# 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/>

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/>

### 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. }; |

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

标签：深搜

### 精选题解

* 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

同“0079 单词搜索”。

### 复杂度

### 代码

## 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. }; |

# 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

1. (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：

1. 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）。
2. 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：

1. 落到1。这种情况为快乐数。



1. 进入循环，永远不会到1。



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. }; |

## 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

## 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. }; |

## 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”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. /\*\* 2. \* Definition for singly-linked list. 3. \* struct ListNode { 4. \* int val; 5. \* ListNode \*next; 6. \* ListNode(int x) : val(x), next(NULL) {} 7. \* }; 8. \*/ 9. **class** **Solution** { 10. **public**: 11. ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) { 12. stack<int> s1, s2; 13. **while** (l1) { 14. s1.push(l1->val); 15. l1 = l1->next; 16. } 17. **while** (l2) { 18. s2.push(l2->val); 19. l2 = l2->next; 20. } 21. ListNode \*head = **nullptr**; 22. int sum = 0; 23. int carry = 0; 24. **while** (!s1.empty() || !s2.empty() || carry>0) { 25. sum = 0; 26. **if** (!s1.empty()) { 27. sum += s1.top(); 28. s1.pop(); 29. } 30. **if** (!s2.empty()) { 31. sum += s2.top(); 32. s2.pop(); 33. } 34. sum += carry; 35. carry = sum / 10; 36. **auto** tmp = **new** ListNode(sum%10); 37. tmp->next = head; 38. head = tmp; 39. } 40. **return** head; 41. } 42. }; |

# 树

## 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

* 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）
  + <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/>
* 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
  + <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：

* 树的一个子树对应着数组中一段连续的子数组
* 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
* 后序遍历汇总，子数组最后一个元素对应子树的根结点
* 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**

* 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。
  + 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
* 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
  + 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
  + 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
* 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
* 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）

* 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
  + 前序遍历数组：左→右→根
  + 后序遍历数组：根→左→右
* 所以为了能用到前序构造的方法，将后序遍历数组反转
  + 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：

* + 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）
  + 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
  + 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：

* + 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）
  + 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
  + 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
  + 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

* 首先顾名思义，单调栈肯定是单调的
* 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
* 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
* 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）
  + 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
  + 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
* 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：**

* 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。
* 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：**

* 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。
* 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

|  |
| --- |
| 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. }; |

## 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
  + <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
  + 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：

* 初始时idx=0，(left, right)=(INT\_MIN, INT\_MAX)
* 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
* 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）
  + 为了写法简便，将idx声明为public变量
  + 注意idx++ 的简洁写法
* 接着调用递归，生成root的左右子结点（代码1第8 ~ 9行）
  + 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
  + 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。

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

该思路如下：

* 初始化：以数组第一个元素生成根结点root，并入栈
* 顺序遍历数组的每个元素
  + 以当前元素生成结点cur，将栈顶结点赋给top
  + 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
  + 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
  + 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

* 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
* 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. }; |

# 双指针

## 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
  + 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--
  + ++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。
  + 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
  + 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
  + 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. }; |

# 栈

## 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

|  |
| --- |
| 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

## 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
  + <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
  + <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. }; |

## 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. }; |

## 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

* 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
  + <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/>
* 官方题解 - 把数字翻译成字符串
  + <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/>
  + <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：

* 对应的数字在10 ~ 25之间，也即 s[i-1]==’1’ || (s[i-1]==’2’ && s[i] <=5)
* 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：

* dp[i+1] 表示以字符 s[i] 结尾的字符串的翻译方法总数
* 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

* 时间复杂度：O(n)。数字转换成字符串；遍历字符串。
* 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. }; |

# 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

* 官方题解 - 数组中的逆序对
  + <https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/shu-zu-zhong-de-ni-xu-dui-by-leetcode-solution/>
* 暴力解法、分治思想、树状数组 - 数组中的逆序对
  + 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/
* 【算法】排序算法之归并排序 - 知乎
  + <https://zhuanlan.zhihu.com/p/124356219>
* 归并排序 - 维基百科，自由的百科全书
  + <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：

* **分割：**分成左右两个子数组
* **解决：**递归地归并排序子数组
* **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：

* 先移动b1，碰到逆序后，再移动b2
  + 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
* 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。
  + 注意arr[b1] <= arr[b2]，必须是 <=，不能是 <，因为其补集必须为 arr[b1] > arr[b2]
  + 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

* 时间复杂度：O(nlogn)。归并排序的时间复杂度。计算起来略复杂。
  + 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
    - 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。
    - 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)。



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