刷题笔记

by xian2207, wszhangxian@126.com

Part A：LeetCode篇

参考：

[1] 中英题目+答案：https://github.com/haoel/leetcode

[2] 答案和难度分类：https://github.com/zhuli19901106/leetcode-zhuli/tree/master/algorithms

1 Two Sum

```
Given an array of integers, return **indices** of the two numbers such that they
add up to a specific target.

You may assume that each input would have **exactly** one solution, and you may
not use the same element twice.

Example:
Given nums = [2, 7, 11, 15], target = 9,
Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

翻译：给定一个数组，如果两个元素之和等于target value，则返回这两个元素的下标。

思路：暴力遍历

1. 第一个数与后续所有数字相加，看是否等于target value。等于则返回这两个元素的下标；
2. 接着让第二个数字根后续所有数字相加，同上，直到所有数字两两遍历完。

方法1： 使用数组并返回指针

```
class Solution{
    public:
    int* get_indices(const int arr[], int size, int target){
        for(int i = 0; i < size-1; i++){
            for(int j = i+1; j < size; j++){
                bool is_target_found = ((arr[i]+arr[j])==target);
                if(is_target_found){
                    static int indices[2] = {0};
                    indices[0] = i;
                    indices[1] = j;
                    return indices;
                }
            }
        }
        return NULL;
    }
}
int main(){
    Solution soln;
    const int arr[] = {1,2,3,4,5};
    const int size =  sizeof(arr)/size(arr[0]);
    int target = 3;
    int* index = NULL;
    index = soln.get_indices(arr, size, target);
    if(!index){
        printf("NULL!\n");
    }
    printf("i= %d; j= %d\n",i,j);
    return 0;
}
```

注意：

1. public不能忘，否则不能调用该函数；
2. 定义数组是const int arr[]，则传递数组也是const int arr[]；定义数组用int arr[], 传递也是int arr[]；
3. 在栈上定义数组，返回指针时，必须static int indices[]，不能用int indices这样的local变量；
4. 计算数组长度最好在传参之前给定实值或定义宏计算数组长度，不建议在被调函数里计算；
5. 注意对指针的非空判断，显得严密。

方法2：使用vector, 返回vector.

```
class Solution{
    public:
    vector<int> get_indices(vector<int>v, int target){
        int len = v.size();
        for(int i = 0; i < len-1; i++){
            for(int j = i+1; j < len; j++){
                if(v[i]+v[j] == target){
                    return {i,j}; //返回有值的vector<int>，元素为i,j
                }
            }
        }
        return {};  //返回空的vector<int>
    }
}
```

2 Zigzag Conversion

```
The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of
rows like this: (you may want to display this pattern in a fixed font for better
legibility)
P   A   H   N
A P L S I I G
Y   I   R
And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of
rows:
    string convert(string s, int numRows);

Example 1:
Input: s = "PAYPALISHIRING", numRows = 3
Output: "PAHNAPLSIIGYIR"
```

```
Example 2:
Input: s = "PAYPALISHIRING", numRows = 4
Output: "PINALSIGYAHRPI"
Explanation:
P     I    N
A   L S  I G
Y A   H R
P     I
```

翻译：将字符串 “PAYPALISHIRING” 以Z字形排列成给定的行数，之后从左往右，逐行读取符：“PAHNAPLSIIGYIR” 实现一个将字符串进行指定行数变换的函数: string convert(string s, int numRows);

思路：

1. 排除边界情况，例如一个字符串或给定的字符串作为Z字形的第一列；
2. 非边界按照Z字形排列；
3. 假设要求N行，则最终希望是v[0]存储第一行，v[1]存储第二行，v[N]存第N行；
4. 声明一个string变量，让其等于v[0]+v[1]+...+v[N]，这个变量就是最终的输出。

```
class ZigzagConversion{
    public:
    string convert(string s, int numRows){
        if(numRows <= 1 || numRows >= s.size()){
            return s;
        }
        vector<string> v(numRows);
        int row = 0;
        int step = 0;
        for(int i = 0; i < s.size();i++){
            if(row == numRows-1){
                step = -1;
            }
            if(row == 0){
                step = 1;
            }
            v[row] += s[i];
            row += step;
        }
        string result;
        for(int i = 0; i < numRows; i++){
            result += v[i];
        }
        return result;
    }
};
```

3 Reverse Integer

```
Given a 32-bit signed integer, reverse digits of an integer.

Example 1:
Input: 123
```

```
Output: 321

Example 2:
Input: -123
Output: -321

Example 3:
Input: 120
Output: 21

Note:
Assume we are dealing with an environment which could only store integers within
the 32-bit signed integer range: [−231, 231 − 1]. For the purpose of this
problem, assume that your function returns 0 when the reversed integer
overflows.
```

翻译：数字倒序排列且不能溢出。

思路：

1. 数字逆序要先求余数；
2. 余数乘以10+余数，余数位就会移动到最左端；
3. 除数除以10，除数向右移一位；
4. 重复1-3直到第一步中的余数为0；

```
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX - 1)
class ReverseInteger {
 public:
  int reverse_integer(int x) {
    int y = 0;
    int n;
    while (x != 0) {
      n = x % 10;
      //should be y>(INT_MAX-n)/10, but n/10 is 0, so ignore it.
      if (y > INT_MAX / 10 || y < INT_MIN / 10) {
        return 0;
      }
      y = y * 10 + n;  //(INT_MAX-n)/10才是y乘以10+n之后的值，防止溢出，所以在上面做判
断
      x /= 10;
    }
    return y;
  }
};
```

4 String to Integer

```
Implement atoi which converts a string to an integer. The function first discards
as many whitespace characters as necessary until the first non-whitespace
character is found. Then, starting from this character, takes an optional initial
plus or minus sign followed by as many numerical digits as possible, and
interprets them as a numerical value.

The string can contain additional characters after those that form the integral
number, which are ignored and have no effect on the behavior of this function.
```

```
If the first sequence of non-whitespace characters in str is not a valid integral
number, or if no such sequence exists because either str is empty or it contains
only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned.
Note:
    Only the space character ' ' is considered as whitespace character.
    Assume we are dealing with an environment which could only store integers
within the 32-bit signed integer range: [−231,  231 − 1]. If the numerical value
is out of the range of representable values, INT_MAX (231 − 1) or INT_MIN (−231)
is returned.

Example 1:
Input: "42"
Output: 42

Example 2:
Input: "   -42"
Output: -42
Explanation: The first non-whitespace character is '-', which is the minus sign.
             Then take as many numerical digits as possible, which gets 42.

Example 3:
Input: "4193 with words"
Output: 4193
Explanation: Conversion stops at digit '3' as the next character is not a
numerical digit.

Example 4:
Input: "words and 987"
Output: 0
Explanation: The first non-whitespace character is 'w', which is not a numerical

             digit or a +/- sign. Therefore no valid conversion could be
performed.

Example 5:
Input: "-91283472332"
Output: -2147483648
Explanation: The number "-91283472332" is out of the range of a 32-bit signed
integer.
             Thefore INT_MIN (−231) is returned.
```

翻译：将字符串转化为整数，规则是整数前（1）不能有空白；（2）不能有字符；（3）要有正负号；（4）越界要处理；（5）现有字符后有整数的，返回0即可。

思路：

1. 判断字符串是否为空，否则直接返回0；
2. 判断是否有空白，有则指针后移直到无white space；
3. 在无white space时判断是否存在正负号；
4. 正负号后采用isdigit函数判断是否有数字；
5. 字符串的ASK码减去‘0’的ASK码即为该数字的大小；
6. 字符串从前向后排序得到数字（例如：“123”为：1，$1*10+2$，$(1*10+2)*10+3$）
7. 如果数字越界则返回INT_MAX或INT_MIN

```
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX - 1)

using namespace std;

class StringToInteger {
 public:
  int my_atoi(const char* str) {
    //1 判断空字符串数组
    if (*str == '\0')  {
      return 0;
    }
    int num = 0;
    int sign = 1;
    int index = 0;
    //2 判断while space
    while (str[index] == ' ') {
      index++;
    }
    //3 判断+/-号的情况
    if (str[index] == '-') {
      sign = -1;
      index++;
    } else if (str[index] == '+') {
      index++;
    }
    //4 isdigit测试*str是否具有数字特征
    while (isdigit(str[index]))  {
      int i = str[index] - '0';  // ASK码相减得到数字
      num =num * 10 + i;  //字符串从前到后排序得到数字
      if (num * sign > INT_MAX) {
        num = INT_MAX;
        break;
      } else if (num * sign < INT_MIN) {
        num = INT_MIN;
        break;
      }
      index++;
    }
    return int(sign * num);
  }
};
```

5 Palindrome Number

```
Determine whether an integer is a palindrome. An integer is a palindrome when it
reads the same backward as forward.

Example 1:
Input: 121
Output: true

Example 2:
Input: -121
Output: false
```

```
Explanation: From left to right, it reads -121. From right to left, it becomes
121-. Therefore it is not a palindrome.

Example 3:
Input: 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.
```

翻译：判断是否为回文，即正反读到的一样。

思路：假设原始值为x

1. 先判断边界，负数一定不是回文，返回false；
2. 声明y保存逆序值；
3. 原始值求逆序（先求余数，然后y=y*10+余数；其次x/=10；直到x=0时停下）；
4. 看y是否等于x。

```
class PalindromeNumber {
public:
    bool palindrome_number(int x) {
        if (x < 0) {
            return false;
        }
        const int R = 10;
        int temp = x;
        int y = 0;
        int n = 0;
        while (temp > 0) {
            n = temp % R;
            y = y * R + n;
            temp /= R;
        }
        return y == x;
    }
};
```

6 Rome to Integer

```
Roman numerals are represented by seven different symbols: I, V, X, L, C, D and
M.

Symbol       Value
I             1
V             5
X             10
L             50
C             100
D             500
M             1000

For example, two is written as II in Roman numeral, just two one's added
together. Twelve is written as, XII, which is simply X + II. The number twenty
seven is written as XXVII, which is XX + V + II.
```

```
Roman numerals are usually written largest to smallest from left to right.
However, the numeral for four is not IIII. Instead, the number four is written as
IV. Because the one is before the five we subtract it making four. The same
principle applies to the number nine, which is written as IX. There are six
instances where subtraction is used:

    I can be placed before V (5) and X (10) to make 4 and 9.
    X can be placed before L (50) and C (100) to make 40 and 90.
    C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer. Input is guaranteed to be within
the range from 1 to 3999.

Example 1:
Input: "III"
Output: 3

Example 2:
Input: "IV"
Output: 4

Example 3:
Input: "IX"
Output: 9

Example 4:
Input: "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.

Example 5:
Input: "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

翻译：将罗马数字转换为整数。

思路：

1. 根据输入的字符串，最好用const char *s = "..."（跨平台和混编），先判断字符串是否为空；
2. 不为空时判断每个字符对应的罗马数字的值是多少；
3. 两两对比，如果前一个字符大于后一个，则 result = result - prev + current - prev；如果小于，则 result += current；
4. 返回result即可。

```
int char_to_int(char ch) {
  int d = 0;
  switch (ch) {
    case 'I':
      d = 1;
      break;
    case 'V':
      d = 5;
      break;
    case 'X':
      d = 10;
      break;
    case 'L':
      d = 50;
      break;
    case 'C':
      d = 100;
      break;
    case 'D':
      d = 500;
      break;
    case 'M':
      d = 1000;
      break;
    default:
      break;
  }
  return d;
}
class RomanToInteger {
 public:
  int roman_to_int(const char* s) {
    if (*s == '\0') {
      printf("empty string!\n");
      return -1;
    }
    int s_size = 0;
    while (s[s_size] != '\0') {
      s_size++;
    }
    int result = char_to_int(s[0]);
    for (int i = 1; i < s_size; i++) {
      int prev = char_to_int(s[i - 1]);
      int curr = char_to_int(s[i]);
      // if left<right such as : IV(4), XL(40), IX(9) ...
      if (prev < curr) {
        result = result - prev + (curr - prev);
      } else {
        result += curr;
      }
    }
    return result;
  }
};
```

注意：

1. 定义字符串用const char *s = ”....”，函数里传参也用 const char* s;
2. C语言求字符串长度，不推荐用sizeof，建议用函数求解长度，如 while(*s[i++] != '0') return i 方式。

7 Longest Common Prefix

```
Write a function to find the longest common prefix string amongst an array of
strings. If there is no common prefix, return an empty string "".

Example 1:
Input: ["flower","flow","flight"]
Output: "fl"

Example 2:
Input: ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.

Note: All given inputs are in lowercase letters a-z.
```

翻译：求字符数组中的字符串相同前缀的最大数。

思路：

1. 如果字符串组为空，则返回空的字符串；
2. 非空时，遍历字符串组A,B,C，取第一个A为参考组 A, 依次对比B和C中的元素与A的元素；
3. 重复最多的字符就是string型结果。

```
class LongestCommonPrefix {
public:
    string longest_common_prefix(vector<string>& strs) {
        int n = strs.size();
        if (n == 0) {
            return "";
        }
        string temp = strs[0];
        int i, j;
        int len;
        for (i = 1; i < n; ++i) {
            len = temp.size();
            for (j = 0; j < len; ++j) {
                if (strs[i][j] != temp[j]) {
                    temp.resize(j);
                    break;
                }
            }
        }
        return temp;
    }
};
```

注意：

- C语言定义字符串组：const char *s[] ={"abc, abcd, ad"}，求长度不方便。建议使用 $vector<string>$，因为string 定义的变量求长同vector，可直接xxx.size()；
- 函数返回类型为string型，空则返回 return " "；
- strlen最好少用，适用于char[50], strcpy(..,...,...)求长度。

8 Remove Nth from End of List*

```
Given a linked list, remove the n-th node from the end of list and return its
head.

Example:
Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
Note:
Given n will always be valid.
```

翻译：删除单链表末端的指定位置的元素，后面元素迁移，长度减1。

思路：

9 Valid Parentheses

```
Given a string containing just the characters '(', ')', '{', '}', '[' and ']',
determine if the input string is valid.

An input string is valid if:
Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.

Note that an empty string is also considered valid.

Example 1:
Input: "()"
Output: true

Example 2:
Input: "()[]{}"
Output: true

Example 3:
Input: "(]"
Output: false

Example 4:
Input: "([)]"
Output: false

Example 5:
Input: "{[]}"
Output: true
```

翻译：括号匹配问题。

思路：

1. 大方向是利用stack栈；
2. 字符串如果不是{} [] ()，直接返回false；
3. 如果是则栈先装入 {, [, (；
4. 接着对后续的排查，看是否等同于),], }；
5. 如果配对，则pop掉，否则return false。最终栈空视为配对完毕，返回真。

```
class ValidParentheses {
 public:
  bool valid_parentheses(string s) {
    stack<char> st;
    for (auto ch : s) {
      if (ch == '{' || ch == '[' || ch == '(') {
        st.push(ch);
      } else if (ch == '}' || ch == ']' || ch == ')') {
        if (st.empty()) return false;
        char sch = st.top();
        printf("sch: %c\n", sch);
        if ((sch == '{' && ch == '}') || (sch == '[' && ch == ']') ||
            (sch == '(' && ch == ')')) {
          st.pop();
        } else {
          return false;
        }
      } else {
        return false;
      }
    }
    return st.empty();
  }
};
```

注意：栈是后进先出的

- $\mathrm{stack{<}int{>}\ st}$ 不能像**vector**和**string**通过下标一样访问 **stack**，只能 top, pop, empty, size, push等几个函数来访问或压入元素；
- top的意思是栈顶，必须配合 pop出栈。缺乏pop，会一直卡在栈顶。例如进栈元素分别是3,10,2,4. 则st.top是4，再执行st.top()还是4， 除非配合st.pop(), st.top才会变为2.

10 Merge Two Sorted List*

11 Remove Duplicates from Array

```
Given a sorted array nums, remove the duplicates in-place such that each element
appear only once and return the new length.

Do not allocate extra space for another array, you must do this by modifying the
input array in-place with O(1) extra memory.

Example 1:
Given nums = [1,1,2],
Your function should return length = 2, with the first two elements of nums being
1 and 2 respectively.

Example 2:
Given nums = [0,0,1,1,1,2,2,3,3,4],
Your function should return length = 5, with the first five elements of nums
being modified to 0, 1, 2, 3, and 4 respectively.

Clarification:
Confused why the returned value is an integer but your answer is an array?
Note that the input array is passed in by reference, which means modification to
the input array will be known to the caller as well.
```

```
Internally you can think of this:
// nums is passed in by reference. (i.e., without making a copy)
int len = removeDuplicates(nums);

// any modification to nums in your function would be known by the caller.
// using the length returned by your function, it prints the first len elements.
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

翻译：删除数组中的重复元素，打印它们并返回其新的长度。

思路：

1. 不能额外声明数组变量，所以只能在原来数组上想办法；
2. 取第一个元素，遍历到第二个，对比二者，如果相同，则下标不变，第二个的值去覆盖第一个数组元素；如果不同，则数组下标增加，该下标对应元素为那个不同的值；
3. 接着以上次的值为初值，遍历到第三个，同上，直到所有遍历完。

```
class RemoveDuplicatesFromArray {
 public:
  int remove_duplicates_from_array(int A[], int n) {
    if (n <= 1) {
      return n;
    }
    int index = 0;
    for (int i = 0; i < n - 1; i++) {
      if (A[i] != A[i + 1]) {
        index += 1;
        A[index] = A[i + 1];
      }
    }
    return index + 1;
  }
};
```

12 Remove Elements

```
Given an array nums and a value val, remove all instances of that value in-place
and return the new length.

Do not allocate extra space for another array, you must do this by modifying the
input array in-place with O(1) extra memory.

The order of elements can be changed. It doesn't matter what you leave beyond the
new length.

Example 1:
Given nums = [3,2,2,3], val = 3,
Your function should return length = 2, with the first two elements of nums being
2.
It doesn't matter what you leave beyond the returned length.

Example 2:
Given nums = [0,1,2,2,3,0,4,2], val = 2,
Your function should return length = 5, with the first five elements of nums
containing 0, 1, 3, 0, and 4.
Note that the order of those five elements can be arbitrary.
It doesn't matter what values are set beyond the returned length.
```

翻译：在同一个数组内删除等同于target value的元素，返回新的长度。

思路：以第一个元素为初始，遍历到第二个，对比二者是否相同，相同什么也不做，否则值覆盖并且下标加1。

```
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int pos = 0;
        for (int i=0; i<nums.size(); i++){
            if (nums[i] != val){
                nums[pos] = nums[i];
                pos += 1;
            }
        }
        return pos;
    }
};
```

13 Implement strStr()

```
Implement strStr().

Return the index of the first occurrence of needle in haystack, or -1 if needle
is not part of haystack.

Example 1:
Input: haystack = "hello", needle = "ll"
Output: 2

Example 2:
Input: haystack = "aaaaa", needle = "bba"
Output: -1

Clarification:
What should we return when needle is an empty string? This is a great question to
ask during an interview. For the purpose of this problem, we will return 0 when
needle is an empty string. This is consistent to C's strstr() and Java's
indexOf().
```

翻译：一个字符串中是否存在子串和给定的另一个字符串相等。如果相等，返回该子串首字母的索引。不存在就返回-1。

思路：暴力匹配模式

1. 内循环让小字符串中的每个元素与大字符串的第一个元素遍历，看是否相同。相同继续，不同则退出循环；
2. 内循环退出时要判断：内循环次数是否等于小字符串长度。相等则说明大字符串包含它；否则不含有它；
3. 如果不含有它，外循环遍历大字符串的第二个元素，然后再进入内循环同1-2的操作；

4. 内外循环都结束，如果没有return，则说明大字符串不含小的字符串，直接返回失败即可。

```
class ImplementStr {
 public:
  int strStr(string haystack, string needle) {
    const string &h = haystack;
    const string &n = needle;
    int h_size = h.size();
    int n_size = n.size();
    int i, j;
    for (i = 0; i <= h_size - n_size; ++i) {
      for (j = 0; j < n_size; ++j) {
        if (h[i + j] != n[j]) {
          break;
        }
      }
      if (j == n_size) {
        return i;
      }
    }
    return -1;
  }
};
```

14 Valid Sadoku

```
Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be
validated according to the following rules:

    Each row must contain the digits 1-9 without repetition.
    Each column must contain the digits 1-9 without repetition.
    Each of the 9 3x3 sub-boxes of the grid must contain the digits 1-9 without
repetition.

A partially filled sudoku which is valid.

The Sudoku board could be partially filled, where empty cells are filled with the
character '.'.

Example 1:
Input:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: true

Example 2:
```

```
Input:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner being
    modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is
invalid.

Note:
    A Sudoku board (partially filled) could be valid but is not necessarily
solvable.
    Only the filled cells need to be validated according to the mentioned rules.
    The given board contain only digits 1-9 and the character '.'.
    The given board size is always 9x9.
```

翻译：

```
判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
    数字 1-9 在每一行只能出现一次。
    数字 1-9 在每一列只能出现一次。
    数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
上图是一个部分填充的有效的数独。数独部分空格内已填入了数字，空白格用 '.' 表示。

示例 1:
输入:
[
  ["5","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
输出: true

示例 2:
输入:
[
  ["8","3",".",".","7",".",".",".","."],
  ["6",".",".","1","9","5",".",".","."],
  [".","9","8",".",".",".",".","6","."],
  ["8",".",".",".","6",".",".",".","3"],
  ["4",".",".","8",".","3",".",".","1"],
  ["7",".",".",".","2",".",".",".","6"],
  [".","6",".",".",".",".","2","8","."],
  [".",".",".","4","1","9",".",".","5"],
  [".",".",".",".","8",".",".","7","9"]
]
输出: false
解释: 除了第一行的第一个数字从 5 改为 8 以外，空格内其他数字均与 示例1 相同。
     但由于位于左上角的 3x3 宫内有两个 8 存在, 因此这个数独是无效的。
说明:
    一个有效的数独（部分已被填充）不一定是可解的。
    只需要根据以上规则，验证已经填入的数字是否有效即可。
    给定数独序列只包含数字 1-9 和字符 '.' 。
    给定数独永远是 9x9 形式的。
```

思路：

- 依次检查每行，每列，每个子九宫格是否出现重复元素，如果出现返回false，否则返回true；
- 难点在于表示第i个九宫格每个格点的坐标。

观察行号规律：

第0个九宫格：000111222; 第1个九宫格：000111222; 第2个九宫格：000111222;

第3个九宫格：333444555; 第4个九宫格：333444555; 第5个九宫格：333444555;

第6个九宫格：666777888; 第7个九宫格：666777888; 第8个九宫格：666777888;

可见对于每三个九宫格行号增3；对于单个九宫格，每三个格点行号增1。

因此第i个九宫格的第j个格点的行号可表示为i/3*3+j/3；

如何检查重复？

答：最大九宫格是9*9，初始设为false。利用isdigit检查是否有数字特征，第一次遍历有数字特征的设为true，其他格点计算的地址上对应的数字特征如果地址同设为true的相同，则说明重复，直接返回false。

```
class ValidSadoku {
public:
    bool valid_sadoku(vector<vector<char> > &board) { //board就是9*9
        const int cnt = 9;
        bool row_mask[cnt][cnt] = {false};
        bool col_mask[cnt][cnt] = {false};
        bool area_mask[cnt][cnt] = {false};
        //check each rows and cols
        printf("board.size: %d\n", (int)board.size());
        for(int r=0; r<board.size(); r++){
            for (int c=0; c<board[r].size(); c++){
                if (!isdigit(board[r][c])) continue;
                int idx =  board[r][c] - '0' - 1;  //计算处的数字特征减1代表地址

                //check the rows
                if (row_mask[r][idx] == true){ //字符有相同的数字转化为idx后会一样，从
而判断重复
                    return false;
                }
                row_mask[r][idx] = true;  /遍历第一个默认是false,这里设为true;下次利
用上面判断重复

                //check the cols
                if (col_mask[c][idx] == true) {
                    return false;
                }
                col_mask[c][idx] = true;

                //check the areas
                int area = (r/3) * 3 + (c/3); //格点坐标是难点，利用行号规律
                if (area_mask[area][idx] == true) {
                    return false;
                }
                area_mask[area][idx] = true;
            }
        }
        return true;
    }
};
```

15 Length of Last Word

```
Given a string s consists of upper/lower-case alphabets and empty space
characters ' ', return the length of last word (last word means the last
appearing word if we loop from left to right) in the string.

If the last word does not exist, return 0.Note: A word is defined as a maximal
substring consisting of non-space characters only.

Example:
Input: "Hello World"
Output: 5
```

翻译：计算字符串的长度，遇到空格则返回之前的长度。

思路：

- 先计算string型字符串的总长度n；
- 再判断边界，即是否有空字符，利用总长度减去空的长度就是剩余字符串长度i；
- 非空情况下，判断字符有几个（j），最后利用剩余字符串长度减去剩下的字符串(i-j)。

```
class LengthOfLastWord {
public:
    int length_of_lastWord(string s) {
        int n = s.size();
        int i = n - 1;
        while (i >= 0 && s[i] == ' ') {
            --i;  //i++也行，或者i=i+1;
        }
        int j = i;
        while (j >= 0 && s[j] != ' ') {
            --j; //j++也行，或者j=j+1;
        }
        return i - j; //最终非空的字符串长度
    }
};
```

注意：

- string型变量ss.size()返回的是包括'\0'的长度，所以真正的字符串长度为 `ss.size()-1`；

16 Valid Number

```
Validate if a given string can be interpreted as a decimal number.

Some examples:
"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
" -90e3   " => true
" 1e" => false
"e3" => false
" 6e-1" => true
" 99e2.5 " => false
"53.5e93" => true
" --6 " => false
"-+3" => false
"95a54e53" => false

Note: It is intended for the problem statement to be ambiguous. You should gather
all requirements up front before implementing one. However, here is a list of
characters that can be in a valid decimal number:
    Numbers 0-9
    Exponent - "e"
    Positive/negative sign - "+"/"-"
    Decimal point - "."
Of course, the context of these characters also matters in the input.
```

翻译：判断是否为有效的数字。

思路：

- 把input当做字符串，首先相邻s[i+1]的不能为空；
- 判断+/-号，符号前不能有指数或空格；
- 判断是否为数字；
- 判断.号，前后不能再有.号，相邻不能有指数；
- 判断指数，前后必须接数字，数字前可以接正负号；

```
class ValidNumber {
 public:
  bool valid_number(string s) {
    bool num = false, numAfterE = true, dot = false, exp = false, sign = false;
    int n = s.size();
    for (int i = 0; i < n; ++i) {
      if (s[i] == ' ') {
        if (i < n - 1 && s[i + 1] != ' ' && (num || dot || exp || sign)) {
          return false;
        }
      } else if (s[i] == '+' || s[i] == '-') {
        if (i > 0 && s[i - 1] != 'e' && s[i - 1] != ' ') return false;
        sign = true;
      } else if (s[i] >= '0' && s[i] <= '9') {
        num = true;
        numAfterE = true;
      } else if (s[i] == '.') {
        if (dot || exp) return false;
        dot = true;
      } else if (s[i] == 'e') {
        if (exp || !num) return false;
        exp = true;
        numAfterE = false;
      } else
        return false;
    }
    return num && numAfterE;
  }
};
```

17 Plus One

```
Given a non-empty array of digits representing a non-negative integer, plus one
to the integer.

The digits are stored such that the most significant digit is at the head of the
list, and each element in the array contain a single digit.

You may assume the integer does not contain any leading zero, except the number 0
itself.

Example 1:
Input: [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.

Example 2:
Input: [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
```

翻译：

```
给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。最高位数字存放在数组的首位，
数组中每个元素只存储单个数字。你可以假设除了整数 0 之外，这个整数不会以零开头。

示例 1:
输入: [1,2,3]
输出: [1,2,4]
解释: 输入数组表示数字 123。

示例 2:
输入: [4,3,2,1]
输出: [4,3,2,2]
解释: 输入数组表示数字 4321。
```

思路：末尾数字是9，则会有进位问题，而如果前面位上的数字仍为9，则需要继续向前进位。算法是

- 首先判断最后一位是否为9，若不是，直接加一返回，若是，则该位赋0，再继续查前一位；
- 同样的方法，直到查到第一位。如果第一位原本为9，加一后会产生新的一位，那么最后要做的是，查运算完的第一位是否为0，如果是，则在最前头加一个1。

```
class PlusOne {
 public:
  vector<int> plus_one(vector<int> &digits) {
    int n = digits.size();
    for (int i = n - 1; i >= 0; --i) {
      if (digits[i] == 9) {
        digits[i] = 0;
      } else {
        digits[i] += 1;
        return digits;
      }
    }
    if (digits.front() == 0) {
      digits.insert(digits.begin(), 1);
    }
    return digits;
  }
};
```

18 Add Binary

```
Given two binary strings, return their sum (also a binary string).The input
strings are both non-empty and contains only characters 1 or 0.

Example 1:
Input: a = "11", b = "1"
Output: "100"

Example 2:
Input: a = "1010", b = "1011"
Output: "10101"
```

翻译：字符串为二进制数，如何进行加减法。

思路：

- 先判断两字符串长度是否一样，少的要补齐'0'；
- 从末尾开始，两字符相加。结果为2时，进位carry赋值为1；结果为0,1时，carry赋值为0；
- 将第二步结果放在新的string变量里。

```
class AddBinary {
 public:
  string add_binary(string a, string b) {
    string res;
    int na = a.size();
    int nb = b.size();
    int n = max(na, nb);
    bool carry = false;
    if (na > nb) {
      for (int i = 0; i < na - nb; ++i) {
        b.insert(b.begin(), '0');
      }
    } else if (na < nb) {
      for (int i = 0; i < nb - na; ++i) {
        a.insert(a.begin(), '0');
      }
    }
    for (int i = n - 1; i >= 0; --i) {
      int tmp = 0;
      if (carry) {
        tmp = (a[i] - '0') + (b[i] - '0') + 1;
      } else {
        tmp = (a[i] - '0') + (b[i] - '0');
      }
      if (tmp == 0) {
        res.insert(res.begin(), '0');
        carry = false;
      } else if (tmp == 1) {
        res.insert(res.begin(), '1');
        carry = false;
      } else if (tmp == 2) {
        res.insert(res.begin(), '0');
        carry = true;
      } else if (tmp == 3) {
        res.insert(res.begin(), '1');
        carry = true;
      }
    }
    if (carry) {
      res.insert(res.begin(), '1');
    }
    return res;
  }
};
```

注意：

- insert用法
- string型字符串变量a或b用下标访问类似数组，但长度要注意，是a.size()-1才是真实下标。

19 Climb Stairs

```
Each time you can either climb 1 or 2 steps. In how many distinct ways can you
climb to the top? Note: Given n will be a positive integer.

Example 1:
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps

Example 2:
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

翻译：一次允许爬1到2步楼梯，有多少种办法爬到第n个阶梯。

思路：Fibonacci函数

> n 表示楼层　f(n) 表示 到达n层一共有多少种方法
> n=1 f(1)=1　[1]
> n=2 f(2)=2　[1 1] [2]
> n=3 f(3)=3　[1 1 1] [1 2] [2 1]
> n=4 f(4)=5　[1 1 1 1] [1 2 1] [2 1 1] [1 1 2] [2 2]
> 规律：　f(n) = f(n-1) + f(n-2)

```
class ClimbStairs {
 public:
  int climb_stairs(int n) {
    if (n < 0) {
      return 0;
    }
    int f1 = 0;  //关键
    int f2 = 1;  //关键
    int f3 = 1;  //关键
    int i;
    for (i = 0; i < n; ++i) {
      f3 = f1 + f2;
      f1 = f2;
      f2 = f3;
    }
    return f3;
  }
};
```

20 Remove Duplicates from Sorted List

```
Given a sorted linked list, delete all duplicates such that each element appear
only once.

Example 1:
Input: 1->1->2
Output: 1->2

Example 2:
Input: 1->1->2->3->3
Output: 1->2->3
```

翻译：将有序链表中重复出现的数据删掉。

思路：

- 先创建链表，结构体中包含int型数据和NULL指针域；
- 判断链表是否为空，空就不需要删除，直接返该链表；
- 非空就用得创建临时指针，将其指向要删除的节点。将删除节点指向的下一个节点的指针域赋给原来的链表；
- 删除临时节点。

```
struct ListNode {
  int val;
  ListNode* next;
  ListNode(int x) : val(x), next(nullptr) {}
};

class RemoveDuplicatesFromSortedList {
 public:
  ListNode* remove_duplicates_from_sorted_list(ListNode* head) {
    if (head == nullptr) {
      return head;
    }
    ListNode* ptr = head;
    ListNode* tmp;
    while (ptr->next != nullptr) {
      if (ptr->val == ptr->next->val) {
        tmp = ptr->next;
        ptr->next = tmp->next;
        delete tmp;
      } else {
        ptr = ptr->next;
      }
    }
    return head;
  }
};
```

21 Merge Sorted Array

```
Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one
sorted array.
Note:

    The number of elements initialized in nums1 and nums2 are m and n
respectively.
    You may assume that nums1 has enough space (size that is greater or equal to
m + n) to hold additional elements from nums2.

Example:
Input:
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6],       n = 3
Output: [1,2,2,3,5,6]
```

翻译：将两个有序数组合并，保存重复数字，合并后的数组仍然是有序的。

思路：维护index1, index2, index3，分别对应数组A，数组B，和结果数组。然后A和B同时从后往前遍历，每次遍历中A和B指向的元素大的放入结果数组中，然后该A和B的Index1和index2分别减1，index3不动。

```
class MergeSortedArray {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int i, j, k;
        i = m - 1;
        j = n - 1;
        k = m + n - 1;
        while (i >= 0 && j >= 0) {
            if (nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
        while (j >= 0) { //检查A和B谁多出来了，多的自动放过去
            nums1[k--] = nums2[j--];
        }
    }
};
```

注意：数组必须是有序的，而且是从小向大排列，才能用上面的方法。

22 Same Tree

```
Given two binary trees, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical and
the nodes have the same value.

Example 1:
Input:     1         1
          / \       / \
         2   3     2   3
        [1,2,3],   [1,2,3]

Output: true
Example 2:

Input:     1         1
          /           \
         2             2
        [1,2],     [1,null,2]

Output: false

Example 3:
Input:     1         1
          / \       / \
         2   1     1   2
        [1,2,1],   [1,1,2]

Output: false
```

思路：树要用到递归的思想。

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
class Solution {
 public:
  bool isSameTree(TreeNode *p, TreeNode *q) {
    if (p == NULL || q == NULL) {
      return p == q;
    }
    if (p->val != q->val) {
      return false;
    }
    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
  }
};
```

23 SymetricTree

```
For example, this binary tree [1,2,2,3,4,4,3] is symmetric:

    1
   / \
  2   2
 / \ / \
3  4 4  3

But the following [1,2,2,null,3,null,3] is not:
    1
   / \
  2   2
   \   \
   3    3
```

思路：用到递归思想。很类似Same Tree那道题，但是考虑到对称问题，叶子节点条件变为p->left == q->right.

```
struct TreeNode {
  int val;
  TreeNode* left;
  TreeNode* right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
 public:
  bool isSymmetric(TreeNode* root) {
    if (root == NULL) {
      return true;
    }
    return isMirror(root->left, root->right);
  }
 private:
  bool isMirror(TreeNode* root1, TreeNode* root2) {
    if (root1 == NULL) {  //左边树空
      return root2 == NULL;
    }
    if (root2 == NULL) {  //右边树空
      return root1 == NULL;
    }
    //两棵树都不空的情况
    return root1->val == root2->val && isMirror(root1->left, root2->right) &&
           isMirror(root1->right, root2->left);
  }
};
```

24 Binary Tree Level Order Transversal

```
Given a binary tree, return the level order traversal of its nodes' values. (ie,
from left to right, level by level).

For example: Given binary tree {3,9,20,#,#,15,7},
*      3
*     / \
*    9  20
*      /  \
*     15   7
* return its level order traversal as:
* [
*   [3],
*   [9,20],
*   [15,7]
* ]
if you confused what "{1,#,2,3}" means? > read more on how binary tree is
serialized on OJ.
OJ's Binary Tree Serialization: the serialization of a binary tree follows a
level order traversal, where '#' signifies a path terminator where no node exists
below. Here's an example:
*     1
*    / \
*   2   3
*      /
*     4
*      \
*       5
* The above binary tree is serialized as "{1,2,3,#,#,4,#,#,5}".
```

翻译：二叉树层序遍历。

思路：基本思路是将树放到queue中，利用层序遍历（BFS广度优先）。先建立一个 queue，先把根节点放进去，再找根节点的左右子节点，找到后去掉根节点。此时 queue 里的元素就是下一层的所有节点，用for 循环遍历它们，然后存到一个一维向量里，遍历完之后再把这个一维向量存到二维向量里，以此类推，可以完成层序遍历。

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
 public:
  vector<vector<int>> levelOrder(TreeNode *root) {
    vector<vector<int>> result;
    queue<TreeNode *> q;
    vector<int> level;  //每层结果
    int size, i;
    TreeNode *p;

    if (root == NULL) return result;
    q.push(root);  //入队
    while (!q.empty()) {
      //队列中有几个元素就依次遍历每个元素的左右结点
      level.clear();  //再次压入result时要先清空;
      size = q.size();
      for (i = 0; i < size; i++) {
        p = q.front();  //队首元素值赋给p
        q.pop();        //出队
        level.push_back(p->val);
        if (p->left) {  //依次压入左右结点元素
          q.push(p->left);
        }
        if (p->right) {
          q.push(p->right);
        }
      }
      result.push_back(level);  //添加每层数据
    }
    return result;
  }
};
```

注意：stack和queue的区别，特别是.front()函数的调用，stack返回的是最后入栈的，而queue返回的是队列中的首元素。

25 Maximum Depth of Binary Tree

```
Given a binary tree, find its maximum depth.
The maximum depth is the number of nodes along the longest path from the root
node down to the farthest leaf node. Note: A leaf is a node with no children.

Example:
Given binary tree [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
return its depth = 3.
```

翻译：求二叉树的最大深度

思路：递归思想。

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
 public:
  int maxDepth(TreeNode *root) {
    if (root == NULL) {
      return 0;
    }
    if (!root->left && !root->right) {
      return 1;
    }
    int left = 1, right = 1;
    if (root->left) {
      left += maxDepth(root->left);
    }
    if (root->right) {
      right += maxDepth(root->right);
    }
    return left > right ? left : right;
  }
};
```

26 Binary Tree Level Order Transversal II

```
Given a binary tree, return the bottom-up level order traversal of its nodes'
values. (ie, from left to right, level by level from leaf to root).

For example:
Given binary tree [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7

return its bottom-up level order traversal as:
[
  [15,7],
  [9,20],
  [3]
]
```

翻译：层序遍历，遍历后要倒着写，和24题很想。

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
class BinaryTreeLevelOrderTransversal {
public:
    vector<vector<int> > binary_tree_level_order_transversal_II(TreeNode* root)
{
        if (!root) return {};
        vector<vector<int>> res;
        queue<TreeNode*> q{{root}};
        while (!q.empty()) {
            vector<int> oneLevel;
            for (int i = q.size(); i > 0; --i) {
                TreeNode *t = q.front(); q.pop();
                oneLevel.push_back(t->val);
                if (t->left) q.push(t->left);
                if (t->right) q.push(t->right);
            }
            res.insert(res.begin(), oneLevel);
        }
        return res;
    }
};
```

27 Balanced Binary Tree

```
Given a binary tree, determine if it is height-balanced. For this problem, a
height-balanced binary tree is defined as: a binary tree in which the left and
right subtrees of every node differ in height by no more than 1.

Example 1:
Given the following tree [3,9,20,null,null,15,7]:
    3
   / \
  9  20
    /  \
   15   7
Return true.

Example 2:
Given the following tree [1,2,2,3,3,null,null,4,4]:
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
Return false.
```

翻译：子树的高度差不能超过1，否则不是完全二叉树。

思路：递归思想

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
class BalancedBinaryTree {
public:
    bool isBalanced(TreeNode *root) {
        if (!root) return true;
        if (abs(getDepth(root->left) - getDepth(root->right)) > 1) return false;
        return isBalanced(root->left) && isBalanced(root->right);
    }
    int getDepth(TreeNode *root) {
        if (!root) return 0;
        return 1 + max(getDepth(root->left), getDepth(root->right));
    }
};
```

30 Minimum Depth of Tree

```
Given a binary tree, find its minimum depth. The minimum depth is the number of
nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children.

Example:
Given binary tree [3,9,20,null,null,15,7],
    3
   / \
  9  20
    /  \
   15   7
return its minimum depth = 2.
```

翻译：求二叉树的最小深度，**同25题很类似**。

思路：递归思想，从根节点开始，找到距离左右叶子的高度，返回较小的那个高度即可。

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class MaximumDepthOfBinaryTree {
 public:
  int maximum_depth_of_binary_tree(TreeNode *root) {
    if (root == NULL) {
      return 0;
    }
    if (!root->left && !root->right) {
      return 1;
    }
    int left = 1, right = 1;
    if (root->left) {
      left += maximum_depth_of_binary_tree(root->left);
    }
    if (root->right) {
      right += maximum_depth_of_binary_tree(root->right);
    }
    return left > right ? right : left;
  }
};
```

31 Path Sum

```
Given a binary tree and a sum, determine if the tree has a root-to-leaf path such
that adding up all the values along the path equals the given sum. Note: A leaf
is a node with no children.

Example:
Given the below binary tree and sum = 22,
      5
     / \
    4   8
   /   / \
  11  13  4
 /  \      \
7    2      1
return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.
```

翻译：在二叉树上找是否等于sum的path。

思路：递归

```
struct TreeNode {
  int val;
  TreeNode *left;
  TreeNode *right;
  TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
class PathSum {
public:
    bool path_sum(TreeNode* root, int sum) {
        if (root == NULL) {
            return false;
        }
        return has_path(root, sum);
    }
private:
    bool has_path(TreeNode* root, int sum) {
        if (root->left == NULL && root->right == NULL) {
            return root->val == sum;
        }
        bool res = false;
        if (root->left != NULL) {
            res = res || has_path(root->left, sum - root->val);
        }
        if (root->right != NULL) {
            res = res || has_path(root->right, sum - root->val);
        }
        return res;
    }
};
```

32 Pascal's Triangle

```
Given a non-negative integer numRows, generate the first numRows of Pascal's
triangle. In Pascal's triangle, each number is the sum of the two numbers
directly above it.

Example:
Input: 5
Output:
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

翻译：帕斯卡三角形。

思路：规律是某层第i个元素的值等于它上一层第(i-1)和第i个元素的值的和，所以按照定义做即可。

```
class PascalTriangle {
  vector<vector<int> > pascal_triangle(int numRows) {
    vector<vector<int> > result;
    for (int i = 0; i < numRows; i++) {
      vector<int> v;
      if (i == 0) {
        v.push_back(1);
      } else {
        v.push_back(1);
        for (int j = 0; j < result[i - 1].size() - 1; j++) {  //注意这个下标很要多减去1到上一行
          v.push_back(result[i - 1][j] + result[i - 1][j + 1]); //.size()-1到几列可以这样做
        }
        v.push_back(1);
      }
      result.push_back(v);
    }
    return result;
  }
};
```

33 Min Stack

```
Design a stack that supports push, pop, top, and retrieving the minimum element
in constant time.
    push(x) -- Push element x onto stack.
    pop() -- Removes the element on top of the stack.
    top() -- Get the top element.
    getMin() -- Retrieve the minimum element in the stack.

Example:

MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> Returns -3.
minStack.pop();
minStack.top();      --> Returns 0.
minStack.getMin();   --> Returns -2.
```

翻译：实现一个最小栈，快速取出最小值。

思路：考虑栈的操作不能遍历，且后进先出，所以要想快速得到最小值，需要要额外设置个tmp。如果压栈的值小于tmp，则把它作为min，直接return。

```
class MinStack {
public:
    /** initialize your data structure here. */
    MinStack() {};
    void push(int x) {
        if (minstack.empty())
        {
            minstack.push(x);
            min = x;
        }
        else {
            if (x<=min)//此处必须为<=,否则可以试一下{0，1，0}的情况
            {
                minstack.push(min);
                min = x;
            }
            minstack.push(x);
        }
    }

    void pop() {
        if (minstack.empty())return;
        int index = this->top();
        minstack.pop();
        if (min == index && !minstack.empty())
        {
            min = this->top();
            minstack.pop();
        }
    }

    int top() {
        return minstack.top();
    }

    int getMin() {
        return min;
    }
private:
    int min;
    stack<int>minstack;
};
```

34 Intersection of Two Linked Lists*

35 Compare Version Numbers

```
Compare two version numbers version1 and version2. If version1 > version2 return
1; if version1 < version2 return -1;otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and
the . character.

The . character does not represent a decimal point and is used to separate number
sequences.

For instance, 2.5 is not "two and a half" or "half way to version three", it is
the fifth second-level revision of the second first-level revision.

Example 1:
Input: version1 = "0.1", version2 = "1.1"
Output: -1

Example 2:
Input: version1 = "1.0.1", version2 = "1"
Output: 1

Example 3:
Input: version1 = "7.5.2.4", version2 = "7.5.3"
Output: -1

Example 4:
Input: version1 = "1.01", version2 = "1.001"
Output: 0
Explanation: Ignoring leading zeroes, both “01” and “001" represent the same
number “1”

Example 5:
Input: version1 = "1.0", version2 = "1.0.0"
Output: 0
Explanation: The first version number does not have a third level revision
number, which means its third level revision number is default to "0"
```

翻译：字符串中的数字代表版本，试对比其大小。

思路：利用atoi和c_str把字符串的每个数字提取出来，不要.号。一般分两种情况：

- 1.1和1.2，提取小数点前的数字先对比，分别是“1”和"1"，利用atoi(xx.c_str)转换为整形数字。如果相同，比较小数点后面的数字；
- 1.01和1.001，对比完小数点前的数字后，继续读取小数点后的字符串，"01"和"001"分别用atoi(xx.c_str)形式转换，结果为1和1，对比就相等。
- -1.11不可能，因为版本号必须为non-negative数字，可以包括0或0.xx等。

```
class CompareVersionNumber {
public:
    int compare_version_number(string version1, string version2) {
        int n1 = version1.size(), n2 = version2.size();
        int i = 0, j = 0, d1 = 0, d2 = 0;
        string v1, v2;
        while (i < n1 || j < n2) {
            while (i < n1 && version1[i] != '.') {
                v1.push_back(version1[i++]);
            }
            d1 = atoi(v1.c_str());
            while (j < n2 && version2[j] != '.') {
                v2.push_back(version2[j++]);
            }
            d2 = atoi(v2.c_str());
            if (d1 > d2) return 1;
            else if (d1 < d2) return -1;
            v1.clear(); v2.clear();
            ++i; ++j;
        }
        return 0;
    }
};
```

36 Excel Sheet Column Number

```
Given a positive integer, return its corresponding column title as appear in an
Excel sheet.
For example:
    1 -> A
    2 -> B
    3 -> C
    ...
    26 -> Z
    27 -> AA
    28 -> AB
    ...
Example 1:
Input: 1
Output: "A"

Example 2:
Input: 28
Output: "AB"

Example 3:
Input: 701
Output: "ZY"
```

翻译：类似Excel表格，A对应第1列；B对应第2列；...；Z对应第26列；AB对应第27列...等。

思路：规律A->Z是1到26，AA->AZ是1+26到26+26，就是加26而已。

```
class ExcelSheetColumnNumber {
 public:
  int excel_sheet_column_number(string s) {
    int result = 0;
    int len = s.size();
    for (int i = 0; i < len; i++) {
      result = result * 26 + s[i] - 'A' + 1;
    }
    return result;
  }
};
```

注意：

- C语言和C++关于字符和字符串数组定义的区别，以及如何使用字符串指针定义和打印字符串数组
- 在C++里，string型对象使用size()和length()求字符串长度的值是一样的，遇到空格不会返回。

37 Majority Element

```
Given an array of size n, find the majority element. The majority element is the
element that appears more than [ n/2 ] times.

You may assume that the array is non-empty and the majority element always exist
in the array.

Example 1:
Input: [3,2,3]
Output: 3

Example 2:
Input: [2,2,1,1,1,2,2]
Output: 2
```

翻译：数组中重复出现最多的元素的次数是否大于 n/2，如果是，返回该元素。

思路：

方法1：因为Array is non-empty并且存在于Array中，可以考虑暴力匹配，找到出现次数最多元素后，再判断它的次数是否大于n/2。缺点是时间按复杂度是O(n^2)，空间复杂度O(n);

方法2：

第一步：现在数组中设置arr[0]，其对应的count=1。接着让第二个元素和其对比，相同count+1，不同count-1，当count<=0时；arr[0]换成arr[i]。接着让后续元素和arr[i]对比，对比完后，可以找到出现次数最多的candidate;

第二步：遍历整个数组，让每个元素和candidate对比，相同cnt+1；最后看cnt > n/2是否成立。不成立返回-1，成立返回candidate。

```
class MajorityElement {
 public:
  int majority_element(vector<int> nums) {
    int count = 1;
    int index = 0;
    for (int i = 1; i < nums.size(); i++) {  //遍历找到candidate
      if (nums[i] == nums[index]) {
        count++;
      } else {
        count--;
      }
      if (count <= 0) {
        count = 1;
        index = i;
      }
    }

    count = 0;  //清零，计算candidate出现的次数；如果大于n/2则返回candiate，否则返回-1
    for (int i = 0; i < nums.size(); i++) {
      if (nums[i] == nums[index]) {
        count++;
      }
    }
    if (count > nums.size() / 2) {
      return nums[index];
    }
    return -1;
  }
};
```

40 Factorial Trailing Zeroes

```
Given an integer n, return the number of trailing zeroes in n!.

Example 1:
Input: 3
Output: 0
Explanation: 3! = 6, no trailing zero.

Example 2:
Input: 5
Output: 1
Explanation: 5! = 120, one trailing zero.
Note: Your solution should be in logarithmic time complexity.
```

翻译：给出一个正整数n，计算n!结构后面有几个0.要求：在多项式时间中完成算法。

常规思路：计算n!，然后统计一下后面有几个0，但是这种算法一想就知道肯定会超出时间限制。

巧妙思路：相乘得0，则只有2*5相乘能得到0；而0的个数也只会与2、5的个数有关，且2的幂方数一定比5的幂方数多；

2×5=10；

2×5×2×5=100；

即有几套2和5，则会有几个零。

41 Rotate Array

```
Given an array, rotate the array to the right by k steps, where k is non-
negative.
```

```
Example 1:
Input: [1,2,3,4,5,6,7] and k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]

Example 2:
Input: [-1,-100,3,99] and k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]

Note:Try to come up as many solutions as you can, there are at least 3 different
ways to solve this problem.Could you do it in-place with O(1) extra space?
```

翻译：给定一个数组arr，给定一个正整数k，将arr中的元素依次向右移动k个位置，超边界的放到数组起始位置行程闭环。

思路：新开一个数组，遍历老数组，并老数组的元素按照指定的k个位置取出放到新的数组中去。然后将新数组再放回老数组，算法空间即为O(1)。新开数组是为了保存元素，以防覆盖。难点在于超出边界的下标在新数组中如何处理，算法是 (i+k) % arr_len。

```
class RotateArray {
 public:
  void rotate_array(int nums[], int m, int k) {
    int a[m] = {0};
    int len = sizeof(a)/sizeof(a[0]);
    for (int i = 0; i < m; i++) {
      a[(i + k) % m] = nums[i];
    }
    for (int i = 0; i < m; i++) {
      nums[i] = a[i];
    }
  }
};
```

42 Reverse Bits

```
Reverse bits of a given 32 bits unsigned integer.

Example 1:
Input: 00000010100101000001111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string 00000010100101000001111010011100 represents
the unsigned integer 43261596, so return 964176192 which its binary
representation is 00111001011110000010100101000000.

Example 2:
Input: 11111111111111111111111111111101
Output: 10111111111111111111111111111111
Explanation: The input binary string 11111111111111111111111111111101 represents
the unsigned integer 4294967293, so return 3221225471 which its binary
representation is 10111111111111111111111111111111.
```

翻译：将二进制逆序。

思路：将二进制的每一位与1做和运算，完了该数向右每次移动一位。一共做32次。

- 创建一个sum，将二进制数每一位与1做和（&）运算，如果相同，则 sum = (sum << 1) + 1；佛则 sum = sum << 1 + 0;
- 做完后将原来的二进制数超右移动一位;
- 最后返回 sum即可。

```
class ReverseBits {
 public:
  uint32_t reverse_bits(uint32_t n) {
    uint32_t res = 0;
    for (int i = 0; i < 32; ++i) {
      if (n & 1 == 1) {
        res = (res << 1) + 1;
      } else {
        res = res << 1;
      }
      n = n >> 1;
    }
    return res;
  }
};
```

43 Number of 1 Bits

```
Write a function that takes an unsigned integer and return the number of '1' bits
it has (also known as the Hamming weight).

Example 1:
Input: 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total
of three '1' bits.

Example 2:
Input: 00000000000000000000000010000000
Output: 1
```

```
Explanation: The input binary string 00000000000000000000000010000000 has a total
of one '1' bit.

Example 3:
Input: 11111111111111111111111111111101
Output: 31
Explanation: The input binary string 11111111111111111111111111111101 has a total
of thirty one '1' bits.
```

翻译：计算二进制数中1的个数。

思路：同42题很类似，每个位与1求和运算，然后数字向右移动一位，知道32位移动完。

```
class NumberOf1Bits {
 public:
  uint32_t number_of_1bits(uint32_t n) {
    uint32_t res = 0;
    for (int i = 0; i < 32; ++i) {
      if (n & 1 == 1) {
        res = res + 1;
      }
      n = n >> 1;
    }
    return res;
  }
};
```

44 House Robber

```
You are a professional robber planning to rob houses along a street. Each house
has a certain amount of money stashed, the only constraint stopping you from
robbing each of them is that adjacent houses have security system connected and
it will automatically contact the police if two adjacent houses were broken into
on the same night.

Given a list of non-negative integers representing the amount of money of each
house, determine the maximum amount of money you can rob tonight without alerting
the police.

Example 1:
Input: [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
             Total amount you can rob = 1 + 3 = 4.

Example 2:
Input: [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5
(money = 1).
             Total amount you can rob = 2 + 9 + 1 = 12.
```

翻译：盗贼计划沿着一条街打家劫舍。每间房屋都储存有一定数量的金钱，唯一能阻止你打劫的约束条件就是：由于房屋之间有安全系统相连，如果同一个晚上有两间相邻的房屋被闯入，它们就会自动联络警察，因此不可以打劫相邻的房屋。给定一列非负整数，代表每间房屋的金钱数，计算出在不惊动警察的前提下一晚上最多可以打劫到的金钱数。

思路：利用到动态规划的思想 maxV[i] = max(maxV[i - 2] + a[i],maxV[i-1])；其中数组a[i]为第i个房间隐藏的金钱。maxV[i]表示前i个房间能获得的最多的钱。

方法1：注意指针的用法。

```
class HouseRobber {
 public:
  int house_robber(vector<int>& nums) {
    //处理特殊情况
    if (nums.empty()) return 0;
    if (nums.size() == 1) return nums[0];
    if (nums.size() == 2) return nums[0] > nums[1] ? nums[0] : nums[1];
    //处理正常情况
    int* maxV = new int[nums.size()];
    maxV[0] = nums[0];
    maxV[1] = nums[0] > nums[1] ? nums[0] : nums[1];
    for (int i = 2; i < nums.size(); ++i) {
      maxV[i] = max(maxV[i - 2] + nums[i], maxV[i - 1]);
    }
    int result = maxV[nums.size() - 1];
    delete maxV;
    maxV = NULL;
    return result;
  }
};
```

方法2：巧解。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        int f1, f2, f3;
        f1 = f2 = f3 = 0;
        int  i;
        for (i = 0; i < n; ++i) {
            f3 = max(f1 + nums[i], f2);
            f1 = f2;
            f2 = f3;
        }
        return f3;
    }
};
```

45 Happy Number

```
Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any
positive integer, replace the number by the sum of the squares of its digits, and
repeat the process until the number equals 1 (where it will stay), or it loops
endlessly in a cycle which does not include 1. Those numbers for which this
process ends in 1 are happy numbers.

Example:
Input: 19
Output: true
Explanation:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1
```

翻译：假设一个数每个位上的平方和为1，则该数就是快乐数。试判断一个数是否为快乐数。

以11为例子，4重复出现且不为1

```
1^2+1^2 = 2;
2^2 = 4
4^2 = 16
1^2 + 6^2 = 37
3^2 + 7^2 = 58
5^2 + 8^2 = 89
8^2 + 9^2 = 145
1^2 + 4^2 + 5^2 = 42
4^2 + 2^2 = 20
2^2 + 0^2 = 4
```

思路：快乐数有个特点，每个位上的平方和相加得到的最终会重复为1。如果有重复的且结果不为1，则是非快乐数。所以这里要利用unordered_set的查找性质。一旦发现重复的且不为1，结果必定为非快乐数。

```
class HappyNumber {
public:
    bool happy_number(int n) {
        unordered_set<int> v;
        while (n != 1) {
            int sum = 0;
            while (n) {
                sum += (n % 10) * (n % 10);
                n /= 10;
            }
            n = sum;
            if (v.count(n)) break;  //可以换成 if(v.find(n)!=v.end()) break;
            v.insert(n);
        }
        return n == 1;
    }
};
```

46 Remove Linked List Element

```
Remove all elements from a linked list of integers that have value val.

Example:
Input:  1->2->6->3->4->5->6, val = 6
Output: 1->2->3->4->5
```

翻译：删除单链表中重复出现的元素和其对应节点。

思路：考察的是链表的删除和插入操作。

- 链表如何定义和初始化，最好有首节点（-1），不同于头结点（val）;
- 让dummy指针指向首节点的next，遍历，等同于val值的节点，让其指针域直接指向下一个。
- 返回head。

```
struct ListNode{
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(NULL){}
};

ListNode* remove_linked_list_elements(ListNode* head){
    staic ListNode* dummy = new ListNode(-1); //初始化指针dummy，类型为ListNode*，
指向ListNode
    dummy.next = head;
    while(dummy->next){
        if(dummy->next->val == val){
            dummy->next = dummy->next->next;
        }else{
            dummy = dummy->next;
        }
    }
    return dummy->next;
}
```

有个问题：如果是删除呢？

答：类似20题。

```
ListNode* remove_linked_list_elements(ListNode* head){
    staic ListNode* dummy = new ListNode(-1); //初始化指针dummy，类型为ListNode*，
指向ListNode
    dummy.next = head;
    ListNode* p1;
    while(dummy->next){
        if(dummy->next->val == val){
            p1 = dummy->next;
            dummy = dummy->next->next;
            delete p1;
        }else{
            dummy = dumppy->next;
        }
    }
    return dummy->next;
}
```

上述代码有个问题，如果head第一个节点的指针域是NULL怎么办？

```
class Solution {
public:
    ListNode* remove_linked_list_elements(ListNode* head, int val) {
        ListNode *p1;
        //如果只有一个有效节点，找到它，然后删除哪个节点。也可以不删除，就成了野指针
        while (head != NULL && head->val == val) {
            p1 = head;
            head = head->next;
            delete p1;
        }
        if (head == NULL) {
            return head;
        }
        //如果是好几个节点,操作同上
        p1 = head;
        ListNode* p2;
        while (p1->next != NULL) {
            if (p1->next->val == val) {
                p2 = p1->next;
                p1->next = p2->next;
                delete p2;
            } else {
                p1 = p1->next;
            }
        }
        return head;
    }
};
```

47 Count Primes

```
Count the number of prime numbers less than a non-negative number, n.

Example:
Input: 10
Output: 4
Explanation: There are 4 prime numbers less than 10, they are 2, 3, 5, 7.
```

翻译：给定一个正整数n，统计[1,n]的素数个数。

思路：1不是素数，从2开始，3,5,7是素数，所以给定n后，创建一个数组遍历，如果后续数是2的倍数，例如2,3,4,..,n倍，将这些数标记为false，否则为true。完了设置个count，再遍历下的数组，统计true的个数即可。

```
class CountPrimes {
 public:
  int count_primes(int n) {
    vector<bool> v(n, true); //n个元素，值都是true
    for (int i = 2; i < n; i++) {  //从第二个开始遍历
      for (int j = i * 2; j < n; j = j + i) {  // 将i的2倍、3倍、4倍...都标记为非素数
        v[j] = false;
      }
    }
    int count = 0;
    for (int i = 2; i < n; i++) {
      if (v[i]) count++;
    }
    return count;
  }
};
```

注意：难点在于 i 从几开始。

38 Reverse Linked List

```
Reverse a singly linked list.

Example:
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL
```

翻译：链表逆序表示。

思路：

```
class Solution {
public:
    ListNode* reverseList_iteratively(ListNode* head) {
        ListNode *h=NULL, *p=NULL;
        while (head){
            p = head->next;
            head->next = h;
            h = head;
            head = p;
        }
        return h;
    }
    ListNode* reverseList_recursively(ListNode* head) {
        if (head==NULL || head->next==NULL) return head;
        ListNode *h = reverseList_recursively(head->next);
        head->next->next = head;
        head->next = NULL;
        return h;

    }
    ListNode* reverseList(ListNode* head) {
        return reverseList_iteratively(head);
        return reverseList_recursively(head);
    }
};
```

Part B：牛客网篇

参考：

[1] 腾讯+网易+爱奇艺+招商银行+校招统一算法笔试：https://github.com/shiqitao/NowCoder-Solutions；

[2] 华为：https://github.com/fingthinking/nowcoder/tree/master/huawei；

1 字符串

1.1 计算字符串中最后一个单词的长度

```
题目描述
    计算字符串最后一个单词的长度,单词以空格隔开
输入描述
    一行字符串,非空,长度小于5000
输入描述
    整数N,最后一个单词的长度
输入例子
    hello world
输出例子
    5
```

思路：

- 字符串 `rfind()` 函数应用，先找到空格，因为空格属于字符串的一部分；
- 判断空格的位置pos（0,1,2,..,n）和字符串总长度len（1,2,..,n, 包括空格，但不包括'\0'）的关系；
- 如果pos > len，直接返回pos；否则 返回 len - pos - 1 （减1因为pos从0开始的）.

```
string str;
getline(cin, str);  // 为了输入到string中包含空格,必须使用getline方法
long len = str.length();
long pos = str.rfind(' ');
if(len < 1){  //处理边界
    printf("Error: empty string!\n");
    return 0;
}
if (pos > len) {
    cout << len;
} else {
    cout << len - pos - 1;
}
```

1.2 字符串中目标字符出现的次数

```
题目描述
    写出一个程序，接受一个有字母和数字以及空格组成的字符串，和一个字符，然后输出输入字符串中含
有该字符的个数。不区分大小写。
输入描述
    输入一个有字母和数字以及空格组成的字符串，和一个字符。
输出描述
    输出输入字符串中含有该字符的个数
输入例子
    A B C D E
    A
    1
```

思路：遍历字符串中的每个字符，然后与target对比，有则计数器加1. 难点在于不区分大小写，所以要注意字符串的元素是否加减 `STEP = 'A'-'a'`, `STEP = 65-97 = -32` 大小写转换的步长。

```
int count = 0;
while (getline(cin, str)) {
    cin >> target;
    const int len = str.length();
    for (int i = 0; i < len; i++) {
        char ch = str[i];
        if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')) {
            if (ch == target || ((ch + step) == target) ||
                ((ch - step) == target)) {
            count++;
            }
        }
    }
    cout << count << endl;
    break;
}
```

1.3 字符串分割

```
题目描述
    连续输入字符串，请按长度为8拆分每个字符串后输出到新的字符串数组；
    长度不是8整数倍的字符串请在后面补数字0，空字符串不处理。

输入描述
    连续输入字符串(输入2次,每个字符串长度小于100)
输出描述
    输出到长度为8的新字符串数组

输入例子
abc
123456789
输出例子
abc00000
12345678
90000000
```

`substr（start, length）`方法：返回一个从指定下标（0,1,2..n）开始，并具有指定长度的子字符串

思路：

1. 先判断字符串长度是否小于8，如果是则补0；
2. 如果等于8，直接输出字符串；
3. 如果大于8，需要使用`substr`函数，找到被8整除后的字符串。

```
  string str;
  while (getline(cin, str)) {
    int len = str.size();
    if (len < 8) {  //字符串小于8
      for (int i = 1; i <= (8 - len); i++) {
        str = str + '0';
      }
      cout << str << endl;
      continue;
    }
    if (len == 8) { //字符串等于8
      cout << str << endl;
      continue;
    } else if (len > 8) { //字符串大于8
      int start = 0;
      while (len >= 8) {
        cout << "---" << str.substr(start, 8) << endl;
        start += 8;
        len -= 8;
      }
      if (len > 0) {
        string tmp = str.substr(start, len);
        for (int i = 1; i <= 8 - len; ++i) {
          tmp = tmp + '0';
        }
        cout << tmp << endl;
      }
    }
  }
```

2 数字

2.1 明明的随机数查重和删除

```
题目描述
    明明想在学校中请一些同学一起做一项问卷调查，为了实验的客观性，他先用计算机生成了N个1到1000之间的随机整数（N≤1000），对于其中重复的数字，只保留一个，把其余相同的数去掉，不同的数对应着不同的学生的学号。然后再把这些数从小到大排序，按照排好的顺序去找同学做调查。请你协助明明完成“去重”与“排序”的工作。
Input Param
    n 输入随机数的个数
    inputArray n个随机整数组成的数组

    Return Value
    OutputArray 输出处理后的随机整数
注：测试用例保证输入参数的正确性，答题者无需验证。测试用例不止一组。
输入描述
    输入多行，先输入随机整数的个数，再输入相应个数的整数
输出描述
    返回多行，处理后的结果
输入例子
11
10
20
40
32
67
40
20
89
300
400
15

输出例子
10
```

```
15
20
32
40
67
89
300
400
```

思路：

1. 输入要多行，考虑用`cin`；
2. 要先输入数组的长度，然后输入数组的元素，考虑`for循环+cin`；
3. 利用arr[num]同一位置反复输入自动去重的功能，但要将arr[num]对应的值打标记；
4. 由于数组自动排序，所以大衣呢不为0（没做标记的）的arr[num]即可。

```
int n = -1;
int num = 0;
const int N = 1000;
int arr[1000] = {0};
while(cin >> n){
   if(n<0){
      exit(0);
   }
   for(int i = 0; i < n; i++){
      cin >> num;
      if(arr[num] == 0){   //数组反复输入去重
         arr[num]++;       //打标记
      }
   }
   for(int i = 0; i < N; i++){
      if(arr[i] != 0){
         cout << "--: " << i << endl;
      }
   }
}
```