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：$O(3^k \times 4^{n-k})$。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
7       void dfs(int digit_idx, string& digits) {
8           if (digit_idx == digits.size()) {
9               res.push_back(temp);
10              return ;
11          }
```

0017 电话号码的字母组合.cpp

```
12
13          int board_idx = digits[digit_idx] - '0';
14          for (int i=0; i<board[board_idx].size(); ++i) {
15              temp.push_back(board[board_idx][i]);
16              dfs(digit_idx+1, digits);
17              temp.pop_back();
18          }
19      }
20
21      vector<string> letterCombinations(string digits) {
22          if (digits.size() == 0)
23              return res;
24          dfs(0, digits);
25          return res;
26      }
27  };
28
```

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/

0022 括号生成.cpp

```cpp
class Solution {
public:
    vector<string> res;
    string temp;

    void dfs(int n, int left, int right) {
        if (temp.size() == 2*n) {
            res.push_back(temp);
            return ;
        }

        if (left < n) {
            temp.push_back('(');
            dfs(n, left+1, right);
            temp.pop_back();
        }
        if (right < left) {
            temp.push_back(')');
            dfs(n, left, right+1);
            temp.pop_back();
        }
    }

    vector<string> generateParenthesis(int n) {
        dfs(n, 0, 0);
        return res;
    }
};

```

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。

```
if (idx==candidates.size())
    return ;
if (target==0) {
    res.push_back(temp);
    return ;
}

// This part must be after the previous part
if (target - candidates[idx] < 0) {
    return ;
}
```

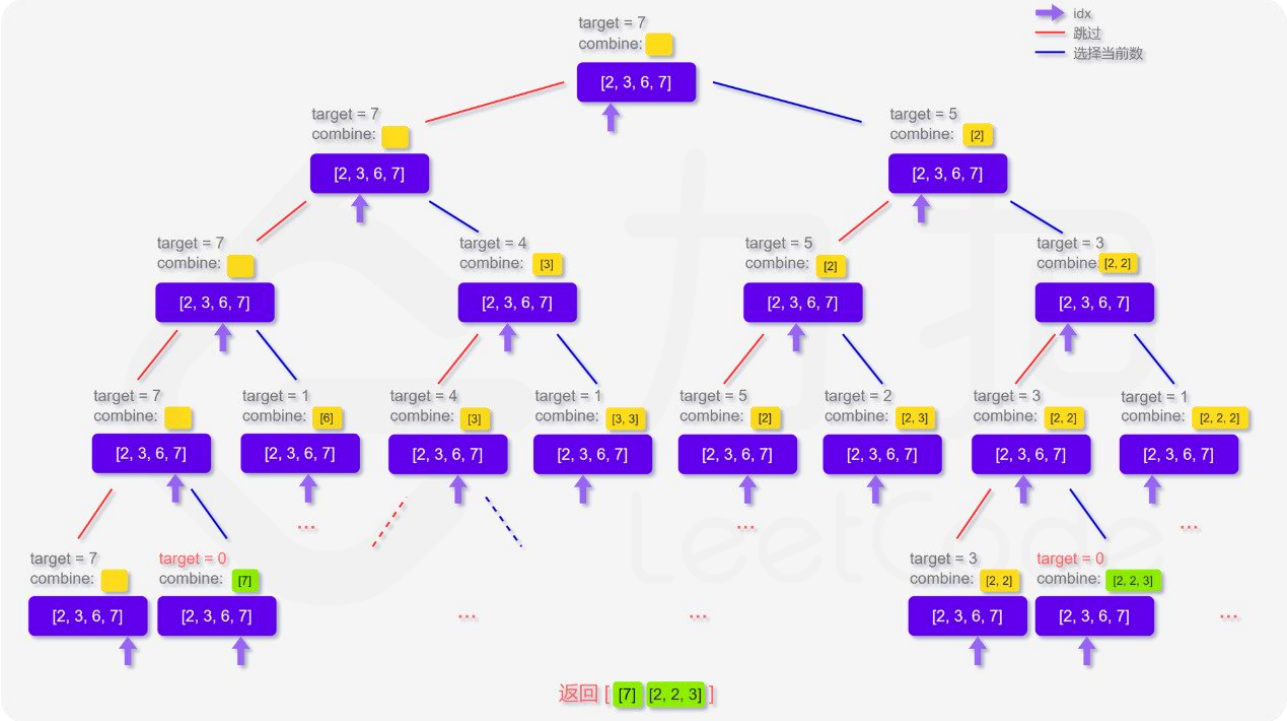
无非是取或不取当前数。区别有两点：

（1）是否出栈入栈（棕色部分）；

（2）dfs 的参数变化（红色部分）。

```
// do not choose candidates[idx]
dfs(candidates, target, idx+1);

// choose candidates[idx]
temp.push_back(candidates[idx]);
dfs(candidates, target-candidates[idx], idx);
temp.pop_back();
```



复杂度

- 时间复杂度：O($n \times 2^n$)。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
6       void dfs(vector<int> & candidates, int target, int idx) {
7           if (idx==candidates.size())
8               return ;
9           if (target==0) {
10              res.push_back(temp);
11              return ;
12          }
13
14          // This part must be after the previous part
```

0039 组合总和.cpp

```
15          if (target - candidates[idx] < 0) {
16              return ;
17          }
18
19          // do not choose candidates[idx]
20          dfs(candidates, target, idx+1);
21
22          // choose candidates[idx]
23          temp.push_back(candidates[idx]);
24          dfs(candidates, target-candidates[idx], idx);
25          temp.pop_back();
26      }
27
28      vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
29          sort(candidates.begin(), candidates.end());
30          dfs(candidates, target, 0);
31          return res;
32      }
33  };
34
```

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($2^n \times n$)。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(2^n)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
6       void dfs(vector<int>& candidates, int target, int idx) {
7           if (target==0) {
8               res.push_back(temp);
9               return ;
10          }
11
```

0040 组合总和 II.cpp

```
12          for (int i=idx; i<candidates.size(); ++i) {
13              if (target - candidates[idx] < 0)
14                  break;
15              if (i>idx && candidates[i] == candidates[i-1])
16                  continue;
17              temp.push_back(candidates[i]);
18              dfs(candidates, target-candidates[i], i+1);
19              temp.pop_back();
20          }
21      }
22
23      vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
24          sort(candidates.begin(), candidates.end());
25          dfs(candidates, target, 0);
26          return res;
27      }
28  };
29
```

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：$O(C_9^k \times k)$。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/

```cpp
class Solution {
public:
    vector<vector<int>> res;
    vector<int> temp;

    void dfs(int k, int target, int cur) {
        if ( (temp.size() + (9-cur+1) < k) || temp.size() > k)
            return ;
        if (temp.size() == k && target == 0) {
            res.push_back(temp);
            return ;
        }

        // not choose cur
        dfs(k, target, cur+1);

        // choose cur
        temp.push_back(cur);
        dfs(k, target-cur, cur+1);
        temp.pop_back();
    }

    vector<vector<int>> combinationSum3(int k, int n) {
        dfs(k, n, 1);
        return res;
    }
};

```

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/

```cpp
class Solution {
public:
    vector<vector<int>> res;

    void backtrack(vector<int> &nums, vector<int> &current, vector<bool> &flags) {
        if (current.size() == flags.size()) {
            res.push_back(current);
        } else {
            for (int i=0; i<nums.size(); ++i) {
                if (not flags[i]) { // nums[i] not in current
                    current.push_back(nums[i]);
                    flags[i] = true;
                    backtrack(nums, current, flags);
                    current.pop_back();
                    flags[i] = false;
                }
            }
        }
    }

```

0046 全排列.cpp

```
21      vector<vector<int>> permute(vector<int>& nums) {
22          if (nums.empty()) {
23              return {};
24          }
25          vector<bool> flags(nums.size(), false); // true: in current; false:
    not in current
26          vector<int> current;
27          backtrack(nums, current, flags);
28          return res;
29      }
30  };
31
```

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
5       void backtrack(vector<int> &nums, int start, int end) {
6           if (start == end) {
7               res.push_back(nums);
8           } else {
9               for (int i=start; i<=end; ++i) {
10                  swap(nums[i], nums[start]);
11                  backtrack(nums, start+1, end);
12                  swap(nums[i], nums[start]);
13              }
14          }
15      }
16
17      vector<vector<int>> permute(vector<int>& nums) {
18          if (nums.empty()) {
19              return {};
20          } else {
21              backtrack(nums, 0, nums.size()-1);
22              return res;
23          }
24      }
25  };
```

0046 全排列 - swap.cpp
26

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;
```

0047 全排列 II.cpp

```
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
14          for (int i=0; i<nums.size(); ++i) {
15              // 哪些情况不取当前的元素：
16              // 1. 已经访问过/在当前路径数组中
17              // 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）
18              //    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，
19              //    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
20              //    此时意味着我们在当前路径数组中存在多个相等的数了
21              if (visited[i] || (i>0 && nums[i]==nums[i-1] && !visited[i-1]))
22                  continue;
23              current.emplace_back(nums[i]);
24              visited[i] = true;
25              backtrack(nums, res, idx+1, current);
26              visited[i] = false;
27              current.pop_back();
28          }
29      }
30
31      vector<vector<int>> permuteUnique(vector<int>& nums) {
32          vector<vector<int>> res;
33          vector<int> current;
34          visited.resize(nums.size());
35          sort(nums.begin(), nums.end());
36          backtrack(nums, res, 0, current);
37          return res;
38      }
39  };
```

0047 全排列 II.cpp
40

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
6   // do not choose cur
7   dfs(cur+1, n, k);
```

复杂度

- 时间复杂度：O($C_n^k \times k$)。其中C_n^k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/

0077 组合.cpp

```cpp
class Solution {
public:
    vector<vector<int>> res;   // result 2d vector
    vector<int> temp;          // temp vector path

    void dfs(int cur, int n, int k) {
    // cur: current element index, choose or not
        if (temp.size() + (n - cur + 1) < k)
            return ;
        if (temp.size() == k) {
            res.push_back(temp);
            return ;
        }

        // choose cur
        temp.push_back(cur);
        dfs(cur+1, n, k);
        temp.pop_back();

        // do not choose cur
        dfs(cur+1, n, k);
    }

    vector<vector<int>> combine(int n, int k) {
        dfs(1, n, k);
        return res;
    }
};

```

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。

```
// choose nums[cur]
temp.push_back(nums[cur]);
dfs(cur+1, nums);
temp.pop_back();

// not choose nums[cur]
dfs(cur+1, nums);
```

复杂度

- 时间复杂度：O($n \times 2^n$)。一共2^n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/

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> temp;

    void dfs(int cur, vector<int> &nums) {
        if (cur == nums.size()) {
            res.push_back(temp);
            return ;
        }

        // choose nums[cur]
        temp.push_back(nums[cur]);
        dfs(cur+1, nums);
        temp.pop_back();

        // not choose nums[cur]
        dfs(cur+1, nums);
    }

    vector<vector<int>> subsets(vector<int>& nums) {
        dfs(0, nums);
```

0078 子集.cpp

```
23          return res;
24      }
25  };
26
```

0090 子集 II

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

标签：回溯

精选题解

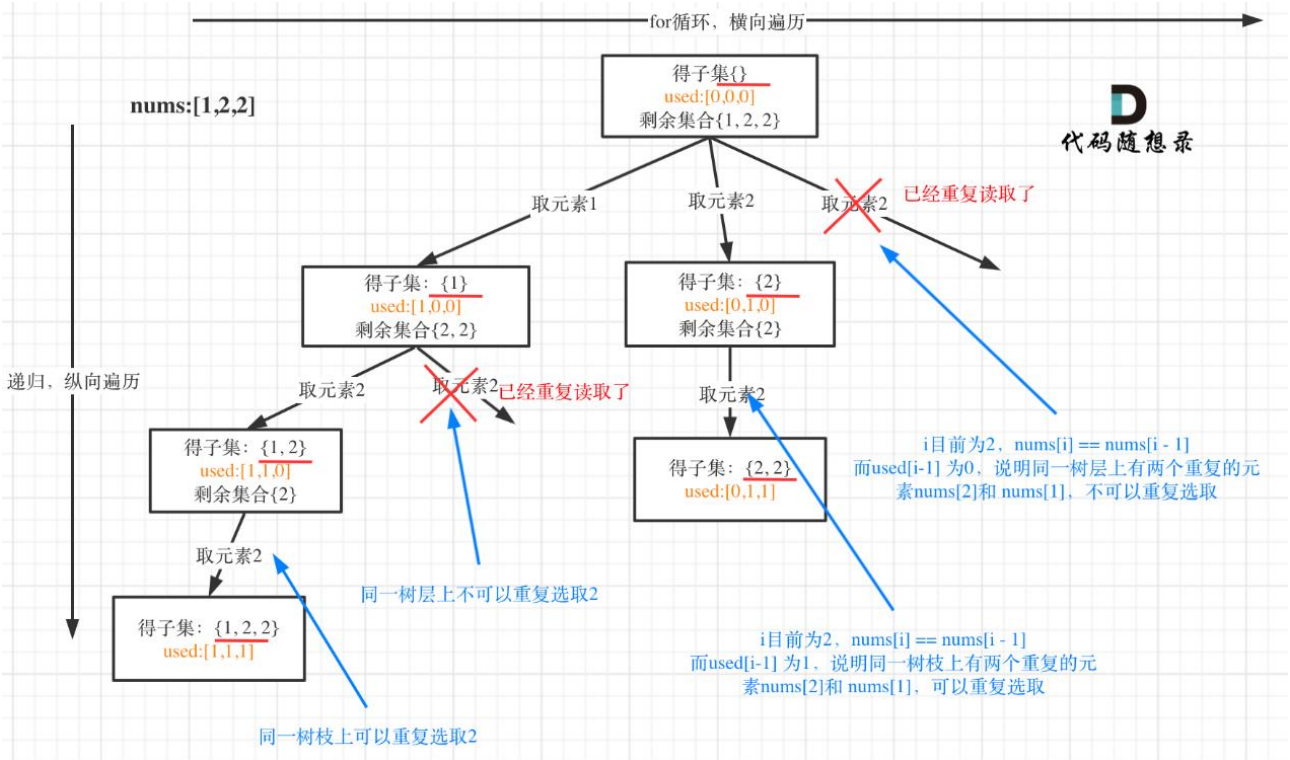
- 90. 子集 II:【彻底理解子集问题如何去重】详解 - 子集 II
  - https://leetcode-cn.com/problems/subsets-ii/solution/90-zi-ji-iiche-di-li-jie-zi-ji-wen-ti-ru-he-qu-zho/

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]。

```
1   // not choose nums[cur]
2   res.push_back(temp);
3
4   // maybe choose nums[cur]
5   for (int i=cur; i<nums.size(); ++i) {
6       if (i>0 && nums[i]==nums[i-1] && !used[i-1])
7           continue;
8       temp.push_back(nums[i]);
9       used[i] = true;
10      backtrack(i+1, nums, used);
11      used[i] = false;
12      temp.pop_back();
13  }
```

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：$O(2^n \times n)$。子集最多有2^n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
```

0090 子集 II

```
6       void backtrack(int cur, vector<int> &nums, vector<bool> &used) {
7           // not choose nums[cur]
8           res.push_back(temp);
9
10          // maybe choose nums[cur]
11          for (int i=cur; i<nums.size(); ++i) {
12              if (i>0 && nums[i]==nums[i-1] && !used[i-1])
13                  continue;
14              temp.push_back(nums[i]);
15              used[i] = true;
16              backtrack(i+1, nums, used);
17              used[i] = false;
18              temp.pop_back();
19          }
20      }
21
22      vector<vector<int>> subsetsWithDup(vector<int>& nums) {
23          vector<bool> used(nums.size(), false);
24          sort(nums.begin(), nums.end());
25          backtrack(0, nums, used);
26          return res;
27      }
28  };
29
```

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
6       void backtrack(int cur, vector<int> &nums) {
7           // not choose nums[cur]
8           res.push_back(temp);
9
10          // maybe choose nums[cur]
11          for (int i=cur; i<nums.size(); ++i) {
12              if (i>cur && nums[i]==nums[i-1])
```

0090 子集 II -v2.cpp

```
13                 continue;
14             temp.push_back(nums[i]);
15             backtrack(i+1, nums);
16             temp.pop_back();
17         }
18     }
19
20     vector<vector<int>> subsetsWithDup(vector<int>& nums) {
21         sort(nums.begin(), nums.end());
22         backtrack(0, nums);
23         return res;
24     }
25 };
26
```

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($M \times N \times 3^L$)。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$M \times N$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3^L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($M \times N$)。visited 数组的空间为$M \times N$；栈的深度至多为min $(L, M \times N)$。

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
11          visited[i][j] = true;
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                  if (check(i_new, j_new, k+1, board, visited, word)) {
19                      visited[i][j] = false;
20                      return true;
21                  }
22              }
23          }
24          visited[i][j] = false;
25          return false;
26      }
27
28      bool exist(vector<vector<char>>& board, string word) {
29          int row_num = board.size();
```

0079 单词搜索.cpp

```
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  };
45
```

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/

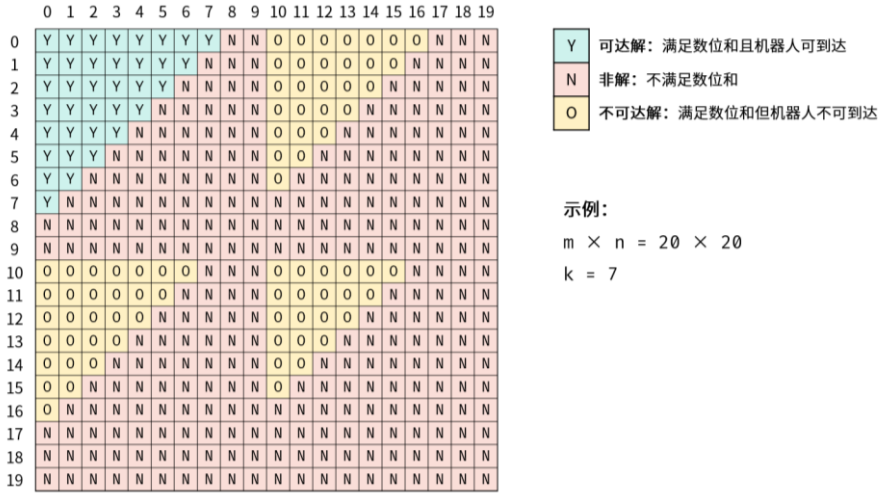
关键思路

审题！审题！审题！

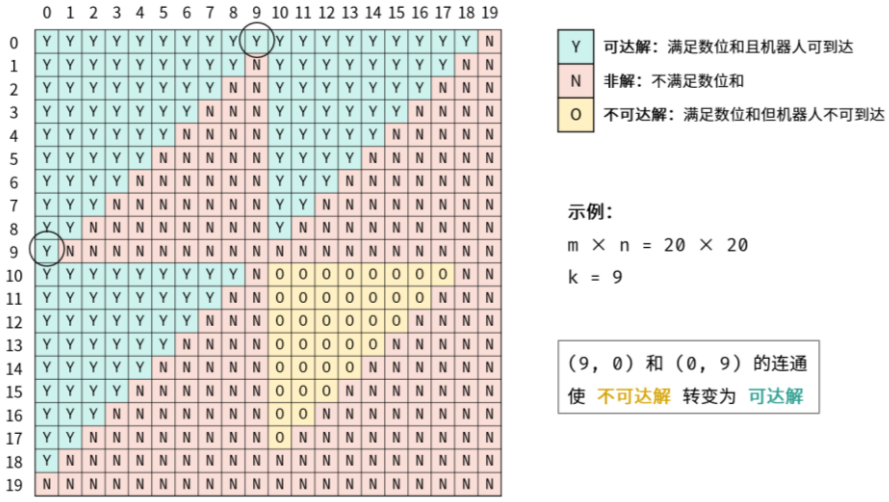
- 这里的 k 并不是指移动 k 步，而仅仅是对行坐标和列坐标数位之和的一个人为限定
- 如果看不懂解法中的某个奇怪的做法，一般都是因为对题意理解有误

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）。

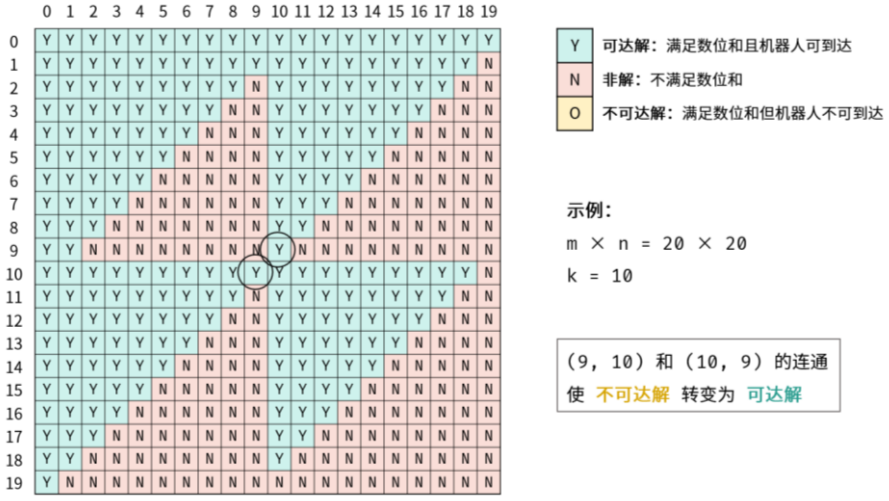
- 尽管机器人在运动时有如下三种情形，但是使用 dfs 向右向下可以保证覆盖完整：



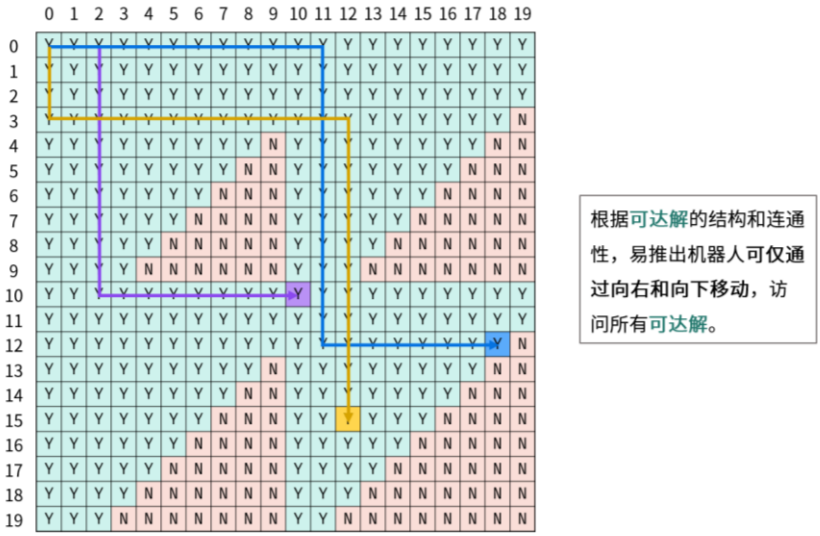
（a）未连通



（b）部分连通

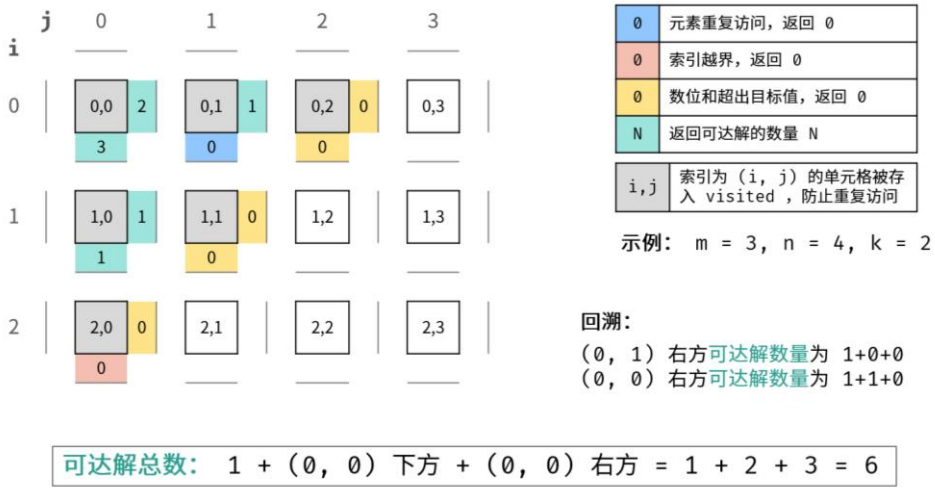


（c）全连通



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($m \times n$)。遍历所有格子。

空间复杂度：O($m \times n$)。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
12      int dfs(int m, int n, int k, int i, int j, vector<vector<bool>>& visited)
    {
```

JZ-13 机器人的运动范围 - dfs.cpp

```
13          if (i>=m || j>=n || visited[i][j] || digitSum(i)+digitSum(j)>k)
14              return 0;
15          visited[i][j] = true;
16          return 1 + dfs(m, n, k, i+1, j, visited) + dfs(m, n, k, i, j+1,
    visited);
17      }
18
19      int movingCount(int m, int n, int k) {
20          vector<vector<bool>> visited(m, vector<bool>(n, false));
21          return dfs(m, n, k, 0, 0, visited);
22      }
23  };
24
```

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
12      int movingCount(int m, int n, int k) {
13          if (k==0)
14              return 1;
15          int res = 1;
16          vector<vector<int>> reachable(m, vector<int>(n, 0));
17          reachable[0][0] = 1;
18          for (int i=0; i<m; ++i) {
19              for (int j=0; j<n; ++j) {
20                  if ( (i==0 && j==0) || digitSum(i)+digitSum(j)>k)
21                      continue;
22                  if (i>=1)
23                      reachable[i][j] |= reachable[i-1][j];
24                  if (j>=1)
```

JZ-13 机器人的运动范围 - iterative.cpp

```
                    reachable[i][j] |= reachable[i][j-1];
                res += reachable[i][j];
            }
        }
        return res;
    }
};

```

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/

```
class Solution {
public:
    vector<string> res;
```

JZ-38 字符串的排列.cpp

```
36      string temp;
37
38      void dfs(string& s, int cnt, vector<bool>& visited) {
39          if (cnt==s.size()) {
40              res.push_back(temp);
41              return ;
42          }
43          for (int i=0; i<s.size(); ++i) {
44              if (visited[i] || i>0 && s[i]==s[i-1] && !visited[i-1])
45                  continue;
46              temp.push_back(s[i]);
47              visited[i] = true;
48              dfs(s, cnt+1, visited);
49              visited[i] = false;
50              temp.pop_back();
51          }
52      }
53
54      vector<string> permutation(string s) {
55          if (s.empty())
56              return res;
57          sort(s.begin(), s.end());
58          vector<bool> visited(s.size(), false);
59          dfs(s, 0, visited);
60          return res;
61      }
62  };
63
```

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（2^{31}-1），最小值 INT_MIN 为-2,147,483,648（-2^{31}），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 $\log_{10} x$。
- 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 };
17
```

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/

```cpp
class Solution {
public:
    bool isPalindrome(int x) {
        if (x<0 || (x%10==0 && x!=0))
            return false;
        int rev = 0;
        while (rev < x) {
            rev = rev * 10 + x % 10;
            x = x / 10;
        }

        return (x == rev || x == rev/10);
    }
};

```

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 $b_{m-1}\cdots b_1 b_0$，那么 n 的十进制表示为 $2^{m-1}\cdot b_{m-1}+\cdots+2^1\cdot b_1+2^0\cdot b_0$，则 x^n 的十进制表示为 $x^{2^{m-1}\cdot b_{m-1}+\cdots+2^1\cdot b_1+2^0\cdot b_0}$，更进一步地，只保留 b_i 中值为 1 的项，也即 $x^n=x^{\sum_{b_i=1}2^i}=\prod_{b_i=1}x^{2^i}$，其中 $0\le i\le m-1$。
- 所以只需要求解两个子问题：
  - 计算 2^i 的值，其中 $0\le i\le m-1$
    - 循环执行 x = x * x 即可
  - 获取 b_i 的值，其中 $0\le i\le m-1$：
    - 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 $x^{-(-n)}$ 也即 $\left(\frac{1}{x}\right)^{-n}$ 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$\left(\frac{1}{x}\right)^{-(n-1)}\cdot x$。

$$
\begin{aligned}
n &= 9 \\
&= 1001_b \\
&= 1 \times 1 + 0 \times 2 + 0 \times 4 + 1 \times 8 \\
&\quad (b_1 \times 2^0 + b_2 \times 2^1 + b_3 \times 2^2 + b_4 \times 2^3)
\end{aligned}
$$

$$
x^n = x^9 = x^{1\times1} x^{0\times2} x^{0\times4} x^{1\times8}
$$

1 蓝色数字代表 b_i

1 橙色数字代表 2^{i-1}

循环	$x^n \times (res)$
第 0 轮	$3^5 \times (1)$
第 1 轮	$9^2 \times (1 \times 3)$
第 2 轮	$81^1 \times (1 \times 3)$
第 3 轮	$6561^0 \times (1 \times 3 \times 81)$
返回	$1 \times 3 \times 81$

```
while n:
  if n & 1:        (即 n % 2 == 1)
    res *= x
  x *= x           (即 x = x ^ 2)
  n >>= 1          (即 n //= 2)
```

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/

```cpp
class Solution {
public:
    double myPow(double x, int n) {
        if (x==0)
            return 0;
        double res=1;
        if (n<0) {
            if (n==INT_MIN) { // 最小的负数会溢出
                return myPow(x,n+1) * x;
            } else {
                x = 1/x;
                n = -n;
            }
        }
        while (n) {
            if (n&1) { // 如果该位为 1，则将 res 乘上此时的 x
                res *= x;
            }
            x *= x;
            n >>= 1;
        }
        return res;
    }
```

0050 pow(x,n).cpp
24 }; 25

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

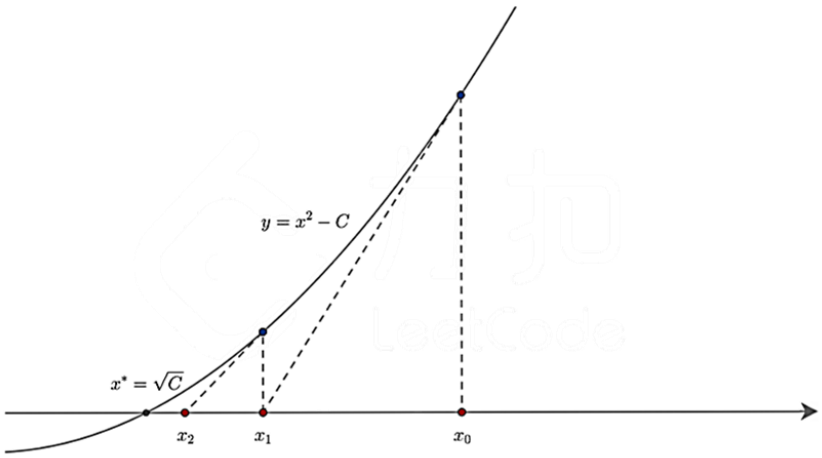
二分法和牛顿迭代法。

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 C 为传入的参数 x（不是后面函数中的 x），那么 C 的平方根就是函数 $y = f(x) = x^2 - C$ 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 $(x_i, f(x_i))$ 作切线与 x 轴相交于 x_{i+1}。
  - 新的交点 x_{i+1} 比旧的交点 x_i 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 $\epsilon = 10^{-7}$ 就足够了。
- 选取迭代初值 $x_0 = C$。
    - 已知 $y = f(x) = x^2 - C$ 的两个零点分别为 $-\sqrt{C}$ 和 $\sqrt{C}$。
    - 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 x_i，则函数上该点为 $(x_i, x_i^2 - C)$，切线斜率为 $2x_i$，切线方程为 $y = 2x_i(x - x_i) + x_i^2 - C = 2x_i x - (x_i^2 + C)$，得新零点 $x_{i+1} = \frac{1}{2}(x_i + \frac{C}{x_i})$。

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/

```cpp
class Solution {
public:
    int mySqrt(int x) {
        int res = 0;
        int low = 0, high = x;
        while (low <= high) {
            int mid = low + (high-low)/2;
            if ((long long) mid*mid <= x) { // 防止整型溢出，做乘法时转成 long long
                res = mid;
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return res;
    }
};

```

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/

```cpp
class Solution {
public:
    int mySqrt(int x) {
        if (x == 0)
            return 0;
        double C = x, x_old = x, x_new;
        while (1) {
            x_new = 0.5 * (x_old + C/x_old);
            if (fabs(x_new - x_old) < 1e-7)
                break;
            x_old = x_new;
        }
        return int(x_old);
    }
};

```

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/

0171 Excel 表列序号.cpp

```cpp
class Solution {
public:
    int titleToNumber(string s) {
        int res = 0;
        for (auto &ch: s) {
            res = res*26 + (ch-'A'+1);
        }
        return res;
    }
};

```

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。
- 5，5^2，5^3，…，5^k 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 5^2 个数有 2 个因子 5，每隔 5^k 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/

```cpp
class Solution {
public:
```

0172 阶乘后的零.cpp

```
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  };
12
```

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

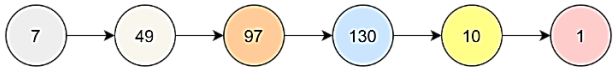
两种方法：

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。

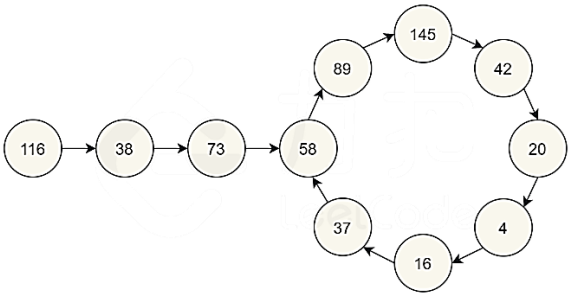
- 也即 Floyd 算法（代码 2 第 15 ~ 17 行），可参考“0141 环形链表”。

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。



2. 进入循环，永远不会到 1。



3. 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
12      bool isHappy(int n) {
13          unordered_set<int> set;
14          while (1) {
15              if (n==1)
16                  return true;
17              n = squareSum(n);
18              if (set.find(n) != set.end()) {
19                  return false;
20              } else {
21                  set.insert(n);
22              }
23          }
24      }
25  };
26
```

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;
```

0202 快乐数 - 快慢指针.cpp

```
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
12      bool isHappy(int n) {
13          int slow = n, fast = n;
14          do {
15              slow = squareSum(slow);
16              fast = squareSum(squareSum(fast));
17          } while (slow != fast);
18          return slow == 1;
19      }
20  };
21
```

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  };
7
```

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/

0326 3 的幂.cpp

```
class Solution {
public:
    bool isPowerOfThree(int n) {
        if (n < 1)
            return false;
        while (n % 3 == 0) {
            n /= 3;
        }

        return n == 1;
    }
};

```

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 $x = \sum_{i=0}^{n-1} a_i 10^i$，其中 a_i 表示每一位上的数字，又 $10^i \equiv (9+1)^i \equiv 1 \, mod \, 9$，所以 $x \equiv \sum_{i=0}^{n-1} a_i \, mod \, 9$。

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/

```cpp
class Solution {
public:
    int addDigits(int num) {
        while (num >= 10) {
            int next = 0;
            while (num > 0) {
                next += num % 10;
                num = num / 10;
            }
            num = next;
        }
        return num;
    }
};

```

代码 2 – 公式法

0258 各位相加 – 公式.cpp

https://leetcode-cn.com/submissions/detail/141362718/

```cpp
class Solution {
public:
    int addDigits(int num) {
        return (num-1) % 9 + 1;
    }
};

```

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 $\frac{n(n+1)}{2}$，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/

```cpp
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int res = nums.size();
        for (int i=0; i<nums.size(); ++i) {
            res ^= i ^ nums[i];
        }
        return res;
    }
};

```

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/

```cpp
class Solution {
public:
    vector<int> selfDividingNumbers(int left, int right) {
        vector<int> res;
        for (int num=left; num<=right; ++num) {
            int tmp = num;
            int mod = 0;
            while (tmp!=0) {
                mod = tmp % 10;
                if (mod==0 || num%mod!=0)
                    break;
                tmp /= 10;
            }
            if (tmp==0)
                res.push_back(num);
        }
        return res;
    }
};
```

0728 自除数.cpp
20

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/

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode *dummy = new ListNode(-1);
        ListNode *p = dummy;
        int sum = 0;
        int carry = 0;
        while (l1 || l2) {
            sum = 0;
            if (l1) {
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2) {
                sum += l2->val;
                l2 = l2->next;
            }
            sum += carry;
            carry = sum / 10;
            p->next = new ListNode(sum % 10);
            p = p->next;
        }
        if (carry>0)
            p->next = new ListNode(1);
        return dummy->next;
    }
};

```

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/

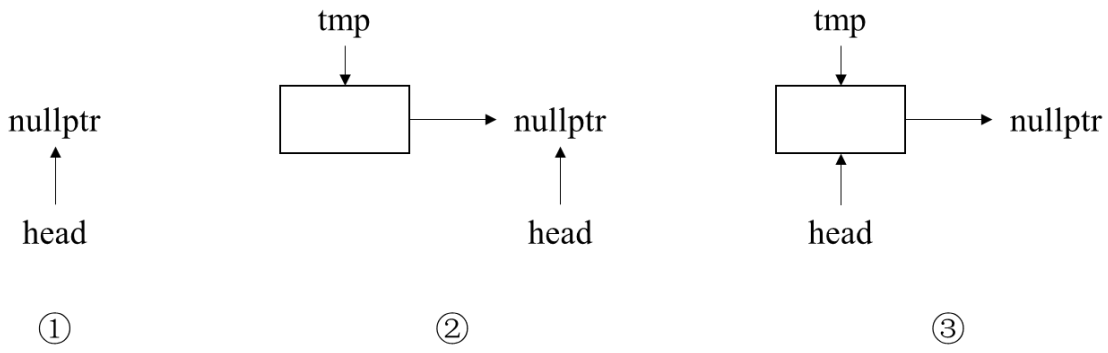
关键思路

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.
```

0445 两数相加 II.cpp

```
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  };
43
```

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

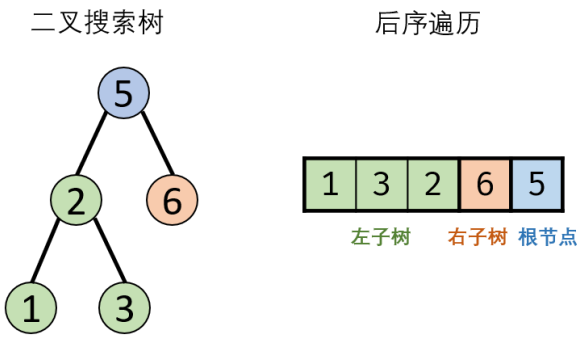
精选题解

- 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

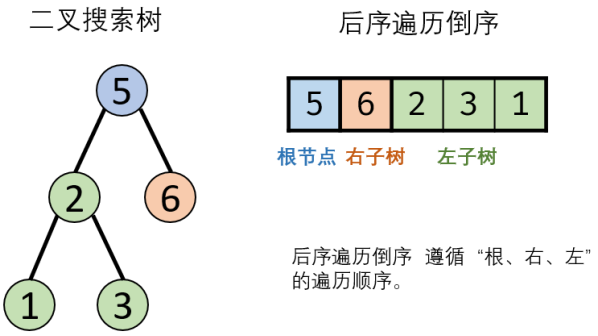
- 递归判断上面分割得到的左右子树各自是否对应正确的后序遍历数组
- 注意代码 1 第 13 行在循环中使用 tmp++的简洁写法

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：

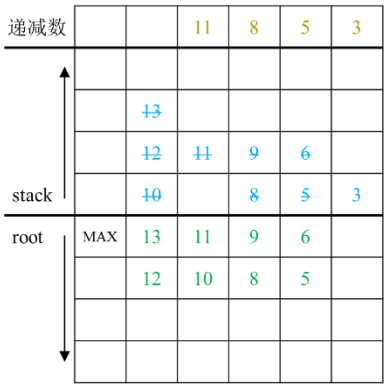
1、如果递增，也即 $n_i < n_{i+1}$，比如上图中 ⑤ < ⑥，则 ⑥ 必为 ⑤ 的右子结点。理由：

- 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 n_{i+1} 必为 n_i 的右子结点，且由于最先遍历根结点，因此 n_{i+1} 还必定是 n_i 紧邻的右子结点

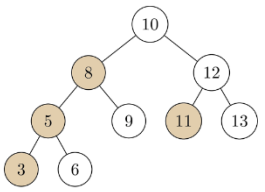
2、如果递减，也即 $n_i > n_{i+1}$，比如上图中 ③ > ①，则 ① 必为其左边某个节点 root 的左子结点，且 root 的值是 ① 左边结点（⑤⑥②③）中最小的，也即 ②。理由：

- 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 n_{i+1} 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 n_{i+1} 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？

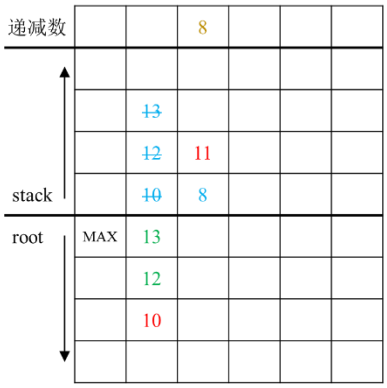


[10, 12, 13, 11, 8, 9, 5, 6, 3]
反转的后序遍历数组
根→右→左

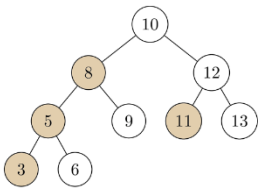



```
stack<int> stk;
int root = INT_MAX;
for (int i=postorder.size()-1; i>=0; --i) {
    int cur = postorder[i];
    while (!stk.empty() && cur<stk.top()) {
        root = stk.top();
        stk.pop();
    }
    if (cur>root) {
        return false;
    }
    stk.push(cur);
}
return true;
```

正确的后序遍历数组



[10, 12, 13, 8, 11, 9, 5, 6, 3]
反转的后序遍历数组
根→右→左




```
stack<int> stk;
int root = INT_MAX;
for (int i=postorder.size()-1; i>=0; --i) {
    int cur = postorder[i];
    while (!stk.empty() && cur<stk.top()) {
        root = stk.top();
        stk.pop();
    }
    if (cur>root) {
        return false;
    }
    stk.push(cur);
}
return true;
```

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(n^2)/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  };
23
```

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
18         return true;
19     }
20 };
21
```

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。**

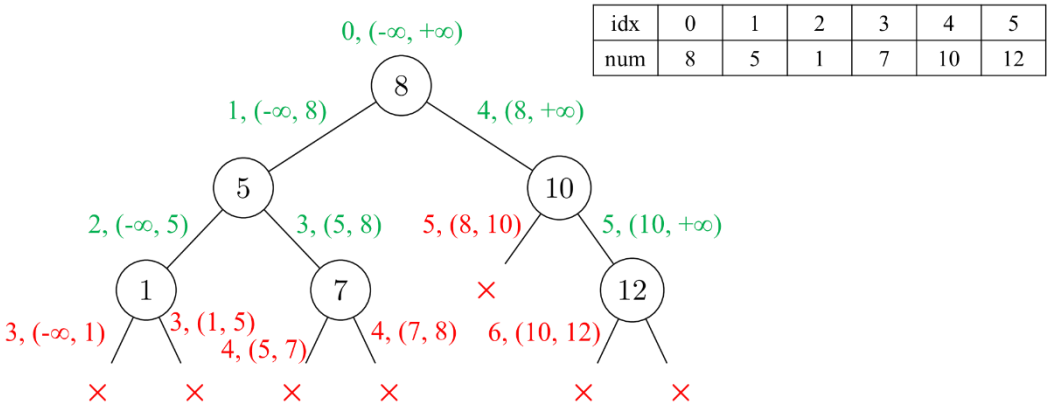
1、递归

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  };
16
```

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) {
```

1008 前序遍历构造二叉搜索树 – iterative.cpp

```
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  };
23
```

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/

```cpp
class Solution {
public:
    vector<int> exchange(vector<int>& nums) {
        int left = 0, right = nums.size()-1;
        while (left<right) {
            if (nums[left] % 2 == 1) {
                ++left;
                continue;
            }
            if (nums[right] % 2 == 0) {
                --right;
                continue;
            }
            swap(nums[left++], nums[right--]);
        }
        return nums;
    }
};

```

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
5       int helper(string &s, int &i) {
6           char op = '+';
7           stack<int> stk;
8           int num = 0;
9           int res = 0;
10          int top = 0;
11          for (i; i<s.size(); ++i) {
12              if (s[i]>='0' && s[i]<='9') { // 计算数字
13                  num = num * 10 + (s[i]-'0');
14              }
15              if (s[i]=='(') { // 左括号进入递归，在新的递归中，stk 和 op 都被重置
```

```
16                 num = helper(s, ++i);
17                 ++i; // 出了右括号，将指针右移一位
18             }
19 
20             // if (i >= s.size()-1 || ((s[i]<'0' || s[i]>'9') && s[i] != ' '))
   {
21             if (i>=s.size()-1 || find(op_v.begin(), op_v.end(), s[i]) !=
   op_v.end() || s[i]==')') {
22             // 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）
23                 if (op=='+') {
24                     stk.push(num);
25                 }
26                 if (op=='-') {
27                     stk.push(-num);
28                 }
29                 if (op=='*') {
30                     top = stk.top();
31                     stk.pop();
32                     stk.push(top*num);
33                 }
34                 if (op=='/') {
35                     top = stk.top();
36                     stk.pop();
37                     stk.push(top/num);
38                 }
39                 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
40                 num = 0;
41             }
42 
43             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
44                 break;
45             }
46         }
47 
48         // 计算栈中数字之和
49         while (!stk.empty()) {
50             res += stk.top();
51             stk.pop();
52         }
53         return res;
54     }
```

```
55
56      int calculate(string s) {
57          int i = 0;
58          return helper(s, i);
59      }
60  };
61
```

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]) {
```

0946 验证栈序列 + JZ-31 栈的压入、弹出序列.cpp

```
                s.pop();
                ++i;
            }
        }
        return s.empty();
    }
};

```

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 $\log_2 num$ 次。
- 空间复杂度：O(1)。

代码

https://leetcode-cn.com/submissions/detail/141431329/

```
class Solution {
public:
    bool isUgly(int num) {
        if (num<=0)
            return false;
```

```
6
7           while (num != 1) {
8               if (num%2==0)
9                   num /= 2;
10              else if (num%3==0)
11                  num /= 3;
12              else if (num%5==0)
13                  num /= 5;
14              else
15                  return false;
16          }
17
18          return true;
19      }
20  };
21
```

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) {
```

JZ-49 0264 丑数 II.cpp

```
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  };
16
```

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。

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 $\sqrt{i}$，for i from 1 to n
- 返回 dp[n]

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 $n = 4^p(8q + 7)$，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：$O(n\sqrt{n})$。外层遍历 n 个数，内层遍历 $\sqrt{n}$ 个数。
- 空间复杂度：O(n)。dp 数组的大小。

复杂度 – 数学定理

- 时间复杂度：$O(\sqrt{n})$。判断能否用两个完全平方数表示时，需要遍历 $\sqrt{n}$ 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/

```cpp
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n+1,0);
        for (int i=1; i<=n; ++i) {
            dp[i] = i;
            for (int j=1; i-j*j>=0; ++j) {
                dp[i] = min(dp[i], dp[i-j*j]+1);
            }
        }
        return dp[n];
    }
};

```

代码 2 – 数学定理

0279 完全平方数 – 公式.cpp

https://leetcode-cn.com/submissions/detail/141560552/

0279 完全平方数 – 公式.cpp

```cpp
class Solution {
public:
    bool isSquare(int x) {
        int sq = int(sqrt(x));
        return sq*sq==x;
    }
    int numSquares(int n) {
        while (n%4==0)
            n>>=2;
        if (n%8==7)
            return 4;
        if (isSquare(n))
            return 1;
        for (int i=1; i<=int(sqrt(n)); ++i){
            if (isSquare(n-i*i))
                return 2;
        }
        return 3;
    }
};

```

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/

```cpp
class Solution {
```

0313 超级丑数.cpp

```
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      }
20  };
21
```

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

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/

```cpp
class Solution {
public:
    string minNumber(vector<int>& nums) {
        vector<string> vs;
        string res;
        for (auto const &n: nums) {
            vs.push_back(to_string(n));
        }

        sort(vs.begin(), vs.end(),
            [](string& s1, string& s2) {
                return s1+s2 < s2+s1;  // if condition is true, then s1 < s2
            }
        );

        for (auto const &s: vs) {
            res += s;
        }
        return res;
    }
};

```