字符串

LeetCode 38. 外观数列

38. 外观数列

难度 简单 411

「外观数列」是一个整数序列，从数字 1 开始，序列中的每一项都是对前一项的描述。前五项如下：

```
1.     1
2.     11
3.     21
4.     1211
5.     111221
```

1 被读作 "one 1"（"一个一"），即 11。
11 被读作 "two 1s"（"两个一"），即 21。
21 被读作 "one 2"，"one 1"（"一个二"，"一个一"），即 1211。

给定一个正整数 *n*（1 ≤ *n* ≤ 30），输出外观数列的第 *n* 项。

注意：整数序列中的每一项将表示为一个字符串。

示例 1:

```
输入：1
输出："1"
解释：这是一个基本样例。
```

示例 2:

```
输入：4
输出："1211"
解释：当 n = 3 时，序列是 "21"，其中我们有 "2" 和 "1" 两组，"2" 可以读作 "12"，也就是出现频次 = 1 而 值 = 2；类似 "1" 可以读作 "11"。所以答案是 "12" 和 "11" 组合在一起，也就是 "1211"。
```

```cpp
/*模拟题
把上一行连续段读出来

*/
class Solution {
public:
    string countAndSay(int n) {
        string res = "1";
```

```
        while (-- n)
        {
            string newRes = "";
            for (int i = 0; i < res.size();)
            {
                int j = i;
                while (j < res.size() && res[i] == res[j]) j ++ ;
                newRes += to_string(j - i) + res[i];
                i = j;
            }
            res = newRes;
        }
        return res;
    }
};
**************************************************

class Solution {
public:
    string countAndSay(int n) {
        string res = "1";
        for(int i = 0;i < n - 1;i ++)
        {
            string ns;
            for(int j = 0;j < res.size();)
            {
                int k = j;
                while(k < res.size() && res[k] == res[j]) k ++;//找到连续的最大长度
                ns += to_string(k - j) + res[j];
                j = k;//要更新j的大小，跳过相同的字符，找下一个字符！！！
            }
            res = ns;//一直进行循环n次，上一个的res是下一个的ns
        }
        return res
    }
};
```

LeetCode 49. 字母异位词分组

49. 字母异位词分组

难度 中等 282

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

```
输入: ["eat", "tea", "tan", "ate", "nat", "bat"],
输出:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

说明:

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

算法

(哈希表) $O(NLlogL)$

定义从 `string` 映射到 `vector<string>` 的哈希表： `unordered_map<string, vector<string>>` ，哈希表用法参见 这里 。

我们将每个字符串的所有字符从小到大排序，将排好序的字符串作为key，然后将原字符串插入key对应的 `vector<string>` 中。

时间复杂度分析：N 是字符串个数，L 是字符串平均长度。对于每个字符串，哈希表和vector的插入操作复杂度都是 $O(1)$，排序复杂度是 $O(LlogL)$。所以总时间复杂度是 $O(NLlogL)$。

```
/*
字符串分组，字母相同，顺序可以不同
eat、tea、ate是一组
tan，nat是一组
bat是一组

用排序加hash

*/
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> hash;//第一个string是原来的字符串，第二个string是排完序的关键字。
        for (auto &str : strs)
        {
            string key = str;//备份
            sort(key.begin(), key.end());
            hash[key].push_back(str);
        }
```

```
        vector<vector<string>> res;
        for (auto item : hash) res.push_back(item.second);//映射的值。

        return res;
    }
};
```

LeetCode 151.翻转字符串里的单词

151. 翻转字符串里的单词

难度 中等 108

给定一个字符串，逐个翻转字符串中的每个单词。

示例 1:

```
输入: "the sky is blue"
输出: "blue is sky the"
```

示例 2:

```
输入: "  hello world!  "
输出: "world! hello"
解释: 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
```

示例 3:

```
输入: "a good   example"
输出: "example good a"
解释: 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。
```

说明:

- 无空格字符构成一个单词。
- 输入字符串可以在前面或者后面包含多余的空格，但是反转后的字符不能包括。
- 如果两个单词间有多余的空格，将反转后单词间的空格减少到只含一个。

算法

(数组翻转) $O(n)$

分两步操作：

1. 将字符串中的每个单词逆序，样例输入变为：`"eht yks si eulb"`；
2. 将整个字符串逆序，样例输入变为：`"blue is sky the"`；

时间复杂度分析：整个字符串总共扫描两遍，所以时间复杂度是 $O(n)$。且每次翻转一个字符串时，可以用两个指针分别从两端往中间扫描，每次交换两个指针对应的字符，所以额外空间的复杂度是 $O(1)$。

```
/*
按照单词进行反转
翻转两次
1.按单词翻转。
2.翻转整个字符串。

连续不等于空格的字符串是单词
reverse是左闭右开的！！！
*/
class Solution {
public:
    string reverseWords(string s) {
        int k = 0;
        for (int i = 0; i < s.size();)
        {
            int j = i;
            while (j < s.size() && s[j] == ' ') j ++ ;//连续不等于空格的字符串是单词
            if (j == s.size()) break;
            i = j;
            while (j < s.size() && s[j] != ' ') j ++ ;
            reverse(s.begin() + i, s.begin() + j);//第一次翻转
            if (k) s[k ++ ] = ' ';
            while (i < j) s[k ++ ] = s[i ++ ];
        }
        s.erase(s.begin() + k, s.end());//把k后面多余的部分删掉（多余的空格）
        reverse(s.begin(), s.end());//再翻转一次

        return s;
    }
};
```

LeetCode 165. 比较版本号

165. 比较版本号

难度 中等 65

比较两个版本号 *version1* 和 *version2*。
如果 *version1* > *version2* 返回 `1`，如果 *version1* < *version2* 返回 `-1`， 除此之外返回 `0`。

你可以假设版本字符串非空，并且只包含数字和 `.` 字符。

`.` 字符不代表小数点，而是用于分隔数字序列。

例如，`2.5` 不是“两个半”，也不是“差一半到三”，而是第二版中的第五个小版本。

你可以假设版本号的每一级的默认修订版号为 `0`。例如，版本号 `3.4` 的第一级（大版本）和第二级（小版本）修订号分别为 `3` 和 `4`。其第三级和第四级修订号均为 `0`。

示例 1:

```
输入：version1 = "0.1", version2 = "1.1"
输出：-1
```

示例 2:

```
输入：version1 = "1.0.1", version2 = "1"
输出：1
```

示例 3:

```
输入：version1 = "7.5.2.4", version2 = "7.5.3"
输出：-1
```

示例 4:

```
输入：version1 = "1.01", version2 = "1.001"
输出：0
解释：忽略前导零，“01” 和 “001” 表示相同的数字 “1”。
```

```cpp
/*
先比较第一位，第二位.....
按位比较，位数不够补0
atoi(string.c_str())把字符串变成数字
*/
class Solution {
public:
    int compareVersion(string version1, string version2) {
        int i = 0, j = 0;
        while (i < version1.size() || j < version2.size())
        {
            int ki = i, kj = j;
            while (ki < version1.size() && version1[ki] != '.') ki ++ ;
            while (kj < version2.size() && version2[kj] != '.') kj ++ ;
```

```
            string si, sj;
            if (i != ki) si = version1.substr(i, ki - i);
            if (j != kj) sj = version2.substr(j, kj - j);
            if (atoi(si.c_str()) < atoi(sj.c_str())) return -1;
            if (atoi(si.c_str()) > atoi(sj.c_str())) return 1;
            if (ki != i) i = ki + 1;
            if (kj != j) j = kj + 1;
        }
        return 0;
    }
};
*************************************************************************
class Solution {
public:
    int compareVersion(string s1, string s2) {
        int i = 0,j = 0;
        while(i < s1.size() || j < s2.size())
        {
            int x = i,y = j;
            while(x < s1.size() && s1[x] != '.') x ++;
            while(y < s2.size() && s2[y] != '.') y ++;
            int a = i == x ? 0 : atoi(s1.substr(i, x - i).c_str());//x == y >>>>
a = 0
            int b = j == y ? 0 : atoi(s2.substr(j, y - j).c_str());
            if(a > b) return 1;
            if(a < b) return -1;
            i = x + 1;
            j = y + 1;
        }
        return 0;
    }
};
```

LeetCode 929.独特的电子邮件地址

929. 独特的电子邮件地址

难度 简单 106

每封电子邮件都由一个本地名称和一个域名组成，以 @ 符号分隔。

例如，在 `alice@leetcode.com` 中， `alice` 是本地名称，
而 `leetcode.com` 是域名。

除了小写字母，这些电子邮件还可能包含 `'.'` 或 `'+'`。

如果在电子邮件地址的**本地名称**部分中的某些字符之间添加句点（`'.'`），则发往那里的邮件将会转发到本地名称中没有点的同一地址。例如，`"alice.z@leetcode.com"` 和 `"alicez@leetcode.com"` 会转发到同一电子邮件地址。（请注意，此规则不适用于域名。）

如果在**本地名称**中添加加号（`'+'`），则会忽略第一个加号后面的所有内容。这允许过滤某些电子邮件，例如 `m.y+name@email.com` 将转发到 `my@email.com`。（同样，此规则不适用于域名。）

可以同时使用这两个规则。

给定电子邮件列表 `emails`，我们会向列表中的每个地址发送一封电子邮件。实际收到邮件的不同地址有多少？

示例：

```
输入：
["test.email+alex@leetcode.com","test.e.mail+bob.cathy@lee
输出：2
解释：实际收到邮件的是 "testemail@leetcode.com" 和
"testemail@lee.tcode.com"。
```

提示：

- `1 <= emails[i].length <= 100`
- `1 <= emails.length <= 100`
- 每封 `emails[i]` 都包含有且仅有一个 `'@'` 字符。

算法

(哈希表，字符串处理) $O(n)$

遍历 `emails` 中的每个邮箱地址，然后依次进行如下操作：

1. 分别找出用户名和域名；
2. 从前往后遍历用户名的每个字符，如果遇到 `'+'`，则直接截断；如果遇到 `'.'`，则忽略该字符；
3. 将新用户名和域名重新组合，再插入哈希表中；

最终哈希表中不同元素的个数就是答案。

时间复杂度分析：`'emails'` 中的每个邮箱地址只会遍历2遍，且哈希表的插入和查找的时间复杂度均是 $O(L)$，L是字符串长度。所以总时间复杂度是 $O(n)$，n 表示字符串总长度。

```
/*
把email分成用户名和域名，用substr求子串，找到@的位置，然后求子串
把用户名过滤生成新的用户名
再将新用户名和域名合起来组成新的email
用unordered_set存储结果，会自动去重
*/
class Solution {
public:
    int numUniqueEmails(vector<string>& emails) {
        unordered_set<string> S;
        for (auto email : emails)
        {
            string local, domain;
            int k = 0;
            while (email[k] != '@') k ++ ;
            local = email.substr(0, k);
            domain = email.substr(k + 1);
            string newLocal;
            for (auto c : local)
            {
                if (c == '+') break;
                if (c != '.') newLocal += c;
            }
            S.insert(newLocal + '@' + domain);
        }
        return S.size();
    }
};
**********************************************************************
class Solution {
public:
    int numUniqueEmails(vector<string>& emails) {
        unordered_set<string> hash;
        for(auto email : emails)
        {
            int at = email.find('@');
            string name;
            for(auto c : email.substr(0,at))
                if(c == '+') break;
                else if(c != '.') name += c;

            string domain = email.substr(at + 1);
            hash.insert(name + '@' + domain);
        }
        return hash.size();
    }
};
```

LeetCode 5. 最长回文子串

5. 最长回文子串

难度 中等 1839

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1:

```
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。
```

示例 2:

```
输入: "cbbd"
输出: "bb"
```

(暴力枚举) $O(n^2)$

由于字符串长度小于1000，因此我们可以用 $O(n^2)$ 的算法枚举所有可能的情况。
首先枚举回文串的中心 i，然后分两种情况向两边扩展边界，直到遇到不同字符为止:

- 回文串长度是奇数，则依次判断 $s[i-k] == s[i+k], k = 1, 2, 3, \dots$
- 回文串长度是偶数，则依次判断 $s[i-k] == s[i+k-1], k = 1, 2, 3, \dots$

如果遇到不同字符，则我们就找到了以 i 为中心的回文串边界。

时间复杂度分析：一共两重循环，所以时间复杂度是 $O(n^2)$。

```
/*
回文子串可以用马拉车Manacher算法降低到O(n)
枚举回文串中心点，从中心点向两边扩展，找到最大扩展长度，直到不相同为止
字符串长度为奇数是一个字母
字符串长度为偶数是两个个字母
*/
class Solution {
public:
    string longestPalindrome(string s) {
        int res = 0;
        string str;
        for (int i = 0; i < s.size(); i ++ )
        {
            for (int j = 0; i - j >= 0 && i + j < s.size(); j ++ )
                if (s[i - j] == s[i + j])
                {
                    if (j * 2 + 1 > res)
                    {
                        res = j * 2 + 1;
                        str = s.substr(i - j, j * 2 + 1);
                    }
                }
                else break;

            for (int j = i, k = i + 1; j >= 0 && k < s.size(); j -- , k ++ )
                if (s[j] == s[k])
```

```
                {
                    if (k - j + 1 > res)
                    {
                        res = k - j + 1;
                        str = s.substr(j, k - j + 1);
                    }
                }
                else break;
        }
        return str;
    }
};
*****************************************************************************
class Solution {
public:
    string longestPalindrome(string s) {
        string res;
        for(int i = 0;i < s.size();i ++)
        {
            for(int j = i,k = i;j >= 0 && k < s.size() && s[k] == s[j];j --,k
++)
                if(res.size() < k - j + 1)
                    res = s.substr(j,k - j + 1);

            for(int j = i,k = i + 1;j >= 0 && k < s.size() && s[k] == s[j];j -
-,k ++)
                if(res.size() < k - j + 1)
                    res = s.substr(j,k - j + 1);
        }
        return res;
    }
};
```

LeetCode 6. Z 字形变换

6. Z 字形变换

难度 中等 591

将一个给定字符串根据给定的行数，以从上往下、从左到右进行 Z 字形排列。

比如输入字符串为 `"LEETCODEISHIRING"` 行数为 3 时，排列如下：

```
L   C   I   R
E T O E S I I G
E   D   H   N
```

之后，你的输出需要从左往右逐行读取，产生出一个新的字符串，比如：`"LCIRETOESIIGEDHN"`。

请你实现这个将字符串进行指定行数变换的函数：

```
string convert(string s, int numRows);
```

示例 1:

```
输入：s = "LEETCODEISHIRING", numRows = 3
输出："LCIRETOESIIGEDHN"
```

示例 2:

```
输入：s = "LEETCODEISHIRING", numRows = 4
输出："LDREOEIIECIHNTSG"
解释：

L     D     R
E   O E   I I
E C   I H   N
T     S     G
```

算法

(找规律) $O(n)$

这种按某种形状打印字符的题目，一般通过手画小图找规律来做。
我们先画行数是4的情况：

```
0     6       12
1   5 7    11 ..
2 4   8 10
3     9
```

从中我们发现，对于行数是 n 的情况：

1. 对于第一行和最后一行，是公差为 $2(n-1)$ 的等差数列，首项是 0 和 $n-1$；
2. 对于第 i 行 $(0 < i < n-1)$，是两个公差为 $2(n-1)$ 的等差数列交替排列，首项分别是 i 和 $2n-i-2$；

所以我们可以从上到下，依次打印每行的字符。

时间复杂度分析：每个字符遍历一遍，所以时间复杂度是$O(n)$.

```
/*
找规律
第一行首项是0，公差是2(n-1)的等差数列
两个等差数列交错
最后一行，首项是n-1公差是2(n-1)的等差数列
*/
class Solution {
public:
    string convert(string s, int numRows) {
        string res;
        if (numRows == 1) return s;
        for (int j = 0; j < numRows; j ++ )
        {
            if (j == 0 || j == numRows - 1)
            {
                for (int i = j; i < s.size(); i += (numRows-1) * 2)
                    res += s[i];
            }
            else
            {
                for (int k = j, i = numRows * 2 - 1 - j - 1;
                        i < s.size() || k < s.size();
                        i += (numRows - 1) * 2, k += (numRows - 1) * 2)
                {
                    if (k < s.size()) res += s[k];
                    if (i < s.size()) res += s[i];
                }
            }
        }
        return res;
    }
};
*************************************************************************
class Solution {
public:
    string convert(string s, int n) {
        if(n == 1) return s;
        string res;
        for(int i = 0;i < n;i ++)
        {
            if(!i || i == n -1)
            {
                for(int j = i;j < s.size();j += 2 * (n - 1)) res += s[j];
            }
            else
            {
                for(int j = i,k = 2 * (n - 1) - i;j < s.size() || k < s.size();j
+= 2 *(n - 1),k += 2 * (n - 1))
                {
                    if(j < s.size()) res += s[j];
                    if(k < s.size()) res += s[k];
                }
            }
        }
        return res;
    }
};
```

LeetCode 3.无重复字符的最长子串

3. 无重复字符的最长子串

难度 中等 3270

给定一个字符串，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

```
输入：“abcabcbb”
输出：3
解释：因为无重复字符的最长子串是 “abc”，所以其长度为 3。
```

示例 2:

```
输入：“bbbbb”
输出：1
解释：因为无重复字符的最长子串是 “b”，所以其长度为 1。
```

示例 3:

```
输入：“pwwkew”
输出：3
解释：因为无重复字符的最长子串是 “wke”，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，“pwke” 是一个 子序列，不是子串。
```

(双指针扫描) $O(n)$

定义两个指针 $i, j(i <= j)$，表示当前扫描到的子串是 $[i, j]$ (闭区间)。扫描过程中维护一个哈希表 `unordered_map<char,int> hash`，表示 $[i, j]$中每个字符出现的次数。
线性扫描时，每次循环的流程如下：

1. 指针 j 向后移一位, 同时将哈希表中 $s[j]$ 的计数加一: `hash[s[j]]++;`
2. 假设 j 移动前的区间 $[i, j]$ 中没有重复字符，则 j 移动后，只有 $s[j]$ 可能出现2次。因此我们不断向后移动 i，直至区间 $[i, j]$中 $s[j]$ 的个数等于1为止;

复杂度分析：由于 i, j 均最多增加n次，且哈希表的插入和更新操作的复杂度都是 $O(1)$，因此，总时间复杂度 $O(n)$.

```
/*
最长且不包含重复字符的子串
先想暴力O(n^2)
for(int i = 0;i < s.size();i ++)
    for(int j = i;j >= 0;j --)
        if(check() == false)
            break;
优化
单调性O(n)
后面指针往后走，前面指针一定单调往后走
后面走一下，判断一下前面指针是不是要往后走
用hash存储每个字母出现的次数
i是前一个指针，j是后一个指针
```

```
*/
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> hash;
        int res = 0;
        for (int i = 0, j = 0; j < s.size(); j ++ )
        {
            hash[s[j]] ++ ;
            while (hash[s[j]] > 1) hash[s[i ++ ]] -- ;//如果有重复字母，一定是是
s[j]!!!
            res = max(res, j - i + 1);
        }
        return res;
    }
};
```

LeetCode 208.实现 Trie (前缀树)

208. 实现 Trie (前缀树)

难度 中等 215

实现一个 Trie (前缀树)，包含 `insert`，`search`，和 `startsWith` 这三个操作。

示例:

```
Trie trie = new Trie();

trie.insert("apple");
trie.search("apple");   // 返回 true
trie.search("app");     // 返回 false
trie.startsWith("app"); // 返回 true
trie.insert("app");
trie.search("app");     // 返回 true
```

说明:

- 你可以假设所有的输入都是由小写字母 `a-z` 构成的。
- 保证所有输入均为非空字符串。

算法

(Trie树) $O(S)$

1. 首先需要定义树中每个结点的结构，包含26个儿子的指针和一个bool变量代表是否是一个单词的结尾。
2. 插入时沿着根节点向下寻找，如果不存在该路径，则通过创建结点来生成路径。
3. 查询和查询前缀时沿着路径查找即可。

```
/*
Trie树
*/
class Trie {
```

```
public:
    struct Node
    {
        bool is_end;
        Node *son[26];//儿子节点
        Node()
        {
            is_end = false;
            for(int i = 0;i < 26;i ++) son[i] = NULL;
        }
    }*root;
    /** Initialize your data structure here. */
    Trie() {
        root = new Node();
    }

    /** Inserts a word into the trie. */
    void insert(string word) {
        auto p = root;
        for(auto c : word)
        {
            int u = c - 'a';//求编号
            if(p->son[u] == NULL) p->son[u] = new Node();//儿子不存在就创建
            p = p->son[u];
        }
        p->is_end = true;
    }

    /** Returns if the word is in the trie. */
    bool search(string word) {
        auto p = root;
        for(auto c : word)
        {
            int u = c - 'a';
            if(p->son[u] == NULL) return false;
            p = p->son[u];
        }
        return p->is_end;
    }

    /** Returns if there is any word in the trie that starts with the given
prefix. */
    bool startsWith(string prefix) {
        auto p = root;
        for(auto c : prefix)
        {
            int u = c - 'a';
            if(p->son[u] == NULL) return false;
            p = p->son[u];
        }
        return true;
    }
};

/**
 * Your Trie object will be instantiated and called as such:
 * Trie* obj = new Trie();
 * obj->insert(word);
```

```
 * bool param_2 = obj->search(word);
 * bool param_3 = obj->startsWith(prefix);
 */
```

LeetCode 273. 整数转换英文表示

273. 整数转换英文表示

难度 困难 56

将非负整数转换为其对应的英文表示。可以保证给定输入小于 $2^{31} - 1$ 。

示例 1:

```
输入：123
输出："One Hundred Twenty Three"
```

示例 2:

```
输入：12345
输出："Twelve Thousand Three Hundred Forty Five"
```

示例 3:

```
输入：1234567
输出："One Million Two Hundred Thirty Four Thousand
Five Hundred Sixty Seven"
```

示例 4:

```
输入：1234567891
输出："One Billion Two Hundred Thirty Four Million Five
Hundred Sixty Seven Thousand Eight Hundred Ninety One"
```

(模拟) $O(logn)$

为了便于处理，我们将所有单词和数字的映射关系存入哈希表。

然后将原问题分解，我们发现如果可以表示0~999，然后配合 thousand、million、billion即可表示出 10^{12}以内的所有数。即：`xxx billion xxx million xxx thousand xxx`，其中`xxx`是0~999的表示方式。

然后考虑如何表示1000以内的数，分情况讨论：

- 如果大于等于100，则需要先写出`x hundred`，`x`是1~9的英文表示；
- 如果末两位大于20，则需要写出`xx-ty y`，`xx-ty`是20~90的英文表示，`y`是1~9的英文表示；
- 如果末两位不超过20，则直接输出相应的英文单词；

时间复杂度分析：计算量与 n 的十进制表示的位数成正比，所以时间复杂度是 $O(logn)$。

```
class Solution {
public:
    int hundred = 100, thousand = 1000, million = 1000000, billion = 1000000000;
    unordered_map<int, string> numbers;
```

```
    string numberToWords(int num) {
        char number20[][30] = {"Zero", "One", "Two", "Three", "Four", "Five",
                               "Six", "Seven", "Eight", "Nine", "Ten", "Eleven",
                               "Twelve", "Thirteen", "Fourteen", "Fifteen",
                               "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
        char number2[][30] = {"Twenty", "Thirty", "Forty", "Fifty", "Sixty",
                              "Seventy", "Eighty", "Ninety"};
        for (int i = 0; i < 20; i ++ ) numbers[i] = number20[i];
        for (int i = 20, j = 0; i < 100; i += 10, j ++ ) numbers[i] =
number2[j];
        numbers[hundred] = "Hundred", numbers[thousand] = "Thousand";
        numbers[million] = "Million", numbers[billion] = "Billion";
        string res;
        for (int k = 1000000000; k >= 100; k /= 1000)
        {
            if (num >= k)
            {
                res += ' ' + get3(num / k) + ' ' + numbers[k];
                num %= k;
            }
        }
        if (num) res += ' ' + get3(num);
        if (res.empty()) res = ' ' + numbers[0];
        return res.substr(1);
    }

    string get3(int num)
    {
        string res;
        if (num >= hundred)
        {
            res += ' ' + numbers[num / hundred] + ' ' + numbers[hundred];
            num %= hundred;
        }
        if (num)
        {
            if (num < 20) res += ' ' + numbers[num];
            else if (num % 10 == 0) res += ' ' + numbers[num];
            else res += ' ' + numbers[num / 10 * 10] + ' ' + numbers[num % 10];
        }
        return res.substr(1);
    }
};
```