**目录**

[Hansimov](https://github.com/Hansimov/cs-notes)

[回溯、搜索 1](#_Toc66290152)

[0017 电话号码的字母组合 1](#_Toc66290153)

[0022 括号生成 2](#_Toc66290154)

[0039 组合总和 3](#_Toc66290155)

[0040 组合总和 II 6](#_Toc66290156)

[0216 组合总和 III 8](#_Toc66290157)

[0046 全排列 9](#_Toc66290158)

[0047 全排列 II 12](#_Toc66290159)

[0077 组合 14](#_Toc66290160)

[0078 子集 15](#_Toc66290161)

[0090 子集 II 17](#_Toc66290162)

[0079 单词搜索 + 剑指 12 矩阵中的路径 20](#_Toc66290163)

[剑指 13 机器人的运动范围 22](#_Toc66290164)

[剑指 38 字符串的排列 26](#_Toc66290165)

[二分查找 28](#_Toc66290166)

[0074 搜索二维矩阵 28](#_Toc66290167)

[0240 搜索二维矩阵 II 29](#_Toc66290168)

[数学 31](#_Toc66290169)

[0007 整数反转 31](#_Toc66290170)

[0009 回文数 32](#_Toc66290171)

[0050 Pow(x, n) 33](#_Toc66290172)

[0069 x的平方根 35](#_Toc66290173)

[0171 Excel表列序号 37](#_Toc66290174)

[0172 阶乘后的零 38](#_Toc66290175)

[0202 快乐数 39](#_Toc66290176)

[0231 2的幂 42](#_Toc66290177)

[0326 3的幂 43](#_Toc66290178)

[0258 各位相加 44](#_Toc66290179)

[0268 丢失的数字 46](#_Toc66290180)

[0728 自除数 47](#_Toc66290181)

[链表 48](#_Toc66290182)

[如何通过数组构造一个链表 48](#_Toc66290183)

[0002 两数相加 49](#_Toc66290184)

[0021 合并两个有序链表 51](#_Toc66290185)

[0023 合并k个升序链表 52](#_Toc66290186)

[0025 K个一组翻转链表 55](#_Toc66290187)

[0141 环形链表 57](#_Toc66290188)

[0142 环形链表 II 58](#_Toc66290189)

[0146 LRU缓存机制 61](#_Toc66290190)

[0206 反转链表 64](#_Toc66290191)

[0445 两数相加 II 65](#_Toc66290192)

[树 67](#_Toc66290193)

[如何通过数组构造二叉树 67](#_Toc66290194)

[0094 二叉树的中序遍历 69](#_Toc66290195)

[0102 二叉树的层序遍历 71](#_Toc66290196)

[0103 二叉树的锯齿形层序遍历 73](#_Toc66290197)

[0104 二叉树的最大深度 74](#_Toc66290198)

[0199 二叉树的右视图 76](#_Toc66290199)

[0113 路径总和II 77](#_Toc66290200)

[0124 二叉树中的最大路径和 78](#_Toc66290201)

[0105 从前序与中序遍历序列构造二叉树 82](#_Toc66290202)

[0236 二叉树的最近公共祖先 84](#_Toc66290203)

[1008 前序遍历构造二叉搜索树 86](#_Toc66290204)

[剑指 33 二叉搜索树的后序遍历序列 89](#_Toc66290205)

[双指针 94](#_Toc66290206)

[0003 无重复字符的最长子串 94](#_Toc66290207)

[0015 三数之和 95](#_Toc66290208)

[0151 翻转字符串里的单词 + 剑指 58-1 翻转单词顺序 97](#_Toc66290209)

[剑指 21 调整数组顺序使奇数位于偶数前面 99](#_Toc66290210)

[剑指 57-1 和为s的两个数字 101](#_Toc66290211)

[剑指 57-2 和为s的连续正数序列 102](#_Toc66290212)

[栈、堆、队列 104](#_Toc66290213)

[0215 数组中的第K个最大元素 104](#_Toc66290214)

[0227 基本计算器 II + 0224 基本计算器 111](#_Toc66290215)

[0946 验证栈序列 + 剑指 31 栈的压入、弹出序列 114](#_Toc66290216)

[动态规划 115](#_Toc66290217)

[0040 接雨水 115](#_Toc66290218)

[0053 最大子序和.cpp 118](#_Toc66290219)

[0070 爬楼梯 119](#_Toc66290220)

[0121 买卖股票的最佳时机 120](#_Toc66290221)

[0238 除自身以外数组的乘积 + 剑指 66 构建乘积数组 121](#_Toc66290222)

[0263 丑数 123](#_Toc66290223)

[0264 丑数 II + 剑指 49 丑数 II 124](#_Toc66290224)

[0279 完全平方数 126](#_Toc66290225)

[0300 最长递增（上升）子序列 128](#_Toc66290226)

[0313 超级丑数 131](#_Toc66290227)

[0647 回文子串 132](#_Toc66290228)

[0005 最长回文子串 133](#_Toc66290229)

[1143 最长公共子序列 136](#_Toc66290230)

[剑指 46 把数字翻译成字符串 136](#_Toc66290231)

[剑指 60 n个骰子的点数 138](#_Toc66290232)

[剑指 62 圆圈中最后剩下的数字 140](#_Toc66290233)

[排序 142](#_Toc66290234)

[剑指 45 把数组排成最小的数 142](#_Toc66290235)

[剑指 51 数组中的逆序对 143](#_Toc66290236)

[位运算 146](#_Toc66290237)

[0136 只出现一次的数字 146](#_Toc66290238)

[0137 只出现一次的数字 II + 剑指 56-2 数组中数字出现的次数 II 147](#_Toc66290239)

[剑指 56-1 数组中数字出现的次数 148](#_Toc66290240)

[集合、哈希表 150](#_Toc66290241)

[0001 两数之和 150](#_Toc66290242)

[剑指 61 扑克牌中的顺子 151](#_Toc66290243)

[其他 152](#_Toc66290244)

[剑指 59-2 左旋转字符串 152](#_Toc66290245)

# 回溯、搜索

## 0017 电话号码的字母组合

链接：<https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number/>

标签：回溯，深搜

### 精选题解

* + <https://leetcode-cn.com/problems/letter-combinations-of-a-phone-number/solution/dian-hua-hao-ma-de-zi-mu-zu-he-by-leetcode-solutio/563076>

### 关键思路

利用字符串vector简化索引，并将输入digits的数字字符映射到其在board中的索引。

1. vector<string> board = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
2. …
3. int board\_idx = digits[digit\_idx] - '0';

### 复杂度

* 时间复杂度：O()。n为输入的字符串长度，k为对应3个字母的字符个数。
* 空间复杂度：SO(n)。递归深度为O(n)；临时数组大小为n。

### 代码

| **0017 电话号码的字母组合.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138862517/ |
| 1. class Solution { 2. public: 3. vector<string> res; 4. string temp; 5. vector<string> board = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"}; 6. void dfs(int digit\_idx, string& digits) { 7. if (digit\_idx == digits.size()) { 8. res.push\_back(temp); 9. return ; 10. } 11. int board\_idx = digits[digit\_idx] - '0'; 12. for (int i=0; i<board[board\_idx].size(); ++i) { 13. temp.push\_back(board[board\_idx][i]); 14. dfs(digit\_idx+1, digits); 15. temp.pop\_back(); 16. } 17. } 18. vector<string> letterCombinations(string digits) { 19. if (digits.size() == 0) 20. return res; 21. dfs(0, digits); 22. return res; 23. } 24. }; |

## 0022 括号生成

链接：<https://leetcode-cn.com/problems/generate-parentheses/>

标签：字符串

### 精选题解

* 官方题解 - 括号生成
  + https://leetcode-cn.com/problems/generate-parentheses/solution/gua-hao-sheng-cheng-by-leetcode-solution/

### 关键思路

* 终止条件：temp.size() == 2\*n。其中n表示生成的括号对数。
* 如果左括号个数小于n，就增加一个左括号并递归。
* 如果右括号个数小于左括号个数，就增加一个右括号并递归。

### 复杂度

### 代码

| **0022 括号生成.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138867187/ |
| 1. class Solution { 2. public: 3. vector<string> res; 4. string temp; 5. void dfs(int n, int left, int right) { 6. if (temp.size() == 2\*n) { 7. res.push\_back(temp); 8. return ; 9. } 10. if (left < n) { 11. temp.push\_back('('); 12. dfs(n, left+1, right); 13. temp.pop\_back(); 14. } 15. if (right < left) { 16. temp.push\_back(')'); 17. dfs(n, left, right+1); 18. temp.pop\_back(); 19. } 20. } 21. vector<string> generateParenthesis(int n) { 22. dfs(n, 0, 0); 23. return res; 24. } 25. }; |

## 0039 组合总和

链接：<https://leetcode-cn.com/problems/combination-sum/>

标签：回溯

### 精选题解

* 官方题解 - 组合总和
  + https://leetcode-cn.com/problems/combination-sum/solution/zu-he-zong-he-by-leetcode-solution/

### 关键思路

几个终止条件（必须保证先后顺序）：

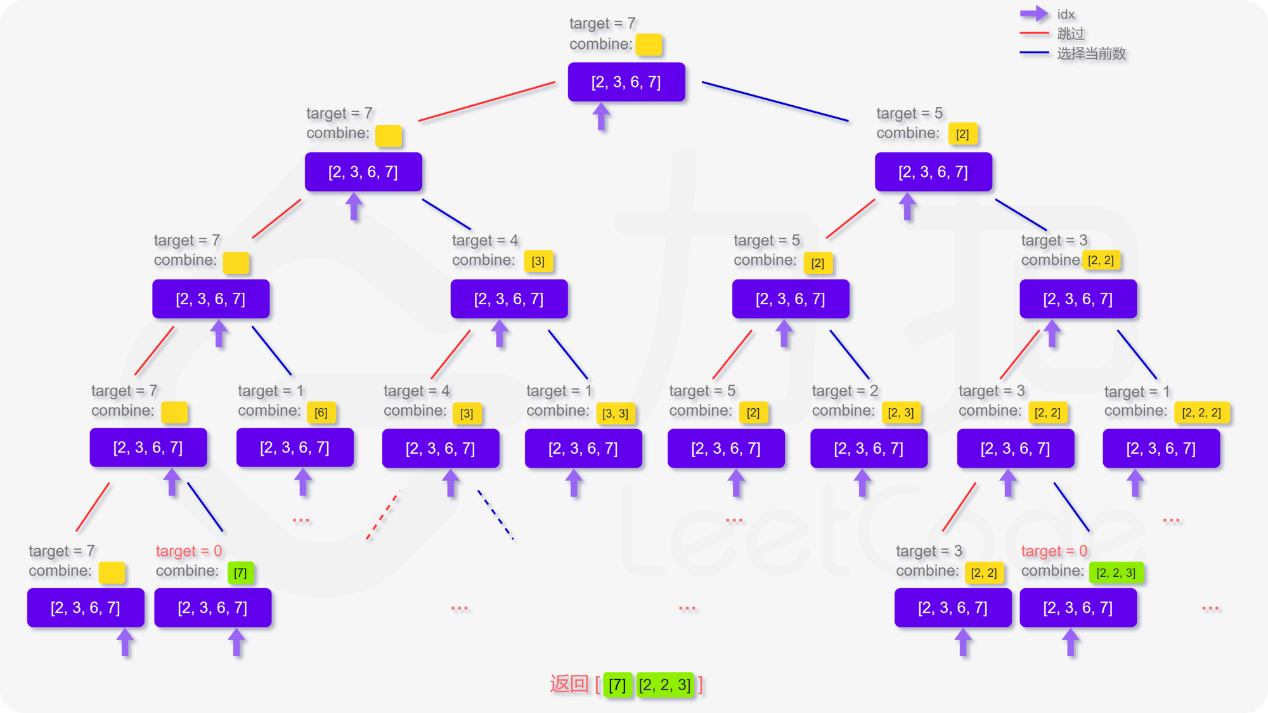
1. idx表示当前指向数字的索引，索引到达最后；
2. 和恰好为target，此时需将temp加入结果数组；
3. 后面的数都比剩余的target大，要利用此条件需要在主函数中将candidates排序。
4. if (idx==candidates.size())
5. return ;
6. if (target==0) {
7. res.push\_back(temp);
8. return ;
9. }
10. // This part must be after the previous part
11. if (target - candidates[idx] < 0) {
12. return ;
13. }

无非是取或不取当前数。区别有两点：

（1）是否出栈入栈（棕色部分）；

（2）dfs的参数变化（红色部分）。

1. // do not choose candidates[idx]
2. dfs(candidates, target, idx+1);
3. // choose candidates[idx]
4. temp.push\_back(candidates[idx]);
5. dfs(candidates, target-candidates[idx], idx);
6. temp.pop\_back();



### 复杂度

* 时间复杂度：O()。n为数组中的元素个数，此处为一个松上界，因为存在大量提前返回和剪枝，因此实际情况远小于该复杂度。
* 空间复杂度：SO(target/min(candidates))。递归层数和临时数组空间最多均为target/min(canditates)。

### 代码

| **0039 组合总和.cpp** |
| --- |
| <https://leetcode-cn.com/submissions/detail/138434940/> |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> temp; 5. void dfs(vector<int> & candidates, int target, int idx) { 6. if (idx==candidates.size()) 7. return ; 8. if (target==0) { 9. res.push\_back(temp); 10. return ; 11. } 13. *// This part must be after the previous part* 14. if (target - candidates[idx] < 0) { 15. return ; 16. } 17. *// do not choose candidates[idx]* 18. dfs(candidates, target, idx+1); 19. *// choose candidates[idx]* 20. temp.push\_back(candidates[idx]); 21. dfs(candidates, target-candidates[idx], idx); 22. temp.pop\_back(); 23. } 24. vector<vector<int>> combinationSum(vector<int>& candidates, int target) { 25. sort(candidates.begin(), candidates.end()); 26. dfs(candidates, target, 0); 27. return res; 28. } 29. }; |

## 0040 组合总和 II

链接：<https://leetcode-cn.com/problems/combination-sum-ii/>

标签：回溯

### 精选题解

* 回溯算法 + 剪枝（Java、Python）
  + <https://leetcode-cn.com/problems/combination-sum-ii/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-m-3/>
  + <https://leetcode-cn.com/problems/combination-sum-ii/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-m-3/225211>

### 关键思路

最重要的是保证选取的数不重复，和“0090 子集 II”中的思路类似，同一树层上不应有相同的数（[1,2] 和 [1,2] 不被允许），同一树枝上可以（[1,2,2] 允许）。

需要注意的是，在“0090 子集 II”的代码1中使用了used数组，判断条件多了!used[i]，因此代码2对这个剪枝方法进行了改进，采用了本题下面的做法。

我们发现，在递归中，**同一个for循环里面的数都是在同一树层中的**，因此在同一个for循环中每个数只能使用一次。

使用candidates[i]==canditates[i-1] 的含义是，将所有相同的数都跳过。

在大多数情况下，上面这条都是够的，只有当i==idx时，有可能出现这两个相同的数是在同一树枝，而不是同一树层，因为idx是循环起点，所以candidates[idx-1] 在本层递归的for循环，而canditates[idx-1] 必然在上层递归的for循环。所以加了一句i>idx的判断条件，用于排除这个例外情况。

1. for (int i=idx; i<candidates.size(); ++i) {
2. if (target - candidates[idx] < 0)
3. break;
4. if (i>idx && candidates[i] == candidates[i-1])
5. continue;
6. temp.push\_back(candidates[i]);
7. dfs(candidates, target-candidates[i], i+1);
8. temp.pop\_back();
9. }

### 复杂度

* 时间复杂度：O()。n为candidates数组长度。递归时每个数都有选或不选两种可能，故有O()的可能；每次复制符合条件的数组则需要O(n)的时间。当然，这里是一个宽松的上界，因为在实际递归中，有很多提前返回和剪枝，因此要远小于该复杂度上界。
* 空间复杂度：SO(n)。递归深度最多为n；temp数组最多n个数。

### 代码

| **0040 组合总和 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138838782/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> temp; 5. void dfs(vector<int>& candidates, int target, int idx) { 6. if (target==0) { 7. res.push\_back(temp); 8. return ; 9. } 10. for (int i=idx; i<candidates.size(); ++i) { 11. if (target - candidates[idx] < 0) 12. break; 13. if (i>idx && candidates[i] == candidates[i-1]) 14. continue; 15. temp.push\_back(candidates[i]); 16. dfs(candidates, target-candidates[i], i+1); 17. temp.pop\_back(); 18. } 19. } 20. vector<vector<int>> combinationSum2(vector<int>& candidates, int target) { 21. sort(candidates.begin(), candidates.end()); 22. dfs(candidates, target, 0); 23. return res; 24. } 25. }; |

## 0216 组合总和 III

链接：<https://leetcode-cn.com/problems/combination-sum-iii/>

标签：回溯

### 精选题解

* 官方题解 - 组合总和 III
  + <https://leetcode-cn.com/problems/combination-sum-iii/solution/zu-he-zong-he-iii-by-leetcode-solution/>

### 关键思路

该题可以视为，从9个数中取出k个数使之和为n。同“0077 组合”类似，只需满足个数为k且和为n即可。官方题解用了accumulate求和，实际上没有利用好递归的特性。

剪枝：temp数组个数大于k，或可取数组中个数加起来也不足k。

无非是选和不选当前数这两种情况。

### 复杂度

时间复杂度：O()。从9个数中取k个数；复制每个符合条件的数组需O(k)。

空间复杂度：O(k)。递归层数最多为k；临时数组大小为k。

### 代码

| **0216 组合总和 III.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138857702/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> temp; 5. void dfs(int k, int target, int cur) { 6. if ( (temp.size() + (9-cur+1) < k) || temp.size() > k) 7. return ; 8. if (temp.size() == k && target == 0) { 9. res.push\_back(temp); 10. return ; 11. } 12. *// not choose cur* 13. dfs(k, target, cur+1); 14. *// choose cur* 15. temp.push\_back(cur); 16. dfs(k, target-cur, cur+1); 17. temp.pop\_back(); 18. } 20. vector<vector<int>> combinationSum3(int k, int n) { 21. dfs(k, n, 1); 22. return res; 23. } 24. }; |

## 0046 全排列

链接：<https://leetcode-cn.com/problems/permutations/>

标签：回溯

### 精选题解

* 回溯算法入门级详解 + 练习（持续更新）
  + <https://leetcode-cn.com/problems/permutations/solution/hui-su-suan-fa-python-dai-ma-java-dai-ma-by-liweiw/>
* 官方题解
  + <https://leetcode-cn.com/problems/permutations/solution/quan-pai-lie-by-leetcode-solution-2/>
* **※ C++ 回溯法/交换法/stl 简洁易懂的全排列**
  + <https://leetcode-cn.com/problems/permutations/solution/c-hui-su-fa-jiao-huan-fa-stl-jian-ji-yi-dong-by-sm/>
* 精选代码
  + <https://leetcode-cn.com/problems/permutations/solution/quan-pai-lie-by-leetcode-solution-2/532710/>

### 复杂度

* 时间复杂度：O(n\*n!)。回溯复杂度O(n!)；每次新的生成数组需要复制n个元素。
* 空间复杂度：SO(n)。长度为n的标记数组；递归时深度最大为n。

### 代码1：标记数组

| **0046 全排列.cpp** |
| --- |
| <https://leetcode-cn.com/submissions/detail/127476452/> |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. void backtrack(vector<int> &nums, vector<int> &current, vector<bool> &flags) { 5. **if** (current.size() == flags.size()) { 6. res.push\_back(current); 7. } **else** { 8. **for** (int i=0; i<nums.size(); ++i) { 9. **if** (not flags[i]) { // nums[i] not in current 10. current.push\_back(nums[i]); 11. flags[i] = true; 12. backtrack(nums, current, flags); 13. current.pop\_back(); 14. flags[i] = false; 15. } 16. } 17. } 18. } 19. vector<vector<int>> permute(vector<int>& nums) { 20. **if** (nums.empty()) { 21. **return** {}; 22. } 23. vector<bool> flags(nums.size(), false); // true: in current; false: not in current 24. vector<int> current; 25. backtrack(nums, current, flags); 26. **return** res; 27. } 28. }; |

### 代码2：交换元素

| **0046 全排列 - swap.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/127482585/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. void backtrack(vector<int> &nums, int start, int end) { 5. **if** (start == end) { 6. res.push\_back(nums); 7. } **else** { 8. **for** (int i=start; i<=end; ++i) { 9. swap(nums[i], nums[start]); 10. backtrack(nums, start+1, end); 11. swap(nums[i], nums[start]); 12. } 13. } 14. } 15. vector<vector<int>> permute(vector<int>& nums) { 16. **if** (nums.empty()) { 17. **return** {}; 18. } **else** { 19. backtrack(nums, 0, nums.size()-1); 20. **return** res; 21. } 22. } 23. }; |

## 0047 全排列 II

题目：<https://leetcode-cn.com/problems/permutations-ii/>

标签：回溯

### 精选题解

* 官方题解
  + <https://leetcode-cn.com/problems/permutations-ii/solution/quan-pai-lie-ii-by-leetcode-solution/>

### 关键思路

定义一个标记数组visited来标记已经填过的数。若visited[i] 为true，表示第i个数已经使用了；若visited[i] 为false，表示第i个数尚未使用。

要解决重复问题，只需保证在填第i个数时，重复数字只被填入一次。方法：对原数组排序，保证相同数字都相邻，然后每次填入的数一定是这个数所在重复数集合中「从左往右第一个未被填过的数字」，即如下的判断条件：

1. if (i > 0 && nums[i] == nums[i-1] && !visited[i-1]) {
2. continue;
3. }

假如排完序后的完整数组nums中有三个连续的数，那么一定只有如下4种状态：[×, ×, ×]，[√, ×, ×]，[√, √, ×]，[√, √, √]。（√ 表示已在生成的数组中，× 表示未在生成的数组中。）

### 复杂度

详见官方题解。

* 时间复杂度：O(n\*n!)。回溯复杂度O(n!)；每次新的生成数组需要复制n个元素。
* 空间复杂度：SO(n)。长度为n的标记数组；递归时深度最大为n。

### 代码

| **0047 全排列 II.cpp** |
| --- |
| https://leetcode-cn.com/problems/permutations-ii/submissions/ |
| 1. class Solution { 2. vector<int> visited; 3. public: 4. void backtrack(vector<int> &nums, vector<vector<int>> &res, int idx, vector<int> &current) { 5. **if** (idx==nums.size()) { 6. res.emplace\_back(current); 7. // \* C++ STL vector 添加元素（push\_back()和emplace\_back()）详解 8. // \* http://c.biancheng.net/view/6826.html 9. // push\_back() 向容器尾部添加元素时，首先会创建这个元素，然后再将这个元素拷贝或者移动到容器中（如果是拷贝的话，事后会自行销毁先前创建的这个元素）； 10. // 而 emplace\_back() 在实现时，则是直接在容器尾部创建这个元素，省去了拷贝或移动元素的过程。 11. **return** ; 12. } 13. **for** (int i=0; i<nums.size(); ++i) { 14. // 哪些情况不取当前的元素： 15. // 1. 已经访问过/在当前路径数组中 16. // 2. 和前一个数相等，且前一个数未被填过（表明该数不是第一个未填的数，故仍然跳过） 17. // 反过来理解，如果前一个相等的数已经被填过，那么此时就可以插入这后一个相等的数了， 18. // 因为我们在上一层嵌套中，已经保证前一个数当时是第一个未被填过的数了 19. // 此时意味着我们在当前路径数组中存在多个相等的数了 20. **if** (visited[i] || (i>0 && nums[i]==nums[i-1] && !visited[i-1])) 21. **continue**; 22. current.emplace\_back(nums[i]); 23. visited[i] = true; 24. backtrack(nums, res, idx+1, current); 25. visited[i] = false; 26. current.pop\_back(); 27. } 28. } 29. vector<vector<int>> permuteUnique(vector<int>& nums) { 30. vector<vector<int>> res; 31. vector<int> current; 32. visited.resize(nums.size()); 33. sort(nums.begin(), nums.end()); 34. backtrack(nums, res, 0, current); 35. **return** res; 36. } 37. }; |

## 0077 组合

链接：<https://leetcode-cn.com/problems/combinations/>

标签：回溯

### 精选题解

* **※ 官方题解 - 组合**
  + <https://leetcode-cn.com/problems/combinations/solution/zu-he-by-leetcode-solution/>
* 回溯算法 + 剪枝（Java） - 组合
  + <https://leetcode-cn.com/problems/combinations/solution/hui-su-suan-fa-jian-zhi-python-dai-ma-java-dai-ma-/>

### 关键思路

合理剪枝：剩余个数是否足够；个数正好则加入并返回。

1. if (temp.size() + (n - cur + 1) < k)
2. return ;

选取当前数，需要考虑入栈出栈；不选取，则跳到下一个。

1. // choose cur
2. temp.push\_back(cur);
3. dfs(cur+1, n, k);
4. temp.pop\_back();
5. // do not choose cur
6. dfs(cur+1, n, k);

### 复杂度

* 时间复杂度：O()。其中表示从n个数中取出k个数的组合数目，k表示每次需要复制k个数。
* 空间复杂度：O(n+k)=O(n)。递归最大层数n；临时数组空间k。

### 代码

| **0077 组合.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138357287/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; // result 2d vector 4. vector<int> temp; // temp vector path 5. void dfs(int cur, int n, int k) { 6. // cur: current element index, choose or not 7. if (temp.size() + (n - cur + 1) < k) 8. return ; 9. if (temp.size() == k) { 10. res.push\_back(temp); 11. return ; 12. } 13. // choose cur 14. temp.push\_back(cur); 15. dfs(cur+1, n, k); 16. temp.pop\_back(); 17. // do not choose cur 18. dfs(cur+1, n, k); 19. } 20. vector<vector<int>> combine(int n, int k) { 21. dfs(1, n, k); 22. return res; 23. } 24. }; |

## 0078 子集

链接：https://leetcode-cn.com/problems/subsets/

标签：回溯，位运算

### 精选题解

* 官方题解 - 子集
  + <https://leetcode-cn.com/problems/subsets/solution/zi-ji-by-leetcode-solution/>

### 关键思路

每个位置有两种情况，选或者不选，所以类似“0077组合”(p14)的思路。最后索引到达n就退出。

1. *// choose nums[cur]*
2. temp.push\_back(nums[cur]);
3. dfs(cur+1, nums);
4. temp.pop\_back();
5. *// not choose nums[cur]*
6. dfs(cur+1, nums);

### 复杂度

* 时间复杂度：O()。一共个子集，每个子集需要O(n)的时间来构造。
* 空间复杂度：SO(n)。递归栈空间O(n)；临时数组空间O(n)。

### 代码

| **0078 子集.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138375047/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> temp; 5. void dfs(int cur, vector<int> &nums) { 6. if (cur == nums.size()) { 7. res.push\_back(temp); 8. return ; 9. } 10. *// choose nums[cur]* 11. temp.push\_back(nums[cur]); 12. dfs(cur+1, nums); 13. temp.pop\_back(); 14. *// not choose nums[cur]* 15. dfs(cur+1, nums); 16. } 17. vector<vector<int>> subsets(vector<int>& nums) { 18. dfs(0, nums); 19. return res; 20. } 21. }; |

## 0090 子集 II

链接：<https://leetcode-cn.com/problems/subsets-ii/>

标签：回溯

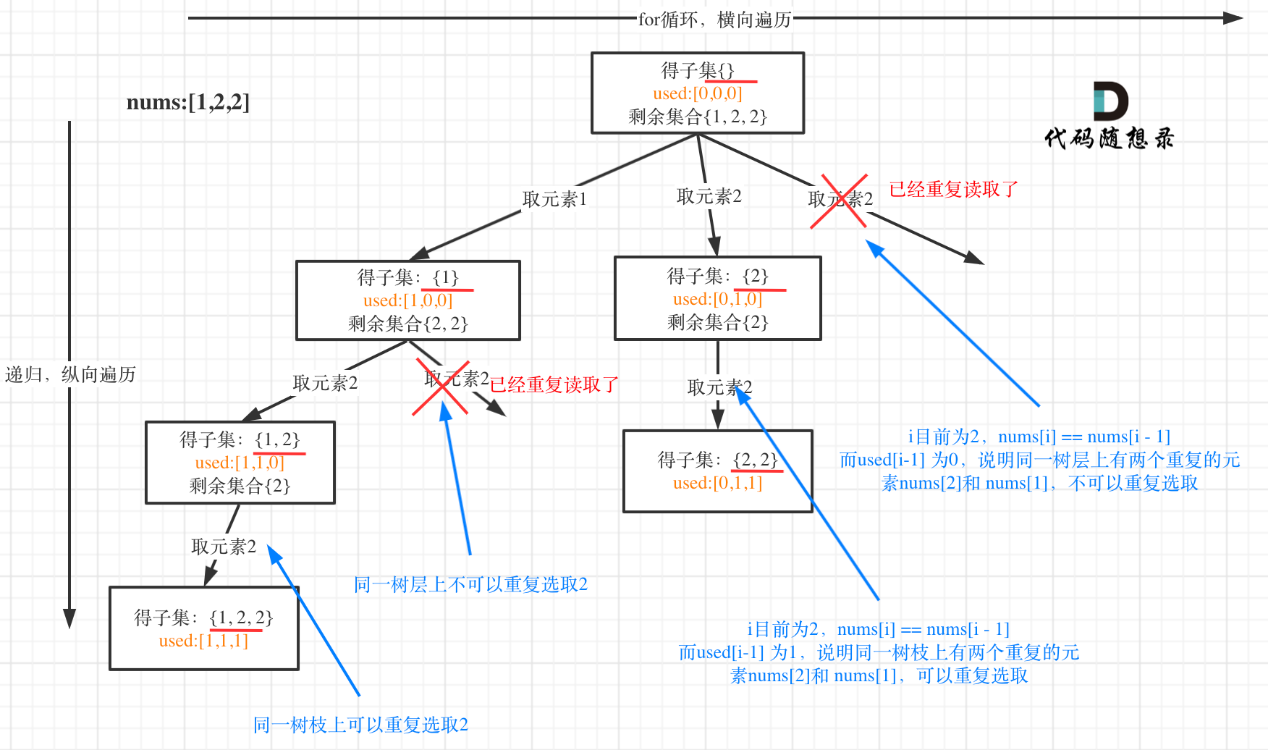
### 精选题解

* 90. 子集 II:【彻底理解子集问题如何去重】详解 - 子集 II
  + <https://leetcode-cn.com/problems/subsets-ii/solution/90-zi-ji-iiche-di-li-jie-zi-ji-wen-ti-ru-he-qu-zho/>

### 关键思路

最重要的是理解used[i-1] 的含义：

1. true：同一**树枝**上选取过值相等的元素，可以重复选取
2. false：同一**树层**上选取过值相等的元素，不可重复选取



类似“0047 全排列 II”，但有几点不同：

1. 循环遍历的起点是cur而不是0。因此，也就不需要判断used[i] 是否为true，因为这时肯定是false。
2. 可直接将temp加入res，视为不选取nums[cur]。
3. // not choose nums[cur]
4. res.push\_back(temp);
5. // maybe choose nums[cur]
6. for (int i=cur; i<nums.size(); ++i) {
7. if (i>0 && nums[i]==nums[i-1] && !used[i-1])
8. continue;
9. temp.push\_back(nums[i]);
10. used[i] = true;
11. backtrack(i+1, nums, used);
12. used[i] = false;
13. temp.pop\_back();
14. }

事实上，还可以对上面的剪枝进行优化，不需要使用used数组，可以参考“0040 组合总和 II”的题解，重点是下面第2行的蓝色语句。代码2就是采用了该剪枝方法的优化解法。

1. for (int i=cur; i<nums.size(); ++i) {
2. if (i>cur && nums[i]==nums[i-1])
3. continue;
4. temp.push\_back(nums[i]);
5. backtrack(i+1, nums);
6. temp.pop\_back();
7. }

### 复杂度

时间复杂度：O()。子集最多有个（元素均不重复）；构造每个子集需要O(n)的时间。

空间复杂度：SO(n)。递归栈空间O(n)；临时数组空间O(n)。

### 代码1

| **0090 子集 II** |
| --- |
| <https://leetcode-cn.com/submissions/detail/138389158/> |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> temp; 5. void backtrack(int cur, vector<int> &nums, vector<bool> &used) { 6. *// not choose nums[cur]* 7. res.push\_back(temp); 8. *// maybe choose nums[cur]* 9. for (int i=cur; i<nums.size(); ++i) { 10. if (i>0 && nums[i]==nums[i-1] && !used[i-1]) 11. continue; 12. temp.push\_back(nums[i]); 13. used[i] = true; 14. backtrack(i+1, nums, used); 15. used[i] = false; 16. temp.pop\_back(); 17. } 18. } 19. vector<vector<int>> subsetsWithDup(vector<int>& nums) { 20. vector<bool> used(nums.size(), false); 21. sort(nums.begin(), nums.end()); 22. backtrack(0, nums, used); 23. return res; 24. } 25. }; |

### 代码2：不使用used数组的剪枝

| **0090 子集 II -v2.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138841714/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> temp; 5. void backtrack(int cur, vector<int> &nums) { 6. *// not choose nums[cur]* 7. res.push\_back(temp); 8. *// maybe choose nums[cur]* 9. for (int i=cur; i<nums.size(); ++i) { 10. if (i>cur && nums[i]==nums[i-1]) 11. continue; 12. temp.push\_back(nums[i]); 13. backtrack(i+1, nums); 14. temp.pop\_back(); 15. } 16. } 17. vector<vector<int>> subsetsWithDup(vector<int>& nums) { 18. sort(nums.begin(), nums.end()); 19. backtrack(0, nums); 20. return res; 21. } 22. }; |

## 0079 单词搜索 + 剑指 12 矩阵中的路径

链接：<https://leetcode-cn.com/problems/word-search/>

<https://leetcode-cn.com/problems/ju-zhen-zhong-de-lu-jing-lcof/>

标签：回溯

### 精选题解

* ※ 官方题解 - 单词搜索
  + <https://leetcode-cn.com/problems/word-search/solution/dan-ci-sou-suo-by-leetcode-solution/>
* 在二维平面上使用回溯法（Python 代码、Java 代码） - 单词搜索
  + <https://leetcode-cn.com/problems/word-search/solution/zai-er-wei-ping-mian-shang-shi-yong-hui-su-fa-pyth/>
* 面试题12. 矩阵中的路径（ DFS + 剪枝 ，清晰图解） - 矩阵中的路径
  + 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/

### 关键思路

* check(i,j,k,…) 表示是否存在一条从board[i][j] 出发的路径与单词子集word[k:]匹配。对k=0，遍历所有的i和j，即可得到二维网格中是否包含整个单词。
* 对不同方向的搜索，可以建立一个vector<pair<int,int>> 表示四个方向。
* 终止条件：board[i][j] != word[k]，返回false；board[i][j] == word[k] && k == word.length()-1，返回true。
* 访问board[i][j] 时将其置为true，递归返回时不要忘记将其置为false。

### 复杂度

详见官方题解。

* 时间复杂度：O()。M表示board的行数，N表示board的列数，L表示字符串长度。需要进行次检查；每个后继字符至多有3个方向（除了第2个字符有4个方向），因此每次检查至多有种分支；事实上由于提前返回和剪枝的存在，实际时间复杂度远低于这个理论上界。
* 空间复杂度：SO()。visited数组的空间为；栈的深度至多为。

### 代码

| **0079 单词搜索.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/138405047/ |
| 1. class Solution { 2. public: 3. vector<pair<int,int>> directions{{0,1},{0,-1},{1,0},{-1,0}}; 4. *// check(i,j,k,...): is exist a path starts from board[i][j] matches word[k:]* 5. bool check(int i, int j, int k, vector<vector<char>> &board, vector<vector<bool>> &visited, string &word) { 6. if (board[i][j] != word[k]) 7. return false; 8. else if (k==word.length()-1) 9. return true; 10. visited[i][j] = true; 11. for (const auto& d: directions) { 12. int i\_new = i + d.first; 13. int j\_new = j + d.second; 14. if (i\_new>=0 && i\_new<board.size() && j\_new>=0 && j\_new<board[0].size()) { 15. if (visited[i\_new][j\_new]) 16. continue; 17. if (check(i\_new, j\_new, k+1, board, visited, word)) { 18. visited[i][j] = false; 19. return true; 20. } 21. } 22. } 23. visited[i][j] = false; 24. return false; 25. } 26. bool exist(vector<vector<char>>& board, string word) { 27. int row\_num = board.size(); 28. if (row\_num<=0) 29. return false; 30. int col\_num = board[0].size(); 31. bool flag = false; 32. vector<vector<bool>> visited(row\_num, vector<bool>(col\_num)); 33. for (int i=0; i<row\_num; ++i) { 34. for (int j=0; j<col\_num; ++j) { 35. flag = check(i, j, 0, board, visited, word); 36. if (flag) 37. return true; 38. } 39. } 40. return false; 41. } 42. }; |

## 剑指 13 机器人的运动范围

链接：<https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/>

标签：回溯

### 精选题解

* 官方题解 - 机器人的运动范围
  + https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/solution/ji-qi-ren-de-yun-dong-fan-wei-by-leetcode-solution/
* 面试题13. 机器人的运动范围（ 回溯算法，DFS / BFS ，清晰图解）
  + <https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/solution/mian-shi-ti-13-ji-qi-ren-de-yun-dong-fan-wei-dfs-b/>
* DFS和BFS两种解决方式 - 机器人的运动范围
  + <https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/solution/dfshe-bfsliang-chong-jie-jue-fang-shi-by-sdwwld/>
* 图解 BFS + DFS - 机器人的运动范围
  + https://leetcode-cn.com/problems/ji-qi-ren-de-yun-dong-fan-wei-lcof/solution/bfs-by-z1m/

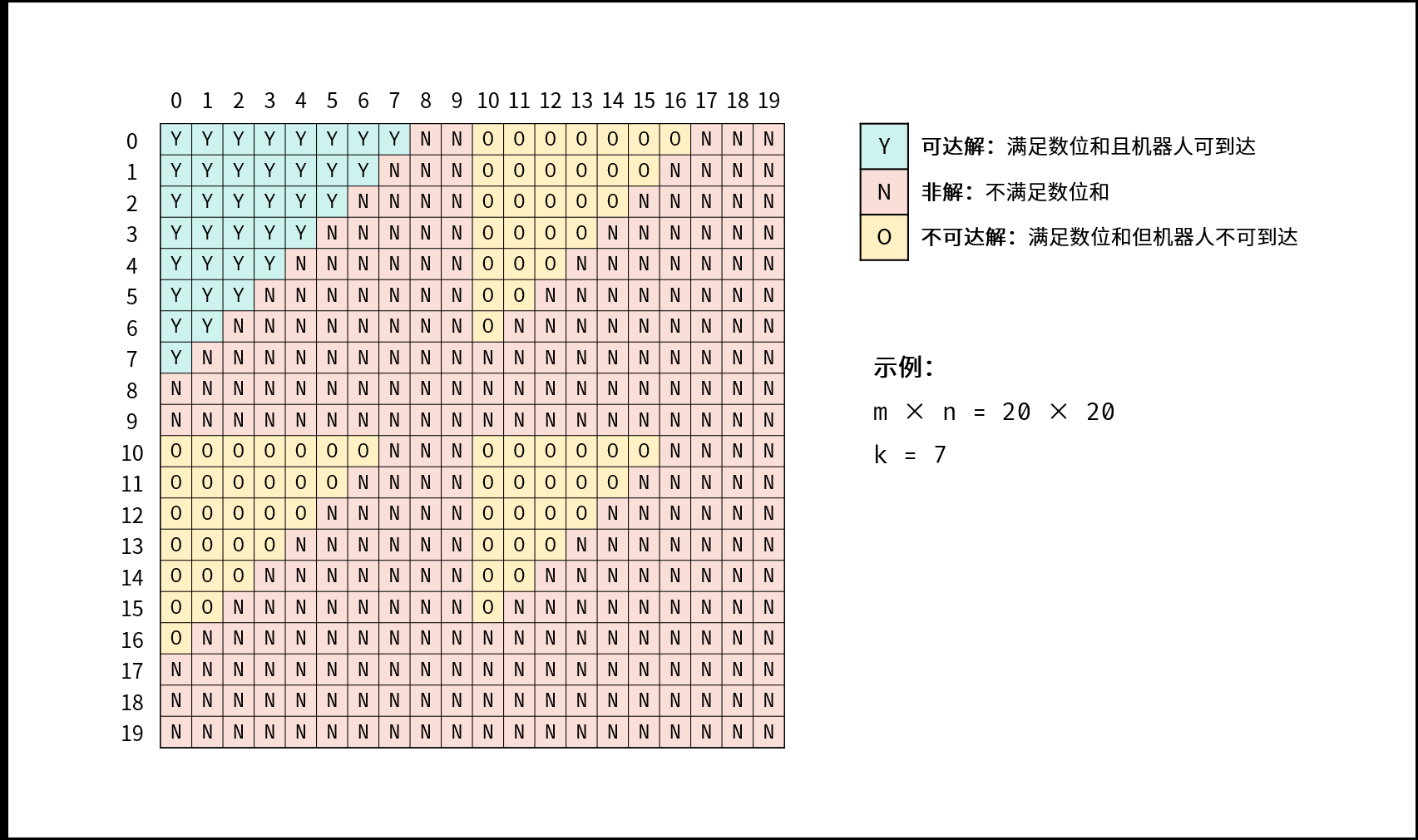
### 关键思路

审题！审题！审题！

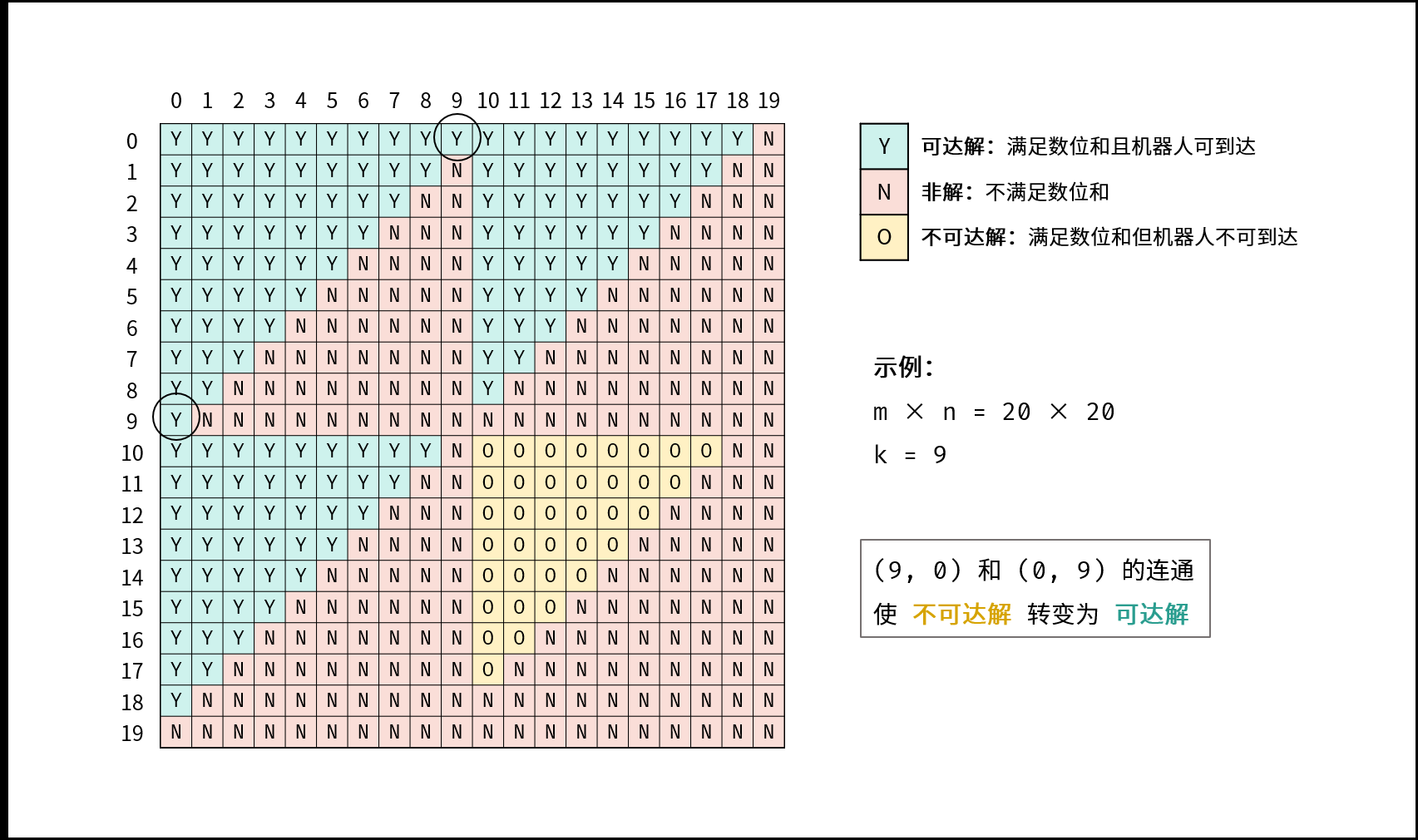
* 这里的k并不是指移动k步，而仅仅是对行坐标和列坐标数位之和的一个人为限定
* 如果看不懂解法中的某个奇怪的做法，一般都是因为对题意理解有误

第一种思路是经典回溯，可用深搜，只需将“行列数位和是否大于k”加入到搜索终止条件中（代码1）。

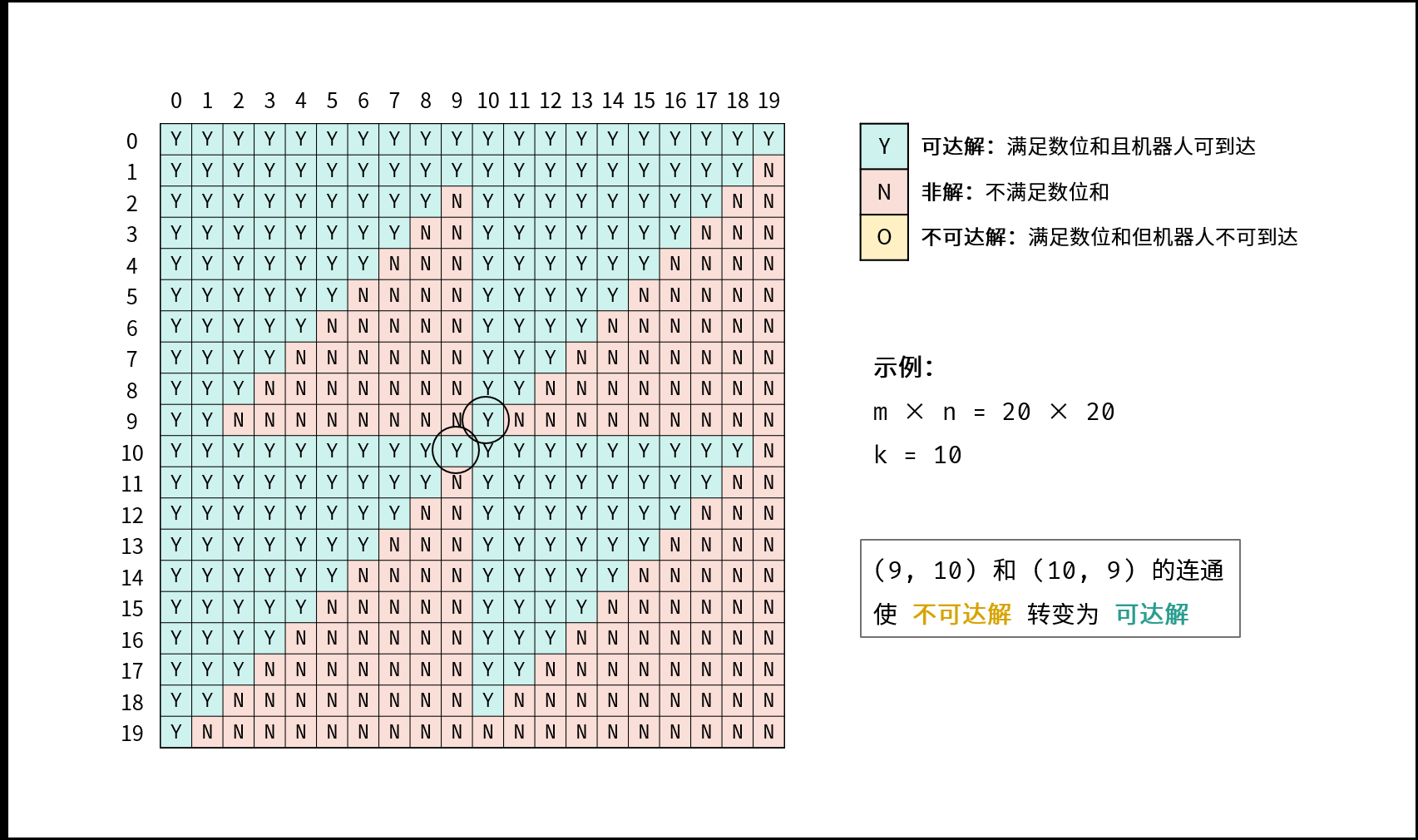
* 尽管机器人在运动时有如下三种情形，但是使用dfs向右向下可以保证覆盖完整：



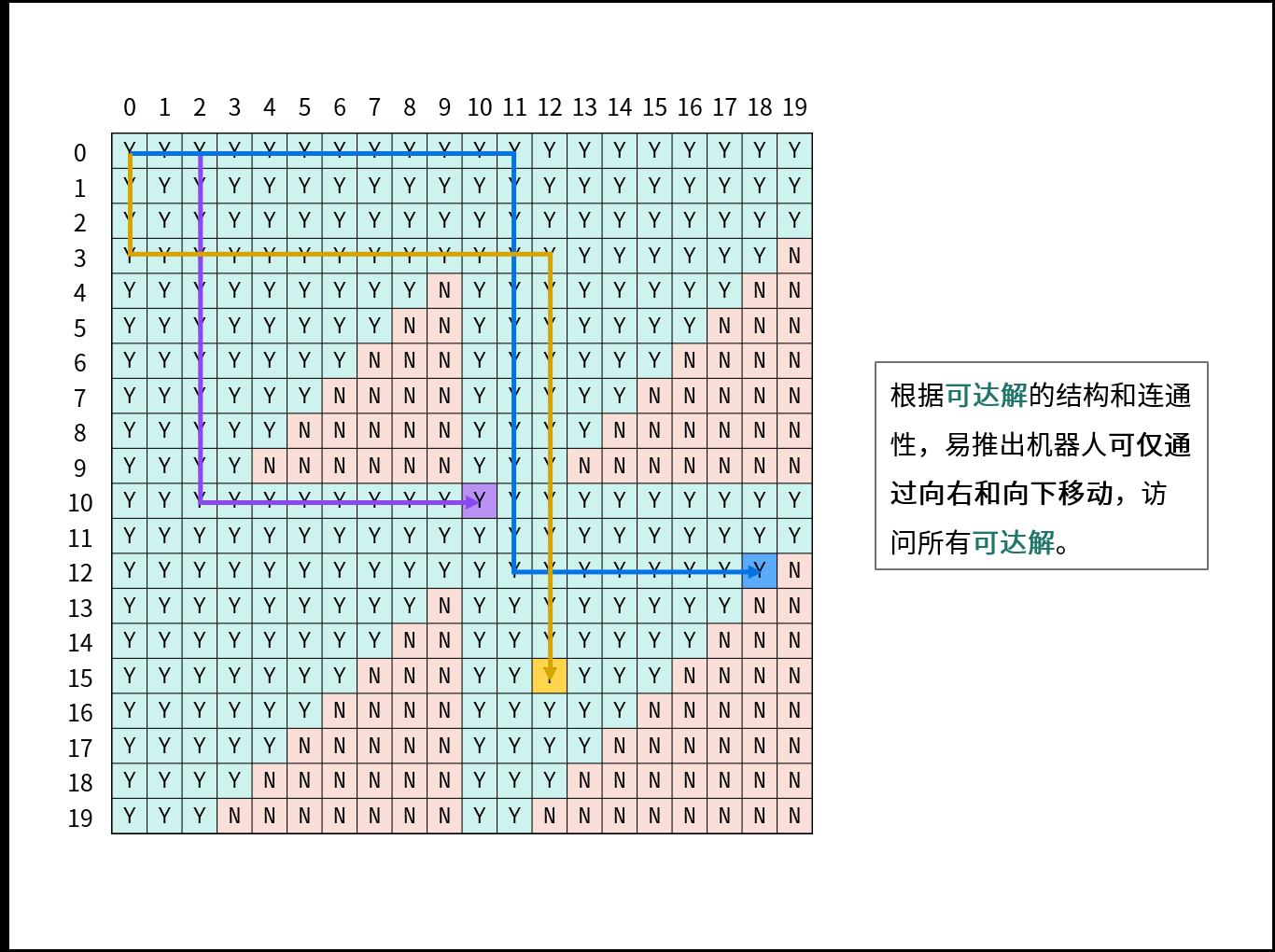
（a）未连通



（b）部分连通

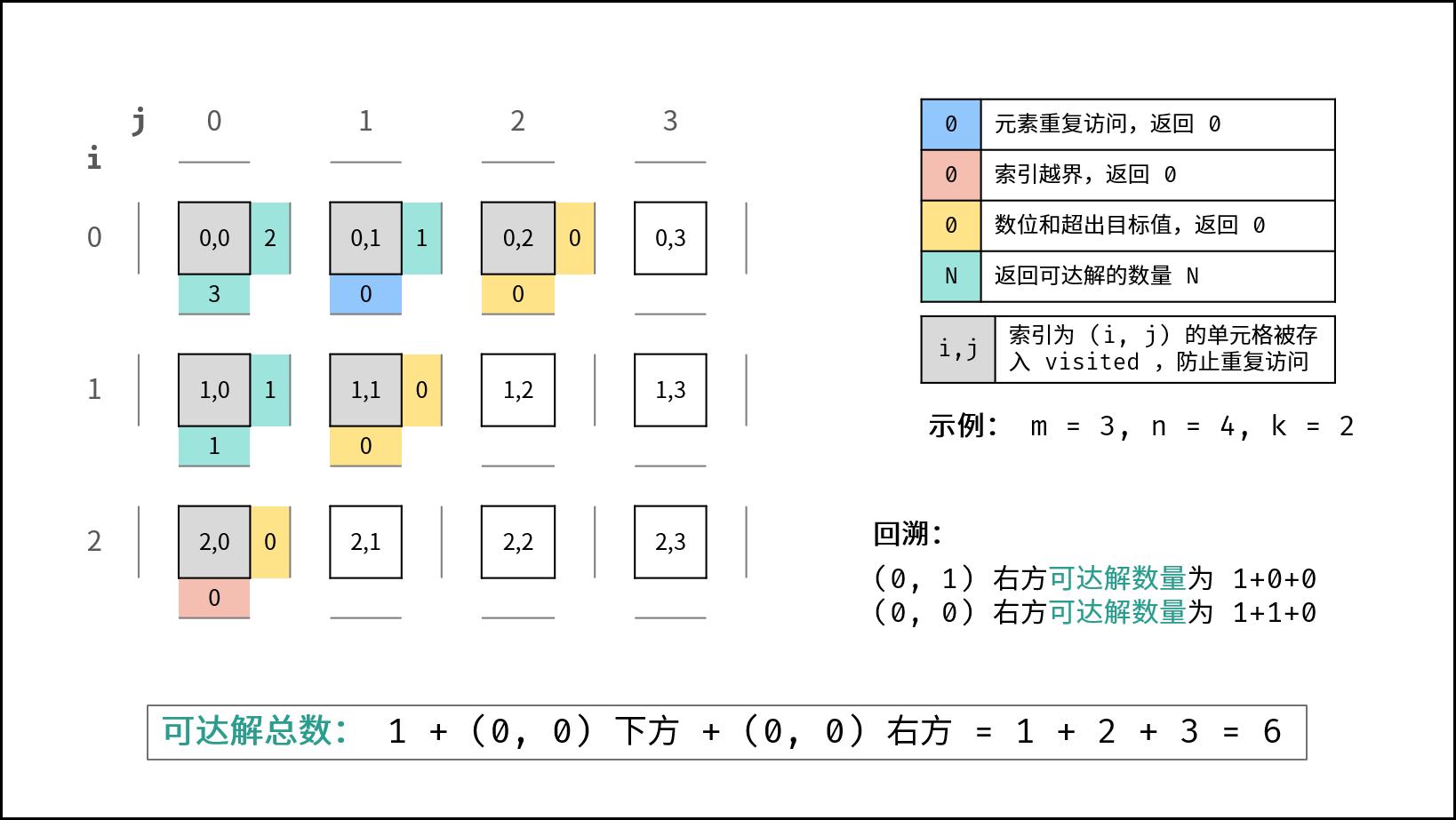


（c）全连通



只需向右向下即可保证覆盖所有可达解

* dfs的返回值含义为：当前分支有多少可达的格子。递推公式为：
  + dfs(i, j) = 1 + dfs(i+1, j) + dfs(i, j+1)



* 还需使用visited数组记录访问过的数组，若该格子被访问过，则终止搜索，比如上图中坐标 (1,1) 就可以同时由 (1,0) 向右或者 (0,1) 向下得到。

第二种思路是直接迭代递推（代码2）。

* 使用数组reachable，表示格子是否可以到达
* 若当前格子的左方格子或上方格子可达，则当前格子可达，对其标记并为计数加1

### 复杂度

* 时间复杂度：O()。遍历所有格子。
* 空间复杂度：O()。dfs中的visited，或是递推中的reachable。

### 代码1 – DFS

| **JZ-13 机器人的运动范围 - dfs.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142175514/ |
| 1. class Solution { 2. public: 3. int digitSum(int x) { 4. int sum = 0; 5. while (x!=0) { 6. sum += x % 10; 7. x /= 10; 8. } 9. return sum; 10. } 11. int dfs(int m, int n, int k, int i, int j, vector<vector<bool>>& visited) { 12. if (i>=m || j>=n || visited[i][j] || digitSum(i)+digitSum(j)>k) 13. return 0; 14. visited[i][j] = true; 15. return 1 + dfs(m, n, k, i+1, j, visited) + dfs(m, n, k, i, j+1, visited); 16. } 17. int movingCount(int m, int n, int k) { 18. vector<vector<bool>> visited(m, vector<bool>(n, false)); 19. return dfs(m, n, k, 0, 0, visited); 20. } 21. }; |

### 代码2 – 迭代递推

| **JZ-13 机器人的运动范围 - iterative.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142178886/ |
| 1. class Solution { 2. public: 3. int digitSum(int x) { 4. int sum = 0; 5. while (x!=0) { 6. sum += x % 10; 7. x /= 10; 8. } 9. return sum; 10. } 11. int movingCount(int m, int n, int k) { 12. if (k==0) 13. return 1; 14. int res = 1; 15. vector<vector<int>> reachable(m, vector<int>(n, 0)); 16. reachable[0][0] = 1; 17. for (int i=0; i<m; ++i) { 18. for (int j=0; j<n; ++j) { 19. if ( (i==0 && j==0) || digitSum(i)+digitSum(j)>k) 20. continue; 21. if (i>=1) 22. reachable[i][j] |= reachable[i-1][j]; 23. if (j>=1) 24. reachable[i][j] |= reachable[i][j-1]; 25. res += reachable[i][j]; 26. } 27. } 28. return res; 29. } 30. }; |

## 剑指 38 字符串的排列

链接：<https://leetcode-cn.com/problems/zi-fu-chuan-de-pai-lie-lcof/>

标签：回溯

### 精选题解

* 面试题38. 字符串的排列（回溯法，清晰图解）
  + <https://leetcode-cn.com/problems/zi-fu-chuan-de-pai-lie-lcof/solution/mian-shi-ti-38-zi-fu-chuan-de-pai-lie-hui-su-fa-by/>
* **※ 回溯法\_面试题38. 字符串的排列**
  + <https://leetcode-cn.com/problems/zi-fu-chuan-de-pai-lie-lcof/solution/hui-su-fa-by-luo-jing-yu-yu/>

### 关键思路

类似“0047 全排列 II”。当然，还有一种通过交换数组元素+set去重的方法，此处略。

### 复杂度

* 时间复杂度：O(n!)。其中n为字符串长度，n个字符均不相同时，全排列最多有n!种可能。
* 空间复杂度：SO(n)。递归层数为n；visited数组大小为n；temp数组大小为n。

### 代码

| **JZ-38 字符串的排列.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/139043826/ |
| 1. class Solution { 2. public: 3. vector<string> res; 4. string temp; 5. void dfs(string& s, int cnt, vector<bool>& visited) { 6. if (cnt==s.size()) { 7. res.push\_back(temp); 8. return ; 9. } 10. for (int i=0; i<s.size(); ++i) { 11. if (visited[i] || i>0 && s[i]==s[i-1] && !visited[i-1]) 12. continue; 13. temp.push\_back(s[i]); 14. visited[i] = true; 15. dfs(s, cnt+1, visited); 16. visited[i] = false; 17. temp.pop\_back(); 18. } 19. } 20. vector<string> permutation(string s) { 21. if (s.empty()) 22. return res; 23. sort(s.begin(), s.end()); 24. vector<bool> visited(s.size(), false); 25. dfs(s, 0, visited); 26. return res; 27. } 28. }; |

# 二分查找

## 0074 搜索二维矩阵

链接：https://leetcode-cn.com/problems/search-a-2d-matrix/

标签：搜索，二分查找

### 精选题解

* 搜索二维矩阵 - 搜索二维矩阵
  + https://leetcode-cn.com/problems/search-a-2d-matrix/solution/sou-suo-er-wei-ju-zhen-by-leetcode/

### 关键思路

将二维数组的行列索引映射到一维的索引，然后使用二分查找。

二维行列到一维索引的映射关系为：r = mid / col\_num；c = mid % col\_num。

* **注意：分母是列数col\_num，而不是行数row\_num（在这个地方吃过两次亏了）**

关于二分查找的实现细节，可以参考本人“[后台面经整理.docx](https://github.com/Hansimov/cs-interviews)”中“C++代码实现”的“二分查找”。

### 复杂度

* 时间复杂度：O(logn)。二分查找的复杂度。
* 空间复杂度：O(1)。

### 代码

| **0074 搜索二维矩阵.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148665982/ |
| 1. **class** **Solution** { 2. **public**: 3. bool searchMatrix(vector<vector<int>>& matrix, int target) { 4. int row\_num = matrix.size(); 5. **if** (row\_num == 0) 6. **return** false; 7. int col\_num = matrix[0].size(); 8. int left = 0, right = row\_num \* col\_num, mid; 9. int idx, r, c, val; 10. **while** (left < right) { 11. mid = left + (right-left) / 2; 12. r = mid / col\_num; 13. c = mid % col\_num; 14. val = matrix[r][c]; 15. **if** (val == target) 16. **return** true; 17. **else** **if** (val<target) 18. left = mid + 1; 19. **else** 20. right = mid; 21. } 22. **return** false; 23. } 24. }; |

## 0240 搜索二维矩阵 II

链接：https://leetcode-cn.com/problems/search-a-2d-matrix-ii/

标签：

### 精选题解

* 思路清晰 - 折线搜索 - 搜索二维矩阵 II
  + <https://leetcode-cn.com/problems/search-a-2d-matrix-ii/solution/si-lu-qing-xi-zhe-xian-sou-suo-by-eloise-kisj/>
* 折线搜索 - 搜索二维矩阵 II
  + https://leetcode-cn.com/problems/search-a-2d-matrix-ii/solution/zhe-xian-sou-suo-by-jason-2-6gk1/

### 关键思路

从左到右、从上到下分别升序，可从**左下**往**右上**（反向也可）搜索，每次剔除一行或一列。

* 当前数大于target则往上，当前数小于target则往右。
* 注意终止条件和边界条件

### 复杂度

* 时间复杂度：O(m+n)。其中m为数组行数，n为列数。
* 空间复杂度：O(1)。

### 代码

| **0240 搜索二维矩阵 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148676177/ |
| 1. class Solution { 2. public: 3. bool searchMatrix(vector<vector<int>>& matrix, int target) { 4. int row\_num = matrix.size(); 5. if (row\_num==0) 6. return false; 7. int col\_num = matrix[0].size(); 8. int r = row\_num-1, c = 0; 9. int val; 10. while (r>=0 && c<col\_num) { 11. val = matrix[r][c]; 12. if (val == target) 13. return true; 14. else if (val < target) 15. ++c; 16. else 17. --r; 18. } 19. return false; 20. } 21. }; |

# 数学

## 0007 整数反转

链接：https://leetcode-cn.com/problems/reverse-integer/

标签：数学

### 精选题解

* 官方题解 - 整数反转
  + <https://leetcode-cn.com/problems/reverse-integer/solution/zheng-shu-fan-zhuan-by-leetcode/>
  + https://leetcode-cn.com/problems/reverse-integer/solution/zheng-shu-fan-zhuan-by-leetcode/82512

### 关键思路

* 每次对x模10，得到最低位数字pop，利用公式res=res\*10+pop得到反转后的整数。
* 需要考虑溢出。也即需要判断结果res是否在res\*10+pop后溢出，以正整数为例，当res > INT\_MAX/10，或者res == INT\_MAX/10 且 pop > INT\_MAX%10时，结果会溢出（代码第8行）；负整数类似（代码第10行）。
* 事实上，对于32位有符号整数，最大值INT\_MAX为2,147,483,647（231-1），最小值INT\_MIN为-2,147,483,648（-231），故INT\_MAX%10的值为7，INT\_MIN%10的值为-8，因此有些解答也直接使用这两个硬编码的数。

### 复杂度

* 时间复杂度：O(log(x))。x中数字个数为 。
* 空间复杂度：O(1)。

### 代码

| **0007 整数反转.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/139089583/ |
| 1. class Solution { 2. public: 3. int reverse(int x) { 4. int res = 0; 5. while (x != 0) { 6. int pop = x % 10; 7. x = x / 10; 8. if (res>INT\_MAX/10 || (res==INT\_MAX/10 && pop > INT\_MAX%10)) 9. return 0; 10. if (res<INT\_MIN/10 || (res==INT\_MIN/10 && pop < INT\_MIN%10)) 11. return 0; 12. res = res \* 10 + pop; 13. } 14. return res; 15. } 16. }; |

## 0009 回文数

链接：<https://leetcode-cn.com/problems/palindrome-number/>

标签：数学

### 精选题解

* 官方题解 - 回文数
  + https://leetcode-cn.com/problems/palindrome-number/solution/hui-wen-shu-by-leetcode-solution/

### 关键思路

* 如果对整个数进行反转，判断是否与原数相等，可能会发生溢出的问题。
* 因此考虑反转数字的后一半，如果反转后的数和前一半相等，则为回文数。
  + 需要考虑位数是奇数还是偶数。
  + 剪枝：x为负，或者个位为0且自身非0。

### 复杂度

* 时间复杂度：O(logn)。循环次数为x的位数（除以二）。
* 空间复杂度：SO(1)。

### 代码

| **0009 回文数.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140135642/ |
| 1. class Solution { 2. public: 3. bool isPalindrome(int x) { 4. if (x<0 || (x%10==0 && x!=0)) 5. return false; 6. int rev = 0; 7. while (rev < x) { 8. rev = rev \* 10 + x % 10; 9. x = x / 10; 10. } 11. return (x == rev || x == rev/10); 12. } 13. }; |

## 0050 Pow(x, n)

链接：<https://leetcode-cn.com/problems/powx-n/>

标签：数学，二分，位运算

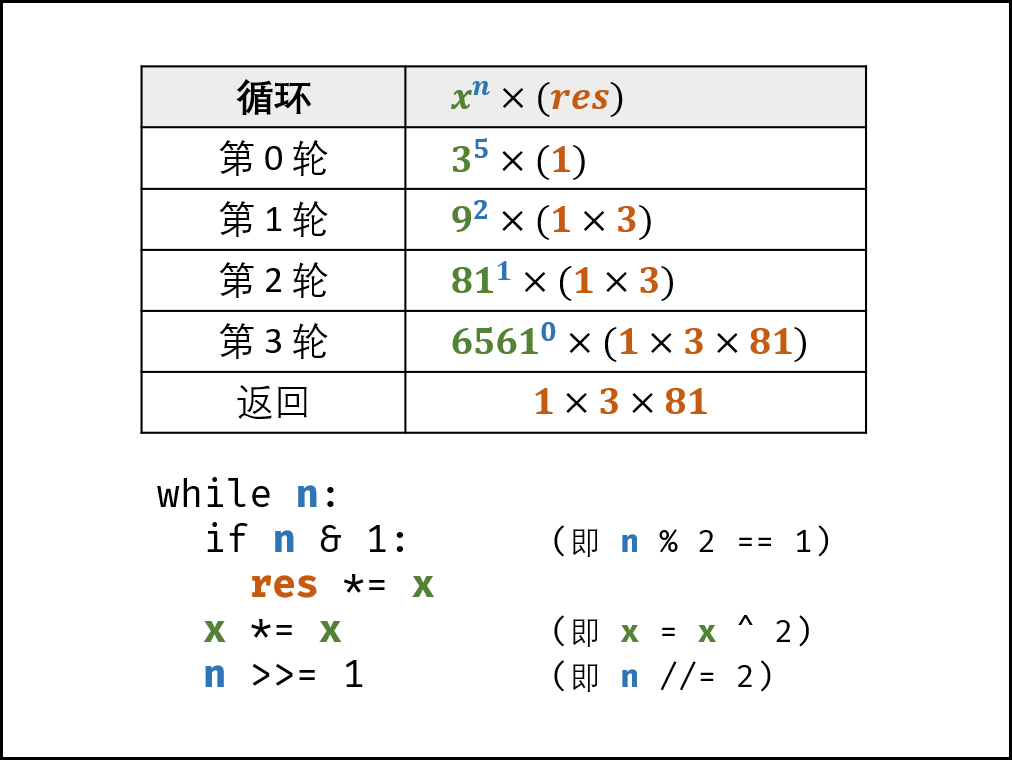
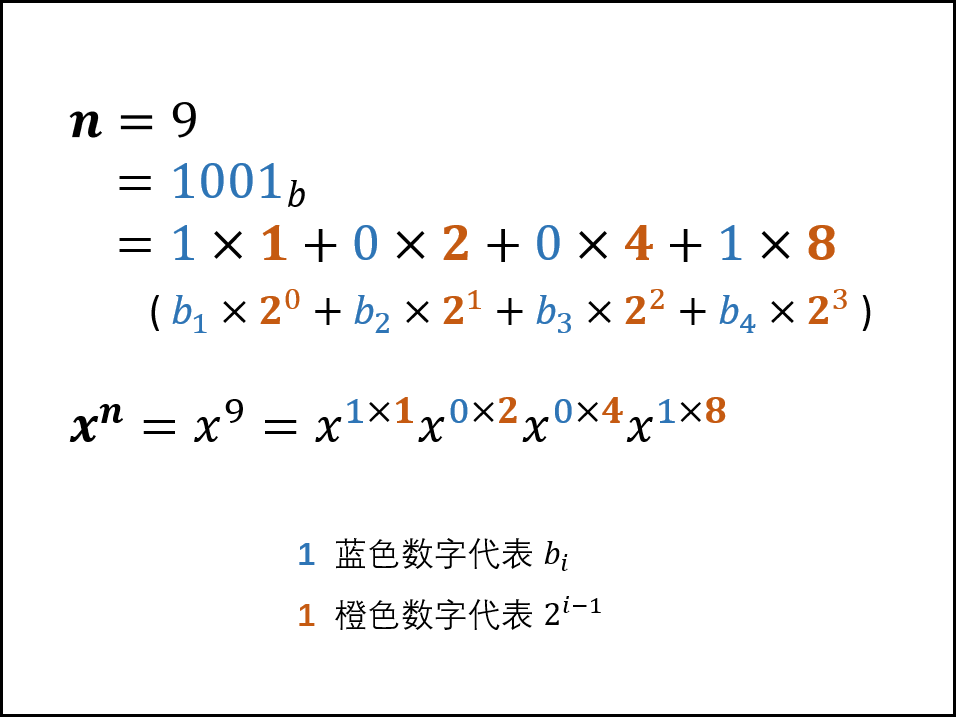
### 精选题解

* 官方题解 - Pow(x, n)
  + <https://leetcode-cn.com/problems/powx-n/solution/powx-n-by-leetcode-solution/>
* 50. Pow(x, n) （快速幂，清晰图解） - Pow(x, n)
  + https://leetcode-cn.com/problems/powx-n/solution/50-powx-n-kuai-su-mi-qing-xi-tu-jie-by-jyd/

### 关键思路

实际上就是快速幂。有两种理解角度，一种是对n进行二进制表示，另一种是对乘法进行分治。分治方法可以结合递归的写法。本文则从二进制角度思考，采用迭代写法。

* 假设n的二进制表示为 ，那么n的十进制表示为 ，则 的十进制表示为 ，更进一步地，只保留 中值为1的项，也即 ，其中 。
* 所以只需要求解两个子问题：
  + 计算 的值，其中
* 循环执行 x = x \* x 即可
  + 获取 的值，其中 ：
* 通过n&1判断n的二进制表示中最后一位是否为1，若为1则将res乘上此时的x
* 通过n>>1不断向右移位，找到为1的二进制位。
* 如果n为负，则计算 也即 即可。
  + 由于INT\_MIN的绝对值比INT\_MAX大1，因此需要单独处理，也即。



### 复杂度

* 时间复杂度：O(logn)。n的二进制位数。
* 空间复杂度：O(1)。

### 代码

| **0050 pow(x,n).cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140385874/ |
| 1. class Solution { 2. public: 3. double myPow(double x, int n) { 4. if (x==0) 5. return 0; 6. double res=1; 7. if (n<0) { 8. if (n==INT\_MIN) { *// 最小的负数会溢出* 9. return myPow(x,n+1) \* x; 10. } else { 11. x = 1/x; 12. n = -n; 13. } 14. } 15. while (n) { 16. if (n&1) { *// 如果该位为 1，则将 res 乘上此时的 x* 17. res \*= x; 18. } 19. x \*= x; 20. n >>= 1; 21. } 22. return res; 23. } 24. }; |

## 0069 x的平方根

链接：<https://leetcode-cn.com/problems/sqrtx/>

标签：数学，二分

### 精选题解

* 官方题解 - x 的平方根 - 力扣（LeetCode）
  + <https://leetcode-cn.com/problems/sqrtx/solution/x-de-ping-fang-gen-by-leetcode-solution/>

### 关键思路

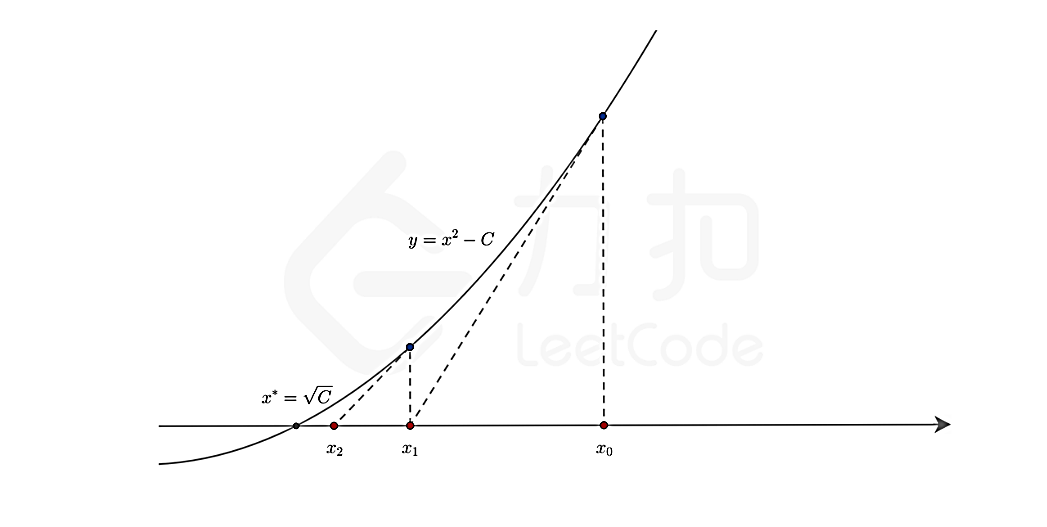
二分法和牛顿迭代法。

二分法：

* 下界low为0，上界high为x，计算mid\*mid是否比x大
* 由于mid\*mid可能会溢出，因此在做乘法时需要事先将mid转换成long long，也即

(long long) mid \* mid <= x

牛顿迭代法：（更详尽的过程可见官方题解）



* 令 为传入的参数x（不是后面函数中的x），那么 的平方根就是函数 的零点。
* 牛顿法的核心思路：利用泰勒级数，从初值向零点快速逼近。
  + 用图形语言来讲，就是从函数曲线上的点 作切线与 轴相交于 。
  + 新的交点 比旧的交点 离零点更近。
  + 经过多次迭代就能得到一个无比接近零点的交点，通常认为其与零点的插值小于 就足够了。
* 选取迭代初值 。
  + 已知 的两个零点分别为 和 。
  + 该函数为下凸函数，不断逼近零点且不会越过零点，为取到正根，初值取*C*。
* 设前一次迭代的零点为 ，则函数上该点为 ，切线斜率为 ，切线方程为 ，得新零点 。

### 复杂度

* 时间复杂度：O(logx)。二分法的迭代次数。牛顿迭代法也是这个时间复杂度，但是由于是二次收敛，因此比二分法更快。
* 空间复杂度：O(1)。

### 代码1 – 二分法

| **0069 x的平方根.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140390739/ |
| 1. class Solution { 2. public: 3. int mySqrt(int x) { 4. int res = 0; 5. int low = 0, high = x; 6. while (low <= high) { 7. int mid = low + (high-low)/2; 8. if ((long long) mid\*mid <= x) { *// 防止整型溢出，做乘法时转成 long long* 9. res = mid; 10. low = mid + 1; 11. } else { 12. high = mid - 1; 13. } 14. } 15. return res; 16. } 17. }; |

### 代码2 – 牛顿迭代法

| **0069 x的平方根 - 牛顿迭代法.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140579199/ |
| 1. class Solution { 2. public: 3. int mySqrt(int x) { 4. if (x == 0) 5. return 0; 6. double C = x, x\_old = x, x\_new; 7. while (1) { 8. x\_new = 0.5 \* (x\_old + C/x\_old); 9. if (fabs(x\_new - x\_old) < 1e-7) 10. break; 11. x\_old = x\_new; 12. } 13. return int(x\_old); 14. } 15. }; |

## 0171 Excel表列序号

链接：<https://leetcode-cn.com/problems/excel-sheet-column-number/>

标签：数学

### 精选题解

略。

### 关键思路

略。

### 复杂度

略。

### 代码

| **0171 Excel表列序号.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140581112/ |
| 1. class Solution { 2. public: 3. int titleToNumber(string s) { 4. int res = 0; 5. for (auto &ch: s) { 6. res = res\*26 + (ch-'A'+1); 7. } 8. return res; 9. } 10. }; |

## 0172 阶乘后的零

### 精选题解

* 详细通俗的思路分析 - 阶乘后的零
  + https://leetcode-cn.com/problems/factorial-trailing-zeroes/solution/xiang-xi-tong-su-de-si-lu-fen-xi-by-windliang-3/

### 关键思路

* 数字中0的个数取决于2\*5的对数，而因子2的个数远大于因子5的个数，故只需求因子5的个数。
* 5，52，53，…，5k分别包含1，2，……，k个因子5，也即每隔5个数有1个因子5，每隔52个数有2个因子5，每隔5k个数有k个因子5。
* 只需不断对n除以5，并累加上除的结果，就能知道因子5的总个数（代码5 ~ 7行）。

### 复杂度

* 时间复杂度：O(logn)。
* 空间复杂度：O(1)。

### 代码

| **0172 阶乘后的零.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140584272/ |
| 1. class Solution { 2. public: 3. int trailingZeroes(int n) { 4. int res = 0; 5. while (n>0) { 6. res += n/5; 7. n /= 5; 8. } 9. return res; 10. } 11. }; |

## 0202 快乐数

链接：<https://leetcode-cn.com/problems/happy-number/>

标签：哈希表，数学，双指针

### 精选题解

* 官方题解 - 快乐数
  + <https://leetcode-cn.com/problems/happy-number/solution/kuai-le-shu-by-leetcode-solution/>
* 使用“快慢指针”思想找出循环，不要使用集合或递归！！ - 快乐数
  + <https://leetcode-cn.com/problems/happy-number/solution/shi-yong-kuai-man-zhi-zhen-si-xiang-zhao-chu-xun-h/>
* 202. 快乐数:【set在哈希法中的应用】详解 - 快乐数
  + <https://leetcode-cn.com/problems/happy-number/solution/202-kuai-le-shu-setzai-ha-xi-fa-zhong-de-ying-yong/>

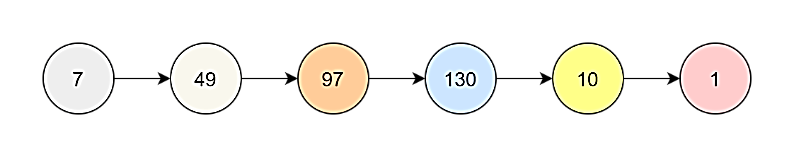
### 关键思路

两种方法：

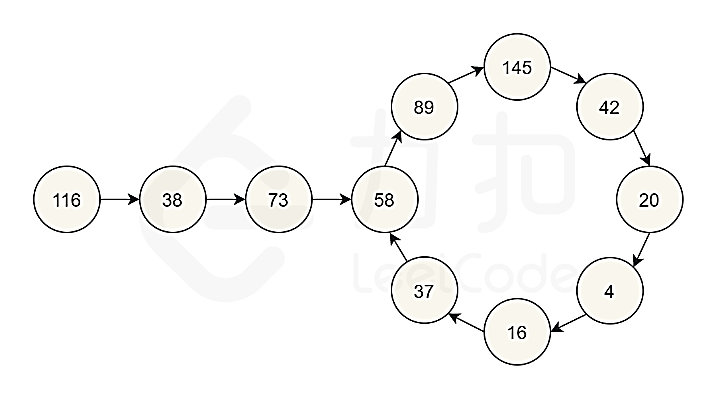
* 哈希表。
  + 用哈希表unordered\_set<int> set记录出现过的数（代码1第13行）。
  + 如果该数为1，则返回true（代码1第15 ~ 16行）；
  + 如果新产生的数在set（不含1）中，则表明存在循环，否则将其加入set中（代码1第18 ~ 21行）。
* 快慢指针。将每个计算得到的数视为链表上的节点，该题则转化为检测链表是否有环。
  + 如果n是快乐数，即最终会落到1，那么快指针会比慢指针先到1；
  + 如果n不是快乐数，那么快指针和慢指针会在环中的某个数上相遇。
  + 也即Floyd算法（代码2第15 ~ 17行），可参考“0141 环形链表”。

每次计算各位数的平方和作为下一个数，最终会有三种情况：

* 落到1。这种情况为快乐数。



* 进入循环，永远不会到1。



* 值越来越大，达到无穷大。这种情况不存在。3位数不超过243，4位或4位以上的数每次计算后都会减少，一直减少到3位数为止。所以无论如何，值都不会变到无穷大。

|  |  |  |
| --- | --- | --- |
| **位数** | **该位数下最大数** | **各位平方和** |
| 1 | 9 | 81 |
| 2 | 99 | 162 |
| 3 | 999 | 243 |
| 4 | 9999 | 324 |
| 13 | 9999999999999 | 1053 |

### 复杂度

详细可看官方题解。

* 时间复杂度：O(logn)。
  + 计算一个n位数各位平方和的时间成本为O(logn)，也即和其位数线性相关。
  + 当位数小于3时，可以确定一定是常数复杂度的；位数大于3时，每一次新的计算都是在外面再嵌套一层log，这些个log嵌套后，总复杂度还是log，也即O(243⋅3+logn+loglogn+logloglogn)... = O(log n)O(logn)。
  + 快慢指针法约为哈希表法的3倍，因为每次循环要计算3次各位平方和。
* 空间复杂度：
  + 哈希表法为O(logn)。当n足够大时，主要取决于n；当n较小时，位数最终都不超过3位，因此循环的次数有限，退化成O(1)。
  + 快慢指针法为O(1)。保存快慢指针。

### 代码1 – 哈希表

| **0202 快乐数 - 哈希表.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140605079/ |
| 1. class Solution { 2. public: 3. int squareSum(int x) { 4. int sum = 0; 5. while (x>0) { 6. sum += (x%10) \* (x%10); *// 注意加上括号，运算优先级* 7. x /= 10; 8. } 9. return sum; 10. } 11. bool isHappy(int n) { 12. unordered\_set<int> set; 13. while (1) { 14. if (n==1) 15. return true; 16. n = squareSum(n); 17. if (set.find(n) != set.end()) { 18. return false; 19. } else { 20. set.insert(n); 21. } 22. } 23. } 24. }; |

### 代码2 – 快慢指针

| **0202 快乐数 - 快慢指针.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/140599733/ |
| 1. class Solution { 2. public: 3. int squareSum(int x) { 4. int sum = 0; 5. while (x>0) { 6. sum += (x%10) \* (x%10); *// 注意加上括号，运算优先级* 7. x /= 10; 8. } 9. return sum; 10. } 11. bool isHappy(int n) { 12. int slow = n, fast = n; 13. do { 14. slow = squareSum(slow); 15. fast = squareSum(squareSum(fast)); 16. } while (slow != fast); 17. return slow == 1; 18. } 19. }; |

## 0231 2的幂

链接：https://leetcode-cn.com/problems/power-of-two/

标签：数学，位运算

### 精选题解

* 2 的幂 （位运算，极简解法+图表解析） - 2的幂
  + <https://leetcode-cn.com/problems/power-of-two/solution/power-of-two-er-jin-zhi-ji-jian-by-jyd/>
  + <https://leetcode-cn.com/problems/power-of-two/solution/power-of-two-er-jin-zhi-ji-jian-by-jyd/251869>

### 关键思路

满足n&(n-1)==0的数即为2的幂。

* 充分性：n & n-1把n最低位的1变0，若n & n-1 == 0，说明n只有一个1，也即n为2的幂。
* 必要性：若n为2的幂，则n的最高位为1、其余位为0，n-1的最高位为0，其余位为1，则n&(n-1)==0。

### 复杂度

* 时间复杂度：O(1)。
* 空间复杂度：O(1)。

### 代码

| **0231 2的幂.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142118868/ |
| 1. class Solution { 2. public: 3. bool isPowerOfTwo(int n) { 4. return n>0 && (n&(n-1))==0; 5. } 6. }; |

## 0326 3的幂

链接：<https://leetcode-cn.com/problems/power-of-three/>

标签：数学

### 精选题解

* 官方题解 - 3的幂
  + https://leetcode-cn.com/problems/power-of-three/solution/3de-mi-by-leetcode/

### 关键思路

* n>=3时，当余数模3为0时一直除以3，到终止时，只有n==1返回true，其余情况都是false。
* n < 3（包含负数）时，只有n==1是3的幂。所以边界条件简化成n<1则返回false。

### 复杂度

* 时间复杂度：O（logn）。
* 空间复杂度：O(1)。

### 代码

| **0326 3的幂.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142115739/ |
| 1. class Solution { 2. public: 3. bool isPowerOfThree(int n) { 4. if (n < 1) 5. return false; 6. while (n % 3 == 0) { 7. n /= 3; 8. } 9. return n == 1; 10. } 11. }; |

## 0258 各位相加

链接：<https://leetcode-cn.com/problems/add-digits/>

标签：数学

### 精选题解

* 详细通俗的思路分析，多解法 - 各位相加
  + <https://leetcode-cn.com/problems/add-digits/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-5-7/>
* 如何证明一个数的数根(digital root)就是它对9的余数？ - 知乎
  + https://www.zhihu.com/question/30972581

### 关键思路

一种是暴力解法。两层while循环，外循环记录内循环各位相加的结果，内循环不断对10取模相加，最终外循环得到小于10的数，即为此题结果。

另一种是数学公式。题中所求的通常称为数根。先列一些数找规律：

原数: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

数根: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 1 2

可以看到，num的树根为 (num - 1) % 9 + 1。这种先减1再加1的写法是为了照顾各种边界情况。也就是说，一个数模9，和它的各位之和模9，结果相同。

证明可以看上面的知乎回答链接，这里简要写一下：将该数表示为 ，其中 表示每一位上的数字，又 ，所以 。

### 复杂度

* 时间复杂度：O(logn)/O(1)。暴力法的复杂度为O(logn+log(logn)+…) = O(logn)。公式法的复杂度为O(1)。
* 空间复杂度：O(1)。

### 代码1 – 暴力解法

| **0258 各位相加 – 暴力.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141361715/ |
| 1. class Solution { 2. public: 3. int addDigits(int num) { 4. while (num >= 10) { 5. int next = 0; 6. while (num > 0) { 7. next += num % 10; 8. num = num / 10; 9. } 10. num = next; 11. } 12. return num; 13. } 14. }; |

### 代码2 – 公式法

| **0258 各位相加 – 公式.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141362718/ |
| 1. class Solution { 2. public: 3. int addDigits(int num) { 4. return (num-1) % 9 + 1; 5. } 6. }; |

## 0268 丢失的数字

链接：<https://leetcode-cn.com/problems/missing-number/>

标签：数学，位运算

### 精选题解

* 官方题解 - 丢失的数字
  + https://leetcode-cn.com/problems/missing-number/solution/que-shi-shu-zi-by-leetcode/

### 关键思路

两种比较好的方法。

* 一种是利用求和公式 ，先算出和，再逐个减掉数组中的数。
* 另一种是位运算，一个数异或两次得到全0。这里有个小技巧，设res初值为n，则将res不断异或上i^nums[i]，其中i从0到n-1。这样相当于异或了(n+1)+n个数，最后只异或了一次的就是丢失的数字。代码使用了这个方法。

### 复杂度

* 时间复杂度：O(n)。无论是求和公式还是位运算，都要遍历n个数。
* 空间复杂度：O(1)。

### 代码

| **0268 丢失的数字.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141433837/ |
| 1. class Solution { 2. public: 3. int missingNumber(vector<int>& nums) { 4. int res = nums.size(); 5. for (int i=0; i<nums.size(); ++i) { 6. res ^= i ^ nums[i]; 7. } 8. return res; 9. } 10. }; |

## 0470 用 Rand7() 实现 Rand10()

链接：https://leetcode-cn.com/problems/implement-rand10-using-rand7/

标签：数学

### 精选题解

* 从最基础的讲起如何做到均匀的生成随机数 - 用 Rand7() 实现 Rand10()
  + https://leetcode-cn.com/problems/implement-rand10-using-rand7/solution/cong-zui-ji-chu-de-jiang-qi-ru-he-zuo-dao-jun-yun-/

### 关键思路

### 复杂度

### 代码

## 0728 自除数

链接：https://leetcode-cn.com/problems/self-dividing-numbers/

标签：数学

### 精选题解

* 官方题解 - 自除数
  + <https://leetcode-cn.com/problems/self-dividing-numbers/solution/zi-chu-shu-by-leetcode/>
  + <https://leetcode-cn.com/problems/self-dividing-numbers/solution/zi-chu-shu-by-leetcode/744077>

### 关键思路

见代码。

### 复杂度

* 时间复杂度：O(nlogn)。区间内共有n个数，每个数至多除logn次。
* 空间复杂度：O(n)。结果数组大小。

### 代码

| **0728 自除数.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142134166/ |
| 1. class Solution { 2. public: 3. vector<int> selfDividingNumbers(int left, int right) { 4. vector<int> res; 5. for (int num=left; num<=right; ++num) { 6. int tmp = num; 7. int mod = 0; 8. while (tmp!=0) { 9. mod = tmp % 10; 10. if (mod==0 || num%mod!=0) 11. break; 12. tmp /= 10; 13. } 14. if (tmp==0) 15. res.push\_back(num); 16. } 17. return res; 18. } 19. }; |

# 链表

## 如何通过数组构造一个链表

参考“cs-notes/\_tests/\_utils.h”

### 关键思路

使用哑节点dum方便处理；记录尾节点tail用于创建循环链表。

### 代码

| **构造链表** |
| --- |
| 1. **struct** ListNode { 2. int val; 3. ListNode \*next; 4. ListNode(int x) : val(x), next(NULL) {} 5. }; 6. ListNode\* constructLinkedList(**const** std::vector<int>&v, int cycle\_pos = -1, bool is\_dum = false) { 7. *// construct linked list* 8. ListNode \*dum = new ListNode(0); 9. ListNode \*p = new ListNode(v[0]); 10. dum->next = p; 11. ListNode \*tail; 12. **for** (int i=1; i<v.size(); ++i) { 13. p->next = new ListNode(v[i]); 14. p = p->next; 15. tail = p; 16. } 17. *// tail node point to node* 18. p = dum->next; 19. **while** ((cycle\_pos--)>=0) { 20. tail->next = p; 21. p = p->next; 22. } 23. **if** (is\_dum) 24. **return** dum; 25. **else** 26. **return** dum->next; 27. } |

## 0002 两数相加

链接：<https://leetcode-cn.com/problems/add-two-numbers/>

标签：链表，递归，数学

### 精选题解

* 官方题解 - 两数相加
  + <https://leetcode-cn.com/problems/add-two-numbers/solution/liang-shu-xiang-jia-by-leetcode-solution/>
* 两数相加 – 两种解法
  + https://leetcode-cn.com/problems/add-two-numbers/solution/liang-shu-xiang-jia-by-gpe3dbjds1/

### 关键思路

当l1和l2至少一个非空时，就不停地将和保存到new出来的新节点，并将l1和l2的指针指向后一个节点。

注意每次都需要加上进位，并计算新的进位。

还需要检查最后一次求和是否产生进位，如果产生了，还需new一个新结点存放进位1。

使用哑结点可以避免边界条件：

1. ListNode \*dummy = new ListNode(-1);

### 复杂度

* 时间复杂度：O(max(n1,n2))。其中n1和n2分别为两个链表中的节点个数。
* 空间复杂度：SO(max(n1,n2))。作为结果返回的链表的结点个数。

### 代码

| **0002 两数相加.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/139058623/ |
| 1. /\*\* 2. \* Definition for singly-linked list. 3. \* struct ListNode { 4. \* int val; 5. \* ListNode \*next; 6. \* ListNode() : val(0), next(nullptr) {} 7. \* ListNode(int x) : val(x), next(nullptr) {} 8. \* ListNode(int x, ListNode \*next) : val(x), next(next) {} 9. \* }; 10. \*/ 11. **class** **Solution** { 12. **public**: 13. ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) { 14. ListNode \*dummy = **new** ListNode(-1); 15. ListNode \*p = dummy; 16. int sum = 0; 17. int carry = 0; 18. **while** (l1 || l2) { 19. sum = 0; 20. **if** (l1) { 21. sum += l1->val; 22. l1 = l1->next; 23. } 24. **if** (l2) { 25. sum += l2->val; 26. l2 = l2->next; 27. } 28. sum += carry; 29. carry = sum / 10; 30. p->next = **new** ListNode(sum % 10); 31. p = p->next; 32. } 33. **if** (carry>0) 34. p->next = **new** ListNode(1); 35. **return** dummy->next; 36. } 37. }; |

## 0021 合并两个有序链表

链接：https://leetcode-cn.com/problems/merge-two-sorted-lists/

标签：链表

### 精选题解

* 官方题解 - 合并两个有序链表
  + https://leetcode-cn.com/problems/merge-two-sorted-lists/solution/he-bing-liang-ge-you-xu-lian-biao-by-leetcode-solu/

### 关键思路

很简单。直接看代码。注意使用哑结点。

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(1)。

### 代码

| **0021 合并两个有序链表.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153331808/ |
| 1. class Solution { 2. public: 3. ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) { 4. ListNode\* dum = new ListNode(-1); 5. ListNode\* p = dum; 6. while (l1 && l2) { 7. if (l1->val < l2->val) { 8. p->next = l1; 9. l1 = l1->next; 10. } else { 11. p->next = l2; 12. l2 = l2->next; 13. } 14. p = p->next; 15. } 16. p->next = l1 ? l1 : l2; 17. return dum->next; 18. } 19. }; |

## 0023 合并k个升序链表

链接：https://leetcode-cn.com/problems/merge-k-sorted-lists/

标签：链表，分治，优先队列

### 精选题解

* C++ 优先队列&两两合并&分治合并（图解） - 合并K个升序链表
  + <https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/c-you-xian-dui-lie-liang-liang-he-bing-fen-zhi-he-/>
* Brief C++ solution with priority\_queue - LeetCode Discuss
  + <https://leetcode.com/problems/merge-k-sorted-lists/discuss/10543/Brief-C%2B%2B-solution-with-priority_queue>
* Merge K sorted linked lists | Set 1 - GeeksforGeeks
  + https://www.geeksforgeeks.org/merge-k-sorted-linked-lists/
* Merge k sorted linked lists | Set 2 (Using Min Heap) - GeeksforGeeks
  + https://www.geeksforgeeks.org/merge-k-sorted-linked-lists-set-2-using-min-heap/
* 合并 K 个排序链表
  + https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/leetcode-23-he-bing-kge-pai-xu-lian-biao-by-powcai/
* 官方题解 - 合并K个升序链表
  + <https://leetcode-cn.com/problems/merge-k-sorted-lists/solution/he-bing-kge-pai-xu-lian-biao-by-leetcode-solutio-2/>

### 关键思路1：优先队列

步骤如下：

* 依次将k个链表头插入优先队列
  + C++的优先队列默认是大根堆，堆顶是最大元素，因此需要修改比较器。
* 使用哑结点dum记录结果链表的头
* 队列非空时，不断将栈顶元素弹出，加入结果链表cur的末尾
  + 此时cur->next指向栈顶元素，所以需要更新cur为cur->next，指向最新弹出的元素在链表中的节点
  + 如果最新弹出的节点仍有后继cur->next，则将该后继加入队列；如果没有后继，则队列的大小就自动减1
* 返回dum->next

### 复杂度1：优先队列

* 时间复杂度：O(nlogk)。这里n为所有链表节点个数之和。优先队列的插入和删除复杂度均为O(logk)。
* 空间复杂度：O(k)。

### 代码1：优先队列

| **0023 合并K个升序链表 - 优先队列.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153375248/ |
| 1. class Solution { 2. public: 3. struct compare { 4. bool operator()(ListNode\* a, ListNode\* b) { 5. return a->val > b->val; *// min heap* 6. } 7. }; 8. ListNode\* mergeKLists(vector<ListNode\*>& lists) { 9. priority\_queue<ListNode\*, vector<ListNode\*>, compare> q; 10. for (auto & ele: lists) 11. if (ele) 12. q.push(ele); 13. ListNode\* dum = new ListNode(-1); 14. ListNode\* cur = dum; 15. while (!q.empty()) { 16. cur->next = q.top(); 17. cur = cur->next; 18. q.pop(); 19. if (cur->next) 20. q.push(cur->next); 21. } 22. return dum->next; 23. } 24. }; |

### 关键思路2-1：分治 - 递归

对k路链表进行分治，三步走：

* **分割：**每层递归将当前的i路链表分成前i/2路和后i/2路
* 处理：分别对前i/2路和后i/2路进行相同的递归
* 合并：使用合并2个有序链表，将上面得到的两路链表合并到一条

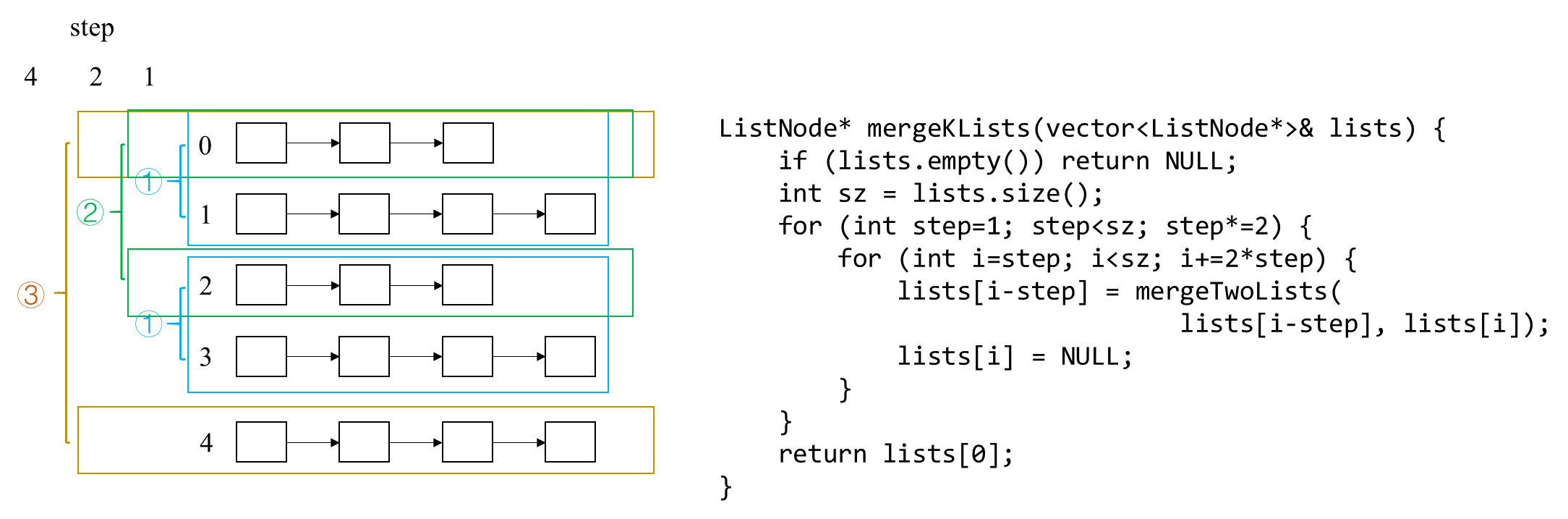
### 代码2-1：分治 – 递归

| **0023 合并K个升序链表 - 分治 递归.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153381918/ |
| 1. class Solution { 2. public: 3. ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) { 4. ListNode \*dum = new ListNode(-1); 5. ListNode \*p = dum; 6. while (l1 && l2) { 7. if (l1->val < l2->val) { 8. p->next = l1; 9. l1 = l1->next; 10. } else { 11. p->next = l2; 12. l2 = l2->next; 13. } 14. p = p->next; 15. } 16. p->next = l1 ? l1 : l2; 17. return dum->next; 18. } 19. ListNode\* merge(vector<ListNode\*> &lists, int L, int R) { 20. if (L>R) 21. return nullptr; 22. if (L==R) 23. return lists[L]; 24. int mid = L + (R-L)/2; 25. return mergeTwoLists(merge(lists, L, mid), merge(lists, mid+1, R)); 26. } 27. ListNode\* mergeKLists(vector<ListNode\*>& lists) { 28. return merge(lists, 0, lists.size()-1); 29. } 30. }; |

### 关键思路2-2：分治 – 迭代

两层循环：（看图更直观，建议拿笔画一画）

* 外循环对step进行迭代，从1开始，每次乘2
* 内循环对i进行迭代，从step开始，每次递增2\*step
  + 合并第i-step路和第i路链表，并且赋给i-step路
  + 将第i路链表置为nullptr
* 最后返回第0路链表
* 不要忘记判断lists为空的情况



### 复杂度2：分治 – 递归+迭代

* 时间复杂度：O(nlogk)。递归和迭代的复杂度分析一样。第1轮合并k/2组链表，每组2个链表，代价O(2\*n/k)，总代价即为O(n)；第2轮合并k/4组链表，每组4个链表，代价O(2\*2\*n/k)，总代价即为O(n）……共O(logk) 轮，故总的时间复杂度为O(nlogk)。
* 空间复杂度：递归为O(logk)，递归栈的最大深度；迭代为O(1)，直接在原链表上归并。

## 0025 K个一组翻转链表

链接：https://leetcode-cn.com/problems/reverse-nodes-in-k-group/

标签：链表

### 精选题解

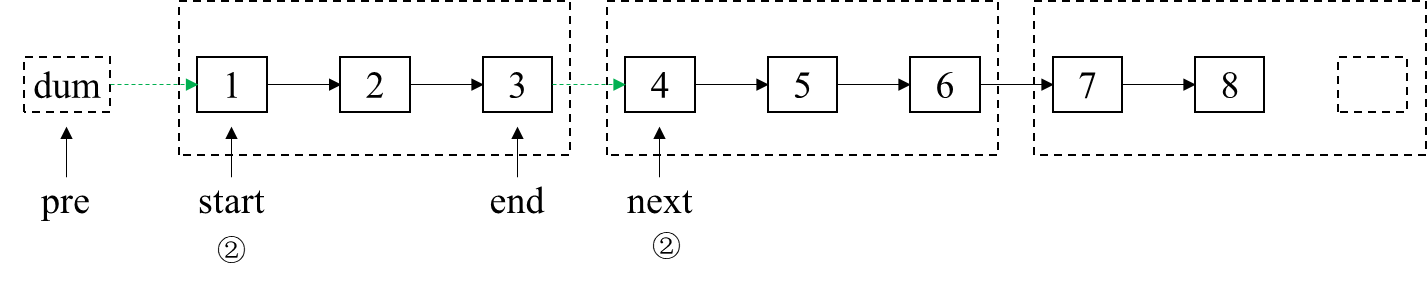
* 官方题解 - K 个一组翻转链表
  + https://leetcode-cn.com/problems/reverse-nodes-in-k-group/solution/k-ge-yi-zu-fan-zhuan-lian-biao-by-leetcode-solutio/

### 关键思路

首先new一个哑结点dum，其next指向head，用于最后返回结果。（代码第15 ~ 16行）

使用四个指针：（代码第17 ~ 18行）

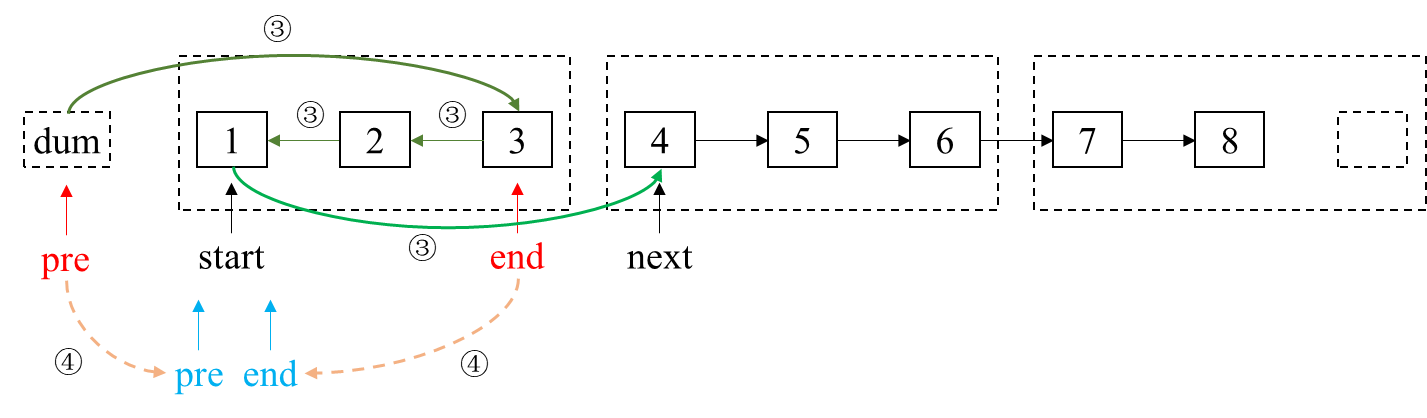
* start和end用于表示k个一组的节点的头尾节点
  + end初始时指向dum
* pre和next表示start的前一个节点和end的下一个节点
  + pre初始时指向dum



start和end，pre和next

分为如下几步：

* end右移k次（代码第20 ~ 21行）
  + 右移k次后，end应当指向k元节点组的最后一个节点，因此如果end为nullptr，表明该组节点数不足k，直接终止循环（代码第23行）
* 由于pre和end已有初值，因此更新next和start（代码第25 ~ 27行）
  + 更新start为pre->next
  + 更新next为end->next
  + 将end的next指向nullptr，也即断开当前k元节点组和下一个k元节点组的联系
* 反转k元节点组，并将其与前后连接（代码第29 ~ 30行）
  + pre->next指向反转后的k元节点组的头节点
  + start->next指向next
* 更新pre和end
  + pre和end都指向start，注意，此时start已经是前一个k元节点组的尾节点



### 复杂度

* 时间复杂度：O(n)。遍历所有节点。
* 空间复杂度：O(1)。

### 代码

| **0025 K个一组翻转链表.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/149355630/ |
| 1. class Solution { 2. public: 3. ListNode\* reverseLinkedList(ListNode\* head) { 4. ListNode\* left=nullptr, \*mid=head, \*right; 5. while (mid) { 6. right = mid->next; 7. mid->next = left; 8. left = mid; 9. mid = right; 10. } 11. return left; 12. } 13. ListNode\* reverseKGroup(ListNode\* head, int k) { 14. ListNode \*dum = new ListNode(0); 15. dum->next = head; 16. ListNode \*start, \*end=dum; *// first and last of k-group* 17. ListNode \*pre=dum, \*next; *// pre of start, next of end* 18. while (end->next) { 19. for (int i=0; i<k && end; ++i) 20. end = end->next; 21. *// not reach k nodes* 22. if (!end) break; 23. *// break current k-group from whole linked list* 24. start = pre->next; 25. next = end->next; 26. end->next = nullptr; 27. *// reverse k-group and connect to* 28. pre->next = reverseLinkedList(start); 29. start->next = next; 30. *// connect current (reversed) k-group with pre and next* 31. pre = start; 32. end = start; 33. } 34. return dum->next; 35. } 36. }; |

## 0141 环形链表

链接：https://leetcode-cn.com/problems/linked-list-cycle/

标签：链表，双指针

### 精选题解

略。

### 关键思路

参考下面的“0142 环形链表 II”，取第一次相遇前的代码即可。

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(1)。

### 代码

| **0141 环形链表.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/150637016/ |
| 1. class Solution { 2. public: 3. bool hasCycle(ListNode \*head) { 4. if (head==NULL) 5. return NULL; 6. ListNode\* slow = head; 7. ListNode\* fast = head; 8. while (fast) { 9. if (fast->next==NULL) 10. return false; 11. slow = slow->next; 12. fast = fast->next->next; 13. if (slow == fast) 14. return true; 15. } 16. return false; 17. } 18. }; |

## 0142 环形链表 II

链接：https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/huan-xing-lian-biao-ii-by-leetcode-solution/

标签：链表，双指针

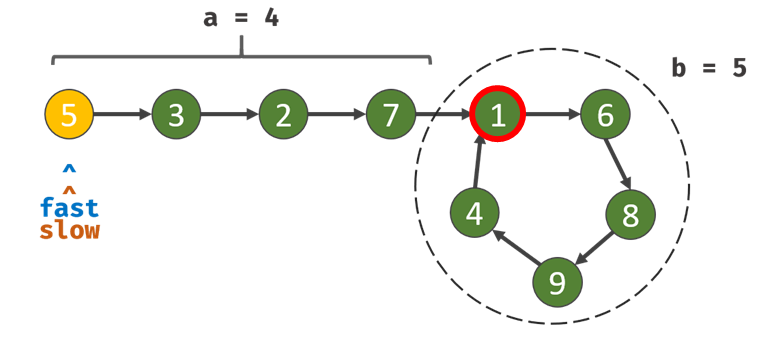
### 精选题解

* 官方题解 - 环形链表 II
  + <https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/huan-xing-lian-biao-ii-by-leetcode-solution/>
* 环形链表 II（双指针法，清晰图解） - 环形链表 II
  + https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/linked-list-cycle-ii-kuai-man-zhi-zhen-shuang-zhi-/

### 关键思路

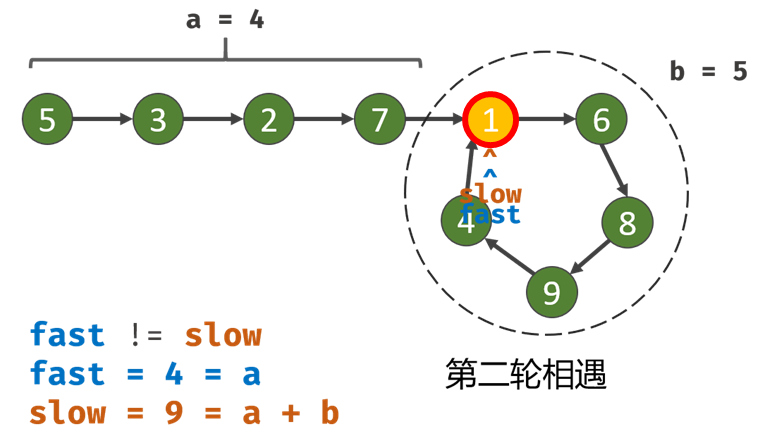
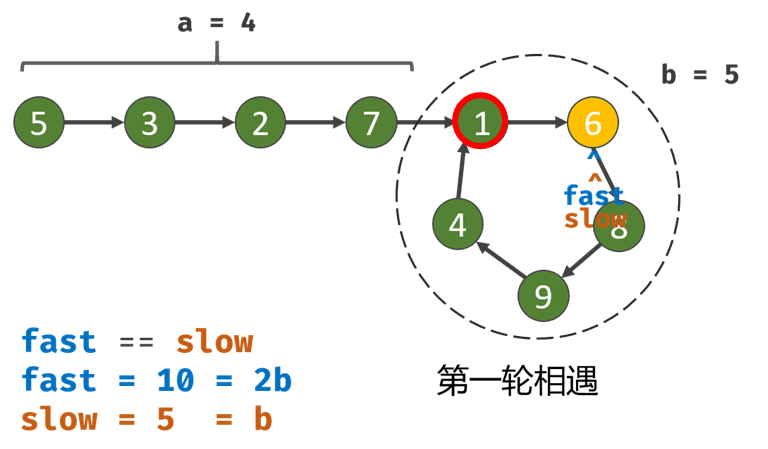
快慢指针fast和slow，二者均从head开始。slow每次右移一个节点，fast每次右移两个节点。若链表中存在环，则二者必然会相遇。

* 相遇的必然性可以这样直观理解：若fast落后slow一个节点，则下一次必相遇；若fast落后slow两个节点，则下一次只会落后一个节点，就和前面的情况相同，也必相遇。



解释一下之后要用到的几个变量的含义

* 距离：指经过的节点间的边的数目。比如右图中，从头节点5到入环位置1共有4条边，则距离为4。这也是为什么为了方便计算距离，我们的解法中快慢指针都从head开始出发（而不是有些题解中fast从head->next出发）
* a：环的入口节点到head的距离
* b：环的长度
* k：环的入口处到第一次相遇处的距离，也即下图中节点1到节点6的距离



该题的核心就是求出a：如果我们能知道a的值，则只需让某个指针从head开始走a步，就能恰好走到环的入口节点。我们想法设法建立起和a相关的等式。

第一次相遇时：

* 慢指针走过的距离为：
  + 这里用到一个事实：两个指针第一次相遇时，慢指针在环内不会超过一圈。当慢指针进入环时，快指针已经在环里了，无论快指针在环内什么位置，都会在慢指针下一次到达入环位置时，追上并重合。
* 快指针走过的距离为：
  + 这里的m是虚指，是任意的正整数，表示fast在环中走的圈数
* 由 ，也即 ，得

从图1可以看出，slow此时距环的入口为k，环长为b，再次走到环的入口还需b-k步。

* b-k这个值非常巧妙，恰好是上面得到的a的等式中的“多余的项”
* 换句话说，如果slow继续每次步进1个节点，则再走 步必然是在环的入口处，因为 恰好走了n-1次完整的环回到原地，多的b-k步则恰好了走到环的入口处。而slow在第一次相遇后继续走的这么多步，恰好等于a！

故我们就有如下方案：当两个指针第一次相遇时，将其中一个指针移动到head（这里为了节省一个指针的空间，可以把fast移动到这），然后两个指针都以1的步长向右移动，当两个指针再次相遇时，相遇的节点必然为环的入口。

* 严格的数学等式已经告诉我们，移动后的fast（步长为1）恰好走了a步，并不会多走

**注：本地的代码还实现了通过数组和pos构造链表并测试的函数，面试时也许有用。**

### 复杂度

* 时间复杂度：O(n)。这里n指链表的节点总个数，易知慢指针总共遍历 个节点，不超过2n，故为O(n)。
* 空间复杂度：O(1)。

### 代码

| **0142 环形链表 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/150633855/ |
| 1. class Solution { 2. public: 3. ListNode \*detectCycle(ListNode \*head) { 4. if (head==NULL) 5. return NULL; 6. ListNode\* slow = head; 7. ListNode\* fast = head; 8. while (fast!=NULL) { 9. if (fast->next==NULL) 10. return NULL; 11. slow = slow->next; 12. fast = fast->next->next; 13. if (fast==slow) { 14. fast = head; 15. while (fast != slow) { 16. slow = slow->next; 17. fast = fast->next; 18. } 19. return fast; 20. } 21. } 22. return NULL; 23. } 24. }; |

## 0146 LRU缓存机制

链接：https://leetcode-cn.com/problems/lru-cache/

标签：链表，哈希表

### 精选题解

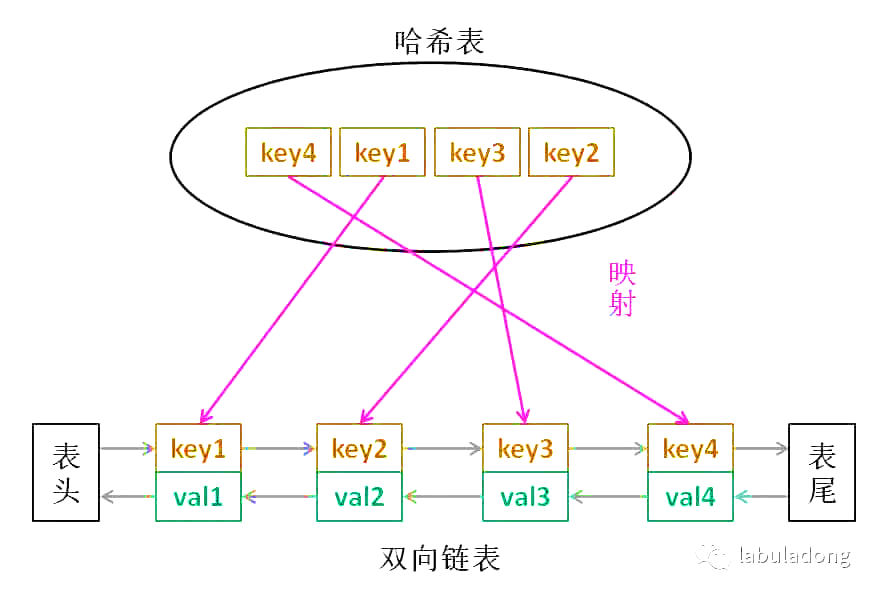
* 官方题解 - LRU 缓存机制：
  + <https://leetcode-cn.com/problems/lru-cache/solution/lruhuan-cun-ji-zhi-by-leetcode-solution/>
* LRU 策略详解和实现 - LRU 缓存机制：
  + <https://leetcode-cn.com/problems/lru-cache/solution/lru-ce-lue-xiang-jie-he-shi-xian-by-labuladong/>
  + <https://leetcode-cn.com/problems/lru-cache/solution/lru-ce-lue-xiang-jie-he-shi-xian-by-labuladong/162138>

### 关键思路

核心思路：哈希表+双向链表。

* 哈希表：将键映射到其在双向链表上的位置
* 双链表：存储键和数据值，靠近表头的是最近使用的，靠近表尾的是最久未使用的

所以首先使用哈希表定位键key在双链表中的位置，然后将其移动到双链表头部。



get：

* 若key不存在，返回 -1
* 若key存在，则先通过该哈希表定位该节点在双链表中的位置，然后将该节点移动到双链表头部（表头插入该节点内容，删除表中对应迭代器），更新哈希表中key对应的value，最后返回其数据值

put：

* 若key不存在
  + 若超出容量，则删除哈希表中对应的键，并删除双链表尾部的节点
  + 创建一个新节点 (key, value)，添加到双链表头部，并将key和位置加入到哈希表
* 若key存在，则类似get操作，先通过哈希表确定其在双链表中的位置，然后将该节点移动到双链表头部（删除该节点内容，表头插入构造的新节点），更新哈希表中key对应的信息

注意点：（可速记get和put主体代码块的行数为4/(6+3)）

* 删除、插入和更新操作容易出错，建议想清楚，或者多写几遍
  + 在双链表中移动完节点后，不要忘记更新其在哈希表中的信息
  + 注意在get中应先push\_front(\*map[key]) 再erase(map[key])
* 可以使用C++ STL，list是双链表，unordered\_map是哈希表
  + list中存储pair<int, int> 的键值对
  + unordered\_map中存储 <int, list<pair<int,int>>::iterator> 的键和迭代器对
  + 注意list中begin() 和front() 的区别：begin() 返回起始迭代器，front() 直接返回迭代器解引用后的内容
  + make\_pair不再需要提供后面的 <int,int> 模板类型，直接用make\_pair(key,value) 即可

### 复杂度

* 时间复杂度：O(1)。其他都易知，事实上，官方文档注明map.size() 的复杂度也为O(1)，dlist.size() 的复杂度官网没有写（有说法是C++11后从线性改成了常数，暂时懒得考证），所以代码中用map.size() 更为稳妥。
* 空间复杂度：O(n)。

### 代码

| **0146 LRU缓存机制.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/150501967/ |
| 1. class LRUCache { 2. private: 3. int cap; 4. list<pair<int,int>> dlist; 5. unordered\_map<int, list<pair<int,int>>::iterator> map; 6. public: 7. LRUCache(int capacity) : cap(capacity) { 8. } 10. int get(int key) { 11. if (map.count(key) == 0) 12. return -1; 13. dlist.push\_front(\*map[key]); 14. dlist.erase(map[key]); 15. map[key] = dlist.begin(); 16. return dlist.front().second; 17. } 19. void put(int key, int value) { 20. if (map.count(key) == 0) { 21. if (map.size() == cap) { 22. map.erase(dlist.back().first); 23. dlist.pop\_back(); 24. } 25. dlist.push\_front(make\_pair(key, value)); 26. map[key] = dlist.begin(); 27. } else { 28. dlist.erase(map[key]); 29. dlist.push\_front(make\_pair(key, value)); 30. map[key] = dlist.begin(); 31. } 32. } 33. }; |

## 0206 反转链表

链接：https://leetcode-cn.com/problems/reverse-linked-list/

标签：链表

### 精选题解

略。

### 关键思路

初始条件：

* mid指向head
* left指向nullptr，这是为了让原链表的头节点在反转后指向nullptr
* right无需赋初值，而是在下面的反转步骤中更新

反转步骤：（注意循环赋值的模式，已用颜色标注出来）

① 由于后续要修改mid和left，因此首先通过mid保存right

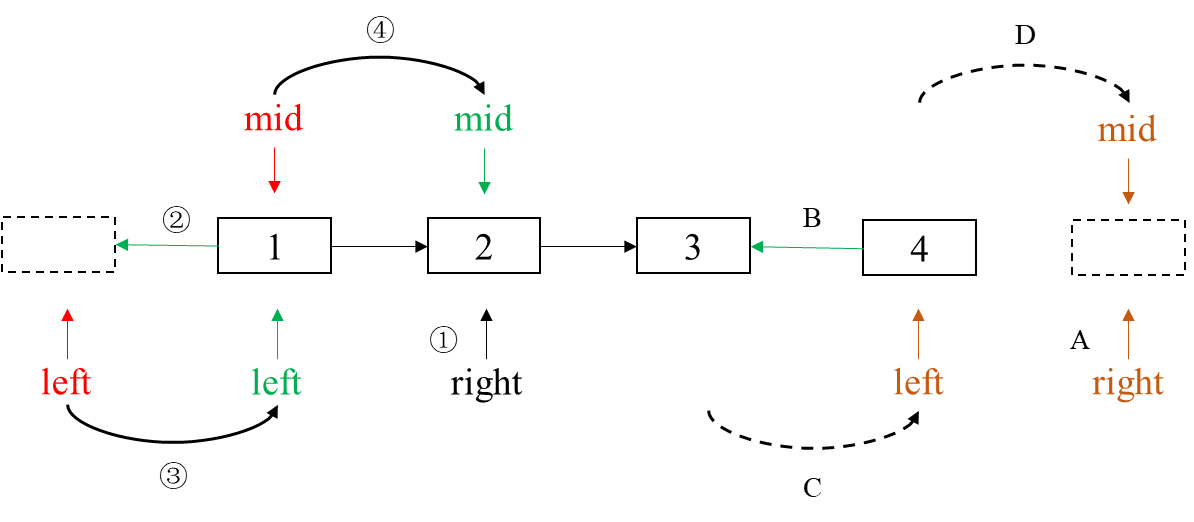
② 反转left和mid之间的指针方向，也即修改mid->next

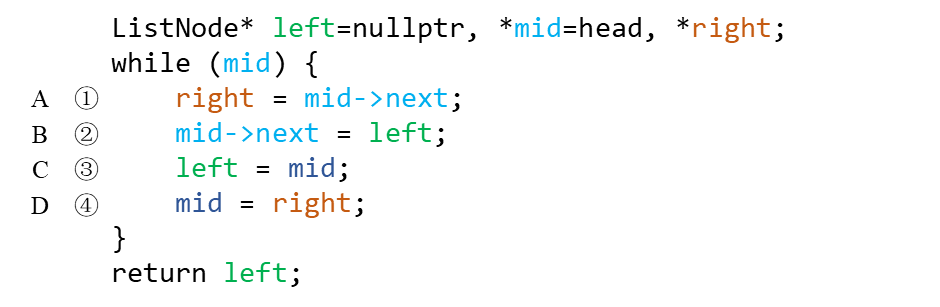
③ 将left右移到mid位置

④ 将mid右移到right位置

终止条件：

* 由于最后一次，mid指向nullptr，因此以while(mid) 作为终止判断条件
* 此时left指向原链表的最后一个节点，也是新链表的头节点，故返回left
* 这种方式无需判断各种边界条件（空节点或仅有一个节点）





### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(1)。

### 代码

| **0206 翻转链表.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/149299253/ |
| 1. class Solution { 2. public: 3. ListNode\* reverseList(ListNode\* head) { 4. ListNode\* left=nullptr, \*mid=head, \*right; 5. while (mid) { 6. right = mid->next; 7. mid->next = left; 8. left = mid; 9. mid = right; 10. } 11. return left; 12. } 13. }; |

## 0445 两数相加 II

链接：https://leetcode-cn.com/problems/add-two-numbers-ii/

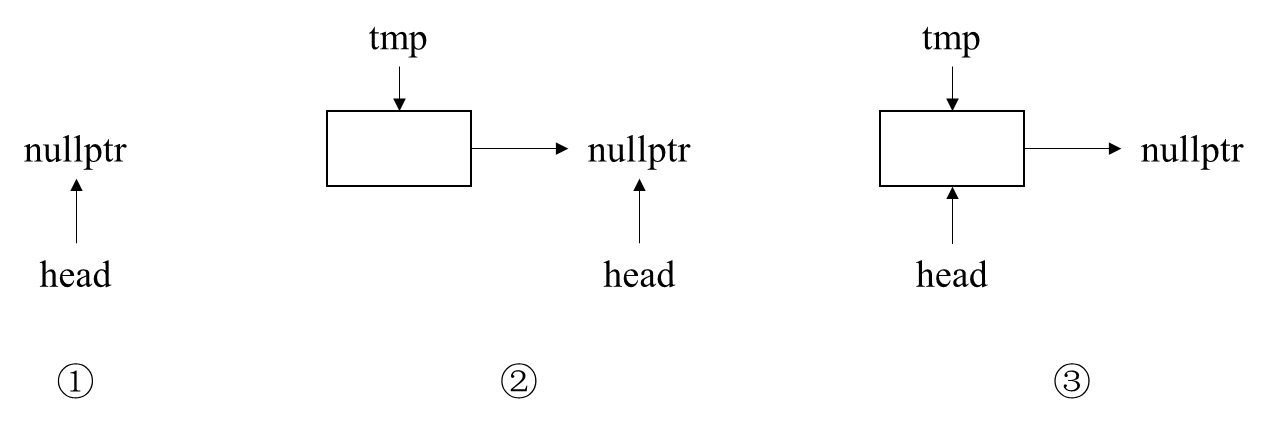
### 精选题解

* 官方题解 - 两数相加 II
* <https://leetcode-cn.com/problems/add-two-numbers-ii/solution/liang-shu-xiang-jia-ii-by-leetcode-solution/>

### 关键思路

与“0002 两数相加”基本类似，只是如下几点不同：

1. 由于头结点存储的是最高位的值，因此需要使用栈这种数据结构，先将列表入栈，再求和（代码第12 ~ 20行）；
2. 在求和时是从低位到高位，而生成的结果链表还应该是高位为头、低位为尾，因此结点需要反方向连接，如图①②③顺序所示（代码第36 ~ 38行）
3. 对循环进行了优化，将carry>0的判断条件也加上了，这样在循环结束后，就无需额外写判断carry值并新增结点的操作（代码第24行）；



### 复杂度

* 时间复杂度：O(max(n1,n2))。其中n1和n2分别为两个链表中的节点个数。
* 空间复杂度：SO(n1+n2)。作为结果返回的链表的结点个数为O(max(n1,n2))；栈空间O(n1+n2)。

### 代码

| **0445 两数相加 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/139069295/ |
| 1. **class** **Solution** { 2. **public**: 3. ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) { 4. stack<int> s1, s2; 5. **while** (l1) { 6. s1.push(l1->val); 7. l1 = l1->next; 8. } 9. **while** (l2) { 10. s2.push(l2->val); 11. l2 = l2->next; 12. } 13. ListNode \*head = **nullptr**; 14. int sum = 0; 15. int carry = 0; 16. **while** (!s1.empty() || !s2.empty() || carry>0) { 17. sum = 0; 18. **if** (!s1.empty()) { 19. sum += s1.top(); 20. s1.pop(); 21. } 22. **if** (!s2.empty()) { 23. sum += s2.top(); 24. s2.pop(); 25. } 26. sum += carry; 27. carry = sum / 10; 28. **auto** tmp = **new** ListNode(sum%10); 29. tmp->next = head; 30. head = tmp; 31. } 32. **return** head; 33. } 34. }; |

# 树

## 如何通过数组构造二叉树

### 参考链接

* 请问 [1, null, 2, 3] 在二叉树测试用例中代表什么 - 力扣（LeetCode）
  + https://support.leetcode-cn.com/hc/kb/article/1194353/
* LeetCode二叉树构造方法，通过一维数组直接构建完整的二叉树（按照LeetCode的格式）
  + <https://blog.csdn.net/USTCsunyue/article/details/106148317>

### 关键思路

本质上就是由层序遍历数组恢复出二叉树，只不过比标准的层序遍历多出了一个null用于标记空节点，由于vector只能使用相同的数据类型，因此这里用 INT\_MIN代替。

这里包含几个关键变量：

* v：输入的数组，INT\_MIN表示空节点
* (vector) aq：父节点队列（上一层）
* (vector) bq：当前节点队列（当前层）
* (int) i：数组索引
* (int) bq\_max：当前层节点队列可填充节点的最大个数
* (int) bq\_filled：当前层节点队列已填充节点的个数
* (int) LR：标记下个节点是插入左孩子还是右孩子

当bq\_filled < bq\_max时：（代码25 ~ 37行）

* 若节点非空，则将当前数组元素new为新的TreeNode，根据LR的值添加为其父节点的左孩子或右孩子
  + 父节点在aq中，其索引为bq\_filled/2
* 若节点为空，不需要操作
* 无论节点是否为空，最后都需要递增i和bq\_filled，以及切换LR（代码35 ~ 37行）

当bq\_filled == bq\_max 时：（代码39 ~ 44行）

* 将bq\_filled置为0，并将bq\_max置为bq.size()\*2
* 清空aq，并将bq的内容复制到aq中
* 清空bq

当i达到v的大小，则退出while循环，最后返回根结点root。

### 代码

| **构造二叉树** |
| --- |
| 1. **struct** TreeNode { 2. int val; 3. TreeNode \*left; 4. TreeNode \*right; 5. TreeNode() : val(0), left(nullptr), right(nullptr) {} 6. TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} 7. TreeNode(int x, TreeNode\* left, TreeNode\* right) : val(x), left(left), right(right) {} 8. }; 9. TreeNode\* constructTree(**const** std::vector<int> & v, bool is\_dum=false) { 10. *// TreeNode\* root = new TreeNode(v[0]);* 11. **if** (v.size()==0) **return** nullptr; 12. std::vector<TreeNode\*> aq; *// last level nodes queue* 13. std::vector<TreeNode\*> bq; *// current level nodes queue* 15. TreeNode\* root = new TreeNode(v[0]); *// root node* 16. aq.push\_back(root); 17. int i = 1; *// index of v* 18. int bq\_max = 2\*aq.size(); *// max num of nodes at current level* 19. int bq\_filled = 0; *// num of nodes already filled at current level* 20. int LR = 1; *// 1: add to left child | 2: add to right child* 22. **while** (i<v.size()) { 23. **if** (bq\_filled < bq\_max) { 24. **if** (v[i] != INT\_MIN) { 25. TreeNode\* tmp = new TreeNode(v[i]); 26. bq.push\_back(tmp); 27. **if** (LR==1) 28. aq[bq\_filled/2]->left = tmp; 29. **else** 30. aq[bq\_filled/2]->right = tmp; 31. } 32. *// if (v[i] == nullptr), also runs the codes below* 33. ++i; 34. ++bq\_filled; 35. LR = (LR==1) ? 2 : 1; 36. } **else** { 37. bq\_filled = 0; 38. bq\_max = bq.size()\*2; 39. aq.clear(); 40. **for** (int i=0; i<bq.size(); ++i) 41. aq.push\_back(bq[i]); 42. bq.clear(); 43. } 44. } 45. **return** root; 46. } |

## 0094 二叉树的中序遍历

链接：https://leetcode-cn.com/problems/binary-tree-inorder-traversal/

标签：树，栈

### 精选题解

* 官方题解 - 二叉树的中序遍历
  + https://leetcode-cn.com/problems/binary-tree-inorder-traversal/solution/er-cha-shu-de-zhong-xu-bian-li-by-leetcode-solutio/

### 关键思路

两种方法，递归和迭代。

递归没什么难度，注意res数组以引用传入递归即可。

迭代的核心思路：（建议拿纸笔画一画，有助于理解）

* 对每个节点，先将左节点压栈，再处理当前节点，最后指向右结点。

细节很多，非常容易写错。（建议多练两遍，文字描述有点绕，不如直接看代码）

* 一开始不需要将root压入stk中，因为内层循环会将root压栈
* 外层循环的判断条件是 (!stk.empty() || root)（代码2第6行）
  + stk开始为空，root开始非nullptr
  + 循环到后面root可能为nullptr，但只要stk中还非空，就继续循环
* 内层循环先将root压栈，之后继续向左向下，不断将左边的子节点压栈，直到最下层（代码2第7 ~ 9行）
* 当最左侧的节点遇到叶节点之后，开始处理栈顶节点，将其值放到res中，并弹出栈顶元素，最后将root指向其右子节点，重复上述流程（代码2第11 ~ 14行）

### 复杂度

* 时间复杂度：均为O(n)。
* 空间复杂度：均为O(n)。

### 代码1：递归

| **0094 二叉树的中序遍历 - 递归.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153002302/ |
| 1. class Solution { 2. public: 3. void recur(TreeNode\* root, vector<int>& res) { 4. if (root==nullptr) return; 5. recur(root->left, res); 6. res.push\_back(root->val); 7. recur(root->right, res); 8. } 9. vector<int> inorderTraversal(TreeNode\* root) { 10. vector<int> res; 11. recur(root, res); 12. return res; 13. } 14. }; |

### 代码2：迭代

| **0094 二叉树的中序遍历 - 迭代.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153028575/ |
| 1. class Solution { 2. public: 3. vector<int> inorderTraversal(TreeNode\* root) { 4. vector<int> res; 5. stack<TreeNode\*> stk; 6. while (!stk.empty() || root) { 7. while (root) { 8. stk.push(root); 9. root = root->left; 10. } 11. root = stk.top(); 12. stk.pop(); 13. res.push\_back(root->val); 14. root = root->right; 15. } 16. return res; 17. } 18. }; |

## 0102 二叉树的层序遍历

链接：https://leetcode-cn.com/problems/binary-tree-level-order-traversal/

标签：树，广搜，队列

### 精选题解

* 官方题解 - 二叉树的层序遍历
  + https://leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/er-cha-shu-de-ceng-xu-bian-li-by-leetcode-solution/
* **BFS 的使用场景总结：层序遍历、最短路径问题 - 二叉树的层序遍历**
  + <https://leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/bfs-de-shi-yong-chang-jing-zong-jie-ceng-xu-bian-l/>

### 关键思路

使用队列记录节点。可以使用优化的双端队列deque，包含在头文件 <deque> 中。

同时记录每一层的节点个数current\_level\_size，每次处理完一层就更新该数值。

节点出队时将其值加入该层数组v，同时将其左右孩子（如果有的话）入队。

最后别忘了每处理完一层就将该层数组v添加到二维数组res中。

注意队列的头部是q.front()，不是q.top()，栈才有这个用法。

**补充：强烈推荐阅读第二个链接中对于BFS使用场景的讲解↑**

“BFS 遍历”、“层序遍历”、“最短路径”实际上是递进的关系。

* 在BFS遍历的基础上区分遍历的每一层，就得到了层序遍历。
* 在层序遍历的基础上记录层数，就得到了最短路径。

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。

### 代码

| **0102 二叉树的层序遍历.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/150900062/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> levelOrder(TreeNode\* root) { 4. vector<vector<int>> res; 5. if (!root) 6. return res; 7. deque<TreeNode\*> q; 8. q.push\_back(root); 9. while(!q.empty()) { 10. int current\_level\_size = q.size(); 11. vector<int> v; 12. for (int i=0; i<current\_level\_size; ++i) { 13. auto t = q.front(); 14. v.push\_back(t->val); 15. q.pop\_front(); 16. if (t->left) 17. q.push\_back(t->left); 18. if (t->right) 19. q.push\_back(t->right); 20. } 21. res.push\_back(v); 22. } 23. return res; 24. } 25. }; |

## 0103 二叉树的锯齿形层序遍历

链接：https://leetcode-cn.com/problems/binary-tree-zigzag-level-order-traversal/

标签：树

### 精选题解

略。

### 关键思路

类似层序遍历，使用队列，记录每层中的节点数，只是需要多加一个need\_reverse变量，用于判断是否进行反转操作。

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。

### 代码

| **0103 二叉树的锯齿形层序遍历.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152992488/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> zigzagLevelOrder(TreeNode\* root) { 4. if (root==nullptr) 5. return {}; 6. vector<vector<int>> res; 7. queue<TreeNode\*> q; 8. q.push(root); 9. bool need\_reverse = false; 10. while (!q.empty()) { 11. int size = q.size(); 12. vector<int> v; 13. while (size--) { 14. TreeNode\* t = q.front(); 15. v.push\_back(t->val); 16. q.pop(); 17. if (t->left) 18. q.push(t->left); 19. if (t->right) 20. q.push(t->right); 21. } 22. if (need\_reverse) 23. reverse(v.begin(), v.end()); 24. res.push\_back(v); 25. need\_reverse = !need\_reverse; 26. } 27. return res; 28. } 29. }; |

## 0104 二叉树的最大深度

链接：https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/

标签：树，递归，深搜，广搜

### 精选题解

* 官方题解 - 二叉树的最大深度
  + https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/solution/er-cha-shu-de-zui-da-shen-du-by-leetcode-solution/

### 关键思路

递归和迭代。此题中的递归本质上是深搜，迭代本质上是广搜。

递归看代码即可，迭代使用队列，每遍历完一层就将深度加1，参考“0102 二叉树的层序遍历”。

### 复杂度

* 时间复杂度：均为O(n)。
* 空间复杂度：均为O(n)。

### 代码1：递归

| **0104 二叉树的最大深度 - 递归.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153051821/ |
| 1. class Solution { 2. public: 3. int maxDepth(TreeNode\* root) { 4. if (root == nullptr) 5. return 0; 6. return 1 + max(maxDepth(root->left),maxDepth(root->right)); 7. } 8. }; |

### 代码2：迭代

| **0104 二叉树的最大深度 - 迭代.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153058048/ |
| 1. class Solution { 2. public: 3. int maxDepth(TreeNode\* root) { 4. if (!root) return 0; 5. int res=0; 6. queue<TreeNode\*> q; 7. q.push(root); 8. while (!q.empty()) { 9. int size = q.size(); 10. while (size--) { 11. TreeNode \* t = q.front(); 12. q.pop(); 13. if (t->left) 14. q.push(t->left); 15. if (t->right) 16. q.push(t->right); 17. } 18. ++res; 19. } 20. return res; 21. } 22. }; |

## 0199 二叉树的右视图

链接：https://leetcode-cn.com/problems/binary-tree-right-side-view/

标签：树

### 精选题解

* https://leetcode-cn.com/problems/binary-tree-right-side-view/solution/er-cha-shu-de-you-shi-tu-by-leetcode-solution/361054

### 关键思路

层序遍历。将每层最右边的节点值加入数组res即可。

* 判断是否在最右边：size为0。

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。

### 代码

| **0199 二叉树的右视图.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153065505/ |
| 1. class Solution { 2. public: 3. vector<int> rightSideView(TreeNode\* root) { 4. if (!root) return {}; 5. vector<int> res; 6. queue<TreeNode \*> q; 7. q.push(root); 8. while (!q.empty()) { 9. int size = q.size(); 10. while (size--) { 11. TreeNode \* t = q.front(); 12. if (t->left) 13. q.push(t->left); 14. if (t->right) 15. q.push(t->right); 16. q.pop(); 17. if (size==0) 18. res.push\_back(t->val); 19. } 20. } 21. return res; 22. } 23. }; |

## 0113 路径总和II

链接：https://leetcode-cn.com/problems/path-sum-ii/

标签：树，深搜

### 精选题解

* 官方题解 - 路径总和 II
  + https://leetcode-cn.com/problems/path-sum-ii/solution/lu-jing-zong-he-ii-by-leetcode-solution/

### 关键思路

深搜，递归。看代码可能比文字描述更清晰一些。

* 若遍历到叶子节点就返回
* path：当前路径
* res：二维数组，所有满足条件的path的数组
* 剩下的就是回溯+深搜的经典思路：将本层root的值加入path中，并将sum减去该值
  + 若此时为叶子节点且sum恰好减为0，则将该path加入res中
  + 否则分别递归root的左右子节点，并将新的sum传入
  + 递归左右子节点后，要从本层返回更上层，所以需要从path中将该层加入的root的值弹出；由于sum并不是引用传递，因此回到上层后sum会自动复原。

注意dfs中不要自作聪明地加入“root的值大于sum时则立刻返回”的判断条件，因为树中节点有可能为负，不必多此一举。

### 复杂度

* 时间复杂度：O(n)。遍历所有节点。（LeetCode官方题解给出的分析有误，评论中也有人指出来了。）
* 空间复杂度：O(n)。path的空间；递归栈的空间。

### 代码

| **0113 路径总和 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153084969/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> res; 4. vector<int> path; 5. void dfs(TreeNode\*root, int sum) { 6. if (!root) return; 7. path.push\_back(root->val); 8. sum -= root->val; 9. if (sum==0 && root->left==nullptr && root->right==nullptr) { 10. res.push\_back(path); 11. } 12. dfs(root->left, sum); 13. dfs(root->right, sum); 14. path.pop\_back(); 15. } 16. vector<vector<int>> pathSum(TreeNode\* root, int targetSum) { 17. dfs(root, targetSum); 18. return res; 19. } 20. }; |

## 0124 二叉树中的最大路径和

链接：https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/

标签：树，递归，深搜

### 精选题解

* 官方题解 - 二叉树中的最大路径和
  + <https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/solution/er-cha-shu-zhong-de-zui-da-lu-jing-he-by-leetcode-/>
* 「手画图解」别纠结递归的细节 | 124.二叉树中的最大路径和
  + https://leetcode-cn.com/problems/binary-tree-maximum-path-sum/solution/shou-hui-tu-jie-hen-you-ya-de-yi-dao-dfsti-by-hyj8/

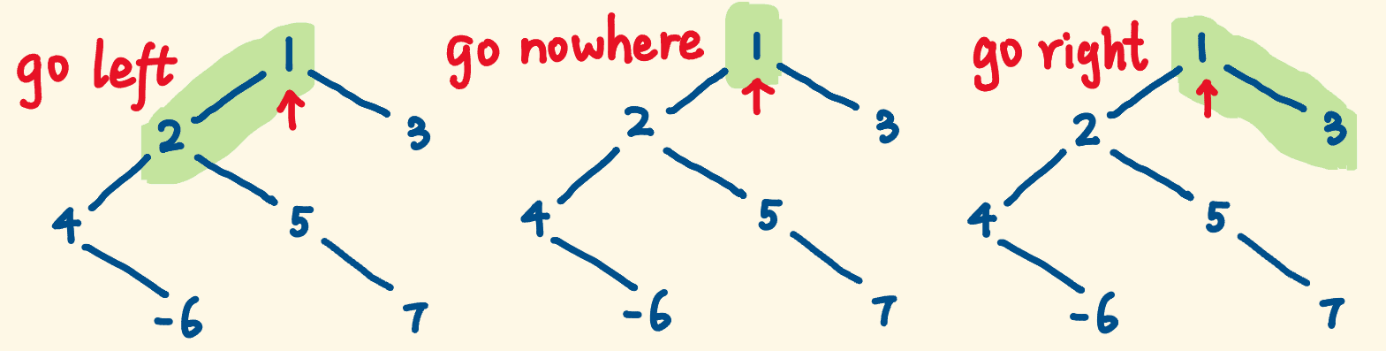
### 关键思路

递归。这题重在理解题意。（建议多练两遍）

路径：一条从树中任意节点出发，**沿父节点-子节点连接，**达到任意节点的序列。同一个节点在一条路径序列中至多出现一次 。该路径至少包含一个节点，且不一定经过根节点。

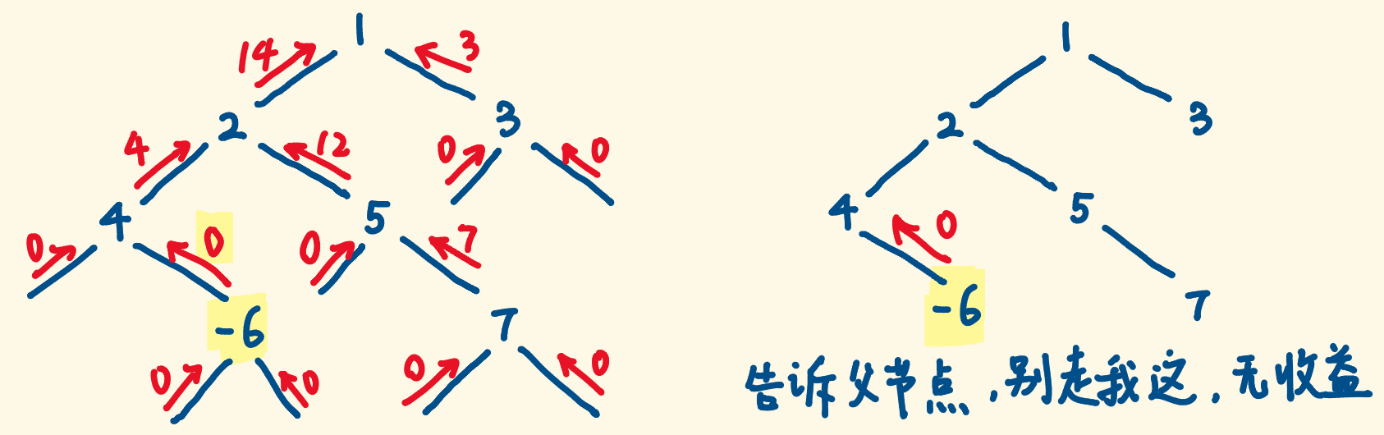
路径每到一个节点，有 3 种选择：

* ① 停在当前节点；② 走到左子节点；③ 走到右子节点。
* 走到子节点，又面临这 3 种选择，递归就是用来处理这种规模不一样的相同问题。
* 注意，不能走进一个分支又掉头回来走另一个分支，路径会重叠，不符合定义。



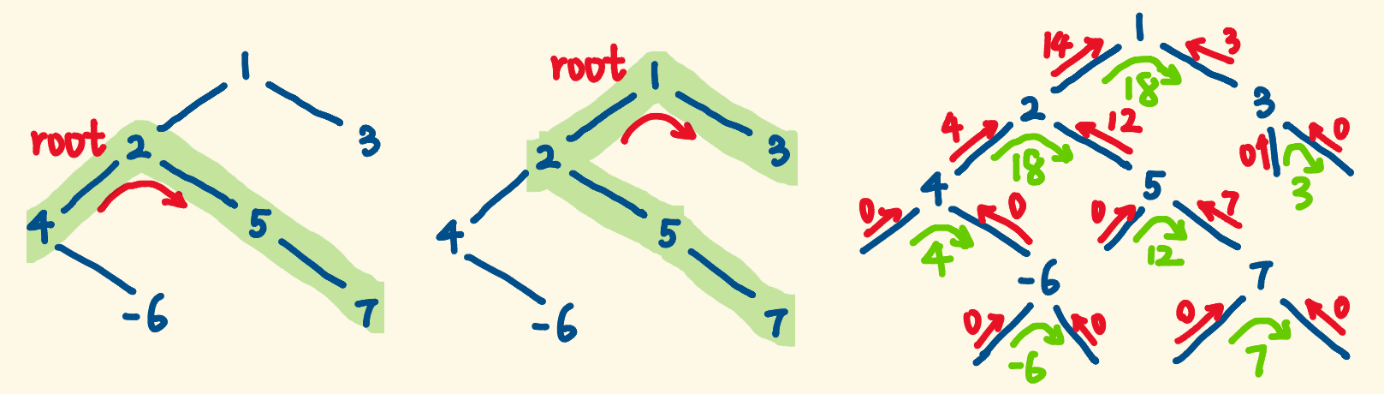
**递归：**

* 对于一个节点而言，它关心自己走入一个子树，能从中捞取的最大收益，不用管具体怎么走。
* 定义dfs函数：返回当前子树能向父节点“提供”的最大路径和。即一条从父节点延伸下来的路径，能在当前子树中获得的最大收益。分为三种情况：
  + 路径停在当前子树的根节点，在这个子树中收益：root.val
  + 走入左子树，在这个子树中的最大收益：root.val + dfs(root.left)
  + 走入右子树，在这个子树中的最大收益：root.val + dfs(root.right)
* 对应了前面所讲，对父节点而言的三种选择，最大收益取最大值：root.val + max(dfs(root.left), dfs(root.right))
* **再次提醒: 一条从父节点延伸下来的路径，不能走入左子树又掉头走右子树，不能两头收益，路径会重叠。**
* 当遍历到null节点时，null 子树提供不了收益，返回 0。
* 如果某个子树 dfs 结果为负，走入它，收益不增反减，该子树应被忽略，杜绝选择走入它的可能，让它返回 0，像null一样如同砍掉。



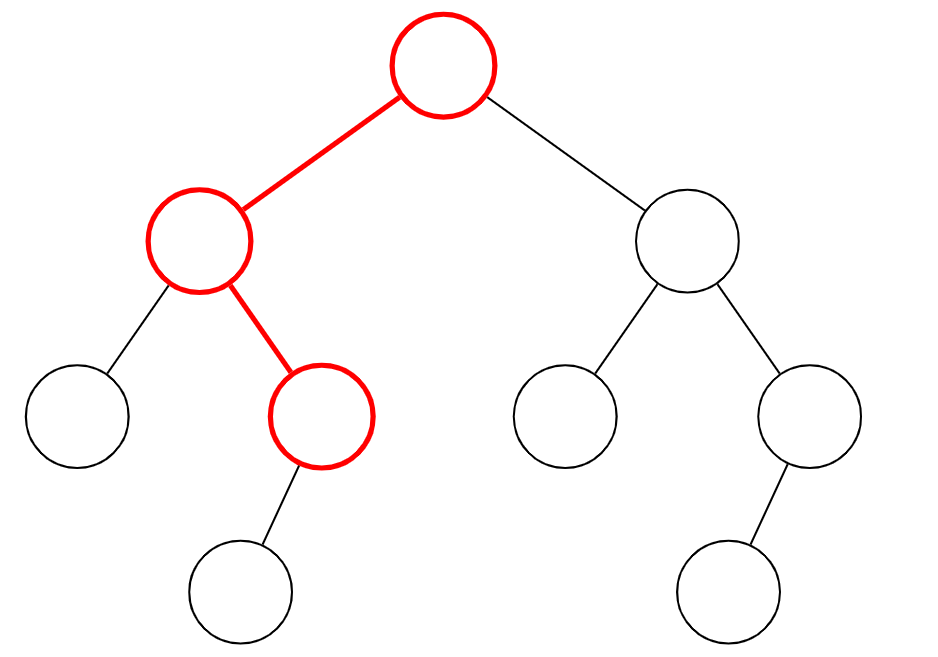
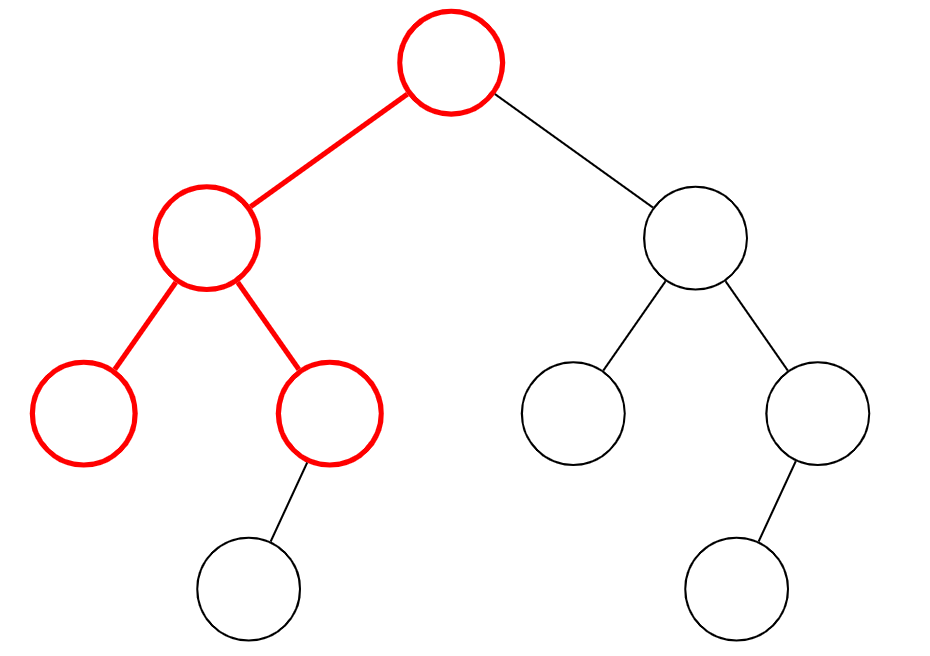
**子树中的内部路径要包含根节点**

* 由题意可知，最大路径和可能产生于局部子树中，如下图1。所以每递归一个子树，都求一下当前子树内部的最大路径和，见下图3，从中比较出最大的。
* **注意: 一个子树内部的路径，要包含当前子树的根节点。**如果不包含，那还算什么属于当前子树的路径，那就是当前子树的子树的内部路径了。
* 所以，一个子树内部的最大路径和 = 左子树提供的最大路径和 + 根节点值 + 右子树提供的最大路径和。即 dfs(root.left) + **root.val** + dfs(root.right)

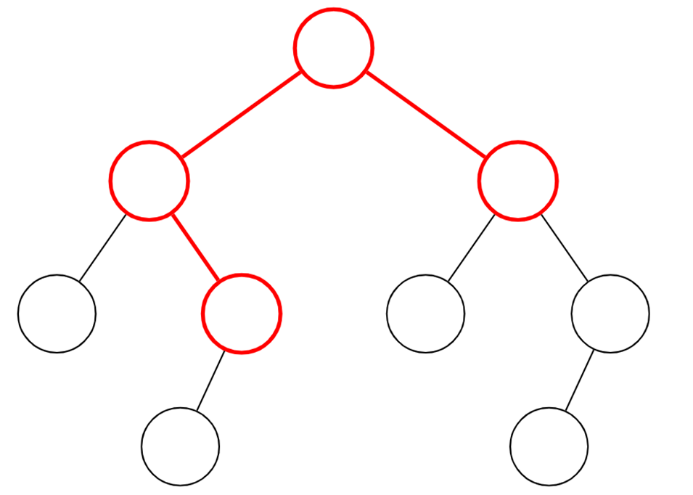


两个递归变量：

* maxGain(TreeNode\*)：每个节点的“最大收益”，也即以该节点为起点的路径中，路径和最大的值。必须包含当前节点的值。
  + 以下图为例，左边的情况不会出现，也即在计算maxGain时，路径中不会同时出现两个子节点；只会出现右边的情况，路径中每层只有一个节点。



* res：最终要返回的最大路径和，在每次递归中，都计算以当前节点为中心的路径和，该路径由左子节点的最大收益路径、当前节点、右子节点的最大收益路径组成，若比原有的大则更新
  + 注意，和计算maxGain不同，计算res的时候，是需要包含左右子节点（及其路径）的，也即下图中的情况



### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。

### 代码

| **0124 二叉树中的最大路径和.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153240509/ |
| 1. class Solution { 2. public: 3. int res=INT\_MIN; 4. int maxGain(TreeNode\* root) { 5. if (!root) return 0; 6. int L\_gain = max(maxGain(root->left),0); 7. int R\_gain = max(maxGain(root->right),0); 8. res = max(res, root->val + L\_gain + R\_gain); 9. return root->val + max(L\_gain, R\_gain); 10. } 11. int maxPathSum(TreeNode\* root) { 12. maxGain(root); 13. return res; 14. } 15. }; |

## 0105 从前序与中序遍历序列构造二叉树

链接：https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/

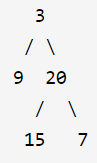
标签：树

### 精选题解

* https://leetcode-cn.com/submissions/detail/79890132/

### 关键思路

核心结构，递归正基于此：前序：[**根**,左,右]；中序：[左,**根**,右]

例如，，前序为 [**3**,**9**,**20**,15,7]，中序为 [**9**,**3**,15,**20**,7]

其中，节点3为根节点，之后在中序遍历中找到3的位置。在中序遍历中，左边都是节点3的左子树，右边都是3的右子树，**重点是找到左右子树的根结点在前序遍历中的索引**，从而将这两个子树区分开来，代入到下一层递归中。

由于中序遍历中3的左边只有1个节点9，因此在前序遍历中只需在节点3之后跳过这1个节点，即为3的右子树的根节点，也即20。

下面来计算左右子树的根节点索引递推公式，其中**pb**是指在**前序遍历**中的索引，ib和ie是指当前子树节点在中序遍历中索引的左右边界

* L\_pb = pb+1：
  + 左子树的根节点前序索引，易知是紧邻原根节点右边一个节点的索引，比如3的左子树的根节点9，其前序索引即为3的索引+1
* L\_ib = ib：
  + 左子树的中序左边界，由于在中序遍历中，左子树节点都在当前根节点的左边，所以其左边界和当前根节点相同
* R\_ie = ie：
  + 右子树的中序右边界，和上一行类似，中序遍历中，右子树节点都在当前根节点的右边，所以其右边界和当前根节点相同
* L\_ie = map[t->val]-1：
  + 左子树的中序右边界，也就是原根节点的左边一个
* R\_ib = map[t->val]+1：
  + 右子树的中序左边界，也就是原根结点的右边一个
* R\_pb = pb+num(L) = pb + (L\_ie - L\_ib + 1) +1= pb +( (map[t-val]-1)-ib+1)+1 = pb + map[t->val] – ib + 1
  + 右子树的根结点前序索引，为原根节点加上左子结点的个数再加1，左子结点的个数用上面得到的L\_ie-L\_ib+1即可求得。

剩下来就是分别对左右子树递归：

t->left = recur(L\_pb, L\_ib, L\_ie);

t->right = recur(R\_pb, R\_ib, R\_ie);

递归的初始条件是：（当然也是最后返回的）

return recur(0, 0, inorder.size()-1)

更新上述边界必须满足如下两个要求，否则直接返回nullptr：

* pb < pre.size()，也即pb的索引不越界
* ib <= ie，也即中序的左右边界中至少有一个节点，注意不要忘了等号。

if (pb<pre.size() && ib<=ie) {

}

return nullptr;

### 代码

| **0105 从前序与中序遍历序列构造二叉树.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152332676/ |
| 1. class Solution { 2. public: 3. unordered\_map<int, int> map; 4. vector<int> pre; 5. TreeNode\* recur(int pb, int ib, int ie) { 6. if (pb<pre.size() && ib<=ie) { 7. auto t = new TreeNode(pre[pb]); 8. int L\_pb = pb+1; 9. int L\_ib = ib; 10. int R\_ie = ie; 11. int L\_ie = map[t->val]-1; 12. int R\_ib = map[t->val]+1; 13. int R\_pb = pb + (L\_ie - L\_ib + 1) + 1; 15. t->left = recur(L\_pb, L\_ib, L\_ie); 16. t->right = recur(R\_pb, R\_ib, R\_ie); 17. return t; 18. } 19. return nullptr; 20. } 21. TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) { 22. for (int i=0; i<inorder.size(); ++i) 23. map[inorder[i]] = i; 24. pre = preorder; 25. return recur(0, 0, inorder.size()-1); 26. } 27. }; |

## 0236 二叉树的最近公共祖先

链接：https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/

标签：树

### 精选题解

* 【236. 二叉树的最近公共祖先】简洁代码 - 二叉树的最近公共祖先
  + 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/
* 236. 二叉树的最近公共祖先（后序遍历 DFS ，清晰图解） - 二叉树的最近公共祖先
  + https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/236-er-cha-shu-de-zui-jin-gong-gong-zu-xian-hou-xu/
* 官方题解 - 二叉树的最近公共祖先
  + https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/er-cha-shu-de-zui-jin-gong-gong-zu-xian-by-leetc-2/

### 关键思路

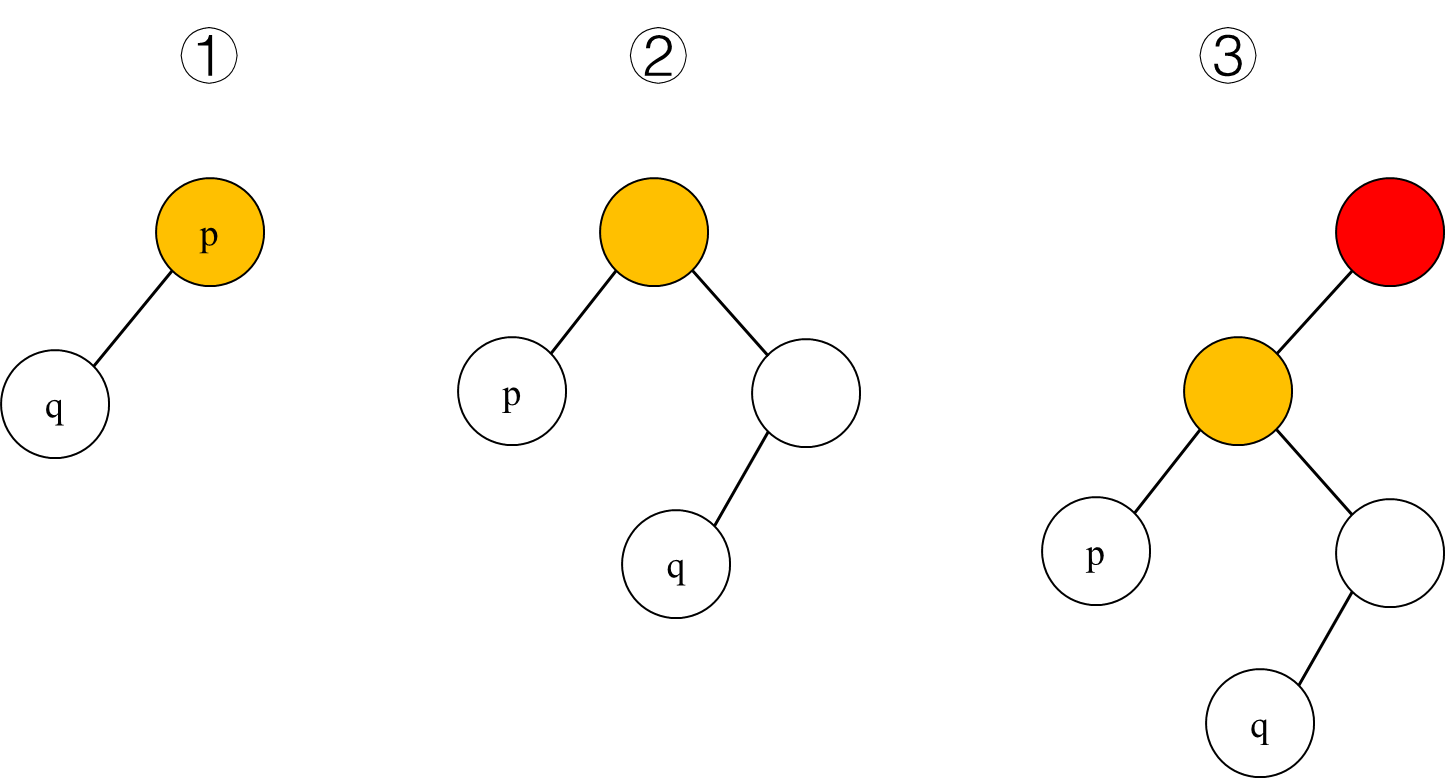
核心思路：递归。

对于每一层子递归，返回节点只有三种情况：空节点，p，q。

令t为当前递归层中的根节点。

通过递归，我们自顶向下搜索，遇到如下情况直接返回：

* t为空（叶节点）：返回空节点
* t为p或q：返回当前节点（代码第4 ~ 5行）
  + 因为没有必要再往下搜索了，设当前节点为p，则无非两种情况
* 若另一个节点（图1中的q）在t的子树中，那么t的父节点的另一侧子节点的递归应当返回空节点，则最终返回到root时，该侧非空，必为该节点t
* 若另一个节点（图2中的q）在另一侧，则更上层的递归一定遇到左右节点均非空的情况，也就跳到了下面一种情况
* 左右节点均非空：这意味着p和q分列其左右子树中（图2），则当前结点必为最近公共祖先（代码第8 ~ 9行）
  + 在这种情况中，我们需要知道左右节点的情况，因此要进入子递归得到左右子节点返回（代码第6~7行）
  + 尽管递归是自顶向下，但是此时已经递归了左右子节点，这意味着该节点一定是最靠近p和q的公共祖先，不会出现图3红色节点的情况（事实上，由于我们只有在遇到空、p和q这三种情况下才会直接返回，因此红色节点所在层递归的返回，一定晚于黄色节点所在层递归的返回）
  + 若左右节点至少有一个为空，则返回另一个（二者都为空则也为空）（代码第10行）



### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。递归栈的深度。最糟情况是退化成链表，则为O(n)。

### 代码

| **0236 二叉树的最近公共祖先.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/150991798/ |
| 1. class Solution { 2. public: 3. TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) { 4. if (!root || root==p || root==q) 5. return root; 6. TreeNode \* left = lowestCommonAncestor(root->left, p, q); 7. TreeNode \* right = lowestCommonAncestor(root->right, p, q); 8. if (left && right) 9. return root; 10. return left ? left : right; 11. } 12. }; |

## 1008 前序遍历构造二叉搜索树

链接：<https://leetcode-cn.com/problems/construct-binary-search-tree-from-preorder-traversal/>

标签：树

### 精选题解

* 官方题解 - 前序遍历构造二叉搜索树
  1. <https://leetcode-cn.com/problems/construct-binary-search-tree-from-preorder-traversal/solution/jian-kong-er-cha-shu-by-leetcode/>
* C++ 中规中矩的4ms解法（dfs 时间O(n)） - 前序遍历构造二叉搜索树
  1. https://leetcode-cn.com/problems/construct-binary-search-tree-from-preorder-traversal/solution/c-zhong-gui-zhong-ju-de-4msjie-fa-dfs-by-gary\_co-5/

### 关键思路

有两种思路，一种是递归，另一种是迭代。**该题理解起来并不简单，建议两种思路都拿纸笔画一画。**

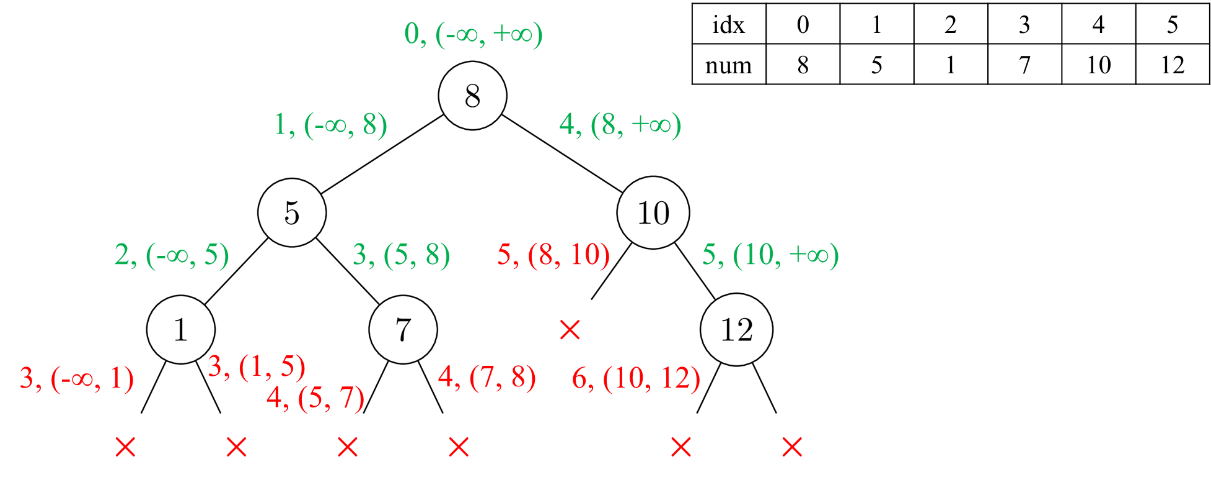
1、递归

由于题中所给是二叉**搜索树**，因此仅给出前序遍历数组即可唯一确定该树结构。

该思路实际上是分治思想。使用一个指针idx顺序指向前序遍历数组，根据当前打算填入的位置的结点值上下界 (left, right)，判断是否进行插入。

以下图为例：

1. 初始时idx=0，(left, right)=(INT\_MIN, INT\_MAX)
2. 终止条件：idx达到数组长度，或指向的值不在将要填入位置的允许范围内，返回nullptr
3. 以idx当前指向的数组元素生成结点root，并将idx加1（代码1第7行）
   1. 为了写法简便，将idx声明为public变量
   2. 注意idx++ 的简洁写法
4. 接着调用递归，生成root的左右子结点（代码1第8 ~ 9行）
   1. 左右子节点和当前结点的不同仅在于：允许的结点值上下界不同
   2. 先调用左子结点的递归，再调用右子结点的递归，也即遵循“根→左→右”的顺序



**2、迭代**

该思路和“剑指 33 二叉搜索树的后序遍历序列”类似，只不过单调栈从递增变成了递减。

1. 上题中后序遍历反转是 “根→右→左”
2. 本题中前序遍历是“根→左→右”

该思路如下：

1. 初始化：以数组第一个元素生成根结点root，并入栈
2. 顺序遍历数组的每个元素
   1. 以当前元素生成结点cur，将栈顶结点赋给top
   2. 如果cur的值小于top的值，表明cur在top的左边，二者关系只能为“根→左”，并且由于前序遍历的特性，cur必为top紧邻的左子子结点，故直接将cur插入为top的左子结点
   3. 如果cur的值大于top的值，表明cur在top的右边，二者关系可能为“根→右”或“左→右”，此时我们需要找到cur对应的根结点（以将cur插入该根中）

* 该根结点是“比cur小的结点”中最大的那个节点，由于stk从底向上是单调递减的，因此不断将栈顶元素赋给top并出栈，直到栈顶的值比cur大
* 此时满足：栈顶元素 > cur > top，故top即为cur的根结点，将cur插入为top的右子节点
  1. 将cur入栈

注意本题中代码2中第9行，与“剑指 33 二叉搜索树的后序遍历序列”代码2中的第5行，二者位置有所不同

1. 本题中需要在每个for循环开始都将栈顶元素赋值给top，这是为了保证能够正常处理“cur为top的左子结点”的情况
2. 而“剑指33”题中并不需要构造二叉搜索树，所以只需记录当前cur和root的大小关系是否合法，不必每次循环都给root赋值（以将cur插入为其子节点）

### 复杂度

### 代码1 – 递归

| **1008 前序遍历构造二叉搜索树 - recursion.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142581774/ |
| 1. class Solution { 2. public: 3. int idx = 0; 4. TreeNode\* helper(vector<int>& preorder, int left, int right) { 5. if (idx == preorder.size() || preorder[idx] < left || preorder[idx] > right) 6. return nullptr; 7. TreeNode\* root = new TreeNode(preorder[idx++]); 8. root->left = helper(preorder, left, root->val); 9. root->right = helper(preorder, root->val, right); 10. return root; 11. } 12. TreeNode\* bstFromPreorder(vector<int>& preorder) { 13. return helper(preorder, INT\_MIN, INT\_MAX); 14. } 15. }; |

### 代码2 – 迭代

| **1008 前序遍历构造二叉搜索树 – iterative.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142618703/ |
| 1. **class** **Solution** { 2. **public**: 3. TreeNode\* bstFromPreorder(vector<int>& preorder) { 4. stack<TreeNode\*> stk; 5. TreeNode\* root = **new** TreeNode(preorder[0]); 6. stk.emplace(root); 7. **for** (int i=1; i<preorder.size(); ++i) { 8. TreeNode\* cur = **new** TreeNode(preorder[i]); 9. TreeNode\* top = stk.top(); 10. **while** (!stk.empty() && cur->val > stk.top()->val) { 11. top = stk.top(); 12. stk.pop(); 13. } 14. **if** (cur->val > top->val) 15. top->right = cur; 16. **else** 17. top->left = cur; 18. stk.emplace(cur); 19. } 20. **return** root; 21. } 22. }; |

## 剑指 33 二叉搜索树的后序遍历序列

链接：<https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-hou-xu-bian-li-xu-lie-lcof/>

标签：树，递归，栈

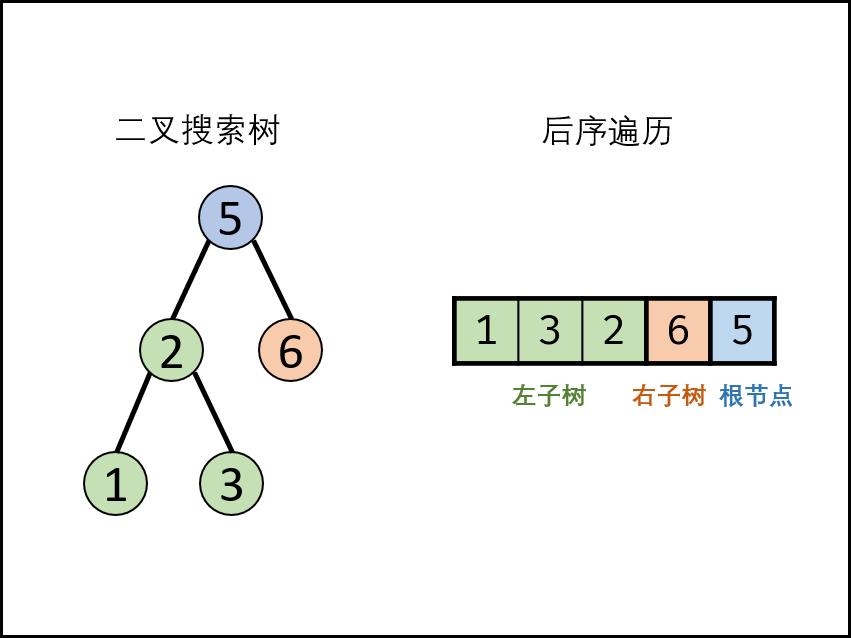
### 精选题解

1. 面试题33. 二叉搜索树的后序遍历序列（递归分治 / 单调栈，清晰图解）
   1. <https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-hou-xu-bian-li-xu-lie-lcof/solution/mian-shi-ti-33-er-cha-sou-suo-shu-de-hou-xu-bian-6/>
2. 递归和栈两种方式解决，最好的击败了100%的用户 - 二叉搜索树的后序遍历序列
   1. <https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-hou-xu-bian-li-xu-lie-lcof/solution/di-gui-he-zhan-liang-chong-fang-shi-jie-jue-zui-ha/>

### 关键思路

有几个重要的基本事实，对于一个二叉搜索树以及对应的正确的后序遍历数组来说：

1. 树的一个子树对应着数组中一段连续的子数组
2. 二叉搜索树左子树的结点值都比根结点小，右子树的结点值都比根结点大
3. 后序遍历汇总，子数组最后一个元素对应子树的根结点
4. 从子数组左边界开始，第一个比根结点大的数，其左侧的数都在左子树，右侧的数（包含自身）都在右子树



**1、递归**

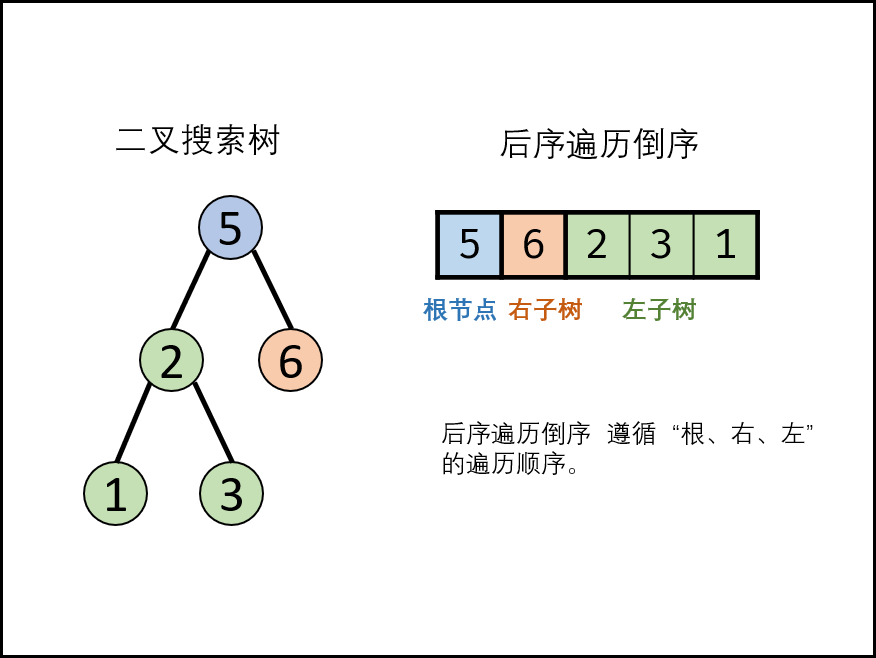
1. 设数组索引的左右边界为left和right，其中postorder[right] 对应根结点的值。
   1. 如果left>=right，表明只有一个或是没有子结点，必为正确的后序遍历，返回true
2. 从左到右找到第一个比根结点大的数，也即左右子树的分界点
   1. 左边的数肯定都是小于根结点的值的，因此只需判断右边的数
   2. 如果右边的数存在比根结点小的，说明不在右子树上却被划为右子树的结点，这表明不是正确的后序遍历，返回false
3. 递归判断上面分割得到的左右子树各自是否对应正确的后序遍历数组
4. 注意代码1第13行在循环中使用tmp++的简洁写法

**2、单调栈**

**这个方法理解起来有点困难，建议拿纸笔画一画。**

为什么想到要用单调栈呢？（参考评论区“失火的夏天”老哥）

1. 通过前序遍历数组构造二叉搜索树时（参考“1008 前序遍历构造二叉搜索树”），很容易想到单调栈，这里就想到后序遍历是否也能用
   1. 前序遍历数组：左→右→根
   2. 后序遍历数组：根→左→右
2. 所以为了能用到前序构造的方法，将后序遍历数组反转
   1. 后序遍历数组反转：根→右←左



我们的终止条件（反例）是：

* 如果**某个左子树上的结点比其根结点大**，那么就违反了二叉搜索树的特性，返回false。

对于反转的后序遍历数组，有如下两种情况：

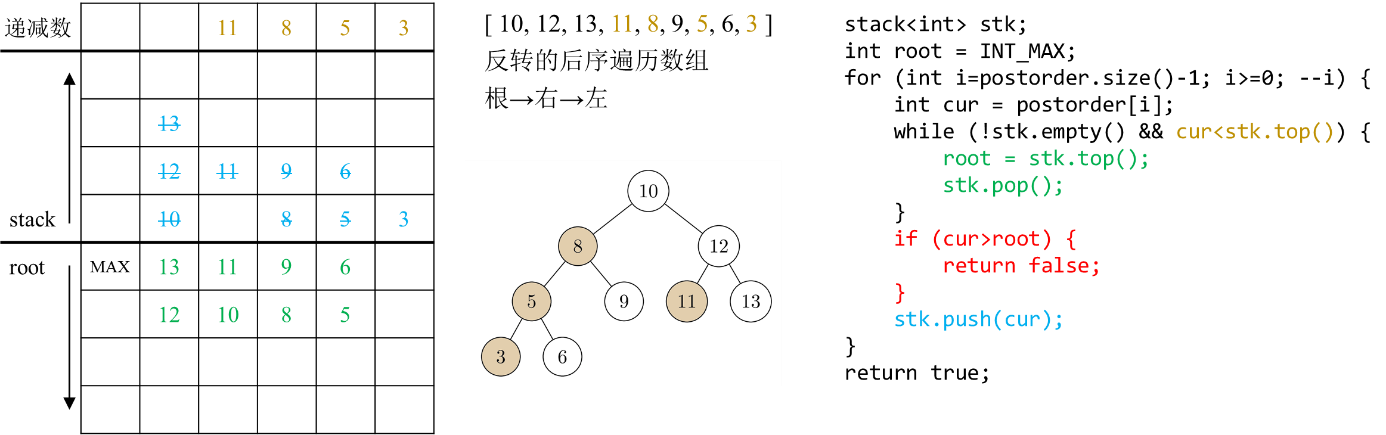
1、如果递增，也即 ，比如上图中 ⑤ < ⑥，则 ⑥ 必为 ⑤ 的右子结点。理由：

* 1. 比当前结点大的数必然在其右侧（指的是二叉搜索树上的右，不是数组中的右）
  2. 而反转后的遍历数组顺序为“根→右→左”，因此只有“根→右”这一种可能
  3. 也即 必为 的右子结点，且由于最先遍历根结点，因此 还必定是 紧邻的右子结点

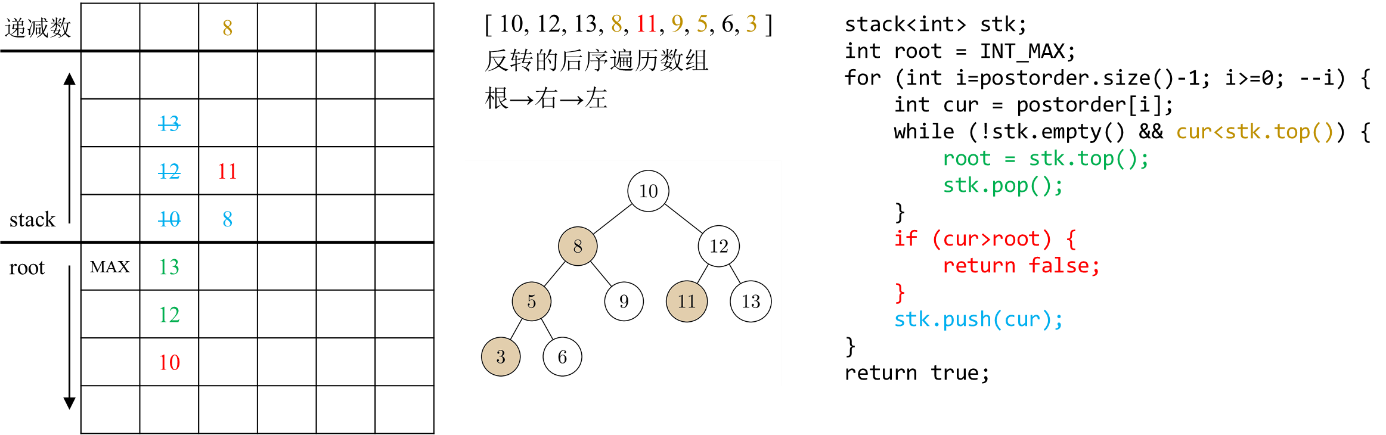
2、如果递减，也即 ，比如上图中 ③ > ①，则 ① 必为其左边某个节点root的左子结点，且root的值是 ① 左边结点（⑤⑥**②**③）中最小的，也即 ②。理由：

* 1. 比当前结点小的数必然在其左侧（指的是二叉搜索树上的左，不是数组中的左）
  2. 而反转后的遍历数组顺序为“根→右→左”，因此有“根→左”和“右→左”这两种可能
  3. 所以和上面一种情况有微妙差别，出现这种情况时，左侧所有结点都必然比 大，因此上面说的“左侧”，其实也可以指数组中的左侧
  4. 这里的“右”肯定是大于“根”的，所以为了找到 的根结点（以进行终止条件的判断），我们就需要找到左边结点中数值最小的那个结点

那么如何利用上面的基本事实，结合单调栈来使用呢？



正确的后序遍历数组



错误的后序遍历数组

1. 首先顾名思义，单调栈肯定是单调的
2. 既然我们选择将“某个左子树上的结点比其根结点大”作为终止条件，那么就不用管情况1，因而遇到递增的数就不断入栈
3. 初始的root值设为INT\_MAX，也即原来的整棵树视为该无限大值结点的左子树，这样就可以复用题中的终止条件，不必做额外的复杂的边界条件判断
4. 当遇到情况2时，新的结点肯定是栈中某个结点的左子结点，不断更新root值为栈顶结点值（也即栈中最大值）并出栈，重复该过程直到当前结点值大于等于栈顶值（也即变成情况1）
   1. 同时一个隐含的变化是，这里出栈的都是最终找到的根结点的右子树结点
   2. 完成while循环中的出栈操作之后，该根结点右边的子结点都已经被剔除掉了
5. 由上一点可知，当前根结点右边的子结点都已经被剔除，如果新的结点大于当前根结点值，那么必然是不合法的，因此返回false

### 复杂度

**递归：**

1. 时间复杂度：O()/O(nlogn)。最坏情况下退化成链表，遍历n个节点，每个节点递归O(n)次。平均情况下为O(nlogn)，该复杂度计算相对复杂，可参考算法导论相应章节。
2. 空间复杂度：O(n)/O(logn)。最坏情况下递归深度为n，递归栈大小为O(n)；平均情况下为O(logn)。

**单调栈：**

1. 时间复杂度：O(n)。遍历树上全部结点，每个结点最多入栈和出栈一次。
2. 空间复杂度：O(n)。栈空间。

### 代码1 - 递归

| **JZ-33 二叉搜索树的后序遍历序列 - recursion.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142222224/ |
| 1. class Solution { 2. public: 3. bool recur(vector<int>& postorder, int left, int right) { 4. if (left >= right) 5. return true; 6. int mid = left; 7. int root = postorder[right]; 8. while (postorder[mid] < root) { 9. ++mid; 10. } 11. int tmp = mid; 12. while (tmp < right) { 13. if (postorder[**tmp++**] < root) { 14. return false; 15. } 16. } 17. return recur(postorder, left, mid-1) && recur(postorder, mid, right-1); 18. } 19. bool verifyPostorder(vector<int>& postorder) { 20. return recur(postorder, 0, postorder.size()-1); 21. } 22. }; |

### 代码2 – 单调栈

| **JZ-33 二叉搜索树的后序遍历序列 - stack.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142355016/ |
| 1. class Solution { 2. public: 3. bool verifyPostorder(vector<int>& postorder) { 4. stack<int> stk; 5. int root = INT\_MAX; 6. for (int i=postorder.size()-1; i>=0; --i) { 7. int cur = postorder[i]; 8. while (!stk.empty() && cur<stk.top()) { 9. root = stk.top(); 10. stk.pop(); 11. } 12. if (cur>root) { 13. return false; 14. } 15. stk.push(cur); 16. } 17. return true; 18. } 19. }; |

# 双指针

## 0003 无重复字符的最长子串

链接：https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/

标签：哈希表，双指针，滑动窗口

### 精选题解

* 官方题解 - 无重复字符的最长子串
  1. <https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/solution/wu-zhong-fu-zi-fu-de-zui-chang-zi-chuan-by-leetc-2/>
* 滑动窗口 - 无重复字符的最长子串
  1. https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/solution/hua-dong-chuang-kou-by-powcai/

### 关键思路

双指针。

核心：假如s[L,R] 是一个无重复字符子串，那么 s[L+1, R] 肯定也是一个无重复子串。

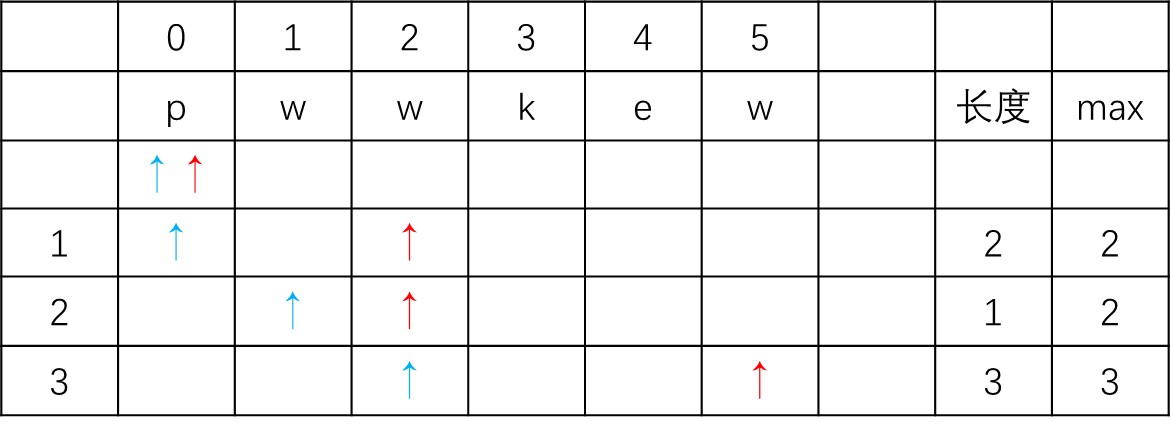
1. 右边界R只需要在前一次的位置继续向右增长就可以，不需要重新从L处开始遍历。

* + 每次都将新的s[R] 插入哈希表中。

2. 关于左指针，当每次得到以s[L] 开头的最长的无重复子串（也即s[R] 对应的字符已在哈希表中）后，就计算该子串长度，并将L加一，并从哈希表中删去s[L]，进入下一次扫描（也即重复步骤1）。

这里相比官方解答有两个小优化：

* 当我们的R到达字符串末尾时就停止遍历，也即左指针不需要遍历所有的字符。
* L和R的初值和边界条件更简单清晰，不容易出错。



### 复杂度

* 时间复杂度：O(n)。左右指针至多遍历全部字符。
* 空间复杂度：O(k)。哈希表中最多存储字符集中的全部字符，k为字符总数。

### 代码

| **0003 无重复字符的最长子串.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/149269570/ |
| 1. class Solution { 2. public: 3. int lengthOfLongestSubstring(string s) { 4. unordered\_set<char> us; 5. int n = s.size(); 6. if (n==0) return 0; 7. int res=0; 8. int L=0, R=0; 9. while (R<n) { 10. while (R<n && !us.count(s[R])) 11. us.insert(s[R++]); 12. res = max(res, R-L); 13. us.erase(s[L++]); 14. } 15. return res; 16. } 17. }; |

## 0015 三数之和

链接：https://leetcode-cn.com/problems/3sum/

标签：双指针

### 精选题解

* 三数之和（排序+双指针，易懂图解） - 三数之和
  + https://leetcode-cn.com/problems/3sum/solution/3sumpai-xu-shuang-zhi-zhen-yi-dong-by-jyd/

### 关键思路

先排序，再用双指针L和R从两边向中间遍历，类似“剑指57-1 和为s的两个数字”，需要证明能够覆盖全部情况。

注意跳过相同数字以避免重复相加。i、L和R的范围已经决定了会将 [1,1,1] （暂时忽略和为0的要求）这种情况包含在内：i不超过nums.size()-3；L从i+1开始；R > L。

### 代码

| **0015 三数之和.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152295583/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> threeSum(vector<int>& nums) { 4. int n = nums.size(); 5. sort(nums.begin(), nums.end()); 6. if (n < 3) 7. return {}; 8. vector<vector<int>> res; 9. int L, R; 10. int sum; 11. for (int i=0; i<n-2; ++i) { 12. if (nums[i] > 0) 13. break; 14. if (i>0 && nums[i]==nums[i-1]) 15. continue; 16. L = i+1; 17. R = n-1; 18. while (L<R) { 19. sum = nums[i] + nums[L] + nums[R]; 20. if (sum<0) { 21. ++L; 22. while (L<R && nums[L]==nums[L-1]) 23. ++L; 24. } else if (sum>0) { 25. --R; 26. while (L<R && nums[R]==nums[R+1]) 27. --R; 28. } else { 29. res.push\_back({nums[i], nums[L], nums[R]}); 30. ++L; 31. --R; 32. while (L<R && nums[L]==nums[L-1]) 33. ++L; 34. while (L<R && nums[R]==nums[R+1]) 35. --R; 36. } 37. } 38. } 39. return res; 40. } 41. }; |

## 0151 翻转字符串里的单词 + 剑指 58-1 翻转单词顺序

链接：https://leetcode-cn.com/problems/reverse-words-in-a-string/

<https://leetcode-cn.com/problems/fan-zhuan-dan-ci-shun-xu-lcof/>

标签：双指针

### 精选题解

* 官方题解 - 翻转字符串里的单词
  1. <https://leetcode-cn.com/problems/reverse-words-in-a-string/solution/fan-zhuan-zi-fu-chuan-li-de-dan-ci-by-leetcode-sol/>

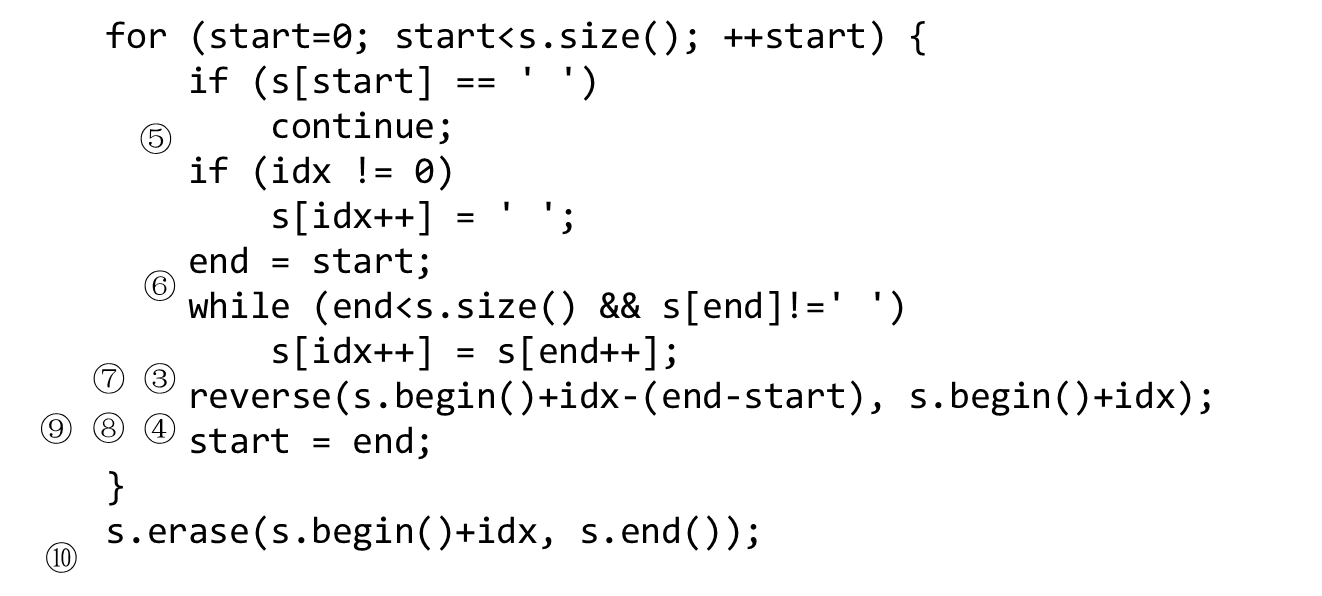
### 关键思路

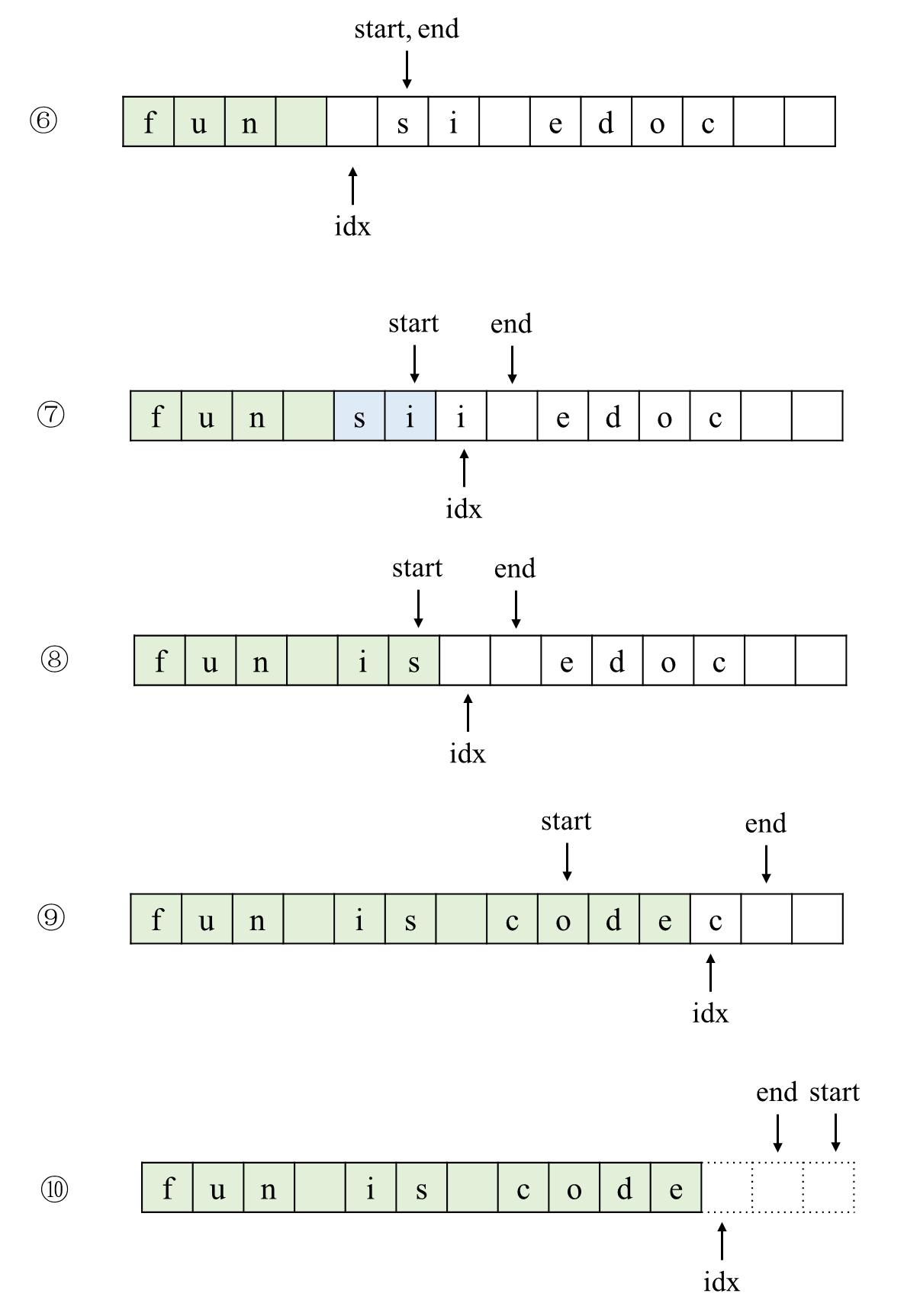
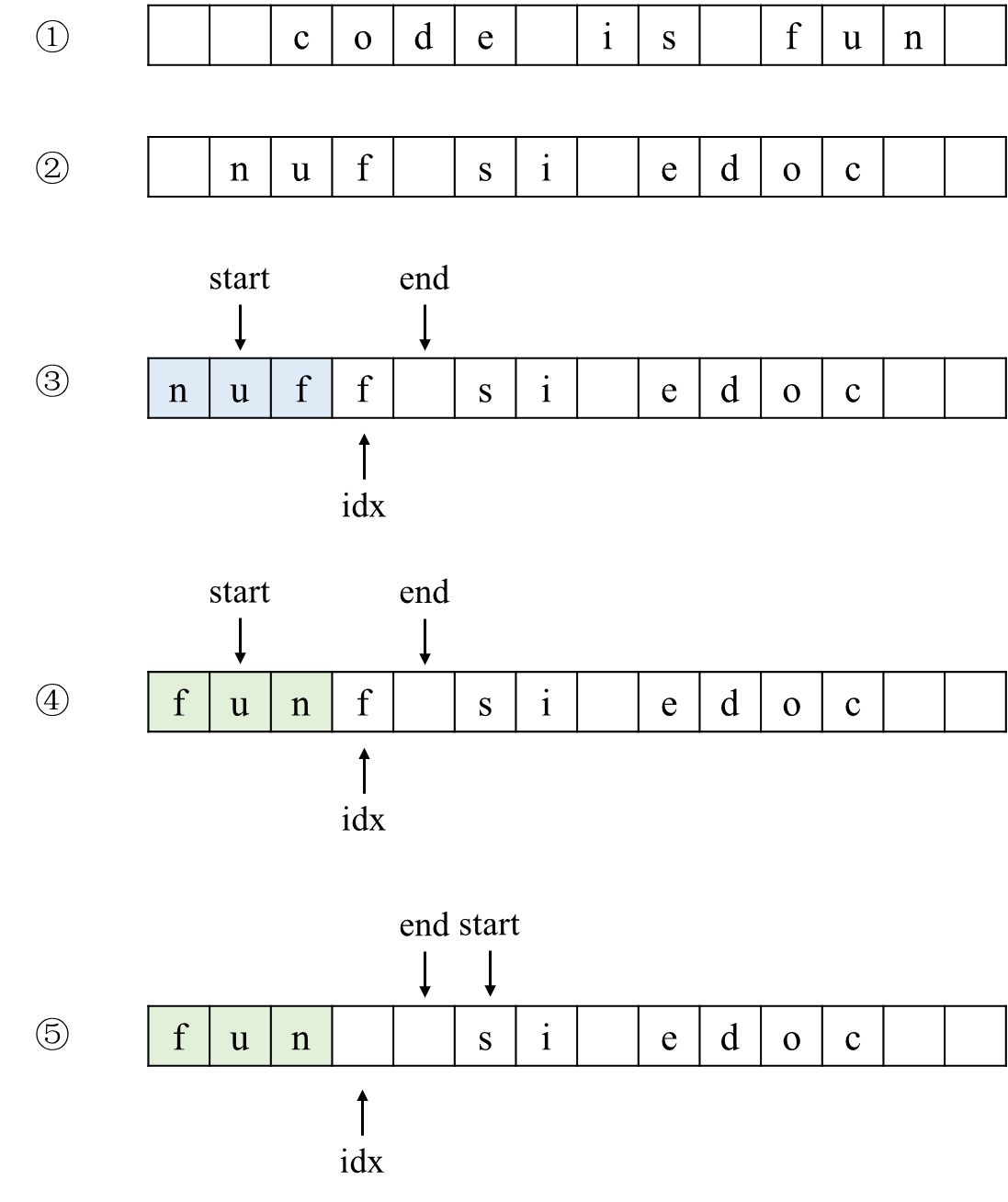
C++可以修改字符串，因此可以使用in-place的算法。

* 使用三个指针
  1. idx：指向string当前写入字符的位置
  2. start：当前扫描单词的开头
  3. end：当前扫描单词的最右字符

流程如下：

* 翻转整个字符串
* 扫描字符串，使start指向第一个非空字符，并将end从start开始
* 扫描后续字符，直到末尾或空格，同时将读到的字符写入索引idx处
* 将从idx向前长度end - start的子字符串翻转
* 将start指向end的位置
* 重复2~5，直到扫描到字符串末尾
* 将字符串中从idx开始到末尾的空间抹除





### 复杂度

* 时间复杂度：O(n)。遍历整个字符串。
* 空间复杂度：O(1)。

### 代码

| **0151 翻转字符串里的单词 + JZ-58-1 翻转单词顺序.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/147737931/ |
| 1. class Solution { 2. public: 3. string reverseWords(string s) { 4. reverse(s.begin(), s.end()); 5. int idx = 0; *// offset of pointer to string s* 6. int start = 0; *// start of current word* 7. int end = 0; *// end of current word* 8. for (start=0; start<s.size(); ++start) { 9. if (s[start] == ' ') 10. continue; 11. if (idx != 0) *// add space between two words* 12. s[idx++] = ' '; 13. end = start; 14. while (end<s.size() && s[end]!=' ') 15. s[idx++] = s[end++]; 16. reverse(s.begin()+idx-(end-start), s.begin()+idx); 17. start = end; 18. } 19. *// erase uesless space of string* 20. s.erase(s.begin()+idx, s.end()); 21. return s; 22. } 23. }; |

## 剑指 21 调整数组顺序使奇数位于偶数前面

链接：<https://leetcode-cn.com/problems/diao-zheng-shu-zu-shun-xu-shi-qi-shu-wei-yu-ou-shu-qian-mian-lcof/>

标签：双指针

### 精选题解

* 首尾双指针，快慢双指针 - 调整数组顺序使奇数位于偶数前面
  1. 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/

### 关键思路

双指针。

* 当left<right时，left从左往右，遇到偶数则停下，right从右往左，遇到奇数则停下
* 当left和right都停下时，交换二者指向的元素，并且left++和right--
  1. ++和--是因为交换过的两个数肯定是正确的位置，无须重复判断
* 重复上面的流程直到left>=right
* 判断奇数和偶数时，如果使用 (nums[left]&1)==1的写法，需要在&运算符的两侧加上括号，因为其运算优先级较低，不加括号会优先运算1==1。
  1. 所以通常还是建议使用nums[left]%2==1的写法，既直观也不容易出错
* 代码中有几个细节和技巧值得记住
  1. 第5~13行中while、if和continue配合使用的技巧，有效避免了更复杂的判断逻辑
  2. 第14行中left++和right--的简洁写法

### 复杂度

* 时间复杂度：O(n)。遍历整个数组。
* 空间复杂度：O(1)。两个整型“指针”。

### 代码

| **JZ-21 调整数组顺序使奇数位于偶数前面.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142187503/ |
| 1. class Solution { 2. public: 3. vector<int> exchange(vector<int>& nums) { 4. int left = 0, right = nums.size()-1; 5. while (left<right) { 6. if (nums[left] % 2 == 1) { 7. ++left; 8. continue; 9. } 10. if (nums[right] % 2 == 0) { 11. --right; 12. continue; 13. } 14. swap(nums[left++], nums[right--]); 15. } 16. return nums; 17. } 18. }; |

## 剑指 57-1 和为s的两个数字

链接：https://leetcode-cn.com/problems/he-wei-sde-liang-ge-shu-zi-lcof/

标签：双指针，哈希表

### 精选题解

* 面试题57. 和为 s 的两个数字（双指针 + 证明，清晰图解） - 和为s的两个数字
  1. https://leetcode-cn.com/problems/he-wei-sde-liang-ge-shu-zi-lcof/solution/mian-shi-ti-57-he-wei-s-de-liang-ge-shu-zi-shuang-/

### 关键思路

两个指针left和right指向两个数，从数组的两端开始，比target小则left加1，比target小则right减1。该题的关键是如何证明一定能覆盖正确的解？

证明思路：假设 (i,j) 是两个符合要求的左右指针对，则必从 (i-1, j) 或 (i, j+1) 而来，以此递推，必然能从 (0, nums.size()-1) 得来。

### 复杂度

* 时间复杂度：O(n)。最差情况下遍历整个数组。
* 空间复杂度：O(1)。

### 代码

| **JZ-57-1 和为s的两个数字.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143553837/ |
| 1. class Solution { 2. public: 3. vector<int> twoSum(vector<int>& nums, int target) { 4. int left = 0, right = nums.size()-1; 5. int sum = 0; 6. while (left < right) { 7. sum = nums[left] + nums[right]; 8. if (sum == target) 9. return vector<int>{nums[left], nums[right]}; 10. else if (sum < target) 11. ++left; 12. else 13. --right; 14. } 15. return {}; 16. } 17. }; |

## 剑指 57-2 和为s的连续正数序列

链接：https://leetcode-cn.com/problems/he-wei-sde-lian-xu-zheng-shu-xu-lie-lcof/

标签：双指针，滑动窗口

### 精选题解

* 什么是滑动窗口，以及如何用滑动窗口解这道题（C++/Java/Python） - 和为s的连续正数序列
  1. https://leetcode-cn.com/problems/he-wei-sde-lian-xu-zheng-shu-xu-lie-lcof/solution/shi-yao-shi-hua-dong-chuang-kou-yi-ji-ru-he-yong-h/

### 关键思路

本题实际上就是求数组 [1, 2, …, n] 中满足和为s的连续子数列。采用滑动窗口。

要解决两个问题：（1）递推；（2）覆盖全部解。

先解决递推方式。设左右指针分别为i和j，则

* 每次只++i或++j，即只将左右边界右移，这样可以保证O(n) 时间复杂度，若允许左移，则为回溯，时间复杂度太高。
* 若sum==targt，则将 [i, j] 加入结果数组，同时这表明以i开头的数组不再有满足条件的，故sum-=i，并++i
* 若sum<target，则++j，并sum+=j
  1. 注意这里要先给j加1，再将其加到sum上
* 若sum>target，表明以i开头的数组没有满足条件的，故sum-=i，并++i



### 复杂度

* 时间复杂度：O(target)。左指针的上界为target/2，故加起来最多遍历target个数。
* 空间复杂度：O(1)。

### 代码

| **JZ-57-2 和为s的连续正数序列.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143560485/ |
| 1. class Solution { 2. public: 3. vector<vector<int>> findContinuousSequence(int target) { 4. int i=1, j=1; 5. int sum=1; 6. vector<vector<int>> res; 7. while (i<=target/2) { 8. if (sum == target) { 9. vector<int> tmp; 10. for (int k=i; k<=j; ++k) { 11. tmp.push\_back(k); 12. } 13. res.push\_back(tmp); 14. sum -= i; 15. ++i; 16. } else if (sum<target) { 17. ++j; 18. sum += j; 19. } else if (sum>target) { 20. sum -= i; 21. ++i; 22. } 23. } 24. return res; 25. } 26. }; |

# 栈、堆、队列

## 0084 柱状图中最大的矩形 + 美团 04 直方图内最大矩形

链接：https://leetcode-cn.com/problems/largest-rectangle-in-histogram/

标签：双指针，单调栈，动态规划

### 精选题解

* 暴力解法、栈（单调栈、哨兵技巧） - 柱状图中最大的矩形
  + <https://leetcode-cn.com/problems/largest-rectangle-in-histogram/solution/bao-li-jie-fa-zhan-by-liweiwei1419/>
* 【柱状图中最大的矩形】单调栈入门，使用单调栈快速寻找边界 - 柱状图中最大的矩形
  + https://leetcode-cn.com/problems/largest-rectangle-in-histogram/solution/84-by-ikaruga/
* 官方题解 - 柱状图中最大的矩形
  + https://leetcode-cn.com/problems/largest-rectangle-in-histogram/solution/zhu-zhuang-tu-zhong-zui-da-de-ju-xing-by-leetcode-/

### 关键思路

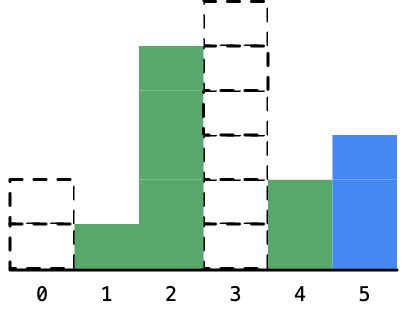
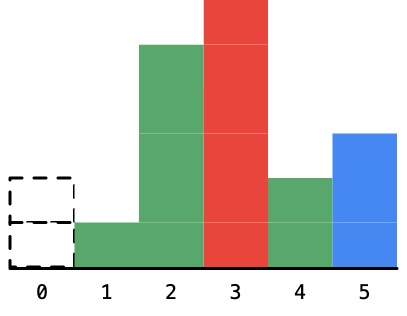
暴力解法：使用双指针，分别搜索每个柱形向左和向右第一个比它低的柱形的索引，然后计算两个指针最终的距离，即可得最大面积heights[i]\*(R-L-1)

* 会超时，尤其是对于 [1,1,1,1,…,1] 这种样例。

单调栈（拿纸笔画一画有助于理解）

尽管暴力解法会超时，但依旧给我们以启发：

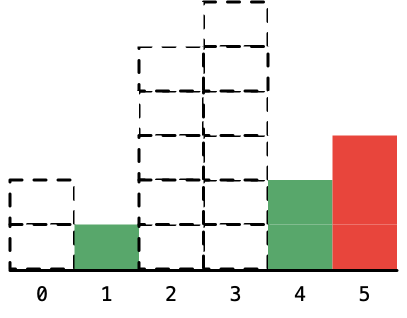
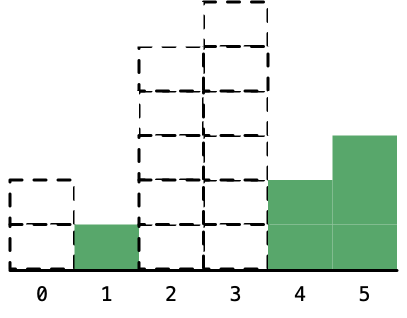
* **一个柱形的最大宽度由其分别向左和向右第一个比它低的柱形的索引确定**
  + **可以使用单调栈保证左边的柱形都比它严格低，那么只需要判定右边的即可**
  + **注意，栈中存放的是索引，而不是值**
* 对于每次出栈的元素，剩下的在栈顶的就是它左边第一个比它小的数，而右边第一个比它小的数则直接遍历得到（见代码更易理解）



遍历到4时，其高度比左边的3小，同时3左边的2也比3小，

从而可以确定3的最大宽度为1

一次遍历完成，之后还要将栈中元素全部出栈：



由于5的左边没有柱形，只需考虑到其左边的柱形的距离，也即到4的距离，故5的宽度为1

单调栈分为单调递增栈和单调递减栈：

1. 单调递增栈：栈内元素保持单调递增的栈

2. 单调递减栈：即栈内元素保持单调递减的栈

操作规则（下面都以单调递增栈为例）：

1. 如果新的元素比栈顶元素大，就入栈

2. 如果新的元素较小，那就一直把栈内元素弹出来，直到栈顶比新元素小

加入这样一个规则之后，会有什么效果

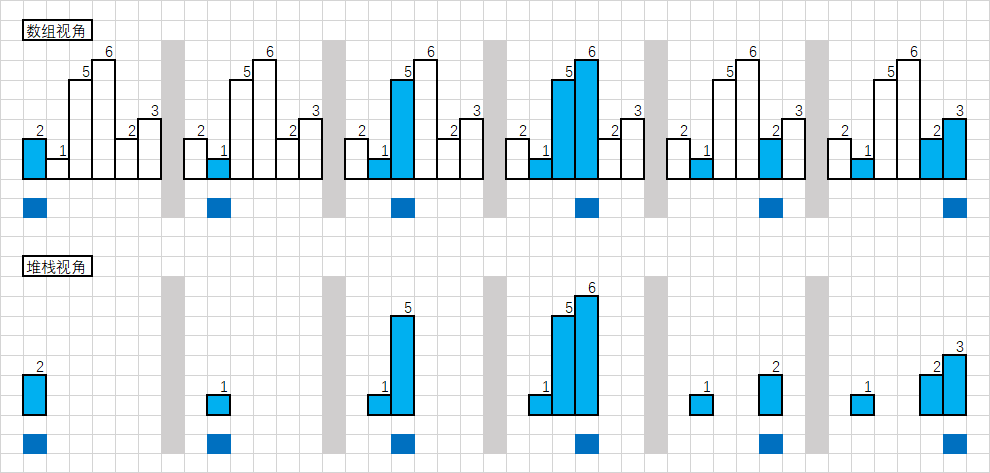
1. 栈内的元素是递增的

2. 当元素出栈时，说明这个新元素是出栈元素向后找第一个比其小的元素

* + 举个例子，如下图4，现在索引在6，栈里是1 5 6 。
  + 接下来新元素是2，那么6需要出栈。
  + 当6出栈时，右边2代表是6右边第一个比6小的元素。

3. 当元素出栈后，说明新栈顶元素是出栈元素向前找第一个比其小的元素

* + 当6出栈时，5成为新的栈顶，那么5就是6左边第一个比6小的元素。



一个完整的例子

通过在数组头部和尾部插入值为0的数，也即“哨兵”，可以减少边界条件的判断：

* 由于头部哨兵存在，栈始终非空
* 由于尾部哨兵存在，原来第一轮遍历后栈中剩余的数，也都能全部处理掉

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。

### 代码

| **MT-04 直方图内最大矩形 + 0084 柱状图中最大的矩形 - 单调栈.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/153742873/ |
| 1. class Solution { 2. public: 3. int largestRectangleArea(vector<int>& heights) { 4. if (heights.empty()) return 0; 5. heights.insert(heights.begin(), 0); 6. heights.push\_back(0); 7. int n = heights.size(); 8. int res = 0; 9. stack<int> s; 10. s.push(0); 11. for (int i=1; i<n; ++i) { 12. while (!s.empty() && heights[i] < heights[s.top()]) { 13. int height = heights[s.top()]; 14. s.pop(); 15. int L\_idx = s.top(); 16. int R\_idx = i; 17. res = max(res, height \* (R\_idx-L\_idx-1)); 18. } 19. s.push(i); 20. } 21. return res; 22. } 23. }; |

## 0215 数组中的第K个最大元素

链接：https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

标签：堆，分治

### 精选题解

* 官方题解- 数组中的第K个最大元素
  + <https://leetcode-cn.com/problems/kth-largest-element-in-an-array/solution/shu-zu-zhong-de-di-kge-zui-da-yuan-su-by-leetcode-/>
* 超详细！详解一道高频算法题：数组中的第 K 个最大元素-五分钟学算法
  + <https://www.cxyxiaowu.com/3024.html>
* 堆树（最大堆、最小堆）详解
  + <https://blog.csdn.net/guoweimelon/article/details/50904346>
* 【数组中的第K个最大元素】堆的使用和实现 - 数组中的第K个最大元素
  + <https://leetcode-cn.com/problems/kth-largest-element-in-an-array/solution/215-by-ikaruga/>

### 关键思路1：快速选择

核心思路：比基准数字小的数放左边，比它大的放右边，这样基准数字的位置就是正确的位置。

实现起来有很多细节需要注意。建议拿纸笔画一画，理解起来更清晰。

* 划分操作：返回随机选取的基准数在原数组中的正确索引，同时保证其左边的数都比它小，右边的数都比它大
* 为了保证线性的时间复杂度，我们每次选取基准数字的索引pivot时，需要随机选取
  + 在C++中，rand() 函数的返回值是 [0, INT\_MAX]，因此需要模上 (R-L+1) 再加L（代码第5行）
  + 设随机选择的基准索引为pivot，交换a[pivot] 和a[R]，目的是便于后面遍历，同时记录该基准值a[pivot]（代码第6 ~ 7行）
* 从L遍历到R-1（因为R处已经保存了基准数），用i记录当前所有比基准数小的数的索引，也即在索引i左边的数都比基准数小。具体操作是，递增j，遇到a[j] 比基准数小的，就将a[j] 和a[i] 交换，同时将i加1，这样就将所有比基准数小的数都移到了索引i左边，如果理解困难可以拿纸笔画一画（代码第9 ~ 13行）
  + 结束后交换a[i] 和a[R]，这样就保证了基准数在数组正确的位置上（代码第15行）
  + 返回i，这里的i是基准数在数组中的索引（也即从0开始）
* 分治解决：递归搜索目标值，并对数组进行处理（代码第19 ~ 29行）
* 首先调用划分操作得到划分的基准索引，然后对索引左右的区间进行递归
* 返回的是（排序后的）数组中索引应为idx的数的值
* 找到第k大的数
* 这里有个细节，第k大的数，在排序后的数组中的索引应为nums.size()-k，比如说，第1大的数，其索引为nums.size()-1。注意大小顺序和索引的区别。
* 快速选择得到的数的索引，实际上是从小到大的

### 复杂度1：快速选择

* 时间复杂度：O(n)。严谨的证明比较复杂，可以参考《算法导论》（中文第三版）“第7章 快速排序”“7.3 快速排序的随机化版本”和“第9章 中位数和顺序统计量”“9.2 期望为线性时间的选择算法”。
* 空间复杂度：O(logn)。递归的最大深度。

### 代码1：快速选择

| **0215 数组中的第K个最大元素.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/149477642/ |
| 1. class Solution { 2. public: 3. int rand\_partition(vector<int> & a, int L, int R) { 4. *// choose random num in [L,R] as pivot* 5. int pivot = rand()%(R-L+1)+L; 6. swap(a[pivot], a[R]); *// put a[pivot] to last* 7. int last = a[R]; 8. *// search correct order-index of a[pivot] from L* 9. int i = L; *// i: the correct index of a[pivot]* 10. for (int j=L; j<R; ++j) 11. if (a[j] < last) 12. *// increment i to mark num of elements smaller than a[pivot]* 13. swap(a[j], a[i++]); 14. *// put a[pivot] to a[i]* 15. swap(a[i], a[R]); 16. return i; 17. } 18. int quick\_select(vector<int>& a, int L, int R, int idx) { 19. int q = rand\_partition(a, L, R); 20. if (q == idx) 21. return a[q]; 22. else { 23. if (q < idx) 24. return quick\_select(a, q+1, R, idx); 25. else 26. return quick\_select(a, L, q-1, idx); 27. } 28. } 29. int findKthLargest(vector<int>& nums, int k) { 30. time\_t t; 31. srand((unsigned) time(&t)); 32. return quick\_select(nums, 0, nums.size()-1, nums.size()-k); 33. } 34. }; |

### 关键思路2：堆排序

【数组中的第K个最大元素】堆的使用和实现 - 数组中的第K个最大元素：<https://leetcode-cn.com/problems/kth-largest-element-in-an-array/solution/215-by-ikaruga/>

【STL学习】堆相关算法详解与C++编程实现（Heap）<https://blog.csdn.net/xiajun07061225/article/details/8553808>

详解二叉堆（基础篇）：<https://zhuanlan.zhihu.com/p/35094011>

二叉堆及优先级队列的实现原理：<https://leetcode-cn.com/circle/article/bNtb4J/>

《C++标准库》“12特殊容器”“12.3 Priority Queue”和“11 STL算法”“11.9 排序算法”“11.9.4 Heap算法”

《STL源码剖析》“第4章 序列式容器”“4.7 heap”和“4.8 priority\_queue”

本题建立一个k元小根堆，遍历数组中的元素并插入堆中，最后得到的堆顶元素即为第K个最大元素。

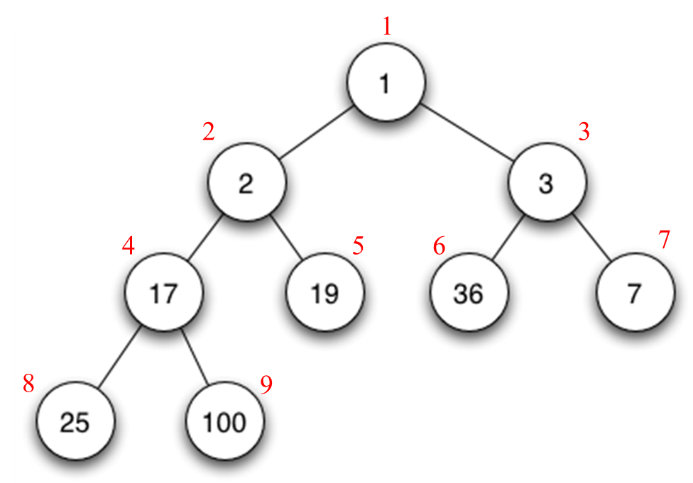
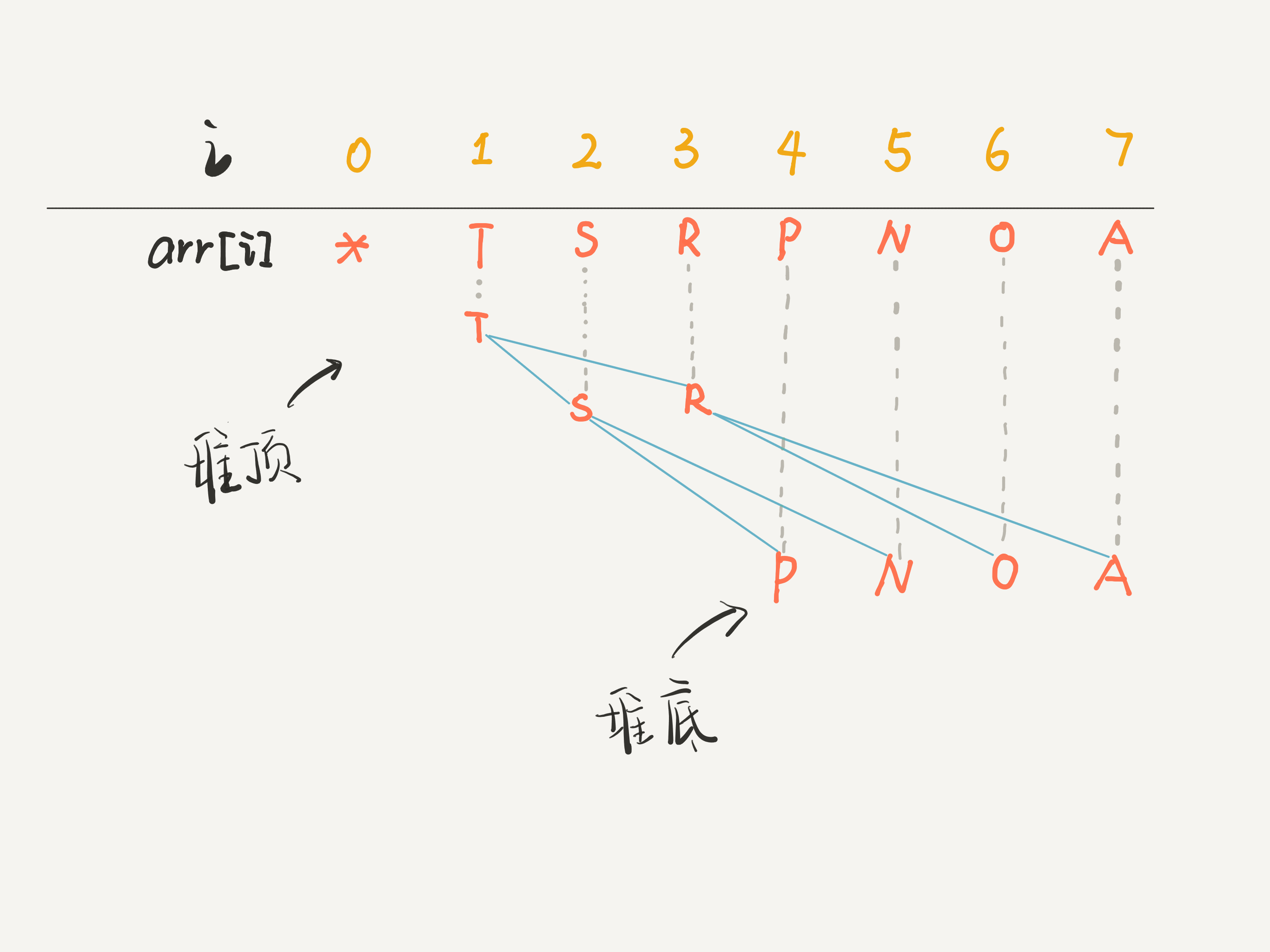
* 在小根堆中，堆顶元素是堆中最小元素，也即剩下的元素都比堆顶大
* 如果堆的大小已经为k，那么遇到新的数时：
  + 若比堆顶元素小，则跳过该步，因为我们只想保留数组中前k个最大的元素
  + 若比堆顶元素大，则弹出栈顶元素，将新数插入堆中，这样就能保证堆中元素一定是目前遍历过的数组中前k个最大的元素，同时插入新元素
* 如果堆的大小未达到k，则直接插入该新数

在C++中，优先队列包含在头文件<queue>当中。priority\_queue<int> 默认是大根堆，其比较函数为less<int>。若需要小根堆，则比较函数应为greater<int>。

当然，不但要会使用STL容器，还要能够自己手动实现一个堆（代码2-2）。

二叉堆：也叫优先队列，是一个**完全二叉树**，任意节点的优先级都不低于其子节点，比如大根堆，任意节点的值都不小于其子节点。

通常使用**数组**来实现堆。

堆，红色为节点在层序遍历中的索引（从1开始）

堆的索引满足如下关系：（为了表示方便，索引从1开始）

* 索引为i的节点的两个子节点索引分别为i\*2和i\*2+1
* 索引为i的节点的父节点索引为 i/2
* 第一个非叶节点的索引为 size/2（也即最右叶节点的父节点）

这样就建立起根据索引在父子节点之间互相跳转，非常有用。

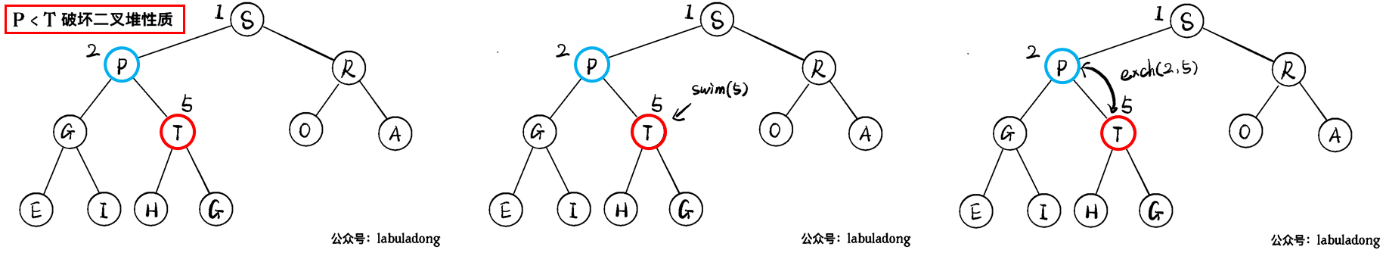
堆的常见应用有两个：**堆排序**和**优先队列**。

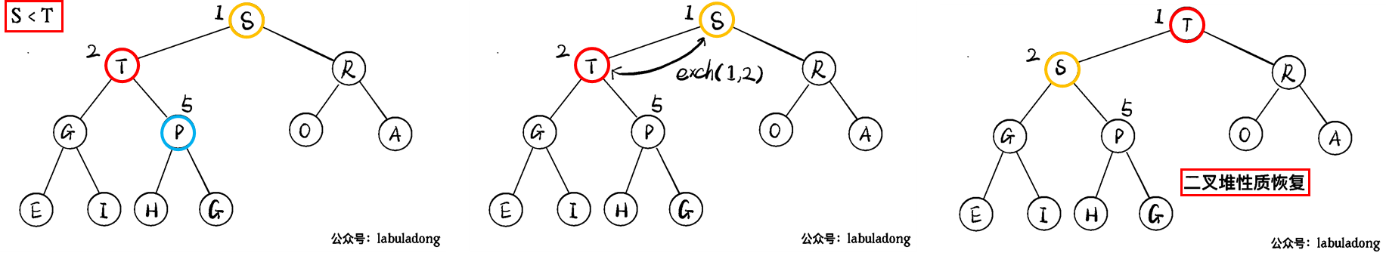
堆的主要操作也有两个：**下沉**（sink）和**上浮**（swim）。

* 下沉和上浮是互逆的操作，但是同时提供这两个，是因为我们一般只能在堆顶和堆底进行操作。
* 下沉：如果当前结点比某个子节点小，那么就将其和该子节点交换
* 上浮：如果当前节点比父节点大，那么就将其和父节点交换

上浮（大根堆）的过程如下图所示：（顺序为从左到右，从上到下）

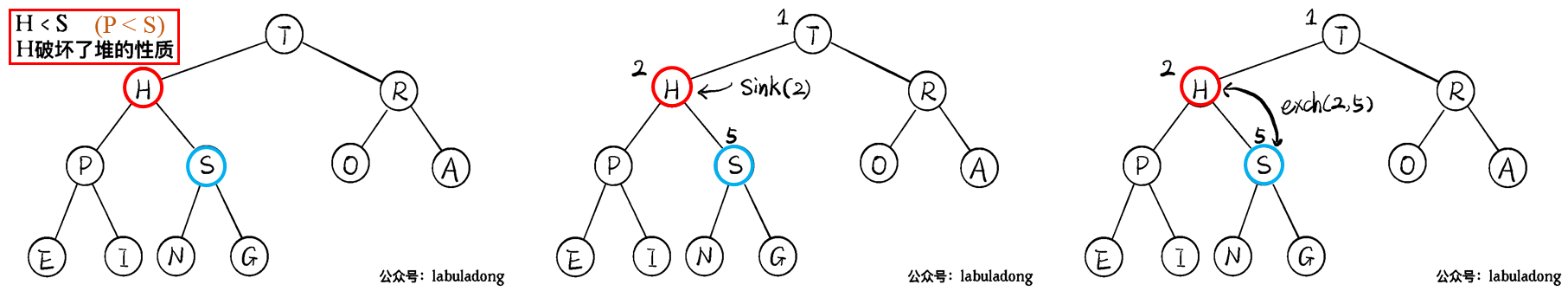
* 以图1中的5号节点T为例，T > P（父），则T应上浮，与P交换；交换后T > S（父），则T继续上浮，与S交换。

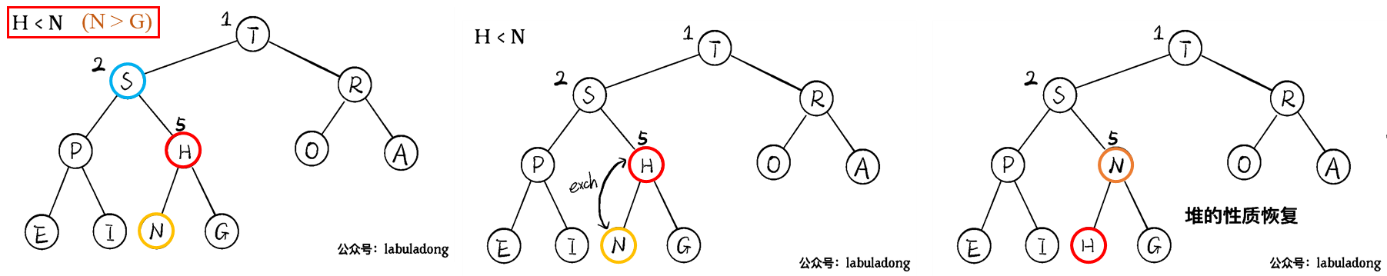




下沉（大根堆）的过程如下图所示：（顺序为从左到右，从上到下）

* 下沉和上浮有一点不同，就是上浮只需要和一个父节点比较大小，下沉则需要和两个子节点比较，如果需要下沉，应该是和较大的那个子节点交换（大根堆）
* 以图1中的2号节点H为例，H < S（P < S），则H应该下沉，故H和S交换；交换后，H < N（N > G），故H继续下沉，和N交换。





优先队列一般提供两个功能，其实就是基于上浮和下沉的操作：

* **插入**（insert） / **压入**（push）
  + 将元素插入到堆底最后一个节点，然后上浮到正确位置
* **删除根结点**（最大堆deleteMax，最小堆deleteMin） / **弹出**（pop）
  + 将堆顶的根节点root和堆底最后一个节点last交换，然后删除root（也即pop\_back数组中的最后一个元素），再将（位于根节点处的）last下沉到正确位置

插入和删除的时间复杂度均为O(logK)，其中K为堆中节点总数。

* 因为操作只有上浮和下沉，所以执行次数最多为堆（二叉树）的高度，也即O(logK)。

**具体的实现参见：cs\_notes/\_tests/impl\_priority\_queue\_simple.cpp**

### 代码2-1：堆排序

| **0215 数组中的第K个最大元素 - 堆.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/149574342/ |
| 1. class Solution { 2. public: 3. int findKthLargest(vector<int>& nums, int k) { 4. priority\_queue<int, vector<int>, greater<int>> pq; *// min heap* 5. for (auto const &n: nums) { 6. if (pq.size()==k) 7. if (pq.top()>=n) 8. continue; 9. else 10. pq.pop(); 11. pq.push(n); 12. } 13. return pq.top(); 14. } 15. }; |

### 代码2-2：堆排序，自己实现优先队列

| **0215 数组中的第K个最大元素 - 堆 - 自实现.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/150278842/ |
| 1. template <typename Comparator> 2. class my\_priority\_queue { 3. public: 4. vector<int> v; 5. Comparator cmp; 6. my\_priority\_queue() : v() { 7. v.push\_back(-1); 8. }; 9. void push(const int & t) { 10. v.push\_back(t); 11. int c\_idx = v.size()-1; 12. int p\_idx = c\_idx/2; 13. while (c\_idx > 1 && cmp(v[p\_idx], v[c\_idx])) { 14. swap(v[p\_idx], v[c\_idx]); 15. c\_idx = p\_idx; 16. p\_idx = c\_idx/2; 17. } 18. } 19. int get\_vip\_child\_idx(int p\_idx) { 20. int c\_idx = p\_idx \* 2; 21. if (c\_idx<v.size()-1 && cmp(v[c\_idx], v[c\_idx+1])) 22. ++c\_idx; 23. return c\_idx; 24. } 25. int pop() { 26. int root\_val = v[1]; 27. int back\_val = v[v.size()-1]; 28. swap(v[1], v[v.size()-1]); 29. v.pop\_back(); 30. int p\_idx = 1; 31. int c\_idx = get\_vip\_child\_idx(p\_idx); 32. while (c\_idx<v.size() && cmp(v[p\_idx], v[c\_idx])) { 33. swap(v[p\_idx], v[c\_idx]); 34. p\_idx = c\_idx; 35. c\_idx = get\_vip\_child\_idx(p\_idx); 36. } 37. return root\_val; 38. } 39. int top() { 40. return v[1]; 41. } 42. int size() { 43. return v.size()-1; 44. } 45. }; 46. class Solution { 47. public: 48. int findKthLargest(vector<int>& nums, int k) { 49. my\_priority\_queue<greater<int>> pq; *// greater: min heap* 50. for (auto const & n : nums) { 51. if (pq.size()==k) 52. if (pq.top() >= n) 53. continue; 54. else 55. pq.pop(); 56. pq.push(n); 57. } 58. return pq.top(); 59. } 60. }; |

## 0227 基本计算器 II + 0224 基本计算器

链接：<https://leetcode-cn.com/problems/basic-calculator-ii/solution/>

https://leetcode-cn.com/problems/basic-calculator/solution/

标签：栈，字符串

### 精选题解

* 拆解复杂问题：实现一个完整计算器 - 基本计算器 II
  + <https://leetcode-cn.com/problems/basic-calculator-ii/solution/chai-jie-fu-za-wen-ti-shi-xian-yi-ge-wan-zheng-ji-/>
  + <https://leetcode-cn.com/problems/basic-calculator-ii/solution/chai-jie-fu-za-wen-ti-shi-xian-yi-ge-wan-zheng-ji-/330884>

### 关键思路

此题建议拿纸笔模拟一下出栈入栈和括号递归的过程，就容易理解了。下面是一个典型的输入例子。

(1 + (3 - 2) \* 5 + 8 / 4)

基本思路如下：

* 运算符op初始设为加号’+’，之后遇到**新的运算符、右括号或字符串末尾**，就将当前的数入栈（加减号），或者将旧的运算符和当前的数运算后再入栈（乘除号），并更新运算符op，重置数字num。**遇到右括号或字符串末尾，则将栈清空。**
  + 清空栈实质上就是将其中的数累加，换句话说，栈中只保留经过运算符“运算”后的数。
  + 很多题解都单独考虑忽视空格，这是因为他们的方法在判断是否更新运算符时，采用了(s[i]<’0’ || s[i]>’9’) 这种写法（代码第20行），实际上这样做并不严谨。
  + 应当采取的做法是，只考虑有可能更新运算符的情况，也即遇到运算符、右括号和字符串末尾（代码第21行），这样其他的字符就自动被忽略。

以上式为例，简述流程：

* 1后面遇到加号+，此时初始运算符为+，则将 +1入栈，更新运算符为 +；
* ……（处理括号的部分下面单独写）
* 8后面遇到除号 /，此时暂存运算符为 +，则将 +8 入栈，更新运算符为 /；
* 遇到4后面的右括号，此时暂存运算符为 /，故将运算得到的num也即4，和栈顶的 +8 结合运算，+8/4=+2，将其入栈。
* 此时已到末尾，将栈中数字累加

如何处理括号：括号本质上就是下一层递归。

* 遇到左括号 (：进入递归。注意在进入下一层递归时，字符串索引应当以引用传递，索引需要 ++i；回到本层递归时，也需要 ++i（代码第17行），这样可以直接继续处理其匹配的右括号后面一个字符。
* 遇到右括号 )：右括号是本层递归的终止标志。一方面，和遇到其他运算符相似，需要将当前数作运算，然后入栈出栈（代码第21行）；另一方面，和遇到字符串末尾相似，需要跳出循环（代码第43 ~ 45 行），将栈中数字累加清空（代码第49 ~ 53 行），最终跳出本层递归。
* 注意，每次新的递归层中，申请的栈都只属于该层，因此最后清空栈时得到的就是本层的计算结果，所以最后只需返回一个数就行。
* 如何结合数字，很简单，直接num = num\*10+(s[i]-‘0’)。
* 注意后面必须加括号，否则类型转换会出错（代码第12 ~ 14行）。

### 复杂度

* 时间复杂度：O(n)。其中n表示字符串长度。遍历所有字符，对k个字符，伴随的操作时间都不超过O(k)。
* 空间复杂度：O(n)。申请的变量栈stk空间最多O(n/2)个数；递归最多有O(n/2)的栈空间。

### 代码

| **0227 基本计算器 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141344102/ |
| 1. class Solution { 2. public: 3. vector<char> op\_v = {'+','-','\*','/'}; 4. int helper(string &s, int &i) { 5. char op = '+'; 6. stack<int> stk; 7. int num = 0; 8. int res = 0; 9. int top = 0; 10. for (i; i<s.size(); ++i) { 11. if (s[i]>='0' && s[i]<='9') { *// 计算数字* 12. num = num \* 10 + (s[i]-'0'); 13. } 14. if (s[i]=='(') { *// 左括号进入递归，在新的递归中，stk 和 op 都被重置* 15. num = helper(s, ++i); 16. ++i; *// 出了右括号，将指针右移一位* 17. } 18. *// if (i >= s.size()-1 || ((s[i]<'0' || s[i]>'9') && s[i] != ' ')) {* 19. if (i>=s.size()-1 || find(op\_v.begin(), op\_v.end(), s[i]) != op\_v.end() || s[i]==')') { 20. *// 遇到新的运算符、右括号和字符串末尾，则出栈入栈，这一写法无需考虑其他特殊字符（比如空格）* 21. if (op=='+') { 22. stk.push(num); 23. } 24. if (op=='-') { 25. stk.push(-num); 26. } 27. if (op=='\*') { 28. top = stk.top(); 29. stk.pop(); 30. stk.push(top\*num); 31. } 32. if (op=='/') { 33. top = stk.top(); 34. stk.pop(); 35. stk.push(top/num); 36. } 37. op = s[i]; *// 更新下一次的运算符，注意必须写在处理完运算符的步骤后面* 38. num = 0; 39. } 40. if (s[i]==')') { *// 右括号跳出循环，执行末尾的清栈步骤，然后回到上一层递归* 41. break; 42. } 43. } 44. *// 计算栈中数字之和* 45. while (!stk.empty()) { 46. res += stk.top(); 47. stk.pop(); 48. } 49. return res; 50. } 51. int calculate(string s) { 52. int i = 0; 53. return helper(s, i); 54. } 55. }; |

## 0946 验证栈序列 + 剑指 31 栈的压入、弹出序列

链接：https://leetcode-cn.com/problems/zhan-de-ya-ru-dan-chu-xu-lie-lcof/

标签：栈

### 精选题解

* 面试题31. 栈的压入、弹出序列（模拟，清晰图解） - 栈的压入、弹出序列
  + https://leetcode-cn.com/problems/zhan-de-ya-ru-dan-chu-xu-lie-lcof/solution/mian-shi-ti-31-zhan-de-ya-ru-dan-chu-xu-lie-mo-n-2/

### 关键思路

* 使用一个栈s来模拟入栈过程，依次将pushed中的元素入栈，用i指向popped中的元素
* 当s非空且s.top()==popped[i] 时，弹出s栈顶并将i加1，重复该判断直到二者不等

### 复杂度

* 时间复杂度：O(n)。遍历所有元素，以及出栈入栈操作。
* 空间复杂度：O(n)。申请的栈空间大小。

### 代码

| **0946 验证栈序列 + JZ-31 栈的压入、弹出序列.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142195251/ |
| 1. class Solution { 2. public: 3. bool validateStackSequences(vector<int>& pushed, vector<int>& popped) { 4. stack<int> s; 5. int i = 0; 6. for (auto num: pushed) { 7. s.push(num); 8. while (!s.empty() && s.top() == popped[i]) { 9. s.pop(); 10. ++i; 11. } 12. } 13. return s.empty(); 14. } 15. }; |

# 动态规划

## 0040 接雨水

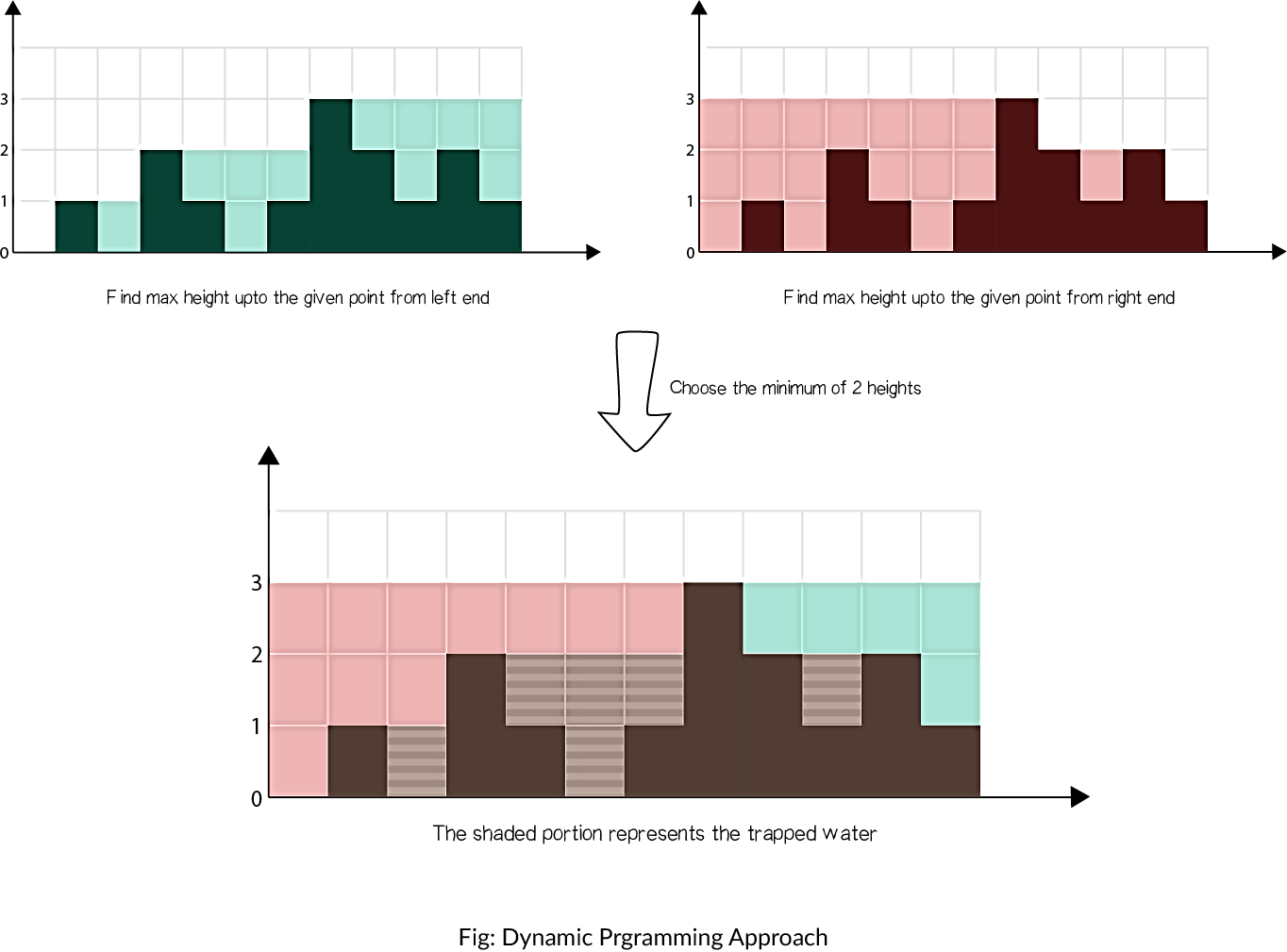
链接：https://leetcode-cn.com/problems/trapping-rain-water/

标签：动态规划，双指针

### 精选题解

* 官方题解 - 接雨水
  + <https://leetcode-cn.com/problems/trapping-rain-water/solution/jie-yu-shui-by-leetcode/>
  + <https://leetcode-cn.com/problems/trapping-rain-water/solution/jie-yu-shui-by-leetcode/327718>

### 关键思路



**动态规划：**

* 两趟遍历，分别记录数组中每个数height[i] 左边所有元素最大数L\_max[i] 和右边所有元素最大数R\_max[i]，注意不包含自己，且初始时L\_max[0] 和R\_max[n-1] 都为0。
* 然后再遍历一遍，累加每个数当前位置能接的水的容量（棕色阴影部分），也即：
  + 应用L\_max[i] 和 R\_max[i] 的较小值减去height[i]
  + 注意外层和0取最大值的含义，如果当前数比左右两侧都高，则为0，不可能为负

**双指针：**

* 使用两个指针L和R记录当前处理的数组元素的索引，从两边向中间移动
* 整数L\_max和R\_max记录heigh[L] 左边的最大值和height[R] 右边的最大值
* 重要事实1：当前数height[i] 接水的容量等于其左边元素最大值L\_max和右边元素最大值R\_max中较小的那个数减去当前数，且最小为0（也即上面的公式）
  + 任意一个数height[i] 的容量，要么被左边最大数减去，要么被右边最大数减去
* 重要事实2：对于从左向右遍历的指针L，其L\_max一定是左侧最大的，而R\_max则未必是右侧最大的，因为从L到R之间的元素都在L右侧，尚未遍历。
  + 因此如果此时L\_max < R\_max，这意味着无论右边的R\_max如何（R\_max只可能变大而不会变小），此时的L\_max都已经决定了位置L处的数能装多少水了。
  + 同理，对于从右向左遍历的指针R，若R\_max < L\_max，则此时的R\_max都已经决定了位置R处的数能装多少水了。
  + 当L\_max == R\_max时，无论放入上面那种情况都可以。本人代码将其纳入更新L的部分，也即使用L\_max <= R\_max的判断条件。

### 复杂度

* 时间复杂度：动态规划和双指针都是O(n)，不过双指针实际运行次数更少。
* 空间复杂度：动态规划为O(n)，双指针为O(1)。

### 代码1：动态规划

| **0042 接雨水 - 动态规划.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152560018/ |
| 1. class Solution { 2. public: 3. int trap(vector<int>& height) { 4. if (height.empty()) 5. return 0; 6. int n = height.size(); 7. vector<int> L\_max(n,0); 8. for (int i=1; i<n; ++i) 9. L\_max[i] = max(L\_max[i-1], height[i-1]); 11. vector<int> R\_max(n,0); 12. for (int i=n-2; i>=0; --i) 13. R\_max[i] = max(R\_max[i+1], height[i+1]); 15. int res = 0; 16. for (int i=0; i<n; ++i) 17. res += max(0, min(L\_max[i], R\_max[i])-height[i]); 18. return res; 19. } 20. }; |

### 代码2：双指针

| **0042 接雨水 - 双指针.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152572563/ |
| 1. class Solution { 2. public: 3. int trap(vector<int>& height) { 4. if (height.empty()) 5. return 0; 6. int res = 0; 7. int L = 0, R = height.size()-1; 8. int L\_max = 0, R\_max = 0; 9. while (L<=R) { 10. if (L\_max <= R\_max) { 11. res += max(0, L\_max-height[L]); 12. L\_max = max(L\_max, height[L]); 13. ++L; 14. } else { 15. res += max(0, R\_max-height[R]); 16. R\_max = max(R\_max, height[R]); 17. --R; 18. } 19. } 20. return res; 21. } 22. }; |

## 0053 最大子序和.cpp

链接：https://leetcode-cn.com/problems/maximum-subarray/

标签：动态规划

### 精选题解

* 官方题解 - 最大子序和
  + https://leetcode-cn.com/problems/maximum-subarray/solution/zui-da-zi-xu-he-by-leetcode-solution/

### 关键思路

：以第i个数结尾的连续子数组的最大和。

有递推关系：

* 因为是以第i个数结尾，所以无论如何都要用这个数只是取决于 是否为正

最后的结果也即

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(1)。

### 代码

| **0053 最大子序和.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152404619/ |
| 1. class Solution { 2. public: 3. int maxSubArray(vector<int>& nums) { 4. int n = nums.size(); 5. if (n==0) return 0; 6. int res = nums[0]; 7. int pre = 0; 8. for (auto const &x : nums) { 9. pre = max(pre+x, x); 10. res = max(res, pre); 11. } 12. return res; 13. } 14. }; |

## 0070 爬楼梯

链接：https://leetcode-cn.com/problems/climbing-stairs/

标签：动态规划

### 精选题解

略。

### 关键思路

注意初始条件，然后有 。

### 复杂度

* 时间复杂度：O(n)。
* 空间复杂度：O(n)。可以用两个中间变量优化成O(1)。

### 代码

| **0070 爬楼梯.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152519412/ |
| 1. class Solution { 2. public: 3. int climbStairs(int n) { 4. if (n<=2) 5. return n; 6. vector<int> dp(n+1,0); 7. dp[1] = 1, dp[2] = 2; 8. for (int i=3; i<=n; ++i) { 9. dp[i] = dp[i-1] + dp[i-2]; 10. } 11. return dp[n]; 12. } 13. }; |

## 0121 买卖股票的最佳时机

链接：https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/

标签：动态规划，单调栈

### 精选题解

* C++ 利用哨兵👨‍✈️，维护一个单调栈📈(图解，直观掌握) - 买卖股票的最佳时机
  + <https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/solution/c-li-yong-shao-bing-wei-hu-yi-ge-dan-diao-zhan-tu-/>
* 股票问题（Python3、C++） - 买卖股票的最佳时机 - 力扣（LeetCode）
  + <https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/solution/gu-piao-wen-ti-python3-c-by-z1m/>

### 关键思路

动态规划。下面是O(n) 空间复杂度的，实际上可以优化成O(1)，也即将dp[i] 用单个变量res替换即可。

min\_price = min(min\_price, prices[i]);

dp[i] = max(dp[i-1], prices[i]-min\_price);

### 代码1：动态规划

| **0121 买卖股票的最佳时机.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152321994/ |
| 1. class Solution { 2. public: 3. int maxProfit(vector<int>& prices) { 4. int n = prices.size(); 5. if (n==0) 6. return 0; 7. int min\_price = prices[0]; 8. vector<int> dp(n,0); 9. for (int i=1; i<n; ++i) { 10. min\_price = min(min\_price, prices[i]); 11. dp[i] = max(dp[i-1], prices[i]-min\_price); 12. } 13. return dp[n-1]; 14. } 15. }; |

### 代码2：O(1)

| **0121 买卖股票的最佳时机 - O1.cpp** |
| --- |
|  |
| 1. class Solution { 2. public: 3. int maxProfit(vector<int>& prices) { 4. int n = prices.size(); 5. if (n==0) 6. return 0; 7. int min\_price = prices[0]; 8. int res=0; 9. for (int i=1; i<n; ++i) { 10. min\_price = min(min\_price, prices[i]); 11. res = max(res, prices[i]-min\_price); 12. } 13. return res; 14. } 15. }; |

## 0238 除自身以外数组的乘积 + 剑指 66 构建乘积数组

链接：<https://leetcode-cn.com/problems/product-of-array-except-self/>

https://leetcode-cn.com/problems/gou-jian-cheng-ji-shu-zu-lcof/

标签：动态规划

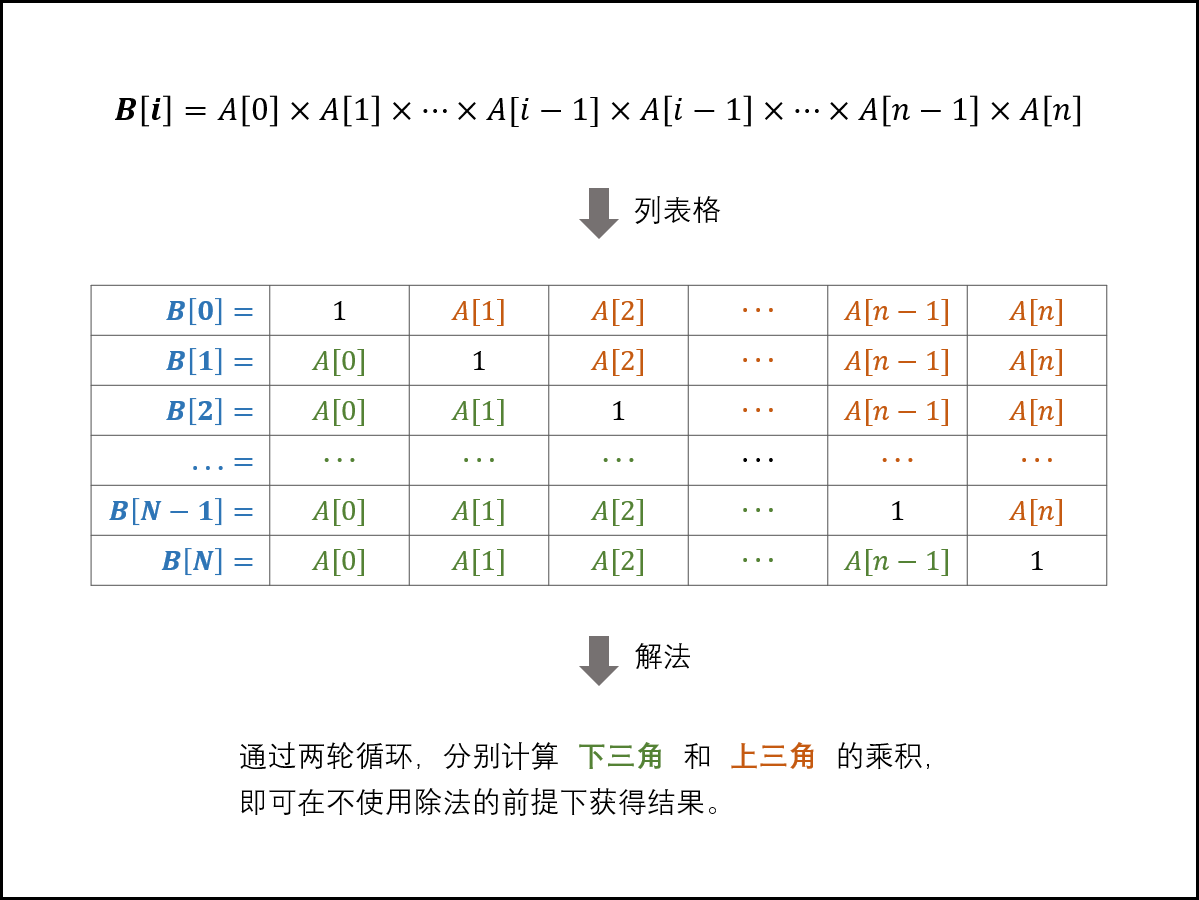
### 精选题解

* 面试题66. 构建乘积数组（表格分区，清晰图解） - 构建乘积数组
  1. <https://leetcode-cn.com/problems/gou-jian-cheng-ji-shu-zu-lcof/solution/mian-shi-ti-66-gou-jian-cheng-ji-shu-zu-biao-ge-fe/>
  2. <https://leetcode-cn.com/problems/gou-jian-cheng-ji-shu-zu-lcof/solution/mian-shi-ti-66-gou-jian-cheng-ji-shu-zu-biao-ge-fe/447760>
* 对称遍历 - 构建乘积数组
  1. https://leetcode-cn.com/problems/gou-jian-cheng-ji-shu-zu-lcof/solution/gou-jian-cheng-ji-shu-zu-dui-cheng-bian-li-by-huwt/

### 关键思路

不能使用除法。对于nums[i]，就是将0 ~ i-1和i+1 ~ n-1索引处的数相乘。

两趟遍历（可以压缩成一次，但没必要），先将i左边的数都乘上，再将i右边的数都乘上。拿纸笔画一下，就更好理解了。



对于res[i] 来说：

* 乘左边的数时（绿色），索引为i-1（代码第9行）
* 乘右边的数时（橙色），从右向左遍历，需要使用一个中间变量R，来记录右边所有数的乘积（代码第13 ~ 15行）

### 复杂度

* 时间复杂度：O(n)。两次线性遍历。
* 空间复杂度：O(1)。只用了中间变量R。结果必须返回的res不计入空间复杂度。

### 代码

| **0238 除自身以外数组的乘积 + JZ-66 构建乘积数组.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148638299/ |
| 1. class Solution { 2. public: 3. vector<int> productExceptSelf(vector<int>& nums) { 4. int n = nums.size(); 5. if (n==0) return {}; 6. vector<int> res(n, 1); 7. *// multiply numsll elements left of current idx* 8. for (int i=1; i<n; ++i) { 9. res[i] = res[i-1] \* nums[i-1]; 10. } 11. *// multiply numsll elements right of current idx* 12. int R = 1; 13. for (int i=n-1; i>=0; --i) { 14. res[i] \*= R; 15. R \*= nums[i]; 16. } 17. return res; 18. } 19. }; |

## 0263 丑数

链接：https://leetcode-cn.com/problems/ugly-number/

标签：数学

### 精选题解

* C语言简单解决263.丑数 - 丑数
  + https://leetcode-cn.com/problems/ugly-number/solution/cyu-yan-jian-dan-jie-jue-263chou-shu-by-t4gpd/

### 关键思路

很简单，直接看代码。

### 复杂度

* 时间复杂度：O(logn)。最多除 次。
* 空间复杂度：O(1)。

### 代码

|  |
| --- |
| https://leetcode-cn.com/submissions/detail/141431329/ |
| 1. class Solution { 2. public: 3. bool isUgly(int num) { 4. if (num<=0) 5. return false; 6. while (num != 1) { 7. if (num%2==0) 8. num /= 2; 9. else if (num%3==0) 10. num /= 3; 11. else if (num%5==0) 12. num /= 5; 13. else 14. return false; 15. } 17. return true; 18. } 19. }; |

## 0264 丑数 II + 剑指 49 丑数 II

链接：<https://leetcode-cn.com/problems/chou-shu-lcof/>

https://leetcode-cn.com/problems/ugly-number-ii

标签：数学，动态规划

### 精选题解

* 丑数II， 合并 3 个有序数组， 清晰的推导思路 - 丑数
  1. <https://leetcode-cn.com/problems/chou-shu-lcof/solution/chou-shu-ii-qing-xi-de-tui-dao-si-lu-by-mrsate/>
* 面试题49. 丑数（动态规划，清晰图解） - 丑数
  1. <https://leetcode-cn.com/problems/chou-shu-lcof/solution/mian-shi-ti-49-chou-shu-dong-tai-gui-hua-qing-xi-t/>

### 关键思路

相当于合并3个有序数组，nums2、nums3和nums5，分别储存每个丑数只乘2、3、5得到的数。任何一个丑数乘上2、3、5之后都必定落在这这三个数组之一，因此将这三个数组合并之后就得到真正的丑数数组，设为dp，不过由于这三个数组存在重合，因此需要去重。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| dp[i] | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 |  |
| nums2 | 1 | 1\*2 | 2\*2 | 3\*2 | 4\*2 | 5\*2 | 6\*2 | 8\*2 | 9\*2 |
| nums3 | 1 | 1\*3 | 2\*3 | 3\*3 | 4\*3 | 5\*3 | 6\*3 | 8\*3 | 9\*3 |
| nums5 | 1 | 1\*5 | 2\*5 | 3\*5 | 4\*5 | 5\*5 | 6\*5 | 8\*5 | 9\*5 |

p2、p3、p5的含义有点不好理解。p*k*的含义是，最新添加进dp并且包含因子*k*的数，在数组nums*k*中的索引。这里的最新不是指最新的dp[i]，而是指包含因子*k*的数中最新添加的。

以下表中i=3一列为例。此时dp[i]=4，p2=2，p3=1，p5=0，也即，包含因子2的最新的丑数为nums2[p2] =4，包含因子3的最新的丑数为nums3[p1]=3，包含因子5的最新的丑数为nums5[p5]=1。其中dp[p2]=4与当前的dp[i]吻合。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| dp[i] | 1 | 2 | 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | 15 | 16 | 18 |
| p2 | 0 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 5 | 6 | 6 | 7 | 8 |
| p3 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 6 |
| p5 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 3 |
| dp[p2]\*2 | 2 | 4 | 4 | 6 | 6 | 8 | 10 | 10 | 12 | 16 | 16 | 18 |  |
| dp[p3]\*3 | 3 | 3 | 6 | 6 | 6 | 9 | 9 | 12 | 12 | 15 | 18 | 18 |  |
| dp[p5]\*5 | 5 | 5 | 5 | 5 | 10 | 10 | 10 | 10 | 15 | 15 | 20 | 20 |  |

那么如何得到当前丑数的下一个丑数呢？只需要计算出三个nums*k*的数组的下一个数，其中最小的就是下一个丑数。如果这个新的丑数和nums*k*中的最新的数相等，则表明该丑数包含在该数组中，则将对应的p*k*加1。

所以我们的代码需要做两件事：

（1）得到三个数组中下一个最小值，作为合并后数组的下一个值（代码第8行）；

（2）通过移动三个数组索引来保证去重（代码第9 ~ 11行），并且由于可能在多个数组中出现，因此每个if是独立的。

### 复杂度

* 时间复杂度：O(n)。遍历n个数。
* 空间复杂度：O(n)。dp数组大小。

### 代码

| **JZ-49 0264 丑数 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141425392/ |
| 1. class Solution { 2. public: 3. int nthUglyNumber(int n) { 4. vector<int> dp(n,1); 5. dp[0] = 1; 6. int p2=0, p3=0, p5=0; 7. for (int i=1; i<n; ++i) { 8. dp[i] = min(min(dp[p2]\*2, dp[p3]\*3), dp[p5]\*5); 9. if (dp[i]==dp[p2]\*2) ++p2; 10. if (dp[i]==dp[p3]\*3) ++p3; 11. if (dp[i]==dp[p5]\*5) ++p5; 12. } 13. return dp[n-1]; 14. } 15. }; |

## 0279 完全平方数

链接：<https://leetcode-cn.com/problems/perfect-squares/>

标签：数学，动态规划

### 精选题解

* 画解算法：279. 完全平方数 - 完全平方数
  + <https://leetcode-cn.com/problems/perfect-squares/solution/hua-jie-suan-fa-279-wan-quan-ping-fang-shu-by-guan/>
* 官方题解 - 完全平方数
  + https://leetcode-cn.com/problems/perfect-squares/solution/wan-quan-ping-fang-shu-by-leetcode/

### 关键思路

一种是动态规划，一种是数学定理。

**1、动态规划**

* 用dp[i] 表示将和为i的完全平方数的最小个数，初值为i（组成的数全为1）
* 则dp[i] = min(dp[i], dp[i-j\*j])，for j from 1 to ，for i from 1 to n
* 返回 dp[n]

**2、数学定理**

* 四平方和定理：每个自然数都可以表示为四个整数的平方和，因此题中每个输入的结果最多为4。
  + 若 ，返回4
  + 若n本身的完全平方数，返回1
  + 若n能由两个完全平方数表示，返回2
  + 其余情况返回3
* 上面之所以用这样的判断顺序，主要是为了降低时间复杂度，毕竟判断是否是平方数代价较大，尽可能早地判断其他情形了。

### 复杂度

动态规划：

* 时间复杂度：O()。外层遍历n个数，内层遍历 个数。
* 空间复杂度：O(n)。dp数组的大小。

数学定理：

* 时间复杂度：O()。判断能否用两个完全平方数表示时，需要遍历 个数。
* 空间复杂度：O(1)。

### 代码1 – 动态规划

| **0279 完全平方数 – 动态规划.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141556794/ |
| 1. class Solution { 2. public: 3. int numSquares(int n) { 4. vector<int> dp(n+1,0); 5. for (int i=1; i<=n; ++i) { 6. dp[i] = i; 7. for (int j=1; i-j\*j>=0; ++j) { 8. dp[i] = min(dp[i], dp[i-j\*j]+1); 9. } 10. } 11. return dp[n]; 12. } 13. }; |

### 代码2 – 数学定理

| **0279 完全平方数 – 公式.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141560552/ |
| 1. class Solution { 2. public: 3. bool isSquare(int x) { 4. int sq = int(sqrt(x)); 5. return sq\*sq==x; 6. } 7. int numSquares(int n) { 8. while (n%4==0) 9. n>>=2; 10. if (n%8==7) 11. return 4; 12. if (isSquare(n)) 13. return 1; 14. for (int i=1; i<=int(sqrt(n)); ++i){ 15. if (isSquare(n-i\*i)) 16. return 2; 17. } 18. return 3; 19. } 20. }; |

## 0300 最长递增（上升）子序列

链接：https://leetcode-cn.com/problems/longest-increasing-subsequence/

标签：动态规划，贪心，二分查找

### 精选题解

* 官方题解 - 最长递增子序列
  + <https://leetcode-cn.com/problems/longest-increasing-subsequence/solution/zui-chang-shang-sheng-zi-xu-lie-by-leetcode-soluti/>
  + https://leetcode-cn.com/problems/longest-increasing-subsequence/solution/zui-chang-shang-sheng-zi-xu-lie-by-leetcode-soluti/291805

### 关键思路

两种方法：动态规划，贪心+二分

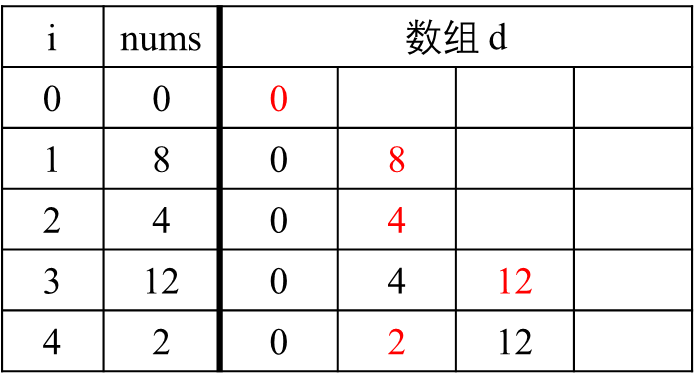
**动态规划：**

* dp[i]：以nums[i] 结尾的子序列的最大长度
* 递推关系：，其中 j 满足 且 。
* 注意dp的初值为1，也即序列中仅有当前数字，否则递推dp[j]+1无法得到正确答案。

**贪心+二分：**略复杂，建议拿纸笔画一画。

* 要得到最长的上升子序列，则上升速度应该尽可能慢，也即在末尾添加的数字尽可能小
* d[k]：长度为k的最长上升子序列末尾元素的最小值
  + d[k] 是关于i单调递增的：若存在j满足 且 ，则将任意长度为i的某最长上升子序列末尾的i-j个元素去掉，那么就是一个长度为j的上升子序列，其末尾元素肯定小于d[i]，也即必有，与假设矛盾，故不存在j满足上述条件，因此 时必有。
* len：目前已经得到的最长上升子序列的长度
* 从前向后遍历nums，遍历到nums[i] 时
  + 若 ，将nums[i] 加入d的末尾，并将len加1
  + 若 ，则二分查找数组d，找到（从右向左）第一个比nums[i] 小的数，比如d[k]，更新d[k+1] = nums[i]
* 也即更新长度为k+1的最长上升子序列末尾的最小值
* 由于这里的d[k] 是从左到右第一个比nums[i] 小的，因此原来的d[k+1] 必然是大于nums[i] 的，故这样更新肯定是把原来的值变得更小
* 这里的二分写法（代码2第14 ~ 24行），参考cs-notes/\_tests/lower\_bound.py的左闭右开写法；最后的结果还要减1，因为lower\_bound是求（从左到右）第一个“不小于”，我们这里要求（从右到左）第一个“小于”，关于这点，参考“后台面经整理.docx”中“lower\_bound和upper\_bound用法”
* 最终返回len，也即最长上升子序列的长度，d则为满足要求的数组

下图是以数组 [0,8,4,12,2] 为例的过程：



### 复杂度

* 时间复杂度：动态规划是O()，贪心二分是O()。
* 空间复杂度：动态规划和贪心二分都是O(n)。

### 代码1：动态规划

| **0300 最长递增子序列 - 动态规划.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152673902/ |
| 1. class Solution { 2. public: 3. int lengthOfLIS(vector<int>& nums) { 4. int n = nums.size(); 5. if (n==0) return 0; 6. vector<int> dp(n,1); 7. for (int i=0; i<n; ++i) { 8. for (int j=0; j<i; ++j) 9. if (nums[i] > nums[j]) 10. dp[i] = max(dp[i], dp[j]+1); 11. } 12. return \*max\_element(dp.begin(), dp.end()); 13. } 14. }; |

### 代码2：贪心+二分

| **0300 最长递增子序列 - 贪心+二分.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152935915/ |
| 1. class Solution { 2. public: 3. int lengthOfLIS(vector<int>& nums) { 4. int n = nums.size(); 5. if (n==0) return 0; 6. vector<int> d; 7. int len = 1; 8. d.push\_back(nums[0]); 9. for (int i=1; i<n; ++i) { 10. if (nums[i] > d[len-1]) { 11. d.push\_back(nums[i]); 12. ++len; 13. } else { 14. *// find (from-right-to-left) first d[i] that less than nums[i]* 15. *// int k = lower\_bound(d.begin(), d.end(), nums[i])-d.begin()-1;* 16. int L=0, R=len, mid; 17. while (L<R) { *// search in [L,R)* 18. mid = L+(R-L)/2; 19. if (d[mid]<nums[i]) 20. L = mid+1; 21. else 22. R = mid; 23. } 24. L = L-1; 25. *////* 26. d[L+1] = nums[i]; 27. } 28. } 29. return len; 30. } 31. }; |

## 0313 超级丑数

链接：<https://leetcode-cn.com/problems/super-ugly-number/>

标签：数学，堆

### 精选题解

### 关键思路

动态规划，思路和“剑指49 / 0264丑数 II”基本一致，只不过数组nums*k*从原来的3个变成了primes.size() 个了。注意dp初始值应当用INT\_MAX填充。

### 复杂度

### 代码

| **0313 超级丑数.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/141979048/ |
| 1. class Solution { 2. public: 3. int nthSuperUglyNumber(int n, vector<int>& primes) { 4. vector<int> dp(n, INT\_MAX); 5. dp[0] = 1; 6. vector<int> ps(primes.size(), 0); 7. for (int i=1; i<n; ++i) { 8. for (int j=0; j<primes.size(); ++j) { 9. dp[i] = min(dp[i], dp[ps[j]] \* primes[j]); 10. } 11. for (int j=0; j<primes.size(); ++j) { 12. if (dp[i] == dp[ps[j]] \* primes[j]) { 13. ++ps[j]; 14. } 15. } 16. } 17. return dp[n-1]; 18. } 19. }; |

## 0647 回文子串

链接：https://leetcode-cn.com/problems/palindromic-substrings/

标签：动态规划

### 精选题解

* 官方题解 - 回文子串
  + <https://leetcode-cn.com/problems/palindromic-substrings/solution/hui-wen-zi-chuan-by-leetcode-solution/>
  + <https://leetcode-cn.com/problems/palindromic-substrings/solution/hui-wen-zi-chuan-by-leetcode-solution/553051>

### 关键思路

有三种思路，动态规划，中心扩展法和Manacher算法。

动态规划：略。

中心扩展法：

* 遍历每个字符，以该字符为中心，向两边扩展，如果扩展到的字符串相同，则+1
* 中心点可能是一个或两个字符，所以代码1中第7 ~ 9行用了一个小技巧，也即使用j来区分中心点的字符个数，并且确定了初始的字符位置

Manacher算法：（TBD）

### 复杂度

中心扩展法：

* 时间复杂度：O()。两层循环。
* 空间复杂度：O(1)。

Manacher算法：（TBD）

### 代码1 – 中心扩展法

| **0647 回文子串.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148686433/ |
| 1. class Solution { 2. public: 3. int countSubstrings(string s) { 4. int n = s.size(); 5. int num = 0; 6. for (int i=0; i<n; ++i) { 7. for (int j=0; j<=1; ++j) { 8. int L = i; 9. int R = i+j; 10. while (L>=0 && R<n && s[L--]==s[R++]) 11. ++num; 12. } 13. } 14. return num; 15. } 16. }; |

### 代码2 – Manacher算法（TBD）

|  |
| --- |
|  |
|  |

## 0005 最长回文子串

链接：https://leetcode-cn.com/problems/longest-palindromic-substring/

标签：动态规划

### 精选题解

* 官方题解 - 最长回文子串
  + https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zui-chang-hui-wen-zi-chuan-by-leetcode-solution/

### 关键思路

部分思路类似“0647 回文子串”。

两种方法：动态规划和中心扩展法。

动态规划：

* dp[i][j]：bool类型，含义是s[i:j] 是否为回文子串，有递推关系

dp[i][j] = dp[i+1][j-1] && s[i]==s[j]

然而i+1看上去是要用到后验的知识，无法递推。

所以这时不能简单地对i和j进行迭代，而是对字符串的长度进行迭代。当字符串的长度为1和2时，很容易判断是否为回文子串。之后当长度大于等于3时，我们知道dp[i+1][j-1] 对应的字符串是比的dp[i][j] 短2个字符的，因此可以递推。

**中心扩展法：**参考“0647 回文子串”。

* 注意代码2第11 ~ 14行：i表示回文串中心，j取值为0和1，用来区分回文中心字符串长度为1和2的情况，往两边扩展时使用L和R变量；注意不要偷懒只用L代替i作为初始的回文中心，不然在j取值为1时，其左边界是j为0时字符串扩展后的值，而不是i的初值。记录L和R距离最大的值是L\_res和R\_res。

### 复杂度

* 时间复杂度：动态规划和中心扩展法都是O()>
* 空间复杂度：动态规划O(n)，中心扩展法O(1)。

### 代码1：动态规划

| **0005 最长回文子串 - 动态规划.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152622548/ |
| 1. class Solution { 2. public: 3. string longestPalindrome(string s) { 4. int n = s.size(); 5. string res; 6. vector<vector<bool>> dp(n, vector<bool>(n,false)); 7. for (int len=0; len<s.size(); ++len) { 8. for (int i=0; i+len<n; ++i) { 9. int j = i+len; 10. if (len==0) 11. dp[i][j] = true; 12. else if (len==1) 13. dp[i][j] = (s[i]==s[j]); 14. else 15. dp[i][j] = dp[i+1][j-1] && (s[i]==s[j]); 16. if (dp[i][j] && (len+1)>res.size()) 17. res = s.substr(i, len+1); 18. } 19. } 20. return res; 21. } 22. }; |

### 代码2：中心扩展法

| **0005 最长回文子串 - 中心扩展.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152644052/ |
| 1. #include <iostream> 2. #include <string> 3. #include <vector> 4. **using** **namespace** std; 5. **class** **Solution** { 6. **public**: 7. string longestPalindrome(string s) { 8. int L\_res=0, R\_res=0; 9. **for** (int i=0; i<s.size(); ++i) { 10. **for** (int j=0; j<=1; ++j) { 11. int L = i; 12. int R = i+j; 13. **while** (L>=0 && R<s.size() && s[L]==s[R]) { 14. --L; ++R; 15. } 16. ++L; --R; 17. **if** (R-L > R\_res-L\_res) { 18. L\_res = L; 19. R\_res = R; 20. } 21. } 22. } 23. **return** s.substr(L\_res, R\_res-L\_res+1); 24. } 25. }; 26. int main() { 27. Solution sol; 28. string s = "cbbd"; 29. cout << sol.longestPalindrome(s) << endl; 31. **return** 0; 32. } |

## 1143 最长公共子序列

链接：https://leetcode-cn.com/problems/longest-common-subsequence/

标签：动态规划

### 精选题解

### 关键思路

### 复杂度

### 代码

## 剑指 46 把数字翻译成字符串

链接：https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/

标签：动态规划

### 精选题解

1. 面试题46. 把数字翻译成字符串（动态规划，清晰图解） - 把数字翻译成字符串
   1. <https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/solution/mian-shi-ti-46-ba-shu-zi-fan-yi-cheng-zi-fu-chua-6/>
2. 官方题解 - 把数字翻译成字符串
   1. <https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/solution/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-by-leetcode-sol/>
   2. <https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/solution/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-by-leetcode-sol/434965>

### 关键思路

动态规划。先写一下常规思路。

将num转换成字符串s，也即 。新增一个字符 ，其能否新增翻译方法同时取决于 和 。

当且仅当如下情况时， 有两种解释方法：

1. 对应的数字在10 ~ 25之间，也即 s[i-1]==’1’ || (s[i-1]==’2’ && s[i] <=5)
2. 对于其他情形， 都只有一种翻译方法。

动态规划数组的含义如下：

1. dp[i+1] 表示以字符 s[i] 结尾的字符串的翻译方法总数
2. 这里用dp[i+1]，是因为我们需要用到 dp[i]=dp[i-1]+dp[i-2] 的递推式，而对于字符串来说，我们从s[1] 开始就应当进行动态规划递推，此时需要判断s[i] 和 s[i-1] 的值，为了保证dp[i-2] 索引合法，故令dp数组开头多了一个数组，这样就变成dp[i+1]=dp[i]+dp[i-1]。当然也可以将dp[0] 视为空字符串对应的翻译方法。

空间复杂度可以优化，因为只涉及到两个递推数，所以用两个变量记录即可。

还有一种不需要转换成字符串的方法，从末尾数字开始递归，判断原理类似，见代码2。

### 复杂度

1. 时间复杂度：O(n)。数字转换成字符串；遍历字符串。
2. 空间复杂度：O(n)。dp数组。

### 代码1 – 转换成字符串，动态规划

| **JZ-46 把数字翻译成字符串 - dp.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143075117/ |
| 1. class Solution { 2. public: 3. int translateNum(int num) { 4. string s = to\_string(num); 5. vector<int> dp(s.size()+1,0); 6. dp[0] = 1; 7. dp[1] = 1; 8. for (int i=1; i<s.size(); ++i) { 9. if (s[i-1]=='1' || (s[i-1]=='2' && s[i]<='5')) 10. dp[i+1] = dp[i] + dp[i-1]; 11. else 12. dp[i+1] = dp[i]; 13. } 14. return dp[s.size()]; 15. } 16. }; |

### 代码2 – 不转换成字符串，递归

| **JZ-46 把数字翻译成字符串 - recursion.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143071925/ |
| 1. class Solution { 2. public: 3. int translateNum(int num) { 4. if (num<10) 5. return 1; 6. if (num%100<26 && num%100>9) 7. return translateNum(num/10) + translateNum(num/100); 8. else 9. return translateNum(num/10); 10. } 11. }; |

## 剑指 60 n个骰子的点数

链接：https://leetcode-cn.com/problems/nge-tou-zi-de-dian-shu-lcof/

标签：数学，动态规划

### 精选题解

1. 详解动态规划及其优化方式 - n个骰子的点数
   1. <https://leetcode-cn.com/problems/nge-tou-zi-de-dian-shu-lcof/solution/nge-tou-zi-de-dian-shu-dong-tai-gui-hua-ji-qi-yo-3/>
2. 概率论乘法公式+动态规划 - n个骰子的点数
   1. https://leetcode-cn.com/problems/nge-tou-zi-de-dian-shu-lcof/solution/gai-lu-lun-cheng-fa-gong-shi-dong-tai-gu-77ts/

### 关键思路

n个骰子共有 种可能，点数之和共有6n-(n-1)=5n+1种可能。

设n个骰子掷出点数和为i的次数为 ，计算其概率则只需除以 。

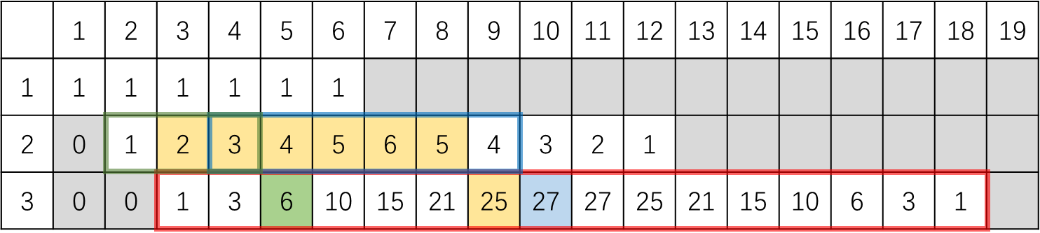
最终的结果是一个数组，每个数组元素为 ，其中 。

状态转移方程为：

具体来说， 表示n个骰子，其由n-1个骰子的情况而来。要从n-1个骰子得到第n个骰子，且和为i，那么：

* 第n个骰子的值为1时，前n-1个骰子点数和应为i-1
* 第n个骰子的值为2时，前n-1个骰子点数和应为i-2
* ……
* 第n个骰子的值为k时，前n-1个骰子点数和应为i-k

图形化表示如下，也即下一行的值是上一行左边连续6个数之和：



二维数组dp的内容

**注意各个循环中的边界条件，最好画个图，这样不容易错。**

* 为了计算方便，行列索引都从1开始。
* 代码第13行的 j-k >= i-1：
  1. 以上图中坐标 (3,5) 处绿色的6为例，其合法的累加范围应为上一行格子中的1、2、3，其列索引为2、3、4
  2. 此时i的值为3，j的值为5，j-k的合法索引范围取决于上一行中的最左边的值，也就坐标 (2,2) 处的1，故j-k的最小值为2，也即i-1

当然，算法还可以在如下两方面提升：

* 空间优化：dp数组不需要二维，只需要一维，因为计算当前值只需要上一行左边格子的值，不会用到下方或右边的格子值。
  1. 注意此时需要从右向左遍历，否则会覆盖掉原本是“上一行”的格子值
  2. 最后返回的时候需要注意将每个数都除以，并且截掉最后一行中前后为0的部分，只保留上图中红色框的内容。erase() 操作可以满足要求，但是容易出错，不如新开一个数组，直接存储结果所需的值。
* 时间优化：以上图中黄色和蓝色为例，从黄色到蓝色的格子，二者相差仅在于黄色最左边的2和蓝色最右边的4，因此无需重新累加6个数，而是在25的基础上加上4-2即可得到27。当然，这个优化降低了代码的可读性，可以不采用。

### 复杂度

* 时间复杂度：O()。计算dp数组中每个元素（格子）的值（累加6个数）都需要常数时间O(1)，一共有6+11+16+…+(5n+1) 个数，也即 个数，故时间复杂度为O()。
* 空间复杂度：O(n)。使用空间优化了的一维dp数组，最多需要容纳3n个数。

### 代码

| **JZ-60 n个骰子的点数.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148455768/ |
| 1. class Solution { 2. public: 3. vector<double> dicesProbability(int n) { 4. vector<int> dp(6\*n+1, 0); 5. for (int i=1; i<=6; ++i) { 6. dp[i] = 1; 7. } 9. *// i: row num; j: col idx; k: 1~6(max)* 10. for (int i=2; i<=n; ++i) { 11. for (int j=6\*i; j>=i; --j) { 12. dp[j]=0; 13. for (int k=1; k<=6 && j-k>=i-1; ++k) { 14. dp[j] += dp[j-k]; 15. } 16. } 17. } 18. double div = pow(6,n); 19. vector<double> res; 20. for (int i=n; i<=6\*n; ++i) { 21. res.push\_back(dp[i]\*1.0 / div); 22. } 23. return res; 24. } 25. }; |

## 剑指 62 圆圈中最后剩下的数字

链接：https://leetcode-cn.com/problems/yuan-quan-zhong-zui-hou-sheng-xia-de-shu-zi-lcof/

标签：数学，动态规划

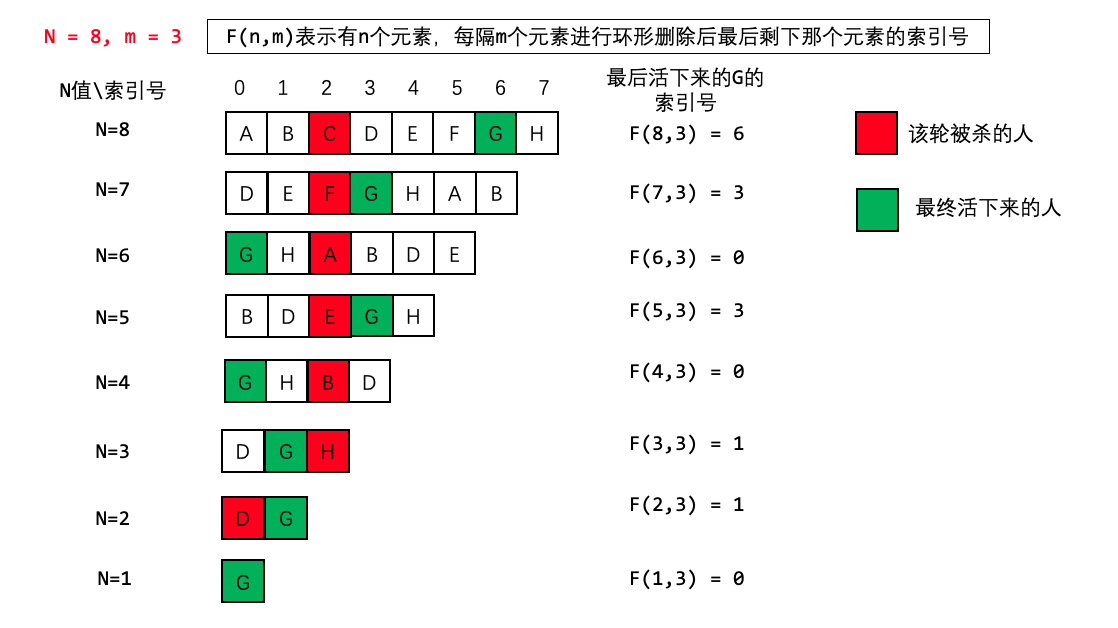
### 精选题解

1. 换个角度举例解决约瑟夫环 - 圆圈中最后剩下的数字
   1. https://leetcode-cn.com/problems/yuan-quan-zhong-zui-hou-sheng-xia-de-shu-zi-lcof/solution/huan-ge-jiao-du-ju-li-jie-jue-yue-se-fu-huan-by-as/

### 关键思路

约瑟夫环。核心：只关心最终活着那个人的序号变化。

下面两张图胜过千言万语。第一张展示了正向的算法流程，n=8，m=3。



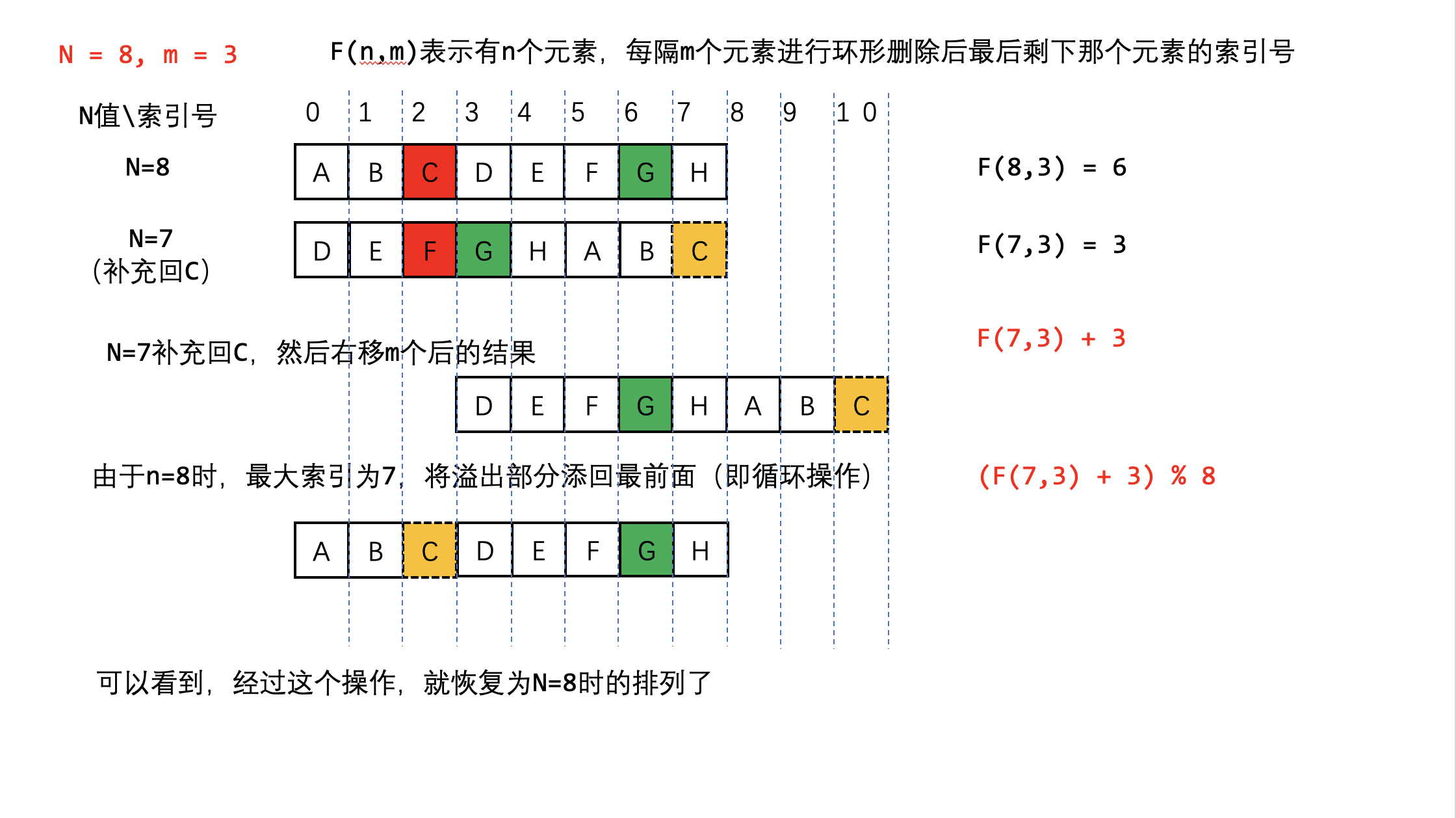
正向的算法流程

假如最后剩下的数字为G，我们关注的是，其在数组中的索引变化情况。如上图所示，易知最后一次（n=1）G的索引必然为0，我们想知道的是，其在第一次（n=8）的索引。

下图展示了如何从n=7的情况变换到n=8的情况，实际上也就建立了从n=i-1到n=i的递推关系（证明从略）：

其中， 表示最终被删掉的那个数在当前数组中的索引，当前数组的元素个数为i。

i的范围为1~n，当i=1时，f(i)=0，也即数组中只剩下了那个元素。



如何从f(i-1) 到f(i)

### 复杂度

1. 时间复杂度：O(n)。
2. 空间复杂度：O(1)。

### 代码

| **JZ-62 圆圈中最后剩下的数字.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148507661/ |
| 1. class Solution { 2. public: 3. int lastRemaining(int n, int m) { 4. int pos = 0; 5. for (int i=2; i<=n; ++i) { 6. pos = (pos + m) % i; 7. } 8. return pos; 9. } 10. }; |

## 美团 03 拜访

链接：https://www.nowcoder.com/practice/12cbdcdf5d1e4059b6ddd420de6342b6?tpId=128&tags=&title=&diffculty=0&judgeStatus=0&rp=1&tab=answerKey

标签：动态规划

### 精选题解

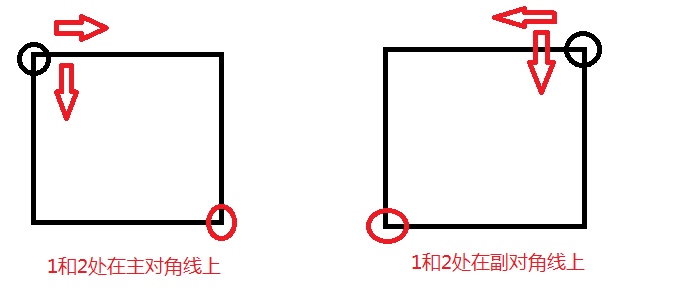
### 关键思路

“只能在左右中选择一个方向，在上下中选择一个方向”应该理解为：

* 左右中只能选一个方向，若选择左只能一直向左走；上下中只能选择一个方向，若选择下只能一直向下。

解题思路：

1. 首先找到1和2的位置
   * 注意到从1走到2与从2走到1所得的路径数相同，即以1为起点或以2为起点是等价的。
   * 所以可以这样处理：**统一从行坐标小的位置走到行坐标大的位置，即向下走。**
2. 1和2的相对位置可以归纳如下：
   * ① 两者位于主对角线上
   * ② 两者位于副对角线上
   * ③ 两者位置重合或处于同一行或同一列（该特殊情形可以合并到 ① ② 中）



1. 之后分别对向左走和向右走的情形应用动态规划求解
   * 二维dp数组，需要初始化首行和首列

### 复杂度

时间复杂度：O()。其中n为map的行数，m为列数。

空间复杂度：O()。

### 代码

| **MT-03 拜访.cpp** |
| --- |
|  |
| 1. class Visit { 2. public: 3. int countPath(vector<vector<int> > map, int n, int m) { 4. *// find 1 and 2* 5. int x1, y1, x2, y2; 6. for (int i=0; i<n; ++i) { 7. for (int j=0; j<m; ++j) { 8. if (map[i][j]==1) { 9. x1 = i; 10. y1 = j; 11. } else if (map[i][j]==2) { 12. x2 = i; 13. y2 = j; 14. } 15. } 16. } 17. *// two man at same position* 18. if (x1==x2 && y1==y2) 19. return 1; 20. *// set x1 as who has smaller row number* 21. if (x1>x2) { 22. swap(x1, x2); 23. swap(y1, y2); 24. } 25. *// use dp* 26. vector<vector<int>> dp(n, vector<int>(m,0)); 27. *// initialize firt column* 28. dp[x1][y1] = 1; 29. for (int i=x1+1; i<=x2; ++i) *// top to bottom* 30. dp[i][y1] = (map[i][y1]==-1) ? 0 : dp[i-1][y1]; 31. *// bi-direction* 32. if (y1<y2) { *// left to right* 33. for (int j=y1+1; j<=y2; ++j) *// left to right* 34. dp[x1][j] = (map[x1][j]==-1) ? 0 : dp[x1][j-1]; 35. for (int i=x1+1; i<=x2; ++i) { *// top-left to bottom-right* 36. for (int j=y1+1; j<=y2; ++j) { 37. dp[i][j] = (map[i][j]==-1) ? 0 : (dp[i-1][j] + dp[i][j-1]); 38. } 39. } 40. } else { *// right to left* 41. for (int j=y1-1; j>=y2; --j) 42. dp[x1][j] = (map[x1][j]==-1) ? 0 : dp[x1][j+1]; 43. for (int i=x1+1; i<=x2; ++i) { 44. for (int j=y1-1; j>=y2; --j) { 45. dp[i][j] = (map[i][j]==-1) ? 0 : (dp[i-1][j] + dp[i][j+1]); 46. } 47. } 48. } 49. return dp[x2][y2]; 50. } 51. }; |

## 美团 05 字符串计数

链接：https://www.nowcoder.com/practice/f72adfe389b84da7a4986bde2a886ec3?tpId=128&tqId=33772&rp=1 &tab=answerKey

标签：动态规划

### 精选题解

### 关键思路

动态规划。dp[i] ：满足字典序s1 < s < s2且长度为i的字符串s的个数。

以s1=ab, s2= ce, len1=1, len2=2为例。

* dp[1]：字符串长度为1，则仅有b和c满足要求，也即dp[1]=2
  + 对于任意的s1[0] 和s2[0]，事实上长度为1的字符串的个数均为s2[0] - s1[0]
  + 实际上，上一步可以拆分为先减去 (s1[0]-‘a’)，再加上(s2[0]+’a’)
* dp[2]：字符串长度为2，满足要求的有ac~az (24)、ba~bz (26)、ca~cd (4)，也即dp[2]=54
  + 事实上，如果不考虑s1[1] 和 s2[1] 的限制的话，那么满足要求的就是所有首字符在s1[0] 和s2[0] 之间（此时包含s1[0] 而不包含s1[0]）且长度为2的字符串（aa~az、ba~bz），也即对于每个首字符，在后面添加a~z，事实上有dp[2] = 26\*dp[1]
  + 然而我们要去掉 (s1[1]-‘a’+1) =2个（aa~ab），并加上 (s2[1]-’a’+1) =5个（ca~ce）
  + 同时，由于ce的长度也为2，则我们需要将ce去掉

### 复杂度

### 代码

# 排序

## 剑指 45 把数组排成最小的数

链接：https://leetcode-cn.com/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/

标签：排序，字符串

### 精选题解

* C++ 先转换成字符串再组合 - 把数组排成最小的数
  + <https://leetcode-cn.com/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/solution/c-xian-zhuan-huan-cheng-zi-fu-chuan-zai-zu-he-by-y/>

### 关键思路

把数字转成字符串，依照字符串的字典序排序。

以三个数字为例：3、5、30，其字符串的排序应为“30”<“3”<“5”。

如何正确排序判断“30”和“3”这种呢？

方法是向sort传入自定义的比较函数。

* 注意到“303”<“330”，也即“30”+“3”<“3”+“30”
* 故使用lambda函数传入sort，返回s1+s2 < s2+s1的符号（代码第12行）
  + sort中比较函数的含义是，若返回值为true，则s1<s2

### 复杂度

* 时间复杂度：O(nlogn)。其中n为总的字符数，sort内部使用快排实现，平均时间复杂度为O(nlogn)。
* 空间复杂度：O(n)。字符串vector和字符串string。

### 代码

| **JZ-45 把数组排成最小的数.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/142628451/ |
| 1. class Solution { 2. public: 3. string minNumber(vector<int>& nums) { 4. vector<string> vs; 5. string res; 6. for (auto const &n: nums) { 7. vs.push\_back(to\_string(n)); 8. } 9. sort(vs.begin(), vs.end(), 10. [](string& s1, string& s2) { 11. return s1+s2 < s2+s1; *// if condition is true, then s1 < s2* 12. } 13. ); 14. for (auto const &s: vs) { 15. res += s; 16. } 17. return res; 18. } 19. }; |

## 剑指 51 数组中的逆序对

链接：https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/

标签：归并排序，分治，递归

### 精选题解

1. 官方题解 - 数组中的逆序对
   1. <https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/shu-zu-zhong-de-ni-xu-dui-by-leetcode-solution/>
2. 暴力解法、分治思想、树状数组 - 数组中的逆序对
   1. https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/bao-li-jie-fa-fen-zhi-si-xiang-shu-zhuang-shu-zu-b/
3. 【算法】排序算法之归并排序 - 知乎
   1. <https://zhuanlan.zhihu.com/p/124356219>
4. 归并排序 - 维基百科，自由的百科全书
   1. <https://zh.wikipedia.org/wiki/归并排序>

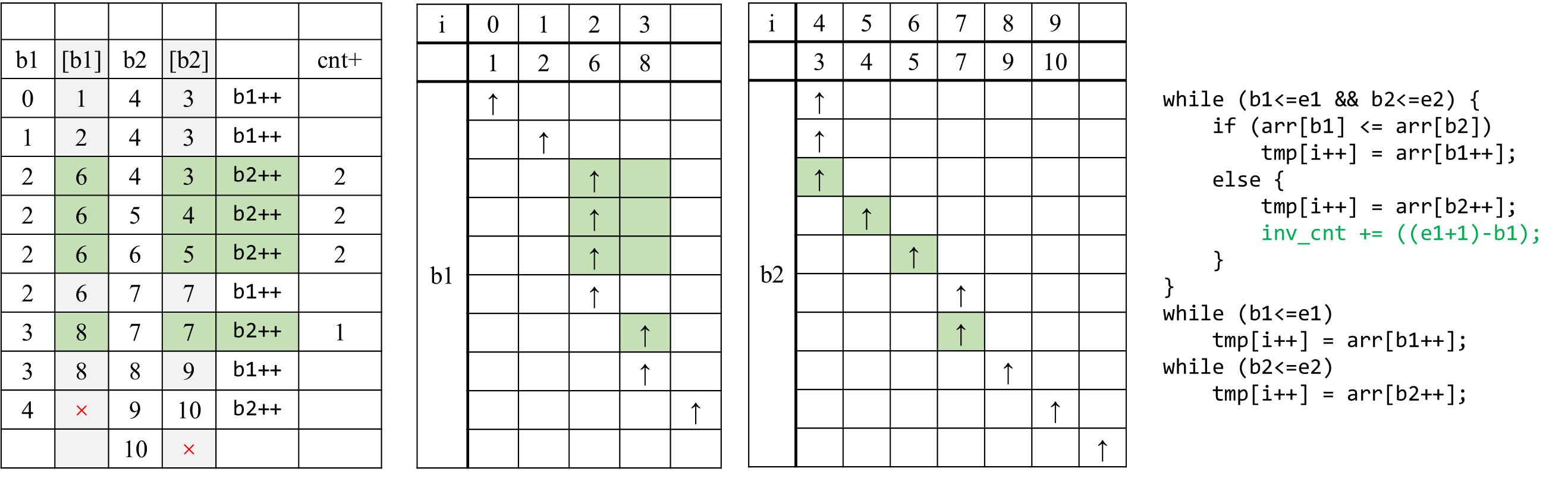
### 关键思路

这题理解起来略复杂，需要熟悉归并排序的思路，参考上面的链接，同时还需要挑个具体的例子画一画，这样才能理解得更透彻。

归并排序事实上分成三步：

1. **分割：**分成左右两个子数组
2. **解决：**递归地归并排序子数组
3. **合并：**将排序好的两个子数组按照正确的顺序合并

本题妙用归并排序的地方在第三步，因为只有在合并时才会出现对逆序的处理。



那么怎样才能计入这些逆序对呢？有两种方法，第一种是在arr[b1] <= arr[b2] 的时候计数，另一种是在arr[b1] > arr[b2] 的时候计数。第二种方法更容易理解，简述如下：

1. 先移动b1，碰到逆序后，再移动b2
   1. 采取第二种方法有个好处，就是移动b1和b2的先后顺序不会对计数造成影响
2. 每当出现逆序，也即arr[b1] > arr[b2] 时，就统计左边数组中比arr[b2] 大的数的个数，该个数很好计算，也就是左边数组从索引b1往后的所有数的个数，也即 (e1+1)-b1。
   1. 注意arr[b1] <= arr[b2]，必须是 <=，不能是 <，因为其补集必须为 arr[b1] > arr[b2]
   2. 事实上，在代码第12 ~ 18行中先处理arr[b1] > arr[b2] 的情况也是正确的，其实更推荐这种写法，因为可以避免上一行中提到的 <= 和 < 的问题。

### 复杂度

1. 时间复杂度：O(nlogn)。归并排序的时间复杂度。计算起来略复杂。
   1. 总时间的递归式为T(n)=2T(n/2)+O(n)，可用递归树解得O(nlogn)。

* 参考：https://blog.csdn.net/touch\_2011/article/details/6785881
  1. 假设最后的子问题用时为常数c，对于n个待排序子问题来说，最后分解为n个时间为c的子问题，则整个问题的规模为cn。
  2. 第一层时间代价为cn，第二层时间代价为cn/2+cn/2=cn，……可知每层代价都是cn，共有 (logn+1) 层。所以总的时间代价为cn\*(logn+1)，也即O(nlogn)。



* 1. 在最坏、最佳、平均情况下归并排序时间复杂度均为O(nlogn)。
  2. 从合并过程中可以看出合并排序稳定。

1. 空间复杂度：O(n)。辅助数组tmp。

### 代码

| **JZ-51 数组中的逆序对.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143187625/ |
| 1. class Solution { 2. public: 3. int merge\_sort\_sub(vector<int>& arr, vector<int>&tmp, int b, int e) { 4. if (b>=e) return 0; 5. *// 分治并排序* 6. int b1 = b, e1 = b + (e-b)/2; 7. int b2 = e1+1, e2 = e; 8. int inv\_cnt = merge\_sort\_sub(arr, tmp, b1, e1) + merge\_sort\_sub(arr, tmp, b2, e2); 9. *// 归并* 10. int i = b; 11. while (b1<=e1 && b2<=e2) { 12. if (arr[b1] <= arr[b2]) 13. *// 必须用 <= 而非 <，因为要计入逆序必须满足 arr[b1] > arr[b2-1]，故取补集* 14. tmp[i++] = arr[b1++]; 15. else { 16. tmp[i++] = arr[b2++]; 17. inv\_cnt += ((e1+1)-b1); *// 计入逆序对数* 18. } 19. } 20. while (b1<=e1) 21. tmp[i++] = arr[b1++]; 22. while (b2<=e2) 23. tmp[i++] = arr[b2++]; 24. copy(tmp.begin()+b, tmp.begin()+e+1, arr.begin()+b); 25. return inv\_cnt; 26. } 27. int merge\_sort(vector<int>& arr) { 28. vector<int> tmp(arr.size(), 0); 29. return merge\_sort\_sub(arr, tmp, 0, arr.size()-1); 30. } 31. int reversePairs(vector<int>& nums) { 32. return merge\_sort(nums); 33. } 34. }; |

# 位运算

## 0136 只出现一次的数字

链接：https://leetcode-cn.com/problems/single-number/

标签：位运算

### 精选题解

略。

### 关键思路

一个数异或两次得0，要找到只出现一次的数，只需将所有数异或一遍即可。

### 复杂度

1. 时间复杂度：O(n)。遍历数组所有数
2. 空间复杂度：O(1)。

### 代码

| **0136 只出现一次的数字.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143271854/ |
| 1. class Solution { 2. public: 3. int singleNumber(vector<int>& nums) { 4. int res = 0; 5. for (auto const& ele: nums) 6. res ^= ele; 7. return res; 8. } 9. }; |

## 0137 只出现一次的数字 II + 剑指 56-2 数组中数字出现的次数 II

链接：<https://leetcode-cn.com/problems/single-number-ii/>

https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-ii-lcof/

标签：位运算

### 精选题解

1. 哈希表和位运算两种解法 - 数组中数字出现的次数 II
   1. https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-ii-lcof/solution/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-ii-ha-xi-bi/

### 关键思路

在C++中，每个整数有32位，对每一位进行计数

1. 若nums中某数n的第i位为1，也即 n & (1<<i) 为true，则将计数器cnt++
2. 若cnt%3==1，表明只出现一次的那个数在第i位为1，将其写入结果，也即res |= (1<<i)

此方法对出现任意次的情形都适用。

### 复杂度

1. 时间复杂度：O(n)。遍历数组所有元素。
2. 空间复杂度：O(1)。

### 代码

| **0137 只出现一次的数字 II + JZ-56-2 数组中出现的次数 II.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143466762/ |
| 1. class Solution { 2. public: 3. int singleNumber(vector<int>& nums) { 4. int res = 0; 5. for (int i=0; i<32; ++i) { 6. int cnt=0; 7. for (auto const &n: nums) *// count number of 1 at i-th bit* 8. if (n & (1<<i)) *// i-th bit of n is 1* 9. ++cnt; 10. if (cnt % 3 == 1) *// set i-th bit to 1* 11. res |= (1<<i); 12. } 13. return res; 14. } 15. }; |

## 剑指 56-1 数组中数字出现的次数

链接：https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-lcof/

标签：位运算

### 精选题解

* 官方题解 - 数组中数字出现的次数
  + https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-lcof/solution/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-by-leetcode/

### 关键思路

设两个仅出现一次的数为a和b，对所有数异或一次，得到的值为a^b。

核心问题是如何将a和b分开。方法是对数组中的数进行分组：

* 既然a和b不相等，那么a^b肯定非0。
* a^b的二进制表示中，为1的位表明二者不同。
* 只需要找到a^b中任意为1的位，以该位是否为1来分割数组，一定能将a和b分开
* 对于其他的数，都出现了两次，因此同一个数必然分到同一组，那么独立地对两个组中的数进行异或操作，依旧能将相同的数消去，这样就分别得到了a和b

### 复杂度

* 时间复杂度：O(n)。两次遍历操作。
* 空间复杂度：O(1)。

### 代码

| **JZ-56-1 数组中数字出现的次数.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/143523927/ |
| 1. class Solution { 2. public: 3. vector<int> singleNumbers(vector<int>& nums) { 4. int tmp = 0; 5. for (auto n: nums) 6. tmp ^= n; 7. int sep = 1; 8. while ((sep & tmp) == 0) 9. sep <<= 1; 10. int a = 0, b = 0; 11. for (auto n: nums) 12. if (sep & n) 13. a ^= n; 14. else 15. b ^= n; 16. return vector<int>{a, b}; 17. } 18. }; |

# 集合、哈希表

## 0001 两数之和

链接：https://leetcode-cn.com/problems/two-sum/

标签：哈希表

### 精选题解

* 官方题解 - 两数之和
  + https://leetcode-cn.com/problems/two-sum/solution/liang-shu-zhi-he-by-leetcode-solution/

### 关键思路

哈希表。查找target-nums[i] 是否在map中。map<int, int>的键为元素的值，值为在数组中的索引。

### 代码

| **0001 两数之和.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/152203318/ |
| 1. class Solution { 2. public: 3. vector<int> twoSum(vector<int>& nums, int target) { 4. int n = nums.size(); 5. if (n<=1) 6. return {}; 7. unordered\_map<int, int> map; 8. for (int i=0; i<n; ++i) { 9. auto itr = map.find(target-nums[i]); 10. if (itr != map.end()) 11. return {itr->second, i}; 12. else 13. map[nums[i]] = i; 14. } 15. return {}; 16. } 17. }; |

## 剑指 61 扑克牌中的顺子

链接：https://leetcode-cn.com/problems/bu-ke-pai-zhong-de-shun-zi-lcof/

标签：数组，排序，集合

### 精选题解

* 面试题61. 扑克牌中的顺子（集合Set / 排序，清晰图解） - 扑克牌中的顺子
  + <https://leetcode-cn.com/problems/bu-ke-pai-zhong-de-shun-zi-lcof/solution/mian-shi-ti-61-bu-ke-pai-zhong-de-shun-zi-ji-he-se/>
* set - C++ Reference
  + https://www.cplusplus.com/reference/set/set/

### 关键思路

5张牌是顺子的充要条件：

* 除大小王外，所有牌无重复
* 除大小王外，设最大的牌为max，最小的牌为min，须
  + 若5个数中没有大小王，则无重复已经保证了，而只有连续的5个数才能满足这一要求，故必然是顺子
  + 若5个数中有大小王，由于大小王可以充当任意数，故必然能填充其他数构成的“空隙”形成顺子

既然是充要条件，也即满足上述条件必然是顺子，不满足的则不是。

有两种思路：

* 先排序，再遍历，根据是否存在相同的相邻数，判断有无重复；最大最小值也即首尾两个数
* 直接遍历，使用集合来判断有无重复；使用两个变量来记录最大最小值
  + 事实上，C++中的set使用红黑树实现，因此自动排序，所以全部插入后，直接就能得到最大值（\*s.rbegin()）和最小值（\*s.begin()）
  + 为了不让大小王影响集合的情况，碰到元素值为0的，直接continue

两个思路都很好写，这里为了熟悉集合的用法，采用第二种方法。

### 复杂度

排序：

* 时间复杂度：O(nlogn)=O(5log5)=O(1)。sort使用快排，复杂度为O(nlogn)。
* 空间复杂度：O(1)。

集合：

* 时间复杂度：O(n)=O(5)=O(1)。一遍扫描。
* 空间复杂度：O(n)=O(5)=O(1)。集合的空间为O(n)；两个记录最大最小值的中间变量。

### 代码

| **JZ-61 扑克牌中的顺子.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148524009/ |
| 1. class Solution { 2. public: 3. bool isStraight(vector<int>& nums) { 4. set<int> s; 5. for (auto const & ele: nums) { 6. if (ele==0) 7. continue; 8. if (s.count(ele)) 9. return false; 10. s.insert(ele); 11. } 12. if (\*s.rbegin()-\*s.begin()>=5) 13. return false; 14. return true; 15. } 16. }; |

# 其他

## 剑指 59-2 左旋转字符串

链接：https://leetcode-cn.com/problems/zuo-xuan-zhuan-zi-fu-chuan-lcof/

标签：字符串

### 精选题解

1. 一行击败 100%，使用 C++ 标准库 rotate（附带底层实现详解） - 左旋转字符串
   1. https://leetcode-cn.com/problems/zuo-xuan-zhuan-zi-fu-chuan-lcof/solution/shi-yong-c-biao-zhun-ku-rotatefu-dai-di-lf1e9/

### 关键思路

详见我在“[后台面经整理.docx](https://github.com/Hansimov/cs-interviews)”中的“C++代码实现”“rotate的实现”。

### 复杂度

1. 时间复杂度：O(n)。
2. 空间复杂度：O(1)。

### 代码

| **JZ-58-2 左旋转字符串.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/148034194/ |
| 1. class Solution { 2. public: 3. string reverseLeftWords(string s, int n) { 4. rotate(s.begin(), s.begin()+n, s.end()); 5. return s; 6. } 7. }; |