```

参数

- str -- 指定检索的字符串
- beg -- 开始索引，默认为 0。
- end -- 结束索引，默认为字符串的长度。

返回值

如果包含子字符串**返回开始的索引值，否则抛出异常。**
实例

以下实例展示了 index()方法的实例：
实例(Python 2.0+)

```
#!/usr/bin/python str1 = "this is string example....wow!!!"; str2 = "exam"; print str1.index(str2); print str1.index(str2, 10); print str1.index(str2, 40);
```

148. 排序链表（排序无序的链表，之前是合并有序的两个升序链表成为一个，就是直两两比较得了。这里用归并算法，并且用 tail 尾部，取不到来表示，数组可以取到）

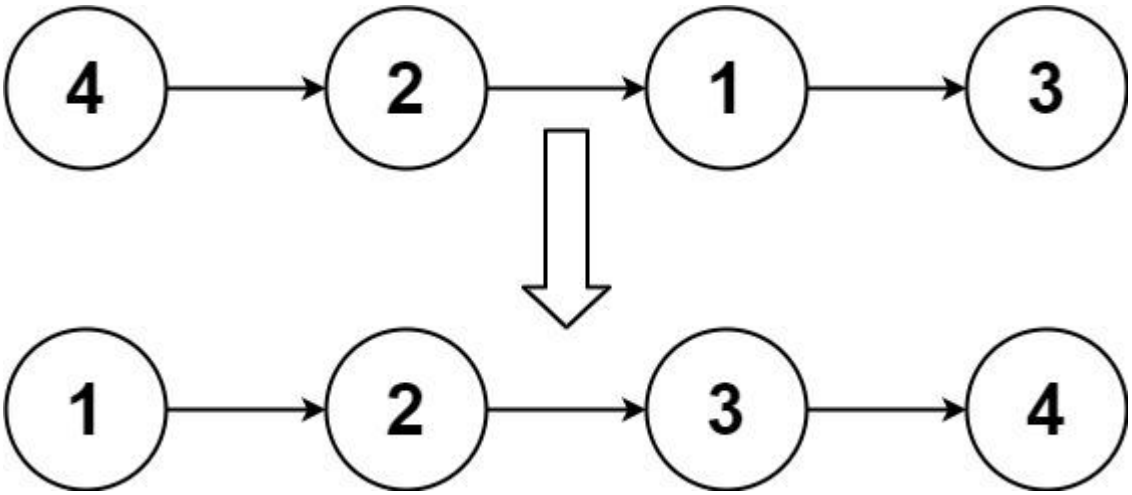
难度中等 1055

给你链表的头结点 *head* ，请将其按 **升序** 排列并返回 **排序后的链表** 。

进阶：

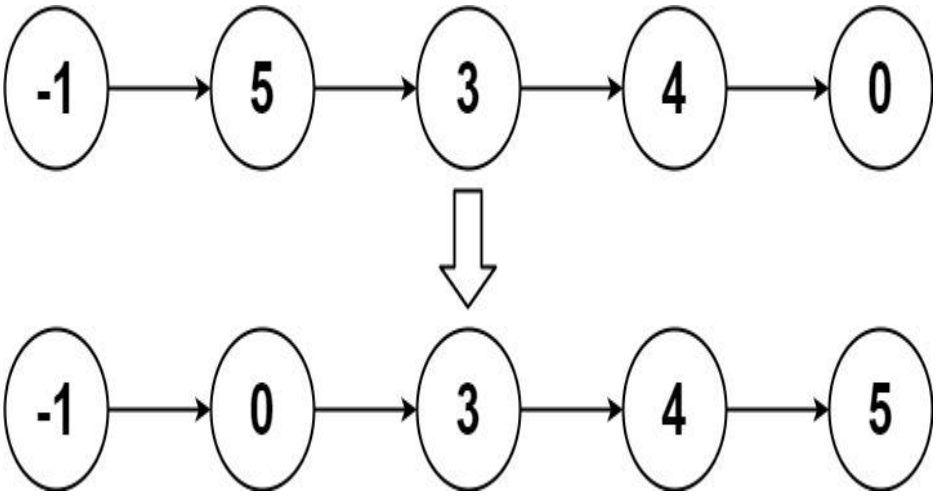
- 你可以在 *O*(*n* *log* *n*) 时间复杂度和常数级空间复杂度下，对链表进行排序吗？

示例 1：



输入：`head = [4,2,1,3]`**输出：**`[1,2,3,4]`

示例 2：



输入：head = [-1,5,3,4,0]输出：[-1,0,3,4,5]
前言
「147. 对链表进行插入排序」要求使用插入排序的方法对链表进行排序，插入排序的时间复杂度是 O(n^2)O(n
2

)，其中 nn 是链表的长度。这道题考虑时间复杂度更低的排序算法。题目的进阶问题要求达到 O(n \log n)O(nlogn) 的时间复杂度和 O(1)O(1) 的空间复杂度，时间复杂度是 O(n \log n)O(nlogn) 的排序算法包括归并排序、堆排序和快速排序（快速排序的最差时间复杂度是 O(n^2)O(n
2
)），其中最适合链表的排序算法是**归并排序**。

归并排序基于分治算法。最容易想到的实现方式是自顶向下的递归实现，考虑到递归调用的栈空间，自顶向下归并排序的空间复杂度是 O(\log n)O(logn)。如果要达到 O(1)O(1) 的空间复杂度，则需要使用自底向上的实现方式。

方法一：自顶向下归并排序
对链表自顶向下归并排序的过程如下。

找到链表的中点，以中点为分界，将链表拆分成两个子链表。寻找链表的中点可以使用快慢指针的做法，快指针每次移动 22 步，慢指针每次移动 11 步，当快指针到达链表末尾时，慢指针指向的链表节点即为链表的中点。

对两个子链表分别排序。

将两个排序后的子链表合并，得到完整的排序后的链表。可以使用「21. 合并两个有序链表」的做法，将两个有序的子链表进行合并。

上述过程可以通过递归实现。递归的终止条件是链表的节点个数小于或等于 11，即当链表为空或者链表**只包含 11 个节点时，不需要对链表进行拆分和排序。**

JavaC++JavaScriptPython3GolangC

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        return sortList(head, nullptr);//最后是空
    }

    ListNode* sortList(ListNode* head, ListNode* tail) {
        if (head == nullptr) {//没有就返回空了
            return head;
        }
        if (head->next == tail) {//只有一个就返回自身了，这里要等于尾部，因为之后换了，尾部取不到的，尾部给了第二段了，这个注意下
            head->next = nullptr;
            return head;
        }
        ListNode* slow = head, *fast = head;
```

```
        while (fast != tail) {
            slow = slow->next;
            fast = fast->next;//先走一步，看下一步
            if (fast != tail) {//不能取尾部后面是  head mid/ mid tail
                fast = fast->next;
            }
        }
        ListNode* mid = slow;
        return merge(sortList(head, mid), sortList(mid, tail));
    }

    ListNode* merge(ListNode* head1, ListNode* head2) {
        ListNode* dummyHead = new ListNode(0);
        ListNode* temp = dummyHead, *temp1 = head1, *temp2 = head2;
        while (temp1 != nullptr && temp2 != nullptr) {
            if (temp1->val <= temp2->val) {
                temp->next = temp1;
                temp1 = temp1->next;
            } else {
                temp->next = temp2;
                temp2 = temp2->next;
            }
            temp = temp->next;
        }
        if (temp1 != nullptr) {
            temp->next = temp1;
        } else if (temp2 != nullptr) {
            temp->next = temp2;
        }
        return dummyHead->next;
    }
};
```


786. 第 K 个最小的素数分数(二分查找，已知是 0-1 之间，取不到的，所以

left<right，每次遍历，找到小于 mid 的数有几个,记录最大的数,如果 cnt=k，返回，大于就缩小右边界，小于增大左边界，巧妙)

难度困难 77

给你一个按递增顺序排序的数组 arr 和一个整数 k 。数组 arr 由 1 和若干 素数 组成，且其中所有整数互不相同。

对于每对满足 0 < i < j < arr.length 的 i 和 j ，可以得到分数 arr[i] / arr[j] 。

那么第 k 个最小的分数是多少呢? 以长度为 2 的整数数组返回你的答案, 这里 answer[0] == arr[i] 且 answer[1] == arr[j] 。

示例 1：

输入：arr = [1,2,3,5], k = 3 输出：[2,5]解释：已构造好的分数,排序后如下所示:

1/5, 1/3, 2/5, 1/2, 3/5, 2/3

很明显第三个最小的分数是 2/5

示例 2：

输入：arr = [1,7], k = 1 输出：[1,7]

提示：

2 <= arr.length <= 1000

1 <= arr[i] <= 3 * 104

arr[0] == 1

arr[i] 是一个 素数 ， i > 0

arr 中的所有数字 互不相同 ，且按 严格递增 排序

1 <= k <= arr.length * (arr.length - 1) / 2

方法一：二分查找【通过】

思路

under(x) 用于求解小于 x 的分数数量，这是一个关于 x 的单调增函数，因此可以使用二分查找求解。

算法

使用二分查找找出一个 x，使得小于 x 的分数恰好有 K 个，并且记录其中最大的一个分数。

我们的二分搜索与其他的二分搜索方法类似：初始有区间 [lo, hi]，中心点 mi = (lo + hi) / 2.0。如果小于 mi 的分数数量小于 K，更新区间为 [mi, hi]，否则更新为 [lo, mi]。更多关于二分搜索的内容，请访问 LeetCode 探索这里。

under(x) 函数有两个目的：返回小于 x 的分数数量以及**小于 x 的最大分数**。在 under(x) 函数中使用滑动窗口的方法：对于每个 primes[j]，找出最大的 i 使得 **primes[i] / primes[j] < x。随着 j （和 primes[j]）的增加， i 也会随之增加。**

二分查找 + DP （复杂度压缩）

rockyL2

发布于 2020-07-15898 二分查找动态规划 C++
本题与 719.找出第 k 小的距离对 思路相同，**最大的分数是 1.0，最小的分数是 0.0。**
二分查找左值为最小分数，右值为最大分数，计算中值，从数组中找到所组成的分数比中值小的数对个数，并保留这些数对中最大的一组数对。如果比中值小的**数对个数等于 K**，那么保留的最大数对就是答案，因为他是所有数对中前 K 小个数对中最大的一个，也就是第 K 小的数对；如果比中值小的数对个数小于 K，**说明中值取小了，令左值为中值；如果比中值小的数对个数大于 K，说明中值取大了，令右值为中值。**
使用 DP 计算数组中比中值小的数对个数 cnt，首先使用一维 dp 数组，dp[i]表示以 A[i]为分子，**使得 A[i]/A[j] < mid 的个数**，这样遍历数组后 cnt = ∑dp[i]，可以将 dp 数组压缩为一个变量 cnt。dp[i]的计算方法是从 i 之后找到第一个 j 使得 A[i]/A[j] < mid，此时从 A[j] 到 A[n-1] 都满足要求，cnt+=n-j，同时 A[j]是能和 A[i]组成最大分数的数，**保留该最大数对**。还有个关键的地方是 j 不用每次都从 i+1 开始循环，因为如果上一次对于 A[i]第一个符合条件的是 A[j]，那么对于 A[i+1] > A[i]，想要让 A[i+1]/A[x] < mid，那么 x 一定大于 j，所以第一个满足 A[i+1] 的 j 一定比满足 A[i]的 j 要大，所以 j 每次从上一次结束的位置继续遍历就可以。这样时间复杂度从 O(n^2) 减到了 O(n)。

```
vector<int> kthSmallestPrimeFraction(vector<int>& A, int K) {
    double left = 0.0;
    double right = 1.0;
    int n = A.size();
    while(left<right){
        double mid = (right+left)/2.0;
        int cnt = 0;
        vector<int> maxi = {0, 1};
        int j = 0;
        for(int i=0;i<n;i++){
            while(j<n && A[i]>=mid*A[j])       j++;//变相除去 i==j
            cnt += n-j;
            if(j<n && maxi[0]*A[j] < A[i]*maxi[1])    maxi = {A[i], A[j]};//每次保留最大的数
        }
        if(cnt==K)    return maxi;
        if(cnt<K)     left = mid;
        else right = mid;
```

```
    }
    return {};
}
```


23. 合并 K 个升序链表(ListNode head, *tail = &head, *aPtr = a, *bPtr = b; 合并两个的。给一个哑结点）

难度困难 1212

给你一个链表数组，每个链表都已经按**升序排列**。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例 1：

输入：lists = [[1,4,5],[1,3,4],[2,6]]输出：[1,1,2,3,4,4,5,6]解释：链表数组如下：

```
[
  1->4->5,
  1->3->4,
  2->6
]
```

将它们合并到一个有序链表中得到。

1->1->2->3->4->4->5->6

示例 2：

输入：lists = []输出：[]

示例 3：

输入：lists = [[]]输出：[]

文字题解

前置知识：合并两个有序链表

思路

在解决「合并 K 个排序链表」这个问题之前，我们先来看一个更简单的问题：如何合并两个有序链表？假设链表 aa 和 bb 的长度都是 nn，如何在 O(n)O(n) 的时间代价以及 O(1)O(1) **的空间代价完成合并？ 这个问题在面试中常常出现，为了达到空间代价是 O(1)O(1)，我们的宗旨是「原地调整链表元素的 next 指针完成合并」**。以下是合并的步骤和注意事项，对这个问题比较熟悉的读者可以跳过这一部分。此部分建议结合代码阅读。

首先我们需要一个变量 head 来保存合并之后链表的头部，你可以把 head 设置为一个虚拟的头（也就是 head 的 val 属性不保存任何值），这是为了方便代码的书写，在整个链表合并完之后，返回它的下一位置即可。

我们需要一个指针 tail 来记录下一个插入位置的前一个位置，以及两个指针 aPtr 和 bPtr 来记录 aa 和 bb 未合并部分的第一位。注意这里的描述，tail 不是下一个插入的位置，aPtr 和 bPtr 所指向的元素处于「待合并」的状态，也就是说它们还没有合并入最终的链表。 当然你也可以给他们赋予其他的定义，但是定义不同实现就会不同。

当 aPtr 和 bPtr 都不为空的时候，取 val 熟悉较小的合并；如果 aPtr 为空，则把整个 bPtr 以及后面的元素全部合并；bPtr 为空时同理。
在合并的时候，应该先调整 tail 的 next 属性，再后移 tail 和 *Ptr（aPtr 或者 bPtr）。那么这里 tail 和 *Ptr 是否存在先后顺序呢？它们谁先动谁后动都是一样的，不会改变任何元素的 next 指针。

2.使用指针间接访问

```
int i=12;int *p=&i;//&为取地址符 cout<<*p<<endl;//这里*是解引用符
```

示例

代码

C++Java

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode *a, ListNode *b) {
        if ((!a) || (!b)) return a ? a : b;
        ListNode head, *tail = &head, *aPtr = a, *bPtr = b;
        while (aPtr && bPtr) {
            if (aPtr->val < bPtr->val) {
                tail->next = aPtr; aPtr = aPtr->next;
            } else {
                tail->next = bPtr; bPtr = bPtr->next;
            }
            tail = tail->next;
        }
        tail->next = (aPtr ? aPtr : bPtr);
        return head.next;
    }

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        ListNode *ans = nullptr;
        for (size_t i = 0; i < lists.size(); ++i) {
            ans = mergeTwoLists(ans, lists[i]);
        }
        return ans;
    }
};
```

作者：LeetCode-Solution

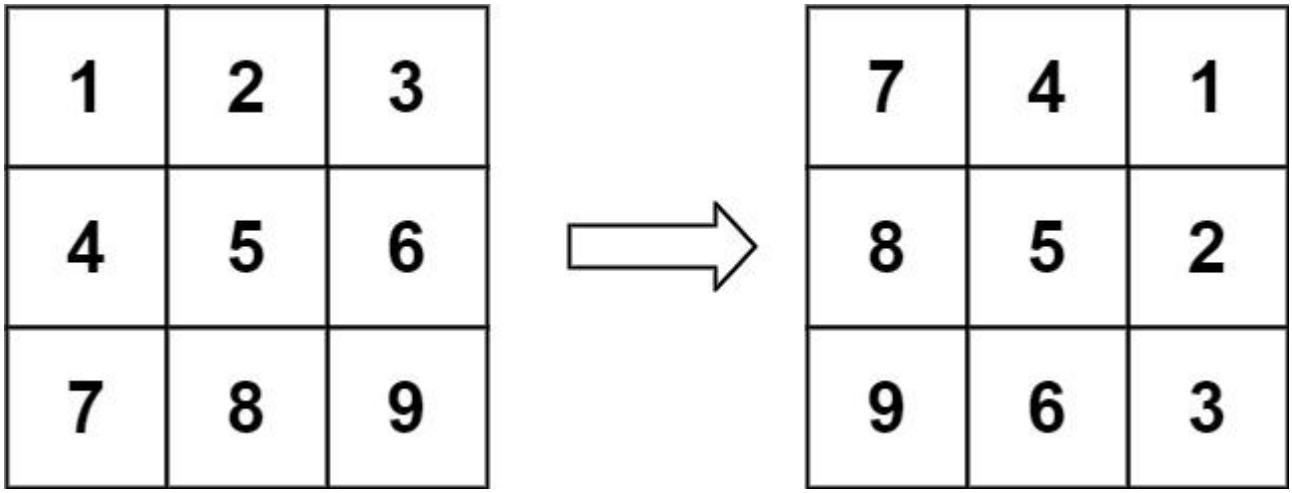
链接：https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/he-bing-kge-pai-xu-lian-biao-by-leetcode-solutio-2/
来源：力扣（LeetCode）

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/he-bing-kge-pai-xu-lian-biao-by-leetcode-solutio-2/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

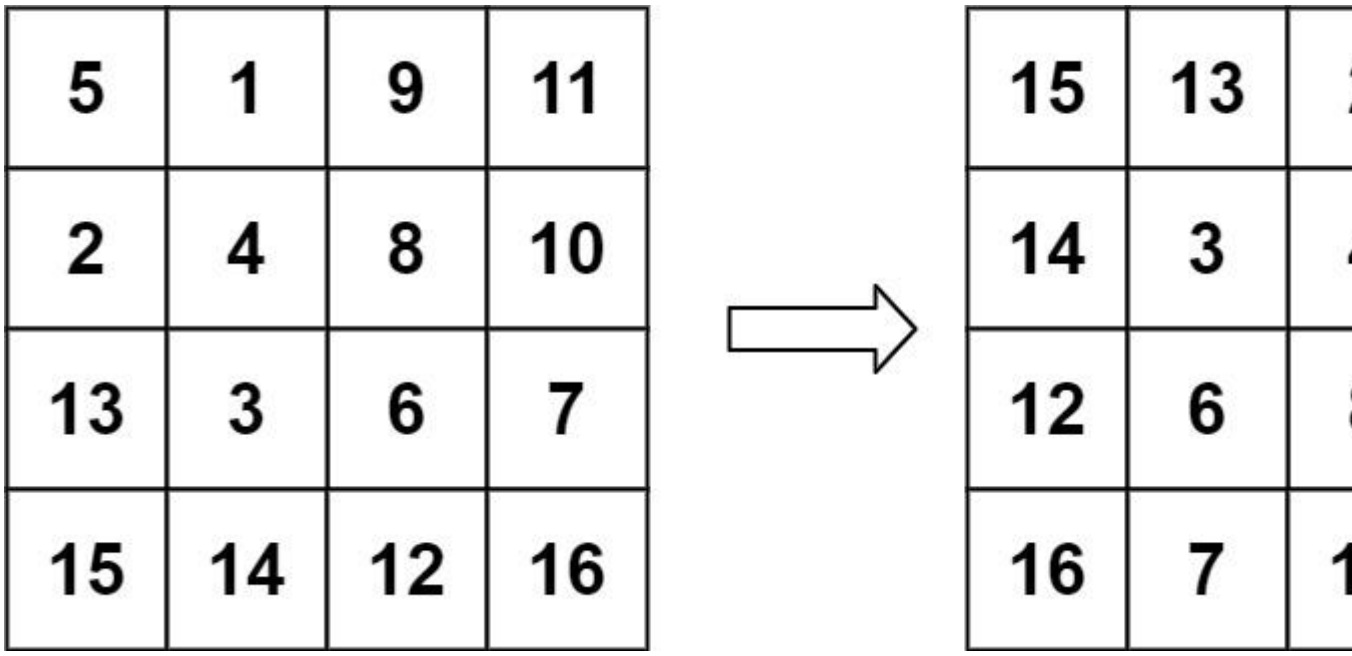
48. 旋转图像（ matrix_new[j][n - i - 1] = matrix[i][j];）

难度中等 820
给定一个 n × n 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。
你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

示例 1：



输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]输出：[[7,4,1],[8,5,2],[9,6,3]]
示例 2：



输入：matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]] 输出：[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

示例 3：

对于矩阵中的第三行和第四行同理。这样我们可以得到规律：

对于矩阵中第 ii 行的第 jj 个元素，在旋转后，它出现在倒数第 ii 列的第 jj 个位置。

我们将其翻译成代码。由于矩阵中的行列从 00 开始计数，因此对于矩阵中的元素 \textit{matrix}[\textit{row}][\textit{col}]matrix[row][col]，在旋转后，它的新位置为 \textit{matrix}_\textit{new}[\textit{col}][n - \textit{row} - 1]matrix
new

C++JavaPython3JavaScriptGolangC

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int n = matrix.size();
        // C++ 这里的 = 拷贝是值拷贝，会得到一个新的数组
        auto matrix_new = matrix;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                matrix_new[j][n - i - 1] = matrix[i][j];
            }
        }
        // 这里也是值拷贝
```

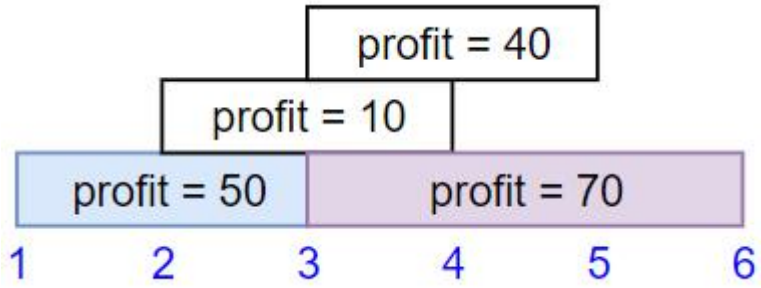
```
            matrix = matrix_new;
        }
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/rotate-image/solution/xuan-zhuan-tu-xiang-by-leetcode-solution-vu3m/
来源：力扣（LeetCode）

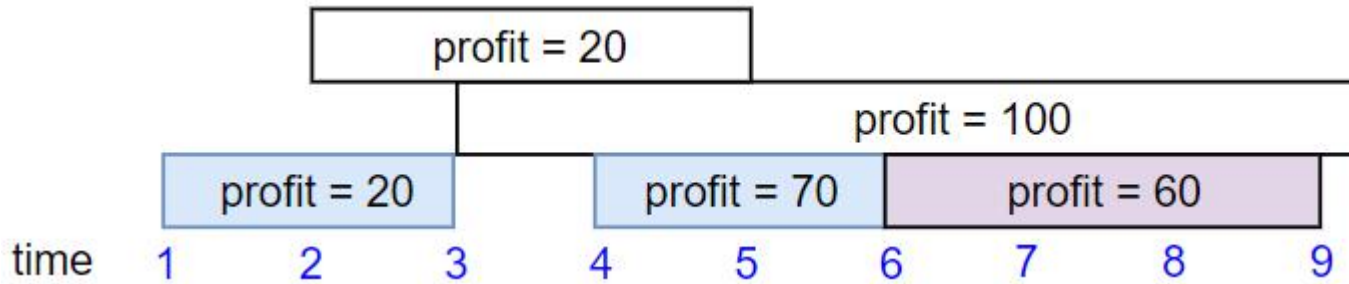
1235. 规划兼职工作（找到最近的，一个，选自己或者不选自己）

难度困难 86
你打算利用空闲时间来做兼职工作赚些零花钱。
这里有 n 份兼职工作，每份工作预计从 startTime[i] 开始到 endTime[i] 结束，报酬为 profit[i]。
给你一份兼职工作表，包含开始时间 startTime，结束时间 endTime 和预计报酬 profit 三个数组，请你计算并返回可以获得的最大报酬。
注意，时间上出现重叠的 2 份工作不能同时进行。
如果你选择的工作在时间 X 结束，那么你可以立刻进行**在时间 X 开始的下一份工作**。

示例 1：



输入：startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]输出：120 解释：我们选出第 1 份和第 4 份工作，
时间范围是 [1-3]+[3-6]，共获得报酬 120 = 50 + 70。
示例 2：



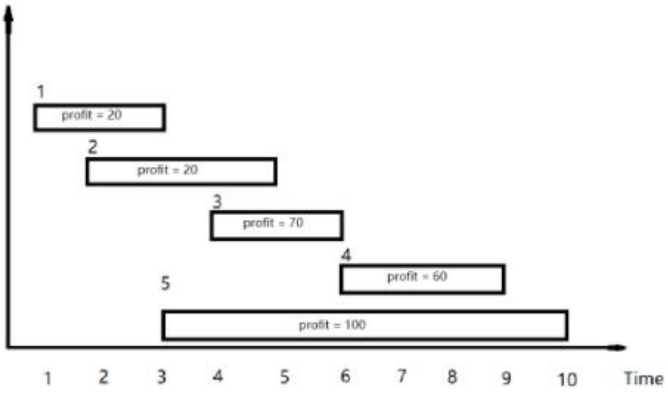
解题思路

参数定义
arr：兼职表，包含开始时间、结束时间和预计报酬
dp：动态规划数组，dp[i]表示以第 i 个元素结尾的最大报酬
思路
首先，构造兼职表 arr，并按照兼职结束时间排序
顺序遍历每一份兼职，当前为第 i 个，则去寻找前一个结束时间**小于等于当前兼职的开始时**间，即 arr[j][1]<=arr[i][0].而结束时间按照从小到大顺序排列了，且动态规划数组中的元素是满足单调性的，所以可以采用二分查找的方法【。
注意：当最后 l==0 时，需要特判 0 处的结束时间是否小于等于 i 处的开始时间。也可以在 arr **数组前补 0，减少边界处理**，见代码二。
复杂度分析
时间复杂度：O(nlognnlogn)
空间复杂度：O(nn)

C++ 动态规划详解

弟弟我到家了 L2
发布于 2019-11-263.7k 动态规划 C++
我们把示例 2 拿出来重新按工作的结束时间排序，如图所示：

我 们 把 示 例 2 拿 出 来 重 新 按 工 作 的 结 束 时 间 排 序 ， 如 图 所 示 ：



具体来说：
我们使用一个 dp 数组，dp[i]表示做包括 i 号工作之前的所有工作能取得的最大收益
再使用一个 prev 数组，prev[i]表示 i 号工作之前最近能做的工作
0 号工作之前没有能做的工作了，所以 prev[0]=0;
1 号工作之前没有能做的工作了，所以 prev[1]=0;
2 号工作之前没有能做的工作了，所以 prev[2]=0;
3 号工作之前**最近能做的工作是 1**，所以 prev[3]=1;
4 号工作之前最近能做的工作是 3，所以 prev[4]=3;
5 号工作之前最近能做的工作是 1，所以 prev[5]=1;
对于**每个兼职工作，都有做与不做两种状态：**
一.假如我们做 1 号工作，能够获得 20 元，加上在 1 号工作之前最近能做的 0 号工作(虚拟

的工作，收益也是 0)的最大收益 0 元；如果不做 1 号工作，能够获得收益是 0，于是做包括 1 号工作之前的所有工作能取的最大收益就是两中情况的最大值 20。

二.假如我们做 5 号工作，能够获得收益是 100，加上在 5 号工作之前最近能做的 prev[5]=1 号工作的最大收益 dp[1]；如果不做 5 号工作，能够获得收益就是做剩余 4 个工作最大收益 dp[4]，于是做包括 5 号工作之前的所有工作能取的最大收益就是两中情况的最大值 =max(dp[1]+profit[5],dp[4])。

所以状态转移方程就是

dp[i]=max(dp[i-1],dp[prev[i]]+profit[i])

具体到题目中来说，由于题目的输入导致一些不同，我们可以开一个二维数组 vector<vector<int>>job;把开始时间、结束时间和收益拷贝过来，再把这个二维数组排成如图所示的顺序，但是这样效率不高，因为拷贝也是要花时间的，一个比较好的办法直接对下标排序获得如图所示的顺序。

prev 数组的获得是直接向前遍历，找**第一个结束时间小于等于当前工作开始时间的工作**，感觉这里应该还是有优化的空间的。

由于加入了一个虚拟的 0 号工作，所以下标还有些变化，具体看代码。这个工作到底有没么必要加呢，如果你有解决方案请告诉我。

C++ iota()函数


小飞将 2019-07-04 17:23:51 7617 收藏 15

分类专栏： 基础

iota 函数对一个**范围数据进行赋值**：

```
template <class ForwardIterator, class T>
  void iota (ForwardIterator first, ForwardIterator last, T val)
{
  while (first!=last) {
    *first = val;
    ++first;
    ++val;
  }
}
```

用法：

```
// iota example
#include <iostream>          // std::cout
#include <numeric>           // std::iota

int main () {
  int numbers[10];

  std::iota (numbers,numbers+10,100);起点是 100，之后一直+1
```

```
    std::cout << "numbers:";
    for (int& i:numbers) std::cout << ' ' << i;
    std::cout << '\n';

    return 0;
}
```

output：

numbers: 100 101 102 103 104 105 106 107 108 109

C++代码

```
class Solution {
public:
    int jobScheduling(vector<int>& startTime, vector<int>& endTime, vector<int>& profit) {
        int n = startTime.size();
        vector<int> job(n, 0);
        iota(job.begin(), job.end(), 0);//从 0 开始  0 1 2n-1
        sort(job.begin(), job.end(), [&](int& a, int& b) {return endTime[a]< endTime[b]; });
按照结束时间排列
        vector<int> prev(n, -1);//第一个设置为-1，表示没有
        for (int i = 1; i < n; ++i) {
            for (int j = i - 1; j >= 0; --j) {//找到最近的
                if (endTime[job[j]] <= startTime[job[i]]) {//job 按照时间结束排列，倒序寻找
                    prev[i] = j;
                    break;
                }
            }
        }
        vector<int> dp(n, 0);
        dp[0] = profit[job[0]];//第一个等于自身
        for (int i = 1; i < n; ++i) {
            if (prev[i] >= 0) {
                dp[i] = max(dp[prev[i]] + profit[job[i]], dp[i - 1]);//选自己或者不选自己
            } else {
                dp[i] = max(profit[job[i]], dp[i - 1]);//=-1,就是当前没有最近的，选自己，或者前面的。
            }
        }
        return dp[n - 1];
    }
};
```

代码

```
class Solution:
    def jobScheduling(self, s: List[int], e: List[int], p: List[int]) -> int:
        n=len(s)
        arr=[]
        for i in range(n):
            arr.append([s[i],e[i],p[i]])

        arr.sort(key=lambda x:x[1])
        arr=[[0,0,0]]+arr
        dp=[0]*(n+1)
        for i in range(1,n+1):
            l,r=0,i-1
            while l<r:
                mid=l+(r-l+1)//2
                if arr[mid][1]<=arr[i][0]:
                    l=mid
                else:
                    r=mid-1

            dp[i]=max(dp[i-1],dp[r]+arr[i][2])

        return dp[-1]
```

作者：yim-6
链接：https://leetcode-cn.com/problems/maximum-profit-in-job-scheduling/solution/python3-dong-tai-gui-hua-er-fen-cha-zhao-ovu5/
来源：力扣（LeetCode）

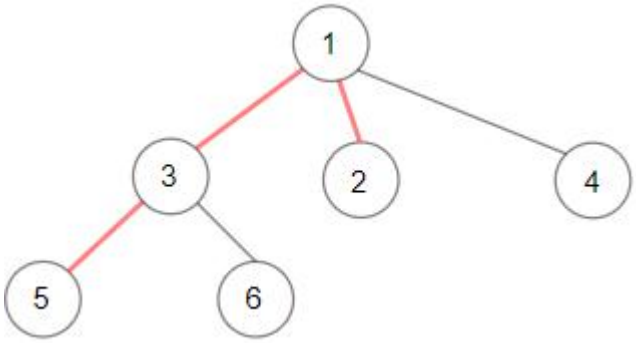
作者：yim-6
链接：https://leetcode-cn.com/problems/maximum-profit-in-job-scheduling/solution/python3-dong-tai-gui-hua-er-fen-cha-zhao-ovu5/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

1522. N 叉树的直径（先 bfs 找到最深的(不需要 visit)，之后最深的，BFS 找到后面的，记得要-2）

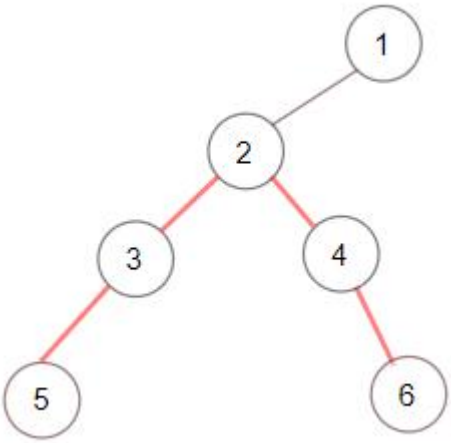
难度中等 8

给定一棵 N 叉树的根节点 root，计算这棵树的直径长度。
N 叉树的直径指的是树中任意两个节点间路径中 最长 路径的长度。这条路径可能经过根节点，也可能不经过根节点。
（N 叉树的输入序列以层序遍历的形式给出，每组子节点用 null 分隔）

示例 1：



输入：root = [1,null,3,2,4,null,5,6]输出：3 解释：直径如图中红线所示。
示例 2：



输入：root = [1,null,2,null,3,4,null,5,null,6]输出：4
示例 3：

解题思路
需要把树转换为图的表示。然后 BFS 搜索到最深的叶子，以最深的叶子 L1 作为图中的搜索起点，再进行一次 BFS 搜索，假设最后找到的节点为 L2，L1 和 L2 直接的距离就是最长路径的长度。

代码

```
/*
// Definition for a Node.
class Node {
public:
```

```
    int val;
    vector<Node*> children;//自身值，加上一个 vector

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
public:
    int diameter(Node* root) {
        if (!root || root->children.empty()) {
            return 0;
        }

        queue<Node*> q;
        q.push(root);
        Node* deepestLeaf = nullptr;
        while (!q.empty()) {
```

作者：jyj407
链接：https://leetcode-cn.com/problems/diameter-of-n-ary-tree/solution/zhong-gui-zhong-ju-1522-n-cha-shu-de-zhi-9tua/
来源：力扣（LeetCode）


```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
```

```
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/
```

代码

```
class Solution {
public:
    int diameter(Node* root) {
        if (!root || root->children.empty()) {
            return 0;
        }

        queue<Node*> q;
        q.push(root);
        Node* deepestLeaf = nullptr;
        while (!q.empty()) {
            auto cur = q.front(); q.pop();
            deepestLeaf = cur;
            for (const auto& child : cur->children) {
                mp[cur].push_back(child);//两者是相连的
                mp[child].push_back(cur);//两者相连的//不会重复访问的
                q.push(child);
            }
        }

        return bfsFindLongestPath(deepestLeaf);
    }

private:
    unordered_map<Node*, vector<Node*>> mp;//节点，到节点集合
    int bfsFindLongestPath(Node* root) {
        cout << root->val << endl;
        unordered_set<Node*> visited;
        queue<Node*> q{{root}};
        int distance = 0;
        while (!q.empty()) {
            for (int k = q.size(); k > 0; k--) {
```

```
                auto cur = q.front(); q.pop();
                if (visited.count(cur)) {
                    continue;
                }
                visited.insert(cur);
                for (const auto& neighbor : mp[cur]) {
                    q.push(neighbor);
                }
            }
            distance++;
        }

        return distance - 2;//最后 distance 还+1 了，需要-2
    }
};
```

作者：jyj407
链接：https://leetcode-cn.com/problems/diameter-of-n-ary-tree/solution/zhong-gui-zhong-ju-1522-n-cha-shu-de-zhi-9tua/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

130. 被围绕的区域（深度搜索，四条边的 0 肯定不被包围，他们的附近是 0 也不被包围，标记 A，其他的 0 变成 X 被包围了。）

难度中等 497

给你一个 *m* x *n* 的矩阵 *board* ，由若干字符 'X' 和 'O' ，找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

示例 1：

输入：board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]**输出：**[["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]**解释：**被围绕的区间不会存在于边界上，换句话说，任何边界上的 'O' 都不会被填充为 'X'。 任何不在边界上，或不与边界上的 'O' 相连的 'O' 最终都会被填充为 'X'。如果两个元素在水平或垂直方向相邻，则称它们是“相连”的。

示例 2：

输入：board = [["X"]]**输出：**[["X"]]

本题给定的矩阵中有三种元素：

字母 X；
被字母 X 包围的字母 O；
没有被字母 X 包围的字母 O。
本题要求将所有被字母 X 包围的字母 O 都变为字母 X ，但很难判断哪些 O 是被包围的，哪些 O 不是被包围的。

注意到题目解释中提到：**任何边界上的 O 都不会被填充为 X**。 我们可以想到，所有的不被包围的 O 都直接或间接与边界上的 O 相连。我们可以利用这个性质判断 O 是否在边界上，具体地说：

对于每一个边界上的 O，我们以它为起点，标记所有与它直接或间接相连的字母 O；
最后我们遍历这个矩阵，对于每一个字母：
如果该字母被标记过，则该字母为没有被字母 X 包围的字母 O，我们将其还原为字母 O；
如果该字母没有被标记过，则该字母为如果该字母被标记过，则该字母为没有被字母 X 包围的字母 O，我们将其还原为字母 O；
如果该字母没有被标记过，则该字母为被字母 X 包围的字母 O，我们将其修改为字母 X。

```
class Solution {
public:
    int n, m;

    void dfs(vector<vector<char>>& board, int x, int y) {
        if (x < 0 || x >= n || y < 0 || y >= m || board[x][y] != 'O') {//过了边界就出去，或者不等于 o
            return;
        }
//找到边界是 0 的进来，之后只要连着是 0 都行
        board[x][y] = 'A';//不被包围，不是边界，不能动，最后还是还原 O
        dfs(board, x + 1, y);//四个方向走
        dfs(board, x - 1, y);
```

```
            dfs(board, x, y + 1);
            dfs(board, x, y - 1);
        }

        void solve(vector<vector<char>>& board) {
            n = board.size();
            if (n == 0) {
                return;
            }
            m = board[0].size();
            for (int i = 0; i < n; i++) {//边界的点都是 0，不会被包围
                dfs(board, i, 0);//第一列
                dfs(board, i, m - 1);//最后一列
            }
            for (int i = 1; i < m - 1; i++) {//从 1-m-1 中间的两个被计算了。
                dfs(board, 0, i);//第一行
                dfs(board, n - 1, i);//最后一行
            }
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < m; j++) {
                    if (board[i][j] == 'A') {//不被包围
                        board[i][j] = 'O';
                    } else if (board[i][j] == 'O') {//要被包围
                        board[i][j] = 'X';
                    }
                }
            }
        }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/surrounded-regions/solution/bei-wei-rao-de-qu-yu-by-leetcode-solution/
来源：力扣（LeetCode）

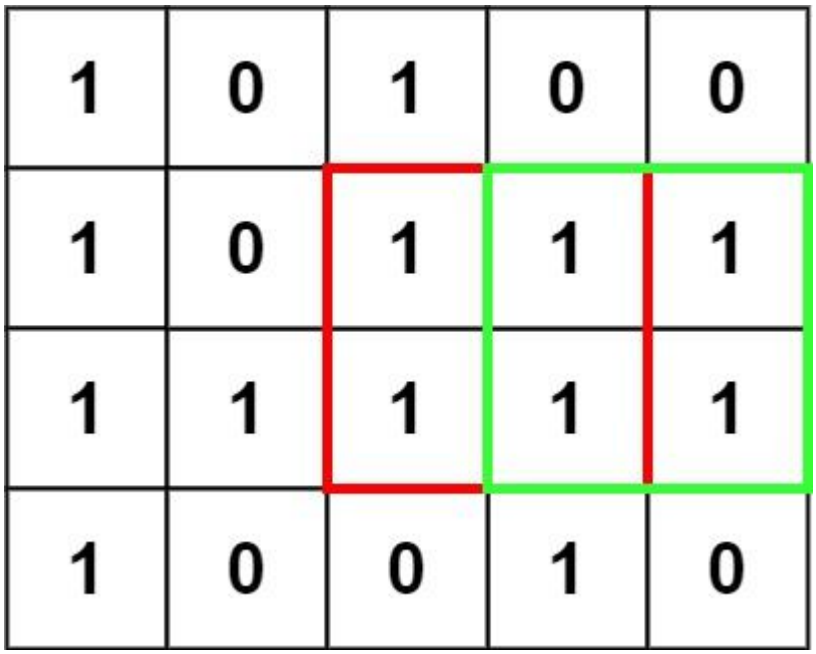
221. 最大正方形（暴力法，左上角的矩形，先中间，在左右两边，动态规划就是右下角与另外三个方向的最小值+1

（自身）边界=1）

难度中等 718

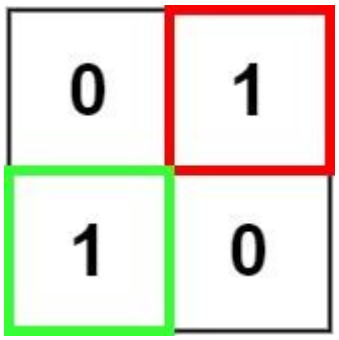
在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。

示例 1：



输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]**输出：**4

示例 2：



输入：matrix = [["0","1"],["1","0"]]**输出：**1

示例 3：

输入：matrix = [["0"]]**输出：**0

方法一：暴力法
由于正方形的面积等于边长的平方，因此要找到最大正方形的面积，首先需要找到最大正方形的边长，然后计算最大边长的平方即可。

暴力法是最简单直观的做法，具体做法如下：

遍历矩阵中的每个元素，每次遇到 11，则将该元素作为正方形的左上角；

确定正方形的左上角后，根据左上角所在的行和列计算可能的最大正方形的边长（正方形的范围不能超出矩阵的行数和列数），在该边长范围内寻找只包含 11 的最大正方形；

每次在下方新增一行以及在右方新增一列，判断新增的行和列是否满足所有元素都是 11。

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/maximal-square/solution/zui-da-zheng-fang-xing-by-leetcode-solution/
来源：力扣（LeetCode）


```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return 0;
        }
        int maxSide = 0;
        int rows = matrix.size(), columns = matrix[0].size();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                if (matrix[i][j] == '1') {
                    // 遇到一个 1 作为正方形的左上角
                    maxSide = max(maxSide, 1);
                    // 计算可能的最大正方形边长
                    int currentMaxSide = min(rows - i, columns - j);
                    for (int k = 1; k < currentMaxSide; k++) {
                        // 判断新增的一行一列是否均为 1
                        bool flag = true;
```

```
                    if (matrix[i + k][j + k] == '0') {//先判断对角线
                        break;
                    }
                    for (int m = 0; m < k; m++) {//对角线的 k 已经加了
                        if (matrix[i + k][j + m] == '0' || matrix[i + m][j + k] == '0') {
                            flag = false;
                            break;
                        }
                    }
                    if (flag) {
                        maxSide = max(maxSide, k + 1);
                    } else {
                        break;
                    }
                }
            }
        }
        int maxSquare = maxSide * maxSide;
        return maxSquare;
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/maximal-square/solution/zui-da-zheng-fang-xing-by-leetcode-solution/
来源：力扣（LeetCode）

方法一虽然直观，但是时间复杂度太高，有没有办法降低时间复杂度呢？

可以使用动态规划降低时间复杂度。我们用 $\textit{dp}(i, j)dp(i,j)$ 表示以 $(i, j)(i,j)$ 为右下角，且只包含 11 的正方形的边长最大值。如果我们能计算出所有 $\textit{dp}(i, j)dp(i,j)$ 的值，那么其中的最大值即为矩阵中只包含 11 的正方形的边长最大值，其平方即为最大正方形的面积。

那么如何计算 $\textit{dp}dp$ 中的每个元素值呢？对于每个位置 $(i, j)(i,j)$，检查在矩阵中该位置的值：

如果该位置的值是 00，则 $\textit{dp}(i, j) = 0dp(i,j)=0$，因为当前位置不可能在由 11 组成的正方形中；

如果该位置的值是 11，则 $\textit{dp}(i, j)dp(i,j)$ 的值由其上方、左方和左上方的三个相邻位置的 $\textit{dp}dp$ 值决定。具体而言，当前位置的元素值等于三个相邻位置的元素中的最

小值加 11，状态转移方程如下：

dp(i, j)=min(dp(i−1, j), dp(i−1, j−1), dp(i, j−1))+1
dp(i,j)=min(dp(i−1,j),dp(i−1,j−1),dp(i,j−1))+1

如果读者对这个状态转移方程感到不解，可以参考 1277. 统计全为 1 的正方形子矩阵的官方题解，其中给出了详细的证明。

此外，还需要考虑边界条件。如果 ii 和 jj 中至少有一个为 00，则以位置 (i, j)(i,j) 为右下角的最大正方形的边长只能是 11，因此 \textit{dp}(i, j) = 1dp(i,j)=1。

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/maximal-square/solution/zui-da-zheng-fang-xing-by-leetcode-solution/
来源：力扣（LeetCode）


```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return 0;
        }
        int maxSide = 0;
        int rows = matrix.size(), columns = matrix[0].size();
        vector<vector<int>> dp(rows, vector<int>(columns));
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                if (matrix[i][j] == '1') {
                    if (i == 0 || j == 0) {
                        dp[i][j] = 1;//状态转移方程，只要是边界等于 1
                    } else {
                        dp[i][j] = min(min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
                    }
                    maxSide = max(maxSide, dp[i][j]);
                }
            }
        }
        int maxSquare = maxSide * maxSide;
        return maxSquare;
    }
};
```

作者：LeetCode-Solution

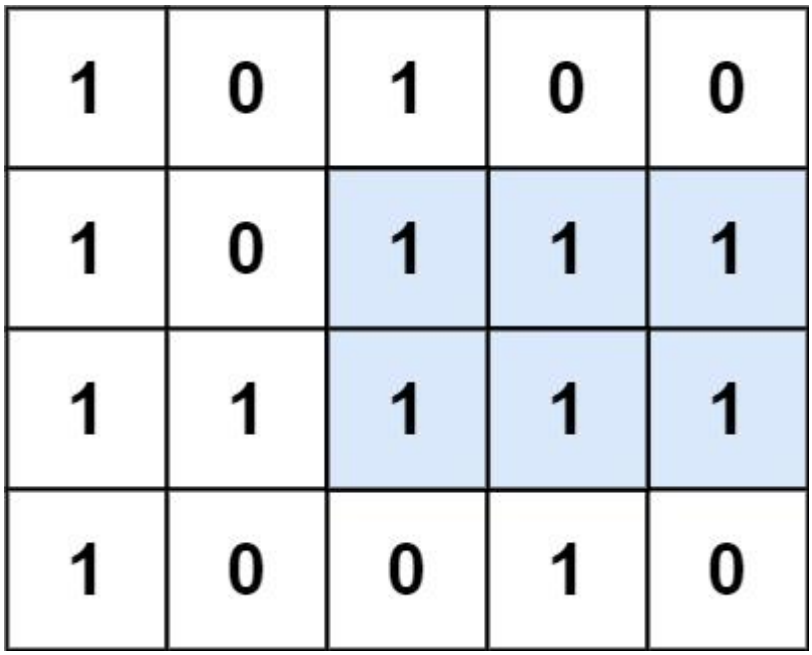
链 接 ：
https://leetcode-cn.com/problems/maximal-square/solution/zui-da-zheng-fang-xing-by-leetcode-solution/
来源：力扣（LeetCode）

85. 最大矩形（每一行，排列成连续的 1 的个数，就是最长的边长，之后比较同一列（0-i 行 j 列，从到到小）的最小的 width，之后乘以高度，就是面积了。这就是柱子的暴力图了，得到长度，再去合并，最小的长度对应的面积，每个柱子的长度，取决于）

难度困难 850

给定一个仅包含 0 和 1 、大小为 rows x cols 的二维二进制矩阵，找出只包含 1 的最大矩形，并返回其面积。

示例 1：



输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]**输出：**6 **解释：**最大矩形如上图所示。

示例 2：

输入：matrix = []**输出：**0

示例 3：

输入：matrix = [["0"]]**输出：**0

方法一：使用柱状图的优化暴力方法
思路与算法

最原始地，我们可以列举每个可能的矩形。我们枚举矩形所有可能的左上角坐标和右下角坐标，并检查该矩形是否符合要求。然而该方法的时间复杂度过高，不能通过所有的测试用例，因此我们必须寻找其他方法。

我们首先计算**出矩阵的每个元素的左边连续 11 的数量**，使用二维数组 \textit{left}left 记录，其中 \textit{left}[i][j]left[i][j] 为矩阵第 ii 行第 jj 列元素的左边连续 11 的数量。

随后，对于矩阵中任意一个点，我们枚举以该点为右下角的全 11 矩形。

具体而言，当考察以 \textit{matrix}[i][j]matrix[i][j] 为右下角的矩形时，我们枚举满足 0 \le k \le i0≤k≤i 的所有可能的 kk，此时矩阵的最大宽度就为

\textit{left}[i][j], \textit{left}[i-1][j], \ldots, \textit{left}[k][j]
left[i][j],left[i−1][j],⋯,left[k][j]

的最小值。

下图有助于理解。给定每个点的最大宽度，可计算出底端黄色方块的最大矩形面积

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/maximal-rectangle/solution/zui-da-ju-xing-by-leetcode-solution-bjlu/
来源：力扣（LeetCode）


```
class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        int m = matrix.size();
        if (m == 0) {
            return 0;
        }
        int n = matrix[0].size();
        vector<vector<int>> left(m, vector<int>(n, 0));

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == '1') {
                    left[i][j] = (j == 0 ? 0: left[i][j - 1]) + 1;//第一行就等于 0+1
                }
            }
        }

        int ret = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == '0') {
                    continue;
                }
                int width = left[i][j];//柱状图
                int area = width;//第一列
                for (int k = i - 1; k >= 0; k--) {//从 0 行到 i-1 行
                    width = min(width, left[k][j]);
                    area = max(area, (i - k + 1) * width);
                }
```

```
                    ret = max(ret, area);
                }
            }
            return ret;
        }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/maximal-rectangle/solution/zui-da-ju-xing-by-leetcode-solution-bjlu/
来源：力扣（LeetCode）

291. 单词规律 II（用回溯方法，逻辑挺多啊，就是看两个映射表是否存在。先取出字母 c，1 如果字母-单词 映射不存在，需要选取单词 1.1 如果单词-字母映射不存在，选取单词长度，对字母-单词 单词-字符，互相赋值，用回溯 1.2 如果单词-字母出现，往前走 2 如果字母-单词存在，找到单词，往前走，并且取出来的单词要相等，就往前走。反正需要字母和单词都对应，都往前走，单词不定并且没有对应要回溯，两个映射都要匹配上，才能前进。）

难度中等 48

给你一种规律 *pattern* 和一个字符串 *str*，请你判断 *str* 是否遵循其相同的规律。

这里我们指的是 **完全遵循**，例如 *pattern* 里的每个字母和字符串 *str* 中每个 **非空** 单词之间，存在着 **双射** 的对应规律。**双射** 意味着**映射双方一一对应，不会存在两个字符映射到同一个字符**串，也不会存在一个字符分别映射到两个不同的字符串。

示例 1：

输入：`pattern = "abab", s = "redblueredblue"`**输出：**`true` **解释：**一种可能的映射如下：

```
'a' -> "red"

'b' -> "blue"
```

示例 2：

输入：`pattern = "aaaa", s = "asdasdasdasd"`**输出：**`true` **解释：**一种可能的映射如下：

```
'a' -> "asd"
```

```
class Solution
{
public:
    unordered_map<char, string> char_word;                    //都用这一个，必须是标准的回溯
    unordered_map<string, char> word_char;                    //都用这一个，必须是标准的回溯

    bool wordPatternMatch(string pattern, string s)
    {
        if (pattern.size() == 0 || s.size() == 0)
            return false;
        return dfs_backtrace(pattern, 0, s, 0);//指针滑动
    }

    bool dfs_backtrace(string & pattern, int pi, string & s, int si)//从左到右指针，开始指针
    {
        if (pi >= pattern.size())//两个索引，如果 pattern 达到终点了，string 也到达终点了
            return si >= s.size();
        if (si >= s.size())//如果 string 提前到达了也不行，pattern 没有到达，+word 可能会越
```

界，需要》size

```
            return false;
        char c = pattern[pi];

        if (char_word.count(c) == 0)                    //1 字母 c 还没对应单词
        {
            for (int i = si; i < s.size(); i ++)        //回溯的框架，使劲匹配。有点暴力
            {
                int cur_word_len = i - si + 1;//字母应该匹配的长度,i 是末尾
                string word = s.substr(si, cur_word_len);
                if (word_char.count(word) == 0)         //1.1 单词也还没对应
                {
                    char_word[c] = word;                //互相赋                借用
                    word_char[word] = c;                //                      借用
                    if (dfs_backtrace(pattern, pi + 1, s, i + 1) == true)//当前的 pi 和 i 都被匹配了，下一步就是 pi+1 和 i+1 了。
                        return true;

                    char_word.erase(c);                 //回溯!!!!!!!!!!!!!!有借有还
                    word_char.erase(word);              //回溯!!!!!!!!!!!!!!有借有还
                }
                else                                    //1.2 单词对应了，要看看是不是 c，不用相互赋值了，world 在变动下匹配
                {
                    if (word_char[word] == c)//这里不要插入 c=world，判定 world 要等于 c
                        if (dfs_backtrace(pattern, pi + 1, s, i + 1) == true)
                            return true;
                }
            }
        }

        else                                            //3 字母 c 已经对应 word 了
        {
            string word = char_word[c];
            if (si + word.size() <= s.size())//当前的与对应的 world 小于 size，从 i 开始走 worldl.size 个，其实是 si+w-1<=s.size-1,索引
            {
                string tmp = s.substr(si, word.size());//判定 world 要等于 tmp
                if (word == tmp)                        //取这一段，就是对应的 word
                {
                    if (dfs_backtrace(pattern, pi + 1, s, si + word.size()) == true)
                        return true;
                }
```

```
                else                                                    //取出的这段不是 c 对应的
word
                    return false;
            }
            else                                                    //剩下的长度都不够 c 对应
的 word 取的。长度都不够，更不可能相等
                return false;
        }

        return false;               //都找遍了。都没行。
    }

};
```

作者：Hanxin_Hanxin
链接：https://leetcode-cn.com/problems/word-pattern-ii/solution/c-python-jing-dian-hui-su-fan-suo-de-xi-f4yut/
来源：力扣（LeetCode）

295. 数据流的中位数

（store.insert(lower_bound(store.begin(), store.end(), num), num);lower_bound,二分查找，第一个大于或者等于的数，在前面插入，就排序了）

难度困难 385

中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

例如，

[2,3,4] 的中位数是 3

[2,3] 的中位数是 (2 + 3) / 2 = 2.5

设计一个支持以下两种操作的数据结构：

- void addNum(int num) - 从数据流中添加一个整数到数据结构中。
- double findMedian() - 返回目前所有元素的中位数。

示例：

```
addNum(1)

addNum(2)

findMedian() -> 1.5

addNum(3)

findMedian() -> 2
```

进阶:

1. 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法？
2. 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法？

```
class MedianFinder {
    vector<int> store; // resize-able container

public:
    // Adds a number into the data structure.
    void addNum(int num)
    {
        if (store.empty())
            store.push_back(num);
        else
            store.insert(lower_bound(store.begin(), store.end(), num), num);          // binary search and insertion combined  从小到大排列
    }

    // Returns the median of current data stream
    double findMedian()
    {
        int n = store.size();
        return n & 1 ? store[n / 2] : (store[n / 2 - 1] + store[n / 2]) * 0.5;
```

```
    }
};
```

作者：LeetCode
链接：https://leetcode-cn.com/problems/find-median-from-data-stream/solution/shu-ju-liu-de-zhong-wei-shu-by-leetcode/
来源：力扣（LeetCode）

关于 lower_bound( )和 upper_bound( )的常见用法

brandong 2018-04-30 16:53:16 171637 收藏 353
分类专栏： 随笔 文章标签： lower_bound() upper_bound()

lower_bound( )和 upper_bound( )都是利用二分查找的方法在一个排好序的数组中进行查找的。

在**从小到大的排序数组中**，

lower_bound(begin,end,num)：从数组的 begin 位置到 end-1 位置二分查找第一个**大于或等于 num 的数字**，找到返回该数字的地址，**不存在则返回 end**。通过返回的地址减去起始地址 begin,得到找到数字在数组中的下标。之后 insert 在他的前面啊。

301. 删除无效的括号（任何时刻，左括号数量小于右括号数量，就是无效，找到要删除的左右括号，之后用 dfs，每个都删除，检查删除后的字符是否满足条件，去除重复的。）

难度困难 389

给你一个由若干括号和字母组成的字符串 *s*，删除最小数量的无效括号，使得输入的字符串有效。

返回所有可能的结果。答案可以按 **任意顺序** 返回。

示例 1:

输入: `"()())()"`**输出:** `["()()()", "(())()"]`

示例 2:

输入: `"(a)())()"`**输出:** `["(a)()()", "(a())()"]`

示例 3:

输入: `")("`**输出:** `[""]`

示例 1：

输入： `s = "()())()"`**输出：** `["(())()","()()()"]`

示例 2：

输入： `s = "(a)())()"`**输出：** `["(a())()","(a)()()"]`

方法一：回溯（深度优先遍历）
1. 什么是「有效的括号」
题目输入的字符串由一系列「左括号」和「右括号」组成，但是有一些额外的括号，使得括号不能正确配对。对于括号配对规则如果还不太清楚的读者，可以先完成问题 20. 有效的括号。

2. 可以一次遍历计算出多余的「左括号」和「右括号」
根据括号匹配规则和根据求解 22. 括号生成 的经验，我们知道：如果当前遍历到的左括号的数目严格小于右括号的数目则表达式无效（这一点非常重要）。
左括号数目小于右括号，当前无效

因此，我们可以遍历一次输入字符串，统计「**左括号」和「右括号」出现的次数。**

当遍历到「右括号」的时候，
如果此时「左括号」的数量不为 00，因为 「右括号」可以与之前遍历到的「左括号」匹配，此时「左括号」出现的次数 -1−1；

如果此时「左括号」的数量为 00，「右括号」数量加 11；
当遍历到「左括号」的时候，「左括号」数量加 11。
通过这样的计数规则，最后「左括号」和「右括号」的数量就是各自最少应该删除的数量。

```
void dfs(string& s, string& cur, int idx, int lcnt, int rcnt, int lnum, int rnum, set<string>& ans){
        if(idx == s.size()){
            if(lcnt == 0 && rcnt == 0) ans.insert(cur);
            return;
        }
        if(lnum < rnum) return; //不合法，剪枝
        if(s[idx] == '(' && lcnt > 0){    //删除左括号
            dfs(s, cur, idx+1, lcnt-1, rcnt, lnum, rnum, ans);
        }else if(s[idx] == ')' && rcnt > 0){    //删除右括号
            dfs(s, cur, idx+1, lcnt, rcnt-1, lnum, rnum, ans);
        }
        //保留该字符
        if(s[idx] == '(') lnum++; //更新计数
        else if(s[idx] == ')') rnum++;
        cur.push_back(s[idx]);
        dfs(s, cur, idx+1, lcnt, rcnt, lnum, rnum, ans);
        cur.pop_back();
    }

    vector<string> removeInvalidParentheses(string s) {
        int lcnt = 0, rcnt = 0;
        for(int i = 0; i < s.size(); i++){
            if(s[i] == '(') lcnt++;
            else if(s[i] == ')'){
```

```
                if(lcnt == 0) rcnt++;
                else lcnt--;
            }
        }
        if(lcnt == 0 && rcnt == 0) return {s};
        vector<string> ans;
        set<string> set;
        string str;
        dfs(s, str, 0, lcnt, rcnt, 0, 0, set);
        for(auto itr = set.begin(); itr != set.end(); itr++)
            ans.push_back(*itr);
        return ans;
    }
```

dfs（找到要删除的左右括号的数量，每次遍历删除的符号，i<size substr（0，i）其实是索引 0-i-1de 得到删除完毕的，检查是否合法）

此题与之前的生成括号方式互为相反的过程，生成时我们需要记录已加入的左边和右边括号个数，删除时我们也需要。
在此题中，解题步骤如下：

我们需要先找出**不合法的左括号个数和右括号个数**
利用 dfs 不断**删**除"（"或者"）"，**直到不合法个数为 0**
检验删除后的括号串是否合法。

```
class Solution {
public:
    vector<string> ans;
    vector<string> removeInvalidParentheses(string s) {
        // 寻找不合法半边括号的个数，完全是左右两边数量
        int left=0;
        int right=0;
```

```
        for(char i:s){
            if(i=='('){
                left++;
            }
            if(i==')'){
                if(left>0)left--;//右边与左边抵消，否则右边自己++
                else right++;
            }
        }
//如果左右都等于 0，就直接返回 return{s}得了。
        // 深度优先遍历，寻找题解
        // left 与 right 为需要删除的数量
        dfs(s, 0, left, right);

        return ans;
    }
    void dfs(string s, int st, int l, int r){
        if(l==0&&r==0){
            if(check(s)){//如果有效
                ans.push_back(s);//当前是有效，就是无效的括号都删除完了
            }
            return;
        }
        for(int i=st;i<s.size();i++){//从上一个索引开始,每个 i 遍历，每个括号都能操作，每次去除一个
             // 去重
            if(i-1>=st&&s[i]==s[i-1])continue;//跳过重复的，i 从 st+1 开始，去重
            if(l>0&&s[i]=='('){//保证每次的都会被选到，每个都遍历
                dfs(s.substr(0, i)+s.substr(i+1, s.size()-i-1), i, l-1, r);//不合法的左括号剪掉 1
            }
//可换成 dfs(s.substr(0,i)+s.substr(i+1,s.size(),l,i-1,r-1)) 除掉这个 l,后面还是从 i 开始的到末尾
            if(r>0&&s[i]==')'){
                dfs(s.substr(0, i)+s.substr(i+1, s.size()-i-1), i, l, r-1);
            }
        }
    }
    // 检验合法性（只是单个括号，做的题目是多个括号，用栈）
    刚好左右两边，相减等于 0 了。
    bool check(string s){
        int cnt=0;
        for(char i:s){
            if(i=='('){
                cnt++;// 左边先加
            }
```

```
                if(i==')'){
                    cnt--;
                    if(cnt<0)return false;//如果右边大于左边，就是 false 了。
                }
            }
            return cnt==0;
        }
};
```

作者：shaft
链接：https://leetcode-cn.com/problems/remove-invalid-parentheses/solution/shen-du-you-xian-sou-suo-jie-ti-by-shaft/
来源：力扣（LeetCode）

639. 解码方法 II（单独或者分开，逻辑很多，开始分为星号与非星号 a bc 三个。分为 0 星号 1-9 因为字母只能是 1-26 的数字，解码总数=当前字符独立解码+当前字符与前一个字符一起解码的，这里好像不考虑第一个为 0，或者，0 前面数字是 1、2 的问题了，认定是有效的。先计算分开的，在计算合并的（分开的肯定可以））

解码都是数字到字母了

难度困难 77

一条包含字母 A-Z 的消息通过以下的方式进行了编码：

```
'A' -> 1

'B' -> 2
```

...

'Z' -> 26

除了上述的条件以外，现在加密字符串可以包含字符 '*'了，**字符'*'可以被当做 1 到 9 当中的任意一个数字**。

给定一条包含数字和字符'*'的加密信息，请确定解码方法的总数。

同时，由于结果值可能会相当的大，所以你应当对 $10^9 + 7$ 取模。（翻译者标注：此处取模主要是为了防止溢出）

示例 1：

输入： "*"

输出： 9

解释： 加密的信息可以被解密为: "A", "B", "C", "D", "E", "F", "G", "H", "I".

示例 2：

输入： "1*"

输出： 9 + 9 = 18（翻译者标注：这里 1*可以分解为 1,* 或者当做 1*来处理，所以结果是 9+9=18）

说明：

DP[i]数组表示字符 I,有几种解码方式

1. 输入的字符串长度范围是 [1, 10^5]。

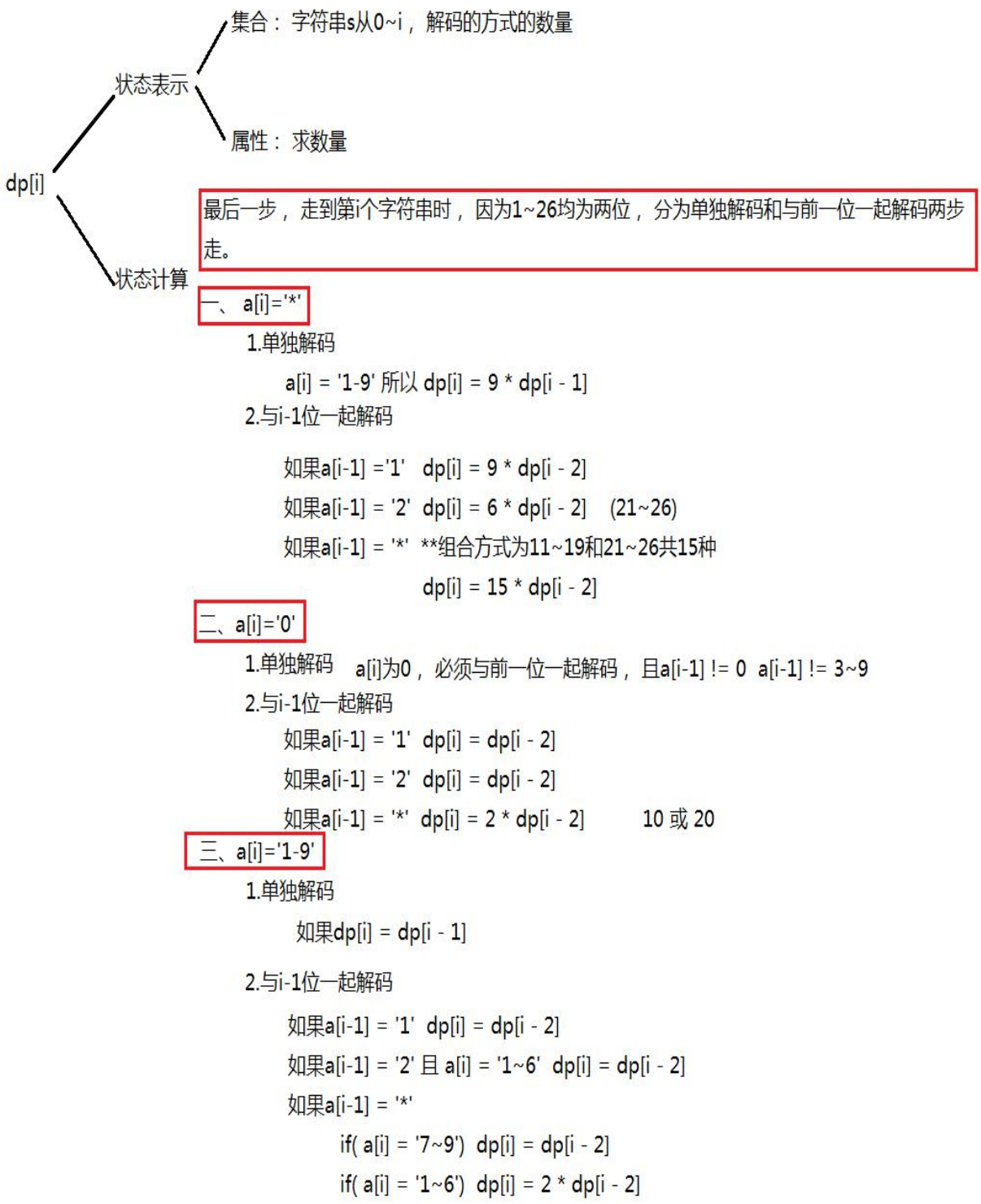
三个变量滚动更新就是了，分两种情况，当前字符是* 和不是 *,从第一个字符到现在具有的解码数量=当前**字符独立解码+当前字符与前一个字符一起解码的，**
然后再将当前字符与前一个字符一起解码的分为**前一个是 1 或者是 2**。 这样分析十分清晰。

1 当前不等于星号
当前不等于 0，因为 0 只能和 1/2 在一起 ，就和之前一样，**分开** c=b 组成 1-9
前一个等于 1 或者星号，合并起来 c+a //只能是 组成 11-19 对下一位没有限制
当前大于 0 小于 6 并且前一个等于 2 或者星号，继续 c+=a; 组成 20-26

2 当前等于星号
先与之前的 b 相乘 b*9 **分开的**
之后如果，前面等于 1 或者星号 11-19 c+a*9
如果前面是 2 21-26 c+a*6
先计算分开的，在计算合并的

```
class Solution {
public:
```

```
#define M 1000000007;
int numDecodings(string s) {
        int len=s.size();
        if(s[0]=='0') return 0;
        //vector<vector<int>> dp(len,vector<int>(11,0));
        long long a=1,b,c=0;//分别是  i-2 i-1 i 的变化
        if(s[0]!='*') b=1;//不可能会是 0 的
        else b=9;
           if(len==1) return b;//返回第二个

        for(int i=1;i<len;++i){
            if(s[i]!='*'){
                c=0;
                if(s[i]!='0') c=b;//当前不等于 0，可以分开算
                if(s[i-1]=='1' || s[i-1]=='*') c+=a;//可以随便合并 10-19 星号只能取 1
                if(s[i]>='0' && s[i]<='6' && (s[i-1]=='2' || s[i-1]=='*')) c+=a;20-26，星号只取 2
            }
            else{
                c=b*9;//不合并，星号可以去 9 个数
                if(s[i-1]=='1' || s[i-1]=='*') c+=a*9;//合并 1
                if(s[i-1]=='2' || s[i-1]=='*') c+=a*6;//合并 2
            }
            c%=M;
            a=b;//往前更新
            b=c;
        }

        return c;
    }
};
```



地方哈就看了返回

386. 字典序排数(dfs 10*i+num这样一直搜

索，直到小于 n，外面从 1 开始，里面从 0 开始，一种 dfs 搜索策略）

难度中等 162

给定一个整数 n, 返回从 1 到 n 的字典顺序。

例如，

给定 n =1 3，返回 [1,10,11,12,13,2,3,4,5,6,7,8,9] 。

请尽可能的优化算法的时间复杂度和空间复杂度。 输入的数据 n 小于等于 5,000,000。

链接：https://leetcode-cn.com/problems/lexicographical-numbers/solution/yong-zhan-mo-ni-di-gui-yong-shi-8ms-by-z-xrlq/

来源：力扣（LeetCode）

解释：

字典序是什么

数字后面每个都接着 0-9 一共是十个数，10*num 是起点，每次遍历玩这一行，才回溯上去，用 dfs 1 10 100 101..109 11 110 111

设想一本英语字典里的单词，何者在前何者在后？

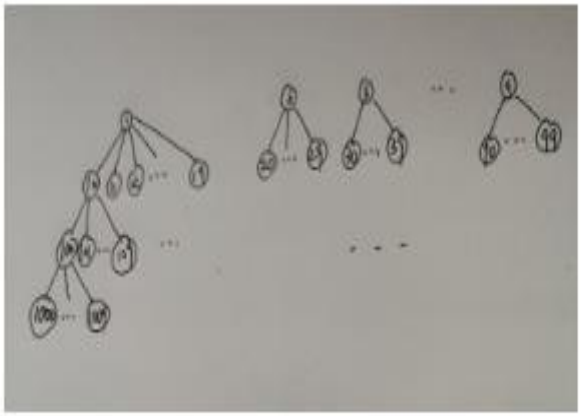
显然的做法是先按照第一个字母、以 a、b、c……z 的顺序排列；如果第一个字母一样，那么比较第二个、第三个乃至后面的字母。如果比到最后两个单词不一样长（比如，sigh 和 sight），**那么把短者排在前**。

通过这种方法，我们可以给本来不相关的单词强行规定出一个顺序。“单词”可以看作是“字母”的字符串，而把这一点推而广之就可以认为是给对应位置元素所属集合分别相同的各个有序多元组规定顺序：下面用形式化的语言说明。

字典序的排列是 `[1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9]`，所以第二小的数字是 `10`。

思路：

建立一个十叉树，如下图所示：



Dfs（对每一个十叉树进行遍历，当大于某个数就返回来，用 dfs，）

```
class Solution {
    vector<int> ans;
public:
    void dfs(int num, int& n) {
        if (num > n) return;//
        ans.push_back(num);//丢进来
        for (int i = 0; i <= 9; ++i) dfs(num * 10 + i, n);//直接往下乘法
    }

    vector<int> lexicalOrder(int n) {
        for (int i = 1; i <= 9; ++i) dfs(i, n);
        return ans;
    }
};
```

作者：heygary
链接：https://leetcode-cn.com/problems/lexicographical-numbers/solution/c-zhong-gui-zhong-ju-de-12msjie-fa-dfs-shi-jian-2/
来源：力扣（LeetCode）

426. 将二叉搜索树转化为排序的双向链表（头部和尾部，用中序遍历，得到的是从小到大的排列，每次更新 last，last 与 node 的关系，当 last 为空，记录起始的 firstnode）

难度中等 92

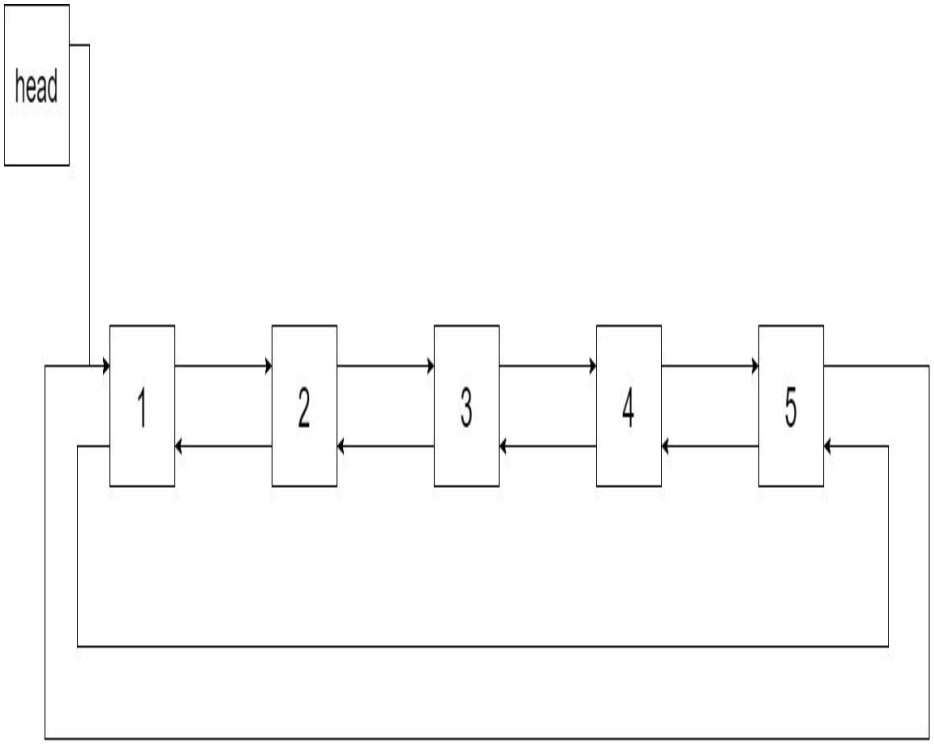
将一个 **二叉搜索树** 就地转化为一个 **已排序的双向循环链表** 。

对于双向循环列表，你可以将左右孩子指针作为双向循环链表的前驱和后继指针，第一个节点的前驱是最后一个节点，最后一个节点的后继是第一个节点。

特别地，我们希望可以 **就地** 完成转换操作。当转化完成以后，树中节点的左指针需要指向前驱，树中节点的右指针需要指向后继。还需要返回链表中最小元素的指针。

示例 1：

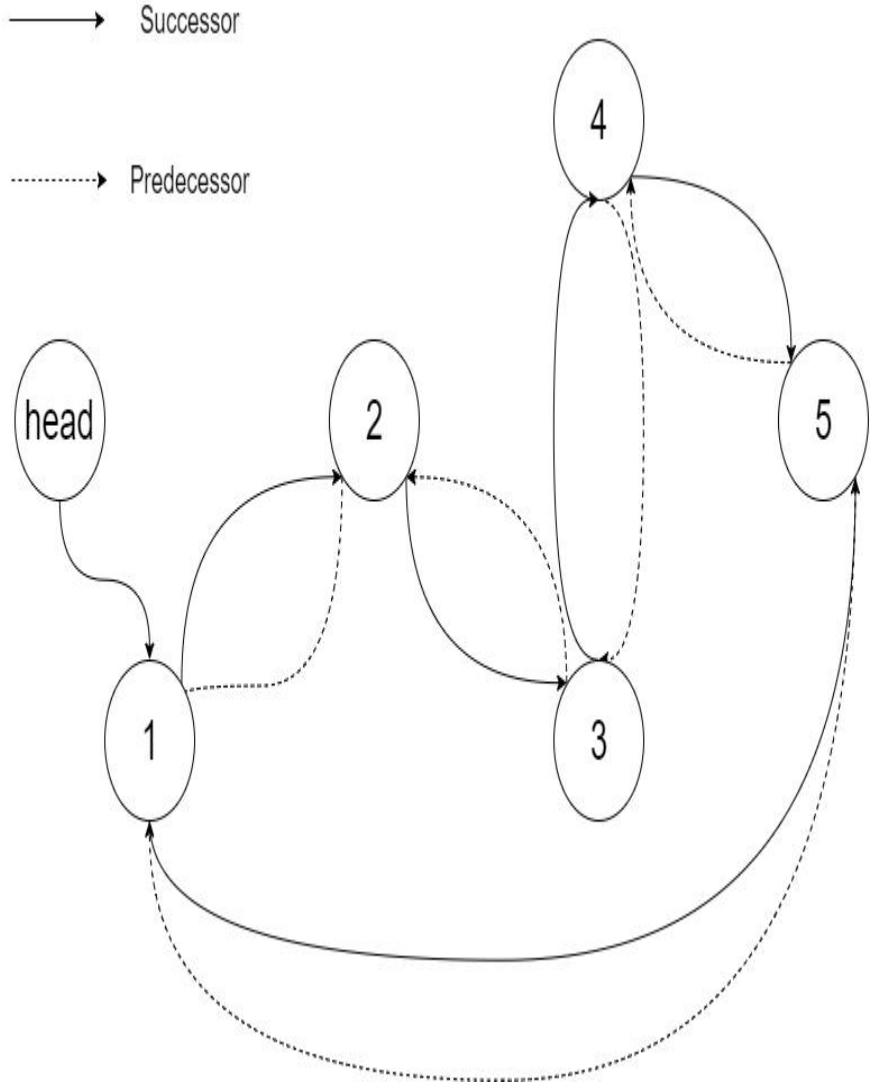
输入：`root = [4,2,5,1,3]`



输出：

```
[1,2,3,4,5]
```

解释：下图显示了转化后的二叉搜索树，实线表示后继关系，虚线表示前驱关系。



示例 2：

如何遍历树
总的来说，有两种遍历树的策略：

深度优先搜索 (DFS)

在深度优先搜索中，我们以 深度 优先，从根开始先抵达某个叶子，再回退以前往下一个分支。

深度优先搜索又可以根据根节点、左子结点和右子结点的顺序关系分为前序遍历，中序遍历和后序遍历。

广度优先搜索 (BFS)

逐层扫描整棵树，按照高度顺序自顶向下。上层的结点比下层更先访问。

下图表示了不同策略下的访问顺序，请按照 1-2-3-4-5 的顺序来比较不同的策略。

由于原地的要求，本问题需要使用深度优先搜索的中序遍历，利用备忘录法实现。

方法一：递归
算法

标准的中序遍历采用 左 -> 根 -> 右 的顺序，其中 左 和 右 的部分调用递归。

本题的处理在于将前一个结点与当前结点链接，因此，必须跟踪最后一个结点，该结点在新的双向链表中是当前最大的。

另外一个细节：我们也需要保留第一个，也就是最小的结点，以完成闭环。

下面是具体算法：

将 first 和 last 结点 初始化为 null。

调用标准中序遍历 helper(root) :

若结点不为 null :

调用左子树递归 helper(node.left)。

若 last 结点不为空，将 last 与当前的 node 链接。

否则初始化 first 结点。// 最后 left，初始化 first 最小

将当前结点标记为最后 : last = node.//当前的

调用右子树递归 helper(node.right)。

将最前与最后的结点链接完成闭环，返回 first 闭环

作者：LeetCode
链接：https://leetcode-cn.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/solution/jiang-er-cha-sou-suo-shu-zhuan-hua-wei-pai-xu-de-s/
来源：力扣（LeetCode）


```
class Solution {
  public:
  // the smallest (first) and the largest (last) nodes
  Node* first = NULL;//头部和尾部
  Node* last = NULL;

  void helper(Node* node) {
    if (node) {//当前不为空
      // left
      helper(node->left);//左，中间做点什么，右
      // node
      if (last) {

        last->right = node;//最后的与当前节点建立关系
        node->left = last;
      }
      else {

        first = node;//一开始，都不存在，记录这个起始点，
      }
      last = node;//每次更新这个最后的节点，当前节点变成最后的。
      // right
      helper(node->right);
    }
  }

  Node* treeToDoublyList(Node* root) {
    if (!root) return NULL;

    helper(root);
    // close DLL
    last->right = first;
    first->left = last;
    return first;
  }
};
```

作者：LeetCode

链 接 ：
https://leetcode-cn.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/solution/jiang-er-cha-sou-suo-shu-zhuan-hua-wei-pai-xu-de-s/
来源：力扣（LeetCode）

在 stl 中的 find 函数和 distance 函数

chen_zan_yu_ 2019-05-19 20:19:45 733 收藏 5
分类专栏： STL

find 函数和 distance 函数都是算法库里的函数
包含在头文件 algorithm 中，算是 STL 的内容
只介绍最简单的用法
find 函数有三个参数，分别代表
(起点，终点后一位，要找的数)
返回一个**地址** **迭代器**
可以是容器，或者数组
如果没有找到，则返回终点后一位的地址
找到了，返回区间[first,end)中第一个值等于 value 的元素的地址
进阶版本是 find_if()函数

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
	vector <int> v;
	for(int i = 0; i < 10; ++i) {
		v.push_back(i);
	}
	vector <int> ::iterator it = find(v.begin(), v.end(), 9);
	if(it != v.end()) cout << "在当前 vector 中\n";
	else cout << "不在当前 vector 中\n";
	cout << endl;

	int a[10];
```

```
	for(int i = 0; i < 10; ++i)
		a[i] = i;
	int *p =    find(a, a + 10, 10);
	if(p == a+10) cout << "不在这个数组中\n";
	else cout << "在这个数组中\n";
}
```

find_if()函数

```
#include <bits/stdc++.h>
using namespace std;

bool cmp(int n)
{
	return (n > 3 && n < 19);
}

int main()
{
	int a[10];
	for(int i = 0; i < 10; ++i) {
		a[i] = i;
	}
	cout << *find_if(a, a+10, cmp);
}
```

distance 是返回容器中两个地址之间的距离

参数为(地址，地址)
返回值为整型

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
	int a[10];
	int dis = distance(a, a+2);
	cout << "距离为: " << dis << endl;

	for(int i = 0; i < 10; ++i) {
		a[i] = i;
	}
	cout << "该数组中, 2 和 7 的距离是: \n";
	cout << distance(find(a, a+10, 2), find(a, a+10, 7));迭代器之间的距离
```

————————————————————

共同点：

查找成功时返回所在位置，失败返回 string::npos 的值，string::npos 一般是 MAX_INT（即 2^32 - 1）

差异：

find(): 查找字符串中第一次出现字符 c、**字符串 s 的位置**；

find_first_of(): 查找字符串中字符 c、**字符数组 s 中任意一个字符**第一次出现的位置。

Bug 分析

程序的目的是，在源字符串 s 中查找目的字符串，若找到，则显示"Found"，并返回目标子串在源字符串中的位置；反之，若未找到，则返回"Not found"。string.find 在未找到时会返回 string::npos。

在 C++中常量 npos 是这样定义的：

static const size_t npos = -1; string:npos

即常量 npos 定义的值为-1. 但又因为 npos 的类型 size_t 是无符号整数类型，所以 npos 实际上是一个正数，并且是 size_t 类型的最大值。

————————————————————

例如：

```
string haystack = "helloworld";
string needle = "world";
cout << haystack.find_first_of(needle) << endl; //2, index of first 'l'
cout << haystack.find(needle) << endl; //5, index of first "world"

needle = "";
cout << haystack.find_first_of(needle) << endl;//string::npos, 因为字符数组 s 为空，haystack 中找不到空字符（区别于'\0'）
cout << haystack.find(needle) << endl;//0, 空串
```

————————————————————

请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/Richard_123/article/details/51140229792. 匹配子序列的单词数（直接用 find,配对，查找子序列，以前用双指针，直接用 find.first 函数，没有，会返回-1，从 pos+1 位置开始查找）

难度中等 140

给定字符串 *S* 和单词字典 *words*, 求 *words[i]* 中是 *S* 的**子序列的单词个数**。

示例:输入:

```
S = "abcde"
```

```
words = ["a", "bb", "acd", "ace"]
```
输出: 3 **解释:** 有三个是 S 的子序列的单词: "a", "acd", "ace"。

注意:

- 所有在 *words* 和 *S* 里的单词都只由小写字母组成。
- *S* 的长度在 *[1, 50000]*。
- *words* 的长度在 *[1, 5000]*。
- *words[i]*的长度在*[1, 50]*。

```
class Solution {
```

```
public:
    int numMatchingSubseq(string S, vector<string>& words) {
        int res = 0, j;
        for (int i = 0; i < words.size(); i ++) {
            int position = -1;
            for (j = 0; j < words[i].size(); j ++) {
                position = S.find_first_of(words[i][j], position + 1); // 从下标 position + 1 开始遍历
                if (position == -1) break;  //若未找到弹出
            }
            if (j == words[i].length()) res ++; //表示 str 已全部被遍历了，则为其子串
        }
        return res;
    }
};
```

91. 解码方法（不行就返回 0，排除 0 的情况，分为合并和不合并，动态规划，dp[n]=f(dp[n-1]， dp[n-2])一个合并一个不合并（前一个和自身合并，前一个不和自身合并），还有等于 0 的情况，字母是 1-26，两位数字 10-19 ，20-26 是否合并，数字到字母,分类讨论，与 i-1 有关）

难度中等 650

一条包含字母 A-Z 的消息通过以下映射进行了 **编码** ：

```
'A' -> 1

'B' -> 2
```

...

'Z' -> 26

要 **解码** 已编码的消息，所有数字必须基于上述映射的方法，反向映射回字母（可能有多种方法）。例如，"111" 可以将 "1" 中的每个 "1" 映射为 "A"，从而得到 "AAA"，或者可以将 "11" 和 "1"（分别为 "K" 和 "A"）映射为 "KA"。**注意，"06" 不能映射为 "F"，因为 "6" 和 "06" 不同。**

给你一个只含数字的 **非空** 字符串 *num*，请计算并返回 **解码** 方法的 **总数** 。

题目数据保证答案肯定是一个 **32 位** 的整数。

示例 1：

输入：s = "12"**输出：**2 **解释：**它可以解码为 "AB"（1 2）或者 "L"（12）。

示例 2：

输入：s = "226"**输出：**3 **解释：**它可以解码为 "BZ" (2 26)，"VF" (22 6)，或者 "BBF" (2 2 6) 。

示例 3：

输入：s = "0"**输出：**0 **解释：**没有字符映射到以 0 开头的数字。含有 0 的有效映射是 'J' -> "10" 和 'T'-> "20" 。由于没有字符，因此没有有效的方法对此进行解码，因为所有数字都需要映射。

算法分析
DP 数组的含义是，当前 i 结尾，几种解码方式
pre i-2 curr i-1

源码
就像 pre curr 1 2 3（yeshicurrent 更新）

```
int numDecodings(string s) {
    if (s[0] == '0') return 0;
    int pre = 1, curr = 1;//dp[-1] = dp[0] = 1//
    for (int i = 1; i < s.size(); i++) {
```

```
            int tmp = curr;//记录 second
            if (s[i] == '0')//如果当前为 0，只能是 10 20 啥的，如果其他就返回 0
                if (s[i - 1] == '1' || s[i - 1] == '2') curr = pre;
                else return 0;
            else if (s[i - 1] == '1' || (s[i - 1] == '2' && s[i] >= '1' && s[i] <= '6'))
                curr = curr + pre;//这里是合并的 11-19 21-26，把两种情况相加 second+first
else(){current=current}//其他情况没有合并，就是 pre=curr=temp 包含了
            pre = tmp;//更新
        }
        return curr;
}
```

作者：pris_bupt
链接：https://leetcode-cn.com/problems/decode-ways/solution/c-wo-ren-wei-hen-jian-dan-zhi-guan-de-jie-fa-by-pr/
来源：力扣（LeetCode）

255. 验证前序遍历序列二叉搜索树（和判定一棵树是否是有效二叉树类似，需要界定最小值与最大值，因为左右在两边区间的两边，就不需要判定 left<right 了，递归，先序遍历是每次取自身，加上左右，之后继续去左边知道没了，那么比当前节点大的肯定是右节点，比当前节点小的肯定是左节点（如果左右都存在，防止不存在，用下标索引来记录）从当前节点到比当前节点大的点（右节点），就是，当前点都右节点之间都是左节点，右节点之后的都是

右节点，第一个比当前节点小的就是左，大的就是右，递归求解。递归求解右+1,end，到start+1,i-1的，出口就是start>end，这是左右树的遍历，前序遍历是node left right，所以第一个比当前大的节点就是右节点，在中间的就是左子树，右边的就是右子树，不断递归！！！）

难度中等 87

给定一个整数数组，你需要验证它是否是一个二叉搜索树正确的先序遍历序列。

你可以假定该序列中的数都是不相同的。

参考以下这颗二叉搜索树：

```
     5
    / \
   2   6
  / \
 1   3
```

示例 1：

输入： `[5,2,6,1,3]`**输出：** `false`

示例 2：

输入：**[5,2,1,3,6]**输出：**true//这个二叉树的前序遍历**

解题思路

解法 1：递归。调用一个 helper 函数 verify,保存递归到当前的**最大值 maxVal 和最小值 minVal。起点 start 和终点 end**。如果 start > end **说明为空树，**返回 true。否则若 rootVal 大于最大值 maxVal 或小于最小值 minVal 直接返回 false。否则查找第一个大于 **rootVal 的值为其右子树**。返回递归判断左右子树是否为二叉搜索树。

//递归

（start 一直在前进）

```
class Solution {
public:
    bool verifyPreorder(vector<int>& preorder) {
        return verify(preorder, 0, preorder.size() - 1, numeric_limits<int>::min(), numeric_limits<int>::max());
    }

private:
    bool verify(vector<int>& preorder, int start, int end, int minVal, int maxVal) {
        if (start > end) {//当前遍历完了
            return true;
        }

        int rootVal = preorder[start];//判断自身
        if (rootVal <= minVal || rootVal >= maxVal) {
            return false;
        }

        int i = start + 1;//从第二个开始查找 找右节点，保证一个节点都会走过。
        for (; i <= end; i++) {
            if (preorder[i] >= rootVal) {//找到比当前大的，就是右节点
                break;//看看 i 等于几，end 可以取到的。
            }
        }
//继续递归
        return verify(preorder, start + 1, i - 1, minVal, rootVal) && //root 的左子树
            verify(preorder, i, end, rootVal, maxVal);//root 的右子树
    }
};
```

//迭代（不好懂）

```
class Solution {
public:
    bool verifyPreorder(vector<int>& preorder) {
        int low = numeric_limits<int>::min();//给定一个最小值
        int i = -1;
        for (const auto& num : preorder) {
            if (num < low) {
                return false;
            }
            while (i >= 0 && num > preorder[i]) {
                low    = preorder[i--];//找到当前的右
            }
            preorder[++i] = num;//左
        }

        return true;
    }
};
```

作者：jyj407
链接：https://leetcode-cn.com/problems/verify-preorder-sequence-in-binary-search-tree/solution/zhong-gui-zhong-ju-255-yan-zheng-qian-xu-lwnq/
来源：力扣（LeetCode）

305. 岛屿数量 II（每次增加后的岛屿数量，直接深度搜索，dfs，回溯就是还要回来，盲走。）

难度困难 73

假设你设计一个游戏，用一个 *m* 行 *n* 列的 2D 网格来存储你的游戏地图。

起始的时候，每个格子的地形都被默认标记为「水」。我们可以通过使用 *addLand* 进行操作，将位置 *(row, col)* 的「水」变成「陆地」。

你将会被给定一个列表，来记录所有需要被操作的位置，然后你需要返回计算出来 **每次 addLand 操作后岛屿的数量**。

注意：一个岛的定义是被「水」包围的「陆地」，通过水平方向或者垂直方向上相邻的陆地连接而成。你可以假设地图网格的四边均被无边无际的「水」所包围。

请仔细阅读下方示例与解析，更加深入了解岛屿的判定。

示例:

输入: `m = 3, n = 3, positions = [[0,0], [0,1], [1,2], [2,1]]`**输出:** `[1,1,2,3]`

解析:

起初，二维网格 *grid* 被全部注入「水」。（`0` 代表「水」，`1` 代表「陆地」）

```
0 0 0

0 0 0

0 0 0
```

操作 #1：*addLand(0, 0)* 将 *grid[0][0]* 的水变为陆地。

```
1 0 0

0 0 0   Number of islands = 1

0 0 0
```

操作 #2：*addLand(0, 1)* 将 *grid[0][1]* 的水变为陆地。

```
1 1 0

0 0 0   岛屿的数量为 1
```

```
0 0 0
```

操作 #3：*addLand(1, 2)* 将 *grid[1][2]* 的水变为陆地。

```
1 1 0

0 0 1    岛屿的数量为 2

0 0 0
```

操作 #4：*addLand(2, 1)* 将 *grid[2][1]* 的水变为陆地。

传统方法（每次计算岛屿数量,这里不能，grid[0]表示未访问，因为你得保留之前的 1 啊，每次 push 进来。）

```
class Solution {
private:
    void dfs(vector<vector<char>>& grid, int r, int c, vector<vector<bool>>& visited) {
        int nr = grid.size();
        int nc = grid[0].size();

        if (r < 0 || c < 0 || r >= nr || c >= nc || grid[r][c] == '0' || visited[r][c]) return;

        visited[r][c] = true;
        dfs(grid, r - 1, c, visited);
        dfs(grid, r + 1, c, visited);
        dfs(grid, r, c - 1, visited);
        dfs(grid, r, c + 1, visited);
    }

    int numIslands(vector<vector<char>>& grid) {
        int nr = grid.size();
        int nc = grid[0].size();

        vector<vector<bool>> visited (nr, vector<bool>(nc, false));
        int num_islands = 0;
```

```
        for (int r = 0; r < nr; ++r) {
            for (int c = 0; c < nc; ++c) {
                if (grid[r][c] == '1' && !visited[r][c]) {//没有被访问，之前是把 1 变成 0
                    ++num_islands;
                    dfs(grid, r, c, visited);
                }
            }
        }

        return num_islands;
    }

public:
    vector<int> numIslands2(int m, int n, vector<pair<int, int>>& positions) {
        vector<int> ans;
        vector<vector<char>> grid (m, vector<char>(n, '0'));
        for (auto pos : positions) {
            grid[pos.first][pos.second] = '1';
            ans.push_back(numIslands(grid));//每次将结果给 push 进来
        }

        return ans;
    }
};
```

方法 3：并查集（搞这么复杂干嘛？）

想法

把二维的网格图当做一个无向图（以邻接矩阵的方式组织），横向或者纵向相邻的节点之间有一条**值为 1 的边**，那么问题就变成了**每次 addLand 操作之后在图中寻找连通部分的问题。**

算法

使用并查集这一数据结构，大小为 m*n ，在图中保存所有的节点，并初始化每个节点的父节点为 -1 表示一个空的图。我们的目标是在每次 addLand 操作以后用**新的陆地更新并查集并维护每块陆地所属的岛屿。**

对于每个在 (row, col) 的 addLand 操作，将**它与邻居合并**。如果它的邻居没有陆地，就初始化一个新的岛屿（将**父节点设为它自己**）。

以下动画可以更好地说明此算法（包括并查集如何进行 路径压缩 和 按秩合并）

```
class UnionFind {
public:
    UnionFind(int N) {
        count = 0;
        for (int i = 0; i < N; ++i) {
            parent.push_back(-1);
            rank.push_back(0);
        }
    }

    bool isValid(int i) const {
        return parent[i] >= 0;
    }

    void setParent(int i) {
        parent[i] = i;
        ++count;
    }

    int find(int i) { // path compression
        if (parent[i] != i) parent[i] = find(parent[i]);
        return parent[i];
    }

    void Union(int x, int y) { // union with rank
        int rootx = find(x);
        int rooty = find(y);
        if (rootx != rooty) {
            if (rank[rootx] > rank[rooty]) parent[rooty] = rootx;
            else if (rank[rootx] < rank[rooty]) parent[rootx] = rooty;
            else {
                parent[rooty] = rootx; rank[rootx] += 1;
            }
            --count;
        }
    }

    int getCount() const {
        return count;
    }

private:
    vector<int> parent;
    vector<int> rank;
```

```
    int count; // # of connected components
};

class Solution {
public:
    vector<int> numIslands2(int m, int n, vector<pair<int, int>>& positions) {
        vector<int> ans;
        UnionFind uf (m * n);

        for (auto& pos : positions) {
            int r = pos.first;
            int c = pos.second;
            // check pos's nei***ors to see if they are in the existing islands or not
            vector<int> overlap; // how many existing islands overlap with 'pos'
            if (r - 1 >= 0 && uf.isValid((r-1) * n + c)) overlap.push_back((r-1) * n + c);
            if (r + 1 < m && uf.isValid((r+1) * n + c)) overlap.push_back((r+1) * n + c);
            if (c - 1 >= 0 && uf.isValid(r * n + c - 1)) overlap.push_back(r * n + c - 1);
            if (c + 1 < n && uf.isValid(r * n + c + 1)) overlap.push_back(r * n + c + 1);

            int index = r * n + c;
            uf.setParent(index);
            for (auto i : overlap) uf.Union(i, index);
            ans.push_back(uf.getCount());
        }

        return ans;
    }
};
```

作者：LeetCode
链接：https://leetcode-cn.com/problems/number-of-islands-ii/solution/dao-yu-shu-liang-ii-by-leetcode/
来源：力扣（LeetCode）

505. 迷宫 II（遇到墙壁前不会停止滚动，那就是使劲的动咯。直道墙壁，这里不像一般的 BFS 一样，是之前到达当前的总距离，小于当前的，就加入，一

般的每一步是平行的（步数一致），这里求得是距离，不是步数，不是一格格走，而是惯性走）

难度中等 70

由空地和墙组成的迷宫中有一个**球**。球可以向**上下左右**四个方向滚动，但在**遇到墙壁前不会停止滚动**。当球停下时，可以选择下一个方向。

给定球的**起始位置，目的地**和**迷宫，**找出让球停在**目的地的最短距离**。距离的定义是球从起始位置（不包括）到目的地（包括）经过的空地**个数**。如果球无法停在目的地，返回 -1。

迷宫由一个 0 和 1 的二维数组表示。 1 表示墙壁，0 表示空地。你可以假定迷宫的边缘都是墙壁。起始位置和目的地的坐标通过行号和列号给出。

示例 1:

输入 1: 迷宫由以下二维数组表示

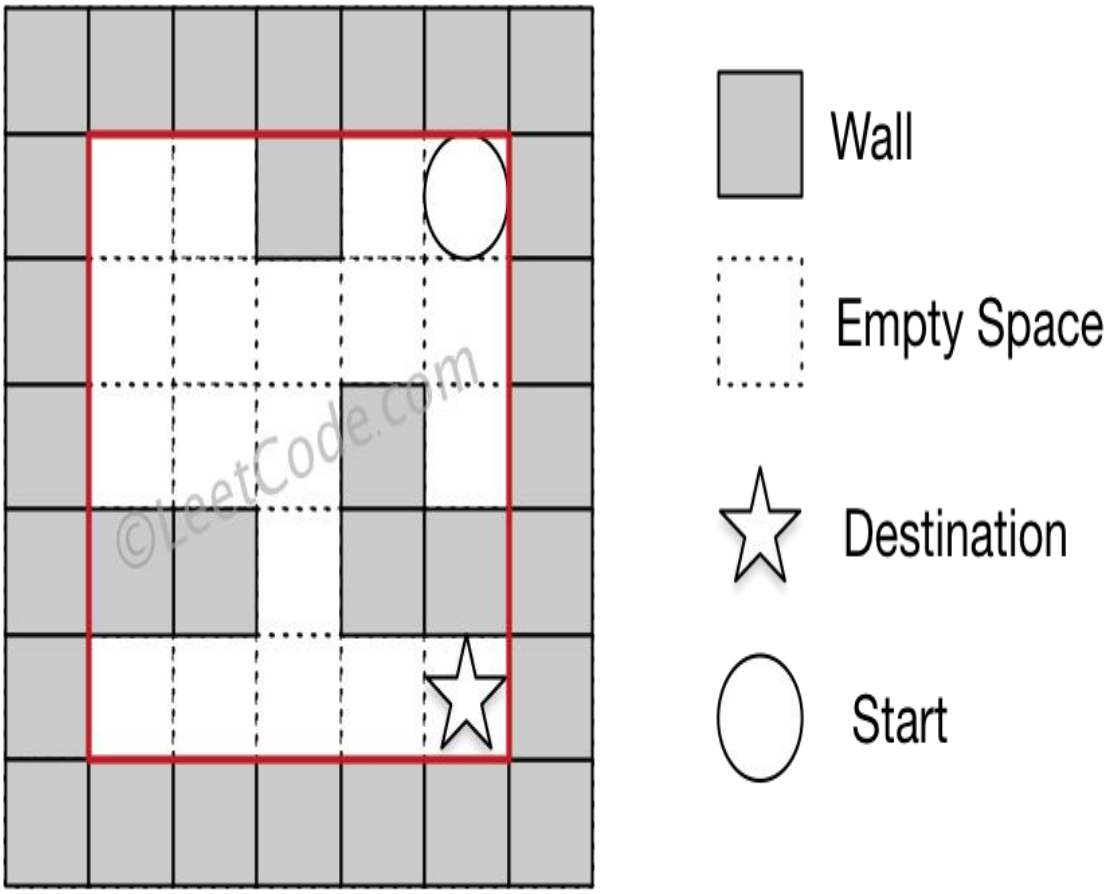
```
0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0
```

输入 2: 起始位置坐标 (rowStart, colStart) = (0, 4)**输入 3:** 目的地坐标 (rowDest, colDest) = (4, 4)

输出: 12

解析： 一条最短路径 : left -> down -> left -> down -> right -> down -> right。

总距离为 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12。



我们同样可以使用广度优先搜索，实现细节与深度优先搜索类似。

注意在一般的广度优先搜索中，**我们不会经过同一个节点超过一次，但在这道题目中，只要从起始位置到当前节点的步数 count 小于之前记录的最小步数 distance[i, j]，我们就会把 (i, j) 再次加入队列中。**

作者：LeetCode
链接：https://leetcode-cn.com/problems/the-maze-ii/solution/mi-gong-ii-by-leetcode/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

```
class Solution {
public:
    int shortestDistance(vector<vector<int>>& maze, vector<int>& start, vector<int>& destination) {
        if (maze.empty() || maze[0].empty()) {
            return -1;
```

```
        }//看看是否为空

        int M = maze.size();// row
        int N = maze[0].size();//column
       // vector<vector<bool>> visited(M, vector<bool>(N, false));  不要了
        vector<vector<int>> distance(M, vector<int>(N, INT_MAX));//每一步，对应的距离，最后得到 destination[0][1]的距离
        vector<vector<int>> dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};//四个方向，可选择的
        queue<vector<int>> q;//可以存储 vector 的队列
        q.push(start);//也作为一个向量
        distance[start[0]][start[1]] = 0;

        while (!q.empty()) {
            for (int k = q.size(); k > 0; k--) {//这里也要有 size，但是不更新步数了
                auto cur = q.front(); q.pop();
                int i = cur[0];
                int j = cur[1];

                for (const auto& dir : dirs) {
                    auto ni = i;
                    auto nj = j;
                    int count = 0;
                    while (ni >= 0 && ni < M && nj >= 0 && nj < N && maze[ni][nj] == 0) {//不是墙壁，不是边界的，可以移动，先使劲走一步，走到头
                        count++;//没走一步，加一步
                        ni += dir[0];
                        nj += dir[1];
                    }
                    count--;
                    ni -= dir[0];
                    nj -= dir[1];//到头了，需要回退一下

                    if (count + distance[i][j] < distance[ni][nj]) {//只要距离比当前小，就加入
                        distance[ni][nj] = count + distance[i][j];
                        q.push({ni, nj});//将这个加入
                    }
                }
            }
        }

        return distance[destination[0]][destination[1]] == INT_MAX ? -1 : distance[destination[0]][destination[1]];
    }
```

};

作者：LeetCode
链接：https://leetcode-cn.com/problems/the-maze-ii/solution/mi-gong-ii-by-leetcode/
来源：力扣（LeetCode）

169. 多数元素（就像众数一样，用一个哈希表搞定，每次更新的是元素，不是次数）

难度简单 925

给定一个大小为 n 的数组，找到其中的多数元素。多数元素是指在数组中出现**次数** 大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

示例 1：

输入：[3,2,3]**输出：**3

示例 2：

输入：[2,2,1,1,1,2,2]**输出：**2

进阶：

- 尝试设计时间复杂度为 O(n)、空间复杂度为 O(1) 的算法解决此问题

方法一：哈希表
思路

我们知道出现次数最多的元素大于 \lfloor \dfrac{n}{2} \rfloor

2
n

」 次，所以可以用哈希表来快速统计每个元素出现的次数。

算法

我们使用哈希映射（HashMap）来存储每个元素以及出现的次数。对于哈希映射中的每个键值对，键表示一个元素，值表示该元素出现的次数。

我们用一个循环遍历数组 nums 并将数组中的每个元素加入哈希映射中。在这之后，我们遍历哈希映射中的所有键值对，返回值最大的键。我们同样也可以在遍历数组 nums 时候使用打擂台的方法，维护最大的值，这样省去了最后对哈希映射的遍历。

JavaPython3C++

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        unordered_map<int, int> counts;
        int majority = 0, cnt = 0;
        for (int num: nums) {
            ++counts[num];
            if (counts[num] > cnt) {
                majority = num;//每次更新最大值对应的元素
                cnt = counts[num];
            }
        }
        return majority;
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/majority-element/solution/duo-shu-yuan-su-by-leetcode-solution/
来源：力扣（LeetCode）

剑指 Offer 21. 调整数组顺序使奇数位于偶数前面(用双指针+swap,

nums[fast] & 1 判断奇数因为会得到 1 除以 2>>1 右移一位)

难度简单 100

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

示例：

输入：`nums = [1,2,3,4]`**输出：**`[1,3,2,4]` **注：**`[3,1,2,4]` 也是正确的答案之一。

```
class Solution {
public:
    vector<int> exchange(vector<int>& nums) {
        int low = 0, fast = 0;
        while (fast < nums.size()) {
            if (nums[fast] & 1) {//奇数丢到前面交换 left 是空或者偶数）
                swap(nums[low], nums[fast]);
                low ++;
            }
            fast ++;
        }
        return nums;
    }
};
```

0 到末尾 一起（非 0 丢到前面，交换 left 是空或者 0）

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int n = nums.size(), left = 0, right = 0;
        while (right < n) {
            if (nums[right]) {//不是 0，就给 left，并且 left 要++
                swap(nums[left], nums[right]);//如果不是 0，就交换
```

```
                left++;//要往前,等待位置
            }
            right++;
        }
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/move-zeroes/solution/yi-dong-ling-by-leetcode-solution/
来源：力扣（LeetCode）

两个比较要一前一后（删除重复的东西，要更新）

删除有序重复数

```
int low = 0, fast = 1;//先走一步
        while (fast < nums.size()) {
            if (nums[fast]！=num[low]) {//与计算 1，奇数不等于 0
                low ++;//先加加有位置，等待位置
num[slow]=num[fast]//更新
            }
            fast ++;
        }
```

链表

```
int low = head, fast =head.next;//先走一步
        while (fast < nums.size()) {
            if (low.val!=fast.val) {//与计算 1，奇数不等于 0
low.next=fast
low=low.next
                low ++;
num[slow]=num[fast]
            }
            fast =fast.next;
        }
```

作者：huwt
链接：https://leetcode-cn.com/problems/diao-zheng-shu-zu-shun-xu-shi-qi-shu-wei-yu-ou-shu-qian-mian-lcof/solution/ti-jie-shou-wei-shuang-zhi-zhen-kuai-man-shuang-zh/
来源：力扣（LeetCode）

剑指 Offer 10- II. 青蛙跳台阶问题（初始值 f[0]=1）

难度简单 139

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

答案需要取模 1e9+7（1000000007），如计算初始结果为：1000000008，请返回 1。

示例 1：

输入：n = 2 **输出：**2

示例 2：

输入：n = 7 **输出：**21

示例 3：

输入：n = 0 **输出：**1

设跳上 nn 级台阶有 f(n)f(n) 种跳法。在所有跳法中，青蛙的最后一步只有两种情况： 跳上 11 级或 22 级台阶。
当为 11 级台阶： 剩 n-1n−1 个台阶，此情况共有 f(n-1)f(n−1) 种跳法；
当为 22 级台阶： 剩 n-2n−2 个台阶，此情况共有 f(n-2)f(n−2) 种跳法。
f(n)f(n) 为以上两种情况之和，即 f(n)=f(n-1)+f(n-2)f(n)=f(n−1)+f(n−2) ，以上递推性质为斐波那契数列。本题可转化为 求斐波那契数列第 nn 项的值 ，与 面试题 10- I. 斐波那契数列 等价，唯一的不同在于起始数字不同。
青蛙跳台阶问题： f(0)=1f(0)=1 , f(1)=1f(1)=1 , f(2)=2f(2)=2 ；
斐波那契数列问题： f(0)=0f(0)=0 , f(1)=1f(1)=1 , f(2)=1f(2)=1 。

作者：jyd
链接：https://leetcode-cn.com/problems/qing-wa-tiao-tai-jie-wen-ti-lcof/solution/mian-shi-ti-10-ii-qing-wa-tiao-tai-jie-wen-ti-dong/
来源：力扣（LeetCode）

1108. IP 地址无效化（如果是.就是+[.]否则就是加上自身）

难度简单 66

给你一个有效的 IPv4 地址 *address*，返回这个 IP 地址的无效化版本。

所谓无效化 IP 地址，其实就是用 "*[.]*" 代替了每个 "."。

示例 1：

输入：`address = "1.1.1.1"`**输出：**`"1[.]1[.]1[.]1"`

示例 2：

输入：`address = "255.100.50.0"`**输出：**`"255[.]100[.]50[.]0"`

```
class Solution {
public:
    string defangIPaddr(string address) {
        string res;

        for (char ch : address)
        {
            if (ch == '.') res += "[.]";
            else res += ch;
        }

        return res;
    }
};
```

作者：wo-yao-chu-qu-luan-shuo
链接：https://leetcode-cn.com/problems/defanging-an-ip-address/solution/ip-di-zhi-wu-xiao-hua-by-wo-yao-chu-qu-l-00ew/

来源：力扣（LeetCode）

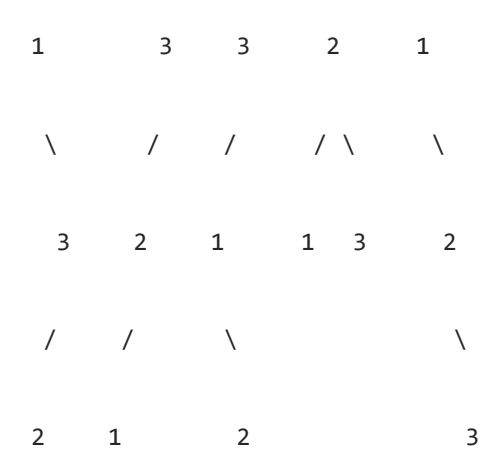
96. 不同的二叉搜索树（递推公式，每个节点为头结点，其他的任意组合 dp[i]+=dp[j-1][i-j]ij 都从 1 开始 j<=I,dp[j-1]代表几个元素在左边，用左右两边元素的大小，来间接表示，选取的头结点,除去自身相加等于 i-1;）

难度中等 1073

给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种？

示例:

输入： 3 **输出：** 5 **解释:**给定 n = 3，一共有 5 种不同结构的二叉搜索树：



96. 不同的二叉搜索树【动态规划】详解！

代码随想录 L6

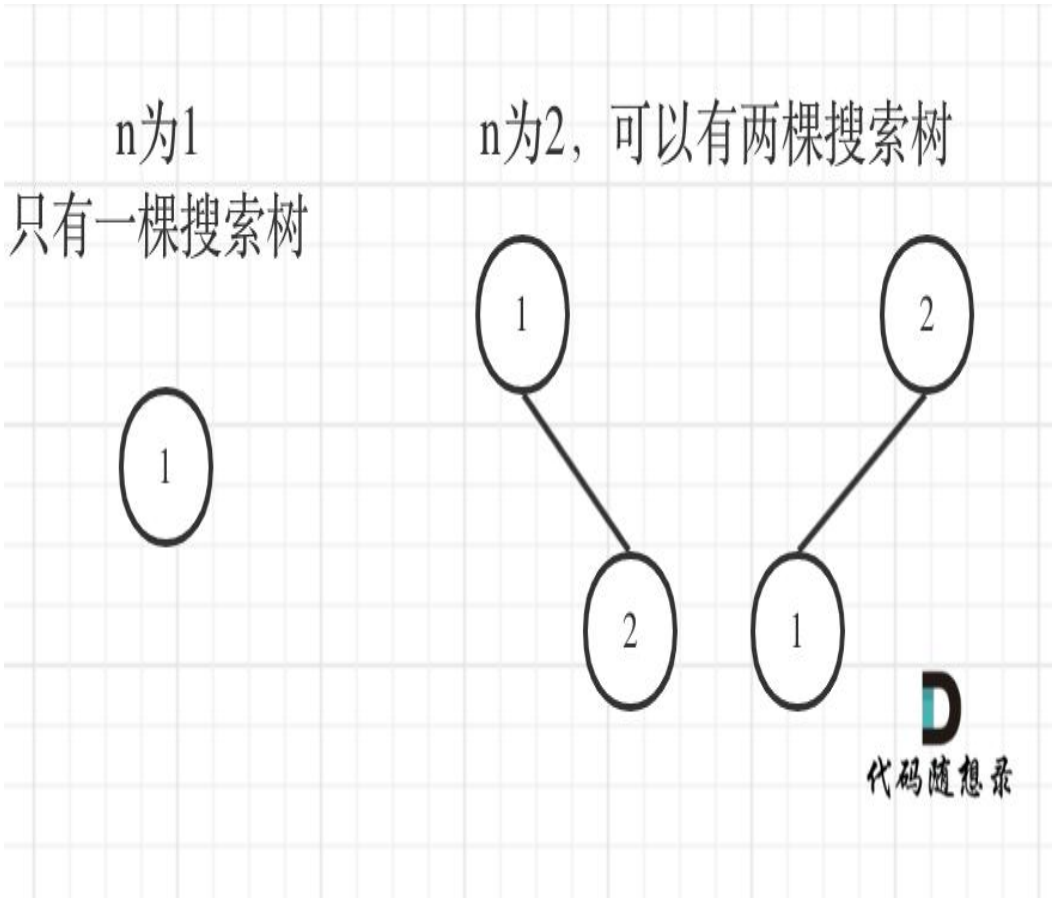
发布于 2021-01-071.3k 动态规划 C++

点击上方关注「代码随想录」，算法路上不迷路！

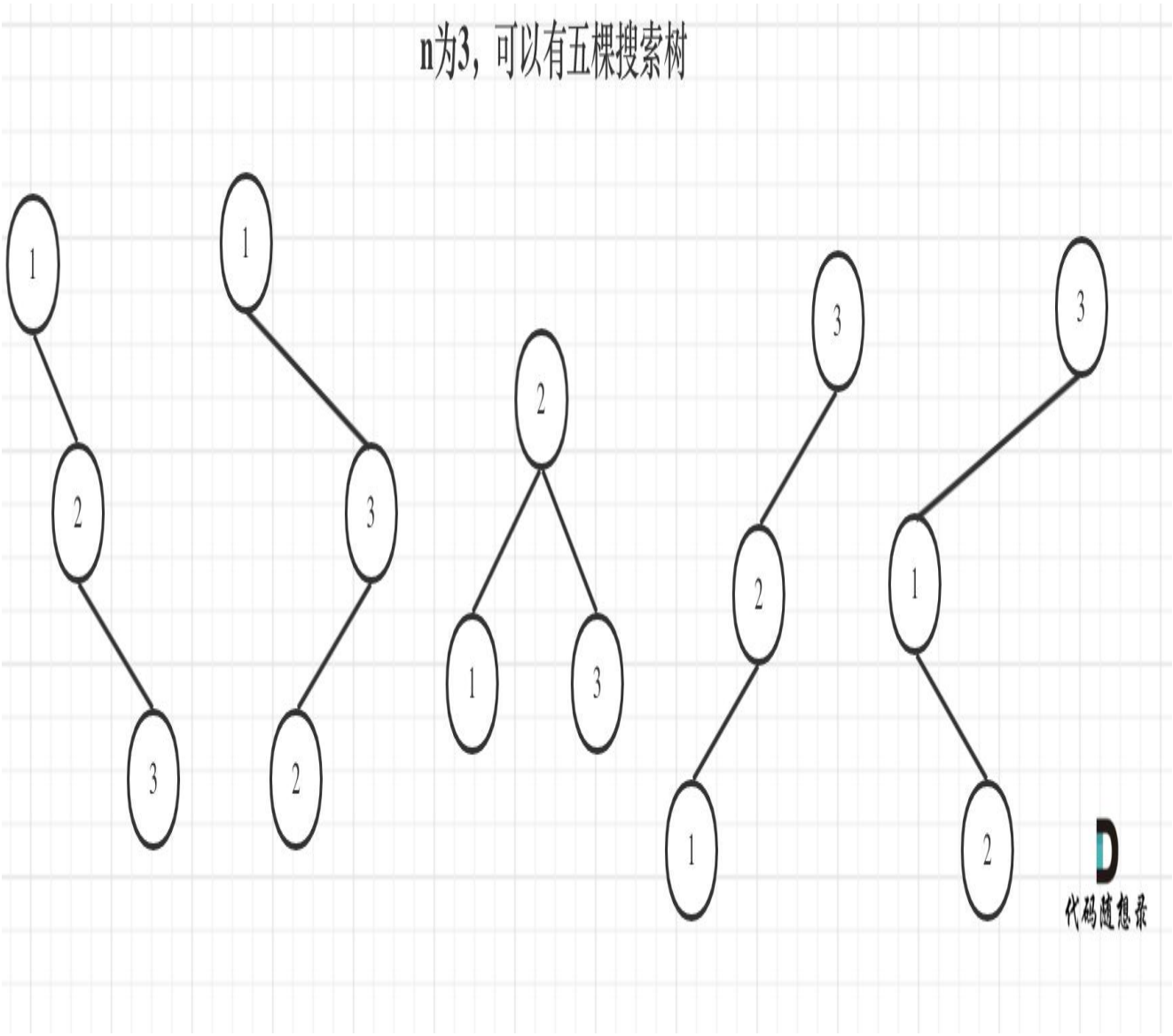
思路

这道题目很多估计很多同学刚一看都比较懵，这得怎么统计呢？

此时我们应该先举几个例子，画画图，看看有没有什么规律，如图：



n 为 1 的时候有一棵树，n 为 2 有两棵树，这个是很直观的。



来看看 n 为 3 的时候，有哪几种情况。

当 1 为头结点的时候，其右子树有两个节点，看这两个节点的布局，是不是和 n 为 2 的时候两棵树的布局是一样的啊！

（可能有同学问了，这布局不一样啊，**节点数值都不一样。别忘了我们就是求树的数量，并不用把搜索树都列出来，所以不用关心其具体数值的差异**）

当 3 为头结点的时候，其左子树有两个节点，看这两个节点的布局，是不是和 **n 为 2 的时候两棵树的布局也是一样的啊！**

当 2 位头结点的时候，其左右子树都只有一个节点，布局是不是和 n 为 1 的时候只有一棵树的布局也是一样的啊！

发现到这里，其实我们就找到的重叠子问题了，其实也就是发现可以通过 dp[1] 和 dp[2] 来推导出来 dp[3]的某种方式。

思考到这里，这道题目就有眉目了。

dp[3]，就是 元素 **1 为头结点搜索树的数量** + 元素 2 为头结点搜索树的数量 + 元素 3 为头结点搜索树的数量

元素 1 为头结点搜索树的数量 = 右子树有 2 个元素的搜索树数量 * 左子树有 0 个元素的搜索树数量

元素 2 为头结点搜索树的数量 = 右子树有 1 个元素的搜索树数量 * 左子树有 1 个元素的搜索树数量

元素 3 为头结点搜索树的数量 = 右子树有 0 个元素的搜索树数量 * 左子树有 2 个元素的搜索树数量

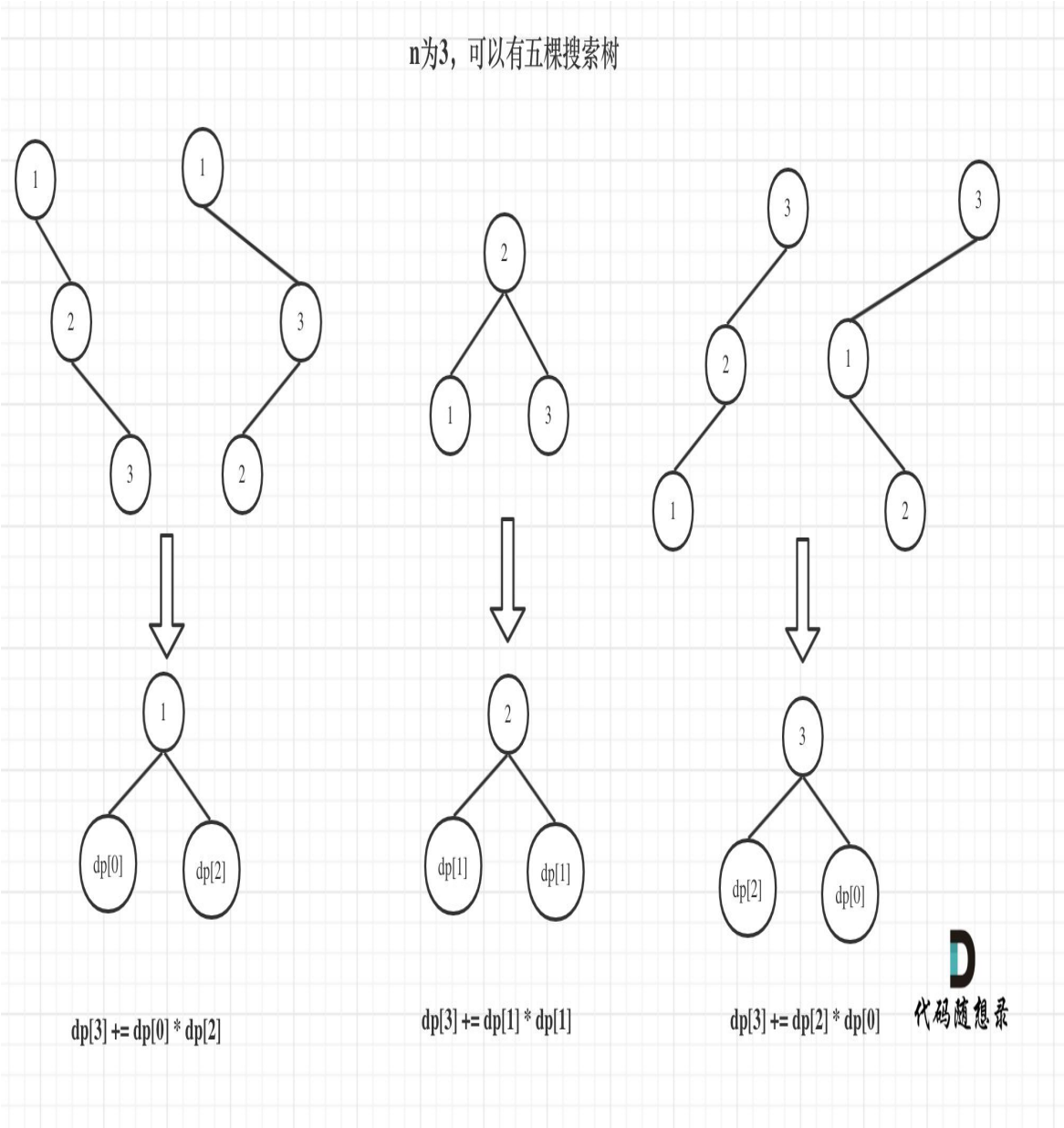
有 2 个元素的搜索树数量就是 dp[2]。

有 1 个元素的搜索树数量就是 dp[1]。

有 0 个元素的搜索树数量就是 dp[0]。

所以 dp[3] = dp[2] * dp[0] + dp[1] * dp[1] + dp[0] * dp[2]

如图所示：



此时我们已经找到的递推关系了，那么可以用动规五部曲在系统分析一遍。

1. 确定 dp 数组（dp table）以及下标的含义

dp[i] ： 1 到 i 为节点组成的二叉搜索树的个数为 dp[i]，也可以理解是 **i 的不同元素节点组成的二叉搜索树的个数为 dp[i] ，都是一样的。**

以下分析如果想不清楚，就来回想一下 dp[i]的定义

1. 确定递推公式

在上面的分析中，其实已经看出其递推关系， dp[i] += dp[以 **j 为头结点左子树节点数量] * dp[以 j 为头结点右子树节点数量]**

j 相当于是头结点的元素，从 1 遍历到 i 为止。

所以递推公式：**dp[i] += dp[j - 1] * dp[i - j];** ，j-1 为 j 为头结点左子树节点数量，i-j 为以 j 为头结点右子树节点数量

1. dp 数组如何初始化

初始化，只需要初始化 dp[0]就可以了，推导的基础，都是 dp[0]。

那么 dp[0]应该是多少呢？

从定义上来讲，空节点也是一个二叉搜索树，这是可以说得通的。

从递归公式上来讲，dp[以 j 为头结点左子树节点数量] * dp[以 j 为头结点右子树节点数量] 中以 j 为头结点左子树节点数量为 0，也需要 dp[以 j 为头结点左子树节点数量] = 1。

所以初始化 dp[0] = 1

1. 确定遍历顺序

首先一定是遍历节点数，依然节点数为 i 的状态是依靠 i 之前节点数的状态。

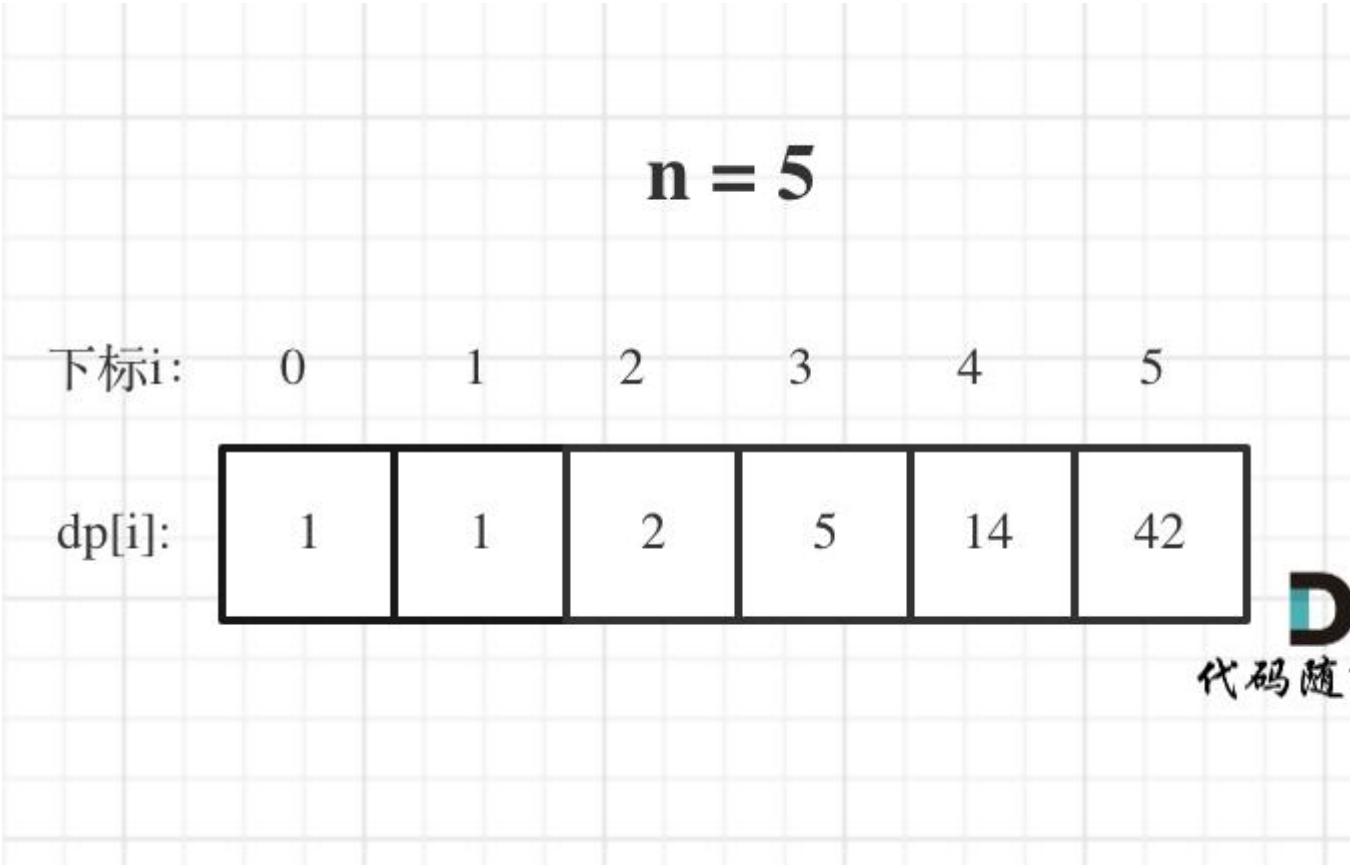
然后就是遍历 i 里面每一个数作为头结点的状态了，用 j 来遍历。

代码如下：

```
for (int i = 1; i <= n; i++) {

    for (int j = 1; j <= i; j++) {

        dp[i] += dp[j - 1] * dp[i - j];

    }

}
```

1. 举例推导 dp 数组

n 为 5 时候的 dp 数组状态如图：



当然如果自己画图举例的话，基本举例到 n 为 3 就可以了，n 为 4 的时候，画图已经比较麻烦了。

我这里列到了 n 为 5 的情况，是为了方便大家 debug 代码的时候，把 dp 数组打出来，看看哪里有问题。

综上分析完毕，C++代码如下：

```
class Solution {public:
    int numTrees(int n) {
        vector<int> dp(n + 1);
        dp[0] = 1;//表示空
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= i; j++) {//1-n 轮流当中间的
                dp[i] += dp[j - 1] * dp[i - j];//除去自身相加等于 i-1;
            }
        }
        return dp[n];
    }
};
```

421. 数组中两个数的最大异或值（抓住

异或的性质，相同的位才是 1，不同是 0，每次先找相反的位，如果位数不够用 0 来补。每次是还原相乘最大的数，！称为逻辑非运算符。用来逆转操作数的逻辑状态，如果条件为 true 则逻辑非运算符将使其为 false。！0=1）

难度中等 231
给定一个非空数组，数组中元素为 $a_0, a_1, a_2, \ldots , a_{n-1}$，其中 $0 \le a_i < 2^{31}$。0-31

找到 a_i 和 a_j 最大的异或 (XOR) 运算结果，其中 $0 \le i, j < n$ 。

你能在 O(*n*)的时间解决这个问题吗？

示例:

输入： `[3, 10, 5, 25, 2, 8]`

输出： `28`

解释： 最大的结果是 **5 ^ 25 = 28.**

C++：Trie 模板题（送板子）

algsCG*L5*

发布于 2021-02-23509 字典树 C++

思路：

- 前缀树解决，每个节点有 2 个分支：我们把数组中的每个元素看出一个 **32 位的 01 串（数值不足 32 在前面补 0）**，将 a0~an-1 对应的 32 位二进制串插入一棵 trie 树（最低位为叶子节点）。
- insert：由于我们要求两个元素的**异或最大值**，所以我们肯定是要从最高位开始考虑的，因此我们存 x 时**从左向右开始存储在 trie 中**。
- search：我们先在 trie 树中找到能与 x 异或取得最大值的另一个数组元素 y，我们采用**尽量走相反的 01 字符指针**的策略，因为异或的运算的法则是**相同得 0，不同得 1**，所以我们尽可能走与 x **当前位相反的字符方向走**，才能得到能和 x 产生最大值的另一个数组元素 y，然**后 res=x^y**。

```
class Trie
{
private:
    bool is_string=false;
    Trie *next[26]={nullptr};//26 个 next 指针
public:
    Trie(){}

    void insert(const string& word)//插入单词
    {
        Trie *root=this;//当前的，指针！
        for(const auto& w:word){
            if(root->next[w-'a']==nullptr)root->next[w-'a']=new Trie();//不相等建立
            root=root->next[w-'a'];
        }
        root->is_string=true;
    }

    bool search(const string& word)//查找单词
    {
        Trie* root=this;
        for(const auto& w:word){
            if(root->next[w-'a']==nullptr)return false;
            root=root->next[w-'a'];
        }
        return root->is_string;
    }

    bool startsWith(string prefix)//查找前缀
    {
        Trie* root=this;
        for(const auto& p:prefix){
```

```
                if(root->next[p-'a']==nullptr)return false;
                root=root->next[p-'a'];
            }
            return true;
        }
};
```

右移运算符>>:

1.无符号

语法格式：需要移位的数字>>移位的次数 n

运算规则：按二进制形式把所有数字向右移动相应的位数，低位移出（舍弃），高位的空位补 0。相当于**除以 2 的 n 次方 /2 的平方，右移两位，乘以 2 的平方，左移两位**

例如:4>>2 ，就是将数字 4 右移 2 位

过程：4 的二进制形式：00000000 00000000 00000000 00000100；然后把低位 2 个 0 移出，其余所有位向右移动 2 位，高位补 0，得到：00000000 00000000 00000000 00000001；十进制数为 1，1=**4÷2²**。

0X01 位操作基础

基本的位操作符有与、或、异或、取反、左移、右移这 6 种，它们的运算规则如下所示：

符号　　描述　　运算规则 by MoreWindows
&　与　两个位都为 1 时，结果才为 1
|　或　两个位都为 0 时，结果才为 0
^　异或　两个位相同为 0，相异为 1
~　取反　　0 变 1，1 变 0
<<　左移　　各二进位全部左移若干位，高位丢弃，低位补 0
>>　右移　　各二进位全部右移若干位，对无符号数，高位补 0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补 0（逻辑右移）

按位与操作

wcg_jishuo 2011-07-25 15:04:13　7347　收藏
分类专栏：　computer

按位与运算符"&"是双目运算符。其功能是参与运算的两数各对应的二进位相与。只有对应的两个二进位均为 1 时，**结果位才为 1，否则为 0**。参与运算的数以补码方式出现。

例如：9&5 可写算式如下：

00001001　　(9 的二进制补码)

&00000101　　(5 的二进制补码)

00000001　　(1 的二进制补码)

可见 9&5=1。

字典树解决

```
class Trie
{
private:
    Trie* next[2]={nullptr};// 构造 01
public:
    Trie(){}

    void insert(int x)   // 在前缀树中插入值 x
    {
        Trie *root=this;
        // 高位存储来 Trie 的前面，所以我们从左向右存储//最多 31
        for(int i=30;i>=0;i--)
        {
            // 取第 i 位的数字，30...0 右移动几位
            int u=x>>i&1; //最终看是 0 还是 1，这个位置就是最高位
            // 若第 u 位为空，则创建一个新节点，然后 root 移动到下一个节点
            if(!root->next[u])root->next[u]=new Trie();
            root=root->next[u];
        }
    }

    int srearch(int x)   // 在前缀树中寻找 x 的最大异或值（自身不参与，只是还原最大值，最后最异或运算，所以存储 0 不重要）
    {
        Trie *root=this;
        // res 表示最大异或值，每次 res*2 表示左移一位，31 循环后左移了 31 位了，+u 表示加上当前的最低位数字
        int res=0;
        for(int i=30;i>=0;i--)
        {
```

```
                int u=x>>i&1;//存在就是可以移动
                // 若 x 的第 u 位存在，我们走到相反的方向去，因为异或总是|值|相反才取最大值的（相同为 0，不同为 1，我们去找相反的）
                if(root->next[!u])root=root->next[!u],res=res*2+!u;
                // 相反方向的节点为空，只能顺着相同方向走了
                else root=root->next[u],res=res*2+u;（数值是这样的，每次乘以 2，就能还原了）
            }
            // 由于上面我们得到的异或另一个数组元素，此时我们需要将这个数组元素与 x 想异或得到 两个数的最大异或值
            res^=x;
            return res;//这就是最大值
        }
};

class Solution {
public:
    int findMaximumXOR(vector<int>& nums) {
        Trie *root=new Trie();
        for(auto x:nums)root->insert(x);
        int res=0;
        for(auto x:nums)
            res=max(res,root->srearch(x));//每次返回最大值
        return res;
    }
};
```

作者：xiaoneng
链接：https://leetcode-cn.com/problems/maximum-xor-of-two-numbers-in-an-array/solution/ctriemo-ban-ti-song-ban-zi-by-xiaoneng-tegw/
来源：力扣（LeetCode）

863. 二叉树中所有距离为 K 的结点

（这就是一个图，把每个节点的父节点，求出来，就可以一直走了左右+父，并且记录走的长度 BFS 框架找到等于的路径，不为空，或者没有访问，就放进去）

难度中等 258

给定一个二叉树（具有根结点 `root`）， 一个目标结点 `target` ，和一个整数值 `K` 。

返回到目标结点 `target` 距离为 `K` 的所有结点的值的列表。 答案可以以任何顺序返回。

示例 1：

输入： `root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2`

输出： `[7,4,1]`

解释：

所求结点为与目标结点（值为 `5`）距离为 `2` 的结点，

值分别为 `7`，`4`，以及 `1`

注意，输入的 `"root"` 和 `"target"` 实际上是树上的结点。

上面的输入仅仅是对这些对象进行了序列化描述。

BFS

```
class Solution {
public:
    vector<int> distanceK(TreeNode* root, TreeNode* target, int K) {
        unordered_map<TreeNode*, TreeNode*> parent;
        findParent(root, nullptr, parent);
        // 使用图的广度优先搜索
        queue<TreeNode*> que;
        unordered_set<TreeNode*> visited;
        que.push(target);
        visited.insert(target);//从当前 target 开始寻找，寻找，左右父
        vector<int> res;//存储满足条件的节点值
        int dist = 0;
        while ( !que.empty() ) {
            int n = que.size();//当前的大小
            while ( n-- ) {
                TreeNode* node = que.front();
                que.pop();
                if ( dist == K ) res.push_back(node->val);//距离相等，放进去
```

```
//父节点 和左右节点都放进来
                    if ( node->left != nullptr && visited.find(node->left) == visited.end() ) {
                        que.push(node->left);
                        visited.insert(node->left);//已经访问过了
                    }
                    if ( node->right != nullptr && visited.find(node->right) == visited.end() ) {
                        que.push(node->right);
                        visited.insert(node->right);
                    }
                    TreeNode* par = parent[node];//找当前节点的父节点
                    if ( par != nullptr && visited.find(par) == visited.end() ) {
                        que.push(par);
                        visited.insert(par);
                    }
                }
                ++dist;//对应刚才的那波的距离
                if ( !res.empty() ) break;//不为空，找到了就停止。就是这个距离，因为左右父在增加会增加
            }
            return res;
        }
        void findParent(TreeNode* node, TreeNode* par, unordered_map<TreeNode*, TreeNode*>& parent) {
            if ( node != nullptr ) {//只要当前节点不为空
                parent[node] = par;//找到每个节点的父节点
                findParent(node->left, node, parent);
                findParent(node->right, node, parent);
            }
        }
};
```

81. 搜索旋转排序数组 II（用标准的小于等于搜索，每次都看是否在升序空间内，不在的话，将升序空间给过滤掉，存在的话，就在里面，如果左右相等，去除重复项。，看 mid 在哪是有序，target 不在哪里，就把哪边砍掉）

难度中等 307
假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,0,1,2,2,5,6]` 可能变为 `[2,5,6,0,0,1,2]`)。

编写一个函数来判断给定的**目标值是否存在于数组中**。若存在返回 `true`，否则返回 `false`。

示例 1:

输入： `nums = [2,5,6,0,0,1,2], target = 0`

输出： `true`

示例 2:

输入： `nums = [2,5,6,0,0,1,2], target = 3`

输出： `false`

二分查找

解题思路：
本题是需要使用二分查找，怎么分是关键，举个例子：

第一类
10111 和 11101 这种。此种情况下 nums[start] == nums[mid]，**分不清到底是前面有序还是后面有序，此时 start++ 即可。相当于去掉一个重复的干扰项。**
第二类
2345671 这种，也就是 nums[start] < nums[mid]。此例子中就是 2 < 5；
这种情况下，**前半部分有序**。因此如果 nums[start] <=target<nums[mid]，**则在前半部分找，否则去后半部分找。**
第三类
6712345 这种，也就是 nums[start] > nums[mid]。此例子中就是 6 > 2；
这种情况下，**后半部分有序**。**因此如果 nums[mid] <target<=nums[end]。则在后半部分找，否则去前半部分找。**

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int n = nums.size();
        if (n == 0)
            return false;
```

```
        int left = 0;
        int right = n - 1;
        while (left <= right) {//左右可以越过
            int mid = ((right - left) >>1) + left;
            if (nums[mid] == target)
                return true;

            if (nums[mid] == nums[left]) {//去除重复干扰项
                ++left;//左边一直加  case 1
                continue;
            }

            if (nums[left] < nums[mid]) {//如果左边升序 left 到 mid 升序
                if (nums[left] <= target && target < nums[mid]) {//target 在中间  收缩右边
                    right = mid - 1;//右边收缩，在里面
                } else {
                    left = mid + 1;//否则在左边，左边一直加
                }
            } else {//如果右边升序，  mid 到 right 升序
                if (nums[mid] < target && target <= nums[right]) {//在里面，只在右边找，左边舍弃掉
                    left = mid + 1;//左边+，在里面
                } else {
                    right = mid - 1;//否则左边搜索
                }
            }
        }
        return false;
    }
};
```

98. 验证二叉搜索树（LONG_MIN, LONG_MAX，带图最大值和最小值很有新意）

难度中等 974

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含**小于**当前节点的数。

- 节点的右子树只包含**大于**当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

输入:

```
    2
   / \
  1   3
```

输出: `true`

示例 2:

输入:

```
    5
   / \
  1   4
     / \
    3   6
```

输出: `false`

解释: 输入为: `[5,1,4,null,null,3,6]`。

根节点的值为 `5` ，但是其右子节点值为 `4`

```
class Solution {
public:
    bool helper(TreeNode* root, long long lower, long long upper) {
        if (root == nullptr) {
            return true;
        }
        if (root -> val <= lower || root -> val >= upper) {
            return false;
        }
        return helper(root -> left, lower, root -> val) && helper(root -> right, root -> val, upper);
    }
    bool isValidBST(TreeNode* root) {
        return helper(root, LONG_MIN, LONG_MAX);
```

```
    }
};
```

二分法，搜索穷尽的时候，r=n-1,l<=r 保证（l=r 可以穷尽）标准，一般都是返回 l，除了求解平方根的 x,返回 r，因为 r 更小。

153. 寻找旋转排序数组中的最小值（旋转之后，前面是从小到大，后面是从小到大，前面的大于后面的，如果 mid 大于最后，表示，当前在左边就要前移 left=mid+1，如果小于，表示当前在右边，right=mid-1;最终取 l,在右边的。相等的肯定是最小值，之后 r 往左移动，l>r 了，退出用标准的小于等于搜索）

难度中等 374

假设按照升序排序的数组在预先未知的某个点上进行了旋转。例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]` 。

请找出其中最小的元素。

示例 1：

输入： `nums = [3,4,5,1,2]`

输出： `1`

示例 2：

输入： `nums = [4,5,6,7,0,1,2]`

输出： `0`

示例 3：

输入： `nums = [1]`

输出： `1`

二分法找旋转点，这个点所在的值即是最小值。
如果有重复数字的话，还需要多一个操作，删除末尾的重复数字：
while (n > 0 && nums[0] == nums[n]) n -- ;

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int n = nums.size();
        int l = 0, r = nums.size()-1;

        if (nums[0] < nums[n - 1]) return nums[0];   //数组完全有序，则返回第一个元素

        while (l <= r)//就不会少了一个 r>=l 了
        {
            int mid = (l + r) >> 1;
            if (nums[mid] <= nums.back()) r = mid-1;   //将中间值与数组末尾元素比较
            else l = mid + 1;//说明还在左边，当前数不可能最小，往前移动
        }

        return nums[l];//返回最小的。
    }
};
```

//你如果是 r=n l<=r ，r 就可以取到，索引 r 不能 mid-1,要 mid 作为开区间的边界。

在这个改进版本的二分搜索算法中，我们需要**找到这个点**。下面是关于变化点的特点：

所有变化点左侧元素 > **数组第一个元素**

所有变化点右侧元素 < **数组第一个元素**

找到数组的中间元素 mid。

如果中间元素 > 数组第一个元素，我们需要在 mid 右边搜索变化点。

如果中间元素 < 数组第一个元素，我们需要在 mid 左边搜索变化点。

nums[mid] > nums[mid + 1]，因此 mid+1 是最小值。

nums[mid - 1] > nums[mid]，因此 mid 是最小值。

C++ vector 的 reserve 和 resize 详解

vector 的 reserve 增加了 vector 的 capacity，但是它的 size 没有改变！而 resize 改变了 vector 的 capacity 同时也增加了它的 size！

原因如下：

reserve 是容器预留空间，但在空间内不真正创建元素对象，所以在没有添加新的对象之前，**不能引用容器内的元素**。加入新的元素时，要调用 push_back()/insert()函数。

resize 是改变容器的大小，且在**创建对象**，因此，调用这个函数之后，就可以引用容器内的对象了，因此当加入新的元素时，用 operator[]操作符，或者用迭代器来引用元素对象。此时再调用 push_back()函数，是加在这个新的空间后面的。

```
vector<vector<int>> a(10);//第一个 vector 可以用[]后面只能用 push_back,不是真正的创建元素

//    a.resize(10);



      cout << boolalpha <<a[1].empty()<< endl;//第二个是空集

      return 0;

}
```

207. 课程表（拓扑图排序（u:neighbor[..]），每个先深度搜索，如果没出现返回的情况，就是认为是 2，如果有就是 false，当某个节点还在搜索中，如果还存在被其他搜索，就肯定

存在环了。环有方向，并查集没有方向）

难度中等 759

你这个学期必须选修 *numCourses* 门课程，记为 *0* 到 *numCourses - 1* 。

在选修某些课程之前需要一些先修课程。先修课程按数组 *prerequisites* 给出，其中 *prerequisites[i] = [ai, bi]* ，表示如果要学习课程 ***ai*** **则 必须 先学习课程** ***bi*** 。

- 例如，先修课程对 *[0, 1]* 表示：想要学习课程 *0* ，你需要先完成课程 *1* 。

请你判断是否可能完成所有课程的学习？如果可以，返回 *true* ；否则，返回 *false* 。

示例 1：

输入：`numCourses = 2, prerequisites = [[1,0]]`**输出：**`true` **解释：**总共有 `2` 门课程。学习课程 `1` 之前，你需要完成课程 `0` 。这是可能的。

示例 2：

输入：`numCourses = 2, prerequisites = [[1,0],[0,1]]`**输出：**`false` **解释：**总共有 `2` 门课程。学习课程 `1` 之前，你需要先完成 课程 `0` ；并且学习课程 `0` 之前，你还应先完成课程 `1` 。这是不可能的。

```
class Solution {
private:
    vector<vector<int>> edges;
    vector<int> visited;
    bool valid = true;

public:
    void dfs(int u) {
        visited[u] = 1;//被访问了
```

```
        for (int v: edges[u]) {//有些到末尾了，就不存在了，直接 u=2
            if (visited[v] == 0) {
                dfs(v);
                if (!valid) {
                    return;//之前被访问了，退出
                }
            }
            else if (visited[v] == 1) {//被访问了，就存在 u v v u 一个环了=1 表示还在搜索中没玩
                valid = false;
                return;
            }
        }
        visited[u] = 2;//入栈不用管了，所有都遍历完了
    }

    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        edges.resize(numCourses);
        visited.resize(numCourses);
        for (const auto& info: prerequisites) {
            edges[info[1]].push_back(info[0]);//有 1 才有 2 的嘛？
        }
        for (int i = 0; i < numCourses && valid; ++i) {//必须有效，无效马上退出，发现环了。
            if (!visited[i]) {//没被访问
                dfs(i);
            }
        }
        return valid;
    }
};
```

通过上述的三种状态，我们就可以给出使用深度优先搜索得到拓扑排序的算法流程，在每一轮的搜索搜索开始时，我们任取一个「未搜索」的节点开始进行深度优先搜索。

我们将当前搜索的节点 u 标记为「搜索中」，遍历该节点的每一个相邻节点 v：

如果 v 为「未搜索」，那么我们开始搜索 v，**待搜索完成回溯到 u；**

如果 v 为「搜索中」，那么我们就找到了**图中的一个环**，因此是不存在拓扑排序的；

如果 v 为「已完成」，那么说明 v 已经在栈中了，而 u 还不在栈中，因此 u 无论何时入栈**都不会影响到 (u, v) 之前的拓扑关系，以及不用进行任何操作**。

当 u 的所有相邻节点都为「已完成」时，我们将 u 放入栈中，并将其标记为「已完成」。

在整个深度优先搜索的过程结束后，如果我们没有找到图中的环，那么栈中存储这所有的 nn 个节点，从栈顶到栈底的顺序即为一种拓扑排序。

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/course-schedule/solution/ke-cheng-biao-by-leetcode-solution/
来源：力扣（LeetCode）

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/course-schedule/solution/ke-cheng-biao-by-leetcode-solution/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

986. 区间列表的交集（双指针，左右移动，取两个区间的 start 的最大值和 end 的最小值作为结果，并且保证 start<=end，哪个 end 更小，就往前走，小的被覆盖了，当某个被遍历完了就停止，有意思）

难度中等 141

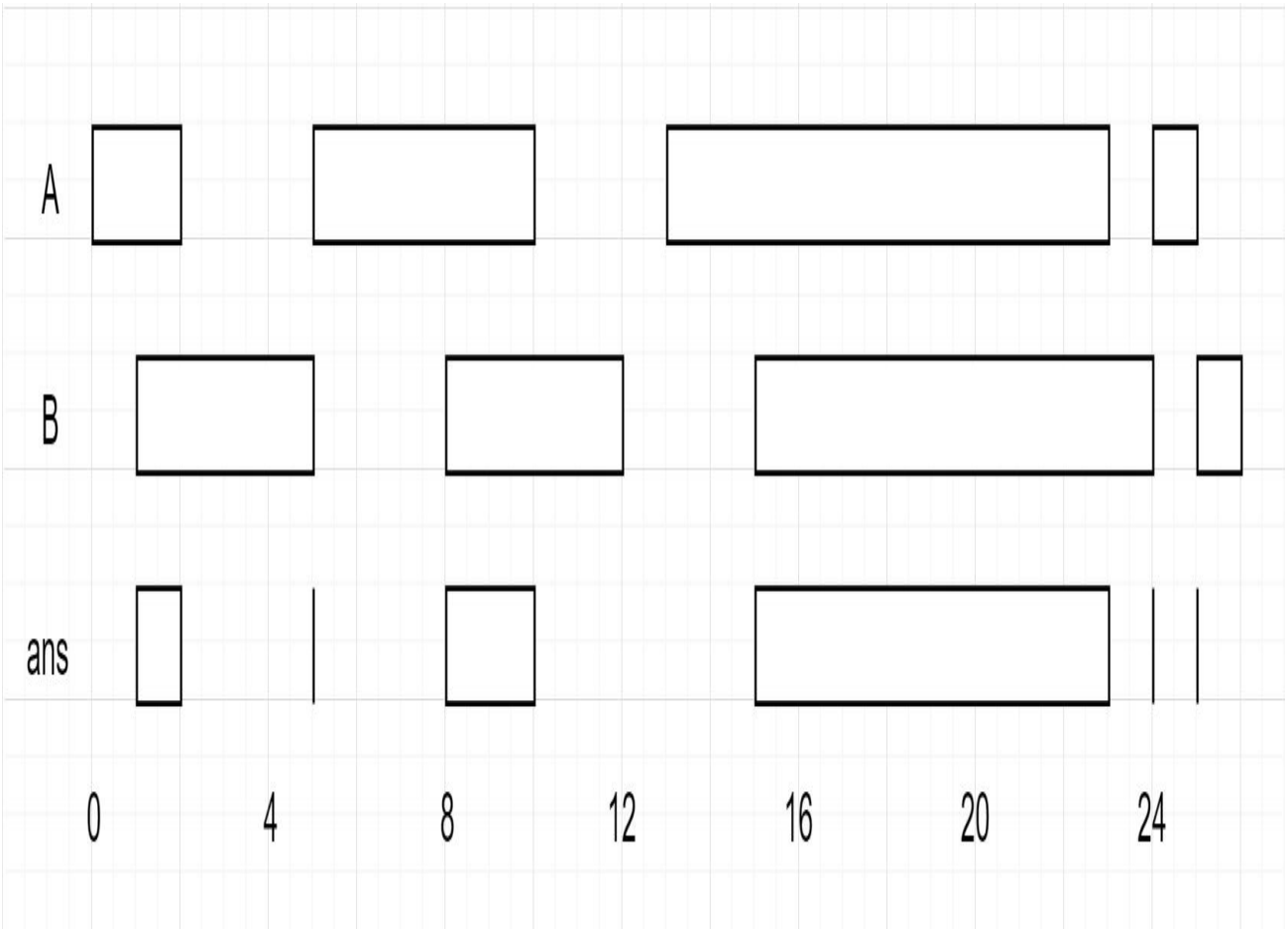
给定两个由一些 **闭区间** 组成的列表，*firstList* 和 *secondList* ，其中 *firstList[i] = [starti, endi]* 而 *secondList[j] = [startj, endj]* 。每个区间列表都是成对 ***不相交*** 的，并且 **已经排序** 。

返回这 两个区间列表的交集 。

形式上，**闭区间** *[a, b]*（其中 *a <= b*）表示实数 *x* 的集合，而 *a <= x <= b* 。

两个闭区间的 交集 **是一组实数，要么为空集，要么为闭区间**。例如，[1,3] 和 [2,4] 的交集为 [2,3] 。

示例 1：



输入：firstList = [[0,2],[5,10],[13,23],[24,25]], secondList = [[1,5],[8,12],[15,24],[25,26]]**输出：**[[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]

示例 2：

输入：firstList = [[1,3],[5,9]], secondList = []**输出：**[]

示例 3：

输入：`firstList = [], secondList = [[4,8],[10,12]]`**输出：**`[]`

示例 4：

输入：`firstList = [[1,7]], secondList = [[3,10]]`**输出：**`[[3,7]]`

解题思路

双指针 i,j 都从 0 开始。交集区间的 start 取当前 i，j 指向区间的起点的最大值，交集区间的 end 取当前 i，j 指向区间的起点的最小值，然后往前移动 i，j 之中**区间结尾值较小的一个**。直到有**一个遍历完所有列表区间元素。**

注意：**不能比较起点**，移动起点小的一个，比如下面就是一个反例，比较起点移动小的会丢失后面的两个区间[14, 15], [16, 20]。个人理解原因是即使当前的 start 大，但是只要其 end 小于另外一个 end，还有一定相交的，因此必须**比较结尾，移动 end 小的一个**。

[[3,5],[9,20]]

[[4,5],[7,10],[11,12],[14,15],[16,20]]

总结一下，关键在于移动 end 较小的一个指针。

代码

```
class Solution {
public:
    vector<vector<int>> intervalIntersection(vector<vector<int>>& A, vector<vector<int>>& B) {
        if (A.empty() || B.empty()) {
            return {};
        }
        vector<vector<int>> res;
        int i = 0;
        int j = 0;
        while (i < A.size() && j < B.size()) {
            int start = max(A[i][0], B[j][0]);//端点最大
            int end = min(A[i][1], B[j][1]);//末尾最小

            if (start <= end) {//必须是 start<=end，才放进来
```

```
                res.push_back({start, end});
            }
            // Move the pointer that points to smaller ending interval forward
            A[i][1] < B[j][1] ? i++ : j++;//末尾小的移动。末尾小的被覆盖了
        }

        return res;
    }
};
```

199. 二叉树的右视图（两个栈，一个存储节点，一个存储深度，先左后右的push，保证先访问右边，如果这层没有数值，更新数值，并且有个哈希表，深度和右视图的点，相当于把每层的东西给拿出来。

）

难度中等 434

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

示例:

输入: [1,2,3,null,5,null,4]输出: [1, 3, 4]解释:

```
   1            <---
 /   \
2     3         <---
 \     \
```

5　　4　　<---

文字题解

初步想法
由于树的形状无法提前知晓，不可能设计出优于 O(n)O(n) 的算法。因此，我们应该试着寻找线性时间解。带着这个想法，我们来考虑一些同等有效的方案。

方法一：深度优先搜索
思路

我们对树进行**深度优先搜索**，在搜索过程中，我们总是**先访问右子树**。那么对于每一层来说，我们在**这层见到的第一个结点一定是最右边的结点。**

算法

这样一来，我们可以存储在每个深度访问的第一个结点，一旦我们知道了树的层数，就可以得到最终的结果数组。

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        unordered_map<int, int> rightmostValueAtDepth;//最右边的值，哈希表
        int max_depth = -1;

        stack<TreeNode*> nodeStack;//节点

        stack<int> depthStack;//深度

        nodeStack.push(root);//放入根节点

        depthStack.push(0);//深度 0

        while (!nodeStack.empty()) {/ BFS+DFS 的意思
            TreeNode* node = nodeStack.top();nodeStack.pop();//节点拿出来

            int depth = depthStack.top();depthStack.pop();//深度拿出来

            if (node != NULL) {
              // 维护二叉树的最大深度
```

```
                max_depth = max(max_depth, depth);//更新最大深度，用作后面 vector 循环


                // 如果不存在对应深度的节点我们才插入，保证第一个就是要的
                if (rightmostValueAtDepth.find(depth) == rightmostValueAtDepth.end()) {
                    rightmostValueAtDepth[depth] =  node -> val;
                }

                nodeStack.push(node -> left);//左右

                nodeStack.push(node -> right);//因为是栈，所以先访问右

                depthStack.push(depth + 1);//左右另个深度

                depthStack.push(depth + 1);
            }
        }

        vector<int> rightView;
        for (int depth = 0; depth <= max_depth; ++depth) {
            rightView.push_back(rightmostValueAtDepth[depth]);//每一层的最右边放进来，就是
```

右视图

```
        }

        return rightView;
    }
};
```

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/binary-tree-right-side-view/solution/er-cha-shu-de-you-shi-tu-by-leetcode-solution/
来源：力扣（LeetCode）

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/binary-tree-right-side-view/solution/er-cha-shu-de-you-shi-tu-by-leetcode-solution/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

274. H 指数（排序+直方图从小到大排序，如果第一个小于总数，那么不可能存在 h 指数等于 n，那么将 h 只能往下减去 1 了。找正方形）

难度中等 138

给定一位研究者论文被引用次数的数组（被引用次数是非负整数）。编写一个方法，计算出研究者的 h 指数。

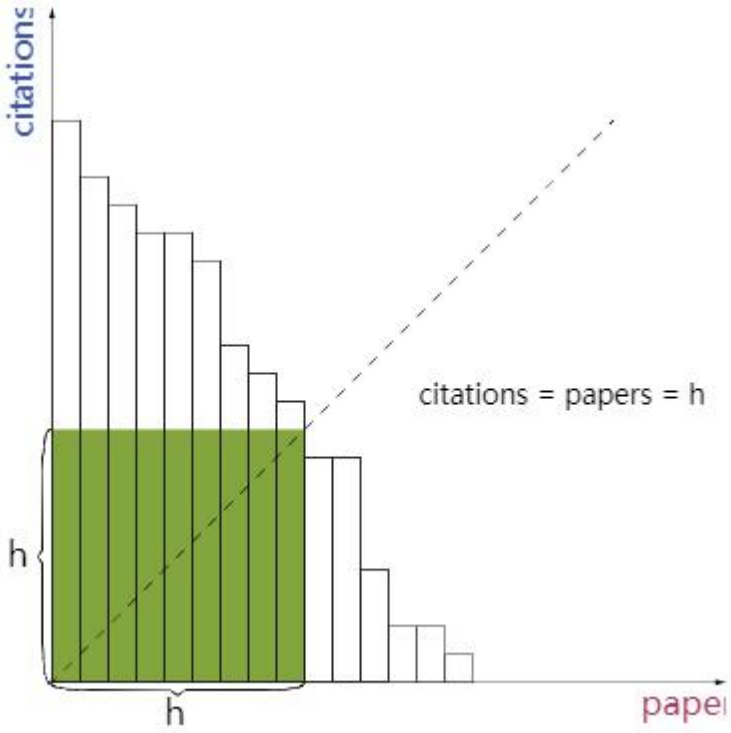
h 指数的定义：h 代表“高引用次数”（high citations），一名科研人员的 h 指数是指他（她）的 （N 篇论文中）**总共**有 h 篇论文分别被引用了至少 h 次。且其余的 N - h 篇论文每篇被引用次数 **不超过** h 次。

例如：某人的 **h 指数是 20，这表示他已发表的论文中，每篇被引用了至少 20 次的论文总共有 20 篇。**

示例：

输入：`citations = [3,0,6,1,5]`**输出：**`3` **解释：**给定数组表示研究者总共有 `5` 篇论文，每篇论文相应的被引用了 `3, 0, 6, 1, 5` 次。

由于研究者有 `3` 篇论文每篇 **至少** 被引用了 `3` 次，其余两篇论文每篇被**引用** 不多于 **`3` 次**，所以她的 `h` 指数是 `3`。



算法

提示：如果 h 有多种可能的值，h

```
class Solution {
public:
    int hIndex(vector<int>& citations) {
        int n = citations.size();
        if(n == 0){
            return 0;
        }//判断初始值

        int h = n;
        sort(citations.begin(),citations.end());//先排序，从小到大
        for(int i=0;i<citations.size();i++){
            if(citations[i] < h){//要一篇论文的引用次数小于总数，将总数-1
                h -= 1;
            }
        }
        return h;
    }
};
```

实参是程序中已经分配了内存空间的参数，它可以被赋予一个具体的值，比如常数、数组、地址（指针），也可以是一个变量名、数组名或表达式，当然也包括指针变量。

形参则是你在写一个被调函数时，为了说明用到的自变量的类型、要进行什么操作而定义的，在调用函数前它不会被分配内存空间，更不会被赋予具体的值。

调用函数时，形参会被分配一个新的内存空间，实参的值就会被“复制”进去，让它在被调函数中参与运算。而实参本身不参与这个运算，它仅仅起到一个传递值的作用（不过在 C++中可以用&改变实参的值）。如果参数的形式是指针，那么“复制”的就是地址。

举个非常简单的例子：

```
int func1(int a)
{
        a++;
        return a;
}

int main()
{
        int b = 5;
        printf("func1 = %d, ", func1(b));
        printf("b = %d\n", b);
        return 0;
}
```

输出 func1 = 6, b = 5，实参 b 的值并没有改变。

作者：「已注销」
链接：https://www.zhihu.com/question/276730270/answer/389068813
来源：知乎

传值和传引用、传指针的区别

海的来信 2014-04-18 09:50:51 12818 收藏 5
分类专栏： 杂学 文章标签： 指针
版权
原文链接：https://blog.csdn.net/a15994269853/article/details/23995383

剑指 Offer 12. 矩阵中的路径（给定随

意的位置，还有给定在单词中的位置，进行深度搜索和回溯，访问过标记为’\0’空字符，与什么都不相等）

难度中等 281

请设计一个函数，用来判断在一个矩阵中是否存在一条**包含某字符串**所有字符的路径。路径可以从矩阵中的**任意一格开始**，每一步可以在矩阵中**向左、右、上、下**移动一格。如果一条路径经过了矩阵的某一格，**那么该路径不能再次进入该格子**。例如，在下面的 3×4 的矩阵中包含一条字符串“bfce”的路径（路径中的字母用加粗标出）。

[["a","**b**","c","e"],
["s","**f**","**c**","s"],
["a","d","**e**","e"]]

但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

示例 1：

输入：`board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"`**输出：**`true`

示例 2：

输入：`board = [["a","b"],["c","d"]], word = "abcd"`**输出：**`false`

```
class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        rows = board.size();
        cols = board[0].size();
        for(int i = 0; i < rows; i++) {
            for(int j = 0; j < cols; j++) {

                if(dfs(board, word, i, j, 0)) return true;//随便选一个，作为起点

            }
        }

        return false;//都没有就是 false 了。

    }
private:
    int rows, cols;
    bool dfs(vector<vector<char>>& board, string word, int i, int j, int k) {

        if(i >= rows || i < 0 || j >= cols || j < 0 || board[i][j] != word[k]) return false;//如果达到终点，
```

或者不相等退出，这是出口，**空串肯定不会和当前字母相等的。**

```
        if(k == word.size() - 1) return true;//如果刚好相等，返回 true。

        board[i][j] = '\0';//遍历完了，不能走回头路

        bool res = dfs(board, word, i + 1, j, k + 1) || dfs(board, word, i - 1, j, k + 1) ||
                        dfs(board, word, i, j + 1, k + 1) || dfs(board, word, i , j - 1, k + 1);

        board[i][j] = word[k];//如果行不通，要复原这个，这就是回溯啊，留给其他人做

        return res;//是往后的

    }
};
```

作者：jyd
链接：
https://leetcode-cn.com/problems/ju-zhen-zhong-de-lu-jing-lcof/solution/mian-shi-ti-12-ju-zhen-zhong-de-lu-jing-shen-du-yo/
来源：力扣（LeetCode）

本问题是典型的矩阵搜索问题，可使用 深度优先搜索（DFS）+ 剪枝 解决。

深度优先搜索： 可以理解为暴力法遍历矩阵中所有字符串可能性。DFS 通过递归，先朝一个方向搜到底，再回溯至上个节点，沿另一个方向搜索，以此类推。
剪枝： 在搜索中，遇到 这条路不可能和目标字符串匹配成功 的情况（例如：**此矩阵元素和目标字符不同、此元素已被访问**），则应立即返回，称之为 可行性剪枝 。

DFS 解析：
递归参数： 当前元素在矩阵 board 中的行列索引 i 和 j ，当前目标字符在 word 中的索引 k 。
终止条件：
返回 falsefalse ： (1) 行或列索引越界 或 **(2) 当前矩阵元素与目标字符不同 或 (3) 当前矩阵元素已访问过 （ (3) 可合并至 (2) ） 。**
返回 truetrue ： k = len(word) - 1 ，即字符串 word 已全部匹配。
递推工作：
标记当前矩阵元素： 将 board[i][j] 修改为 空字符 '' ，代表此元素已访问过，防止之后搜索时重复访问。
搜索下一单元格： 朝当前元素的 上、下、左、右 四个方向开启下层递归，使用 或 连接 （代表只需找到一条可行路径就直接返回，不再做后续 DFS ），并记录结果至 res 。
还原当前矩阵元素： 将 board[i][j] 元素还原至初始值，即 word[k] 。
返回值： 返回布尔量 res ，代表是否搜索到目标字符串。
使用**空字符**（Python: '' , Java/C++: **'\0'** ）做标记是为了防止**标记字符与矩阵原有字符重复**。当存在重复时，此算法会将矩阵原**有字符认作标记字符**，从而出现错误。

C++ 空字符('\0')和空格符(' ')

下忍 2018-08-02 23:53:40 26438 收藏 10
分类专栏： 知识定义 文章标签： 空字符和空格符

从字符串的长度:——>**空字符的长度为 0,空格符的长度为 1**

```
char a[] = "\0";
char b[] = " ";
cout << strlen(a) << endl;     //0
cout << strlen(b) << endl;     //1
```

虽然输出于屏幕是一样的,但是他们还是有区别的:

```
char crr[] = "a b";     //输出是 a b
char brr[] = "a\0b";    //输出是 a      ---------->因为遇到'\0'代表结束
cout << strlen(crr) << endl;//长度是一样的
cout << strlen(brr) << endl;
```

309. 最佳买卖股票时机含冷冻期（动态规划，三种情况，这里包含了冷冻期。冷冻期就是前天你卖出了（那么表示你前天必须持有）的第二天就是冷冻期）前天的状态做变化，今天就处于冷冻期，冷冻的只是一天，基于昨天的状态，做今天的操作，得到的状态就是今天的

难度中等 722

给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（**多次买卖一支股票**）：

- 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
- **卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。**

示例:

输入：[1,2,3,0,2]**输出：**3 **解释：**对应的交易状态为：[买入，卖出，冷冻期，买入，卖出]

方法一：动态规划

思路与算法

我们用 f[i]f[i] 表示第 ii 天结束之后的「累计最大收益」。根据题目描述，由于我们最多只能同时买入（持有）一支股票，并且卖出股票后有**冷冻期的限制**，因此我们会有三种不同的状态：

我们目前**持有一支股票**，对应的「累计最大收益」记为 f[i][0]；
我们目前不持有任何股票，并且**处于冷冻期中**，对应的「累计最大收益」记为 f[i][1]；
我们目前不持有任何股票，并且**不处于冷冻期中**，对应的「累计最大收益」记为 f[i][2]。
这里的「处于冷冻期」指的是在第 i 天结束之后的状态。也就是说：如果第 i 天结束之后处于冷冻期，那么第 i+1 天无法买入股票。
如何进行状态转移呢？在第 ii 天时，我们可以在不违反规则的前提下进行「买入」或者「卖出」操作，此时第 ii 天的状态会从第 i-1i−1 天的状态转移而来；我们也可以不进行任何操作，此时第 ii 天的状态就等同于第 i-1i−1 天的状态。那么我们分别对这三种状态进行分析：
f[i][0]=max(f[i−1][0],f[i−1][2]−prices[i])//持有股票=和之前一样持有股票，或者不处于冷冻期+当前买股票
对于 f[i][1]f[i][1]，我们在第 ii 天结束之后处于冷冻期的原因是在当天卖出了股票，那么说明在第 i-1 天时我们必须持有一支股票，对应的状态为 f[i-1][0]f[i−1][0] 加上卖出股票的正收益 {\it prices}[i]prices[i]。因此状态转移方程为：
f[i][1]=f[i−1][0]+prices[i]//这天处于冷冻期的，等于之前有，并且当天卖出股票了

对于 f[i][2]f[i][2]，我们在第 ii 天结束之后不持有任何股票并且不处于冷冻期，说明当天没有进行任何操作，即第 i-1i−1 天时不持有任何股票：如果处于冷冻期，对应的状态为 f[i-1][1]f[i−1][1]；如果不处于冷冻期，对应的状态为 f[i-1][2]f[i−1][2]。因此状态转移方程为：

f[i][2]=max(f[i−1][1],f[i−1][2])//不处于冷冻期的，等于之前不处于，或者之前是冷冻期

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.empty()) {
            return 0;
        }

        int n = prices.size();
        // f[i][0]: 手上持有股票的最大收益
        // f[i][1]: 手上不持有股票，并且处于冷冻期中的累计最大收益
        // f[i][2]: 手上不持有股票，并且不在冷冻期中的累计最大收益
        vector<vector<int>> f(n, vector<int>(3));//三种情况
```

```
            f[0][0] = -prices[0];
            for (int i = 1; i < n; ++i) {
                f[i][0] = max(f[i - 1][0], f[i - 1][2] - prices[i]);
                f[i][1] = f[i - 1][0] + prices[i];//冷冻只能是前天有，做了一个卖的动作
                f[i][2] = max(f[i - 1][1], f[i - 1][2]);//昨天冻，今天不冻
            }
            return max(f[n - 1][1], f[n - 1][2]);//只能会在不持有这里找
        }
};
```

145. 二叉树的后序遍历（看后面的迭代版本）

难度中等 551

给定一个二叉树，返回它的 *后序* 遍历。

示例:

输入： `[1,null,2,3]`

```
   1
    \
     2
    /
   3
```

输出： `[3,2,1]`

进阶: 递归算法很简单，你可以通过迭代算法完成吗？

递归简单

```
class Solution {
public:
```

```
    void postorder(TreeNode *root, vector<int> &res) {
        if (root == nullptr) {
            return;
        }
        postorder(root->left, res);
        postorder(root->right, res);
        res.push_back(root->val);
    }

    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> res;
        postorder(root, res);
        return res;
    }
};
```

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/er-cha-shu-de-hou-xu-bian-li-by-leetcode-solution/
来源：力扣（LeetCode）

方法二：迭代（因为现在是左右中，其实你可以先中右左遍历，之后翻转）

思路与算法

我们也可以用迭代的方式实现方法一的递归函数，两种方式是等价的，区别在于递归的时候**隐式地维护了一个栈**，而我们在迭代的时候需要显式地将这个栈模拟出来，其余的实现与细节都相同，具体可以参考下面的代码。

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        vector<int> res;
        if (root == nullptr) {
            return res;
        }

        stack<TreeNode *> stk;//一个栈
        TreeNode *prev = nullptr;
        while (root != nullptr || !stk.empty()) {//如果节点不为空，或者栈不为空
            while (root != nullptr) {
```

```
                stk.emplace(root);//左节点不为空，将节点放进去，变成下一个
                root = root->left;
            }
            root = stk.top();
            stk.pop();
            if (root->right == nullptr || root->right == prev) {
                res.emplace_back(root->val);//这里产生值
                prev = root;
                root = nullptr;
            } else {
                stk.emplace(root);
                root = root->right;
            }
        }
        return res;
    }
};
```

易理解迭代版本

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode *root) {
        if (!root) return {};
        vector<int> vec;
        stack<TreeNode *> stk;
        TreeNode *prev = nullptr;
        auto node = root;
        while (!stk.empty() || node) {
            // 1.遍历到最左子节点
            while (node) {
                stk.emplace(node);
                node = node->left;
            }
            node = stk.top(); stk.pop();
            // 2.遍历最左子节点的右子树(右子树存在  &&  未访问过)
            if (node->right && node->right != prev) {
                //  重复压栈以记录当前路径分叉节点
                stk.emplace(node);
                node = node->right;
            } else {
                //  后序：填充 vec 在 node->left 和 node->right 后面
                //  注意：此时 node 的左右子树应均已完成访问
                vec.emplace_back(node->val);
```

```
                // 避免重复访问右子树[记录当前节点便于下一步对比]
                prev = node;
                // 避免重复访问左子树[设空节点]
                node = nullptr;
            }
        }
        return vec;
    }
};
```

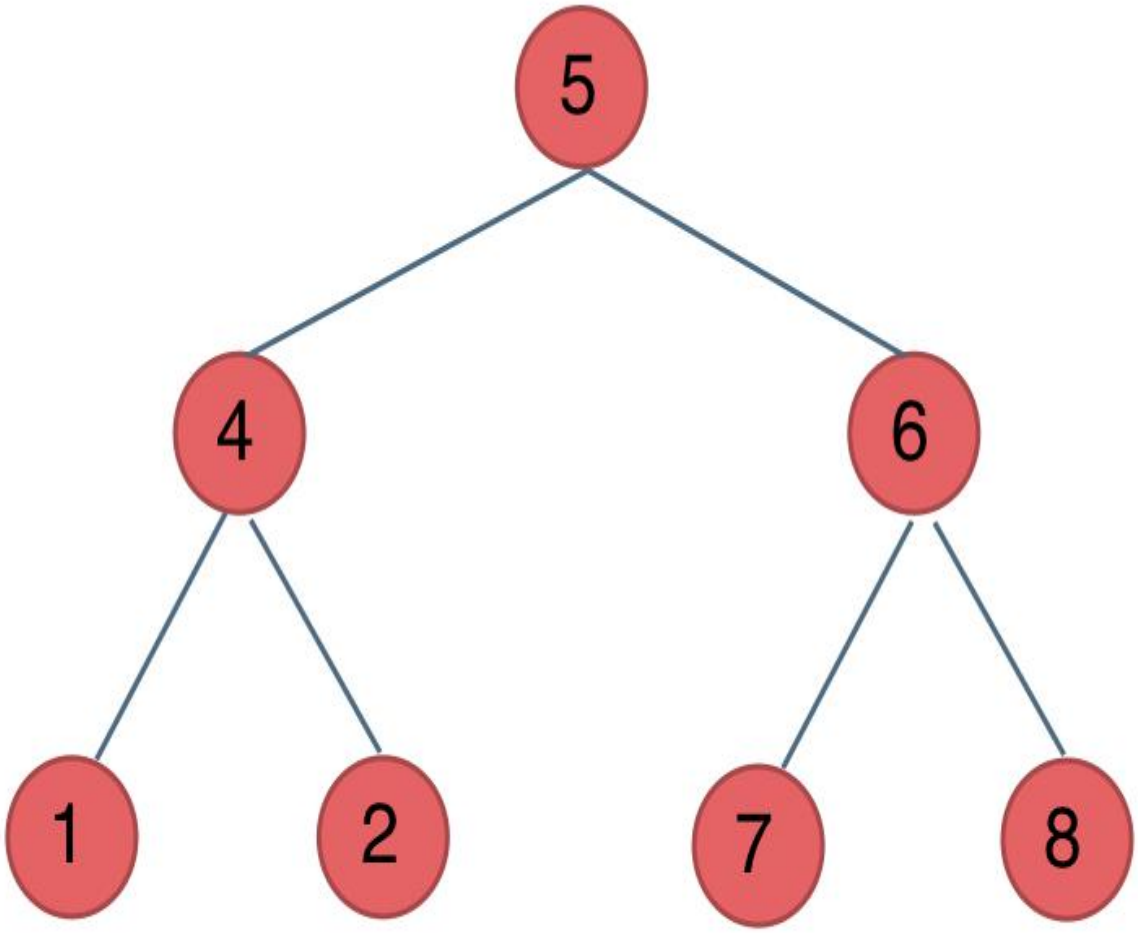
二叉树总结三种递归+迭代遍历方式

这篇文章，彻底讲清楚应该如何写递归，并给出了前中后序三种不同的迭代法，然后分析迭代法的代码风格为什么没有统一，最后给出统一的前中后序迭代法的代码，帮大家彻底吃透二叉树的深度优先遍历

递归法。（中序遍历的二叉树，不能，还原回去，因为不知道 root，是哪个，前和后，就是在前面或者后面）

二叉树深度优先遍历
前序遍历： 0144.二叉树的前序遍历
后序遍历： 0145.二叉树的后序遍历
中序遍历： 0094.二叉树的中序遍历
二叉树广度优先遍历
层序遍历：0102.二叉树的层序遍历
这几道题目建议大家都做一下，本题解先只写二叉树深度优先遍历，二叉树广度优先遍历请看题解二叉树：层序遍历登场！

先帮大家明确一下二叉树的遍历规则：



以上述中，前中后序遍历顺序如下：

前序遍历（中左右）：5 4 1 2 6 7 8
中序遍历（左中右）：1 4 2 5 7 6 8
后序遍历（左右中）：1 2 4 7 8 6 5

作者：carlsun-2
链接：https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/bang-ni-dui-er-cha-shu-bu-zai-mi-mang-che-di-chi-t/
来源：力扣（LeetCode）

这里帮助大家确定下来递归算法的三个要素。每次写递归，都按照这三要素来写，可以保证大家写出正确的递归算法！

确定递归函数的参数和返回值：

确定哪些参数是递归的过程中需要处理的，那么就在递归函数里加上这个参数， 并且还要明确每次递归的返回值是什么进而确定递归函数的返回类型。

确定终止条件：

写完了递归算法, 运行的时候，经常会遇到栈溢出的错误，就是没写终止条件或者终止条件写的不对，操作系统也是用一个栈的结构来保存每一层递归的信息，如果递归没有终止，操作系统的内存栈必然就会溢出。

确定单层递归的逻辑：

确定每一层递归需要处理的信息。在这里也就会重复调用自己来实现递归的过程。

好了，我们确认了递归的三要素，接下来就来练练手

作者：carlsun-2
链接：https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/bang-ni-dui-er-cha-shu-bu-zai-mi-mang-che-di-chi-t/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。以下以前序遍历为例：

确定递归函数的参数和返回值：因为要打印出前序遍历节点的数值，所以参数里需要传入 vector 在放节点的数值，除了这一点就不需要在处理什么数据了也不需要有返回值，所以递归函数返回类型就是 void，代码如下：

```
void traversal(TreeNode* cur, vector<int>& vec)
```

确定终止条件：在递归的过程中，如何算是递归结束了呢，当然是**当前遍历的节点是空了**，那么本层递归就要要结束了，所以如果当前遍历的这个节点是空，就直接 return，代码如下：

```
if (cur == NULL) return;
```

确定单层递归的逻辑：前序遍历是中左右的循序，所以在单层递归的逻辑，是要先取中节点的数值，代码如下：

```
vec.push_back(cur->val);      // 中
traversal(cur->left, vec);    // 左
traversal(cur->right, vec); // 右
```

单层递归的逻辑就是按照中左右的顺序来处理的，这样二叉树的前序遍历，基本就写完了，

在看一下完整代码：

前序遍历：

```
class Solution {
public:
    void traversal(TreeNode* cur, vector<int>& vec) {
        if (cur == NULL) return;
        vec.push_back(cur->val);       // 中
        traversal(cur->left, vec);    // 左
        traversal(cur->right, vec); // 右
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        traversal(root, result);
        return result;
    }
};
```

那么前序遍历写出来之后，中序和后序遍历就不难理解了，代码如下：

中序遍历：

```
    void traversal(TreeNode* cur, vector<int>& vec) {
        if (cur == NULL) return;
        traversal(cur->left, vec);    // 左
        vec.push_back(cur->val);       // 中
        traversal(cur->right, vec); // 右
    }
```

后序遍历：

```
    void traversal(TreeNode* cur, vector<int>& vec) {
        if (cur == NULL) return;
        traversal(cur->left, vec);    // 左
        traversal(cur->right, vec); // 右
        vec.push_back(cur->val);       // 中
    }
```

迭代法

为什么可以用迭代法（非递归的方式）来实现二叉树的前后中序遍历呢？

我们在栈与队列：匹配问题都是栈的强项中提到了，递归的实现就是：每一次递归调用都会把函数的局部变量、参数值和返回地址等压入调用栈中，然后递归返回的时候，从栈顶弹出上一次递归的各项参数，所以这就是**递归为什么可以返回上一层位置的原因**。

此时大家应该知道我们用栈也可以是实现二叉树的前后中序遍历了。

前序遍历（迭代法）
我们先看一下前序遍历。

前序遍历是中左右，每次先处理的是**中间节点**，那么先将跟节点放入栈中，然后将右孩子加入栈，再加入左孩子。（先处理中间的）

为什么要先加入 右孩子，再加入左孩子呢？ 因为这样出栈的时候才是**中左右的顺序**。

不难写出如下代码: （注意代码中空节点不入栈）

前序迭代

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        stack<TreeNode*> st;
        vector<int> result;
        if (root == NULL) return result;
        st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();                    // 中
            st.pop();
            result.push_back(node->val);
            if (node->right) st.push(node->right);        // 右（空节点不入栈）
            if (node->left) st.push(node->left);          // 左（空节点不入栈）
        }
        return result;
    }
};
```

中序遍历（迭代法（左找右），先将左到底，左拿出来，左找右，右还有左，放进去，右没有继续拿左）

先左节点访问到终点，到底了，取出来，并且对应访问他的右节点，如果右节点为空，继

续访问他的左节点，不为空，加入进去。
为了解释清楚，我说明一下 刚刚在迭代的过程中，其实我们有两个操作：

处理：将元素放进 result 数组中
访问：遍历节点
分析一下为什么刚刚写的前序遍历的代码，不能和中序遍历通用呢，因为前序遍历的顺序是中左右，先访问的**元素是中间节点，要处理的元素也是中间节点，所以刚刚才能写出相对简洁的代码，因为要访问的元素和要处理的元素顺序是一致的**，都是中间节点。

那么再看看中序遍历，中序遍历是左中右，先访问的是二叉树顶部的节点，然后一层一层向下访问，直到到达树左面的最底部，再开始**处理节点**（也就是在把节点的数值放进 result 数组中），这就造成了处理顺序和访问顺序是不一致的。

那么在使用迭代法写中序遍历，就需要借用**指针的遍历来帮助访问节**点，栈则用来处理节点上的元素。

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        TreeNode* cur = root;//当前指针
        while (cur != NULL || !st.empty()) {
            if (cur != NULL) { // 指针来访问节点，访问到最底层
                st.push(cur); // 将访问的节点放进栈
                cur = cur->left;                    // 左
            } else {
                cur = st.top(); // 从栈里弹出的数据，就是要处理的数据（放进 result 数组里的数据，左边弹出之后）
                st.pop();
                result.push_back(cur->val);         // 中
                cur = cur->right;                   // 右
            }
        }
        return result;
    }
};
```

作者：carlsun-2
链接：https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/bang-ni-dui-er-cha-shu-bu-zai-mi-mang-che-di-chi-t/
来源：力扣（LeetCode）

后序遍历（迭代法）

再来看后序遍历，先序遍历是中左右，后续遍历是左右中，那么我们只需要调整一下先序遍历的代码顺序，就变成**中右左的遍历顺序**，然后在反转 result 数组，输出的结果顺序就是左右中了，如下图：

所以后序遍历只需要前序遍历的代码稍作修改就可以了，代码如下

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        stack<TreeNode*> st;
        vector<int> result;
        if (root == NULL) return result;
        st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();
            st.pop();
            result.push_back(node->val);
            if (node->left) st.push(node->left); // 相对于前序遍历，这更改一下入栈顺序（空节点不入栈）
            if (node->right) st.push(node->right); // 空节点不入栈
        }
        reverse(result.begin(), result.end()); // 将结果反转之后就是左右中的顺序了
        return result;
    }
};
```

作者：carlsun-2
链接：https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/bang-ni-dui-er-cha-shu-bu-zai-mi-mang-che-di-chi-t/
来源：力扣（LeetCode）

二叉树前中后迭代方式统一写法（产生值，就是和我们遍历一样，如果他的前

面 leftright 都搞定了，栈就压完了，就一个个出来呗）

此时我们在二叉树：一入递归深似海，从此 offer 是路人中用递归的方式，实现了二叉树前中后序的遍历。

在二叉树：听说递归能做的，栈也能做！中用栈实现了二叉树前后中序的迭代遍历（非递归）。

之后我们发现迭代法实现的先中后序，其实风格也不是那么统一，除了先序和后序，有关联，中序完全就是另一个风格了，一会用栈遍历，一会又用指针来遍历。

实践过的同学，也会发现使用迭代法实现先中后序遍历，很难写出统一的代码，不像是递归法，实现了其中的一种遍历方式，其他两种只要稍稍改一下节点顺序就可以了。

其实针对三种遍历方式，使用迭代法是可以写出统一风格的代码！

重头戏来了，接下来介绍一下统一写法。

我们以中序遍历为例，在二叉树：听说递归能做的，栈也能做！中提到说使用栈的话，无法同时解决访**问节点（遍历节点）和处理节点（将元素放进结果集）不一致的情**况。

那我们就将**访问的节点放入栈中，把要处理的节点也放入栈中但是要做标记**。

如何标记呢，就是要处理的节点放入栈之后，紧接着放入一个**空指针作为标记**。 这种方法也可以叫做标记法。

迭代法中序遍历(左中右，换成右中左)

中序遍历代码如下：（详细注释）

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if (root != NULL) st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();
            if (node != NULL) {//等于空弹出下一个，标记
                st.pop(); // 将该节点弹出，避免重复操作，下面再将右中左节点添加到栈
中
```

```
                if (node->right) st.push(node->right);  // 添加右节点（空节点不入栈）

                st.push(node);                          // 添加中节点
                st.push(NULL); // 中节点访问过，但是还没有处理，加入空节点做为标记。

                if (node->left) st.push(node->left);    // 添加左节点（空节点不入栈）
            } else { // 只有遇到空节点的时候，才将下一个节点放进结果集
                st.pop();           // 将空节点弹出
                node = st.top();    // 重新取出栈中元素
                st.pop();
                result.push_back(node->val); // 加入到结果集
            }
        }
        return result;
    }
};
```

作者：carlsun-2
链接：https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/bang-ni-dui-er-cha-shu-bu-zai-mi-mang-che-di-chi-t/
来源：力扣（LeetCode）

迭代法前序遍历（中左右，变成右左中）

迭代法前序遍历代码如下： (注意此时我们和中序遍历相比仅仅改变**了两行代码的顺序**)

```
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if (root != NULL) st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();
            if (node != NULL) {
                st.pop();
                if (node->right) st.push(node->right);  // 右
                if (node->left) st.push(node->left);    // 左
                st.push(node);                          // 中
                st.push(NULL);
            } else {
```

```
                st.pop();
                node = st.top();
                st.pop();
                result.push_back(node->val);
            }
        }
        return result;
    }
};
```

作者：carlsun-2
链接：https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/bang-ni-dui-er-cha-shu-bu-zai-mi-mang-che-di-chi-t/
来源：力扣（LeetCode）

迭代法后序遍历（左右中 中右左）

后续遍历代码如下： (注意此时我们和中序遍历相比仅仅改变了两行代码的顺序)

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if (root != NULL) st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();
            if (node != NULL) {
                st.pop();
                st.push(node);                          // 中
                st.push(NULL);

                if (node->right) st.push(node->right);  // 右
                if (node->left) st.push(node->left);    // 左

            } else {
                st.pop();
                node = st.top();
                st.pop();
                result.push_back(node->val);
            }
```

```
            }
            return result;
        }
};
```

总结

此时我们写出了统一风格的迭代法，不用在纠结于前序写出来了，中序写不出来的情况了。

但是统一风格的迭代法并不好理解，而且想在面试直接写出来还有难度的。

所以大家根据自己的个人喜好，对于二叉树的前中后序遍历，选择一种自己容易理解

1091. 二进制矩阵中的最短路径（8 个选择）

难度中等 95

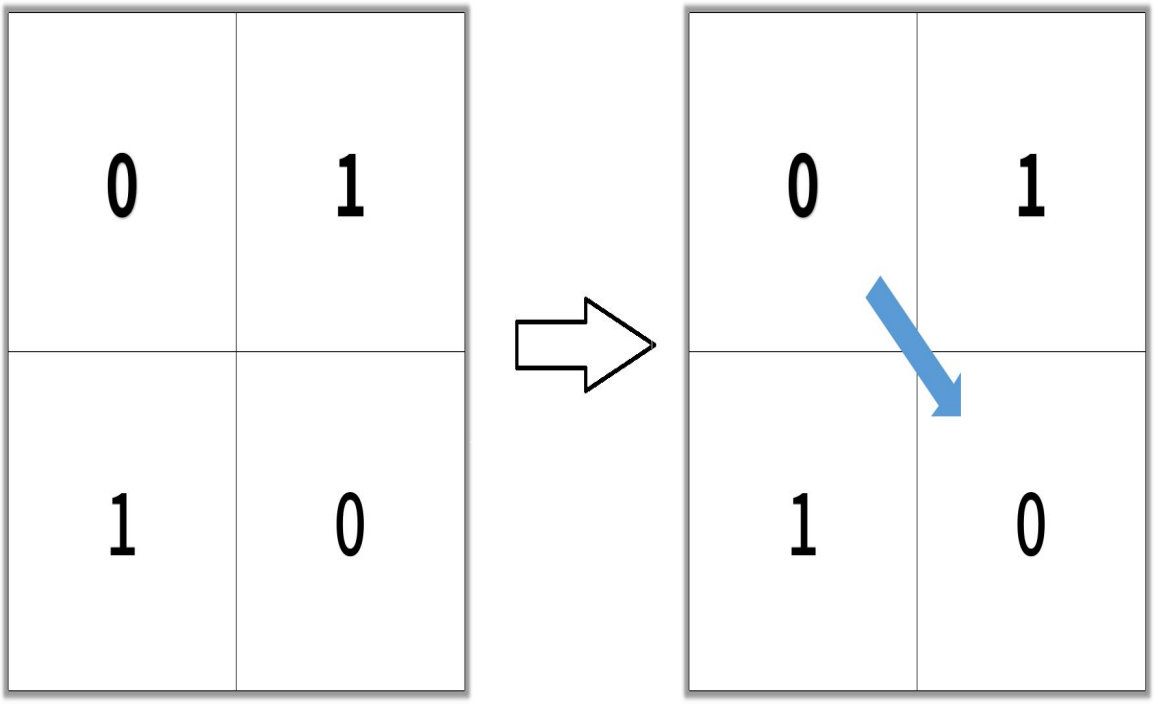
给你一个 `n x n` 的二进制矩阵 *grid* 中，返回矩阵中最短 **畅通路径** 的长度。如果不存在这样的路径，返回 `-1` 。

二进制矩阵中的 畅通路径 是一条从 **左上角** 单元格（即，`(0,0)`）到 右下角 单元格（即，`(n - 1, n - 1)`）的路径，该路径同时满足下述要求：

- 路径途**经的所有单元格都的值都是 `0`** 。
- 路径中所有相邻的单元格应当在 **8 个方向之一** 上连通（即，相邻两单元之间彼此不同且共享一条边或者一个角）。

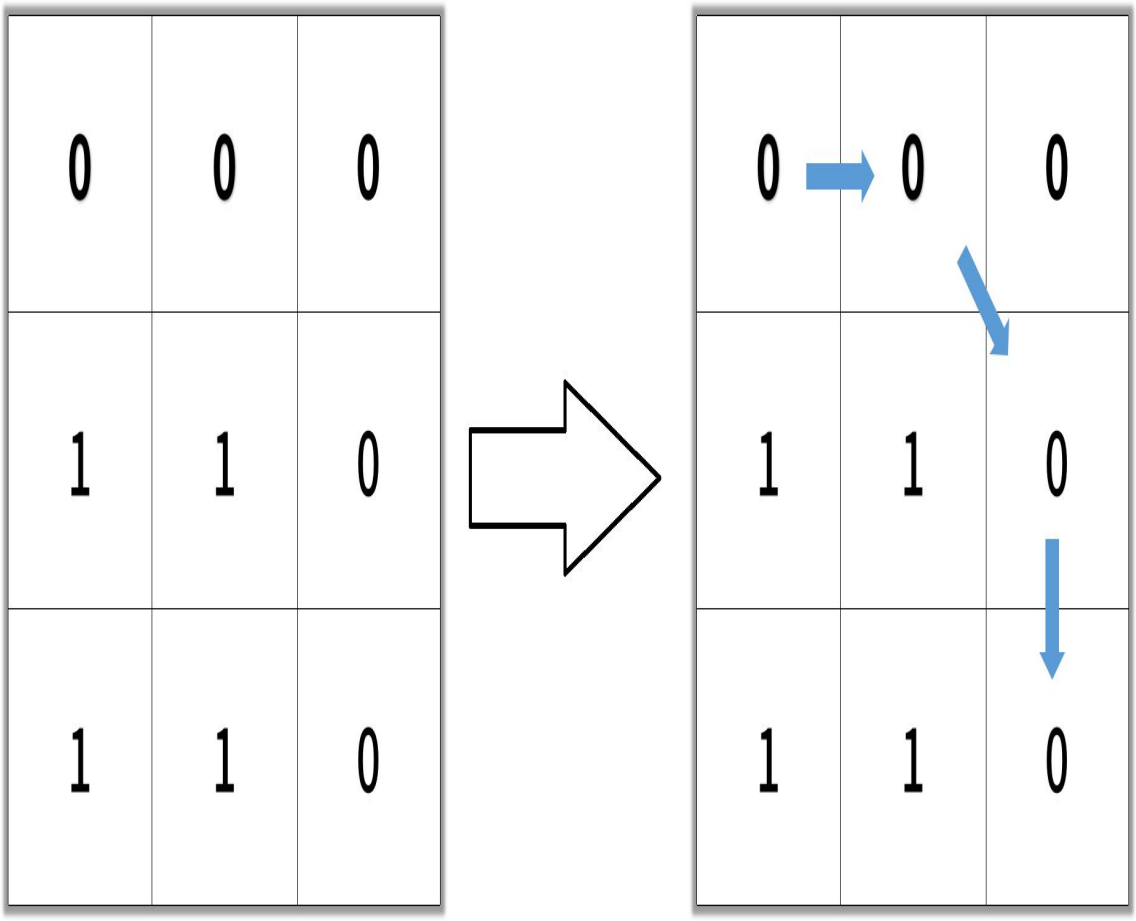
畅通路径的长度 是该路径途经的单元格总数。

示例 1：



输入：grid = [[0,1],[1,0]]输出：2

示例 2：



输入：grid = [[0,0,0],[1,1,0],[1,1,0]]输出：4

示例 3：

输入：grid = [[1,0,0],[1,1,0],[1,1,0]]输出：-1

提示：

- *n == grid.length*
- *n == grid[i].length*

- 1 <= n <= 100

- grid[i][j] 为 0 或 1

```
class Solution {
public:
    vector<vector<int>>dir={{0,1},{0,-1},{1,0},{-1,0},{1,1},{1,-1},{-1,1},{-1,-1}};//8 个数组
    int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
        if(grid[0][0]==1)return -1;
        int n=grid.size();
        queue<pair<int,int>>q;
        q.push(make_pair(0,0));
        int length=1;//包括自身
        grid[0][0]=2;    //将访问过的点标记为 2
        while(!q.empty()){
            int l=q.size();          //遍历当前队列所有的元素
            for(int i=0;i<l;i++){
                int x=q.front().first;
                int y=q.front().second;
                q.pop();
                if(x==n-1&&y==n-1)return length;//等于左下角，返回路径
                for(int j=0;j<8;j++){
                    int x1=x+dir[j][0];
                    int y1=y+dir[j][1];
                    if(x1<0||y1<0||x1>=n||y1>=n||grid[x1][y1])continue;    //越界或者不
满足访问条件跳过==1 或者==2 访问过了
                    grid[x1][y1]=2;//就是走过了
                    q.push(make_pair(x1,y1));
                }
            }
            length++;
        }
        return -1;

    }
};
```

作者：mei-you-ni-de-liu-yue-tian
链接：https://leetcode-cn.com/problems/shortest-path-in-binary-matrix/solution/cbiao-zhun-de-bfs-by-mei-you-ni-de-liu-yue-tian/
来源：力扣（LeetCode）

397. 整数替换（典型的 BFS, 几个选择）

难度中等 89

给定一个正整数 n，你可以做如下操作：

1. 如果 n 是偶数，则用 n / 2 替换 n 。
2. 如果 n 是奇数，则可以用 n + 1 或 n - 1 替换 n 。

n 变为 1 所需的最小替换次数是多少？

示例 1：

输入： n = 8 **输出：** 3 **解释：** 8 -> 4 -> 2 -> 1

示例 2：

输入： n = 7 **输出：** 4 **解释：** 7 -> 8 -> 4 -> 2 -> 1

或 7 -> 6 -> 3 -> 2 -> 1

示例 3：

输入： n = 4 **输出：** 2

```
class Solution {
public:
    int integerReplacement(int n) {
        int step = 0;
        unordered_set<long> mem;//走过的路程
        queue<long> que;
        que.push(n);
        while(!que.empty()) {
            int size = que.size();
```

```
            for (int i = 0; i < size; i++) {
                long cur = que.front();
                que.pop();
                if (cur == 1)
                    return step;//等于 0 步长
                if (mem.count(cur))
                    continue;

                if (cur % 2) {
                    que.push(cur + 1);//奇数的选择
                    que.push(cur - 1);
                } else {
                    que.push(cur / 2);//偶数的选择
                }
            }
            step++;
        }

        return step;
    }
};
```

作者：michael-201
链接：https://leetcode-cn.com/problems/integer-replacement/solution/bfs-by-michael-201/
来源：力扣（LeetCode）

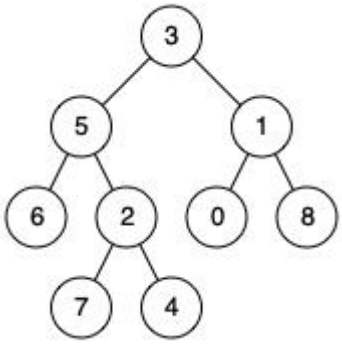
236. 二叉树的最近公共祖先（这是二叉树，不是二叉搜索树，两个指定节点，不是多个，和多个类似，只要 root 在里面或者等于 p q 就返回 root）

难度中等 1030

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

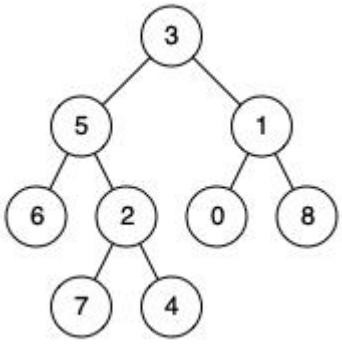
百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

示例 1：



输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1 **输出：**3 **解释：**节点 5 和节点 1 的最近公共祖先是节点 3 。

示例 2：



输入：root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4 **输出：**5 **解释：**节点 5 和节点 4 的最近公共祖先是节点 5 。因为根据定义最近公共祖先节点可以为节点本身。

解题思路：
两个节点 p,q 分为两种情况：

p 和 q 在相同子树中
p 和 q 在不同子树中
从根节点遍历，递归向左右子树查询节点信息
递归终止条件：如果当前节点为空或等于 p 或 q，则返回当前节点

递归遍历左右子树，如果左右子树查到节点都不为空，则表明 p 和 q 分别在左右子树中，因此，当前节点即为最近公共祖先；
如果左右子树其中一个不为空，则返回非空节点。
代码

C++

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || root == p || root == q) return root;
        TreeNode *left = lowestCommonAncestor(root->left, p, q);
        TreeNode *right = lowestCommonAncestor(root->right, p, q);
        if (left && right) return root;
        return left ? left : right;
    }
};
```

作者：guohaoding
链接：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/236-er-cha-shu-de-zui-jin-gong-gong-zu-xian-jian-j/
来源：力扣（LeetCode）

具体思路：
（1） 如果当前结点 rootroot 等于 NULL，则直接返回 NULL
（2） 如果 rootroot 等于 pp 或者 qq ，那这棵树一定返回 pp 或者 qq
（3）然后递归左右子树，因为是递归，使用函数后可认为左右子树已经算出结果，用 leftleft 和 rightright 表示
（4） 此时若 leftleft 为空，那最终结果只要看 rightright；若 rightright 为空，那最终结果只要看 leftleft
（5） 如果 leftleft 和 rightright 都非空，因为只给了 pp 和 qq 两个结点，都非空，说明一边一个，因此 rootroot 是他们的最近公共祖先
（6） 如果 leftleft 和 rightright 都为空，则返回空（其实已经包含在前面的情况中了）

时间复杂度是 O(n)O(n)：每个结点最多遍历一次或用主定理，空间复杂度是 O(n)O(n)：需要系统栈空间

作者：Wilson79
链接：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/c-jing-dian-di-gui-si-lu-fei-chang-hao-li-jie-shi-/
来源：力扣（LeetCode）

递归（和多个节点一样）

```
class Solution {
public:
```

```
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == NULL)
            return NULL;
        if(root == p || root == q) //多个就换成了，是否在里面
            return root;

        TreeNode* left =    lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        if(left == NULL)
            return right;
        if(right == NULL)
            return left;
        if(left && right) // p 和 q 在两侧
            return root;

        return NULL; //  必须有返回值，两个都是 NULL
    }
};
```

作者：Wilson79
链接：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/c-jing-dian-di-gui-si-lu-fei-chang-hao-li-jie-shi-/
来源：力扣（LeetCode）

139. 单词拆分(背包问题，完全背包)

难度中等 899

给定一个**非空**字符串 s 和一个包含**非空**单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明：

- 拆分时可以重复使用字典中的单词。
- 你可以假设字典中没有重复的单词。

示例 1：

输入： s = "leetcode", wordDict = ["leet", "code"]**输出：** true **解释：** 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2：

输入： s = "applepenapple", wordDict = ["apple", "pen"]**输出：** true **解释：** 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。

注意你可以重复使用字典中的单词。

示例 3：
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]输出: false
背包问题
单词就是物品，字符串 s 就是背包，单词能否组成字符串 s，就是问物品能不能把背包装满。

拆分时可以重复使用字典中的单词，说明就是一个完全背包！

动规五部曲分析如下：

确定 dp 数组以及下标的含义
dp[i]：字符串长度为 i 的话，dp[i]为 true，表示可以拆分为一个或多个在字典中出现的单词。

确定递推公式
如果确定 dp[j] 是 true，且 [j, i] 这个区间的子串出现在字典里，那么 dp[i]一定是 true。（j < i ）。

所以递推公式是 if([j, i] 这个区间的子串出现在字典里 && dp[j]是 true) 那么 dp[i] = true。

dp[0]表示如果字符串为空的话，说明出现在字典里。

下标非 0 的 dp[i]初始化为 false，只要没有被覆盖说明都是不可拆分为一个或多个在字典中出现的单词。

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size() + 1, false);
        dp[0] = true;
        for (int i = 1; i <= s.size(); i++) {    // 遍历背包 //表示个数
            for (int j = 0; j < i; j++) {          // 遍历物品
                string word = s.substr(j, i - j); //substr(起始位置，截取的个数)j i-1 索引
```

```
                    if (wordSet.find(word) != wordSet.end() && dp[j]) {//在里面，并且前面是true
                        dp[i] = true;//可以 break 了。
                    }
                }
            }
            return dp[s.size()];
        }
};
```

时间复杂度：O(n^3)，因为 substr 返回子串的副本是 O(n)的复杂度（这里的 n 是 substring 的长度）
空间复杂度：O(n)

作者：carlsun-2
链接：https://leetcode-cn.com/problems/word-break/solution/139-dan-ci-chai-fen-hui-su-fa-wan-quan-b-0zwf/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

方法一：动态规划
思路和算法

我们定义 $\textit{dp}[i]$dp[i] 表示字符串 ss 前 ii 个字符组成的字符串 s[0..i-1]s[0..i−1] 是否能被空格拆分成若干个字典中出现的单词。从前往后计算考虑转移方程，每次转移的时候我们需要枚举包含位置 i-1i−1 的最后一个单词，看它是否出现在字典中以及除去这部分的字符串是否合法即可。公式化来说，我们需要枚举 s[0..i-1]s[0..i−1] 中的分割点 jj ，看 s[

是否合法即可，因此我们可以得出如下转移方程：

$$\textit{dp}[i]=\textit{dp}[j]\ \&\&\ \textit{check}(s[j..i-1])$$
dp[i]=dp[j] && check(s[j..i−1])

其中 $\textit{check}(s[j..i-1])$check(s[j..i−1]) 表示子串 s[j..i-1]s[j..i−1] 是否出现在字典中。

对于检查一个字符串是否出现在给定的字符串列表里一般可以考虑哈希表来快速判断，同时也可以做一些简单的剪枝，枚举分割点的时候倒着枚举，如果分割点 jj 到 ii 的长度已经大于字典列表里最长的单词的长度，那么就结束枚举，但是需要注意的是下面的代码给出的是不带剪枝的写法。

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/word-break/solution/dan-ci-chai-fen-by-leetcode-solution/
来源：力扣（LeetCode）

代码（dp[i]表示 s[0..i-1] 是否能被空格拆分成若干个字典中出现的单词,dp[size]是最后的结果，从0，。。size 所以 vector 是 dp（size+1）初始化前面存在和后面是单词就行，dp[0]表示空集，true,string word = s.substr(j, i - j); //substr(起始位置，截取的个数),是 i-j，）

```
   unordered_set<string> wordSet(wordDict.begin(), wordDict.end());
用这个赋值 set 呗
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        auto wordDictSet = unordered_set <string> ();
        for (auto word: wordDict) {
            wordDictSet.insert(word);
        }
        auto dp = vector <bool> (s.size() + 1);//默认是 false
        dp[0] = true;
        for (int i = 1; i <= s.size(); ++i) {
            for (int j = 0; j < i; ++j) {//单词是 0...j j...i-1 看后面的单词  dp[i]表示 s[0..i-1]
                if (dp[j] && wordDictSet.find(s.substr(j, i - j)) != wordDictSet.end()) {//在里面
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[s.size()];
    }
};
```

作者：LeetCode-Solution
链接：
https://leetcode-cn.com/problems/word-break/solution/dan-ci-chai-fen-by-leetcode-solution/
来源：力扣（LeetCode）

143. 重排链表（反转后面的，里面有个标准的快慢指针模板，重要，交叉合并链表，记得 mid 尾部给个 null）

难度中等 544

给定一个单链表 L：L0→L1→…→Ln-1→Ln ，
将其重新排列后变为： L0→Ln→L1→Ln-1→L2→Ln-2→…

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1:

给定链表 1->2->3->4，重新排列为 1->4->2->3.

示例 2:

给定链表 1->2->3->4->5，重新排列为 1->5->2->4->3.

方法二：寻找链表中点 + 链表逆序 + 合并链表（合并是交叉合并呗，奇数取后面的，后面就是一半一半了）

注意到目标链表即为将原链表的左半端和反转后的右半端交叉合并后的结果。

这样我们的任务即可划分为三步：

找到原链表的中点（参考「876. 链表的中间结点」）。

我们可以使用快慢指针来 O(N)O(N) 地找到链表的中间节点。

将**原链表的右半端反转（参考「206. 反转链表」）**。

我们可以使用迭代法实现链表的反转。

将原链表的两端合并。

因为两链表长度相差不超过 **1**，因此直接合并即可

作者：LeetCode-Solution

链接：
https://leetcode-cn.com/problems/reorder-list/solution/zhong-pai-lian-biao-by-leetcode-solution/

来源：力扣（LeetCode）


```
class Solution {
public:
    void reorderList(ListNode* head) {
        if (head == nullptr) {
            return;
        }
        ListNode* mid = middleNode(head);
        ListNode* l1 = head;
        ListNode* l2 = mid->next;//后面是一半的，偶数是一半，奇数是后面的
        mid->next = nullptr;//前端自己加一个 null 尾巴
        l2 = reverseList(l2);
        mergeList(l1, l2);
    }
```

```
    ListNode* middleNode(ListNode* head) {//标准的快慢指针，返回 slow 重要
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast->next != nullptr && fast->next->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }

    ListNode* reverseList(ListNode* head) {//标准反转
        ListNode* prev = nullptr;
        ListNode* curr = head;
        while (curr != nullptr) {
            ListNode* nextTemp = curr->next;
            curr->next = prev;
            prev = curr;
            curr = nextTemp;
        }
        return prev;//返回 prev 头节点，等于最后的 current
    }

    void mergeList(ListNode* l1, ListNode* l2) {//交叉合并
        ListNode* l1_tmp;
        ListNode* l2_tmp;
        while (l1 != nullptr && l2 != nullptr) {
            l1_tmp = l1->next;//每次前链表拿出下一个
            l2_tmp = l2->next;//每次拿出后链表下一个

            l1->next = l2;//前链表的下一个是后链表
            l1 = l1_tmp;//更新前链表

            l2->next = l1;//后的下一个是前链表
            l2 = l2_tmp;//更新后链表
        }
    }
};
```

作者：LeetCode-Solution

链接：
https://leetcode-cn.com/problems/reorder-list/solution/zhong-pai-lian-biao-by-leetcode-solution/

来源：力扣（LeetCode）

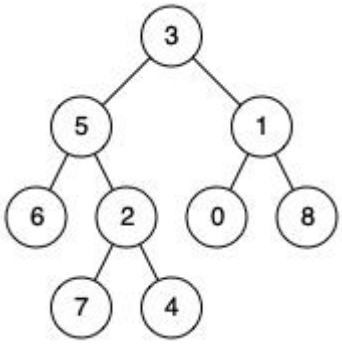
1676. 二叉树的最近公共祖先 IV（这不是二叉搜索树哦，后续遍历+递归，如果root 在当前 nodes 里面/空集，返回自身，因为一直向上递归，得到的是最近的 node，不是的话，看他的左右，都有，就是 root，或者不为空的就是，两者都为空，返回空，一直把左右往上走，因为最后的公共节点肯定是最晚出现的节点之后的）

难度中等 4

给定一棵二叉树的根节点 *root* 和 *TreeNode* 类对象的数组（列表） *nodes*，返回 *nodes* 中所有节点的最近公共祖先（LCA）。数组（列表）中所有节点都存在于该二叉树中，且二叉树中所有节点的值都是互不相同的。

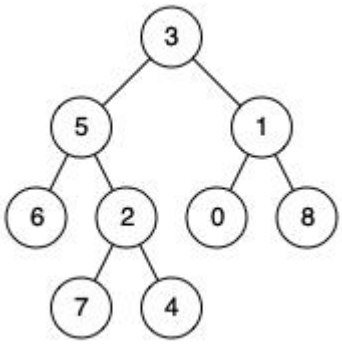
我们扩展二叉树的最近公共祖先节点在维基百科上的定义：“对于任意合理的 *i* 值， *n* 个节点 *p1* 、 *p2*、…、 *pn* 在二叉树 *T* 中的最近公共祖先节点是**后代**中包含所有节点 *pi* 的最深节点（我们允许一个节点是其自身的后代）”。一个节点 *x* 的后代节点是节点 *x* 到某一叶节点间的路径中的节点 *y*。

示例 1:



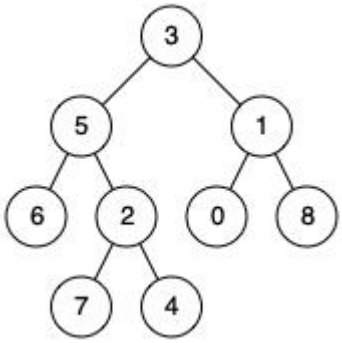
输入: root = [3,5,1,6,2,0,8,null,null,7,4], nodes = [4,7]**输出:** 2 **解释:** 节点 4 和 7 的最近公共祖先是 2。

示例 2:



输入: root = [3,5,1,6,2,0,8,null,null,7,4], nodes = [1]**输出:** 1 **解释:** 单个节点的最近公共祖先是该节点本身。

示例 3:



输入： root = [3,5,1,6,2,0,8,null,null,7,4], nodes = [7,6,2,4]**输出：** 5 **解释：** 节点 7、6、2 和 4 的最近公共祖先节点是 5

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution
{
public:
```

```
    TreeNode* lowestCommonAncestor(TreeNode* root, vector<TreeNode*> &nodes)
    {   // dfs_LRN 后序遍历求 LCA
        if (root==NULL)//如果没有，就是自身了
            return root;//返回 NULL
        if (count(nodes.begin(), nodes.end() ,root))//存在，压栈一样，最先出来，被后面覆盖。
            return root;
        TreeNode* L = lowestCommonAncestor(root->left, nodes);
        TreeNode* R = lowestCommonAncestor(root->right, nodes);
        if (L && R)
            return root;//两个都有
        else if(L && R==NULL)
            return L;
        else if (L == NULL && R)
            return R;
        else
            return NULL;  //不存在，L R 都是 NULL 就返回 NULL,给递归的，都是 NULL
    }
};
```

作者：Hanxin_Hanxin

链接：
https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree-iv/solution/c-python3-hou-xu-bian-li-di-gui-dfs_lrn-eqiss/

来源：力扣（LeetCode）

对于 vector，我们可以很方便地寻找其中符合条件的元素的个数。

要注意它们不是 vector 的类函数，不要用 vector.去调用。

用法一览：

一. count 函数：返回元素值为 target 的元素个数。

```
int num=count(vector1.begin(),vector2.begin(),target);   //注意不是 vector 的类函数哟！！
```

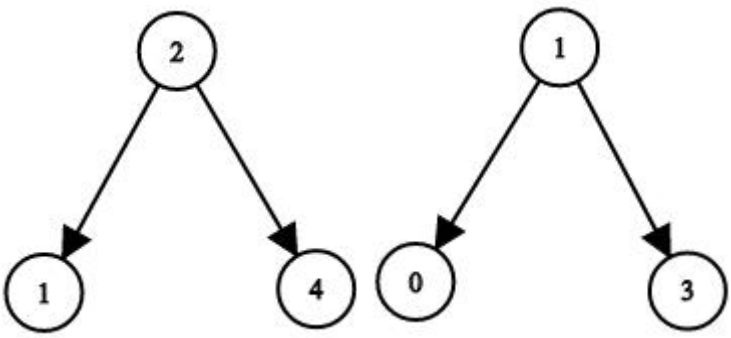
1214. 查找两棵二叉搜索树之和（从小到大的中序遍历，转换成两个数组，用双指针两边移动，从小到大的数组，取一个最小值，一个最大值，不断移动指针呗。）

难度中等 27

给出两棵二叉搜索树，请你从两棵树中各找出一个节点，使得这两个节点的值之和等于目标值 *Target*。

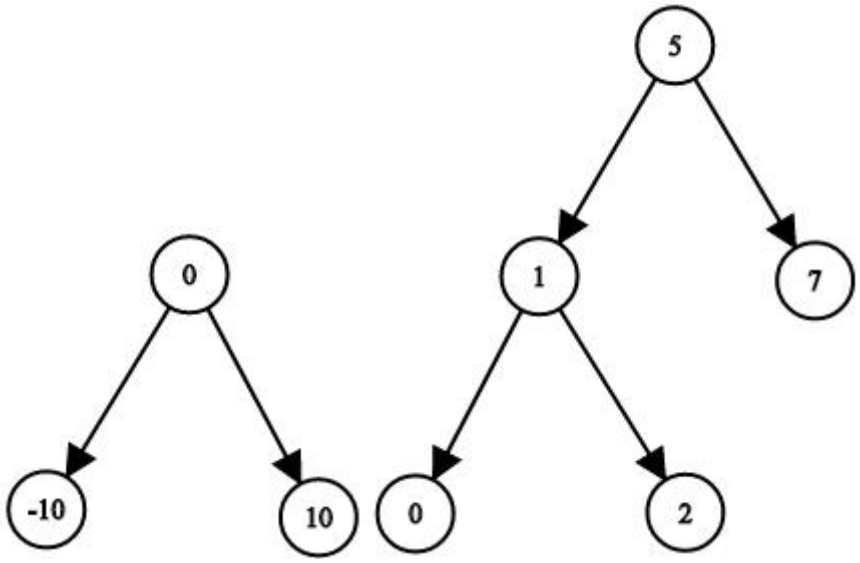
如果可以**找到返回 *True*，否则返回 *False*。**

示例 1：



输入：root1 = [2,1,4], root2 = [1,0,3], target = 5 **输出：**true **解释：**2 加 3 和为 5 。

示例 2：



输入：root1 = [0,-10,10], root2 = [5,1,7,0,2], target = 18 **输出：**false

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
```

```
private:
    void inOrder(TreeNode* root, vector<int> &ans) {
        if (root == NULL) { return ; }
        inOrder(root->left, ans);
        ans.push_back(root->val);
        inOrder(root->right, ans);
    }
public:
    bool twoSumBSTs(TreeNode* root1, TreeNode* root2, int target) {
        vector<int> tree1;
        vector<int> tree2;
        inOrder(root1, tree1);//中序遍历
        inOrder(root2, tree2);
        int l1 = tree1.size(), l2 = tree2.size();
        int i = 0, j = l2 - 1;
        while (i < l1 && j >= 0) {
            int add = tree1[i] + tree2[j];
            if (add > target) {
                j--;
            }
            else if (add < target) {
                i++;
            }
            else {
                return true;
            }
        }
        return false;
    }
};
```

作者：dwqe

链接：
https://leetcode-cn.com/problems/two-sum-bsts/solution/er-cha-shu-bian-li-shuang-zhi-zhen-qiu-jie-liang-s/

来源：力扣（LeetCode）

std::move 原理实现与用法总结

转载

文章目录

右值引用（及其支持的 Move 语意和完美转发）是 C++0x 加入的最重大语言特性之一。从实践角度讲，它能够完美解决 C++中长久以来为人所诟病的临时对象效率问题。从语言本身讲，它健全了 C++中的引用类型在左值右值方面的缺陷。从库设计者的角度讲，它给库设计者又带来了一把利器。从库使用者的角度讲，不动一兵一卒便可以获得“免费的”效率提升。

一、左值与右值

- 左值：指表达式结束后依然存在的**持久对象，可以取地址，**具名变量或对象 。

- 右值：表达式结束后就不再存在的**临时对象，不可以取地址，**没有名字。

```
int a;
int b;

a = 3;
b = 4;
a = b;
b = a;

// 以下写法不合法。
3 = a;
a + b = 4;
```

二、左值引用和右值引用

引用是 C++语法做的优化，**引用的本质还是靠指针来实现的**。引用相当于变量的别名。引用可以改变指针的指向，还可以改变指针所指向的值。

- 左值引用：type &引用名 = 左值表达式；
- 右值引用：type &&引用名 = 右值表达式；

三、std::move 详解

1、std::move 简介

在 C++11 中，标准库在中提供了一个有用的函数 std::move，std::move 并不能移动任何东西，它唯一的功能是**将一个左值引用强制转化为右值引用，继而可以通过右值引用使用该值，以用于移动语义**。从实现上讲，std::move 基本等同于一个类型转换：

`static_cast<T&&>(lvalue);`

```
#include <iostream>
#include <utility>
#include <vector>
#include <string>
int main()
{
    std::string str = "Hello";
    std::vector<std::string> v;
    //调用常规的拷贝构造函数，新建字符数组，拷贝数据
    v.push_back(str);
    std::cout << "After copy, str is \"" << str << "\"\n";
    //调用移动构造函数，掏空 str，掏空后，最好不要使用 str
    v.push_back(std::move(str));
    std::cout << "After move, str is \"" << str << "\"\n";
    std::cout << "The contents of the vector are \"" << v[0]
                                    << "\", \"" << v[1] << "\"\n";
}
```

2、std::move 详解

std::move 的函数原型定义：

```
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
        return static_cast<typename remove_reference<T>::type &&>(t);
}
```

首先，函数参数 T&&是一个指向模板类型参数的右值引用，**通过引用折叠**，此参数可以与任何类型的实参匹配（**可以传递左值或右值**，这是 std::move 主要使用的两种场景)。关于引用折叠如下：

- 所有右值引用折叠到右值引用上仍然是一个右值引用。（A&&&& 变成 A&&） 。

- 所有的其他引用类型之间的折叠都将变成左值引用。（A& & 变成 A&; A& && 变成 A&; A&& & 变成 A&）。

简单来说，右值经过 T&&传递类型保持不变还是右值，而左值经过 T&&变为普通的左值引用。

```
//原始的，最通用的版本
template <typename T> struct remove_reference{
    typedef T type;  //定义 T 的类型别名为 type
};

//部分版本特例化，将用于左值引用和右值引用
template <class T> struct remove_reference<T&> //左值引用
{ typedef T type; }

template <class T> struct remove_reference<T&&> //右值引用
{ typedef T type; }

//举例如下,下列定义的 a、b、c 三个变量都是 int 类型
int i;
remove_refrence<decltype(42)>::type a;            //使用原版本，
remove_refrence<decltype(i)>::type  b;            //左值引用特例版本
remove_refrence<decltype(std::move(i))>::type  b;  //右值引用特例版本
```

std::move 实现，首先，通过右值引用传递模板实现，利用引用折叠原理将右值经过 T&&传递类型保持不变还是右值，而左值经过 T&&变为普通的左值引用，**以保证模板可以传递任意实参，且保持类型不变。**然后我们通过 **static_cast<>进行强制类型转换返回 T&&右值引用**，而 static_cast 之所以能使用类型转换，是通过 remove_refrence::type 模板移除 T&&，T&的引用，获取具体类型 T（**模板偏特化**）。

3、std::move 的优点

- std::move 语句可以将左值变为右值而**避免拷贝构造**。

- std::move 是将对象的状态或者所有权从一个对象转移到另一个对象，只是转移，没有内存的搬迁或者内存拷贝。

```
void foo(const std::string& n)
{
	std::cout << "lvalue" << std::endl;
}

void foo(std::string&& n)
{
	std::cout << "rvalue" << std::endl;
}

void bar()
{
	foo("hello");			// rvalue 右值（只能在右边出现）
	std::string a = "world";
	foo(a);				// lvalue（左值）
	foo(std::move(a));		// rvalue（强行转换）
}

int main()
{
	std::vector<std::string> a = {"hello", "world"};
	std::vector<std::string> b;

	b.push_back("hello");
	b.push_back(std::move(a[1]));

	std::cout << "bsize: " << b.size() << std::endl;
```

```
        for (std::string& x: b)
            std::cout << x << std::endl;
        bar();
        return 0;
}
```

第一种： 在<istream>中的 getline()函数有两种重载形式：

istream& getline (char* s, streamsize n);
istream& getline (char* s, streamsize n, char delim);

作用是：从 istream 中读取至多 n 个字符(包含结束标记符)保存在 s 对应的数组中。即使还没读够 n 个字符，

如果遇到 **delim 或 字数达到限制（没有字符了），则读取终止，delim 都不会被保存进 s 对应的数组中**

会停止(1)到文件结束，(2)遇到函数的定界符，(3)输入达到最大限度。

468. 验证 IP 地址（逻辑判断，看看是 IPV4 还是 IPV6，边界问题，还有怎么用 getline,考虑末尾问题，遇到这个分割符号，或者/n 会停止，之前的字符拿出来。）

难度中等 85

编写一个函数来验证输入的字符串是否是有效的 IPv4 或 IPv6 地址。

- 如果是有效的 IPv4 地址，返回 "IPv4" ；
- 如果是有效的 IPv6 地址，返回 "IPv6" ；
- 如果不是上述类型的 IP 地址，返回 "Neither" 。

IPv4 地址由十进制数和点来表示，每个地址包含 4 个十进制数，其范围为 0 - 255， 用(".")分割。比如，`172.16.254.1`；

同时，**IPv4 地址内的数不会以 0 开头**。比如，地址 `172.16.254.01` 是不合法的。

IPv6 地址由 8 组 16 进制的数字来表示，每组表示 16 比特。这些组数字通过 (":")分割。比如, `2001:0db8:85a3:0000:0000:8a2e:0370:7334` 是一个有效的地址。而且，我们可以加入一些以 0 开头的数字，字母可以使用大写，也可以是小写。所以， `2001:db8:85a3:0:0:8A2E:0370:7334` 也是一个有效的 IPv6 address 地址 (**即，忽略 0 开头，忽略大小写**)。

然而，我们不能因为**某个组的值为 0，而使用一个空的组，以至于出现 (::)** 的情况。 比如， `2001:0db8:85a3::8A2E:0370:7334` 是无效的 IPv6 地址。

同时，在 IPv6 地址中，多余的 0 也是不被允许的。比如， `02001:0db8:85a3:0000:0000:8a2e:0370:7334` 是无效的。

示例 1：

输入：`IP = "172.16.254.1"`**输出：**`"IPv4"`**解释：**有效的 IPv4 地址，返回 `"IPv4"`

示例 2：

输入：`IP = "2001:0db8:85a3:0:0:8A2E:0370:7334"`**输出：**`"IPv6"`**解释：**有效的 IPv6 地址，返回 `"IPv6"`

示例 3：

输入：`IP = "256.256.256.256"`**输出：**`"Neither"`**解释：**既不是 IPv4 地址，又不是 IPv6 地址

解题思路

题目本身不难，考虑所有边界条件即可：

IPv4：

段地址数只能为 4；

以 0 开头时只允许长度为 1；

段地址必须是可以进行 int()的字符串；

int()之后必须在[0,255]区间；

IPv6：

段地址数只能为 8；

段地址**只允许长度为[1,4]区间；**

段地址每个字符必须是合法的 16 进制字符，例如 G 不合法；

作者：darkwhite

链接：
https://leetcode-cn.com/problems/validate-ip-address/solution/yong-shi-ji-bai-96jian-dan-si-lu-by-darkwhite/

来源：力扣（LeetCode）


```
class Solution {
public:
    string validIPAddress(string IP) {
        if (isValidIPv4(IP)) return "IPv4";
        if (isValidIPv6(IP)) return "IPv6";
```

```
        return "Neither";
    }

    // 优雅的 split
    void split(const string s, vector<string>& vs, const char delim= ' '){
        istringstream iss(s);
        string temp;
        while (getline(iss,temp,delim)){
            vs.emplace_back(move(temp));
        }
        if (!s.empty() && s.back() == delim) vs.push_back({});//加这句的原因是
getline 不会识别最后一个 delim,避免误判
"172.16.254.1.","2001:0db8:85a3:0:0:8A2E:0370:7334:"之类的情况
    }

    // 判定是否 IPv4
    bool isValidIPv4(string IP){
        vector<string> vs;
        split(IP,vs,'.');
        if (vs.size()!=4) return false;//总共必须 4 个块

        for (auto &v:vs) {
            if (v.empty() || (v.size()>1 && v[0] == '0') || v.size()>3) return false;
//如果其中是空的，或者多余一个，以 0 开头，或者大于 3 位
            for (auto c:v) {
                if (!isdigit(c)) return false;//每个数必须是数字
            }
            int n = stoi(v);
            if (n<0||n>255) return false;//整个数大于 0-255
        }

        return true;
    }

    // 判定是否 IPv6
    bool isValidIPv6(string IP){
        vector<string> vs;
        split(IP,vs,':');
        if (vs.size()!=8) return false;
        for (auto &v:vs) {
            if (v.empty() || v.size()>4 ) return false;//只允许 1-4
            for (auto c:v){
                if (!(isdigit(c) || (c>='a'&&c<='f') || (c>='A'&&c<='F'))) return
false;//如果其中三者都不满足，就是 false 了。
```

```
            }
        }

        return true;

    }
};
```

532. 数组中的 k-diff 数对

难度中等 121

给定一个整数数组和一个整数 **k**，你需要在数组里找到**不同的** k-diff 数对，并返回不同的 **k-diff 数对** 的数目。

这里将 **k-diff** 数对定义为一个整数对 *(nums[i], nums[j])*，并满足下述全部条件：

- *0 <= i, j < nums.length*
- *i != j*
- $|nums[i] - nums[j]| == k$

注意，$|val|$ 表示 *val* 的绝对值。

示例 1：

输入：`nums = [3, 1, 4, 1, 5], k = 2` **输出：**`2` **解释：**数组中有两个 `2-diff` 数对，`(1, 3)` 和 `(3, 5)`。

尽管数组中有两个 `1`，但我们只应返回不同的数对的数量。

示例 2：

输入：nums = [1, 2, 3, 4, 5], k = 1 **输出：**4 **解释：**数组中有四个 1-diff 数对, (1, 2), (2, 3), (3, 4) 和 (4, 5)。

示例 3：

输入：nums = [1, 3, 1, 5, 4], k = 0 **输出：**1 **解释：**数组中只有一个 0-diff 数对，(1, 1)。

示例 4：

输入：nums = [1,2,4,4,3,3,0,9,2,3], k = 3

之前做的，来补个题解。

扫描一遍，用两个 hash 或 set（当哈希用）存已访问的数和已发现的 k-diff 中的较小值。。 O(n) 复杂度

```
def findPairs(self, nums: List[int], k: int) -> int:
        if k<0:
            return 0
        saw, diff = set(), set()
        for i in nums:
            if i-k in saw:
                diff.add(i-k)
```

```
            if i+k in saw:
                diff.add(i)
            saw.add(i)
        return len(diff)
```

作者：yybeta

链接：https://leetcode-cn.com/problems/k-diff-pairs-in-an-array/solution/ha-xi-onzui-jian-dan-jie-fa-by-mai-mai-mai-mai-zi/

来源：力扣（LeetCode）

解题思路

查找 k-diff 数对时，保证前数大于后数。

为了保证 k-diff 不重复，我们对每个 k-diff 数组中的最大值进行标记，将 k-diff 中前数的 value 设为 1.

代码（用哈希，加分类讨论 k==0，只要返回重复两次及以上的元素，大的在前面 nums[i],找小的，大的被访问了计数 1）

```
class Solution {
public:
    int findPairs(vector<int>& nums, int k) {
        unordered_map<int, int> map;
        int count = 0;
```

```
        if(k < 0)//
            return 0;

        if(k == 0) {
        //当 k=0 时，其实就是找数组中有没有重复元素 且重复元素个数大于等于 2 时 我们都视为只有 1 组 K-diff
            for(int i = 0; i < nums.size(); i++) {
            ++map[nums[i]];
            if(map[nums[i]] == 2)
                count++;
            }
            return count;
        }
        //k 为其他数字时，先构建 hash 表，将 value 初始化为 0.有没有被当做开头
        for(int i = 0; i < nums.size(); i++){
            map[nums[i]] = 0;
        }
        for(int i = 0; i < nums.size(); i++) {
            if(map.find(nums[i] - k) != map.end()) {
            //若 nums[i]-k 存在于数组中。注：nums[i]-k 必然小于 nums[i] diff 数对为 {num[i], nums[i]-k} 都是正整数 从大到小
                if(map[nums[i]] == 0 ) {//已经存在的 diff 数对中还没有出现过 nums[i] 为第一个数
                    count++;//数对数量+1
                    map[nums[i]] = 1;//给 nums[i]标记，表示已经有 nums[i]开头的数对了，后面不再考虑
                }
            }
        }
        return count;
    }
};
```

作者：Fisnake

链接：
https://leetcode-cn.com/problems/k-diff-pairs-in-an-array/solution/c-ha-xi-biao-fang-fa-xiao-bai-ye-neng-kan-dong-by-/

来源：力扣（LeetCode）

1438. 绝对差不超过限制的最长连续子数组（滑动窗口法+multiset 有序集合）

难度中等 174

给你一个整数数组 *nums* ，和一个表示限制的整数 *limit*，请你返回最长连续子数组的长度，该子数组中的任意两个元素之间的绝**对差必须小于或者等于 *limit* 。**

如果不存在满足条件的子数组，则返回 *0* 。

示例 1：

输入：`nums = [8,2,4,7], limit = 4` **输出：**`2` **解释：**所有子数组如下：

`[8]` 最大绝对差 `|8-8| = 0 <= 4.`

`[8,2]` 最大绝对差 `|8-2| = 6 > 4.`

`[8,2,4]` 最大绝对差 `|8-2| = 6 > 4.`

`[8,2,4,7]` 最大绝对差 `|8-2| = 6 > 4.`

`[2]` 最大绝对差 `|2-2| = 0 <= 4.`

`[2,4]` 最大绝对差 `|2-4| = 2 <= 4.`

`[2,4,7]` 最大绝对差 `|2-7| = 5 > 4.`

`[4]` 最大绝对差 `|4-4| = 0 <= 4.`

`[4,7]` 最大绝对差 `|4-7| = 3 <= 4.`

[7] 最大绝对差 |7-7| = 0 <= 4.

因此，满足题意的最长子数组的长度为 2 。

示例 2：

输入：nums = [10,1,2,4,7,2], limit = 5 **输出：**4 **解释：**满足题意的最长子数组是 [2,4,7,2]，其最大绝对差 |2-7| = 5 <= 5 。

方法一：滑动窗口 + 有序集合

思路和解法

我们可以枚举每一个位置作为右端点，找到其对应的最靠左的左端点，满足区间中最大值与最小值的差不超过 \textit{limit}limit。

注意到随着右端点向右移动，左端点也将向右移动，于是我们可以使用滑动窗口解决本题。

为了方便统计当前窗口内的最大值与最小值，我们可以使用平衡树：

语言自带的红黑树，例如 \texttt{C++}C++ 中的 \texttt{std::multiset}std::multiset，\texttt{Java}Java 中的 \texttt{TreeMap}TreeMap；

第三方的平衡树库，例如 \texttt{Python}Python 中的 \texttt{sortedcontainers}sortedcontainers（事实上，这个库的底层实现并不是平衡树，但各种操作的时间复杂度仍然很优秀）；

手写 \texttt{Treap}Treap 一类的平衡树，例如下面的 \texttt{Golang}Golang 代码。

来维护窗口内元素构成的有序集合。

作者：LeetCode-Solution

链接：
https://leetcode-cn.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/solution/jue-dui-chai-bu-chao-guo-xian-zhi-de-zui-5bki/

来源：力扣（LeetCode）


```
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        multiset<int> s;//排序
        int n = nums.size();
        int left = 0, right = 0;
        int ret = 0;
        while (right < n) {
            s.insert(nums[right]);
            while (*s.rbegin() - *s.begin() > limit) {
                s.erase(s.find(nums[left++]));//当最大值大于的话，将最左边删除，并且left++
            }
            ret = max(ret, right - left + 1);更新长度
            right++;//更新 right
        }
        return ret;
    }
};
```

作者：LeetCode-Solution

链接：
https://leetcode-cn.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/solution/jue-dui-chai-bu-chao-guo-xian-zhi-de-zui-5bki/

来源：力扣（LeetCode）

33. 搜索旋转排序数组（二分查找）

难度中等 1253

整数数组 *nums* 按升序排列，数组中的值 **互不相同** 。

在传递给函数之前，*nums* 在预先未知的某个下标 *k*（*0 <= k < nums.length*）上进行了 **旋转**，使数组变为 *[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]*（下标 **从 0 开始** 计数）。例如，*[0,1,2,4,5,6,7]* 在下标 3 处经旋转后可能变为 *[4,5,6,7,0,1,2]* 。

给你 **旋转后** 的数组 *nums* 和一个整数 *target* ，如果 *nums* 中存在这个目标值 *target* ，则返回它的索引，否则返回 *-1* 。

示例 1：

输入：nums = [4,5,6,7,0,1,2], target = 0 **输出：**4

示例 2：

输入：nums = [4,5,6,7,0,1,2], target = 3 **输出：**-1

示例 3：

输入：nums = [1], target = 0 **输出：**-1

提示：

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n = (int)nums.size();
        if (!n) {
            return -1;
        }
        if (n == 1) {
            return nums[0] == target ? 0 : -1;
        }
        int l = 0, r = n - 1;
        while (l <= r) {
            int mid = (l + r) / 2;
            if (nums[mid] == target) return mid;//中间相等
            if (nums[0] <= nums[mid]) {//如果左边有序，并且在左边，缩小右边，否则缩小
左边
                if (nums[0] <= target && target < nums[mid]) {
                    r = mid - 1;
                } else {
                    l = mid + 1;
                }
            } else {//同理，右边有序，缩小左边
                if (nums[mid] < target && target <= nums[n - 1]) {
                    l = mid + 1;
                } else {
                    r = mid - 1;
                }
            }
        }
        return -1;
    }
};
```

作者：LeetCode-Solution

链接：
https://leetcode-cn.com/problems/search-in-rotated-sorted-array/solution/sou-suo-xuan-zhuan-pai-xu-shu-zu-by-leetcode-solut/

来源：力扣（LeetCode）

222. 完全二叉树的节点个数（左右递归，不讲伍德）

难度中等 459

给你一棵 **完全二叉树** 的根节点 *root* ，求出该树的节点个数。

完全二叉树 的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该**层最左边的若干位置**。若最底层为第 *h* 层，则该层包含 *1~ 2h* 个节点。

示例 1：

输入：root = [1,2,3,4,5,6]**输出：**6

示例 2：

输入：root = []**输出：**0

简洁代码

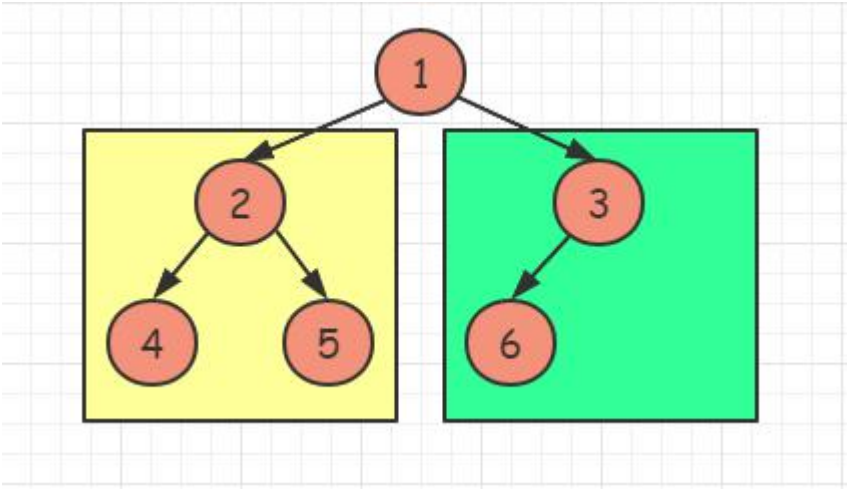
```
class Solution {
public:
    void countNodes(TreeNode* root){
        if(!root) return 0;
        return countNodes(root->left) + countNodes(root->right) + 1;
    }
};
```

```
class Solution {
public:
    void countNodes(TreeNode* root){
        if(!root) return 0;
        return countNodes(root->left) + countNodes(root->right) + 1;
    }
};
```

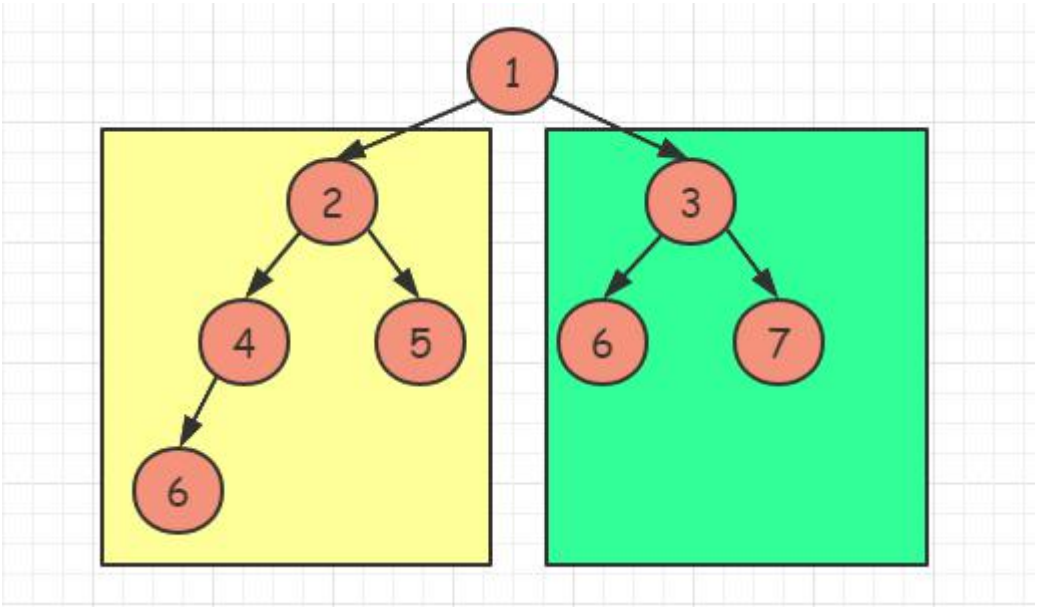
2．根据完全二叉树的性质简化遍历次数

这是一棵完全二叉树：除最后一层外，其余层全部铺满；且最后一层向左停靠

如果根节点的左子树深度等于右子树深度，则说明左子树为满二叉树



如果根节点的左子树深度大于右子树深度，则说明右子树为满二叉树



如果知道子树是满二叉树，那么就可以轻松得到该子树的节点数目：(1<<depth) - 1; // depth 为子树的深度；为了加快幂的运算速度，可以使用移位操作符

接着我们只需要接着对另一子树递归即可

时间复杂度为 O(logn * logn)，空间复杂度为 O(1)【不考虑递归调用栈】

```
class Solution {
public:
    // 统计树的深度
    int countLevels(TreeNode* root) {
        int levels = 0;
        while (root) {
            root = root->left; levels += 1;
        }
        return levels;
    }
    int countNodes(TreeNode* root){
        // 2. 利用完全二叉树性质简化遍历次数
        if(root == nullptr) return 0;
```

```
        int left_levels = countLevels(root->left);
        int right_levels = countLevels(root->right);
        // 左子树深度等于右子树深度，则左子树是满二叉树
        if(left_levels == right_levels){
            return countNodes(root->right) + (1<<left_levels);//还有自身节点 2^n
        }else{
            return countNodes(root->left) + (1<<right_levels);
        }
    }
};
```

对于这份代码，我们可以较容易的看到，还是存在重复遍历的情况；例如：遍历完左右子树的深度后，再下一轮迭代时，还是会重复一次子树深度的遍历，不想改了~~~

3．二分查找

根据完全二叉树的性质，我们可以清楚的知道：总节点数 = 倒数第二层以上的节点数 + 最后一层的节点数

除最后一层外，这棵树为满二叉树，节点数为：2^depth_prev - 1，depth_prev 为倒数第二层树的深度

最后一层的节点数的范围是 [1, 2^depth_prev]；并且依次靠左排列

所以现在的问题就转换为判断最后一层节点数

先不说怎么判断一个节点是否存在，假设这个方法为 is_exist()，我们先了解一下二分法如何使用：

最后一层节点情况： 1 代表非空节点，0 代表空节点

所以现在的问题就可以简单的看成最后一个 1 的索引位置，怎么用二分法就不用我说了。

接下来是如何判断最后一层某个节点是否存在，也就是 is_exist()函数：

给定最后一层某节点的位置索引 index，将他和分界线比大小，就可以判断该节点在左子树还是右子树，例如：现在查找 6 这个节点，索引为 3，大于分界线 2，所以 6 在右子树；对右子树重复操作即可，剩下的工作就交给迭代了~~~

```
class Solution {

public:

    // 求二叉树的深度

    int countLevels(TreeNode* root) {

          int levels = 0;

          while (root) {

              root = root->left; levels += 1;

        }
```

```
    return levels;
}

/*
* 功能： 判断最后一层第 index 个索引是否存在
* root： 二叉树根节点
* index：判断最后一层索引为 index 的节点是否存在，索引范围是[1, 2^depth]
* depth：倒数第二层的深度，这是因为满二叉树最后一层的节点数等于 2^depth
*/
bool is_exist(TreeNode* root, int index, int depth) {
    TreeNode* node = root;
    while (depth) {
        // 最后一层分界线
        int mid = ((1 << depth) >> 1);
        if (index > mid) {
            // 如果在右子树，需要更新索引值
            index -= mid;
            node = node->right;
        }
```

```
        else {
            node = node->left;
        }
        depth -= 1;
    }
    return node != nullptr;
}

int countNodes(TreeNode* root) {
    // 3. 二分查找
    if (root == nullptr) return 0;
    // 二叉树深度
    int depth = countLevels(root);
    // 倒数第二层深度
    int depth_prev = depth - 1;

    int start = 1, end = (1 << depth_prev), mid = 0;
    while (start <= end) {
        mid = start + ((end - start) >> 1);
```

```
            if (is_exist(root, mid, depth_prev)) start = mid + 1;
            else end = mid - 1;
        }
        // start - 1 为最后一层节点数
        int ret = (1 << depth_prev) - 1 + start - 1;
        return ret;
    }
};
```

说实话，个人感觉我这个二分查找部分没写好，不想改了，有兴趣的可以自行优化~~~

哪里有问题，还请指出呀！！！

作者：zuo-10

链接：
https://leetcode-cn.com/problems/count-complete-tree-nodes/solution/c-san-chong-fang-fa-jie-jue-wan-quan-er-cha-shu-de/

来源：力扣（LeetCode）


```
class Solution {public:
    int countNodes(TreeNode* root) {
        TreeNode* cur =root;
```

```
        int level =1;
        int h = getDepth(root);
        int numbers=0;
        while(cur){
            TreeNode* temp = cur->right;
            if(level+getDepth(temp)==h &&temp){
                cur =cur->right;
                numbers +=pow(2,h-level-1);
                level +=1;
            }else{
                cur =cur->left;
                level+=1;
            }
        }
        return numbers+pow(2,h-1);
    }

    int getDepth(TreeNode* root){
        int depth =0;
        while(root){
            root=root->left;
            depth++;
        }
        return depth;
    }
};
```

142. 环形链表 II（找环的起点，用快慢指针）

难度中等 929

给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 *null*。

为了表示给定链表中的环，我们使用整数 *pos* 来表示链表尾连接到链表中的位置（索引从 0 开始）。

如果 *pos* 是 *-1*，则在该链表中没有环。**注意，*pos* 仅仅是用于标识环的情况，并不会作为参数传递到函数中。**

说明：不允许修改给定的链表。

进阶：

- 你是否可以使用 *O(1)* 空间解决此题？

示例 1：

```
class Solution {

public:

    int calculate(string s) {

        stack<int> ops;

        ops.push(1);

        int sign = 1;
```

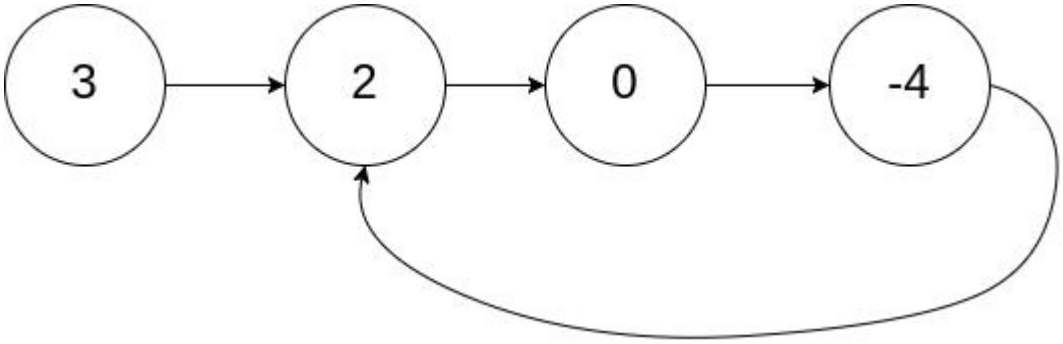
```
int ret = 0;

int n = s.length();

int i = 0;

while (i < n) {

    if (s[i] == ' ') {

        i++;

    } else if (s[i] == '+') {

        sign = ops.top();

        i++;

    } else if (s[i] == '-') {

        sign = -ops.top();

        i++;

    } else if (s[i] == '(') {

        ops.push(sign);

        i++;

    } else if (s[i] == ')') {

        ops.pop();

        i++;

    } else {
```

```
                long num = 0;
                while (i < n && s[i] >= '0' && s[i] <= '9') {
                    num = num * 10 + s[i] - '0';
                    i++;
                }
                ret += sign * num;
            }
        }
        return ret;
    }
};
```

作者：LeetCode-Solution

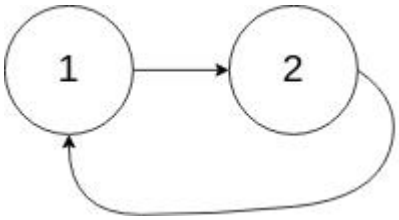
链接：https://leetcode-cn.com/problems/basic-calculator/solution/ji-ben-ji-suan-qi-by-leetcode-solution-jvir/

来源：力扣（LeetCode）



输入：head = [3,2,0,-4], pos = 1 **输出：**返回索引为 1 的链表节点**解释：**链表中有一个环，其尾部连接到第二个节点。

示例 2：



输入：head = [1,2], pos = 0 **输出：**返回索引为 0 的链表节点**解释：**链表中有一个环，其尾部连接到第一个节点。

示例 3：


```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while (fast != nullptr) {
            slow = slow->next;
            if (fast->next == nullptr) {
                return nullptr;//就没有环了
            }
            fast = fast->next->next;
```

```
            if (fast == slow) {//当两者相等
                ListNode *ptr = head;//一个从起点一个从，环相遇的地方
                while (ptr != slow) {//当两者相等，就是环的入口
                    ptr = ptr->next;
                    slow = slow->next;
                }
                return ptr;
            }
        }
        return nullptr;
    }
};
```

作者：LeetCode-Solution

链接：
https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/huan-xing-lian-biao-ii-by-leetcode-solution/

来源：力扣（LeetCode）

集合 MAP


```
class Solution {

public:

    ListNode *detectCycle(ListNode *head) {

        unordered_set<ListNode *> visited;

        while (head != nullptr) {

            if (visited.count(head)) {

                return head;
```

```
            }
            visited.insert(head);
            head = head->next;
        }
        return nullptr;
    }
};
```

作者：LeetCode-Solution

链接：https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/huan-xing-lian-biao-ii-by-leetcode-solution/

来源：力扣（LeetCode）

224. 基本计算器（只有加减号，双栈，一个存储数字，一个存储运算符，先符号，后数字，先得到的是符号+ -，之后是数字，可以看做，符号（数字）的加法,有数字就合并，括号的话，将之前的res 和操作符，入栈，并且 res=0, ops=1，遇到右括号出栈，res=res*ops+nums，

防止最后一个，这个数字，操作完了，num=0）

难度困难 526

给你一个字符串表达式 `s` ，请你实现一个基本计算器来计算并返回它的值。

示例 1：

输入：`s = "1 + 1"`**输出：**`2`

示例 2：

输入：`s = " 2-1 + 2 "`**输出：**`3`

示例 3：

输入：`s = "(1+(4+5+2)-3)+(6+8)"`**输出：**`23`

和基本计算器 II 相比，本题多了左右括号，少了 * 和 / ，但其实大同小异吧。

有括号的题目其实我们已经可以条件反射**用栈来解决了**，用一个栈来存放 '(' 外计算得到的数值，一个栈用于**存放 '(' 外的运算符**。如果遇到 ')'，就更新一下括号内外表达式的运算结果。

```
class Solution {
public:
    int calculate(string s) {
```

```
        stack <int> nums, ops;
        long num = 0;
        int res = 0;
        int op = 1;
        for (char c : s) {
            if (isdigit(c)) {//第一个肯定数字吧
                num = num * 10 + c - '0';
            }
            else {
                res += op * num;//得到这个数
                num = 0;//重新开始
                if (c == '+') op = 1;
                else if (c == '-') op = -1;
                else if (c == '(') {//括号将当前的加进去
                    nums.push(res);
                    ops.push(op);
                    res = 0;//当前为 1 结果为 ros
                    op = 1;
                }
                else if (c == ')' && ops.size()) {//保证有符号
                    res = ops.top() * res + nums.top();
                    ops.pop();
                    nums.pop();
                }//top 就是括号内的加上之前的带的符号，+之前的 result
（nums）
            }
        }
        res += op * num;//最后数字结束了还有一个
        return res;
    }
};
```

772. Basic Calculator III（每次做完，记得 num=0，ops 保留上次的。用 Python 出栈来解答，pop(0)每次删除第一个元素）

左括号就递归，s 每次都 pop 掉，默认第一个 sign +,如果 c 不是数字，并且不是空格，或者到结束了，长度为 0，就按照四则运算来呗，已知前面的数和前面的符号，进行四则运算，更新当前数字-0，和符号。如果是右括号，break，并且返回 stack，总的。

Python3 · 智能模式

```python
class Solution:
    def calculate(self,s: str) -> int:
        def helper(s: list) -> int:
            stack = []
            sign = '+'
            num = 0
            while len(s) > 0:
                c = s.pop(0)
                if c.isdigit():
                    num = 10 * num + int(c)
                # 遇到左括号开始递归
                if c == '(':
                    num = helper(s)
                if (not c.isdigit() and c != '') or len(s) == 0:
                    if sign == '+':
                        stack.append(num)
                    elif sign == '-':
                        stack.append(-num)
                    elif sign == '*':
                        stack[-1] = stack[-1] * num
                    elif sign == '/':
                        stack[-1] = int(stack[-1] / float(num))
                    num = 0
                    sign = c
                    if c == ')': break
            return sum(stack)
        return helper(list(s))
```

```cpp
class Solution {
public:
    int calculate(string s) {
        stack<long> nums;
        stack<char> ops;
        const auto higher = [](char a, char b) {
            if ((a == '*' || a == '/') && (b == '+' || b == '-'))
return true;
            return false;
        };
        const auto op = [&]() {
            const auto b = nums.top(); nums.pop();
            const auto a = nums.top(); nums.pop();
            const auto o = ops.top(); ops.pop();
            switch (o) {
                case '+':
                    nums.push(a + b);
                    break;
                case '-':
```

```
                    nums.push(a - b);
                    break;
                case '*':
                    nums.push(a * b);
                    break;
                case '/':
                    nums.push(a / b);
                    break;
            }
        };
        long num = 0;
        for (int i = 0; i < s.length(); ++i) {
            auto c = s[i];
            if (isdigit(c)) {
                num = c - '0';
                while (i + 1 < s.length() && isdigit(s[i + 1])) {
                    num = num * 10 + (s[i + 1] - '0');
                    ++i;
                }
                nums.push(num);
                num = 0;
            } else if (c == '(') {
                ops.push(c);
                if (s[i + 1] == '-') {
                    nums.push(0);
                    ++i;
                    ops.push('-');
                }
            } else if (c == '+' || c == '-' || c == '*' || c == '/')
{
                if (c == '-' && nums.empty()) {
                    nums.push(0);
                    ops.push(c);
                    continue;
                }
                while (!ops.empty() && ops.top() != '(' && !higher(c,
ops.top())) {
                    op();
                }
                ops.push(c);
            } else if (c == ')') {
                while (!ops.empty() && ops.top() != '(') {
                    op();
                }
```

```
                ops.pop();
            }
        }
        while (!ops.empty()) {
            op();
        }
        return nums.top();
    }
};
```

968. 监控二叉树（分三个状态进行叠加，a root 放摄像头，覆盖整颗数需要的摄像头，b 无论放不放 root，覆盖整颗数，c 无论放不放 root，覆盖整颗子树，注意 null，abc==最大值 0 0，a<=b<=c，最终结果一定是 b，a 就是自身放+左右子树，不确认放，覆盖子树，因为 root 放了之后，左右子树肯定可以，=lc+rc+1， int b = min(a, min(la + rb, ra + lb));//无论放不放，子树总要有个放啊，root 才可以，int c = min(a, lb + rb);保证左右两个子树都被覆盖）

难度困难 270

给定一个二叉树，我们在树的节点上安装摄像头。

节点上的每个摄影头都可以监视**其父对象、自身及其直接子对象。**

计算监控树的所有节点所需的最小摄像头数量。

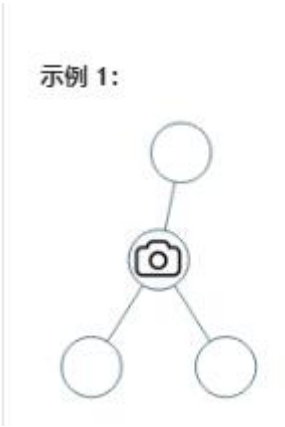
示例 1：

输入：`[0,0,null,0,0]`

输出：`1`

解释：如图所示，一台摄像头足以监控所有节点。

示例 2：



方法一：动态规划（很巧妙）
思路与算法

本题以二叉树为背景，不难想到用递归的方式求解。本题的难度在于如何从左、右子树的状态，推导出父节点的状态。

假设当前节点为 root，其左右孩子为 left,right。如果要覆盖以 root 为根的树，有两种情况：**(分为 root 处放不放摄像头)**

若在 root 处安放摄像头，则孩子 left,right 一定也会被监控到。此时，只需要保证 left 的两棵子树被覆盖，同时保证 right 的两棵子树也被覆盖即可。
否则， 如果 root 处不安放摄像头，则除了覆盖 root 的两棵子树之外，孩子 left,right 之一必须要安装摄像头，从而保证 root 会被监控到。

为了表述方便，我们约定：如果某棵树的所有节点都被监控，**则称该树被「覆盖」**。

根据上面的讨论，能够分析出，对于每个节点 root ，需要维护三种类型的状态：

状态 aa： root 必须放置摄像头的情况下，覆盖整棵树需要的摄像头数目。
状态 bb：覆盖整棵树需要的摄像头数目，无论 root 是否放置摄像头。
状态 cc：覆盖两棵子树需要的摄像头数目，无论节点 root 本身是否被监控到。
根据它们的定义，一定有 a≥b≥c。

对于节点 root 而言，设其左右孩子 left,right 对应的状态变量分别为 (l_a,l_b,l_c)
根据一开始的讨论，我们已经得到了求解 a,ba,b 的过程：
对于 cc 而言，要保证两棵子树被完全覆盖，要么 \textit{root}root 处放置一个摄像头，需要的摄像头数目为 aa；要么 \textit{root}root 处不放置摄像头，此时两棵子树分别保证自己被覆盖，需要的摄像头数目为 l_b + r_bl

需要额外注意的是，对于 \textit{root}root 而言，如果**其某个孩子为空，则不能通过在该孩子处放置摄像头的方式，监控到当前节点**。因此，该孩子对应的变量 aa 应当返回一个大整数，用于标识不可能的情形。

最终，根节点的状态变量 bb 即为要求出的答案。

代码

C++JavaGolangJavaScriptPython3C

```
struct Status {
    int a, b, c;
};

class Solution {
public:
    Status dfs(TreeNode* root) {
        if (!root) {
            return {INT_MAX / 2, 0, 0};//第一个表示不存在，后面就是 0 0 用作计算
        }
        auto [la, lb, lc] = dfs(root->left);
        auto [ra, rb, rc] = dfs(root->right);
        int a = lc + rc + 1;
//root 有的话，那么他的左右子树都能够被检测到所以，就加上 lc（无论左右根（当前左右子树）被覆盖的情况）
        int b = min(a, min(la + rb, ra + lb));//
//无论 root+覆盖整个=min(有 root 覆盖整个，（无论 root）min(左右至少有个 root 覆盖)，左右有个，那么祖 root 会被覆盖
        int c = min(a, lb + rb);
//无论 root+覆盖子树=min(有 root 覆盖整个，（无论 root）min(左右覆盖整个)
        return {a, b, c};
    }
```

```
    int minCameraCover(TreeNode* root) {
        auto [a, b, c] = dfs(root);
        return b;
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/binary-tree-cameras/solution/jian-kong-er-cha-shu-by-leetcode-solution/
来源：力扣（LeetCode）

726. 原子的数量（指针移动，括号就是递归，本身里面就会按照字典序排列，用 map，递归）

难度困难 119
给定一个化学式 `formula`（作为字符串），返回每种原子的数量。

原子总是以一个大写字母开始，接着跟随 0 个或任意个小写字母，表示原子的名字。

如果数量大于 1，原子后会跟着数字表示原子的数量。如果数量等于 **1 则不会跟数字**。例如，H2O 和 H2O2 是可行的，但 H1O2 这个表达是不可行的。

两个化学式连在一起是新的化学式。例如 H2O2He3Mg4 也是化学式。

一个括号中的化学式和数字（可选择性添加）也是化学式。例如 (H2O2) 和 (H2O2)3 是化学式。

给定一个化学式，输出所有原子的数量。格式为：第一个（按字典序）原子的名子，跟着它的数量（如果数量大于 1），然后是第二个原子的名字（按字典序），跟着它的数量（如果数量大于 1），以此类推。

示例 1:

输入:

```
formula = "H2O"
```

输出: "H2O"

解释:

原子的数量是 `{'H': 2, 'O': 1}`。

示例 2:

输入:

```
formula = "Mg(OH)2"
```

输出: `"H2MgO2"`

解释:

原子的数量是 `{'H': 2, 'Mg': 1, 'O': 2}`。

就用简单的递归，思路清晰，例外情况统一处理，不用正则表达式，更不用 LL 文法

jasonL5

发布于 2019-08-282.0k 递归 C++

leetcode 数据库题目全部题解

本题就是带括号表达式求值的一种。

难点在于如何处理括号和求括号内部的表达式。

解法一 递归求解括号中的表达式

整体分成三大部分：

1. 在没有遇到括号前，逐个**解析原子名称和数量**。
2. 在遇到左括号时，递归解析表达式，将结果存入一个哈希表中。例如，(SO3)2，遇到左括号，递归解析 SO3。
3. 递归返回之后，看右括号后边有**没有数字**。当有数字时，要取出当倍数。例如：(SO3)2，右括号后边的数字 2 是括号内部表达式的倍数。

答案

用 map 自动排序的字典

```
tring countOfAtoms(string formula) {
        map<string,int> sub;
        int r=dfs(formula,0,sub);
        //排序
        string ans;
        for (auto iter = sub.begin();iter!=sub.end();iter++){
                ans+=(*iter).first;
                if ((*iter).second != 1)
                    ans+=to_string((*iter).second);
            }
            return ans;
```

map<string,int> mymap; 最后统计每个字符串出现的次数，而且自动按照字典序排序

我写的（string 自动排成字典序，大写字母（可能有小写），表示，开始判断只可能是左括号，还有大写字母，右括号和数字都被遍历了，isalpha，是字母，islower 是小写，isdigit 是数字，没有 whileP 要++，括号的话，递归，将里面的拿出来乘以括号后的数字，没有的话，就找出数字。）

```
#include <iostream>
#include<limits>
#include<vector>
#include<map>
#include<algorithm>
#include<sstream>

using namespace std;
class Solution {
public:


    string countOfAtoms(string formula) {
        map<string,int> sub;//最后的样子
        int r=dfs(formula,0,sub);
        //排序
        string ans;
        for (auto iter = sub.begin();iter!=sub.end();iter++){
            ans+=(*iter).first;
```

```
            if ((*iter).second != 1)
                ans+=to_string((*iter).second);//倍数
        }
        return ans;
}

int dfs(const string& s,int beg,map<string,int>& out){
        int p=beg;
        while(p<s.size()){

            //第一个是否是大写字母
            if(s[p] == '('){//进去递归
                map<string,int> sub2;//容器
                int next_p = dfs(s,++p,sub2);//break，返回的是括号，等待跳出来的
                //更新 p 到')'的下一个位置
                p = next_p + 1;
                //括号后是否有倍数
                long long f = 0;
                while(p < s.size() && isdigit(s[p])){
                    f = 10*f + (s[p]-'0');
                    p++;
                }
                if(f==0) f=1;//就一个
                //累加括号内的计数
                for(auto & x : sub2){
                    out[x.first] += x.second * f;//要乘起来
                }
            }
            else if(s[p]==')'){
                break;//这个跳出来
            }
            else{
                int beg_of_name = p;
                //不可能是数字开头，一定是个大写字母，跳过
                ++p;//second

                //后面是小写字母时，全部跳过，直到数字或大写字母或'()'
                while(p < s.size() && isalpha(s[p]) && islower(s[p])){
                    p++;
                }
                string name=s.substr(beg_of_name,p-beg_of_name);//p 取不到，后面长度
                //获取个数
                long long count = 0;
                while(p < s.size() && isdigit(s[p])){
```

```
                count = count * 10 + (s[p]-'0');
                p++;
            }
            if(count == 0) count=1;//如果没有，就是 1，加 1
            //cout << name <<","<<count<<endl;
            out[name] += count;//统计所有的个数
        }
    }
    return p;//返回下一个位置
}
};


int main()
{
    Solution s;
    cout<<s.countOfAtoms("K4(ONM(SO3)2)2")<<endl;
    return 0;
}
```

我写用 map

```
class Solution {
public:


    string countOfAtoms(string formula) {
        map<string,int> sub;
        int r=dfs(formula,0,sub);
        //排序
        string ans;
        for (auto iter = sub.begin();iter!=sub.end();iter++){//排好序，全部拿出来，不等于 1 要相乘
            ans+=(*iter).first;
            if ((*iter).second != 1)
                ans+=to_string((*iter).second);
        }
        return ans;
    }

    int dfs(const string& s,int beg,map<string,int>& out){
        int p=beg;
        while(p<s.size()){
```

```
            //第一个是否是大写字母
            if(s[p] == '('){
                map<string,int> sub;//临时存储的
                int next_p = dfs(s,++p,sub);
                //更新 p 到')'的下一个位置
                p = next_p + 1;
                //括号后是否有倍数
                long long f = 0;
                while(p < s.size() && isdigit(s[p])){//括号后的数字拿出来
                    f = 10*f + (s[p]-'0');
                    p++;
                }
                if(f==0) f=1;
                //累加括号内的计数
                for(auto & x : sub){
                    out[x.first] += x.second * f;//字典序，自己会放好
                }
            }
            else if(s[p]==')'){
                break;
            }
            else{
                int beg_of_name = p;
                //不可能是数字开头，一定是个大写字母，跳过
                ++p;

                //后面是小写字母时，全部跳过，直到数字或大写字母或'('
                while(p < s.size() && isalpha(s[p]) && islower(s[p])){
                    p++;
                }
                string name=s.substr(beg_of_name,p-beg_of_name);
                //获取个数
                long long count = 0;
                while(p < s.size() && isdigit(s[p])){
                    count = count * 10 + (s[p]-'0');
                    p++;
                }
                if(count == 0) count=1;
                //cout << name <<","<<count<<endl;
                out[name] += count;
            }
        }
        return p;
}
```

};

c++ aggregate ’std::stringstream ss’ has incomplete type and cannot be defined

绯浅 yousa 2017-09-24 17:36:15　23333　收藏 2
分类专栏：　c++
版权
c++ aggregate ‘std::stringstream ss’ has incomplete type and cannot be defined
这个问题是使用了 stringstream 这个类，但没有包含头文件 **ssteam 的缘故**

增加#include <sstream>即可解决
————————————————————

streamstring 在调用 str()时，会返回临时的 string 对象。而因为是临时的对象，所以它在整个表达式结束后将会被析构。

如果需要进一步操作 string 对象，先把其值赋给一个 string 变量后再操作。

Cpp 代码

```cpp
 stringstream ss("01234567890123456789012345678901234567890123456789");
 //错误用法
 const char* cstr2 = ss.str().c_str();
//正确用法
const string& str2 = ss.str();
const char* cstr2 = str2.c_str();
```

使用 stringstream 时的清空操作

在 C++中可以使用 stringstream 来很方便的进行类型转换，字符串串接，不过注意重复使用同一个 stringstream 对象时要 先继续清空，而清空很容易想到是 clear 方法，而在 stringstream 中这个方法实际上是清空 stringstream 的状态（比如出错等），真 正清空内容需要使用.str("")方法。

980. 不同路径 III（递归+回溯，走过的当做-1，visit）

难度困难 129

在二维网格 grid 上，有 4 种类型的方格：

- 1 表示起始方格。且**只有一个起始方格**。
- 2 表示结束方格，且**只有一个结束方格**。
- 0 表示我们**可以走过的空方格**。
- -1 表示我们无法跨越的障碍。

返回在四个方向（上、下、左、右）上行走时，从起始方格到结束方格的**不同路径的数目。**

每一个无障碍方格都要通过一次，但是一条路径中不能重复通过同一个方格。

示例 1：

输入： `[[1,0,0,0],[0,0,0,0],[0,0,2,-1]]`

输出： 2

解释： 我们有以下两条路径：

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2)

2. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2)

示例 2：

输入： `[[1,0,0,0],[0,0,0,0],[0,0,0,2]]`

输出： 4

解释： 我们有以下四条路径：

1. (0,0),(0,1),(0,2),(0,3),(1,3),(1,2),(1,1),(1,0),(2,0),(2,1),(2,2),(2,3)

2. (0,0),(0,1),(1,1),(1,0),(2,0),(2,1),(2,2),(1,2),(0,2),(0,3),(1,3),(2,3)

3. (0,0),(1,0),(2,0),(2,1),(2,2),(1,2),(1,1),(0,1),(0,2),(0,3),(1,3),(2,3)

4. (0,0),(1,0),(2,0),(2,1),(1,1),(0,1),(0,2),(0,3),(1,3),(1,2),(2,2),(2,3)

1，元组简介

`tuple` 是一个固定大小的不同类型值的集合，是泛化的 `std::pair`。我们也可以把他当做

一个通用的结构体来用，不需要创建结构体又获取结构体的特征，在某些情况下可以取代结构体使程序更简洁，直观。std::tuple 理论上可以**有无数个任意类型的成员变量**，而 std::pair 只能是 2 个成员，因此在需要保存 **3 个及以上的数据时就需要使用 tuple 元组**了。

————————————————————

1.此类题目，数据量很小，暴力 dfs，一般用回溯算法，用一个 visited 辅助

2.但是本题，可以把起点和终点记录下来，原数组可以起 visited 数组的作用

```
class Solution {
public:
    // 网格中 0 的个数
    int zero_grid=0;//私有变量

    int uniquePathsIII(vector<vector<int>>& grid) {
        int res=0;
        int row=0;
        int col=0;
        for(int i=0; i<grid.size(); i++)
            for(int j=0; j<grid[i].size(); j++){
                if(grid[i][j]==1){
                    row=i;//先找到起始点
                    col=j;//找到起始点
                }
                else if(grid[i][j]==0)
                    zero_grid++;//每个 0 都要经过
                else{};
            }
        backtrack(grid, res, 0, row, col);

        return res;
    }

    void backtrack(vector<vector<int>>& grid, int& res, int step, int row, int col){
        // 两个返回条件
        // 注意合并一些判断条件
        if(row<0||row>=grid.size()||col<0||col>=grid[0].size()||grid[row][col]==-1)
            return;//越界或者不能走
        if(grid[row][col]==2){//这是终止点
            if(step>zero_grid)//步长=终点+0 的个数，所以是大于
```

```
                res++;//几个结果
            return;
        }
        // 做出选择
        grid[row][col]=-1;//走过了，认为是 visit
        // 在选择列表中进行递归循环
        backtrack(grid, res, step+1, row-1, col);//每个选择
        backtrack(grid, res, step+1, row, col+1);
        backtrack(grid, res, step+1, row+1, col);
        backtrack(grid, res, step+1, row, col-1);
        // 撤销选择
        grid[row][col]=0;//回溯到上面

    }
};
```

作者：enlighten-9
链接：https://leetcode-cn.com/problems/unique-paths-iii/solution/can-kao-yi-xia-hui-su-suan-fa-de-kuang-jia-by-enli/
来源：力扣（LeetCode）

132. 分割回文串 II（动态规划，dp[i]，表示 s[0..i]最小分割次数）

难度困难 405
给你一个字符串 `s`，请你将 `s` 分割成一些子串，使每个子串都是回文。

返回符合要求的 **最少分割次数** 。

示例 1：

输入： `s = "aab"`

输出： `1`

解释： 只需一次分割就可将 *`s`* 分割成 `["aa","b"]` 这样两个回文子串。

示例 2：

输入： `s = "a"`

输出： `0`

示例 3：

输入： `s = "ab"`

输出： `1`

JavaScript Python

方法一：动态规划

思路与算法

设 $f[i]$ 表示字符串的前缀 $s[0..i]$ 的**最少**分割次数。要想得出 $f[i]$ 的值，我们可以考虑枚举 $s[0..i]$ 分割出的最后一个回文串，这样我们就可以写出状态转移方程：

$$f[i] = \min_{0 \le j < i} \{f[j]\} + 1, \quad \text{其中 } s[j+1..i] \text{ 是一个回文串}$$

即我们枚举最后一个回文串的起始位置 $j+1$，保证 $s[j+1..i]$ 是一个回文串，那么 $f[i]$ 就可以从 $f[j]$ 转移而来，附加 1 次额外的分割次数。

注意到上面的状态转移方程中，我们还少考虑了一种情况，即 $s[0..i]$ 本身就是一个回文串。此时其不需要进行任何分割，即：

$$f[i] = 0$$

那么我们如何知道 $s[j+1..i]$ 或者 $s[0..i]$ 是否为回文串呢？我们可以使用与「131. 分割回文串的官方题解」中相同的预处理方法，将字符串 s 的每个子串是否为回文串预先计算出来，即：

> 设 $g(i,j)$ 表示 $s[i..j]$ 是否为回文串，那么有状态转移方程：
>
> $$g(i,j) = \begin{cases} \texttt{True}, & i \ge j \\ g(i+1,j-1) \wedge (s[i] = s[j]), & \text{otherwise} \end{cases}$$
>
> 其中 $\wedge$ 表示逻辑与运算，即 $s[i..j]$ 为回文串，当且仅当其为空串（$i > j$），其长度为 1（$i = j$），或者首尾字符相同且 $s[i+1..j-1]$ 为回文串。

这样一来，我们只需要 $O(1)$ 的时间就可以判断任意 $s[i..j]$ 是否为回文串了。通过动态规划计算出所有的 f 值之后，最终的答案即为 $f[n-1]$，其中 n 是字符串 s 的长度。

```
class Solution {
public:
```

```
    int minCut(string s) {
        int n = s.size();
        vector<vector<int>> g(n, vector<int>(n, true));//i<=i 默认就是了

        for (int i = n - 1; i >= 0; --i) {//倒序遍历，得到所有是否是回文的子集，字串是否回文
            for (int j = i + 1; j < n; ++j) {
                g[i][j] = (s[i] == s[j]) && g[i + 1][j - 1];
            }
        }

        vector<int> f(n, INT_MAX);//默认最大
        for (int i = 0; i < n; ++i) {
            if (g[0][i]) {
                f[i] = 0;//自身就是，就不要分割了。
            }
            else {
                for (int j = 0; j < i; ++j) {
                    if (g[j + 1][i]) {//如果后面的是，就将前面的+1
                        f[i] = min(f[i], f[j] + 1);
                    }
                }
            }
        }

        return f[n - 1];
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/palindrome-partitioning-ii/solution/fen-ge-hui-wen-chuan-ii-by-leetcode-solu-norx/
来源：力扣（LeetCode）

218. 天际线问题(只能是左端点，用multiset <x,y>,自动排序，x 轴从左到右，如果 y 都是左端点，更负的在前面，也就是更高的在前面，排列好后，遍历，

给定一个从小到大排列的 multiset height，开始是 0,还有一个记录最后位置的 last{0,0}，左端点就放入到 heights,如果 height 最大值，不等于 last[1]，就需要更新，last,并且插入，如果是右端点需要移除。始终要最高点，最高点有变化，就要插入）

难度困难 377

城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。给你所有建筑物的位置和高度，请返回由这些建筑物形成的 **天际线** 。

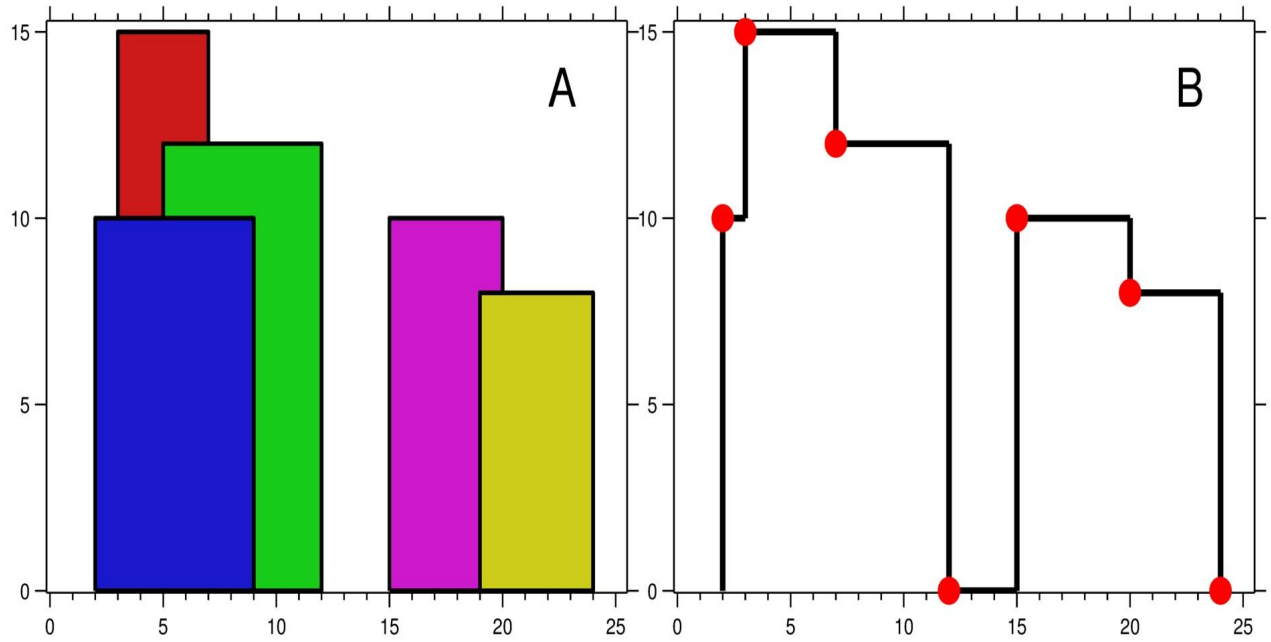
每个建筑物的几何信息由数组 `buildings` 表示，其中三元组 `buildings[i] = [lefti, righti, heighti]` 表示：

- $left_i$ 是第 `i` 座建筑物左边缘的 `x` 坐标。
- $right_i$ 是第 `i` 座建筑物右边缘的 `x` 坐标。
- $height_i$ 是第 `i` 座建筑物的高度。

天际线 应该表示为由 “关键点” 组成的列表，格式 $[[x_1,y_1],[x_2,y_2],...]$ ，并按 **x 坐标** 进行 **排序** 。**关键点是水平线段的左端点**。列表中最后一个点是最右侧建筑物的终点，`y` 坐标始终为 `0` ，仅用于标记天际线的终点。此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。

注意：输出天际线中不得有连续的相同高度的水平线。例如 `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` 是不正确的答案；三条高度为 5 的线应该在最终输出中合并为一个：`[...[2 3], [4 5], [12 7], ...]`

示例 1：



输入：`buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]`

输出：`[[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]`

解释：

图 A 显示输入的所有建筑物的位置和高度，

图 B 显示由这些建筑物形成的天际线。图 B 中的红点表示输出列表中的

很巧妙的做法，利用了 muliset 这一数据结构自动排序的特性。

multiset 中的元素是 pair，对 pair 排序默认的方式是，**先比较 first，哪个小则排在前；first 相等则 second 小的排在前**。 而 first 这里表示横坐标，second 为负时，表示建筑的左侧在这一位置，其绝对值表示建筑在的高度；second 为正时，表示建筑的右侧在这一位置。

所以对 muliset 遍历时，首先会**取出横坐标小的点**。如果 2 个点横坐标相等，会先取出 second 小的点，对于负数来说，其实就是高度更高的建筑。也就是说，两个点上有高度不同的建筑，会先取高的出来放入高度集合，集合中**高度最大值和之前高度不同**，就直接放入结果。后面更低高度的建筑加入并**不会改变最大高度**。

如果 second 为正，表示建筑物在此处结束，需要把相应高度从高度集合中删除。有相同建筑同时在此结束，则会先让**较低的建筑离开**，因为它们不会改变最大高度。只有当最高的建筑物离开时，才进行改变。

如果一个位置既有建筑物进来，又有建筑物离开，会先选择进来的，同理。 总结起来，我就是想说，这里把建筑物起始点的高度设为负数，真的很巧妙。

巧妙，multset 遍历，自动排序，first 相等的话，更大的负号，也就是更高的在前面，如果这个高度没出去的话（还在这个建筑的区间内），下次进来的，还是被这个高度给覆盖了，这个点就没用了。前面有个 0，表示，高度都出去了，就是 0 了。

```
class Solution {
public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        multiset<pair<int, int>> all;//存储每个点，按照 first 排序，从小到大，左边在前，负数
        vector<vector<int>> res;//结尾

        for (auto& e : buildings) {
            all.insert(make_pair(e[0], -e[2])); // critical point, left corner，负数左边
            all.insert(make_pair(e[1], e[2])); // critical point, right corner
        }

        multiset<int> heights({0}); // {{0}}保存当前位置所有高度。从低到高，保证没有高度了，就变为 0 了，可以放进去，自动排序，哦，哥哥，从小到大
        vector<int> last = {0, 0}; // 保存上一个位置的横坐标以及高度 最高的
        for (auto& p : all) {
            if (p.second < 0) heights.insert(-p.second); // 左端点，高度入堆
            else heights.erase(heights.find(p.second)); // 右端点，移除高度，剩下的不等，也要放进来

            // 当前关键点，最大高度（最后一个）
            auto maxHeight = *heights.rbegin();//这是最大的数，相等的话，就被覆盖了，不会进去

            // 当前最大高度如果不同于上一个高度，说明这是一个转折点
            if (last[1] != maxHeight) {//始终要最高点，最高点，
                // 更新 last，并加入结果集
                last[0] = p.first;
                last[1] = maxHeight;
                res.push_back(last);
            }
        }

        return res;
    }
};
```

作者：ivan_allen
链接：https://leetcode-cn.com/problems/the-skyline-problem/solution/218tian-ji-xian-wen-ti-sao-miao-xian-fa-by-ivan_al/
来源：力扣（LeetCode）

273. 整数转换英文表示（先构建 1-19 20-90 的 map 表，分别按照三位，相除 base=1000 取余，得到对应位数的百位，百位第一位不为 0，直接加，不过 hundred,前后都要空格，第二位等于 0，第三位不等于 0 加上 1-19 的三位，如果第二位 2-9，就是 20-90，在加上第三位，否则肯定在 1-19 之间，num2*10+num3 映射得了，后面都要加一个空格）

难度困难 139

将非负整数 num 转换为其对应的英文表示。

示例 1：

输入： num = 123

输出： "One Hundred Twenty Three"

示例 2：

输入： num = 12345

输出： "Twelve Thousand Three Hundred Forty Five"

示例 3：

输入： num = 1234567

输出： "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

示例 4：

输入： num = 1234567891

输出： "One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand Eight Hundred Ninety One"

一.思路

1234567890，按照西方数字的划分，从**低位到高位 3 位划分一个单位**，依次为千 **Thousand**，**百万 Million**，**十亿 Billion**，直到十亿就不用划分，划分后就是 1/234/567/890

用英文表达就是 1 Billion 234 Million 567 Thousand 890,接下来只需要把这些数字翻译成百十个位的表达就行了

所以最终结果就是"One Billion Two Hundred Thirty Four Million Five Hundred Sixty Seven Thousand Eight Hundred Ninety"

具体空格处理可以直接看代码

作者：wen-jian-69

链接：https://leetcode-cn.com/problems/integer-to-english-words/solution/cdi-gui-xie-fa-dai-ma-han-zhu-shi-by-wen-jmfu/

来源：力扣（LeetCode）


```
class Solution {
    unordered_map<int, string> oneToNineteen;//直接对应 1-19
    unordered_map<int, string> twtyToNity;//对应 20-90
    //处理百十个位
    void help(int hundred, string& ans) {
        //处理百
            int num1 = hundred / 100, num2 = hundred % 100 / 10, num3 = hundred % 10;//三位取出来
            if (num1 != 0) ans += oneToNineteen[num1] + " Hundred ";//直接加,百位，前后带一个空格
            if (num2 == 0) {
//第二位等于 0，第三位不等于 0  加第三位
                if (num3 != 0) ans += oneToNineteen[num3] + " ";//有个空格
            } else if (2 <= num2 && num2 <= 9) {//第二位
                ans += twtyToNity[num2] + " ";//第二位是 2-9，直接加
                if (num3 != 0) ans += oneToNineteen[num3] + " ";//第三位还有加
            } else {
                ans += oneToNineteen[num2 * 10 + num3] + " ";//都有，1-19 直接加
            }

    }
    //记录 1-19、20-90 的英文//百以内的
```

```
    void numOfEnlish() {
        oneToNineteen[1] = "One", oneToNineteen[2] = "Two", oneToNineteen[3] = "Three", oneToNineteen[4] = "Four",
        oneToNineteen[5] = "Five", oneToNineteen[6] = "Six", oneToNineteen[7] = "Seven", oneToNineteen[8] = "Eight",
        oneToNineteen[9] = "Nine", oneToNineteen[10] = "Ten", oneToNineteen[11] = "Eleven",
        oneToNineteen[12] = "Twelve", oneToNineteen[13] = "Thirteen", oneToNineteen[14] = "Fourteen",
        oneToNineteen[15] = "Fifteen", oneToNineteen[16] = "Sixteen", oneToNineteen[17] = "Seventeen",
        oneToNineteen[18] = "Eighteen", oneToNineteen[19] = "Nineteen";
        twtyToNity[2] = "Twenty", twtyToNity[3] = "Thirty", twtyToNity[4] = "Forty", twtyToNity[5] = "Fifty",
        twtyToNity[6] = "Sixty", twtyToNity[7] = "Seventy", twtyToNity[8] = "Eighty", twtyToNity[9] = "Ninety";
    }
    //转换为英文
    void transform(int num, string& ans) {
        int base = 1000;
        int hundred = num % base; //记录为百的三位数，取余
        int thousand = num / (base) % base; //记录为千的三位数 10^3
        int million = num / (base * base) % base; //记录为百万的三位数 10^6
        int billion = num / (base * base * base) % base; //记录为十亿的三位数 10^9
        if (billion != 0) help(billion, ans), ans += "Billion "; //处理十亿
        if (million != 0) help(million, ans), ans += "Million "; //处理百万
        if (thousand != 0) help(thousand, ans), ans += "Thousand "; //处理千
        help(hundred, ans); //处理百
    }
public:
    string numberToWords(int num) {
        if (num == 0) return "Zero"; //遇到为 0 直接返回 zero
        numOfEnlish();
        string ans;
        transform(num, ans);
        ans.erase(ans.size() - 1); //删除最后一位多出的空格
        return ans;
    }
};
```

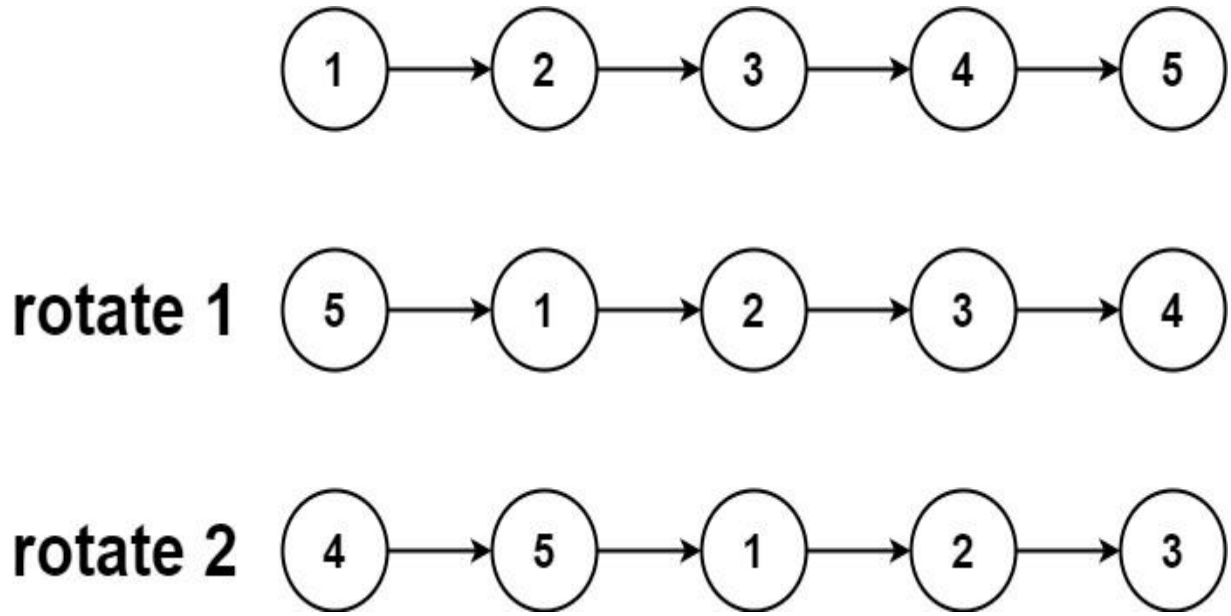
作者：wen-jian-69
链接：https://leetcode-cn.com/problems/integer-to-english-words/solution/cdi-gui-xie-fa-dai-ma-han-zhu-shi-by-wen-jmfu/
来源：力扣（LeetCode）

61. 旋转链表（k%n，往后，闭合为环，找断开环的部分， int add = n - k % n;//从最后一步需要走的步数，走完这个就是新链表的尾部）

难度中等 513

给你一个链表的头节点 head ，旋转链表，将链表每个节点向右移动 k 个位置。

示例 1：



输入：head = [1,2,3,4,5], k = 2

输出：[4,5,1,2,3]

方法一：闭合为环
思路及算法

记给定链表的长度为 nn，注意到当向右移动的次数 k \geq nk≥n 时，我们仅需要向右移动 k \bmod nkmodn 次即可。**因为每 nn 次移动都会让链表变为原状**。这样我们可以知道，新链表的最后一个节点为原链表的第 (n - 1) - (k \bmod n)(n−1)−(kmodn) 个节点（从 00 开始计数）。

这样，我们可以先将给定的链表连接成环，**然后将指定位置断开**。

具体代码中，我们首先计算出链表的长度 nn，并找到该链表的末尾节点，将其与头节点相连。这样就得到了**闭合为环的链表**。然后我们找到新链表的最后一个节点（即原链表的第 (n - 1) - (k \bmod n)(n−1)−(kmodn) 个节点），将当前闭合为环的链表断开，即可得到我们所需要的结果。

特别地，当链表长度不大于 11，或者 kk 为 nn 的倍数时，新链表将与原链表相同，我们无需进行任何处理。

方法一：闭合为环

思路及算法

记给定链表的长度为 n，注意到当向右移动的次数 $k \geq n$ 时，我们仅需要向右移动 $k \bmod n$ 次即可。因为每 n 次移动都会让链表变为原状。这样我们可以知道，新链表的最后一个节点为原链表的第 $(n-1)-(k \bmod n)$ 个节点（从 0 开始计数）。

这样，我们可以先将给定的链表连接成环，然后将指定位置断开。

具体代码中，我们首先计算出链表的长度 n，并找到该链表的末尾节点，将其与头节点相连。这样就得到了闭合为环的链表。然后我们找到新链表的最后一个节点（即原链表的第 $(n-1)-(k \bmod n)$ 个节点），将当前闭合为环的链表断开，即可得到我们所需要的结果。

特别地，当链表长度不大于 1，或者 k 为 n 的倍数时，新链表将与原链表相同，我们无需进行任何处理。

代码

C++JavaPython3JavaScriptGolangC

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (k == 0 || head == nullptr || head->next == nullptr) {
            return head;
        }
        int n = 1;
        ListNode* iter = head;
        while (iter->next != nullptr) {
            iter = iter->next;
            n++;//先取出有几个节点，包括自身
        }
        int add = n - k % n;//从最后一步需要走的步数
        if (add == n) {
            return head;
        }
        iter->next = head;//闭合成环//最后一个
        while (add--) {
            iter = iter->next;//得到尾部节点
```

```
            }
            ListNode* ret = iter->next;//断开，头节点
            iter->next = nullptr;//尾部
            return ret;
        }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/rotate-list/solution/xuan-zhuan-lian-biao-by-leetcode-solutio-woq1/
来源：力扣（LeetCode）

664. 奇怪的打印机（选取中间的 k，如果 s[k]=s[end]共用一个减去 1）

难度困难 104

有台奇怪的打印机有以下两个特殊要求：

1. 打印机每次只能打印同一个字符序列。
2. 每次可以在**任意起始和结束位置打印新字符**，并**且会覆盖掉原来已有的字符**。

给定一个只包含小写英文字母的字符串，你的任务是计算这个打印机打印它需要的最少次数。

示例 1:

输入： `"aaabbb"` **输出：** 2 **解释：** 首先打印 `"aaa"` 然后打印 `"bbb"`。

示例 2:

输入： `"aba"` **输出：** 2 **解释：** 首先打印 `"aaa"` 然后在第二个位置打印 `"b"` 覆盖掉原来的字符 `'a'`。

提示: 输入字符串的长度不会超过 100。

dp[i][j]的含义是 s[i]到 s[j]的打印次数；

在不进行压缩的情况下 s[i,j]是 j-i+1 的；

我们可以吧[i,j]分成两个部分，即 dp[i,j] = dp[i,k]+dp[k+1,j]；

如 aba，可以分成 a ba；但是这时 dp[0][2] = 3，答案错位，因为 ba 中的那个 a 是可以和 a 中的那个 a 公用一次打印次数的。所以应该在原来的基础上-1；

这个操作其实和官方解法的当 s[i] == s[k]的时候 dp[i,j] = dp[i,k-1]+dp[k+1,j]这个思路是一样的；

可能有朋友就问了为什么不是 s[k] == s[k+1]或者 s[i] == s[j]的时候操作一波呢

因为 s[k] == s[k+1]这种方法没办法处理 aa abba 中间有间隔的情况，只能处理连续相等的情况

同样 s[i] == s[j]也一样。

```
class Solution {
public:
    int strangePrinter(string s) {
        int n = s.size();
        if(n <= 1) return n;
        vector<vector<int>> dp(n,vector<int>(n,1000));
        for(int i =0;i<n;i++) dp[i][i] = 1;
        for(int len = 1;len<n;len++){//用长度更新，1 到 n-1step
            for(int i = 0;i+len<n;i++){//索引小于等于 n-1。从 i 到 i+len 中间
                dp[i][i+len] = len+1;//每次更新这个
                for(int k = i;k<(i+len);k++){//中间取任意一个  和戳气球类似
                    int total = dp[i][k] + dp[k+1][i+len];
                    if(s[k] == s[i+len]) total--;//如果中间和最后相等，减去 1；共用一个
                    dp[i][i+len] = min(total,dp[i][i+len]);//分最小的
                }

            }
        }
        return dp[0][n-1];
    }
};
```

倒序（类似戳气球，不过，两边相等要减去 1，初始值给 j-i+1,或者给 1000 也行，k 和 j 相等减去，注意是第二个，和最后一个，其他的只能处理中间连续

相等）

这个更好理解（类似戳气球）

```cpp
class Solution {
public:
    int strangePrinter(string s) {
        int n = s.size();
        if(n <= 1) return n;
        vector<vector<int>> dp(n,vector<int>(n,1000));
        for(int i =0;i<n;i++) dp[i][i] = 1;
        for(int i =n-2;i>=0;i--){
            for(int j = i+1;j<n;j++){
                dp[i][j] = j-i+1;
                for(int k = i;k<j;k++){
                    int total = dp[i][k] + dp[k+1][j];
                    if(s[k] == s[j]) total--;
                    dp[i][j] = min(total,dp[i][j]);
                }

            }
        }
        return dp[0][n-1];
    }
};
```

作者：subshall

链接：
https://leetcode-cn.com/problems/strange-printer/solution/c-dong-tai-gui-hua-by-subshall-2/

来源：力扣（LeetCode）

489. 扫地机器人（怎么回到最初的状态，两次右转，后传，右转，标准 dfs，

四个方向走，如果四个方向都完了，就转动右边 DFS 遍历，采用进换退 思路。

进：当前方向一路向前。

换：前方走不了了，换方向。

退：从当前格子上能走路都走完了，原路回退）

难度困难 86

房间（用格栅表示）中有一个扫地机器人。格栅中的每一个格子有空和障碍物两种可能。

扫地机器人提供 4 个 API，可以向前进，向左转或者向右转。每次转弯 90 度。

当扫地机器人试图进入障碍物格子时，它的碰撞传感器会探测出障碍物，使它停留在原地。

请利用提供的 4 个 API 编写让机器人清理整个房间的算法。

```
interface Robot {

  // 若下一个方格为空，则返回 true，并移动至该方格

  // 若下一个方格为障碍物，则返回 false，并停留在原地

  boolean move();



  // 在调用 turnLeft/turnRight 后机器人会停留在原位置

  // 每次转弯 90 度

  void turnLeft();
```

```
  void turnRight();



  // 清理所在方格

  void clean();

}
```

示例:

输入:

```
room = [

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

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

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

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

  [1,1,1,1,1,1,1,1]

],

row = 1,

col = 3
```

解析:

房间格栅用 `0` 或 `1` 填充。`0` 表示障碍物，`1` 表示可以通过。

机器人从 `row=1`，`col=3` 的初始位置出发。在左上角的一行以下，三列以右。

注意:

1. 输入只用于初始化房间和机器人的位置。你需要“盲解”这个问题。换而言之，你必须在对房间和机器人位置一无所知的情况下，只使用 4 个给出的 API 解决问题。
2. 扫地机器人的初始位置一定是空地。
3. 扫地机器人的初始方向向上。
4. 所有可抵达的格子都是相连的，亦即所有标记为 1 的格子机器人都可以抵达。
5. 可以假定格栅的四周都被墙包围。

更快 用 pair

标准代码

我们从起始位置开始，记录当前的位置为 cell = (0, 0)，以及机器人的朝向 direction = 0；

将起始位置进行清扫，并进行标记（即清扫过的格子也算作障碍）；

依次选择四个朝向 up，right，down 和 left 进行深度优先搜索，相邻的两个朝向仅差一次向右旋转的操作；

对于选择的朝向，检查下一个格子是否有障碍，如果没有，则向对应朝向移动一格，并开始新的搜索；

如果有，则向右旋转。

如果四个朝向都搜索完毕，则回溯到上一次搜索。

```
class Solution {
    int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
    set<pair<int, int> > vis;//表示走过的

    void goBack(Robot& robot) {//回去，两个左转，加上前进在加上两个左转
        robot.turnLeft();robot.turnLeft();
        robot.move();
        robot.turnLeft();robot.turnLeft();
    }

    void dfs(int x, int y, Robot& robot, int dir) {
        robot.clean();
        // mark wall and visited
        vis.insert({x, y});
```

```
            for (int i = 0; i < 4; ++ i) {
                int ndir = (dir + i) % 4;//四个方向，朝向
                int nx = x + dx[ndir], ny = y + dy[ndir];

                if (vis.count({nx, ny}) == 0) {
                    if (robot.move()) {//可以走
                        dfs(nx, ny, robot, ndir);
                        goBack(robot);//回溯
                    }else {
                        vis.insert({nx, ny});
                    }
                }
                robot.turnRight();//不行就像右转，四个方向都遍历完了
            }
        }


public:
    void cleanRoom(Robot& robot) {
        dfs(0, 0, robot, 0);
    }
};
```

代码

```
class Solution {
public:
    vector<vector<int>> dirs = {{-1,0}, {0,1}, {1,0}, {0,-1}};//顺时针方向的改变
    void dfs(Robot& robot, unordered_set<string>& visited, int x, int y, int dir){
        robot.clean();//清扫
        for(int i = 1; i <= 4; i++){//转四次后返回进入函数的初始状态
            robot.turnRight();//右转
            int new_dir = (dir + i) % 4;//下一步的方向
            int xx = x + dirs[new_dir][0];//下一步位置 x
            int yy = y + dirs[new_dir][1];//下一步位置 y
            string next = to_string(xx) + ',' + to_string(yy);
            if(visited.find(next) != visited.end())
                continue;
            visited.insert(next);//记录探查过
            if(robot.move()){//是空地
                dfs(robot, visited, xx, yy, new_dir);//继续扫
                //恢复原状
                robot.turnRight();
                robot.turnRight();//向后转
```

```
                    robot.move();//前进
                    robot.turnRight();
                    robot.turnRight();//向后转
                }
            }
        }
        void cleanRoom(Robot& robot) {
            unordered_set<string> visited;
            visited.insert("0,0");
            dfs(robot, visited, 0, 0, 0);
        }
};
```

作者：SY_rabbit
链接：https://leetcode-cn.com/problems/robot-room-cleaner/solution/c-dfs-by-tmoonli/
来源：力扣（LeetCode）

作者：vtim-i
链接：https://leetcode-cn.com/problems/robot-room-cleaner/solution/hui-su-tan-suo-suo-you-ge-zi-by-vtim-i-lrg5/
来源：力扣（LeetCode）

850. 矩形面积 II（用区间合并的方式做的，确实比较难，难在区间怎么合并，考虑是否需要合并这样的类型，按照 x 的逆序，去取 y 的对数，之后相加，当前进来的 left right，先判定与那个对相交，如果当前 it 不到 end，或者 it。second>left，it 就一直++，如果最终的 it=end 或者，it.left>right，就是落在两个区间内，或者到了最后，直接插入 it 之前即可，否则，就是有交集了

(第一个相交的)，并起来，就是两者的两端端点的最远处，就是新的区间，如果当前的，还要判断合并后的区间是否与下一个区间相交，it++就是下一个，当前 it.second 大于 it++.left，就是有交集，更新当前 it。second 的长度，并且删除掉 it，指向下一个 it）

难度困难 70

我们给出了一个（轴对齐的）矩形列表 *rectangles* 。 对于 *rectangle[i] = [x1, y1, x2, y2]*，其中（x1，y1）是矩形 *i* 左下角的坐标，（x2，y2）是该矩形右上角的坐标。

找出平面中所有矩形叠加覆盖后的总面积。 由于答案可能太大，**请返回它对 10 ^ 9 + 7 取模的结果**。

示例 1：

输入：`[[0,0,2,2],[1,0,2,3],[1,0,3,1]]`**输出：**6 **解释：**如图所示。

示例 2：

输入：`[[0,0,1000000000,1000000000]]`**输出：**49 **解释：**答案是 `10^18` 对 `(10^9 + 7)` 取模的结果， 即 `(10^9)^2 → (-7)^2 = 49` 。

2. 坐标压缩
该方法是一种比较奇妙的解法，由于横纵坐标其实对于矩形来说如何压缩是互不影响的，因为压缩是整体的，x 和 y 如何压缩不重要（因为可以按照扫描线的方式逐行按照下面方式计算面积），所以我们其实可以单独只考虑 x 轴的情况，来分析这种做法的巧妙之处，考虑所有的矩形宽度都是 1，而长度各不相同，例如我们有[0,1],[0,100],[2,10],[3,20]这样四个矩形，

这里数组中第一个是矩形的左边界坐标，第二位是右边界坐标，而宽度假设都为 1，那么此时我们可以将所有横坐标按照从大到小去重后组织成[0,1,2,3,10,20,100],我们对横坐标按照不同的覆盖情况进行分别压缩，将所有的坐标映射到 0,1,2,3……序列上，得到对应关系[0,1,2,3,4,5,6]<->[0,1,2,3,10,20,100],这样我们计算从左到右的面积：

[0,1]区间上压缩是 1:1 的，所以面积为 1；
[1,2]区间上的压缩比是 1:1，所以面积为 1；
[2,3]区间上的压缩比是 1:1，所以面积为 1；
[3,4]区间上的压缩比为 7:1，所以面积为 7；
[4,5]区间上的压缩比为 10:1，所以面积是 10；
[5,6]区间上的压缩比为 90:1，所以面积是 90；
总面积为 100，然后我们可以将纵坐标按照宽度为 1，分别计算结果

作者：run916
链接：https://leetcode-cn.com/problems/rectangle-area-ii/solution/rong-chi-yuan-li-zuo-biao-ya-suo-sao-miao-xian-xia/
来源：力扣（LeetCode）


```
class Solution {
public:
    int rectangleArea(vector<vector<int>>& rect) {
        int N=rect.size();
        unordered_set<int> xval;
        unordered_set<int> yval;
        for(auto r:rect){
            xval.insert(r[0]);
            xval.insert(r[2]);
            yval.insert(r[1]);
            yval.insert(r[3]);
        }
        vector<int> x;
        vector<int> y;
        for(auto v:xval){
            x.push_back(v);
        }
        for(auto v:yval){
            y.push_back(v);
        }
        sort(x.begin(),x.end());
        sort(y.begin(),y.end());
        unordered_map<int,int> hashx;
        unordered_map<int,int> hashy;
        for(int i=0;i<x.size();i++){
```

```
            hashx.emplace(x[i],i);
        }
        for(int i=0;i<y.size();i++){
            hashy.emplace(y[i],i);
        }
        vector<vector<bool>> grid(x.size(),vector<bool>(y.size(),false));
        for(auto r:rect){
            for(int i=hashx[r[0]];i<hashx[r[2]];i++){
                for(int j=hashy[r[1]];j<hashy[r[3]];j++){
                    grid[i][j]=true;
                }
            }
        }
        long int ans=0;
        for(int i=0;i<grid.size();i++){
            for(int j=0;j<grid[0].size();j++){
                if(grid[i][j]){
                    ans+=(long int)(x[i+1]-x[i])*(y[j+1]-y[j]);
                }
            }
        }
        ans%=1000000007;
        return ans;

    }
};
```

作者：run916
链接：https://leetcode-cn.com/problems/rectangle-area-ii/solution/rong-chi-yuan-li-zuo-biao-ya-suo-sao-miao-xian-xia/
来源：力扣（LeetCode）

题意：给出多个矩形的左上角和右下角坐标，求这些矩形面积的并，即相重的面积只算一次

解题思路：

首先离散化：把读进来的**坐标变为一条竖边，一个矩形有两个竖边**：左竖边和右竖边，左竖边用与制造覆盖，右竖边用于消除覆盖，要标记好哪些是左竖边，哪些是右竖边。建一个结构体数组来存放竖边，并按照竖边的很坐标升序排序（从小到大）。

把所有的纵坐标用一个数组存放，并升序排序，离散化结束。

建一个线段树用于查看目前那些纵坐标被覆盖了，从而计算当前的被覆盖纵坐标长度。结合竖边的左右性质就能不断更新线段树，从而计算新的矩形。
数据很水。

线段树

线段树是一种二叉搜索树，与区间树相似，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。

对于线段树中的每一个非叶子节点[a,b]，它的左儿子表示的区间为[a,(a+b)/2]，右儿子表示的区间为[(a+b)/2+1,b]。因此线段树是平衡二叉树，最后的**子节点数目为 N**，即整个线段区间的长度。

使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为 O(logN)。而未优化的空间复杂度为 2N，因此有时需要离散化让空间压缩。

线段树至少支持下列操作:

Insert(t,x):将包含在区间 int 的元素 x 插入到树 t 中;

Delete(t,x):从线段树 t 中删除元素 x;

Search(t,x):返回一个指向树 t 中元素 x 的指针。

一、基本概念

1、线段树是一棵二叉搜索树，它储存的是一个区间的信息。

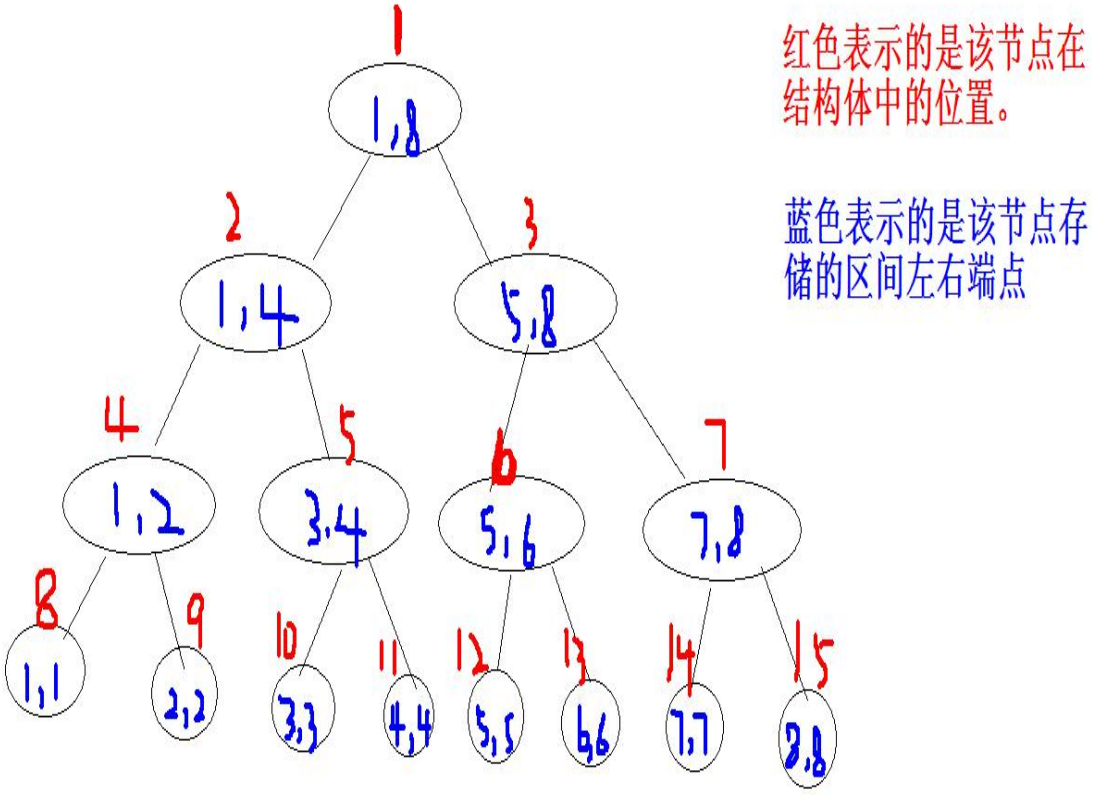
2、每个节点以结构体的方式存储，结构体包含以下几个信息：

区间左端点、右端点；（这两者必有）

这个区间要维护的信息（事实际情况而定，数目不等）。

3、线段树的基本思想：二分。

4、线段树一般结构如图所示：



5、特殊性质：

由上图可得，

1、每个节点的左孩子区间范围为[l，mid]，右孩子为[mid+1,r]

2、对于结点 k，左孩子结点为 2*k，右孩子为 2*k+1，这符合完全二叉树的性质

二、线段树的基础操作

注：以下基础操作均以引例中的求和为例，结构体以此为例：

线性扫描方法（区间合并）

第一步：将所有矩形的 x 坐标放入 xVec
第二步：按照升序排序 x 坐标
第三步：在 xVec 中进行去重
第四步：逆序扫描 x 坐标段，计算在(xVec[i], xVec[i + 1])段中的面积
假设 xVec[i] = 1, xVec[i + 1] = 2，则所有矩形在这个 x 区间段中存在交集的矩形 y 轴段是
矩形[0,0,2,2]中的 Y 轴区间段[0, 2]，矩形[1,0,2,3]中的 Y 轴区间段[0, 3]，矩形[1,0,3,1]中的 Y 轴区间段[0,1]，合并时候 Y 轴区间段段为[0, 3]（类似区间的合并）
distanceX = xVec[i + 1] - xVec[i] = 1， distanceY = 3 - 0 = 3，所以 **S[2] = distanceX * distanceY = 3**

```
//  存储的是 x1 y1 x2y2 类型的
    class Solution {
    public:
        int rectangleArea(vector<vector<int>>& rectangles) {
            vector<int> xVec;//用于存放所有矩形的 x 坐标
            int sumArea = 0, mod = 7 + 1e9;
            //第一步：将所有矩形的 x 坐标放入 xVec
            for (const auto &rectangle : rectangles){
                xVec.push_back(rectangle[0]);
                xVec.push_back(rectangle[2]);
            }
            //第二步：按照升序排序 x 坐标
            sort(xVec.begin(), xVec.end());
            //第三步：在 xVec 中进行去重（unique 是 STL 中的去重函数，它会把重复的元素移动到 xVec 的后端，并且返回第一个重复的值的迭代器）
            xVec.erase(unique(xVec.begin(), xVec.end()), xVec.end());
            //第四步：逆序扫描 x 坐标段，计算在(xVec[i], xVec[i + 1])段中的面积
            for (int i = xVec.size() - 2; i >= 0; --i){
                list<pair<int, int>> myList;//按照递增的顺序存储所有不重复、重叠的 y 轴坐标段  关键
                //如果 rectangle 在(xVec[i], xVec[i + 1])段中有面积，则将它两个 y 轴坐标放入 list，那必须的，肯定不会出现超过多少的，是按照 x 的排列去取 y 对的。
                for (const auto &rectangle : rectangles){
                    if (rectangle[0] <= xVec[i] && rectangle[2] >= xVec[i + 1]){
                        addRange(myList, rectangle[1], rectangle[3]);
                    }
                }
                //然后我们需要把 y 轴这些区间段的长度进行求和
```

```
                long distanceY = 0;
                for (const auto &item : myList){
                    distanceY += item.second - item.first;//这是所有的 y 的间隔
                }
                //xVec[i + 1] - xVec[i]表示 x 轴区间段的宽度，distanceY 是在这个 x 轴段 y 轴各个区间段长度的和
                sumArea = (sumArea + (xVec[i + 1] - xVec[i]) * distanceY) % mod;
            }
            return sumArea;
        }
        //添加 Range[left, right]
        void addRange(list<pair<int, int>> &myList, int left, int right) {
            auto it = myList.begin();
            //第一步：确定[left，right]插入的位置（第一个与它有交集的元素）
            while (it != myList.end() && it->second < left) {//如果对列的右端点，小于左端点只能前进，肯定没有交集，只能下一个，如果进来的左小于右，肯定有交点，就是两个线段，最大的一定要大于最小的。
                ++it;
            }
            //第二步：判断是插入到 list 中还是修改 list
            //如果是插入尾端、或者两个 Range 之间（list 中没有 Range 与它有交集），直接插入
            if (it == myList.end() || it->first > right) {//如果与下一个没有交集，就是在中间，或者没有，直接插入
                myList.insert(it, { left, right});
            }
            else {
                //此时修改 it 指向的元素
                it->first = min(it->first, left);
                it->second = max(it->second, right);//并起来
                //然后看 it 后面是否有需要合并的 Rande
                auto beforeIt = it++;
                while (it != myList.end() && beforeIt->second >= it->first) {
                    beforeIt->second = max(it->second, beforeIt->second);
                    it = myList.erase(it);//更新 it 为下一个。Erase 指向删除的下个指针
                }
            }
        }
```

};

作者：hestyle

来源：CSDN

原文：https://blog.csdn.net/qq_41855420/article/details/90738327

作者：he-style

链接：https://leetcode-cn.com/problems/rectangle-area-ii/solution/leetcode-ju-xing-mian-ji-iisao-miao-fa-by-he-style/

来源：力扣（LeetCode）

1192. 查找集群内的「关键连接」（用dfn自身的序列号和lower*来判断，首先建立每个节点的互通关系，前子节点的回溯大于父节点的序号，就是表示回不去了 unordered_map<int, unordered_set<int>> g；建立一个图）

难度困难 125

力扣数据中心有 n 台服务器，分别按从 0 到 n-1 的方式进行了编号。

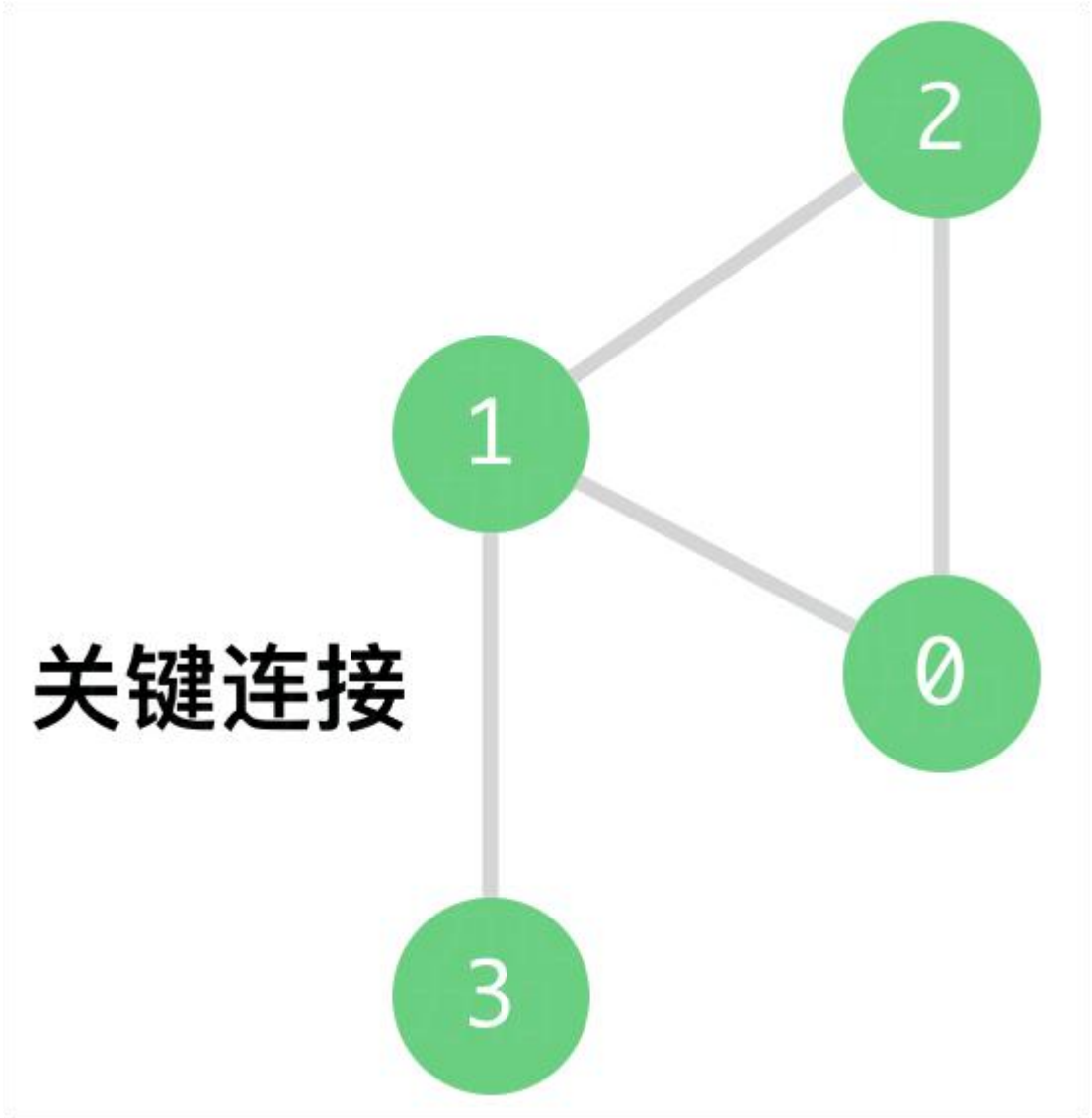
它们之间以「服务器到服务器」点对点的形式相互连接组成了一个内部集群，其中连接 connections 是无向的。

从形式上讲，*connections[i] = [a, b]* 表示服务器 *a* 和 *b* 之间形成连接。任何服务器都可以直接或者间接地通过网络到达任何其他服务器。

「关键连接」是在该集群中的重要连接，也就是说，假如我们将它移除，便会导致某些服务器无法访问其他服务器。

请你以任意顺序返回该集群内的所有 「关键连接」。

示例 1：



输入：`n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]`**输出：**`[[1,3]]`**解释：**`[[3,1]]` 也是正确的。

解题思路

首先，题目的意思很明确，就是找到无向图中的**所有桥**，我们分析可以发现，对于一条边 a-b，如果不存在一条从 b 出发能回到 a 的路径（此路径不能再使用边 b-a），那么 a-b 就是一条桥。由于需要记录边的信息，所以需要使用 DFS，同时我们使用 map 记录所有节点的深度（即该节点距离起始节点的跳数），到达节点 b 之后，我们需要知道节点 b 是否有一条路径回到 a，此时我们初始化 ancentdepth=b.depth，我们遍历节点 b 的所有邻居节点 c（不包括 a），如果 c 已经访问则 ancentdepth=min(map[c],ancentdepth)，否则 ancentdepth=min(dfs(c,b.depth+1),ancentdepth)，遍历结束后若 ancentdepth<b.depth 则 a-b 非桥，否则 a-b 必为桥。

解题思路

如果一个边是关键路径，当且仅当该边不在环中。

使用深度优先搜索查找环。

首先转换成邻接表

深度优先访问

对每一个节点记录访问深度

如果**子节点的深度小于等于当前节点了，说明我们找到环了！**去掉该边

同时记录返回当前节点的最小深度，以便去掉整个环

最后剩下的边就是关键路径~

tarjan 算法 免费编辑 添加义项名

添加义项

?

所属类别：

其他数学相关

一种由 Robert Tarjan 提出的求解有向图强连通分量的线性时间的算法。

算法介绍

如果两个顶点可以相互通达，则称两个顶点**强连通**(strongly connected)。如果有向图 G 的每两个顶点都强连通，称 G 是一个**强连通图**。有向图的极大强连通子图，称为**强连通分量**(strongly connected components)。

例如：在上图中，{1 , 2 , 3 , 4 } , { 5 } ， { 6 } 三个区域可以相互连通，称为这个图的强连通分量。

Tarjan 算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量。

再 Tarjan 算法中，有如下定义。

DFN[i] ：在 DFS 中该节点被搜索的次序(时间戳)

LOW[i] ：为 i 或 i 的子树能够追溯到的最早的栈中节点的次序号

当 DFN[i]==LOW[i]时，为 i 或 i 的子树可以构成一个强连通分量。

下图中，子图{1,2,3,4}为一个强连通分量，因为顶点 1,2,3,4 两两可达。{5},{6}也分别是两个强连通分量。

Tarjan 算法是用来求有向图的强连通分量的。求有向图的强连通分量的 Tarjan 算法是以其发明者 Robert Tarjan 命名的。Robert Tarjan 还发明了求双连通分量的 Tarjan 算法。

Tarjan 算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未**处理的节点加入一个堆栈**，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量。

定义 DFN(u)为节点 u 搜索的次序编号(时间戳)，Low(u)为 u 或 u 的子树能够追溯到的最早的栈中节点的次序号。

当 DFN(u)=Low(u)时，**以 u 为根的搜索子树上所有节点是一个强连通分量**。

接下来是对算法流程的演示。

从节点 1 开始 DFS，把遍历到的节点加入栈中。搜索到节点 u=6 时，DFN[6]=LOW[6]，找到了一个强连通分量。退栈到 u=v 为止，{6}为一个强连通分量。

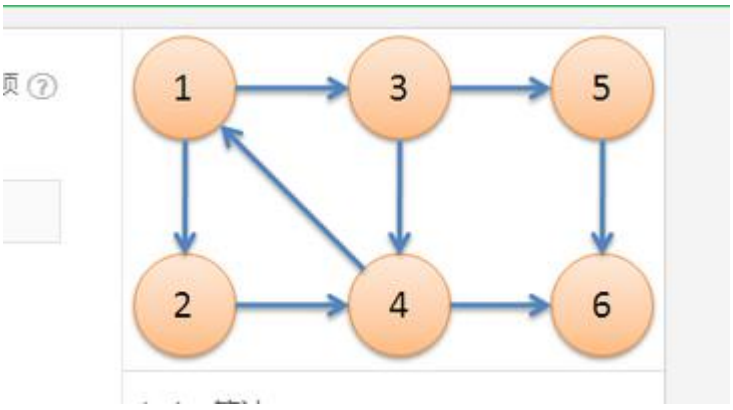
返回节点 5，发现 DFN[5]=LOW[5]，退栈后{5}为一个强连通分量。

返回节点 3，继续搜索到节点 4，把 4 加入堆栈。发现节点 4 向节点 1 有后向边，节点 1 还在栈中，所以 LOW[4]=1。节点 6 已经出栈，(4,6)是横叉边，返回 3，(3,4)为树枝边，所以 LOW[3]=LOW[4]=1。

继续回到节点 1，最后访问节点 2。访问边(2,4)，4 还在栈中，所以 LOW[2]=DFN[4]=5。返回 1 后，发现 DFN[1]=LOW[1]，把栈中节点全部取出，组成一个连通分量{1,3,4,2}。

至此，算法结束。经过该算法，求出了图中全部的三个强连通分量{1,3,4,2},{5},{6}。

可以发现，运行 Tarjan 算法的过程中，每个顶点都被访问了一次，且只进出了一次堆栈，每条边也只被访问了一次，所以该算法的时间复杂度为 O(N+M)。



解题思路（tarjan 的变形，当前子节点的回溯大于父节点的序号，就是表示回不去了）

即寻找图中的桥， 使得去掉该边后，图的连通分量变多。

```
class Solution {
public:
    vector<int> low;//回溯的子节点
    vector<int> dfn;//是否被访问和记录访问顺序
    vector<int> pa;//某个节点的父节点
    int times = 1;
    vector<vector<int>> res;
    vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
        low.resize(n);//回溯的最少
        dfn.resize(n);//初始化为 0，未被访问
        pa = vector<int>(n, -1);

        unordered_map<int, unordered_set<int>> g;

        for (auto& conn : connections) {
            g[conn[0]].insert(conn[1]);
            g[conn[1]].insert(conn[0]);
        }

        for (int i = 0; i < n; ++i) {
            if (!dfn[i]) dfs(i, g);//未被访问
        }

        return res;
    }

    void dfs(int i, unordered_map<int, unordered_set<int>>& g) {
        dfn[i] = low[i] = times++;//当前的节点++和可回溯的节点
```

```
            for (auto iter = g[i].begin(); iter != g[i].end(); ++iter) {
                if (!dfn[*iter]) {//子节点未被访问
                    pa[*iter] = i;//当前访问的节点的父节点都是 i
                    dfs(*iter, g);

                    if (low[*iter] > dfn[i]) {//如果子节点回溯的序号大于自身（父节点次数），就是不强相关，因为回不去了
                        res.push_back({i, *iter});
                    }

                    low[i] = min(low[i], low[*iter]);//i 的最小次数是子节点最小的
                } else if (*iter != pa[i]) {
                    low[i] = min(low[i], low[*iter]);//（前提是，当前节点 i 是 iter 的父亲节点）如果当前节点访问过了并且不是 i 的父节点，当前节点的回溯序号，等于子节点最小的。因为 i 是他的父亲,比*iter 早，所以回溯可以变得更小。
                }
            }
        }
};
```

作者：thebreak

链接：https://leetcode-cn.com/problems/critical-connections-in-a-network/solution/tarjarsuan-fa-qiu-qiao-by-thebreak-vvl7/

来源：力扣（LeetCode）

127. 单词接龙（用一个巧妙的东西来表示 hit h*t hi* *it 是在一起的）

难度困难 721

字典 *wordList* 中从单词 *beginWord* 和 *endWord* 的 **转换序列** 是一个按下述规格形成的序列：

- 序列中第一个单词是 *beginWord* 。
- 序列中最后一个单词是 *endWord* 。
- 每次转换只能改变一个字母。
- 转换过程中的中间单词必须是字典 *wordList* 中的单词。

给你两个单词 *beginWord* 和 *endWord* 和一个字典 *wordList* ，找到

从 *beginWord* 到 *endWord* 的 **最短转换序列** 中的 **单词数目** 。如果不存在这样的转换序列，返回 0。

示例 1：

输入：`beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]`**输出：**`5` **解释：**一个最短转换序列是 `"hit" -> "hot" -> "dot" -> "dog" -> "cog",` 返回它的长度 `5`。

示例 2：

输入：`beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]`
输出：`0` **解释：**`endWord "cog"` 不在字典中，所以无法进行转换。

提示：

- *1 <= beginWord.length <= 10*
- *endWord.length == beginWord.length*
- *1 <= wordList.length <= 5000*
- *wordList[i].length == beginWord.length*

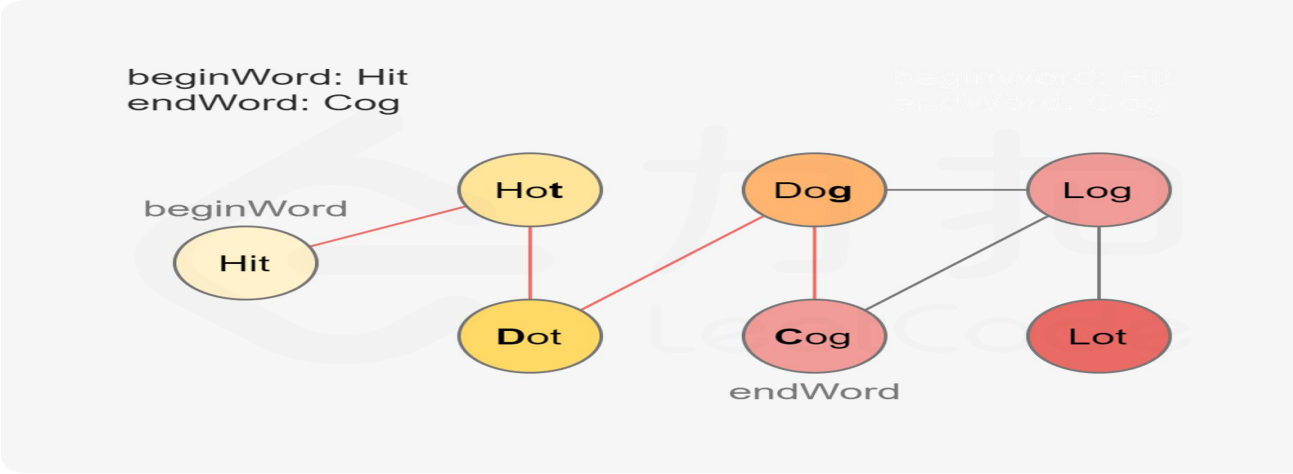
- *beginWord*、*endWord* 和 *wordList[i]* 由小写英文字母组成
- *beginWord* != *endWord*
- *wordList* 中的所有字符串 **互不相同**

方法一：广度优先搜索 + 优化建图

思路

本题要求的是最短转换序列的长度，看到最短首先想到的就是广度优先搜索。想到广度优先搜索自然而然的就能想到图，但是本题并没有直截了当的给出图的模型，因此我们需要把它抽象成图的模型。

我们可以把每个单词都抽象为一个点，如果两个单词可以只改变一个字母进行转换，那么说明他们之间有一条双向边。因此我们只需要把满足转换条件的点相连，就形成了一张图。



基于该图，我们以 beginWord 为图的起点，以 endWord 为终点进行广度优先搜索，寻找 beginWord 到 endWord 的最短路径。

算法

基于上面的思路我们考虑如何编程实现。

首先为了方便表示，我们先给每一个单词标号，即给每个单词分配一个 id。创建一个由单词 word 到 id 对应的映射 wordId，并将 beginWord 与 wordList 中所有的单词都加入这个映射中。之后我们检查 **endWord 是否在该映射内**，若不存在，则输入无解。我们可以使用哈希表实现上面的映射关系。

然后我们需要建图，依据朴素的思路，我们可以**枚举每一对单词的组合**，判断它们是否恰好相差一个字符，以判断这两个单词对应的节点是否能够相连。但是这样效率太低，我们可以优化建图。

具体地，我们可以创建虚拟节点。对于单词 hit，我们创建三个虚拟节点 *it、h*t、hi*，并让 hit 向这三个虚拟节点分别连一条边即可。如果一个单词能够转化为 hit，那么该单词必然会连接到这三个虚拟节点之一。对于每一个单词，我们枚举它连接到的虚拟节点，把该单词对应的 id 与这些虚拟节点对应的 id 相连即可。

最后我们将起点加入队列开始广度优先搜索，当搜索到终点时，我们就找到了最短路径的长度。注意因为添加了虚拟节点，所以我们得到的距离为实际最短路径长度的两倍。同时我们并未计算起点对答案的贡献，所以我们应当返回距离的一半再加一的结果。

```
#include <iostream>
#include<vector>
#include<algorithm>
#include<limits.h>
#include<map>
#include<set>
#include<unordered_map>
#include <sstream>
#include<stack>
#include<unordered_set>
#include<queue>
using namespace std;
class Solution {
public:
    unordered_map<string, int> wordId;
    vector<vector<int>> edge;//建立一个图
    int nodeNum = 0;//每个单词有个 id int

    void addWord(string& word) {
        if (!wordId.count(word)) {
            wordId[word] = nodeNum++;
            edge.emplace_back();//enlarger size
        }
    }

    void addEdge(string& word) {
        addWord(word);
        int id1 = wordId[word];//获取他的 id
        for (char& it : word) {
            char tmp = it;
            it = '*';//it 还是里面的，可以直接改//进行变换
            addWord(word);
            int id2 = wordId[word];
            edge[id1].push_back(id2);//这些都是连接到一起的
            edge[id2].push_back(id1);
            it = tmp;//还原
        }
    }
```

```
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        for (string& word : wordList) {
            addEdge(word);//加边
        }
        addEdge(beginWord);
        if (!wordId.count(endWord)) {
            return 0;
        }
        vector<int> dis(nodeNum, INT_MAX);//最大的
        int beginId = wordId[beginWord], endId = wordId[endWord];// must exit
        dis[beginId] = 0;//设置 0,没取

        queue<int> que;
        que.push(beginId);
        while (!que.empty()) {//不妨加上 size
            int x = que.front();
            que.pop();
            if (x == endId) {
                return dis[endId] / 2 + 1;//每次多走两部 hit h*t hot  加上起点的贡献
            }
            for (int& it : edge[x]) {
                if (dis[it] == INT_MAX) {// not visit and go
                    dis[it] = dis[x] + 1;
                    que.push(it);
                }
            }
        }
        return 0;
    }
};




int main()
{
```

```
        Solution s;
       string beginWord = "hit";
       string endWord = "cog";
      vector<string> wordList = {"hot","dot","dog","lot","log","cog"};


    int a=s.ladderLength(beginWord,endWord,wordList);
   cout<<a<<endl;


        return 0;
   }
```

689. 三个无重叠子数组的最大和

难度困难 106

给定数组 nums 由正整数组成，找到三个互不重叠的子数组的最大和。

每个子数组的长度为 k，我们要使这 3*k 个项的和最大化。

返回每个区间起始索引的列表（索引从 0 开始）。如果有多个结果，返回字典序最小的一个。

示例:

输入: `[1,2,1,2,6,7,5,1], 2` **输出:** `[0, 3, 5]` **解释:** 子数组 `[1, 2], [2, 6], [7, 5]` 对应的起始索引为 `[0, 3, 5]`。

我们也可以取 `[2, 1]`，但是结果 `[1, 3, 5]` 在字典序上更大

方法一：

用一个数组 W 去考虑每个间隔的和，其中每个间隔都是给定的长度 K。要创建 W，我们可以使用前缀和，或者将间隔的和管理为沿数组滑动的窗口。

我们讨论如何简化问题：给定数组 W 和整数 W，i + K <= j 和 j + K <= k 的索引 (i, j, k) 的字典最小元组是什么，它使 W[i]+W[j]+W[k] 最大化？

算法：

算法：

核心思路

这道题应该算是困难题目里比较简单的一道了，最关键的就是知道怎么优化来提高效率。首先给出一种暴力法，先穷举出所有的可能，然后去发现可以优化的地方。

暴力法（用前缀和，获取，以某个数结尾的等宽的和，注意 sum 长度比 num 多 1，注意边界条件，I j m 结尾的三个数，三层循环）

想要穷举所有可能，对于题目中提到的 3 个子数组，显然需要至少三层循环，我们设 i ，j ，m 分别表示三个子数组的结尾下标(开头也相同)，那就需要三层循环来遍历 i ，j ，m，然后还要从这三个下标分别向前求 k 个数的和，也就是说总共需要四层循环，复杂度十分可怕，代码就不给出了。

优化一

这是一种很常见的方法，就是求前缀和，通过前缀和来求子数组的和可以省去一层循环，即知道结尾下标之后向前遍历求和的过程。

代码

```
class Solution {
    public int[] maxSumOfThreeSubarrays(int[] nums, int k) {
        int[] ans = new int[3];
        int[] sum = new int[nums.length + 1];
        for (int i = 0; i < nums.length; i++) {
            //求前缀和
            sum[i + 1] = sum[i] + nums[i];
        }
        for (int i = nums.length; i >= k; i--) {
            //求长度为 k 的子数组的间隔和
            sum[i] = sum[i] - sum[i - k];
        }
        int maxSum = 0;
        //遍历
        for (int i = k; i <= nums.length - 2 * k; i++) {
            for (int j = i + k; j <= nums.length - k; j++) {
                for (int m = j + k; m <= nums.length; m++) {
                    if (maxSum < sum[i] + sum[j] + sum[m]) {
                        maxSum = sum[i] + sum[j] + sum[m];
                        ans[0] = i - k;
                        ans[1] = j - k;
                        ans[2] = m - k;
                    }
                }
            }
        }
        return ans;
    }
}
```

思路仍是暴力，但是通过求前缀和的预处理，把最内层的循环省掉，提高一些效率。

优化二(用前缀和，获取，以某个数结尾的等宽的和，注意 sum 长度比 num 多 1，左右遍历的时候，左边从 k 开始，右边从最后开始，得到的是末端的位置,前缀和，减去的那个数取不到)

优化二

一种直观的想法：如果只找数组中一个最大的数，直接用一个变量遍历一遍即可；那么如果 i , j , m 的下标范围是确定的，问题就会变得同上边的想法，变得很简单。不过这道题要求的

是长度为 k 的子数组，范围就是可变的了。**但是如果固定中间位置的下标 j ，此时他前后两个子数组 i , m 的下标就变成固定范围了，固定范围肯定就会有一个最大值(即最优解)，问题有局部最优解，也就须要用到 DP 了。**
所以我**们使用一个 left 数组保存第一个子数组的局部最优解，用一个 right 数组保存最后一个子数组的局部最优解，然后遍历每一个可能中间下标 j，就可**以求出最终答案，并且可以省去两层循环，效率大大提高。

代码

```
class Solution {
    public int[] maxSumOfThreeSubarrays(int[] nums, int k) {
        int[] ans = new int[3];
        int[] sum = new int[nums.length + 1];// 长度+1，sum[0] = 0 方便求前缀和
        int[] left = new int[nums.length + 1];// 长度与 sum 一致方便运算、理解
        int[] right = new int[nums.length + 1];// 同上
        // 求前缀和
        for (int i = 0; i < nums.length; i++) {
            sum[i + 1] = sum[i] + nums[i];
        }
        // 求间隔为 k 的和，从下标 k 开始求，以结尾为主
        for (int i = nums.length; i >= k; i--) {
            sum[i] = sum[i] - sum[i - k];
        }
        // 求从左边开始，和最大的值最早出现的下标
        int maxNum = 0;
        int index = 0;
        for (int i = k; i <= nums.length - k * 2; i++) {//右边至少留 2k 个数
            if (maxNum >= sum[i]) {// >=号保证存储的是最先出现的下标
                left[i] = index;
            } else {
                left[i] = i;
                index = i;
                maxNum = sum[i];
            }
        }
        // 求从右边开始，和最大的值最早出现的下标
//意思是，从 i 末端到结尾得最大数值，所以需要从后往前遍历，认为 x 是大于 3k 的，后面相加也是 3k 的
        maxNum = 0;
        index = 0;
        for (int i = nums.length; i >= k * 2; i--) {
            if (maxNum > sum[i]) {// >号保证存储的是最先出现的下标，更大
                right[i] = index;
            } else {
                right[i] = i;
```

```
                maxNum = sum[i];
                index = i;
            }
        }
        // 遍历 j 可能的位置，求出所有可能的最大值
//左边的末端最多可以取到 i-k，右边的末端至少要取到 i+k
        maxNum = 0;
        for (int i = k * 2; i <= nums.length - k; i++) {
            if (maxNum < sum[i] + sum[left[i - k]] + sum[right[i + k]]) {
                maxNum = sum[i] + sum[left[i - k]] + sum[right[i + k]];
                ans[0] = left[i - k] - k;// - k 由于求间隔和的时候，和是存储在每个间隔最后一个元素的下标
                ans[1] = i - k;//末尾本来是 i-k+1 包含 k 个数，sum 比 num 多了一位，所以相减 = i-k
                ans[2] = right[i + k] - k;
            }
        }

        return ans;
    }
}
```

我这里将和数组、left 数组、right 数组大小都设为 nums.length + 1，主要是为了方便运算，也方便理解，实际上用到的大小只有 nums.length - k 罢了。如有内容错误的地方还请指出，感谢相遇~

代码

```
class Solution {
    vector<int> maxSumOfThreeSubarrays(vector<int>& nums, int k) {

        vector<int>ans(3);
        int n=nums.size();
        vector<int> sum (n + 1);// 长度+1，sum[0] = 0 方便求前缀和
        vector<int> left (n + 1);// 长度与 sum 一致方便运算、理解
        vector<int> right (n + 1);// 同上
        // 求前缀和
        for (int i = 0; i < n; i++) {
            sum[i + 1] = sum[i] + nums[i];
        }
        // 求间隔为 k 的和，从下标 k 开始求，以 i 结尾间隔为 k 的和
        for (int i = n; i >= k; i--) {
            sum[i] = sum[i] - sum[i - k];
        }
        // 求从左边开始，和最大的值（结尾）最早出现的下标，从左到右的遍历
```

```
        int maxNum = 0;
        int index = 0;
        for (int i = k; i <= n - k * 2; i++) {//后面留 n-2k+1 n 一共 2k 个数
            if (maxNum >= sum[i]) {// >=号保证存储的是最先出现的下标
                left[i] = index;//相等的话，是之前的 index
            } else {
                left[i] = i;
                index = i;
                maxNum = sum[i];
            }
        }
        // 求从右边开始，和最大的值最早出现的下标
//右边倒序，求得是左边界
        maxNum = 0;
        index = 0;
//意思是，从 i 末端到结尾得最大数值，所以需要从后往前遍历，认为 x 是大于 3k 的，后面相加也是 3k 的
能取到的最左边到末尾的最大数值，应该是
        for (int i = n; i >= k * 2; i--) {//这里遍历的是
            if (maxNum > sum[i]) {// >号保证存储的是最先出现的下标，更大
                right[i] = index;
            } else {
                right[i] = i;
                maxNum = sum[i];
                index = i;//记录上一次最大的值
            }
        }
        // 遍历 j 可能的位置，求出所有可能的最大值，遍历中间末尾
        maxNum = 0;
        for (int i = k * 2; i <= n - k; i++) {//留下 n-k+1 到 n 至少一个，这里是 sum
            if (maxNum < sum[i] + sum[left[i - k]] + sum[right[i + k]]) {//3k
                maxNum = sum[i] + sum[left[i - k]] + sum[right[i + k]];
                ans[0] = left[i - k] - k;// - k 由于求间隔和的时候，和是存储在每个间隔最后一个元素的下标
                ans[1] = i - k;//末尾本来是 i-k+1 包含 k 个数，sum 比 num 多了一位，所以相减 = i-k
                ans[2] = right[i + k] - k;
            }
        }

        return ans;
    }
}
```

c++ 动态规划 样例易懂

张小统 L4

发布于 2020-05-13798 动态规划 C++

题意

从一个数组中找到不重叠的三个子数组，每个子数组长度为 k，找到三个子数组最大和，返回下标。
最大和想一下，大概使用 dp 来求最优解。

本题是返回下标，不过我们先不去关心如何找到下标，我们先关心如何找到最大的和。
知道怎么找最大和才能找到它的下标啊。

dp 找最大和

分析这个题中的状态，有两个，下标 i 和到 i 处已经找到子数组数目 j
所以用二维 dp[i][j]来表示**到下标 i 处已经找到的子数组为 j 组**的最大和
下标为 i 的子数组指什么呢？包含下标 i 处的 k 个元素组成的子数组

用例子来解释 dp[i][j]的含义和下标 i 处子数组的概念
例：

[1,2,1,2,6,7,5,1],k=2

下标为 0 的子数组即为[1,2]

下标为 1 的子数组即为[2,1] 很显然，因为不能重叠，下标 0 和 1 的子数组不能同时被选中

dp[0][1]就是说到下标 0 处已经选择了 1 个子数组，所形成的最大和，我们很容易知道，

dp[0][0]=3

*dp[1][2]*就是说到下标 1 处已经选择了 2 个子数组，所形成的最大和，我们很容易知道，不可能发生这种情况

因为假如同时选下标 0 和 1 处的子数组，会造成重叠

到最后，我们可以知道，我们就求 *dp[n-k][3]*

为什么是 n-k，还不是因为最后 k-1 个不能被选为下标，因为不够 k 个

弄明白 dp[i][j]和下标 i 处子数组的概念后，我们开始算最大和。

dp[0][1]->dp[n-k][3]

我们的初始状态是不是就是 dp[0][1]，表示到下标 0 处选 1 个数组，用这个得到 dp[n-k][3]
需要完成两个维度上的增加，第一个是 i，第二个是 j

我们分别来看，如何从 dp[i]增加到 dp[i+1]呢？

dp[i][1]->dp[i+1][1]:

到新的位置 i+1，我们可以做两种选择，选择以 i+1 为下标的子数组，或者不选只有这两种选择。选的话，那么最大和就是以 i+1 为下标的子数组，不选的话，就是 dp[i][1]
这不就代表:

dp[i+1][1]=max(以 i+1 为下标的子数组和,dp[i][1])

dp[i][2]->dp[i+1][2]:

假如想选择 2 个子数组，是不是最早在下标为 k 处，不然会有重复？
在 i+1 处依然做两个选择，选择以 i+1 为下标的子数组，或者不选。
选择就是以 i+1 为下标的子数组和+dp[i][1]，以 i+1 为下标子数组的和是确定的，另一个子数组的和我们要最大的
是不是就是 dp[i][1]，这个里面存储了一个子数组最大和
不选的话，是不是就是 dp[i][2]
这不就代表：

dp[i+1][2]=max(以 i+1 为下标的子数组和+dp[i][1],dp[i][2])

dp[i][3]->dp[i+1][3]:
相信各位读到这里的大佬一定会了，这个方程你们自己来吧！
它应该是这样的：

dp[i+1][3]=max(?,?) ?自己填

这样我们就完成了 dp[0][1]->dp[n-k][3]的转化了！

如何得到下标呢？

从最大和得到下标，其实是很简单的，我们的 dp 数组原来记录的是最大和，现在我们用它记录下标，其实是一样的
为什么呢？
假如说现在我们在 dp[i][2]处记录下标 xt,yy
那么以 xt 为下标子数组的和，和以 yy 为下标子数组的和加起来，是不是就是我们原来记录的最大和？
每一个最大和都是由子数组得到的，而子数组可以用下标表示，所以我们记录下标就可以了
在进行比较的时候，在把下标还原为最大和
以 i 为下标子数组的最大和，我们直接用前缀和就可以算了

假如读者明白最大和可以用下标来代替，并且自己独立完成了我留给大家完成的状态转移方程，那就可以写代码了！

我给大家一份参考的代码，这个代码和我交的不一样，是为了让大家方便理解才写的
没有经过测试，可能有错误

```
vector<int> maxSumOfThreeSubarrays(vector<int>& nums, int K) { // 注意原题这里是小 k，为了和题解中的符号一致，输入改为大 K
    int N = nums.size();
    vector<int> W(N - K + 1, 0);

    int sum = 0;
    for (int i = 0; i < N; ++i) {
```

```
        sum += nums[i];
        if (i >= K) { sum -= nums[i - K]; }
        if (i >= K - 1) { W[i - K + 1] = sum; }
    }

    vector<int> left(W.size(), 0);
    int best = 0;
    for (int i = 0; i < W.size(); ++i) {
        if (W[i] > W[best]) { best = i; } // 注意这里是 >，为了输出是字典序列
        left[i] = best;
    }

    vector<int> right(W.size(), 0);
    best = W.size() - 1;
    for (int i = W.size() - 1; i >= 0; --i) {
        if (W[i] >= W[best]) { best = i; } // 注意这里是 >=，为了输出是字典序列
        right[i] = best;
    }

    vector<int> ans{-1, -1, -1};
    for (int j = K; j < W.size() - K; ++j) {
        int i = left[j - K], k = right[j + K];
        if (ans[0] == -1 || W[i] + W[j] + W[k] >
                W[ans[0]] + W[ans[1]] + W[ans[2]]) {
            ans[0] = i;
            ans[1] = j;
```

```
                ans[2] = k;
            }
        }
        return ans;
    }
```

864. 获取所有钥匙的最短路径

难度困难 75

给定一个二维网格 *grid*。 "." 代表一个空房间， "#" 代表一堵墙， "@" 是起点，（"a", "b", ...）代表钥匙，（"A", "B", ...）代表锁。

我们从起点开始出发，一次移动是指向四个基本方向之一行走一个单位空间。我们不能在网格外面行走，也无法穿过一堵墙。如果途经一个钥匙，我们就把它捡起来。除非我们手里有对应的钥匙，否则无法通过锁。

假设 K 为钥匙/锁的个数，且满足 1 <= K <= 6，字母表中的前 K 个字母在网格中都有自己对应的一个小写和一个大写字母。换言之，每个锁有唯一对应的钥匙，每个钥匙也有唯一对应的锁。另外，代表钥匙和锁的字母互为大小写并按字母顺序排列。

返回获取所有钥匙所需要的移动的最少次数。如果无法获取所有钥匙，返回 -1 。

示例 1：

输入：`["@.a.#","###.#","b.A.B"]`**输出：**`8`

示例 2：

输入：`["@..aA","..B#.","....b"]`**输出：**`6`

提示：

1. *1 <= grid.length <= 30*
2. *1 <= grid[0].length <= 30*
3. *grid[i][j]* 只含有 *'.', '#', '@', 'a'-'f'* 以及 *'A'-'F'*
4. 钥匙的数目范围是 *[1, 6]*，每个钥匙都对应一个不同的字母，正好打开一个对应的锁。

解题思路

主要思路：解题汇总链接

（1）使用广度优先搜索，配合状态压缩；

（2）先遍历原数组，找出 grid 中的**起始点位置和总的钥匙的个数**，并将起始点的位置压入到队列中；

（3）队列中元素存储位置和当前位置的状态，由于钥匙最多只有 6 个，则钥匙的**状态最多只有 64 个**，故可以使用状态压缩，来保存各个钥匙在某个位置时，获得的状态；

用一个二进制来表示当前手里拥有的钥匙的状态，**二进制第 i 位是 0 或是 1 代表手里是否拥有第 i 把**钥匙，因为钥匙数不超过 6，所以只需要 6 个二进制位就可以代表所有可能的手里钥匙的状态，用一个三维数组 dis[i][j][k]代表走到第 i 行 j 列手里拥有的钥匙状态为 k 时需要的步数，剩下的就是简单的 bf

作者：Gaaakki

链接：
https://leetcode-cn.com/problems/shortest-path-to-get-all-keys/solution/java-bfszhuang-tai-ya-suo-by-gaaakki/

来源：力扣（LeetCode）

（4）为了记录之前出现过的位置的状态，使用三维数组来保存从初始位置到各个位置的距离，数组初始置为-1，来标识没有访问过当前位置；

题目要求找到所有的钥匙。将给出的网格当作图，用图遍历的方法进行求解。这样的思路，本身不难想到。

例如：有三把钥匙 a,b,c

"@..c.",

"BD.#C",

"a.Adb"

在这个例子中，要拿到钥匙 a，要先拿到钥匙 b 和 d。必然要拿到钥匙 c。

不论是 bfs 和 dfs 算法，都有节点去重的环节。一个节点遍历后，将其置为已遍历。之后就不会再次遍历此节点，保证算法能正确退出。

考虑上面的例子。通过 c 和 C，拿到 b，d 后。在二维图上，这一路径都被标志为已遍历。算法中，无法按原路返回了。即无法从 b，d 再退到 c。

但是要拿到 a，必须 从 b，d 再退到 c。

这就产生矛盾了。要保证算法正确退出，必须要对已经过的节点，设置为已遍历。又要求从 b，d 按原路返回到 c。

解决办法。

从 c 到 b，d，是为了拿 b，d。在拿到 b，d 前，是没有 b，d 的。

当拿到 b，d 后，返回 c 时，就多了 b 和 d 的信息。

因此，在二维图的基础上，加上已拥有的钥匙信息。将图从二维扩充到三维图，这就解决了上面的矛盾。

尽管从 b，d 回到 c，在二维图上是按原路返回，但在三维图上，却不是遍历已经遍历过的节点。

三维图节点：

(x,y,k)—二维坐标和已拿到的钥匙信息。这是将钥匙对应到整数 k 的低 6 个 bit。6 个 bit 中，每个 bit 对应一个钥匙。

作者：jason-2

链接：
https://leetcode-cn.com/problems/shortest-path-to-get-all-keys/solution/yin-shi-tu-shang-de-yan-du-you-xian-sou-suo-suan-f/

来源：力扣（LeetCode）


```
class Solution {
public:
    int shortestPathAllKeys(vector<string>& grid) {
        int rows = grid.size(), cols = grid[0].size();
        int count_k = 0;
        queue<vector<int>> q;
        //遍历原数组，找出钥匙的个数和初始的位置
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (grid[i][j] >= 'a'&&grid[i][j] <= 'z') {
                    ++count_k;
                }
                else if (grid[i][j] == '@') {
                    q.push({ i,j ,0});//一个钥匙也没有 000
                }
            }
        }
        //各个方向
        vector<vector<int>> next = { {1,0},{-1,0},{0,1},{0,-1} };
        //三维数组，记录到各个位置的距离
        vector<vector<vector<int>>> distance(30, vector<vector<int>>(30, vector<int>(64, -1)));//位置和钥匙状态
        distance[q.front()[0]][q.front()[1]][0]=0;//初始化位置的距离
        int target = (1 << count_k) - 1;//最终所有的钥匙都要找到，既终止位置的状态 2^n-1
        while (!q.empty()) {//终止队列为空
            vector<int> cur_pos = q.front();//当前位置
            q.pop();
            for (int i = 0; i < next.size(); ++i) {//各个方向
                int next_r = cur_pos[0] + next[i][0];
                int next_c = cur_pos[1] + next[i][1];
                if (next_r >= 0 && next_r < rows&&next_c >= 0 && next_c < cols && grid[next_r][next_c] != '#') {//当前方向的位置可能访问
                    if (grid[next_r][next_c] >= 'A'&&grid[next_r][next_c] <= 'Z'
                        &&!(cur_pos[2]&(1<<(grid[next_r][next_c] - 'A')))) {//有锁，但没有当前锁的钥匙 锁固定 100，就看这个位了
```

```
                    continue;
                }
                int cur_state = cur_pos[2];//当前获得的钥匙状态，虽然回溯回来了，但是当前状态是改变了
                if (grid[next_r][next_c] >= 'a'&&grid[next_r][next_c] <= 'z') {//当前方向的位置是钥匙
                    cur_state |= 1 << (grid[next_r][next_c] - 'a');//将当前钥匙的状态加入
                    if (cur_state == target) {//到达终止位置
                        return  distance[cur_pos[0]][cur_pos[1]][cur_pos[2]]  +  1;// 加上初始位置
                    }
                }
                //当前位置之前没有访问过，则加入当前位置
                if (distance[next_r][next_c][cur_state] == -1) {
                    distance[next_r][next_c][cur_state]=                    =
distance[cur_pos[0]][cur_pos[1]][cur_pos[2]] + 1;//回去也是+，通过状态更新返回去，30*30*64个，状态改变了，没有访问，回退回去了。
                    q.push({ next_r ,next_c ,cur_state });
                }
            }
        }
    }
    return -1;
  }
};
```

作者：wpl-w
链接：https://leetcode-cn.com/problems/shortest-path-to-get-all-keys/solution/bfszhuang-tai-ya-suo-by-wpl-w-mi0u/
来源：力扣（LeetCode）

44. 通配符匹配（问号只能匹配任意字符，*可以匹配任何字符串（空的）以pattern 的字符作为匹配对象，如果是问号，匹配，如果是字母，两者都相等，向前走，如果是星号，分为不匹配和匹

配多个）

难度困难 649

给定一个字符串 (*s*) 和一个字符模式 (*p*) ，实现一个支持 '*?*' 和 '***' 的通配符匹配。

'?' 可以匹配任何单个字符。

'*' 可以匹配任意字符串（包括空字符串）。

两个字符串**完全匹配**才算匹配成功。

说明:

- *s* 可能为空，且只包含从 *a-z* 的小写字母。
- *p* 可能为空，且只包含从 *a-z* 的小写字母，以及字符 *?* 和 ***。

示例 1:

输入:

s = "aa"

p = "a"**输出:** false **解释:** "a" 无法匹配 "aa" 整个字符串。

示例 2:

输入:

s = "aa"

p = "*"**输出:** true **解释:** '*' 可以匹配任意字符串。

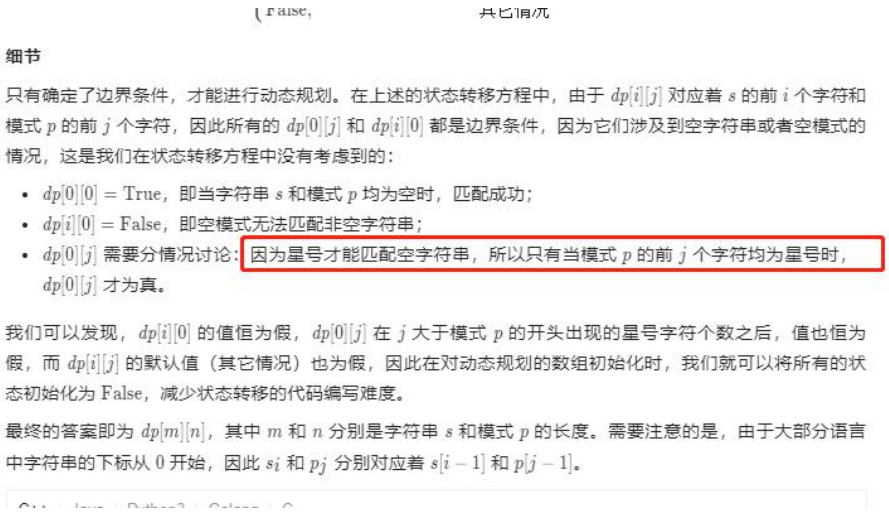
示例 3:

输入：

s = "cb"

p = "?a"输出：false 解释：'?' 可以匹配 'c'，但第二个 'a' 无法匹配 'b'。

我们用 dp[i][j] 表示字符串 s 的前 i 个字符和模式 p 的前 j 个字符是否能匹配//前 i 个字符，索引就是 i-1



细节

只有确定了边界条件，才能进行动态规划。在上述的状态转移方程中，由于 $dp[i][j]$ 对应着 s 的前 i 个字符和模式 p 的前 j 个字符，因此所有的 $dp[0][j]$ 和 $dp[i][0]$ 都是边界条件，因为它们涉及到空字符串或者空模式的情况，这是我们在状态转移方程中没有考虑到的：

- $dp[0][0] = \text{True}$，即当字符串 s 和模式 p 均为空时，匹配成功；
- $dp[i][0] = \text{False}$，即空模式无法匹配非空字符串；
- $dp[0][j]$ 需要分情况讨论：因为星号才能匹配空字符串，所以只有当模式 p 的前 j 个字符均为星号时，$dp[0][j]$ 才为真。

我们可以发现，$dp[i][0]$ 的值恒为假，$dp[0][j]$ 在 j 大于模式 p 的开头出现的星号字符个数之后，值也恒为假，而 $dp[i][j]$ 的默认值（其它情况）也为假，因此在对动态规划的数组初始化时，我们就可以将所有的状态初始化为 False，减少状态转移的代码编写难度。

最终的答案即为 $dp[m][n]$，其中 m 和 n 分别是字符串 s 和模式 p 的长度。需要注意的是，由于大部分语言中字符串的下标从 0 开始，因此 s_i 和 p_j 分别对应着 $s[i-1]$ 和 $p[j-1]$。

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.size();//i 是字符串
        int n = p.size();//j 是模式
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));
        dp[0][0] = true;//都是空串匹配成功
        for (int i = 1; i <= n; ++i) {
            if (p[i - 1] == '*') {
                dp[0][i] = true;//当他的前面是星号，才能匹配空串，一直是星号
            }
            else {
                break;
            }
        }
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (p[j - 1] == '*') {
                    dp[i][j] = dp[i][j - 1] | dp[i - 1][j];//星号，匹配 0 次或者多次，状态转移

                }
```

```
                else if (p[j - 1] == '?' || s[i - 1] == p[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];//当前的等于之前的。
                }
            }
        }
        return dp[m][n];
    }
};
```

方法一：动态规划
思路与算法

在给定的模式 pp 中，只会有三种类型的字符出现：

小写字母 a-za−z，可以匹配对应的一个小写字母；

问号 ??，可以匹配任意一个小写字母；

星号 ** ，可以匹配任意字符串，可以为空，也就是匹配零或任意多个小写字母。

其中「小写字母」和「问号」的匹配是确定的，而「星号」的匹配是不确定的，因此我们需要枚举所有的匹配情况。为了减少重复枚举，我们可以使用动态规划来解决本题。

我们用 dp[i][j] **表示字符串 s 的前 i 个字符和模式 p 的前 j 个字符是否能匹配**。在进行状只有确定了边界条件，才能进行动态规划。在上述的状态转移方程中，由于 \textit{dp}[i][j]dp[i][j] 对应着 ss 的前 ii 个字符和模式 pp 的前 jj 个字符，因此所有的 \textit{dp}[0][j]dp[0][j] 和 \textit{dp}[i][0]dp[i][0] 都是边界条件，因为它们涉及到空字符串或者空模式的情况，这是我们在状态转移方程中没有考虑到的：
初始条件

\textit{dp}[0][0] = \text{True}**dp[0][0]=True，即当字符串 ss 和模式 pp 均为空时，匹配成功；**

\textit{dp}[i][0] = \text{False}**dp[i][0]=False，即空模式无法匹配非空字符串；**

\textit{dp}[0][j]dp[0][j] 需要分情况讨论：**因为星号才能匹配空字符串，所以只有当模式 pp 的前 jj 个字符均为星号时，\textit{dp}[0][j]dp[0][j] 才为真。**

我们可以发现，\textit{dp}[i][0]dp[i][0] 的值恒为假，\textit{dp}[0][j]dp[0][j] 在 jj 大于模式 pp 的开头出现的星号字符个数之后，值也恒为假，而 \textit{dp}[i][j]dp[i][j] 的默认值（其它情况）也为假，因此在对动态规划的数组初始化时，我们就可以将所有的状态初始化为 \text{False}False，减少状态转移的代码编写难度。

最终的答案即为 \textit{dp}[m][n]dp[m][n]，其中 mm 和 nn 分别是字符串 ss 和模式 pp 的长度。需要注意的是，由于大部分语言中字符串的下标从 00 开始，因此 s_is

i

和 p_jp
j

分别对应着 s[i-1]s[i−1] 和 p[j-1]p[j−1]。

- 如果 p_j 是星号，那么同样对 s_i 没有任何要求，但是星号可以匹配零或任意多个小写字母，因此状态转移方程分为两种情况，即使用或不使用这个星号：

$$dp[i][j] = dp[i][j-1] \vee dp[i-1][j]$$

其中 $\vee$ 表示逻辑或运算。如果我们不使用这个星号，那么就会从 $dp[i][j-1]$ 转移而来；如果我们使用这个星号，那么就会从 $dp[i-1][j]$ 转移而来。

最终的状态转移方程如下：

$$dp[i][j] = \begin{cases} (s_i \text{ 与 } p_j \text{ 相同}) \wedge dp[i-1][j-1], & p_j \text{ 是小写字母} \\ dp[i-1][j-1], & p_j \text{ 是问号} \\ dp[i][j-1] \vee dp[i-1][j], & p_j \text{ 是星号} \end{cases}$$

我们也可以将前两种转移进行归纳：

$$dp[i][j] = \begin{cases} dp[i-1][j-1], & s_i \text{ 与 } p_j \text{ 相同或者 } p_j \text{ 是问号} \\ dp[i][j-1] \vee dp[i-1][j], & p_j \text{ 是星号} \\ \text{False}, & \text{其它情况} \end{cases}$$

细节

只有确定了边界条件，才能进行动态规划。在上述的状态转移方程中，由于 $dp[i][j]$ 对应着 s 的前 i 个字符和模式 p 的前 j 个字符，因此所有的 $dp[0][j]$ 和 $dp[i][0]$ 都是边界条件，因为它们涉及到空字符串或者空模式的情况，这是我们在状态转移方程中没有考虑到的：

- $dp[0][0] = \text{True}$，即当字符串 s 和模式 p 均为空时，匹配成功；
- $dp[i][0] = \text{False}$，即空模式无法匹配非空字符串；
- $dp[0][j]$ 需要分情况讨论：因为星号才能匹配空字符串，所以只有当模式 p 的前 j 个字符均为星号时，$dp[0][j]$ 才为真。

我们可以发现，$dp[i][0]$ 的值恒为假，$dp[0][j]$ 在 j 大于模式 p 的开头出现的星号字符个数之后，值也恒为假，而 $dp[i][j]$ 的默认值（其它情况）也为假，因此在对动态规划的数组初始化时，我们就可以将所有的状态初始化为 False，减少状态转移的代码编写难度。

最终的答案即为 $dp[m][n]$，其中 m 和 n 分别是字符串 s 和模式 p 的长度。需要注意的是，由于大部分语言中字符串的下标从 0 开始，因此 s_i 和 p_j 分别对应着 $s[i-1]$ 和 $p[j-1]$。

440. 字典序的第 K 小数字（找到需要经过的 step，当 cur<=n+1 停止，step>=k，肯定在里面需要展开 cur*10，k-step，否则 cur+1，前进一个节点）

难度困难 202

给定整数 n 和 k，找到 1 到 n 中字典序第 k 小的数字。

注意：1 ≤ k ≤ n ≤ 109。

示例 ：

输入：

```
n: 13   k: 2
```

输出：

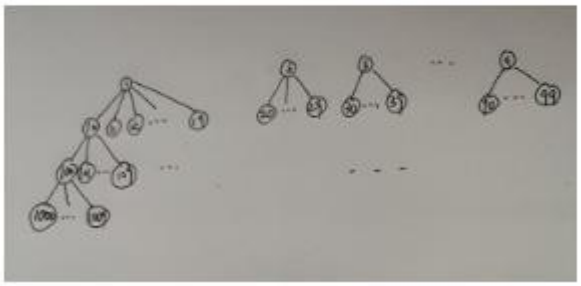
```
10
```

解释：

字典序的排列是 `[1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9]`，所以第二小的数字是 `10`。

思路：

建立一个十叉树，如下图所示：



感谢大佬手把手的教学！补充一下 cpp 实现，注意 int 会溢出的问题

我们可以从 1 到 100，慢慢地将数字一个一个地插入到它们该插入的地方，以此来查找其规律之所在。

从 1 开始，1，2，3，4，5，6，7，8，9

从 10 开始，1，10…19，2，3，4，5，6，7，8，9

从 20 开始，1，10…19，2，20…29，3，4，5，6，7，8，9

以此类推，所有的两位数，都插入了到与他们 10 位数相等的个位数后面。

再让我们来看看百位数如何插入

从 100 开始，1，10,100-109,11,110-119,12,120-129…2，21-29…

从 200 开始，1，10,100-109…2,20,200-209，21,210-219…3，31-39…

以此类推，**所有的三位数，都插入了到与他们百位至十位数相等的十位数后面**。

这里我们就已经能得到整个数组的排列规律了：整个数组可以构成一棵树，树的先序遍历结果就是数组字典序排列顺序。

(下面只以 1 为例)


```
class Solution {public:
    long getCount(long prefix, long n) {
        long cur = prefix;
        long next = cur + 1;
        long count = 0;
        while(cur <= n) {
            count += min(n+1, next) - cur;//同一层有几个
            cur *= 10;//下一个节点
            next *= 10;//下下个节点
```

```
        }
        return count;//返回的是步数
    }

    int findKthNumber(int n, int k) {
        long p = 1;
        long prefix = 1;
        while(p < k) {
            long count = getCount(prefix, n);//当前前缀几个数
            if (p + count > k) {
                /// 说明第 k 个数，在这个前缀范围里面
                prefix *= 10;
                p++;
            } else if (p+count <= k) {
                /// 说明第 k 个数，不在这个前缀范围里面，前缀需要扩大+1
                prefix++;
                p += count;
            }
        }
        return static_cast<int>(prefix);
    }
};
```

解题思路：
乍一看这一题貌似毫无头绪，什么是字典序？如何定位这个数？没错，刚接触这个题目的时候，我的脑筋里也是一团乱麻。

但是我觉得作为一个拥有聪明才智的程序员来说，最重要的能力就是迅速抽象问题、拆解问题的能力。经过一段时间的思考，我的脑筋里还是没有答案。

哈哈。

言归正传，我们来分析一下这个问题。

首先，什么是字典序？

什么是字典序？
简而言之，就是根据数字的前缀进行排序，

比如 10 < 9，因为 10 的前缀是 1，比 9 小。

再比如 112 < 12，因为 112 的前缀 **11 小于 12**。

这样排序下来，会跟平常的升序排序会有非常大的不同。先给你一个直观的感受，一个数乘 10，或者加 1，哪个大？可能你会吃惊，后者会更大。

但其实掌握它的本质之后，你一点都不会吃惊。

问题建模：
画一个图你就懂了。

十叉树

每一个节点都拥有 10 个孩子节点，因为作为一个前缀 ，它后面可以接 0~9 这十个数字。而且你可以非常容易地发现，整个字典序排列也就是对十叉树进行先序遍历。**1, 10, 100, 101, ... 11, 110 ...**

回到题目的意思，我们需要找到排在第 k 位的数。找到他的排位，需要搞清楚三件事情:

怎么确定一个**前缀下所有子节点的个数**？
如果第 k 个数在当前的前缀下，怎么继续往下面的子节点找？
如果第 k 个数不在当前的前缀，**即当前的前缀比较小，如何扩大前缀**，增大寻找的范围？
接下来 ，我们一一拆解这些问题。

理顺思路：
1. 确定指定前缀下所有子节点数
现在的任务就是给定一个前缀，返回下面子节点总数。

我们现在的思路就是用下一个前缀的起点减去当前前缀的起点，那么就是当前前缀下的所有子节点数总

作者：user7056K
链接：https://leetcode-cn.com/problems/k-th-smallest-in-lexicographical-order/solution/ben-ti-shi-sha

ng-zui-wan-zheng-ju-ti-de-shou-mo-sh/
来源：力扣（LeetCode）

思路:
十叉树，用题目的测试用例来举例子。

我们求字典序第 k 个就是上图前序遍历访问的第 k 节点！但是不需要用前序遍历，如果我们能通过数学方法求出**节点 1 和节点 2 之间需要走几步，减少很多没必要的移动**。

其实只需要按层节点个数计算即可，图中节点 1 和节点 2 在第二层，因为 n = 13，节点 1 可以移动到节点 2（同一层）所以**在第二层需要移动 1 步。**

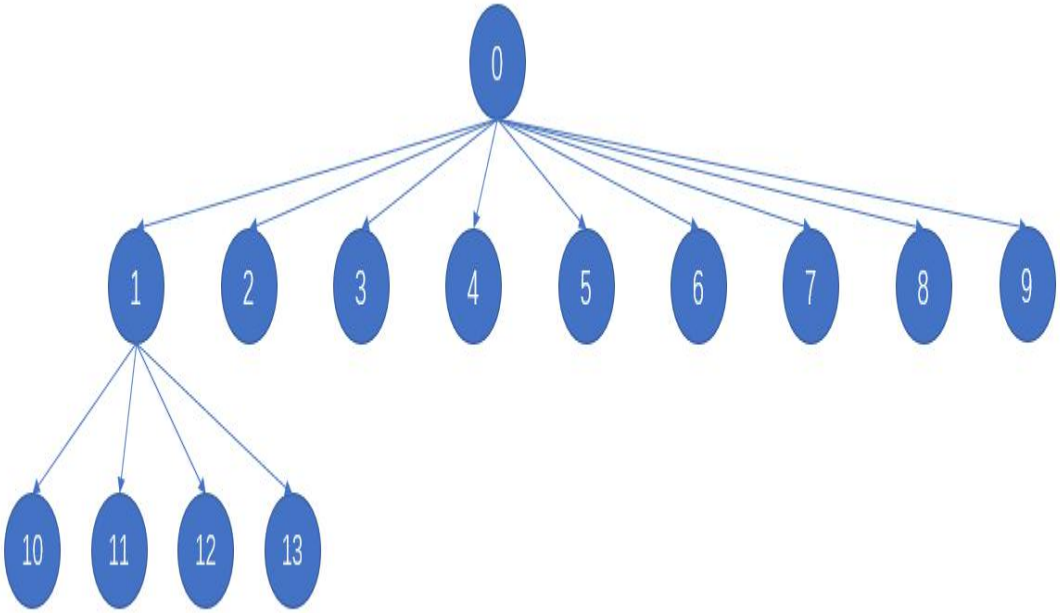
第三层，移动个数就是 (13 - 10 + 1) = 4 （min（13 + 1， 20） - 10）

所以节点 1 到节点 2 需要移动 1 + 4 = 5 步

当移动步数小于等于 k，说明需要向右节点移动，图中就是节点 1 移动到节点 2。

当移动步数大于 k，说明目标值在节点 1 和节点 2 之间，我们要向下移动！即从节点 1 移动到节点 10。

十叉树，用题目的测试用例来举例子。



我们求字典序第 k 个就是上图前序遍历访问的第 k 节点！但是不需要用前序遍历，如果我们能通过数学方法求出**节点 1 和节点 2** 之间需要走几步，减少很多没必要的移动。

其实只需要**按层节点个数计算**即可，图中节点 1 和节点 2 在第二层，因为 n = 13，节点 1 可以移动到节点 2（同一层）所以在第二层需要移动 **1 步**。

第三层，移动个数就是 (13 - 10 + 1) = 4 `（min（13 + 1， 20） - 10）`

所以节点 1 到节点 2 需要移动 1 + 4 = 5 步，**包括自身所以 13+1**

当移动步数**小于等于 k**，说明需要向**右节点移动**，图中就是节点 1 移动到节点 2。

当**移动步数大于 k**，说明目标值在节点 1 和节点 2 之间，我们要向下移动！即从节点 1 移动到节点 10。

代码：

```
class Solution:
    def findKthNumber(self, n: int, k: int) -> int:

        def cal_steps(n, n1, n2):
            step = 0
            while n1 <= n:
                step += min(n2, n + 1) - n1//n+1  自身算一个，n2 也取不到的
                n1 *= 10
                n2 *= 10
            return step

        cur = 1
        k -= 1//自身 cur=1，算一步

        while k > 0:
            steps = cal_steps(n, cur, cur + 1)
            if steps <= k://假设 23,n1=100 不动，当时就 21,需要往前走
                k -= steps//减去前面的 step，还是这一层
                cur += 1
            else:
                k -= 1//大的话，在里面，要往下面的节点找，减去父节点
                cur *= 10

        return cur
```

作者：powcai
链接：https://leetcode-cn.com/problems/k-th-smallest-in-lexicographical-order/solution/shi-cha-shu-by-powcai/
来源：力扣（LeetCode）

3）那么，难点就变成了 计算出同一层两个相邻的节点的子节点的个数，也就是代码中的 steps

3.1）当前节点为 curr （从 curr = 1 开始），则同一层的下一个节点为 curr+1;

3.2)计算节点 curr 到节点 curr+1 之间的子节点个数 steps

3.2.1)如果子节点个数 大于 k，说明第 k 小的树一定在子节点中，

继续向下一层寻找：curr *=10;

k -= 1;（原因：向下一层寻找，肯定要减少前面的父节点，即 在上一层中的第 k 个数，在下一层中是第 k-1 个数）

3.2.2)如果子节点个数 小于或者等于 k，说明第 k 小的树不在子节点中，

继续向同一层下一个节点寻找：curr +=1;

k -= steps;（原因：向下一层寻找，肯定要减少前面的所有的字节点）

以此类推，直到 k 为 0 推出循环，此时 cur 即为所求。

————————————————————

LeetCode#440 字典序的第 k 小数字

Android_ZzT 关注
2019.08.29 17:41:13 字数 359 阅读 447

input

- n:数字大小
- k:第 k 个小的数

example

```
input: (13,2)
output: 10
reason: [1, 10, 11, 12, 13, 2, 3, 4, 5, 6, 7, 8, 9]， 10 是第 2 小的数字
```

思路

- 理解数字的字典序，可以理解为一颗十叉树，根为 1，子节点为 10-19，根为 2，子节点为 20-29
- 输入 k，可理解为要在数组中走的步数
- 定义 5 个变量，result = 1 最终结果；lastSteps = k -1 表示剩余步数；left = result 左节点;right = result+1 右节点; count = 0 记录需要跨过的节点数
- 先寻找最终走到终点的范围在哪，也就是找到不满的十叉树的根节点是谁
- 当 left <= n 时，证明最终位置在当前节点的右边或者下边
- 如果 right 比 n + 1 小，left right 各向下一层继续找，同时累计 count += min(right, n+1) - left
- 直到 left 大于 n，将累加的节点和当前剩余步数相比较
- 如果 lastSteps > count，证明 **left 的子节点能全部走完，所以需要向右走一步** lastSteps -= count; result += 1;
- 如果 lastSteps <= count，证明 l**eft 的子节点已经足够走完剩余的步数了，这时向下走一步，lastSteps--; result*=10;**
- 当剩余步数为 0 时，result 就是最终结果

代码

```
class Solution {

    public void int findKthNumber(int n, int k) {

        int result = 1;

        int lastSteps = k - 1; //默认走了一步，走到第一个数

        while (lastSteps > 0) {

            long left = result; //用 long 类型，防止 n 达到 2 的 32 次方附近时，left * 10 会溢出

            long right = result + 1;

            int count = 0;

            while(left <= n) { //向下或向右探索，计算需要跨过的节点数
```

```
                count += Math.min(right, n+1) - left;

                left *= 10;

                right *= 10;

            }



            if (count > lastSteps) { //节点太多，跨不过去，只能向下走一层

                lastSteps --;

                result *= 10;

            } else { //节点不足，当前剩余步数足够走完全部节点，直接跨过这个节点以及他的子节点，向右走一步

                lastSteps -= count;

                result ++;

            }

        }

        return result;

    }



}
```

不同的子序列（定义）

我们令 `dp[i][j]`代表 `s` 的前 `j` 个字符中有多少个 `t[:i]`的子序列，我们写出状态转移方程，

当 `s[j]=t[i]`时：

dp[i][j]=dp[i-1][j-1]+dp[i][j-1]//可以往前，或者往前找后面的子序列，边界条件为 0

当 s[j]！=t[i]时：

今天的每日一题，Leetcode 第 115 题：不同的子序列。

题目描述

115. 不同的子序列

难度 困难 439 收藏 分享 切换为英文 接收动态 反馈

给定一个字符串 `s` 和一个字符串 `t` ，计算在 `s` 的子序列中 `t` 出现的个数。

字符串的一个 **子序列** 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，`"ACE"` 而 `"AEC"` 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

```
输入：s = "rabbbit", t = "rabbit"
输出：3
解释：
如下图所示，有 3 种可以从 s 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)
rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
```

示例 2：

```
输入：s = "babgbag", t = "bag"
输出：5
解释：
如下图所示，有 5 种可以从 s 中得到 "bag" 的方案。
(上箭头符号 ^ 表示选取的字母)
babgbag
^^ ^
babgbag
^^    ^
babgbag
^    ^^
babgbag
  ^  ^^
babgbag
    ^^^
```

题目分析

这是一道 hard 的题目，像这类子串匹配类型的问题基本都是用「动态

规划」进行求解。

我们令 `s` 的长度为 `lens`，`t` 的长度为 `lent`，我们知道当 `lens` 大于 `lent` 的话，肯定是不存在这样的子序列的，直接返回 0。

我们首先定义一个二维的动态规划数组 `dp[lent][lens]`，我们令 `dp[i][j]` 代表 `s` 的前 `j` 个字符中有多少个 `t[:i]`的子序列，我们写出状态转移方程，

当 `s[j]=t[i]`时：

`dp[i][j]=dp[i-1][j-1]+dp[i][j-1]`

当 `s[j]! =t[i]`时：

`dp[i][j]=dp[i][j-1]`。

对于这个状态转移方程，我自己是通过模拟了几个样例总结出来的，写下自己的总结过程和大家分享

给定例子，`s="rabc"`, `t="rab"`，当 `i=2,j=3` 时（为了和代码一致，从 0 开始计数），`t[2]="b"`, `s[3]="c"`， 因为 `s[3]! =t[2]`，所以有没有 `c` 这个字符对于 `s[:3]`中子序列 `t[:2]`出现了多少次没有影响，也就是说 `rabc` 中出现了多少次子序列 `rab` 和 `rab` 中出现了多少次子序列 `rab` 是等价的，那么 `dp[2][3]=dp[2][3-1]`。

对于另外一个例子，`s="rabb"`, `t="rab"`,当 `i=2, j=3` 时，`t[2]="b"`,`s[3]="b"`， 因为 `s[3]=t[2]`，所以就不能完全丢掉 `s[3]`这个字符。那么也就存在两种情况

- 当 `s[3]`这个字符正好用来匹配时，`"rabb"`中存在多少个子串”`rab`“等价于`"rab"`中存在多少个子串“`ra`”。**也就是 `dp[i-1][j-1];`**

- 当不使用 `s[3]`这个字符的话，`"rabb"`中存在多少个子串”`rab`“等价于`"rab"`中存在多少个子串“`rab`”。也就是 `dp[i][j-1]`。

合并这两种情况，我们可以得到 `dp[i][j]=dp[i][j-1]+dp[i-1][j-1]`。

题目代码

我们给出 C++和 python 的代码

```
//C++
class Solution {
public:
    int numDistinct(string s, string t) {
        int lens = s.length();
        int lent = t.length();
        if (lens<lent || lens==0 || lent==0) return 0;
        vector<vector<double>> dp(lent+1, vector<double>(lens+1,0));
        for(int i=0;i<dp[0].size();i++) dp[0][i]=1;
        for(int i=1;i<dp.size();i++)dp[i][0]=0;
        s = " "+s;
        t = " "+t;
        for(int i=1; i<t.length();i++){
            for (int j = 1;j<s.length();j++){
                char chart = t[i];
                char chars = s[j];
                if (chars==chart){
                    dp[i][j]=dp[i-1][j-1]+dp[i][j-1];
                }
                else{
                    dp[i][j]=dp[i][j-1];
                }
            }
        }
        return dp[t.length()-1][s.length()-1];


    }
};
```

剑指 Offer 51. 数组中的逆序对

难度困难 376

在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

示例 1:

输入: [7,5,6,4]**输出:** 5

方法一：归并排序
预备知识

「归并排序」是分治思想的典型应用，它包含这样三个步骤：

分解： 待排序的区间为 [l, r][l,r]，令 我们把 [l, r][l,r] 分成 [l, m][l,m] 和 [m + 1, r][m+1,r]
解决： 使用归并排序递归地排序两个子序列
3： 把两个已经排好序的子序列 [l, m][l,m] 和 [m + 1, r][m+1,r] 合并起来
在待排序序列长度为 11 的时候，**递归开始「回升」，因为我们默认长度为 11 的序列是排好序的。**

思路

那么求逆序对和归并排序又有什么关系呢？关键就在于「归并」当中「并」的过程。我们通过一个实例来看看。假设我们有两个已排序的序列等待合并，分别是 L = \{ 8, 12, 16, 22, 100 \}L={8,12,16,22,100} 和 R = \{ 9, 26, 55, 64, 91 \}R={9,26,55,64,91}。一开始我们用指针 lPtr = 0 指向 LL 的首部，rPtr = 0 指向 RR 的头部。记已经合并好的部分为 MM。

```
L = [8, 12, 16, 22, 100]     R = [9, 26, 55, 64, 91]    M = []
       |                              |
    lPtr                           rPtr
```

我们发现 lPtr 指向的元素小于 rPtr 指向的元素，于是把 lPtr 指向的元素放入答案，并把 lPtr 后移一位。

```
L = [8, 12, 16, 22, 100]     R = [9, 26, 55, 64, 91]    M = [8]
          |                            |
       lPtr                         rPtr
```

这个时候我们把左边的 88 加入了答案，我们发现右边没有数比 88 小，所以 88 对逆序对总数的「贡献」为 00。

接着我们继续合并，把 99 加入了答案，此时 lPtr 指向 1212，rPtr 指向 2626。

L = [8, 12, 16, 22, 100] R = [9, 26, 55, 64, 91] M = [8, 9]
          | |
        lPtr   rPtr

此时 lPtr 比 rPtr 小，把 lPtr 对应的数加入答案，并考虑它**对逆序对总数的贡献为 rPtr** 相对 RR 首位置的偏移 11（即右边只有一个数比 1212 小，所以只有它和 1212 构成逆序对），以此类推。

我们发现用这种「算贡献」的思想在合并的过程中计算逆序对的数量的时候，**只在 lPtr 右移的时候计算，是基于这样的事实：当前 lPtr 指向的数字比 rPtr 小，但是比 RR 中 [0 ... rPtr - 1] 的其他数字大，[0 ... rPtr - 1] 的其他数字本应当排在 lPtr 对应数字的左边，但是它排在了右边，所以这里就贡献了 rPtr 个逆序对。**

利用这个思路，我们可以写出如下代码。

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/shu-zu-zhong-de-ni-xu-dui-by-leetcode-solution/
来源：力扣（LeetCode）

copy 函数和 vector 类

cstswanghua 2017-09-15 15:31:01 365 收藏
分类专栏： C++学习笔记 文章标签： 函数

算法函数 copy。该算法主要用于容器之间元素的拷贝，即将迭代器区间[first，last)的元素复制到由复制目标 result **给定的区间[result，result+(last-first))中//后续取不到**

其函数为 copy(_first,_last,集合中拷贝的起始地址)，其中值得注意的是_last 这个参数，它表示我们要拷贝的集合的末端地址再+1

```
#include<algorithm>
#include<iostream>
#include<iterator>
#include<string>
```

```
#include<vector>
using namespace std;
 int myarr[]={1,2,3,4,5};

    vector<int> myvector;
    vector<int>::iterator myindex;
//我们要把 myarr 数组里的数拷贝到集合 myvector 里，


//首先定义集合 myvector 的长度：

myvector.resize(5);//不定义长度拷贝时会崩溃
//接下来写 copy 语句：


    copy ( myarr, myarr+5, myvector.begin() ); //在这里， _first 是 myarr 也就是&myarr[0]，注意_last 不是 myarr+4 而是+5

//注意 vector 的方法里，begin 是第一个元素的地址，但 end 是最后一个元素地址+1

//也就是说下面 2 个地址是一样的

myindex=myvector.end()-1;//这个实际是 vector 集合最后一个元素的地址

//那么&*myindex==&myvector[4]
```

————————————————————————

如果要把一个序列（sequence）拷贝到一个容器（container）中去，通常用 std::copy 算法，代码如下：

```
std::copy(start, end, std::back_inserter(container));
```

这里，start 和 end 是输入序列（假设有 N 各元素）的迭代器（iterator），container 是一个**容器，该容器的接口包含函数 push_back。假设 container 开始是空的**，那么 copy 完毕后它就包含 N 个元素，并且顺序与原来队列中的元素顺序一样。标准库提供的 back_inserter 模板函数很方便，因为它为 container 返回一个 back_insert_iterator 迭代器，这样，复制的元素都被追加到 container 的末尾了。

现在假设 container 开始非空（例如：container 必须在循环中反复被使用好几次）。那么，要达到原来的目标，必须先调用 clear 函数然后才能插入新序列。这会导致旧的元素对象被析构，新添加进来的被构造。不仅如此，container 自身使用的动态内存也会被释放然后又创建，就像 list，map，set 的节点。某些 vector 的实现在调用 clear 的时候甚至会释放所有内存。

通常，考虑到在一个已有的元素上直接 copy 覆盖更高效。keyi 这样做：

std::copy(start, end, container.begin());

在这里你在 container 的头部执行了 copy－over（覆盖赋值）操作，但是，如果 container 的大小小于输入序列的长度 N 的话，这段代码会导致崩溃（crash）。

eg1:

```
    int a[3]={1,2,3};

    int b[3];
    std::copy(a,a+3,b);
    for(int j=0;j<3;j++)
       cout<< P>
```

eg2:

```
  vector temp(3);
 int a[3]={1,2,3};

 std::copy(a,a+3,&temp.front());
 cout<<<ENDL;
 for(int j=0;j
  cout<<<"   ?;
```

copy 只负责复制，不负责申请空间，所以复制前必须有足够的空间
原文链接：https://blog.csdn.net/cstswanghua/article/details/77992246

代码

```
class Solution {
public:
    int mergeSort(vector<int>& nums, vector<int>& tmp, int l, int r) {
        if (l >= r) {
            return 0;//一个数就没了
        }

        int mid = (l + r) / 2;
        int inv_count = mergeSort(nums, tmp, l, mid) + mergeSort(nums, tmp, mid + 1, r);//之前的
        int i = l, j = mid + 1, pos = l;
```

```
        while (i <= mid && j <= r) {//j 能取到最后一位了， r+1
            if (nums[i] <= nums[j]) {
                tmp[pos] = nums[i];
                ++i;
                inv_count += (j - (mid + 1));//当左边小于，右边，把右边的拿出来,偏移了几位（当前的 j 比 i 大啊。J 不算数，比较当前放进去的 i,j 左边的偏移都是） 找到小于 i 的数 }看 j 之前有几个
            else {
                tmp[pos] = nums[j];
                ++j;
            }
            ++pos;
        }
        for (int k = i; k <= mid; ++k) {
            tmp[pos++] = nums[k];
            inv_count += (j - (mid + 1));//左边还有，j 能取到最后一位，找到比当前 i 小的数 }
        for (int k = j; k <= r; ++k) {
            tmp[pos++] = nums[k];
        }
        copy(tmp.begin() + l, tmp.begin() + r + 1, nums.begin() + l);//将这个数据丢给，是 r+1, 往后面的。假设 l=0，就是 begin 了。
        return inv_count;
    }

    int reversePairs(vector<int>& nums) {
        int n = nums.size();
        vector<int> tmp(n);
        return mergeSort(nums, tmp, 0, n - 1);
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/shu-zu-zhong-de-ni-xu-dui-by-leetcode-solution/
来源：力扣（LeetCode）

435. 无重叠区间

难度中等 388
给定一个区间的集合，找到需要移除区间的**最小数量**，**使剩余区间互不重叠**。

注意:

1. 可以认为区间的终点总是大于它的起点。
2. 区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

示例 1:

输入: `[ [1,2], [2,3], [3,4], [1,3] ]`

输出: `1`

解释: 移除 `[1,3]` 后，剩下的区间没有重叠。

示例 2:

输入: `[ [1,2], [1,2], [1,2] ]`

输出: `2`

解释: 你需要移除两个 `[1,2]` 来使剩下的区间没有重叠。

示例 3:

输入: `[ [1,2], [2,3] ]`

输出: `0`

解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

方法一：动态规划
思路与算法

题目的要求等价于「选出最多数量的区间，使得它们互不重叠」。由于选出的区间互不重叠，因此我们可以将它们按照端点从小到大的顺序进行排序，并且无论我们按照左端点还是右端点进行排序，得到的结果都是唯一的。

C++JavaPython3GolangCJavaScript

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.empty()) {
            return 0;
        }

        sort(intervals.begin(), intervals.end(), [](const auto& u, const auto& v) {
            return u[0] < v[0];//左端点从小到大排序
        });

        int n = intervals.size();
        vector<int> f(n, 1);//定义是这个作为末尾的最长不重叠区间
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                if (intervals[j][1] <= intervals[i][0]) {
                    f[i] = max(f[i], f[j] + 1);//已经左端点排序了。因此只要第 jj 个区间的右端点 没有越过第 ii 个区间的左端点 l_il ，那么第 jj 个区间就与第 ii 个区间不重叠。

                }
            }
        }
        return n - *max_element(f.begin(), f.end());//减去最多的
    }
};
```

复杂度分析

时间复杂度：O(n^2)O(n
2
)，其中 nn 是区间的数量。我们需要 O(n \log n)O(nlogn) 的时间对所有的区间按照左端点进行升序排序，并且需要 O(n^2)O(n
2
) 的时间进行动态规划。由于前者在渐进意义下小于后者，因此总时间复杂度为 O(n^2)O(n
2
)。

注意到方法一本质上是一个「最长上升子序列」问题，因此我们可以将时间复杂度优化至 O(n \log n)O(nlogn)，具体可以参考「300. 最长递增子序列的官方题解」。

空间复杂度：O(n)O(n)，即为存储所有状态 f_if
i

需要的空间。

方法二：贪心
思路与算法

我们不妨想一想应该选择哪一个区间作为首个区间。

假设在某一种最优的选择方法中，[l_k, r_k][l
k

,r
k

] 是首个（即最左侧的）区间，那么它的左侧没有其它区间，右侧有若干个不重叠的区间。设想一下，如果此时存在一个区间 [l_j, r_j][l
j

,r
j

]，使得 r_j < r_kr
j

<r
k

，即区间 jj 的右端点在区间 kk 的左侧，那么我们将区间 kk 替换为区间 jj，其与剩余右侧被选择的区间仍然是不重叠的。而当我们将区间 kk 替换为区间 jj 后，就得到了另一种最优的选择方法。

我们可以不断地寻找右端点在首个区间右端点左侧的新区间，将首个区间替换成该区间。那么当我们无法替换时，首个区间就是所有可以选择的区间中右端点最小的那个区间。因此我们将所有区间按照右端点从小到大进行排序，那么排完序之后的首个区间，就是我们选择的首个区间。

如果有多个区间的右端点都同样最小怎么办？由于我们选择的是首个区间，因此在左侧不会有其它的区间，那么左端点在何处是不重要的，我们只要任意选择一个右端点最小的区间即可。

当确定了首个区间之后，所有与首个区间不重合的区间就组成了一个规模更小的子问题。由于我们已经在初始时将所有区间按照右端点排好序了，因此对于这个子问题，我们无需再次进行排序，只要找出其中与首个区间不重合并且右端点最小的区间即可。用相同的方法，我们可以依次确定后续的所有区间。

在实际的代码编写中，我们对按照右端点排好序的区间进行遍历，并且实时维护上一个选择

区间的右端点 \textit{right}right。如果当前遍历到的区间 [l_i, r_i][l
i

,r
i

] 与上一个区间不重合，即 l_i \geq \textit{right}l
i

≥right，那么我们就可以贪心地选择这个区间，并将 \textit{right}right 更新为 r_ir
i

。

代码

C++JavaPython3GolangCJavaScript

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.empty()) {
            return 0;
        }

        sort(intervals.begin(), intervals.end(), [](const auto& u, const auto& v) {
            return u[1] < v[1];//右边排序
        });

        int n = intervals.size();
        int right = intervals[0][1];
        int ans = 1;
        for (int i = 1; i < n; ++i) {//贪心
            if (intervals[i][0] >= right) {
                ++ans;//多要一支箭
                right = intervals[i][1];
            }
        }
        return n - ans;
    }
};
```

复杂度分析

时间复杂度：O(n \log n)O(nlogn)，其中 nn 是区间的数量。我们需要 O(n \log n)O(nlogn) 的

时间对所有的区间按照右端点进行升序排序，并且需要 O(n)O(n) 的时间进行遍历。由于前者在渐进意义下大于后者，因此总时间复杂度为 O(n \log n)O(nlogn)。

空间复杂度：O(\log n)O(logn)，即为排序需要使用的栈空间。

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/non-overlapping-intervals/solution/wu-zhong-die-qu-jian-by-leetcode-solutio-cpsb/
来源：力扣（LeetCode）


```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.empty()) {
            return 0;
        }

        sort(intervals.begin(), intervals.end(), [](const auto& u, const auto& v) {
            return u[0] < v[0];
        });

        int n = intervals.size();
        vector<int> f(n, 1);
        for (int i = 1; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                if (intervals[j][1] <= intervals[i][0]) {
                    f[i] = max(f[i], f[j] + 1);
                }
            }
        }
        return n - *max_element(f.begin(), f.end());
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/non-overlapping-intervals/solution/wu-zhong-die-qu-jian-by-leetcode-solutio-cpsb/
来源：力扣（LeetCode）

链接：
https://www.nowcoder.com/questionTerminal/247e8d0be6724b7baacc4029344

1fe76
来源：牛客网

小马车队

小马智行(Pony.ai)在广州南沙区有一支稳定运营的自动驾驶车队，可以将南沙区的地图看做一个二维的网格图，小马智行的广州 office 在(0, 0)位置。
公司现在有 n 台车，每天会按如下规则从围绕南沙区进行路测：
1. 初始 n 辆车都在公司。
2. 放眼整个南沙地图，每过一分钟， 若有一个网格的车数大于等于 8， 则这个网格同时会有 8 辆车分别前往上，下，左，右，左上，左下，右上，右下的网格，不停执行该步骤直到所有的车辆的位置都固定不变。
作为小马智行车辆控制中心的一员， 你需要监管车辆运营的情况， 你需要等到所有车辆的位置固定之后，进行 q 次抽样统计，每次需要统计出以(x1,y1)(x_1, y_1)(x1 ,y1)为左下角，以(x2,y2)(x_2, y_2)(x2 ,y2)为右上角的矩形范围内车辆的车辆的数目。

输入描述:
第一行为 n 和 q， 分别代表初始 office 内的车辆数和抽样的次数。
之后 q 行，每行包含 4 个变量 x1x_1x1 ，y1y_1y1 ，x2x_2x2 ，y2y_2y2 含义见题目描述。
1≤n≤105,1≤q≤105,x,y∈[-109，109]1 \leq n \leq 10^5, 1 \leq q \leq 10^5, x, y \in [-10^9， 10^9]1≤n≤105,1≤q≤105,x,y∈[-109，109]后， 进行 q 次抽样，每次查询以(x1,y1)(x_1, y_1)(x1 ,y1)为左下角，以(x2,y2)(x_2, y_2)(x2 ,y2)为右上角的矩形范围内车辆的数目。

输出描述:
输出 q 次抽样的结果，每次结果独占一行。
示例 1
输入
8 2
0 0 0 0
-1 -1 1 1
输出
0
8
说明
第 0 分钟所有车辆都在 office 处。

第 1 分钟及以后，8 辆车分别在(-1, -1)，(-1, 0)，(-1, 1)，(0, -1)，(0, 1)，(1, -1)，(1, 0)，(1, 1)这 8 个位置。

```
#include
#include
using namespace std;
const int ms = 128;
void print1(int m[ms][ms], int s = 8) {
	for(int i=-s; i<=s; i++) {
		for(int j=-s; j<=s; j++)
			if(abs(i)<=abs(j)) cout << m[abs(i)][abs(j)] << ' ';
			else cout << m[abs(j)][abs(i)] << ' ';
		cout << endl;
	}
	cout << "----------------" << endl;
}
void print2(int m[ms*2][ms*2], int s = 8) {
	for(int i=ms-s; i<=ms+s; i++) {
		for(int j=ms-s; j<=ms+s; j++)
			cout << m[i][j] << ' ';
		cout << endl;
	}
	cout << "----------------" << endl;
}
void weight(int bff[ms][ms], int x, int y, int f = 0) {
	if(x y) return;
	if(x==0 && y==0) bff[x][y] += 4;
	else if(x==0 && !(f==1)) bff[x][y] += 2;
	else if(x==y && !(f==2)) bff[x][y] += 2;
	else bff[x][y] += 1;
}
void trans(int bff[ms][ms], int x, int y) {
	if(x==0 && y==1) bff[x][y] -= 6;
	else if(y-x==1) bff[x][y] -= 7;
	else bff[x][y] -= 8;
	weight(bff, x-1, y-1, x==y?2:0);
	weight(bff, x-1, y);
	weight(bff, x-1, y+1);
	weight(bff, x, y-1, x==0?1:0);
	weight(bff, x, y+1, x==0?1:0);
	weight(bff, x+1, y-1);
	weight(bff, x+1, y);
	weight(bff, x+1, y+1, x==y?2:0);
}
void update(int m[ms][ms], bool & noUpdate) {
	static int s = 1;
	int bff[ms][ms];
```

```
        noUpdate = true;
        memset(bff, 0, sizeof(bff));
        if(m[0][0]>=8) {
                noUpdate = false;
                bff[0][0] -= 8; bff[0][1] ++;
                bff[1][1] ++;
        }
        for(int i=0; i < s; i++)
                for(int j=i; j < s; j++)
                if(m[i][j]>=8 && !(i==0 && j==0)) {
                        noUpdate = false;
                        trans(bff, i, j);
                }
        if(bff[0][s]>0) s++;
        for(int i=0; i< s; i++)
                for(int j=i; j < s; j++)
                m[i][j] += bff[i][j];
}
const int c = 100;
void dev(int & x, int offset = 0) {
        x += offset;
        if(x>c) x = c;
        else if(x<-c) x = -c;
        x += ms;
}
int main() {
        int m[ms][ms];
        int n, q;
        memset(m, 0, sizeof(m));
        cin >> n >> q;
        bool noUpdate = false;
        m[0][0] = n;
        while(!noUpdate) update(m, noUpdate);
        int mm[ms*2][ms*2];
        int sum[ms*2][ms*2];
        memset(mm, 0, sizeof(mm));
        memset(sum, 0, sizeof(sum));
        for(int i=0; i<= c; i++)
                for(int j=i; j <= c; j++) {
                        mm[ms+i][ms+j] = m[i][j];
                        mm[ms+i][ms-j] = m[i][j];
                        mm[ms-i][ms+j] = m[i][j];
                        mm[ms-i][ms-j] = m[i][j];
                        mm[ms+j][ms+i] = m[i][j];
```

```
                mm[ms+j][ms-i] = m[i][j];
                mm[ms-j][ms+i] = m[i][j];
                mm[ms-j][ms-i] = m[i][j];
            }
        for(int i=ms-c; i<= ms+c; i++)
            for(int j=ms-c; j <= ms+c; j++) {
                sum[i][j] = mm[i][j] + sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1];
            }
        //print2(mm);
        //print2(sum);
        while(q--) {
            int x1, y1, x2, y2;
            cin >> x1 >> y1 >> x2 >> y2;
            dev(x1,-1); dev(y1,-1); dev(x2); dev(y2);
            //cout << " " << x1 << " " << y1 << " " << x2 << " " << y2 << endl;
            //cout << " " << sum[x1][y1] << " " << sum[x2][y2] << " " << sum[x1][y2] << " " << sum[x2][y1] << endl;
            cout << sum[x1][y1] - sum[x1][y2] - sum[x2][y1] + sum[x2][y2] << endl;
        }
}
```

牛客 721590970 号

应注意 n 大小，最后有车格子组成**的图形大小是有限的且是正方形。**根据对称性，先算 1/8 小块的车流量，然后再映射到大的地图上。

安全区域（找正方形呗，先换成 01，每个数作为左上角呗，遍历，先遍历对角线，看看有几个，边长 k，就加起来。）

```
'''
给定一个 n*m 的二维格点地图，每个位置要么是字符'.'表示空地，要么是'@'表示有敌人在这里．规定给定一个 d(1 <= d <= min(m, n))，如果一个 d*d 的区域内没有任何敌人，则认为这片区域是安全的．
问给定的地图中有多少个这样安全的区域．
'''
```

```
## 二维前缀和，将原始的地图转换成 01 矩阵，然后 t_sum[i][j] 保存从（0,0） 到 （i，j）的矩形中所有元素的和
## 然后遍历二维矩阵 利用前缀和 计算以(i,j)作为左上角的 d*d 的区域内的元素和，如果是 0，说明是安全区+1
## 总体时间复杂度 O(N*M)def convert(data): # 转换成 01 矩阵    n=len(data)
    m=len(data[0])
    matrix=[[0]*m for i in range(n)]
    for i in range(n):
        for j in range(m):
            if data[i][j]=='@':
                matrix[i][j]=1
    return matrix

def func(matrix,d):
    n=len(matrix)
    m=len(matrix[0])

    t_sum=[[0]*m for i in range(n)]

    for j in range(m):
        t_sum[0][j]=t_sum[0][j-1]+matrix[0][j]

    for i in range(1,n):
        t_sum[i][0]=t_sum[i-1][0]+matrix[i][0]

    for i in range(1,n):
        for j in range(1,m):
            t_sum[i][j]=t_sum[i][j-1]+t_sum[i-1][j]-t_sum[i-1][j-1]+matrix[i][j]

    result=1
    for i in range(n):
        if i+d>=n:
            break
        for j in range(m):
            if j+d>=m:
                break
            ni=i+d
            nj=j+d
            if i==0 and j==0:
                        tmp=t_sum[ni][nj]
            elif i==0:
                tmp=t_sum[ni][nj]-t_sum[ni][j-1]
            elif j==0:
                tmp=t_sum[ni][nj] -t_sum[i-1][nj]
            else:
             tmp=t_sum[ni][nj]-t_sum[ni][j-1]-t_sum[i-1][nj]+t_sum[i-1][j-1]
```

```
            if tmp==0:
                result+=1
    return result
```

三个点，其他点都不在里面（先构造三角形，如果点在里面，随便替换掉，其中一个点，叉乘同号，判定在里面）

```
'''
二维平面上有 N 个点，每个点用二维坐标表示，找到三个点让他们组成三角形，使得其他所有的点都不在三角形内部，返回这样的三个点的坐标。
'''
###### 想了个 Nlog（N）的，先对所有的点按照 x 轴坐标排序，然后连续的三个点就有可能是满足条件的点。但是需要判断一下多个点是否共线这种情况。def func(points):
    points.sort(key=lambda x:x[0])
    for i in range(len(points)-2):
        j=i+1
        k=i+2
        xs=[p[0] for p in points[i:i+3]]
        ys=[p[1] for p in points[i:i+3]]
        if len(set(xs))==1:
            continue
        elif len(xs)==2:
            return points[i:i+3]
        if len(set(ys))==1 or (ys[1]-ys[0] )/(xs[1]-xs[0]) == (ys[2]-ys[0])/(xs[2]-xs[0]):
            continue
        else:
            return points[i:i+3]
    return []
# PS:面试官表示还有 O(N)的做法，首先选三个不共线的点组成原始的三角形，然后依次扫描其他的点，如果他们是在三角形的外部则直接跳过，否则的话该点替换掉其中的一个点，最后所有的点都判断完毕之后得到的三角形上的三个点也满足题目要求。
'''
```

众数最大（区间内的和+k>=区间最大值+数目，就移动 left,否则移动 right，要加也是先加更小的啊，顶多到最大的贪

心思想。众数的数目最大，而不是值最大，就是区间尽肯能的长）

```
给定一个包含 int 数据的数组，和一个整数 k。 每一次可以对数组中的一个数进行加一操作，最多可以做 k 次操作，k 可以不用完。
希望能够使数组中相同的数字数目最多（也就是众数的数目最大，而不是值最大），返回这个最大值

例如：
{1,2,4,4}  k=2 可以操作得到->{1,4,4,4}，因此结果为 3
'''
```

思路是先对数组排序，然后用一个左右指针表示的区间内的数字调整到区间内的最大值
如果区间内的和+k >= 区间最大值* 区间数目（爆了），则当前区间可以在不多于 k 次的操作下调整到相同的值，此时更新结果最大值并，左移 left 指针
否则的话左移 right 指针
直到左指针小于 0 时终止循环。或者 right 不可能大于最大长度

```
def func(nums,k):
    nums.sort()

    pre_sum = [nums[0]]
    for i in range(1, len(nums)):
        cur_sum = pre_sum[-1] + nums[i]
        pre_sum.append(cur_sum)

    left = len(nums) - 1//从最大开始滑动
    right = len(nums) - 1
    result = 1
    while left >= 0 and right >= 0:
        if right < result - 1://右边小于长度最大值，就直接退出了，肯定不会最大
            break
        t_sum = pre_sum[right] - pre_sum[left] + nums[left]  # why add nums[right]//前缀和，之间
 if t_sum + k >= (right - left + 1) * nums[right]://最大值右边
            result = max(result, (right - left + 1))
            left -= 1
        else:
            right -= 1

    return result
```

作者：业余选手キライ！
链接：
https://www.nowcoder.com/discuss/551909?type=0&order=0&pos=9&page=1&channel=-1&source_id=discuss_tag_nctrack
来源：牛客网

叉乘同号判断在三角形里面

叉乘（cross product）

相对于点乘，叉乘可能更有用吧。2 维空间中的叉乘是：V1(x1, y1) X V2(x2, y2) = x1y2 -y1x2

看起来像个标量，**事实上叉乘的结果是个向量，方向在 z 轴上**。上述结果是它的模。在二维空间里，让我们暂时忽略它的方向，将结果看成一个向量，那么这个结果类似于上述的点积，我们有：A x B = |A||B|Sin(θ)

然而角度 θ和上面点乘的角度有一点点不同，他是有正负的，是指从 A 到 B 的角度。另外还有一个有用的特征那就是叉积的绝对值就是 A 和 B 为两边说形成的**平行四边形的面积**。也就是 AB 所包围三角形面积的两倍。这个还是很显然的啦。在计算面积时，我们要经常用到叉积。

方向：a 向量与 b 向量的向量积的方向与这两个向量所在平面垂直，且遵守右手定则。（一个简单的确定满足“右手定则”的结果向量的方向的方法是这样的：若坐标系是满足右手定则的，当右手的四**指从 a 以不超过 180 度的转角转向 b 时，**竖起的大拇指指向是 **c** 的方向。）

也可以这样定义（等效）：

向量积**|c|=|a×b|=|a||b|**sin<**a，b**>

即 **c** 的长度在数值上等于以 **a，b，**夹角为θ组成的平行四边形的面积。

而 **c** 的方向垂直于 a 与 b 所决定的平面，**c** 的指向按右手定则从 a 转向 b 来确定。

*运算结果 **c** 是一个伪向量。这是因为在不同的坐标系中 **c** 可能不同。[1]

该算法和算法 2 类似，可以看作是对算法 2 的简化，也是用到向量的叉乘。假设三角形的三个点按照顺时针（或者逆时针）顺序是 A,B,C。对于某一点 P，求出三个向量 PA,PB,PC，然后计算以下三个叉乘（^表示叉

乘符号）：

t1 = PA^PB,

t2 = PB^PC,

t3 = PC^PA,

如果 t1，t2，t3 同号（同正或同负），那么 P 在三角形内部，否则在外部。

一面

项目 15min

．判断无向图是否为二叉树（树）。

思路：要判断一个图是否为树，首先要知道树的定义。
一棵树必须具备如下特性：

（1）是一个全连通图（所有节点相通）

（2）无回路

其中（2）等价于：（3）**图的边数=节点数-1**

因此我们可以利用特性（1）（2）或者（1）（3）来判断。

方法一：广度优先搜索。要判断连通性，广度优先搜索法是一个天然的选择，时间复杂度 O(n)，空间复杂度 O(n)。

```
public class Solution {
    public boolean validTree(int n, int[][] edges) {
        Map<Integer, Set<Integer>> graph = new HashMap<>();
        for(int i=0; i<edges.length; i++) {
            for(int j=0; j<2; j++) {
                Set<Integer> pairs = graph.get(edges[i][j]);
```

```
                    if (pairs == null) {
                        pairs = new HashSet<>();
                        graph.put(edges[i][j], pairs);
                    }
                    pairs.add(edges[i][1-j]);
                }
            }
            Set<Integer> visited = new HashSet<>();
            Set<Integer> current = new HashSet<>();
            visited.add(0);
            current.add(0);
            while (!current.isEmpty()) {
                Set<Integer> next = new HashSet<>();
                for(Integer node: current) {
                    Set<Integer> pairs = graph.get(node);
                    if (pairs == null) continue;
                    for(Integer pair: pairs) {
                        if (visited.contains(pair)) return false;
                        next.add(pair);
                        visited.add(pair);
                        graph.get(pair).remove(node);
                    }
                }
                current = next;
            }
            return visited.size() == n;
        }
}
```

简书代码

判断条件

1. n 个点若不存在 n-1 条边一定存在环
2. 由一个点可以访问到其它所有点，点的总数为 n

```
public class Solution {

    /**

     * @param n an integer
```

```
     * @param edges a list of undirected edges
     * @return true if it's a valid tree, or false
     */
    public boolean validTree(int n, int[][] edges) {
        if (n == 0) {
            return false;
        }
        // 判断图是否是树依据上述的第二个条件
        if (edges.length != n - 1) {
            return false;
        }

        // set 用于去重，不包含重复元素的集合
        Map<Integer, Set<Integer>> graph = initializeGraph(n, edges);

        // bfs
        // queue 里面存的是结点下标
        Queue<Integer> queue = new LinkedList<>();
        Set<Integer> hash = new HashSet<>();

        queue.offer(0);
        hash.add(0);     // hashset 没有 offer 用法
        // queue 结合 while 使用来保证遍历全部结点
```

```
        while (!queue.isEmpty()) {

            int node = queue.poll();

            // foreach 用法，neighbor 是变量名

            // graph.get(node) 对应的是一个集合

            for (Integer neighbor : graph.get(node)) {

                // hash 表用于去除重复结点，来保证队列中没有添加重复结点

                // 树只能从上往下遍历，但图没有方向，A 是 B 的相邻结点，B 也是 A 的相邻结点，所以要去重

                if (hash.contains(neighbor)) {

                    continue;

                }

                hash.add(neighbor);

                queue.offer(neighbor);

            }

        }



        // 当存在 n-1 条边且有 n 个结点连通时说明图是树

        return (hash.size() == n);

    }



    // 根据结点和边初始化一张图出来

    private Map<Integer, Set<Integer>> initializeGraph(int n, int[][] edges) {

        // set 的不包含重复元素特性特别重要，set 在后边代表的两点间建立边的关系
```

```java
// 若某一点重复加了一个点两次则证明出现了环，初始化必须保证无环，算法才有意义
Map<Integer, Set<Integer>> graph = new HashMap<>();
for (int i = 0; i < n; i++) {
    // hashset 用于存储不重复整型对象，
    // hashmap 中的 put 方法用于关联指定值与指定键，
    // 本行代码用于创建 n 个映射
    graph.put(i, new HashSet<Integer>());
}

// 注意此处不是 n，n 代表结点数
// i 循环的是边数，边数小于 n，若写成 n 则在 i = n - 1 时代码会卡住，程序超时
for (int i = 0; i < edges.length; i++) {
    int u = edges[i][0];
    int v = edges[i][1];
    graph.get(u).add(v);
    graph.get(v).add(u);
}
  // 在图中建立边的连接实际上就是建立两个整数间的不重复映射关系
  // get() 返回指定键映射的值,即 graph 代表 hashset 数组，
  // graph.get(v).add(u) 代表 hashset.add()
  // u 和 v 代表边的两个端点在 graph.get(u) 中 u,v  代表索引值 i，
  // u.add(v) 是指加一条 u 到 v 的边(graph 中下标为 u 的 set 加入
  // 一个值为 v 的元素)，把题目提供的输入数据中的边数组进行处理
```

```
        return graph;
    }
}
```

————————————————————

复制代码

```
bool Solve(int n, const vector<vector<int>>& edges) {
    if (n == 0) return true;

    vecotr<vector<int>> G(n);
    for (const auto& x : edges) {
            int& u = x[0], v = x[1];
        G[u].emplace_back(v);
        G[v].emplace_back(u);
    }

    vector<int> pre(n, 0);
    for (int i = 0; i < n; i++) {
            pre[i] = i;
    }

    function<int(int)> find = [&](int x) {
            return x == pre[x] ? x : pre[x] = find(pre[x]);
          };

    function<int(int, int)> unite = [&](int u, int v) {
            u = find(u); v = find(v);
        if (u == v) return 0;
        pre[u] = v;
        return 1;
    };

    bool ok = 1;

    function<void(int, int)> dfs = [&](int u, int p) {
        int tot = 0;
```

```
31          for (int i = 0; i < G[u].size(); i++) {
32                  int& v = G[u][i];
33              if (v == p) continue;
34              if (!unite(u, v)) {
35                      ok = 0;
36                  break;
37              }
38              tot++;
39              dfs(v, u);
40          }
41          if (tot > 2) {
42                  ok = 0;
43          }
44      };
45      dfs(0, -1);
46
47      int tot = 0;
48      for (int i = 0; i < n; i++) {
49              if (find(i) == i) tot++;
50      }
51
52      return ok && (tot == 1);
53}
```

二面

. n 个时间段，让你计算最多多少段时间相互不重叠，证明[算法](#)正确性？贪心 10min

复制代码

```
1 int Solve(vector<vector<int>>& seg) {
2          int n = seg.size();
3      if (n == 0) {
4              return 0;
5      }
6          sort(seg.begin(), seg.end(), [&](const vector<int>& a, const
7 vector<int>& b) {
8              if (a[1] == b[1]) return a[0] < b[0];
9          return a[1] < b[1];
10     });
11     int ret = 1;
```

```
    int ed = seg[0][1];
    for (int i = 1; i < n; i++) {
            if (seg[i][0] >= ed) {
                ed = seg[i][1];
            ret++;箭头
        }
    }
    return ret;
}
```

- n 个点（1E6），给 k 个点，求这 k 个点的 LCA？提到 bitset 维护状态，每次返回子树根 merge，后来发现没必要维护具体点的出现状态，所以维护在 k 个点中的点数就可以。 15min

复制代码

```
int Solve(int n, int root, const vecotr<vector<int>>& G, const vector<int>& query) {
        // G[i][j] = v: i 的第 j 条边是 v。 G: G[i][j] == G[j][i]
    int m = query.size();
    unordered_set<int> qset(query.begin(), query.end());
    int lca = root, ldep = -1;
    function<int(int, int, int)> dfs = [&](int u, int p, int dep) {
        int tot = 0;
            for (int i = 0; i < G[u].size(); i++) {
                int v = G[u][i];
            if (v == p) continue;
            int tmp = dfs(v, u, dep + 1);
            if (ldep != -1) return INT_MAX;
            tot += tmp;
        }
        tot += (qset.find(u) != qset.end());
        if (tot == m && dep > ldep) {
                ldep = dep;
            lca = u;
        }
        return tot;
    };
    dfs(root, -1, 0);
    return lca;
}
```

作者：业余选手キライ！
链接：
https://www.nowcoder.com/discuss/551909?type=0&order=0&pos=9&page=1&c

hannel=-1&source_id=discuss_tag_nctrack
来源：牛客网

102. 二叉树的层序遍历(标准的 BFS 啦，ret.push_back(vector <int> ());// 每次新增加一个空集，重要，增加空间，增加空间，push——back)

难度中等 830
给你一个二叉树，请你返回其按 **层序遍历** 得到的节点值。 （即逐层地，从左到右访问所有节点）。

示例：
二叉树：[3,9,20,null,null,15,7],

```
    3
   / \
  9  20
    /  \
   15   7
```

返回其层序遍历结果：

```
[
  [3],//第一层
  [9,20],
  [15,7]
]
```

通过次数 283,961
提交次数 442,135

```
class Solution {
public:
```

```
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector <vector <int>> ret;
        if (!root) {
            return ret;//空集就返回自身
        }

        queue <TreeNode*> q;//一个节点
        q.push(root);//将根节点放进去
        while (!q.empty()) {
            int currentLevelSize = q.size();
            ret.push_back(vector <int> ());//每次新增加一个空集，重要，增加空间
            for (int i = 1; i <= currentLevelSize; ++i) {
                auto node = q.front(); q.pop();
                ret.back().push_back(node->val);//将同一层的这个值，放进去
                if (node->left) q.push(node->left);//必须不为空
                if (node->right) q.push(node->right);
            }
        }

        return ret;
    }
};
```

作者：LeetCode-Solution
链接：https://leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/er-cha-shu-de-ceng-xu-bian-li-by-leetcode-solution/
来源：力扣（LeetCode）

移动障碍物（图我以为 dfs 呢啦啦）

一道题：N*M 的矩阵，有障碍物。给起点和终点，问从起点到终点最少需要移除多少块障碍物。50min

对空白缩点，**重新建图跑堆优化 dijkstra.**

复制代码

```
using pii = pair<int, int>;

const int dx[5] = {0, 0, 1, -1};
const int dy[5] = {1, -1, 0, 0};

int Solve(const vector<vector<int>>& G, int sx, int sy, int ex, int ey)
```

```
7 {
8         int n = G.size();
9     if (n == 0) return 0;
10    int m = G[0].size();
11    vector<int> pre(n * m + 1, 0);
12    int ret = 0;
13
14    for (int i = 0; i < n * m; i++) {
15            pre[i] = i;
16    }
17
18    function<int(int, int)> ok = [&](int x, int y) {
19            return x >= 0 && x < n && y >= 0 && y < m;
20    };
21
22    function<int(int, int)> id = [&](int x, int y) {
23            return x * m + y;
24    };
25
26    function<pair<int, int>(int)> di = [&](int pos) {
27            int x = pos / m;
28        int y = pos % m;
29        return make_pair(x, y);
30    };
31
32        function<int(int)> find = [&](int x) {
33            return x == pre[x] ? x : pre[x] = find(pre[x]);
34    };
35
36    function<void(int, int)> unite = [&](int x, int y) {
37            pre[find(x)] = find(y);
38    };
39
40        for (int i = 0; i < n; i++) {
41            for (int j = 0; j < m; j++) {
42            if (G[i][j] == 1) continue;
43            for (int k = 0; k < 4; k++) {
44                int xx = i + dx[k];
45                int yy = j + dy[k];
46                    if (!ok(xx, yy)) continue;
47                if (G[xx][yy] == 0) unite(id(i, j), id(xx, yy));
48            }
49        }
50    }
```

```
51
52    vector<pii> graph[n * m];
53    for (int i = 0; i < n; i++) {
54            for (int j = 0; j < m; j++) {
55            for (int k = 0; k < 4; k++) {
56                int xx = i + dx[k];
57                int yy = j + dy[k];
58                    if (!ok(xx, yy)) continue;
59                if (find(id(i, j)) == find(id(xx, yy))) continue;
60                graph[id(i, j)].emplace_back(1, id(xx, yy));
61                graph[id(xx, yy)].emplace_back(1, id(i, j));
62            }
63        }
64    }
65
66        vector<int> dis(n * m, 1E5);
67    priority_queue<pii, vector<pii>, greater<pii>> pq;
68    dis[find(id(sx, sy))] = 0;
69        pq.emplace(dis[find(id(sx, sy))], id(sx, sy));
70        while (!pq.empty()) {
71            pii cur = pq.top(); pq.pop();
72        int w = cur.first, v = cur.second;
73        for (int i = 0; i < graph[v].size(); i++) {
74            int u = graph[v][i].second;
75                int d = w + graph[v][i].first;
76            if (dis[v] + d < dis[u]) {
77                    dis[u] = dis[v] + d;
78                pq.emplace(dis[u], u);
79            }
80        }
81    }
82    return dis[find(id(ex, ey))];
  }
```