# 回溯、DFS

## 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. /\* 2. \* 【46】C++ 回溯法/交换法/stl 简洁易懂的全排列 3. \* https://leetcode-cn.com/problems/permutations/solution/c-hui-su-fa-jiao-huan-fa-stl-jian-ji-yi-dong-by-sm/ 4. \*/ 5. class Solution { 6. public: 7. vector<vector<int>> res; 8. void backtrack(vector<int> &nums, vector<int> &current, vector<bool> &flags) { 9. **if** (current.size() == flags.size()) { 10. res.push\_back(current); 11. } **else** { 12. **for** (int i=0; i<nums.size(); ++i) { 13. **if** (not flags[i]) { // nums[i] not in current 14. current.push\_back(nums[i]); 15. flags[i] = true; 16. backtrack(nums, current, flags); 17. current.pop\_back(); 18. flags[i] = false; 19. } 20. } 21. } 22. } 23. vector<vector<int>> permute(vector<int>& nums) { 24. **if** (nums.empty()) { 25. **return** {}; 26. } 27. vector<bool> flags(nums.size(), false); // true: in current; false: not in current 28. vector<int> current; 29. backtrack(nums, current, flags); 30. **return** res; 31. } 32. }; |

### 代码2：交换元素

| **0046 全排列 - swap.cpp** |
| --- |
| https://leetcode-cn.com/submissions/detail/127482585/ |
| 1. /\* 2. \* 【46】C++ 回溯法/交换法/stl 简洁易懂的全排列 - 全排列 - 力扣（LeetCode） 3. \* https://leetcode-cn.com/problems/permutations/solution/c-hui-su-fa-jiao-huan-fa-stl-jian-ji-yi-dong-by-sm/ 4. \* 全排列 - 全排列 - 力扣（LeetCode） 5. \* https://leetcode-cn.com/problems/permutations/solution/quan-pai-lie-by-leetcode-solution-2/ 6. \*/ 7. class Solution { 8. public: 9. vector<vector<int>> res; 10. void backtrack(vector<int> &nums, int start, int end) { 11. **if** (start == end) { 12. res.push\_back(nums); 13. } **else** { 14. **for** (int i=start; i<=end; ++i) { 15. swap(nums[i], nums[start]); 16. backtrack(nums, start+1, end); 17. swap(nums[i], nums[start]); 18. } 19. } 20. } 21. vector<vector<int>> permute(vector<int>& nums) { 22. **if** (nums.empty()) { 23. **return** {}; 24. } **else** { 25. backtrack(nums, 0, nums.size()-1); 26. **return** res; 27. } 28. } 29. }; |

## 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. /\* 2. \* 全排列 II - 全排列 II - 力扣（LeetCode） 3. \* https://leetcode-cn.com/problems/permutations-ii/solution/quan-pai-lie-ii-by-leetcode-solution/ 4. \*/ 5. class Solution { 6. vector<int> visited; 7. public: 8. void backtrack(vector<int> &nums, vector<vector<int>> &res, int idx, vector<int> &current) { 9. **if** (idx==nums.size()) { 10. res.emplace\_back(current); 11. // \* C++ STL vector 添加元素（push\_back()和emplace\_back()）详解 12. // \* http://c.biancheng.net/view/6826.html 13. // 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）； 14. // 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。 15. **return** ; 16. } 17. **for** (int i=0; i<nums.size(); ++i) { 18. // 哪些情况不取当前的元素： 19. // 1. 已经访问过/在当前路径数组中 20. // 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） 21. // 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， 22. // 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 23. // 此时意味着我们在当前路径数组中存在多个相等的数了 24. **if** (visited[i] || (i>0 && nums[i]==nums[i-1] && !visited[i-1])) 25. **continue**; 26. current.emplace\_back(nums[i]); 27. visited[i] = true; 28. backtrack(nums, res, idx+1, current); 29. visited[i] = false; 30. current.pop\_back(); 31. } 32. } 33. vector<vector<int>> permuteUnique(vector<int>& nums) { 34. vector<vector<int>> res; 35. vector<int> current; 36. visited.resize(nums.size()); 37. sort(nums.begin(), nums.end()); 38. backtrack(nums, res, 0, current); 39. **return** res; 40. } 41. }; |

## 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”(p5)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

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. bool tmp\_flag = false; 12. for (const auto& d: directions) { 13. int i\_new = i + d.first; 14. int j\_new = j + d.second; 15. if (i\_new>=0 && i\_new<board.size() && j\_new>=0 && j\_new<board[0].size()) { 16. if (visited[i\_new][j\_new]) 17. continue; 18. tmp\_flag = check(i\_new, j\_new, k+1, board, visited, word); 19. if (tmp\_flag) { 20. visited[i][j] = false; 21. return true; 22. } 23. } 24. } 25. visited[i][j] = false; 26. return false; 27. } 28. bool exist(vector<vector<char>>& board, string word) { 29. int row\_num = board.size(); 30. if (row\_num<=0) 31. return false; 32. int col\_num = board[0].size(); 33. bool flag = false; 34. vector<vector<bool>> visited(row\_num, vector<bool>(col\_num)); 35. for (int i=0; i<row\_num; ++i) { 36. for (int j=0; j<col\_num; ++j) { 37. flag = check(i, j, 0, board, visited, word); 38. if (flag) 39. return true; 40. } 41. } 42. return false; 43. } 44. }; |

## 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. }; |